Chương 3

Tìm kiếm trên không gian trạng thái

Nội dung

1. Không gian trạng thái
2. Tìm kiếm theo chiều sâu trước
3. Tìm kiếm theo bề rộng trước
4. Tìm kiếm tốt nhất trước
5. Uniform cost search
6. Giải thuật A*
7. Giải bài toán 8-puzzle bằng giải thuật A*
8. Kết luận

1. Không gian trạng thái

Một Không gian trạng thái (state space) là 1 bộ [N, A, S, GD] trong đó:

- **N** (node): các *nút*/ *trạng thái* của đồ thị
- **A** (arc): các *cung*/ các *liên kết* giữa các nút
- **S** (start state): chứa *trạng thái ban đầu* của bài toán
- **G** (Goal state): chứa các *trạng thái đích* của bài toán

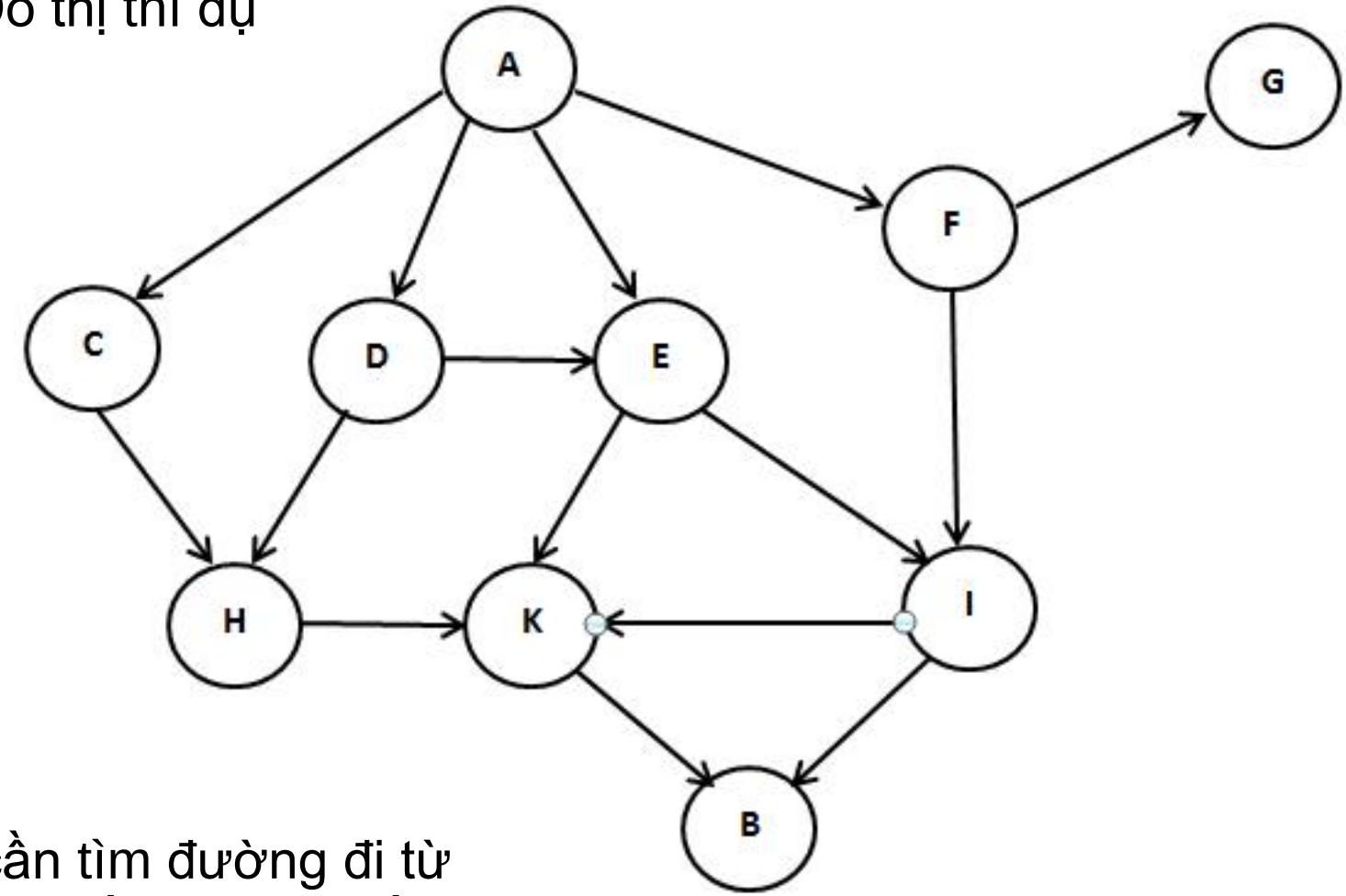
Không gian trạng thái có thể diễn tả bằng hình thức ***một đồ thị*** trong đó các nút biểu diễn các trạng thái và các cung biểu diễn các hành động để biến đổi từ trạng thái này sang trạng thái khác.

→ ***Đường đi của lời giải*** (solution path) là một con đường đi qua đồ thị này từ một nút thuộc S đến một nút thuộc G

Cây tìm kiếm

- Quá trình tìm kiếm trên không gian trạng thái có thể biểu diễn thành **cây tìm kiếm** (search tree).
- Cây tìm kiếm là một cây mà trong đó nút rễ biểu diễn trạng thái ban đầu (start state) và tập hợp các nút con của một nút bao gồm những trạng thái đến được từ nút đó bằng cách thực hiện một **hành động** nào đó.
- Một **hành động** (action**)** là một tác vụ nào đó mà giải thuật tìm kiếm chọn để thực hiện.
- **Hệ số rẽ nhánh** (branch factor) trong một cây tìm kiếm là số nút con trung bình mà một nút có thể có.

Hình 3.1: Đồ thị thí dụ

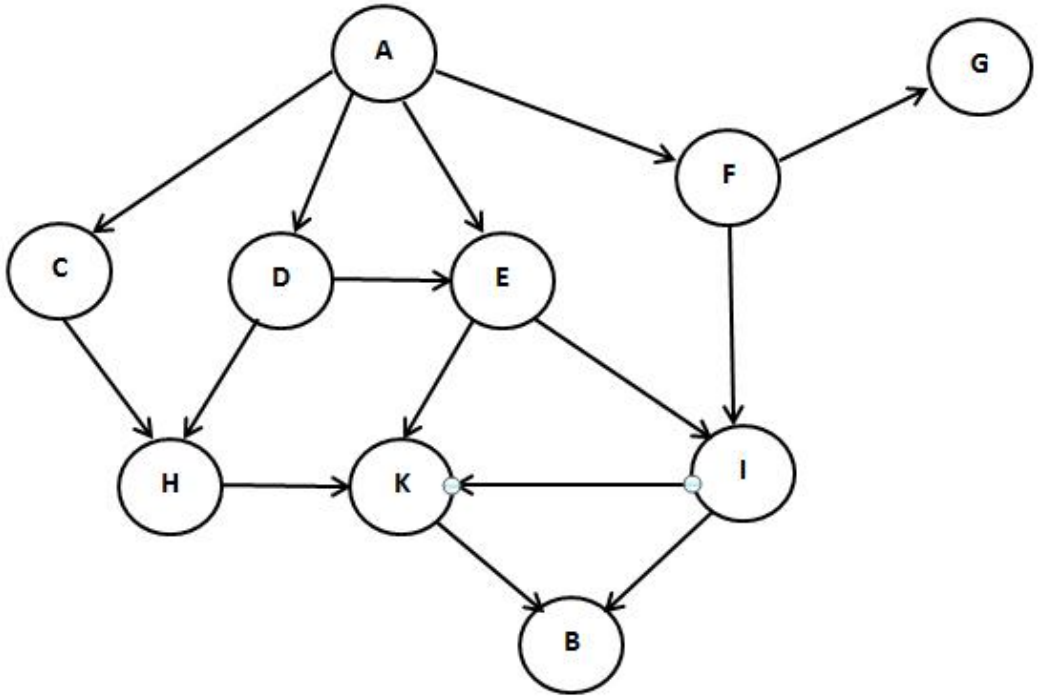


Bài toán cần tìm đường đi từ thành phố A đến thành phố B. Không gian trạng thái cho bài toán này có A là ***trạng thái ban đầu*** và B là ***trạng thái đích***

Biểu diễn đồ thị không gian trạng thái

Có thể biểu diễn đồ thị này bằng *tập danh sách kề*. Với ngôn ngữ Python, tập danh sách kề này có thể được diễn tả bằng dictionary ADJ.

```
ADJ = {}
ADJ['A'] = ['C', 'D','E','F']
ADJ['B'] = []
ADJ['C'] = ['H']
ADJ['D'] = ['E','H']
ADJ['E'] = ['I','K']
ADJ['F'] = ['G','I']
ADJ['G'] = []
ADJ['H'] = ['K']
ADJ['I'] = ['B','K']
ADJ['K'] = ['B']
```



Phương pháp Tìm Kiếm trên đồ thị không gian trạng thái

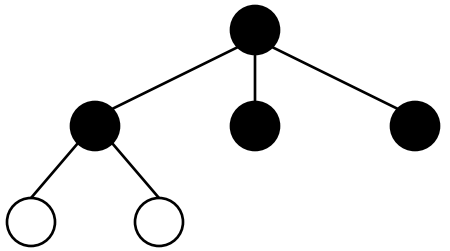
1. Tìm kiếm theo chiều sâu trước
2. Tìm kiếm theo bề rộng trước
3. Tìm kiếm tốt nhất trước
4. Uniform cost search
5. Giải thuật A*

Phương pháp 1 và 2 được xếp loại *tìm kiếm mù* (uninformed search) và ba phương pháp còn lại được gọi là *tìm kiếm có thông tin* (informed search) hay *tìm kiếm heuristic* (heuristic search)

Phương pháp tìm kiếm mù

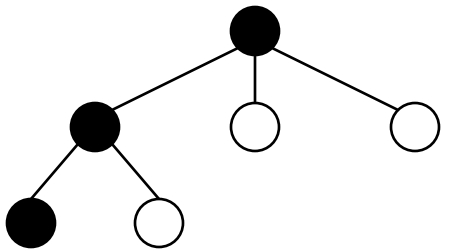
- Tìm kiếm theo chiều rộng trước

  Triển khai tất cả các nút
  của một mức (level) trước.



- Tìm kiếm theo chiều sâu trước

  Triển khai một trong những nút
  tại mức sâu nhất.



2. Tìm kiếm chiều sâu trước

Mã giả của giải thuật *depth-first* search như sau:

1. Đưa nút ban đầu vào list OPEN và nút này được gán một con trỏ rỗng.

2. Nếu OPEN là list rỗng, in ra thông báo “FAILURE” và dừng.

3. Chọn nút đầu tiên trong OPEN và gán nó vào biến N. Xóa nút này khỏi list OPEN và đưa nó vào list CLOSED. Nếu N là nút đích, xuất danh sách kết quả dựa vào chuỗi con trỏ khởi đầu bằng con trỏ liên kết với nút N.

4. Sinh ra một danh sách L gồm các nút kế cận của nút N và xóa khỏi L những nút nào từng xuất hiện trong danh sách CLOSED.

5. Xóa bỏ khỏi OPEN những nút nào đồng thời xuất hiện trong L. Ghép kề danh sách L vào phần đầu của OPEN và mỗi nút trong L được gắn một con trỏ đến N.

