ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC BÁCH KHOA

TRÍ TUỆ NHÂN TẠO

Khoa Công Nghệ Thông Tin

TS. Nguyễn Văn Hiệu

ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC BÁCH KHOA

Khoa Công Nghệ Thông Tin
TS. Nguyễn Văn Hiệu

TRÍ TUỆ NHÂN TẠO

Chương 2: Không gian trạng thái và tìm kiếm mù

Nội dung

- Giải quyết vấn đề
- Không gian trạng thái
- Tìm kiếm trên không gian trạng thái
- Tìm kiếm theo chiều rộng
- Tìm kiếm theo chiều sâu
- Tìm kiếm chiều sâu có giới hạn
- Bài tập
- Tìm kiếm đều giá

Giải quyết vấn đề

- Phát biểu chính xác bài toán:
  - Hiện trạng ban đầu;
  - Kết quả mong muốn,...
- Phân tích bài toán.
- Thu thập và biểu diễn dữ liệu, tri thức cần thiết để giải bài toán.
- Ra quyết định chọn kỹ thuật/ giải quyết thích hợp.

Không gian trạng thái

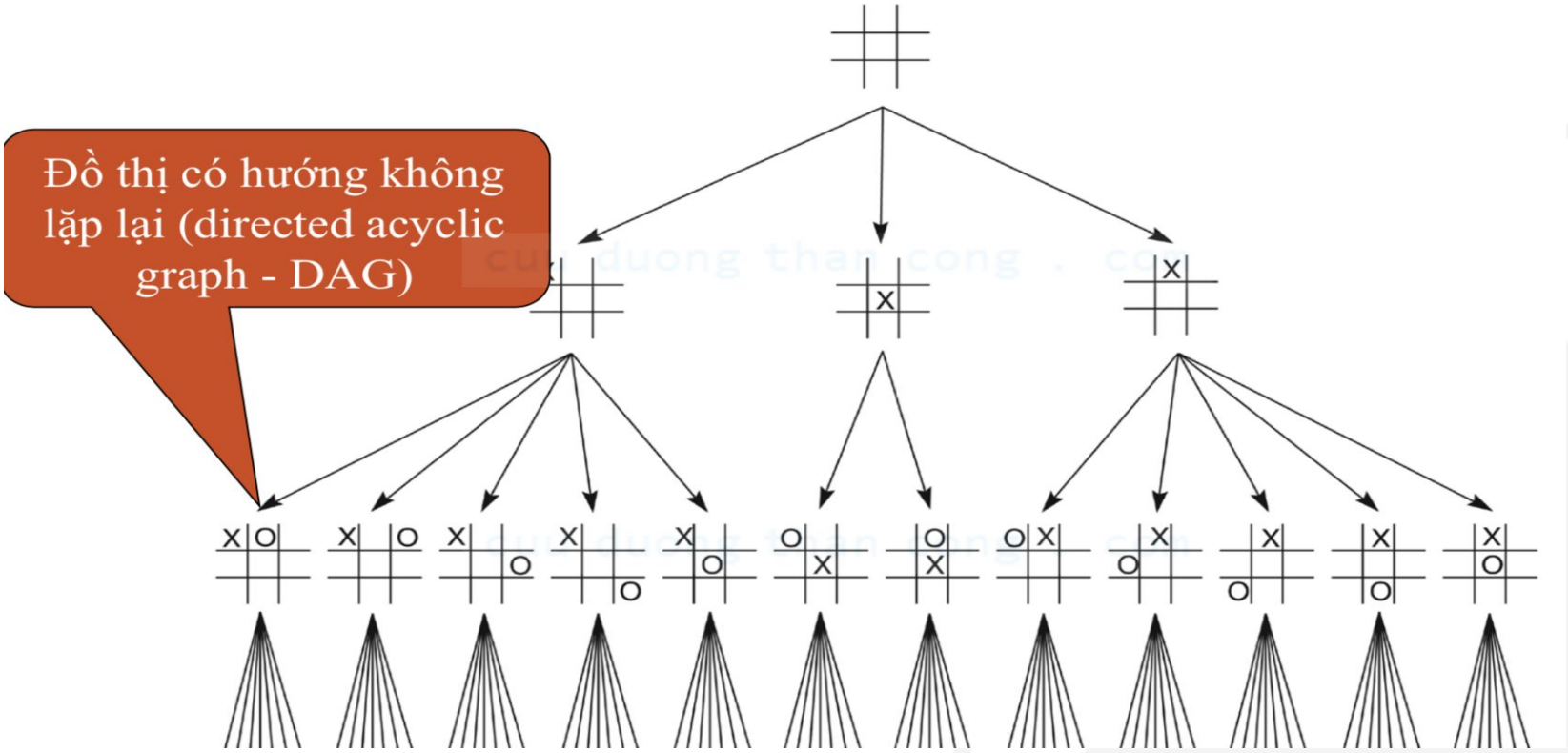
- Tập tất cả trạng thái có thể có và tập toán tử của bài toán
- Tồn tại nhiều cách để biểu diễn bài toán
- Đồ thị để biểu diễn bài toán:- đồ thị không gian trạng thái
- Bài toán 7 cây cầu ở Konigsberg của Euler
- Đồ thị không gian trạng thái để phân tích cấu trúc và độ phức tạp của bài toán.

Không gian trạng thái

- Đồ thị không gian trạng thái là một bộ ký hiệu [V, E, S, G] trong đó:
  - V - tập đỉnh/tập tất cả các trạng thái;
  - E - tập cạnh/ tập cung/ tập toán tử;
  - S - trạng thái đầu;
  - G- trạng thái đích.
- Đặc tính của trạng thái đích G:
  - Đặc tính có thể đo lường được trong quá trình tìm kiếm.
  - Ví dụ: Tic-tac- toe, chess,...
  - Đặc tính của đường đi được hình thành trong quá trình tìm kiếm. Ví dụ : travelling salesman problem, transportation problem,...
- Lời giải (solution path) là một con đường đi từ trạng thái S đến trạng thái G.

Không gian trạng thái

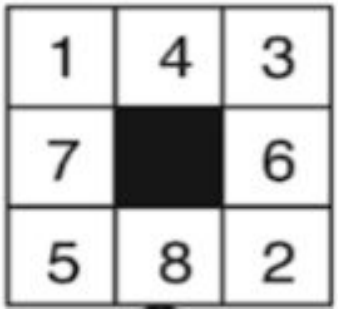
- Ví dụ một phần không gian trạng thái của trò chơi Tic-tac-toe



