Trí Tuệ Nhân Tạo

(Artificial Intelligence)

Lê Thanh Hương

Viện Công nghệ thông tin và Truyền thông
Trường Đại Học Bách Khoa Hà Nội

Nội dung môn học

Chương 1. Tổng quan

Chương 2. Tác tử thông minh

Chương 3. Giải quyết vấn đề

- 3.1. Tìm kiếm cơ bản
- 3.2. Tìm kiếm với tri thức bổ sung
- 3.3. Tìm kiếm dựa trên thỏa mãn ràng buộc

Chương 4. Tri thức và suy diễn

Chương 5. Học máy

Giải quyết vấn đề bằng tìm kiếm

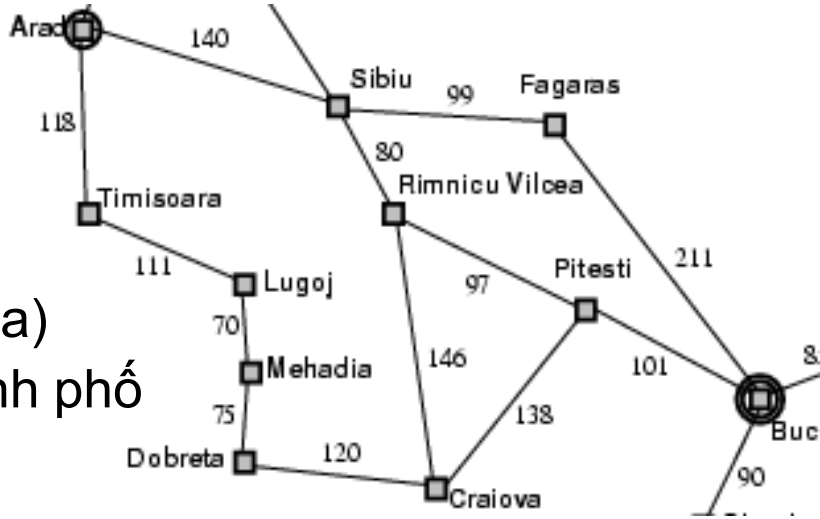
- Giải quyết vấn đề bằng tìm kiếm
  - Tìm chuỗi các hành động cho phép đạt đến (các) trạng thái mong muốn
- Các bước chính
  - Xác định **mục tiêu** cần đạt đến (goal formulation)
    - Là một tập hợp của các trạng thái (đích)
    - Dựa trên: trạng thái hiện tại (của môi trường) và đánh giá hiệu quả hành động (của tác tử)
  - Phát biểu **bài toán** (problem formulation)
    - Với một mục tiêu, xác định các *hành động* và *trạng thái* cần xem xét
  - Quá trình **tìm kiếm** (search process)
    - Xem xét các chuỗi hành động có thể
    - Chọn chuỗi hành động tốt nhất
- Giải thuật tìm kiếm
  - Đầu vào: một bài toán (cần giải quyết)
  - Đầu ra: một giải pháp, dưới dạng một chuỗi các hành động cần thực hiện

Tác tử giải quyết vấn đề

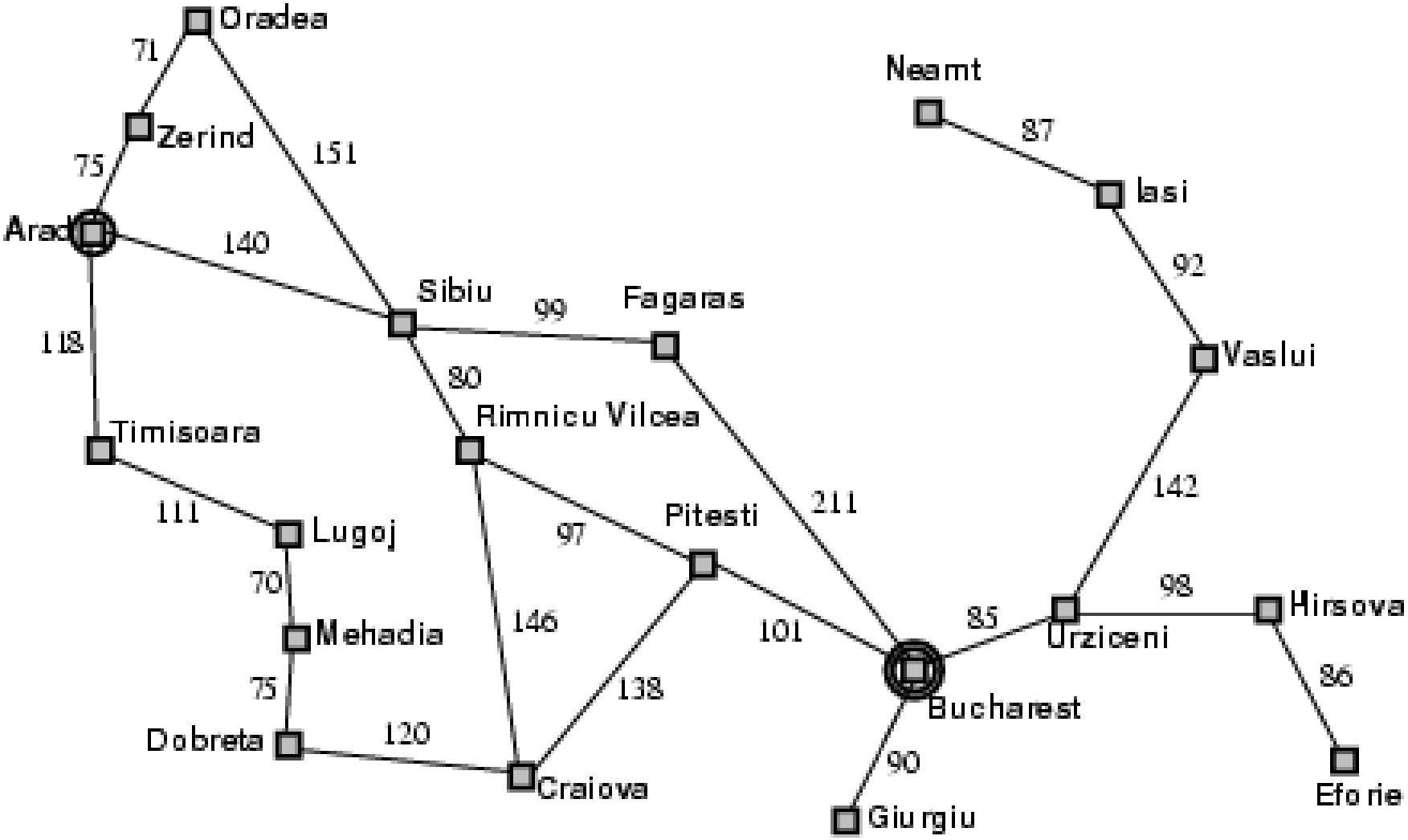
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Giải quyết vấn đề bằng tìm kiếm

- Một người du lịch đang trong chuyến đi du lịch ở Rumani
  - Anh ta hiện thời đang ở Arad
  - Ngày mai, anh ta có chuyến bay khởi hành từ Bucharest
  - Bây giờ, anh ta cần di chuyển (lái xe) từ Arad đến Bucharest
- Phát biểu **mục tiêu**:
  - Cần phải có mặt ở Bucharest
- Phát biểu **bài toán**:
  - Các *trạng thái*: các thành phố (đi qua)
  - Các *hành động*: lái xe giữa các thành phố
- **Tìm kiếm** giải pháp:
  - Chuỗi các thành phố cần đi qua, ví dụ: Arad, Sibiu, Fagaras, Bucharest



Giải quyết vấn đề bằng tìm kiếm

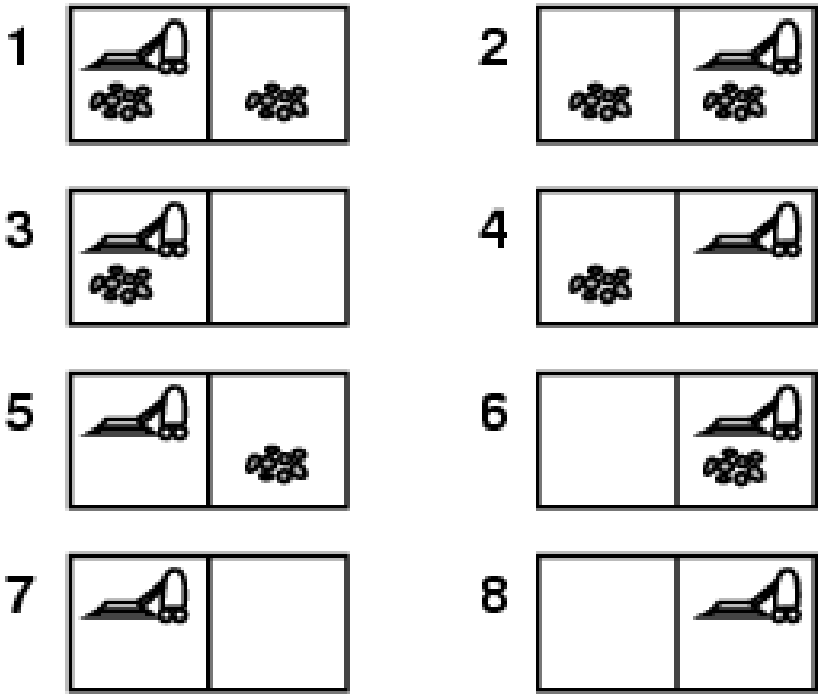


Các kiểu bài toán

- **Xác định, có thể quan sát hoàn toàn** → Bài toán trạng thái đơn
  - Tác tử biết chính xác trạng thái tiếp theo mà nó sẽ chuyển qua
  - Giải pháp của bài toán: một chuỗi hành động
- **Không quan sát được** → Bài toán thiếu cảm nhận
  - Tác tử có thể không biết là nó đang ở trạng thái nào
  - Giải pháp của bài toán: một chuỗi hành động
- **Không xác định** và/hoặc **có thể quan sát một phần** → Bài toán có sự kiện ngẫu nhiên
  - Các nhận thức cung cấp các thông tin mới về trạng thái hiện tại
  - Giải pháp của bài toán: một kế hoạch (chính sách)
  - Thường kết hợp đan xen giữa: tìm kiếm và thực hiện
- **Không biết về không gian trạng thái** → Bài toán thăm dò

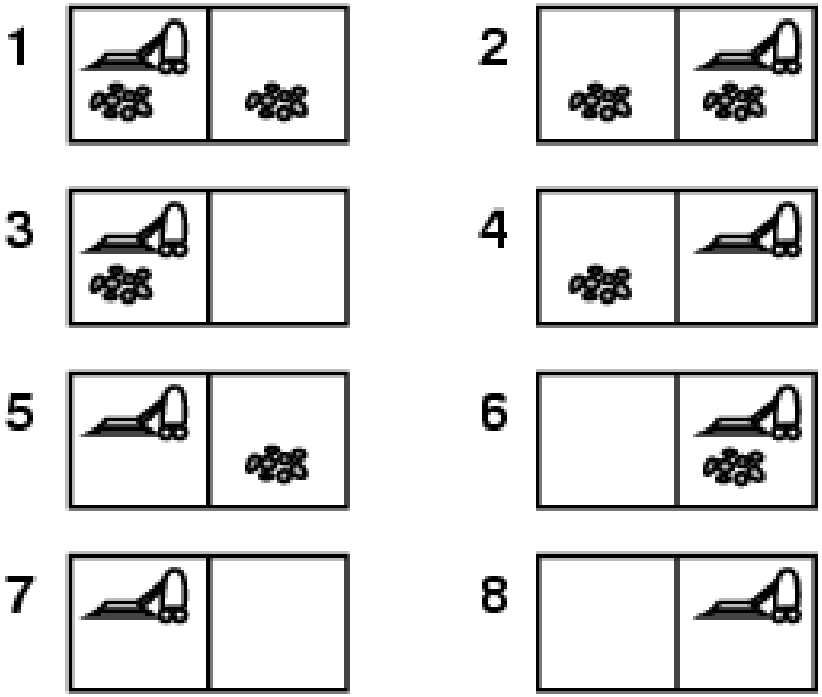
Ví dụ: Bài toán máy hút bụi (1)

- Nếu là **bài toán trạng thái đơn**
  - Bắt đầu ở trạng thái #5.
- Giải pháp?



