331 – Intro to AI
Week02
(R&N Chapter 3)
Uninformed Search

T.J. Borrelli

Uninformed vs informed search

- Uninformed search

  – Can only distinguish goals from non-goal states

- Informed search

– Have information about progress towards the optimal solution

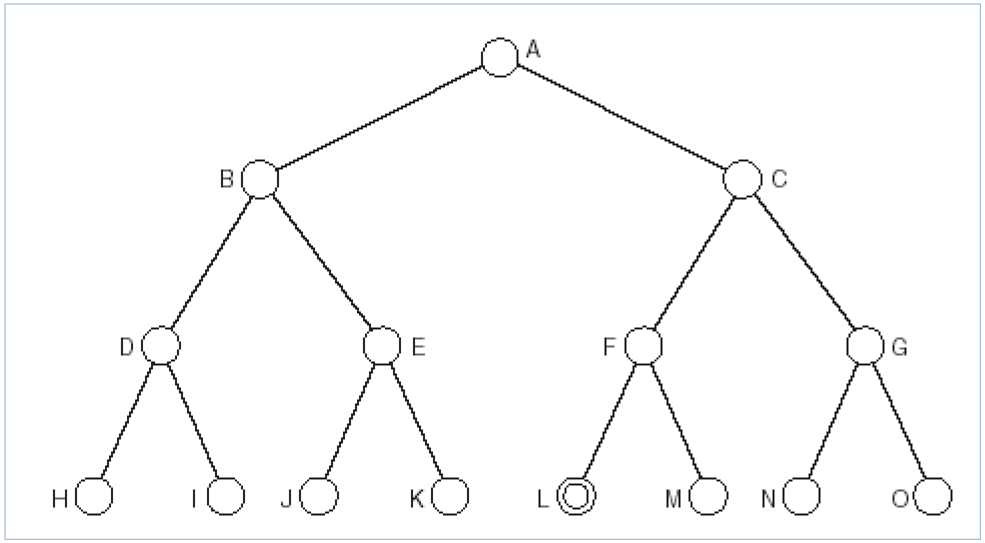
– can drastically improve time complexity

State Space Representation of a Problem

- In the *state-space representation* of a problem, the nodes of a tree (called the search tree) correspond to partial problem solution states, and the links correspond to steps in a problem-solving process
- The root of the tree is the start state
- Goal states are leaf nodes (but not the other way around)
- A state-space search characterizes problem-solving as the process of finding a solution path from the start state to a goal

Search Tree

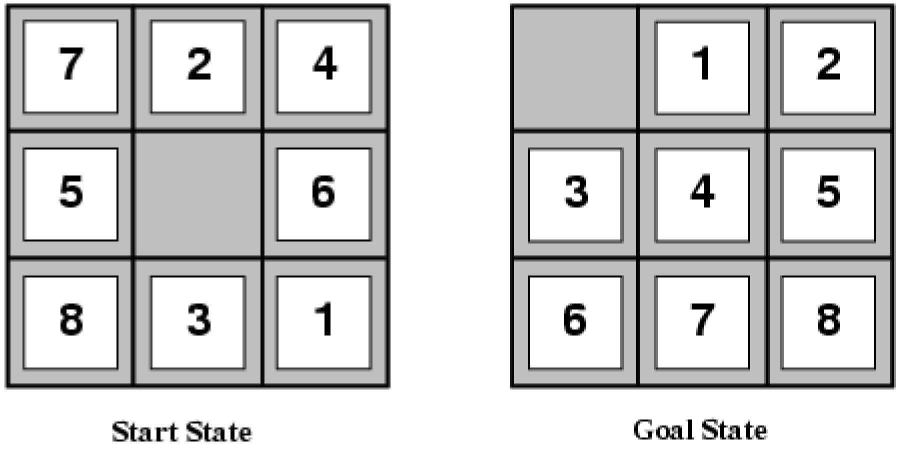
- A is the root node (start state).
- L is the goal node (goal state).
- H, I, J, K, M, N and O are other leaf nodes.
- There is only one solution path: A, C, F, L



Generating States

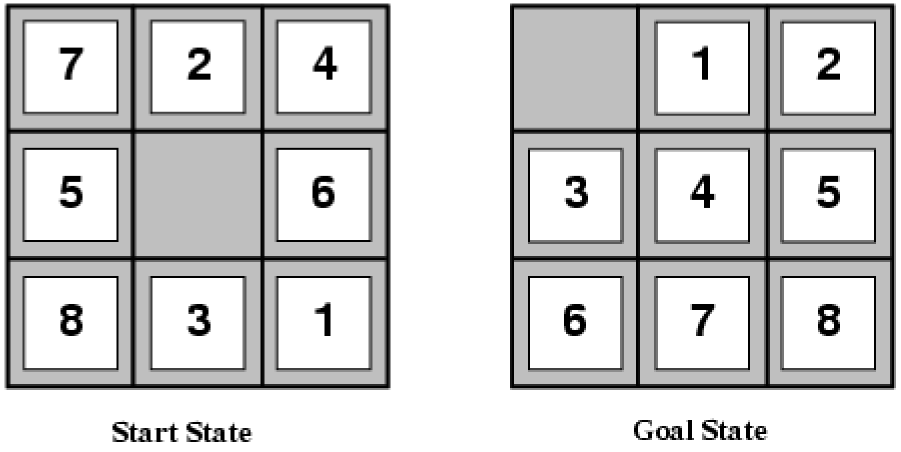
- The generation of new states is done by applying operators (e.g., legal moves) to existing states on a path
- The path from the start state to a goal state contains the series of operations that lead to the problem solution

The 8-Puzzle



- states?
- actions?
- goal test?
- path cost?

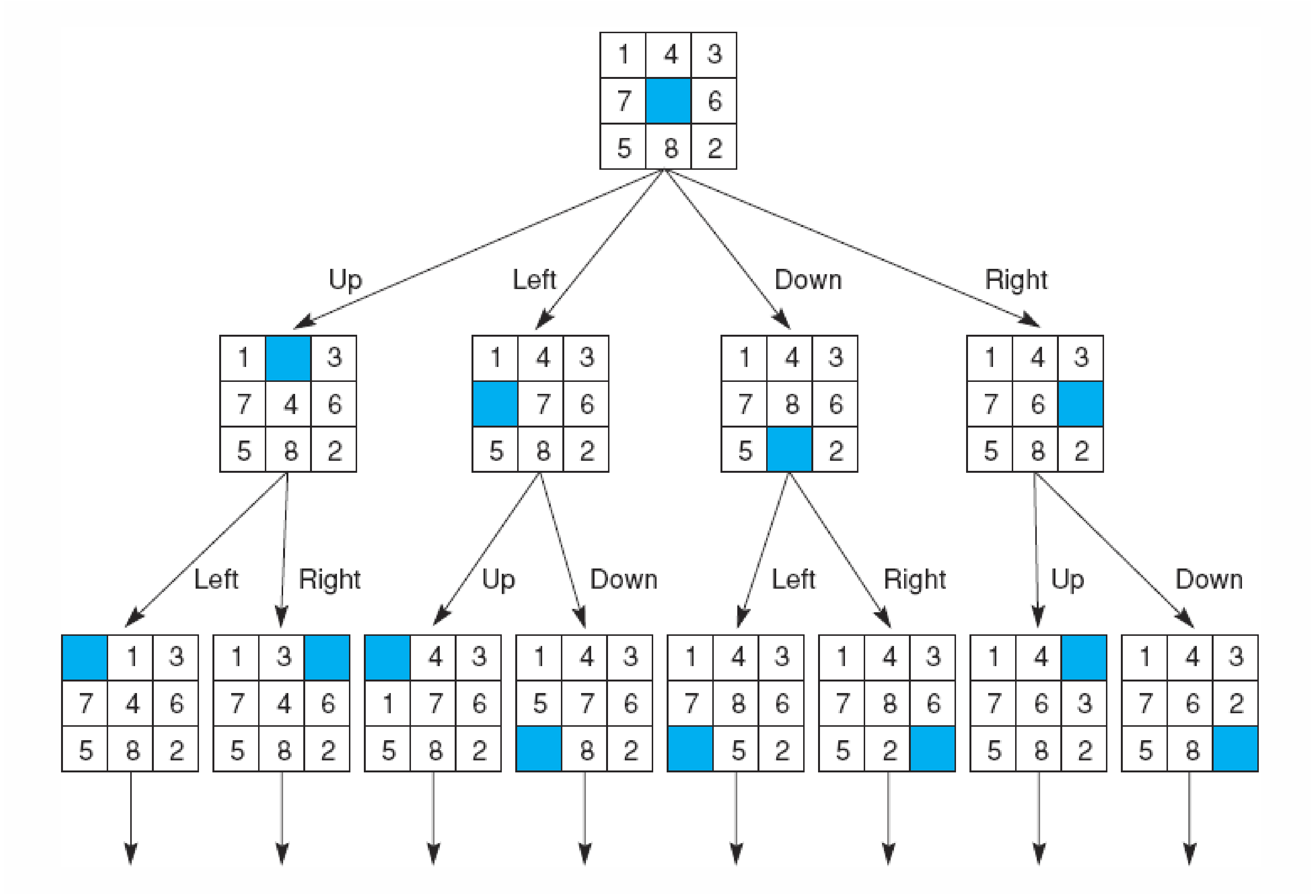
The 8-Puzzle



- states? locations of tiles (2D array)
- actions? move blank left, right, up, down
- goal test? goal state (given)
- path cost? 1 per move

