Solving Problems by Searching

Chapter 3

Outline

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

Problem-solving agents

function SIMPLE-PROBLEM-SOLVING-AGENT (*percept*)
returns an action
  inputs: *percept* a percept
  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 do**
    goal ← FORMULATE-GOAL (*state*)
    problem ← FORMULATE-PROBLEM (*state,goal*)
    seq ← SEARCH (*problem*)
  action ← FIRST (*seq*)
  seq ← REST (*seq*)
  return *action*

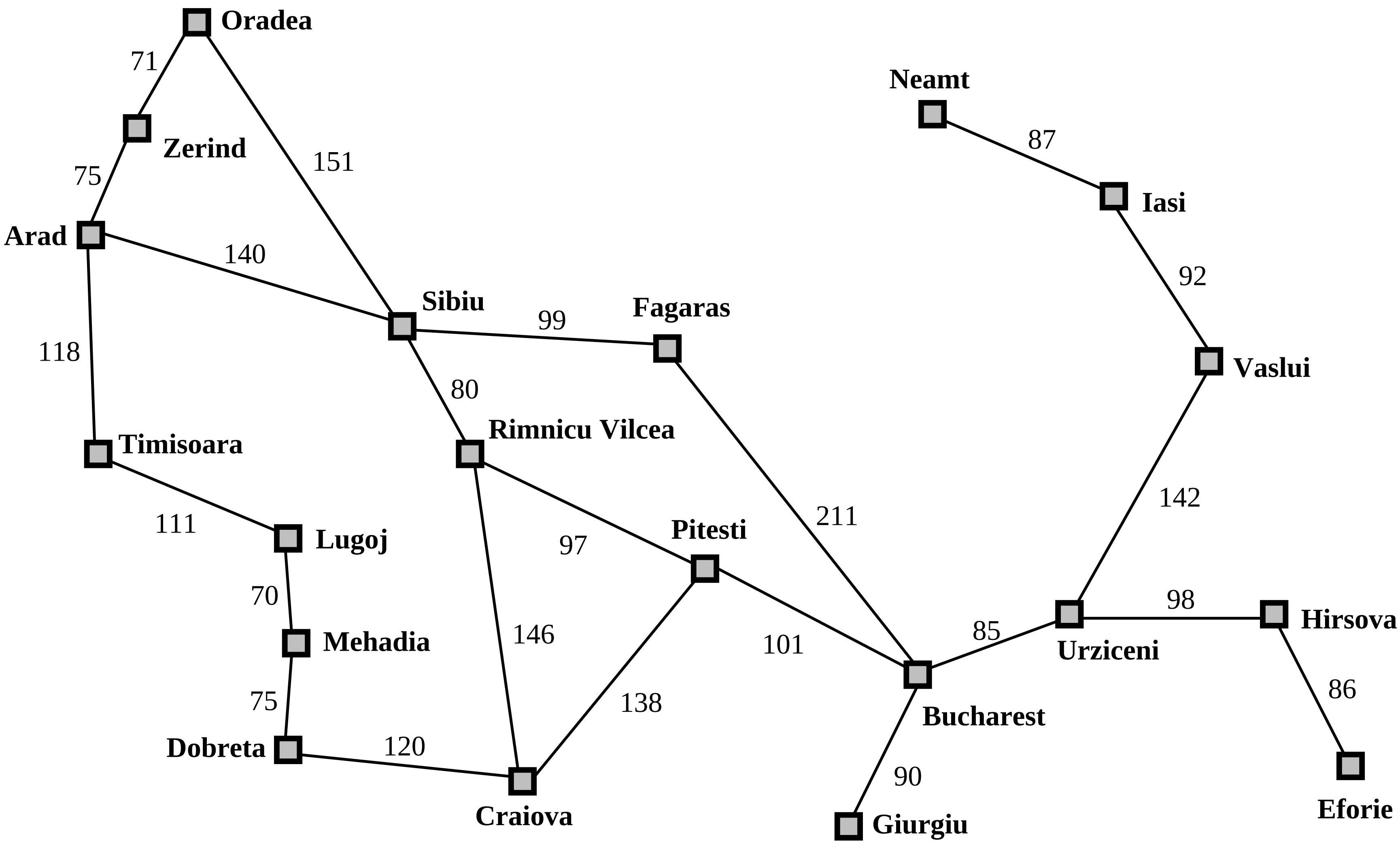
Problem-solving agents (cont'd)

- Restricted form of general agent
- This is *offline* problem solving; solution executed "eyes closed"
- *Online* problem solving involves acting without complete knowledge
- Assumes: static, observable, discrete, deterministic

