Solving Problems by Searching

AIMA Sections 3.1–3.3

Outline

- ◇ Problem-solving agents
- ◇ Problem types
- ◇ Problem formulation
- ◇ Example problems
- ◇ General search algorithm

Problem-solving agents

```
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 ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Problem-solving agents

Restricted form of general agent: **Goal based agents**

- formulate a goal and a problem given the current state
- search for a solution
- execute the solution **ignoring** perceptions

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

An example: Traveling in Romania

Example (Holidays in Romania)

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

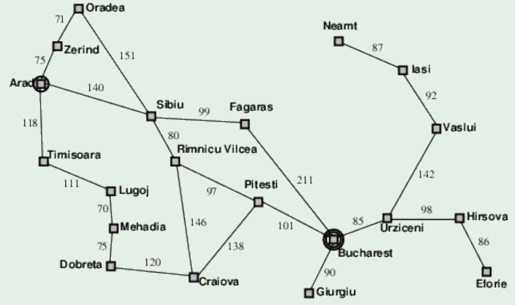
Formulate goal:
- be in Bucharest

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

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

An example: Traveling in Romania

Example (Holidays in 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

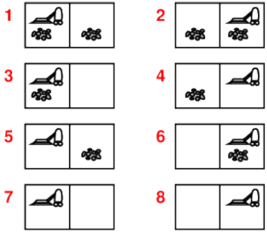
solution is a contingent plan or a policy

often **interleave** search, execution

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

Example: vacuum world

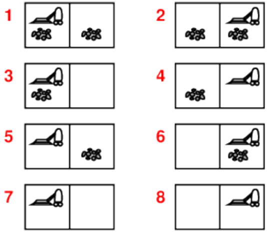
Single-state, start in #5. Solution??



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??

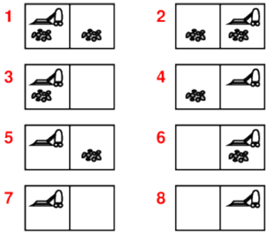


Example: vacuum world

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??

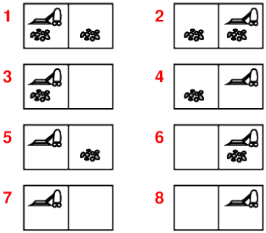


Example: vacuum world

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., $S(A) = \{< Arad \rightarrow Zerind, Zerind >, \ldots\}$

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.

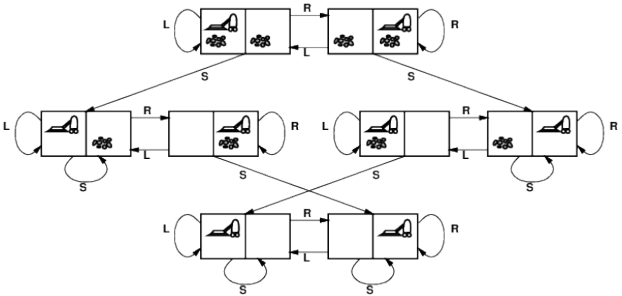
$c(x, a, y)$ is the step cost, assumed to be ≥ 0

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

Selecting a state space

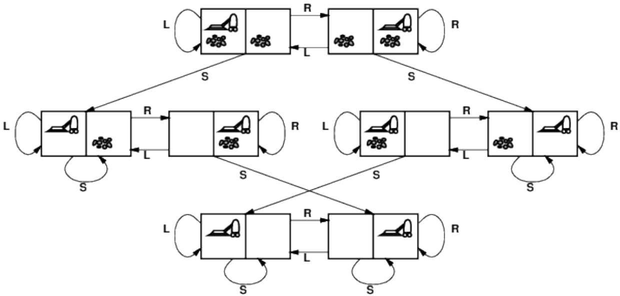
Real world is absurdly complex
$\Rightarrow$ state space must be **abstracted** for problem solving
(Abstract) state = set of real states
(Abstract) action = complex combination of real actions
e.g., "Arad $\rightarrow$ Zerind" represents a complex set
of possible routes, detours, rest stops, etc.
For 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



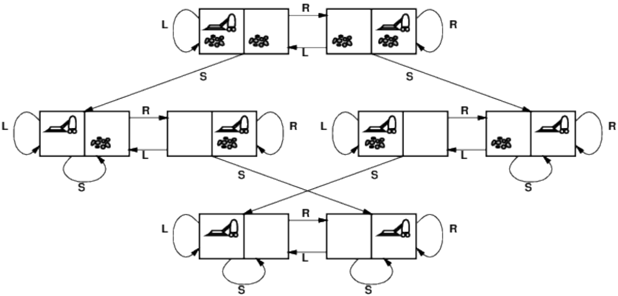
states??:
actions??:
goal test??:
path cost??:

Example: vacuum world state space graph



states??: discrete dirt and robot locations (ignore dirt amounts)
actions??:
goal test??:
path cost??:

Example: vacuum world state space graph



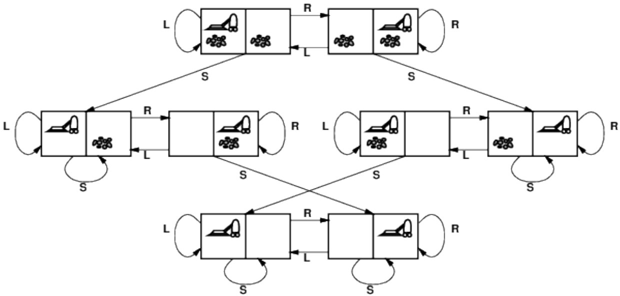
states??: discrete dirt and robot locations (ignore dirt amounts)

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

goal test??:

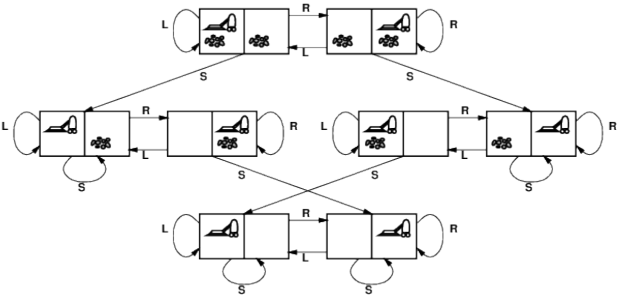
path cost??:

Example: vacuum world state space graph



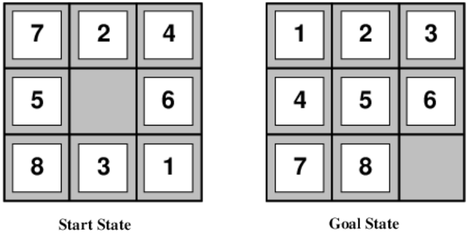
states??: discrete dirt and robot locations (ignore dirt amounts)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??:

Example: vacuum world state space graph



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

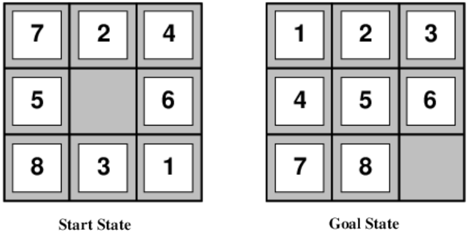
Example: The 8-puzzle



states??:

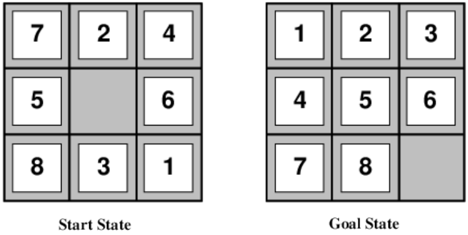
actions??:
goal test??:
path cost??:

Example: The 8-puzzle



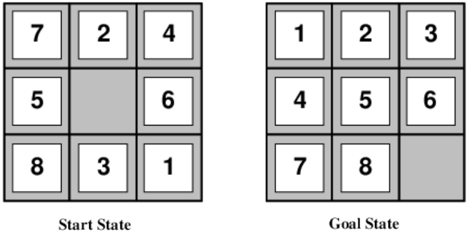
states??: integer locations of tiles (ignore intermediate positions)
actions??:
goal test??:
path cost??:

Example: The 8-puzzle



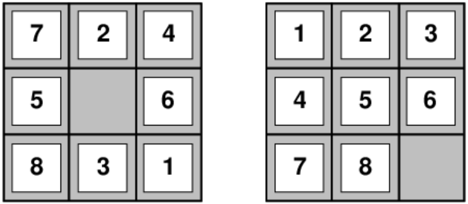
states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down
goal test??:
path cost??:

Example: The 8-puzzle



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

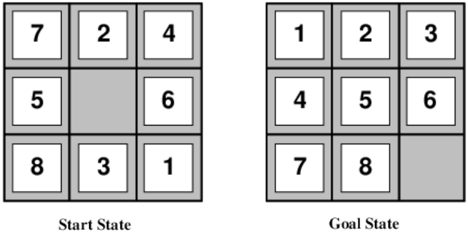
Example: The 8-puzzle



Start State   Goal State

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

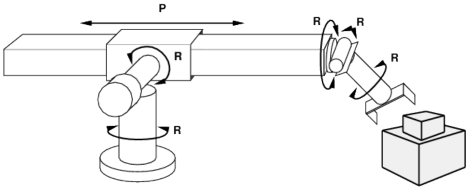
Example: The 8-puzzle



states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down
goal test??: given goal state
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!**

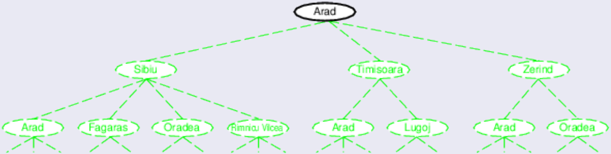
path cost??: time to execute

Tree search algorithm

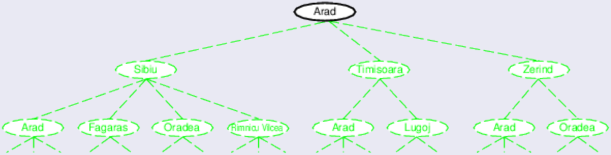
Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

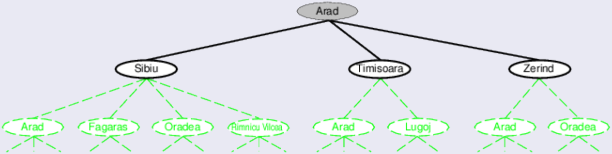
function Tree-Search(*problem, strategy*) **returns** a solution, or failure
initialize the search tree using the initial state of *problem*
loop do
if no candidates for expansion **then return** failure
choose a leaf node for expansion according to *strategy*
if node contains a goal state **then return** the solution
else add successor nodes to the search tree (expansion)
end

Tree search example

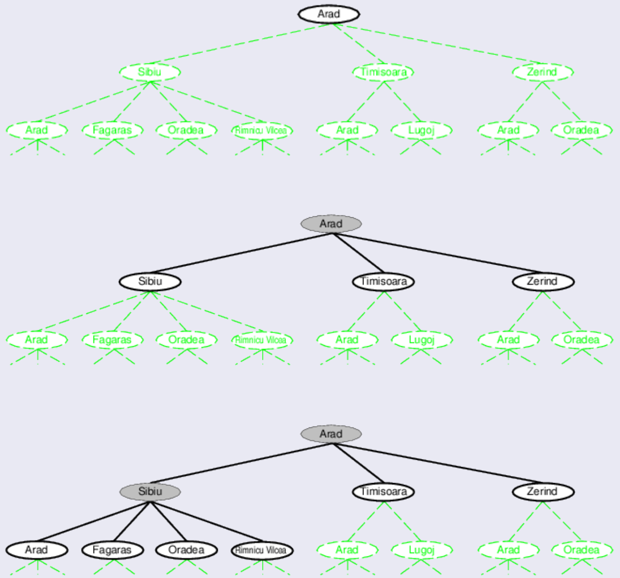


Tree search example





Tree search example



Implementation: states vs. nodes

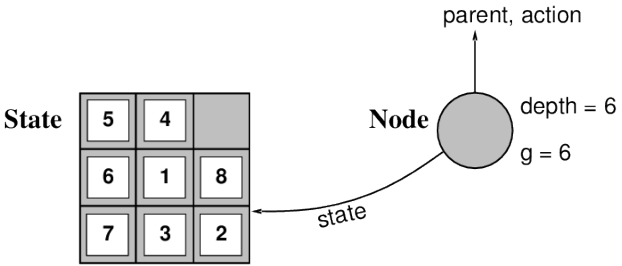
A state is a (representation of) a physical configuration

A node is a data structure constituting part of a search tree
  includes parent, action, children, depth, path cost (i.e., $g(x)$)

States do not have parents, actions,children, depth, or path cost!

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, action, children, depth, path cost (i.e., $g(x)$)
States do not have parents, actions,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, frontier) returns a solution, or failure
  frontier ← Insert(Make-Node(problem.Initial-State))
  loop do
    if frontier is empty then return failure
    node ← Pop(frontier)
    if problem.Goal-Test(node.State) then return node
    frontier ← InsertAll(Expand(node, problem))
  end loop
```

