Applied AI

Lecture 2

Dr Artie Basukoski

[slides adapted from Artificial Intelligence: A Modern Approach, Russel and Norvigl]

Agenda

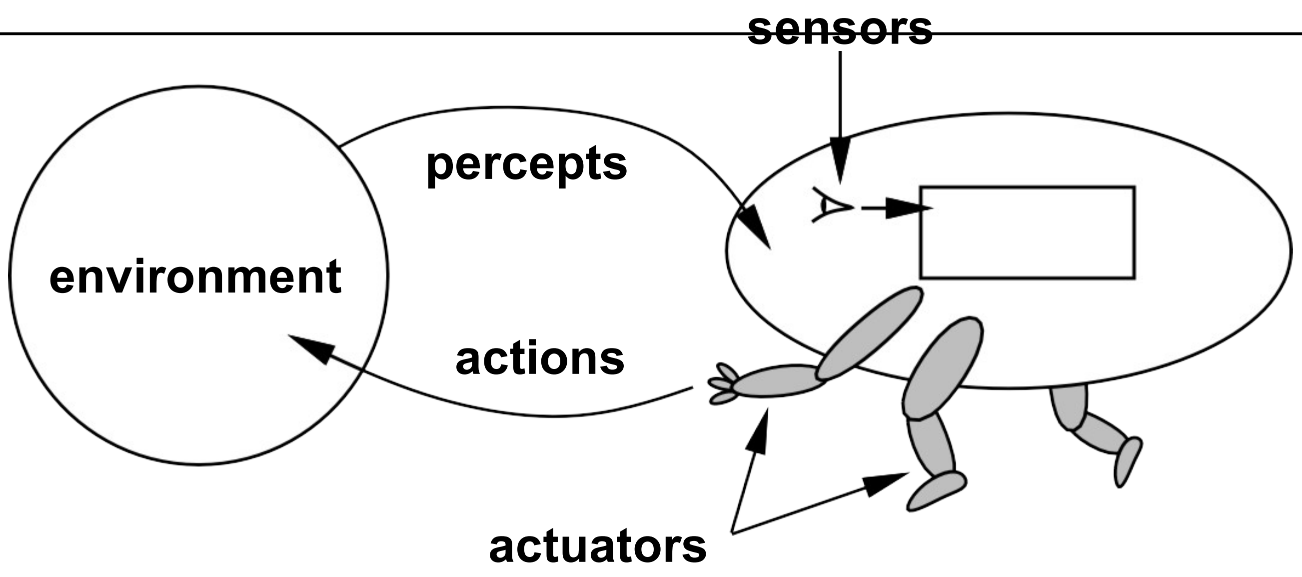
- Agents and environments
- Types of agents
- Representing problems
- Selecting a state space
- Tree search algorithms
- Breadth first search
- Depth first search

- Next steps

Goal

- Identify the concept of an intelligent agent.
- Develop a small set of design principles for building successful agents.
- Agents should be rational – one that does the right thing.
- Behaviour depends on the environment and the goals that we define for the agents.
- We define a number of basic agent designs.

Agents and environments



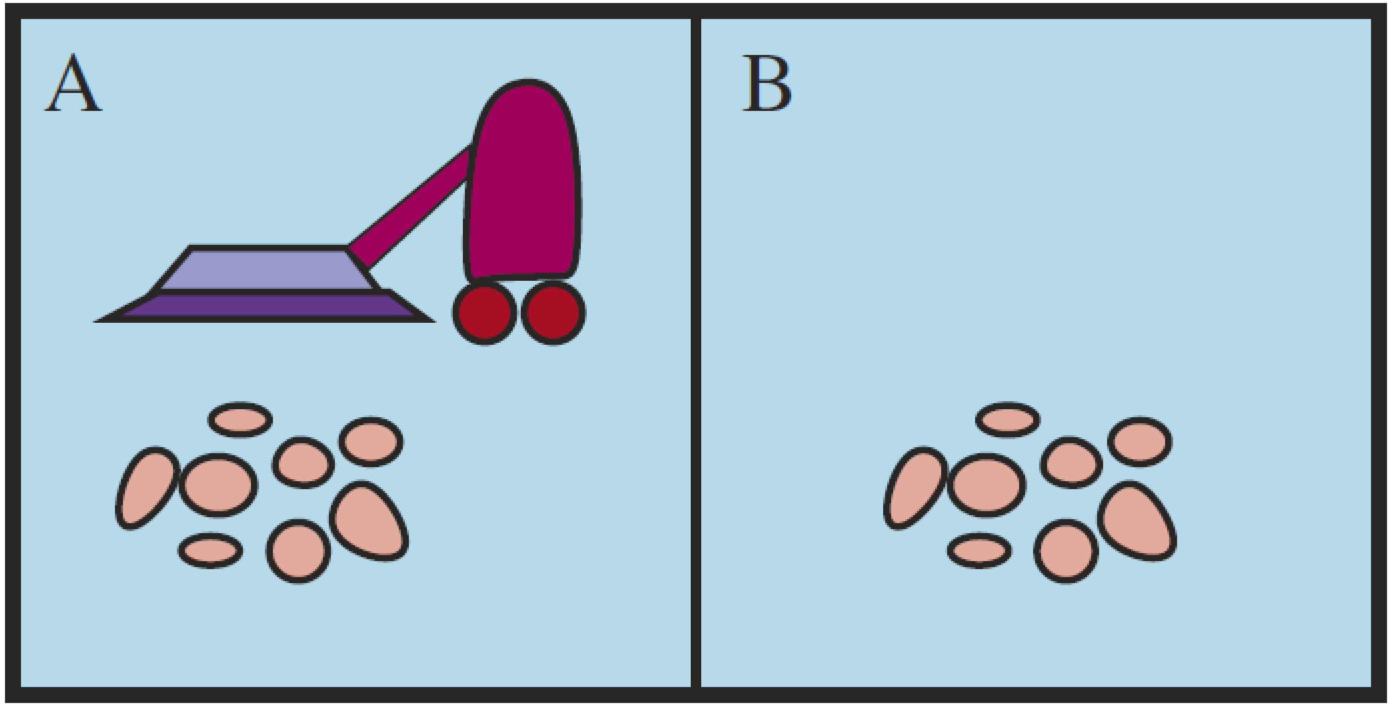
Agents include humans, robots, machines, etc.

The agent function maps from percept histories to actions:

$f : P^* \rightarrow A$

The agent program runs on the physical

Vacuum-cleaner world



Percepts: location and contents, e.g., [*A*, *Dirty*]

Actions: *Left*, *Right*, *Suck*, *N oOp*

A vacuum-cleaner agent

Percept sequence	Action
[*A*, *Clean*]	*Right*
[*A*, *Dirty*]	*Suck*
[*B*, *Clean*]	*Left*
[*B*, *Dirty*]	*Suck*
[*A*, *Clean*], [*A*, *Clean*]	*Right*
[*A*, *Clean*], [*A*, *Dirty*]	*Suck*
.	.

function Reflex-Vacuum-Agent([*location*,*status*]) returns an action

if *status* = *Dirty* then return *Suck*
else if *location* = *A* then return
Right else if *location* = *B* then
return *Left*

What is the right function?
Can it be implemented in a small agent

Rationality

Fixed performance measure evaluates the environment sequence

- one point per square cleaned up in time T ?
- one point per clean square per time step, minus one per move?
- penalize for $> k$ dirty squares?

A rational agent chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date

Rational /= omniscient

- percepts may not supply all relevant information

Rational /= clairvoyant

- action outcomes may not be as expected

Hence, rational /= successful

Rational ⇒ exploration, learning

PEAS

To design a rational agent, we must specify the task environment

Consider, e.g., the task of designing an automated taxi:

Performance measure: safety, destination, profits, legality, comfort, . . .

