CSE 482 - Artificial Intelligence

Artificial Intelligence: A Modern Approach

Original Slides by S. Russell
Modified by A. H. Özer

Problem Solving and Search

Outline

- ◇ Problem-solving agents
- ◇ Problem types
- ◇ Problem formulation
- ◇ Example problems
- ◇ Basic search algorithms

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*

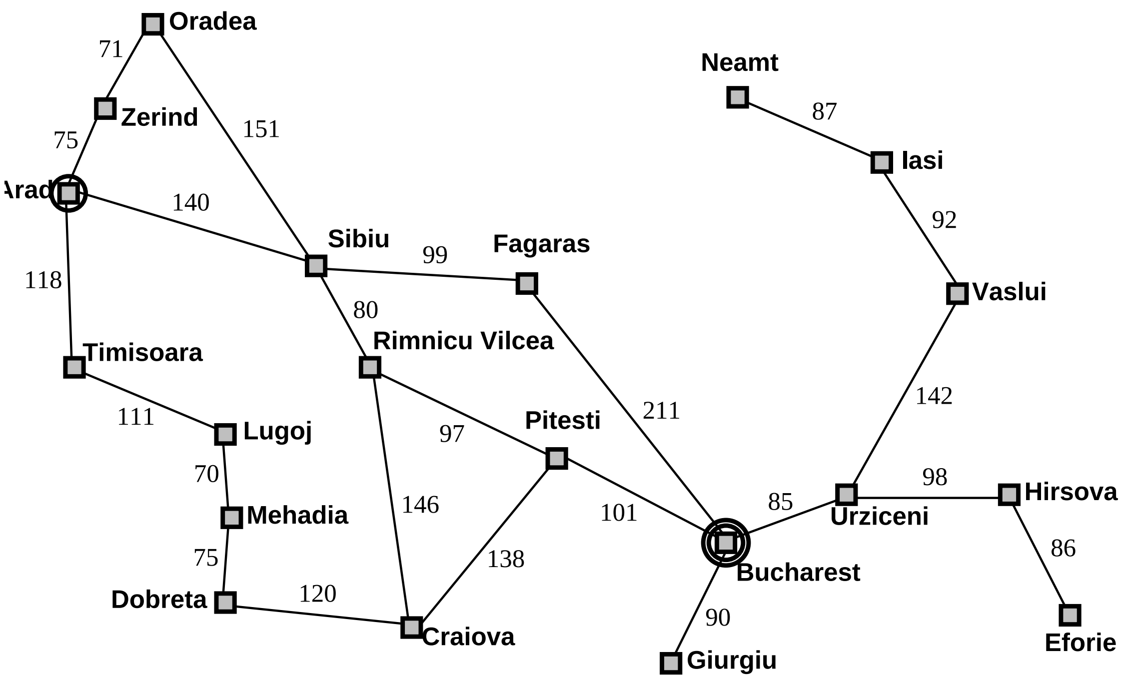
Problem-Solving Agents

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

Example: Romania

An agent is on holiday in Romania; currently in Arad.

He must be in Bucharest tomorrow to catch his flight.



Problem Definition

A **problem** can be defined formally by five components:

♢ The **initial state** that the agent starts in. e.g. $In(Arad)$

♢ A description of the possible **actions** available to the agent.

– Given a particular state s, $ACTIONS(s)$ returns the set of actions that can be executed in s.

– $ACTIONS(In(Arad)) = \{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$

♢ A description of what each action does, i.e. the **transition model**.

– $RESULT(s, a)$ returns the state that results from doing action a in state s.

– e.g. $RESULT(In(Arad), Go(Zerind)) = In(Zerind)$

Problem Definition

♢ The **goal test** which determines whether a given state is a goal state. e.g. $In(Bucharest)$

♢ A **path cost** function that assigns a numeric cost to each sequence of states.

– The cost function reflects agent's own performance measure.

Problem Definition

The **state space** is the set of all states reachable from the initial state by any sequence of actions.

– The initial state, actions and the transition model define the state space.

– The state space forms a **directed graph** in which the nodes are the states, the arcs are the actions.

– A **path** in the state space is a sequence of states connected by a sequence of actions.

A **solution** is an action sequence (or path) that leads from the initial state to a goal state.

The **optimum solution** is the solution that has the lowest path cost among all solutions, i.e. the shortest path.

Selecting a State Space

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

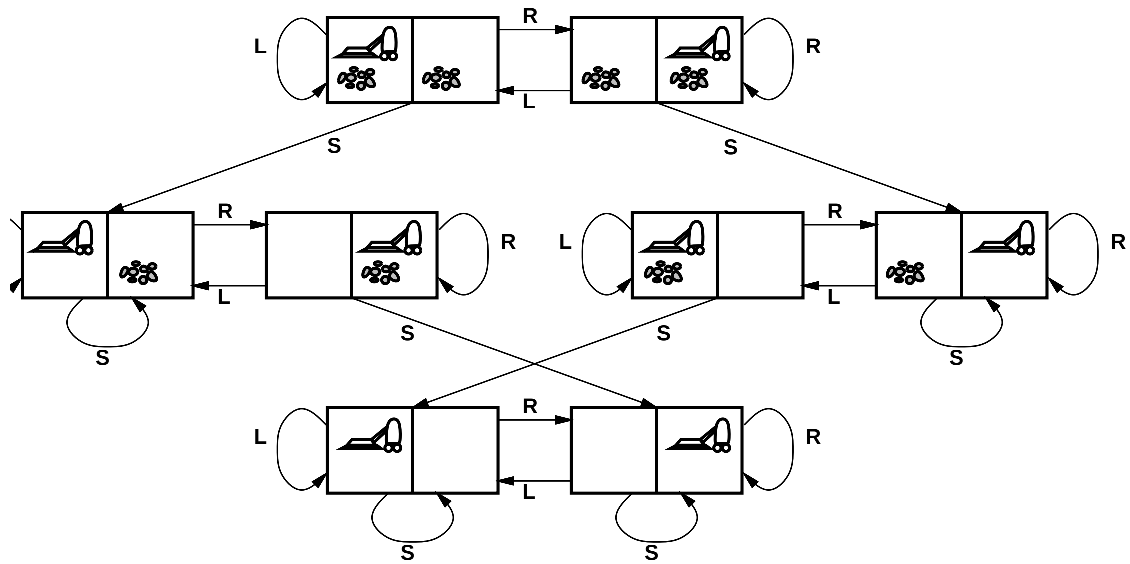
(Abstract) state = set of real states

(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: vacuum world state space graph



