Problem solving and search

AI

Problem Solving Agents

- When the correct action to take is not immediately obvious, an agent may need to plan ahead to consider a **sequence of actions** that form a ***path to a goal state*.**
- Such agents is called a problem-solving agent, and the computational process it undertakes is called **search**.

- Problem Solving Phases (four-phase problem-solving process):
  - **Goal formulation:** Goals organize behaviour by limiting the objectives and hence the actions to be considered.
  - **Problem formulation:** The agent devises a description of the states and actions necessary to reach the goal.
  - **Search:** Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal.
    - Such a sequence is called a solution.
  - **Execution**: The agent can now execute the actions in the solution, one at a time.

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

- Credits: AIMA by Peter N.

Example: Romania

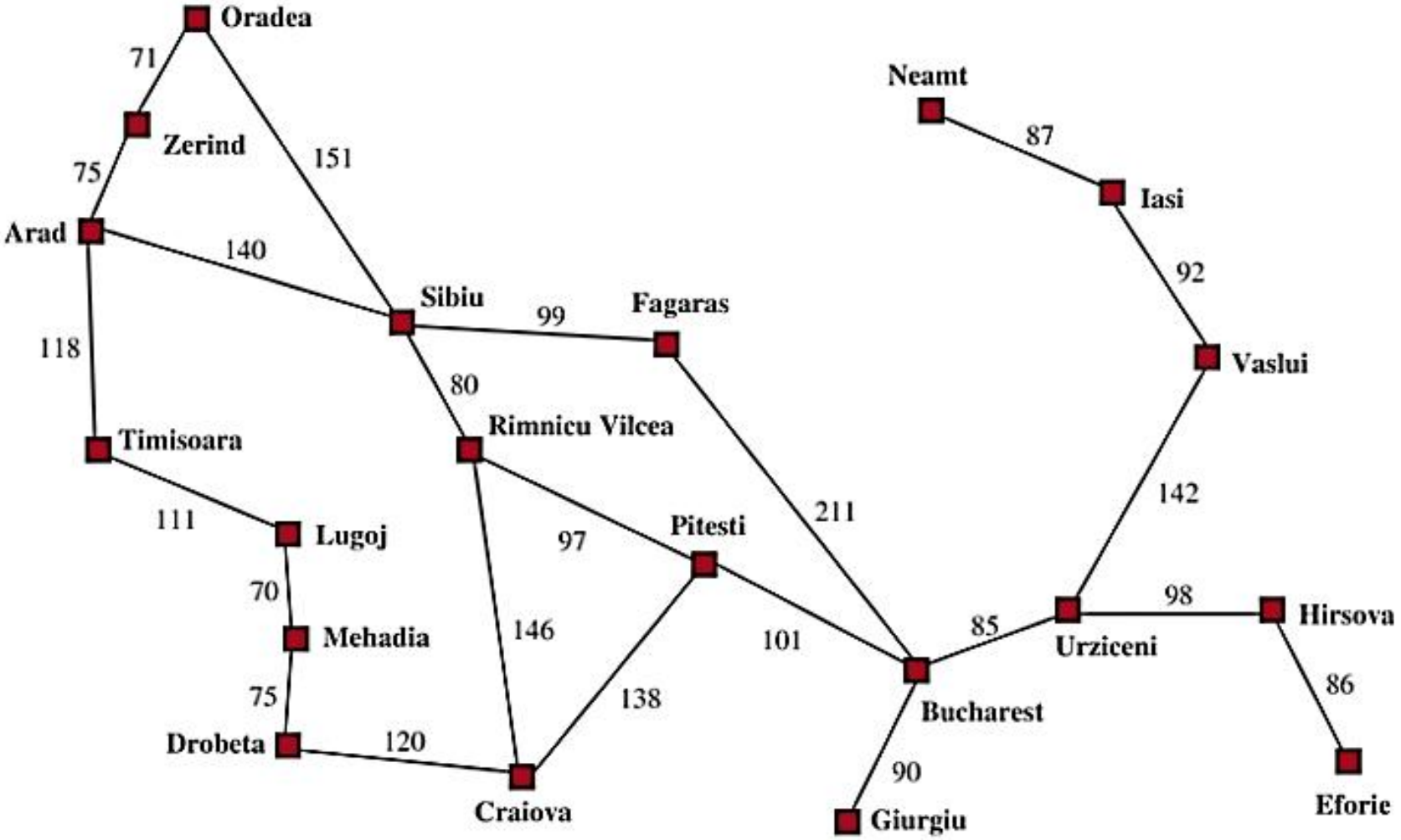


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

- Credits: AIMA by Peter N.

Search problems and solutions

- A search problem can be defined formally as follows:
  - A set of possible **states** that the environment can be in. We call this the **state space**.
  - The **initial state** that the agent starts in. For example: Arad.
  - A set of **one or more goal states**.
  - The **actions** available to the agent.
    - » ACTIONS (Arad) = {ToSibiu, ToTimisoara, ToZerind }
  - **A transition model**, which describes what each action does.
  - An **action cost function**, denoted by ACTION-COST(s,a,s").

Search problems and solutions

- A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state.

- An **optimal solution** has the lowest path cost among all solutions.

- The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.

Formulating problems

- Our formulation of the problem of getting to goal is a **model—**an abstract mathematical description.

- The process of removing detail from a representation is called **abstraction**.

- A good problem formulation has the right level of detail.

*"The choice of a **good abstraction** involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out."*

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

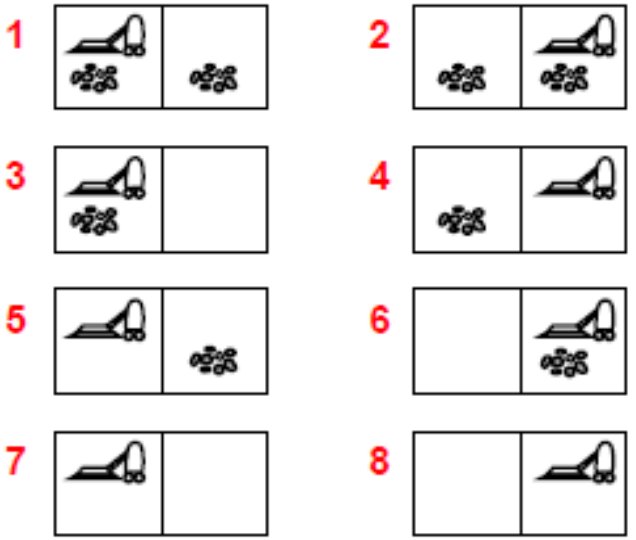
Unknown state space $\Longrightarrow$ exploration problem ("online")

Example: vacuum world

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

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??
$[Right, \text{if } dirt \text{ then } Suck]$