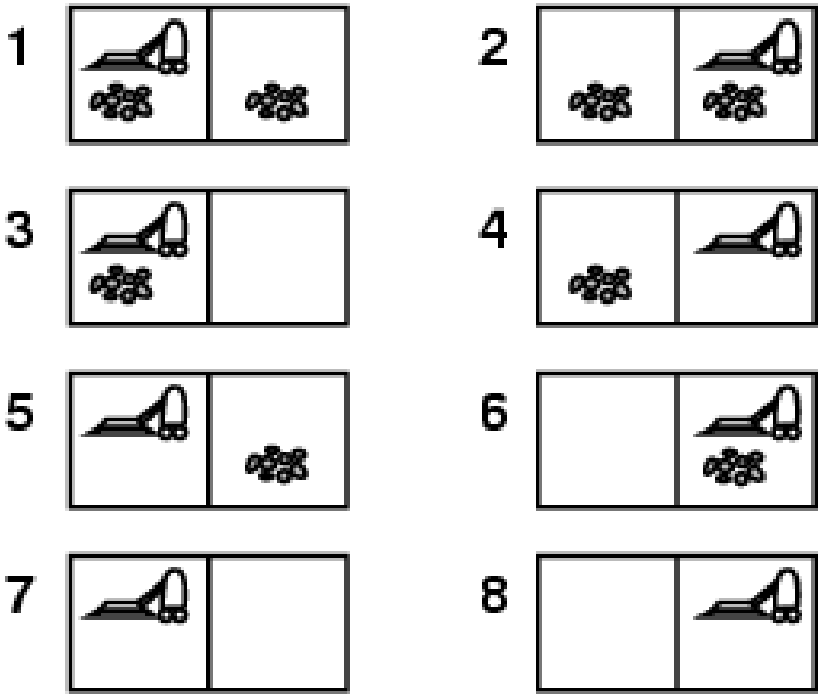
Ví dụ: Bài toán máy hút bụi (2)

- Nếu là **bài toán trạng thái đơn**
  - Bắt đầu ở trạng thái #5.
- Giải pháp?
  - *[Sang phải, Hút bụi]*



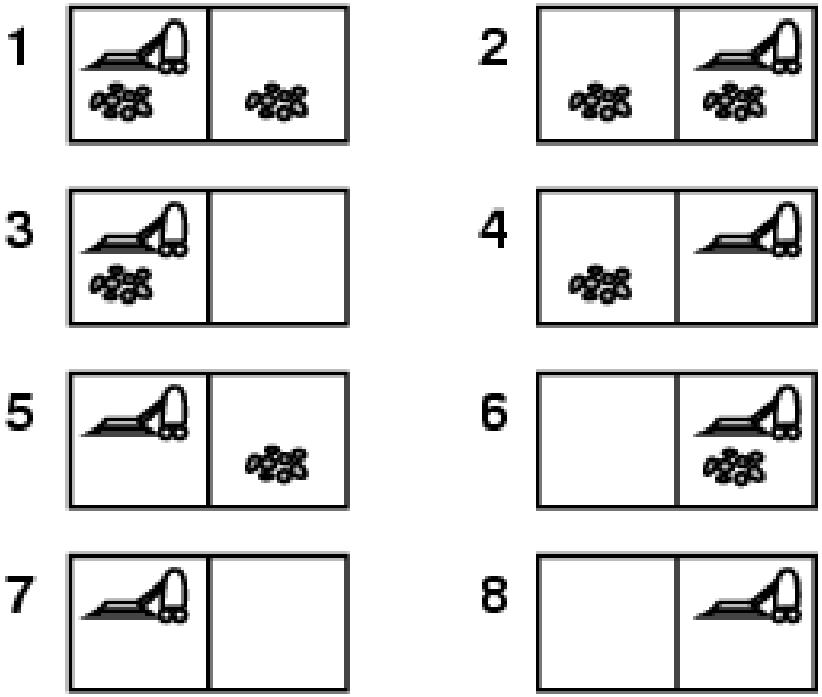
Ví dụ: Bài toán máy hút bụi (3)

- Nếu là **bài toán thiếu cảm nhận**
  - Bắt đầu (có thể) ở trạng thái {#1,#2,#3,#4,#5,#6,#7,#8}
  - Luôn bắt đầu bằng di chuyển sang phải
- Giải pháp?



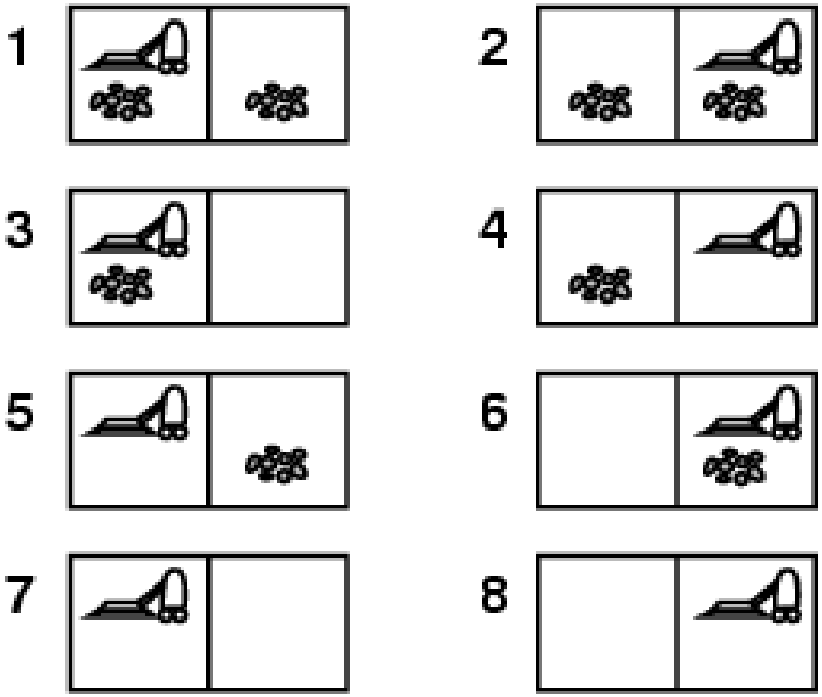
Ví dụ: Bài toán máy hút bụi (4)

- Nếu là **bài toán thiếu cảm nhận**
  - Bắt đầu (có thể) ở trạng thái {#1,#2,#3,#4,#5,#6,#7,#8}
  - Luôn bắt đầu bằng di chuyển sang phải
- Giải pháp?
  - *[Sang phải, Hút bụi, Sang trái, Hút bụi]*



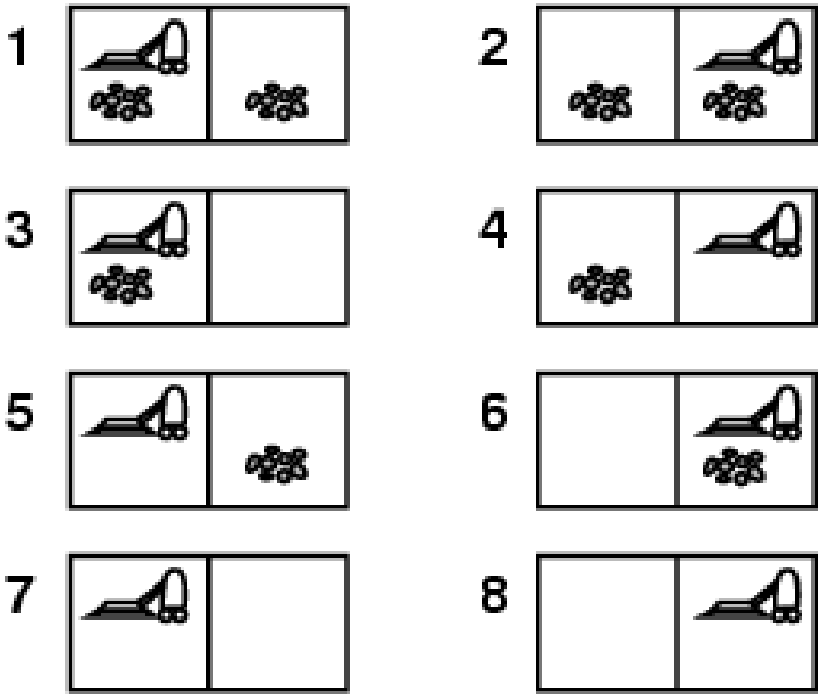
Ví dụ: Bài toán máy hút bụi (5)

- Nếu là **bài toán có sự kiện ngẫu nhiên**
  - Bắt đầu ở trạng thái #5
  - Không xác định: Hút bụi có thể làm bẩn một cái thảm sạch!
  - Có thể quan sát một phần: vị trí, mức độ bẩn ở vị trí hiện thời
- Giải pháp?



Ví dụ: Bài toán máy hút bụi (6)

- Nếu là **bài toán có sự kiện ngẫu nhiên**
  - Bắt đầu ở trạng thái #5
  - Không xác định: Hút bụi có thể làm bẩn một cái thảm sạch!
  - Có thể quan sát một phần: vị trí, mức độ bẩn ở vị trí hiện thời
- Giải pháp?
  - *[Sang phải, **if** Bẩn **then** Hút bụi]*



Phát biểu bài toán trạng thái đơn

Bài toán được định nghĩa bởi 4 thành phần:

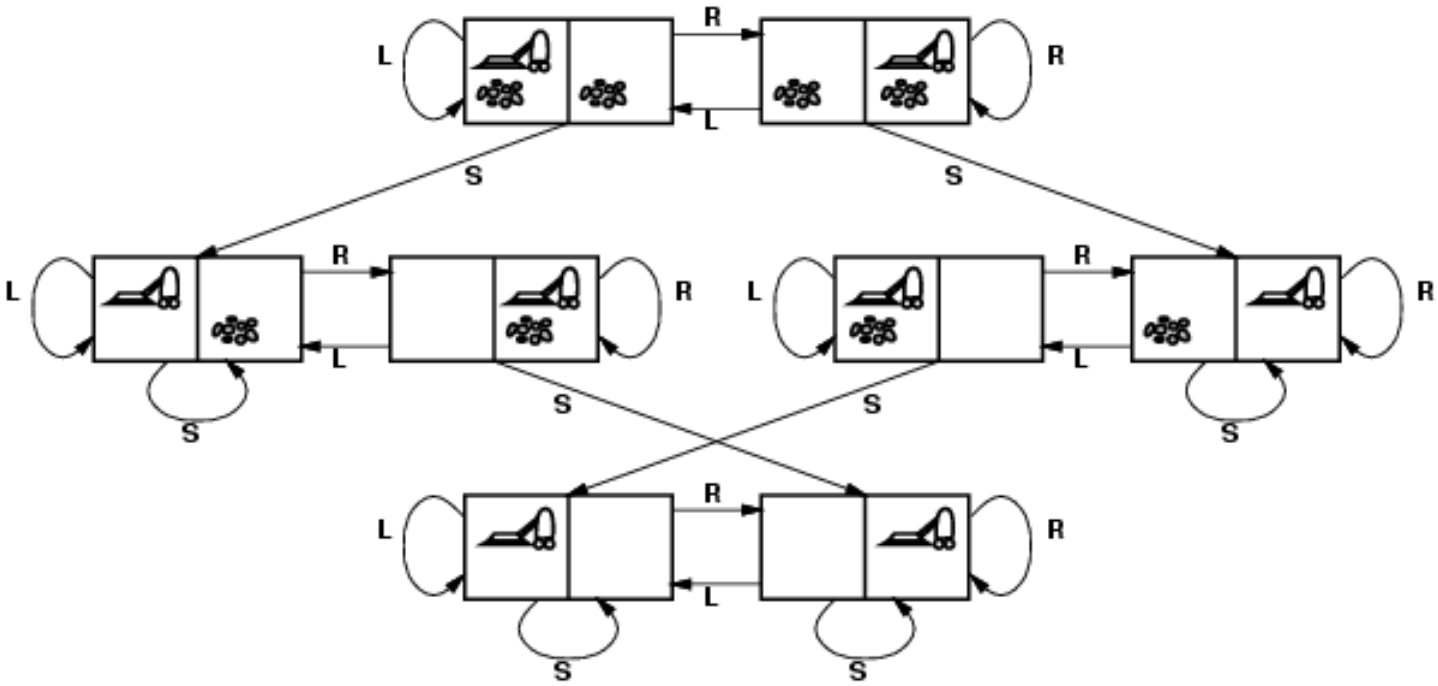
- Trạng thái đầu
  - Ví dụ: "đang ở thành phố Arad"
- Các hành động – Xác định bởi hàm chuyển trạng thái:
  S(trạng_thái_hiện_thời) = tập các cặp <hành_động, trạng_thái_tiếp_theo>
  - Ví dụ: *S(Arad)* = {<*Arad → Zerind*, *Zerind*>, … }
- Kiểm tra mục tiêu, có thể là
  - Trực tiếp – ví dụ: Trạng thái hiện thời *x* = "Đang ở thành phố Bucharest"
  - Gián tiếp – ví dụ: *HếtCờ(x)*, *Sạch*(x), …
- Chi phí đường đi (giải pháp)
  - Ví dụ: Tổng các khoảng cách, Số lượng các hành động phải thực hiện,…
  - *c(x,a,y)* ≥ 0 là chi phí bước (bộ phận) – chi phí cho việc áp dụng hành động *a* để chuyển từ trạng thái *x* sang trạng thái *y*
- Một giải pháp: Một chuỗi các hành động cho phép dẫn từ trạng thái đầu đến trạng thái đích

Xác định không gian trạng thái

- Các bài toán thực tế thường được mô tả phức tạp
  → Không gian trạng thái cần được khái quát (abstracted) để phục vụ cho việc giải quyết bài toán
- Trạng thái (khái quát) = Một tập các trạng thái thực tế
- Hành động (khái quát) = Một kết hợp phức tạp của các hành động thực tế
  - Ví dụ: Hành động "Arad → Zerind" biểu diễn một tập kết hợp các đường, đường vòng, chỗ nghỉ, …
- Để đảm bảo việc thực hiện (quá trình tìm kiếm), bất kỳ trạng thái thực tế nào cũng phải có thể đạt đến được từ trạng thái thực tế khác
- Giải pháp (khái quát) = Một tập các đường đi giải pháp trong thực tế

Đồ thị không gian trạng thái (1)

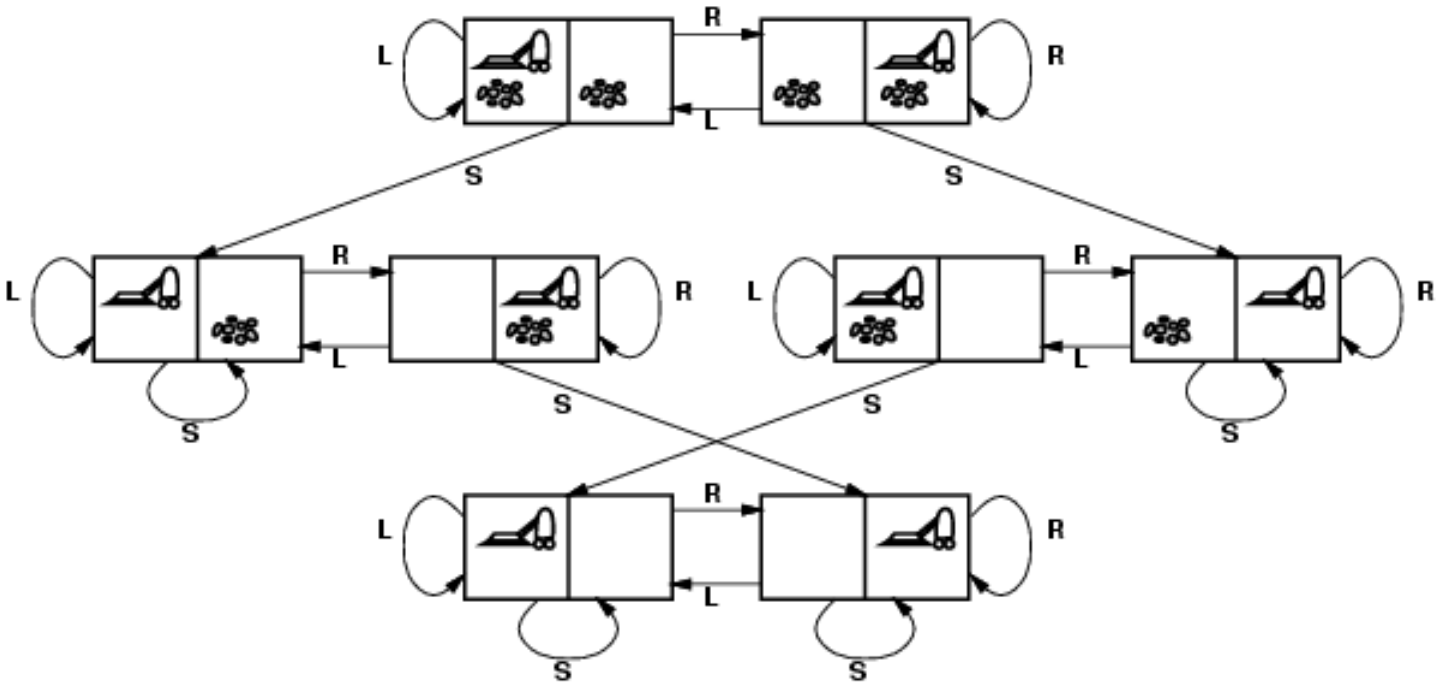
Bài toán máy hút bụi



- Các trạng thái?
- Các hành động?
- Kiểm tra mục tiêu?
- Chi phí đường đi?

