# Artificial Intelligence and Machine Learning

21CS54

# Course Outline

- ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING
- Course Code 21CS54
- Teaching Hours/Week (L:T:P: S) 3:0:0:0
- Total Hours of Pedagogy 40
- CIE Marks 50
- SEE Marks 50
- Total Marks 100

# Course Outcomes

- CO 1. Apply the knowledge of searching and reasoning techniques for different applications.
- CO 2. Have a good understanding of machine leaning in relation to other fields and fundamental issues and challenges of machine learning.
- CO 3. Apply the knowledge of classification algorithms on various dataset and compare results
- CO 4. Model the neuron and Neural Network, and to analyze ANN learning and its applications.
- CO 5. Identifying the suitable clustering algorithm for different pattern

# What is AI

- The definition of AI is given in two dimension
  - One is concerned with thought processes and reasoning
  - Other one addresses the behavior
- The definitions on the left measure success in terms of fidelity to human performance, whereas the ones on the right measure against an ideal performance measure, called rationality.

- A system is rational if it does the “right thing,” given what it knows.

# Definition

| **Thinking Humanly** | **Thinking Rationally** |
| --- | --- |
| "The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985)<br><br>"[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)<br><br>"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)<br><br>"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)<br><br>"AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998) |

**Figure 1.1** Some definitions of artificial intelligence, organized into four categories.

# Acting humanly: The Turing Test approach

- Turing Test
  - Alan Turing (1950)
  - Was designed to provide satisfactory operational definition of intelligence
- For a computer to pass the test, it should posses the following capabilities
  - NLP- to enable it to communicate successfully in English
  - Knowledge representation to store what it knows or hears
  - Automated reasoning to use the stored information to answer questions and to draw new conclusion
  - Machine learning to adapt to new circumstances and to detect and extrapolate patterns

# Acting humanly: The Turing Test approach

- Total Turing Test
- In addition to Turing test, tests perceptual abilities
  - Computer vision to perceive objects
  - Robotics to manipulate objects and move about

- Turing test remains relevant even after 60+ years of design

- AI researchers focus – study the underlying principles of intelligence

# Thinking humanly: The cognitive modeling approach

- Understand the actual working of the human brain
- Introspection: trying to catch our own thoughts.
- Psychological experiments: observing person in action.
- Brain imaging: observing the brain in action.

- Cognitive science – brings together computer models from AI and experimental techniques from psychology.

# Thinking rationally: The “laws of thought” approach

- Greek philosopher Aristotle – right reasoning process
- Syllogisms – patterns of argument structures that always yielded correct conclusions
- Ex: All mammals are animals.

  All elephants are mammals.

  Therefore, all elephants are animals
- logic – notations for statements and relations among them
- Problems with the approach
  - Difficult to state informal knowledge as formal facts
  - Exhaust computational resources if not guided in right logic path

# Acting rationally: The rational agent approach

- Agent – something that acts
- Computer agent – operate autonomously, perceive environment, persist, adapt and pursue goals
- Rational agent – acts to achieve the “best outcome”
- Two advantages:
  - More general than the “laws of thought” approach
  - More amenable to scientific development
- Limited rationality – acting appropriately when there is not enough time to do all computations.

# Agents

| Human Agents | |
|---|---|
| Sensors | Eyes, Ears, Nose, Skin, Tongue |
| Actuators | Hands, legs, vocal treat |

| Robotic Agents | |
|---|---|
| Sensors | Camera, Infrared sensors |
| Actuators | Various motors |

| Software Agents | |
|---|---|
| Sensors | Receives keystrokes, files content, network packets |
| Actuators | Display screen, writing files, sending network packets |

# Summary of definition

| | Human Performance | Ideal (wisdom)Performance |
|---|---|---|
| Thought processes | Thinking Humanly | Thinking Rationally |
| Behavior | Acting Humanly | Acting Rationally |

# The foundations of Artificial Intelligence

- Philosophy
- Mathematics
- Economics
- Neuroscience
- Psychology
- Computer Engineering
- Control theory and Cybernetics
- Linguistics

# Philosophy

- 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?
- Rationalism – power of reasoning in understanding of the world
- Dualism – part of the human brain is exempt from physical laws
- Materialism – brains operation under physical laws in mind
- Empiricism – “Nothing is in the understanding which was not first in the senses”

# Mathematics

- What are the formal rules to draw valid conclusions?
- What can be computed?
- How do we reason with uncertain information?
- Algorithm – steps with logic and computation
- Problem types – computable, tractability, NP-completeness
- Greatest contribution of Mathematics to AI – probability theory

# Game Theory

- Decision theory, which combines probability theory with utility theory, provides a formal and complete framework for decisions

- For small economies, the situation is much more like a game:
  - The actions of one player can significantly affect the utility of another (either positively or negatively). Von Neumann and Morgenstern's development of game theory

  Economists did not address the third question listed above, namely, how to make rational decisions when payoffs from actions are not immediate but instead result from several actions taken in sequence

# Economics

- 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?

- Most people think of economics as being about money, but economists will say that they are really studying how people make choices that lead to preferred outcomes

# Operation Research

- This topic was pursued in the field of operations research, which emerged in World War II from efforts in Britain to optimize radar installations, and later found civilian applications in complex management decisions.
- The work of Richard Bellman (1957) formalized a class of sequential decision problems called Markov decision processes

- The pioneering AI researcher Herbert Simon (1916–2001) won the Nobel Prize in economics in 1978 for his early work showing that models based on satisficing—making decisions that are "good enough," rather than laboriously calculating an optimal decision—gave a better description of actual human behavior

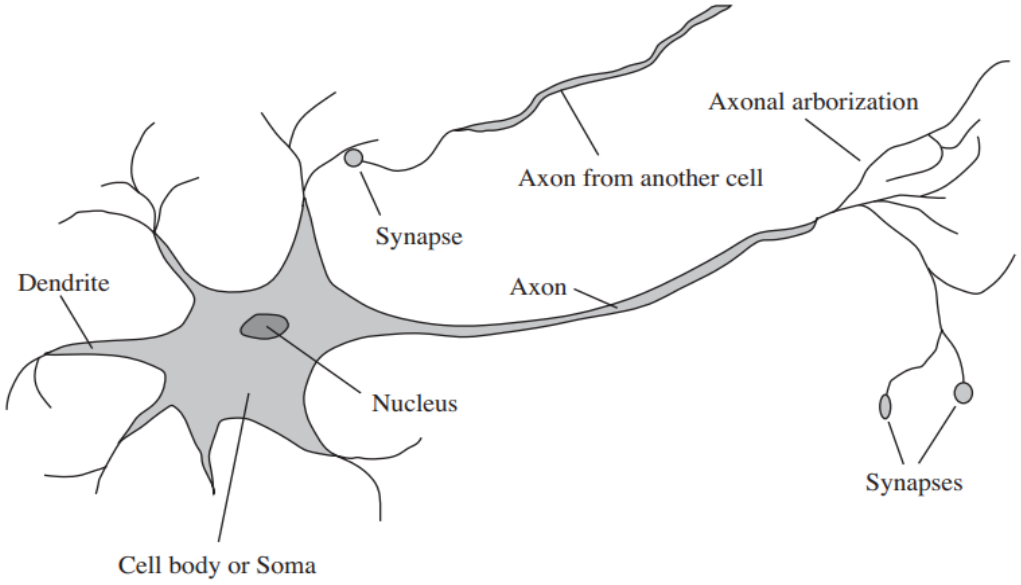
# Neuroscience

- How do brains process information

- Neuroscience is the study of the nervous system, particularly the brain.
- Although the exact way in which the brain enables thought is one of the great mysteries of science
  - The fact that it does enable thought has been appreciated for thousands of years because of the evidence that strong blows to the head can lead to mental incapacitation.

# Neuroscience

- The measurement of intact brain activity began in 1929 with the invention by Hans Berger of the electroencephalograph (EEG).
- The recent development of functiona magnetic resonance imaging (fMRI) (Ogawa et al., 1990; Cabeza and Nyberg, 2001) i giving neuroscientists unprecedentedl detailed images of brain activity, enablin measurements that correspond i interesting ways to ongoing cognitive processes




The truly amazing conclusion is that a collection of simple cells can lead to thought, action, and consciousness or, in the pithy words of John Searle (1992), brains cause minds.

# Neuroscience

| | 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 | $10^{11}$ bits RAM | $10^{11}$ neurons |
| | $10^{15}$ bits disk | $10^{13}$ bits disk | $10^{14}$ synapses |
| Cycle time | $10^{-9}$ sec | $10^{-9}$ sec | $10^{-3}$ sec |
| Operations/sec | $10^{15}$ | $10^{10}$ | $10^{17}$ |
| Memory updates/sec | $10^{14}$ | $10^{10}$ | $10^{14}$ |

**Figure 1.3** A crude comparison of the raw computational resources available to the IBM BLUE GENE supercomputer, a typical personal computer of 2008, and the human brain. The brain's numbers are essentially fixed, whereas the supercomputer's numbers have been increasing by a factor of 10 every 5 years or so, allowing it to achieve rough parity with the brain. The personal computer lags behind on all metrics except cycle time.

# Psychology

- How do humans and animals think and act?

- Biologists studying animal behavior, on the other hand, lacked introspective data and developed an objective methodology, as described by H. S. Jennings (1906) in his influential work Behavior of the Lower Organisms
- Applying this viewpoint to humans, the behaviorism movement, led by John Watson (1878–1958), rejected any theory involving mental processes on the grounds that introspection could not provide reliable evidence.

# Psychology

- Cognitive psychology, which views the brain as an information-processing device, can be traced back at least to the works of William James (1842–1910).
- Helmholtz also insisted that perception involved a form of unconscious logical inference.
- Craik specified the three key steps of a knowledge-based agent:
  - the stimulus must be translated into an internal representation,
  - the representation is manipulated by cognitive processes to derive new internal representations
  - these are in turn retranslated back into action.

