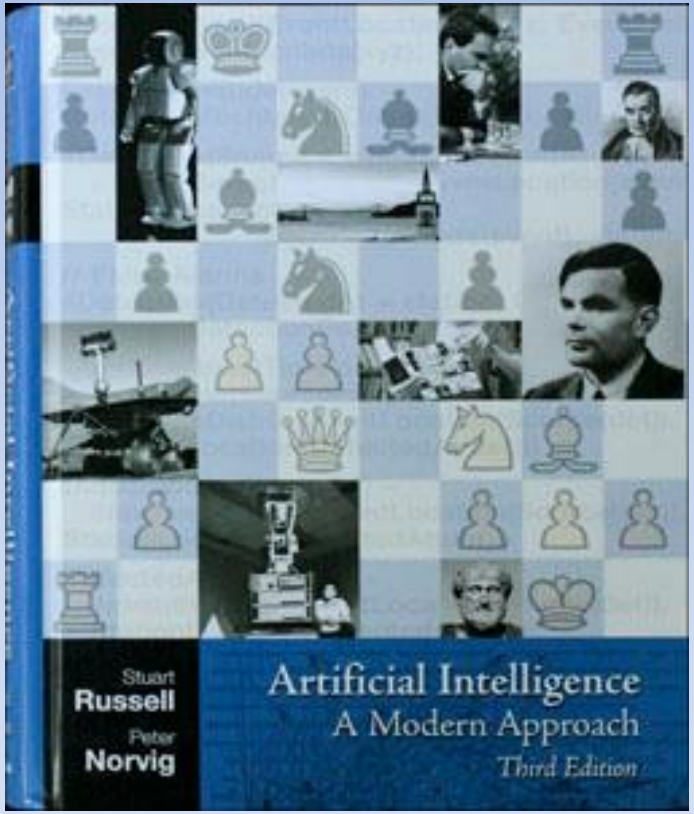
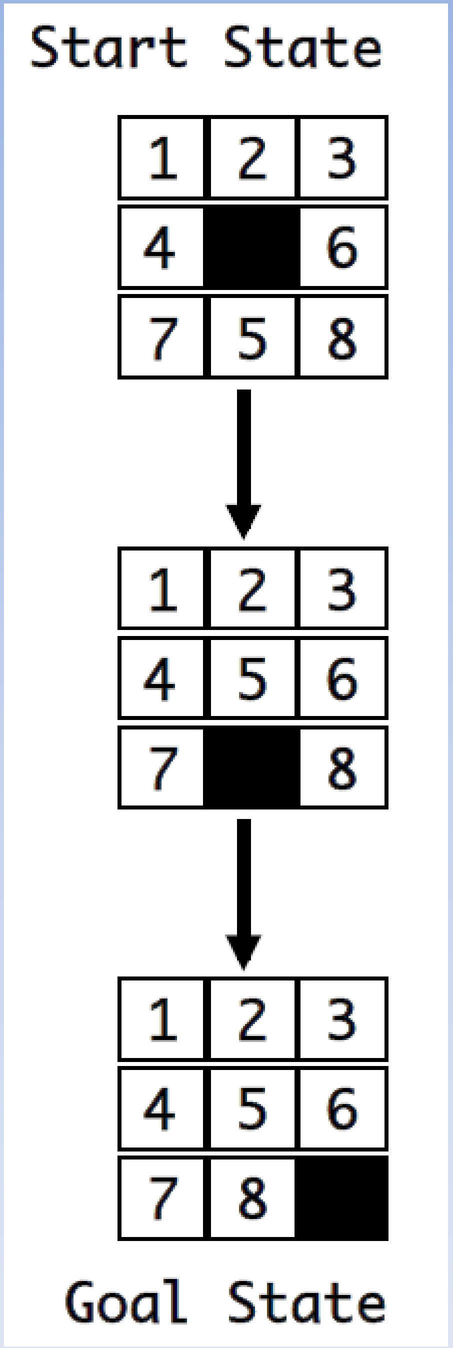
Chapter 3: Uninformed Search





Why is Searching Such Sorrow??

- Search Starts Not Knowing:
  - The size of the tree!
  - The shape of the tree!
  - Depth of the goal states!
- How big can the search tree be?
  - Given a Constant Branching Factor **b**
  - Given One Goal at depth **d**
  - Search tree that includes goal can have
    - **B^n different branches in the tree (worst case)!!**
- Examples:
  - B = 2, d = 10: $b^d = 2^{10} = 1024$
  - b = 10, d= 10: $b^d =$ **10^{10} = 10,000,000,000**

Tree Search Algorithms
Choosing Next Leaf to Turn Over

Def treeSearch(problem)

  ’returns a solution or failure‘

  initialize frontier using problem initial state

  while true:

    if not frontier: **return** None

    choose a leaf node and remove from frontier

    if node is goal state: **return** node.solution

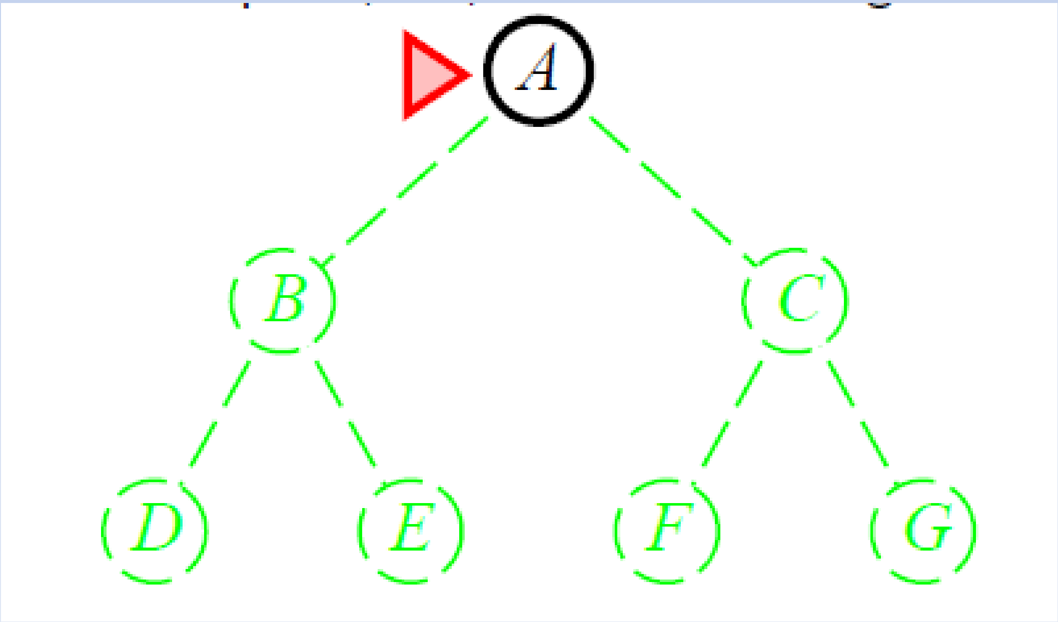
    expand chosen node, adding results to frontier

Search Strategies

- Strategy defined by how we order node expansion in Tree.
- Strategies are Evaluated:
  - Completeness
  - Time Complexity
  - Space Complexity
  - Optimality
- Time and Space Complexity measure by:
  - b: Maximum branching factor
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (∞ ?)

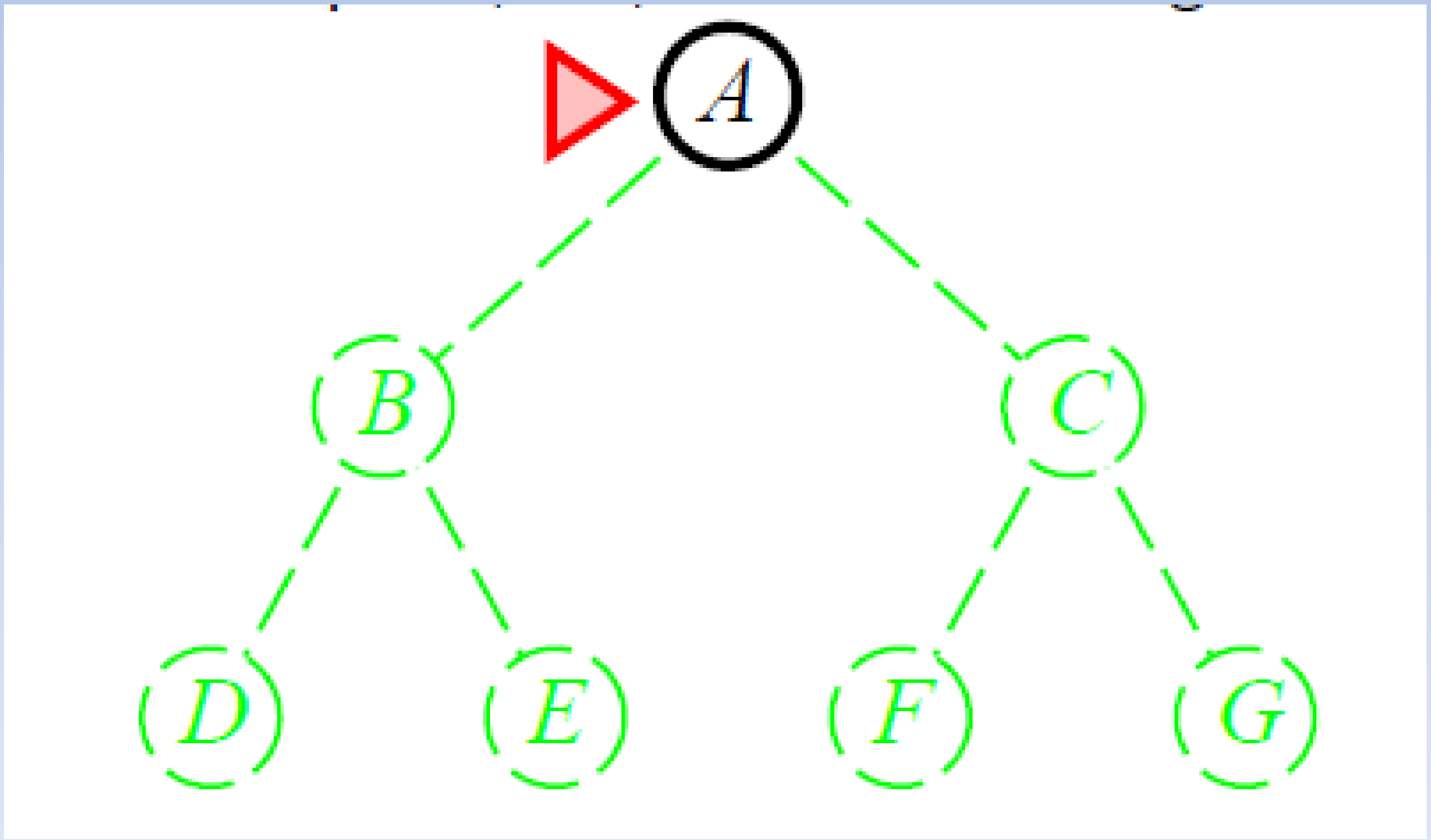
Breadth-First Search

- Strategy: Expand the shallowest unexpanded leaf node.
- Implementation:
  - Frontier is FIFO queue
  - New successors go to the end of the Frontier.



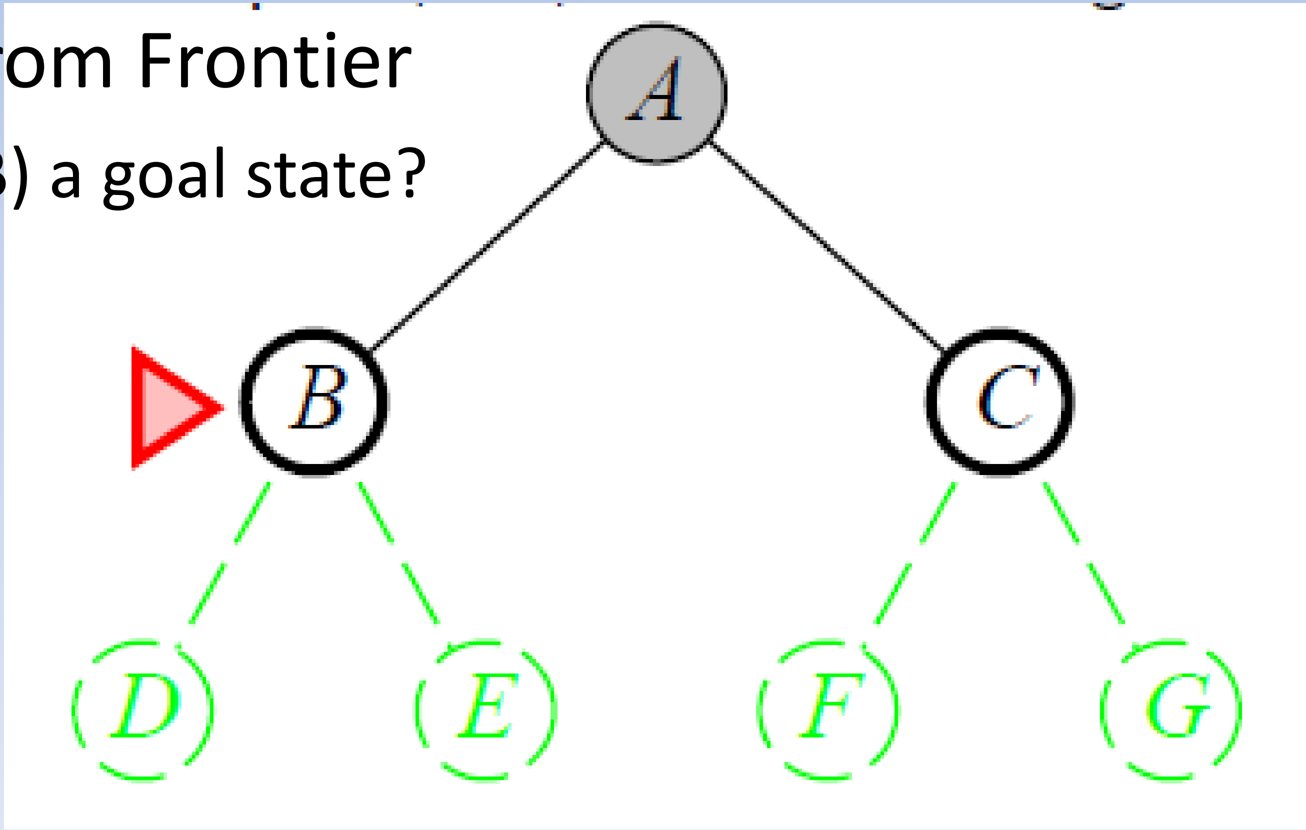
Breadth-First Search

- Is (A) a Goal State



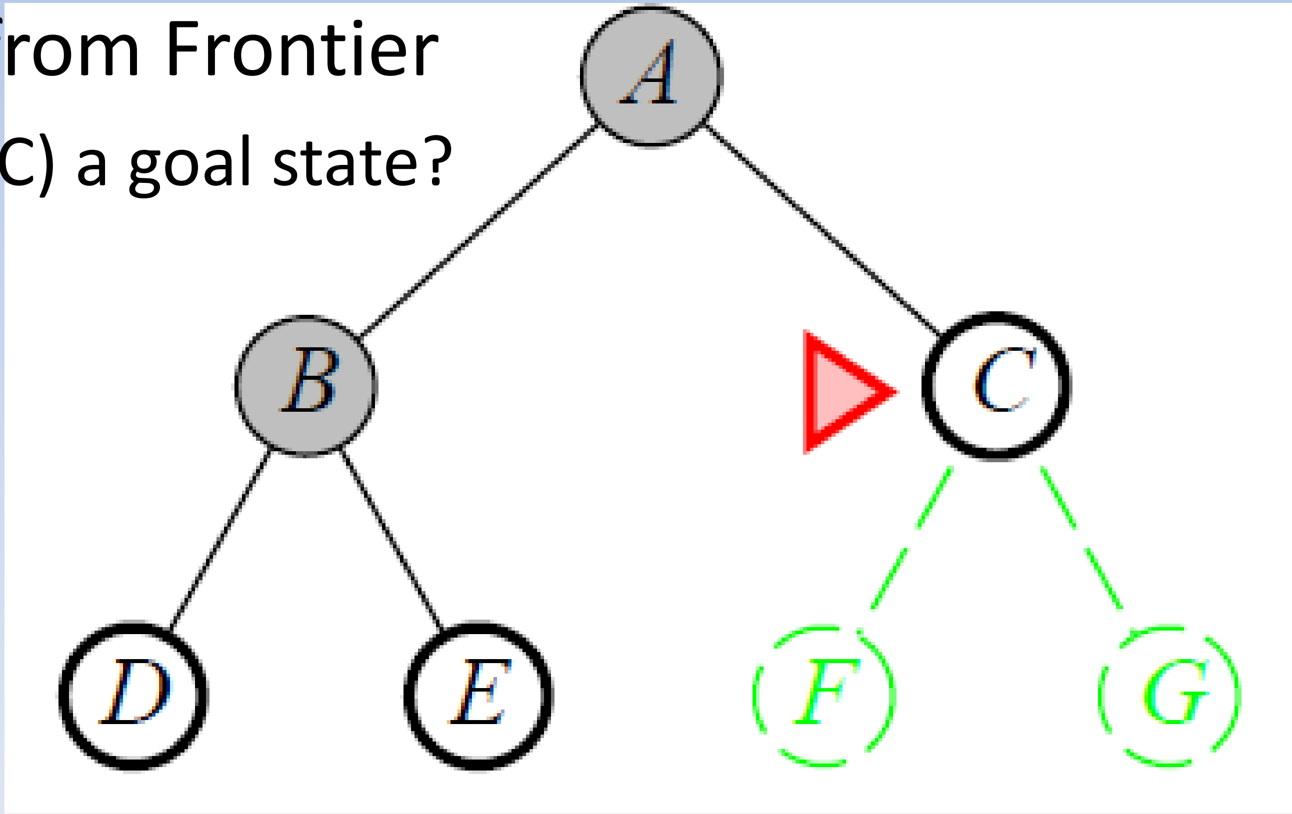
Breadth-First Search

- Expand (A):
  - Frontier = [B, C]
- Pop from Frontier
  - Is (B) a goal state?



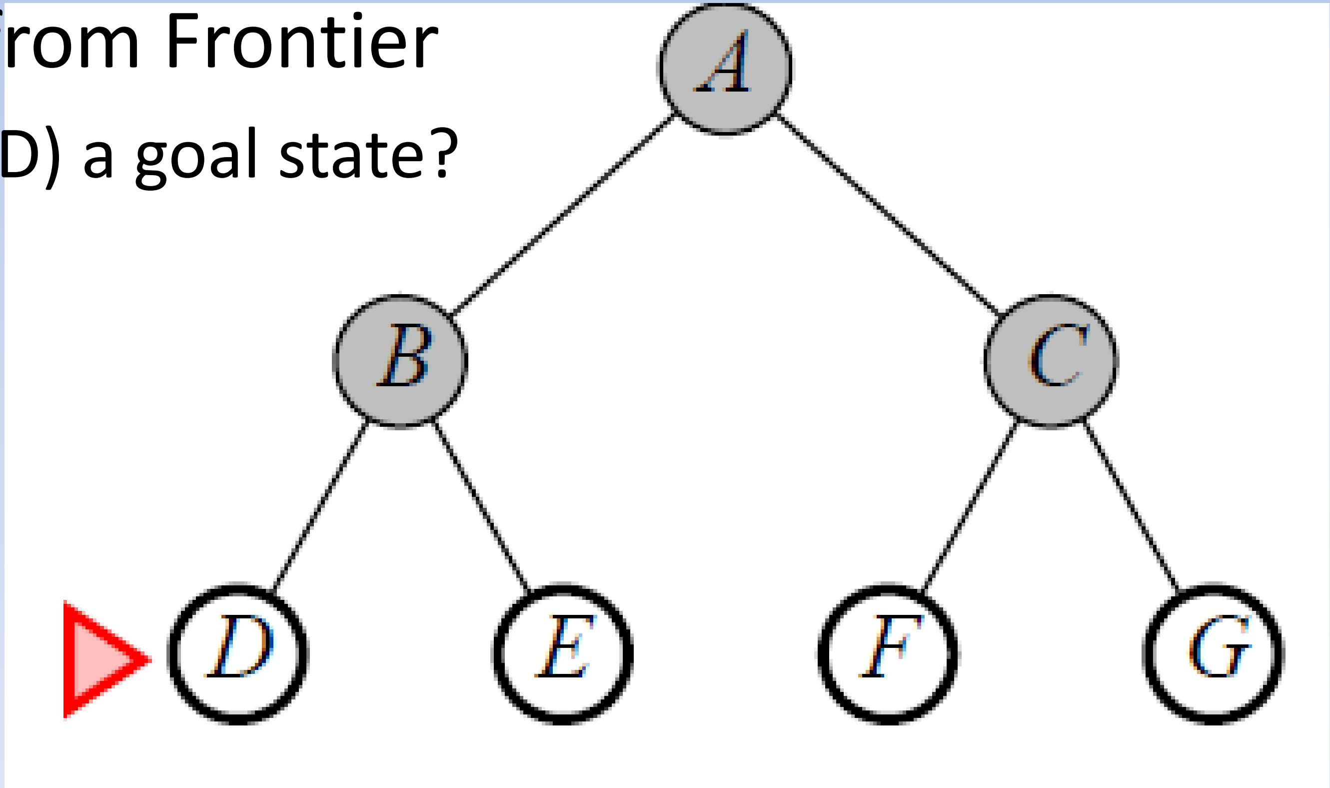
Breadth-First Search

- Expand (B):
  - Frontier = [C, D, E]
- Pop from Frontier
  - Is (C) a goal state?



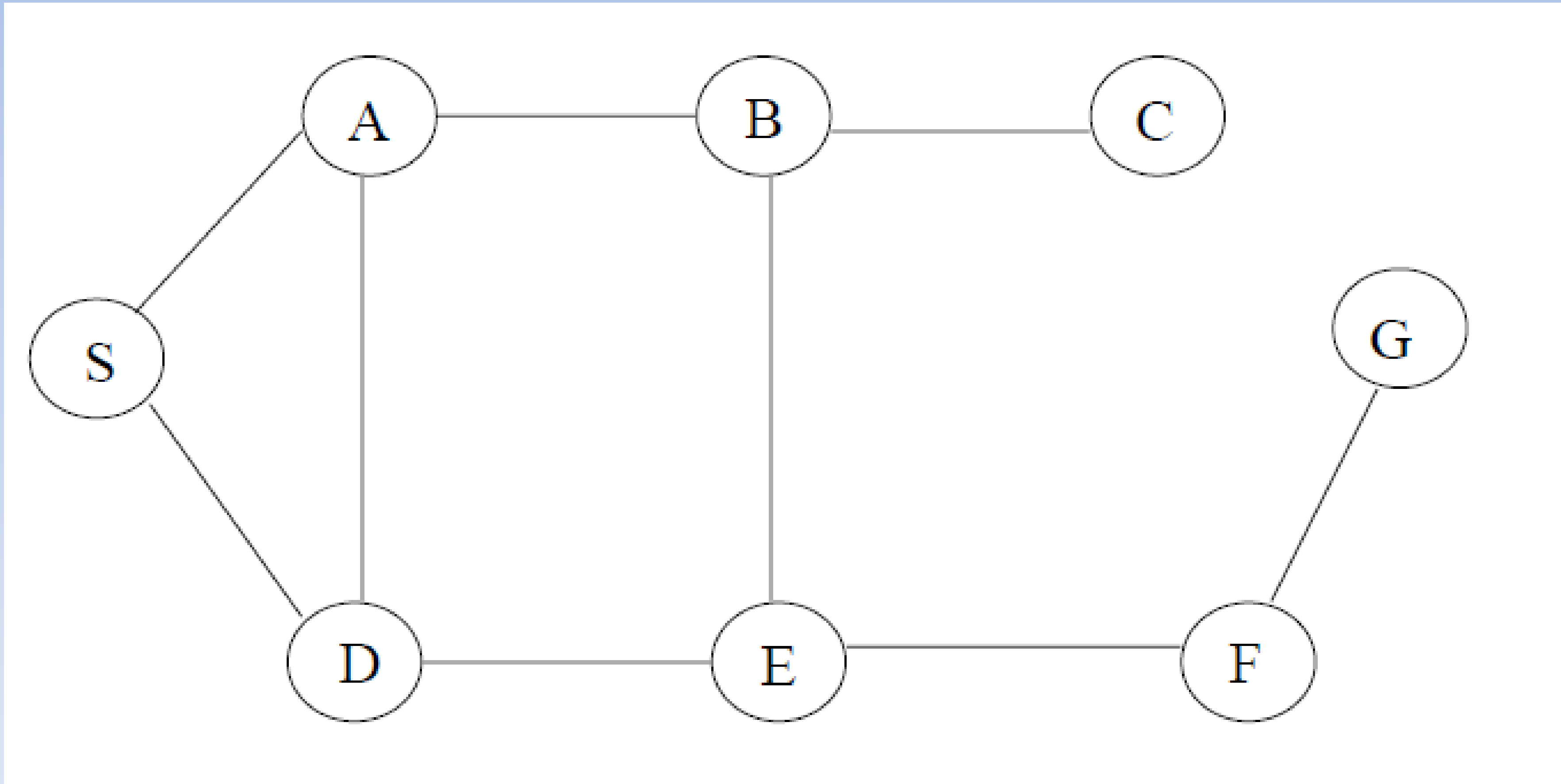
Breadth-First Search

- Expand (C):
  - Frontier = [D, E, F, G]
- Pop from Frontier
  - Is (D) a goal state?



