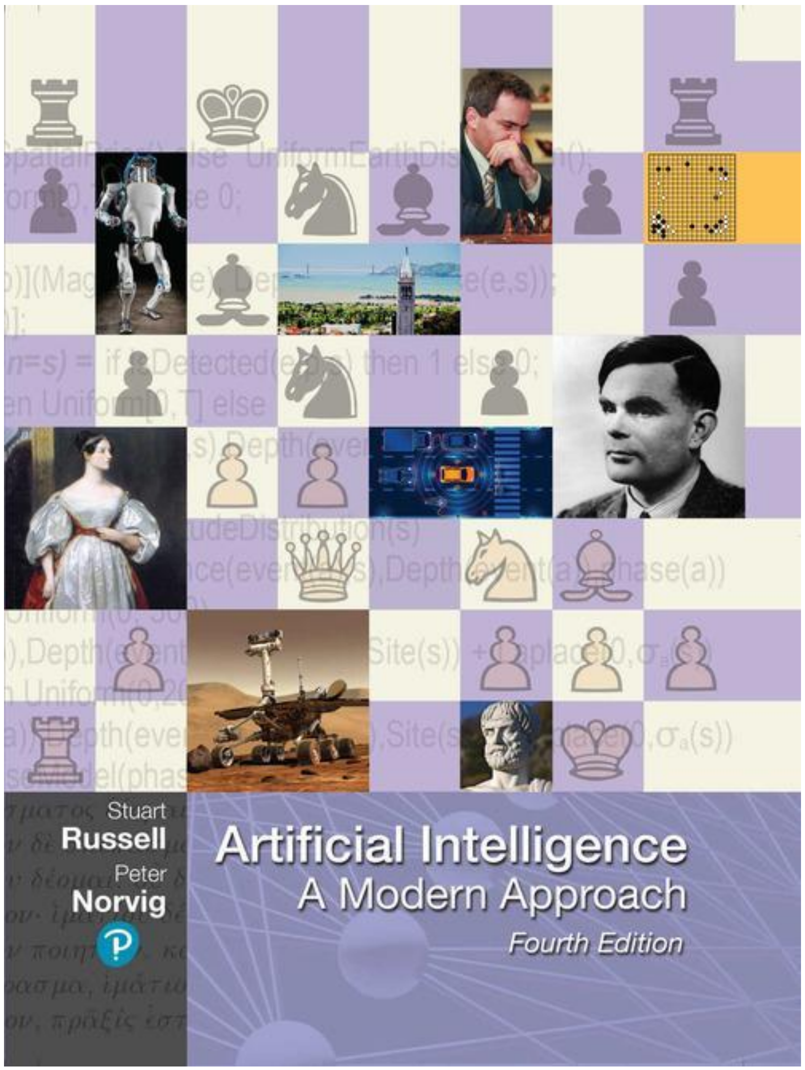
Artificial Intelligence: A Modern Approach

Fourth Edition



Chapter 3

Solving Problems By Searching

Outline

- Problem-solving agents
- Example Problems
- Problem formulation
- Search Algorithms
- Uninformed Search Strategies
- Informed (Heuristic) Search Strategies
- Heuristic Functions

Problem Definition

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.

• A set of one or more **goal states**. Sometimes there is one goal state (e.g., Bucharest), sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number).

• The **actions** available to the agent.

• An **action cost** function. A problem-solving agent should use a cost function that reflects its own performance measure (e.g., length in miles or time to complete action).

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

We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs. An **optimal solution has the lowest path cost among all solutions.**

Problem-solving agents

Restricted form of general agent:

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 ← Recommendation(*seq, state*)
  seq ← Remainder(*seq, state*)
  return *action*

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

Example: Romania

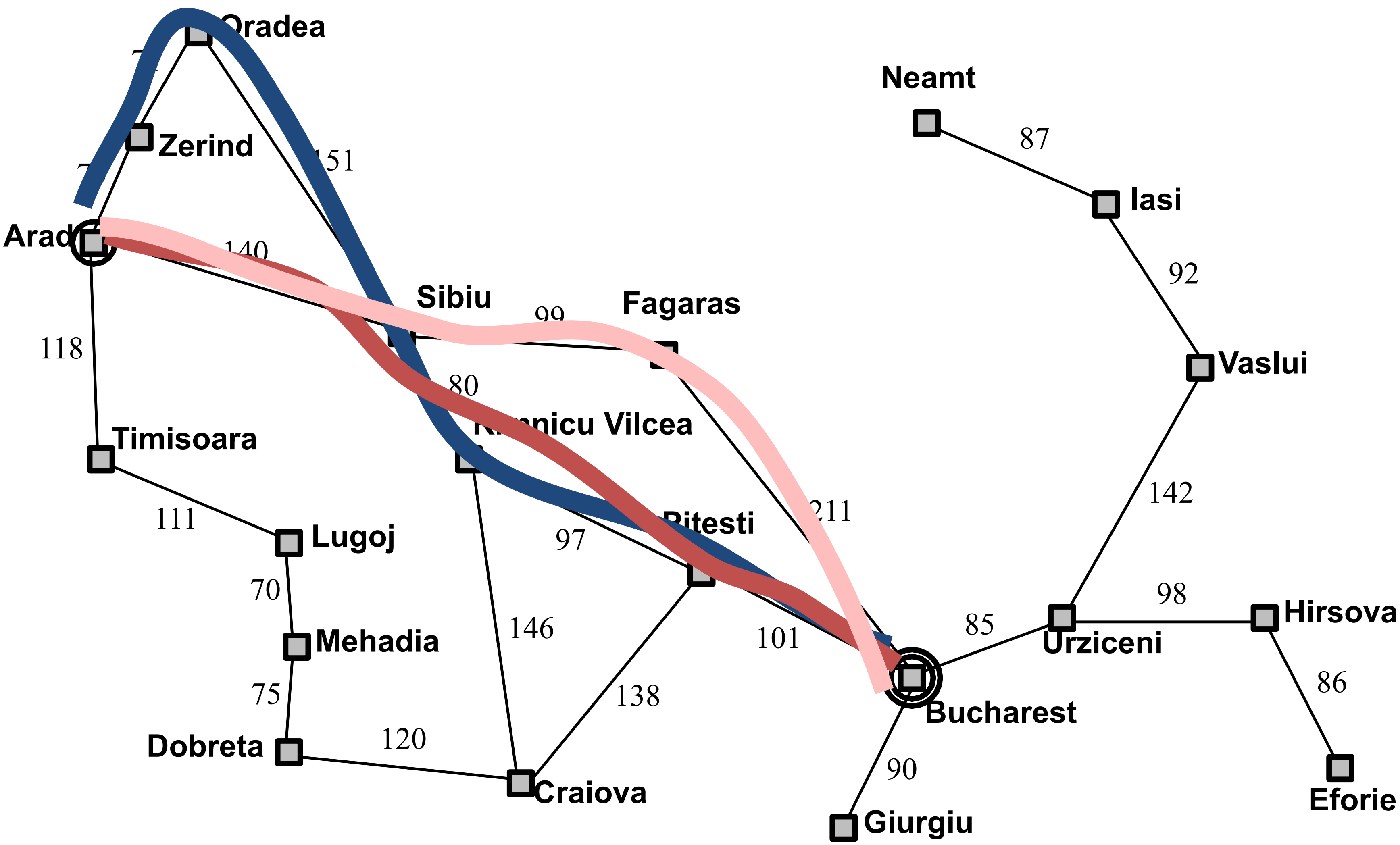
On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:
    be in Bucharest

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

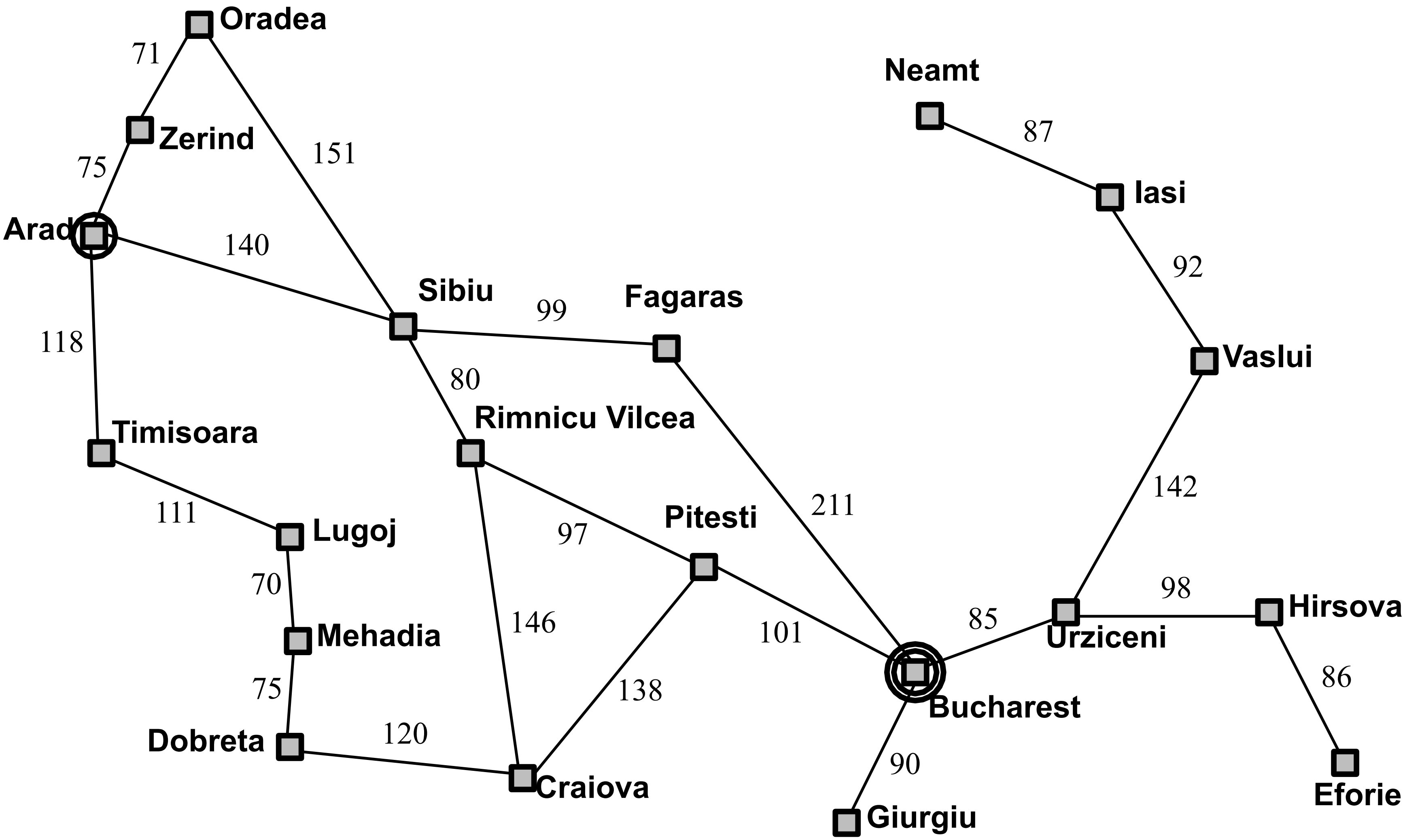
Find solution:
    sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