Đồ thị không gian trạng thái (2)

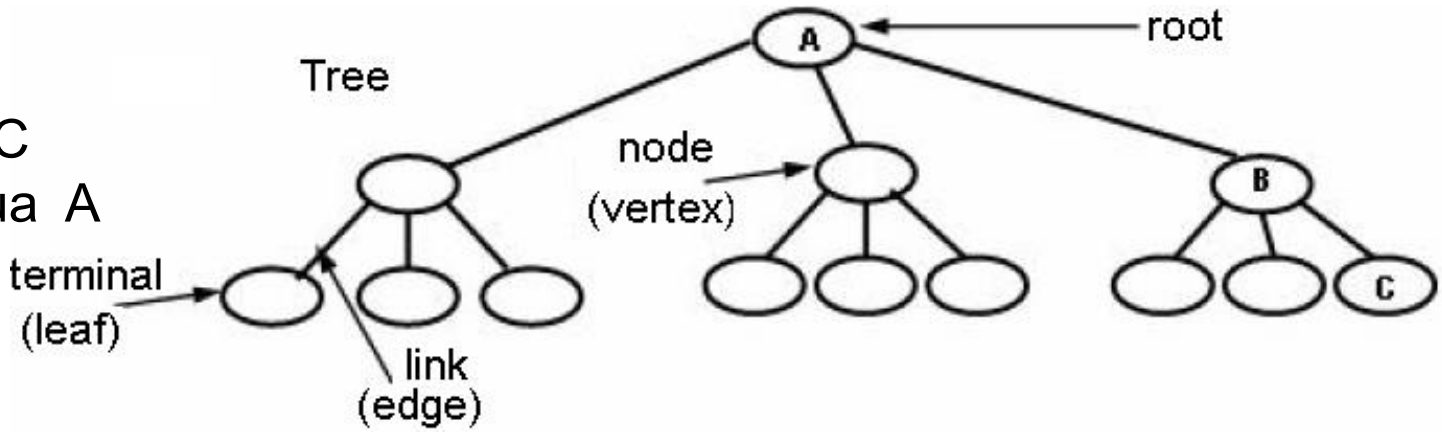
Bài toán máy hút bụi

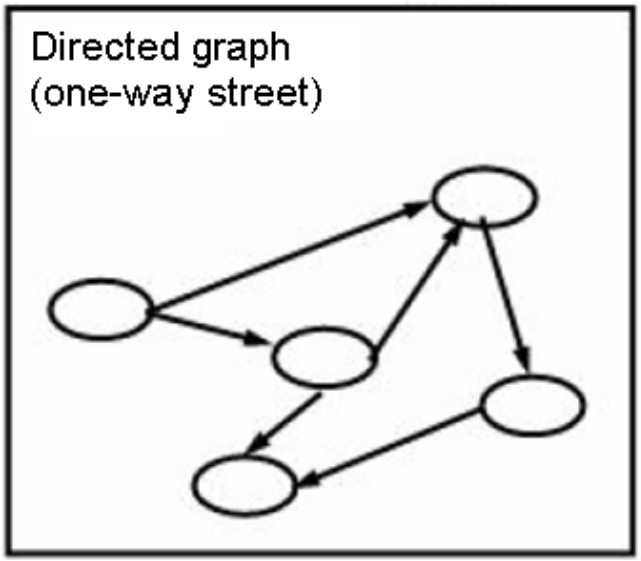


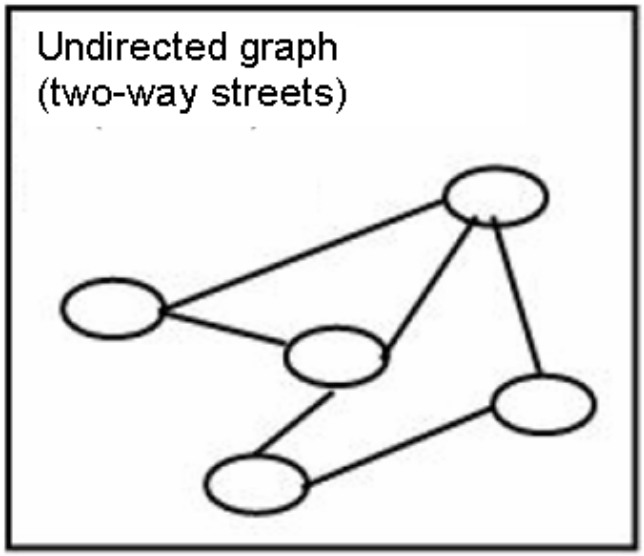
- Các trạng thái? *Chỗ bẩn và vị trí máy hút bụi*
- Các hành động? *Sang trái*, *sang phải*, *hút bụi*, *không làm gì*
- Kiểm tra mục tiêu? *Không còn chỗ (vị trí) nào bẩn*
- Chi phí đường đi? *1 (mỗi hành động)*, *0 (không làm gì cả)*

Biểu diễn bằng cây và đồ thị

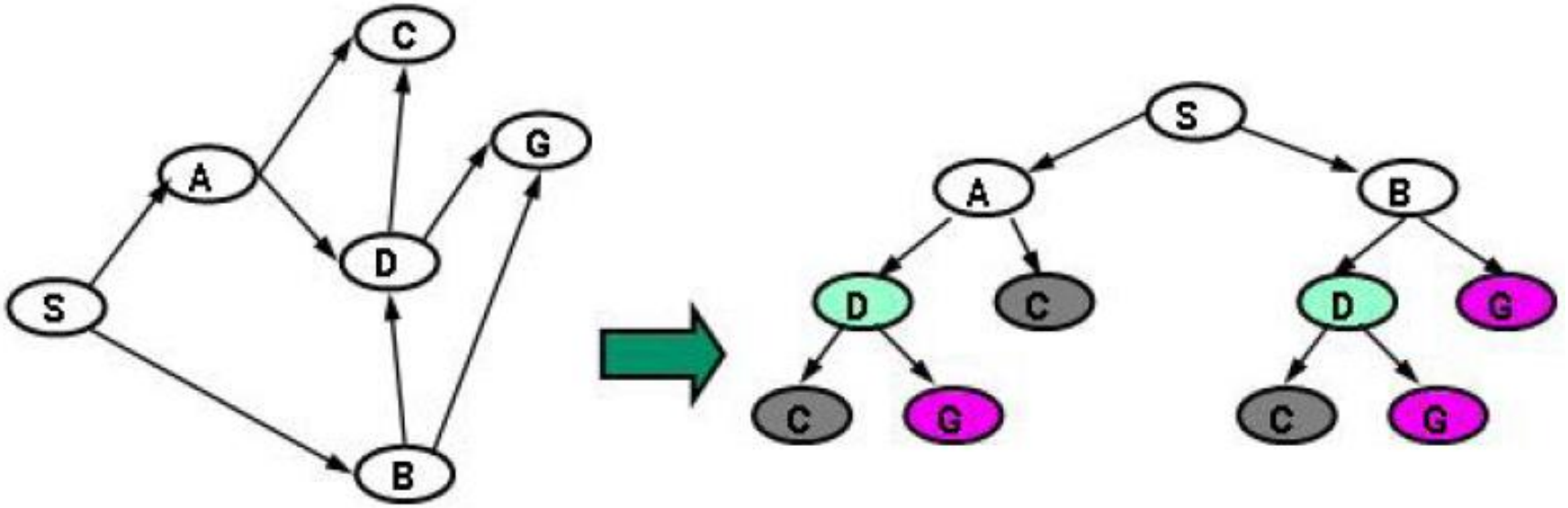
B là cha của C
C là con của B
A là tổ tiên của C
C là con cháu của A







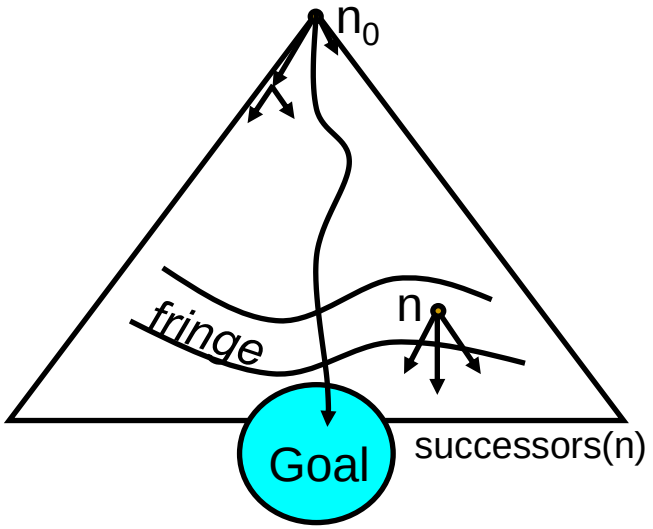
Đồ thị tìm kiếm → Cây tìm kiếm



- Các bài toán tìm kiếm trên đồ thị có thể được chuyển thành các bài toán tìm kiếm trên cây
  - Thay thế mỗi liên kết (cạnh) vô hướng bằng 2 liên kết (cạnh) có hướng
  - Loại bỏ các vòng lặp tồn tại trong đồ thị (để tránh không duyệt 2 lần đối với một nút trong bất kỳ đường đi nào)

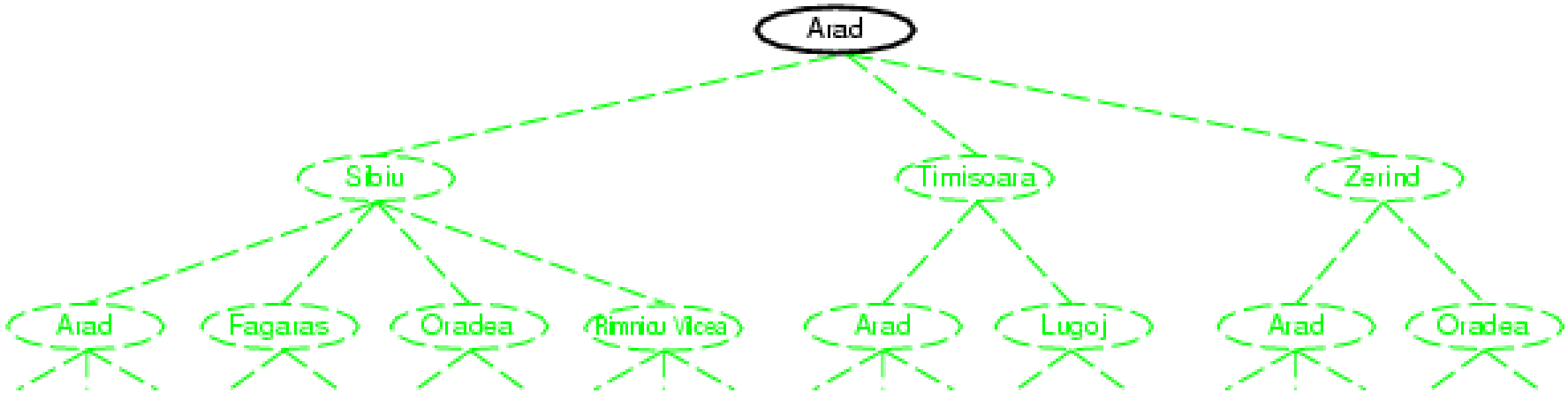
Các giải thuật tìm kiếm theo cấu trúc cây

- Ý tưởng:
  - Khám phá (xét) không gian trạng thái bằng cách sinh ra các trạng thái kế tiếp của các trạng thái đã khám phá (đã xét)
  - Còn gọi là phương pháp khai triển (phát triển) các trạng thái