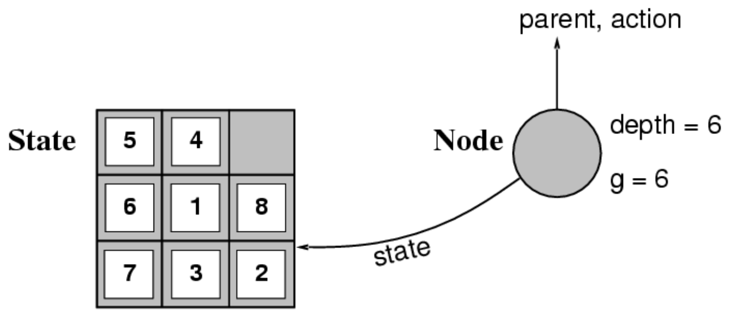
[Note: optimal solution of *N*-Puzzle family is NP-hard]

The 8-Puzzle



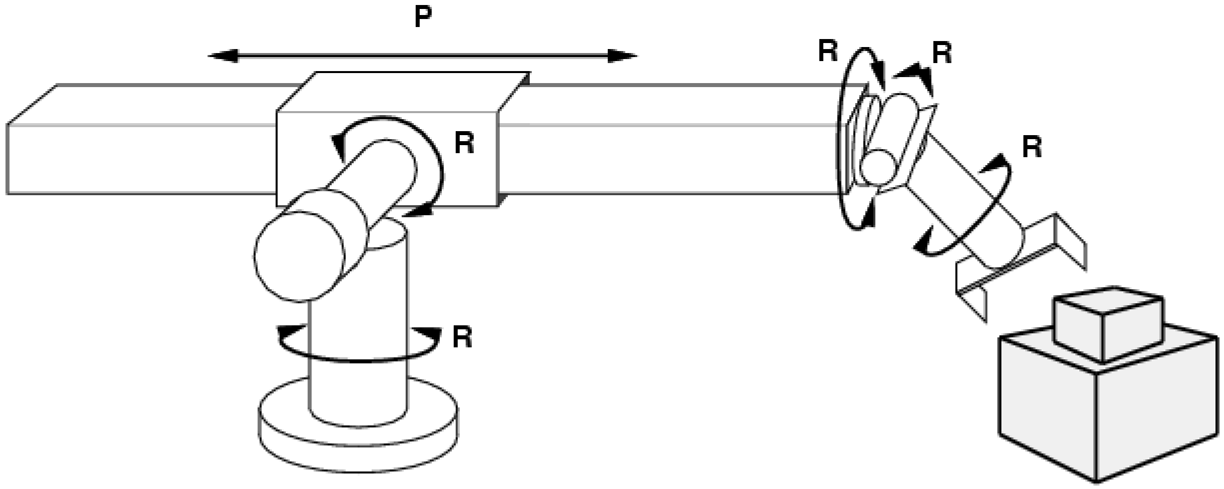
States vs. Nodes

- A *state* is a representation of a physical configuration
- A *node* is a data structure that represents part of a search tree. A node includes state, parent node, action, the path cost *g(x)*, and depth



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

Robotic Assembly



states?	coordinates of robot joints and angles
actions?	new coordinates of robot joints and angles
goal test?	complete assembly of part
path cost?	time to execute

Missionaries and Cannibals

Problem Description:
Three missionaries and three cannibals are on one side of a river, with a canoe.
They all want to get to the other side of the river.
The canoe can only hold one or two people at a time.
At no time should there be more cannibals than missionaries on either side of the river as this would probably result in the missionaries being eaten.
How can they all cross the river and end up on the other side safely?

Missionaries and Cannibals

- The first step in solving the problem is to choose a suitable representation
- Represent the number of cannibals, missionaries, and canoes on each side of the river as a three element array of integers

  [number of cannibals, number of missionaries, number of canoes]

- Start state is therefore:

  <u>left side of river</u>   <u>right side of river</u>

  3,3,1   0,0,0

- An example of a state that must be avoided is **2,1,1**

Missionaries and Cannibals

- In fact, since the system is closed, we only need to represent one side of the river, as we can deduce the other side
- We will represent the right side of the river, and omit the left side
- So start state is:

0,0,0

Missionaries and Cannibals

Now we have to choose suitable operators that can be applied:

1. Move one cannibal across the river
2. Move two cannibals across the river
3. Move one missionary across the river
4. Move two missionaries across the river
5. Move one missionary and one cannibal

Missionaries and Cannibals

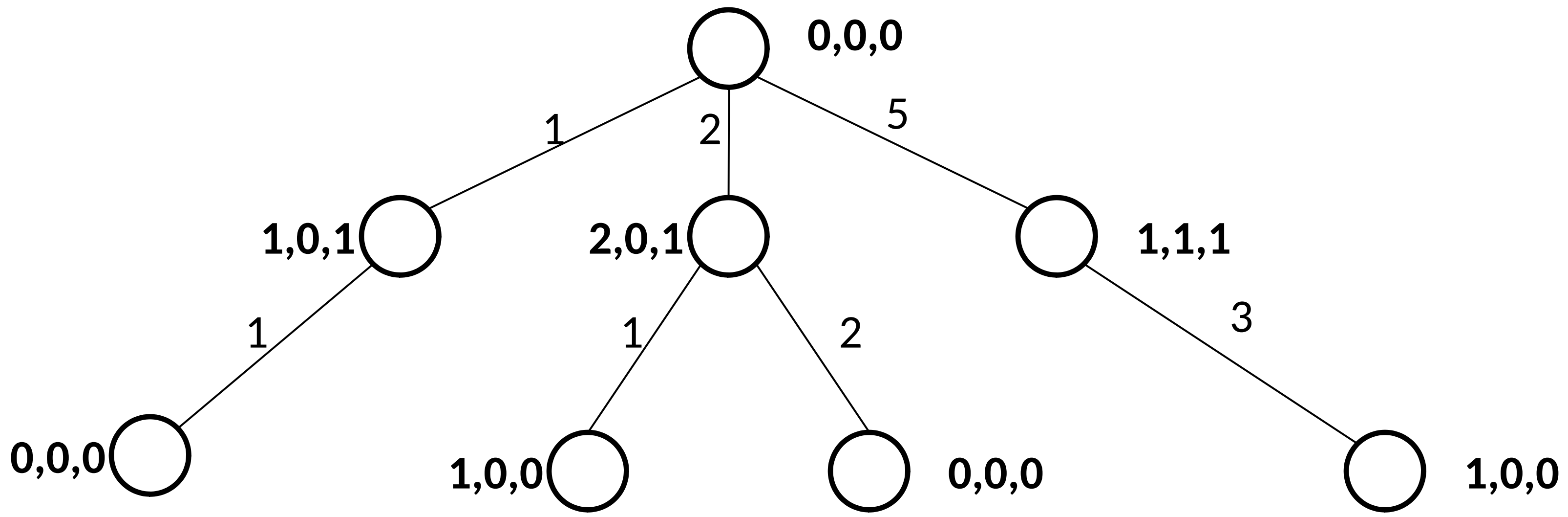
- So if we applied operator 5 (move one cannibal and one missionary to the right side) to the state represented by **1,1,0** then we would result in state **2,2,1**. One cannibal, one missionary, and the canoe have now moved to the right side
- Applying operator 3 (move one missionary to the left side) to this state would lead to an illegal/failing state **2,1,0**

Missionaries and Cannibals

- In addition, we need to have a test that can identify if we have reached the goal state **3,3,1** (or 3,3,0 but this is not possible)
- The path cost could be the number of steps taken, or the number of times an operator is applied
  - In some cases it is desirable to find a solution that minimizes cost

Missionaries and Cannibals

Below is the first three levels of the search tree for this problem (edges are labeled with the operator that has been applied):



