Module1

Source Book

Stuart J. Russell and Peter Norvig, “Artificial Intelligence”, 3rd Edition, Pearson,2015

Topics

- What is AI?
- Foundations of AI
- History of AI
- Agents, The structure of agents.
- Problem Solving Agents
- Example problems,
- Searching for Solutions,
- Uninformed Search Strategies: Breadth First search, Depth First Search

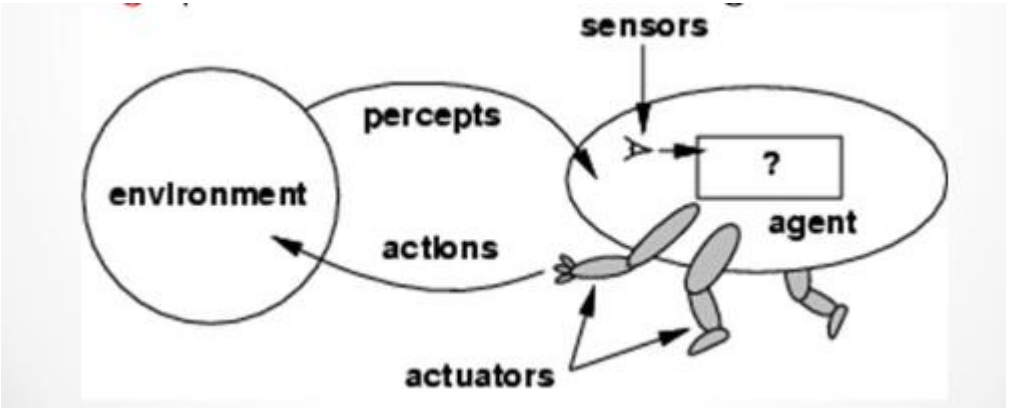
What is AI?

- Artificial Intelligence (AI) is a field of computer science dedicated to develop systems capable of performing tasks that would typically require human intelligence.
- These tasks include **learning, reasoning, problem-solving, perception, understanding natural language**, and even interacting with the environment.

According to Russell and Norvig, AI can be defined as follows:

" *AI (Artificial Intelligence) is the study of agents that perceive their environment, reason about it, and take actions to achieve goals.*"

Agent

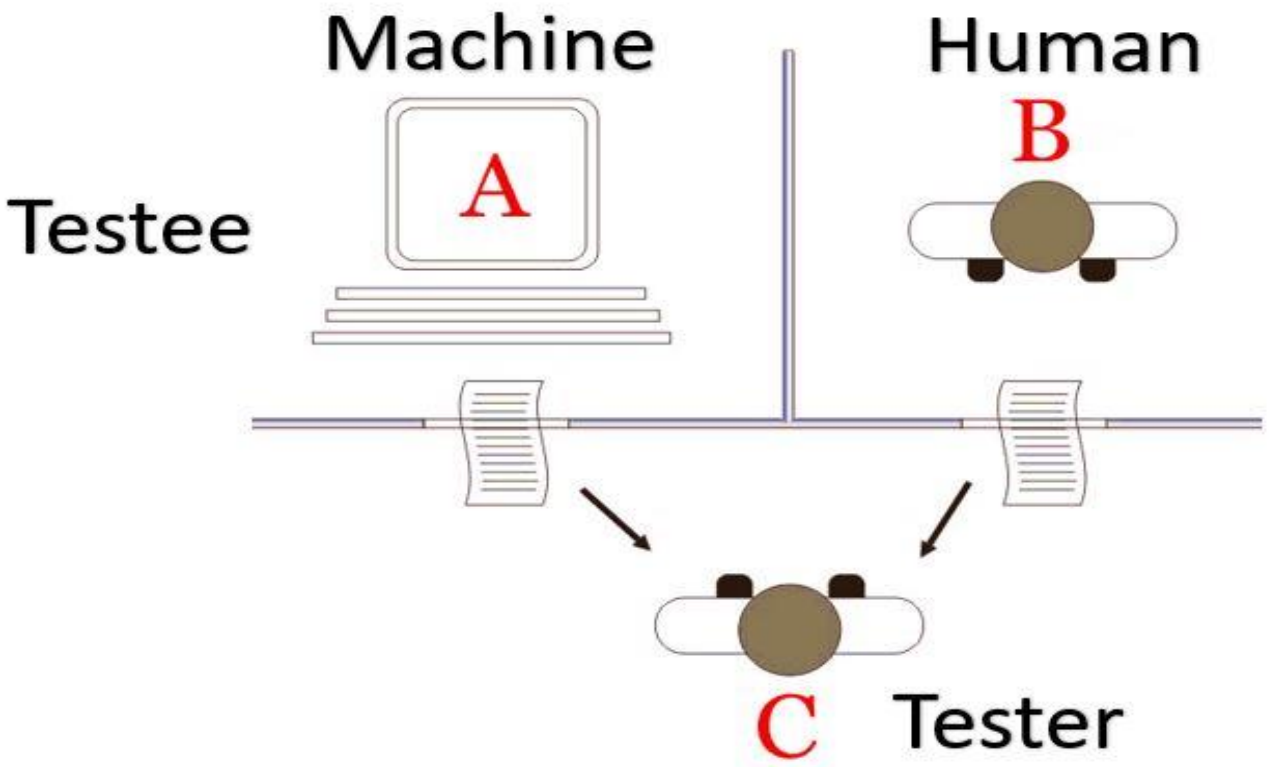


- In AI, an agent is any entity that can perceive its environment and take actions to achieve its goals.
- These agents can be physical robots, software programs, or any system capable of interacting with the world.

Eight explanations of AI shown in two groups

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Turing Test



The Foundations of Artificial Intelligence

1. Philosophy: In the philosophical exploration of AI, following questions arises:

- Can **formal rules** be used to draw valid conclusions?
- How does the **mind arise** from a physical brain?
- Where does **knowledge come** from?
- How does knowledge lead to **action?**

2. Mathematics: In the mathematical exploration of AI, following questions arises:

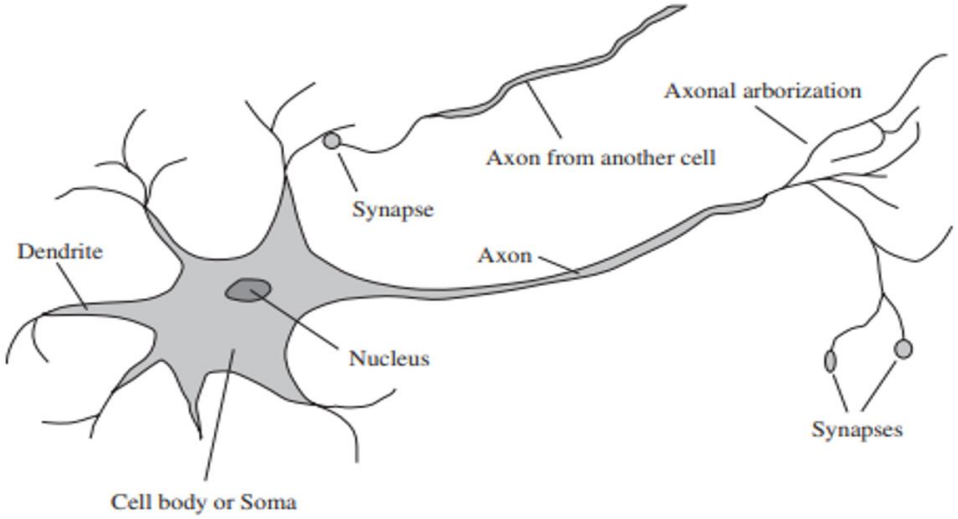
- What are the formal rules to draw valid conclusions?
- What can be computed?
- How do we reason with uncertain information?

3.Economics: In the Economics of AI, various attempts have been done to address the following questions:

- How should we make decisions so as to maximize payoff?
- How should we do this when others may not go along?
- How should we do this when the payoff may be far in the future

The Foundations of Artificial Intelligence

4.Neuroscience (How do brains process information?)



	Supercomputer	Personal Computer	Human Brain
Computational units	10^4 CPUs, 10^{12} transistors	4 CPUs, 10^9 transistors	10^{11} neurons
Storage units	10^{14} bits RAM 10^{15} bits disk	10^{11} bits RAM 10^{13} bits disk	10^{11} neurons 10^{14} synapses
Cycle time	10^{-9} sec	10^{-9} sec	10^{-3} sec
Operations/sec	10^{15}	10^{10}	10^{17}
Memory updates/sec	10^{14}	10^{10}	10^{14}

5. Psychology: How do humans and animals think and act?

6.Computer Engineering (How can we build an efficient computer?)

For artificial intelligence (AI) to succeed, two key elements are essential: **intelligence and an artifact**, with the **computer** being the chosen artifact.

7. Control Theory and Cybernetics (How can artifacts operate under their own control?)

8. Linguistics (How does language relate to thought?)

[Source Book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015]

History of AI

1.3.1 The Gestation of Artificial Intelligence (1943–1955)

1943: Warren McCulloch and Walter Pitts develop the first mathematical model of a neural network.

1949: Hebbian learning, formulated by Donald Hebb, becomes a lasting influence on neural network development.

1950: Alan Turing introduces the Turing Test and key AI principles.

1951: UNIVAC I, the first commercially produced computer, is used for statistical analysis, laying the groundwork for data processing.

1951: McCarthy earns his PhD and later plays a pivotal role in establishing AI at Dartmouth College.

1.3.2 The Birth of Artificial Intelligence (1956)

1956: John McCarthy organizes a landmark AI workshop at Dartmouth College, marking the official birth of artificial intelligence.

1956: The term "artificial intelligence" is coined at the Dartmouth Conference.

History of AI

1.3.3 Early Enthusiasm, Great Expectations (1952–1969)

Late 1950s: IBM produces early AI programs challenging predefined task limitations.

1958: McCarthy defines the Lisp language and introduces time-sharing.

1963: Marvin Minsky establishes Stanford's AI lab, emphasizing practical functionality.

1965: Joseph Weizenbaum creates ELIZA, an early natural language processing program.

1966–1973: Setbacks occur as early successes fail to scale, leading to reduced support for AI research.

1969: The Stanford Research Institute develops Shakey, the first mobile robot with reasoning abilities.

1.3.4 A Dose of Reality (1966–1973)

1969: Despite setbacks, the discovery of back-propagation learning algorithms for neural networks leads to a resurgence of interest.

1970s: Initial enthusiasm for AI fades, leading to the first "AI winter" as progress stalls

History of AI

1.3.5 Knowledge-Based Systems: The Key to Power? (1969–1979)

1969: DENDRAL exemplifies a shift towards domain-specific knowledge.

Late 1970s: The Heuristic Programming Project explores expert systems, emphasizing domain-specific knowledge.

1979: Marvin Minsky and Seymour Papert publish "Perceptrons," a book critical of certain AI approaches.

1.3.6 AI Becomes an Industry (1980–Present)

Early 1980s: The first successful commercial expert system, R1, is implemented at Digital Equipment Corporation.

1981: Japan's "Fifth Generation" project and the U.S.'s Microelectronics and Computer Technology Corporation respond to AI's growing influence.

1980s: Rapid growth followed by the "AI Winter," a period of decline in the AI industry.

1985: Expert systems, software that emulates decision-making of a human expert, gain popularity.

History of AI

1.3.7 The Return of Neural Networks (1986–Present)

Mid-1980s: Rediscovery of the back-propagation learning algorithm leads to the emergence of connectionist models.

Late 1980s: The AI industry experiences a decline known as the AI Winter.

1.3.8 AI Adopts the Scientific Method (1987–Present)

Late 1980s: AI shifts towards a more scientific and application-focused approach, experiencing a revival in the late 1990s.

1990-2005: Neural Networks Resurgence and Practical Applications

1997: IBM's Deep Blue defeats chess champion Garry Kasparov.

1999: Rodney Brooks introduces the concept of "embodied intelligence" with Cog, a humanoid robot.

History of AI

1.3.9 The Emergence of Intelligent Agents (1995–Present)

Late 1980s: The SOAR architecture addresses the "whole agent" problem.

Late 1990s–2000s: AI technologies underlie Internet tools, contributing to search engines, recommender systems, and website aggregators.

1.3.10 The Availability of Very Large Data Sets (2001–Present)

Late 1990s: A revival in AI with a shift towards a more scientific approach.

2000s: Emphasis on the importance of large datasets in AI research, leading to significant advancements.

2001: The DARPA Grand Challenge initiates research in autonomous vehicles.

2005: Stanford's Stanley wins the DARPA Grand Challenge, showcasing advances in self-driving technology.

2006: Geoffrey Hinton and colleagues publish a paper on deep learning, reigniting interest in neural networks.

2011: IBM's Watson wins Jeopardy!, demonstrating the power of natural language processing.

2012: AlexNet, a deep convolutional neural network, achieves a breakthrough in image recognition at the ImageNet competition.

2012-Present: AI in the Mainstream and Ethical Concerns

2014: Facebook's AI lab introduces DeepFace for facial recognition, reaching human-level accuracy.

2016: AlphaGo, an AI developed by DeepMind, defeats world champion Go player Lee Sedol.

2018: OpenAI releases GPT-2, a large-scale language model.

2020s: AI applications become integral in various industries, raising concerns about ethics, bias, and job displacement.

