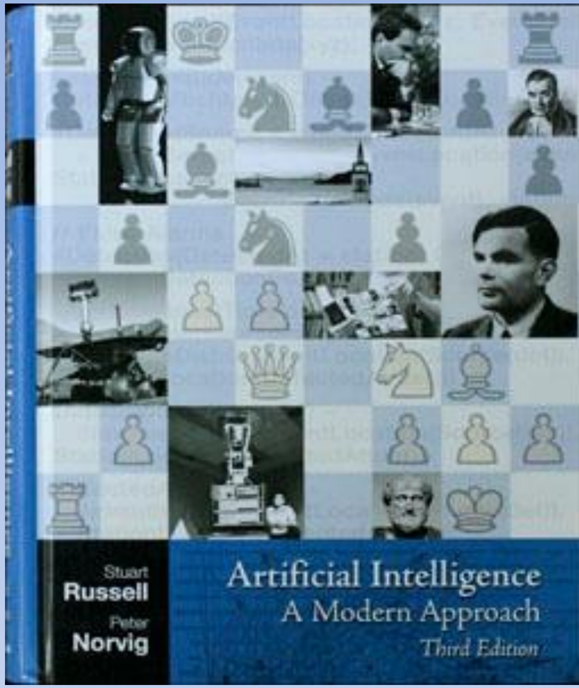



# Chapter 3: Solving Problems by Searching

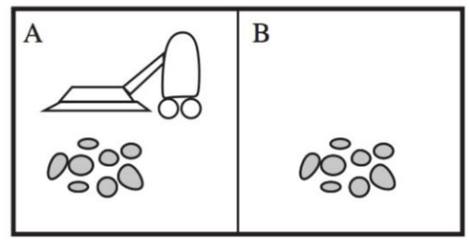



**Figure 2.2** A vacuum-cleaner world with just two locations.

| Percept sequence | Action |
|---|---|
| $[A, Clean]$ | *Right* |
| $[A, Dirty]$ | *Suck* |
| $[B, Clean]$ | *Left* |
| $[B, Dirty]$ | *Suck* |
| $[A, Clean], [A, Clean]$ | *Right* |
| $[A, Clean], [A, Dirty]$ | *Suck* |
| ⋮ | ⋮ |
| $[A, Clean], [A, Clean], [A, Clean]$ | *Right* |
| $[A, Clean], [A, Clean], [A, Dirty]$ | *Suck* |
| ⋮ | ⋮ |

**Figure 2.3** Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

# Chapter 3: Solving Problems by Searching

- Introducing **Problem-Solving Agent**
  - A Goal-Based Agent
- Use ATOMIC Representations
  - States of the world have no INTERNAL STRUCTURE
  - Later Planning Agents (Ch#7-10) have factored or structured representations.
- GOAL used to help simplify maximizing performance measure.

# Romania Example

- I’m in Arad, Romania!
  - Sight Seeing
  - Photos

WAIT:
I have a non-refundable ticket leaving out of Bucharest tomorrow!!!

# Romania Example

- I’m in Arad, Romania!
- I have a non-refundable ticket leaving out of Bucharest tomorrow!!!
- Better Adopt Goal: IN BUCHAREST!

# Romania Example

- Goal Formulation:
  – Given: Current Situation, Performance Measure
  – Generate: Goal
  – First Step in Problem Solving
- Problem Formulation:
  – Given: Goal
  – Decide: Actions and States to consider.

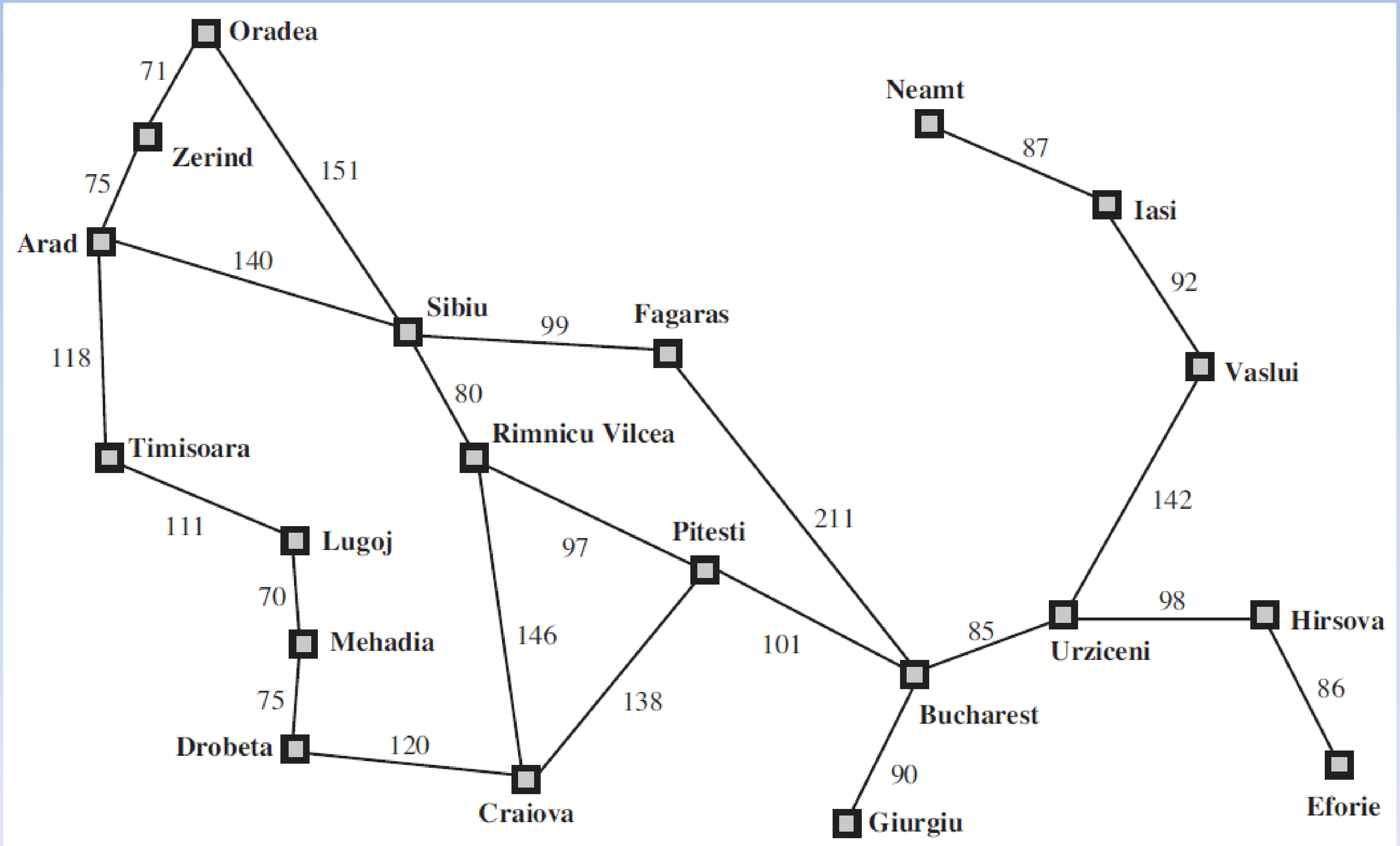
# Simple Problem-Solving Agent

- Agent's Dilemma:
  - Given several immediate options (High Road, Low Road, …)
  - Which options leads to my goal
- Agent examines the results of available actions to find a sequence leading to goal.

# Simple Problem-Solving Agent Assumptions

- Environment is OBSERVABLE
  - Agent always know where it is.
  - I’m in Fresno, Agent in Arad.
- Environment is DISCRETE
  - Finite number of actions from any state.
- Environment is KNOWN
  - We have a map.
- Environment is DETERMINISTIC
  - Roads don’t magically deliver us to different destinations.
- SEARCH:
  - Process of looking for the sequence of actions that lead to goal.