Example: Map Navigation

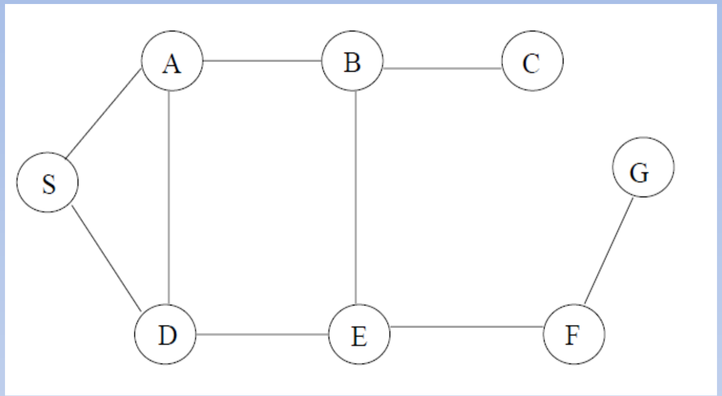
- S=start state, G=goal, links=legal transitions

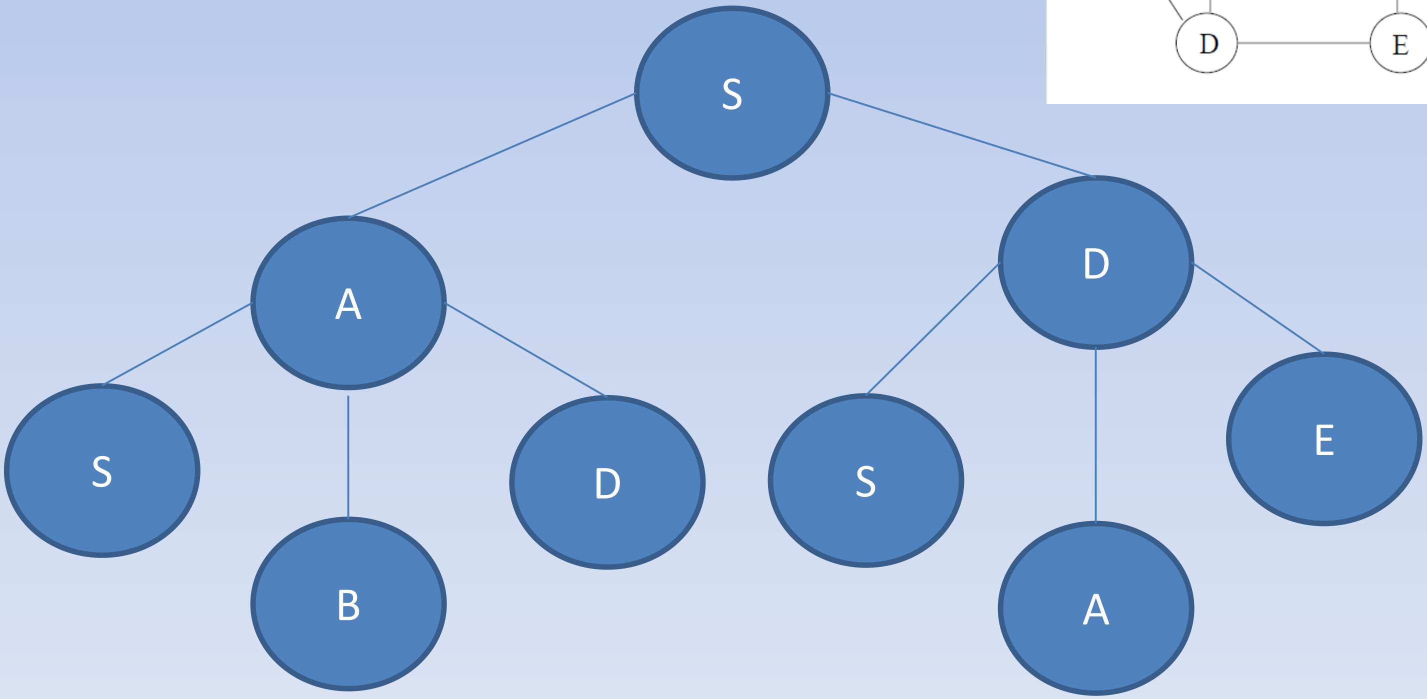


Tree-Search vs. Graph-Search

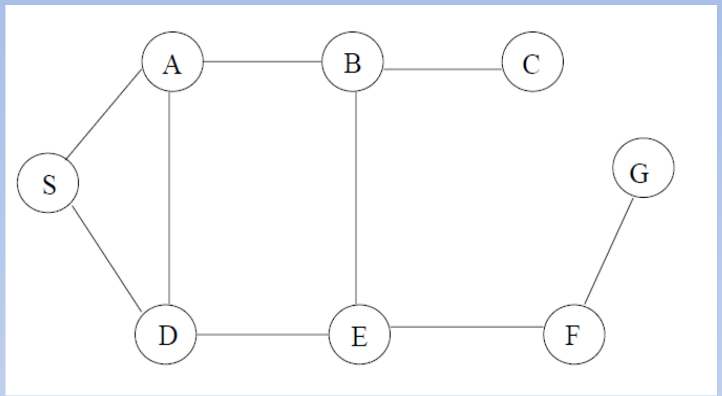
- **Search-tree(problem),** returns a solution or failure
- Frontier <- initial state
- Loop do
  - If frontier is empty return failure
  - Choose a leaf node and remove from frontier
  - If node is goal, return the corresponding solution
  - Expand the chosen node, adding its children to the frontier
- **Graph-search(Problem),** returns a solution or failure
- Frontier <- initial state, explore <- empty
- Loop do
  - If frontier is empty return failure
  - Choose a leaf node and remove from frontier
  - If node is goal, return the corresponding solution
  - Add the node to the explored
  - Expand the chosen node, adding its children to the frontier, Only if not in explored

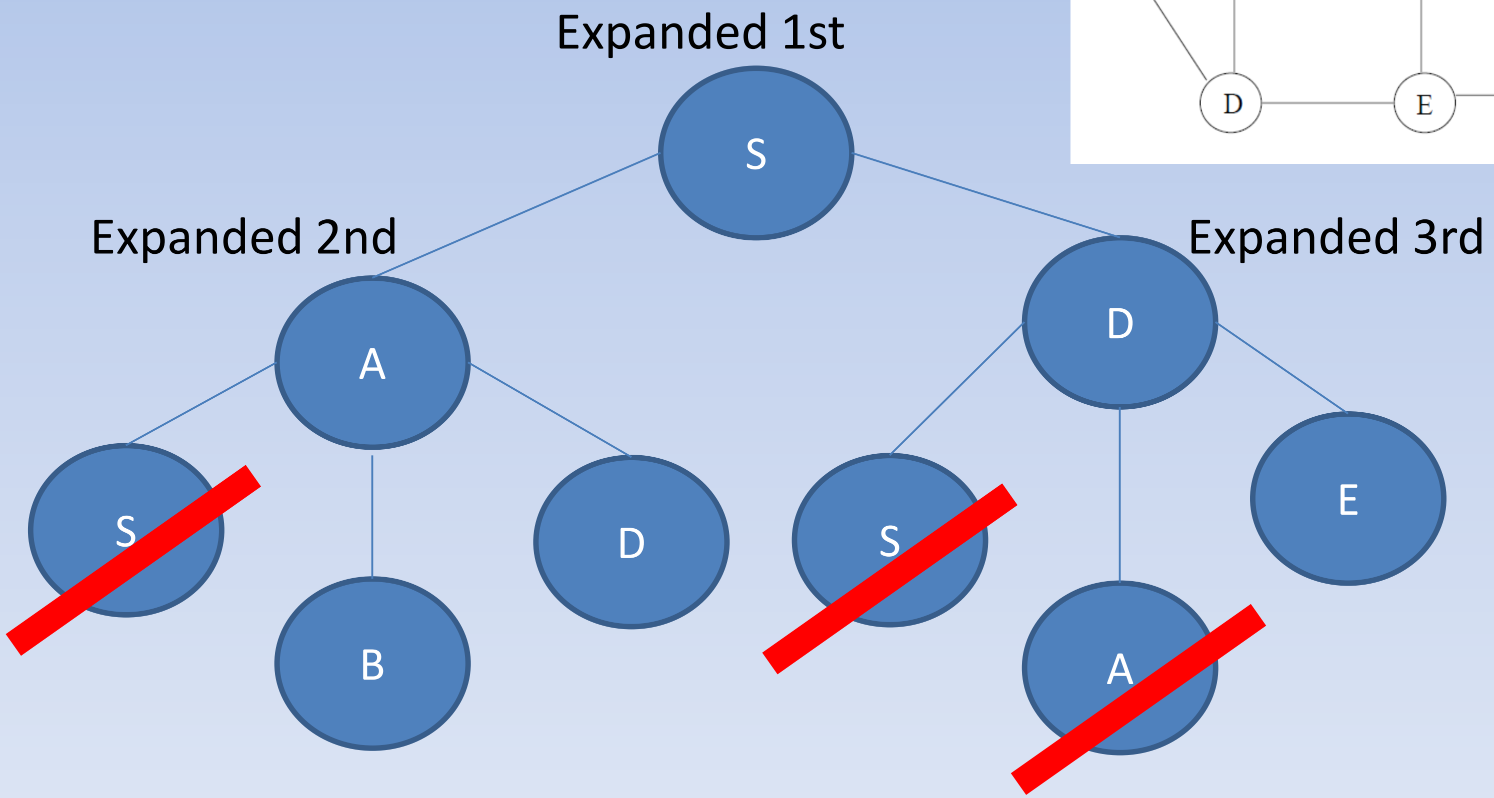
Initial BFS Search Tree





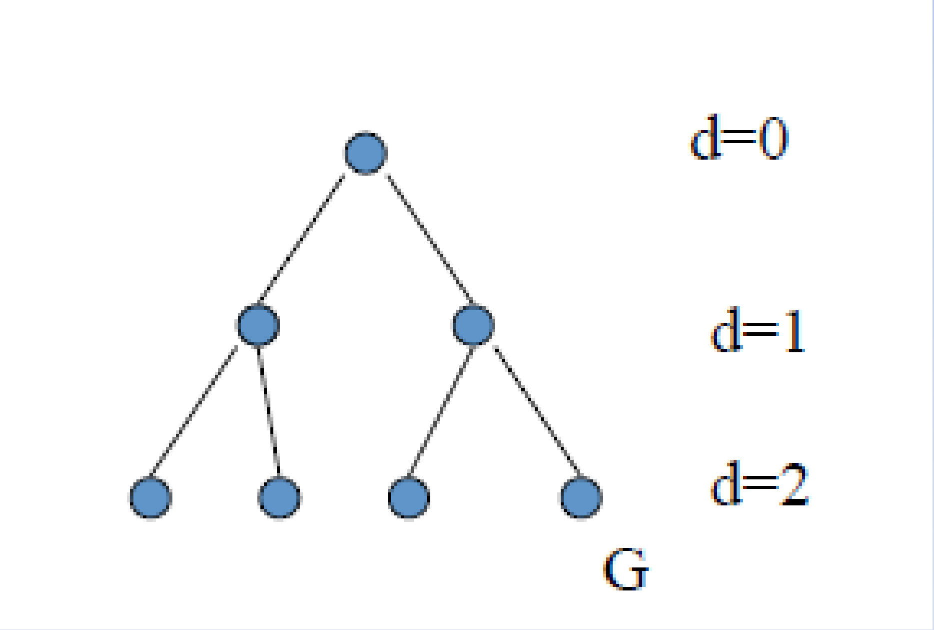
Initial BFS Graph Search





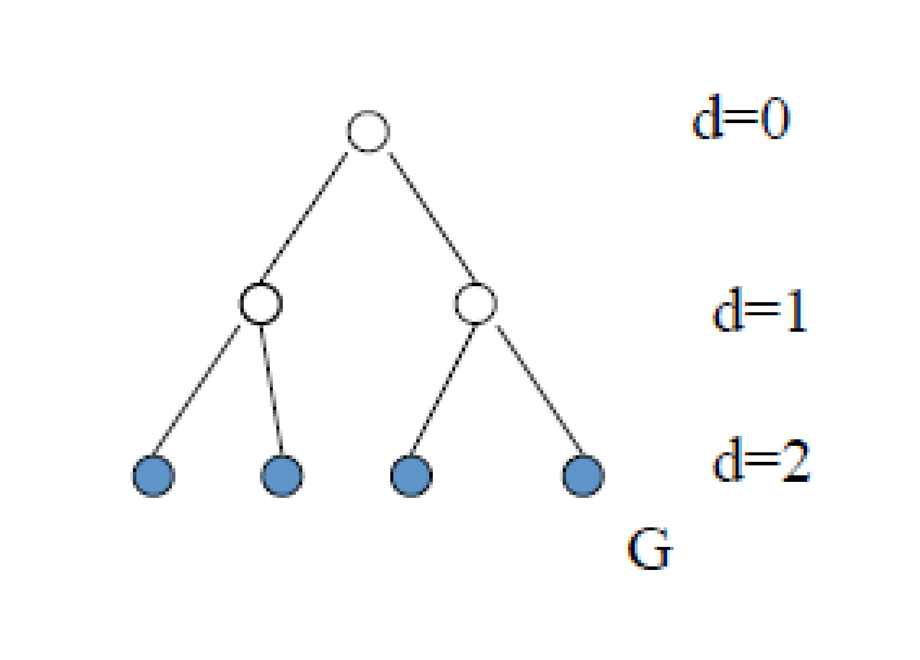
Properties of Breadth-First Search

- Complete: YES!
  - For finite b
- Time Complexity
  - Assume (worst case) that there is 1 goal leaf at RHS.
  - So BFS will expand all nodes
    - $1 + b + b^2 + \cdots + b^d$
    - $O(b^d)$



Properties of Breadth-First Search

- Space Complexity
  - How many nodes can be in the Frontier (worst case)?
  - At depth d there are b^d unexpanded nodes in the Frontier $O(b^d)$
- Optimal: YES
  - ONLY IF cost = 1 per step
  - Not optimal in general!

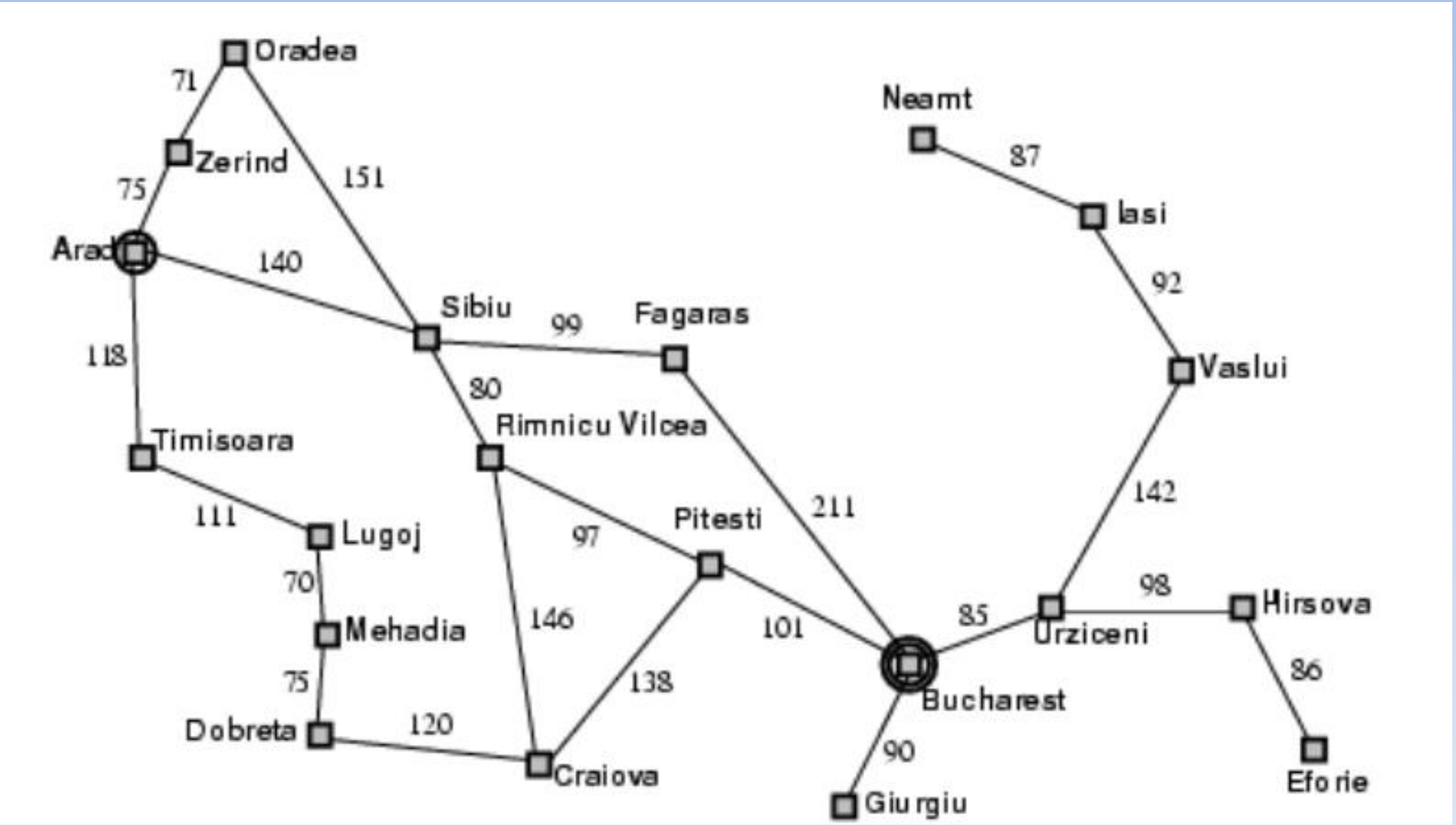


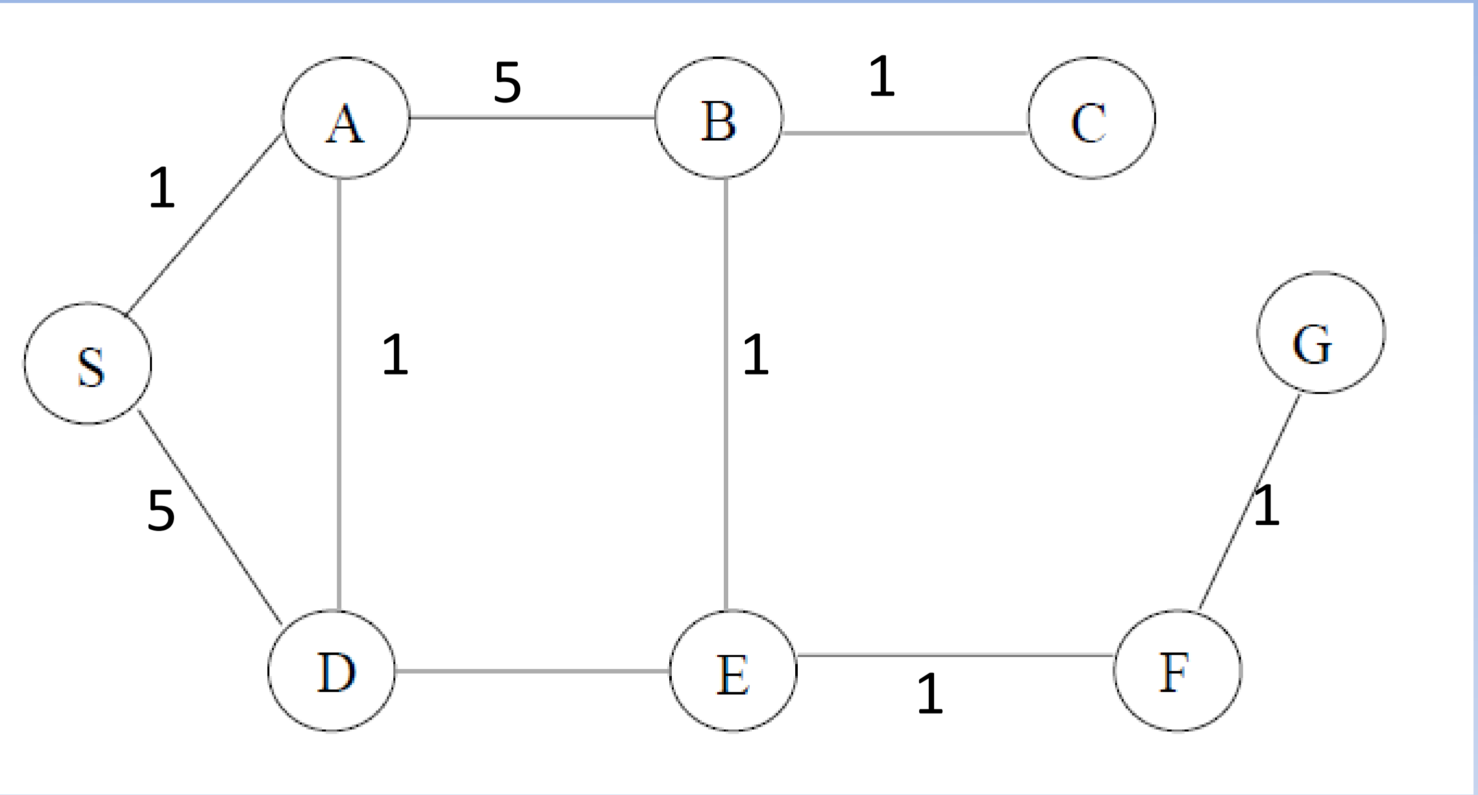
Example of Time and Memory Requirements for Breadth-First Search

Depth of Solution	Nodes ExpandedTime	Memory	
0	1	1 millisecond	100 bytes
2	111	0.1 seconds	11 kbytes
4	11,111	11 seconds	1 megabyte
8	10^8	31 hours	11 giabytes
12	10^{12}	35 years	111 terabytes

