

SOLVING PROBLEMS BY SEARCHING

AIMA2E CHAPTER 3, SECTIONS 1–5

Outline

- ◇ Problem-solving agents
- ◇ Problem types
- ◇ Problem formulation
- ◇ Example problems
- ◇ Basic search algorithms
- ◇ Avoiding repeated States
- ◇ Searching with partial information

Why Search?

- Implementation of **goal-based agents program**.

The program searches through agents action space to find the most appropriate set of actions that leads agent to its goal by minimum cost

⇒ **Problem-solving agent**.

- We need to define **Problem** and **Solution**

Search Types

- **Uninformed:**
The agent has no information about the underlying problem other than its definition.
- **Informed:**
The agent have some idea of where to look for solutions

Basic Definitions

- **Goal formulation:**

Defining the goal states along the performance measure.

- **Problem formulation:**

Deciding what actions and state to consider (level of abstraction).

It may vary during the problem-solving process from fine-to-coarse and vice versa.

Search and Solution

- **Search:** An agent with several immediate options of unknown values can decide what to do by first examining different possible sequence of actions that lead to states of known values, and then choosing the best sequence.
- **Solution:** Selected sequence of actions
- **Execution:** Once a solution is found, the actions it recommends can be carried out.
- **Problem-solving:**
Goal definition → Problem definition → Search → Solution

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.

Environment for problem-solving agents

- ◇ Static
- ◇ Observable
- ◇ Discrete
- ◇ Deterministic

Solutions are executed without paying attention to the percepts

⇒ **open-loop system**

Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

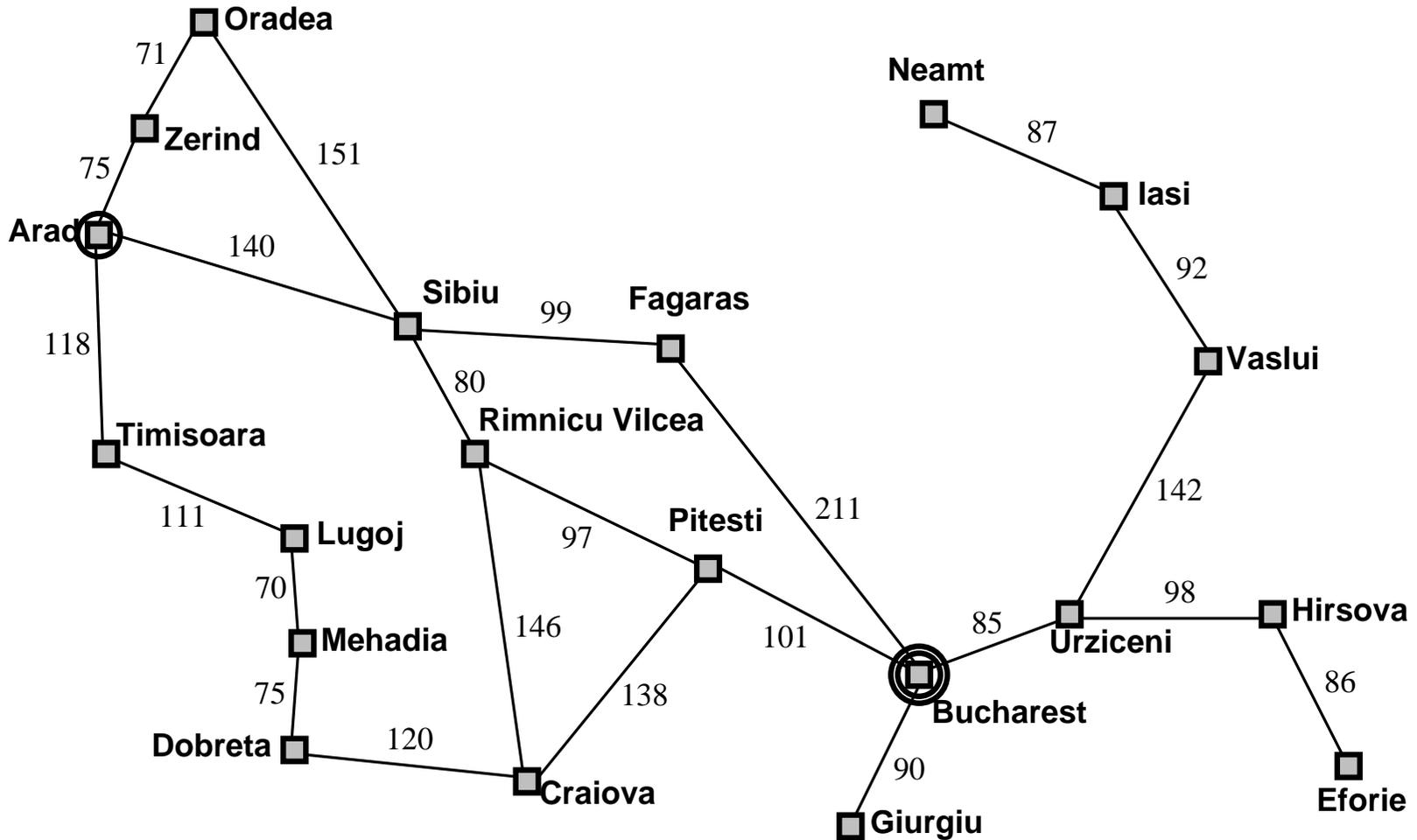
states: various cities

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

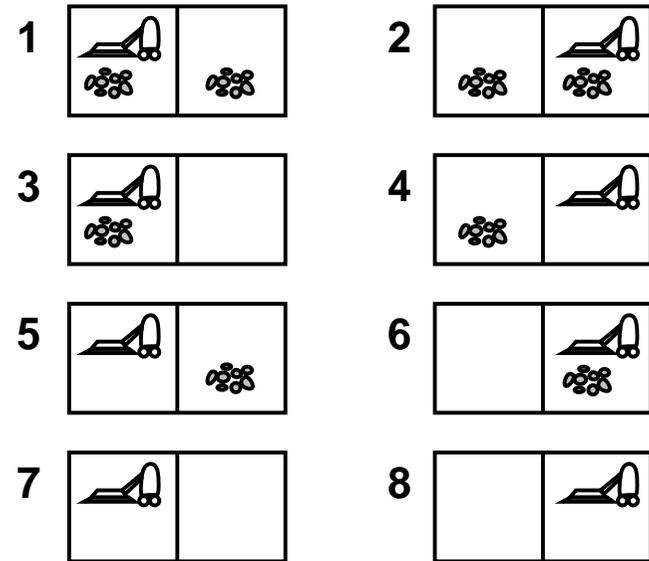
solution is a *tree* or *policy*

often *interleave* search, execution

Unknown state space \implies *exploration problem* (“online”)

Example: vacuum world

Single-state, start in #5. [Solution??](#)



