

ARTIFICIAL INTELLIGENCE

UNIT-I

Syllabus

Introduction: What is AI, Foundations of AI, History of AI, The State of Art.

Intelligent Agents: Agents and Environments, Good Behavior: The Concept of Rationality,
The Nature of Environments, The Structure of Agents.

CHAPTER-1 INTRODUCTION

- ✓ The field of artificial intelligence, or AI, goes further still: it attempts not just to understand but also to *build* intelligent entities.
- ✓ AI is one of the newest sciences. Work started in earnest soon after World War II, and the name itself was coined in 1956.
- ✓ Along with molecular biology, AI is regularly cited as the "field I would most like to be in" by scientists in other disciplines.
- ✓ A student in physics might reasonably feel that all the good ideas have already been taken by Galileo, Newton, Einstein, and the rest. AI, on the other hand, still has openings for several full-time Einsteins.
- ✓ AI systematizes and automates intellectual tasks and is therefore potentially relevant to any sphere of human intellectual activity. In this sense, it is truly a universal field.

1.1.1. What is AI?

Thinking Humanly "The exciting new effort to make computers think . . . <i>machines with minds</i> , 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)	Thinking Rationally "The study of mental faculties through the use of computational models." (Chamiak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly "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)	Acting Rationally "Computational Intelligence is the study of the design of intelligent agents." (Poole <i>et al.</i> , 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

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

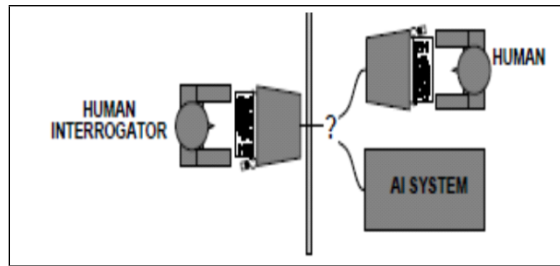
- ✓ Views of AI fall into four categories:

Thinking Humanly	Thinking Rationally
Acting Humanly	Acting Rationally

- ✓ Note: A system is rational if it does the "right

A. Acting Humanly : The Turing Test Approach

- ✓ Alan Turing (1950) - "Computing machinery and intelligence":
- ✓ Can machines think? Can machines behave intelligently?
- ✓ Operational test for intelligent behavior: the Imitation Game.



- ✓ Predicted that by 2000, a machine might have a 30% chance of fooling a lay person for 5 minutes.
- ✓ Anticipated all major arguments against AI in following 50 years.
- ✓ Suggested major components of AI: knowledge, reasoning, language understanding, learning, computer vision & Robotics.
- **Problem:** Turing test is not reproducible, constructive, or amenable to mathematical analysis.

B. Thinking humanly: The Cognitive modeling approach

- ✓ 1960s - cognitive revolution": information processing psychology replaced prevailing orthodoxy of behaviorism".
- ✓ Requires scientific theories of internal activities of the brain.
- ✓ What level of abstraction? Knowledge" or circuits"?
- **What is cognitive science??**

The study of thought, learning, and mental organization, which draws on aspects of psychology, linguistics, philosophy, and computer modeling.

- **How to validate? Requires**
 - 1) Predicting and testing behavior of human subjects (top-down) or
 - 2) Direct identification from neurological data (bottom-up).
- ✓ Both approaches (roughly, Cognitive Science and Cognitive Neuroscience) are now distinct from AI

C. Thinking rationally : The “laws of thought” approach

- ✓ Normative (or prescriptive) rather than descriptive
- ✓ Aristotle: what are correct arguments/thought processes?
- ✓ Several Greek schools developed various forms of logic: **notation and rules of derivation for thoughts;**
- ✓ May or may not have proceeded to the idea of mechanization.
- ✓ Direct line through mathematics and philosophy to modern AI
- **Problems:**
 - 1) Not all intelligent behavior is mediated by logical deliberation
 - 2) What is the purpose of thinking? What thoughts should I have?

D. Acting rationally : The rational agent approach

- ✓ Rational behavior: doing the right thing
- ✓ The right thing: that which is expected to maximize goal achievement, given the available information.
- ✓ Doesn't necessarily involve thinking e.g., blinking reflex, but thinking should be in the service of

- ✓ Aristotle (Nicomachean Ethics): □Every art and every inquiry, and similarly every action and pursuit, is thought to aim at some good.

1.1.2. Foundations of AI...

The different foundations which are contributors for AI are:

- Philosophy
- Mathematics
- Economics
- Neuroscience
- Psychology
- Computer engineering
- Control theory and cybernetics
- Linguistics

i. Philosophy(428 B .C .-present):

- ✓ Can formal rules be used to draw valid conclusions?
- ✓ How does the mental mind arise from a physical brain?
- ✓ Where does knowledge come from?
- ✓ How does knowledge lead to action?
- ✓ Aristotle (384-322 B.C.) was the first to formulate a precise set of laws governing the rational part of the mind.

ii. Mathematics(B.C 800-present):

- ✓ What are the formal rules to draw valid conclusions?
- ✓ What can be computed?
- ✓ How do we reason with uncertain information?
- ✓ Besides logic and computation, the third great contribution of mathematics to AI is PROBABILITY - the theory of probability.
- ✓ The Italian Gerolamo Cardano (1501-1576) first framed the idea of probability, describing it in terms of the possible outcomes of gambling events.
- ✓ Thomas Bayes (1702-1761) proposed a rule for updating probabilities in the light of new evidence.
- ✓ Bayes' rule and the resulting field called Bayesian analysis form the basis of most modern approaches to uncertain reasoning in AI systems.

iii. Economics (1776-present):

- ✓ 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 fix in the future?
- ✓ Work in economics and operations research has contributed much to our notion of rational Agents.
- ✓ Herbert Simon (1916-2001), the pioneering AI researcher, won the Nobel prize in economics in 1978 for his early SATISFICING work showing that models based on satisficing-making decisions that are "good

enough“.

✓

iv Neuroscience (1861-present):

- ✓ How do brains process information?
- ✓ Neuroscience is the study of the nervous system, particularly the brain.
- ✓ Paul Broca's (1824-1880) study of aphasia (speech deficit) in brain-damaged patients.
- ✓ He proved speech production was localized to a portion of the left hemisphere now called NEURONS Broca's area.
- ✓ Carnillo Golgi (1843-1926) developed a staining technique allowing the observation of individual neurons in the brain.
- ✓ The recent development of functional magnetic resonance imaging (MRI) (Ogawa et al., 1990) is giving neuroscientists unprecedentedly detailed images of brain activity.

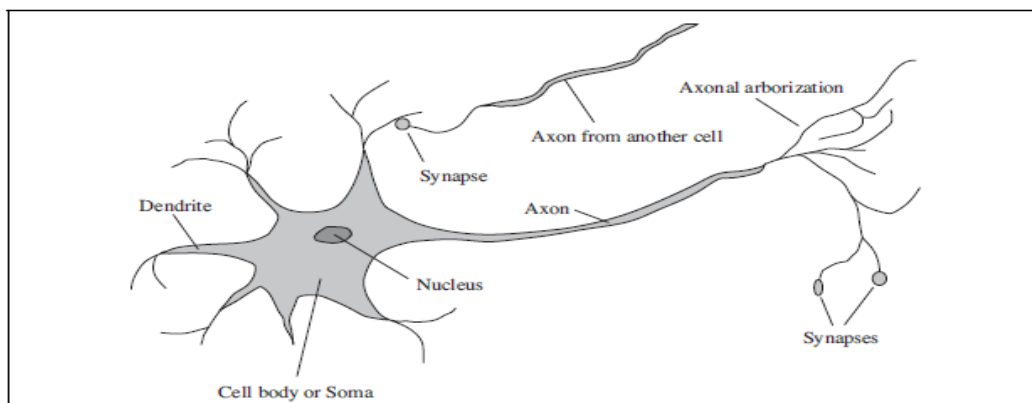


Figure 1.2 The parts of a nerve cell or neuron. Each neuron consists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. The axon stretches out for a long distance, much longer than the 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).

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

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

- **NOTE :** Moore's Law predicts that the CPU's gate count will equal the brain's neuron count around 2020.

v Psychology (1879-present):

- ✓ How do humans and animals think and act?
- ✓ The origin of scientific psychology are traced back to the work of German physiologist Hermann von Helmholtz(1821-1894) and his student Wilhelm Wundt(1832 – 1920).
- ✓ In 1879,Wundt opened the first laboratory of experimental psychology at the university of Leipzig.
- ✓ In US, the development of computer modeling led to the creation of the field of cognitive science.
- ✓ The field can be said to have started at the workshop in September 1956 at MIT.

vi Computer Engineering (1940-present):

- ✓ How can we build an efficient computer?
- ✓ For artificial intelligence to succeed, we need two things: intelligence and an artifact.
- ✓ The computer has been the artifact of choice.
- ✓ AI also owes a debt to the software side of computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs.

vii Control theory and Cybernetics (1948-present):

- ✓ How can artifacts operate under their own control?
- ✓ Ktesibios of Alexandria (c. 250 B.C.) built the first self-controlling machine: a water clock.
- ✓ It contains regulator that kept the flow of water running through it at a constant, predictable pace.
- ✓ Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of
- ✓ systems that maximize an objective function over time.

Viii Linguistics:

- ✓ How does language relate to thought?
- ✓ Modern linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called computational linguistics or natural language processing.

1.1.3. History of AI...

A. Gestation, 1943–56:

- ✓ McCulloch & Pitts (1943)
Artificial neural net—proved equivalent to Turing machine;
- ✓ Shannon, Turing (1950)
Chess playing programs
- ✓ Marvin Minsky (1951)
First neural net computer—SNARC
- ✓ Dartmouth College (1956)
Term “AI” coined by John McCarthy
- ✓ Newell & Simon presented LOGIC THEORIST program

B. Early enthusiasm, 1952–69:

- ✓ Lots of progress.
- ✓ Programs written that could:
 - plan, learn
 - play games,
 - prove theorems, in general, solve problems.
- ✓ Major feature of the period were microworlds—toy problem domains.

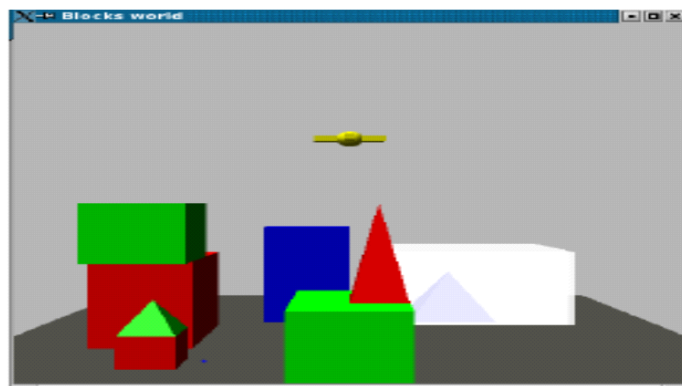
Major centres established:



Minsky, MIT
McCarthy, Stanford
Newell & Simon, CMU

For the next 20 years these places dominated the field.

- ✓ Example: blocks world.
- ✓ Planners simulated manipulating simple shapes like this:



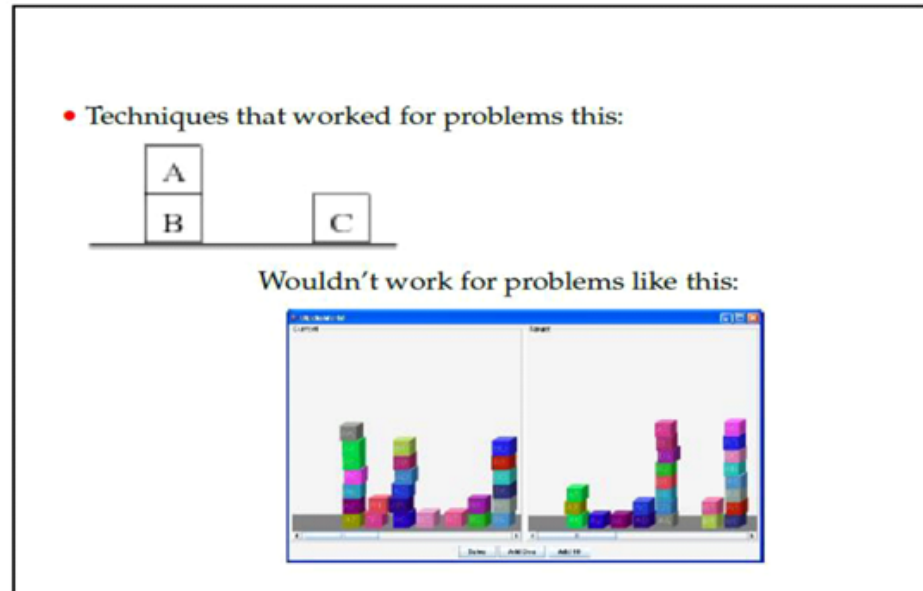
- ✓ Even some simple natural language handling:

```
Person: GRASP THE PYRAMID.
Computer: I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.
Person: FIND A BLOCK WHICH IS TALLER THAN THE ONE
        YOU ARE HOLDING AND PUT IT INTO THE BOX.
Computer: BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH
        IS TALLER THAN THE ONE I AM HOLDING.
Computer: OK. (does it)
Person: WHAT DOES THE BOX CONTAIN?
Computer: THE BLUE PYRAMID AND THE BLUE BLOCK.
Person: WHAT IS THE PYRAMID SUPPORTED BY?
Computer: THE BOX.
Person: HOW MANY BLOCKS ARE NOT IN THE BOX?
Computer: FOUR OF THEM.

• The promise was "It'll scale, honest..."
```

C. A dose of reality, 1966–1973:

- ✓ Techniques developed on microworlds would not scale.
- ✓ Implications of complexity theory—developed in late 1960s, early 1970s—began to be appreciated:
 - brute force techniques will not work.
 - works in principle does not mean works in practice.
- ✓ Lots of early programs did symbol manipulation without any real understanding of the domain they were in.



D. Knowledge based systems, 1969–1979:

- ✓ General purpose, brute force techniques don't work, so use knowledge rich solutions.
- ✓ Early 1970s saw emergence of expert systems as systems capable of exploiting knowledge about tightly focused domains to solve problems normally considered the domain of experts.
 - ✓ Ed Feigenbaum's knowledge principle: [Knowledge] is power, and computers that amplify that knowledge will amplify every dimension of power.
 - ✓ Expert systems success stories:
 - MYCIN — blood diseases in humans;
 - DENDRAL — interpreting mass spectrometers;
 - R1/XCON — configuring DEC VAX hardware;
 - PROSPECTOR — finding promising sites for mineral deposits;
 - ✓ Expert systems emphasised knowledge representation: rules, frames, semantic nets.
- **Problems:**
 - the knowledge elicitation bottleneck;
 - marrying expert system & traditional software;
 - breaking into the mainstream.

E. AI makes money, 1980–present:

- ✓ R1 was the first commercial expert system.
- ✓ Led to a boom in expert systems companies.

- Like an earlier dot com boom
- ✓ Most of those companies failed to deliver increased productivity for their customers.
- ✓ They went bust.
- ✓ AI suffered from all the broken promises, but didn't go away.

F. AI and the scientific method, 1987– present:



- ✓ Connection back to other fields.
 - Probability/statistics
 - Control theory
 - Economics
 - Operations research.
- ✓ New industries spawned.
 - Data mining is machine learning.
- ✓ Low-key successes.
 - Expert systems embedded in Windows.
- ✓ Increased emphasis on shared datasets, common problems, competitions.

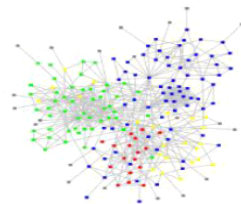
G. Intelligent agents, 1993–present:

- ✓ Emphasis on understanding the interaction between agents and environments.
- ✓ AI as component, rather than as end in itself.
 - “Useful first” paradigm — Etzioni (NETBOT, US\$35m)
 - “Raisin bread” model — Winston.
- ✓ More about interaction between components, emergent intelligence, and doing well enough to be useful.



H. Large datasets, 2001–present:

- ✓ Change in emphasis from algorithm to data.
 - Perhaps with enough data, we don't have to be smart to be good.
- ✓ For example, how to fill in a gap in a picture.
 - Perhaps you have a picture of your ex in front of a great landscape.
- ✓ Poll lots of similar pictures for a matching piece.
 - Moving from 10,000 images to 2,000,000 led to a big jump in performance.
- ✓ Wisdom of the crowd.
- ✓ The growth of the internet makes this much easier in 2010 than it was in 1990.



- ✓ Currently a lot of interest in mining data from social networks.
- ✓ If people are connected, they are similar in some sense.

1.1.4. THE STATE OF THE ART:

Here we sample a few applications;

1. **Robotic vehicles:** A driverless robotic car named STANLEY sped through the rough terrain of the Mojave desert at 22 mph, finishing the 132-mile course first to win the 2005 DARPA Grand Challenge.
2. **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.
3. **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.
4. **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.
5. **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.
6. **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.
7. **Robotics:** The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use.
8. **Machine Translation:** A computer program automatically translates from Arabic to English.

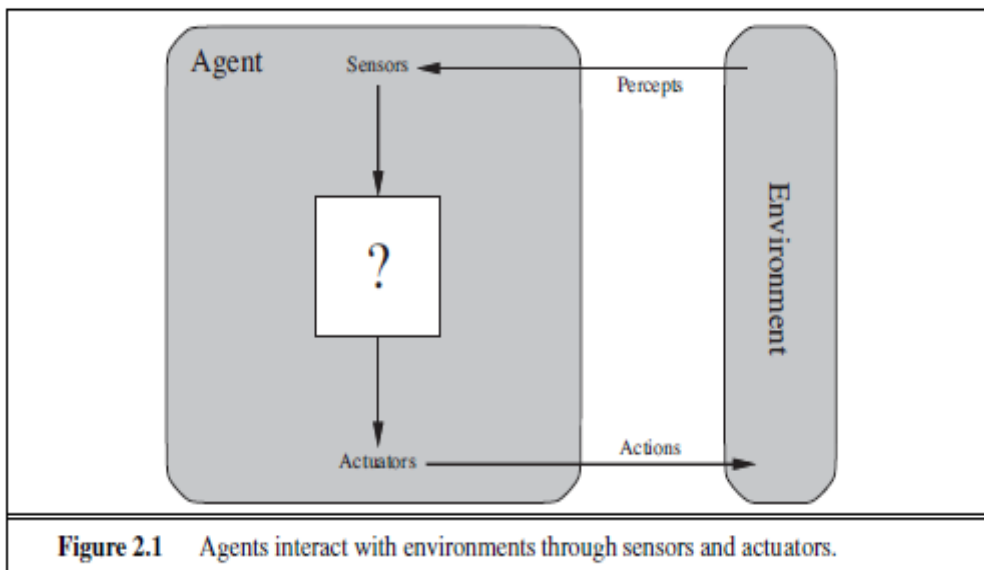
1.2.1. AGENTS AND ENVIRONMENT :

➤ Agent Definition...

An agent is a system that:

- is situated in an environment,
 - is capable of perceiving its environment, and
 - is capable of acting in its environment with the goal of satisfying its design objectives.
- (or)

Anything that can be viewed as perceiving its environment through sensors and SENSOR acting upon that environment through actuators.



(OR)

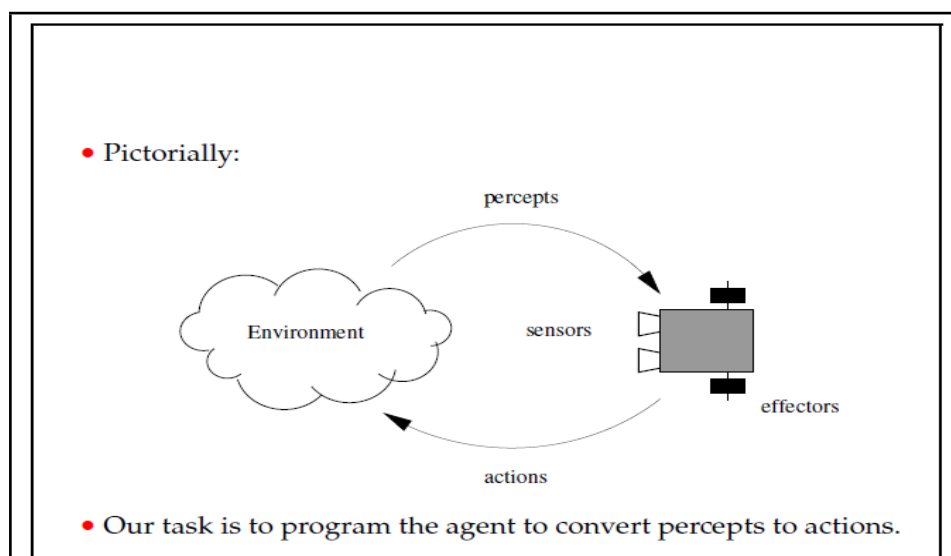


Figure : An agent example representation

➤ **Examples for different agents:**

i. **Human “agent”:**

- environment: physical world;
- sensors: eyes, ears, . . .
- effectors: hands, legs, . . .

ii. **Software agent:**

- environment: (e.g.) UNIX operating system;
- sensors: keystrokes, file contents, n/w packets . . .
- effectors: rm, chmod, . . .

iii. **Robot:**

- environment: physical world;
- sensors: sonar, camera;
- effectors: wheels.

➤ **Important terms...**

✓ **Percept:**

We use the term percept to refer to the agent's perceptual inputs at any given instant.

✓ **Percept Sequence:**

An agent's percept sequence is the complete history of everything the agent has ever perceived.

✓ **Agent function: $f:P^* \rightarrow A$**

Mathematically speaking, we say that an agent's behavior is described by the agent function that **maps any given percept sequence to an action**.

➤ **Agent function VS Agent program:**

The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

Example: The vacuum-cleaner world...

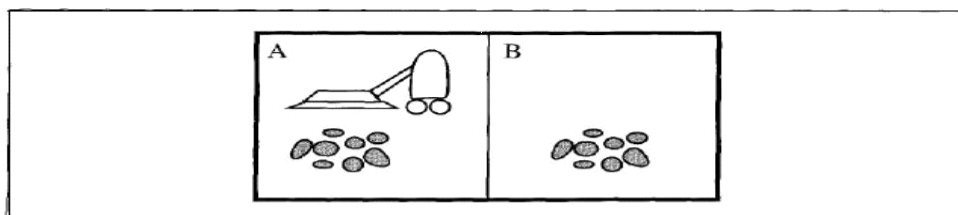


Figure 2.2 A vacuum-cleaner world with just two locations.

Percept sequence	Action
<i>[A, Clean]</i>	<i>Right</i>
<i>[A, Dirty]</i>	<i>Suck</i>
<i>[B, Clean]</i>	<i>Left</i>
<i>[B, Dirty]</i>	<i>Suck</i>
<i>[A, Clean], [A, Clean]</i>	<i>Right</i>
<i>[A, Clean], [A, Dirty]</i>	<i>Suck</i>
<i>[A, Clean], [A, Clean], [A, Clean]</i>	<i>Right</i>
<i>[A, Clean], [A, Clean], [A, Dirty]</i>	<i>Suck</i>

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

1.2.2 Good behavior – Concept of Rationality...

➤ Rational Agent...

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.

Obviously, doing the right thing is better than doing the wrong thing. The right action is the one that will cause the agent to be most successful.

➤ Performance measures:

A performance measure embodies the criterion for success of an agent's behavior.

When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.

- ✓ This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well.

➤ Rationality :

What is 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.

➤ Omniscience, learning, and autonomy...

- ✓ An omniscient agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.
- ✓ Doing actions in order to modify future percepts-sometimes called information gathering-is an important part of rationality.
- ✓ Our definition requires a rational agent not only to gather information, but also to learn as much as possible from what it perceives.
- ✓ To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy. A rational agent should be autonomous-it should learn what it can to compensate for partial or incorrect prior knowledge.

1.2.3 The Nature of Environments :

- ✓ Task environments :

These are essentially the "problems" to which rational agents are the "solutions."

- ✓ Specifying the task environment : PEAS

The rationality of the simple vacuum-cleaner agent, needs specification of ...

- the performance measure - P
- the environment - E
- the agent's actuators and sensors - A & S

PEAS Example...

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.

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.

➤ Properties of task environments...

- I. Fully observable vs. partially observable
- II. Deterministic vs. stochastic
- III. Episodic vs. sequential
- IV. Static vs. dynamic
- V. Discrete vs. continuous
- VI. Single agent vs. Multiagent

I. Fully observable vs. partially observable...

- ✓ If an agent's sensors give it access to the 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;
- ✓ An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor.

II. Deterministic vs. stochastic...

If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; other-wise, it is stochastic.

III. Episodic vs. sequential...

- ✓ In an episodic task environment, the agent's experience is divided into atomic episodes.
- ✓ Each episode consists of the agent perceiving and then performing a single action.
- ✓ Crucially, the next episode does not depend on the actions taken in previous episodes.
- ✓ In sequential environments, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential.

IV. Static vs. Dynamic...

- ✓ If the environment can change while an agent is deliberating, then 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; if it hasn't decided yet, that counts as deciding to do nothing.
- ✓ If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**. Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next. Chess, when played with a clock, is semidynamic. Crossword puzzles are static.

V. Discrete vs. continuous...

- ✓ A discrete-state environment such as a chess game has a finite number of distinct states.
- ✓ Chess also has a discrete set of percepts and actions.
- ✓ Taxi driving is a continuous state and continuous-time problem: the speed and location of the taxi and of the other
- ✓ vehicles sweep through a range of continuous values and do so smoothly over time.
- ✓ Taxi-driving actions are also continuous (steering angles,.)

VI. Single agent vs. multiagent...

- ✓ An agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent

playing chess is in a two-agent environment.

- ✓ As one might expect, the hardest case is partially observable, stochastic, sequential, dynamic, continuous, and Multiagent.

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.

1.2.4 The Structure of Agents...

➤ Agent programs:

- ✓ The job of AI is to design the agent program that implements the agent function mapping percepts to actions.
- ✓ Agent = architecture + program
- ✓ The agent programs all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators.
- ✓ Notice the difference between the agent program, which takes the current percept as input, and the agent function, which takes the entire percept history.

