Solving problems by searching

Chapter 3

In which we see how an agent can find a sequence of actions that achieves its goals when no single action will do.

Outline

◊ Problem-solving agents

◊ Problem formulation

◊ Basic search algorithms

◊ Informed (heuristic) search strategies

Example: Romania

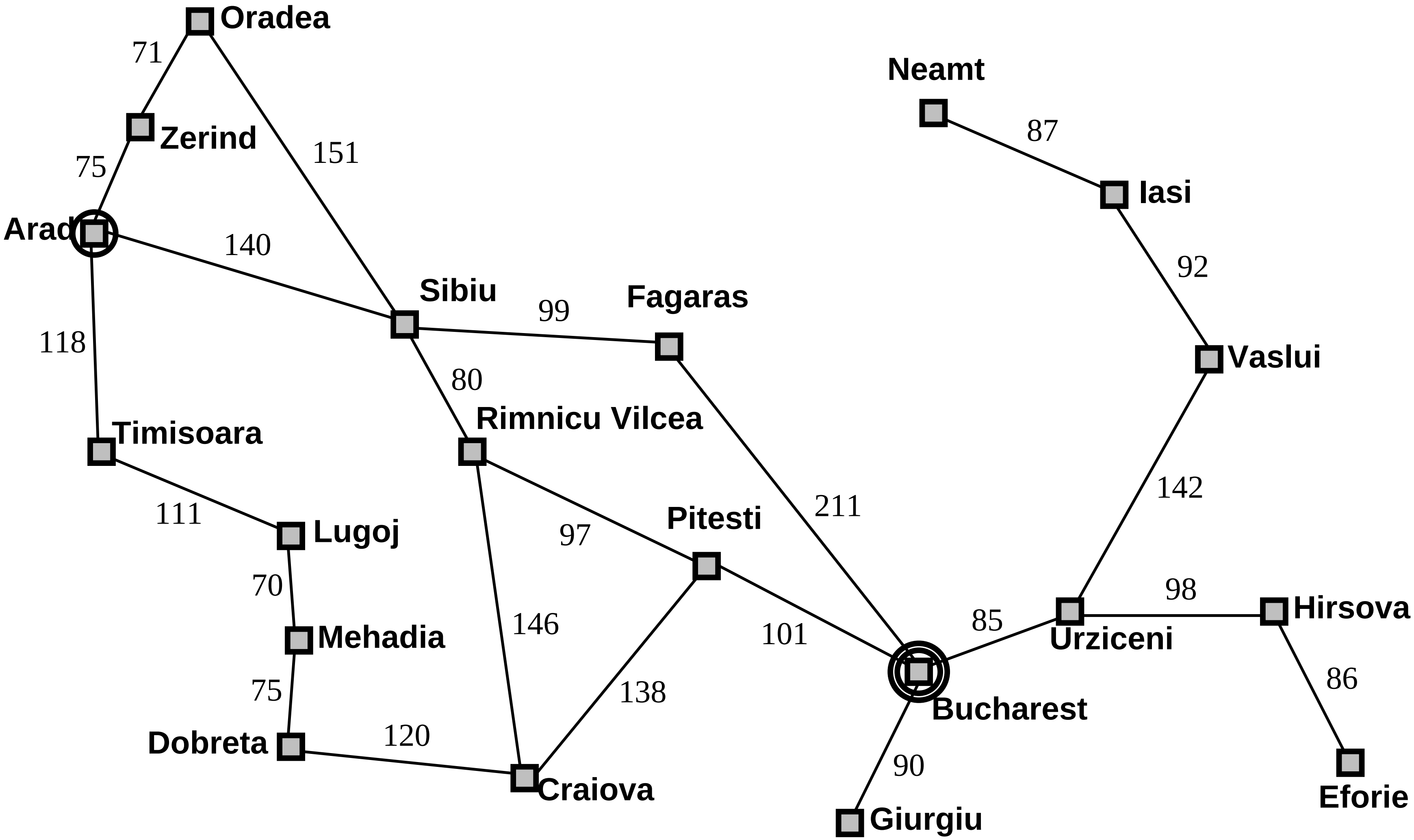
On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:
 be in Bucharest

Formulate problem: (granularity level)
 states: various cities
 actions: drive between cities

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

Example: Romania



Properties of the environment

observable - so the agent always knows the current state.

discrete - so at any given state there are only finitely many actions to choose from.

known, so the agent knows which states are reached by each action.

deterministic - so each action has exactly one outcome

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

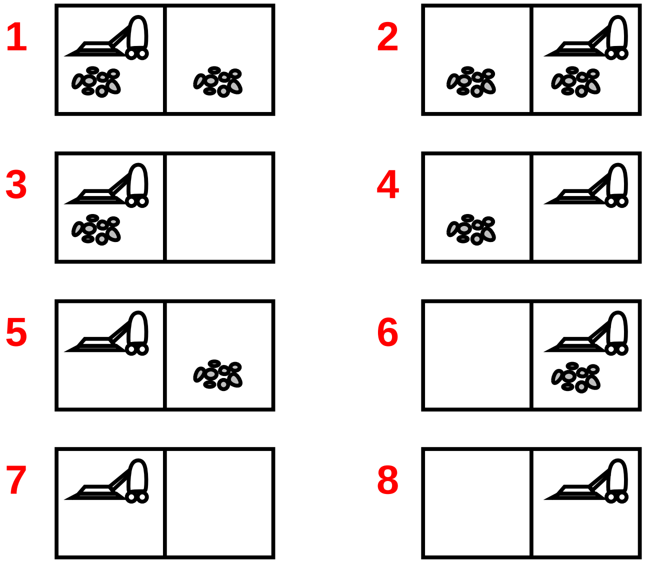
percepts provide **new** information about current state
solution is a contingent plan or a policy
often **interleave** search, execution

The agent might plan to drive from Arad to Sibiu and then to Rimnicu Vilcea but may also need to have a contingency plan in case it arrives by accident in Zerind instead of Sibiu.

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

Example: vacuum world

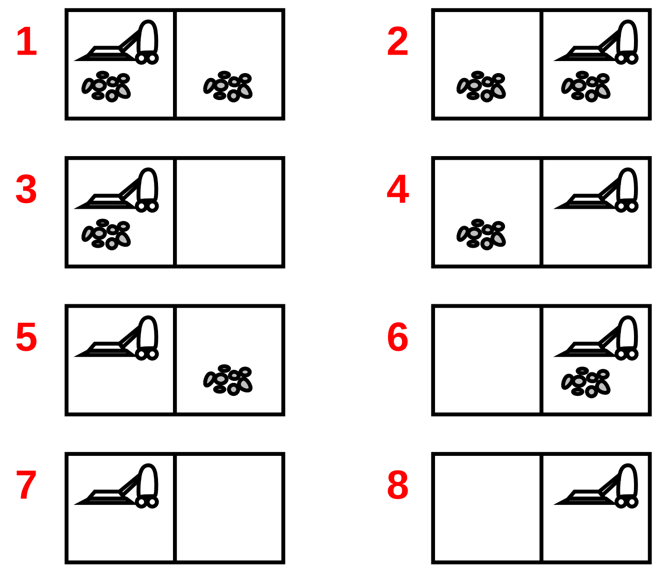
Single-state, start in #5. Solution??
Conformant



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??

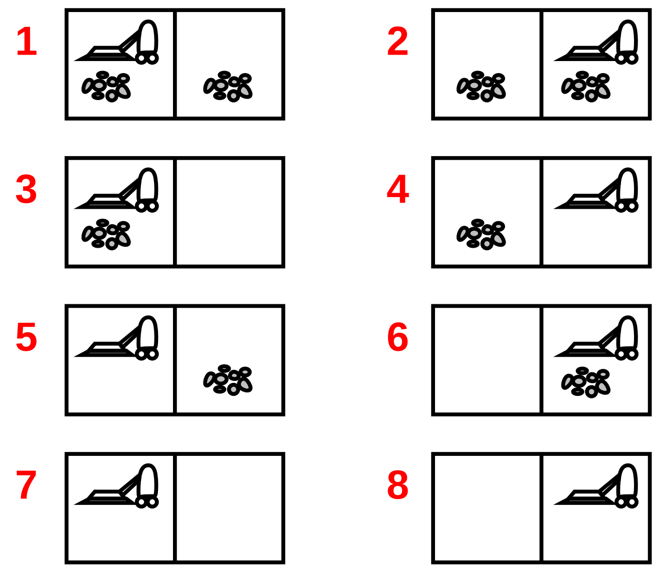


Example: vacuum world

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??

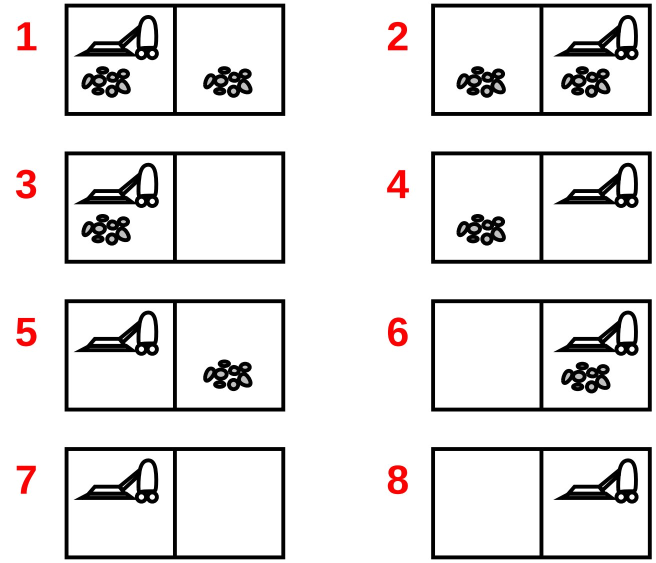


Example: vacuum world

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



Single-state problem formulation

♢ A problem is defined by five items:

1. initial state   e.g., "at Arad"
2. actions: Actions(s) returns applicabile actions in s: (Go(Sibiu), Go(Timisoara), Go(Zerind)
3. transition model - set of action–state pairs (succesor function Result(s,a)): e.g., Result(In(Arad), Go(Zerind))=In(Zerind)
4. goal test - determines if whether a given state is a goal state
   explicit, e.g., $x s =$ "at Bucharest"
   implicit, e.g., $x s =$ checkmate
5. path cost (additive) - reflects agent's own performance measure
   e.g., sum of distances, number of actions executed, etc.
   $c(s, a, s')$ 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