6. Quay về bước 2.

DFS áp dụng vào đồ thị thí dụ

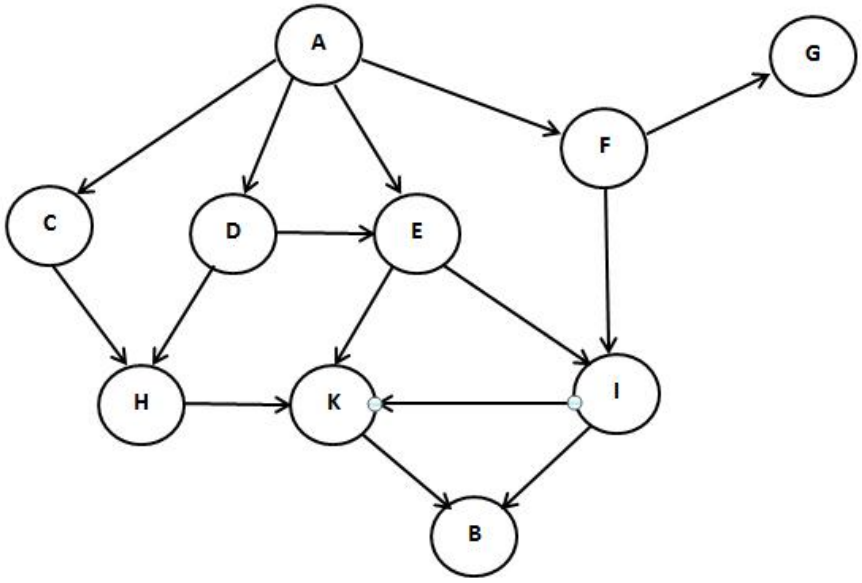
1. Khởi tạo: OPEN = [‘A’], CLOSED = []
2. OPEN = [‘C’, ‘D’, ‘E’, ‘F’], CLOSED = [‘A’]
3. OPEN = [‘H’, ‘D’, ‘E’, ‘F’], CLOSED = [‘A’, ‘C’]
4. OPEN = [‘K’, ‘D’, ‘E’, ‘F’], CLOSED = [‘A’, ‘C’, ‘H’]
5. OPEN = [‘B’, ‘D’, ‘E’, ‘F’], CLOSED = [‘A’, ‘C’, ‘H’, ‘K’]

Lối đi tìm thấy từ nút ban đầu đến nút đích:

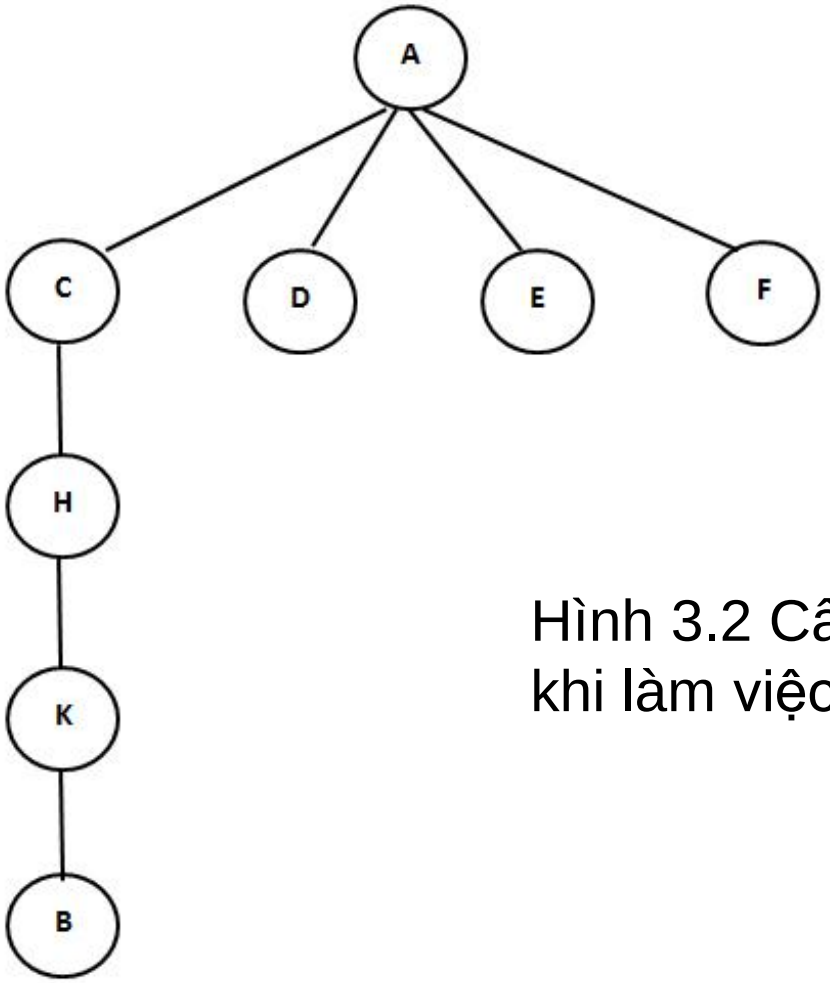
[’A’, ‘C’, ‘H’ , ‘K’, ‘B’]

COUNT = 8

Tổng số nút được xét là 8.



Cây tìm kiếm (search tree) của DFS trên đồ thị thí dụ



Hình 3.2 Cây tìm kiếm của DFS khi làm việc trên đồ thị thí dụ

Module hiện thực giải thuật tìm kiếm theo chiều sâu trước bằng Python được cho trong slide kế tiếp.

```
PREDECESSOR = {}                    # PREDECESSOR  là một tự điển
COUNT = 0
def dfgs(start, goal):
    OPEN = [start]                  # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    while OPEN != []:               # Step 2
        n = OPEN[0]                 # Step 3
        OPEN = OPEN[1:]
        CLOSED.append(n)
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)
        lst = successors(n)         # Step 4
        lst = listDifference(lst, CLOSED)
    # successors are added to the beginning of OPEN.
    # If any successor was already on OPEN, move it up.
        OPEN = lst + listDifference(OPEN, lst)          # Step 5
        for elt in lst:
            PREDECESSOR[elt] = n
    # end of loop. (Step 6)
```

Các hàm hỗ trợ

```
def extractPath(n):
    if PREDECESSOR[n]==None:
        return [n]
    return extractPath(PREDECESSOR[n]) + [n]

def successors(n):
     return ADJ[n]

def listDifference(list1, list2):
# Trả về một bản sao của  list1 trong đó các phần tử có xuất hiện trong list2 bị loại bỏ.
   list3 = list1[:]
   for elt in list2:
     if elt in list3
        list3.remove(elt)          # remove là một phương thức của list
return list3
```

Hàm test

Hàm *test* được dùng để chạy thử chương trình.

```
def test():
    start = ’A’
    goal = ’B’
    print("Starting a depth-first graph search from " + start)
    path = dfgs(start, goal)
    print(path)
    global COUNT
    print(str(COUNT) + " nodes expanded during the search.")
```

Lối đi tìm thấy từ nút ban đầu đến nút đích:

[’A’, ‘C’, ‘H’ , ‘K’, ‘B’]

COUNT = 8

Tổng số nút được xét là 8.

3. Tìm kiếm chiều rộng trước

Với một điều chỉnh nhỏ trên hàm dfgs, ta có thể chuyển nó thành hàm breadthfgs để thực hiện *tìm kiếm theo chiều rộng trước*. Tại các bước 4 và 5, ta thay đổi cách mà list L được gộp vào OPEN. Thay vì ghép L đứng trước list OPEN, ta ghép list L vào phía sau list OPEM. Tức là ta thay câu lệnh

OPEN = lst + listDifference(OPEN, lst)

bằng câu lệnh:

OPEN = OPEN + listDifference(lst, OPEN)

Và như vậy những nút mới sinh ra được đưa vào cuối list OPEN.

Mã Python của giải thuật BFS được cho ở slide kế tiếp.

```
def breadthfgs(start, goal):
    OPEN = [start]                # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    while OPEN != []:             # Step 2
        n = OPEN[0]               # Step 3
        OPEN = OPEN[1:]
        CLOSED.append(n)
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)
        lst = successors(n)               # Step 4
        lst = listDifference(lst, CLOSED)
         lst = listDifference(lst, OPEN)
       # the new successors are added to the end of OPEN:
        OPEN = OPEN + lst                 # Step 5
        for elt in lst:
            PREDECESSOR[elt] = n
# end of loop. (Step 6)
```

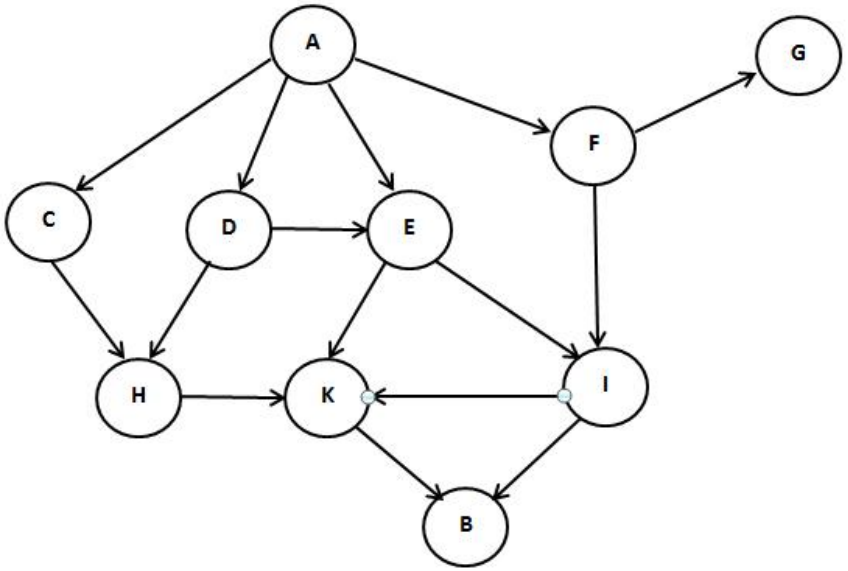
BFS áp dụng vào đồ thị thí dụ

1. Khởi tạo: OPEN = [‘A’], CLOSED = []
2. OPEN = [‘C’, ‘D’, ‘E’, ‘F’], CLOSED = [‘A’]
3. OPEN = [‘D’, ‘E’, ‘F’, ‘H’], CLOSED = [‘A’, ‘C’]
4. OPEN = [‘E’, ‘F’, ‘H”], CLOSED = [‘A’, ‘C’, ‘D’]
5. OPEN = [‘F’ , ‘H’, ‘K’, ‘I’], CLOSED = [‘A’, ‘C’, ‘D’, ‘E’]
6. OPEN = [‘H’, ‘K’, ‘I’, ‘G’], CLOSED = [‘A’, ‘C’, ‘D’, ‘E’ , ‘F’]
7. OPEN = [‘K’, ‘I’, ‘G’], CLOSED = [‘A’, ‘C’, ‘D’, ‘E’ , ‘F’, ‘H’]
8. OPEN = [‘I’, ‘G’, ‘B’], CLOSED = [‘A’, ‘C’, ‘D’, ‘E’ , ‘F’, ‘H’, ‘K’]
9. OPEN = [‘G’, ‘B’], CLOSED = [‘A’, ‘C’, ‘D’, ‘E’ , ‘F’, ‘H’, ‘K’, ‘I’]
10. OPEN = [‘B’], CLOSED = [‘A’, ‘C’, ‘D’, ‘E’ , ‘F’, ‘H’, ‘K’, ‘I’, ‘G’]