- Assuming b=10, 1000 nodes/sec, 100 bytes/node

Remember Romania!
Actions have Costs

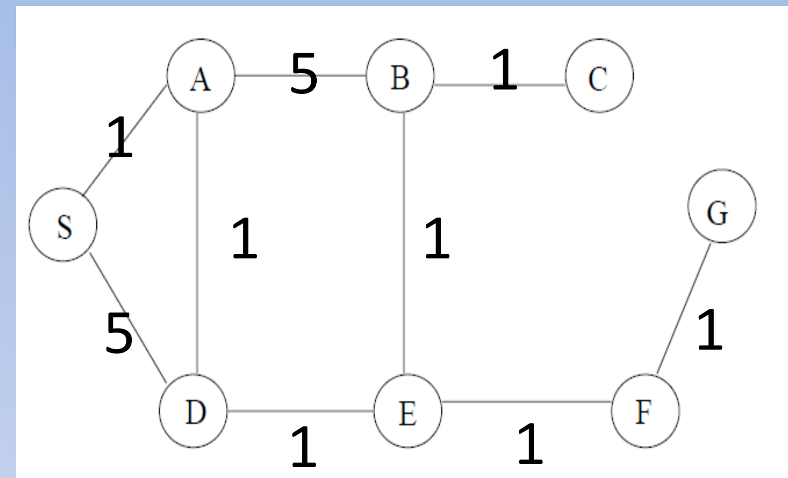


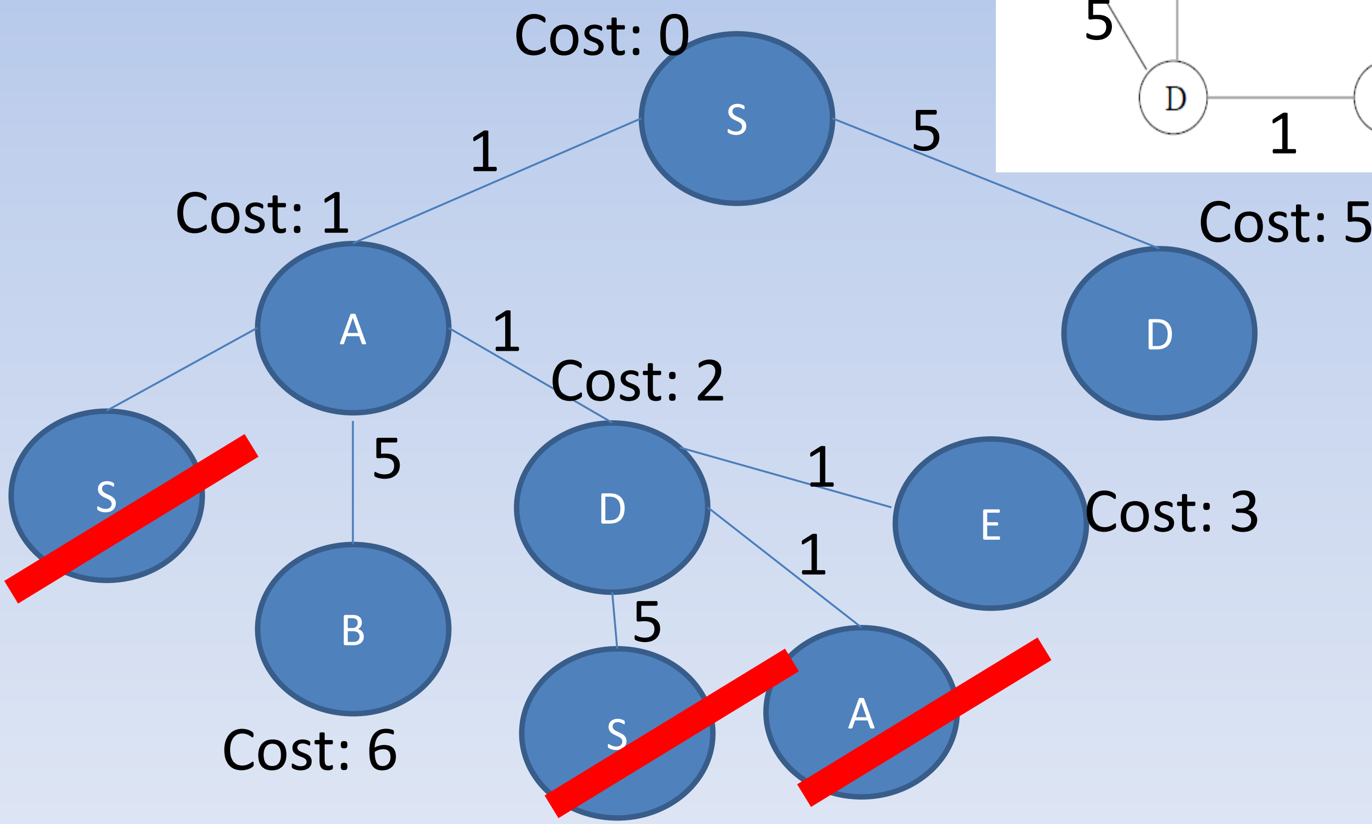

A
B
C
S
G
D
E
F
5
1
1
1
1
5
1
1

Uniform Cost Search

- Uniform Cost Search introduces a Cost function g(n) on each node.
- g(n) is the cost to reach node ‘n’ from the initial state.
- g(n) = g(n.parent) + cost(n.action, n.parent)
- BFS removed Nodes from Frontier FIFO (First-In-First-Out
- UCS removes Nodes from Frontier Lowest Cost First (Heap)

Initial UCS Search Tree





Frontier: [('S', 0)]

S, 0: Is it a goal?

Node 1 Expanding: S => ['A', 'D']

Frontier: [('A', 1), ('D', 5)]

A,1: Is it a goal?

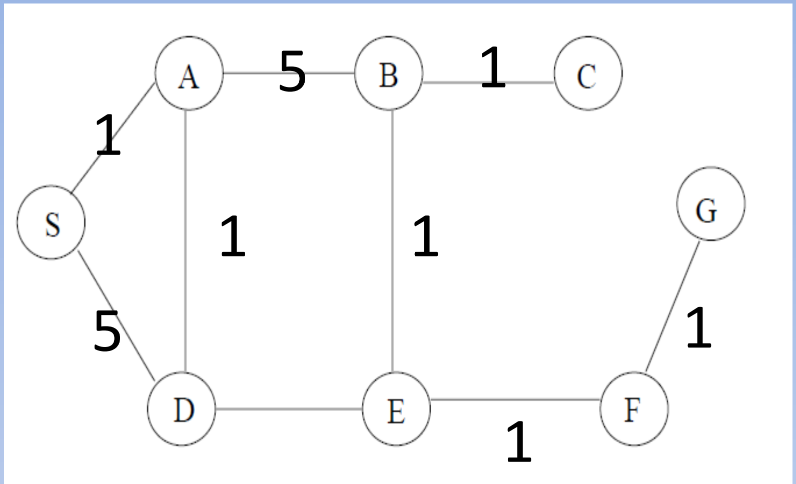
Node 2 Expanding: A => ['B', 'D', 'S']

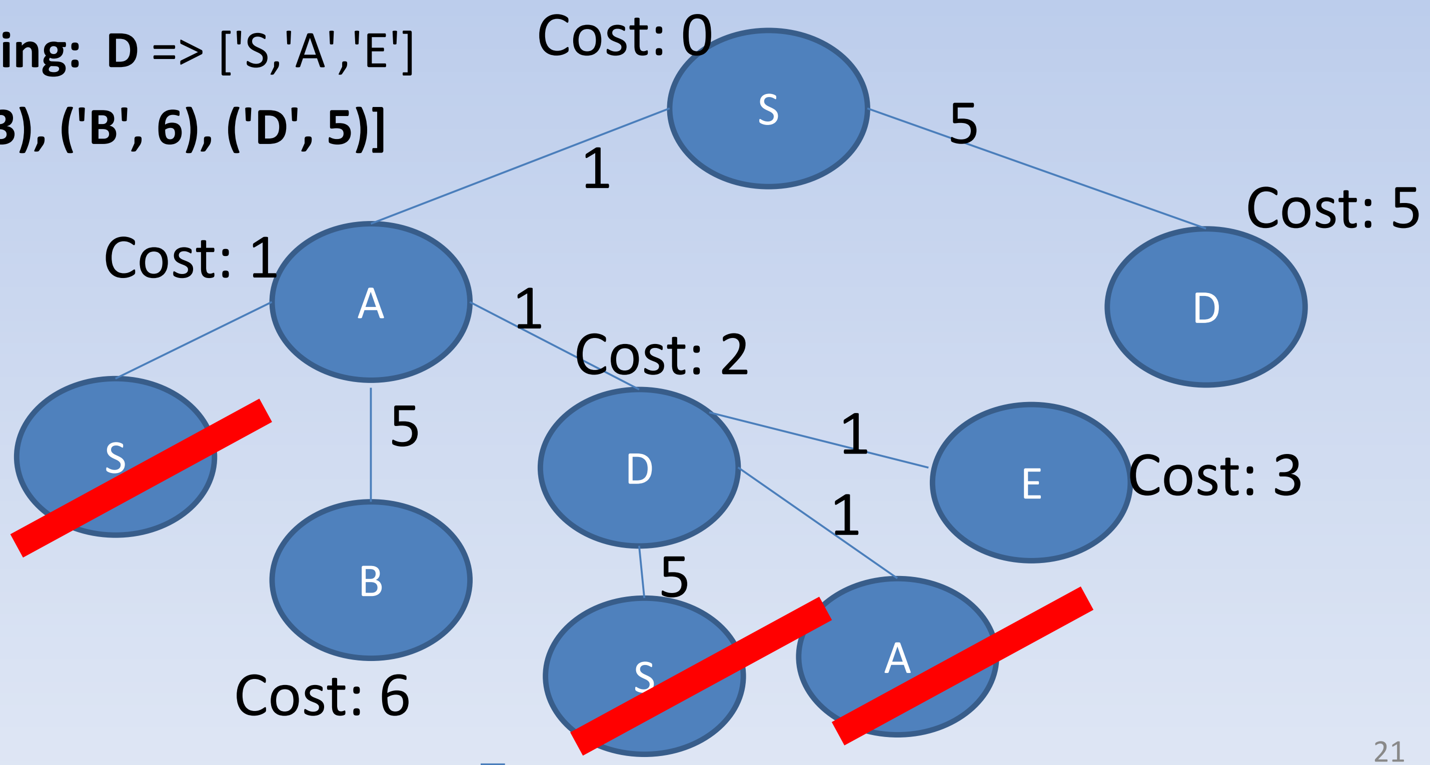
Frontier: [('D', 2), ('B', 6), ('D', 5)]

D,2: Is it a goal?

Node 3: Expanding: D => ['S,'A','E']

Frontier = [('E', 3), ('B', 6), ('D', 5)]





Frontier: [('E', 3), ('B', 6), ('D', 5)]

E,3: Is it a goal?

Expanding: E => ['B', 'D', 'F']

Frontier: [('B', 4), ('F', 4), ('D', 5), ('B', 6)]

B,4: Is it a goal?

Expanding: B => ['A', 'C', 'E']

Frontier: [('F', 4), ('C', 5), ('D', 5), ('B', 6)]

F,4: Is it a goal?

Expanding: F => ['E', 'G']

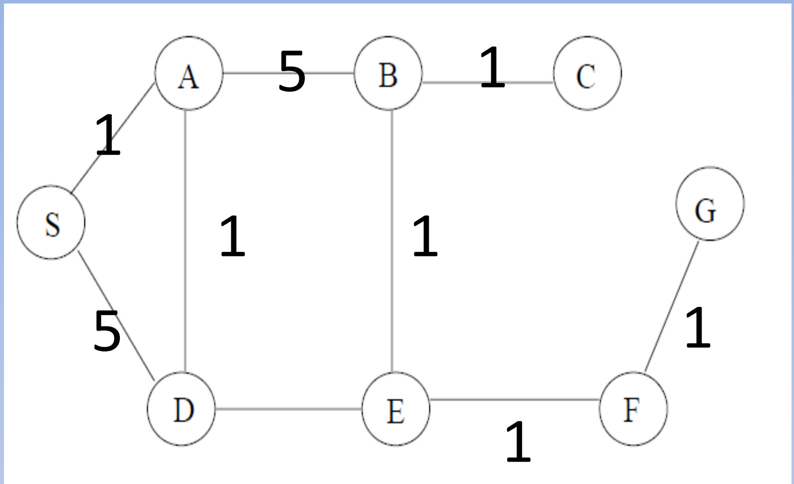
Frontier: [('G', 5), ('D', 5), ('B', 6), ('C', 5)]

G,5: Is it a goal?

Solution = ['A', 'D', 'E', 'F', 'G']

States Expanded = ['S', 'A', 'D', 'E', 'B', 'F']

Solution Cost = 5



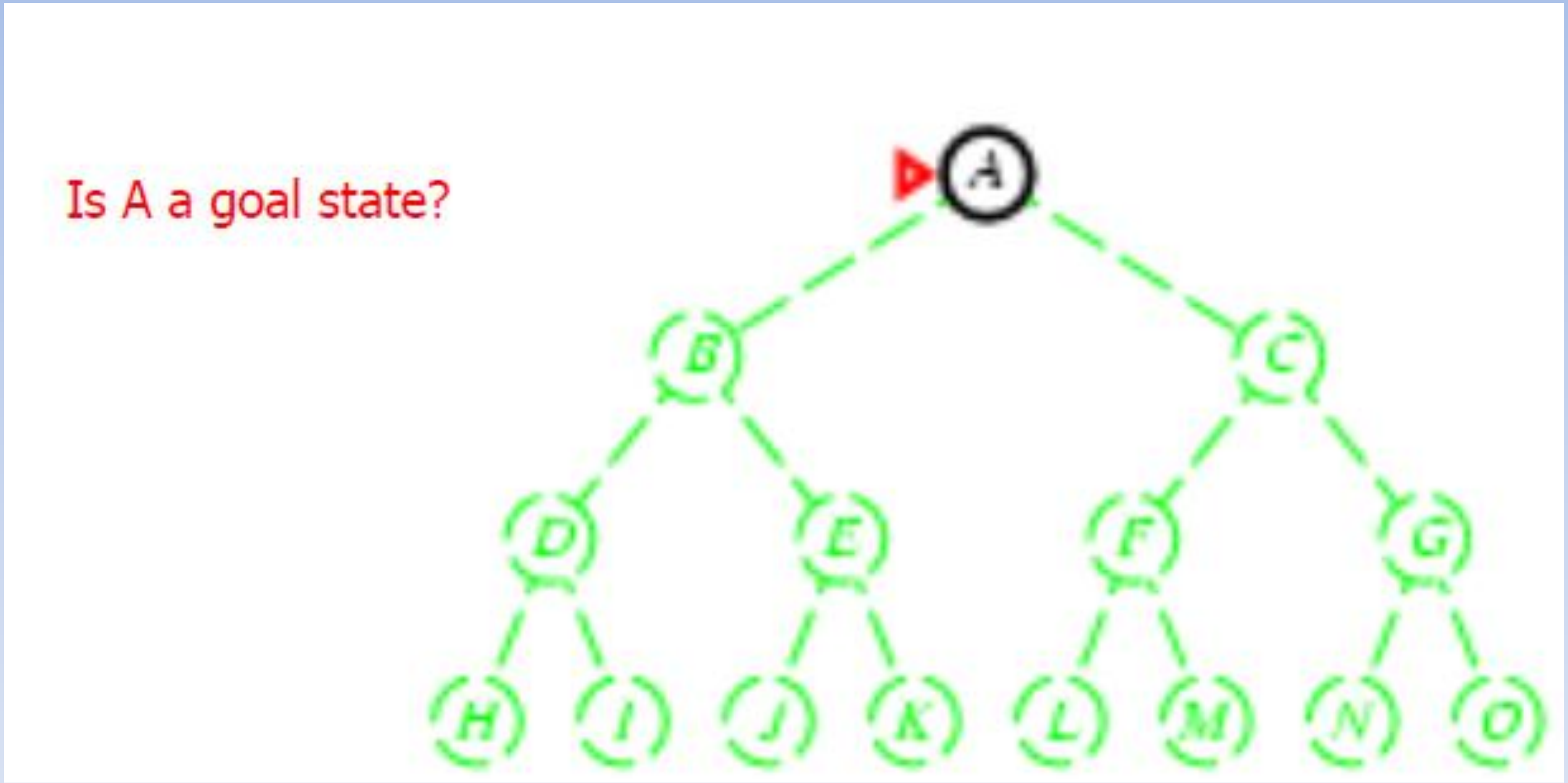
Properties of UCS

- Implementation
  - Frontier (Fringe) = queue ordered by path cost, lowest first.
  - Equivalent to Breadth-First if step costs all equal.
- Complete?: YES, as long as step cost >= ε
- Time Complexity: Expands the number of nodes where g <= cost of optimal solution C^*
  - $O(b^{[C^*/\varepsilon]})$
- Space Complexity: Also the number of nodes where g<= cost of optimal solution $O(b^{[C^*/\varepsilon]})$
- Optimal: Yes, nodes expanded in increasing order of g(n)

Depth-First Search

- Expand deepest unexpanded node
- Implementation:
  – Frontier (Fringe) = LIFO queue, i.e., put successors at front.

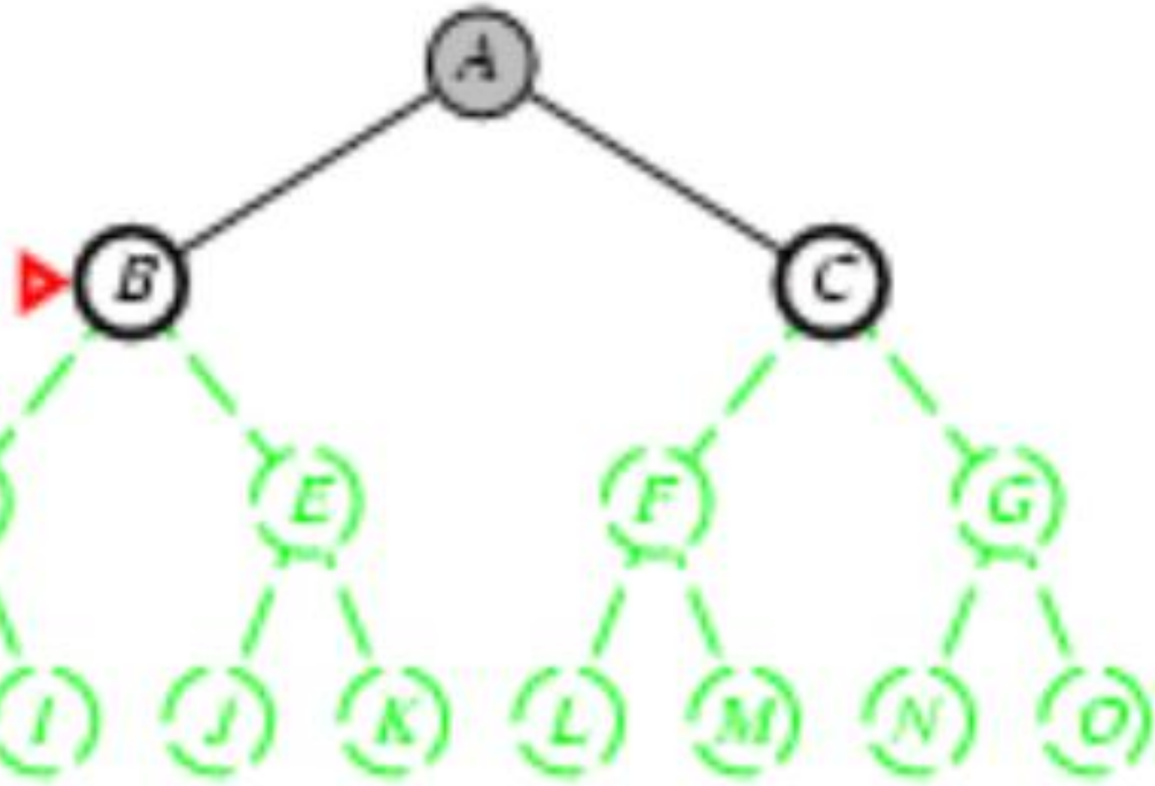
Depth-First Search



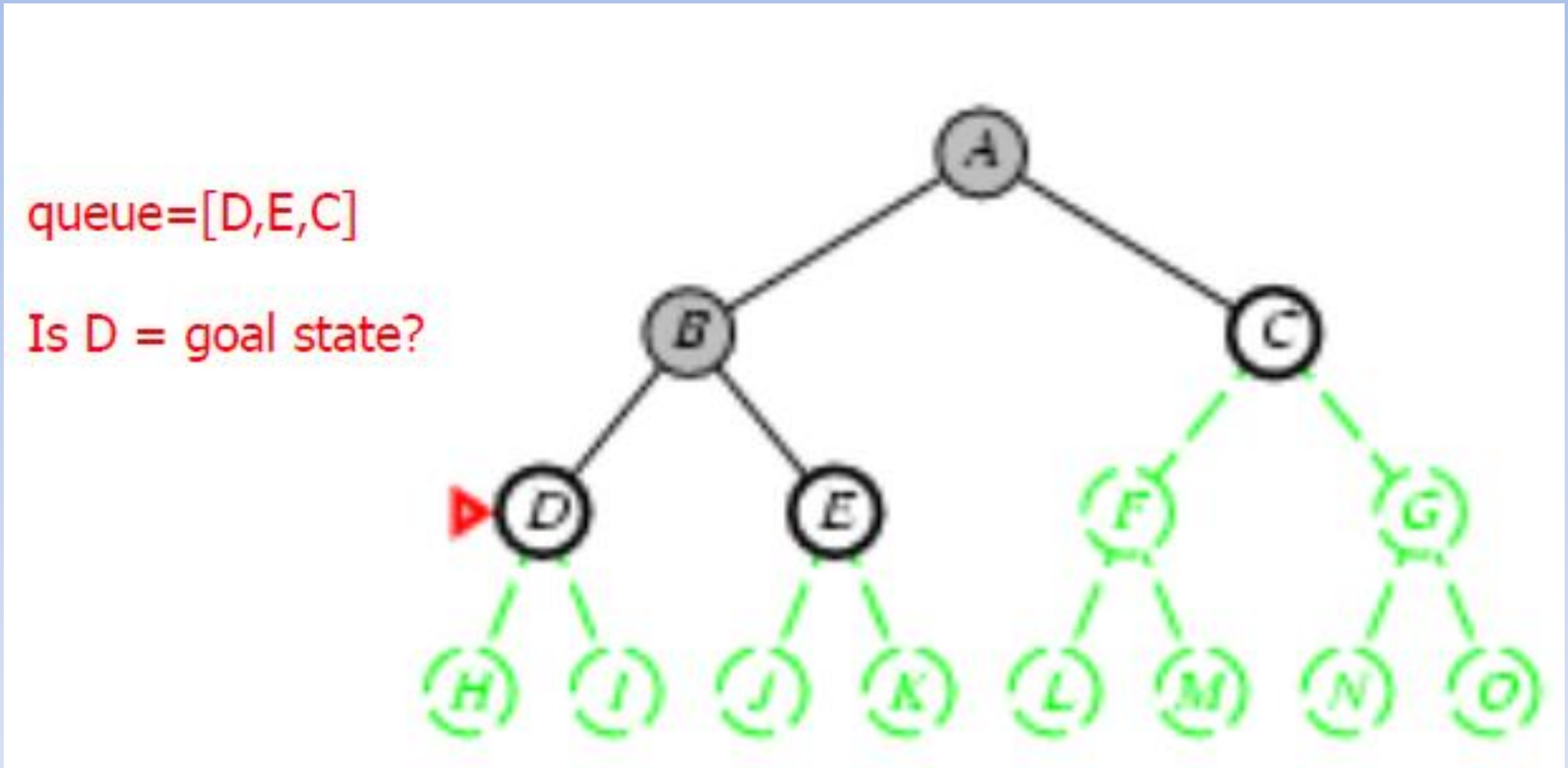
Depth-First Search

queue=[B,C]

Is B a goal state?