# Computer Engineering

- How can we build an efficient computer?
- For artificial intelligence to succeed, we need two things: intelligence and an artifact.
- First operational computer – Alan Turing 1940 – decipher German messages during World War II.
- Computers were developed for military and research purposes.
- Moore’s law – computer hardware and computing power growth
- Modern day – GPUs, cloud - enormous computing infrastructure

# Control theory and cybernetics

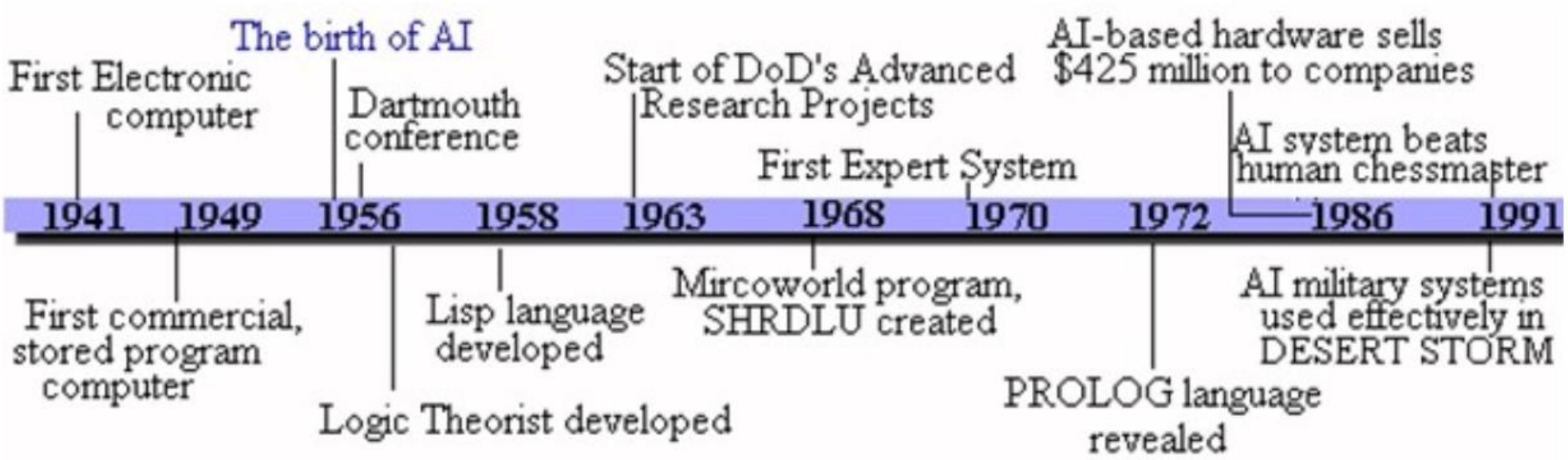
- How can artifacts operate under their own control?
- Ktesibios of Alexandria – first self-controlling machine – a water clock with regulator for controlling flow of water
- Steam Engine , thermostat – control systems
- Modern control theory - design of systems that maximize an objective function over time

# Linguistics

- How does language relate to thought?
- Computational Linguistics

  or
- Natural Language Processing
- Understanding a language – not just sentences or structures
- Involves understanding of subject matter and context
- Knowledge representation

# History


The birth of AI
First Electronic computer
Dartmouth conference
Start of DoD's Advanced Research Projects
First Expert System
AI-based hardware sells $425 million to companies
AI system beats human chessmaster
1941
1949
1956
1958
1963
1968
1970
1972
1986
1991
First commercial, stored program computer
Logic Theorist developed
Lisp language developed
Mircoworld program, SHRDLU created
PROLOG language revealed
AI military systems used effectively in DESERT STORM

# The history of artificial intelligence

- First recognized work on AI : Warren McCulloch and Walter Pitts in 1943
- Three sources:
  - knowledge of the basic physiology and function of neurons in the brain
  - formal analysis of propositional logic
- Turing's theory of computation
- Model of artificial connected neurons
- on and off state – stimulus by connected neighboring neurons

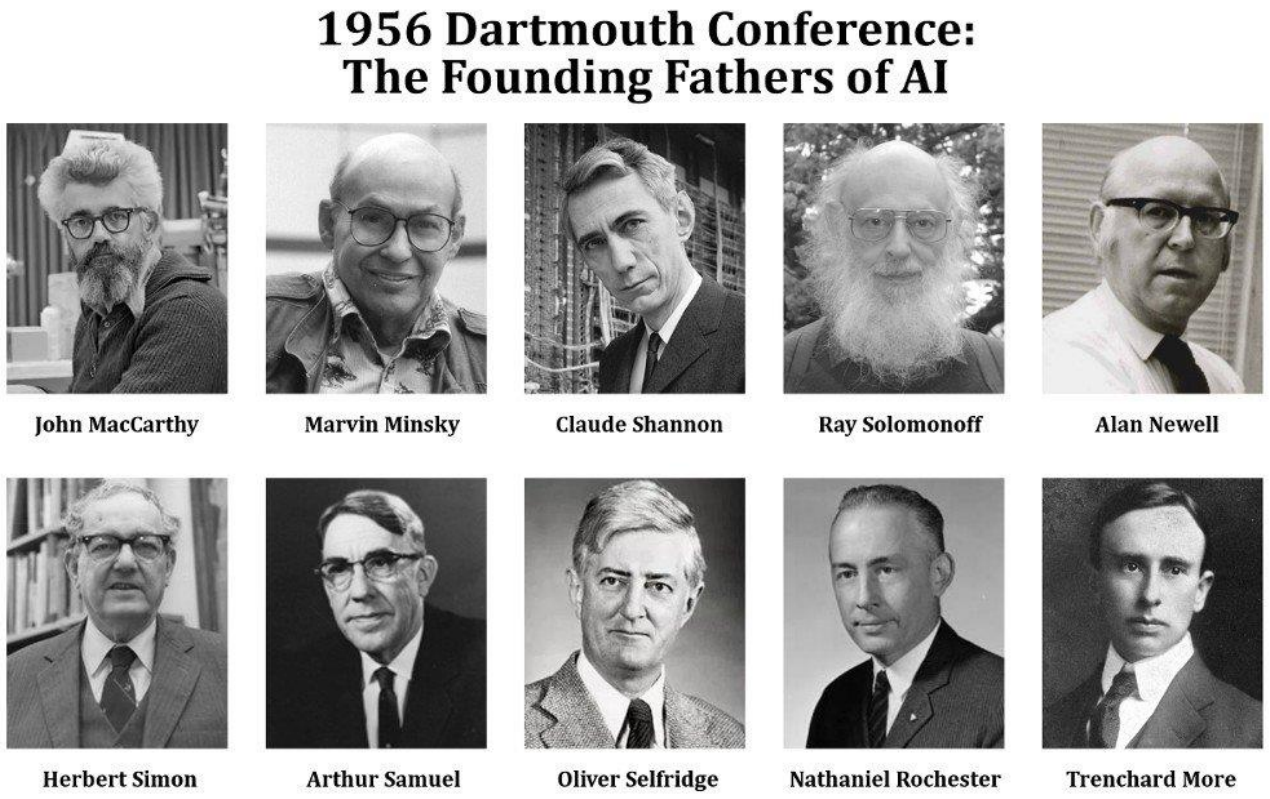
# The history of artificial intelligence

- Hebbian learning – updating rule for modifying connection strengths between neurons
- Undergraduate students at Harvard – built SNARC – first neural network computer
- Alan Turing – introduced Turing Test, machine learning, genetic algorithms, and reinforcement learning

# The birth of AI

- John McCarthy – one of the founding fathers of AI
- Dartmouth workshop, summer 1956
- 10 attendees from various prestigious institutions

# The birth of AI




scienceabc.com

# The history of artificial intelligence

- GPS – General Problem Solver
- Designed to imitate human problem-solving protocols
- Physical symbol system hypothesis - a physical symbol system has the necessary and sufficient means for general intelligent action
- 1958 – Lisp dominant language for AI
- Adaline – network of parallel elements
- Perceptron - element

# The history of artificial intelligence

- Genetic algorithms - making an appropriate series of small mutations to a machine-code program, one can generate a program with good performance for any particular task.

- Knowledge based systems – domain specific knowledge for better reasoning

- Expert systems – Medical diagnosis

# The history of artificial intelligence

- Back-propagation algorithm – 1969
- Hidden Markov Models
- Bayesian decision Theory
- Deep learning – High data availability with high computing power

# Problem solving agents

In which we see how an agent can find a sequence of actions
that achieves its goals when no single action will do

