ANNAMACHARYA UNIVERSITY

EXCELLENCE IN EDUCATION; SERVICE TO SOCIETY
(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016)
RAJAMPET-516126:A.P; INDIA

DEPARTMENT OF MECHANICAL ENGINEERING

LECTURE NOTES

COURSE TITLE	**: AI & ML for Mechanical Engineering**
COURSE CODE	**: 23A0371T**
REGULATION	**: R-23**
YEAR & SEMESTER	**: IV YEAR I SEM**
CREDITS	**: 3**
ACADEMIC YEAR	**: 2026-27**
PREPARED BY	**: Mr.G Amarnath**

ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES: RAJAMPET
(An Autonomous Institution)
Department of Mechanical Engineering

Title of the Course: **AI & ML for Mechanical Engineering**
Category: PC
Course Code: 23A0371T
Branch/es: Mechanical Engineering
Year & Semester: IV Year I Semester

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	0	0	3

Course Objectives:
1. To Knowledge of Artificial Intelligence, focusing on intelligent agents, problem-solving techniques, and state-space search approaches.
2. To Understand and apply various problem-solving and search techniques, including uniform and heuristic search strategies in artificial intelligence.
3. To Explore and apply local search techniques for solving Constraint Satisfaction Problems (CSPs) and adversarial search strategies to make optimal decisions.
4. To Apply various statistical reasoning techniques for knowledge representation and reasoning in AI, as well as logic programming and reasoning methods.
5. To Familiar in fundamental concepts of Machine Learning techniques, as well as classification, regression, clustering problems, and an introduction to neural networks and deep learning.

Course Outcomes:
At the end of the course, the student will be able to
1. Design intelligent agents, define problems using state-space models, and apply AI techniques.
2. Implement and compare different search algorithms (both uniform and heuristic), apply and analyze appropriate strategies for solving AI problems.
3. Solve CSPs using local search methods and implement adversarial search algorithms to make optimal decisions in competitive game scenarios.
4. Utilize statistical and logical reasoning methods, to represent knowledge and perform forward and backward reasoning in AI applications.
5. Understanding and apply various machine learning techniques, along with an introduction to neural networks and deep learning.

Unit 1 Introduction to Artificial Intelligence and Problem-Solving Agent 10

Problems of AI, AI technique, Tic –Tac – Toe problem. Intelligent Agents, Agents & environment, nature of environment, structure of agents, goal-based agents, utility-based agents, learning agents. Defining the problem as state space search, production system, problem characteristics, and issues in the design of search programs.

Unit 2 Search Techniques 10

Problem solving agents, searching for solutions; uniform search strategies: breadth first search, depth first search, depth limited search, bidirectional search, comparing uniform search strategies. Heuristic search strategies Greedy best -first search, A* search, AO* search, memory bounded heuristic search: local search algorithms & optimization problems: Hill climbing search, simulated annealing search,
local beam search.

Unit 3 Constraint Satisfaction Problems and Game Theory 09

Local search for constraint satisfaction problems. Adversarial search, Games, optimal decisions & strategies in games, the minimax search procedure, alpha-beta pruning, additional refinements, iterative deepening.

Unit 4 Knowledge & Reasoning: Statistical Reasoning 09

Probability and Bays 'Theorem, Certainty Factors and Rule-Base Systems, Bayesian Networks, Dempster-Shafer Theory, Fuzzy Logic. AI for knowledge representation, rule-based knowledge representation, procedural and declarative knowledge, Logic programming, Forward and backward reasoning.

Unit 5 Introduction to Machine Learning: Exploring sub-discipline of AI 10

Machine Learning, Supervised learning, Unsupervised learning, Reinforcement learning, Classification problems, Regression problems, Clustering problems, Introduction to neural networks and deep learning.

Prescribed Textbooks:

1. S. Russell and P. Norvig, —Artificial Intelligence: A Modern Approach‖, Prentice Hall, Third Edition, 2015.
2. Nils J. Nilsson, —Artificial Intelligence: A New Synthesis‖, 1st Edition, Morgan-Kaufmann, 1998.

Reference Books:

1. Elaine Rich, Kevin Knight, & Shiva shankar B Nair, -Artificial Intelligence‖, McGraw Hill, 3rd ed.,2017.
2. Patterson, —Introduction to Artificial Intelligence & Expert Systems‖, Pearson, 1st ed. 2015.
3. Saroj Kaushik, —Logic & Prolog Programming‖, New Age International, Ist edition, 2002.
4. Joseph C. Giarratano,Gary D. Riley, —Expert Systems: Principles and Programming‖, 4th Edition, 2007.

Online Learning Resources:

- https://nptel.ac.in/courses/113104517
- https://nptel.ac.in/courses/127104664
- https://nptel.ac.in/courses/110104164
- https://nptel.ac.in/courses/106106226

CO-PO Mapping:

Course Outcomes	Engineering Knowledge	Problem Analysis	Design/Development of solutions	Conduct investigations of complex problems	Modern tool usage	The engineer and society	Environment and sustainability	Ethics	Individual and team work	Communication	Project management and finance	Life-long learning	PSO1	PSO2
23A0371T.1	3	3	2	1	2	-	-	-	-	-	-	1	-	-
23A0371T.2	3	3	2	2	3	-	-	-	-	-	-	1	-	-
23A0371T.3	3	3	3	2	2	-	-	-	1	-	-	1	-	-
23A0371T.4	3	3	2	2	2	-	-	-	-	1	-	1	-	-
23A0371T.5	3	3	3	2	3	-	-	-	-	1	-	2	-	-

Artificial Intelligence
UNIT-I

Introduction:

Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by John McCarthy in 1956.

Intelligence is the ability to acquire, understand and apply the knowledge to achieve goals in the world.

AI is the study of the mental faculties through the use of computational models

AI is the study of intellectual/mental processes as computational processes.

AI program will demonstrate a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some task.

AI is unique, sharing borders with Mathematics, Computer Science, Philosophy,

Psychology, Biology, Cognitive Science and many others.

Although there is no clear definition of AI or even Intelligence, it can be described as an attempt to build machines that like humans can think and act, able to learn and use knowledge to solve problems on their own.

What is AI?

In Figure 1.1 there are eight definitions of AI, laid out along two dimensions. The definitions on top are concerned with *thought processes* and *reasoning*, whereas the ones on the bottom address *behavior*. The definitions on the left measure success in terms of fidelity to *human* performance, whereas RATIONALITY 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.

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)

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

Acting humanly: The Turing Test approach

The Turing Test, proposed by Alan Turing TURING TEST (1950), was designed to provide a satisfactory operational definition of intelligence. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer. The computer would need to possess the following capabilities:

natural language processing 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 conclusions;
machine learning to adapt to new circumstances and to detect and extrapolate patterns.
computer vision to perceive objects, and
robotics to manipulate objects and move about.

Thinking humanly: The cognitive modeling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get *inside* the actual workings of human minds.

Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking," that is, irrefutable reasoning processes. His **syllogisms** provided patterns for argument structures that always yielded correct conclusions when given correct premises—for example, "Socrates is a man; all men are mortal; therefore, Socrates is mortal." These laws of thought were LOGIC supposed to govern the operation of the mind; their study initiated the field called **logic**.

Acting rationally: The rational agent approach

An **agent** is just something that acts (*agent* comes from the Latin *agere*, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

History of AI:

Important research that laid the groundwork for AI:

In 1931, Goedel layed the foundation of Theoretical Computer Science**1920-30s**:

He published the first universal formal language and showed that math itself is either flawed or allows for unprovable but true statements.

In 1936, Turing reformulated Goedel's result and church's extension thereof.

In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the **Dartmouth Conference**, the first conference devoted to the subject.

In 1957, The **General Problem Solver (GPS)** demonstrated by Newell, Shaw & Simon

In 1958, John McCarthy (MIT) invented the Lisp language.

In 1959, Arthur Samuel (IBM) wrote the first game-playing program, for checkers, to achieve sufficient skill to challenge a world champion.

In 1963, Ivan Sutherland's MIT dissertation on Sketchpad introduced the idea of interactive graphics into computing.

In 1966, Ross Quillian (PhD dissertation, Carnegie Inst. of Technology; now CMU) demonstrated semantic nets

In 1967, Dendral program (Edward Feigenbaum, Joshua Lederberg, Bruce Buchanan, Georgia Sutherland at Stanford) demonstrated to interpret mass spectra on organic chemical compounds. First successful knowledge-based program for scientific reasoning.

In 1967, Doug Engelbart invented the mouse at SRI

In 1968, Marvin Minsky & Seymour Papert publish Perceptrons, demonstrating limits of simple neural nets.

In 1972, Prolog developed by Alain Colmerauer.

In Mid 80's, Neural Networks become widely used with the Backpropagation algorithm (first described by Werbos in 1974).

1990, Major advances in all areas of AI, with significant demonstrations in machine learning, intelligent tutoring, case-based reasoning, multi-agent planning, scheduling, uncertain reasoning, data mining, natural language understanding and translation, vision, virtual reality, games, and other topics.

In 1997, Deep Blue beats the World Chess Champion Kasparov

In 2002,iRobot, founded by researchers at the MIT Artificial Intelligence Lab, introduced **Roomba**, a vacuum cleaning robot. By 2006, two million had been sold.

Foundations of AI:

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?

Mathematics

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

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?

Neuroscience

- How do brains process information?

The Figure 1.2 depicts the basic biological neuron structure.

Psychology

- How do humans and animals think and act?

Computer engineering

- How can we build an efficient computer?

Control theory and cybernetics

- How can artifacts operate under their own control?

Linguistics

- How does language relate to thought?

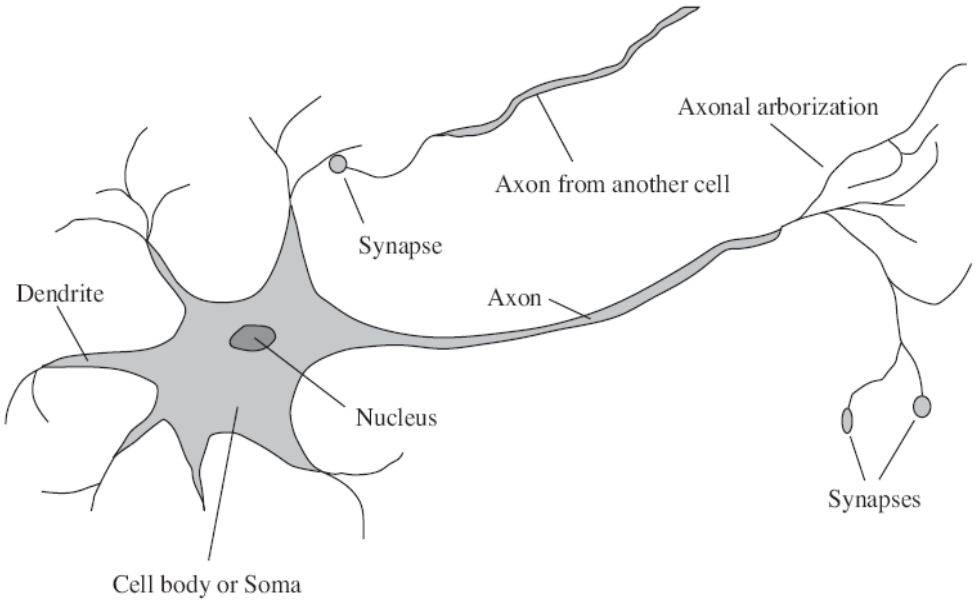


Figure 1.2 The parts of a nerve cell or neuron. Each neuron consists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. The axon stretches out for a long distance, much longer than the scale in this diagram indicates. Typically, an axon is 1 cm long (100 times the diameter of the cell body), but can reach up to 1 meter. A neuron makes connections with 10 to 100,000 other neurons at junctions called synapses. Signals are propagated from neuron to neuron by a complicated electrochemical reaction. The signals control brain activity in the short term and also enable long-term changes in the connectivity of neurons. These mechanisms are thought to form the basis for learning in the brain. Most information processing goes on in the cerebral cortex, the outer layer of the brain. The basic organizational unit appears to be a column of tissue about 0.5 mm in diameter, containing about 20,000 neurons and extending the full depth of the cortex about 4 mm in humans).

The state of Art:

What can AI do today? A concise answer is difficult because there are so many activities inso many subfields. Here we sample a few applications

Robotic vehicles: A driverless robotic car named STANLEY sped through the rough terrain of the Mojave dessert at 22 mph, finishing the 132-mile course first to win the 2005 DARPA Grand Challenge.

Speech recognition: A traveler calling United Airlines to book a flight can have the entire conversation guided by an automated speech recognition and dialog management system.

Autonomous planning and scheduling: A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et al.*, 2000).

Game playing: IBM's DEEP BLUE became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).

Spam fighting: Each day, learning algorithms classify over a billion messages as spam, saving the recipient from having to waste time deleting what, for many users, could comprise 80% or 90% of all messages, if not classified away by algorithms.

Logistics planning: During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation.

Robotics: The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use.

Machine Translation: A computer program automatically translates from Arabic to English, allowing an English speaker to see the headline "Ardogan Confirms That Turkey Would Not Accept Any Pressure, Urging Them to Recognize Cyprus."

INTELLIGENT AGENTS:

The concept to develop a small set of design principles for building successful agents -systems that can reasonably be called **intelligent**.

AGENTS AND ENVIRONMENTS:

- An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.
- This simple idea is illustrated in Figure 2.1. A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and soon for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

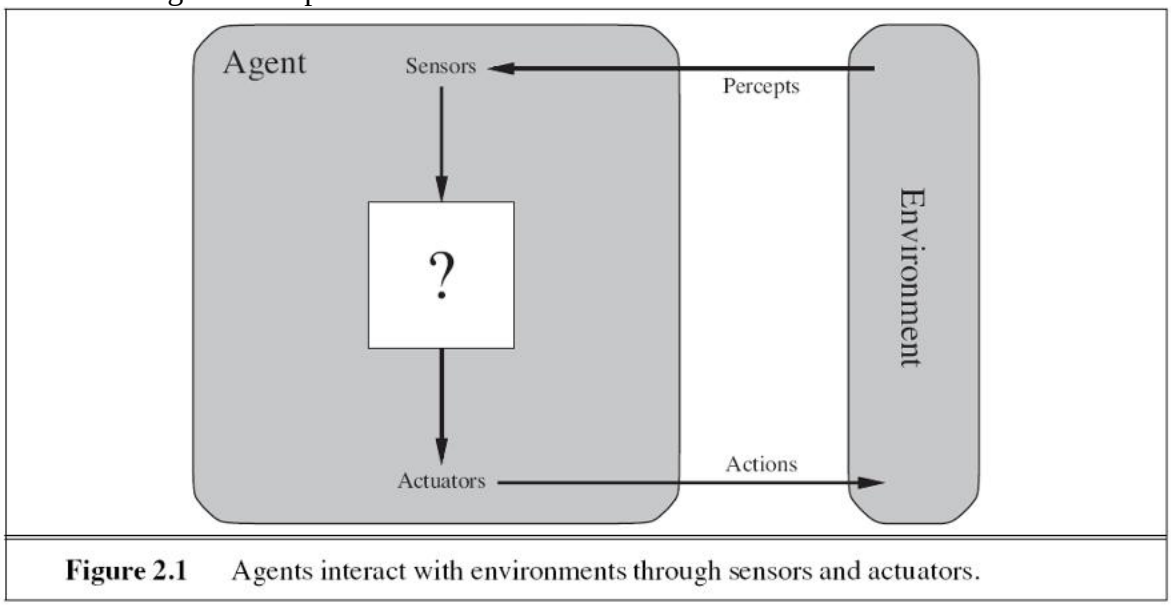


Figure 2.1 Agents interact with environments through sensors and actuators.

- The term **percept** to refer to the agent's perceptual inputs at any given instant. An agent's **percept sequence** is the complete history of everything the agent has ever perceived.
- An agent's behavior is described by the **agent function** that maps any given percept sequence to an action.
- *Internally*, the agent function for an artificial agent will be implemented by an **agent program**.

To illustrate these ideas, we use a very simple example—the vacuum-cleaner world shown in Figure 2.2. This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square. A partial tabulation of this agent function is shown in Figure 2.3

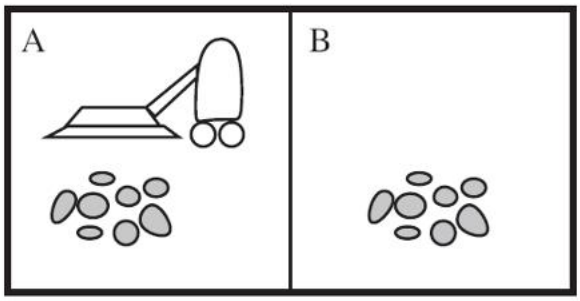


Figure 2.2 A vacuum-cleaner world with just two locations.

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

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

THE CONCEPT OF RATIONALITY

- A **rational agent** is one that does the right thing—conceptually speaking, every entry in the table for the agent function is filled out correctly.
- Rational at any given time depends on four things:
  - The performance measure that defines the criterion of success.
  - The agent's prior knowledge of the environment.
  - The actions that the agent can perform.
  - The agent's percept sequence to date.

Definition of a rational agent: *For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*

THE NATURE OF ENVIRONMENTS:

Task environments, which are essentially the "problems" to which rational agents are the "solutions." we had to specify the performance measure, the environment, and the agent's actuators and sensors. We group all these under the heading of the **task environment**. For the acronymically minded, we call PEAS this the **PEAS** (**P**erformance, **E**nvironment, **A**ctuators, **S**ensors) description. In designing an agent, the first step must always be to specify the task environment as fully as possible.

Figure 2.4 summarizes the PEAS description for the taxi's task environment.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

Some more examples are present in the Figure 2.5

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

Properties of task environments:

The range of task environments that might arise in AI is obviously vast. We can, however, identify a fairly small number of dimensions along which task environments can be categorized.

Fully observable vs. partially observable: FULLY OBSERVABLE If an agent's sensors give it access to the PARTIALLY OBSERVABLE complete state of the environment at each point in time, then we say that the task environment is fully observable. A task environment is effectively fully observable if the sensors detect all aspects that are relevant to the choice of action; relevance, in turn, depends on the performance measure.

Deterministic vs. **stochastic**. If the next state of the environment is completely deter- STOCHASTIC mined by the current state and the action executed by the agent, then we say

the environment is deterministic; otherwise, it is stochastic. a **nondeterministic** environment is one in which actions are characterized by their *possible* outcomes, but no probabilities are attached to them.

Episodic vs. **sequential**: In an episodic task environment, the agent's experience is SEQUENTIAL divided into atomic episodes. In each episode the agent receives a percept and then performs a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. Many classification tasks are episodic.

In sequential environments, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences.

Static vs. **dynamic**: If STATIC the environment can change while an agent is deliberating, then DYNAMIC we say the environment is dynamic for that agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. Dynamic environments, on the other hand, are continuously asking the agent what it wants to do. Taxi driving is clearly dynamic. Crossword puzzles are static.

Discrete vs. **continuous**: The discrete/continuous distinction applies to the *state* of the CONTINUOUS environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent. For example, the chess environment has a finite number of distinct states (excluding the clock). Chess also has a discrete set of percepts and actions. Taxi driving is a continuous-state and continuous-time problem.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

THE STRUCTURE OF AGENTS:

The job of AI is to design an agent program that implements the agent function— the mapping from percepts to actions. We assume this program will run on some sort of ARCHITECTURE computing device with physical sensors and actuators—we call this the architecture:

agent = architecture + program

The agent program takes just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

For example:

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

we outline four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents; and
- Utility-based agents.

Simple reflex agents:
The simplest kind of agent is the **simple** reflex **agent**. These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. An agent program for this agent is shown in Figure 2.8.

```
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
```

Figure 2.8 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in Figure 2.3.

A more general and flexible approach is first to build a general-purpose interpreter for condition– action rules and then to create rule sets for specific task environments. Figure 2.9 gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action.

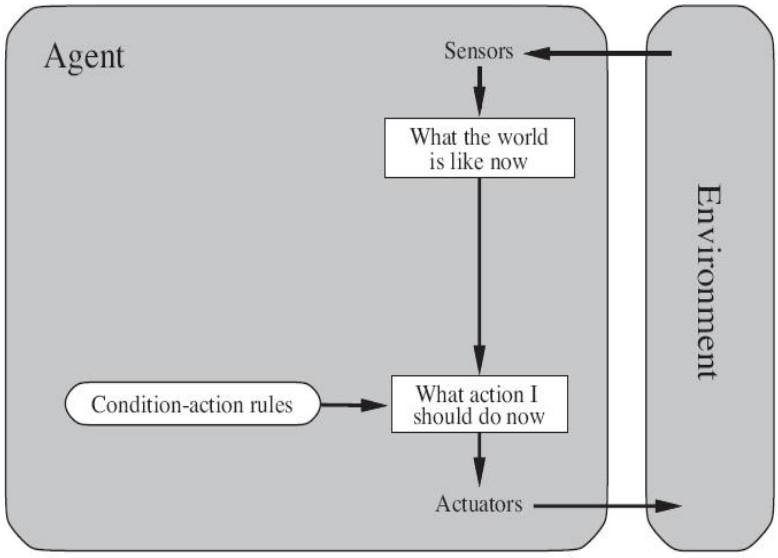


Figure 2.9 Schematic diagram of a simple reflex agent.

The agent program, which is also very simple, is shown in Figure 2.10. The INTERPRET-INPUT function generates an abstracted description of the current state from the percept, and the RULE-MATCH function returns the first rule in the set of rules that matches the given state description. The agent in Figure 2.10 will work *only if the correct decision can be made on the basis of only the current percept—that is, only if the environment is fully observable.*

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

Model-based reflex agents:

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now*. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. An agent that uses such a model is called a **model-based agent**.

Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works. The agent Program is shown in Figure 2.12. The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. The details of how models and states are represented vary widely depending on the type of environment and the particular technology used in the agent design.

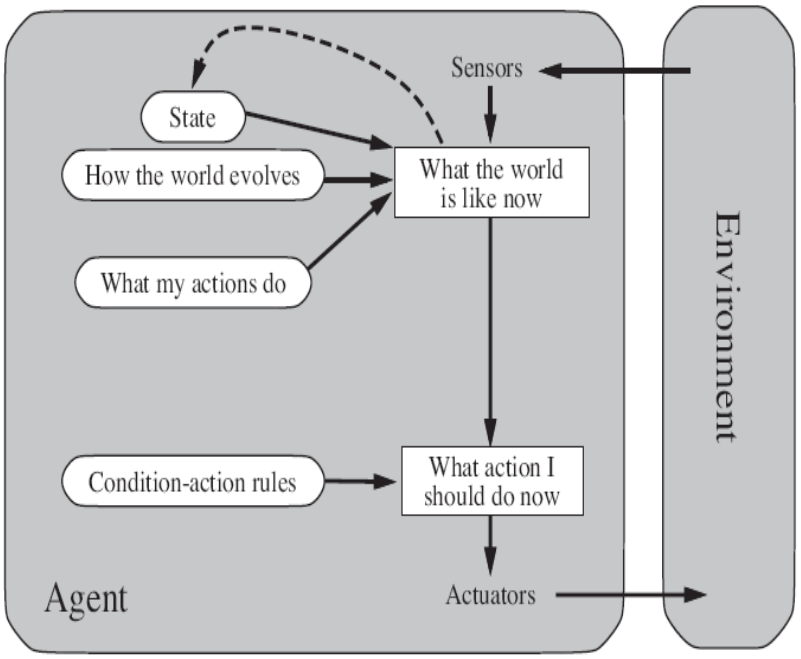


Figure 2.11 A model-based reflex agent.

function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
persistent: *state*, the agent's current conception of the world state
model, a description of how the next state depends on current state and action
rules, a set of condition–action rules
action, the most recent action, initially none

state ← UPDATE-STATE(*state*, *action*, *percept*, *model*)
rule ← RULE-MATCH(*state*, *rules*)
action ← *rule*.ACTION
return *action*

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Goal-based agents:

Knowing something about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the GOAL agent needs some sort of **goal** information that describes situations that are desirable—for example, being at the passenger's destination. The agent program can combine this with the model (the same information as was used in the model based reflex agent) to choose actions that achieve the goal. Figure 2.13 shows the goal-based agent's structure.

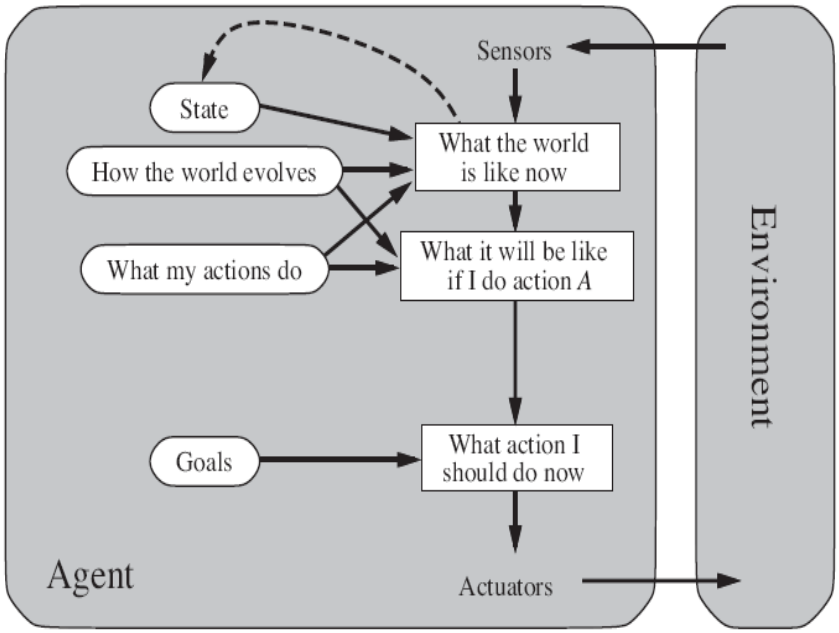


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Utility-based agents:

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent.

An agent's **utility function** is essentially an internalization of the performance measure. If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure. The utility-based agent structure appears in Figure 2.14.

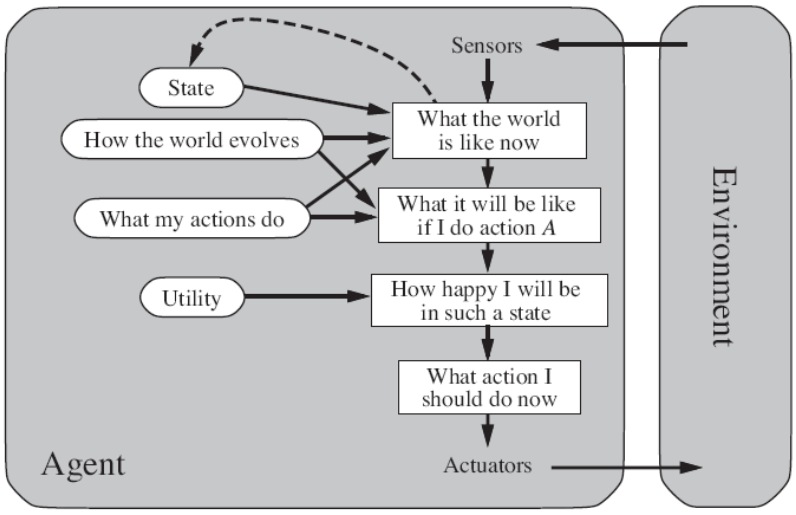


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome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

1. Reference: Stuart J.Russell, Peter Norvig, "Artificial Intelligence A Modern Approach", 3rd Ed, Pearson Education/ Prentice Hall 2019. (In debt thanks)

Artificial Intelligence
UNIT-II

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.
- Goals help organize behaviour by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider. **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.

Well-defined problems and solutions

A **problem** can be defined formally by five 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** ACTIONS available to the agent. Given a particular states, ACTIONS(s) returns the set of actions that can be executed in s.
- 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.
- 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.
- A **path** PATH COST **cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A simple "formulate, search, execute" design for the agent, as shown in Figure 3.1.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

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

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

EXAMPLE PROBLEMS

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 × 22 = 8 possible world states. A larger environment with n locations has n ▪ 2^n 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 *Suck*ing in a clean square have no effect. The complete state space is shown in Figure 3.3.
• **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.

The Figure 3.3 gives a clear idea.

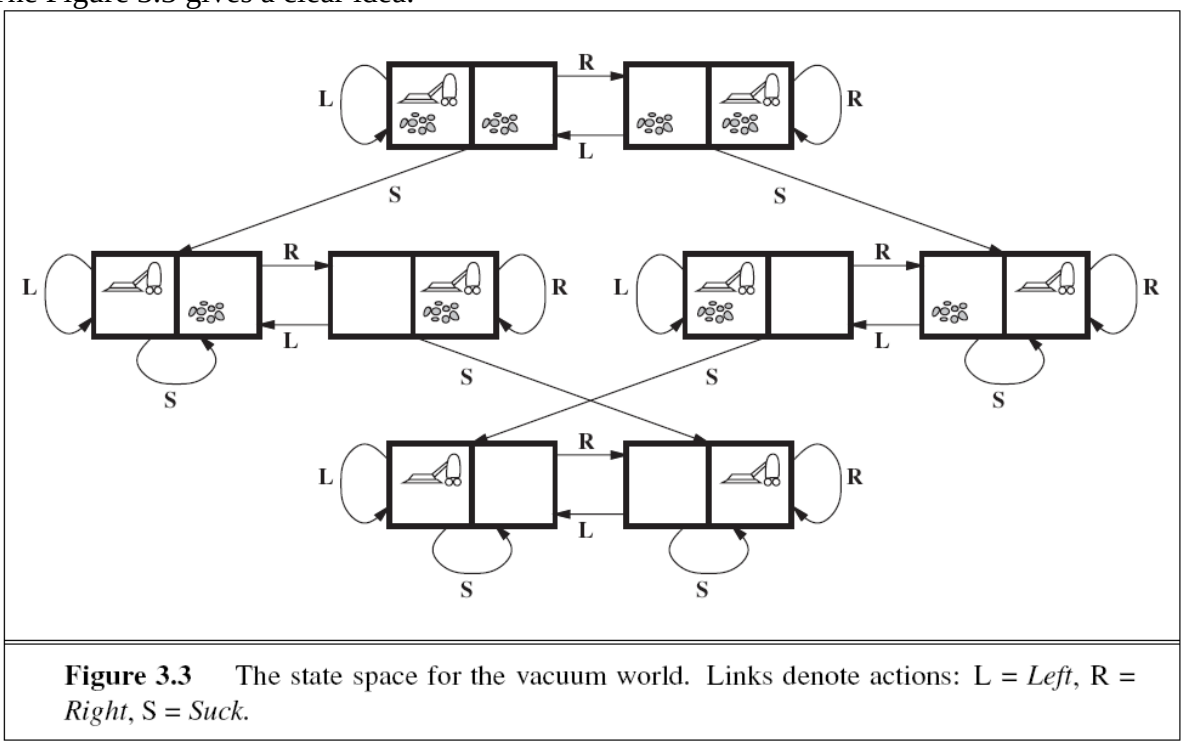


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

8-Puzzle Problem:

8-Puzzle problem 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:
• **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 *eft*, *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;
• **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.
The Figure 3.4 gives clear idea.

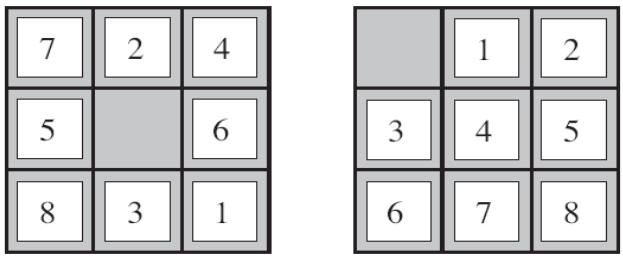


Figure 3.4 A typical instance of the 8-puzzle.

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.

The Figure 3.5 gives clear idea

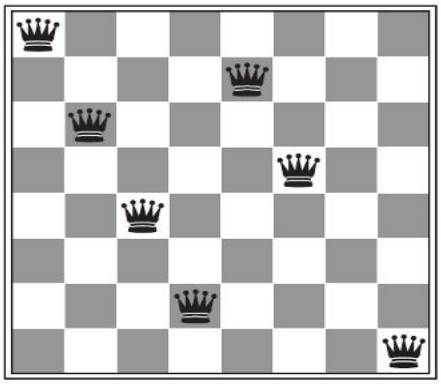


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

Route Finding Problem:

• **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.

