Chapter 3

Solving Problems by Search: Problem Formulation

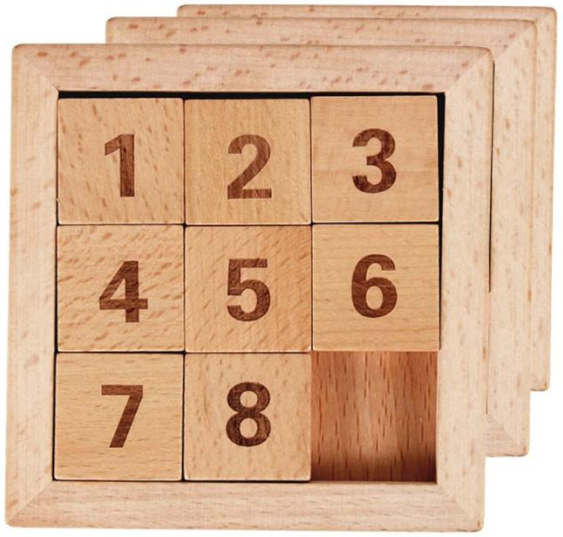
Introduction

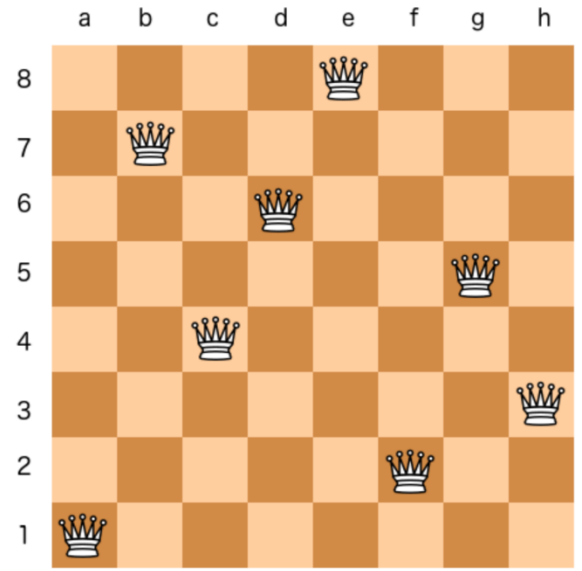
- The simplest reflex agents base their actions on a direct mapping from states to actions.
- Such agents cannot operate well when this mapping is too large to store and would take too long to learn.
- Goal-based agents consider future actions and the desirability of their outcomes.
- One kind of goal-based agent is called a **problem-solving agent**.
- Problem-solving agents use **atomic** representations

Problem solving agents

- Many problems in AI can be modeled as search problems:

1. Toy problems:
   - Puzzle-solving (8, 15-puzzle)
   - Vacuum world
   - 8-queens





Problem solving agents

- Many problems in AI can be modeled as search problems:

2. Real world problems
   - Route-finding in airline travel planners
   - Travelling Salesman Problem:
   - planning drill movements for automatic circuit board drills
- Determine the optimal sequence of drill movements to create holes in a circuit board.
  - Determine the optimal sequence of drill movements to create holes in a circuit board.
    - **States**: Positions of the drill and the set of holes drilled so far.
    - **Actions**: Move the drill to a new position and drill a hole.
    - **Goal**: Drill all required holes with minimal movement time or distance.
    - **Cost**: Time or distance traveled by the drill.

Problem solving agents

- VLSI layout (cell layout and channel routing)
  - **Problem: Arrange components (cells) on a chip and route connections between them efficiently.**
    - **States**: Current arrangement of cells and routing paths.
    - **Actions**: Move a cell or add a routing path.
    - **Goal**: Minimize the chip area and ensure all connections are properly routed.
    - **Cost**: Chip area, wire length, or number of routing layers.

Problem solving agents

- Automatic manufacturing (assembly sequencing)
- **Problem**: Determine the optimal sequence of steps to assemble a product.
- **Modeling as a Search Problem**:
  - **States**: Current state of the assembly (which parts are assembled).
  - **Actions**: Add a new part or perform an assembly step.
  - **Goal**: Complete the assembly with minimal time or cost.
  - **Cost**: Time, cost, or number of steps.

Search Algorithms

- **Uninformed** search algorithms: are given no information about the problem other than its definition
  - Solve problem but not efficiently

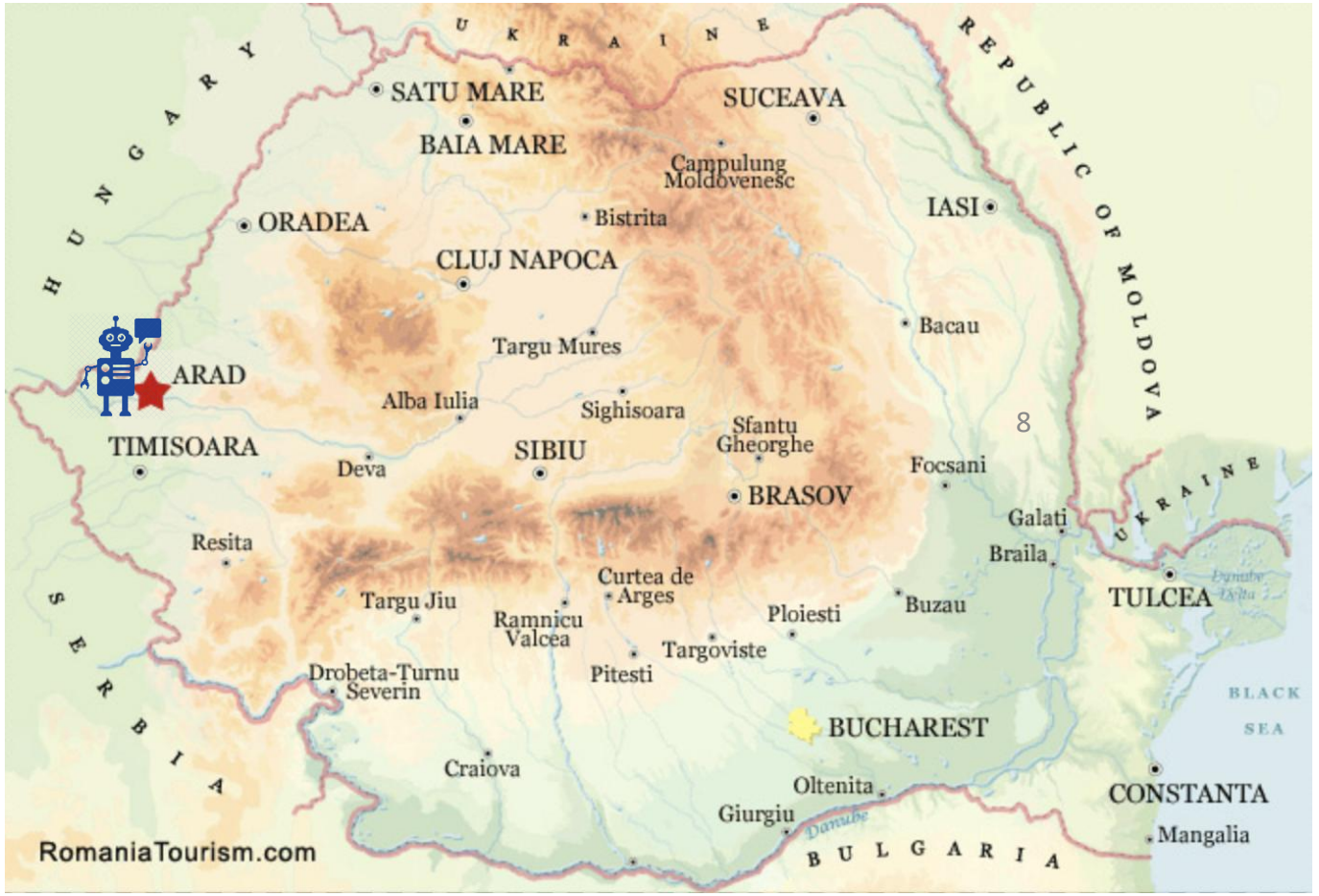
- **Informed** search algorithms: are given some guidance on where to look for solutions
  - More efficient

