



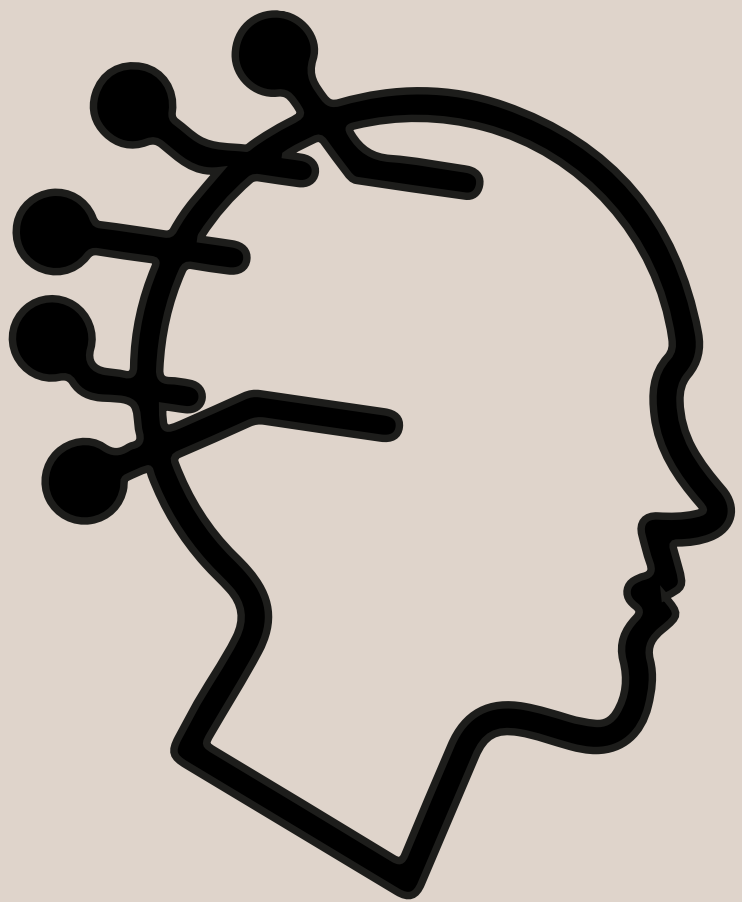
HEURISTIC

SEARCH



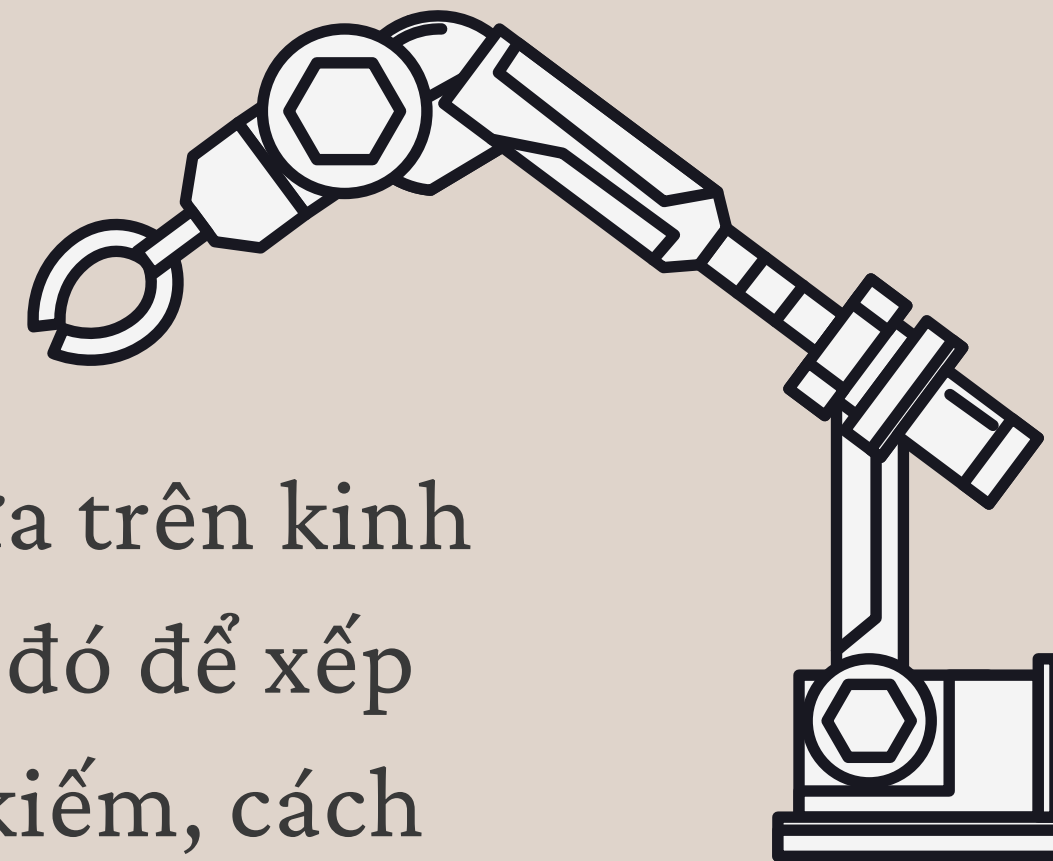
Khái niệm về Heuristic

Heuristic: là các kỹ thuật dựa trên kinh nghiệm để giải quyết vấn đề, nhằm đưa ra một giải pháp mà không được đảm bảo là tối ưu

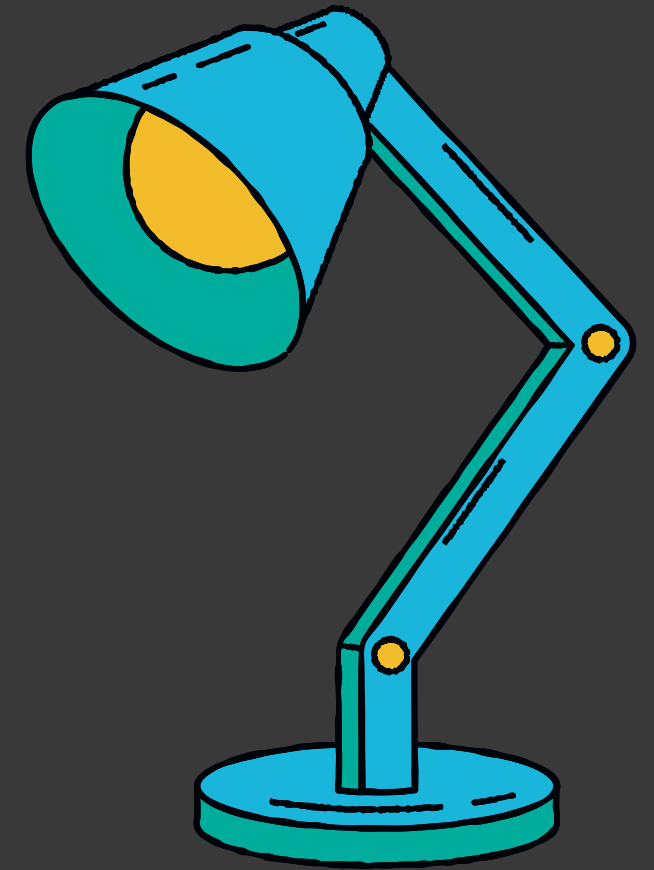


Heuristic function

Hàm đánh giá dựa trên kinh nghiệm, dựa vào đó để xếp hạng thứ tự tìm kiếm, cách chọn hàm đánh giá quyết định nhiều đến kết quả tìm kiếm.



Thuật toán sử dụng hàm đánh giá



01

Beam

Một thuật toán tìm kiếm heuristic. Nó được sử dụng trong các bài toán như dịch máy, nhận dạng giọng nói, tóm tắt văn bản,...

02

Hill Climbing

Một trong những kỹ thuật dùng để tìm kiếm tối ưu cục bộ cho một bài toán tối ưu.

03

Greedy Best First search

Một chiến lược tìm kiếm với tri thức bổ sung từ việc sử dụng các tri thức cụ thể của bài toán.

04

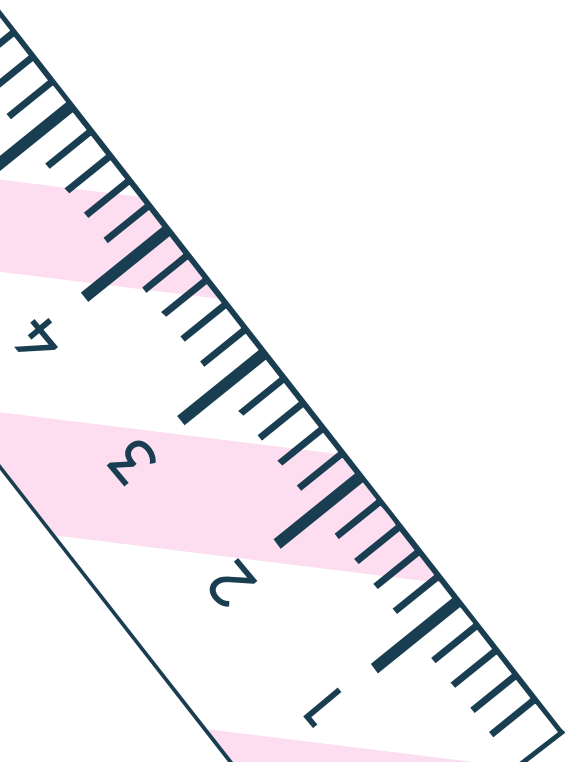
A*

Giải thuật tìm kiếm trong đồ thị, tìm đường đi từ một đỉnh hiện tại đến đỉnh đích có sử dụng hàm để ước lượng khoảng cách