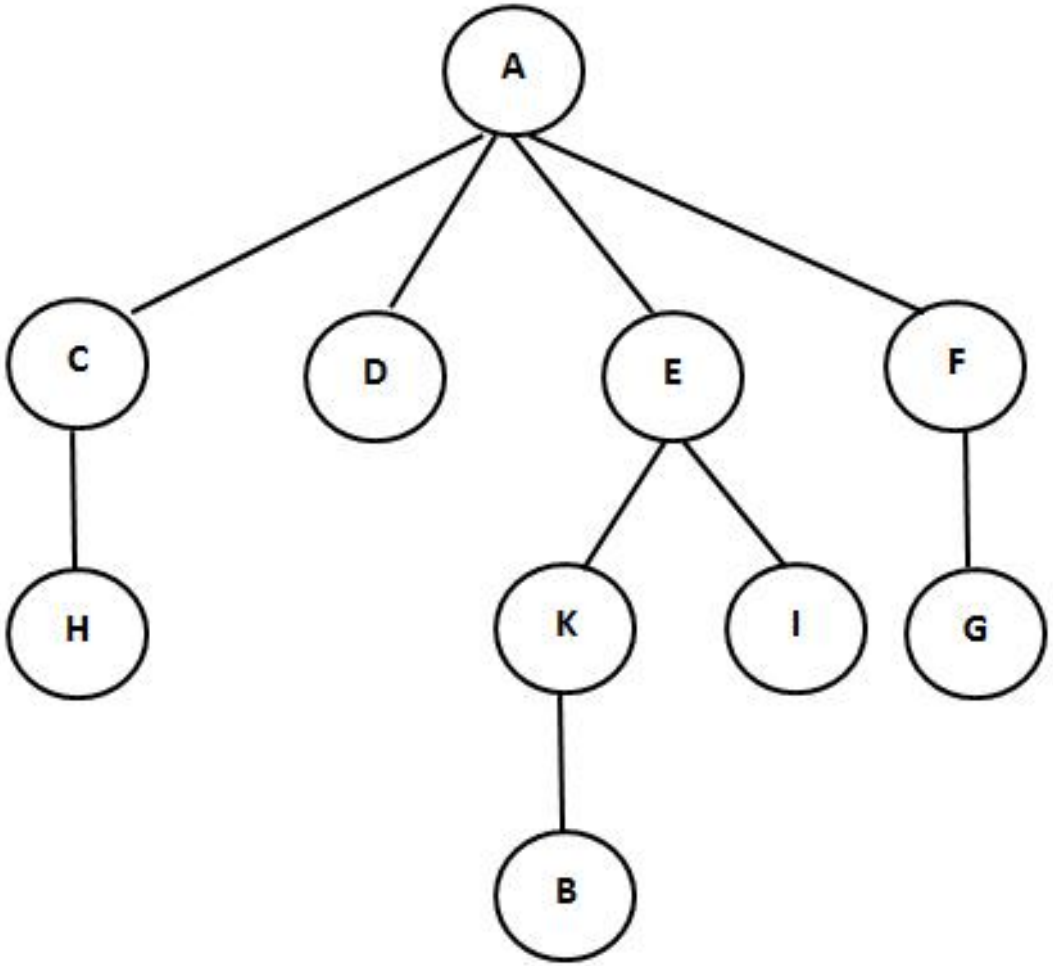
Lối đi tìm thấy từ nút ban đầu đến nút đích
[’A’, ‘E’, ‘K’, ‘B’]

COUNT = 10

Tổng số nút được xét là 10.



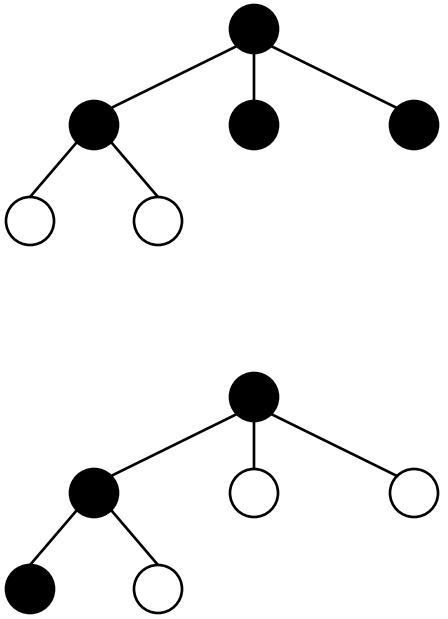
Cây tìm kiếm của BFS trên đồ thị thí dụ



Hình 3.3 Cây tìm kiếm của BFS khi làm việc trên đồ thị thí dụ

So sánh DFS với BFS

Tiêu chí	Breadth-First	Depth-First
Thời gian	b^d	b^m
Chỗ bộ nhớ	b^d	bm
Có tối ưu?	Yes	No
Có đầy đủ?	Yes	No



b: branching factor d: solution depth m: maximum depth

4. Tìm kiếm tốt nhất trước (Best-First Search)

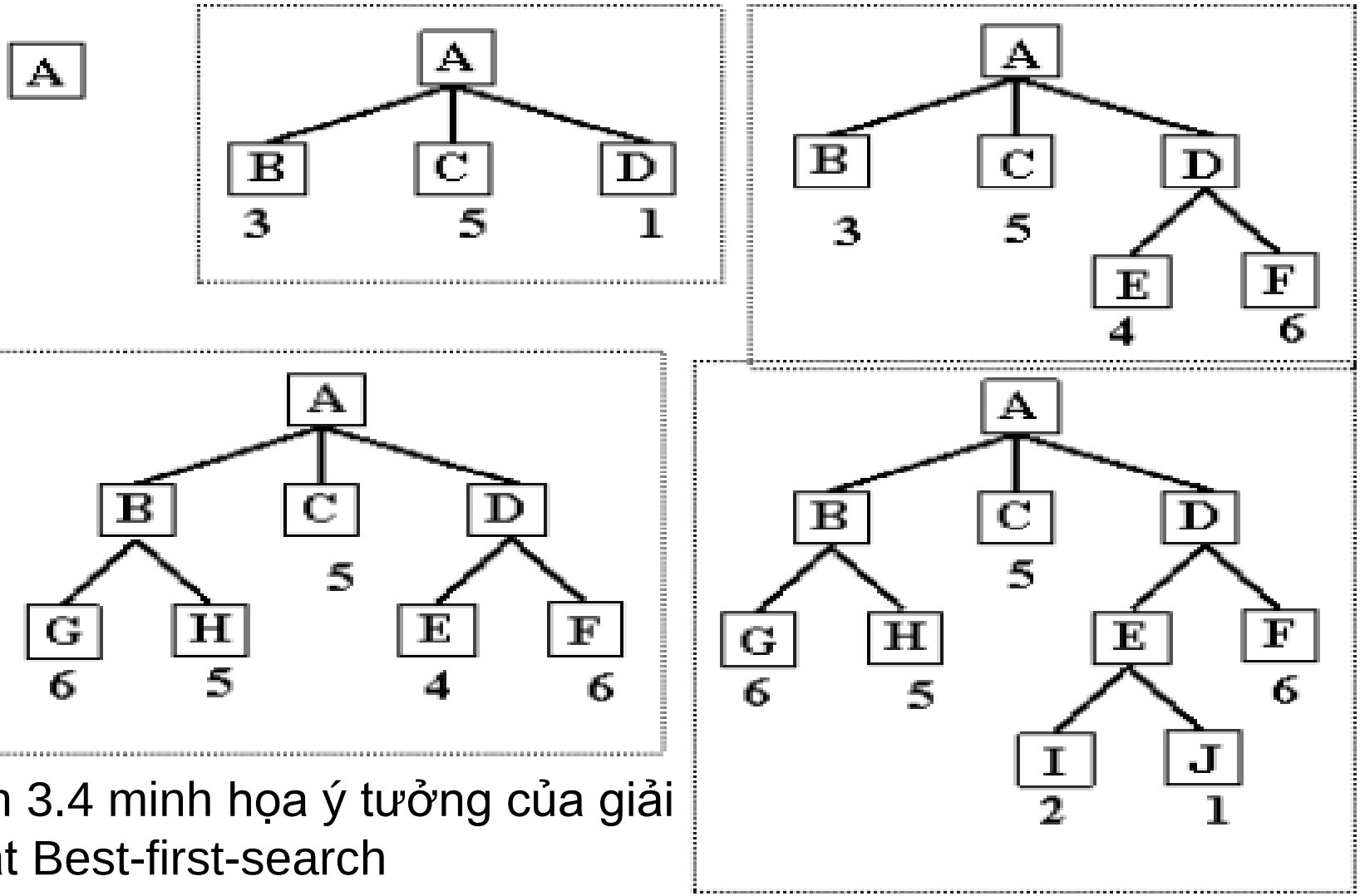
- Tìm kiếm chiều sâu trước và Tìm kiếm chiều rộng trước được gọi là tìm kiếm mù (uninformed search)
- Tìm kiếm tốt nhất trước: kết hợp tìm kiếm chiều sâu + tìm kiếm chiều rộng và ý tưởng

→ Mở rộng sự tìm kiếm theo các nút có ***nhiều tiềm năng*** chứa trạng thái đích hơn các nút khác.

Hàm đánh giá (Evaluation Function)

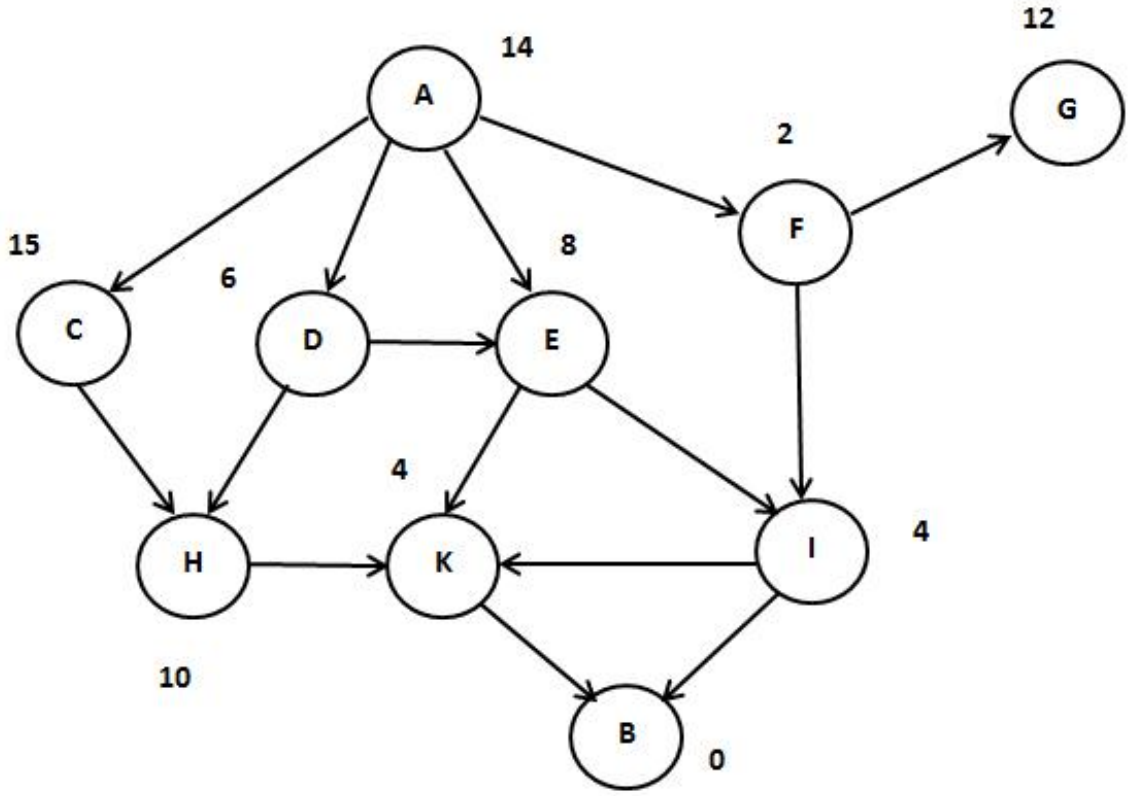
- Một hàm*f* có thể gán một nút trong không gian trạng thái đến một số thực để phục vụ cho việc ước lượng độ tốt hoặc chi phí gây ra nếu nút đó được chọn để đi tiếp quá trình tìm kiếm. Hàm *f* như vậy được gọi là *hàm đánh giá*.

Ví dụ minh họa Best-first search



Hình 3.4 minh họa ý tưởng của giải thuật Best-first-search

Cho một đồ thị như trong Hình 3.5 mà tại mỗi nút có kèm chi phí ước lượng để đi từ nút đó đến nút đích (B)



Hình 3.5

Trước tiên ta lập một tự điển FN để lưu trị ước lượng được dùng để tính hàm đánh giá *f*:

FN = {'A':14, 'B':0, 'C': 15, 'D': 6,
'E': 8, 'F':2, 'G':12, 'H':10,
'I':4, 'K':4}

Mã giả của Best-first-search

1. Đặt nút khởi đầu S vào list OPEN, tính *f*(S), và gắn trị hàm này với S. Gắn một con trỏ null với S.

2. Nếu list OPEN rỗng, in ra thông báo “FAILURE” và dừng.

3. Chọn một nút N từ list OPEN sao cho *f*(N) ≤ *f*(M) với mọi nút M trong OPEN

4. Đặt nút N vào list CLOSED và xóa bỏ N ra khỏi list OPEN.