➤ Agent Types/Categories...

i. Table-driven agents:

-- use a percept sequence/action table in memory to find the next action. They are implemented by a (large) lookup table.

ii. Simple reflex agents:

-- are based on condition-action rules, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.

iii. Agents with memory (Model):

-- have internal state, which is used to keep track of past states of the world.

iv. Agents with goals:

-- are agents that, in addition to state information, have goal information that describes desirable situations. Agents of this kind take future events into consideration.

v. Utility-based agents:

--base their decisions on classic axiomatic utility theory in order to act rationally.

vi. Learning Agents:

i. Table Driven Agent:

```
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.

➤ Drawbacks:

- ✓ Table lookup of percept-action pairs defining all possible condition-action rules necessary to interact in an environment.

➤ Problems :

- Too big to generate and to store (Chess has about 10^{120} states, for example)
- No knowledge of non-perceptual parts of the current state
- Not adaptive to changes in the environment; requires entire table to be updated if changes occur
- Looping: Can't make actions conditional
 - Take a long time to build the table
 - No autonomy
 - Even with learning, need a long time to learn the table entries

ii. Simple Reflex Agent:

- ✓ 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. E.g. the vacuum-agent
- ✓ Large reduction in possible percept/action situations.
- ✓ Implemented through condition-action rules.
- ✓ Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking.
- ✓ In other words, some processing is done on the visual input to establish the condition we call “The car in front is braking.” Then, this triggers some established connection in the agent program to the action “initiate braking.”
- ✓ We call such a connection a **condition–action rule**,⁵ written as **if** *car-in-front-is-braking* **then** *initiate-braking*.
- ✓ **Example : Vacuum Cleaner : If dirty then suck**

```

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.

```

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.

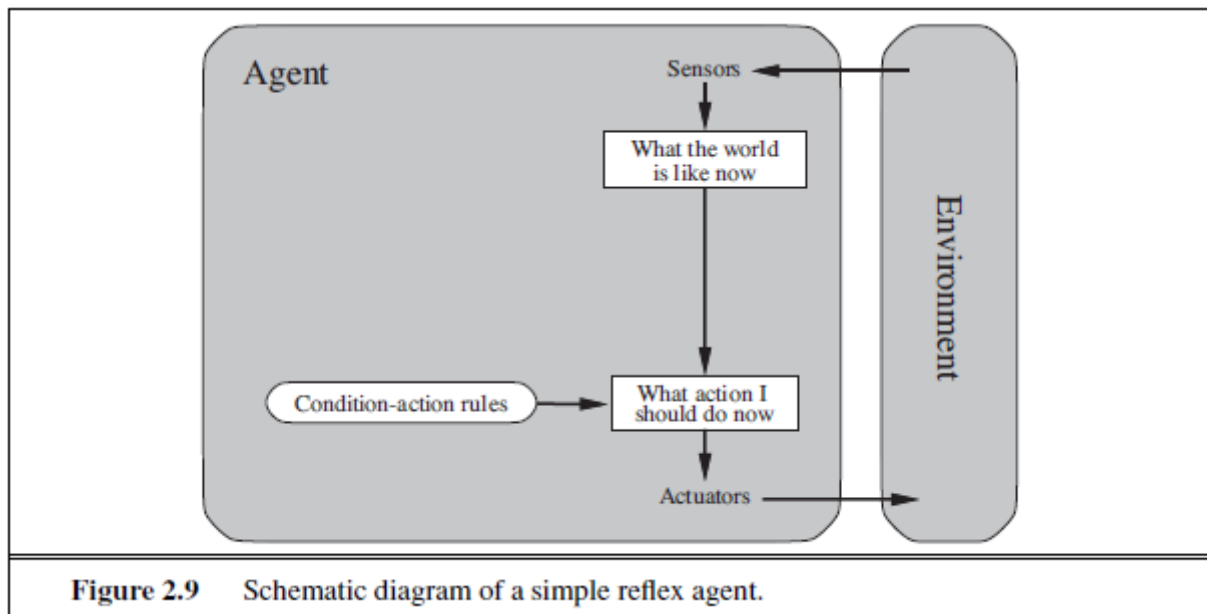


Figure 2.9 Schematic diagram of a simple reflex agent.

➤ Characteristics:

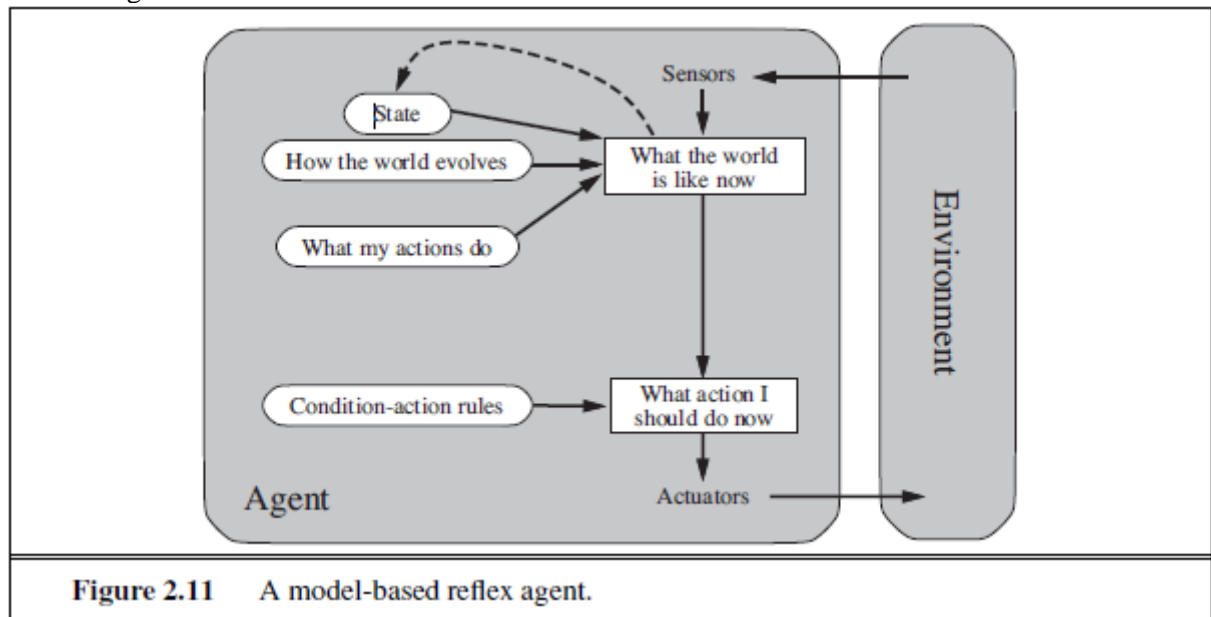
- ✓ Only works if the environment is fully observable.
- ✓ Lacking history, easily get stuck in infinite loops.
- ✓ One solution is to randomize actions.

iii. 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.
- ✓ Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the

agent program.

- ✓ First, we need some information about how the world evolves independently of the agent-for example, that an overtaking car generally will be closer behind than it was a moment ago.
- ✓ Second, we need some information about how the agent's own actions affect the world-for example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago.
- ✓ This knowledge about "how the world works"-whether implemented in simple Boolean circuits or in complete scientific theories-is called a model of the world. An agent that uses such a model is called a model-based 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

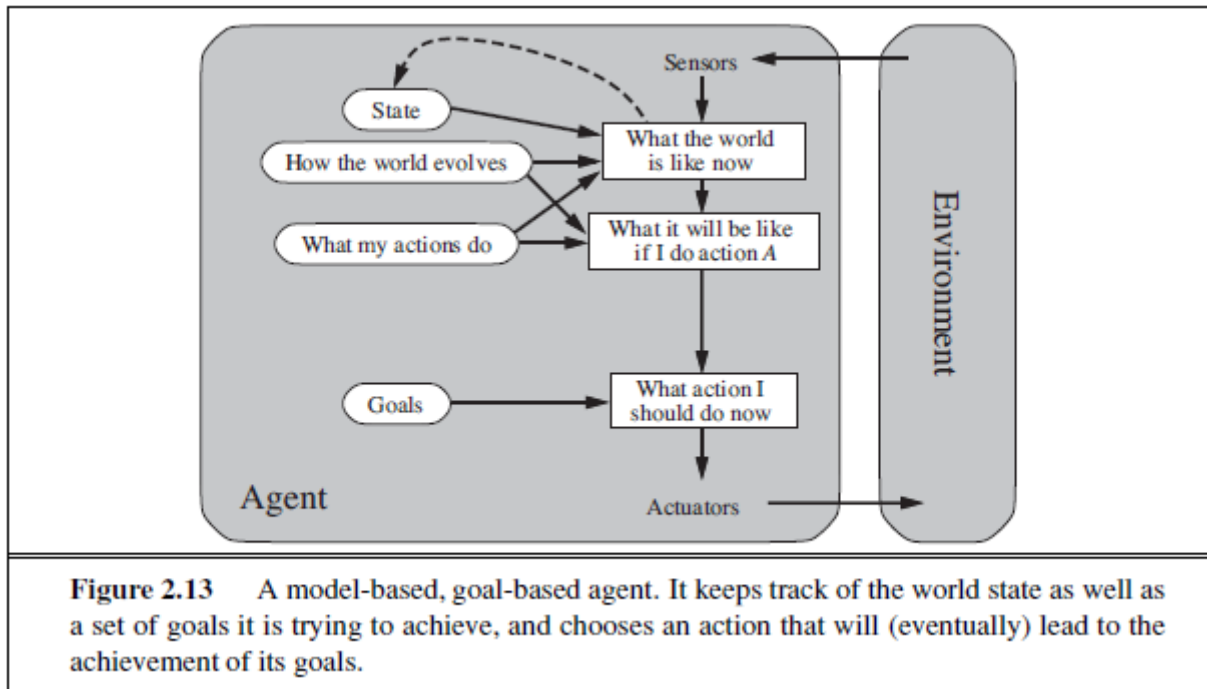
  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

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.

iv Goal Based Agents:

- ✓ Knowing 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 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 information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal.
- ✓ Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal.

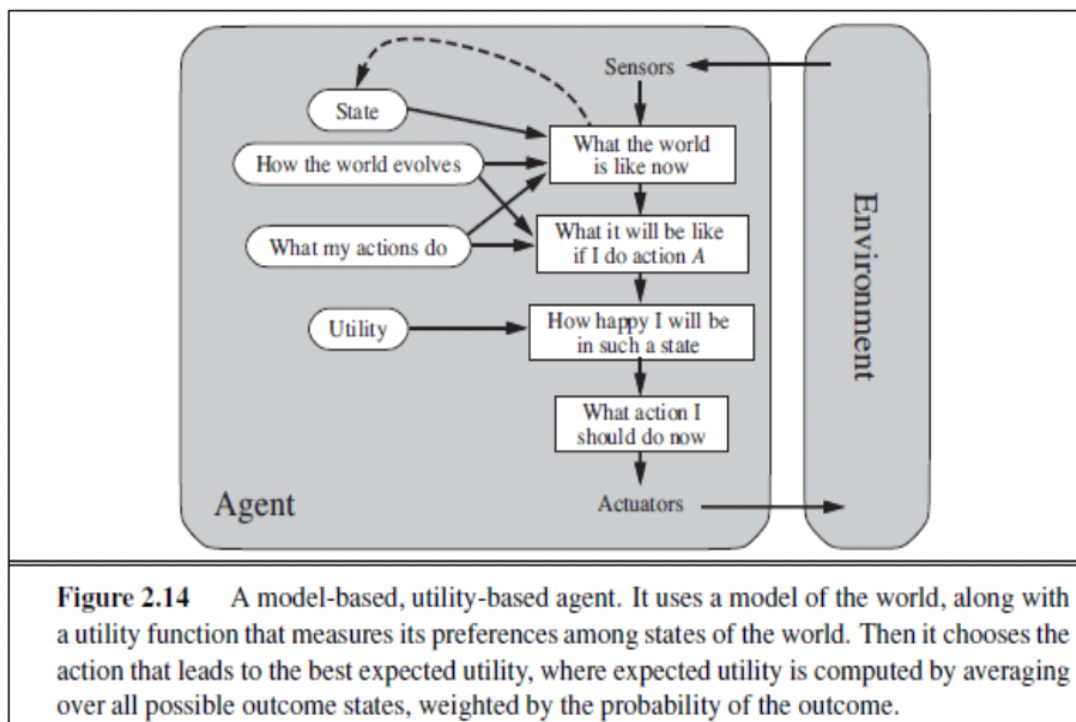


- ✓ **Search** and **planning** are the subfields of AI devoted to finding action sequences that achieve the agent's goals.
- ✓ Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.
- ✓ If it starts to rain, the agent can update its knowledge of how effectively its brakes will operate; this will automatically cause all of the relevant behaviors to be altered to suit the new conditions.

V Utility based Agents:

- ✓ Goals alone are not really enough to generate high-quality behavior in most environments. For example, there are many action sequences that 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, whereas a more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved.
- ✓ Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.
- ✓ A **utility function** maps a state (or a sequence of states) onto a real number, which describes the associated degree of happiness.
- ✓ A complete specification of the utility function allows rational decisions in two kinds of cases where goals are

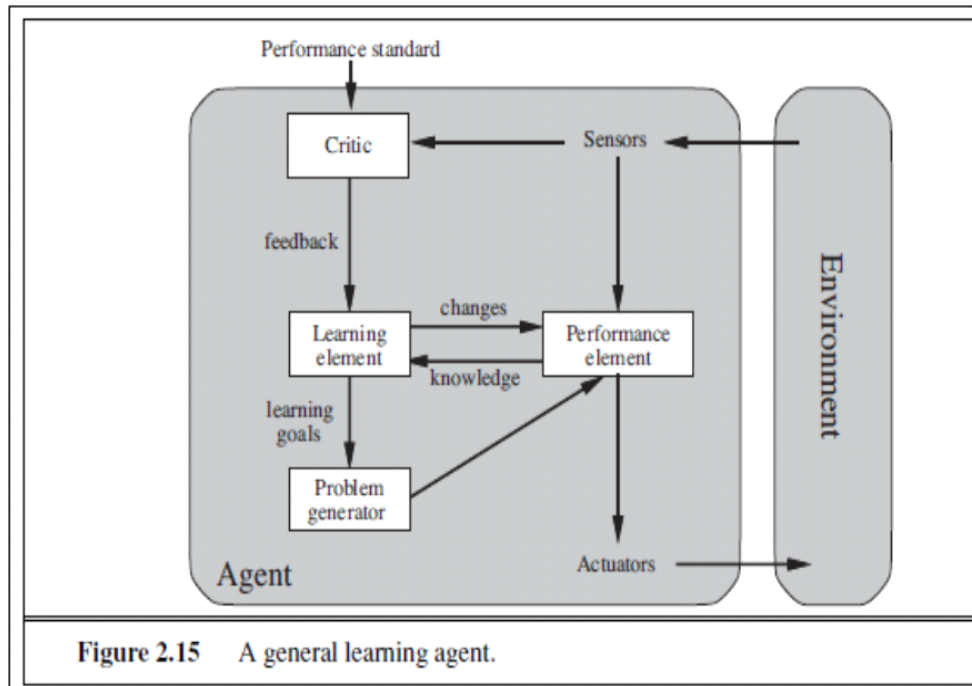
inadequate.



- ✓ First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff.
- ✓ Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed up against the importance of the goals.
- ✓ Partial observability and stochasticity are ubiquitous in the real world, and so, therefore, is decision making under uncertainty.
- ✓ Technically speaking, a rational utility-based agent chooses the action that maximizes the **expected utility** of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome.

Vi Learning Agents:

- ✓ All agents can improve their performance through learning.
- ✓ A learning agent can be divided into four conceptual components as shown in figure:
 - a. Learning element
 - b. Performance element
 - c. Critic
 - d. Problem generator



a. Learning element - is responsible for making improvements.

b. Performance element - is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

c. Critic - The learning element uses feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future.

d. Problem generator - It is responsible for suggesting actions that will lead to new and informative experiences.

ARTIFICIAL INTELLIGENCE

UNIT – II

Syllabus

Solving Problems by searching: Problem Solving Agents, Example problems, Searching for Solutions, Uninformed Search Strategies, Informed search strategies, Heuristic Functions.

Beyond Classical Search: Local Search Algorithms and Optimization Problems, Local Search in Continuous Spaces, Searching with Nondeterministic Actions, Searching with partial observations, online search agents and unknown environments.

Chapter – 1

Solving Problems by searching

2.1.1. Problem Solving Agents :

Problem formulation is the process of deciding what actions and states to consider, given a goal .

- ✓ An important aspect of intelligence is goal-based problem solving.
- ✓ The solution of many problems can be described by finding a sequence of actions that lead to a desirable goal.
- ✓ Each action changes the state and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

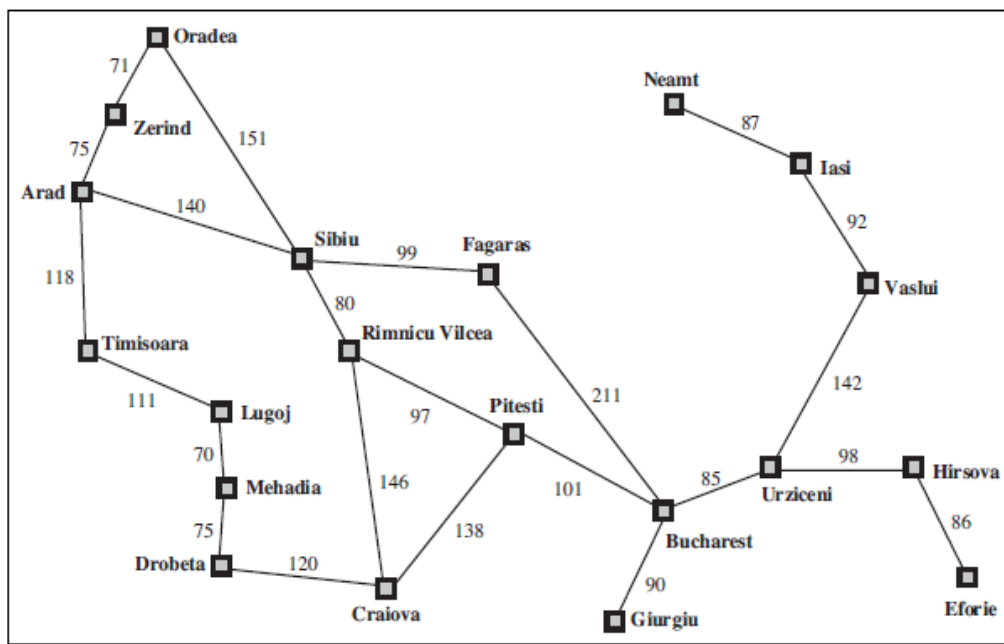
➤ What is Search?

- ✓ Search is the systematic examination of states to find path from the start/root state to the goal state.
- ✓ The set of possible states, together with operators defining their connectivity constitute the search space.
- ✓ The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

(i) Problem Solving Agents:

- ✓ A Problem solving agent is a goal-based agent .It decide what to do by finding sequence of actions that lead to desirable states.
- ✓ The agent can adopt a goal and aim at satisfying it.

Example : From Arad(Romania) to Bucharest



- ✓ Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.
- ✓ The agent's task is to find out which sequence of actions will get to a goal state.
- ✓ Problem formulation is the process of deciding what actions and states to consider given a goal.
- ✓ An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value, and then choosing the best sequence.
- ✓ The process of looking for sequences actions from the current state to reach the goal state is called search.
- ✓ The search algorithm takes a problem as input and returns a solution in the form of action sequence.

- ✓ Once a solution is found, the execution phase consists of carrying out the recommended action..
- ✓ Simple “formulate, search, execute” design for the agent...

