# Structure of Agents

- Kinds of agents
  - Simple reflex agents
  - Model based reflex agents
  - Goal-based agents
  - Utility-based agents
  - Learning agents

# Simple Reflex Agent

- Use simple “if then” rules also called **condition-action rule**
- Can be short sighted
- These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

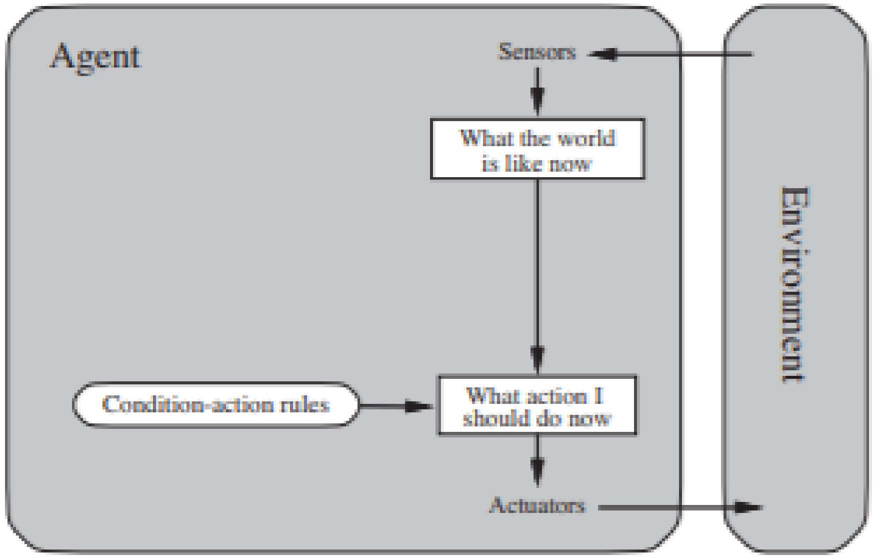



**Figure 2.9** Schematic diagram of a simple reflex agent.

# Model based reflex Agent

- Store previously-observed information
- Can reason about unobserved aspects of current state

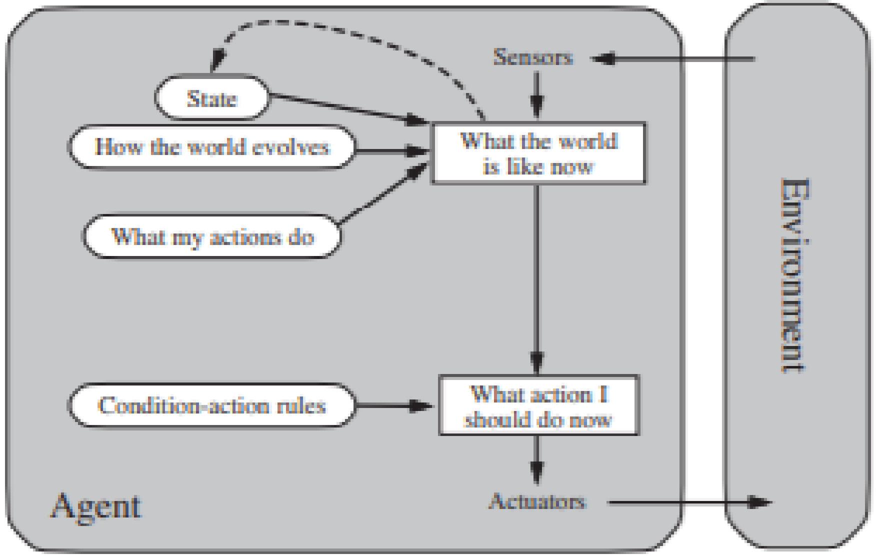



**Figure 2.11** A model-based reflex agent.

# Goal-based Agent

- Goal reflects desires of agents
- May project actions to see if consistent with goals
- Takes time, world may change during reasoning
- They have specific goals or objectives that they try to achieve, and they take actions based on the current percepts and their internal state to reach those goals