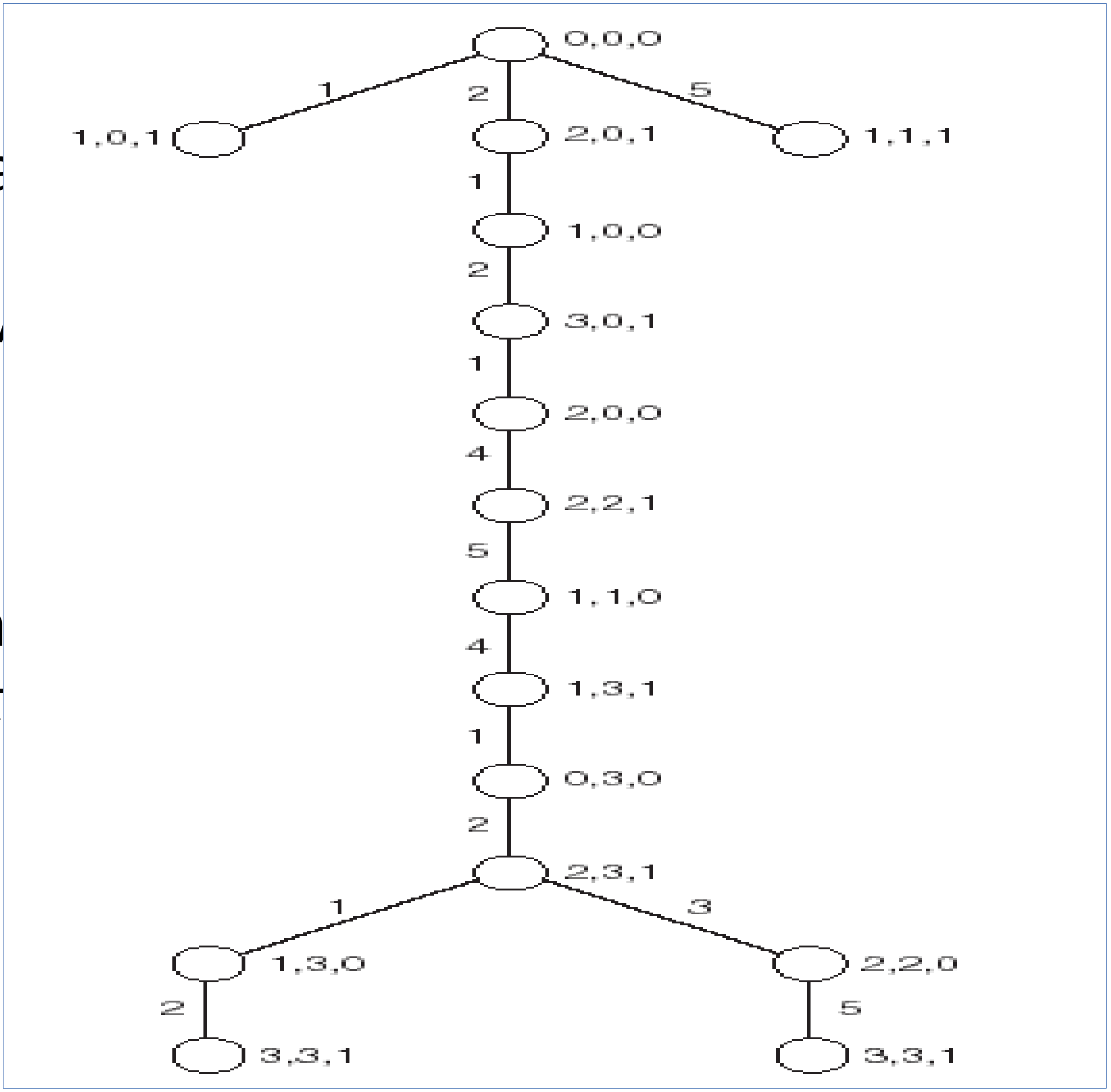
By extending this path to include all possible paths, and the states those paths lead to, a solution can be found. A solution to the problem would be represented as a path from the root node to a goal node.

Missionaries and Cannibals

- Note that this tree has some cycles
  - Applying operator 1 (moving one cannibal to the other side), and then applying the same operator again, we return to the start state
- This is a perfectly valid way to try to solve the problem, but not a very efficient one
- A more effective representation would be one that did not include any cycles

Missionaries And Cannibals Search Tree

- Cycles and repea
  states
- have been remov
- Nodes represent
- edges represent
  operators
- There are two pa
- lead to the solut



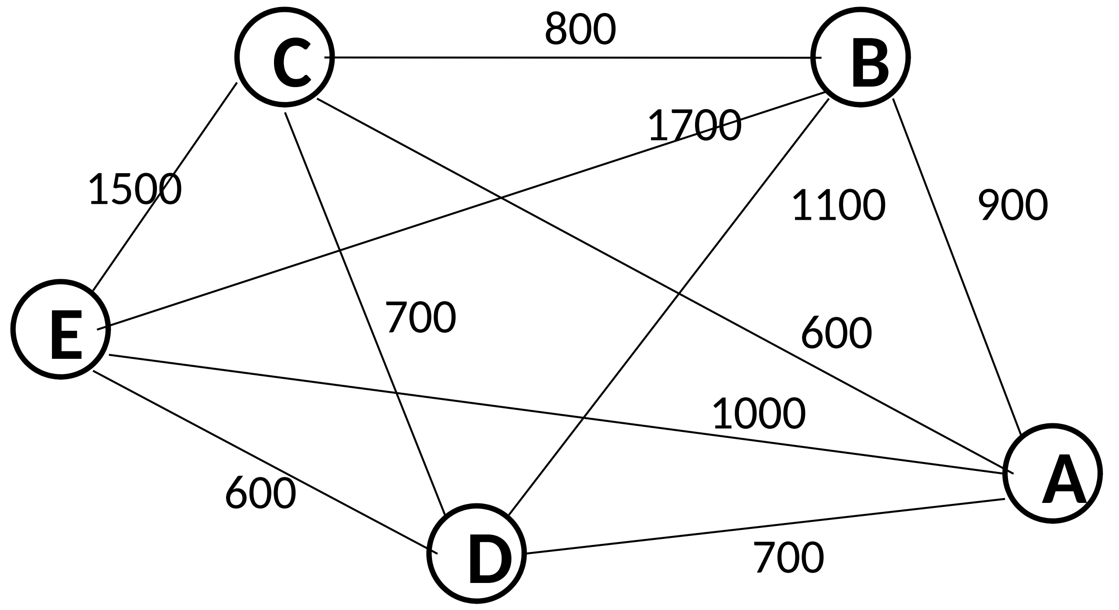
Traveling Salesperson (TSP)

Problem Description:

A salesperson must visit each of a set of cities and then return home. The problem is to find the shortest path that lets the salesperson visit each city. The salesperson lives in Atlanta and is traveling by plane (assume that direct flights are possible between any pair of cities).

City A – Atlanta
City B – Boston
City C – Chicago
City D – Dallas
City E – El Paso

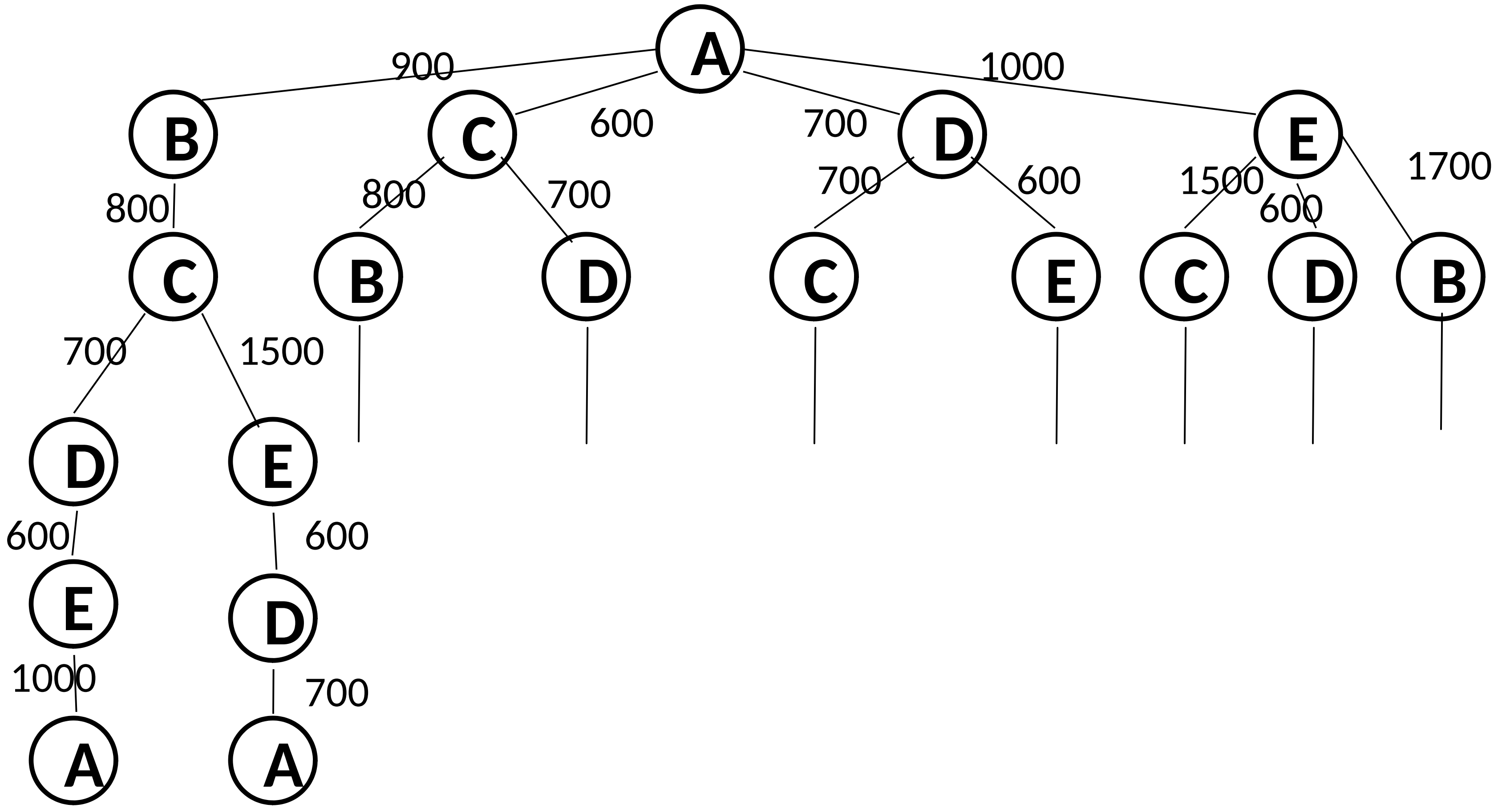
Traveling Salesperson (TSP)