```

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

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

```

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

➤ **The agent design assumes the Environment is:**

- **Static** : The entire process carried out without paying attention to changes that might be occurring in the environment.
- **Observable** : The initial state is known and the agent’s sensor detects all aspects that are relevant to the choice of action.
- **Discrete** : With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken.
- **Deterministic** : The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions.

(ii) Well-defined problems and solutions:

A well-defined problem can be described by:

- Initial state
- Operator or successor function - for any state x returns $s(x)$, the set of states reachable from x with one action
- State space - all states reachable from initial by any sequence of actions
- Path - sequence through state space
- Path cost - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- Goal test - test to determine if at goal state
- The step cost of taking action ‘ a ’ to go from state x to state y is denoted by $c(x,a,y)$. The step cost for Romania are shown in next figure in route distances. It is assumed that the step costs are non negative.
- A solution to the problem is a path from the initial state to a goal state.
- An optimal solution has the lowest path cost among all solutions.

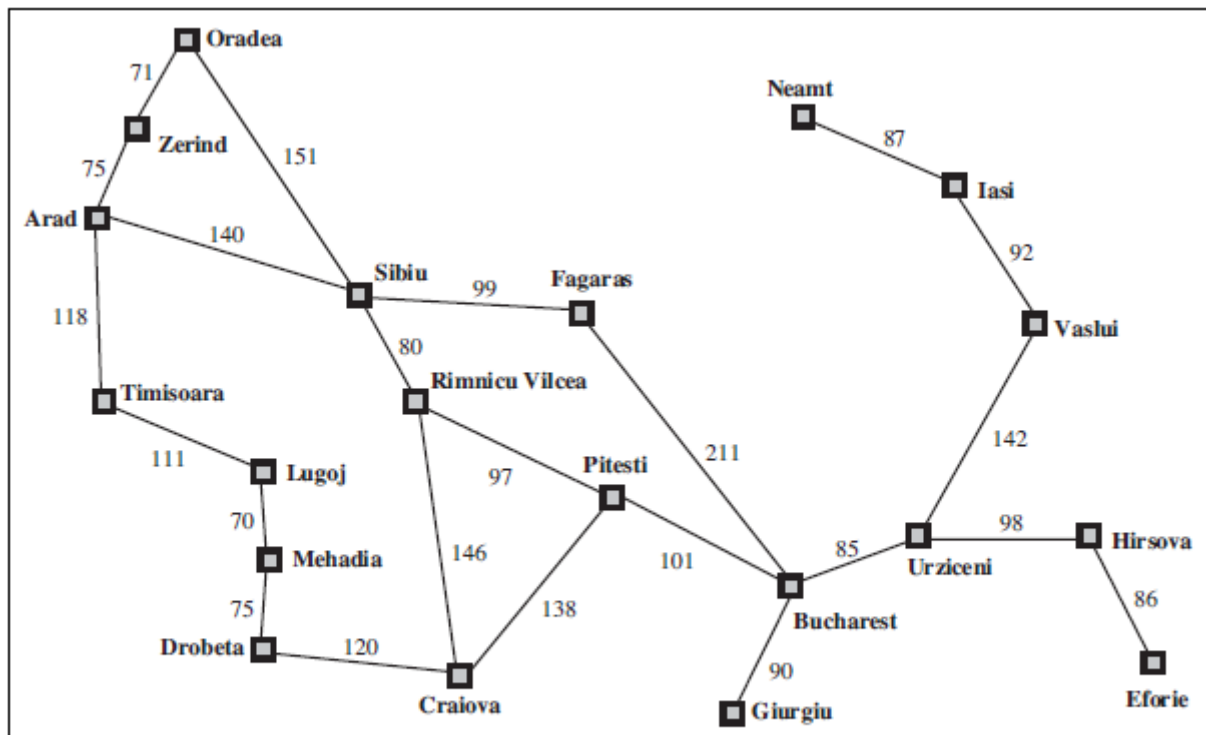


Figure 3.2 A simplified road map of part of Romania.

2.1.2. Example Problems:

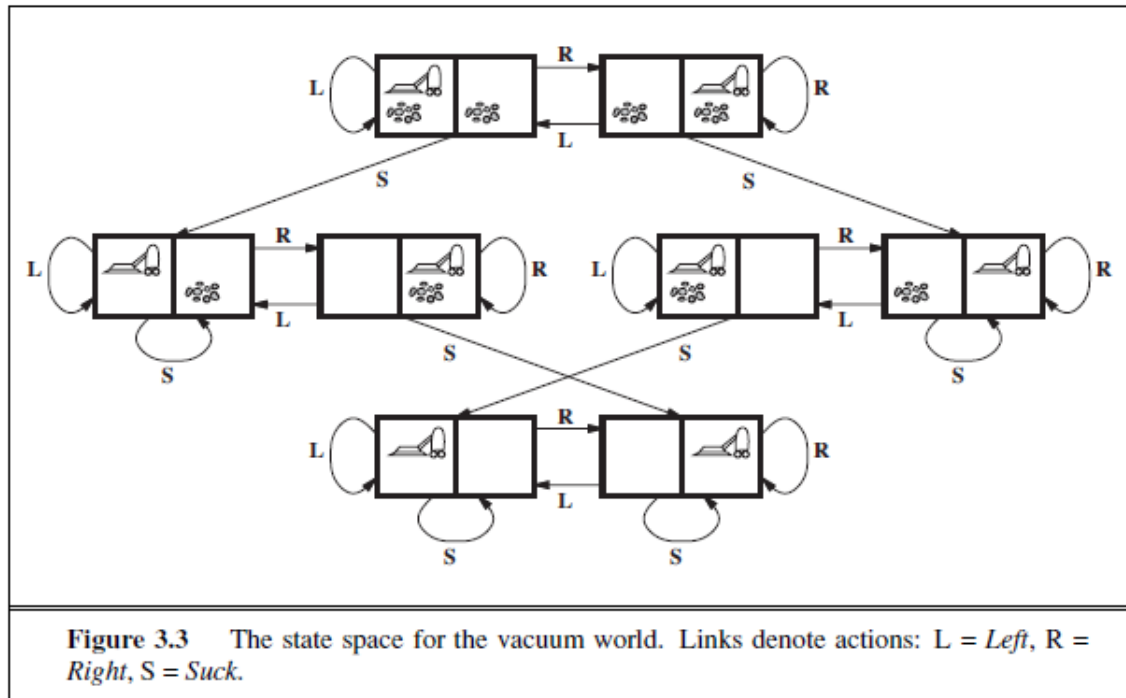
➤ Toy vs Real World problems...

- ✓ A toy problem is intended to illustrate or exercise various problem-solving methods.
- ✓ It can be given a concise, exact description and hence is usable by different researchers to compare the REALWORLD performance of algorithms.
- ✓ A real-world problem is one whose solutions people actually PROBLEM care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

1. Toy Problems:

➤ Example 1: Vacuum world

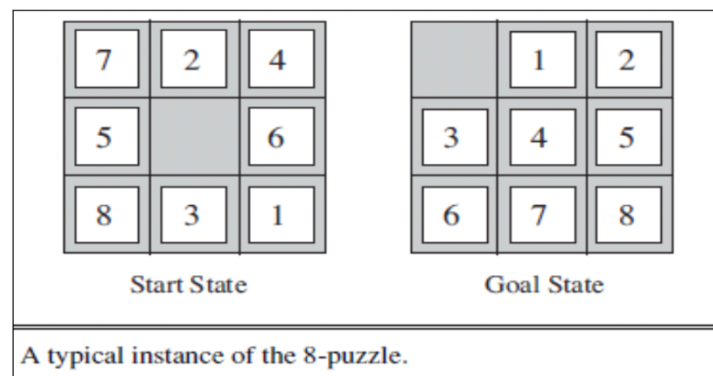
- i. **States:** The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are $2 \times 2^2 = 8$ possible world states.
- ii. **Initial state:** Any state can be designated as initial state.
- iii. **Successor function :** This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure.
- iv. **Goal Test :** This tests whether all the squares are clean.
- v. **Path test :** Each step costs one ,so that the the path cost is the number of steps in the path.



- ✓ **States?:** integer dirt and robot locations (ignore dirt amounts etc.)
- ✓ **Actions?:** Left, Right, Suck, NoOp
- ✓ **Goal test?:** no dirt
- ✓ **Path cost?:** 1 per action (0 for NoOp)

Example 2: The 8-puzzle:

- ✓ An 8-puzzle consists of a 3x3 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 the goal state, as shown in Figure.



➤ The problem formulation is as follows :

- i. **States :** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- ii. **Initial state :** Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- iii. **Successor function :** This generates the legal states that result from trying the four actions (blank moves Left, Right, Up or down).

iv. **Goal Test** : This checks whether the state matches the goal configuration shown in figure(Other goal configurations are possible).

v. **Path cost** : Each step costs 1,so the path cost is the number of steps in the path.

- ✓ The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete.
- ✓ The 8-puzzle has $9!/2 = 181,440$ reachable states and is easily solved.
- ✓ The 15 puzzle (4 x 4 board) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.
- ✓ The 24-puzzle (on a 5 x 5 board) has around 1025 states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

i. **States?**: Integer locations of tiles (ignore intermediate positions)

ii. **Actions?**: Move blank left, right, up, down (ignore unjamming etc.)

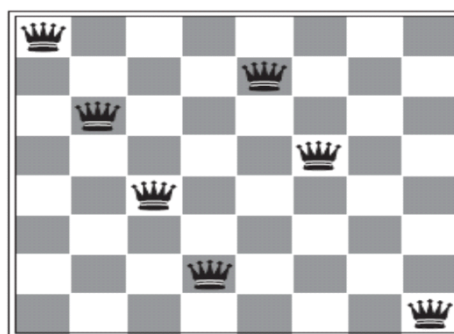
iii. **Goal test?**: Goal state (given).

iv. **Path cost?**: 1 per move.

[Note: optimal solution of n-Puzzle family is NP-hard]

➤ **Example 3: 8 Queen Problem:**

- ✓ The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other.(A queen attacks any piece in the same row, column or diagonal).
- ✓ An Incremental formulation involves operators that augments the state description, starting with an empty state for 8-queens problem, this means each action adds a queen to the state.
- ✓ A complete-state formulation starts with all 8 queens on the board and move them around.
- ✓ In either case the path cost is of no interest because only the final state counts.



➤ **The first incremental formulation one might try is the following :**

- i. **States** : Any arrangement of 0 to 8 queens on board is a state.
- ii. **Initial state** : No queen on the board.
- iii. **Successor function** : Add a queen to any empty square.
- iv. **Goal Test** : 8 queens are on the board, none attacked.
 - ✓ In this formulation, we have $64 \cdot 63 \dots 57 = 3 \times 10^{14}$ possible sequences to investigate.
 - ✓ A better formulation would prohibit placing a queen in any square that is already attacked.

- ✓ States : Arrangements of n queens ($0 \leq n \leq 8$), one per column in the left most columns ,with no queen attacking another are states.
- ✓ Successor function : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.
- ✓ This formulation reduces the 8-queen state space from 3×10^{14} to just 2057, and solutions are easy to find.

2. Real-world Problems:

- i. Route finding problem
- ii. Airline Travel problem
- iii. Touring problems
- iv. Travelling salesperson problem
- v. VLSI Layout
- vi. ROBOT Navigation
- vii. Automatic Assembly problem
- viii. Internet searching

(i) Route-Finding Problem :

- **Route-Finding Problem** is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications.
- Some, such as Web sites and in-car systems that provide driving directions.
- Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications.
- Consider the airline travel problems that must be solved by a travel-planning Web site:

• **States:** Each state obviously includes a location (e.g., an airport) 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.

(ii) Touring problems :

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem “Visit every city at least once, starting and ending in Bucharest”.

(iii) Traveling Salesperson Problem (TSP) :

- The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once.
- The aim is to find the *shortest* tour.

The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.

(iv) VLSI layout problem :

- A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
- The layout problem comes after the logical design phase and is usually split into two parts:

→ **Cell layout**

→ **Channel routing.**

→ **Cell layout :**

- In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function.

→ **Channel routing :**

- Channel routing finds a specific route for each wire through the gaps between the cells.

(v) Robot Navigation :

- **Robot navigation** is a generalization of the route-finding problem described earlier.
- Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.
- For a circular robot moving on a flat surface, the space is essentially two-dimensional.

(vi) Automatic assembly sequencing :

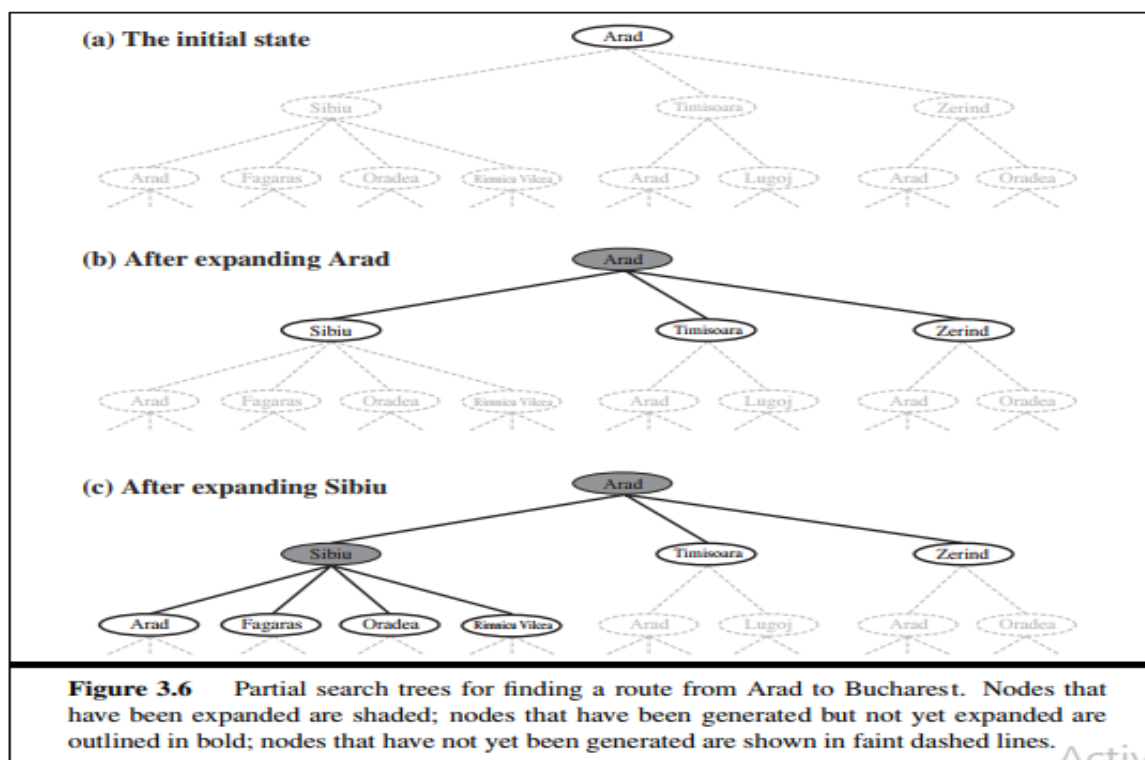
- **Automatic assembly sequencing** of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972).
- Progress since then has been slow but sure.
- In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the

wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.

- Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation.
- Another important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

2.1.3. SEARCHING FOR SOLUTIONS :

- A solution is an action sequence, so search algorithms work by considering various possible action sequences.
- The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem.
- The diagram shows the first few steps in growing the search tree for finding a route from Arad to Bucharest.
- By **expanding** the current state , and thereby **generating** a new set of states.



- Each of these six nodes is a **leaf node**, that is, a node with no children in the tree.
- The set of all leaf nodes available for expansion at any given point is called the **frontier**(Many authors call it the **open list**)
the frontier of each tree consists of those nodes with bold outlines.
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

- Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.

```

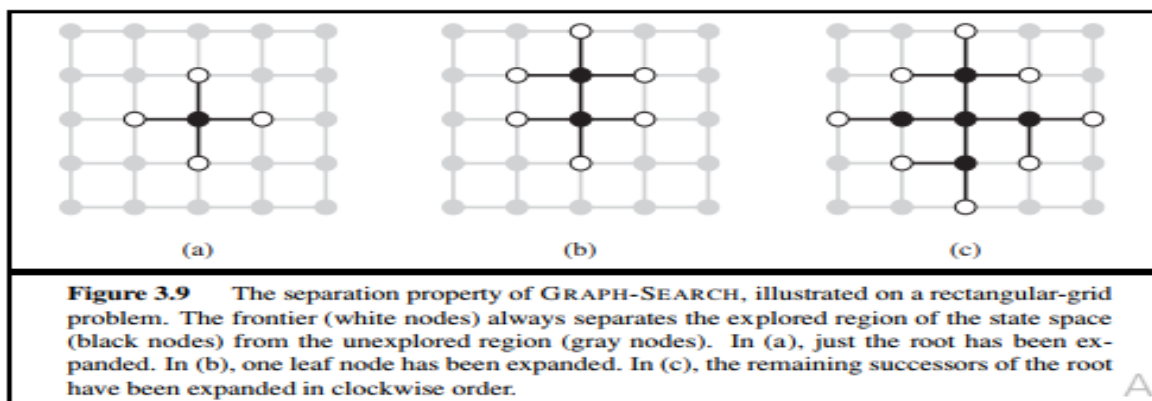
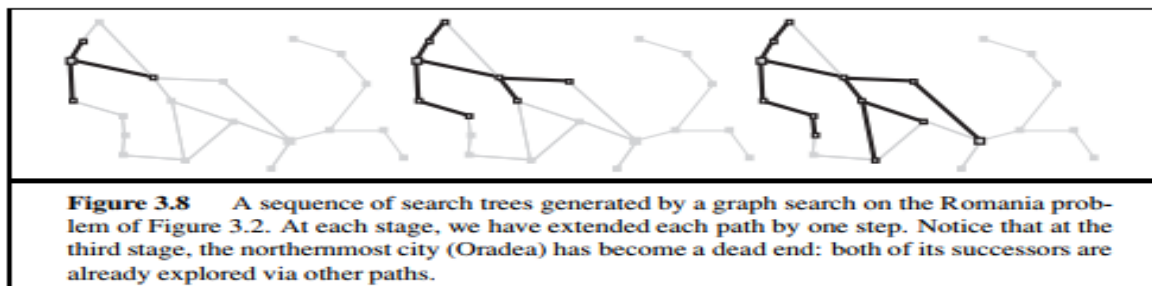
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
  
```

```

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
  
```

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

- Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one way to get from one state to another.
- As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is *systematically* examining the states in the state space, one by one, until it finds a solution.



(i) Infrastructure for search algorithms :

- Search algorithms require a data structure to keep track of the search tree that is being constructed.

- For each node n of the tree, we have a structure that contains four components:
 - **n.STATE**: the state in the state space to which the node corresponds;
 - **n.PARENT**: the node in the search tree that generated this node;
 - **n.ACTION**: the action that was applied to the parent to generate the node;
 - **n.PATH-COST**: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

➤ Queue :

- Now that we have nodes, we need somewhere to put them.
- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a **queue**.
- The operations on a queue are as follows:
 - **EMPTY?(queue)** returns true only if there are no more elements in the queue.
 - **POP(queue)** removes the first element of the queue and returns it.
 - **INSERT(element, queue)** inserts an element and returns the resulting queue.
- Queues are characterized by the *order* in which they store the inserted nodes.
- Three common variants are
 - The first-in, first-out or **FIFO queue**, which pops the *oldest* element of the queue;
 - LIFO QUEUE the last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element
 - PRIORITY QUEUE of the queue; and the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

(ii) Measuring problem-solving performance:

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

2.1.4. Uninformed Search strategies :

- ✓ Uninformed Search Strategies have no additional information about states beyond that provided in the problem definition.

➤ **There are six uninformed search strategies as given below:**

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bi-directional search

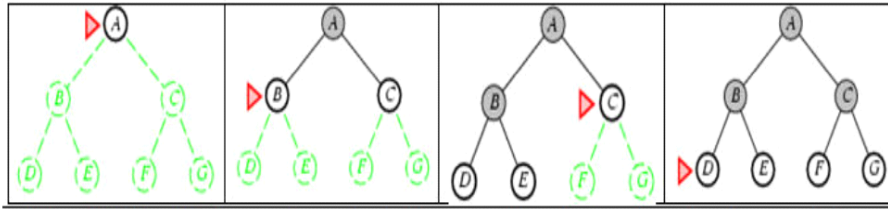
1. Breadth-first search(BFS):

- ✓ Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on.
- ✓ In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- ✓ Breadth-First-Search is implemented by calling TREE SEARCH with an empty fringe that is a first-in-first out (FIFO) queue, assuring that the nodes that are visited first will be expanded first.
- ✓ In other words, calling TREE-SEARCH (problem, FIFO-QUEUE()) results in breadth-first-search.
- ✓ The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.

➤ **BFS Algorithm...**

```
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)
```

Example for BFS Algorithm...



➤ Properties of BFS...

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

➤ Time and memory requirements....

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-firstsearch. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

2. Uninform-cost Search(UCS)...

- ✓ Instead of expanding the shallowest node, uniform-cost search expands the node 'n' with the lowest path cost.
- ✓ Uniform-cost search does not care about the number of steps a path has, but only about their total cost.
- ✓ Expand least-cost unexpanded node.
- ✓ We think of this as having an evaluation function: $g(n)$, which returns the path cost to a node n .
- ✓ fringe = queue ordered by evaluation function, lowest first.
- ✓ Equivalent to breadth-first if step costs all equal
- ✓ Complete and optimal.
- ✓ Time and space complexity are as bad as for breadth-first search.

UCS Algorithm:

```

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child

```

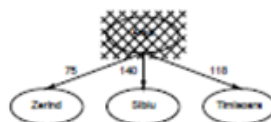
Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

➤ Process...

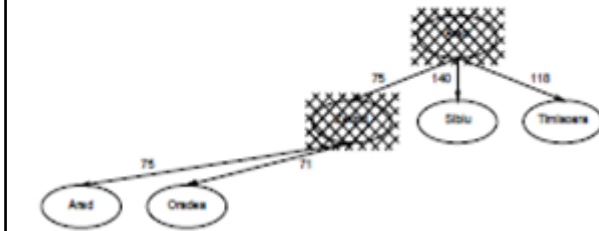
- Add the node representing the initial state into the fringe.

And

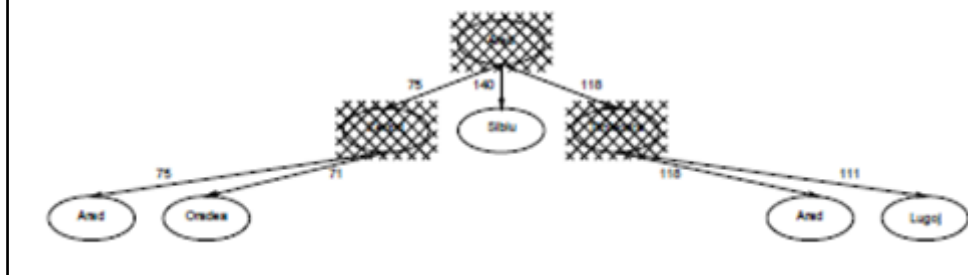
Remove the first node in the fringe and add its children
The queue is ordered with the cheapest first.



Remove the first node in the fringe and add its children — they are added in priority order.



Repeat.



➤ Properties of UCS....

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq \text{cost of optimal solution}$, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq \text{cost of optimal solution}$, $O(b^{\lceil C^*/\epsilon \rceil})$

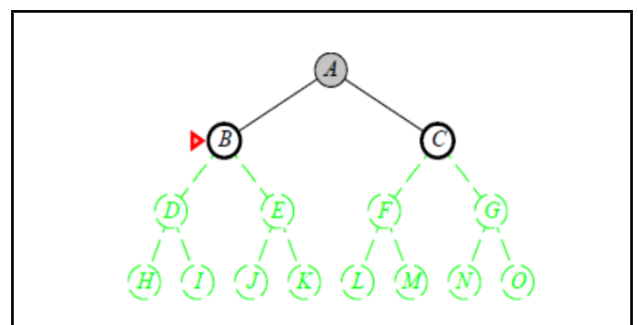
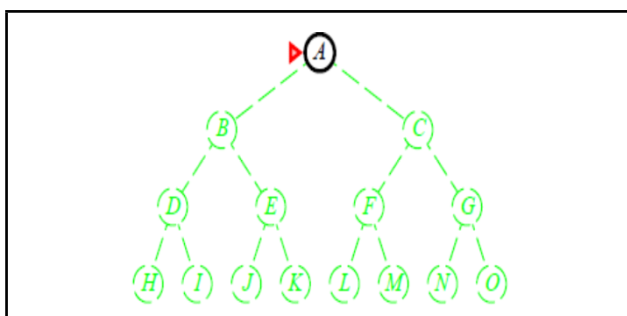
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

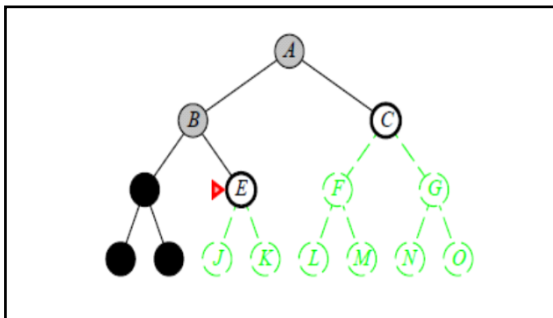
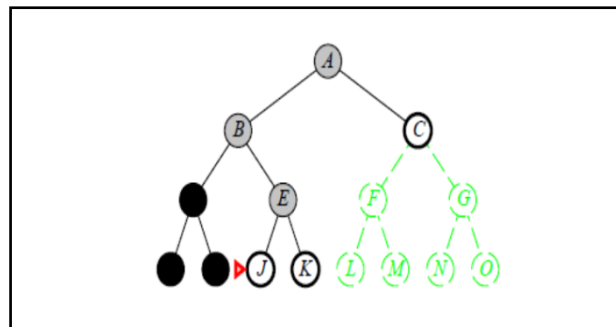
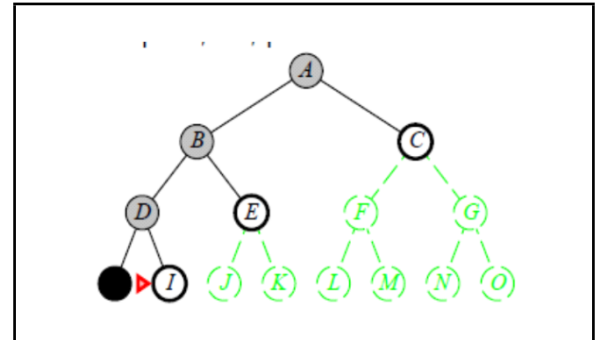
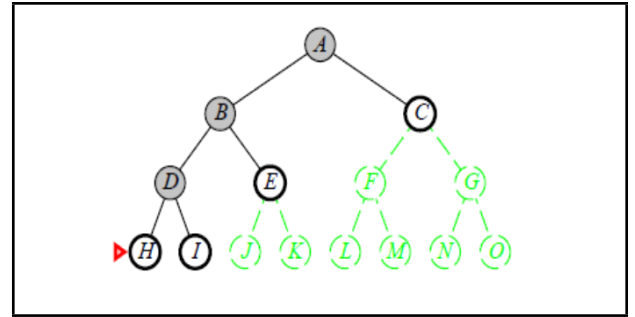
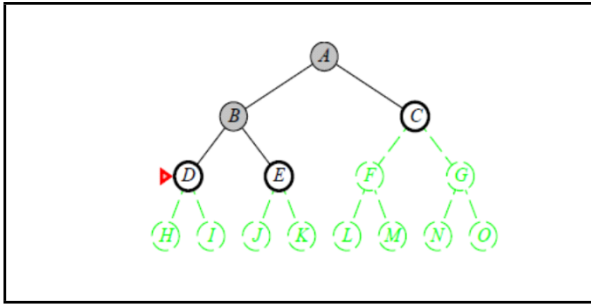
3. Depth First Search(DFS)...

- ✓ Depth-first-search always expands the deepest node in the current fringe of the search tree.
- ✓ The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- ✓ As those nodes are expanded, they are dropped from the fringe ,so then the search “backs up” to the next shallowest node that still has unexplored successors.

➤ Implementation:

fringe = LIFO queue, i.e., put successors at front.





➤ Properties of Depth First Search(DFS):

- ✓ **Complete?** - No: Fails in infinite-depth spaces, spaces with loops. Modify to avoid repeated states along Path complete infinite spaces
- ✓ **Time??** : $O(b^m)$: Terrible if m is much larger than 'd', but if solutions are dense, may be much faster than

breadth first search.

- ✓ **Space??** : $O(b^m)$, i.e., linear space!
- ✓ **Optimal??** : No

4. Depth-Limited Search...

- ✓ The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit 'l'.
- ✓ That is, nodes at depth 'l' are DEPTH-LIMITED treated as if they have no successors. This approach is called depth limited search.
- ✓ The SEARCH depth limit solves the infinite-path problem.
- ✓ Unfortunately, it also introduces an additional source of incompleteness if we choose 'l' < d, that is, the

shallowest goal is beyond the depth limit. (This is likely when d is unknown.)

- ✓ Depth-limited search will also be non-optimal if we choose ' l ' $> d$.
- ✓ Its time complexity is $O(b)$ and its space complexity is $O(b)$. Depth-first search can be viewed as a special case of depth-limited search with ' l ' $= \infty$.

```
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?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
        child  $\leftarrow$  CHILD-NODE(problem, node, action)
        result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
        if result = cutoff then cutoff_occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

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

5. Iterative deepening depth-first search...

- ✓ Iterative deepening search (or iterativedeepening-depth-first-search) is a general strategy often used in combination with depth-first-search, that finds the better depth limit.
- ✓ It does this by gradually increasing the limit – first 0, then 1, then 2, and so on – until a goal is found.
- ✓ This will occur when the depth limit reaches d , the depth of the shallowest goal node.
- ✓ Iterative deepening combines the benefits of depth-first and breadth-first-search.
- ✓ Like depth-first-search, its memory requirements are modest; $O(bd)$ to be precise.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  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.

- ✓ In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

➤ Four iterations of iterative deepening search on a binary tree is shown below.

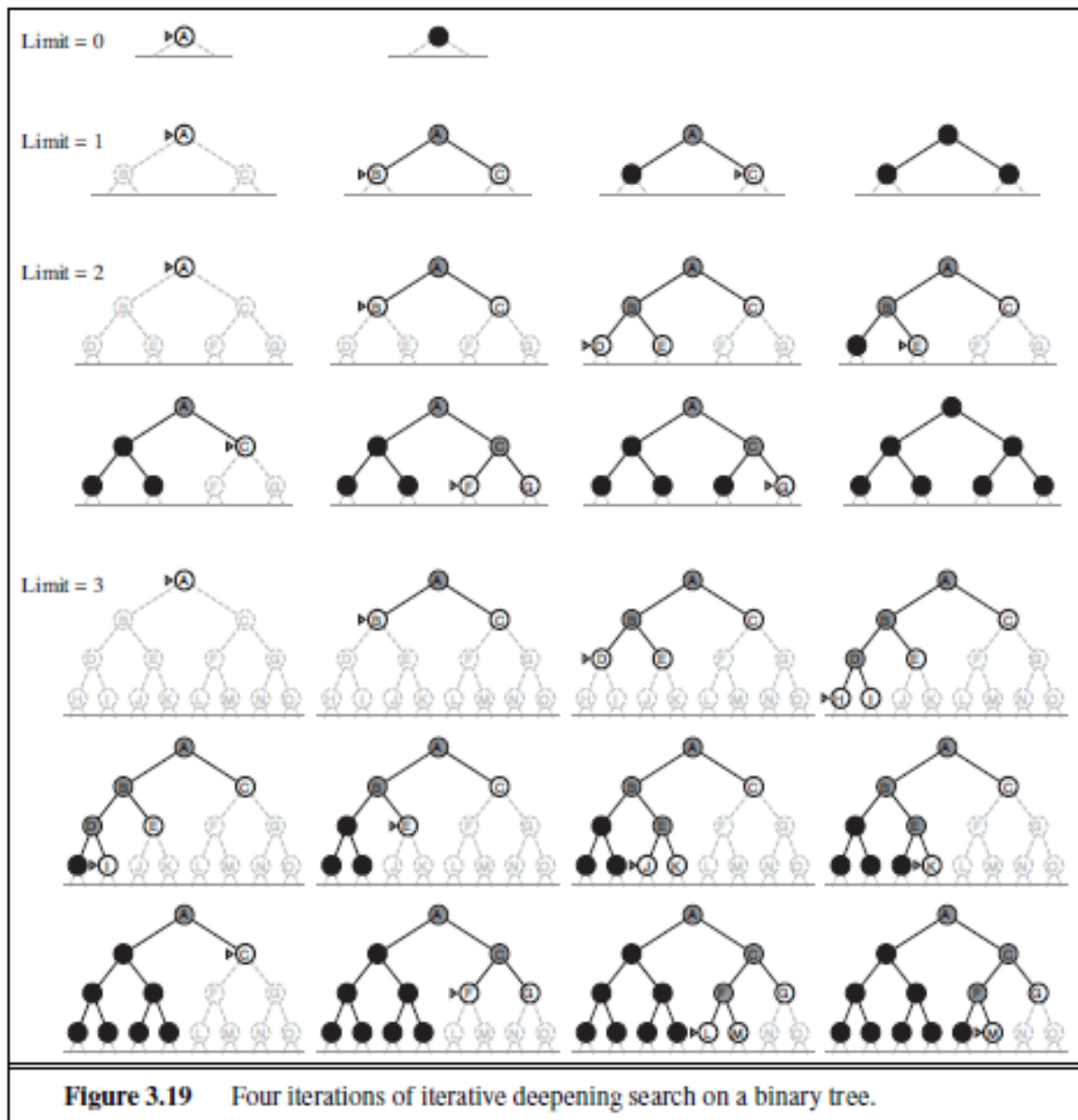


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

➤ Properties of IDS....