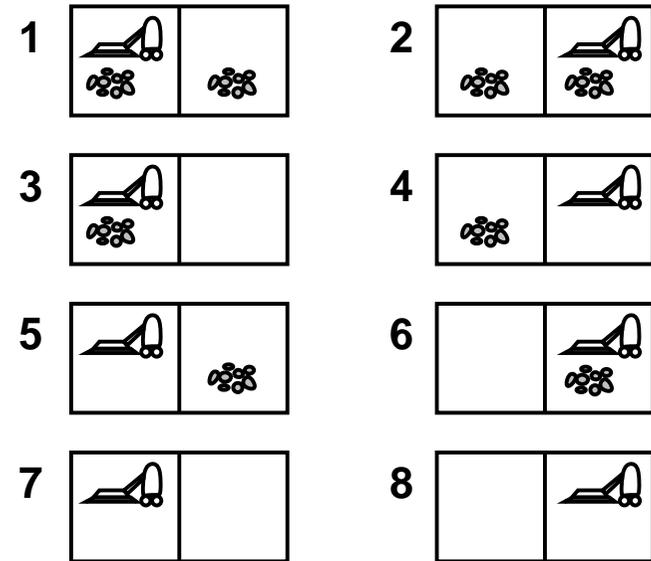
Example: vacuum world

Single-state, start in #5. **Solution??**

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}. **Solution??**



Example: vacuum world

Single-state, start in #5. [Solution??](#)

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}. [Solution??](#)

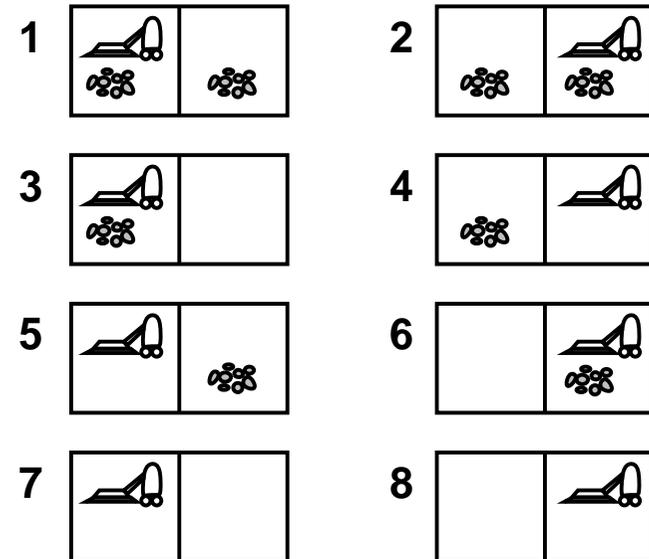
[*Right, Suck, Left, Suck*]

Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

Local sensing: dirt, location only.

[Solution??](#)



Example: vacuum world

Single-state, start in #5. **Solution??**

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}. **Solution??**

[*Right, Suck, Left, Suck*]

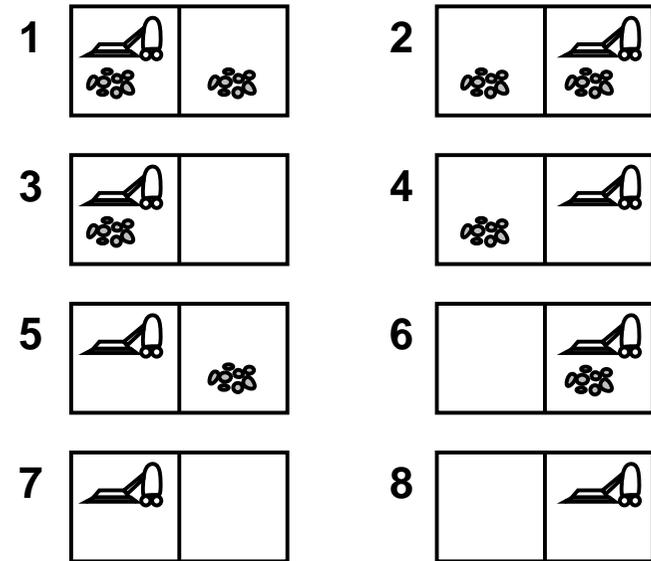
Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

Local sensing: dirt, location only.

Solution??

[*Right, if dirt then Suck*]



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., $S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots\}$

goal test, can be

explicit, e.g., $x = \text{“at Bucharest”}$

implicit, e.g., $\text{NoDirt}(x)$

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

Optimal solution has the lowest path cost among all solutions.

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 =

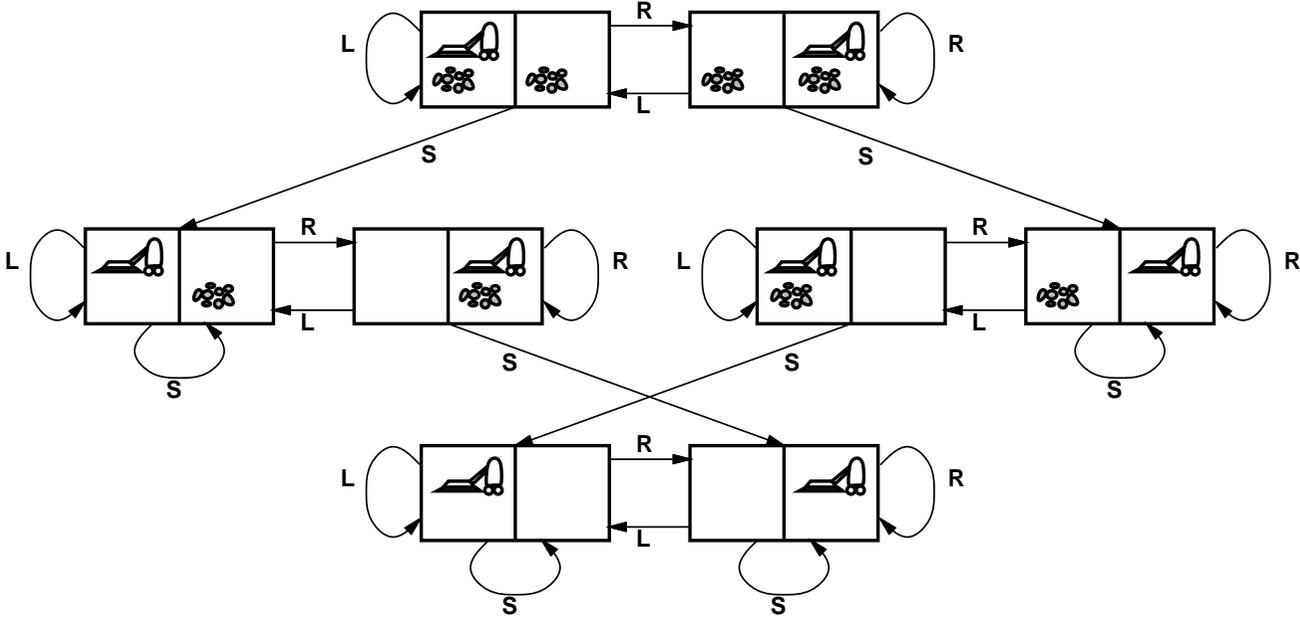
set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!

