ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

# Artificial Intelligence

## Lecture 3 - Search

School of Information and Communication Technology - HUST

# Outline

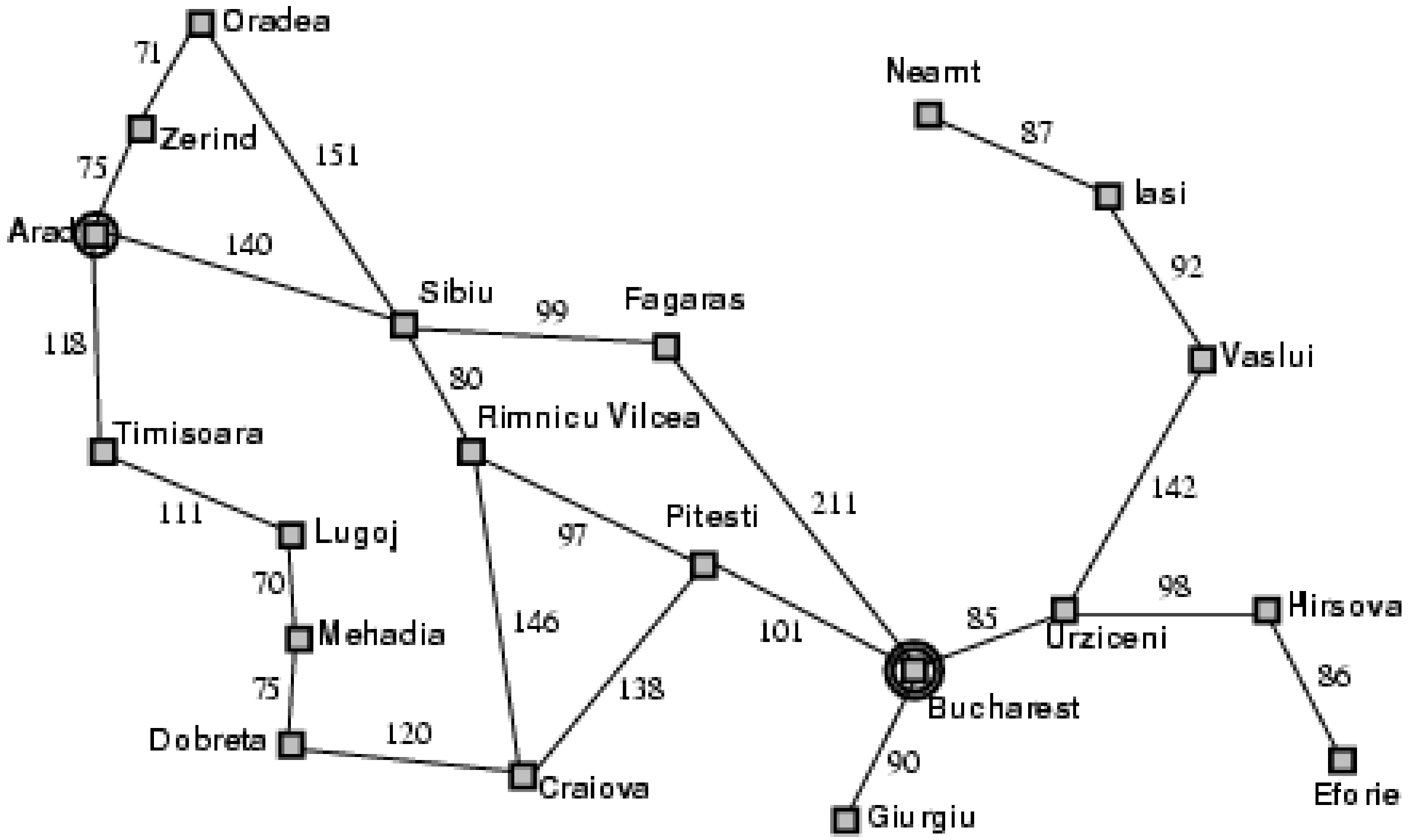
- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
  - breadth-first search
  - depth-first search
  - depth-limited search
  - iterative deepening depth-first search

# Problem-solving agents

```
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
```

# Example 1: Route Planning




- Performance: Get from Arad to Bucharest as quickly as possible
- Environment: The map, with cities, roads, and guaranteed travel times
- Actions: Travel a road between adjacent cities

# Example 2: Finding letters

- Replace letters by numbers from 0 to 9 such as no different letter is replaced by the same number and satisfying the following constraint:

```
    SEND          CROSS
+   MORE        + ROADS
  ------        -------
   MONEY         DANGER
```

# Example 3: Pouring water

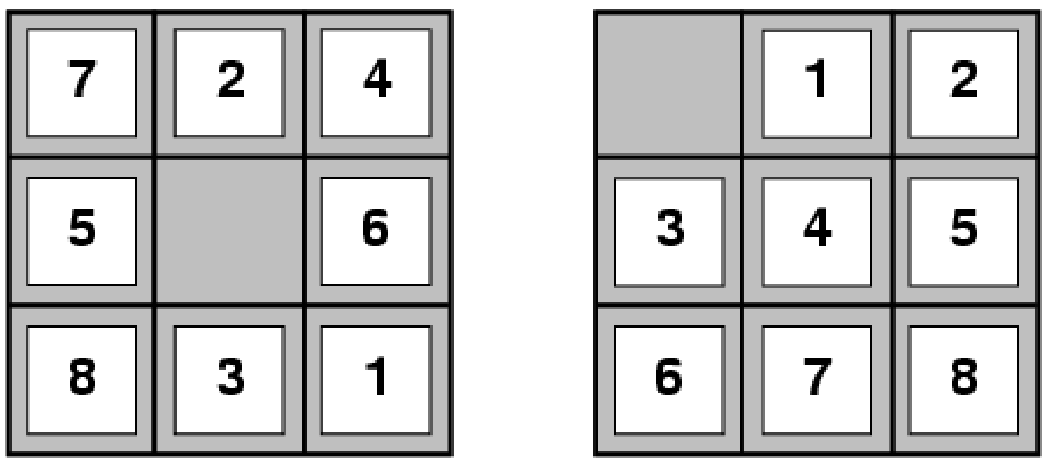
- Given 2 containers A(m litres), B(n litres). Finding a method to measure k litres ( $k \leq \max(m,n)$ ) by 2 containers A, B and a container C
- Actions (how):

  C → A; C → B; A → B; A → C; B → A; B → C
- Conditions: no overflow, pouring all water
- Eg: m = 5, n = 6, k = 2    (what)
- Mathematical model:

  $$\underset{A}{(x}, \underset{B}{y)} \rightarrow \underset{A}{(x'}, \underset{B}{y')}$$

# Example 4: The 8-puzzle

- Trong bảng ô vuông n hàng, n cột, mỗi ô chứa 1 số nằm trong phạm vi từ 1 → $n^2$ -1 sao cho không có 2 ô có cùng giá trị. Còn đúng 1 ô bị trống. Xuất phát từ 1 cách sắp xếp nào đó của các đó của các số trong bảng, hãy dịch chuyển các ô trống sang phải, sang trái, lên trên, xuống dưới để đưa về bảng:

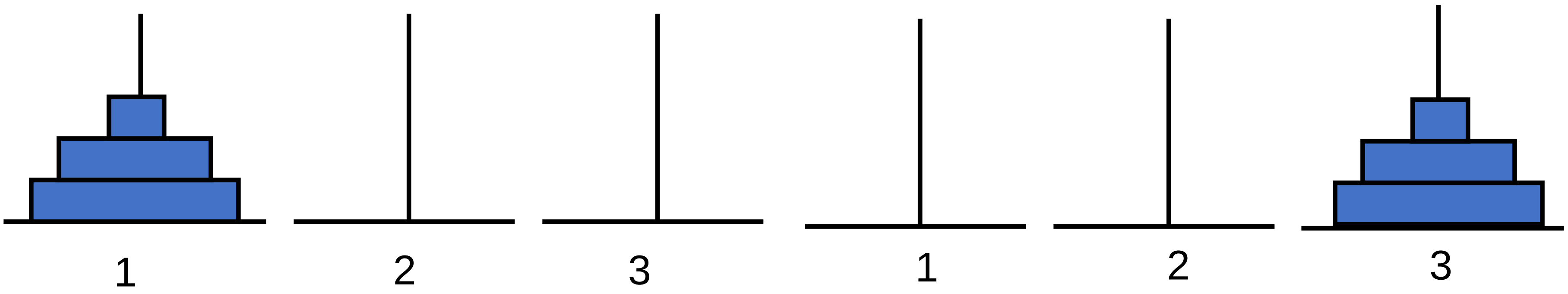



Start State

Goal State

# Example 5: Hà Nội tower

- Cho 3 cọc 1,2,3. Ở cọc 1 ban đầu có n đĩa, sắp theo thứ tự to dần từ trên xuống dưới. Hãy tìm cách chuyển n đĩa đó sang cọc 3 sao cho:
  - Mỗi lần chỉ chuyển 1 đĩa
  - Ở mỗi cọc không cho phép đĩa to nằm trên đĩa con




Bài toán tháp Hà Nội với n = 3

# Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave → search, execution
- Unknown state space → exploration problem

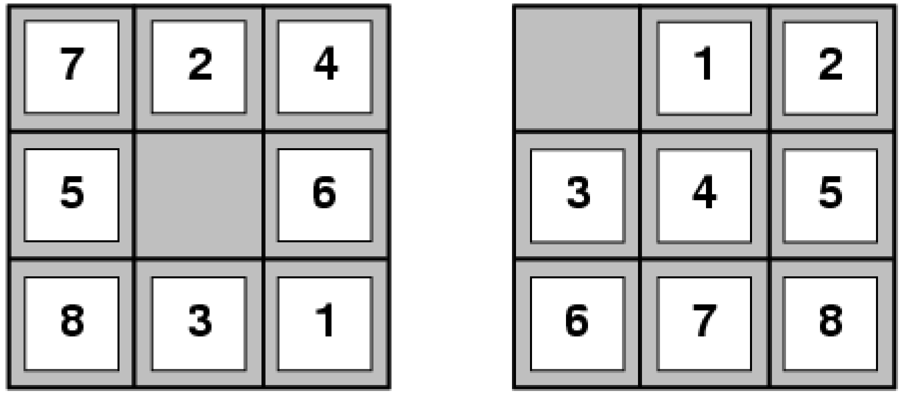
# Search Problem Definition

A problem is defined by four items:

1. initial state: e.g., Arad
2. actions or successor function $S(x)$ = set of action-state pairs
   - e.g., $S(Arad) = \{<Arad \rightarrow Zerind, Zerind>, \ldots\}$
3. goal test, can be
   - explicit, e.g., $x$ = Bucharest
   - implicit, e.g., $Checkmate(x)$
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - $c(x,a,y)$ is the step cost, assumed to be $\geq 0$

- A solution is a sequence of actions leading from the initial state to a goal state