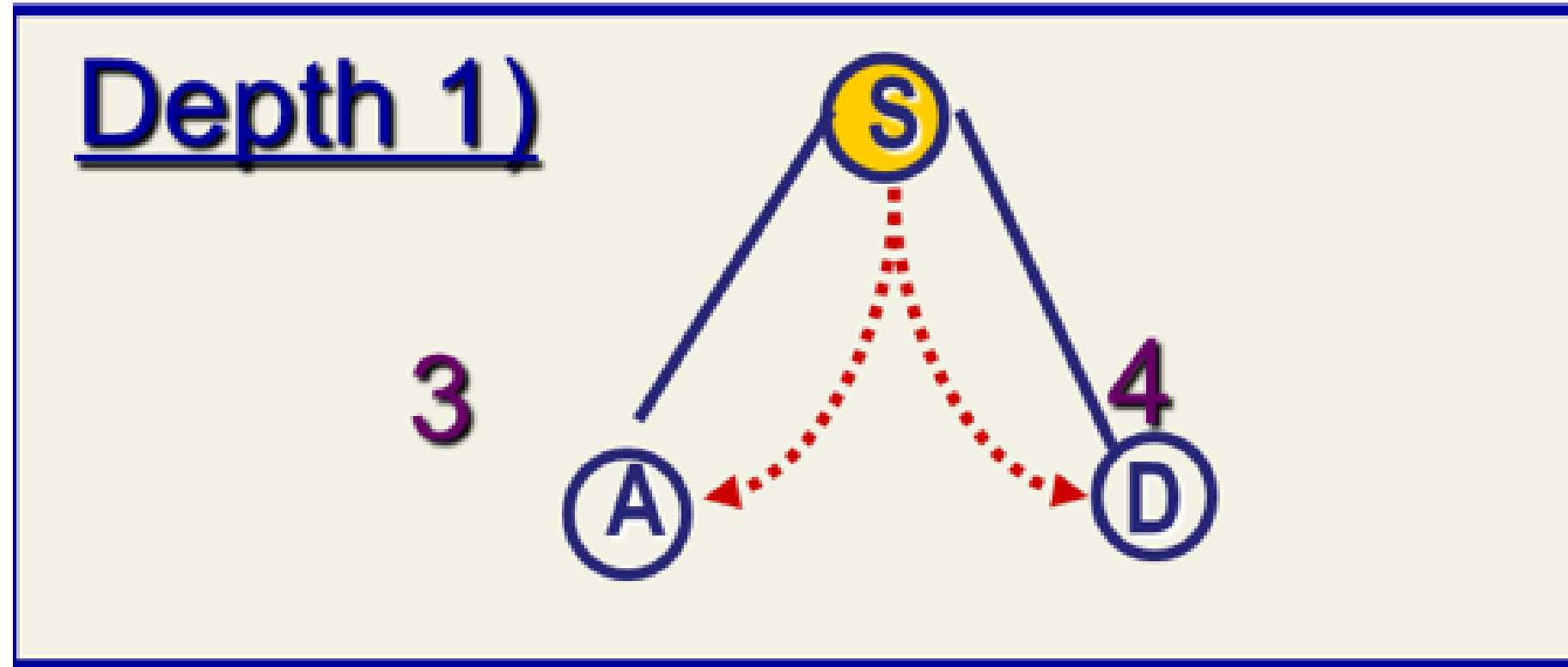
BEAM SEARCH

- Tương tự như tìm kiếm tốt nhất đầu tiên
- Tuy nhiên chỉ phát triển k đỉnh ở mức tiếp theo chứ không phát triển toàn bộ
- Ưu điểm: độ phức tạp tính toán tốt hơn
- Nhược điểm: không tìm kiếm toàn bộ, nên có thể không tìm thấy đỉnh tối ưu nhất

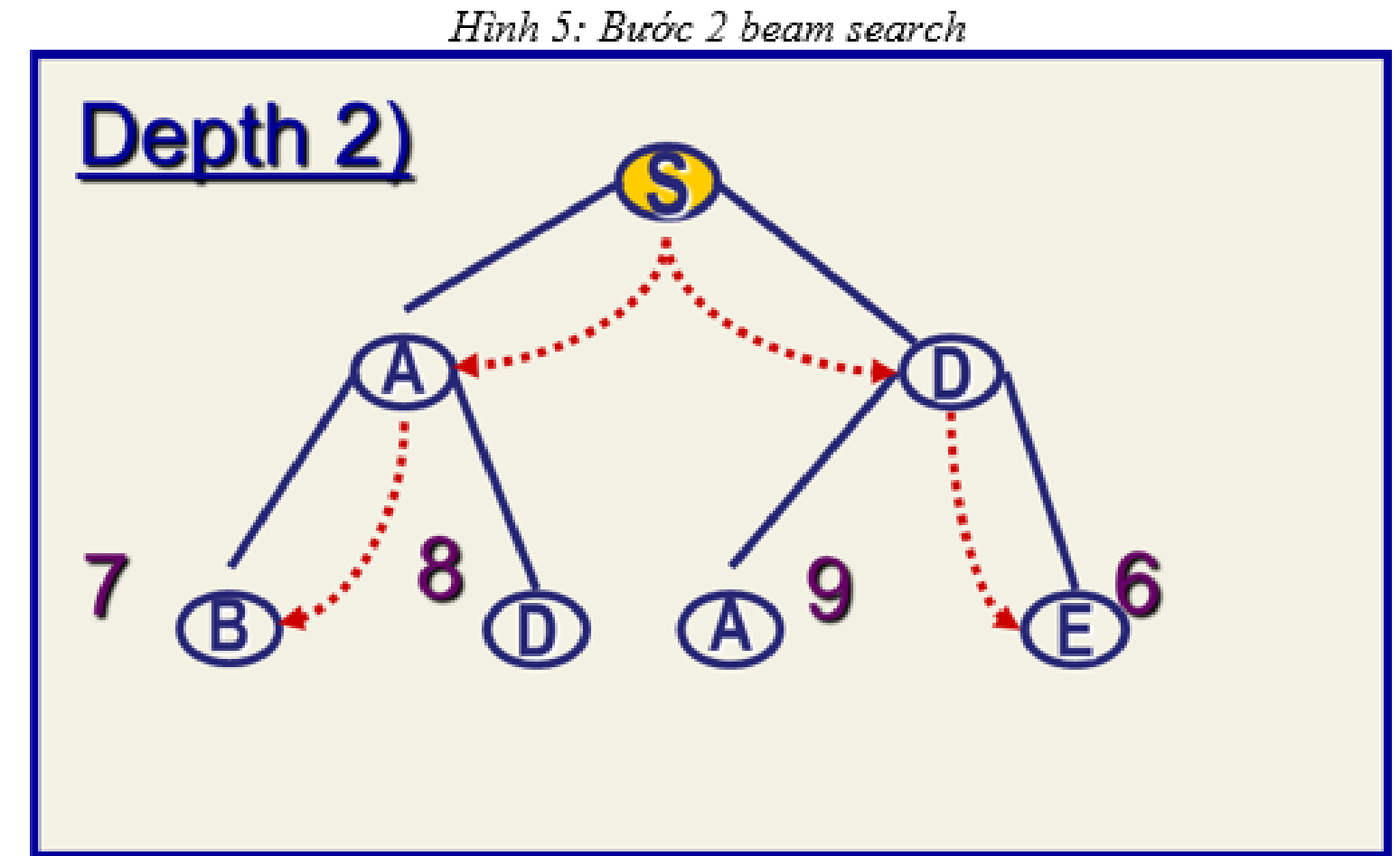
Ví dụ beam search với $k = 2$:



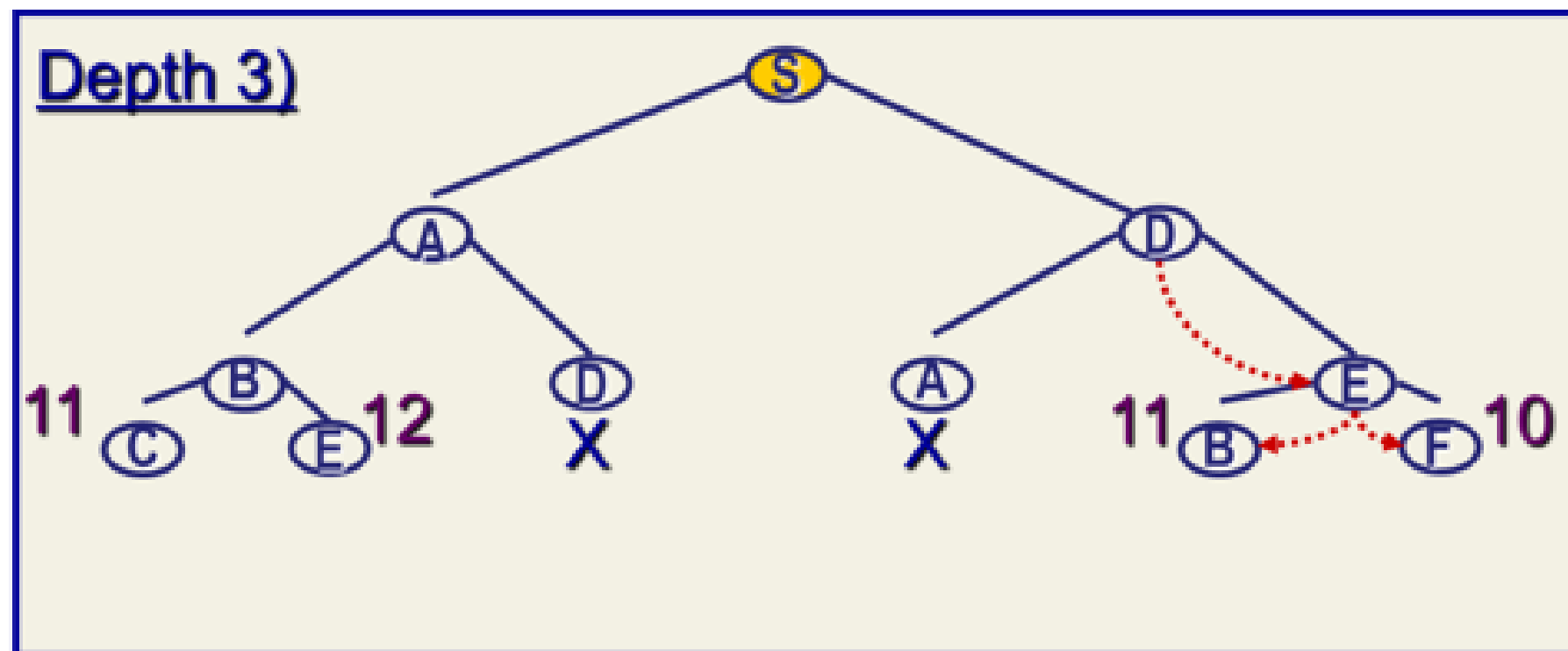
Ví dụ



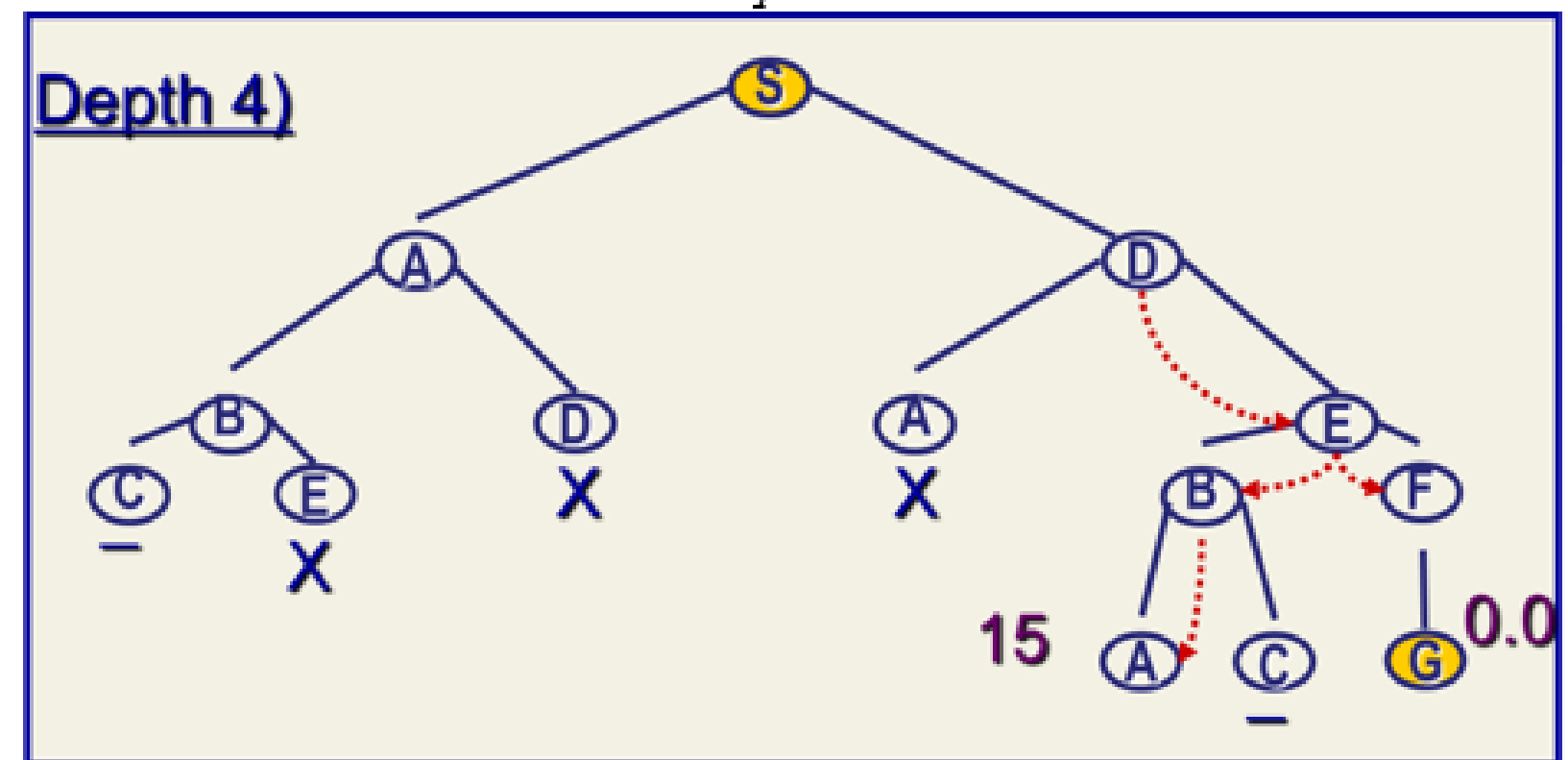
Hình 4: Bước 1 beam search



Hình 5: Bước 2 beam search



Hình 6: Bước 3 beam search

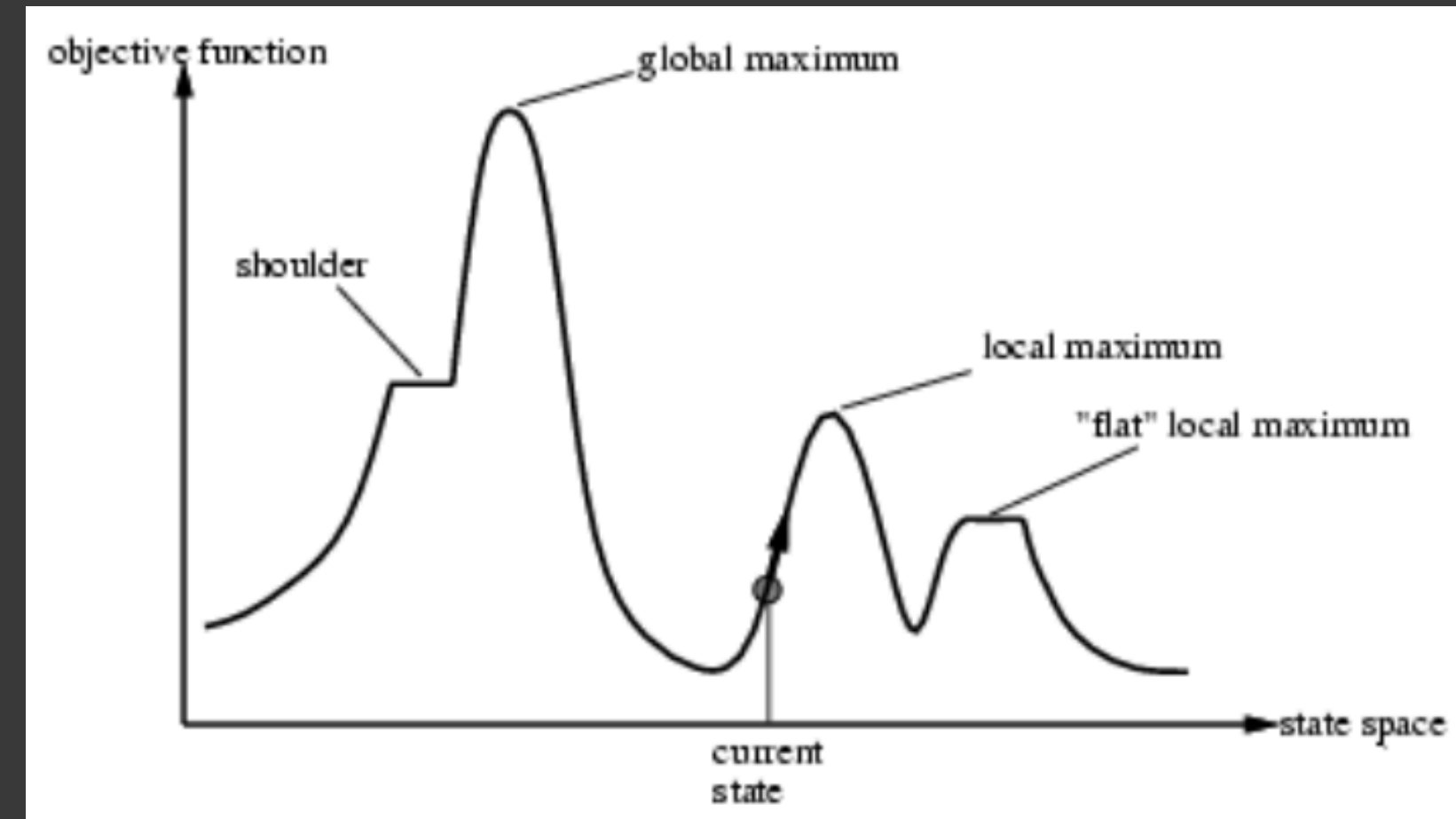


Hình 7: Kết quả beam search

Hill Climbing search

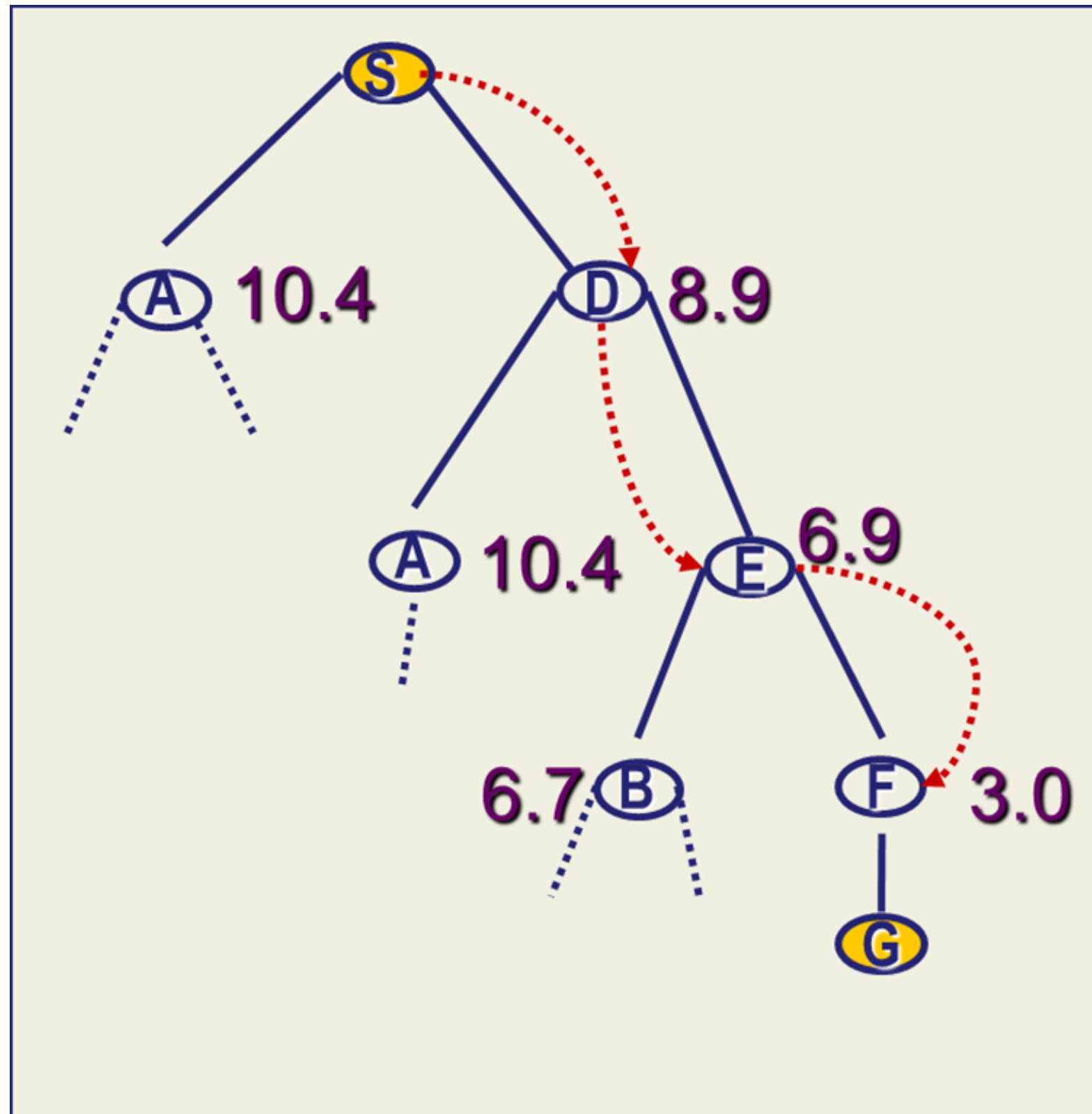
- Tìm kiếm leo đồi = tìm kiếm theo độ sâu + hàm đánh giá
- Khác với tìm kiếm theo độ sâu, ta phát triển một đỉnh u thì bước tiếp theo, ta chọn trong số các đỉnh con của u, đỉnh có nhiều hứa hẹn nhất để phát triển

