# BÀI TẬP LÝ THUYẾT – NHẬP MÔN TRÍ TUỆ NHÂN TẠO – LẦN 1

**Họ và tên: Nguyễn Thanh Kiên.**

**MSSV: 22110092**

## Nhắc lại lý thuyết: Searching Algorithms

1. Các thuật toán tìm kiếm yêu cầu một cấu trúc dữ liệu để theo dõi cây tìm kiếm đang được xây dựng. Đối với mỗi Node n trong cây, ta có một cấu trúc bao gồm 4 thành phần:
    i. node.STATE: trạng thái trong không gian trạng thái mà node tương ứng.
    ii. node.PARENTS: Node trong cây mà nó tạo ra Node khác.
    iii. node.ACTION: hành động mà được áp dụng vào node cha để tạo ra node.
    iv. node.PATH-COST: chi phí của đường đi từ trạng thái ban đầu đến các node, kí hiệu g(n).
2. Cần một nơi để lưu trữ các Node, đường biên (frontier) cần được lưu trữ sao cho thuật toán tìm kiếm có thể dễ dàng chọn Node tiếp theo để mở rộng theo cách mà nó ưu tiên. Cấu trúc dữ liệu phù hợp là hàng đợi (queue). Các thao tác trên Queue bao gồm:
    a. EMTY?(queue): Trả về giá trị True chỉ khi không còn giá trị nào trong Queue.
    b. POP(queue): loại bỏ phần tử đầu tiên của Queue và trả về nó.
    c. INSERT(element, queue): chèn một phần tử vào và trả về hàng đợi sau khi chèn.
3. Đo lường hiệu suất giải quyết vấn đề (Measuring problem-solving performance): là xem xét các tiêu chí có thể để lựa chọn thuật toán phù hợp. Có thể đánh giá hiệu suất thuật toán theo 4 cách sau:
    i. Complete (tính hoàn thiện): Nếu tồn tại giải pháp, thuật toán có đảm bảo tìm được nó hay không?
    ii. Optimality (tính tối ưu): Thuật toán có tìm được giải pháp tối ưu hay không? (Giải pháp tối ưu là giải pháp có chi phí đường đi thấp nhất trong số các giải pháp)
    iii. Time complexity (Độ phức tạp về thời gian): Mất bao lâu để tìm được một giải pháp?

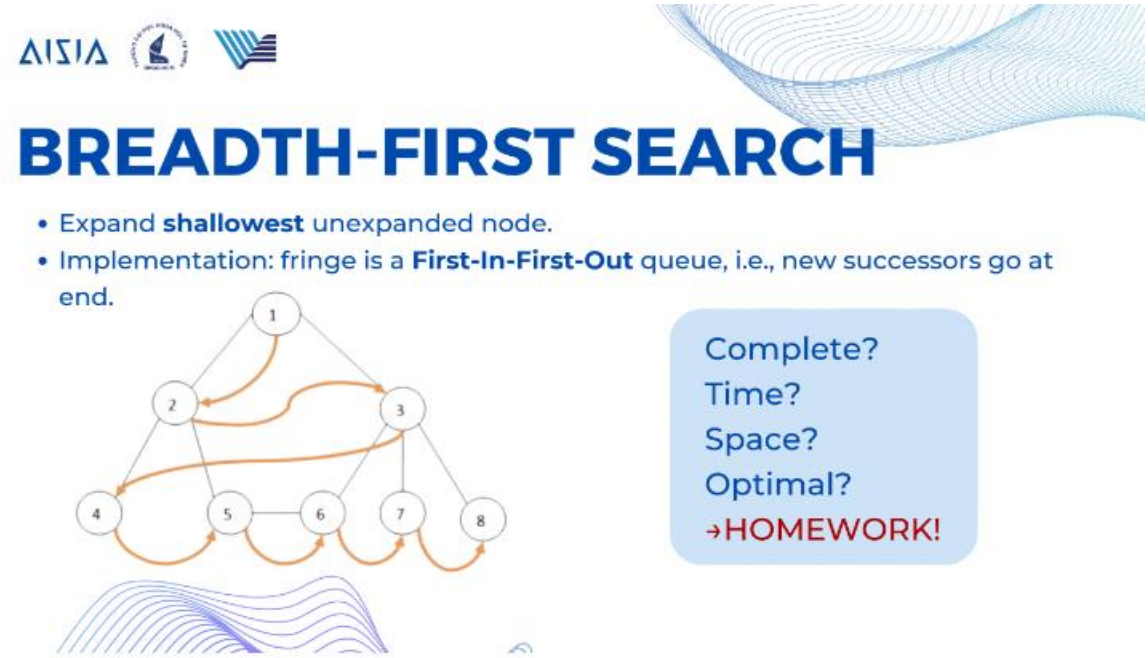
iv. Space complexity (Độ phức tạp về không gian): Cần bao nhiêu bộ nhớ để thực hiện quá trình tìm kiếm?

4. Độ phức tạp về không gian và thời gian luôn được xem là thước đo về độ khó của vấn đề. Trong lý thuyết Khoa học Máy tính, thước đo thông thường được định nghĩa là kích thước của không gian trạng thái: |V| + |E|, trong đó, |V| là tập hợp các Node, và |E| là tập hợp các Edges – links (cạnh). Trong trí tuệ nhân tạo, Đồ thị được ngầm định biểu diễn bằng State, Actions, Model, .. và tất cả thường là vô hạn (không đếm được). Vì vậy, độ phức tạp được biểu diễn bằng 3 đại lượng:
    i. b: hệ số phân nhánh (branching factor): số lượng kế thừa tối đa của bất kỳ Node nào.
    ii. d: độ sâu (depth) của Node mục tiêu nông nhất, số bước dọc theo đường đi từ gốc.
    iii. m: độ dài tối đa (maximum length) của bất kỳ đường đi nào trong không gian.

    . Thời gian thường được đo lường theo số lượng Node tạo ra trong quá trình tìm kiếm, và không gian được đo lường bằng số lượng tối đa Node được lưu trữ trong bộ nhớ.

5. Heuristic Function: hàm đánh giá có thể chấp nhận được, được coi là một ước lượng chi phí, vì Node có đánh giá thấp nhất sẽ được mở rộng trước tiên. Cách triển khai tìm kiếm đồ thị tốt nhất giống với cách triển khai UCS,, ngoại trừ việc sử dụng *f* thay vì *g* để sắp xếp hàng đợi ưu tiên. Hầu hết các thuật toán tìm kiếm tốt nhất bao gồm một thành phần của f là hàm heuristic, kí hiệu là h(n) = ước lượng chi phí của con đường rẻ nhất từ trạng thái tại nút n đến trạng thái mục tiêu.