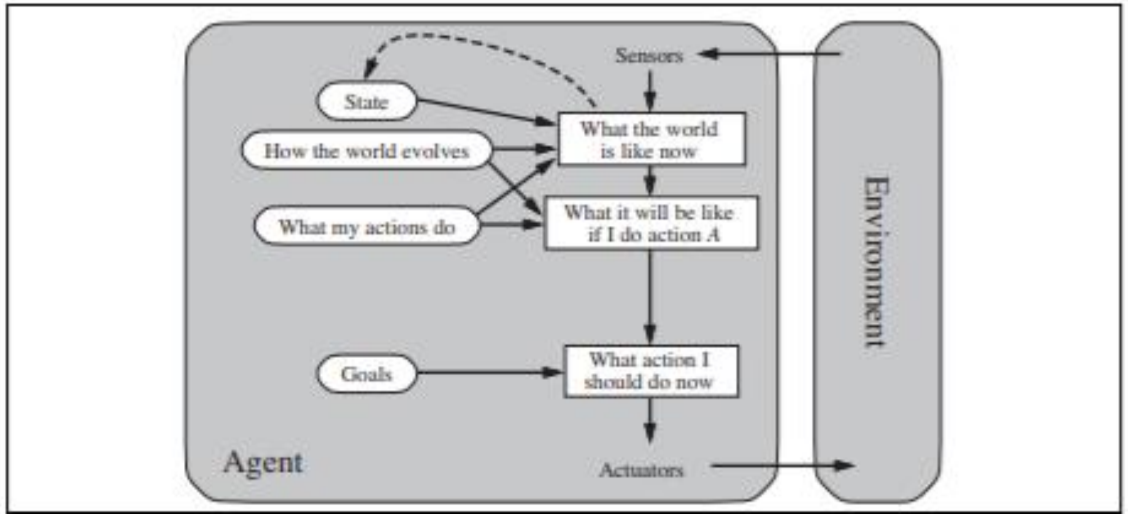



**Figure 2.13** A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

# Utility-based Agent

- Goals alone are not enough to generate high-quality behavior in most environments
- Utility is a measure of the value or desirability of a particular state or outcome. The agent uses utility to determine the actions it should take in order to achieve its goals.
- They take into account the long-term consequences of their actions to maximize a specific utility function.

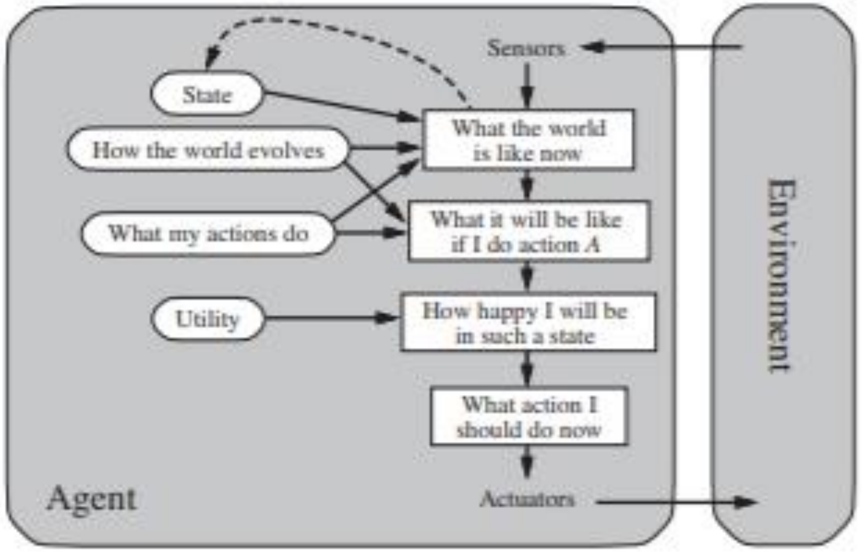



**Figure 2.14** A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

# Learning Agent

- They are able to improve their performance over time by learning from their experiences and adjusting their behavior accordingly

- The agent uses past experiences and feedback to continuously improve its decision-making and problem-solving abilities.

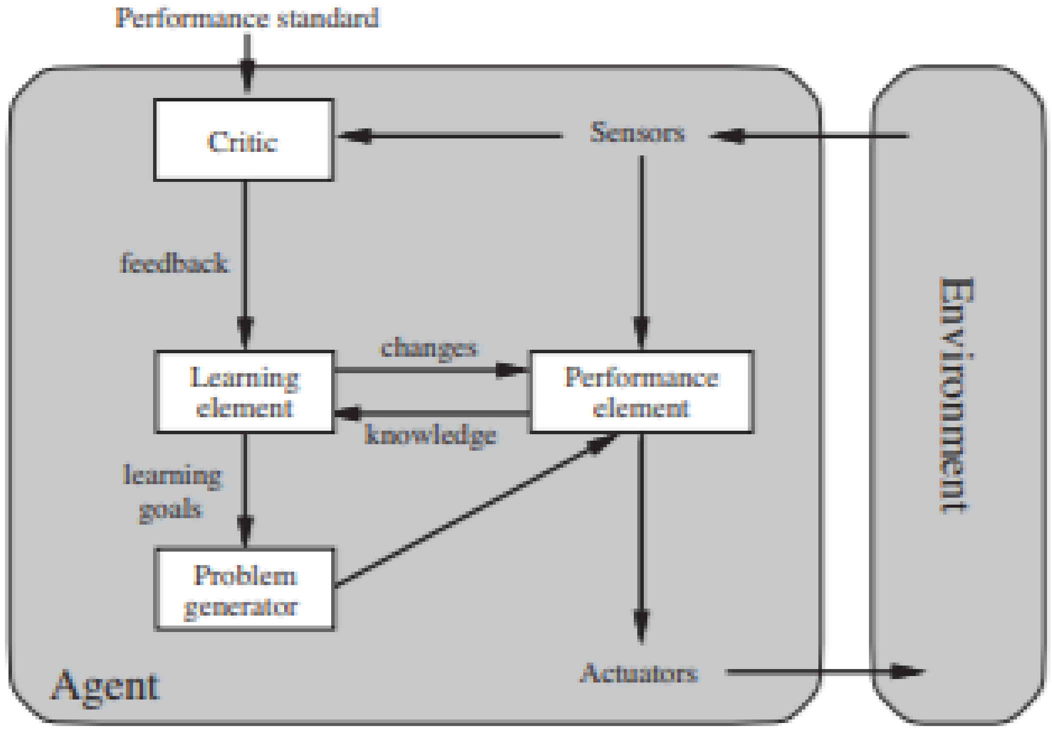



**Figure 2.15** A general learning agent.

# Homework

- Readings
  - CH 2- Intelligent Agent (Section 2.1 - 2.4 )