# Map



Oradea
71
Neamt
Zerind
87
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
85
Hirsova
Mehadia
146
101
Urziceni
86
75
138
Bucharest
Drobeta
120
90
Craiova
Giurgiu
Eforie

# Simple Problem-Solving Agent Search

- SEARCH:
  - Process of looking for the sequence of actions that lead to goal.
- Thinking (Searching) Agent closes eyes to the world.
- Executes solution (sequence of actions) found one step at a time.

# Simple 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
```

**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.

# Simple 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*

Contains Action Sequence

Return First 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.

# Simple Problem-Solving Agent: Knows Action Sequence



```
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, percep
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null actio
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Contains Action Sequence

Return First 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.

# Simple Problem-Solving Agent: Doesn't Know Action Sequence



**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*

SEARCH

**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.

# Search Problem

- Initial State
- Possible Actions:
  - Actions(s) : Available actions in state ‘s’
- Action Behaviors (Transition Model)
  - Result(s, a): State that results from doing action ‘a’ in state ‘s’
- Goal Test
- Path Cost

# Successor Function

- SuccessorFN: Used by many treatments of problem solving.
- SuccessorFN: Defined with Actions(state), Result(state, action)

```
Def Successors(state):

    return [(action, Result(state, action)) for action in Actions(state)]
```

# Modeling / Abstraction

- State Space Defined by:
  - Initial State, Possible Actions, and Action Behaviors
- Abstraction required to create representation
  - Detail removed from state descriptions
  - Detail removed from action behaviors

# Modeling / Abstraction Navigation Example

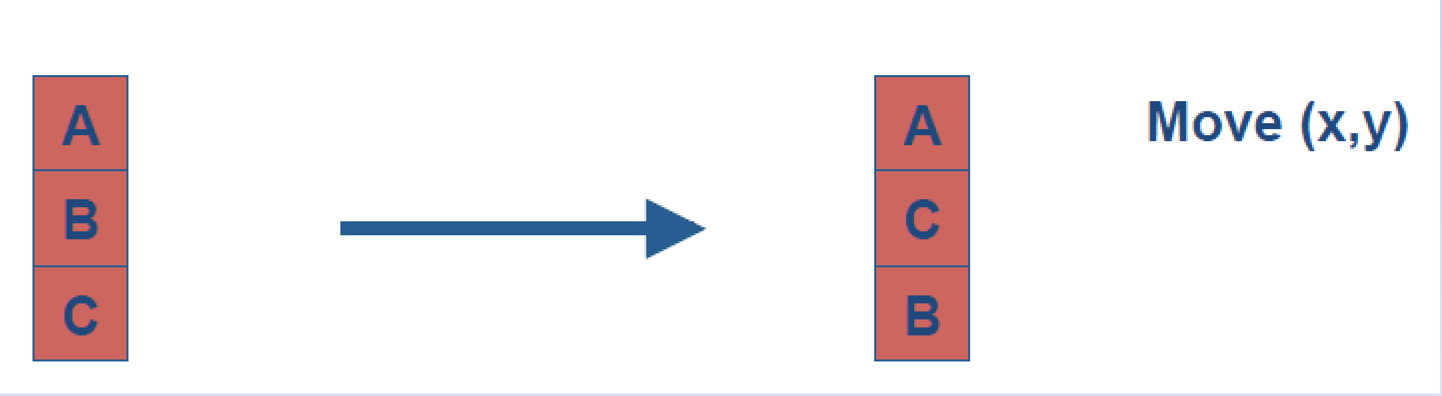
- How do we define States & Operators ?
  - First step is to abstract “the big picture”
    - i.e., solve a map problem
      - Nodes=cities,  Links=freeways/roads (high-level description)
    - This description is an abstraction of the real problem
  - Details later, like freeway onramps, refueling, etc.
- Abstraction is critical for automated problem solving
  - Approximate/simplified model of the world:
  - Good abstractions retain all important details.

# Modeling / Abstraction

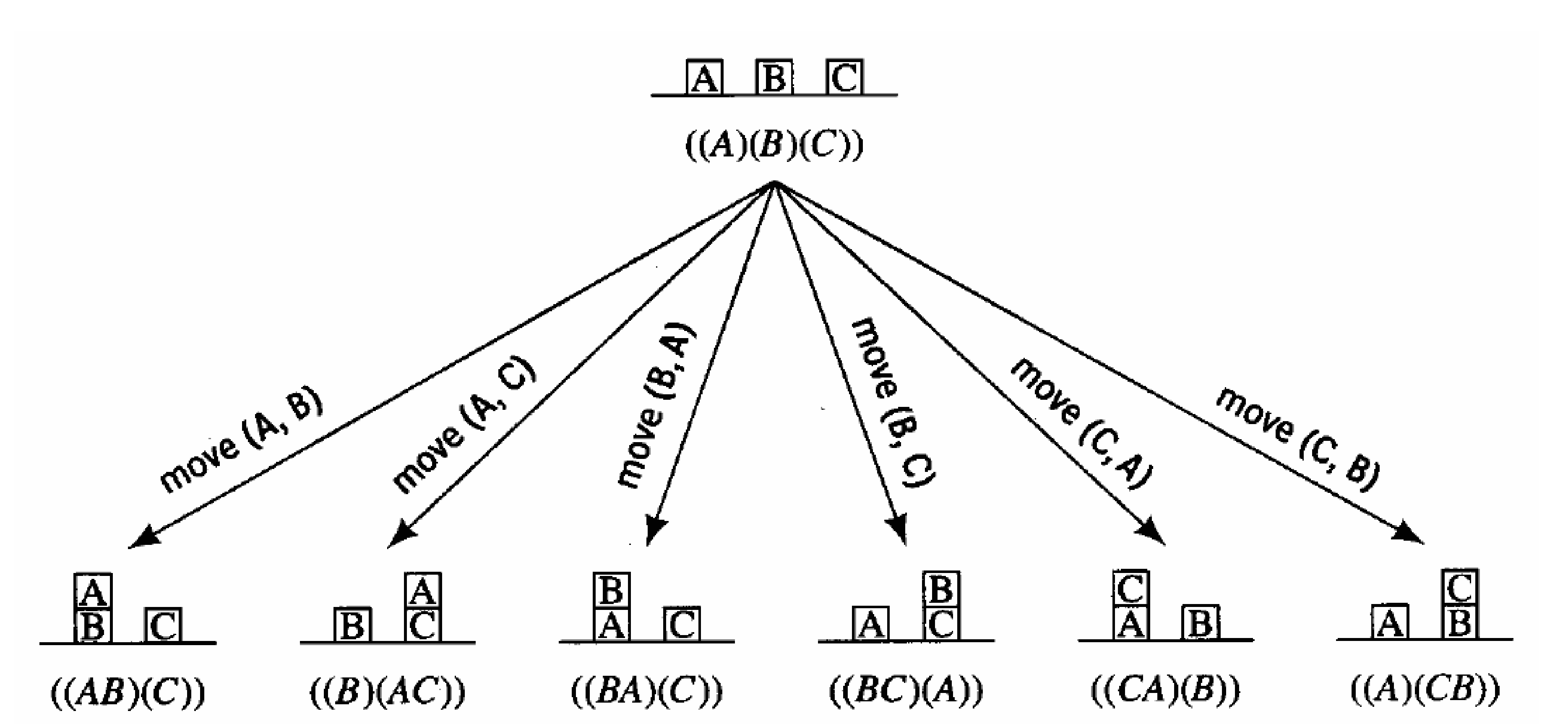
- State Space Defined by:
  - Initial State, Possible Actions, and Action Behaviors
- Abstraction required to create representation
  - Detail removed from state descriptions
  - Detail removed from action behaviors
- **Valid Abstraction:**
  - **Any abstract solution can be expanded into actual solution.**
- **Useful Abstraction:**
  - **Executing abstract solution is easier than original problem.**

# State Space: Robot Block World

- Given a set of blocks in a certain configuration,
- Move the blocks into a goal configuration
- Example:
  - (CBA) -> (BCA)

