Search Algorithms I

2

2 Search Algorithms

2.1 Problem-solving agents

2.2 Basic search algorithms

- Best-first search
- Blind search

2.3 Heuristic search

- Greedy search
- A* search
- Recursive best-first search
- Beam search$^\#$

Problem-solving agents

Problem-solving agents: finding sequences of actions that lead to desirable states (goal-based)

State: some description of the current world states
- **abstracted** for problem solving as state space

Goal: a set of world states

Action: transition between world states

Search: the algorithm takes a problem as input and returns a solution in the form of an action sequence

The agent can execute the actions in the solution $\Leftarrow$ symbolism

How to solve problems??

The algorithm described in natural language $\Rightarrow$ Pseudocode

Problem-solving agents

```
def SIMPLE-PROBLEM-SOLVING-AGENT( problem)
        s, an action sequence, initially empty
        state, some description of the current world state
        g, a goal, initially null
        problem, a problem formulation
    state ← UPDATE-STATE(state, problem)
    if s is empty then
        g ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, g)
        s ← SEARCH( problem)
        if s=failure then return a null action
    action ← FIRST(s, state)
    s ← REST(s, state)
    return an action
```

Note: offline vs. online problem solving

Example: Romania

On holiday in Romania, currently in Arad
Flight leaves tomorrow from Bucharest

Formulate goal

be in Bucharest

Formulate problem

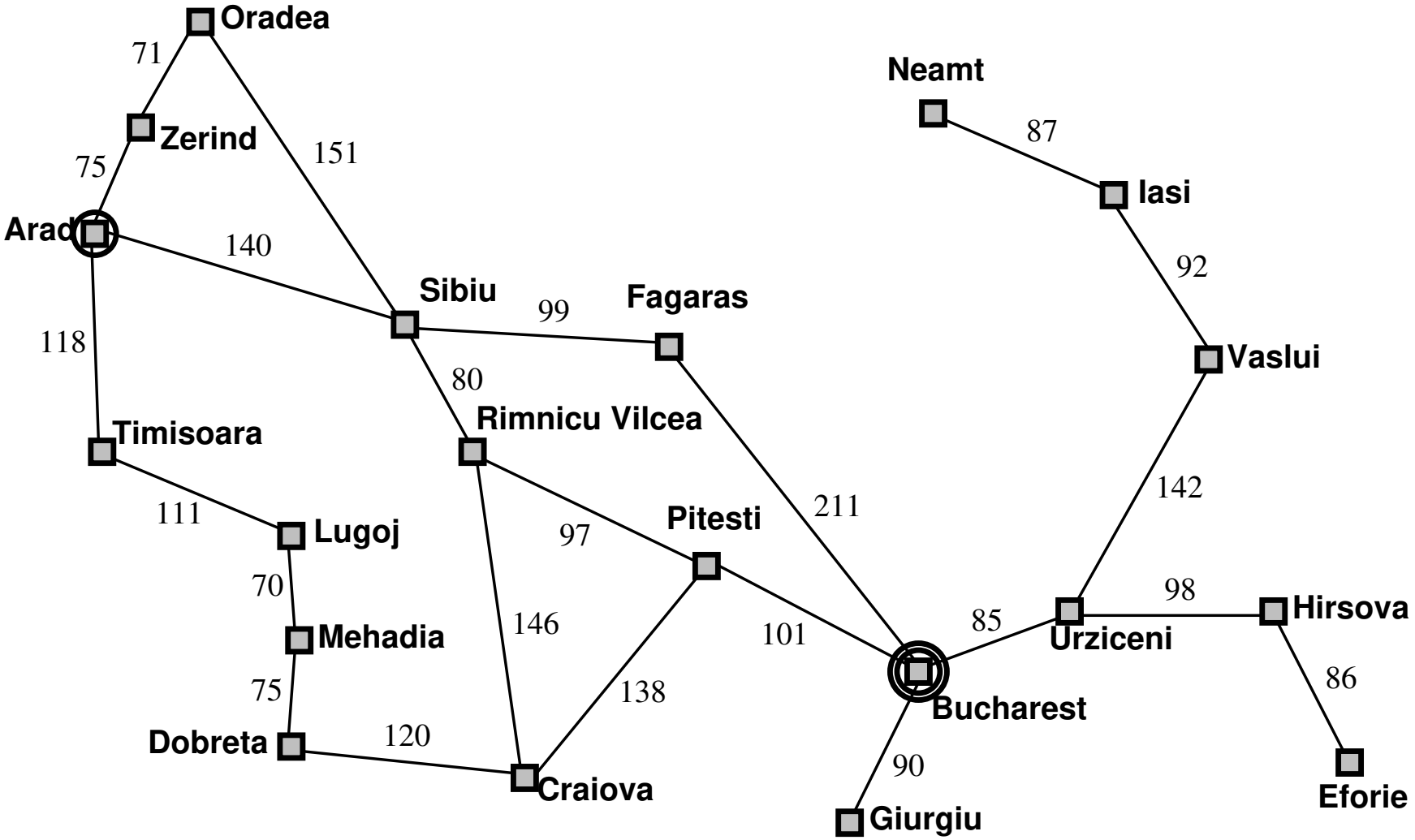
states: various cities

actions: drive between cities

Find solution

the sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem types

Deterministic, fully observable $\Longrightarrow$ single-state problem

The agent knows exactly which state it will be in; the solution is a sequence

Non-observable $\Longrightarrow$ conformant problem

The agent may have no idea where it is; the solution (if any) is a sequence

Nondeterministic and/or partially observable $\Longrightarrow$ contingency problem

percepts provide **new** information about current state

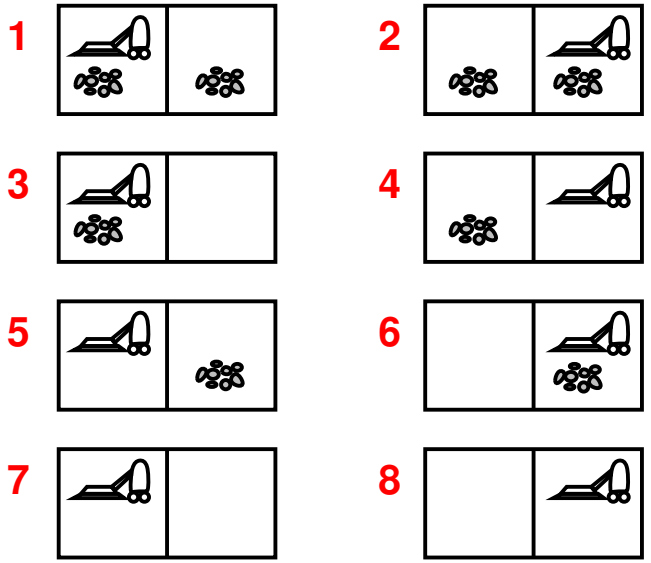
solution is a contingent plan or a policy

often **interleave** search, execution

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

Example: vacuum world

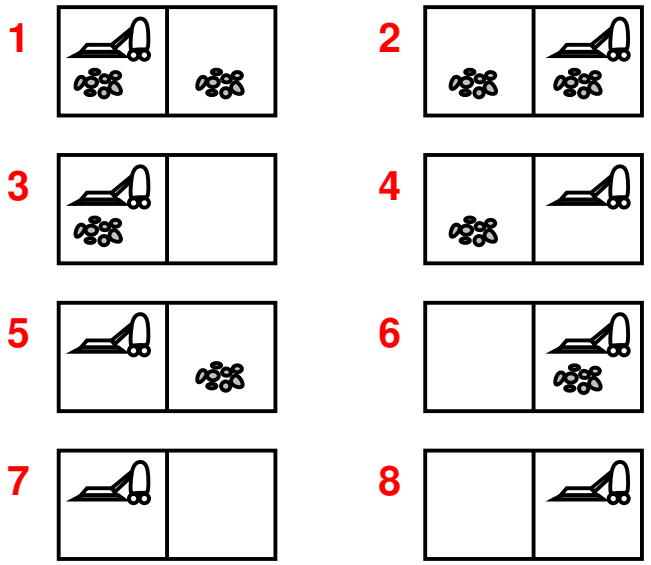
Single-state, start in #5. Solution??



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??

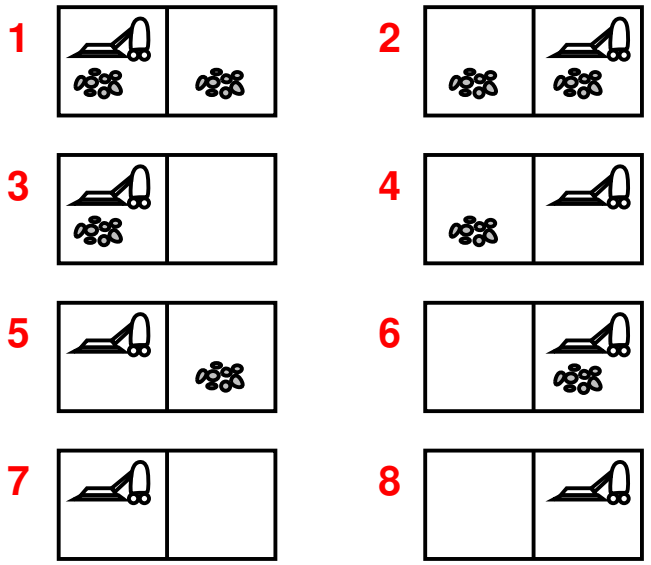


Example: vacuum world

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??

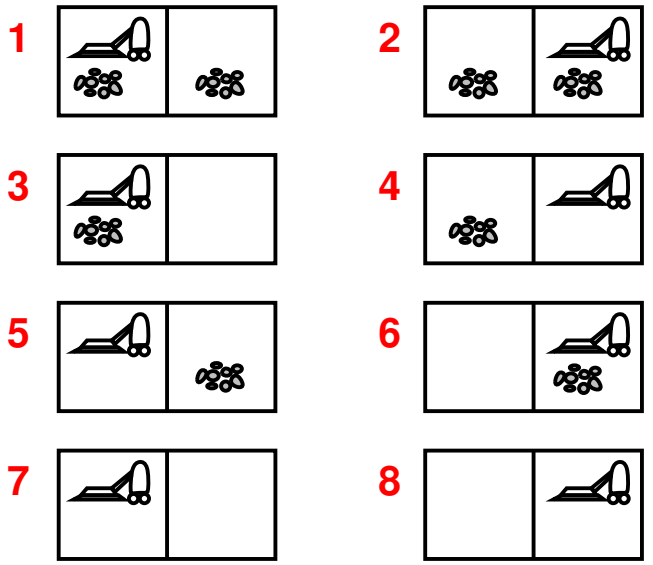


Example: vacuum world

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



Problem formulation

A problem is defined formally by five components

- initial state that the agent starts
  - any state $s \in S$ (set of states), the initial state $S_0 \in S$
    e.g., $In(Arad)$ ("at Arad")

- actions: given a state s, $\textsc{Action}(s)$ returns the set of actions that can be executed in s
  e.g., from the state $In(Arad)$, the applicable actions are
  $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$

- transition model: a function $\textsc{Result}(s, a)$ (or $\textsc{Do}(a, s)$) that returns the state that results from doing action a in the state s;
  - also use the term *successor* to refer to any state reachable from a given state by a single action
  e.g., $\textsc{Result}(In(Arad), Go(Zerind)) = In(Zerind)$

Problem formulation

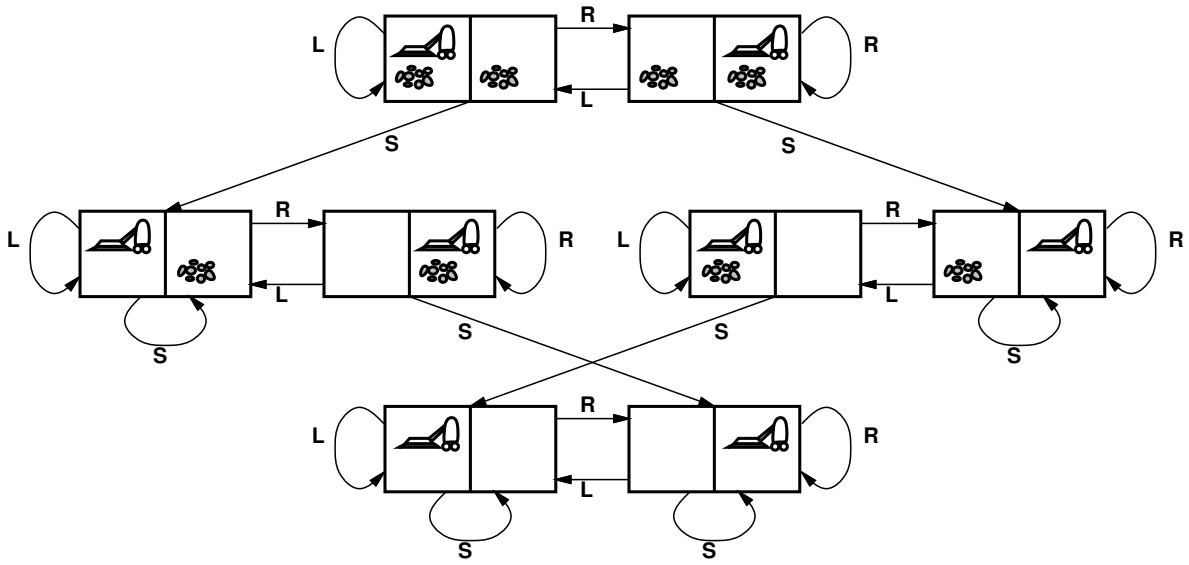
- goal test, can be
  - explicit, e.g., $x = In(Bucharest)$
  - implicit, e.g., $NoDirt(x)$