# Artificial Intelligence

CH-3: Solving Problem by Searching

# Today’s Topic

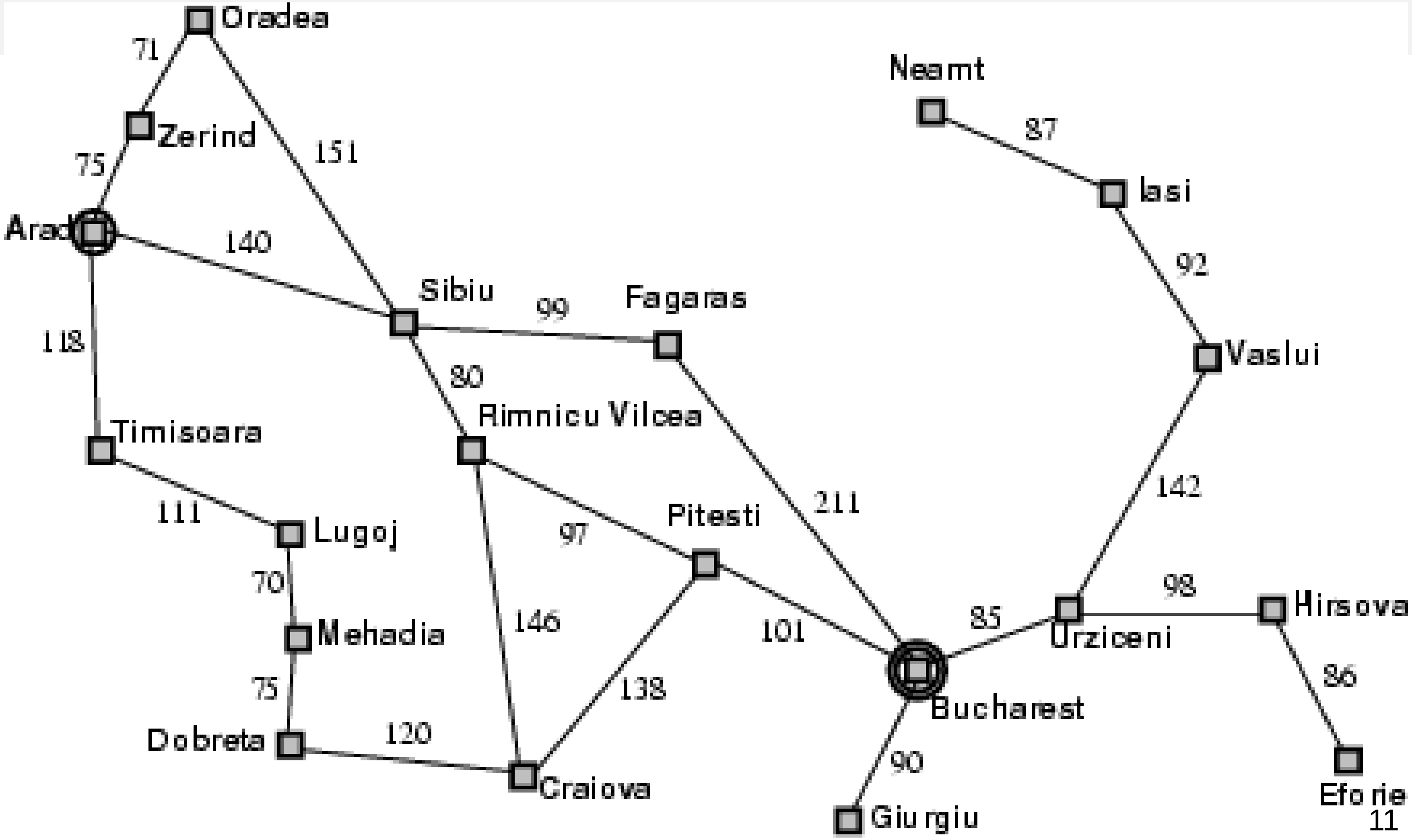
- Problem Solving Agent
- Problem formulation
  - – What to DO to get a GOAL {What agent type is it?}
- Example problems
- Basic search algorithms

# Problem Solving Agent

Steps:

- Goal formulation
- Problem formulation
- Search
- Execute

# Example: Romania




11

# Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

# *Problem*

**A problem is defined by four items:**

1. initial state e.g., "at Arad"
2. Action, operator or successor function *S(x)* = set of action–state pairs
   – e.g., *S(Arad)* = {*<Arad →Zerind, Zerind>, … *}

Initial state + successor function – state space
It defines the **possible actions and resulting states** that can be reached from the state "Arad".

The notation <Arad →Zerind, Zerind> represents an action-state pair, where "Arad" is the current state, "Zerind" is the next state, and " →Zerind" represents the action that leads from "Arad" to "Zerind".

