Outline

- Problem-solving agents
  - Problem types
  - Problem formulation
  - Example problems
- Tree search algorithms
  - Breadth-First Search
  - Depth-First Search
  - Limited-Depth Search
  - Iterative Deepening
- Graph search algorithms

Problem-Solving Agents

- Simple reflex agents
  - have a direct mapping from states to actions
  - typically too large to store
  - would take too long to learn
- Goal-Based agents
  - can consider future actions and the desirability of their outcomes
- Problem-Solving Agents
  - special case of Goal-Based Agents
  - find sequences of actions that lead to desirable states
- Uninformed Problem-Solving Agents
  - do not have any information except the problem definition
- Informed Problem-Solving Agents
  - have knowledge where to look for solutions

Formulate-Search-Execute Design

- Formulate:
  - Goal formulation:
    - A *goal* is a set of world states that the agents wants to be in (where the goal is achieved)
    - Goals help to organize behavior by limiting the objectives that the agent is trying to achieve
  - Problem formulation:
    - Process of which actions and states to consider, given a goal
- Search:
  - the process of finding the solution for a problem in the form of an action sequence
  - *an agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that lead to states of known value, and then choosing the best*
- Execute:
  - perform the first action of the solution sequence

Simple Problem-Solving Agent

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**
goal ← FORMULATE-GOAL(*state*)
problem ← FORMULATE-PROBLEM(*state*, *goal*)
seq ← SEARCH(*problem*)
action ← RECOMMENDATION(*seq*, *state*)
seq ← REMAINDER(*seq*, *state*)
return *action*

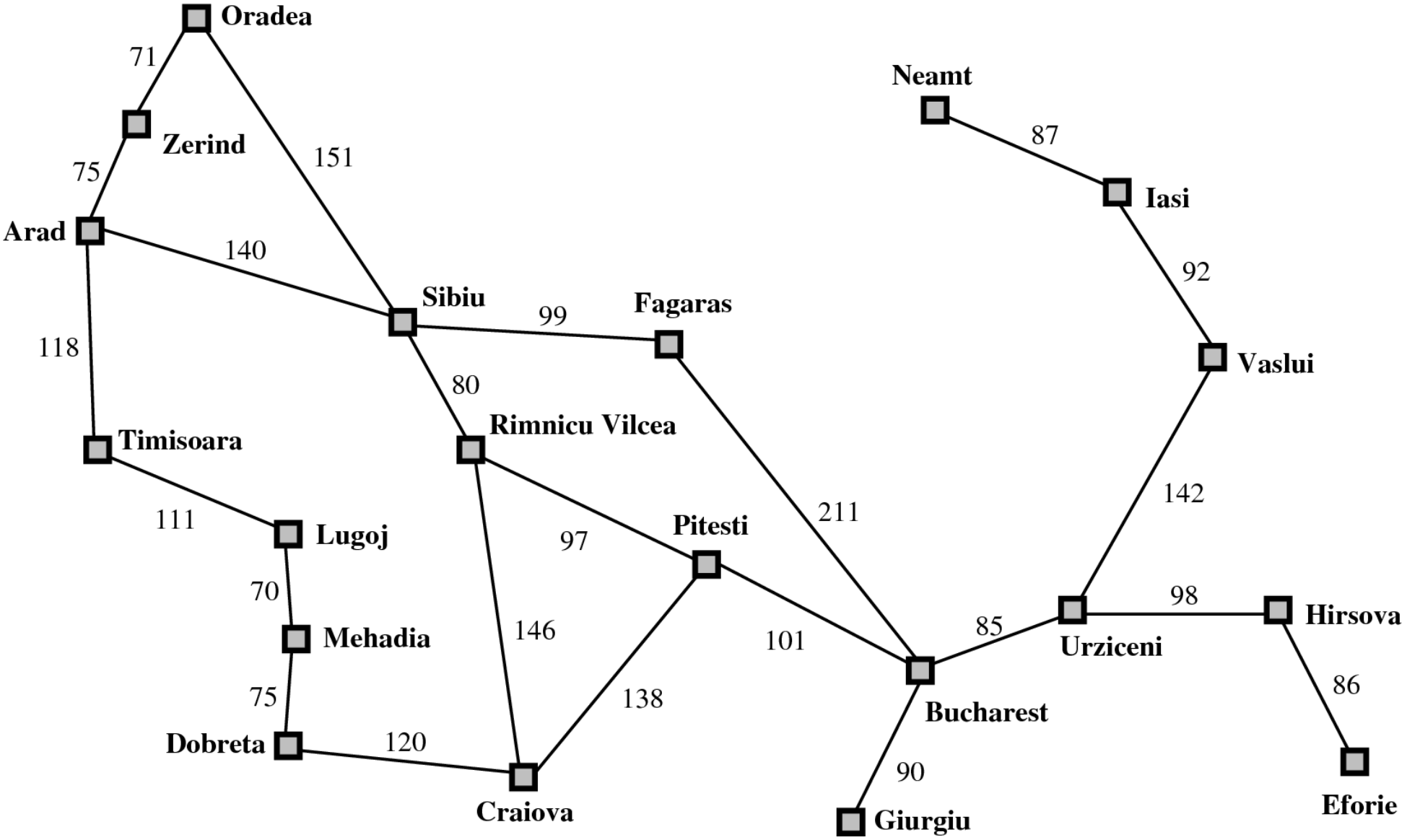
Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest

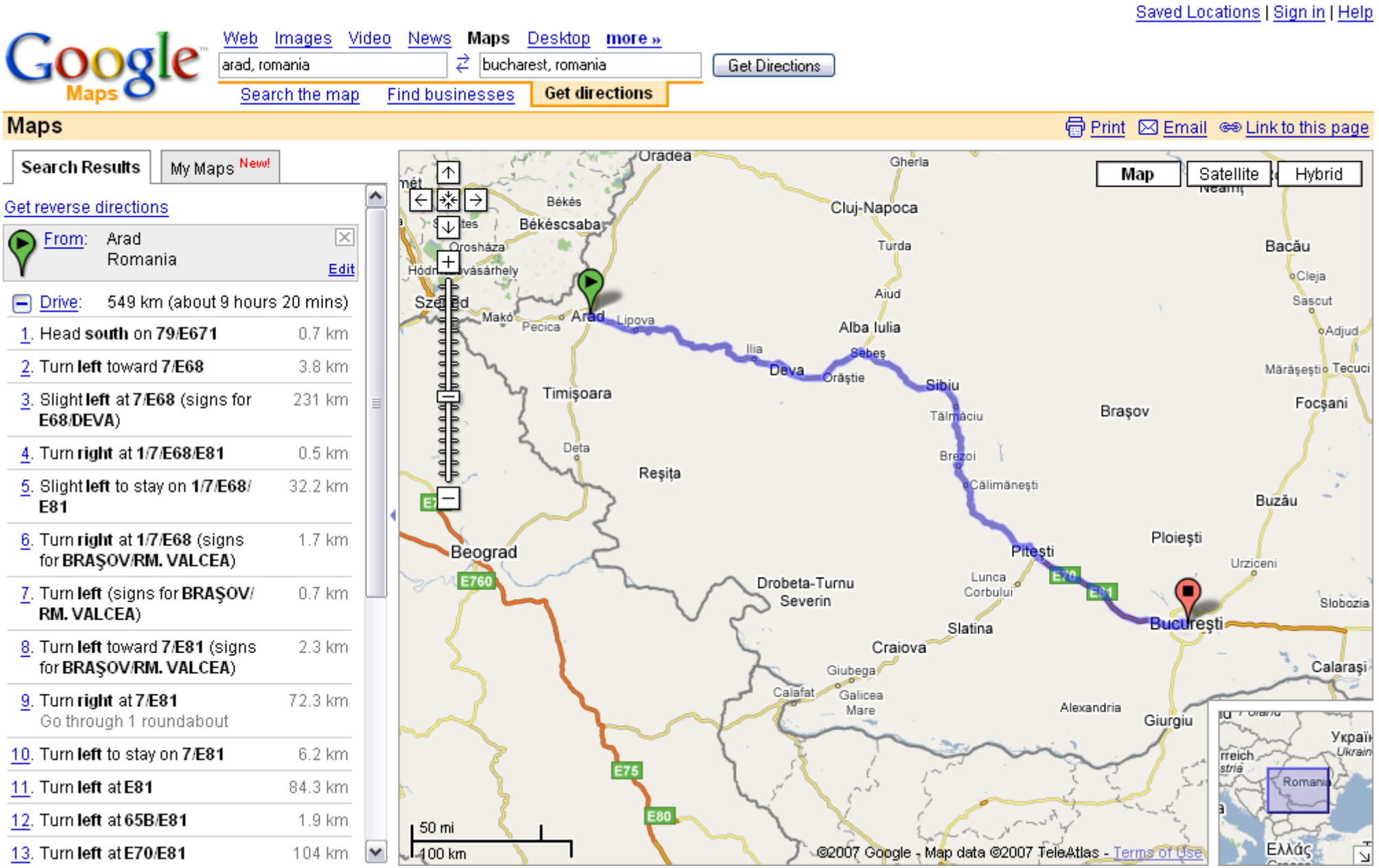
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Rimnicu Vilcea, Pitesti

- Assumption:
  - agent has a map of Romania, i.e., it can use this information to find out which of the three ways out of Arad is more likely to go to Bucharest

Example: Romania



Example: Romania



Assumptions about the Environment

- static
  - we do not pay attention to possible changes in the environment
- observable
  - we can at least observe our initial state
- discrete
  - possible actions can be enumerated
- deterministic
  - the expected outcome of an action is always identical to the true outcome
  - once we have a plan, we can execute it „with eyes closed“

→ easiest possible scenario

Problem Types

- Single-State Problem

  deterministic, fully observable

  - agent knows exactly which state it will be in
  - solution is a sequence
- Conformant Problem (sensorless problem)

  non-observable

  - agent may have no idea where it is
  - solution (if any) is a sequence
- Contingency Problem

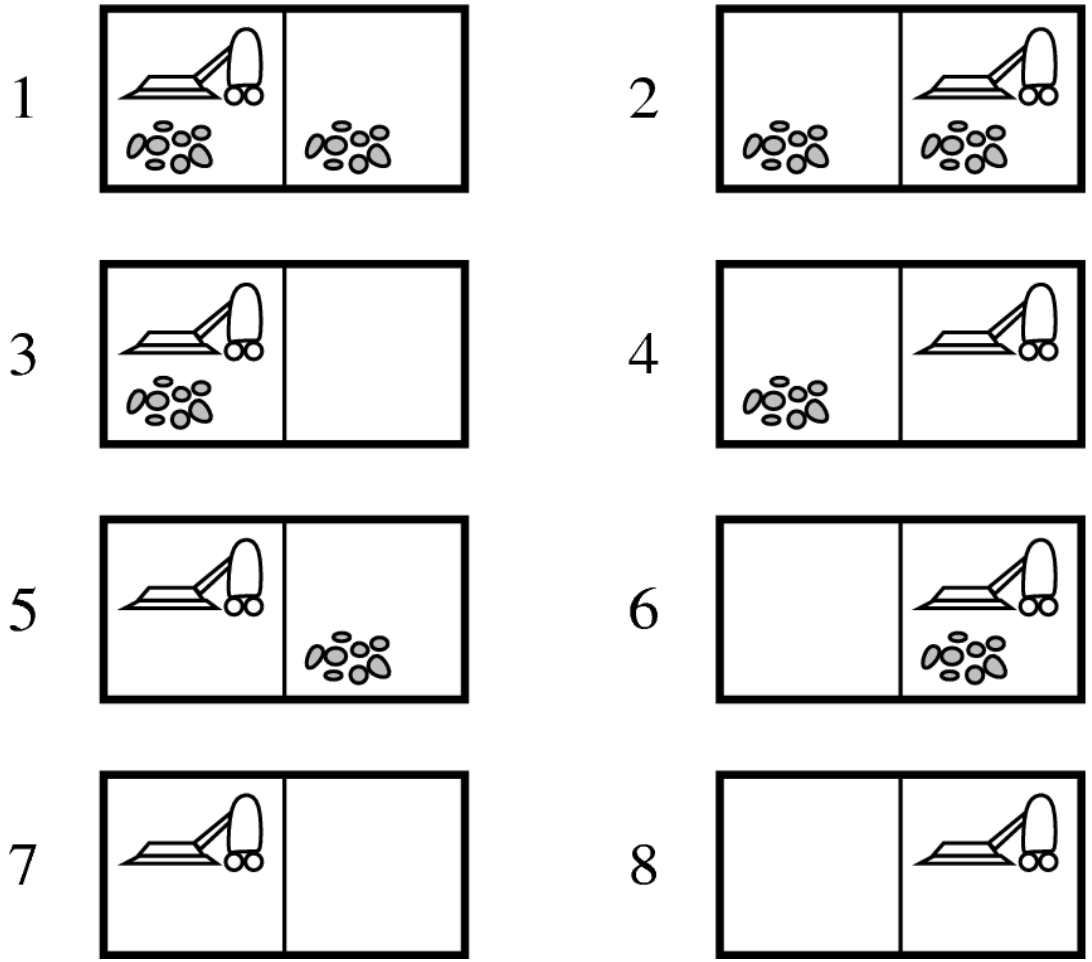
  nondeterministic and/or partially observable