# Problem Solving Agents

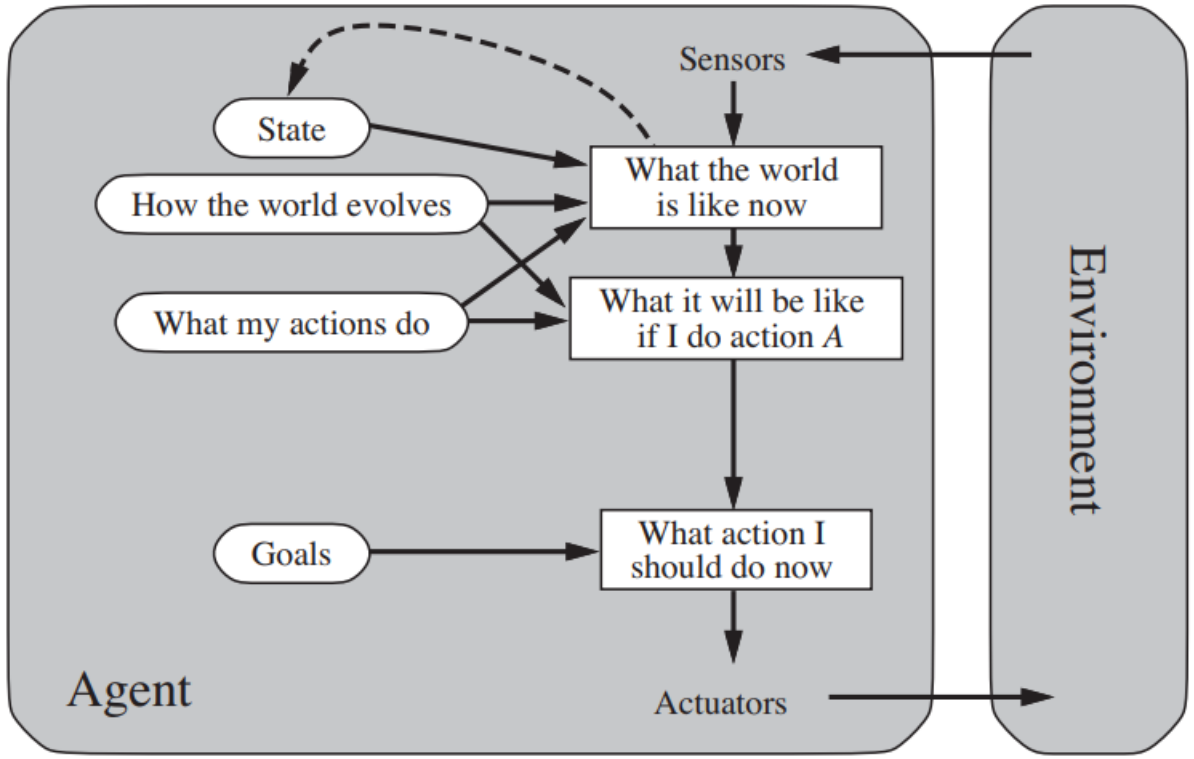
- The job of AI is to design an agent program that implements the agent function— the mapping from percepts to actions

- agent = architecture + program

- goal-based agent called a problem-solving agent are discussed in this chapter

# Goal Based

- Goal-based agents that use more advanced factored or structured representations are usually called planning agent



# Goal Based Agents

- The use atomic representation
  - That is, states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms

- Goal-based agents that use more advanced factored or structured representations are usually called planning agents
- Our discussion of problem solving begins with precise definitions of problems and their solutions and give several examples to illustrate these definitions.

# Goal Based Agents

- We then describe several general-purpose search algorithms that can be used to solve these problems
- Several uninformed search algorithms—algorithms that are given no information about the problem other than its definition.
- Informed search algorithms, on the other hand, can do quite well given some guidance on where to look for solutions.

# Problem Solving Agents

- Intelligent agents are supposed to maximize their performance measure.
- achieving this is sometimes simplified if the agent can adopt a goal and aim at satisfying it.
- Let us first look at why and how an agent might do this.

# Example – Agent 1

- Suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day.
  - In that case, it makes sense for the agent to adopt the goal of getting to Bucharest.
- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it need to consider
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

# Problem Formulation

- Problem formulation is the process of deciding what actions and states to consider, given a goal.
- For now, let us assume that the agent will consider actions at the level of driving from one major town to another. Each state therefore corresponds to being in a particular town.

- Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad. Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.
- None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.

# Problem formulation

- If the agent has no additional information—i.e., if the environment is unknown in the sense
  - Then it has no choice but to try one of the actions at random
- But suppose the agent has a map of Romania.
  - The point of a map is to provide the agent with information about the states it might get itself into and the actions it can take
- The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest.
- Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions
- *An agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.*

# Problem formulation- assumptions

- We assume that the environment is observable, so the agent always knows the current state.
  - For the agent driving in Romania, it's reasonable to suppose that each city on the map has a sign indicating its presence to arriving drivers.
- We also assume the environment is discrete, so at any given state there are only finitely many actions to choose from.
  - This is true for navigating in Romania because each city is connected to a small number of other cities.
- We will assume the environment is known, so the agent knows which states are reached by each action. (Having an accurate map suffices to meet this condition for navigation problems.)
- Finally, we assume that the environment is deterministic, so each action has exactly one outcome.

- The process of looking for a sequence of actions that reaches the goal is called search.
  - A search algorithm takes a problem as input and returns a solution in the form of an action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the execution phase.
- Thus, we have a simple “formulate, search, execute” design for the agent, as shown in Figure 3.1.
- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