# ***Problem***

**A problem is defined by four items:**

3. goal test: defined in problem e.g. Bucharest

4. path cost (additive) : assigns cost to a path
   - e.g., sum of distances, number of actions executed, etc.
   - $c(x,a,y)$ is the step cost, assumed to be ≥ 0
   - More than one solutions… select a preferable solution

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

# Formulating Problem

- Formulation of Problem includes: initial state, actions, transition model, goal test and path cost---Model
- Real-World factors:
  - the traveling companions
  - the current radio program
  - the scenery out of the window,
  - the proximity of law enforcement officers,
  - the distance to the next rest stop, the condition of the road, the weather, and so on

The process of removing detail from a representation is called **abstraction**

# Problem-solving agents

```
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
```

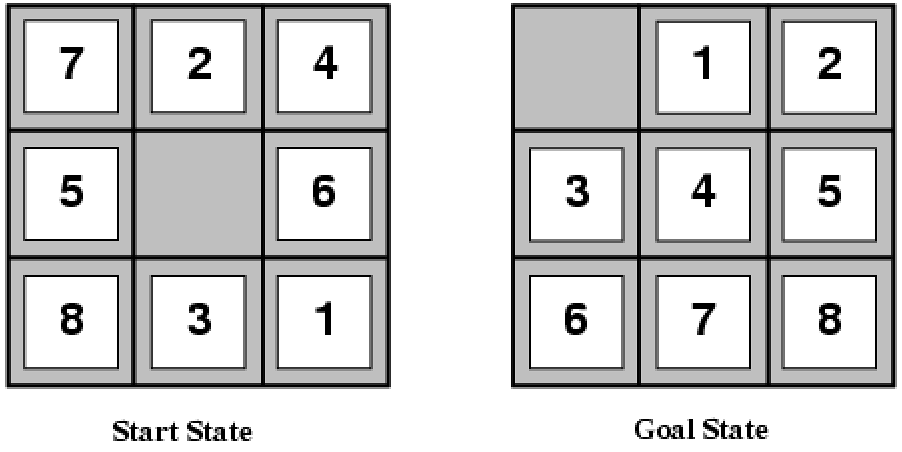
**Figure 3.1** A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

# Example Problems

- Toy Problems:
  - It is intended to illustrate various problem-solving methods
  - Use to compare performance of algorithms
- Real-world Problems:
  - Real-world problems are complex, real-life challenges that require a solution
  - It is the one whose solutions people actually care about

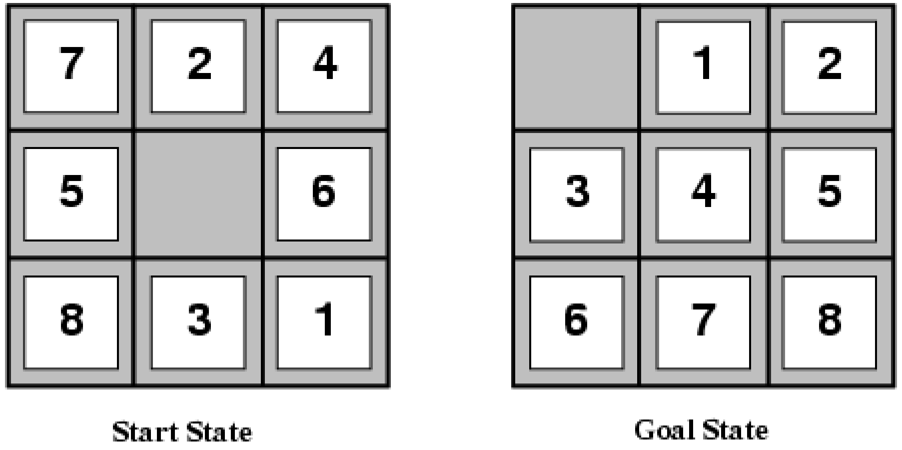
# Example: The 8-puzzle

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



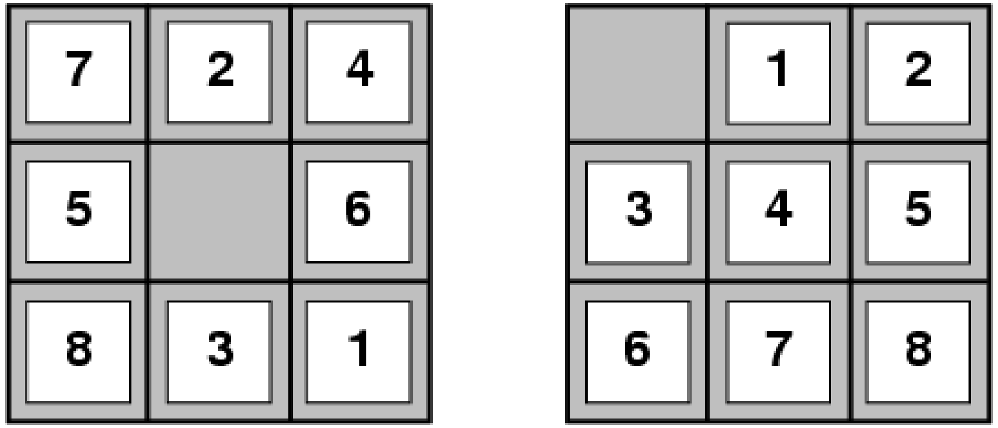
# Example: The 8-puzzle

- 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-family Puzzle is NP-hard]