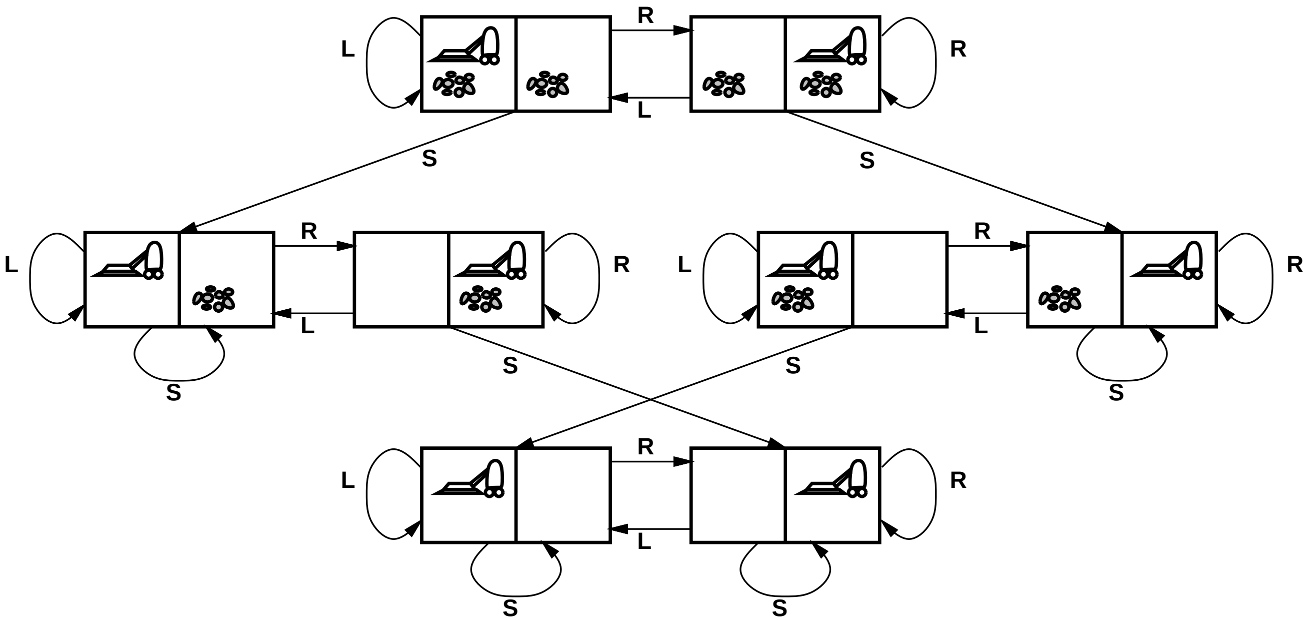
(Abstract) state = set of real states

(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



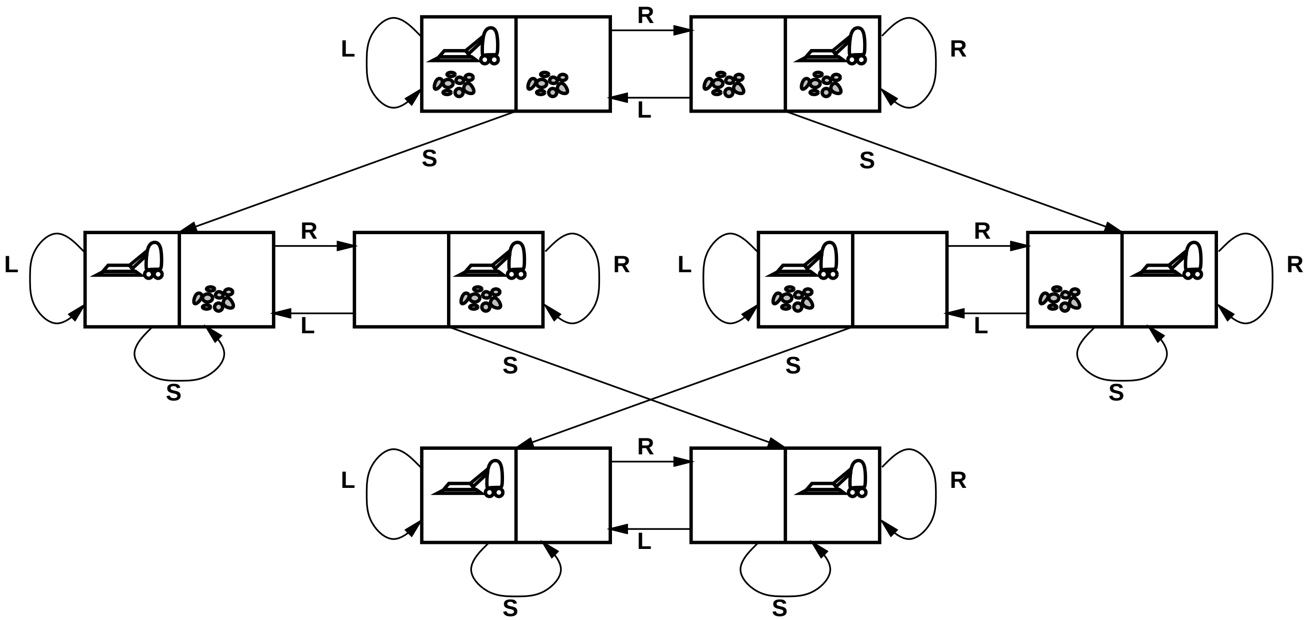
states??

actions??

goal test??

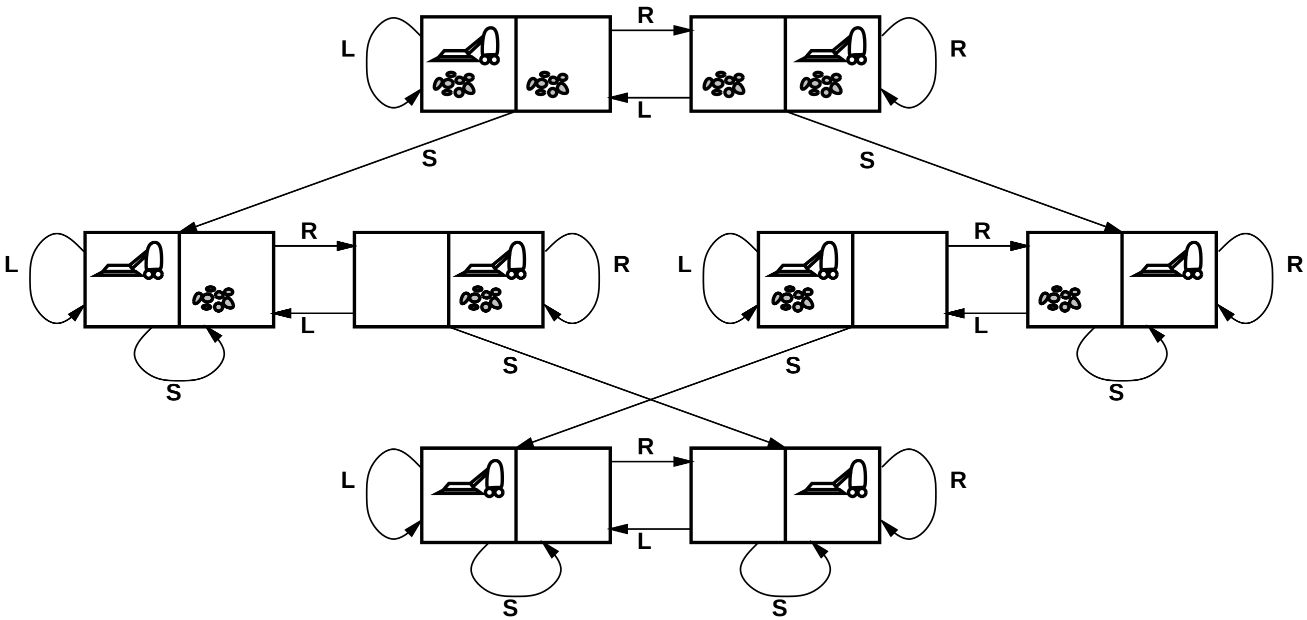
path cost??

Example: vacuum world state space graph



states??: 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.)

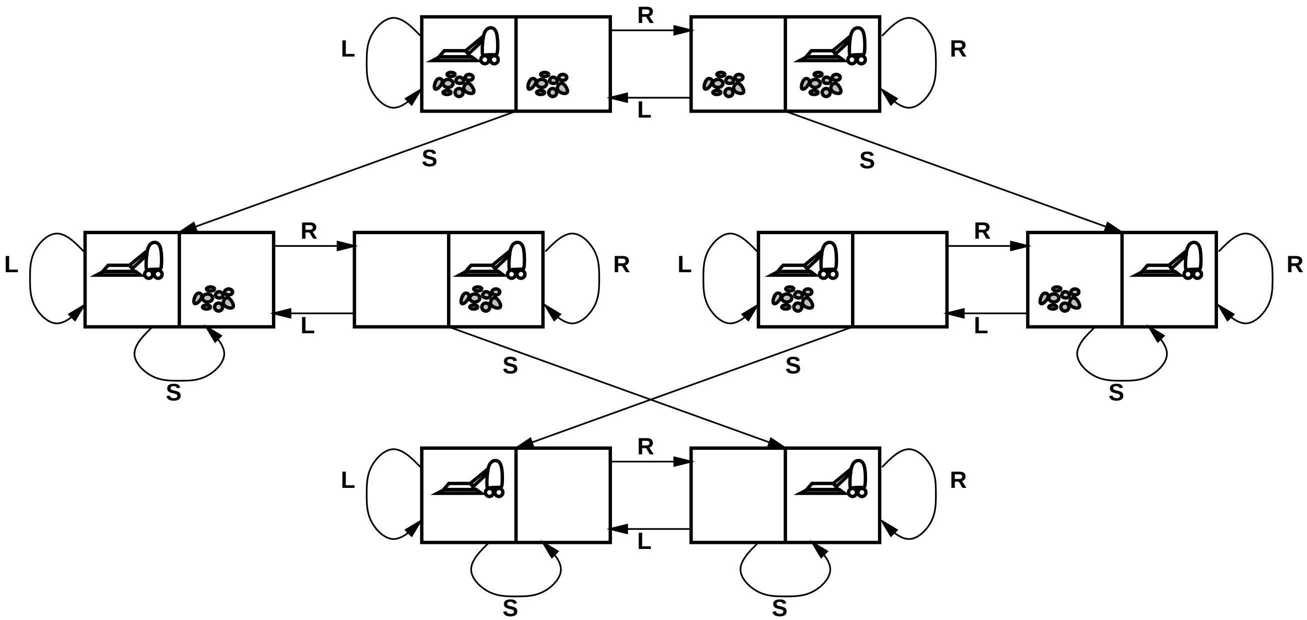
actions??: $Left$, $Right$, $Suck$, $NoOp$

transition model??:

goal test??

path cost??

Example: vacuum world state space graph



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

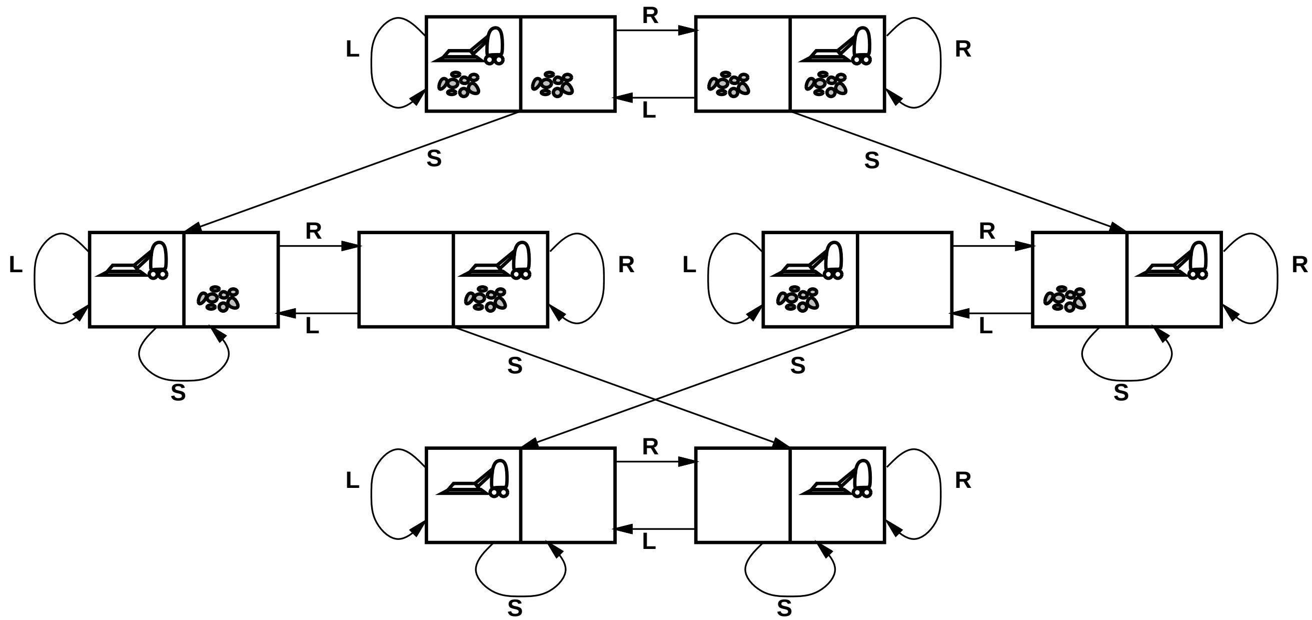
actions??: $Left$, $Right$, $Suck$, $NoOp$

transition model??:

goal test??: no dirt

path cost??

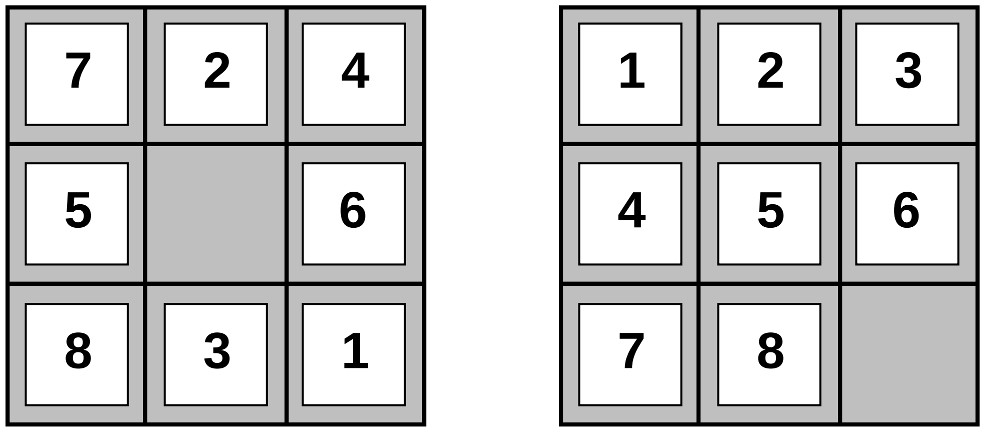
Example: vacuum world state space graph



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

Vacuum world vs. real world:

Example: The 8-puzzle



Start State **Goal State**

states??

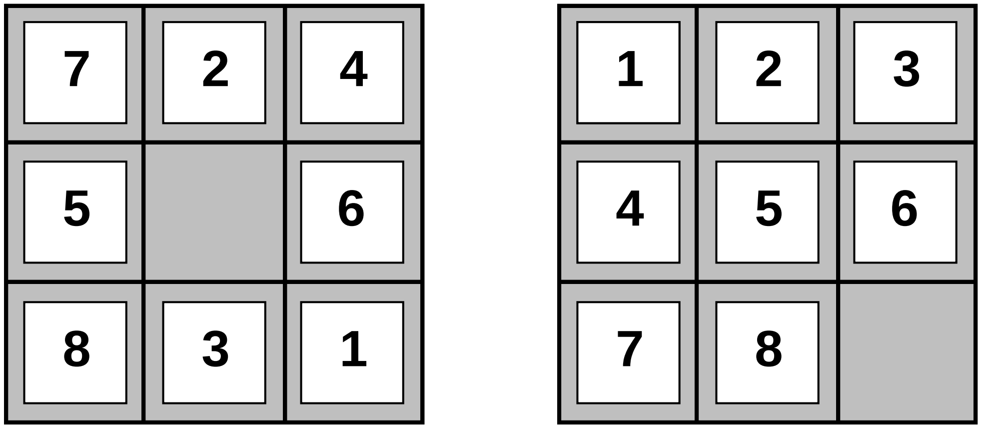
actions??

transition model??

goal test??

path cost??

