Solving problems by searching

CHAPTER 3

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

- Goal-based agent called a problem-solving agent
- Problem-solving agents use atomic representations, that is, states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms.
- Goal-based agents that use more advanced factored or structured representations are usually called planning agents.

- We will see several **uninformed search** algorithms—algorithms that are given no information about the problem other than its definition.
- Although some of these algorithms can solve any solvable problem, none of them can do so efficiently.
- **Informed search algorithms**, on the other hand, can do quite well given some guidance on where to look for solutions.
- the simplest kind of task environment, for which the solution to a problem is always a fixed sequence of actions.

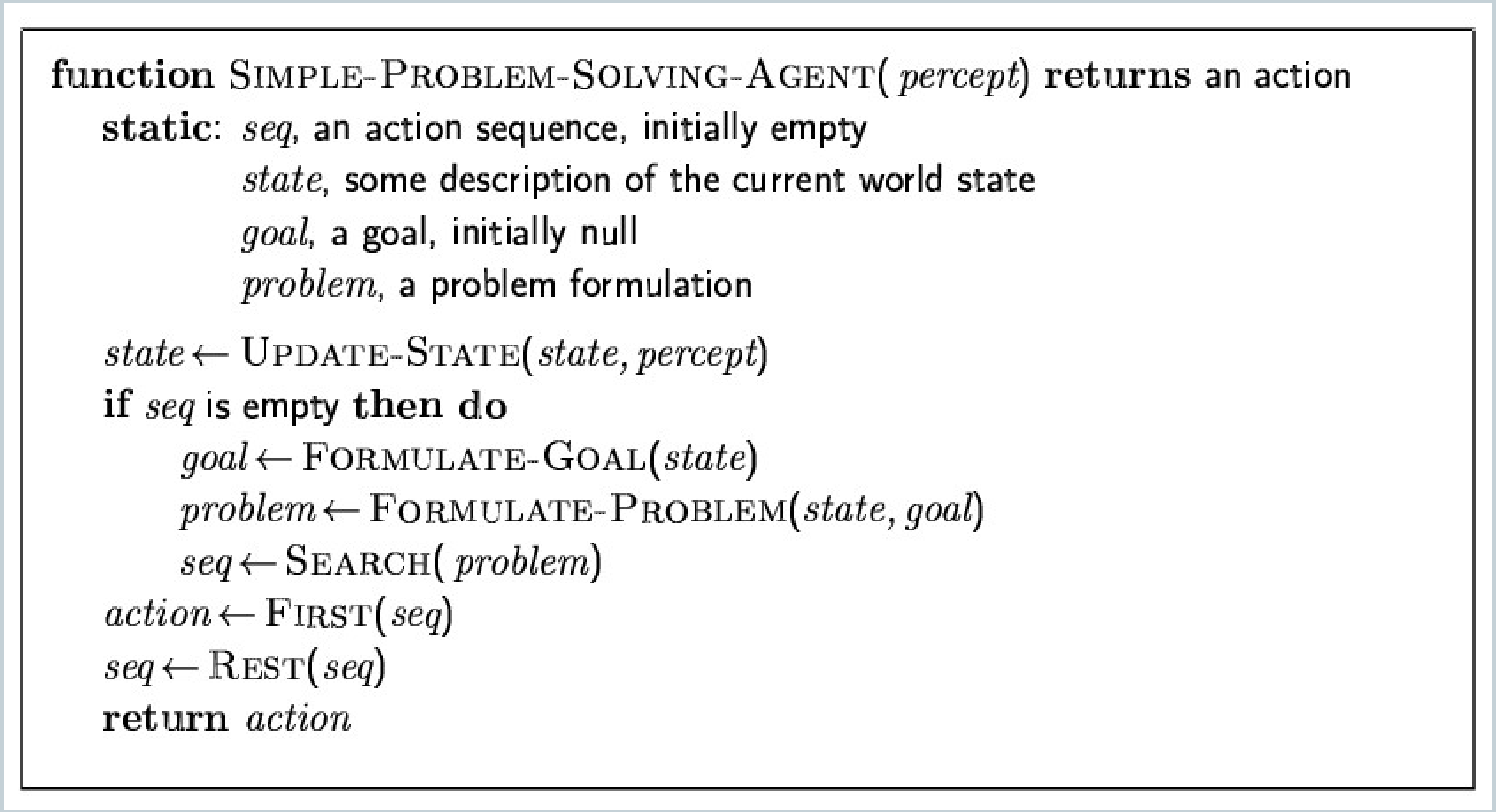
Recall - important

- Asymptotic complexity
- O() notation
- NP-completeness

Problem Solving Agents

- Intelligent agents are supposed to maximize their performance measure.
- Simplified if the agent can adopt a goal and aim at satisfying it

