Artificial Intelligence

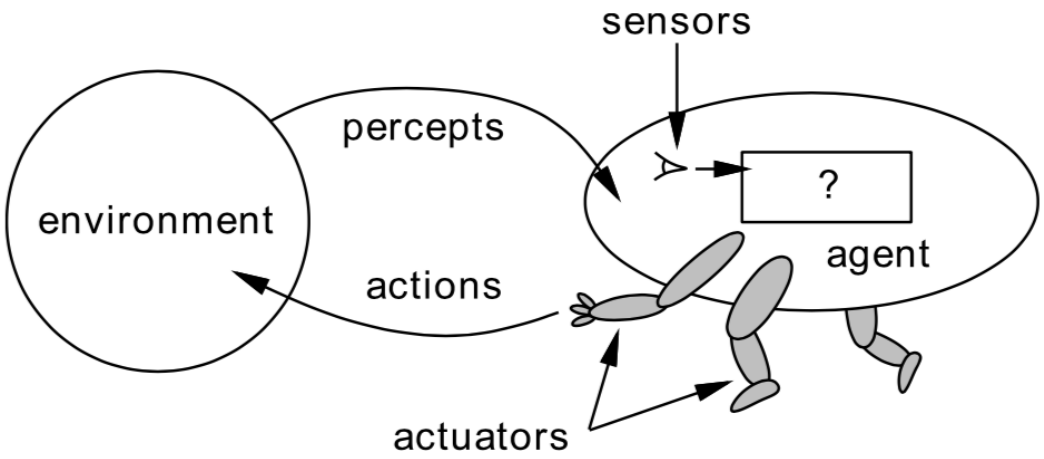
Problem solving and search *AIMA* Ch. 3

Lectures 2 and 3

Outline

- ♢ Simple problem solving agents
- ♢ Problem formulation
- ♢ Example problems
- ♢ Basic tree search algorithms
- ♢ Graph search

Rational Agents



Agent perceives its environment through **sensors** and acts upon the environment through **actuators**.

The **percept sequence** is the history of everything the agent has perceived. The **agent function** maps each percept sequence to an action.

For each percept sequence, a **rational agent** selects the best action according to a **performance measure**, given the percept sequence and the built-in knowledge of the agent.

Problem-solving Agents

A **problem-solving agent** looks for a sequence of actions leading to a desirable **goal** state.

Three separate agent design phases:

♢ **Problem formulation phase**: decide actions and states to consider.

♢ **Search phase**: finding a solution given as a sequence of actions.

♢ **Execution phase**: apply the recommended actions.

Simple problem solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent: 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)
        if seq = failure then return a null action
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Assumptions

♢ Fully observable environment

Sensors detect all aspects relevant to action choice. Initial state fully known.

♢ Deterministic environment

Effect of applying actions known precisely; no errors and no uncertainty (solutions are executed without even paying attentiton to the percepts !)

♢ Static environment

Nothing changes while the agent is computing the response

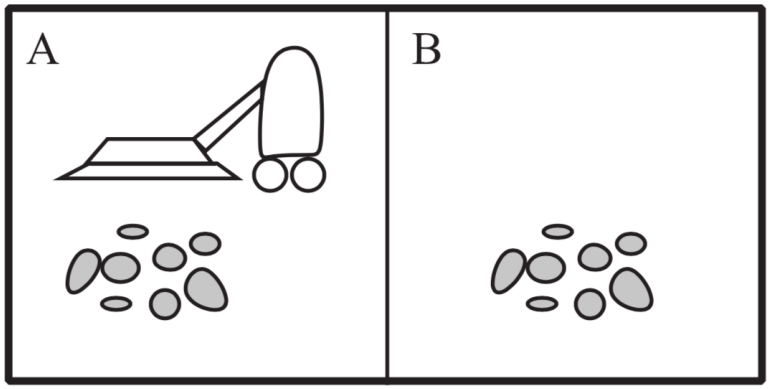
♢ Discrete envoronment

Time, environment state, and actions are discrete variables

♢ Single-agent

No interaction with other agents.

Example: Vacuum world



Fully observable: agent knows whether each square is clean or dirty.

Deterministic: e.g., the cleaning action always cleans all dirt.

Static: dirt doesn't appear out of the blue.

Discrete: Two possible positions of agent; each cell is either clean or dirty.

Single-agent: The dirt is an object rather than another agent!

Problem formulation

We can specify our problem using the following five components:

– initial state: State the agent starts in.

– actions: a function $\textsc{Actions}(s)$ gives the set of actions that can be applied to a state s

– transition model: a function $\textsc{Result}(s,a)$ gives the state that is obtained by applying action a to state s
  – the function is also called a successor function, and the result is called a successor of s

– goal test: checks whether a state is a goal

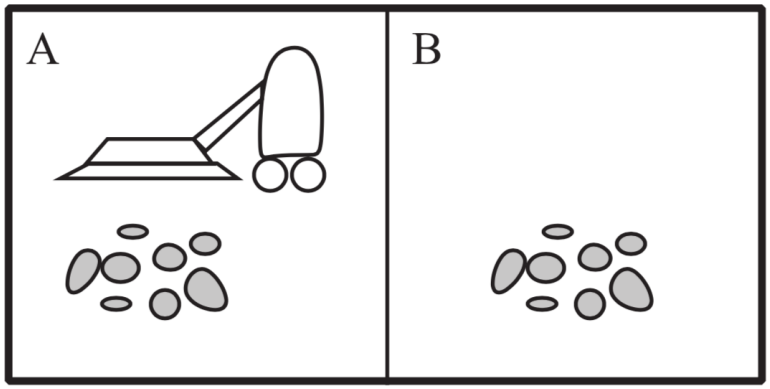
– path cost: a function that assigns a numeric value $pc(p)$ to each path p
  – paths are defined on the following slide

Path

– path: a sequence $s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n$ where
  – $s_0, s_1, s_2, \ldots, s_n$ are states,
  – $a_1, a_2, \ldots, a_n$ are actions, and
  – $a_i \in \textsc{Actions}(s_{i-1})$ and $s_i = \textsc{Result}(s_{i-1}, a_i)$ for each $1 \leq i \leq n$

$\Rightarrow$ We omit actions if each pair of states is connected by at most one action

Example: Vacuum world



– states: integer dirt and robot locations (ignore dirt amounts etc.)
– initial state: as in the picture above.
– actions: $Left$, $Right$, $Suck$, $NoOp$
– goal test: no dirt on either square
– step cost: 1 for action other than $NoOp$, 0 for $NoOp$