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

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

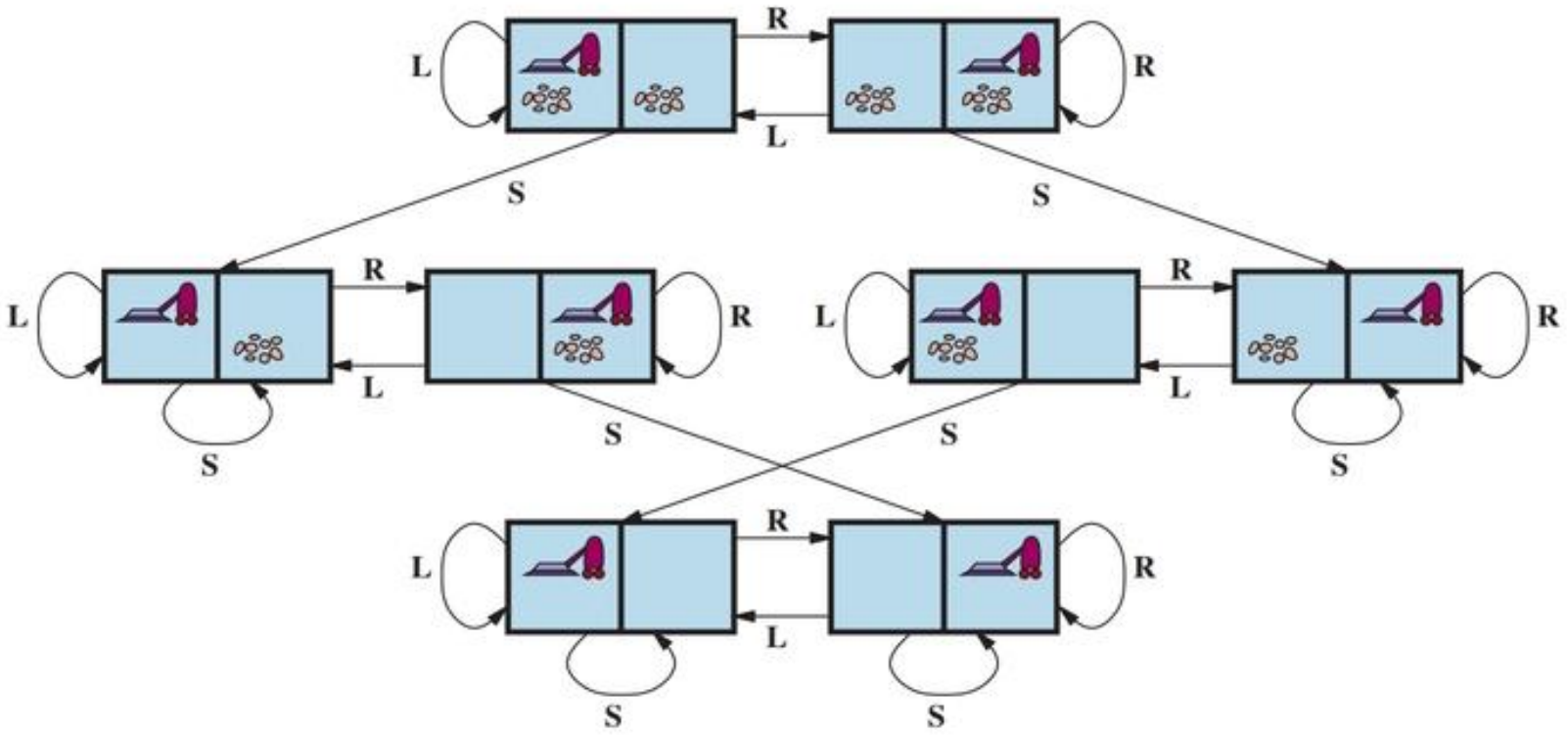
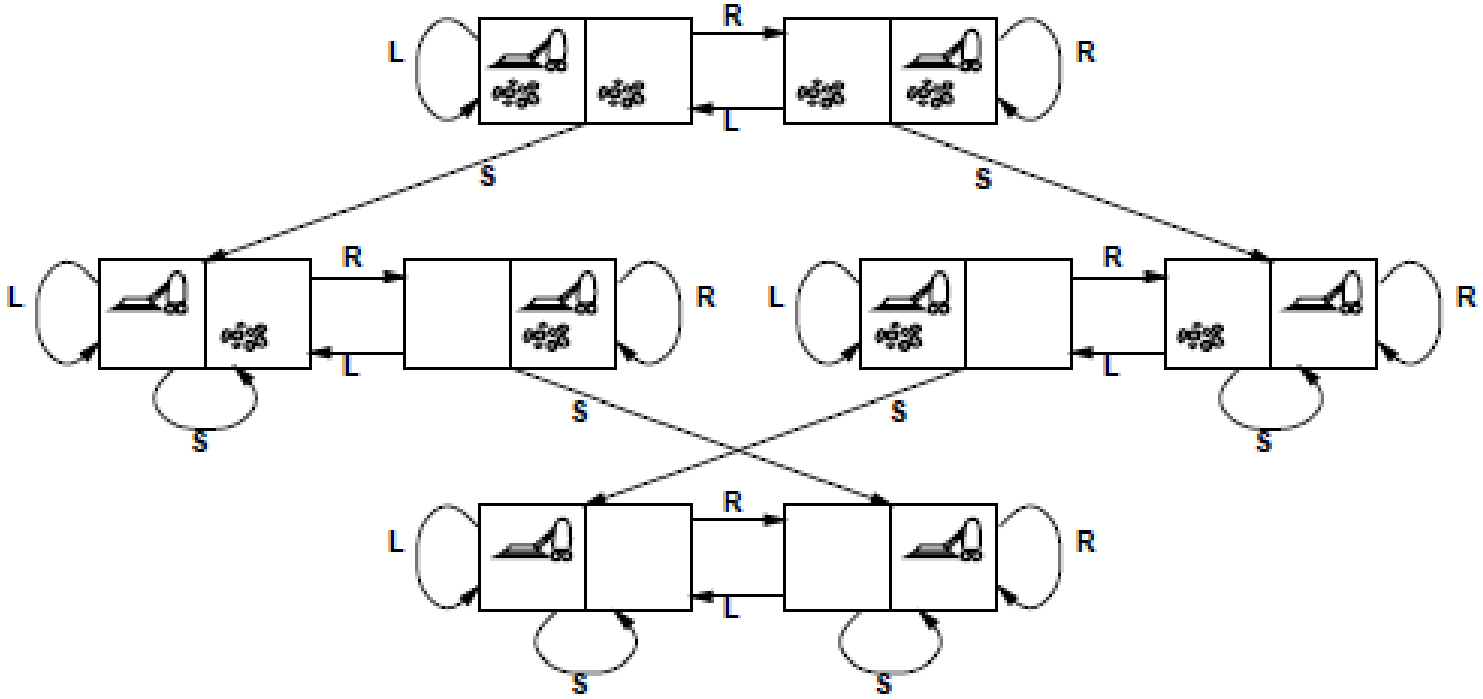


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Example: vacuum world state space graph