Problem-solving agents



```
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 do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

- An agent on holiday in Romania; currently in Arad.
- Agent's performance measure contains many factors: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife and so on
- Flight leaves tomorrow from Bucharest (non refundable ticket)
- It makes sense for the agent to adopt the goal of getting to Bucharest
- Courses of action that don't reach Bucharest on time can be rejected without further consideration.
- The agent's decision problem is greatly simplified.

GOAL FORMULATION

- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- **Goal formulation**, based on the current situation and the agent’s performance measure, is the first step in problem solving.
- Consider a goal to be a set of world states—exactly those states in which the goal is satisfied. The agent’s task is to find out how to act, now and in the future, so that it reaches a goal state.
- Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider.

Problem formulation

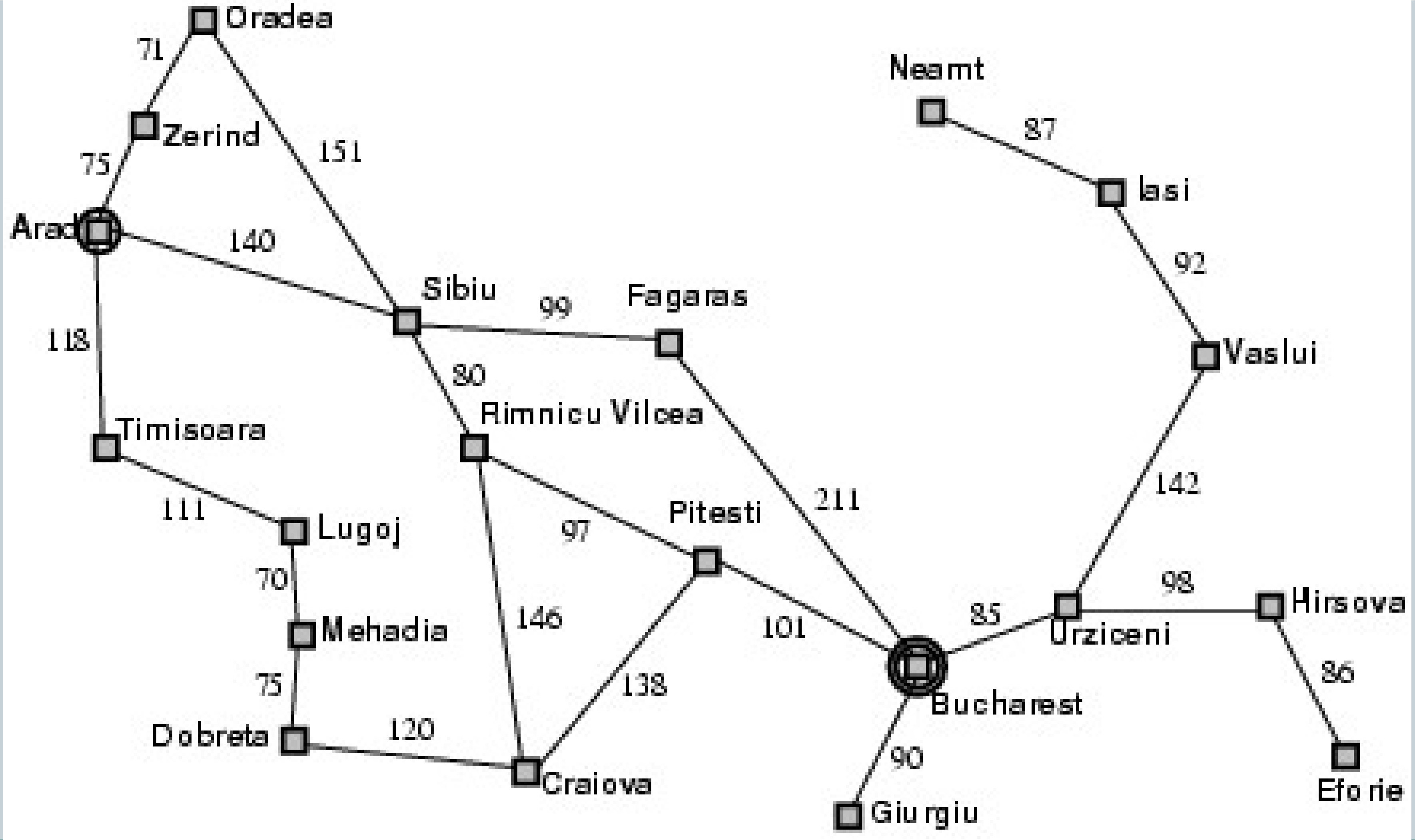
- Problem formulation is the ***process of deciding what actions and states to consider, given a goal***.
- For now, let us assume that the agent will consider actions at the level of driving from one major town to another.
- Each state therefore corresponds to being in a particular town.

- Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad.
- Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.
- None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.
- In other words, the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action.

- If the agent has no additional information—i.e., if the environment is unknown in the sense.
- But suppose the agent has a map of Romania. The point of a map is to provide the agent with information about the states it might get itself into and the actions it can take.
- The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest.
- Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey.

- Assume that the environment is **observable**, so the agent knows the current state. In Romania, it's reasonable that each city on the map has a sign indicating its presence.
- Assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from. This is true for navigating in Romania because each city is connected to a small number of other cities.
- Assume the environment is **known**, so the agent knows which states are reached by each action. (Having an accurate map suffices to meet this condition for navigation problems.)
- Assume that the environment is **deterministic**, so each action has exactly one outcome. Under ideal conditions, this is true for the agent in Romania—it means that if it chooses to drive from Arad to Sibiu, it does end up in Sibiu.

Example: Romania



Example: Romania

- 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

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase

Well-defined problems and solutions

- A **problem** can be defined formally by five components
- The **initial** state that the agent starts in. For eg., the initial state for our agent in Romania might be described as In(Arad).
- A description of the possible **actions** available to the agent. Given a particular state *s*, *ACTIONS(s)* returns the set of actions that can be executed in s.
- We say that each of these actions is **applicable** in *s*. For example, from the state In(Arad), the ***applicable actions*** are {Go(Sibiu),Go(Timisoara),Go(Zerind)}.

Transition model

- A description of what each action does; the formal name for this is the **transition model**, specified by a function *RESULT(s,a)* that returns the state that results from doing action a in state s.
- The term **successor** is used to refer to any state reachable from a given state by a single action.
- For eg,
- RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

State Space, Graph and Path

- Together, the **initial state, actions, and transition model** implicitly define the **state space** of the problem—the set of all states reachable from the initial state by any sequence of actions.
- State space forms a directed network or **graph** in which the nodes are **states** & links between nodes are **actions**.
- (The map of Romania can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction.)
- A **path** in the state space is a sequence of states connected by a sequence of actions.

Goal test

- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent’s goal in Romania is the singleton set {In(Bucharest)}.
- Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For eg, in chess, the goal is to reach a state called “checkmate,” where the opponent’s king is under attack and can’t escape.

Path Cost

- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.
- To get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers.
- The cost of a path can be described as the sum of the costs of the individual actions along the path.
- The **step cost** of taking action *a* in state *s* to reach state *s'* is denoted by *c(s, a, s')*. The step costs for Romania are route distances.
- step costs are nonnegative

Optimal solution

- The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm.
- A **solution** to a problem is an action sequence that leads from the initial state to a goal state.
- Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

Formulating problems

- Formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost seems reasonable, but it is still a model—an abstract mathematical description—and not the real thing.
- Compare the simple state description we have chosen, In(Arad), to an actual cross country trip, where the state of the world includes so many things: the traveling companions, the scenery out of the window, the proximity of law enforcement officers, the distance to next rest stop, condition of the road, weather, and so on

Abstraction

- All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called **abstraction**.

- In addition to **abstracting** the state description, we must abstract the actions themselves. A driving action has many effects.
  Besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent (as they say, travel is broadening). Our formulation takes into account only the change in location.

Example Problems

- The problem-solving approach has been applied to a vast array of task environments. Distinguishing between toy and real-world problems.
- A toy problem is intended to illustrate or exercise various problem-solving methods.
- Given a concise, exact description & hence is usable by different researchers to compare the performance of algorithms.
- A real-world problem is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

Toy Problems - Vacuum World

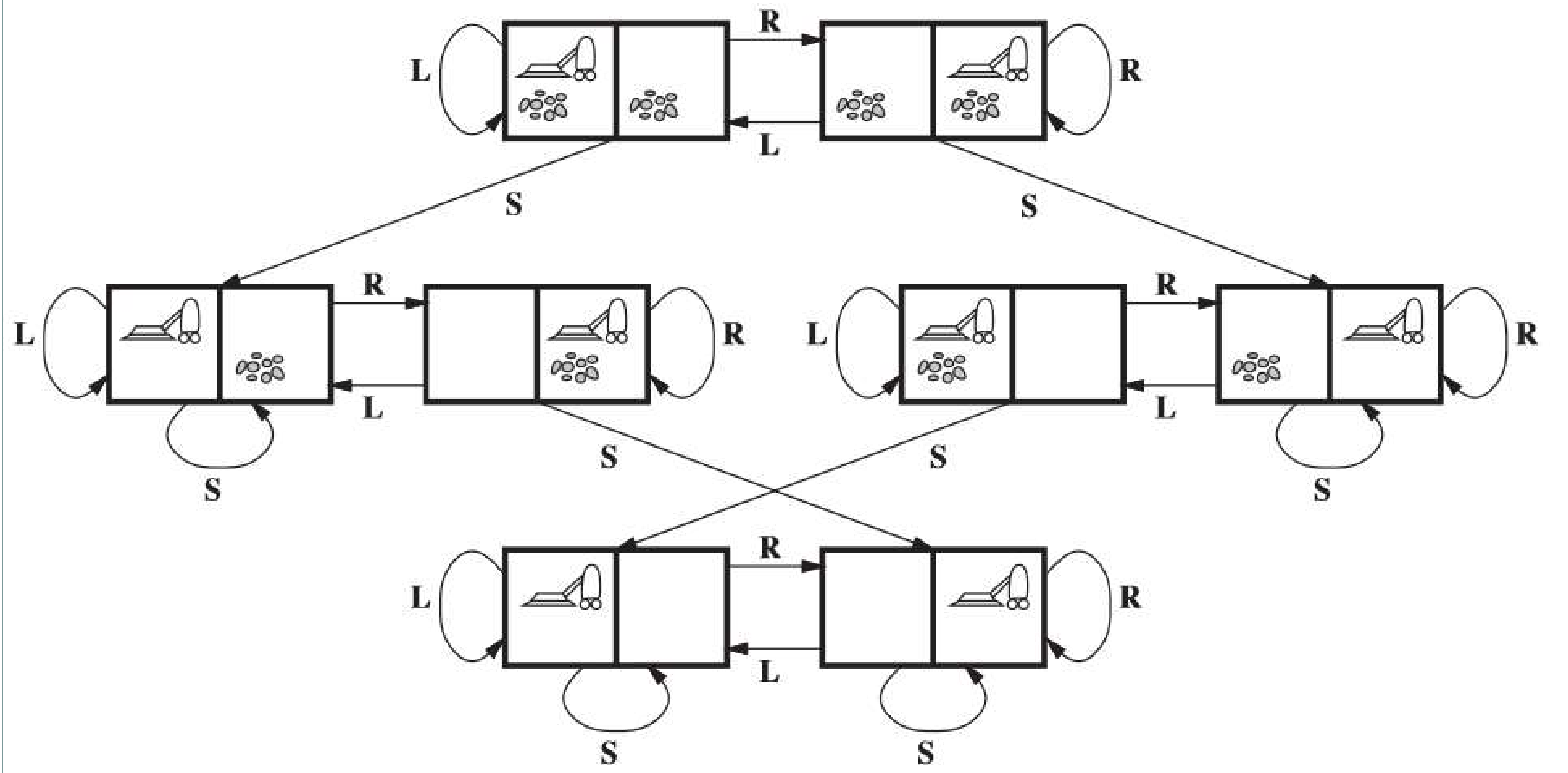
- Formulated as a problem as follows
- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states.
- A larger environment with *n* locations has $n - 2^n$ states.
- **Initial state:** Any state can be designated as the initial state.

- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include Up and Down.
- **Transition model:** The actions have their expected effects, except that moving Left in the left most square, moving Right in the right most square, and Sucking in a clean square have no effect.
- **Goal test**: Checks whether all the squares are clean.
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

Problem types

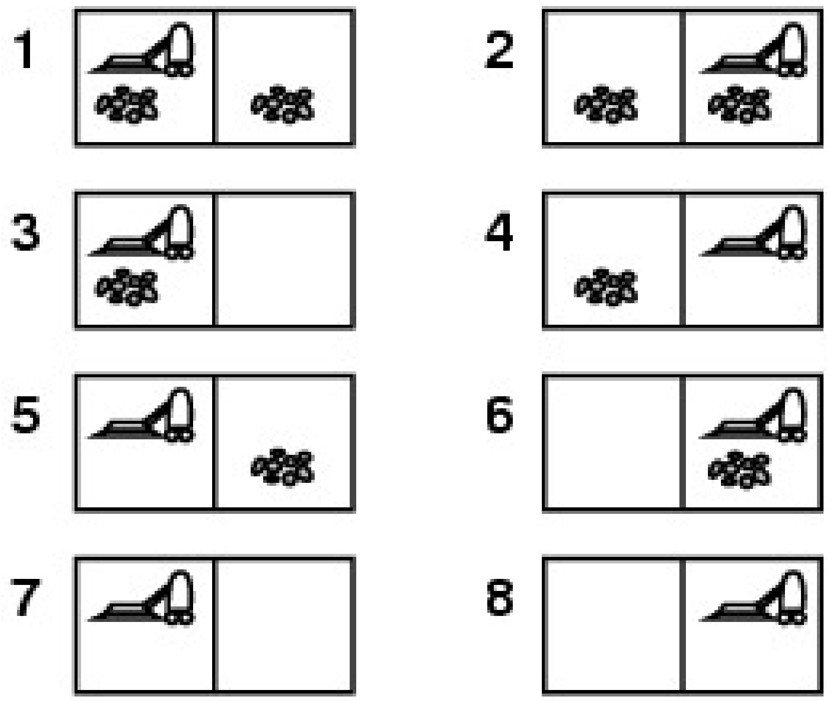
- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave} search, execution
- Unknown state space → exploration problem

State space for the vacuum world



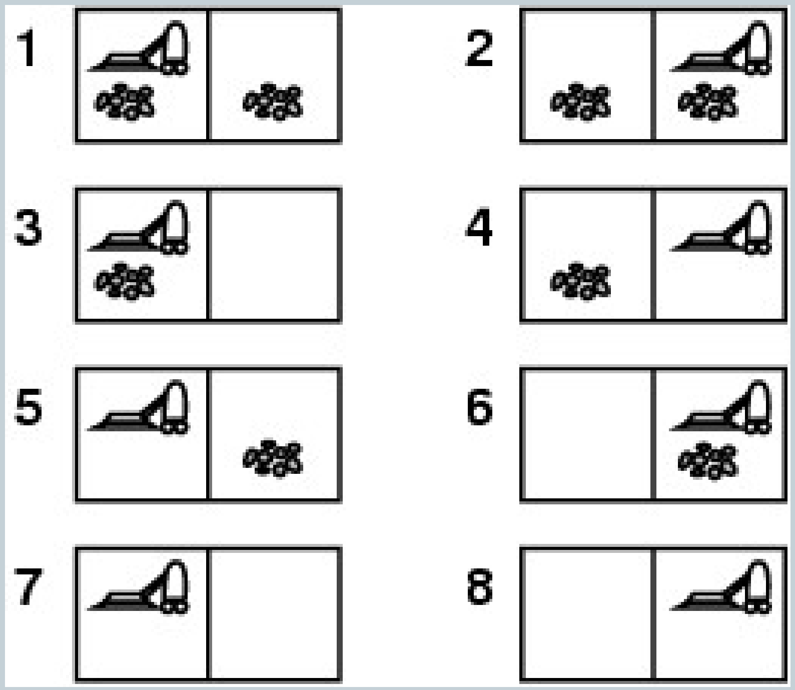
Example: vacuum world

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



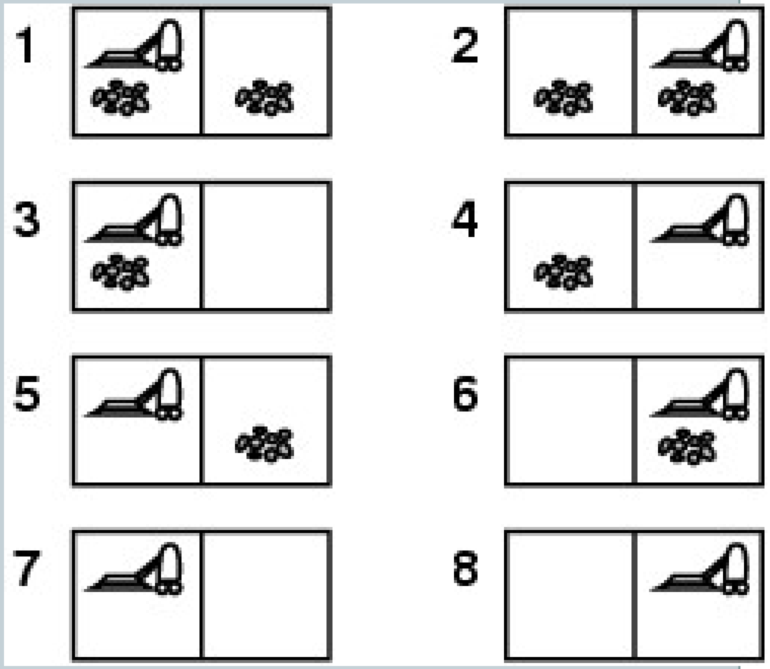
Example: vacuum world

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



Example: vacuum world

- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean]*, i.e., start in #5 or #7
    Solution?
  - *[Right,* ***if*** *dirt* ***then*** *Suck]*
  -



Single-state problem formulation

A problem is defined by four items:

1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. 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

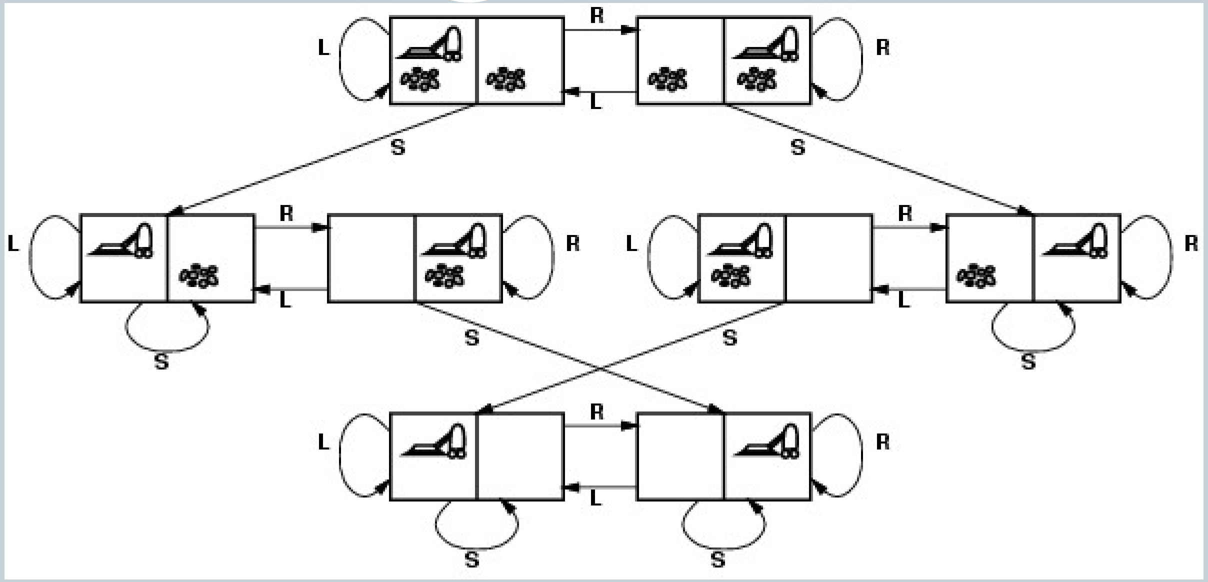
- A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

- Real world is absurdly complex
  → state space must be abstracted for problem solving

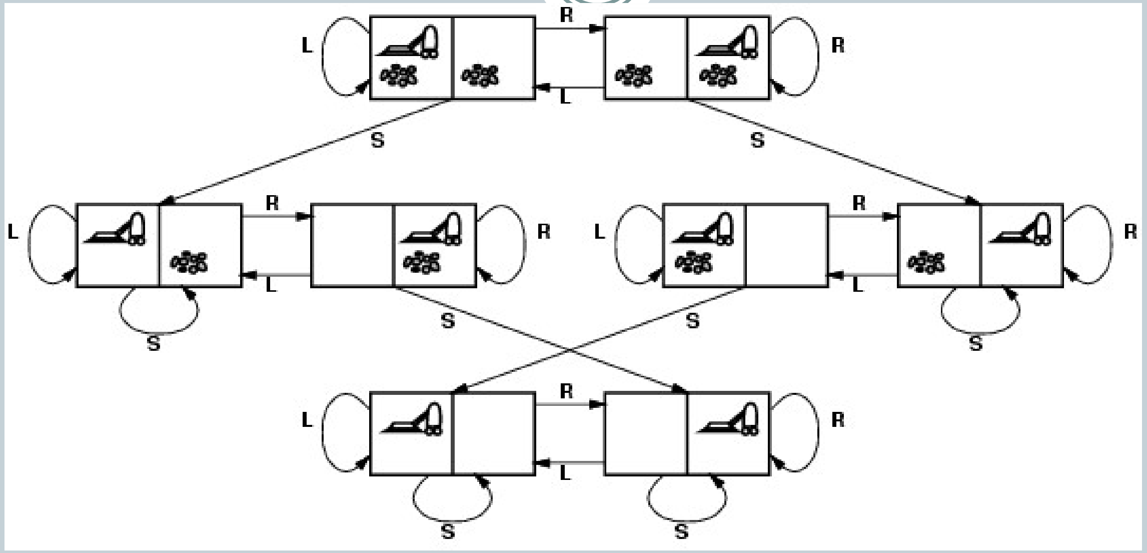
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

Vacuum world state space graph



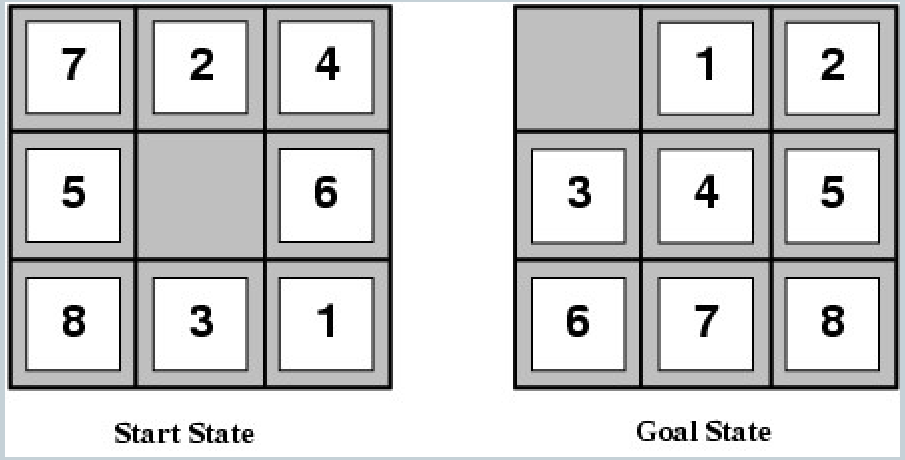
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



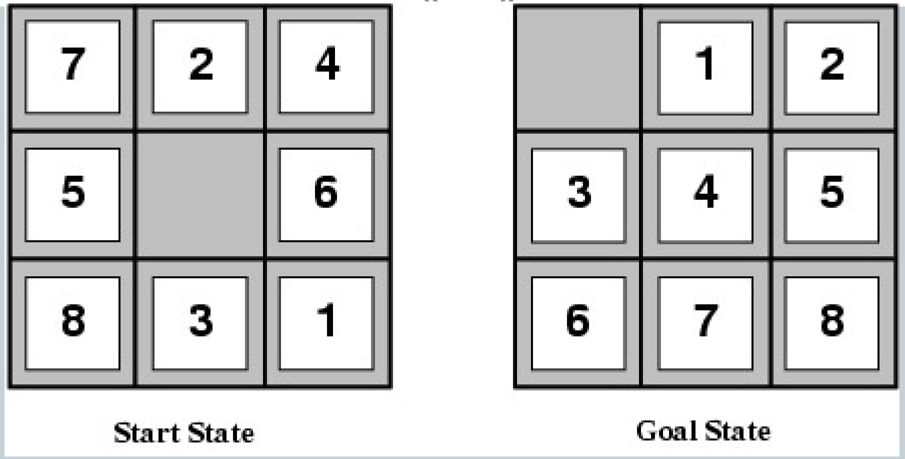
- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



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

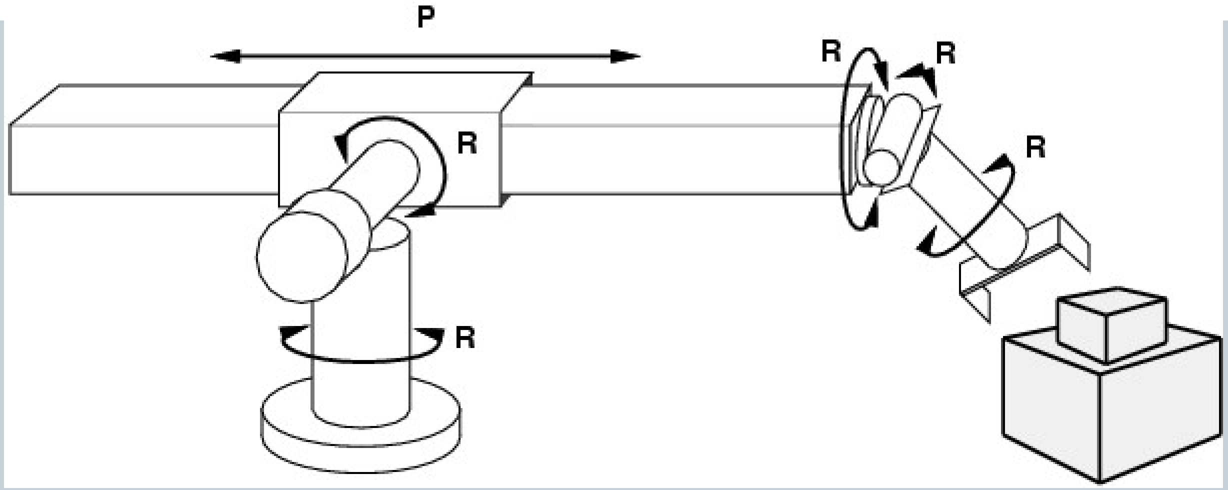
Example: The 8-puzzle



- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move
-

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: robotic assembly

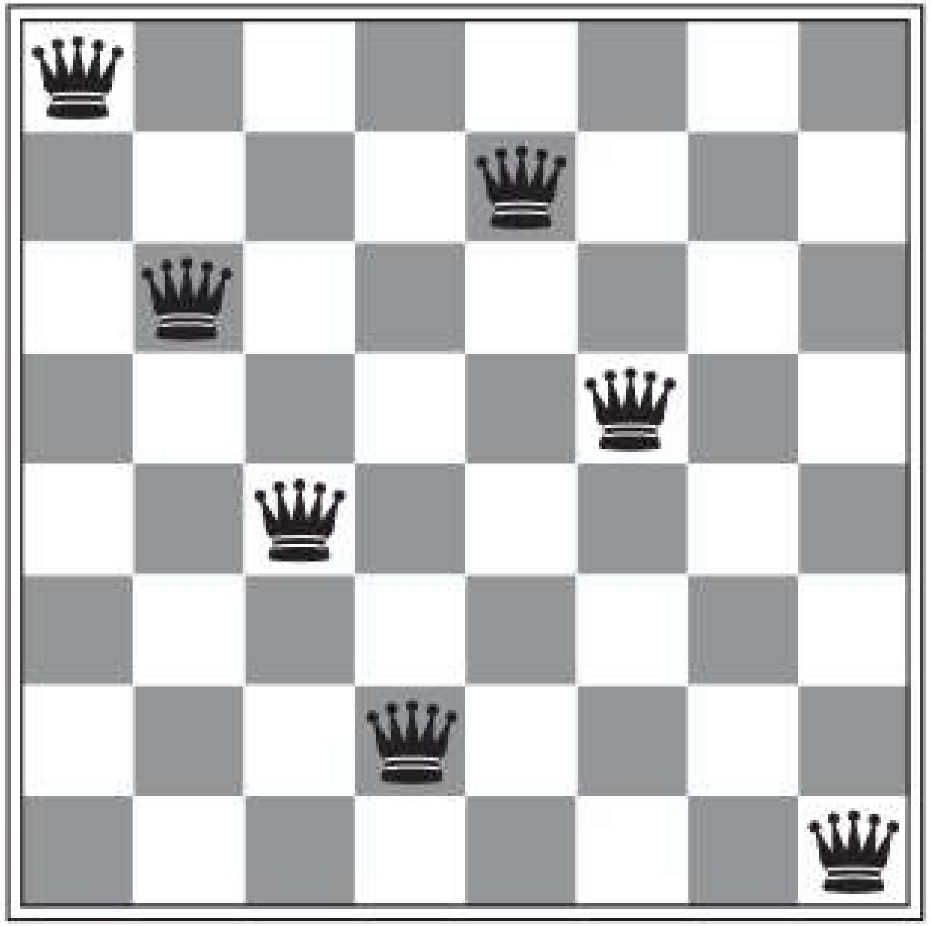


- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

8 Queen Problem

- States: Any arrangement of 0 to 8 queens on the board is a state.
- Initial state: No queens on the board.
- Actions: Add a queen to any empty square.
- Transition model: Returns the board with a queen added to the specified square.
- Goal test: 8 queens are on the board, none attacked.

Almost a solution to the 8-queens problem



Real world problems

- The route-finding problem is defined in terms of specified locations & transitions along links.
- They are used in a variety of applications, such as Web sites & in-car systems that provide driving directions.
- Robot navigation is a generalization of the route-finding problem
- Touring problems are closely related to route-finding problems, but with an important difference that is -- need to visit every city.
- The traveling salesperson problem (TSP) is a touring problem in which each city must be visited exactly once.

Searching for Solution – Search Tree

- A solution is an action sequence, so search algorithms work by considering various possible action sequences.
- The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are **actions** and the **nodes** correspond to states in the state space of the problem.
- The first few steps in growing the search tree for finding a route from Arad to Bucharest is shown.

Root node, Parent node and Child node

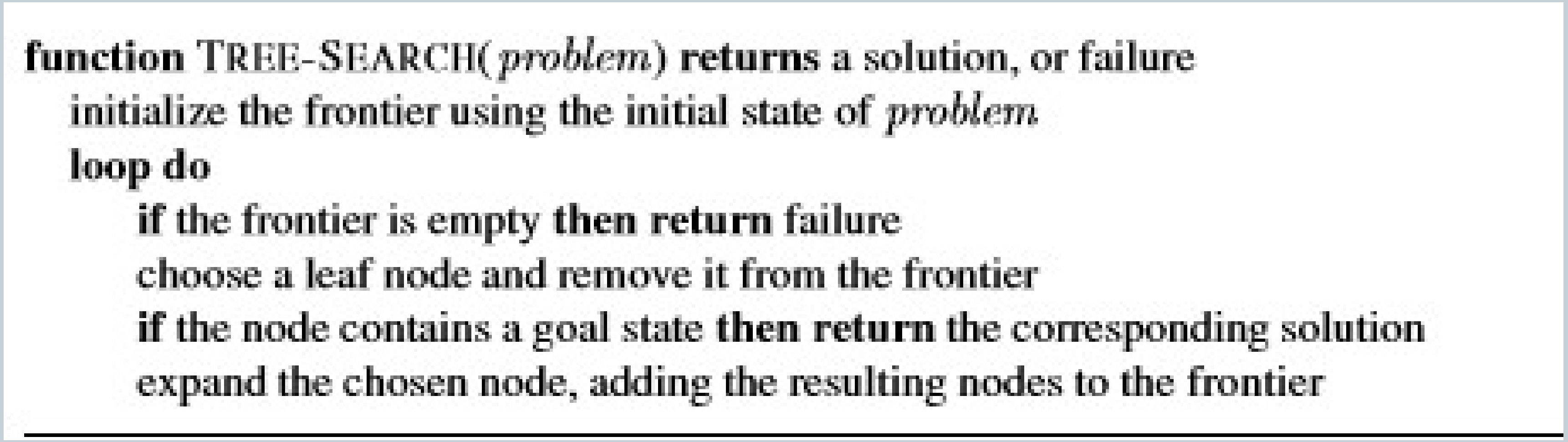
- **Root node** of the tree corresponds to the initial state, *In(Arad)*.
- The first step is to test whether this is a goal state. Then consider taking various actions.
- Do by **expanding** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states.
- Add three branches from the **parent node** *In(Arad)* leading to three new **child nodes**: *In(Sibiu)*, *In(Timisoara)*, and *In(Zerind)*.
- Choose which of these three possibilities to consider further.

Frontier

- The set of all leaf nodes available for expansion at any given point is called the **frontier**.
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.
- Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called search strategy

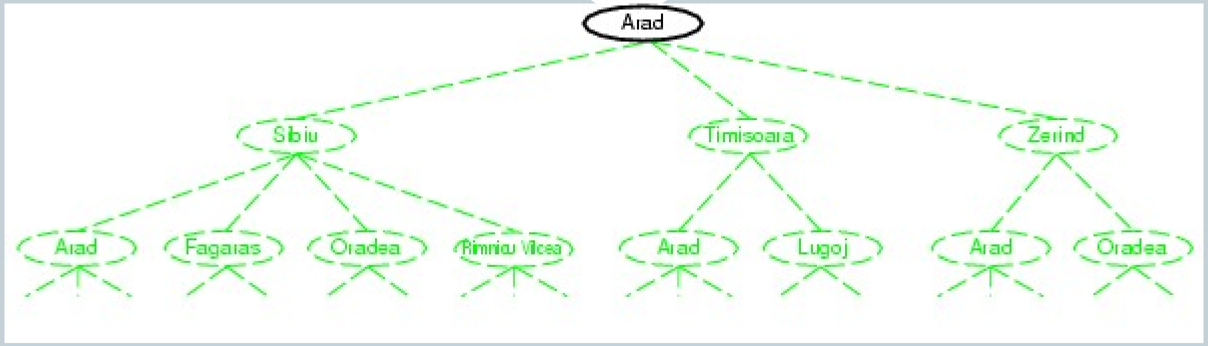
Tree search algorithms

- Basic idea:
  - exploration of state space by generating successors of already-explored states (expanding)

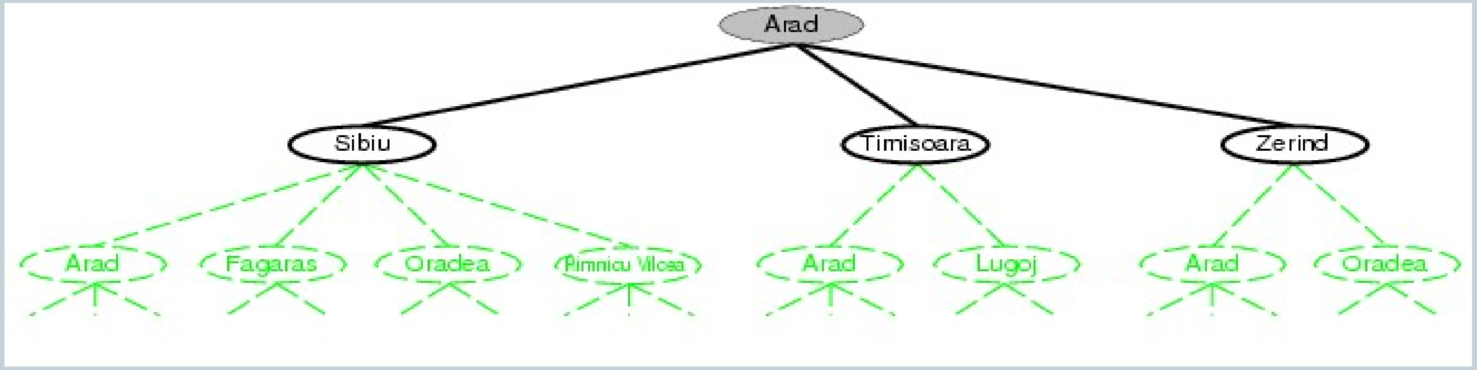