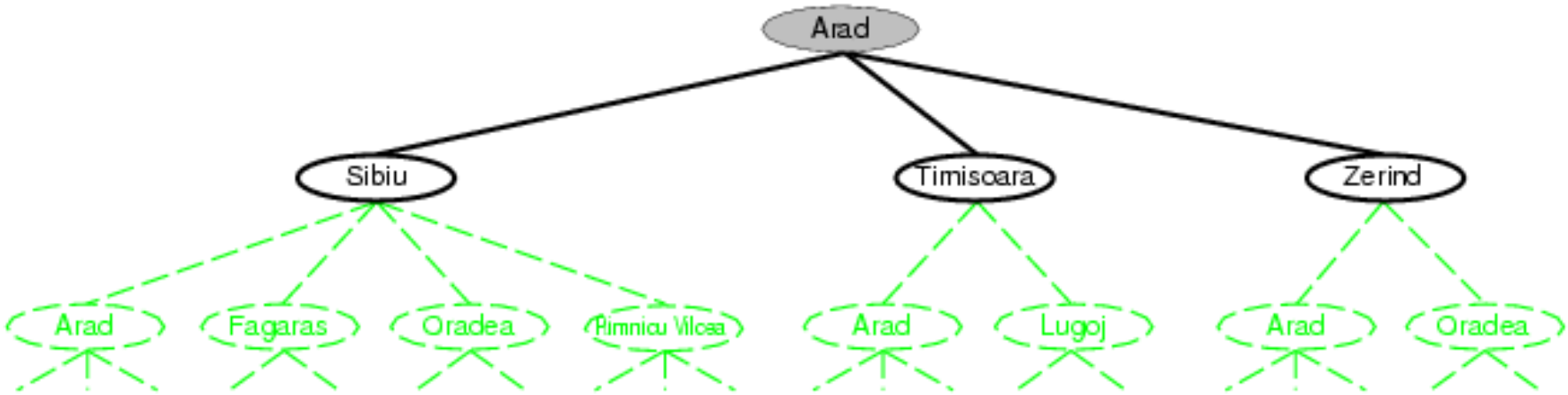


function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree

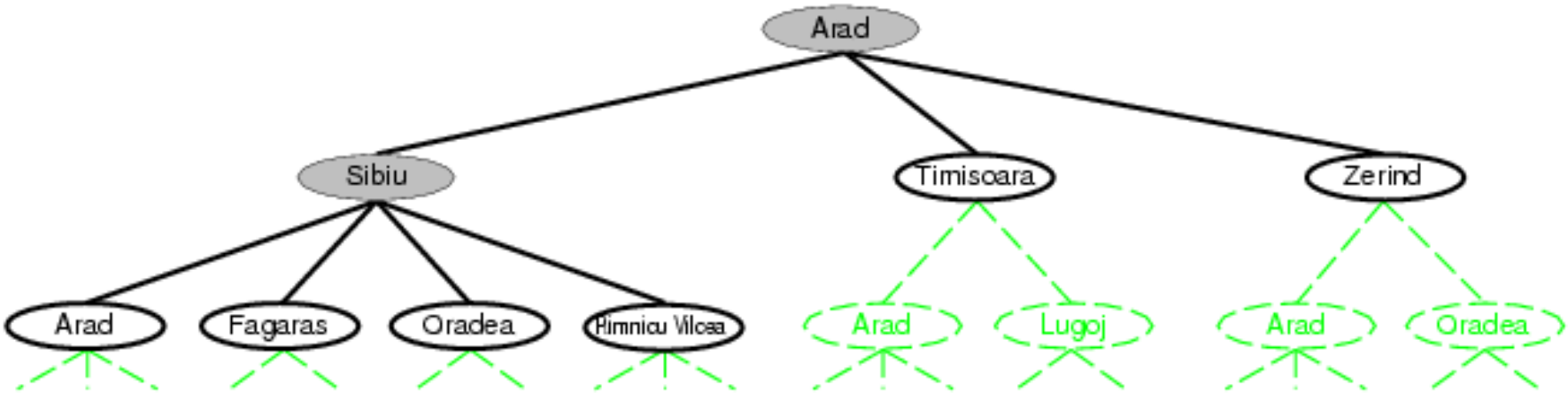
Ví dụ biểu diễn theo cấu trúc cây (1)



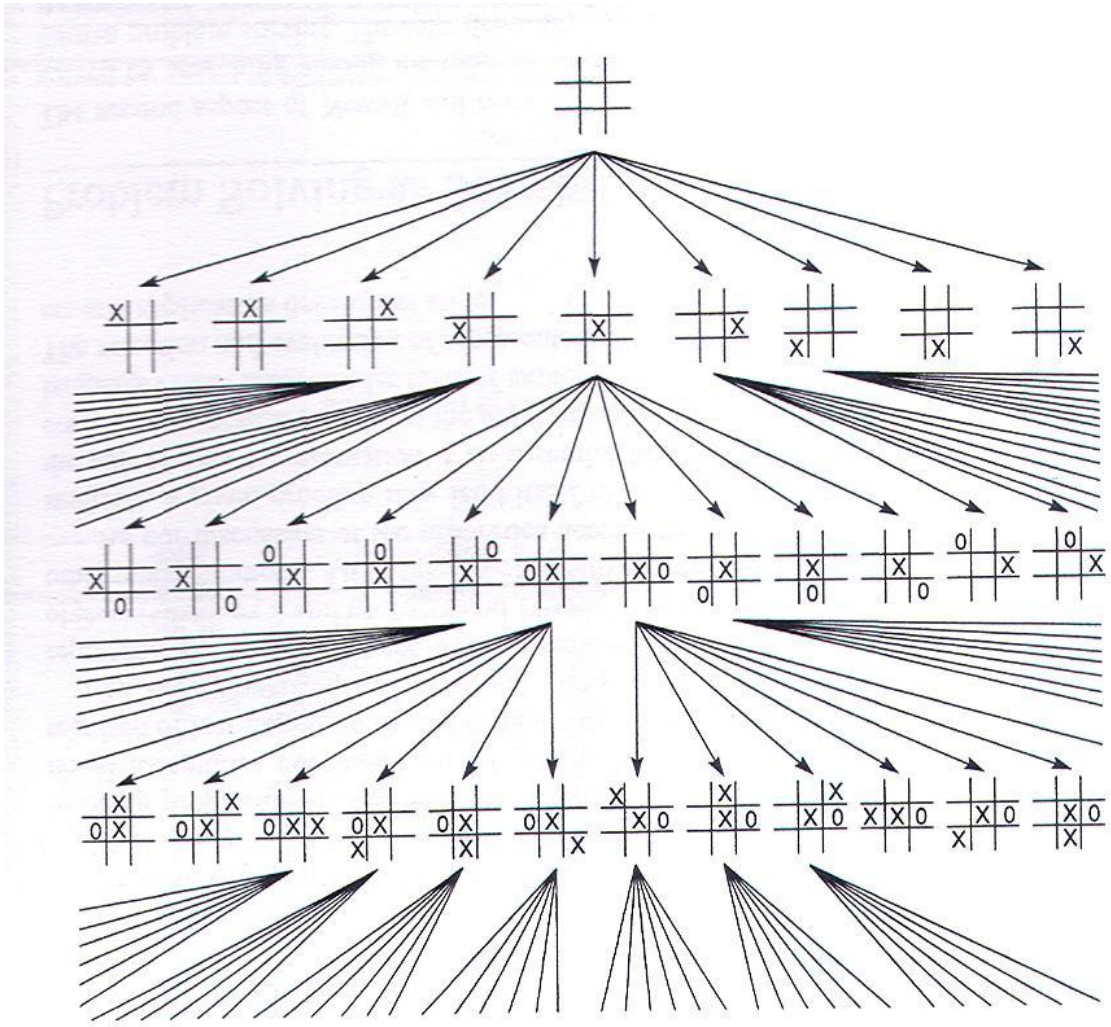
Ví dụ biểu diễn theo cấu trúc cây (2)



Ví dụ biểu diễn theo cấu trúc cây (3)



Ví dụ trò chơi cờ ca-rô (Tic-Tac-Toe)

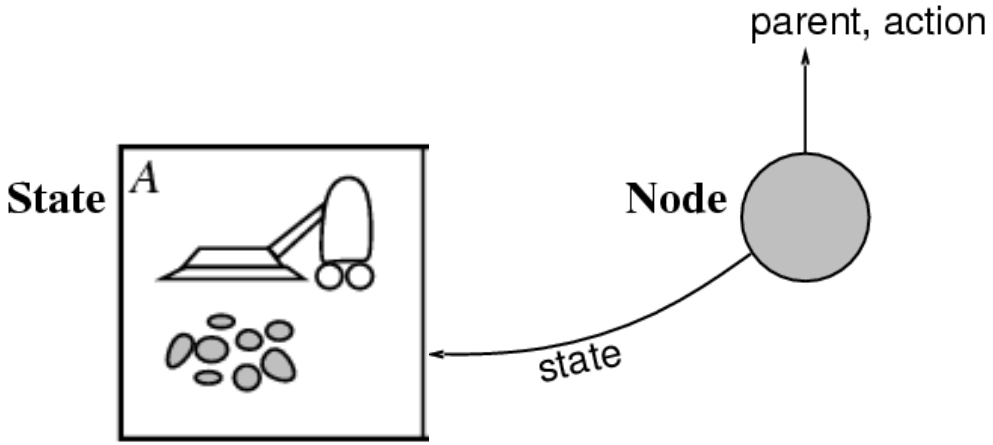


Tìm kiếm theo cấu trúc cây - Giải thuật

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

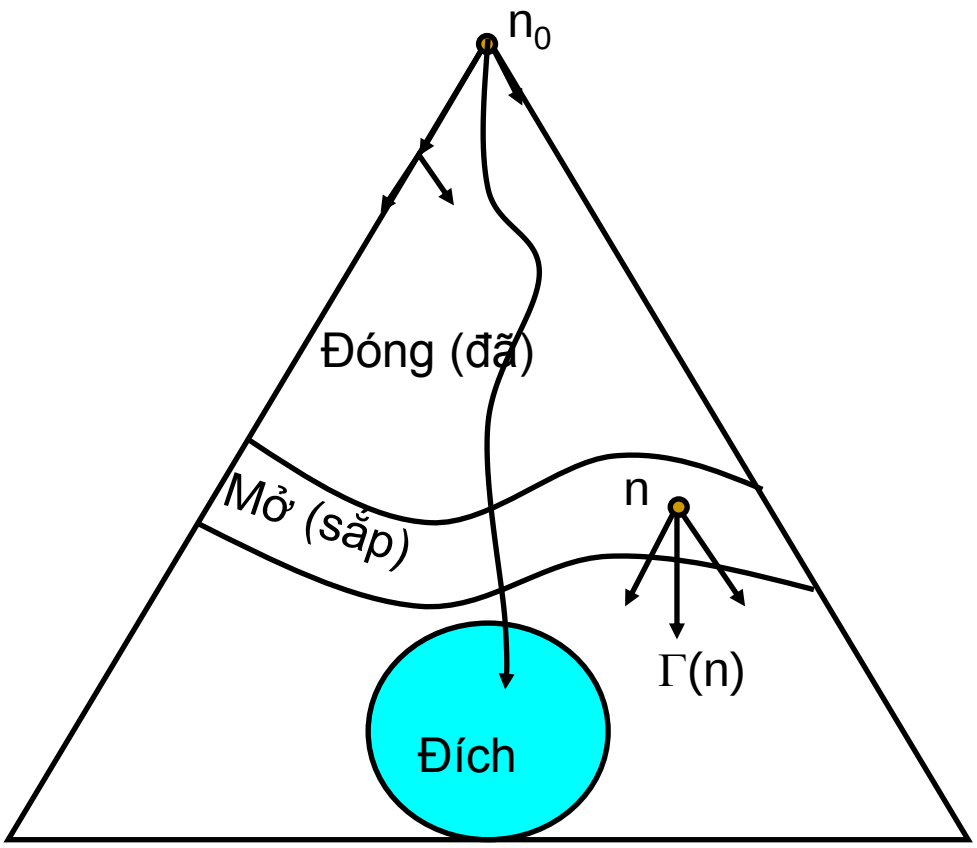
Biểu diễn cây tìm kiếm (1)

- Một *trạng thái* là một biểu diễn của một hình trạng (configuration) thực tế
- Một *nút* (của cây) là một phần cấu thành nên cấu trúc dữ liệu của một cây tìm kiếm
  - Một nút chứa các thuộc tính: *trạng thái*, *nút cha*, *nút con*, *hành động*, *độ sâu*, *chi phí đường đi g(x)*



- Hàm `Expand` tạo nên các nút mới,
  - Gán giá trị cho các thuộc tính (của nút mới)
  - Sử dụng hàm `Successor-Fn` để tạo nên các trạng thái tương ứng với các nút mới đó

Biểu diễn cây tìm kiếm (2)



Các chiến lược tìm kiếm

- Một chiến lược tìm kiếm được xác định bằng việc chọn trình tự phát triển (khai triển) các nút
- Các chiến lược tìm kiếm được đánh giá theo các tiêu chí:
  - *Tính hoàn chỉnh*: Có đảm bảo tìm được một lời giải (nếu thực sự tồn tại một lời giải)?
  - *Độ phức tạp về thời gian*: Số lượng các nút được sinh ra
  - *Độ phức tạp về bộ nhớ*: Số lượng tối đa các nút được lưu trong bộ nhớ
  - *Tính tối ưu*: Có đảm bảo tìm được lời giải có chi phí thấp nhất?
- Độ phức tạp về thời gian và bộ nhớ được đánh giá bởi:
  - *b:* Hệ số phân nhánh tối đa của cây tìm kiếm
  - *d:* Độ sâu của lời giải có chi phí thấp nhất
  - *m*: Độ sâu tối đa của không gian trạng thái (độ sâu của cây) – có thể là ∞

Các chiến lược tìm kiếm cơ bản

- Các chiến lược tìm kiếm cơ bản (uninformed search strategies) **chỉ sử dụng các thông tin chứa trong định nghĩa của bài toán**
  - Tìm kiếm theo chiều rộng (Breadth-first search)
  - Tìm kiếm với chi phí cực tiểu (Uniform-cost search)
  - Tìm kiếm theo chiều sâu (Depth-first search)
  - Tìm kiếm giới hạn độ sâu (Depth-limited search)
  - Tìm kiếm sâu dần (Iterative deepening search)