An example of a route finding problem is shown in the Figure 3.2

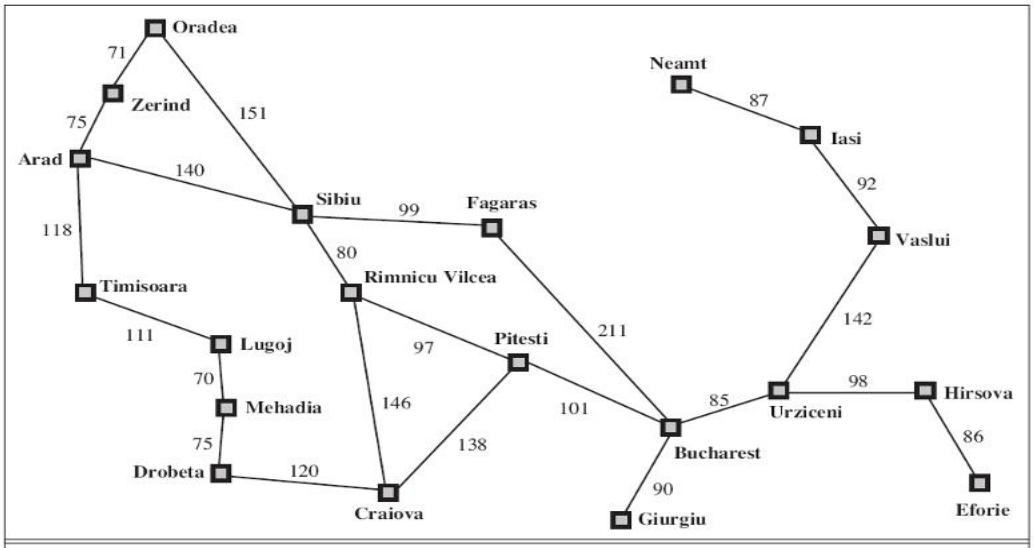


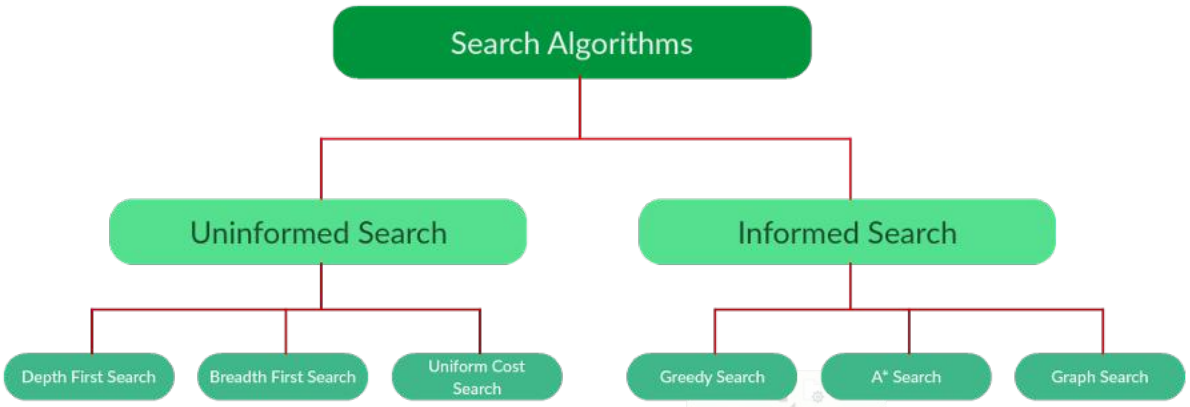
Figure 3.2 A simplified road map of part of Romania.

UNINFORMED SEARCH STRATEGIES:

This section covers several search strategies that come under the heading of **uninformed search** (also called **blind search**).

The term means that the strategies have no additional information about states beyond that provided in the problem definition.

All they can do is generate successors and distinguish a goal state from a non-goal state.



Following are the various types of uninformed search algorithms:

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

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

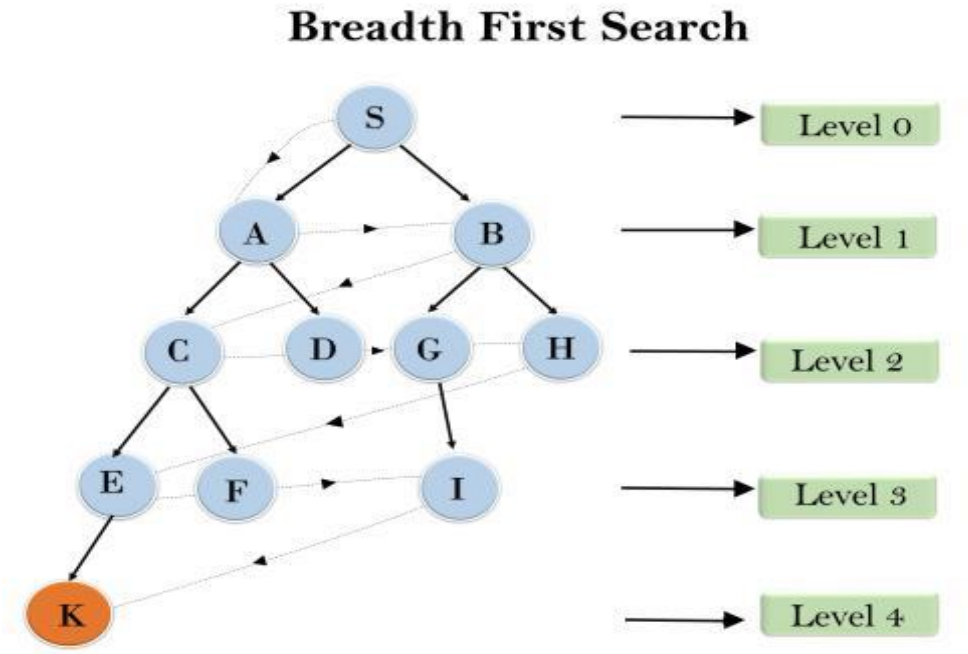
It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



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.

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

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.
Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.

- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

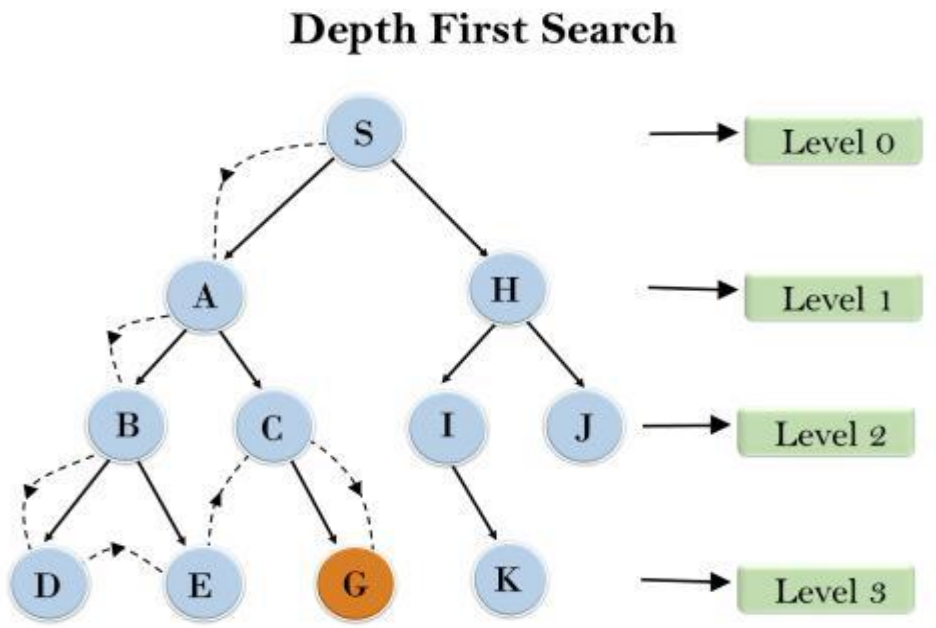
Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.



Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

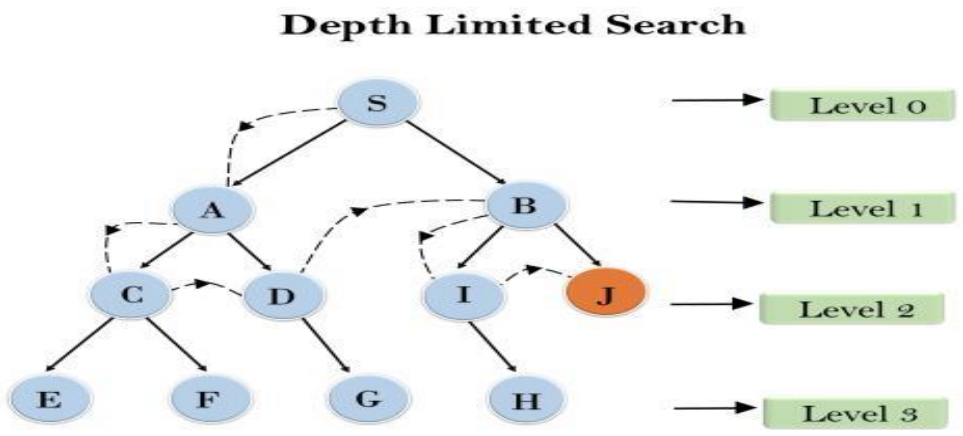
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.
Time Complexity: Time complexity of DLS algorithm is **O(b^{ℓ})**.
Space Complexity: Space complexity of DLS algorithm is O**(b×ℓ)**.
Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

4. Uniform-cost Search Algorithm:

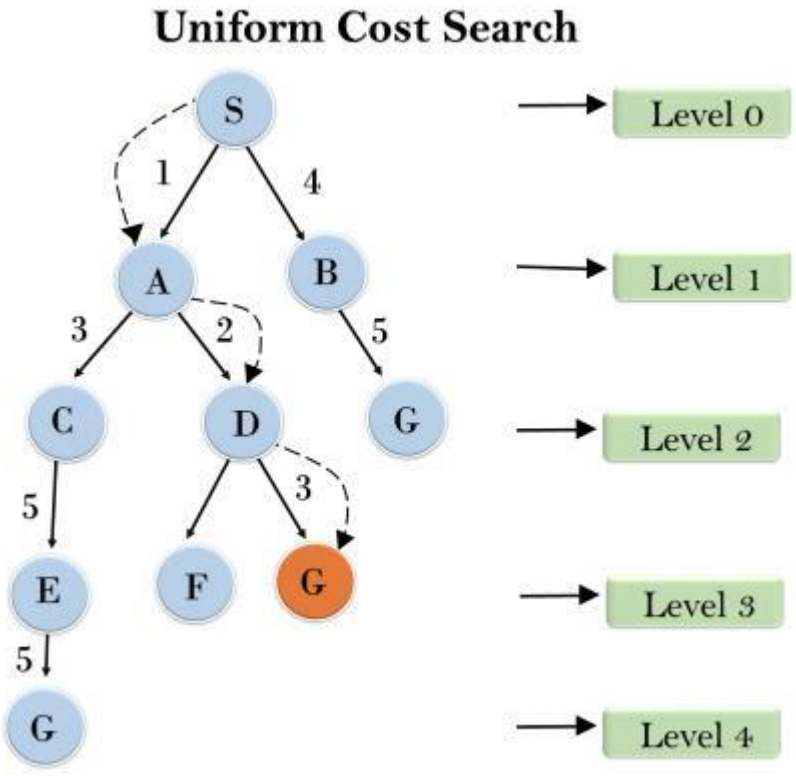
Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge.
The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.
A uniform-cost search algorithm is implemented by the priority queue.
Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is**O(b$^{1 + \lfloor C^*/\varepsilon \rfloor}$)/**.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **O(b$^{1 + \lfloor C^*/\varepsilon \rfloor}$)**.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepeningdepth-first Search:

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

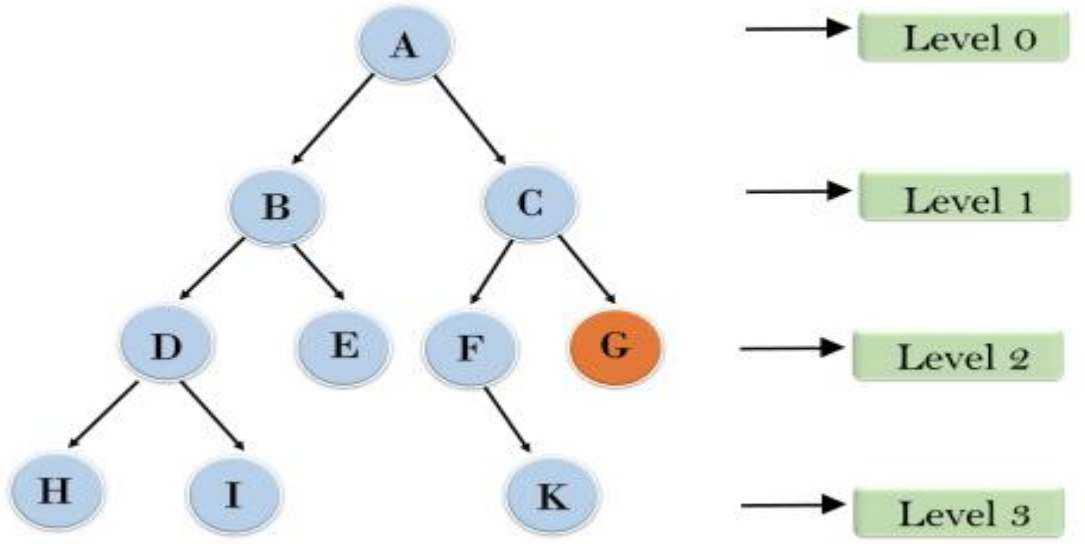
- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Completeness:
This algorithm is complete is ifthe branching factor is finite.
Time Complexity:
Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(b^d)**.
Space Complexity: The space complexity of IDDFS will be **O(bd)**.
Optimal:IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

Iterative deepening depth first search



1'stIteration----->A

2'ndIteration---->A,B,C

3'rdIteration------>A,B,D,E,C,F,G

4'thIteration------>A,B,D,H,I,E,C,F,K,G

In the fourth iteration, the algorithm will find the goal node.

6. Bidirectional Search Algorithm:

- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

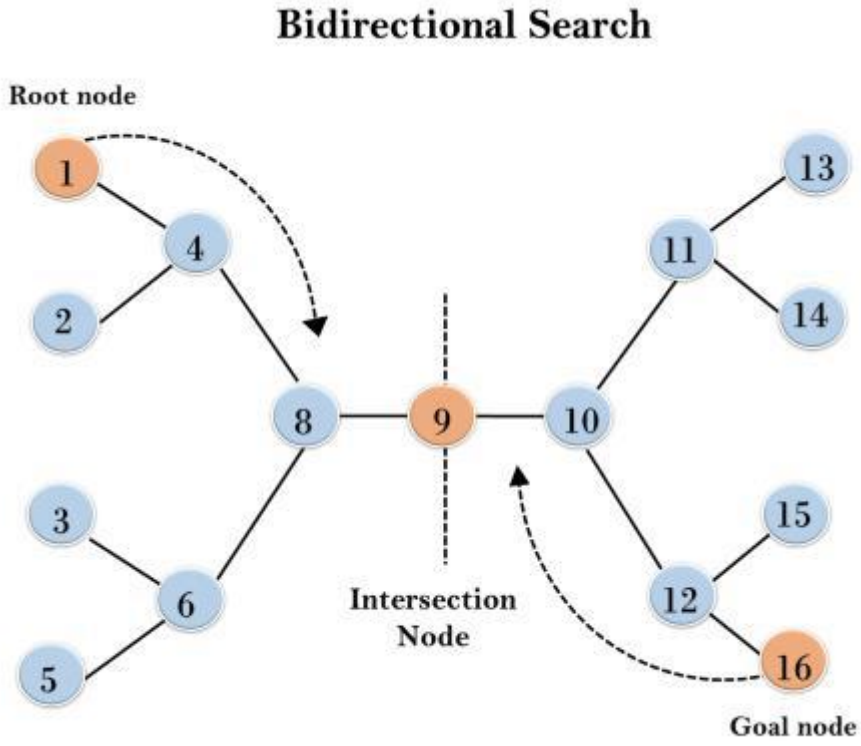
- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Example: In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.
Time Complexity: Time complexity of bidirectional search using BFS is **O(b^d)**.
Space Complexity: Space complexity of bidirectional search is **O(b^d)**.
Optimal: Bidirectional search is Optimal.

Pseudo codes for uninformed search strategies are:

1) 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.

2) Depth Limited Search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

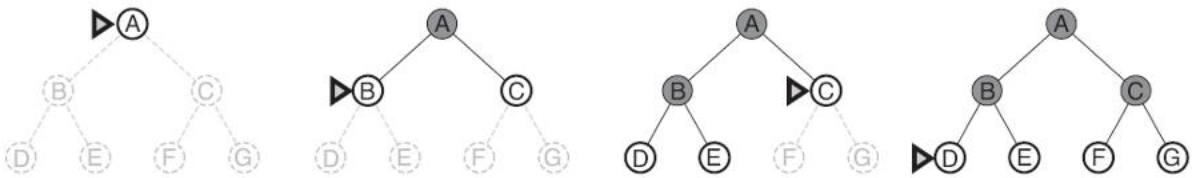
3) Iterative Deepening Search:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

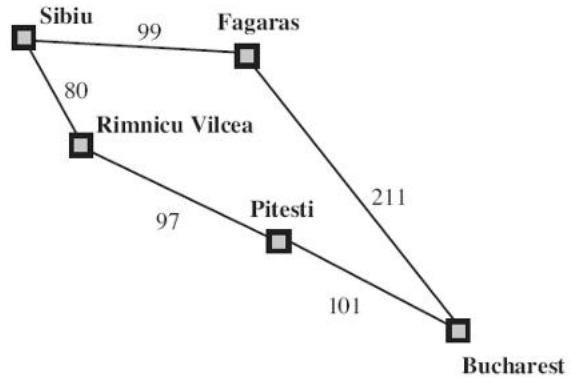
Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Some more examples for uninformed search strategies:

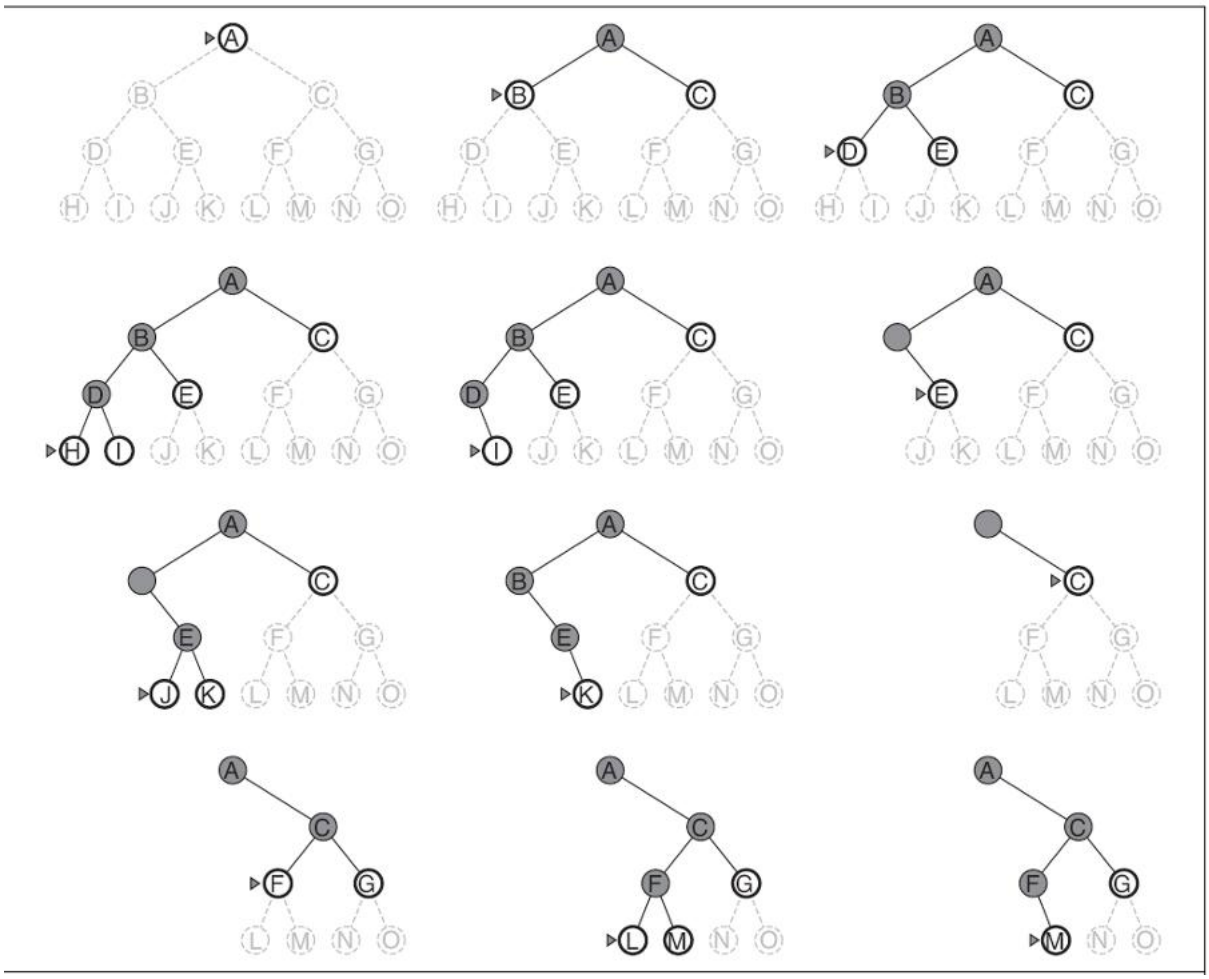
BFS:



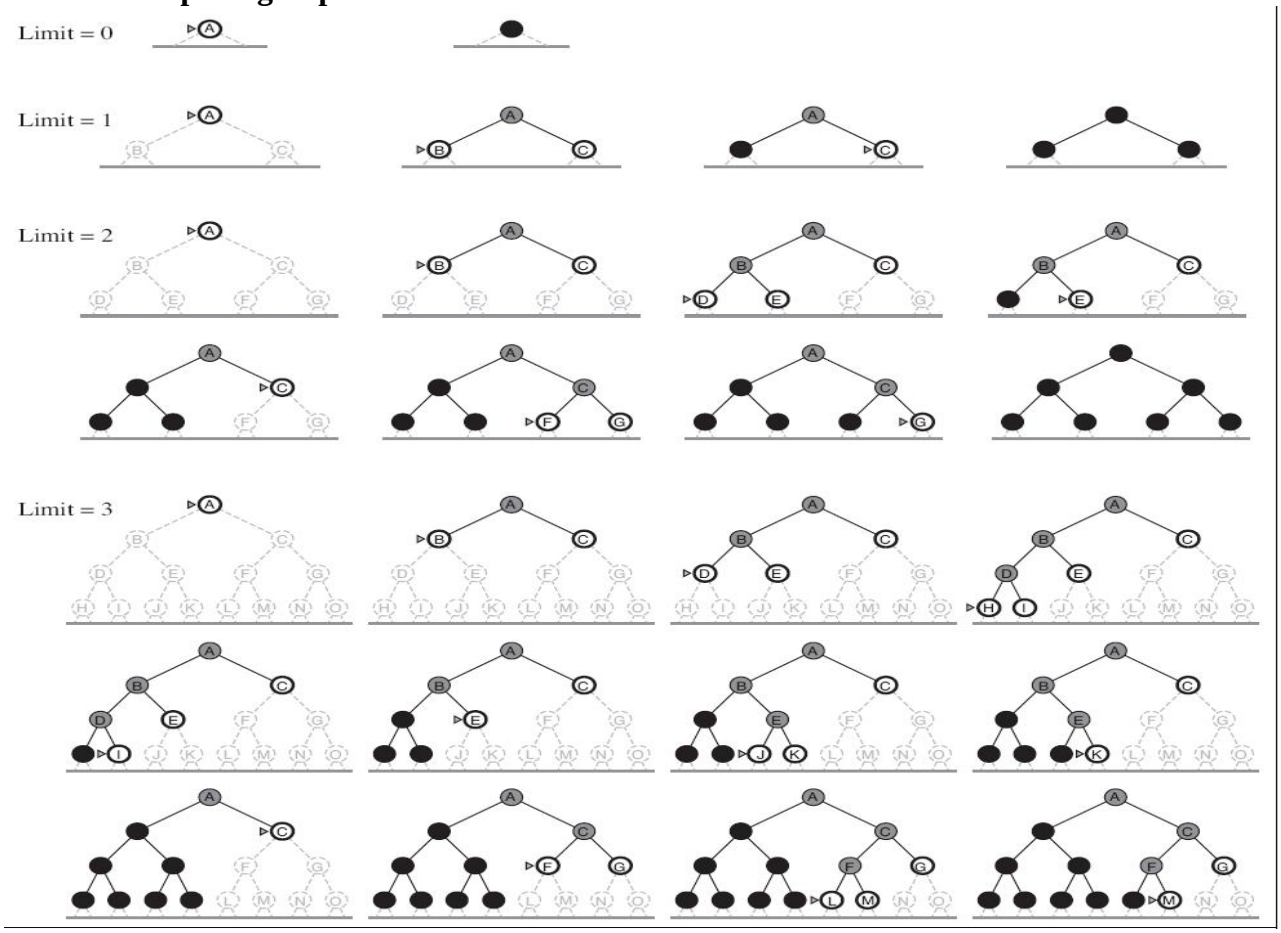
Uniform cost search:



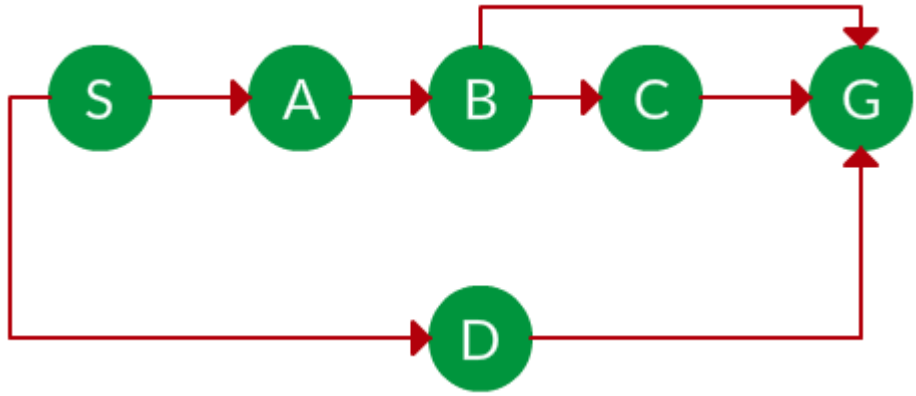
DFS:



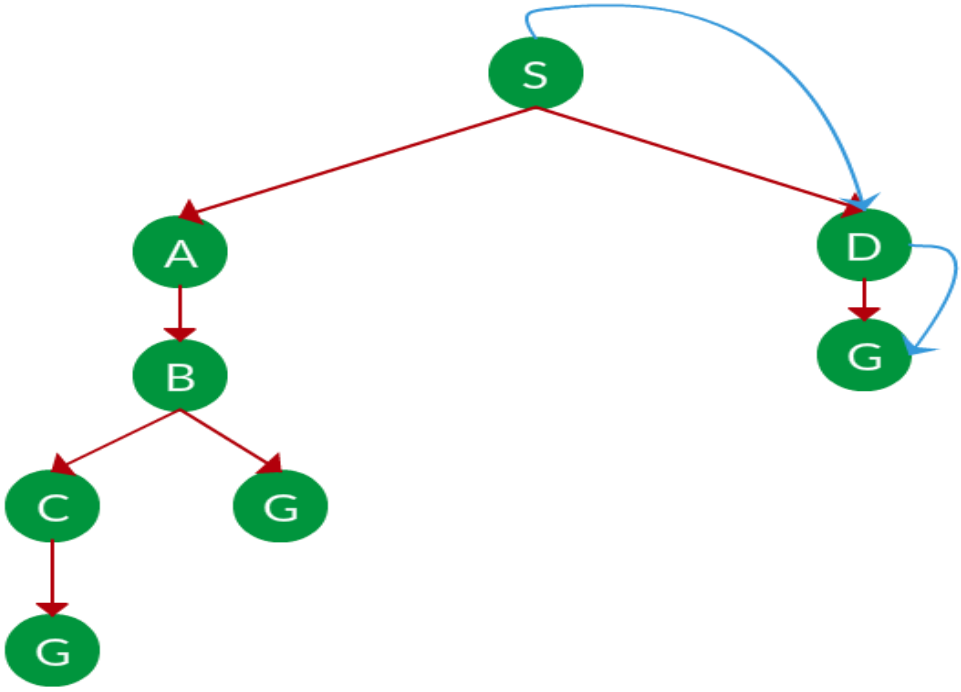
Iterative Deepening Depth First Search:



Question. Which solution would BFS find to move from node S to node G if run on the graph below?

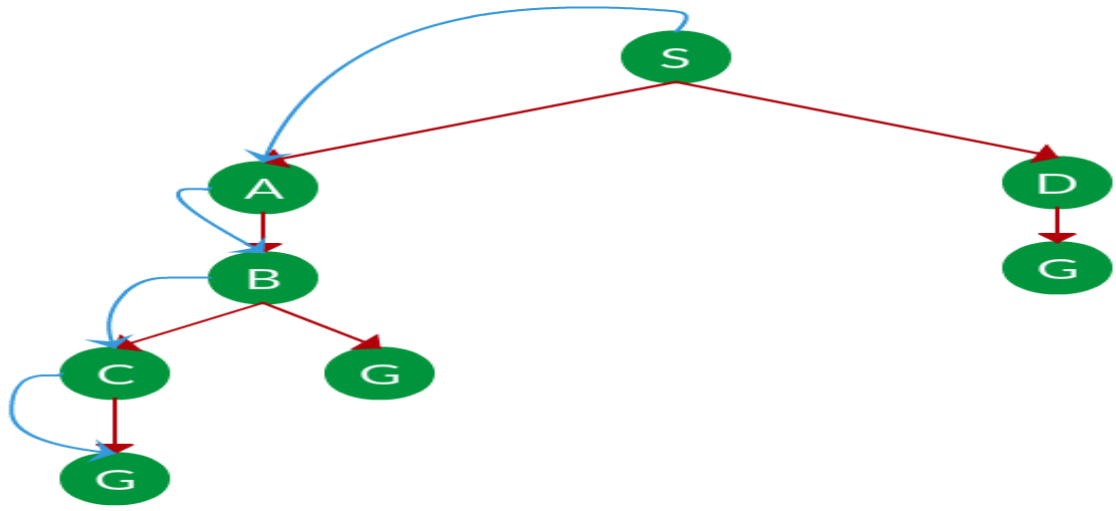


Solution.



Path: S -> D -> G

Question. Which solution would DFS find to move from node S to node G if run on the graph above?



Path: S -> A -> B -> C -> G

Comparing uninformed search strategies

Figure 3.21 compares search strategies in terms of the four evaluation criteria

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.

Artificial Intelligence
UNIT-II
INFORMED (HEURISTIC) SEARCH STRATEGIES

- An **informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.
- Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

1.) Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms.

With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e. f(n)= g(n).

1. h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

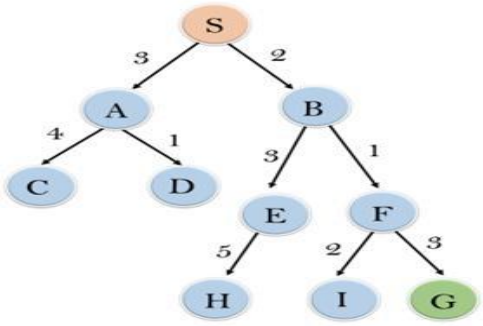
Advantages:

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.