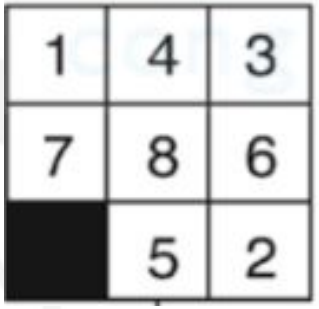
Không gian trạng thái

- Ví dụ trò chơi 8 ô chữ

trạng thái đầu



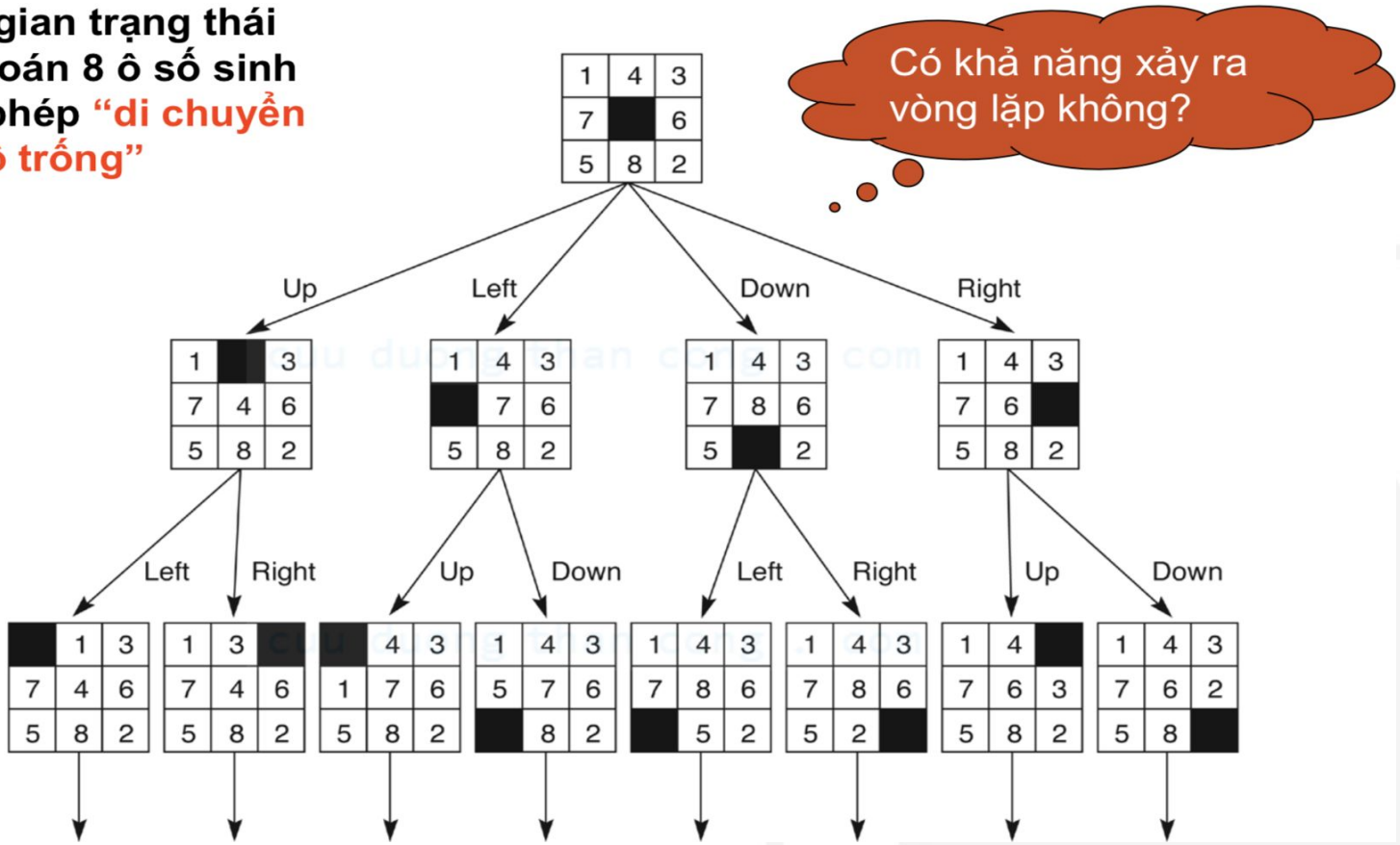
trạng thái đích



- Cách biểu diễn trạng thái của trò chơi này như thế nào ?

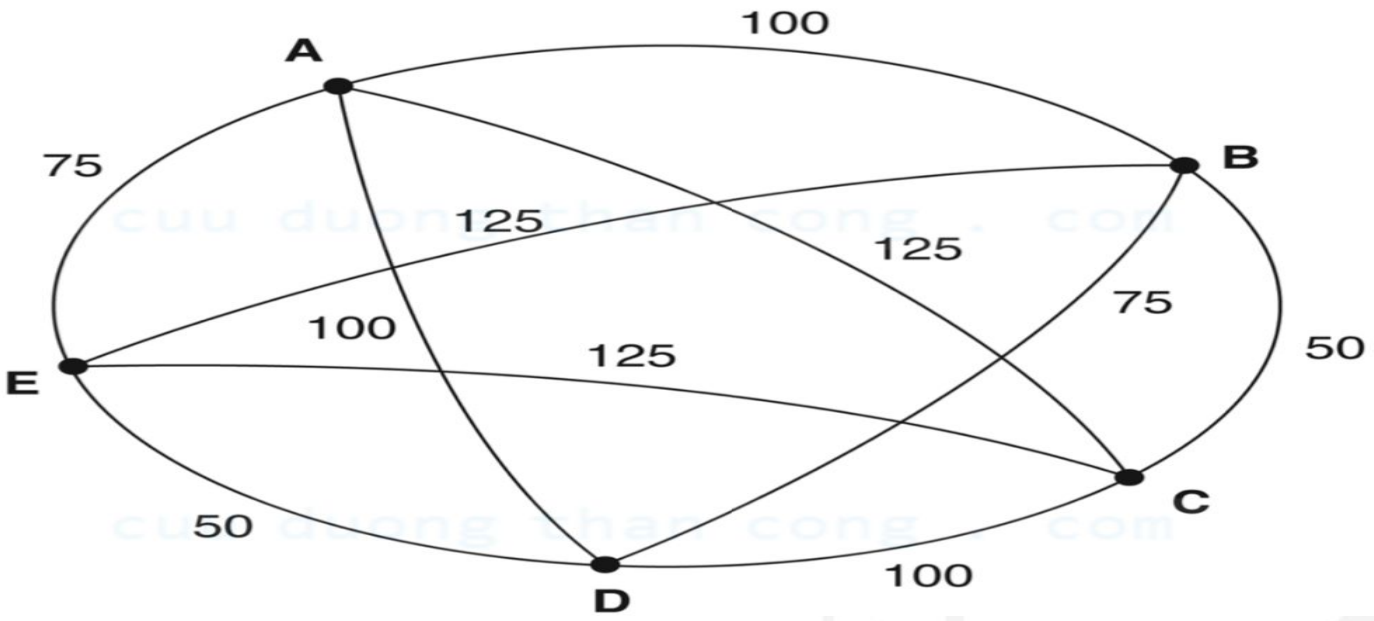
Không gian trạng thái

- **Không gian trạng thái của bài toán 8 ô số sinh ra bằng phép "di chuyển ô trống"**

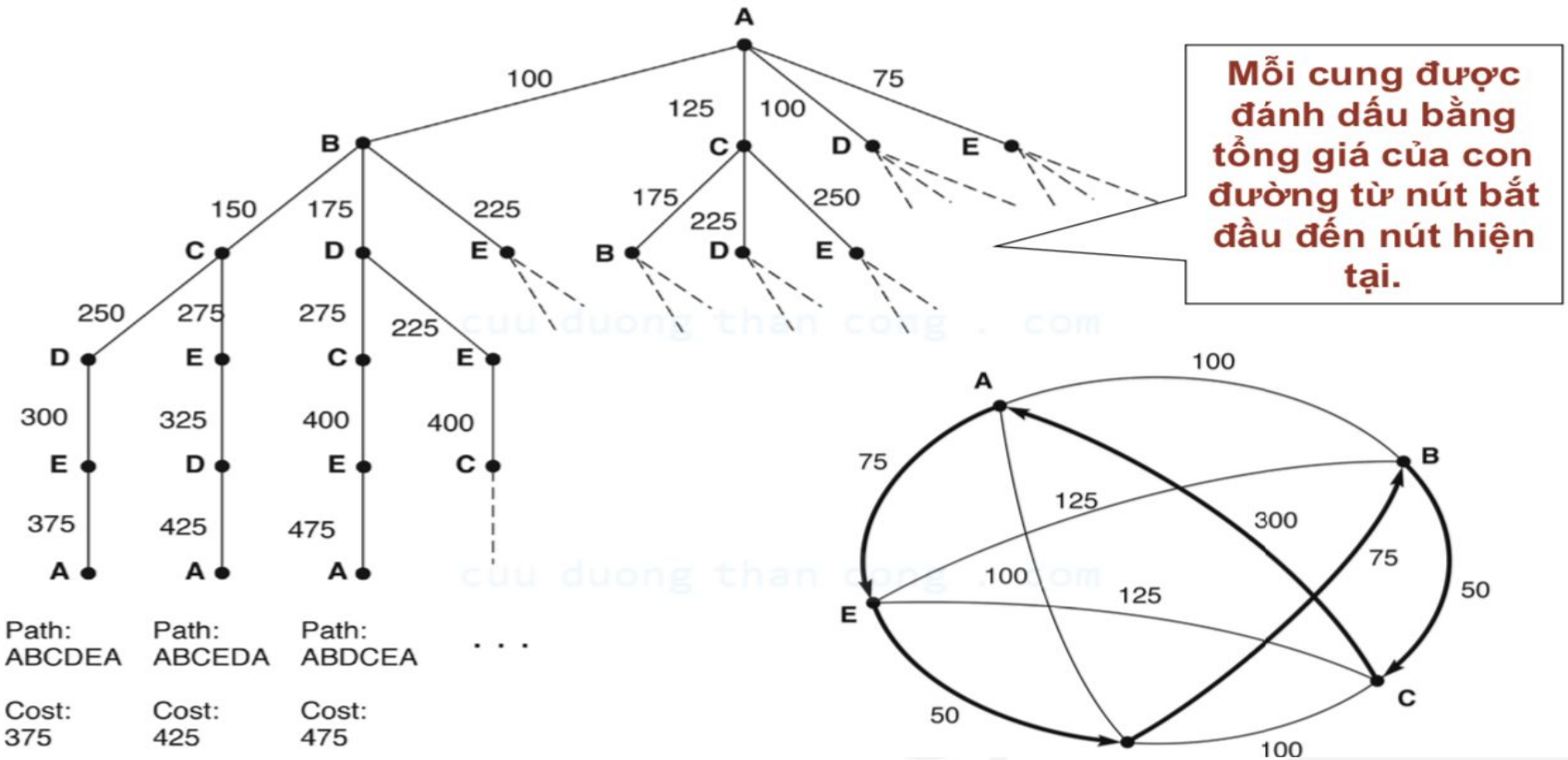


Không gian trạng thái

- Cần biểu diễn không gian trạng thái cho bài toán người giao hàng như thế nào?



Không gian trạng thái



Không gian trạng thái

- Yếu tố chính để xác định không gian trạng thái:
  - Trạng thái;
  - Hành động(toán tử);
  - Kiểm tra trạng thái thỏa đích;
  - Chi phí mỗi bước chuyển trạng thái.

Tìm kiếm trên không gian trạng thái

- Chiến lược tìm kiếm: lựa chọn thứ tự xét các trạng thái.

- Đánh giá chiến lược :
  - ***đủ***: liệu có tìm được lời giải (nếu có).
  - ***độ phức tạp thời gian***: số lượng trạng thái phải xét.
  - ***độ phức tạp lưu trữ***: tổng dung lượng bộ nhớ phải lưu trữ .
  - ***tối ưu***: có lời giải tối ưu.

- Độ phực tạp thời gian và lưu trữ có thể được đo bằng:
  - **b**: độ phân nhánh của cấu trúc cây.
  - **d**: độ sâu của lời giải ngắn nhất.
  - **m**: độ sâu tối đa của không gian trạng thái (có thể vô hạn).

