# Solving Problems by Searching

# Two Types of Searching

- **Uninformed search** algorithms—algorithms that are given no information about the problem other than its definition
  - Examples: BFS, DFS, Uniform-Cost Search, Depth-Limited Search, Iterative deepening depth-first search, Bidirectional Search, etc.

- **Informed search** algorithms, on the other hand, can do quite well given some guidance on where to look for solutions
  - Examples: Greedy BFS, A* Search, iterative-deepening A* (IDA*), RBFS (recursive best-first search), SMA* (simplified memory-bounded A*), etc.

# Problem Solving Agent

**function** SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action
  **persistent**:

  *seq*, an action sequence, initially empty
  *state*, some description of the current world state
  *goal*, a goal, initially null
  *problem*, a problem formulation

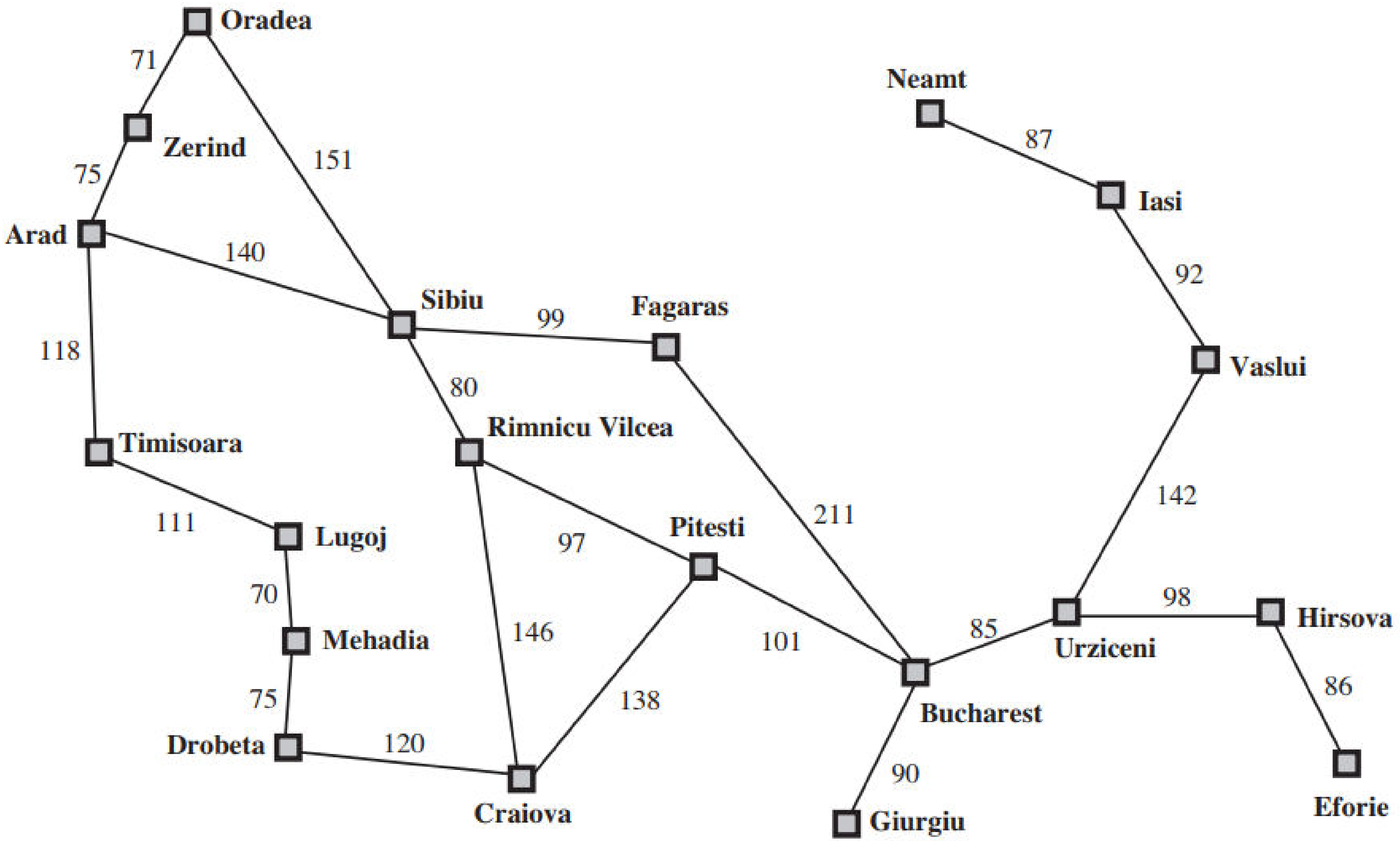
*state* ← UPDATE-STATE(*state*, *percept*)
**if** *seq* is empty **then**
  *goal* ← FORMULATE-GOAL(*state*)
  *problem* ← FORMULATE-PROBLEM(*state*, *goal*)
  *seq* ← SEARCH(*problem*)
  **if** *seq* = *failure* **then return** a null action
*action* ← FIRST(*seq*)
*seq* ← REST(*seq*)

**return** *action*

# Problem Solving Agent

- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
- 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**.

# A simplified road map of part of Romania



# Problem Definition

- A problem can be defined formally by five components:
- **Initial State:** The **initial state** that the agent starts in.
  - For example, the initial state for our agent in Romania might be described as *In(Arad)*.

- **Action**: 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*.
  - For example, from the state *In(Arad)*, the applicable actions are {*Go(Sibiu)*, *Go(Timisoara)*, *Go(Zerind)*}.

# Problem Definition

- **Transition Model:** A description of what each action does 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 to refer to any state reachable from a given state by a single action.
  - For example, RESULT(*In(Arad)*, *Go(Zerind)*) = *In(Zerind)* .

  - 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
  - The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions.
  - A **path** in the state space is a sequence of states connected by a sequence of actions.

# Problem Definition

- **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)*}.

- **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.
  - For the agent trying to get to Bucharest, the cost of a path might be its length in kilometers.

# Vacuum World

- This can be 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 $n \times 2^n$ 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.

# Vacuum World

- This can be formulated as a problem as follows:
  - **Transition model:** The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.

  - **Goal test:** This 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.