Environment: streets/freeways, traffic, pedestrians, weather, . . .

Actuators: steering, accelerator, brake, horn, speaker/display, . . .

Sensors: video, accelerometers, engine sensors, keyboard, GPS, . . .

Internet shopping agent

Performance measure: price, quality, appropriateness, efficiency

Environment: current and future WWW sites, vendors, shippers

Actuators: display to user, follow URL, fill in form

Sensors: HTML pages (text, graphics, scripts)

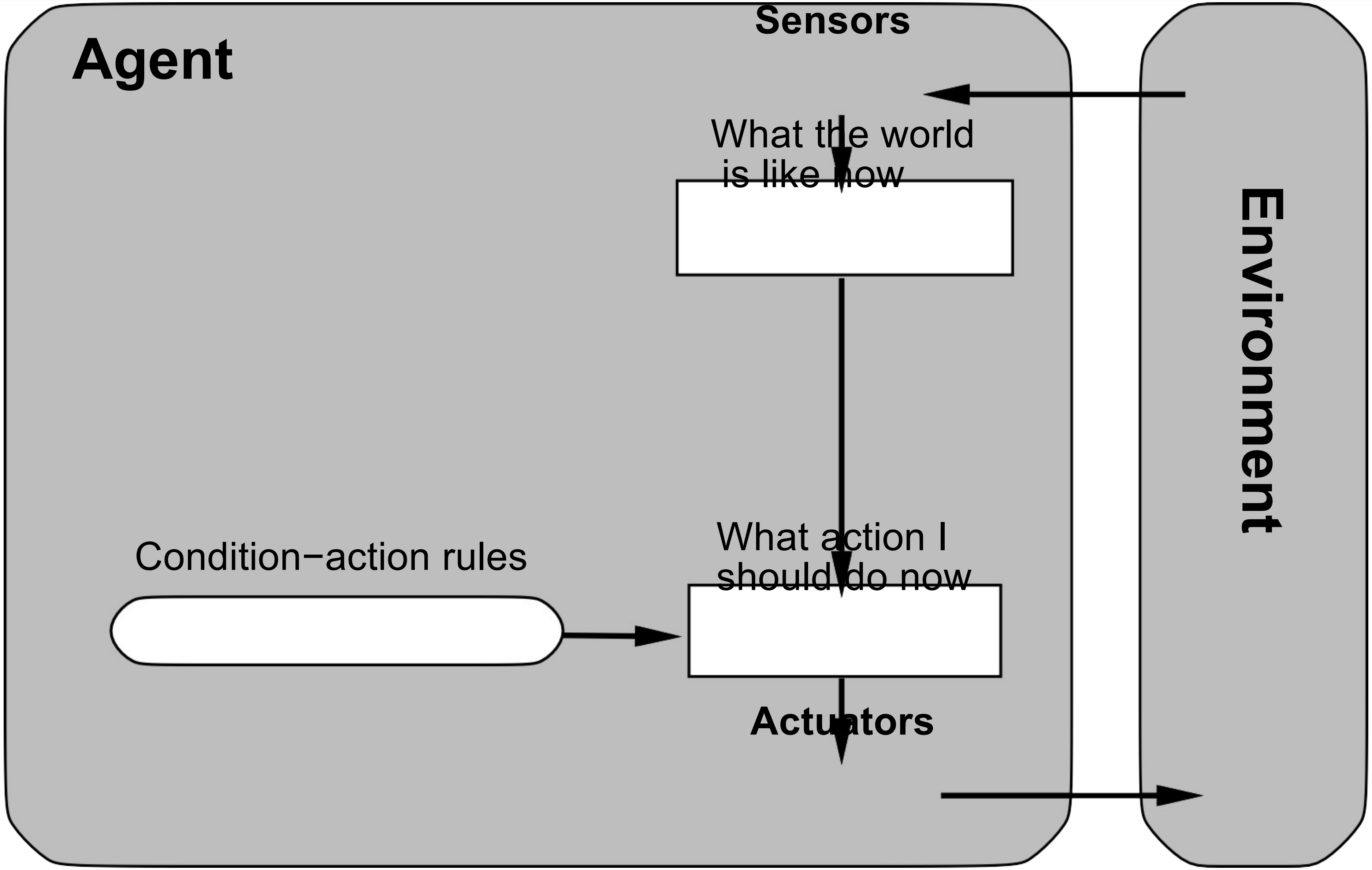
Environment types

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

The environment type largely determines the agent design

The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

Simple reflex agents



Example

function Reflex-Vacuum-Agent([*location*,*status*]) returns an action

if *status* = *Dirty* then return *Suck*
else if *location* = *A* then return
Right else if *location* = *B* then
return *Left*