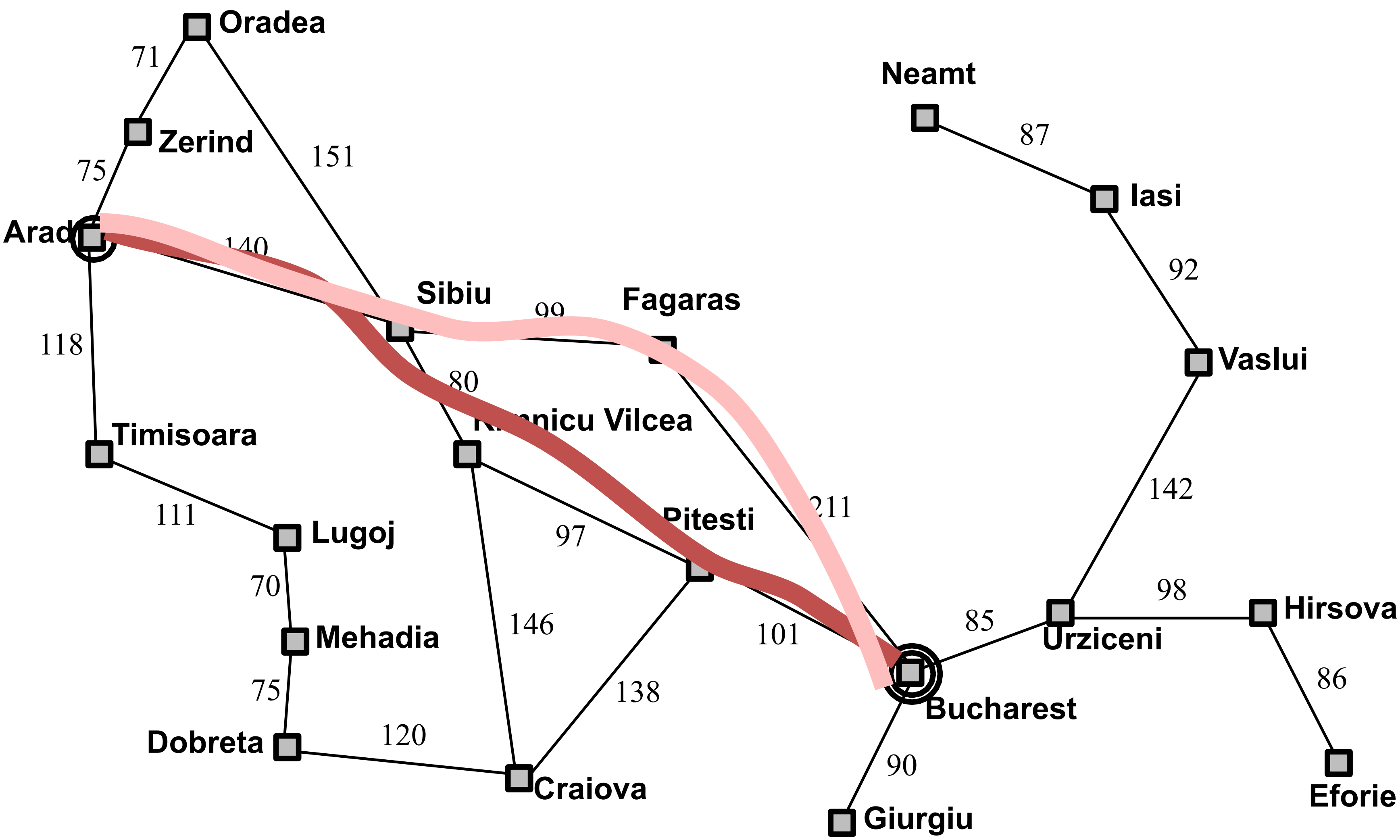
75	140	140	
71	99	80	
151	211	97	

Example: Romania



75	140	140	
71	99	80	
151	211	97	

Example: Romania



75	140	140	
71	99	80	
151	211	97	

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

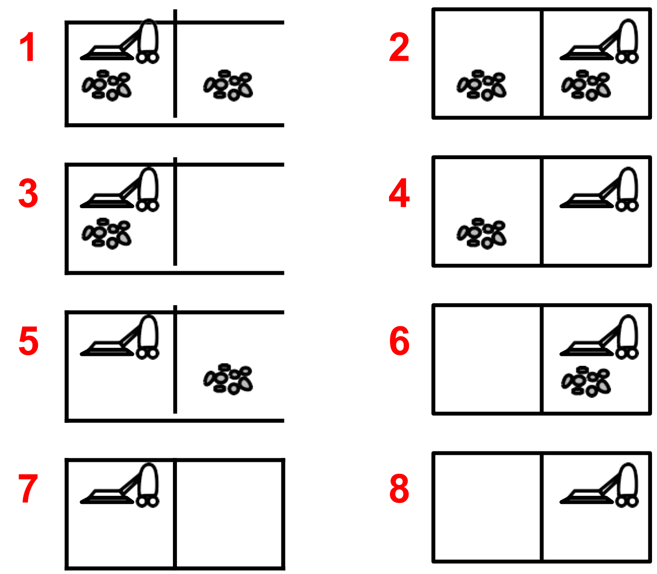
Nondeterministic and/or partially observable $\Longrightarrow$ contingency problem percepts provide new information about current state

solution is a contingent plan or a policy often interleave search, execution

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

Example: vacuum world

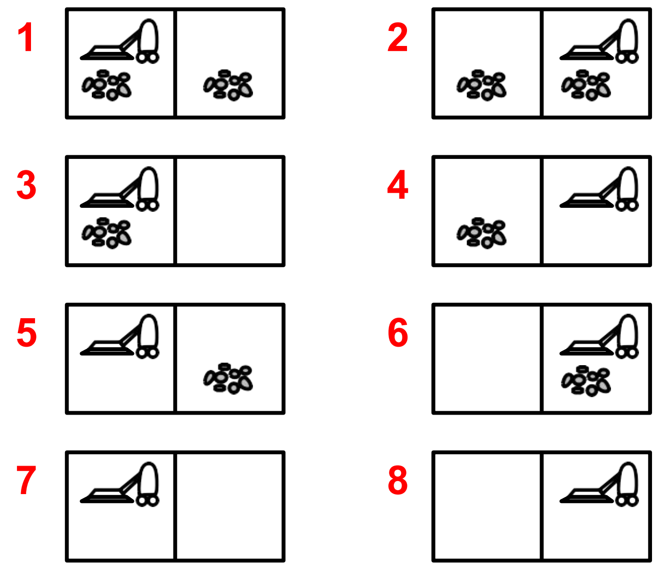
Single-state, start in #5. Solution??



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??

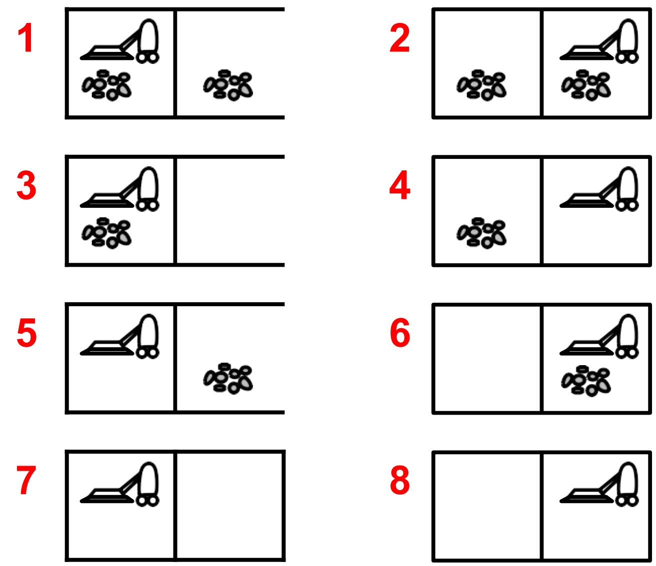


Example: vacuum world

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??

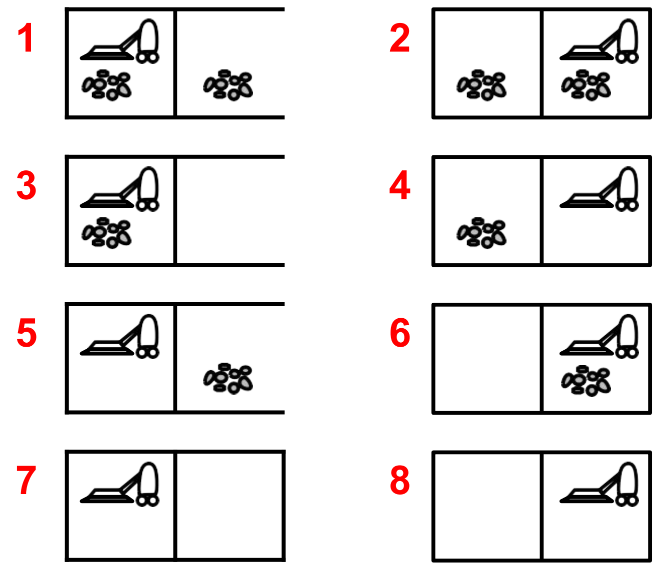


Example: vacuum world

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

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
⇒ state space must be abstracted for problem solving

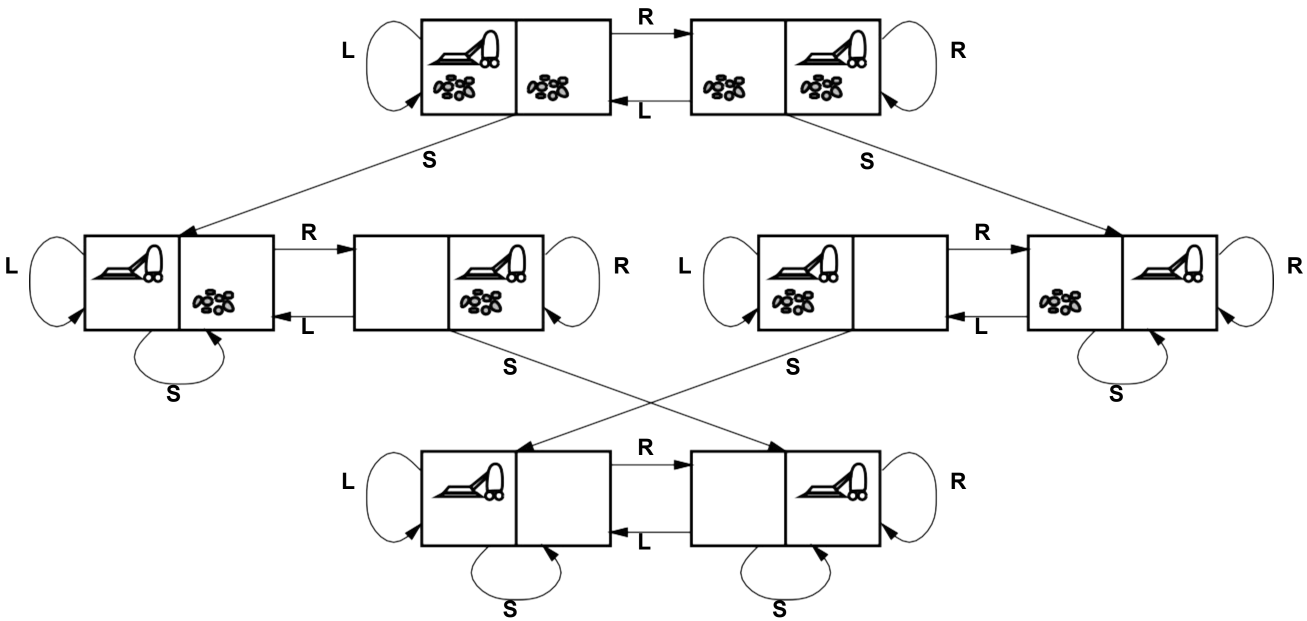
(Abstract) state = set of real states

(Abstract) action = complex combination of real actions e.g., "Arad → 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

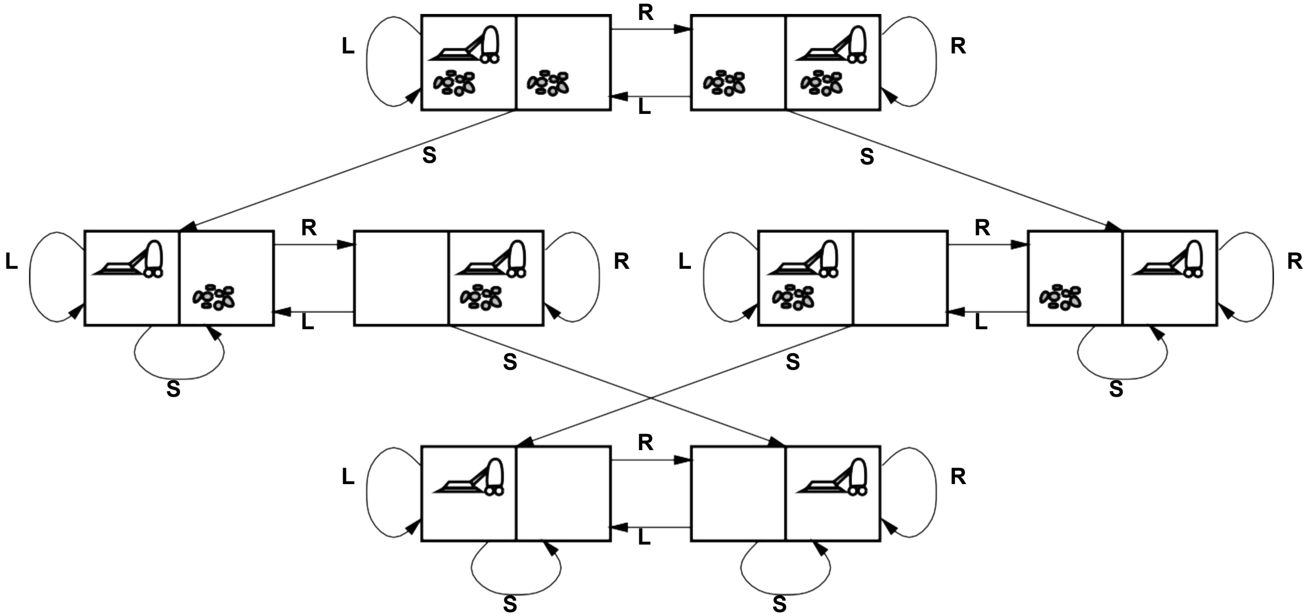
Each abstract action should be "easier" than the original problem!

Example: vacuum world state space graph



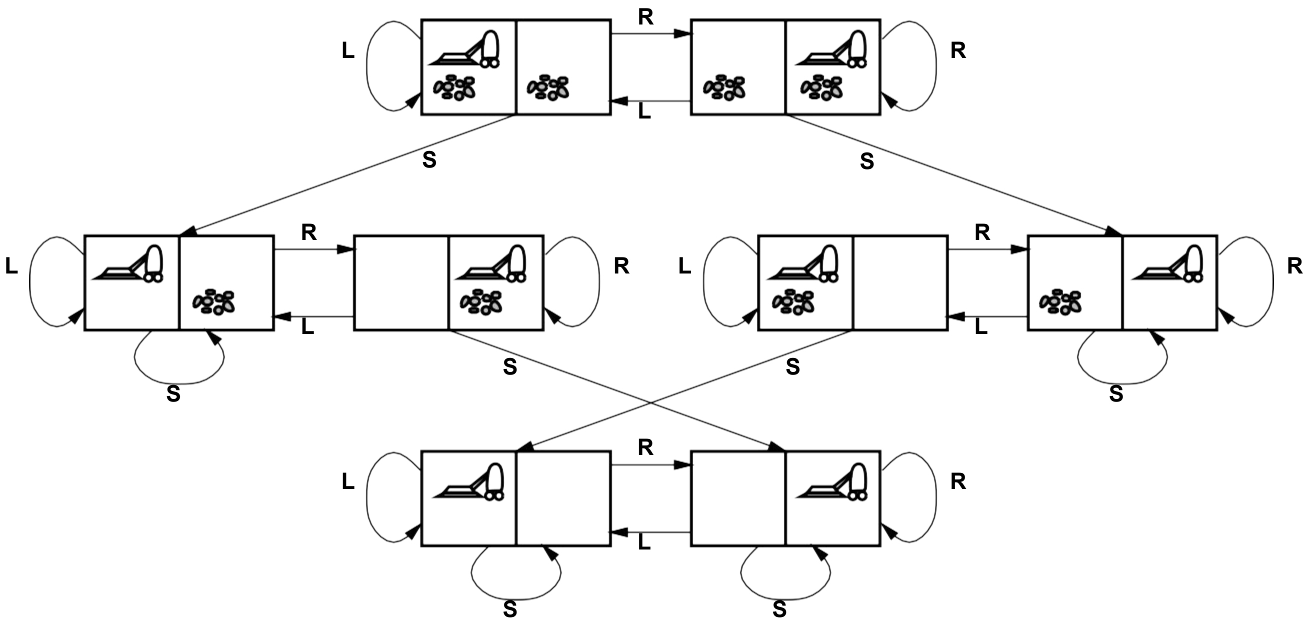
states??
actions??
goal
test??
path
cost??