```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding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

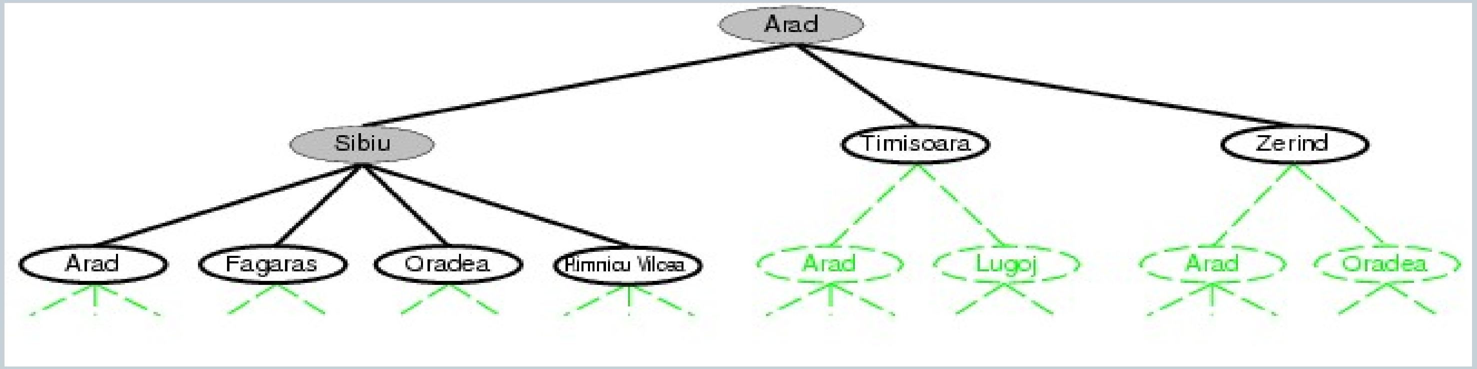
Tree search example



Tree search example



Tree search example



Loopy path

- notice a peculiar thing : it includes the path from Arad to Sibiu and back to Arad again!
- In(Arad) is a **repeated state** in the search tree, generated in this case by a **loopy path**.
- Considering such loopy paths means that the complete search tree for Romania is infinite because there is no limit to how often one can traverse a loop.
- State Space has only 20 states.
- loops can cause certain algorithms to fail, making otherwise solvable problems unsolvable.

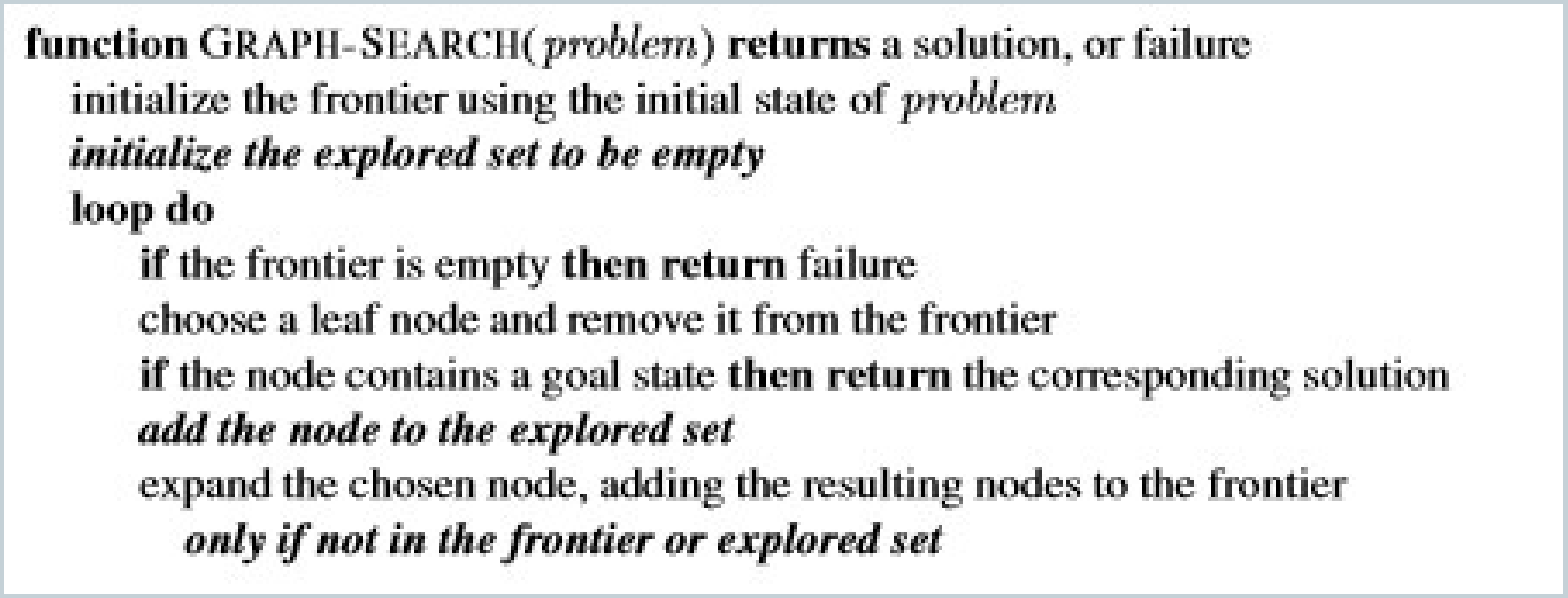
Redundant paths

- Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one way to get from one state to another.
- Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long).
- Obviously, the second path is redundant—it's just a worse way to get to the same state.

Graph search

- *Algorithms that forget their history are doomed to repeat it.*
- The way to avoid exploring redundant paths is to remember where one has been.
- Augment the TREE-SEARCH algorithm with a data structure called the explored set, which remembers every expanded node.
- Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.
- The new algorithm is called GRAPH-SEARCH

Algorithm – Graph Search



```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be 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 the corresponding solution
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
         only if not in the frontier or explored set
```

Search Tree

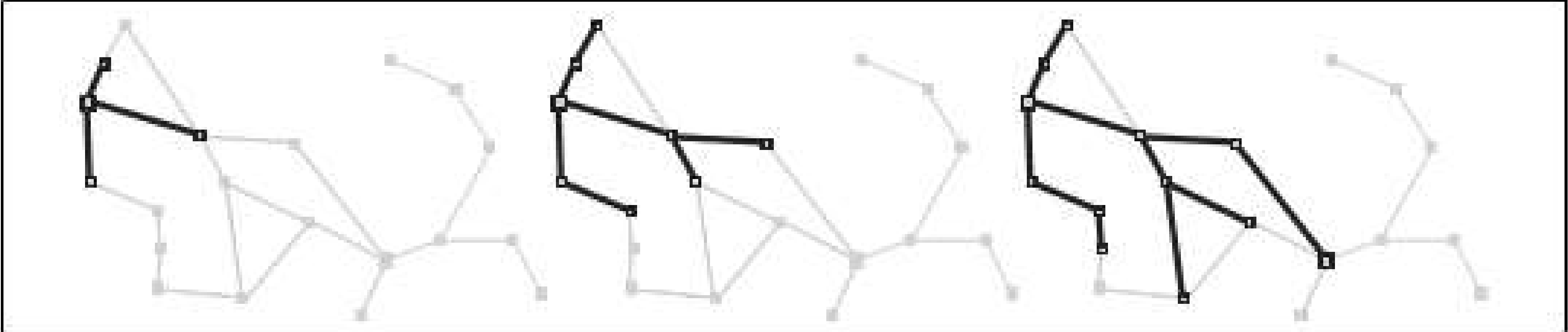


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Rectangular grid

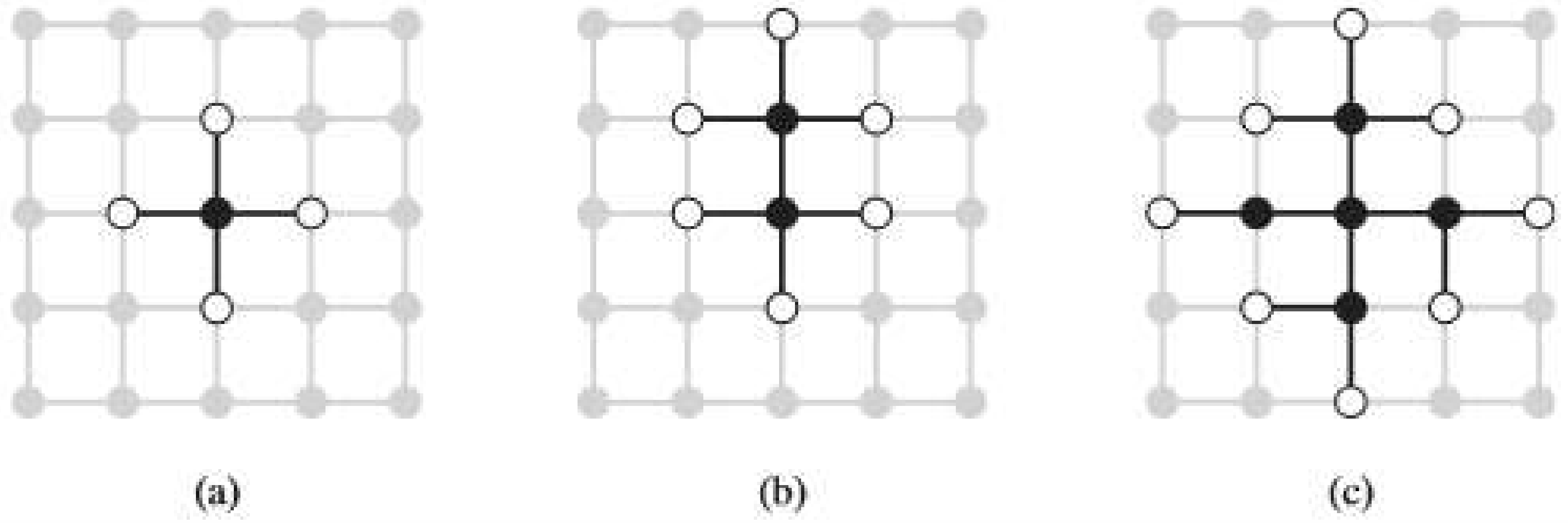


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node *n* of the tree, a structure contains four components:
- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

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 state, parent node, action, path cost *g(x)*, depth

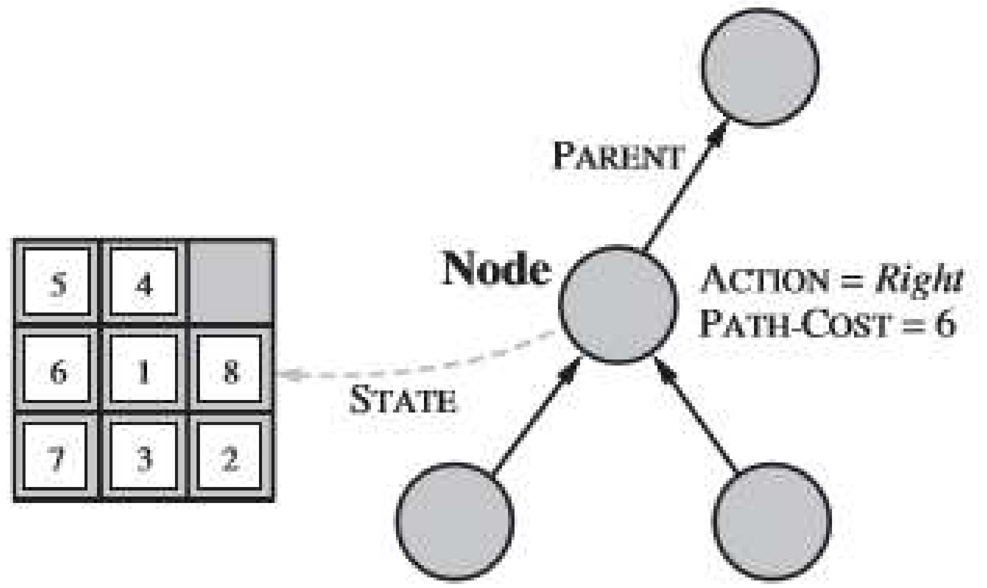


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Queue structure

- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- appropriate data structure for this is a **queue**.
- Operations on a queue are as follows:
  - EMPTY? (queue) returns true only if there are no more elements in the queue.
  - POP(queue) removes the first element of the queue and returns it.
  - INSERT(element, queue) inserts an element and returns the resulting queue.