# Vacuum World

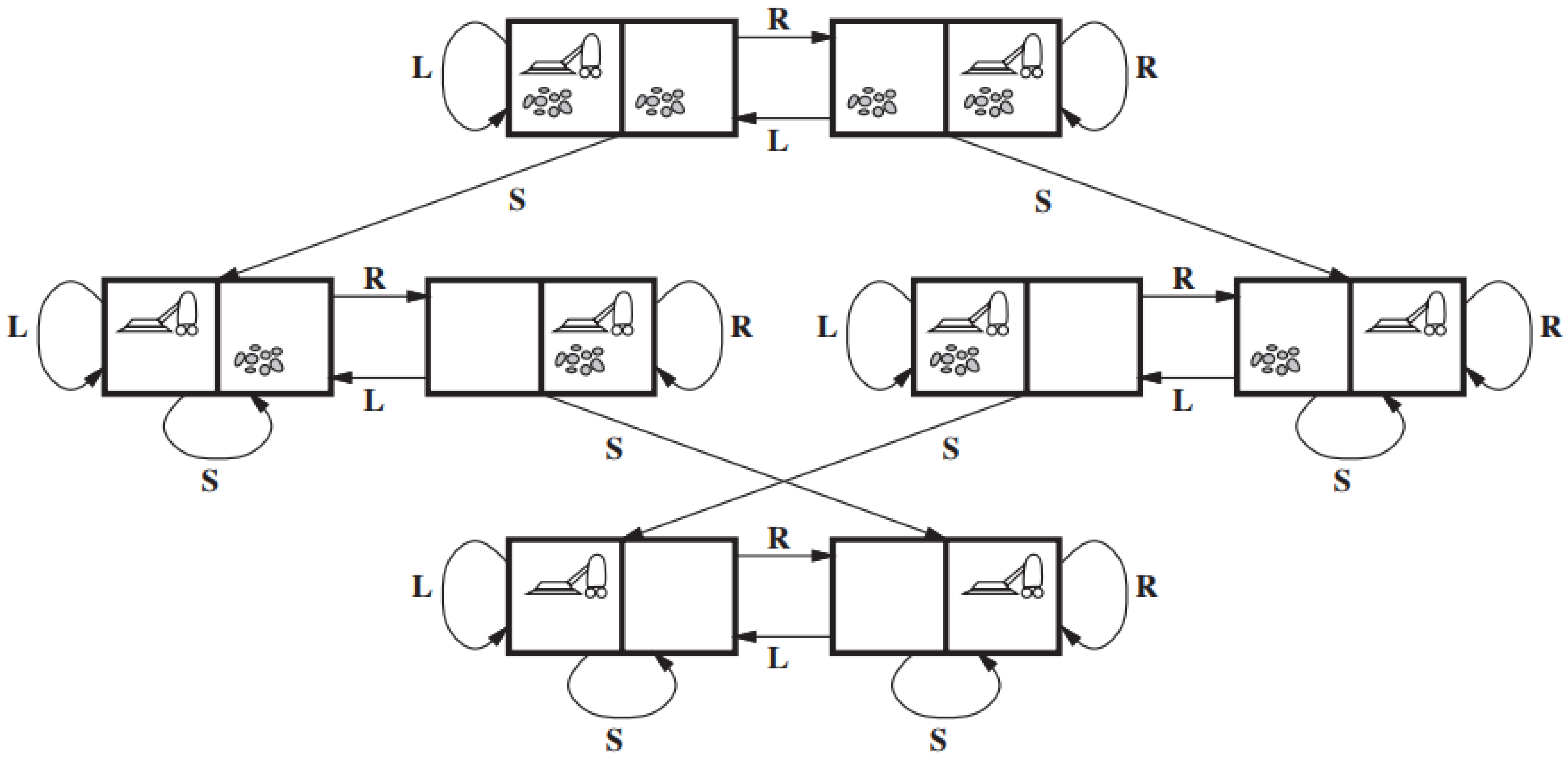
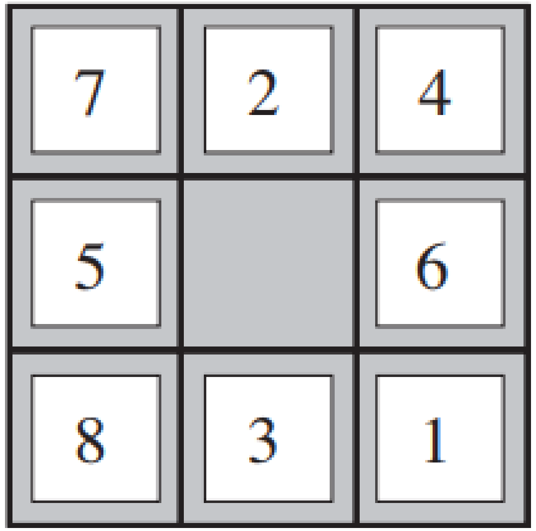
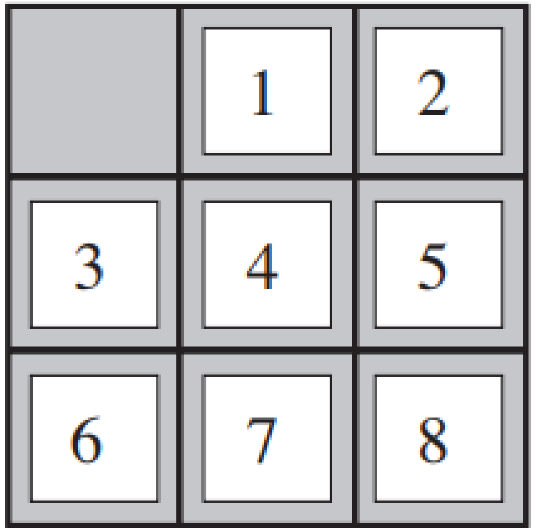



**Figure:** The state space for the vacuum world. Links denote actions: L = Left, R = Right, S = Suck.

# 8-puzzle




Start State




Goal State

# 8-puzzle

- The standard problem formulation is as follows:
  - **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

  - **Initial state:** Any state can be designated as the initial state.
    - Note that any given goal can be reached from exactly half of the possible initial states.

  - **Actions:** The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.

# 8-puzzle

- The standard problem formulation is as follows:
  - **Transition model:** Given a state and action, this returns the resulting state.
    - For example, if we apply Left to the start state (See in Figure), the resulting state has the 5 and the blank switched.
  - **Goal test:** This checks whether the state matches the goal configuration.
  - **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

# Solution

- 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.

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

# Solve a Problem

- 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.
- Then **expand** the current state; that is, apply each legal action to the current state, thereby generate a new set of states.
- 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.
- There may be **repeated state** in the search tree.

# Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree.
- For each node n of the tree, we have a structure that 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.
- The appropriate data structure for this is a **queue:** FIFO queue, LIFO queue and Priority queue

# Measuring problem-solving performance

- We can evaluate an algorithm’s performance in four ways:
  - **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?

# Measuring problem-solving performance

- In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite.
- For these reasons, 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.
- Space in terms of the maximum number of nodes stored in memory.

# Measuring problem-solving performance

- To assess the effectiveness of a search algorithm, we can consider one of the following:
- **Search cost:** which typically depends on the time complexity but can also include a term for memory usage.
- **Total cost:** which combines the search cost and the path cost of the solution found.

# Uninformed Search

- Uninformed search (also called blind search)
- The strategies have no additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- All search strategies are distinguished by the order in which nodes are expanded.
- Examples: BFS, DFS, Uniform-Cost Search, Depth-Limited Search, Iterative deepening depth-first search, Bidirectional Search, etc (**self study**)

# Informed (Heuristic) Search

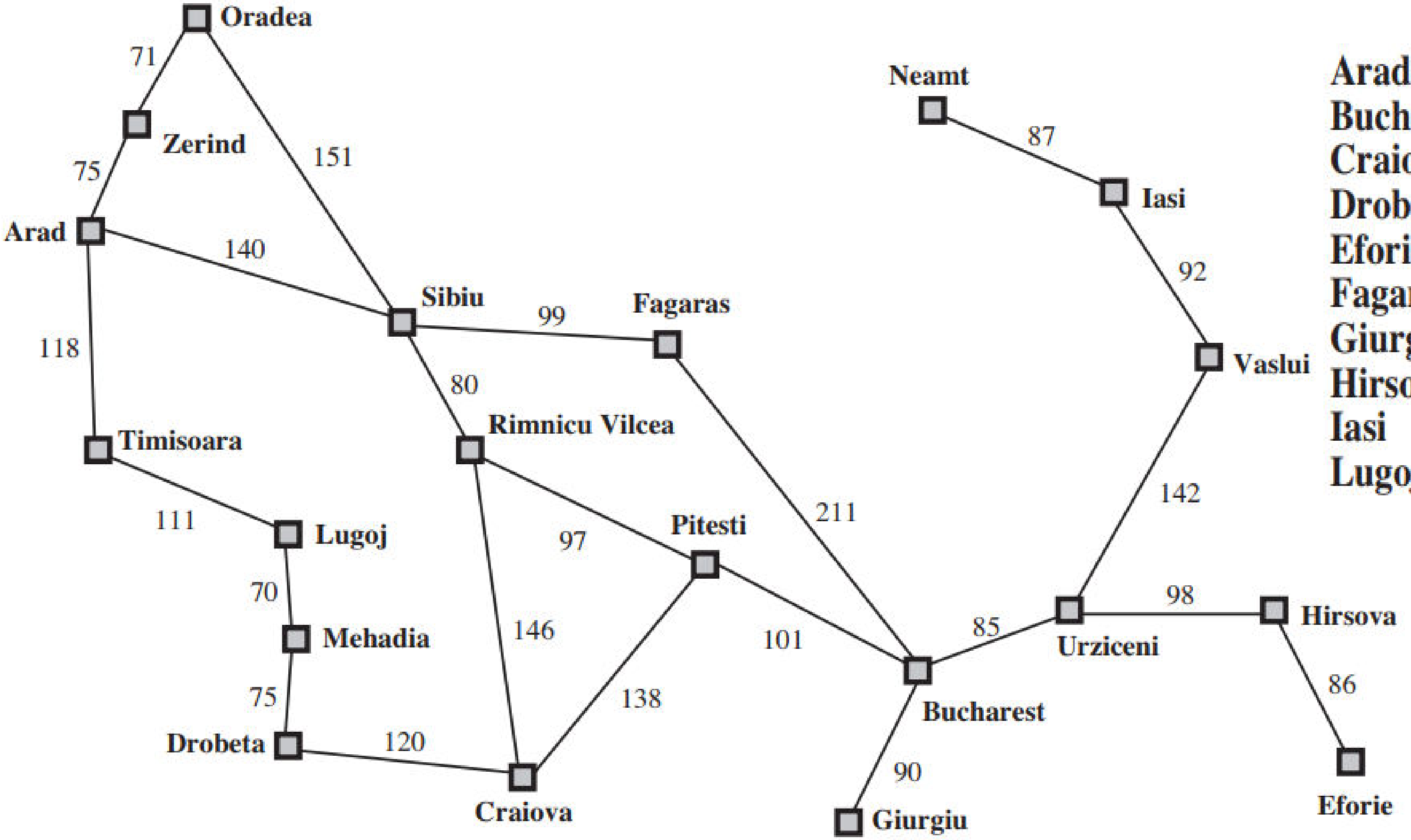
- **Informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself
- Can find solutions more efficiently than can an uninformed strategy.

- A **heuristic function**, denoted $h(n)$:
  - $h(n)$ = estimated cost of the cheapest path from the state at node $n$ to a goal state
  - Example: h(Arad) = the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest (Goal).