  - percepts provide new information about current state
  - solution is a contingent plan (tree) or a policy
  - search and execution often interleaved
- Exploration Problem

  state-space is not known

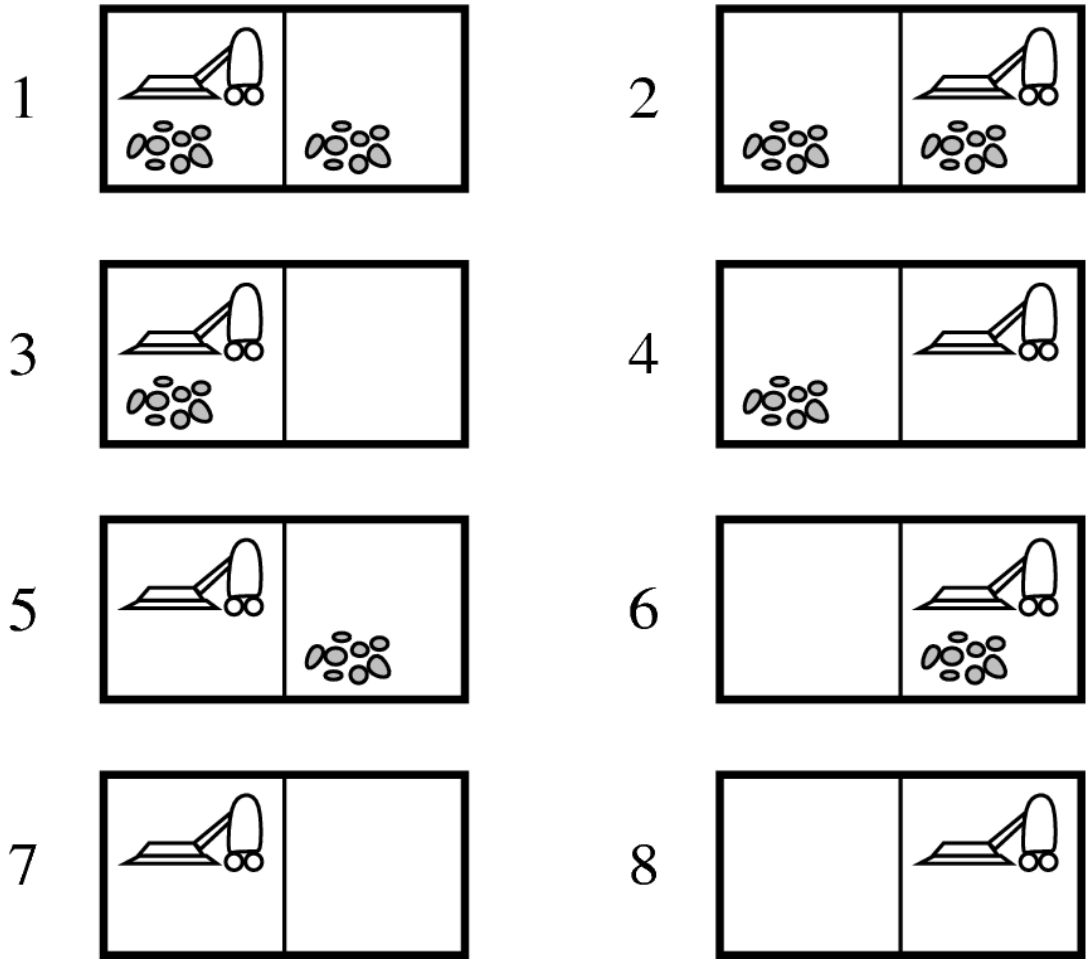
Example: Vacuum World

- Single-state Problem
  - start in #5
  - goal
    - no dirt
- Solution
  - [*Right, Suck*]



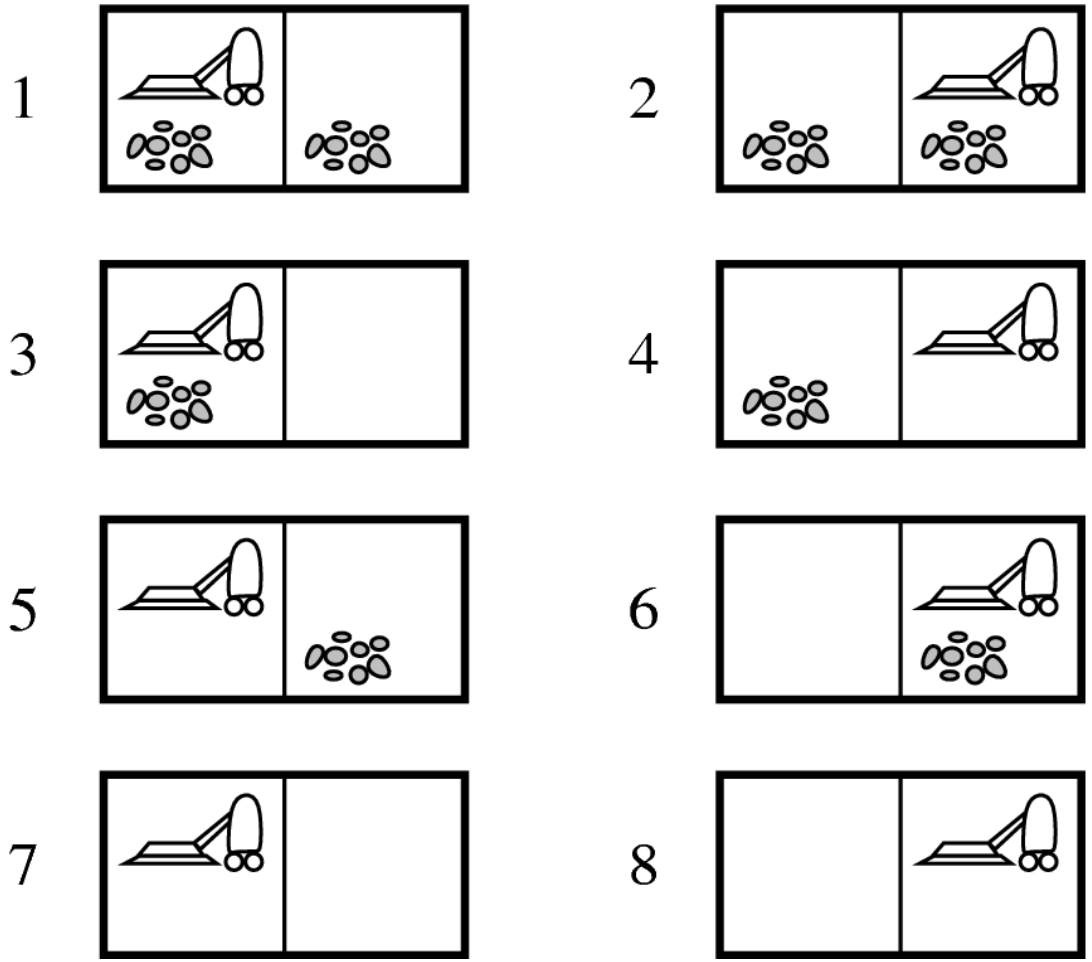
Example: Vacuum World

- Conformant Problem
  - start in any state (we can't sense)
    - $start \leftarrow \{1,2,3,4,5,6,7,8\}$
  - actions
    - e.g., *Right* goes to $\{2,4,6,8\}$
  - goal
    - no dirt
- Solution
  - [*Right, Suck, Left, Suck*]



Example: Vacuum World

- Contingency Problem
  - start in #5
  - indeterministic actions
    - *Suck* can dirty a clean carpet
  - sensing
    - dirt at current location?
  - goal
    - no dirt
- Solution
  - [*Right,* **if** *dirt* **then** *Suck*]



Single-state Problem Formulation

A problem is defined by four items:

- initial state
  - e.g., "at Arad"
- description of actions and their effects
  - typically as a successor function that maps a state s to a set $S(s)$ of action-state pairs
  - e.g., $S(\text{„at Arad“}) = \{\text{<„goto Zerind“, „at Zerind“>}, \ldots\}$
- goal test, can be
  - explicit, e.g., $s =$ "at Bucharest"
  - implicit, e.g., $\text{Checkmate}(s)$, $\text{NoDirt}(s)$
- path cost (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - $c(s_1, a, s_2)$ are the costs for one step (one action),
  - assumed to be ≥ 0

Single-State Problems

Yes

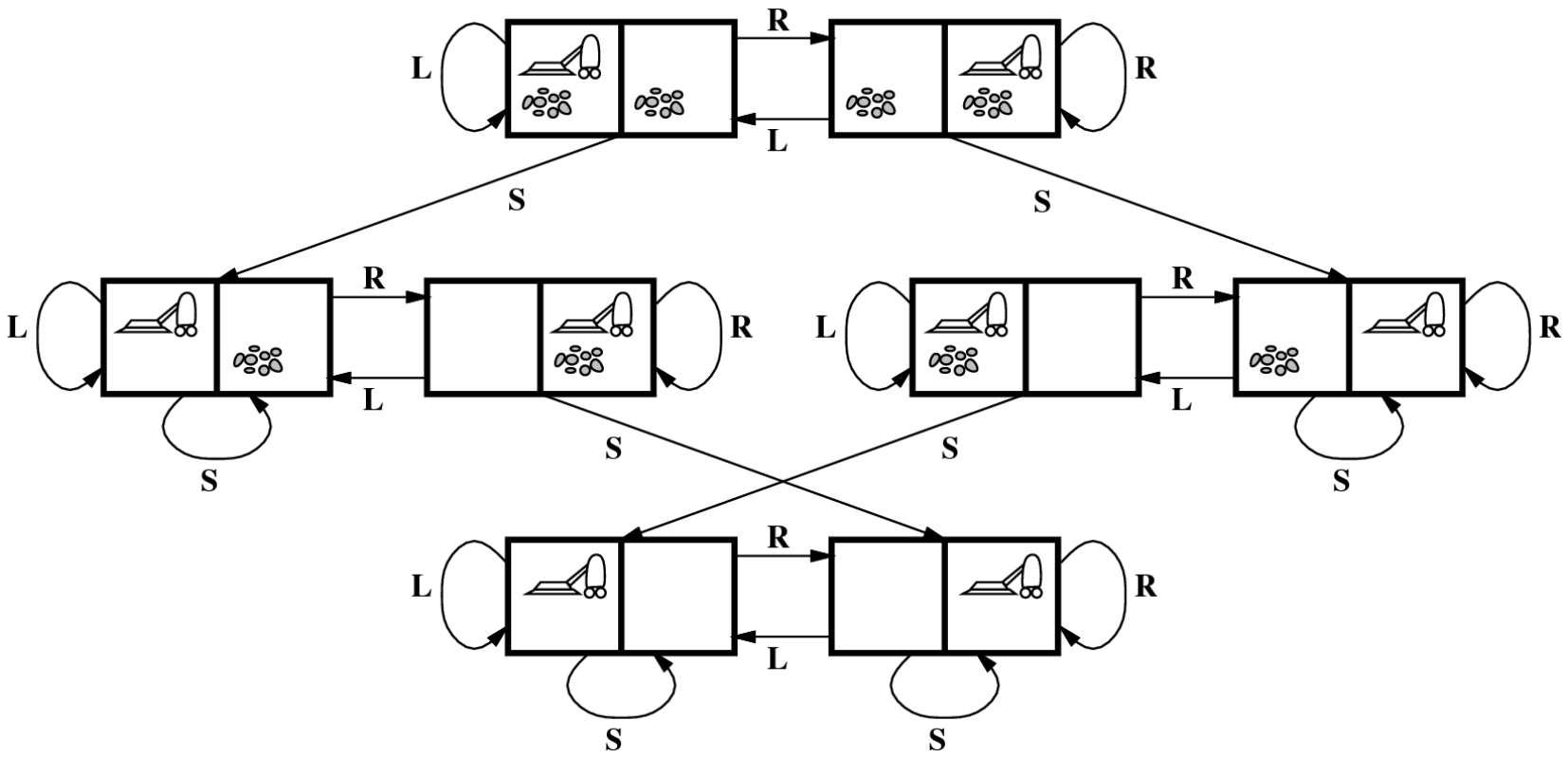
- 8-queens puzzle
- 8-puzzle
- Towers of Hanoi
- Cross-Word puzzles
- Sudoku
- Chess, Bridge, Scrabble puzzles
- Rubik's cube
- Sobokan
- Traveling Salesman Problem