Tìm kiếm trên không gian trạng thái

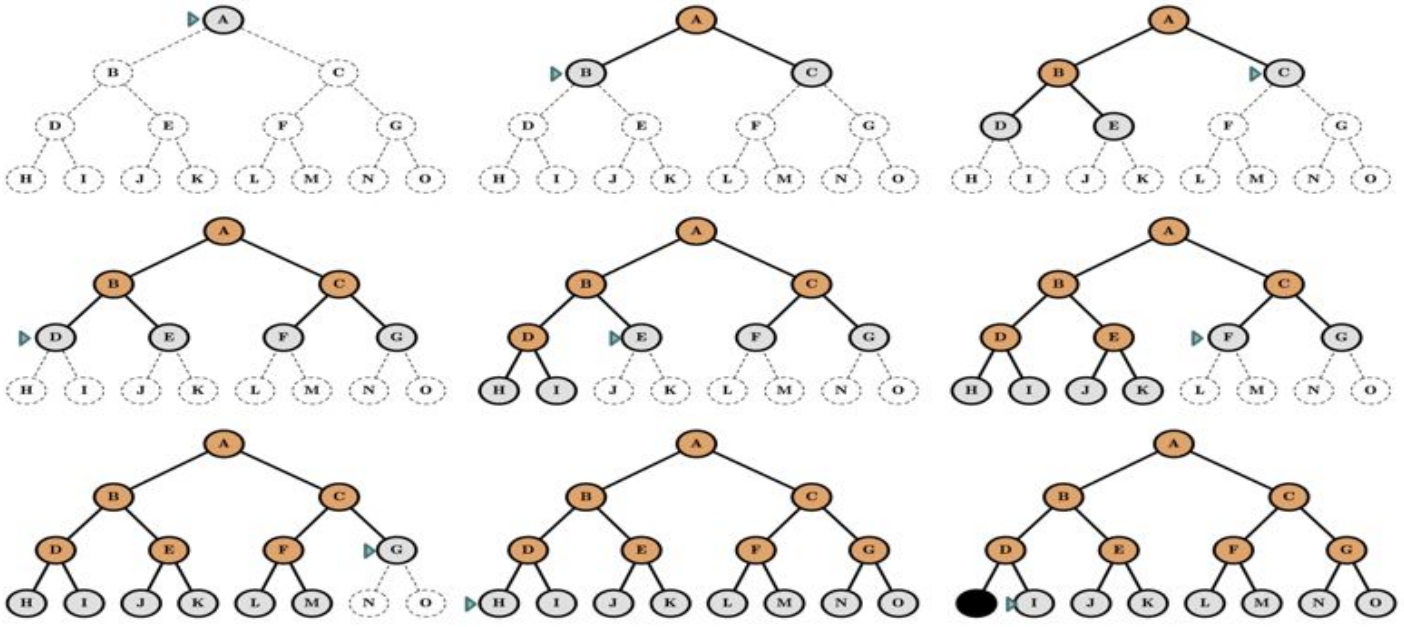
- **Chiến lược tìm kiếm mù**
  - **Tìm kiếm theo chiều rộng.**
  - Tìm kiếm đều giá (uniform-cost search).
  - **Tìm kiếm theo chiều sâu**.
  - Tìm kiếm theo chiều sâu có giới hạn.

Tìm kiếm theo chiều rộng

- Tìm kiếm theo từng tầng (từ trên xuống)
- Phát triển trạng thái gần với trạng thái hiện tại.
- Tập trạng thái được chia thành 3 khu vực:
  - Khu vực đã phát triển(explored):tập trạng thái đã xét
  - Khu vực chờ phát triển(frontier):tập trạng thái đang chờ xét
  - Khu vực chưa phát triển(unexplored): tập trạng thái chưa xét

Tìm kiếm theo chiều rộng

- Minh hoạ sự phát triển các đỉnh theo thuật toán tìm kiếm theo chiều rộng



- A->B->C->D->E->F->G->H->I->J->K->L->M->N->0

Tìm kiếm theo chiều rộng

- Thuật toán tìm kiếm theo chiều rộng được mô tả bởi thủ tục:

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

Tìm kiếm theo chiều rộng

- Bài tập[5 phút]. Viết danh sách các biến đổi trạng thái trong hàng đợi **frontier ?**

frontier = {A}

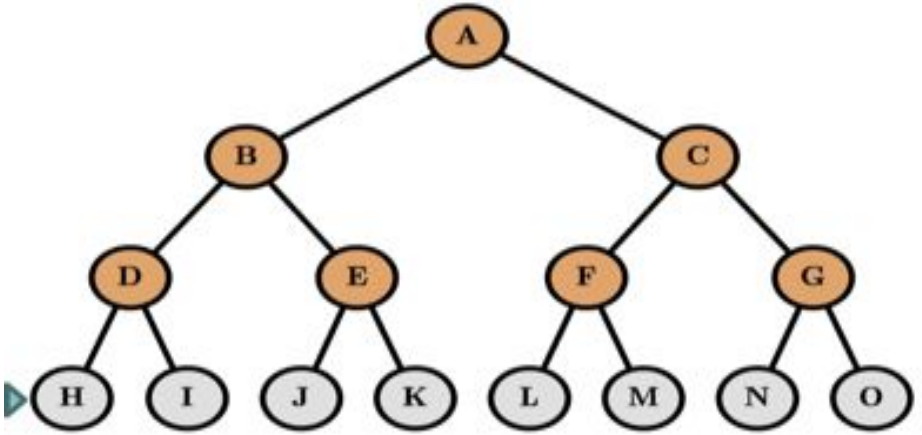
frontier = {B, C}

frontier = {C, D, E}

frontier = {D, E, F, G}

frontier = {E, F, G, H, I}

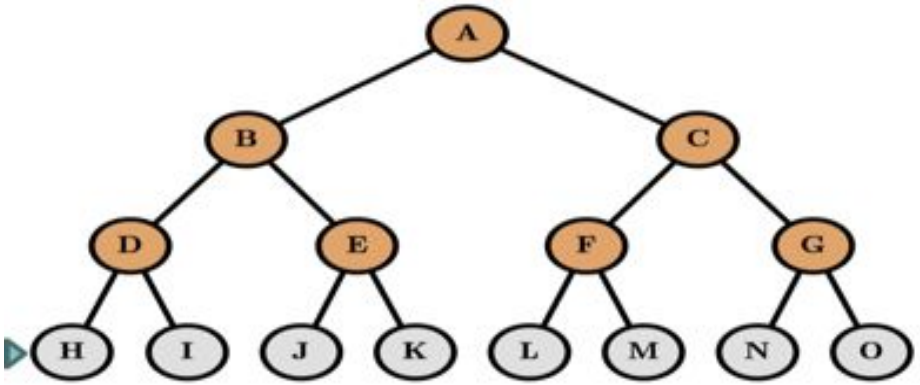
frontier = …




```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
      returns SUCCESS or FAILURE :

      frontier = Queue.new(initialState)
      explored = Set.new()

      while not frontier.isEmpty():
            state = frontier.dequeue()
            explored.add(state)

            if goalTest(state):
                  return SUCCESS(state)

            for neighbor in state.neighbors():
                  if neighbor not in frontier ∪ explored:
                        frontier.enqueue(neighbor)

      return FAILURE
```

Demo 1(10 phút)

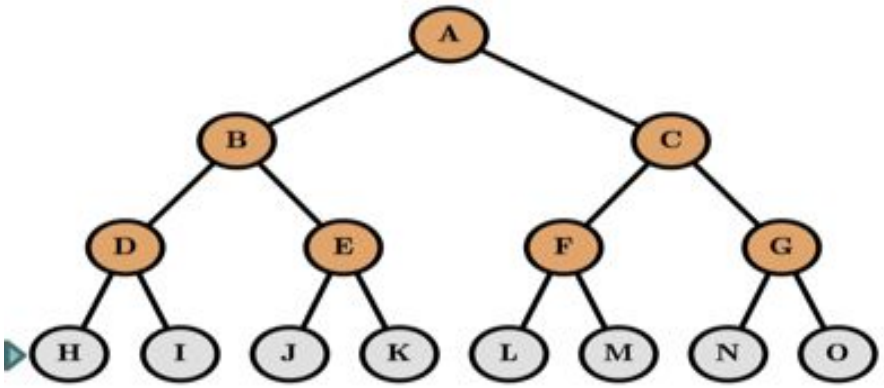
```
13  if __name__ == '__main__':
14      graph = {
15      'A' : ['B','C'],
16      'B' : ['D','E'],
17      'C' : ['F','G'],
18      'D' : ['H','I'],
19      'E' : ['J','K'],
20      'F' : ['L','M'],
21      'G' : ['N','O'],
22      'H' : [],
23      'I' : [],
24      'J' : [],
25      'K' : [],
26      'L' : [],
27      'M' : [],
28      'N' : [],
29      'O' : []
30      }
31      result = BFS('A','O')
32      if result:
33          s = 'explored: '
34          for i in result:
35              s += i + ' '
36              print(s)
37      else:
38          print("404 Not Found!")
```




```
1  def BFS(initialState, goal):
2      frontier = [initialState]
3      explored = []
4      while frontier:
5          state = frontier.pop(0)
6          explored.append(state)
7          if goal == state:
8              return explored
9          for neighbor in graph[state]:
10             if neighbor not in (explored and frontier):
11                 frontier.append(neighbor)
12     return False
```