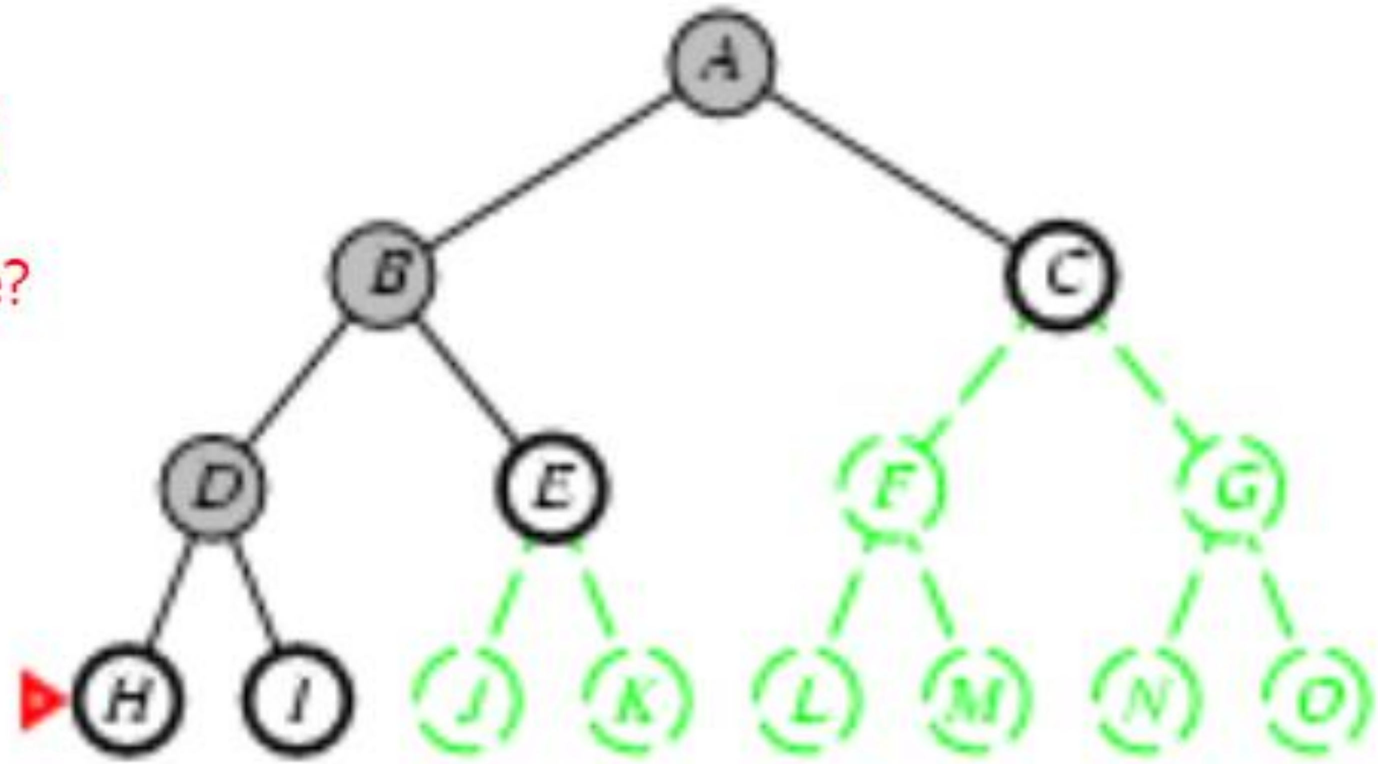
Depth-First Search



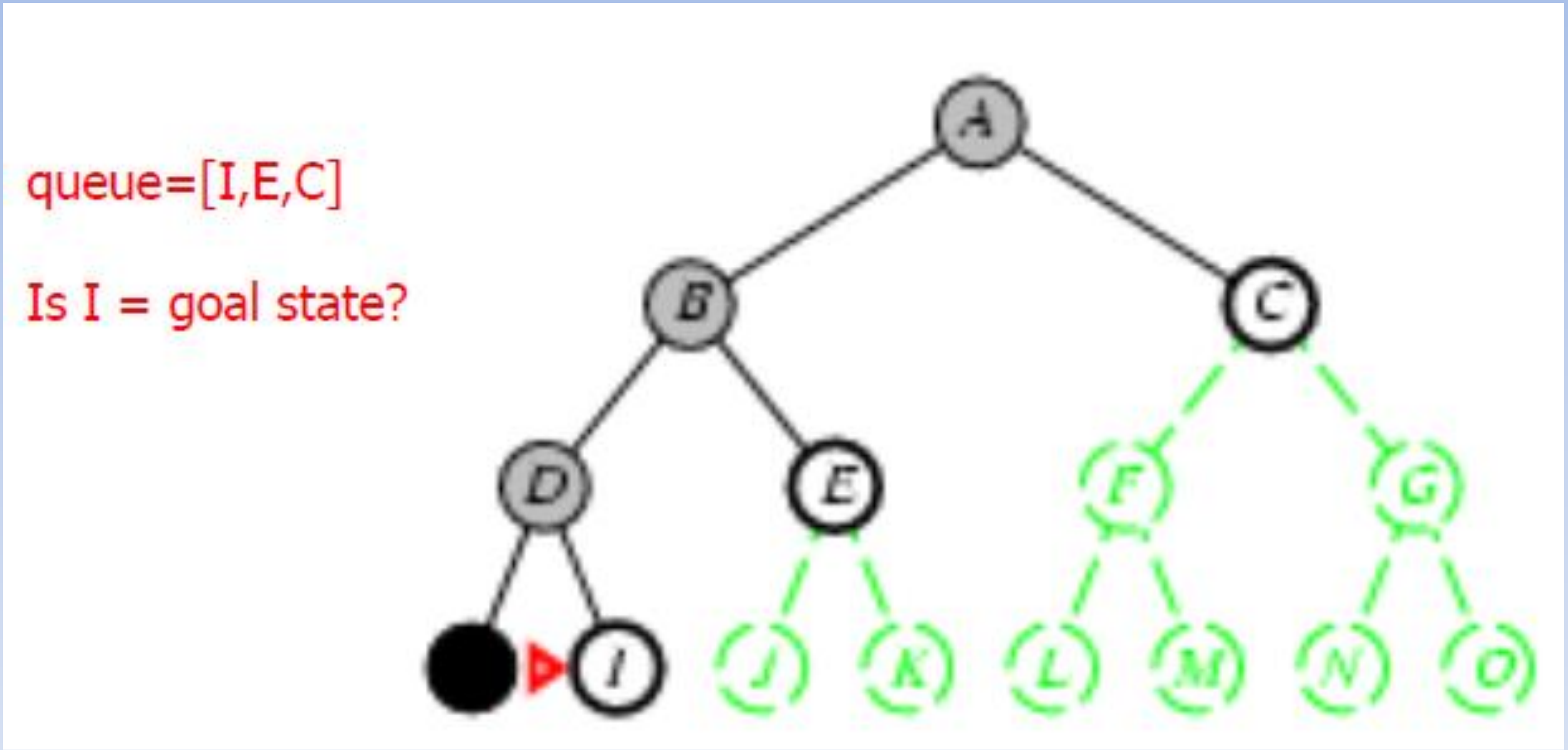
Depth-First Search

queue=[H,I,E,C]

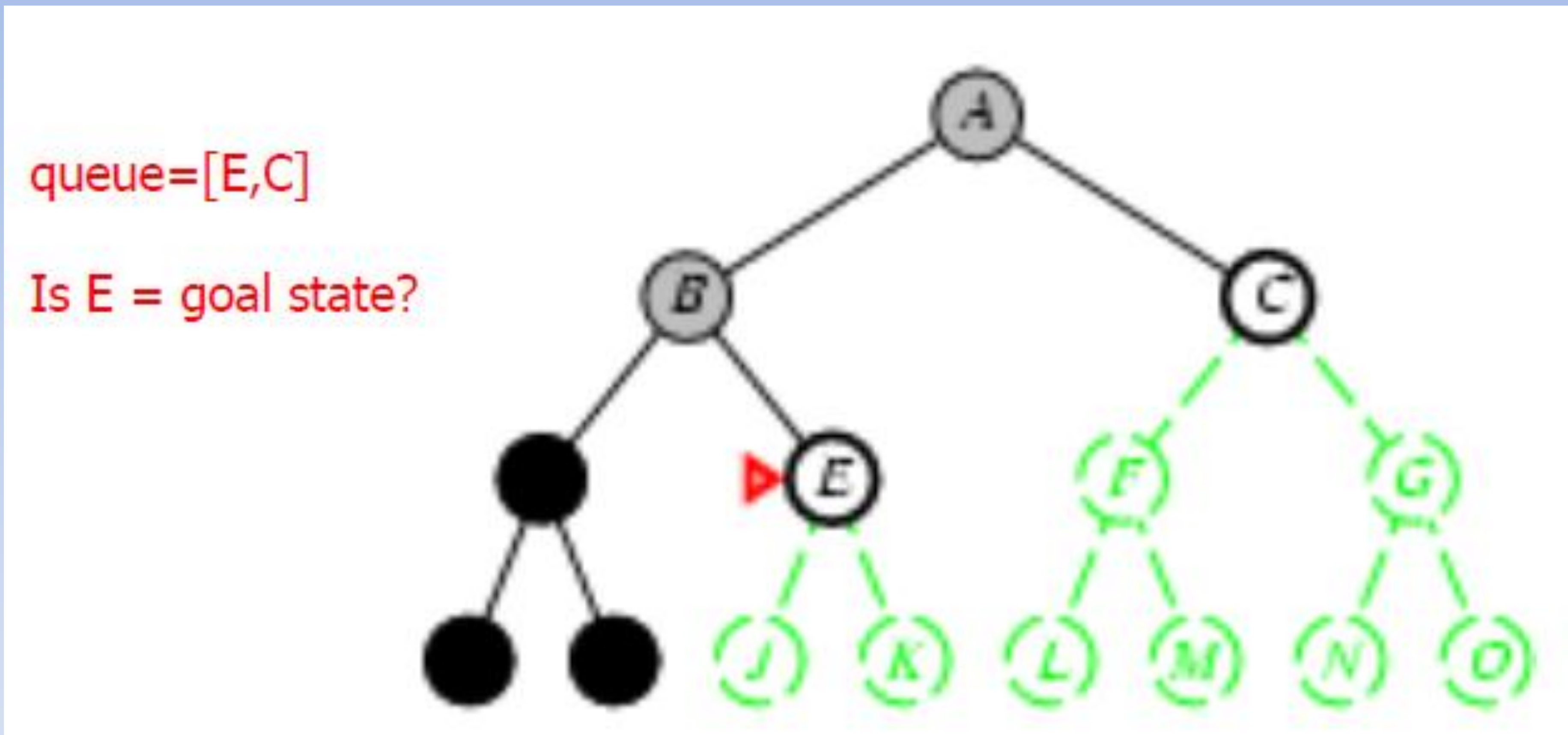
Is H = goal state?



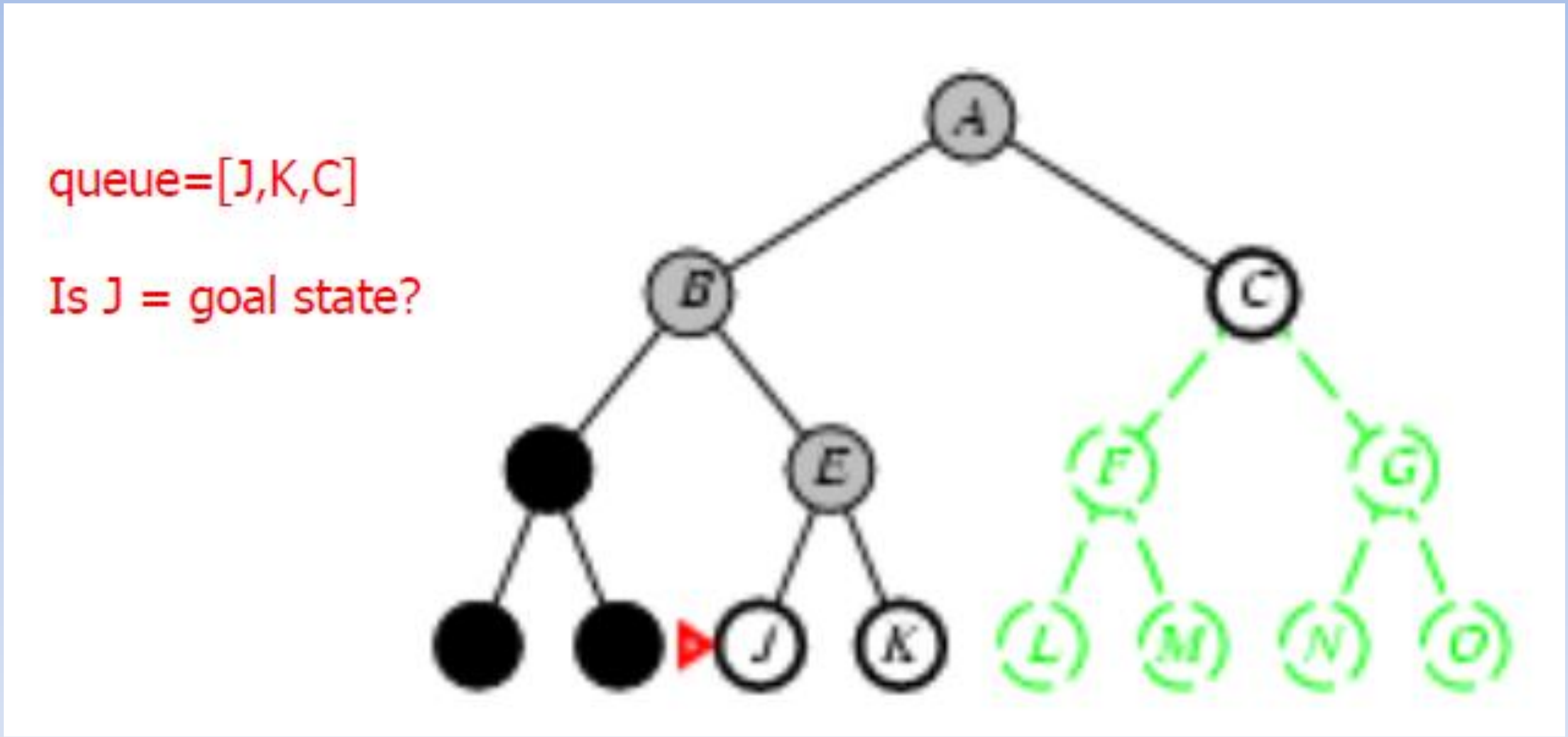
Depth-First Search



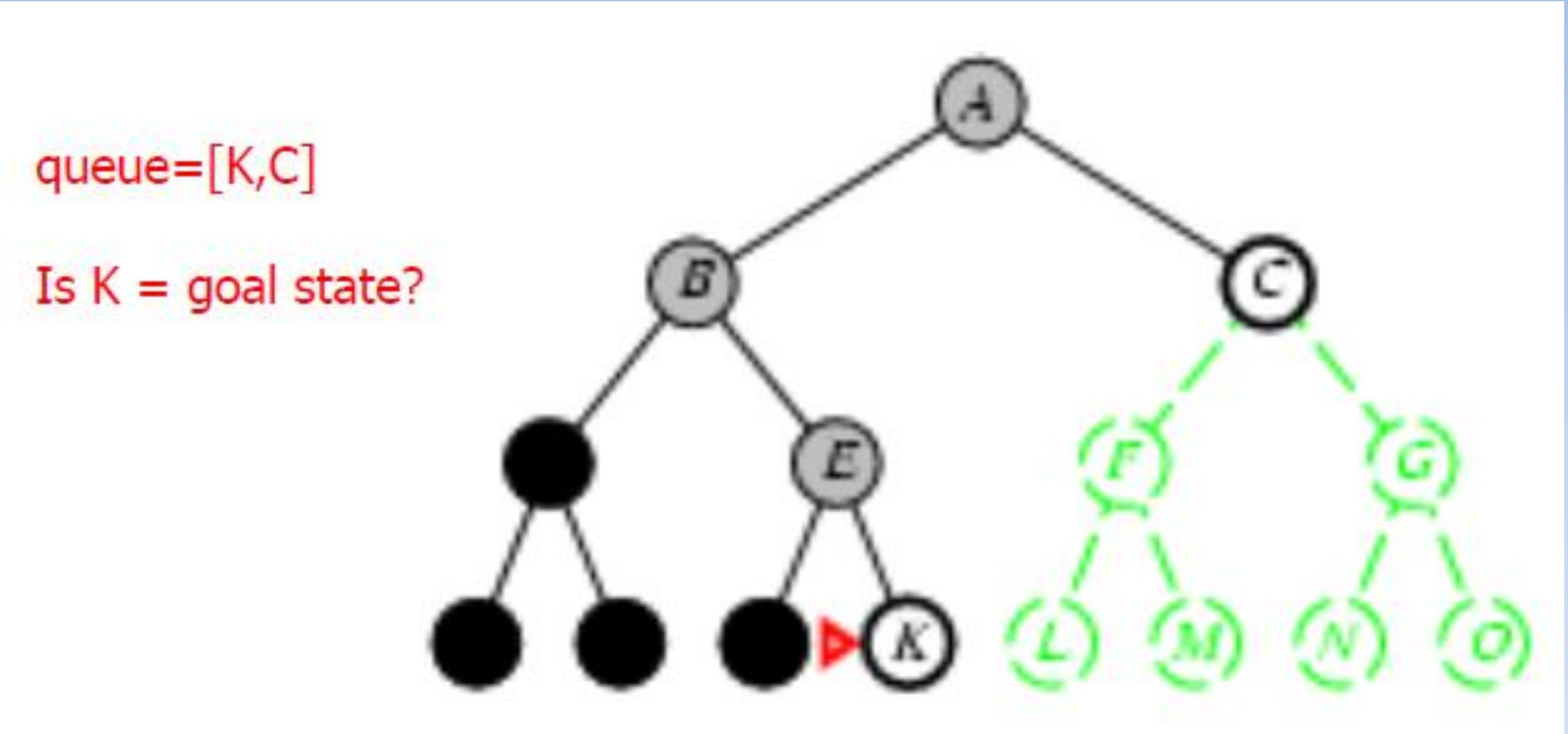
Depth-First Search



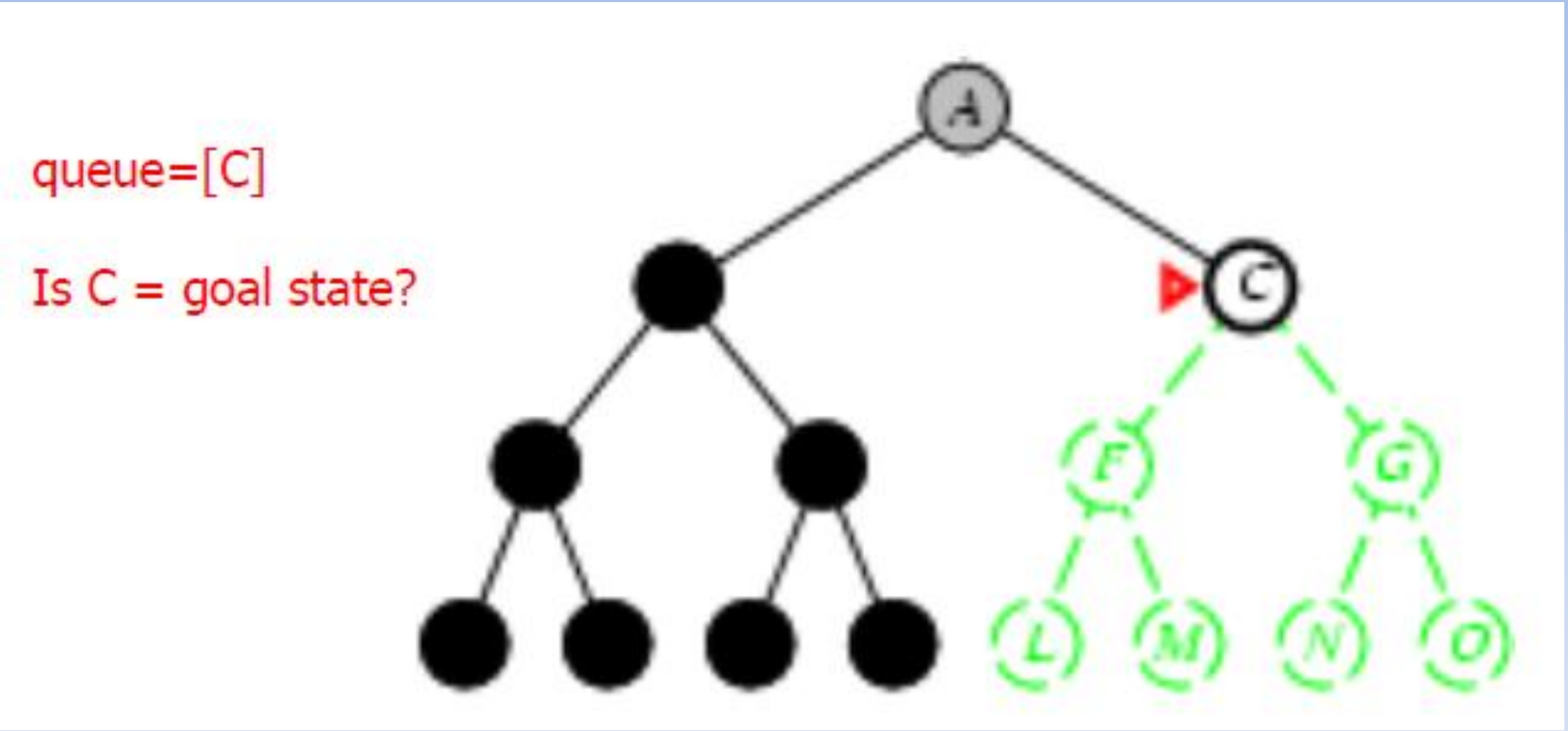
Depth-First Search



Depth-First Search



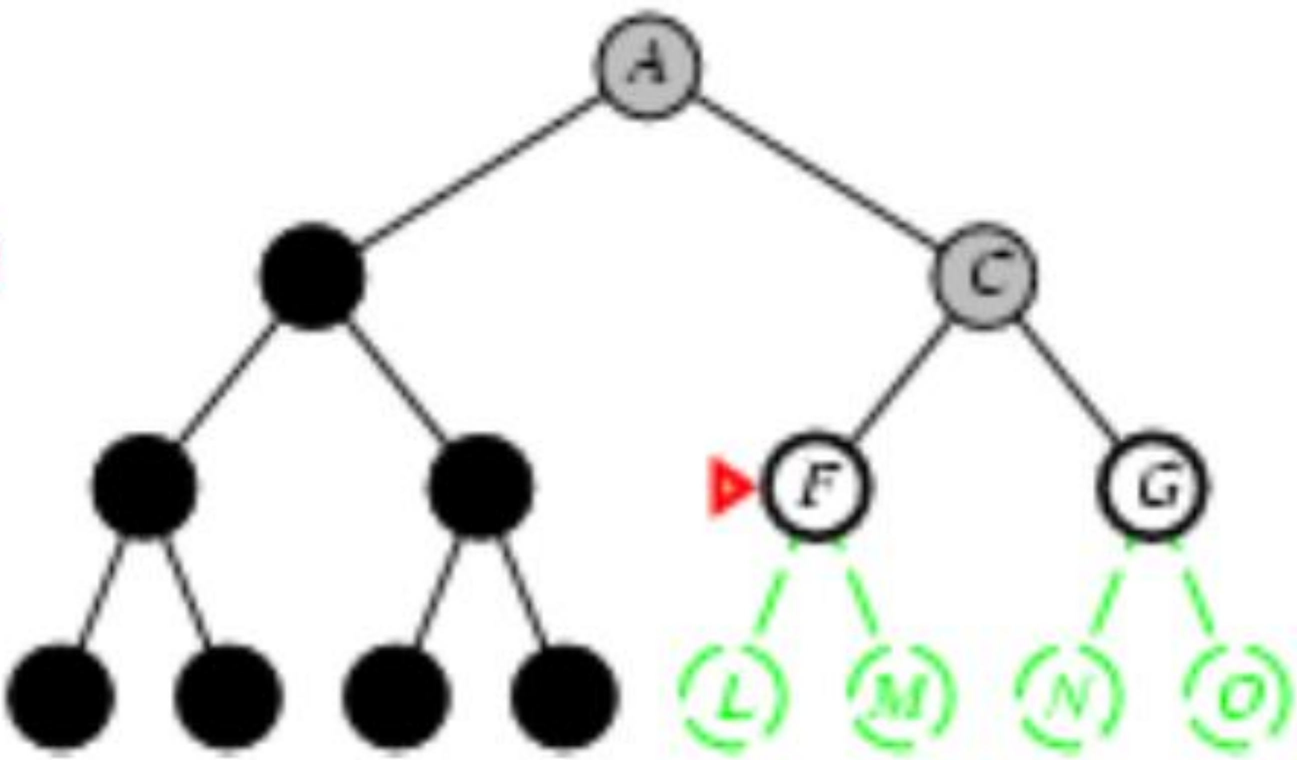
Depth-First Search



Depth-First Search

queue=[F,G]

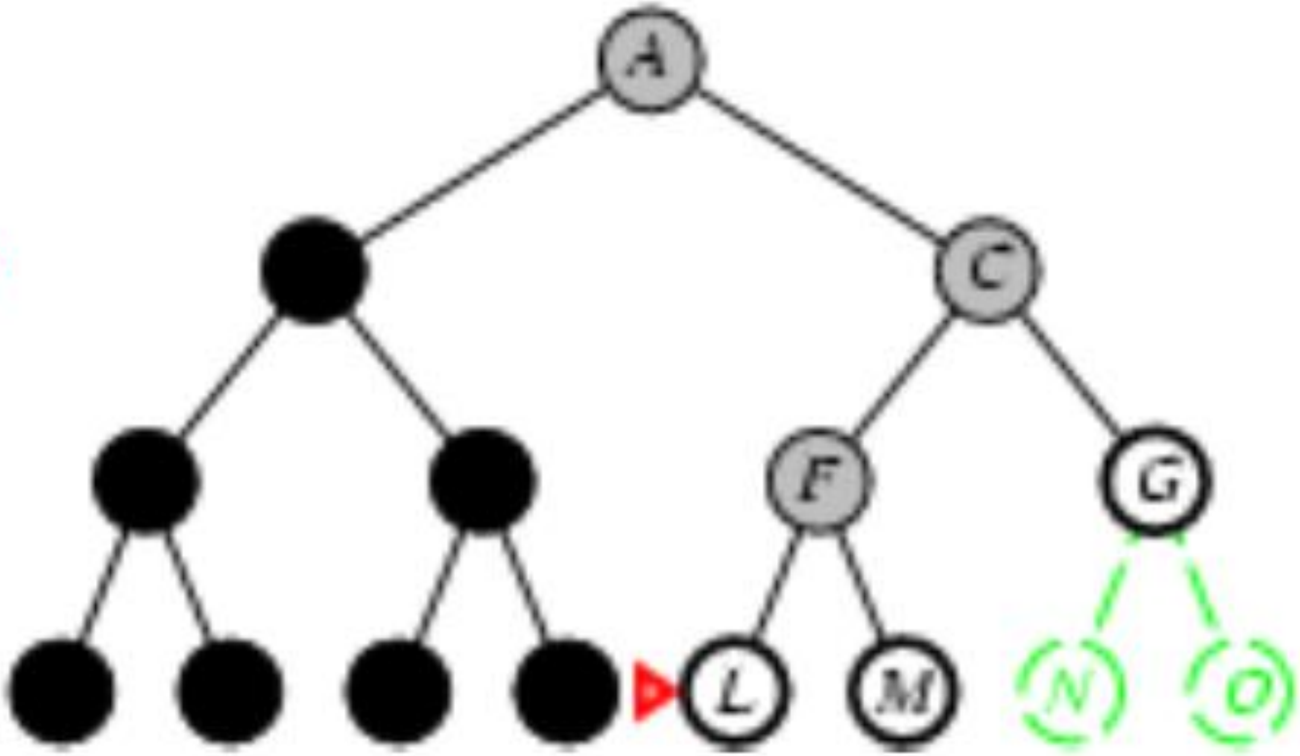
Is F = goal state?