Topics

- What is AI?
- Foundations of AI
- History of AI,
- Agents, The structure of agents.
- Problem Solving Agents
- Example problems,
- Searching for Solutions,
- Uninformed Search Strategies: Breadth First search, Depth First Search

Agent

An agent is defined as anything capable of perceiving its environment through sensors and acting upon that environment through actuators. This basic concept is depicted in Figure 2.1.

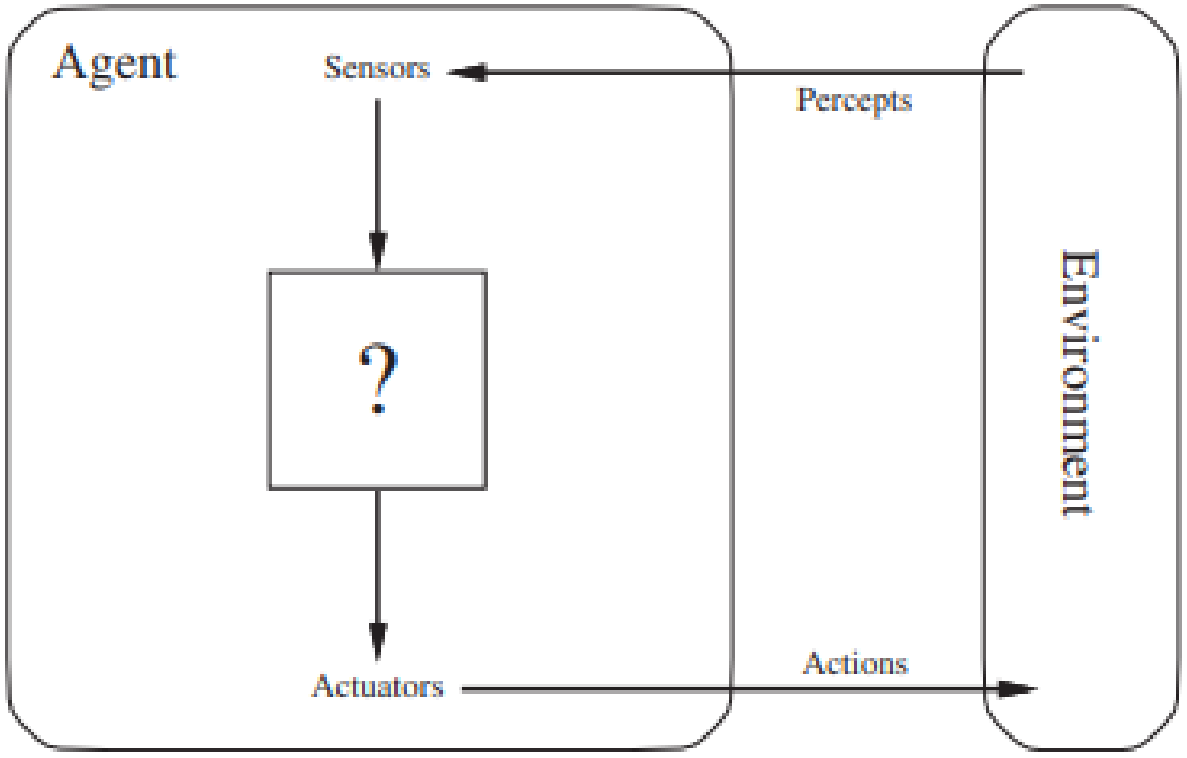


Figure 2.1 Agents interact with environments through sensors and actuators.

Agent	Sensors	Actuators
Human	Eyes, Ears and other Sensory Organs	Hands, Legs, Vocal Tract
Robot	Cameras, Infrared Range Finders	Various Motors
Software	Keystrokes, File Contents, Network Packets	Displaying on the screen, Writing Files, Sending Network Packets

Word	Meaning
Percept	The term **"percept"** refers to an agent's perceptual inputs at any given moment.
Percept Sequence	Percept sequence encompasses the complete history of everything the agent has perceived
Table	A table serves as an external representation of an agent's behaviour, specifically presented in tabular form that outlines the agent's actions for every possible percept sequences.
Agent Program	An agent program represents the internal implementation of an agent's behaviour.

Vacuum Cleaner World

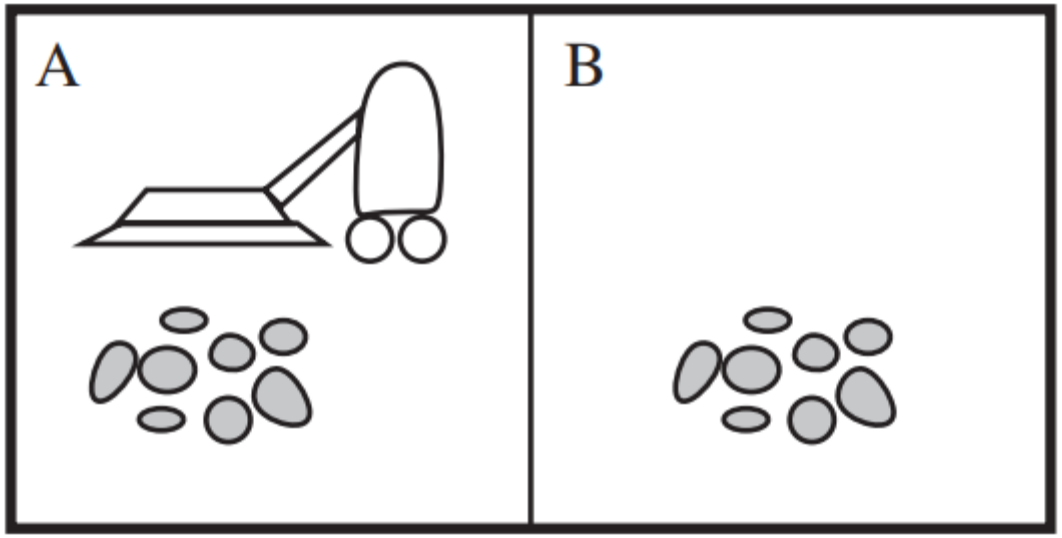


Table [Agent function]

- The **agent function**, considers the entire percept history.
- Agent Function is represented using Table.

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

Agent Program

- The **agent program**, operates based on the current percept,
- The **agent programs** are presented in a simple pseudocode language

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Structure of Intelligent Agents

Agent = Architecture + Agent Program

- **Architecture = the machinery that an agent executes on.**
- **Agent Program = an implementation of an agent function.**

Types of Agents

Agent Type	Description
Simple Reflex Agents	Select actions based on the current percept, without considering the history of past percepts.
Model-Based Reflex Agents	Maintain an internal model of the world, considering the history of percepts for decision-making.
Goal-Based Agents	Designed to achieve specific objectives, using internal goals to determine actions.
Utility-Based Agents	Evaluate actions based on a utility function, quantifying the desirability of different outcomes.
Learning Agents	Improve performance over time through learning from experience.

Problem Solving Agents

Problem Solving Agent is a type of goal based intelligent agent in artificial intelligence that is designed to **analyse a situation, identify problems or goals, and then take actions to achieve those goals**.

- In the city of **Arad,** Romania, an agent on a touring holiday has a performance measure with various goals, **such as improving suntan, language skills, exploring sights, and avoiding hangovers.**
- The decision problem is complex if no goal is fixed.

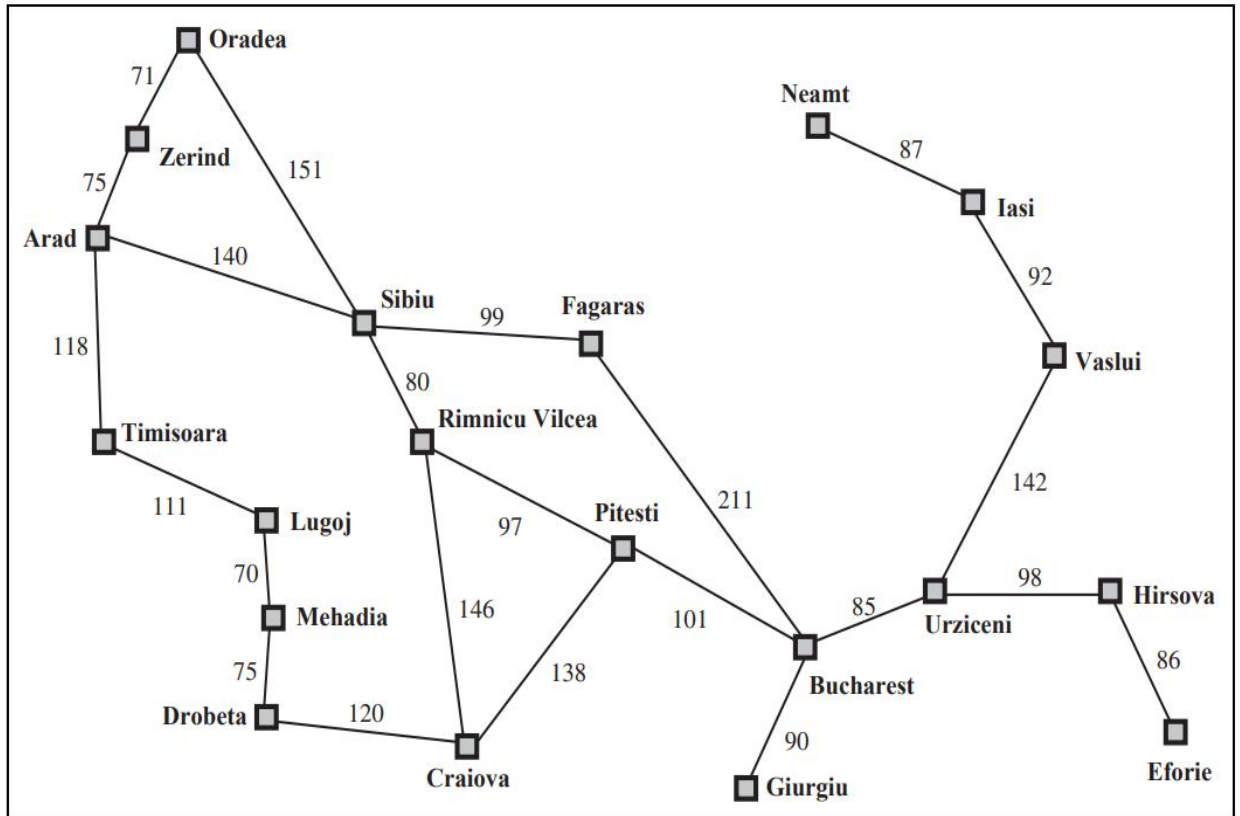


Figure 3.2 A simplified road map of part of Romania.

Steps followed by Problem Solving Agents

1. **Goal Formulation**
2. **Problem Formulation**
3. **Search Solution**
4. **Execution**
5. **Learning (Optional)**
6. **Feedback and Iteration**

"Formulate, Search, Execute" framework for the 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.

Well Defined Problems and Solutions

A problem can be defined formally by five components:

1. **State Representation**: Encompasses the initial state from which the agent begins its problem-solving journey, represented, for example, as "*In(Arad)*."
2. **Actions and Applicability**: Describes the set of possible actions available to the agent in a given state, denoted as ACTIONS(s). For instance, in the state *In(Arad)*, applicable actions include {*Go(Sibiu)*, *Go(Timisoara)*, *Go(Zerind)*}.
3. **Transition Model**: Specifies the consequences of actions through the transition model, represented by the function RESULT(s,a), which yields the state resulting from performing action a in state s. For example, RESULT(*In(Arad),Go(Zerind))=In(Zerind)*.
4. **Goal Specification and Test**: Defines the goal state or states and includes a test to determine whether a given state satisfies the goal conditions. In the example, the goal is represented as the singleton set *{In(Bucharest)}*.
5. **Cost Functions**: Encompasses both the path cost function, assigning a numeric cost to each path, and the step cost, denoted as $c(s,a,s')$, which represents the cost of taking action **a** in state **s** to reach state s'. The cost functions play a crucial role in evaluating and optimizing the performance of the agent's solution.

Example Problems: Toy and Real-world problems.

A toy problem is designed to showcase or test various problem-solving techniques, featuring a precise and concise description. This allows different researchers to use it for comparing algorithm performances.

On the other hand, **a real-world problem** is one that holds significance for people, lacking a single universally agreed-upon description. However, we can provide a general sense of their formulations.

Toy Problems

1. Vacuum Cleaner World
2. 8 Puzzle
3. 8 Queens
4. Math's Sequences

Vacuum World

The problem can be formalized as follows:

1. **States:** The state is defined by the agent's location and the presence of dirt in specific locations. The agent can be in one of two locations, each potentially containing dirt. Consequently, there are 8 possible world states (2×2^2). For a larger environment with n locations, there would be $n \cdot 2^n$ states.
2. **Initial state**: Any state can serve as the initial state.
3. **Actions:** In this uncomplicated environment, each state presents three actions: **Left, Right, and Suck**. More extensive environments might also include **Up** and **Down**.
4. **Transition model**: Actions produce expected effects, except for instances where moving **Left** in the leftmost square, moving **Right** in the rightmost square, and **Sucking in a clean square** result in no effect.
5. **Goal test**: This assesses whether **all squares are clean**.
6. **Path cost**: Each step incurs a **cost of 1**, making the path cost equivalent to the number of steps taken in the path.