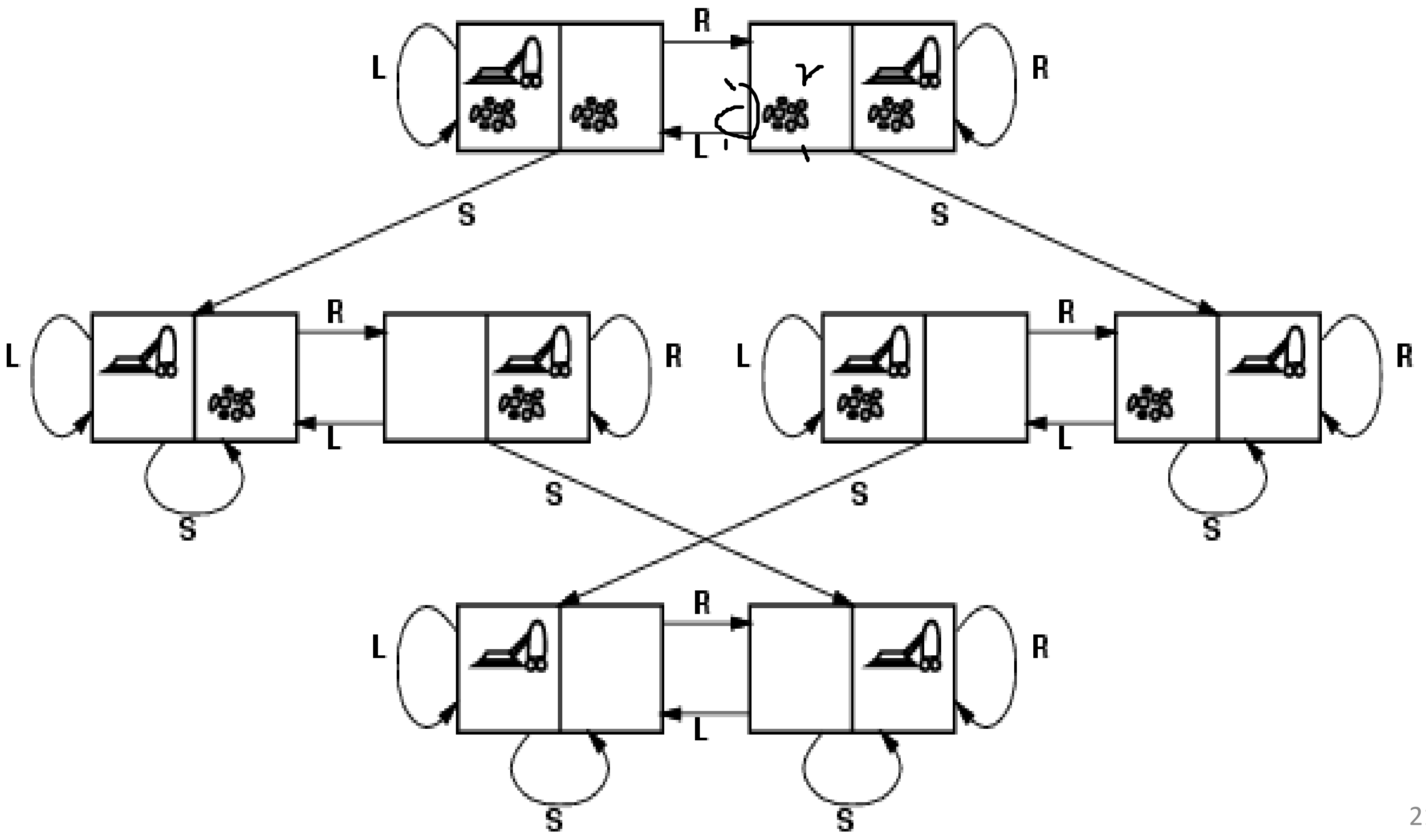
- How many moves will be required to reach the goal state?