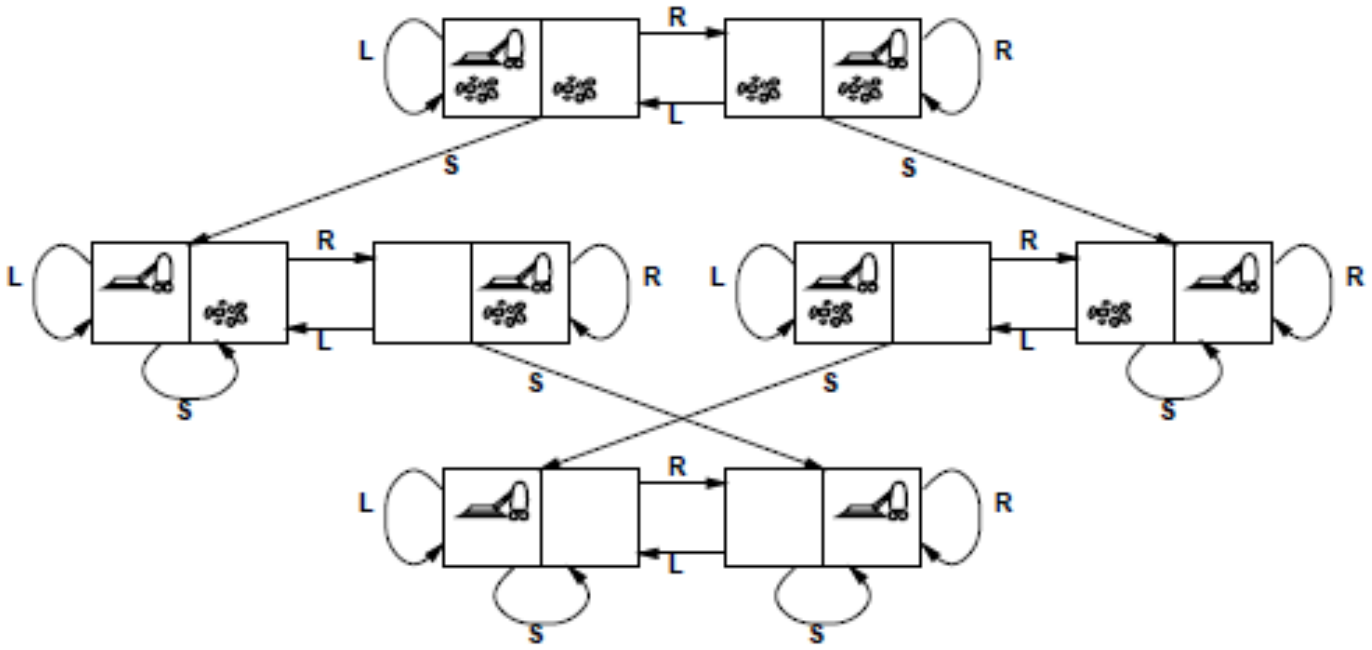
states??

actions??

goal test??

path cost??

Example: vacuum world state space graph



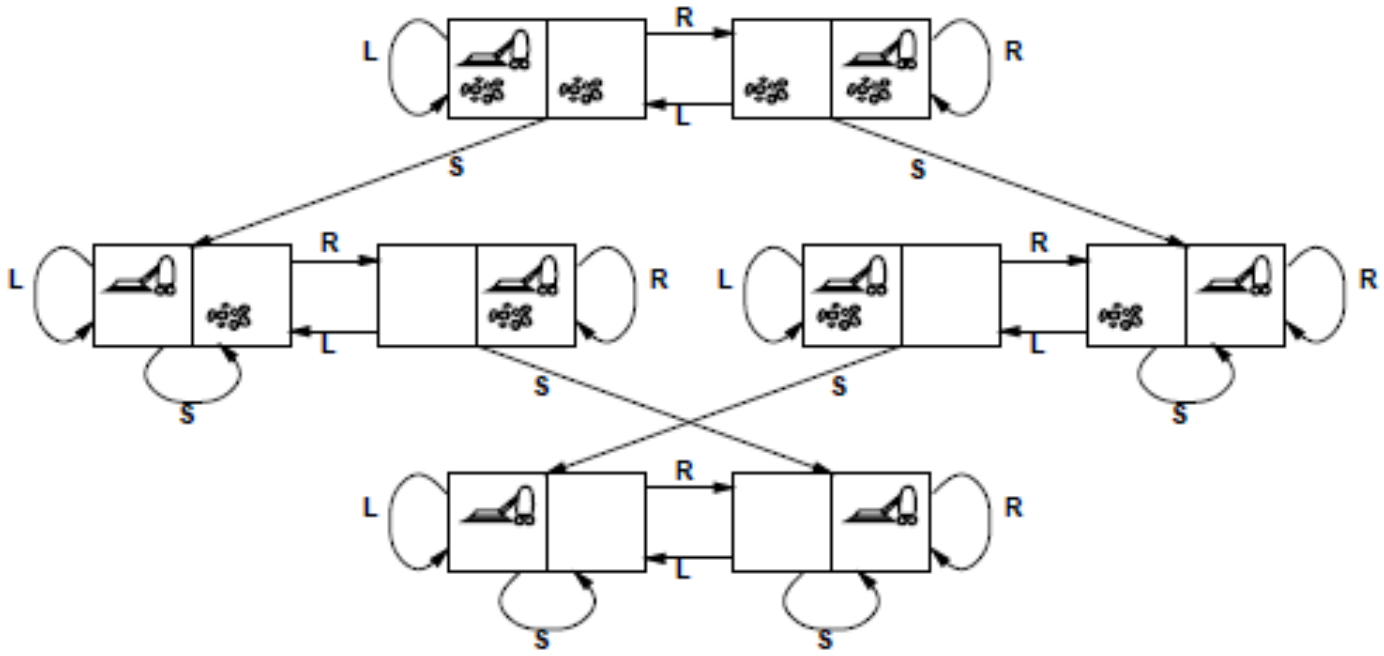
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??

goal test??

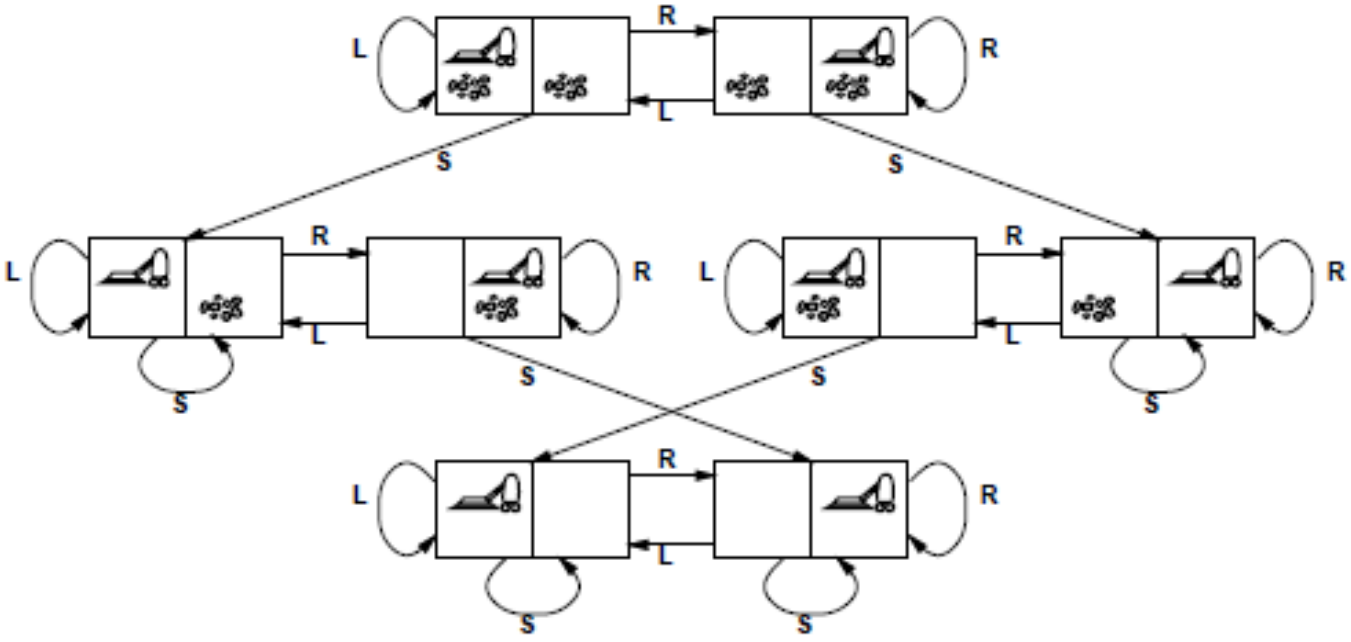
path cost??