Use example from Jupyter lab

Create example from the vacuum environment

Problem-solving agents

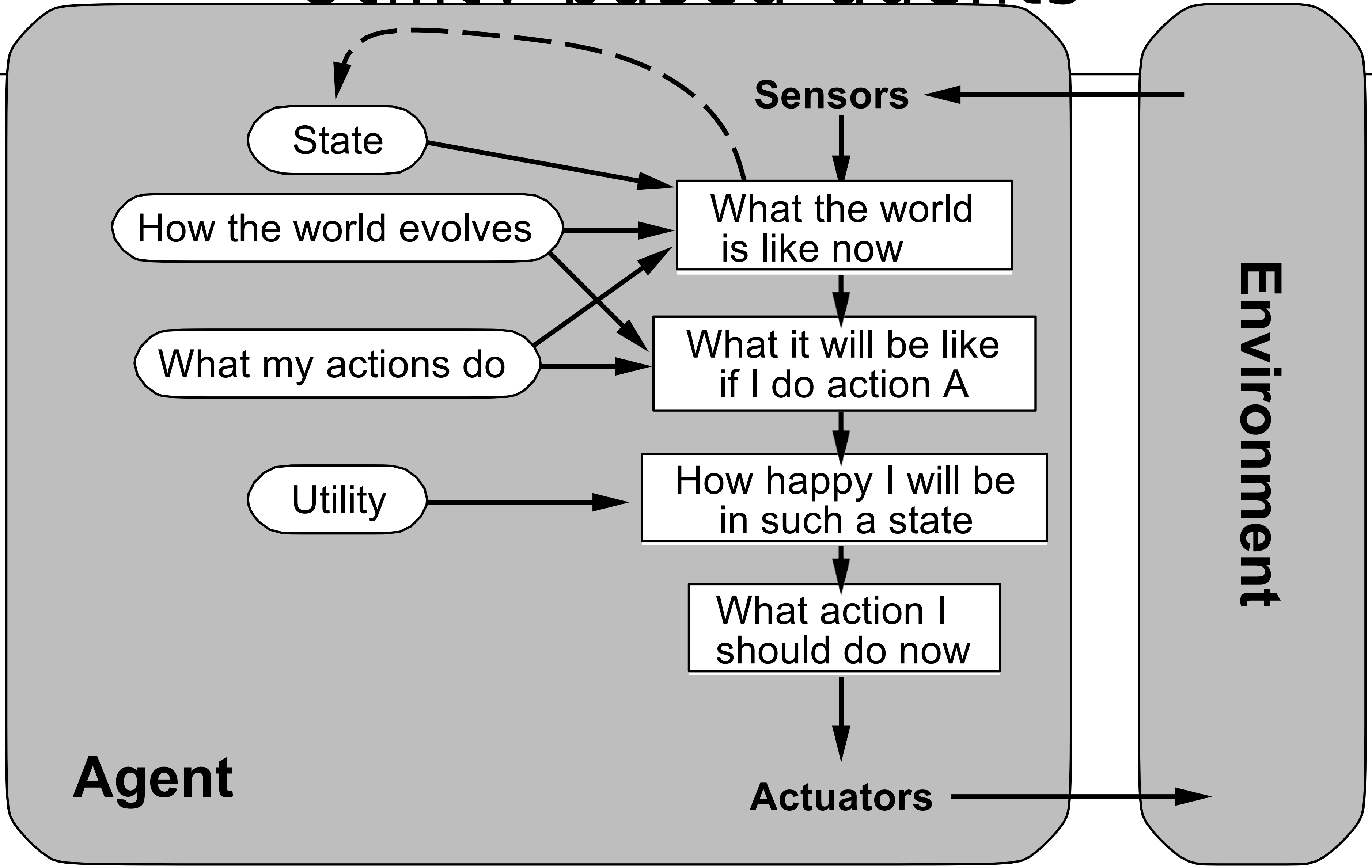
Restricted form of general agent:

```
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

Utility-based agents



Summary

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

PEAS descriptions define task environments

Environments are categorized along several dimensions:

Example: Romania

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

Formulate goal:

be in Bucharest

Formulate problem:

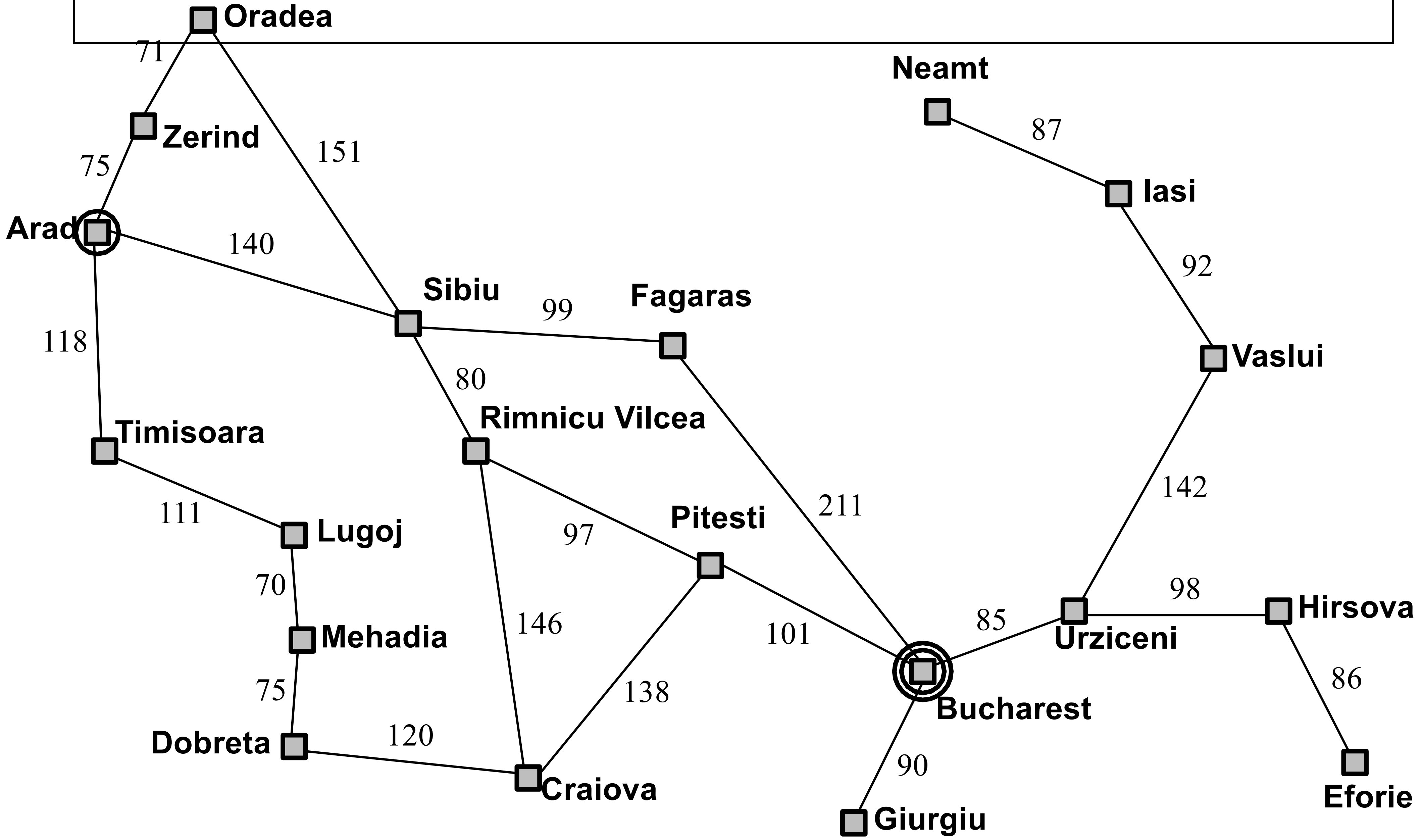
states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem types

Deterministic, fully observable =⇒ single-state problem

Agent knows exactly which state it will be in; solution is a sequence

Non-observable =⇒ conformant problem