Additional definitions

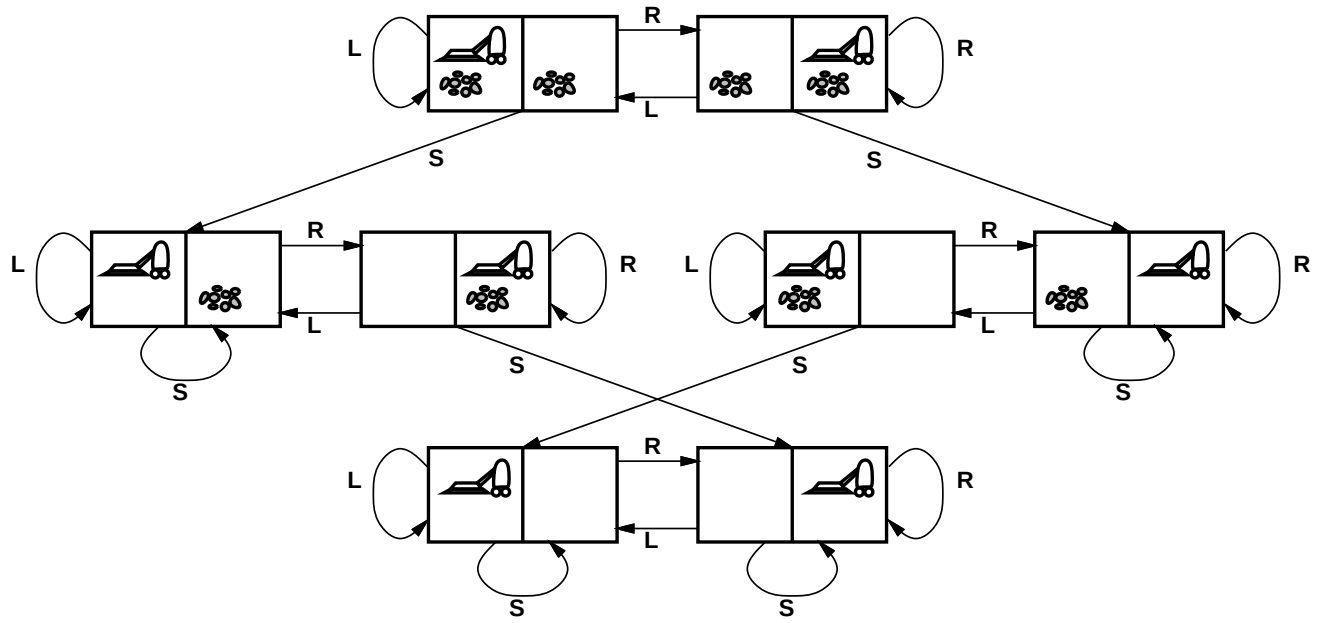
– state space: initial state plus all states reachable from the initial state
  ⇒ can be viewed as a graph where
    – states are nodes and
    – labeled edges correspond to actions

– solution: a path $s_0, a_1, s_1, \ldots, a_n, s_n$ such that
  – s_0 is the initial state, and
  – s_n is a goal state

– optimal solution: a solution whose cost is lowest among all solutions
  – does not need to be unique

These are **derived from**, and not part of problem formalization!

Example: Vacuum World



- state space: as in the picture.

- optimal solution: suck dirt, move right, then such dirt again.

Example: Word Ladder

Change word w_1 into w_2 by mutating one letter at a time such that each intermediate word is an English word

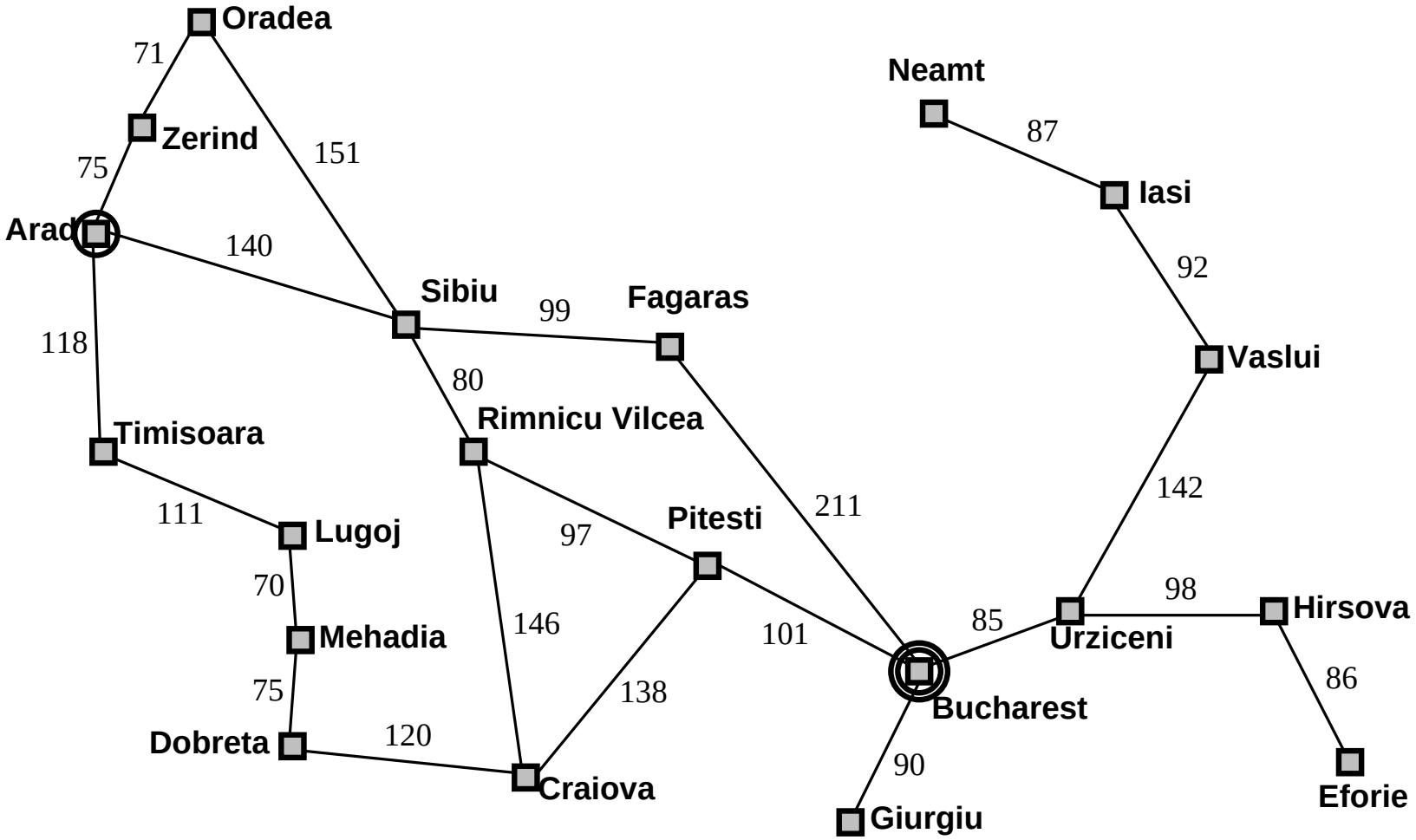
- e.g., FOOD $\rightarrow$ MOOD $\rightarrow$ MOOR $\rightarrow$ POOR

We can specify the problem like this:

- states: all English words
- actions: the set of all integers–letter pairs
- successor function: $w_2 = \textsc{Result}(w_1, (i, c))$ iff
  - w_1 is an English word,
  - i is smaller than or equal to the length of w_1,
  - w_2 is obtained by changing the i-th character of w_1 into c, and
  - w_2 is an English word

Example: Romania

On holiday in Romania; currently in Arad. Goal: be in Bucharest.



Example: Romania

– initial state: 'at Arad'

– actions: for each state s of the form 'at x' and each neighbor city y of x, ACTIONS(s) contains an action of the form 'drive to y'

– transition model:

s	a	RESULT(s,a)
'at Arad'	'drive to Zerind'	'at Zerind'
'at Arad'	'drive to Sibiu'	'at Sibiu'
'at Arad'	'drive to Timisoara'	'at Timisoara'
'at Zerind'	'drive to Arad'	'at Arad'
...	...	...

– goal test: {'at Bucharest'}

– path cost: the length of the path in km

Example: Romania (cont'd)

– state space: {'at x' | x is a city on the map of Romania }

– actions: each 'drive to x' where x is a city on the map of Romania

– example solution:

'at Arad', 'drive to Sibiu', 'at Sibiu', 'drive to Fagaras', 'at Fagaras',
'drive to Bucharest', 'at Bucharest'

– the optimal solution (unique in this case):

'at Arad', 'drive to Sibiu', 'at Sibiu',
'drive to Riminicu Vilcea', 'at Riminicu Vilcea',
'drive to Pitesti', 'at Pitesti',
'drive to Bucharest', 'at Bucharest'

'at' and 'drive to' are used for **precision**:

– states describe the environment and the agent at a point in time

– states **are not** cities: 'being at Arad' is not the same as 'Arad'

– actions transform a state into another state

Path cost: Variations

♢ Path cost can be specified...

