Problem solving and search

Chapter 3

Reminders

Assignments and lecture notes are posted on D2L

Project 0 due Thu 9-06 at 5pm

Project 1 will be posted today or tomorrow, due 9-18 at 5pm

Outline

◇ Course goal: Acting Rationality

◇ Agents and environments

◇ Problem-solving agents

◇ Problem types

◇ Problem formulation

◇ Example problems

◇ Basic search algorithms

What is AI?

Systems that think like humans	Systems that think rationally
Systems that act like humans	Systems that act rationally

Acting rationally

Rational behavior: doing the right thing

The right thing: that which is expected to maximize goal achievement, given the available information

Doesn't necessarily involve thinking—e.g., blinking reflex—but thinking should be in the service of rational action

Aristotle (Nicomachean Ethics):

> Every art and every inquiry, and similarly every action and pursuit, is thought to aim at some good
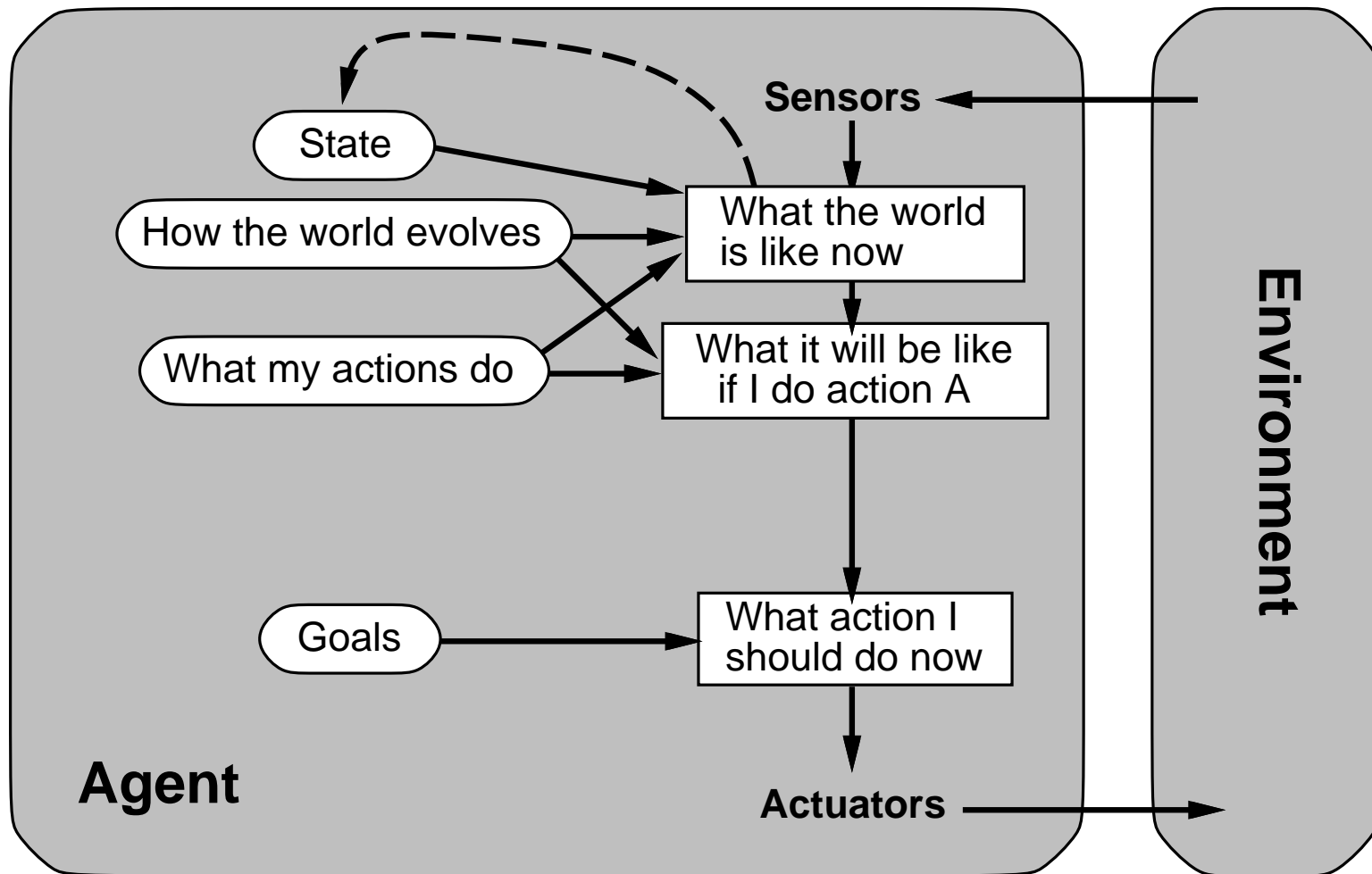
Agents and environments
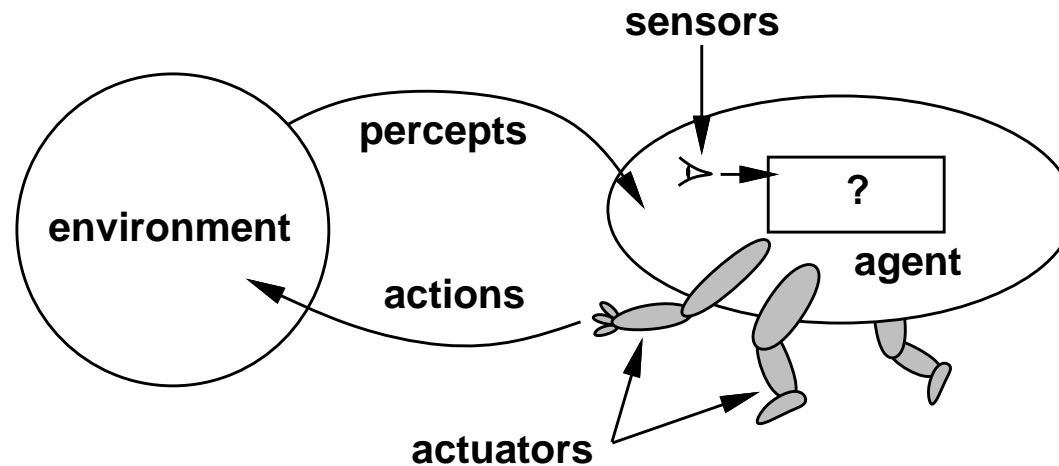
Agents include humans, robots, softbots, thermostats, etc.