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

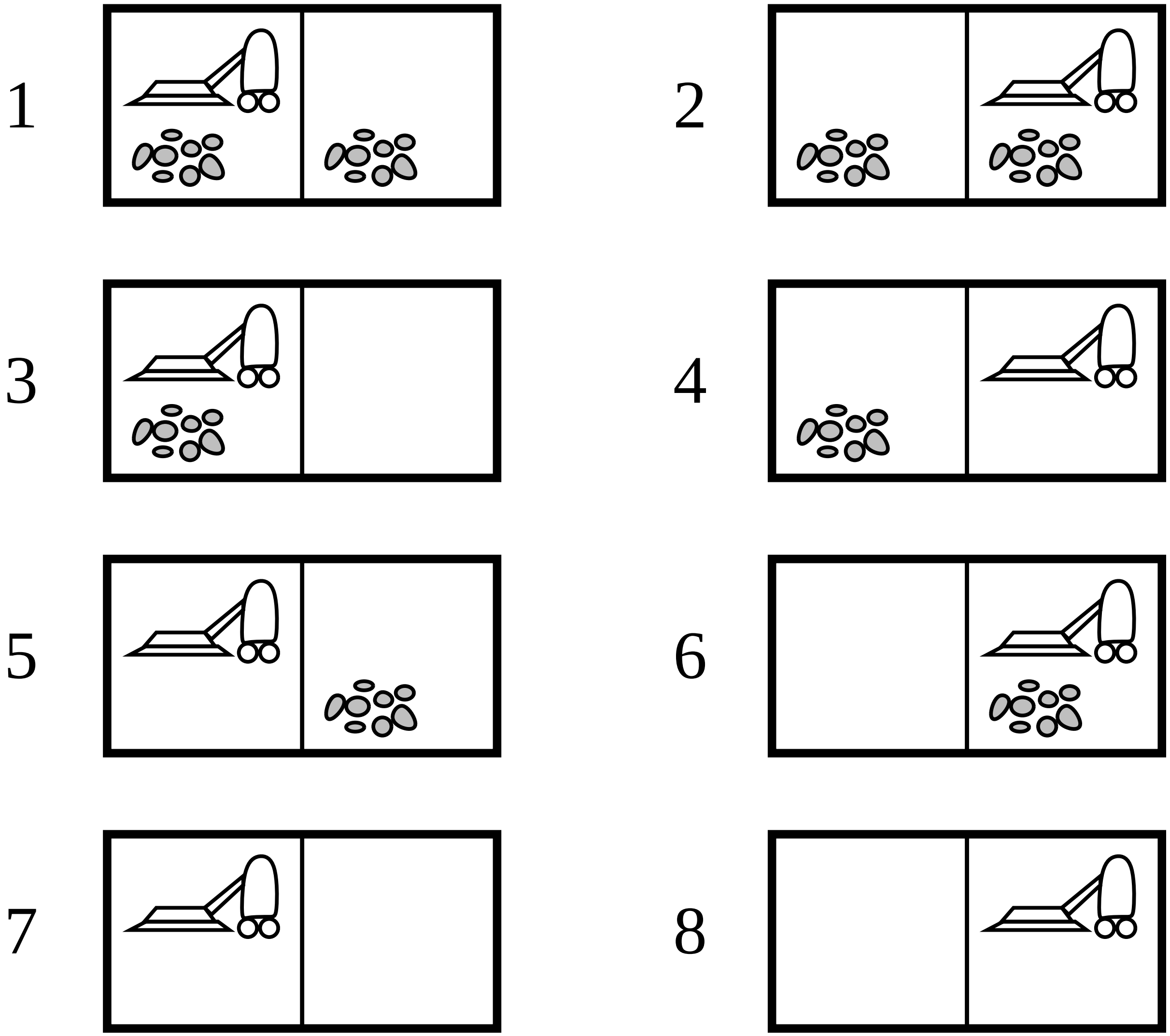
Example: Romania



Problem types

- Deterministic, fully observable $\Longrightarrow$ *single-state problem*
  Agent knows exactly which state it will be in; solution is a sequence
- Non-observable $\Longrightarrow$ *conformant problem*
  Agent may have no idea where it is; solution (if any) is a sequence
- Nondeterministic and/or partially observable $\Longrightarrow$ *contingency problem*
  percepts provide *new* information about current state
  solution is a *tree* or *policy*
  often *interleave* search, execution
- Unknown state space $\Longrightarrow$ *exploration problem* ("online")

Example: vacuum world



Example: vacuum world

- Single-state, start in #5.
Solution??
- $[Right, Suck]$

Example: vacuum world

- Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$.
  e.g., $Right$ goes to $\{2, 4, 6, 8\}$.
  Solution??
- $[Right, Suck, Left, Suck]$

Example: vacuum world

- Contingency, start in #5 or #7
  Murphy's Law: if a carpet can get dirty it will
  Local sensing: dirt, location only.
  Solution??
- $[Right, \textbf{if}\ dirt\ \textbf{then}\ Suck]$