No

- Tetris
  - dynamic not static
- Solitaire
  - only partially observable

State Space of a Problem

- State Space
  - the set of all states reachable from the initial state
  - implicitly defined by the initial state and the successor function



State Space of a Problem

- State Space
  - the set of all states reachable from the initial state
  - implicitly defined by the initial state and the successor function
- Path
  - a sequence of states connected by a sequence of actions
- Solution
  - a path that leads from the initial state to a goal state
- Optimal Solution
  - solution with the minimum path cost

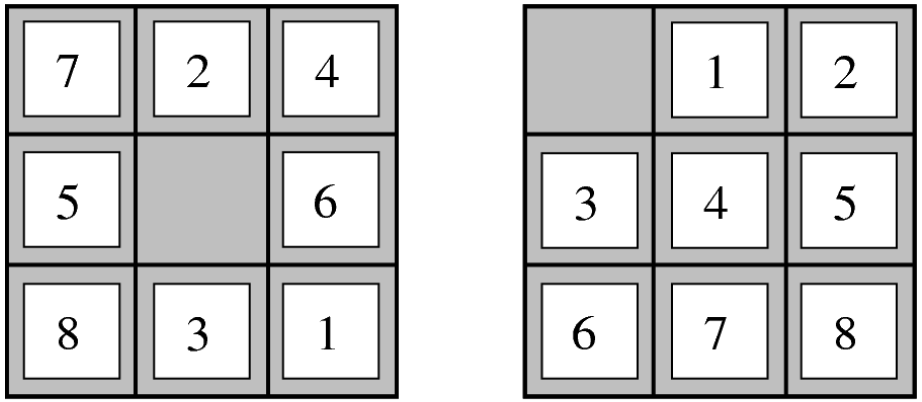
Selecting a State Space

Real world is absurdly complex

→ state space must be abstracted for problem solving

- (Abstract) state
  - corresponds to a set of real states
- (Abstract) action
  - corresponds to a complex combination of real actions
  - e.g., "go from Arad to Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - for guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
  - each abstract action should be "easier" than the original problem
- (Abstract) solution
  - corresponds to a set of real paths that are solutions in the real world

Example: The 8-puzzle

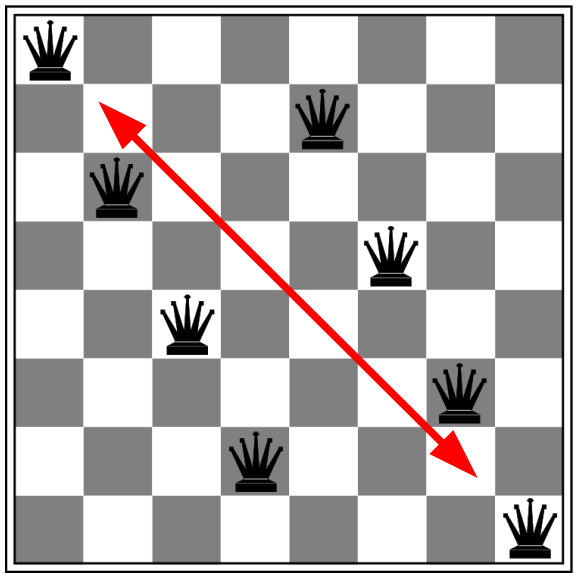


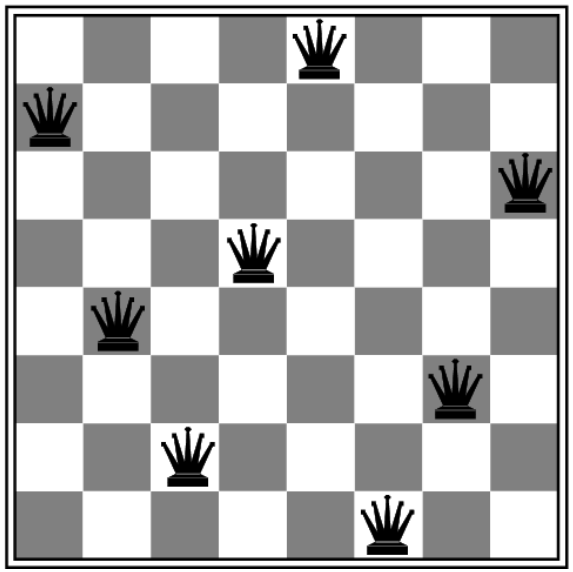
Start State Goal State

- states?
  - location of tiles
    - ignore intermediate positions during sliding
- goal test?
  - situation corresponds to goal state
- path cost?
  - number of steps in path (each step costs 1)
- actions?
  - move blank tile (left, right, up, down)
    - easier than having separate moves for each tile
    - ignore actions like unjamming slides if they get stuck

Example: The 8-Queens Problem

conflict





no conflict

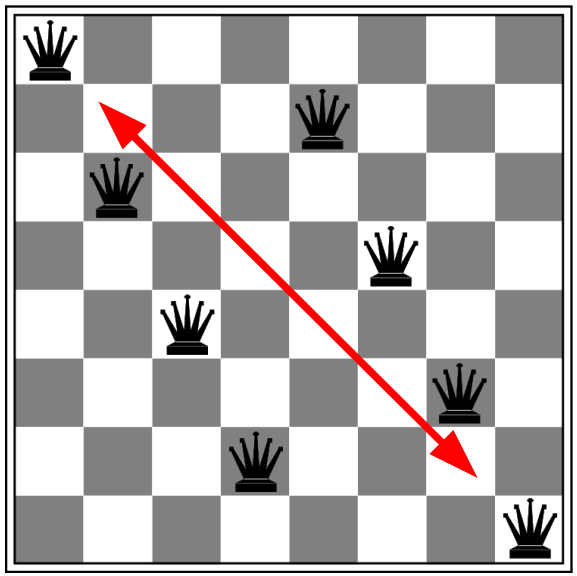
- states?
  - any configuration of 8 queens on the board
- goal test?
  - no pair of queens can capture each other
- actions?
  - move one of the queens to another square
- path cost?
  - not of interest here

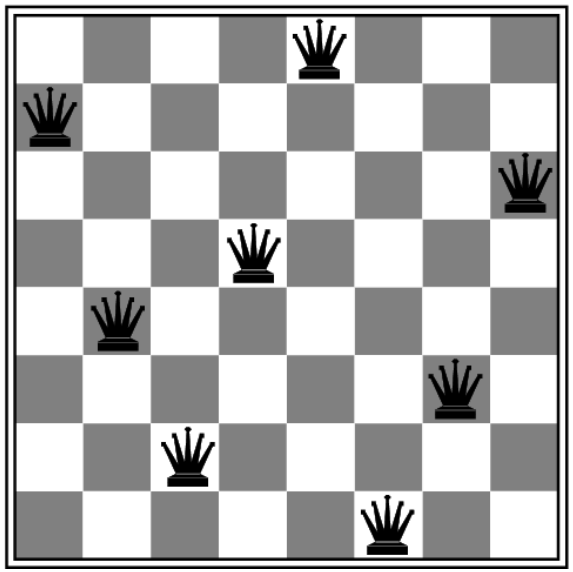
inefficient complete-state formulation

$\rightarrow 64 \cdot 63 \cdot ... \cdot 57 \approx 3 \cdot 10^{14}$ states

Example: The 8-Queens Problem

conflict





no conflict

- states?
  - n non-attacking queens in the left n columns
- goal test?
  - no pair of queens can capture each other
- actions?
  - add queen in column $n + 1$
  - without attacking the others
- path cost?
  - not of interest here

more efficient incremental formulation
→ only 2057 states

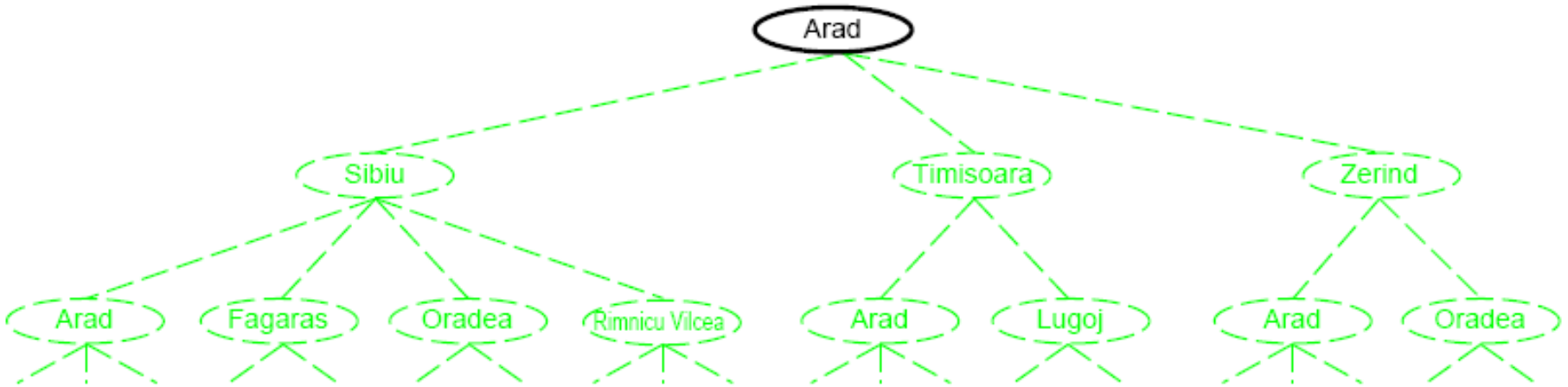
Tree Search Algorithms

- Treat the state-space graph as a tree
- Expanding a node
  - offline, simulated exploration of state space by generating successors of already-explored states (successor function)
- Search strategy
  - determines which node is expanded next
- General algorithm:

```
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
    end
```