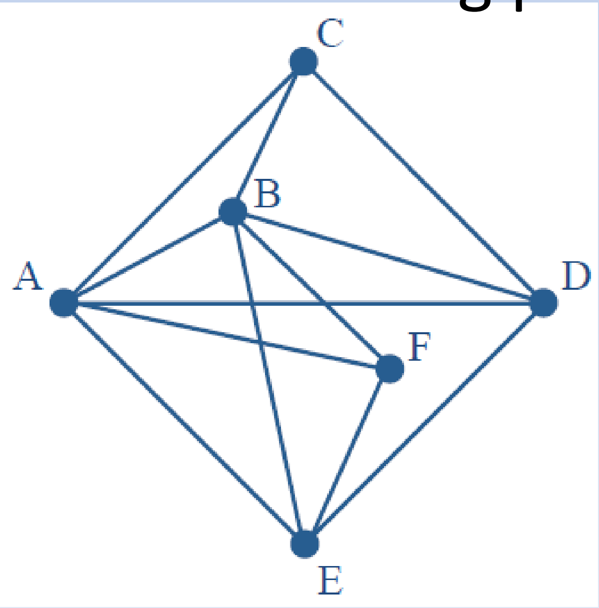


# Operator Description



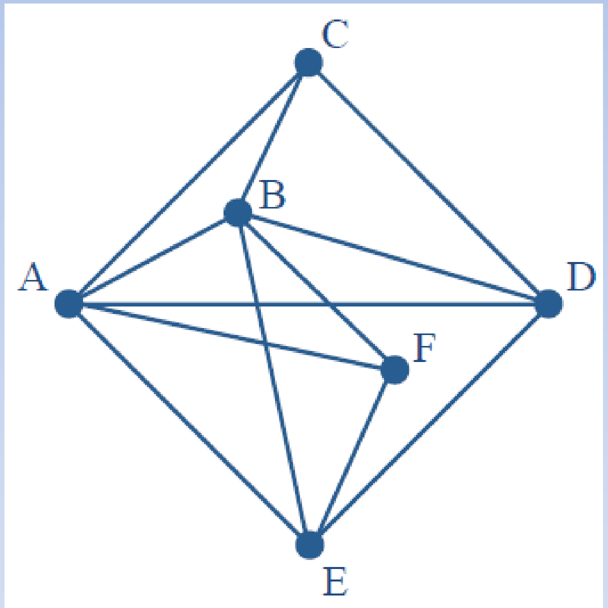
# Traveling Salesman Problem

- Our salesman needs to visit a bunch of cities.
  - He needs to visit each city only once.
  - He doesn't want to waste extra travel time.
- GOAL:
  - Find the shortest tour that visits all cities without visiting any city twice and return to starting point.
- STATE:
  - Sequence of Cities Visited.
- Start State:
  - First City (A)

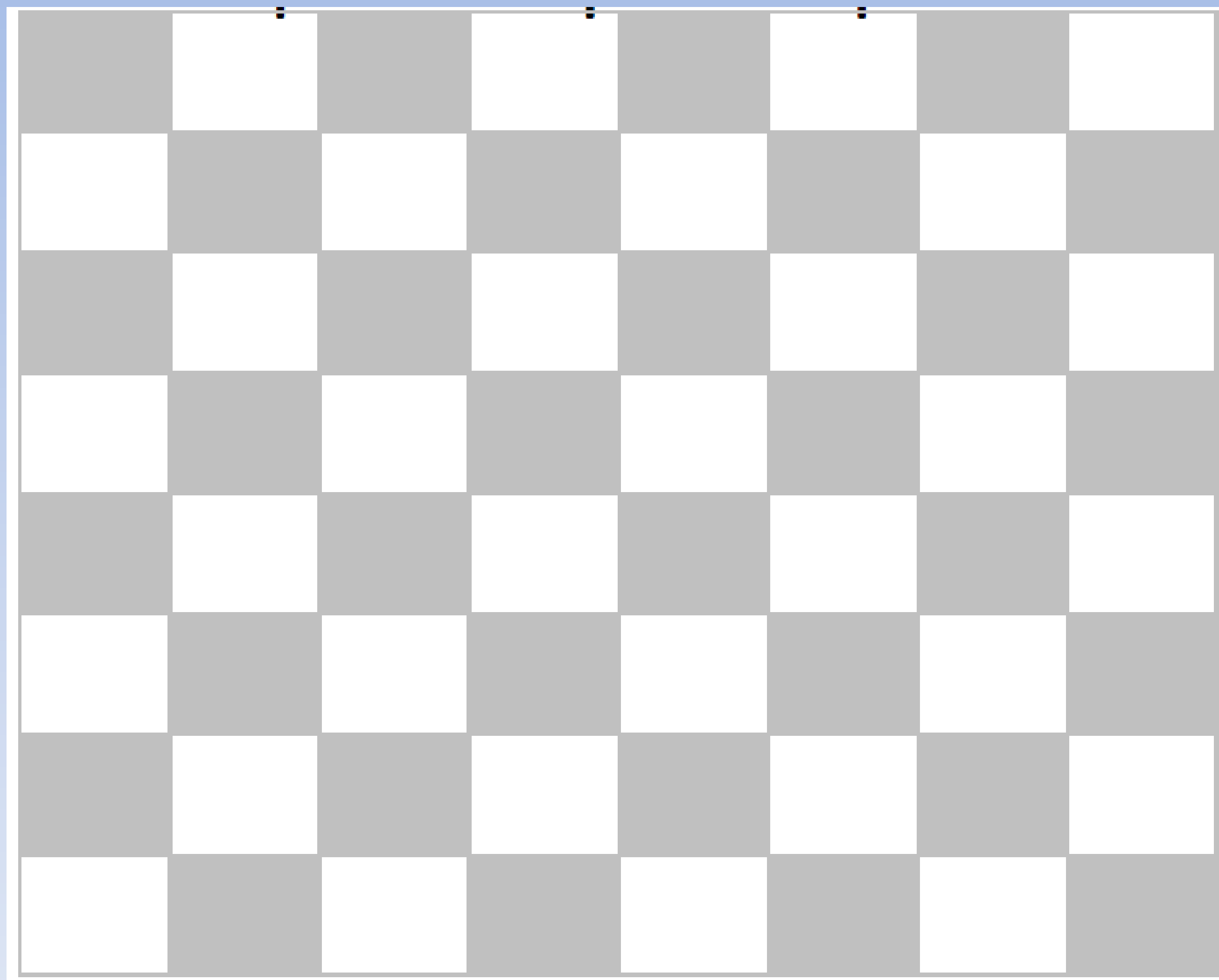


# Traveling Salesman Problem

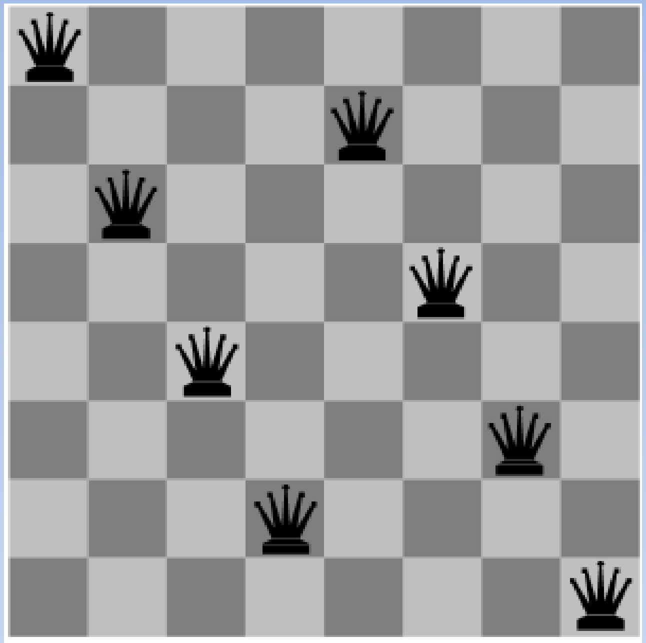
- GOAL:
  - Find the shortest tour that visits all cities without visiting any city twice and return to starting point.
- STATE:
  - Sequence of Cities Visited.
- Start State:
  - First City (A)
- Solution:
  - Complete Tour
- Transition Model:
  - {a, c, d} -> {a, c, d, X | X ∉ a, c, d}
  - don’t revisit state