# Greedy best-first search

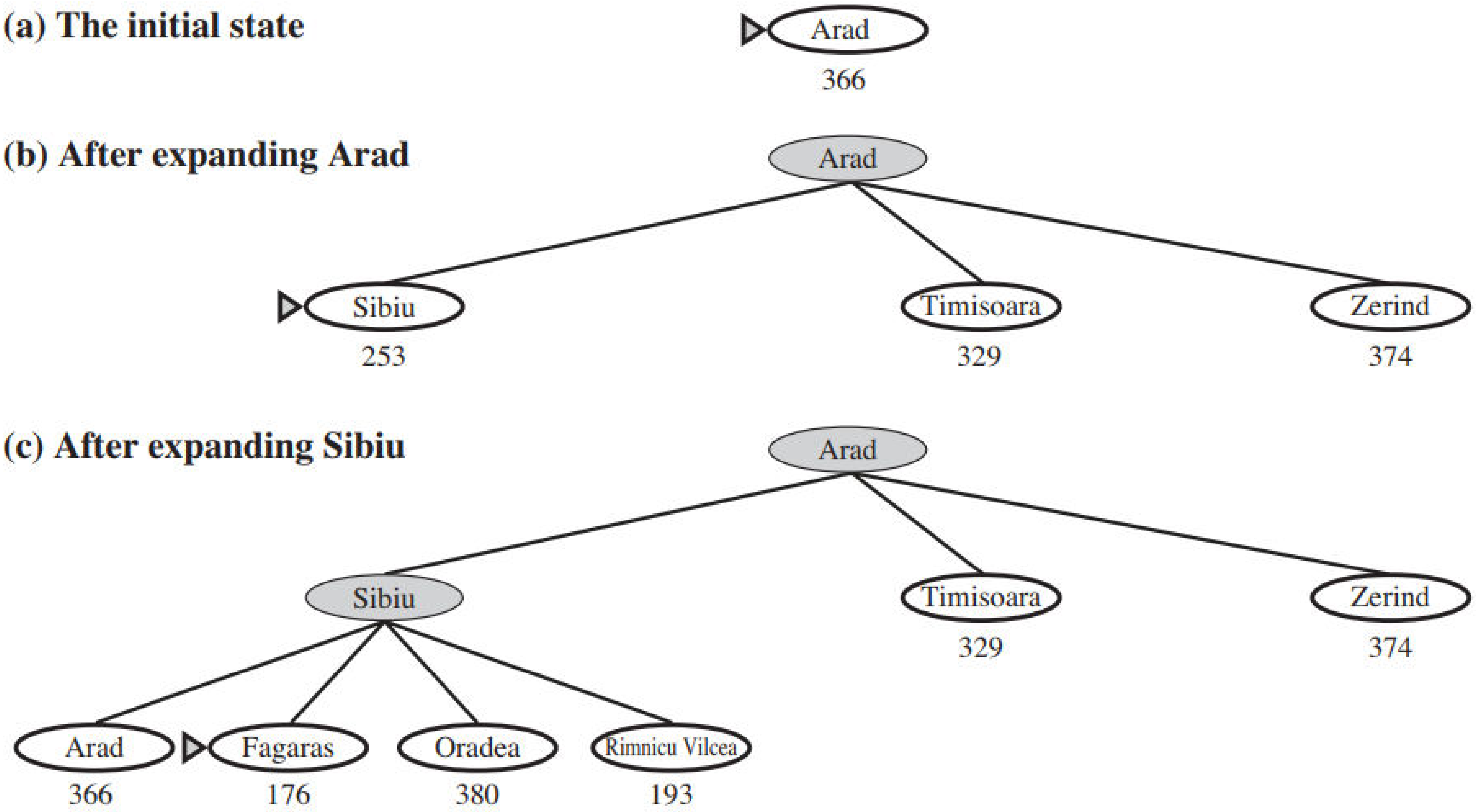
- Greedy best-first search tries to expand the node that is closest to the goal
- It evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$
  - Example: If the goal is Bucharest, $h_{SLD}(In(Arad)) = 366$, the straight line distance from Arad to the Bucharest.