Tìm kiếm theo chiều rộng – BFS

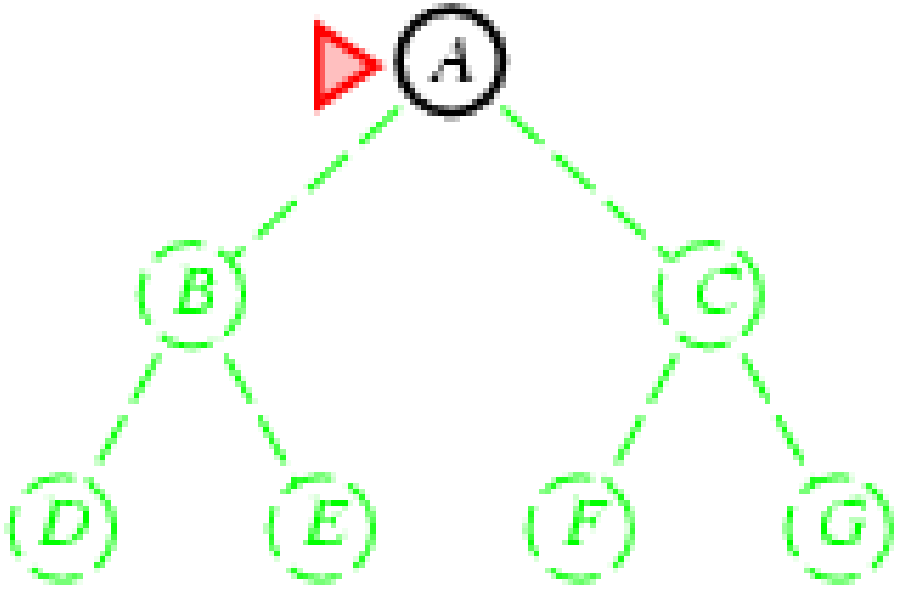
- Phát triển các nút chưa xét theo chiều rộng – Các nút được xét theo thứ tự độ sâu tăng dần
- Cài đặt giải thuật BFS
  - *fringe* là một cấu trúc kiểu hàng đợi FIFO (các nút mới được bổ sung vào cuối của *fringe*)
- Các ký hiệu được sử dụng trong giải thuật BFS
  - *fringe*: Cấu trúc kiểu hàng đợi (queue) lưu giữ các nút (trạng thái) ***sẽ*** được duyệt
  - *closed*: Cấu trúc kiểu hàng đợi (queue) lưu giữ các nút (trạng thái) ***đã*** được duyệt
  - *G=(N,A)*: Cây biểu diễn không gian trạng thái của bài toán
  - n_0: Trạng thái đầu của bài toán (nút gốc của cây)
  - *ĐICH*: Tập các trạng thái đích của bài toán
  - $\Gamma(n)$: Tập các trạng thái (nút) con của trạng thái (nút) đang xét *n*

BFS – Giải thuật

```
BFS (N, A, n0, ĐICH)
{
   fringe ← n0;
   closed ← ∅;
   while  (fringe ≠ ∅)  do
   {     n ← GET_FIRST(fringe);          // lấy phần tử đầu tiên của fringe
         closed ← closed ⊕ n;
         if  (n ∈ ĐICH)  then  return  SOLUTION(n);
         if  (Γ(n) ≠ ∅) then fringe ← fringe ⊕ Γ(n);
   }
   return  ("No solution");
}
```

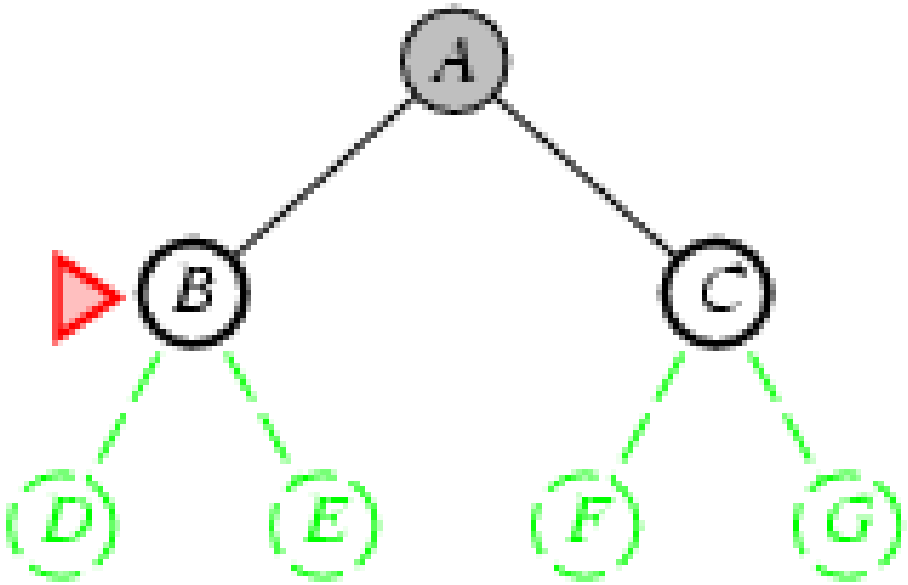
BFS – Ví dụ (1)

- Phát triển các nút chưa xét theo chiều rộng – Các nút được xét theo thứ tự độ sâu tăng dần



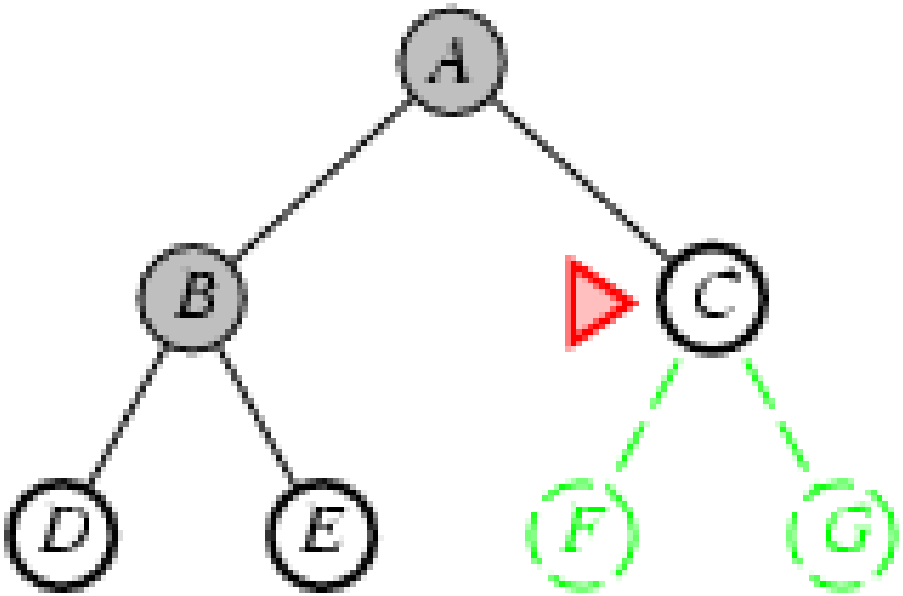
BFS – Ví dụ (2)

- Phát triển các nút chưa xét theo chiều rộng – Các nút được xét theo thứ tự độ sâu tăng dần



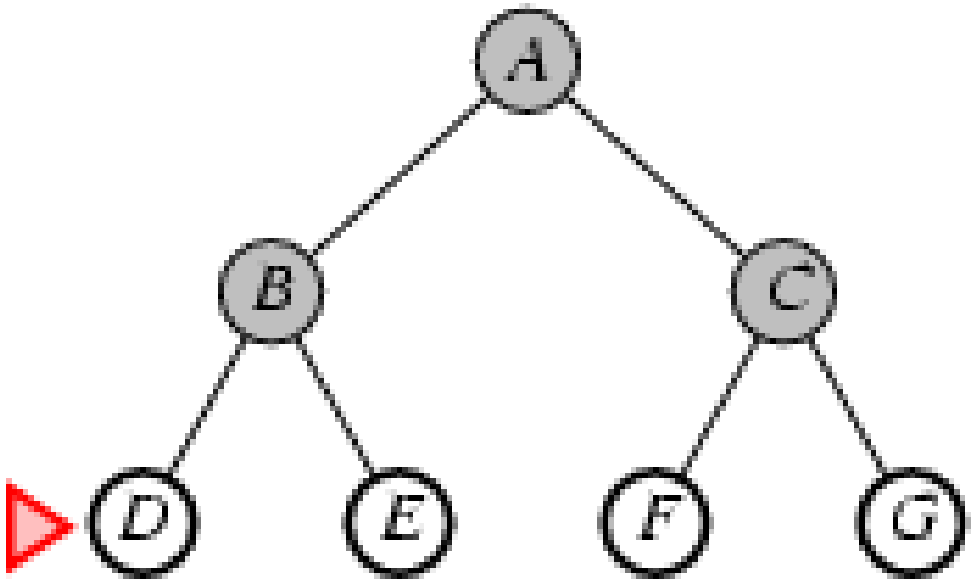
BFS – Ví dụ (3)

- Phát triển các nút chưa xét theo chiều rộng – Các nút được xét theo thứ tự độ sâu tăng dần

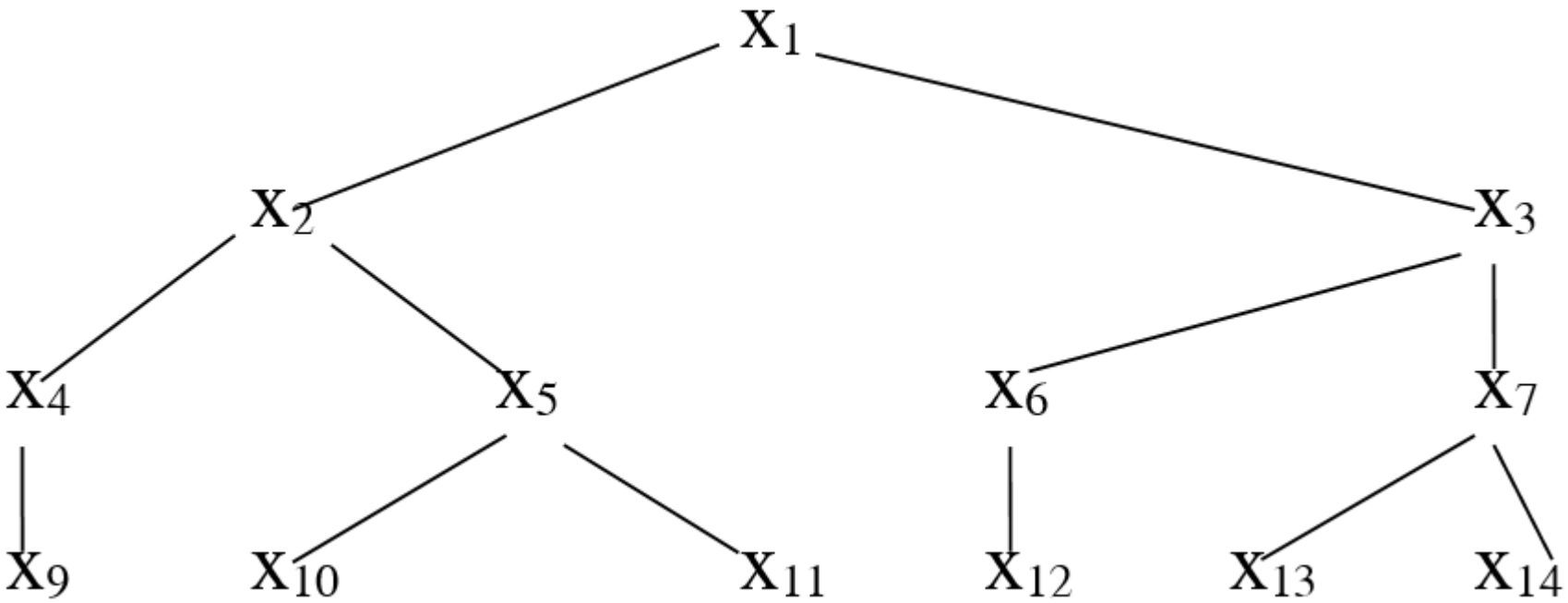


BFS – Ví dụ (4)

- Phát triển các nút chưa xét theo chiều rộng – Các nút được xét theo thứ tự độ sâu tăng dần