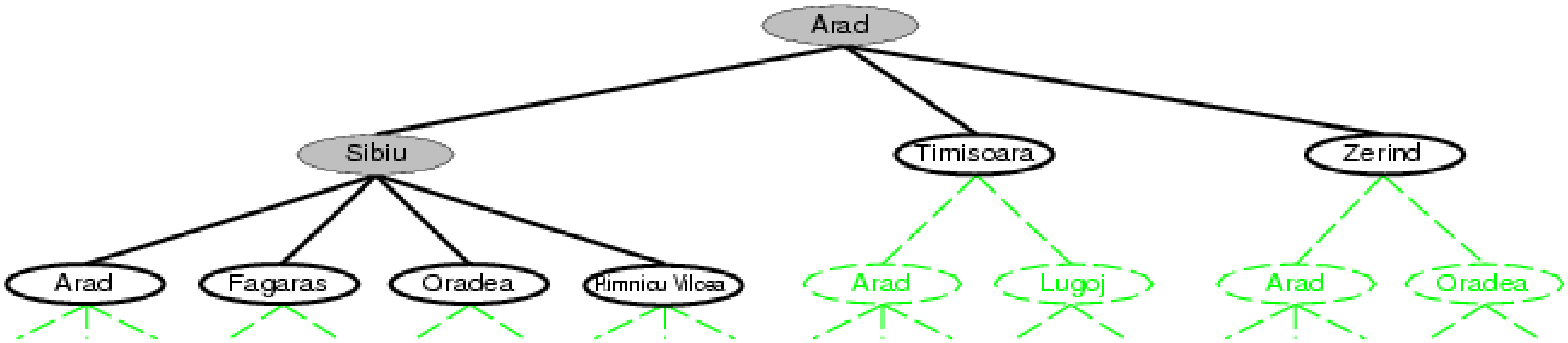
# Example: The 8-puzzle




Start State        Goal State

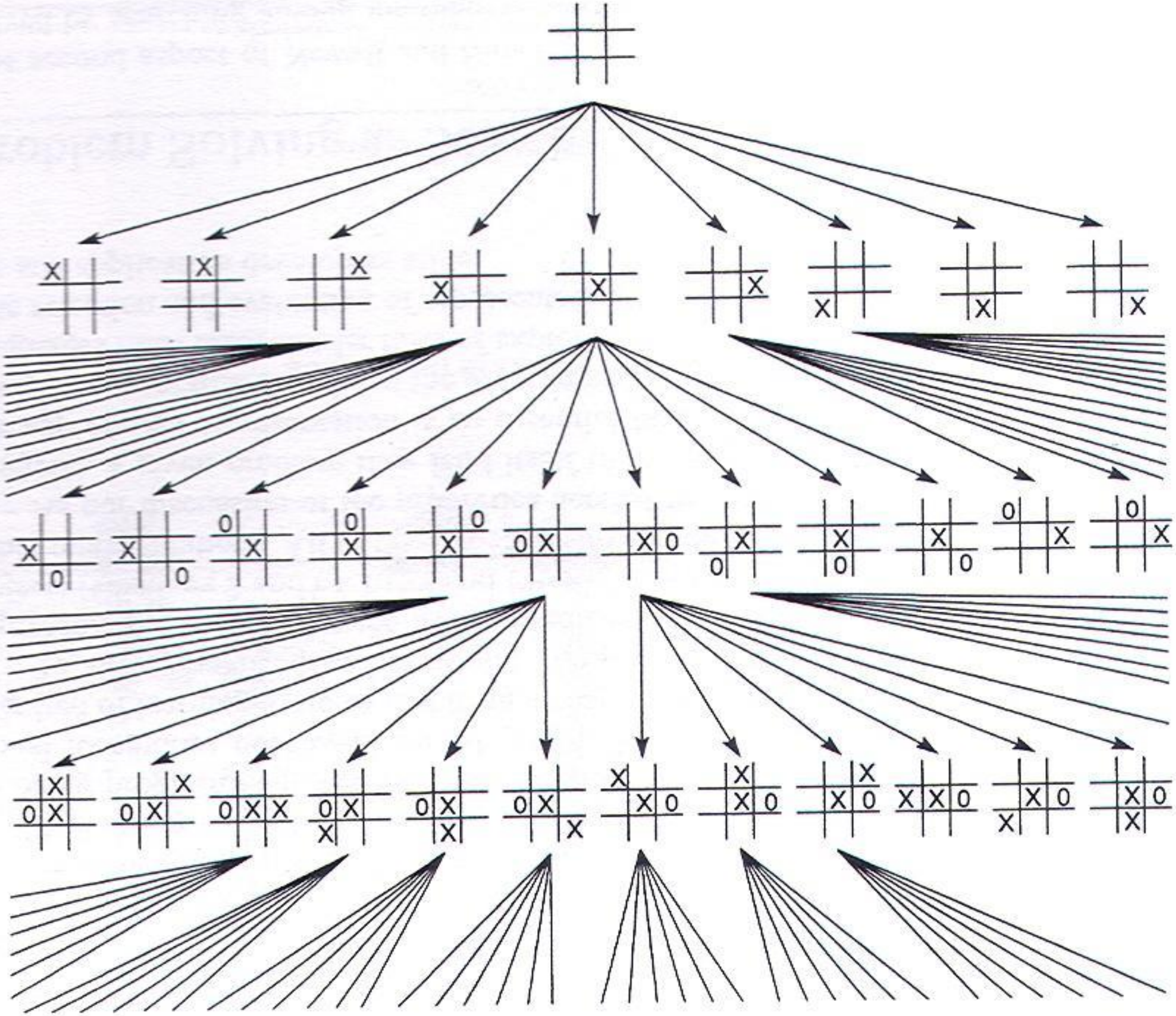
- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

# Search tree



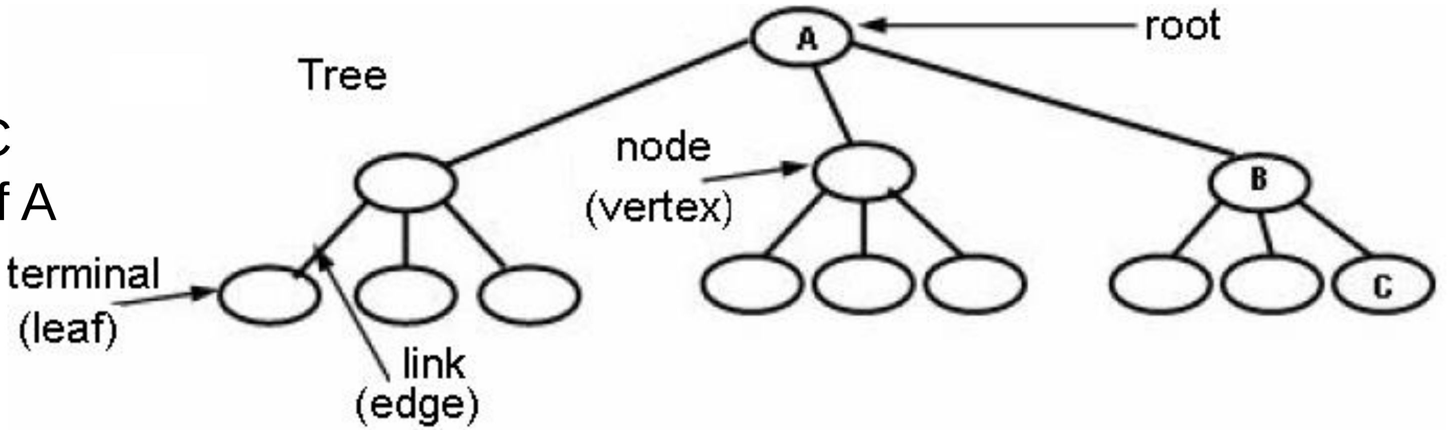

- Search trees:
  - Represent the branching paths through a state graph.
  - Usually **much** larger than the state graph.
  - Can a finite state graph give an infinite search tree?