- cost (action or path): a function that assigns a numeric cost to each path (of actions)
  - e.g., the 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
- $[a_1, a_2, \cdots, a_n]$

leading from the initial state to a goal state

Example: vacuum world state space graph



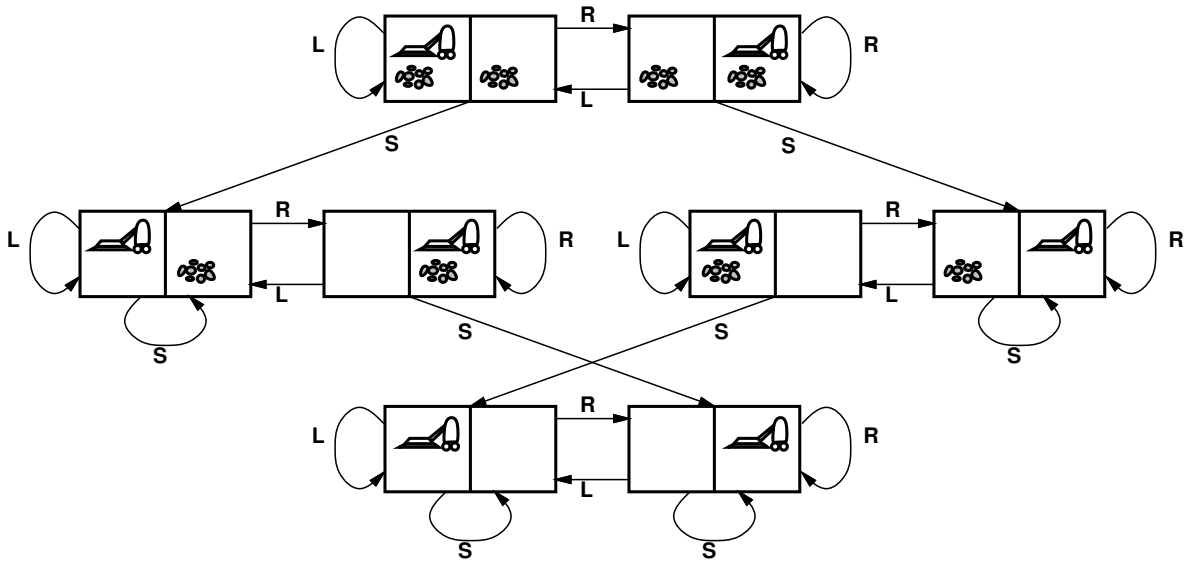
states??

actions??

goal??

cost??

Example: vacuum world state space graph



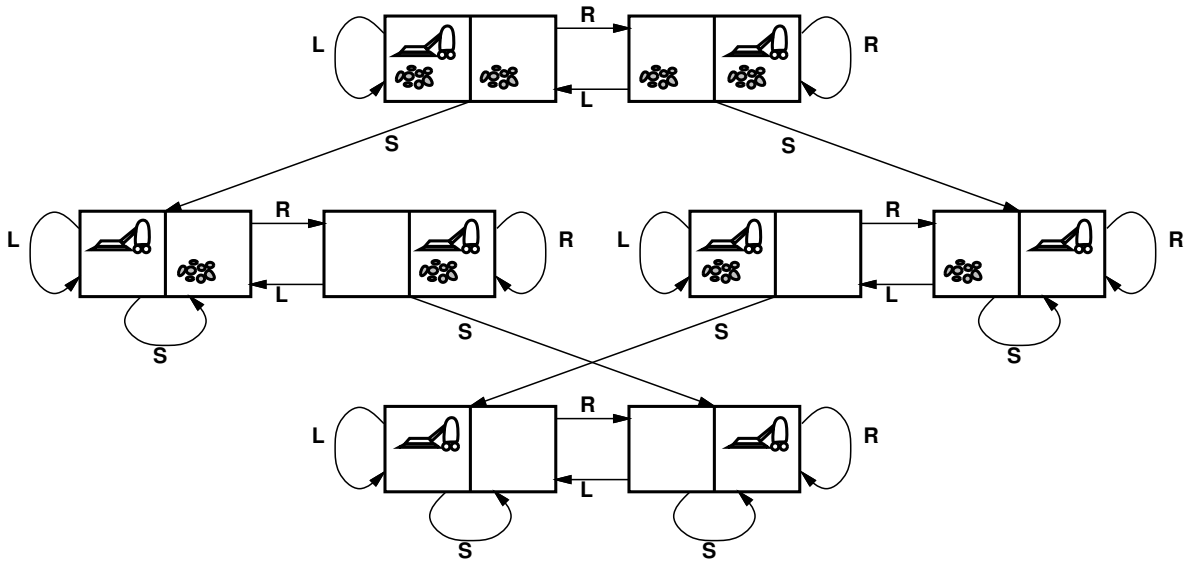
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??

goal??

cost??

Example: vacuum world state space graph



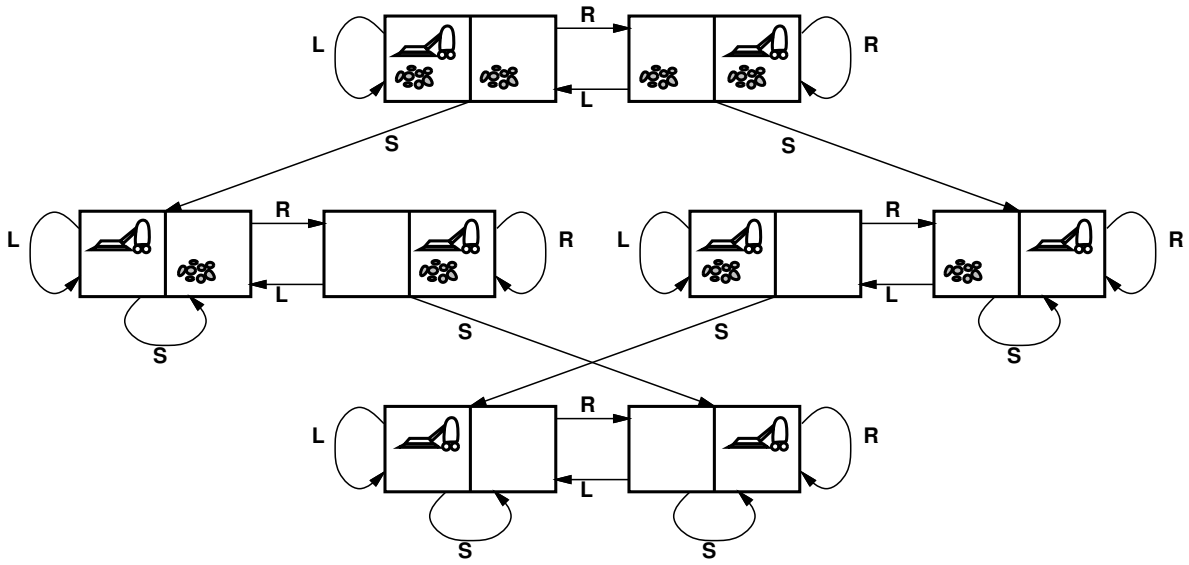
states??: integer dirt and robot locations (ignore dirt amounts etc.)

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

goal??

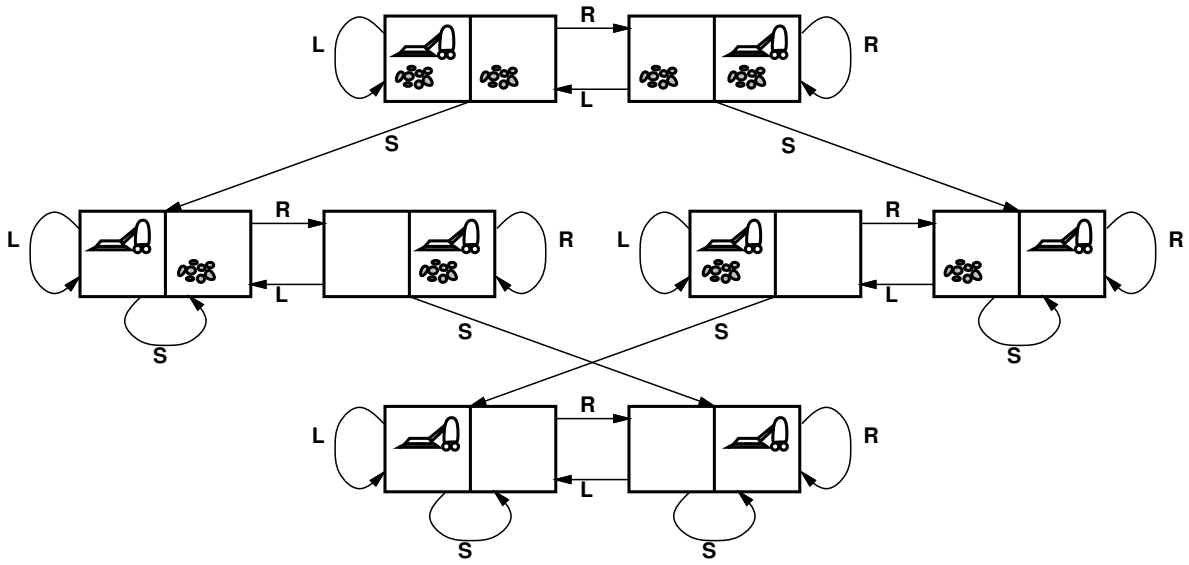
cost??

Example: vacuum world state space graph



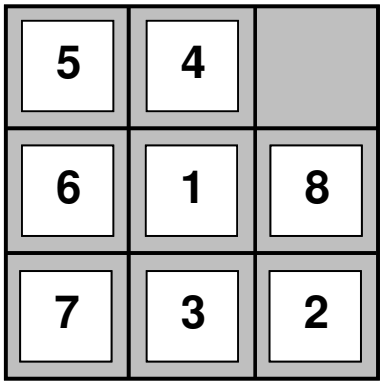
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal??: no dirt
cost??

Example: vacuum world state space graph

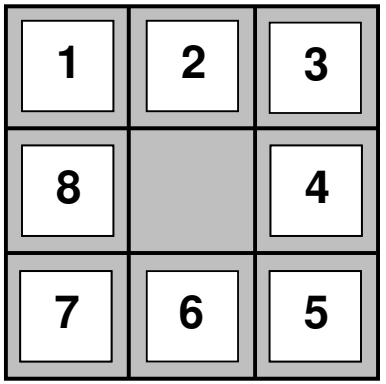


states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal??: no dirt
cost??: 1 per action (0 for $NoOp$)

Example: the 8-puzzle



Start State



Goal State

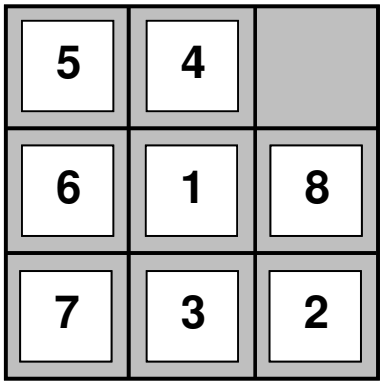
states??

actions??

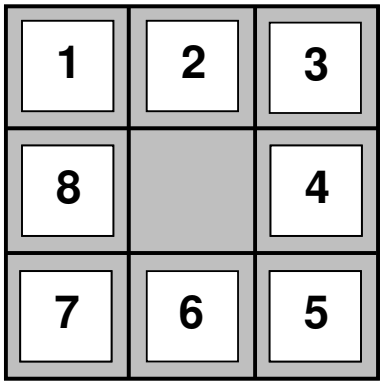
goal??

cost??

Example: the 8-puzzle



Start State



Goal State

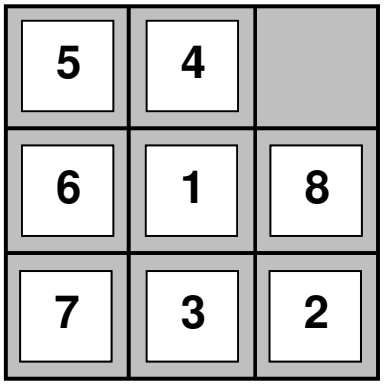
states??: integer locations of tiles (ignore intermediate positions)

actions??