5. Nếu N là nút đích, trả về lối đi tìm thấy từ nút S đến nút N dựa vào chuỗi con trỏ khởi đầu bằng con trỏ liên kết với nút N. Và dừng giải thuật.

6. Với mỗi nút kế cận J, nếu đã hiện diện trong OPEN, tính lại *f*(J) và so sánh trị này với trị *f* trước đó của J. Nếu trị f mới nhỏ hơn thì gắn trị này với nút J và đặt nó vào OPEN cho đúng vị trí, và gắn nó với con trỏ chỉ từ J ngược về N.

7. Với mỗi nút kế cận J của N mà không hề hiện diện trong OPEN hay CLOSED:

(a) tính *f*(J) và liên kết trị này với J.

(b) Đặt J vào OPEN.

(c) gắn J với con trỏ chỉ từ J ngược về N.

8. Quay lại Step 2.

```
PREDECESSOR = {}
COUNT = 0
def bestfs(start, goal):
#Performs a best-first search from start for goal.
    OPEN = [start]                          # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    while OPEN != []:                                # Step 2
        n = deleteMin(OPEN, FN)                              # Step 3
        OPEN = OPEN[1:]                             # Step 4
        CLOSED.append(n)
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)                    # Step 5
        lst = successors(n)                         # Step 6
        lst = listDifference(lst, CLOSED)
      # successors are added to OPEN.
        OPEN = lst +listDifference(OPEN, lst)    # Step 7
        for elt in lst:
            PREDECESSOR[elt] = n
# end of loop. This is implicitly Step 8
```

Hàm deleteMin

- Hàm *deleteMin* để xác định phần tử "tốt nhất" dựa vào một hàm đánh giá. Hàm sẽ xóa bỏ phần tử này ra khỏi danh sách và trả nó về trình gọi.

```
def deleteMin(lst, fn):
    minVal = 9999            # một giá trị cực lơn
    minElt = None
    for e in lst:
        temp = fn[e]         # fn là một tự điển
        if temp < minVal: minVal = temp; minElt = e
    lst.remove(minElt)
return minElt
```

Hàm test

Để thử hàm bestfs, chúng ta định nghĩa hàm test như sau.

```
def test():
    start = ’A’
    goal = ’B’
    print ("Starting a depth-first graph search from " + start)
    path = bestfs(start, goal)
    print(path)
    global COUNT
    print(str(COUNT) + " nodes expanded during the search.")
```

Best First Search áp dụng vào đồ thị thí dụ

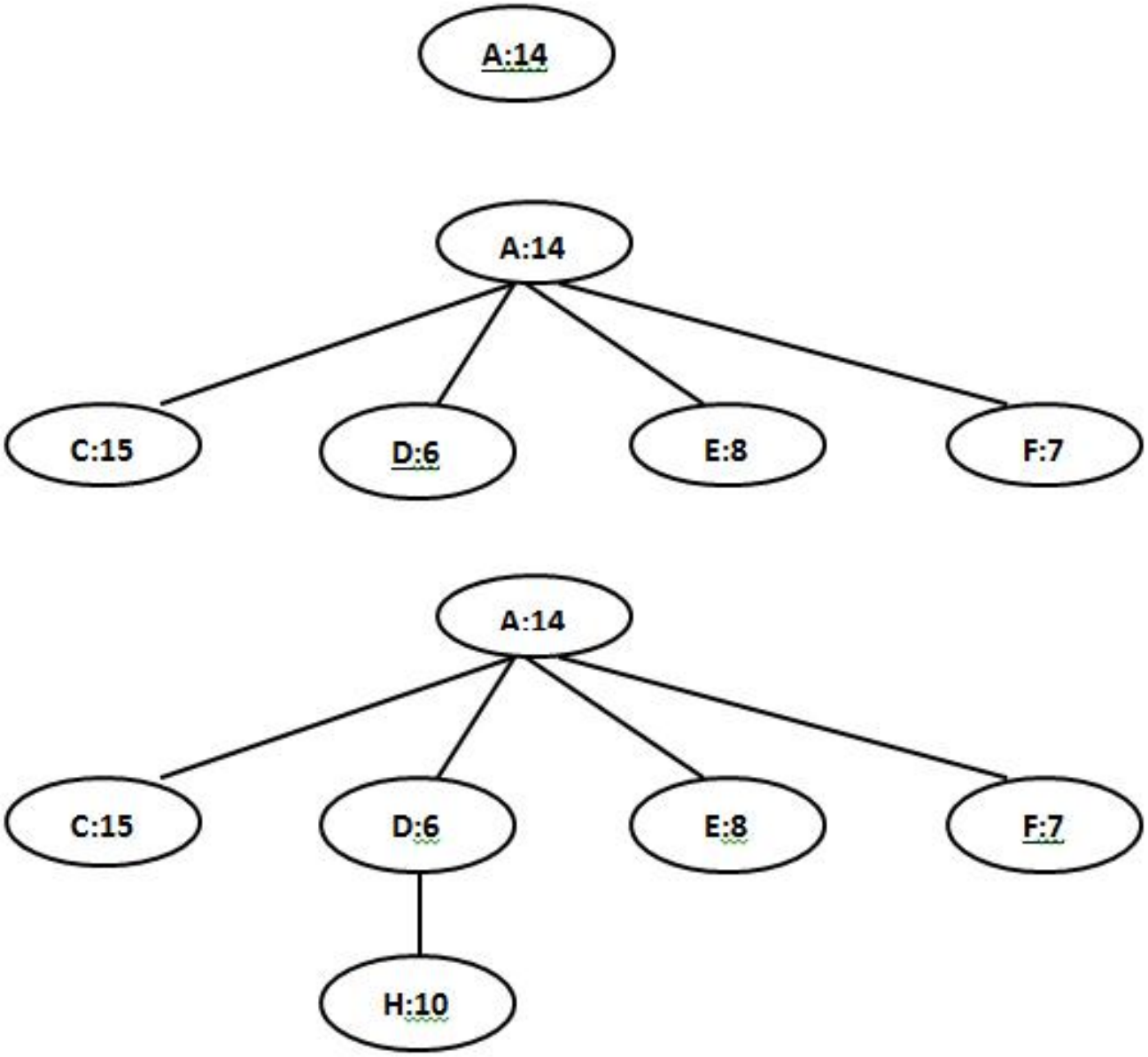
1. Khởi tạo: OPEN = [‘A’:14], CLOSED = []
2. OPEN = [‘D’:6, ‘F’:7, ‘E’:8, ‘C’:15], CLOSED = [‘A’]
3. OPEN = [‘F’:7, ‘E’:8, ‘H’:10, ‘C’:15], CLOSED = [‘A’, ‘D’]
4. OPEN = [‘I’:4, E’:8, ‘H’:10, ‘G’:12, ‘C’:15], CLOSED = [‘A’, ‘D’, ‘F’]
5. OPEN = [‘B’:0, ‘K’:2, ‘E’:8, ‘H’:10, ‘G’:12, ‘C’:15], CLOSED = [‘A’, ‘D’, ‘F’, ‘I’]

Running the *test* function gives us the path:

[’A’, ‘F’, ‘I’, ‘B’]

COUNT = 10 (Tổng số nút được xét là 10).