Example: vacuum world state space graph



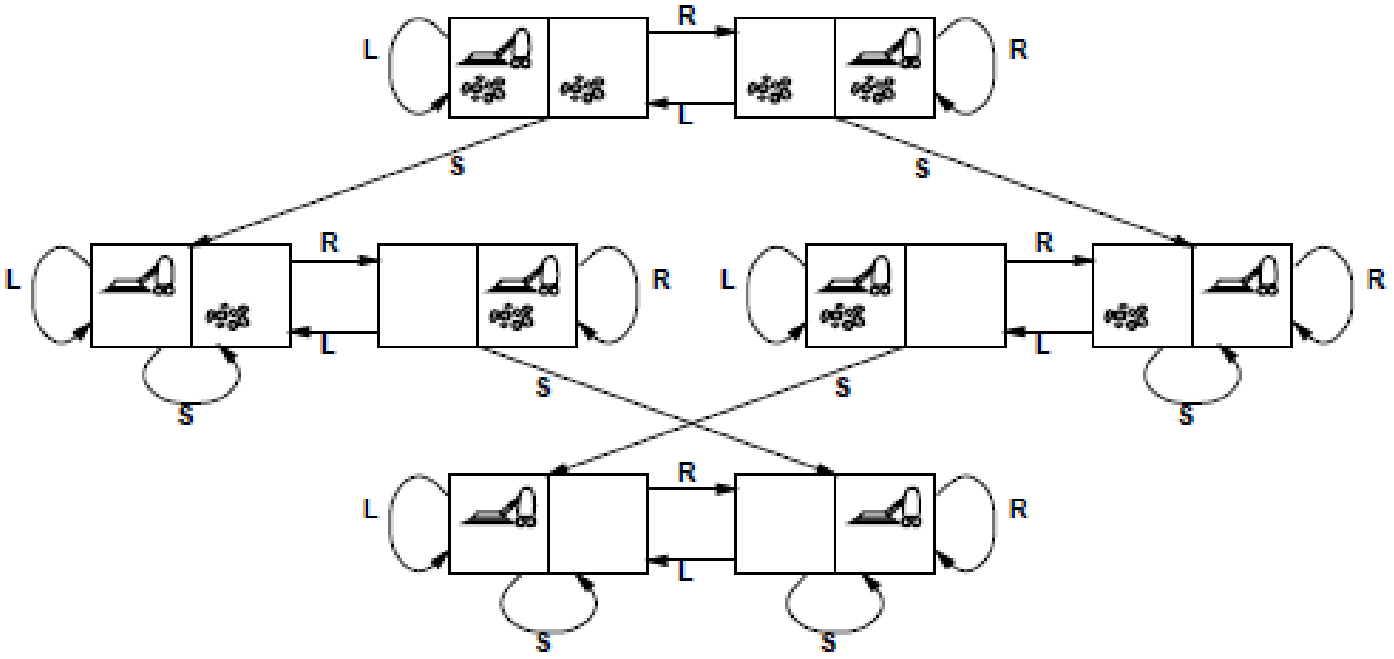
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??
path cost??

Example: vacuum world state space graph



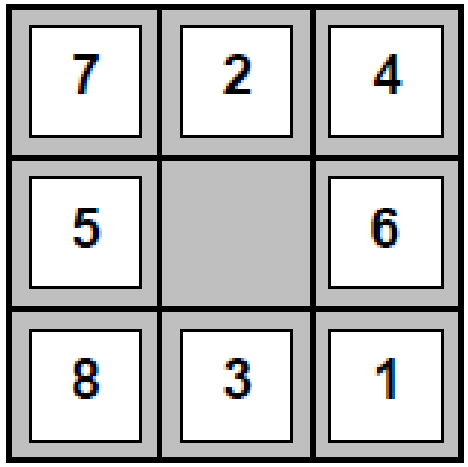
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt
path cost??

Example: vacuum world state space graph

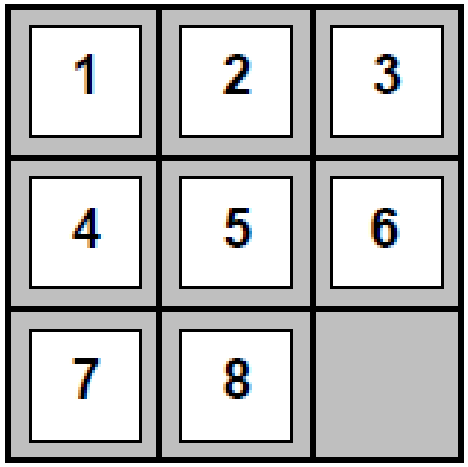


states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt
path cost??: 1 per action (0 for $NoOp$)

Example: The 8-puzzle



Start State



Goal State

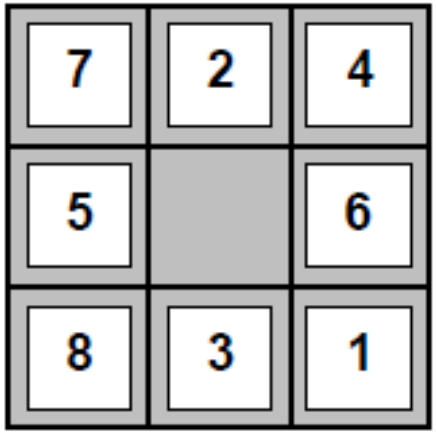
states??

actions??

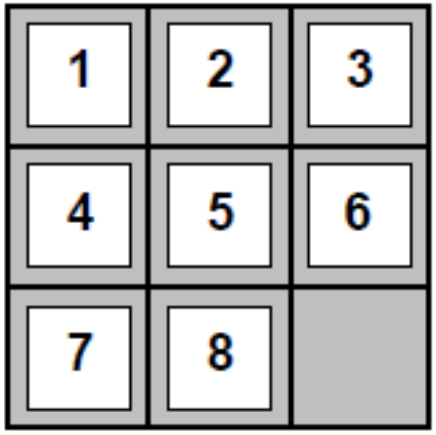
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

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

actions??