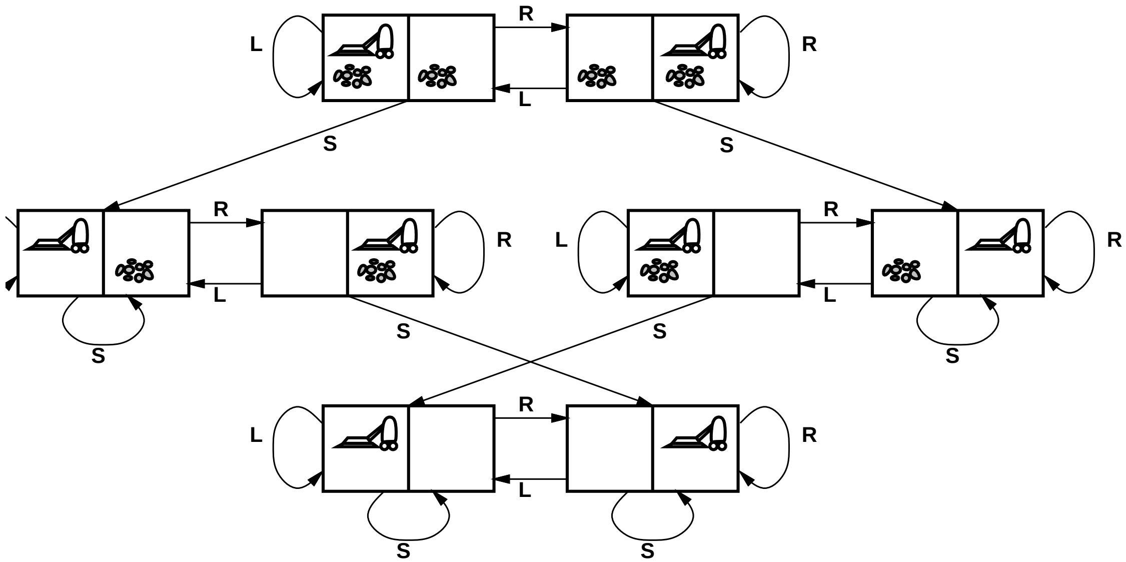
states?

actions?

goal test?

path cost?

Example: vacuum world state space graph



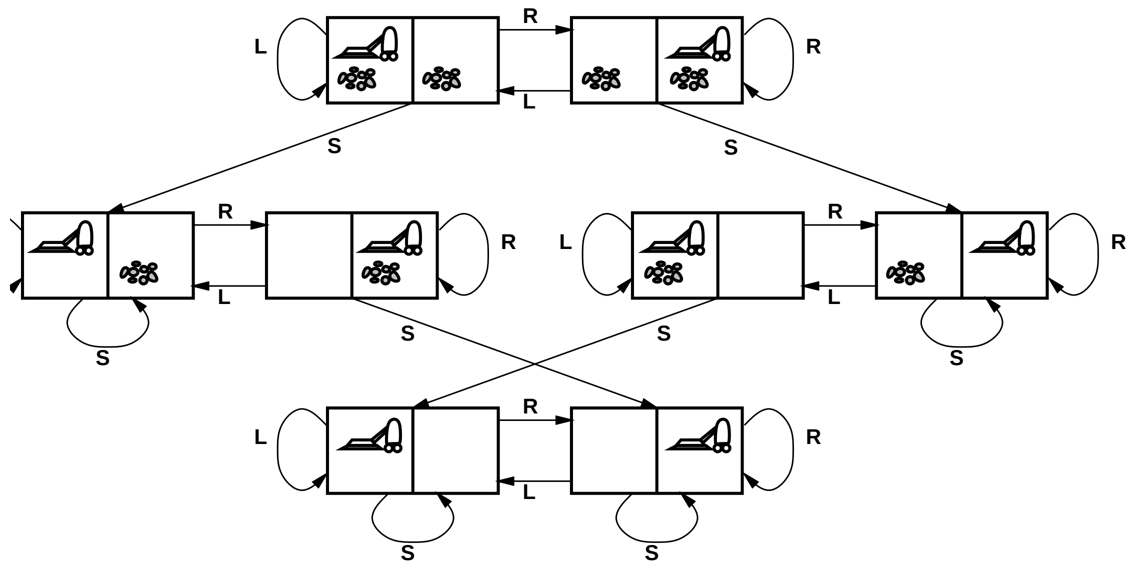
states?: integer dirt and robot locations (ignore dirt amounts etc.)

actions?

goal test?

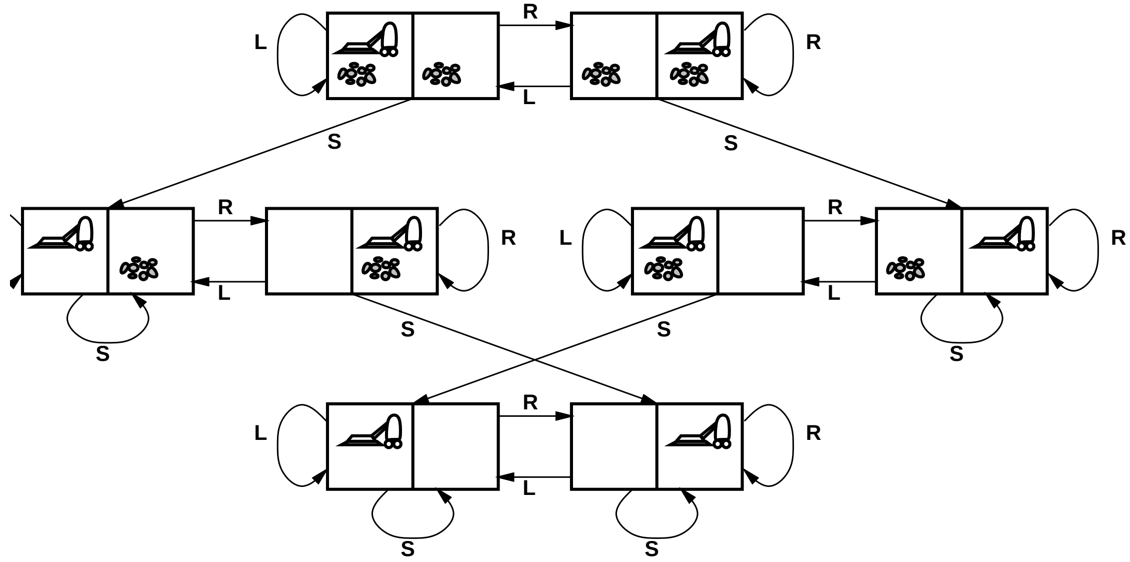
path cost?