Depth-First Search

queue=[L,M,G]

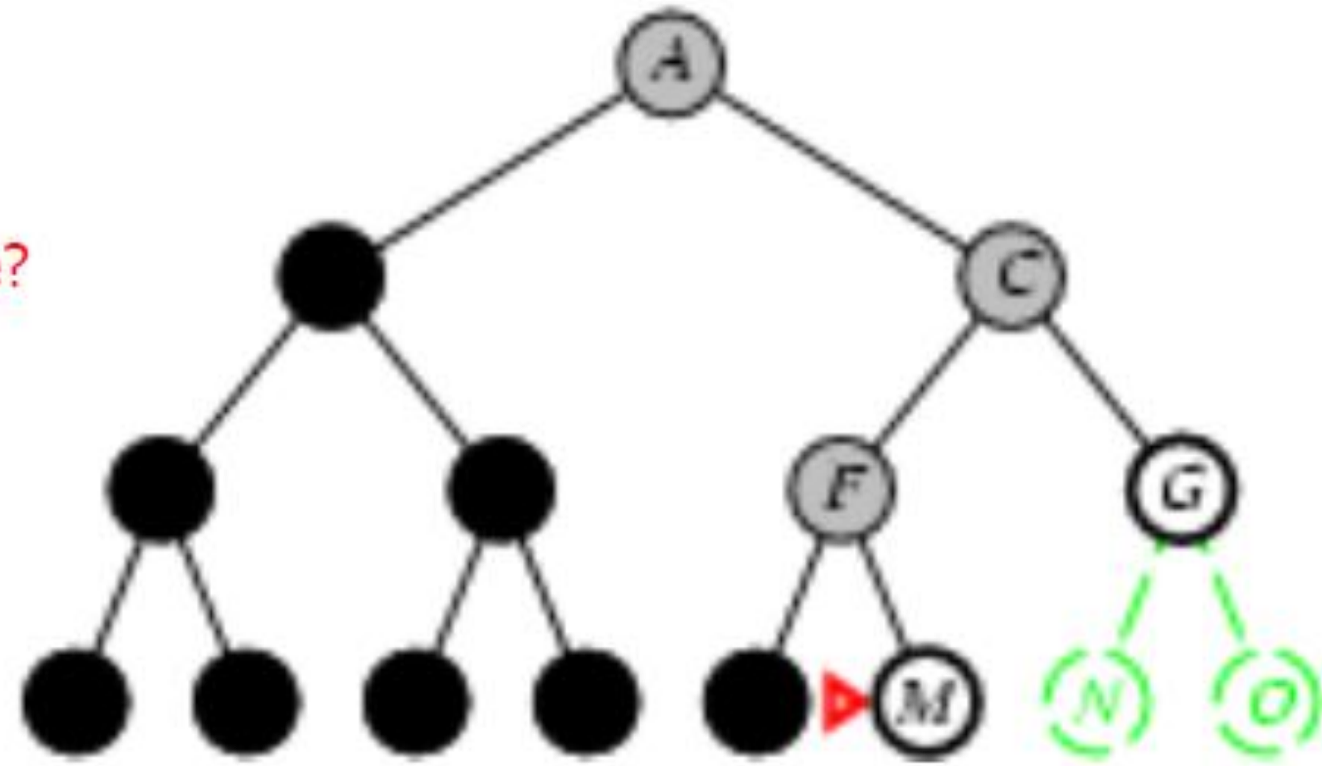
Is L = goal state?



Depth-First Search

queue=[M,G]

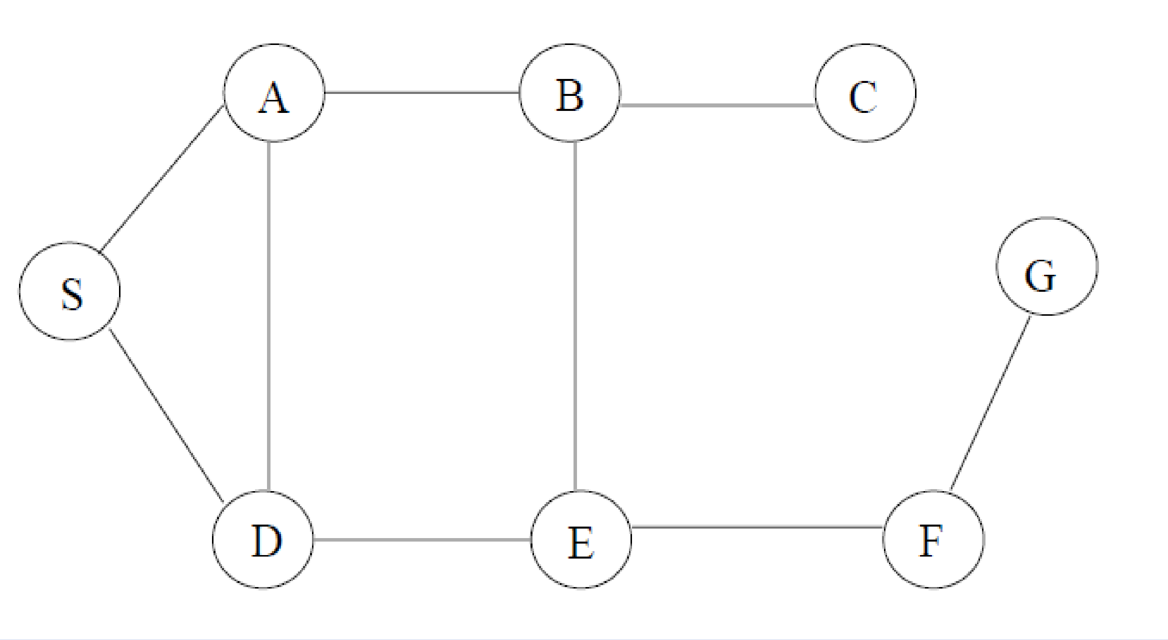
Is M = goal state?



Depth-First Search

Example:

Always apply actions in alphabetical order.

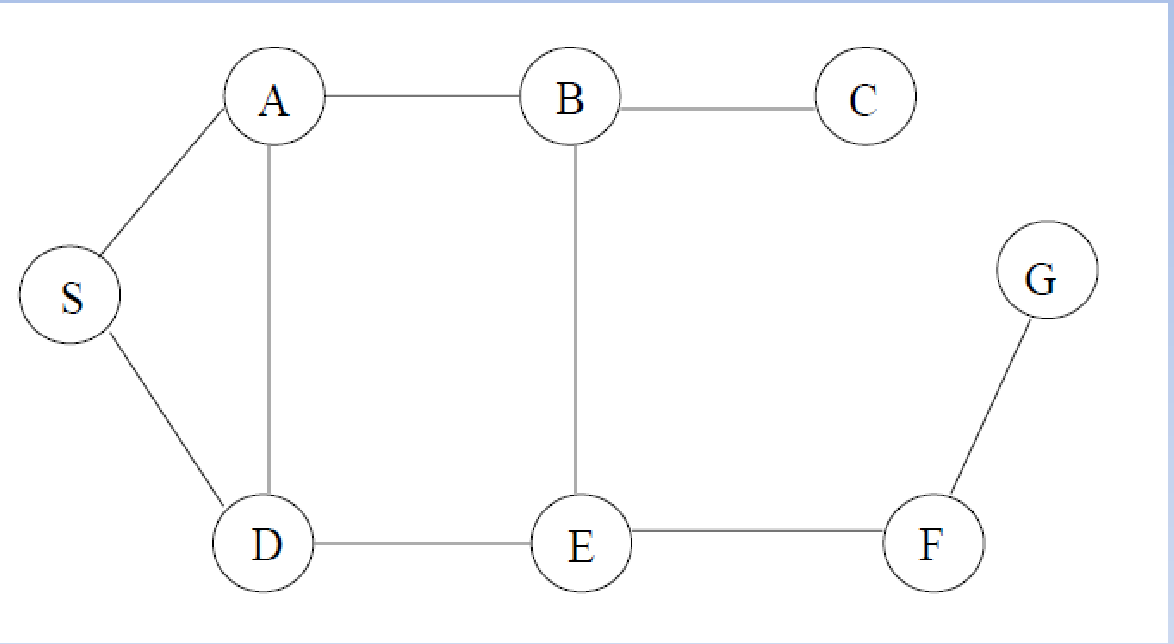


This Time: Depth-First Search Get's Lucky

Frontier: ['S']
S: Is it a goal?
Node 1 Expanding: S => ['A', 'D']

Frontier: ['A', 'D']
D: Is it a goal?
Node 2 Expanding: D => ['A', 'E', 'S']
Frontier: ['A', 'A', 'E']
E: Is it a goal?

Node 3 Expanding: E => ['B', 'D', 'F']
Frontier: ['A', 'A', 'B', 'F']
F: Is it a goal?



Node 4 Expanding: F => ['E', 'G']
Frontier: ['A', 'A', 'B', 'G']
G: Is it a goal? YES! Solution = ['D', 'E', 'F', 'G'] States Expanded = ['S', 'D', 'E', 'F']

Properties of Depth-First Search

- Complete: No. Fails in Infinite-depth spaces. Can fail in spaces with loops unless modified to avoid repeated states along path.
  - Complete in Finite Spaces.
- 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 ∞)
- Time??: $O(b^m)$ TERRIBLE if m is much larger than d. But if solutions are dense, may be much faster than breadth-first search.
- Space??: O(bm), i.i., linear space!
- Optimal??: No

Comparing DFS and BFS

- BFS is optimal, DFS is not
- Time Complexity worse-case is the same, but
  - In the worst-case BFS is always better than DFS
  - Sometimes, on the average DFS is better if:
    - Many Goals, No Loops, and No Infinite Paths
- BFS is much worse memory-wise
  - DFS can be linear space
  - BFS may store the whole search space.
- In General:
  - BFS is better If goal is not deep, If long paths, If many loops, If small search space.
  - DFS is better If many goals, Not many loops
  - DFS is much better in terms of memory.

Depth-Limited Search

- Depth-First Search with a depth limit ℓ,
  - Nodes at depth ℓ have no successors
- Recursive Implementation:

```
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 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
```

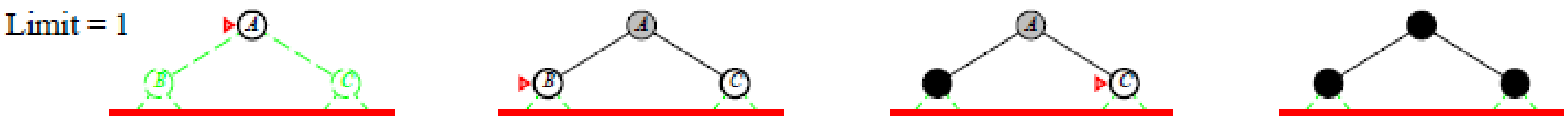
Iterative Deepening Search

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution
    inputs: *problem*, a problem

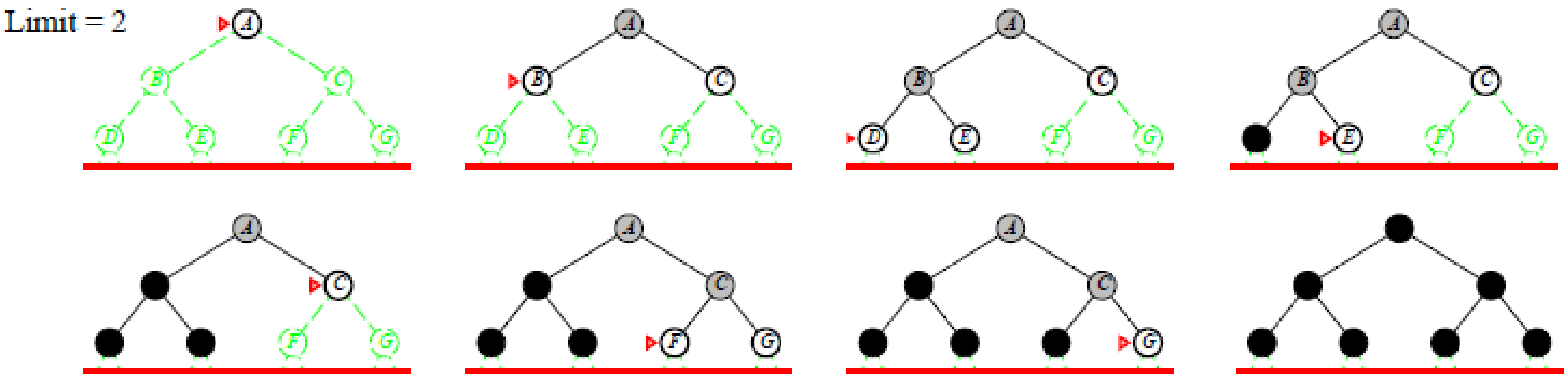
    for *depth* ← 0 **to** ∞ **do**
        result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
        if *result* $\neq$ cutoff **then return** *result*
    end

Iterative Deepening Search: $\ell = 0$

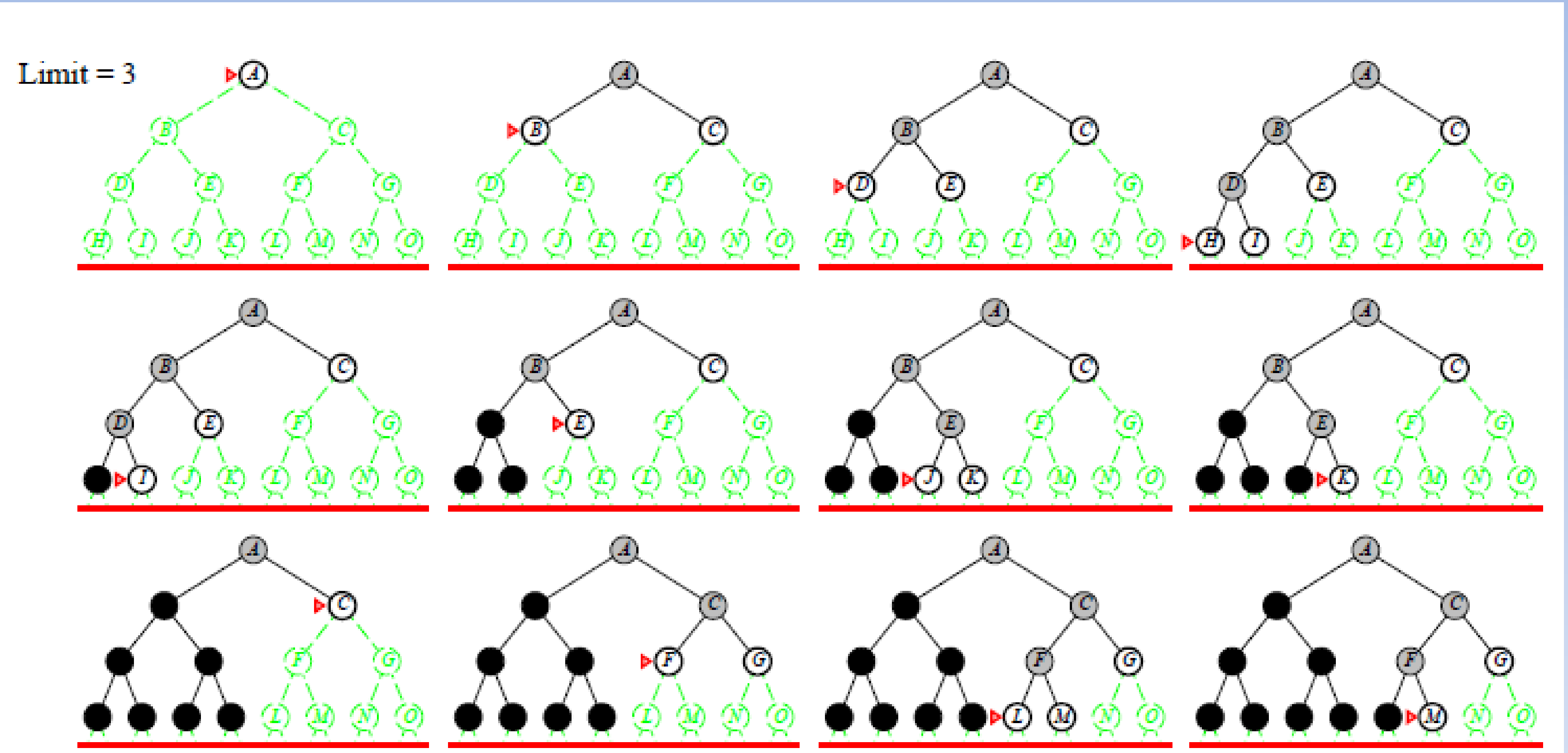
Iterative Deepening Search: $\ell = 1$



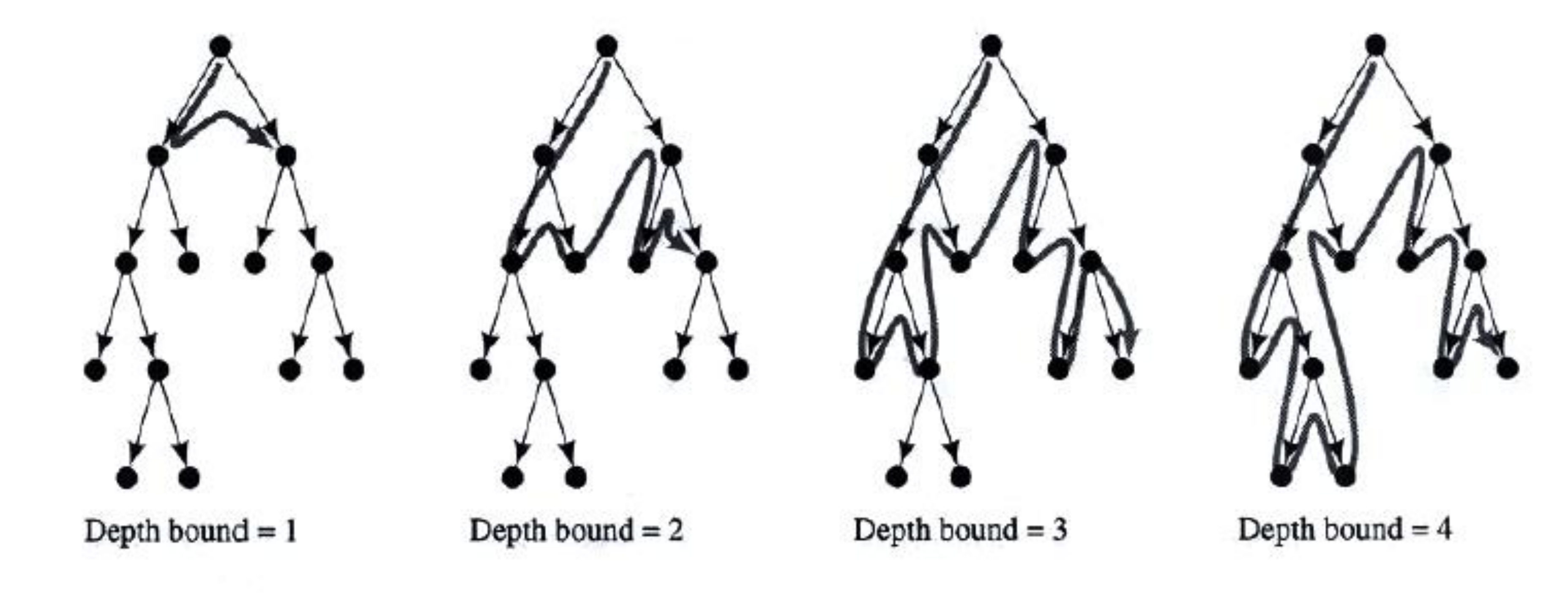
Iterative Deepening Search: $\ell = 2$



Iterative Deepening Search: $\ell = 3$



Iterative Deepening Search



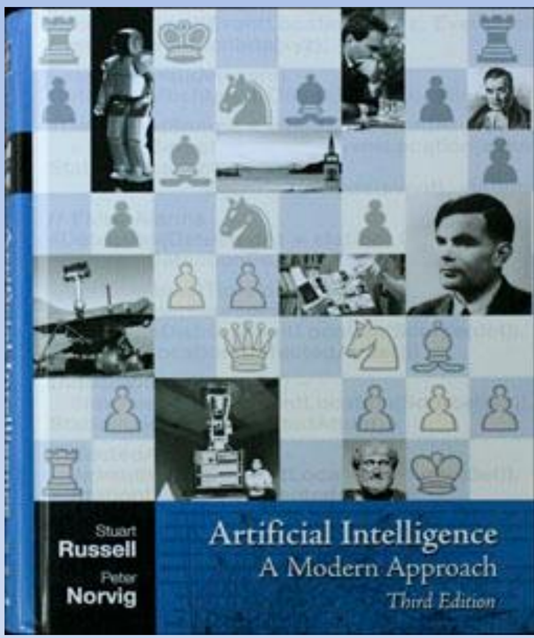
Properties of Iterative Deepening Search

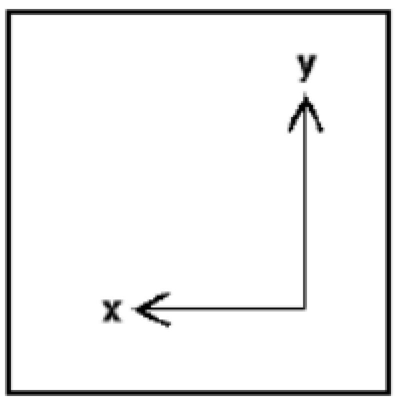
- Complete??: Yes
- Time??: $(d+1)b^0+db^1+(d-1)b^2+\ldots+b^d = O(b^d)$
- Space??: O(bd) Linear!
- Optimal??: Yes, if step cost = 1
  - Can be modified to explore uniform-cost tree.
- Numerical comparison for b=10 and d=5, solution at far right leaf:
  - N(IDS)=50+400+3000+20000+100000=123,450
  - N(BFS)=10+100+10000+100000+999990=1,111,100
- IDS does better because the nodes at depth d are not expanded
- BFS can be modified to apply goal test when a node is generate.