# Greedy best-first search Example

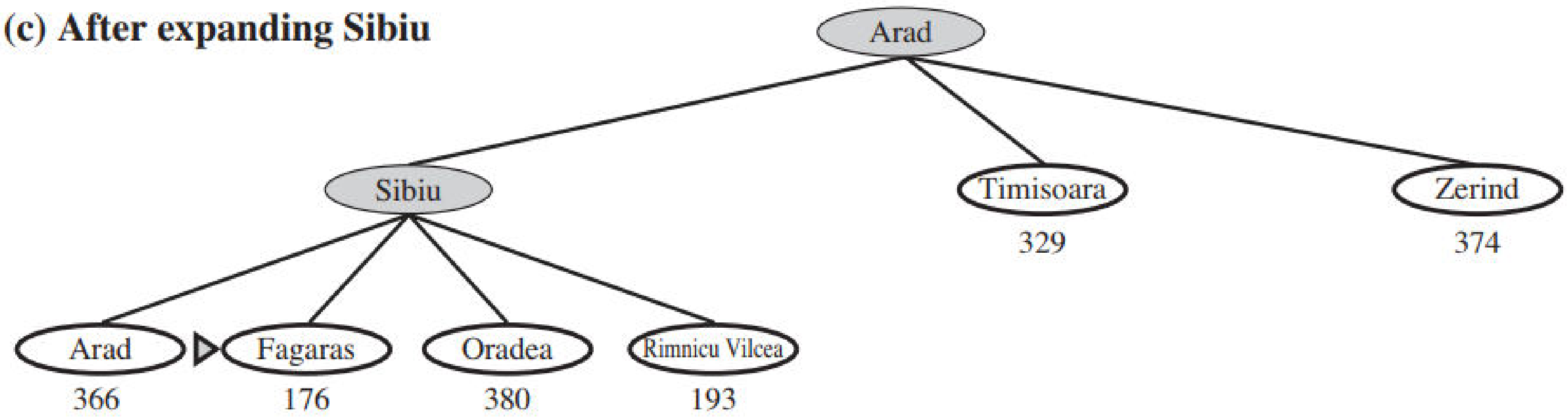


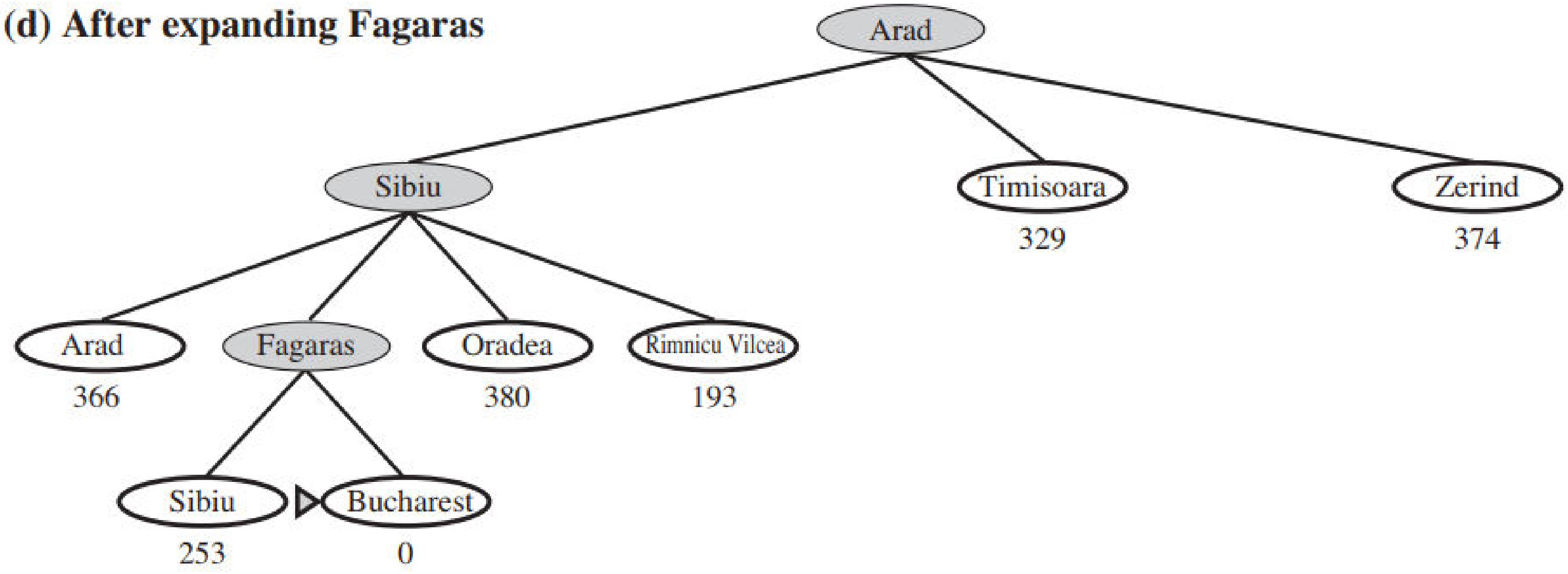

| | | | |
|---|---|---|---|
| Arad | 366 | Mehadia | 241 |
| Bucharest | 0 | Neamt | 234 |
| Craiova | 160 | Oradea | 380 |
| Drobeta | 242 | Pitesti | 100 |
| Eforie | 161 | Rimnicu Vilcea | 193 |
| Fagaras | 176 | Sibiu | 253 |
| Giurgiu | 77 | Timisoara | 329 |
| Hirsova | 151 | Urziceni | 80 |
|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

# Greedy best-first search Example



# Greedy best-first search Example



# A* Search

- The most widely known form of best-first search is called A∗ search
- the cost
- The estimated cost of the cheapest solution through *n*:

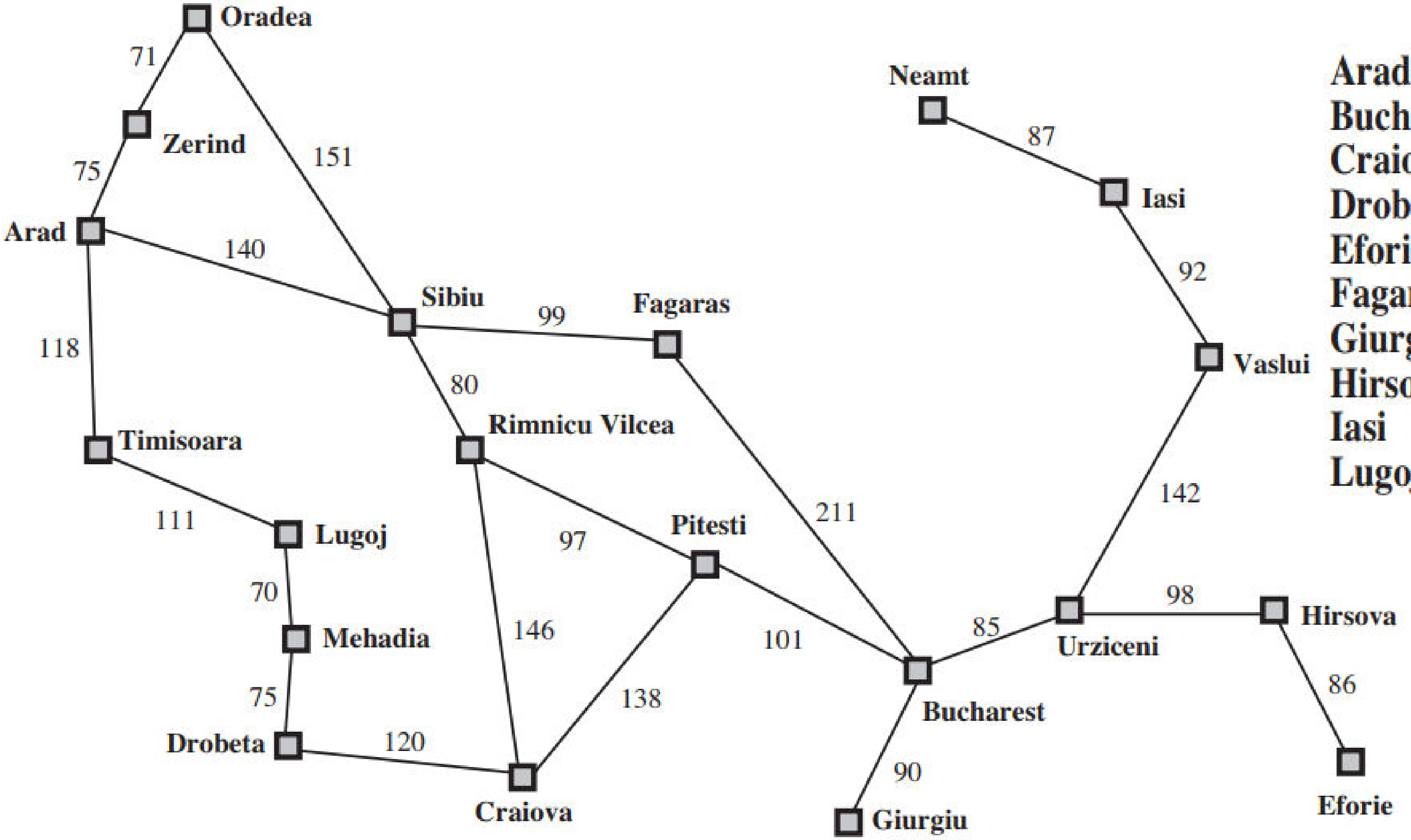
$$f(n) = g(n)+h(n)$$

  - *g(n)* gives the path cost from the *start node* to node *n*,
  - *h(n)* is the estimated cost of the cheapest path from *n* to the *goal*
- Algorithm uses two lists:
  - Frontier/open list ←the set of all leaf nodes available for expansion (can be maintained by a priority queue ordered by *f(n)*)
  - Explored/closed list ← the set of all explored nodes