Sample Problems

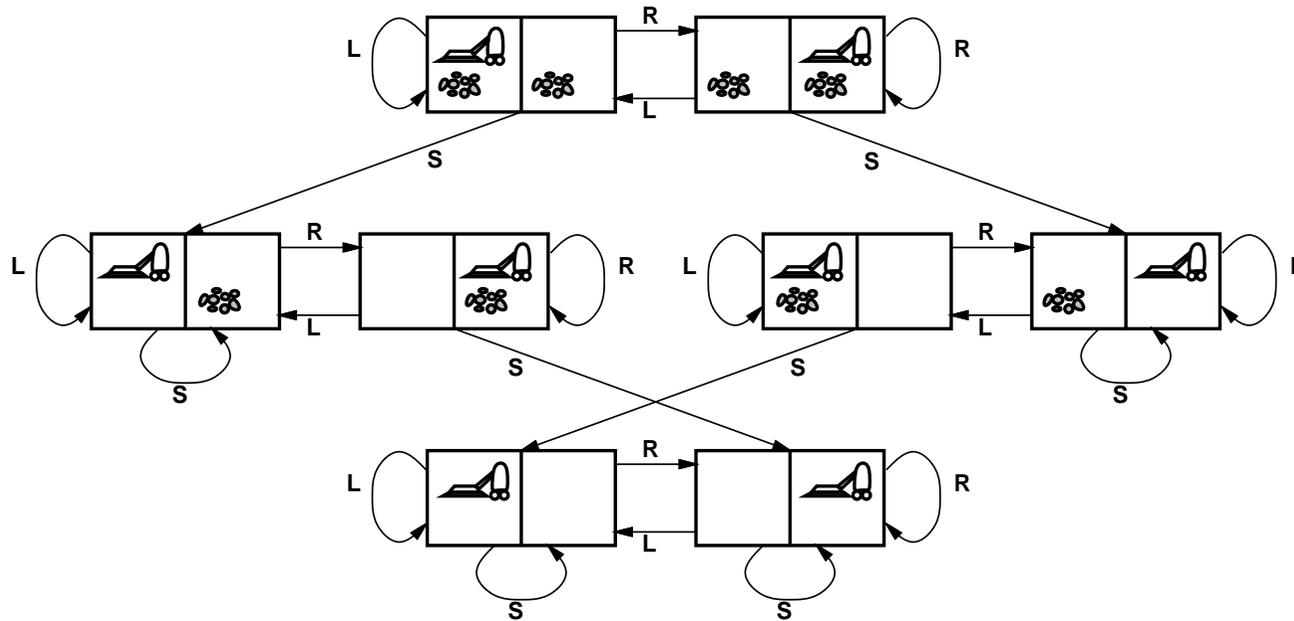
- **Toy problems**
- **Real world problems**

Example: vacuum world state space graph



states??
actions??
goal test??
path cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt *amounts*)

actions??: *Left, Right, Suck, NoOp*

goal test??: no dirt

path cost??: 1 per action (0 for *NoOp*)

Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states??

actions??

goal test??

path cost??

Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states??: integer locations of tiles (ignore intermediate positions)

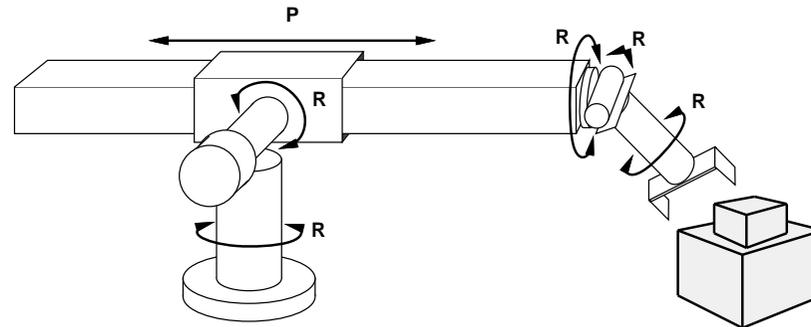
actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

[Note: optimal solution of n -Puzzle family is NP-hard]

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

Other Real World Problems

- **Route-finding problem**
- **Touring problems**
Travelling salesperson problem (TSP)
- **VLSI Layout**
Cell layout
Channel routing
- **Robot navigation**
- **Protein design**
- **Internet Searching**

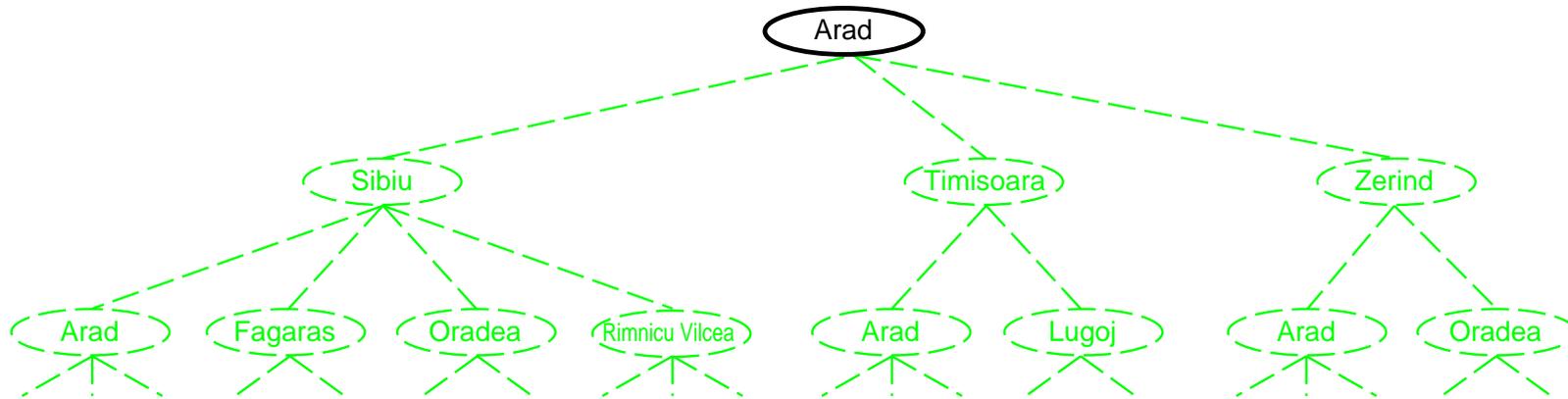
Tree search algorithms

Basic idea:

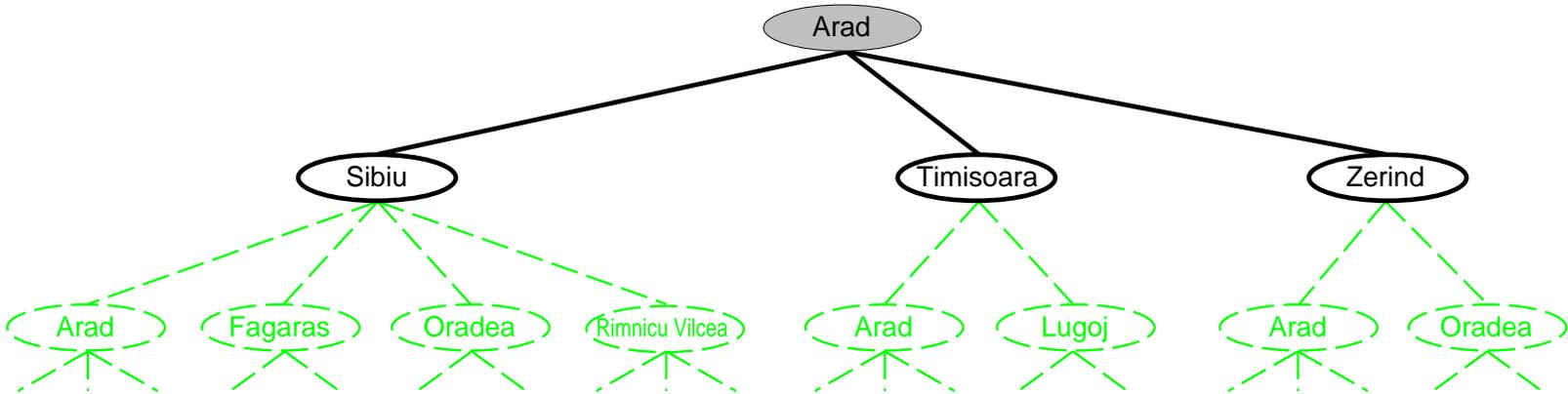
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

```
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
```