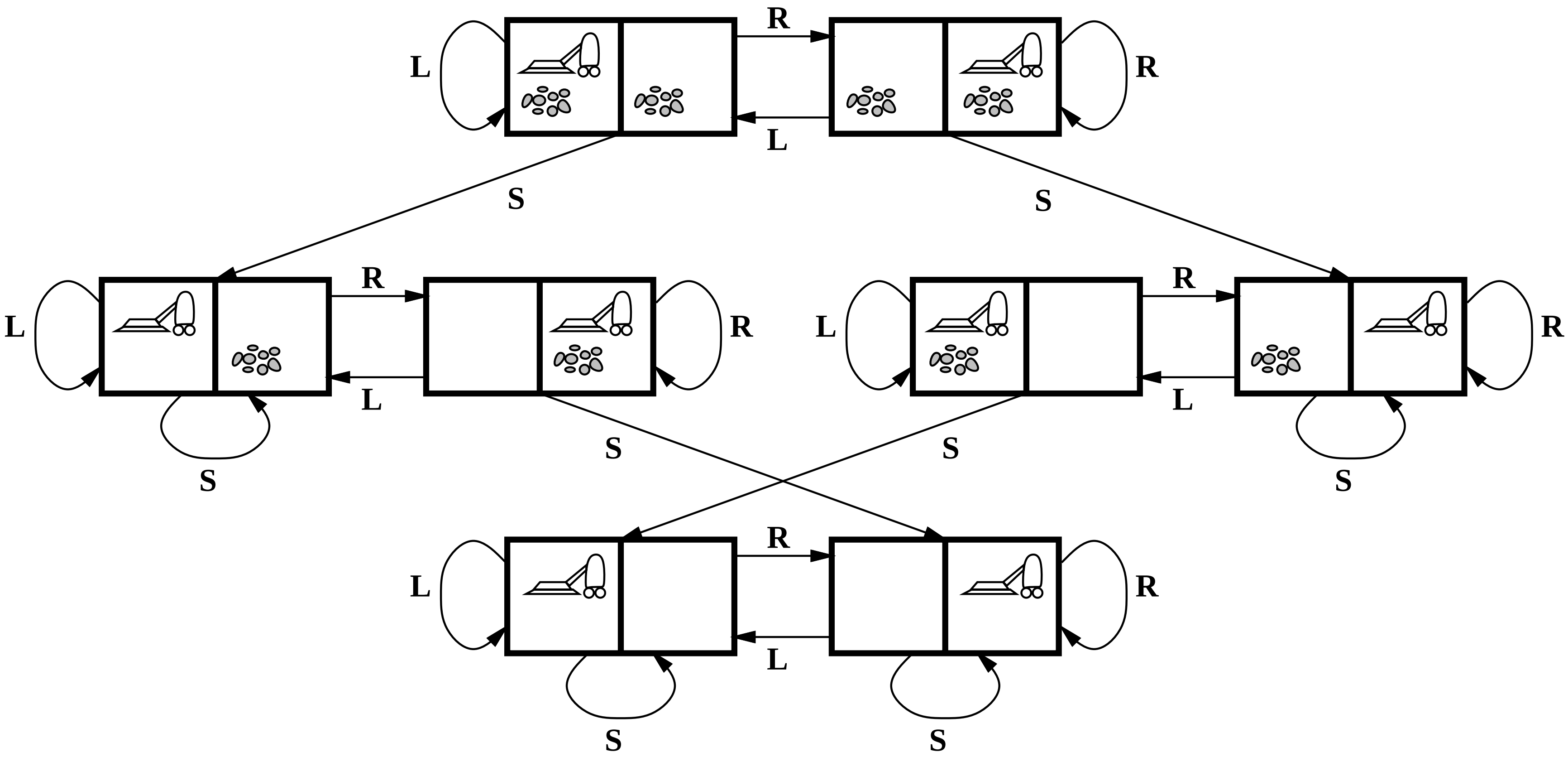
Single-state problem formulation

- A *problem* is defined by four items:
  - *initial state* e.g., “at Arad”
  - *successor function* $S(x)$ = set of action–state pairs
    e.g., $S(Arad) = \{< Arad \rightarrow Zerind, Zerind >, \ldots\}$
  - *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*, 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 $\longrightarrow$ 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!