...explicitly for each path

– e.g., as a table associating each path p with $pc(p)$

...using step cost: a partial function assigning to states x and y and action a a numeric value $c(x, a, y)$

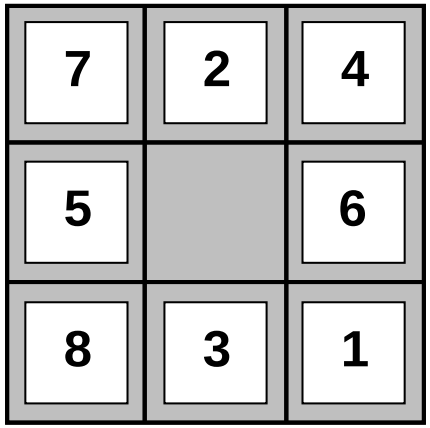
– defined if $a \in \textsc{Actions}(x)$ and $y = \textsc{Result}(x, a)$, and it specifies the cost of going from x to y via a

– also written $\textsc{Step-Cost}(x, a)$ (assuming actions are deterministic)

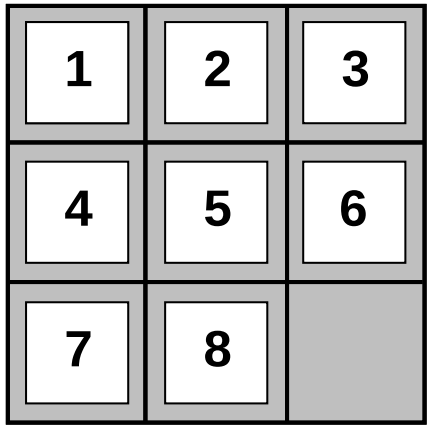
– we then define $pc(s_0, a_1, s_1, \ldots, a_n, s_n) = \Sigma_{1 \leq i \leq n}\, c(s_{i-1}, a_i, s_i)$

x	a	y	$c(x, a, y)$
'at Arad'	'drive to Zerind'	'at Zerind'	75
'at Arad'	'drive to Sibiu'	'at Sibiu'	140
...			
'at Sibiu'	'drive to Arad'	'at Arad'	140

Example: The 8-puzzle



Start State

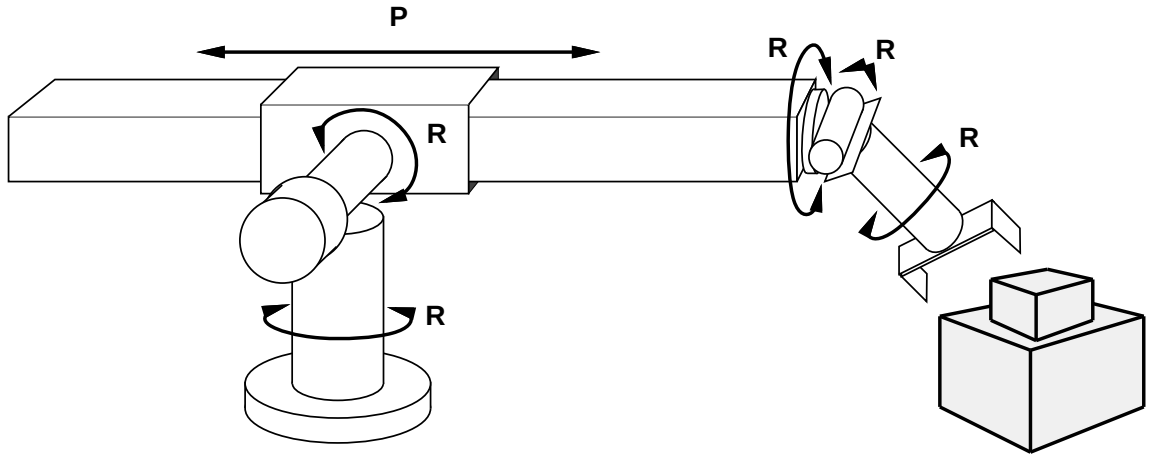


Goal State

states: integer locations of tiles (ignore intermediate positions)
actions: move blank left, right, up, down (ignore unjamming etc.)
goal test: the goal state given above
step cost: 1 per move

[Note: finding the shortest solution of an n-puzzle is NP-hard]

Example: Robotic assembly



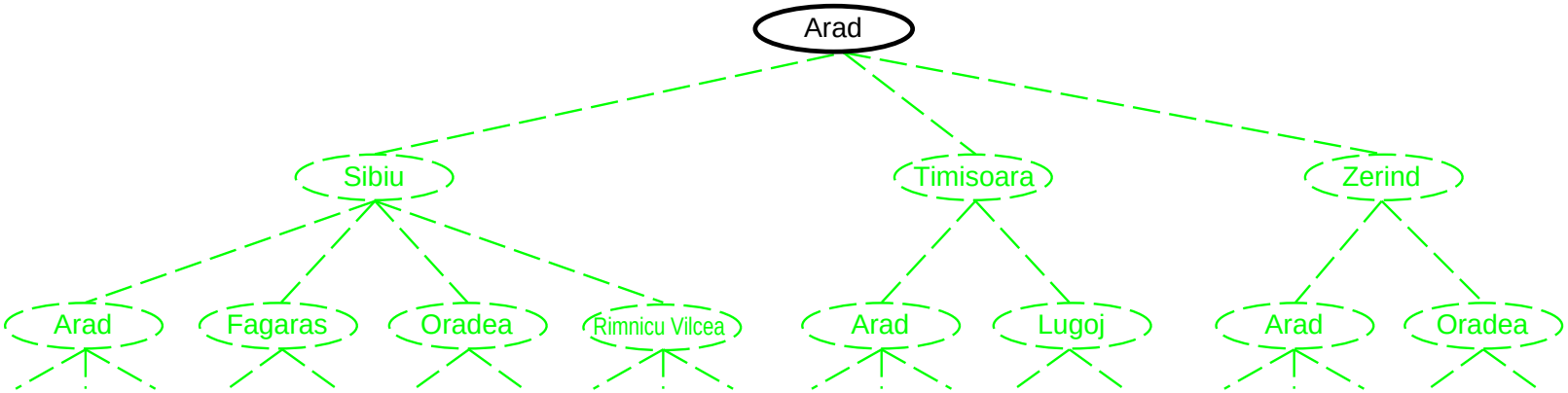
– states: real-valued coordinates of robot joint angles and
the parts of the object to be assembled
– actions: continuous motions of robot joints
– goal test: complete assembly
– path cost: time to execute

Tree search algorithms

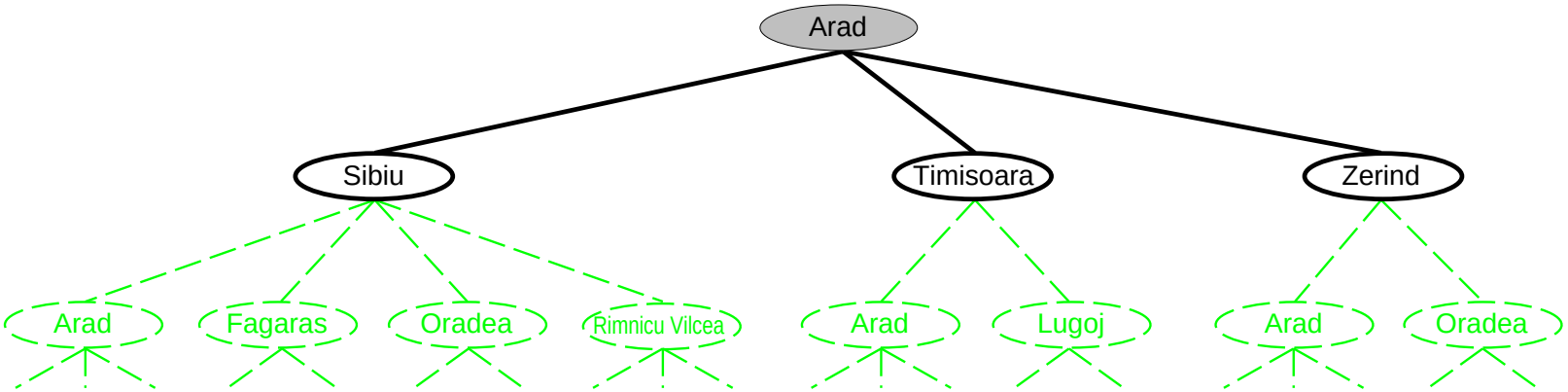
Basic idea: simulate exploration of state space by generating successors of already explored states (a.k.a. expanding states)

```
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
  end
```