**Bài tập: Uniformed Search Strategies.**




. Expland shallowest unexpanded node: Cần mở rộng Node mà nó là Node chưa được mở rộng và nông nhất.

. Implementation: fringe is a First – In – First – Out queue, i.e., new successors go at end.: Triển khai: Vùng biên là một Queue (hàng đợi) vào trước ra trước (FIFO), tức là các Node kế tiếp mới sẽ được thêm vào cuối hàng.

**Thuật toán tìm kiếm theo chiều rộng (BFS)** là một trường hợp của thuật toán tìm kiếm đồ thị tổng quát, trong đó Node chưa được mở rộng nông nhất được chọn để mở rộng. Bằng cách sử dụng hàng đợi FIFO, các Node mới (luôn ở độ sâu sâu hơn so với Node Parents của chúng) được thêm vào phía sau Queue, và các Node cũ, nông hơn các Node mới, sẽ được mở rộng trước.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

**Figure 3.11** Breadth-first search on a graph.

**4 Tiêu chí:**

**Complete**: Dễ dàng nhận thấy rằng BFS có tính hoàn thiện. Nếu Node mục tiêu nông nhất ở một độ sâu hữu hạn d, BFS sẽ tìm thấy Node đó sau khi tạo ra tất cả các Node nông hơn (với điều kiện hệ số phân nhánh – b là hữu hạn). Ngay khi một Node mục tiêu được tạo ra, nó luôn là Node mục tiêu nông nhất vì tất cả các Node nông hơn đã được tạo ra trước đó và không đáp ứng được tiêu chí mục tiêu (Goal tests).

**Optimal:** BFS có tính tối ưu nếu chi phí đường đi là một hàm không giảm theo độ sâu của Node. Tối ưu ở đây là tạo ra đường dẫn tối ưu, không phải là đường dẫn nhanh nhất có thể, vì vậy, BFS chỉ tối ưu khi các đường đi không có trọng số, hay các Actions đều có cùng 1 chi phí (cost).

**Time:** Node gốc của cây tìm kiếm tạo ra b Node ở độ sâu đầu tiên ($d_1$), mỗi Node trong đó tạo ra b Node nữa, tổng cộng là $b^2$ ở độ sâu thứ hai ($d_2$), mỗi Node này lại tạo ra thêm b Node nữa, dẫn đến $b^3$ ở độ sâu thứ ba ($d_3$), vòng lặp cứ tiếp tục như vậy. Trường hợp xấu nhất, khi tới độ sâu cuối cùng (vô hạn $d_n$, n -> ∞, $d_n$ -> d), tổng số Node được tạo ra là :

$$b + b^2 + b^3 + .. + b^d = O(b^d).$$

Nếu thuật toán áp dụng kiểm tra mục tiêu cho các Node khi nó được chọn để mở rộng thay vì được tạo ra, thì toàn bộ các layer (lớp) các các Node ở độ sâu d sẽ được mở rộng trước khi phát hiện mục tiêu và độ phức tạp Thời gian sẽ là $O(b^{d+1})$.

**Space:** Ở tìm kiếm đồ thị (Graph Search), tập Explored sẽ lưu trữ mọi Node đã được mở rộng, và độ phức tạp không gian luôn nằm trong một hệ số b của độ phức tạp thời gian. Đối với BFS, mỗi Node được tạo ra luôn được lưu trữ trong bộ nhớ, vì vậy sẽ có $O(b^{d-1})$ node trong Explored Set và $O(b^d)$ node trong đường biên, vì vậy độ phức tạp không gian của BFS là $O(b^d)$. Không gian bộ nhớ là một vấn đề lớn của BFS so với thời gian thực thi.

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 milliseconds | 107 kilobytes |
| 4 | 11,110 | 11 milliseconds | 10.6 megabytes |
| 6 | $10^6$ | 1.1 seconds | 1 gigabyte |
| 8 | $10^8$ | 2 minutes | 103 gigabytes |
| 10 | $10^{10}$ | 3 hours | 10 terabytes |
| 12 | $10^{12}$ | 13 days | 1 petabyte |
| 14 | $10^{14}$ | 3.5 years | 99 petabytes |
| 16 | $10^{16}$ | 350 years | 10 exabytes |

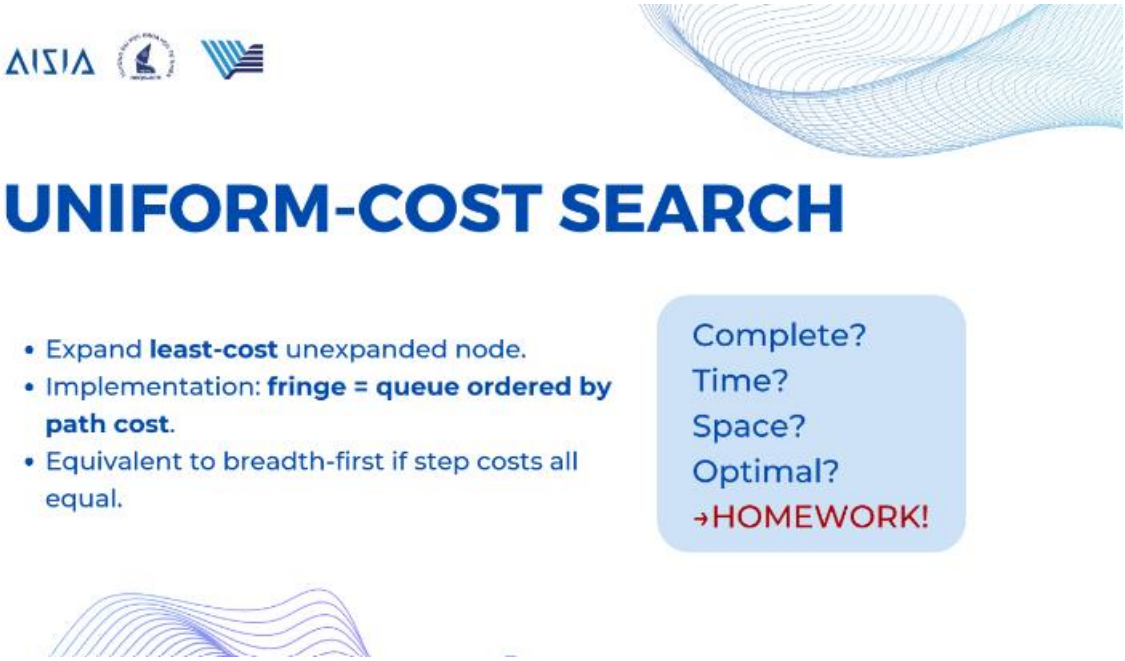
Ở độ sâu 12, chúng ta cần 13 ngày để giải quyết vấn đề. Điều đó hoàn toàn khả thi, nhưng sẽ không có một máy tính nào có 1 petabyte bộ nhớ. Vì vậy, các bài toán tìm kiếm có độ phức tạp là một hàm mũ không thể được giải quyết bằng các phương pháp thông tin với bất kỳ trường hợp nào ngoại trừ trường hợp nhỏ nhất.