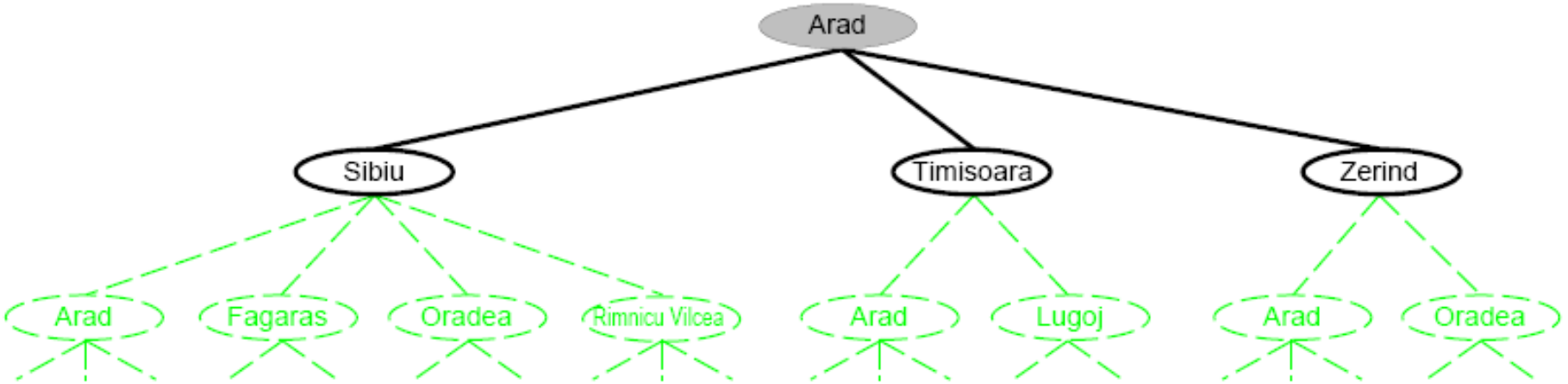
Tree Search Example

- Initial state: start with node Arad



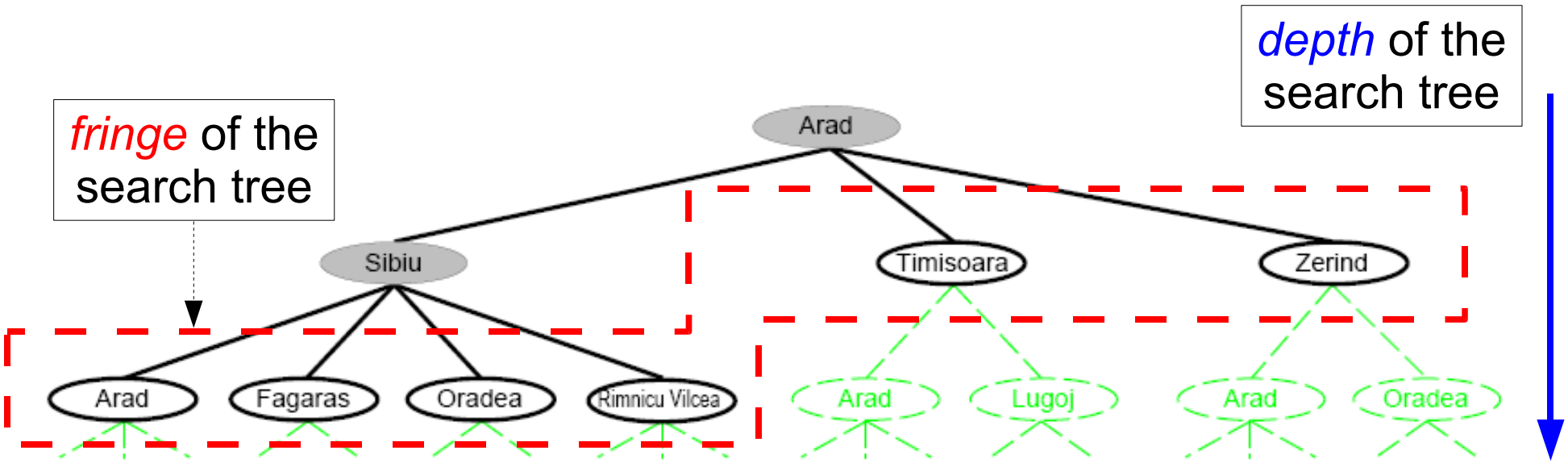
Tree Search Example

- Initial state: start with node Arad
- expand node Arad



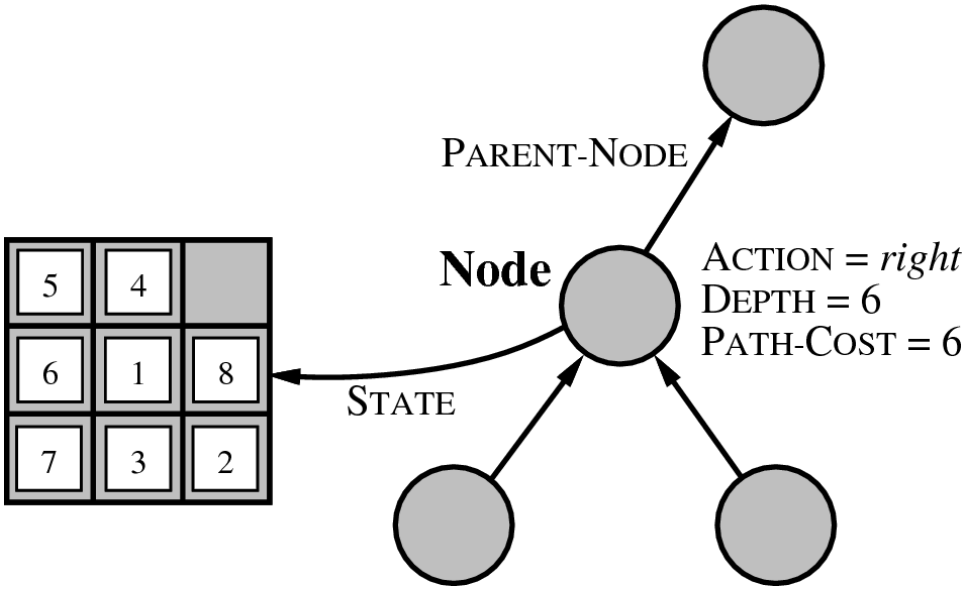
Tree Search Example

- Initial state: start with node Arad
- expand node Arad
- expand node Sibiu



States vs. Nodes

- State
  - (representation of) a physical configuration
- Node
  - data structure constituting part of a search tree
  - includes
    - state
    - parent node
    - action
    - path cost *g(x)*
    - depth
- Expand
  - creates new nodes
  - fills in the various fields
  - uses the successor function to create the corresponding states