Các bước của best-first-search trên đồ thị thí dụ



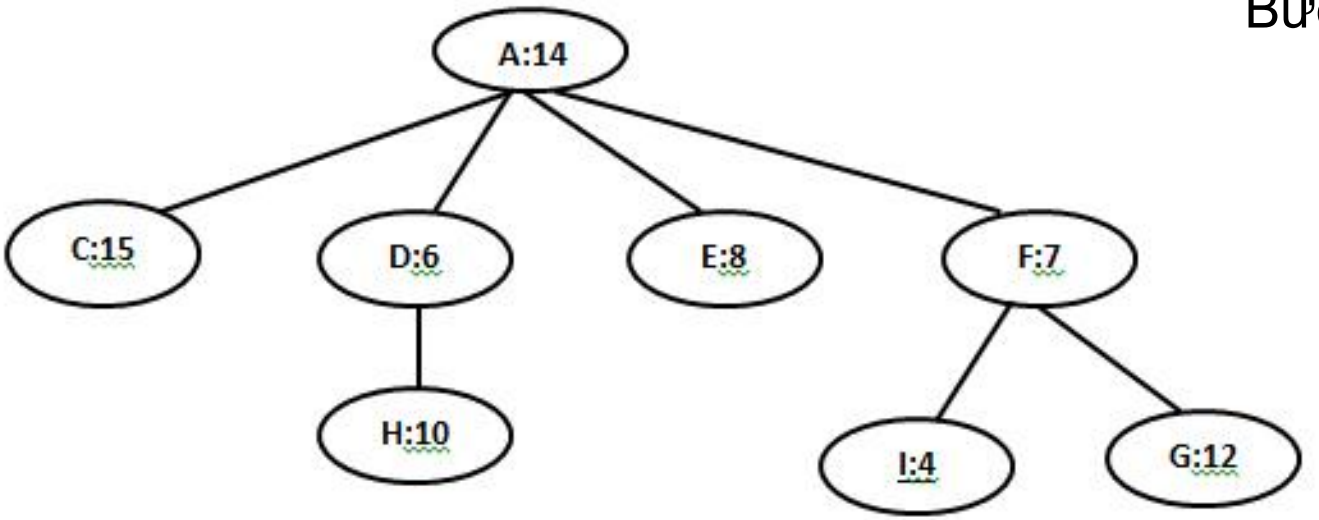
Bước 1

Bước 2

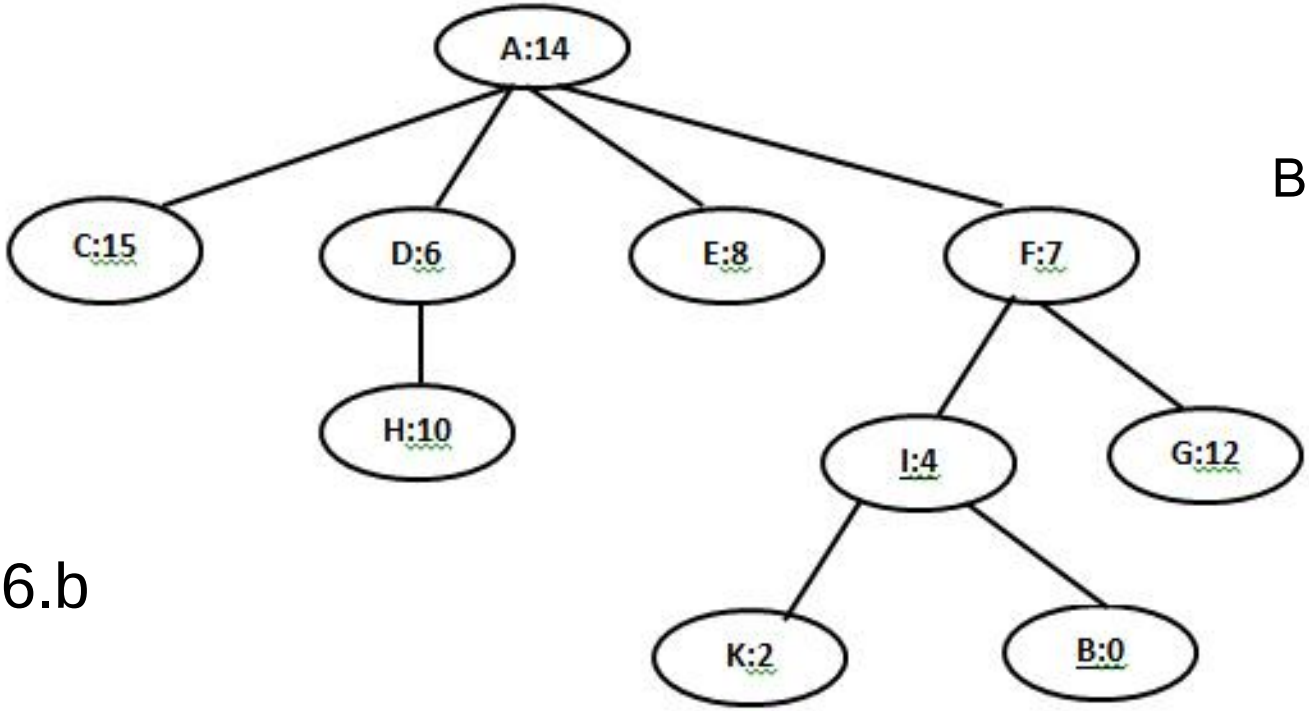
Bước 3

Hình 3.6.a

Bước 4



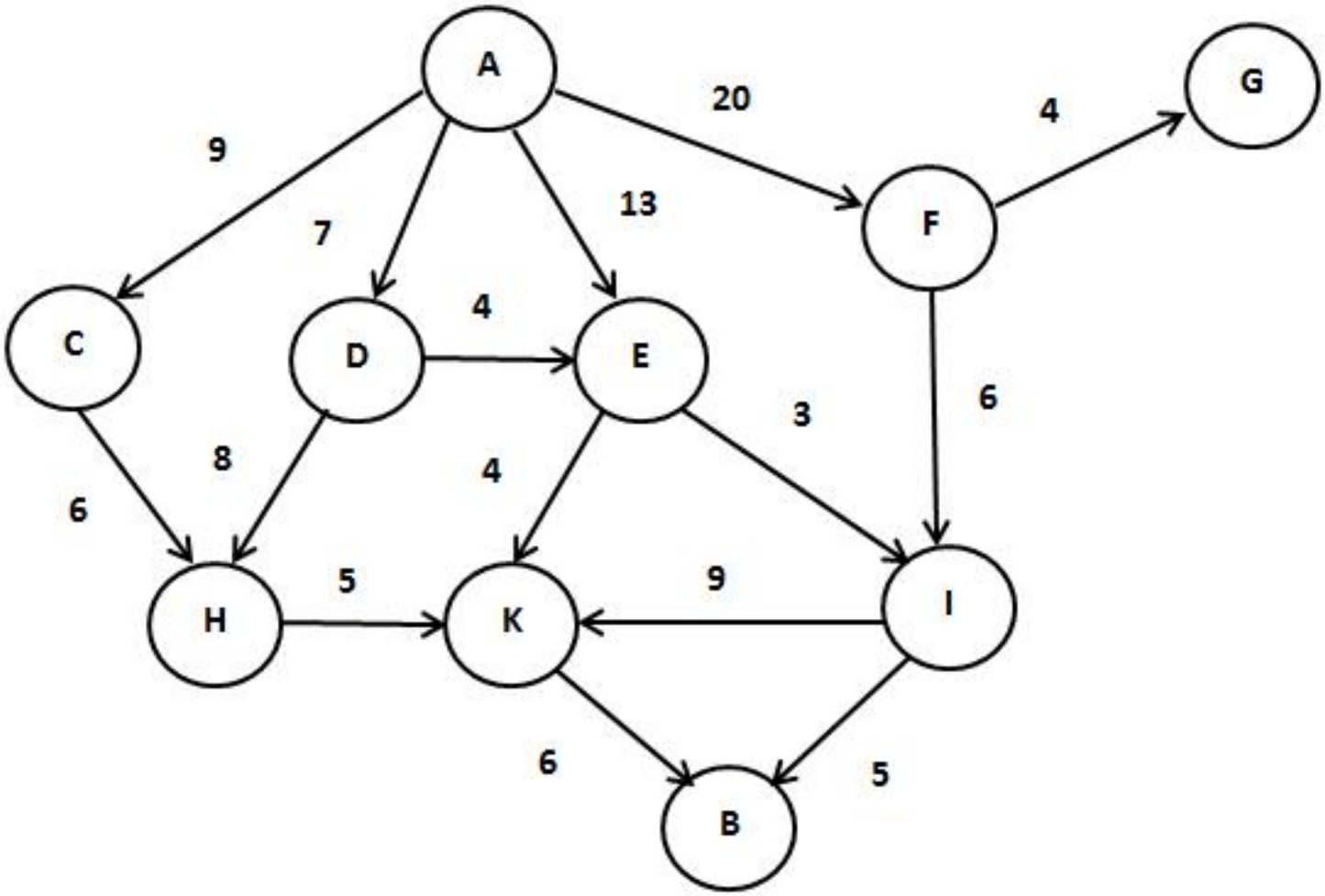
Bước 5



Hình 3.6.b

5. Giải thuật Uniform Cost Search

- Giả sử chúng ta có một bản đồ trong đó có ghi khoảng cách trên cạnh nối giữa hai thành phố có đường nối. Và chúng ta muốn tìm lối đi ngắn nhất từ một thành phố khởi đầu đến một thành phố đích. Những khoảng cách có ghi trên bản đồ cung cấp một cơ sở để đánh giá lộ trình tốt nhất muốn tìm.
- Một giải thuật có sử dụng thông tin như vậy để tìm lộ trình ngắn nhất được gọi là giải thuật ***uniform cost search***.
- Thêm thông tin khoảng cách, ta có được một đồ thị có trọng số như trong hình vẽ sau đây.



Hình 3.7 Một đồ thị có ghi trị khoảng cách trên các cạnh

Biểu diễn đồ thị có trọng số

- Ta có thể biểu diễn đồ thị có trọng số bằng một tự điển có tên DISTANCE. Đây là một tự điển mà mỗi phần tử của nó lại chứa đựng một tự điển. Có một tự điển được dùng để chứa những khoảng cách từ một thành phố đến một số thành phố khác.

Các khoảng cách được ghi như sau trong tự điển:

```
DISTANCE = {}
DISTANCE['A'] = {'C':9,'D':7,'E':13,'F':20}
DISTANCE['C'] = {'H':6}
DISTANCE['D'] = {'E':4,'H':8,}
DISTANCE['E'] = {'I':3,'K':4}
DISTANCE['F'] = {'G':4,'I':6}
DISTANCE['G'] = {}
DISTANCE['H'] = {'K':5}
DISTANCE['I'] = {'B':5,'K':9}
```

Tác vụ then chốt trong Uniform cost search

Để có thể áp dụng breadth-first search trong giải thuật uniform cost search, ta điều chỉnh để nó triển những nút mới mà có tổng khoảng cách ngắn nhất từ nút khởi đầu đến những nút đó. Như vậy mỗi khi tạo ra nút kế cận M của nút N, ta nên thực hiện

(1) kiểm tra xem M có hiện diện trong list CLOSED, nếu có thì không xét nó,

(2) tính tổng khoảng cách lối đi từ nút M đến nút khởi đầu như sau

Temp = NodeDistance(N) + ArcDistance(M, N);

(3) kiểm tra xem M có hiện diện trong list OPEN, nếu có, ta nên so sánh giá trị hiện hành của NodeDistance(M) với Temp và nếu Temp nhỏ hơn thì xóa bỏ sự hiện diện trước đó của M trong OPEN; và

(4) cho NodeDistance(M) = Temp, và đưa M vào vị trí thích hợp trong list OPEN theo giá trị tăng dần của NodeDistance.

Mã Python của giải thuật Uniform Cost Search

```
def unifcost(start, goal):
    OPEN = [start]      # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    FVALUE[start] = 0
    while OPEN != []:          # Step 2
        n = deleteMin(OPEN, FVALUE)          # Step 3
        CLOSED.append(n)                     # Step 4
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)     # Step 5
        lst = successors(n)           # Step 6
        lst = listDifference(lst, CLOSED)
```

```
        for elt in lst:                 # Step 7
            temp = f(elt, n)
            if elt in OPEN:
                if temp < FVALUE[elt]:
                    FVALUE[elt] = temp
                    PREDECESSOR[elt] = n
                    OPEN.remove(elt)
                    insert(elt, OPEN)
            else:
                if not elt in CLOSED:
                     FVALUE[elt] = temp
                    PREDECESSOR[elt] = n
                    OPEN = insert(elt, OPEN)
# end of loop. (Step 8)
```

Các hàm hỗ trợ

```
# hàm f tính khoảng cách từ nút khởi đầu đến nút m
def f(m, n):
return FVALUE[n] + DISTANCE[m][n]      # DISTANCE là một tự điển

def deleteMin(lst, fn):
    minVal = 9999
    minElt = None
    for e in lst:
        temp = fn[e]
        if temp < minVal: minVal = temp; minElt = e
    lst.remove(minElt)
return minElt

def insert(elt, lst):
    if lst == []:
       return [elt]
    if FVALUE[elt] < FVALUE[lst[0]]:
        return [elt]+lst
    else:
        return [lst[0]] + insert(elt, lst[1:])
```

Hàm test

Để thử hàm unifcost, chúng ta định nghĩa hàm test như sau.

```
def test():
    start = ’A’
    goal = ’B’
    print( "Starting a depth-first graph search from " + start)
    path = unifcost(start, goal)
    print( path)
    global COUNT
    print(str(COUNT) + " nodes expanded during the search.")
    print(’Total distance is ’ + str(FVALUE[goal]))
```

Uniform Cost Search áp dụng vào đồ thị thí dụ

1. Khởi tạo: OPEN = [‘A’:14], CLOSED = []
2. OPEN = [‘D’:**7**, ‘C’:9, ‘E’:13, ‘F’:20], CLOSED = [‘A’]
3. OPEN = [‘C’:**9**, ‘E’:7+4=11, ‘H’:7+8=15, ‘F’:20], CLOSED = [‘A’, ‘D’]
4. OPEN = [‘E’:7+4=**11**, ‘H’:7+8=15, ‘F’:20], CLOSED = [‘A’, ‘D’, ‘C’]
5. OPEN = [‘J’:11+3=**14**, ‘K’:11+4=15, ‘H’:15, ‘F’:20], CLOSED = [‘A’, ‘D’, ‘C’, ‘E’]
6. OPEN = [‘K’:11+4=**15**, ‘H’:15, ‘B’:14+5=19, ‘F’:20], CLOSED = [‘A’, ‘D’, ‘C’, ‘E’ , ‘J’]
7. OPEN = [‘H’:**15**, ‘B’:14+5=19, ‘F’:20], CLOSED = [‘A’, ‘D’,‘C’, ‘E’, ‘J’, ‘K’]
8. OPEN = [‘B’:14+5=**19,** ‘F’:20], CLOSED = [‘A’, ‘D’,‘C’, ‘E’, ‘J’, ‘K’ , ‘H’]

Chạy hàm *test* function ta đạt được lối đi như sau:

[’A’, ‘D’, ‘E’, ‘J’, ‘B’]

6. Giải thuật A*

- A* là một dạng cải tiến của giải thuật Best-first-search. A* đánh giá các nút bằng cách kết hợp hàm g(n), chi phí để đi từ nút ban đầu đến nút n và hàm h(n) là chi phí để đi từ nút n đến nút đích:

  $$f(n) = g(n) + h(n)$$

- Do h(n) là giá trị ước lượng của lối đi ngắn nhất từ nút n đến nút đích nên ta có:

  f(n) là chi phí ước lượng của lối đi tốn ít chi phí nhất mà đi qua nút n.

- A* có thể đem lại lời giải tối ưu nếu hàm h(n) không ước lượng chi phí ***lớn hơn*** chi phí từ nút n đi đến nút đích.

Mã giả của giải thuật A*

1. Open:={s}; Closed:={};
2. Trong khi (Open khác rỗng)
   3 Chọn đỉnh p tốt nhất trong Open
   4 Nếu p là đỉnh kết thúc thì thoát.
   5 Di chuyển p qua Closed và 6. tạo danh sách các đỉnh q có nối với p
      7.1 Nếu q có trong Open:
         Nếu (**g(q) > g(p) +cost(p,q)**) thì
            g(q) = g(p) +cost(p,q)
            f(q) = g(q) + h(q);
            Nút trước của q là p;
      7.2 Nếu q chưa có trong Open thì
         g(q) = g(p) + cost(p,q);
         f(q) = g(q) + h(q);
         Thêm q vào Open;
         Nút trước q là p;
      7.3 Nếu q có trong Closed thì
         Nếu (**g(q)>g(p)+cost(p,q)**) thì
            Di chuyển q vào Open;
            Nút trước của q là p;
8. Kết thúc vòng lặp.