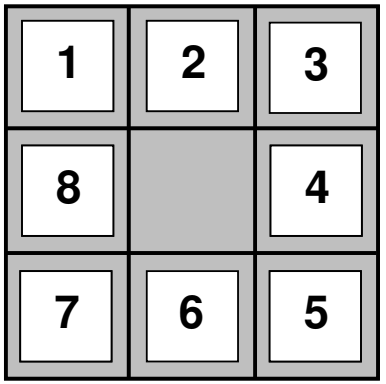
goal??

cost??

Example: the 8-puzzle



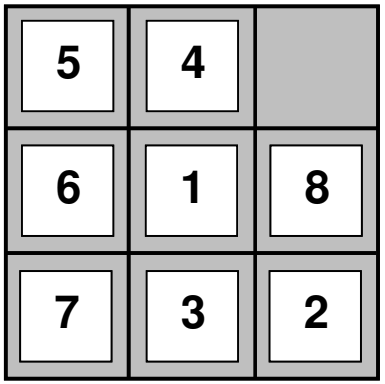
Start State



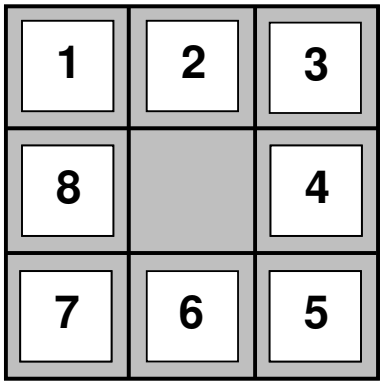
Goal State

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

Example: the 8-puzzle



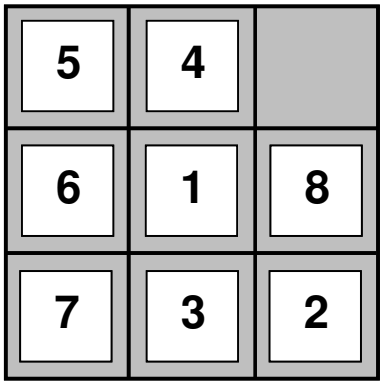
Start State



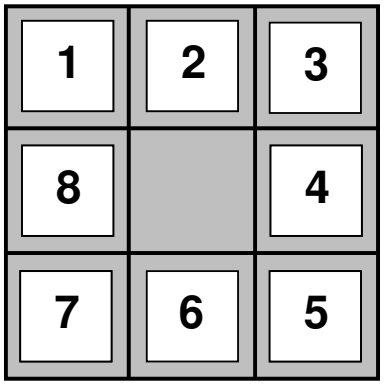
Goal State

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

Example: the 8-puzzle



Start State

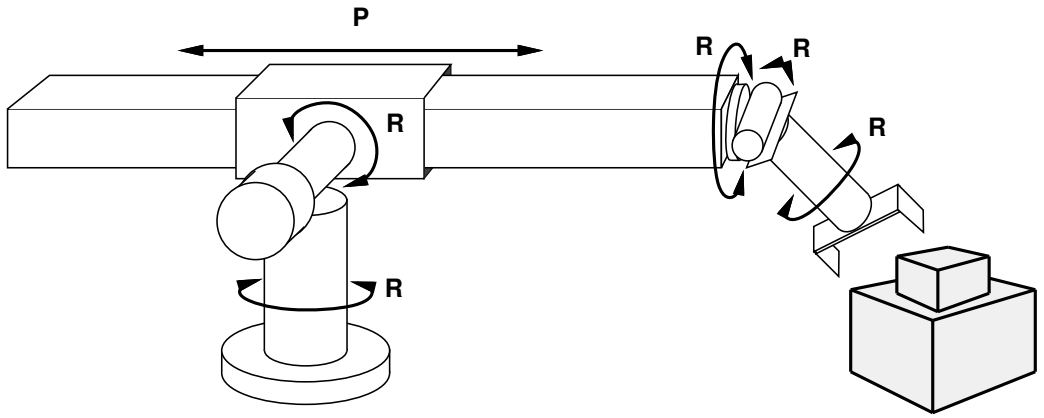


Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal??: $=$ goal state (given)
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??: complete assembly **with no robot included**

cost??: time to execute

Basic search algorithms

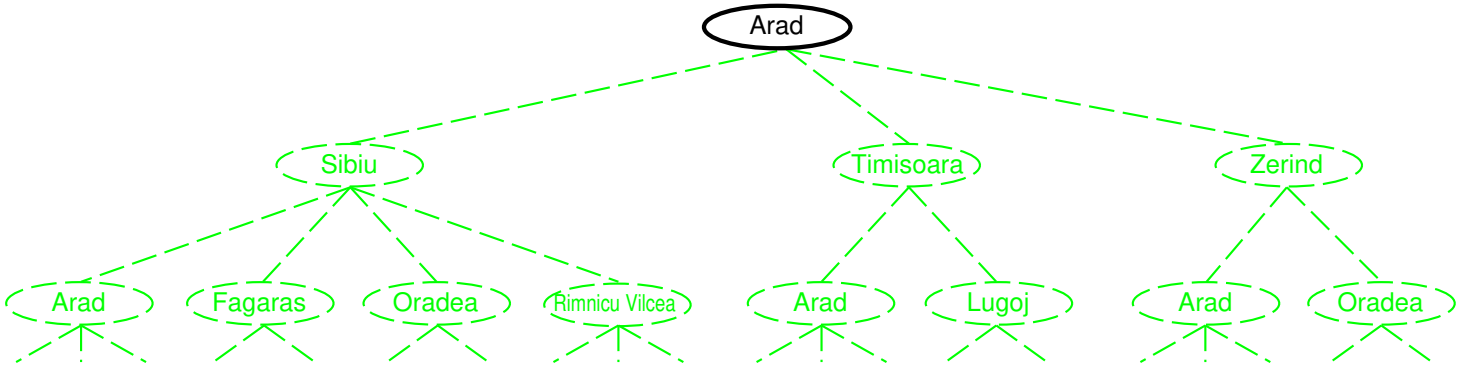
Simulated (offline) exploration of state space of a tree (TREE-SEARCH)
by generating successors of already-explored states

```
def TREE-SEARCH( problem)
    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 by certain strategy
        if the node contains a goal state then return the corresponding solution
        expand the node and add the resulting nodes to the search tree
```

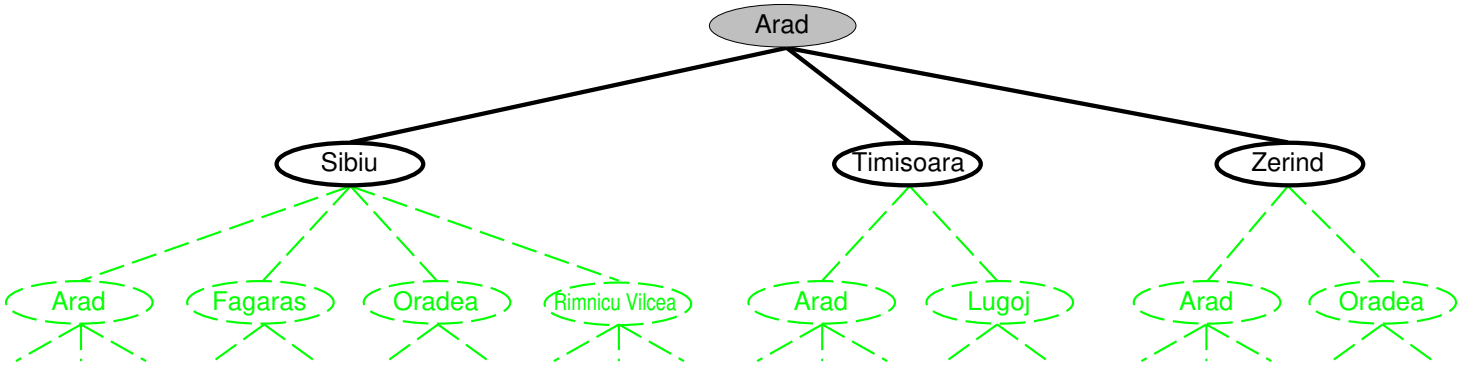
Frontier: all the leaf nodes available for expansion at moment, which separates two regions of the state-space graph (tree)

– an interior region where every state has been expanded
– an exterior region of states that have not yet been reached

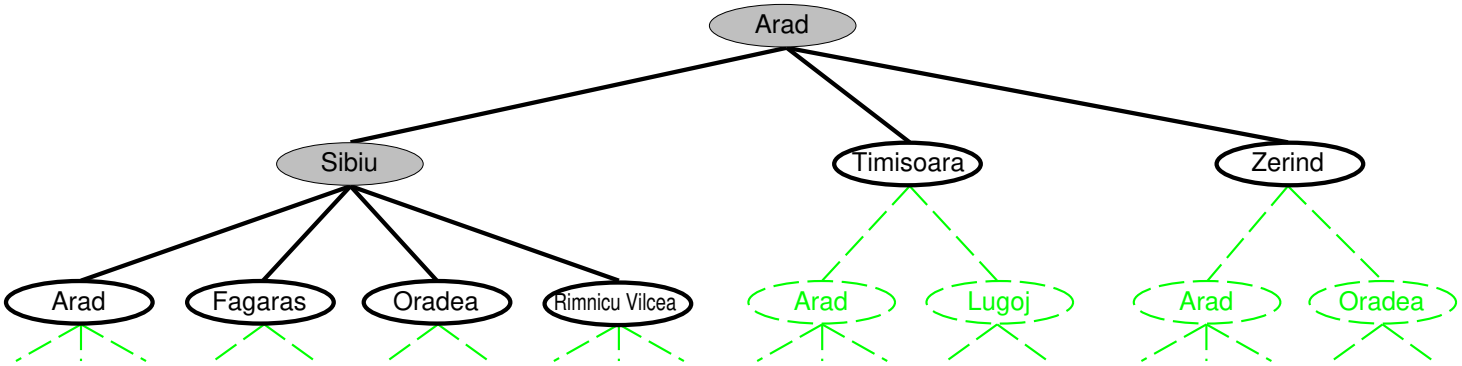
Example: tree search



Example: tree search



Example: tree search



Note: loopy path (repeated state) in the leftmost

To avoid exploring redundant paths by using a data structure

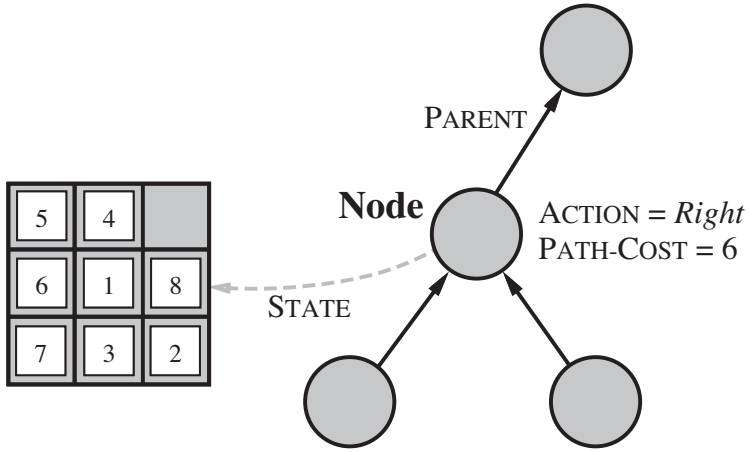
reached set

– remembering every expanded node

— newly generated nodes in the reached set can be discarded

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



Dot notation (.) for data structures and methods (more self-explained)

- n.STATE: state (in the state space) corresponds to the node
- n.PARENT: node (in the search tree that generated this node)
- n.ACTION: action applied to the parent to generate the node
- n.COST: cost, $g(x)$, of path from initial state to n

Implementation: states vs. nodes

Queue: a data structure to store the frontier, with the operations

- Is-Empty(*frontier*) returns true only if there are no nodes in the frontier
- Pop(*frontier*) removes the top node from the frontier and returns it
- Top(*frontier*) returns (but does not remove) the top node of the frontier
- Add(*node*, *frontier*) inserts node into its proper place in the queue

More self-explained

Priority queue first pops the node with the minimum cost

FIFO (first-in-first-out) queue first pops the node that was added to the queue first

LIFO (last-in-first-out) queue (a.k.a stack) pops first the most recently added node

Best-first search

Search strategy: **refinement** (pseudocode) for the frontier: choosing a node n with minimum value of some evaluation function $f(n)$

```
def BEST-FIRST-SEARCH(problem, f)
    note ← NOTES(problem.INITIAL)
    frontier ← a queue ordered by f, with note as an element  //not defined yet
    reached ← a lookup table, with one entry with key problem.INITIAL and value note

    while not IS-EMPTY(frontier) do
      note ← POP(frontier)
      if problem.IS-GOAL(node.STATE) then return node
      for each child in EXPAND(problem,node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST<reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
    return failure
    def EXPAND(problem,node) yield nodes
```

Best-first search

```
def EXPAND(problem,node) yield nodes
  s ← node.STATE
  for each action in problem.ACTION(s)
    s′ ← problem.RESULT(s,action) do
    cost ← node.PATH-COST+problem.ACTION-COST(s,action,s′)
    yield NODE(STATE=s′,PARENT=node,ACTION=action,PATH-COST=cost)
```