Demo 2(10 phút)

```
14 if __name__ == '__main__':
15     tree = Tree()
16     tree.create_node('A', 'A') #root
17     tree.create_node('B', 'B', 'A')
18     tree.create_node('C', 'C', 'A')
19     tree.create_node('D', 'D', 'B')
20     tree.create_node('E', 'E', 'B')
21     tree.create_node('F', 'F', 'C')
22     tree.create_node('G', 'G', 'C')
23     tree.create_node('H', 'H', 'D')
24     tree.create_node('I', 'I', 'D')
25     tree.create_node('J', 'J', 'E')
26     tree.create_node('K', 'K', 'E')
27     tree.create_node('L', 'L', 'F')
28     tree.create_node('M', 'M', 'F')
29     tree.create_node('N', 'N', 'G')
30     tree.create_node('O', 'O', 'G')
31     result = BFS(tree.get_node("A"), 'O')
32
33     if result:
34         s = 'explored: '
35         for i in result:
36             s += i.tag + ' '
37             print(s)
38     else:
39         print("404 Not Found!")
```




```
1  from treelib import Tree, Node
2  def BFS(initialState: Node, goal):
3      frontier = [initialState]
4      explored = []
5      while frontier:
6          state = frontier.pop(0)
7          explored.append(state)
8          if goal == state.tag:
9              return explored
10         for neighbor in tree.children(state.identifier):
11             if neighbor not in (explored and frontier):
12                 frontier.append(neighbor)
13     return False
```

Tìm kiếm theo chiều rộng

- Trạng thái nào phát sinh trước sẽ được duyệt trước, nên frontier phải cấu trúc dữ liệu FIFO.
- Đánh giá thuật toán theo chiều rộng
  - ***Tính đủ ?***
    - có (nếu b hữu hạn)
  - ***Độ phức tạp thời gian ?***:
    - 1 + 1.b + b.b + … + b^(d-1). b = 0(b^d)
  - ***Độ phức tạp về lưu trữ ?***:
    - 0(b^m)
  - ***Tính tối ưu ?***:
    - có(nếu chi phí cho mỗi bước chuyển là 1 đơn vị).
- Độ phức tạp về thời gian và không gian theo cấp số nhanh.

Tìm kiếm theo chiều rộng

- Tìm kiếm theo chiều rộng tệ đến mức nào ?

Tìm kiếm theo chiều rộng

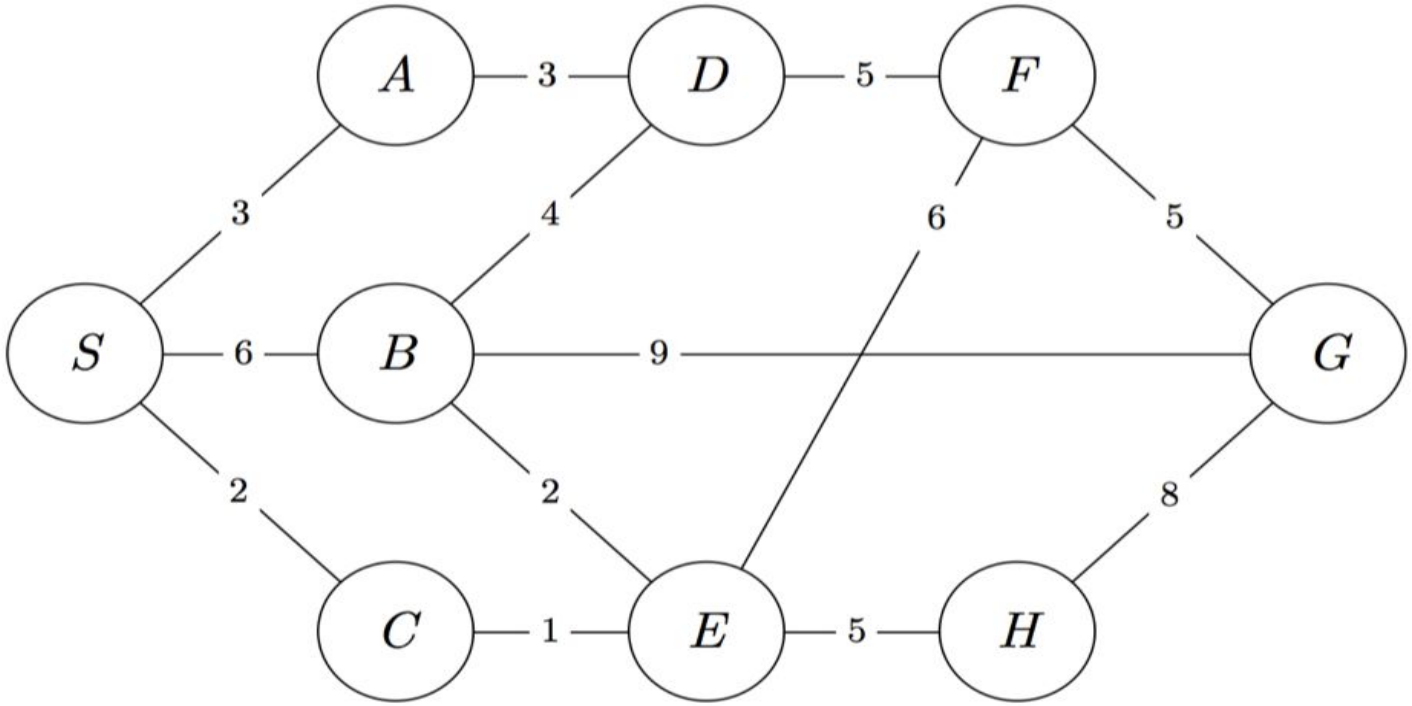
- Thực nghiệm

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

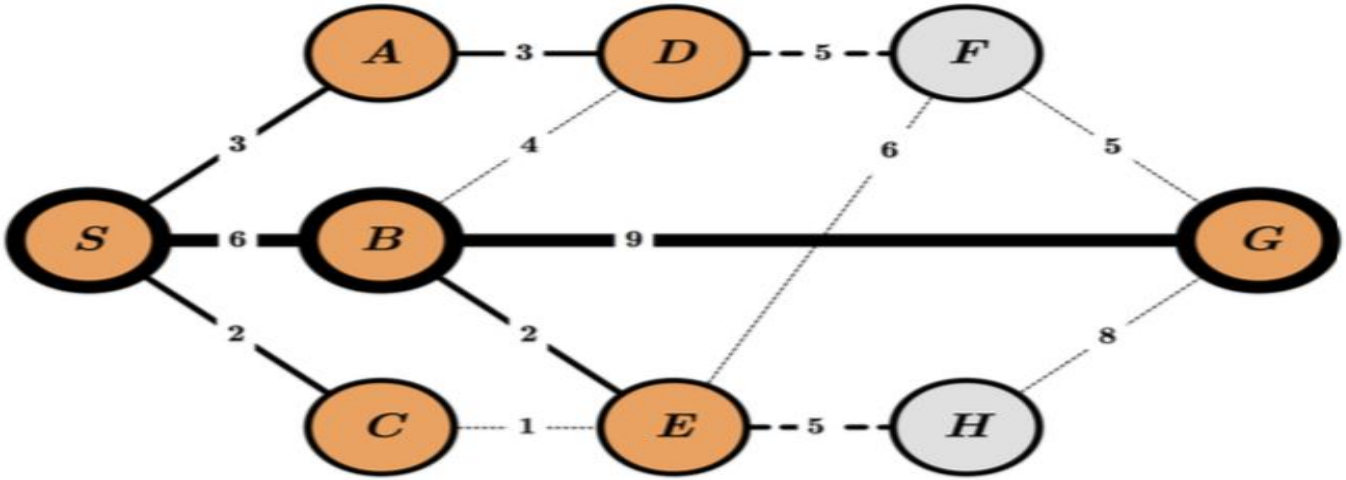
- b = 10, 1 giây kiểm tra được 1 triệu nodes, lưu 1 node mất 1.000byte
- **Độ phức tạp thời gian và không gian(bộ nhớ) là bất lợi của thuật toán**

Bài tập

- Hãy xây dựng thứ tự khám phá và đường đi của các đỉnh sử kỷ tìm kiếm theo chiều rộng.