Summary of Algorithms

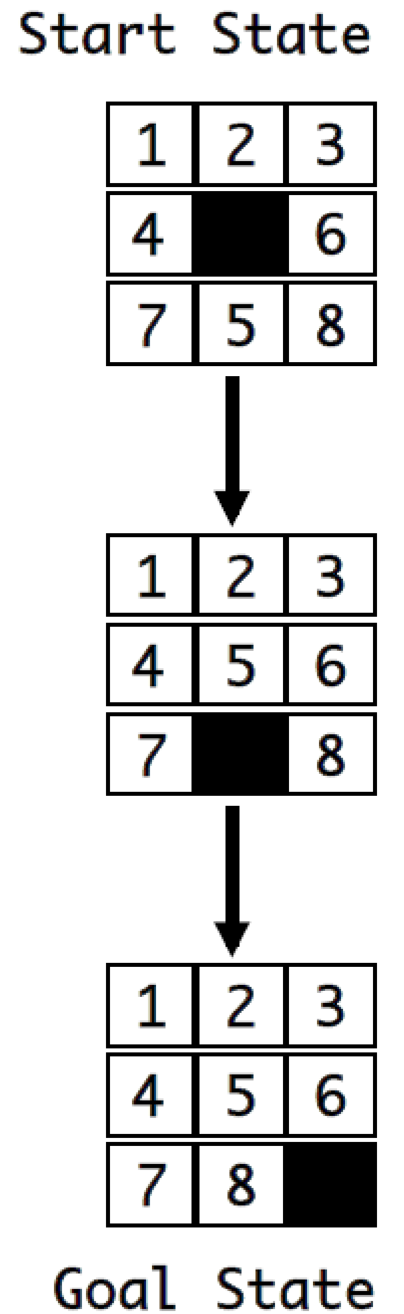
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

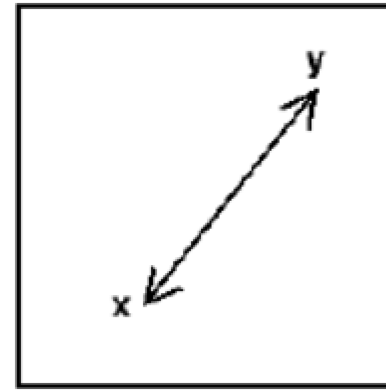
Chapter 3: Informed Search





Manhattan





Euclidean

Tree Search Algorithms
Choosing Next Leaf to Turn Over

Def `treeSearch(problem)`

’returns a solution or failure‘

initialize frontier using problem initial state

while `true:`

if `not frontier:` **return** `None`

STRATEGY **_choose a leaf node and remove from frontier_** STRATEGY

if `node is goal state:` **return** `node solution`

expand chosen node, adding results to frontier

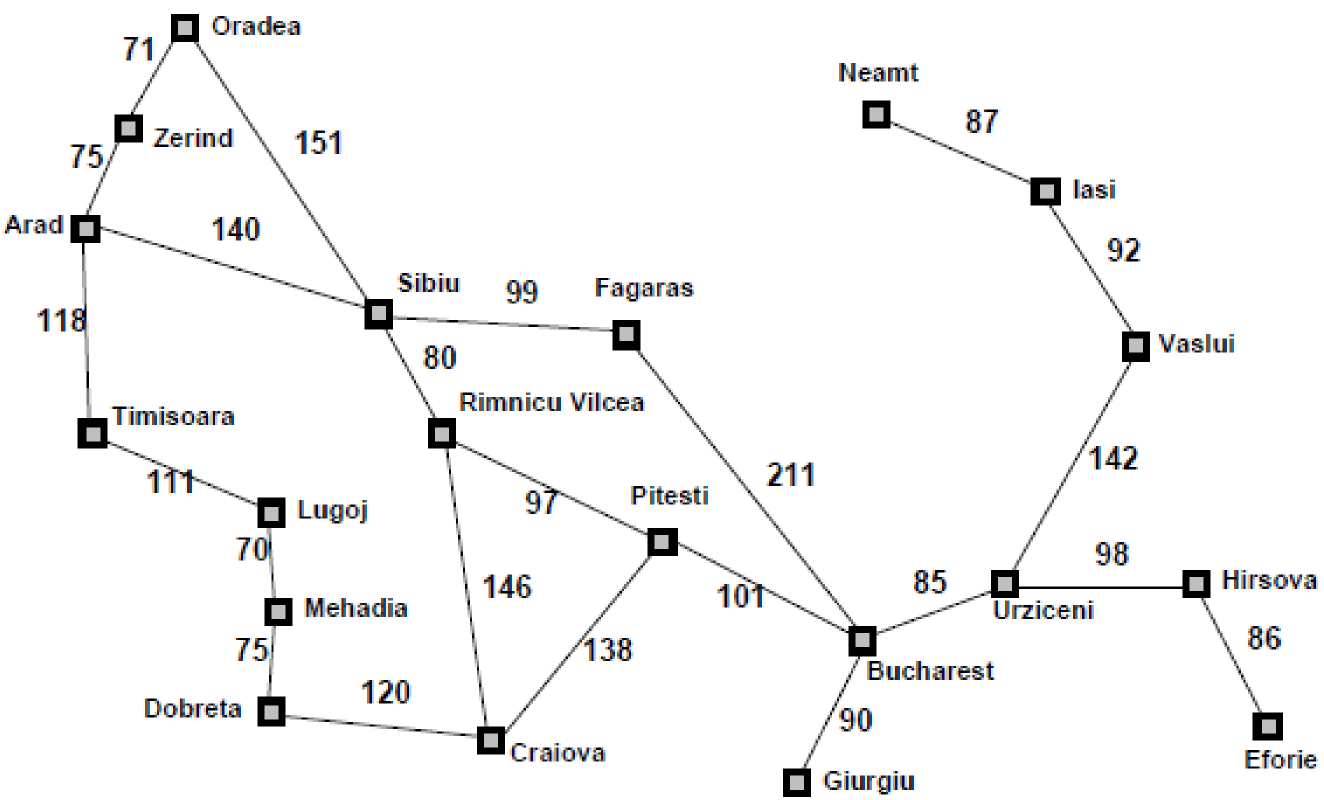
Best-First Search

- IDEA: Use an Evaluation Function for each to estimate “desirability”
- Expand most desirable unexpanded node.

- Implementation: Frontier is a queue sorted in decreasing order of desirability (heap).

Revisit: Getting to Bucharest

- NOW: With distance table.



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

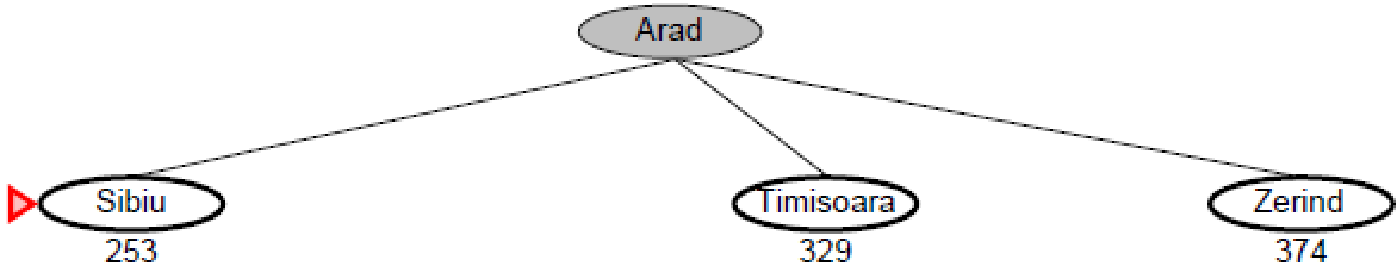
Heuristic: Distance Table

- Evaluation function h(n) (heuristic)
  - Estimate of cost from n to the closest goal.
- $H_{SLD}(n)$ is straight-line distance from n to Bucharest.
- Greedy search expands the node that appears to be closest to goal.

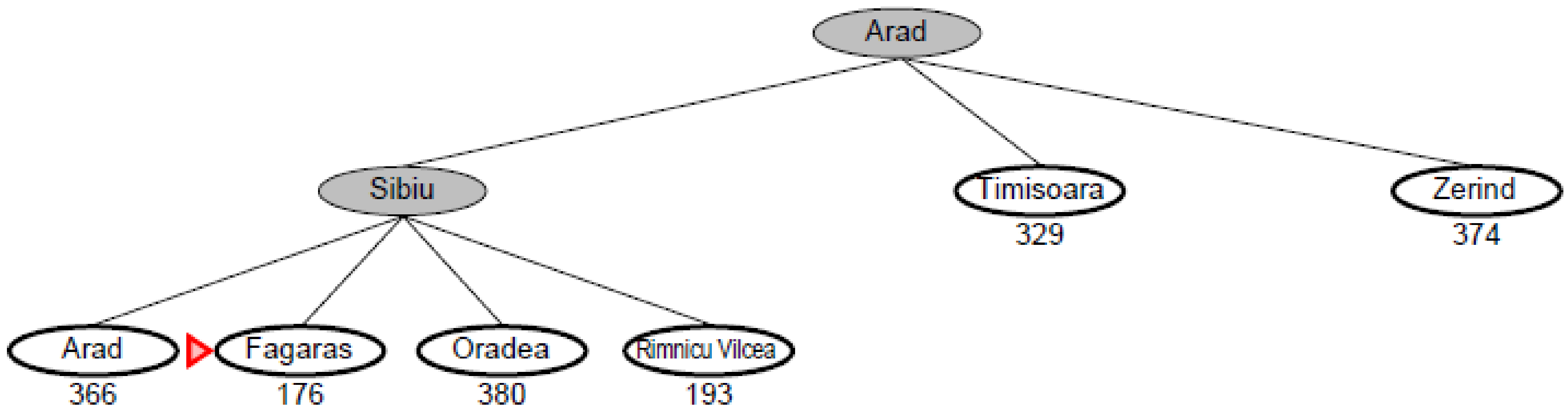
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Search Example

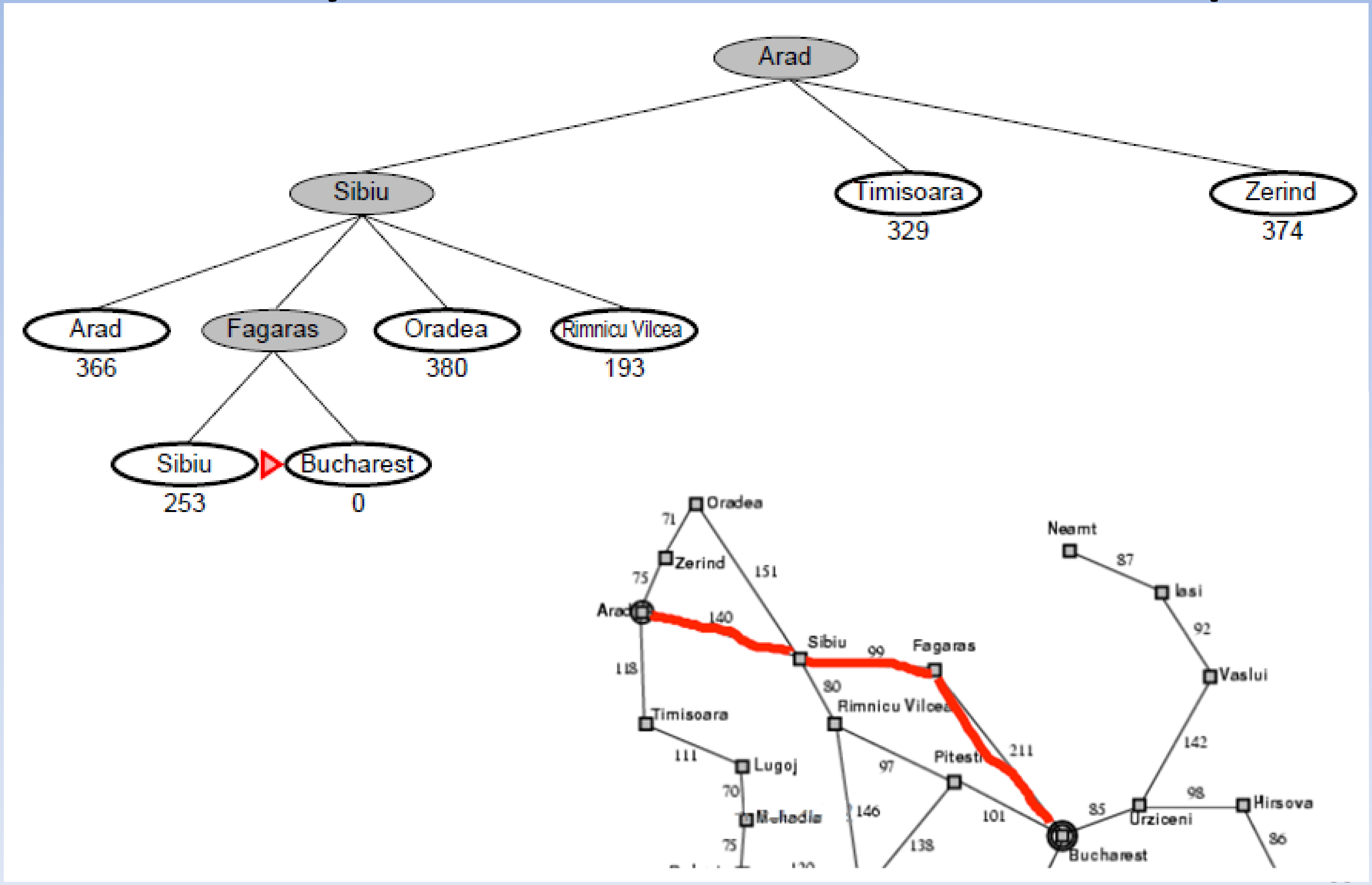
Greedy Best-First Search Example



Greedy Best-First Search Example



Greedy Best-First Search Example



Properties of Greedy Best-First Search

- Complete?? : NO- can get stuck in loops, e.g., with Oradea as a goal, Lasi -> Neamt -> Lasi -> Neamt ->
  - Complete in finite space with repeated-state checking.
- Time?? : $O(b^m)$, but a good heuristic can give dramatic improvement.
- Space?? : $O(b^m)$, keeps all nodes in memory.
- Optimal?? NO.

Challenge

- Can we incorporate heuristic evaluation function into Systematic Search???

Informed Search: Heuristic Search

- How should we use our heuristic knowledge (heuristic function) in systematic search.
- Where?
  - In Node Expansion
  - Hill Climbing
- Greedy Best-First Search
  - Select the best from ALL the ndoes encountered so far in Frontier.
  - “Good” use of of heuristic knowledge
- Heuristic estimates the value of a node
  - Promise of a node
  - Difficulty of solving the subproblem
  - Quality of solution represented by node
- f(n): Heuristic evaluation function
  - Depends on n, goal, search so far, domain

A* Search

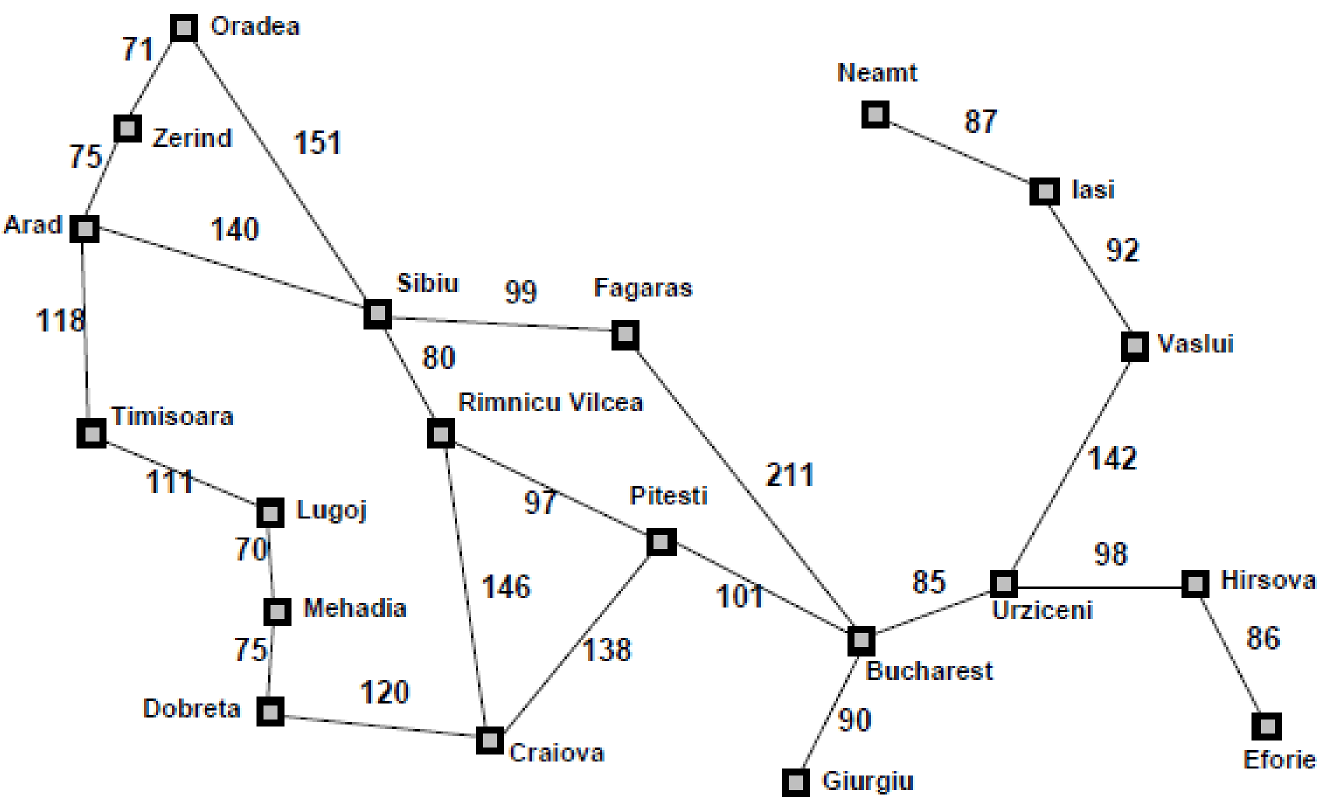
- IDEA: Avoid expanding paths that are already expensive.
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal.
- $f(n)$ = estimated total cost of path through n to goal.

Greedy vs A*

- Greedy Best-First Search just looked at estimated cost to goal from current location.
- A* looks at estimated Total Cost of solution:
  - Estimated cost to goal like greedy.
  - Cost to arrive at node from start state.

Revisit: Getting to Bucharest

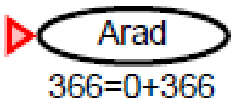
- NOW: Systematic Search w/ distance table.



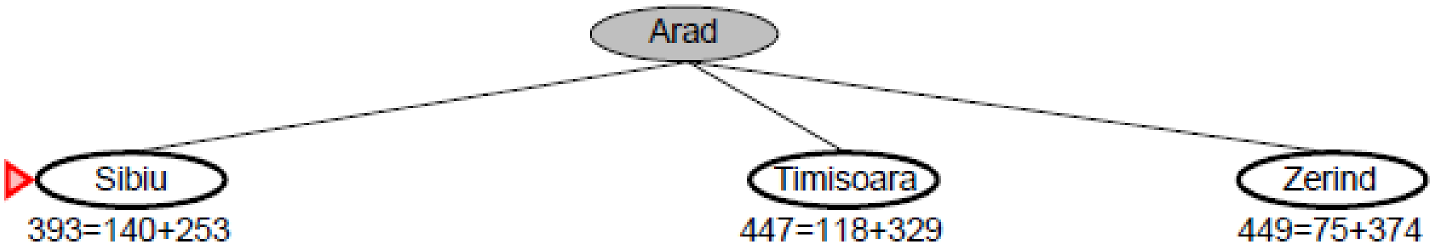
Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Example