# Steps in problem formulation
## Formulate- Search -Execute

```
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 Problem and solution

- Problem can be defined with 5 components-
  - The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as $In(Arad)$
  - A description of the possible actions available to the agent. Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is applicable in s. For example, from the state In(Arad), the applicable actions are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$.
  - A description of what each action does; the formal name for this is the transition model, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s. We also use the term successor to refer to any state reachable from a given state by a single action. For example, we have $RESULT(In(Arad), Go(Zerind)) = In(Zerind)$

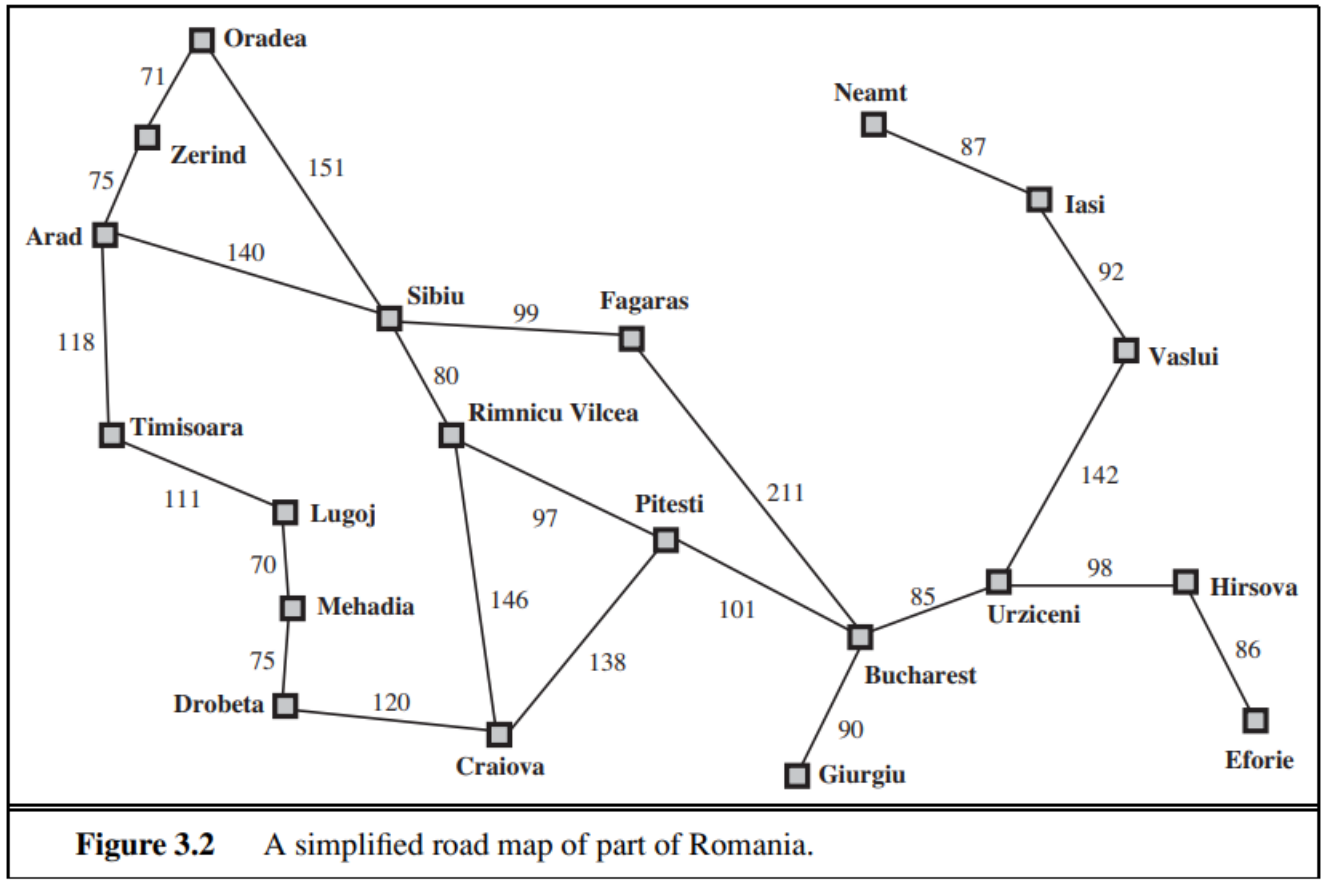



**Figure 3.2** A simplified road map of part of Romania.

# Well Defined Problem and Solution

- Together, the initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions.
- The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions. (The map of Romania shown in Figure 3.2 can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction.)
- A path in the state space is a sequence of states connected by a sequence of actions.

# Well-defined Problem and solution

- The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set $\{In(Bucharest)\}$.
- A path cost function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers.
- In this chapter, we assume that the cost of a path can be described as the sum of the costs of the individual actions along the path

- The step cost of taking action a in state s to reach state s is denoted by c(s, a, s ). The step costs for Romania are shown in Figure 3.2 as route distances. We assume that step costs are nonnegative.

- The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm.
- A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

# Formulating Problems

- In the preceding section we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost
- The process of removing detail from a representation ABSTRACTION is called abstraction
- In addition to abstracting the state description, we must abstract the actions themselves. A driving action has many effects.
- Can we be more precise about defining the appropriate level of abstraction?

# Abstraction level definition

- Now consider a solution to the abstract problem:
  - the path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths.
  - The abstraction is valid if we can expand any abstract solution into a solution in the more detailed world; a sufficient condition is that for every detailed state that is “in Arad,” there is a detailed path to some state that is “in Sibiu,” and so on
  - The abstraction is useful if carrying out each of the actions in the solution is easier than the original problem; in this case they are easy enough that they can be carried out without further search or planning by an average driving agent.

# Abstraction level definition

- The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.
- Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world

# Example-2

- We list some of the best known here, distinguishing between toy and real-world problems
- A TOY PROBLEM toy problem is intended to illustrate or exercise various problem-solving methods
- A real-world problem is one whose solutions people actually care about.
- Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

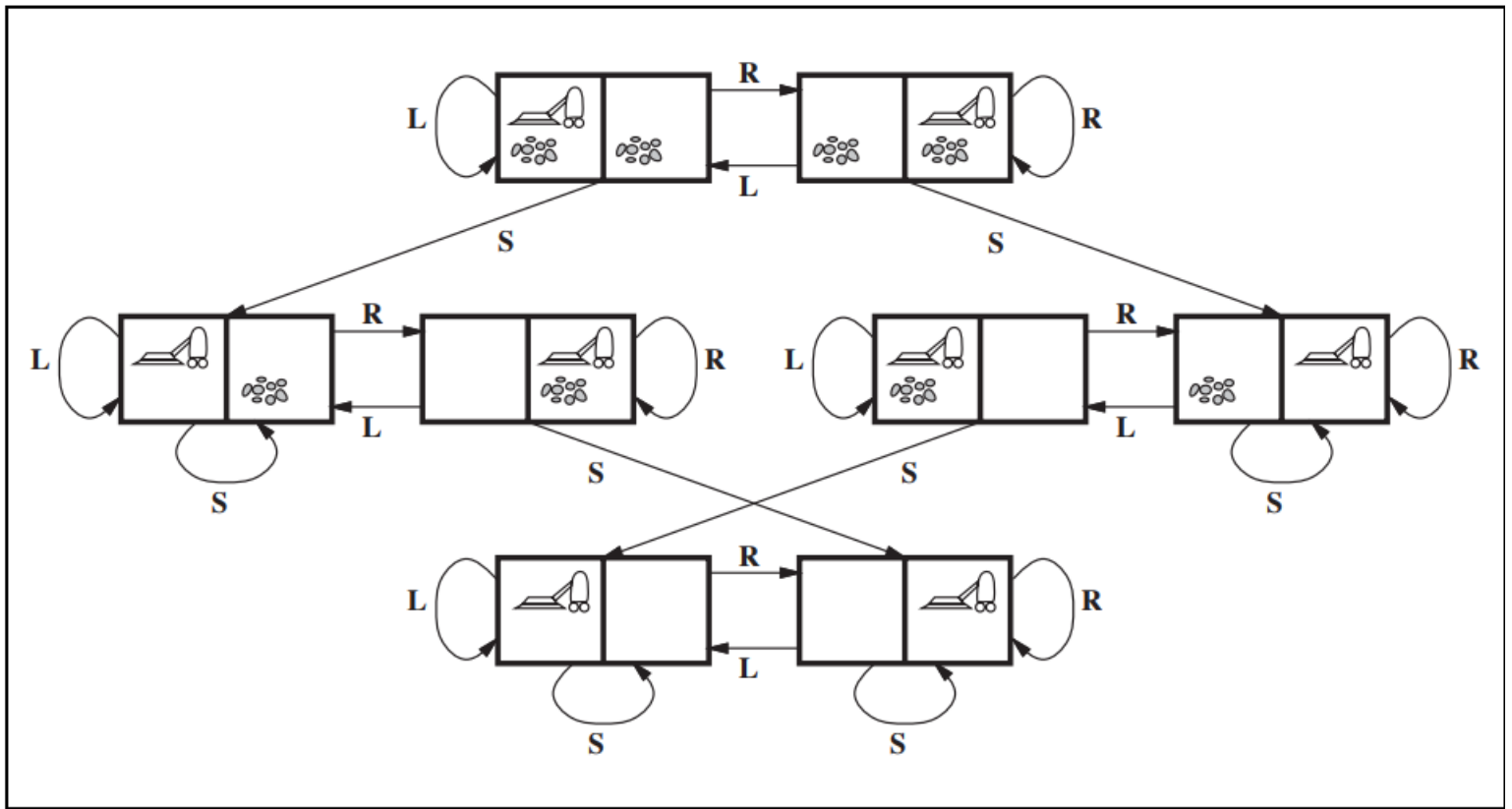



**Figure 3.3** The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

# Toy problems

- The first example we examine is the vacuum world. This can be formulated as a problem as follows:
  - States: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 22 = 8$ possible world states. A larger environment with n locations has $n \cdot 2n$ states.
  - Initial state: Any state can be designated as the initial state.
  - Actions: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
  - Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
  - Goal test: This checks whether all the squares are clean.
  - Path cost: Each step costs 1, so the path cost is the number of steps in the path.

# 8 – puzzle

- consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

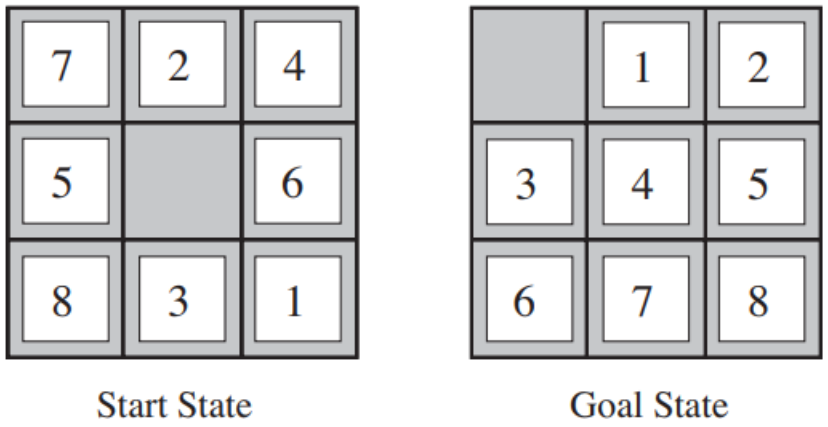



**Figure 3.4** A typical instance of the 8-puzzle.

# 8- puzzle

- States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states
- Actions: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
- Transition model: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure 3.4, the resulting state has the 5 and the blank switched
- Goal test: This checks whether the state matches the goal configuration
- Path cost: Each step costs 1, so the path cost is the number of steps in the path

- The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI.

# Problem formulation

- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state:** Any random arrangement of numbers state can be designated as the initial state.
- **Actions:** movements of the blank space to *Left, Right, Up, or Down. Different subsets of these are possible depending on where* the blank is.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left to the start state in Figure the resulting state has the 5 and the blank* switched.
- **Goal test:** **This checks whether the state matches the goal configuration shown in Figure** 3.4. (Other goal configurations are possible.)
- **Path cost:** **Each step costs 1, so the path cost is the number of steps in the path.**

# 8-queen 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

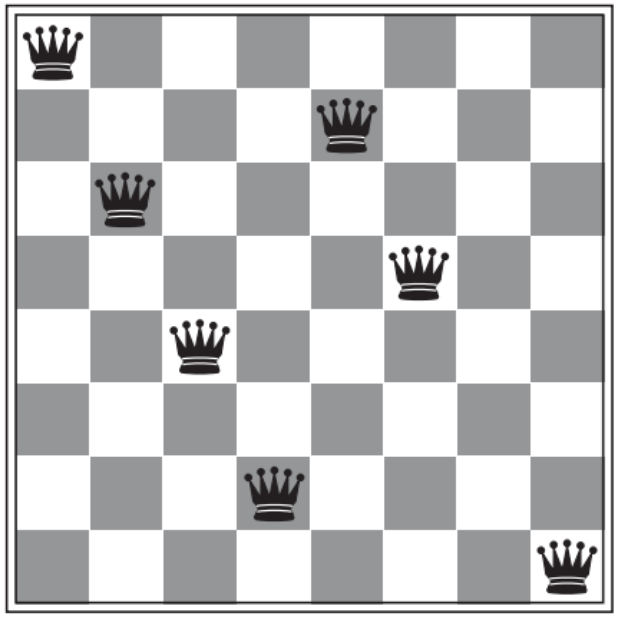


**Figure 3.5** Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

# 8-queen Problem

- 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

# 8-queen Problem

- In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 1014$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:
  - States: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
  - Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
  - Transition model: Returns the board with a queen added to the specified square.
  - Goal test: 8 queens are on the board, none attacked.

# 8-queen Problem

- In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 1014$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:
  - States: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
  - Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

  This formulation reduces the 8-queens state space from $1.8 \times 1014$ to just 2,057, and solutions are easy to find.

# Real World

1. Route finding problem (Travel planning website)
2. Touring problems (Travelling salesman problem)
3. VLSI layout problem
4. Robot navigation
5. Automatic assembly sequencing
6. Protein design

- Assume-
- Source to destination on a particular day in calendar
- If there is a direct flight that will be provided
- Let us assume there are  no direct flight but then he has to travel from multiple paths(indirect)
- Information of all these paths needs to defined by travel website

## Problem definition

# Route Finding Problem (Travel planning Website)

- States: Each state obviously includes a location (e.g., an airport) and the current time.
  - Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “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

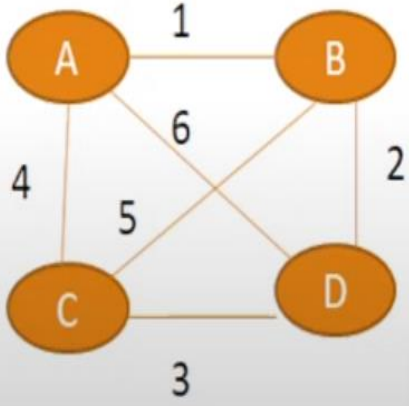
# Touring Problem(Travelling salesman problem)

# Touring Problem(Travelling salesman problem)

- NP hard- no of cites increases the solution is very difficult
- N- cities will be there
- He selects 1 city as src he will have to define hamhamiltonaian cycle
- Start from a city visit every city only once and come back to same city

## Problem definition

It's a NP- Hard problem to find the shortest tour that starts and ends at same city with other cities visited exactly once(Finding optimal hamiltonian cycle).