Đoạn code Python cho BFS:

Hoc_1.py

```python
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = []
queue = []

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

```
Following is the Breadth-First Search
5 3 7 2 4 8 Press any key to continue . . .
```




. Expand least – cost unexpanded node: Mở rộng Node chưa mở rộng có chi phí thấp nhất.

. Implementation: fringe = queue ordered by path cost: Triển khai: đường biên là một hàng đợi được sắp xếp theo chi phí đường đi.

. Equivalent to breadth – first if step costs all equal: Tương đường với BFS nếu tất cả các chi phí (step costs) bằng nhau.

**Thuật toán tìm kiếm có chi phí (UCS)** là thuật toán mở rộng Node n với chi phí đường đi thấp nhất g(n) thay vì mở rộng Node nông nhất giống như Thuật toán tìm kiếm theo chiều rộng (BFS). Nó thực hiện bằng cách lưu trữ đường biên (fringe) như một hàng đợi (Queue) ưu tiên được sắp xếp theo g. Khác biệt của UCS so với BFS là kiểm tra mục tiêu được áp dụng cho một Node khi nó được chọn để mở rộng thay vì nó được tạo ra lần đầu (lý do là Node mục tiêu được tạo ra có thể nằm trên một đường đi không tối ưu), và sẽ có một Test được thêm vào nếu xảy ra trường hợp tìm thấy một đường đi tối ưu hơn đến một Node hiện có trên đường biên.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

**4 Tiêu chí:**

**Complete**: UCS không quan tâm đến số bước của một đường đi, mà chỉ quan tâm đến tổng chi phí của chúng. Vì vậy, với một chuỗi hành động có chi phí bằng không vô hạn (zero – cost actions, ví dụ: NoOp actions), UCS sẽ mắc kẹt trong một vòng lặp vô hạn.

. Giả sử tất cả các chi phí đều lớn hơn 0 (đơn vị tuyến tính), nhưng lại bé hơn một số dương nhỏ nào đó, ta gọi số này là $\epsilon$.

. Bài toán giả sử rằng chúng ta có một đường đi vô hạn với chi phí đầu tiên là $\frac{1}{2}$, tiếp theo là $\frac{1}{4}$, sau đó là $\frac{1}{8}$, cuối cùng là $\frac{1}{2^n}$ với n tiến ra vô cùng. Sử dụng Tổng của cấp số nhân vô hạn, khi n tiến ra vô cùng, chi phí của đường đi bằng 1 ngay cả khi có vô số Node STATES trên đường đi đó. Vì vậy với chi phí lớn hơn 1, ví dụ, chi phí là 2, thì UCS sẽ không thể tìm được bất kỳ giải pháp nào cho bài toán đường đi này và vì vậy nó sẽ không phải là một thuật toán có tính hoàn thiện. Vì vậy UCS có tính hoàn thiện được đảm bảo miễn là chi phí của mỗi bước là không âm và vượt quá một hằng số dương $\epsilon$ nhỏ nào đó, để chính xác hơn thì phải tồn tại một số $\epsilon$ cố định sao cho trong mỗi phạm vi kích thước $\epsilon$ có một số lượng hữu hạn các Node STATES.

**Optimal:** Mỗi khi UCS tìm kiếm đều chọn một Node n để mở rộng, đường đi tối ưu đến Node đó luôn được tìm thấy (nếu không phải vậy, sẽ có một Node khác trên đường biên không nằm trên đường đi tối ưu từ Node bắt đầu đến Node n, theo định nghĩa, thì Node đó sẽ có chi phí g thấp hơn n và sẽ được chọn trước). Sau đó, vì chi phí luôn lớn hơn 0, nên đường đi không bao giờ trở nên ngắn hơn khi các Node được thêm vào. Những lý do trên kết hợp lại cho ta thấy rằng UCS đều mở rộng các Node theo thứ tự

chi phí đường đi tối ưu của chúng, Node mục tiêu đầu tiên được chọn phải là giải pháp tối ưu, đều này cho thấy rằng UCS có tính tối ưu trong tổng thể.

**Time:** Bởi vì UCS là thuật toán mở rộng Node n theo chi phí đường đi, nên độ phức tạp của nó không dễ dàng biểu diễn qua 2 toán tử b và d. Giả sử C* là chi phí của của giải pháp tối ưu mà UCS đã tìm được, và $\epsilon$ là chi phí nhỏ nhất của mỗi hành động phải trả.

Nếu hệ số phân nhánh là b, mỗi lần ta mở rộng ra một Node, ta sẽ tiếp tục gặp thêm k Node nữa. Do đó sẽ có 1 Node ở cấp 0, b Node ở cấp 1, $b^2$ Node ở cấp 2,… $b^k$ Node ở cấp k. Giả sử thuật toán dừng lại khi đạt cấp k, khi đó, sử dụng chuỗi hình học, tổng số Node đã duyệt qua là:

$$1 + b + b^2 + \dots + b^k = \frac{b^{k+1}}{b-1} = O(b^k)$$

Vì vậy, nếu Node mục tiêu nằm layer k, thì ta cần mở rộng $O(b^k)$ tổng số Node để tìm Node mong muốn. Vì C* là chi phí cuối cùng và mỗi bước cho ta đến gần Node mục tiêu hơn một khoảng ít nhất là $\epsilon$, cộng với việc chúng ta bắt đầu tại cấp 0 và kết thúc ở $C^*/\epsilon$ nên khoảng cách mỗi bước sẽ là:

$$0, \epsilon, 2\epsilon, 3\epsilon, \dots (C^*/\epsilon)\epsilon$$

Vì vậy số bước cần thực hiện là $\frac{C^*}{\epsilon} + 1$. Vì vậy có $\frac{C^*}{\epsilon} + 1$ lớp, cho nên tổng số trạng thái chúng ta cần mở rộng hay độ phức tạp về thời gian trong trường hợp xấu nhất là $O\left(b^{1+\frac{C^*}{\epsilon}}\right)$, mà nó có thể sẽ lớn hơn rất nhiều so với $b^d$. Lý do vì UCS có thể khám phá các cây lớn với các bước nhỏ trước khi khám phá các đường đi liên quan đến các bước lớn hơn mà hữu ích. Khi tất cả chi phí đều bằng nhau, thì là $O\left(b^{1+\frac{C^*}{\epsilon}}\right) = O(b^d)$, nhưng UCS thực hiện nhiều công việc hơn vì mở rộng các Node ở độ sâu d một cách không cần thiết.

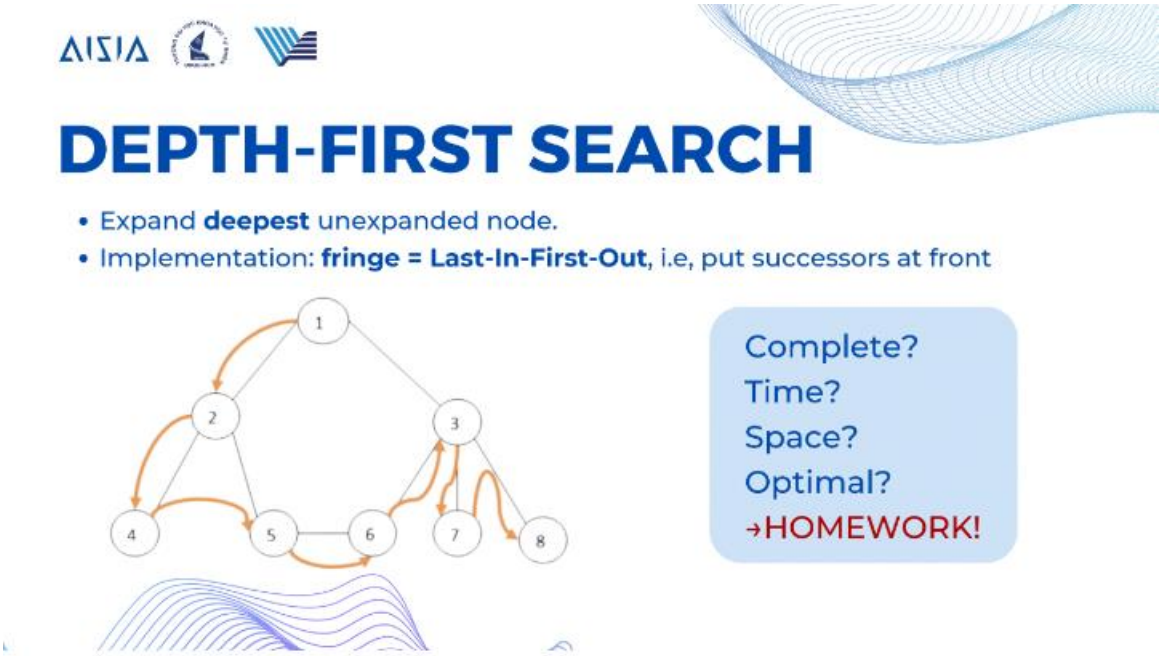
**Space:** UCS phải lưu trữ tất cả các Node trong Queue ưu tiên (priority queue), điều này tốn rất nhiều không gian, đặc biệt khi C* lớn và $\epsilon$ rất nhỏ. Vì vậy, UCS phải lưu trữ các Node đã mở rộng cho tới khi tìm được giải pháp. Vậy nên, độ phức tạp về không gian của nó cũng giống như độ phức tạp thời gian trong trường hợp tệ nhất là $O\left(b^{1+\frac{C^*}{\epsilon}}\right)$. Cũng giống như thời gian, khi tất cả chi phí bằng nhau thì $O\left(b^{1+\frac{C^*}{\epsilon}}\right) = O(b^d)$.

Đoạn code Python cho UCS:

```python
import heapq

graph = {
    '5': [('3', 2), ('7', 1)],
    '3': [('2', 4), ('4', 3)],
    '7': [('8', 5)],
    '2': [],
    '4': [('8', 2)],
    '8': []
}

def ucs(graph, start, goal):

    priority_queue = []
    heapq.heappush(priority_queue, (0, start))

    visited = set()
    costs = {start: 0}

    while priority_queue:

        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node in visited:
            continue

        print(f"Visiting {current_node} with cost {current_cost}")
```

```python
        if current_node == goal:
            print(f"Reached goal {goal} with cost {current_cost}")
            return current_cost

        visited.add(current_node)


        for neighbor, edge_cost in graph[current_node]:
            new_cost = current_cost + edge_cost


            if neighbor not in visited or new_cost < costs.get(neighbor, float('inf')):
                costs[neighbor] = new_cost
                heapq.heappush(priority_queue, (new_cost, neighbor))

    return float('inf')


start_node = '5'
goal_node = '8'
print(f"Uniform Cost Search from {start_node} to {goal_node}:")
result = ucs(graph, start_node, goal_node)
print(f"Minimum cost to reach {goal_node} is {result}")
```

```
C:\Program Files (x86)\Micros
Uniform Cost Search from 5 to 8:
Visiting 5 with cost 0
Visiting 7 with cost 1
Visiting 3 with cost 2
Visiting 4 with cost 5
Visiting 2 with cost 6
Visiting 8 with cost 6
Reached goal 8 with cost 6
Minimum cost to reach 8 is 6
Press any key to continue . . .
```




. Expand deepest unexpanded node: Mở rộng Node sâu nhất mà nó chưa mở rộng.

. Implementation: fringe = Last – In – First – Out, i.e, put successors at front: Triển khai: Hàng đợi Vào sau Ra trước, đặt các Node kế thừa ở phía trước hàng đợi.

**Thuật toán tìm kiếm theo chiều sâu (DFS)** là mở rộng Node sâu nhất trong hàng đợi hiện tại của cây tìm kiếm. Quá trình này tiến hành ngay lập tức đến mức sâu nhất của cây, nơi các Node không còn kế thừa được nữa. Khi các Node này được mở rộng và không còn Node kế thừa nào tiếp tục, nó sẽ bị loại khỏi Queue và quá trình sẽ lùi lại đến Node sâu tiếp theo mà Node đó còn Node kế thừa chưa được khám phá. Điều này có nghĩa là Node được tạo gần đây nhất sẽ được chọn để mở rộng, và Node này sẽ là Node sâu nhất chưa được mở rộng, vì nó sâu hơn so với Node cha của nó, và Node cha đó cũng từng là Node sâu nhất chưa được mở rộng khi được chọn.

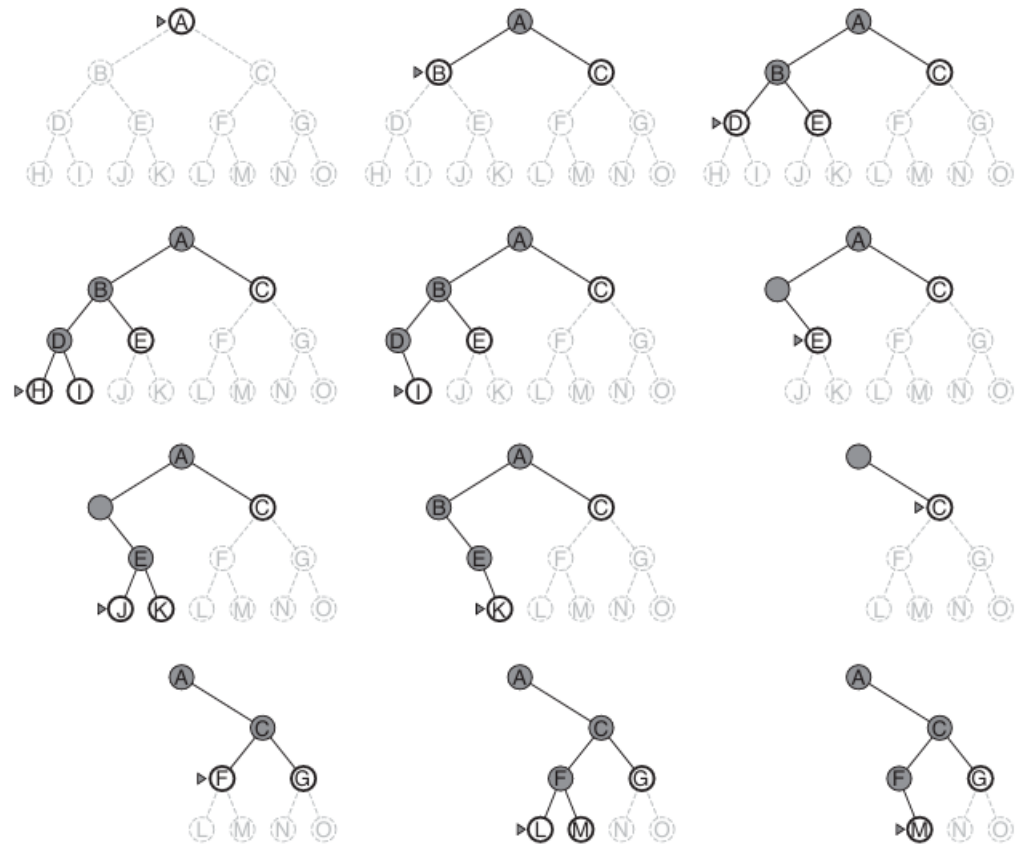


**Figure 3.16** Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and $M$ is the only goal node.

**4 Tiêu chí:**

**Complete**: Các thuộc tính của DFS phụ thuộc rất nhiều vào việc sử dụng Graph – Search hay Tree – Search:

. Ở Đồ thị tìm kiếm, sau khi đã khám phá một Node, nó sẽ theo dõi và tự động lưu trữ lại, hay nó sẽ tránh lặp lại các trạng thái và các đường dẫn dư thừa, tránh được các vòng lặp vô hạn, cuối cùng nó sẽ mở rộng mọi Node, hoàn chỉnh trong không gian trạng thái hữu hạn. Các Redundant paths – đường đi dự phòng, là các đường đi khác nhau từ cùng một Node bắt đầu đến Node cuối cùng của đồ thị. Graph Search vẫn sẽ khám phá tất cả các đường đi này, nhưng đến khi đến một Node mà nó đã truy cập trước đó, nó sẽ không đi xa hơn nữa mà sẽ sao lưu và tìm kiếm thêm các đường dẫn mà nó chưa thử. Vì vậy, DFS có tính hoàn thiện trong Graph – Search.