The agent function maps from percept histories to actions:

$$f : \mathcal{P}^* \to \mathcal{A}$$

The agent program runs on the physical architecture to produce f

Goal-based agents

State → What the world is like now

Sensors

How the world evolves → What the world is like now

What my actions do → What it will be like if I do action A

What the world is like now → What it will be like if I do action A

Goals → What action I should do now

What it will be like if I do action A → What action I should do now

Agent

Actuators

Environment

Big picture

Agents interact with environments through actuators and sensors

The agent function describes what the agent does in all circumstances

The performance measure evaluates the environment sequence

A perfectly rational agent maximizes expected performance

Agent programs implement (some) agent functions

Environments are categorized along several dimensions:
 observable? deterministic? episodic? static? discrete? single-agent?

Several basic agent architectures exist:
 reflex, reflex with state, goal-based, utility-based

Problem-solving agents

Restricted form of general 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
```

Note: this is offline problem solving; solution executed "eyes closed."
Online problem solving involves acting without complete knowledge.

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest
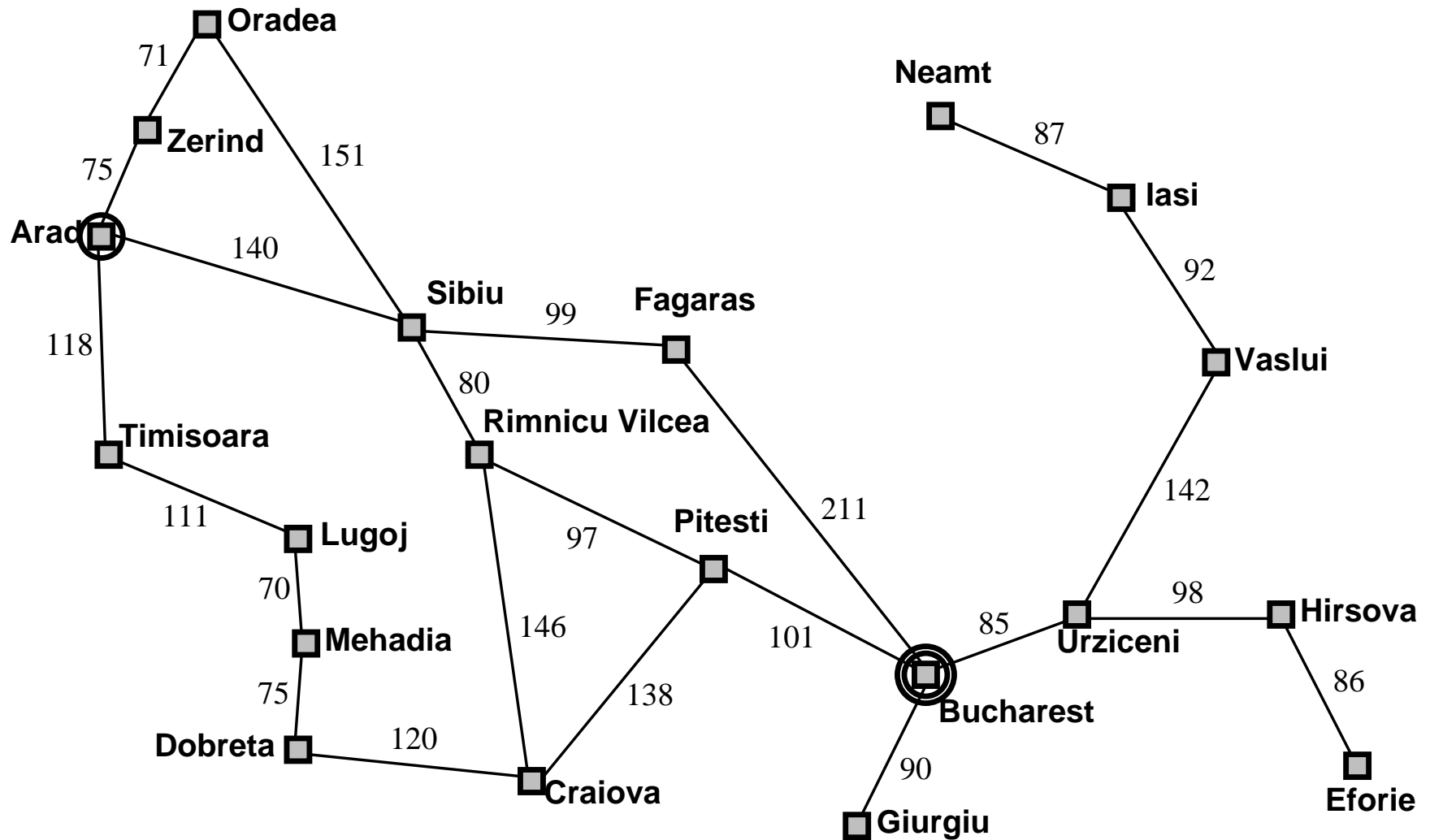
Formulate goal:
       be in Bucharest

Formulate problem:
       states: various cities
       actions: drive between cities

Find solution:
       sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania

Problem types

Deterministic, fully observable $\implies$ single-state problem
 Agent knows exactly which state it will be in; solution is a sequence

Non-observable $\implies$ conformant problem
 Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable $\implies$ contingency problem
 percepts provide **new** information about current state
 solution is a contingent plan or a policy
 often **interleave** search, execution

Unknown state space $\implies$ exploration problem ("online")

Single-state problem formulation

A problem is defined by four items:

initial state   e.g., "at Arad"

successor function $S(x) =$ set of action–state pairs
  e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \ldots\}$

goal test, can be
  explicit, e.g., $x =$ "at Bucharest"
  implicit, e.g., $NoDirt(x)$

path cost (additive)
  e.g., sum of distances, number of actions executed, etc.