Traveling Salesperson (TSP)

- To solve this problem using search, a different representation is needed
- A node in the search tree represents a city that has been reached by the path up to that point
  - Each node represents the path from city A to the city named at that node
- Edges represent the distance to the next city on the path

Traveling Salesperson



Traveling Salesperson (TSP)

- As with the Missionaries and Cannibals (M&C) example, cyclical paths have been excluded from the tree
- Unlike the M&C tree, this tree does allow repeated states
  - This is because in this problem each state must be visited once, and so a complete path must include all states

Combinatorial Explosion

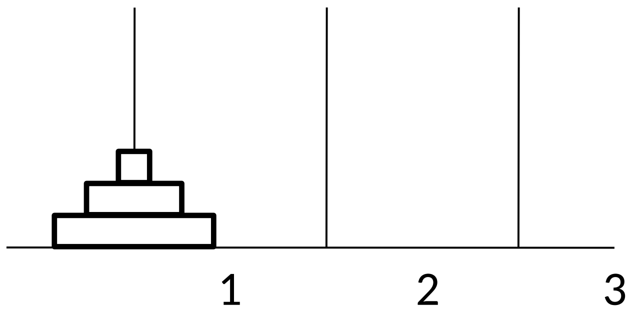
- In total there will be (n-1)! possible paths for a Traveling Saleperson Problem (TSP) with n cities
  - This is because we are constrained in our starting city and, thereafter, have a choice of any combination of (n-1) cities
  - In problems with a small number of cities such as 5 or even 10 this means that the complete search tree can be evaluated by a computer program without much difficulty
  - If the problem consists of 40 cities, then there would be 39! paths, which is roughly $2*10^{46}$
  - Methods that try to examine all of these paths are called **brute-force search methods**
  - To solve search problems with large trees **heuristics** must be used

Towers of Hanoi

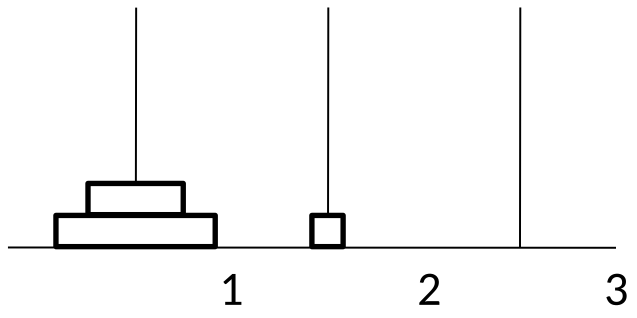
Problem Description:

You have three pegs and a number of disks of different sizes. The aim is to move from the starting state, where all the disks are on the first peg (in size order smallest on top) to the goal state where all the pegs are on the third peg, also in size order. You are allowed to move one disk at a time, as long as there are no disks on top of it, and as long as you do not move it on top of a peg that is smaller than it.

Towers of Hanoi



Start state



State after one disk has been moved from peg 1 to peg 2

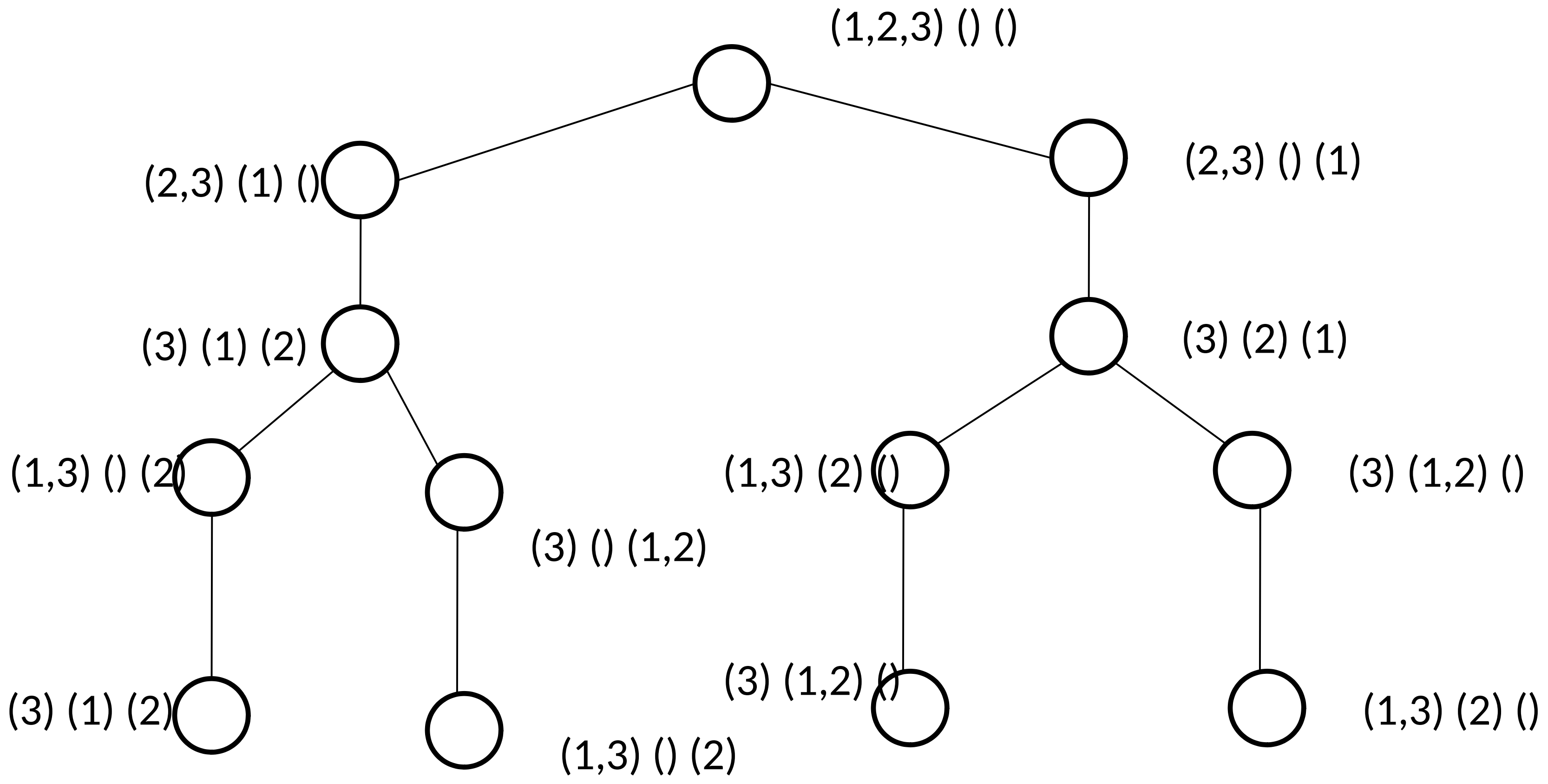
Towers of Hanoi

Operators:

1. Move disk from peg 1 to peg 2
2. Move disk from peg 1 to peg 3
3. Move disk from peg 2 to peg 3
4. Move disk from peg 1 to peg 2
5. Move disk from peg 3 to peg 1
6. Move disk from peg 3 to peg 2
7. Move disk from peg 1 to peg 2

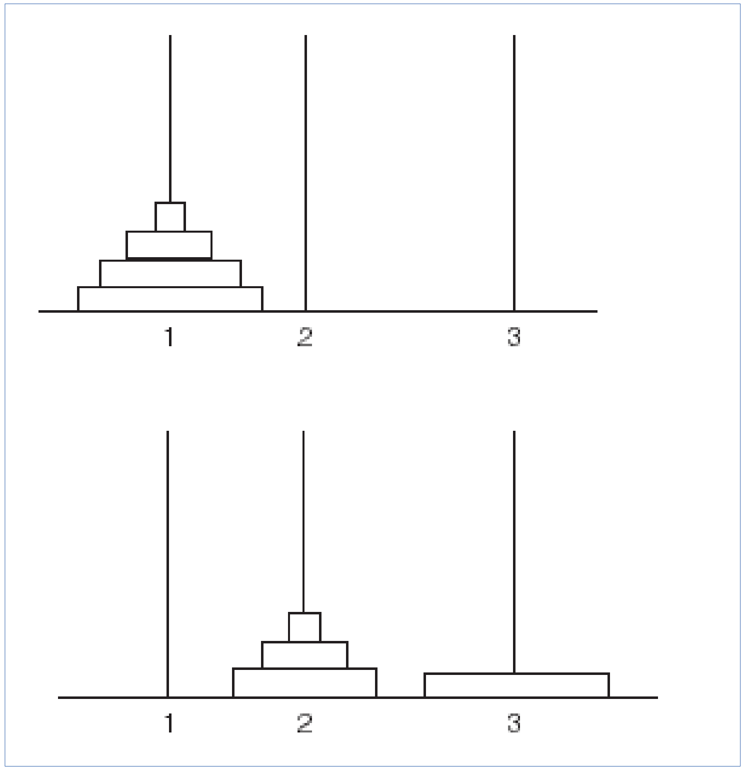
Represent a state using vectors of numbers, where 1 represents the smallest disk and 3 the largest disk.
Starting state: (1,2,3) () () and goal state: () (1,2,3) ()
State on right in previous slide: (2,3) (1) ()