. Ở Cây tìm kiếm, mọi thứ lại hoàn toàn khác. Vì tìm kiếm cây không lưu trữ các Node đã khám phá, nên ta không biết được rằng liệu Node hiện tại có được khám phá hay không, nên DFS sẽ có thể khám phá lại Node đó và sẽ lặp lại mãi mãi, tạo một vòng lặp vô hạn. Vì Tree – Search chỉ theo dõi đường đi hiện tại nên nó có thể chạy trên cùng một đường đi nhiều lần trong cùng một tìm kiếm. Giả sử Node gốc có 2 nhánh, mỗi nhánh đó lại dẫn đến cùng một Node, và có một đường đi dài dẫn ra Node đó. Việc tìm kiếm Cây sẽ đi theo con đường dài đó hai lần, mỗi lần cho ra mỗi nhánh trong số 2 nhánh dẫn đến nó. Vì vậy, DFS không có tính hoàn thiện trong Tree – Search.

**Optimal:** DFS không phải là một thuật toán tối ưu, vì nó không đảm bảo tìm được đường đi tốt nhất hoặc không tìm được đường đi (trong trường hợp Tree – Search vô hạn). Vì DFS mở rộng theo chiều sâu, đi càng sâu càng tốt trước khi quay lại thăm các nhánh khác, nên trong quá trình này, như đã nói ở phần Complete, nó có thể tìm thấy được đường đi không tối ưu vì nó không kiểm tra tất cả các đường đi trước khi quyết định kết thúc. DFS có thể tìm một đường đi rất sâu trong Tree – Search trước khi tìm thấy các đường đi ở mức nông hơn, và những đường đi này không có chi phí tối ưu hay nhỏ nhất.

**Time:** Độ phức tạp thời gian của DFS bị giới hạn bởi kích thước của không gian trạng thái (có thể vô hạn). Nếu ta có thể truy cập từng Node trong thời gian O(1), thì với hệ số phân nhánh b và độ sâu tối đa là m, tổng số Node trong trường hợp này sẽ là trường hợp xấu nhất, cộng với việc sử dụng chuỗi hình học, ta được:

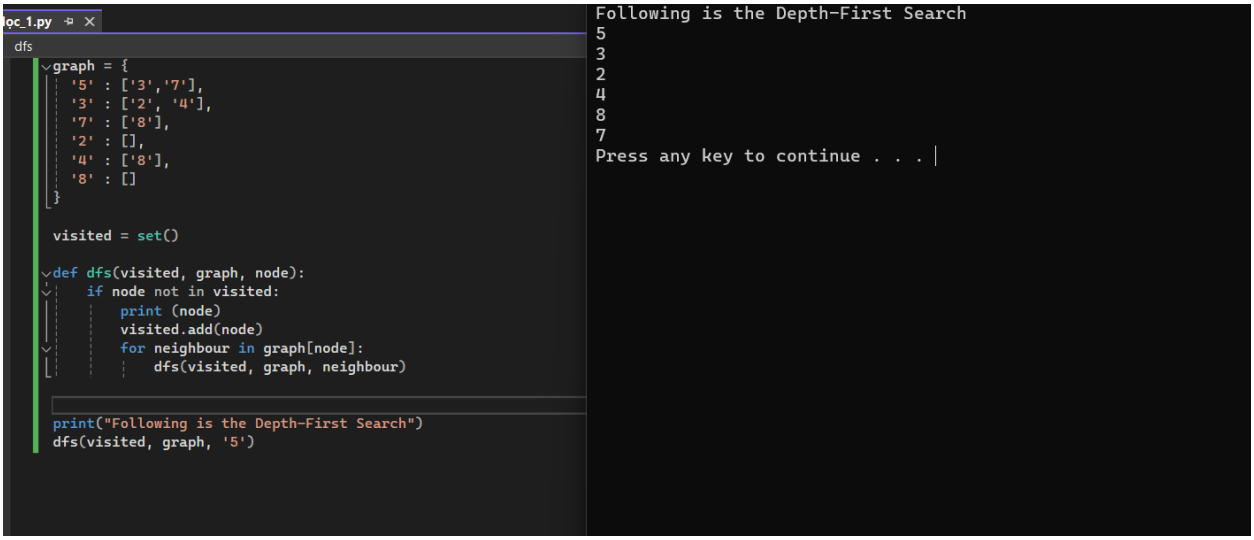
$$1 + b + b^2 + .. + b^{m-1} = \frac{b^{m-1}}{b-1} = o(b^m)$$

Do đó độ phức tạp thời gian sẽ là $o(b^m)$, có thể lớn hơn rất nhiều so với kích thước của không gian trạng thái, và m có thể lớn hơn d rất nhiều, và nếu cây không có giới hạn thì m là vô hạn

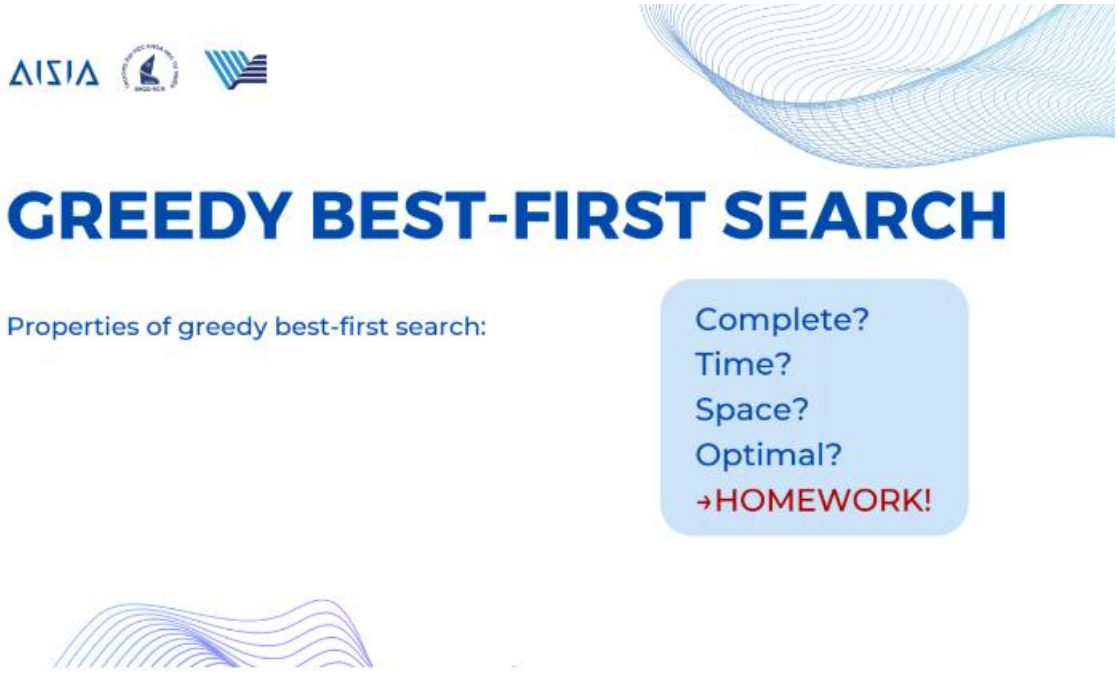
**Space:** Đối với mỗi Node, ta phải lưu trữ các Node anh em của nó để khi truy cập tất cả các Node con và quay lại Node cha, ta có thể biết rằng Node anh em nào sẽ được khám phá tiếp theo. Đối với m Node trong đường đi, ta phải lưu trữ thêm b Node cho mỗi Node trong số m Node. Vì vậy với DFS, ta cần lưu trữ khoảng O(bm) Node. Vì vậy, độ phức tạp không gian của DFS là O(bm). Vì O(bm) < O(d) nên độ phức tạp không gian

của DFS là đặc điểm nổi trội của nó, là khác biệt thuận lợi để ta cân nhắc sử dụng DFS hay BFS.

Đoạn code Python cho DFS:



**Bài tập: Informed Search Strategies.**




**Thuật toán tìm kiếm tham lam (Greedy Best – First Search)** là thuật toán cố gắng mở rộng Node gần nhất với mục tiêu và tìm giải pháp một cách nhanh chóng bằng cách đánh giá các Node bằng hàm heuristic, tức là $f(n) = h(n)$. Thuật toán có cái tên ngộ nghĩnh bởi vì tại mỗi bước, nó cố gắng tiến gần mục tiêu nhanh nhất có thể.

**4 Tiêu chí:**

**Complete**: Giống như DFS, Greedy Best – First Search không có tính hoàn thiện. Vì khi GBFS không đạt được Node mục tiêu, nó sẽ mở rộng các Node và rơi vào vòng lặp vô hạn. Nếu một Node được chọn để mở rộng là một điểm không có Node kế thừa, thuật toán sẽ không thể tiến xa hơn, nhưng nếu có một Node khác cũng gần với Node mục tiêu hơn nhưng không nằm trên đường đi đúng, thuật toán quay lại Node ban đầu và tiếp tục mở rộng lại.

**Optimal:** Vì GBFS chỉ dựa vào hàm $h(n)$ để đưa ra quyết định, mà không xem xét chi phí từ Node gốc đến Node hiện tại $g(n)$, nên nó sẽ dẫn đến việc chọn các Node không tối ưu, không đảm bảo rằng thuật toán sẽ khám phá các khả năng, đường đi có thể đến khi đạt được giải pháp. Vì vậy, GBFS không có tính tối ưu.

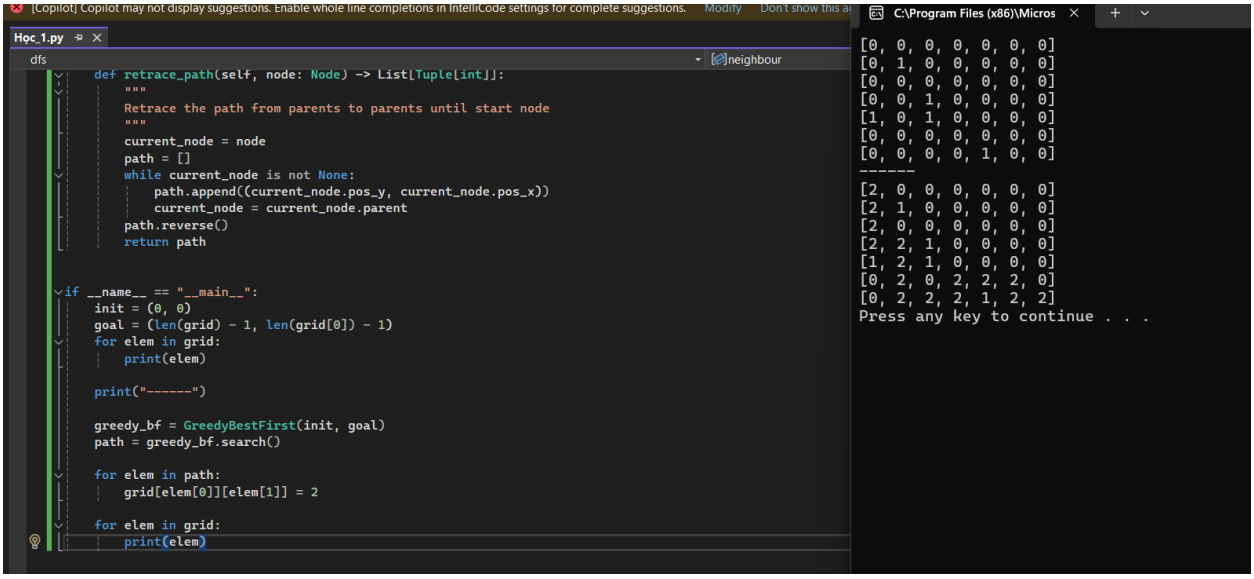
**Time:** Trong trường hợp xấu nhất, GFBS phải mở rộng mọi Node cho đến độ sâu m (là độ sâu tối đa của cây tìm kiếm), nếu mỗi Node sinh ra b Node con, thì giống như BFS, số lượng Node mà thuật toán phải tìm kiếm sẽ tăng theo cấp số nhân với mỗi độ sâu. Tức là số lượng Node tổng cộng có thể tìm kiếm sẽ là $b^m$. Vì vậy, độ phức tạp thời gian của GFBS là $o(b^m)$, trong trường hợp hàm Heuristic tốt, độ phức tạp này có thể giảm đáng kể. Mức độ giảm phụ thuộc vào bài toán và chất lượng của hàm *h(n).*

**Space:** Với tìm kiếm theo chiều sâu (DFS), ta phải quay lại các Node là Node con chưa được mở rộng của Parents (hoặc Node Parents khi Parents của bạn không còn con nào không được mở rộng (và cứ tiếp tục như vậy khi đi lên cây)). Vì vậy, độ phức tạp về không gian bị giới hạn bởi tổ tiên của Node và con cháu của những tổ tiên này.

Với GBFS, ta có thể chuyển đến bất kỳ Node nào được đánh giá nhưng chưa được mở rộng, ta hoàn toàn đi xuống các đường đi trước đó. Vì vậy, thuật toán khi quay lui có thể thực hiện các bước nhảy ngẫu nhiên khắp Cây, khiến nhiều Node chị em không được mở rộng.

Tuy nhiên, ta phải nhớ giá trị của hàm đánh giá cho tất cả các Node này vì có thể chúng sẽ là Node tiếp theo khi quá trình quay lui này xảy ra. Vì vậy, trong trường hợp xấu nhất, gần như toàn bộ cây cần được khi nhớ. Hay độ phức tạp không gian là $O(b^m)$.

Code python:



AISIA

# ADMISSIBLE HEURISTICS

- Theorem: if h(n) is admissible, A* using Tree-Search is optimal.
- PROOF: (HOMEWORK 2)
  - Suppose a suboptimal goal node G2 appears on the fringe and let the cost of the optimal solution be C*. Prove that f(G2)>C*.
  - Consider a fringe node n that is on an optimal solution path. Prove that f(n) < C*.
  - So, G2 will not be expanded and A* must return an optimal solution.
- If we use the Graph-Search instead of Tree-Search, the proof may fail.
- HOMEWORK 3: EXERCISE 4.4 (BOOK)