Example

Goal: Getting to Bucharest

Goal formulation organizes behavior by limiting objectives/actions

Problem formulation decides what actions and states to consider, given a goal:
actions at the level of “move the left foot forward an inch” vs “move to city”



The process of looking for a sequence of actions that reaches the goal is called **search**.

Getting to Bucharest

```
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 Algorithms

- 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.

Problem definition: Modeling

- A search problem is defined by:
  1. **Initial state** (**State**: information necessary to solve the problem)
  2. **Actions**
  3. **Transition Model:** *Result(Current State, Action) = New State*
  4. **Goal state(s)**: can be defined *Explicitly* (e.g. 8-puzzle) or *Implicitly* (e.g. chess)
  5. **Step cost** (optional): the cost of each action

State Space

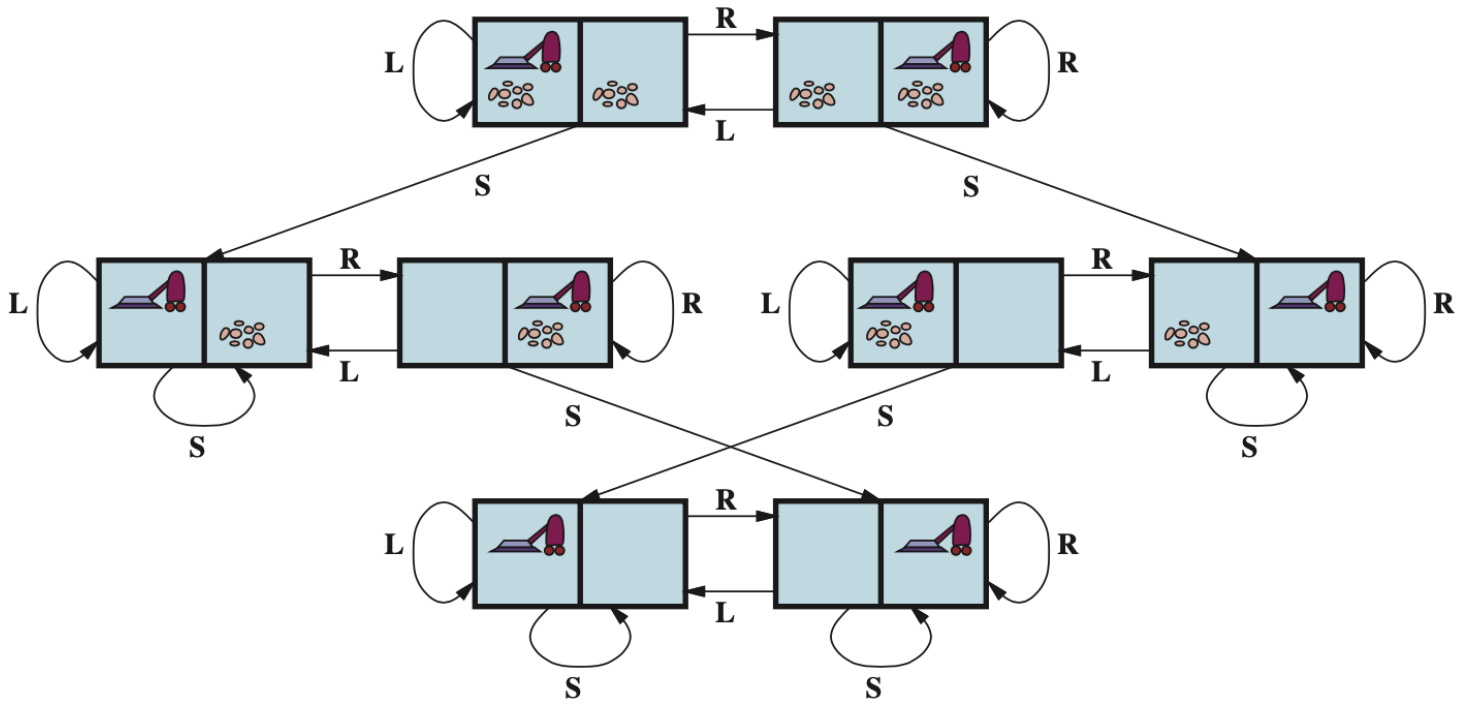
- The state space is a graph:
  - *nodes* are *states*
  - *links* between nodes are *actions*
  - *path* is a sequence of *states* connected by a sequence of *actions*

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

State space