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant

Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

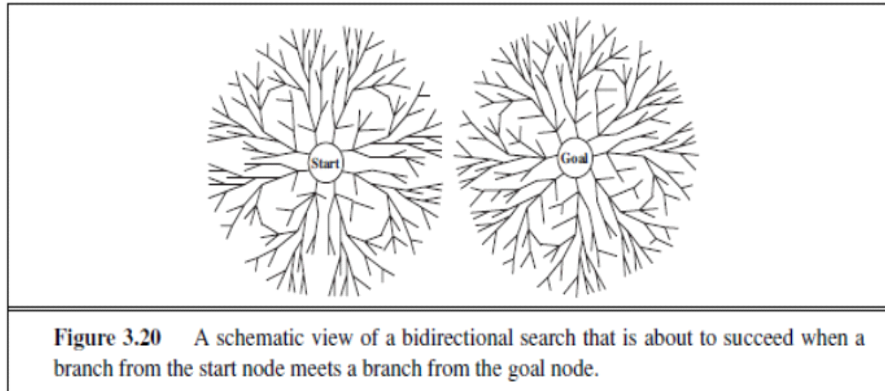
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

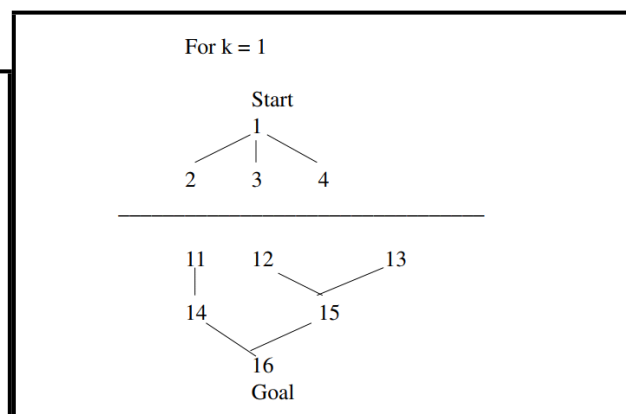
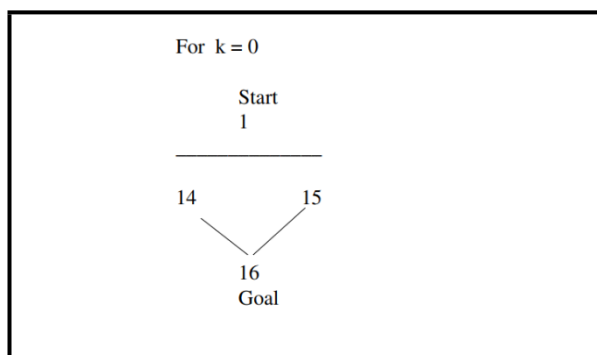
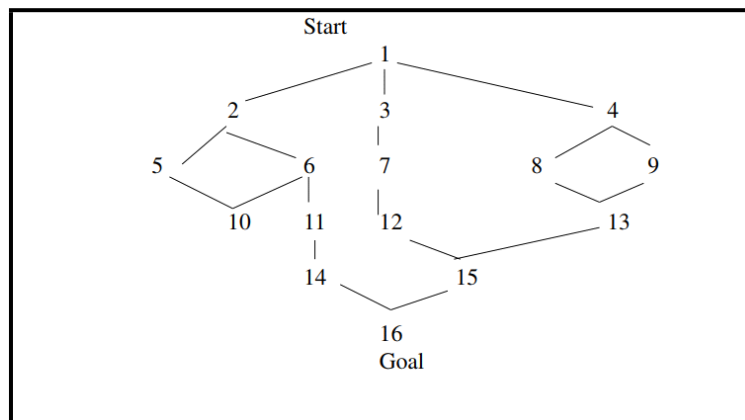
BFS can be modified to apply goal test when a node is generated

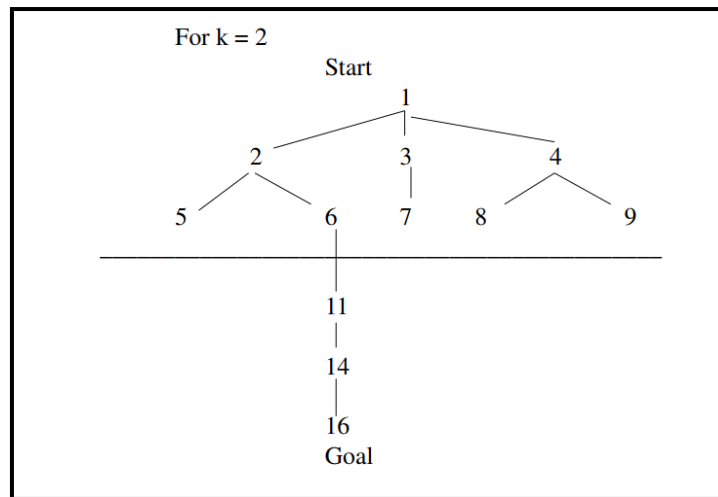
6. Bi-directional search...

- ✓ The idea behind bidirectional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.
- ✓ The motivation is that $b^{d/2} + b^{d/2}$ much less than or the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.



➤ Example Graph:





2.4.7 Comparing uninformed search strategies...

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
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.

2.1.5. Informed or Heuristic search strategies...

- ✓ Informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself.
- ✓ It can find solutions more efficiently than uninformed strategy.
- ✓ Best-first search is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function $f(n)$.
- ✓ The node with lowest evaluation is selected for expansion, because the evaluation measures the distance to the goal.
- ✓ This can be implemented using a priority-queue, a data structure that will maintain the fringe in ascending order of f values.
- ✓ A **heuristic function** or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.
- ✓ The key component of Best-first search algorithm is a heuristic function, denoted by $h(n)$:
 $h(n)$ = estimated cost of the cheapest path from node n to a goal node.
- ✓ Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

➤ Heuristic Searching methods...

- i. Greedy Best First search
- ii. A* search

1. Greedy Best First search...

- ✓ Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to a solution quickly.
- ✓ It evaluates the nodes by using the heuristic function $f(n) = h(n)$.
- ✓ Taking the example of Route-finding problems in Romania , the goal is to reach Bucharest starting from the city Arad.
- ✓ We need to know the straight-line distances(SLD) to Bucharest from various cities as shown in Figure.
- ✓ For example, the initial state is In(Arad) ,and the straight line distance heuristic $h_{SLD}(In(Arad))$ is found to be 366.

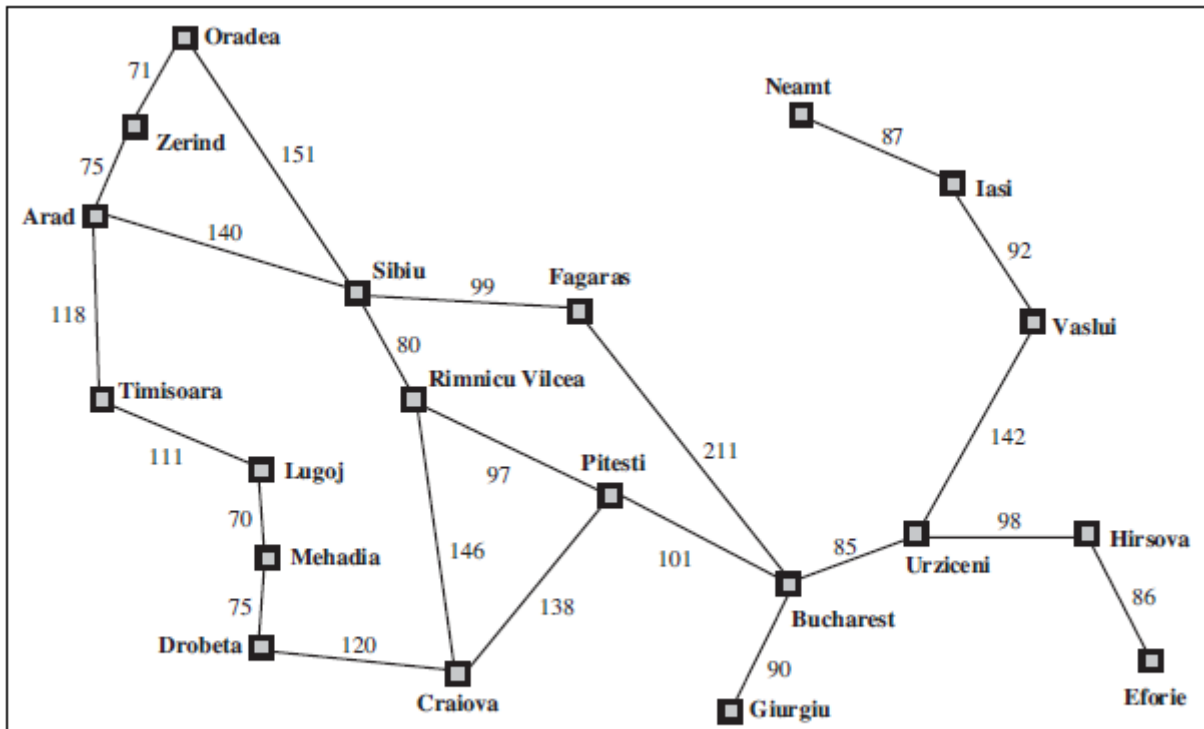


Figure 3.2 A simplified road map of part of Romania.

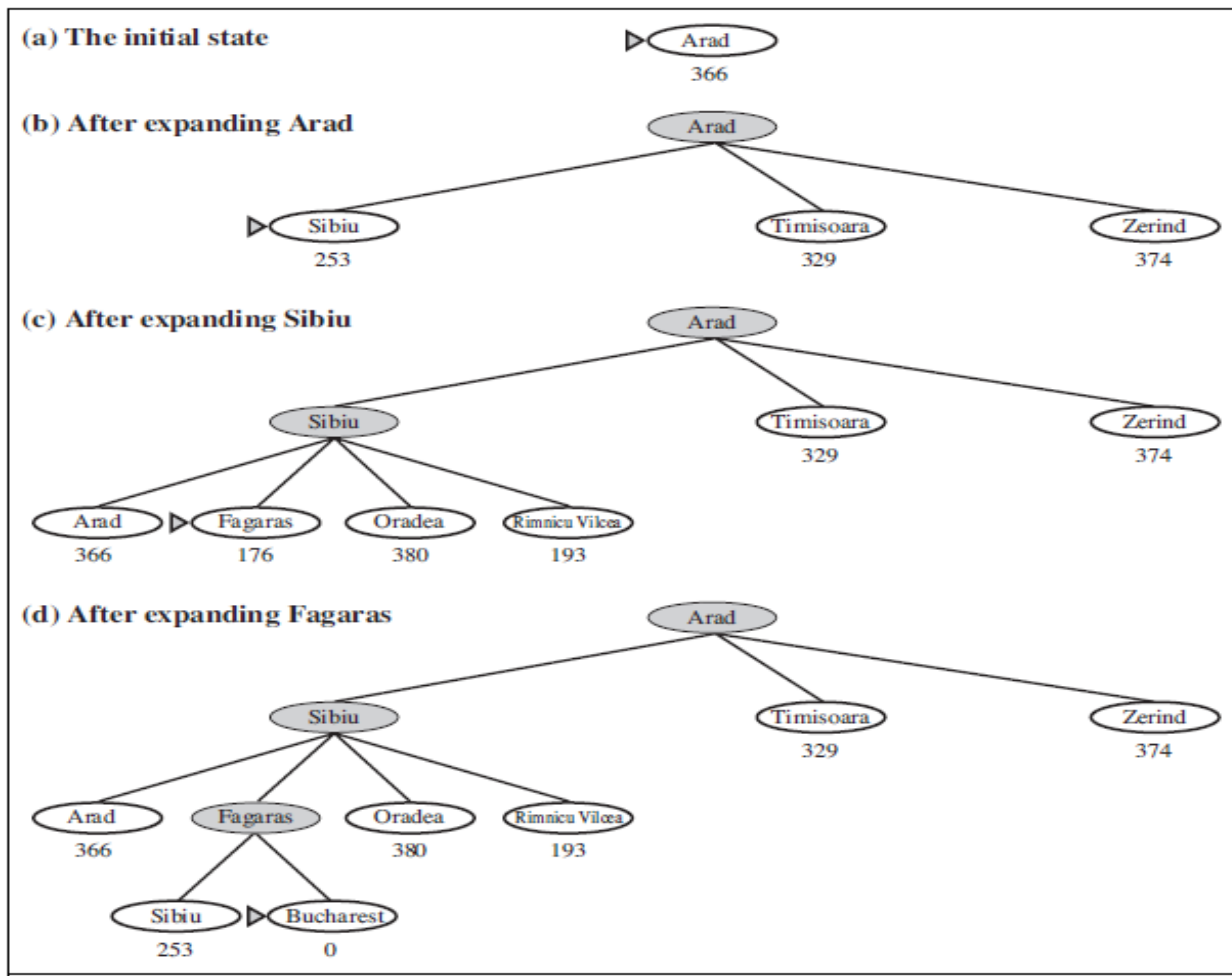
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

- ✓ The figure shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest.
- ✓ The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara.

- ✓ The next node to be expanded will be Fagaras, because it is closest.
- ✓ Fagaras in turn generates Bucharest, which is the goal.

FINAL PATH : Arad → Sibiu → Fagaras → Bucharest.



➤ Greedy search Algorithm...

```

function GREEDY-SEARCH(problem, fringe) returns a solution, or
failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
      fringe ← SORTBYHVALUE(fringe)
  end
  
```

2. A* Search...

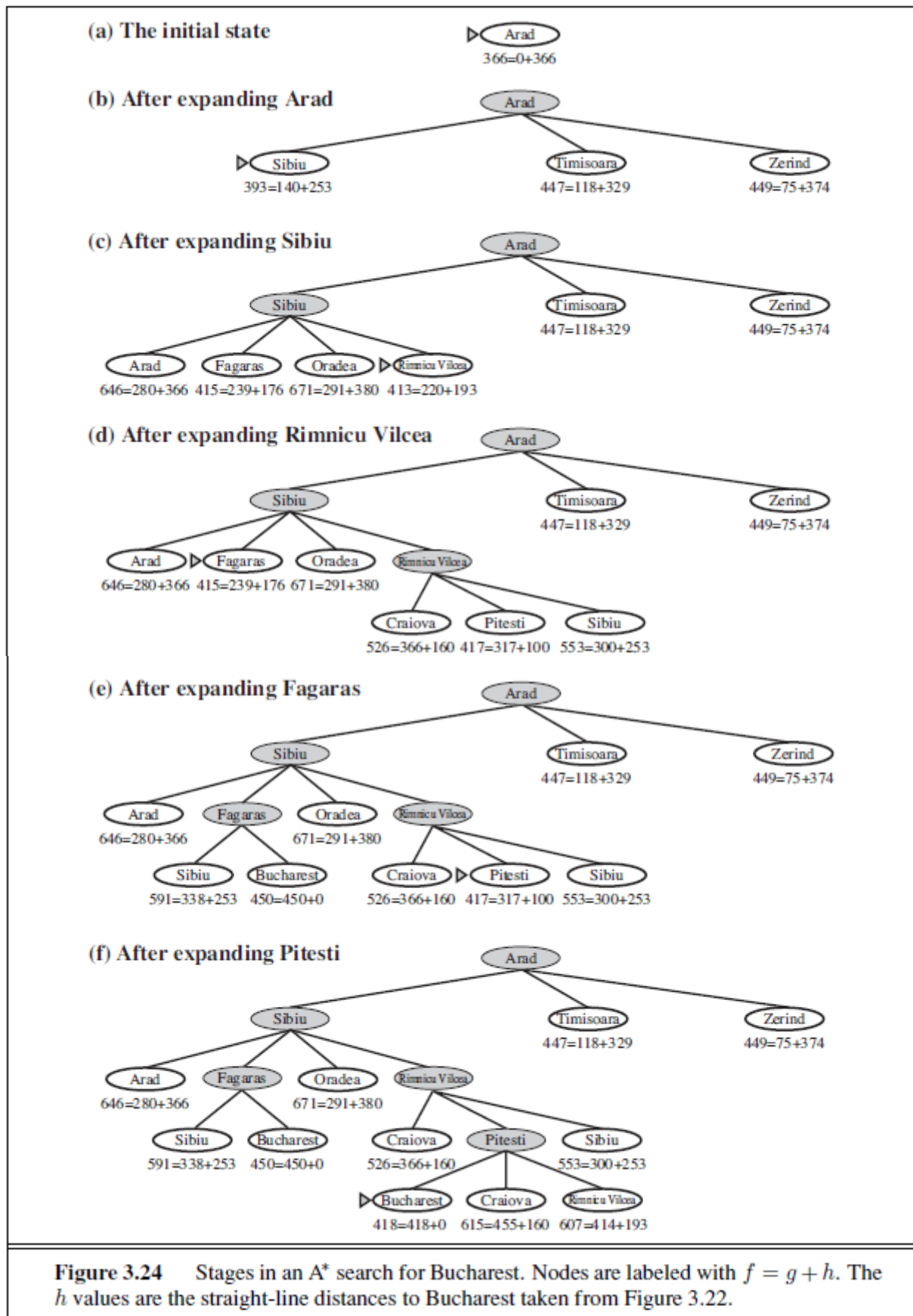
- ✓ A* is very efficient search strategy and it is the most widely used form of best-first search.
- ✓ Basic idea is to combine uniform cost search and greedy search.
- ✓ The evaluation function $f(n)$ is obtained by combining $g(n)$ = the cost to reach the node, and $h(n)$ = the cost to get from the node to the goal :
$$f(n) = g(n) + h(n).$$

(or)

□ $f(n) = g(n) + h(n)$ where

 - $g(n)$ is path cost of n ;
 - $h(n)$ is expected cost of cheapest solution from n .
- ✓ It Aims to minimise overall cost.
- ✓ **Algorithm for A* search strategy:**

```
function A-STAR-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
      fringe ← SORTBYFVALUE(fringe)
  end
```



FINAL PATH: Arad → Sibiu → Rimnicu → Pitesti → Bucharest.

➤ Optimality of A*...

- ✓ A* search is both complete and optimal.
- ✓ The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses $g + h$ instead of g .
- ✓ A* is optimal in precise sense—it is guaranteed to find a minimum cost path to the goal.

There are a set of conditions under which A* will find such a path:

1. Each node in the graph has a finite number of children.
2. All arcs have a cost greater than some positive .
3. For all nodes in the graph $h(n)$ always underestimates the true distance to the goal.

➤ Conditions for the Optimality...

1. The first condition we require for optimality is that $h(n)$ be an admissible heuristic.

An admissible heuristic is one that never overestimates the cost to reach the goal.

2. A second, slightly stronger condition called consistency (or sometimes monotonicity) is required only for applications of A* to graph search.

- A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' .

➤ Problems in A*...

- ✓ Computation time, A*'s main drawback.
- ✓ Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A* usually runs out of space long before it runs out of time.
- ✓ For this reason, A* is not practical for many largescale problems.
- ✓ There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

3. Memory-bounded Heuristic search...

- ✓ Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the Operation of standard best-first search, but using only linear space.
- ✓ Its structure is similar to that of a recursive depth first search, but rather than continuing indefinitely down the current path, it uses the f-limit variable to keep track of the f-value of the best alternative path.
- ✓ If the current node exceeds this limit, the recursion unwinds back to the alternative path.
- ✓ As the recursion unwinds, 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.

➤ RBFS Algorithm...

```

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE),  $\infty$ )

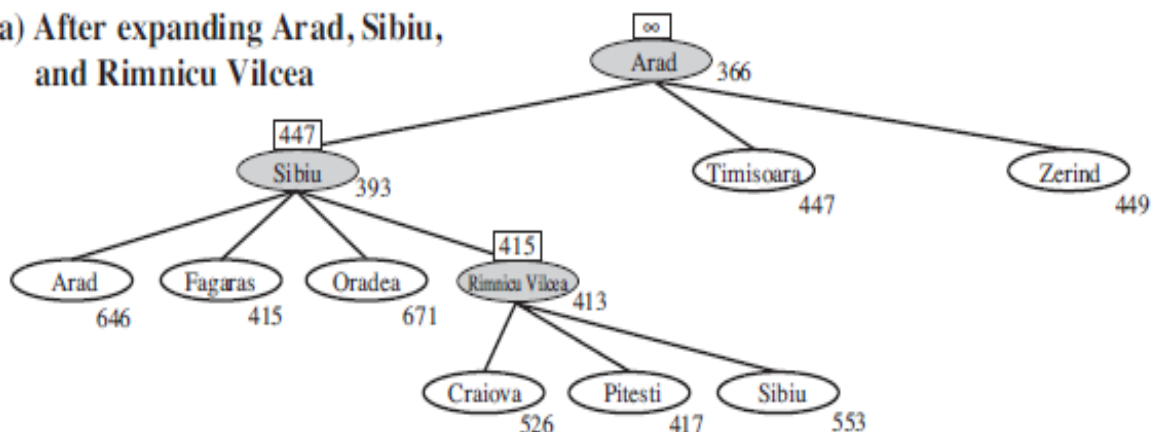
function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors  $\leftarrow []$ 
  for each action in problem.ACTIONS(node.STATE) do
    add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure,  $\infty$ 
  for each s in successors do /* update f with value from previous search, if any */
     $s.f \leftarrow \max(s.g + s.h, \text{node}.f)$ 
  loop do
    best  $\leftarrow$  the lowest f-value node in successors
    if best.f > f_limit then return failure, best.f
    alternative  $\leftarrow$  the second-lowest f-value among successors
    result, best.f  $\leftarrow$  RBFS(problem, best,  $\min(f\_limit, \text{alternative})$ )
    if result  $\neq$  failure then return result
  
```

Figure 3.26 The algorithm for recursive best first search.

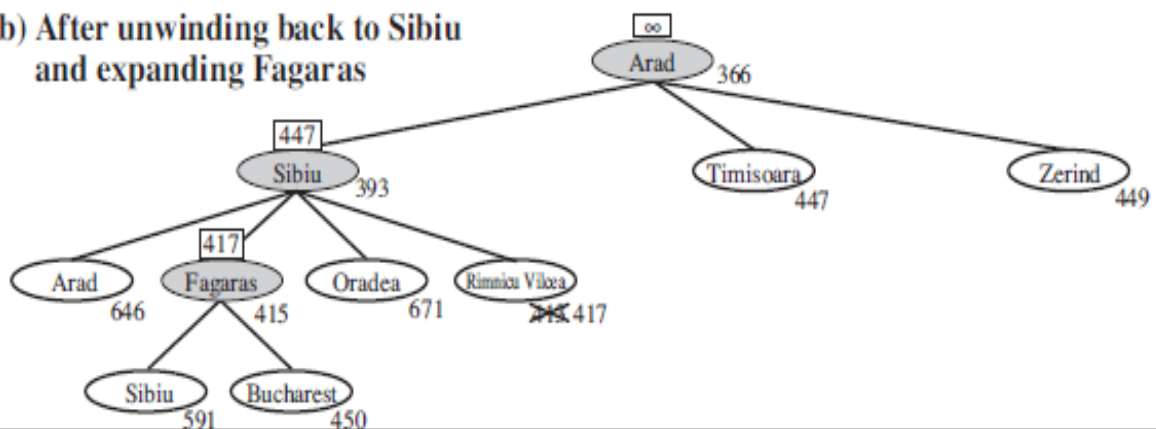
178

➤ Stages in RBFS Algorithm...

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti

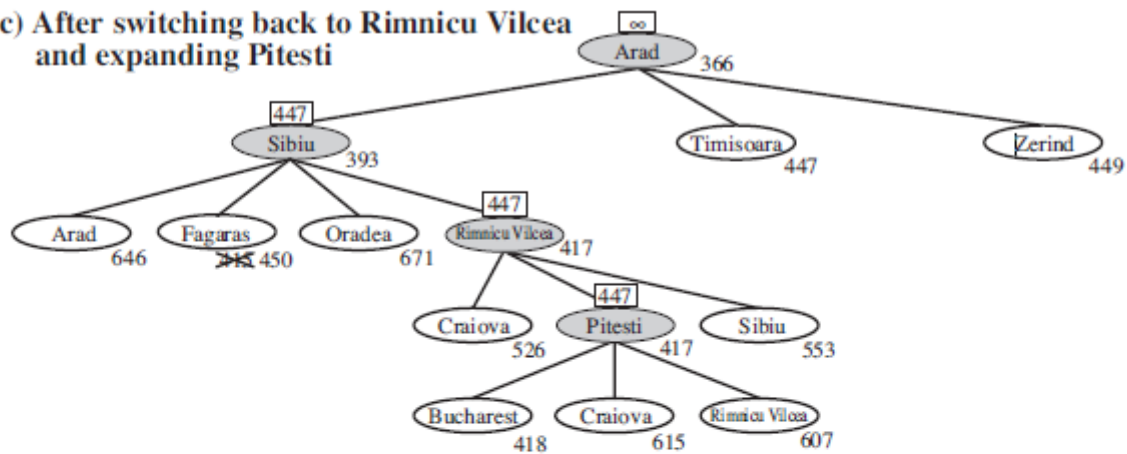


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f -limit value for each recursive call is shown on top of each current node, and every node is labeled with its f -cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is 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.

➤ RBFS Evaluation :

- ✓ RBFS is a bit more efficient than IDA* Still excessive node generation (mind changes).
- ✓ Like A*, optimal if $h(n)$ is admissible. □ Space complexity is $O(bd)$.
- ✓ IDA* retains only one single number (the current f -cost limit).
- ✓ Time complexity difficult to characterize Depends on accuracy if $h(n)$ and how often best path changes.
- ✓ IDA* and RBFS suffer from too little memory.

