2

Problem Solving and Search

Claudia Chirita

School of Informatics, University of Edinburgh

Based on slides by: Jacques Fleuriot, Michael Rovatsos, Michael Herrmann, Vaishak Belle

2.a

Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Restricted form of general agent

EXAMPLE · ROMANIA

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

Formulate goal:

be in Bucharest

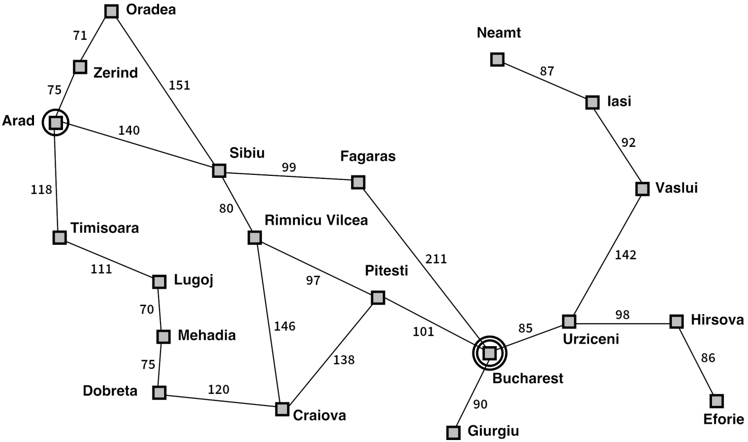
Formulate problem:

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 $\longrightarrow$ sensorless problem (conformant)

agent may have no idea where it is

solution is a sequence

Nondeterministic and/or **partially observable** $\longrightarrow$ contingency problem

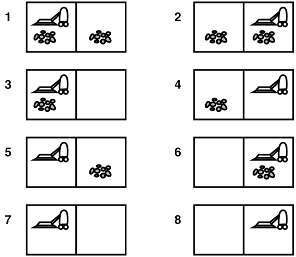
percepts provide new information about current state

often interleave search, execution

Unknown state space $\longrightarrow$ exploration problem

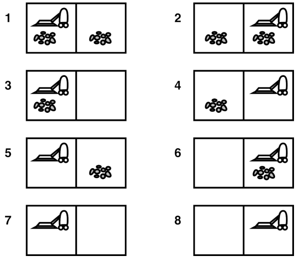
EXAMPLE · VACUUM WORLD

❓ **Single-state**, start in 5.



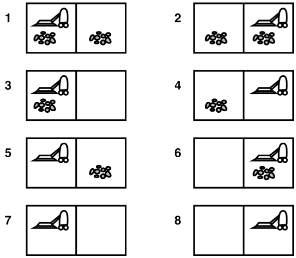
EXAMPLE · VACUUM WORLD

Single-state, start in 5.
Solution: *[Right, Suck]*



EXAMPLE · VACUUM WORLD

Single-state, start in 5.
Solution: *[Right, Suck]*

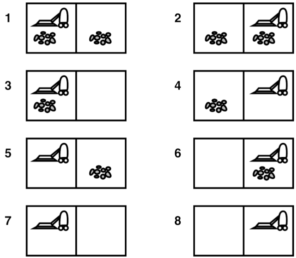


Sensorless, start in {1 ,2, 3, 4, 5, 6, 7, 8}.
e.g. *Right* goes to {2, 4, 6, 8}.

EXAMPLE · VACUUM WORLD

Single-state, start in 5.
Solution: *[Right, Suck]*

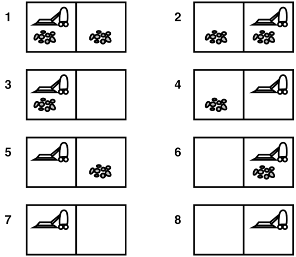
Sensorless, start in {1 ,2, 3, 4, 5, 6, 7, 8}.
e.g. *Right* goes to {2, 4, 6, 8}.
Solution: *[Right, Suck, Left, Suck]*