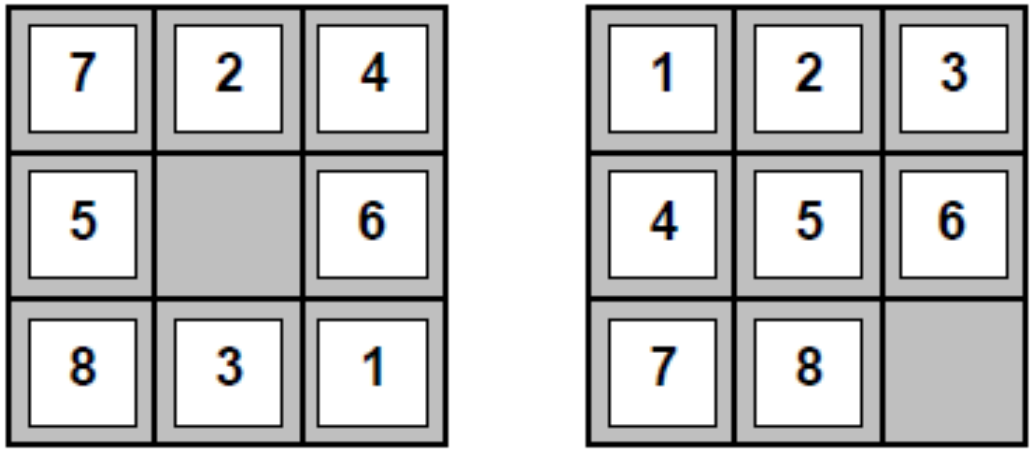
goal test??

path cost??

Example: The 8-puzzle

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

Example: The 8-puzzle

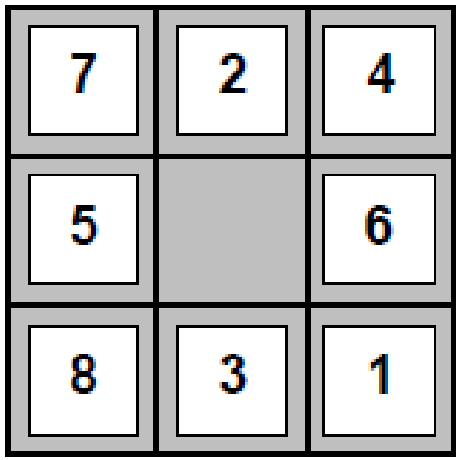


Start State

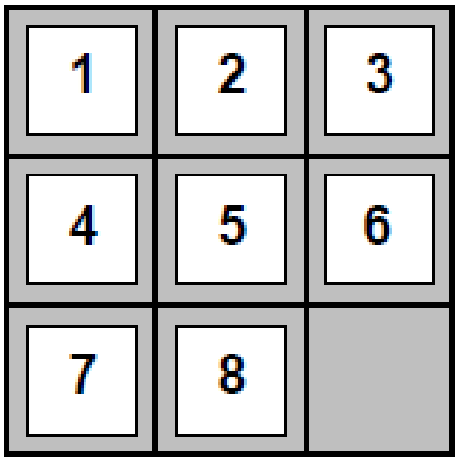
Goal State

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

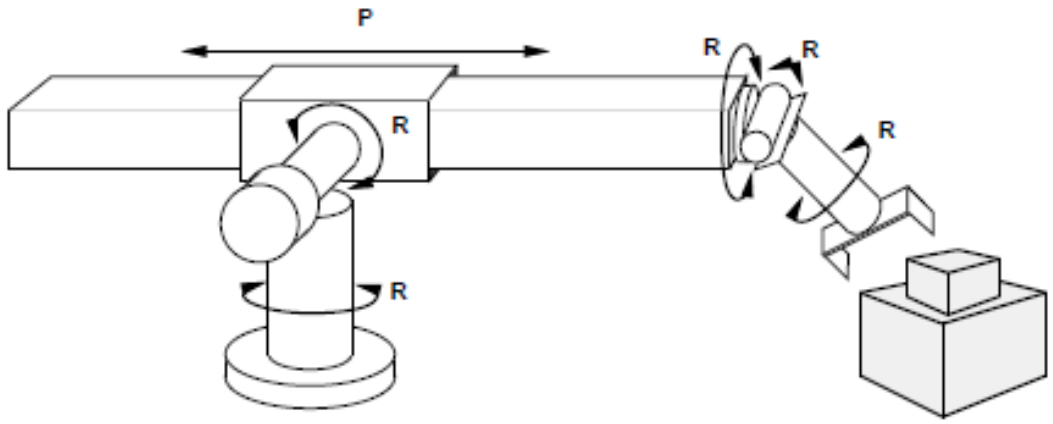


Goal State

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: 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

Search Algorithms

- A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.
- **A search tree** over the **statespace** graph, forms various paths from the initial state, trying to find a path that reaches a goal state.
- Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions.
- *The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another. The search tree describes paths between these states, reaching towards the goal.*

Search Tree

- We can **expand** the node, by considering the available ACTIONS for that state, using the RESULT function to see where those actions lead to, and generating a new node (**called a child node or successor node**) for each of the resulting states.

- Essence of search—following up one option now and putting the others aside for later.. We call this the **frontier** of the search tree.

- Note that the **frontier separates** two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

Tree search algorithms

Basic idea:

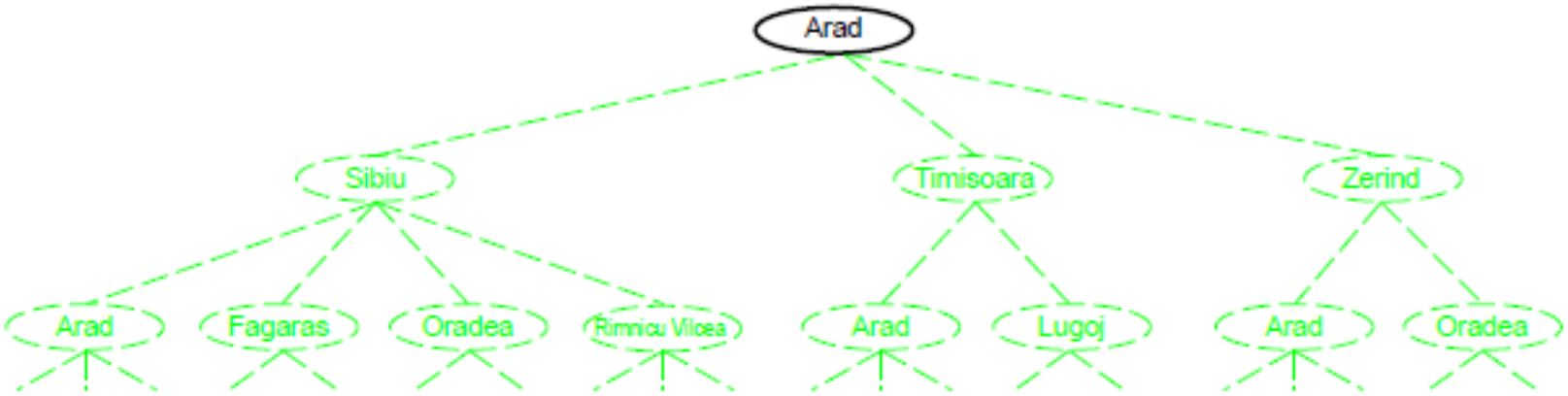
offline, simulated exploration of state space