Example: vacuum world state space graph



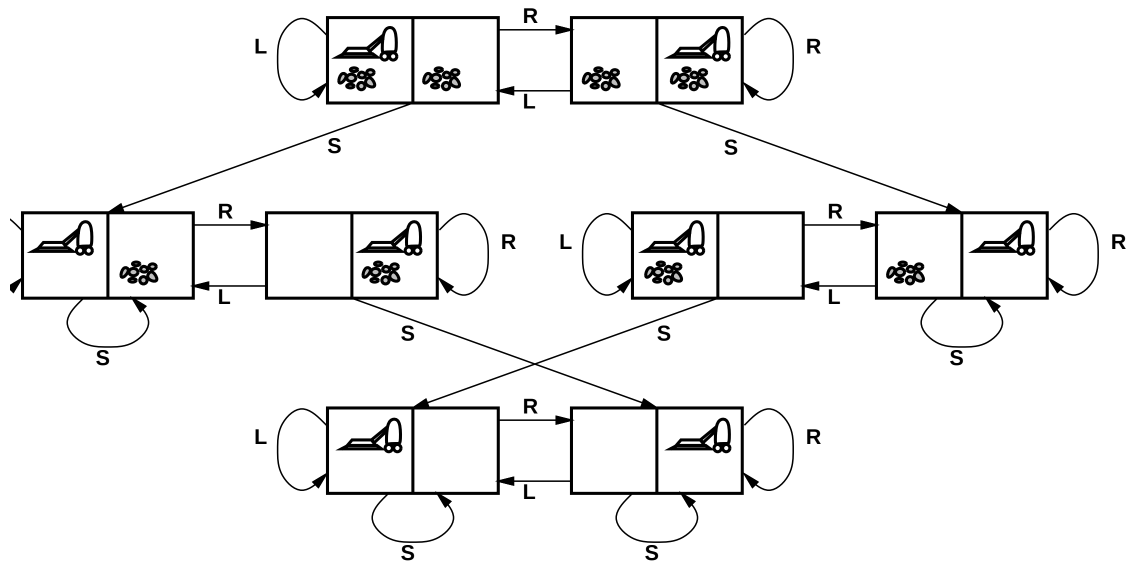
states?: integer dirt and robot locations (ignore dirt amounts etc.)
actions?: *Left*, *Right*, *Suck*, *NoOp*
goal test?
path cost?

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?

Example: vacuum world state space graph



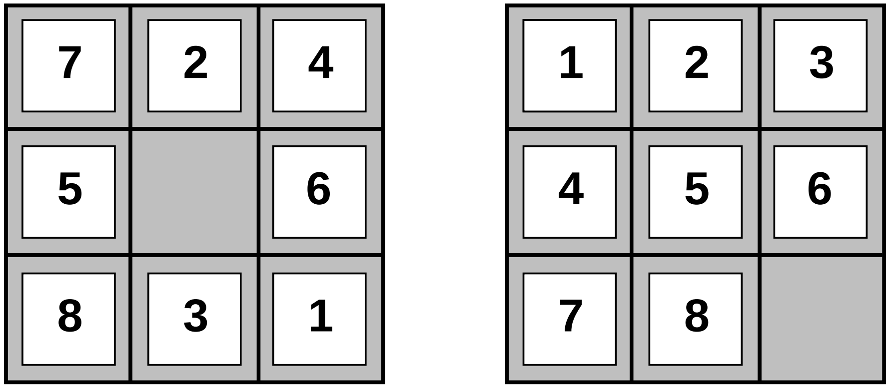
states?: integer dirt and robot locations (ignore dirt amounts etc.)

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

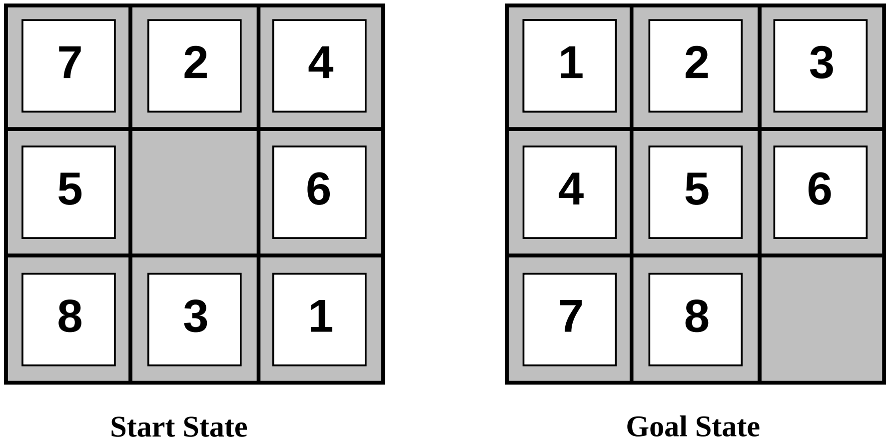
states?

actions?

goal test?

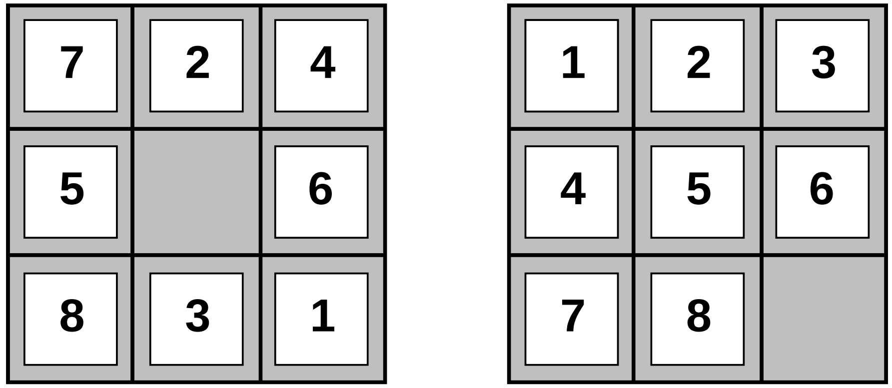
path cost?

Example: The 8-puzzle



states?: integer locations of tiles (ignore intermediate positions)
actions?
goal test?
path cost?

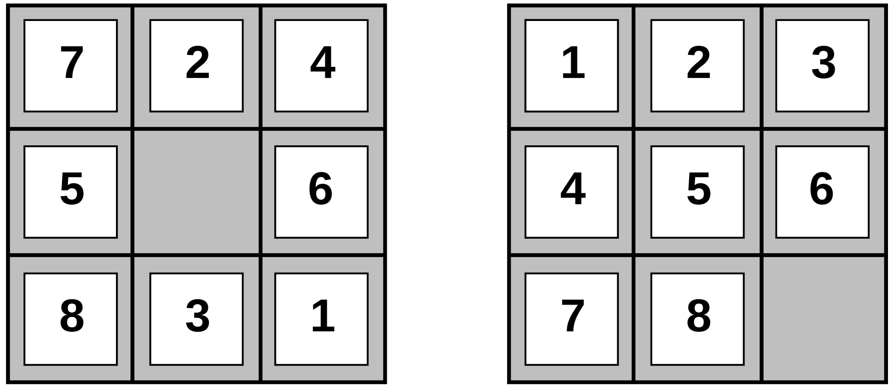
Example: The 8-puzzle



Start State **Goal State**

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

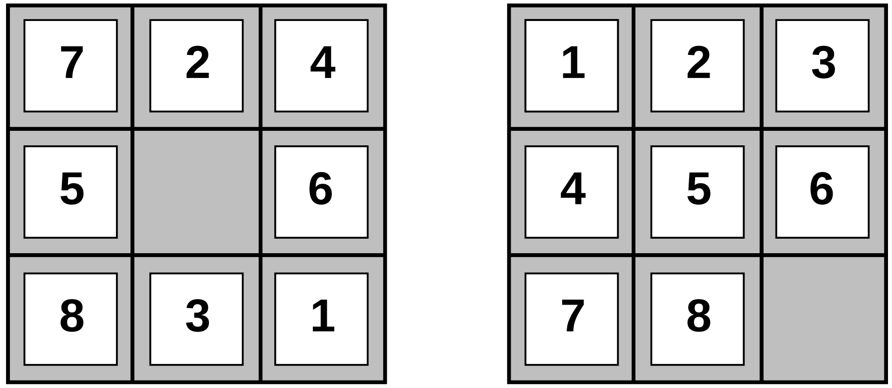
Example: The 8-puzzle



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?