A-B-C-D-A=15
A-B-D-C-A=10 (Optimal tours)
A-C-D-B-A= 10
A-C-B-D-A= 17
A-D-C-B-A= 15
A-D-B-C-A= 17

For any fully connected graph with N cities, total (N-1)! Solutions exist

# Travelling sales man problem

- State: N cities and nodes
- Initial state: any source city
- Action: moving the neighboring city in the path
- Transition model: reaching next city based in the action
- Goal test: checking optimal tour path(Hamiltonian path)
- Path cost: optimal tour cost

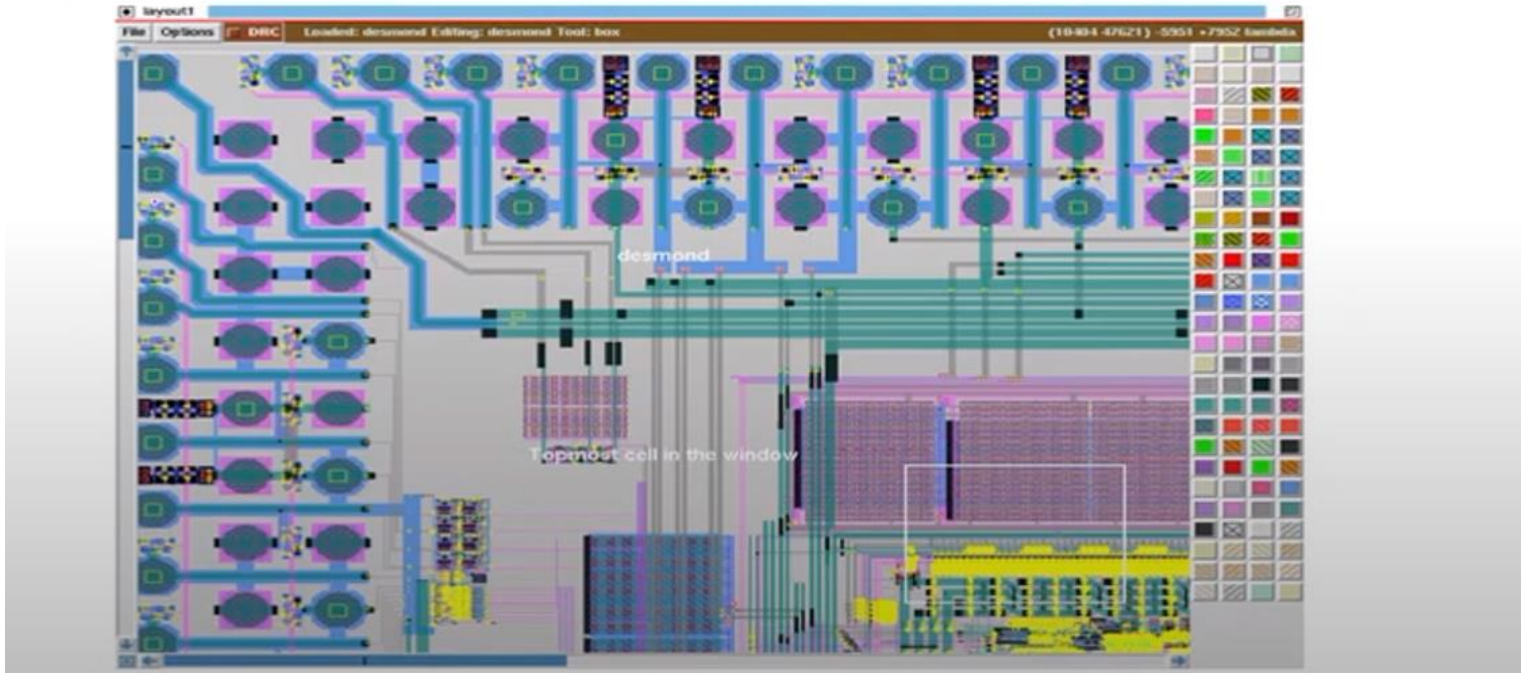
# VLSI layout problem

- Placing millions of components in 1*1 or 1*2board such a way that all components are placed comfortably also giving the required spacing between each component

- The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.

**Problem definition**

positioning millions of components and connections on a chip to **minimize area, minimize circuit delays, minimize stray capacitances, and maximize, manufacturing yield.**



# VLSI Layout Problem

It comes with two parts:

**1. Cell layout :** Placing primitive components of the circuit are grouped into cells, each of which performs some **recognized function**. Each cell has a fixed **footprint (size and shape)** and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.

**2. Channel routing:** finds a specific route for each wire through the gaps between the cells

# VLSI Layout design

- States: Placement of electronic components(gates and transistor) on silicon wafer, current floorplan and placement details
- Initial state: specified by initial floorplan, placement and design parameters
- Actions: Modify the placement of components, adjust the routing paths, refine power distribution
- Transition Model: Update placement of components, adjusted routing information
- Goal: minimize the area while meeting the power consumption and signal delay constraint
- Path cost: depends on Area utilization, power consumption, signal delay, design rules

# Robot Navigation

- 2D space or 3D space
- Our problem delas with working in 3D space
- It requires a complicated algorithm to convert 3D to 2D

When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. Which is complicated than circular robot moving on a flat surface, the space is essentially two-dimensional.

# Robotic Navigation

## Explanation

- Let's consider an example where a robot needs to navigate from its initial position to a specific location in an environment with obstacles. The solution involves a sequence of actions, such as moving forward, rotating to avoid obstacles, and continuously sensing the environment to update the robot's knowledge. The goal is to reach the destination with the least possible path cost.
- The actual implementation of the solution would involve designing algorithms for robotic navigation, obstacle avoidance, and path planning. Sensor fusion techniques and localization algorithms may be used to enhance the robot's understanding of its surroundings. Real-time decision-making and adaptability to dynamic environments are essential aspects of the robotic navigation solution.

# Robotic Navigation

- States: Current Location of robot, orientation, information about obstacles in the environment, sensor data about surrounding
- Initial state: specified by starting position and orientation of the robot
- Actions: Move the robot in specific direction, rotate the robot to a new orientation, sense the environment to update about obstacles
- Transition Model: Update location and orientation of the robot, update information about obstacles based on sensor data
- Goal test: check if robot is at the destination specified in the task
- Path cost: Depends on Distance travelled, number of rotations, time taken for navigation, avoidance of obstacles, energy consumption

# Automatic Assembly sequencing

- Right from packing the milk till fixing up of Mercedes car
- Find out sequence of those assemblies so that we will finish it on the designated time
- If any sequence is missed then redo the task
- Make sure all components placed well

# Automatic Assembly sequencing

## Explanation:

- Let's consider an example where components need to be automatically assembled to create a complex product. The solution involves a sequence of actions, such as attaching components in the correct order, joining subassemblies efficiently, and performing assembly operations with precision. The goal is to complete the entire assembly sequence with the least possible path cost.
- The actual implementation of the solution would involve designing algorithms for automatic assembly, considering factors such as kinematics, collision avoidance, and resource optimization. Robotics and automation technologies are often employed to execute the assembly operations efficiently. Quality control measures may be integrated into the solution to ensure the final product meets specified standards.

# Automatic Assembly sequencing

- States: Current configuration of assembled components, position and orientation, information about partially assembled, state of completion for each step in the assembly sequence
- Initial state: specified by unassembled components and the initial configuration
- Action: attach a component to assembly, positions or orient a component, join subassemblies, perform specific operation
- Transition Model: updated configuration with newly attached components, adjusted position or orientation of components, updated information about subassembly completion, progress in the assembly

# Automatic Assembly sequencing

- Goal Test: Completion of assembly sequence
- Path cost: depends on time taken for each assembly operation, complexity of attaching components, efficiency in subassembly joining, resource utilization and quality control

Solution: Given the problem definition, a solution involves finding a sequence of actions that lead from the initial state to a state where the goal test is satisfied. The solution path is determined by optimizing the assembly process, minimizing time and resource utilization, and ensuring the completion of the assembly sequence.

# Protein design

- Club or mix pack multiple amino acid in a small capsule

Explanation:

The solution involves a sequence of actions, such as mutating amino acids to enhance binding sites, adjusting the protein's three-dimensional structure for optimal interactions, and introducing structural constraints to improve stability. The goal is to design a protein that fulfills the functional requirements with the least possible path cost.

# Problem Formulation

- State: Current amino acid sequence, structure of the protein, information about structural constraints, energy levels and stability metrics
- Initial State: Specified by the initial amino acid sequence and structural parameters
- Actions: Mutate an amino acid in the sequence, adjust conformation, Introduce or modify structural constraints, Optimize energy levels through interactions
- Transitional Model: Updated amino acid sequence, Refined structural constraints, Improved or modified energy levels and stability

# Problem Formulation

- Goal Test: Has protein structure been designed that meets the specified functional requirements and stability criteria?
- Path Cost: depends on, Number and type of mutations, efficiency of structural adjustments, quality of introduced constraints, energy optimization success, stability improvements.

# Searching for Solution

- A solution is an action sequence, so search algorithms work by considering various possible action sequences
- The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem

Figure shows the first few steps in growing the search tree for finding a route from Arad to Bucharest

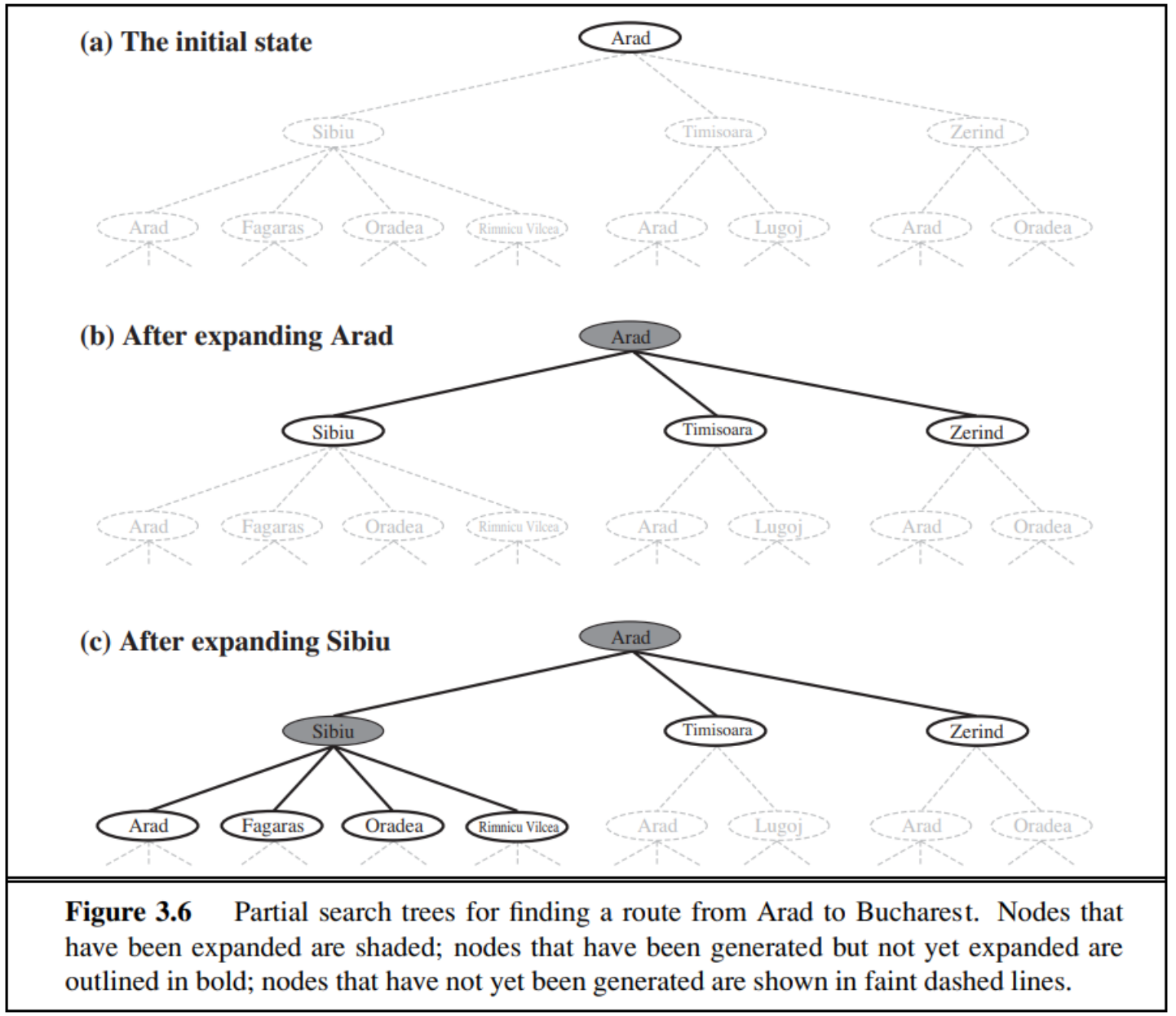



**Figure 3.6** Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

# Steps for finding solution

- The root node of the tree corresponds to the initial state, In(Arad). The first step is to test whether this is a goal state