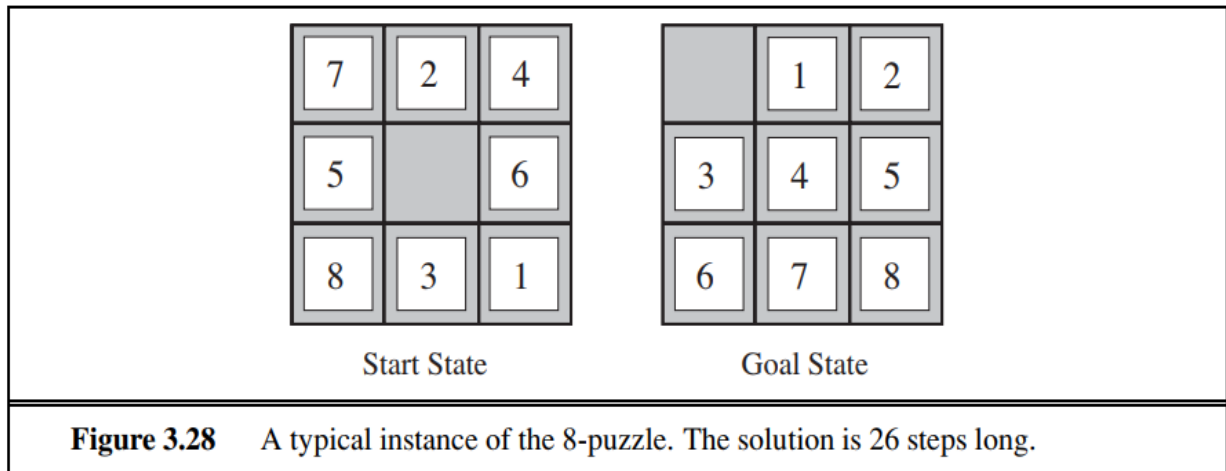
2.1.6. HEURISTIC FUNCTIONS :

- The 8-puzzle was one of the earliest heuristic search problems.
- The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration .
- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3.
- When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.

This means that an exhaustive tree search to depth 22 would look at about

$$3^{22} \approx 3.1 \times 10^{10} \text{ states.}$$

- A graph search would cut this down by a factor of about 170,000 because only $9!/2 = 181,440$ distinct states are reachable.



- The corresponding number for the 15-puzzle is roughly , so the next order of business is to find a good heuristic function.
- If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal.
- There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:
- **h1** = the number of misplaced tiles. All of the eight tiles are out of position, so the start state would have $h1 = 8$.

$h1$ is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.

- $h2$ = the sum of the distances of the tiles from their goal positions.

Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances.

- This is sometimes called the **city block distance** or **Manhattan distance**.

(i) The effect of heuristic accuracy on performance :

- One way to characterize the quality of a heuristic is the **effective branching factor** b^* .

If the total number of nodes generated by A* for a particular problem is N and the solution depth is d , then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N + 1$ nodes. Thus,

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d .$$

- For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

(ii) Generating admissible heuristics from relaxed problems :

- We have seen that both h1 (misplaced tiles) and h2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h2 is better.
- How might one have come up with h2? Is it possible for a computer to invent such a heuristic mechanically.
h1 and h2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for *simplified* versions of the puzzle.
- If the rules of the puzzle were changed, so that a tile could move anywhere instead of just to the adjacent empty square, then h1 would give the exact number of steps in the shortest solution.
- Similarly, if a tile could move one square in any direction, even onto an occupied square, then h2 would give the exact number of steps in the shortest solution.
- A problem with fewer restrictions on the actions is called a **relaxed problem**.

“the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem”.

- Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent**.
- If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.
- For example, if the 8-puzzle actions are described as

→ A tile can move from square A to square B if

- → A is horizontally or vertically adjacent to B **and** B is blank,

We can generate three relaxed problems by removing one or both of the conditions:

(a) A tile can move from square A to square B if A is adjacent to B.

(b) A tile can move from square A to square B if B is blank.

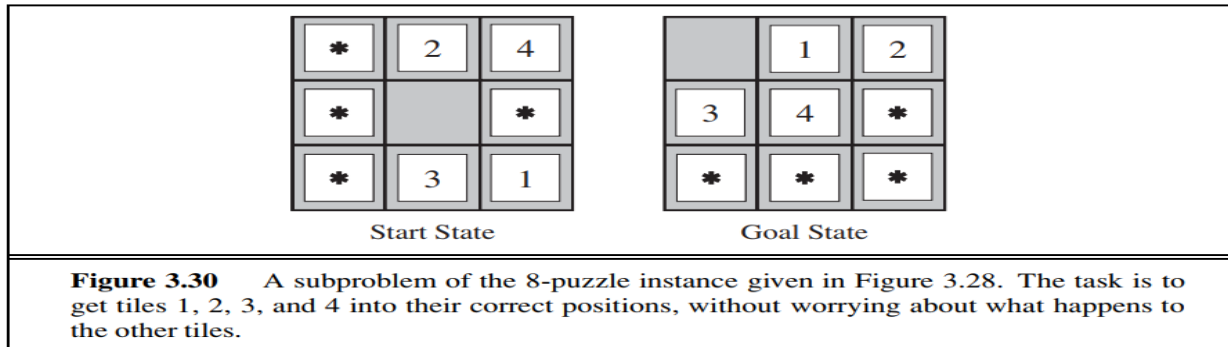
(c) A tile can move from square A to square B.

From (a), we can derive h2 (Manhattan distance). The reasoning is that h2 would be the proper score if we moved each tile in turn to its destination.

From (c), we can derive h1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step.

(iii) Generating admissible heuristics from subproblems : (Pattern databases)

- Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem.
- For example, a subproblem of the 8-puzzle instance is



- The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions.
- Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem.
- It turns out to be more accurate than Manhattan distance in some cases.
- The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank.
- Then we compute an admissible heuristic hdb for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database.
- The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered .
- Each database yields an admissible heuristic, and these heuristics can be combined, and by taking the maximum value.

A combined heuristic of this kind is much more accurate than the Manhattan distance .

- One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be *added*, since the two subproblems seem not to overlap.
- Would this still give an admissible heuristic? The answer is no.
- So, we record not the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem.
- This is the idea behind **disjoint pattern databases**.

With such databases, it is possible to solve random 15-puzzles in a few milliseconds .

(iv) Learning heuristics from experience :

- Experience” here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides examples from which $h(n)$ can be learned.

Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, a learning algorithm can be used to construct a function $h(n)$ that can (with luck) predict solution costs for other states that arise during search.

ARTIFICIAL INTELLIGENCE

UNIT – III

Syllabus

Reinforcement Learning: Introduction, Passive Reinforcement Learning, Active Reinforcement Learning, Generalization in Reinforcement Learning, Policy Search, applications of RL .

Natural Language Processing: Language Models, Text Classification, Information Retrieval, Information Extraction.

Chapter – 1

Reinforcement Learning

3.1.1. INTRODUCTION:

- A supervised learning agent needs to be told the correct move for each position it encounters, but such feedback is seldom available.
- In the absence of feedback from a teacher, an agent can learn a transition model for its own moves and can perhaps learn to predict the opponent's moves, but *without some feedback about what is good and what is bad, the agent will have no grounds for deciding which move to make.*
- The agent needs to know that something good has happened when it (accidentally) checkmates the opponent, and that something bad has happened when it is checkmated—or vice versa, if the game is suicide chess.

This kind of feedback is called a reward, or reinforcement.

- In games like chess, the reinforcement is received only at the end of the game. In other environments, the rewards come more frequently.
- In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement.
- In animals seem to be hardwired to recognize pain and hunger as negative rewards and pleasure and food intake as positive rewards.
- Rewards, where they served to define optimal policies in Markov decision processes (MDPs).
- An optimal policy is a policy that maximizes the expected total reward. The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment.
- Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose." This is reinforcement learning in a nutshell.
- In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels.

Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein. To keep the concept manageable, we will concentrate on simple environments and simple agent designs.

- Thus, the agent faces an unknown Markov decision process. We will consider three of the agent designs first :
- A **utility-based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility.
- A **Q-learning** agent learns an **action-utility function**, or **Q-function**, giving the expected utility of taking a given action in a given state.

A **reflex agent** learns a policy that maps directly from states to actions.

- A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead.
- A Q-learning agent, on the other hand, can compare the expected utilities for its available choices without needing to know their outcomes, so it does not need a model of the environment.
- On the other hand, because they do not know where their actions lead, Q-learning agents cannot look ahead; this can seriously restrict their ability to learn

(i) **Passive learning**, where the agent's policy is fixed and the task is to learn the utilities of states (or state-action pairs); this could also involve learning a model of the environment.

(ii) **Active learning**, where the agent must also learn what to do.

→ The principal issue is **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it.

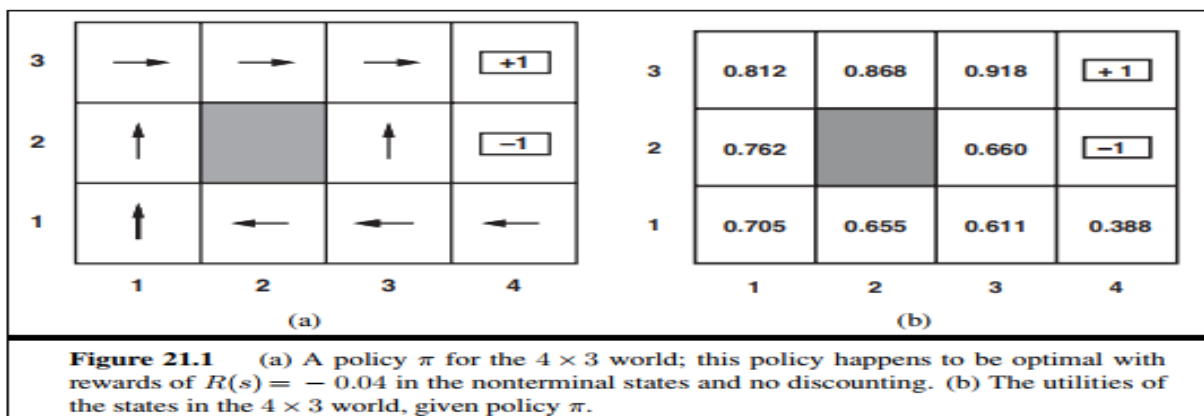
3.1.2. PASSIVE REINFORCEMENT LEARNING :

- To keep things simple, we start with the case of a passive learning agent using a state-based representation in a fully observable environment.
- In passive learning, the agent's policy π is fixed: in state s , it always executes the action $\pi(s)$.

Its goal is simply to learn how good the policy is—that is, to learn the utility function $U \pi(s)$.

- We will use as our example the 4×3 world shows a policy for that world and the corresponding utilities.
- Clearly, the passive learning task is similar to the **policy evaluation** task, part of the **policy iteration** algorithm.

The main difference is that the passive learning agent does not know the **transition model** $P(s' | s, a)$, which specifies the probability of reaching state s' from state s after doing action a ; nor does it now the **reward function** $R(s)$, which specifies the reward for each state.



- The agent executes a set of trials in the environment using its policy π . In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state.
- The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed. As in Equation we write :

where $R(s)$ is the reward for a state, S_t (a random variable) is the state reached at time t when executing policy π , and $S_0 = s$. We will include a discount factor γ in all of our equations, but for the 4×3 world we will set $\gamma = 1$.

(i) Direct utility estimation :

- A simple method for **direct utility estimation** was invented in the late 1950s in the area of **adaptive control theory** by Widrow and Hoff (1960).
- The idea is that the utility of a state is the expected total reward from that state onward (called the expected **reward-to-go**), and each trial provides a *sample* of this quantity for each state visited.
- For example, the first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1) and so on.

Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table.

- It is clear that direct utility estimation is just an instance of supervised learning where each example has the state as input and the observed reward-to-go as output.
- Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known.
- Unfortunately, it misses a very important source of information, namely, the fact that the utilities of states are not independent! *The utility of each state equals its own reward plus the expected utility of its successor states.*

That is, the utility values obey the Bellman equations for a fixed policy :

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi(s)) U^\pi(s')$$

(ii) Adaptive dynamic programming :

- An **adaptive dynamic programming** (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method.

For a passive learning agent, this means plugging the learned transition model $P(s' | s, \pi(s))$ and the observed rewards $R(s)$ into the Bellman equations to calculate the utilities of the states.

- Alternatively, we can adopt the approach of **modified policy iteration** , using a simplified value iteration process to update the utility estimates after each change to the learned model.

- Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.
- The process of learning the model itself is easy, because the environment is fully observable. This means that we have a supervised learning task where the input is a state–action pair and the output is the resulting state.

In the simplest case, we can represent the transition model as a table of probabilities.

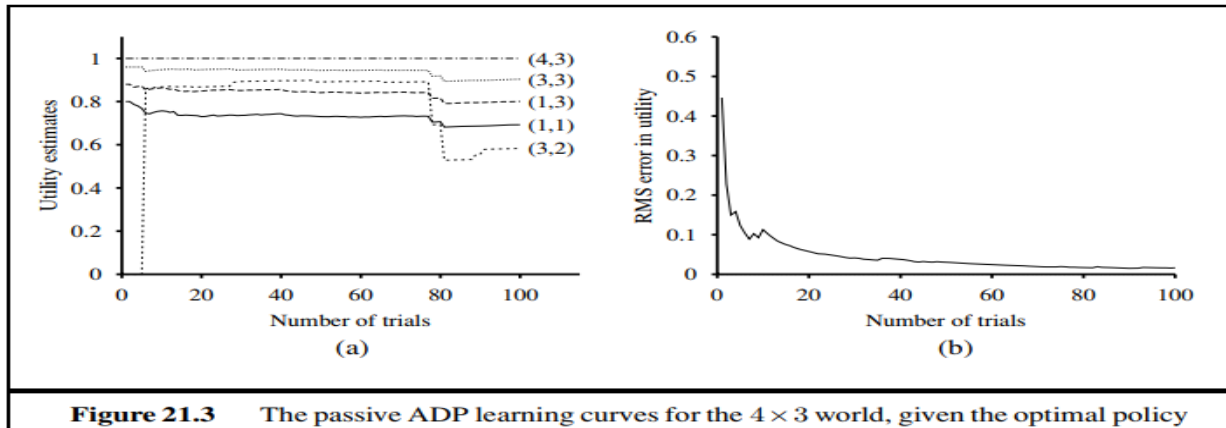


Figure 21.3 The passive ADP learning curves for the 4×3 world, given the optimal policy

- There are two mathematical approaches that have this flavor.
- The first approach, **Bayesian reinforcement learning**, assumes a prior probability $P(h)$ for each hypothesis h about what the true model is; the posterior probability $P(h | e)$ is obtained in the usual way by Bayes' rule given the observations to date.

The second approach, derived from **robust control theory**, allows for a *set* of possible models H and defines an optimal robust policy as one that gives the best outcome in the *worst case* over H :

(iii) Temporal-difference learning :

- Solving the underlying MDP as in the preceding section is not the only way to bring the Bellman equations to bear on the learning problem.
- Another way is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations.
- Consider, for example, the transition from (1,3) to (2,3). Suppose that, as a result of the first trial, the utility estimates are $U^\pi(1, 3) = 0.84$ and $U^\pi(2, 3) = 0.92$.

Now, if this transition occurred all the time, we would expect the utilities to obey the equation

$$U^\pi(1, 3) = -0.04 + U^\pi(2, 3)$$

- So $U^\pi(1, 3)$ would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state s to state s' , we apply the following update to $U^\pi(s)$:
- Here, α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states, it is often called the **temporal-difference**, or TD, equation.

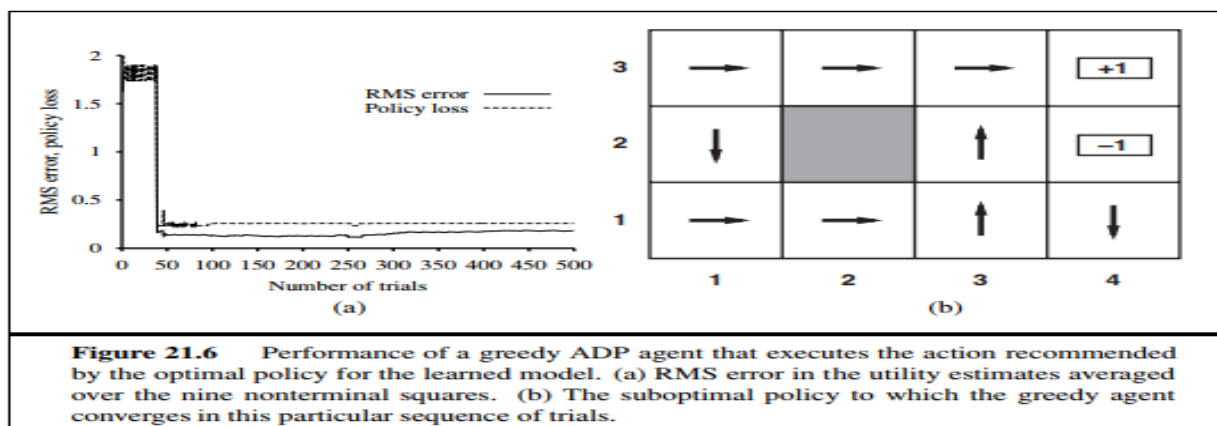
3.1.3. ACTIVE REINFORCEMENT LEARNING:

- A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take.
- Let us begin with the adaptive dynamic programming agent and consider how it must be modified to handle this new freedom.
- The utilities it needs to learn are those defined by the *optimal* policy; they obey the Bellman equations, which we repeat here for convenience:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' | s, a) U(s') .$$

(i) Exploration :

- In Figure it shows the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step.
- The agent *does not* learn the true utilities or the true optimal policy! What happens instead is that, in the 39th trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1),(3,2), and (3,3).
- After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3).
- We call this agent the **greedy agent**. Repeated experiments show that the greedy agent *very seldom* converges to the optimal policy for this environment and sometimes converges to really horrendous policies.



- By improving the model, the agent will receive greater rewards in the future. An agent therefore must make a tradeoff between **exploitation** to maximize its reward.
- Pure exploitation risks getting stuck in a rut. Pure exploration to improve one's knowledge is of no use if one never puts that knowledge into practice. With greater understanding, less exploration is necessary.
- A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes.

An ADP agent using such a scheme will eventually learn the true environment model.

- In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any).
- An **n-armed bandit** has n levers. The gambler must choose which lever to play on each successive coin—the one that has paid off best, or maybe one that has not been tried?
- This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only “by accident” thereafter.

However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided.

(ii) Learning an action-utility function :

- Now that we have an active ADP agent, let us consider how to construct an active temporal difference learning agent.
- The most obvious change from the passive case is that the agent is no longer equipped with a fixed policy, so, if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one-step look-ahead.

The model acquisition problem for the TD agent is identical to that for the ADP agent.

- Suppose the agent takes a step that normally leads to a good destination, but because of non-determinism in the environment the agent ends up in a catastrophic state. The TD update rule will take this as seriously.
- There is an alternative TD method, called **Q-learning**, which learns an action-utility representation instead of learning utilities. We will use the notation $Q(s, a)$ to denote the value of doing action a in state s. Q-values are directly related to utility values as follows:

$$U(s) = \max_a Q(s, a) .$$

- Q-functions may seem like just another way of storing utility information, but they have a very important property:

“a TD agent that learns a Q-function does not need a model of the form $P(s' | s, a)$, either for learning or for action selection.”

- For this reason, Q-learning is called a **model-free** method.
- As with utilities, we can write a constraint equation :

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' | s, a) \max_{a'} Q(s', a') .$$

```

function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $Q$ , a table of action values indexed by state and action, initially zero
                $N_{sa}$ , a table of frequencies for state–action pairs, initially zero
                $s, a, r$ , the previous state, action, and reward, initially null

  if TERMINAL?( $s$ ) then  $Q[s, None] \leftarrow r'$ 
  if  $s$  is not null then
    increment  $N_{sa}[s, a]$ 
     $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$ 
     $s, a, r \leftarrow s', \operatorname{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$ 
  return  $a$ 

```

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

- Q-learning has a close relative called SARSA (for State-Action-Reward-State-Action).
- The update rule for SARSA is

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a)) ,$$

3.1.4. GENERALIZATION IN REINFORCEMENT LEARNING :

- So far, we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple.
- Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger.
- With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more. This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question.
- Backgammon and chess are tiny subsets of the real world, yet their state spaces contain on the order of 1020 and 1040 states, respectively.
- It would be absurd to suppose that one must visit all these states many times in order to learn how to play the game.
- One way to handle such problems is to use **function approximation**, which simply means using any sort of representation for the Q-function other than a lookup table.
- The representation is viewed as approximate because it might not be the case that the *true* utility function or Q-function can be represented in the chosen form.
- For example, we described an **evaluation function** for chess that is represented as a weighted linear

function of a set of **features** (or **basis functions**) f_1, \dots, f_n :

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \dots + \theta_n f_n(s)$$

- A reinforcement learning algorithm can learn values for the parameters $\theta = \theta_1, \dots, \theta_n$ such that the evaluation function \hat{U}_θ approximates the true utility function.
- Instead of, say, values in a table, this function approximator is characterized by, say, $n = 20$ parameters an *enormous* compression.
- Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers.
- If the approximation is good enough, however, the agent might still play excellent chess.
- Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit.
- *The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.*
- That is, the most important aspect of function approximation is not that it requires less space, but that it allows for inductive generalization over input states.
- For reinforcement learning, it makes more sense to use an *online* learning algorithm that updates the parameters after each trial.
- Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that $\hat{U}_\theta(1, 1)$, currently 0.8, is too large and must be reduced.
- There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.
- Function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an *observable* environment is a *supervised* learning problem, because the next percept gives the outcome state.
- For a *partially observable* environment, the learning problem is much more difficult.
- If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters .
- Inventing the hidden variables and learning the model structure are still open problems.

3.1.5. POLICY SEARCH :

- The final approach we will consider for reinforcement learning problems is called policy search.
- In some ways, policy search is the simplest of all the methods, the idea is to keep twiddling the policy as long as its performance improves, then stop.

- Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions.
- We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space .
- For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \max_a \hat{Q}_\theta(s, a)$$

- Each Q-function could be a linear function of the parameters θ , or it could be a nonlinear function such as a neural network.
- Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q functions, then policy search results in a process that learns Q-functions.
- *This process is not the same as Q-learning!* In Q-learning with function approximation, the algorithm finds a value of θ such that \hat{Q}_θ is “close” to Q^* , the optimal Q-function.
- Policy search, on the other hand, finds a value of θ that results in good performance; the values found by the two methods may differ very substantially.

One problem with policy representations of the kind is that the policy is a *discontinuous* function of the parameters when the actions are discrete.

- That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another.
- This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult.

For this reason, policy search methods often use a **stochastic policy** representation $\pi_\theta(s, a)$, which specifies the *probability* of selecting action a in state s . One popular representation is the **softmax function**:

$$\pi_\theta(s, a) = e^{\hat{Q}_\theta(s, a)} / \sum_{a'} e^{\hat{Q}_\theta(s, a')}$$

- For the case of a stochastic policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of the gradient at θ , $\nabla_\theta \rho(\theta)$, directly from the results of trials executed at θ .
- In this case, the policy value is just the expected value of the reward, and we have

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a (\nabla_\theta \pi_\theta(s_0, a)) R(a)$$

Suppose that we have N trials in all and the action taken on the j th trial is a_j . Then

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0, a)) R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^N \frac{(\nabla_\theta \pi_\theta(s_0, a_j)) R(a_j)}{\pi_\theta(s_0, a_j)}$$

- For the sequential case, this generalizes to :

$$\nabla_{\theta} \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^N \frac{(\nabla_{\theta} \pi_{\theta}(s, a_j)) R_j(s)}{\pi_{\theta}(s, a_j)}$$

- Policy search is carried out by evaluating each candidate policy using the *same* set of random sequences to determine the action outcomes.

It can be shown that the number of random sequences required to ensure that the value of *every* policy is well estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain.

3.1.6. APPLICATIONS OF REINFORCEMENT LEARNING:

(i) Applications to game playing :

→ The first significant application of reinforcement learning was also the first significant learning program of any kind—the checkers program written by Arthur Samuel (1959, 1967).

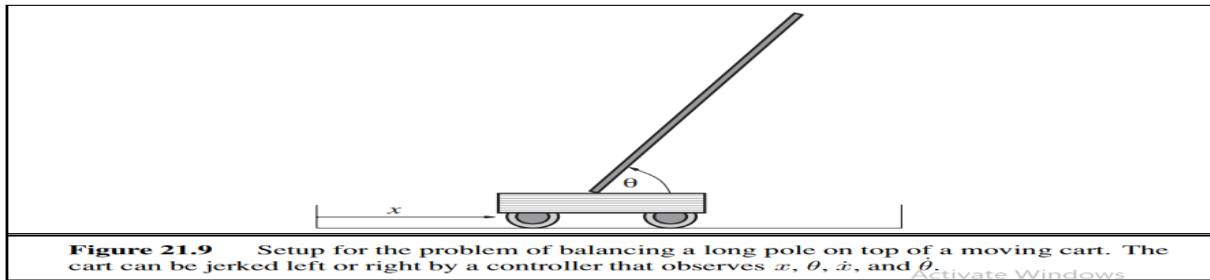
- Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation to update the weights.
- There were some significant differences, however, between his program and current methods.
- First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree.
- Second difference was that the program did *not* use any observed rewards .

→ **Gerry Tesauro's backgammon program TD-GAMMON** (1992) forcefully illustrates the potential of reinforcement learning techniques.

- In earlier work Tesauro tried learning a neural network representation of $Q(s, a)$ directly from examples of moves labeled with relative values by a human expert.
- This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts.
- The TD-GAMMON project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game.

(ii) Application to robot control :

- The setup for the famous cart–pole balancing problem, also known as the **inverted pendulum**.