Mã Python của giải thuật A*

```
def AStar(start, goal):
    OPEN = [start]          # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    GVALUE[start] = 0           # GVALUE là một tự điển
    FVALUE[start] = f(start)     # FVALUE là một tự điển
    global COUNT
    COUNT+=1
    while OPEN != []:                # Step 2
        n = deleteMin(OPEN, f)              # Step 3
        CLOSED.append(n)                # Step 4
        COUNT += 1
        if n == goal:
            return extractPath(n)    # Step 5
        lst = successors(n)           # Step 6
        lst = listDifference(lst, CLOSED)
        for elt in lst:
          if not ((elt in OPEN) or (elt in CLOSED)):
            GVALUE[elt]=g(elt,n)
            FVALUE[elt]=f(elt)
```

```
            OPEN = insert(elt, OPEN)
             PREDECESSOR[elt] = n
          else:
            z = PREDECESSOR[elt]
            if z:
               temp = FVALUE[elt] - GVALUE[z] - arcDist(z,elt) \
                         + GVALUE[n] + arcDist(n,elt)
            else:
                temp = FVALUE[elt]
            if temp < FVALUE[elt]:
                  GVALUE[elt] = GVALUE[elt] - FVALUE[elt] + temp
                  FVALUE[elt] = temp
                  if elt in OPEN:
                       OPEN.remove(elt)
                       OPEN = insert(elt, OPEN)
                  if elt in CLOSED:
                       OPEN = insert(elt, OPEN)
                       CLOSED.remove(elt)
# end of loop.  Step 8
```

Các hàm hỗ trợ

```
def f(n):
    return GVALUE[n] + h[n]

def g(m, n):
    return GVALUE[n] + arcDist(m, n)
```

Hàm sau đây để truy đạt thông tin khoảng cách và nếu thông tin cần tìm không hiện hữu, hàm sẽ trả về một giá trị cực lớn.

```
def arcDist(n1, n2):
   try:
      ad = DISTANCE[n1][n2]
   except KeyError:
      ad = 9999
   return ad
```

h là một tự điển chưa hàm đánh giá h

```
h = {'A':14, 'B':0, 'C': 15, 'D': 6,
        'E': 8, 'F':2, 'G':12, 'H':10,
         'I':4, 'K':4}
```

Hàm test

Để thử hàm Astar, chúng ta định nghĩa hàm test như sau.

```
def test():
    start = ’A’
    goal = ’B’
    print("Starting an A* search from " + start)
    path = AStar(start, goal)
    print(path)
    global COUNT
    print(str(COUNT) + " nodes expanded during the search.")
    print(’Total distance is ’ + str(FVALUE[goal]))
```

Lối đi tìm thấy:

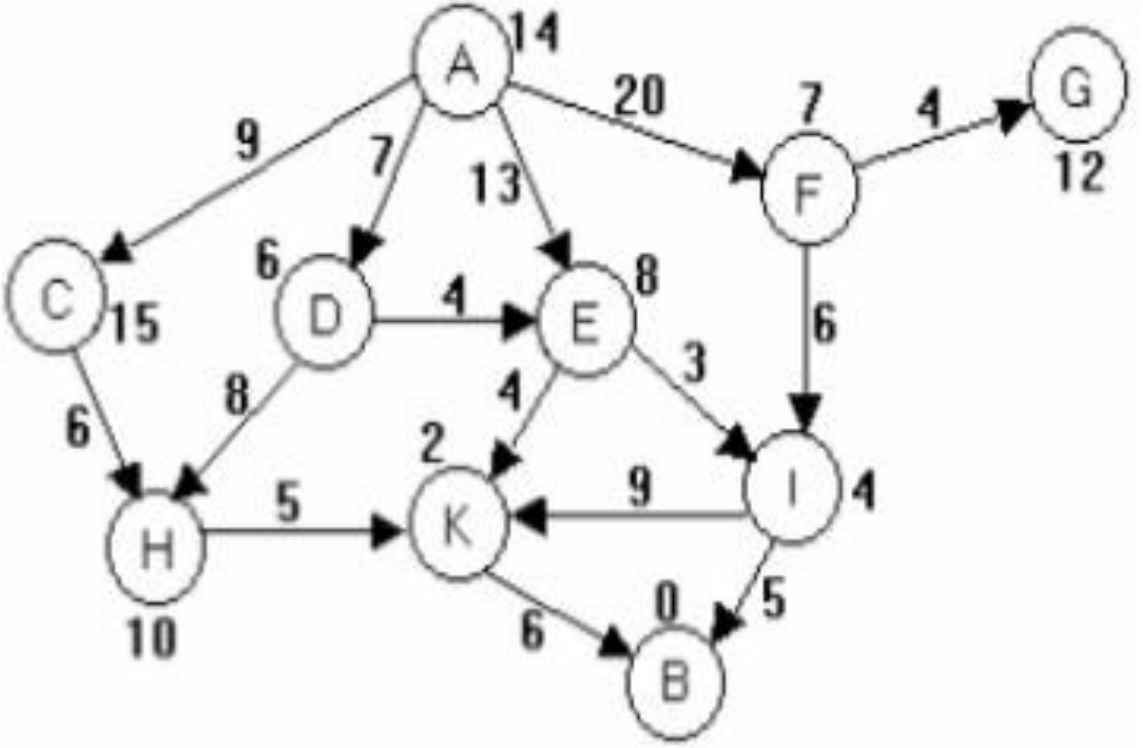
[‘A’, ‘D’, ‘E’, ‘I’, ‘B’]

COUNT = 9

Tổng khoảng cách của lối đi is 19

Thuật giải A* - Ví dụ

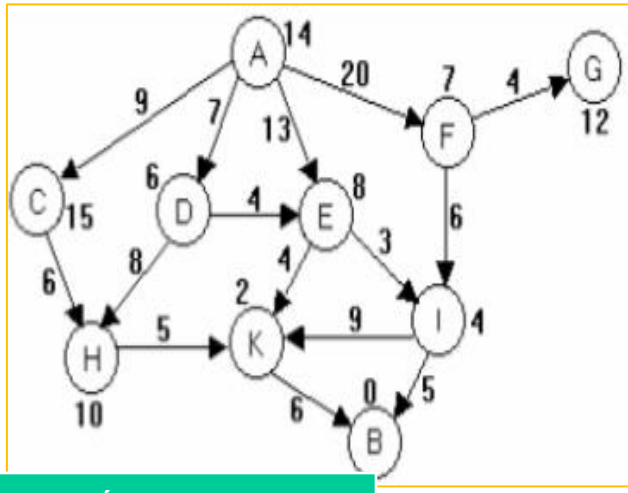
- Cho đồ thị, trạng thái ban đầu A. Tìm đường đi ngắn nhất đến trạng thái đích B



Hình 3.8

- Giá trị tại mỗi đỉnh là giá trị hàm *h* tương ứng
- Giá trị tại mỗi cạnh là độ dài đường đi giữa hai đỉnh

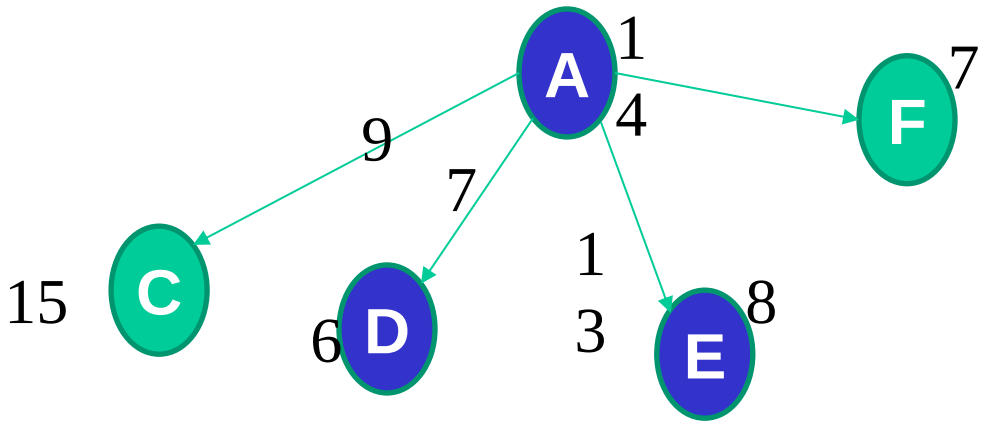
Thuật giải A* - Ví dụ



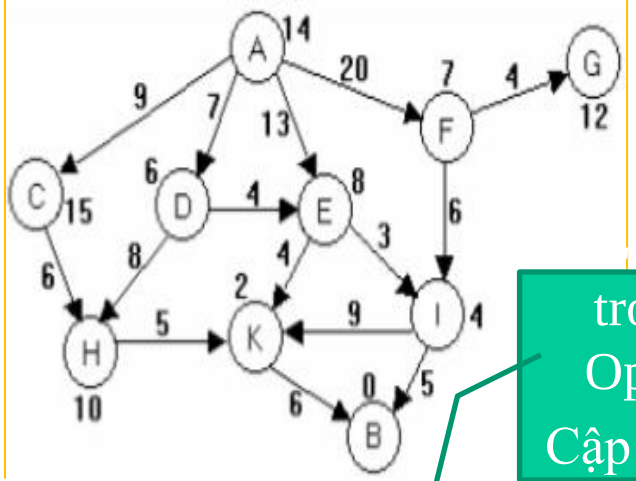
Đỉnh đã có trong Open

Đỉnh đã có trong Closed

Bước	Open	Chọn p	Closed	Các đỉnh q nối với p
1	A			
2	D,E,C,F	A	A	C (g_C=9, h_C=15, f_C=24) D (g_D=7, h_D=6, f_D=13) E (g_E=13, h_E=8, f_E=21) F (g_F=20, h_F=7, f_F=27)



Thuật giải A* - Ví dụ

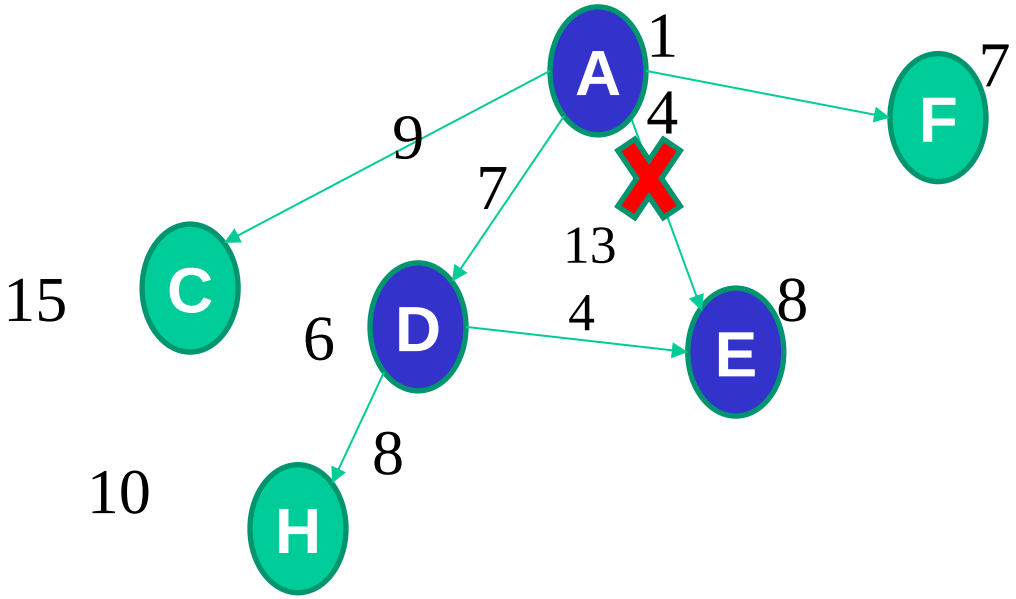


Đỉnh đã có trong Open

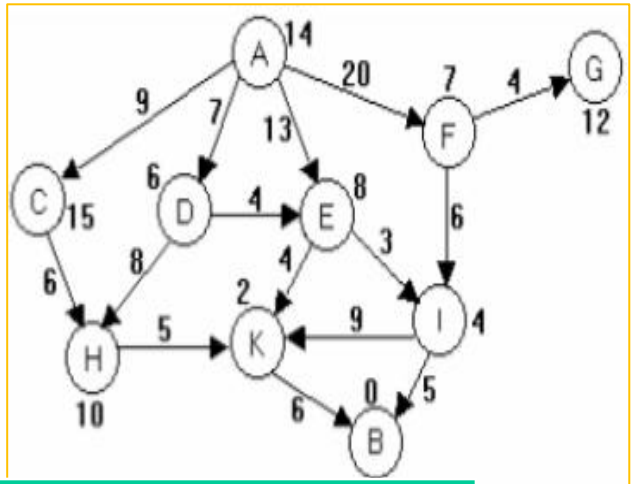
Đỉnh đã có trong Closed

trong Open: Cập nhật

Bước	Open	Chọn p	Closed	Các đỉnh q nối với p
3	E (f=19) C (f=24) H (f=25) F (f=27)	D	A, D	E (g_E=g_D+kc(D,E)=7+4, h_E=8, f_E=19) H (g_H=g_D+kc(D,H)=7+8, h_H=10, f_H=25)