- Then we need to consider taking various actions.
  - We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states.
  - In this case, we add three branches from the parent node In(Arad) leading to three new child nodes: In(Sibiu), In(Timisoara), and In(Zerind).
  - Now we must choose which of these three possibilities to consider further.

# Steps for finding solution

- This is the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.
- Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get In(Arad), In(Fagaras), In(Oradea), and In(RimnicuVilcea).
- We can then choose any of these four or go back and choose Timisoara or Zerind.
- Each of these six nodes is a leaf node, that is, a node with no children in the tree.
  - The set of all leaf nodes available for expansion at any given point is called the frontier.

# Steps for finding solution

- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.
- The general TREE-SEARCH algorithm is shown informally in Figure 3.7. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called search strategy

**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***

**Figure 3.7** An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

# Steps for finding solution

- The search tree includes the path from Arad to Sibiu and back to Arad again!
- We say that In(Arad) is a repeated state in the search tree, generated in this case by a loopy path. Considering such loopy paths means that the complete search tree for Romania is infinite because there is no limit to how often one can traverse a loop.
- Loopy paths are a special case of the more general concept of redundant paths, which exist whenever there is more than one way to get from one state to another.
  - Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long). Obviously, the second path is redundant—it's just a worse way to get to the same state. If you are concerned about reaching the goal, there's never any reason to keep more than one path

# Steps for finding solution

- Redundant paths are unavoidable.
  - This includes all problems where the actions are reversible, such as route-finding problems and sliding-block puzzles. Route finding on a rectangular grid (used in Figure) is a particularly important example in computer games.
  - In such a grid, each state has four successors, so a search tree of depth ‘d’ that includes repeated states has ‘$4^d$’ leaves; but there are only about $2d^2$ distinct states within d steps of any given state. For d = 20, this means about a trillion nodes but only about 800 distinct states.
  - Thus, following redundant paths can cause a tractable problem to become intractable. This is true even for algorithms that know how to avoid infinite loops

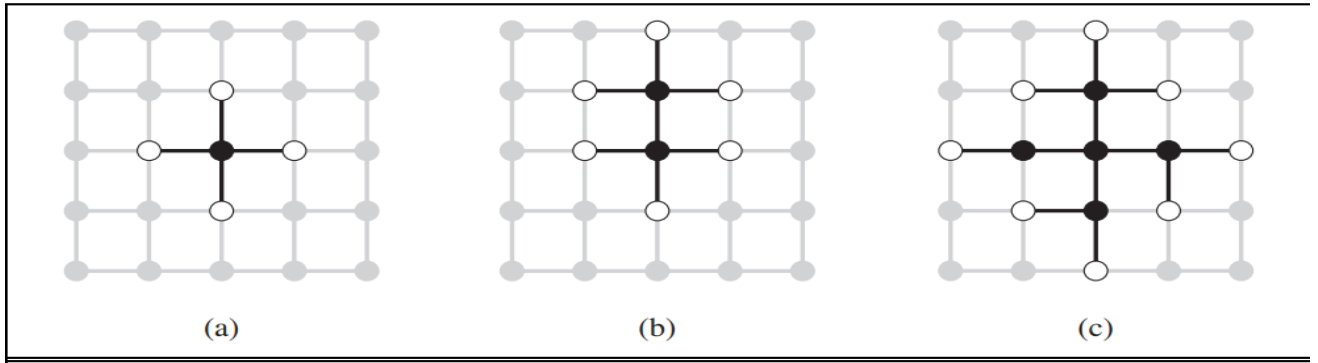



**Figure 3.9** The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has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

# Steps for finding solution

- The way to avoid exploring redundant paths is to remember where one has been.
- To do this, we augment the TREE-SEARCH algorithm with a data structure called the explored set (also known as the closed list), which remembers every expanded node.
- Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.
- The new algorithm, called GRAPH-SEARCH, is shown informally in Figure 3.7. The specific algorithms in this chapter draw on this general design.

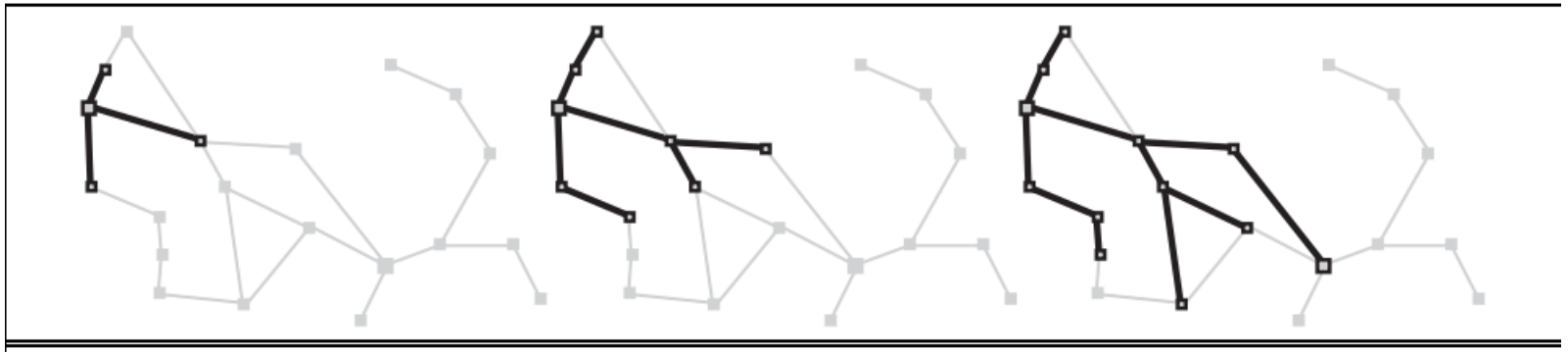


**Figure 3.8** A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other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

# Steps for finding solution

- The search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph, as shown in Figure 3.8.
- The algorithm has another nice property: the frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier. (If this seems completely obvious, try Exercise 3.13 now.)
- This property is illustrated in Figure 3.9. As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution

# 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

# Infrastructure for search algorithms

- The node data structure is depicted in Figure 3.10.
- Notice how the PARENT pointers string the nodes together into a tree structure.
- These pointers also allow the solution path to be extracted when a goal node is found; we use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root.

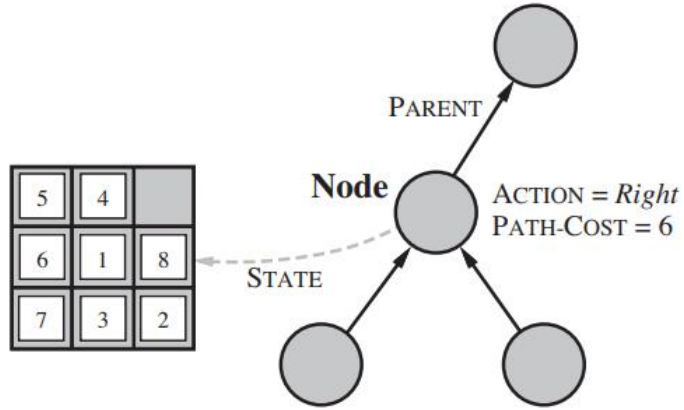



**Figure 3.10** Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

# Infrastructure for search algorithms

- Now that we have nodes, we need somewhere to put them. The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a queue. The operations on a queue are as follows:
  - EMPTY (queue) returns true only if there are no more elements in the queue
  - POP(queue) removes the first element of the queue and returns it
  - INSERT(element, queue) inserts an element and returns the resulting queue

# Infrastructure for search algorithms

- Queues are characterized by the order in which they store the inserted nodes.
- Three common FIFO QUEUE variants are:
  - first-in, first-out or FIFO queue, which pops the oldest element of the queue; LIFO QUEUE
  - last-in first-out or LIFO queue (also known as a stack), which pops the newest element of the queue;
  - priority queue, which pops the element of the queue with the highest priority according to some ordering function.

# Measuring problem-solving performance

- Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them.
- We can evaluate an algorithm's performance in four ways:
  - Completeness: Is the algorithm guaranteed to find a solution when there is one?
  - Optimality: Does the strategy find the optimal solution?
  - Time complexity: How long does it take to find a solution?
  - Space complexity: How much memory is needed to perform the search?

# Measuring problem-solving performance

- In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite.
- For these reasons, complexity is expressed in terms of three quantities: b, the branching factor or maximum number of successors of any node; ‘d’, the depth of the shallowest goal node (i.e., the number of steps along the path from the root); and m, the maximum length of any path in the state space.
- Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.
- For the most part, we describe time and space complexity for search on a tree; for a graph, the answer depends on how “redundant” the paths in the state space are.

# Measuring problem-solving performance

- To assess the effectiveness of a search algorithm, we can consider just the 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.
- For the problem of finding a route from Arad to Bucharest, the search cost is the amount of time taken by the search and the solution cost is the total length of the path in kilometres
- Thus, to compute the total cost, we have to add milliseconds and kilometers.
- There is no “official exchange rate” between the two, but it might be reasonable in this case to convert kilometers into milliseconds by using an estimate of the car’s average speed.
- This enables the agent to find an optimal tradeoff point at which further computation to find a shorter path becomes counterproductive.

# Uninformed Search strategies

- The term means that the strategies have no additional information about states beyond that provided in the problem definition
- Breadth-first search
- Depth-first search

# Breadth-first search

- Is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breadth-first search is an instance of the general graph-search algorithm (Figure 3.7) in which the shallowest unexpanded node is chosen for expansion.
- This is achieved very simply by using a FIFO queue for the frontier.
- Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion.
- This decision is explained below, where we discuss time complexity. Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found.
- Thus, breadth-first search always has the shallowest path to every node on the frontier.

# Algorithm Breadth First Search