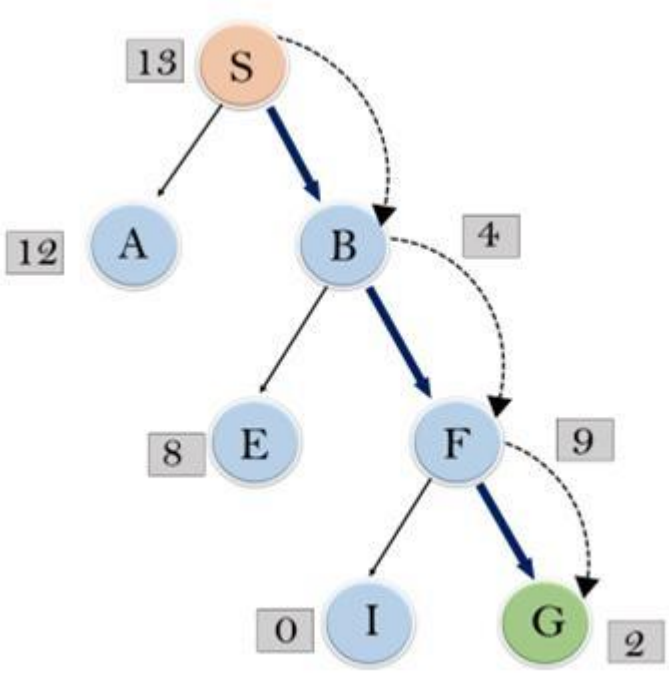
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0



Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F] : Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

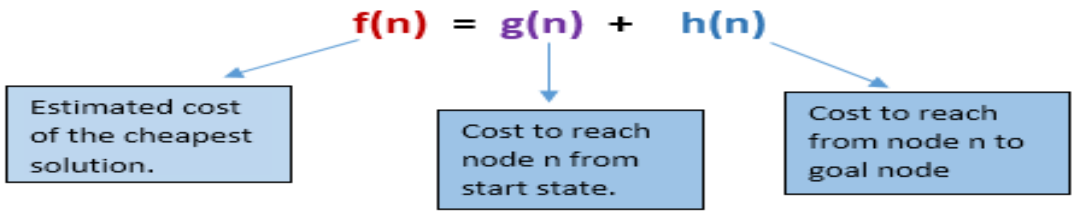
Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.**Space Complexity:** The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space. **Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.**Optimal:** Greedy best first search algorithm is not optimal.

2.) A* Search Algorithm:

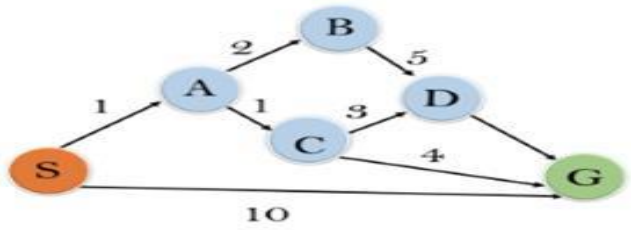
A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n).

It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function.

This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

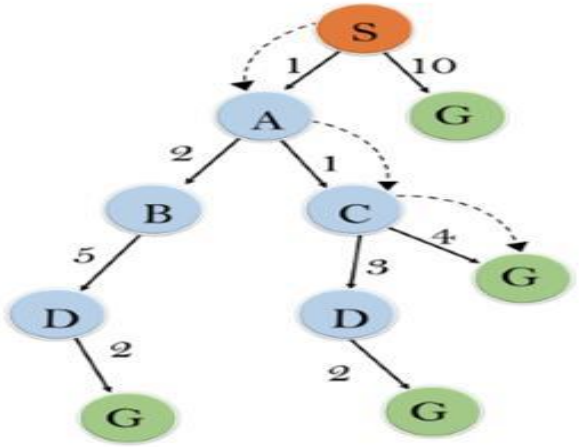


Algorithm of A* search:
Step1: Place the starting node in the OPEN list.
Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
Step 6: Return to **Step 2**.
Advantages:
A* search algorithm is the best algorithm than other search algorithms.
A* search algorithm is optimal and complete.
This algorithm can solve very complex problems.
Disadvantages:
It does not always produce the shortest path as it mostly based on heuristics and approximation.
A* search algorithm has some complexity issues.
The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.
Example:



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}
Iteration1: {(S--> A, 4), (S-->G, 10)}
Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
Iteration 4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6.
Complete: A* algorithm is complete
Optimal: A* search algorithm is optimal if it follows below two conditions:
Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
Consistency: Second required condition is consistency for only A* graph-search.
If the heuristic function is admissible, then A* tree search will always find the least cost path.
Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.
Space Complexity: The space complexity of A* search algorithm is O(b^d)

Memory bounded Heuristic search or RBFS (Recursive Best First Search):

- **Recursive best-first search** (RBFS) is RECURSIVE a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space.

- RBFS replaces the f-value of each node along the path with a **backed-up value**—the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time.

An Example: Figure 3.27 shows how RBFS reaches Goal node Bucharest.

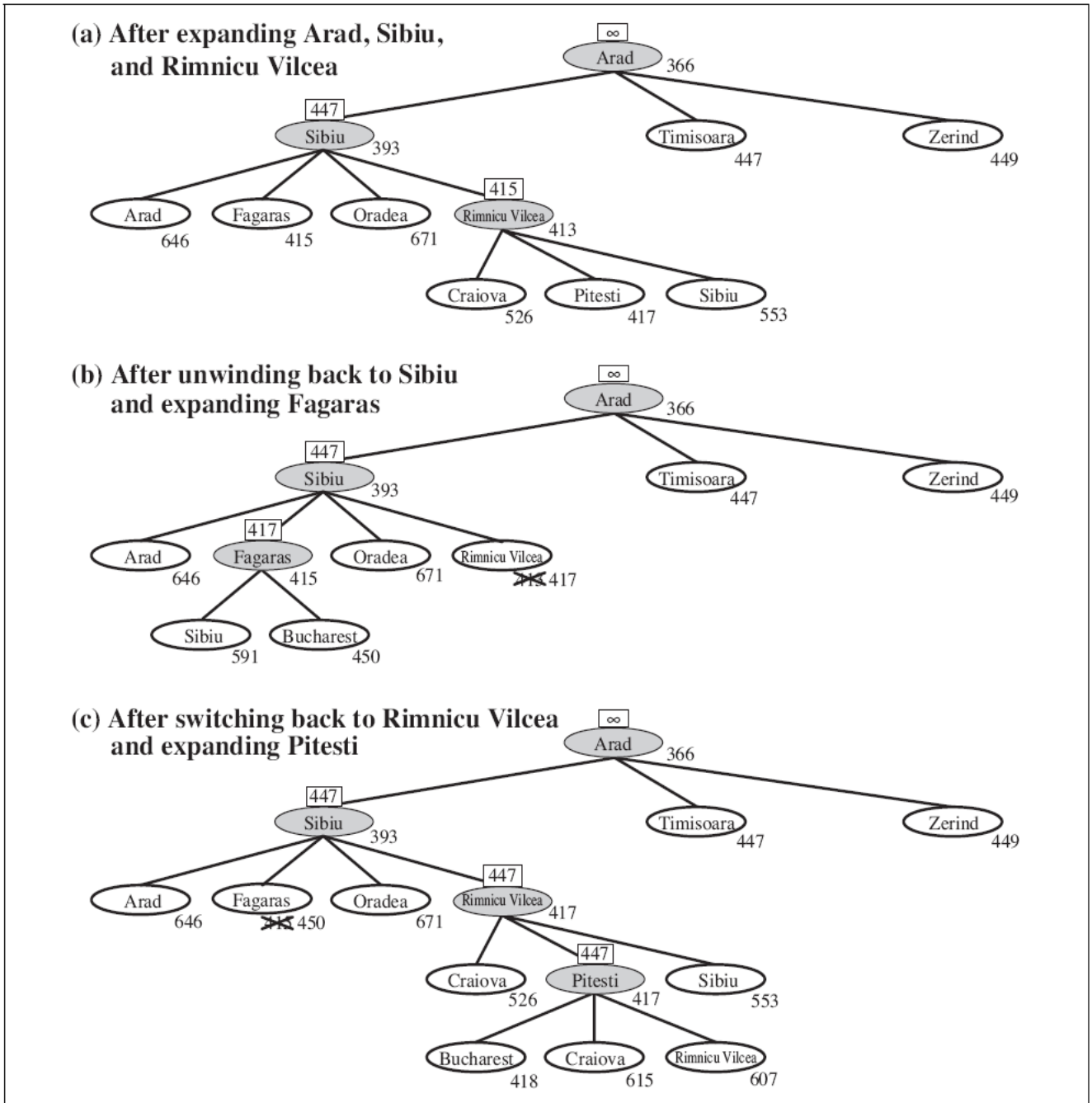


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Heuristic Functions in Artificial Intelligence:

Heuristics function: Heuristic is **a function which is used in Informed Search, and it finds the most promising path**. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal

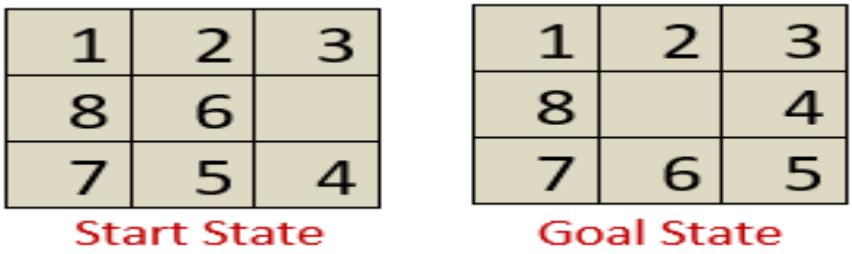
A good heuristic function is determined by its efficiency. More is the information about the problem, more is the processing time.

Properties of a Heuristic search Algorithm

Use of heuristic function in a heuristic search algorithm leads to following properties of a heuristic search algorithm:

- **Admissible Condition:** An algorithm is said to be admissible, if it returns an optimal solution. If a heuristic returns an optimal solution then it is said to be an admissible heuristic.
- **Completeness:** An algorithm is said to be complete, if it terminates with a solution (if the solution exists).
- **Dominance Property:** If there are two admissible heuristic algorithms **A1** and **A2** having **h1** and **h2** heuristic functions, then **A1** is said to dominate **A2** if **h1** is better than **h2** for all the values of node **n.**
- **Optimality Property:** If an algorithm is **complete, admissible**, and **dominating** other algorithms, it will be the best one and will definitely give an optimal solution.

An Example:



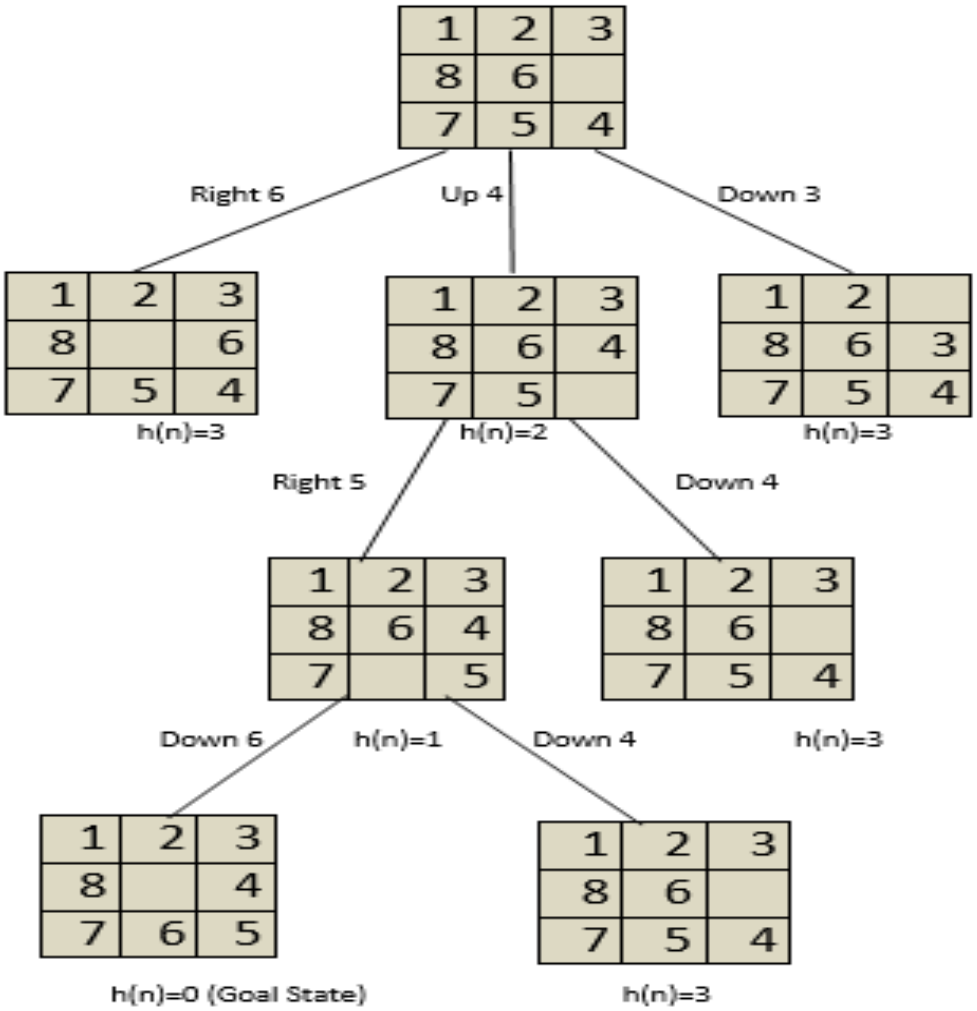
A heuristic function for the 8-puzzle problem is defined below:

h(n)=Number of tiles out of position.

So, there is total of three tiles out of position i.e., 6,5 and 4. Do not count the empty tile present in the goal state). i.e. h(n)=3. Now, we require to minimize the value of **h(n) =0.**

We can construct a state-space tree to minimize the h(n) value to 0, as shown below

The Heuristic information can be related to the **nature of the state, cost of transforming from one state to another, goal node characterstics,** etc., which is expressed as a heuristic function.



LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS:

- If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all.
- **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node.

- Although local search algorithms are not systematic, they have two key advantages:
  - they use very little memory—usually a constant amount; and
  - they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

Local search algorithms are useful for solving pure **op timization problems**, in which the aim is to find the best state according to an **objective function**.

1) **Hill-Climbing Search:**

- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance travelled by the salesman.

- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

The state-space landscape is a graphical representation of the hill-climbing algorithm which is shown in Figure 4.1.

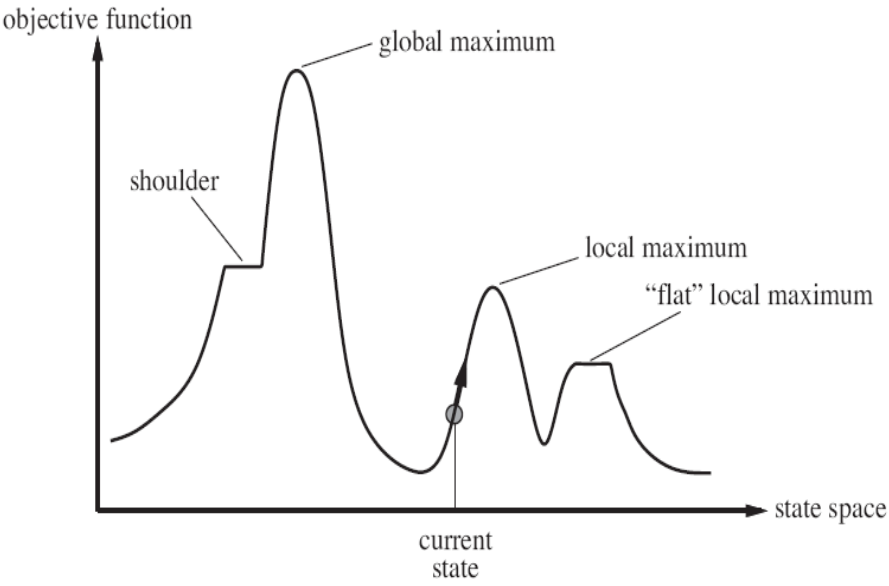


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
Current state: It is a state in a landscape diagram where an agent is currently present.
Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.
Shoulder: It is a plateau region which has an uphill edge.

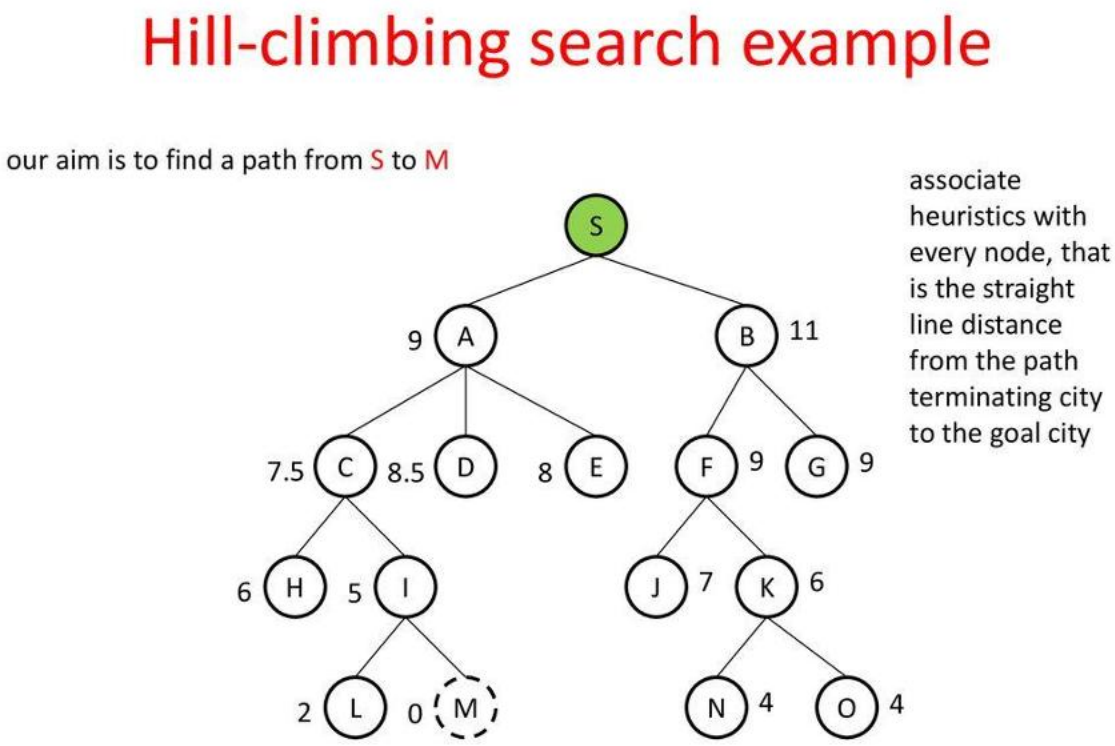
Algorithm for Simple Hill Climbing:

Step 1: **Evaluate the initial state, if it is goal state then return success and Stop.**

- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.

- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
    a. If it is goal state, then return success and quit.
    b. Else if it is better than the current state then assign new state as a current state.
    c. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

Example:



Path: S-A-C-I-M

2) Simulated Annealing:

- A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum.
- And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient.
- **Simulated Annealing** is an algorithm which yields both efficiency and completeness.
- In mechanical term **Annealing** is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state.

- The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path.
- Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

3) Genetic Algorithms:

A **genetic algorithm** (or **GA**) is a variant of stochastic beam search in which are generated by combining *two* parent states rather than by modifying a single state.

Example: 8-queens problem

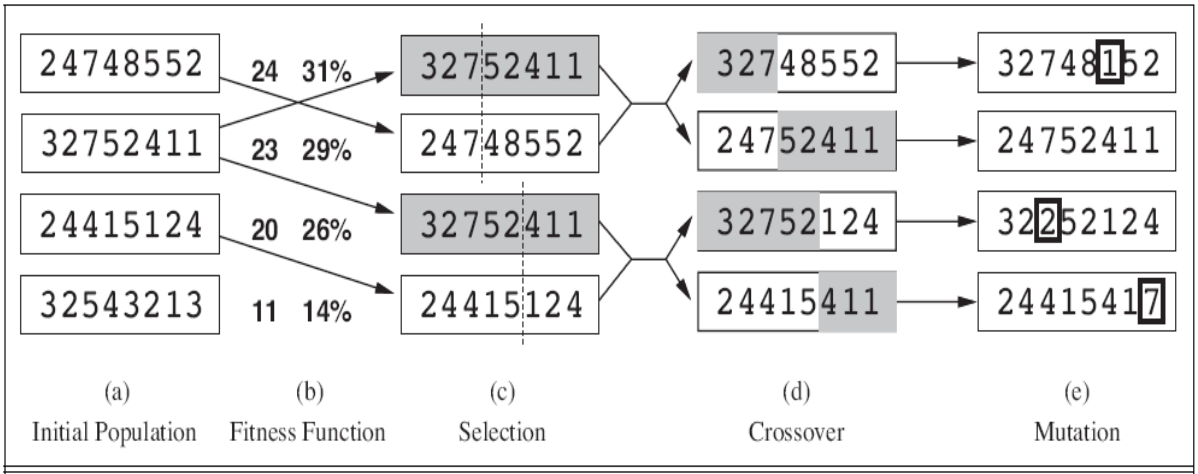


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Graphical Illustration:

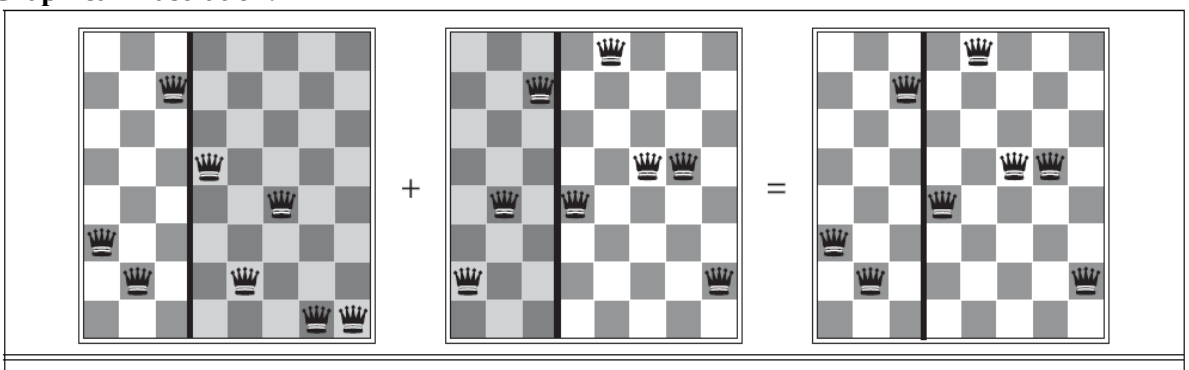


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

GAs begin with a set of k randomly generated states, called the **population**. Each state, or **individual**, is represented as a string over a finite alphabet.

The production of the next generation of states is shown in Figure 4.6(b)–(e). In (b), Feach state is rated by the objective function, or (in GA terminology) the **fitness function**. A fitness function should return higher values for better states.

In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all.4 for each pair to be mated, a **crossover** point is chosen randomly from the positions in the string. In Figure 4.6, the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.

In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent.

Finally, in (e), each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring.

The algorithm is shown in Figure 4.8

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
      new_population ← empty set
      for i = 1 to SIZE(population) do
          x ← RANDOM-SELECTION(population, FITNESS-FN)
          y ← RANDOM-SELECTION(population, FITNESS-FN)
          child ← REPRODUCE(x, y)
          if (small random probability) then child ← MUTATE(child)
          add child to new_population
      population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

Constraint Satisfaction Problem:

Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.

A constraint satisfaction problem consists of three components, X,D, and C:
X is a set of variables, {X1, . . . ,Xn}.
D is a set of domains, {D1, . . . ,Dn}, one for each variable.
C is a set of constraints that specify allowable combinations of values.
Each domain Di consists of a set of allowable values, {v1,…, vk} for variable Xi. Each constraint Ci consists of a pair _scope, rel _, where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

Example Problem: Map Coloring

Consider a map of Australia showing each of its states and territories (Figure 6.1(a)).

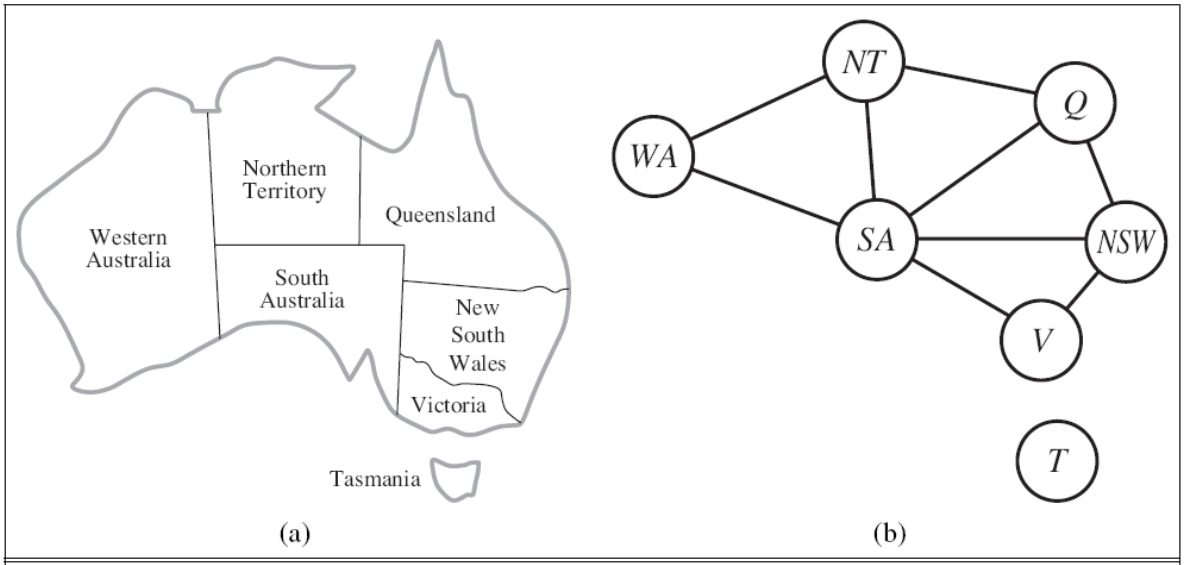


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions.

$$X = \{WA, NT, Q, NSW, V, SA, T\} .$$

The domain of each variable is the set $D_i = \{red, green, blue\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,$$
$$WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\} .$$

Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} .$$

There are many possible solutions to this problem, such as

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

A constraint graph is shown in Figure 6.1(b).

Cryptarithmetic:

Another example is provided by **cryptarithmetic** puzzles. (See Figure 6.2(a).) Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 6.2(a), this would be represented as the global constraint Alldiff (F, T, U, W, R, O). The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

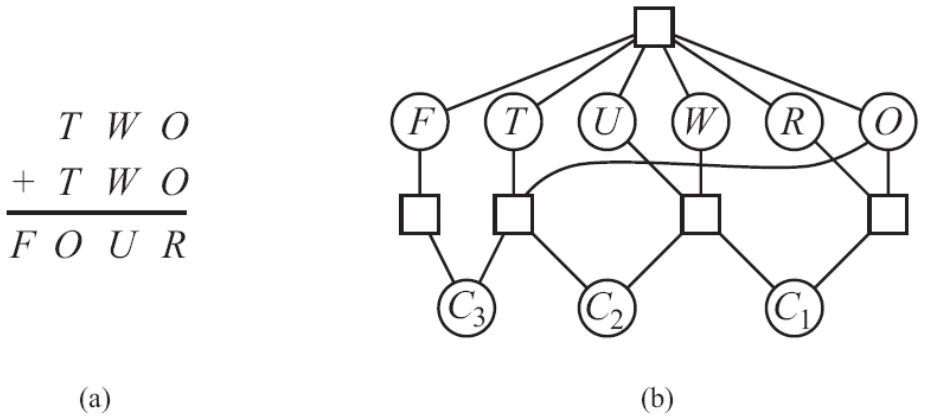


Figure 6.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns.

Where C10, C100, and C1000 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column.

These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b).

A hyper graph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints.