Measuring problem-solving performance

- A search strategy is defined by picking the order of node expansion.
- Strategies are evaluated along the following dimensions:
  - **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
  - **Optimality**: Does the strategy find the optimal solution
  - **Time complexity**: How long does it take to find a solution?
  - **Space complexity**: How much memory is needed to perform the search

Time and Space Complexity

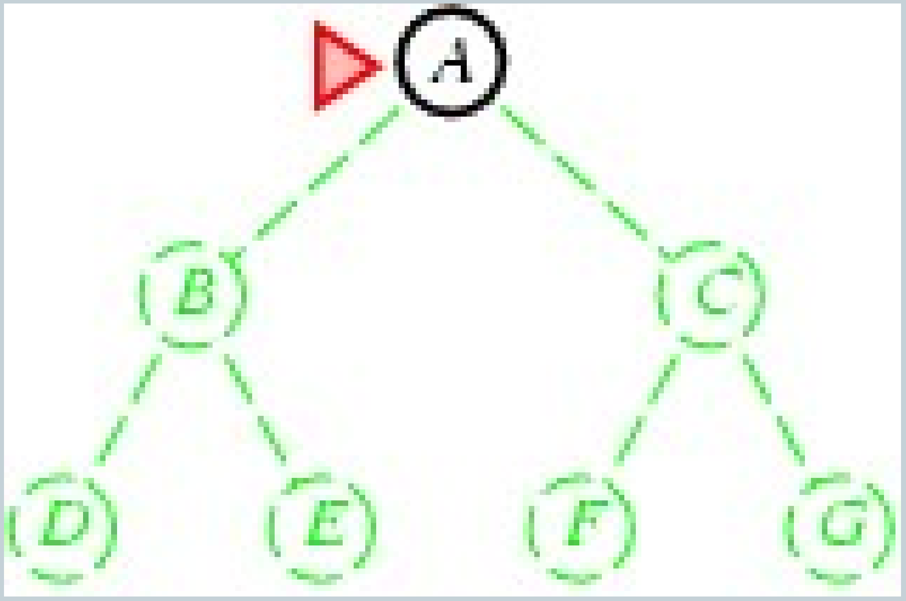
- Complexity is expressed in terms of three quantities:
  - b, the branching factor or maximum number of successors of any node;
  - d, the depth of the shallowest goal node (i.e., the number of steps along the path from the root);
  - m, the maximum length of any path in the state space.
- Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
- Also known as Blind search
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

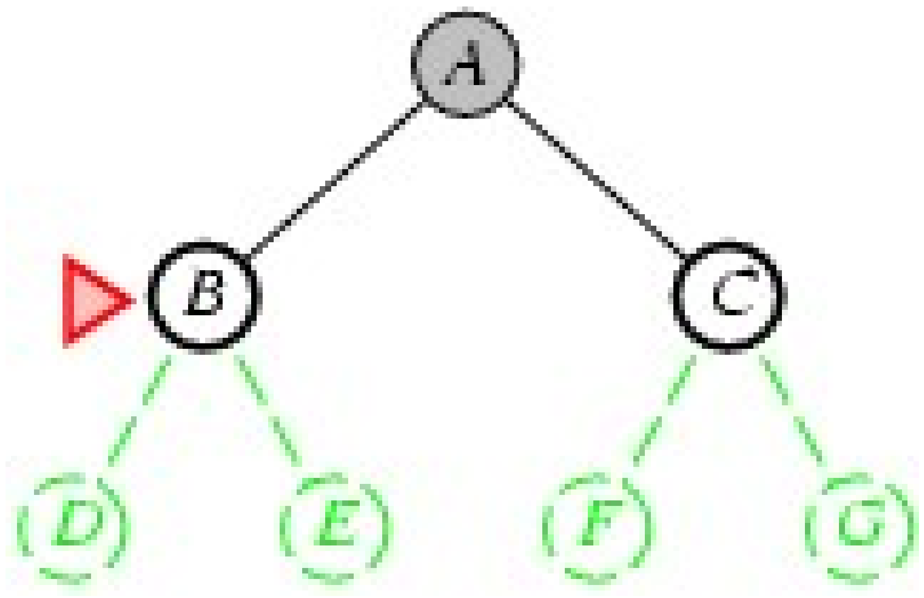
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - is a FIFO queue, i.e., new successors go at end
  -



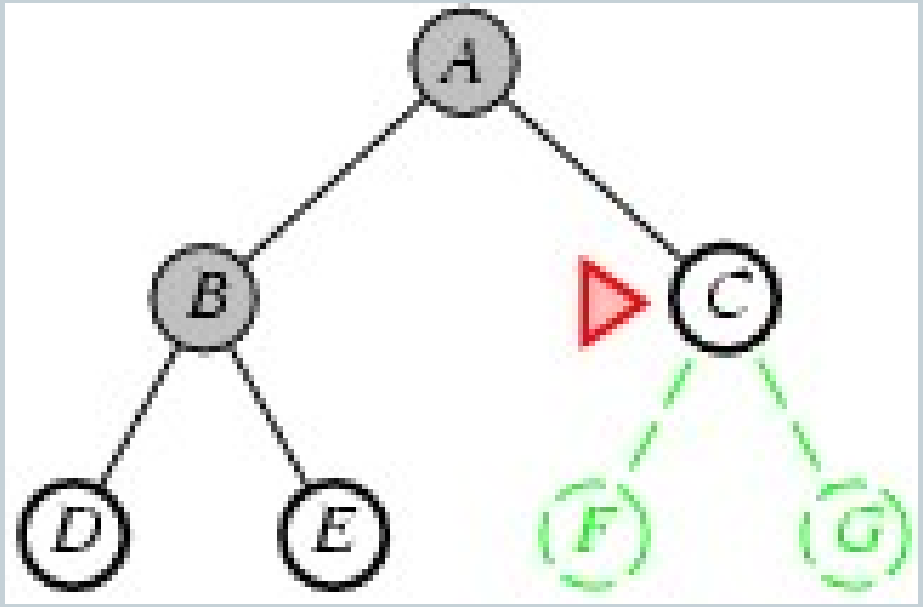
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - is a FIFO queue, i.e., new successors go at end



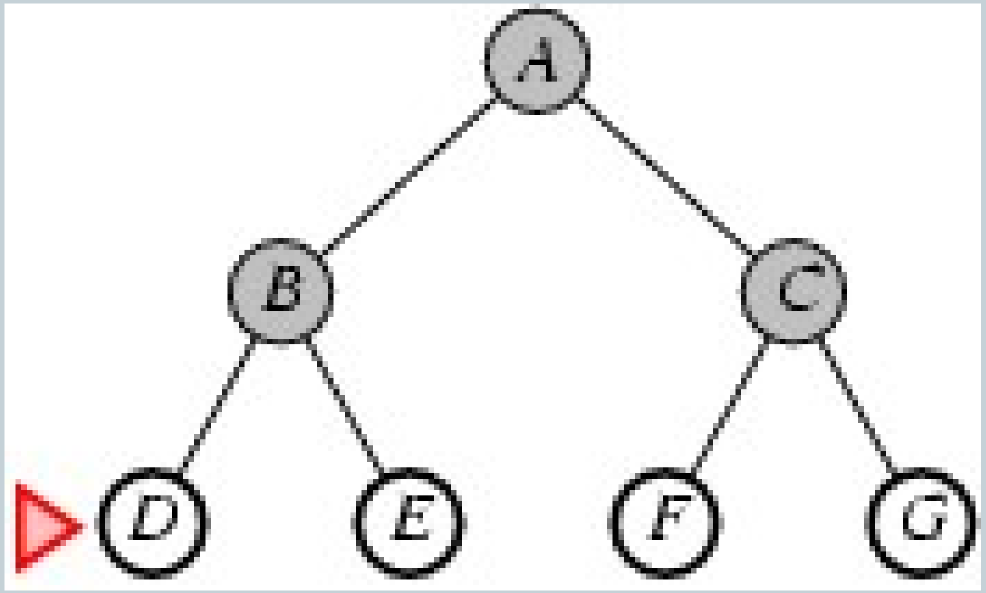
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - is a FIFO queue, i.e., new successors go at end
  -



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - is a FIFO queue, i.e., new successors go at end
  -



Properties of breadth-first search

- Complete? Yes (if *b* is finite)
- If the shallowest goal node is at some finite depth *d*, it will eventually find it after generating all shallower nodes (provided the branching factor b is finite).
- Time? $1+b+b^2+b^3+\dots+b^d + b(b^d-1) = O(b^d)$
- Search a uniform tree where every state has *b* successors. The root of the search tree generates *b* nodes at the 1st level, each of which generates *b* more nodes, for a total of b^2 at the second level. Each of these generates *b* more nodes, yielding b^3 nodes at the third level, and so on.
- Suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is $b + b^2 + b^3 + \cdots + b^d = O(b^d)$.

- Space? *$O(b^{d+1})$* (keeps every node in memory)
- For breadth-first graph search in particular, every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier. so the space complexity is O(bd), i.e., it is dominated by the size of the frontier.
- Optimal? Yes (if cost = 1 per step)

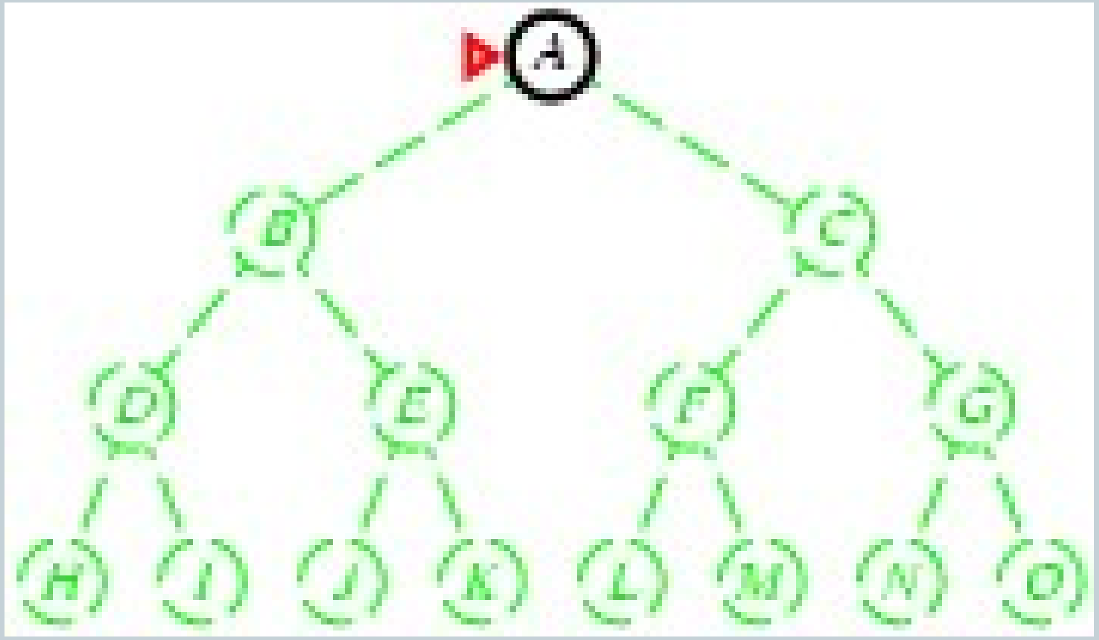
- Space is the bigger problem (more than time)

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *Frontier* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost≥ ε(some small positive constant)
- Time and Space ?
  - Number of nodes with $g \leq$ cost of optimal solution, $O(b^{1+ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Optimal? Yes – nodes expanded in increasing order of *g(n)*

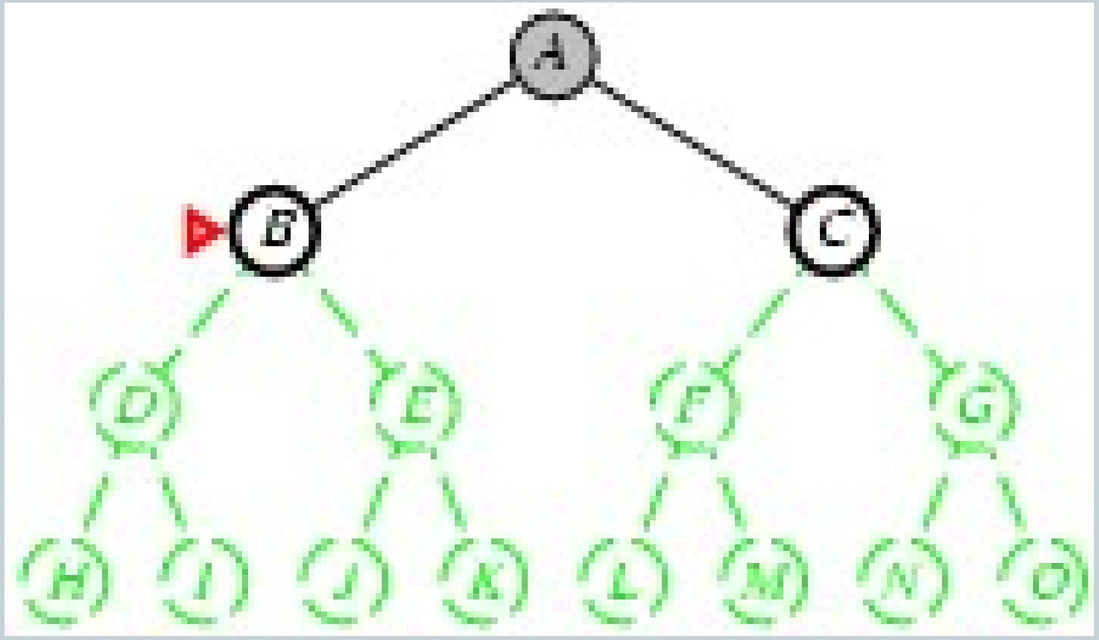
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front

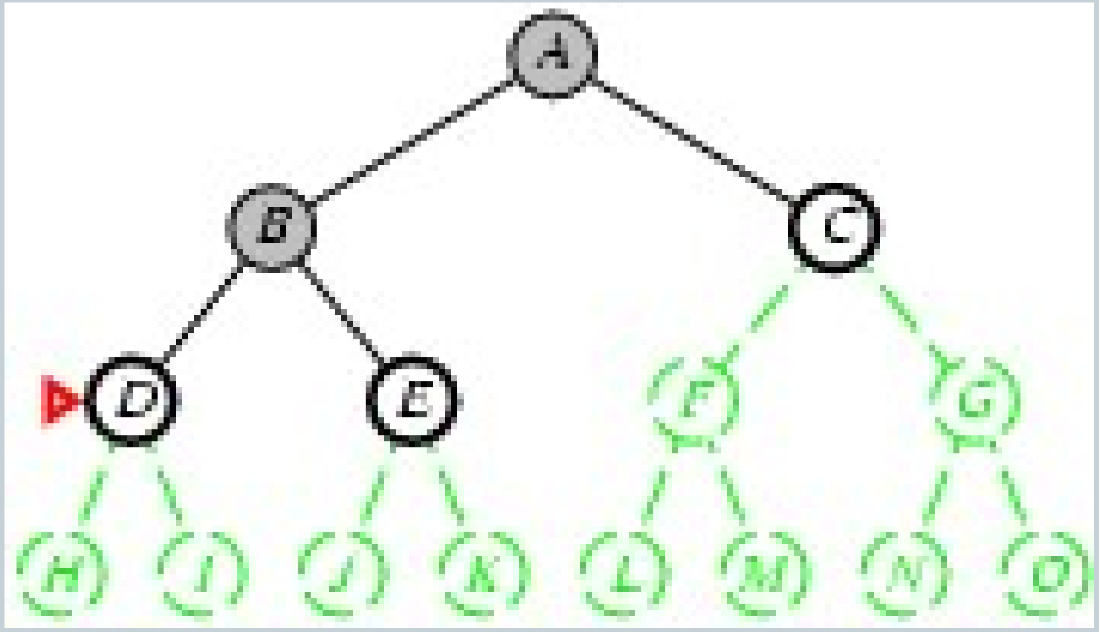


Depth-first search

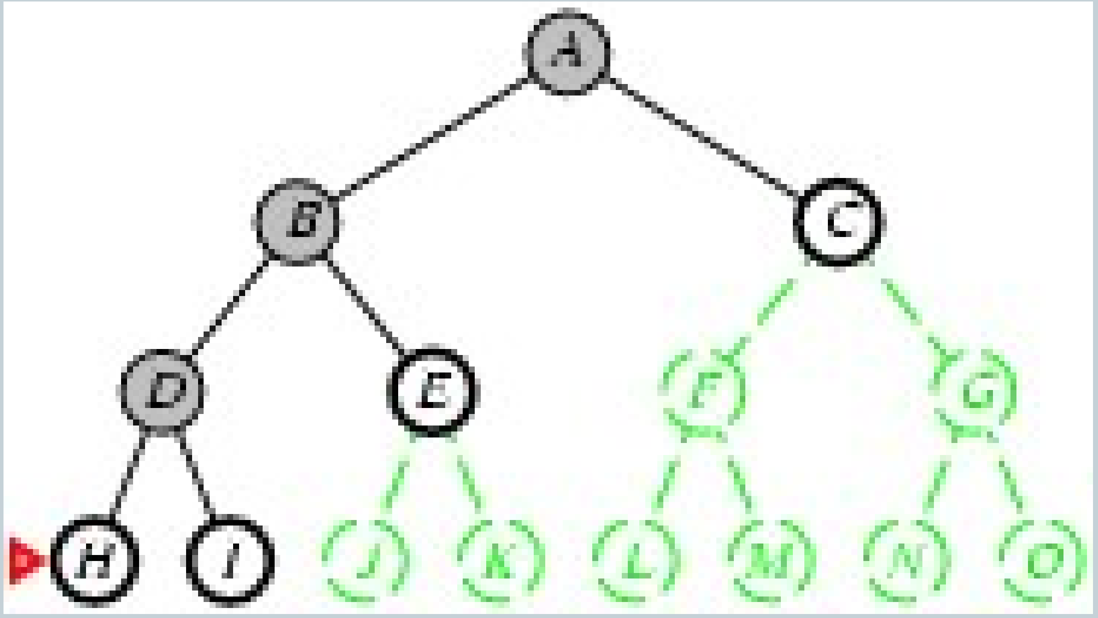
- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  -