Note: EXPAND will be used in later

Search strategies

A strategy is defined by picking the **order of node expansion**

- Uninformed (blind) search strategies
- Informed (heuristic) search strategies

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 (maybe ∞)

Blind search

Blind (Uninformed) strategies use only the information available in the problem definition

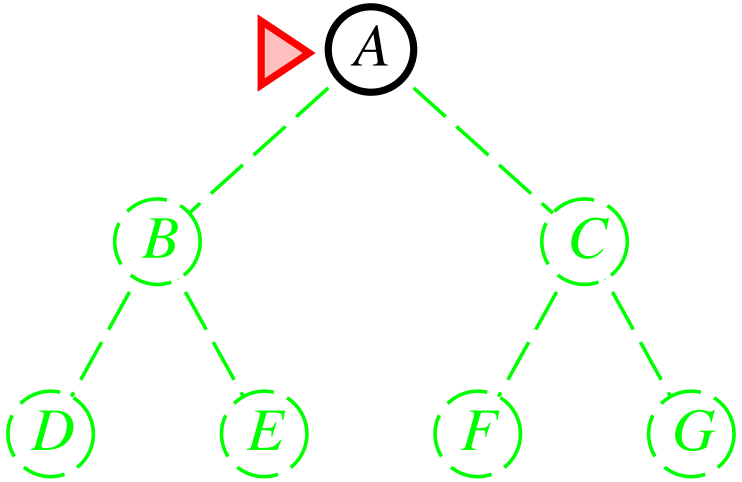
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

BFS: Expand shallowest unexpanded node

Implementation

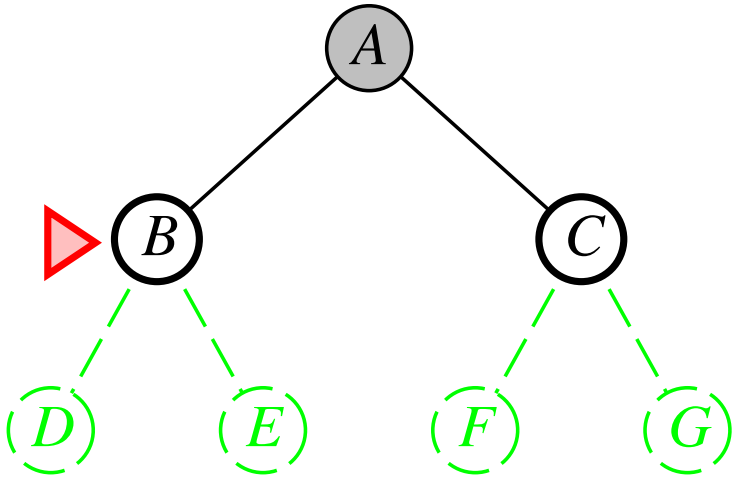
frontier = FIFO



Breadth-first search

Expand the shallowest unexpanded node

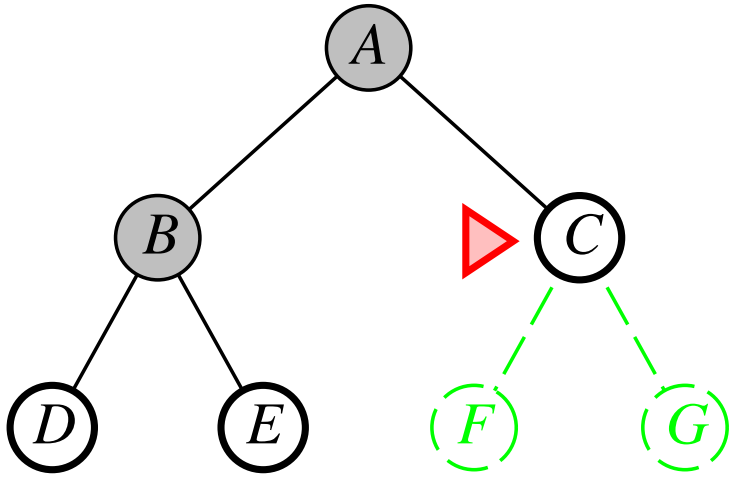
Implementation



Breadth-first search

Expand the shallowest unexpanded node

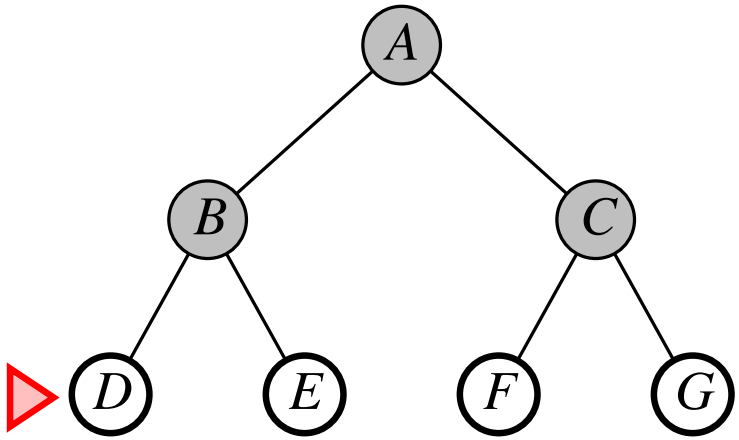
Implementation



Breadth-first search

Expand the shallowest unexpanded node

Implementation



Breadth-first search

```
def BREADTH-FIRST-SEARCH( problem)
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node    // solution
    frontier ← a FIFO queue with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
      node ← POP(frontier)
      for each child in EXPAND(problem,node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then return child    // solution
        if s is not in reached then
          add s to reached
          add child to frontier
    return failure
```

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
(the shallowest goal node is not necessarily 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

$O(b^d)$: $d = 16$, $b = 10$, 1 million nodes/second, 1K bytes/node
Time — 350 years
Space — 10 exabytes (1EB=10^9GB)
– Space is the big problem; can easily generate nodes at 100MB/sec, so 24hrs = 8640GB

Python: search classes*

```
"""
Search

Create classes of problems and problem instances, and solve them with calls to the various search functions.
"""

import sys
from collections import deque
from utils import *

class Problem:
    def __init__(self, initial, goal=None):
        self.initial = initial
        self.goal = goal
    def actions(self, state):
        raise NotCodedError
    def result(self, state, action):
        raise NotCodedError
    def is_goal(self, state):
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal
    def path_cost(self, c, state1, action, state2):
        return c + 1
    def value(self, state):
        raise NotCodedError
```

Python: search classes*

```
class Node:
    """A node in a search tree."""
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1
    def __repr__(self):
        return "<Node {}>".format(self.state)
    def __lt__(self, node):
        return self.state < node.state
    def expand(self, problem):
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]
    def child_node(self, problem, action):
        next_state = problem.result(self.state, action)
        next_node = Node(next_state, self, action, problem.path_cost(self.path_cost, self.state, action, next_state))
        return next_node
    def solution(self):
        return [node.action for node in self.path()[1:]]
```

Python: search classes*

```
# class Node continued
def path(self):
    node, path_back = self, []
    while node:
        path_back.append(node)
        node = node.parent
    return list(reversed(path_back))
def __eq__(self, other):
    return isinstance(other, Node) and self.state == other.state
def __hash__(self):
    return hash(self.state)
```

Python: breadth-first search*

```
"""
BFS

Implemente the pseudocode by calling function.
"""
def breadth_first_search(problem):
    frontier = deque([Node(problem.initial)])  # FIFO queue
    while frontier:
        node = frontier.popleft()
        if problem.is_goal(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
```

Note: It is helpful to use an AI code tool (say ChatGPT/DeepSeek/Github Copilot) to translate the pseudocodes to Python codes

— Refer to Natural Language Understanding for understanding

ChatGPT: breadth-first search*

```
def breadth_first_search(problem):
    node = Node(problem.Initial)

    if problem.Is_Goal(node.State):
        return node  # Solution found

    frontier = Queue()  # FIFO queue with node as an element
    frontier.put(node)
    reached = set([problem.Initial])

    while not frontier.empty():
        node = frontier.get()

        for child in problem.Expand(node):
            s = child.State

            if problem.Is_Goal(s):
                return child  # Solution found

            if s not in reached:
                reached.add(s)
                frontier.put(child)

    return "failure"
```

ChatGPT: breadth-first search*

ChatGPT notes: You will need to define the Node class and its attributes based on your problem's requirements

- problem.Initial: This refers to the initial state of your problem. Replace it with the appropriate call to retrieve the initial state
- problem.Is_Goal(state): This is a function that checks if a given state is the goal state of your problem. Replace it with the implementation specific to your problem
- problem.Expand(node): This function generates the child nodes of a given node in your problem. Modify it according to your problem's requirements

Please ensure that you have the appropriate imports, such as Queue from the queue module, and that you define the necessary classes and functions for your specific problem before running this code

Uniform-cost search

UCS (aka Dijkstra's algorithm): Expand the least-cost unexpanded node

Implementation

$frontier$ = 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 $g(n)$

Uniform-cost search

$O(b^{\lceil C^*/\epsilon \rceil})$

– can be much greater than $O(b^d)$
(explore large trees involving large perhaps useful steps)

– all step costs are equal, $O(b^{\lceil C^*/\epsilon \rceil})$ is just $O(b^{d+1})$

UCS is similar to BFS

– except that BFS stops as soon as it generates a goal
whereas UCS examines all the nodes at the goal's depth
to see if one has a lower cost

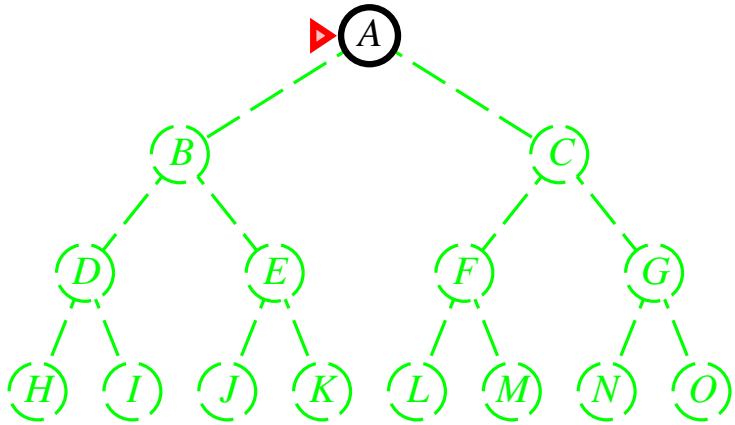
strictly more work by expanding nodes at depth d unnecessarily