Towers of Hanoi



Problem Reductions

- To solve the Towers of Hanoi problem with 4 disks, you can first solve the same problem with 3 disks
- The solution to the overall problem is solved recursively on smaller sized problems:
- Step 1: Move N-1 disks from Peg 1(source) to Peg 2 (spare)
- Step 2: Move **one** disk from Peg 1 (source) to Peg 3 (destination)
- Step 3: Move N-1 disks from Peg 2 (spare) to Peg 3 (destination)



Search Strategies

A state space may be searched in two directions:

- From the given data of a problem instance toward a goal (data-driven, or *forward chaining*)
- From a goal back to the data (goal-driven or *backward chaining*)

Search Strategies

Use data-driven search if:

1. All or most of the data are given in the initial problem statement
2. There are a large number of potential goals, but there are only a few ways to use the facts and given information of a particular problem instance
3. It is difficult to form a goal or hypothesis

Search Strategies

Use goal-driven search if:

1. A goal or hypothesis is given in the problem statement or can easily be formulated (e.g., math theorem or medical diagnosis)
2. There are a large number of rules that match facts of the problem and thus produce an increasing number of conclusions or goals
3. Problem data is not given and must be acquired

Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - Completeness: Does it always find a solution if one exists?
  - Optimality: Does it always find a least-cost solution?
  - Time complexity: The number of nodes generated
  - Space complexity: The maximum number of nodes in memory
- Time and space complexity are measured in terms of
  - *b*: Maximum branching factor of the search tree (i.e. number of successive nodes of a parent)
  - *d*: Depth of the least-cost solution
  - *m*: Maximum depth of the state space (may be ∞)

Brute-Force Search

- Search methods that examine every node in the search tree – also called **exhaustive**
- “Generate and Test” is the simplest brute force search method:
  - **Generate possible solutions to the problem**
  - **Test each one in turn to see if it is a valid solution**
  - **Stop when a valid solution is found**
- The method used to generate possible solutions must be carefully chosen

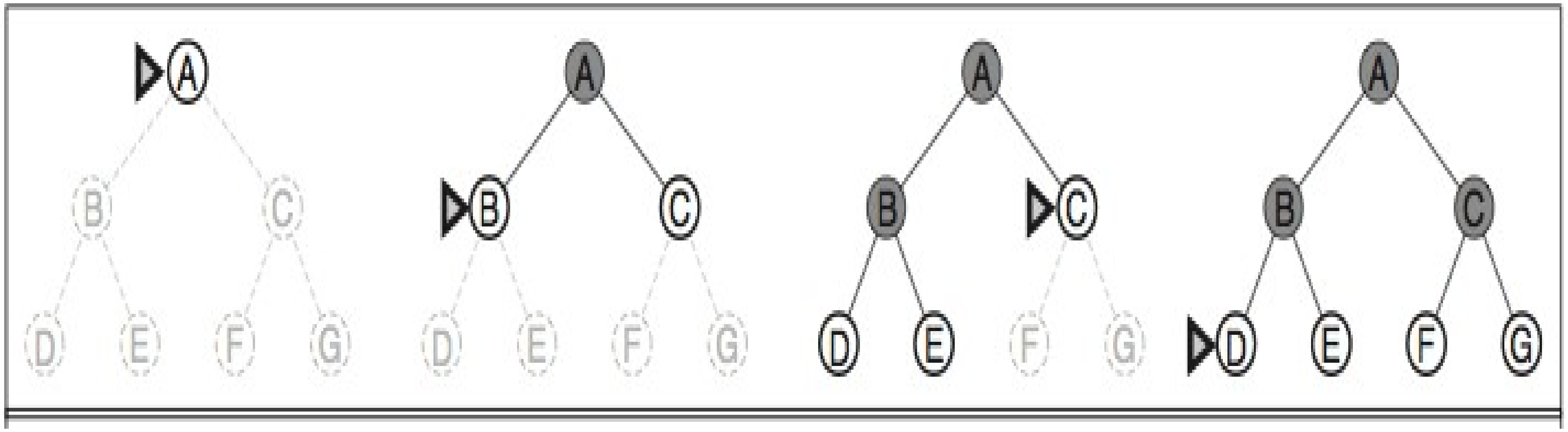
Uninformed Search Strategies

- Uninformed search strategies use only the information available in the problem definition
  - Breadth-first search
  - Depth-first search
  - Uniform-cost search (Dijkstra’s shortest path)
  - Depth-limited search
  - Iterative deepening search

Breadth-First Search

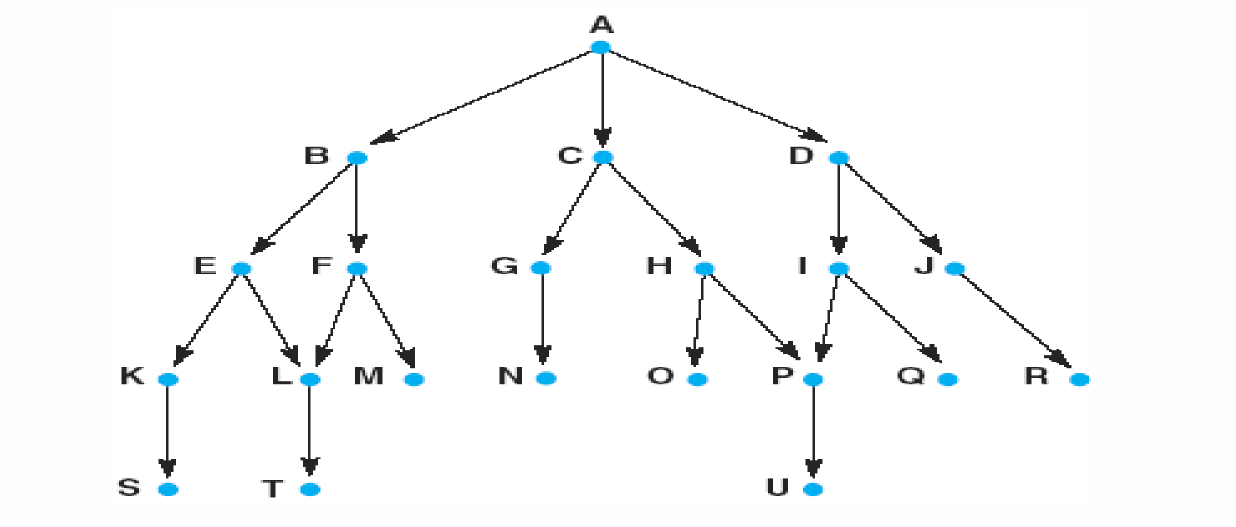
```
function breadth_first_search:

    open := [Start]                                              % initialize
    closed := [ ]
    while open ≠ [ ] do                                          % states remain
        Remove (dequeue) leftmost state from open, call it X
        if X is goal then return SUCCESS                         % goal found
        else
            generate children of X
            if any child of X is goal then return SUCCESS        % goal found
            else
                put X on closed
                discard children of X if on open or closed       % loop check
                put remaining children on right end of open (enqueue)
    return FAILURE
```