Depth-first search

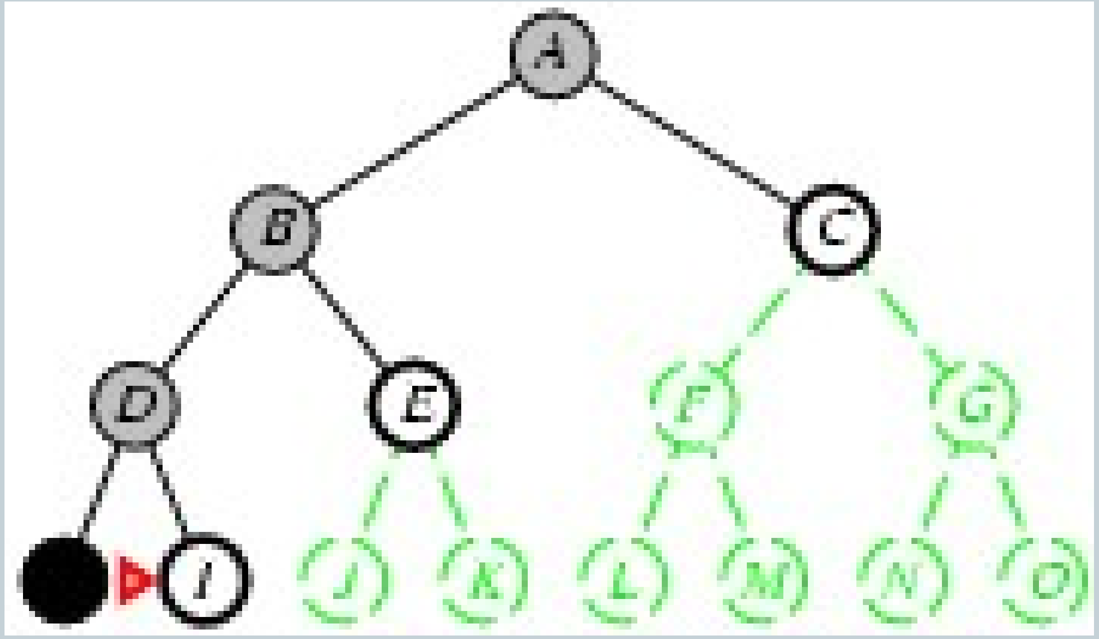
- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  - 

Depth-first search

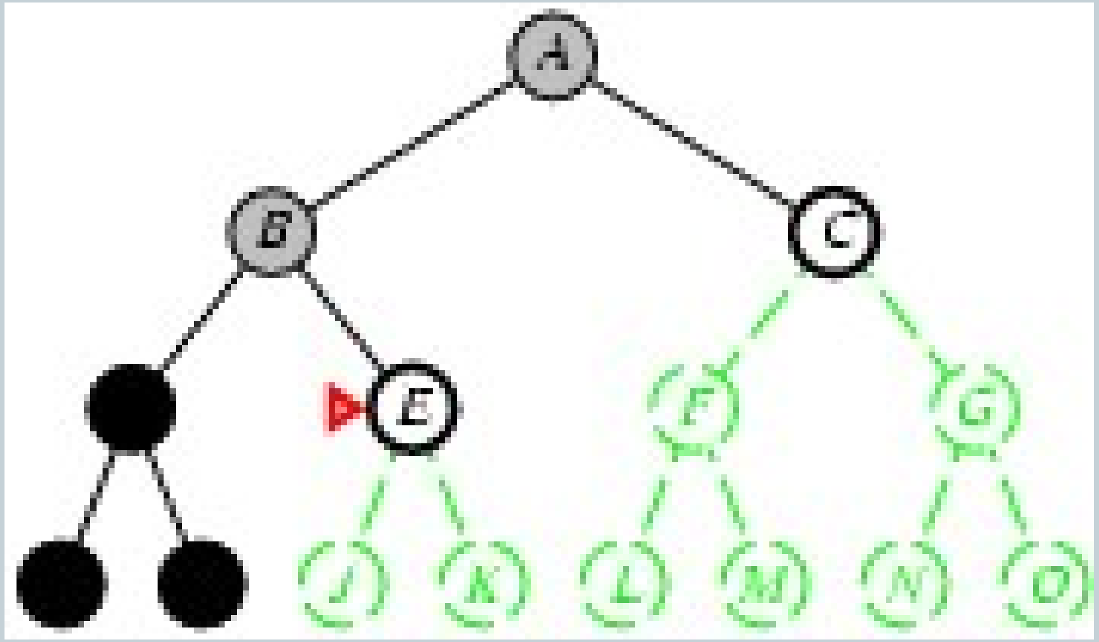
- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  - 

Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  -

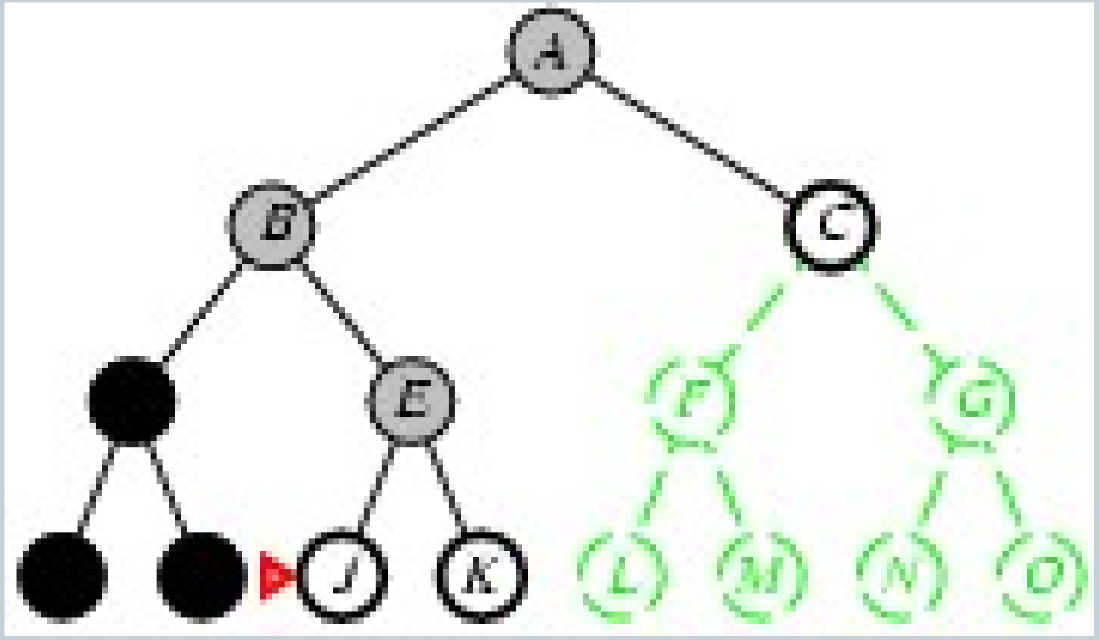


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  - 

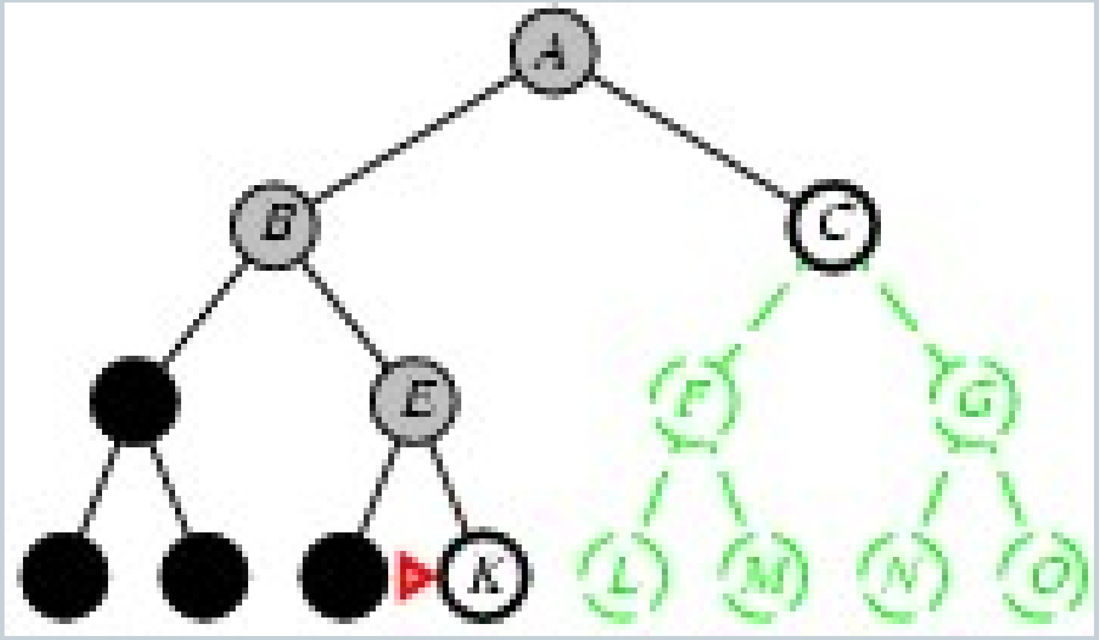
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  -

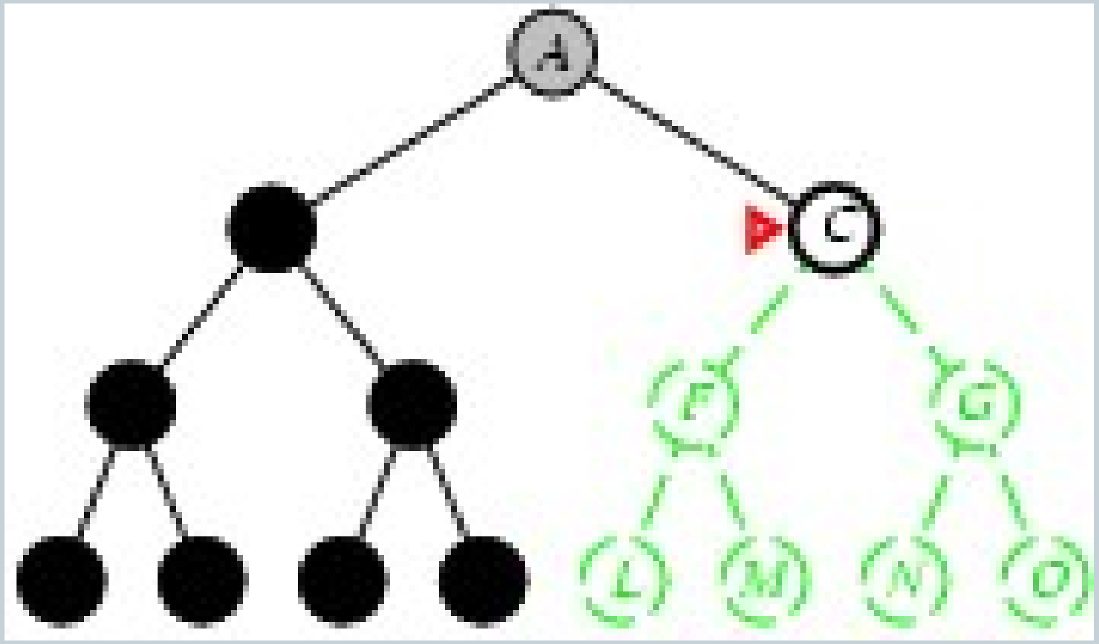


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  -

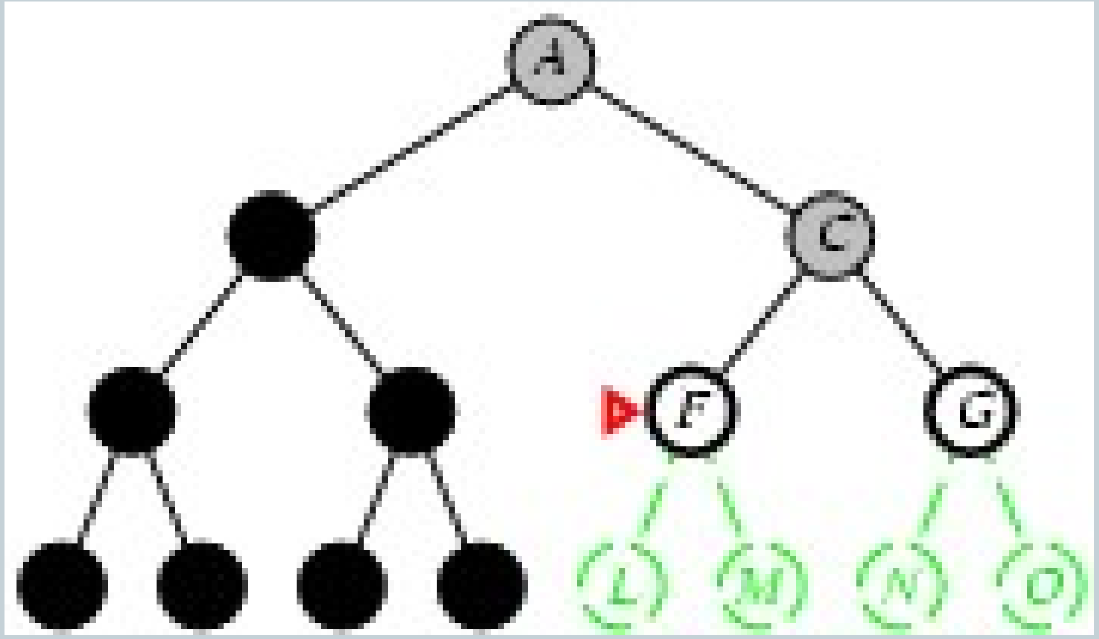


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  - 

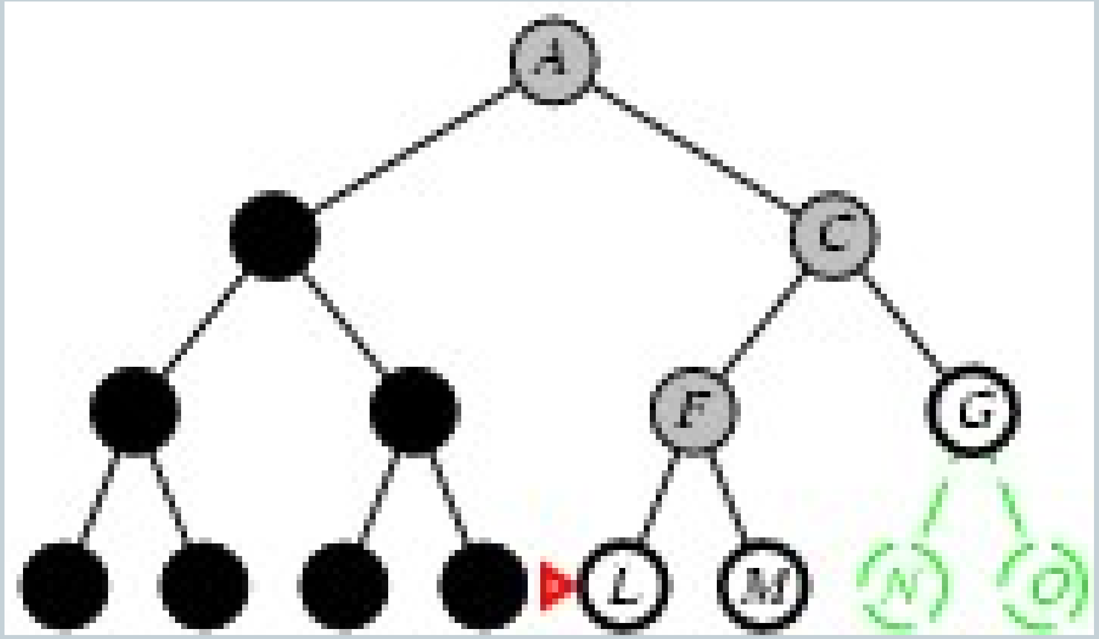
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  -