Depth-first search

DFS: Expand deepest unexpanded node

Implementation

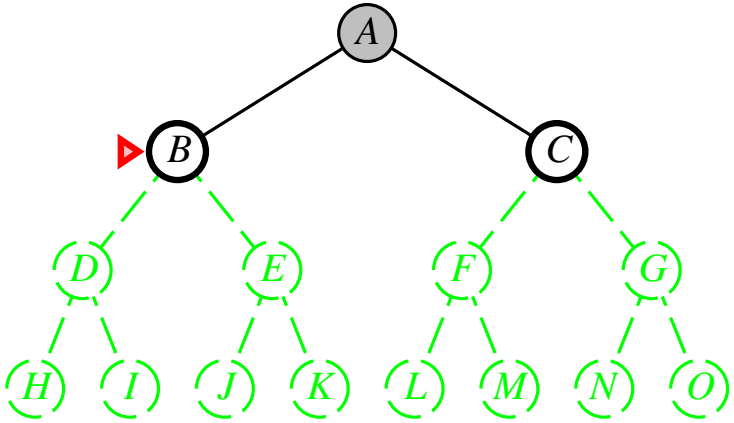
$frontier$ = LIFO



Depth-first search

Expand the deepest unexpanded node

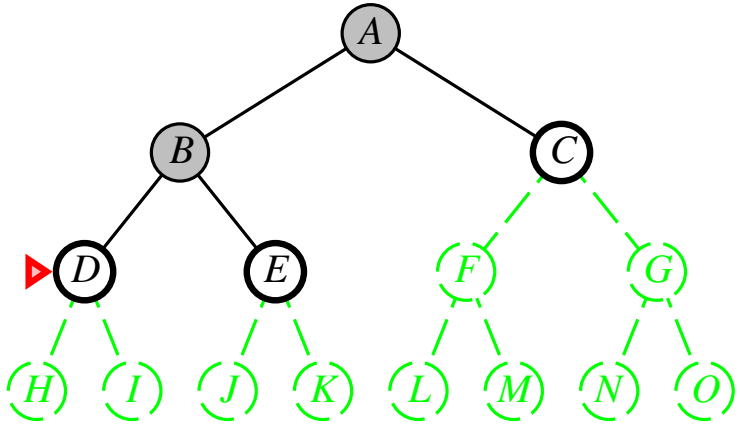
Implementation



Depth-first search

Expand the deepest unexpanded node

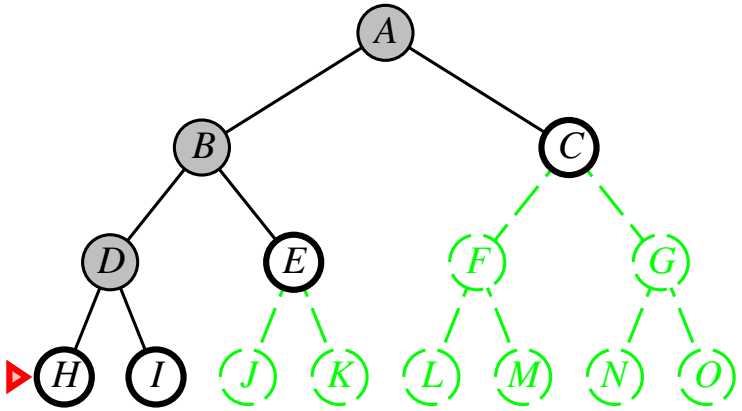
Implementation



Depth-first search

Expand the deepest unexpanded node

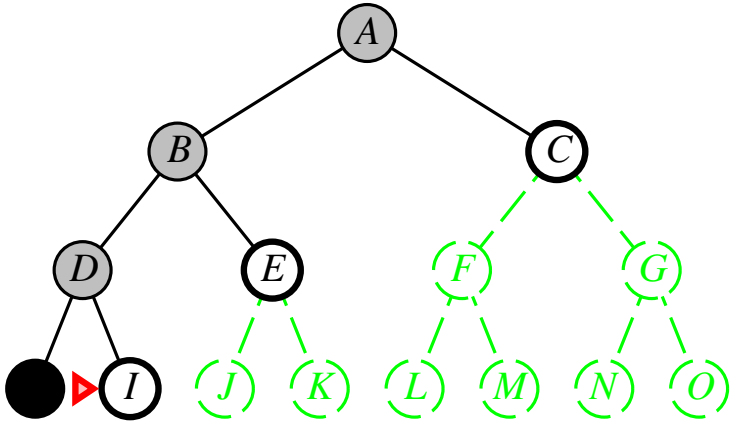
Implementation



Depth-first search

Expand the deepest unexpanded node

Implementation

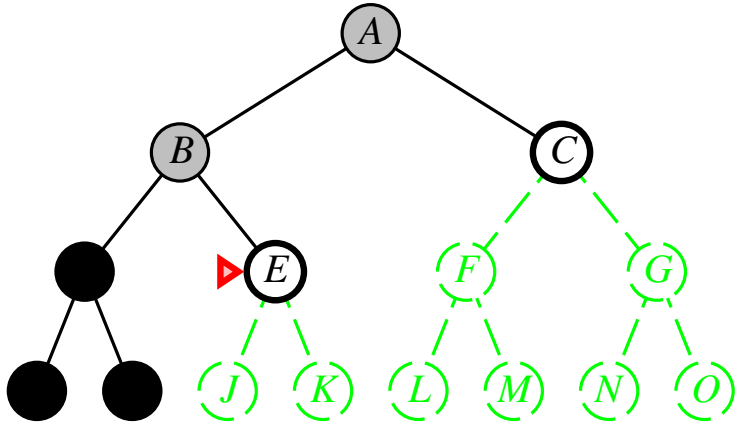


Back-propagation (backtracking)

Depth-first search

Expand the deepest unexpanded node

Implementation

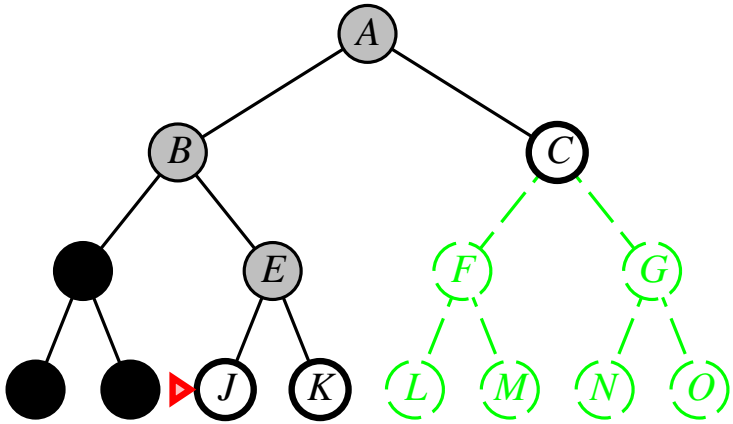


Back-propagation (backtracking)

Depth-first search

Expand the deepest unexpanded node

Implementation

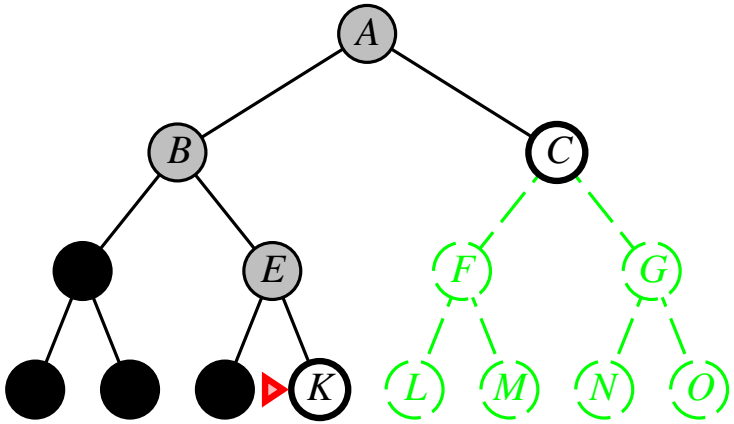


Back-propagation (backtracking)

Depth-first search

Expand the deepest unexpanded node

Implementation

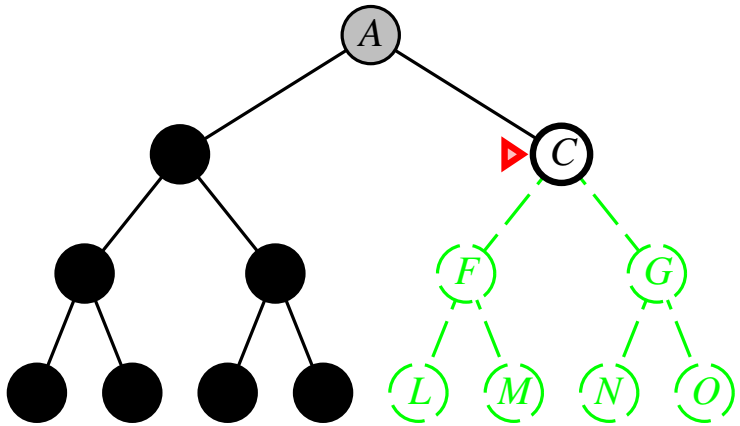


Back-propagation (backtracking)

Depth-first search

Expand the deepest unexpanded node

Implementation

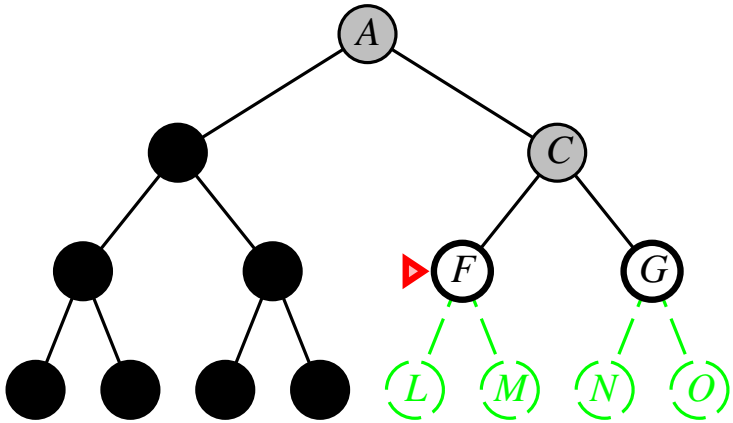


Back-propagation (backtracking)

Depth-first search

Expand the deepest unexpanded node

Implementation

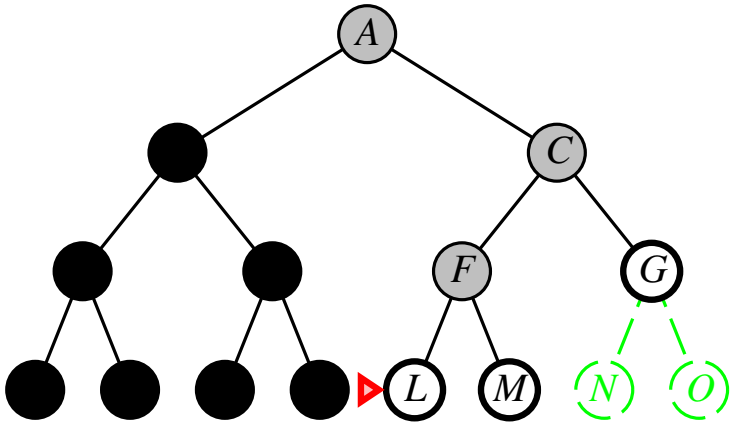


Back-propagation (backtracking)

Depth-first search

Expand the deepest unexpanded node

Implementation

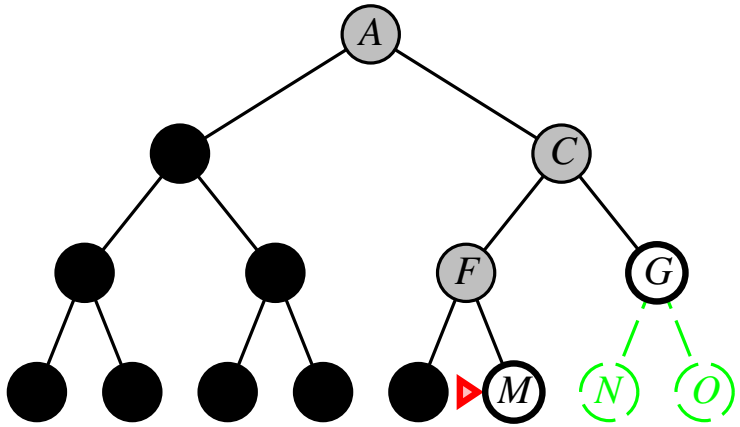


Back-propagation (backtracking)

Depth-first search

Expand the deepest unexpanded node

Implementation



Back-propagation (backtracking)

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

DLS= depth-first search /w depth limit l; *cutoff* no solution within l

Recursive implementation (recursion vs. iteration)

```
def DEPTH-LIMITED-SEARCH( problem, l)
    return RECURSIVE-DLS(NODE(problem.INITIAL), problem, l)

def RECURSIVE-DLS(node, problem, l)
    frontier ← a stack with NODE(problem.INITIAL) as an element
    result ← failure
    if problem.IS-GOAL(node.STATE)  then return node
    if DEPTH(node)>l then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem,node) do
      result ← RECURSIVE-DLS(child, problem, l)
    return result
```