Đỉnh hứa hẹn nhất là đỉnh có hàm đánh giá nhỏ nhất



VÍ DỤ

Tìm đường từ S tới G



Greedy - best first search

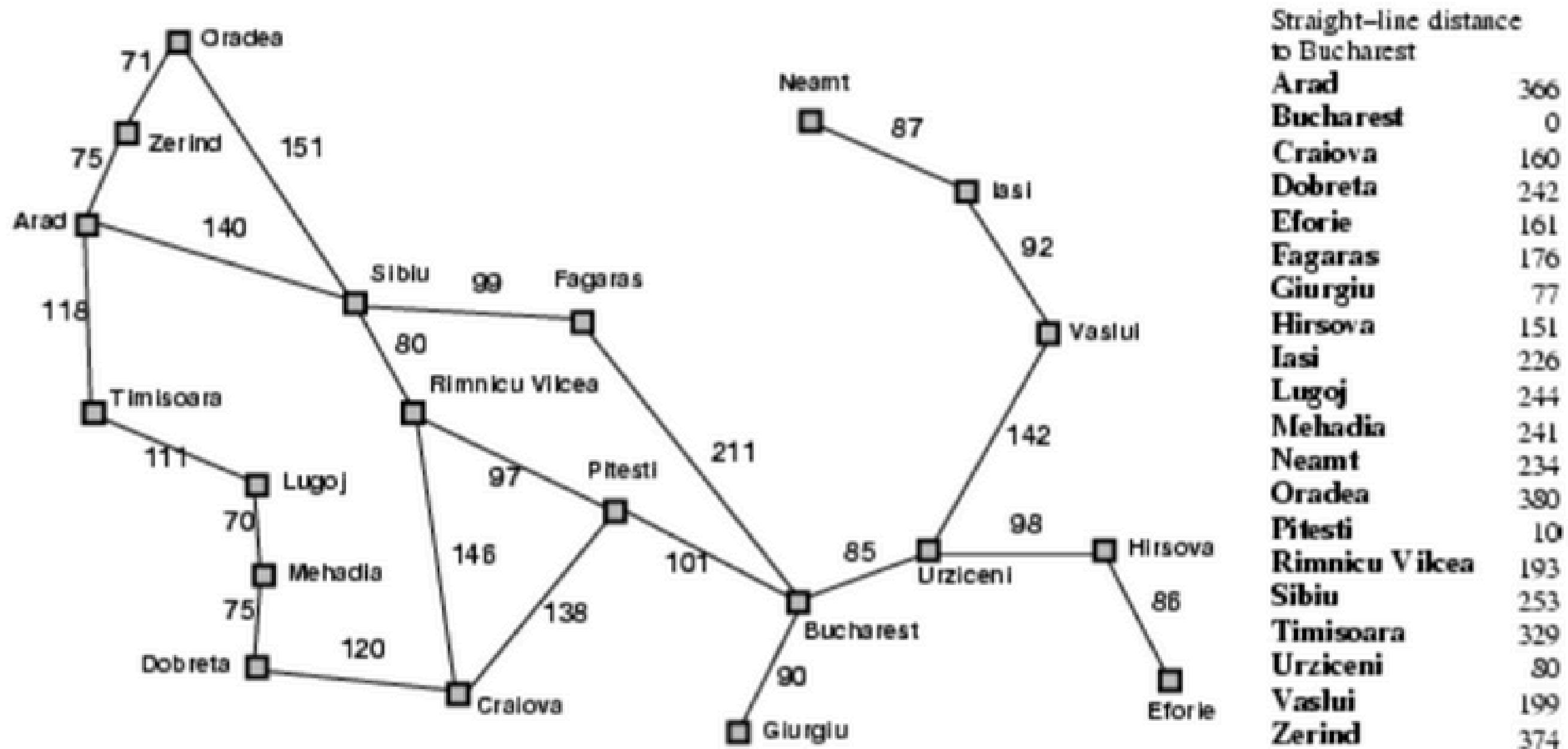
- Hàm đánh giá $f(n)$ là hàm heuristic $h(n)$
- Hàm heuristic $h(n)$ đánh giá chi phí để đi từ nút hiện tại n đến nút đích (mục tiêu)
- Phương pháp tìm kiếm Greedy - best first search sẽ xét (phát triển) nút "có vẻ" gần với nút đích (mục tiêu) nhất.

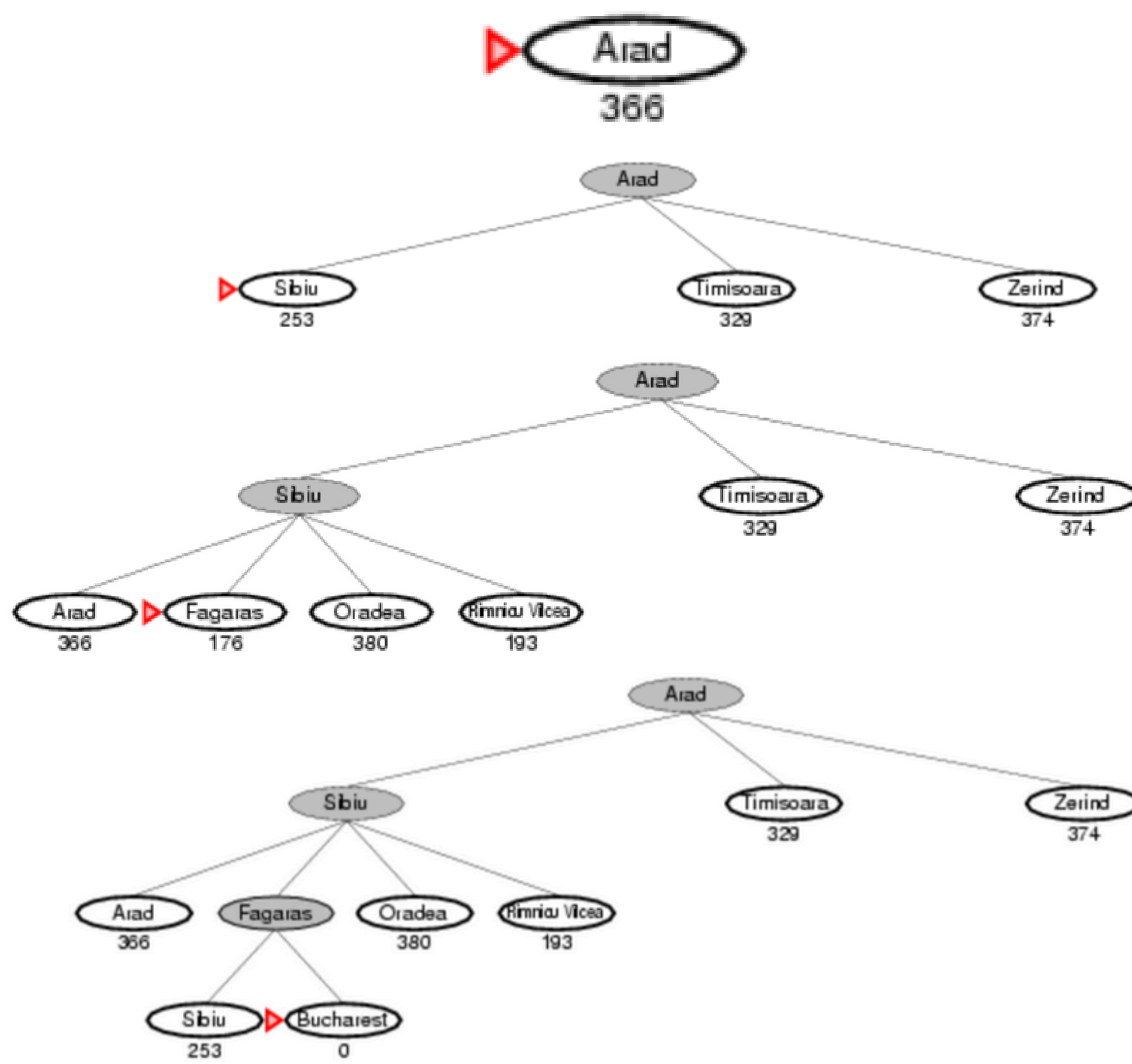


VÍ DỤ

Trong bài toán tìm đường đi từ Arad đến

Bucharest, sử dụng: $hSLD(n)$ = Ước lượng khoảng cách đường thẳng (“chim bay”) từ thành phố hiện tại n đến Bucharest





Đặc điểm của Greedy best-first search



Tính hoàn chỉnh

☐ Không – Vì có thể vướng (chết tắc) trong các vòng lặp



Độ phức tạp về thời gian

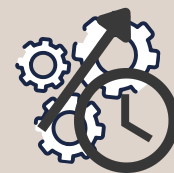
☐ $O(b^m)$

☐ Một hàm heuristic tốt có thể mang lại cải thiện lớn



Độ phức tạp về bộ nhớ

☐ $O(b^m)$ – Lưu giữ tất cả các nút trong bộ nhớ

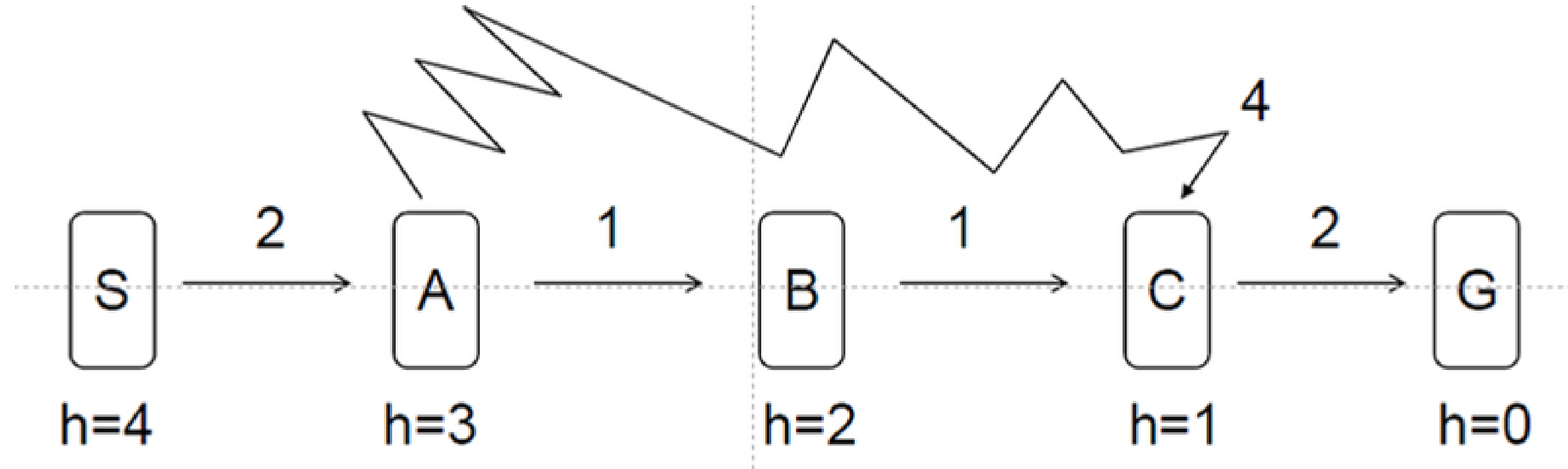


Tính tối ưu

☐ Không

G-BFS hiệu quả nhưng thật sự chưa chắc đã tối ưu, đôi khi hơi ngớ ngẩn .