Some more examples:

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Greedy Best First Search:

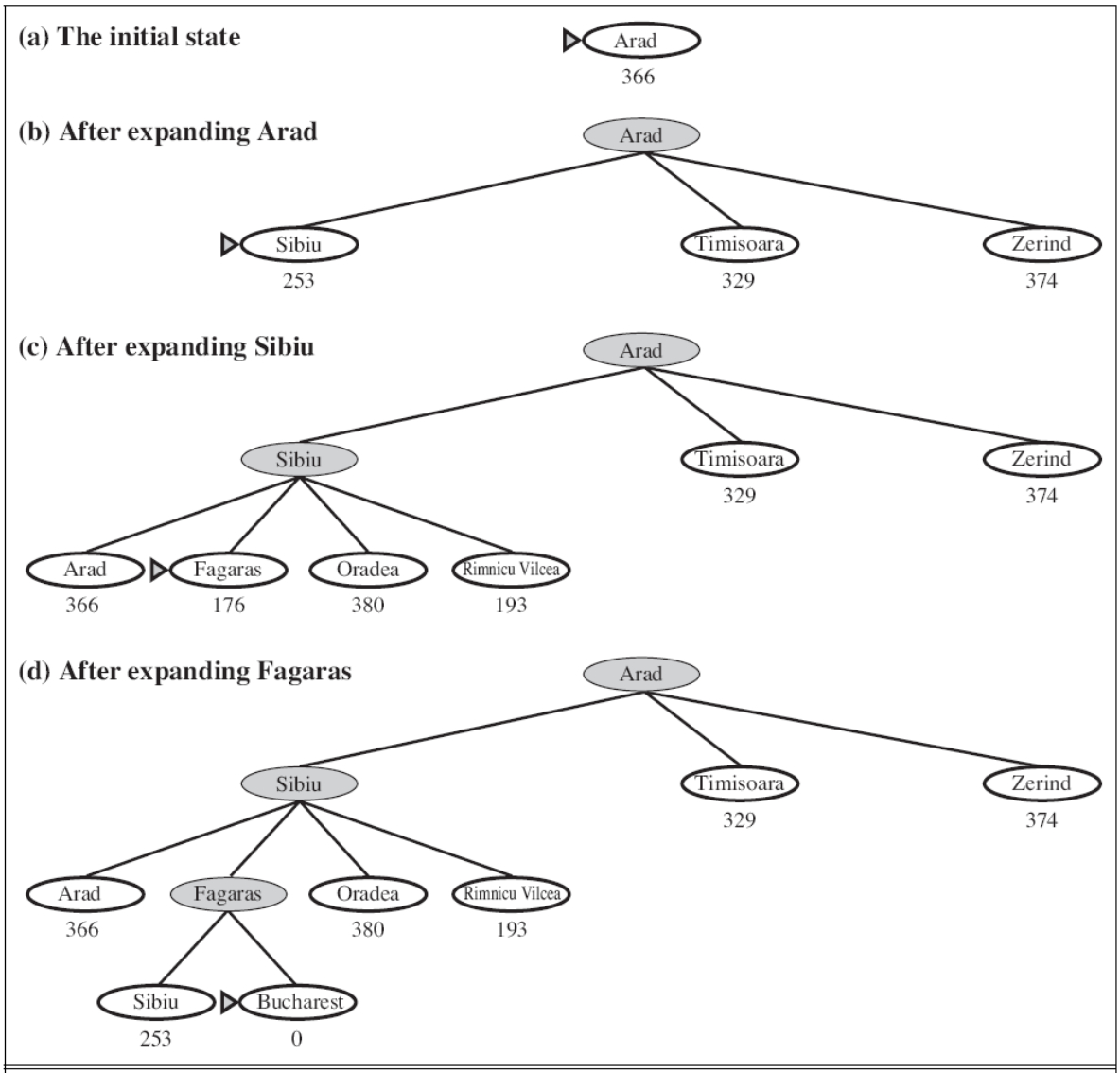


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* Search:

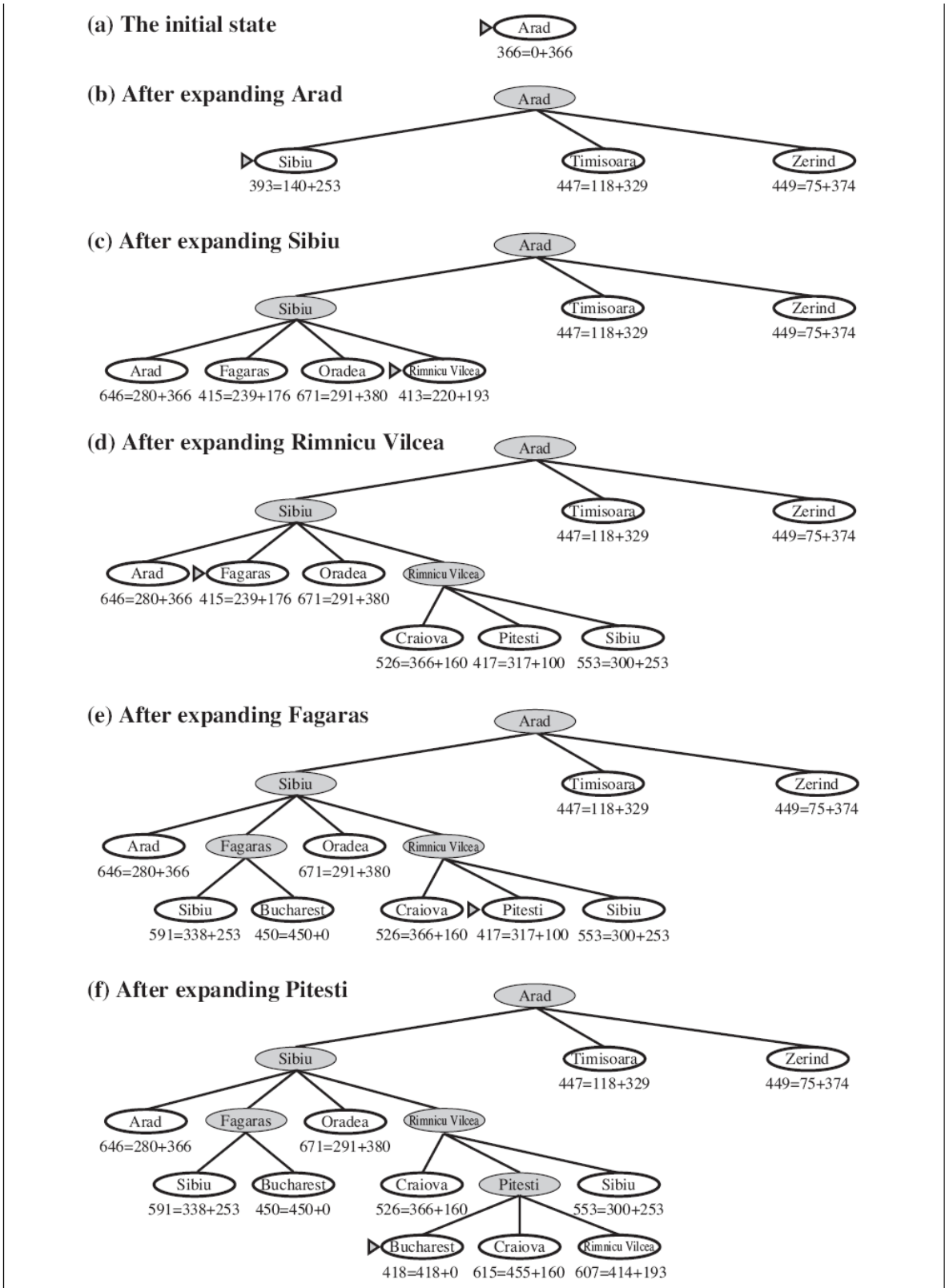


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

RBFS:

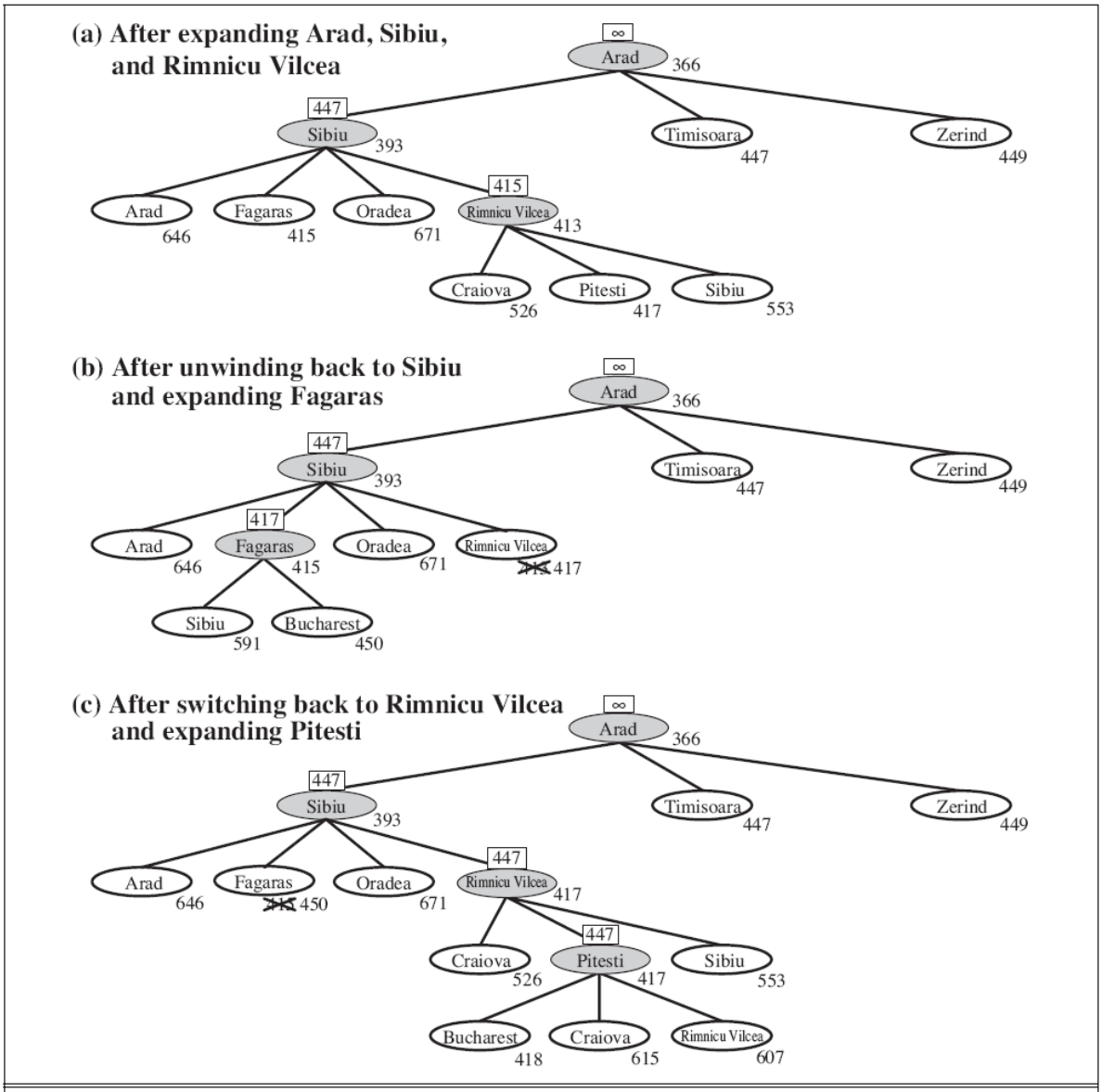


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Unit-III -Artificial Intelligence

Knowledge Reasoning and Inference (Chapter 7,8 and 9 from 3rd Edition)

- The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**.
- Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world.
- Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.
- **Inference**—that is, deriving new sentences from old.
- The agent maintains a knowledge base, KB, which may initially contain some **background knowledge**.
- The agent maintains a knowledge base, KB, may initially contain some **background knowledge**.

The psuedocode for generic knowledge based agents is shown in Figure 7.1

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

THE WUMPUS WORLD:

- The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow.
- Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in).
- The only mitigating feature of this bleak environment is the possibility of finding a heap of gold.

A typical wumpus world is shown in Figure 7.2

The PEAS description for Wumpus world are given by

• **Performance measure**: +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.

• **Environment**: A 4×4 grid of rooms. The agent always starts in the square labelled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly.

• **Actuators**:

- The agent can move *Forward*, *TurnLeft* by 90◦, or *TurnRight* by 90◦. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. If an agent tries to move forward and bumps into a wall, then the agent does not move.
- The action *Grab* can be used to pick up the gold if it is in the same square as the agent.
- The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing.
- Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].

• **Sensors**: The agent has five sensors, each of which gives a single bit of information: – In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
– In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
– In the square where the gold is, the agent will perceive a *Glitter*.
– When an agent walks into a wall, it will perceive a *Bump*.
– When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

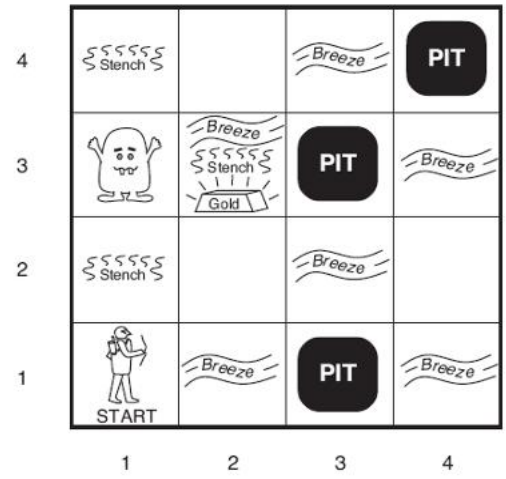


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

- The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**.
- Each such symbol stands for a proposition that can be true or false.
- We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, W1,3 and North.

Basics of Logic:
The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, $W_{1,3}$ and North.

There are two proposition symbols with fixed meanings: True is the always-true proposition and False is the always-false proposition. **Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives**. There are five connectives in common use:

1) ¬ (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).

2) $\wedge$ (and). A sentence whose main connective is $\wedge$, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The $\wedge$ looks like an "A" for "And.")
3) $\vee$ (or). A sentence using $\vee$, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction** of the **disjuncts** ($W_{1,3} \wedge P_{3,1}$) and $W_{2,2}$. (Historically, the $\vee$ comes from the Latin "vel," which means "or." For most people, it is easier to remember $\vee$ as an upside-down $\wedge$.)
4) $\Rightarrow$ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is ($W_{1,3} \wedge P_{3,1}$), and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as $\supset$ or $\rightarrow$.
5) $\Leftrightarrow$ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**. Some other books write this as $\equiv$.

The grammar for propositional logic is shown in Figure 7.7

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	*True* | *False* | *P* | *Q* | *R* | ...
ComplexSentence	→	(*Sentence*) | [*Sentence*]
	|	¬ *Sentence*
	|	*Sentence* ∧ *Sentence*
	|	*Sentence* ∨ *Sentence*
	|	*Sentence* ⇒ *Sentence*
	|	*Sentence* ⇔ *Sentence*
OPERATOR PRECEDENCE	:	¬, ∧, ∨, ⇒, ⇔

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Semantics:

The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the **truth value**—true or false—for every proposition symbol.

For example, if the sentences in the knowledge base make use of the proposition symbols P1,2, P2,2, and P3,1, then one possible model is m1 = {P1,2 =false, P2,2 =false, P3,1 =true}.

The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model.

Atomic sentences are easy:

• True is true in every model and False is false in every model.

• The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m1 given earlier, P1,2 is false.

For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here "iff" means "if and only if"):

· $\neg$P is true iff P is false in m.

· P $\wedge$ Q is true iff both P and Q are true in m.

· P $\vee$ Q is true iff either P or Q is true in m.

· P $\Rightarrow$ Q is true unless P is true and Q is false in m.

· P $\Leftrightarrow$ Q is true iff P and Q are both true or both false in m.

The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

Logical Equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as α ≡ β. For example, we can easily show (using truth tables) that P ∧ Q and Q ∧ P are logically equivalent; other equivalences are shown in Figure 7.11.

An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other: α ≡ β if and only if α |= β and β |= α.

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

Validity: A sentence is valid if it is true in *all* models. For example, the sentence P ∨¬P is valid. Valid sentences are also known as **tautologies**

Satisfiability: A sentence is satisfiable if it is true in, or satisfied by, *some* model.

Validity and satisfiability are of course connected: α is valid iff ¬α is unsatisfiable; contrapositively, α is satisfiable iff ¬α is not valid. We also have the following useful result: α |= β *if and only if the sentence* (α ∧ ¬β) *is unsatisfiable.*

Sound: An inference algorithm that derives only entailed sentences is called **sound** or **truth preserving**. Soundness is a highly desirable property.

Inference Rules:

Modus Ponens: The Modus Ponens is written

$$\frac{\alpha \Rightarrow \beta, \qquad \alpha}{\beta}$$

The notation means that, whenever any sentences of the form α ⇒ β and α are given, then the sentence β can be inferred. For example, if (WumpusAhead ∧WumpusAlive) ⇒ Shoot and (WumpusAhead ∧ WumpusAlive) are given, then Shoot can be inferred.

And-Eliminations: Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}$$

For example, from (WumpusAhead ∧ WumpusAlive), WumpusAlive can be inferred.

Conjunctive normal form: *every sentence of propositional logic is logically equivalent to a conjunction of clauses.* A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF.**

Example: We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence B1,1 ⇔ (P1,2 ∨ P2,1) into CNF. The steps are as follows:

1. Eliminate ⇔, replacing α ⇔ β with (α ⇒ β) ∧ (β ⇒ α).

(B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .

2. Eliminate ⇒, replacing α ⇒ β with ¬α ∨ β:

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬(P1,2 ∨ P2,1) ∨ B1,1) .

3. CNF requires ¬ to appear only in literals, so we "move ¬ inwards" by repeated application of the following equivalences from Figure 7.11:

¬ (¬α) ≡ α (double-negation elimination)

¬ (α ∧ β) ≡ (¬α ∨ ¬β) (De Morgan)

¬ (α ∨ β) ≡ (¬α ∧ ¬β) (De Morgan)

In the example, we require just one application of the last rule:

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ ((¬P1,2 ∧ ¬P2,1) ∨ B1,1) .

4. Now we have a sentence containing nested ∧ and ∨ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing ∨ over ∧ wherever possible.

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬P1,2 ∨ B1,1) ∧ (¬P2,1 ∨ B1,1) .

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

Resolution:

Unit Resolution: The unit resolution inference rule is written

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k},$$

where each l is a literal and l_i and m are **complementary literals** (i.e., one is the negation of the other). Thus, the CLAUSE unit resolution rule takes a **clause**—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of UNIT CLAUSE one literal, also known as a **unit clause**.

Full Resolution: The unit resolution rule can be generalized to the full **resolution** rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n},$$

where l_i and m_j are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals.

First Order Logic:

Syntax and Semantics of First Order Logic:

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The semantics must relate sentences to models in order to determine truth. For this to happen, we need an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.

The syntax of the first order logic shown in Figure 8.3.

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
& \mid & \neg\ \mathit{Sentence} \\
& \mid & \mathit{Sentence} \wedge \mathit{Sentence} \\
& \mid & \mathit{Sentence} \vee \mathit{Sentence} \\
& \mid & \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
& \mid & \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
& \mid & \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
\\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term}, \ldots) \\
& \mid & \mathit{Constant} \\
& \mid & \mathit{Variable} \\
\\
\mathit{Quantifier} & \rightarrow & \forall \mid \exists \\
\mathit{Constant} & \rightarrow & A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

Term: A **term** is a logical expression that refers TERM to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg.

Atomic Sentence: An **atomic sentence** (or **atom** for short) is formed from a predicate symbol optionally followed by a ATOM parenthesized list of terms, such as Brother (Richard , John). This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.6 Atomic sentences can have complex terms as arguments. Thus, Married(Father (Richard),Mother (John)) states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).

*An atomic sentence is **true** in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

Complex Sentences: We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus.

Ex:

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \wedge Brother(John, Richard)$$
$$King(Richard) \vee King(John)$$
$$\neg King(Richard) \Rightarrow King(John)\ .$$

Qunatifiers: Quantifiers express properties of entire collection of objects, instead of enumerating the objects by name. First-order logic contains two standard quantifiers, called *universal* and *existential*. Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as "Squares neighboring the wumpus are smelly" and "All kings are persons" are the bread and butter of first-order logic. "All kings are persons," is written in first-order logic as

$$\forall x \;\; King(x) \;\Rightarrow\; Person(x)$$

Existential quantification ($\exists$)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

$$\exists x \;\; Crown(x) \wedge OnHead(x, John) \; .$$

$\exists x$ is pronounced "There exists an x such that ..." or "For some x ...".

Intuitively, the sentence $\exists x \;\; P$ says that P is true for at least one object x. More precisely, $\exists x \;\; P$ is true in a given model if P is true in *at least one* extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

Richard the Lionheart is a crown $\wedge$ Richard the Lionheart is on John's head;
King John is a crown $\wedge$ King John is on John's head;
Richard's left leg is a crown $\wedge$ Richard's left leg is on John's head;
John's left leg is a crown $\wedge$ John's left leg is on John's head;
The crown is a crown $\wedge$ the crown is on John's head.

Connections between $\forall$ and $\exists$

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x \;\; \neg Likes(x, Parsnips) \quad \text{is equivalent to} \quad \neg\exists x \;\; Likes(x, Parsnips) \; .$$

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

$$\forall x \;\; Likes(x, IceCream) \quad \text{is equivalent to} \quad \neg\exists x \;\; \neg Likes(x, IceCream) \; .$$

Because $\forall$ is really a conjunction over the universe of objects and $\exists$ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{lll} \forall x \;\; \neg P \equiv \neg\exists x \;\; P & & \neg(P \vee Q) \equiv \neg P \wedge \neg Q \\ \neg\forall x \;\; P \equiv \exists x \;\; \neg P & & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\ \forall x \;\; P \equiv \neg\exists x \;\; \neg P & & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\ \exists x \;\; P \equiv \neg\forall x \;\; \neg P & & P \vee Q \equiv \neg(\neg P \wedge \neg Q) \; . \end{array}$$

Thus, we do not really need both $\forall$ and $\exists$, just as we do not really need both $\wedge$ and $\vee$. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

Equality: We can use the **equality symbol** to signify that two terms refer to the same object. For example, Father (John) = Henry says that the object referred to by Father (John) and the object referred to by Henry are the same.

Using the first order logic:

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a person, and all kings are persons:

$$\text{TELL}(KB,\ King(John))\ .$$
$$\text{TELL}(KB,\ Person(Richard))\ .$$
$$\text{TELL}(KB,\ \forall x\ \ King(x)\ \Rightarrow\ Person(x))\ .$$

We can ask questions of the knowledge base using ASK. For example,

$$\text{ASK}(KB,\ King(John))$$

returns true.

The kinship domain:

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent." Clearly, the objects in our domain are people. We have two unary predicates, Male and Female. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother , Sister , Child , Daughter, Son, Spouse, Wife, Husband, Grandparent , Grandchild , Cousin, Aunt, and Uncle. We use functions for Mother and Father , because every person has exactly one of each of these (at least according to nature's design).

We can go through each function and predicate, writing down what w of the other symbols. For example, one's mother is one's female parent:

$$\forall m, c\ \ Mother(c) = m\ \Leftrightarrow\ Female(m) \wedge Parent(m, c)\ .$$

One's husband is one's male spouse:

$$\forall w, h\ \ Husband(h, w)\ \Leftrightarrow\ Male(h) \wedge Spouse(h, w)\ .$$

Male and female are disjoint categories:

$$\forall x\ \ Male(x)\ \Leftrightarrow\ \neg Female(x)\ .$$

Parent and child are inverse relations:

$$\forall p, c\ \ Parent(p, c)\ \Leftrightarrow\ Child(c, p)\ .$$

A grandparent is a parent of one's parent:

$$\forall g, c\ \ Grandparent(g, c)\ \Leftrightarrow\ \exists p\ \ Parent(g, p) \wedge Parent(p, c)\ .$$

A sibling is another child of one's parents:

$$\forall x, y\ \ Sibling(x, y)\ \Leftrightarrow\ x \neq y \wedge \exists p\ \ Parent(p, x) \wedge Parent(p, y)\ .$$

Axioms:

1. The only sets are the empty set and those made by adjoining something to a set:

$$\forall s \;\; Set(s) \;\Leftrightarrow\; (s = \{\}) \vee (\exists x, s_2 \;\; Set(s_2) \wedge s = \{x|s_2\}) \; .$$

2. The empty set has no elements adjoined into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element:

$$\neg\exists x, s \;\; \{x|s\} = \{\} \; .$$

3. Adjoining an element already in the set has no effect:

$$\forall x, s \;\; x \in s \;\Leftrightarrow\; s = \{x|s\} \; .$$

4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined with some element y, where either y is the same as x or x is a member of s_2:

$$\forall x, s \;\; x \in s \;\Leftrightarrow\; \exists y, s_2 \;\; (s = \{y|s_2\} \wedge (x = y \vee x \in s_2)) \; .$$

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:

$$\forall s_1, s_2 \;\; s_1 \subseteq s_2 \;\Leftrightarrow\; (\forall x \;\; x \in s_1 \;\Rightarrow\; x \in s_2) \; .$$

6. Two sets are equal if and only if each is a subset of the other:

$$\forall s_1, s_2 \;\; (s_1 = s_2) \;\Leftrightarrow\; (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1) \; .$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cap s_2) \;\Leftrightarrow\; (x \in s_1 \wedge x \in s_2) \; .$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cup s_2) \;\Leftrightarrow\; (x \in s_1 \vee x \in s_2) \; .$$

Propositional vs First order

Inference rules for quantifiers:

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x \quad King(x) \wedge Greedy(x) \implies Evil(x)$$

Then it seems quite permissible to infer any of the following sentences:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$
$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$$
$$\vdots$$

The rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.[1] To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)} \; .$$

For example, from the sentence

$$\exists x \;\; Crown(x) \wedge OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

Unification and Lifting:

Generalized Modus Ponens: For atomic sentences pi, pi, and q, where there is a substitution θ such that

$$\text{such that } \text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i), \text{ for all } i,$$

$$\frac{p_1', \;\; p_2', \;\; \ldots, \;\; p_n', \;\; (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

There are $n+1$ premises to this rule: the n atomic sentences p_i' and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

p_1' is $King(John)$	p_1 is $King(x)$
p_2' is $Greedy(y)$	p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$	q is $Evil(x)$
$\text{SUBST}(\theta, q)$ is $Evil(John)$.	

Generalized Modus Ponens is a **lifted** LIFTING version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.

Unification: This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \; .$$

$$\begin{aligned}
&\text{UNIFY}(Knows(John, x), \; Knows(John, Jane)) = \{x/Jane\} \\
&\text{UNIFY}(Knows(John, x), \; Knows(y, Bill)) = \{x/Bill, y/John\} \\
&\text{UNIFY}(Knows(John, x), \; Knows(y, Mother(y))) = \{y/John, x/Mother(John)\} \\
&\text{UNIFY}(Knows(John, x), \; Knows(x, Elizabeth)) = fail \; .
\end{aligned}$$

The unification algorithm shown in Figure 9.1

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Ex:
Given a sentence to be stored, it is possible to construct indices for *all possible* queries that unify with it. For the fact Employs(IBM ,Richard), the queries are
Employs(IBM ,Richard) Does IBM employ Richard?
Employs(x,Richard) Who employs Richard?
Employs(IBM , y) Whom does IBM employ?
Employs(x, y) Who employs whom?

Forward Chaining: Forward chaining idea is simple, start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses.

Definite clauses such as Situation ⇒ Response are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and forward chaining can be implemented very efficiently.

First-order definite clauses closely resemble propositional definite clauses (page 256): they are disjunctions of literals of which *exactly one is positive*. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Not every knowledge base can be converted into a set of definite clauses because of the single-positive-literal restriction, but many can. Consider the following problem:

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

"... it is a crime for an American to sell weapons to hostile nations":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) . \quad (9.3)$$

"Nono ... has some missiles." The sentence $\exists x\ Owns(Nono, x) \wedge Missile(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M_1:

$$Owns(Nono, M_1) \quad (9.4)$$
$$Missile(M_1) \quad (9.5)$$

"All of its missiles were sold to it by Colonel West":

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) . \quad (9.6)$$

We will also need to know that missiles are weapons:

$$Missile(x) \Rightarrow Weapon(x) \quad (9.7)$$

and we must know that an enemy of America counts as "hostile":

$$Enemy(x, America) \Rightarrow Hostile(x) . \quad (9.8)$$

"West, who is American ...":

$$American(West) . \quad (9.9)$$

"The country Nono, an enemy of America ...":

$$Enemy(Nono, America) . \quad (9.10)$$

The forward chaining algorithm and the proof tree generated shown in Figure 9.3 and Figure 9.4

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
      new ← { }
      for each rule in KB do
          (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
          for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                      for some p'1, ..., p'n in KB
              q' ← SUBST(θ, q)
              if q' does not unify with some sentence already in KB or new then
                  add q' to new
                  φ ← UNIFY(q', α)
                  if φ is not fail then return φ
      add new to KB
  return false
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

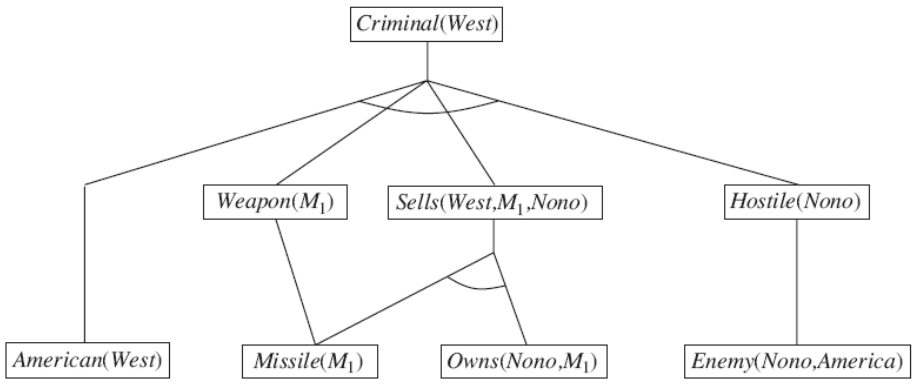


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Planning: (Chpater 11 from 2nd Edition)

- The task of coming up with a **sequence of actions that will achieve a goal** is called **planning.**
- Planning in AI is about **decision-making** actions performed by robots or computer programs to **achieve a specific goal**.
- Execution of the plan is about **choosing a sequence of tasks** with a high probability of **accomplishing a specific task**.

There are two types of planning environments:

- The environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called **classical planning** environments.
- In contrast, nonclassical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs.

The language of planning problems: The language of planning problems has three components.

1. **Representation of States.** Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals.
    a. **-EX:** At(Plane1, Melbourne) A At(Plane2, Sydney)
        i. might represent a state in the package delivery problem.
2. **Representation of Goals. A** goal is a partially specified state, represented as a conjunction of positive ground literals.
    a. **-Ex** Rich ∧ Famous
3. **Representation of Actions.** An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed. For example, an action for flying a plane from one location to another is:

$$\begin{aligned} &Action(Fly(p, from, \text{to}),\\ &\quad \text{PRECOND:} At(p, \text{from}) \text{ A } Plane(p) \text{ A } Airport(from) \wedge Airport(to)\\ &\quad \text{EFFECT:} \neg At(p, \text{from}) \text{ A } At(p, \text{to})) \end{aligned}$$

An action schema consists of three parts:

- The action name and parameter list-for example, Fly (p, from, to)-serves to identify the action.
- The **precondition** is a conjunction of function-free positive literals stating what must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.
- The **effect** is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in the state resulting from the action, whereas a negative literal $\neg P$ is asserted to be false.

The basic classical planners represents using the following two basic languages.

- **STRIPS Language (STanford Research Institute Problem Solver):**
    1. One of the most important restrictions in STRIPS Language is that literals be *function* ***free.***

2. With this restriction, we can be sure that any action schema for a given problem can be propositionalized- i.e, turned into a finite **collection of purely propositional action representations** with **no variables.**

- **ADL (Action Description Language):** ADL overcomes the drawbacks of STRIPS. It has more expressiveness and extensions compare to STRIPS.
  Example: the fly action could be written as

$$Action(Fly(p : Plane, from : Airport, to : Airport),$$
$$\text{PRECOND:} At(p, from) \wedge (from \neq to)$$
$$\text{EFFECT:} \neg At(p, from) \wedge At(p, to)).$$

The Comparison of STRIPS and ADL languages for representing planning problems is shown in Figure 11.1

STRIPS **Language**	ADL **Language**
Only positive literals in states: *Poor* ∧ *Unknown*	Positive and negative literals in states: $\neg$*Rich* ∧ $\neg$*Famous*
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect P ∧ $\neg Q$ means add P and delete Q.	Effect P ∧ $\neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q.
Only ground literals in goals: *Rich* ∧ *Famous*	Quantified variables in goals: $\exists x At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: *Rich* ∧ *Famous*	Goals allow conjunction and disjunction: $\neg$*Poor* ∧ (*Famous* ∨ *Smart*)
Effects are conjunctions.	Conditional effects allowed: **when** P: E means E is an effect only if P is satisfied.
No support for equality.	Equality predicate (x = y) is built in.
No support for types.	Variables can have types, as in (p : *Plane*).

Figure 11.1 Comparison of STRIPS and ADL languages for representing planning problems. In both cases, goals behave as the preconditions of an action with no parameters.

Example-1: (Air Cargo transport)

- The air cargo transport problem involves **loading** and **unloading cargo** onto and off of **planes** and **flying it** from place to place.
- The problem can be defined with **three actions**: Load, Unload, and Fly.
- The actions affect **two predicates: *In(c,* p)** means that cargo c is inside plane p, and ***At(x, a)*** means that object x (either plane or cargo) is at airport a.
- **Note:** that cargo is **not *At*** anywhere when it is *In* a plane, so ***At*** really means "available for use at a given location."

The problem is defined in pure **STRIPS Language** shown in Figure 11.2

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$
$\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$
$\wedge Airport(JFK) \wedge Airport(SFO))$
$Goal(At(C_1, JFK) \wedge At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $\neg At(c, a) \wedge In(c, p))$
$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $At(c, a) \wedge \neg In(c, p))$
$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to))$

Figure 11.2 A STRIPS problem involving transportation of air cargo between airports.

- **The following is a solution to the air cargo problem.**

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P2, SFO)].$$

Example-2: The spare tire problem

- Consider the problem of changing a flat tire.
- The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk.
- There are four actions: *removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle,* and *leaving the car* unattended overnight.
- We assume that the car is in a particularly bad neighborhood, so that the: effect of leaving it overnight is that the tires disappear.

The ADL description of the problem is shown in Figure 11.3.

$Init(At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
PRECOND: $At(Spare, Trunk)$
EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
PRECOND: $At(Flat, Axle)$
EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
PRECOND: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$
EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
$\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

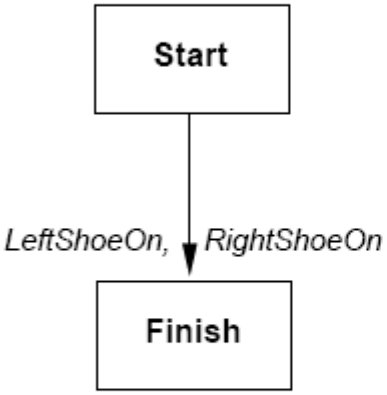
Figure 11.3 The simple spare tire problem.