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable =⇒

contingency problem percepts provide new information about current state

solution is a contingent plan or a policy

often interleave search, execution

Unknown state space =⇒ 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., $S(Arad) = \{(Arad \rightarrow Zerind, Zerind), \ldots\}$

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

Selecting a state space

Real world is absurdly complex

⇒ state space must be abstracted for problem solving

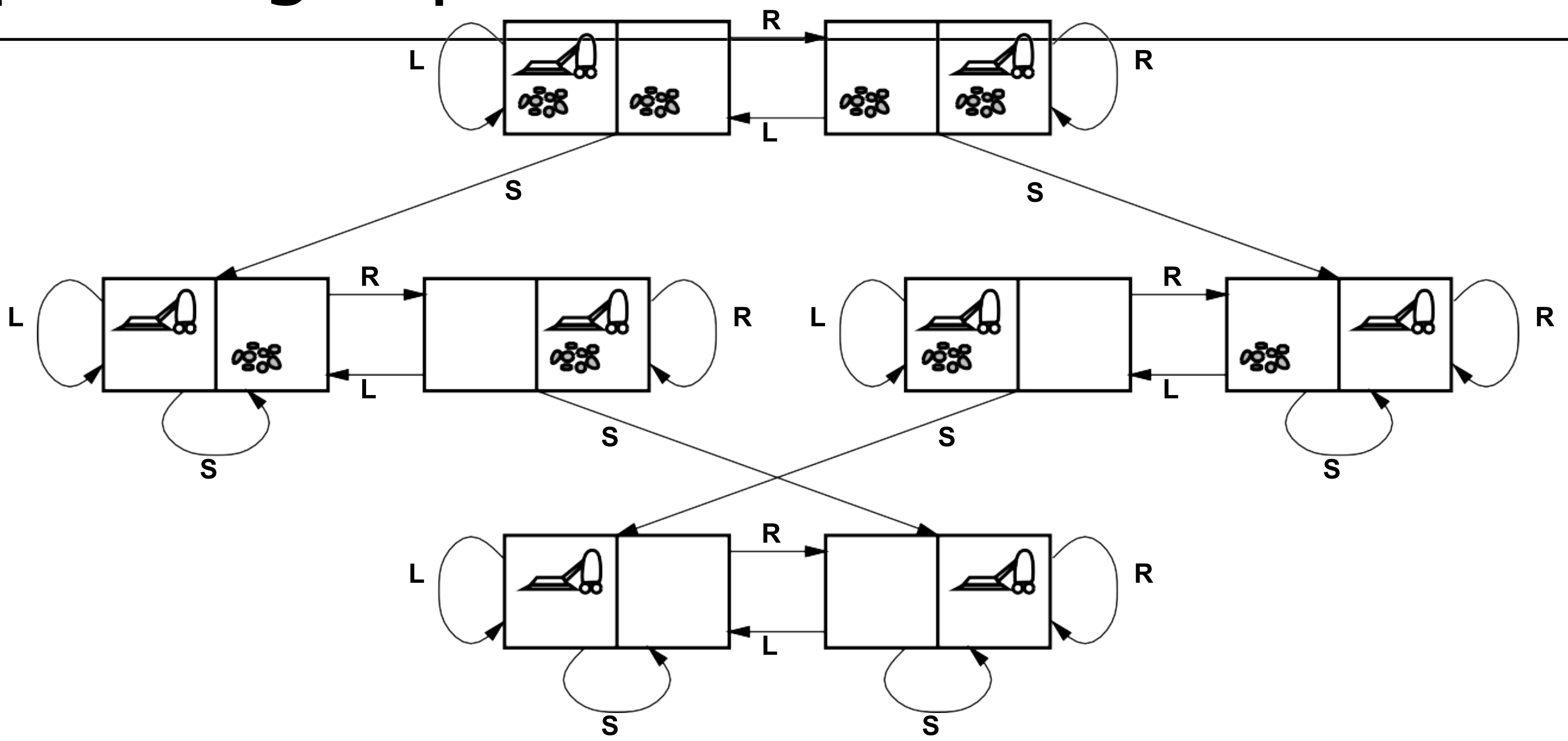
(Abstract) state = set of real states

(Abstract) action = complex combination of real actions e.g., “Arad → 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 =

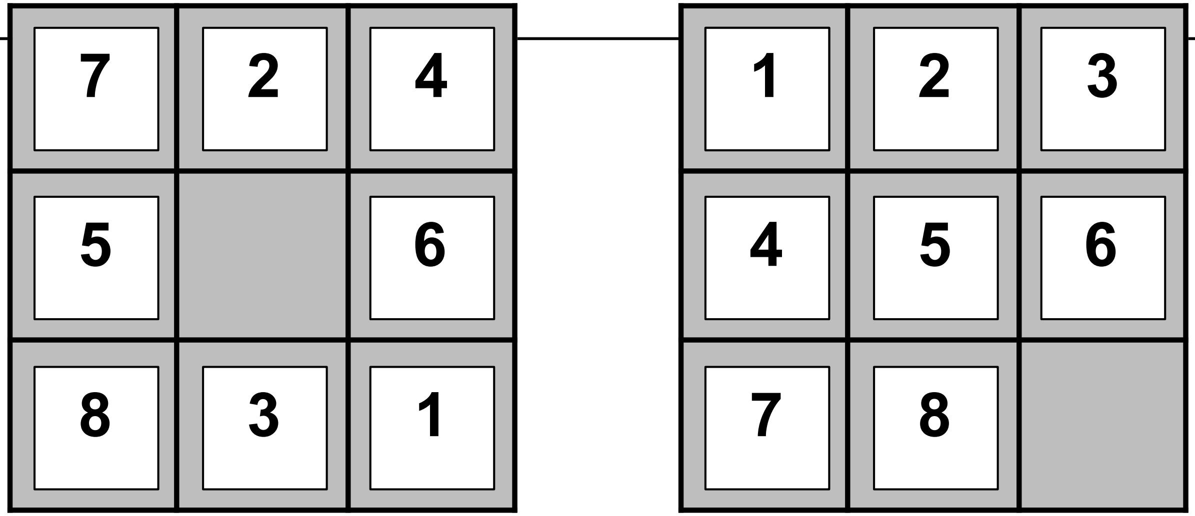
set of real paths that are

Example: vacuum world state space graph