EXAMPLE · VACUUM WORLD

Single-state, start in 5.
Solution: *[Right, Suck]*

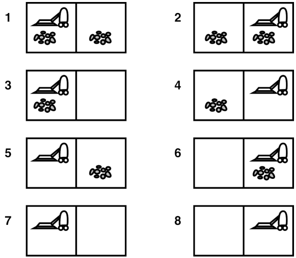
Sensorless, start in {1 ,2, 3, 4, 5, 6, 7, 8}.
e.g. *Right* goes to {2, 4, 6, 8}.
Solution: *[Right, Suck, Left, Suck]*



Contingency, start in 5.
Nondeterminism: *Suck* may dirty a clean carpet
Partially observable: can only see dirt at current location

EXAMPLE · VACUUM WORLD

Single-state, start in 5.
Solution: *[Right, Suck]*



Sensorless, 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.
Nondeterminism: *Suck* may dirty a clean carpet
Partially observable: can only see dirt at current location
Solution: *[Right, **if** dirt **then** Suck]*

2.b

Problem formulation

SINGLE-STATE PROBLEM FORMULATION

PROBLEM defined by:

1. **initial state** e.g. "at Arad"
2. **successor function** $S(x)$ = set of action–state pairs

   e.g. $S(\mathit{Arad}) = \{\langle \mathit{Arad} \rightarrow \mathit{Zerind}, \mathit{Zerind} \rangle, \ldots\}$
3. **goal test**

   explicit e.g. x = "at Bucharest"

   implicit e.g. $\mathit{Checkmate}(x)$
4. **path cost** (additive)

   e.g. sum of distances, number of actions executed, etc.

   $c(x, a, y)$ · the step cost of taking action a in state x to reach state y; assumed to be $\geqslant 0$

A **solution** is a sequence of actions 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

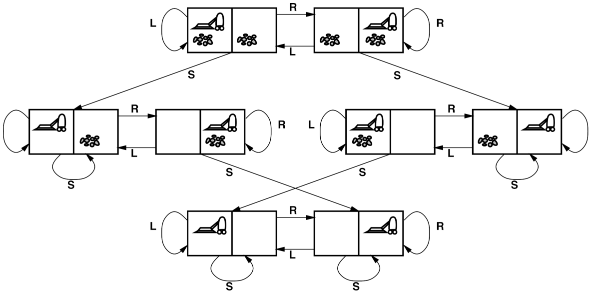
(Abstract) action = complex combination of real actions

- 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, solutions in the real world

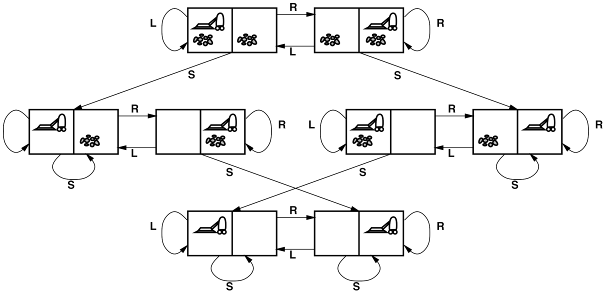
Each abstract action should be easier than the original problem!