  $c(x, a, y)$ is the step cost, assumed to be ≥ 0

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

Selecting a state space

Real world is absurdly complex
$\Rightarrow$ state space must be **abstracted** for problem solving

(Abstract) state $=$ set of real states

(Abstract) action $=$ complex combination of real actions
    e.g., "Arad $\rightarrow$ Zerind" represents a complex set
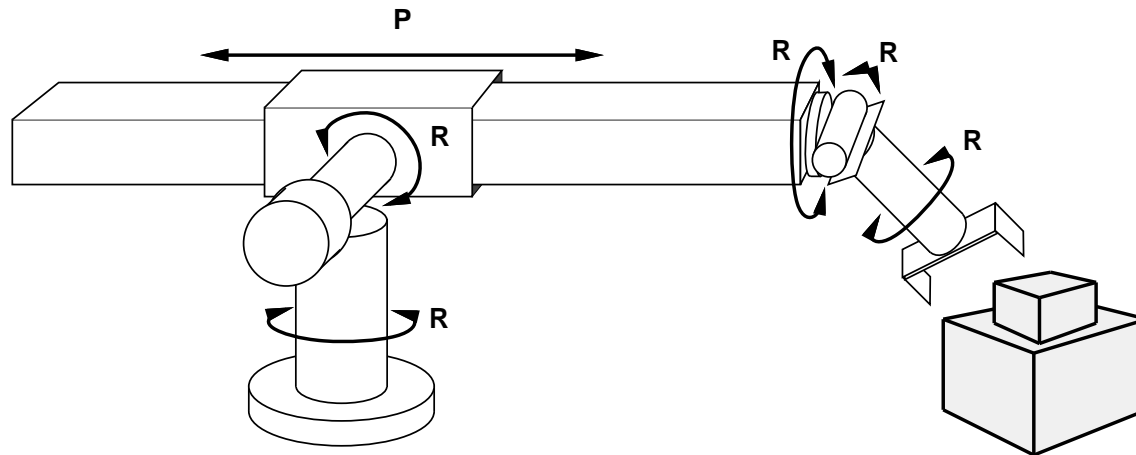   of possible routes, detours, rest stops, etc.
For guaranteed realizability, **any** real state "in Arad"
    must get to some real state "in Zerind"

(Abstract) solution $=$
        set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!

Example: robotic assembly

states??: real-valued coordinates of robot joint angles
      parts of the object to be assembled

actions??: continuous motions of robot joints

goal test??: complete assembly **with no robot included!**

path cost??: time to execute

Tree search algorithms

Basic idea:

    offline, simulated 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

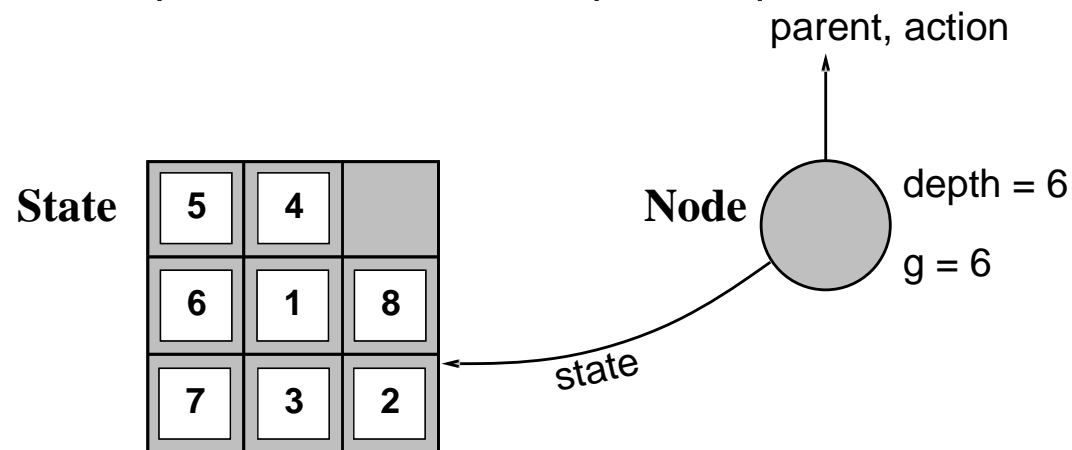# Tree search example
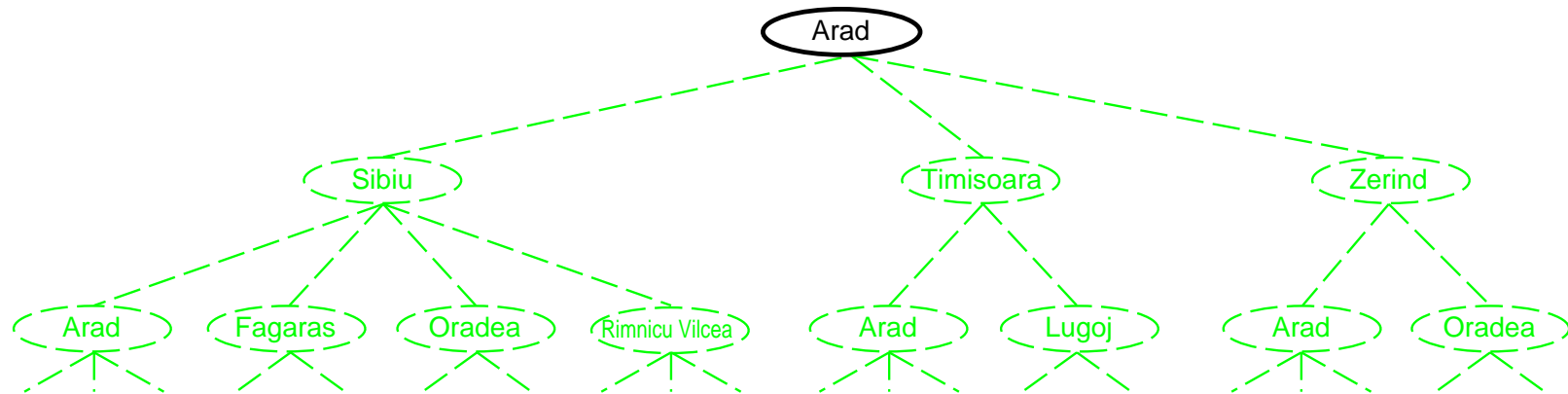
Tree search example
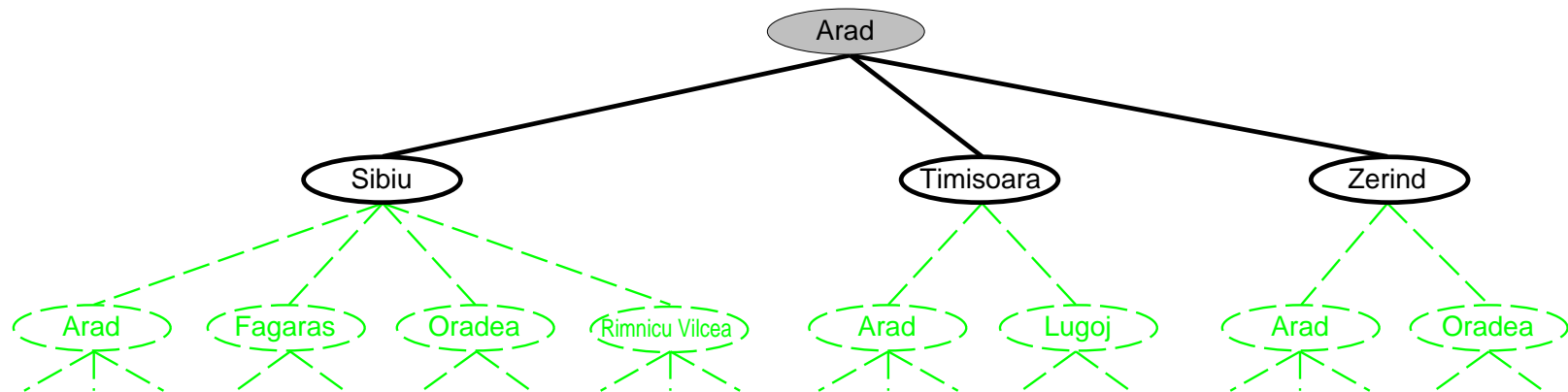
Tree search example