Iterative deepening search

IDS repeatedly applies DLS with increasing limits

```
def ITERATIVE-DEEPENING-SEARCH( problem)
    for depth=0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening search l=0

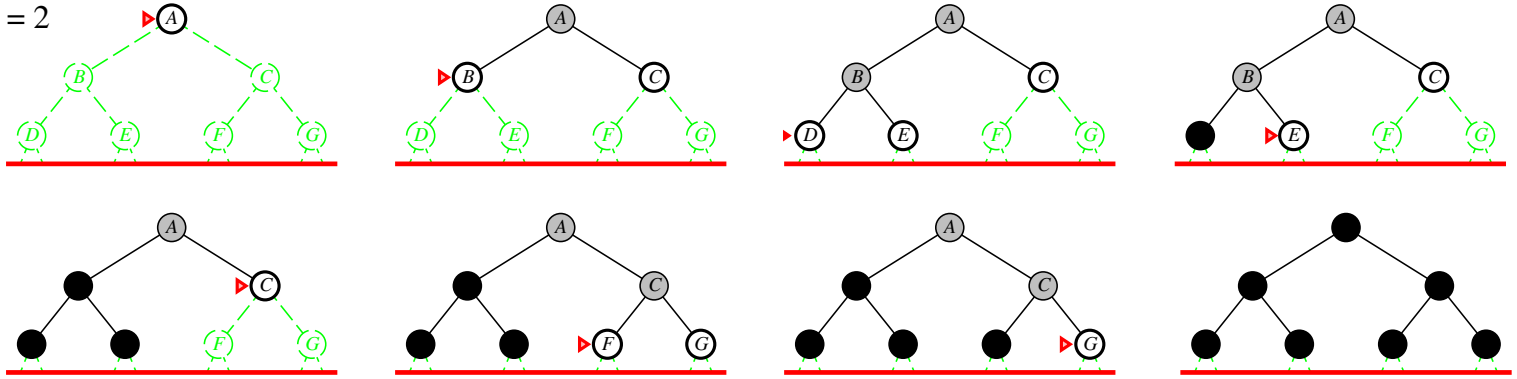
Limit = 0

Iterative deepening search $l = 1$

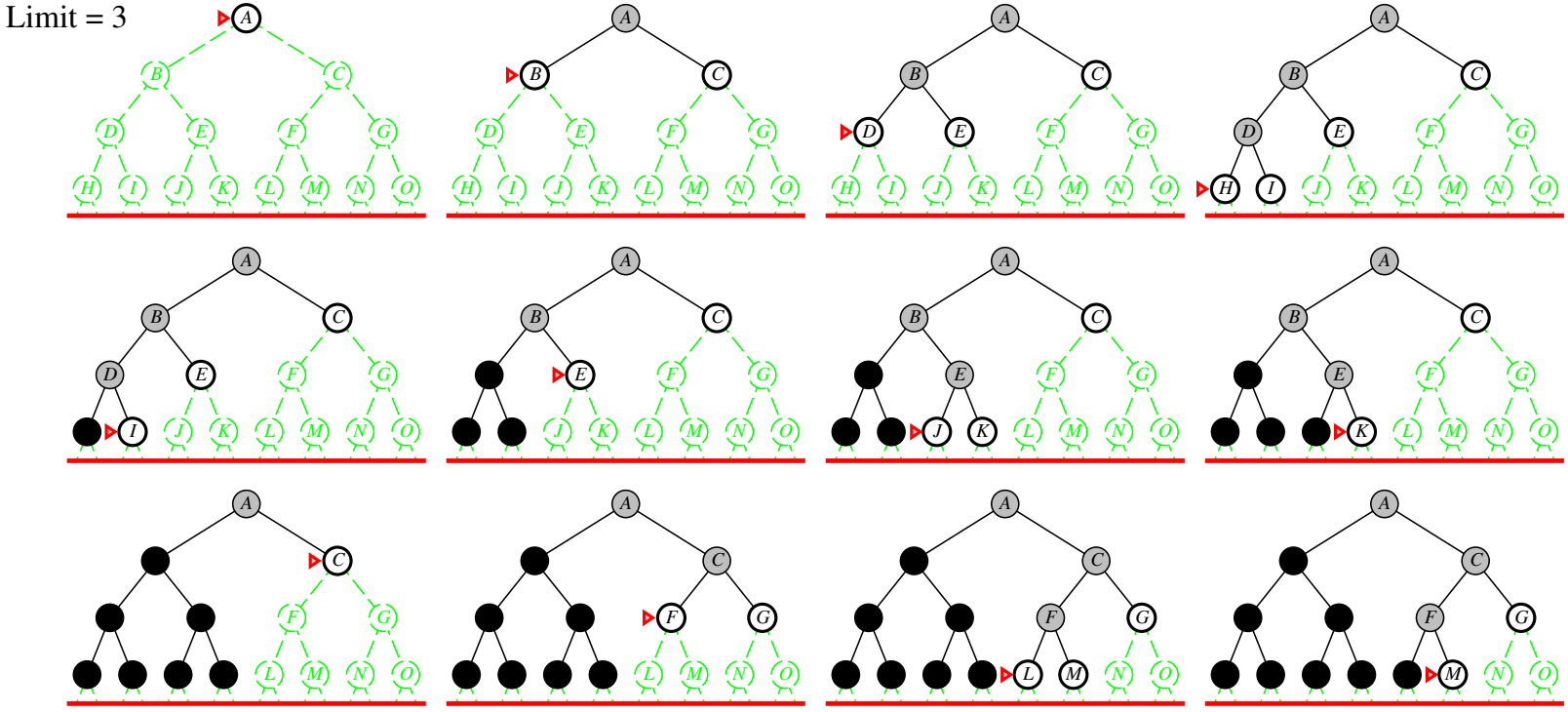
Limit = 1

Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$



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
Comparison for $b = 10$ and $d = 5$, solution at far right leaf

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 11,111,100$$

IDS is asymptotically the same as BFS

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**

Bidirection search*

Idea: run two simultaneous searches – hoping that two searches meet in the middle

– one forward from the initial state

– another backward from the goal

$2b^{d/2}$ is much less than b^d

Implementation: check to see whether the frontiers of the two searches intersect;

– if they do, a solution has been found

The first solution found may not be optimal; some additional search is required to make there is not another short-cut across the gap

Both-ends-against-the-middle (BEATM) endeavors to combine the best features of top-down and bottom-up designs into one process

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

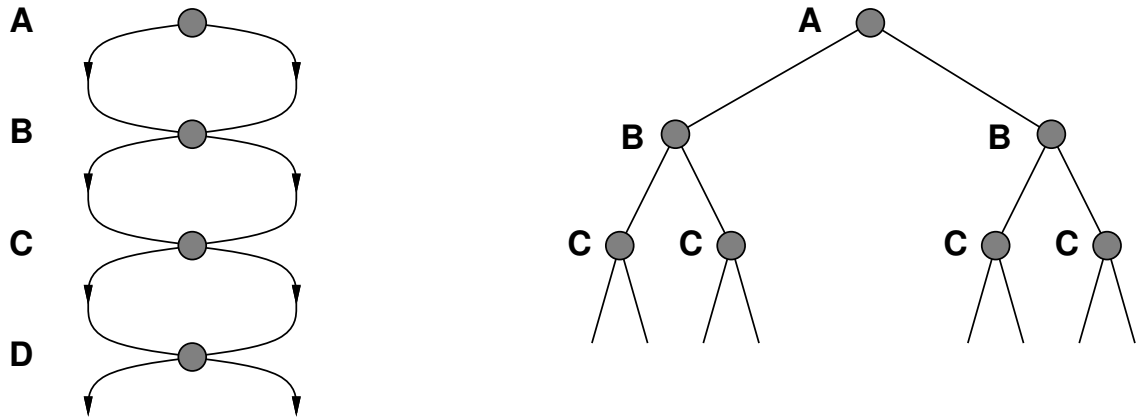
Can a more efficient search algorithm replace exhaustive search??

- Essentially, the P vs. NP question
- Extremely, improvements (say heuristic)

Graph search: repeated states*

Graph $\Rightarrow$ Tree

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



$d+1$ states space $\Rightarrow$ 2^d paths

All the tree-search versions of algorithms can be extended to the graph-search versions by checking the repeated states

Graph search*