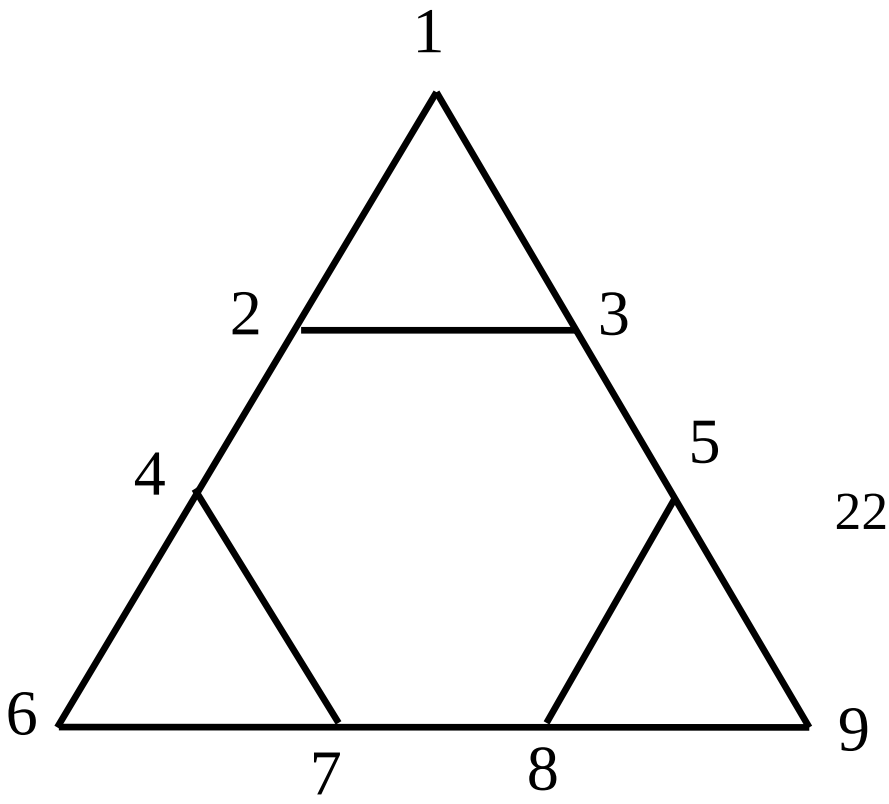


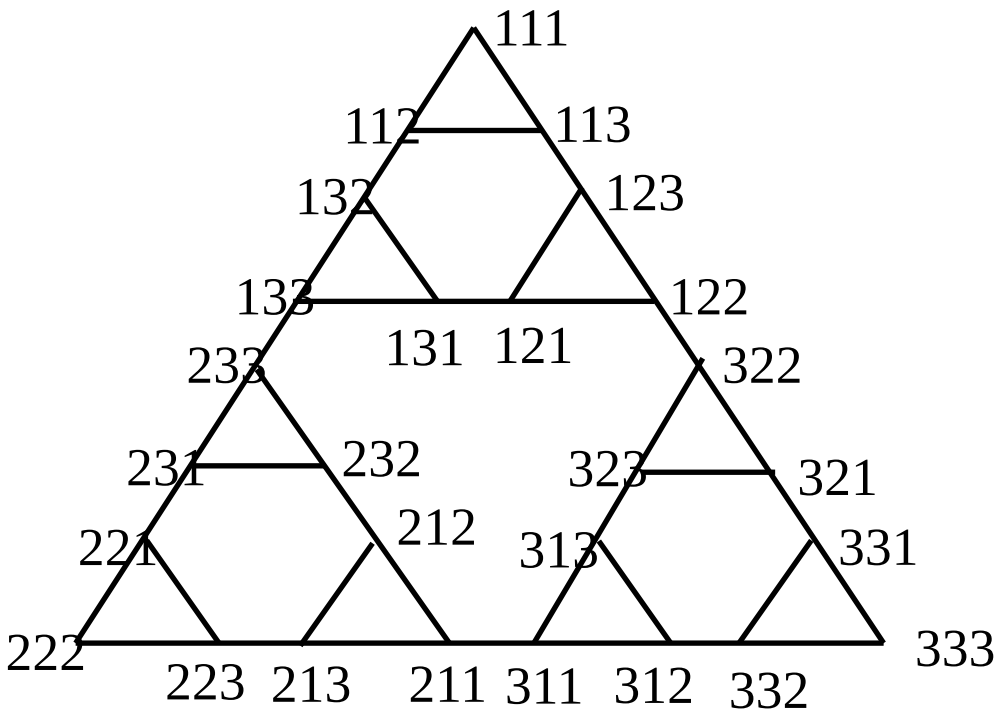
Bài tập 1: Vẽ bảng minh họa quá trình duyệt của thuật toán BFS cho câv tìm kiếm dưới đâv. đỉnh đầu x1. đích x12



n	Γ(n)	Fringe	Closed

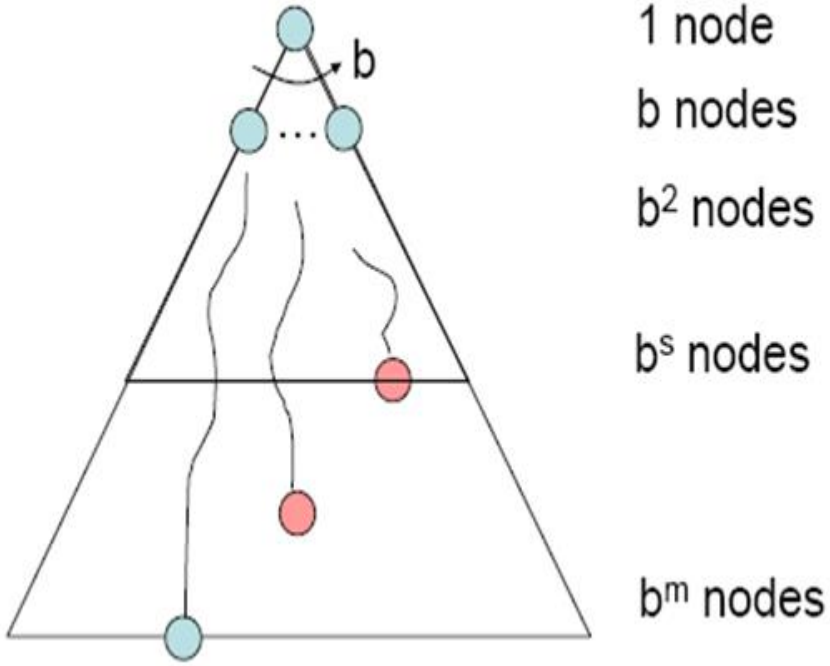
Bài tập 2: Vẽ bảng minh họa quá trình tìm kiếm lời giải trên KGTT của bài toán Tháp Hà nội sử dụng BFS, đỉnh xuất phát 1, đích 9.





BFS – Các đặc điểm

- Tính hoàn chỉnh?
  - Có (nếu *b* là hữu hạn)
- Độ phức tạp về thời gian?
  - $1+b+b^2+b^3+\ldots+b^d = O(b^{d+1})$
- Độ phức tạp về bộ nhớ?
  - $O(b^{d+1})$ – Lưu tất cả các nút trong bộ nhớ)
- Tính tối ưu?
  - Có (nếu chi phí =1 cho mỗi bước)

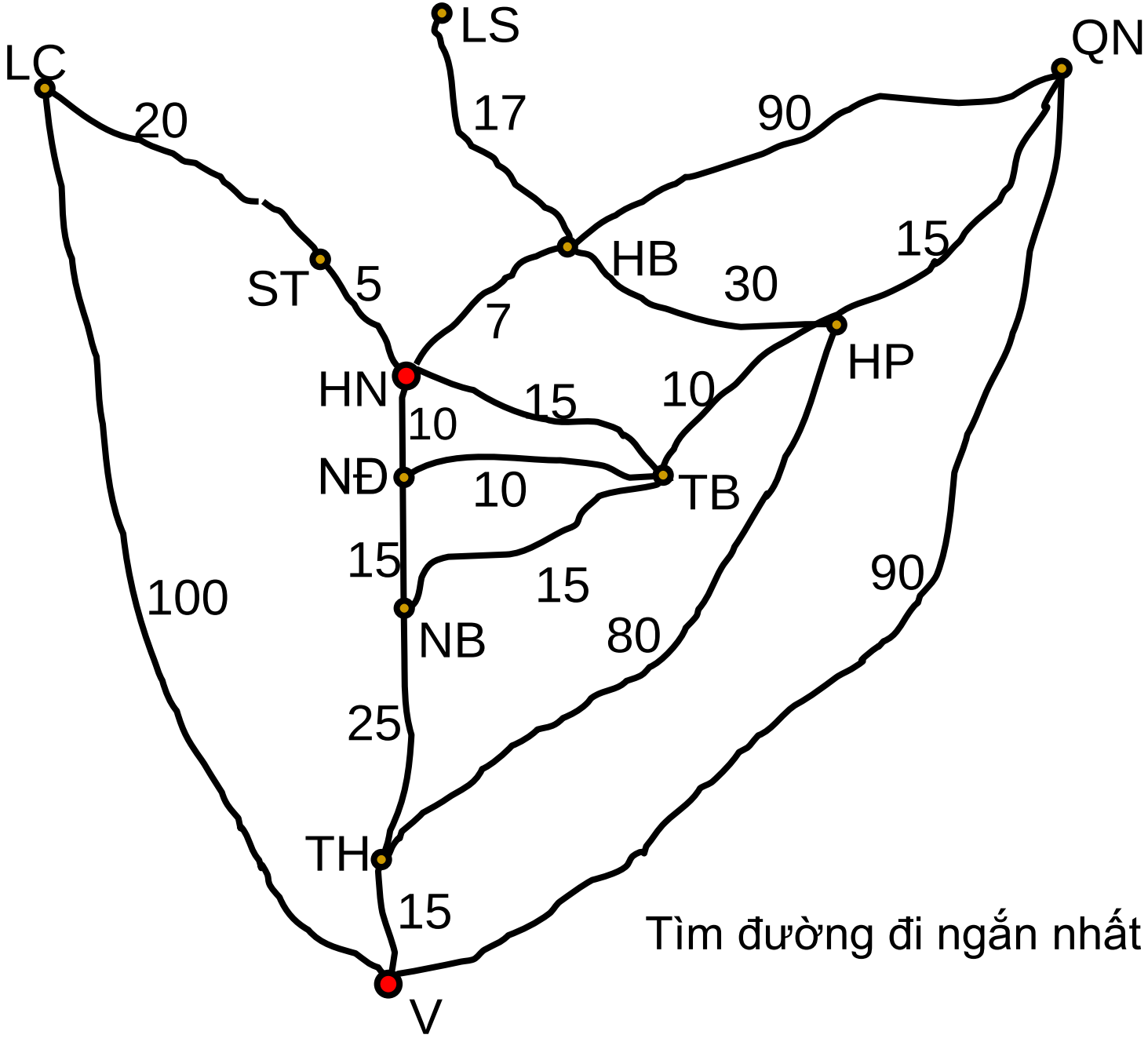


Tìm kiếm với chi phí cực tiểu (UCS)

- Phát triển các nút chưa xét có chi phí thấp nhất – Các nút được xét theo thứ tự chi phí (từ nút gốc đến nút đang xét) tăng dần
- Cài đặt:
  - *fringe* là một cấu trúc hàng đợi, trong đó các phần tử được sắp xếp theo chi phí đường đi
- Trở thành phương pháp tìm kiếm theo chiều rộng, nếu các chi phí ở mỗi bước (mỗi cạnh của cây tìm kiếm) là như nhau

UCS: Giải thuật

```
UCS (N, A, n0, ĐICH, c)
{
   fringe ← n0;
   closed ← ∅;
   while  (fringe ≠ ∅)  do
   {     n ← GET_LOWEST_COST(fringe);        // lấy phần tử có chi phí
                                             // đường đi c(n) nhỏ nhất
         closed ← closed ⊕ n;
         if  (n ∈ ĐICH)  then  return  SOLUTION(n);
         if  (Γ(n) ≠ ∅) then fringe ← fringe ⊕ Γ(n);
   }
   return  ("No solution");
}
```



Tìm đường đi ngắn nhất từ HN đến V

UCS: Các đặc điểm

- Tính hoàn chỉnh?
  - Có (nếu chi phí ở mỗi bước ≥ ε > 0)
- Độ phức tạp về thời gian?
  - Phụ thuộc vào tổng số các nút có chi phí ≤ chi phí của lời giải tối ưu: $O(b^{\lceil C^*/\varepsilon \rceil})$, trong đó C^* là chi phí của lời giải tối ưu
- Độ phức tạp về bộ nhớ?
  - Phụ thuộc vào tổng số các nút có chi phí ≤ chi phí của lời giải tối ưu: $O(b^{\lceil C^*/\varepsilon \rceil})$
- Tính tối ưu?
  - Có (nếu các nút được xét theo thứ tự tăng dần về chi phí *g(n)*)

Tìm kiếm theo chiều sâu (DFS)

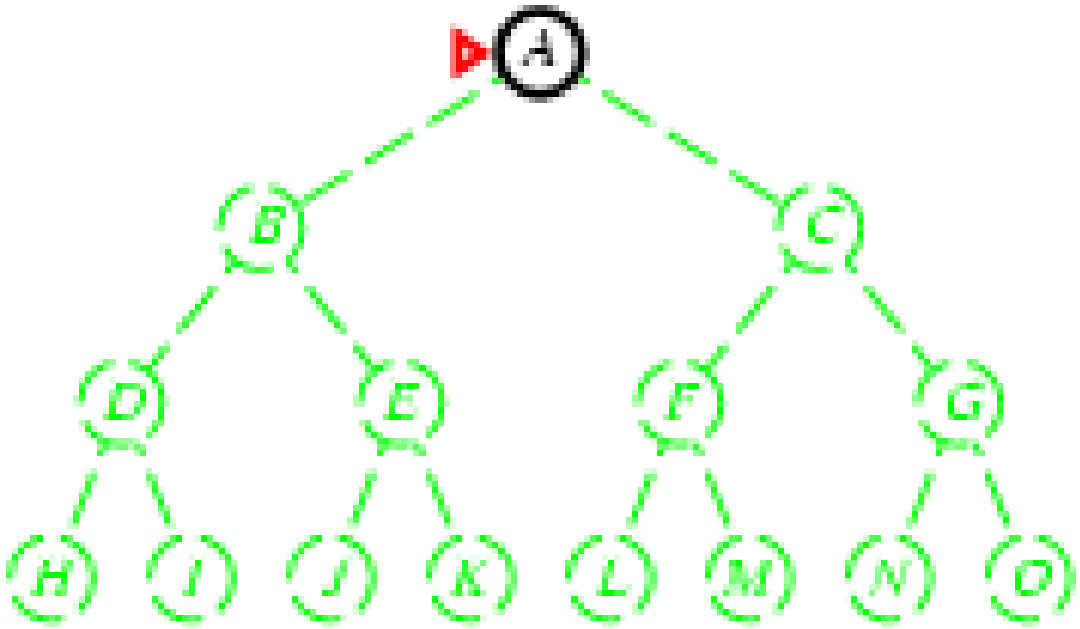
- Phát triển các nút chưa xét theo chiều sâu – Các nút được xét theo thứ tự độ sâu giảm dần
- Cài đặt:
  - *fringe* là một cấu trúc kiểu ngăn xếp LIFO (Các nút mới được bổ sung vào đầu của *fringe*)

DFS: Giải thuật

```
DFS (N, A, n0, ĐICH)
{
   fringe ← n0;
   closed ← ∅;
   while  (fringe ≠ ∅)  do
   {     n ← GET_FIRST(fringe);          // lấy phần tử đầu tiên của fringe
         closed ← closed ⊕ n;
         if  (n ∈ ĐICH)  then  return  SOLUTION(n);
         if  (Γ(n) ≠ ∅) then fringe ← Γ(n) ⊕ fringe;
   }
   return  ("No solution");
}
```