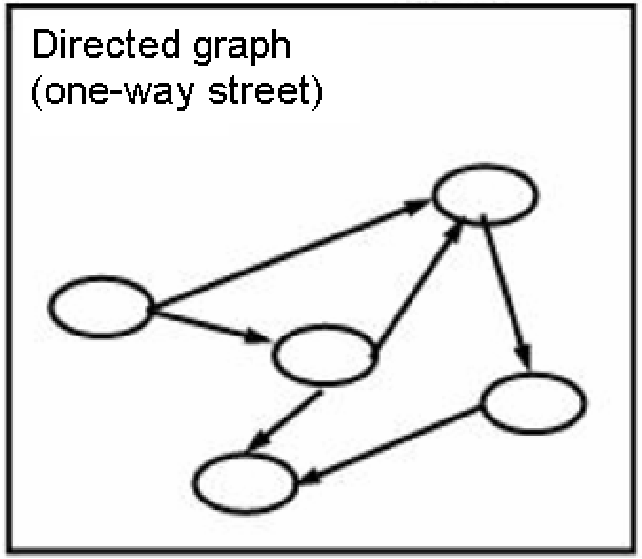
# Search space of  the game Tic-Tac-Toe

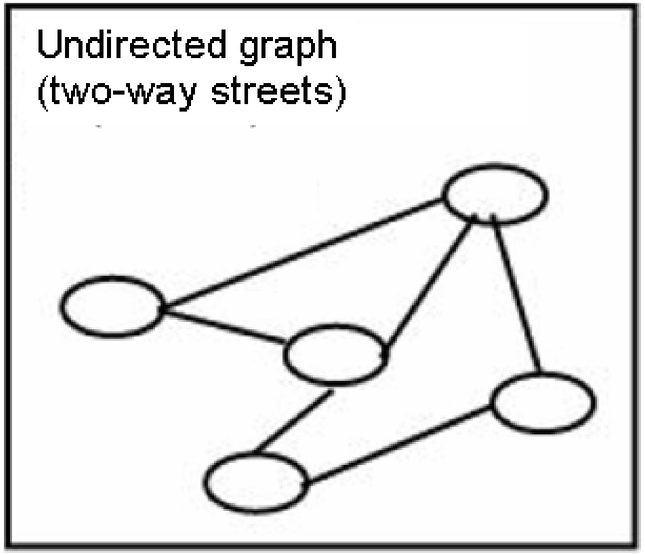


# Tree and graph

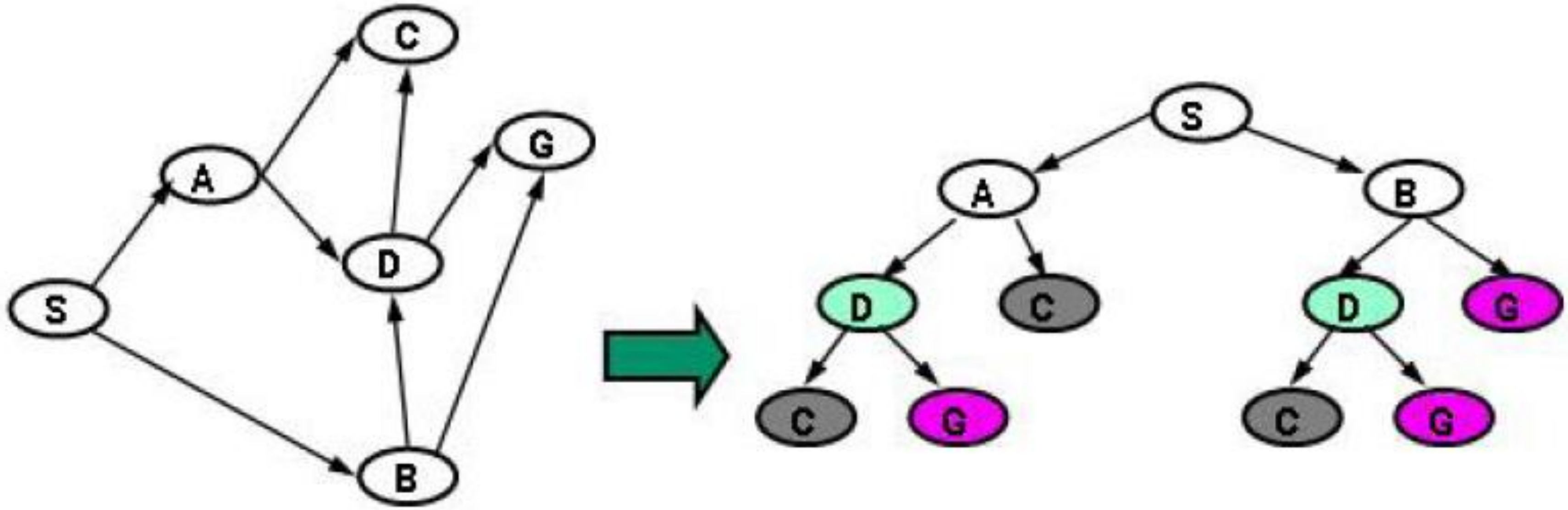
B is parent of C
C is child of B
A is ancestor of C
C is decendant of A



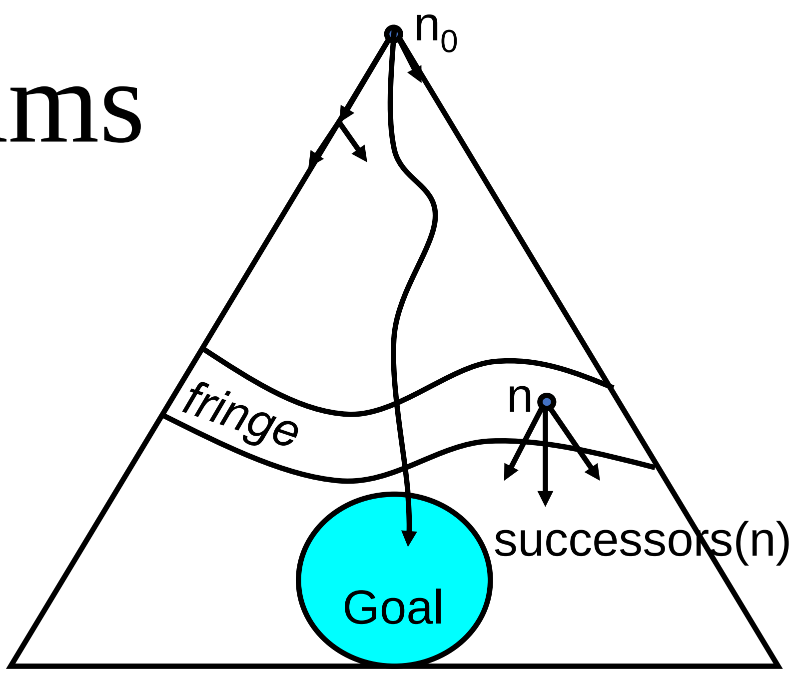
# Convert from search graph to search tree




- We can turn graph search problems into tree search problems by:
  - replacing undirected links by 2 directed links
  - avoiding loops in path (or keeping trach of visited nodes globally)

# Tree search algorithms

- Basic idea:
  - simulated exploration of state space by generating successors of already-explored states




```
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
```

# Implementation: general tree search

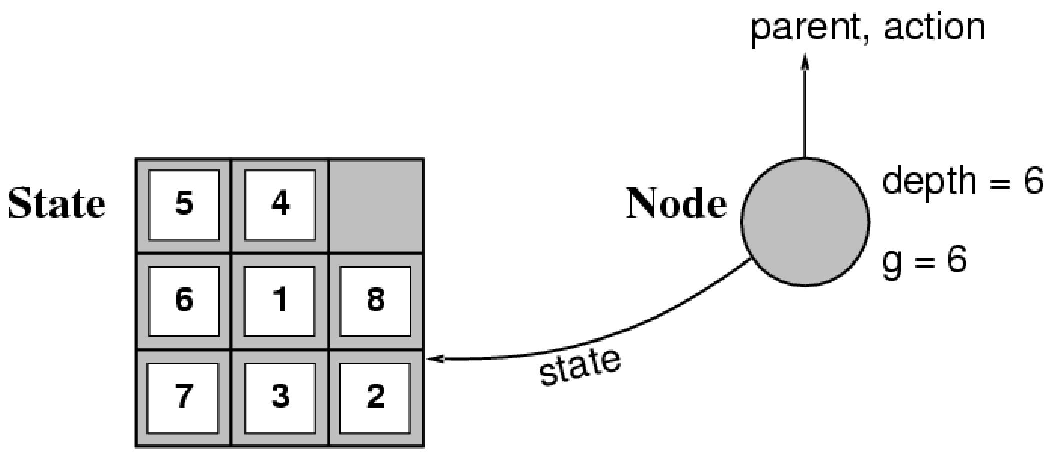
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if fringe is empty then return failure
       node ← REMOVE-FRONT(fringe)
       if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   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