states: integer dirt and robot locations (ignore dirt amounts etc.) actions: *Left*, *Right*, *Suck*, *NoOp*
goal test: no dirt
path cost: 1 per action (0 for *NoOp*)

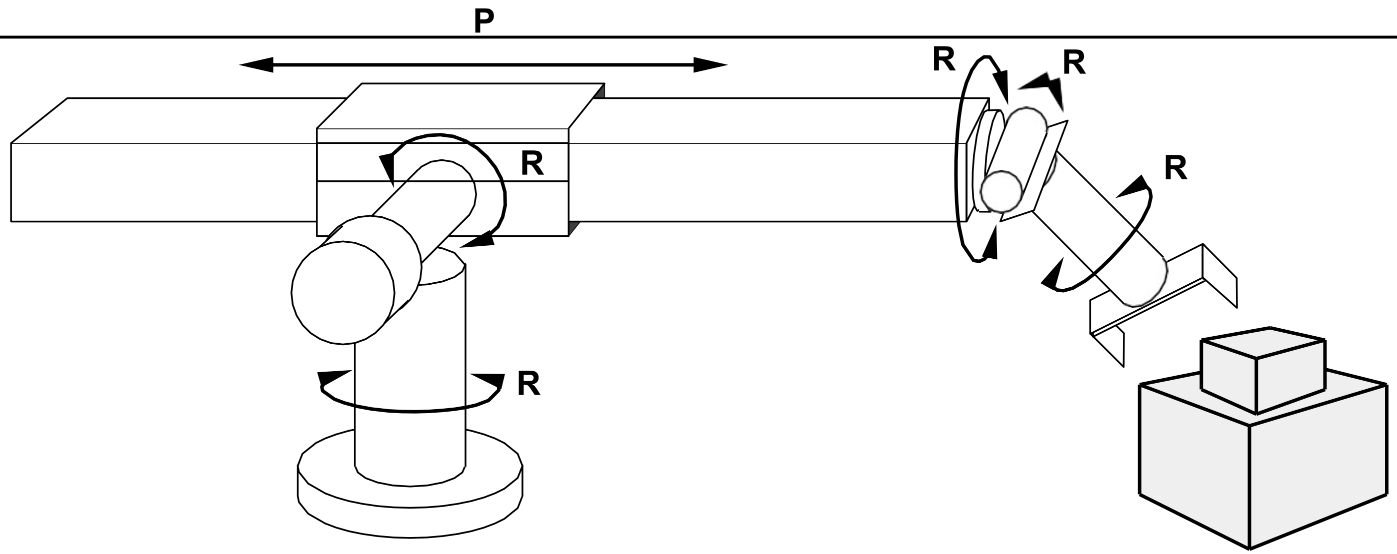
Example: The 8-puzzle



Start State
Goal State

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

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

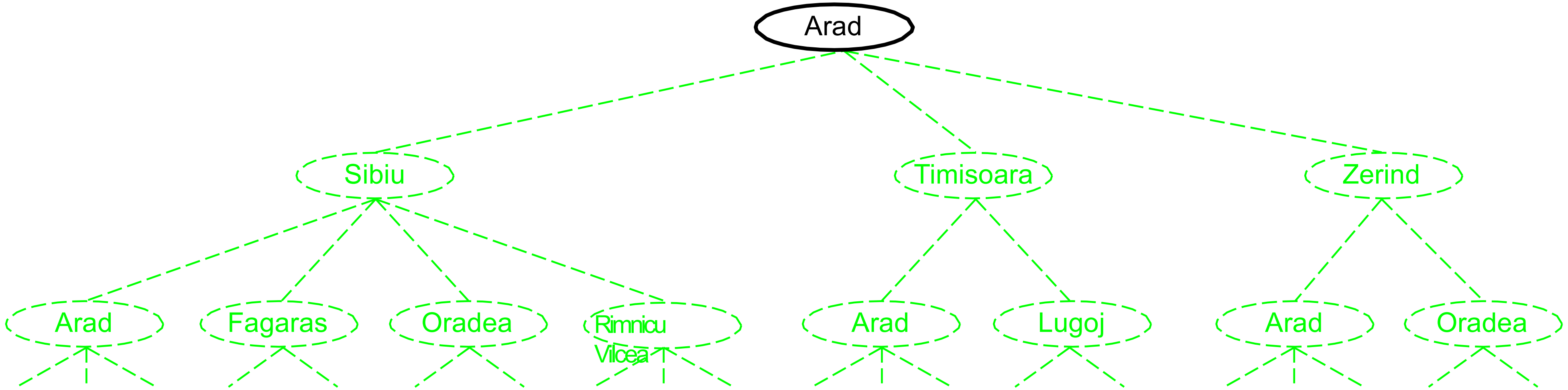
Basic idea:

offline, simulated exploration of state space

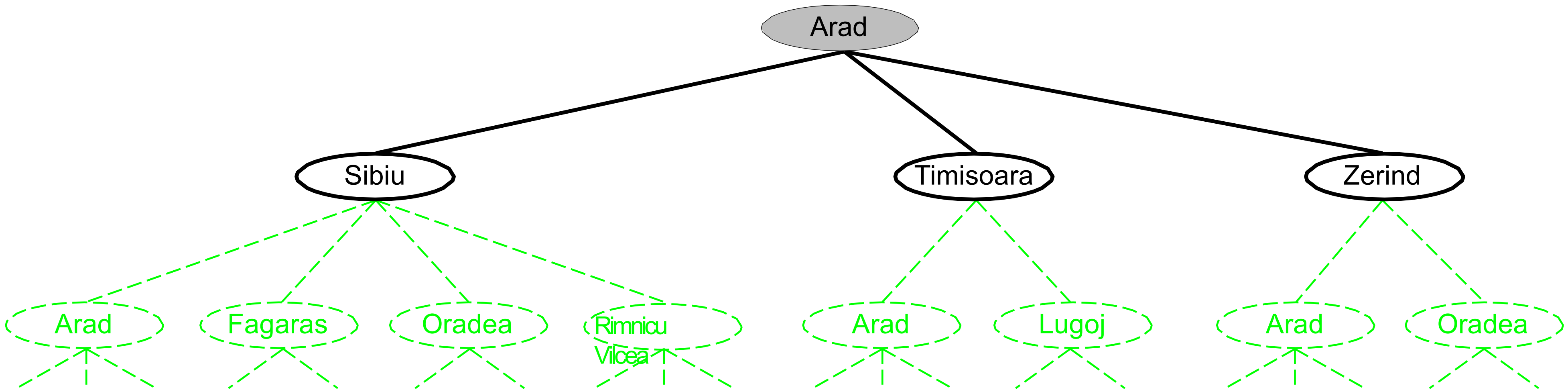
by generating successors of already-explored states (a.k.a. expanding states)

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

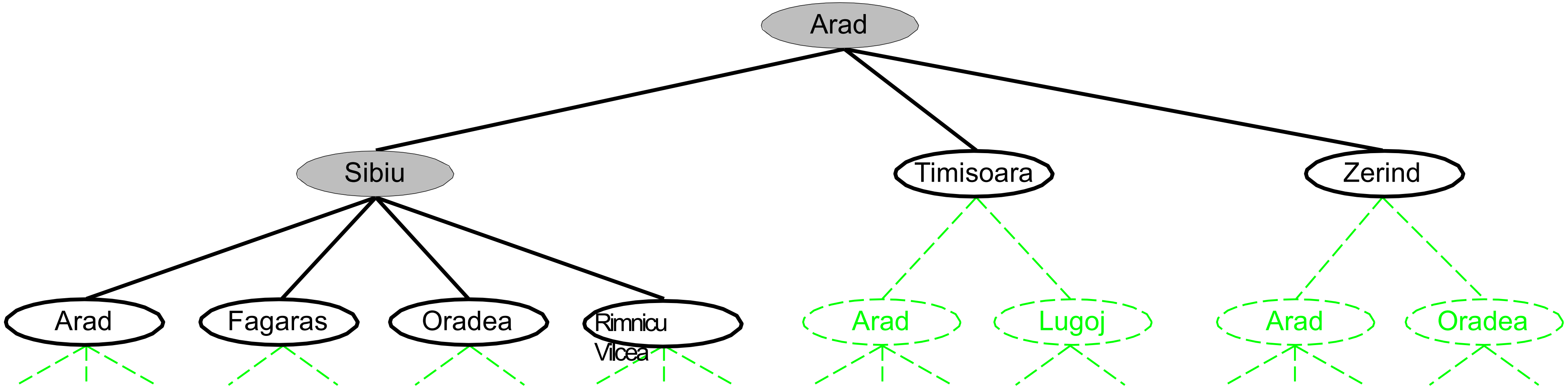
Tree search example



Tree search example



Tree search example

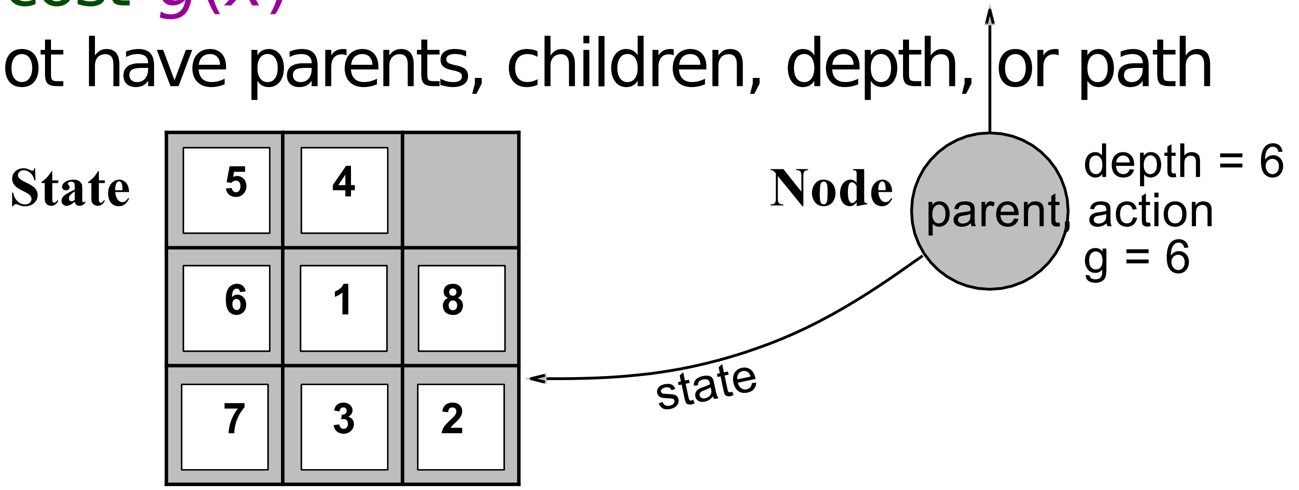


Implementation: states vs. nodes

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;
```