Example: The 8-puzzle

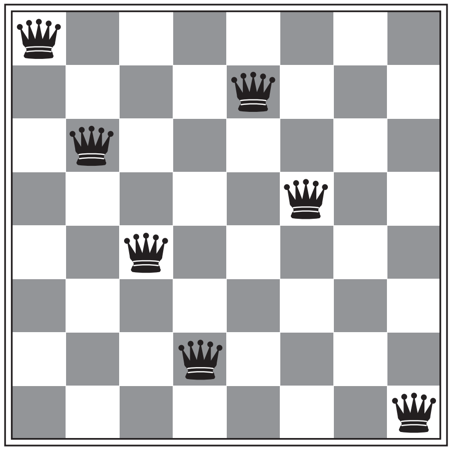


Start State **Goal State**

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

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

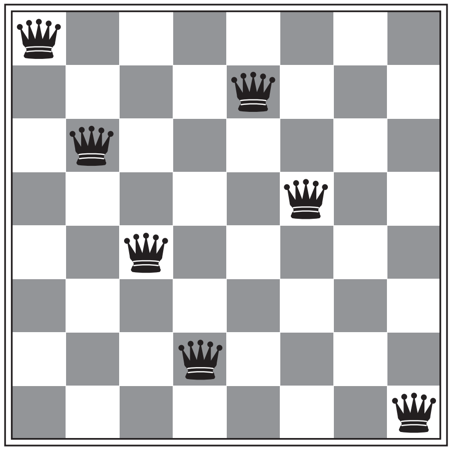
Example: The 8-queens Problem



states?: Any arrangement of 0 to 8 queens on the board.
initial state?: No queens on the board.
actions?: Add a queen to any empty square.
goal test?: 8 queens are on the board, none attacked.
path cost?: none

[Note: $64 \cdot 63 \ldots 57 \approx 1.8 \cdot 10^{14}$ possible paths to search!]

The 8-queens Problem - Alternative



states?: All possible arrangements of n queens $(1 \leq n \leq 8)$, one per column in the leftmost n columns, with no queen attacking another.
actions?: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

[Note: Reduces the state space from $1.8 \cdot 10^{14}$ states to 2057!]

Infinite State Space

Donald Knuth (1964) conjectured that starting with number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.

e.g. $\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$

states?: Positive numbers.
initial state?: 4.
actions?: Apply factorial, square root, or floor operation (factorial for integers only).
goal test?: State is the desired integer.

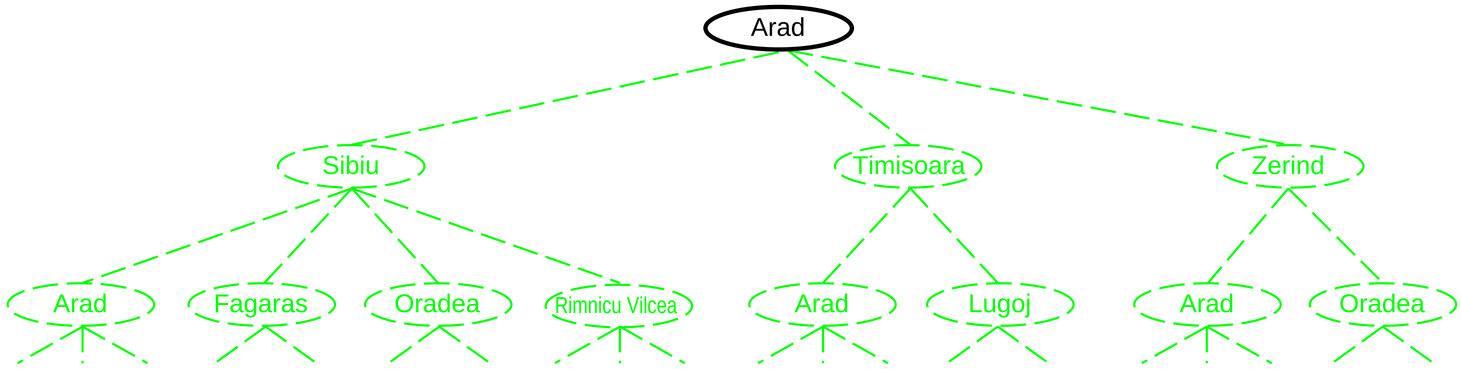
[Note: (4!)! $=$ 620448401733239439360000]

Tree search algorithms

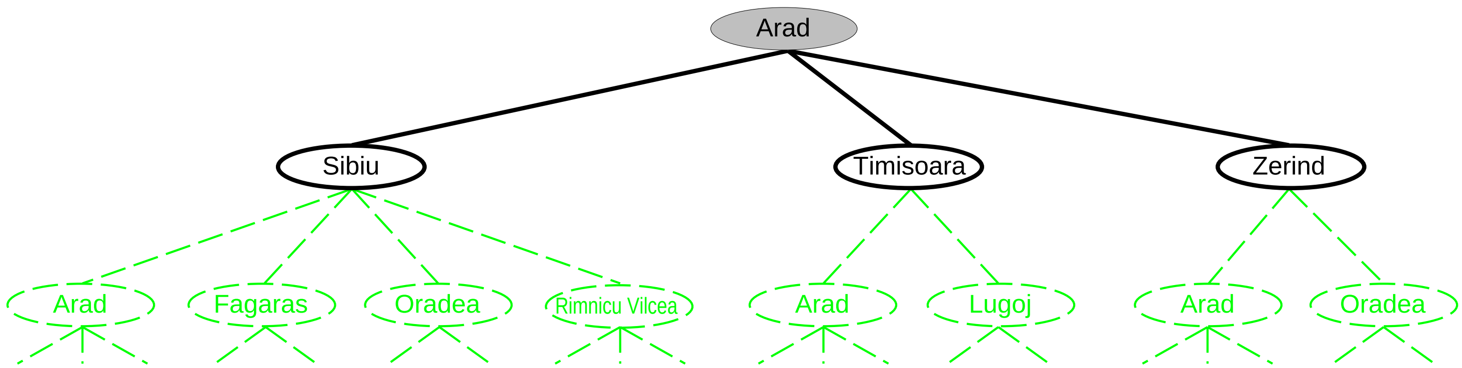
Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure
initialize the frontier (nodes to be visited) using the initial state of *problem*
loop do
if the frontier is empty **then return** failure
choose a leaf node for expansion according to *strategy* and remove it from the frontier
if the node contains a goal state **then return** the corresponding solution
expand the node and add the resulting nodes to the frontier
end