Đáp án: BFS



Queue:

S	A	B	C	D	E	G	F	H

Order of Visit:

S *A* *B* *C* *D* *E* *G*

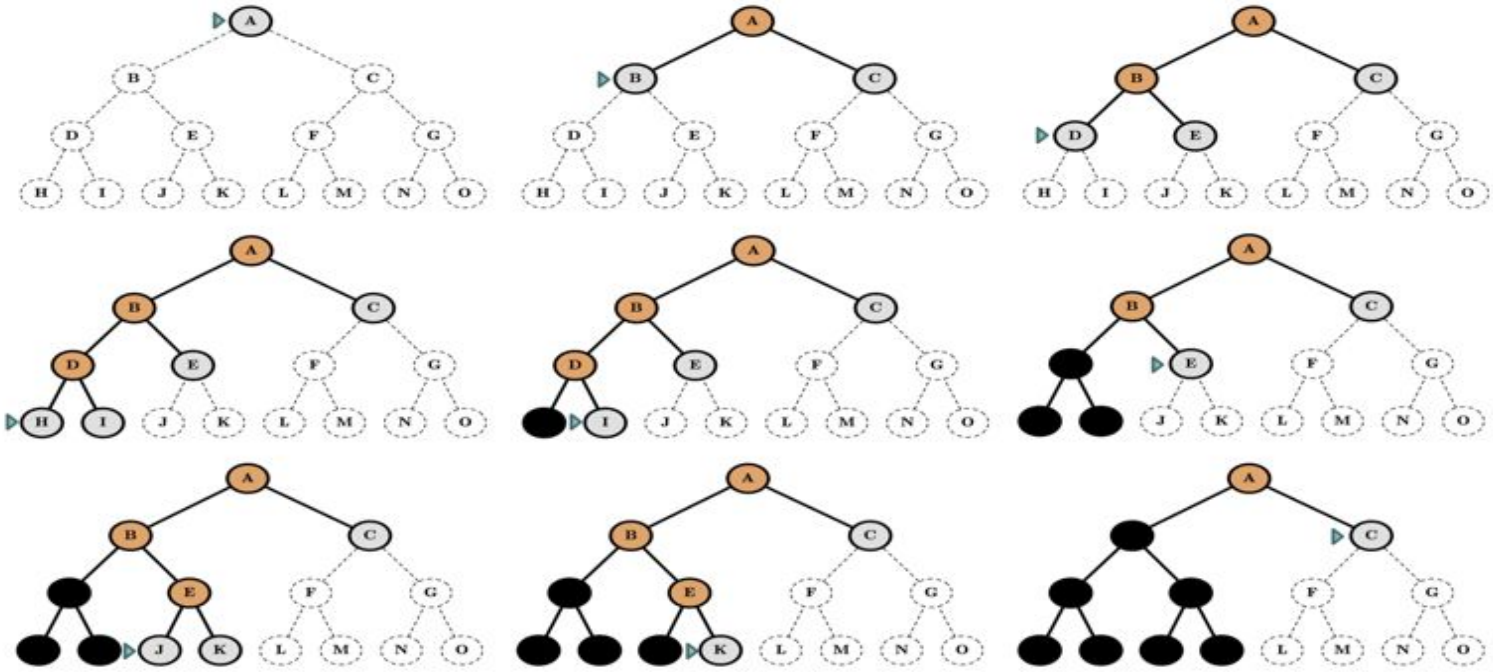
Tìm kiếm theo chiều sâu

Tìm kiếm theo chiều sâu

- Phát triển các trạng thái ở sâu so với trạng thái hiện tại
- Tập trạng thái được chia thành 3 khu vực:
  - Khu vực đã phát triển(explored): trạng thái đã xét;
  - Khu vực chờ phát triển(frontier): trạng thái đang chờ xét;
  - Khu vực chưa phát triển(unexplored): trạng thái chưa xét.

Tìm kiếm theo chiều sâu

- Minh hoạ việc phát triển các đỉnh theo thuật toán tìm kiếm theo chiều sâu



Tìm kiếm theo chiều sâu

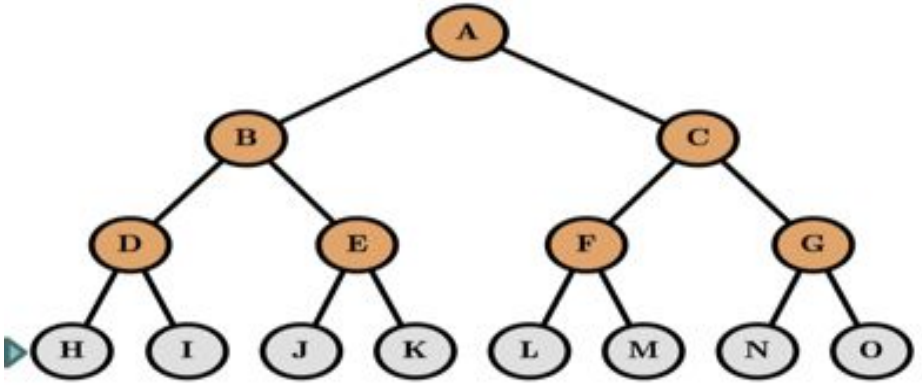
- Thuật toán tìm kiếm theo chiều sâu được mô tả bởi thủ tục

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
      returns SUCCESS or FAILURE :

      frontier = Stack.new(initialState)
      explored = Set.new()

      while not frontier.isEmpty():
            state = frontier.pop()
            explored.add(state)

            if goalTest(state):
                  return SUCCESS(state)

            for neighbor in state.neighbors():
                  if neighbor not in frontier ∪ explored:
                        frontier.push(neighbor)

      return FAILURE
```

Tìm kiếm theo chiều sâu

- Hãy làm rõ quá trình biến đổi của danh sách **frontier ?**

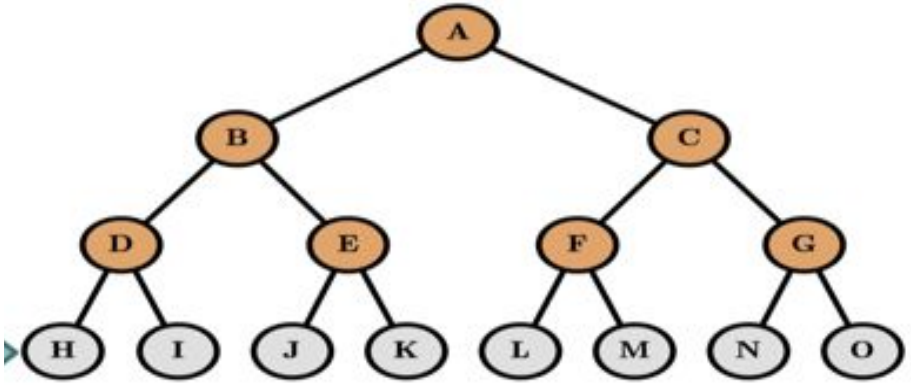
Frontier = {A}
Frontier = {B,C,}
Frontier = {B, F, G}
Frontier = {B, F, N,O}
Frontier = {B, F, N }
Frontier = {B, F}
Frontier = {B,L, M}
Frontier = {….}




```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
      returns SUCCESS or FAILURE :

      frontier = Stack.new(initialState)
      explored = Set.new()

      while not frontier.isEmpty():
            state = frontier.pop()
            explored.add(state)

            if goalTest(state):
                  return SUCCESS(state)

            for neighbor in state.neighbors():
                  if neighbor not in frontier ∪ explored:
                        frontier.push(neighbor)

      return FAILURE
```

Demo 3(10 phút)

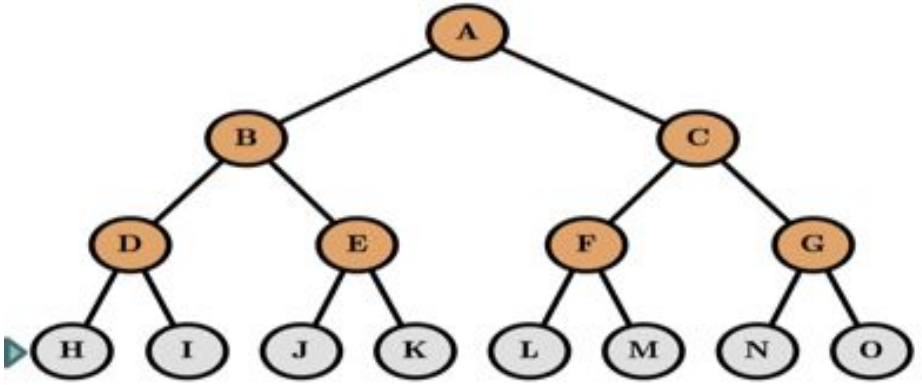
```
if __name__ == '__main__':
    nodeA = Node("A")
    nodeB = Node("B")
    nodeC = Node("C")
    nodeD = Node("D")
    nodeE = Node("E")
    nodeF = Node("F")
    nodeG = Node("G")
    nodeH = Node("H")
    nodeI = Node("I")
    nodeJ = Node("J")
    nodeK = Node("K")
    nodeL = Node("L")
    nodeM = Node("M")
    nodeN = Node("N")
    nodeO = Node("O")
    nodeA.addChild([nodeB, nodeC])
    nodeB.addChild([nodeD,nodeE])
    nodeC.addChild([nodeF,nodeG])
    nodeD.addChild([nodeH, nodeI])
    nodeE.addChild([nodeJ,nodeK])
    nodeF.addChild([nodeL,nodeM])
    nodeG.addChild([nodeN,nodeO])
    result = DFS(nodeA, 'H')
    if result:
        s = 'explored: '
        for i in result:
            s += i.name + " "
            print(s);
    else:
        print('404 Not Found!')
```




```
class Node:
    def __init__(sefl, name):
        sefl.name = name
        sefl.children = []
    def addChild(self, list):
        for c in list:
            self.children.append(c)
def DFS(initialState, goal):
    frontier = [initialState]
    explored = []
    while frontier:
        state = frontier.pop()
        explored.append(state)
        if goal == state.name:
            return explored
        for child in state.children:
            if child not in (explored and frontier):
                frontier.append(child)
    return False
```

Demo 4(5 phút)