Example: The 8-puzzle



Start State **Goal State**

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

Example: The 8-puzzle

7	2	4
5		6
8	3	1

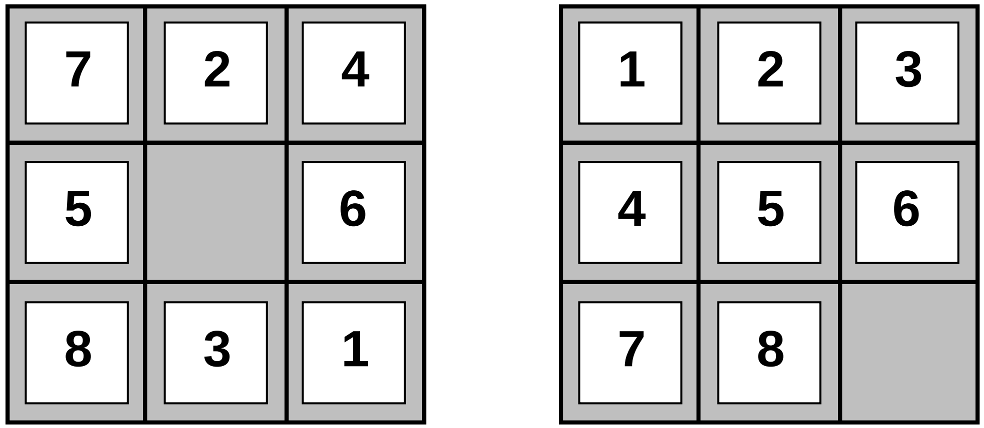
Start State

1	2	3
4	5	6
7	8	

Goal State

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

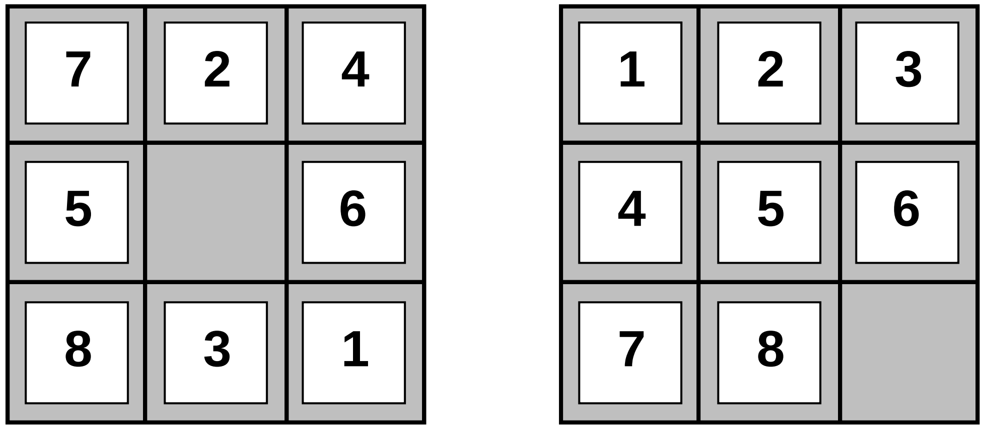
Example: The 8-puzzle



Start State **Goal State**

<u>states</u>??: integer locations of tiles (ignore intermediate positions)
<u>actions</u>??: move blank left, right, up, down (ignore unjamming etc.)
<u>goal test</u>??: $=$ goal state (given)
<u>path cost</u>??

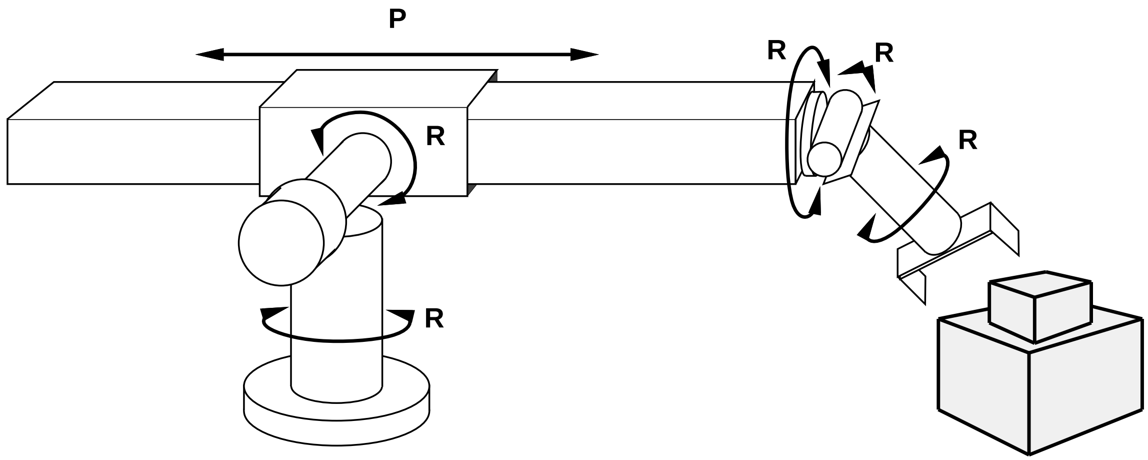
Example: The 8-puzzle



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

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

Outline

◇ Problem-solving agents

◇ Problem formulation

◇ Uninformed search algorithms

◇ Informed (heuristic) search strategies

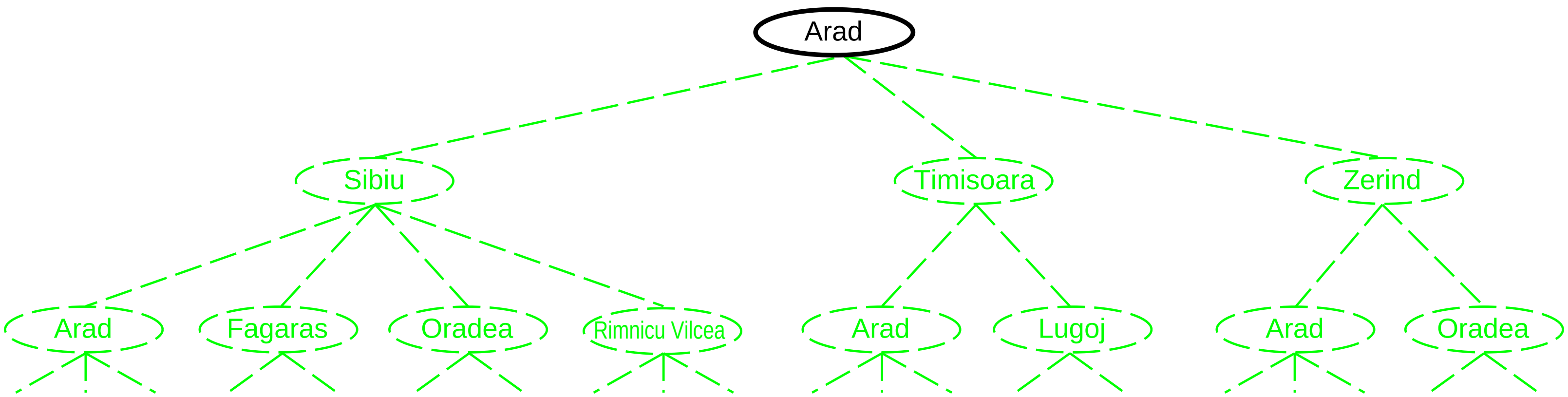
Tree search algorithms

Basic idea:

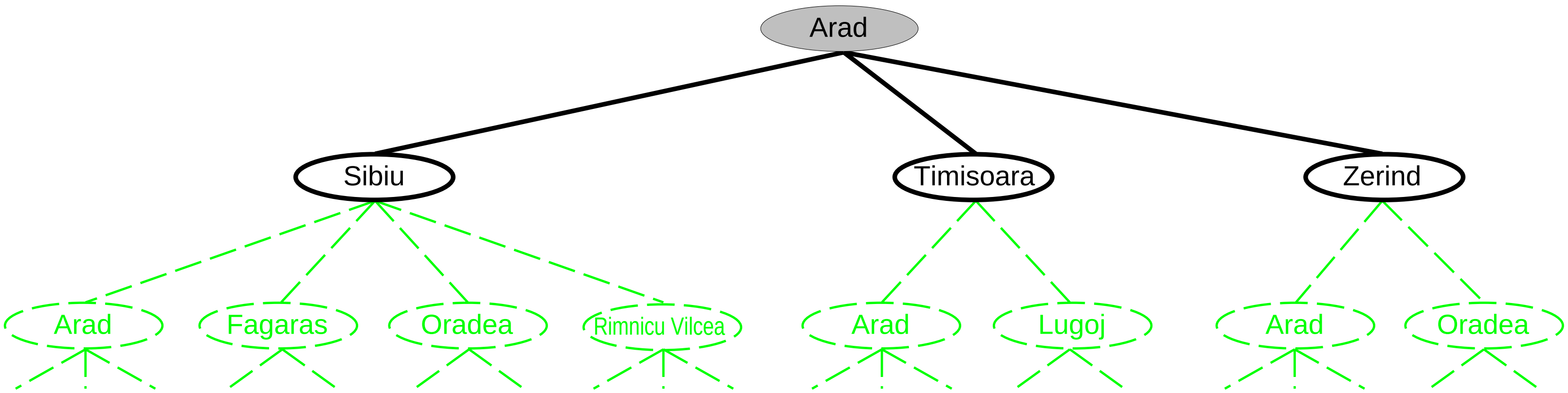
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

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

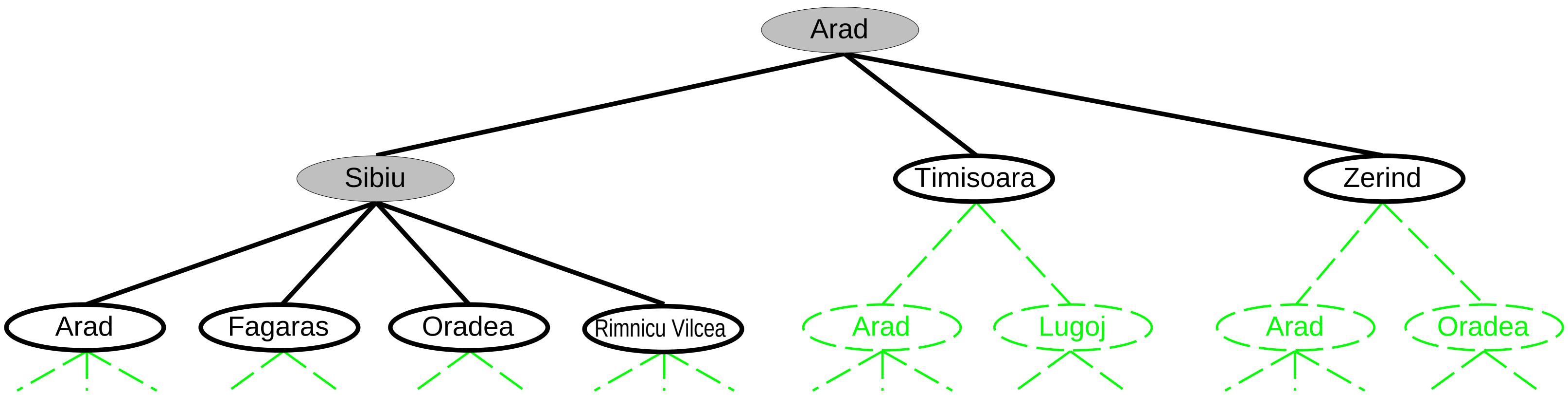
Tree search example



Tree search example

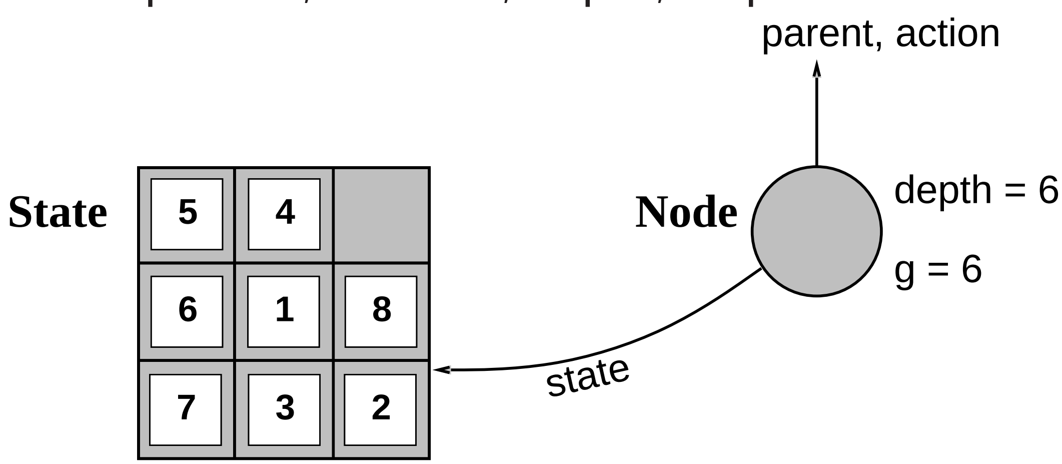


Tree search example



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.