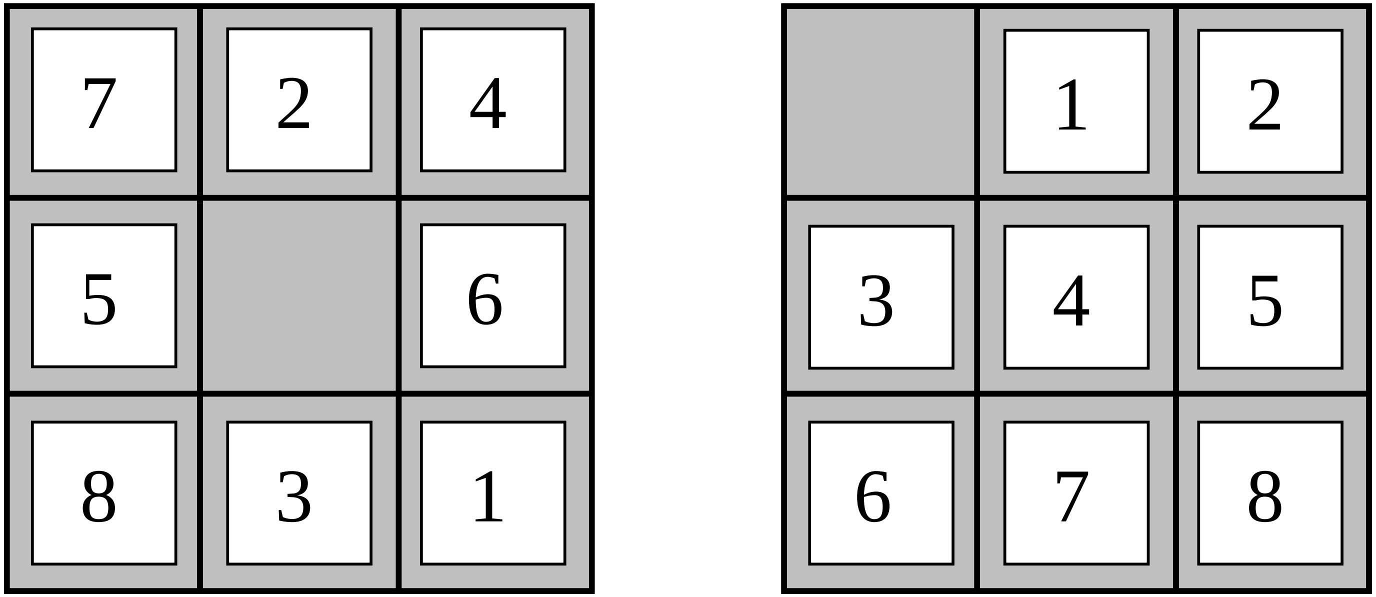
Example: vacuum world state space graph



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

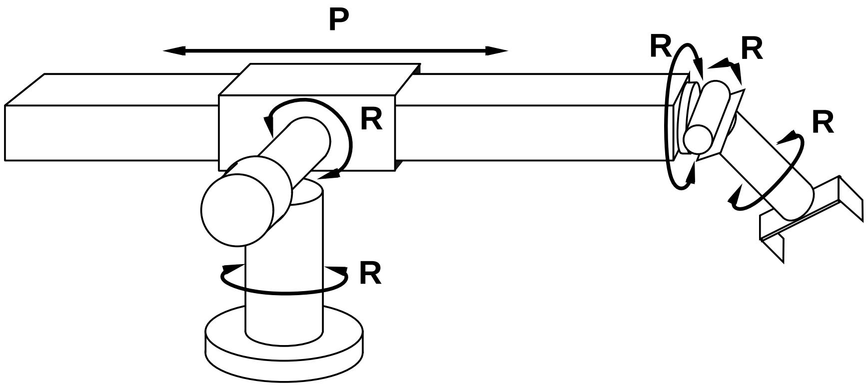


Start State   Goal State

Example: The 8-puzzle

- 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



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)