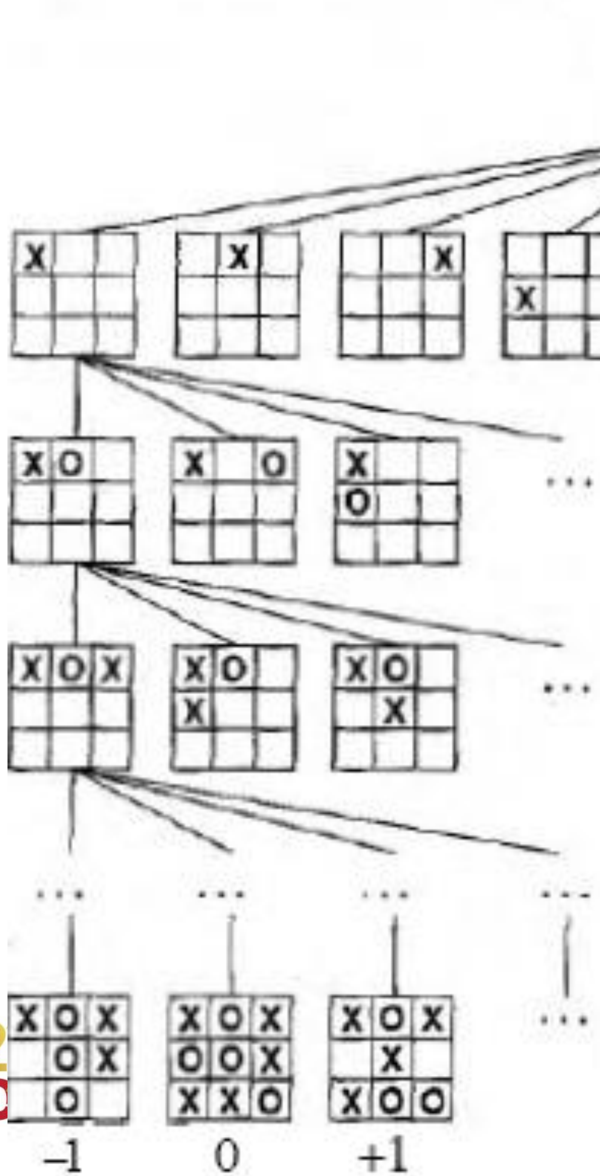
# Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(x)$, depth




- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.
-

# Implementation: states vs. nodes



# Search strategies

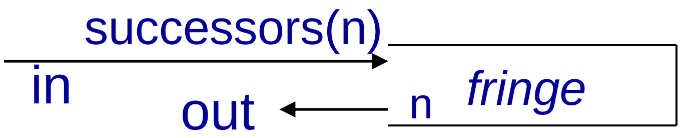
- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - $b$: maximum branching factor of the search tree
  - $d$: depth of the least-cost solution
  - $m$: maximum depth of the state space (may be $\infty$)

# Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition

- Breadth-first search
  - Expand shallowest unexpanded node
  - *fringe* = queue (FIFO)

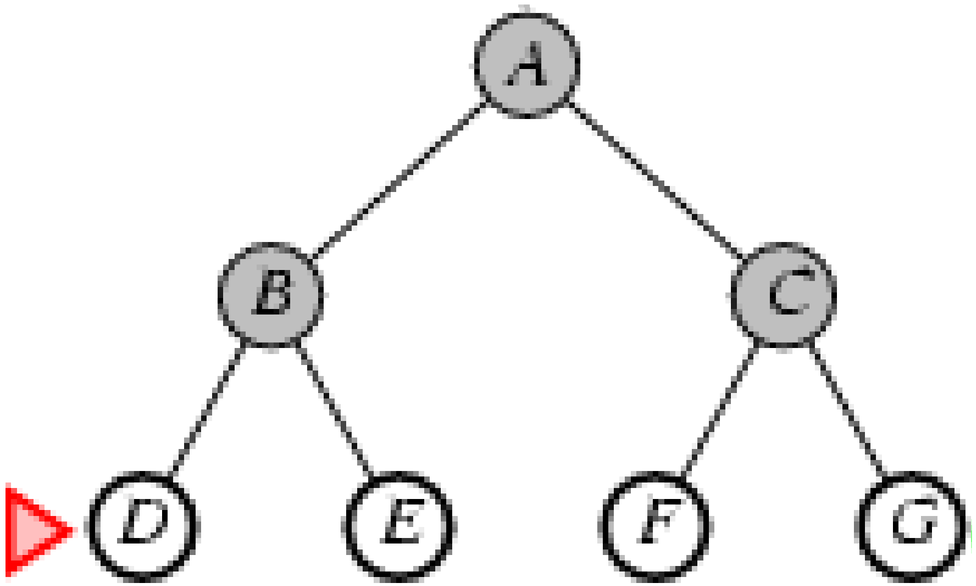

- Depth-first search
  - Expand deepest unexpanded node
  - *fringe* = stack (LIFO)




- Depth-limited search: depth-first search with depth limit
- Iterative deepening search