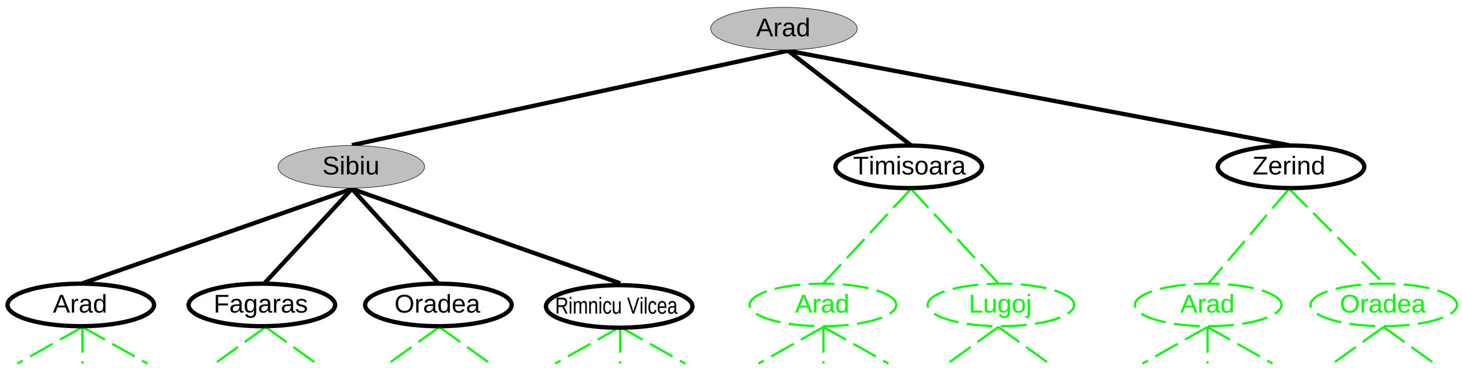
Tree search example



Tree search example



Tree search example



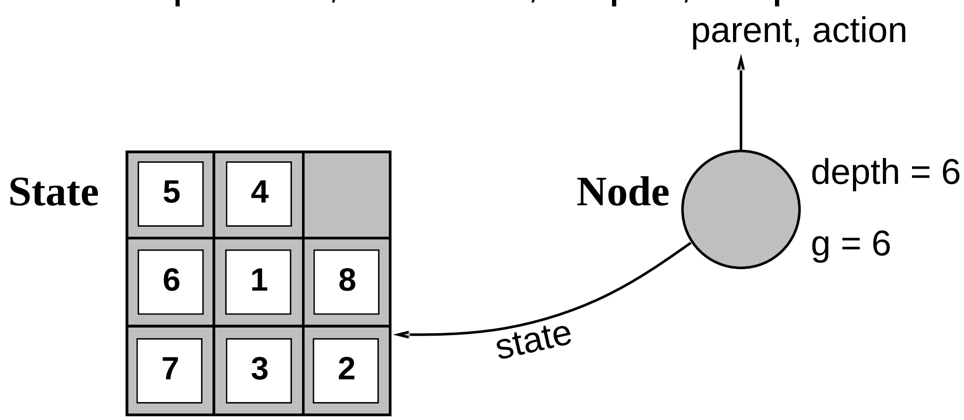
Implementation: states vs. nodes

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 successor function ($RESULT$) of the problem to create the corresponding states.

General Graph Search

function GRAPH-SEARCH(*problem*, *strategy*) **returns** a solution, or failure
  initialize the frontier (nodes to be visited) using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if there the frontier is empty **then return** failure
    choose a leaf node for expansion according to *strategy* and remove it from the search tree
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the node and add the resulting nodes to the frontier
     only if not in the frontier or explored set
  end

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

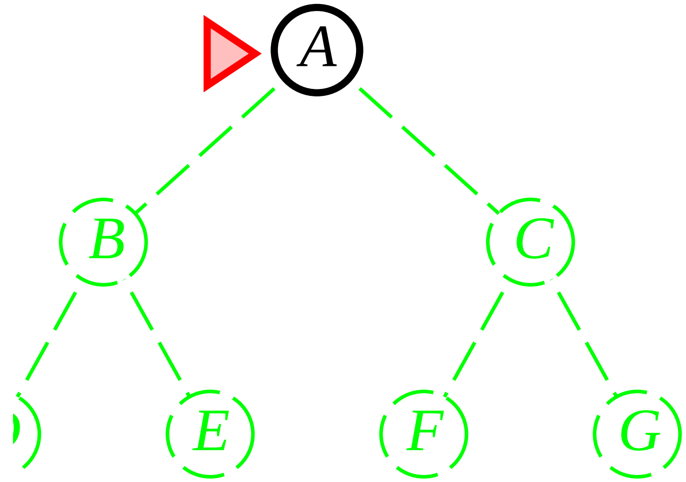
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