- **State space** is the set of all states **reachable** from the initial state by any sequence of actions.
- The state space constitutes a directed **graph:**
  - Nodes are states
  - Edges are actions.
- The state space can be very large :
  - Eight puzzle: $\frac{9!}{2} = 181{,}440$ (why divide by 2?)
  - Theorem Proving: Infinite
  - Chess: 10^{47} possible states, 10^{120} possible games (in an average length game)
  - Checkers: 10^{20} possible states, 10^{31} possible games

Example: the vacuum world



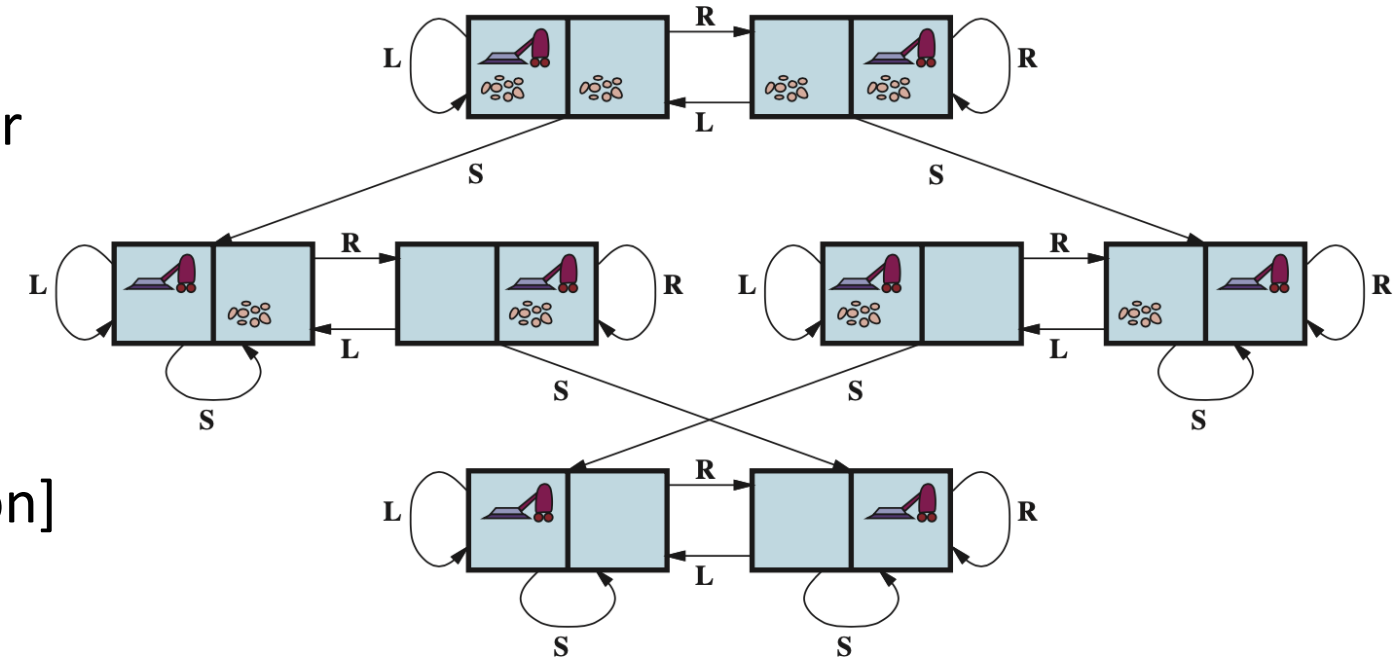
- **States**:

Agent location on map, dirt (yes,no), available map positions

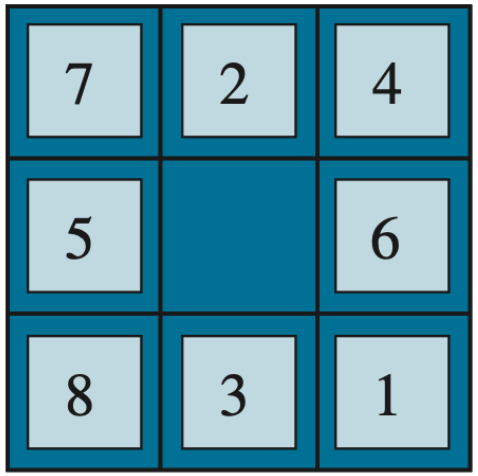
2 * 2 * 2 = 8

Example: the vacuum world

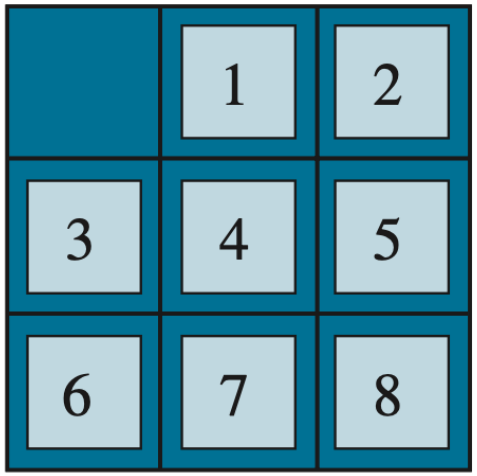
- **Initial state:** any
- **Actions:** Left, Right, Suck
- **Transition model**: actions have their expected effects, except:
  - *Left* in the leftmost square
  - *Right* in the rightmost square
  - *Suck*ing in a clean square
- **Goal states:** [(Clean,Clean), Location]
- **Cost:** 1 per action



Example: the 8-puzzle



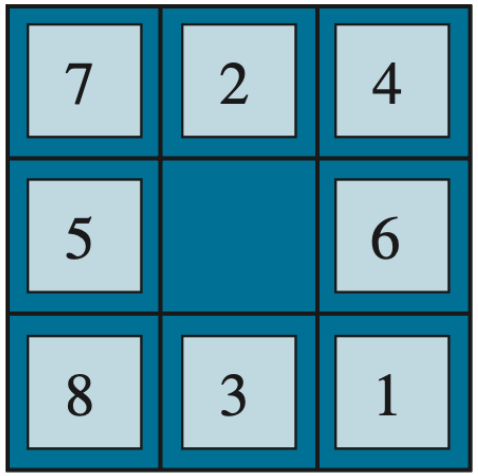
Start State



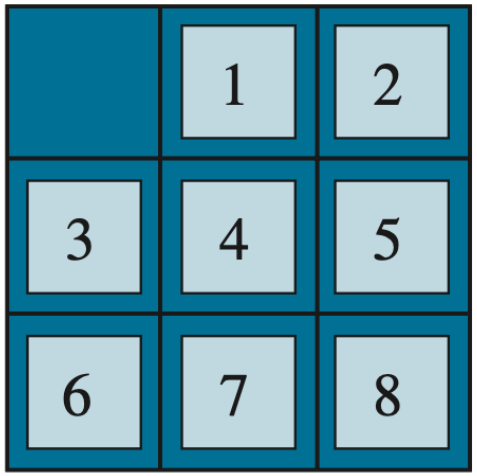
Goal State