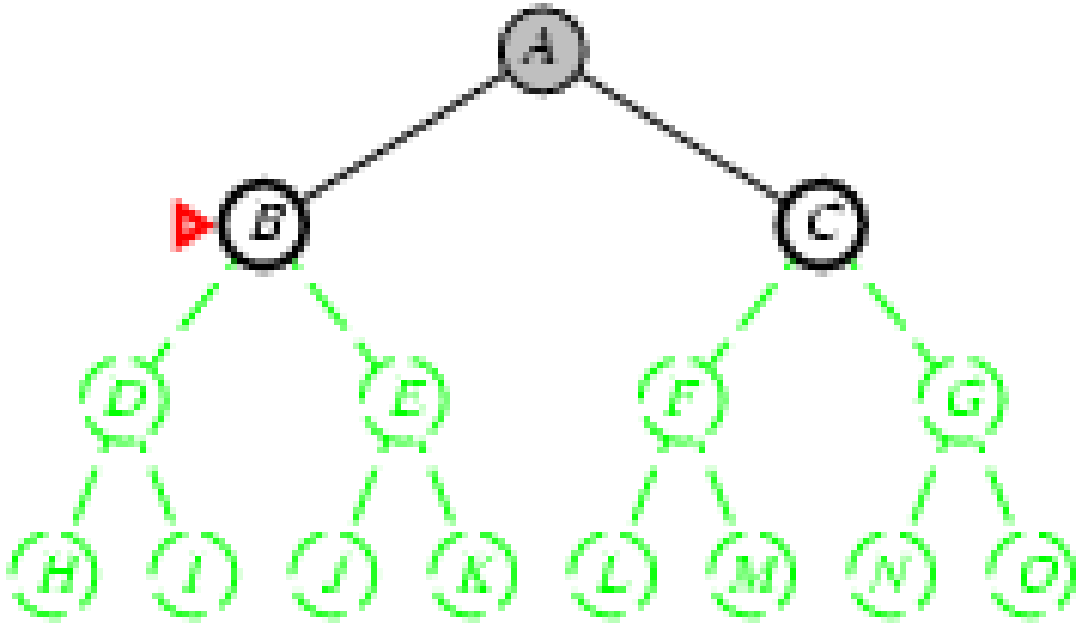
DFS: Ví dụ (1)

- Phát triển các nút chưa xét theo chiều sâu – Các nút được xét theo thứ tự độ sâu giảm dần



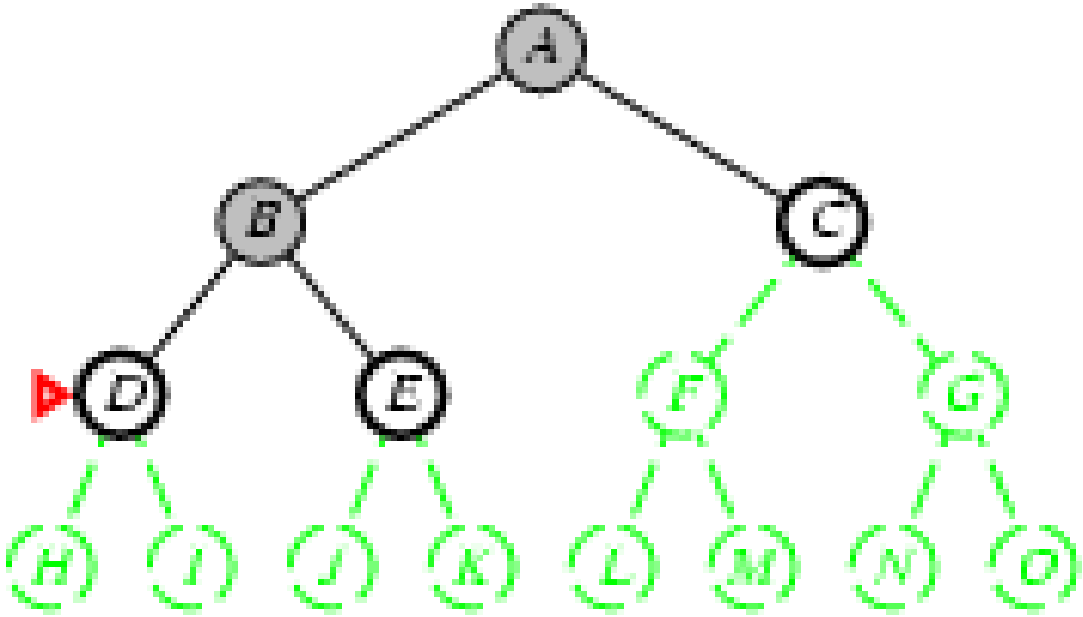
DFS: Ví dụ (2)

- Phát triển các nút chưa xét theo chiều sâu – Các nút được xét theo thứ tự độ sâu giảm dần



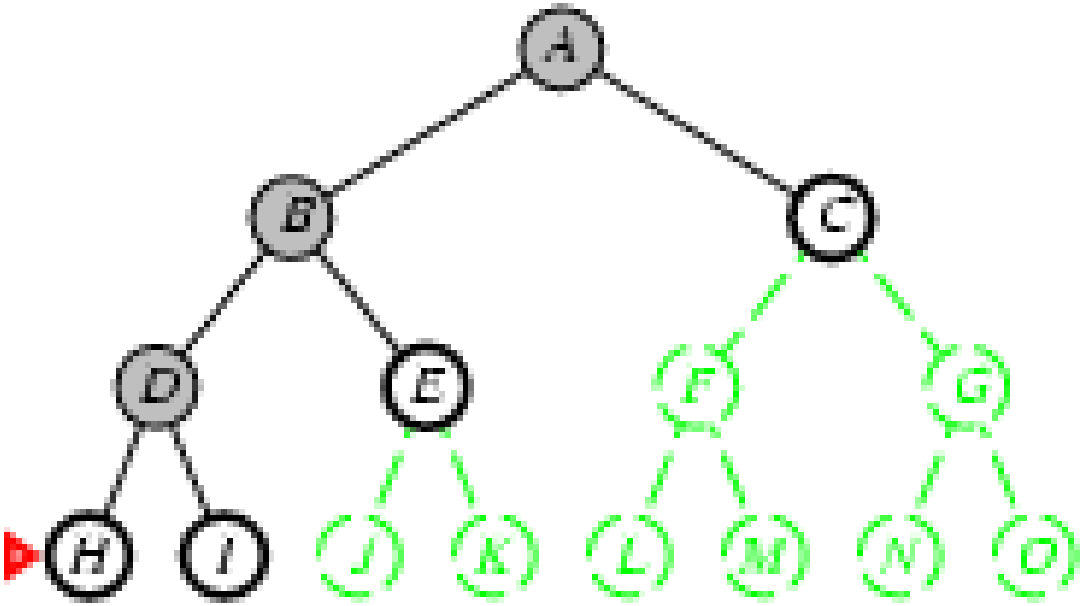
DFS: Ví dụ (3)

- Phát triển các nút chưa xét theo chiều sâu – Các nút được xét theo thứ tự độ sâu giảm dần



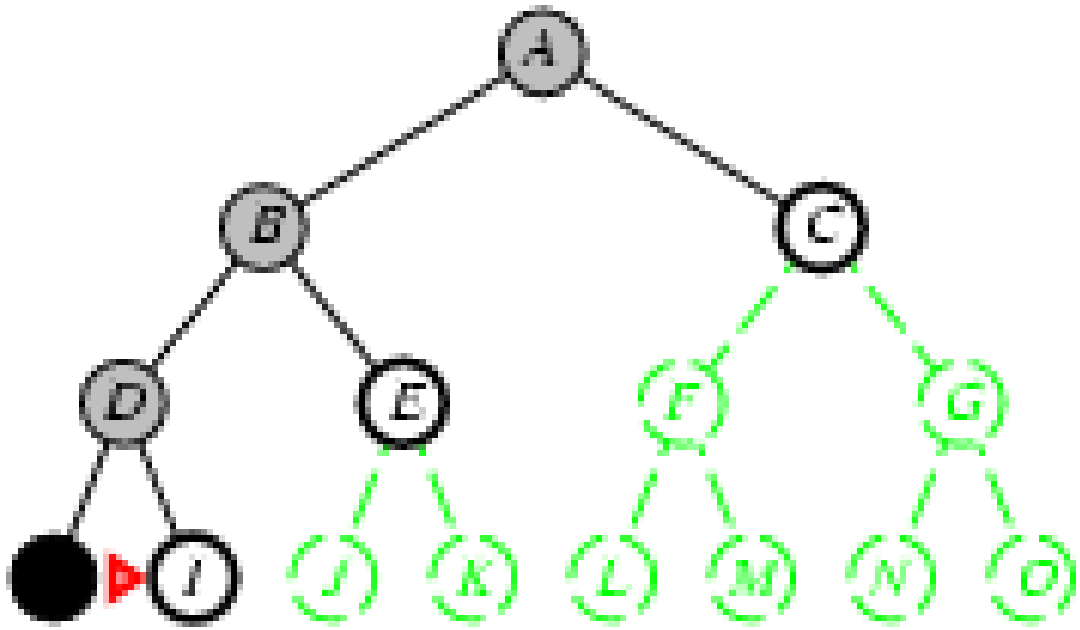
DFS: Ví dụ (4)

- Phát triển các nút chưa xét theo chiều sâu – Các nút được xét theo thứ tự độ sâu giảm dần



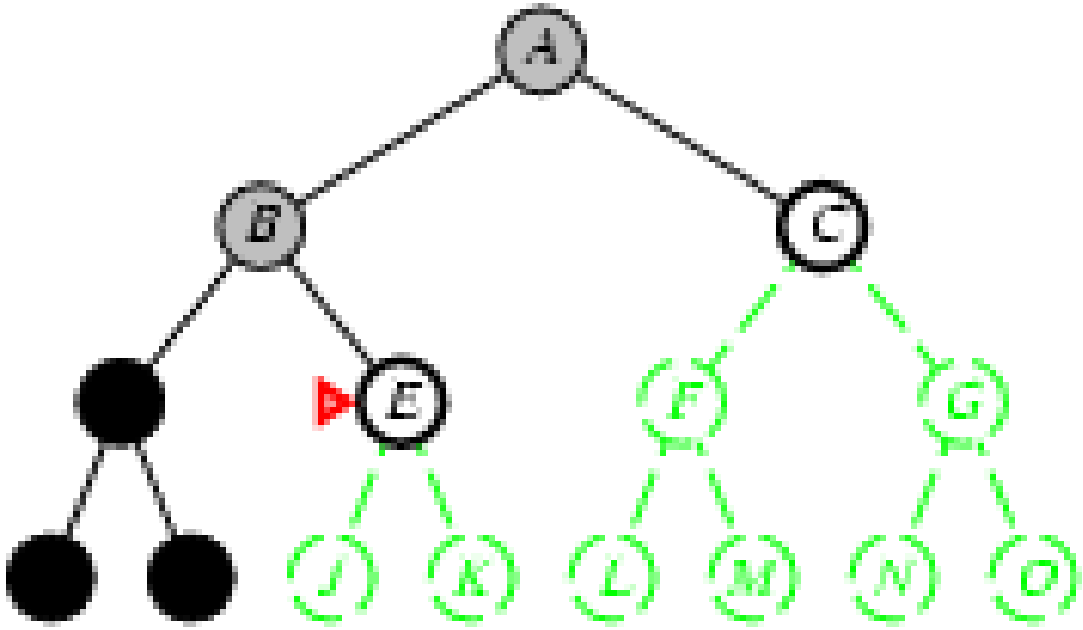
DFS: Ví dụ (5)

- Phát triển các nút chưa xét theo chiều sâu – Các nút được xét theo thứ tự độ sâu giảm dần



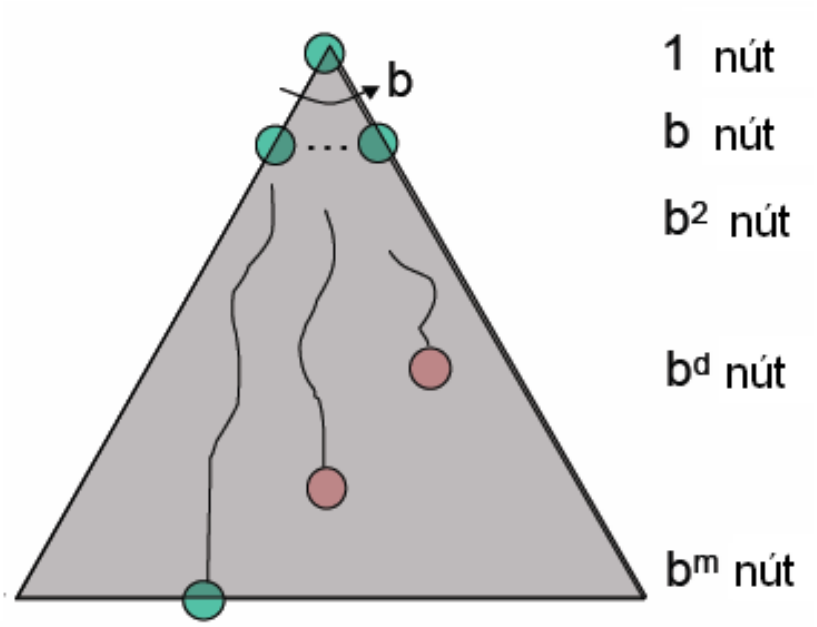
DFS: Ví dụ (6)

- Phát triển các nút chưa xét theo chiều sâu – Các nút được xét theo thứ tự độ sâu giảm dần



DFS: Các đặc điểm

- Tính hoàn chỉnh?
  - Không, chỉ hoàn chỉnh với không gian trạng thái hữu hạn
- Độ phức tạp về thời gian?
  - $O(b^m)$: rất lớn, nếu m lớn hơn nhiều so với d
- Độ phức tạp về bộ nhớ?
  - $O(bm)$ - độ phức tạp tuyến tính
- Tính tối ưu?
  - Không