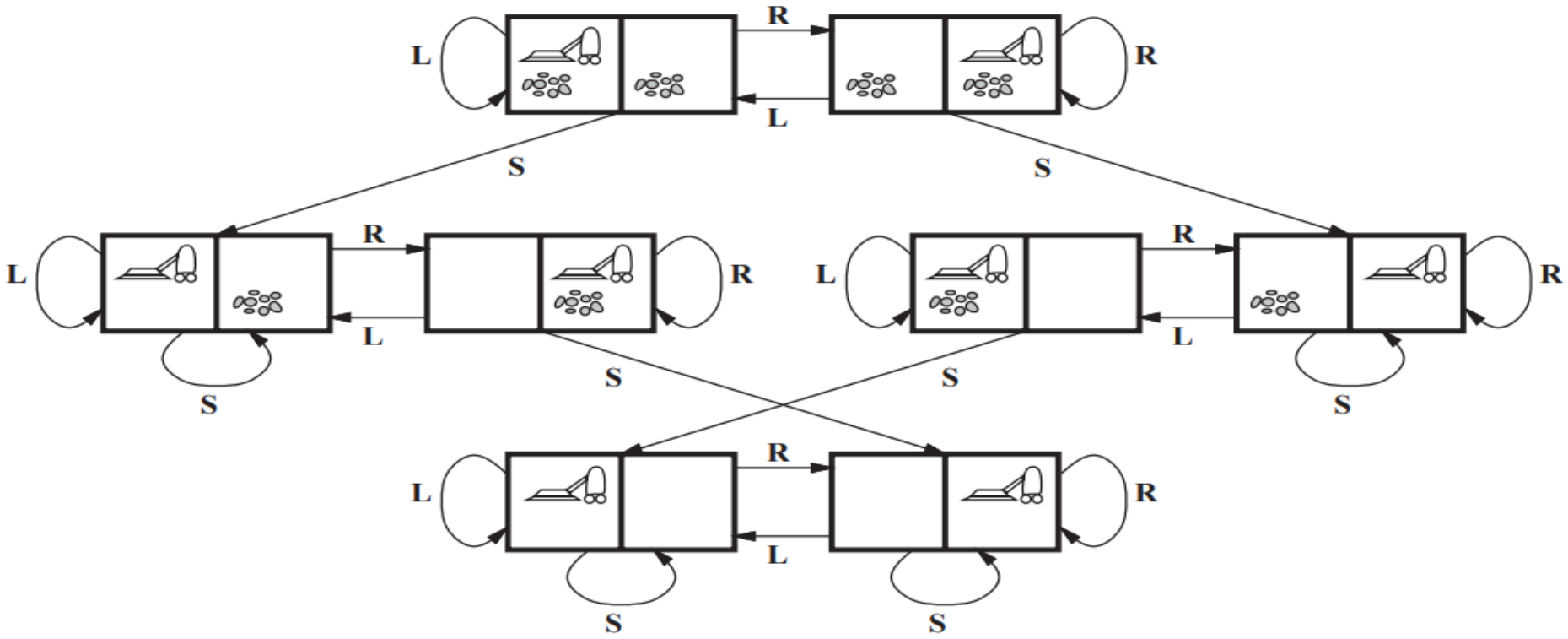


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

8 Puzzle

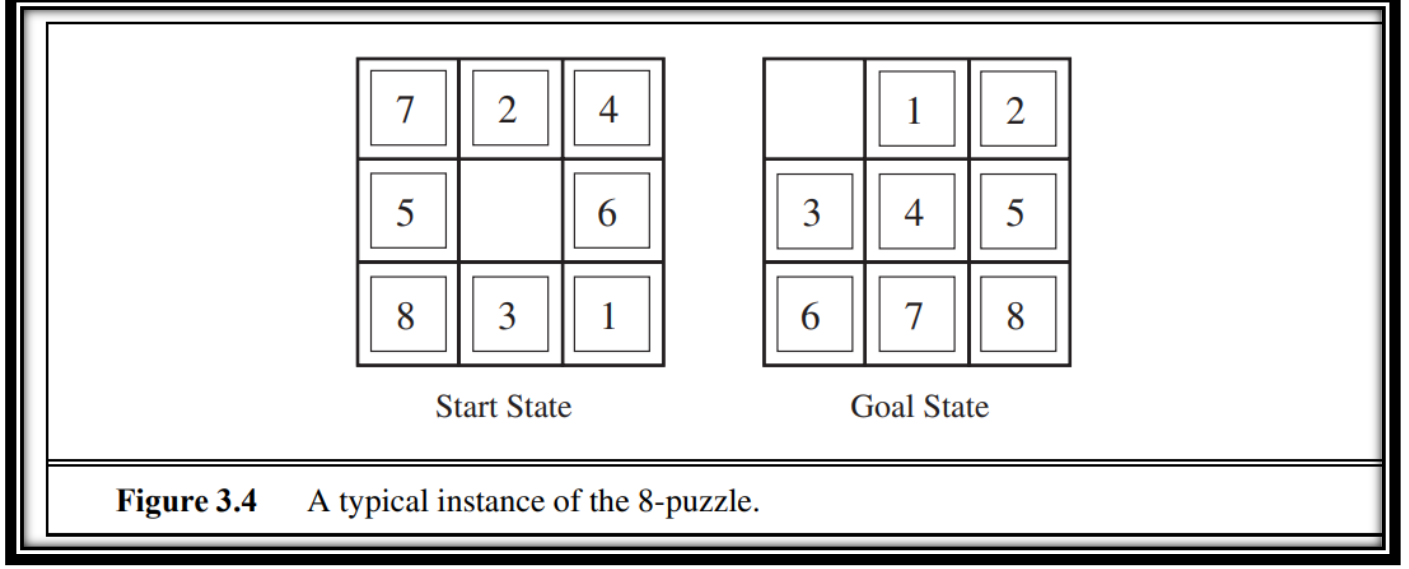
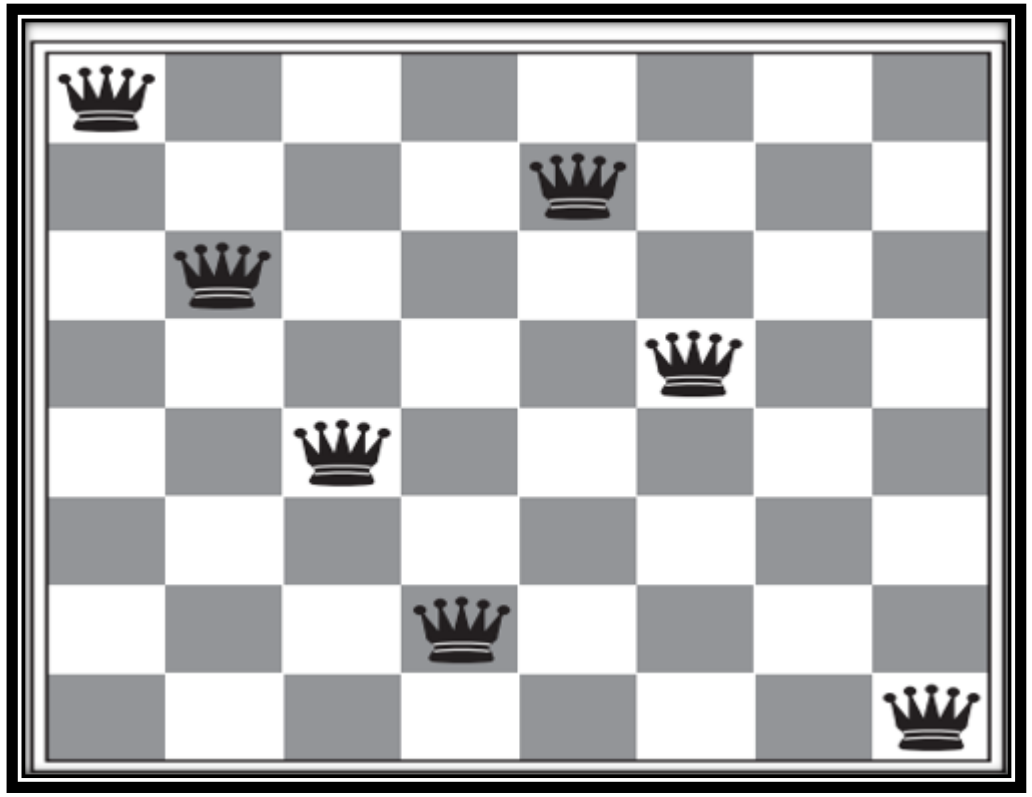


Figure 3.4 A typical instance of the 8-puzzle.

1. **States:** A state description indicates the position of each of the eight tiles and the empty space within the nine squares.
2. **Initial state:** Any state can be designated as the initial state.
3. **Actions:** In its simplest form, actions are defined as movements of the empty space—Left, Right, Up, or Down. Different subsets of these actions are possible based on the current location of the empty space.
4. **Transition model**: Given a state and an action, the model returns the resulting state. For instance, applying Left to the starting state in Figure 3.4 would switch the positions of the 5 and the empty space.
5. **Goal test:** This checks if the state aligns with the specified goal configuration shown in Figure 3.4. Other goal configurations are also conceivable.
6. **Path cost:** Each step incurs a cost of 1, making the path cost equivalent to the number of steps taken in the path.

The 8-queens problem



- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.
- (A queen attacks any piece in the same row, column or diagonal.)
- Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

- **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.

Math's Sequences

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} \right\rfloor = 5 \ .$$

The problem definition is very simple:

1. **States**: Positive numbers.
2. **Initial state**: 4.
3. **Actions**: Apply factorial, square root, or floor operation (factorial for integers only).
4. **Transition model**: As given by the mathematical definitions of the operations.
5. **Goal test**: State is the desired positive integer

Topics

- What is AI?
- Foundations of AI
- History of AI
- Agents, The structure of agents.
- Problem Solving Agents
- Example problems,
- Searching for Solutions,
- Uninformed Search Strategies: Breadth First search, Depth First Search

Real-world problems

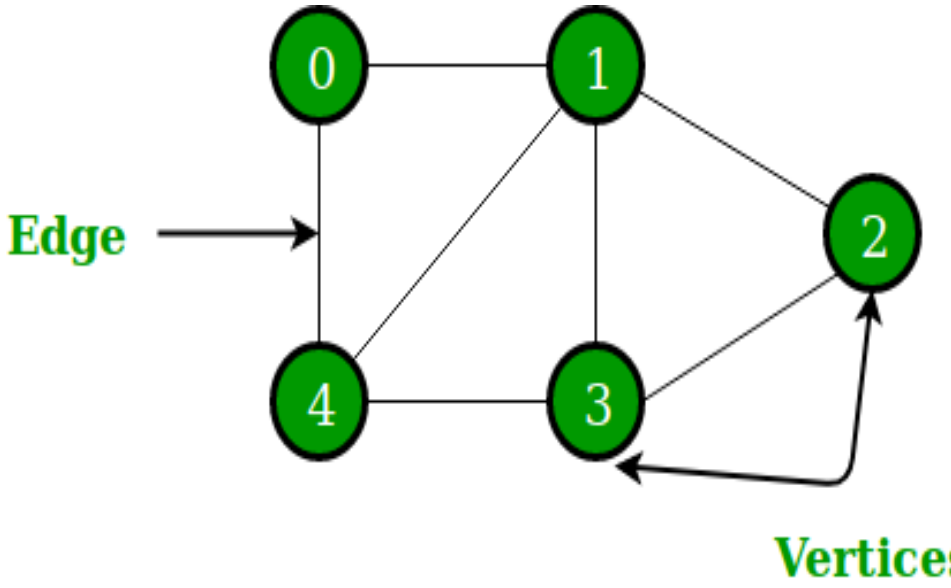
1. Route Finding Problem
2. Touring Problem
3. Traveling Salesperson Problem
4. VLSI Layout
5. Robot Navigation

Consider the airline travel problems that must be solved by a travel-planning Web site:

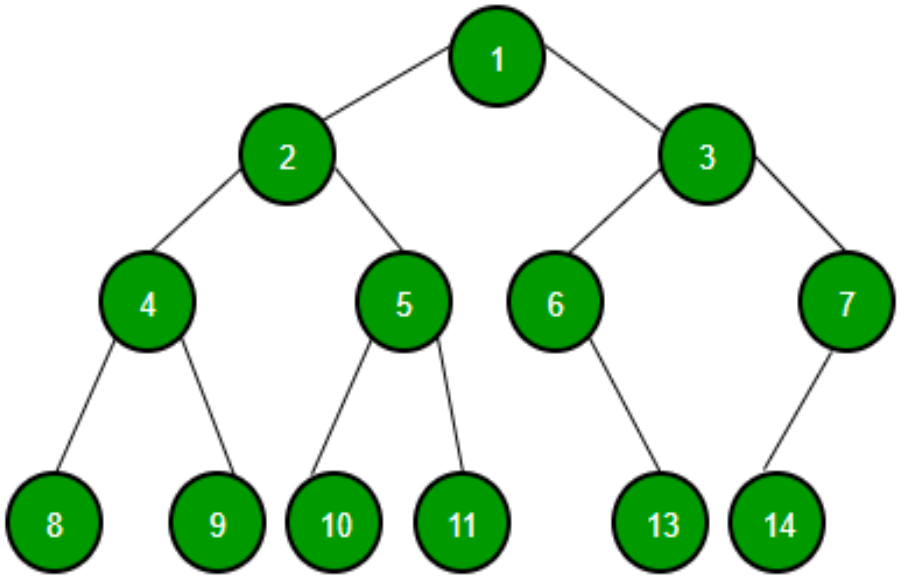
- **States:** Each state obviously includes a **location** (e.g., an airport) ,**current time**, **domestic or international, "historical" aspects.**
- **Initial state**: This is specified by the user's query.
- **Actions**: ***Take any flight from the current location***, in ***any seat class***, ***leaving after the current time***, ***leaving enough time for within-airport transfer if needed***.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test**: Are we at the final destination specified by the user?
- **Path cost**: This depends on *monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on*.