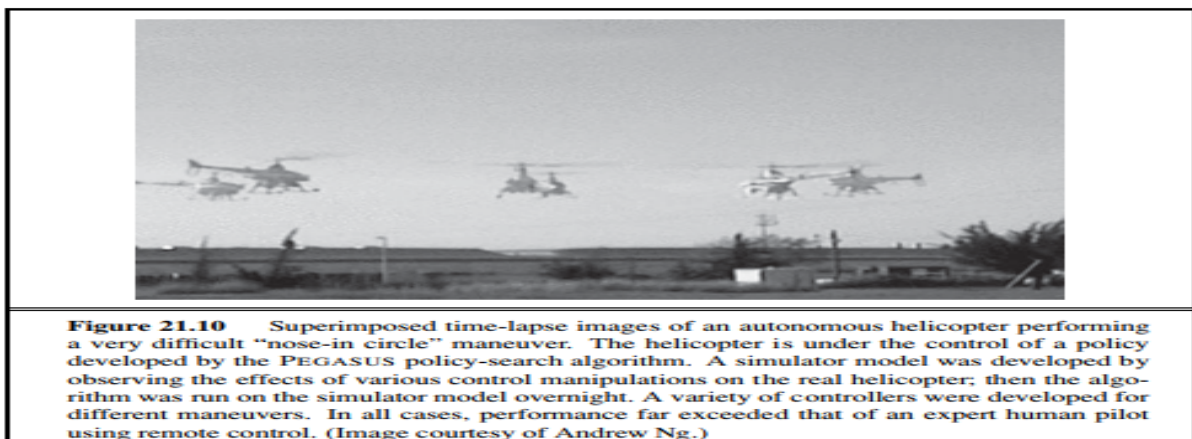


- The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown.
- The cart–pole problem differs from the problems described earlier in that the state variables x , θ , \dot{x} , and $\dot{\theta}$ are continuous.
- The actions are usually discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only about 30 trials.

- Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation.
- The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track.
- Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence.
- It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect.
- Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward, or by using a continuous-state, nonlinear function approximator such as a neural network.
- Nowadays, balancing a *triple* inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans.

➔ Still more impressive is the application of reinforcement learning to **helicopter flight**.



- This work has generally used policy search as well as the PEGASUS algorithm with simulation based on a learned transition model.

ARTIFICIAL INTELLIGENCE

UNIT – III

Chapter – 2

Natural Language Processing

3.2.1. LANGUAGE MODELS :

- Formal languages, such as the programming languages Java or Python, have precisely defined language models.
- A **language** can be defined as a set of strings; “print(2 + 2)” is a legal program in the language Python, whereas “2)+(2 print” is not.
- Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by set of rules called a **grammar**.
- Formal languages also have rules that define the meaning or **semantics** of a program; for example, the rules say that the “meaning” of “2 + 2” is 4, and the meaning of “1/0” is that an error is signaled.
- Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences.
- Everyone agrees that “Not to be invited is sad” is a sentence of English, but people disagree on the grammaticality of “To be not invited is sad.”
- Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set.
$$P(S = \text{words})$$
- Natural languages are also **ambiguous**. Because we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.
- Finally, natural languages are difficult to deal with because they are very large, and constantly changing.
- Thus, our language models are, at best, an approximation. We start with the simplest possible approximations and move up from there.

(i) **N-gram character models :**

- Ultimately, a written text is composed of **characters**—letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages).
- Thus, one of the simplest language models is a probability distribution over sequences of characters.
- We write $P(C1:N)$ for the probability of a sequence of N characters, C1 through CN.
- In one Web collection, $P(\text{“the”}) = 0.027$ and $P(\text{“zgq”}) = 0.000000002$.
- A sequence of written symbols of length n is called an n-gram (from the Greek root for writing or letters), with special case “unigram” for 1-gram, “bigram” for 2-gram, and “trigram” for 3-gram.
- A model of the probability distribution of n-letter sequences is thus called an **n-gram model**. (But be careful: we can have n-gram models over sequences of words, syllables, or other units; not just over characters.)
- An n-gram model is defined as a **Markov chain** of order n - 1.
- In a Markov chain the probability of character c_i depends only on the immediately preceding characters, not on any other characters.
- So in a trigram model (Markov chain of order 2) we have :

$$P(c_i | c_{1:i-1}) = P(c_i | c_{i-2:i-1})$$

- We can define the probability of a sequence of characters $P(c1:N)$ under the trigram model by first

factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^N P(c_i | c_{1:i-1}) = \prod_{i=1}^N P(c_i | c_{i-2:i-1})$$

- We call a body of text a **corpus** (plural *corpora*), from the Latin word for *body*.
- What can we do with n-gram character models? One task for which they are well suited is **language identification**.
- For Example, given a text, determine what natural language it is written in.
- This is a relatively easy task; even with short texts such as “Hello, world” or “Wie geht es dir,” it is easy to identify the first as English and the second as German.
- Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused.
- One approach to language identification is to first build a trigram character model of each candidate language, where the variable L ranges over languages.
- That gives us a model of $P(\text{Text} | \text{Language})$, but we want to select the most probable language given the text, so we apply Bayes’ rule followed by the Markov assumption to get the most probable language:

$$\begin{aligned} \ell^* &= \operatorname{argmax}_{\ell} P(\ell | c_{1:N}) \\ &= \operatorname{argmax}_{\ell} P(\ell) P(c_{1:N} | \ell) \\ &= \operatorname{argmax}_{\ell} P(\ell) \prod_{i=1}^N P(c_i | c_{i-2:i-1}, \ell) \end{aligned}$$

(ii) Smoothing *n*-gram models :

- The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution.
- For common character sequences such as “_th” any English corpus will give a good estimate: about 1.5% of all trigrams. On the other hand, “_ht” is very uncommon—no dictionary words start with ht. The process of adjusting the probability of low-frequency counts is called **smoothing**.
- The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for $P(X = \text{true})$ should be $1/(n+2)$.
- That is, he assumes that with two more trials, one might be true and one false. Laplace smoothing (also called add-one smoothing) is a step in the right direction, but performs relatively poorly.
- A better approach is a **backoff model**, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n - 1)-grams.
- **Linear interpolation smoothing** is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as :

$$\hat{P}(c_i | c_{i-2:i-1}) = \lambda_3 P(c_i | c_{i-2:i-1}) + \lambda_2 P(c_i | c_{i-1}) + \lambda_1 P(c_i)$$

- where $\lambda_3 + \lambda_2 + \lambda_1 = 1$. The parameter values λ_i can be fixed, or they can be trained with an expectation–maximization algorithm.
- It is also possible to have the values of λ_i depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models.

(iii) Model evaluation :

- With so many possible n-gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ , etc.—how do we know what model to choose? We can evaluate a model with cross-validation.
- Split the corpus into a training corpus and a validation corpus. Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus.
- The evaluation can be a task-specific metric, such as measuring accuracy on language identification.
- Alternatively we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better.
- This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue.
- A different way of describing the probability of a sequence is with a measure called **perplexity**, defined as :

$$\text{Perplexity}(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}}$$

- Perplexity can be thought of as the reciprocal of probability, normalized by sequence length.
- It can also be thought of as the weighted average branching factor of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100.
- If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

(iv) N-gram word models :

- Now we turn to n-gram models over words rather than characters.
- All the same mechanism applies equally to word and character models. The main difference is that the **vocabulary**—the set of symbols that make up the corpus and the model—is larger.
- There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating “A” and “a” as the same symbol or by treating all punctuation as the same symbol.
- But with word models we have at least tens of thousands of symbols, and sometimes millions.
- The wide range is because it is not clear what constitutes a word.
- In English a sequence of letters surrounded by spaces is a word, but in some languages, like Chinese, words are not separated by spaces, and even in English many decisions must be made to have a clear policy on word boundaries: how many words are in.

- Word n-gram models need to deal with **out of vocabulary** words.
- With character models, we didn't have to worry about someone inventing a new letter of the alphabet.
- But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model.

3.2.2. TEXT CLASSIFICATION :

- We now consider in depth the task of **text classification**, also known as **categorization**: given a text of some kind, decide which of a predefined set of classes it belongs to.
- Language identification and genre classification are examples of text classification
- **spam detection** classifying an email message as spam or not-spam(ham).
- A training set is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox.
- Note that we have two complementary ways of talking about classification.
- In the language-modeling approach, we define one n-gram language model for $P(\text{Message} \mid \text{spam})$ by training on the spam folder, and one model for $P(\text{Message} \mid \text{ham})$ by training on the inbox.
- Then we can classify a new message with an application of Bayes' rule:

$$\operatorname{argmax}_{c \in \{\text{spam}, \text{ham}\}} P(c \mid \text{message}) = \operatorname{argmax}_{c \in \{\text{spam}, \text{ham}\}} P(\text{message} \mid c) P(c)$$

- where $P(c)$ is estimated just by counting the total number of spam and ham messages. This approach works well for spam detection, just as it did for language identification.
- If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero.
- This unigram representation has been called the **bag of words** model.
- You can think of the model as putting the words of the training corpus in a bag and then selecting words one at a time.
- The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text.
- Higher-order n-gram models maintain some local notion of word order.
- It can be expensive to run algorithms on a very large feature vector, so often a process of **feature selection** is used to keep only the features that best discriminate between spam and ham.
- Once we have chosen a set of features, we can apply any of the supervised learning techniques we have seen; popular ones for text categorization include k-nearest-neighbors, support vector machines, decision trees, naive Bayes, and logistic regression.
- All of these have been applied to spam detection, usually with accuracy in the 98%–99% range. With a carefully designed feature set, accuracy can exceed 99.9%.

(i) **Classification by data compression :**

- Another way to think about classification is as a problem in **data compression**.
- A lossless compression algorithm takes a sequence of symbols, detects repeated patterns in it, and writes a description of the sequence that is more compact than the original.
- For example, the text “0.142857142857142857” might be compressed to “0.[142857]*3.”
- To do classification by compression, we first lump together all the spam training messages and compress them as a unit.
- We do the same for the ham. Then when given a new message to classify, we append it to the spam messages and compress the result.
- We also append it to the ham and compress that. Whichever class compresses better—adds the fewer number of additional bytes for the new message—is the predicted class.

3.2.3. 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 “AI book” into a search engine and see a list of relevant pages.
- An **information retrieval** (henceforth **IR**) system can be **characterized** by :

→ **A corpus of documents.** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.

→ **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].

→ **A result set.** This is the subset of documents that the IR system judges to be **relevant** to the query.

→ **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.

- The earliest IR systems worked on a **Boolean keyword model**. Each word in the document collection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not.

The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true.

- This model has the advantage of being simple to explain and implement.
- However, it has some **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.
- ➔ Third, it can be hard to formulate an appropriate query, even for a skilled user.

(i) **IR scoring functions :**

- Most IR systems have abandoned the Boolean model and use models based on the statistics of word counts. We describe the **BM25 scoring function**.
- A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores.

In the BM25 function, the score is a linear weighted combination of scores for each of the words that make up the query.

- **Three factors** affect the weight of a query term:
 - ➔ First, the frequency with which a query term appears in a document (also known as TF for term frequency). For the query documents that mention “farming” frequently will have higher scores.
 - ➔ Second, the inverse document frequency of the term, or IDF. The word “in” appears in almost every document, so it has a high document frequency, and thus a low inverse document frequency, and thus it is not as important to the query.
 - ➔ Third, the length of the document. A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate.
- The BM25 function takes all three of these into account.
- Then, given a document d_j and a query consisting of the words $q_{1:N}$, we have :

$$BM25(d_j, q_{1:N}) = \sum_{i=1}^N IDF(q_i) \cdot \frac{TF(q_i, d_j) \cdot (k + 1)}{TF(q_i, d_j) + k \cdot (1 - b + b \cdot \frac{|d_j|}{L})}$$

- $IDF(q_i)$ is the inverse document frequency of word q_i , given by :

$$IDF(q_i) = \log \frac{N - DF(q_i) + 0.5}{DF(q_i) + 0.5} .$$

(ii) **IR system evaluation :**

- 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
 - ➔ precision.
- **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 $1 - .60 = .40$.
- In a very large document collection, such as the World Wide Web, recall is difficult to compute, because there is no easy way to examine every page on the Web for relevance.
- All we can do is either estimate recall by sampling or ignore recall completely and just judge precision.

(iii) IR refinements :

- There are many possible refinements to the system described here, and indeed Web search engines are continually updating their algorithms as they discover new approaches and as the Web grows and changes.
- One common refinement is a better model of the effect of document length on relevance.
- Singhal *et al.* (1996) observed that simple document length normalization schemes tend to favor short documents too much and long documents not enough.
- They propose a *pivoted* document length normalization scheme; the idea is that the pivot is the document length at which the old-style normalization is correct; documents shorter than that get a boost and longer ones get a penalty.
- The BM25 scoring function uses a word model that treats all words as completely independent, but we know that some words are correlated.
- Many IR systems attempt to account for these correlations.
- 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 can hurt precision.
- 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 publication data.
- On the Web, hypertext **links** between documents are a crucial source of information.

(iv) The PageRank algorithm :

- **PageRank** was one of the two original ideas that set Google’s search apart from other Web search engines when it was introduced in 1997. (The other innovation was the use of anchor text—the underlined text in a hyperlink).
- PageRank was invented to solve the problem of the tyranny of TF scores: if the query is [IBM], how do we make sure that IBM’s home page, *ibm.com*, is the first result, even if another page mentions the term “IBM” more frequently?

The idea is that *ibm.com* has many in-links (links to the page), so it should be ranked higher: each in-link is a vote for the quality of the linked-to page.

- But if we only counted in-links, then it would be possible for a Web spammer to create a network of pages and have them all point to a page of his choosing, increasing the score of that page.

- Therefore, the PageRank algorithm is designed to weight links from high-quality sites more heavily.
- What is a highquality site? One that is linked to by other high-quality sites.
- The definition is recursive, but we will see that the recursion bottoms out properly. The PageRank for a page p is defined as:

$$PR(p) = \frac{1-d}{N} + d \sum_i \frac{PR(in_i)}{C(in_i)}$$

- where $PR(p)$ is the PageRank of page p , N is the total number of pages in the corpus, in_i are the pages that link in to p , and $C(in_i)$ is the count of the total number of out-links on page in_i .
- The constant d is a damping factor. It can be understood through the **random surfer model** : imagine a Web surfer who starts at some random page and begins exploring.

(v) The HITS algorithm :

- The Hyperlink-Induced Topic Search algorithm, also known as “Hubs and Authorities” or HITS, is another influential link-analysis algorithm .
- HITS differs from PageRank in several ways.
- First, it is a query-dependent measure: it rates pages with respect to a query.
- Given a query, HITS first finds a set of pages that are relevant to the query. It does that by intersecting hit lists of query words, and then adding pages in the link neighborhood of these pages
- Both PageRank and HITS played important roles in developing our understanding of Web information retrieval.
- These algorithms and their extensions are used in ranking billions of queries daily as search engines steadily develop better ways of extracting yet finer signals of search relevance.

(vi) Question answering :

- Information retrieval is the task of finding documents that are relevant to a query, where the query may be a question, or just a topic area or concept.
- **Question answering** is a somewhat different task, in which the query really is a question, and the answer is not a ranked list of documents but rather a short response—a sentence, or even just a phrase.
- There have been question-answering NLP (natural language processing) systems since the 1960s, but only since 2001 have such systems used Web information retrieval to radically increase their breadth of coverage.

3.2.4. INFORMATION EXTRACTION :

- Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of a particular class of object and for relationships among objects.
- A typical task is 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.
- In a limited domain, this can be done with high accuracy. As the domain gets more general, more complex linguistic models and more complex learning techniques are necessary.

(i) Finite-state automata for information extraction:

- The simplest type of information extraction system is an **attribute-based extraction** system that assumes that the entire text refers to a single object and the task is to extract attributes of that object.
- For example, the problem of extracting from the text “IBM ThinkBook 970. Our price: \$399.00” the set of attributes {Manufacturer=IBM, Model=ThinkBook970, Price=\$399.00}.
- We can address this problem by defining a **template** (also known as a pattern) for each attribute we would like to extract. The template is defined by a finite state automaton, the simplest example of which is the **regular expression**, or regex.
- Here we show how to build up a regular expression template for prices in dollars:

[0-9]	matches any digit from 0 to 9
[0-9]+	matches one or more digits
[.] [0-9] [0-9]	matches a period followed by two digits
([.] [0-9] [0-9]) ?	matches a period followed by two digits, or nothing
[\$] [0-9]+ ([.] [0-9] [0-9]) ?	matches \$249.99 or \$1.23 or \$1000000 or ...

- Templates are often defined with three parts: a prefix regex, a target regex, and a postfix regex.
- For prices, the target regex is as above, the prefix would look for strings such as “price:” and the postfix could be empty.
- The idea is that some clues about an attribute come from the attribute value itself and some come from the surrounding text.
- One step up from attribute-based extraction systems are **relational extraction** systems, which deal with multiple objects and the relations among them.
- Thus, when these systems see the text “\$249.99,” they need to determine not just that it is a price, but also which object has that price.
- A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions.
- A relational extraction system can be built as a series of **cascaded finite-state transducers**.
- That is, the system consists of a series of small, efficient 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.

- **FASTUS consists of five stages:**

1. Tokenization
2. Complex-word handling
3. Basic-group handling
4. Complex-phrase handling
5. Structure merging

1. FASTUS's first stage is **tokenization**, which segments the stream of characters into tokens (words, numbers, and punctuation). Some tokenizers also deal with markup languages such as HTML, SGML, and XML.
2. The second stage handles **complex words**, including collocations such as “set up” and “joint venture,” as well as proper names such as “Bridgestone Sports Co.”
3. 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.
4. The fourth stage combines the basic groups into **complex phrases**.
5. The final stage **merges structures** that were built up in the previous step.

(ii) Probabilistic models for information extraction:

- When information extraction must be attempted from noisy or varied input, simple finite-state approaches fare poorly.
- It is too hard to get all the rules and their priorities right; it is better to use a probabilistic model rather than a rule-based model.
- The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM.
- HMM models a progression through a sequence of hidden states, **xt**, with an observation **et** at each step.
- To apply HMMs to information extraction, we can either build one big HMM for all the attributes or build a separate HMM for each attribute. We'll do the second.
- HMMs have **two big advantages** over FSAs for extraction.
 - First, HMMs are probabilistic, and thus tolerant to noise.
 - Second, HMMs can be trained from data; they don't require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.

(iii) Conditional random fields for information extraction :

- One issue with HMMs for the information extraction task is that they model a lot of probabilities that we don't really need.
- Modeling this directly gives us some freedom. We don't need the independence assumptions of the Markov model—we can have an **xt** that is dependent on **x1**.
- A framework for this type of model is the **conditional random field**, or CRF, which models a

conditional probability distribution of a set of target variables given a set of observed variables.

- Like Bayesian networks, CRFs can represent many different structures of dependencies among the variables.
- One common structure is the **linear-chain conditional random field** for representing Markov dependencies among variables in a temporal sequence.
- Thus, HMMs are the temporal version of naive Bayes models, and linear-chain CRFs are the temporal version of logistic regression.

(iv) Ontology extraction from large corpora :

- So far we have thought of information extraction as finding a specific set of relations (e.g., speaker, time, location) in a specific text (e.g., a talk announcement).
- A different application of extraction technology is building a large knowledge base or ontology of facts from a corpus.
- This is different in three ways:
- First it is open-ended—we want to acquire facts about all types of domains, not just one specific domain.
- Second, with a large corpus, this task is dominated by precision, not recall—just as with question answering on the Web .
- Third, the results can be statistical aggregates gathered from multiple sources, rather than being extracted from one specific text.

(v) Automated template construction :

- Fortunately, it is possible to *learn* templates from a few examples, then use the templates to learn more examples, from which more templates can be learned, and so on.
- In one of the first experiments of this kind, Brin (1999) started with a data set of just five examples:

(“Isaac Asimov”, “The Robots of Dawn”)
(“David Brin”, “Startide Rising”)
(“James Gleick”, “Chaos—Making a New Science”)
(“Charles Dickens”, “Great Expectations”)
(“William Shakespeare”, “The Comedy of Errors”)

- Clearly these are examples of the author–title relation, but the learning system had no knowledge of authors or titles.
- The words in these examples were used in a search over a Web corpus, resulting in 199 matches. Each match is defined as a tuple of seven strings,

(*Author, Title, Order, Prefix, Middle, Postfix, URL*) ,

- where *Order* is true if the author came first and false if the title came first, *Middle* is the characters between the author and title, *Prefix* is the 10 characters before the match, *Suffix* is the 10 characters

after the match, and *URL* is the Web address where the match was made.

(vi) Machine reading :

- Automated template construction is a big step up from handcrafted template construction, but it still requires a handful of labeled examples of each relation to get started.
- To build a large ontology with many thousands of relations, even that amount of work would be onerous; we would like to have an extraction system with *no* human input of any kind—a system that could read on its own and build up its own database.
- Such a system would be relation-independent; would work for any relation. In practice, these systems work on *all* relations in parallel, because of the I/O demands of large corpora.
- They behave less like a traditional information extraction system that is targeted at a few relations and more like a human reader who learns from the text itself; because of this the field has been called **machine reading**.

ARTIFICIAL INTELLIGENCE

UNIT – IV

Syllabus

Natural Language for Communication: Phrase structure grammars, Syntactic Analysis, Augmented Grammars and semantic Interpretation, Machine Translation, Speech Recognition.

Perception: Image Formation, Early Image Processing Operations, Object Recognition by appearance, Reconstructing the 3D World, Object Recognition from Structural information, Using Vision.

Chapter – 1

Natural Language for Communication

INTRODUCTION

Communication is the intentional exchange of information brought about by the production SIGN and perception of signs drawn from a shared system of conventional signs. Most animals use signs to represent important messages: food here, predator nearby, approach, withdraw, let's mate.

4.1.1. PHRASE STRUCTURE GRAMMARS :

- The n-gram language models were based on sequences of words.
- The big issue for these models is **data sparsity**—with a vocabulary of, say, trigram probabilities to estimate, and so a corpus of even a trillion words will not be able to supply reliable estimates for all of them.
- We can address the problem of sparsity through generalization.
- Despite the exceptions, the notion of a **lexical category** (also known as a **part of speech**) such as *noun* or *adjective* is a useful generalization—useful in its own right, but more so when we string together lexical categories to form **syntactic categories** such as *noun phrase* or *verb phrase*, and combine these syntactic categories into trees representing the **phrase structure** of sentences: nested phrases, each marked with a category .

GENERATIVE CAPACITY :

- Grammatical formalisms can be classified by their **generative capacity**: the set of languages they can represent.
- Chomsky (1957) describes four classes of grammatical formalisms that differ only in the form of the rewrite rules.
- The classes can be arranged in a hierarchy, where each class can be used to describe all the languages that can be described by a less powerful class, as well as some additional languages.
- Here we list the hierarchy, most powerful class first:
 1. **Recursively enumerable** grammars use unrestricted rules: both sides of the rewrite rules can have any number of terminal and nonterminal symbols, as in the rule $A B C \rightarrow D E$.
 - These grammars are equivalent to Turing machines in their expressive power.
 2. **Context-sensitive grammars** are restricted only in that the right-hand side must contain at least as many symbols as the left-hand side.
 - The name “context-sensitive” comes from the fact that a rule such as $A X B \rightarrow A Y B$ says that an X can be rewritten as a Y in the context of a preceding A and a following B.

Context-sensitive grammars can represent languages such as (a sequence of n copies of a followed by the same number of bs and then cs).

3. In **context-free grammars** (or **CFGs**), the left-hand side consists of a single nonterminal symbol. Thus, each rule licenses rewriting the nonterminal as the right-hand side in *any* context.

CFGs are popular for natural-language and programming-language grammars, although it is now widely accepted that at least some natural languages have constructions that are not context-free (Pullum, 1991).

Context-free grammars can represent $a^n b^n$, but not $a^n b^n c^n$.

4. **Regular** grammars are the most restricted class. Every rule has a single nonterminal on the left-hand side and a terminal symbol optionally followed by a nonterminal on the right-hand side.

Regular grammars are equivalent in power to finite state machines. They are poorly suited for programming languages, because they cannot represent constructs such as balanced opening and closing parentheses.

The closest they can come is representing a^*b^* , a sequence of any number of *a*s followed by any number of *b*s.

- There have been many competing language models based on the idea of phrase structure; we will describe a popular model called the **probabilistic context-free grammar**, or PCFG.
- A **grammar** is a collection of rules that defines a **language** as a set of allowable strings of words. Probabilistic means that the grammar assigns a probability to every string.
- Here is a PCFG rule:

$VP \rightarrow \text{Verb} [0.70]$

$VP \rightarrow NP [0.30]$

- Here *VP* (*verb phrase*) and *NP* (*noun phrase*) are **non-terminal symbols**. The grammar also refers to actual words, which are called **terminal symbols**.
- This rule is saying that with probability 0.70 a verb phrase consists solely of a verb, and with probability 0.30 it is a VP followed by an NP.

(i) **The lexicon of**

- First we define the **lexicon**, or list of allowable words. The words are grouped into the lexical categories familiar to dictionary users: nouns, pronouns, and names to denote things; verbs to denote events; adjectives to modify nouns; adverbs to modify verbs; and function words: articles (such as *the*), prepositions (*in*), and conjunctions (*and*).
- Each of the categories ends in . . . to indicate that there are other words in the category.
- For nouns, names, verbs, adjectives, and adverbs, it is infeasible even in principle to list all the words. Not only are there tens of thousands of members in each class, but new ones—like *iPod* or

biodiesel—are being added constantly.

- These five categories are called **open classes**.
- For the categories of pronoun, relative pronoun, article, preposition, and conjunction we could have listed all the words with a little more work. These are called **closed classes**; they have a small number of words (a dozen or so).
- Closed classes change over the course of centuries, not months. For example, “thee” and “thou” were commonly used pronouns in the 17th century, were on the decline in the 19th, and are seen today only in poetry and some regional dialects.

(ii) The Grammar of \mathcal{E}_0

- The next step is to combine the words into phrases.

A grammar for \mathcal{E}_0 with rules for each of the six syntactic categories and an example for each rewrite rule.

$\mathcal{E}_0 :$	$S \rightarrow NP VP$	[0.90]	I + feel a breeze
	$S Conj S$	[0.10]	I feel a breeze + and + It stinks
NP	$\rightarrow Pronoun$	[0.30]	I
	$Name$	[0.10]	John
	$Noun$	[0.10]	pits
	$Article Noun$	[0.25]	the + wumpus
	$Article Adjs Noun$	[0.05]	the + smelly dead + wumpus
	$Digit Digit$	[0.05]	3 4
	$NP PP$	[0.10]	the wumpus + in 1 3
VP	$NP RelClause$	[0.05]	the wumpus + that is smelly
	$\rightarrow Verb$	[0.40]	stinks
	$VP NP$	[0.35]	feel + a breeze
	$VP Adjective$	[0.05]	smells + dead
	$VP PP$	[0.10]	is + in 1 3
$Adjs$	$VP Adverb$	[0.10]	go + ahead
	$\rightarrow Adjective$	[0.80]	smelly
PP	$Adjective Adjs$	[0.20]	smelly + dead
	$\rightarrow Prep NP$	[1.00]	to + the east
$RelClause$	$\rightarrow RelPro VP$	[1.00]	that + is smelly

Figure 23.2 The grammar for \mathcal{E}_0 , with example phrases for each rule. The syntactic categories are sentence (S), noun phrase (NP), verb phrase (VP), list of adjectives ($Adjs$), prepositional phrase (PP), and relative clause ($RelClause$).