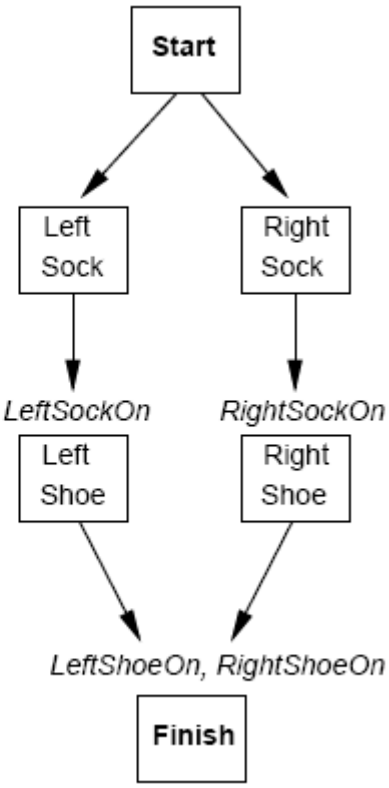
- **The following is a solution for spare tire problem:**
  [Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]

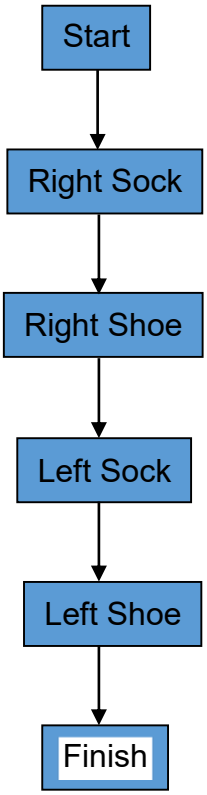
Partially Ordered Plan: A partially ordered Plan is a collection of steps:

- **Start step has the initial state description and its effect**

- **<u>Finish step</u> has the goal description as its precondition**
- **<u>Causal links</u> from outcome of one step to precondition of another step**
- **<u>Temporal ordering</u> between pairs of steps**
- **An open condition is a precondition of a step not yet causally linked**
- **A plan is <u>complete</u> iff every precondition is achieved**
- **A precondition is <u>*achieved*</u> iff it is the effect if an earlier step and no possibly intervening step undoes it**







IV. Uncertainty Handling

Agents almost never have access to the whole truth about their environment. Agents must, therefore, act under **uncertainty.**

Handling Uncertain Knowledge:

Let us try to write rules for dental diagnosis using first-order logic, so that we can see how the logical approach breaks down. Consider the following rule:

$$\forall p \;\; Symptom(p, Toothache) \Rightarrow Disease(p, Cavity) .$$

The problem is that this rule is wrong. Not all patients with toothaches have cavities; some of them have gum disease, an abscess, or one of several other problems:

$$\forall p \;\; Symptom(p, Toothache) \Rightarrow$$
$$Disease(p, Cavity) \, V \, Disease(p, GumDisease) \vee Disease(p, Abscess)$$

Unfortunately, in order to make the rule true, we have to add an almost unlimited list of possible causes. We co~~tlryd turning the rule into a causal rule:

$$\forall p \;\; Disease(p, Cavity) \Rightarrow Symptom(p, Toothache) .$$

Trying to use first-order logic to cope with a domain like medical diagnosis thus fails for three main reasons:

Laziness: It is too much work to list the complete set of antecedents or cons

Theoretical ignorance: Medical science has no complete theory for the domain.

Practical ignorance: Even if we know all the rules, we might be uncertain about a particular patient because not all the necessary tests have been or can be run.

Probability provides a way of **summarizing** *the uncertainty that comes from our laziness and ignorance.*

To make such choices, an agent must first have **preferences** between the different possible **outcomes** of the various plans.

Preferences, as expressed by utilities, are combined with probabilities in the general theory of rationa1 decisions called **decision theory:**

Decision theory = probability theory + utility theory.

The fundamental idea of decision theory is that *an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action.* This is called the principle of **Maximum Expected Utility (MEU).**

A typical decision theoretic agent has shown in Figure 13.1

```
function DT-AGENT(percept) returns an action
  persistent: belief_state, probabilistic beliefs about the current state of the world
              action, the agent's action

  update belief_state based on action and percept
  calculate outcome probabilities for actions,
      given action descriptions and current belief_state
  select action with highest expected utility
      given probabilities of outcomes and utility information
  return action
```

Figure 13.1 A decision-theoretic agent that selects rational actions.

Basic Probability Notation:

Degrees of belief are always applied to propositions. The basic element of the language is the random1 variable. The random variables are typically divided into three kinds.

Boolean random variables, such as *Cavity*, have the domain *(true, false).* We will often abbreviate a proposition such as *Cavity = true* simply by the lowercase name *cavity.* Similarly, *Cavity = false* would be abbreviated by ¬*cavity.*

Discrete random variables, which include Boolean random variables as a special case, take on values from a *countable* domain. For example, the domain of *Weather* might be *(sunny, rainy, cloudy, snow).* The values in the domain must be mutually exclusive and exhaustive. Where no confusion arises, we: will use, for example, *snow* as an abbreviation for *Weather = snow.*

Continuous random variables take on values from the: real numbers. The domain can be either the entire real line or some subset such as the interval [0, 1]. For example, the proposition X = 4.02 asserts that the random variable .*X* has the exact value 4.02. Propositions concerning continuous random variables can also be inequalities, such as *X* <= 4.02.

Atomic events: An atomic event is a complete specification of the state of the world about which the agent is uncertain.

Atomic events have some important properties:

They are mutually exclusive-at most one can actually be the case. For example, *cavity* ∧ *toothache* and *cavity* ∧ ¬*toothache* cannot both be the case.

The set of all possible atomic events is exhaustive-at least one must be the case. That is, the disjunction of all atomic events is logically equivalent to *true.*

Any particular atomic event entails the truth or falsehood of every proposition, whether simple or complex. For example, the atomic event *cavity* ∧ ¬*toothache* entails the truth of *cavity* and the falsehood of *cavity* ⇒ *toothache.*

Any proposition is logically equivalent to the disjunction of all atomic events that entail the truth of the proposition. For example, the proposition *cavity* is equivalent to disjunction of the atomic events *cavity* ∧*toothache* and ¬*cavity* ∨ ¬*toothache.*

Prior probability:

The **unconditional** or **prior probability** associated with a proposition *a* is the degree of belief accorded to it in the absence of any other information; it is written as *P (a).* For example, if the prior probability that I have a cavity is 0.1, then we would write *P (Cavity = true) = 0.1* or *P (cavity)* = 0.1.

Ex:
P (Weather = sunny) = 0.7
P (Weather = rain) = 0.2
P (Weather = cloudy) = 0.08
P (Weather = snow) = 0.02.
We may simply write

P (Weather) = (0.7, 0.2, 0.08, 0.02).

Conditional Probabilities:

Conditional probabilities can be defined in terms of unconditional probabilities. The defining equation is

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

which holds whenever $P(b) > 0$. This equation can also be written as

$$P(a \wedge b) = P(a|b)P(b)$$

The above equation is called product rule. We can also frame it as below.

$$P(a \wedge b) = P(b|a)P(a) .$$

The Axioms of Probability:

There are three axioms which are often called Kolmogorov's axioms.

1. All probabilities are between 0 and 1. For any proposition a,

$$0 \leq P(a) \leq 1$$

2. Necessarily true (i.e., valid) propositions have probability 1, and necessarily false (i.e., unsatisfiable) propositions have probability 0.

$$P(true) = 1 \qquad P(false) = 0 .$$

Next, we need an axiom that connects the probabilities of logically related propositions. The simplest way to do this is to define the probability of a disjunction as follows:

3. The probability of a disjunction is given by

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) .$$

Using the axioms of probability:

We can derive a variety of useful facts from the basic axioms. For example,

$$\begin{aligned} P(a \vee \neg a) &= P(a) + P(\neg a) - P(a \wedge \neg a) && \text{(by axiom 3 with } b = \neg a) \\ P(true) &= P(a) + P(\neg a) - P(false) && \text{(by logical equivalence)} \\ 1 &= P(a) + P(\neg a) && \text{(by axiom 2)} \\ P(\neg a) &= 1 - P(a) && \text{(by algebra).} \end{aligned}$$

INFERENCE USING FULL JOINT DISTRIBUTIONS:

We will use the full joint distribution as the "knowledge base" from which answers to all questions may be derived.

Example: Let a domain consisting of just the three Boolean variables *Toothache, Cavity,* and *Catch* (the dentist's nasty steel probe catches in my tooth). The full joint distribution is a 2 x 2 x 2 table as shown in Figure 13.3.

	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

Notice that the probabilities in the joint distribution sum to 1, as required by the axioms of probability. Now we can calculate the probability of any proposition given simple or complex.

Example1) the probability of finding cavity or toothache P (cavity ∨ toothache) can compute as
P(cavity ∨ toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28 .

Example 2) the probability of having cavity P(cavity) gives the unconditional or marginal probability of cavity
P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2.

Example 3) the probability of a cavity, given evidence of a toothache, as follows i.e.

$$P(cavity \mid toothache) = \frac{P(cavity \wedge toothache)}{P(toothache)}$$
$$= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6$$

Example 4) the probability of no cavity, given evidence of toothache, as follows i.e.

$$P(\neg cavity \mid toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)}$$
$$= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4$$

In the similar way, we can find compute the required probabilities.

Independence:

For example, if we add a fourth variable, Weather the full joint distribution then becomes P(Toothache, Catch, Cavity, Weather), which has 2 × 2 × 2 × 4 = 32 entries. It contains four "editions" of the table shown in Figure 13.3, one for each kind of weather. What relationship do these editions have to each other and to the original three-variable table? For example, how are P (toothache, catch, cavity, cloudy) and P (toothache, catch, cavity) related?

We can use the product rule:
P (toothache, catch, cavity, cloudy) = P (cloudy | toothache, catch, cavity) P (toothache, catch, cavity).
Therefore, the following assertion seems reasonable:
P(cloudy | toothache, catch, cavity) = P(cloudy)
From this, we can deduce

P(toothache, catch, cavity, cloudy) = P(cloudy)P(toothache, catch, cavity)

This property is called **independence or marginal independence or absolute independence.** In particular, the weather is independent of one's dental problems. Independence between propositions a and b can be written as
P(a | b)=P(a) or P(b | a)=P(b) or P(a ∧ b)=P(a)P(b).

Independence assertions are usually based on knowledge of the domain. As the toothache–weather example illustrates, they can **dramatically reduce the amount of information necessary to specify the full joint distribution.**
If the complete set of variables can be divided into independent subsets, then the full joint distribution can be factored into separate joint distributions on those subsets. For example, the full joint distribution on the outcome of n independent coin flips, P (C_1, . . . ,C_n), has 2^n entries, but it can be represented as the product of n single-variable distributions P(C_i). An example is shown in Figure 13.4

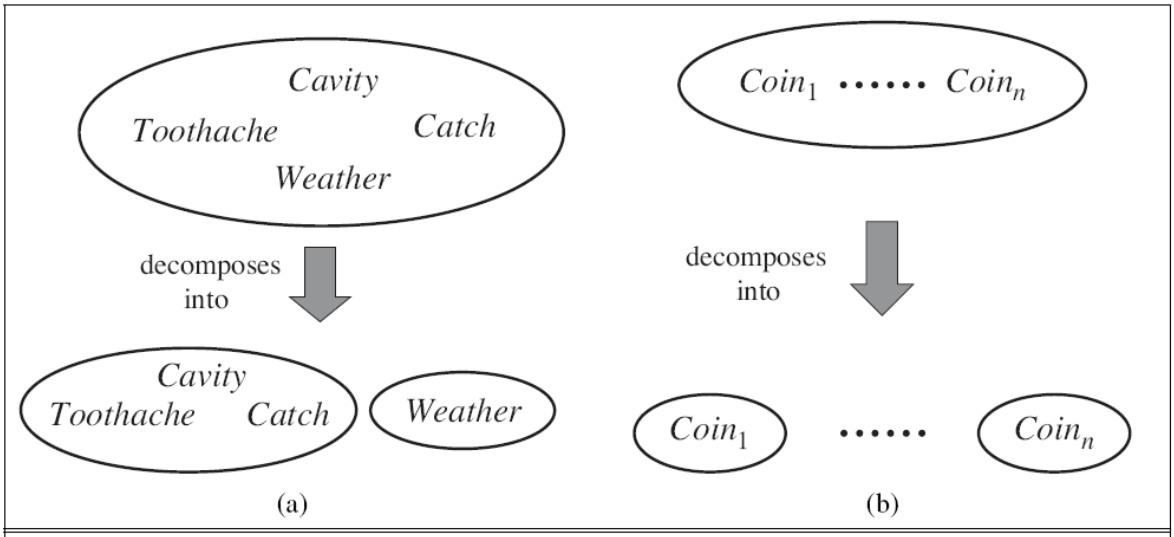


Figure 13.4 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

BAYES' RULE AND ITS USE:

We defined the **product rule**. It can actually be written in two forms:

$$P(a \wedge b) = P(a \mid b)P(b) \quad \text{and} \quad P(a \wedge b) = P(b \mid a)P(a)$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b \mid a) = \frac{P(a \mid b)P(b)}{P(a)} .$$

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem). This simple equation underlies most modern AI systems for probabilistic inference.
The more general case of Bayes' rule for multivalued variables can be written in the **P** notation as follows:

$$\mathbf{P}(Y \mid X) = \frac{\mathbf{P}(X \mid Y)\mathbf{P}(Y)}{\mathbf{P}(X)} ,$$

We will also have occasion to use a more general version conditionalized on some background evidence **e**:

$$\mathbf{P}(Y \mid X, \mathbf{e}) = \frac{\mathbf{P}(X \mid Y, \mathbf{e})\mathbf{P}(Y \mid \mathbf{e})}{\mathbf{P}(X \mid \mathbf{e})} .$$

Applying Bayes' rule: The simple case:

Bayes' rule does not seem very useful. It allows us to compute the single term P(b | a) in terms of three terms: P(a | b), P(b), and P(a). That seems like two steps backwards, but Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth.

We can also perceive as evidence the *effect* of some unknown *cause* and we would like to determine that cause. In that case, Bayes' rule becomes

$$P(cause \mid effect) = \frac{P(effect \mid cause)P(cause)}{P(effect)} .$$

The conditional probability P (*effect* CAUSAL | cause) quantifies the relationship in the **causal** direction, whereas P (cause|*effect*) describes the **diagnostic** direction. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships (that is, the doctor knows P (symptoms|disease)) and want to derive a diagnosis, P(disease| symptoms).

An Example Problem:

For example, a doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 70% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1%.

Solution: Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have

$$\begin{aligned} P(s \mid m) &= 0.7 \\ P(m) &= 1/50000 \\ P(s) &= 0.01 \\ P(m \mid s) &= \frac{P(s \mid m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014 . \end{aligned}$$

Inference:-That is, we expect less than 1 in 700 patients with a stiff neck to have meningitis.

Observation: Notice that even though a stiff neck is quite strongly indicated by meningitis (with probability 0.7), the probability of meningitis in the patient remains small. This is because the prior probability of stiff necks is much higher than that of meningitis.

The same process can be applied when using Bayes' rule. We have

$$\mathbf{P}(M \mid s) = \alpha \langle P(s \mid m)P(m), P(s \mid \neg m)P(\neg m) \rangle .$$

The general form of Bayes' rule with normalization is

$$\mathbf{P}(Y \mid X) = \alpha \mathbf{P}(X \mid Y)\mathbf{P}(Y)$$

where α is the normalization constant needed to make the entries in **P**(Y |X) sum to 1.

Using Bayes' rule: Combining evidence:

The bayes rule can be applied when we have two or more pieces of evidence. For example consider the full joint distribution of dentist shown in Figure 13.3. We are reproducing the table here.

$$\mathbf{P}(Cavity \mid toothache \wedge catch) = \alpha \mathbf{P}(toothache \wedge catch \mid Cavity)\mathbf{P}(Cavity) .$$

	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

If we know full joint distribution, we can answer the queries,

$$\mathbf{P}(Cavity \mid toothache \wedge catch) = \alpha \langle 0.108, 0.016 \rangle \approx \langle 0.871, 0.129 \rangle .$$

The Bayes rule can be reformulated as

$$\mathbf{P}(Cavity \mid toothache \wedge catch) = \alpha \mathbf{P}(toothache \wedge catch \mid Cavity)\mathbf{P}(Cavity) .$$

Conditional Independence: It would be nice if Toothache and Catch were independent, but they are not: if the probe catches in the tooth, then it is likely that the tooth has a cavity and that the cavity causes a toothache. These variables *are* independent, however, *given the presence or the absence of a cavity*. Each is directly caused by the cavity, but neither has a direct effect on the other: toothache depends on the state of the nerves in the tooth, whereas the probe's accuracy depends on the dentist's skill, to which the toothache is irrelevant. Mathematically, this property is written as

$$\mathbf{P}(toothache \wedge catch \mid Cavity) = \mathbf{P}(toothache \mid Cavity)\mathbf{P}(catch \mid Cavity) .$$

This equation expresses the **conditional independence** CONDITIONAL of toothache and catch given Cavity.

The general definition of **conditional independence** of two variables X and Y , given a third variable Z, is

$$\mathbf{P}(X, Y \mid Z) = \mathbf{P}(X \mid Z)\mathbf{P}(Y \mid Z) .$$

$$\mathbf{P}(Toothache, Catch \mid Cavity) = \mathbf{P}(Toothache \mid Cavity)\mathbf{P}(Catch \mid Cavity)$$

We can derive the decomposition as

$$\begin{aligned}&\mathbf{P}(Toothache, Catch, Cavity)\\ &= \mathbf{P}(Toothache, Catch \mid Cavity)\mathbf{P}(Cavity) \quad \text{(product rule)}\\ &= \mathbf{P}(Toothache \mid Cavity)\mathbf{P}(Catch \mid Cavity)\mathbf{P}(Cavity) \quad \text{(usi}\end{aligned}$$

The full joint distribution can be written as

$$\mathbf{P}(Cause, Effect_1, \ldots, Effect_n) = \mathbf{P}(Cause)\prod_i \mathbf{P}(Effect_i \mid Cause) .$$

Such a probability distribution is called a **naive Bayes** model—"naive" because it is often used (as a simplifying assumption) in cases where the "effect" variables are *not* actually conditionally independent given the cause variable. (The naive Bayes model is sometimes called a **Bayesian classifier**, a somewhat careless usage that has prompted true Bayesians to call it the **idiot Bayes** model.)

Probabilistic Reasoning:

The importance of independence and conditional independence relationships *are useful* in simplifying probabilistic representations of the world.

A systematic way to represent such relationships explicitly in the form of **Bayesian networks**. We define the syntax and semantics of these networks and show how they can be used to capture uncertain knowledge in a natural and efficient way.

REPRESENTING KNOWLEDGE IN AN UNCERTAIN DOMAIN:

The independence and conditional independence relationships among variables can greatly reduce the number of probabilities that need to be specified in order to define the full joint distribution. This section introduces a data structure called a **Bayesian network** to represent the dependencies among variables. Bayesian networks can represent essentially *any* full joint probability distribution and in many cases can do so very concisely.

What are Bayesian networks?

A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. The full specification is as follows:

1. Each node corresponds to a random variable, which may be discrete or continuous.
2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y, X is said to be a *parent* of Y. The graph has no directed cycles (and hence is a directed acyclic graph, or DAG.
3. Each node X_i has a conditional probability distribution $\mathbf{P}(X_i \mid \text{Parents}(X_i))$ that quantifies the effect of the parents on the node.

- The topology of the network—the set of nodes and links—specifies the conditional independence relationships that hold in the domain, in a way that will be made precise shortly.
- The *intuitive* meaning of an arrow is typically that X has a *direct influence* on Y, which suggests that causes should be parents of effects.
- It is usually easy for a domain expert to decide what direct influences exist in the domain—much easier, in fact, than actually specifying the probabilities themselves.
- Once the topology of the Bayesian network is laid out, we need only specify a conditional probability distribution for each variable, given its parents.

- We will see that the combination of the topology and the conditional distributions suffices to specify (implicitly) the full joint distribution for all the variables.

Example 1: The variables Toothache, Cavity, Catch, and Weather. We know that Weather is independent of the other variables; furthermore, we know that Toothache and Catch are conditionally independent, given Cavity. These relationships are represented by the Bayesian network structure shown in Figure 14.1.

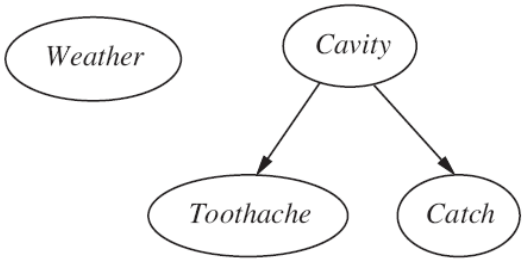


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

Formally, the conditional independence of Toothache and Catch, given Cavity, is indicated by the *absence* of a link between Toothache and Catch. Intuitively, the network represents the fact that Cavity is a direct cause of Toothache and Catch, whereas no direct causal relationship exists between Toothache and Catch.

Example 2: You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and often misses the alarm altogether.

Query: Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

- A Bayesian network for this domain appears in Figure 14.2.

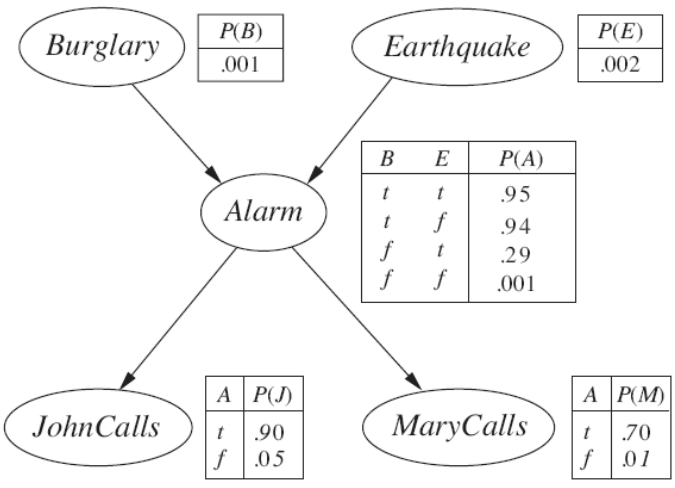


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

- The network structure shows that burglary and earthquakes directly affect the probability of the alarm's going off, but whether John and Mary call depends only on the alarm.
- The network thus represents our assumptions that they do not perceive burglaries directly, they do not notice minor earthquakes, and they do not confer before calling.

The conditional distributions in Figure 14.2 are shown as a **conditional probability table**, or CPT. Each row in a CPT contains the conditional probability of each node value for a **conditioning case**. A conditioning case is just a possible combination of values for the parent nodes—a miniature possible world.

Example problem: What is the probability that the alarm has sounded, but neither a burglary nor an earth quake has occurred, and both John and Mary call.

$$\begin{aligned} P(j, m, a, \neg b, \neg e) &= P(j \mid a)P(m \mid a)P(a \mid \neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.000628 \end{aligned}$$

Exercise Problems: What is the probability that the alarm has sounded, when there is a burglary but not an earth quake has occurred, and john calls but Mary does not call.

P (a, j, ¬m,b, ¬e) = **(0.90 * 0.30 * 0.94 *0.001 * 0.998)**

The semantics of Bayesian network:

There are two ways in which one can understand the semantics of Bayesian networks.

- The first is to see the network as a representation of the joint probability distribution.
- The second is to view it as an encoding of a collection of conditional independence statements.
- The two views are equivalent, but the first turns out to be helpful in understanding how to *construct* networks, whereas the second is helpful in designing inference procedures.

Representing the full joint distribution:

The conditional probabilities $\mathbf{P}(X_i \mid Parents(X_i))$ can be think like as members of $\theta(X_i \mid Parents(X_i))$.

A generic entry in the joint distribution is the probability of a conjunction of particular assignments to each variable, such as P(X1 =x1 ∧ . . . ∧ Xn =xn). We use the notation P(x1, . . . , xn) as an abbreviation for this. The value of this entry is given by the formula.

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} \theta(x_i \mid parents(X_i))$$

Further we can rewrite the equation as

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i))$$

Method for constructing the Baye's network:

$$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1)P(x_{n-1}, \ldots, x_1) .$$

Then we repeat the process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product:

$$\begin{aligned} P(x_1, \ldots, x_n) &= P(x_n \mid x_{n-1}, \ldots, x_1)P(x_{n-1} \mid x_{n-2}, \ldots, x_1) \cdots P(x_2 \mid x_1)P(x_1) \\ &= \prod_{i=1}^{n} P(x_i \mid x_{i-1}, \ldots, x_1) . \end{aligned}$$

This identity is called the **chain rule**. we see that the specification of the joint distribution is equivalent to the general assertion that, for every variable Xi in the network

$$\mathbf{P}(X_i \mid X_{i-1}, \ldots, X_1) = \mathbf{P}(X_i \mid Parents(X_i))$$

Provided that Parents (Xi) ⊆ {Xi−1, . . .,X1}.

Bayesian network is a correct representation of the domain only if each node is conditionally independent of its other predecessors in the node ordering, given its parents. We can satisfy this condition with this methodology:

1. ***Nodes:*** First determine the set of variables that are required to model the domain. Now order them, {X1, . . . ,Xn}. Any order will work, but the resulting network will be more compact if the variables are ordered such that causes precede effects.

2. *Links:* For i = 1 to n do:

• Choose, from $X_1, \ldots, X_{i-1}$, a minimal set of parents for X_i.

• For each parent insert a link from the parent to X_i.

• CPTs: Write down the conditional probability table, **P** (X_i|Parents(X_i)).

Intuitively, the parents of node Xi should contain all those nodes in X1, . . . , Xi−1 that *directly influence* Xi. For example, suppose we have completed the network in Figure 14.2 except for the choice of parents for MaryCalls. MaryCalls is certainly influenced by whether there is a Burglary or an Earthquake, but not *directly* influenced. Intuitively, our knowledge of the domain tells us that these events influence Mary's calling behavior only through their effect on the alarm. Also, given the state of the alarm, whether John calls has no influence on Mary's calling. Formally speaking, we believe that the following conditional independence statement holds:

P(MaryCalls ⎪ JohnCalls , Alarm, Earthquake, Burglary) = **P**(MaryCalls ⎪ Alarm)

Thus, Alarm will be the only parent node for MaryCalls.

Compactness and node ordering:

The compactness of Bayesian networks is an example of a general property of **locally structured** (also called **sparse**) systems. In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components.

An Example:

Even in a locally structured domain, we will get a compact Bayesian network only if we choose the node ordering well. What happens if we happen to choose the wrong order? Consider the burglary example again. Suppose we decide to add the nodes in the order MaryCalls, JohnCalls, Alarm, Burglary, and Earthquake. We then get the somewhat more complicated network shown in Figure 14.3(a).

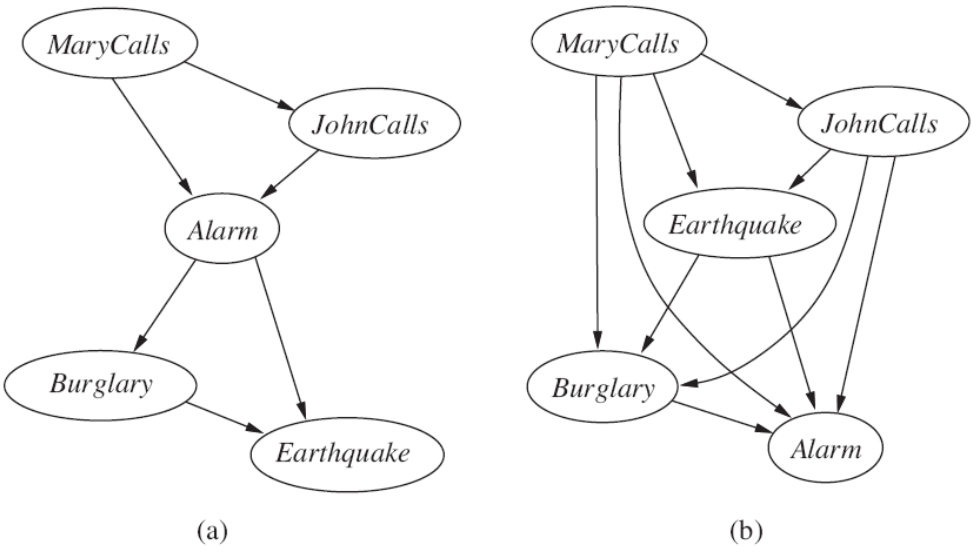


Figure 14.3 Network structure depends on order of introduction. In each network, we have introduced nodes in top-to-bottom order.

The process goes as follows:
• Adding MaryCalls: No parents.
• Adding JohnCalls: If Mary calls, that probably means the alarm has gone off, which of course would make it more likely that John calls. Therefore, JohnCalls needs MaryCalls as a parent.

• Adding Alarm: Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither calls, so we need both MaryCalls and JohnCalls as parents.

• Adding Burglary: If we know the alarm state, then the call from John or Mary might give us information about our phone ringing or Mary's music, but not about burglary: **P**(Burglary | Alarm, JohnCalls ,MaryCalls) = **P**(Burglary | Alarm) . Hence we need just Alarm as parent.

• Adding Earthquake: If the alarm is on, it is more likely that there has been an earthquake. (The alarm is an earthquake detector of sorts.) But if we know that there has been a burglary, then that explains the alarm, and the probability of an earthquake would be only slightly above normal. Hence, we need both Alarm and Burglary as parents.

The resulting network has two more links than the original network in Figure 14.2 and requires three more probabilities to be specified. What's worse, some of the links represent tenuous relationships that require difficult and unnatural probability judgments, such as assessing the probability of Earthquake, given Burglary and Alarm.
Figure 14.3(b) shows a very bad node ordering: MaryCalls, JohnCalls, Earthquake, Burglary, Alarm. This network requires 31 distinct probabilities to be specified—exactly the same number as the full joint distribution. It is important to realize, however, that any of the three networks can represent *exactly the same joint distribution*. The last two versions simply fail to represent all the conditional independence relationships and hence end up specifying a lot of unnecessary numbers instead.
Conditional independence relations in Bayesian networks:
The "topological" semantics that specifies the conditional independence relationships encoded by the graph structure, and from this we can derive the "numerical" semantics. The topological semantics2 specifies that each variable is conditionally independent of its non-**descendants**,

given its parents. For example, in Figure 14.2, JohnCalls is independent of Burglary, Earthquake, and MaryCalls given the value of Alarm. The definition is illustrated in Figure 14.4(a). From these conditional independence assertions and the interpretation of the network parameters $\theta(X_i \mid \text{Parents}(X_i))$ as specifications of conditional probabilities $\mathbf{P}(X_i \mid \text{Parents}(X_i))$, the full joint distribution given in Equation (14.2) can be reconstructed. In this sense, the "numerical" semantics and the "topological" semantics are equivalent.

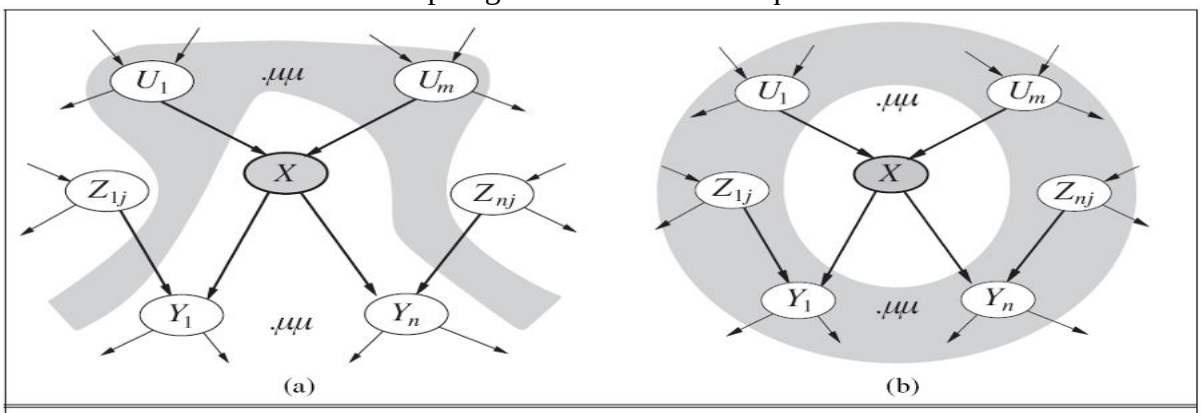


Figure 14.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

Another important independence property is implied by the topological semantics: a node is conditionally independent of all other nodes in the network, given its parents, children, and children's parents—that is, given its **Markov blanket**. (Exercise 14.7 asks you to prove this.) For example, Burglary is independent of JohnCalls and MaryCalls, given Alarm and Earthquake. This property is illustrated in Figure 14.4(b).

V. Learning

An agent is **learning** if it improves its performance LEARNING on future tasks after making observations about the world.

Forms of Learning:

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which *component* is to be improved?
- What *prior knowledge* the agent already has?
- What *representation* is used for the data and the component?
- What *feedback* is available to learn from?

Components to be learned:

The components of these agents include:

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible actions the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe classes of states whose achievement maximizes the agent's utility.

Forms of Learning or Feedback to learn from:

There are three *types of feedback* that determine the three main types of learning:

1) In **unsupervised learning** the agent learns patterns in the input even though no explicit feedback is supplied. The most common unsupervised learning task is **clustering**: detecting potentially useful clusters of input examples. For example, a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days" without ever being given labeled examples of each by a teacher.

2) In **reinforcement learning** the agent learns from a series of reinforcements—rewards or punishments. For example, the lack of a tip at the end of the journey gives the taxi agent an indication that it did something wrong. The two points for a win at the end of a chess game tells the agent it did something right. It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it.