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

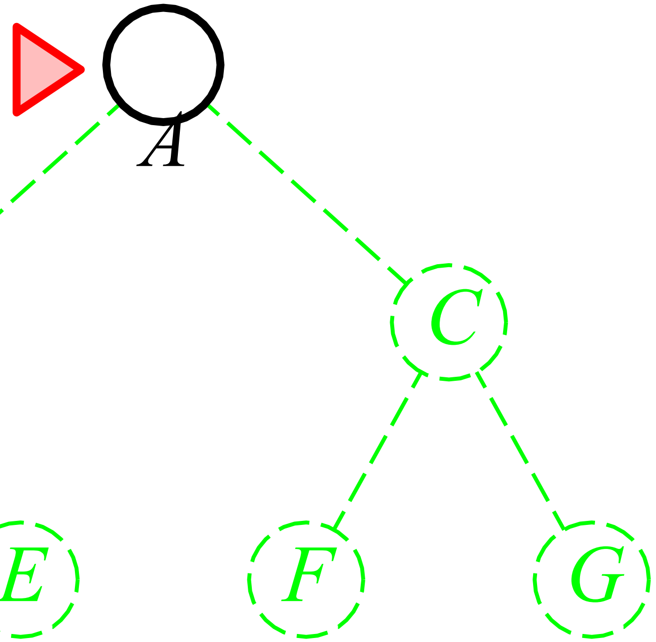
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