Two different nodes can contain the same world state

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(STATE[node], action,
result)
        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 (root-goal)
m—maximum depth of any path in the state space (may be ∞)

search cost: the amount of time taken by search

solution cost: the total length of the path
total cost $=$ search cost $+$ solution cost

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

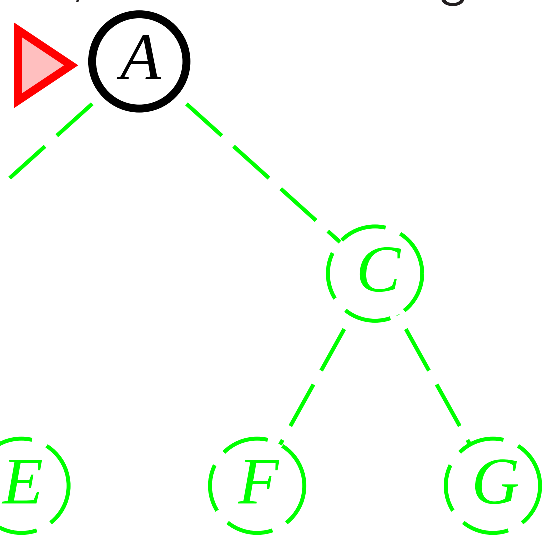
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

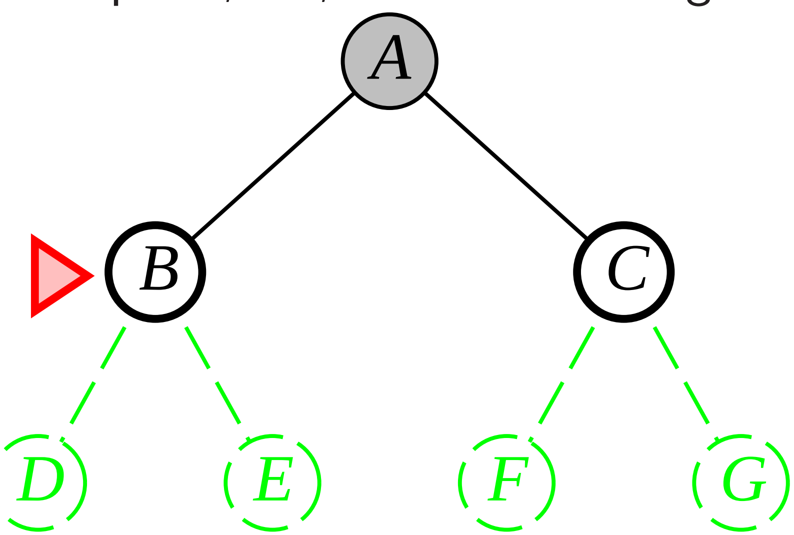


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

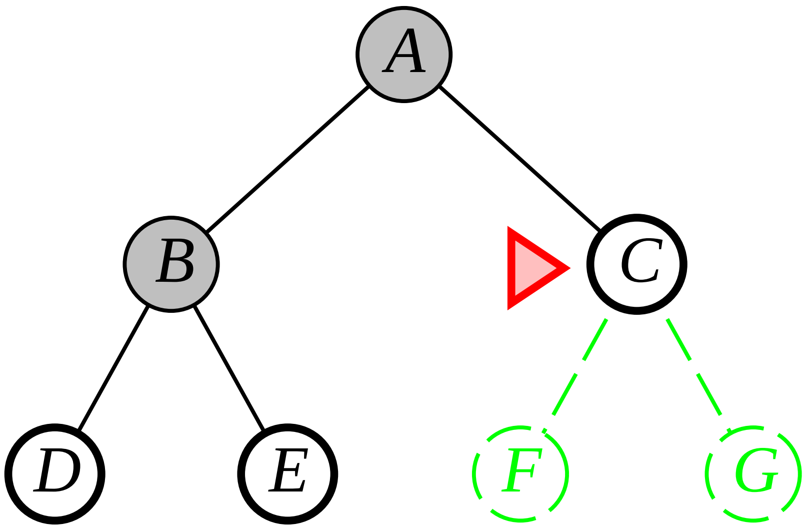


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

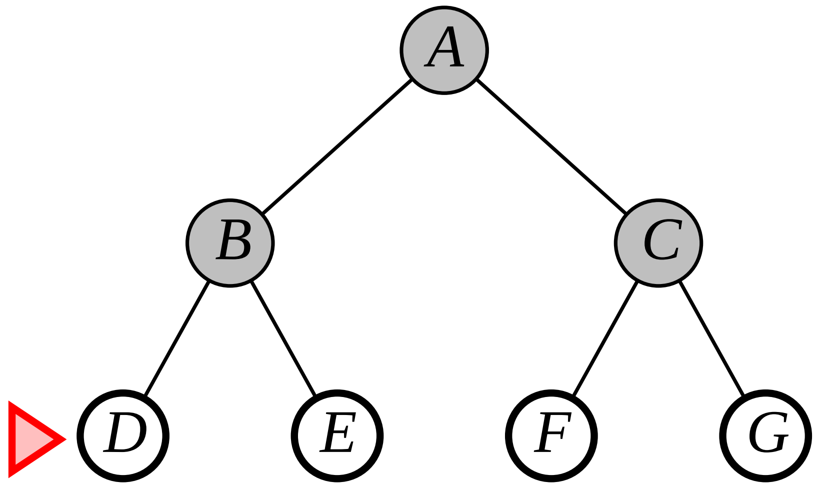


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Breadth-first search on a graph

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
   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)   /* chooses the shallowest node in 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)