Parameter	Graph	Tree
Description	Graph is a non−linear data structure that can have more than one path between vertices.	Tree is also a non−linear data structure, but it has only one path between two vertices.
Loops	Graphs can have loops.	Loops are not allowed in a tree structure.
Root Node	Graphs do not have a root node.	Trees have exactly one root node.
Traversal Techniques	Graphs have two traversal techniques namely, breadth−first search and depth−first search.	Trees have three traversal techniques namely, pre−order, in−order, and post−order.

Graph



Trees



Note : The set of all leaf nodes available for expansion at any given point is called the **frontier**

Searching for Solutions

- The **SEARCH TREE** possible action sequences starting at the initial state form a search tree with the initial state NODE at the root; the branches are actions and the nodes correspond to states in the state space of the problem
- **Expanding** the current state applying each legal action to the current state, thereby **generating** a new set of state.

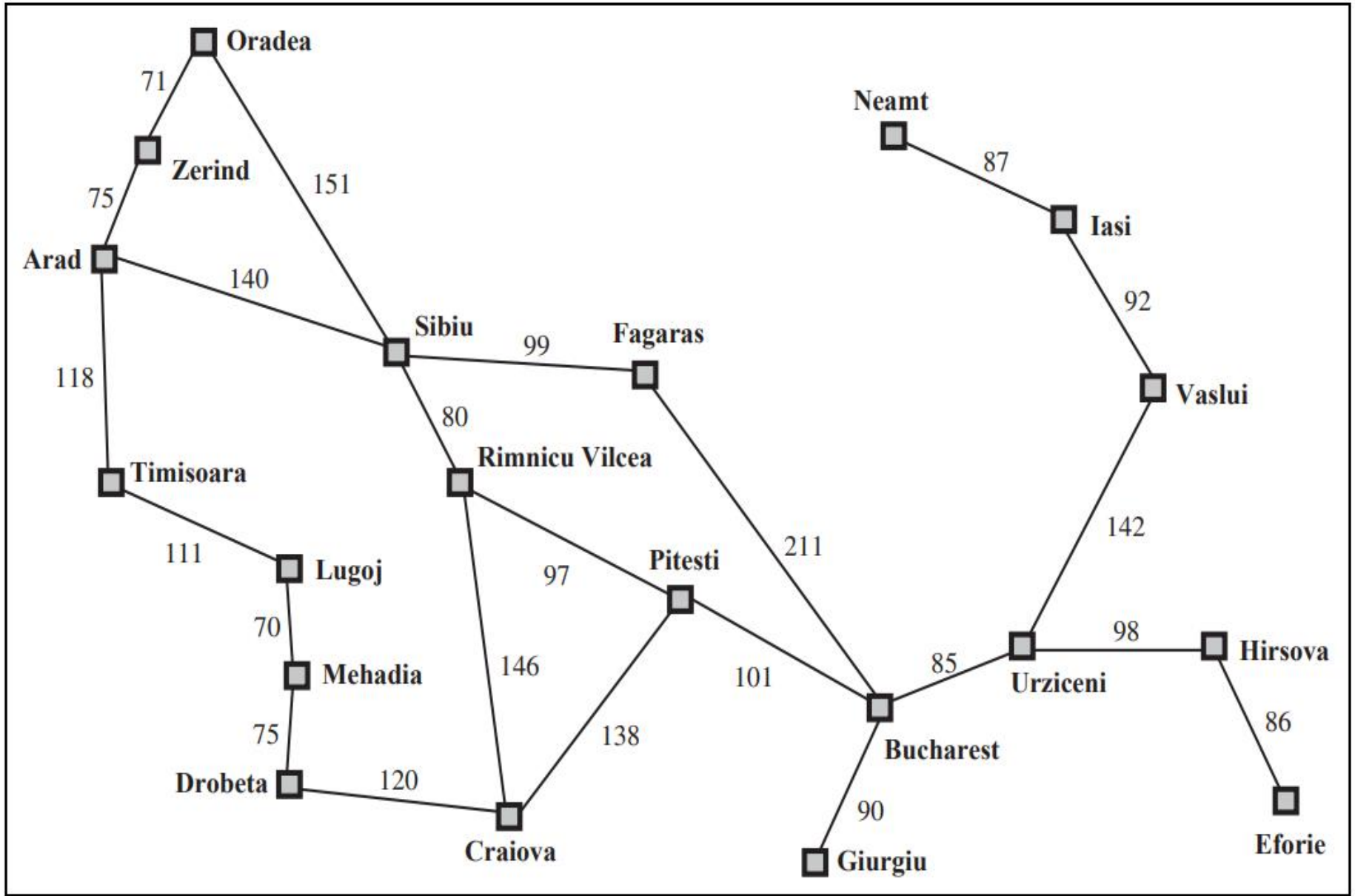
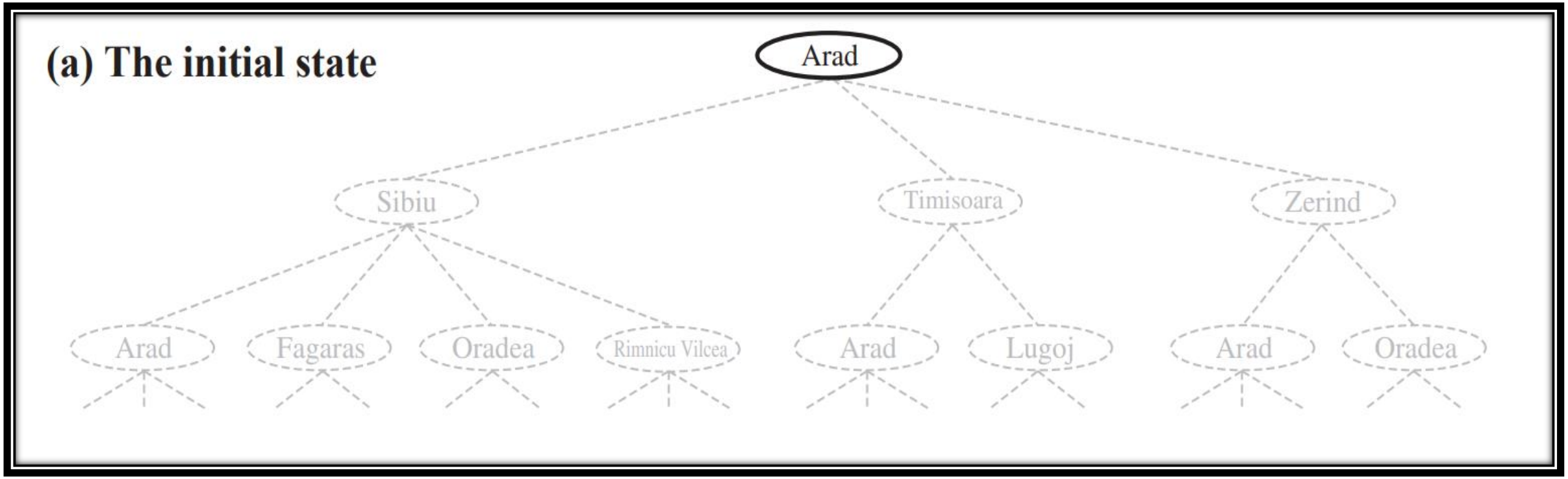
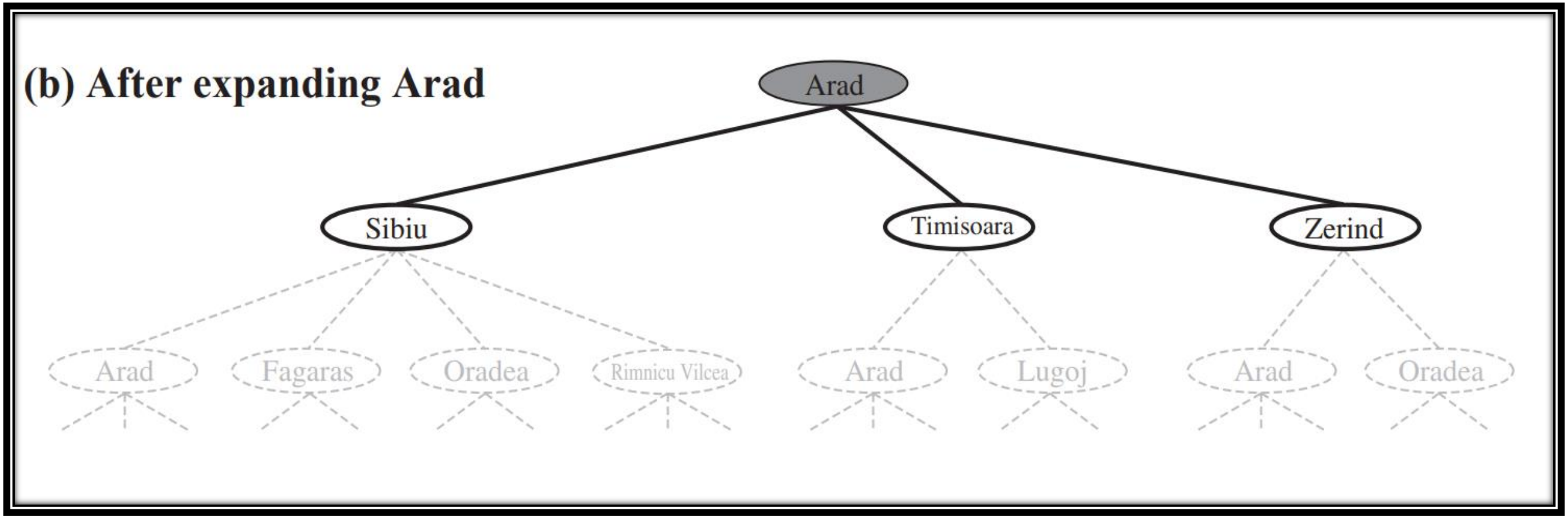


Figure 3.2 A simplified road map of part of Romania.

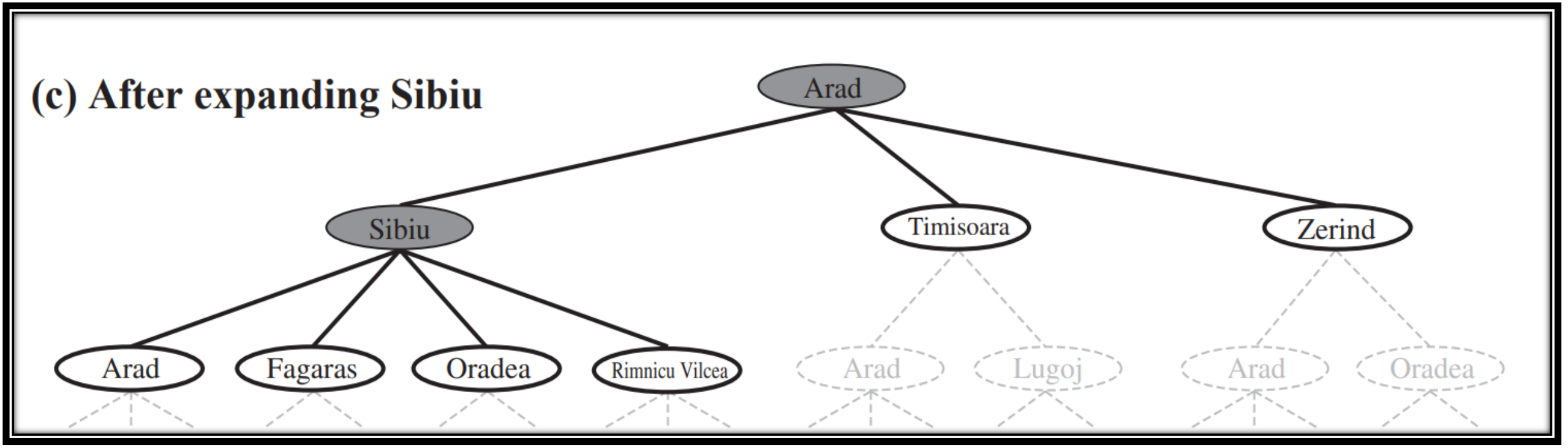
Partial search trees for finding a route from Arad to Bucharest



Partial search trees for finding a route from Arad to Bucharest



Partial search trees for finding a route from Arad to Bucharest



Tree Based Searching

```
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
```

Graph based Searching

```
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
```