Example: vacuum world state space graph



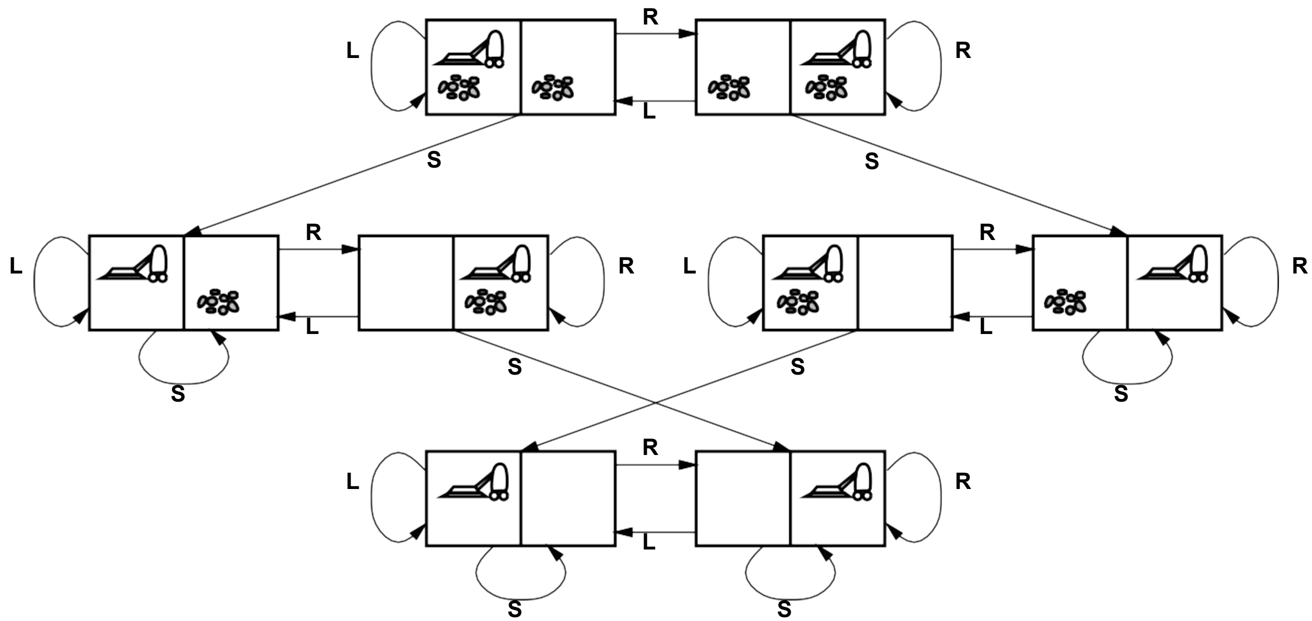
states??: integer dirt and robot locations (ignore dirt amounts etc.) actions??
goal
test??
path
cost??

Example: vacuum world state space graph



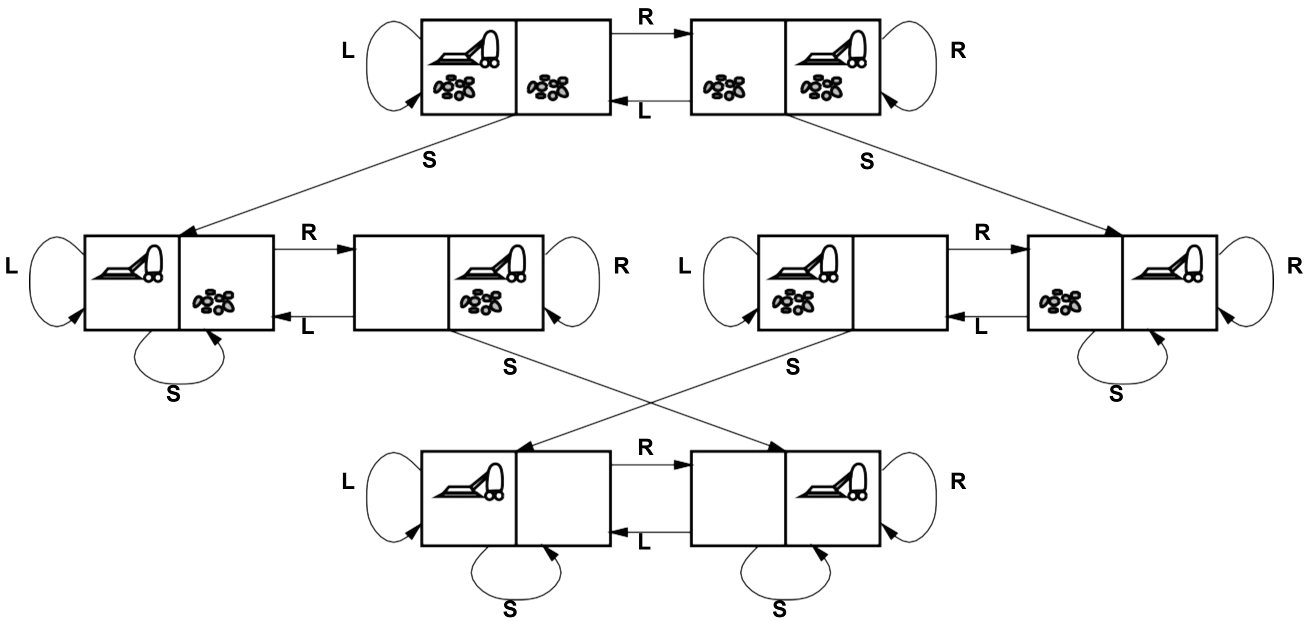
states??: integer dirt and robot locations (ignore dirt amounts etc.) actions??: *Left*, *Right*, *Suck*, *N oOp*
goal
test??
path
cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.) actions??: *Left*, *Right*, *Suck*, *N oOp* 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

7	2	4
5		6
8	3	1

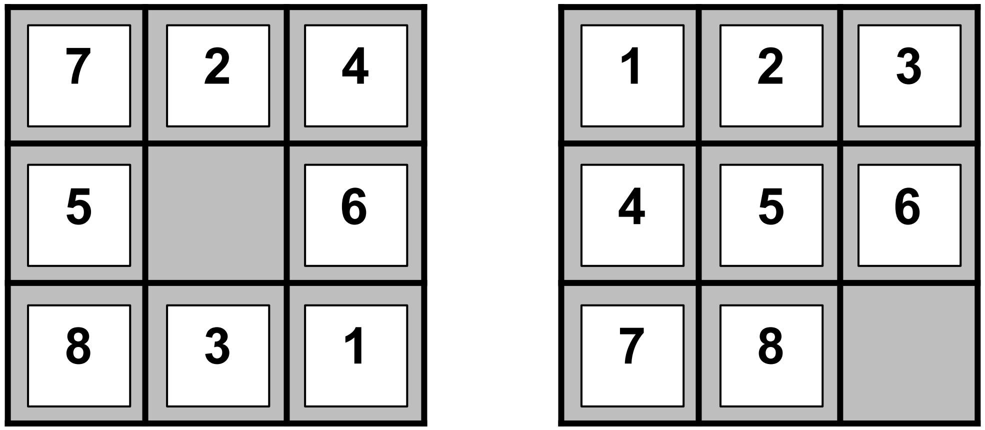
Start State

1	2	3
4	5	6
7	8	

Goal State

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

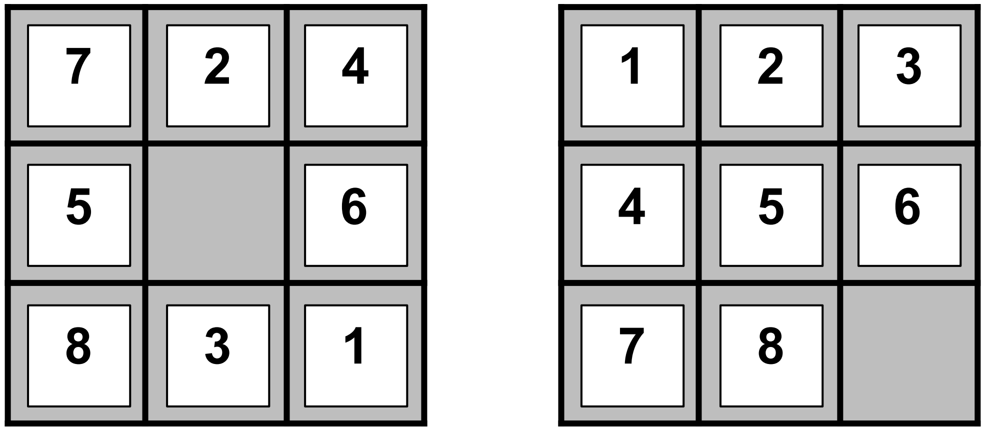
Example: The 8-puzzle



Start State Goal State

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

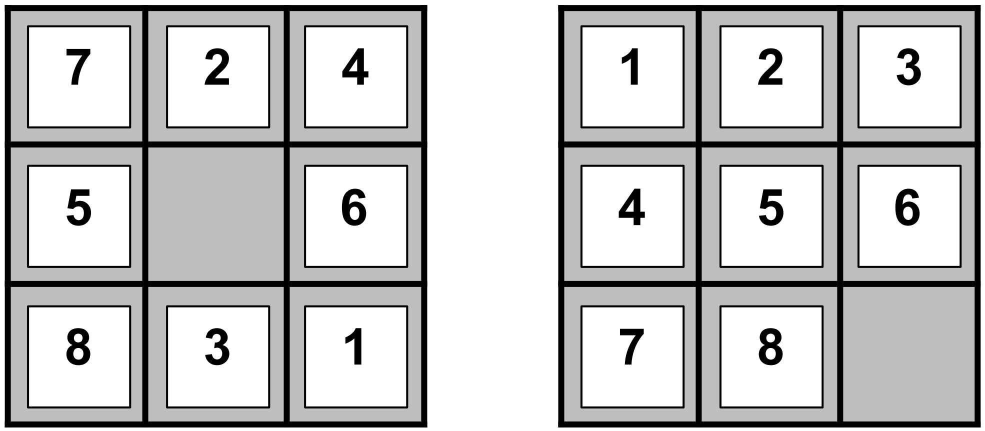
Example: The 8-puzzle



Start State **Goal State**

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

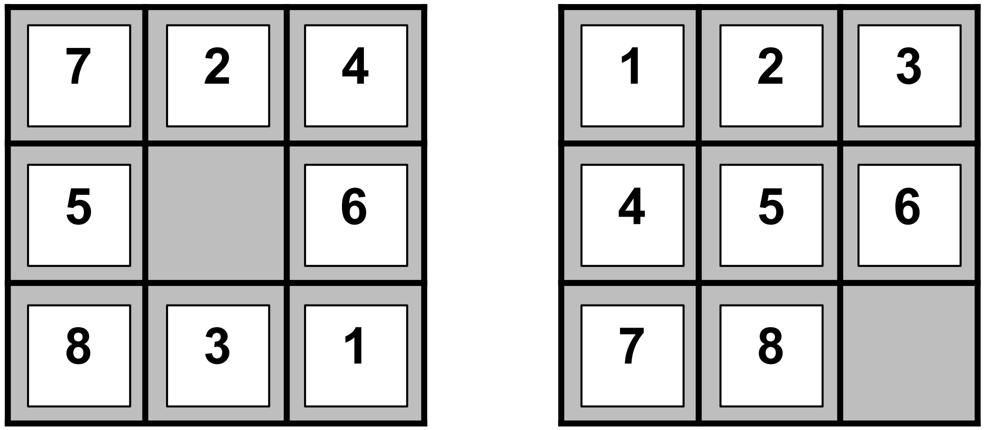
Example: The 8-puzzle



Start State **Goal State**

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

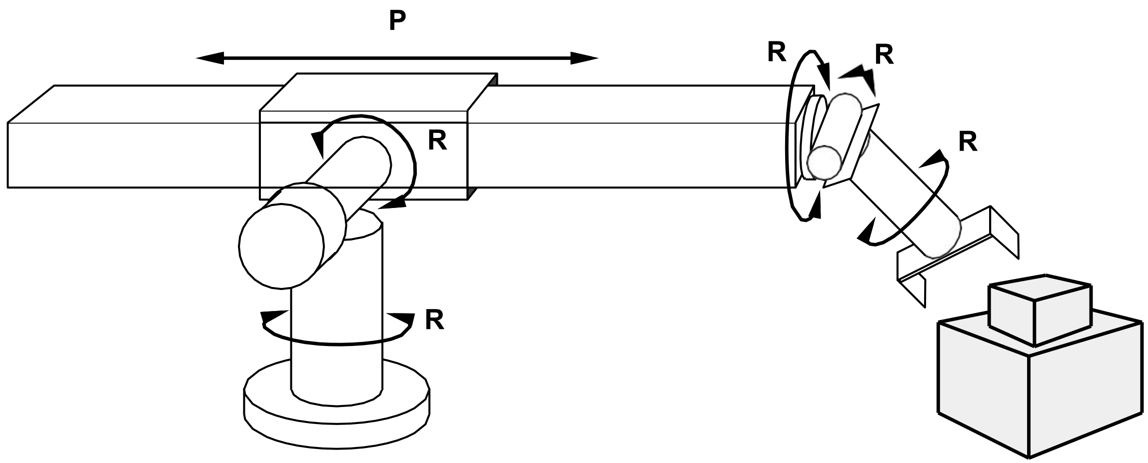


Start State Goal State

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

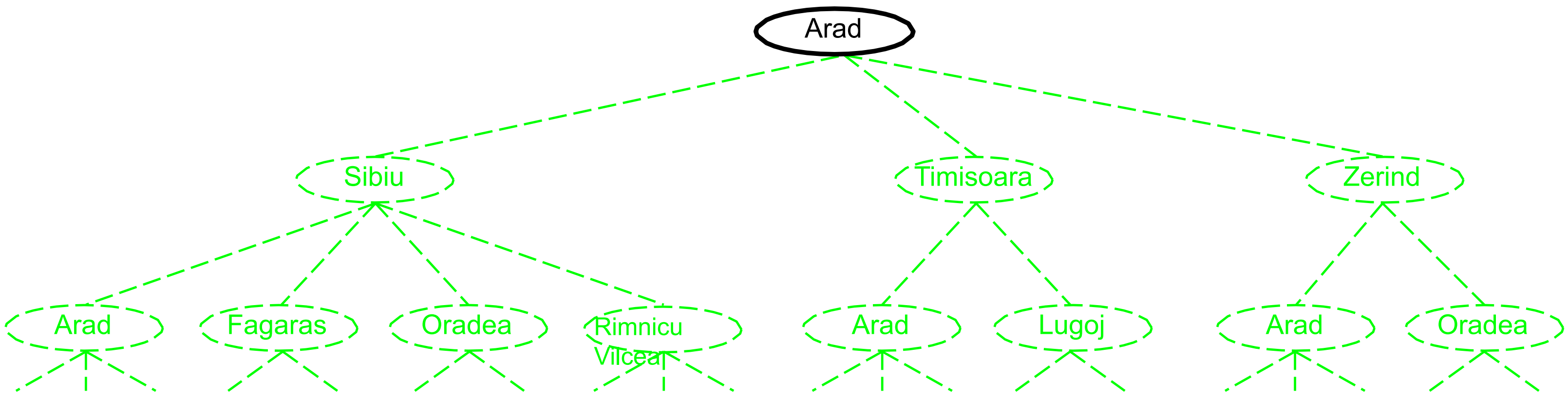
Tree search algorithms

Basic idea:

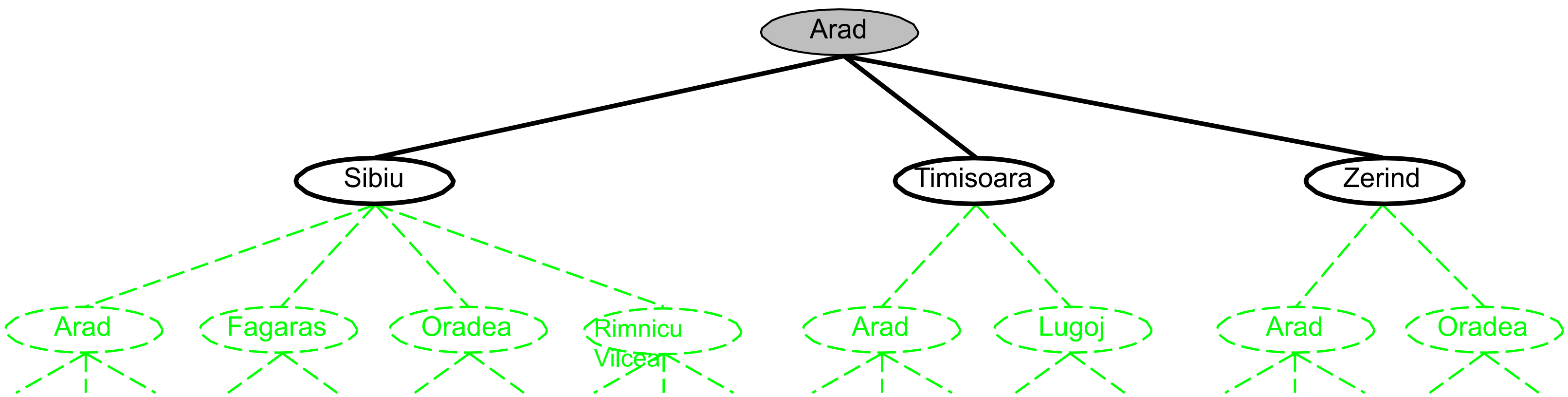
offline, simulated exploration of state space
by generating successors of already-explored
states (a.k.a. expanding states)

```
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
```

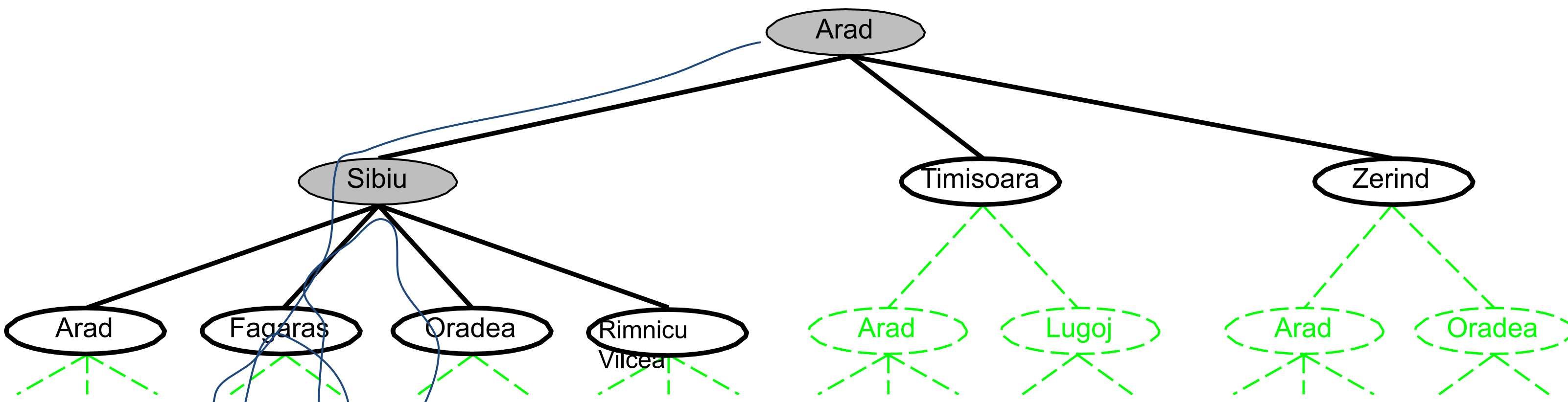
Tree search example



Tree search example



Tree search example

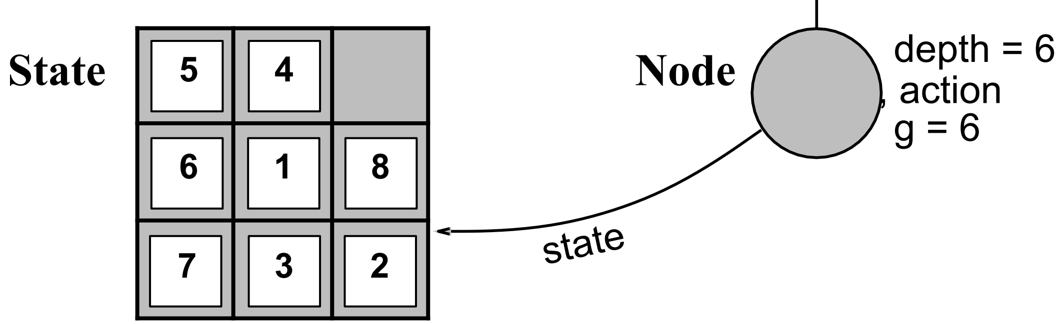


Implementation:states vs. nodes

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

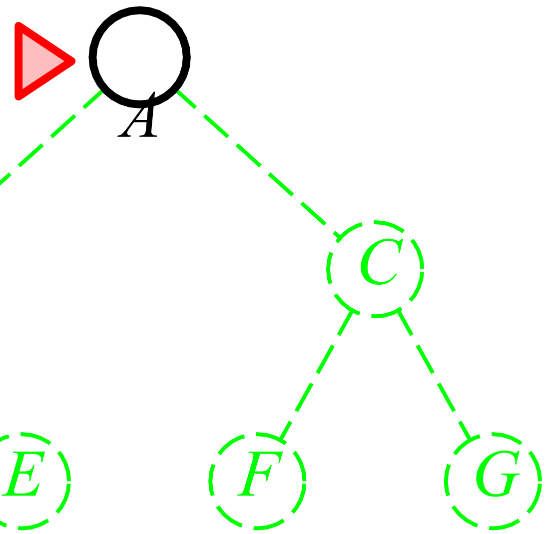
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

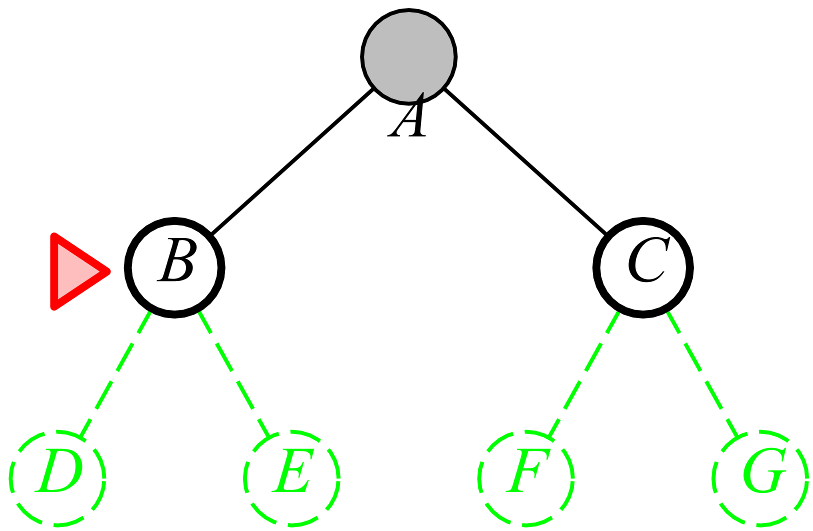


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

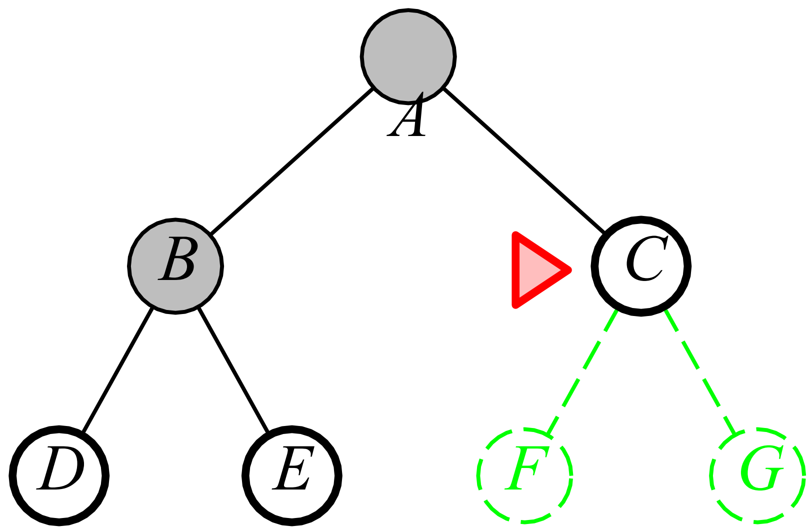


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

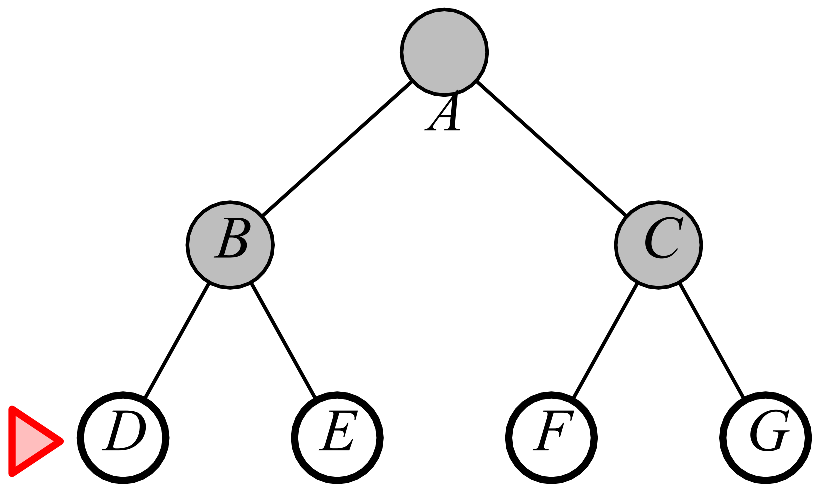


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

memory) Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

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 $\geq c$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/c \rceil})$

where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/c \rceil})$

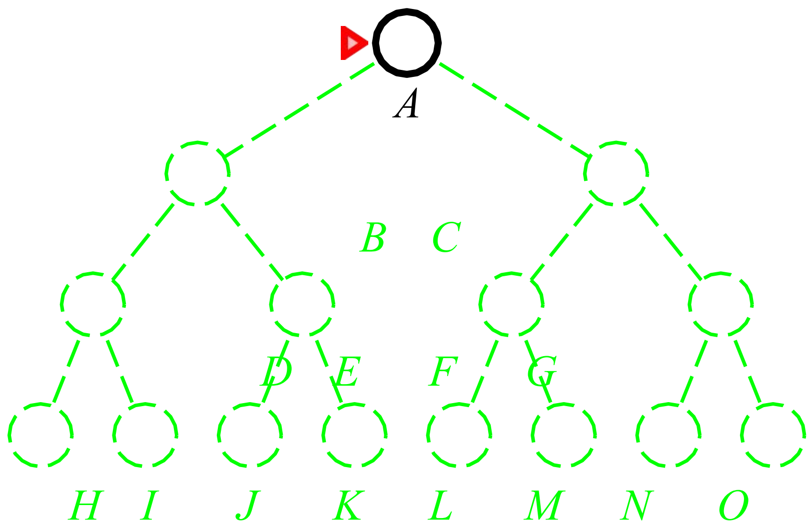
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