EXAMPLE · VACUUM WORLD STATE SPACE GRAPH



- states
- actions
- goal test
- path cost

EXAMPLE · VACUUM WORLD STATE SPACE GRAPH



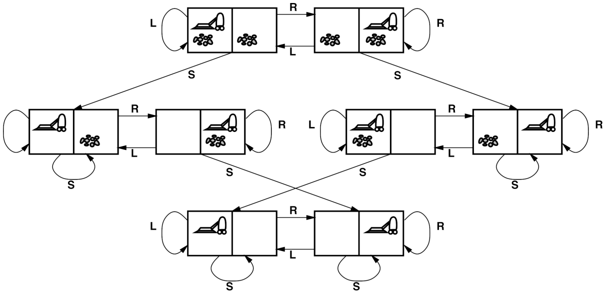
states pair of dirt and robot locations

actions

goal test

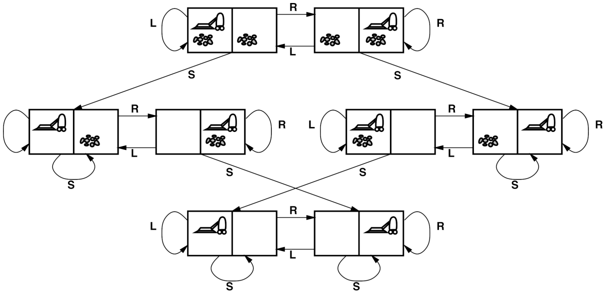
path cost

EXAMPLE · VACUUM WORLD STATE SPACE GRAPH



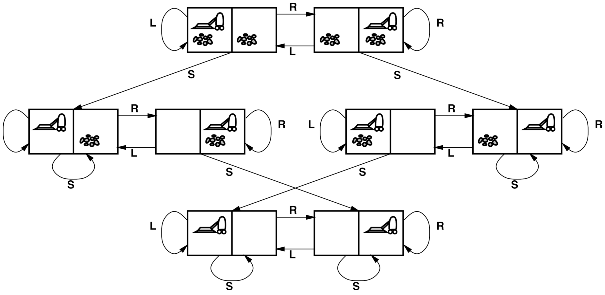
states	pair of dirt and robot locations
actions	*Left, Right, Suck*
goal test	
path cost	

EXAMPLE · VACUUM WORLD STATE SPACE GRAPH



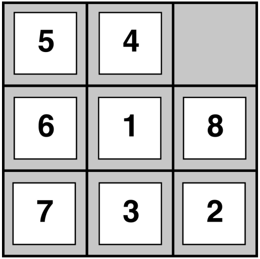
states	pair of dirt and robot locations
actions	*Left, Right, Suck*
goal test	no dirt at any location
path cost	

EXAMPLE · VACUUM WORLD STATE SPACE GRAPH

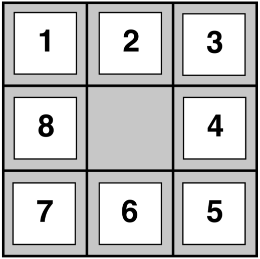


states	pair of dirt and robot locations
actions	*Left, Right, Suck*
goal test	no dirt at any location
path cost	1 per action

EXAMPLE · THE 8-PUZZLE



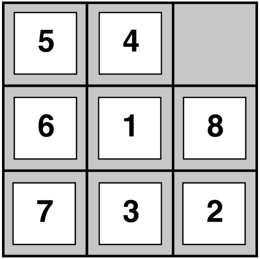
Start State



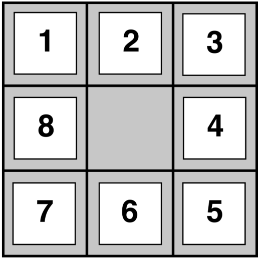
Goal State

- states
- actions
- goal test
- path cost

EXAMPLE · THE 8-PUZZLE



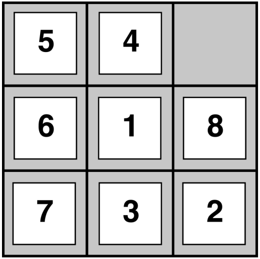
Start State



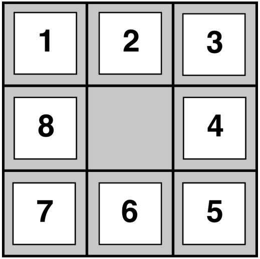
Goal State

states	integer locations of tiles
actions	
goal test	
path cost	

EXAMPLE · THE 8-PUZZLE



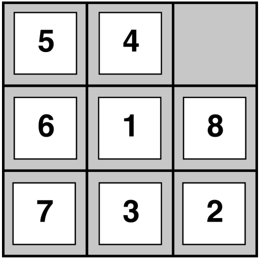
Start State



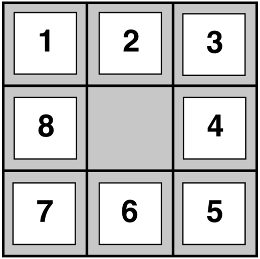
Goal State

states	integer locations of tiles
actions	move blank left, right, up, down
goal test	
path cost	