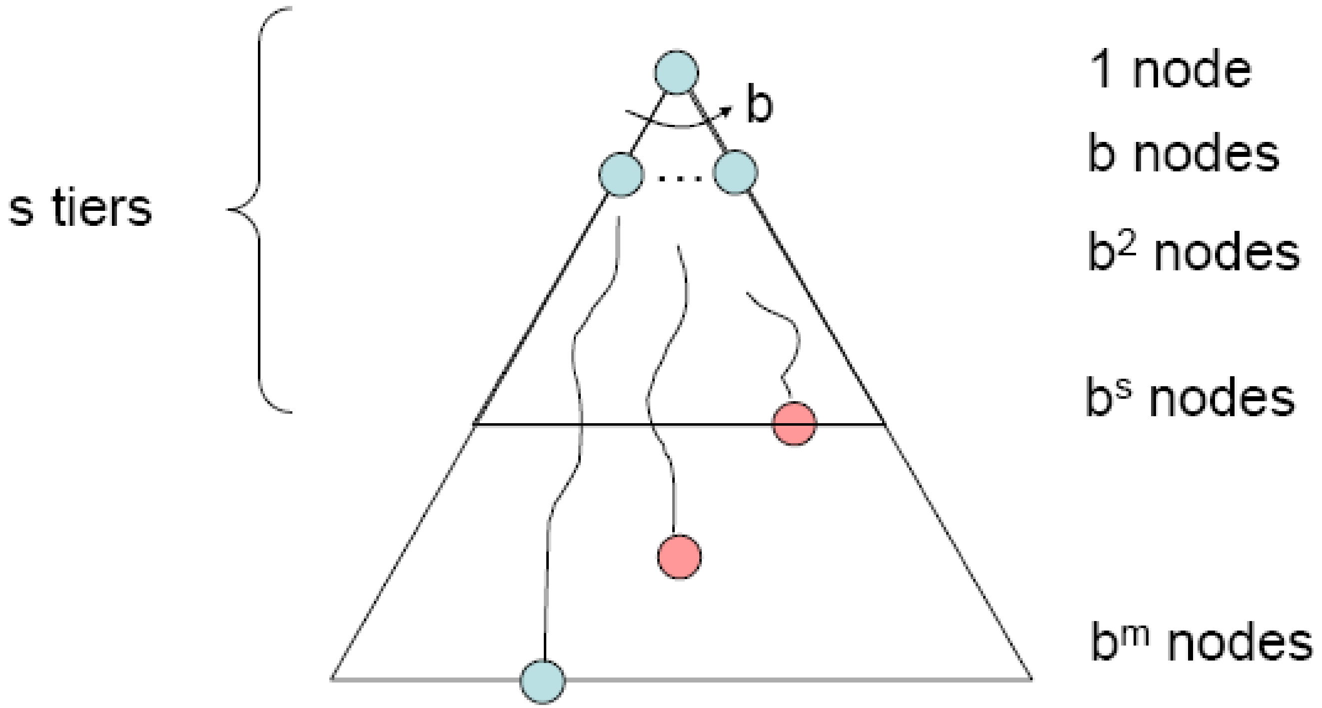
# Breadth-first search

- Expand shallowest unexpanded node



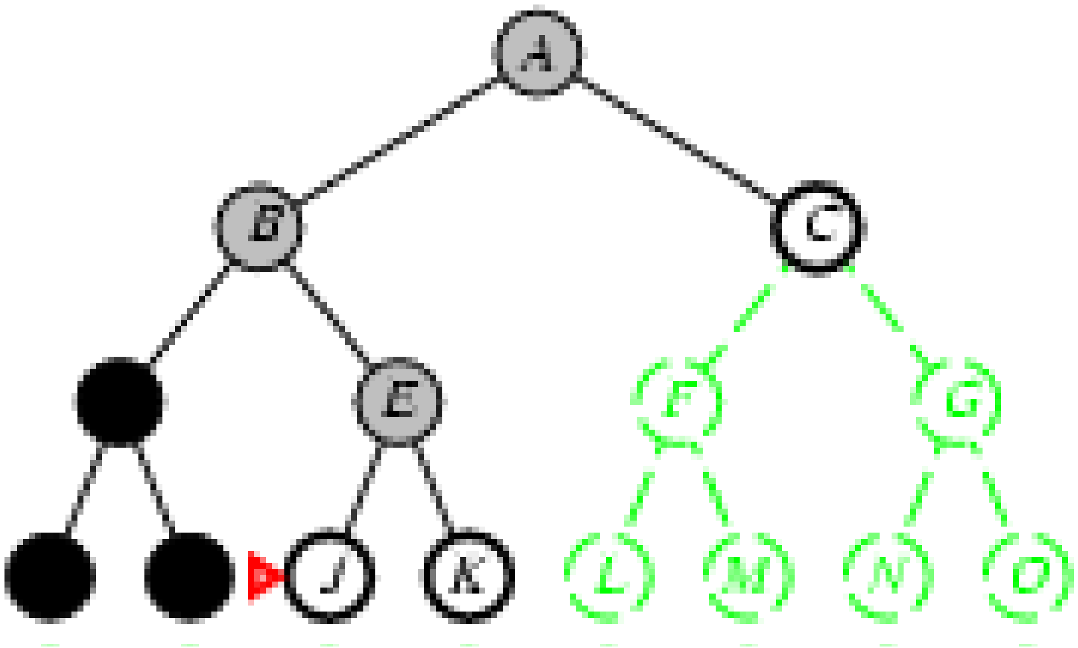
# Breadth-first search (con't)

- Complete? Yes (if $b$ is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)



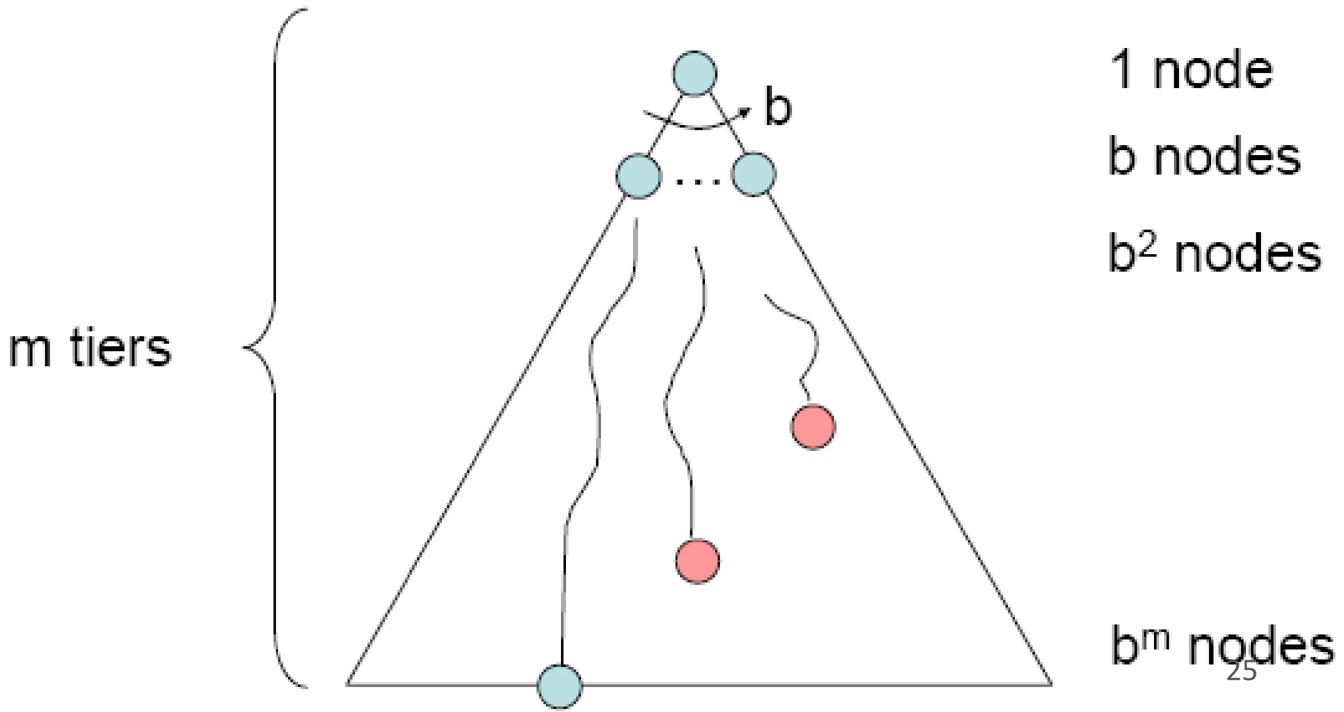
# Depth-first search

- Expand deepest unexpanded node



# Depth-first search (con't)

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path → complete in finite spaces
- Time? $O(b^m)$: terrible if $m$ is much larger than $d$
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- 
- Optimal? No
- 