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

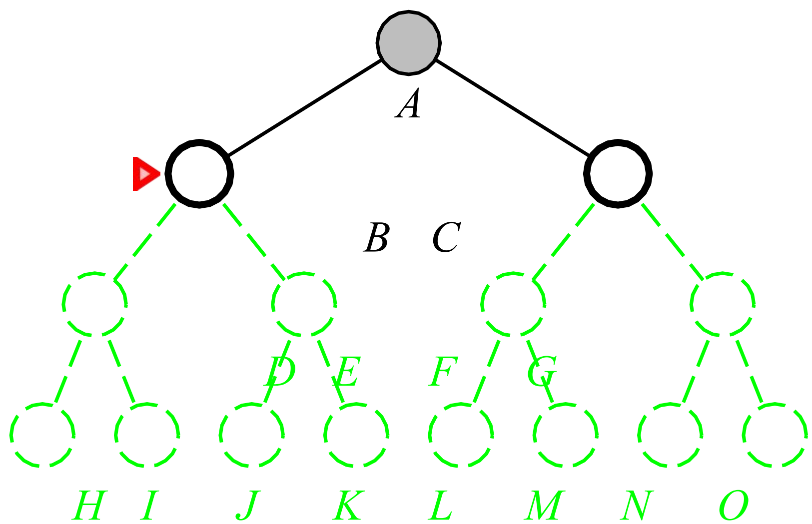


Depth-first search

Expand deepest unexpanded node

Implementation:

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

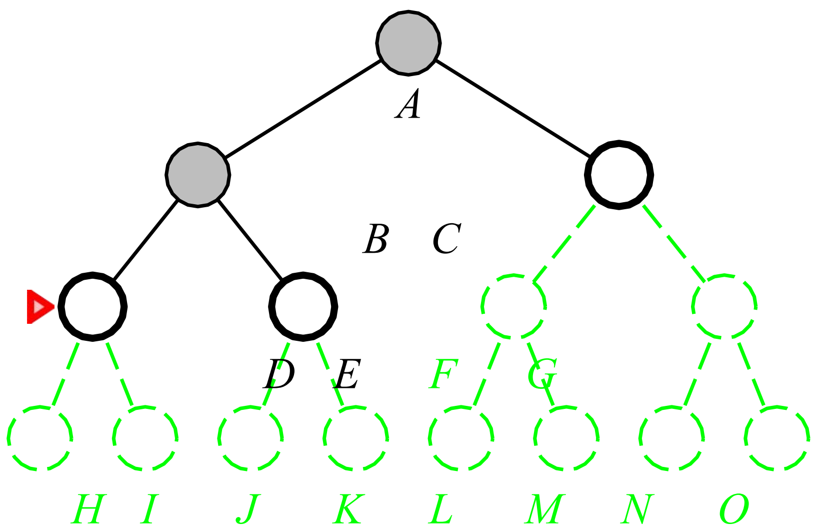


Depth-first search

Expand deepest unexpanded node

Implementation:

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

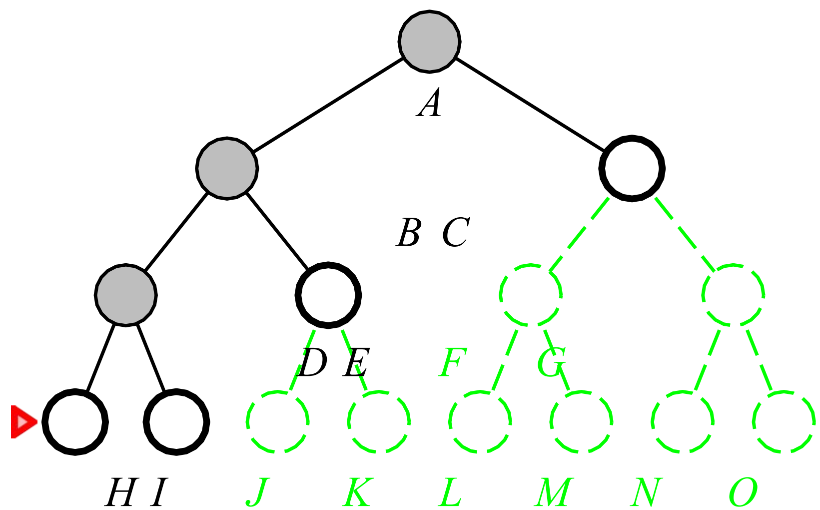


Depth-first search

Expand deepest unexpanded node

Implementation:

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

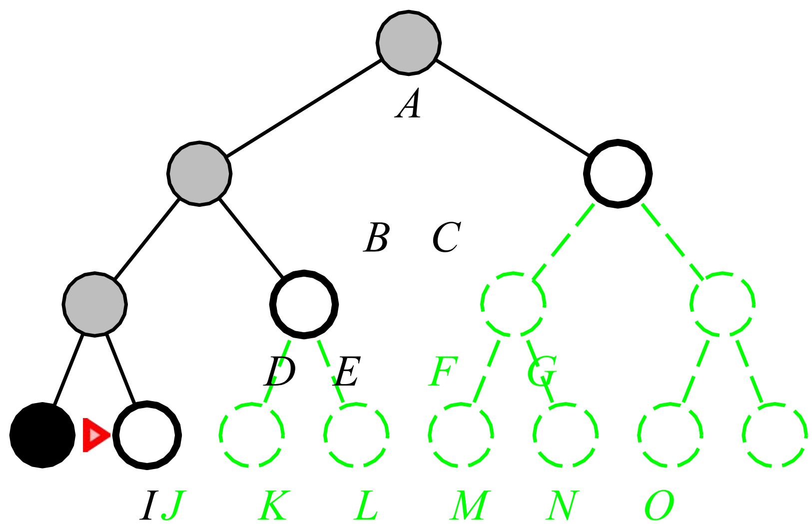


Depth-first search

Expand deepest unexpanded node

Implementation:

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

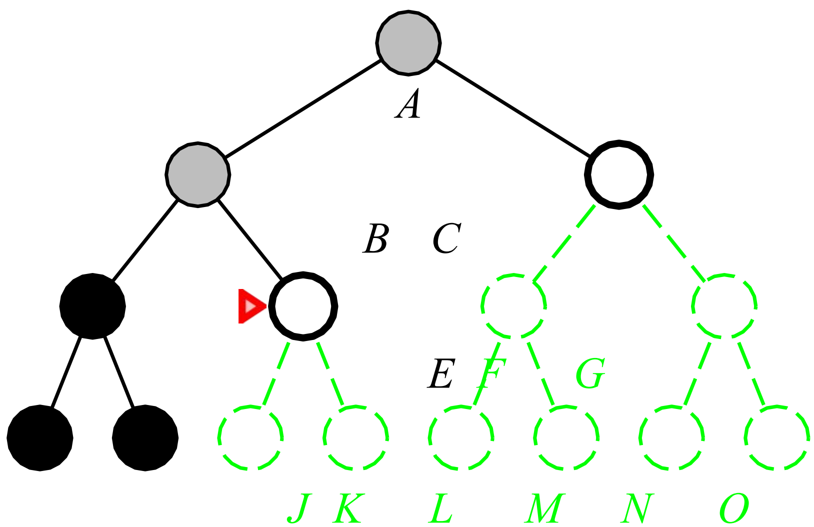


Depth-first search

Expand deepest unexpanded node

Implementation:

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

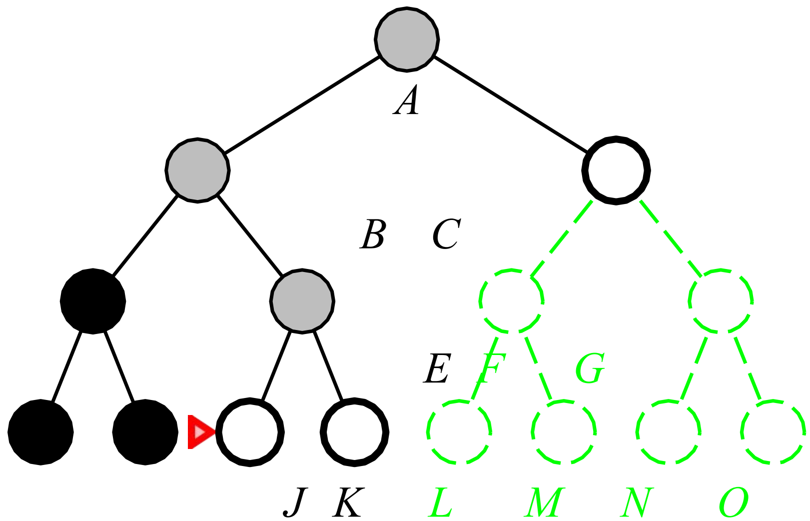


Depth-first search

Expand deepest unexpanded node

Implementation:

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

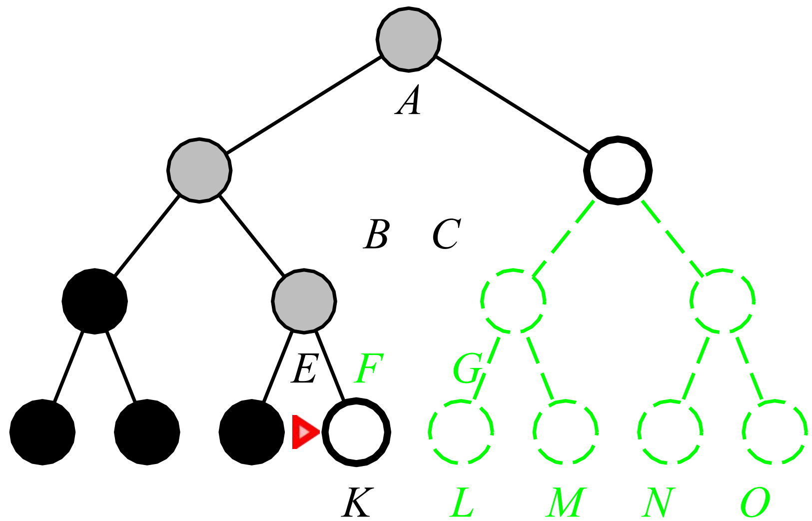


Depth-first search

Expand deepest unexpanded node

Implementation:

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

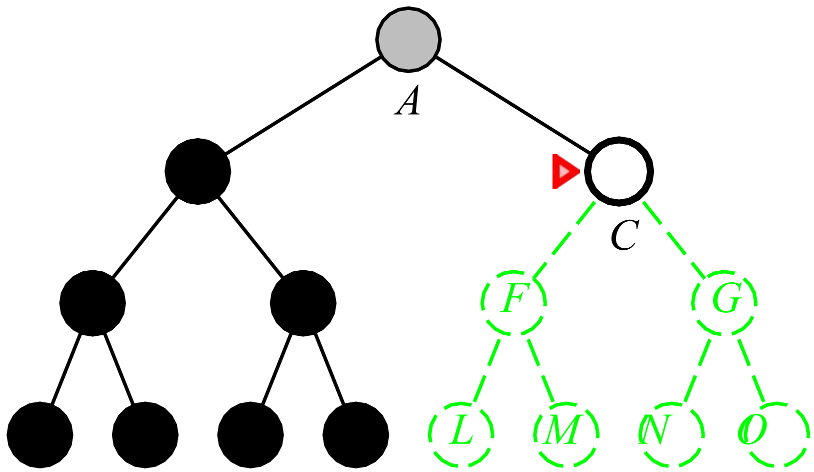


Depth-first search

Expand deepest unexpanded node

Implementation:

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

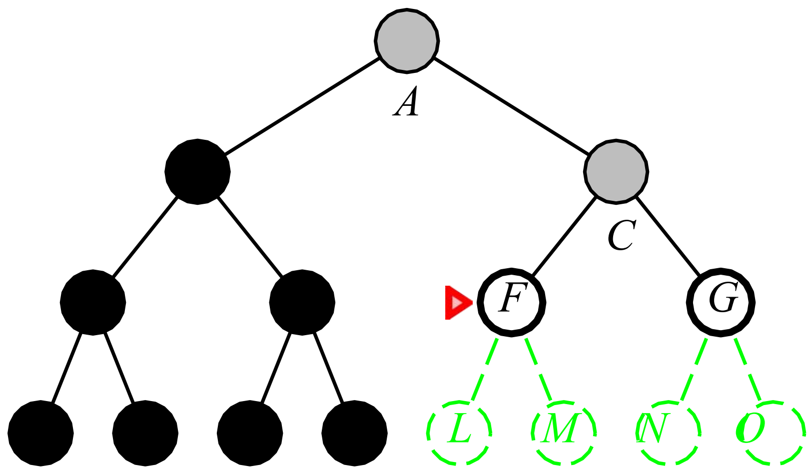


Depth-first search

Expand deepest unexpanded node

Implementation:

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

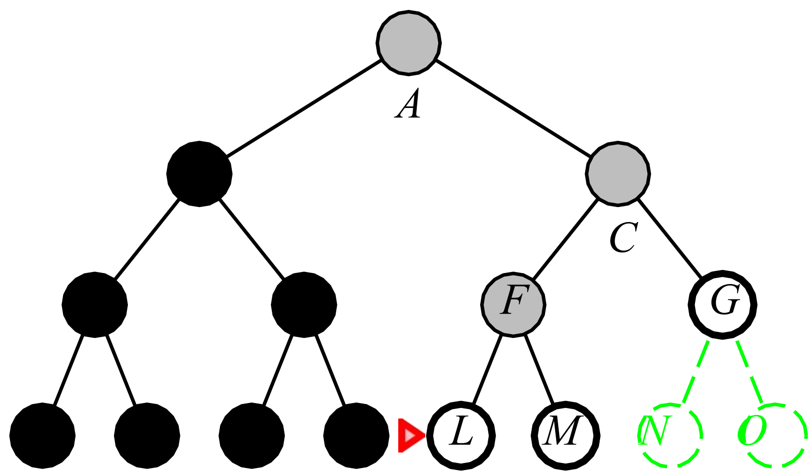


Depth-first search

Expand deepest unexpanded node

Implementation:

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

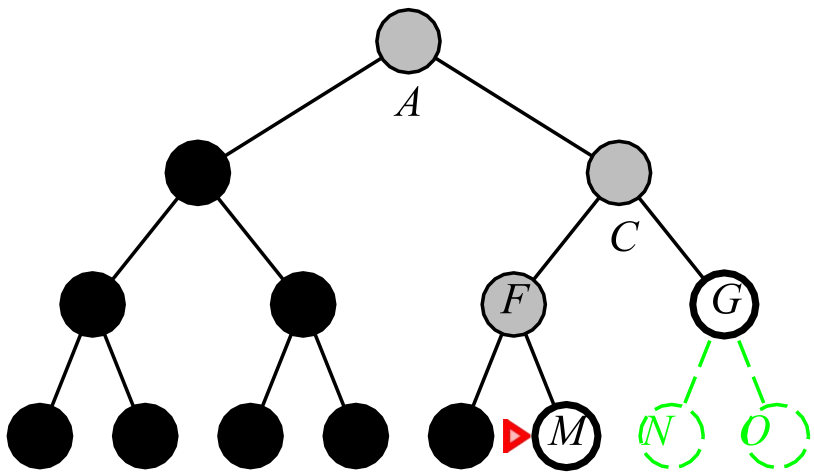


Depth-first search

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 Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

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

Depth-limited search

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

Recursive implementation:

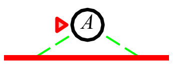
```
function Depth-Limited-Search( problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

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
```

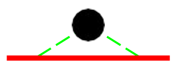
Iterative deepening search

```
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
```

Iterative deepening search $l = 0$

Limit = 0

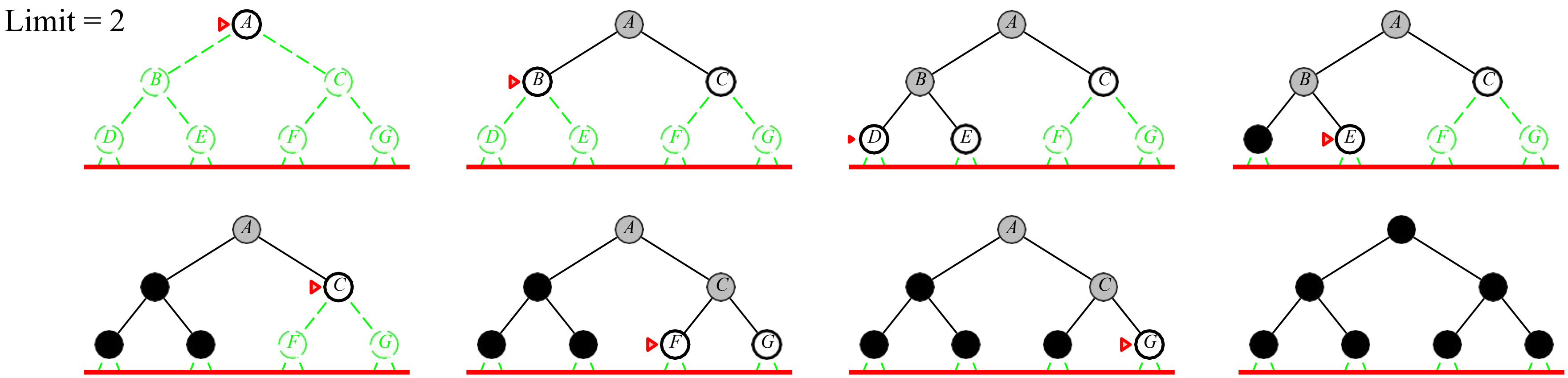




Iterative deepening search $l = 1$

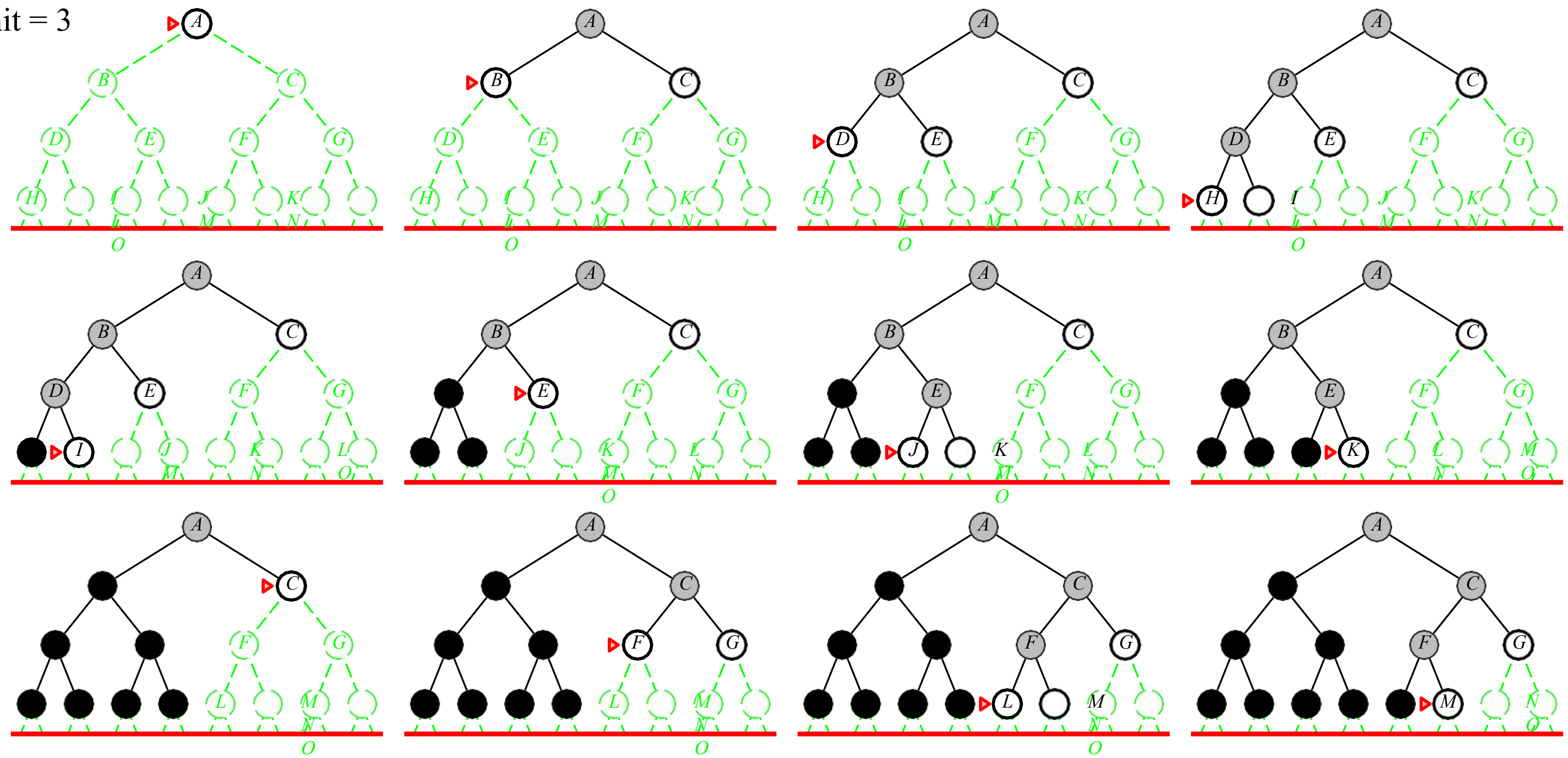
Limit = 1

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

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:

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

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

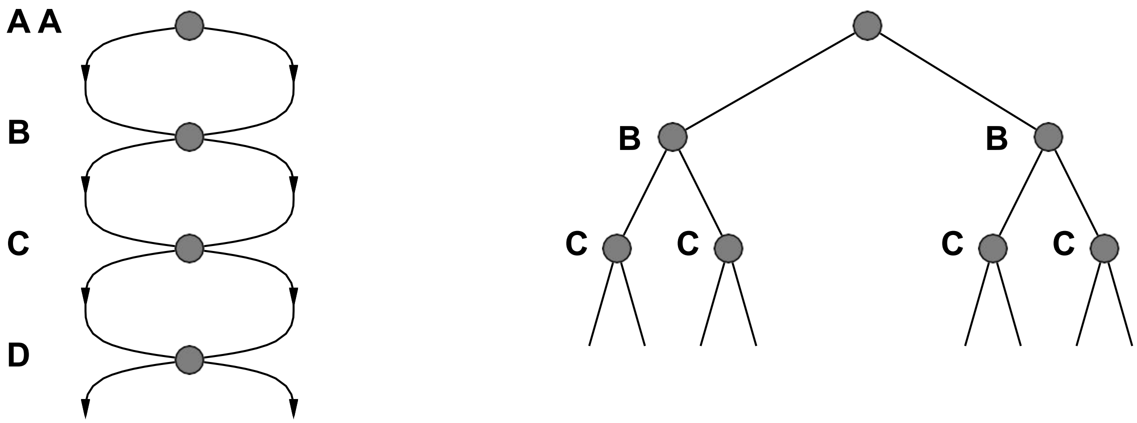
Summary of Uninformed Search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Repeated states

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



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 fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State[node]) then return node
        if State[node] is not in closed then
            add State[node] to closed
            fringe ← InsertAll(Expand(node, problem), fringe)
    end
```

Review: 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] applied to State(node) succeeds return node
        fringe ← InsertAll(Expand(node, problem), fringe)
```

A strategy is defined by picking the order of node expansion

Best-first search

Idea: use an evaluation function for each node

– estimate of "desirability"

⇒ Expand most desirable unexpanded

node Implementation:

fringe is a queue sorted in decreasing order of desirability

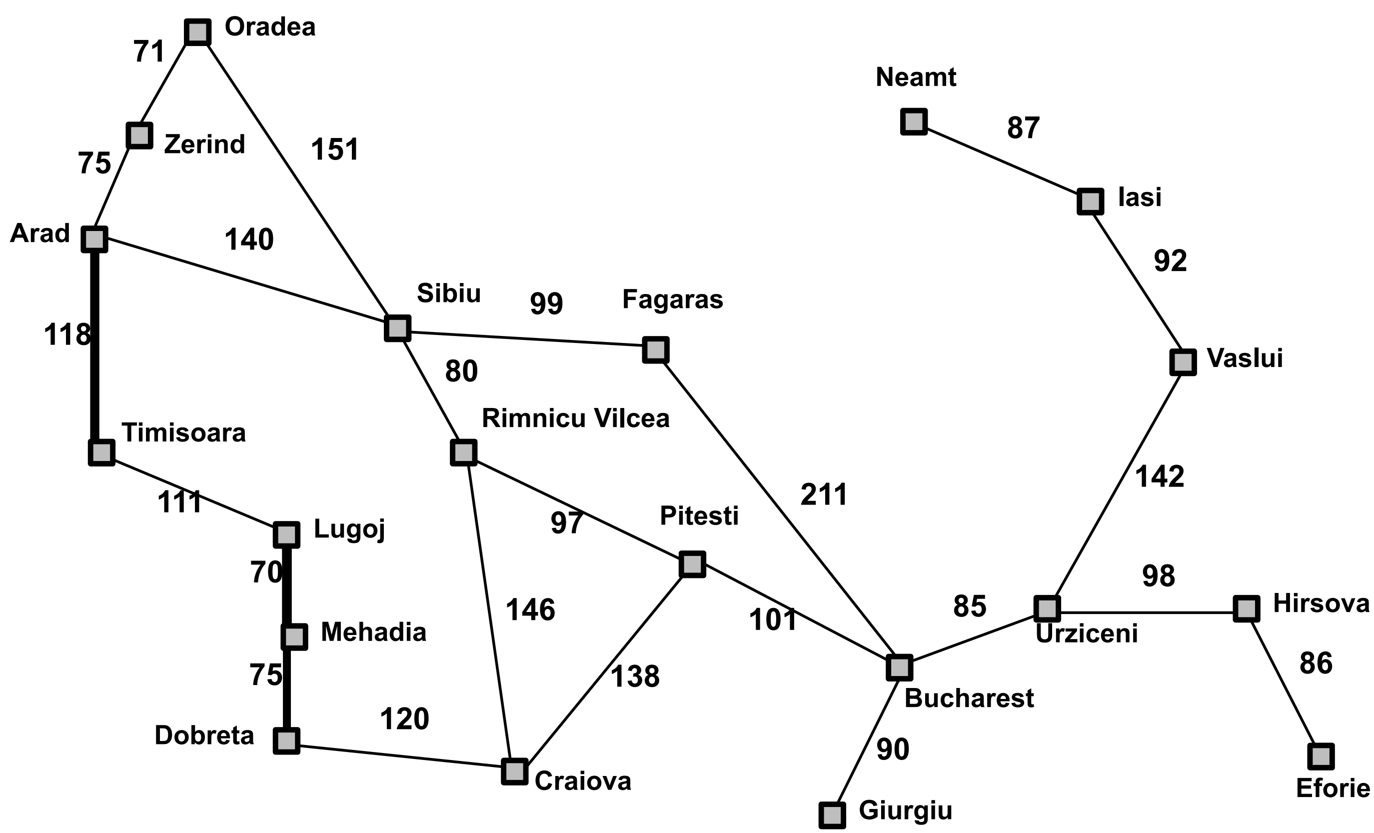
Special cases:

greedy

search A*

search

Romania with step costs in km



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

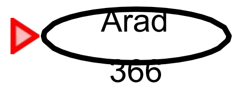
Evaluation function $h(n)$ (heuristic)