by generating successors of already-explored states

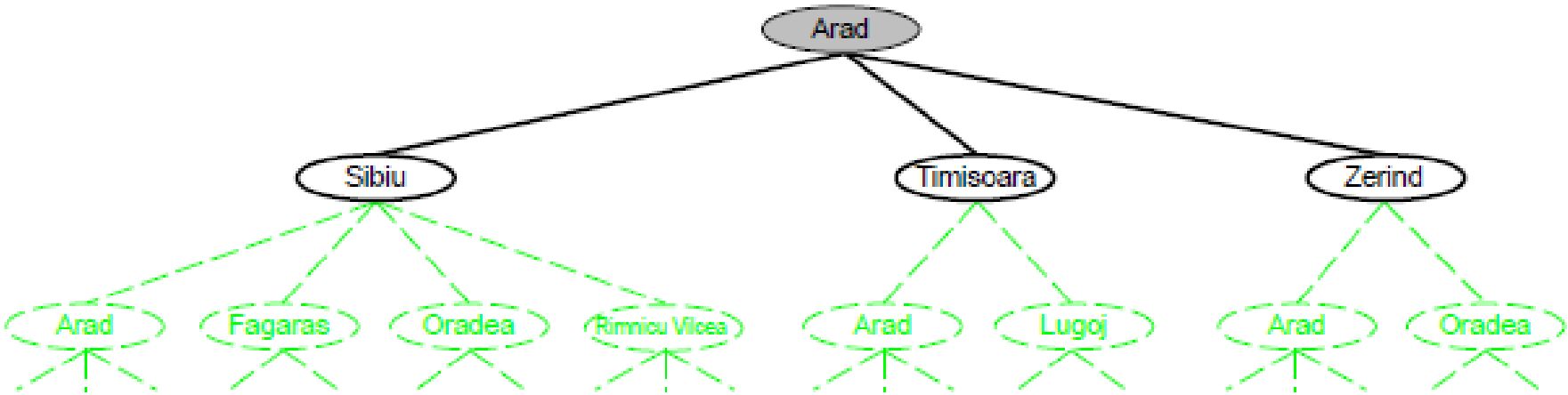
(a.k.a. expanding states)

```
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
```

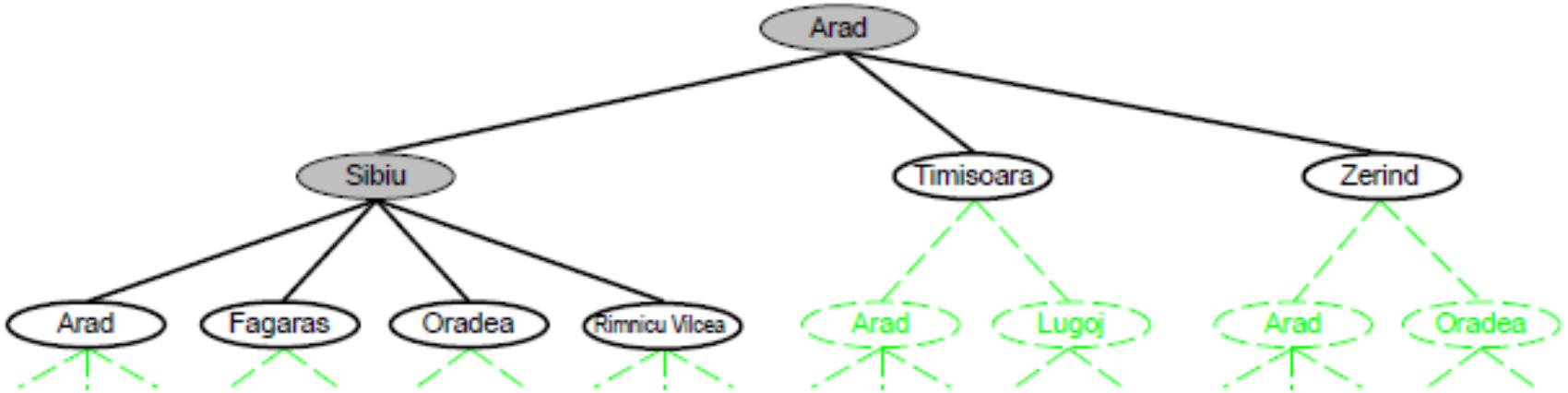
Tree search example



Tree search example



Tree search example



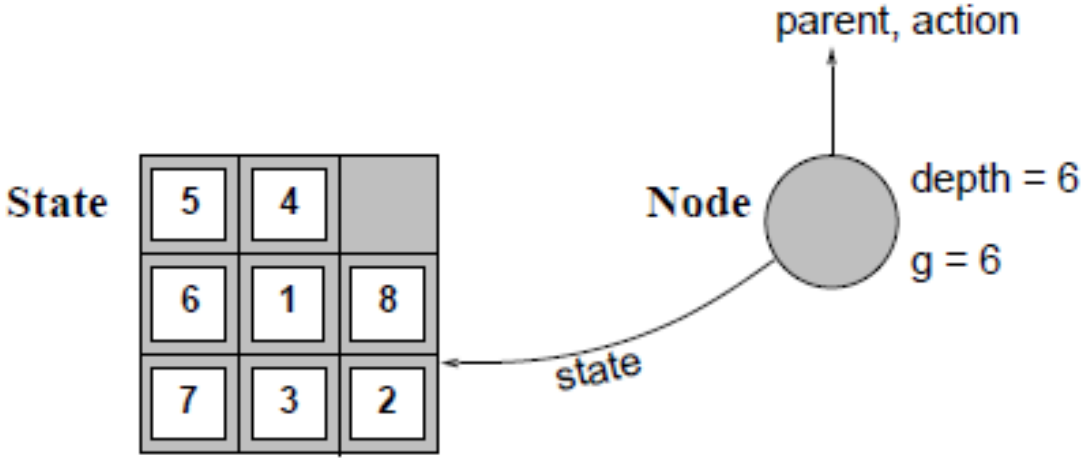
Implementation: states vs. nodes

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, STATE(node)) then 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

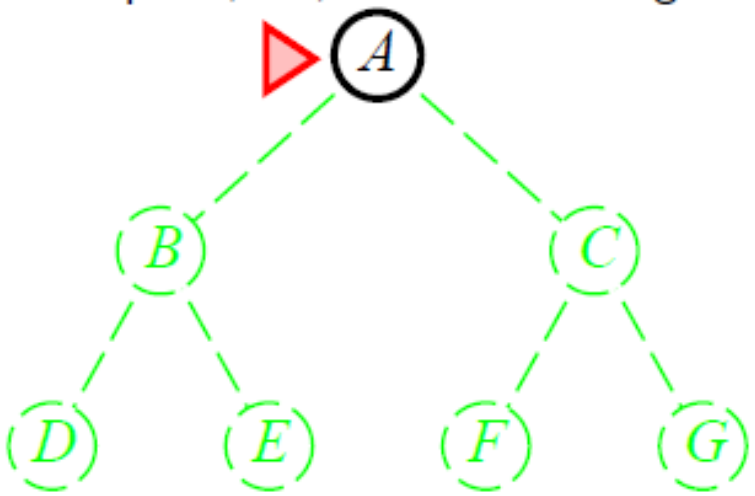
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