Activ.

4.1.2. SYNTACTIC ANALYSIS (PARSING) :

- **Parsing** is the process of analyzing a string of words to uncover its phrase structure, according to the rules of a grammar.
- Consider the following two sentences:

1. Have the students in section 2 of Computer Science 101 take the exam.

2. Have the students in section 2 of Computer Science 101 taken the exam?

- Even though they share the first 10 words, these sentences have very different parses, because the first is a command and the second is a question.
- By using left-to-right parsing algorithm would have to guess whether the first word is part of a command or a question and will not be able to tell if the guess is correct until at least the eleventh word, *take* or *taken*.
- If the algorithm guesses wrong, it will have to backtrack all the way to the first word and reanalyze the whole sentence under the other interpretation.
- To avoid this source of inefficiency we can use dynamic programming: *every time we analyze a substring, store the results so we won't have to reanalyze it later*.
- For example, once we discover that “the students in section 2 of Computer Science 101” is an NP, we can record that result in a data structure known as a **chart**.
- Algorithms that do this are called **chart parsers**.
- There are many types of chart parsers; we describe a bottom-up version called the **CYK algorithm**, after its inventors, **John Cocke, Daniel Younger, and Tadeo Kasami**.

CYK algorithm :

```
function CYK-PARSE(words, grammar) returns P, a table of probabilities
  N ← LENGTH(words)
  M ← the number of nonterminal symbols in grammar
  P ← an array of size [M, N, N], initially all 0
  /* Insert lexical rules for each word */
  for i = 1 to N do
    for each rule of form (X → wordsi [p]) do
      P[X, i, 1] ← p
  /* Combine first and second parts of right-hand sides of rules, from short to long */
  for length = 2 to N do
    for start = 1 to N − length + 1 do
      for len1 = 1 to N − 1 do
        len2 ← length − len1
        for each rule of the form (X → Y Z [p]) do
          P[X, start, length] ← MAX(P[X, start, length],
                                     P[Y, start, len1] × P[Z, start + len1, len2] × p)
  return P
```

Figure 23.5 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable derivation for the whole sequence and for each subsequence. It returns the whole table, *P*, in which an entry *P*[*X*, *start*, *len*] is the probability of the most probable *X* of length *len* starting at position *start*. If there is no *X* of that size at that location, the probability is 0.

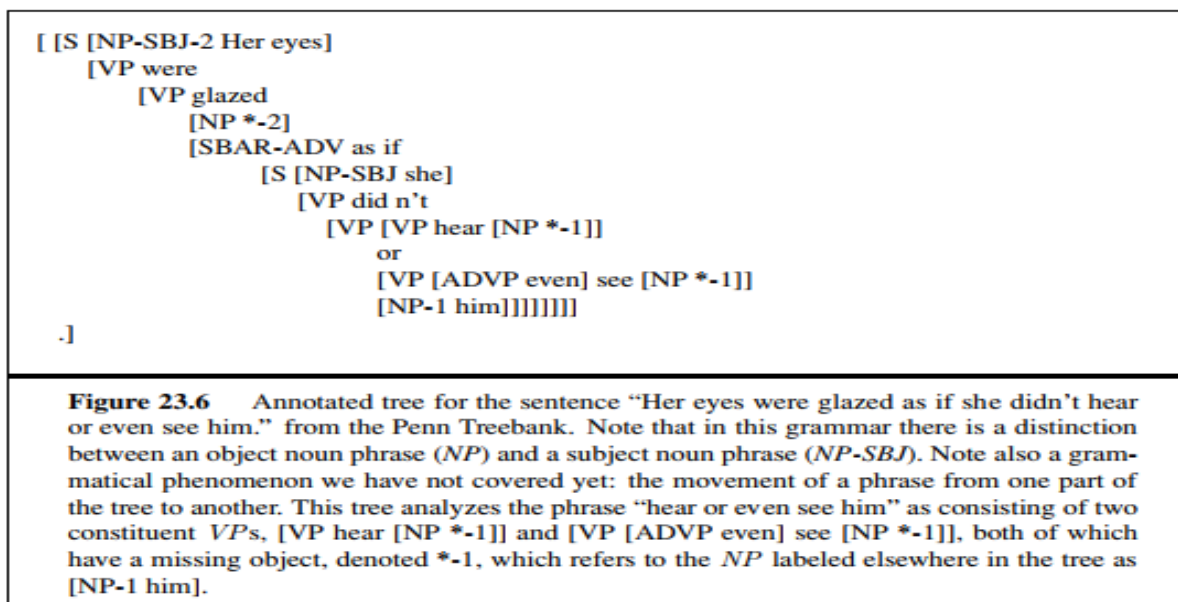
- The CYK algorithm requires a grammar with all rules in one of two very specific formats: lexical rules of the form $X \rightarrow \text{word}$, and syntactic rules of the form $X \rightarrow Y Z$.
- This grammar format, called **Chomsky Normal Form**, may seem restrictive, but it is not: any context-free grammar can be automatically transformed into Chomsky Normal Form.

(i) **Learning probabilities for PCFGs :**

- A PCFG has many rules, with a probability for each rule.
- This suggests that **learning** the grammar from data might be better than a knowledge engineering approach.
- Learning is easiest if we are given a corpus of correctly parsed sentences, commonly called a **treebank**.

The Penn Treebank is the best known; it consists of 3 million words which have been annotated with part of speech and parse-tree structure, using human labor assisted by some automated tools.

- Annotated tree from the Penn Treebank :



- Given a corpus of trees, we can create a PCFG just by counting (and smoothing).
- In the example above, there are two nodes of the form $[S[NP \dots][VP \dots]]$. We would count these, and all the other subtrees with root *S* in the corpus.

If there are 100,000 *S* nodes of which 60,000 are of this form, then we create the rule:

$$S \rightarrow NP VP [0.60] .$$

(ii) **Comparing context-free and Markov models :**

- The problem with PCFGs is that they are context-free.
- That means that the difference between $P(\text{"eat a banana"})$ and $P(\text{"eat a bandanna"})$ depends only on $P(\text{Noun} \rightarrow \text{"banana"})$ versus $P(\text{Noun} \rightarrow \text{"bandanna"})$ and not on the relation between “eat” and the respective objects.
- A Markov model of order two or more, given a sufficiently large corpus, *will* know that “eat a banana” is more probable.
- We can combine a PCFG and Markov model to get the best of both. The simplest approach is to estimate the probability of a sentence with the geometric mean of the probabilities computed by both models.

Another problem with PCFGs is that they tend to have too strong a preference for shorter sentences.

4.1.3. AUGMENTED GRAMMARS AND SEMANTIC INTERPRETATION :

- In this concept, we see how to extend context-free grammars.

(i) Lexicalized PCFGs :

To get at the relationship between the verb “eat” and the nouns “banana” versus “bandanna,” we can use a **lexicalized PCFG**, in which the probabilities for a rule depend on the relationship between words in the parse tree, not just on the adjacency of words in a sentence.

- Of course, we can’t have the probability depend on every word in the tree, because we won’t have enough training data to estimate all those probabilities.
- It is useful to introduce the notion of the **head** of a phrase—the most important word. Thus, “eat” is the head of the VP “eat a banana” and “banana” is the head of the NP “a banana.”
- We use the notation $VP(v)$ to denote a phrase with category VP whose head word is v . We say that the category VP is **augmented** with the head variable v .

Here is an **augmented grammar** that describes the verb–object relation:

$$\begin{array}{ll}
 VP(v) \rightarrow Verb(v) NP(n) & [P_1(v, n)] \\
 VP(v) \rightarrow Verb(v) & [P_2(v)] \\
 NP(n) \rightarrow Article(a) Adjs(j) Noun(n) & [P_3(n, a)] \\
 Noun(\text{banana}) \rightarrow \text{banana} & [p_n] \\
 \dots & \dots
 \end{array}$$

(ii) Formal definition of augmented grammar rules:

- Augmented rules are complicated, so we will give them a formal definition by showing how an augmented rule can be translated into a logical sentence.

- The sentence will have the form of a definite clause, so the result is called a **definite clause grammar**, or DCG.

That gives us

$$Article(a, s_1) \wedge Adjs(j, s_2) \wedge Noun(n, s_3) \wedge Compatible(j, n) \\ \Rightarrow NP(n, Append(s_1, s_2, s_3)) .$$

- This definite clause says that if the predicate *Article* is true of a head word *a* and a string *s*₁, and *Adjs* is similarly true of a head word *j* and a string *s*₂, and *Noun* is true of a head word *n* and a string *s*₃, and if *j* and *n* are compatible, then the predicate *NP* is true of the head word *n* and the result of appending strings *s*₁, *s*₂, and *s*₃.
- The translation from grammar rule to definite clause allows us to talk about parsing as logical inference.
- This makes it possible to reason about languages and strings in many different ways.
For example, it means we can do bottom-up parsing using forward chaining or top-down parsing using backward chaining .
- In fact, parsing natural language with DCGs was one of the first applications of (and motivations for) the Prolog logic programming language.
- It is sometimes possible to run the process backward and do **language generation** as well as parsing.

(iii) **Case agreement and subject–verb agreement:**

- We splitting NP into two categories, NPS and NPO, to stand for noun phrases in the subjective and objective case, respectively.
- We would also need to split the category Pronoun into the two categories PronounS (which includes “I”) and PronounO (which includes “me”).
- The top part of Figure shows the grammar for **case agreement**; we call the resulting language \mathcal{E}_1

$\mathcal{E}_1 :$	$S \rightarrow NP_S VP \mid \dots$
	$NP_S \rightarrow Pronoun_S \mid Name \mid Noun \mid \dots$
	$NP_O \rightarrow Pronoun_O \mid Name \mid Noun \mid \dots$
	$VP \rightarrow VP NP_O \mid \dots$
	$PP \rightarrow Prep NP_O$
	$Pronoun_S \rightarrow \mathbf{I} \mid \mathbf{you} \mid \mathbf{he} \mid \mathbf{she} \mid \mathbf{it} \mid \dots$
	$Pronoun_O \rightarrow \mathbf{me} \mid \mathbf{you} \mid \mathbf{him} \mid \mathbf{her} \mid \mathbf{it} \mid \dots$
	\dots
$\mathcal{E}_2 :$	$S(head) \rightarrow NP(Sbj, pn, h) VP(pn, head) \mid \dots$
	$NP(c, pn, head) \rightarrow Pronoun(c, pn, head) \mid Noun(c, pn, head) \mid \dots$
	$VP(pn, head) \rightarrow VP(pn, head) NP(Obj, p, h) \mid \dots$
	$PP(head) \rightarrow Prep(head) NP(Obj, pn, h)$
	$Pronoun(Sbj, 1S, \mathbf{I}) \rightarrow \mathbf{I}$
	$Pronoun(Sbj, 1P, \mathbf{we}) \rightarrow \mathbf{we}$
	$Pronoun(Obj, 1S, \mathbf{me}) \rightarrow \mathbf{me}$
	$Pronoun(Obj, 3P, \mathbf{them}) \rightarrow \mathbf{them}$
	\dots

Figure 23.7 Top: part of a grammar for the language \mathcal{E}_1 , which handles subjective and objective cases in noun phrases and thus does not overgenerate quite as badly as \mathcal{E}_0 . The portions that are identical to \mathcal{E}_0 have been omitted. Bottom: part of an augmented grammar for \mathcal{E}_2 , with three augmentations: case agreement, subject–verb agreement, and head word. *Sbj*, *Obj*, *1S*, *1P* and *3P* are constants, and lowercase names are variables.

- Unfortunately, \mathcal{E}_1 still overgenerates. English requires **subject–verb agreement** for person and number of the subject and main verb of a sentence.

For example, if “I” is the subject, then “I smell” is grammatical, but “I smells” is not. If “it” is the subject, we get the reverse.

(iv) Semantic interpretation :

To show how to add semantics to a grammar, we start with an example that is simpler than English: the semantics of 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 23.8 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 shows a grammar for arithmetic expressions, where each rule is augmented with a variable indicating the semantic interpretation of the phrase.
- The semantics of a digit such as “3” is the digit itself. The semantics of an expression such as “3 + 4” is the operator “+” applied to the semantics of the phrase “3” and the phrase “4.”

The rules obey the principle of **compositional semantics**.

- The semantics of a phrase is a function of the semantics of the subphrases. Figure shows the parse tree for $3 + (4 \div 2)$ according to this grammar. The root of the parse tree is $Exp(5)$, an expression whose semantic interpretation is 5.

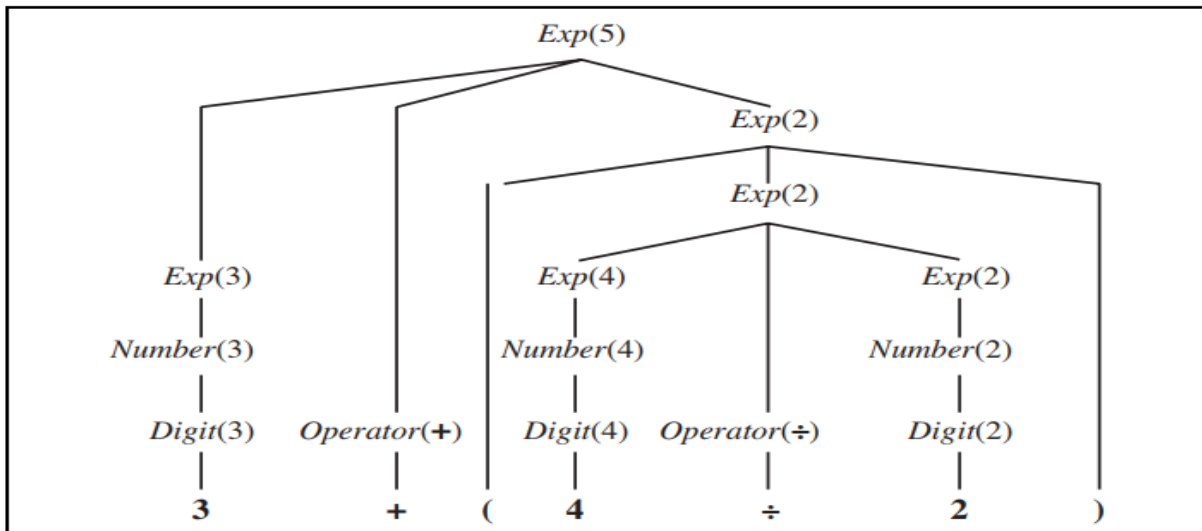


Figure 23.9 Parse tree with semantic interpretations for the string “ $3 + (4 \div 2)$ ”.

(v) Complications :

- The grammar of real English is endlessly complex. We will briefly mention some examples.

1. Time and tense:

Suppose we want to represent the difference between “John loves Mary” and “John loved Mary.”

- English uses verb tenses (past, present, and future) to indicate the relative time of an event. One good choice to represent the time of events is the event calculus notation
- In event calculus we have

John loves mary: $E1 \sqsubseteq \text{Loves}(\text{John}, \text{Mary}) \sqsubseteq \text{During}(\text{Now}, \text{Extent}(E1))$

John loved mary: $E2 \sqsubseteq \text{Loves}(\text{John}, \text{Mary}) \sqsubseteq \text{After}(\text{Now}, \text{Extent}(E2))$.

2. Quantification :

- Consider the sentence “Every agent feels a breeze.”

The sentence has only one syntactic parse under $E0$, but it is actually semantically ambiguous; the preferred meaning is “For every agent there exists a breeze that the agent feels,” but an acceptable alternative meaning is “There exists a breeze that every agent feels.”

3. Pragmatics:

- We have shown how an agent can perceive a string of words and use a grammar to derive a set of possible semantic interpretations.

4. Long-distance dependencies:

Questions introduce a new grammatical complexity. In “Who did the agent tell you to give the gold to?” the final word “to” should be parsed as [PP to], where the “ ” denotes a gap or **trace** where an NP is missing; the missing NP is licensed by the first word of the sentence, “who.”

5. Ambiguity :

In some cases, hearers are consciously aware of ambiguity in an utterance.

Types of ambiguities :

→ **Lexical ambiguity**, in which a word has more than one meaning.

→ **Syntactic ambiguity**, refers to a phrase that has multiple parses.

→ **Semantic ambiguity**, The syntactic ambiguity leads to a **semantic ambiguity**, because one parse means the other.

- **Disambiguation**, is the process of recovering the most probable intended meaning of an utterance.
- To do disambiguation properly, we need to combine four models:

→ **world model**

→ **mental model**

→ **language model**

→ **acoustic mode**

4.1.4. MACHINE TRANSLATION :

- Machine translation is the automatic translation of text from one natural language (the source) to another (the target).

It was one of the first application areas envisioned for computers (Weaver, 1949), but it is only in the past decade that the technology has seen widespread usage.

- Historically, there have been **three main applications** of machine translation.

→ **Rough translation**, as provided by free online services, gives the “gist” of a foreign sentence or document, but contains errors.

→ **Pre-edited translation** is used by companies to publish their documentation and sales materials in multiple languages.

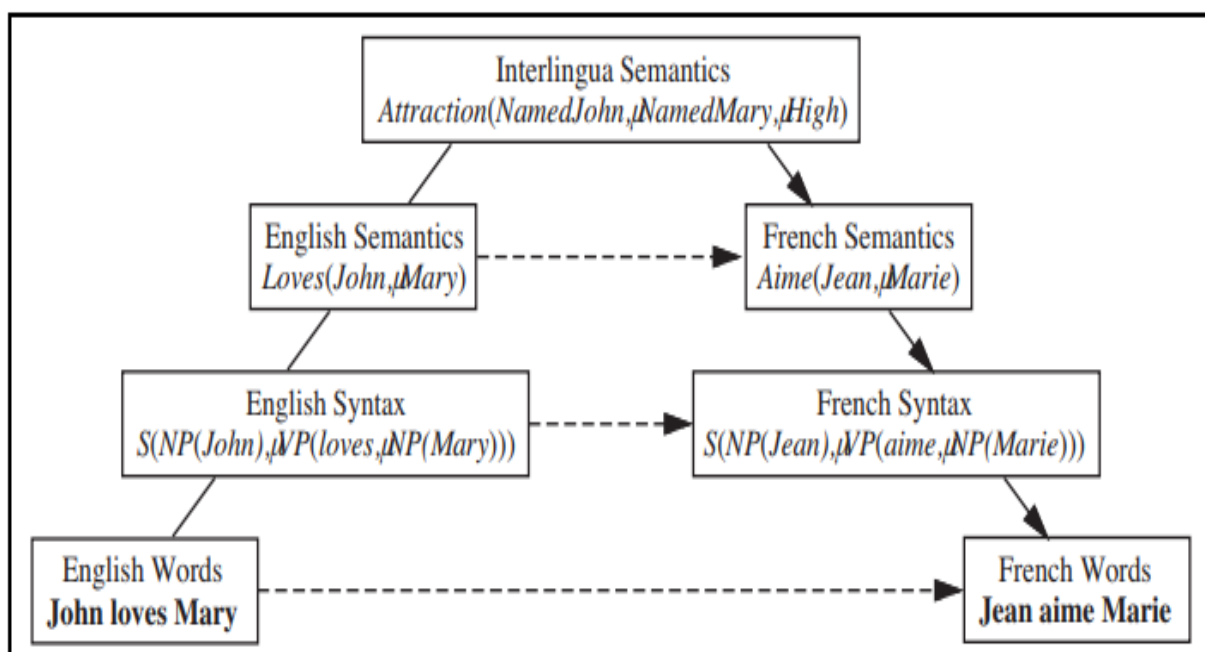
The original source text is written in a constrained language that is easier to translate automatically, and the results are usually edited by a human to correct any errors.

→ **Restricted-source translation** works fully automatically, but only on highly stereotypical language, such as a weather report.

- Translation is difficult because, in the fully general case, it requires in-depth understanding of the text. This is true even for very simple texts—even “texts” of one word.

(i) **Machine translation systems :**

- All translation systems must model the source and target languages, but systems vary in the type of models they use.
- Some systems attempt to analyze the source language text all the way into an interlingua knowledge representation and then generate sentences in the target language from that representation.
- This is difficult because it involves **three unsolved problems**:
 - creating a complete knowledge representation of everything;
 - parsing into that representation; and
 - generating sentences from that representation.
- Other systems are based on a **transfer model**.
- They keep a database of translation rules (or examples), and whenever the rule (or example) matches, they translate directly.
- Transfer can occur at the lexical, syntactic, or semantic level.
- For example, a strictly syntactic rule maps English [*Adjective Noun*] to French [*Noun Adjective*]. A mixed syntactic and lexical rule maps French [S1 “et puis” S2] to English [S1 “and then” S2].



(ii) **Statistical machine translation :**

- Now that we have seen how complex the translation task can be, it should come as no surprise that the most successful machine translation systems are built by training a probabilistic model using statistics gathered from a large corpus of text.
- This approach does not need a complex ontology of interlingua concepts, nor does it need handcrafted grammars of the source and target languages, nor a hand-labeled treebank.
- All it needs is data—sample translations from which a translation model can be learned. To translate a sentence in, say, English (e) into French (f), we find the string of words f^* that maximizes

$$f^* = \underset{f}{\operatorname{argmax}} P(f | e) = \underset{f}{\operatorname{argmax}} P(e | f) P(f) .$$

- Here the factor $P(f)$ is the target **language model** for French; it says how probable a given sentence is in French. $P(e|f)$ is the **translation mode**.
- All that remains is to learn the phrasal and distortion probabilities. We sketch the procedure;

➔ **Find parallel texts:**

First, gather a parallel bilingual corpus. For example, a **Hansard** is a record of parliamentary debate. Canada, Hong Kong, and other countries produce bilingual Hansards, the European Union publishes its official documents in 11 languages, and the United Nations publishes multilingual documents.

➔ **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.

➔ **Align sentences:** For each sentence in the English version, determine what sentence(s) it corresponds to in the French version.

Usually, the next sentence of English corresponds to the next sentence of French in a 1:1 match, but sometimes there is variation: one sentence in one language will be split into a 2:1 match, or the order of two sentences will be swapped, resulting in a 2:2 match.

➔ **Align phrases:** Within a sentence, phrases can be aligned by a process that is similar to that used for sentence alignment, but requiring iterative improvement.

➔ **Extract distortions:** Once we have an alignment of phrases we can define distortion probabilities. Simply count how often distortion occurs in the corpus for each distance $d = 0, \pm 1, \pm 2, \dots$, and apply smoothing.

➔ **Improve estimates with EM:** Use expectation–maximization to improve the estimates of $P(f | e)$ and $P(d)$ values.

We compute the best alignments with the current values of these parameters in the E step, then

update the estimates in the M step and iterate the process until convergence.

4.1.5. SPEECH RECOGNITION :

- **Speech recognition** is the task of identifying a sequence of words uttered by a speaker, given the acoustic signal.
- It has become one of the mainstream applications of AI—millions of people interact with speech recognition systems every day to navigate voice mail systems, search the Web from mobile phones, and other applications.
- Speech is an attractive option when hands-free operation is necessary, as when operating machinery. Speech recognition is difficult because the sounds made by a speaker are ambiguous and, well, noisy.
- Several issues that make speech problematic :
- First, **segmentation**: written words in English have spaces between them, but in fast speech there are no pauses in “wreck a nice” that would distinguish it as a multiword phrase as opposed to the single word “recognize.”
- Second, **coarticulation**: when speaking quickly the “s” sound at the end of “nice” merges with the “b” sound at the beginning of “beach,” yielding something that is close to a “sp.”
- Another problem that does not show up in this example is **homophones**—words like “to,” “too,” and “two” that sound the same but differ in meaning.
- As usual, the most likely sequence can be computed with the help of Bayes’ rule to be:

$$\operatorname{argmax}_{word_{1:t}} P(word_{1:t} | sound_{1:t}) = \operatorname{argmax}_{word_{1:t}} P(sound_{1:t} | word_{1:t}) P(word_{1:t}) .$$

- Here $P(sound_{1:t} | word_{1:t})$ is the **acoustic model**. It describes the sounds of words—that “ceiling” begins with a soft “c” and sounds the same as “sealing.”
- $P(word_{1:t})$ is known as the **language model**. It specifies the prior probability of each utterance—for example, that “ceiling fan” is about 500 times more likely as a word sequence than “sealing fan.”
- This approach was named the **noisy channel model** by Claude Shannon (1948).
- He described a situation in which an original message (the *words* in our example) is transmitted over a noisy channel (such as a telephone line) such that a corrupted message (the *sounds* in our example) are received at the other end.
- Once we define the acoustic and language models, we can solve for the most likely sequence of

words using the Viterbi algorithm .

- Most speech recognition systems use a language model that makes the Markov assumption—that the current state Word t depends only on a fixed number n of previous states—and represent Word t as a single random variable taking on a finite set of values, which makes it a Hidden Markov Model (HMM).

Thus, speech recognition becomes a simple application of the HMM methodology,

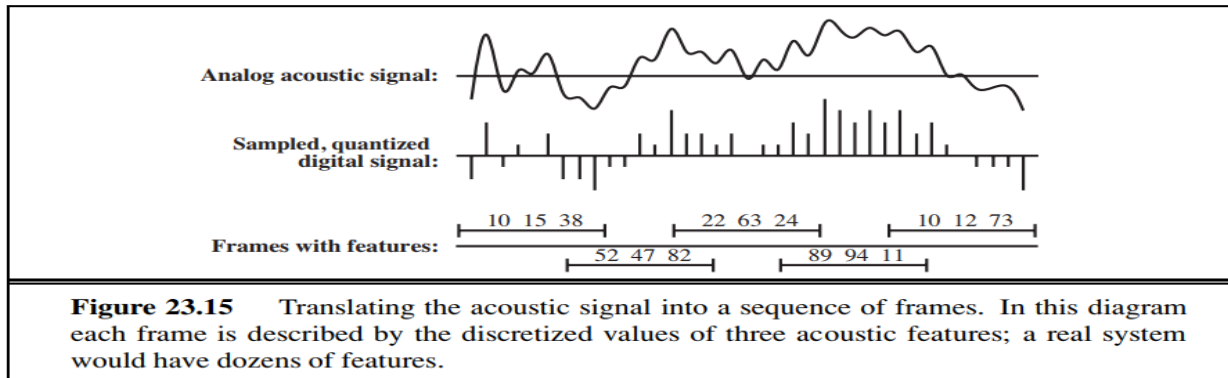
(i) **Acoustic model :**

- Sound waves are periodic changes in pressure that propagate through the air. When these waves strike the diaphragm of a microphone, the back-and-forth movement generates an electric current.
- An analog-to-digital converter measures the size of the current—which approximates the amplitude of the sound wave—at discrete intervals called the **sampling rate**.