Nếu chỉ sử dụng Greedy - Best First Search thì đường đi sẽ là: $S \rightarrow A \rightarrow C \rightarrow G$ ($2+4+2=8$)

Trong khi nếu tối ưu thì sẽ là $S \rightarrow A \rightarrow B \rightarrow C \rightarrow G$ (Chỉ tốn $2+1+1+2=6$)



A* search

Tránh việc xét (phát triển) các nhánh tìm kiếm đã xác định (cho đến thời điểm hiện tại) là có chi phí cao.

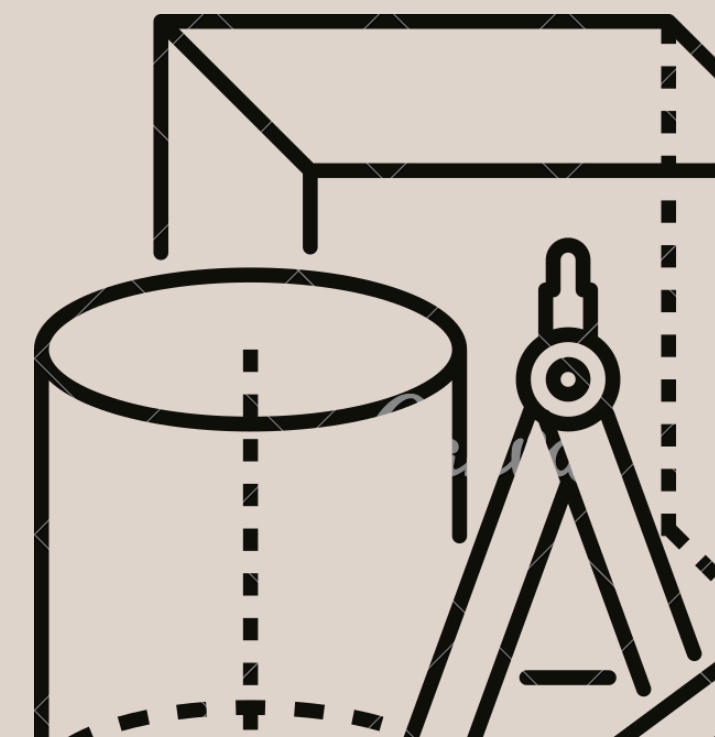


Sử dụng hàm đánh giá $f(n) = g(n) + h(n)$

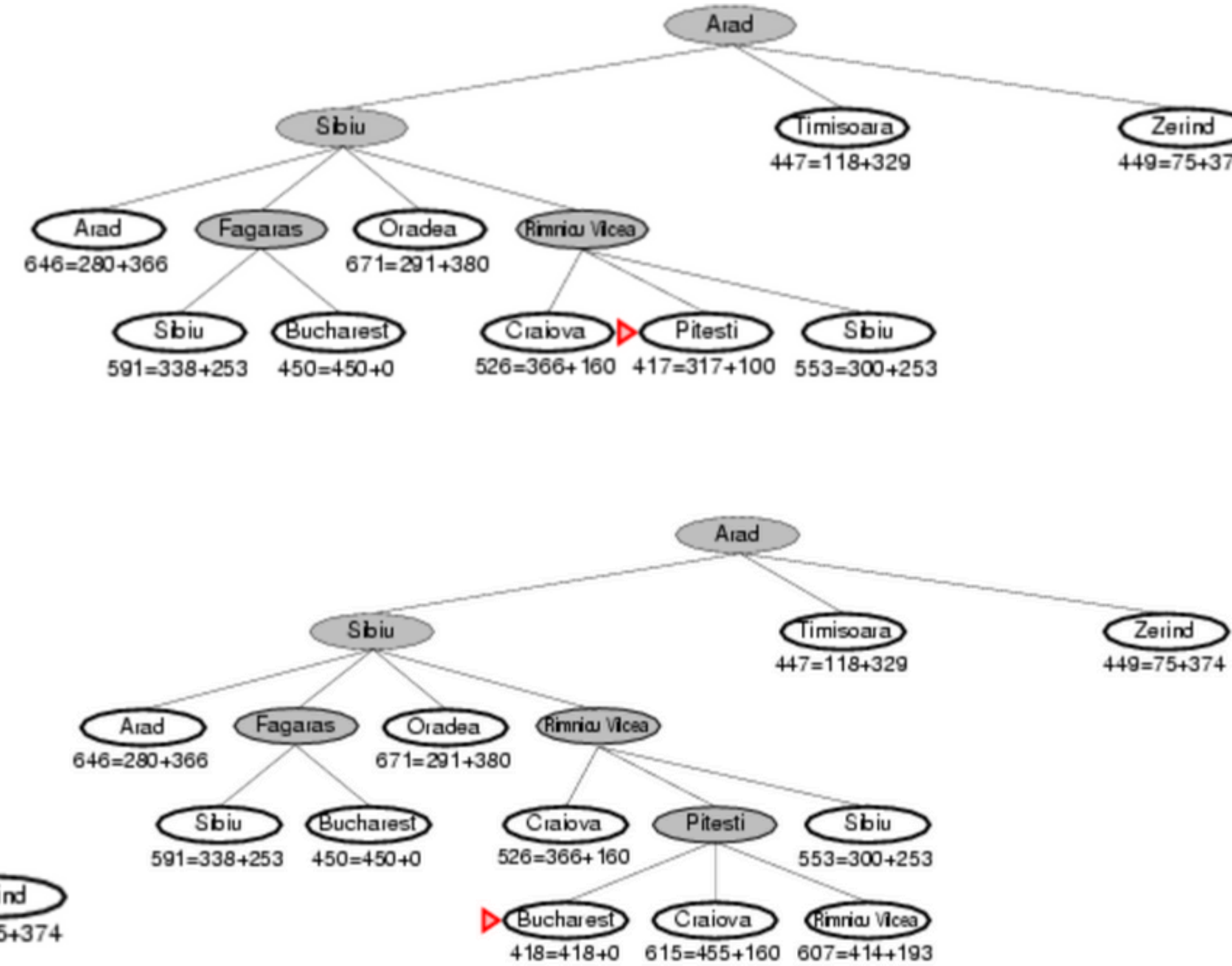
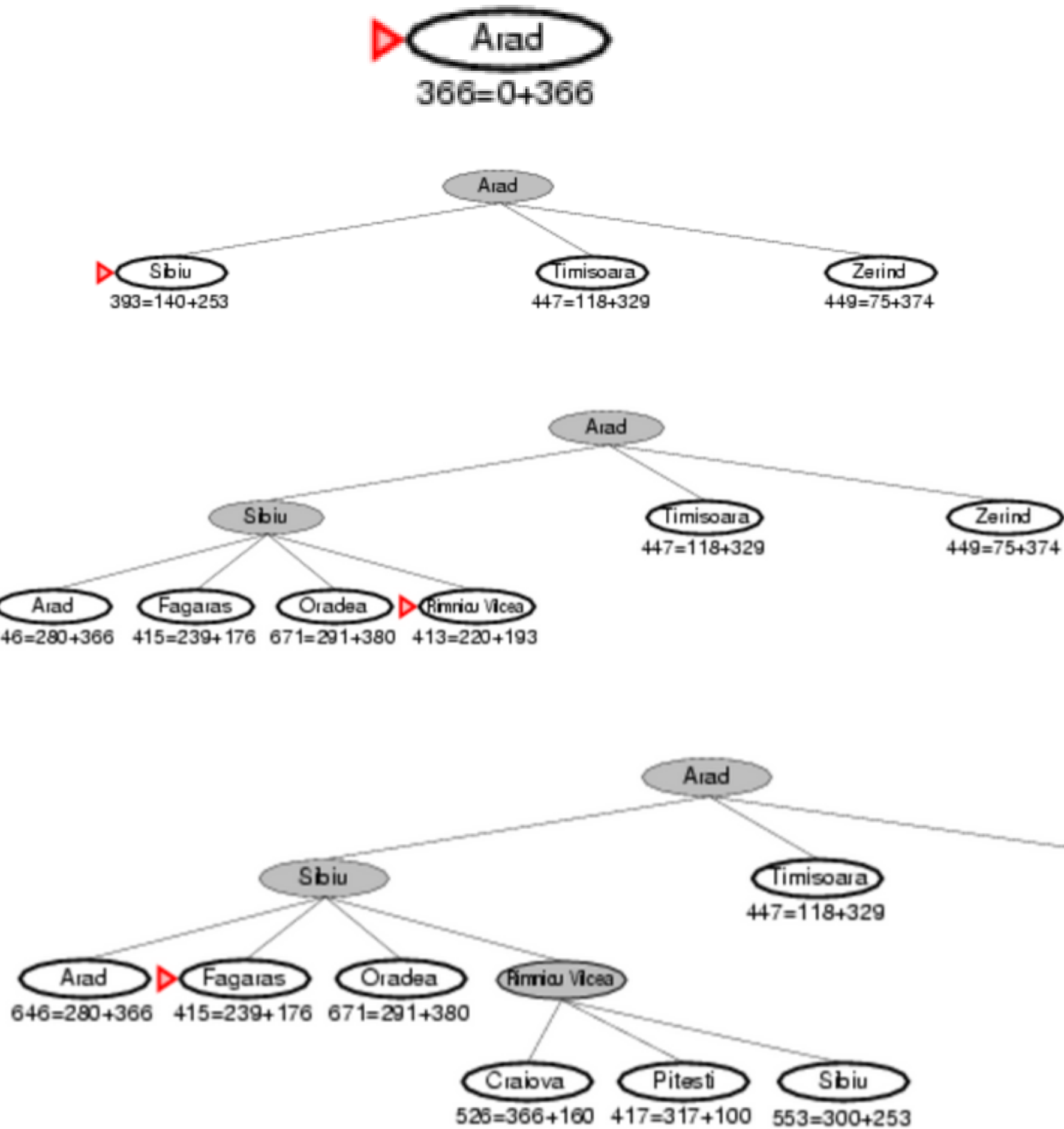
□ $g(n)$ = chi phí từ nút gốc cho đến nút hiện tại n

□ $h(n)$ = chi phí ước lượng từ nút hiện tại n tới đích

□ $f(n)$ = chi phí tổng thể ước lượng của đường đi qua nút hiện tại n đến đích



VÍ DỤ



Đặc điểm A*search

- Nếu không gian các trạng thái là hữu hạn và có giải pháp để tránh việc xét (lặp) lại các trạng thái, thì giải thuật A* là hoàn chỉnh (tìm được lời giải) – nhưng không đảm bảo là tối ưu
- Nếu không gian các trạng thái là hữu hạn và không có giải pháp để tránh việc xét (lặp) lại các trạng thái, thì giải thuật A* là không hoàn chỉnh
- Nếu không gian các trạng thái là vô hạn, thì giải thuật A* là không hoàn chỉnh
- Khi nào A* tối ưu ?