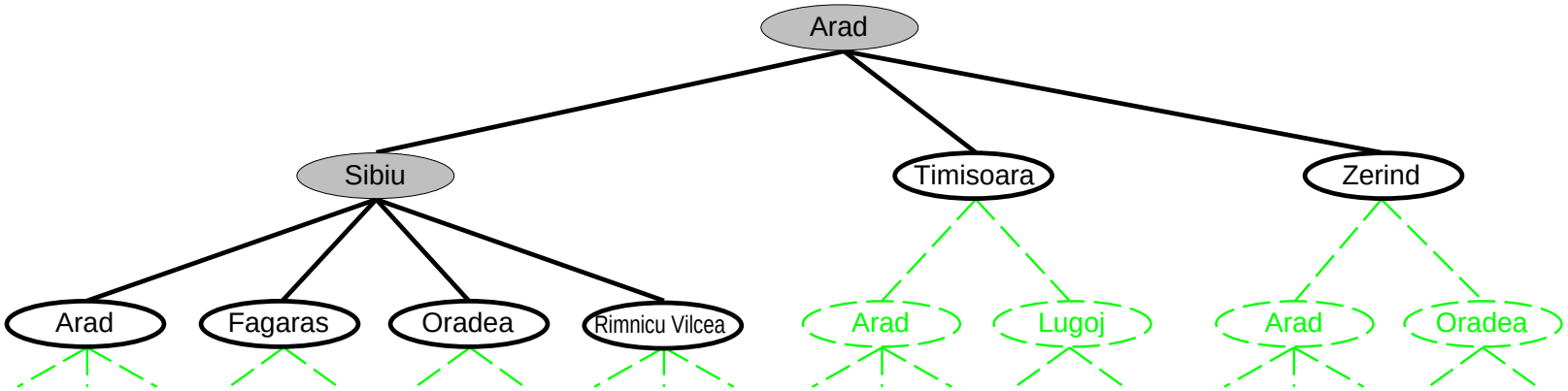
Tree search example



Tree search example



Tree search example



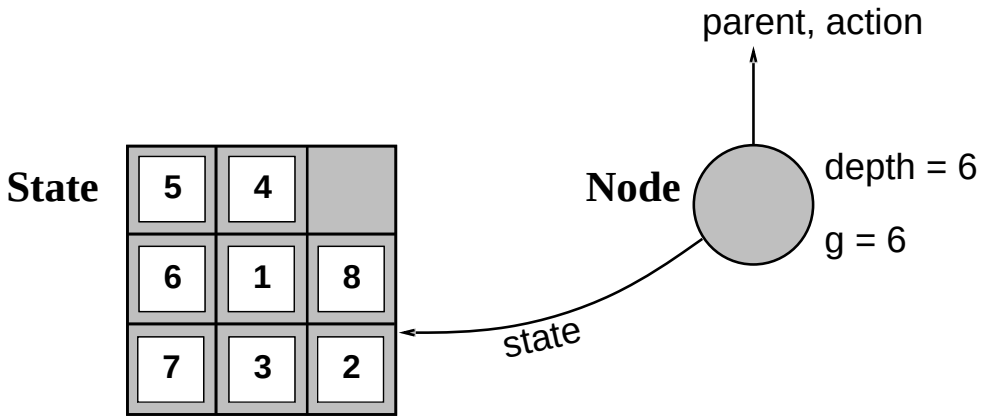
States vs. nodes

A state is (a representation of) a physical configuration

A node is a data structure constituting part of a search tree

– it includes parent, children, depth, path cost $g(x)$

States do not have parents, children, depth, or path cost!



The Child-Node function creates a new node, fills in various fields, and uses the Result function to create the corresponding state.

General tree search

```
function TREE-SEARCH(problem, frontier) returns a solution, or failure
    INSERT(ROOT-NODE(problem.INITIAL-STATE), frontier)
    while not EMPTY?(frontier) do
        node ← REMOVE(frontier)
        if problem.GOAL-TEST applied to node.STATE succeeds return node
        for each action in problem.ACTIONS(node.STATE) do
            INSERT(CHILD-NODE(problem, node, action), frontier)
    return failure
```