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 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
```

Search Strategies

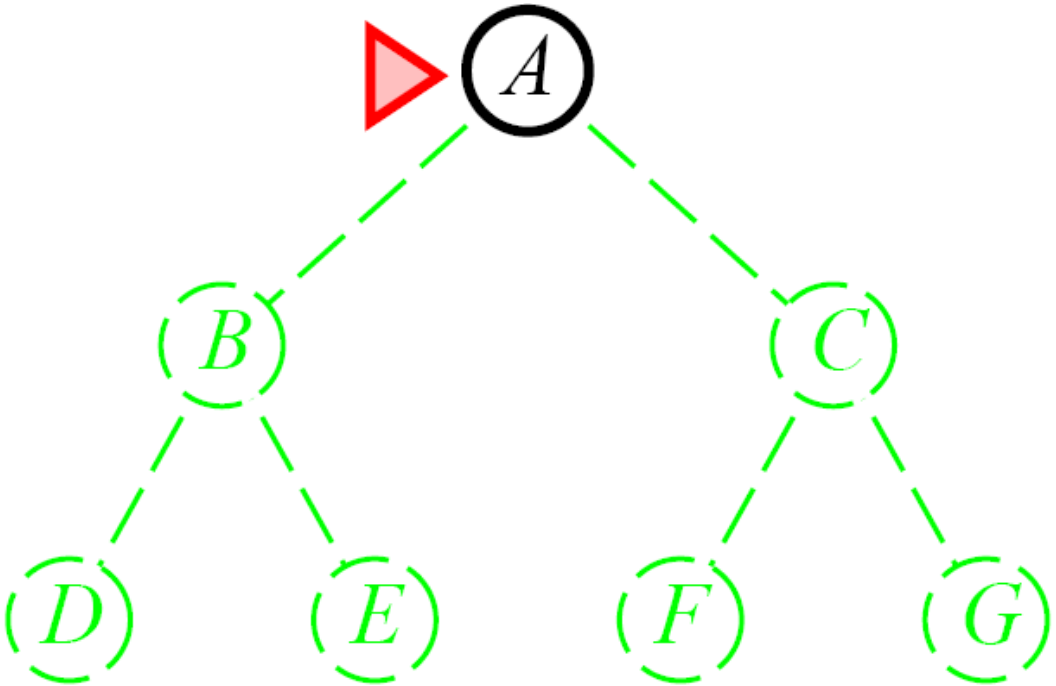
- A search strategy is defined by picking the order of node expansion
  - implementation in a queue
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number 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 ∞)

Search Strategies

- Uninformed (blind) search strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
- Informed (heuristic) search strategies have knowledge that allows to guide the search to promising regions
  - Greedy Search
  - A* Best-First Search

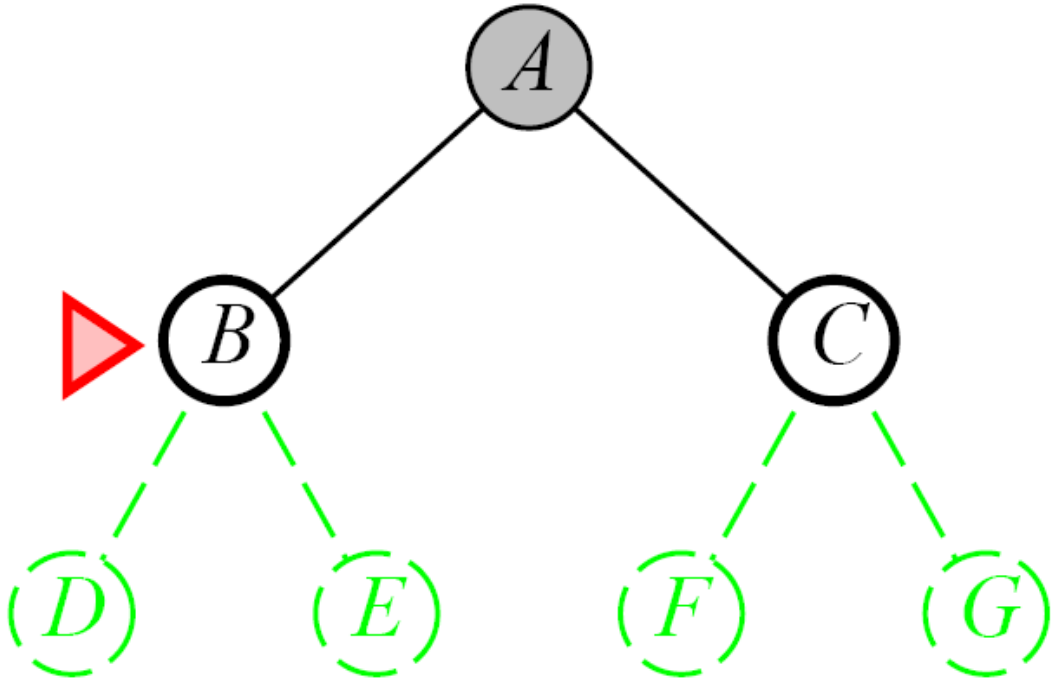
Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)
- Implemetation:
  - expand the shallowest unexpanded node
  - fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)



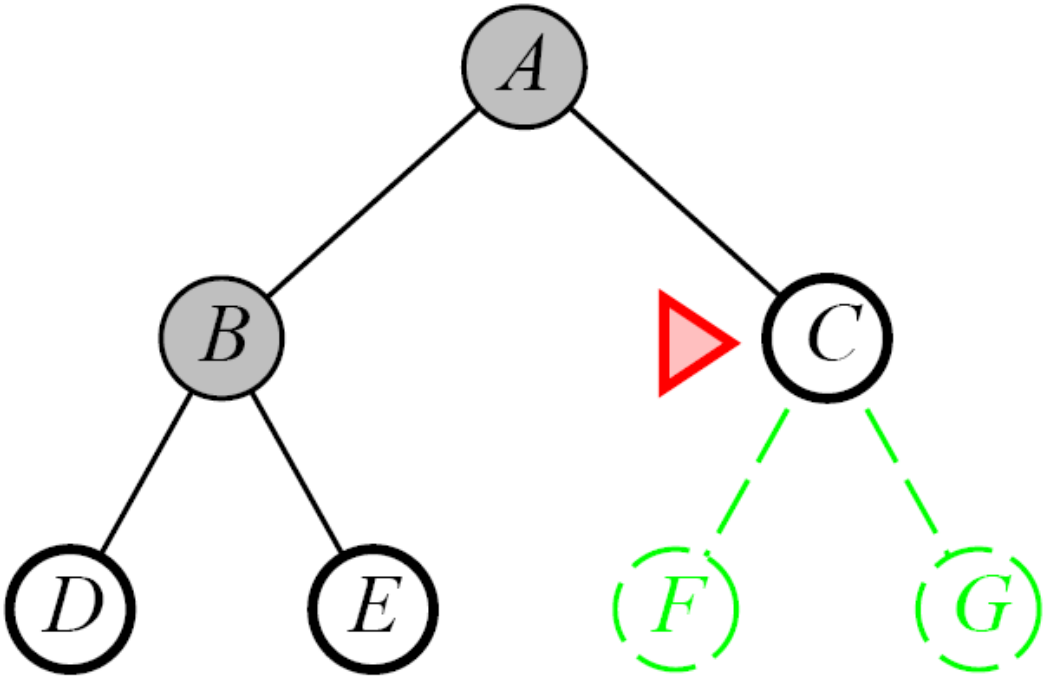
Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)
- Implemetation:
  - expand the shallowest unexpanded node
  - fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)



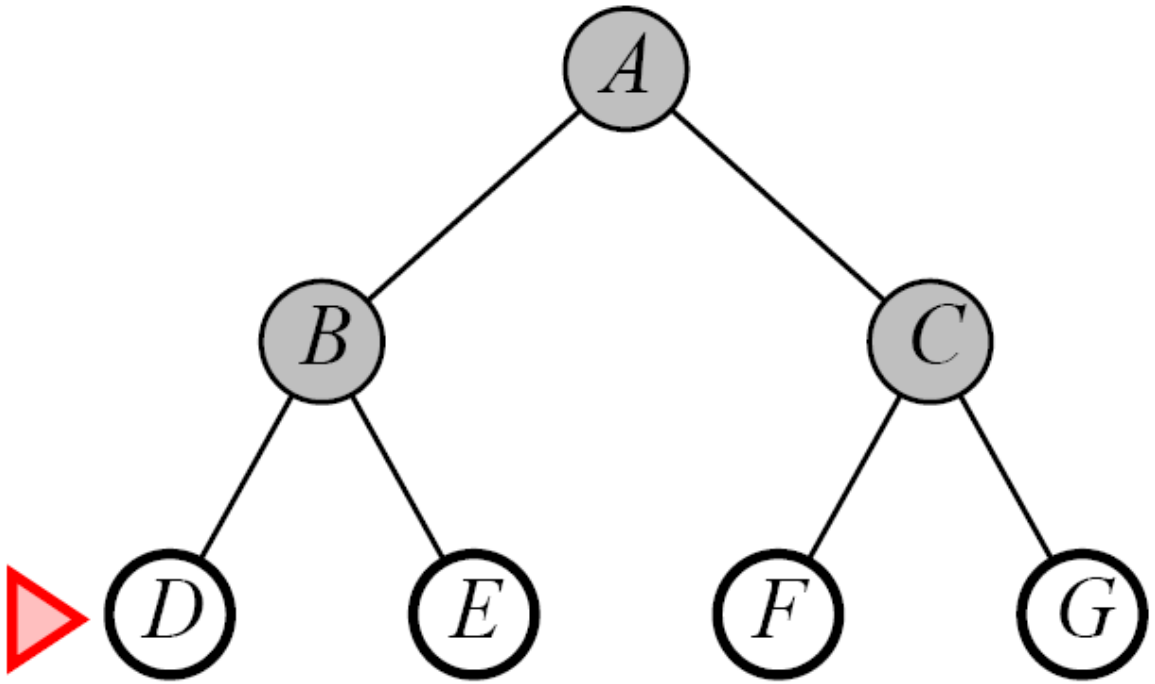
Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)
- Implemetation:
  - expand the shallowest unexpanded node
  - fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)



Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)
- Implemetation:
  - expand the shallowest unexpanded node
  - fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)



Properties of Breadth-First Search

- Completeness
  - Yes (if b is finite)
- Time Complexity
  - each depth has b times as many nodes as the previous
  - each node is expanded
  - except the goal node in level d
    - worst case: goal is last node in this level

$\Rightarrow 1+b+b^2+b^3+...+b^d+(b^{(d+1)}-b)=O(b^{d+1})$

- Space Complexity
  - every node must remain in memory