Breadth-First Search

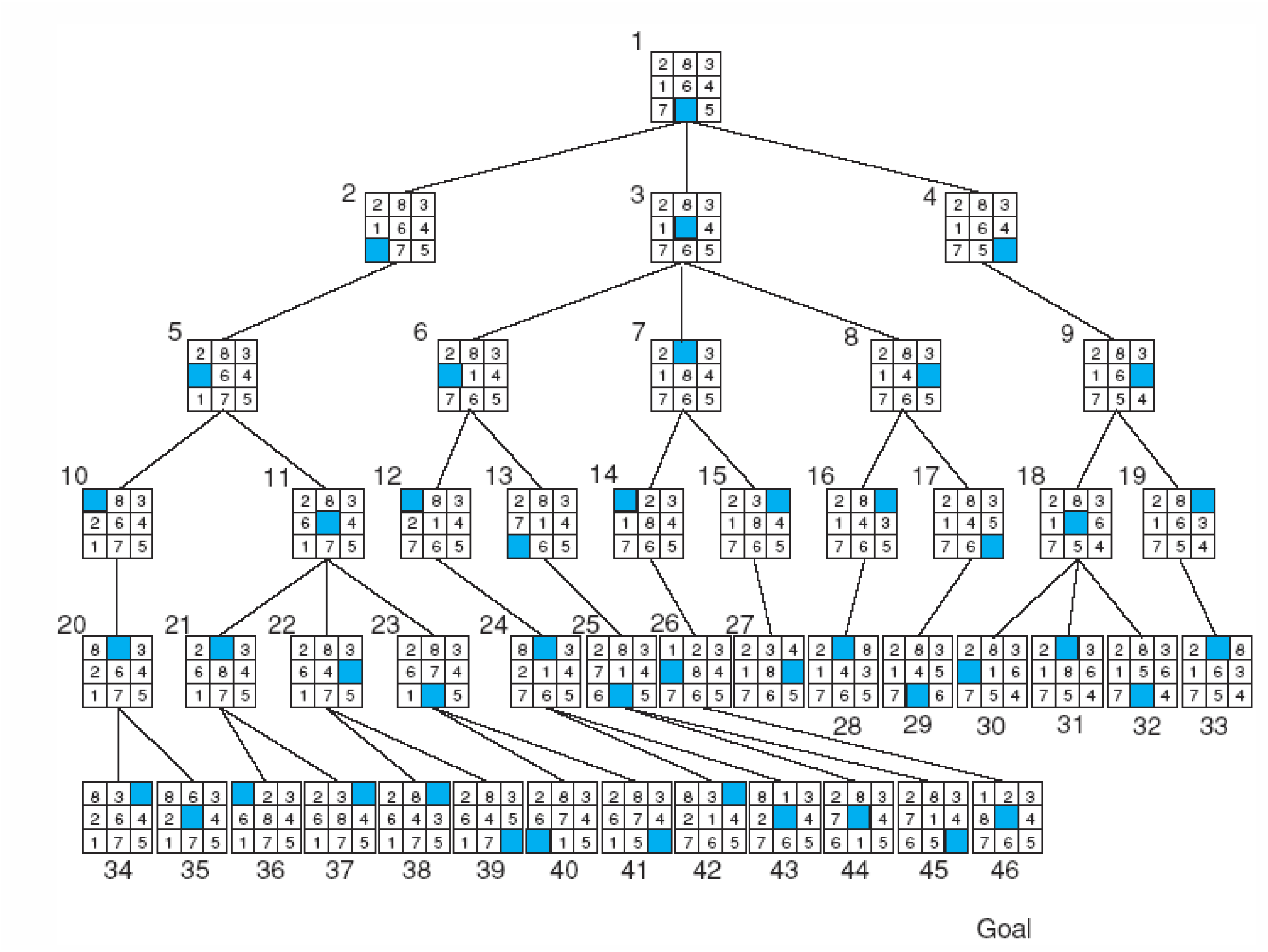


Breadth-First Search

1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [C,D,E,F]; closed = [B,A]**
4. **open = [D,E,F,G,H]; closed = [C,B,A]**
5. **open = [E,F,G,H,I,J]; closed = [D,C,B,A]**
6. **open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]**
7. **open = [G,H,I,J,K,L,M]** (as L is already on open); **closed = [F,E,D,C,B,A]**
8. **open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]**
9. and so on until either U is found or **open** = [ ]



Breadth-First Search of 8-Puzzle



Properties of Breadth-First Search

- Complete?
  - Yes (if *b* is finite)
- Time?

$1 + b + b^2 + b^3 + \dots + b^d + b^{d+1}$ = $O(b^{d+1})$

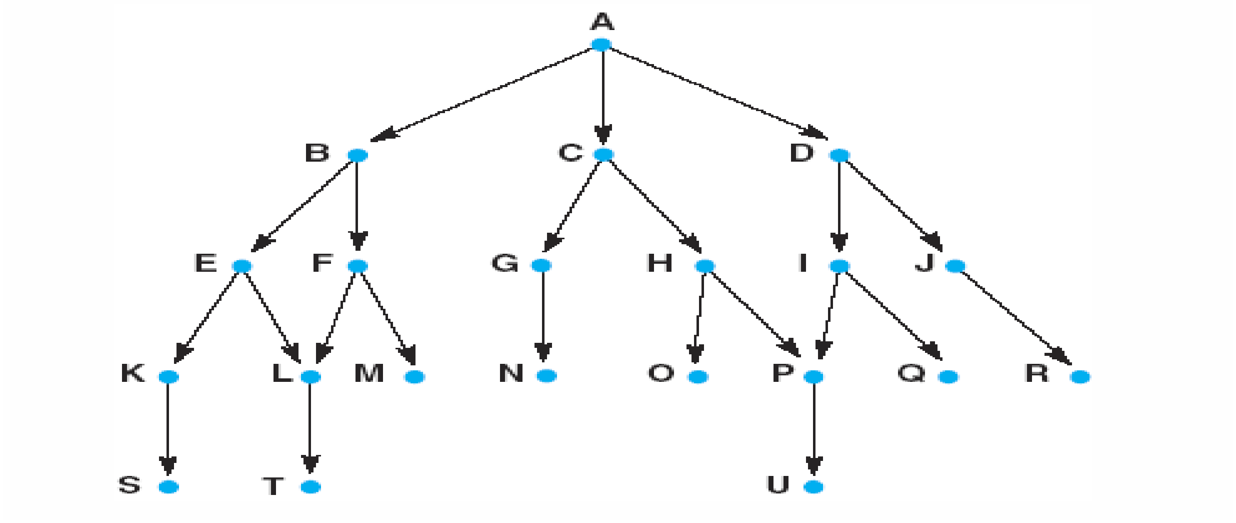
- Space?
  - $O(b^d)$ (keeps every node in memory)
- Optimal?
  - Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

Depth-First Search

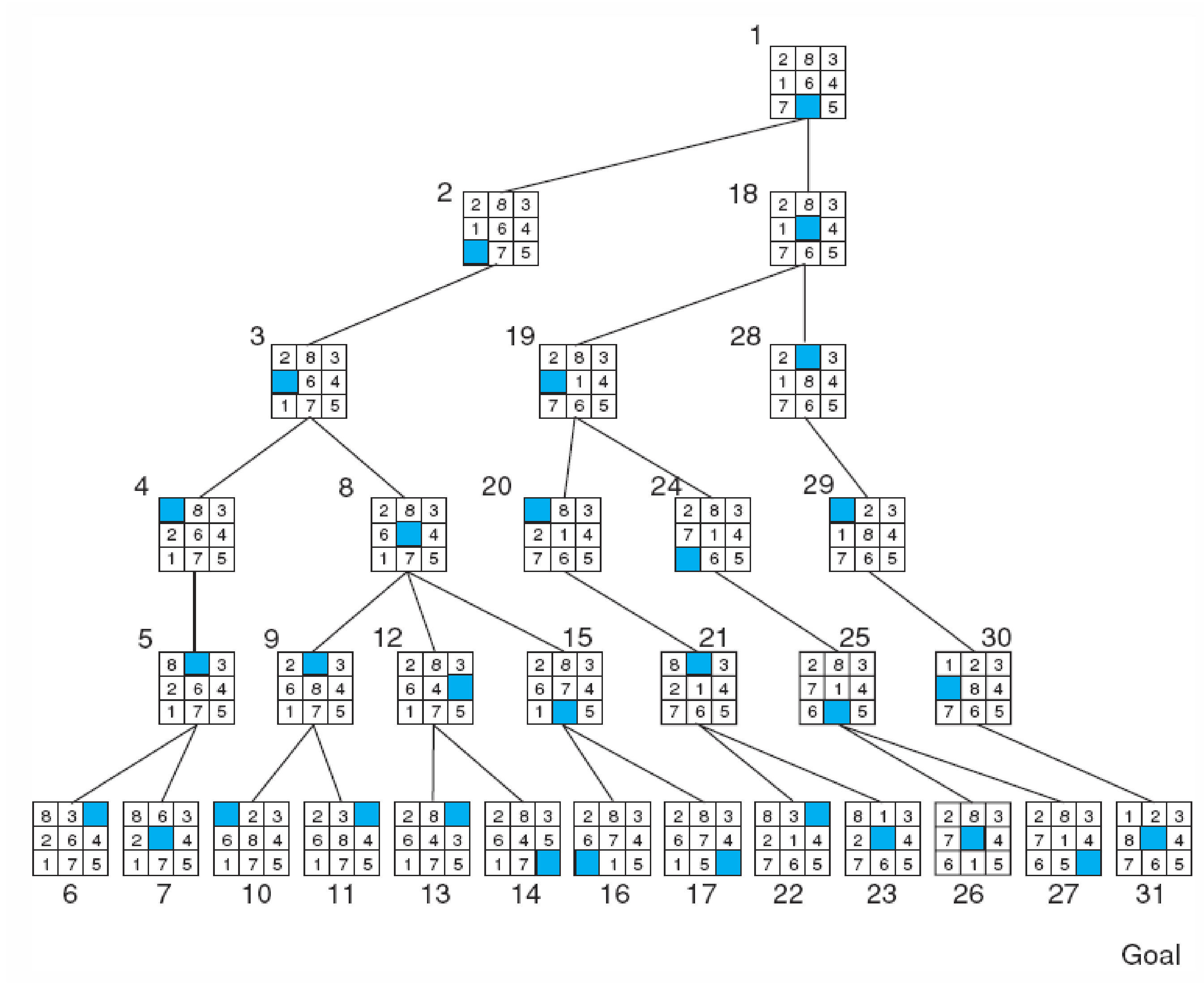
```
function depth_first_search:
    open := [Start]                                          % initialize
    closed := [ ]
    while open ≠ [ ] do                                      % states remain
        remove leftmost state from open, call it X
        if X is goal then return SUCCESS                     % goal found
        else
            generate children of X
            if any child of X is goal then return SUCCESS    % goal found
            else
                put X on closed
                discard children of X if on open or closed   % loop check
                put remaining children on left end of open
    return FAILURE
```