Các ước lượng chấp nhận được

- Một ước lượng $h(n)$ được xem là chấp nhận được nếu đối với mọi nút n : $0 \leq h(n) \leq h^*(n)$, trong đó $h^*(n)$ là chi phí thật (thực tế) để đi từ nút n đến đích
- Một ước lượng chấp nhận được không bao giờ đánh giá quá cao (overestimate) đối với chi phí để đi tới đích
 - Thực chất, ước lượng chấp nhận được có xu hướng đánh giá “lạc quan”
- Ví dụ: Ước lượng $h_{SLD}(n)$ đánh giá thấp hơn khoảng cách đường đi thực tế
- Định lý: Nếu $h(n)$ là đánh giá chấp nhận được, thì phương pháp tìm kiếm A^* sử dụng giải thuật TREESEARCH là tối ưu

Đặc điểm A^* search

01

- Tính hoàn chỉnh?
- Có (trừ khi có rất nhiều các nút có chi phí $f \leq f(G)$)

02

- Độ phức tạp về thời gian?
- Bậc của hàm mũ – Số lượng các nút được xét là hàm mũ của độ dài đường đi của lời giải

03

- Độ phức tạp về bộ nhớ?
- Lưu giữ tất cả các nút trong bộ nhớ

04

- Tính tối ưu?
- Có (đối với điều kiện đặc biệt)

SO SÁNH A* SEARCH VÀ GREEDY - BEST FIRST SEARCH

A* SEARCH

A* search sử dụng kinh nghiệm và chi phí thực tế
 $f(n) = h(n) + g(n)$

A* sử dụng kiến thức thực tế thông qua $g(n)$ là chi phí từ điểm xuất phát đến n

A* luôn có thể tìm thấy đích đến, miễn là nó tồn tại

A* giữ lại tất cả các nút trong bộ nhớ khi tìm kiếm

GREEDY - BEST FIRST SEARCH

Greedy - best first search chỉ sử dụng kinh nghiệm
 $f(n) = h(n)$

Greedy không sử dụng kiến thức thực tế

Greedy có thể bị chết tắc, bỏ sót đích

G-BFS chỉ giữ lại các nút biên hoặc nút giáp ranh bộ nhớ khi tìm kiếm

Cả 2 đều có độ phức tạp thời gian là $O(b^n)$

b là số con của cây

n là độ sâu của cây

Áp dụng thuật toán A* cho bài toán 8 puzzle.

1. Giới thiệu về bài toán 8 số.

- Bài toán 8-puzzle (hay còn gọi là 8 số) là một bài toán trí tuệ cổ điển, quen thuộc với những người bắt đầu tiếp cận với môn Trí tuệ nhân tạo. Bài toán có nhiều phiên bản khác nhau dựa theo số ô, như 8-puzzle, 15-puzzle,...
- Bài toán gồm một bảng ô vuông kích thước 3x3, có tám ô được đánh số từ 1 tới 8 và một ô trống. Trạng thái ban đầu, các ô được sắp xếp một cách ngẫu nhiên, nhiệm vụ của người chơi là tìm cách đưa chúng về trạng thái mục tiêu đúng thứ tự như hình bên:
- Trong quá trình giải bài toán, tại mỗi bước, ta giả định chỉ có ô trống là di chuyển. Như vậy, tối đa ô trống có thể có 4 khả năng di chuyển (lên trên, xuống dưới, sang trái, sang phải).

1	2	3
4	5	6
7	8	

2. Điều kiện của trạng thái đầu.

- Có những trạng thái của bảng số không thể chuyển về trạng thái đích. Người ta chứng minh được rằng, để có thể chuyển từ trạng thái đầu tới trạng thái đích, thì trạng thái đầu này phải thỏa mãn điều kiện được xác định như sau:

- Ta xét lần lượt từ trên xuống dưới, từ trái sang phải, với mỗi ô số đang xét (giả sử là ô thứ i), ta kiểm tra xem phía sau có bao nhiêu ô số có giá trị nhỏ hơn ô đó. Sau đó ta tính tổng $N = n_1 + n_2 + \dots + n_8$.

- Ta có quy tắc chung sau cho bài toán n -puzzle:

Nếu số ô vuông lẻ:

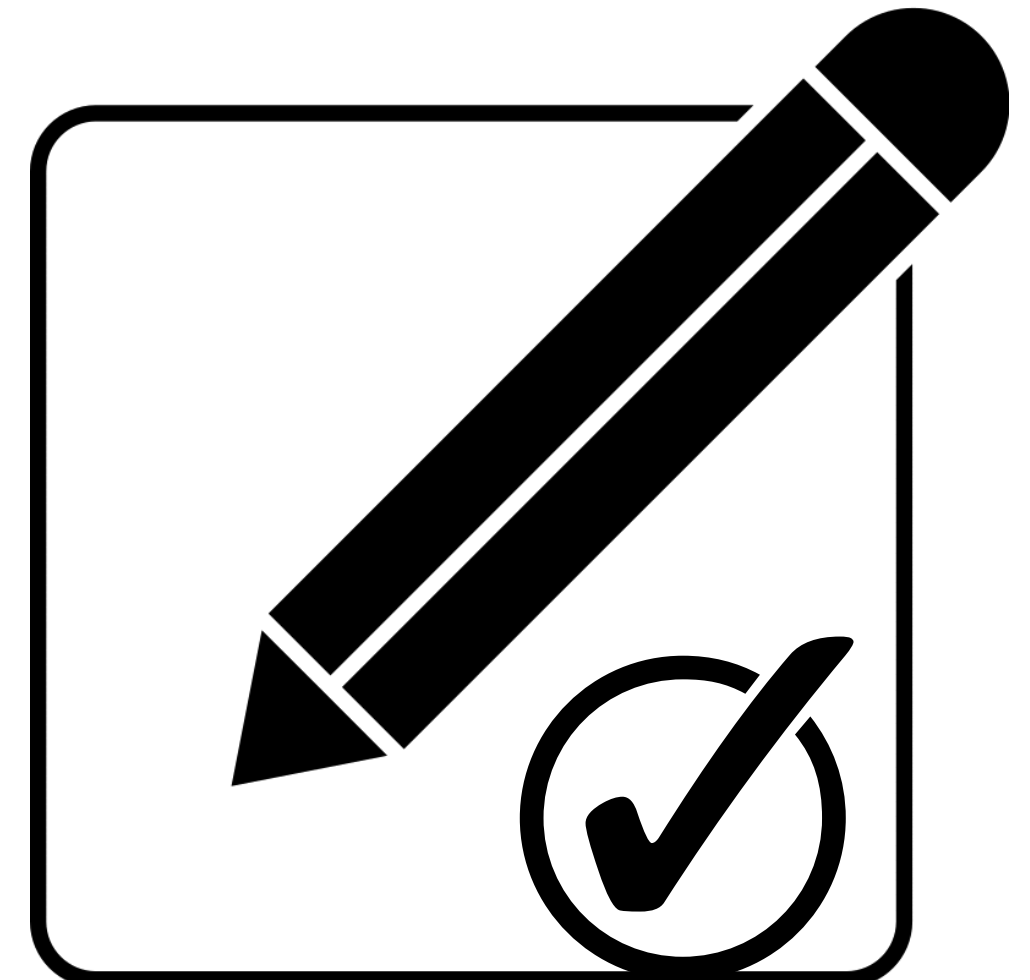
$N / 2$ dư 0

Nếu số ô vuông chẵn:

$N / 2$ dư 0 và ô trống phải nằm ở hàng chẵn xét từ trên xuống.

$N / 2$ dư 1 và ô trống phải nằm ở hàng lẻ xét từ trên xuống.

Cụ thể, ta đang xét bài toán 8-puzzle – có 9 ô vuông nên trạng thái đầu phải thỏa mãn điều kiện $N / 2$ dư 0



Ví dụ:

Cho trạng thái đầu sau 2-0-6-8-7-5-4-3-1:

Xét ô thứ nhất có giá trị 2: Phía sau có 1 ô nhỏ hơn (1) $\Rightarrow n_1 = 1$

Xét ô thứ hai có giá trị 6: Phía sau có 4 ô nhỏ hơn (5,4,3,1) $\Rightarrow n_2 = 4$

Xét ô thứ ba có giá trị 8: Phía sau có 5 ô nhỏ hơn (7,5,4,3,1) $\Rightarrow n_3 = 5$

Xét ô thứ tư có giá trị 7: Phía sau có 4 ô nhỏ hơn (5,4,3,1) $\Rightarrow n_4 = 4$

Xét ô thứ năm có giá trị 5: Phía sau có 3 ô nhỏ hơn (4,3,1) $\Rightarrow n_5 = 3$

Xét ô thứ sáu có giá trị 4: Phía sau có 2 ô nhỏ hơn (3,1) $\Rightarrow n_6 = 2$

Xét ô thứ bảy có giá trị 3: Phía sau có 1 ô nhỏ hơn (1) $\Rightarrow n_7 = 1$

Xét ô thứ tám có giá trị 1: Phía sau không còn ô nào nhỏ hơn $\Rightarrow n_8 = 0$

$$N = 1 + 4 + 5 + 4 + 3 + 2 + 1 + 0 = 20$$

Mà $20 / 2$ dư 0 \rightarrow Thỏa mãn

- Những trạng thái của bảng số mà có thể chuyển về trạng thái đích gọi là cấu hình hợp lệ, ngược lại gọi là cấu hình không hợp lệ.