3) In **supervised learning** the agent observes some example input–output pairs and learns a function that maps from input to output. In component 1 above, the inputs are percepts and the output are provided by a teacher who says "Brake!" or "Turn left." In component 2, the inputs are camera images and the outputs again come from a teacher who says "that's a bus." In 3, the theory of braking is a function from states and braking actions to stopping distance in feet. In this case the output value is available directly from the agent's percepts (after the fact); the environment is the teacher.

4) In **semi-supervised learning** we are given a few labeled examples and must make what we can of a large collection of unlabelled examples. Even the labels themselves may not be the oracular truths that we hope for. Imagine that you are trying to build a system to guess a person's age from a photo. You gather some labeled examples by snapping pictures of people and asking their age.

Inductive Learning:

An algorithm for deterministic supervised learning is given as input the correct value of the unknown function for particular inputs and must try to recover the unknown function or

something close to it. More formally, we say that an **example** is a pair *(x, f* (z)), where x is the input and *f(x)* is the output of the function applied to *x.* The task of **pure inductive inference** (or **induction)** is this:
Given a collection of examples of *f*, return a function *h* that approximates f.

The function h is called a **hypothesis.** The reason that learning is difficult, from a conceptual point of view, is that it is not easy to tell whether any p,articular h is a good approximation of *f* . A good hypothesis will **generalize** well-that is, will predict unseen examples correctly. This is the fundamental **problem of induction.**

Example: Figure: 18.1 shows a familiar example: fitting a function of a single variable to some data points. The examples are *(x,* f *(x))* pairs, where both x and f (x) are real numbers. We choose the **hypothesis space** H-the set of hypotheses we will consider-to be the set of polynomials of degree at most ***k,*** such as $x^5 + 3x^2 + 2$, and so on. Figure 18.l(a) shows some data with an exact fit by a straight line (a polynomial of degree I). The line is CONSISTENT called a **consistent** hypothesis because it agrees with all the data. Figure 18.l(b) shows a high-degree polynomial that is also consistent with the same data.

This illustrates the first issue in inductive learning: *how do we choose from among multiple consistent hypotheses?*
Ockham's razor: One answer is Ockham's razor: prefer the *simplest* hypothesis consistent with the data. Intuitively, this makes sense, because hypotheses that are no simpler than the data themselves are failing to extract any *pattern* from the data. Defining simplicity is not easy, but it seems reasonable to say that a degree- 1 polynomial is simpler than a degree-12 polynomial.

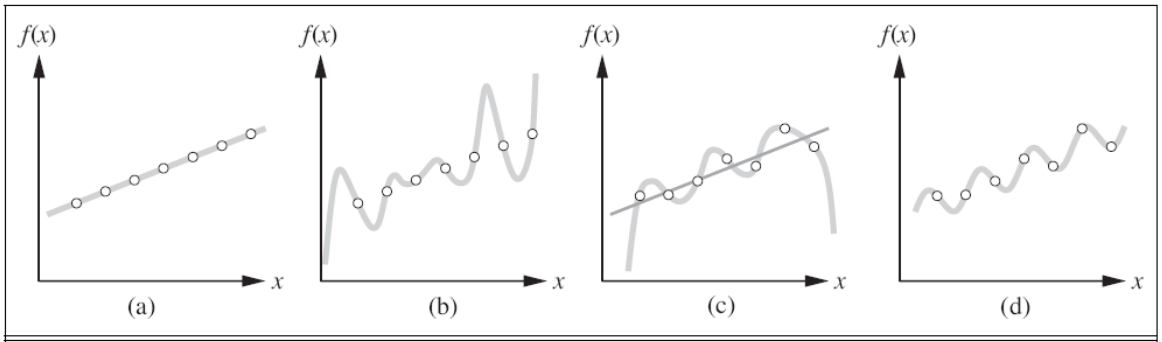


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

Figure 18.1(c) shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial for an exact fit. There are just 7 data points, so a polynomial with 7 parameters does not seem to be finding any pattern in the data and we do not expect it to generalize well. A straight line that is not consistent with any of the data points, but might generalize fairly well for unseen values of x, is also shown in (c). *In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better.* In Figure 18.1(d) we expand the hypothesis space H to allow polynomials over both x and sin(x), and find that the data in (c) can be fitted exactly by a simple function of the form ax + b + c sin(x).

We say that a learning problem is **realizable** if the hypothesis space contains the true function. Unfortunately, we cannot always tell whether a given learning problem is realizable, because the true function is not known.
Supervised learning can be done by choosing the hypothesis h^* that is most probable given the data:

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(h|data) \ .$$

By Bayes' rule this is equivalent to

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(data|h) \, P(h) \ .$$

Learning Decision Trees:

Decision tree induction is one of the simplest and yet most successful forms of machine learning. We first describe the representation—the hypothesis space—and then show how to learn a good hypothesis.

The decision tree representation:

A **decision tree** represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value. The input and output values can be discrete or continuous. For now we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each example input will be classified as true (a **positive** example) or false (a **negative** example).

Example: As an example, we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** WillWait. First we list the attributes that we will consider as part of the input:

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. Hungry: whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant's price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai, or burger).
10. WaitEstimate: the wait estimated by the host (0–10 minutes, 10–30, 30–60, or >60).

A Boolean decision tree consists of an (**x**, y) pair, where **x** is a vector of values for the input attributes, and y is a single Boolean output value. A training set of 12 examples is shown in Figure 18.3. The positive examples are the ones in which the goal WillWait is true (**x**1, **x**3, . . .); the negative examples are the ones in which it is false (**x**2, **x**5, . . .).

Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Figure 18.3 Examples for the restaurant domain.

Expressiveness of decision tree:

A decision tree for the above data shown in Figure 18.2.

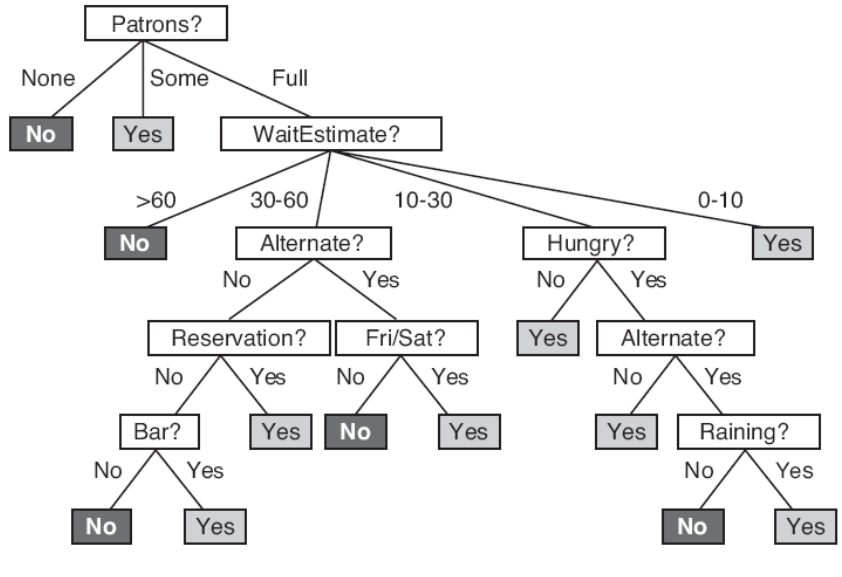


Figure 18.2 A decision tree for deciding whether to wait for a table.

A Boolean decision tree is logically equivalent to the assertion that the goal attribute is true if and only if the input attributes satisfy one of the paths leading to a leaf with value true. Writing this out in propositional logic, we have

$$Goal \Leftrightarrow (Path_1 \vee Path_2 \vee \cdots)$$

The logical rules can be expressed in the form of propositional logic

$$Path = (Patrons = Full \wedge WaitEstimate = 0\text{–}10)$$

Choosing the best attribute: The greedy search used in decision tree learning is designed to approximately minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect attribute divides the examples into sets, each of which are all positive or all negative and thus will be leaves of the tree.

We use **entropy** for choosing the best attribute, the fundamental quantity in information theory. Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy.

Information gain is used to find the entropy given in the dataset. The **information gain** from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$$

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k+n_k}{p+n} B(\frac{p_k}{p_k+n_k})$$

The information gain ranges from 0 to 1. For example, Gain(pat)=0.541 and Gain(Type)=0.2 then we consider that **patron** is the best attribute to split on. So, the decision tree shown in Figure 18.6 using Information gain as an entropy measure.

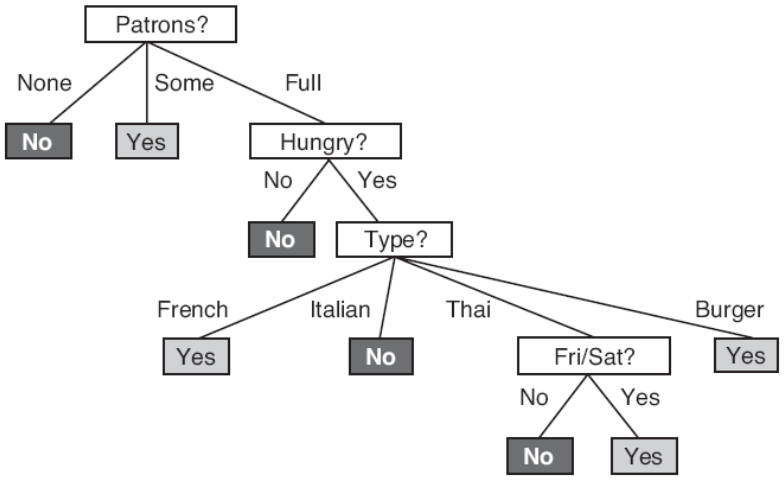


Figure 18.6 The decision tree induced from the 12-example training set.

The decision tree learning algorithm: The learning algorithm to construct a decision tree is shown in Figure 18.5.

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns
a tree

  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← {e : e ∈ examples and e.A = v_k}
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

Figure 18.5 The decision-tree learning algorithm. The function IMPORTANCE is described in Section 18.3.4. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

Learning Curve: We can evaluate the accuracy of a learning algorithm with a **learning curve**, as shown in Figure 18.7. We have 100 examples at our disposal, which we split into a training set and a test set. We learn a hypothesis h with the training set and measure its accuracy with the test set. We do this starting with a training set of size 1 and increasing one at a time up to size 99. For each size we actually repeat the process of randomly splitting 20 times, and average

the results of the 20 trials. The curve shows that as the training set size grows, the accuracy increases. (For this reason, learning curves are also called **happy graphs**.) In this graph we reach 95% accuracy, and it looks like the curve might continue to increase with more data.

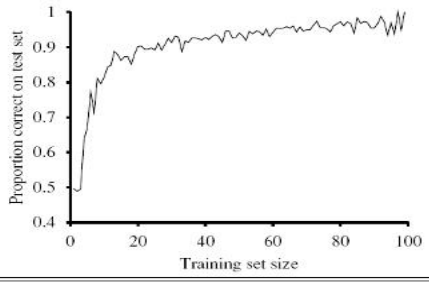


Figure 18.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

Generalization and overfitting: On some problems, the DECISION-TREE-LEARNING algorithm will generate a large tree when there is actually no pattern to be found. This problem is called **overfitting**. A general phenomenon, overfitting occurs with all types of learners, even when the target function is not at all random.

For decision trees, a technique called **decision tree pruning** combats overfitting. Pruning works by eliminating nodes that are not clearly relevant. We start with a full tree, as generated by DECISION-TREE-LEARNING. We then look at a test node that has only leaf nodes as descendants. If the test appears to be irrelevant—detecting only noise in the data then we eliminate the test, replacing it with a leaf node. We repeat this process, considering each test with only leaf descendants, until each one has either been pruned or accepted as is.

Broadening the applicability of decision trees:
In order to extend decision tree induction to a wider variety of problems, a number of issues must be addressed. We will briefly mention several, suggesting that a full understanding is best obtained by doing the associated exercises:
•**Missing data**: In many domains, not all the attribute values will be known for every example. The values might have gone unrecorded, or they might be too expensive to obtain.
•**Multivalued attributes**: When an attribute has many possible values, the information gain measure gives an inappropriate indication of the attribute's usefulness.
•**Continuous and integer-valued input attributes**: Continuous or integer-valued attributes such as Height and Weight, have an infinite set of possible values. Rather than generate infinitely many branches, decision-tree learning algorithms typically find the **split point** that gives the highest information gain.
•**Continuous-valued output attributes**: If we are trying to predict a numerical output value, such as the price of an apartment, then we need a **regression tree** rather than a classification tree. A regression tree has at each leaf a linear function of some subset of numerical attributes, rather than a single value.

ENSEMBLE LEARNING:

The idea of **ensemble learning** methods is to select a whole collection, or **ensemble,** of hypotheses from the hypothesis space and combine their predictions. For example, we might generate a hundred different decision trees from the same training set and have them vote on the best classification for a new example. The motivation for ensemble learning is simple. Consider an ensemble of M = 5 hypotheses and suppose that we combine their predictions using simple majority voting. For the ensemble to misclassify a new example, *at least three of*

the five hypotheses have to misclassify it. The hope is that this is much less likely than a misclassification by a single hypothesis.

Suppose we assume that each hypothesis h_i in the ensemble has an error of p-that is, the probability that a randomly chosen example is rnisclassified by hi is p. Furthermore, suppose we assume that the errors made by each hypothesis are *independent.* In that case, if p is small, then the probability of a large number of misclassifications occurring is minuscule. For example, a simple calculation (Exercise 18.14) shows that using an ensemble of five hypotheses reduces an error rate of 1 in 10 down to an error rate of less than 1 in 100.

Example:
Figure 18.8 shows how this can result in a more expressive hypothesis space. If the original hypothesis space allows for a simple and efficient learning algorithm, then the ensemble method provides a way to learn a much more expressive class of hypotheses without incurring much additional computational or algorithmic complexity.

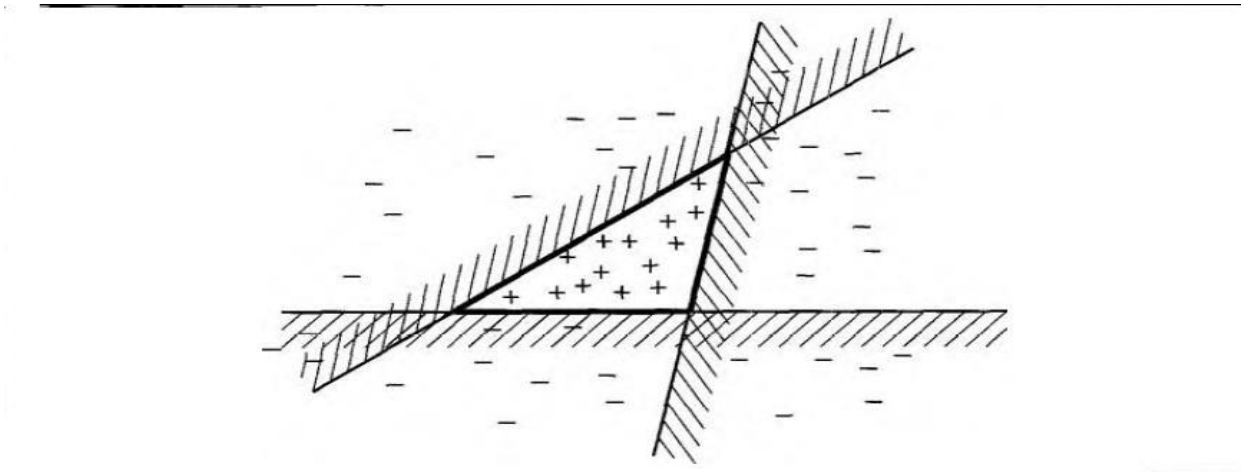


Figure 18.8 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the non-shaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

The most widely used ensemble method is called **boosting.** To understand how it works, we need first to explain the idea of a **weighted training set.** In such a training set, each example has an associated weight $w_J > 0$. The higher the weight of an example, the higher is the importance attached to it during the learning of a hypothesis. It is straightforward to modify the learning algorithms we have seen so far to operate with weighted training sets.

Boosting starts with w_j, = 1 for all the examples (i.e., a normal training set). From this set, it generates the first hypothesis, *hl.* This hypothesis wall classify some of the training examples correctly and some incorrectly. We would like the next hypothesis to do better on the misclassified examples, so we increase their weights while decreasing the weights of the correctly classified examples. From this new weighted training set, we generate hypothesis *h2.* The process continues in this way until we have generated M hypotheses, where M is an input to the boosting algorithm. The final ensemble hypothesis is a weighted-majority combination of all the M hypotheses, each weighted according to how well it performed on the training set. Figure 18.9 shows how the algorithm works conceptually.

The boosting algorithm shown in Figure 18.10.

```
function ADABOOST(examples, L, M) returns a weighted-majority hypothesis
  inputs: examples, set of N labelled examples (x1, y1), ..., (xN, yN)
          L, a learning algorithm
          M, the number of hypotheses in the ensemble
  local variables: w, a vector of N example weights, initially 1/N
                   h, a vector of M hypotheses
                   z, a vector of M hypothesis weights

  for m = 1 to M do
      h[m] ← L(examples, w)
      error ← 0
      for j = 1 to N do
          if h[m](xj) ≠ yj then error ← error + w[j]
      for j = 1 to N do
          if h[m](xj) = yj then w[j] ← w[j] . error/(1 − error)
      w ← NORMALIZE(w)
      z[m] ← log (1 − error)/error
  return WEIGHTED-MAJORITY(h, z)
```

Figure 18.10 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the trainng examples. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in h, with votes weighted by z.

Example: Let us see how well boosting does on the restaurant data. We will choose as our original hypothesis space the class of **decision stumps,** which are decision trees with just one test at the root. The lower curve in Figure 18.1 1(a) shows that unboosted decision stumps are not very effective for this data set, reaching a prediction performance of only 8 1 % on 100 training examples. When boosting is applied (with M = 5), the performance is better, reaching 93% after 100 examples.

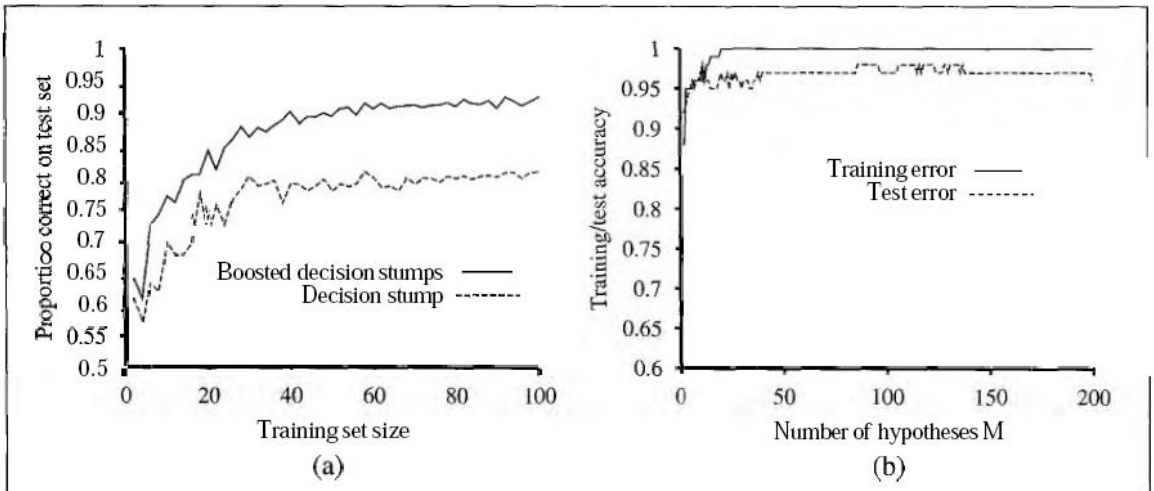


Figure 18.11 (a) Graph showing the performance of boosted decision stumps with M = 5 versus decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of M, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training, accuracy reaches 1, i.e., after the ensemble fits the data exactly.

An interesting thing happens as the ensemble size M increases. Figure 18.11(b) shows the training set performance (on 100 examples) as a function of M. Notice that the error reaches zero (as the boosting theorem tells us) when M is 20; that is, a weighted-majority combination of 20 decision stumps suffices to fit the 100 examples exactly. As more stumps are added to

the ensemble, the error remains at zero. The graph also shows that *the test set performance continues to increase long afer the training set error has reached zero.* At $M = 20$, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as $M = 137$, before gradually dropping to 0.95.

Computational Learning Theory:

Computational learning theory, a field at the intersection of AI, statistics, and theoretical computer science. The underlying principle is the following: *any hypothesis that is seriously wrong will almost certainly be "found out" with high probability after a small number of examples, because it will make an incorrect prediction. Thus, any hypothesis that is consistent with a suficiently large set of training* examples *is unlikely to be seriously wrong: that is, it must beprobably approximately correct.* Any learning algorithm that returns hypotheses that are probably approximately correct is called a **PAC-learning** algorithm.

How many examples are needed?

In order to put these insights into practice, we will need some notation:
Let X be the set of all possible examples.
Let D be the distribution from which examples aire drawn.
Let **H** be the set of possible hypotheses.
Let *N* be the number of examples in the training set.
Initially, we will assume that the true function *f* is a member of H. Now we can define the ERROR **error** of a hypothesis h with respect to the true functiion *f* given a distribution *L)* over the examples a.s the probability that h is different from *f* on an example:

$$\text{error}(h) = P(h(x) \neq f(x) | x \text{ drawn from D}) .$$

A hypothesis h is called **approximately correct** if $\text{error}(h) \leq \epsilon$, where ϵ is a smal
nstant. The plan of attack is to show that after seeing N examples, with high probability
l consistent hypotheses will be approximately correct. One can think of an approximatel
rrect hypothesis as being "close" to the true function in hypothesis space: it lies inside wha
called the **ϵ-ball** around the true function f. Figure 18.12 shows the set of all hypothese
, divided into the ϵ-ball around f and the remainder, which we call $\mathbf{H}_{\text{bad}}$.

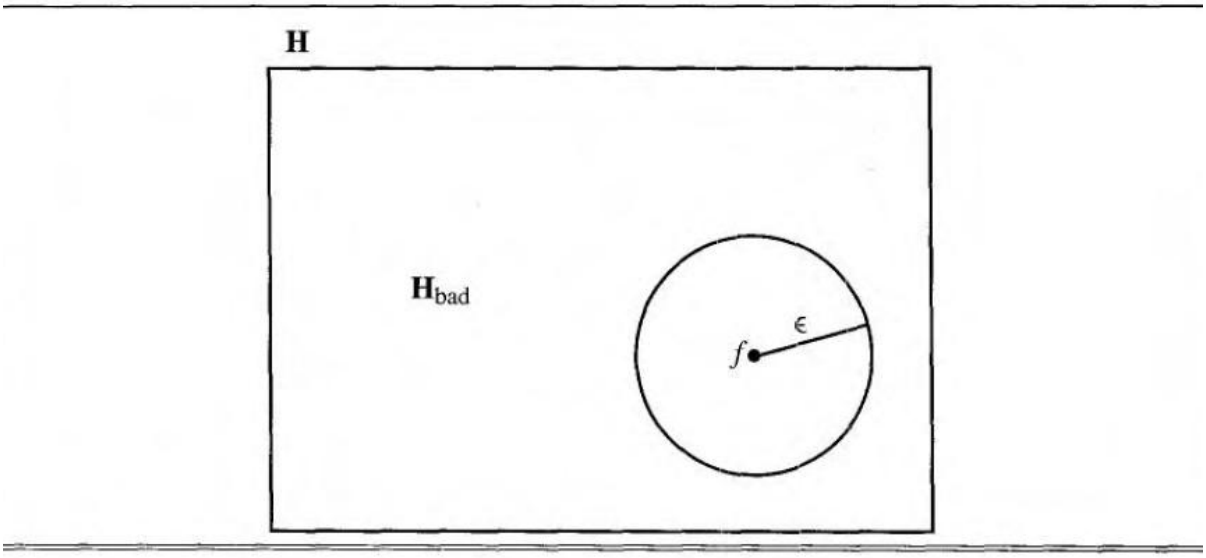


Figure 18.12 Schematic diagram of hypothesis space, showing the "ϵ-ball" around the true function f.

Learning decision lists:

A **decision list** is a logical expression of a restricted form. It consists of a series of tests, each of which is a conjunction of literals. If a test succeeds when applied to an example description, the decision list specifies the value to be returned. If the test fails, processing continues with the next test in the list.6 Decision lists resemble decision trees, but their overall structure is simpler. In contrast, the individual tests are more complex. Figure 18.13 shows a decision list that represents the following hypothesis:

$$\forall x \;\; WillWait(x) \Leftrightarrow Patrons(x, Some) \vee (Patrons(x, Full) \wedge Fri/Sat(x))$$

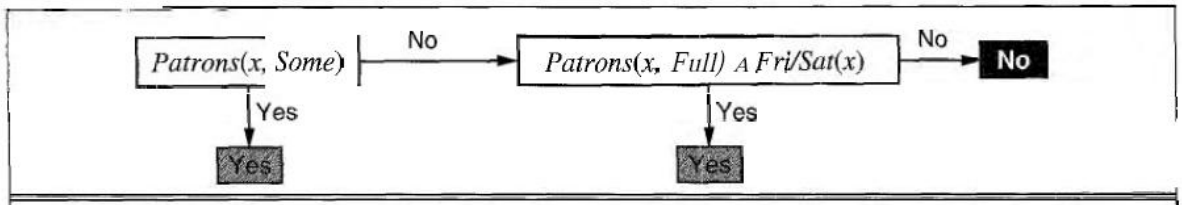


Figure 18.13 A decision list for the restaurant problem.

The decision list algorithm is shown in Figure 18.14.

function DECISION-LIST-LEARNING(*examples*) **returns** a decision list, or *failure*

if *examples* is empty **then return** the trivial decision list *No*
$t \leftarrow$ a test that matches a nonempty subset *examples*, of *examples*
such that the members of *examples*, are all positive or all negative
if there is no such t **then return** *failure*
if the examples in *examples*, are positive **then** o $\leftarrow$ *Yes* **else** $o \leftarrow$ *No*
return a decision list with initial test t and outcome o and remaining tests given by
DECISION-LIST-LEARNING(*examples* – *examples*,)

Figure 18.14 An algorithm for learning decision lists.

It would seem reasonable to prefer small tests that match large sets of uniformly classified examples, so that the overall decision list will be as compact as possible. The simplest strategy is to find the smallest test t that matches any uniformly classified subset, regardless of the size of the subset. Even this approach works quite well, as Figure 18.15 suggests.

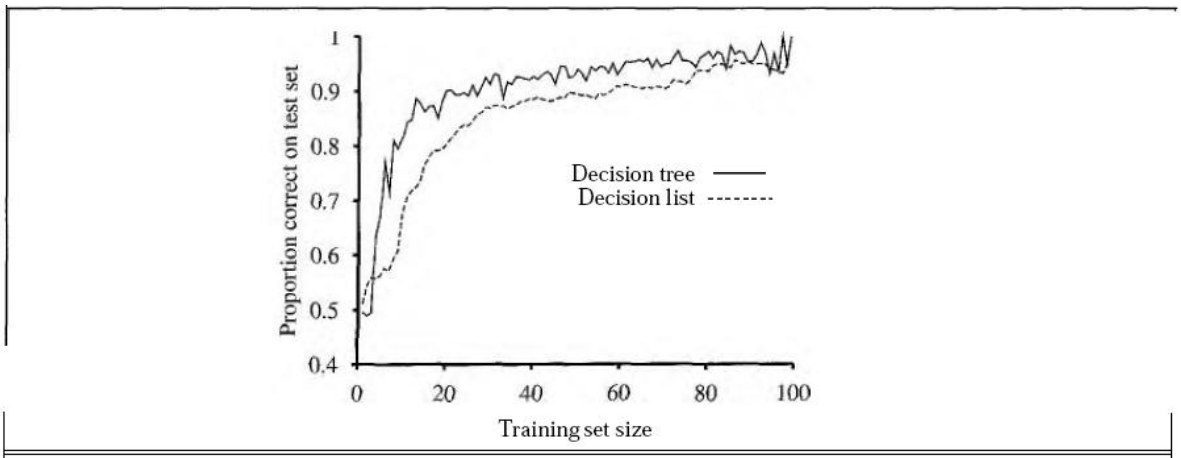


Figure 18.15 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

Statistical Learning: Instance Based Learning:

In contrast to parametric learning, **nonparametric learning** methods allow the hypothesis complexity to grow with the data. The more data we have, the wigglier the hypothesis can be. We will look at two very simple families of nonpararnetric **instance-based learning** (or **memory-based learning)** methods, so called because they construct hypotheses directly from the training instances themselves.

Nearest-neighbor models:

The key idea of **nearest-neighbor** models is that the properties of any particular input point x are likely to be similar to those of points in the neighborhood (of x. For example, if we want to do **density estimation-that** is, estimate the value of an unknown probability density at **x** then we can simply measure the density with which points are scattered in the neighborhood of x. This sounds very simple, until we realize that we need to specify exactly what we mean by "neighborhood." If the neighborhood is too small, it won't contain any data points; too large, and it may include all the data points, resulting in a density estimate that is the same everywhere. One solution is to define the neighborhood to be just big enough to include *k* points, where k is large enough to ensure a meaningful estimate. For fixed *k,* the size of the neighborhood varies-where data are sparse, the neighborhood is large, but where data are dense, the neighborhood is small.

Example: Figure 20.12(a) shows an example for data scattered in two dimensions. Figure 20.13 shows the results of k-nearest-neighbor density estimation from these data with k = **3,** 10, and 40 respectively. For *k* == **3,** the density estimate at any point is based on only **3** neighboring points and is highly variable. For *k* = 10, the estimate provides a good reconstruction of the true density shown in Figure 20.12(b). For k = 40, the neighborhood becomes too large and structure of the data is altogether lost. In practice, using a value of k somewhere between 5 and 10 gives good results for most low-dimensional data sets. A good value of k can also be chosen by using cross-validation.

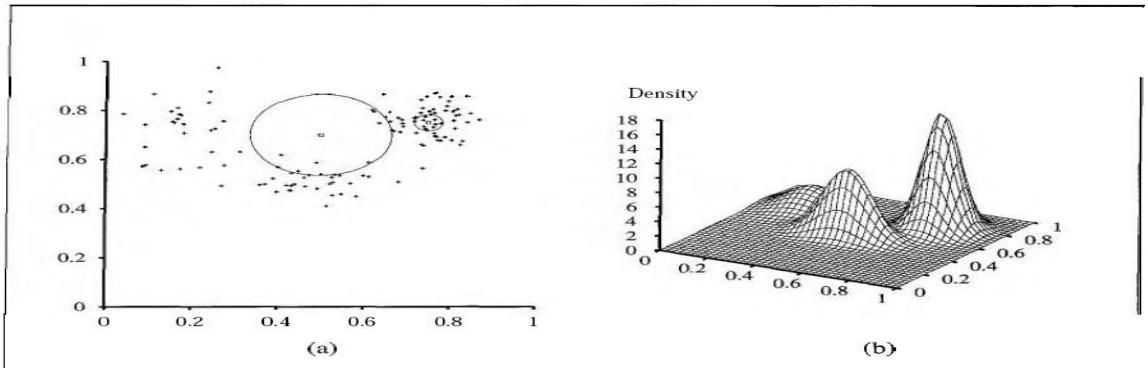


Figure 20.12 (a) A 128-point subsample of the data shown in Figure 20.8(a), together with two query points and their 10-nearest-neighborhoods. (b) A 3-D plot of the mixture of Gaussians from which the data were generated.

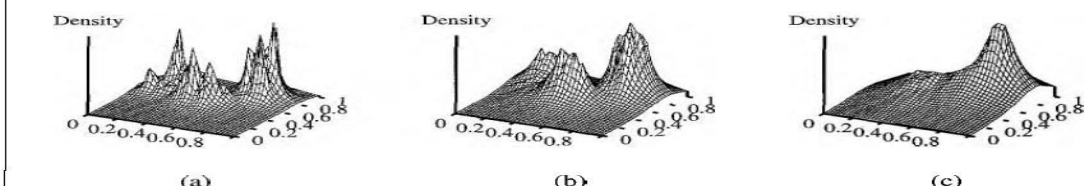


Figure 20.13 Density estimation using k-nearest-neighbors, applied to the data in Figure 20.12(a), for $k = 3$, 10, and 40 respectively.