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action),
        PARENT = parent, ACTION = action, DEPTH = parent.DEPTH + 1,
        PATH-COST = parent.PATH-COST +
                    problem.STEP-COST(parent.STATE, action)
```

Search strategies

A 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/expanded
- 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 ∞)

Uninformed search strategies

Uninformed strategies use only the information from problem's definition

$\Rightarrow$ No information about **closeness** to a goal

– Breadth-first search

– Uniform-cost search

– Depth-first search

– Depth-limited search

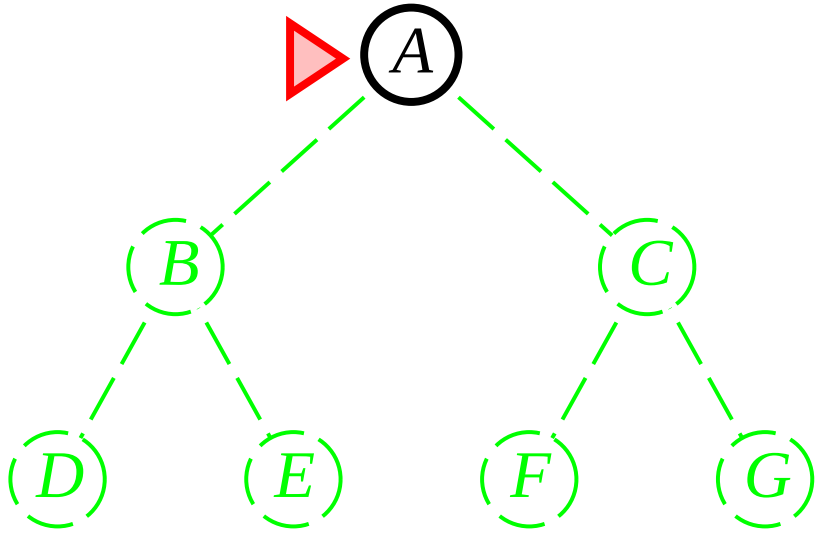
– Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier = FIFO queue—that is, new successors go at end

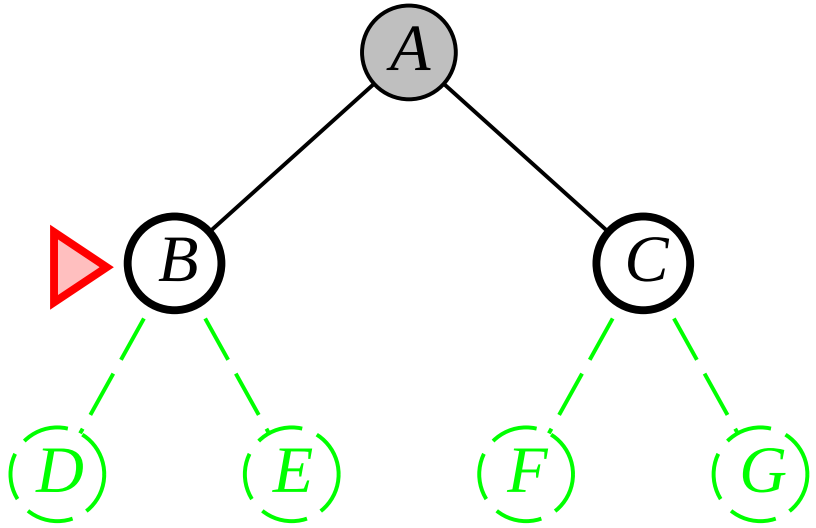


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier = FIFO queue—that is, new successors go at end

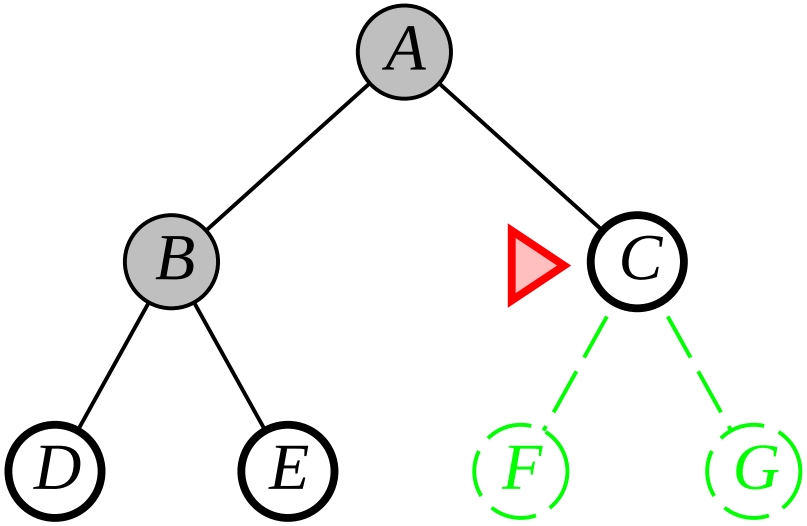


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier = FIFO queue—that is, new successors go at end

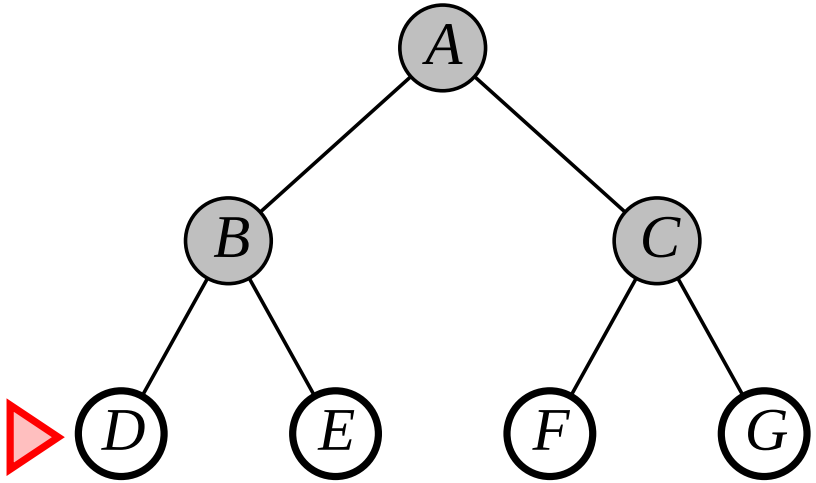


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier = FIFO queue—that is, new successors go at end



Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Expand all but the last node at level d in the worst case (goal itself not expanded), generating $b(b^d - 1)$ nodes.

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

But ... hang on a second, didn't you learn in your Algorithms class that BFS is a linear time algorithm?

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