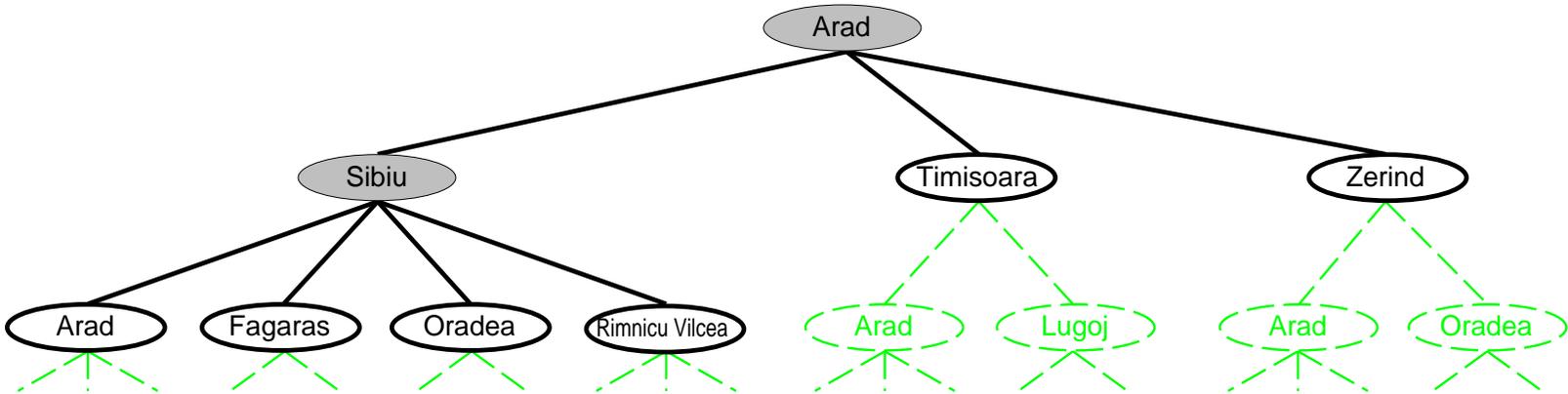
Tree search example



Tree search example



Tree search example



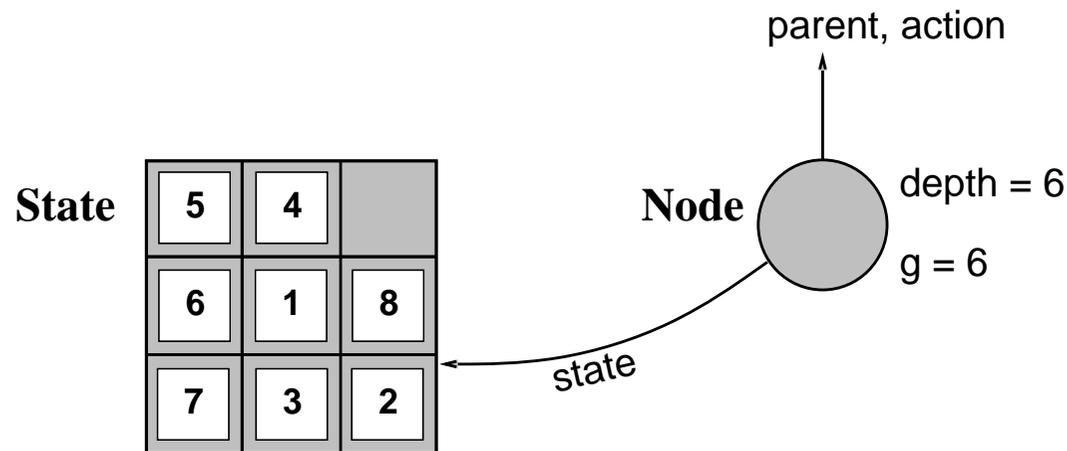
Implementation: states vs. nodes

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] applied to STATE(node) succeeds 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 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

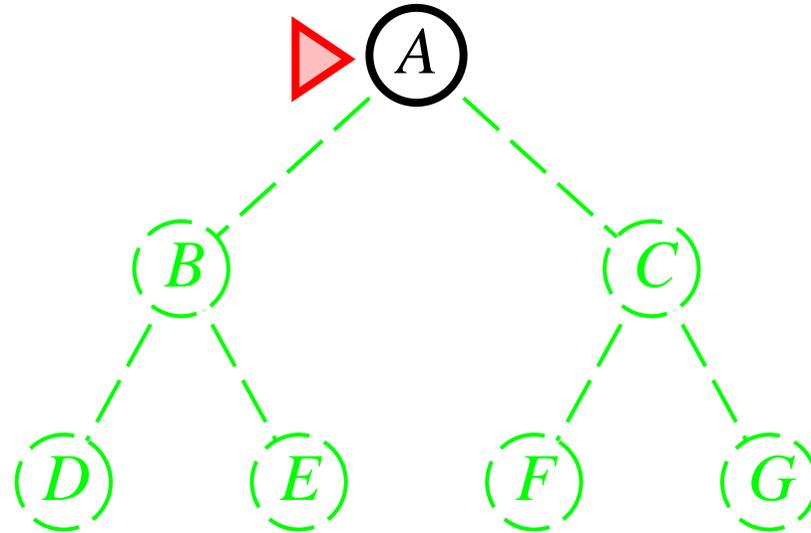
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