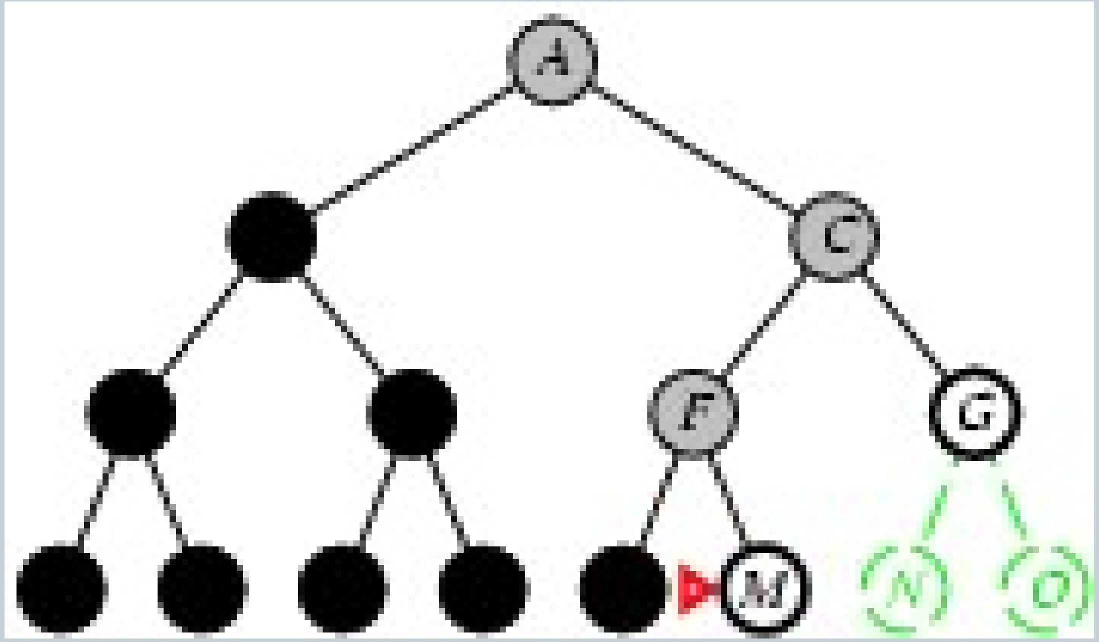


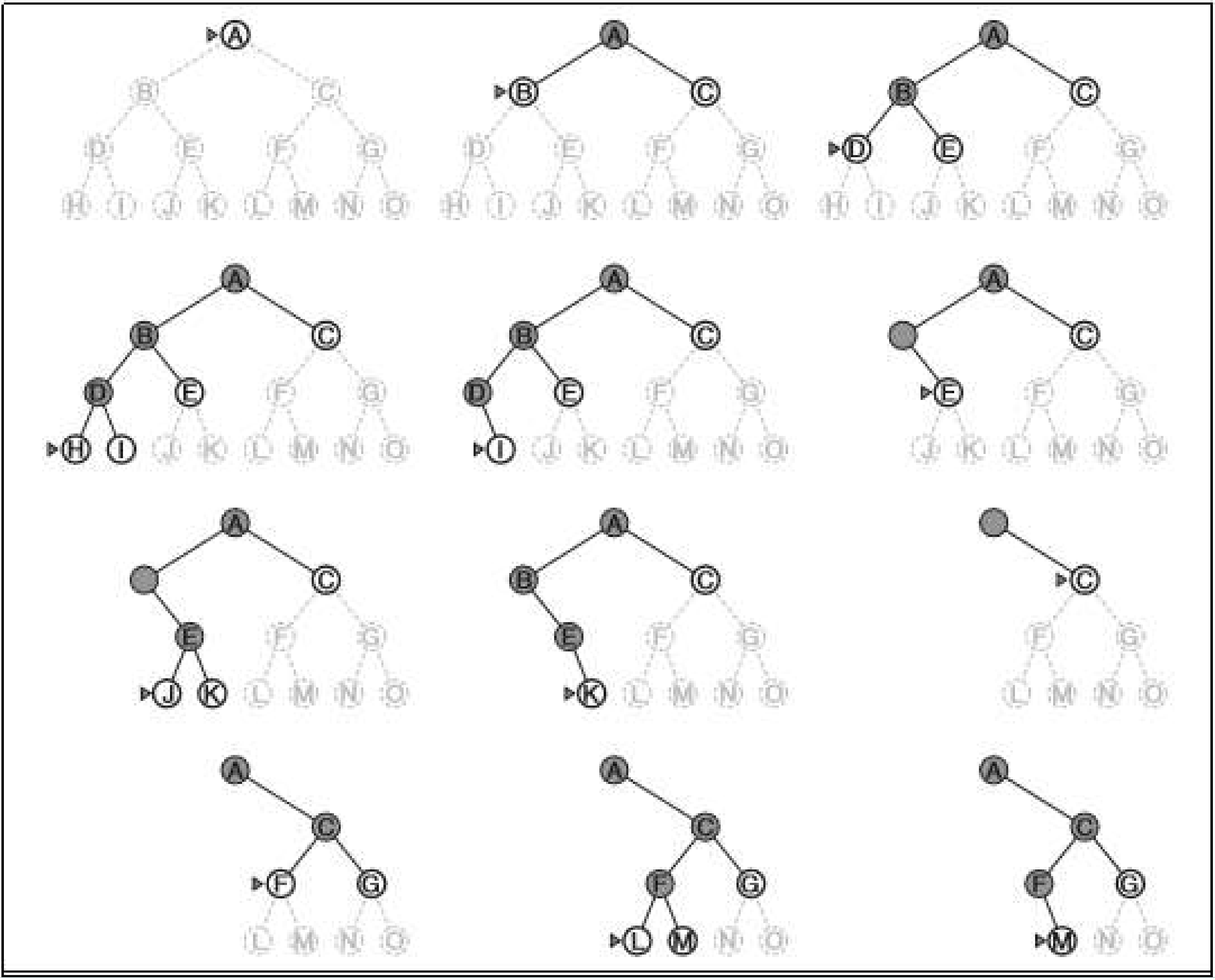
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  -



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - LIFO queue, i.e., put successors at front
  - 


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  - → 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

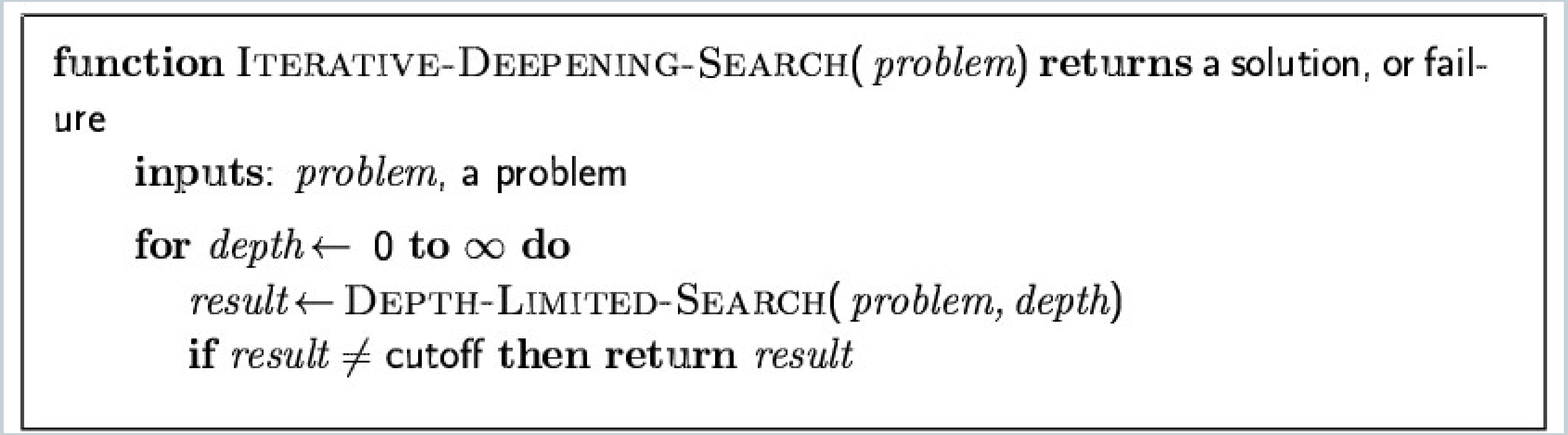
Space Complexity

- So far, depth-first search seems to have no clear advantage over breadth-first search, so why do we include it?
- The reason is the space complexity.
- For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.
- For a state space with branching factor b and maximum depth m, depth-first search requires storage of only **O(bm)** nodes.

Depth-limited search

- The failure of depth-first search in infinite state spaces is removed by supplying depth-first search with a predetermined depth limit l.
- Nodes at depth l are treated as if they have no successors.
- Depth-first search with depth limit *l*
- Introduces an additional source of incompleteness if we choose l<d, that is, the shallowest goal is beyond the depth limit.
- Will also be nonoptimal if we choose l >d.
- time complexity is $O(b^l)$; space complexity is O(bl)

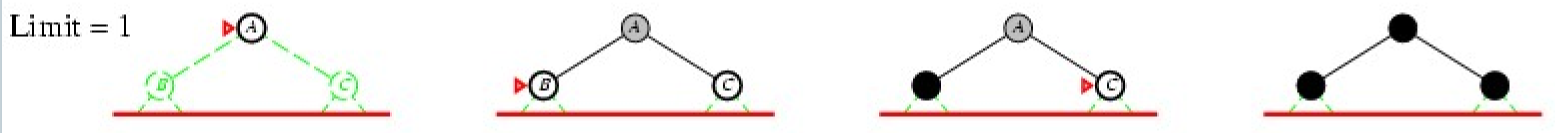
Iterative deepening search



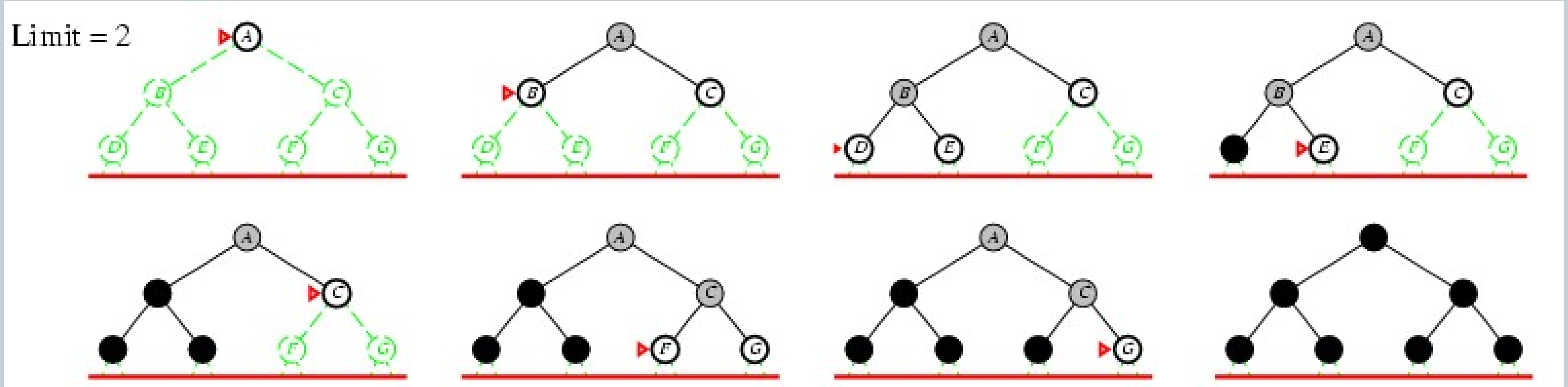
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening search l =0

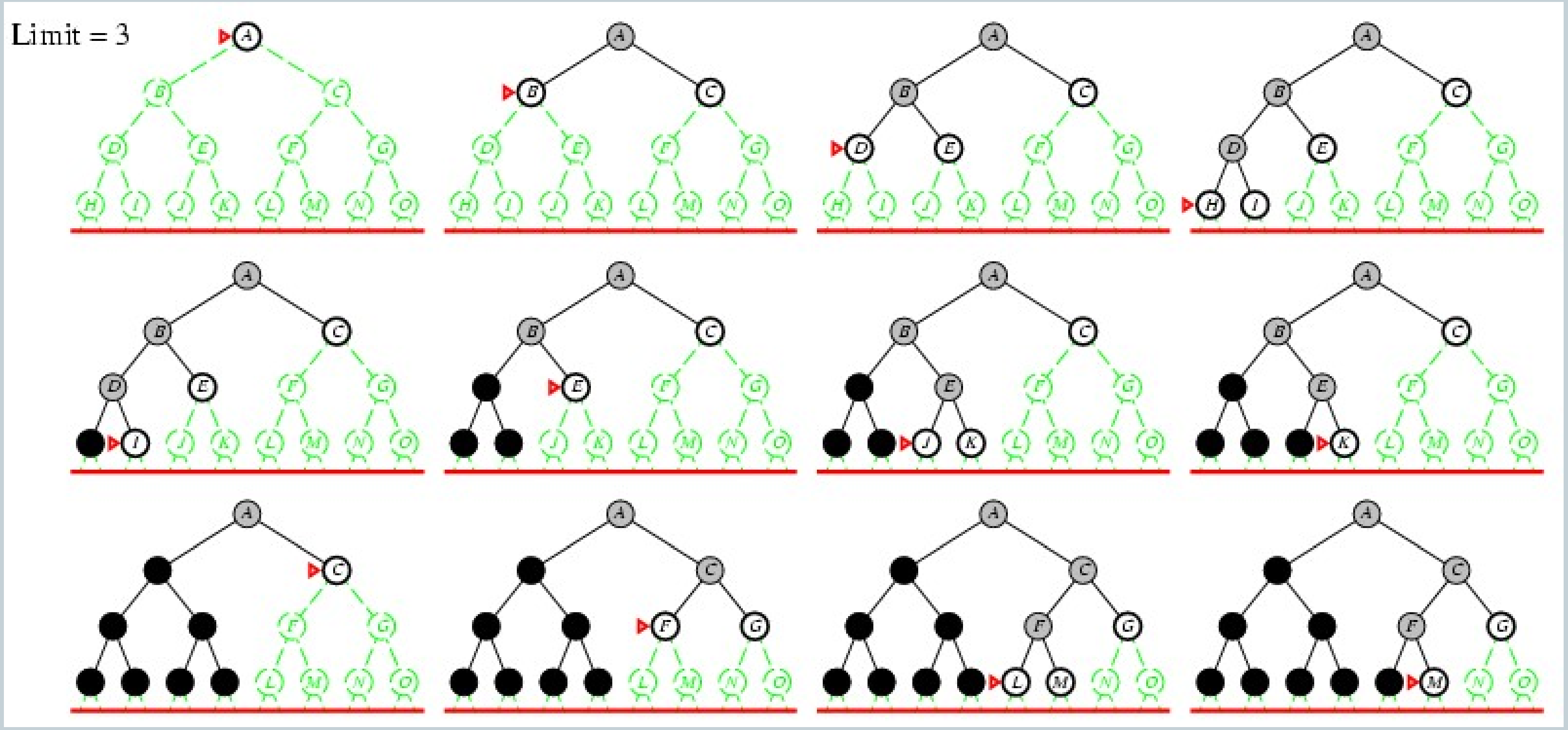
Iterative deepening search l =1



Iterative deepening search *l* =2



Iterative deepening search *l* =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

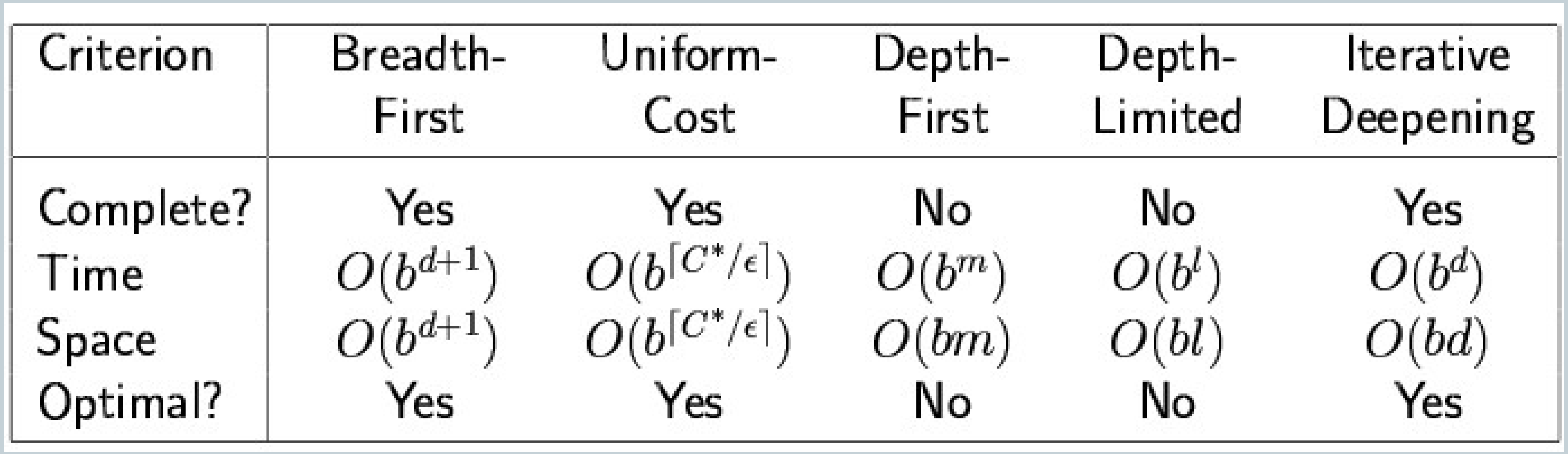
$$N_{IDS} = (d+1)b^0 + d\ b\hat{}^1 + (d-1)b\hat{}^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
- $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
- $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

Properties of iterative deepening search

- Complete? Yes
- Time? $(d)b + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$
- the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1
- In general, iterative deepening is the **preferred uninformed search method** when the search space is large and the depth of the solution is not known.

Summary of algorithms



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

INFORMED (HEURISTIC) SEARCH STRATEGIES

- GENERIC BEST FIRST SEARCH
- GREEDY BEST FIRST SEARCH
- A* SEARCH

Informed Search

- Informed search strategy
- one that uses problem-specific knowledge beyond the definition of the problem itself
- can find solutions more efficiently than can an uninformed strategy
- **Best-first search** is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which *a node is selected for expansion based on an evaluation function, f(n).*

Evaluation function

- The **evaluation function** is construed as a cost estimate, so the node with the lowest evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of ***f*** instead of ***g*** to order the priority queue.
- The choice of ***f*** determines the search strategy.