Theorem: if h(n) is admissible, A* using Tree – Search is optimal.

Chứng minh rằng nếu hàm Heuristic h(n) có thể chấp nhận được, thuật toán A* cho Cây tìm kiếm có tính tối ưu.

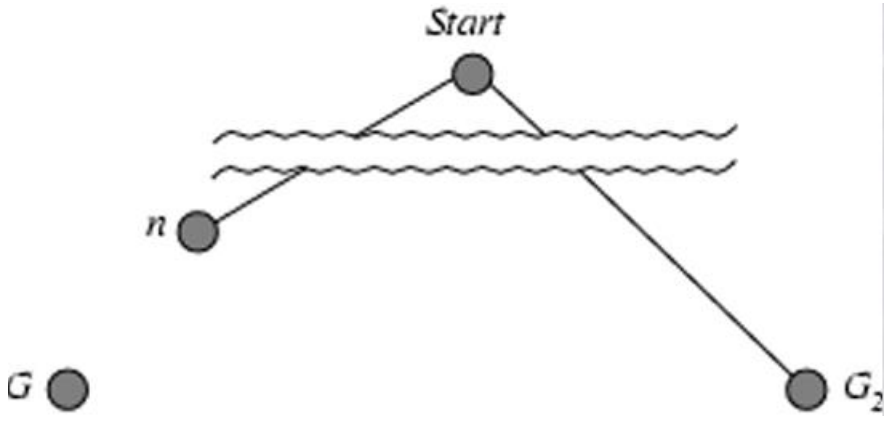
Ta định nghĩa lại một vài khái niệm.

C* : chi phí giải pháp tối ưu, là tổng chi phí thấp nhất từ Node gốc đến Node mục tiêu.

*f(n)*: Tổng chi phí mà thuật toán sử dụng để đánh giá Node n, $f(n) = g(n) + h(n)$.

*g(n)*: Chi phí từ Node gốc đến Node n. Là tổng chi phí cần thiết để đi từ điểm bắt đầu đến điểm hiện tại n.

*h(n)*: hàm ước lượng chi phí từ Node n đến Node mục tiêu.



Giả sử một Node mục tiêu không tối ưu $G_2$ xuất hiện trên đường biên và chi phí của giải pháp tối ưu là C*.

Vì $G_2$ là một Node mục tiêu không tối ưu, có nghĩa là không có cách nào để đi từ Node gốc đến Node mục tiêu qua $G_2$ mà có chi phí nhỏ hơn hoặc bằng C*. Ta tính được chi phí từ Node gốc đến Node Mục tiêu thông qua Node $G_2$ bằng tổng chi phí từ Node gốc đến Node $G_2$ cộng với chi phí ước lượng từ $G_2$ đến Node mục tiêu.

Tức là: $f(G_2) = g(G_2) + h(G_2)$

Vì $G_2$ không phải là Node tối ưu, tức không có giải pháp nào tốt hơn đi qua Node $G_2$ để đến Node mục tiêu. Do đó: $f(G_2) = g(G_2) + h(G_2) > C^*$ (1)

Xét một Node n trên đường biên và nằm trên đường đi giải pháp tối ưu.

Vì n là Node nằm trên đường đi giải pháp tối ưu, nên tổng của Chi phí từ Node gốc đến Node n và chi phí thực tế từ Node n đến Node mục tiêu sẽ bé hơn hoặc bằng chi phí tối ưu từ Node gốc đến Node mục tiêu, hay: $g(n) + h^*(n) \le C^*$

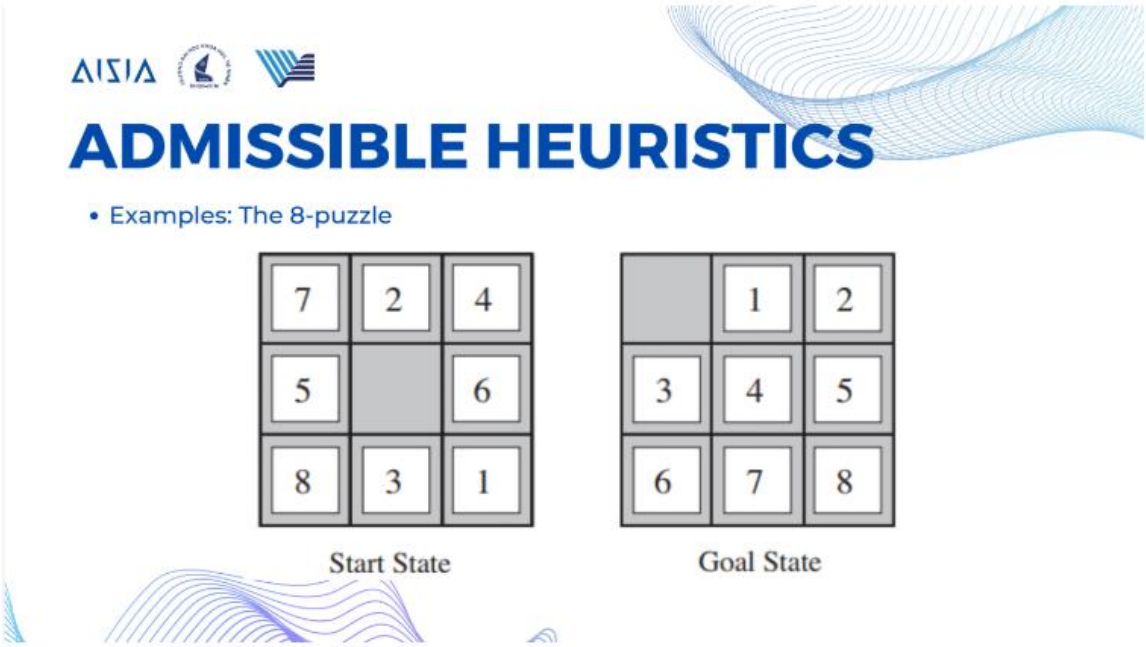
Vì h(n) là admissible, nên: $h(n) \le h^*(n)$

Nên: $f(n) = g(n) + h(n) \le g(n) + h^*(n) \le C^*$ (2)

Từ (1) và (2) ta được $f(n) = g(n) + h(n) \le g(n) + h^*(n) \le C^* < f(G_2)$

Vì vậy, $G_2$ sẽ không được mở rộng bởi A* và A* sẽ tìm ra giải pháp tối ưu.

**Câu 6:**




Xác định khoảng cách Manhattan h(n) và sử dụng thuật toán tree-search để giải ví dụ trên?

Khoảng cách Manhattan là tổng khoảng cách ngắn nhất giữa hai ô theo chiều ngang và chiều dọc. Nó không tính khoảng cách chéo, chỉ xem xét các bước di chuyển lên, xuống, trái và phải. Do đó, khoảng cách Manhattan giữa hai ô sẽ là số bước cần thiết (ngắn nhất) để di chuyển từ ô này sang ô khác mà không thể di chuyển chéo.

Vậy số bước cần thiết của ô 1 để về đúng vị trí là: 3 (lên, lên, trái).

Vậy số bước cần thiết của ô 2 để về đúng vị trí là: 1 (trái).

Vậy số bước cần thiết của ô 3 để về đúng vị trí là: 2 (lên, trái).

Vậy số bước cần thiết của ô 4 để về đúng vị trí là: 2 (trái, xuống).

Vậy số bước cần thiết của ô 5 để về đúng vị trí là: 2 (phải, phải)

Vậy số bước cần thiết của ô 6 để về đúng vị trí là: 3 (trái, trái, xuống).

Vậy số bước cần thiết của ô 7 để về đúng vị trí là: 3 (phải, xuống, xuống).

Vậy số bước cần thiết của ô 8 để về đúng vị trí là: 2 (phải, phải).

Vậy khoảng cách Manhattan là: 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.

Sử dụng Tree – Search để giải Puzzle