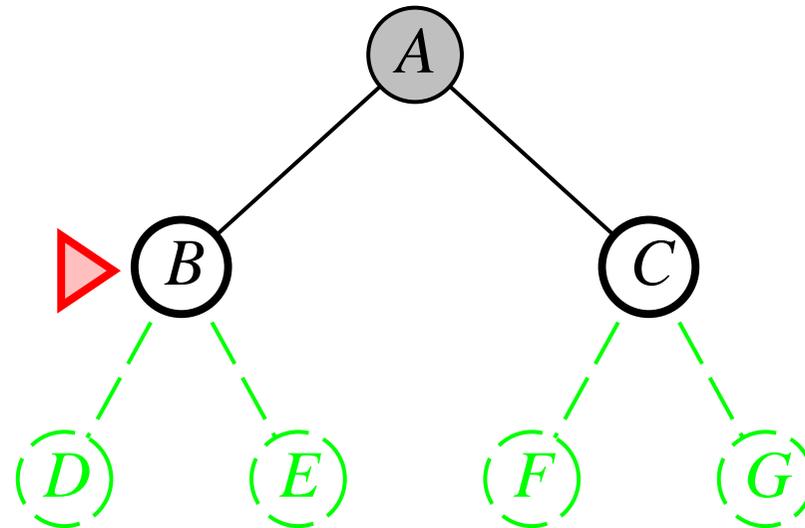


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

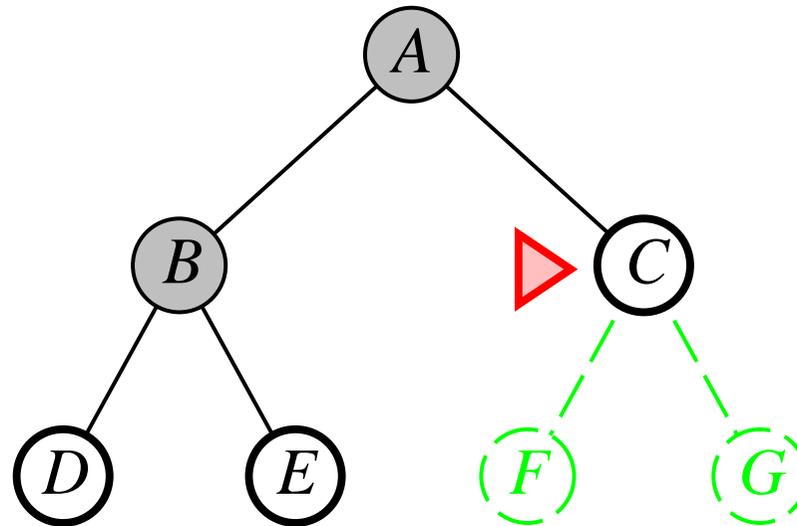


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

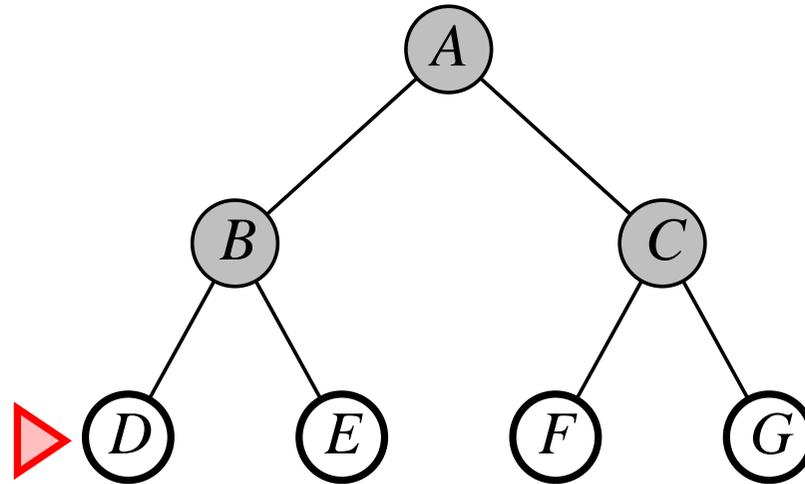


Breadth-first search

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

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

Optimal?? Yes (if path cost is a non-decreasing function of node depth e.g. cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 10MB/sec
so 24hrs = 860GB.

Uniform-cost search

Expand least-cost unexpanded node (a.k.a. **Least-cost search**)

Implementation:

fringe = queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon > 0$

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, $O(b^{\lceil C^*/\epsilon \rceil})$

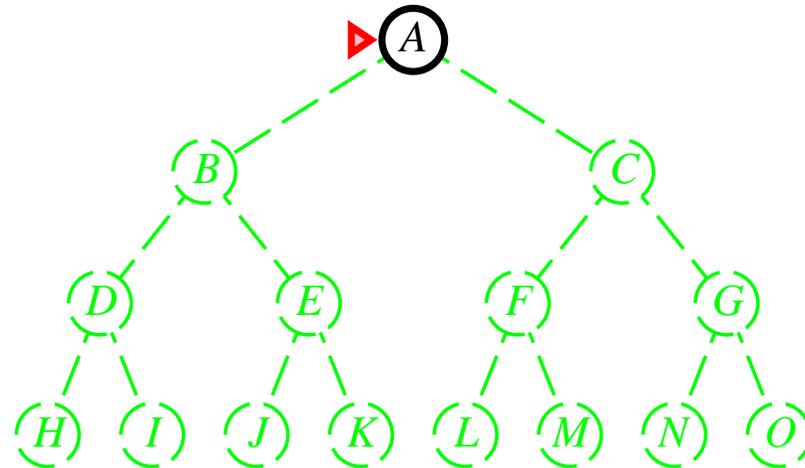
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

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

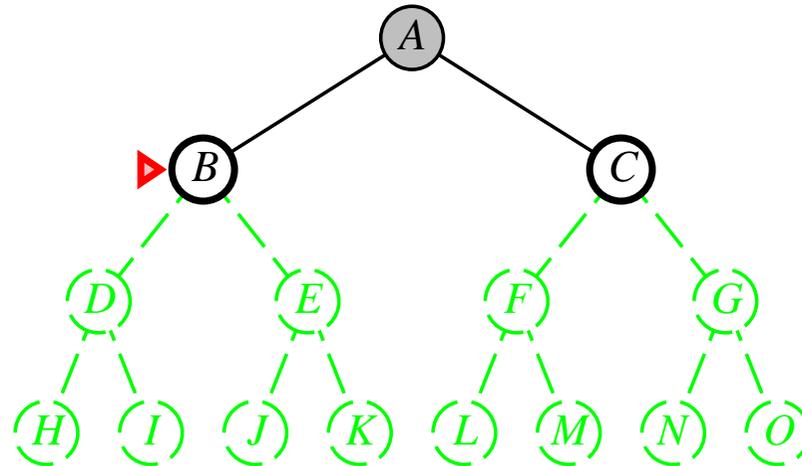


Depth-first search

Expand deepest unexpanded node

Implementation:

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

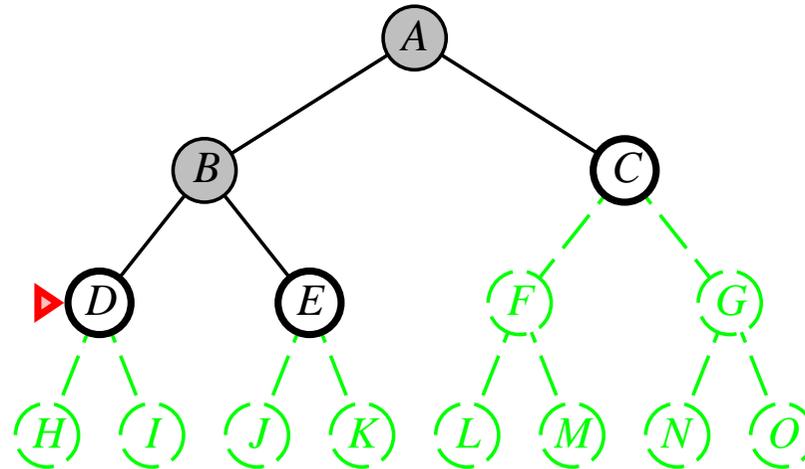


Depth-first search

Expand deepest unexpanded node

Implementation:

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

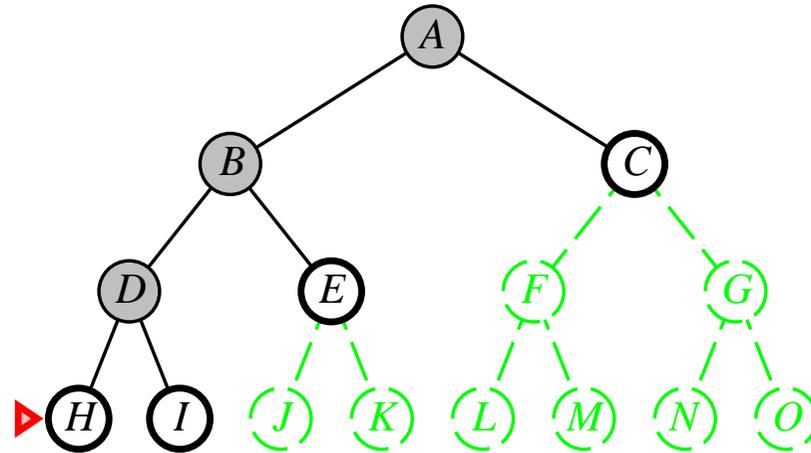


Depth-first search

Expand deepest unexpanded node

Implementation:

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

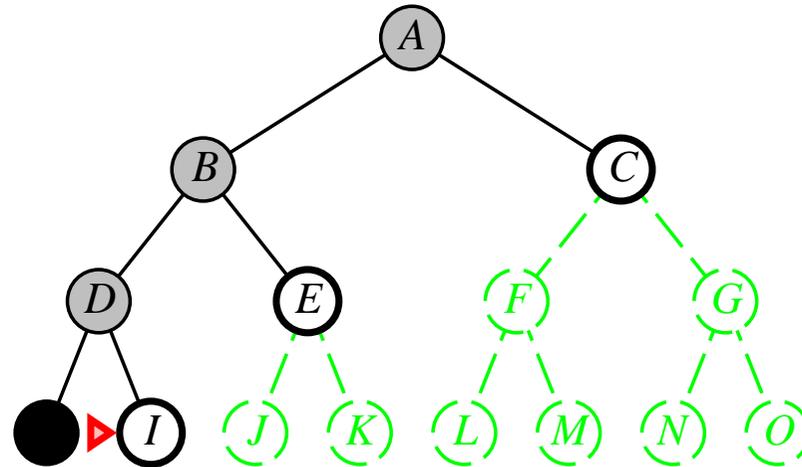


Depth-first search

Expand deepest unexpanded node

Implementation:

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

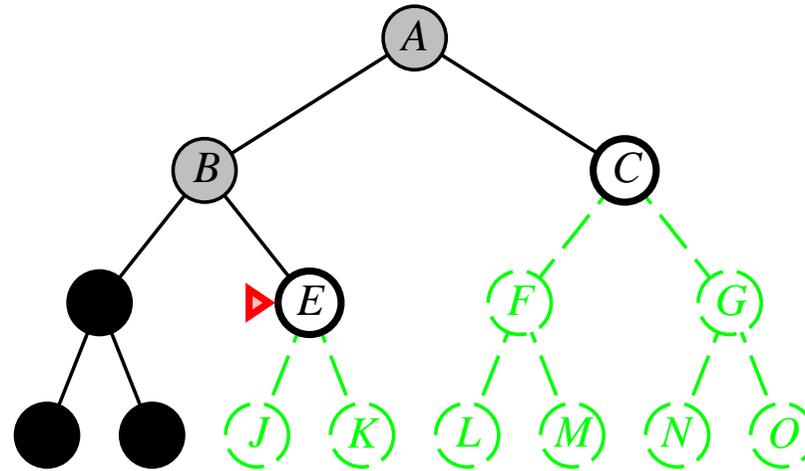


Depth-first search

Expand deepest unexpanded node

Implementation:

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

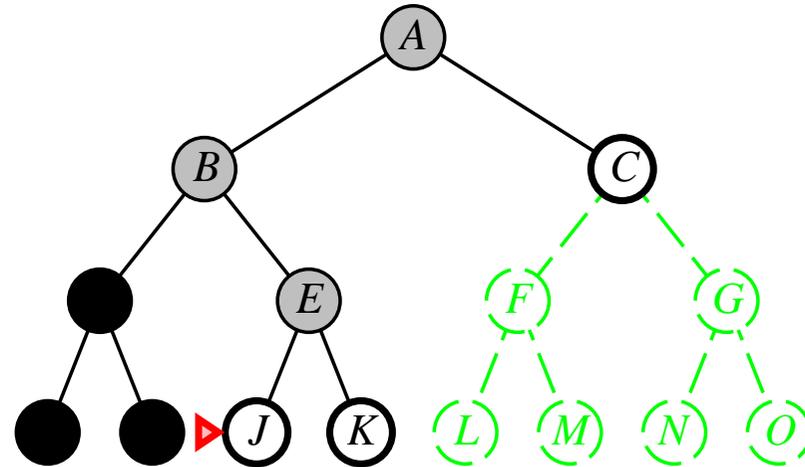


Depth-first search

Expand deepest unexpanded node

Implementation:

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

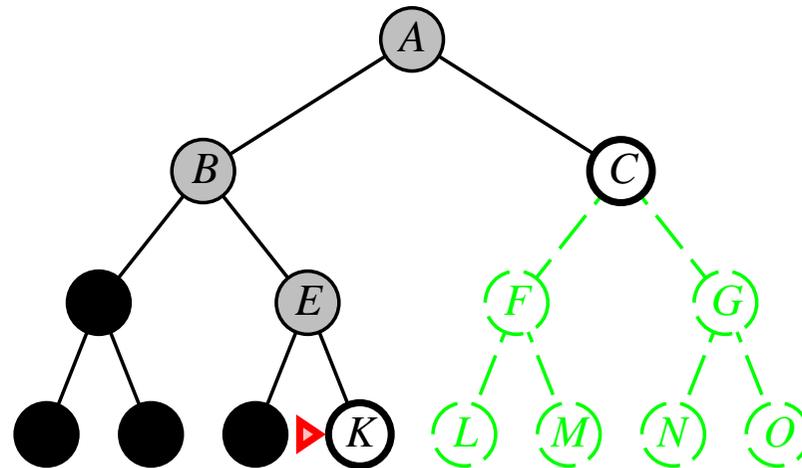


Depth-first search

Expand deepest unexpanded node

Implementation:

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

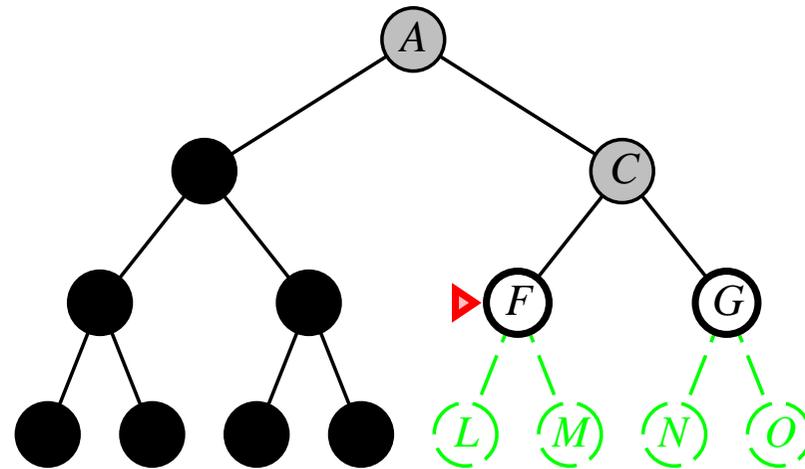


Depth-first search

Expand deepest unexpanded node

Implementation:

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

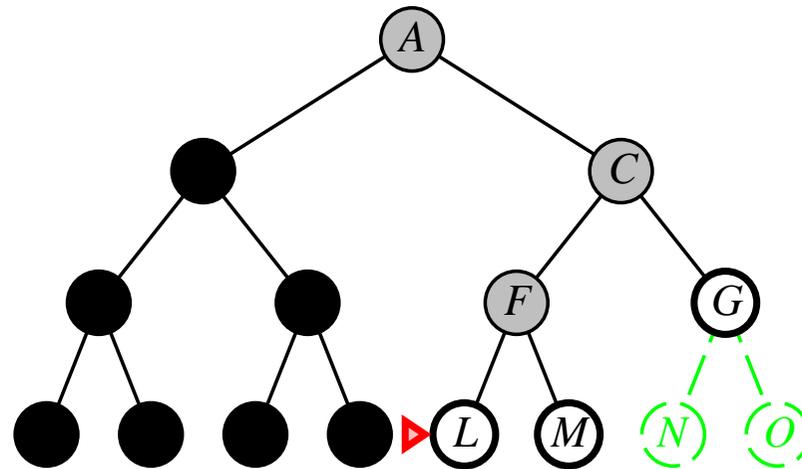


Depth-first search

Expand deepest unexpanded node

Implementation:

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

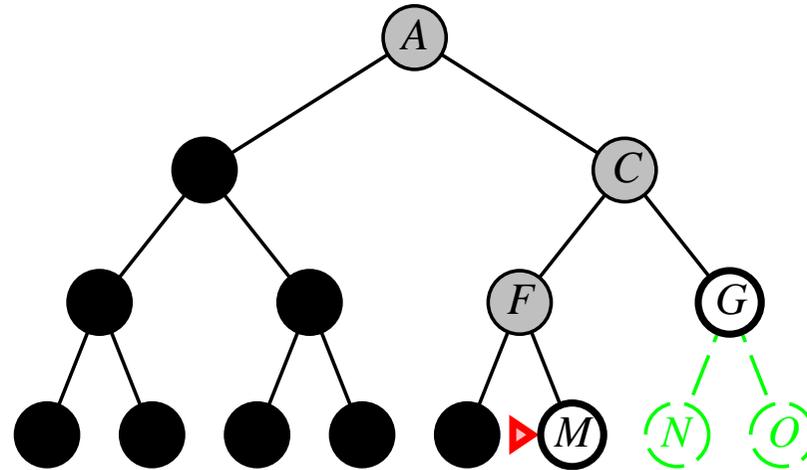


Depth-first search

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

⇒ 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

⇒ 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

⇒ 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

Backtracking search

A variant of depth-first search

Only **one successor** is generated at a time rather than all successors;

Each partially expanded node remembers which successor to generate next.

Required space: $O(m)$ instead of $O(bm)$

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 ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Properties of Depth-limited search

Diameter of state space is a suitable choice for l limit.

Complete?? No: if $l < d$

Time?? $O(b^l)$

Space?? $O(bl)$, i.e., linear space!