To identify the nearest neighbors of a query point, we need a distance metric, **D(x_1, x_2).** It is also possible to use the nearest-neighbor idea for direct supervised learning. Given a test example with input x, the output y = h(x) is obtained from the y-values of the k nearest neighbors ad x. In the discrete case, we can obtain a single prediction by majority vote. In the continuous case, we can average the k values or do local linear regression, fitting a hyperplane to the k points and predicting the value at x according to the hyperplane.
Advantage: The k-nearest-neighbor learning algorithm is very simple to implement, requires little in the way of tuning, and often performs quite well.

Neural Networks:

A neuron is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals. The brain's information-processing capacity is thought to emerge primarily from *networks* of such neurons. For this reason, some of the earliest **A1** work aimed to create artificial **neural networks.** (Other names for the field include **connectionism, parallel distributed processing,** and **neural computation.)** Figure 20.15 shows a simple mathematical model of the neuron devised by McCulloch and Pitts (1943). Roughly speaking, it "fires" when a linear combination of its inputs exceeds some threshold.

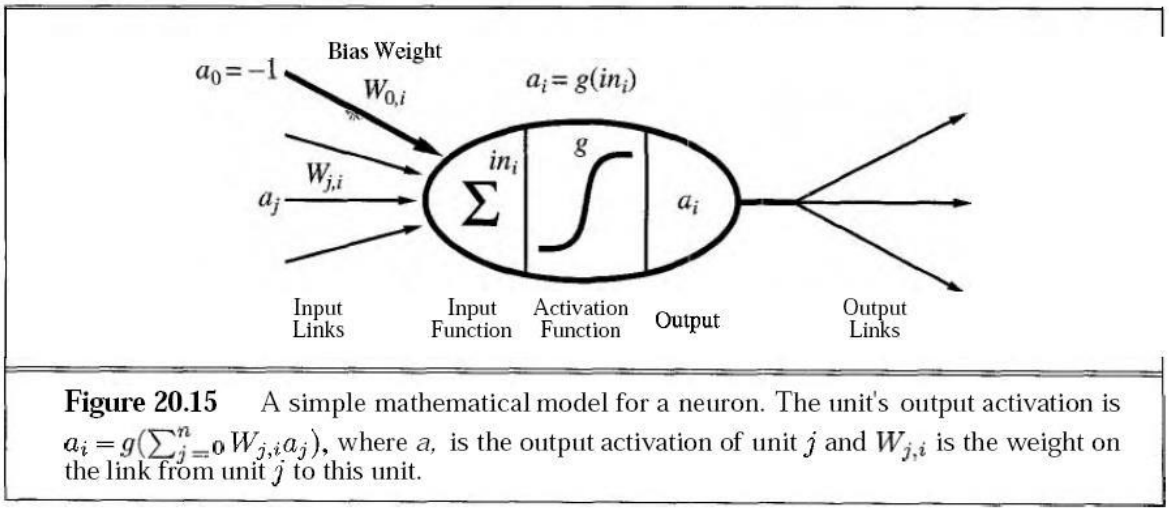


Figure 20.15 A simple mathematical model for a neuron. The unit's output activation is $a_i = g(\sum_{j=0}^{n} W_{j,i} a_j)$, where a_j is the output activation of unit j and $W_{j,i}$ is the weight on the link from unit j to this unit.

Units in neural networks:
Neural networks are composed of nodes or **units** (see Figure 2:0.15) connected by directed **links. A** link from unit j to unit i serves to propagate tlhe **activation** a_j from **j** to *i.* Each link also has a numeric **weight** $W_{j,i}$ *a*ssociated with it, which determines the strength and sign of the connection. Each unit i first computes a weighted sum of its inputs:

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j \ .$$

Then it applies an **activation function** g to this sum to derive the output:

$$a_i = g(in_i) = g\left(\sum_{j=0}^{n} W_{j,i} a_j\right) \ .$$

Notice that we have included a **bias weight** $W_{o,i}$ connected to a fixed input a_o = -1 The activation function g is designed to meet two things. First, we want the unit to be "active" (near +1) when the "right" inputs are given, and "inactive" (near 0) when the "wrong" inputs are given. Second, the activation needs to be nonlinear, otherwise the entire neural network

collapses into a simple linear function. Two choices for g are shown in Figure 20.16: the **threshold** function and the **sigmoid function** (also known as the **logistic function).** The sigmoid function has the advantage of being differentiable, which we will see later is important for the weight-learning algorithm. Notice that both functions have a threshold (either hard or soft) at zero; the bias weight sets $W_{o,i}$ the *actual* threshold for the unit, in the sense that the unit is activated when the weighted sum of "real" inputs.

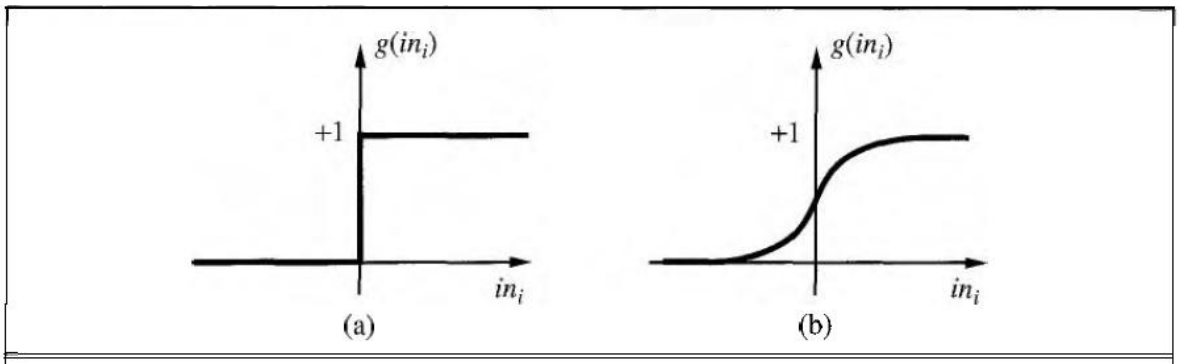


Figure 20.16 (a) The **threshold** activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the **sign** function is used instead, which outputs ±1 depending on the sign of the input.) (b) The **sigmoid** function $1/(1+e^{-x})$.

Network Structures: There are two main categories of neural network structures: acyclic or **feed-forward networks** and cyclic or **recurrent networks.** A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A recurrent network, on the other hand, feeds its outputs back into1 its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behaviour.

Feed-Forward Network: Let us look more closely into the assertion that a feed-forward network represents a function of its inputs. Consider the simple network shown in Figure 20.18, which has two input units, two **hidden units,** and an output unit. (To keep things simple, we have omitted the bias units in this example.) Given an input vector *x == (xl , x2),* the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$ and the network computes

$$\begin{aligned} a_5 &= g(W_{3,5}a_3 + W_{4,5}a_4) \\ &= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2)) \,. \end{aligned}$$

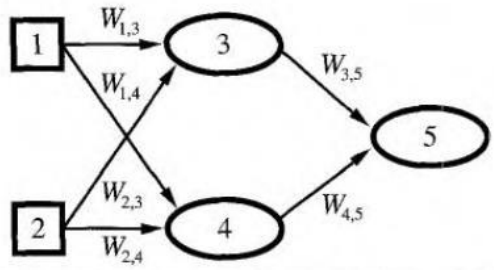


Figure 20.18 A very simple neural network with two inputs, one hidden layer of two units, and one output.

A neural network can be used for classification or regression. For Boolean classification with continuous outputs (e.g., with sigmoid units), it is traditional to have a single output unit, with a value over *0.5* interpreted as one class and a value bellow 0.5 as the other. For k-way

classification, one could divide the single output unit's range intlo *k* portions, but itt is more common to have *k* separate output units, with the value of each one representing the relative likelihood of that class given the current input.

Single layer feed-forward neural networks (perceptrons): A network with all the inputs connected directly to the outputs is called a **single-layer neural network,** or a **perceptron** network. Since each output unit is independent of the others each weight affects only one of the outputs-we can limit our study to perceptrons with a single output unit, as explained in Figure 20.19(a).

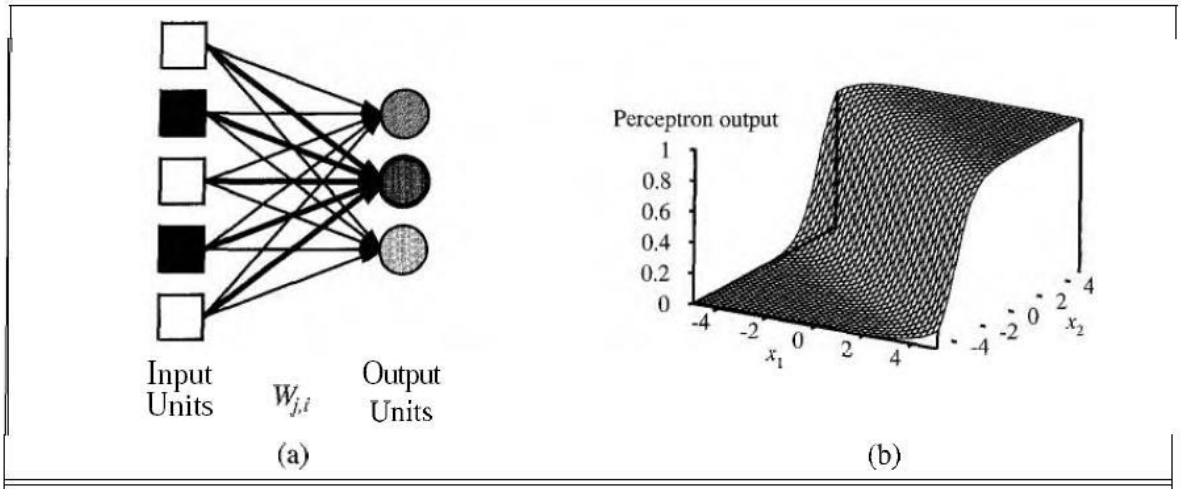


Figure 20.19 (a) A perceptron network consisting of three perceptron output units that share five inputs. Looking at a particular output unit (say the second one, outlined in bold), we see that the weights on its incoming links have no effect on the other output units. (b) A graph of the output of a two-input perceptron unit with a sigmoid activation function.

The threshold perceptron returns 1 if and only if the weighted sum of its inputs (including the bias) is positive: $\mathbf{W} \cdot \mathbf{x} > 0$

Now, the equation W. x = 0 defines a hyperplane in the input space, so the perceptron returns 1 if and only if the input is on one side of that hyperplane. For this reason, the threshold LINEARSEPAF~ATOR perceptron is called a **linear separator.** Figure 18.21(a) and (b) show this hyperplane (a line, in two dimensions) for the perceptron representations of the AND and OR functions of two inputs. The perceptron can represent these functions because there is some line that separates all the white dots from all the black dots. Such functions are called **linearly separable.**

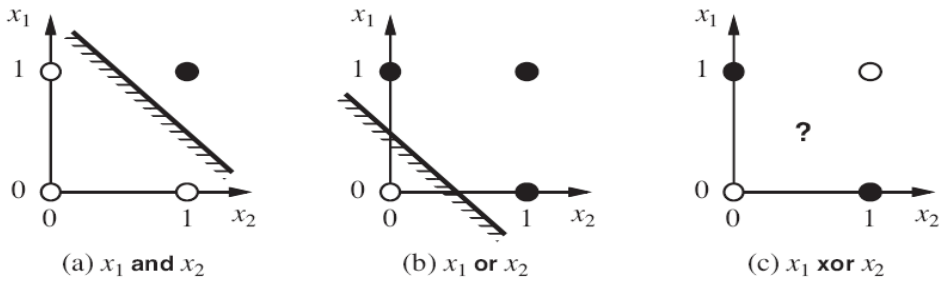


Figure 18.21 Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.

Figure 18.21(c) shows an example of a function that is *not* linearly separable-the XOR function. Clearly, there is no way for a threshold perceptron to learn this function. In general, *threshold perceptrons can ,represent only linearly separable functions.*
The Gradient descent learning algorithm is used in perceptron for learning. The perceptron learning algorithm is outlined in Figure 20.21

function PERCEPTRON-LEARNING(*examples*, *network*) **returns** a perceptron hypothesis
inputs: *examples*, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output y
network, a perceptron with weights W_j, $j = 0 \ldots n$, and activation function g

repeat
for each e **in** *examples* **do**
$in \leftarrow \sum_{j=0}^{n} W_j x_j[e]$
$Err \leftarrow y[e] - g(in)$
$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS(*network*)

Figure 20.21 The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g. For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.

Figure 20.22 shows the learning curve for a perceptron on two different problems. On the left, we show the curve for learning the majority function with 11 Boolean inputs (i.e., the function outputs a 1 if 6 or more inputs are 1). On the right, we have the restaurant example. The solution problem is easily represented as a decision tree, but is not linearly separable. The best plane through the data correctly classifies only 65%.

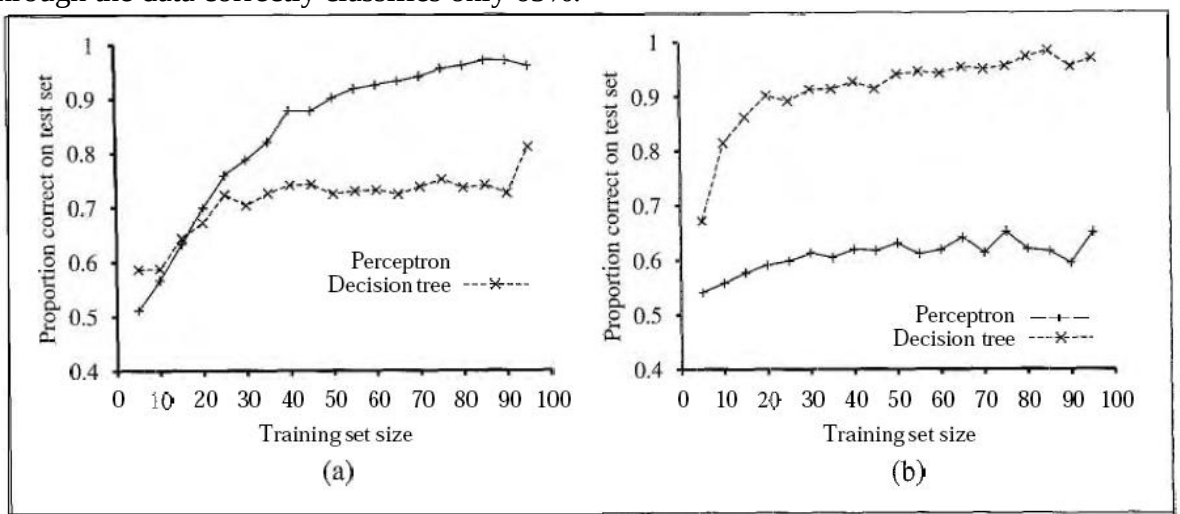


Figure 20.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *Will Wait* predicate in the restaurant example.

Multilayer feed-forward neural networks:

A Multilayer feed-forward neural networks consists of hidden layers. The most common case involves a single hidden layer,as in Figure 20.24. The advantage of adding hidden layers is that it enlarges the space of hypotheses that the network can represent. With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented. Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

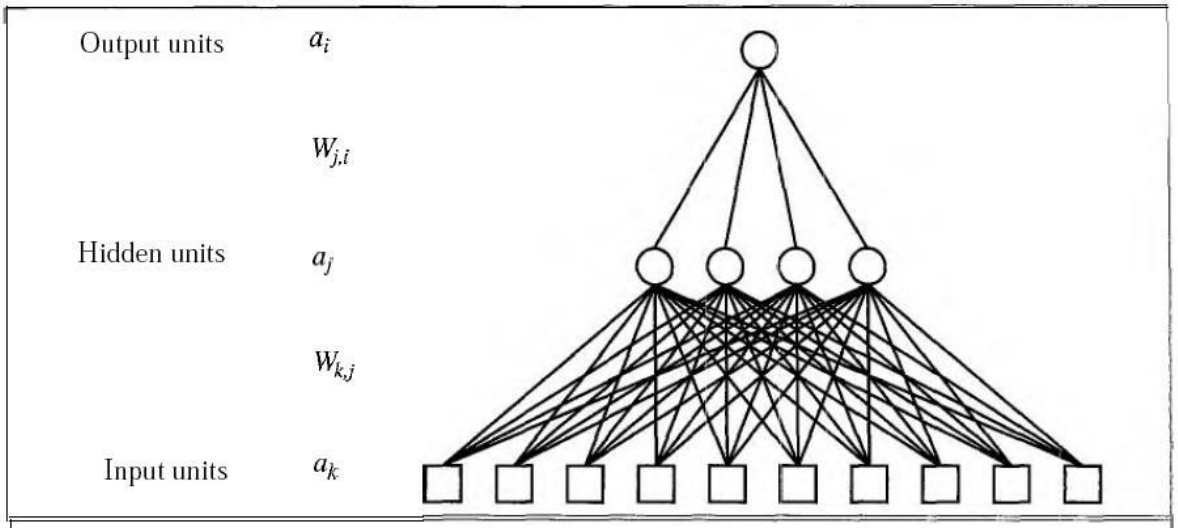


Figure 20.24 A multilayer neural network with one hidden layer and 10 inputs, suitable for the restaurant problem.

Learning algorithms for multilayer networks are similar to the perceptron learning algorithm show in Figure 20.21. One minor difference is that we may have several outputs, so we have an output vector $\mathbf{h_w}(x)$ rather than a single value, and each example has an output vector **y.** The major difference is that, whereas the error y- $\mathbf{h_w}$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data does not say what value the hidden nodes should have. It turns out that we can **back-propagate** the ell-or from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient. First, we will describe the process with an intuitive justification; then, we will show the derivation.

The back-propagation process can be summarized as follows:

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
  - Propagate the Δ values back to the previous layer.
  - Update the weights between the two layers.

The detailed algorithm is shown in Figure 20.25.

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with L layers, weights W_{j,i}, activation function g

  repeat
    for each e in examples do
      for each node j in the input layer do a_j ← x_j[e]
      for ℓ = 2 to L do
        in_i ← Σ_j W_{j,i} a_j
        a_i ← g(in_i)
      for each node i in the output layer do
        Δ_i ← g'(in_i) x (y_i[e] − a_i)
      for ℓ = L − 1 to 1 do
        for each node j in layer ℓ do
          Δ_j ← g'(in_j) Σ_i W_{j,i} Δ_i
          for each node i in layer ℓ + 1 do
            W_{j,i} ← W_{j,i} + α × a_j x Δ_i
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

Figure 20.25 The back-propagation algorithm for learning in multilayer networks.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 ,$$

The error can be back propagated through network:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j ,
\end{aligned}$$

Figure 20.26 shows single hidden layer network performs on the restaurant problem. In Figure 20.26, we show two curves. The first is a training curve, which shows the mean squared error on a given training set of 100 restaurant examples during the weight-updating process. This demonstrates that the network does indeed converge to a perfect fit to the training data. The second curve is the standard learning curve for the restaurant data. The neural network does learn well, although not quite as fast as decision-tree learning; this is perhaps not surprising, because the data were generated from a simple decision tree in the first place.

Advantage: Neural networks are capable of far more complex learning tasks of course, although it must be said that a certain amount of twiddling is needed to get the network structure right and to achieve convergence to something close to the global optimum in weight space.

Learning neural network structures: we also need to understand how to find the best network structure. If we choose a network that is too big, it will be able to memorize all the examples by forming a large lookup table, but will not necessarily generalize well to inputs that have not been seen before. In other words, like all statistical models, neural networks are subject to overfitting when there are too many parameters in the model.

If we stick to fully connected networks, the only choices to be made concern the number of hidden layers and their sizes. The usual approach is to try several and keep the best. The cross-validation techniques of Chapter 18 are needed if we are to avoid peeking at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies. The optimal brain damage algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.
Several algorithms have been proposed for growing a larger network from a smaller one. One, the tiling algorithm, resembles decision-list learning. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

VI. Language Processing and Present and Future of AI

An agent could communicate with another agent (human or software), using utterances in a common language. Complete syntactic and semantic analysis of the utterances is *necessary* to extract the full meaning of the utterances, and is *possible* because the utterances are short and restricted to a limited domain. Probabilistic language models deals with this.
It follows the **corpus-based** approach to language understanding. A corpus (plural *corpora)* is a large collection of text, such as the billions of pages that make up the World Wide Web.

Syntactic Analysis (Parsing): *Parsing can be seen as a process of searching for a parse tree.* There are two extreme ways of specifying the search space (and many variants in between). First, we can start with the *S* symbol and search for a tree that has the words as its leaves. This is called top-down parsing (because the S is drawn at the top of the tree). Second, we could start with the words and search for a tree with root S. This is called bottom-up parsing.

Top-down parsing can be precisely defined as a search problem as follows:
1) The initial state is a parse tree consisting of the root S and unknown children: [S: ?]. In general, each state in the search space is a parse tree.
2) The successor function selects the leftmost node in the tree with unknown children. It then looks in the grammar for rules that have the root label of the node on the left-hand side. For each such rule, it creates a successor state where the? is replaced by a list corresponding to the right-hand side of the rule. For example, in £0 there are two rules for S, so the tree *[S:?*] would be replaced by the following two successors:

[S: [S: ?] *[Conjunction:?]* [S: ?]]
[S: *[NP:?]* [*VP:* ?]]

The second of these has seven successors, one for each rewrite rule of *NP*.
3) The **goal test** checks that the leaves of the parse tree comespond exactly to the input string, with no unknowns and no uncovered inputs.
Disadvantage: One big problem for top-down parsing is dealing with so-called **left-recursive** rules

The formulation of bottom-up parsing as a search is as follows:
1) The **initial state** is a list of the words in the input string, each viewed as a parse tree that is just a single leaf node-for example; **[the, wumpus, is, dead].** In general, each state in the search space is a list of parse trees.
2) The **successor function** looks at every position ***i*** in the list of trees and at every righthand side of a rule in the grammar. If the subsequence of the list of trees starting at *i* matches the right-hand side, then the subsequence is replaced by a new tree whose category is the left-hand side of the rule and whose children are the subsequence. By "matches," we mean that the category of the node is the same as the element in the righthand side. For example, the rule *Article* + **the** matches the subsequence consisting of the first node in the list **[the, wumpus, is, clead],** so a successor state would be *[[Article:***the], wumpus, is, dead].**
3) The **goal test** checks for a state consisting of a single tree with root S.
See Figure 22.5 for an example of bottom-up parsing.

step	*list of nodes*	*subsequence*	*rule*
INIT	**the wumpus is dead**	**the**	Article → **the**
2	*Article* **wumpus is dead**	**wumpus**	Noun → **wumpus**
3	*Article Noun* **is dead**	*Article Noun*	*NP* → *Article Noun*
4	*NP* **is dead**	**is**	*Verb* → **is**
5	*NP Verb* **dead**	**dead**	*Adjective* → **dead**
6	**NP** *Verb Adjective*	*Verb*	*VP* → *Verb*
7	*NP VP Adjective*	*VP Adjective*	*VP* → *VP Adjective*
8	*NP VP*	*NP VP*	*S* → *NP VP*
GOAL	S		

Figure 22.5 Trace of a bottom up parse on the string "The wumpus is dead." We start with a list of nodes consisting of words. Then we replace subsequences that match the right-hand side of a rule with a new node whose root is the left-hand side. For example, in the third line the Article and Noun nodes are replaced by an N P node that has those two nodes as children. The top-down parse would produce a similar trace, but in the opposite direction.

Augmented Grammars: **Augment** the existing rules of the grammar instead of introducing new rules. First we will show
how a normal, unaugmented rule can be interpreted as a definite clause. We consider each category symbol to be a predicate on strings, so that N*P(s)* is true if the string *s* forms an *NP*. The CFG rule

$$S \rightarrow NP\ VP$$

is shorthand for the definite clause

$$NP(s_1)\ A\ VP(s_2) \Rightarrow S(s_1 + s_2)$$

The real benefit of the DCG approach is that we can *augment* the category symbols with additional arguments other than the string argument. For example, the rule

$$NP(case) \rightarrow Pronoun(case)$$

is shorthand for the definite clause

$$Pronoun(case, s_1) \Rightarrow NP(case, s_1)\ .$$

Definition of DCG rules:

- The notation $X \rightarrow Y\ Z\ \ldots$ translates as $Y(s_1)\ \text{A}\ Z(s_2)\ A \ldots \Rightarrow X(s_1 + s_2 + \ldots)$.
- The notation $X \rightarrow Y \mid Z \mid \ldots$ translates as $Y(s) \vee Z(s) \vee \ldots \Rightarrow X(s)$.
- In either of the preceding rules, any nonterminal symbol Y can be augmented with one or more arguments. Each argument can be a variable, a constant, or a function of arguments. In the translation, these arguments precede the string argument (e.g., *NP(case)*translates as $NP(case, s_1)$).
- The notation $\{P(\ldots)\}$ can appear on the right-hand side of a rule and translates verbatim into P(...). This allows the grammar writer to insert a test for P(...) without having the automatic string argument added.
- The notation $X \rightarrow$ **word** translates as $X([word])$.

Verb subcategorization: The idea is that the category *Verb* is broken into subcategories---one for verbs that have no object, one for verbs that take a single object, and so on.
Subcategorization list is the empty list, *[ ]* .The rules for verb subcategorization are:

$$
\begin{aligned}
VP(subcat) \rightarrow\ & Verb(subcat) \\
|\ & VP(subcat + [NP])NP(Objective) \\
|\ & VP(subcat + [Adjective])Adjective \\
|\ & VP(subcat + [PP])PP .
\end{aligned}
$$

The second step is to change the rule for *S* to say that it requires a verb phrase that has all its complements and thus has the subcat list []. Now the new rule is

$$S \rightarrow NP(Subjective)\ VP([])$$

Example: Figure 22.13 shows a parse tree using this grammar.

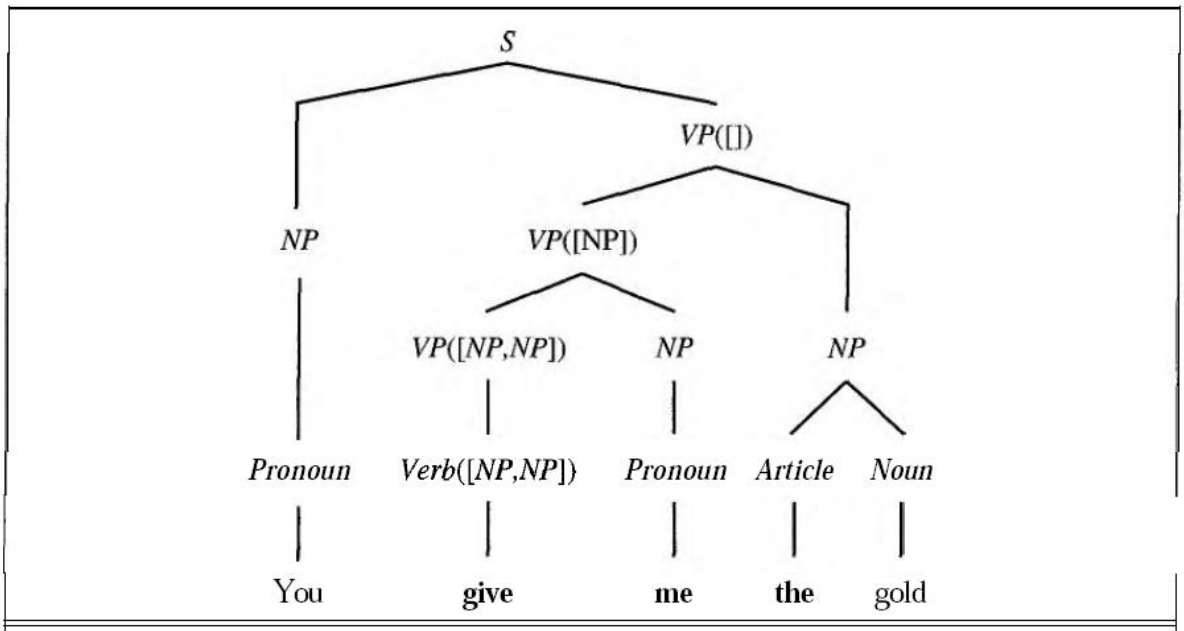


Figure **22.13** Parse tree for "You give me the gold" showing subcategorization of the verb and verb phrase.

Semantic Interpretation: Semantics is the extraction of the **meaning** of utterances.
Example: in logic, the meaning of P ∧ Q is determined by the meaning of P, Q, and ∧; in arithmetic, the meaning of x+ y is determined by the meaning of x, y, and +.

Figure 22.14 shows how DCG notation can be used to augment a grammar for arithmetic expressions.

$$Exp(x) \rightarrow Exp(x_1)\ Operator(op)\ Exp(x_2)\ \{x = Apply(op, x_1, x_2)\}$$
$$Exp(x) \rightarrow (\ Exp(x)\ )$$
$$Exp(x) \rightarrow Number(x)$$
$$Number(x) \rightarrow Digit(x)$$
$$Number(x) \rightarrow Number(x_1)\ Digit(x_2)\ \{x = 10 \times x_1 + x_2\}$$
$$Digit(x) \rightarrow x\ \{0 \leq x \leq 9\}$$
$$Operator(x) \rightarrow x\ \{x \in \{+, -, \div, \times\}\}$$

Figure 22.14 A grammar for arithmetic expressions, augmented with semantics. Each variable x_i represents the semantics of a constituent. Note the use of the $\{test\}$ notation to define logical predicates that must be satisfied, but that are not constituents.

Figure 22.15 shows shows the parse tree for **3** + (4 t 2) according to this grammar. The root of the parse tree is *Exp(5),* an expression whose semantic interpretation is 5.

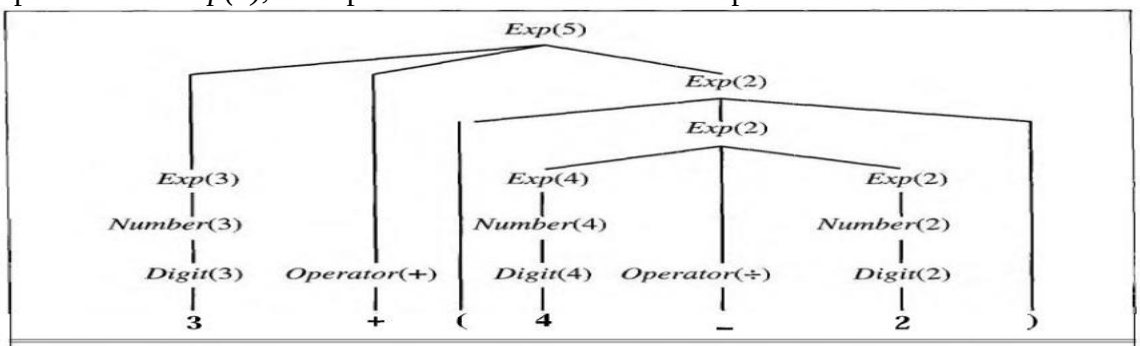


Figure 22.15 Parse tree with semantic interpretations for the string "3 + (4 ÷ 2)".

Pragmatic Interpretation: We have shown how an agent can perceive a string of words arid use a grammar to derive a set of possible semantic interpretations. Now we address the problem of completing the interpretation by adding context-dependent information about the current situation to each candidate interpretation. The most obvious need for pragmatic information is in resolving the meaning of indexicals, which are phrases that refer directly to the current situation. For example, in the sentence "I am in Boston today," the interpretations of the indexicals "I" and "today" depend on who uttered the sentence when. We represent indexicals by "constants" (such as *Speaker)* which actually are fluents-that is, they depend on the s-ituation. The hearer who perceives a speech act should also perceive who the speaker is and use this information to resolve the indexical.

Probabilistic Language Models:

A probabilistic language model defines a probability distribution over a (possibly infinite) set of strings. Example models are Unigram, Bigram and trigram etc.

A unigram model assigns a probability $P(w)$ to each word in the lexicon. The model assumes that words are chosen independently, so the probability of a string is just the product of the probability of its words, given by $\prod_i P(w_i)$.

The following 20-word sequence was generated at random from a unigram model of the words in this book:

logical are as are confusion a may right tries agent goal the was diesel more object then information-gathering search is

A bigram model assigns a probability $P(w_i|w_{i-1})$ to each word, given the previous word. Figure 15.21 listed some of these bigram probabilities. A bigram model of this book generates the following random sequence:

planning purely diagnostic expert systems are very similar computational approach would be represented compactly using tic tac toe a predicate

In general, an n-gram model conditions on the previous n – 1 words, assigning a probability for $P(w_i|w_{i-(n-1)} \ldots w_{i-1})$. A trigram model of this book generates this random sequence:

planning and scheduling are integrated the success of naive bayes model is just a possible prior source by that time

Another approach is **linear interpolation smoothing,** which combines trigram, bigram, and unigrani models by linear interpolation. We define our probability estimate as where c3+c2+c1=1.