Depth-First Search

1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [E,F,C,D]; closed = [B,A]**
4. **open = [K,L,F,C,D]; closed = [E,B,A]**
5. **open = [S,L,F,C,D]; closed = [K,E,B,A]**
6. **open = [L,F,C,D]; closed = [S,K,E,B,A]**
7. **open = [T,F,C,D]; closed = [L,S,K,E,B,A]**
8. **open = [F,C,D]; closed = [T,L,S,K,E,B,A]**
9. **open = [M,C,D],** as L is already on **closed**; **closed = [F,T,L,S,K,E,B,A]**
10. **open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]**
11. **open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]**

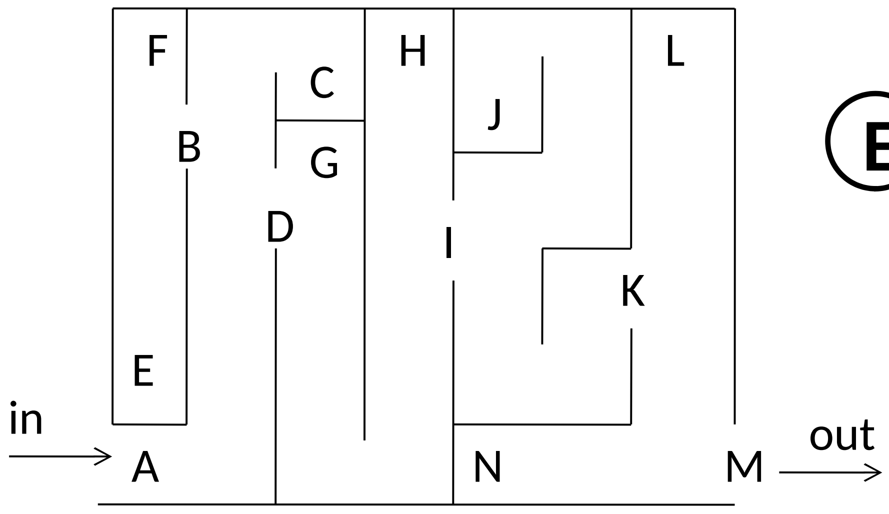


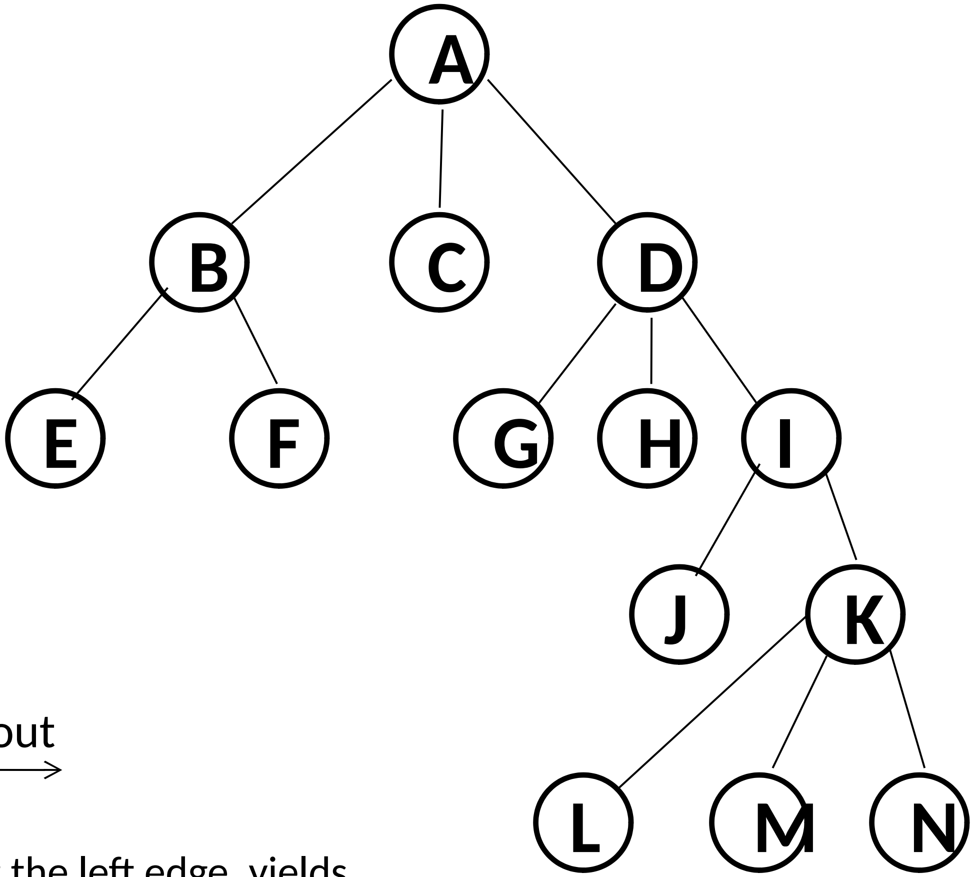
Depth-First Search of 8-Puzzle



Depth-First Search of a Maze

A is the entrance to the maze
M is the exit from the maze
C, E, F, G, H, J, L, and N are dead ends
B, D, I, and K are points in the maze where a choice can be made as to which direction to go next





Running one’s hand along the left edge, yields
A, B, E, F, C, D, G, H, I, J, K, L, M

Properties of Depth-First Search

- Complete? No: fails in infinite-depth spaces and 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

Breadth-First vs. Depth-first

Scenario	Depth-First	Breadth-First
Some paths are extremely long, or even infinite	Performs badly	Performs well
All paths are of similar length	Performs well	Performs well
All paths are of similar length, and all paths lead to a goal state	Performs well	Wasteful of time and memory
High branching factor	Performance depends on other factors	Performs badly

Uniform-Cost Search

- Instead of expanding the shallowest node, as in breadth-first search, expand the node with the *lowest path cost*
  - Note that if all step costs are equal, this is identical to breadth-first search
  - May get stuck in an infinite loop if it ever expands a node with a zero-cost action leading back to the same state
  - Fix by guaranteeing that every step is greater than or equal to some small positive constant, ε. (this means that the cost of a path always increases as we go along a path)
  - The nodes will be expanded in order of increasing path cost
  - Thus, the first goal node selected for expansion is the optimal solution

Properties of Uniform-Cost Search

- Complete?
  - Yes (if step cost >= ε for positive ε)
- Time?
  - $O(b^{\lfloor C^*/\varepsilon \rfloor + 1})$ where C^* is the cost of the optimal solution
- Space?
  - $O(b^{\lfloor C^*/\varepsilon \rfloor + 1})$
- Optimal?
  - Yes

Depth-Limited Search

- Depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors
- Recursive implementation

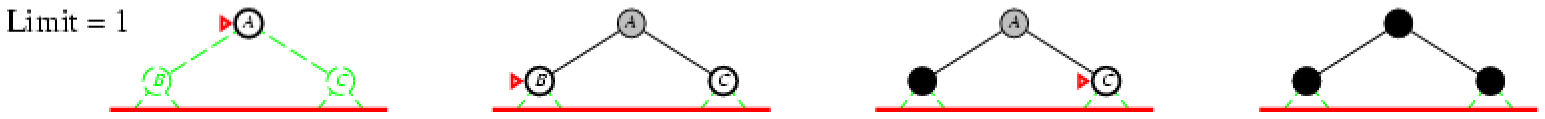
Iterative Deepening Search

- An exhaustive search method based on both depth-first and breadth-first search
- Carries out depth-first search to depth of 1, then to depth of 2, 3, and so on until a goal node is found
- Efficient in memory use, and can cope with infinitely long branches
- Not as inefficient in time as it might appear, particularly for very large trees, in which it only needs to examine the largest row (the last one) once