Start State Goal State

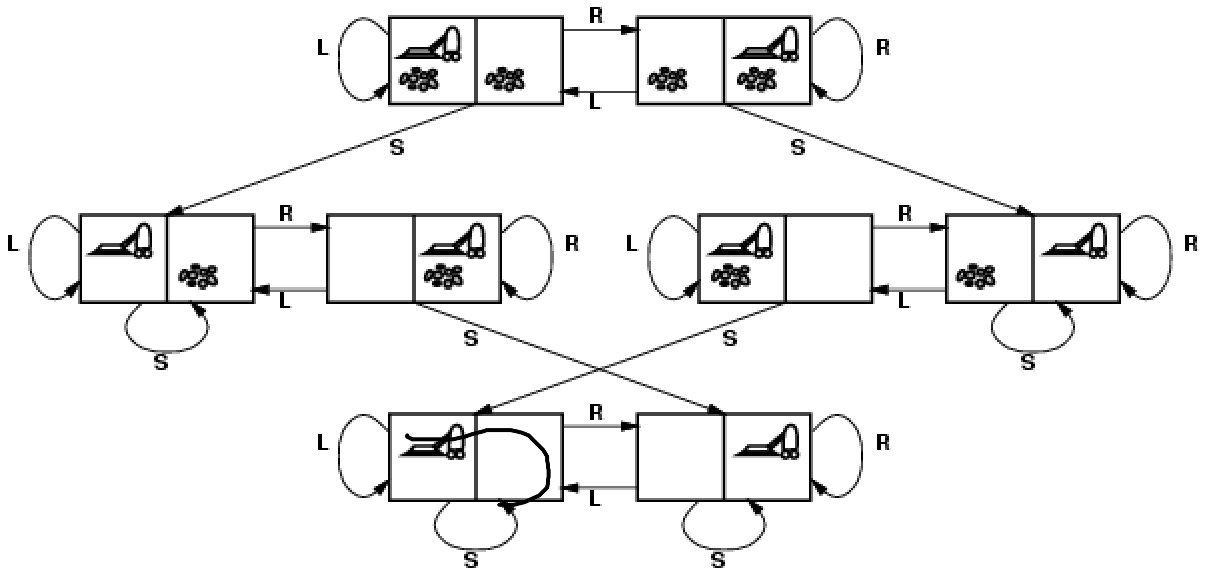
# Vacuum world state space graph

- 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



# Problem Formulation Examples

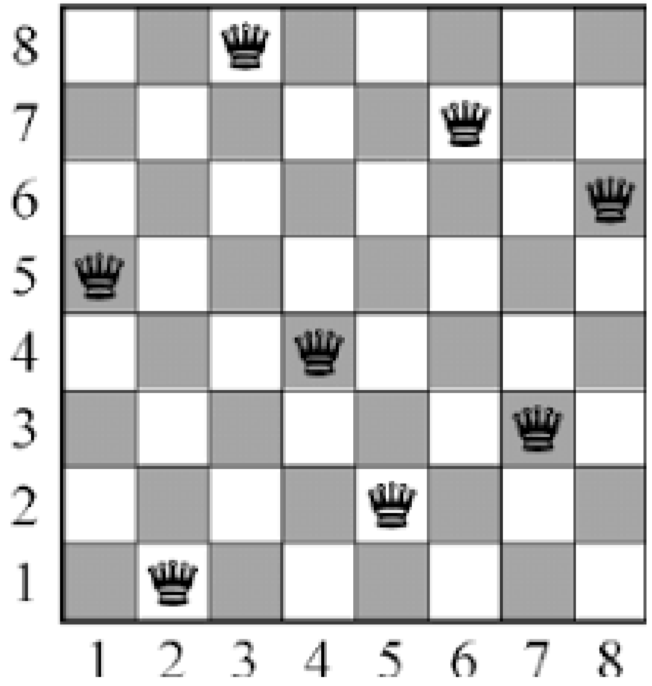
- Toy Problem (Vacuum Cleaner)=done
- 8-Puzzle=done
- 8-Queen Problem
- Route-finding Problem
    - Routing in networks; Operations research (military, business etc); air-line travel planning systems
- Touring Problem (TSP=traveling salesperson Problem)
    - State space include current city + set of cities already visited
    - Application to stocking machines on shop floors; automatic circuit drills

# Real-life Examples

- Robot navigation (Ch.25),
- VLSI Layout
- Automatic assembly sequencing (Michie, 1972),
- Protein sequencing(Ch.10)
- Scheduling problems

# 8-Queen Problem

- Place 8 Queens on chessboard such that no queen attack any other
  - no two queens can be placed on the same row, column, or diagonal.
- Problem Formulation
  - **Complete-state**: all queens placed on the board
  - **Incremental**: maintaining a partially filled board and adding one queen at a time, so it does not threaten any of the previously placed queens



# Complete State formulation

- Considers the entire state of the board as a single entity
- Starts with all 8 queens on the board and moves them around until a solution is found.
- **goal test:** 8 queens on board, none attacked
- **path cost:** irrelevant
- **states:** any arrangement of 0-8 queens on the board
- **operators:** add or remove a queen to/from any square
- 64^8 possible combinations to investigate??

# Complete State formulation

A more sensible choice would use the fact that placing a queen where it is already under attack cannot work because subsequent placings will not undo the attack. So, try the following instead:

- **States**: Arrangement of n (0 to 8) queens on board, one per column in the leftmost n columns, with no queen attacking any other
- **Action**: Add queen in leftmost empty column such that queen is not attacking any other queen

# Incremental formulation

- States: Any arrangement of 0 to 8 queens on the board
- Initial state: No queens on board
- Action: Add queen to any empty cell(square)
- Goal Test: 8 queens on board and no attack
- Path cost: Not Interested
- multiplying the number of possibilities for each queen

# Searching for Solutions

- Search the state space:
  - State space can be represented by a **search tree**
  - Root of the tree is the initial state also called **search node**
  - Children generated through successor function
  - The choice of which state to expand is determined by **search strategy**