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)
```

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 from the frontier $O(b^d)$ and explored set $O(b^{d-1})$ in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

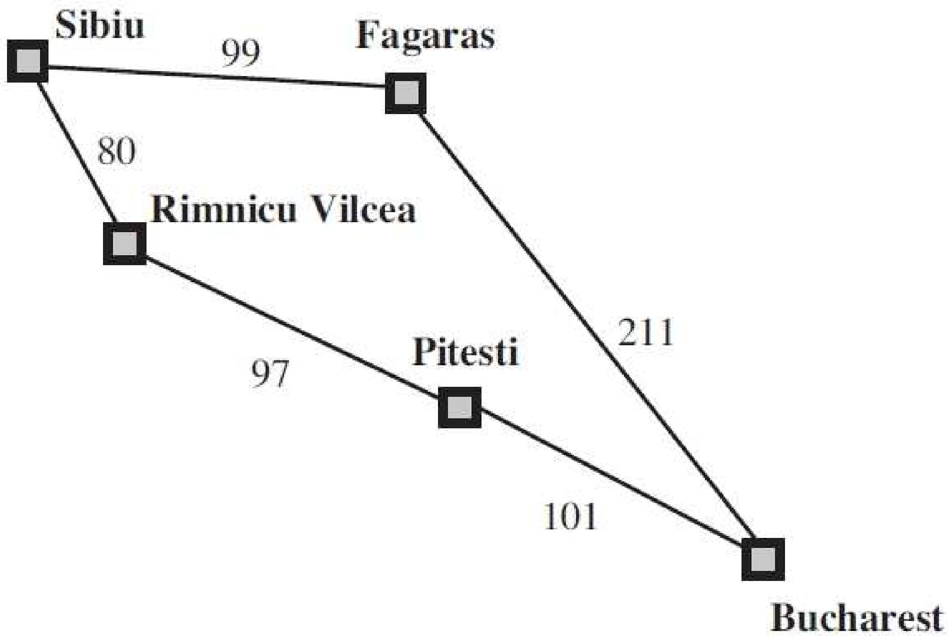
Optimal?? Yes (if cost = 1 per step); not optimal in general

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-cost search

Expand the node with the lowest path cost g(n)



Implementation:

fringe = queue ordered by path cost, lowest first

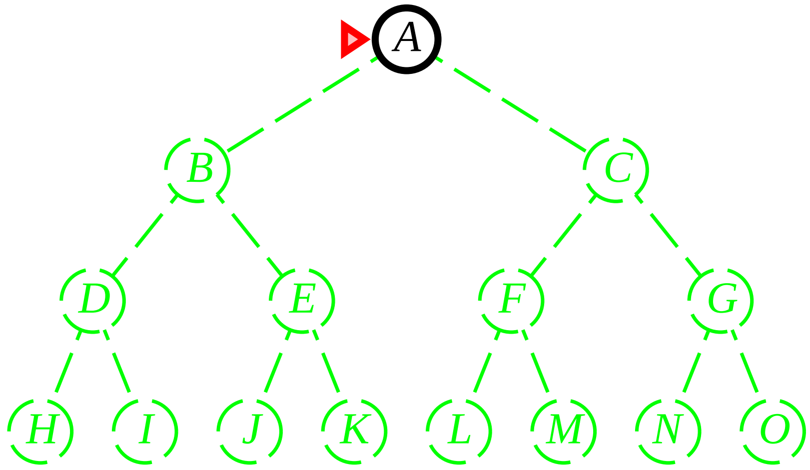
Equivalent to breadth-first if step costs all equal

Depth-first search

Expand deepest unexpanded node

Implementation:

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

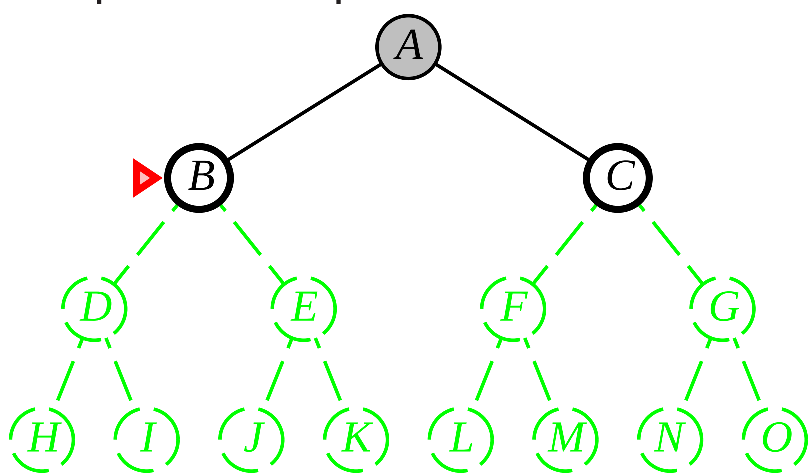


Depth-first search

Expand deepest unexpanded node

Implementation:

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

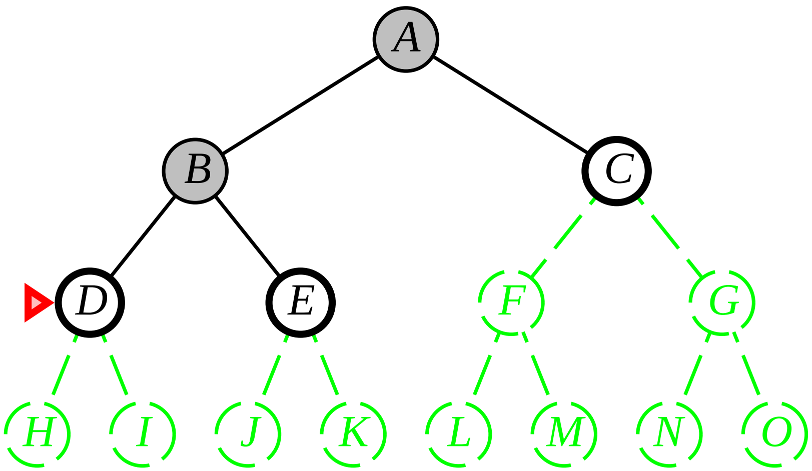


Depth-first search

Expand deepest unexpanded node

Implementation:

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

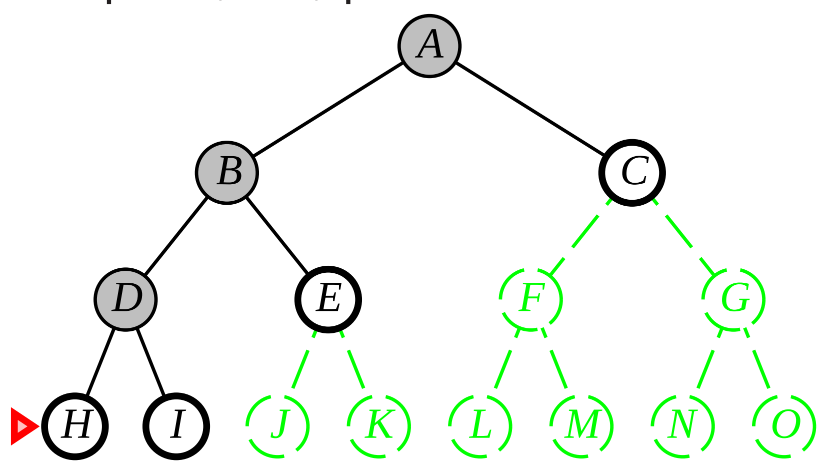


Depth-first search

Expand deepest unexpanded node

Implementation:

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

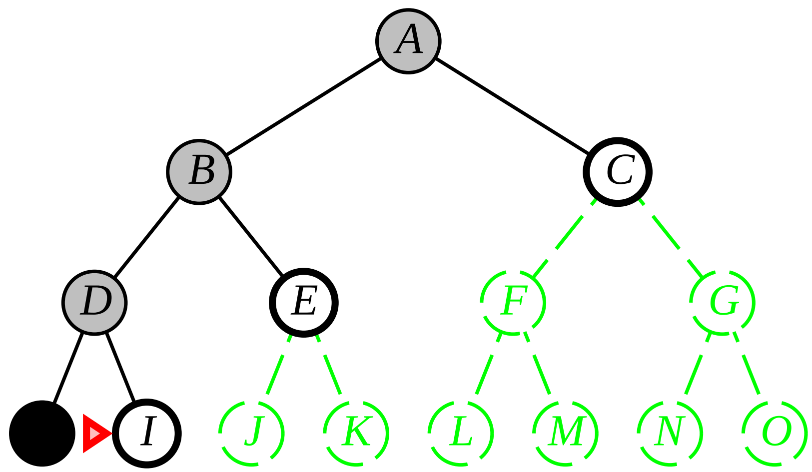


Depth-first search

Expand deepest unexpanded node

Implementation:

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

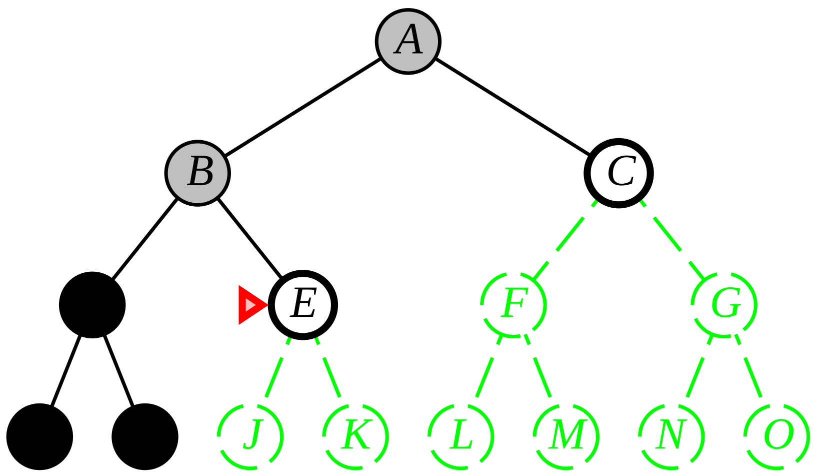


Depth-first search

Expand deepest unexpanded node

Implementation:

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

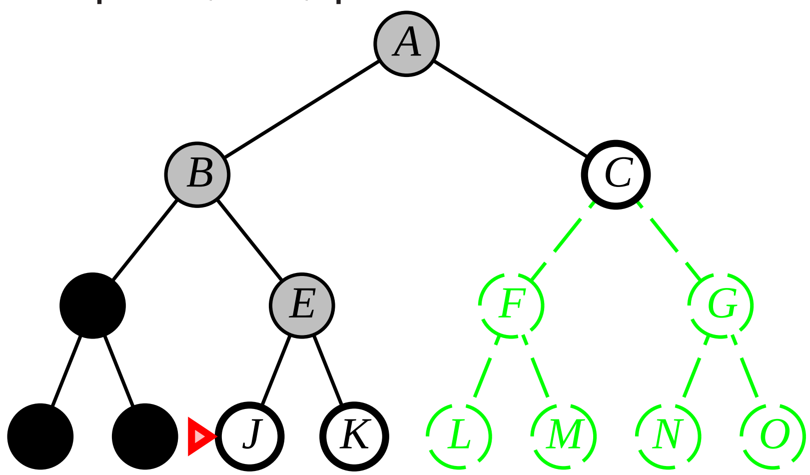


Depth-first search

Expand deepest unexpanded node

Implementation:

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

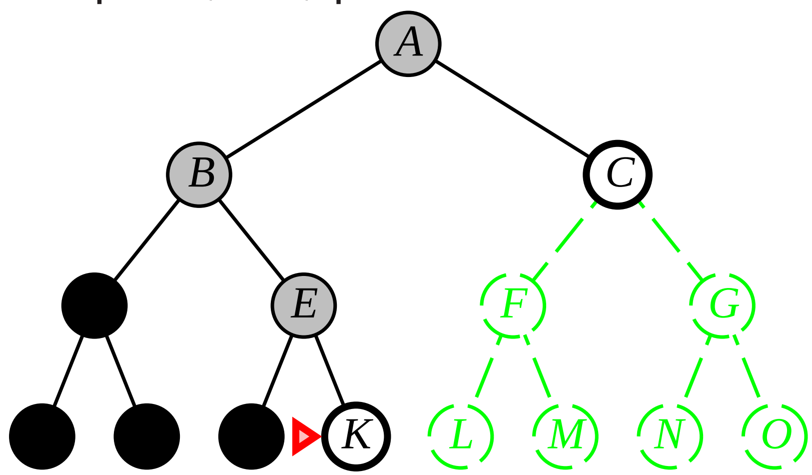


Depth-first search

Expand deepest unexpanded node