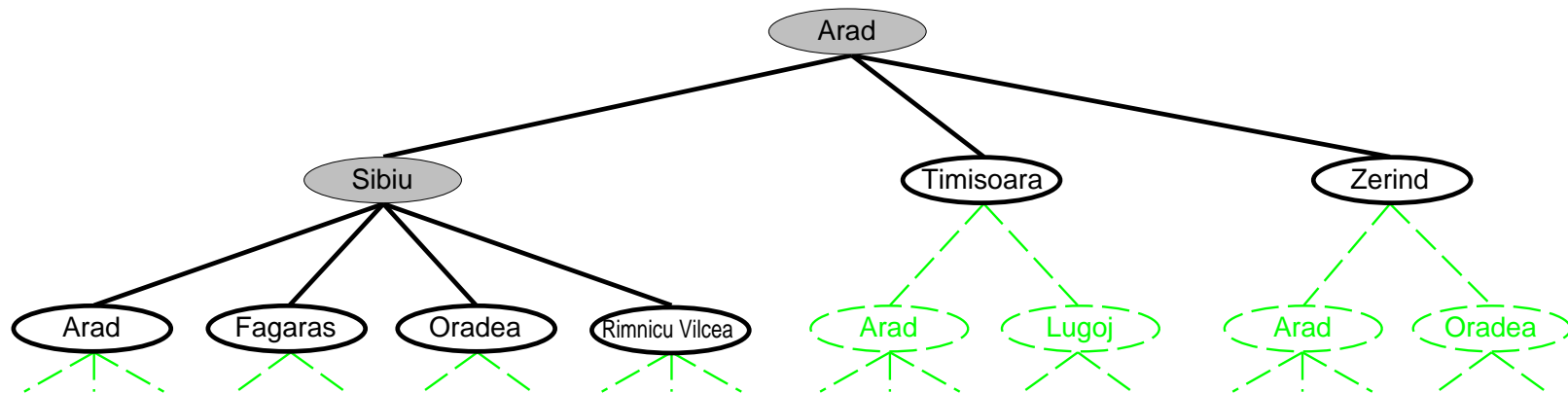
Implementation: states vs. nodes

A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree
      includes parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!

The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem 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(STATE[*node*], *action*, *result*)
      DEPTH[*s*] ← DEPTH[*node*] + 1
      add *s* to *successors*
   return *successors*

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 available
in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search
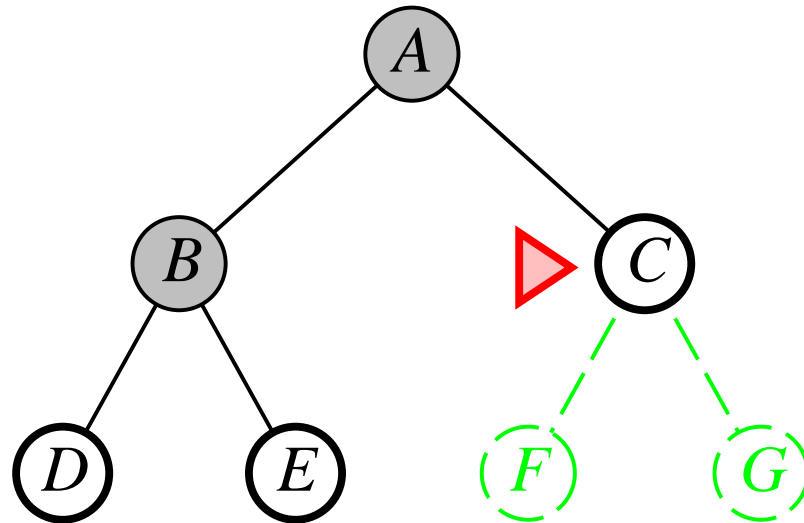
Iterative deepening search

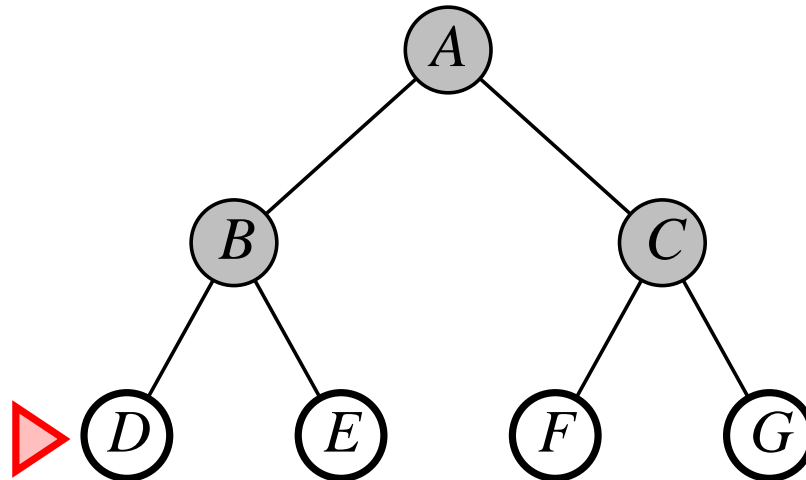# Breadth-first search

Expand shallowest unexpanded node

Implementation:

    fringe is a FIFO queue, i.e., new successors go at end
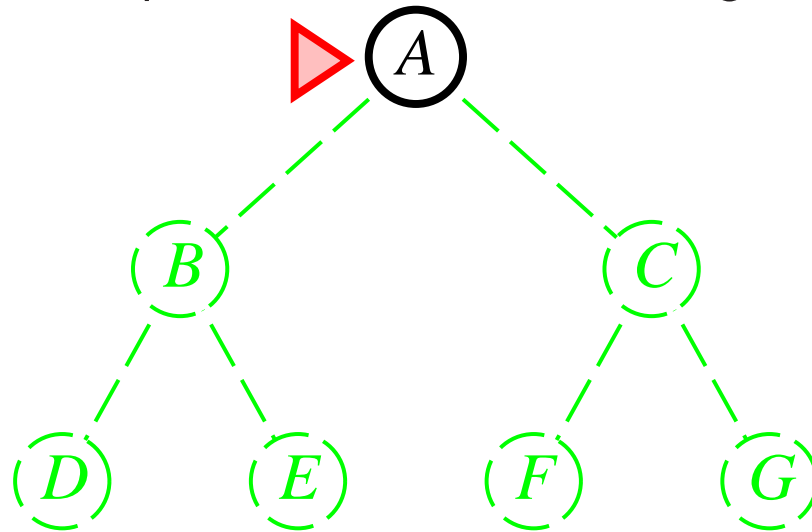
Breadth-first search

Expand shallowest unexpanded node

Implementation:

    fringe is a FIFO queue, i.e., new successors go at end
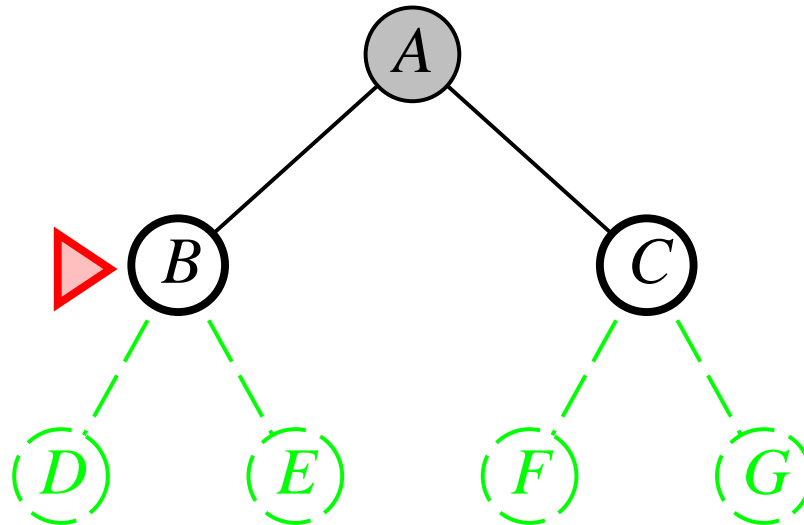
Breadth-first search

Expand shallowest unexpanded node

Implementation:

  fringe is a FIFO queue, i.e., new successors go at end

Breadth-first search

Expand shallowest unexpanded node

Implementation:
      fringe is a FIFO queue, i.e., new successors go at end

Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

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

Space??

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

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

Optimal??

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

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 $= 8640$GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

<u>Complete</u>?? Yes, if step cost $\geq \epsilon$

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

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

<u>Optimal</u>?? Yes—nodes expanded in increasing order of $g(n)$
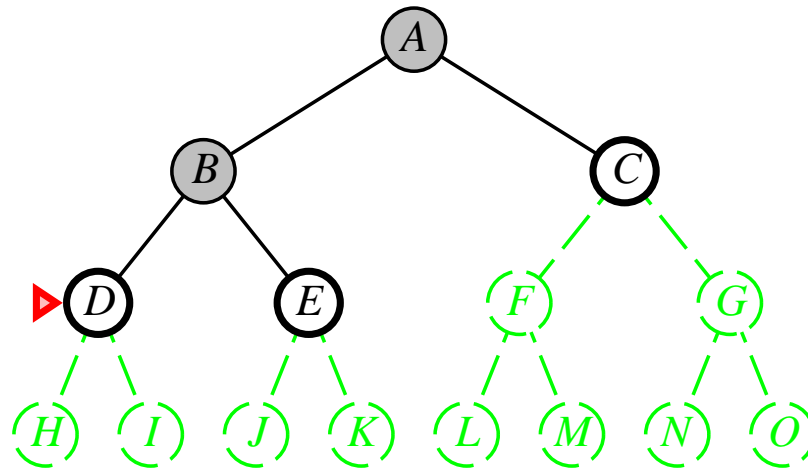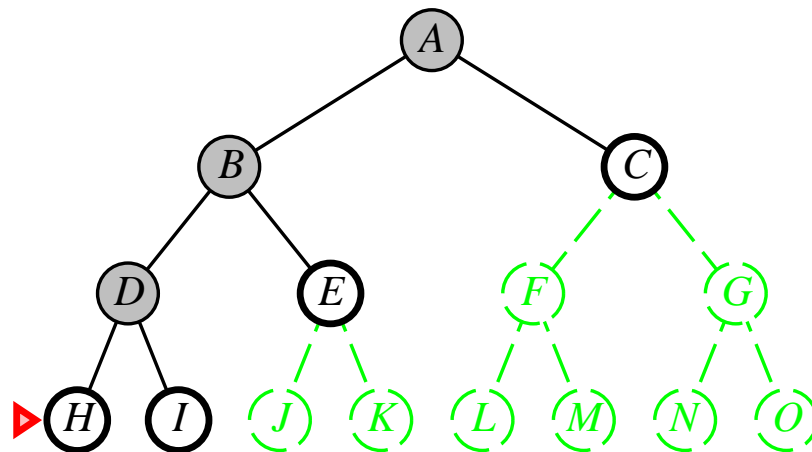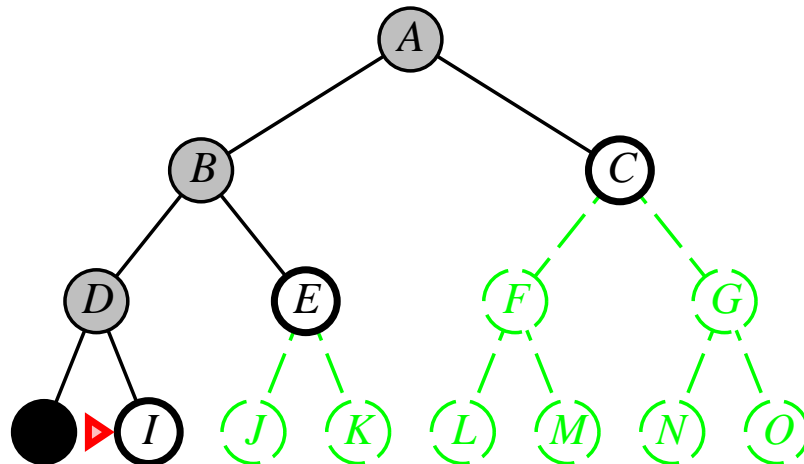
Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe =$ LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:

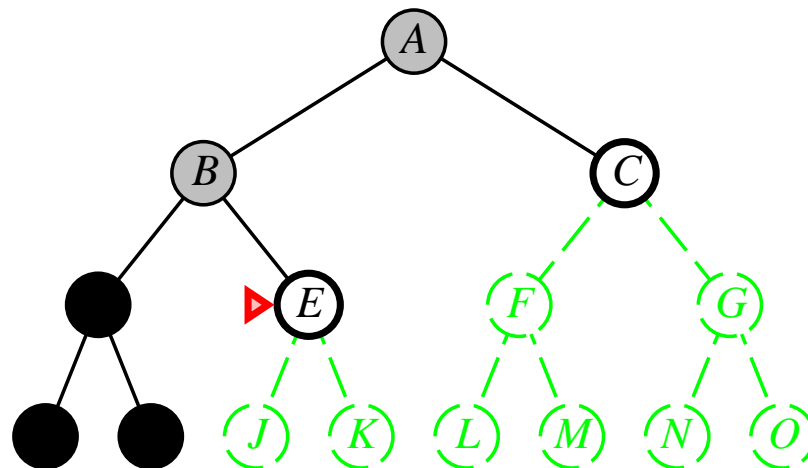$fringe = $ LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:
$fringe$ = LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:
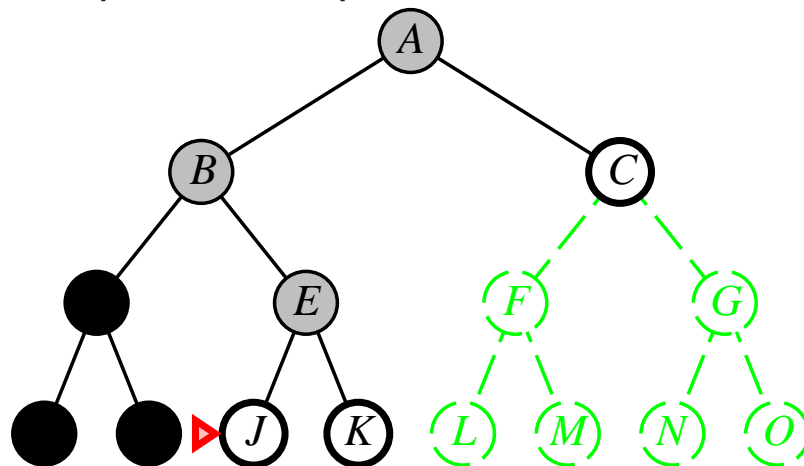 $fringe =$ LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:

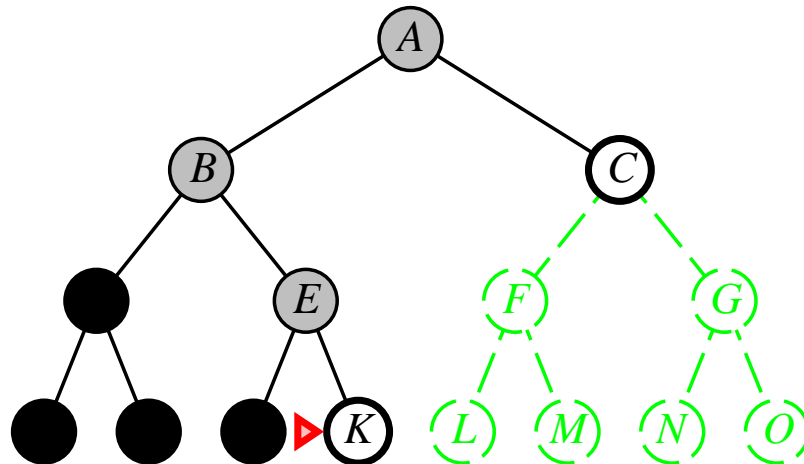$fringe =$ LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe =$ LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:

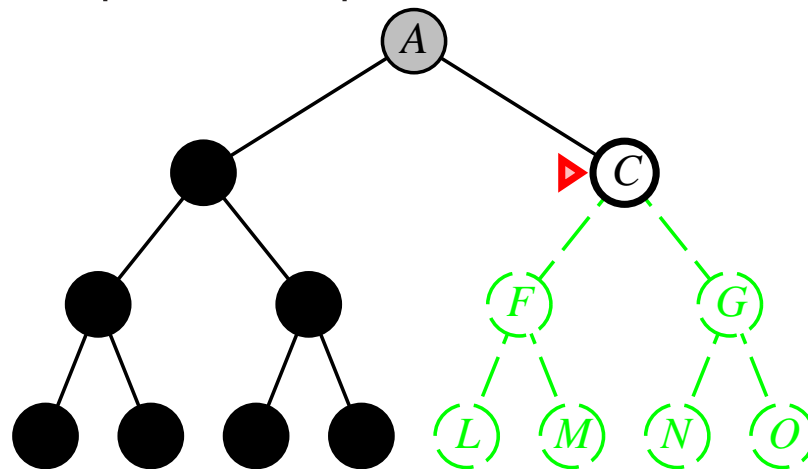 $fringe$ = LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:
$fringe$ = LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:

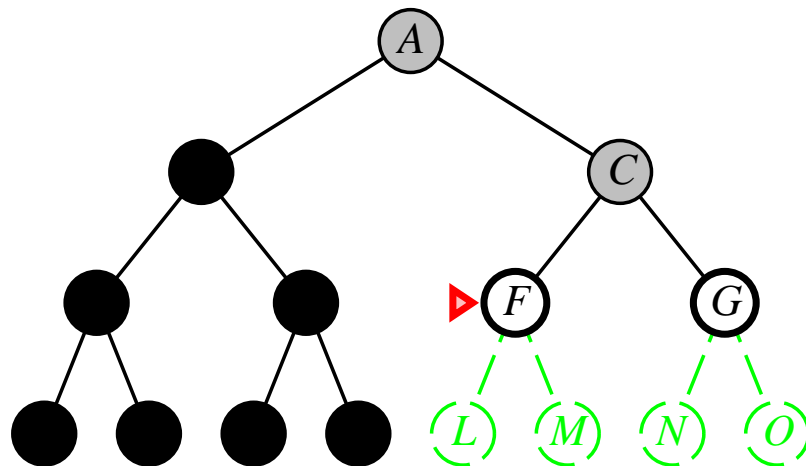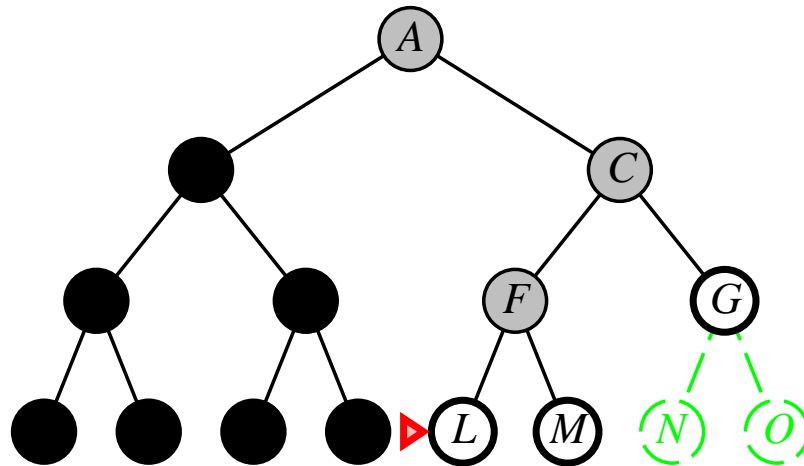$fringe$ = LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:
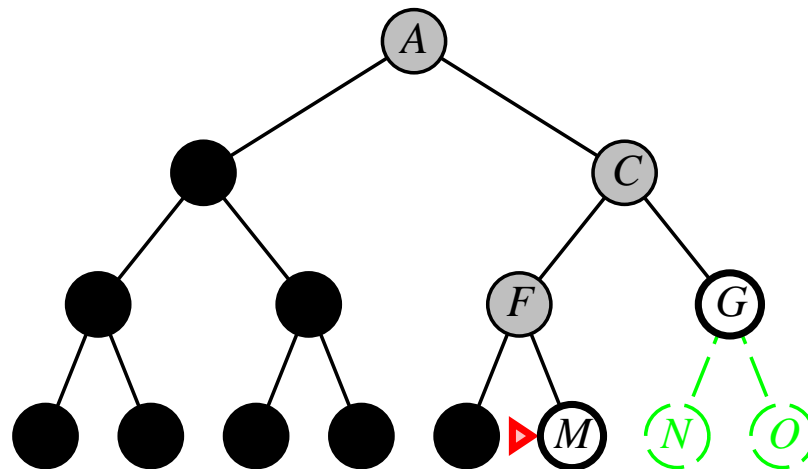$fringe = $ LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

Properties of depth-first search

Complete??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
        Modify to avoid repeated states along path
            ⇒ complete in finite spaces

Time??

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??

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??

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

Depth-limited search

$=$ depth-first search with depth limit l,

i.e., 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* $\neq$ *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*)
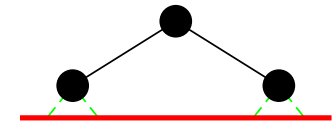   **if** *result* ≠ cutoff **then return** *result*
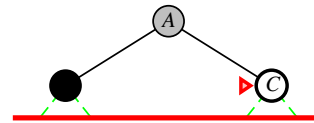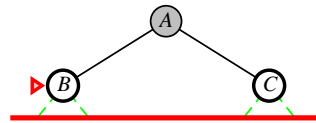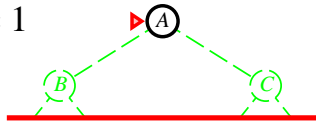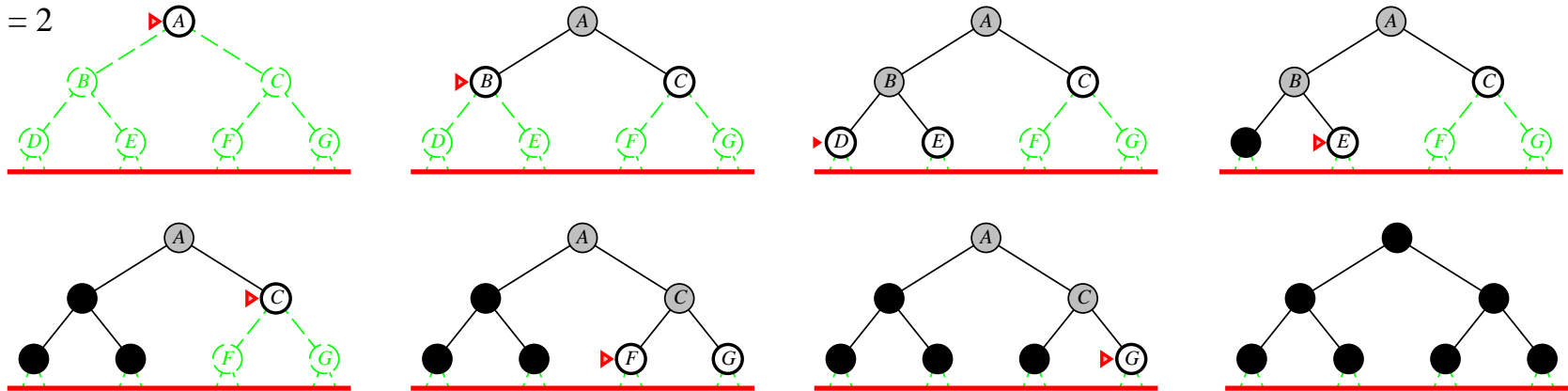 **end**