```
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.

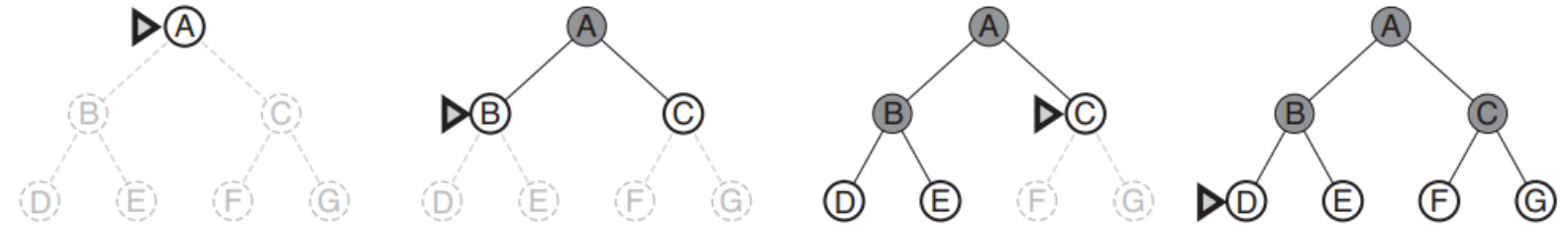


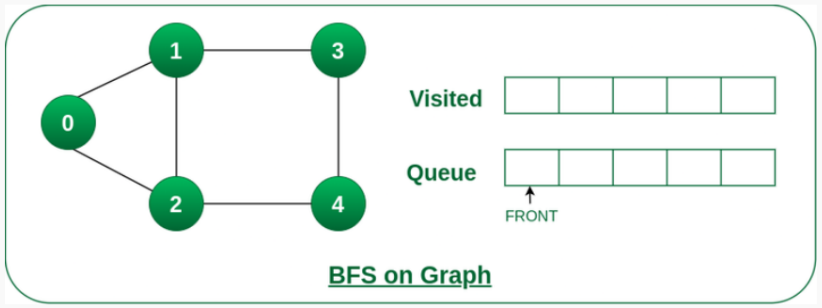

**Figure 3.12** Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

# Breadth First Search

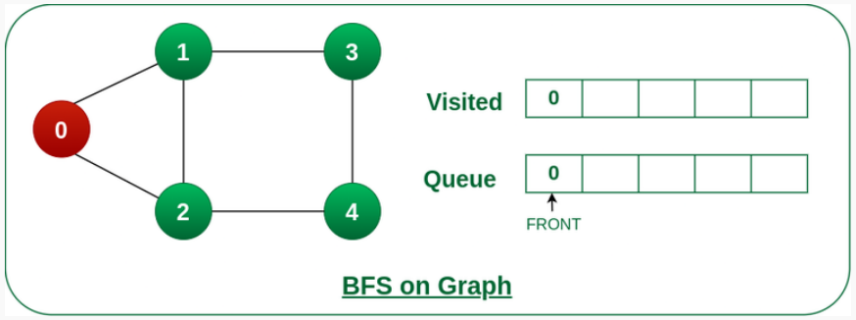
- We can easily see that it is complete—if the shallowest goal node is at some finite depth ‘d’, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite).
- Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test.
- Now, the shallowest goal node is not necessarily the optimal one; technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node.

# Example

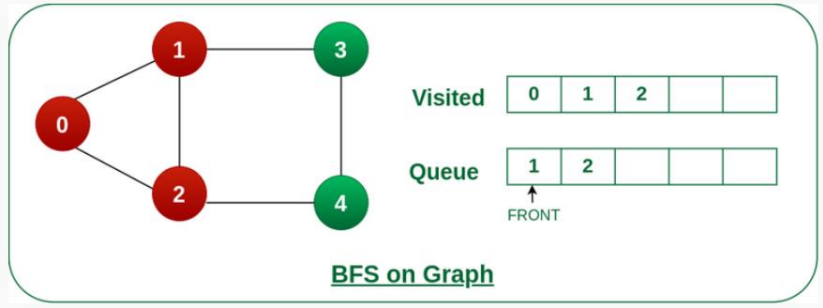
***Step1:*** *Initially queue and visited arrays are empty.*



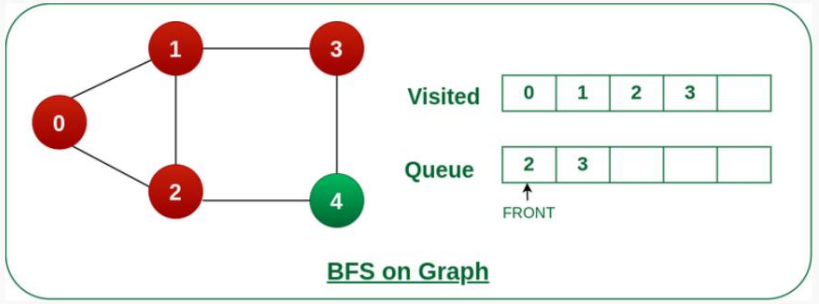

***Step2:*** *Push node 0 into queue and mark it visited.*



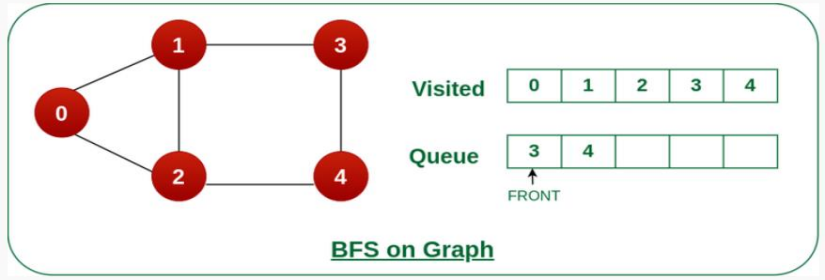

***Step 3:*** *Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



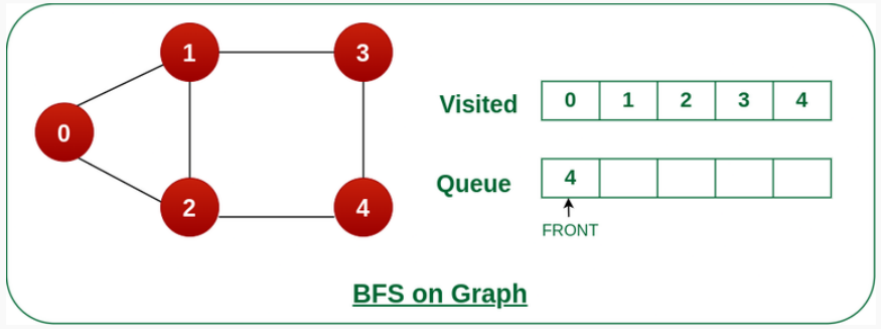

***Step 4:*** *Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



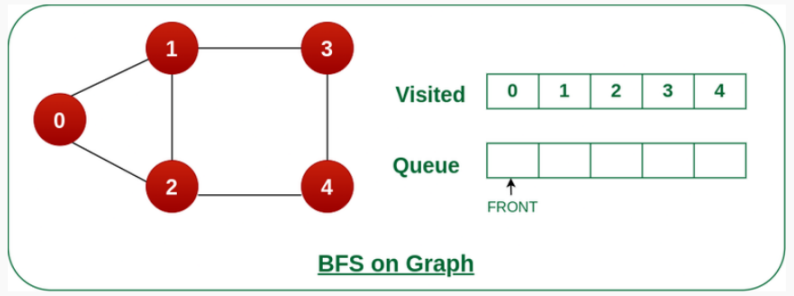

***Step 5:*** *Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*




***Step 6:*** *Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*
*As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.*