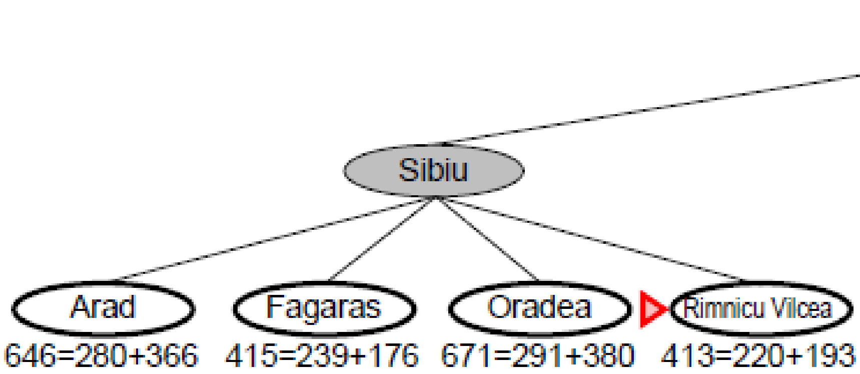


A* Example

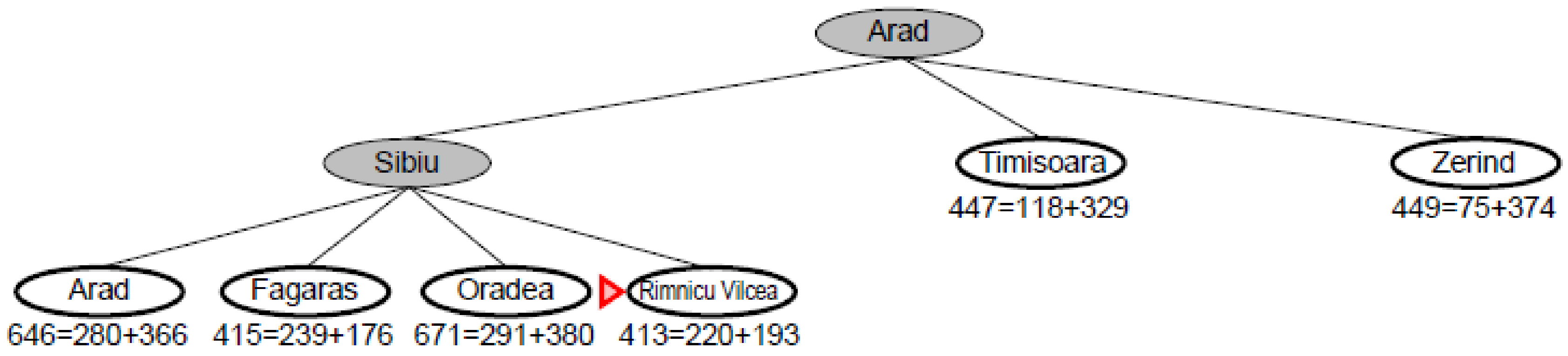


Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

A* Example

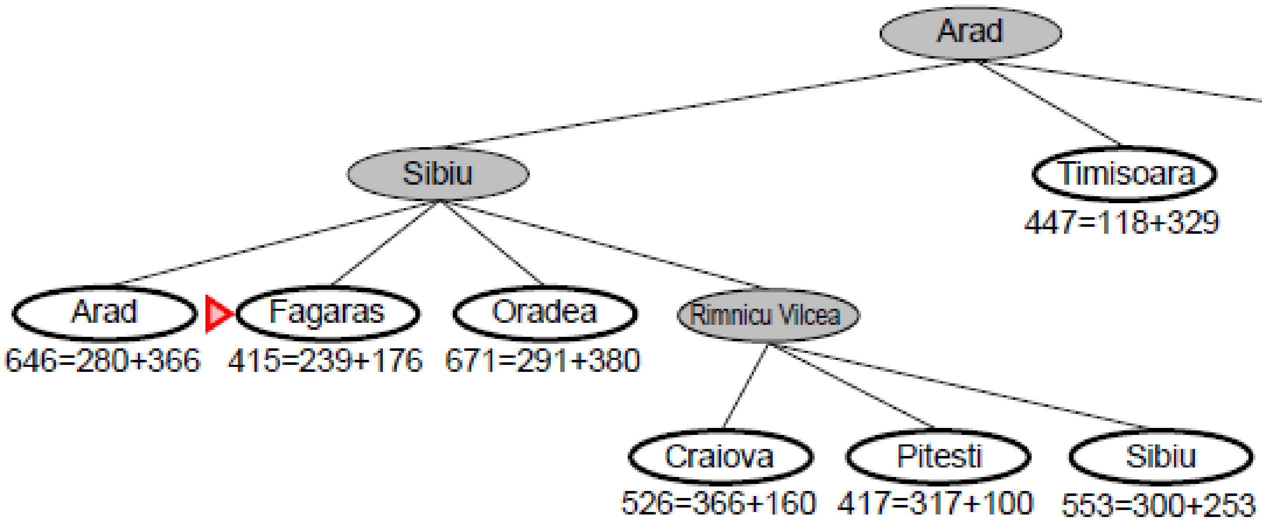

Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

A* Example

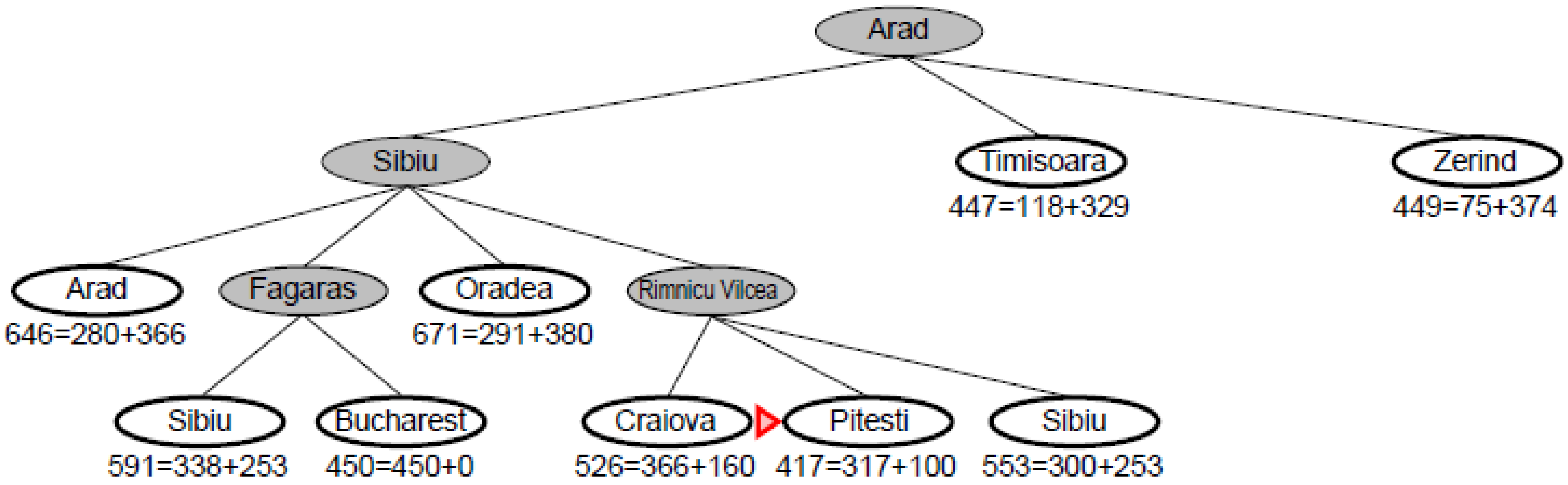


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

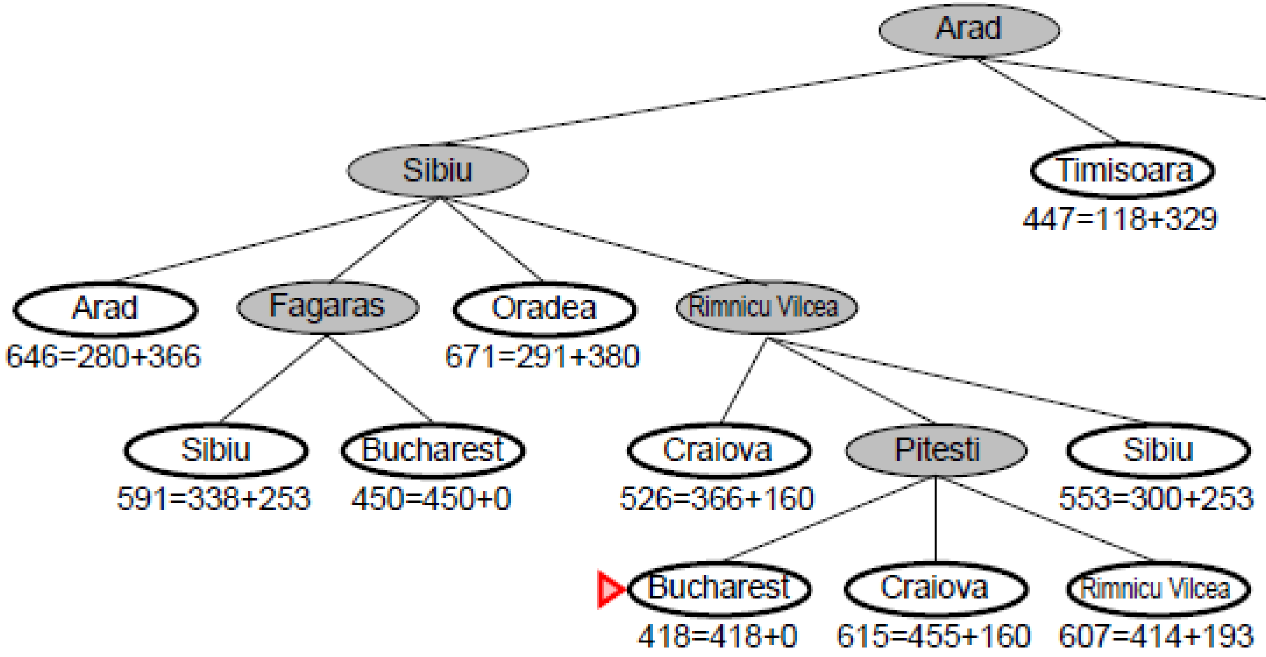
A* Example


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Example

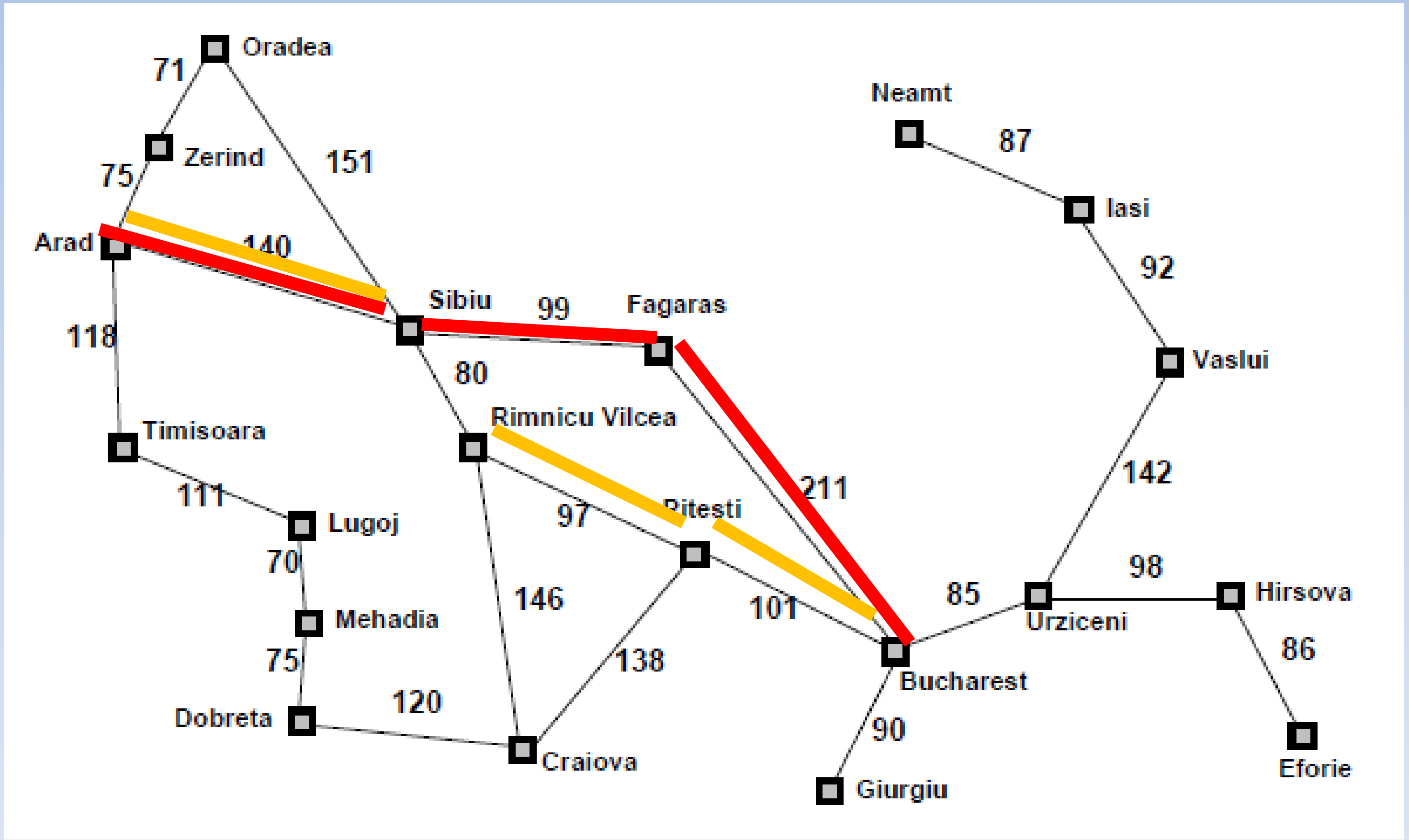

Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Example


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

Solution Comparison

- Greedy Best-First Search: 140+99+211=450
- A*: 140+80+97+101=418



Properties of A*

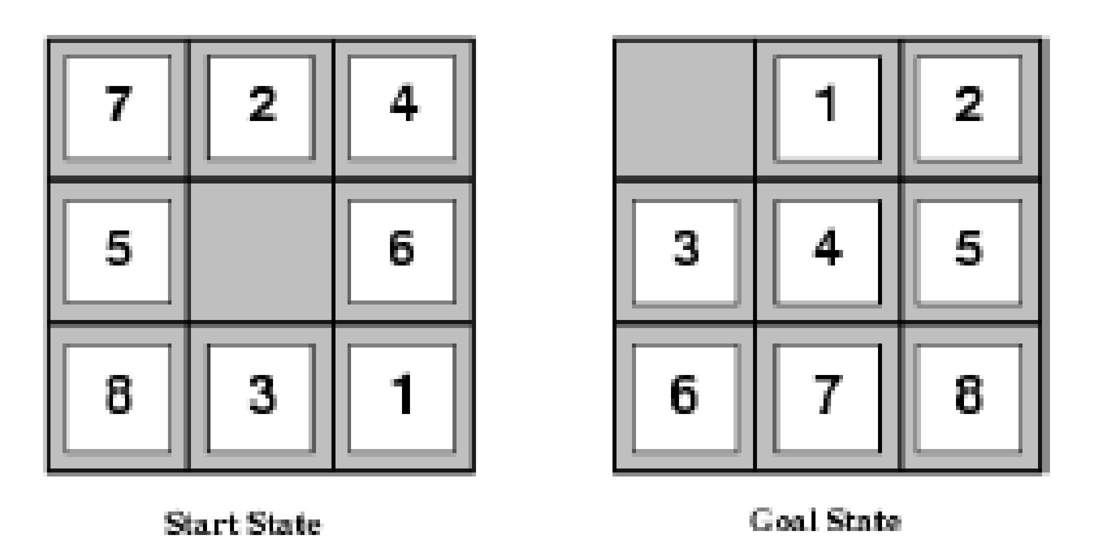
- **Complete??:** YES, Unless:
  - Infinitely Many nodes N where $f(N) \leq f(Goal)$
- **Time??:** Exponential in relative error in the heuristic function multiplied by the length of solution to find.
- **Space??;** Keeps all nodes in memory.
- **Optimal??:** Yes, cannot expand f_{i+1} until f_i is finished.
- A* expands all nodes N where f(N) is less than the optimal cost path (C^*)

Heuristics: A second look

- What are Heuristics:
  - A Rule of thumb: Intuition
  - A quick way to estimate how close we are to the goal. How close is a state to the goal..
- In Romania Example: Used Air Distance for Freeway Estimate.

Heuristics & 8-Puzzle

- 8-Puzzle
  - Heuristic 1: number of misplaced tiles.
  - Heuristic 2: Sum of the manhattan distance of each tile from its goal position.
- h1(Start)=8
- h2(Start)=18
  - 3+1+2+2+2+3+3+2

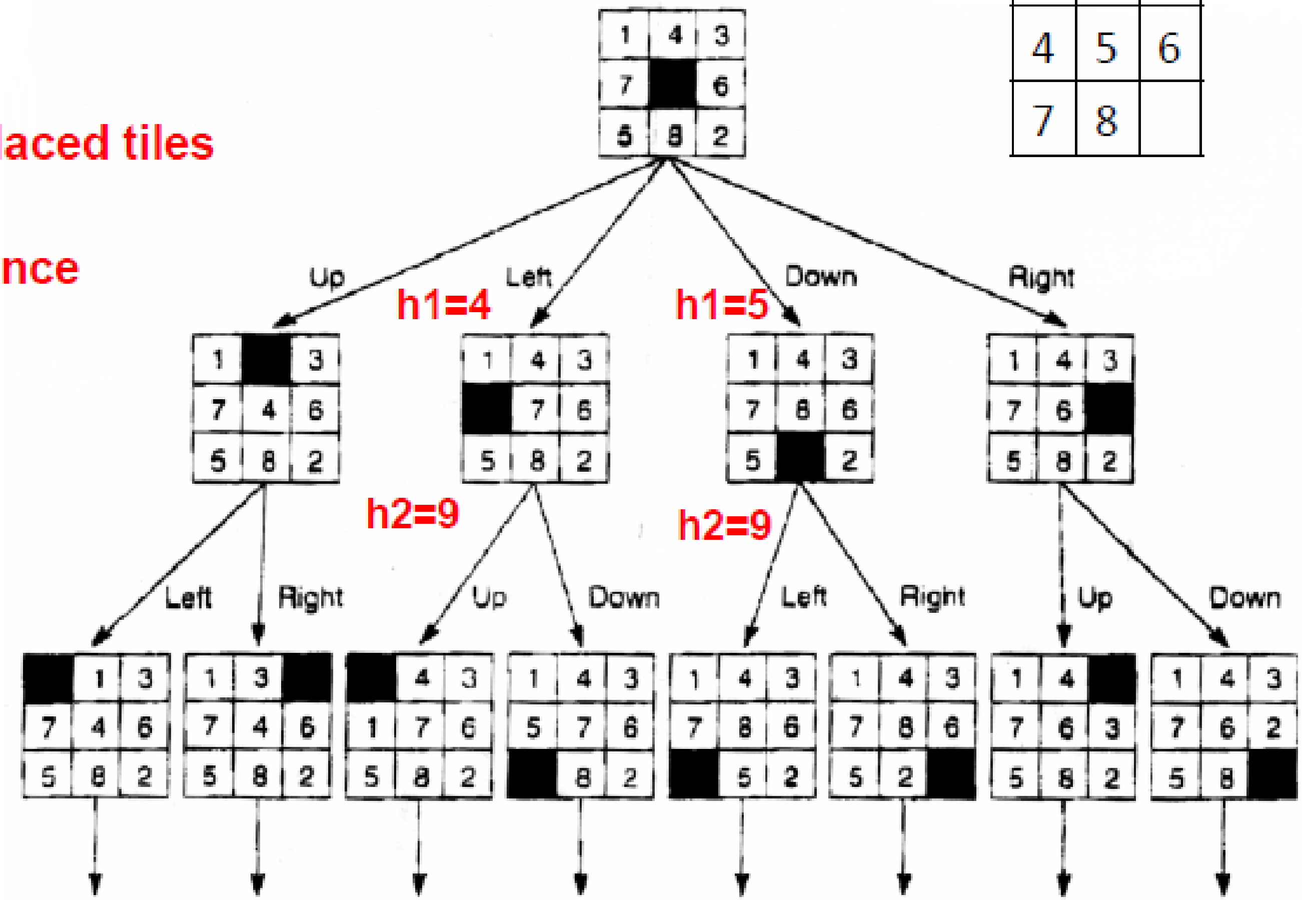


State Space of the 8 Puzzle Problem

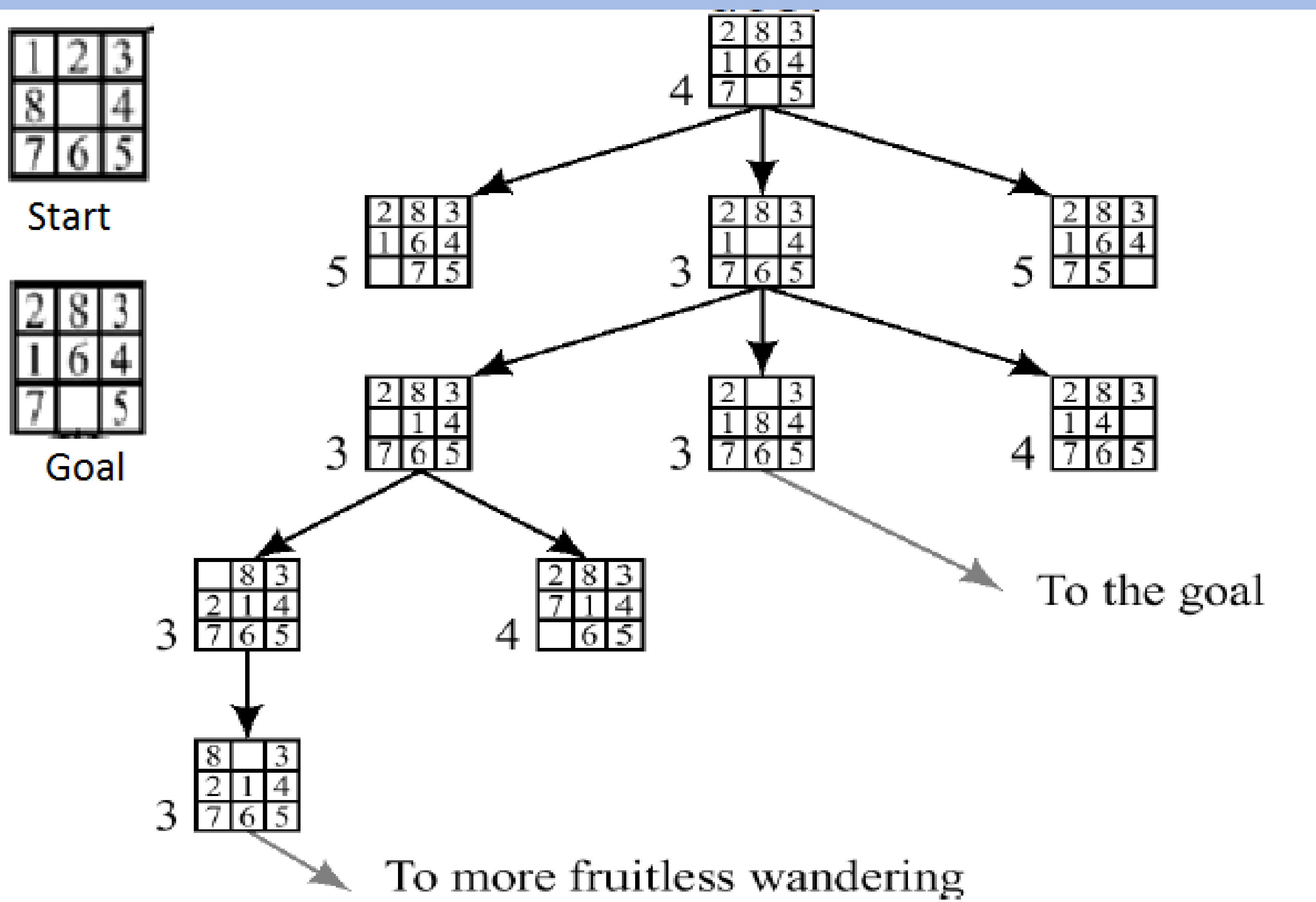
1	2	3
4	5	6
7	8	

h1 = number of misplaced tiles

h2 = Manhattan distance

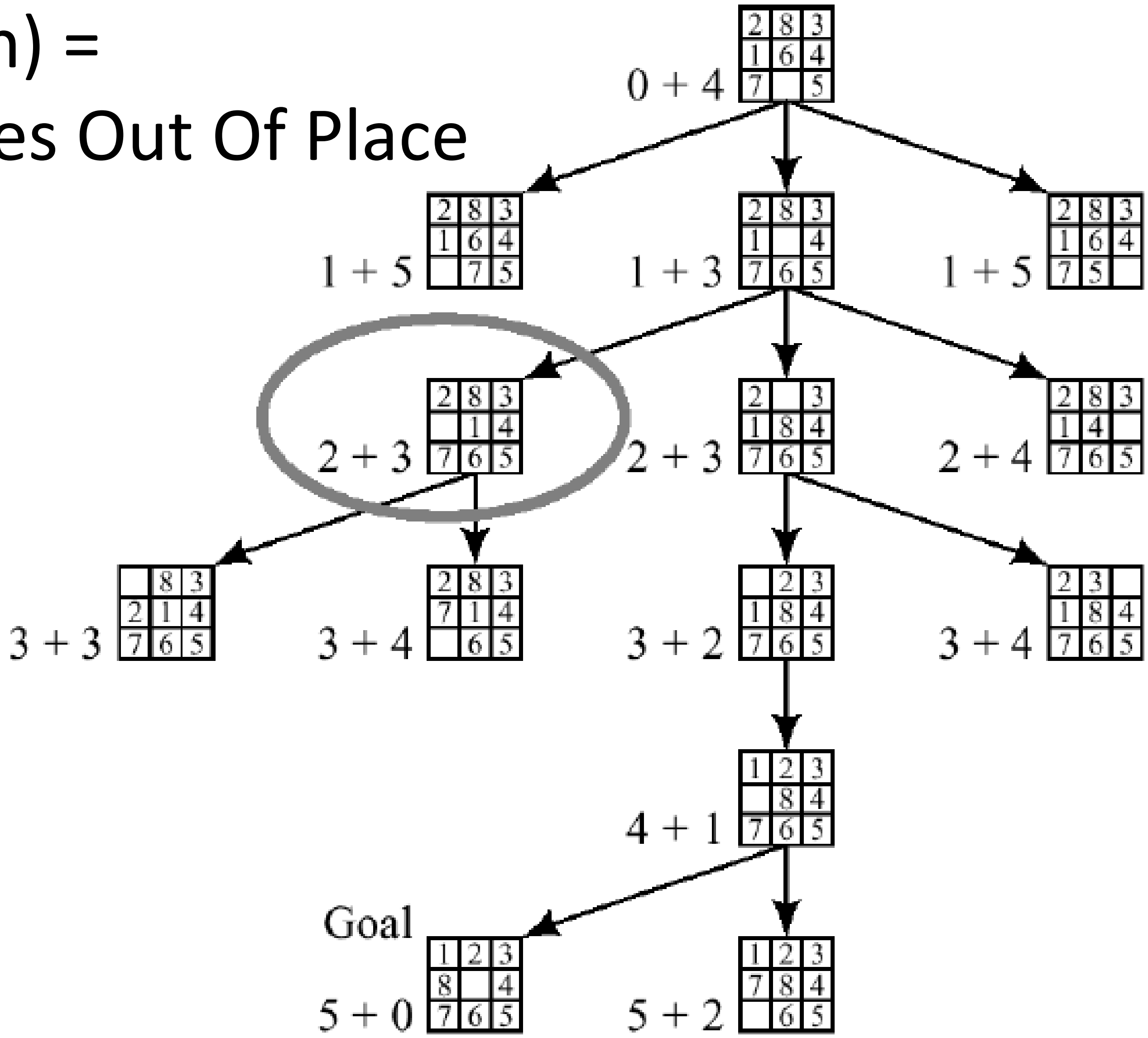


Greedy Search using Misplaced Tiles



A* Example: 8-Puzzle

h(n) =
Tiles Out Of Place



Key Concepts w/ Heuristics Terminology

- GOAL: find the minimum sum-cost path
- Notation:
  - $c(n, n')$= cost of arc (n, n')
  - $g(n)$ = cost of the current path from start state to current node n in the search tree.
  - $h(n)$ = estimate of the cheapest cost of a path from node n to a goal.
    - Heuristic Function
  - Special Evaluation Function: $f = g+h$

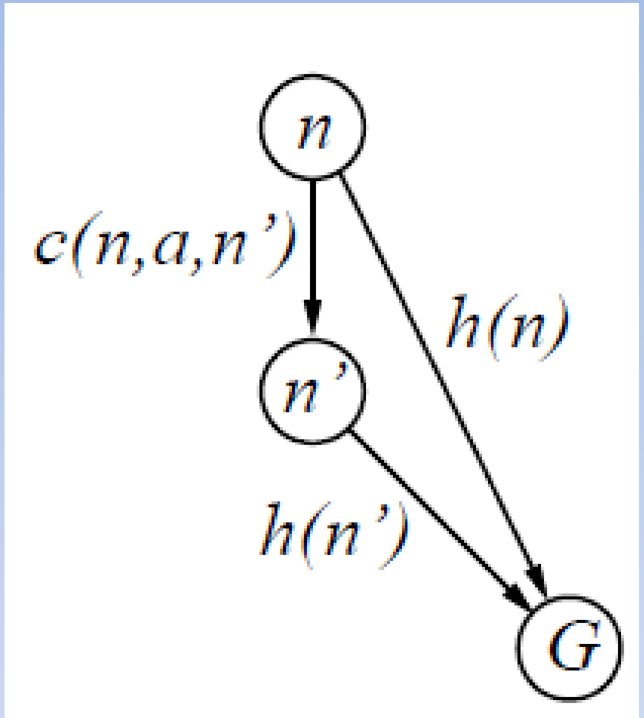
Key Concepts w/ Heuristics More Terminology

- f(n) estimates the cheapest cost solution path that goes through n.
  - h*(n) is the true cheapest cost from n to a goal.
  - g*(n) is the true shortest path from start state s to n.
- If the heuristic function, h, aways underestimates the true cost then A* is guaranteed to find an optimal solution.
  - h(n) is smaller than h*(n)

Key Concepts w/ Heuristics
1. Admissible

- The heuristic function h(n) is called **admissible** if h(n) is never larger than h*(n).
  - h(n) is always less than or equal to cheapest cost from n to goal.
- A* is admissible if it uses an admissible heuristic, and h(goal) = 0.