Heuristic Function

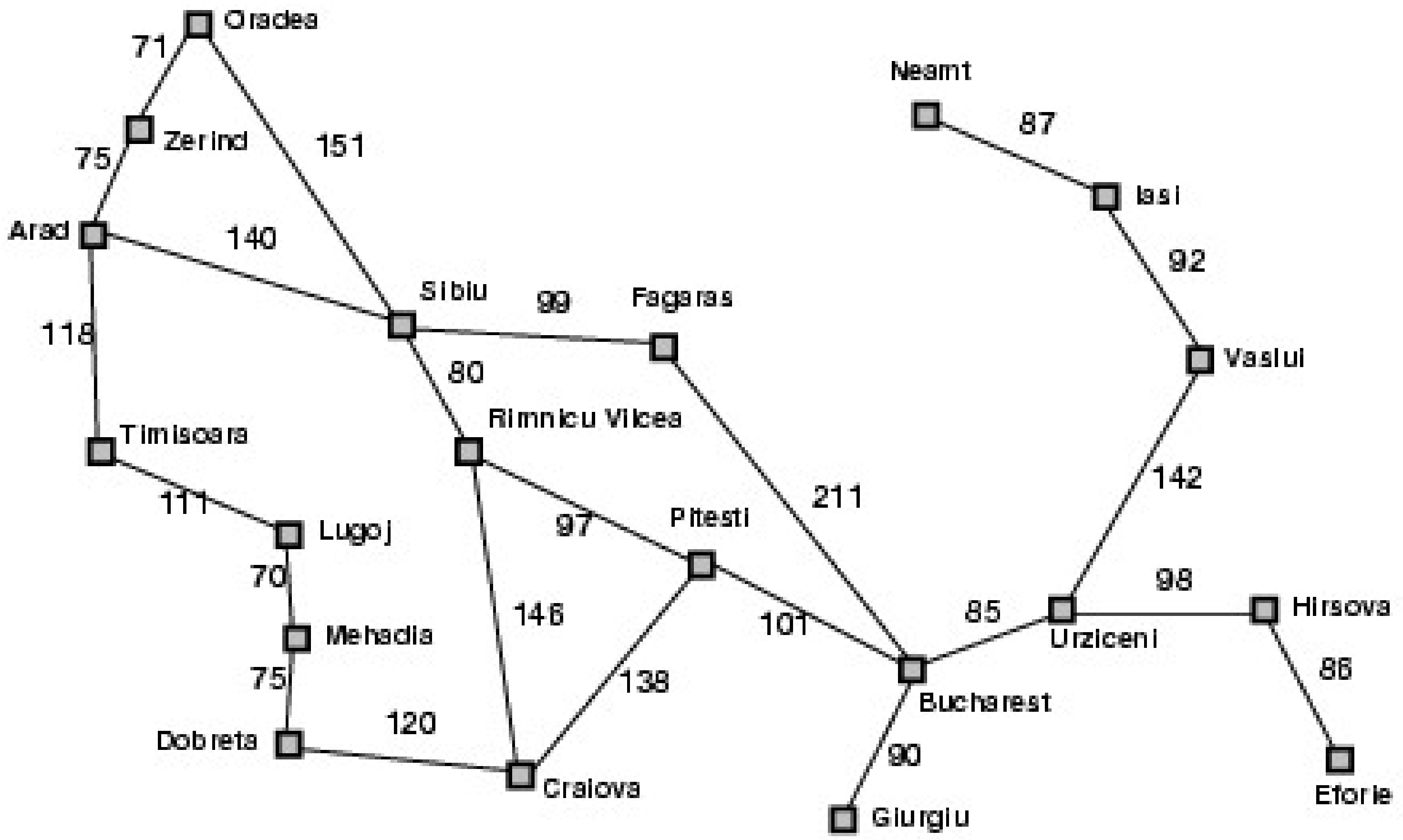
- Most best-first algorithms include as a component of ***f*** a heuristic function, denoted **h(n):**
- **h(n)** = estimated cost of the cheapest path from the state at node n to a goal state.
- h(n) takes a node as input, but, unlike g(n), it depends only on the state at that node.
- For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.
- Heuristic functions are considered as arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then h(n)=0.

Greedy best first search

- Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n)=h(n)$.
- Consider the route-finding problems in Romania; use the straightline distance heuristic, (h_{SLD}).
- If the goal is Bucharest, know the straight-line distances to Bucharest.

Romania with step costs in km



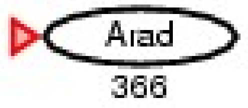
Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

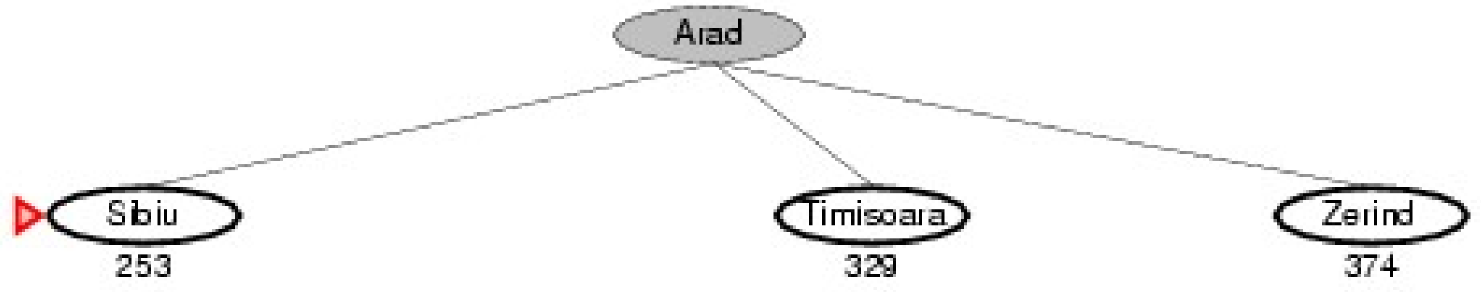
Greedy best-first search

- $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal
- The values of $\mathrm{h}_{\mathrm{SLD}}$ cannot be computed from the problem description itself.
- It takes a certain amount of experience to know that $\mathrm{h}_{\mathrm{SLD}}$ is correlated with actual road distances and is, therefore, a useful heuristic.

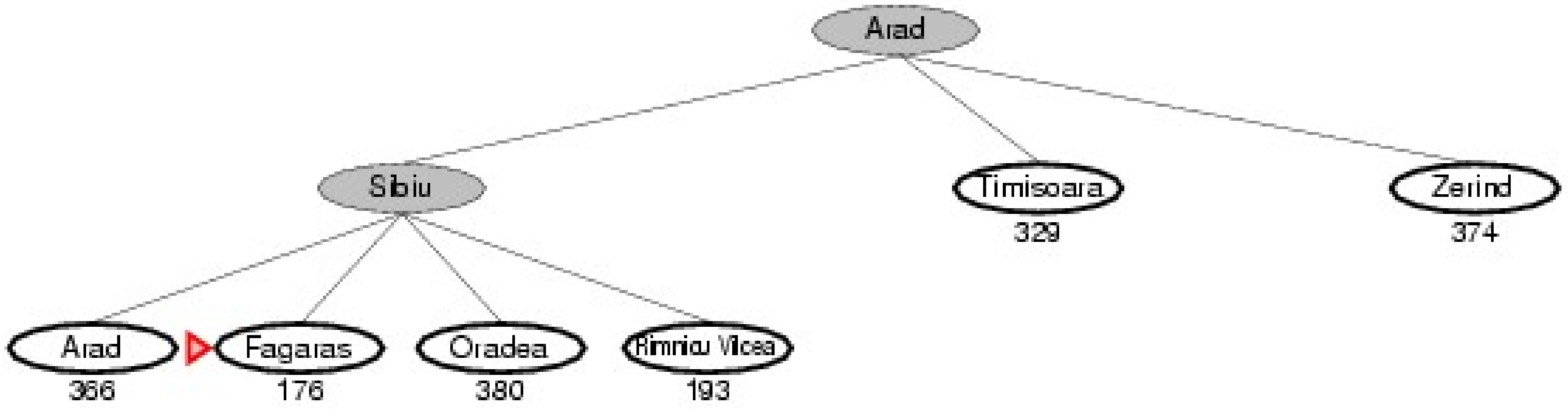
Greedy best-first search example



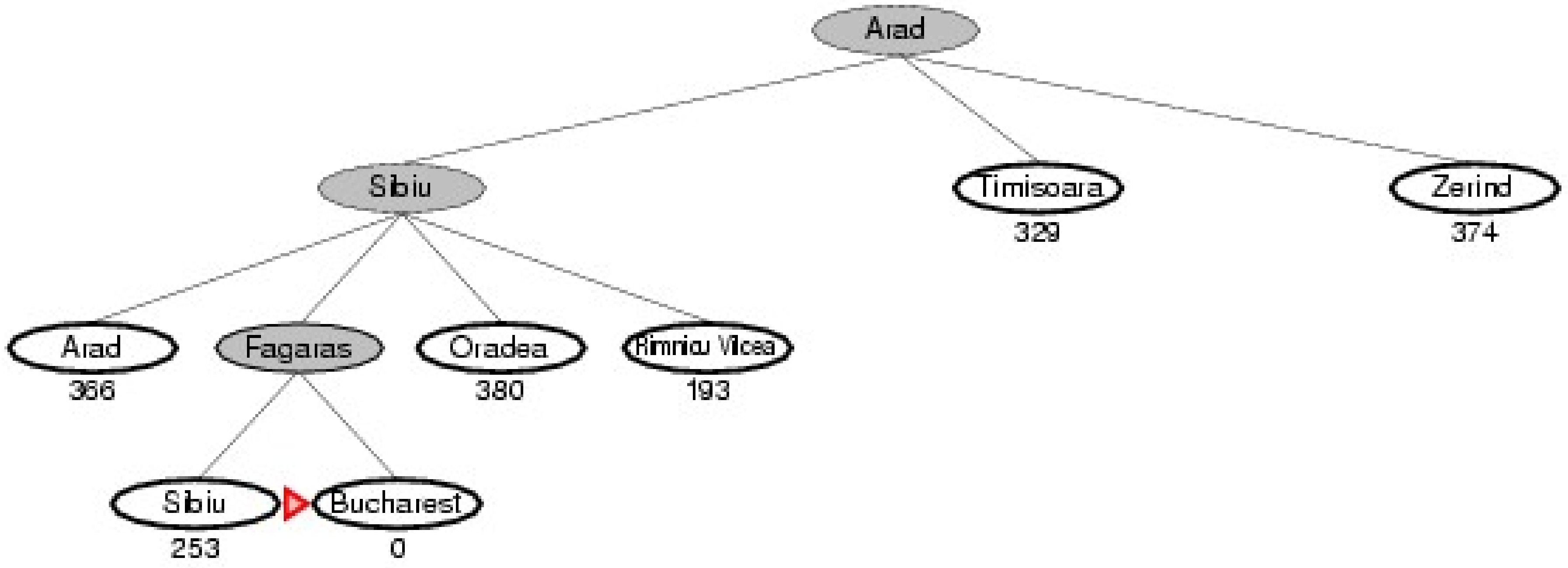
Greedy best-first search example



Greedy best-first search example



Greedy best-first search example



Properties of greedy best-first search

- It is **not optimal,** however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.
- Shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.
- Also **incomplete** even in a finite state space, much like depth-first search.
- Consider the problem of getting from Iasi to Fagaras.

Properties of greedy best-first search

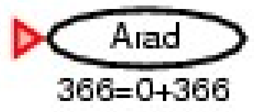
- Complete? No – can get stuck in loops, e.g., Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

A^* search (A star search)

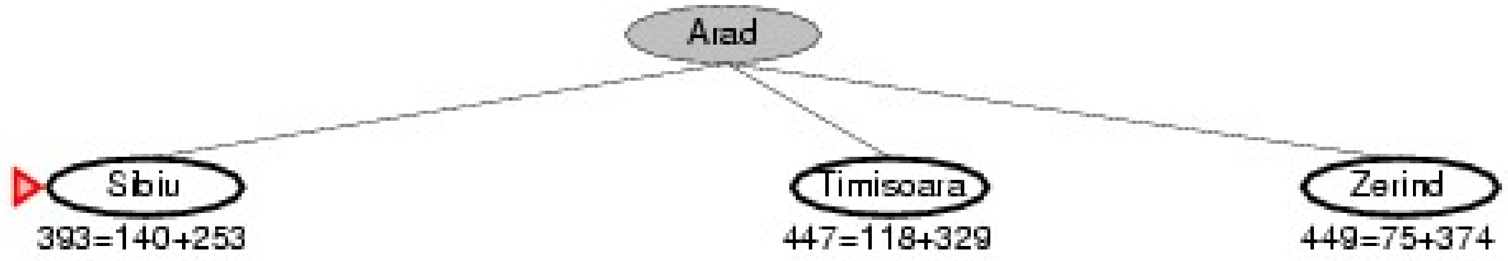
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n)+h(n)$.
- It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, **A* search is both complete and optimal**.
- The algorithm is identical to UNIFORM-COST-SEARCH except that A^* search uses $g + h$ instead of g.