# Tree Search Example

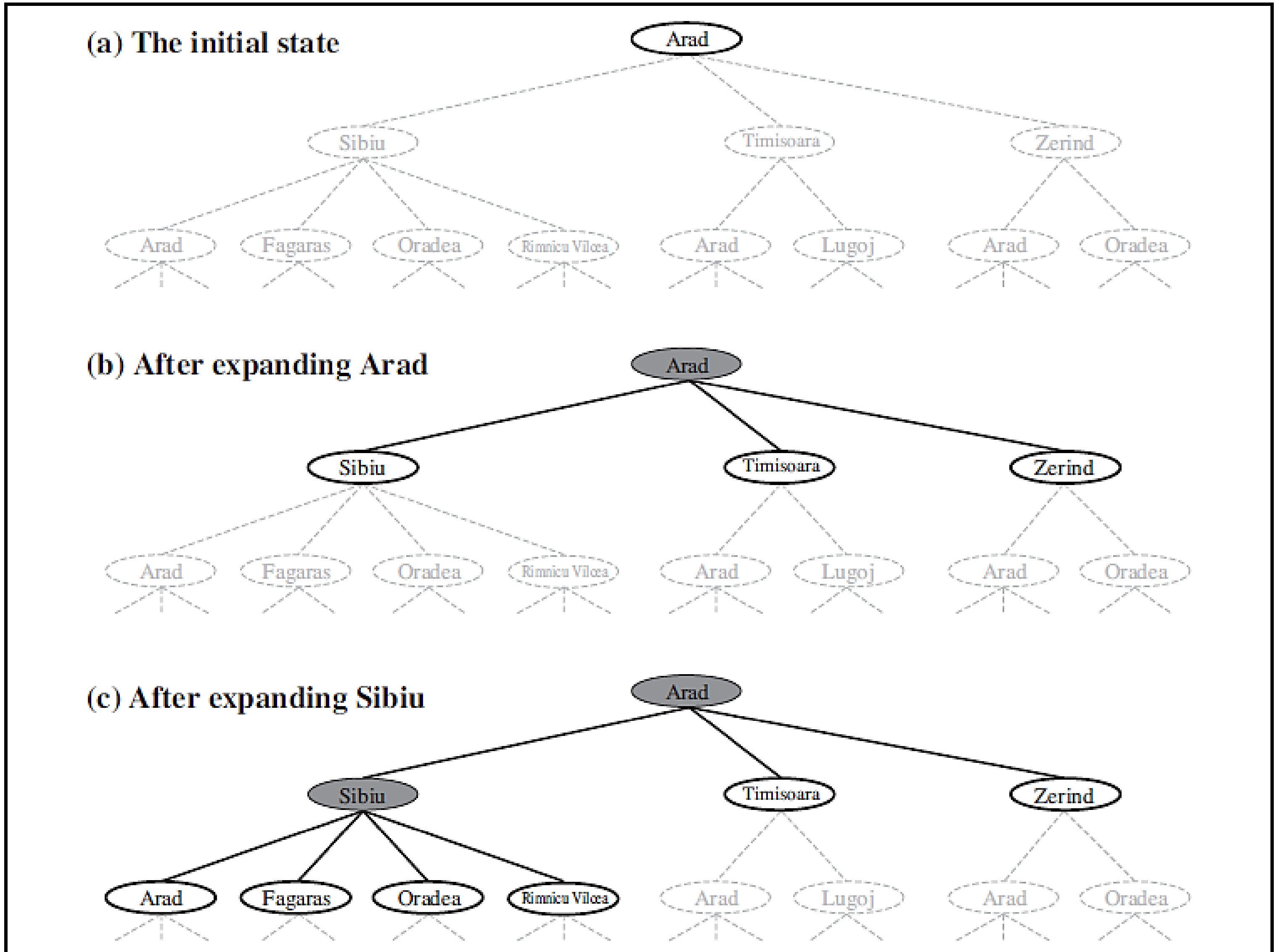
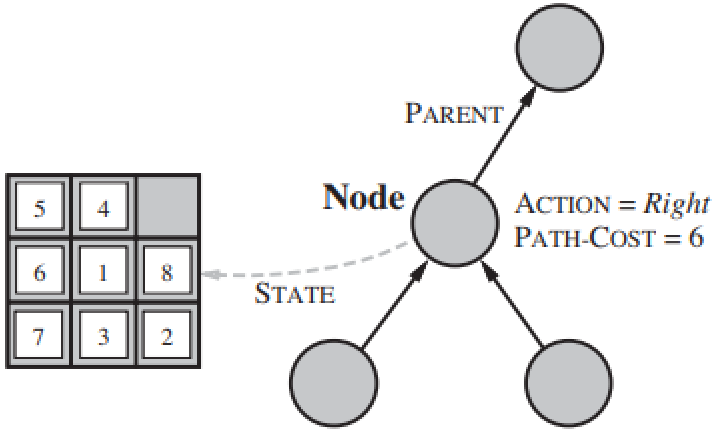