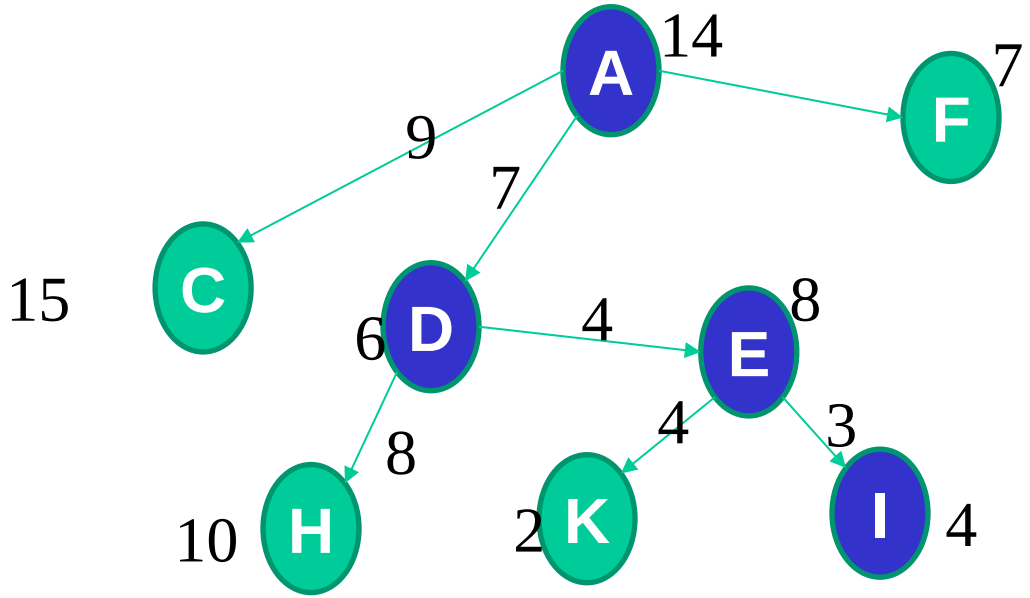
Thuật giải A* - Ví dụ



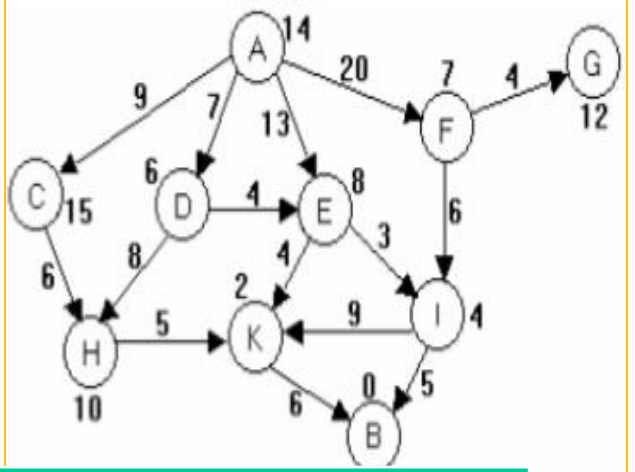
Đỉnh đã có trong Open

Đỉnh đã có trong Closed

Bước	Open	Chọn p	Closed	Các đỉnh q nối với p
4	K, I, C, H, F	E	A, D, E	I ($g_I=g_E+kc(E,I)=11+3$, $h_I=4$, $f_I=18$) K ($g_K=g_E+kc(E,K)=11+4$, $h_K=2$, $f_K=17$)



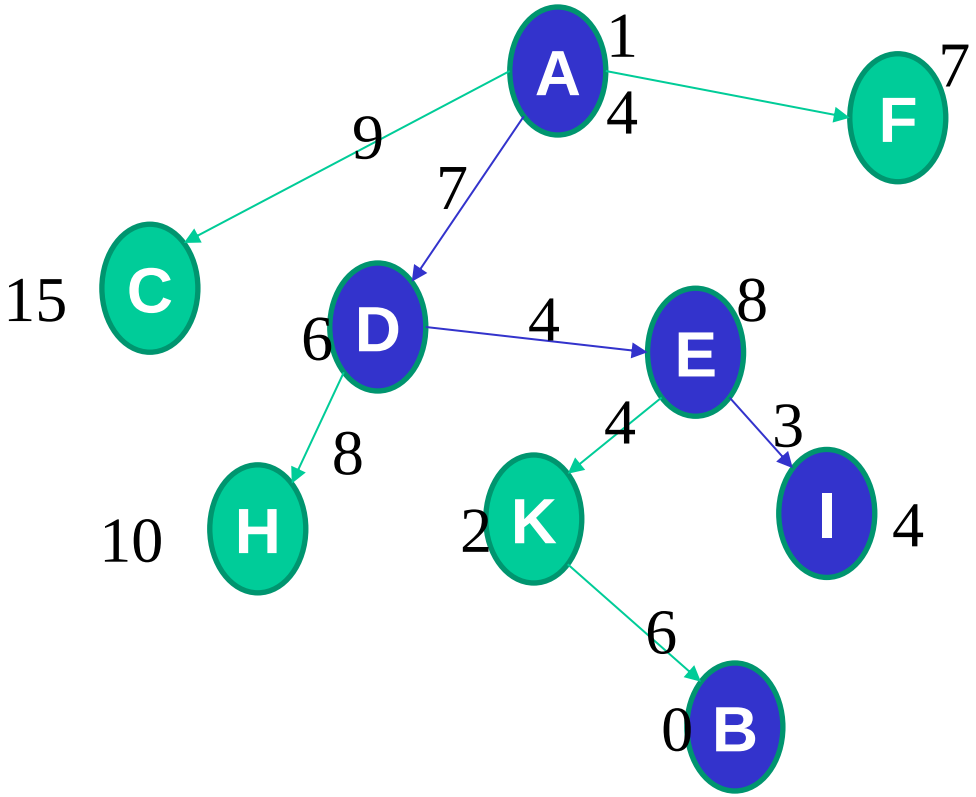
Thuật giải A* - Ví dụ



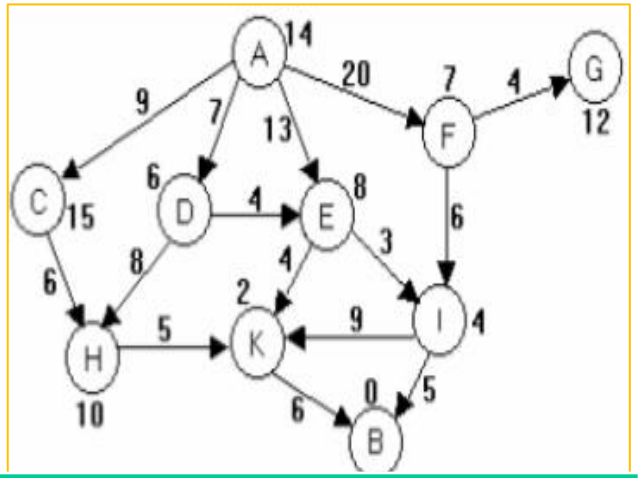
Đỉnh đã có trong Open

Đỉnh đã có trong Closed

Bước	Open	Chọn p	Closed	Các đỉnh q nối với p
5	I, B, C, H, F	K	A, D, E, K	B ($g_B=g_K+kc(K,B)=15+6$, $h_B=0$, $f_B=21$)



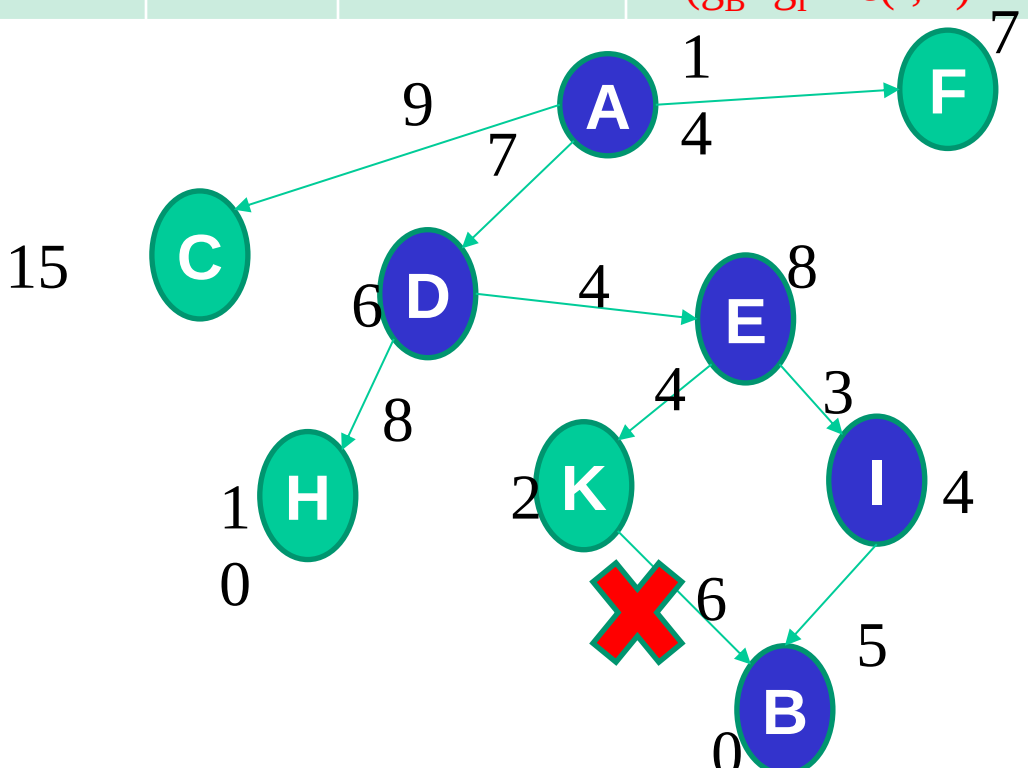
Thuật giải A* - Ví dụ



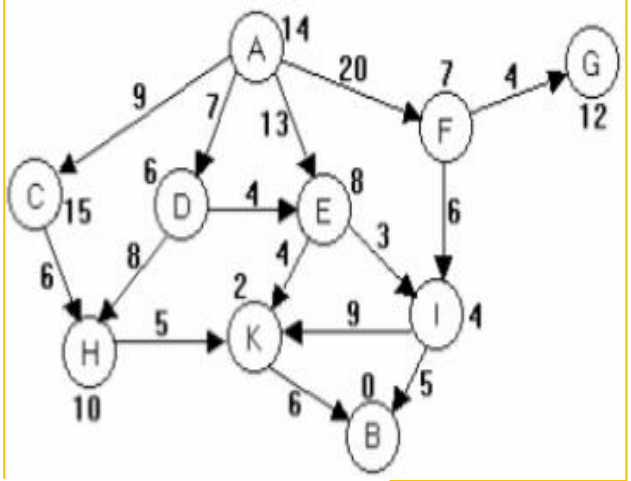
Đỉnh đã có trong Open

Đỉnh đã có trong Closed

Bước	Open	Chọn p	Closed	Các đỉnh q nối với p
6	B, C, H, F	I	A, D, E, K, I	K (g_K<g_I+kc(I,K)=14+9)→*Giữ K trong Closed* B (g_B=g_I+kc(I,B)=14+5, h_B=0, f_B=19)