# 8-Queens Problem

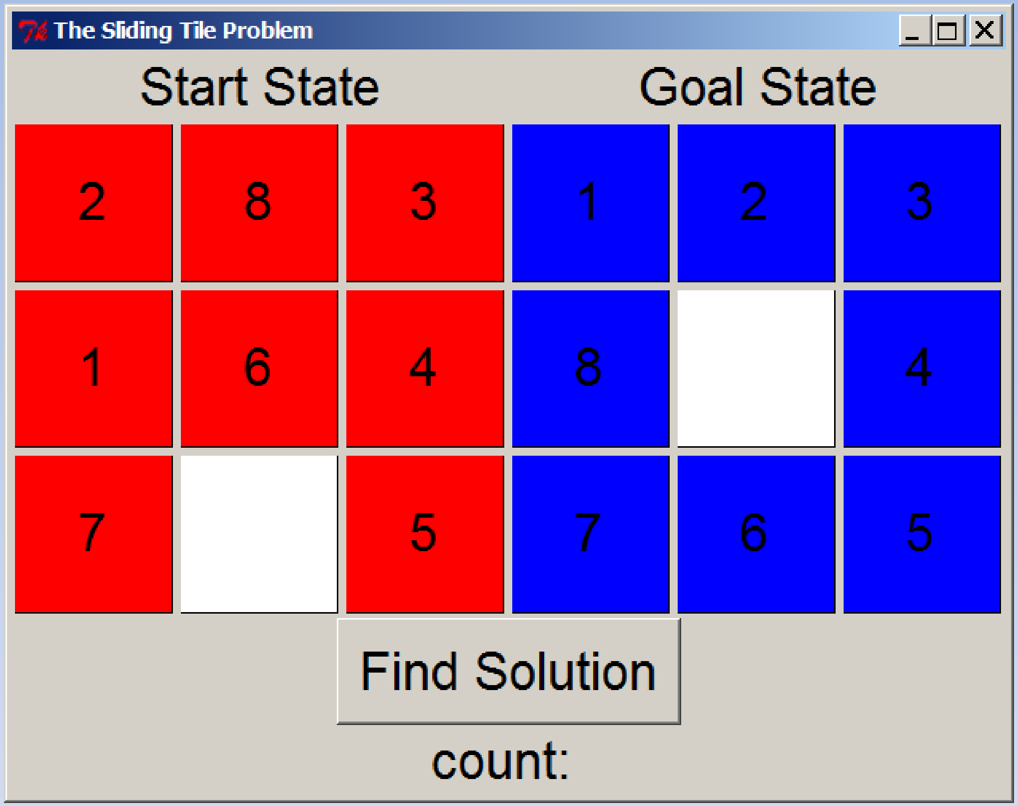


# 8-Queens



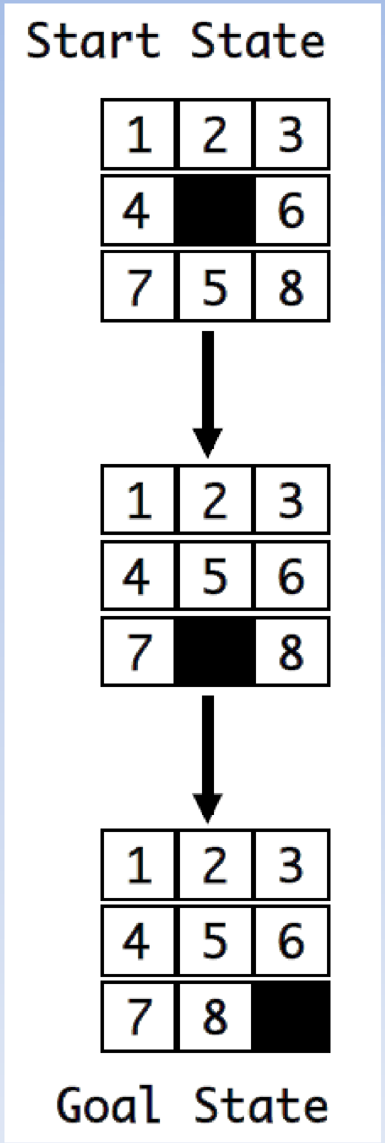
- States:
  - Any arrangement of n<=8 queens
  - OR: Arrangements of n<=8 queens:
    - In leftmost n columns, 1 per column,
    - such that no queen attacks any other.
- Initial State:
  - No Queens on board.
- Actions:
  - Add Queen to empty square.
  - OR: Add Queen to leftmost empty square such that it is not attacked by other Queens.
- Goal Test:
  - 8 Queens on board
  - None Attacked
- Path Cost: 1 per move

# The Sliding Tile Problem

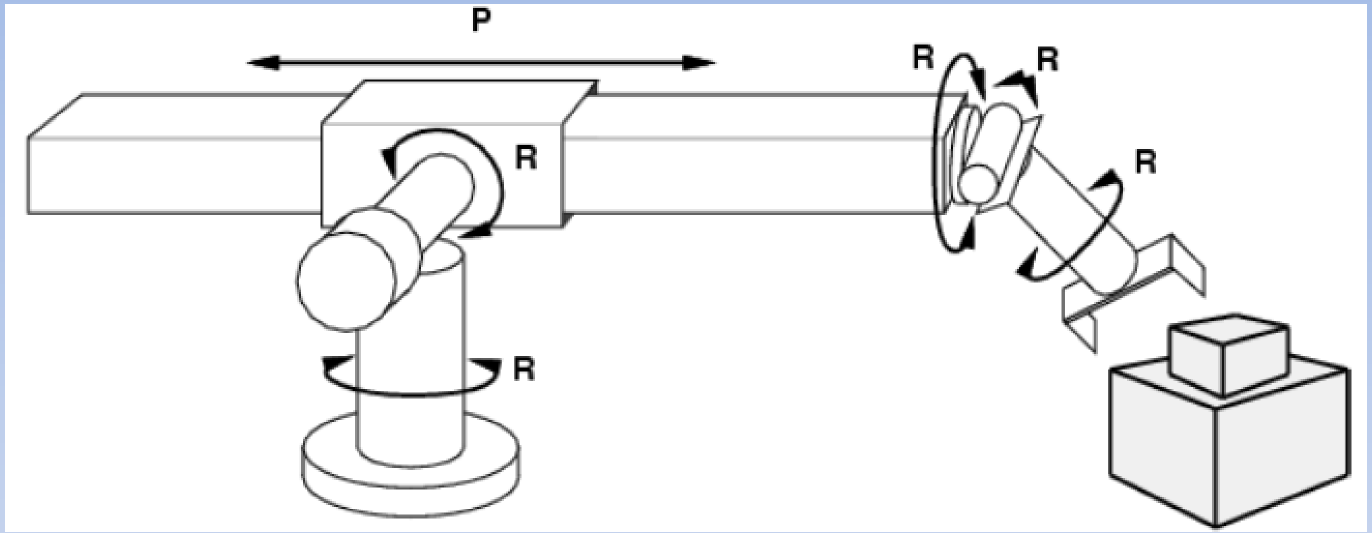



- Actions:
  - Up, Down, Left, Right
  - OR: move(x, loc_y, loc_z)

# The “8-Puzzle” Problem



# 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

# Formulating Problems; Another Angle

- Problem types
  - Satisficing: 8-queen
  - Optimizing: Traveling salesperson
- Object sought
  - Board configuration,
  - Sequence of moves
  - A strategy (contingency plan)
- Satisfying leads to optimizing since “small is quick”
- For traveling salsperson
  - Satisficing easy, optimizing hard
- Semi-optimizing
  - Find a good solution

# Searching the State Space

- States, operators, **control strategies**
- The search space graph is implicit
- The control strategy generates a small search tree.
- Systematic search
  - Do not leave any stone unturned
- Efficiency
  - Do not turn any stone more than once

# Question 3.9

- Missionaries and Cannibals Problem
  - 3 missionaries and 3 cannibals are on one side of a river, along with a boat that can hold one or two people.
  - Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.
- This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).
- Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution.