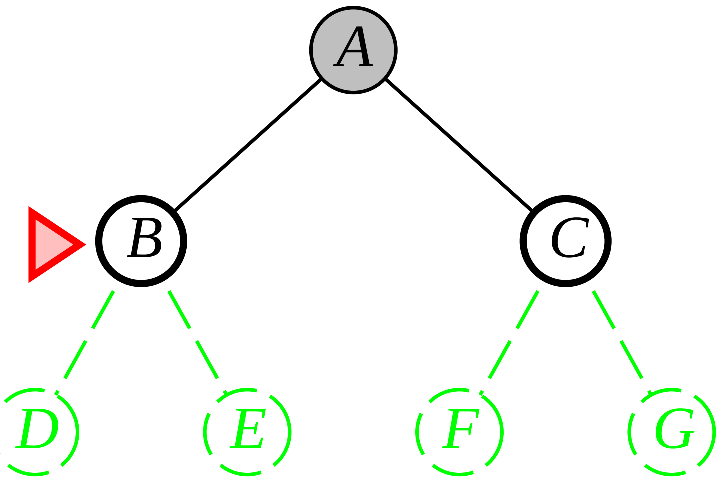


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

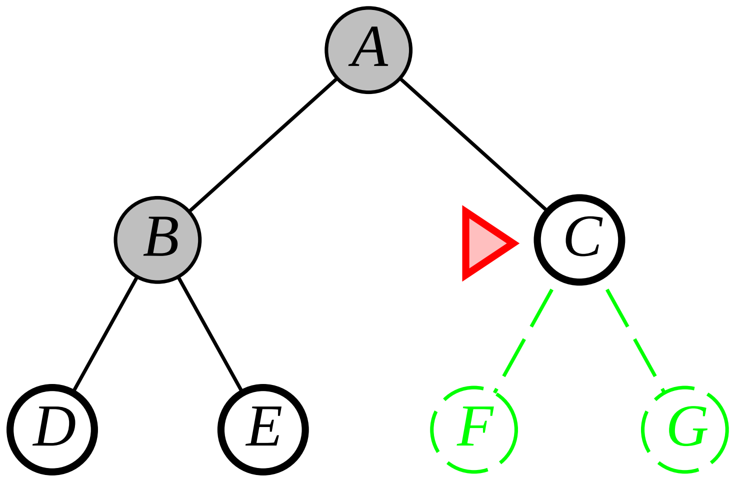


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

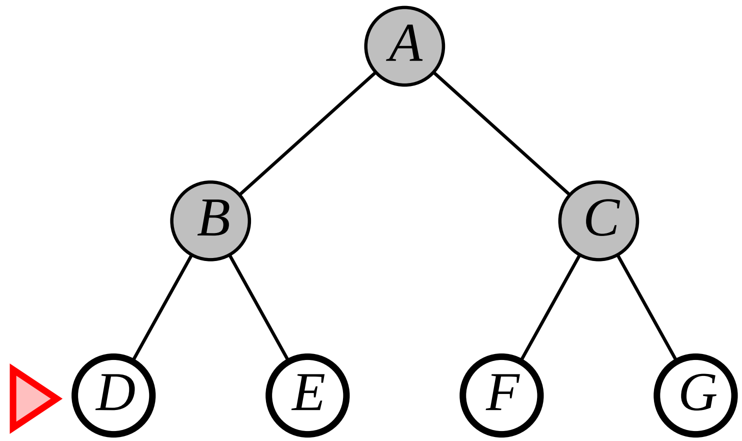


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns soln/fail/cutoff
  node ← a node with STATE=problem.INITIAL-STATE,PATH-COST=0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?((frontier)) then return failure
    node ← POP(frontier)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
      if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
      frontier ← INSERT(child, frontier)
```

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

Space?

Properties of breadth-first search

Complete? Yes (if b is finite)

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

Space? $O(b^d)$ (keeps every node in memory)

Optimum?

Properties of breadth-first search

Complete? Yes (if b is finite)

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

Space? $O(b^d)$ (keeps every node in memory)

Optimum? Yes (if cost $=$ 1 per step); not optimum in general

For a branching factor $b = 10$, depth $d = 16$, 1 million nodes/second, 1000 bytes/node:

The number of nodes: 10^{16}, Time: 350 years, Memory: 10 exabytes

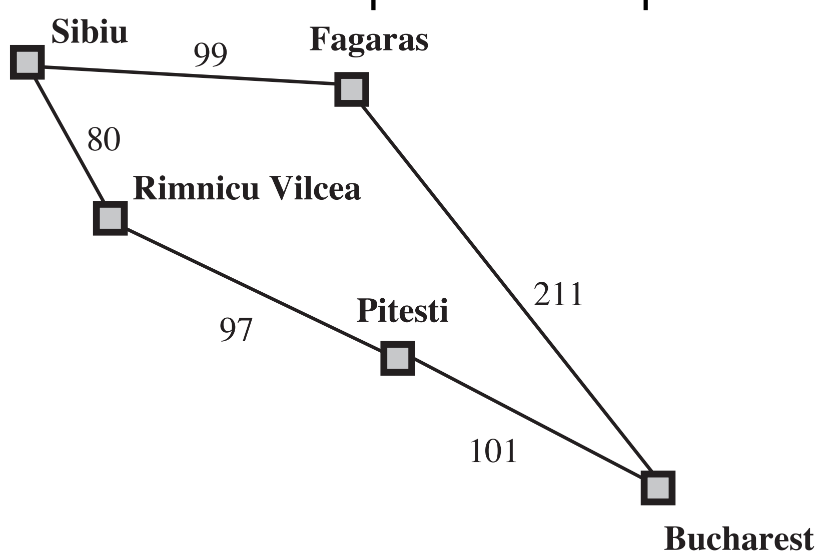
Uniform-cost search

Expand least-cost unexpanded node.

Implementation:

$\quad$ *frontier* = priority queue ordered by path cost, lowest first.

Equivalent to breadth-first if step costs all equal.



Uniform-cost search

Complete? Yes, if step cost $\geq \epsilon$ (No-Op operation?)

Time? # of nodes with $g \leq$ cost of optimum solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
where C^* is the cost of the optimum solution

Space? # of nodes with $g \leq$ cost of optimum solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