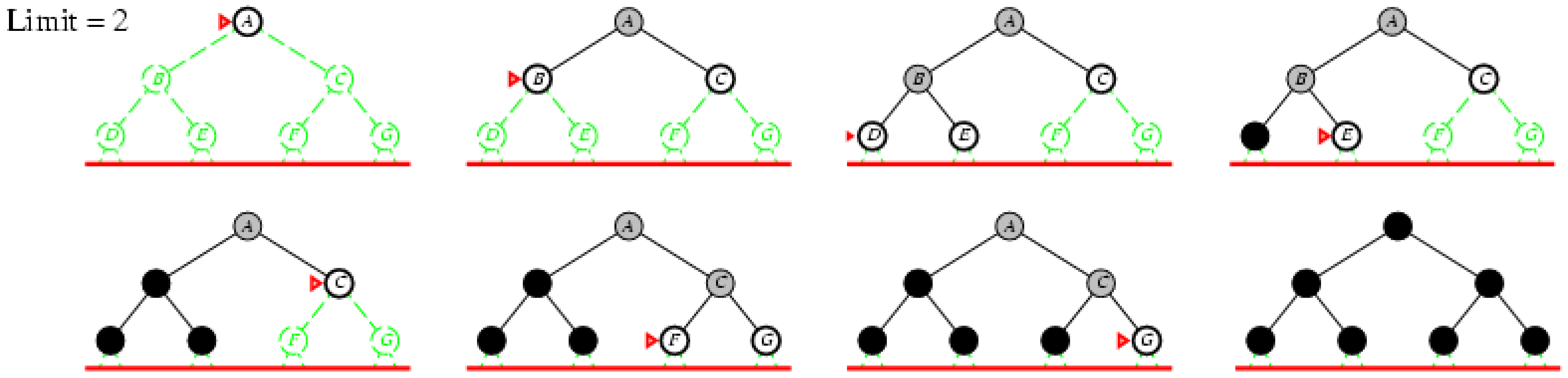
Iterative Deepening Search, *D* = 0

Limit = 0

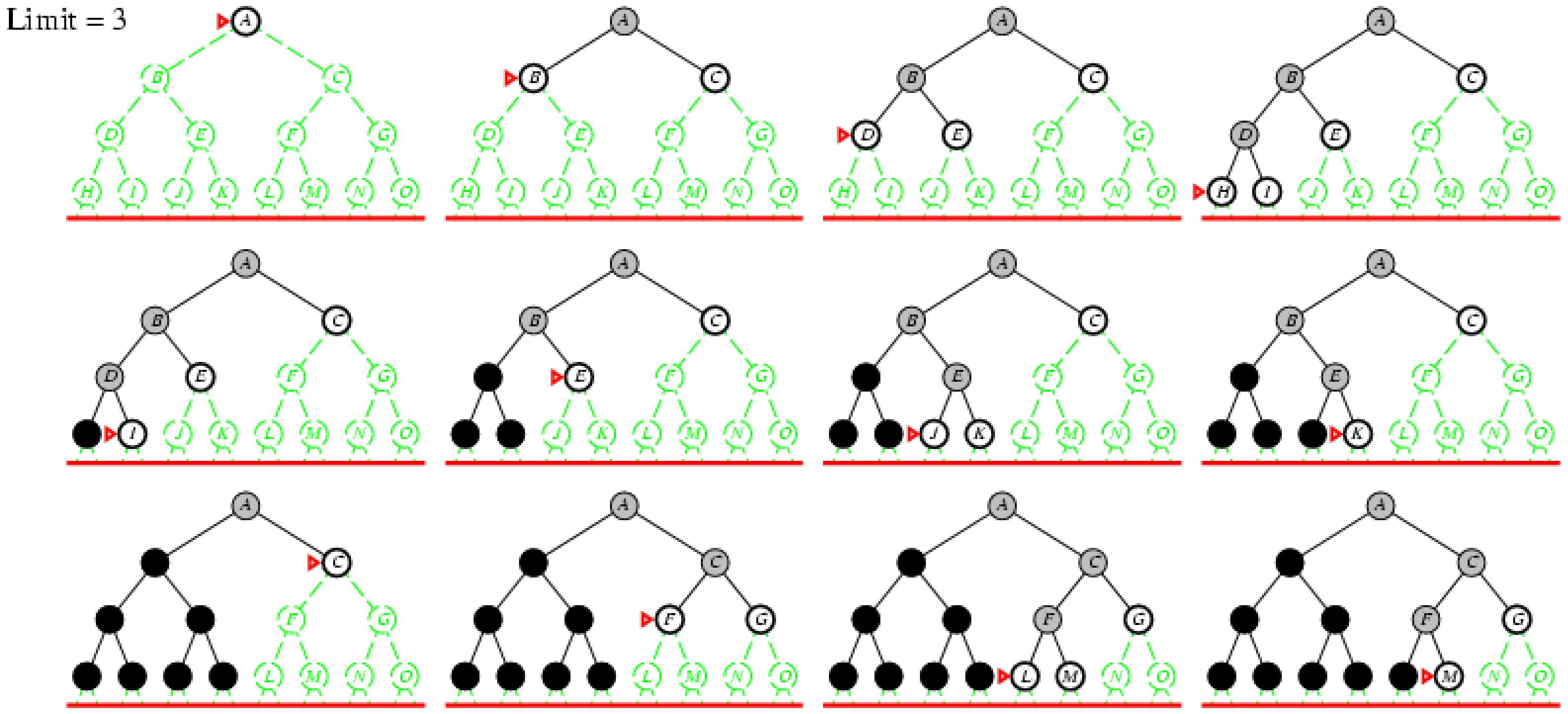
Iterative Deepening Search, *D* = 1



Iterative Deepening Search, *D* = 2



Iterative Deepening Search, *D* = 3



Iterative Deepening Analysis

- IDS combines the efficiency of memory use of depth-first search, with the advantage that branches of the search tree that are infinite or extremely large will not sidetrack the search
- It also shares the advantage of breadth-first search in that it will always find the path that involves the fewest steps through the tree
  - Although this is not necessarily the best path

Iterative Deepening Analysis

- It appears that IDS would be an extremely inefficient way to search a tree because we are repeatedly starting the search at the root
- It is almost as efficient as depth-first or breadth-first because for most trees, the majority of nodes are in the deepest level
  - All three approaches spend most of their time examining these nodes

Iterative Deepening Analysis

For a tree of depth *d* and a branching factor *b*, the total number of nodes is: 1 root node, *b* nodes in the first layer, b^2 nodes in the second layer, … b^n nodes in the n^{th} layer. Hence, the total number of nodes is:

$$1 + b + b^2 + b^3 + \ldots + b^d = (b^{d+1} - 1) / (b - 1)$$

For example, for a tree with a depth of 2 and a branching factor of 2, there are (8-1)/(2-1) = 7 nodes. Using depth-first search or breadth-first search, this means the total number of nodes to be examined is 7.

Iterative Deepening Analysis

Using Iterative Deepening Search, nodes must be examined more than once, resulting in the following progression:

$(d + 1) + b(d) + b^2 (d - 1) + b^3 (d - 2) + \ldots + b^d$

= time complexity of $O(b^d)$

= space complexity of $O(bd)$

For a small tree, $d = 2$ and $b = 2$, IDS is less efficient in time than depth-first search or breadth-first search:

$(2+1) + 2(2) + 2^2 = 11$ nodes visited

Iterative Deepening Analysis

However, for a larger tree, d = 4 and b = 10, the tree has the following number of nodes: (10^5 -1) / (10 -1) = 11,111.

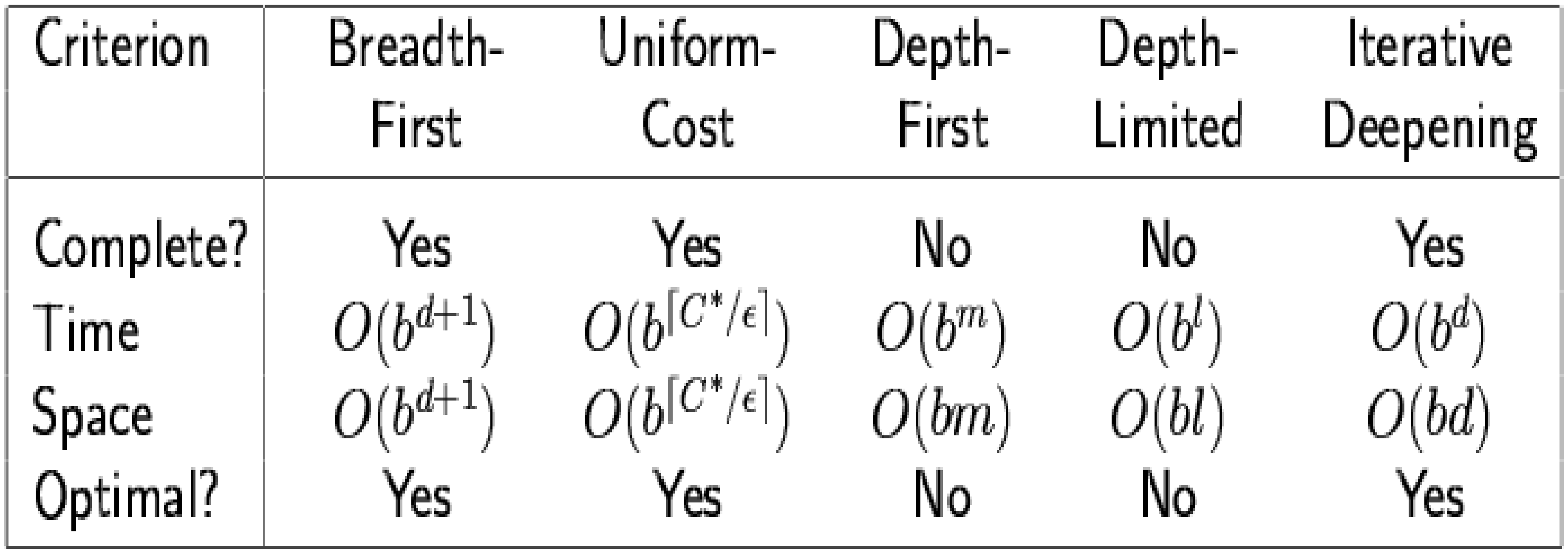
IDS will examine the following number of nodes:
(4+1) + 10(4) + 100(3) + 1,000(2) + 10,000 = 12,345 nodes.

Hence, as the tree gets larger, we see that the majority of nodes to be examined (in this case, 10,000 out of 12,345) are in the last row, which needs to be examined at most once.

Properties of Iterative Deepening

- 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 Search 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