= estimate of cost from n to the closest goal

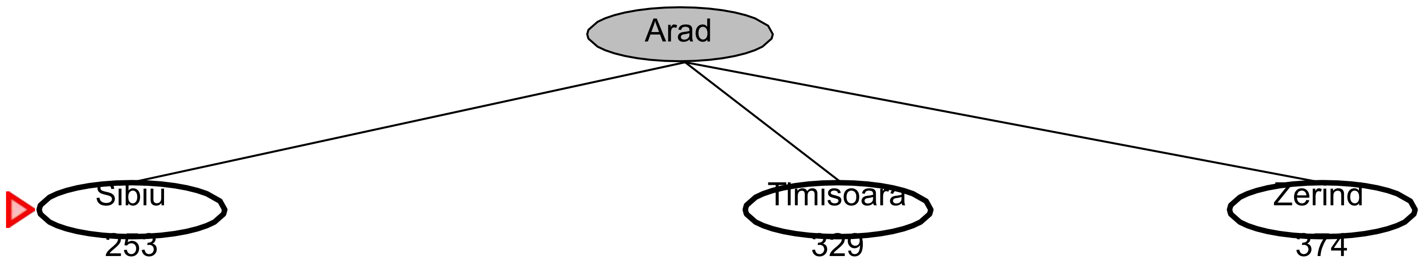
E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

Greedy search expands the node that appears to be closest to goal

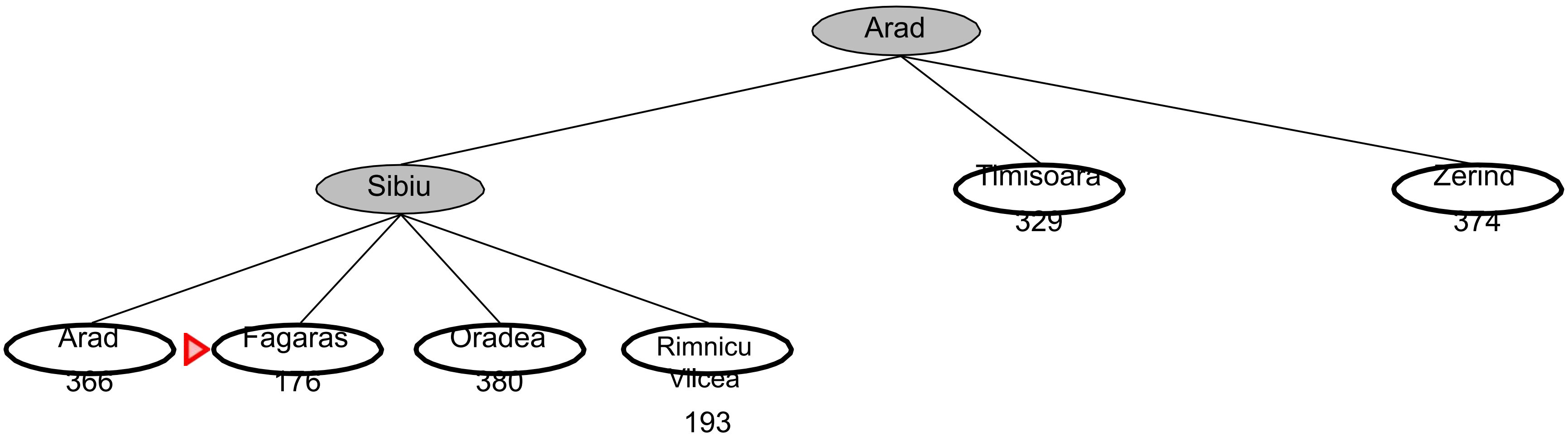
Greedy search example



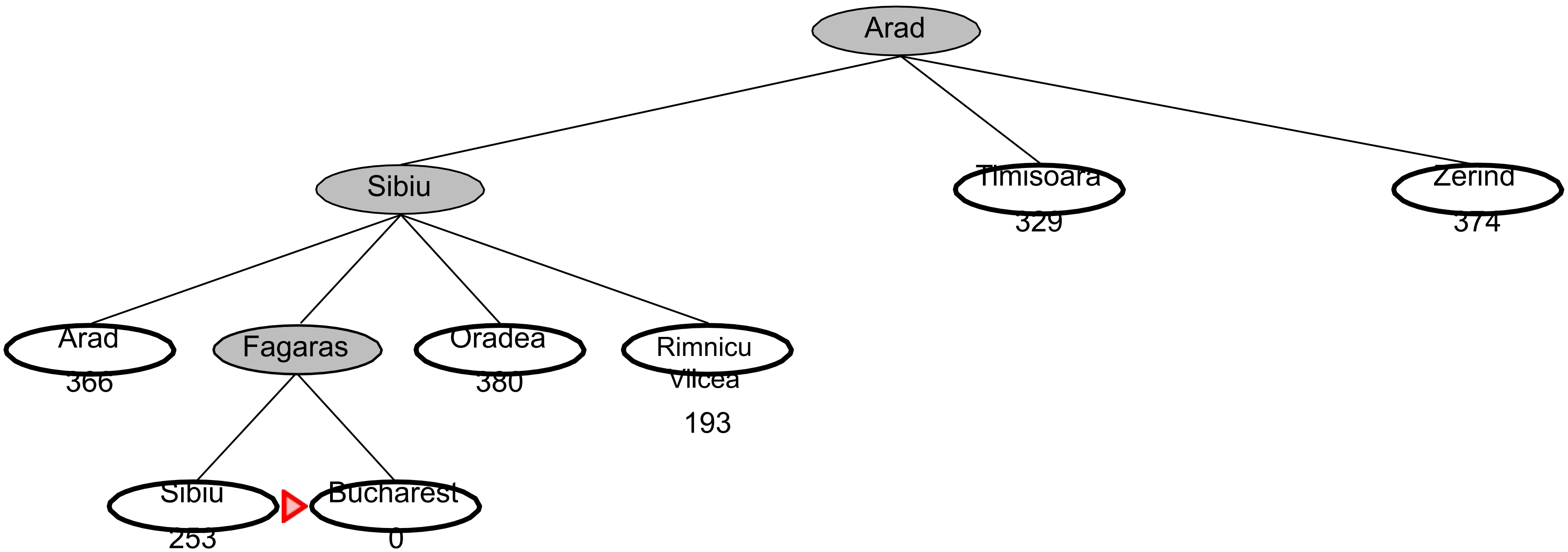
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g., with Oradea as goal, Iasi → Neamt → Iasi → Neamt →
Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No–can get stuck in loops,
e.g., Iasi → Neamt → Iasi →
Neamt →
Complete in finite space with repeated-state checking
Time?? $O(b^m)$, but a good heuristic can give dramatic

improvement Space??

Properties of greedy search

Complete?? No–can get stuck in loops,
e.g., Iasi → Neamt → Iasi →
Neamt →
Complete in finite space with repeated-state checking
Time?? $O(b^m)$, but a good heuristic can give dramatic

improvement Space?? $O(b^m)$—keeps all nodes in memory

Optimal??

Properties of greedy search

Complete?? No–can get stuck in loops,
e.g., Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$
Neamt $\rightarrow$
Complete in finite space with repeated-state checking
Time?? $O(b^m)$, but a good heuristic can give dramatic

improvement Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No

Pearson

A^* search

Idea: avoid expanding paths that are already expensive
Evaluation function $f(n) = g(n) + h(n)$

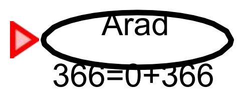
$g(n)$ = cost so far to reach n

$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

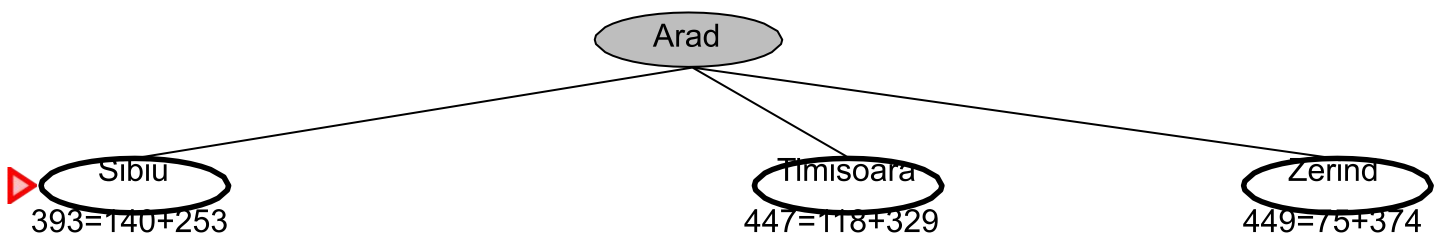
A* search uses an admissible heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n.
(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)
E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road