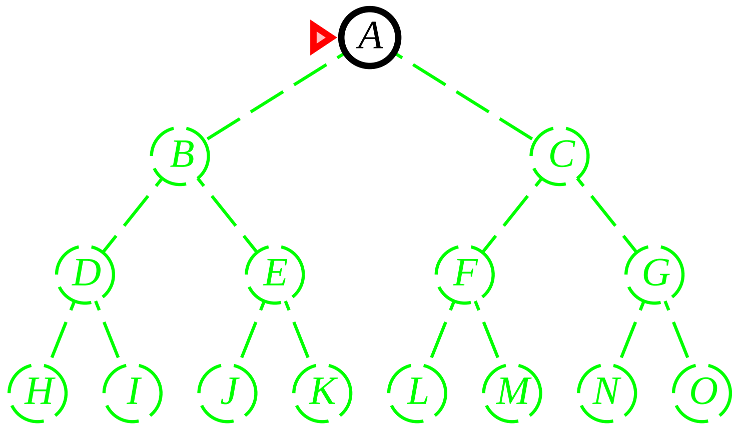
Optimum? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

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

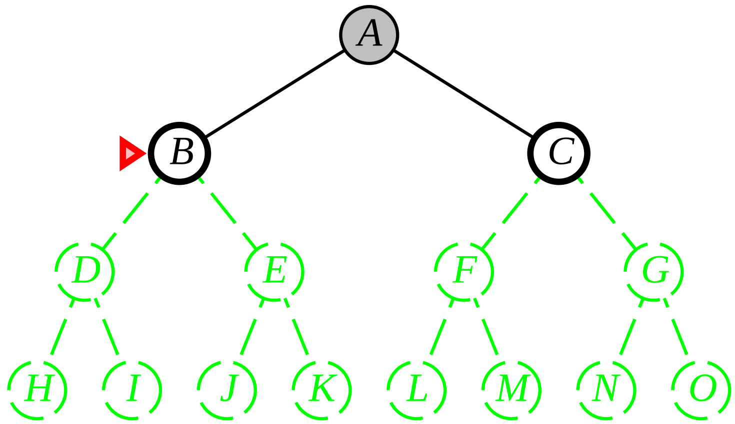


Depth-first search

Expand deepest unexpanded node

Implementation:

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

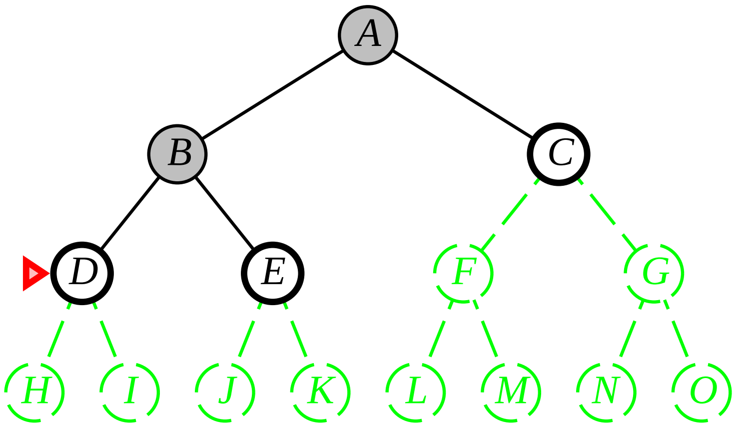


Depth-first search

Expand deepest unexpanded node

Implementation:

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

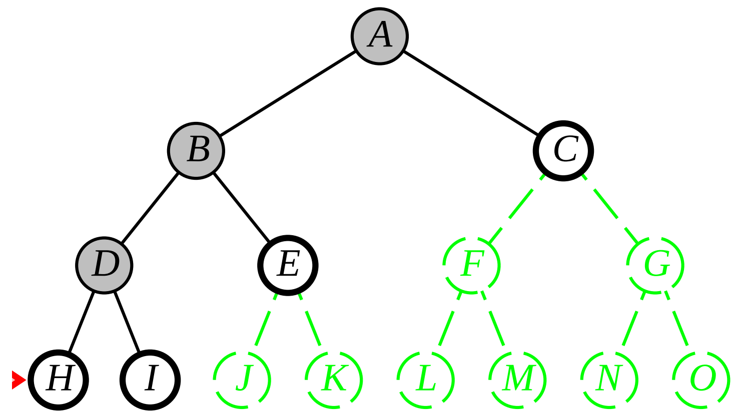


Depth-first search

Expand deepest unexpanded node

Implementation:

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

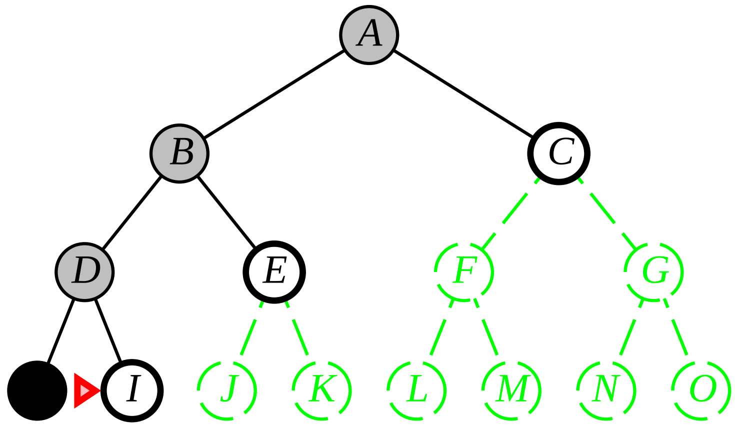


Depth-first search

Expand deepest unexpanded node

Implementation:

$\textit{frontier} =$ LIFO queue, i.e., put successors at front

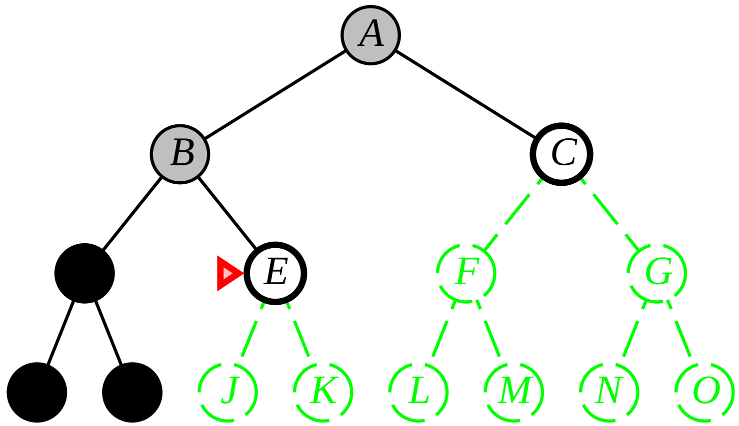


Depth-first search

Expand deepest unexpanded node

Implementation:

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

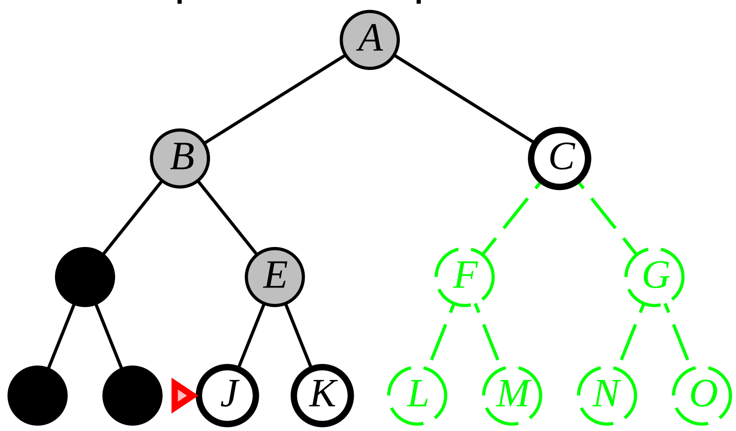


Depth-first search

Expand deepest unexpanded node

Implementation:

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

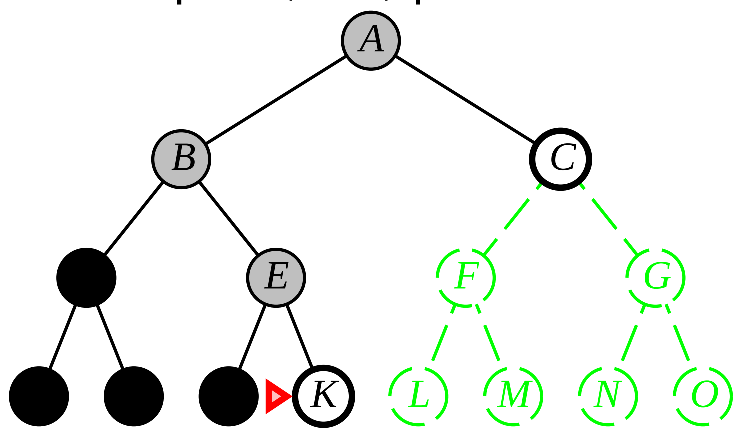


Depth-first search

Expand deepest unexpanded node

Implementation:

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

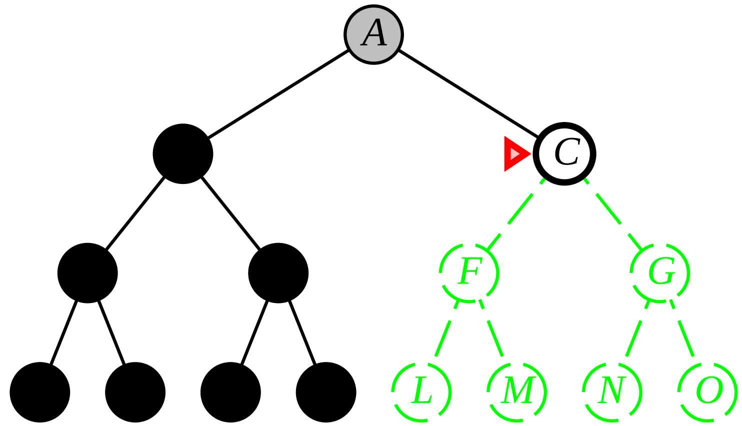


Depth-first search

Expand deepest unexpanded node

Implementation:

$\textit{frontier}$ = 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
$\Rightarrow$ 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!

Optimum?

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!

Optimum? No

Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors.

Depth-limited search