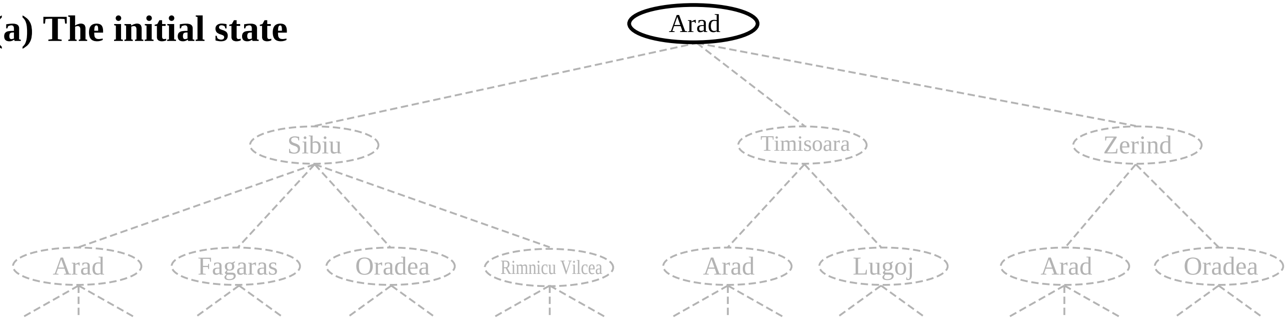
Tree search algorithms

function TREE-SEARCH (*problem, strategy*)
returns a solution, or failure

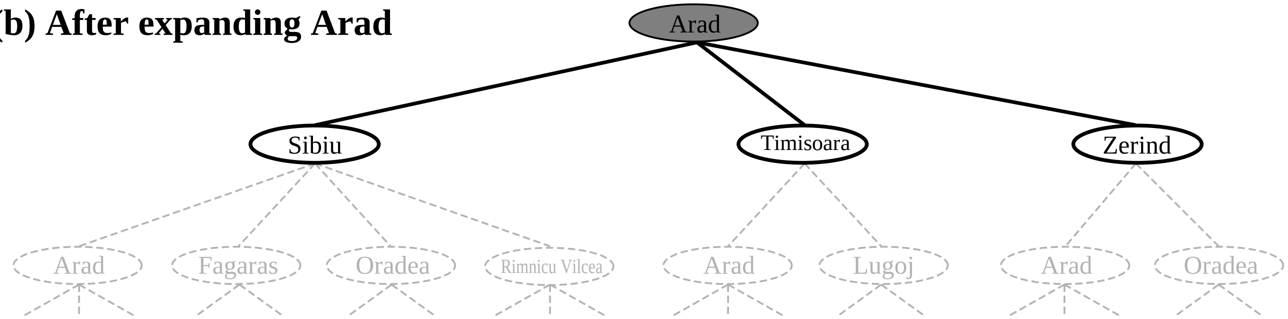
```
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state
    then return the corresponding solution
  else expand the node and add the resulting
    nodes to the search tree
end
```