# A* Search Pseudocode

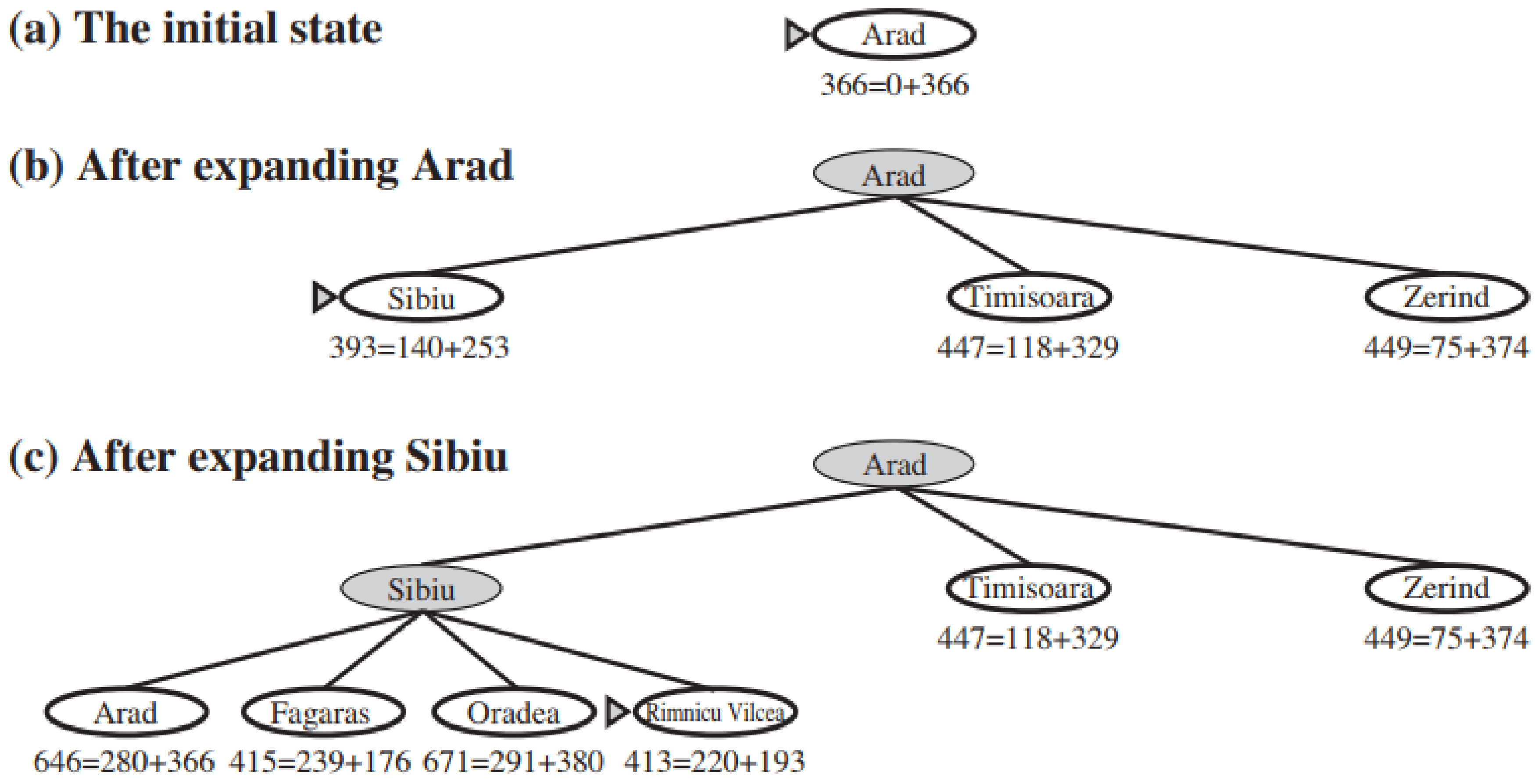
**function** A*-SEARCH(*problem*) **returns** a solution, or failure
*node* ←a node with STATE = *problem*.INITIAL-STATE, *f(node) = g(node)*
*frontier* ← a priority queue ordered by *f*-cost, with *node* as the only element initially
*explored* ← an empty set
**loop do**
    **if** EMPTY?( *frontier*) **then return** failure
    *node* ← POP( *frontier*) /* chooses the lowest *f*-cost node in frontier */
    **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
    add *node*.STATE to *explored*
    **for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
        *child* ← CHILD-NODE(*problem*, *node*, *action*)
        *f(child) = g(child) + h(child)*
        **if** *child*.STATE is not in *explored* or *frontier* **then**
            *frontier* ← INSERT(*child*, *frontier*)
        **else if** *child*.STATE is in *frontier* with higher *f(child)* **then**
            replace that frontier node with *child*