- **State space**: 2D array representing the positions of the tiles, $n! = 9!$
- **Initial state**: given
- **Actions**: move blank left, right, up, down
- **Transition model**: Given a state and action, this returns the resulting state
- **Goal states**: all tiles ordered from 1 to 8 and the blank tile in the bottom right corner
- **Cost**: 1 per action

Example: the 8-puzzle



Start State



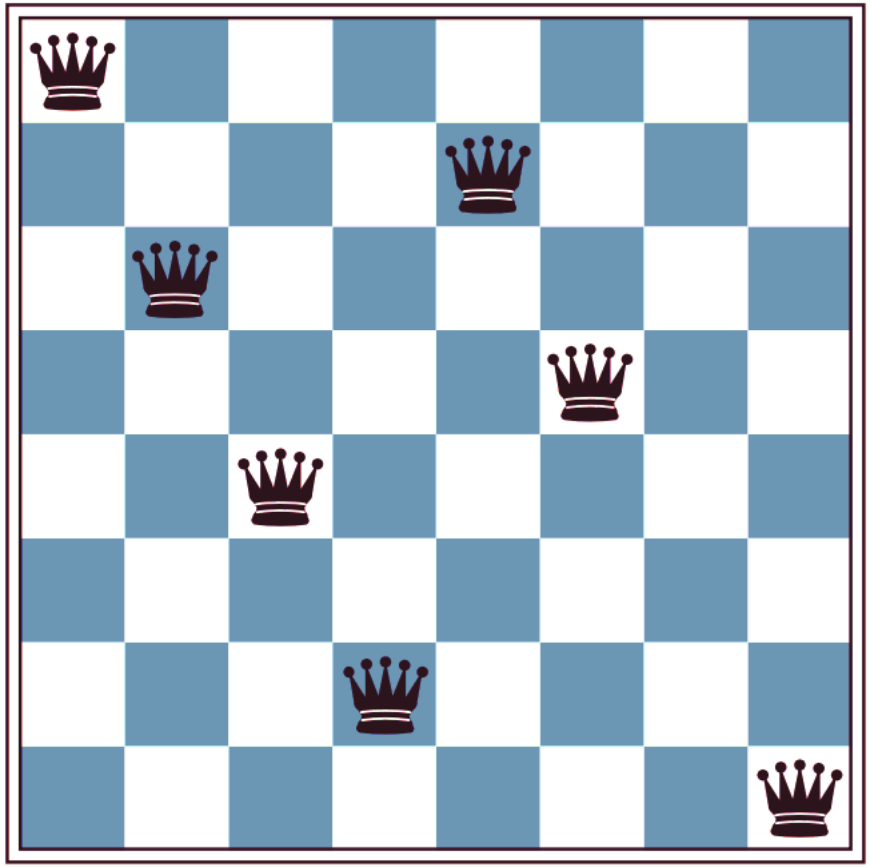
Goal State

Note: state space is two disconnected components of the same size, so realistically given an initial state, can reach only $\frac{9!}{2} = 181{,}440$

- **State space**: 2D array representing the positions of the tiles, $n! = 9!$
- **Initial state**: given
- **Actions**: move blank left, right, up, down
- **Transition model**: Given a state and action, this returns the resulting state
- **Goal states**: all tiles ordered from 1 to 8 and the blank tile in the bottom right corner
- **Cost**: 1 per action

Example: 8 Queens

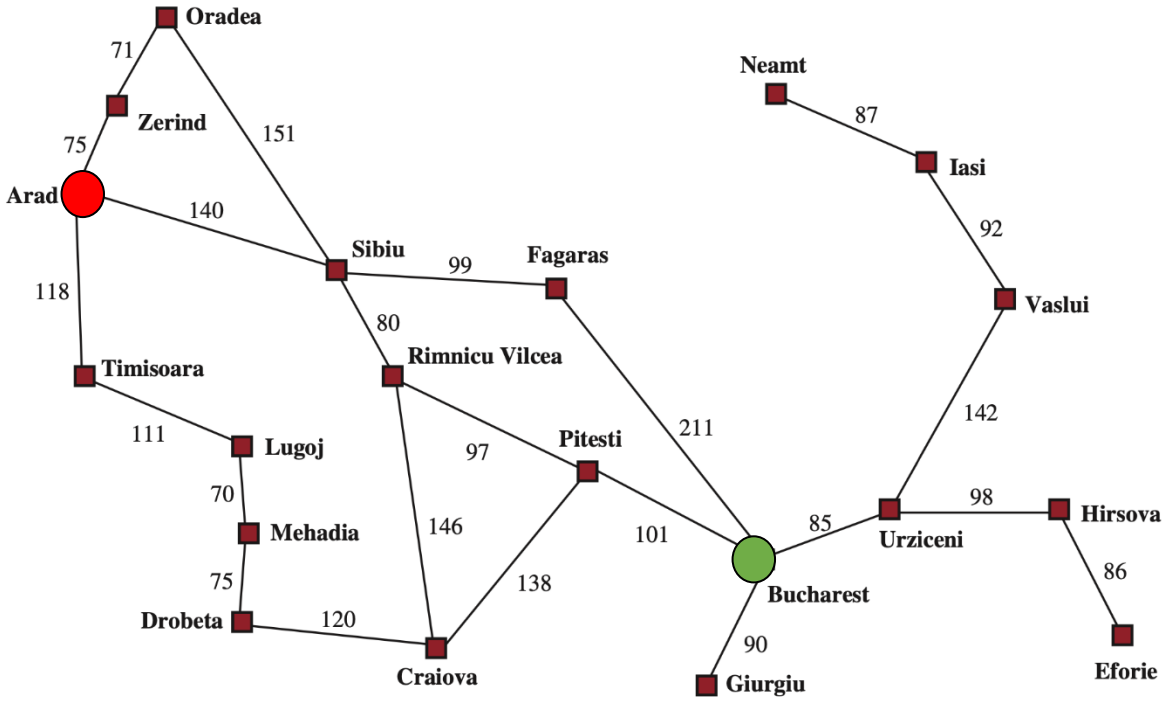
- **States**: Any arrangement of 0 to 8 queens on the board is a state
- **Initial state**: No queens on the board
- **Actions**: Add a queen to any empty square
- **Transition model**: Returns the board with a queen added to the specified square