```
def GRAPH-SEARCH( problem)
    initialize the frontier using the initial state of problem
    initialize the reached set to empty
    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 node
        add node to the reached set
        expand the chosen node and add the resulting nodes to the frontier
            only if not in the frontier or reached set
```

Note: using **reached set** to avoid exploring redundant paths

Heuristic Search

Informed (heuristic) strategies use problem-specific knowledge
to find solutions more efficiently

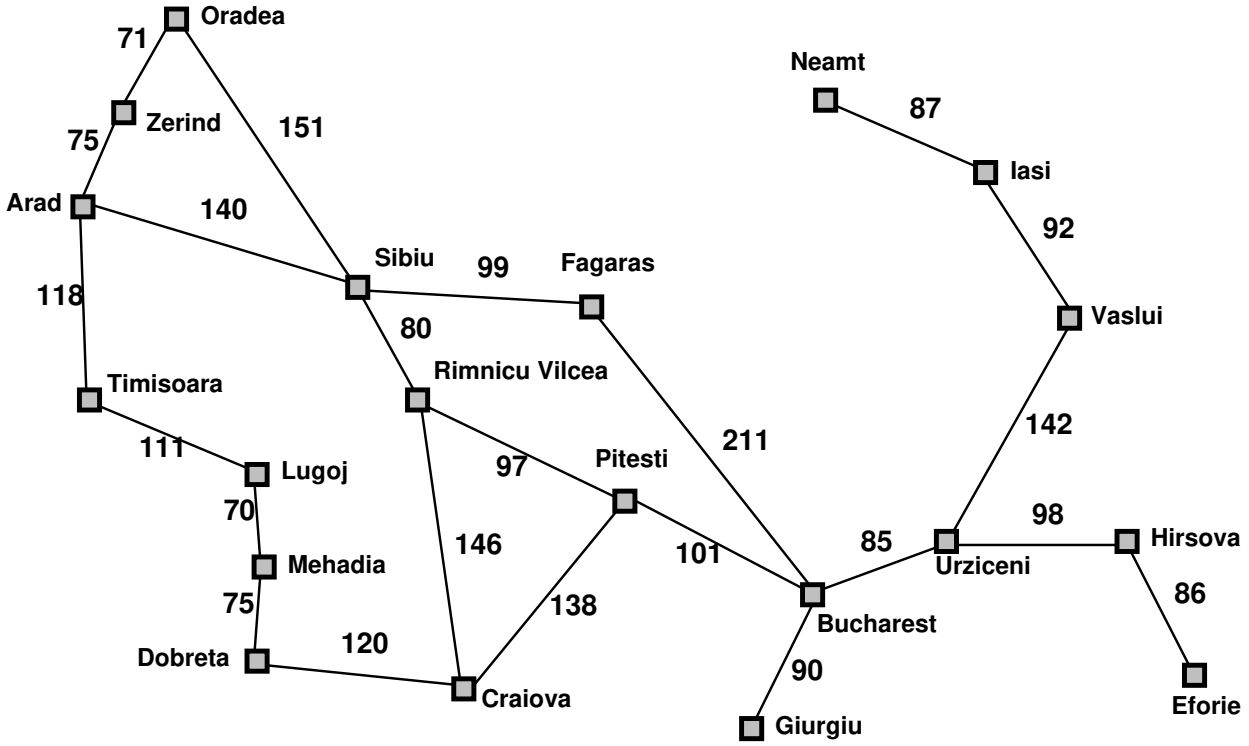
Best-first search: use an **evaluation function** for each node
– f: estimate of "desirability"

$\Rightarrow$ Expand the most desirable unexpanded node

Implementation
QueueingFn = insert successors in decreasing order of desirability

Basic heuristics
- Greedy (best-first) search
- A* search
- Recursive best-first search
- Beam 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

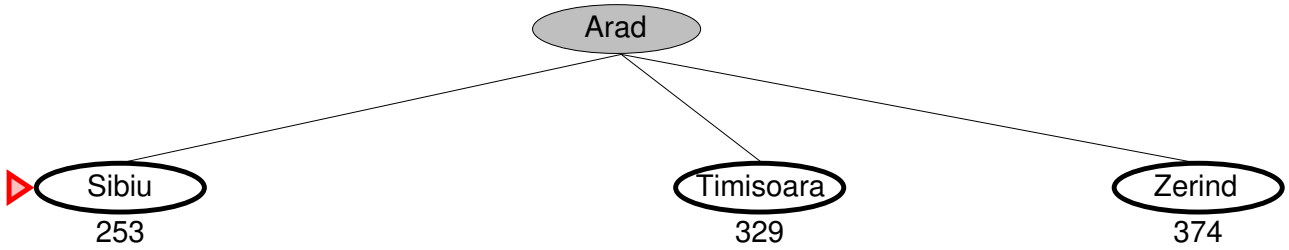
Evaluation function $h(n)$ (**h**euristic)
$=$ estimate of cost from n to the closest goal

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

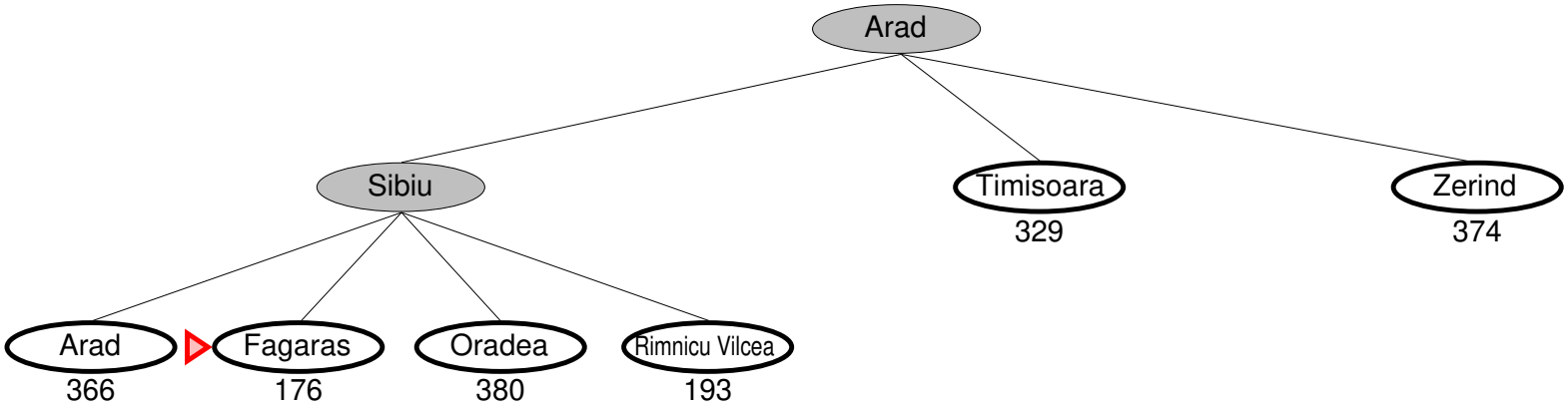
Greedy search expands the node that appears to be closest to goal

Greedy search example

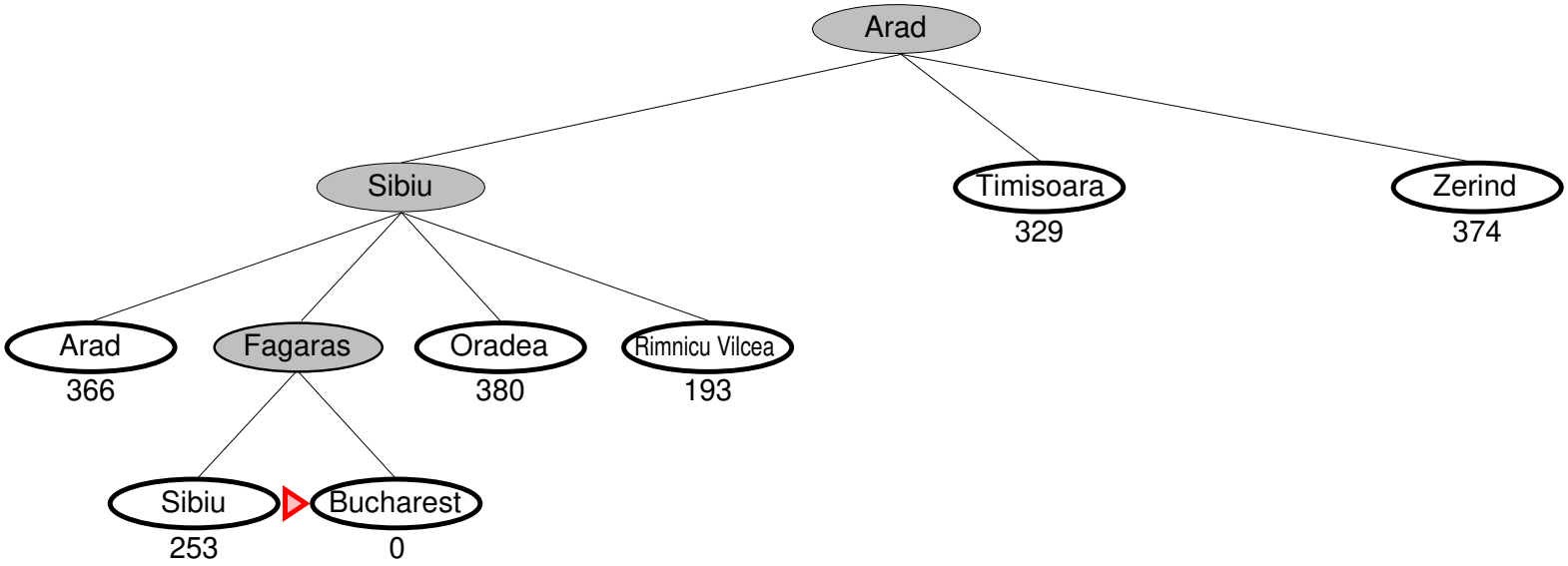
Greedy search example



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 $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$ $(h_{\mathrm{SLD}}(n))$

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No–can get stuck in loops

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??

Properties of greedy search

Complete?? No–can get stuck in loops

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??

Properties of greedy search

Complete?? No–can get stuck in loops

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

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

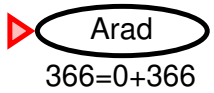
Algorithm: identical to Uniform-Cost-Search except for using $g + h$ instead of g

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)

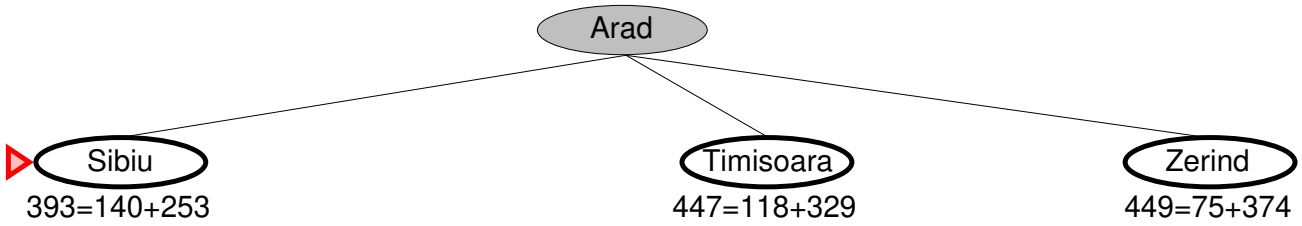
E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

Theorem: A* search is optimal

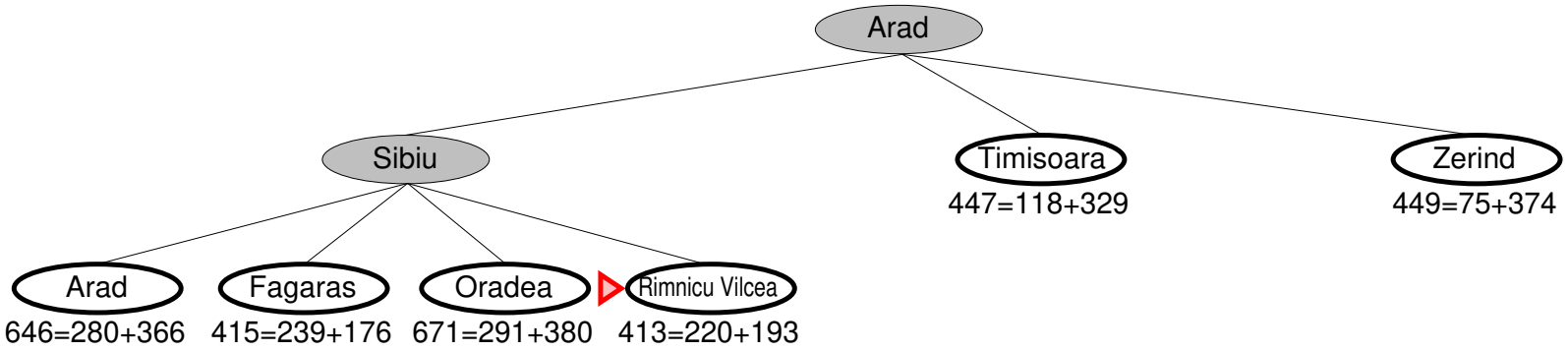
A* search example



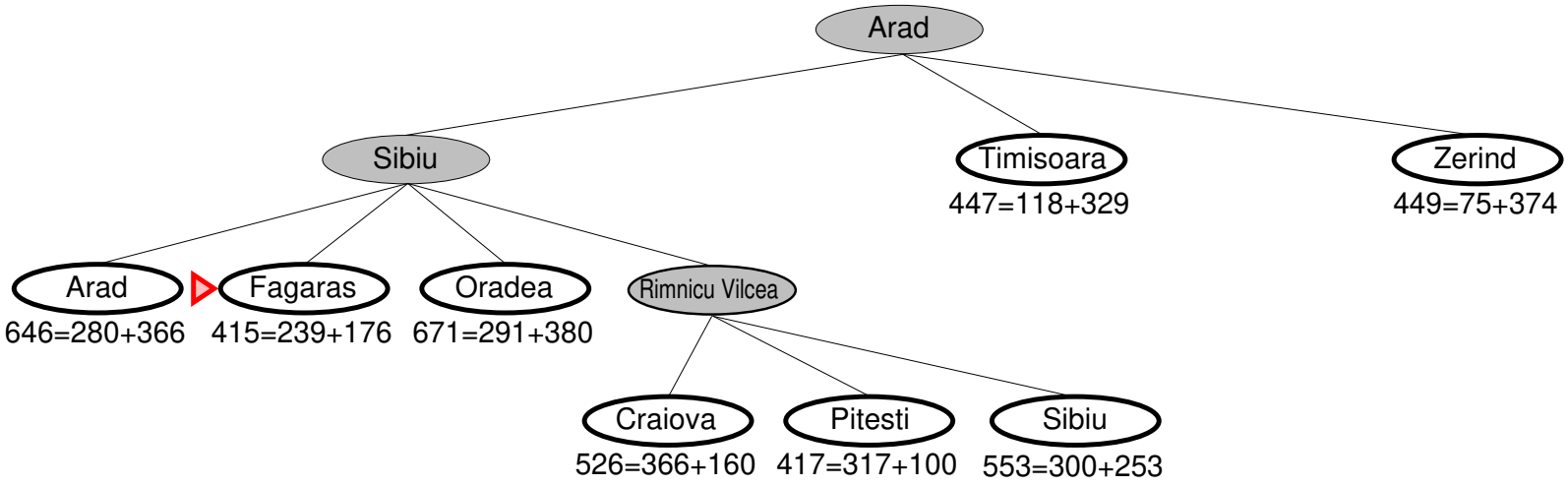
A* search example



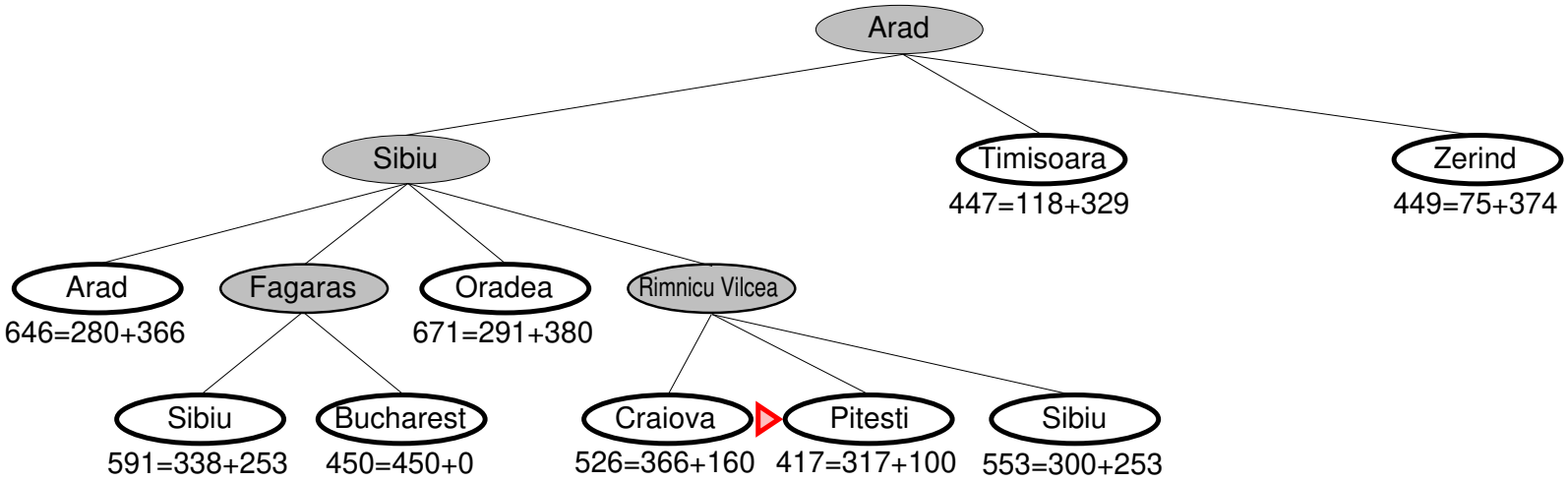
A* search example



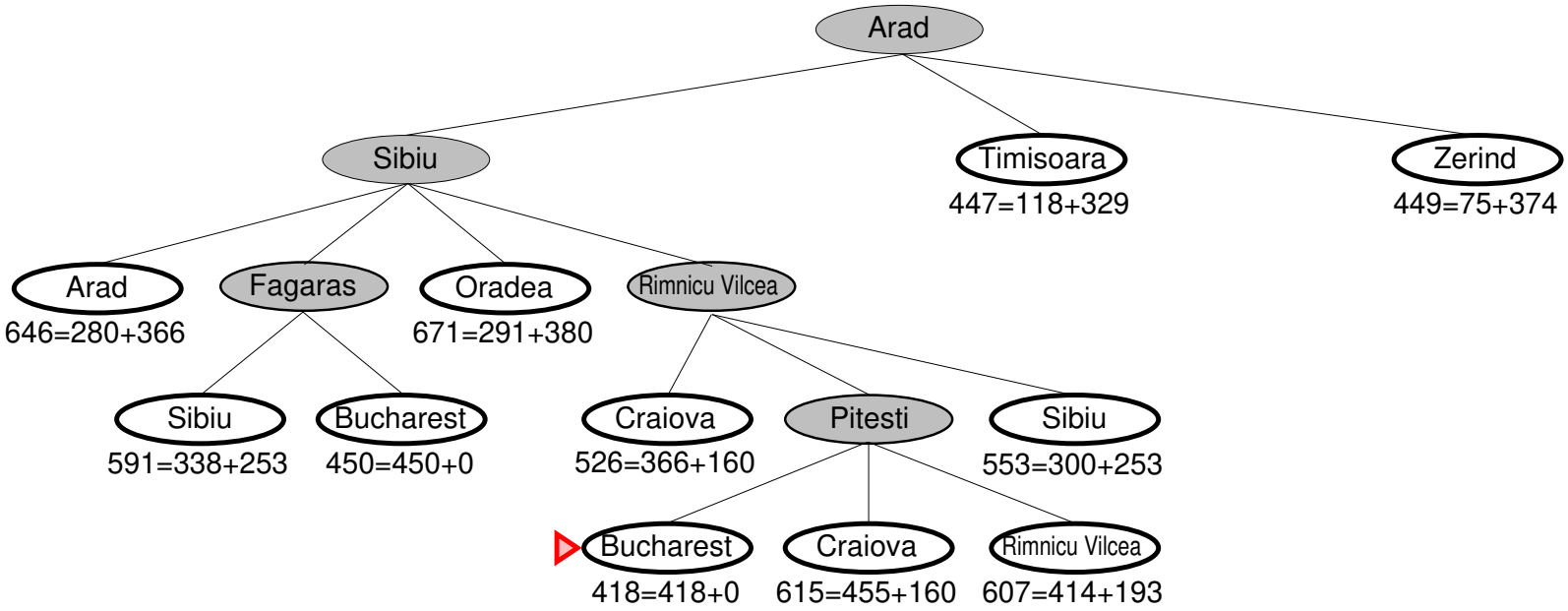
A* search example



A* search example

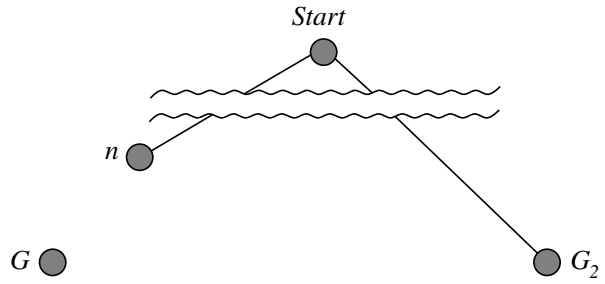


A* search example



Optimality of A*

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



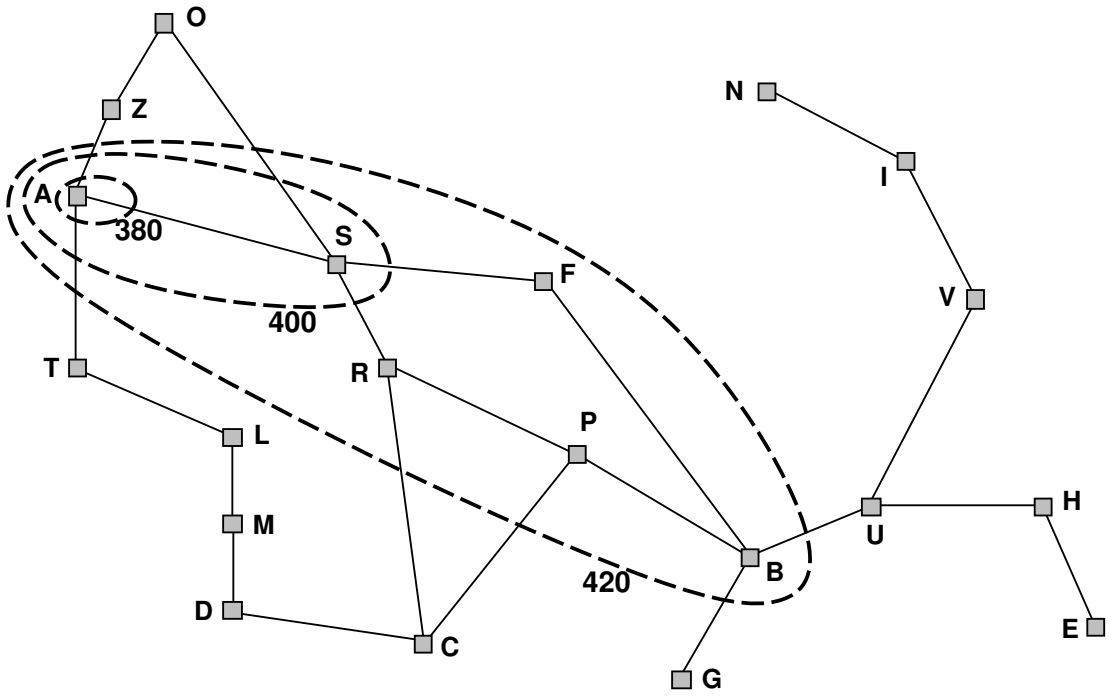
$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

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

Optimality of A*

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

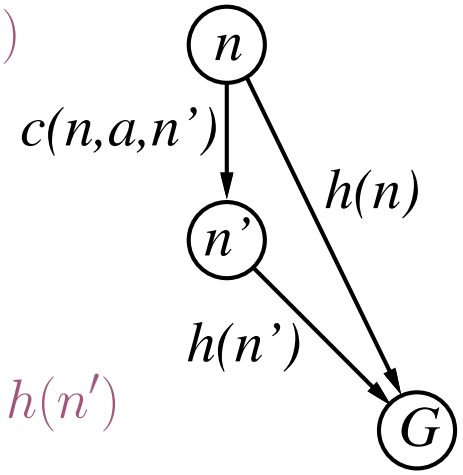
Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Heuristic consistency

A heuristic is consistent (monotonic) if

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



If h is consistent, we have

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

I.e., $f(n)$ is nondecreasing along any path (proof of the lemma)

Note

- the consistency is stronger than the admissibility
- the graph-search version of A* is optimal if $h(n)$ is consistent
- the inconsistent heuristics can be effective by enhancemence

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 $\times$ length of soln.]

– absolute error: $\Delta = h^* - h$, relative error: $\epsilon = (h^* - h)/h^*$

– exponential in the maximum absolute error, $O(b^{\Delta})$

– for constant step costs, $O(b^{\epsilon d})$, or $O((b^{\epsilon})^d)$ w.r.t. h^*

(Polynomial in various variants of the heuristics)

Space??

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.]
(Polynomial in various variants of the heuristics)

Space?? Keeps all nodes in memory
– usually running out of space long before running out of time
– overcome space problem without sacrificing optimality or completeness, at a small cost in execution time

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.]
(Polynomial in various variants of the heuristics)

Space?? Keeps all nodes in memory

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

(C^* is the cost of the optimal solution path)
A* expands all nodes with $f(n) < C^*$
A* expands some nodes with $f(n) = C^*$
A* expands no nodes with $f(n) > C^*$
prune – eliminating possibilities without having to examine

A* variants*

Problem: A* expands a lot of nodes

– taking more time and space

– memory: stored in *frontier* (what to expand next) and in *reached* states (what have visited); usually the size of frontier is much smaller than reached

Satisficing search: willing to accept solutions that are suboptimal, but are "good enough" (exploring fewer nodes); usually incomplete

- Weighted A*: $f(n) = g(n) + W \times h(n)$, $W > 1$
  – $W = 1$: A*; $W = 0$: UCS; $W = \infty$: Greedy
  – fewer states explored than A*
  – bounded suboptimal search: within a constant factor W of optimal cost

A* variants*

- Iterative-deepening A* (IDA*): similar iterative-deepening search to depth-first
  - without the requirement to keep all reached states in memory
  - a cost of visiting some states multiple times
- Memory-bounded A* (MA*) and simplified MA* (SMA*)
  - proceed just like A*, expanding the best leaf until memory is full
- Anytime A* (ATA*, time-bounded A*)
  - can return a solution even if it is interrupted before it ends (so-called anytime algorithm)
  - different than A*, the evaluation function of ATA* might be stopped and then restarted at any time
- Recursive best-first search (RBFS), see below
- Beam search, see below

Recursive best-first search

RBFS: recursive algorithm

– best-first search, but using only linear space

– similar to recursive-DLS, but

– – using the *f-limit* variable to keep track of the f-value of the best alternative path available from any ancestor of the current node

Complete?? Yes, like to A*

Time?? Exponential, depending both on the accuracy of f and on how often the best path changes as nodes are expanded

Space?? linear in the depth of the deepest optimal solution

Optimal?? Yes, like to A* (if $h(n)$ is admissible)

Recursive best-first algorithm#