frontier = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost non-negative and $\geq \epsilon$

May get into an infinite loop if we had zero-cost actions leading to same state (e.g., NoOp in the vacuum world)

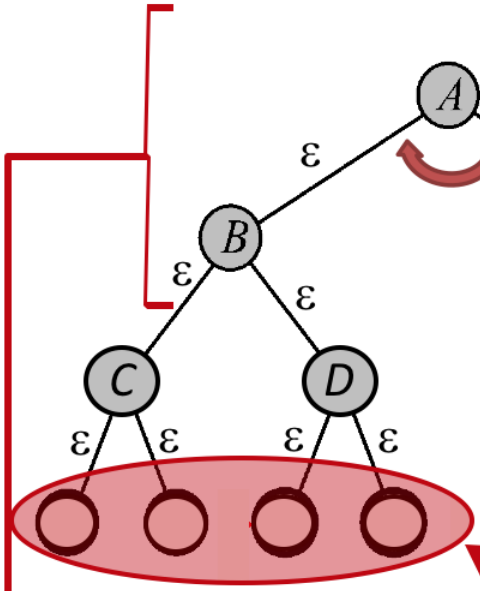
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Uniform-cost search analysis



branching factor: *the (maximum) number of actions available at any state*
b

Assume that a solution is found here

time: *estimated by the total number of nodes generated*
$b + b^2 + b^3 + \ldots + b^{1+c^*/\varepsilon} - b \in O(b^{1+c^*/\varepsilon}) \quad (b>1)$

depth: *the level in the tree where the (least-cost) solution is found*
d

space: *the storage requirement to hold the frontier*
$O(b^{1+c^*/\varepsilon})$

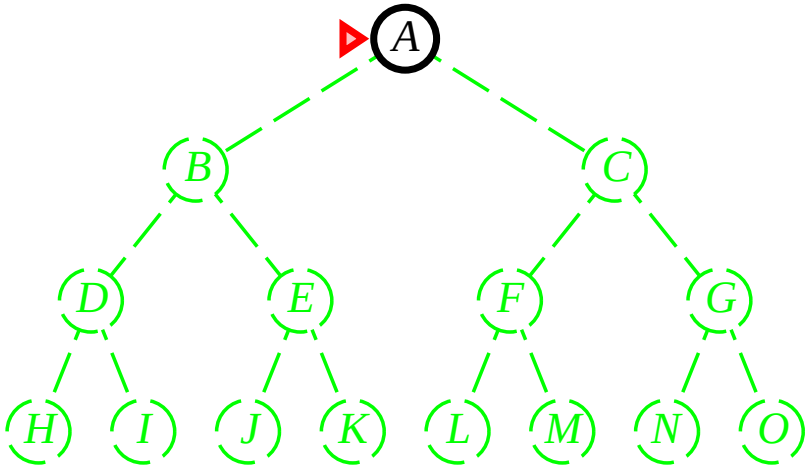
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 0



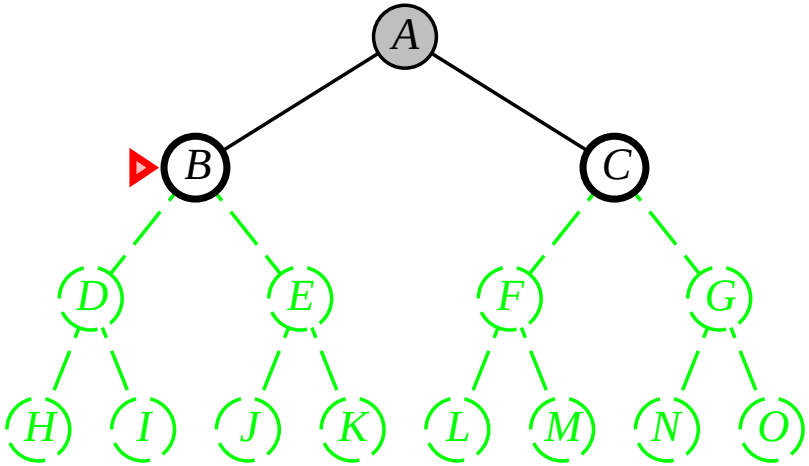
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 1



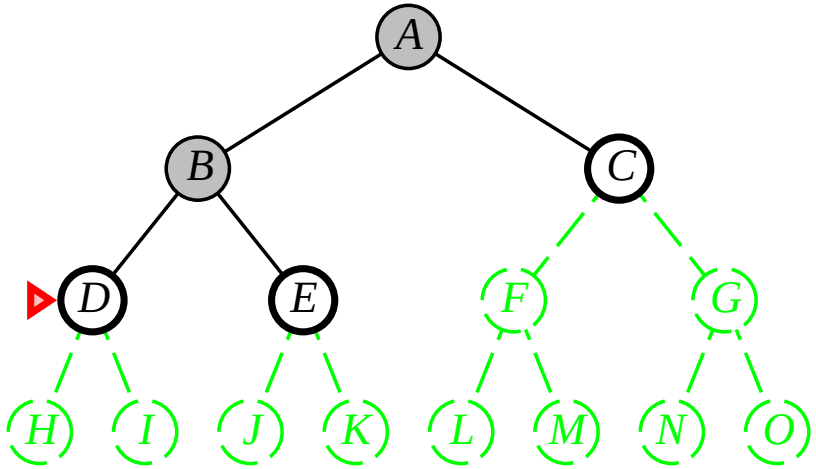
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 2



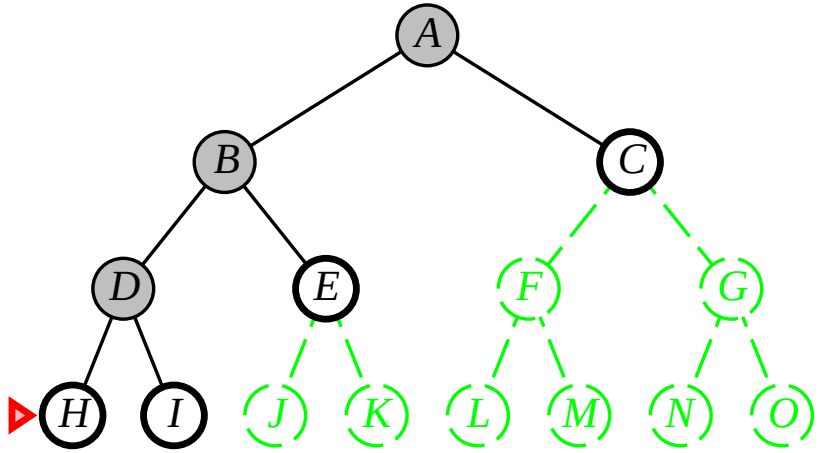
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 3



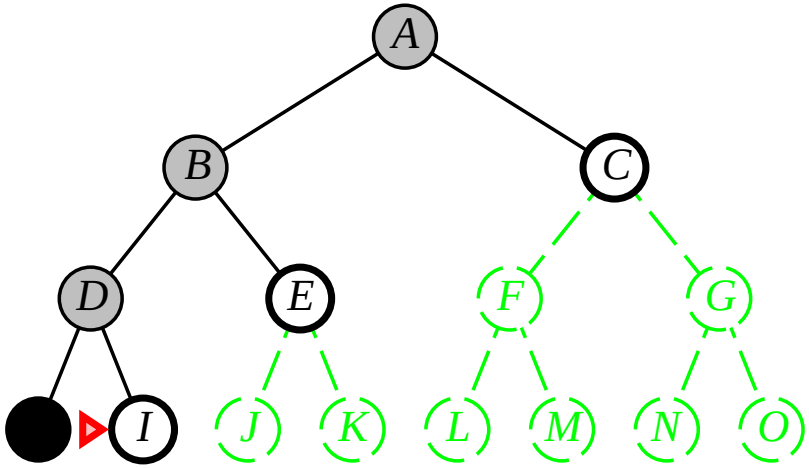
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 4



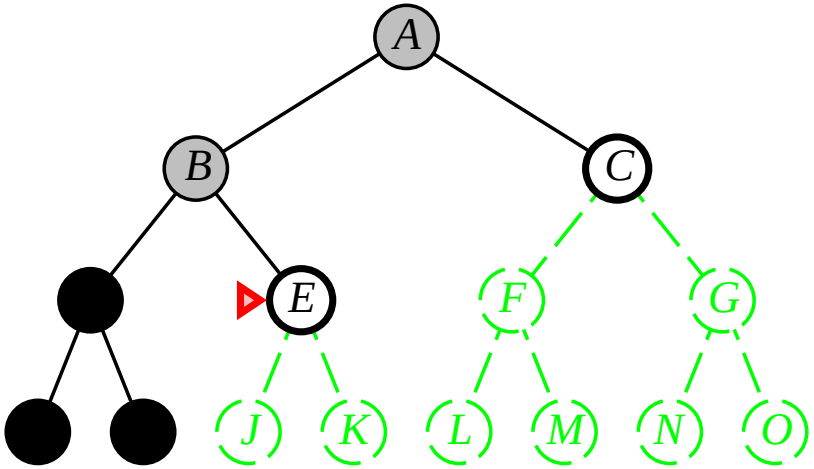
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 5



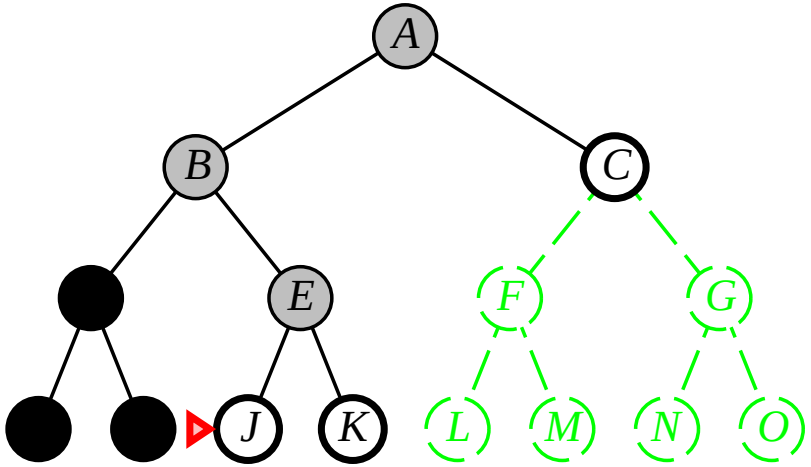
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 6



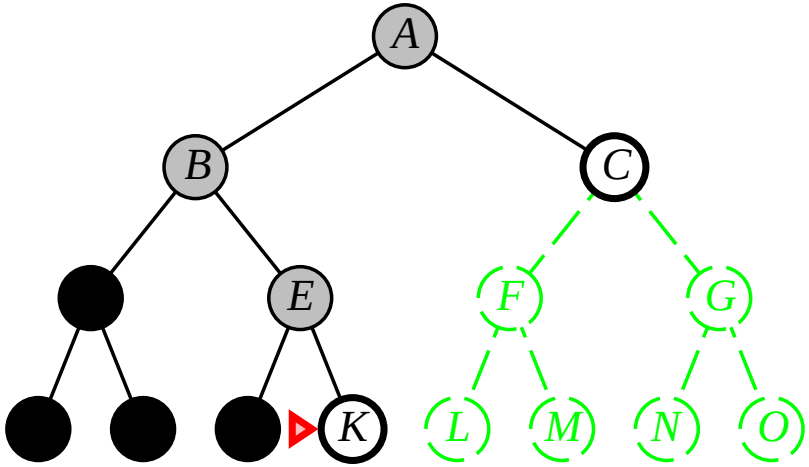
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 7



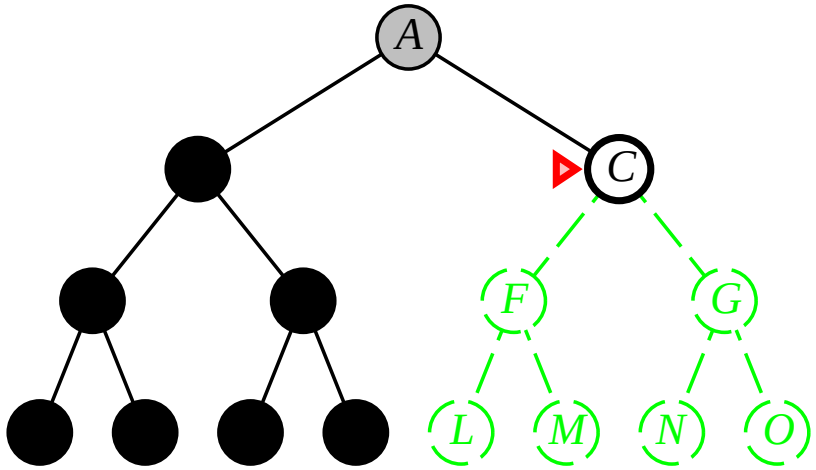
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 8