Sequence of Search Trees

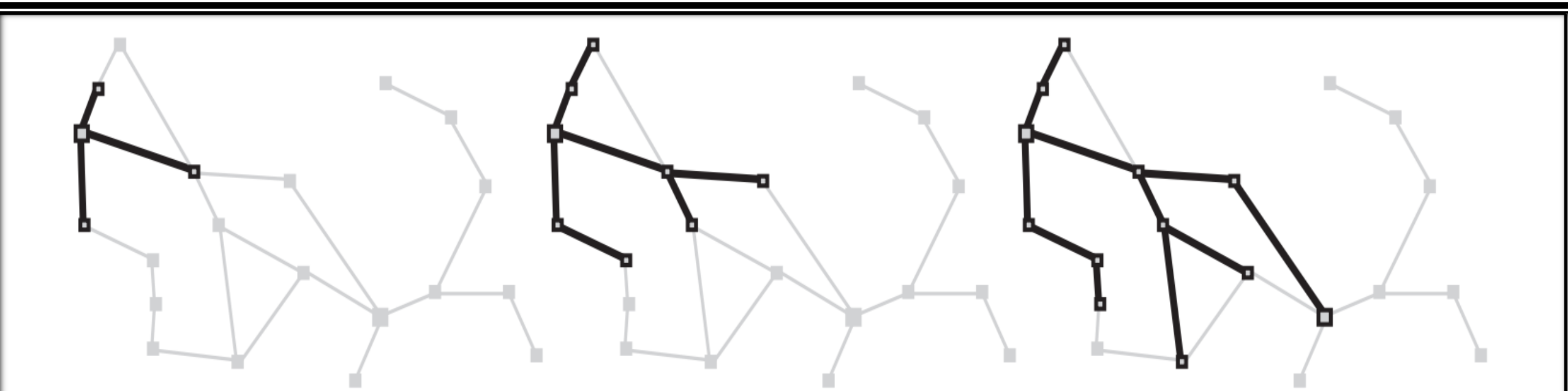


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Graph Search

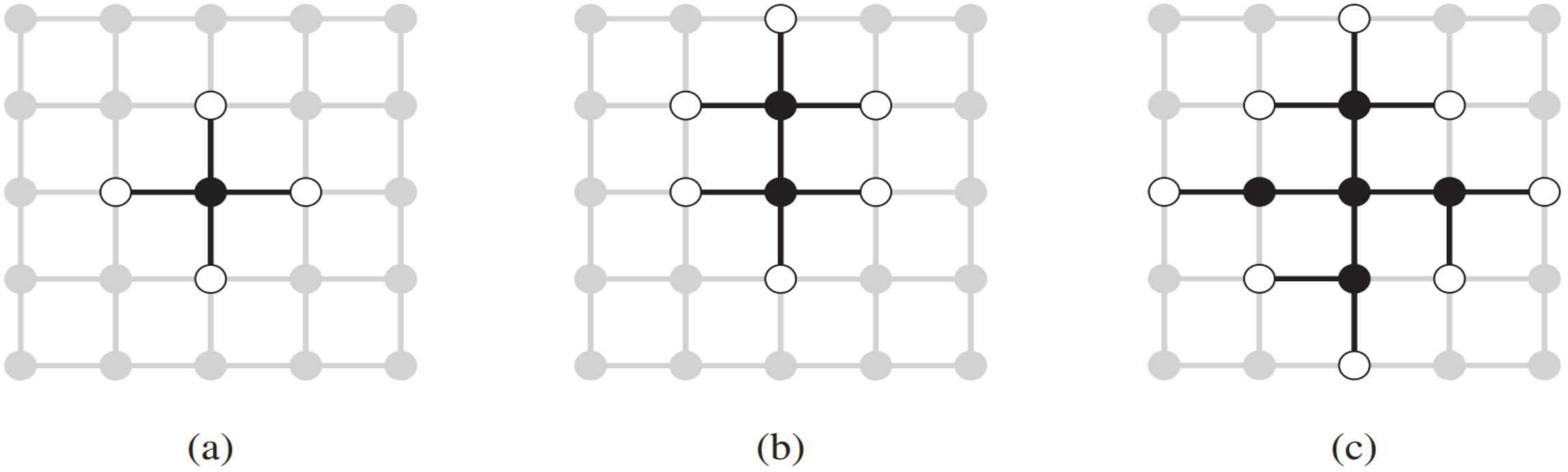


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

NODE

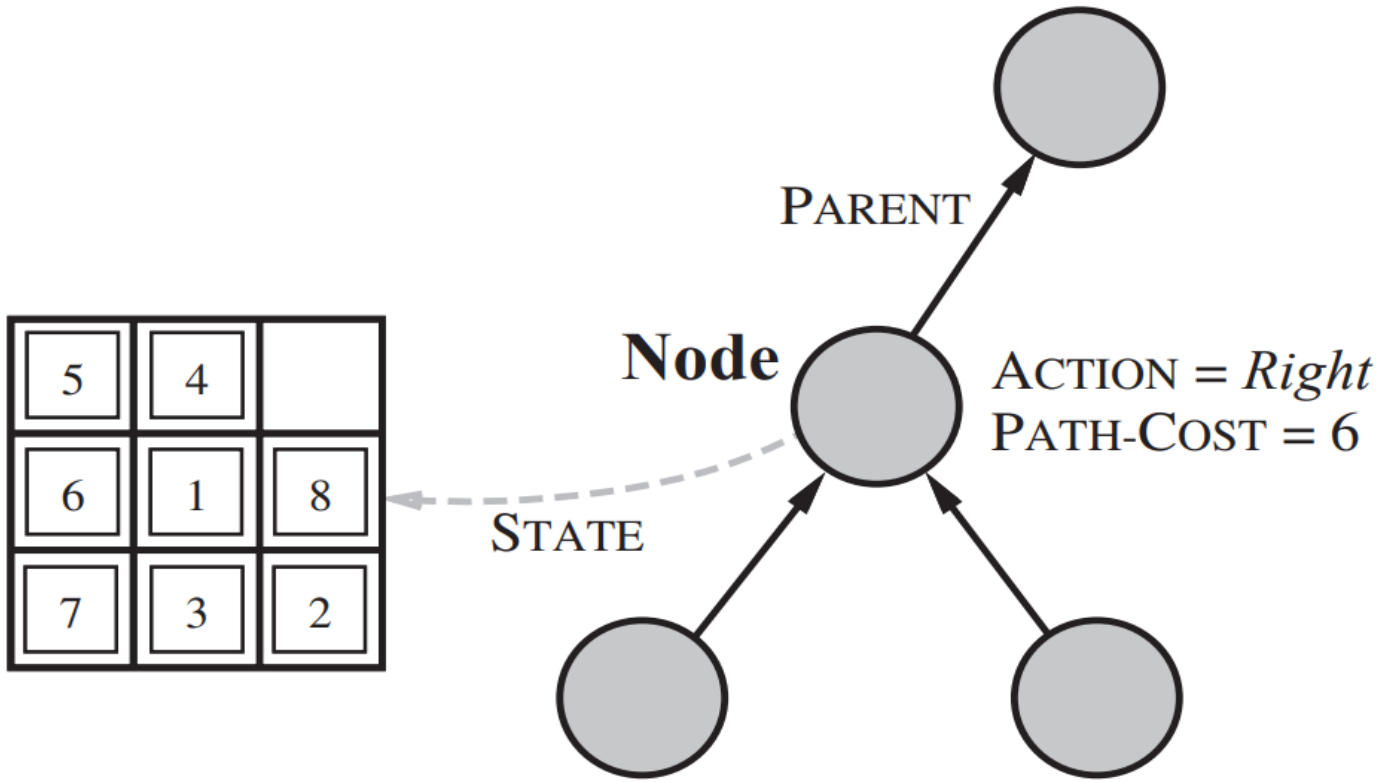
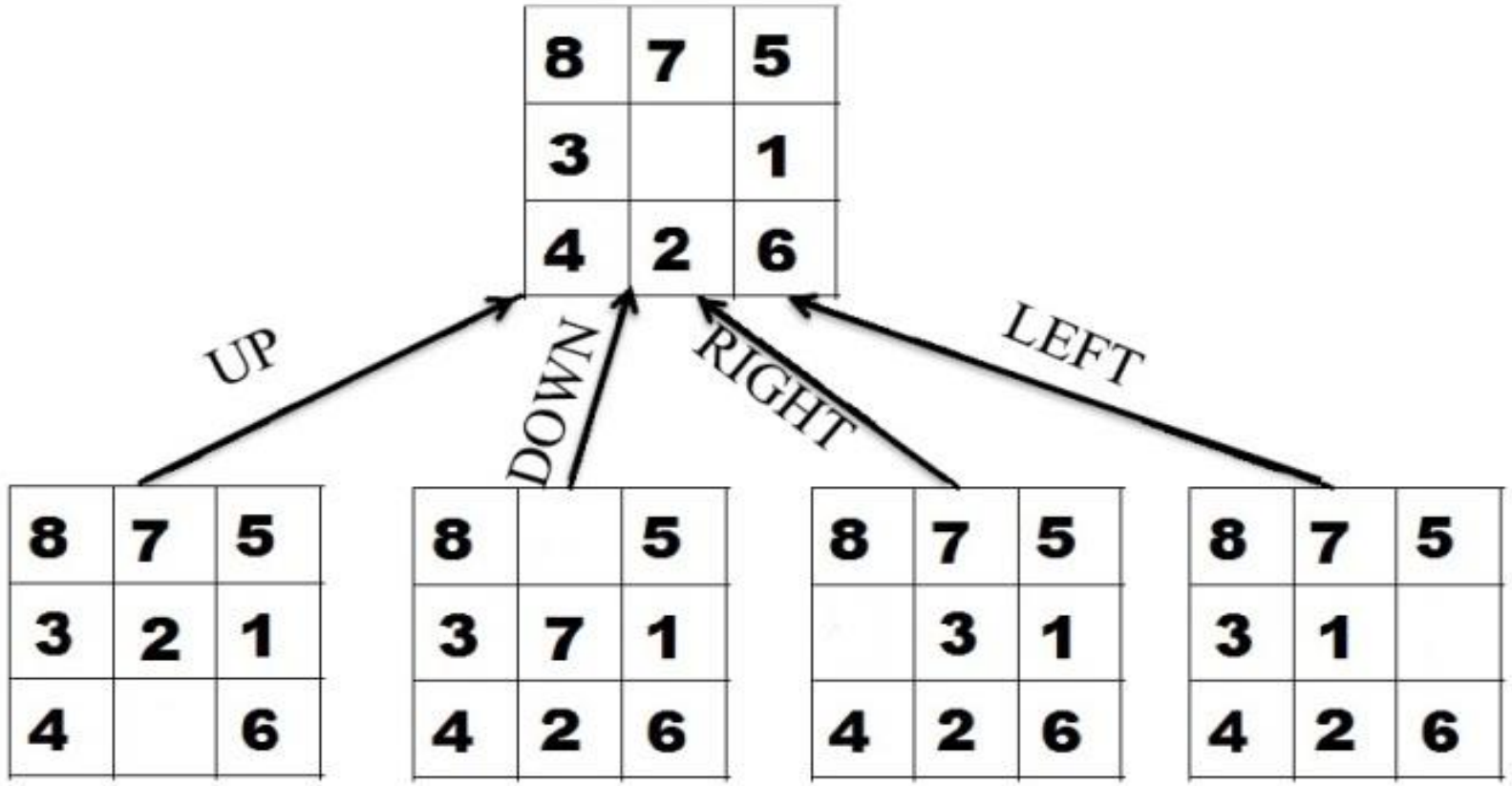


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.


8 7 5
3 1
4 2 6
UP
DOWN
RIGHT
LEFT
8 7 5
3 2 1
4 6
8 5
3 7 1
4 2 6
8 7 5
3 1
4 2 6
8 7 5
3 1
4 2 6

Function: “CHILD NODE”

```
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)
```

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?