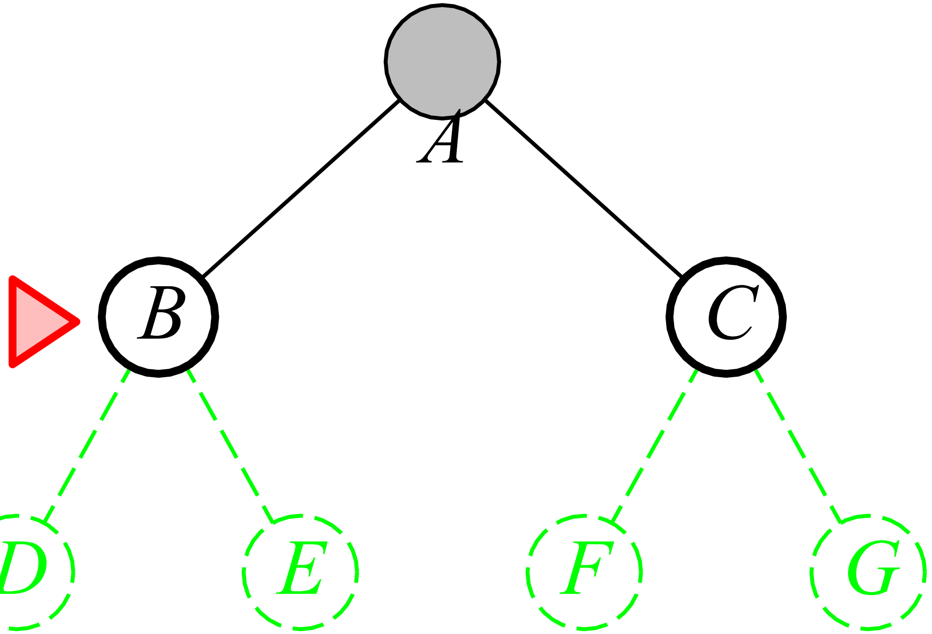


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

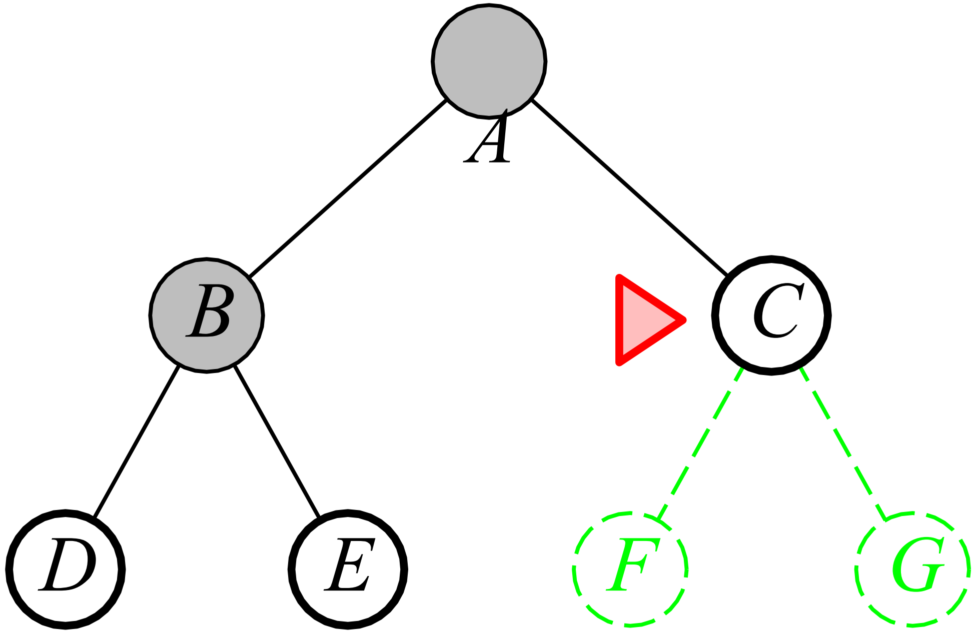


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

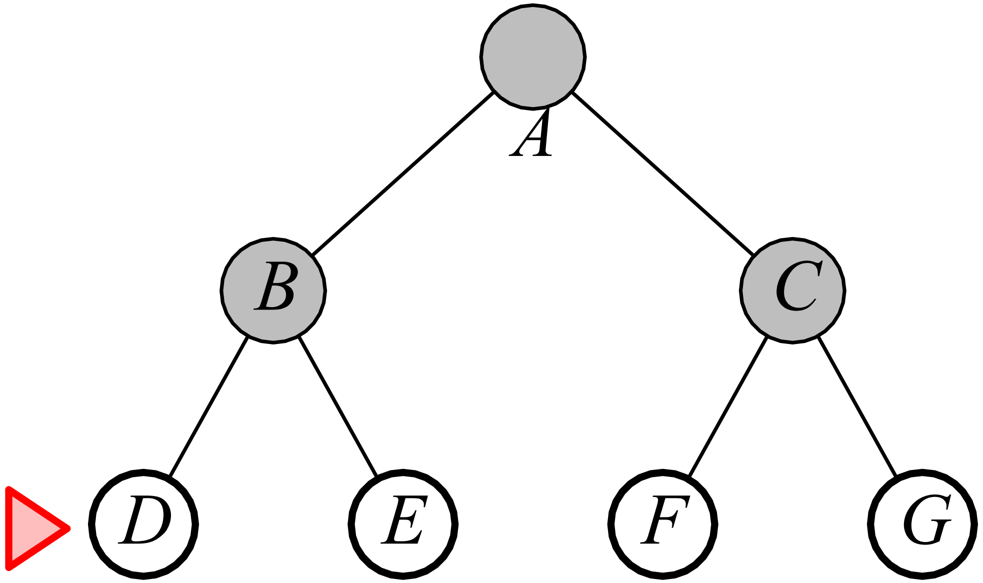


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

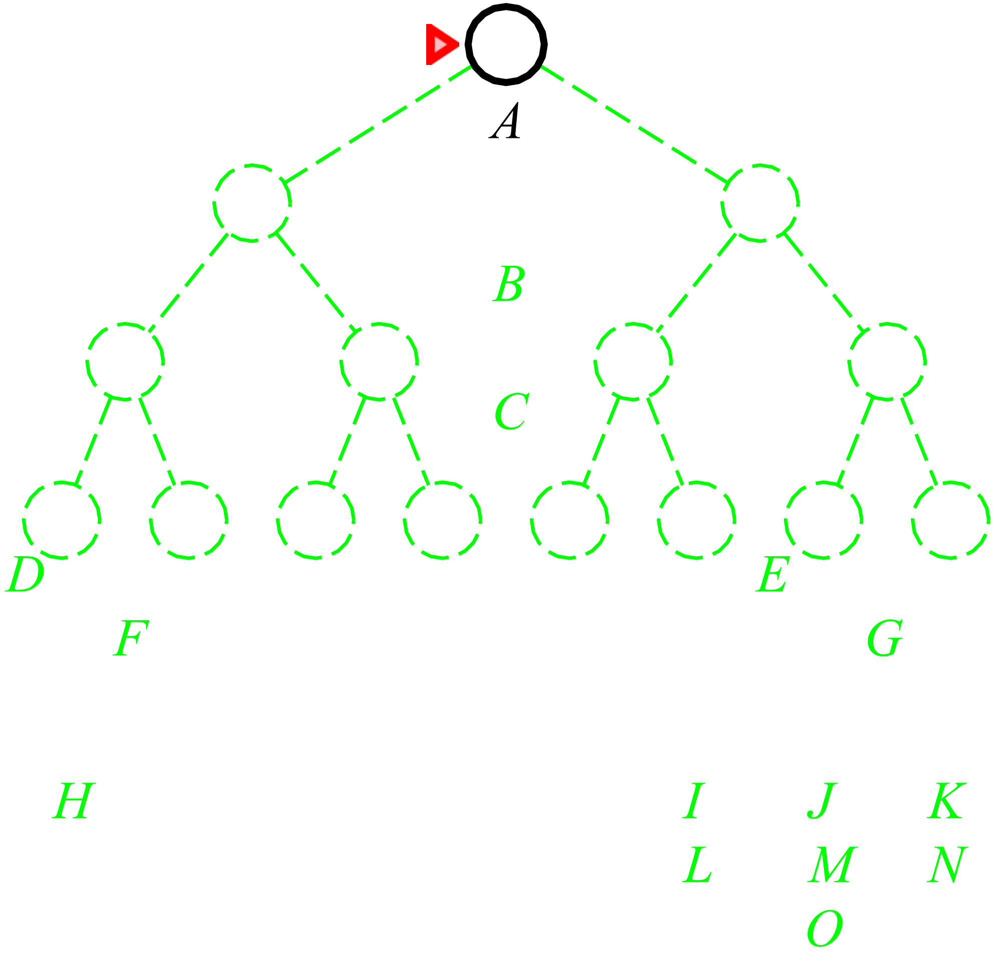


Depth-first search

Expand deepest unexpanded node

Implementation:

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

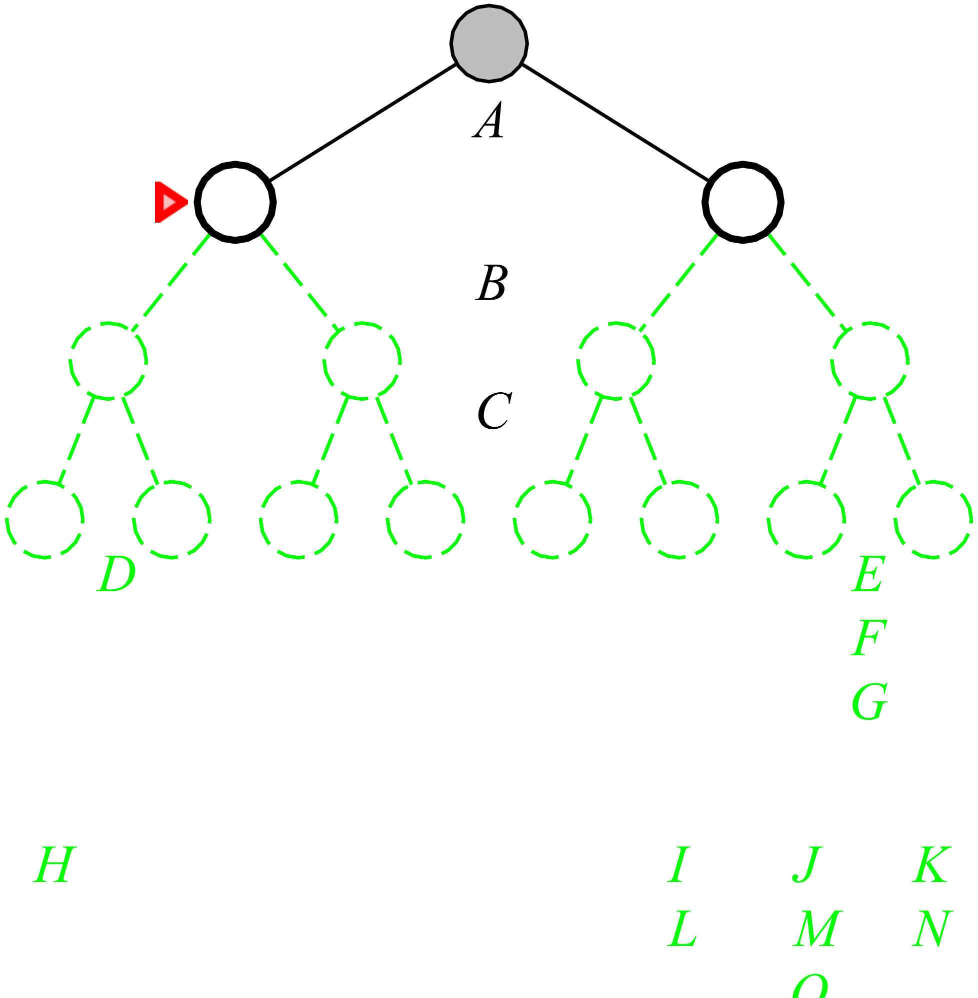


Depth-first search

Expand deepest unexpanded node

Implementation:

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

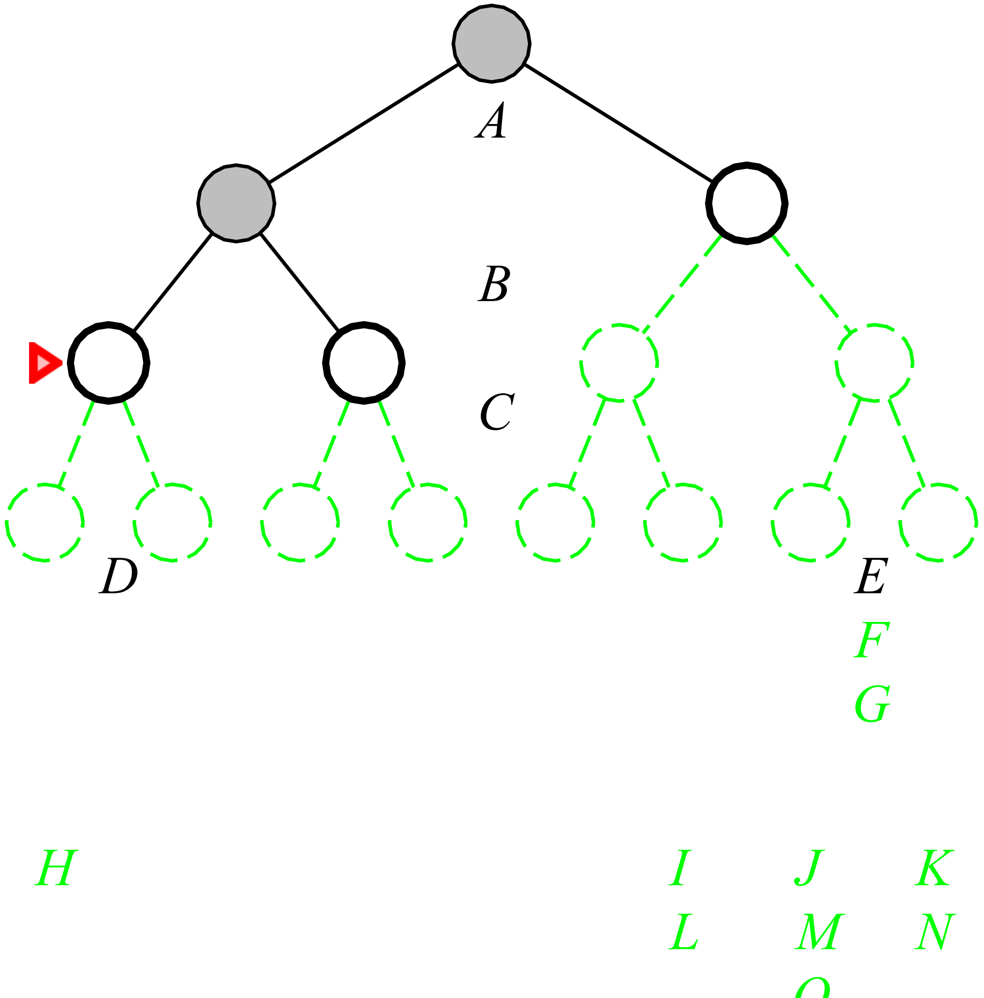


Depth-first search

Expand deepest unexpanded node

Implementation:

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

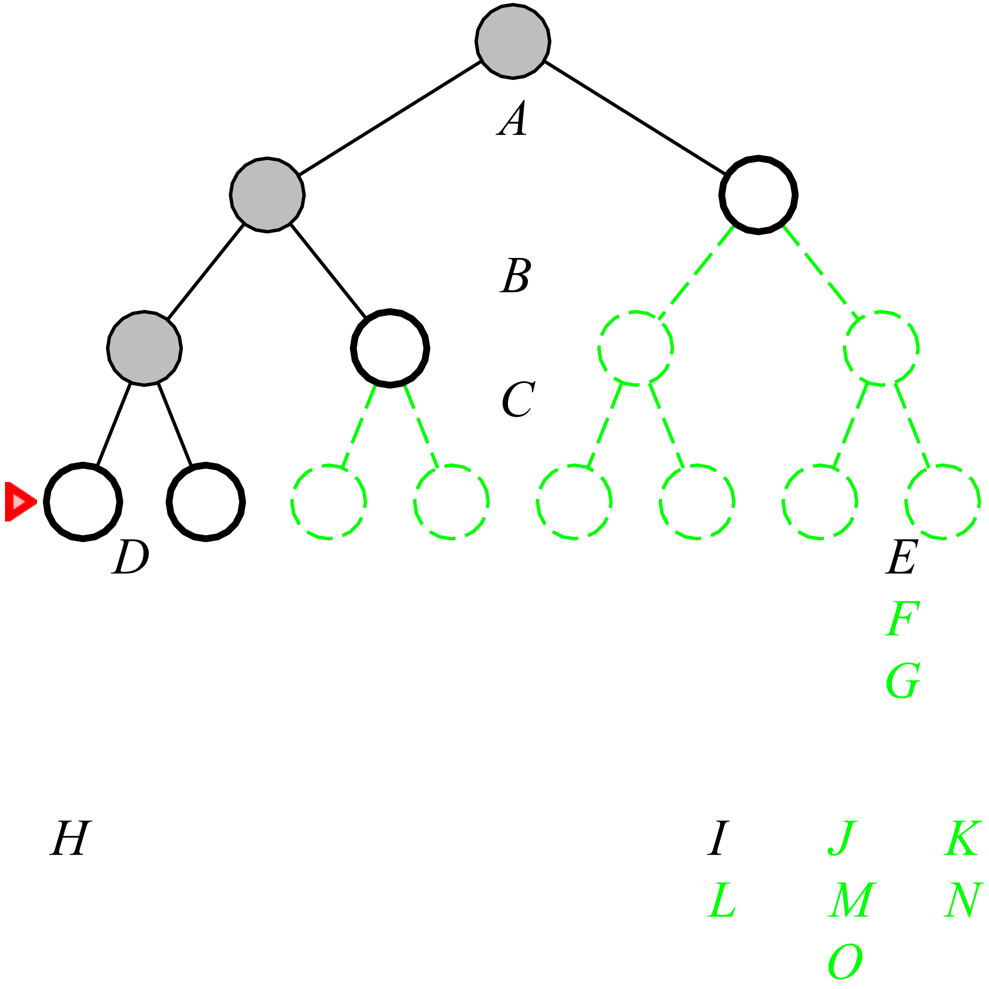


Depth-first search

Expand deepest unexpanded node

Implementation:

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

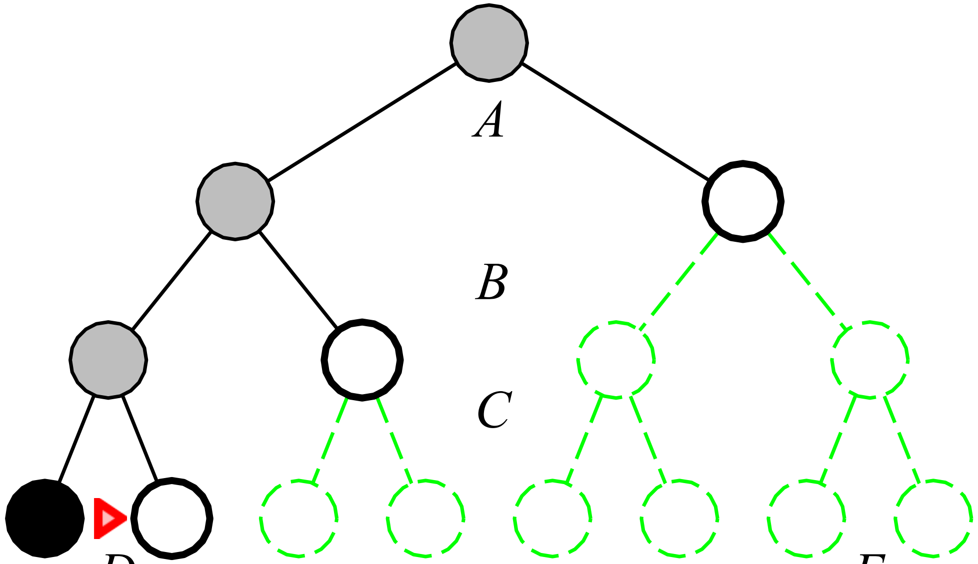


Depth-first search

Expand deepest unexpanded node

Implementation:

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

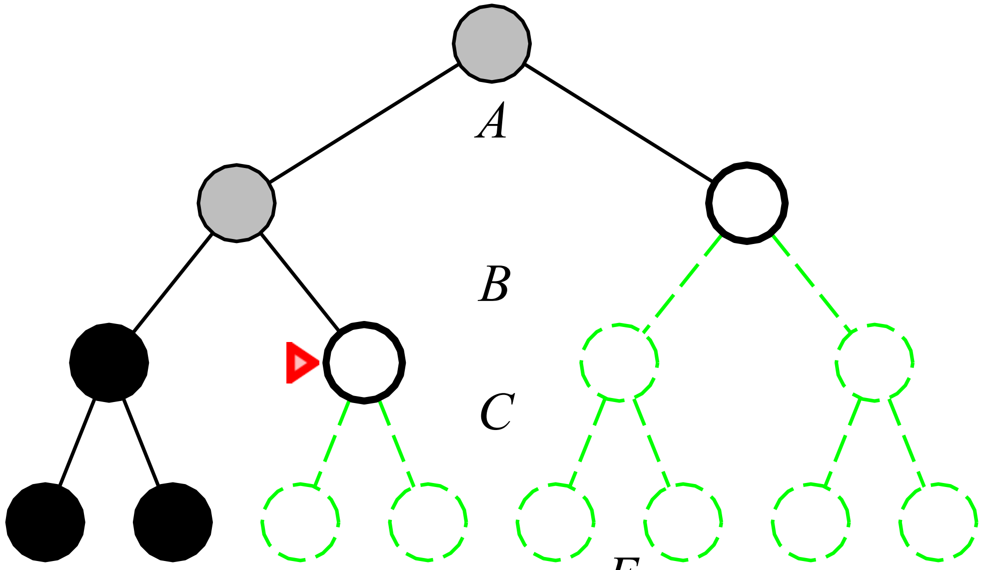


Depth-first search

Expand deepest unexpanded node

Implementation:

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



Questions Discussion?