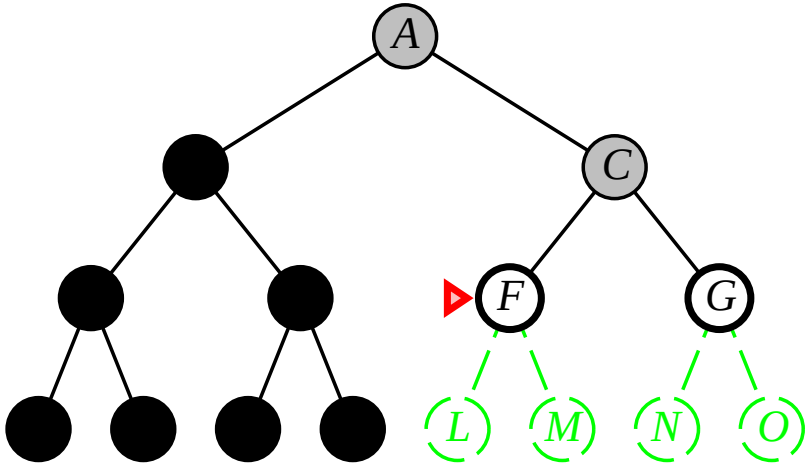
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 9



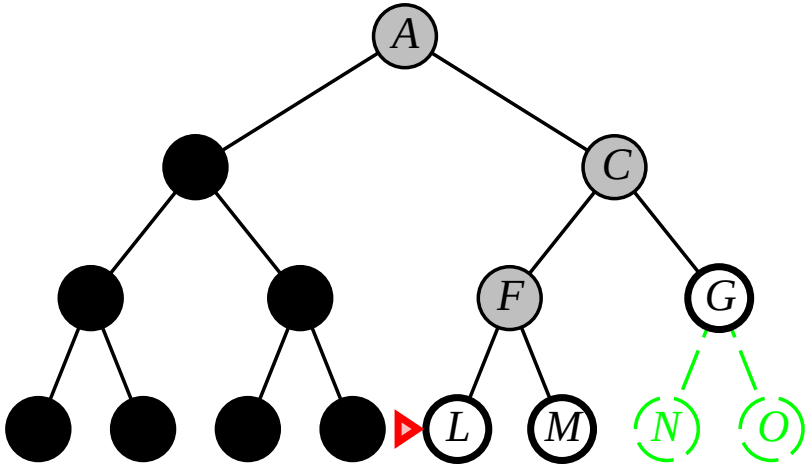
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 10



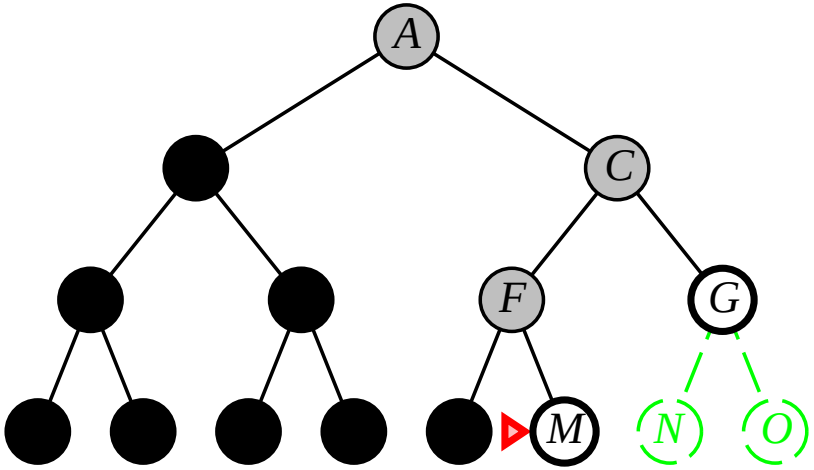
Depth-first search

Expand deepest unexpanded node

Implementation:

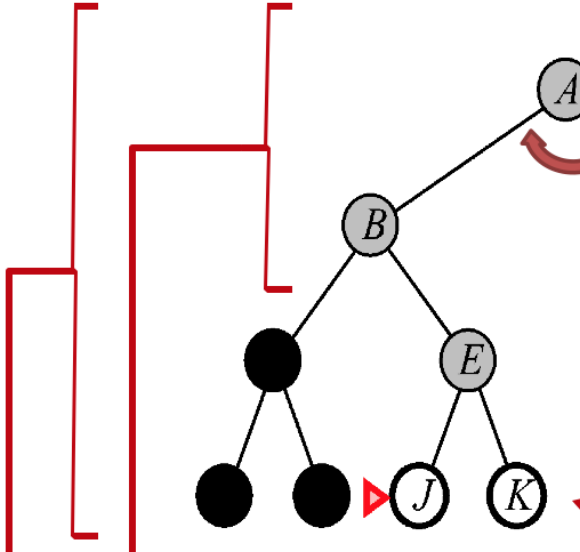
frontier = LIFO queue (i.e., *stack*), so put successors at front

Expansions: 11



Analysis: DFS

Analysis – DFS



branching factor: *the (maximum) number of actions available at any state*
b

Assume that a solution is found here

time: *estimated by the total number of nodes generated*
$O(b^m)$ $(b>1)$

depth: *the level in the tree where the solution is found*
d

space: *the storage requirement to hold the frontier*
$O(mb)$

max depth: *the maximum level in the tree*
m (may be infinite)

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ 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

$\diamondsuit$ *but* very easy to implement using the call-stack!

Depth-limited search

Depth-first search with depth limit l

$\Rightarrow$ i.e., nodes at depth l have no successors

Recursive implementation:

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

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if problem.GOAL-TEST applied to node.STATE succeeds return node
  else if node.DEPTH = limit then return cutoff
  else for each action in problem.ACTIONS(node.STATE) do
    successor ← CHILD-NODE(problem, node, action)
    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 ≠ cutoff then return result
    end
```