Searching Strategies/Algorithms

Uninformed (Blind)Search Strategies

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening depth-first search
6. Bidirectional search

Informed (Heuristic) Search Strategies

1. Greedy best-first search
2. A* search: Minimizing the total estimated solution cost
3. Memory-bounded heuristic search
4. AO* Search
5. Problem Reduction
6. Hill Climbing

Difference between Uninformed and Informed Search Techniques

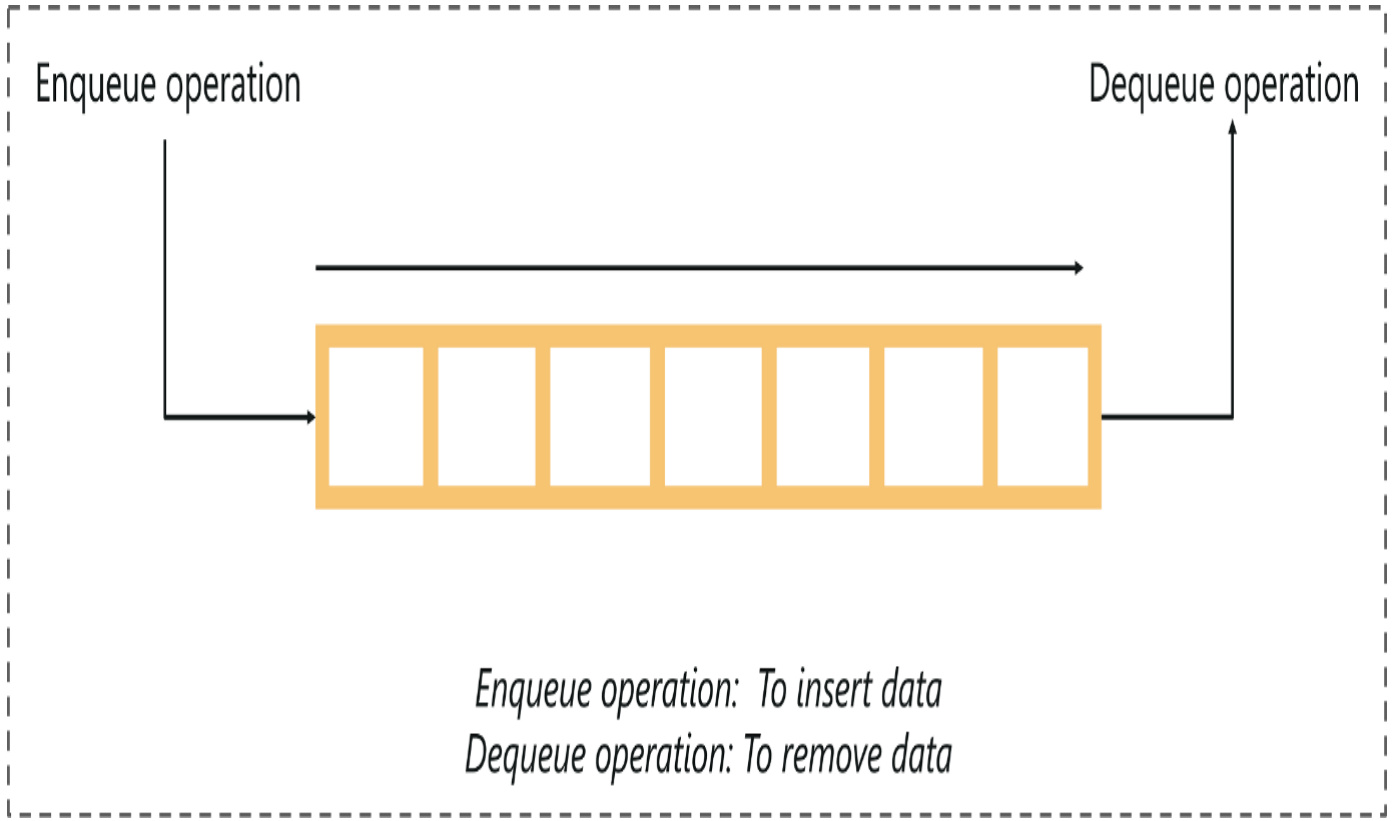
Characteristic	Uninformed Search	Informed Search
Information Utilization	No additional knowledge	Additional knowledge (heuristics)
Decision Making	Based solely on structure of the search space	Uses heuristics to intelligently guide the search
Completeness	Completeness depends on the specific algorithm	Completeness depends on the specific algorithm and heuristic
Optimality	May not guarantee the most optimal solution	A* is optimal under certain conditions
Efficiency	May be less efficient for certain problems due to exhaustive exploration	Generally more efficient due to heuristic guidance
Examples	BFS, DFS, Uniform Cost Search, etc.	A*, Greedy Best-First Search,etc.

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Enqueue and Dequeue Operations



Enqueue Operation: Adding an element to the end of the queue.
Example: If the queue is [A, B, C] and you enqueue the node D, the queue becomes [A, B, C, D].

Dequeue Operation: Removing an element from the front of the queue.
Example: If the queue is [A, B, C, D] and you dequeue, the result is A, and the queue becomes [B, C, D].

Breadth First Search

1. **Initialize a queue with the initial state (usually the root node).**
2. **While the queue is not empty:**
   a. **Dequeue a node from the front of the queue.**
   b. **If the node contains the goal state, return the solution.**
   c. **Otherwise, enqueue all the neighbouring nodes that have not been visited.**
3. **If the queue becomes empty and the goal state is not found, then there is no solution.**

Note : The set of all leaf nodes available for expansion at any given point is called the frontier

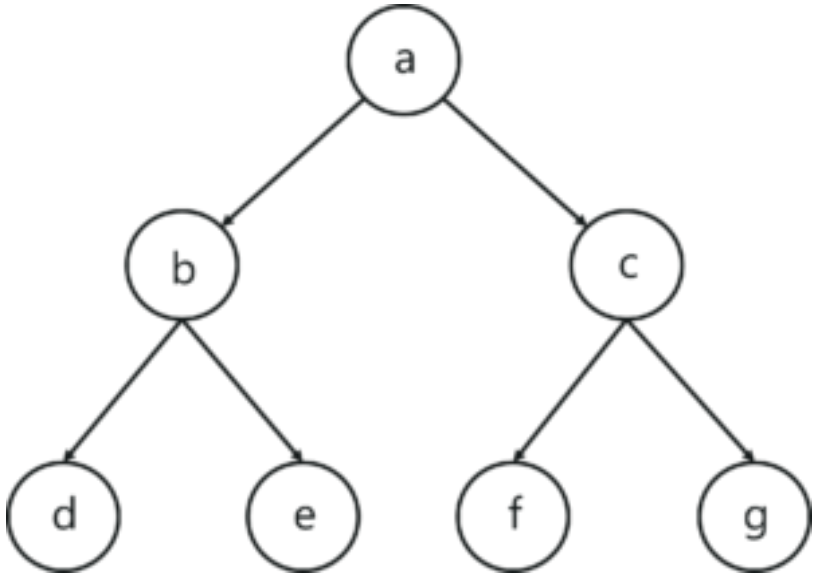
Pseudo Code

```
function BFS(initial_state, goal_state):
    initialize an empty queue
    enqueue initial_state to the queue

    while queue is not empty:
        current_node = dequeue from the front of the queue

        if current_node is the goal_state:
            return solution
        for each neighbor of current_node:
            if neighbor has not been visited:
                mark neighbor as visited
                enqueue neighbor to the queue

return no solution
```

Note:

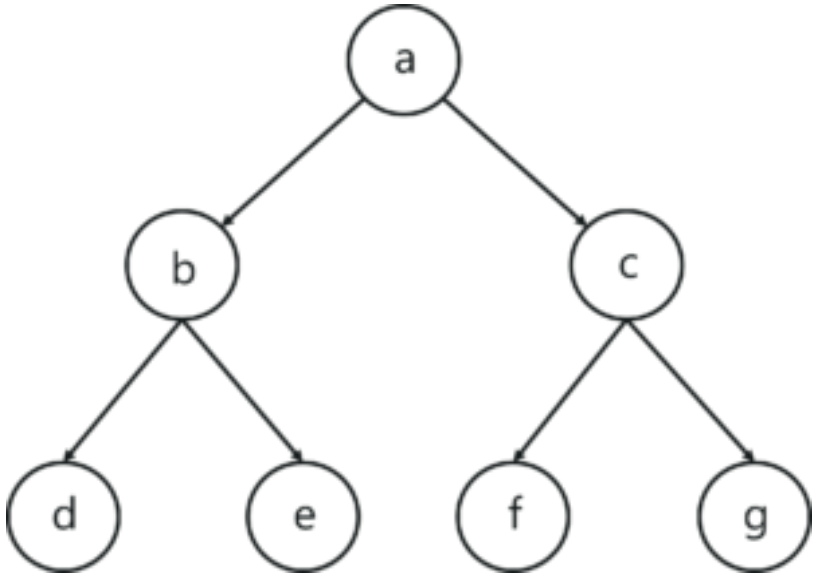
- The queue is a **First-In-First-Out (FIFO)** data structure, meaning that the first element enqueued will be the first to be dequeued.
- The algorithm ensures that all nodes at a particular depth level are explored before moving on to the nodes at the next level.
- BFS is complete and optimal for searching in a state space with a uniform cost per step.
- It may require a lot of memory for large state spaces due to the need to store all generated nodes.

Example



Step	Details	Visited Nodes	QUEUE
1	Initialization: Start with the root node A.	V = {}	a
2	Node **a,** Visited, Dequeue node **a** and enqueue neighbour nodes **b** and **c** to queue	V = {a}	.. b c b
3	Node **b** visited, deque node **b** and enqueue neighbour nodes **d** and **e** to queue.	V = {a,b}	c .. d c e d c
4	Node **c** visited, deque node **c** and enqueue neighbour nodes **f** and **g** to queue	V = {a,b,c}	. e d . f e d . g f e d

Example



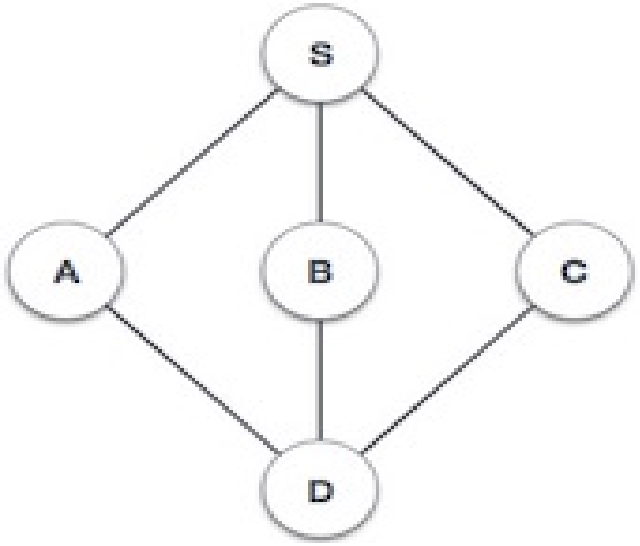
5	Node **d** visited, deque node **d**	V = {a,b,c,d}	g f e
6	Node **e** visited, deque node **e**	V = {a,b,c,d,e}	g f
7	Node **f** visited, deque node **f**	V = {a,b,c,d,e,f}	g
8	Node **g** visited, deque node **g**	V = {a,b,c,d,e,f,g}	

Pseudo Code BFS

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

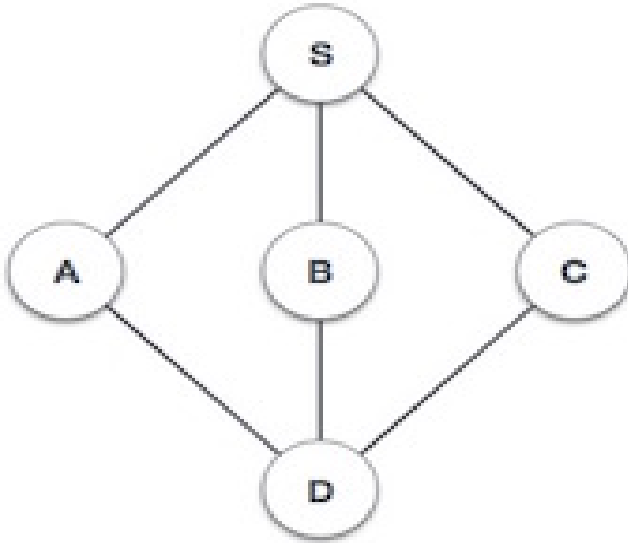
Figure 3.11 Breadth-first search on a graph.

Example: BFS on Simple Graph



Step	Description	Visited Nodes	Queue
1	Initialize the queue.	V= {}	S
2	Node **S** Visited, Dequeue node **S** and enqueue neighbour nodes **A, B and C** to queue.	V= {S}	A B A C B A
3	Node **A** Visited, Dequeue node **A** and enqueue neighbour node **D**	V= {S,A}	D C B

Example: BFS on Simple Graph



4	Node **B** Visited, Dequeue node **B**	V= {S,A,B}	D C
5	Node **C** Visited, Dequeue node **C**	V= {S,A,B,C}	D
6	Node **D** Visited, Dequeue node **D**	V= {S,A,B,C,D}	

Breadth First Search: S A B C D

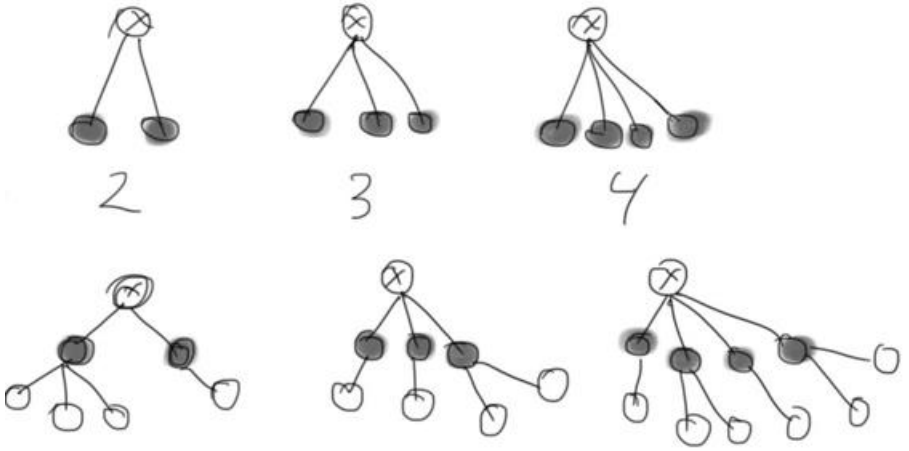
Time and Space Complexity of BFS

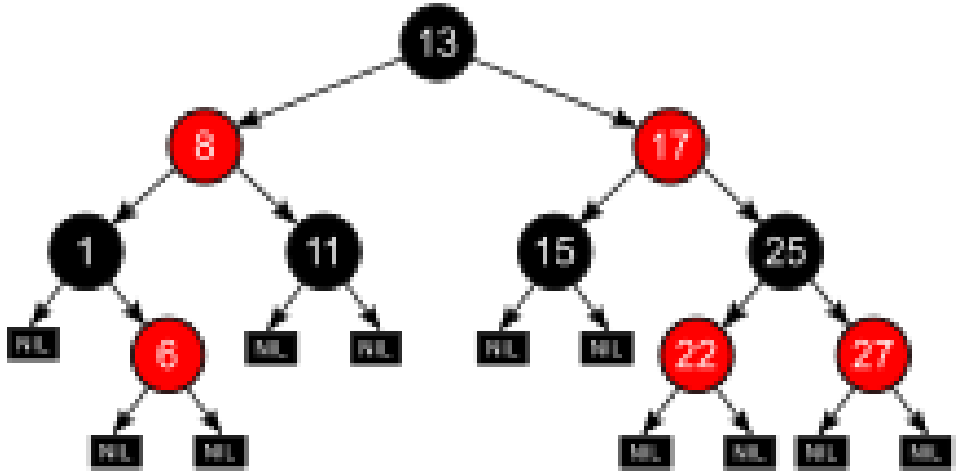
Time Complexity:

- The time complexity of BFS in a uniform tree is expressed as O(b^d), where 'b' is the branching factor and 'd' is the depth of the solution.
- Applying the goal test upon node expansion instead of generation would result in a higher time complexity of O(b^(d+1)).