Implementation:

$\quad fringe =$ LIFO queue, i.e., put successors at front

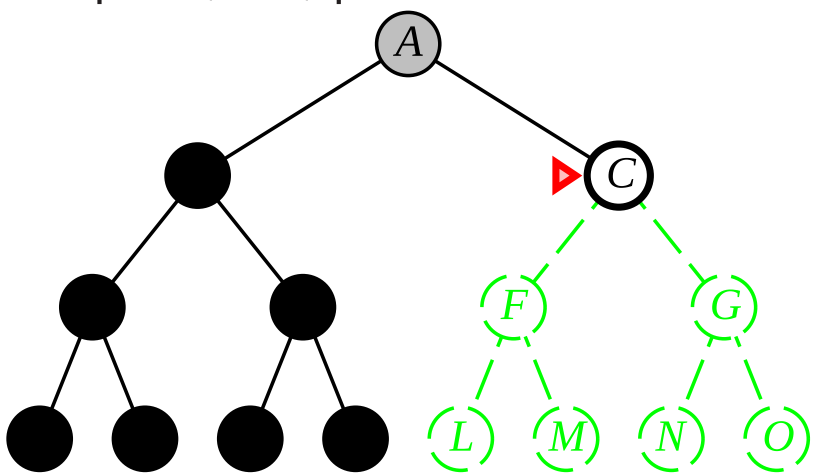


Depth-first search

Expand deepest unexpanded node

Implementation:

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

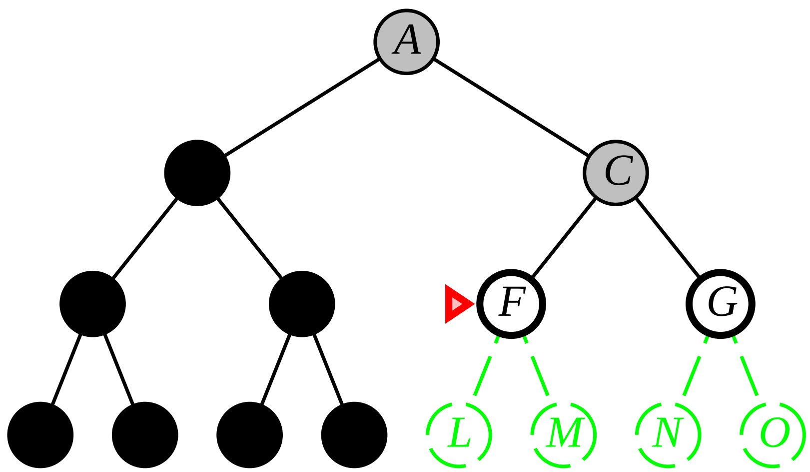


Depth-first search

Expand deepest unexpanded node

Implementation:

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

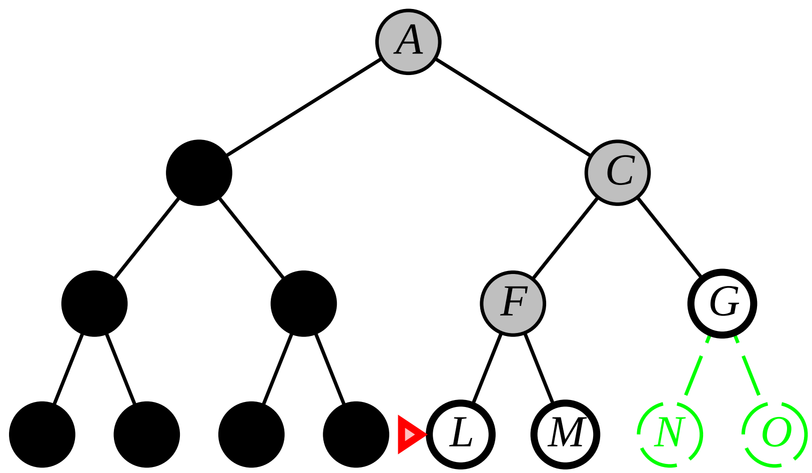


Depth-first search

Expand deepest unexpanded node

Implementation:

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

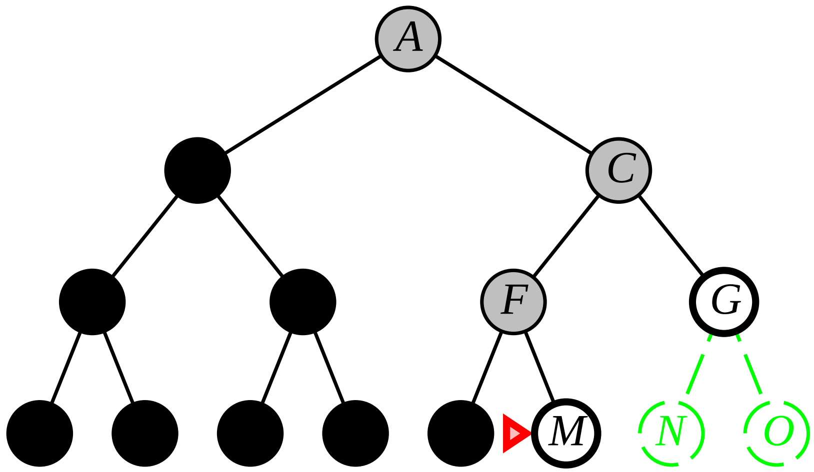


Depth-first search

Expand deepest unexpanded node

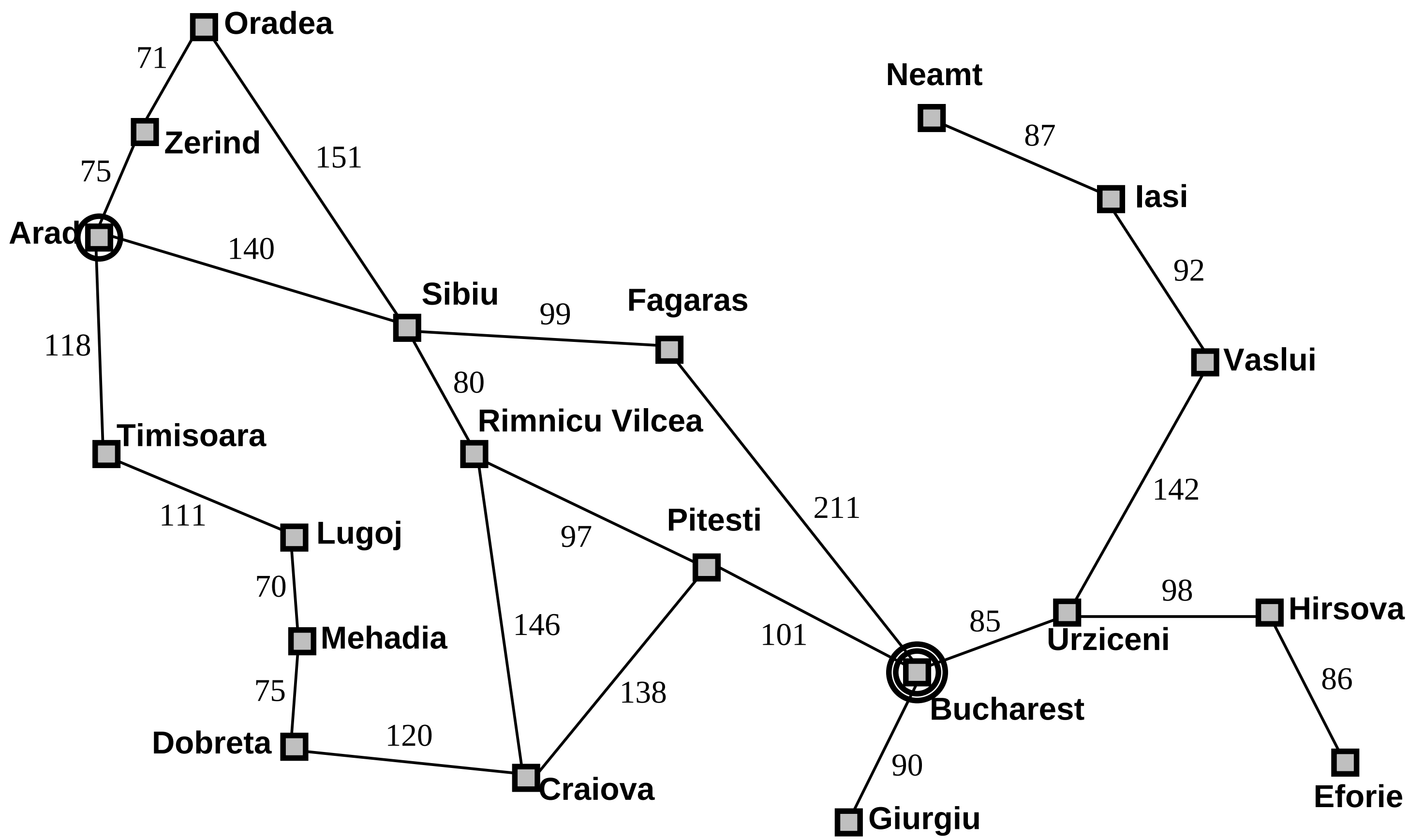
Implementation:

fringe = 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
Arad-Sibiu-Arad-Sibiu....

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??

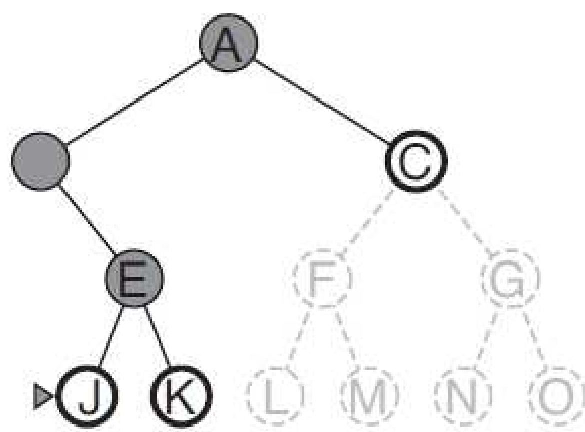
Properties of depth-first search

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!

Optimal??



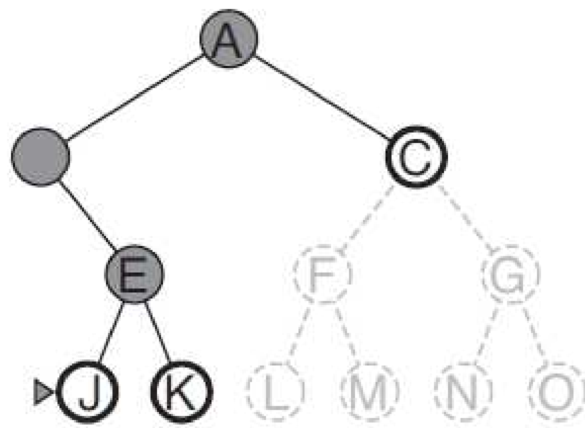
Properties of depth-first search

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 (if J and C are goal nodes, the algorithm returns J)



Depth-limited search

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

a new source of incompleteness ($l < d$) and nonoptimality ($l > d$)

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*

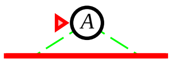
Iterative deepening search

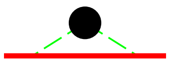
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution
  inputs: *problem*, a problem

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

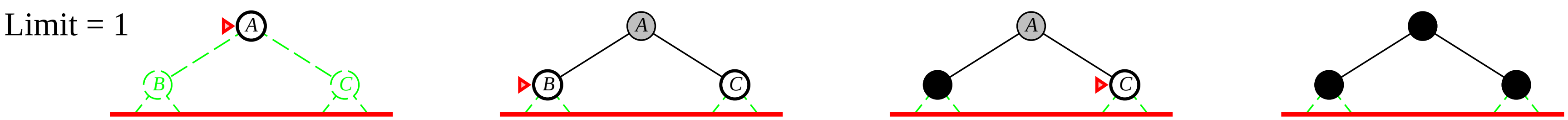
Iterative deepening search $l = 0$

Limit = 0

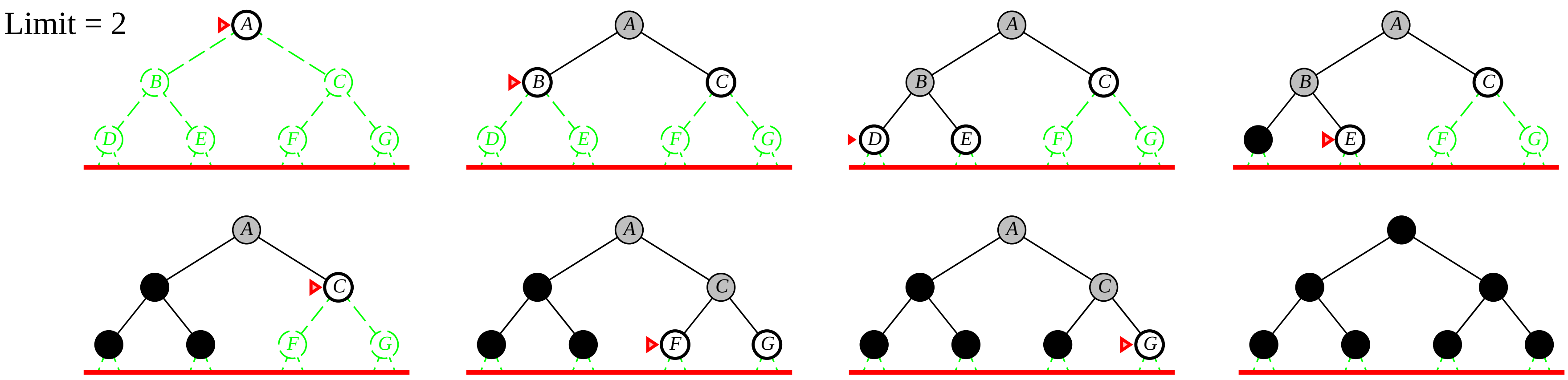




Iterative deepening search $l = 1$