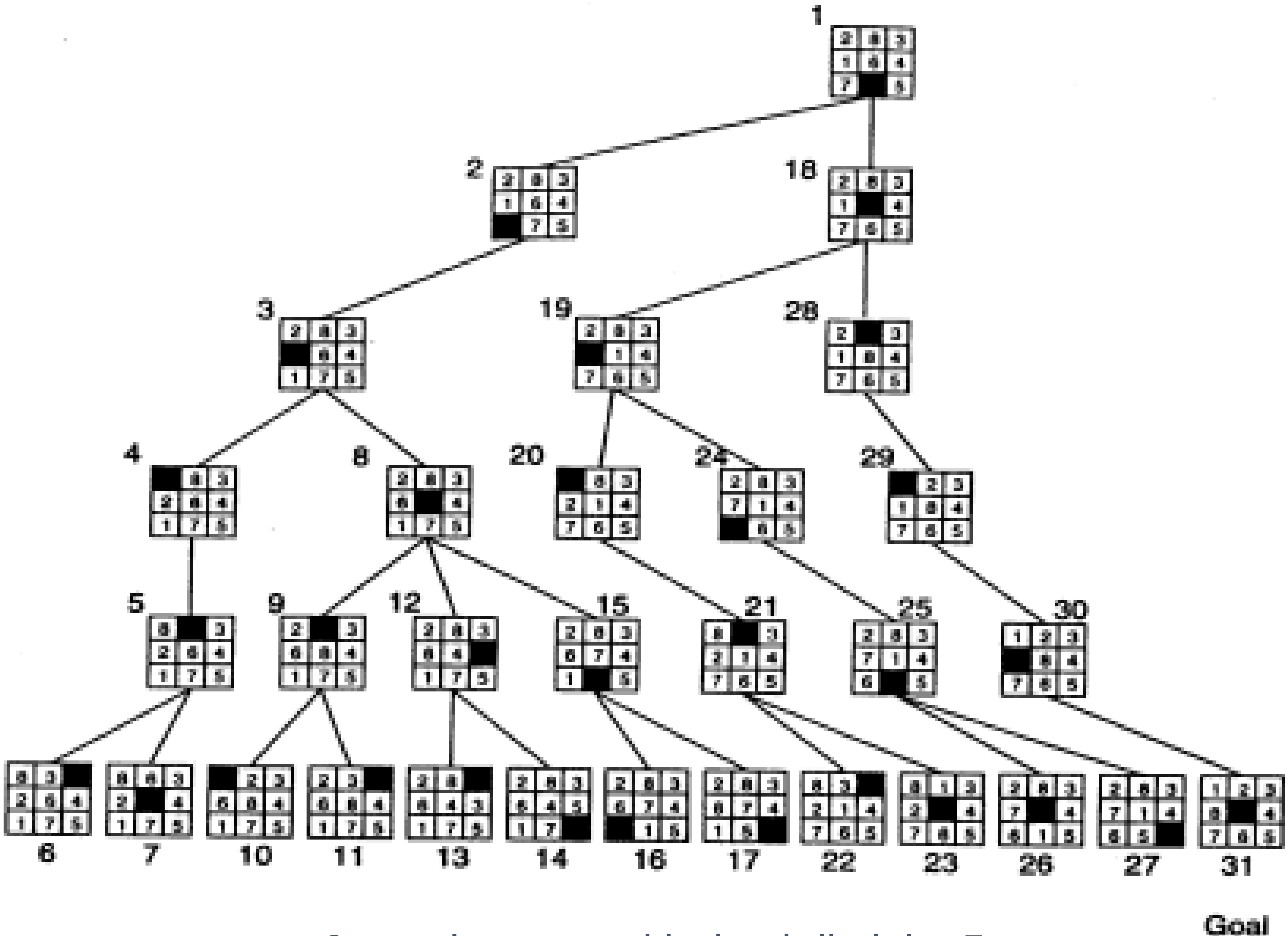
# Depth-limited search

- Depth-first search can get stuck on infinite path when a different choice would lead to a solution

⇒ Depth-limited search = depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors

```
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
```



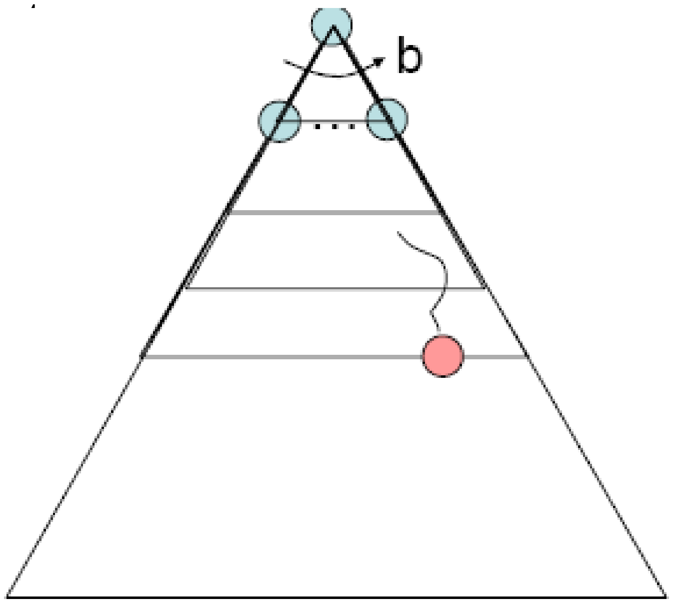

8-puzzle game with depth limit l = 5

# Iterative deepening search

- Problem with depth-limited search: if the shallow goal is beyond the depth limit, no solution is foun