Space Complexity:

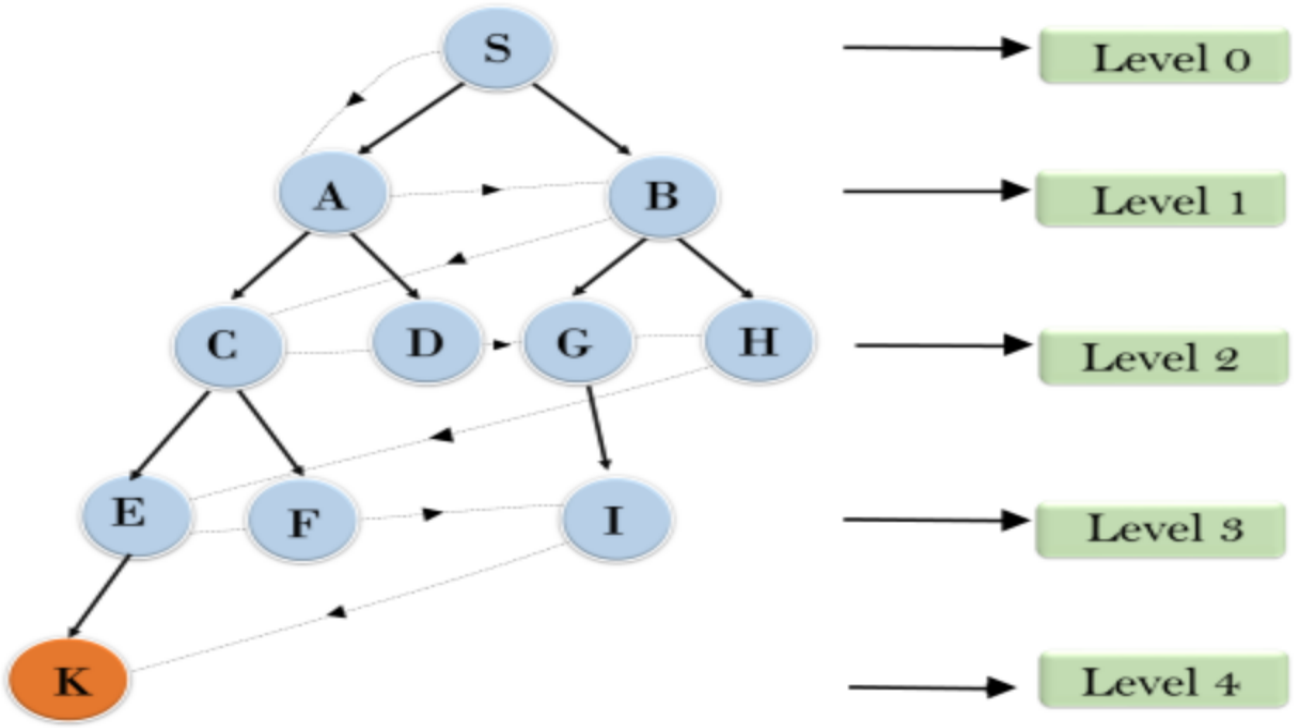
- For breadth-first graph search, where every generated node is kept in memory, the space complexity is O(b^d).
- The space complexity is dominated by the size of the frontier, which holds nodes yet to be explored.





- The **branching factor** is the number of children at each node, the outdegree. If this value is not uniform, an *average branching factor* can be calculated.
- The average branching factor can be quickly calculated as the number of non-root nodes (the size of the tree, minus one; or the number of edges) divided by the number of non-leaf nodes (the number of nodes with children)

Breadth First Search

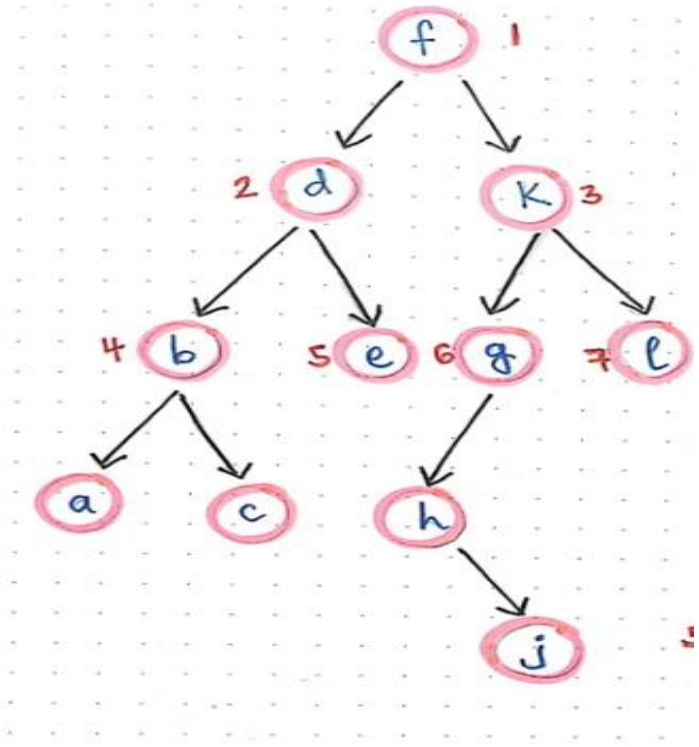


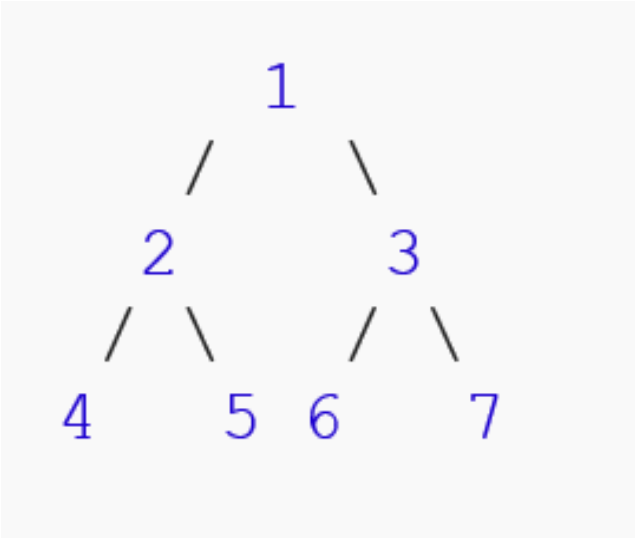
Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

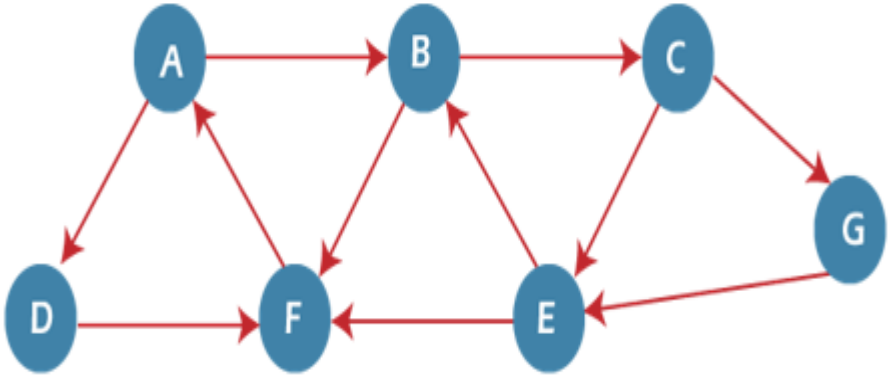
$$T(b) = 1 + b^2 + b^3 + \ldots\ldots + b^d = O(b^d)$$

T= 1 + 2 + 4 + 8 + 16 = 31

Exercise: Apply BFS on Following







Depth First Search

- Depth-First Search (DFS) is a tree traversal algorithm that explores as far as possible along each branch before backtracking.
- In the context of a binary tree, DFS can be implemented using **recursion or a Stack (LIFO QUEUE**). Let's go through the DFS algorithm on a binary tree with an example.

DFS Algorithm on Binary Tree:

Recursive Approach:

- Start at the root node.
- Visit the current node.
  - Recursively apply DFS to the left subtree.
  - Recursively apply DFS to the right subtree.

Stack-Based Approach:

- Push the root node onto the stack.
- While the stack is not empty:
  - Pop a node from the stack.
  - Visit the popped node.
  - Push the right child onto the stack (if exists).
  - Push the left child onto the stack (if exists).

DFS Algorithm with LIFO Queue:

Initialization:

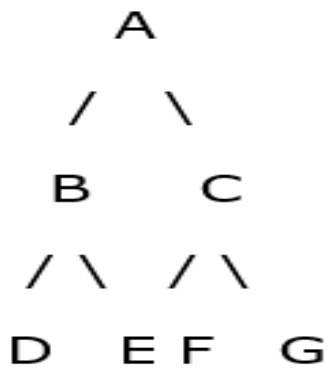
Push the root node onto the LIFO queue.

Traversal:

While the LIFO queue is not empty:

- Pop a node from the LIFO queue.
- Visit the popped node.
- Push the right child onto the LIFO queue (if exists).
- Push the left child onto the LIFO queue (if exists).

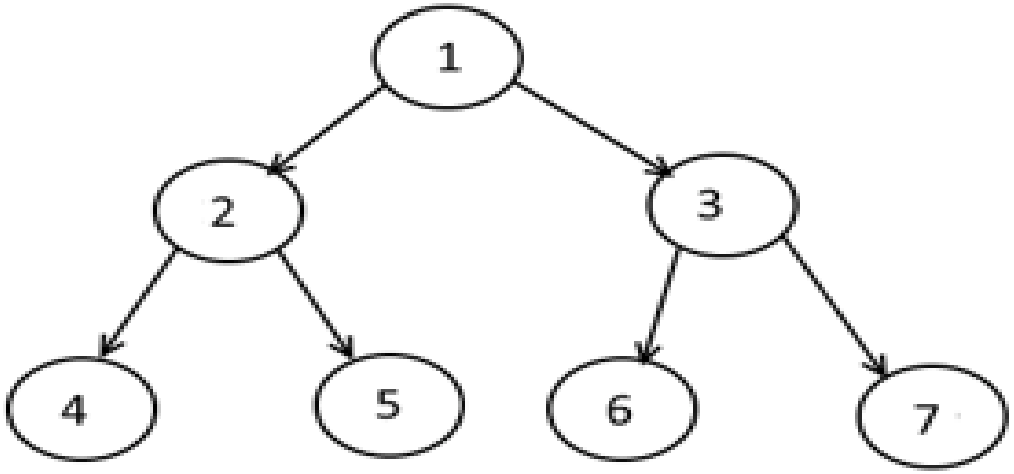
Example



Step	Description	Visited Nodes	LIFO QUEUE
1	Push 'A' onto the LIFO queue	{}	[A]
2	Pop 'A', visit it, and push its children 'B' and 'C' onto the LIFO queue	{A}	[C, B]
3	Pop 'B', visit it, and push its children 'E' and 'D' onto the LIFO queue.	{A, B}	[C, E, D]
4	Pop 'D', visit it (no children to push)	{A,B,D}	[C, E]
5	Pop 'E', visit it (no children to push)	{A,B,D,E}	[C]
6	Pop 'C', visit it, and push its children 'F' and 'G'	{A,B,D,E,C}	[G,F]
7	Pop 'F', visit it (no children to push)	{A,B,D,E,C,F}	[G]
8	Pop 'G', visit it (no children to push)	{A,B,D,E,C,F,G}	[]