```
None
[7, 2, 4]
[5, 0, 6]
[8, 3, 1]

D
[7, 2, 4]
[5, 3, 6]
[8, 0, 1]

R
[7, 2, 4]
[5, 3, 6]
[8, 1, 0]

U
[7, 2, 4]
[5, 3, 0]
[8, 1, 6]

L
[7, 2, 4]
[5, 0, 3]
[8, 1, 6]

L
[7, 2, 4]
[0, 5, 3]
[8, 1, 6]
```

```
U
[0, 2, 4]
[7, 5, 3]
[8, 1, 6]

R
[2, 0, 4]
[7, 5, 3]
[8, 1, 6]

R
[2, 4, 0]
[7, 5, 3]
[8, 1, 6]

D
[2, 4, 3]
[7, 5, 0]
[8, 1, 6]

L
[2, 4, 3]
[7, 0, 5]
[8, 1, 6]
```

```
D
[2, 4, 3]
[7, 1, 5]
[8, 0, 6]

L
[2, 4, 3]
[7, 1, 5]
[0, 8, 6]

U
[2, 4, 3]
[0, 1, 5]
[7, 8, 6]

R
[2, 4, 3]
[1, 0, 5]
[7, 8, 6]

U
[2, 0, 3]
[1, 4, 5]
[7, 8, 6]

L
[0, 2, 3]
[1, 4, 5]
[7, 8, 6]
```

```
D
[1, 2, 3]
[0, 4, 5]
[7, 8, 6]

R
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

R
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

D
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Total number of steps: 21
Total amount of time in search: 0.175784349441528 32 second(s)
```

Số lượng Steps: 21

Thời gian chạy 0.17s

Bài làm của em đã xong, cảm ơn cô đã đọc và theo dõi. Em cảm ơn cô nhiều.