Iterative deepening search $l = 0$
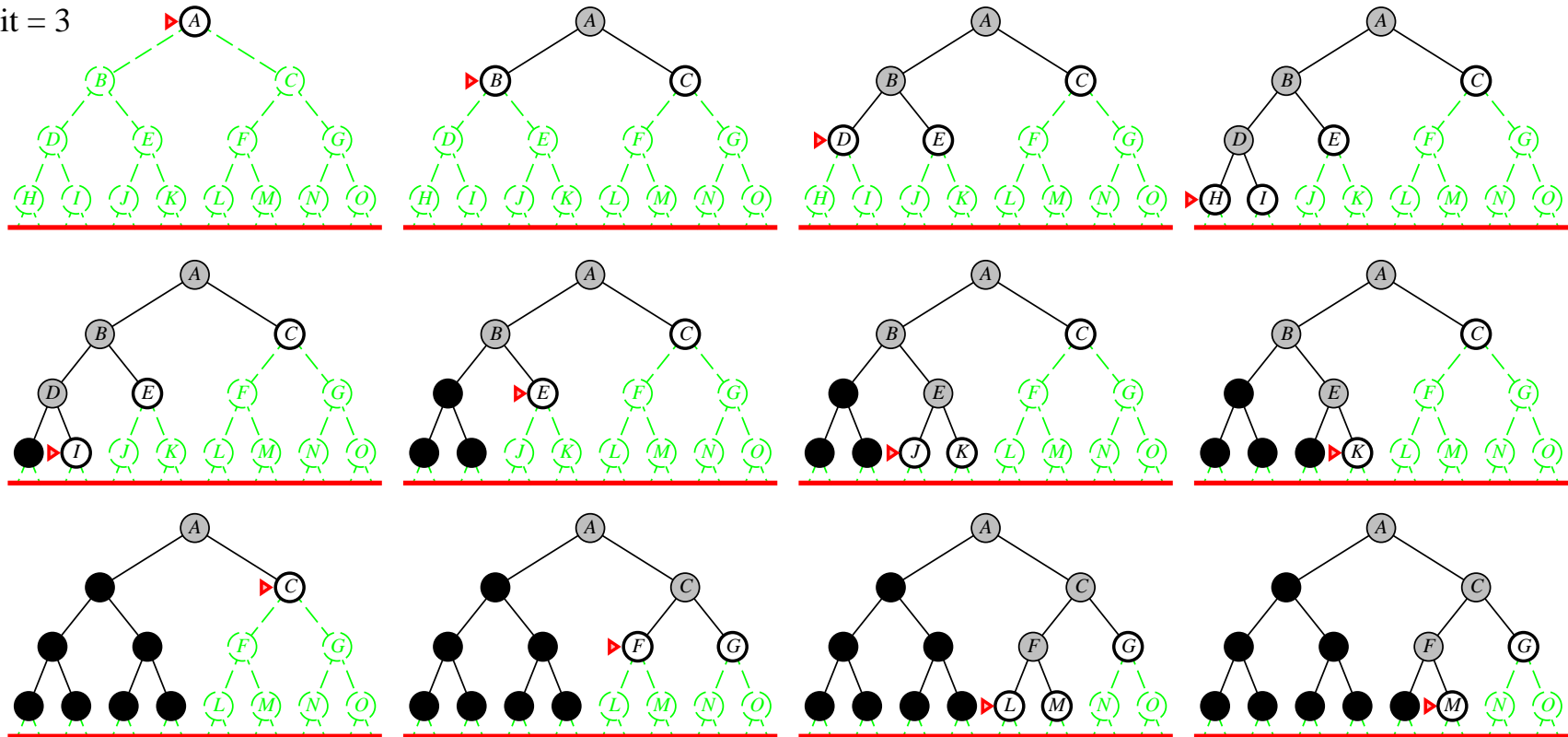
Limit = 0

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??

Properties of iterative deepening search

Complete?? Yes

Time??

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??

Properties of iterative deepening search

<u>Complete</u>?? Yes

<u>Time</u>?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

<u>Space</u>?? $O(bd)$

<u>Optimal</u>??

Properties of iterative deepening search

<u>Complete</u>?? Yes

<u>Time</u>?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$

<u>Space</u>?? $O(bd)$

<u>Optimal</u>?? 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:

$$N(\mathsf{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\mathsf{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

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+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*

Repeated states

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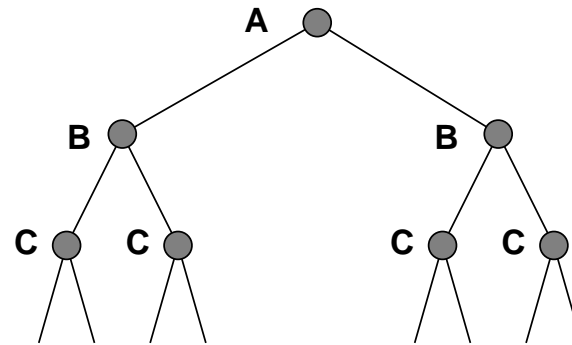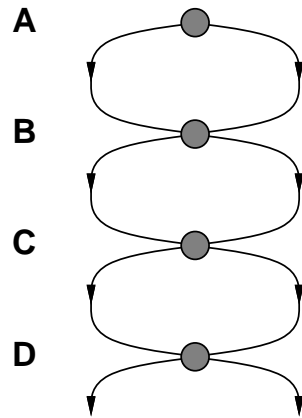
Graph search

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

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