# Question 3.9

- Missionaries and Cannibals Problem
  - 3 missionaries and 3 cannibals are on one side of a river, along with a boat that can hold one or two people.
  - Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.
- This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).
- Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution.

# Question 3.9

- Here is one possible representation:
  - A state is a six-tuple of integers listing the number of missionaries, cannibals, and boats on the first side, and then the second side of the river.
  - The goal is a state with 3 missionaries and 3 cannibals on the second side.
  - The cost function is one per action, and the successors of a state are all the states that move 1 or 2 people and 1 boat from one side to another.

# Question 3.9

- Representation = (# of Missionaries on left, # of Cannibals on left, # of boats on left, # of Missionaries on right, # of Cannibals on right, # of boats on right
- Initial State = (3, 3, 1, 0, 0, 0)
- After MoveRight(1, 1):
  - (2, 2, 0, 1, 1, 1)

# Question 3.9

- The search space is small, so any optimal algorithm works.
  - It suffices to eliminate moves that circle back to the state just visited.
  - From all but the first and last states, there is only one other choice.
- Why do you think people have a hard time solving this puzzle, given that the state space is so simple?
  - It is not obvious that almost all moves are either illegal or revert to the previous state.
  - There is a feeling of a large branching factor, and no clear way to proceed.

```
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
```

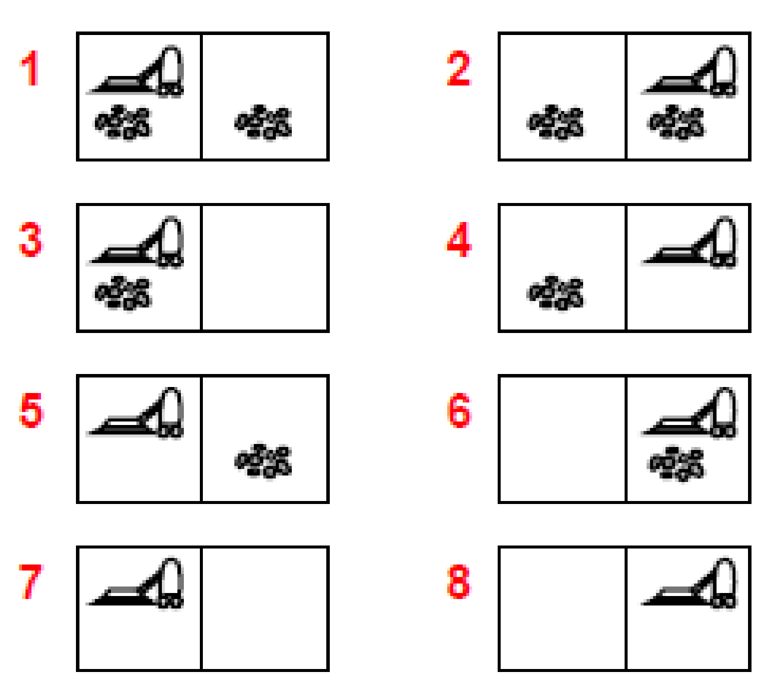
**Figure 3.1** A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

- Offline Problem Solving
  - Solution Executed EYES-CLOSED.
- Online problem solving involves acting without complete knowledge.

# Problem Types

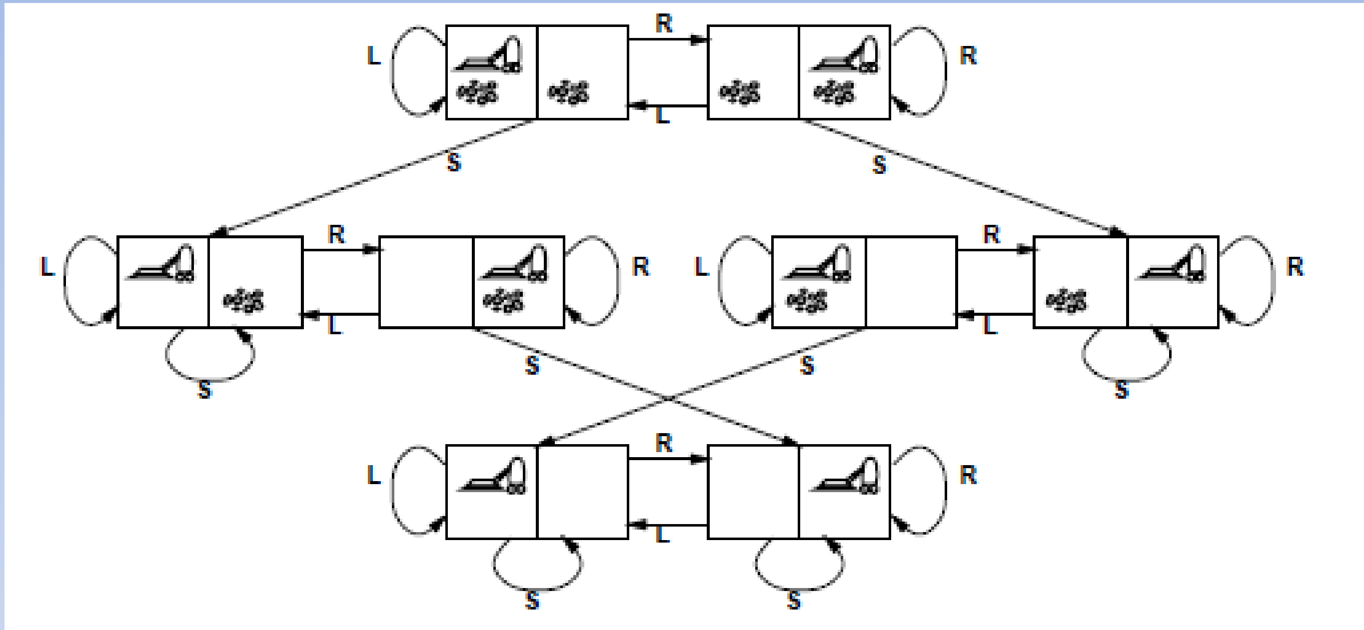
- Deterministic & Fully Observable
  - Single-State Problem
  - Action Sequence Solution
- Non-observable
  - Conformant Problem
  - Agent may have no idea where it is
  - Solution (if any) is a sequence
- Nondeterministic and/or partially observable
  - Contingency Problem
  - Percepts provide new information about current state
  - Solution is a contingent plan or a policy
  - Often interleave search and execution
- Unknown State Space: Exploration Problem (“online”)

# Vacuum World




- Single-state:
  - Start in #5
  - Solution=[Right, Suck]
- Conformant:
  - Start={1,2,3,4,5,6,7,8}
  - Right goes to {2,4,6,8}
  - Solution=[Right, Suck, Left, Suck]
- Contingency: Start = #5
  - Murphy’s Law: ‘Suck’ can dirty clean carpet
  - Local sensing: dirt, location only
  - Solution =[Right, if Dirt then Suck]

# Vacuum World State Space Graph




- States: integer dirt and robot locations (ignore dirt amount, etc.)
- Actions: Left, Right, Suck, NoOp
- Goal Test: no dirt
- Path Cost: 1 per action (0 for NoOp)

# Tree Search Algorithms

- Basic Idea:
  - Offline, Simulation exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