    - it is either a fringe node or an ancestor of a fringe node

$\Rightarrow O(b^{d+1})$

- Optimality
  - Yes, for uniform costs (e.g., if cost = 1 per step)

Combinatorial Explosion

- Breadth-first search
  - branching factor $b = 10$, 10,000 nodes/sec, 1000 bytes/node

Depth	Nodes	Time	Memory
2	1100	.11 secs	1 MB
4	111 100	11 secs	106 MB
6	10^7	19 mins	10 GB
8	10^9	31 hours	1 TB
10	10^{11}	129 days	101 TB
12	10^{13}	35 years	10 PetaBytes
14	10^{15}	3523 years	1 ExaByte

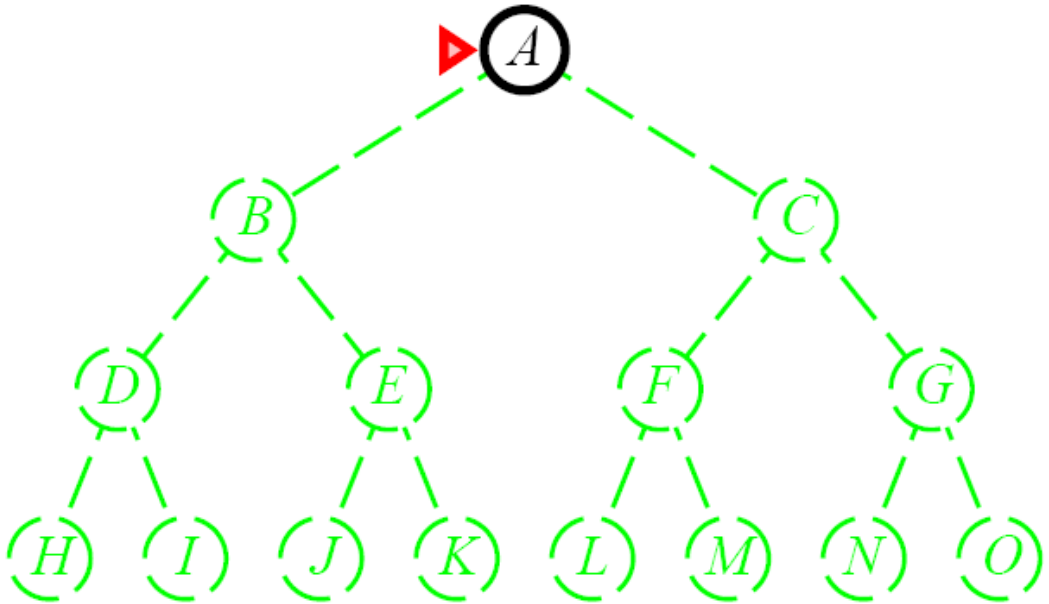
  - Space is the bigger problem
    - can easily generate nodes at 100MB/sec ⇒ 24hrs = 8640 GB

Uniform-Cost Search

- Breadth-first search can be generalized to cost functions
  - each node now has associated costs
  - costs accumulate over path
  - instead of expanding the shallowest path, expand the least-cost unexpanded node
  - breadth-first is special case where all costs are equal
- Implementation
  - fringe = queue ordered by path cost
- Completeness
  - yes, if each step has a positive cost (cost ≥ ε)
  - otherwise infinite loops are possible
- Space and Time complexity $b^{1+O(\lfloor C^*/\epsilon \rfloor)}$
  - number of nodes with costs < costs of optimal solution
- Optimality
  - Yes – nodes expanded in increasing order of path costs

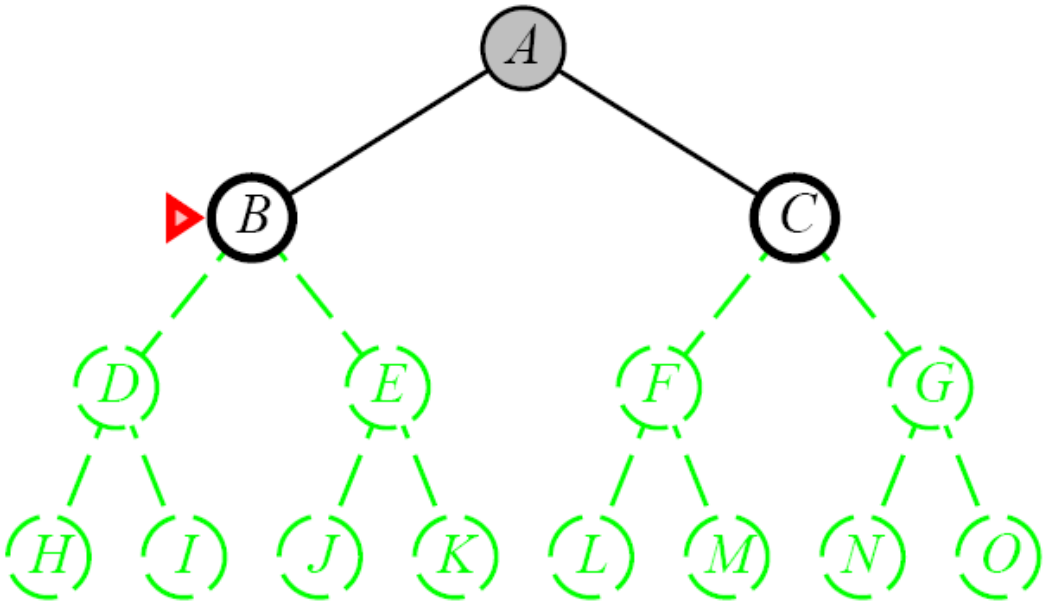
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



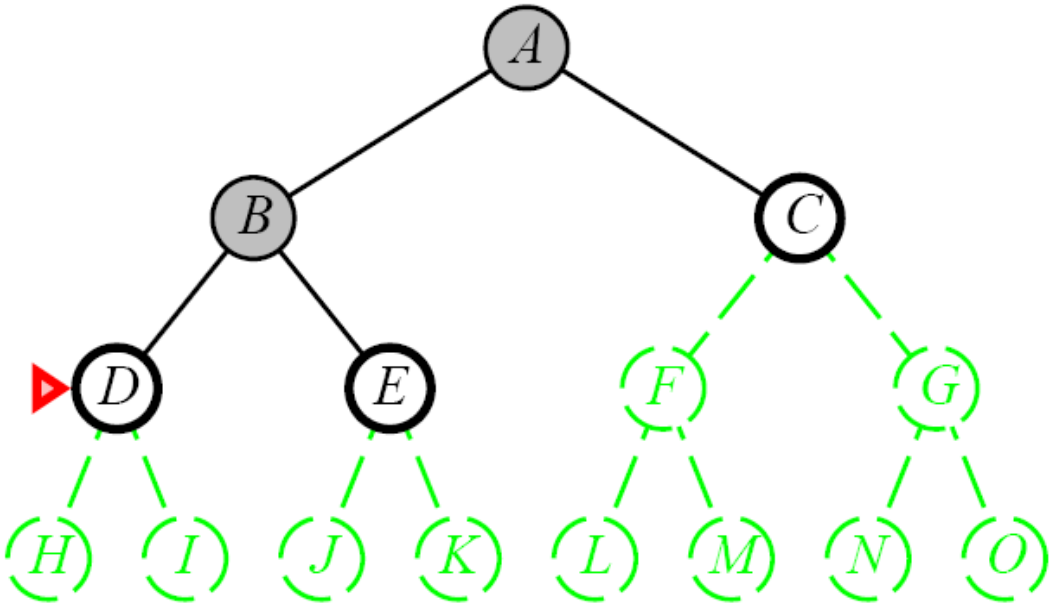
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



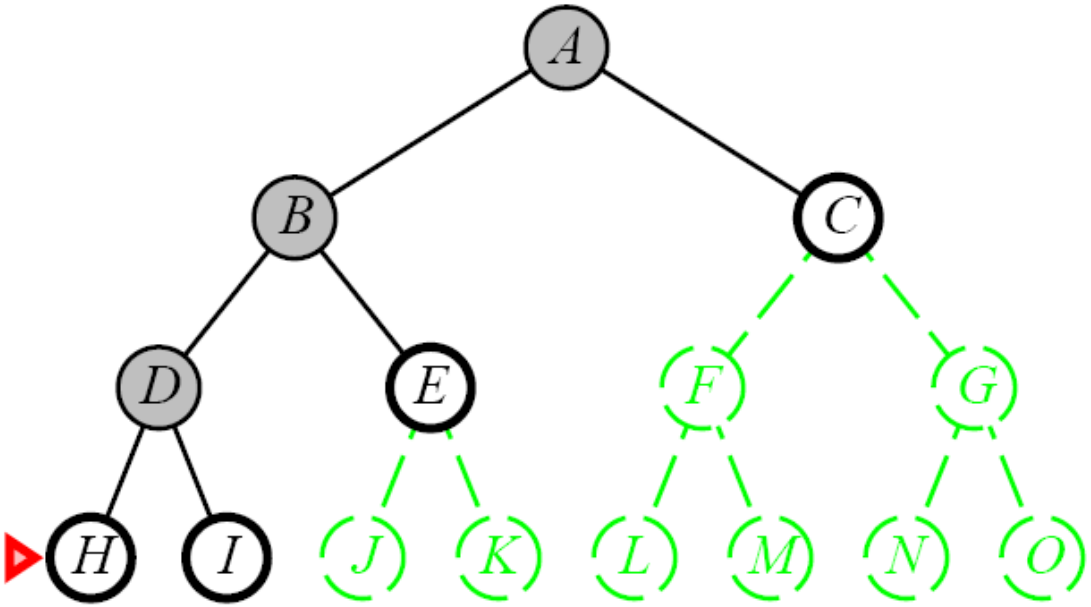
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



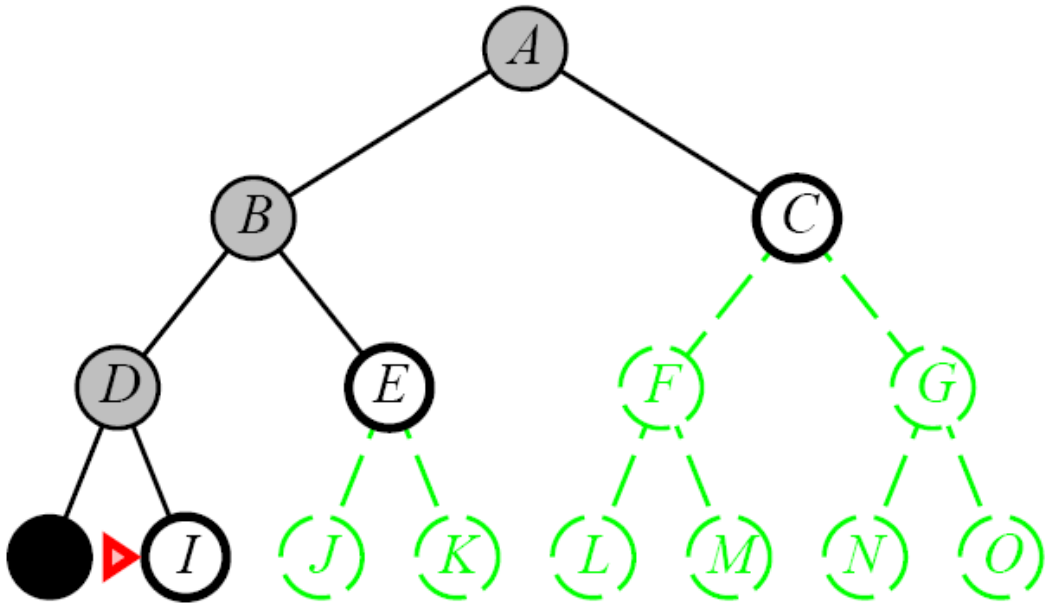
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



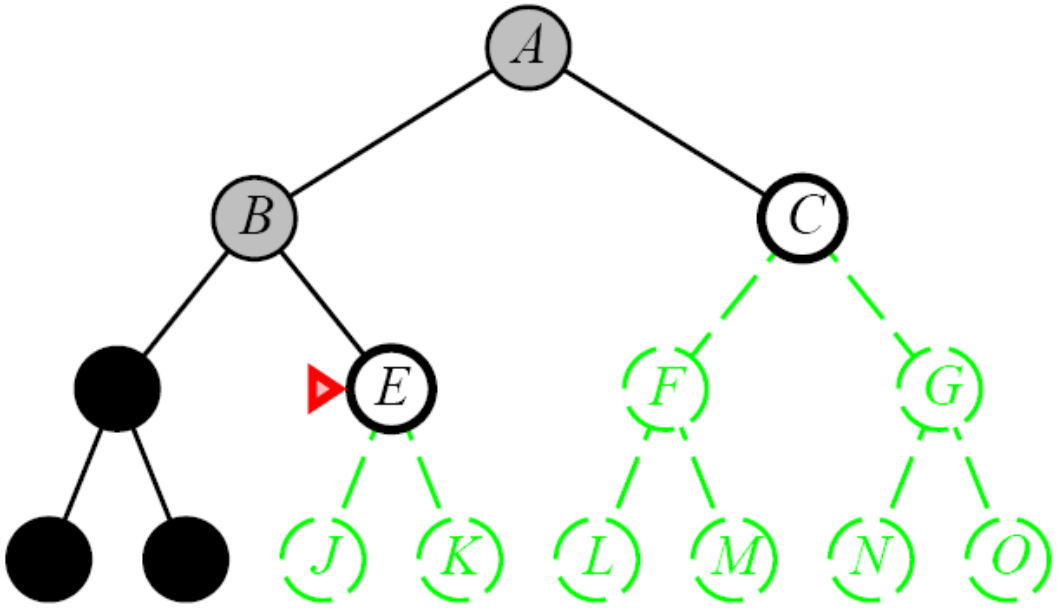
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



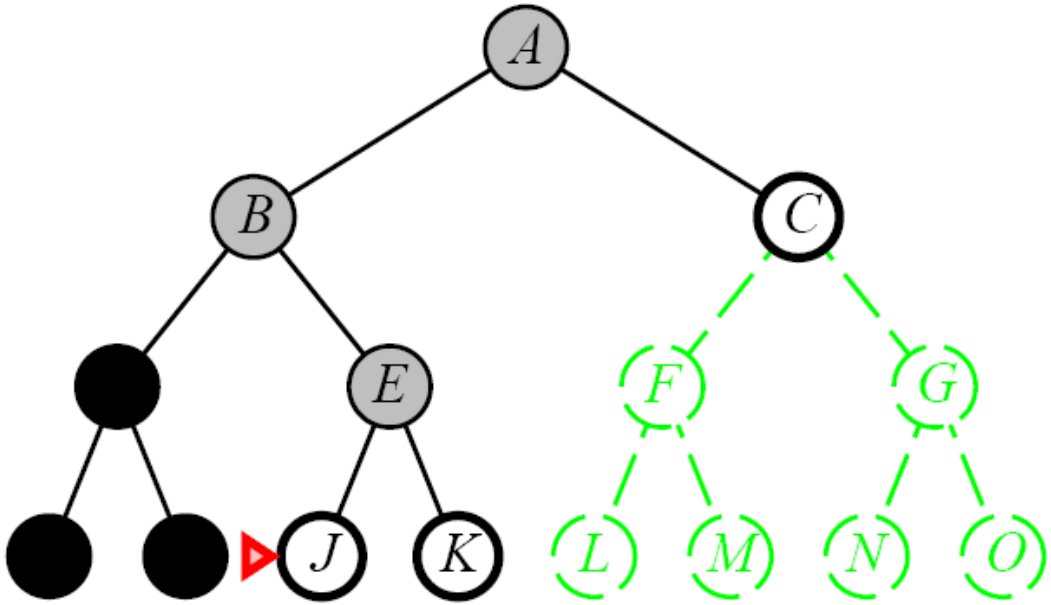