Recursive implementation:

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

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0  then return cutoff
   else
      cutoff-occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
         child ← CHILD-NODE(problem, node, action)
         result ← RECURSIVE-DLS(child, problem, limit-1)
         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

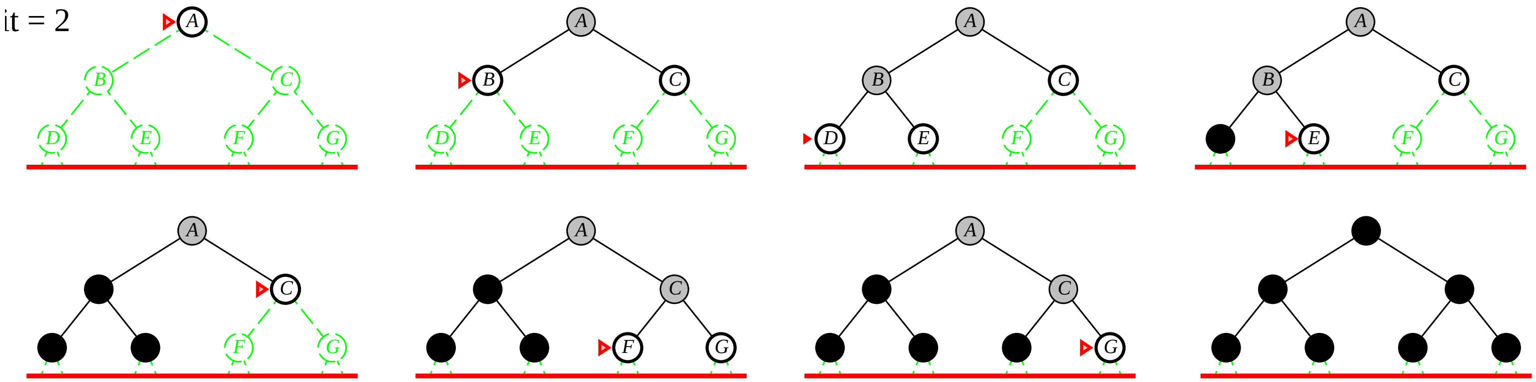
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution
    inputs: *problem*, a problem

    for *depth* $\leftarrow$ 0 **to** ∞ **do**
        result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
        if *result* $\neq$ cutoff **then return** *result*
    end

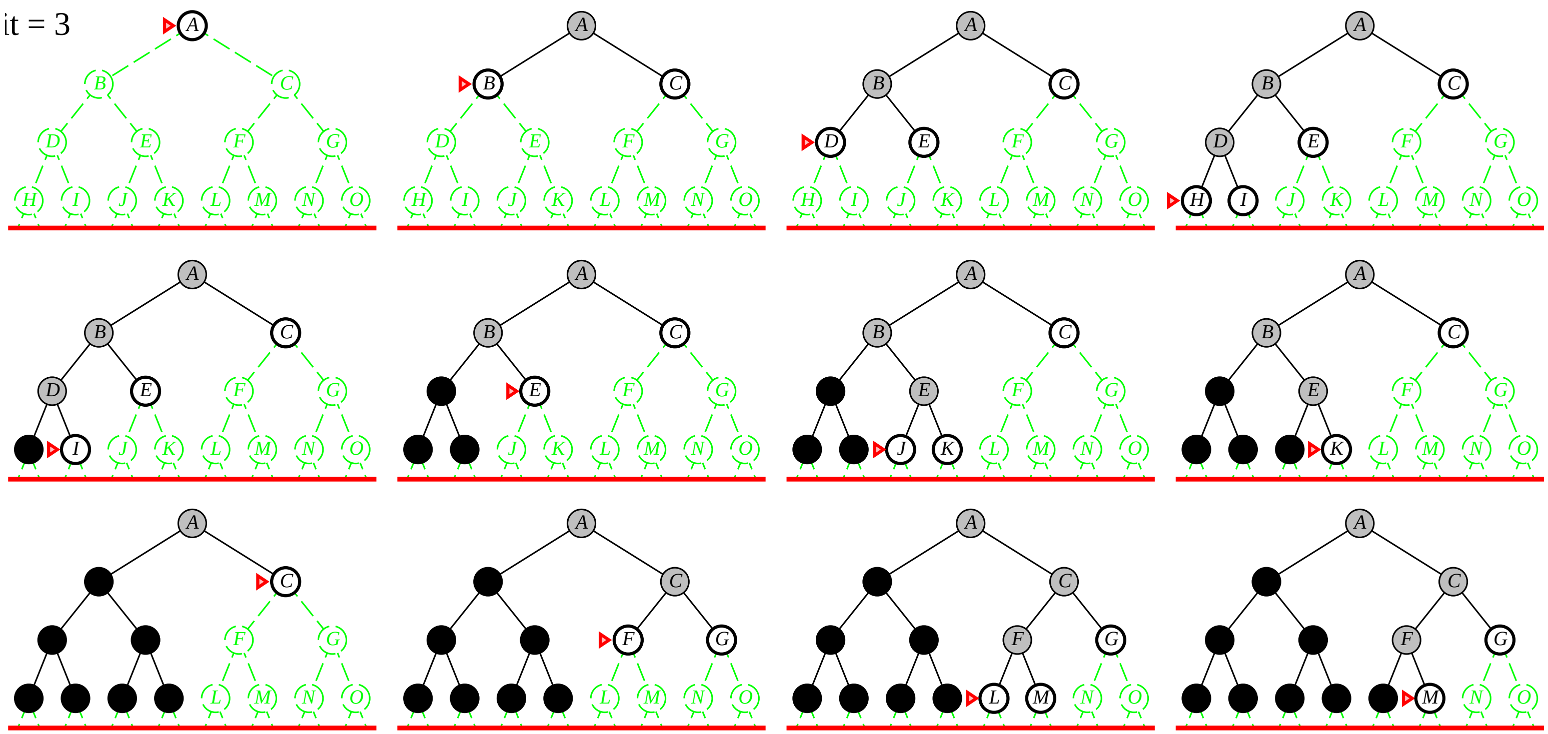
Iterative deepening search $l = 0$

Iterative deepening search $l = 1$

Iterative deepening search $l = 2$



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

Complete? Yes

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

Space? $O(bd)$

Optimum?

Properties of iterative deepening search

Complete? Yes

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

Space? $O(bd)$

Optimum? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree: **iterative lengthening search**.

Properties of iterative deepening search

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

$$\begin{aligned} N(\mathsf{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\mathsf{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 \end{aligned}$$

Revisiting in IDS does not incur much overhead.

In general, IDS is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	Yes (if finite)	Yes, if $l \geq d$	Yes
Time	b^d	$b^{1+\lfloor C^*/\epsilon \rfloor}$	b^m	b^l	b^d
Space	b^d	$b^{1+\lfloor C^*/\epsilon \rfloor}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*