Tìm kiếm giới hạn độ sâu (DLS)

= Phương pháp tìm kiếm theo chiều sâu (DFS) + Sử dụng giới hạn về độ sâu *l* trong quá trình tìm kiếm

$\rightarrow$ các nút ở độ sâu *l* không có nút con

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Tìm kiếm sâu dần (IDS)

- Vấn đề với giải thuật tìm kiếm với giới hạn độ sâu (DLS)
  - Nếu tất cả các lời giải (các nút đích) nằm ở độ sâu lớn hơn giới hạn độ sâu *l*, thì giải thuật DLS thất bại (không tìm được lời giải)
- Giải thuật tìm kiếm sâu dần
  - Áp dụng giải thuật DFS đối với các đường đi (trong cây) có độ dài <=1
  - Nếu thất bại (không tìm được lời giải), tiếp tục áp dụng giải thuật DFS đối với các đường đi có độ dài <=2
  - Nếu thất bại (không tìm được lời giải), tiếp tục áp dụng giải thuật DFS đối với các đường đi có độ dài <=3
  - …(tiếp tục như trên, cho đến khi: 1) tìm được lời giải, hoặc 2) toàn bộ cây đã được xét mà không tìm được lời giải)

IDS: Giải thuật (1)

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

IDS: Ví dụ (1)

Giới hạn độ sâu *l* = 0

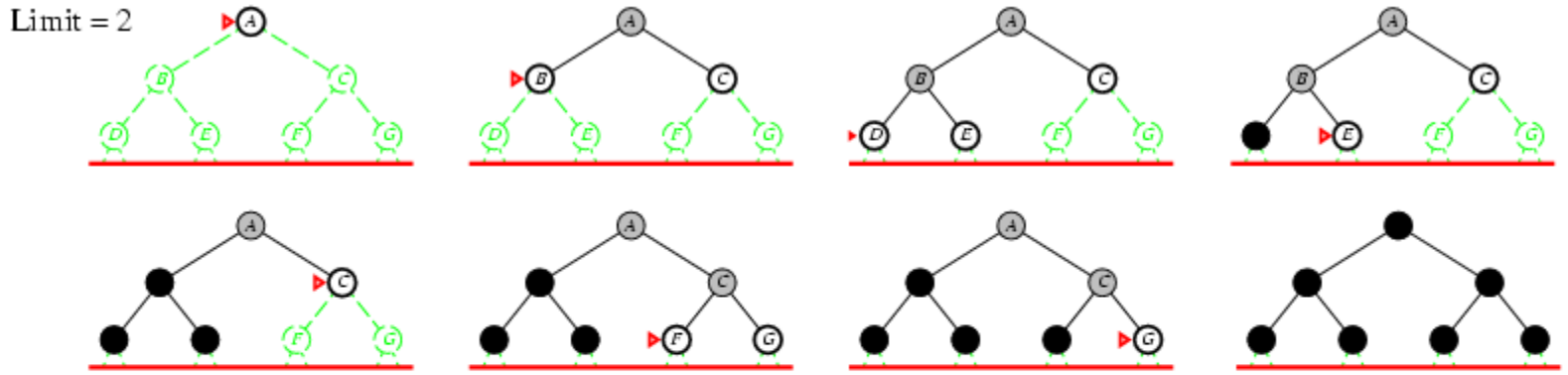
Limit = 0

IDS: Ví dụ (2)

Giới hạn độ sâu *l* = 1

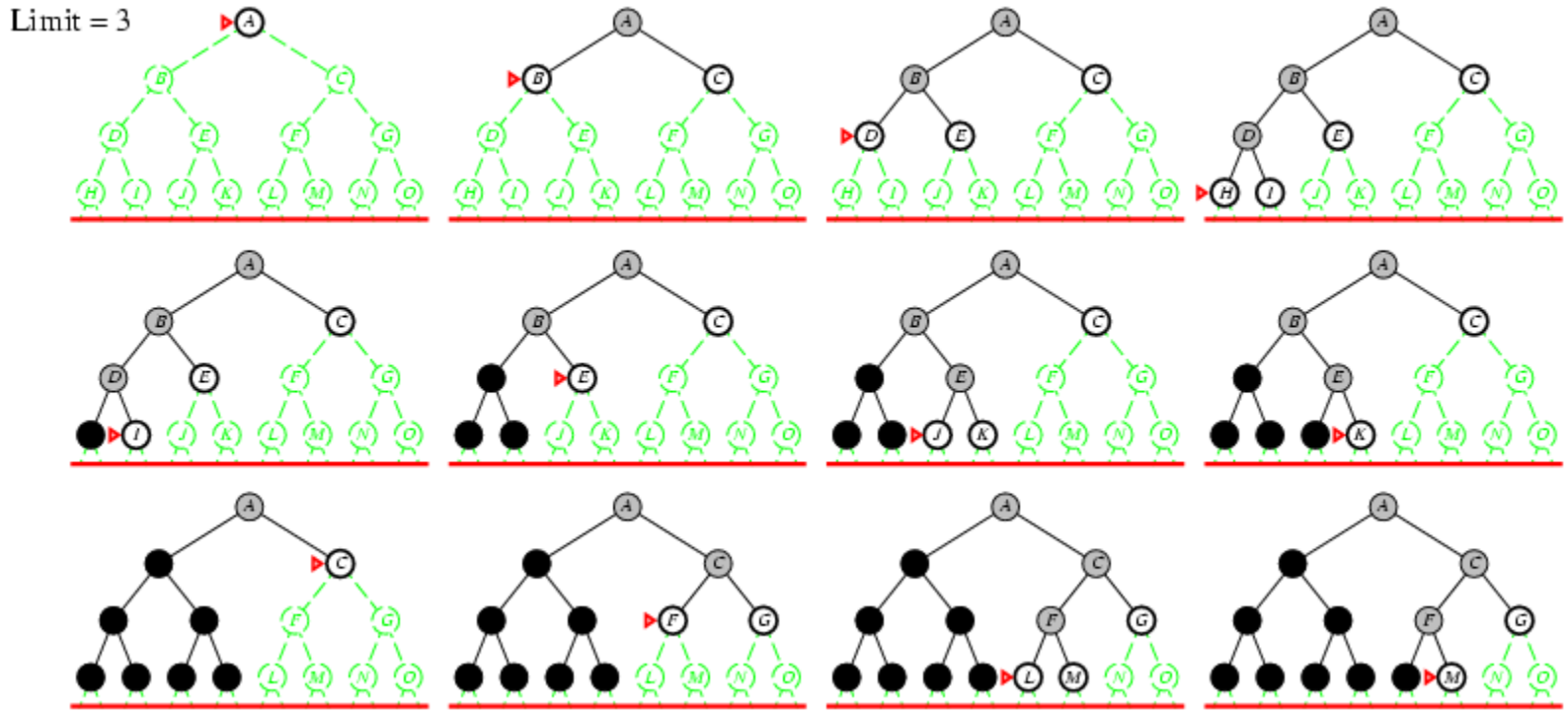
IDS: Ví dụ (3)

Giới hạn độ sâu *l* = 2



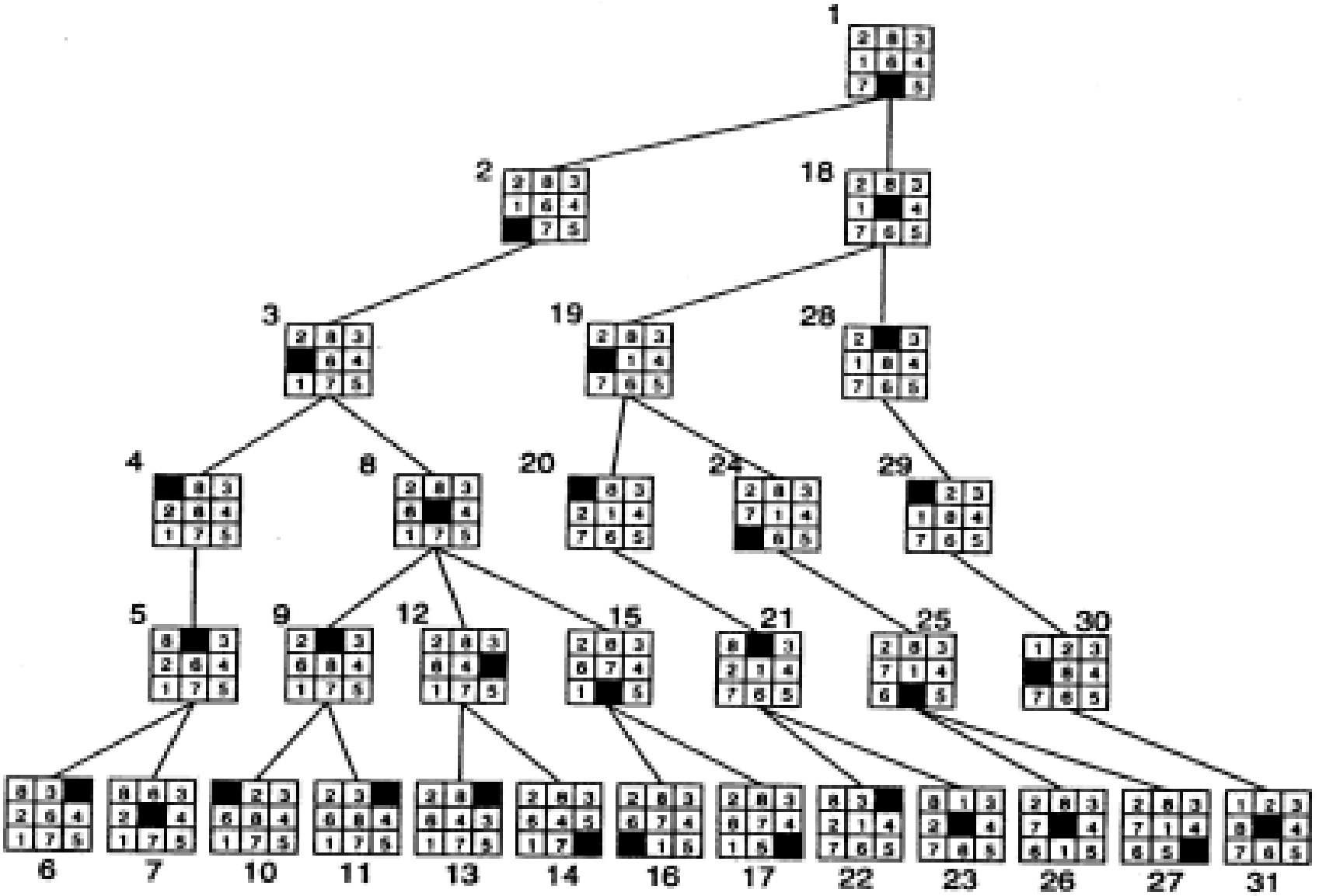
IDS: Ví dụ (4)

Giới hạn độ sâu *l* = 3



IDS: Giải thuật (2)

```
IDS (N, A, n0, ĐICH, l)                          // l: giới hạn độ sâu
{
    fringe ← n0;          closed ← ∅;            depth ← l;
    while  (fringe ≠ ∅)  do
    {   n ← GET_FIRST(fringe);      // lấy phần tử đầu tiên của  fringe
        closed ← closed ⊕ n;
        if  (n ∈ ĐICH)  then  return  SOLUTION(n);
        if  (Γ(n) ≠ ∅) then
        {       case  d(n)  do                   // d(n): độ sâu của nút n
                [0..(depth-1)]:  fringe ← Γ(n) ⊕ fringe;
                depth:           fringe ← fringe ⊕ Γ(n);
                (depth+1):       {    depth ← depth + l;
                                      if  (l=1)  then  fringe ← fringe ⊕ Γ(n)
                                                 else  fringe ← Γ(n) ⊕ fringe;
                                 }
         }
    }
    return  (“No solution”);
}
```



Trò chơi ô đố 8-puzzle với ngưỡng sâu 5

DLS vs. IDS

- Với độ sâu *d* và hệ số phân nhánh *b*, thì số lượng các nút được sinh ra trong giải thuật tìm kiếm giới hạn độ sâu là:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Số lượng các nút được sinh ra trong giải thuật tìm kiếm sâu dần là:

$$N_{IDS} = (d+1).b^0 + d.b^1 + (d-1).b^2 + \ldots + 3.b^{d-2} + 2.b^{d-1} + 1.b^d$$

- Ví dụ với *b = 10*, *d = 5*:
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
  - Lãng phí = (123,456 - 111,111)/111,111 = 11%

IDS: Các đặc điểm

- Tính hoàn chỉnh?
  - Có
- Độ phức tạp về thời gian?
  - $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^{d+1})$
- Độ phức tạp về bộ nhớ?
  - $O(bd)$
- Tính tối ưu?
  - Có, nếu chi phí cho mỗi bước (mỗi cạnh của cây tìm kiếm) = 1

So sánh giữa các giải thuật tìm kiếm cơ bản

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^{d+1})$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes (some cases)	Yes (some cases)	No	No	Yes (some cases)

Tổng kết

- Việc phát biểu bài toán thường yêu cầu việc khái quát hóa các chi tiết của bài toán thực tế, để có thể định nghĩa không gian trạng thái sao cho việc xét (khám phá) các trạng thái trong quá trình tìm kiếm được thuận tiện
- Có nhiều chiến lược tìm kiếm cơ bản
  - Tìm kiếm theo chiều rộng (BFS)
  - Tìm kiếm theo chiều sâu (DFS)
  - Tìm kiếm với chi phí cực tiểu (UCS)
  - Tìm kiếm giới hạn độ sâu (DLS)
  - Tìm kiếm sâu dần (IDS)
- Phương pháp tìm kiếm sâu dần (IDS)
  - Chi phí về bộ nhớ ở mức hàm tuyến tính
  - Chi phí về thời gian chỉ nhiều hơn một chút so với các phương pháp tìm kiếm cơ bản khác