Tree search example

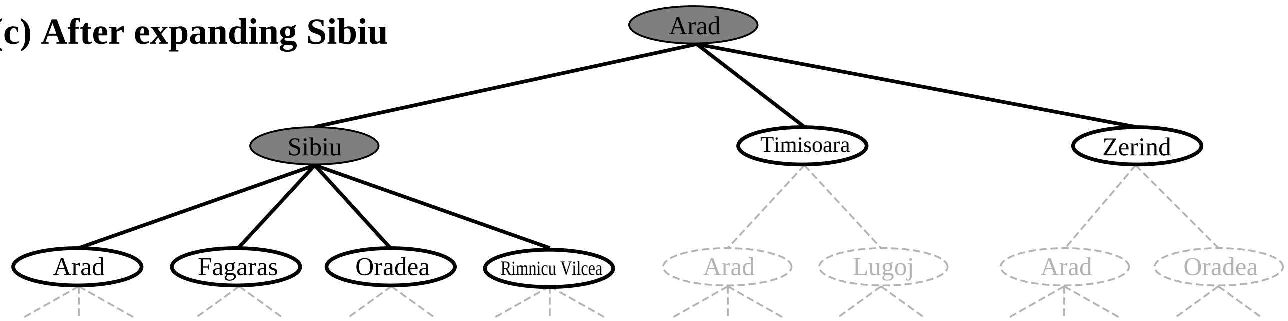
(a) The initial state



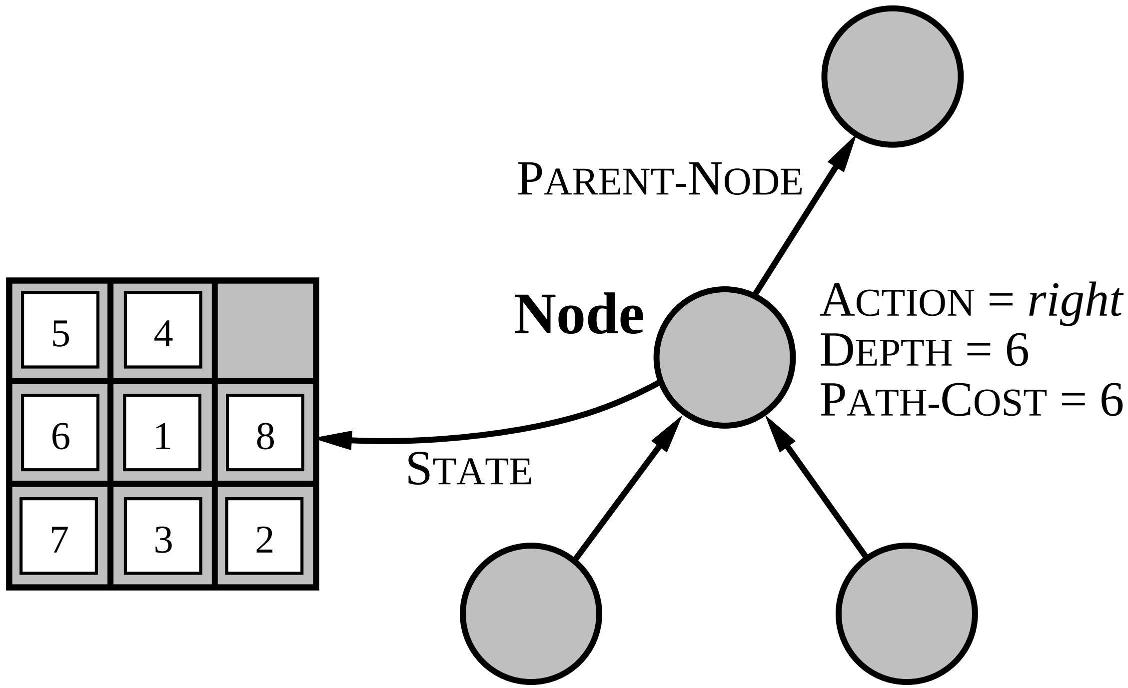
(b) After expanding Arad



(c) After expanding Sibiu



Implementation: states vs. nodes



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 EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Implementation: general tree search

```
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
      STATE[s] ← result
      PARENT-NODE[s] ← node
      ACTION[s] ← action
      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?

Search strategies

- 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

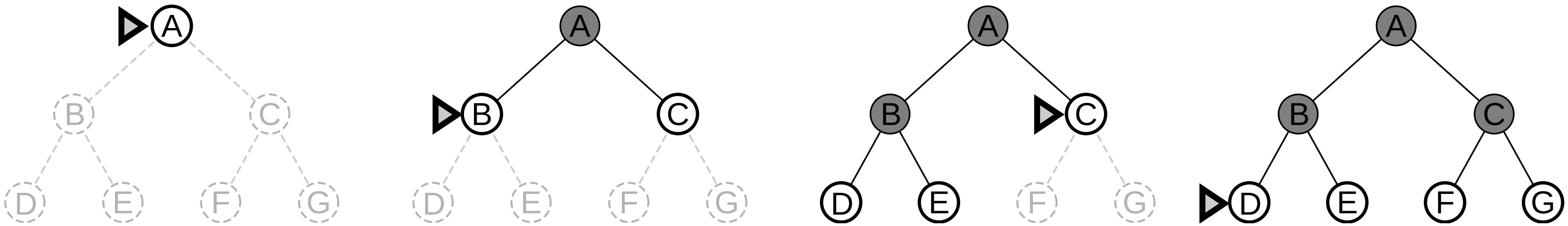
Uninformed strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Iterative broadening search (not in the textbook)

Breadth-first search

- Expand shallowest unexpanded node
- Implementation: *fringe* is a FIFO queue, i.e., new successors go at end

Progress of breadth-first search



Properties of breadth-first search

- Complete: Yes (if b is finite)
- Time: $b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., number of nodes generated is exponential 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 10MB/sec
so 24hrs = 860GB.

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation: *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal

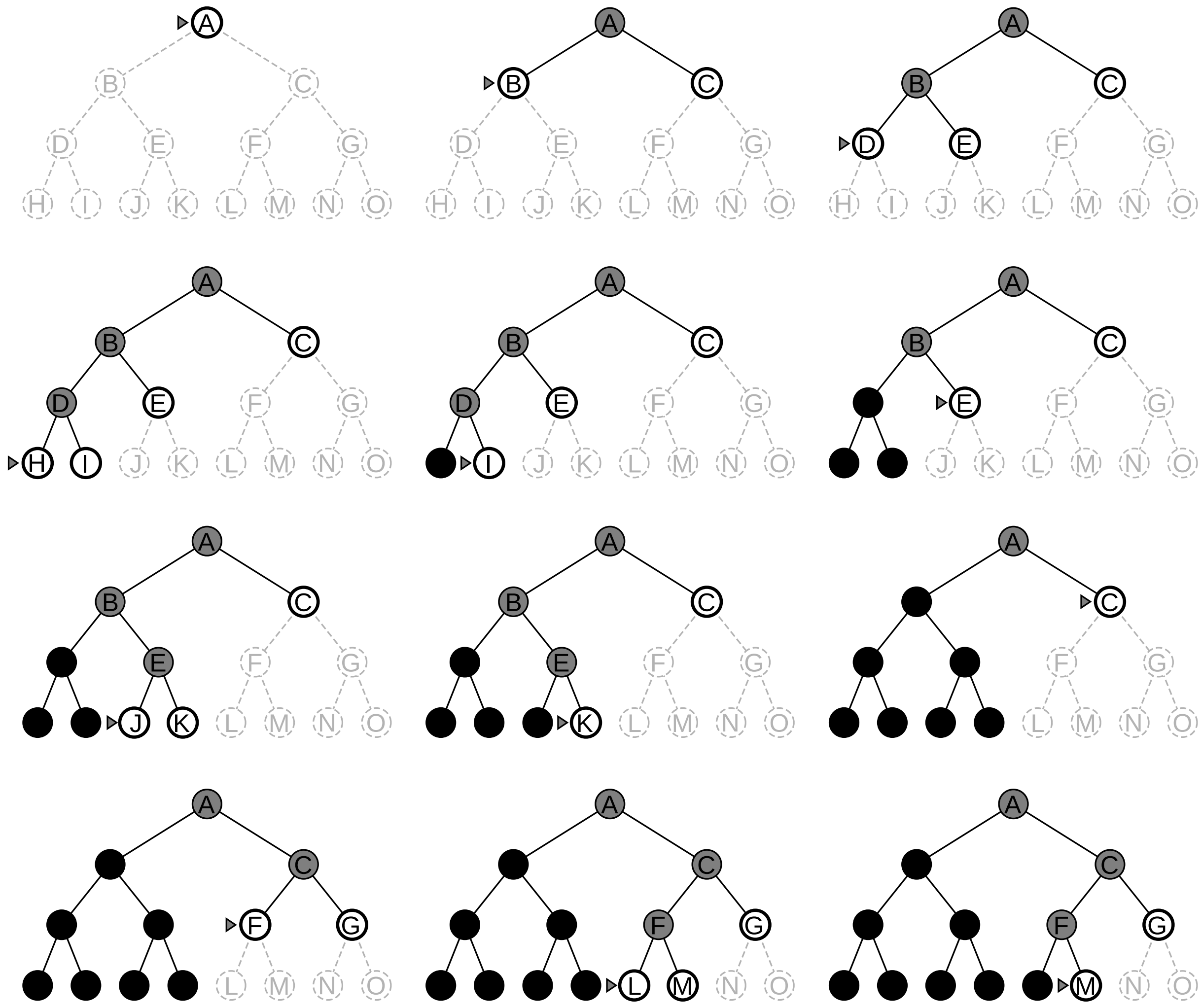
Properties of uniform-cost search

- Complete: Yes, if step cost $\geq \epsilon$
- Time: # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  where C^* is the cost of the optimal solution
- Space: # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- 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

Progress of depth-first search



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
- A recursive implementation is shown on the next page

Depth-limited search