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



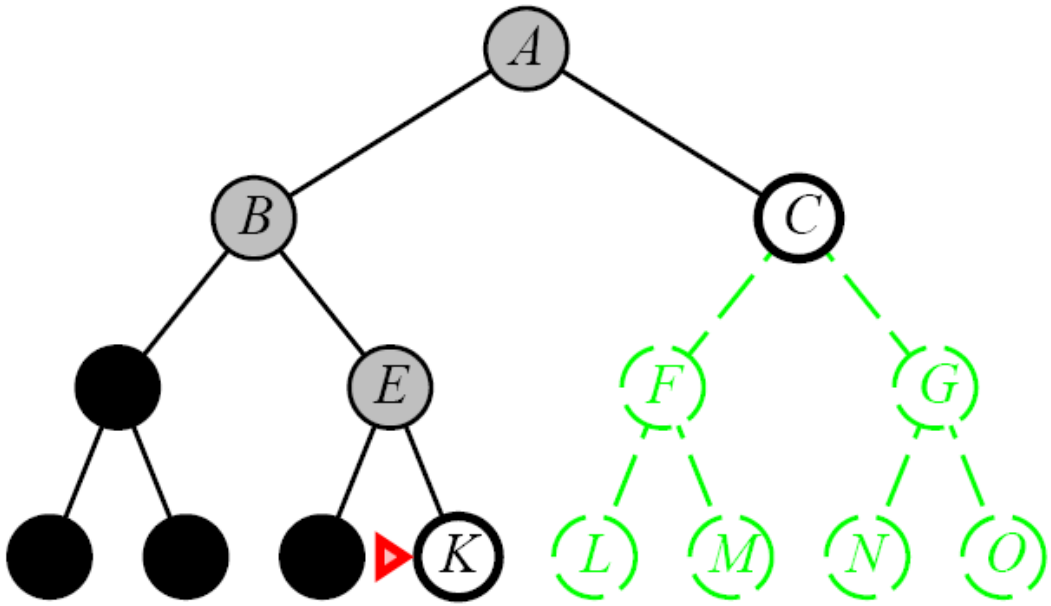
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



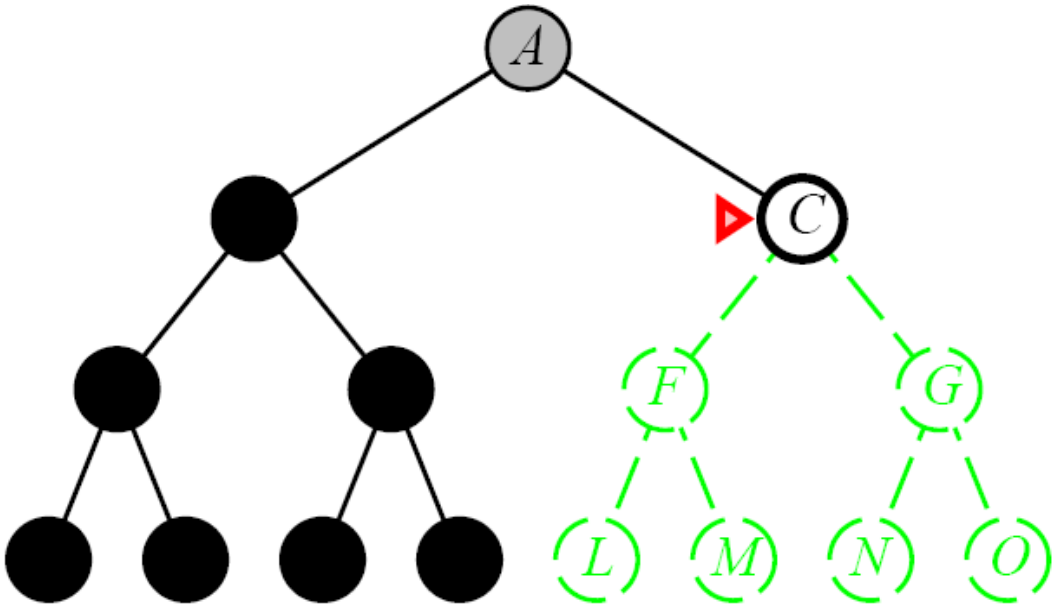
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



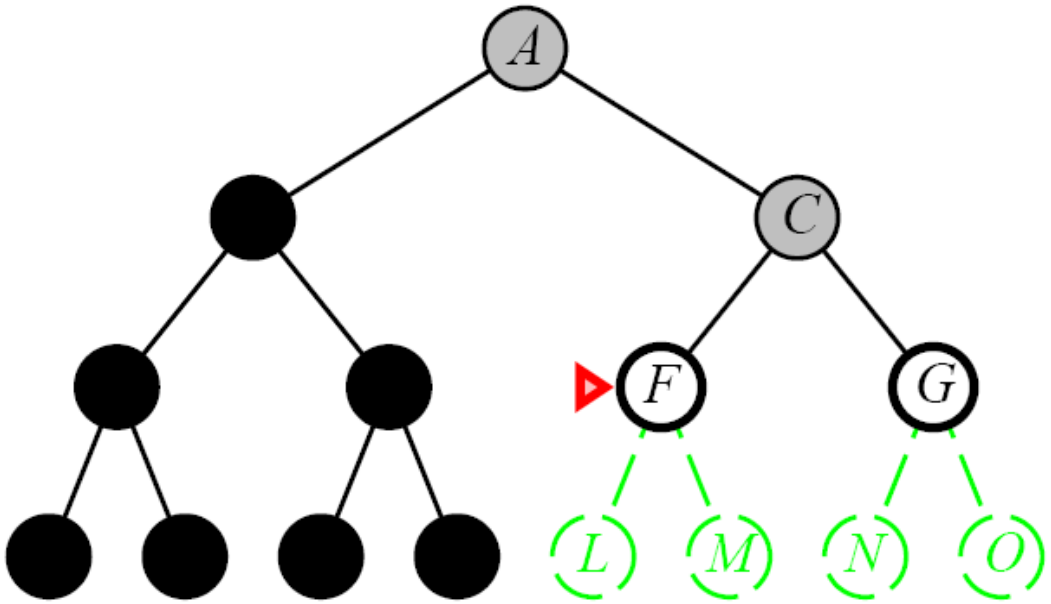
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



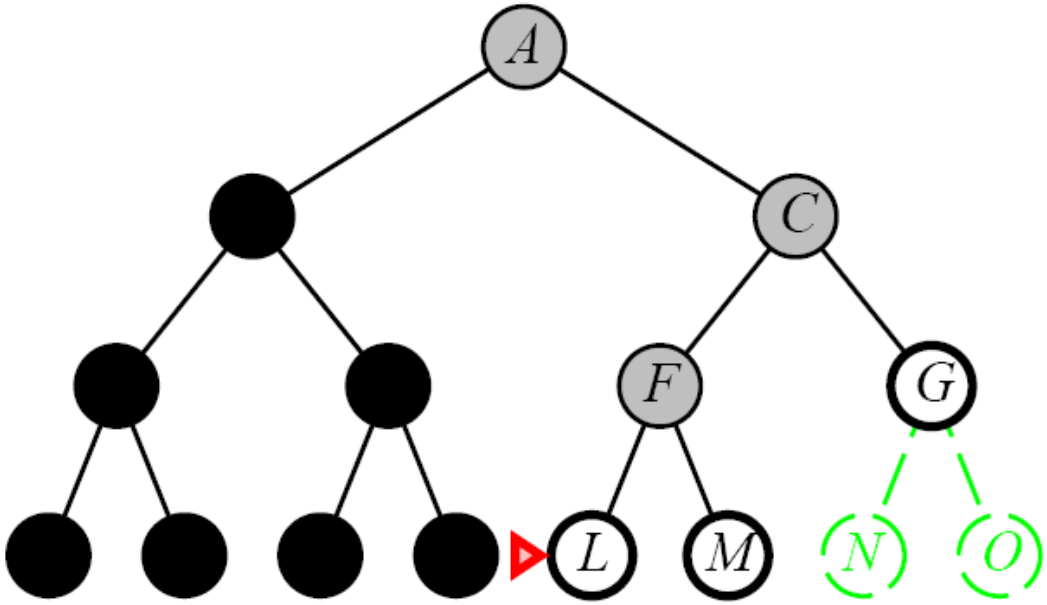
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



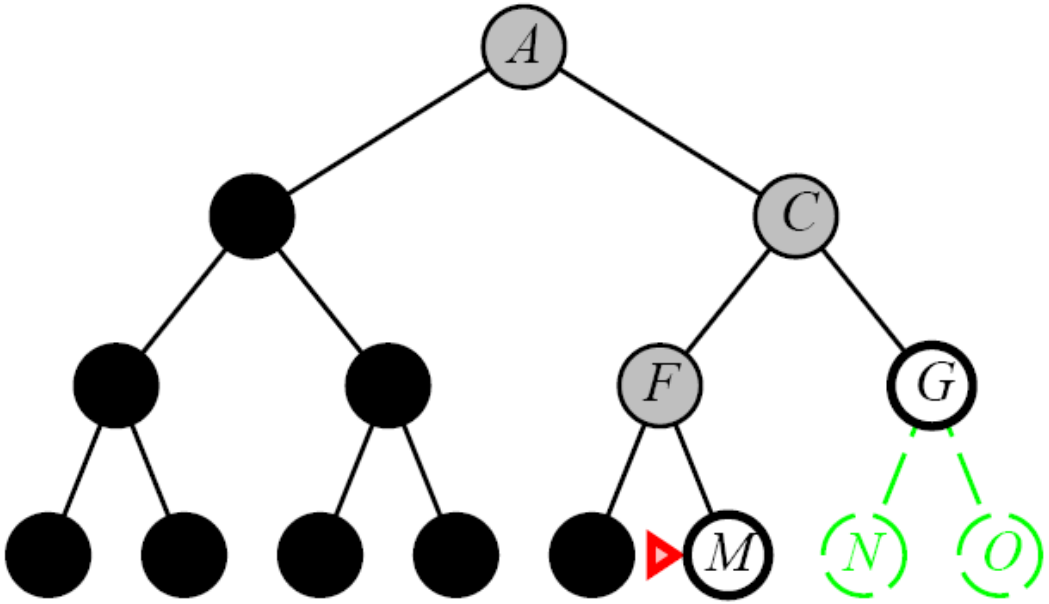
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implemetation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)



Properties of Depth-First Search

- Completeness
  - No, fails in infinite-depth search spaces and spaces with loops
  - complete in finite spaces if modified so that repeated states are avoided
- Time Complexity
  - has to explore each branch until maximum depth $m \Rightarrow O(b^m)$
  - terrible if $m > b$
  - but may be faster than breadth-first if solutions are dense
- Space Complexity
  - only nodes in current path and their unexpanded siblings need to be stored

  $\Rightarrow$ only linear complexity $O(m \cdot b)$
- Optimality
  - No, longer (more expensive) solutions may be found before shorter (cheaper) ones

Backtracking Search

Even more space-efficient variant

- does not store all expanded nodes, but only the current path

  $\Rightarrow O(m)$

  - if no further expansion is possible, go back to the predecessor
  - each node is able to generate the *next* successor
- only needs to store and modify one state
  - actions can do and undo changes on this one state

Depth-limited Search

- depth-first search is provided with a depth limit l
  - nodes with depths $d > l$ are not considered → incomplete
  - if $d < l$ it is not optimal (like depth-first search)
  - time complexity $O(b^l)$, space complexity $O(bl)$