***Steps 7:*** *Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*
*As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.*



# Drawback of BFS

- The news about time and space is not so good.
- Imagine searching a uniform tree where every state has ‘b’ successors. The root of the search tree generates ‘b’ nodes at the first level, each of which generates ‘b’ more nodes, for a total of $b^2$ at the second level. Each of these generates b more nodes, yielding $b^3$ nodes at the third level, and so on.
- Now suppose that the solution is at depth ‘d’. In the worst case, it is the last node generated at that level.
- Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

# Drawback of BFS

- As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of ‘b’ of the time complexity.
- For breadth-first graph search in particular, every node generated remains in memory. There will be $O(b^d - 1)$ nodes in the explored set and O(bd) nodes in the frontier

# Depth-first search

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The progress of the search is illustrated in Figure 3.16.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- The depth-first search algorithm is an instance of the graph-search algorithm in Figure 3.7; whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
- A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

# Depth First Search

- As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in
- turn. (A recursive depth-first algorithm incorporating a depth limit is shown in Figure 3.17.)

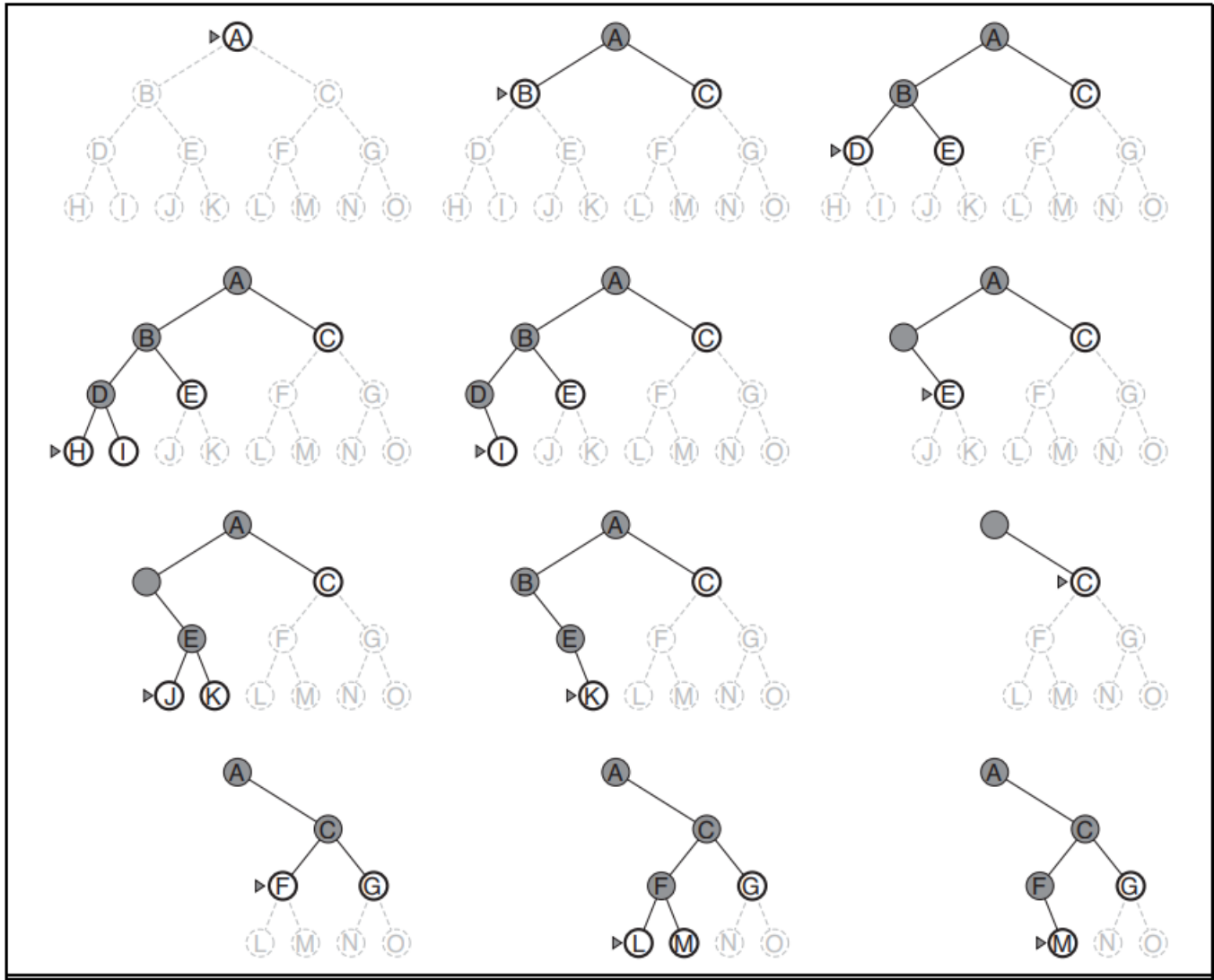


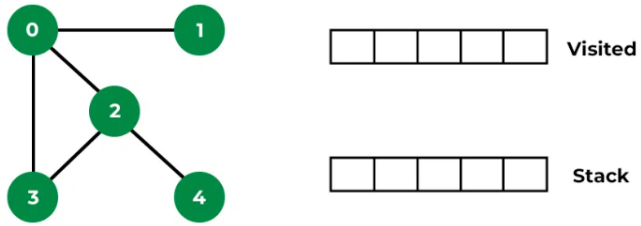
**Figure 3.16** Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and $M$ is the only goal node.

# Backtracking in DFS

- A variant of depth-first search called backtracking search uses still less memory.
- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next.
- In this way, only O(m) memory is needed rather than O(bm). Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by modifying the current state description directly rather than copying it first.
- This reduces the memory requirements to just one state description and O(m) actions. For this to work, we must be able to undo each modification when we go back to generate the next successor. For problems with large state descriptions, such as robotic assembly, these techniques are critical to success

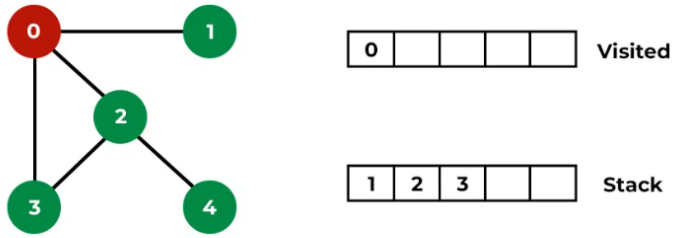
# Example

***Step1:*** *Initially stack and visited arrays are empty.*



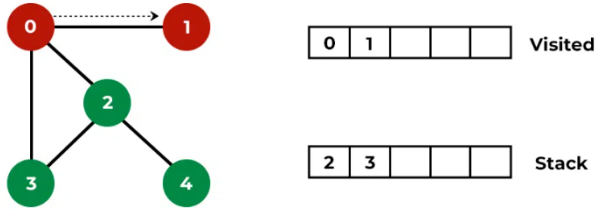

DFS on Graph

***Step 2:*** *Visit 0 and put its adjacent nodes which are not visited yet into the stack.*



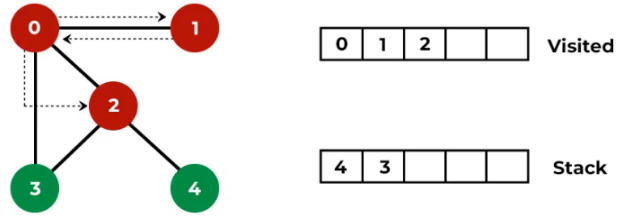

DFS on Graph

***Step 3:*** *Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



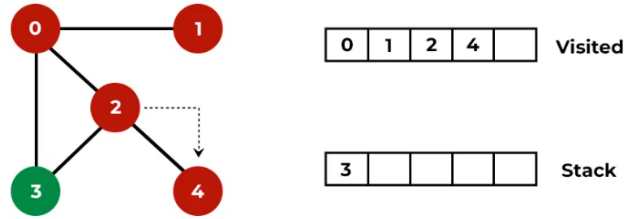

DFS on Graph

***Step 4:*** *Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.*



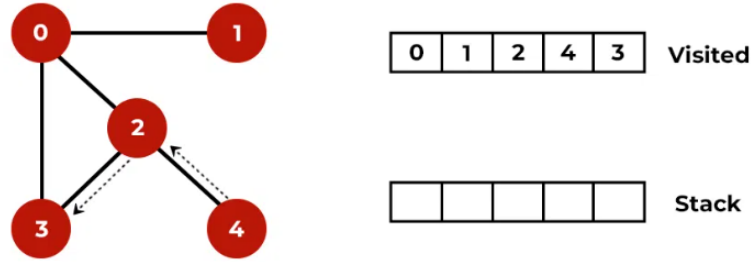

DFS on Graph

***Step 5:*** *Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*




DFS on Graph

***Step 6:*** *Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*




DFS on Graph