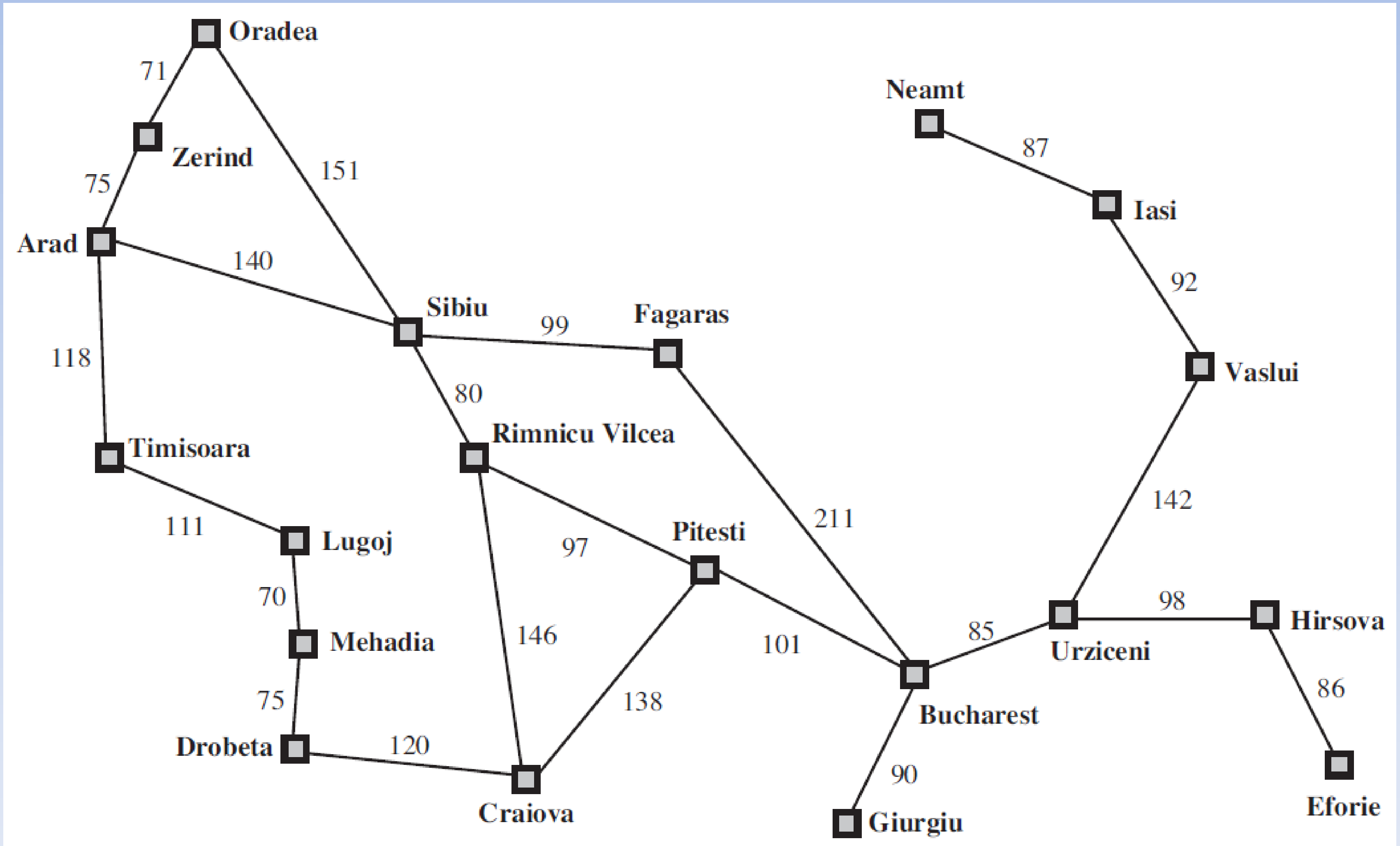
# Tree Search Algorithms (Fig 3.7)