# A* Search Example

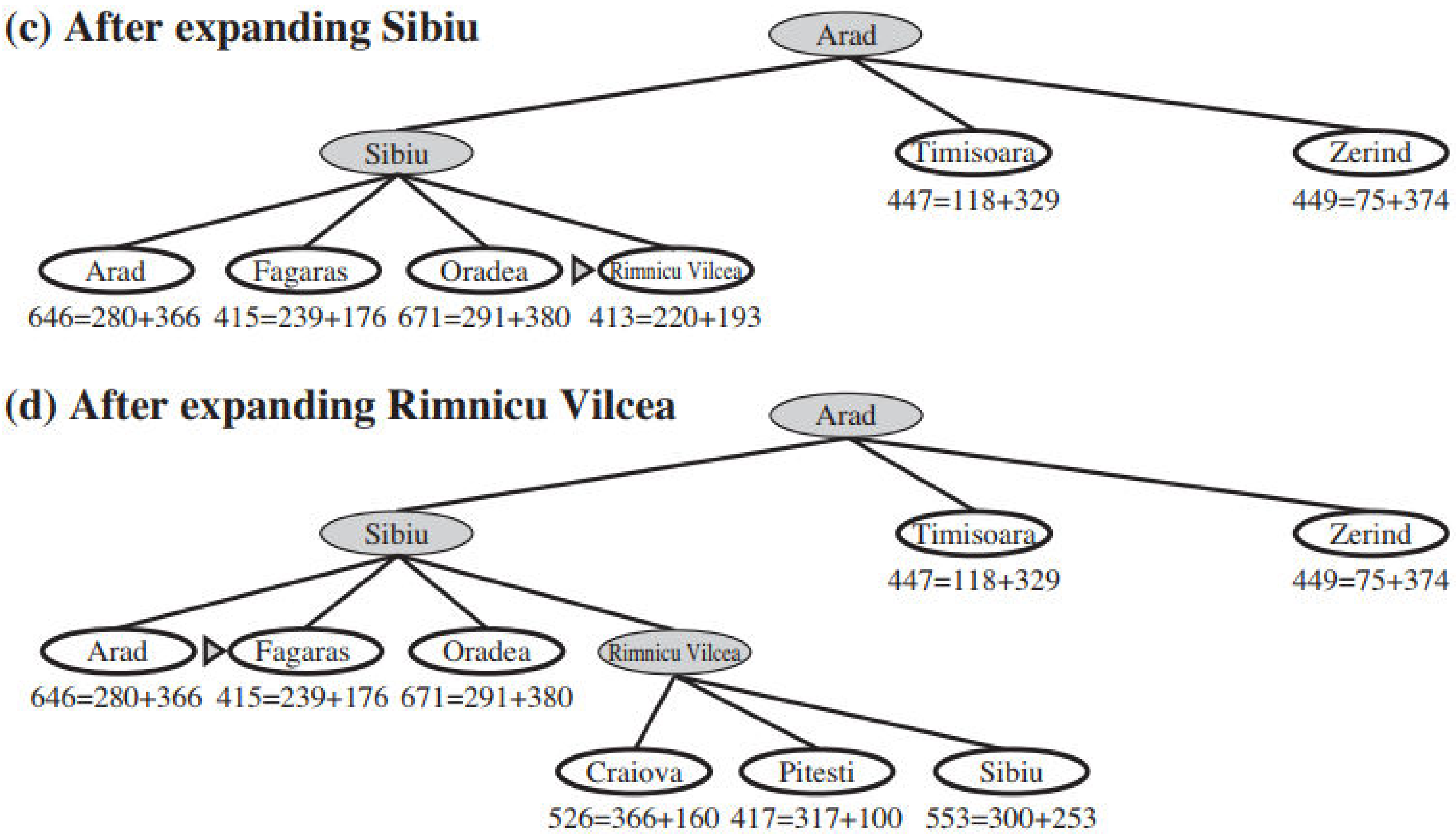



| | | | |
|---|---|---|---|
| Arad | 366 | Mehadia | 241 |
| Bucharest | 0 | Neamt | 234 |
| Craiova | 160 | Oradea | 380 |
| Drobeta | 242 | Pitesti | 100 |
| Eforie | 161 | Rimnicu Vilcea | 193 |
| Fagaras | 176 | Sibiu | 253 |
| Giurgiu | 77 | Timisoara | 329 |
| Hirsova | 151 | Urziceni | 80 |
|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