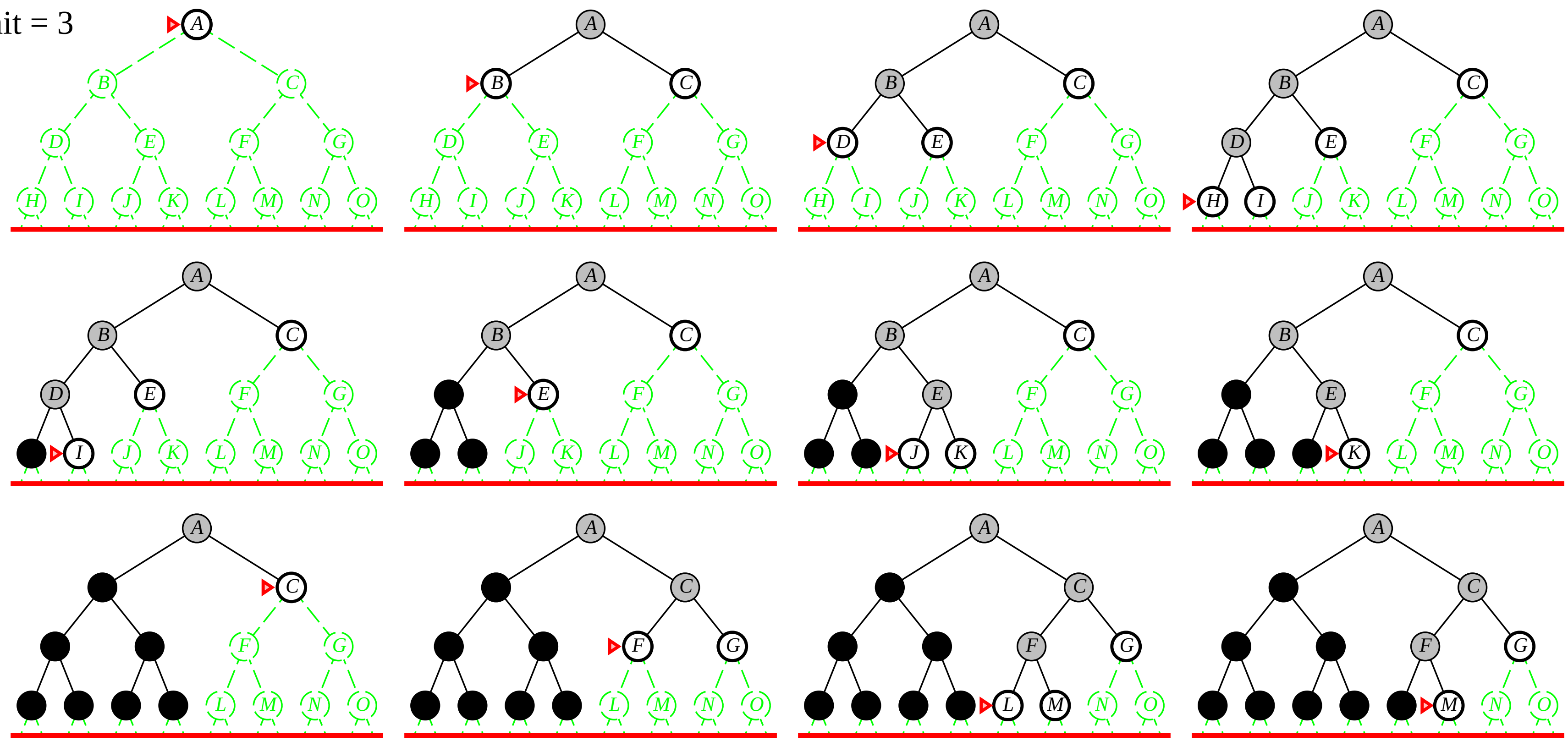


Iterative deepening search $l = 2$



Iterative deepening search $l = 3$

Limit = 3



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? - nodes generated $N(IDS)(d)b^1 + (d-1)b^2 + \ldots + (1)b^d$
recall $N(BFS) = b^1 + b^2 + ... + b^d$)
b=10, d=4

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?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

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

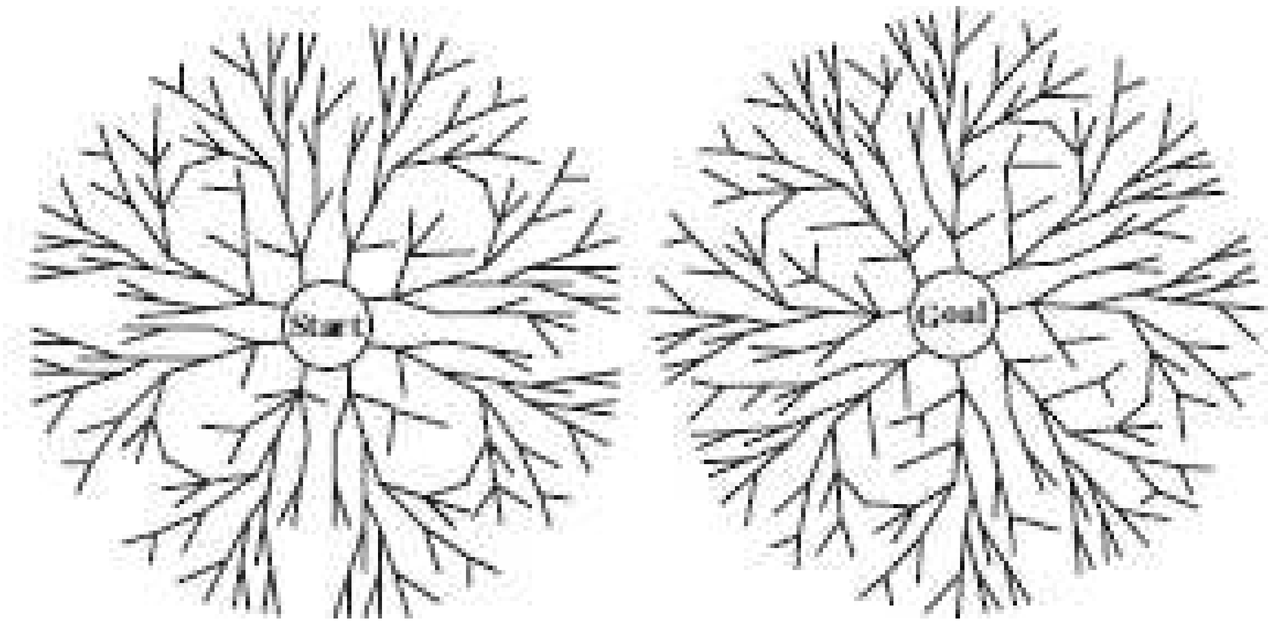
Space?? $O(bd)$

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:

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

Bidirectional search



$b^{d/2} + b^{d/2} < b^d$

Replacing the goal test: check wheather the frontiers intersect

Solution is not optimal

What if the goal is an abstract description: "no queen attacks another queen"?

Outline

◊ Problem-solving agents

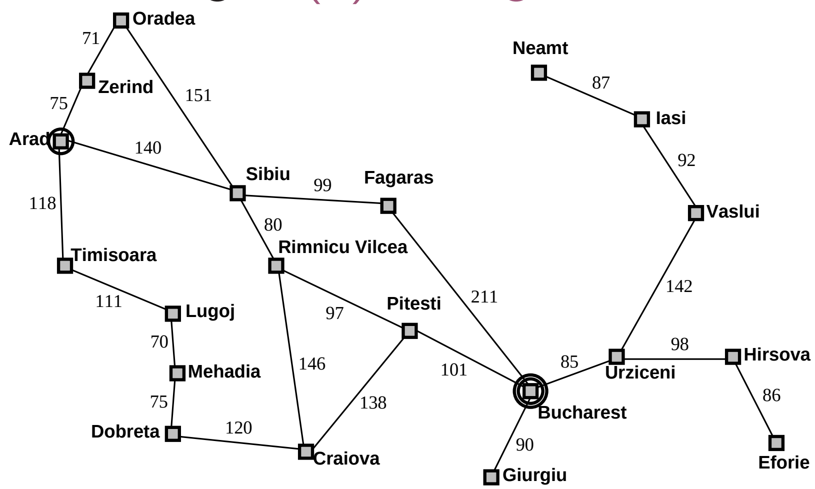
◊ Problem formulation

◊ Basic search algorithms

◊ Informed (heuristic) search strategies

Greedy best first search

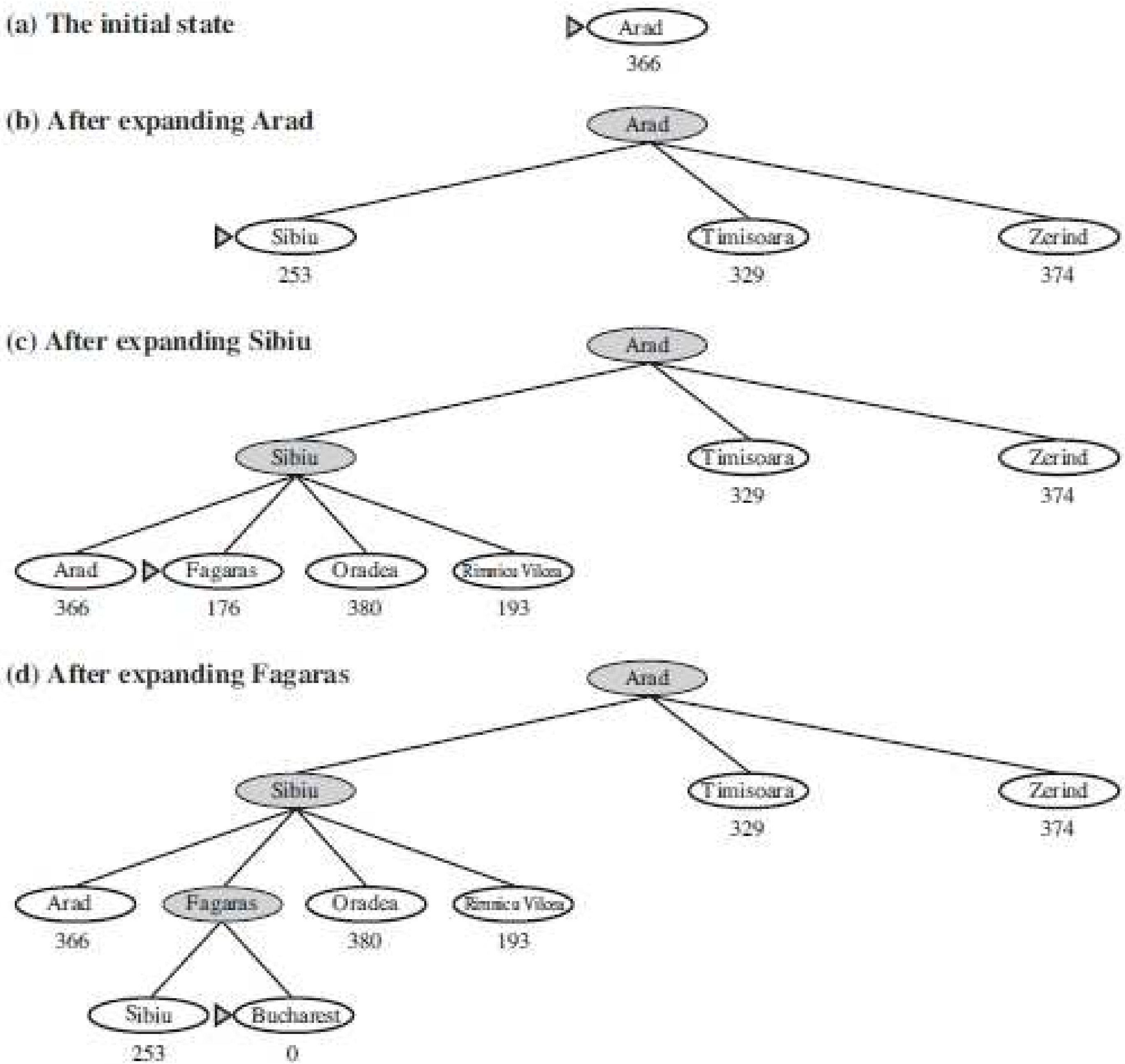
Heuristic function h(n)= estimated cost of the cheapest path from the state at node n to a goal state, e.g., h(n)=straight line distance



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

h_{sld} cannot be computed from the problem description itself

Search cost is minimal: without expanding a node not on the solution path. Is it optimal? Is it complete? (Iasi-Fagaras)

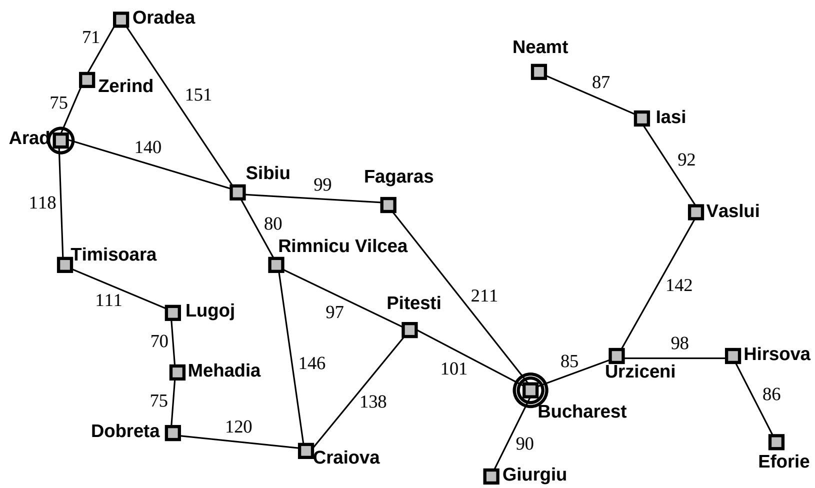

(a) The initial state
Arad
366
(b) After expanding Arad
Arad
Sibiu
253
Timisoara
329
Zerind
374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193
(d) After expanding Fagaras
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

A* search: minimising the total estimated solution cost