```
13  if __name__ == '__main__':
14      graph = {
15      'A' : ['B','C'],
16      'B' : ['D','E'],
17      'C' : ['F','G'],
18      'D' : ['H','I'],
19      'E' : ['J','K'],
20      'F' : ['L','M'],
21      'G' : ['N','O'],
22      'H' : [],
23      'I' : [],
24      'J' : [],
25      'K' : [],
26      'L' : [],
27      'M' : [],
28      'N' : [],
29      'O' : []
30      }
31      result = DFS('A','H')
32      if result:
33          s = 'explored: '
34          for i in result:
35              s += i + ' '
36              print(s)
37      else:
38          print("404 Not Found!")
```




```
1  def DFS(initialState, goal):
2      frontier = [initialState]
3      explored = []
4      while frontier:
5          state = frontier.pop(len(frontier)-1)
6          explored.append(state)
7          if goal == state:
8              return explored
9          for neighbor in graph[state]:
10             if neighbor not in (explored and frontier):
11                 frontier.append(neighbor)
12     return False
```

Tìm kiếm theo chiều sâu

- Nhận xét về thuật toán
  - Đối với BFS luôn tìm được nghiệm (nếu bài toán có nghiệm)
  - Đối với DFS chỉ tìm được nghiệm(nếu không gian hữu hạn), vì nếu không gian tìm kiếm vô hạn, thì thuật toán tìm kiếm theo chiều sâu chọn nhánh để đi, nếu đi đúng nhánh vô hạn mà nghiệm không nằm trên nhánh đó.

Tìm kiếm theo chiều sâu

- Đánh giá thuật toán theo chiều rộng
  - ***Tính đủ ?***
    - không (nếu không gian tìm kiếm vô hạn hoặc lặp)
    - đủ (nếu khử lặp và không gian tìm kiếm hữu hạn)
  - ***Độ phức tạp thời gian ?***:
    - 0(b^d) - số lượng nút được duyệt trong trường hợp xấu nhất
    - Nếu hàm đánh giá tốt thì, thời gian thực tế giảm đáng kể
  - ***Độ phức tạp không gian gian?***:
    - 0(b.m): Tại một thời điểm, DFS chỉ lưu một nhánh duy nhất của cây
  - ***Tính tối ưu ?***:
    - không

Tìm kiếm theo chiều sâu

- Quay trở lại với bảng đánh giá tìm kiếm theo chiều rộng

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

b = 10, 1 giây kiểm tra được 1 triệu nodes, lưu 1 node mất 1.000byte

- **Với d = 16, có thể giảm dung lượng bộ nhớ từ 10 exabyte trên BFS xuống bao nhiêu ….. trên tìm kiếm theo chiều sâu(DFS) ?**

Tìm kiếm theo chiều sâu

- Quay trở lại với bảng đánh giá tìm kiếm theo chiều rộng

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

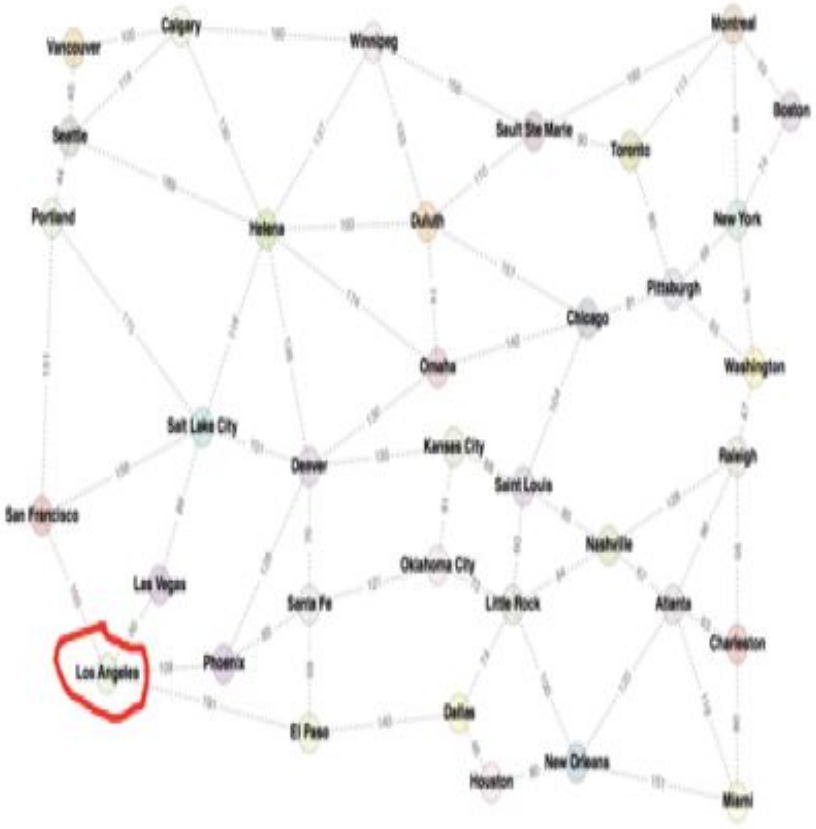
b = 10, 1 giây kiểm tra được 1 triệu nodes, lưu 1 node mất 1.000byte

- **Với d = 16, có thể giảm dung lượng bộ nhớ từ 10 exabytes trên BFS xuống 156 kilobytes DFS**

Tìm kiếm chiều sâu có giới hạn

Tìm kiếm theo chiều sâu có giới hạn

- Giả thiết nếu biết trước bài toán thì không duyệt hết độ sâu trong thuật toán tìm kiếm theo chiều sâu.
- Chọn độ sâu L, để phát triển thuật toán tìm kiếm theo chiều sâu (các đỉnh ở độ sâu L, không chứa đỉnh con, không chứa đỉnh cháu)
- Ý tưởng: Qua thực nghiệm chỉ ra rằng không có một thành phố nào tới thành phố khác với độ sâu vượt quá 36, nên L = 36

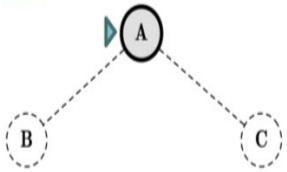


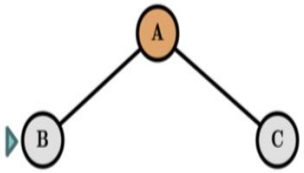
Tìm kiếm theo chiều sâu có giới hạn

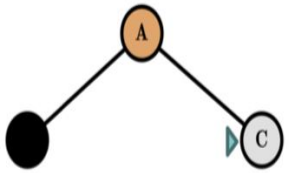
Limit = 0

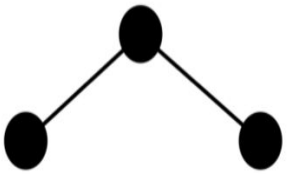
Tìm kiếm theo chiều sâu có giới hạn

Limit = 1



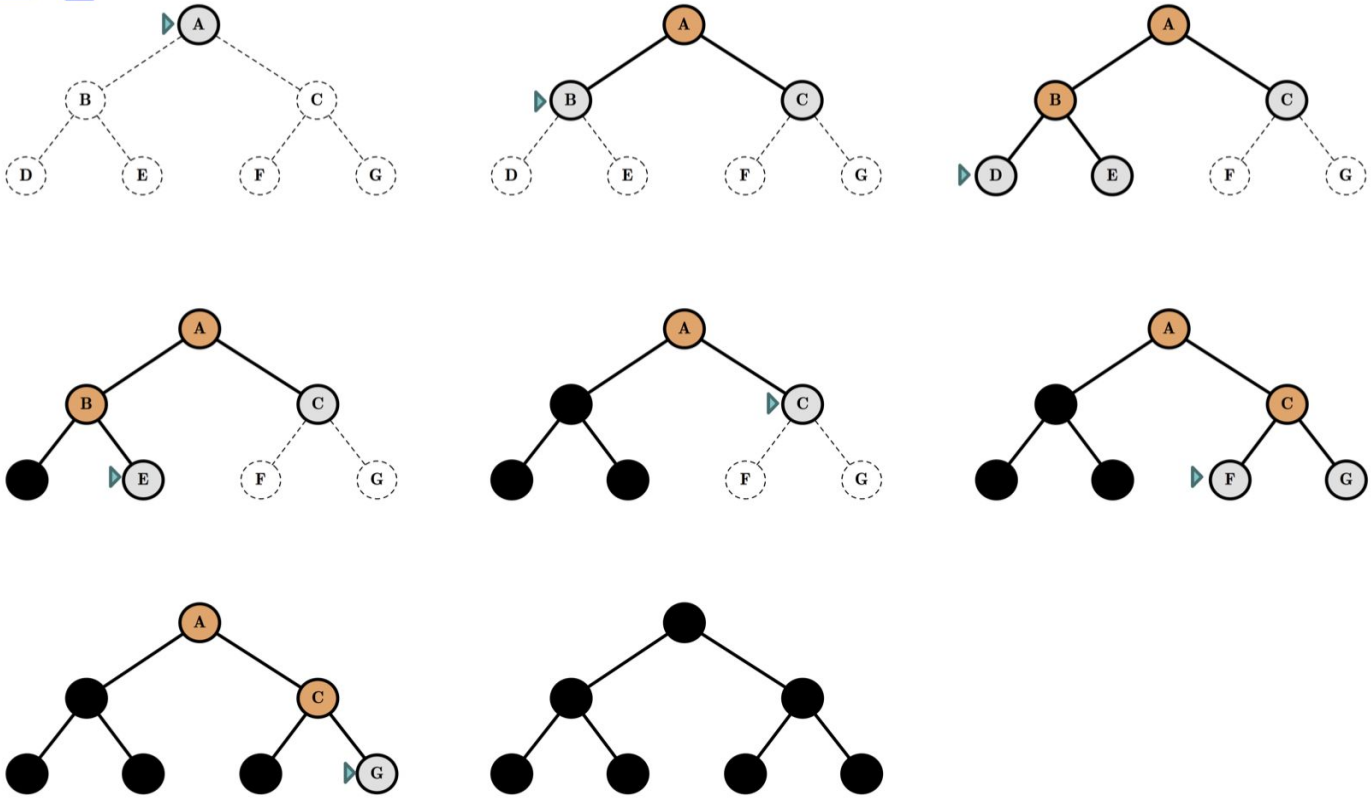






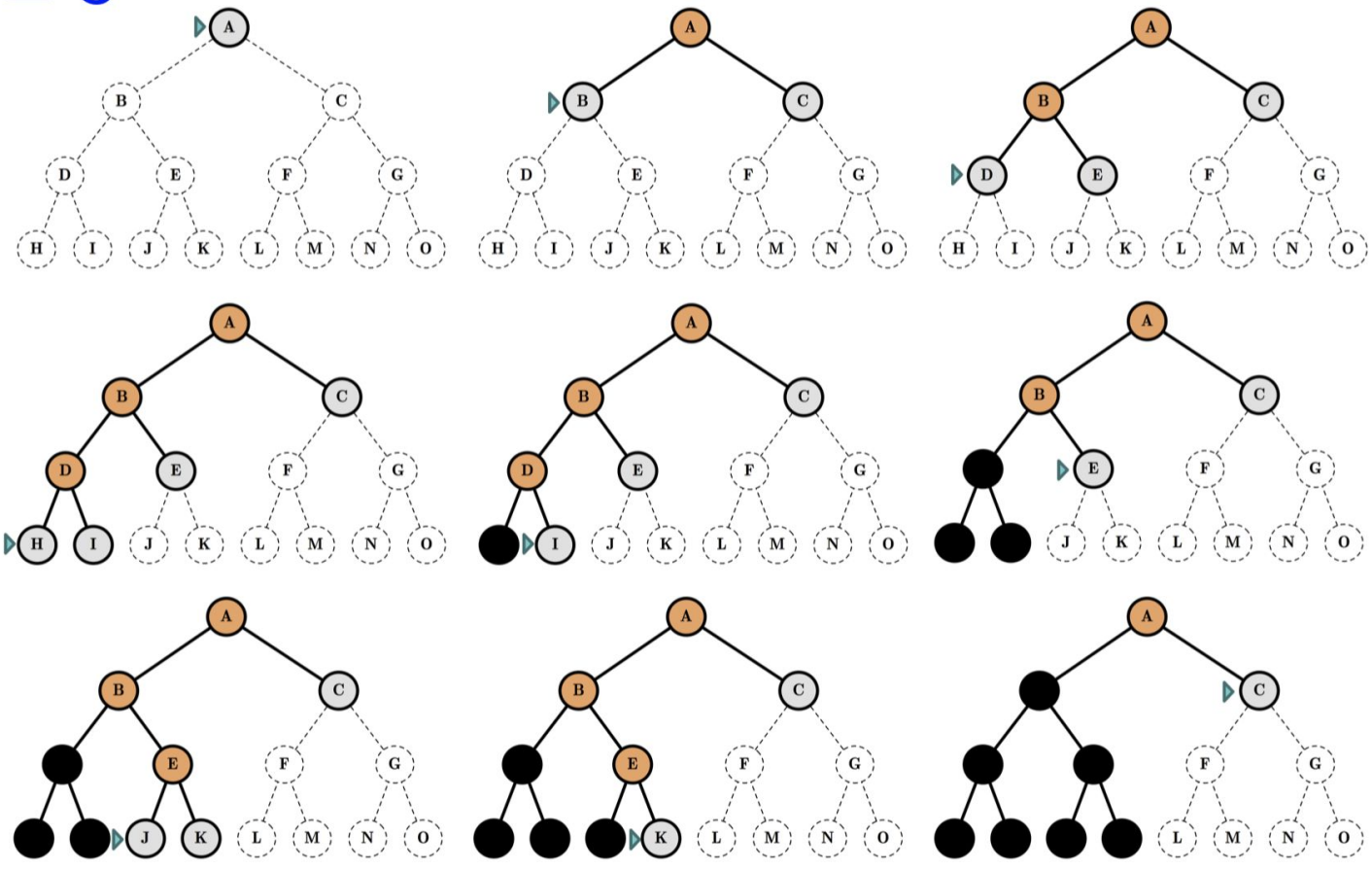
Tìm kiếm theo chiều sâu có giới hạn

Limit = 2



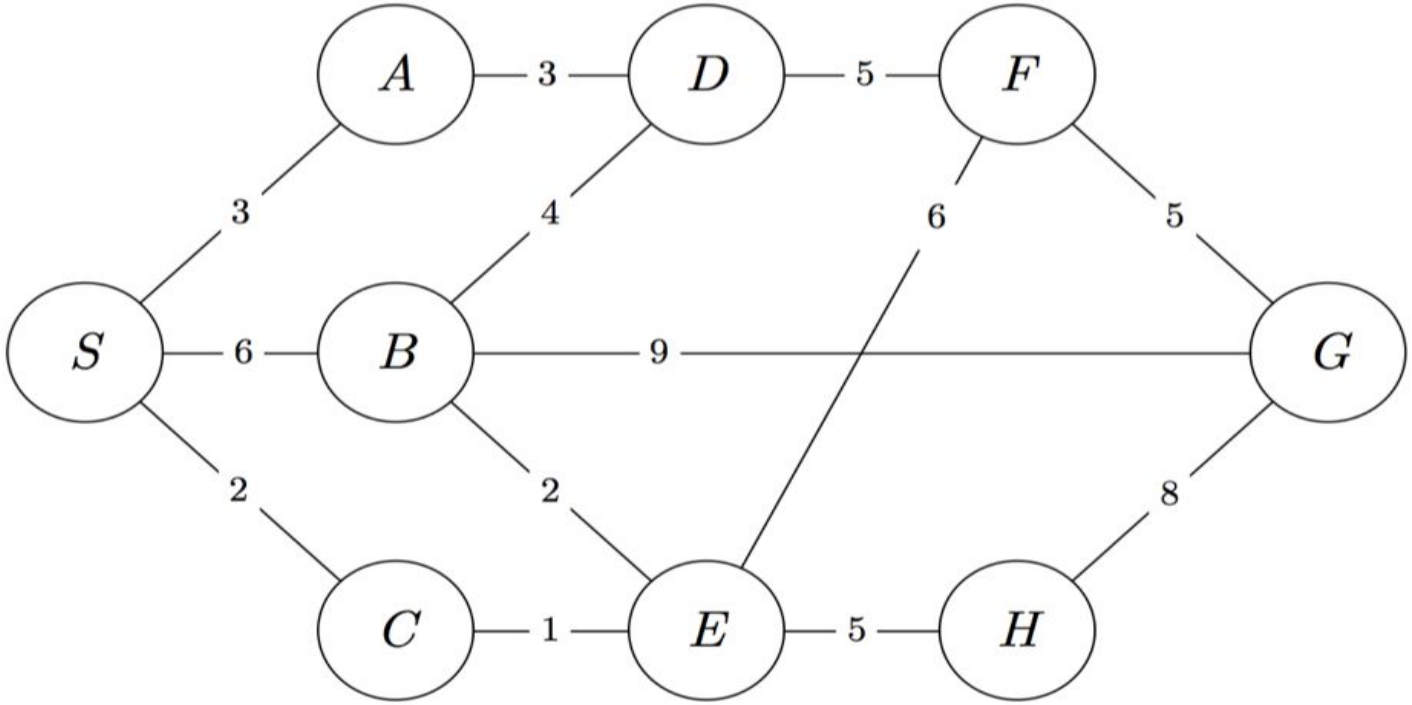
Tìm kiếm theo chiều sâu có giới hạn

Limit = 3