Key Concepts w/ Heuristics
2. Consistent

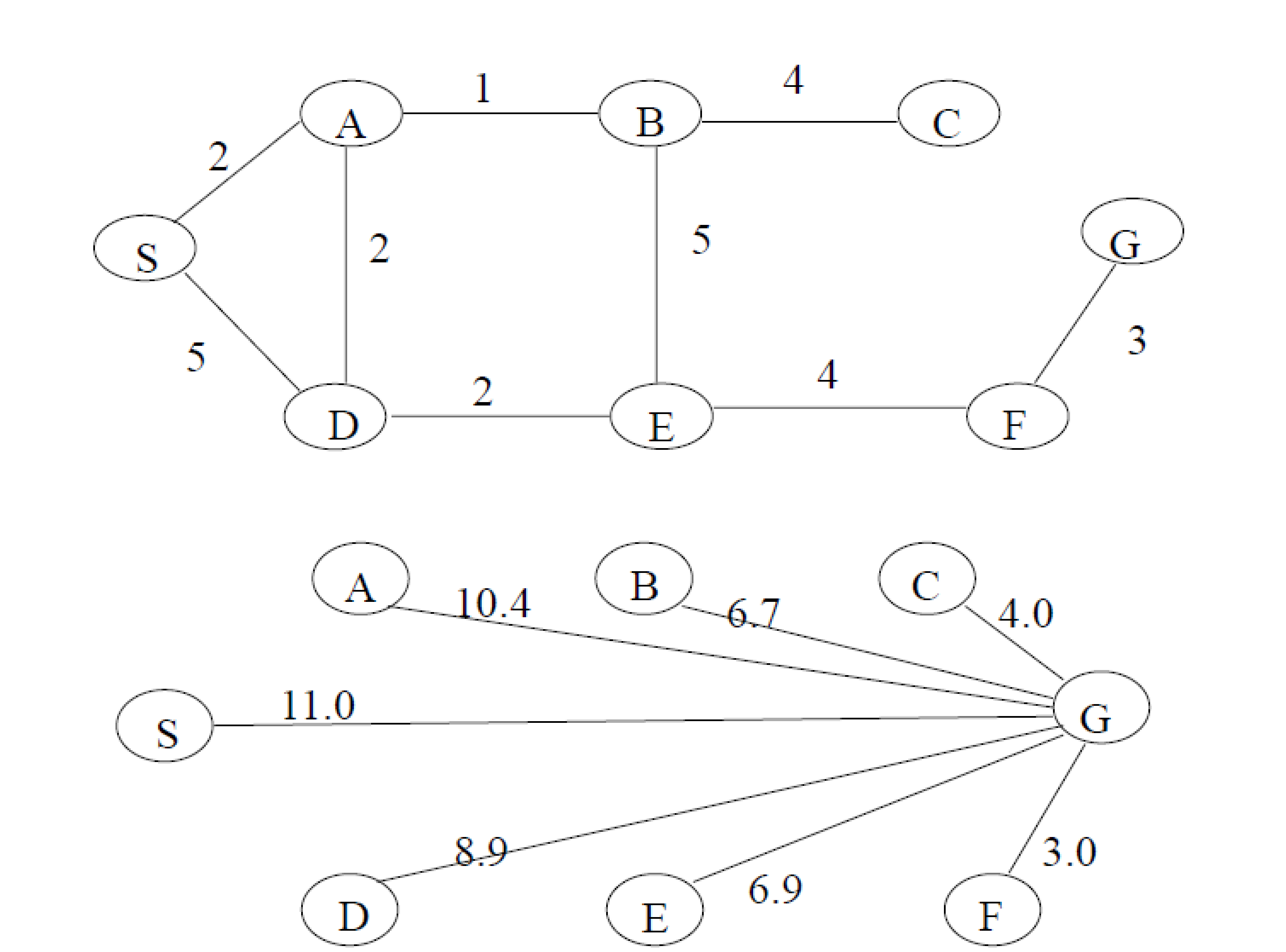


- The heuristic function h(n) is called **consistent** if for every node n, every successor n' of n generated by any action a, $h(n) \leq c(n,a,n') + h(n')$

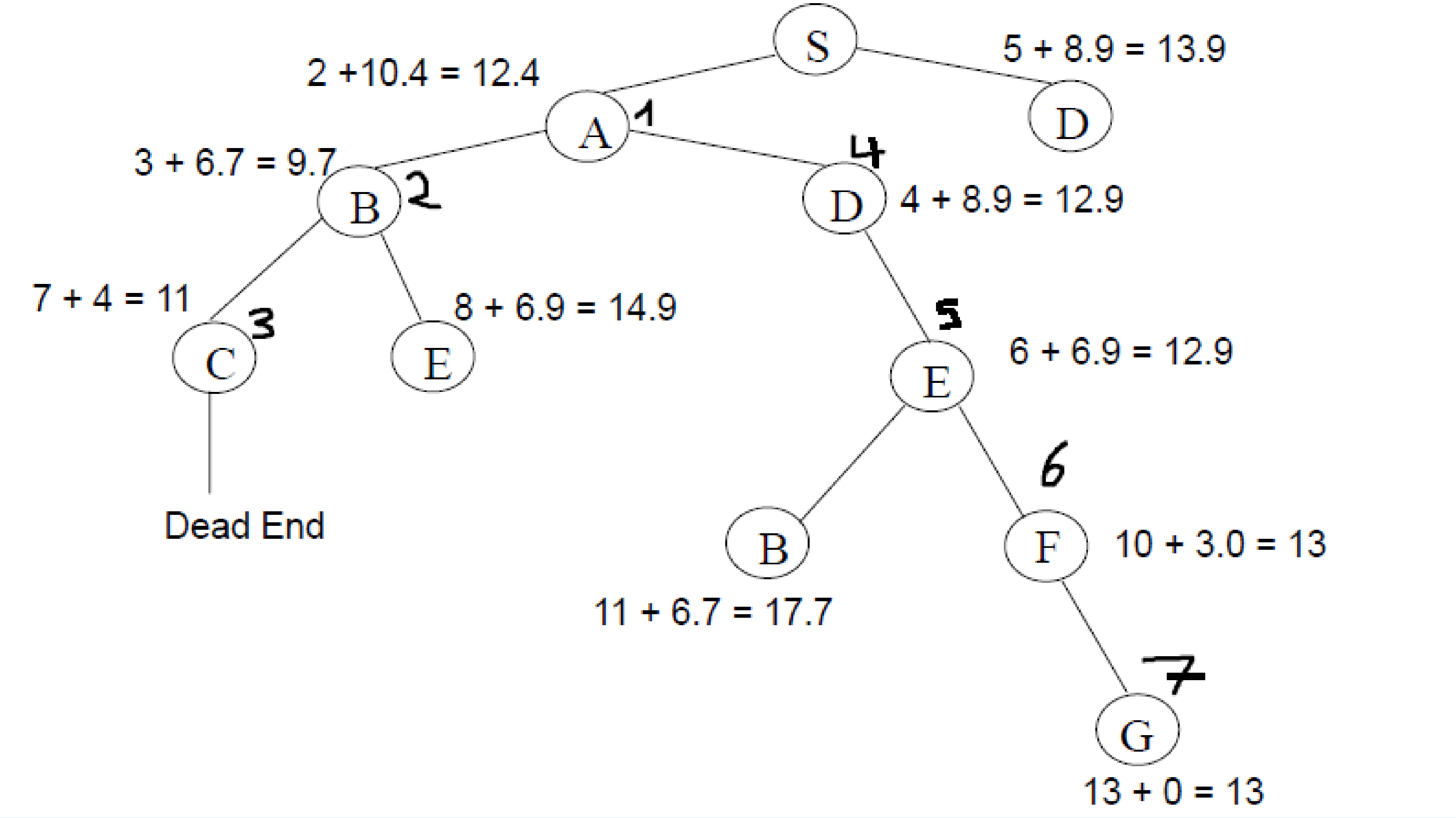
- If h is consistent:
  - $f(n') = g(n') + h(n')$
    $= g(n) + c(n,a,n') + h(n')$
    $\geq g(n) + h(n)$
    $= f(n)$

- f(n) is non-decreasing on any path.

Another A* Example

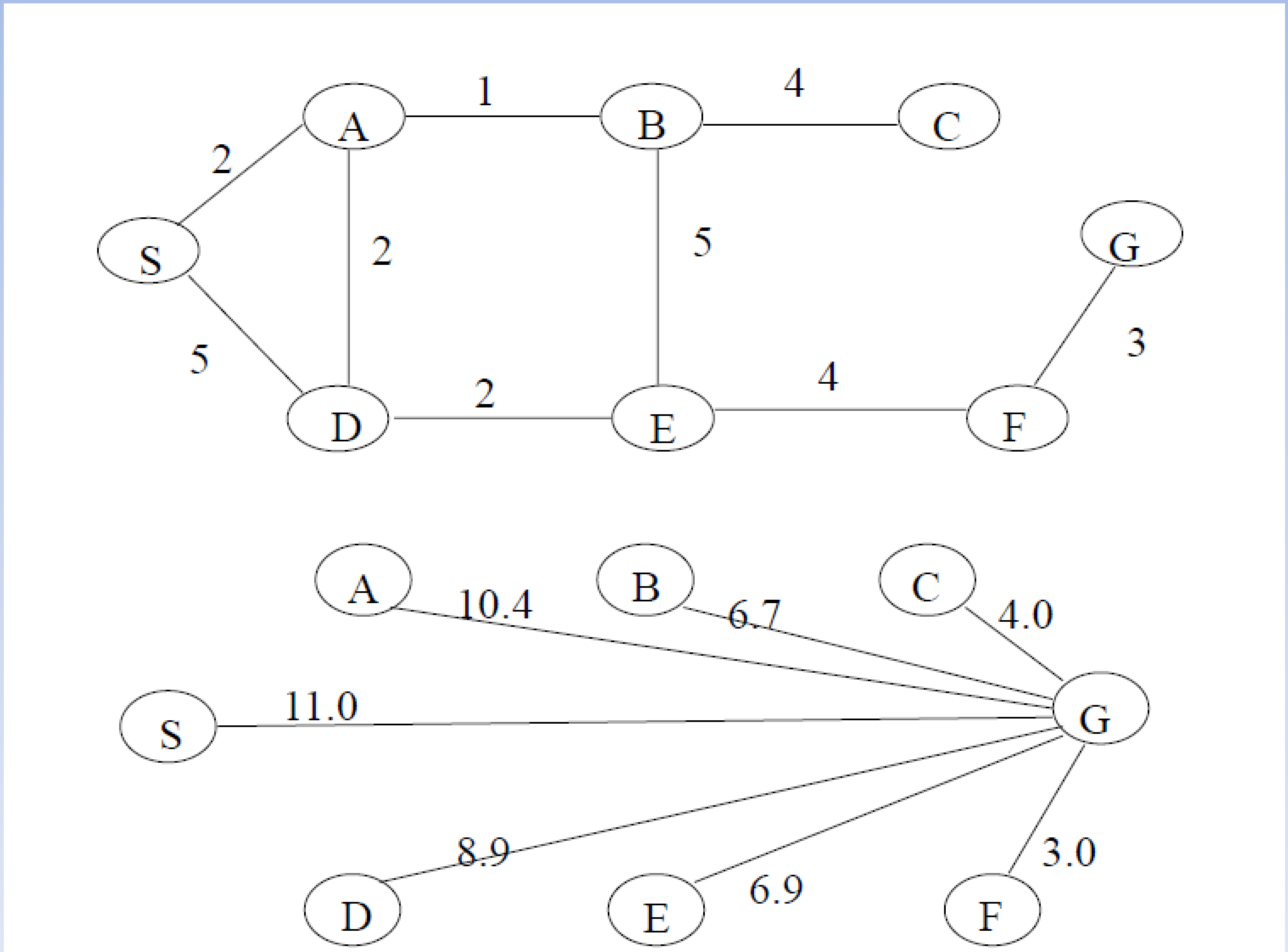


A* In Action

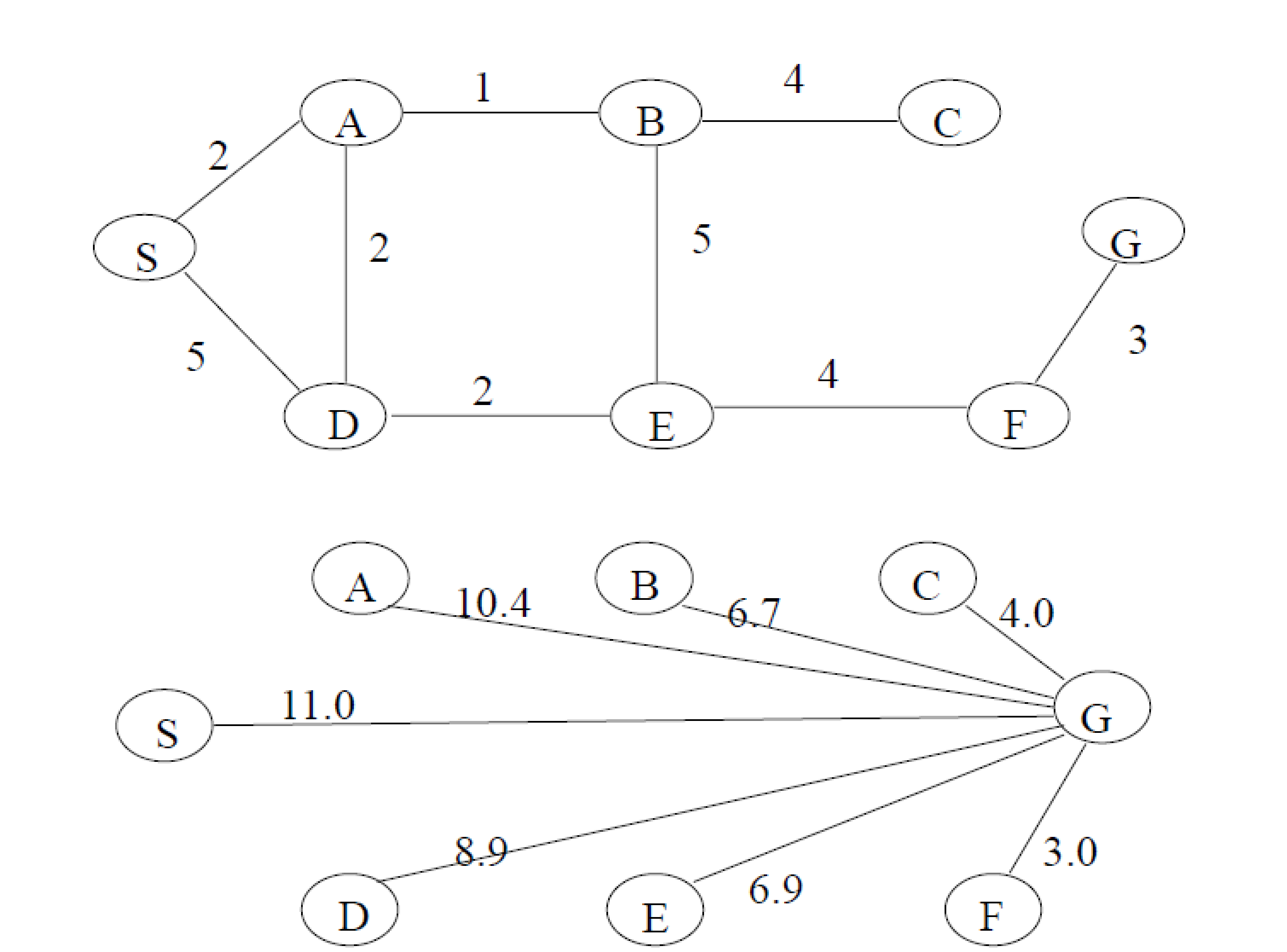


S
2 +10.4 = 12.4
5 + 8.9 = 13.9
A
1
D
3 + 6.7 = 9.7
B
2
4
D
4 + 8.9 = 12.9
7 + 4 = 11
3
C
E
8 + 6.9 = 14.9
5
E
6 + 6.9 = 12.9
Dead End
6
B
F
10 + 3.0 = 13
11 + 6.7 = 17.7
7
G
13 + 0 = 13

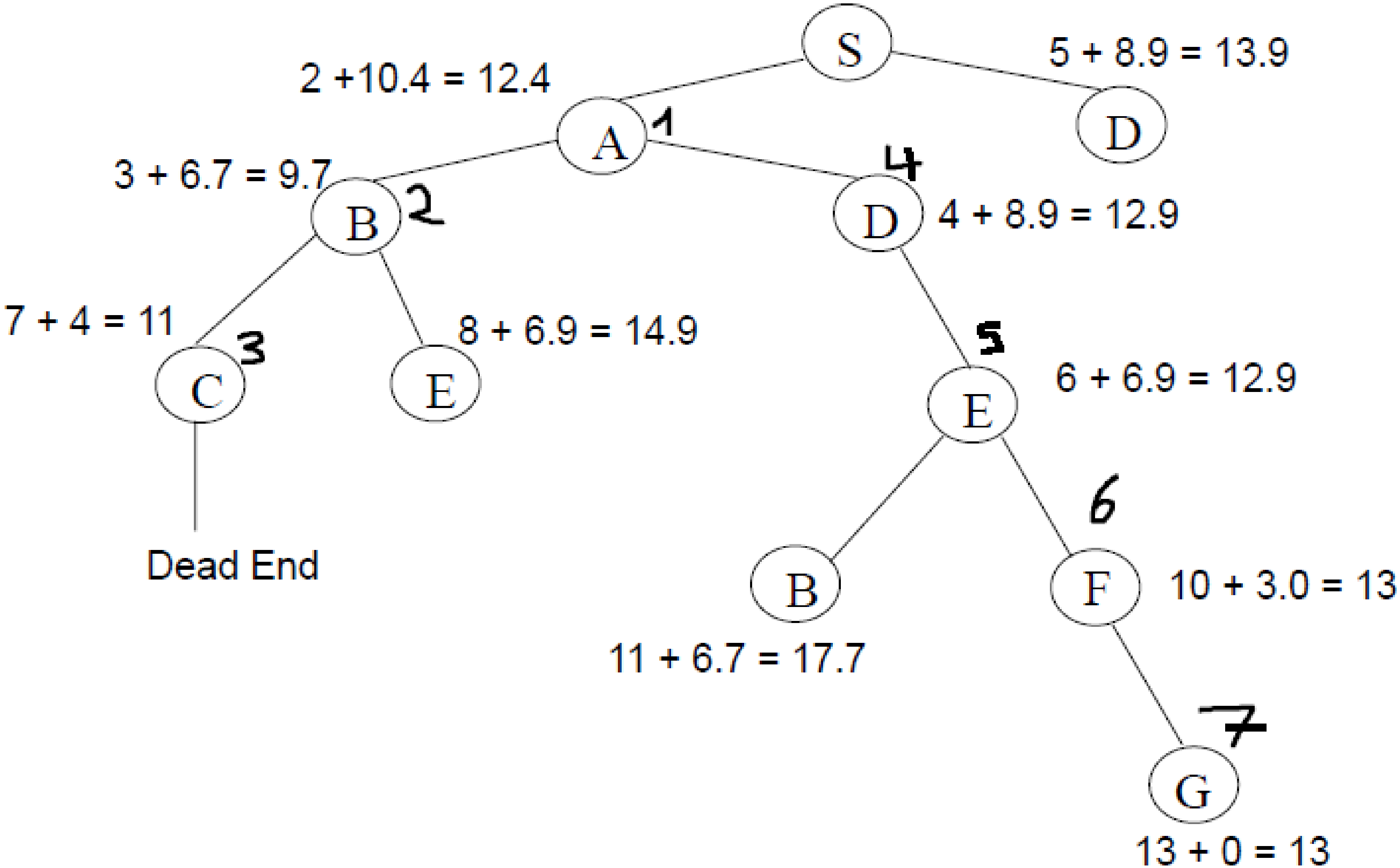
Is Our Heuristic Admissible?



Is Our Heuristic Consistent?



Is Heuristic Consistent?
If f(n) Non-Decreasing Along all Paths?



Effectiveness of A* Search Algorithm

Average number of nodes expanded

d	IDS	A*(h1)	A*(h2)
2	10	6	6
4	112	13	12
8	6384	39	25
12	364404	227	73
14	3473941	539	113
20	------------7276	676	

Average over 100 randomly generated 8-puzzle problems
h1 = number of tiles in the wrong position
h2 = sum of Manhattan distances

Dominance

- Definition: If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 **dominates** h_1
- Ask Yourself:

Is h_2 better for search???

Dominance: Results

- Typical search costs (average number of nodes expanded):

- d=12
  - IDS: 3,644,035 nodes
  - $A^*(h_1)$ = 227 nodes
  - $A^*(h_2)$ = 73 nodes
- d=24
  - IDS = Too many nodes to explore
  - $A^*(h_1)$ = 39,135 nodes
  - $A^*(h_2)$ = 1,641 nodes

- H_2 DOMINATES Search Results!

IDA*

- A* main problem is Memory Requirements
- Let's Adapt Iterative Deepening to A*
  - Simplest way to reduce memory requirements.
- Iterative deepening uses a depth cutoff
- Iterative deepening A* uses f-cost (g+h) as cutoff.
- Each iterative cutoff is set to the smallest f-cost of any node that exceeded the cutoff on previous iteration.
- IDA* is practical for many problems.

Iterative-Deepening A*

function IDA*(*problem*) **returns** a solution
  inputs: *problem*, a problem

  fmax ← h(*initial state*)
  for i ← 0 **to** ∞ **do**
    result ← LIMITED-F-SEARCH(*problem*, *fmax*)
    if *result* is a solution **then return** *result*
    else *fmax* ← result
  end

function LIMITED-F-SEARCH(*problem*, *fmax*) **returns** solution or number

  depth-first search, backtracking at every node n such that $f(n) > fmax$
  if the search finds a solution
  then return the solution
  else return $\min\{f(m) \mid$ the search backtracked at node $m\}$

RBFS

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
   return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

Figure 3.26 The algorithm for recursive best-first search.

Inventing Heuristics

- Problem Relaxation
- Pattern Databases
- Learning Heuristics from Experience

Relaxed Problems

- A problem with fewer restrictions on the actions
- The cost of an optimal solution to relaxed problem is admissible heuristic for original.
- If the rules of the 8-puzzle are relaxed so a tile can move anywhere, we have (number of misplaced tiles) heuristic.
- If the rules of the 8-puzzle are relaxed so a tile can move to any adjacent square, we have (Manhattan Distance) heuristic.

Relaxed Problems
Traveling Salesman Problem

- Salesman needs to visit a bunch of cities.
- Only wants to visit each city once.
- Find a tour. A tour is:
  1. A graph
  2. Connected
  3. Each node has degree 2 (In/Out)
- Relaxing 3 yields Minimum Spanning Tree (MST) Heuristic
- MST is a lower bound on the shortest tour.

Pattern Databases

- Store optimal solution length for Subproblems (patterns)
- Use Pattern Database to create admissible heuristic by looking up corresponding subproblem in DB.