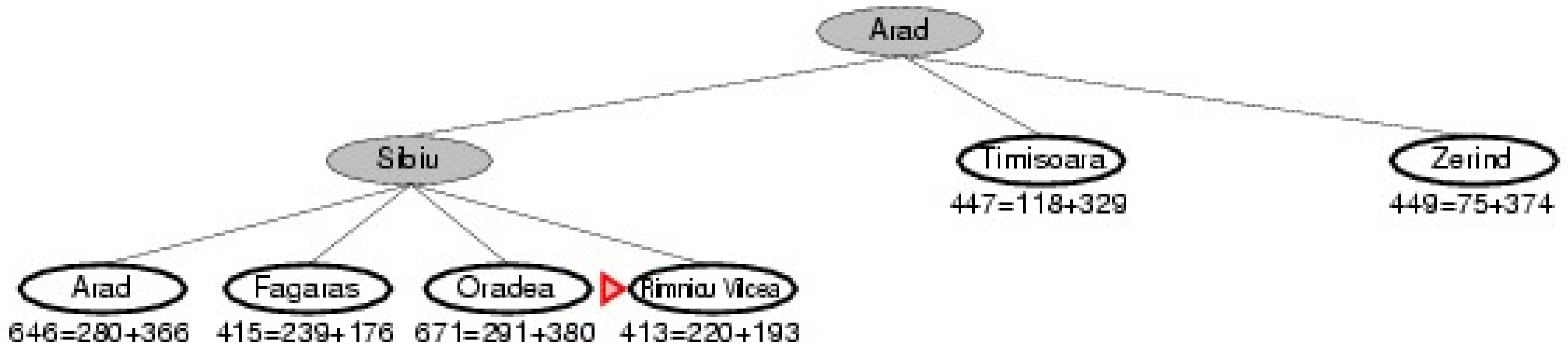
A* search example



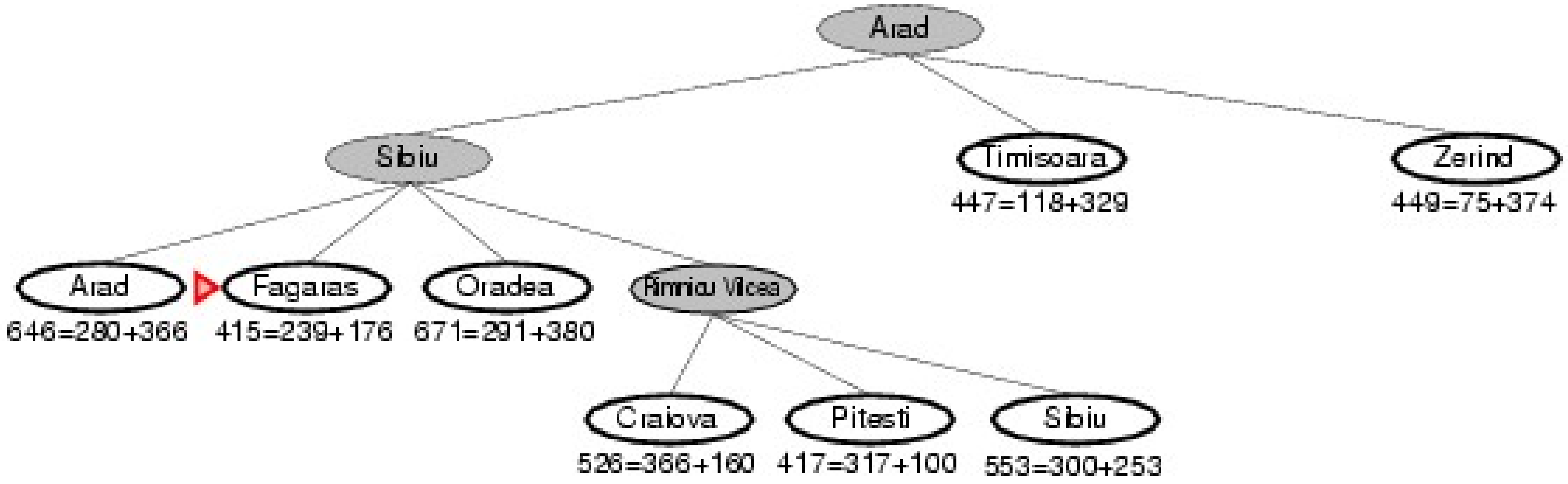
A* search example



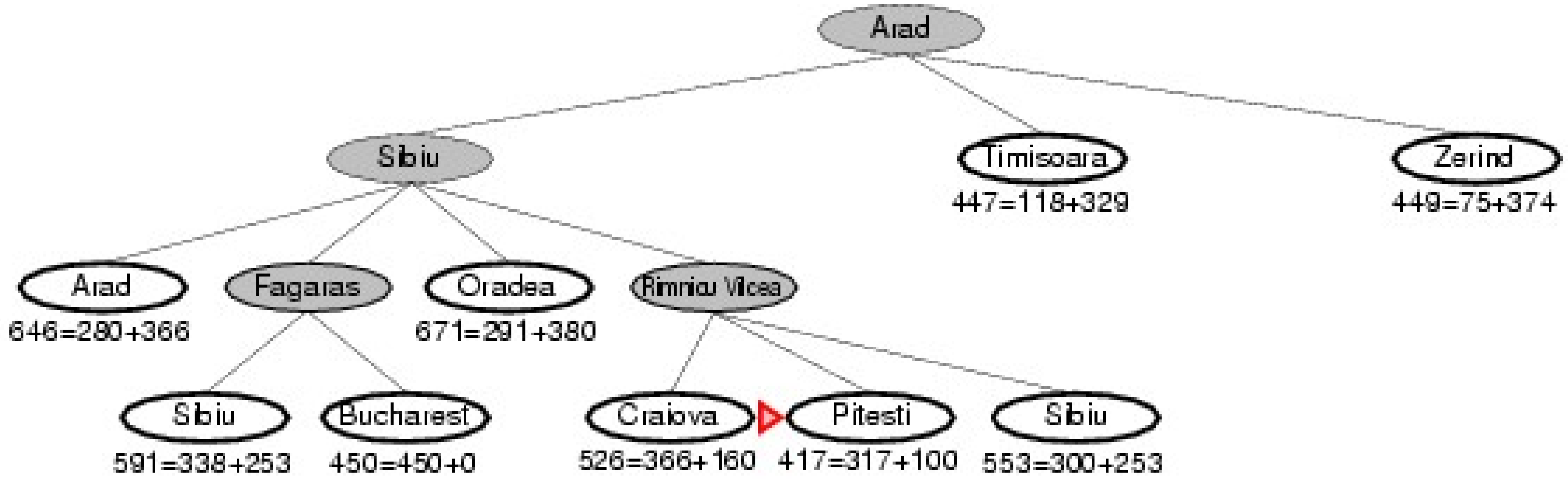
A* search example



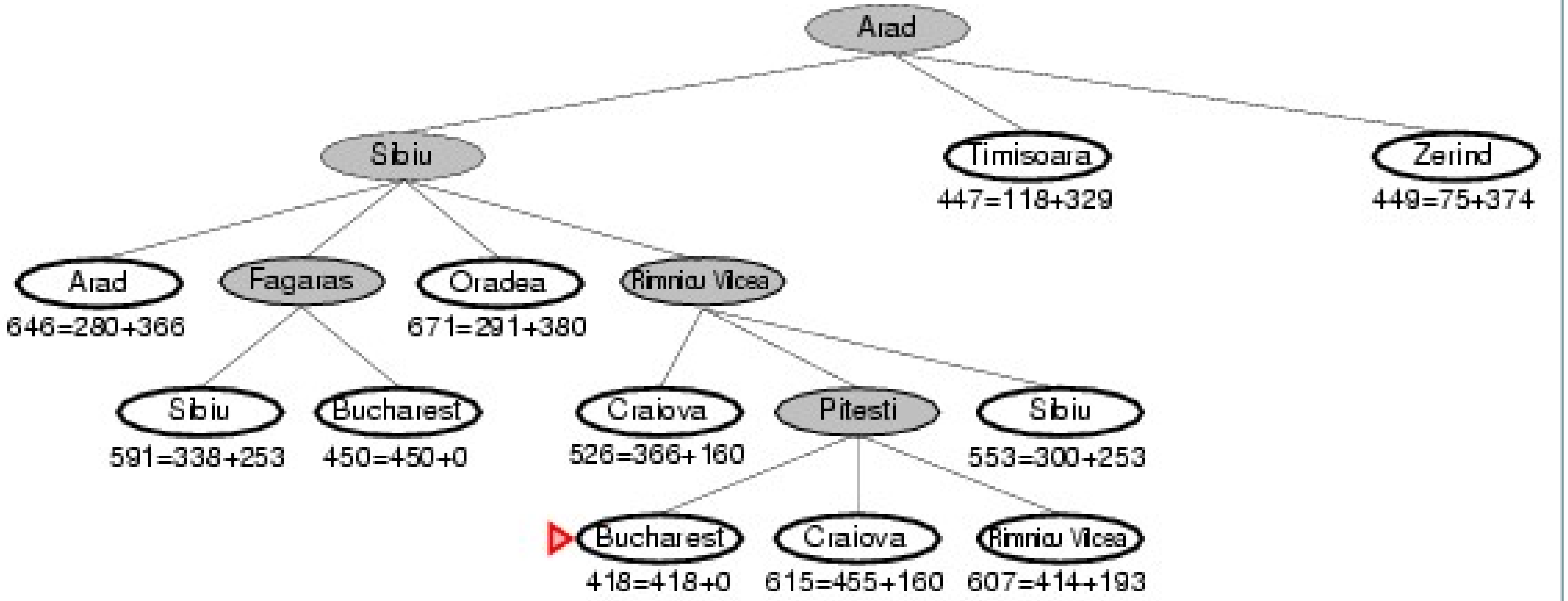
A^* search example



A* search example



A* search example



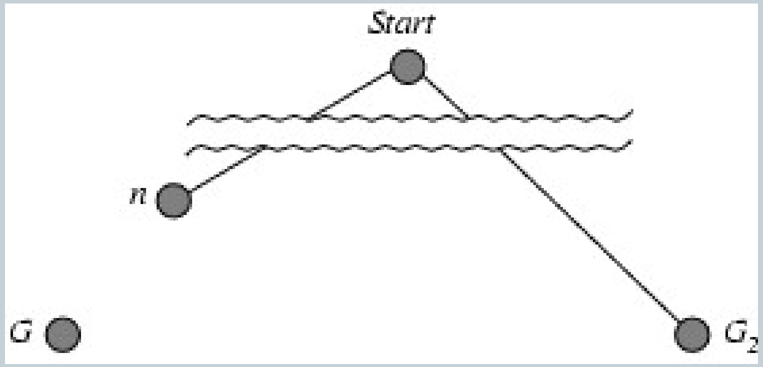
Admissible heuristics

- Conditions for optimality: Admissibility and consistency
- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic.
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance).

Optimal - admissible

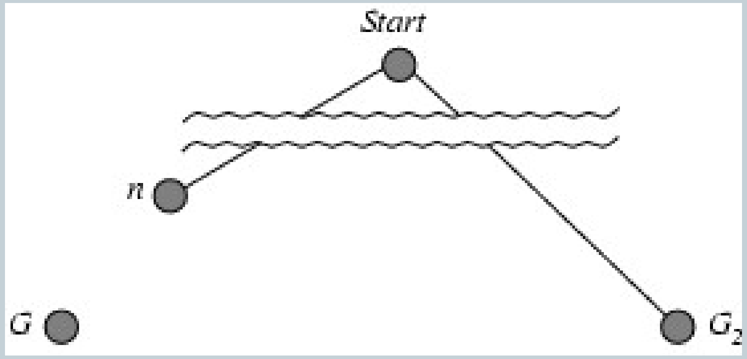
- Theorem: If *h(n)* is admissible, A* using `TREE-SEARCH` is optimal.
- Proof follows
- Recall
- Evaluation function *f(n)* = *g(n)* + *h(n)*
- *g(n)* = cost so far to reach *n*
- *h(n)* = estimated cost from *n* to goal
- *f(n)* = estimated total cost of path through *n* to goal

Optimality of A* (proof)

- Suppose some **suboptimal goal $\boldsymbol{G_2}$** has been generated and is in the frontier. Let *n* be an unexpanded node in the frontier such that *n* is on a shortest path to an **optimal goal *G***.
- 

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

Optimality of A* (proof)

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

• $f(G_2)$	$> f(G)$	from above
• $h(n)$	$\le h^*(n)$	since *h* is admissible
• $g(n) + h(n)$	$\le g(n) + h^*(n)$	
• $f(n)$	$\le f(G)$	

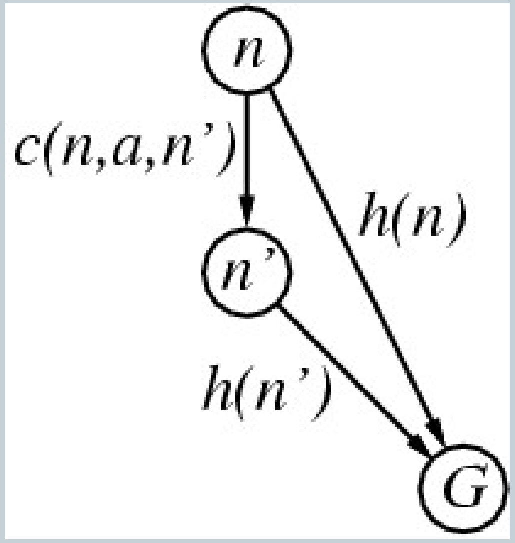
- Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion

CONSISTENCY

- A second, slightly stronger condition called consistency (or sometimes monotonicity) is required only for applications of A* to graph search.
- A heuristic *h(n)* is consistent if, for every node *n* and every successor *n′* of *n* generated by any action *a*, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to *n′* plus the estimated cost of reaching the goal from *n′*:
  - $h(n) \leq c(n,a,n')+h(n')$.
- This is a form of the general triangle inequality.

Consistent heuristics

- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,
- $h(n) \le c(n,a,n') + h(n')$
- If h is consistent, we have
- $$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\ge g(n) + h(n) \\ &= f(n) \end{aligned}$$



- i.e., $f(n)$ is non-decreasing along any path.

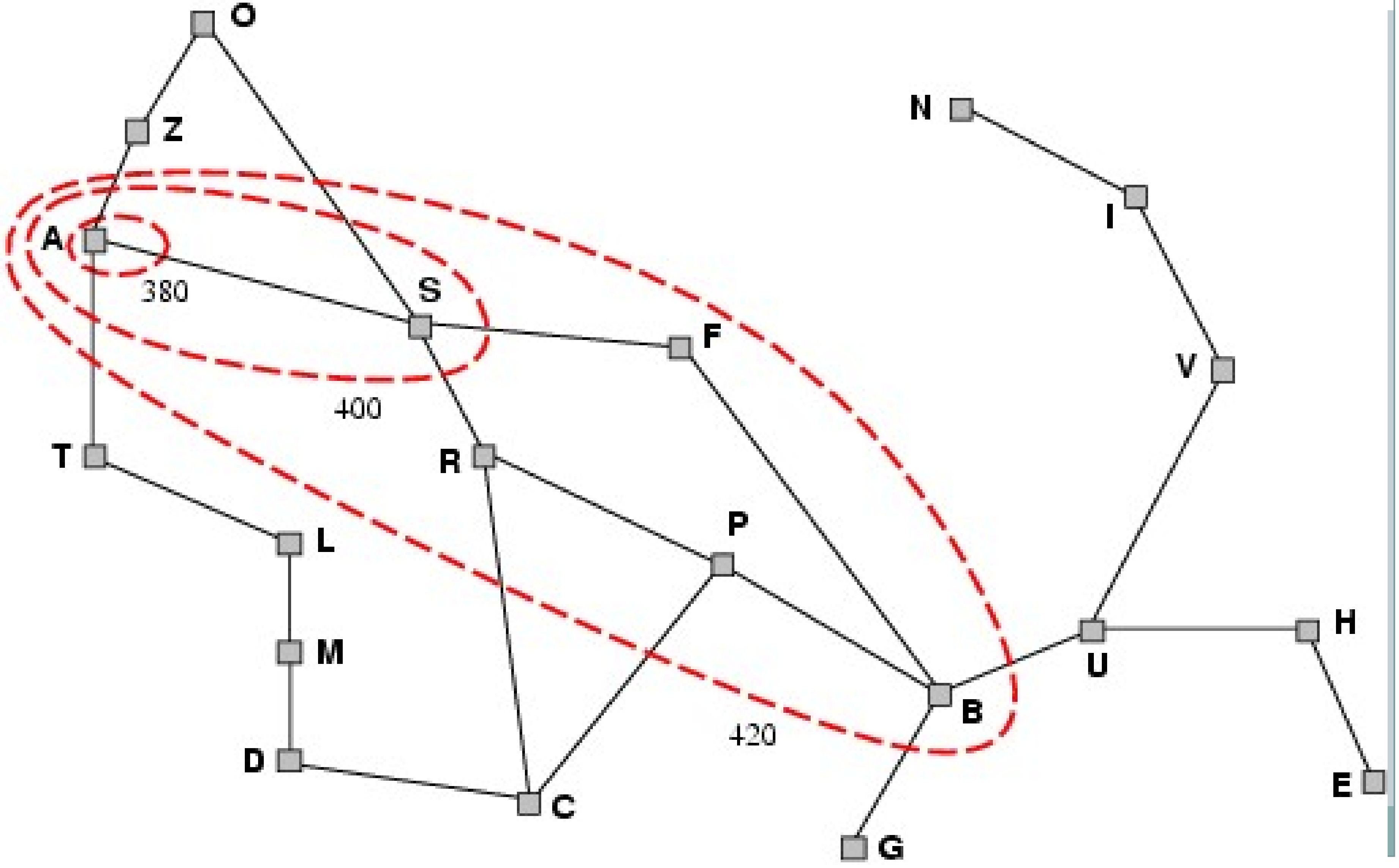
- Theorem: If $h(n)$ is consistent, A^* using `GRAPH-SEARCH` is optimal.
- Proof: If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing.
- The proof follows directly from the definition of consistency.
- Suppose n' is a successor of n; then $g(n') = g(n) + c(n,a,n')$ for some action a, and we have
- $f(n')=g(n')+h(n') = g(n)+c(n,a,n')+h(n') \geq g(n)+h(n)=f(n)$.

- To prove that whenever *A** selects a node n for expansion, the optimal path to that node has been found.
- f is nondecreasing along any path, n' would have lower f-cost than n and would have been selected first.
- It follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of f(n).
- Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have h=0) and all later goal nodes will be at least as expensive.
- The fact that f-costs are nondecreasing along any path also means that we can draw contours in the state space, just like the contours in a topographic map.

Optimality of A*

- A* expands nodes in order of increasing f value
- The fact that f-costs are non decreasing along any path also means that we can draw contours in the state space, just like the contours in a topographic map.
- Inside the contour labelled 400, all nodes have f(n) less than or equal to 400, and so on.
- A∗ expands the frontier node of lowest f-cost, we can see that an A∗ search fans out from the start node, adding nodes in concentric bands of increasing f-cost.
- Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$

Example of Romanian Map



Properties of A* search

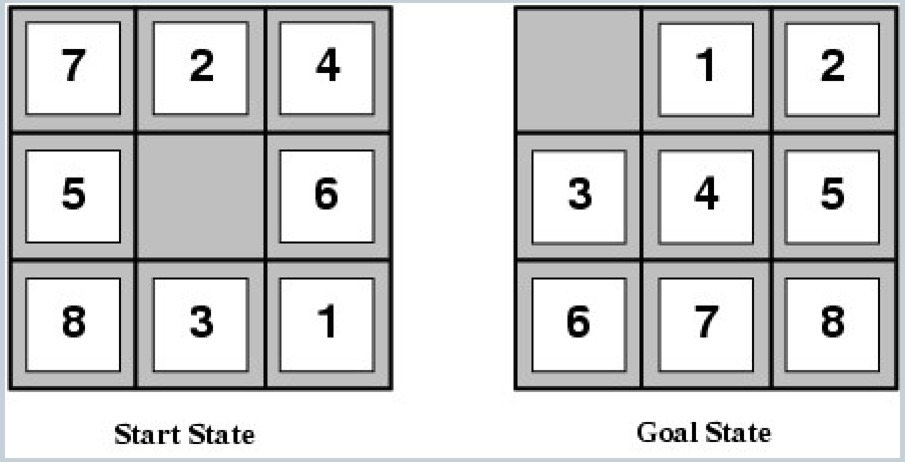
- Complete? Yes (unless there are infinitely many nodes with f ≤ *f(G)*)
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes

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)



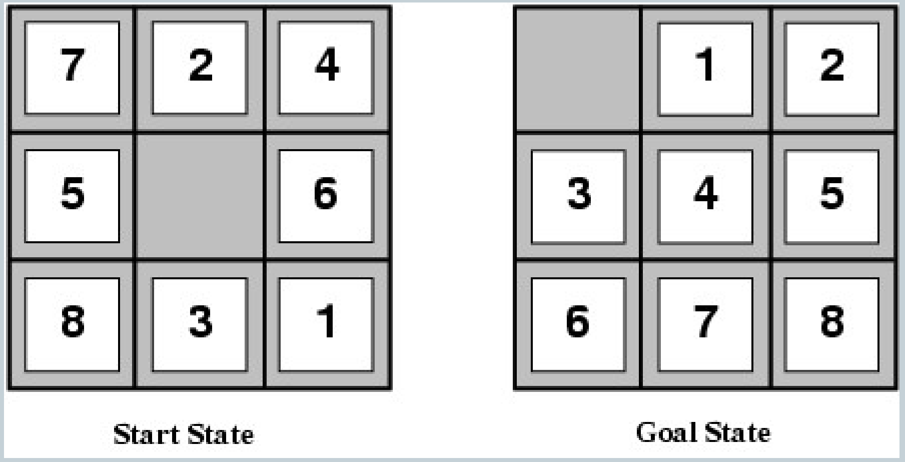
- $h_1(S)$ = ?
- $h_2(S)$ = ?

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)



- $h_1(S)$ = ? 8
- $h_2(S)$ = ? 3+1+2+2+2+3+3+2 = 18

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 dominates h_1
- h_2 is better for search
-
- Typical search costs (average number of nodes expanded):
- $d=12$ IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes
- $d=24$ IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original 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

Summary

- **Informed search methods** may have access to a heuristic function *h(n)* that estimates the cost of a solution from n.
- The **generic best-first search** algorithm selects a node for expansion according to an evaluation function.
- **Greedy best-first search** expands nodes with minimal h(n). It is not optimal but is often efficient.
- **A∗ search** expands nodes with minimal *f(n)=g(n)+h(n)*. A∗ is complete and optimal, provided that *h(n)* is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A∗ is still prohibitive.