Optimal?? No

Depth-first search is a special case of Depth-limited search for $l = \infty$

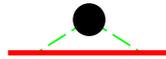
Iterative deepening search

It combines the benefits of breadth-first and depth-first 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
```

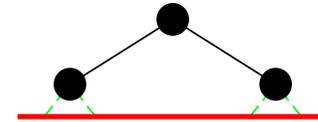
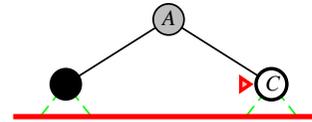
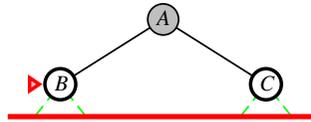
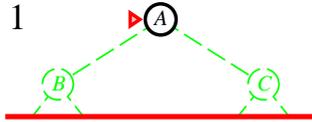
Iterative deepening search $l = 0$

Limit = 0



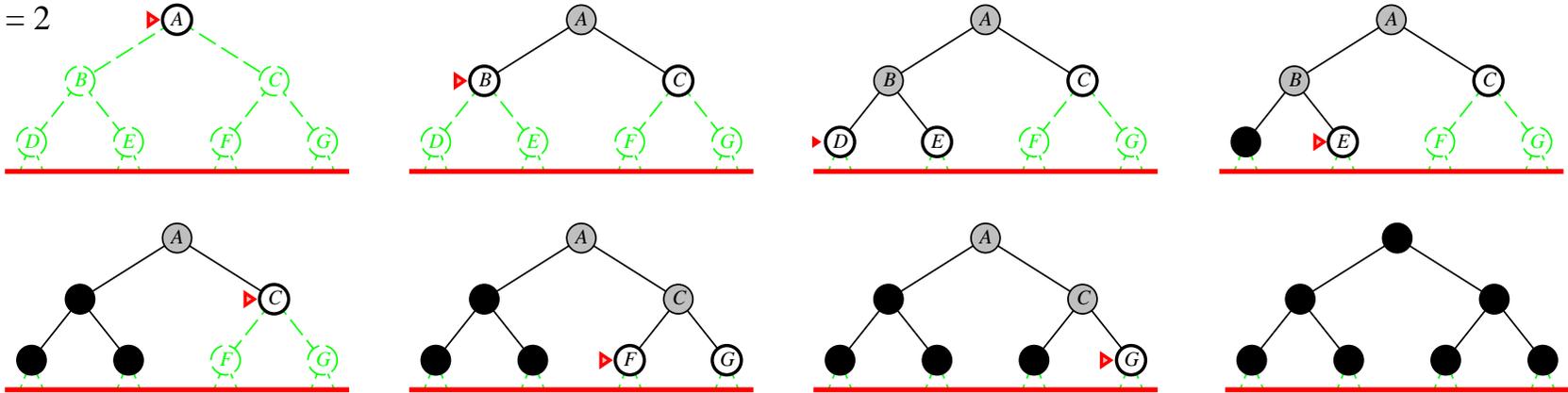
Iterative deepening search $l = 1$

Limit = 1



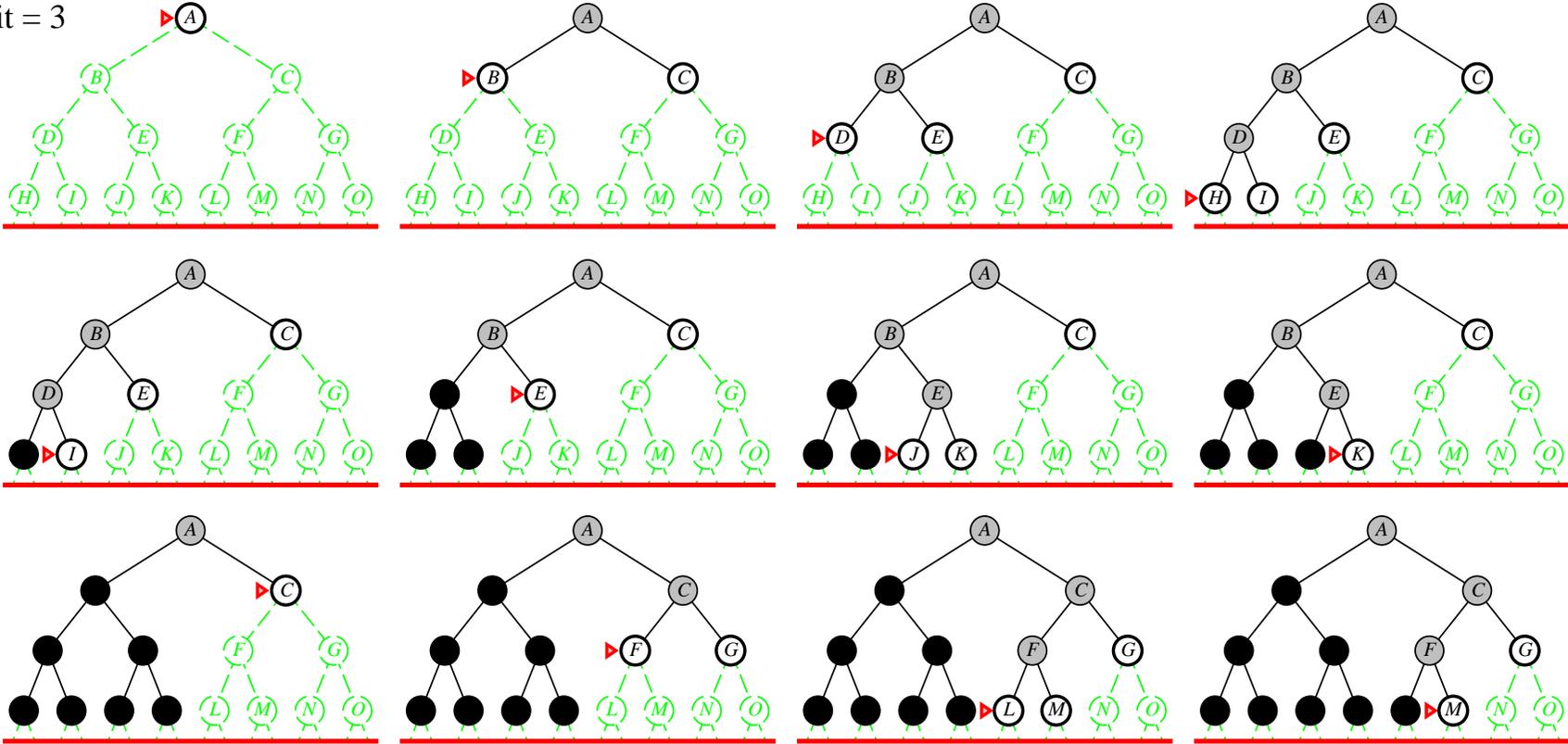
Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$

Limit = 3



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if path cost is a non-decreasing function of node depth
e.g. cost = 1 per step

Can be modified to explore uniform-cost tree : **Iterative lengthening Search**

Numerical comparison for $b = 10$ and $d = 5$, solution at far right:

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

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

Summary of algorithms

Criterion	Breadth- First	Uniform- Cost	Depth- First	Depth- Limited	Iterative Deepening
Complete?	Yes ^a	Yes ^{a,b}	No	Yes, if $l \geq d$	Yes ^a
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 ^c	Yes	No	No	Yes ^c

b = branching factor

d = depth of the shallowest solution

l = depth limit

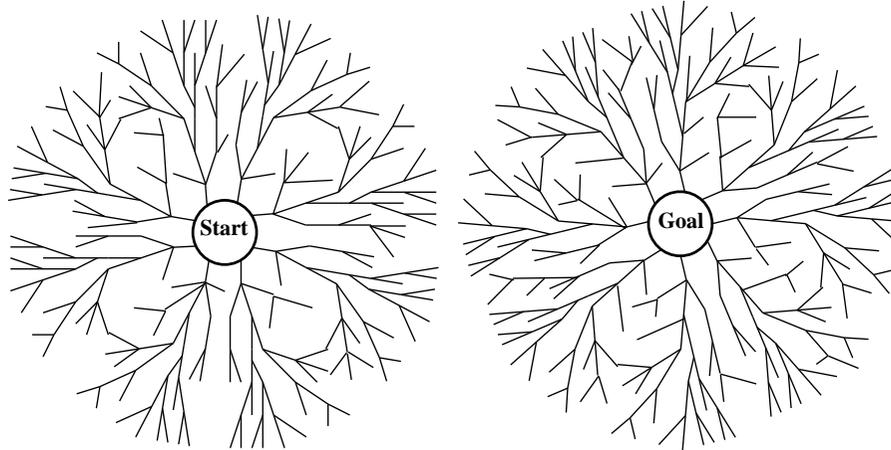
m = maximum depth of the search tree

^a complete if b is finite

^b complete if step cost $\geq \epsilon$ for $\epsilon > 0$

^c optimal if path cost is a non-decreasing function of node depth (e.g. step costs are all identical)

Bidirectional Search



A big problem: finding predecessors of a state

Complete?? if both directions use breadth-first search **and** b is finite.

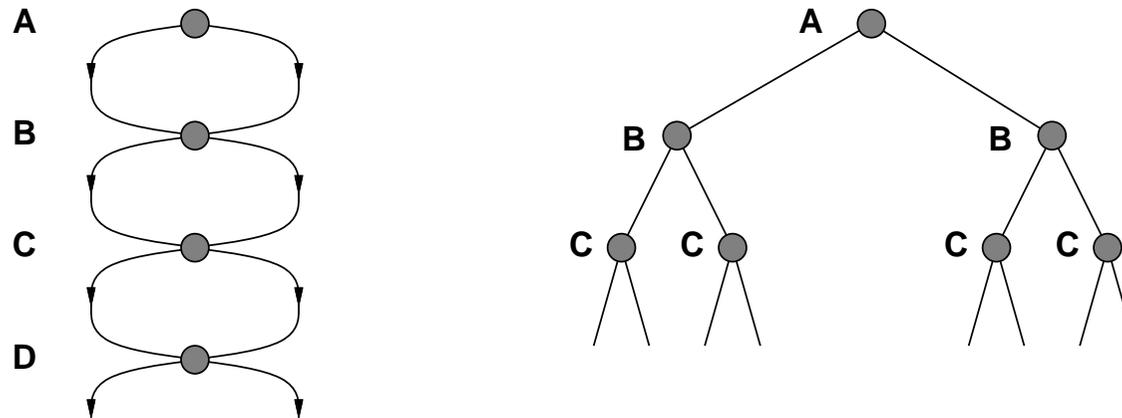
Time?? $O(2b^{d/2}) = O(b^{d/2})$

Space?? $O(2b^{d/2}) = O(b^{d/2})$

Optimal?? if both directions use breadth-first search **and** path cost is a non-decreasing function of node depth

Avoiding repeated states

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



Solution: exploration of state space graph using closed list.

Closed list: a list of expanded node (can be implemented by hash table)

Required space is proportional to the size of state space: $O(|S|) \leq O(b^d)$

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