g(n) - the cost to reach the solution

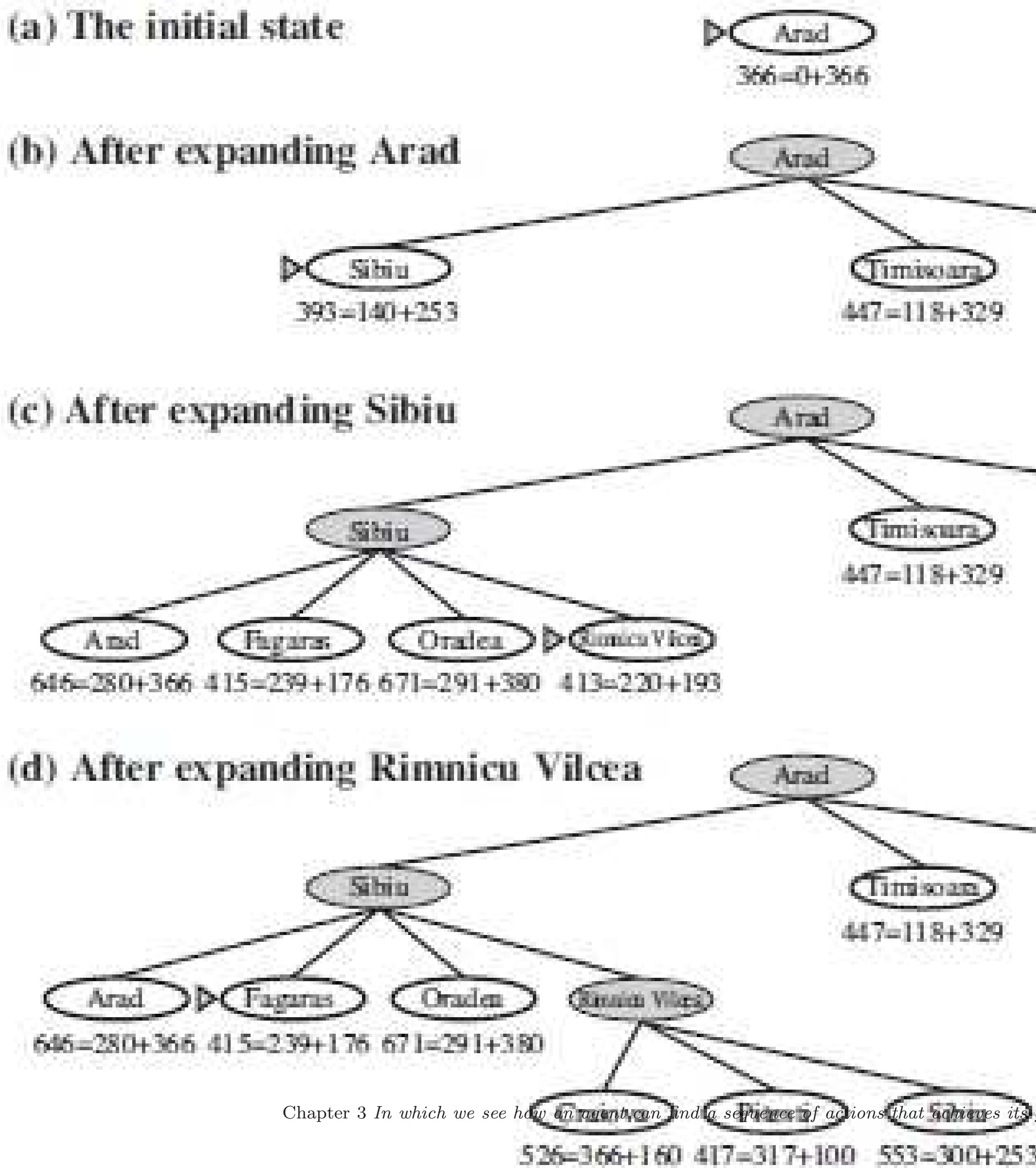
h(n) - the cost to get from the node to the goal

f(n)= g(n)+ h(n)



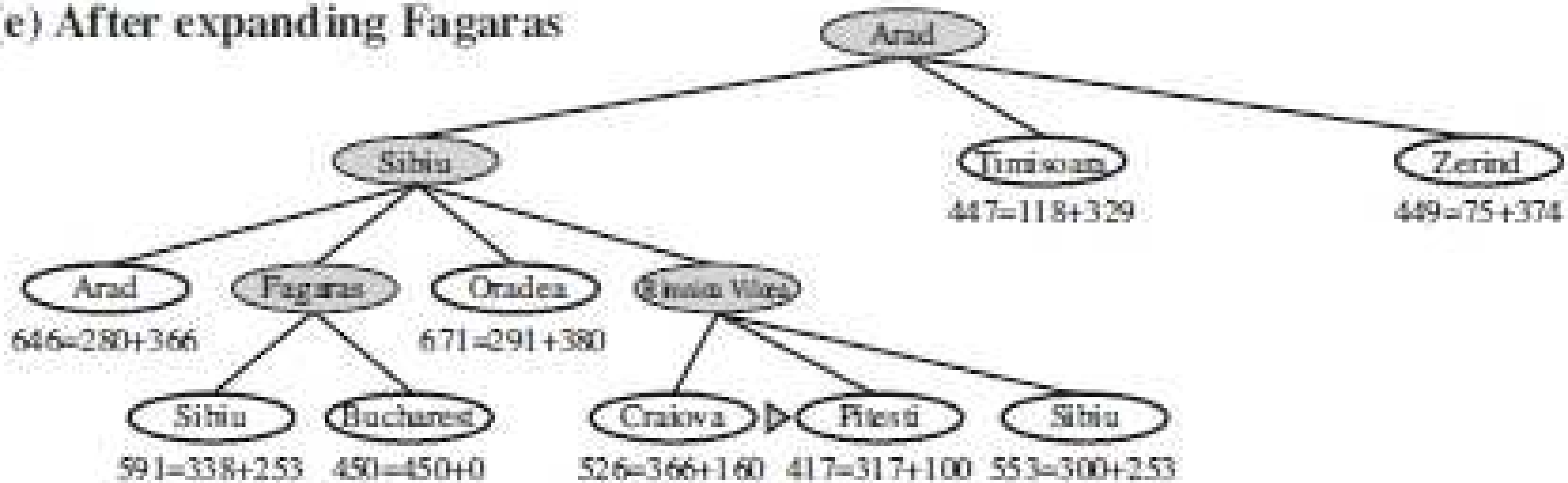
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

A* search: minimising the total estimated solution cost

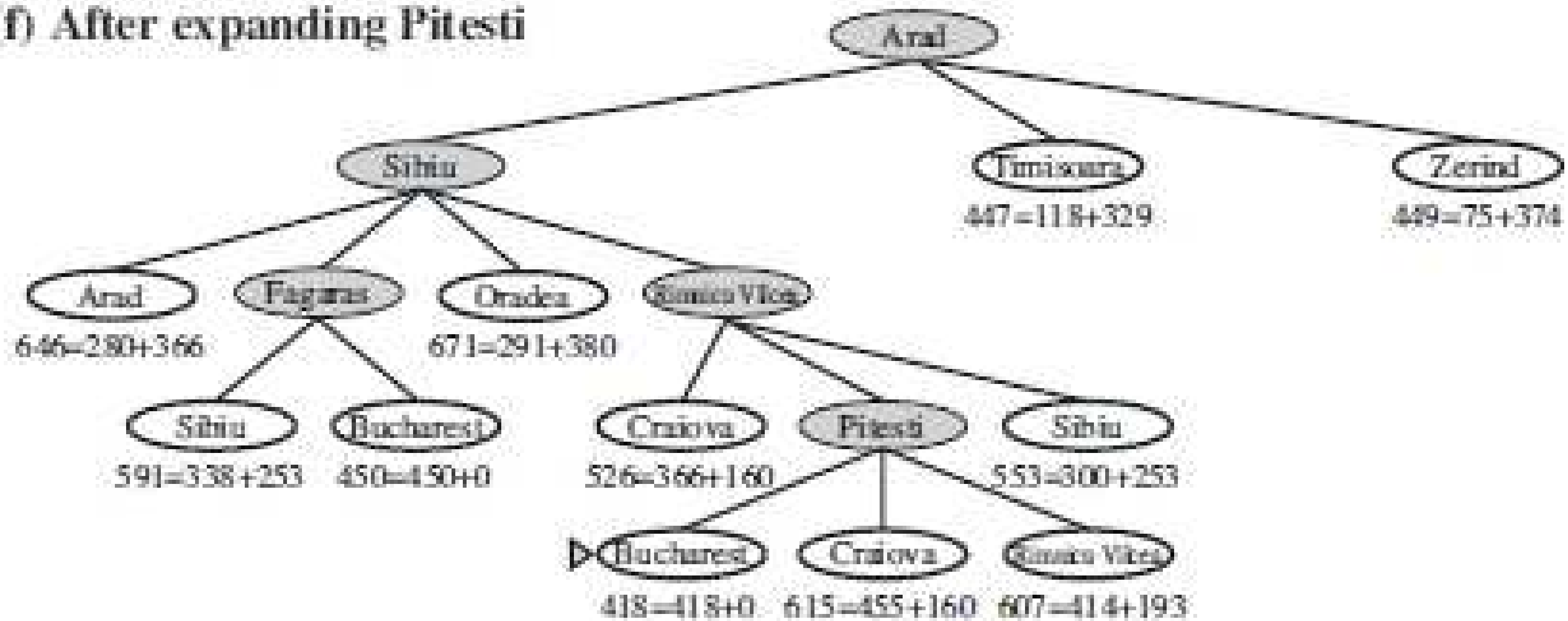

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
526=366+160
417=317+100
553=300+253

A* search: minimising the total estimated solution cost

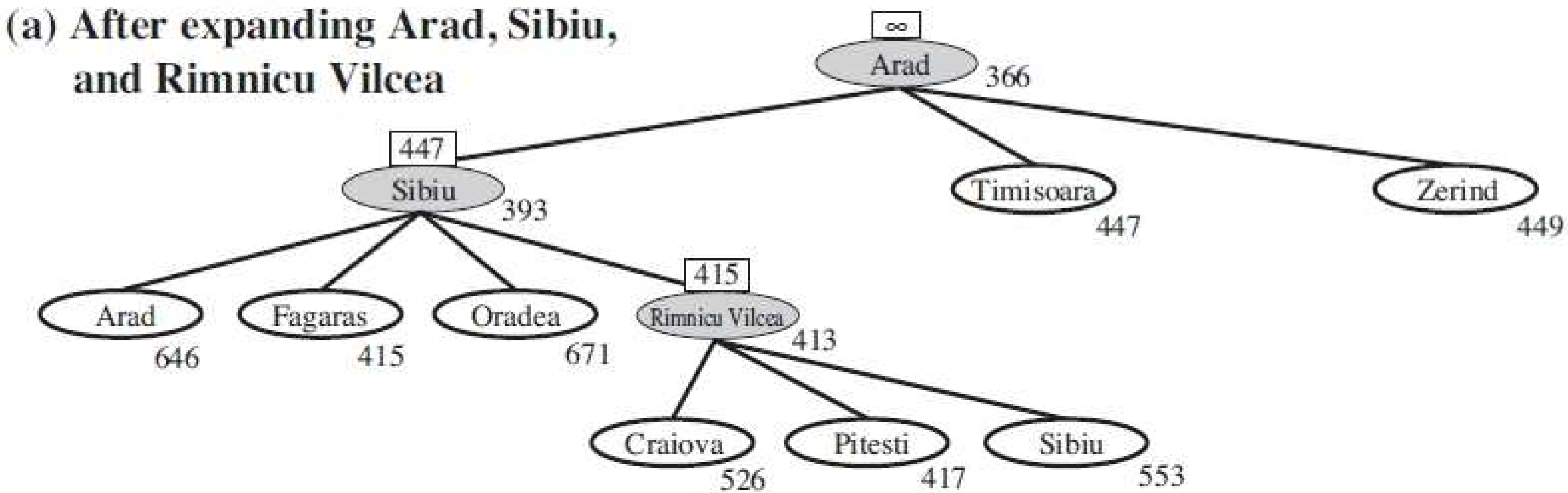
(e) After expanding Fagaras



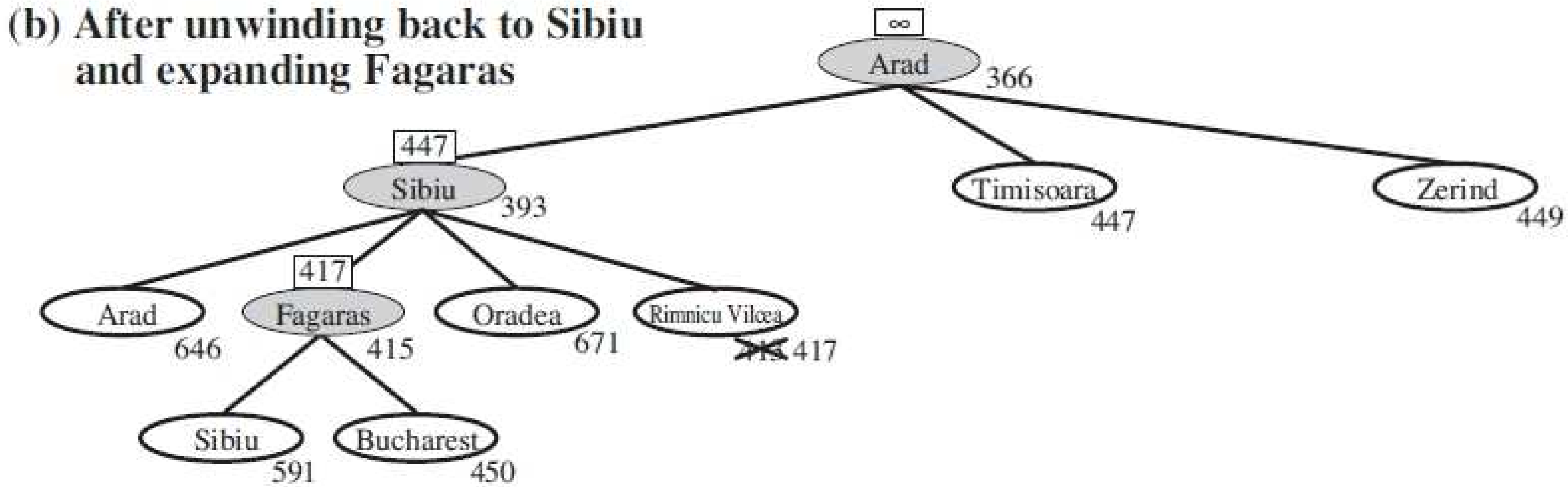
(f) After expanding Pitesti



Memory-bounded heuristic search 1

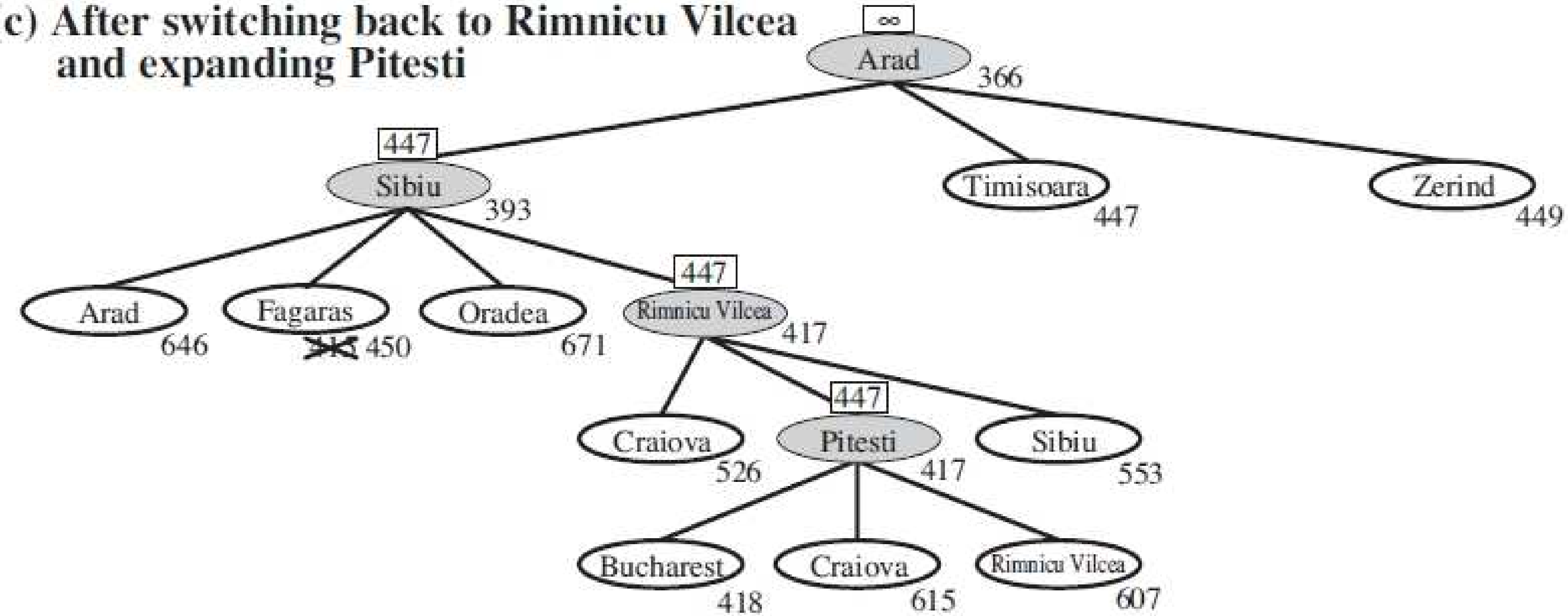


Memory-bounded heuristic search 2

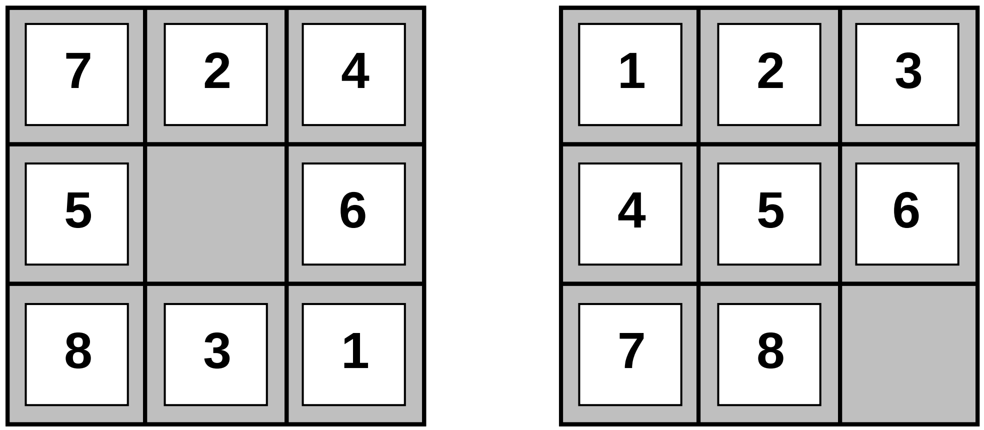


Memory-bounded heuristic search 3

(c) After switching back to Rimnicu Vilcea and expanding Pitesti



Heuristic functions



Start State **Goal State**

h_1 - the number of misplaced tiles

h_2 - the sum of the distances of the tiles from their goal positions

Is it possible for a computer to invent such a heuristic mechanically?

Summary

◊ Methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the agent can construct sequences of actions that achieve its goals - **search**.

◊ Before an agent can start searching for solutions, a goal must be identified and a welldefined problem must be formulated.

◊ A problem: 1) initial state, 2) a set of actions, 3) a transition model describing the results of those actions, 4) a goal test function, and 5) a path cost function.

◊ Search algorithms: completeness, optimality, time complexity, and space complexity.

◊ Uninformed search: breadth-first, uniform cost, depth-first, iterative deepening, bidirectional search

◊ Informed (heuristic) search: greedy-best first, A^*, memory bounded