**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

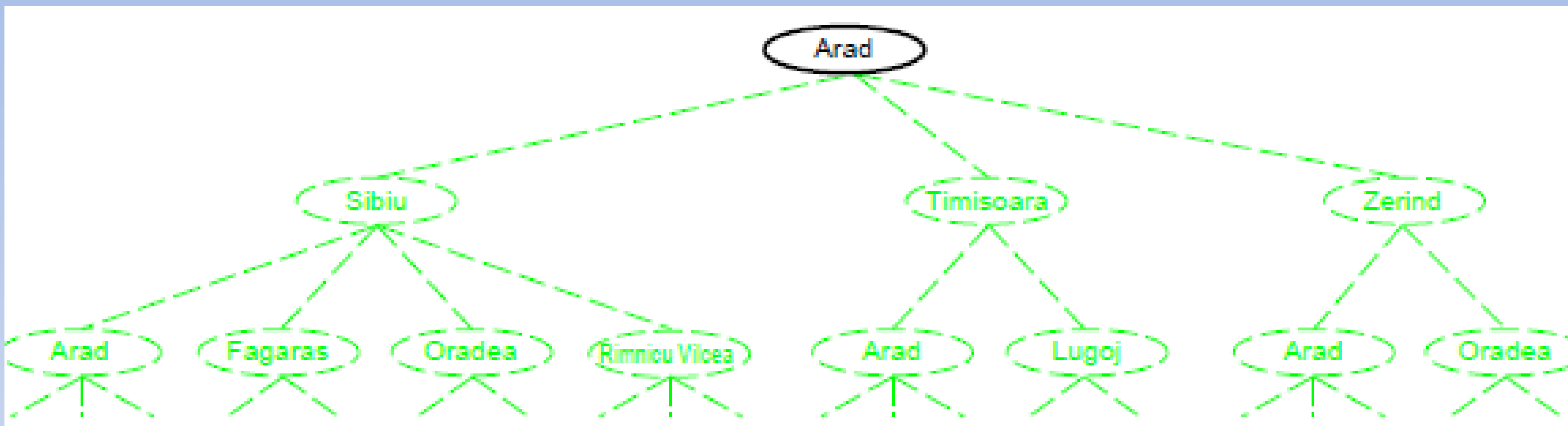
---

**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***

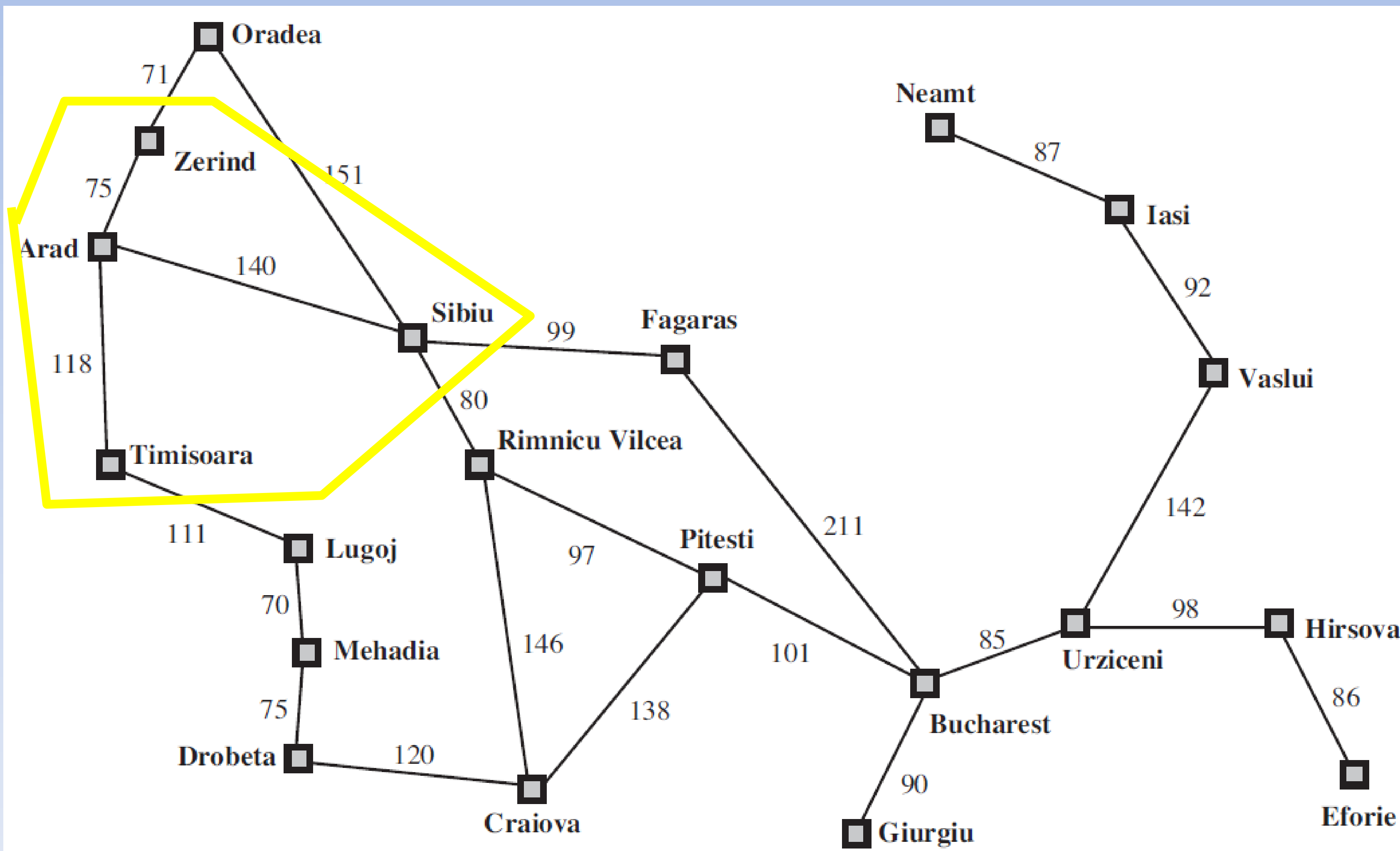
# Map


Oradea
71
Neamt
Zerind
151
87
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
Pitesti
97
70
98
Hirsova
85
146
Mehadia
101
Urziceni
86
75
138
Bucharest
Drobeta
120
90
Craiova
Giurgiu
Eforie

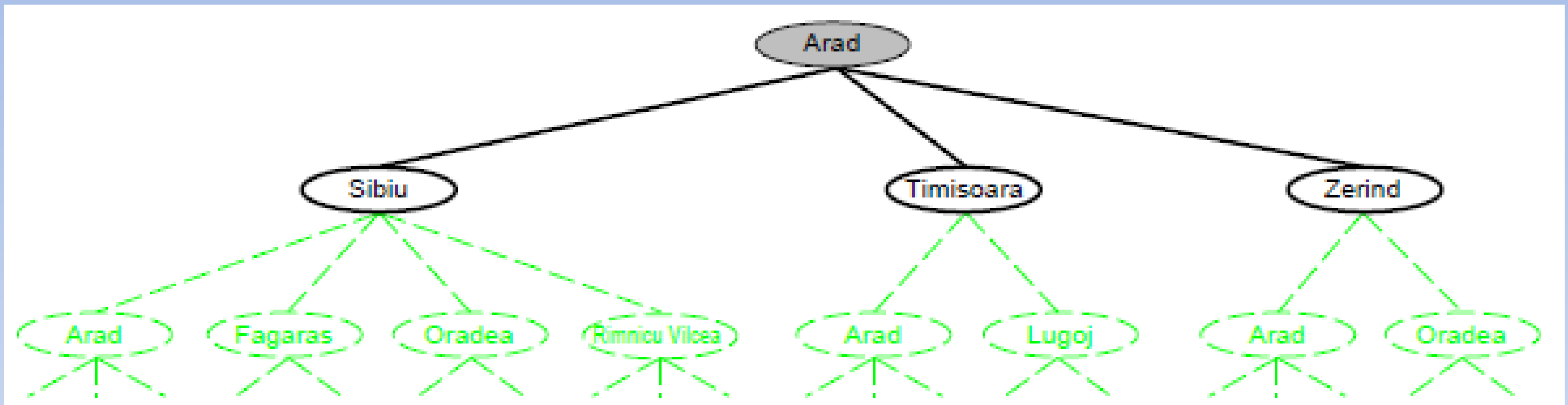
# Tree Search Example



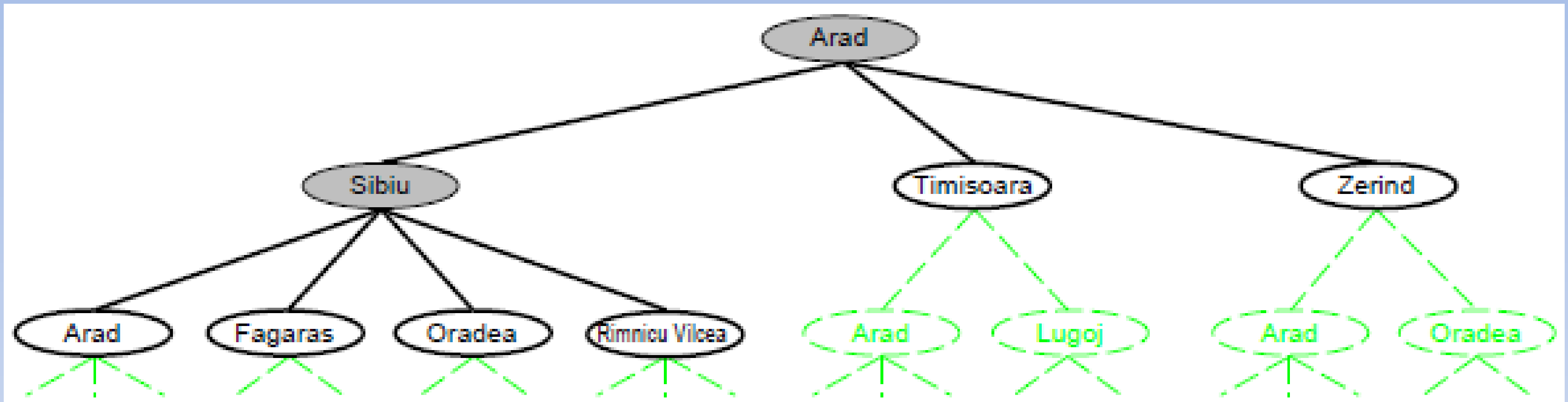
# Map


Oradea
71
Neamt
Zerind
151
87
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
Mehadia
146
101
85
Urziceni
Hirsova
75
138
Bucharest
86
Drobeta
120
90
Craiova
Giurgiu
Eforie