Implementation: expand nodes

```
function Expand( node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in Successor-Fn(problem, node.State)
    do
       s ← a new Node
       s.Parent-Node ← node;
       s.Action ← action;
       s.State ← result
       s.Path-Cost ← node.Path-Cost +
                     Step-Cost(node.State, action, result)
       s.Depth ← node.Depth + 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:

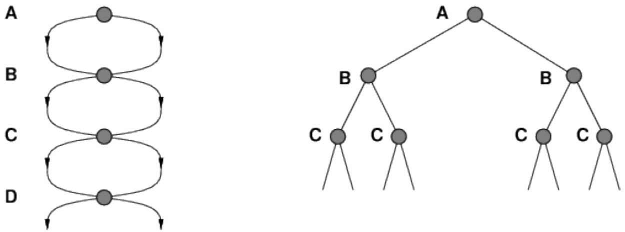
- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

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 ∞)

Repeated states

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



Graph search


```
function Graph-Search( problem, frontier) returns a solution, or failure
  explored ← an empty set
  frontier ← Insert(Make-Node(problem.Initial-State))
  loop do
    if frontier is empty then return failure
    node ← Pop(frontier)
    if problem.Goal-Test(node.State) then return node
    if node.State is not in explored then
      add node.State to explored
      frontier ← InsertAll(Expand(node, problem))
    end if
  end loop
```