Thuật giải A* - Ví dụ

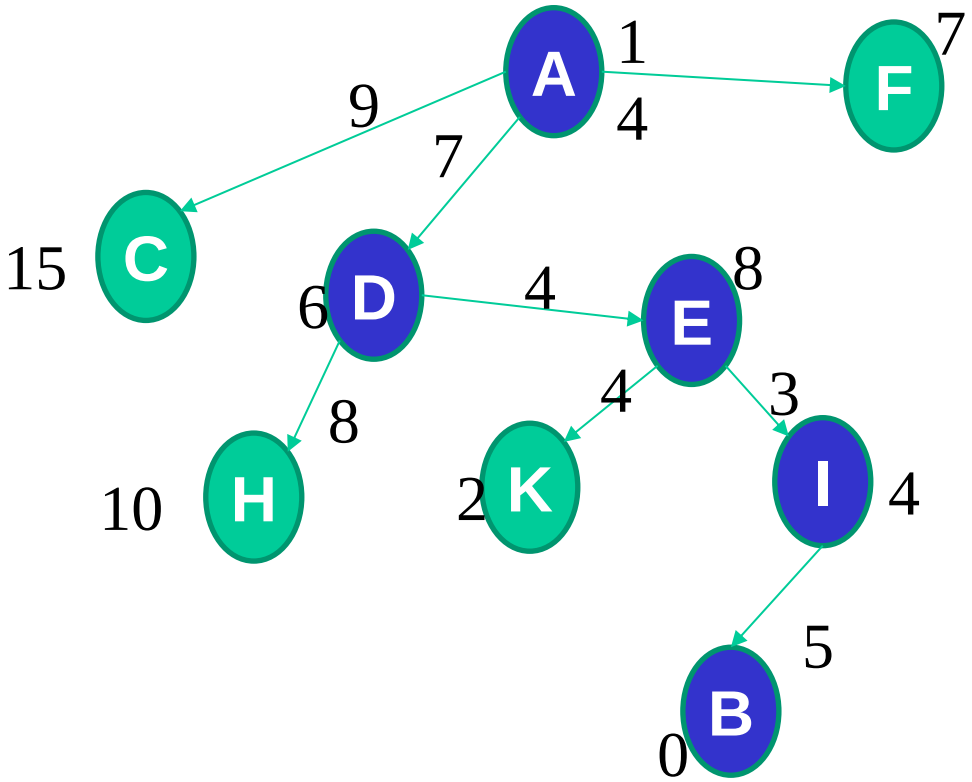


Đỉnh đã có trong Open

Đỉnh đã có trong Closed

Bước	Open	Chọn p	Closed	Các đỉnh q nối với p
7	C, H, F	B	A, D, E, K, I, B	

Kết thúc



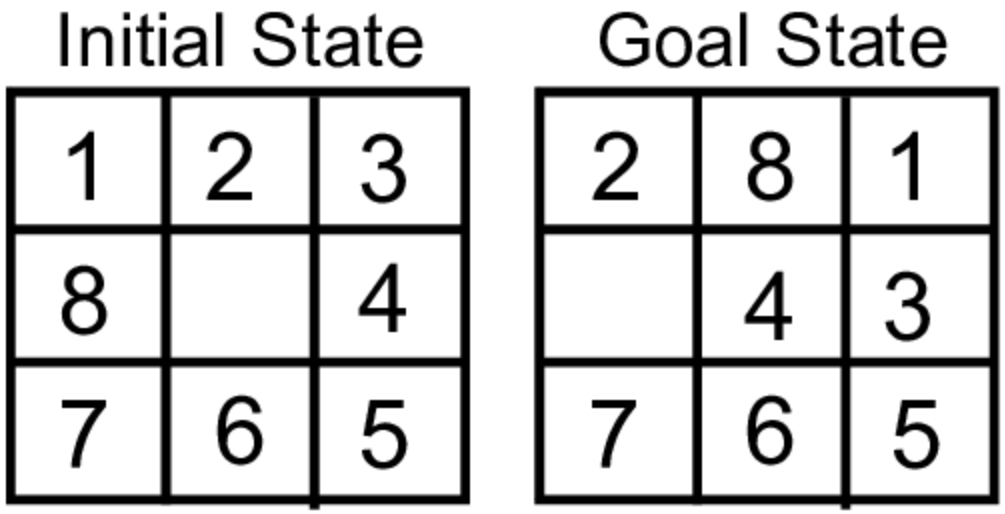
7. Giải bài toán 8-puzzle bằng giải thuật A*

Bài toán 8-puzzle

- Bài toán **8-puzzle** là một bài toán đố nổi tiếng gồm 8 miếng vuông (tile). Bài toán 8-Puzzle có 3 hàng và 3 cột. Bài toán có 8 miếng và một miếng để trống để các miếng khác có thể di chuyển đến.
- Cấu hình ban đầu và cấu hình đích của bài toán được cung cấp.
- Để giải bài toán, ta có thể di chuyển các miếng từng bước một đến ô trống sao cho đạt được cấu hình đích.

Bài toán 8-Puzzle (tt.)

Hình 3.10. Trạng thái ban đầu và trạng thái đích của bài toán 8-Puzzle



Các miếng trong trạng thái ban đầu có thể được di chuyển đến ô trống theo một thứ tự nào đó để đạt đến trạng thái đích.

Các qui tắc để giải bài toán 8-puzzle

- Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space.
- The empty space can only move in four directions:
  1. Up
  2.Down
  3. Right or
  4. Left
- The empty space cannot move diagonally and can take **only one step at a time** (i.e. move the empty space one position at a time).

A* Algorithm

- Trong giải thuật A* chúng ta dùng hai list: '**open list**' và '**closed list**'
- List **open** chứa tất cả những nút vừa được sinh ra và chưa hiện diện trong list **closed .** Mỗi nút mà đã được thăm dò và có những nút kế cận của nó được triển khai thì được đưa vào list **closed** và những nút kế cận của nó thì được đưa vào list **open**.
- Mỗi nút có một con trỏ chỉ đến nút cha của nó.
- Ban đầu, list open chỉ chứa nút khởi đầu (**start node**).

Hàm f

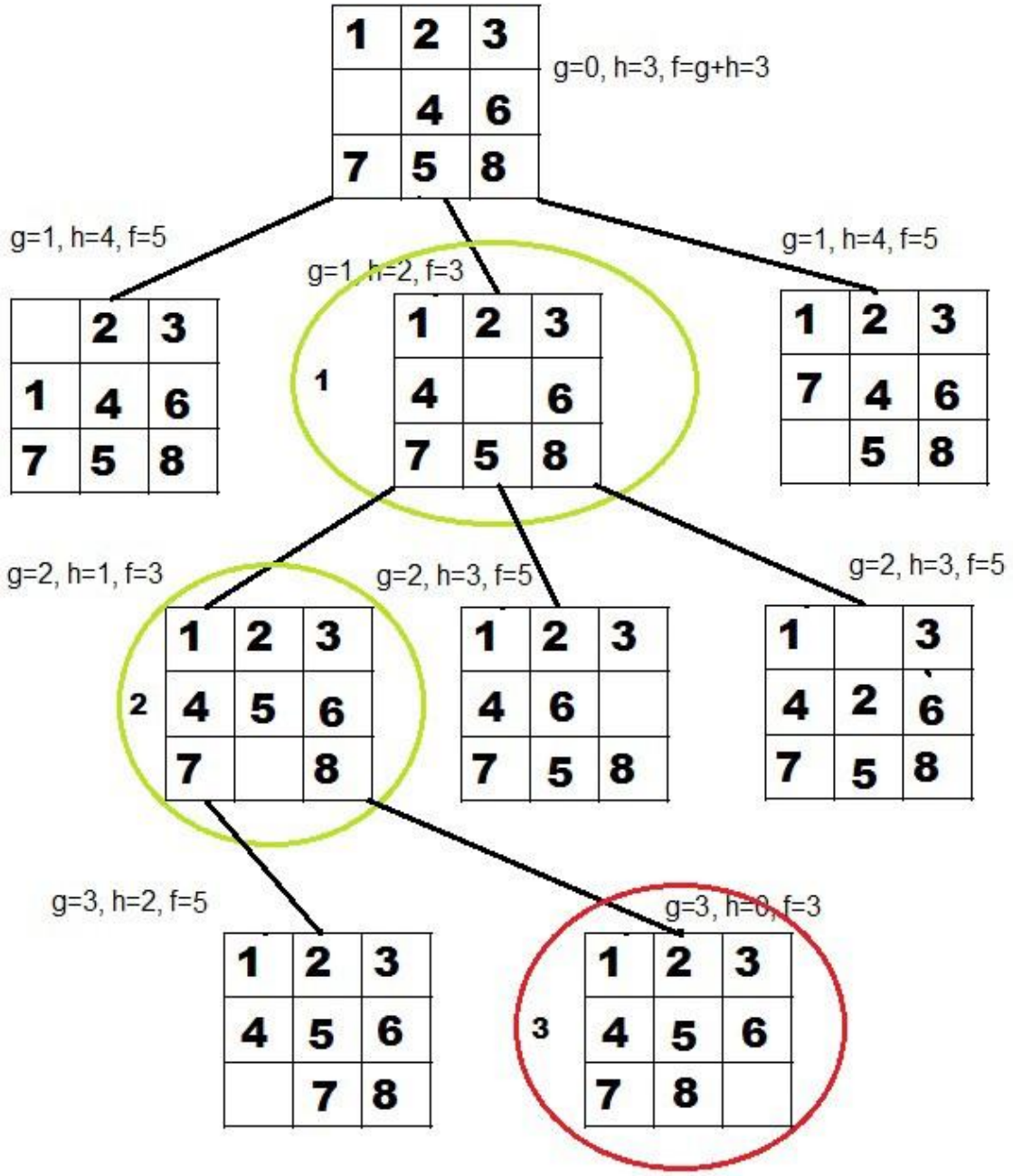
- Nút kế tiếp được chọn từ list open dựa vào trị f của nó, nút có trị f nhỏ nhất được chọn và triển khai để thăm dò.
- **f-score = h-score + g-score**
- A* sử dụng kết hợp hàm đánh giá heuristic (**h-score**: mức độ xa đối với nút đích) và hàm đánh giá **g-score** (thí dụ, số nút đã duyệt qua từ nút khởi đầu đến nút hiện hành).
- Trong bài toán 8-Puzzle, ta có thể định nghĩa **h-score** như là số ô bị sai vị trí của trạng thái hiện hành so với trạng thái đích.
- **g-score** tính bằng số nút đã duyệt qua từ nút khởi đầu đến nút hiện hành.

Hàm f

- Từ Hình 3.10, ta có thể tính **h-score** bằng cách so sánh trạng thái ban đầu với trạng thái đích và đếm số miếng bị sai vị trí.
  Như vậy, **h-score** = 5 và **g-score** = 0 vì số nút đã duyệt từ nút khởi đầu đến nút hiện hành là 0.

Cách giải thuật A* giải bài toán 8-Puzzle

- Đầu tiên, chúng ta di chuyển đến ô trống bằng tất cả những cách có thể từ trạng thái ban đầu và tính hàm f của mỗi trạng thái mới sinh ra. Điều này được gọi là mở rộng trạng thái hiện hành.
- Sau khi mở rộng trạng thái hiện hành, trạng thái này được đưa vào list **closed** và các trạng thái vừa mới tạo ra được đưa vào list **open** . Trong list open trạng thái nào có trị f nhỏ nhất sẽ được chọn và triển khai tiếp.
- Quá trình này tiếp diễn cho đến khi trạng thái đích xuất hiện như là trạng thái hiện hành.



Hình 3.11. Diễn tiến giải thuật A* để giải bài toán 8-puzzle.

Kết luận

- Tìm kiếm trên không gian trạng thái có thể giúp giải quyết một số bài toán trong AI
- DFS and BFS thuộc loại giải thuật tìm kiếm mù, không sử dụng thông tin hỗ trợ.
- Best first search, uniform cost search và A* thuộc loại tìm kiếm dựa vào heuristic, có sử dụng thông tin hỗ trợ để di chuyển tới trạng thái đích nhanh hơn.