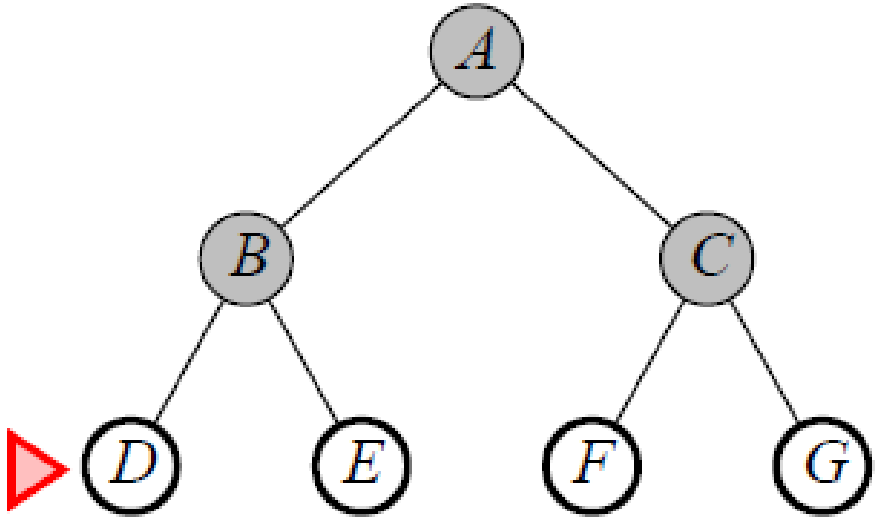


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

- As a typical real-world example, consider a problem with branching factor b = 10, processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node.

- A search to depth d = 10 would take ~ 3 hours, but would require ~ 11 terabytes of memory. The memory requirements are a bigger problem for breadth-first search than the execution time. But time is still an important factor.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

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

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 $g(n)$

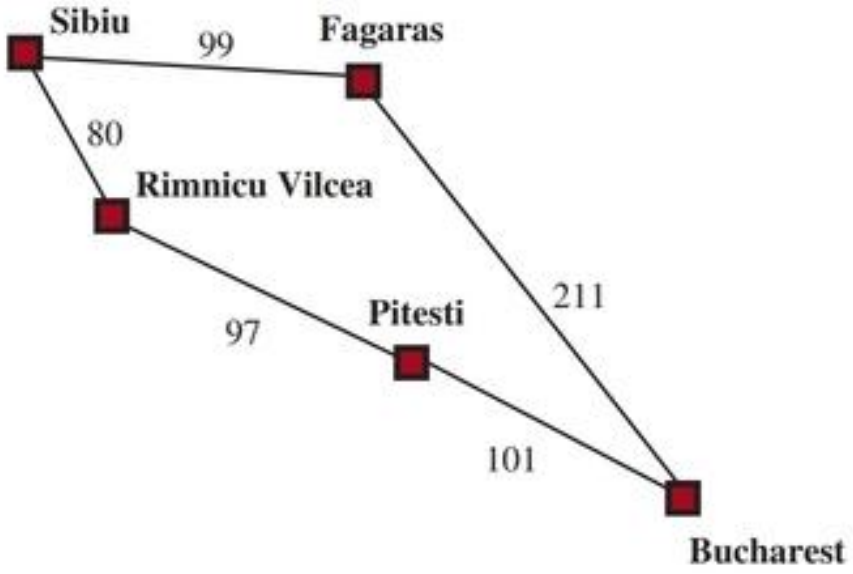
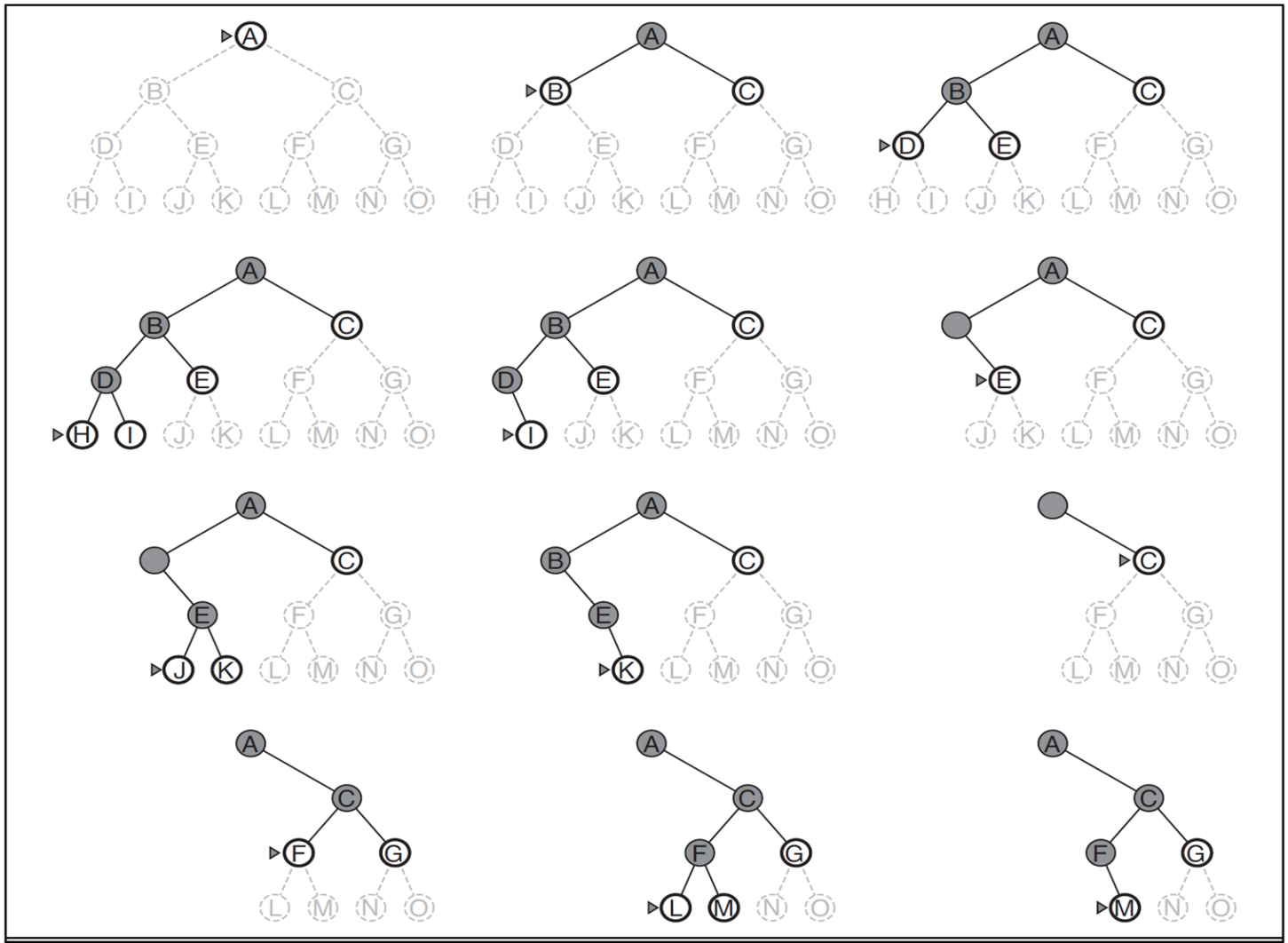


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

Depth-first search

- Depth-first search always expands the deepest node.
- It uses a LIFO (also known as stack), that means the most recently generated node is chosen for expansion.