- **Goal test**: 8 queens are on the board, none attacked.



This is an **incremental formulation**, can also have **complete-state formulation**

Example: route navigation
How to find a route from Arad to Bucharest?

- **States**: cities
- **Initial state**: Arad
- **Actions**: move from one city to an adjacent city
- **Transition Model**: the current location is the new location chosen by the action
- **Goal states**: Bucharest
- **Cost**: the distance between two adjacent cities



Missionaries & Cannibals

- 3 missionaries and 3 cannibals need to cross a river
- 1 boat that can carry 1 or 2 people
- Find a way to get everyone to the other side, without ever leaving the group of missionaries in one place outnumbered by cannibals in that place
- Check on this link:
  - http://www.learn4good.com/games/puzzle/boat.htm

Problem Formulation

- States:
  - <m, c, b> representing the # of missionaries and the # of cannibals, and the position of the boat
- Initial state:
  - <3, 3, 1>
- Actions:
  - take 1 missionary, 1 cannibal, 2 missionaries, 2 cannibals, or 1 missionary and 1 cannibal across the river
- Transition model:
  - state after an action
- Goal test:
  - <0, 0, 0>
- Path cost:
  - number of crossing

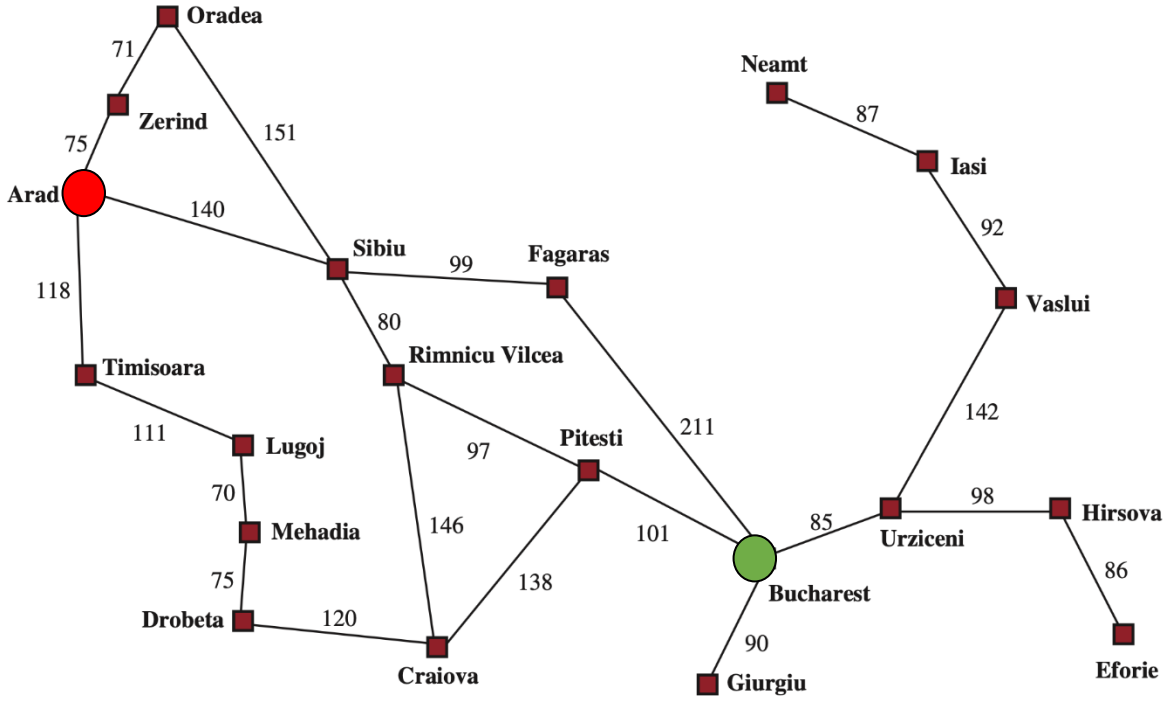
Searching: Optimization

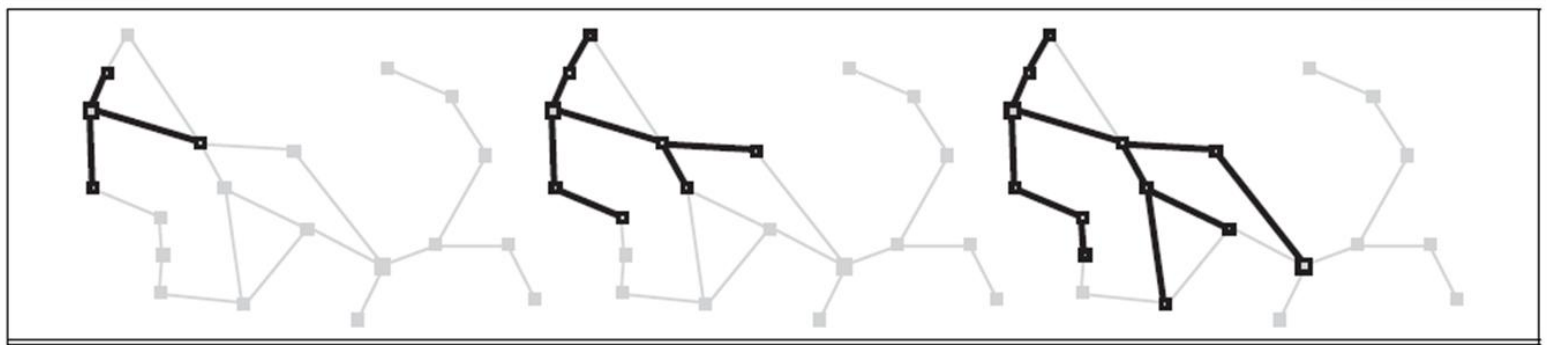
The principle:

1. Start from the initial state
2. Test if it is a goal
3. Generate the **successors** (or children) (**node expanding**)
4. Attach the successors to the parent → result in a tree : **search tree**
5. Choose the next node to expand: **search strategy**
6. Go to 2

- Strategy = algorithm: different strategy means different algorithm

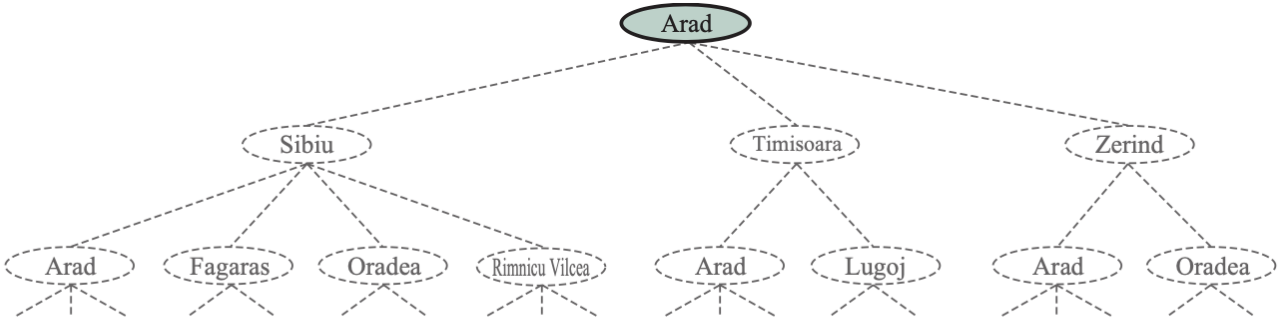
Searching: example

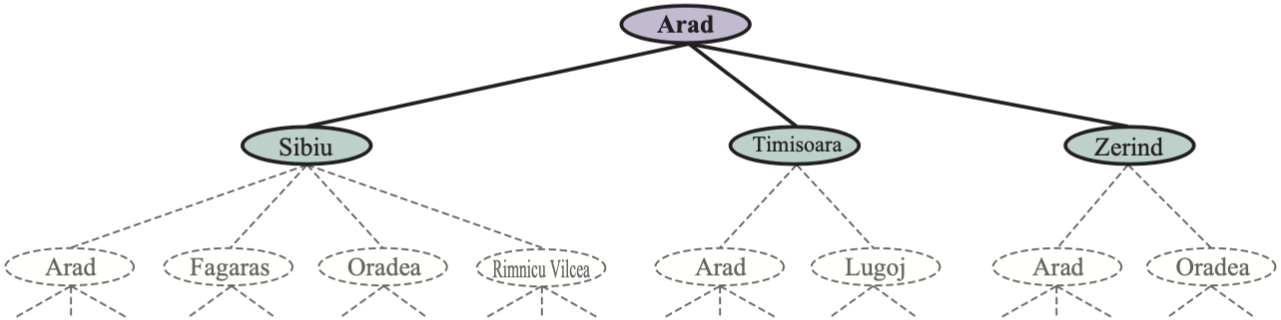




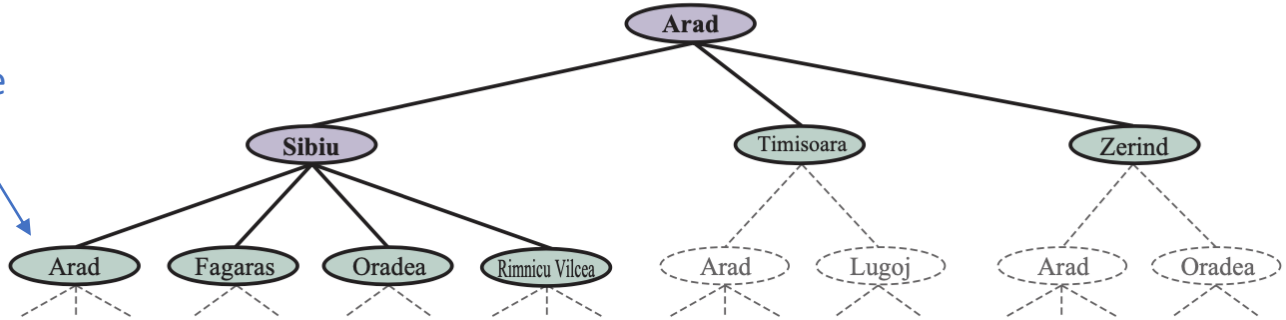
Search tree

Searching: example





Repeated State

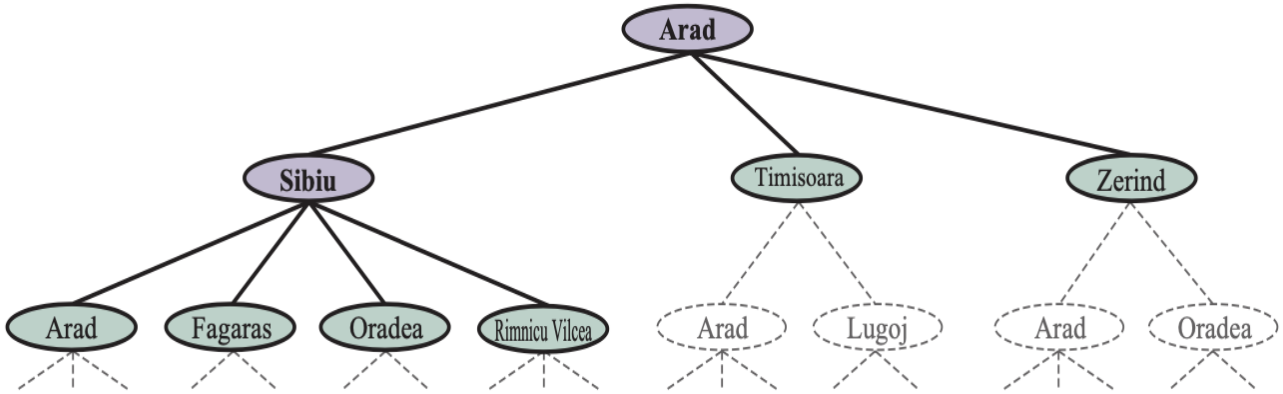


Green leaf nodes: Frontier nodes, Generated

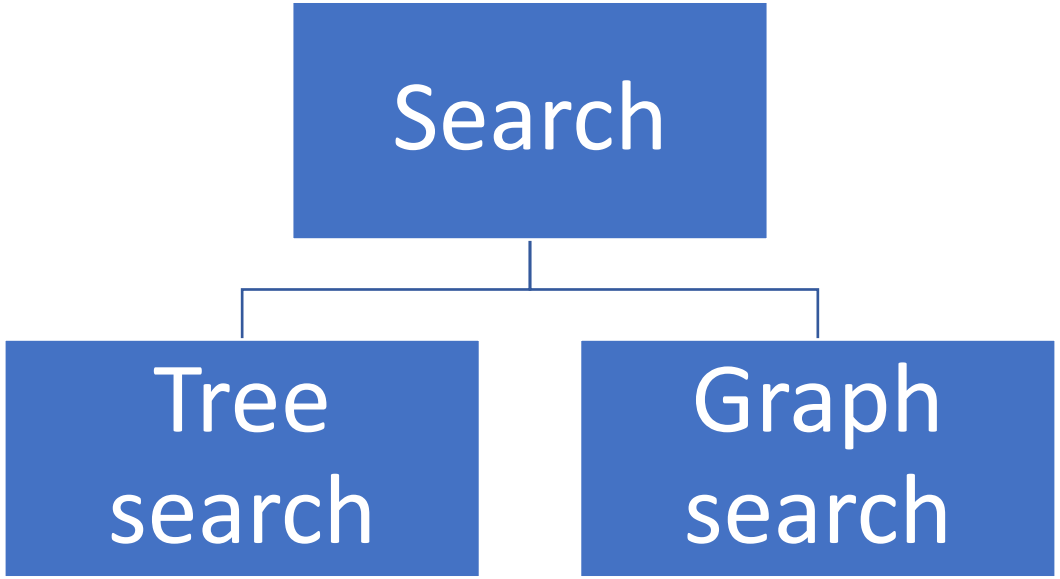
Lavender: Expanded

Terminology

- **Expanding** a node: generate its successor.
- **Explored set**: contains expanded nodes.
- **Frontier** (or **fringe**) contains nodes that have been generated but not yet expanded



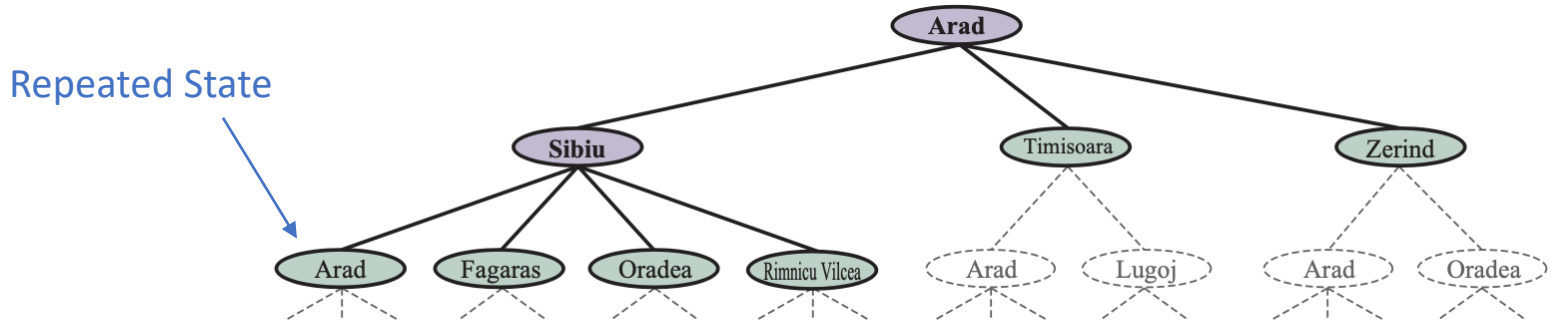
Tree search vs. graph search



- Tree search: allow repeated states.
- Graph search: avoid repeated states.