```
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

- Main problem with depth-limited search is setting of l
- Simple solution:
  - try all possible $l = 0, 1, 2, 3, \ldots$

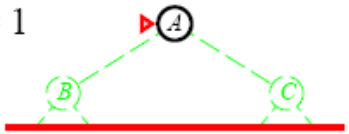
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

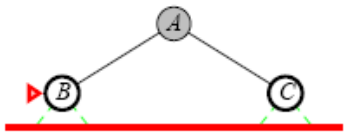
  - costs are dominated by the last iteration, thus the overhead is marginal

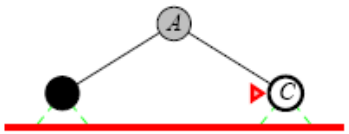
Iterative Deepening Search

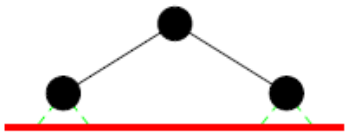
Limit = 0

Limit = 1

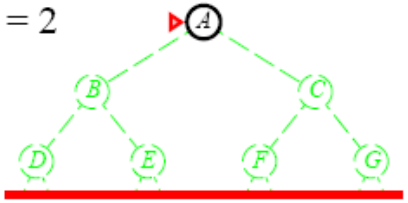


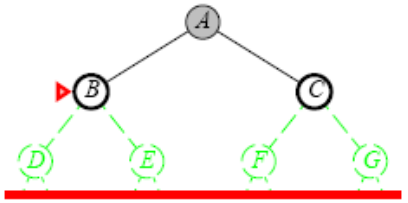


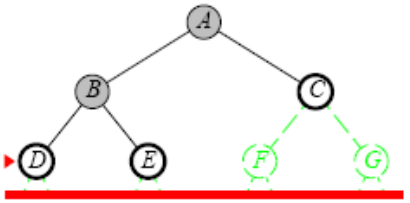


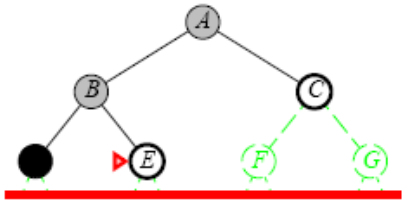


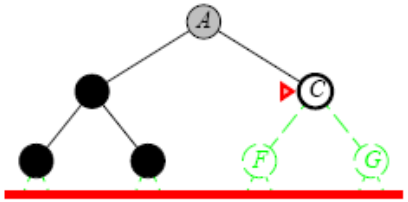
Limit = 2

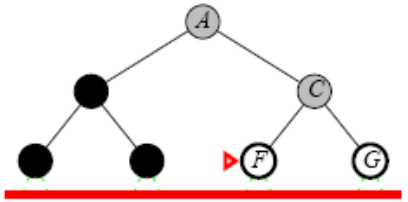


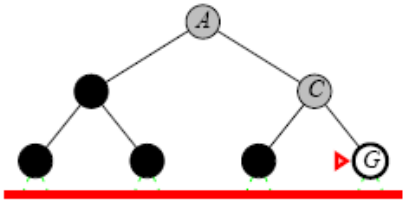


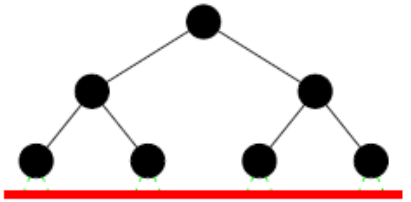




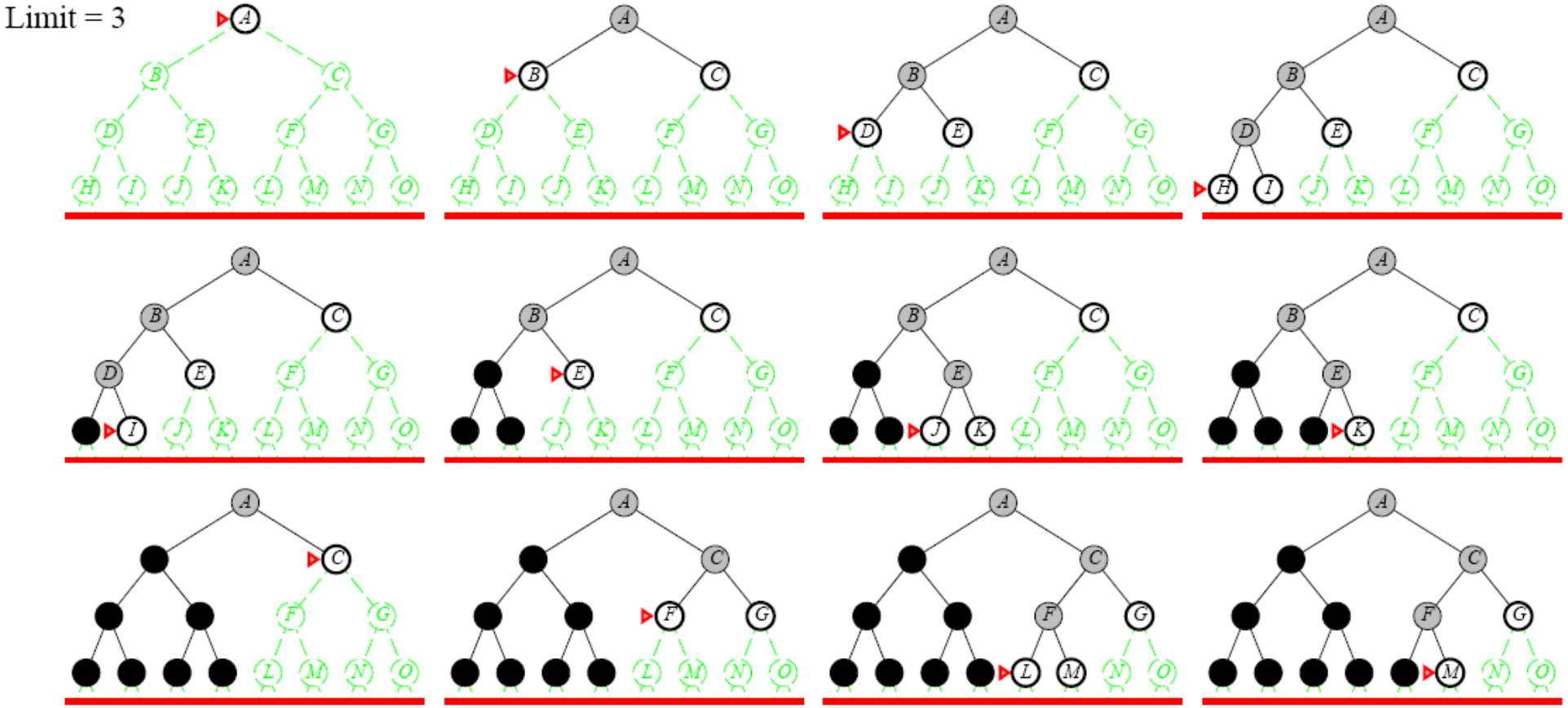








Iterative Deepening Search



Properties of Iterative Deepening Search

- Completeness
  - Yes (no infinite paths)
- Time Complexity
  - first level has to be searched d times
  - last level has to be searched once

$$\Rightarrow d \cdot b + (d-1)b^2 + \ldots + 1 \cdot b^d = \sum_{i=1}^{d} (d-i+1) \cdot b^i$$

- Space Complexity

  ⇒ only linear complexity $O(bd)$
- Optimality
  - Yes, the solution is found at the minimum depth

⇒ combines advantages of depth-first and breadth-first search

Comparison of Time Complexities

Worst-case (goal is in right-most node at level d)

- Breadth-First Search

$$N_{BFS} = b + b^2 + \ldots + b^d + b^{d+1} - (b) = b^{d+1} - b + \sum_{i=1}^{d} b^i$$

- Depth-Limited Search

$$N_{DLS} = b + b^2 + \ldots + b^d = \sum_{i=1}^{d} b^i$$

- Iterative Deepening

$$N_{IDS} = d \cdot b + (d-1) b^2 + \ldots + 1 \cdot b^d = \sum_{i=1}^{d} (d - i + 1) \cdot b^i$$

Example: $b = 10, d = 5$

BFS expands nodes at depth d+1

$$N_{BFS} = 10 + 100 + 1000 + 10{,}000 + 100{,}000 + 999{,}990 = 1{,}111{,}100$$

$$N_{DLS} = 10 + 100 + 1000 + 10{,}000 + 100{,}000 = 111{,}110$$

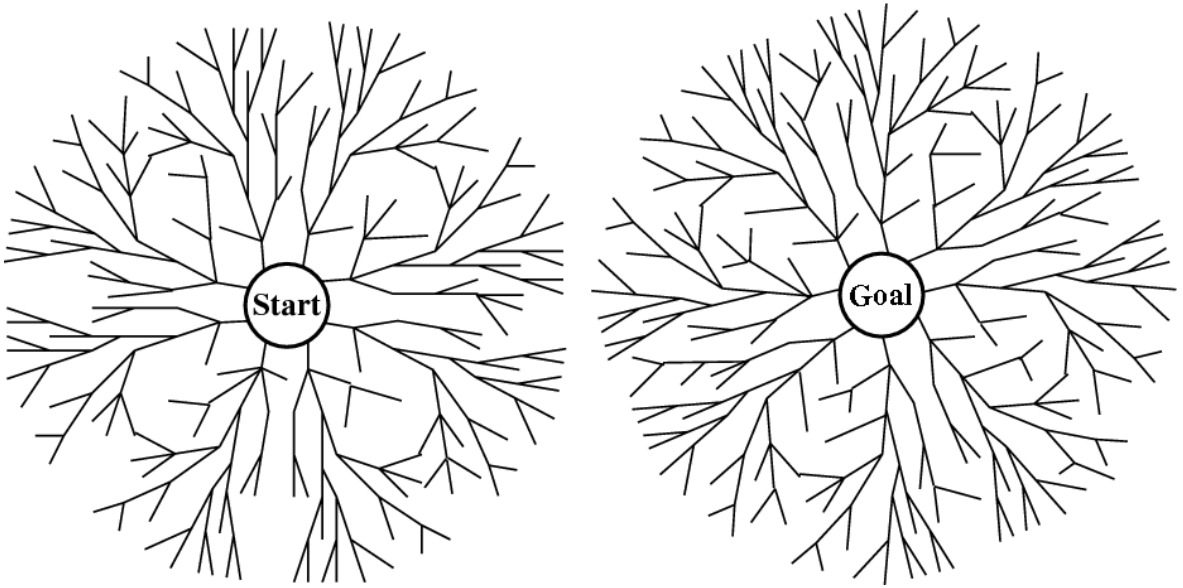
$$N_{IDS} = 50 + 400 + 3000 + 20{,}000 + 100{,}000 = 123{,}450$$

Overhead of IDS only ca. 10%

Bidirectional Search

- Perform two searches simultaneously
  - forward starting with initial state
  - backward starting with goal state

  check whether generated node is in fringe of the other search



- Properties
  - reduction in complexity $(b^{d/2} + b^{d/2} \ll b^d)$
  - only possible if actions can be reversed
  - search paths may not meet for depth-first bidirectional search

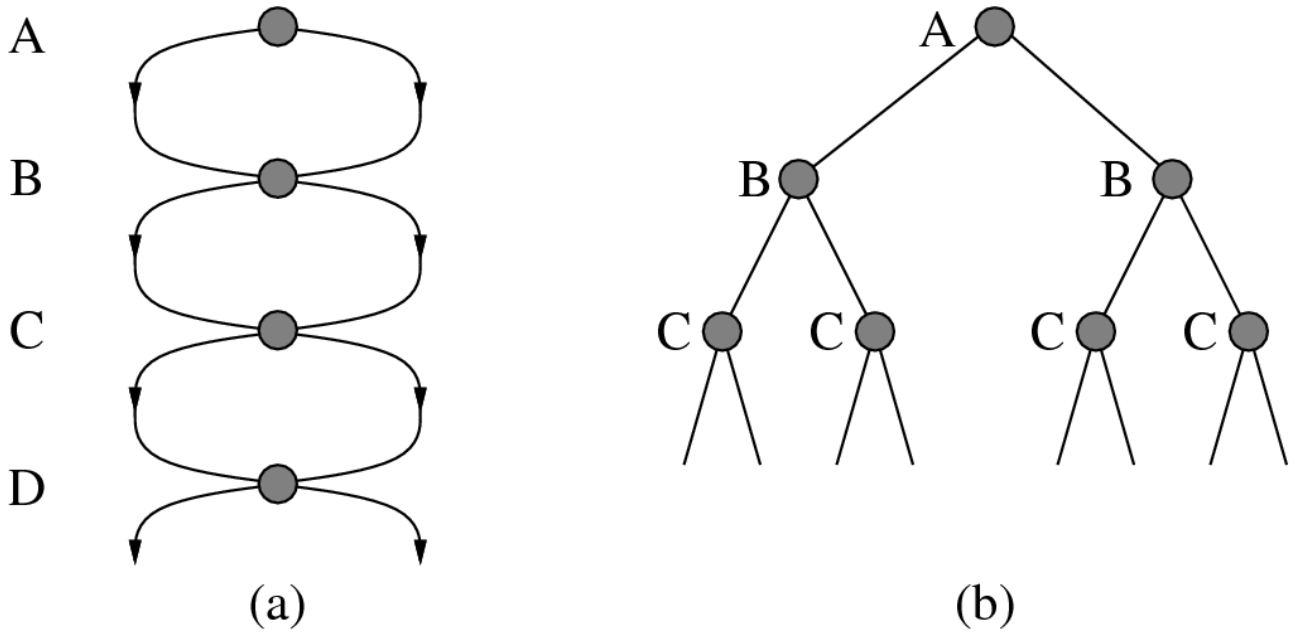
Repeated States

- Failure to detect repeated states can turn a linear problem into an exponential one!

Ribbon Example

- two connections from each state to the next

d states but state space is 2^d

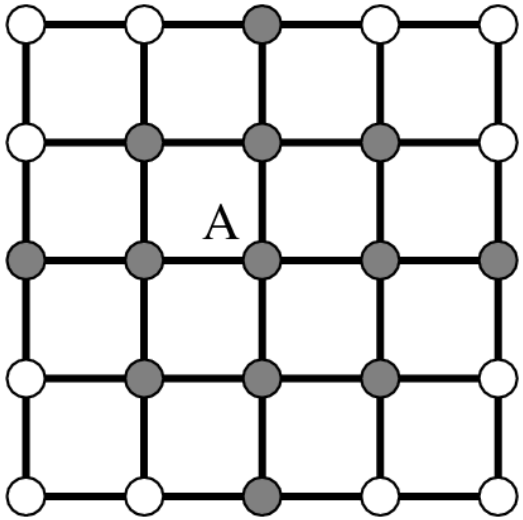


Repeated States

- Failure to detect repeated states can turn a linear problem into an exponential one!

(more realistic) Grid Example

- each square on grid has 4 neighboring states in
- thus, game tree w/o repetitions has 4^d nodes
- but only about $2d^2$ different states are reachable in d steps



Graph Search

- remembers the states that have been visited in a list called *closed*
- Note: the fringe list is often also called the *open* list

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    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 node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Summary of Algorithms

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed 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*