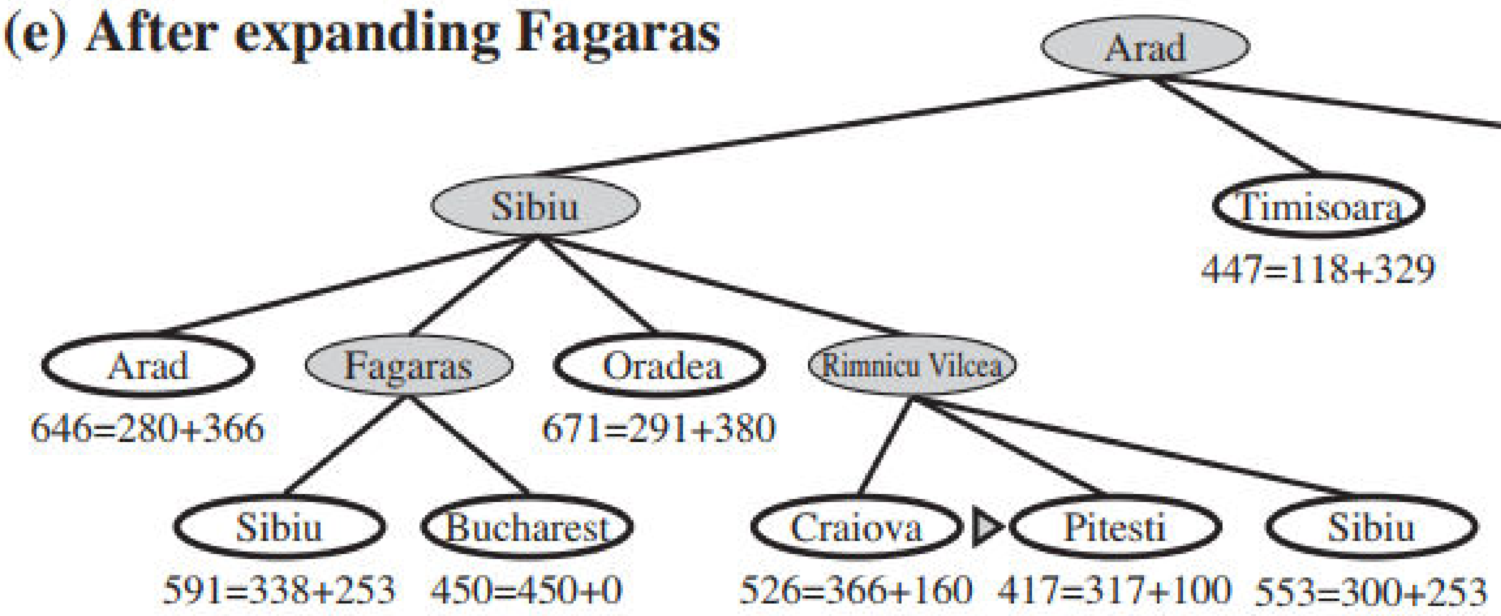
# A* Search Example



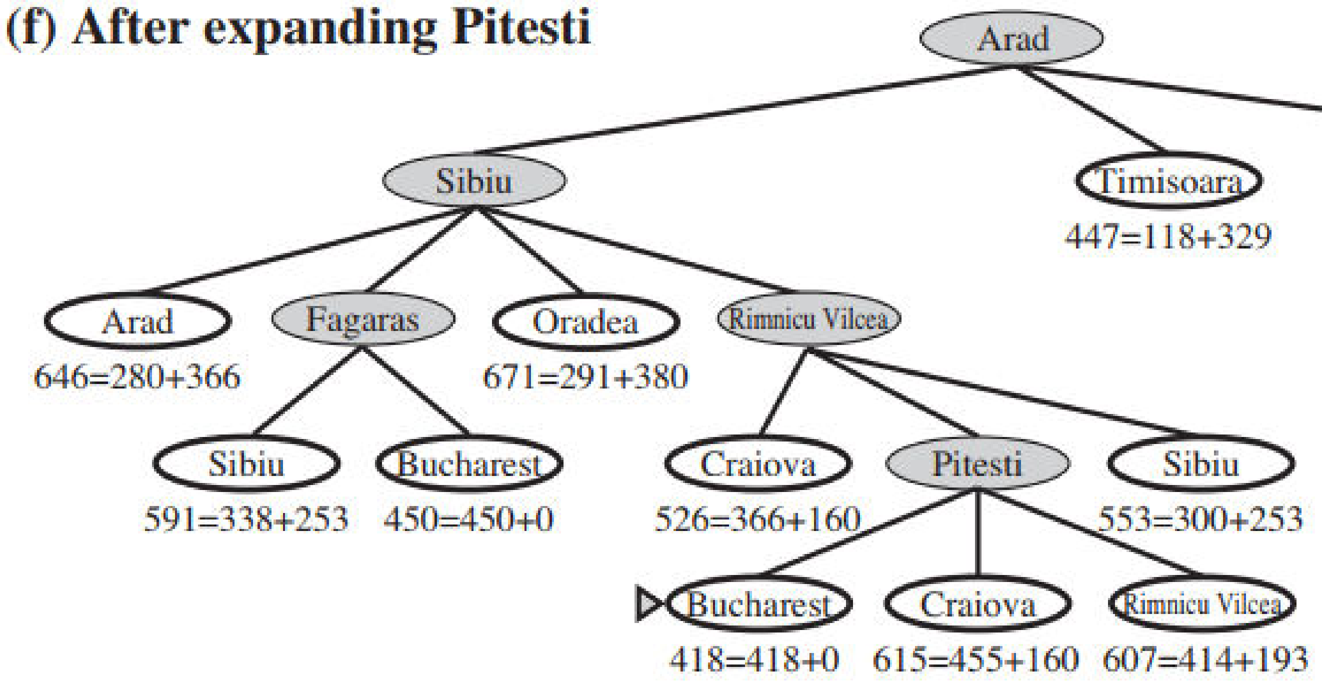
# A* Search Example



# A* Search Example



# A* Search Example



# Admissible heuristics

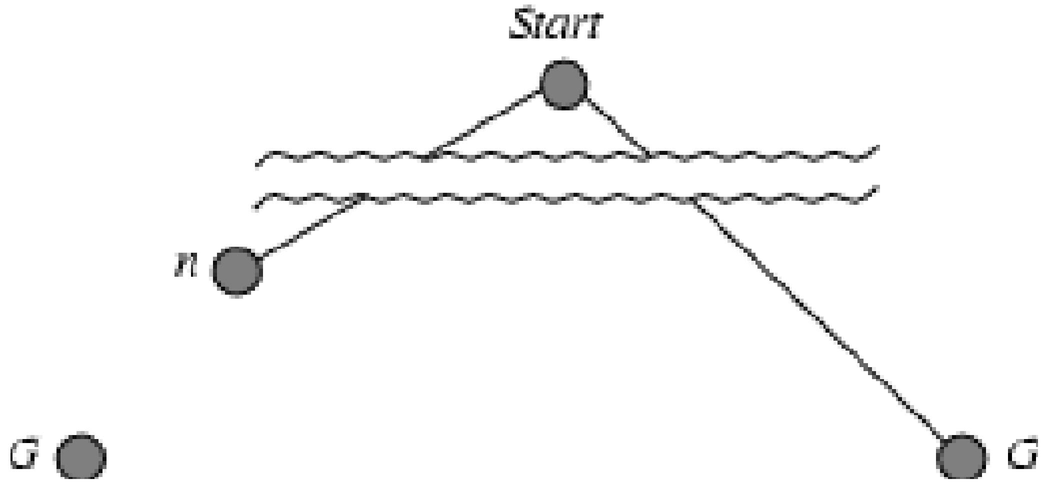
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic.

- 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 $n$.

- Example: $h_{SLD}(n)$ = *Straight Line Distance* (never overestimates the actual road distance)

# Optimality of A* Search

- If *h(n)* is admissible, A* using TREE- SEARCH is optimal.
- Proof:
- Suppose some suboptimal goal *G2* 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*.

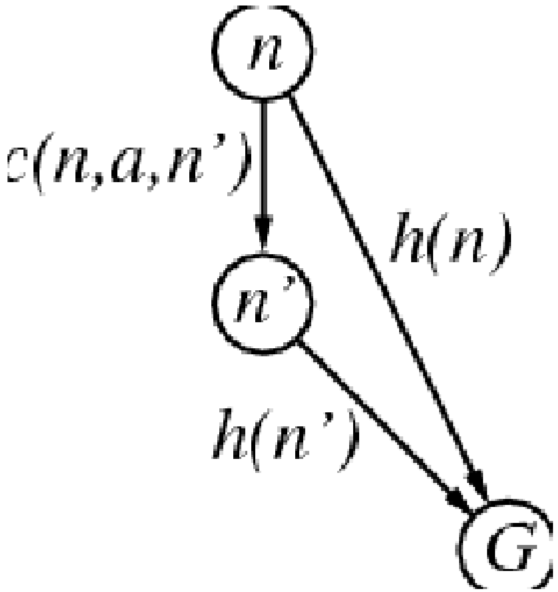


# Optimality of A* Search

- $f(G2) = g(G2)$, since $h(G2) = 0$
- $f(G) = g(G)$, since $h(G) = 0$
- $g(G2) > g(G)$, since $G2$ is suboptimal
- Therefore, $f(G2) > f(G)$
- $h(n) \leq h^*(n)$, since $h$ is admissible
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq g(n) + h^*(n) < f(G) < f(G2)$
- Hence $f(G2) > f(n)$, and A* will never select $G2$ for expansion.

# Optimality of A* Search

- If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal.
- Proof:
- A heuristic is consistent (or monotonic) if for every node $n$, every successor $n'$ of $n$ generated by any action $a$: $h(n) \leq c(n,a,n') + h(n')$



# Optimality of A* Search

- If $h$ is consistent, we have:

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

$$= g(n) + c(n,a,n') + h(n')$$

$$\geq g(n) + h(n) = f(n)$$

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

# Properties of A*

- Complete: Yes (unless there are infinitely many nodes with $f \leq f(G)$ ).
- Time: Exponential.
- Space: Keeps all nodes in memory, so also exponential.
- Optimal: Yes (provided h admissible or consistent).
- Optimally Efficient: Yes (no algorithm with the same heuristic is guaranteed to expand fewer nodes).