Tree Search: Repeated states ☹

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

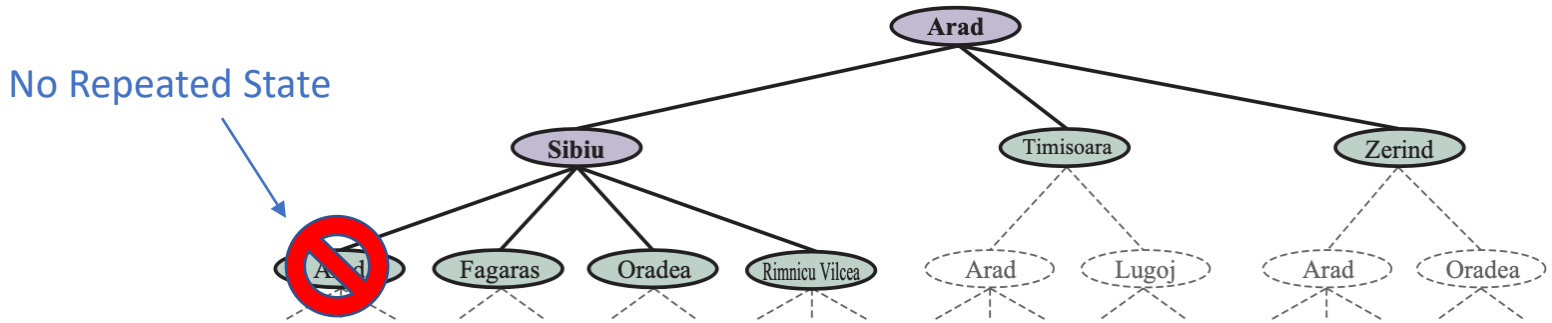
- Tree search is easy to implement
- Works only in certain search spaces: for instance trees
- If there are cycles it can loop forever (depending on the strategy)
- Duplicated states increase the size of the search space

Graph Search: No Repeated States ☺

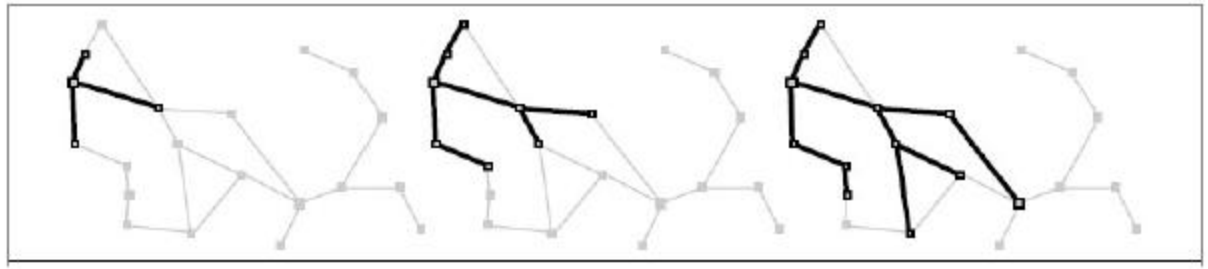
function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the 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

Graph Search

- In a graph search, the explored set is separated from the rest of the graph by the frontier.



Graph Search Examples



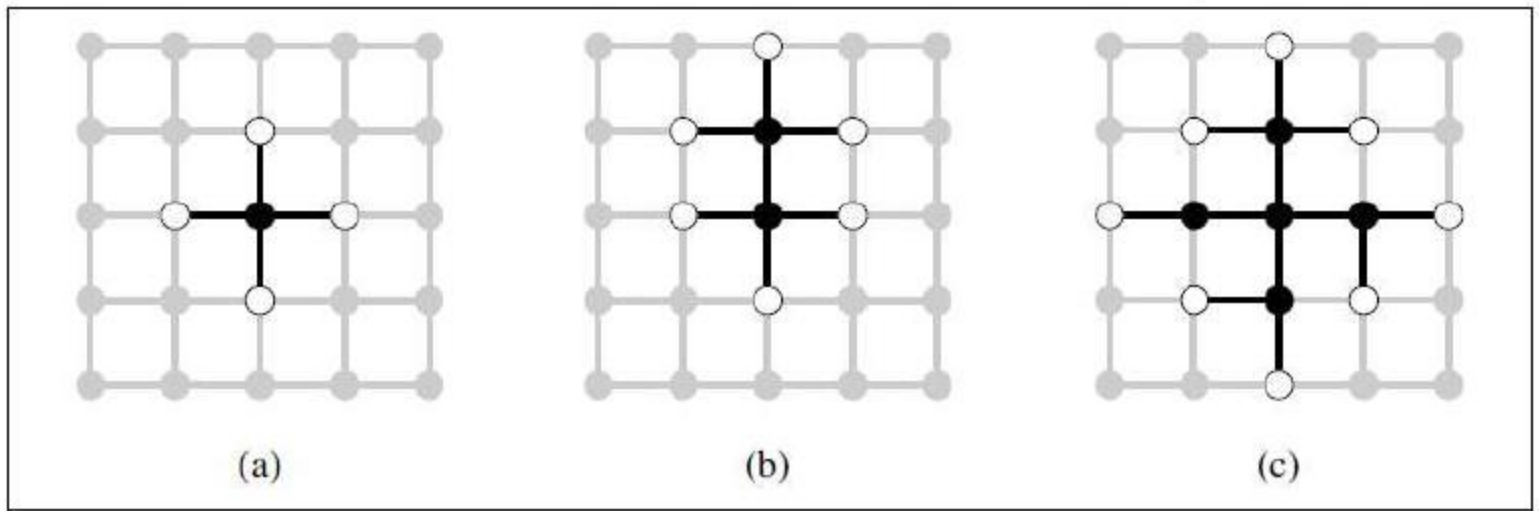


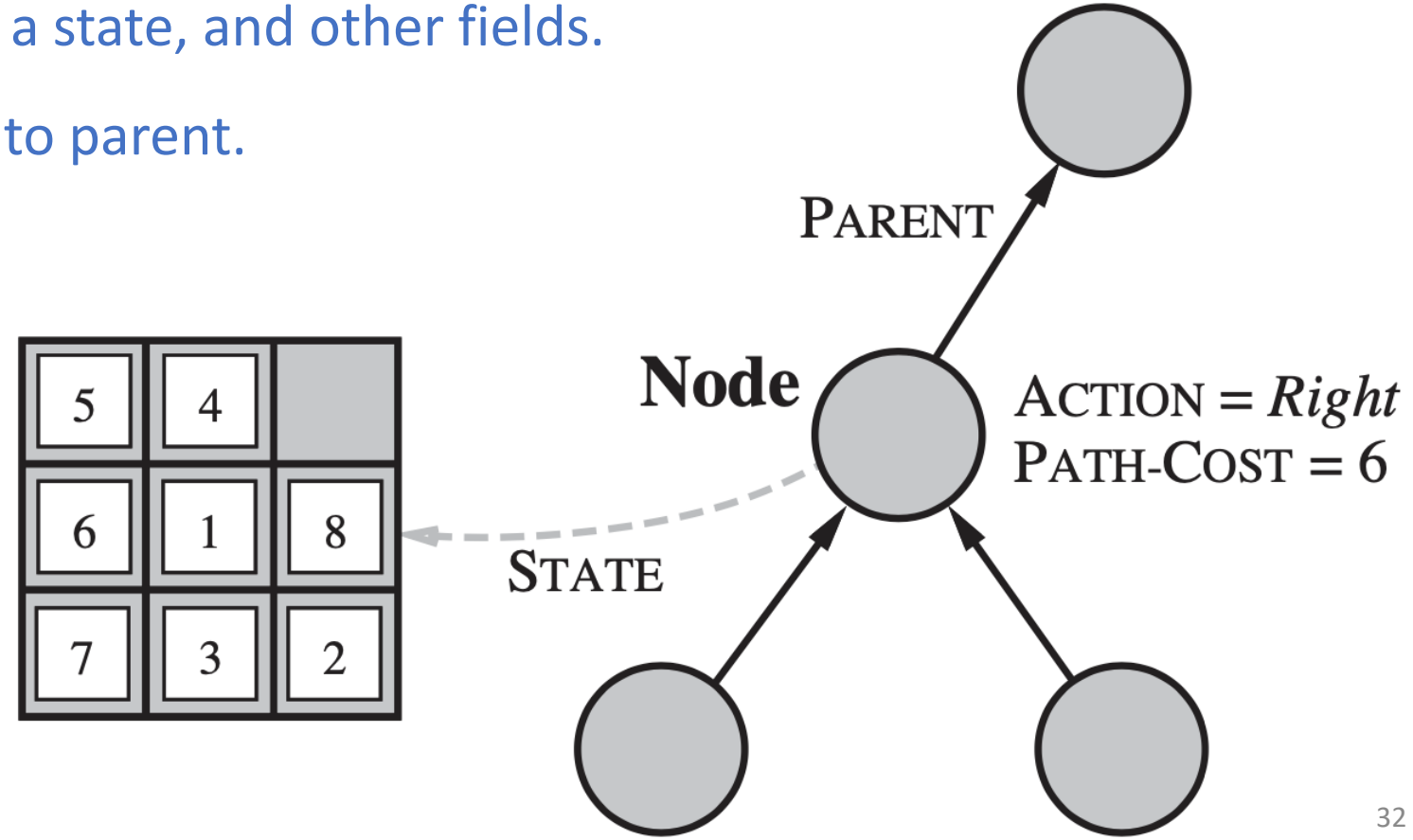
Figure 3.9 FILES: figures/graph-separation.eps. The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been 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

- 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

Node vs. State

- Nodes are the data structures from which the search tree is constructed.
- Each node has a parent, a state, and other fields.
- Arrows point from child to parent.



Searching: remarks

- The search space can be:
  - Explicit: map
  - Implicit: 8-puzzle, chess etc.
- The search tree is always explicit: it must be stored in memory somewhere.

Search Strategies

- 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?
  - optimality: does it always find a least-cost solution?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
- Time and space complexity are measured in terms of
  - ***b***: maximum branching factor of the search tree
  - ***d***: depth of the least-cost solution
  - ***m***: maximum depth of the state space (may be ∞)
- Search cost (time), total cost (time+space)

Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

1. Completeness: Is the algorithm guaranteed to find a solution when there is one?
2. Optimality: Does the strategy find the optimal solution?
3. Time complexity: How long does it take to find a solution?
4. Space complexity: How much memory is needed to perform the search?

Measuring problem-solving performance

- Time and space complexity are measured in terms of
  - $\boldsymbol{b}$: maximum branching factor of the search tree
  - $\boldsymbol{d}$: depth of the best solution
  - $\boldsymbol{m}$: maximum depth of the state space (may be infinite)