Perplexity: It is the measure of the language models. We can compute the perplexity of a model on a test string of words.

$$Perplexity(words) = 2^{-\log_2(P(words))/N},$$

where N is the number of words. The lower the perplexity, the better the model.

Segmentation: The task of segmentation is finding the words boundaries in a text with no spaces. This task is necessary in Japanese and Chinese, languages that are written with no spaces between words, but we assume that most readers will be more comfortable with English. The sentence

Itiseasytoreadwordswithoutspaces

Figure 23.1 shows a version of the Viterbi algorithm specifically designed for the segmentation problem.

```
function VITERBI-SEGMENTATION(text, P) returns best words and their probabilities
  inputs: text, a string of characters with spaces removed
          P, a unigram probability distribution over words

  n ← LENGTH(text)
  words ← empty vector of length n + 1
  best ← vector of length n + 1, initially all 0.0
  best[0] ← 1.0
  /* Fill in the vectors best, words via dynamic programming */
  for i = 0 to n do
    for j = 0 to i − 1 do
      word ← text[j:i]
      w ← LENGTH(word)
      if P[word] x best[i- w] ≥ best[i] then
        best[i] ← P[word] × best[i − w]
        words[i] ← word
  /* Now recover the sequence of best words */
  sequence ← the empty list
  i ← n
  while i > 0 do
    push words[i] onto front of sequence
    i ← i − LENGTH(words[i])
  /* Return sequence of best words and overall probability of sequence */
  return sequence, best[i]
```

Figure **23.1** A Viterbi-based word segmentation algorithm. Given a string of words with spaces removed, it recovers the most probable segmentation into words.

It takes as input a unigram word probability distribution, P(word), and a string. Then, for each position *i* in the string, it stores in best[i] the probability of the most probable string spanning from the start up to ***i.*** It also stores in words[i] the word ending at position ***i*** that yielded the best probability. Once it has built up the best and words arrays in a dynamic programming fashion, it then works backwards through words to find the best path. In this case, with the unigram model from the book, the best sequence of words is indeed "It is easy to read words without spaces," with probability 10^{-25}.

Probabilistic context-free grammars:

n-gram models take advantage of co-occurrence statistics in the corpora, but they have no notion of grammar at distances greater than n. An alternative language model is the **prob**abilistic context-free grammar, or PCFG, which consists of a CFG wherein each rewrite rule has an associated probability. The sum of the probalbilities across all rules with the same left-hand side is 1. Figure 23.2 shows a PCFG for a portion of the *Eo(epsilon)* grammar.

$$
\begin{array}{rcl}
S & \rightarrow & NP\ VP\ [1.00] \\
NP & \rightarrow & Pronoun\ [0.10] \\
 & | & Name\ [0.10] \\
 & | & Noun\ [0.20] \\
 & | & Article\ Noun\ [0.50] \\
 & | & NP\ PP\ [0.10] \\
VP & \rightarrow & Verb\ [0.60] \\
 & | & VP\ NP\ [0.20] \\
 & | & VP\ PP\ [0.20] \\
PP & \rightarrow & Preposition\ NP\ [1.00] \\
Noun & \rightarrow & \textbf{breeze}\ [0.10] \mid \textbf{wumpus}\ [0.15] \mid \textbf{agent}\ [0.05] \mid \ldots \\
Verb & \rightarrow & \textbf{sees}\ [0.15] \mid \textbf{smells}\ [0.10] \mid \textbf{goes}\ [0.25] \mid \ldots \\
Pronoun & \rightarrow & \textbf{me}\ [0.05] \mid \textbf{you}\ [0.10] \mid \textbf{I}\ [0.25] \mid \textbf{it}\ [0.20] \mid \ldots \\
Article & \rightarrow & \textbf{the}\ [0.30] \mid \textbf{a}\ [0.35] \mid \textbf{every}\ [0.05] \mid \ldots \\
Preposition & \rightarrow & \textbf{to}\ [0.30] \mid \textbf{in}\ [0.25] \mid \textbf{on}\ [0.05] \mid \ldots
\end{array}
$$

Figure 23.2 A probabilistic context-free grammar (PCFG) and lexicon for a portion of the $\mathcal{E}_0$ grammar. The numbers in square brackets indicate the probability that a left-hand-side symbol will be rewritten with the corresponding rule.

In the PCFG model, the probability of a string, *P(words),* is just the sum of the probabilities of its parse trees. The probability of a given tree is the product of the probabilities of all the rules that make up the nodes of the tree. Figure: 23.3 shows how to compute the probability of a sentence. It is possible to compute this probability by using a CFG chart parser to enumerate the possible parses and then simply adding up the probabilities. However, if we are interested only in the most probable parse then enumerating the unlikely ones is wasteful. We can use a variation (of the Viterbi algorithm to find the most probable parse efficiently, or we can use a best-first search technique (such as A*).

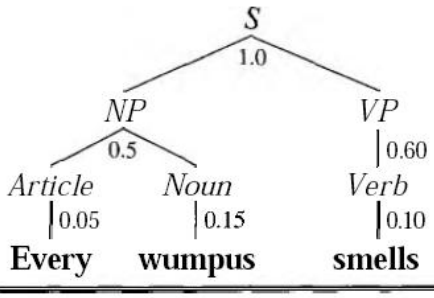


Figure 23.3 Parse tree for the sentence "Every wumpus smells," showing the probabilities of each subtree. The probability of the tree as a whole is 1.0 x 0.5 x 0.05 x 0.15 x 0.60 x 0.10 = 0.000225. Since this tree is the only parse of the sentence, that number is also the probability of the sentence.

Learning probabilities for PCFGs: To create a PCFG, we have all the difficulty of constructing a CFG, combined with the problem of setting the probabilities for each rule. This suggests that **learning** the grammar from data might be better than a knowledge engineering approach.

Learning rule structure for PCFGs:

Now suppose the structure of the grammar rules is not known. Our first problem is that the space of possible rule sets is infinite, so we don't know how many rules to consider nor how long each rule can be. One way to sidestep this problem is to learn a grammar in Chomsky normal form, which means that every rule is in one of the two forms

$$X \rightarrow Y\ Z$$
$$\mathbf{X} \rightarrow \mathbf{t}\ ,$$

where X, Y and Z .are nonterminals and t is a terminal. Any context-free grammar can be rewritten as a Chomsky normal form grammar that recognizes the exact same language.

An alternative approach called Bayesian model merging.The approach starts by building local models (grammars) of each sentence and then uses minimum description length to merge models.

Information retrieval:

Information retrieval is the task of finding documents that are relevant to a user's need for information. The best-known examples of information retrieval systems are search engines on the World Wide Web. A Web user can type a query such as [A1 book] into a search engine and see a list of relevant pages. In this section, we will see how such systems are built. An information retrieval (henceforth IR) system can be characterized by:

1. **A corpus of documents**. Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.
2. **Queries posed in a query language.** A query specifies what the user wants to know. The query language can be just a list of words, such as [AI book]; or it can specify
a phrase of words that must be adjacent, as in [“AI book”]; it can contain Boolean operators as in [AI AND book]; it can include non-Boolean operators such as [AI NEAR book] or [AI book site:www.aaai.org].
3. **A result set.** This is the subset of documents that the IR system judges to be relevant to the query. By relevant, we mean likely to be of use to the person who posed the query, for the particular information need expressed in the query.
4. **A presentation of the result set.** This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three dimensional space, rendered as a two-dimensional display.

There are two approaches in IR model:

i) Boolean Keyword model: Each word in the document collection is treated as a Boolean feature that is tnxe of ah document if the word occurs in the document and false if it does not. So the feature "retrieval" is true for the current chapter but false for Chapter 15. The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true. For example, the query [information AND retrieval] is true for the current chapter and false for Chapter 15.

Advantage: This model has the advantage of being simple to explain and implement.

Disadvantages: First, the degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation. Second, Boolean expressions are unfamiliar to users who are not programmers or logicians.

ii) Language Modelling: Language modeling approach, which estimates a language model for each document and then, for each query, computes the probability of the query, given the document's language model. Using *r* to denote the value *R = true,* we can rewrite the probability as follows:

$$\begin{aligned} P(r|D,Q) &= P(D,Q|r)P(r)/P(D,Q) && \text{(by Bayes' rule)} \\ &= P(Q|D,r)P(D|r)P(r)/P(D,Q) && \text{(by chain rule)} \\ &= \alpha\, P(Q|D,r)P(r|D)/P(D,Q) && \text{(by Bayes' rule, for fixed } D\text{)} . \end{aligned}$$

Evaluating IR System: How do we know whether an IR system is performing well? We undertake an experiment in which the system is given a set of queries and the result sets are scored with respect to human relevance judgments. Traditionally, there have been two measures used in the scoring: recall and precision. We explain them with the help of an example. Imagine that an IR system has returned a result set for a single query, for which we know which documents are and are not relevant, out of a corpus of 100 documents. The document counts in each category are given in the following table:

	In result set	Not in result set
Relevant	30	20
Not relevant	10	40

Precision measures the proportion of documents in the result set that are actually relevant. In our example, the precision is 30/(30 + 10)=.75. The false positive rate is 1 − .75=.25. **Recall** measures the proportion of all the relevant documents in the collection that are in the result set. In our example, recall is 30/ (30 + 20) =.60. The false negative rate is 1 −.60=.40.

IR Refinements:

Stemming: For example, if the query is [couch], it would be a shame to exclude from the result set those documents that mention "COUCH" or "couches" but not "couch." Most IR systems do **case folding** of "COUCH to "couch," and many use a **stemming** algorithm to reduce "couches" to the stem form "couch." This typically yields a small increase in recall.

Synonyms: The next step is to recognize **synonyms,** such as "sofa" for "couch." As with stemming, this has the potential for small gains in recall, but with a danger of decreasing precision if applied too aggressively. Those interested in football player Tim Couch would not want to wade through results about sofas.

Bigram: Many IR systems use word **bigrams** to some extent, although few implement a complete probabilistic bigram model.

Spelling Correction: Spelling correction routines can be used to correct for errors in both documents and queries.

Metadata: As a final refinement, IR can be improved by considering **metadata-data** outside of the text of the document. Examples include human-supplied keywords and hypertext links between documents.

Presentation and Implementation of IR system:

Document Classification and Clustering: An alternative approach is to present the result stet as a labeled tree rather than an ordered list. With document classification, the results are classified into a preexisting taxonomy of topics. For example, a collection of news stories might be classified into World News, Local news, Business, Entertainment, and Sports. With docuiment clustering, the tree of categories is created from scratch for each result set. Classification is appropriate when there are a small number of topics in a collection, and clustering is appropriate for broader collections such as the World Wide Web. In either case, when the user issues a query, the result set is shown organized into folders based on the categories.

Any classification or clustering algorithms can be used.

K-means clustering creates a flat set of exactly k categories. It works as follows:

1. Pick k documents at random to represent the k categories.
2. Assign every document to the closest category.
3. Compute the mean of each cluster and use the k means to represent the new values of the k categories.
4. Repeat steps (2) and (3) until convergence.

Information Extraction: Information extraction is the process of creating database entries by skimming a text and looking for occurrences of a particular class of object or event and for relationships among those objects and events.

Ex: We could be trying to extract instances of addresses from web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation.

Information extraction systems are mid-way between information retrieval systems and full-text parsers, in that they need to do more than consider a document as a bag of words, but less than completely analyze every sentence.

FASTUS: Information extraction systems often are built by using **cascaded finite-state transducers.** That is, they consist of a series of finite-state automata (FSAs), where each automaton receives text as input, transduces the text into a different format, and passes it along to the next automaton. This is appropriate because each FSA can be efficient and because together they can extract the necessary information.

A typical system is FASTUS, which consists of the following five stages:

1. Tokenization
2. Complex word handling
3. Basic group handling
4. Complex phrase handling
5. Structure merging

FASTUS's first stage is **tokenization,** which segments the stream of characters into tokens (words, numbers, and punctuation).

The second stage handles **complex words,** including collocations such as "set up" and "joint venture," as well as proper names such as "Prime Minister Tony Blair" and "Bridgestone Sports Co."

The third stage handles **basic groups,** meaning noun groups and verb groups. The idea is to chunk these into units that will be managed by the later stages.

Ex:

```
 1 NG: Bridgestone Sports Co.) 10 NG: a local concern
 2 VG: said                    11 CJ: and
 3 NG: Friday                  | 12 NG: a Japanese trading house
 4 NG: it                      | 13 VG: to produce
 5 VG: had set up              | 14 NG: golf clubs
 6 NG: a joint venture         | 15 VG: to be shipped
 7 PR: in                      | 16 PR: to
 8 NG: Taiwan                  | 17 NG: Japan
 9 PR: with
```

Here NG means noun group, VG is verb group, PR is preposition, and CJ is conjunction.

The fourth stage combines the basic groups into **complex phrases**
The final stage **merges structures** that were built up in the previous step.

Machine Translation:

Machine translation is the automatic translation of text from one natural language (the source) to another (the target). This process has proven to be useful for a number of tasks, including the following:

1. Rough translation, in which the goal is just to get the gist of a passage. Ungrammatical and inelegant sentences are tolerated as long as the meaning is clear.

2. Restricted-source translation, in which the subject rnatter and format of the source text are severely limited. One of the most successful examples is the TAUM-METEO system, which translates weather reports from English to French. It works because the language used in weather reports is highly stylized and regular.

3. Preedited translation, in which a human preedits the source document to make it conform to a restricted subset of English (or whatever tlhe original language is) before machine translation.

4. Literary translation, in which all the nuances of the source text are preserved. This is currently beyond the state of the art for machine translation.

Example: The SYSTRAN translation from Italian to English as follows:

Italian: In capitolo 22 abbiamo visto come un agente potrebbe comunicare con un altro agente (essere umano o software) che usando le espressioni in un linguaggio reciprocamente accordato. Completare sintattico e l'analisi semantica delle espressionii. necessaria da estrarre il significato completo del utterances ed è possibile perch6 le espressioni sono corte e limitate ad un settore limitato.	English: In chapter 22 we have seen as an agent could communicate with an other agent (to be human or software) that using the expressions in a language mutual come to an agreement. Complete syntactic and the semantic analysis of the expressions is necessary to extract the complete meant one of the utterances and is possible because the expressions short and are limited to a dominion.

Machine Translation Systems: Translation is difficult because, in the general case, it requires in-depth understanding of the text, and that in turn requires in-depth understanding of the situation that is being communicated. A representation language that makes all the distinctions necessary for a set of languages is called an **interlingua.**

A schematic diagram for Machine Translation System shown in Figure 23.12

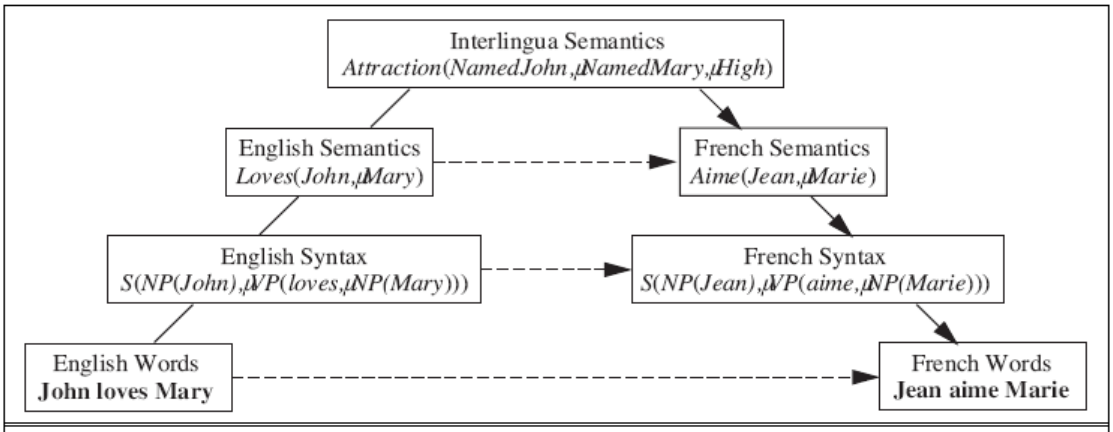


Figure 23.12 The Vauquois triangle: schematic diagram of the choices for a machine translation system (Vauquois, 1968). We start with English text at the top. An interlingua-based system follows the solid lines, parsing English first into a syntactic form, then into a semantic representation and an interlingua representation, and then through generation to a semantic, syntactic, and lexical form in French. A transfer-based system uses the dashed lines as a shortcut. Different systems make the transfer at different points; some make it at multiple points.

Statistical machine translation:

We can express the problem of translating an English sentence E into, say, a French sentence F by the following application of Bayes' rule::

$$\begin{aligned} \text{argmax}_F\, P(F|E) &= \text{argmax}_F\, P(E|F)P(F)/P(E) \\ &= \text{argmax}_F\, P(E|F)P(F)\ . \end{aligned}$$

This rule says we should consider all possible French sentences F and choose the one that maximizes the product P (E| F) P (F). The factor P (E) can be ignored because it is the same for every F. The factor P (F) is the **language model** for French; it says how probable a given sentence is in French. P (E1F) is the **translation model;** it says how probable an English sentence is as a translation, given a French sentence.

Learning probabilities for machine translation:

We have outlined a model for P(F |E) that involves four sets of parameters:

$$\begin{aligned} &\text{Language model: } P(word_i|word_{i-1}) \\ &\text{Fertility model: } P(Fertility = n|word_F) \\ &\text{Word choice model: } P(word_E|word_F) \\ &\text{Offset model: } P(Offset = o|pos, len_E, len_F) \end{aligned}$$

Segment into sentences: The unit of translation is a sentence, so we will have to break the corpus into sentences. Periods are strong indicators of the end of a sentence, but consider "Dr. J. R. Smith of Rodeo Dr. arrived."; only the final period ends a sentence. Sentence segmentation can be done with about 98% accuracy.

Estimate the French language model $P(word_i|word_{i-1})$: Considering just the French half of the corpus, count the frequencies of word pairs and do smoothing to give an estimate of $P(word_i|word_{i-1})$. For example we might have $P(\text{Eiffel}|\text{tour}) = .02$.

Estimate the initial fertility model $P(Fertility = n|word_F)$: Given a French sentence of length m that is aligned to an English sentence of length n, consider this as evidence that each French word has fertility n/m. Consider all the evidence over all sentences to get a fertility probability distribution for each word.

Estimate the initial word choice model $P(word_E|word_F)$: Look at all the French sentences that contain, say, "brun." The words that appear most frequently in the English sentences aligned with these sentences are the likely word-to-word translations of "brun."

Estimate the initial offset model $P(Offset = o|pos, len_E, len_F)$: Now that we have the word choice model, use it to estimate the offset model. For an English sentence of length n that is aligned to a French sentence of length m, look at each French word in the sentence (at position i) and at each English word in the sentence (at position j) that is a likely word choice for the French word, and consider that as evidence for $P(Offset = i - j|i, n, \mathbf{m})$.

Improve all the estimates: Use EM (expectation–maximization) to improve the estimates. The hidden variable will be a **word alignment vector** between sentence-aligned sentence pairs. The vector gives, for each English word, the position in the French sentence of the corresponding French word. For example, we might have the following:

Source French:	Le	chien	brun	n'	est	pas	all6	a	la	maison
Target English:	The	brown	dog	did	not		go			home
Word alignment:	1	3	2	5	4		7			10

Example:

Source French:	Le	chien	brun	n'	est	pas	all6	à	la	maison
Fertility model:	1	1	1	1	1	0	1	0	0	1
Transformed French:	Le	chien	brun	n'	est		all6			maison
Word choice model:	The	dog	brown	not	did		go			home
Offset model:	0	+1	−1	+1	−1		0			0
Target English:	The	brown	dog	did	not		go			home

Philosophical Foundations and AI Present and Future:

Weak AI: the assertion that machines could possibly act intelligently (or, WEAK **AI** perhaps better, act *as i f* they were intelligent) is called the **weak A1** hypothesis by philosophers.

Strong AI: the assertion that machines that do so are *actually* thinking (as opposed to *simulating* thinking) is called the **strong A1** hypothesis.

WEAK AI: CAN MACHINES ACT INTELLIGENTLY?

Some philosophers have tried to prove that A1 is impossible; that machines cannot possibly act intelligently. Some have used their arguments to call for a stop to A1 research:

The argument from disability:
The "argument from disability" makes the claim that "a machine can never do X." As examples of X, Turing lists the following:

Be kind, resourceful, beautiful, friendly, have initiative, have a sense of humor, tell right from wrong, make mistakes, fall in love, enjoy strawberries and cream, make someone fall in love with it, learn from experience, use words properly, be the subject of its own thought, have as much diversity of behavior as man, do something really new.

Russel and Norvig's Counter Part: Turing had to use his intuition to guess what would be possible in the future, but we have the luxury of looking back at what computers have already done. It is undeniable that computers now do many things that previously were the domain of humans alone. Programs play chess, checkers and other games, inspect parts on assembly lines, check the spelling of word processing documents, steer cars and helicopters, diagnose diseases, and do hundreds of other tasks as well as or better than humans. Computers have made small but significant discoveries in astronomy, mathematics, chemistry, mineralogy, biology, computer science, and other fields. Each of these required performance at the level of a human expert.

The mathematical objection: Certain mathematical questions are in principle unanswerable by particular formal systems. Godel's incompleteness theorem is the most famous example of this. Briefly, for any formal axiomatic system F powerful enough to do arithmetic, it is possible to construct a so-called "Godel sentence" *G(F)* with the following properties:

- *G(F)* is a sentence of F, but cannot be proved within F.
- If *F* is consistent, then *G(F)* is true.

Philosophers such as J. R. Lucas (1961) have claimed that this theorem shows that machines are mentally inferior to humans, because machines are formal systems that are limited by the incompleteness theorem.

Russel and Norvig's Counter Part: No human could compute the sum of 10 billion 10 digit numbers in his or her lifetime, but a computer could do it in seconds. Still, we do not see this as a fundamental limitation in the human's ability to think. Humans were behaving intelligently for thousands of years before they invented mathematics, so it is unlikely that mathematical reasoning plays more than a peripheral role in what it means to be intelligent.

An agent should not be too ashamed that it cannot establish the truth of some sentence while other agents can.

Most importantly, even if we grant that computers have limitations on what they can prove, there is no evidence that humans are immune from those limitations. It is all too easy to show rigorously that a formal system cannot do X, and then claim that humans can do X using their own informal method, without giving any evidence for this claim.

The argument from informality:
One of the most influential and persistent criticisms of A1 as an enterprise was raised by Turing as the "argument from informality of behavior." Essentially, this is the claim that human behavior is far too complex to be captured by any simple set of rules and that because computers can do no more than follow a set of rules, they cannot generate behavior as

intelligent as that of humans. The inability to capture everything in a set of logical rules is called the **qualification problem** in AI.

Russel and Norvig's Counter Part: Background commonsense knowledge, the qualification problem, uncertainty, learning, compiled forms of decision making, the importance of considering situated agents rather than disembodied inference engines-have by now been incorporated into standard intelligent agent design. In our view, this is evidence of AI's progress, not of its impossibility.

STRONG **AI:** CAN MACHINES REALLY THINK?

Many philosophers have claimed that a machine that passes the Turing Test would still not be *actually* thinking, but would be only a *simulation* of thinking. Again, the objection was foreseen by Turing.

The mind-body problem: The **mind-body problem** asks how mental states and processes are related to bodily (specifically, brain) states and processes. As if that wasn't hard enough, we will generalize the problem to the "mind-architecture" problem, to allow us to talk about the possibility of machines having minds.

The "brain in a vat" experiment:

Imagine, if you will, that your brain was removed from your body at birth and placed in a marvelously engineered vat. The vat sustains your brain, allowing it to grow and develop. At the same time, electronic signals are fed to your brain from a computer simulation of an entirely fictitious world, and motor signals from your brain are intercepted and used to modify the simulation as appropriate. Then the brain could have the mental state.

The brain prosthesis experiment: Suppose neurophysiology has developed to the point where the input-output behavior and connectivity of all the neurons in the human brain are perfectly understood. Suppose further that we can build microscopic electronic devices that mimic this behavior and can be smoothly interfaced to neural tissue. Lastly, suppose that some miraculous surgical technique can replace individual neurons with the corresponding electronic devices without interrupting the operation of the brain as a whole. The experiment consists of gradually replacing all the neurons in someone's head with electronic devices and then reversing the process to return the subject to his or her normal biological state.

It draws two conclusions:
1. The causal mechanisms of consciousness that generate these kinds of outputs in normal brains are still operating in the electronic version, which is therefore conscious.
2. The conscious mental events in the normal brain have no causal connection to behavior, and are missing from the electronic brain, which is therefore not conscious.

Chinee's Room Arguments:

The real claim made by Searle rests upon the following four axioms (Searle, 1990):

1. Computer programs are formal, syntactic entities.
2. Minds have mental contents, or semantics.
3. Syntax by itself is not sufficient for semantics.
4. Brains cause minds.

From the first three axioms he concludes that programs are not sufficient for minds. In other words, an agent running a program might be a mind, but it is not necessarily a mind just by

virtue of running the program. From the fourth axiom he concludes "Any other system capable of causing minds would have to have causal powers (at least) equivalent to those of brains." From there he infers that any artificial brain would have to duplicate the causal powers of brains, not just run a particular program, and that human brains do not produce mental phenomena solely by virtue of running a program.

THE ETHICS AND RISKS OF DEVELOPING ARTIFICIAL INTELLIGENCE:

If the effects of A1 technology are more likely to be negative than positive, then it would be the moral responsibility of workers in the field to redirect their research. Many new technologies have had unintended negative side-effects: the internal combustion engine brought air pollution.

All scientists and engineers face ethical considerations of how they should act on the job, what projects should or should not be done, and how they should be handled.

AI, however, seems to pose some fresh problems beyond that of, say, building bridges that don't fall down:

- People might lose their jobs to automation.
- People might have too much (or too little) leisure time.
- People might lose their sense of being unique.
- People might lose some of their privacy rights.
- The use of A1 systems might result in a loss of accountability.
- The success of A1 might mean the end of the human race.

AI Present and Future:

Agent Components:

Interaction with the environment through sensors and actuators: For much of the history of AI, this has been a glaring weak point. With a few honorable exceptions, **A1** systems were built in such a way that humans had to supply the inputs and interpret the outputs, while robotic systems focused on low-level tasks in which high-level reasoning and planning were largely absent. This was due in part to the great expense and engineering effort required to get real robots to work at all. The situation has changed rapidly in recent years with the availability of ready-made programmable robots, such as the four-legged robots.

Keeping track of the state of the world: This is one of the core capabilities required for an intelligent agent. It requires both perception and updating of internal representations.

Agent Architectures:

It is natural to ask, "Which of the agent architectures should an agent use?" The answer is, "All of them!" We have seen that reflex responses are needed for situations in which time is of the essence, whereas knowledge-based deliberation allows the agent to plan ahead.
A complete agent must be able to do both, using a **hybrid architecture.** One important property of hybrid architectures is that the boundaries between different decision components are not fixed. Example see Figure 27.2.

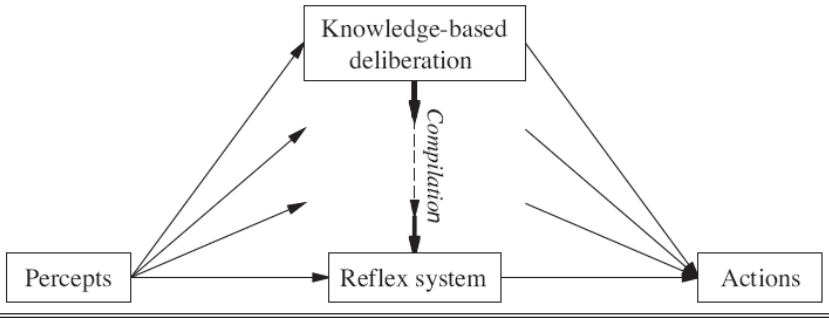


Figure 27.2 Compilation serves to convert deliberative decision making into more efficient, reflexive mechanisms.

Agents also need ways to control their own deliberations. They must be able to cease deliberating when action is demanded, and they must be able to use the time available for deliberation to execute the most profitable computations. For example, a taxi-driving agent that sees an accident ahead must decide in a split second either to brake or to take evasive action. It should also spend that split second thinking about the most important questions, such as whether the lanes to the left and right are clear and whether there is a large truck close behind, rather than worrying about wear and tear on the tires or where to pick up the next passenger. These issues are usually studied under the heading of **real-time AI**.

Two techniques can be used:

(1) **anytime algorithms:** An anytime algorithm is an algorithm whose output quality improves gradually over time, so that it has a reasonable decision ready whenever it is interrupted. Such algorithms are controlled by a **metalevel** decision procedure that assesses whether further computation is worthwhile. Example of an anytime algorithms include iterative deepening in game-tree search and MCMC in Bayesian networks.

(2) **decision-theoretic metareasoning:** This method applies the theory of information value to the selection of individual computations. The value of a computation depends on both its cost (in terms of delaying action) and its benefits (in terms of improved decision quality). Metareasoning techniques can be used to design better search algorithms and to guarantee that the algorithms have the anytime property. Metareasoning is expensive, of course, and compilation methods can be applied so that the overhead is small compared to the costs of the computations being controlled. Metalevel reinforcement learning may provide another way to acquire effective policies for controlling deliberation: in essence, computations that lead to better decisions are reinforced, while those that turn out to have no effect are penalized. This approach avoids the myopia problems of the simple value-of-information calculation.

Are we going in the right direction?
Now it is time to consider again what exactly the goal of AI is. We want to build agents, but with what specification in mind? Here are four possibilities.

Perfect rationality. A perfectly rational agent acts at every instant in such a way as to maximize its expected utility, given the information it has acquired from the environment. We have seen that the calculations necessary to achieve perfect rationality in most environments are too time consuming, so perfect rationality is not a realistic goal.

Calculative rationality. This is the notion of rationality that we have used implicitly in designing logical and decision-theoretic agents, and most of theoretical AI research has focused on this property. A calculative rational agent *eventually* returns what *would have been* the rational choice at the beginning of its deliberation. This is an interesting property for a system to exhibit, but in most environments, the right answer at the wrong time is of no value. In practice, AI system designers are forced to compromise on decision quality to obtain reasonable overall performance; unfortunately, the theoretical basis of calculative rationality does not provide a well-founded way to make such compromises.

Bounded Rationality: Bounded rationality works primarily by **satisficing**—that is, deliberating only long enough to come up with an answer that is "good enough." It is not a formal specification for intelligent agents, however, because the definition of "good enough" is not given by the theory. Furthermore, satisficing seems to be just one of a large range of methods used to cope with bounded resources.

Bounded optimality (BO). A bounded optimal agent behaves as well as possible, *given its computational resources*. That is, the expected utility of the agent program for a bounded optimal agent is at least as high as the expected utility of any other agent program running on the same machine.

Of these four possibilities, bounded optimality seems to offer the best hope for a strong theoretical foundation for AI. It has the advantage of being possible to achieve: there is always at least one best program—something that perfect rationality lacks. Bounded optimal agents are actually useful in the real world, whereas calculative rational agents usually are not, and satisficing agents might or might not be, depending on how ambitious they are.

What if AI Does Succeeded?

Intelligent computers are more powerful than dumb ones, but will that power be used for good or ill? Those who strive to develop AI have a responsibility to see that the impact of their work is a positive one. The scope of the impact will depend on the degree of success of AI. Even **modest successes** in AI have already changed the ways in which computer science is taught. AI has made possible new applications such as speech recognition systems, inventory control systems, surveillance systems, robots, and search engines.

We can expect that **medium-level successes** in AI would affect all kinds of people in their daily lives. So far, computerized communication networks, such as cell phones and the Internet, have had this kind of pervasive effect on society, but AI has not. AI has been at work behind the scenes—for example, in automatically approving or denying credit card transactions for every purchase made on theWeb—but has not been visible to the average consumer.

Finally, it seems likely that a **large-scale success** in AI—the creation of human-level intelligence and beyond—would change the lives of a majority of humankind. The very nature of our work and play would be altered, as would our view of intelligence, consciousness, and the future destiny of the human race. AI systems at this level of capability could threaten human autonomy, freedom, and even survival. For these reasons, we cannot divorce AI research from its ethical consequences.