EXAMPLE · THE 8-PUZZLE



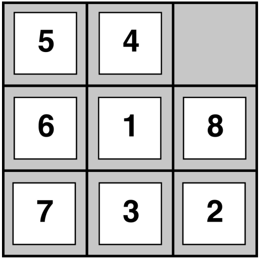
Start State



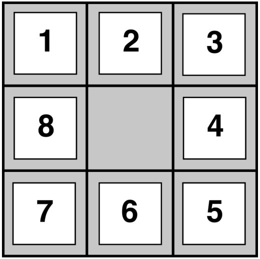
Goal State

states	integer locations of tiles
actions	move blank left, right, up, down
goal test	= goal state (given)
path cost	

EXAMPLE · THE 8-PUZZLE



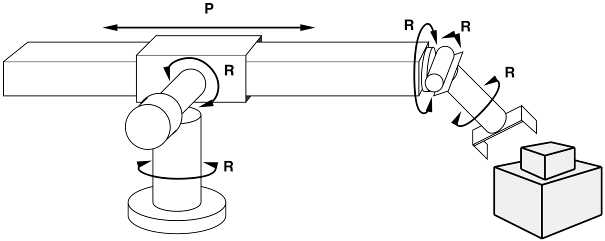
Start State



Goal State

states	integer locations of tiles
actions	move blank left, right, up, down
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
path cost	time to execute

2.c

Searching for solutions

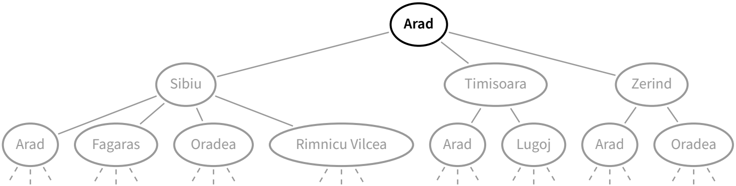
TREE SEARCH ALGORITHMS

Basic idea:

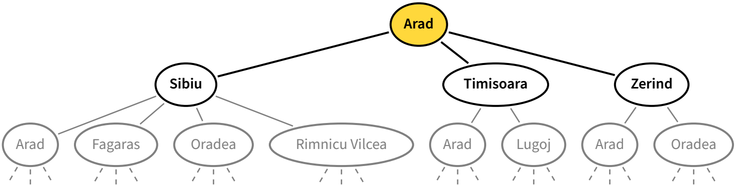
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. *expanding* states)

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

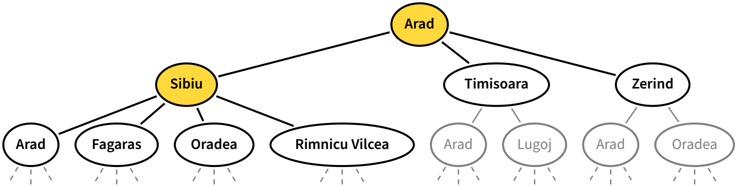
TREE SEARCH EXAMPLE



TREE SEARCH EXAMPLE



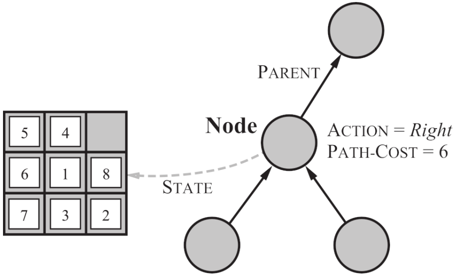
TREE SEARCH EXAMPLE



A **state** is a (representation of) a physical configuration.

A **node** is a data structure constituting part of a search tree
includes *state*, *parent*, *action*, *path cost*.

Using these it is easy to compute the components for a child node.



IMPLEMENTATION · GENERAL TREE SEARCH

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

function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT(*parent*.STATE, *action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)

TAKE-HOME MESSAGE

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.