Bài tập 1

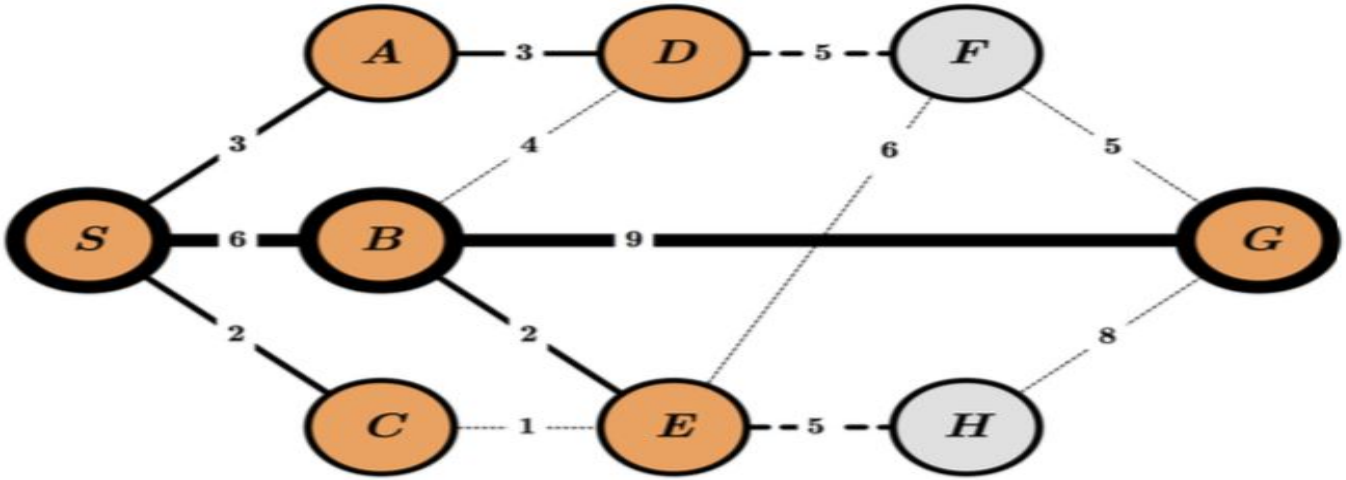
- Hãy xây dựng thứ tự khám phá và đường đi của các đỉnh sử kỷ tìm kiếm theo chiều rộng, tìm kiếm theo chiều sâu và tìm kiếm đều giá.



Bài tập 2

- Hãy viết chương trình với các thuật toán tìm kiếm theo chiều rộng, thuật toán tìm kiếm theo chiều sâu.

Đáp án bài tập 1: BFS



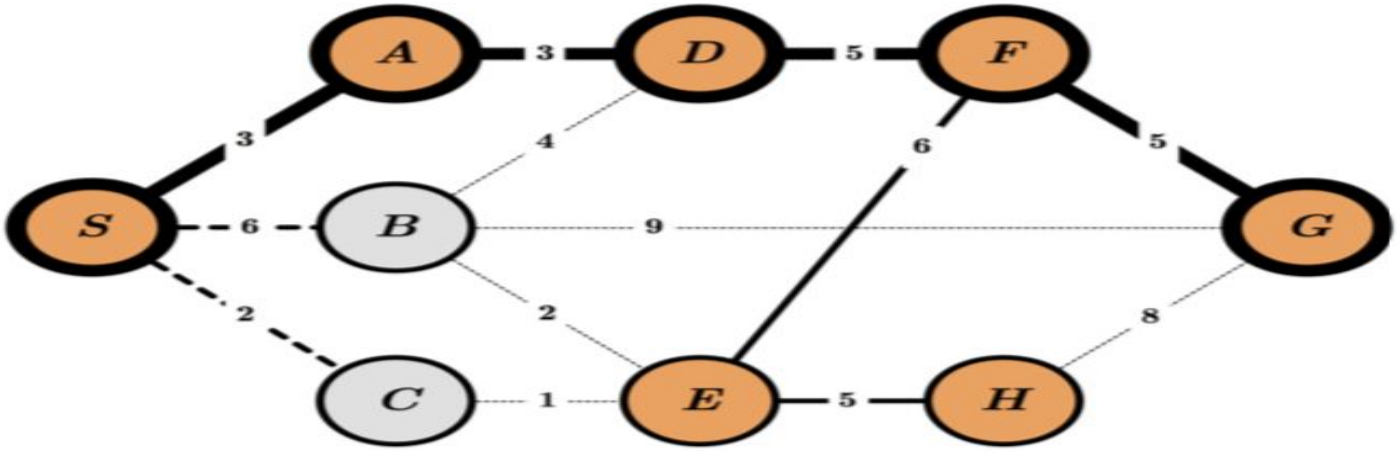
Queue:

S	A	B	C	D	E	G	F	H

Order of Visit:

S *A* *B* *C* *D* *E* *G*

Đáp án bài tập 1: DFS



Stack:

S	C	B	A	D	F	G	E	H

Order of Visit:

S A D F E H G

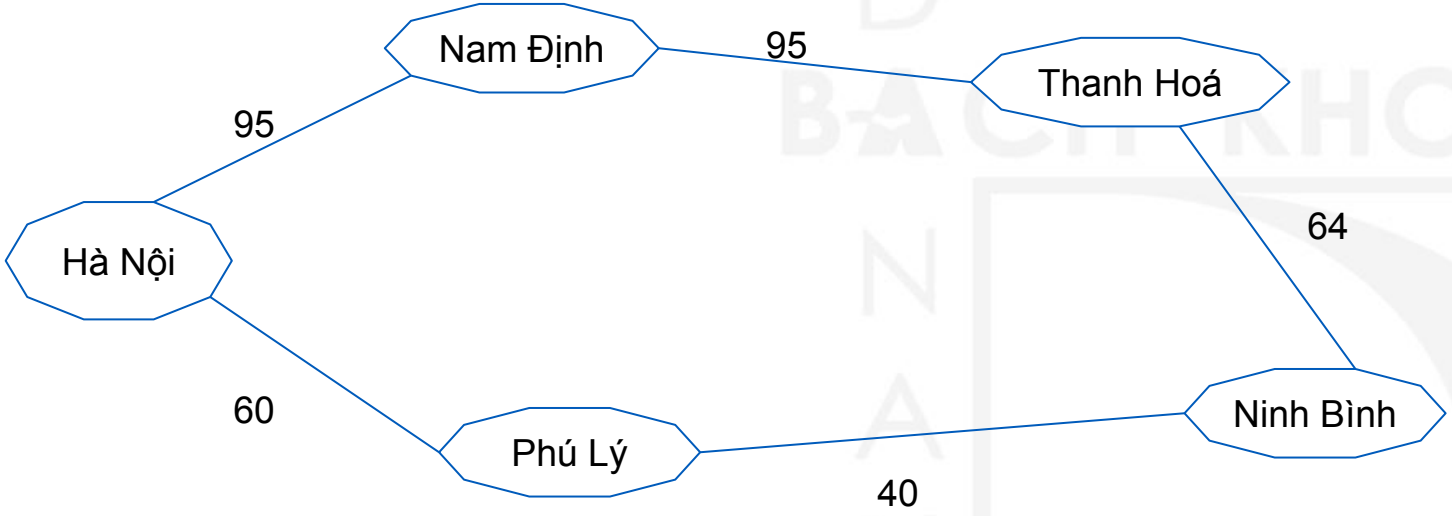
Tìm kiếm đều giá (Uniform-cost search)

Tìm kiếm đều giá

- Tương như tìm kiếm theo chiều rộng nếu các đỉnh có "giá" như nhau
- Ý tưởng: xét đỉnh có "giá" nhỏ nhất trước
- Cài đặt: sử dụng hàng đợi có sự ưu tiên về "giá"
- Cải tiến BFS: ưu tiên chi phí, không ưu tiên độ sâu.
- Mỗi đỉnh n viếng thăm có hàm chi phí bé nhất g(n)

Tìm kiếm đều giá

- Mỗi đỉnh n viếng thăm có hàm chi phí bé nhất g(n)



- Từ Hà Nội đến Thanh Hoá. Sử dụng BFS, thì cho đường đi Hà Nội - Nam Định - Thanh Hoá, nhưng sử dụng UCS, thì cho đường đi Hà Nội - Phú Lý - Ninh Bình - Thanh Hóa với khoảng cách ngắn hơn.

Tìm kiếm đều giá

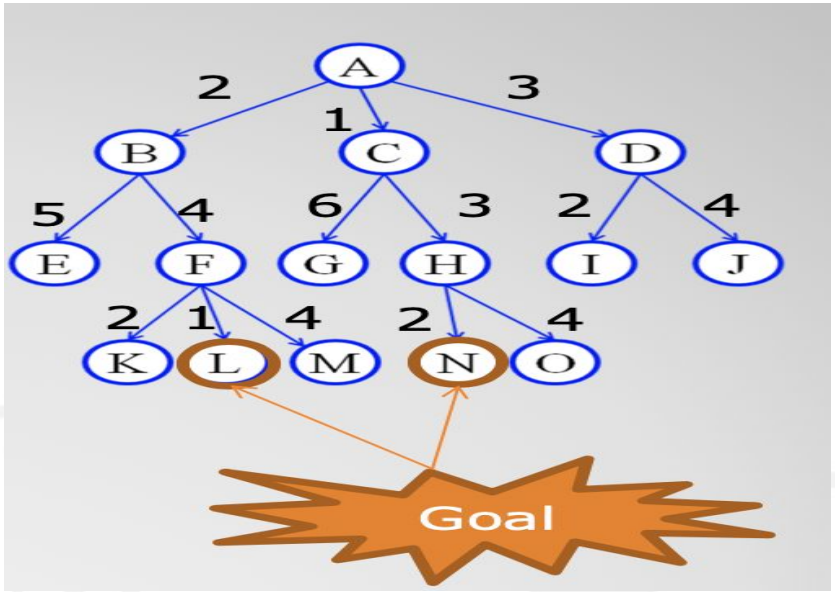
- Mỗi đỉnh n viếng thăm có hàm chi phí bé nhất g(n)
- Thuật toán tìm kiếm đều giá được mô tả bởi thủ tục:

```
function UNIFORM-COST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Cập nhật "giá"

Tìm kiếm đều giá

- Hãy làm rõ quá trình biến đổi của danh sách **frontier ?**



frontier = {A(0)}
frontier = {B(2), C(1),D(3)}
frontier = {B(2), G(1+6),H(1+3),D(3)}
frontier = {E(2+5),F(2+4),G(7),H(4),D(3)}

frontier = …

```
function UNIFORM-COST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```