```
function DEPTH-LIMITED-SEARCH (problem, limit)
returns a solution, or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE( INITIAL-STATE[problem]),
      problem,limit)

function RECURSIVE-DLS (node, problem, limit)
returns a solution, or failure/cutoff
    cutoff-occured? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(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

- Complete: No (similar to DFS)
- Time: $O(b^l)$, where l is the depth-limit
- Space: $O(bl)$, i.e., linear space (similar to DFS)
- Optimal: No

Iterative deepening search

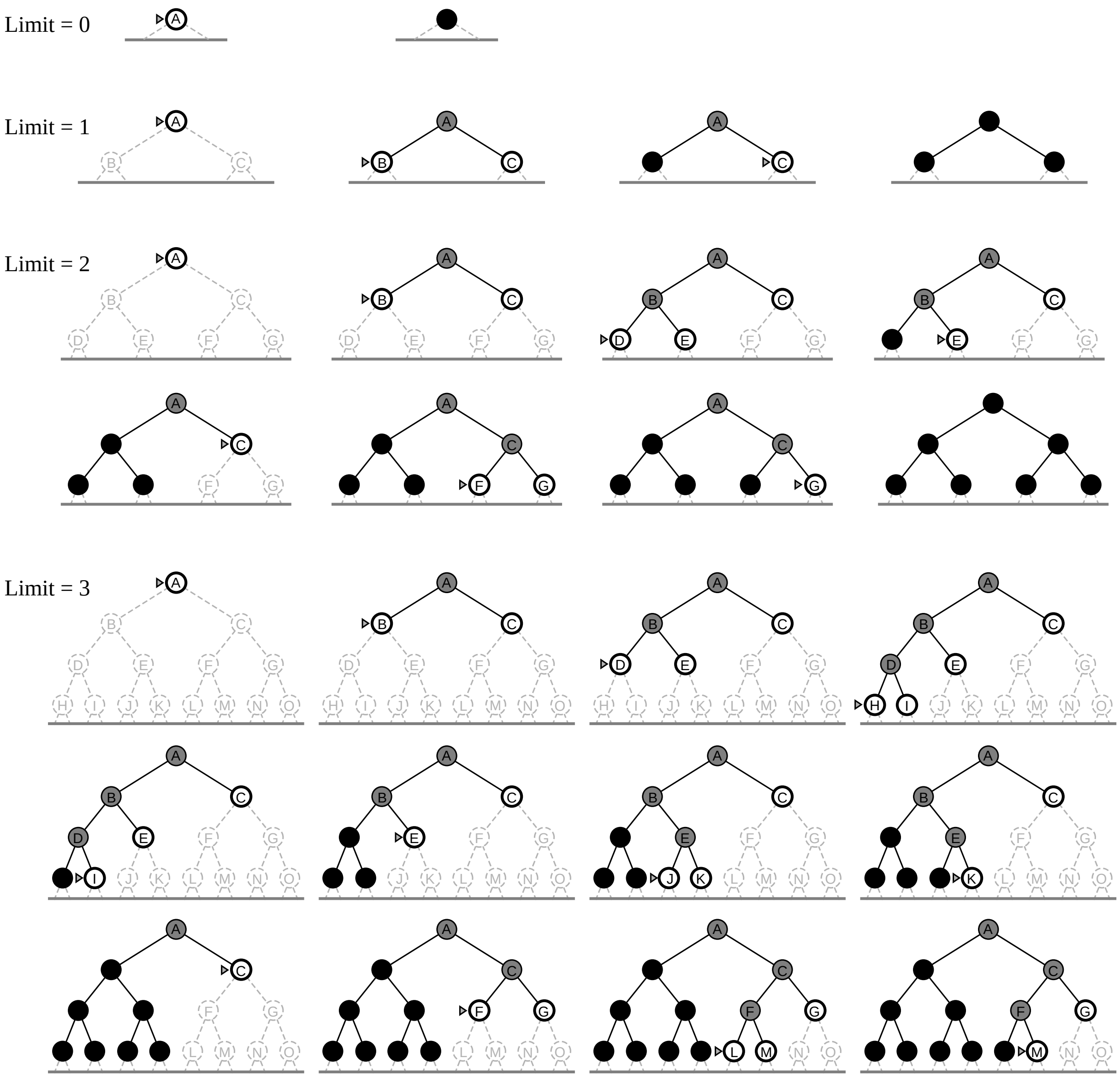
- Do iterations of depth-limited search starting with a limit of 0 . If you fail to find a goal with a particular depth limit, increment it and continue with the iterations.
- Combines the linear space complexity of DFS with the completeness property of BFS.