Space requirements are modest, as for depth-first search

Repeats work at each new maximum search depth
– but, extra work is modest!

Iterative deepening search $l = 0$

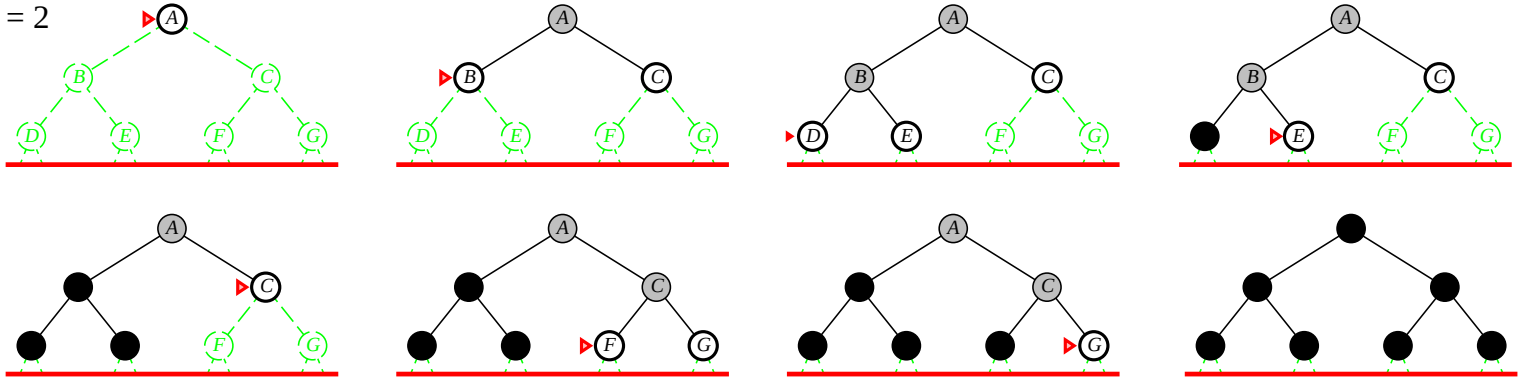
Limit = 0

Iterative deepening search $l = 1$

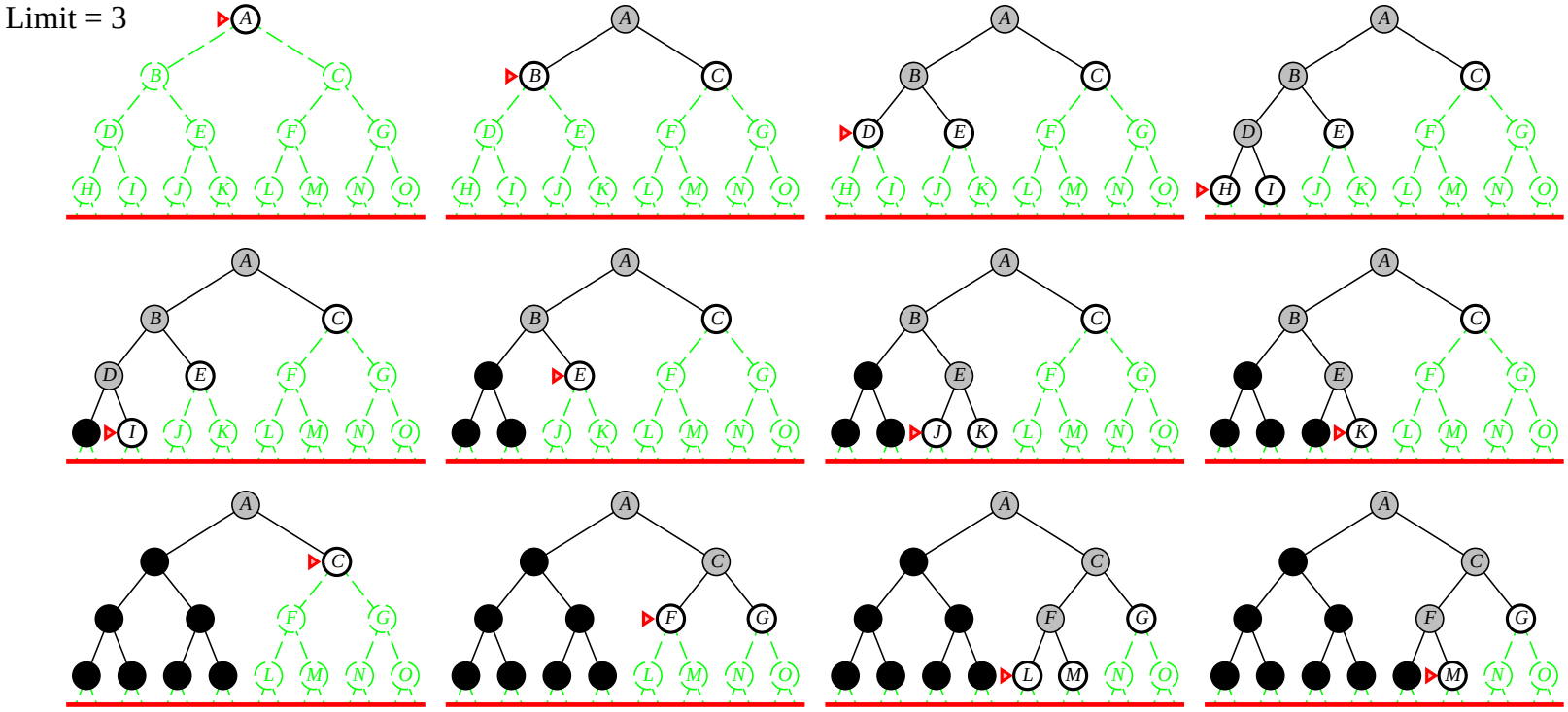
Limit = 1

Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$



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?? $O(bd)$

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:

$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is **generated**

Summary of algorithms

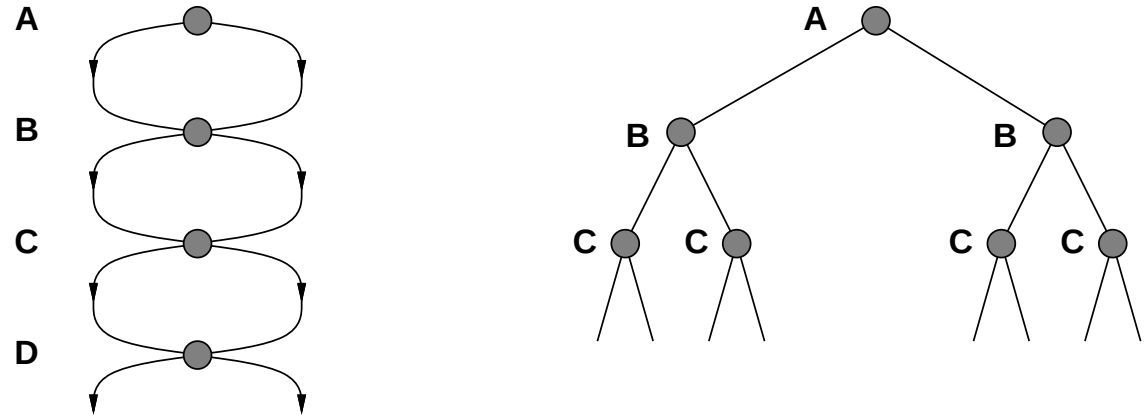
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^d	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	$> b^d$
Space	b^d	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes*	No	No	Yes

♢ * means that the property holds if the branching factor b is finite

♢ Note: this is slightly different to that in the book — but fairer, I think!

Repeated states

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



♢ Recognizing states that we have encountered before can be nontrivial:

- there may be too many of them to remember
- it takes too long to compare states (long *signature*)
- the state-space may be continuous (and we want to recognize states that are "close" to something we have seen before)

Graph search

```
function GRAPH-SEARCH(problem, frontier) returns a solution, or failure
    explored ← an empty set
    INSERT(ROOT-NODE(problem.INITIAL-STATE), frontier)
    while not EMPTY?(frontier) do
        node ← REMOVE(frontier)
        add node.STATE to explored
        if problem.GOAL-TEST applied to node.STATE succeeds return node
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored then
                INSERT(child, frontier)
    return failure
```

Requires being able to compare states for equality

A slightly more efficient implementation also checks whether *frontier* contains a node with state equal to *child*.STATE

Properties of graph search

Complete?? Yes

Time?? proportional to state space size;
can be exponentially more efficient than tree search

Space?? proportional to state space size;
can be exponentially more efficient than tree search

Optimal?? not in general;
yes with uniform cost and breadth-first search

Avoiding repeated states on a path

Graph search requires memory linear in the size of the state space

⇒ can be problematic in practice

We can check for repeated states on the current **path** only

– applicable to both depth-first and breadth-first search

– no additional memory required

– ensures completeness (and termination) of depth-first search

⇒ Not the same as graph search

⇒ Can be useful when state spaces are large

– c.f. AND-OR search in Lectures 15 and 16

Summary

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

Graph search can be exponentially more efficient than tree search