**Figure 3.6** Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

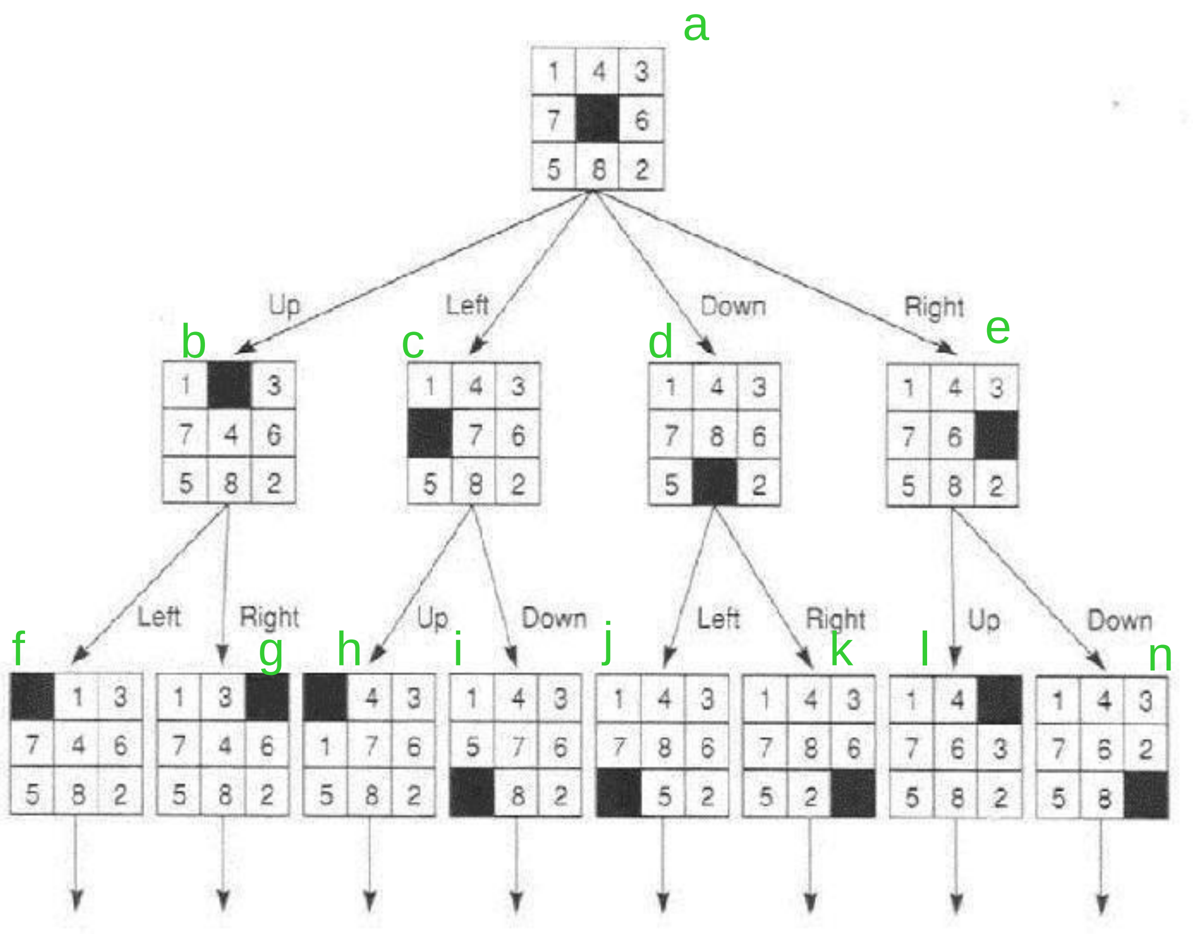
# Implementation: states vs. nodes

- 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




- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

# State-space to graph formulation & Searching



# Search Graph

A ***graph*** consists of a set $N$ of ***nodes*** and a set $A$ of ordered pairs of nodes, called ***arcs or edges***.

Node $n_2$ is a ***neighbor*** of $n_1$ if there is an arc from $n_1$ to $n_2$. That is, if $\langle n_1, n_2 \rangle \in A$.

A ***path*** is a sequence of nodes $n_0, n_1, n_2, .., n_k$ such that $\langle n_{i-1}, n_i \rangle \in A$.

A ***cycle*** is a non-empty path such that the start node is the same as the end node

A ***directed acyclic graph*** (DAG) is a graph with no cycles

Given a start node and goal nodes, a ***solution*** is a path from a start node to a goal node.

# Search strategies & Measuring Problem-solving Algorithm's Performance

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - Completeness: Does it always find a solution if one exists?
  - Time complexity: How long does it take to solve the problem (number of nodes generated)
  - Space complexity: How much memory is needed to perform the search (maximum number of nodes in memory)
  - Optimality: does it always find a least-cost solution?

# Measuring Problem-solving Algorithm's Performance

- In AI, graph is implicitly represented by initial state and successor function and complexities is expressed in terms of three quantities:
  - Branching factor (b) – maximum number of successors of any node
  - Depth (d) – depth of shallowest goal node
  - m – maximum length of any path in the state space
- Time is often measured in terms of no of nodes generated during the search and space in terms of maximum number of nodes stored in memory
- To assess the effectiveness of a search algorithm, we also consider search cost— which typically depends on the time complexity but can also include a term for memory usage—or we can use the total cost, which combines the search cost and **the path cost** of the solution found.

# Branching Factor

The ***forward branching factor*** of a node is the number of  arcs going out of the node

The ***backward branching factor*** of a node is the number  of arcs going into the node