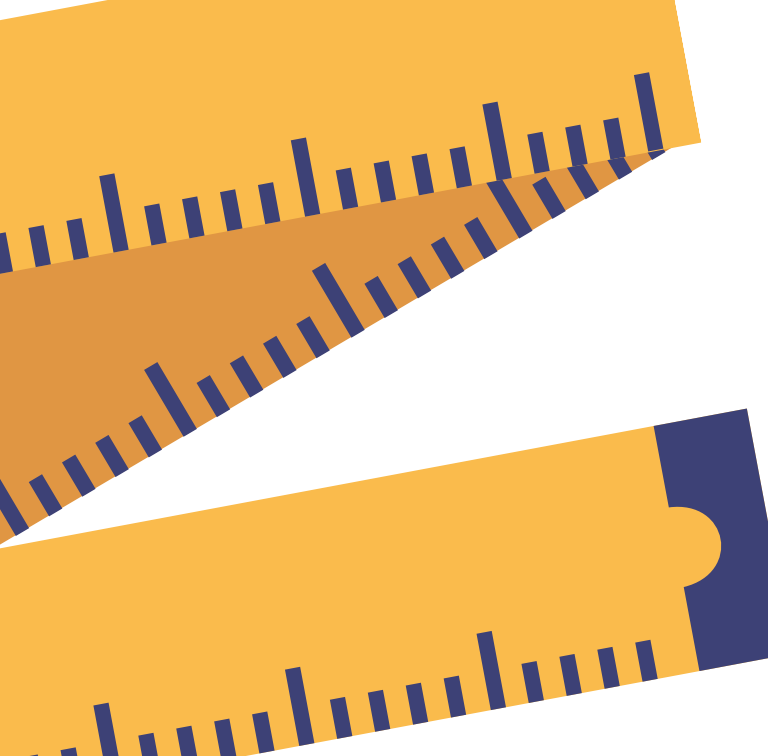
- Với bảng số kích thước $m \times m$ (m là cạnh) thì ta có không gian trạng thái là $(m \times m)!$

\rightarrow Với bài toán 8-puzzle, các trạng thái có thể có của bảng số là:

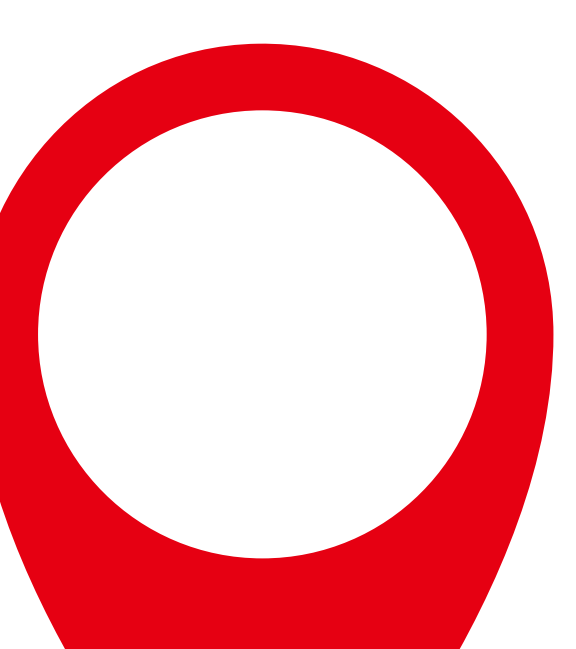
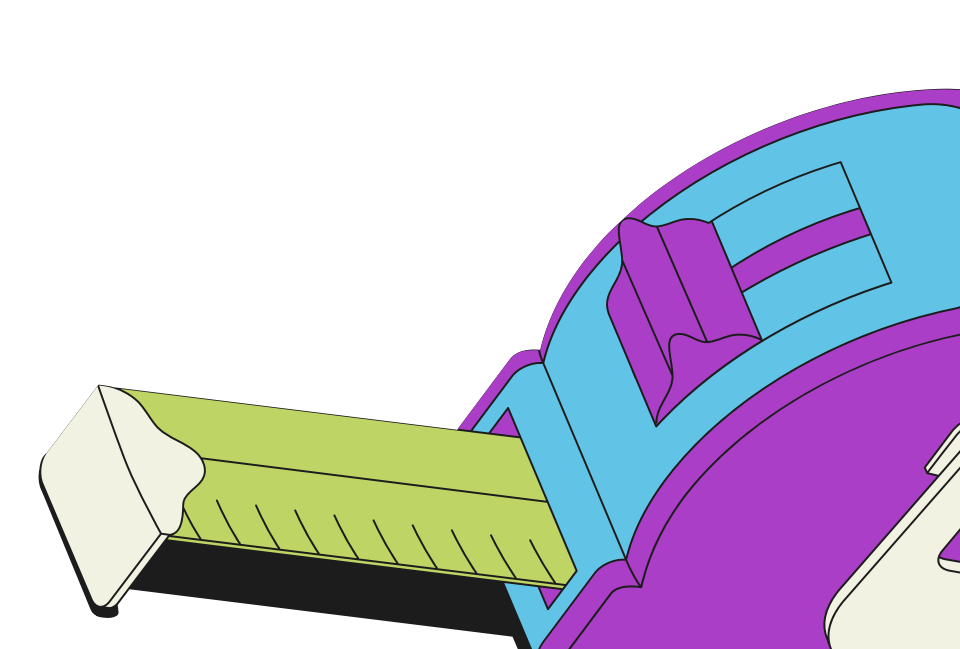
$$(3 \times 3)! = 362880$$

2		6
8	7	5
4	3	1

Trạng thái đầu của ô số



3. Áp dụng khoảng cách manhattan để tính hàm ước lượng heuristic $h(n)$.

- Hàm heuristic được sử dụng trong thuật toán A^* ước tính chi phí đi từ trạng thái hiện tại đến trạng thái mục tiêu. Trong bài toán 8 Puzzle, một hàm heuristic phổ biến là khoảng cách Manhattan, được tính bằng tổng số bước di chuyển tối thiểu cần thiết để di chuyển mỗi ô số từ vị trí hiện tại đến vị trí mục tiêu.
- 
- 

Ví dụ:

Có trạng thái bài toán sau

Có 5 ô số nằm sai vị trí so với trạng thái đích

=> $h1 = 5$

Tính khoảng cách Manhattan, xác định tọa độ (dòng, cột) của ô khi ở sai vị trí và khi ở đúng vị trí:

Ô số 3 khi ở sai vị trí có tọa độ (0,1)

Ô số 3 khi ở đúng vị trí có tọa độ (0,2)

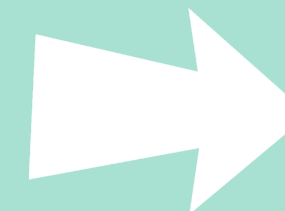
=> Khoảng cách Manhattan của ô số 3 bằng $|0 - 0| + |2 - 1| = 1$

Tương tự, ô số 4 có khoảng cách Manhattan là 3, ô số 8 có khoảng cách Manhattan là 2, ô số 6 có khoảng cách Manhattan là 2, ô số 2 có khoảng cách Manhattan là 3. Những ô số 1, 5, 7 đã ở đúng vị trí nên có khoảng cách Manhattan là 0.

=> $h2 = 0 + 1 + 3 + 2 + 0 + 0 + 2 + 3 = 11$

Hình 12: Trạng thái đầu

1	3	4
8	5	
7	6	2



Hình 13: Trạng thái đích

1	2	3
4	5	6
7	8	

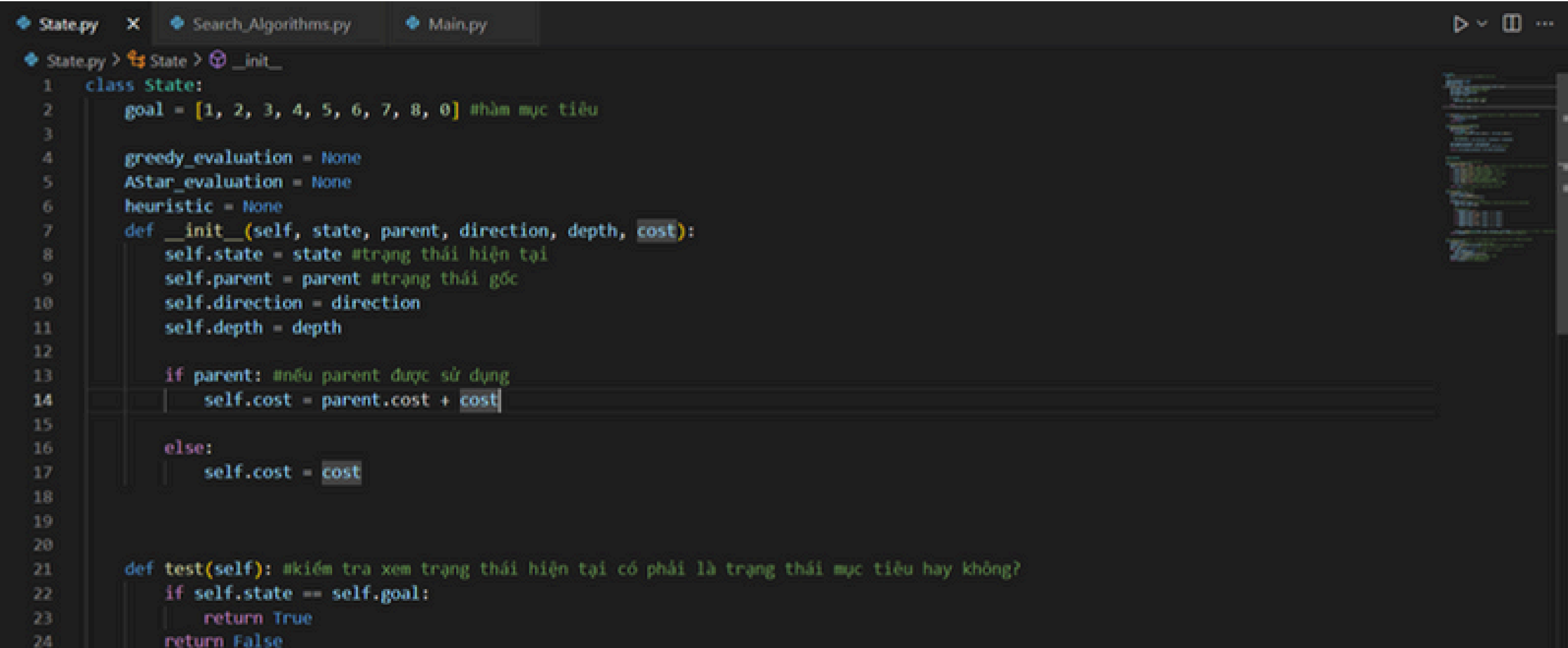
4. Giải thuật của bài toán.

- Để áp dụng thuật toán A^* vào bài toán 8 Puzzle, chúng ta cần biểu diễn bài toán dưới dạng đồ thị, với các nút là các trạng thái có thể xảy ra của bảng 8 Puzzle và các cạnh là các di chuyển hợp lệ giữa các trạng thái.
- Ban đầu ta có Open là tập chứa các trạng thái chưa được xét (sắp xếp theo f tăng dần), là một list chứa các trạng thái tiềm năng. Close là tập các trạng thái đã được xét. Ban đầu Open chỉ chứa trạng thái ban đầu, tập Close rỗng.
- Thuật toán:
- Lấy trạng thái có giá trị $f(n)$ (tổng chi phí ước tính từ trạng thái ban đầu đến trạng thái n) thấp nhất từ danh sách mở.
- Di chuyển trạng thái này sang danh sách đóng.
- Mở rộng trạng thái này bằng cách xem xét tất cả các di chuyển hợp lệ.
- Đối với mỗi di chuyển, tính toán heuristic và chi phí di chuyển.
- Cập nhật giá trị $f(n)$ và $g(n)$ (chi phí di chuyển thực tế từ trạng thái ban đầu đến trạng thái n).
- Nếu đã tìm thấy trạng thái đích -> backtracking lại.