Iterative deepening search

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

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

Iterative deepening search



Properties of iterative deepening search

- Complete: Yes
- Time: $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space: $O(bd)$
- Optimal: Yes, if step cost = 1
  Can be modified to explore uniform-cost tree

Numerical comparison of the number of nodes generated for $b = 10$ and $d = 5$, solution at far right:

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

Iterative broadening search

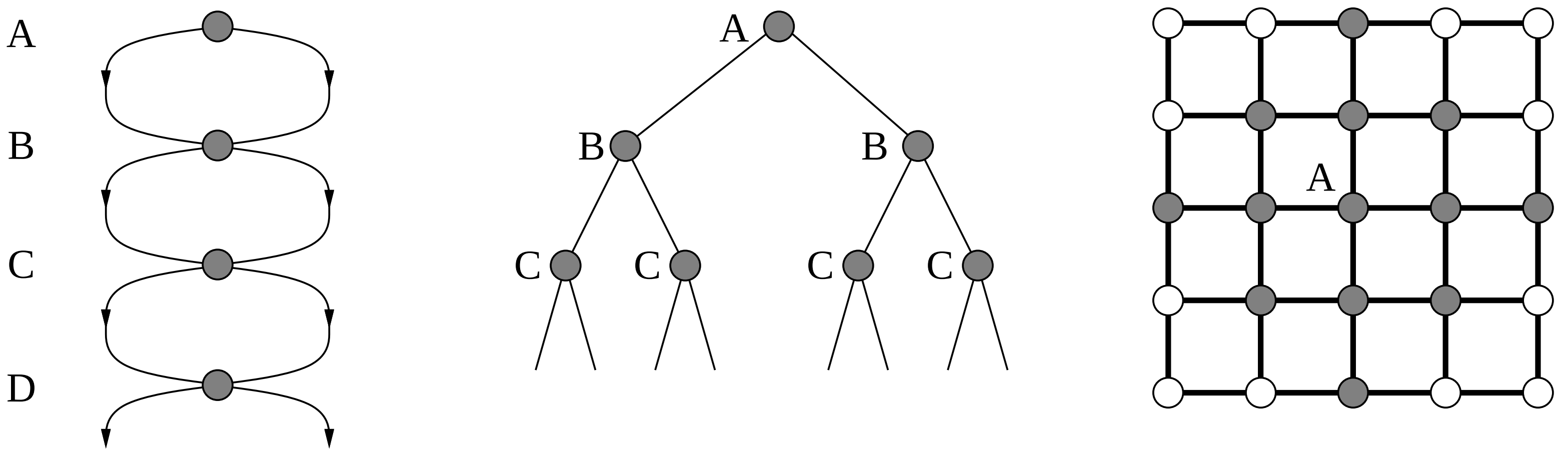
- Iterative deepening is iterations of DFS with a depth cutoff. Iterative broadening is iterations of DFS with a breadth cutoff.
- Iterate c from 2 to b, where b is the maximum branching factor. At every iteration, take only c children of every node expanded, simply discard the remaining children.
- Algorithm??
- Properties??

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}	$b^{1+\lfloor C^*/\epsilon \rfloor}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{1+\lfloor C^*/\epsilon \rfloor}$	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!



(a) (b) (c)

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 EMPTY?(*fringe*) **then return** failure
  node ← REMOVE-FIRST(*fringe*)
  if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
    then return SOLUTION(*node*)
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERT-ALL(EXPAND(*node, problem*), *fringe*)

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