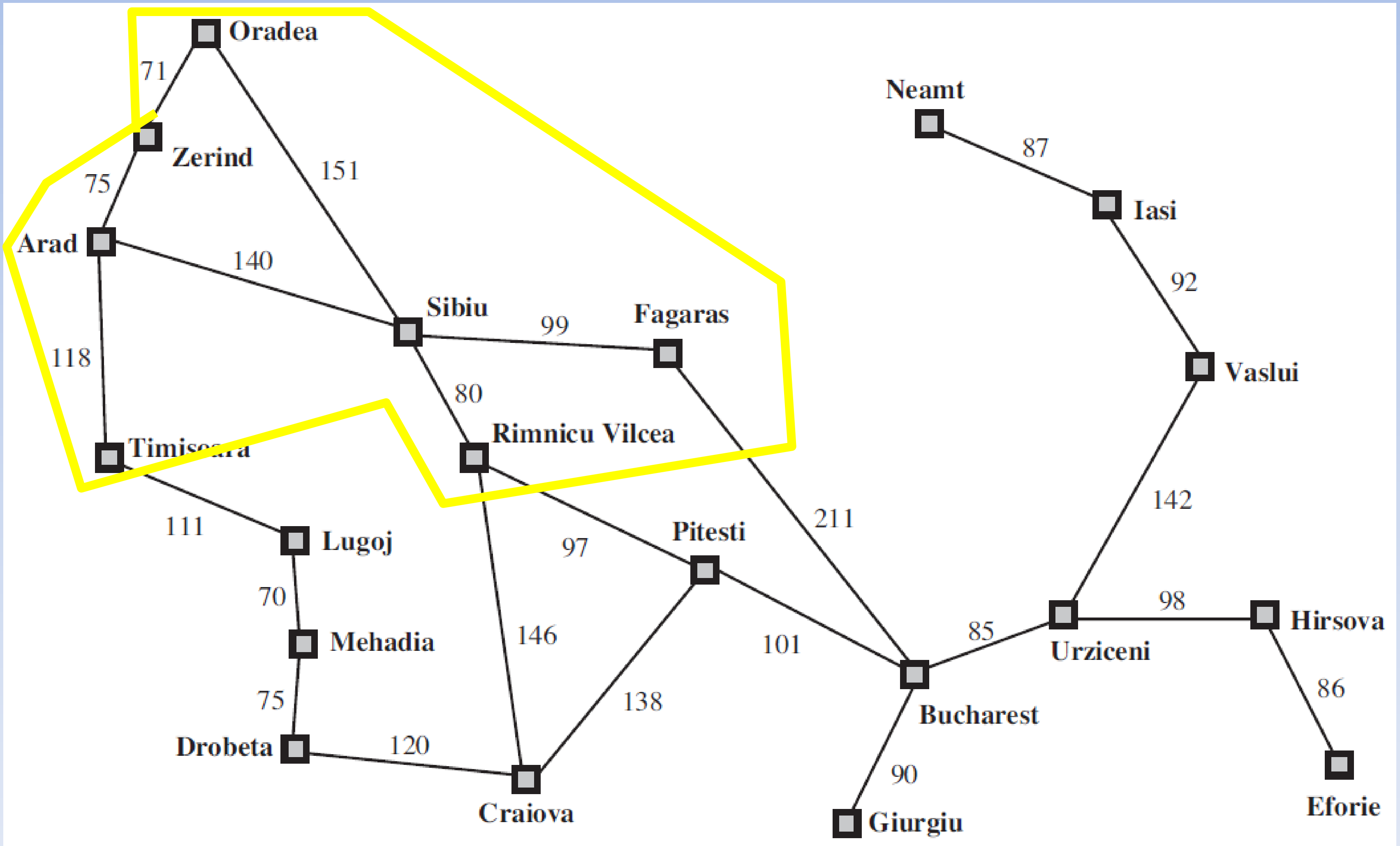
# Tree Search Example



# Tree Search Example



# Map



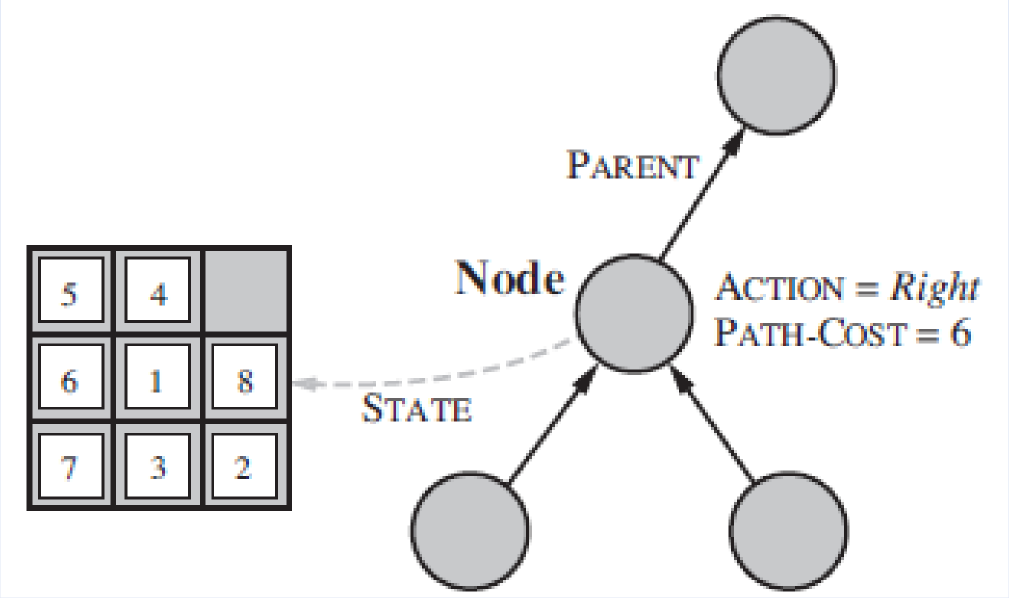
Oradea
71
Neamt
Zerind
151
87
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
Hirsova
Mehadia
146
101
85
Urziceni
75
138
86
Bucharest
Drobeta
120
90
Craiova
Giurgiu
Eforie

# Searching State Space

- Search Space Graph is Implicit

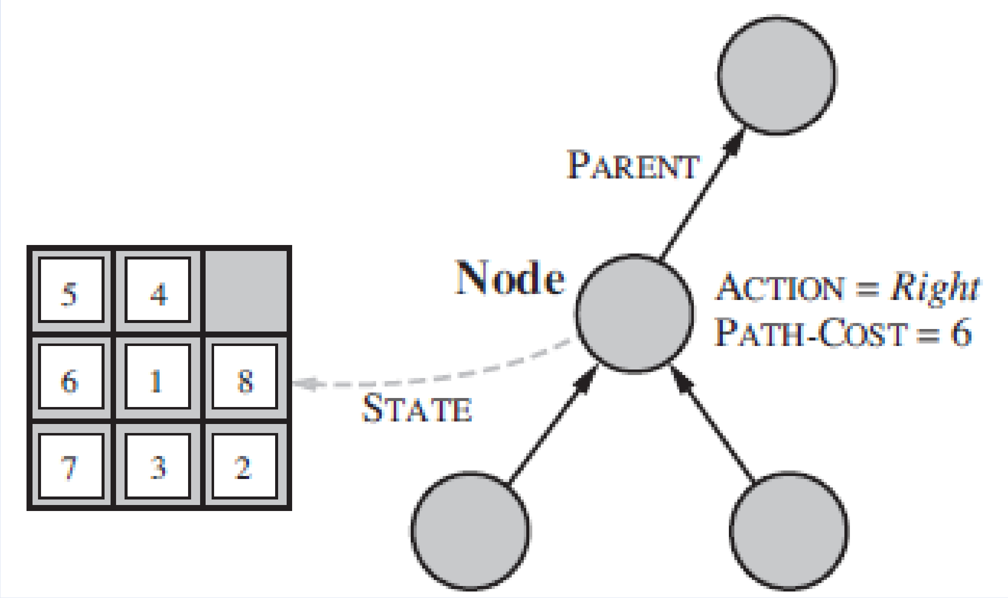
# States Versus Nodes

- State: Representation of a physical configuration.
- Node: Data structure constituting part of a search tree, including:
  - Parent, Children, Depth, or Path Cost (g).
- States do not have parents, children, depth or path cost.



# States Versus Nodes

**function** CHILD-NODE(*problem*, *parent*, *action*) **returns** a node
  **return** a node with
    STATE = *problem*.RESULT(*parent*.STATE, *action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)



# Search Strategies

- Strategies are defined by picking the Order of Node Expansion
- Strategies are evaluated by:
  - Completeness: does it always find a solution if one exists?
  - Time Complexity: number of nodes generated/expanded.
  - Space Complexity: maximum nodes of nodes in memory.
  - Optimality: Does it always find a least-cost solution.
- Time and Space Complexity are measured by:
  - b: maximum branching factor of search tree.
  - d: depth of least-cost solution
  - m: maximum depth of the state space (may be $\infty$)

# Uninformed Search Strategies

- Uninformed Strategies: use only the information available to the problem definition.
  - Breadth-first Search
  - Depth-first Search
  - Uniform-Cost Search
  - Depth-Limited Search
  - Iterative Deepening Search

# Remembering Graphs

- Lets look more formally at graphs
- Graph G=(V, E)
  - V: Set of vertices
  - E: Set of edges
- Two representations
  - Adjacency Lists
  - Adjacency Matrix