Resulting DFS Traversal Order: **A,B,D,E,C,F,G**

Exercise: Apply DFS for the Following



Discuss the Time and Space Complexity of DFS

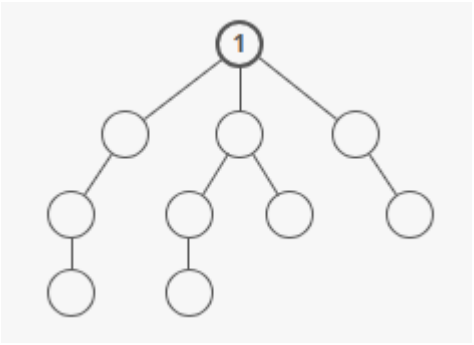
DFS and BFS Time and Space Complexity

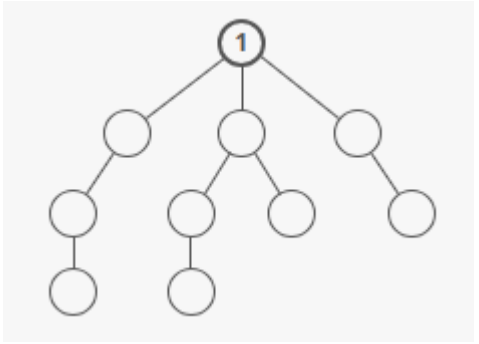
Strategy	Graph		Binary Tree	
	Time Complexity	**Space Complexity**	**Time Complexity**	**Space Complexity**
BFS	**O(V + E)**	**O(V)**	$O(b^d)$	$O(b^d)$
DFS	**O(V + E)**	**O(V)**	$O(b^d)$	$O(bd)$

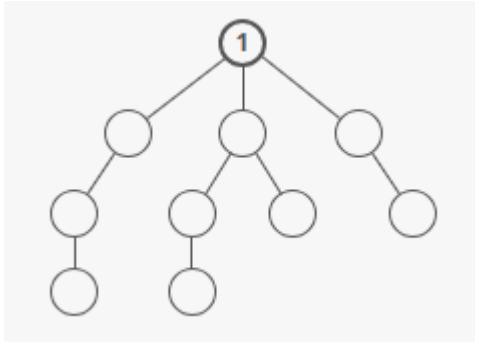
Topics

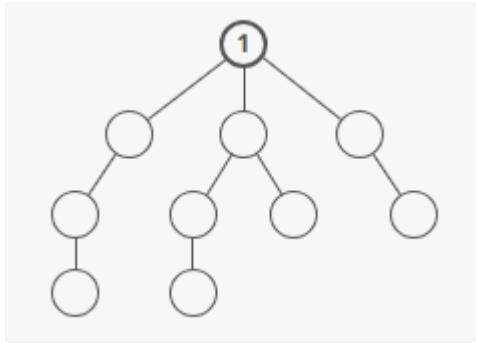
- What is AI?
- Foundations of AI
- History of AI
- Agents, The structure of agents.
- Problem Solving Agents
- Example problems,
- Searching for Solutions,
- Uninformed Search Strategies: Breadth First search, Depth First Search

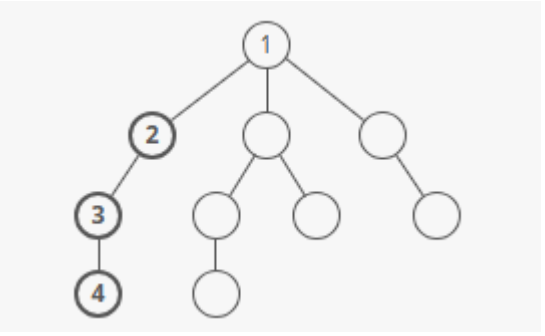
End of Module1

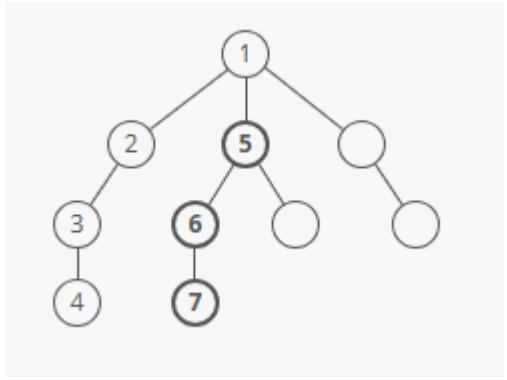

1

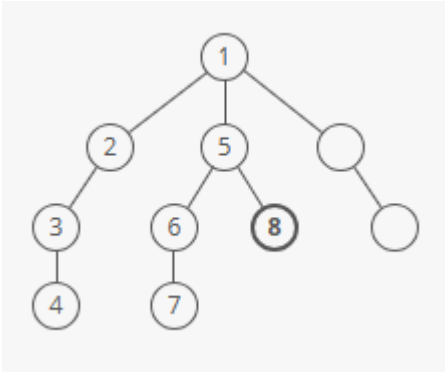

1

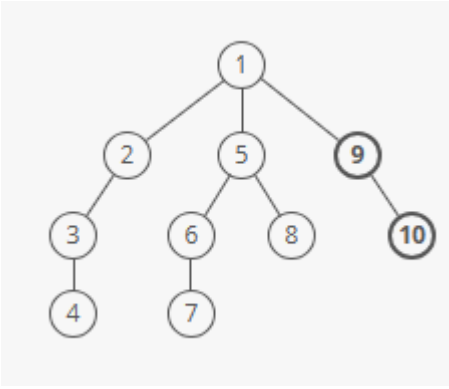

1

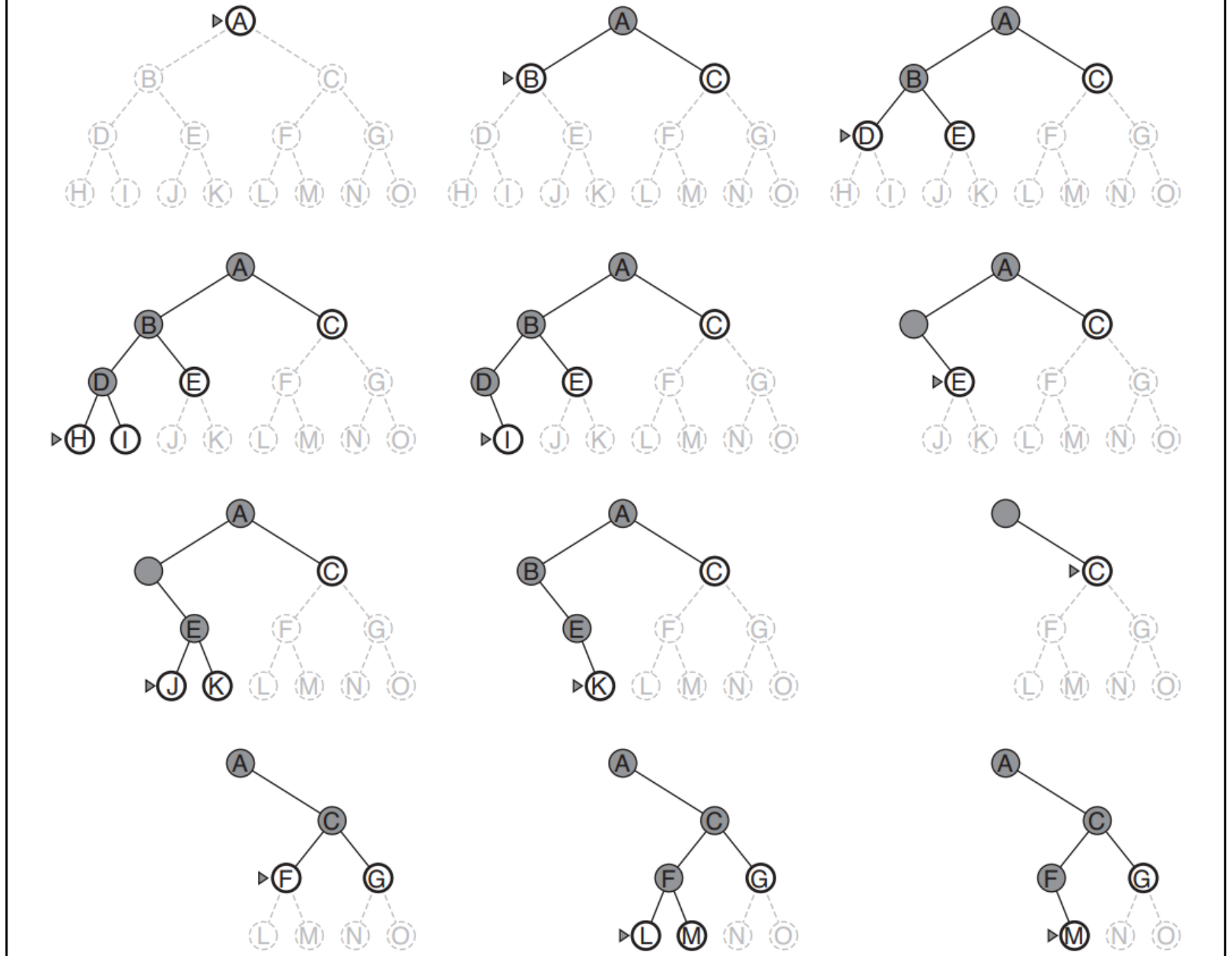

1


1
2
3
4


1
2
5
3
6
4
7

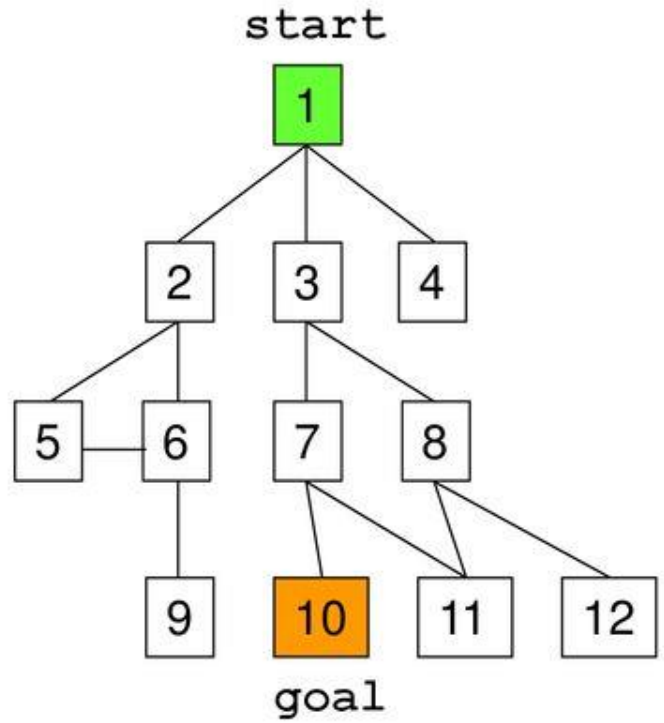

1
2
5
3
6
8
4
7


1
2
5
9
3
6
8
10
4
7


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

Depth First Search—Backtracking

- Problem
  - Discover a path from **`start`** to **`goal`**
- Solution
  - Go deep
    - If there is an unvisited neighbor, go there
  - Backtrack
    - Retreat along the path to find an unvisited neighbor
- Outcome
  - If there is a path from **`start`** to **`goal`**, DFS finds one such path

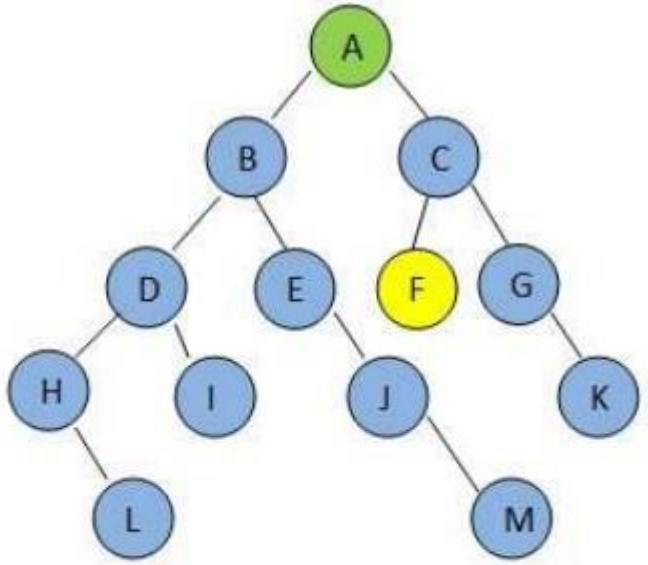


Depth-First Search

- Idea:
  - Starting at a node, follow a path all the way until you cannot move any further
  - Then backtrack and try another branch
  - Do this until all nodes have been visited

  - Similar to finding a route in a maze

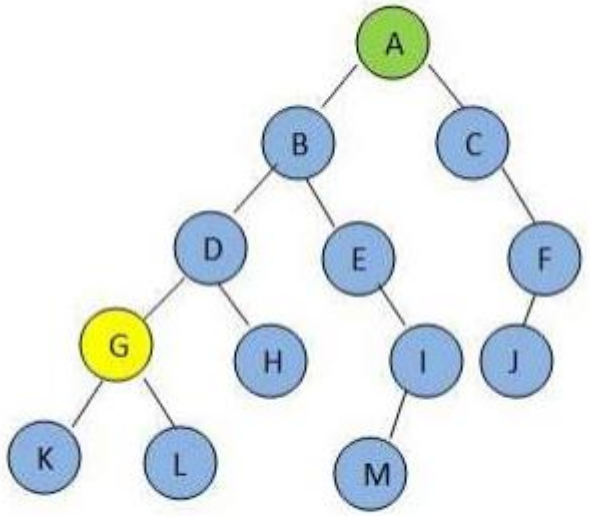
For BFS algorithm, visiting a node's siblings before its children, while in DFS algorithm, visiting a node's children before its siblings



Before countering goal node F:

BFS algorithm encounters nodes: ABCDE

DFS algorithm encounters nodes: ABDHLIEJMC



Before countering goal node G:

BFS algorithm encounters nodes: ABCDEF

DFS algorithm encounters nodes: ABD