Depth-first search

- **Complete – Finite search space but incomplete if there is a loop in tree.**
- **Not Optimal**
- Time complexity $O(b^m)$
- Space Complexity $O(bm)$
- Here, *m* is the maximum depth of any node.
- **Depth-first tree search needs to store only a single path from the root to a leaf node.**
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

Depth-limited search

- The failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit .
- Time complexity $O(b^{\ell})$
- Space complexiy $O(b\ell)$
- Depth-first search can be viewed as a special case of depth-limited search with $\ell = \infty$

Depth-limited search

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

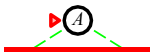
Recursive implementation:

```
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
```

Iterative deepening search

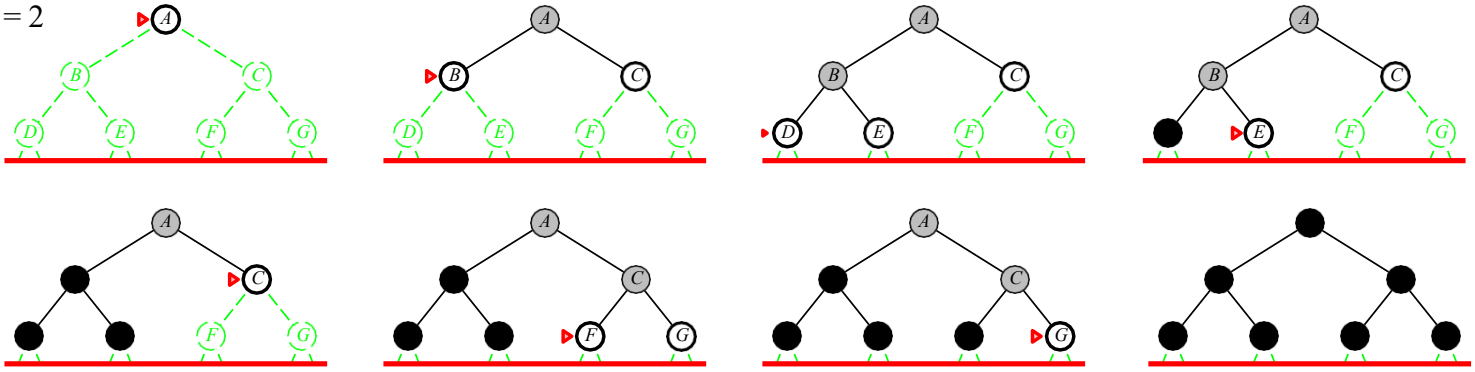
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result /= cutoff then return result
    end
```

Limit = 0

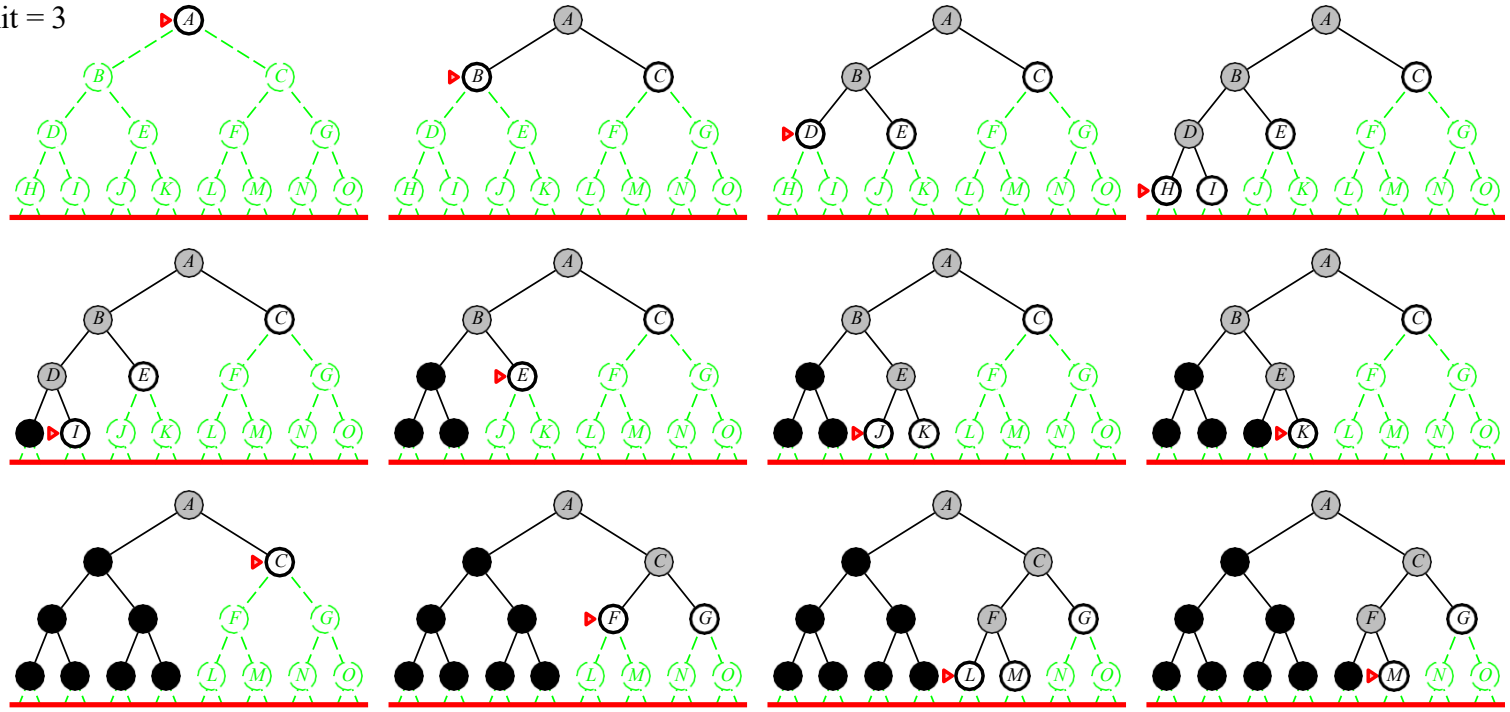


Limit = 1

Limit = 2



Limit = 3



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?? $O(bd)$

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

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

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

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

IDS does better because other nodes at depth d are not expanded BFS can be

modified to apply goal test when a node is generated

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}	$b^{\lfloor C^*/\epsilon \rfloor}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lfloor C^*/\epsilon \rfloor}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

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

- Graph search can be exponentially more efficient than tree search