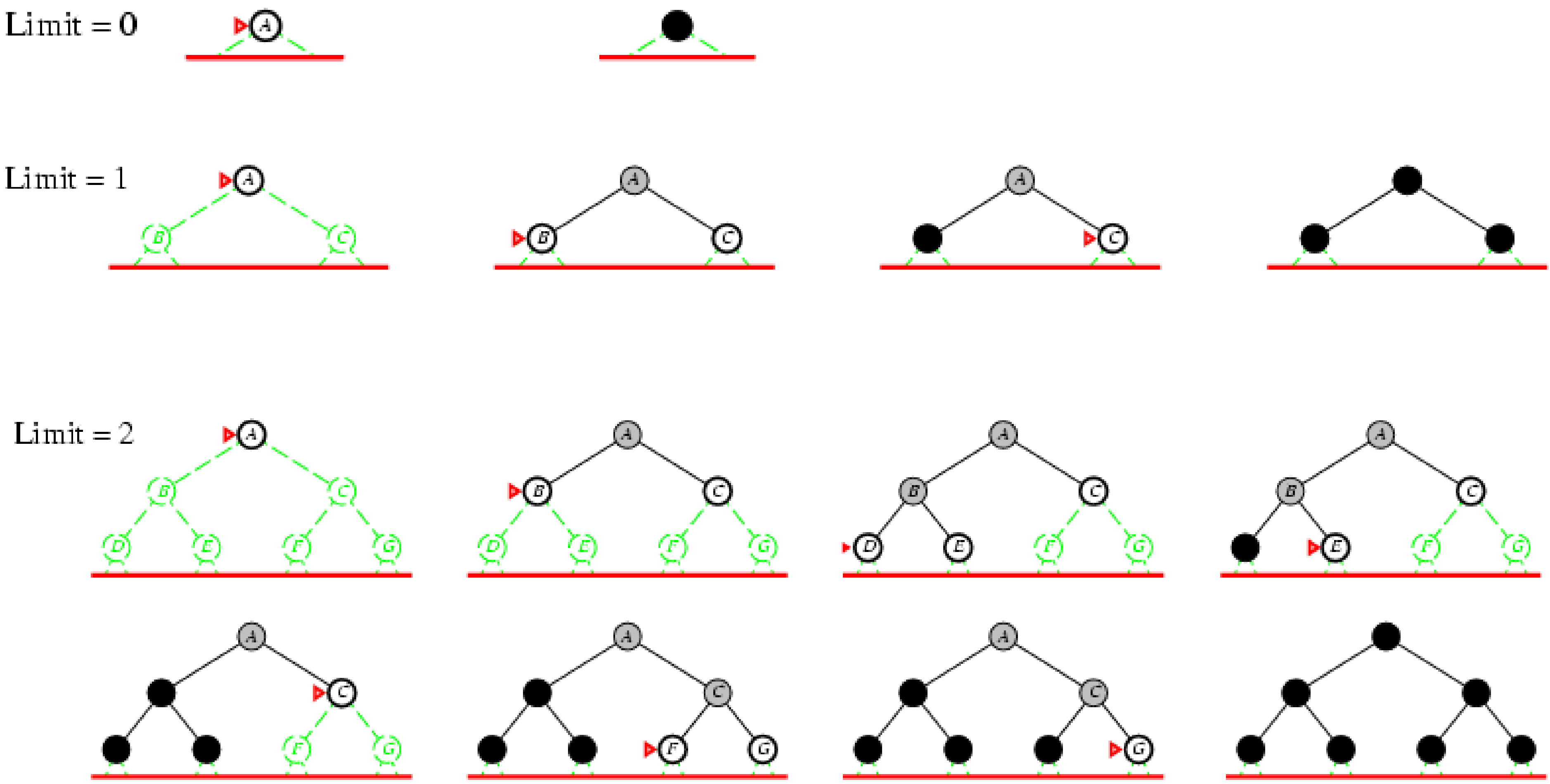
⇒ Iterative deepening search:

1. Do a DFS which only searches for paths of length 1 (DFS gives up on any path of length 2)
2. If "1" failed, do a DFS which only searches paths of 2 or less.
3. If "2" failed, do a DFS which only searches paths of 3 or less.
4. ….and so on.



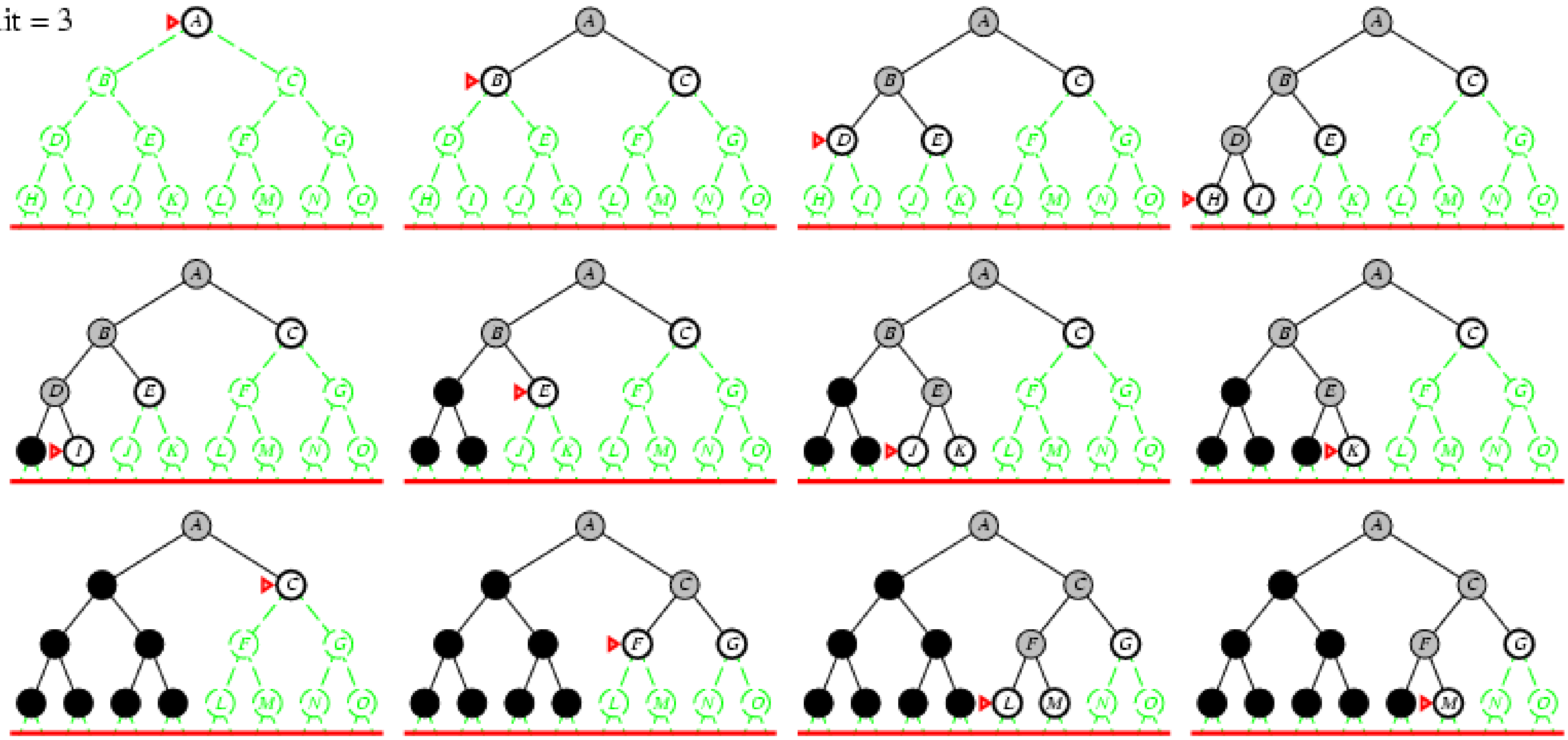

**function** ITERATIVE-DEEPENING-SEARCH( *problem*) **returns** a solution, or failure
  **inputs**: *problem*, a problem
  **for** *depth* ← 0 **to** ∞ **do**
    *result* ← DEPTH-LIMITED-SEARCH( *problem*, *depth*)
    **if** *result* ≠ cutoff **then return** *result*

# Iterative deepening search (con't)



# Iterative deepening search (con't)

Limit = 3



# Iterative deepening search (con't)

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

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

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

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

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

# Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

# Summary of algorithms

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