5. Lập trình về bài toán 8 puzzle.

a. Trạng thái



```
State.py X Search_Algorithms.py Main.py
State.py > State > __init__
1 class State:
2     goal = [1, 2, 3, 4, 5, 6, 7, 8, 0] #hàm mục tiêu
3
4     greedy_evaluation = None
5     AStar_evaluation = None
6     heuristic = None
7     def __init__(self, state, parent, direction, depth, cost):
8         self.state = state #trạng thái hiện tại
9         self.parent = parent #trạng thái gốc
10        self.direction = direction
11        self.depth = depth
12
13        if parent: #nếu parent được sử dụng
14            self.cost = parent.cost + cost
15
16        else:
17            self.cost = cost
18
19
20
21    def test(self): #kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu hay không?
22        if self.state == self.goal:
23            return True
24        return False
```

a. Trạng thái

```
26 #tính toán khoảng cách manhattan
27 def Manhattan_Distance(self ,n):
28     self.heuristic = 0
29     for i in range(1 , n*n):
30         distance = abs(self.state.index(i) - self.goal.index(i))
31
32         #hàm đánh giá
33         self.heuristic = self.heuristic + distance/n + distance*n
34
35     self.greedy_evaluation = self.heuristic
36     self.AStar_evaluation = self.heuristic + self.cost #f=g+h
37
38     return( self.greedy_evaluation, self.AStar_evaluation)
39
40
41
42 @staticmethod
43
44 #loại bỏ các nước đi chuyển bất lợi
45 def available_moves(x,n):
46     moves = ['Left', 'Right', 'Up', 'Down'] #list này chứa toàn bộ các nước đi chuyển có thể xảy ra
47     if x % n == 0: #khoảng trống ở cột trái
48         moves.remove('Left') #loại bỏ phương án dịch trái
49     if x % n == n-1: #khoảng trống ở cột phải
50         moves.remove('Right') #loại bỏ phương án dịch phải
51     if x - n < 0: #khoảng trống ở hàng trên
52         moves.remove('Up') #loại bỏ phương án dịch lên
53     if x + n > n*n - 1: #khoảng trống ở hàng dưới
54         moves.remove('Down') #loại bỏ phương án dịch xuống
55
56     return moves #trả về phương án dịch chuyển khả thi
```

a. Trạng thái

```
58 #xét các trạng thái lân cận
59 def expand(self, n):
60     x = self.state.index(0)
61     moves = self.available_moves(x, n)
62
63     children = [] #lưu trữ các lân cận
64     for direction in moves: #thử các phương án dịch chuyển khả thi có trong moves
65         temp = self.state.copy()
66
67         #hoán đổi vị trí
68         if direction == 'Left':
69             temp[x], temp[x - 1] = temp[x - 1], temp[x]
70         elif direction == 'Right':
71             temp[x], temp[x + 1] = temp[x + 1], temp[x]
72         elif direction == 'Up':
73             temp[x], temp[x - n] = temp[x - n], temp[x]
74         elif direction == 'Down':
75             temp[x], temp[x + n] = temp[x + n], temp[x]
76
77
78     children.append(State(temp, self, direction, self.depth + 1, 1)) #thêm các lân cận và truyền thêm các thuộc tính
79     return children #trả về các lân cận có thể đạt được từ trạng thái đang xét
82 #xét trạng thái hiện tại và trả về hướng đi chuyển, xét đến khi nào không còn parent
83 def solution(self):
84     solution = [] #lưu trữ các hướng giải quyết
85     solution.append(self.direction) #thêm trạng thái hiện tại vào danh sách
86     path = self
87     while path.parent != None: #đò ngược lại các parent
88         path = path.parent
89         solution.append(path.direction)
90     solution = solution[::-1] #xóa đi phần tử cuối
91     solution.reverse() #đảo ngược lại list
92     return solution #trả về tập phương án
```

b. Áp dụng thuật toán A*

```
1  from State import State
2  from queue import PriorityQueue
3  from queue import Queue
4  from queue import LifoQueue
5
6  def AStar_search(given_state , n):
7      frontier = PriorityQueue() #xác định độ ưu tiên trong hàng chờ
8      explored = [] #mảng dùng để lưu trữ các trường hợp đã xét
9      counter = 0
10     root = State(given_state, None, None, 0, 0)
11     evaluation = root.Manhattan_Distance(n) #tính toán hàm đánh giá A* cho trạng thái xuất phát
12     frontier.put((evaluation[1], counter, root)) #dựa theo hàm đánh giá A*
13
14     while not frontier.empty(): #khi vẫn còn trạng thái cần xét
15         current_node = frontier.get()
16         current_node = current_node[2]
17         explored.append(current_node.state) #thêm nút hiện tại vào close
18
19         if current_node.test(): #kiểm tra xem trạng thái hiện tại có phải là trạng thái đích hay không
20             return current_node.solution(), len(explored) #trả về lời giải và số lượng trạng thái đã duyệt
21
22         children = current_node.expand(n) #xét các lân cận của nút vừa xét
23         for child in children:
24             if child.state not in explored: #nếu lân cận đó chưa được đưa vào close
25                 counter += 1 #tăng thêm bộ đếm
26                 evaluation = child.Manhattan_Distance(n) #sử dụng hàm đánh giá A* cho lân cận
27                 frontier.put((evaluation[1], counter, child)) #thêm lân cận vào frontier
28     return
```

c. Hàm chạy chính

```
1  from Search_Algorithms import AStar_search
2
3  #khởi tạo trạng thái ban đầu
4  n = 3
5  print("Enter your" ,n,"*",n, "puzzle")
6  root = []
7  for i in range(0,n*n):
8      p = int(input())
9      root.append(p)
10
11  print("The given state is:", root)
12
13
14  #tính toán tổng N
15  def inv_num(puzzle):
16      inv = 0
17      for i in range(len(puzzle)-1):
18          for j in range(i+1 , len(puzzle)):
19              if (( puzzle[i] > puzzle[j]) and puzzle[i] and puzzle[j]):
20                  inv += 1
21      return inv
22
23  #xét xem có giải được trạng thái ban đầu này không: nếu như N lẻ thì không giải được
24  def solvable(puzzle):
25      inv_counter = inv_num(puzzle)
26      if (inv_counter %2 ==0):
27          return True
28      return False
29
30
31  #1,8,2,0,4,3,7,6,5 giải được
32  #2,1,3,4,5,6,7,8,0 không giải được
33
34  if solvable(root):
35      print("Solvable, please wait. \n")
36      AStar_solution = AStar_search(root, n)
37      print('A* Solution is ', AStar_solution[0]) #trả về phương án dịch chuyển
38      print('Number of explored nodes is ', AStar_solution[1]) #trả về số trạng thái đã duyệt
39
40
41  else:
42      print("Not solvable")
```


6. Test case

a. Các trường hợp không có lời giải.

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

$N = 2+5+0+2+1+1+2+0+0 = 13$ (số lẻ)

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

$N = 3+4+0+0+1+0+1+0 = 9$ (số lẻ)

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

$N = 6+1+0+0+0+1+1+0 = 9$ (số lẻ)

b. Các trường hợp có lời giải.

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

$N = 7+0+1+4+2+0+0+0 = 14$ (số chẵn)

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

$N = 3+3+2+2+3+2+1+0 = 16$ (số chẵn)

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

$N = 4+3+0+0+3+0+0+0 = 10$ (số chẵn)

```
The given state is: [3, 7, 0, 4, 2, 5, 8, 1, 6]
Not solvable
```

```
The given state is: [4, 6, 1, 0, 2, 5, 3, 8, 7]
Not solvable
```

```
The given state is: [7, 2, 1, 3, 4, 0, 6, 8, 5]
Not solvable
```

```
The given state is: [0, 8, 1, 3, 7, 5, 2, 4, 6]
Solvable, please wait.

A* Solution is  ['Down', 'Down', 'Right', 'Up', 'Up', 'Right', 'Down', 'Left', 'Up', 'Left', 'Down', 'Right', 'Right', 'Up', 'Left', 'Left', 'Down', 'Down', 'Right', 'Up', 'Right', 'Down']
```

```
The given state is: [4, 5, 0, 3, 6, 8, 7, 2, 1]
Solvable, please wait.

A* Solution is  ['Down', 'Down', 'Left', 'Up', 'Right', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Left', 'Up', 'Right', 'Right', 'Down', 'Left', 'Up', 'Right', 'Down', 'Down']
```

```
The given state is: [5, 4, 1, 2, 8, 3, 6, 0, 7]
Solvable, please wait.

A* Solution is  ['Left', 'Up', 'Up', 'Right', 'Right', 'Down', 'Left', 'Down', 'Right', 'Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Left', 'Up', 'Right', 'Down', 'Down', 'Right']
```

V. Kết luận.

- Nếu không gian trạng thái là hữu hạn và có lời giải thì có thể nói giải thuật A* là hoàn chỉnh, nhưng có thể chưa tối ưu. Muốn A* tối ưu thì hàm $h(n)$ phải có tính chấp nhận được – tức là nó không bao giờ đánh giá cao hơn chi phí nhỏ nhất thực sự của việc đi tới đích.
 - Độ phức tạp của A* phụ thuộc vào hàm đánh giá $h(n)$.
 - Thuật toán A* khá hiệu quả khi giải bài toán 8 puzzle. Giải thuật A* là thuật toán tìm kiếm trong đồ thị có thông tin phản hồi, sử dụng đánh giá heuristic để xếp loại từng nút và duyệt nút theo hàm đánh giá này.
- > Phương pháp tìm kiếm heuristic đã chứng minh được giá trị của mình trong việc giải quyết các bài toán tối ưu hóa và tìm kiếm. Với khả năng cung cấp giải pháp nhanh chóng thông qua việc ước lượng và đánh giá, các thuật toán heuristic không chỉ tiết kiệm thời gian mà còn đảm bảo hiệu quả trong nhiều tình huống phức tạp. Tuy nhiên, cần phải lưu ý rằng phương pháp này không phải lúc nào cũng đưa ra giải pháp chính xác nhất, mà đôi khi chỉ là giải pháp gần đúng. Do đó, việc lựa chọn và tinh chỉnh hàm heuristic sao cho phù hợp với từng bài toán cụ thể là yếu tố then chốt để tối ưu hóa hiệu suất của thuật toán. Trong tương lai, sự phát triển của các phương pháp heuristic hứa hẹn sẽ mở ra những khả năng mới trong việc giải quyết các thách thức mà thế giới hiện đại đang đối mặt. Hiện tại, tìm kiếm có thông tin được ứng dụng rộng rãi trong nhiều lĩnh vực như trò chơi, robot học, lập kế hoạch, chẩn đoán y tế, tra cứu thông tin,...



Thank
You