distance Theorem: A* search is optimal

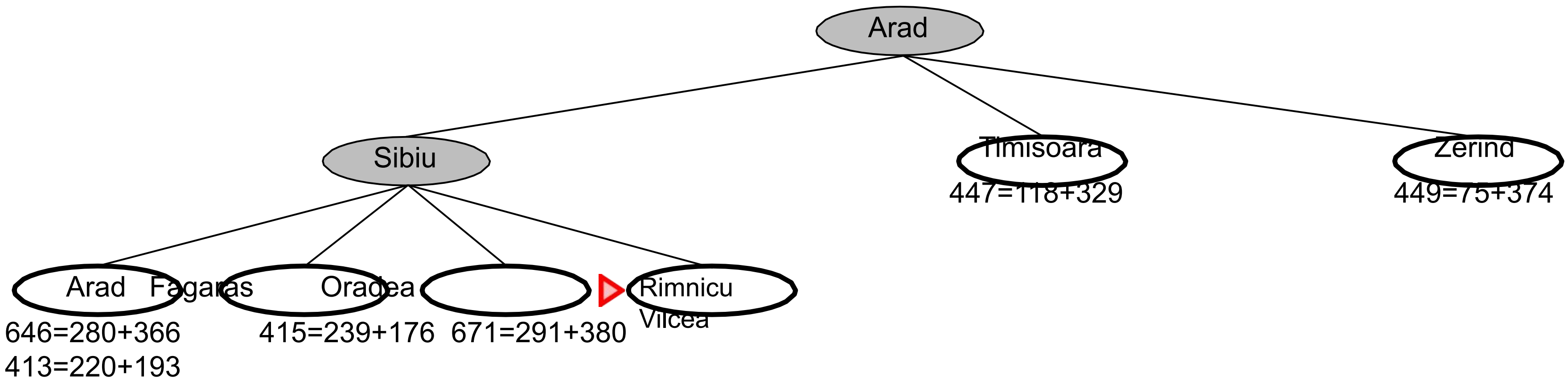
A^* search example



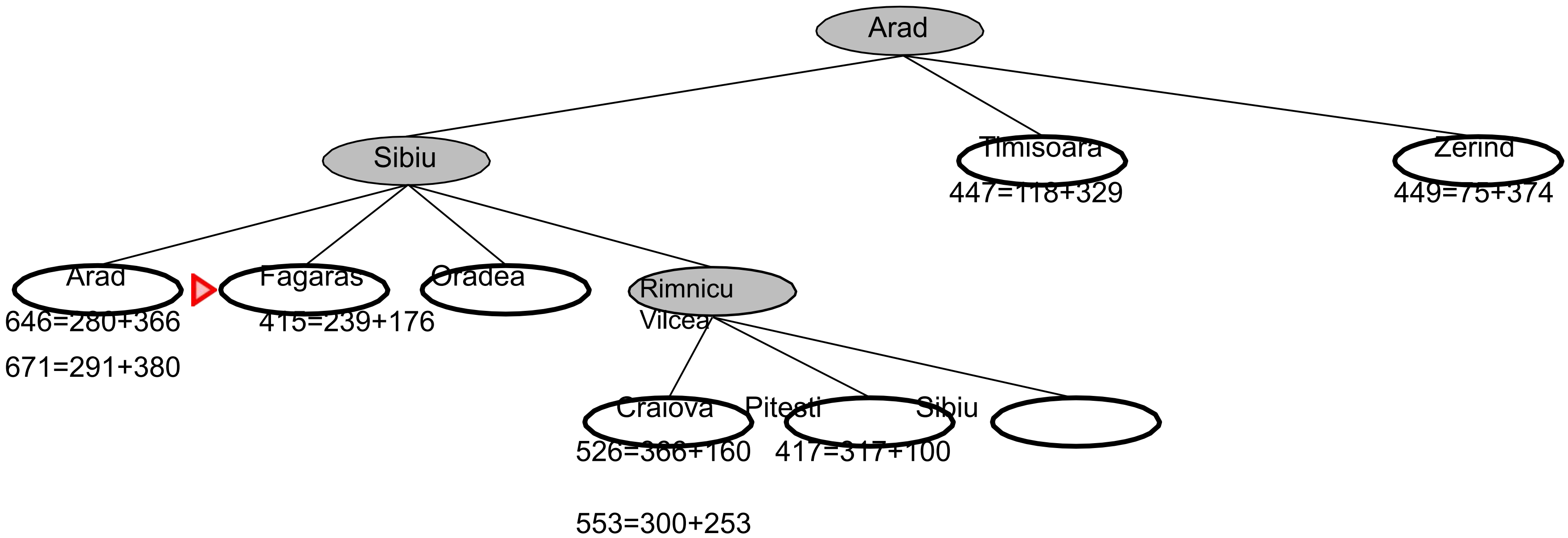
A* search example



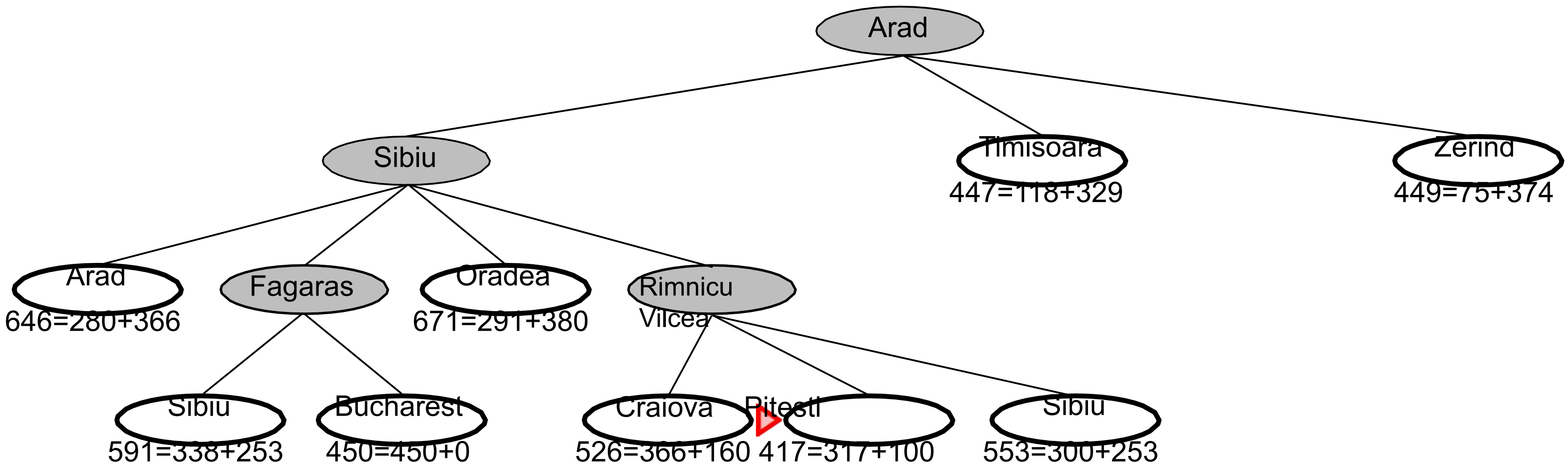
A* search example



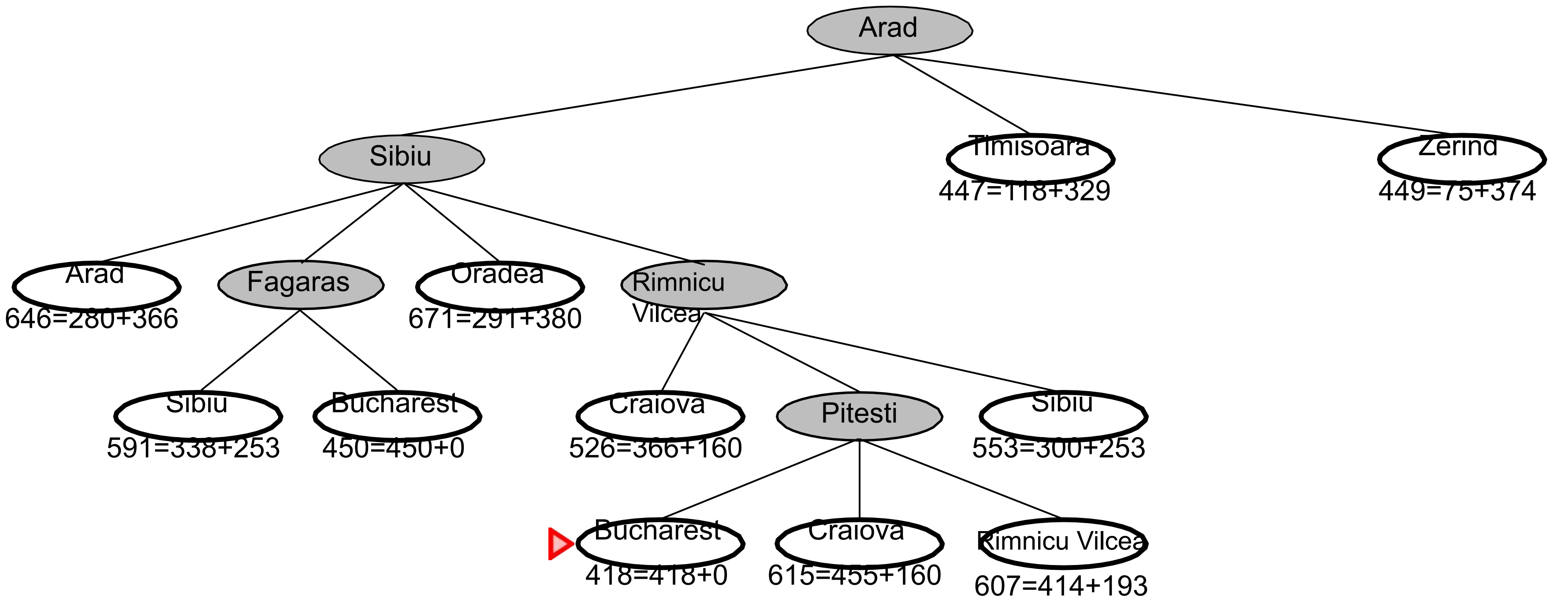
A* search example



A* search example

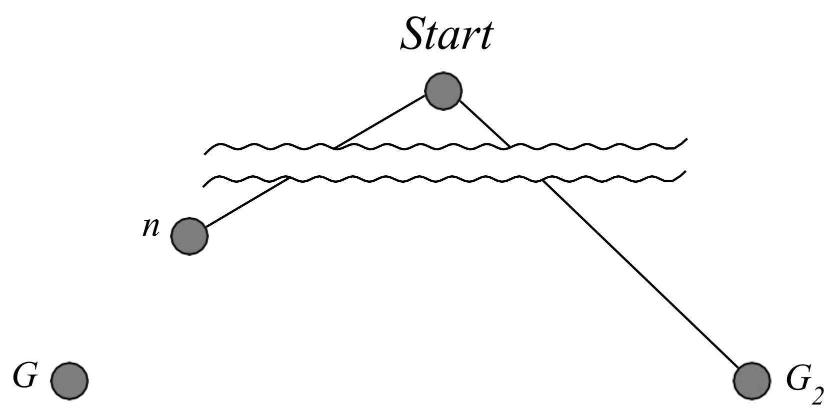


A* search example



Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



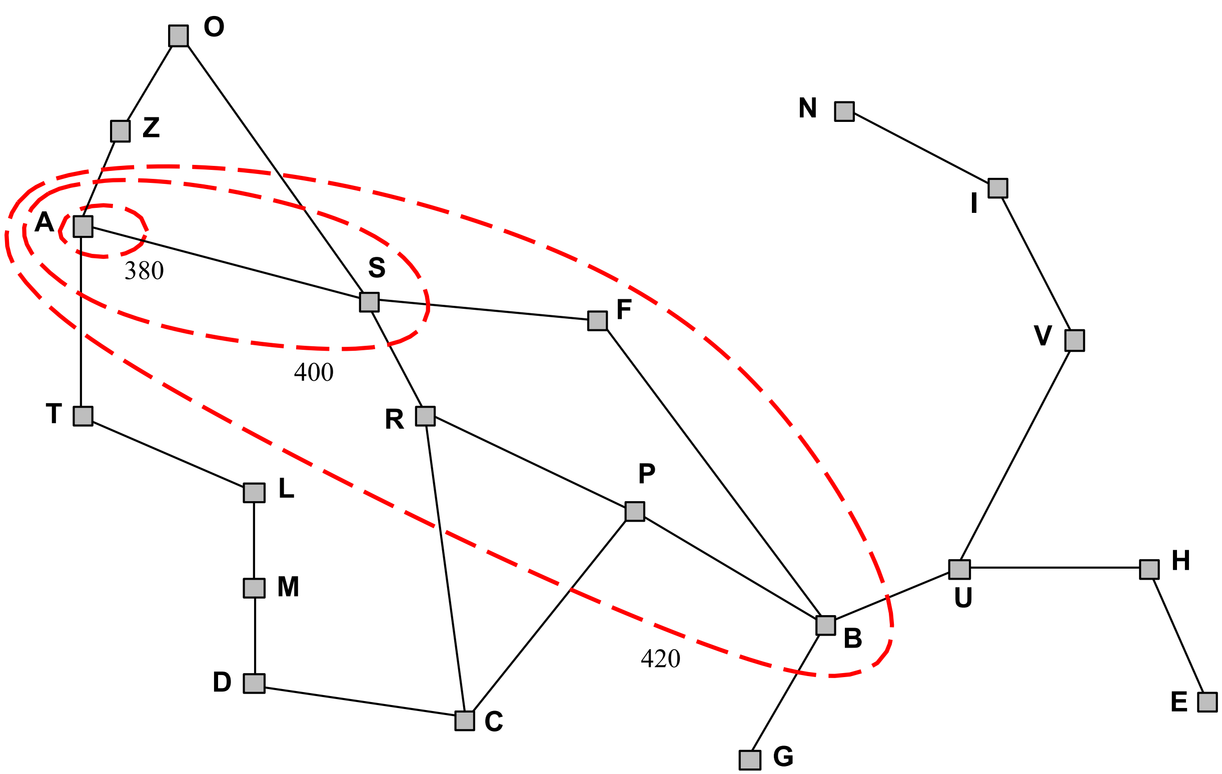
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value*

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers) Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A*

Complete??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in h × length of soln.] Space??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in h × length of soln.]

Space?? Keeps all nodes in memory

Optimal??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$

A* expands some nodes with $f(n) = C^*$

A* expands no nodes with $f(n) > C^*$

Proof of lemma: Consistency

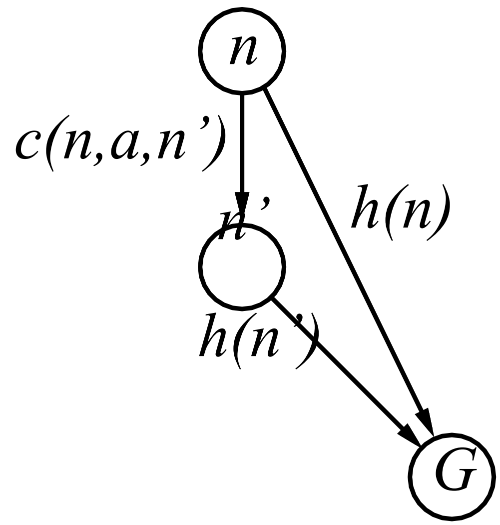
A heuristic is consistent if

$$h(n) \leq c(n, a, n) + h(n)$$

If h is consistent, we have

$$
\begin{aligned}
f(n) = \quad & g(n) + h(n) \\
= \quad & g(n) + c(n, a, n) + h(n) \\
\geq \; & g(n) + h(n) \\
= \quad & f(n)
\end{aligned}
$$

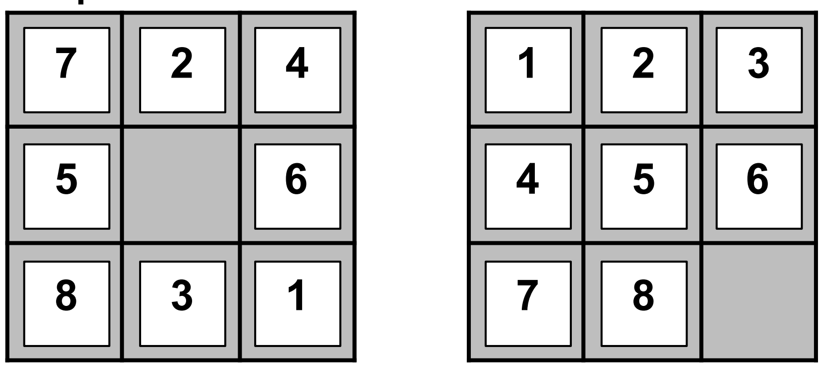
I.e., $f(n)$ is nondecreasing along any path.



Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State Goal State

$h_1(S)$ =??
$h_2(S)$ =??

Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$h_1(S)$ =?? 6
$h_2(S)$ =?? 4+0+3+3+1+0+2+1 = 14

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible), then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes
A*(h_1) = 539 nodes
A*(h_2) = 113 nodes

$d = 24$ IDS ≈ 54,000,000,000 nodes
A*(h_1) = 39,135 nodes
A*(h_2) = 1,641 nodes

Given any admissible heuristics h_a, h_b,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

Relaxed problems

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

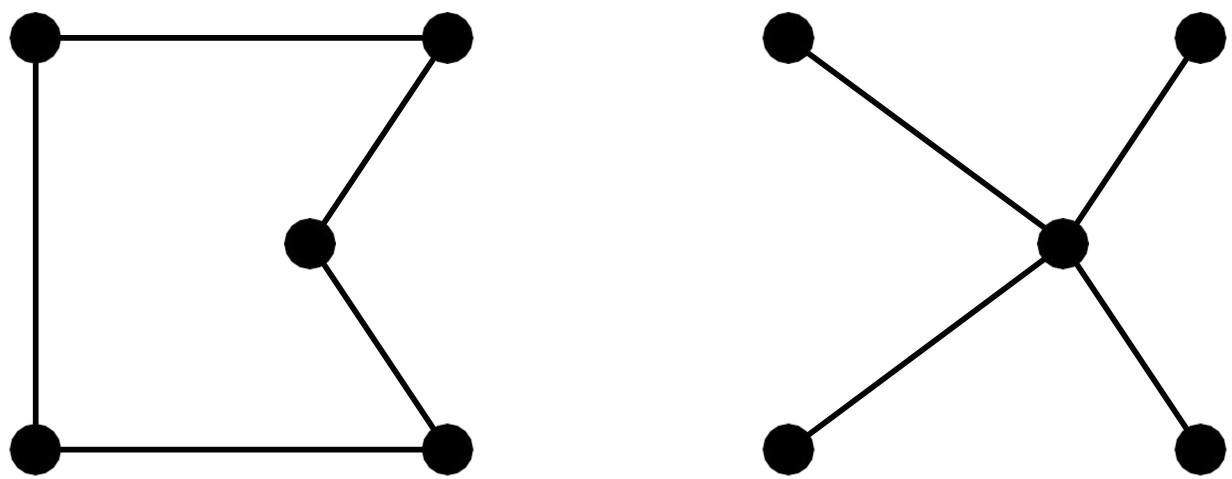
If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP) Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $O(n^2)$ and is a lower bound on the shortest (open) tour

Summary

A problem consists of five parts: the **initial state**, a set of **actions**, a **transition model** describing the results of those actions, a set of **goal states**, and an **action cost function**.

Uninformed search methods have access only to the **problem definition**. Algorithms build a search tree in an attempt to find a solution.

Informed search methods have access to a **heuristic** function h(n) that estimates the cost of a solution from n.