```
def RECURSIVE-BEST-FIRST-SEARCH( problem)
    solution,f-value ← RBFS(problem,NOTE(problem.INITIAL),∞)
    return solution

def RBFS(problem, node, f-limit)
    if problem.IS-GOAL(node.STATE) then return node
    successors ← LIST(EXPAND(node))
    if successors is empty then return failure, ∞
    for each s in successors do // update f with value from previous search
        s.f ← max(s.PATH-COST + h(s), node.f )
    while true do
        best ← the lowest f-value node in successors
        if best.f > f-limit then return failure, best.f
        alternative ← the second-lowest f-value among successors
        result, best.f ← RBFS(problem, best, min(f-limit, alternative))
        if result≠failure then return result,best.f
```

Beam search#

Idea: keep k states instead of 1; choose top k of all their successors

- keeping the k nodes with the best f-scores
- limiting the size of the frontier, saving memory
- incomplete and suboptimal

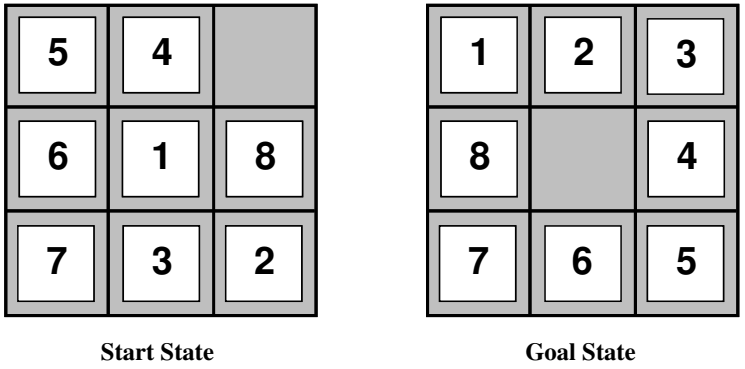
Not the same as k searches run in parallel

Searches that find good states recruit other searches to join them

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)

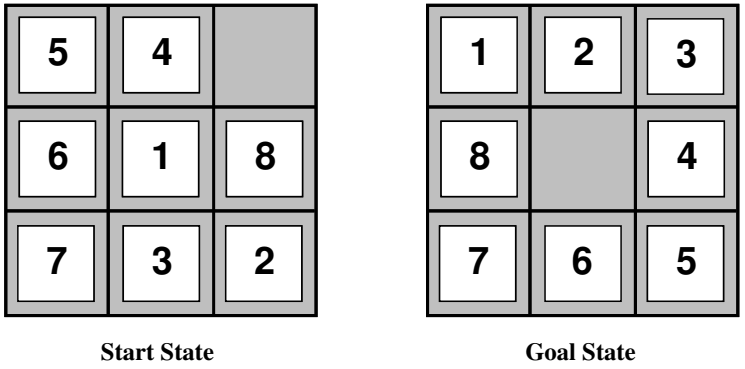


$h_1(S)$ =??
$h_2(S)$ =??

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)



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

New kind of distance??

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 $\approx$ 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

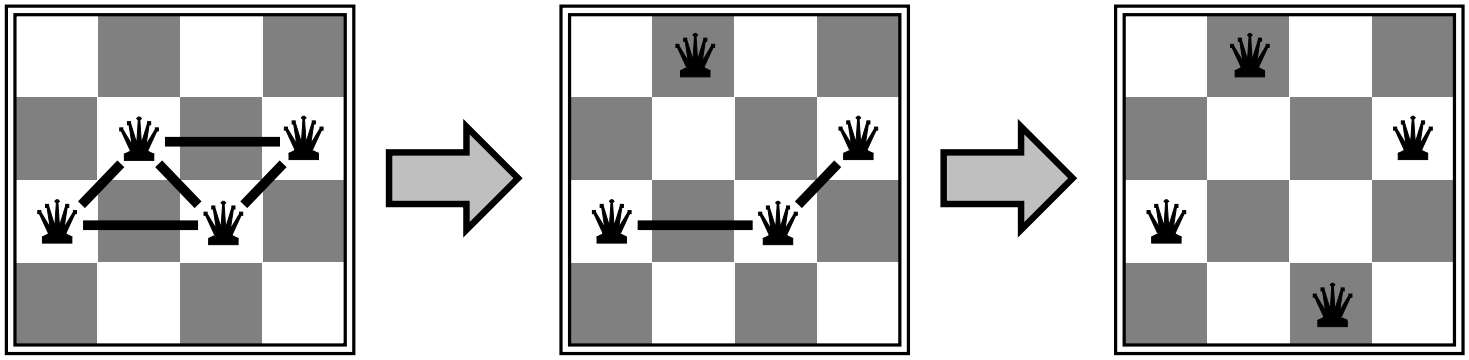
If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

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

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce the number of conflicts



Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1$ million

Ref: Pearl, J (1984), *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison- Wesley