Speech sounds, which are mostly in the range of 100 Hz (100 cycles per second) to 1000 Hz, are typically sampled at a rate of 8 kHz. (CDs and mp3 files are sampled at 44.1 kHz.)

- The precision of each measurement is determined by the **quantization factor**; speech recognizers typically keep 8 to 12 bits.
- That means that a low-end system, sampling at 8 kHz with 8-bit quantization, would require nearly half a megabyte per minute of speech.
- Since we only want to know what words were spoken, not exactly what they sounded like, we don't need to keep all that information.
- We only need to distinguish between different speech sounds. Linguists have identified about 100 speech sounds, or **phones**, that can be composed to form all the words in all known human languages.
- Roughly speaking, a phone is the sound that corresponds to a single vowel or consonant, but there are some complications: combinations of letters, such as “th” and “ng” produce single phones, and some letters produce different phones in different contexts all the phones that are used in English.
- A **phoneme** is the smallest unit of sound that has a distinct meaning to speakers of a particular language.
- For example, the “t” in “stick” sounds similar enough to the “t” in “tick” that speakers of English consider them the same phoneme.
- First, we observe that although the sound frequencies in speech may be several kHz, the *changes* in the content of the signal occur much less often, perhaps at no more than 100 Hz.

- Therefore, speech systems summarize the properties of the signal over time slices called **frames**.



(ii) Language model :

- For general-purpose speech recognition, the language model can be an n-gram model of text learned from a corpus of written sentences.
- However, spoken language has different characteristics than written language, so it is better to get a corpus of transcripts of spoken language.
- For task-specific speech recognition, the corpus should be task-specific: to build your airline reservation system, get transcripts of prior calls.

It also helps to have task-specific vocabulary, such as a list of all the airports and cities served, and all the flight numbers.

(iii) Building a speech recognizer :

- The quality of a speech recognition system depends on the quality of all of its components— the language model, the word-pronunciation models, the phone models, and the signal processing algorithms used to extract spectral features from the acoustic signal.
- The accuracy of a system depends on a number of factors. First, the quality of the signal matters: a high-quality directional microphone aimed at a stationary mouth in a padded room will do much better than a cheap microphone transmitting a signal over phone lines from a car in traffic with the radio playing.

PERCEPTION:

Perception provides agents with information about the world they inhabit by interpreting the response of **sensors**.

A sensor measures some aspect of the environment in a form that can be used as input by an agent program. The sensor could be as simple as a switch, which gives one bit telling whether it is on or off, or as complex as the eye. A variety of sensory modalities are available to artificial agents.

Some robots do **active sensing**, meaning they send out a signal, such as radar or ultrasound, and sense the reflection of this signal off of the environment.

The problem for a vision-capable agent then is: Which aspects of the rich visual stimulus should be considered to help the agent make good action choices, and which aspects should be ignored? Vision—and all perception—serves to further the agent’s goals, not as an end to itself.

In the **recognition** approach an agent draws distinctions among the objects it encounters based on visual and other information. Recognition could mean labelling each image with a yes or no as to whether it contains food that we should forage, or contains Grandma’s face.

IMAGE FORMATION

Imaging distorts the appearance of objects. For example, a picture taken looking down a long straight set of railway tracks will suggest that the rails converge and meet. As another example, if you hold your hand in front of your eye, you can block out the moon, which is not smaller than your hand. As you move your hand back and forth or tilt it, your hand will seem to shrink and grow in the image, but it is not doing so in reality. Models of these effects are essential for both recognition and reconstruction.

Images without lenses: The pinhole camera

Image sensors gather light scattered from objects in a **scene** and create a two-dimensional **image**. In the eye, the image is formed on the retina, which consists of two types of cells: about 100 million rods, which are sensitive to light at a wide range of wavelengths, and 5 million cones. Cones, which are essential for colour vision, are of three main types, each of which is sensitive to a different set of wavelengths.

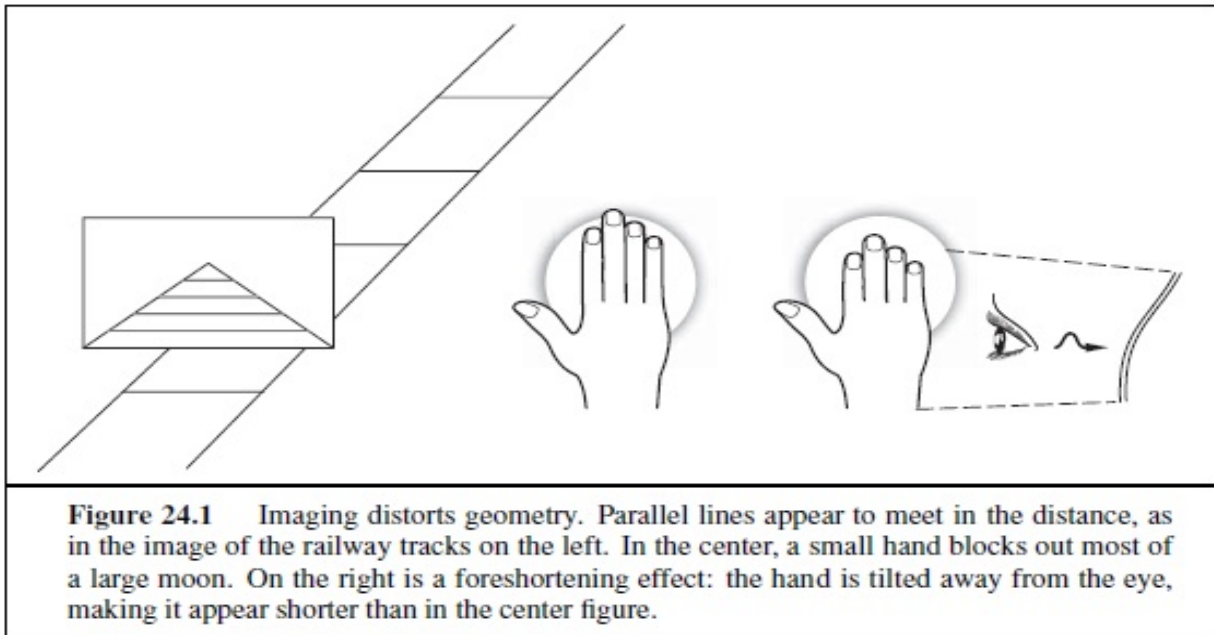
In cameras, the image is formed on an image plane, which can be a piece of film coated with silver halides or a rectangular grid of a few million photosensitive **pixels**, each a complementary metal-oxide semiconductor (CMOS) or charge-coupled device (CCD).

Each photon arriving at the sensor produces an effect, whose strength depends on the wavelength of the photon. The output of the sensor is the sum of all effects due to photons observed in some time

window, meaning that image sensors report a weighted average of the intensity of light arriving at the sensor.

To see a focused image, we must ensure that all the photons from approximately the same spot in the scene arrive at approximately the same point in the image plane.

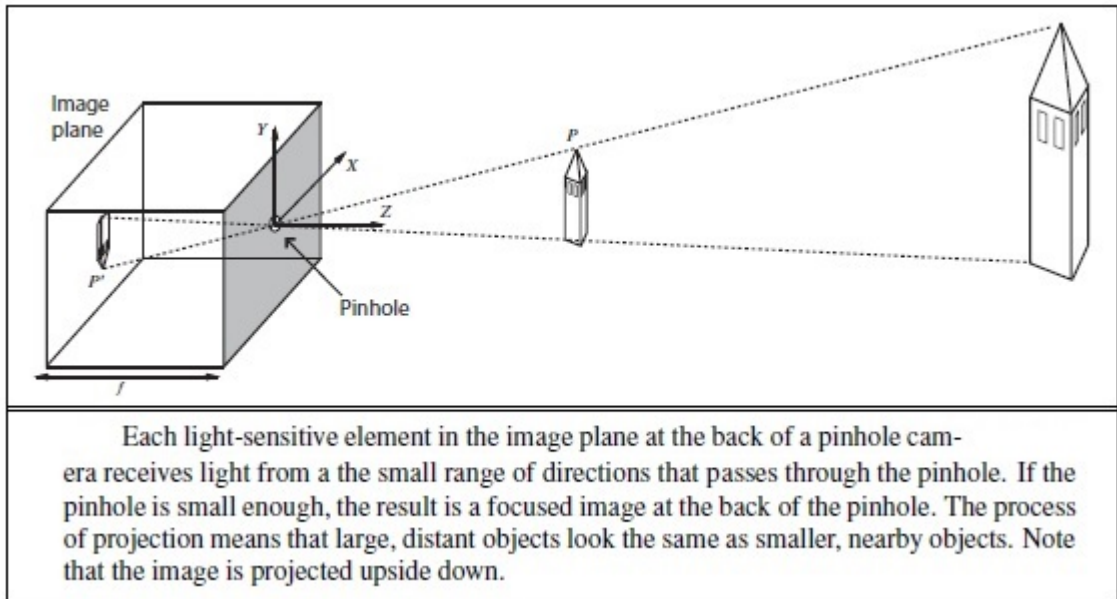
The simplest way to form a focused image is to view stationary objects with a **pinhole camera**, which consists of a pinhole opening, O, at the front of a box, and an image plane at the back of the box. Photons from the scene must pass through the pinhole, so if it is small enough then nearby photons in the scene will be nearby in the image plane, and the image will be in focus.



The geometry of scene and image is easiest to understand with the pinhole camera. We use a three-dimensional coordinate system with the origin at the pinhole, and consider a point P in the scene, with coordinates (X, Y, Z). P gets projected to the point P' in the image plane with coordinates (x, y, z). If f is the distance from the pinhole to the image plane, then by similar triangles, we can derive the following equations.

$$\frac{-x}{f} = \frac{X}{Z}, \quad \frac{-y}{f} = \frac{Y}{Z} \quad \Rightarrow \quad x = \frac{-fX}{Z}, \quad y = \frac{-fY}{Z}.$$

These equations define an image-formation process known as **perspective projection**.



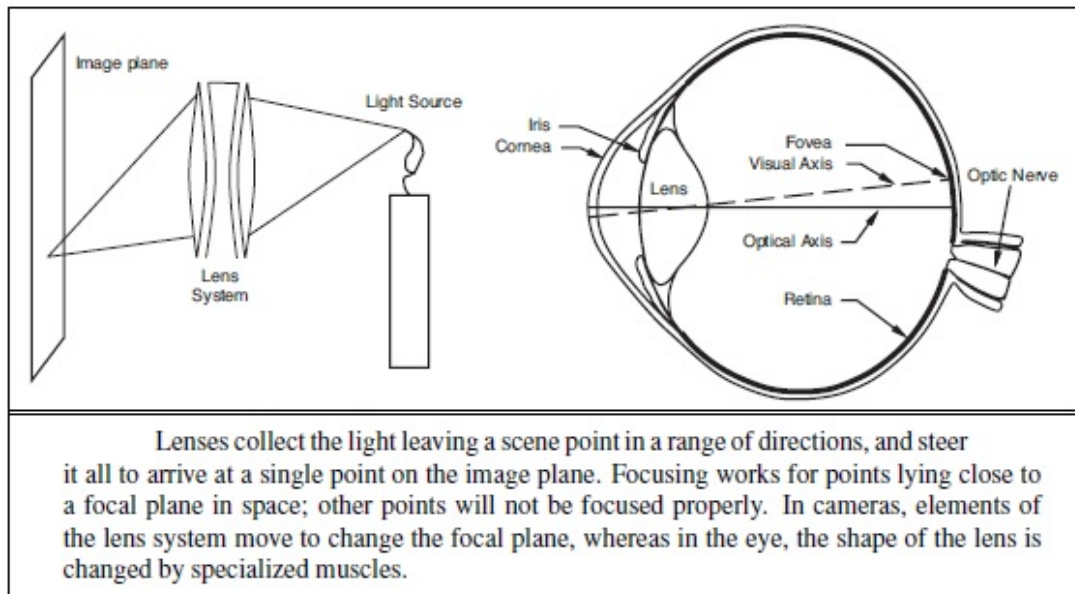
Under perspective projection, distant objects look small. This is what allows you to cover the moon with your hand (Figure 24.1). An important result of this effect is that parallel lines converge to a point on the horizon. (Think of railway tracks, Figure 24.1.) A line in the scene in the direction (U, V, W) and passing through the point (X_0, Y_0, Z_0) can be described as the set of points $(X_0 + \lambda U, Y_0 + \lambda V, Z_0 + \lambda W)$, with λ varying between $-\infty$ and $+\infty$. Different choices of (X_0, Y_0, Z_0) yield different lines parallel to one another. The projection of a point P_λ from this line onto the image plane is given by

$$\left(f \frac{X_0 + \lambda U}{Z_0 + \lambda W}, f \frac{Y_0 + \lambda V}{Z_0 + \lambda W} \right)$$

As $\lambda \rightarrow \infty$ or $\lambda \rightarrow -\infty$, this becomes $p_\infty = (fU/W, fV/W)$ if $W \neq 0$. This means that two parallel lines leaving different points in space will converge in the image—for large λ , the image points are nearly the same, whatever the value of (X_0, Y_0, Z_0) (again, think railway tracks,). We call p_∞ the **vanishing point** associated with the family of straight lines with direction (U, V, W) . Lines with the same direction share the same vanishing point.

Lens systems

The drawback of the pinhole camera is that we need a small pinhole to keep the image in focus. But the smaller the pinhole, the fewer photons get through, meaning the image will be dark. We can gather more photons by keeping the pinhole open longer, but then we will get **motion blur**—objects in the scene that move will appear blurred because they send photons to multiple locations on the image plane. If we can't keep the pinhole open longer, we can try to make it bigger. More light will enter, but light from a small patch of object in the scene will now be spread over a patch on the image plane, causing a blurred image.



Vertebrate eyes and modern cameras use a **lens** system to gather sufficient light while keeping the image in focus. A large opening is covered with a lens that focuses light from nearby object locations down to nearby locations in the image plane. However, lens systems have a limited **depth of field**: they can focus light only from points that lie within a range of depths (centered around a **focal plane**). Objects outside this range will be out of focus in the image. To move the focal plane, the lens in the eye can change shape in a camera, the lenses move back and forth.

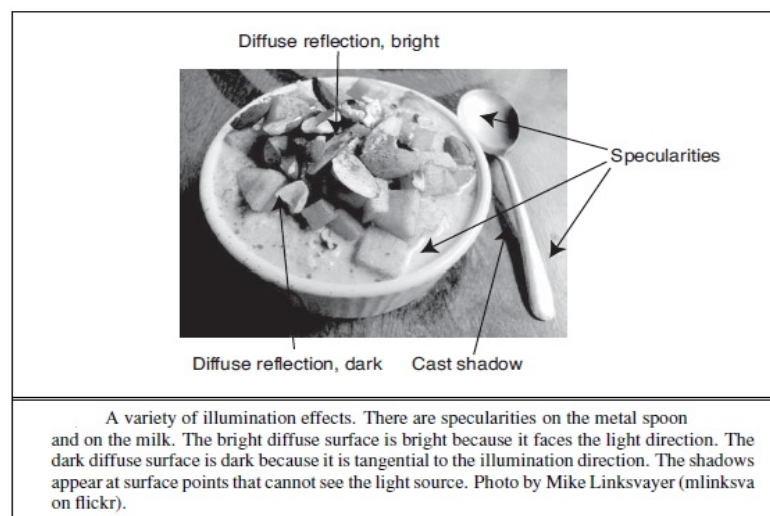
Scaled orthographic projection

Perspective effects aren't always pronounced. For example, spots on a distant leopard may look small because the leopard is far away, but two spots that are next to each other will have about the same size. This is because the difference in distance to the spots is small compared to the distance to them, and so we can simplify the projection model. The appropriate model is **scaled orthographic projection**. The idea is as follows: If the depth Z of points on the object varies within some range $Z_0 \pm \Delta Z$, with $\Delta Z \ll Z_0$, then the perspective scaling factor f/Z can be approximated by a constant $s = f/Z_0$. The equations for projection from the scene coordinates (X, Y, Z) to the image plane become $x = sX$ and $y = sY$. Scaled orthographic projection is an approximation that is valid only for those parts of the scene with not much internal depth variation. For example, scaled orthographic projection can be a good model for the features on the front of a distant building.

Light and shading

The brightness of a pixel in the image is a function of the brightness of the surface patch in the scene that projects to the pixel. We will assume a linear model (current cameras have nonlinearities at the extremes of light and dark, but are linear in the middle). Image brightness is a strong, if ambiguous, cue to the shape of an object, and from there to its identity. People are usually able to distinguish the three main causes of varying brightness and reverse-engineer the object's properties.

The first cause is **overall intensity** of the light. Even though a white object in shadow may be less bright than a black object in direct sunlight, the eye can distinguish relative brightness well, and perceive the white object as white. Second, different points in the scene may **reflect** more or less of the light. Third, surface patches facing the light are brighter than surface patches tilted away from the light, an effect known as **shading**. Most surfaces reflect light by a process of **diffuse reflection**. Diffuse reflection scatters light evenly across the directions leaving a surface, so the brightness of a diffuse surface doesn't depend on the viewing direction. The behaviour of a perfect mirror is known as specular **reflection**. Some surfaces—such as brushed metal, plastic, or a wet floor—display small patches where specular reflection has occurred, called **secularities**. These are easy to identify, because they are small and bright. For almost all purposes, it is enough to model all surfaces as being diffuse with secularities.



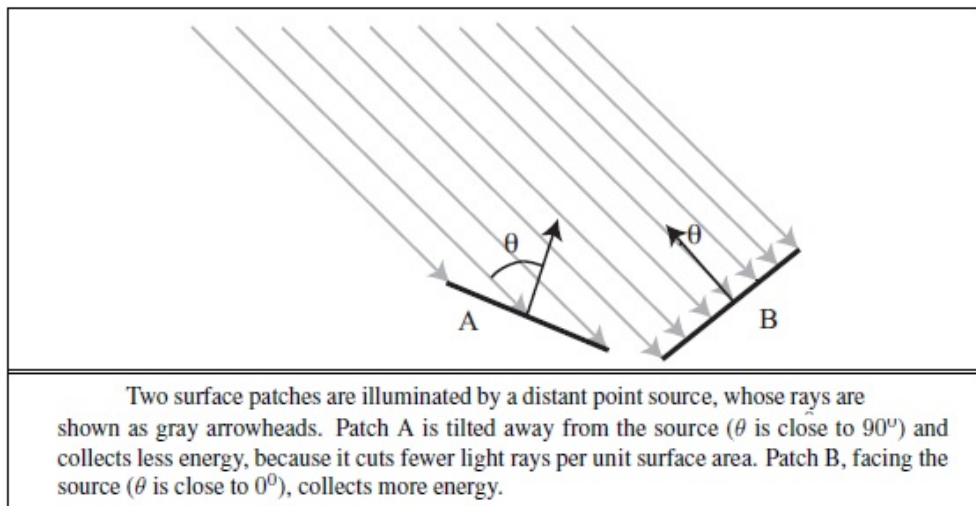
The main source of illumination outside is the sun, whose rays all travel parallel to one another. We model this behaviour as a **distant point light source**. This is the most important model of lighting, and is quite effective for indoor scenes as well as outdoor scenes. The amount of light collected by a surface patch in this model depends on the angle θ between the illumination direction and the normal to the surface.

A diffuse surface patch illuminated by a distant point light source will reflect some fraction of the light it collects; this fraction is called the **diffuse albedo**. White paper and snow have a high albedo, about 0.90, whereas flat black velvet and charcoal have a low albedo of about 0.05 (which means that 95% of

the incoming light is absorbed within the fibres of the velvet or the pores of the charcoal). **Lambert's cosine law** states that the brightness of a diffuse patch is given by

$$I = \rho I_0 \cos \theta$$

where ρ is the diffuse albedo, I_0 is the intensity of the light source and θ is the angle between the light source direction and the surface. Lambert's law predicts bright image pixels come from surface patches that face the light directly and dark pixels come from patches that see the light only tangentially, so that the shading on a surface provides some shape information. If the surface is not reached by the light source, then it is in **shadow**. Shadows are very seldom a uniform black, because the shadowed surface receives some light from other sources. Outdoors, the most important such source is the sky, which is quite bright. Indoors, light reflected from other surfaces illuminates shadowed patches. These **interreflections** can have a significant effect on the brightness of other surfaces, too. These effects are sometimes modelled by adding a constant **ambient illumination** term to the predicted intensity.



Colour

Fruit is a bribe that a tree offers to animals to carry its seeds around. Trees have evolved to have fruit that turns red or yellow when ripe, and animals have evolved to detect these colour changes. Light arriving at the eye has different amounts of energy at different wavelengths; this can be represented by a spectral energy density function. Human eyes respond to light in the 380–750nm wavelength region, with three different types of colour receptor cells, which have peak receptiveness at 420nm (blue), 540nm (green), and 570nm (red). The human eye can capture only a small fraction of the full spectral energy density function—but it is enough to tell when the fruit is ripe.

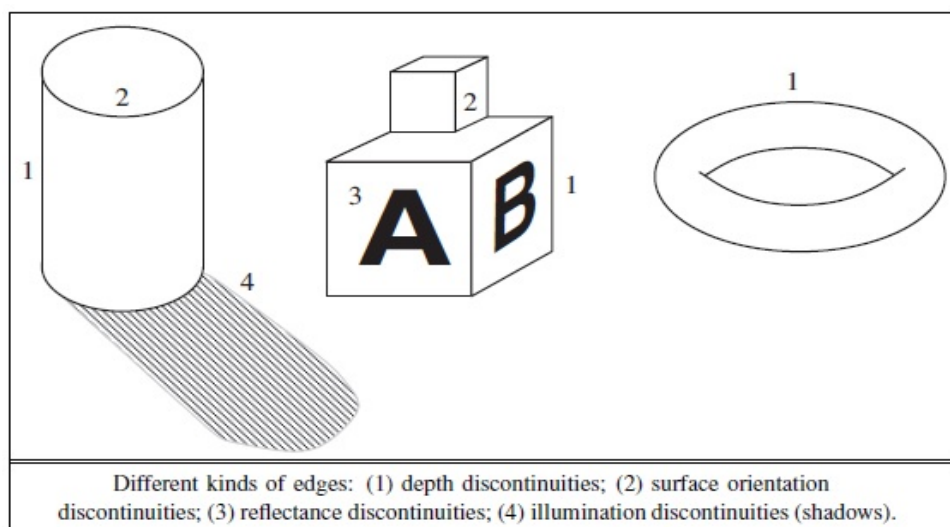
The **principle of trichromacy** states that for any spectral energy density, no matter how complicated, it is possible to construct another spectral energy density consisting of a mixture of just three colours—usually red, green, and blue—such that a human can't tell the difference between the

two. That means that our TVs and computer displays can get by with just the three red/green/blue (or R/G/B) colour elements. It makes our computer vision algorithms easier, too. Each surface can be modelled with three different albedos for R/G/B. Similarly, each light source can be modelled with three R/G/B intensities. We then apply Lambert's cosine law to each to get three R/G/B pixel values. This model predicts, correctly, that the same surface will produce different coloured image patches under different-coloured lights. In fact, human observers are quite good at ignoring the effects of different coloured lights and are able to estimate the colour of the surface under white light, an effect known as **colour constancy**. Quite accurate colour constancy algorithms are now available; simple versions show up in the "auto white balance" function of your camera. Note that if we wanted to build a camera for mantis shrimp, we would need 12 different pixel colours, corresponding to the 12 types of colour receptors of the crustacean.

EARLY IMAGE-PROCESSING OPERATIONS

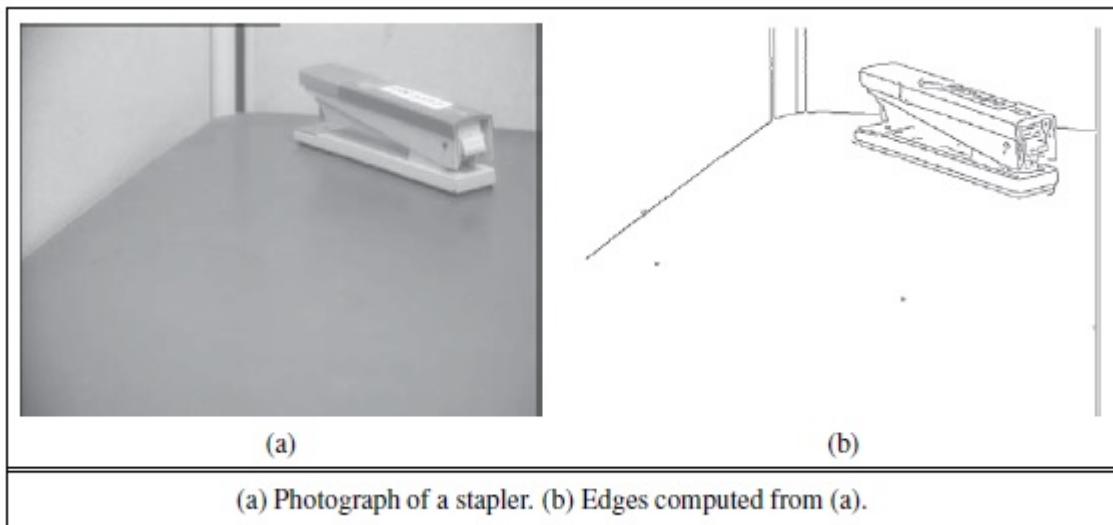
We have seen how light reflects off objects in the scene to form an image consisting of, say, five million 3-byte pixels. With all sensors there will be noise in the image, and in any case there is a lot of data to deal with. So how do we get started on analysing this data?

we will study three useful image-processing operations: edge detection, texture analysis, and computation of optical flow. These are called "early" or "low-level" operations because they are the first in a pipeline of operations. Early vision operations are characterized by their local nature (they can be carried out in one part of the image without regard for anything more than a few pixels away) and by their lack of knowledge: we can perform these operations without consideration of the objects that might be present in the scene. This makes the low-level operations good candidates for implementation in parallel hardware—either in a graphics processor unit (GPU) or an eye. We will then look at one mid-level operation: segmenting the image into regions.

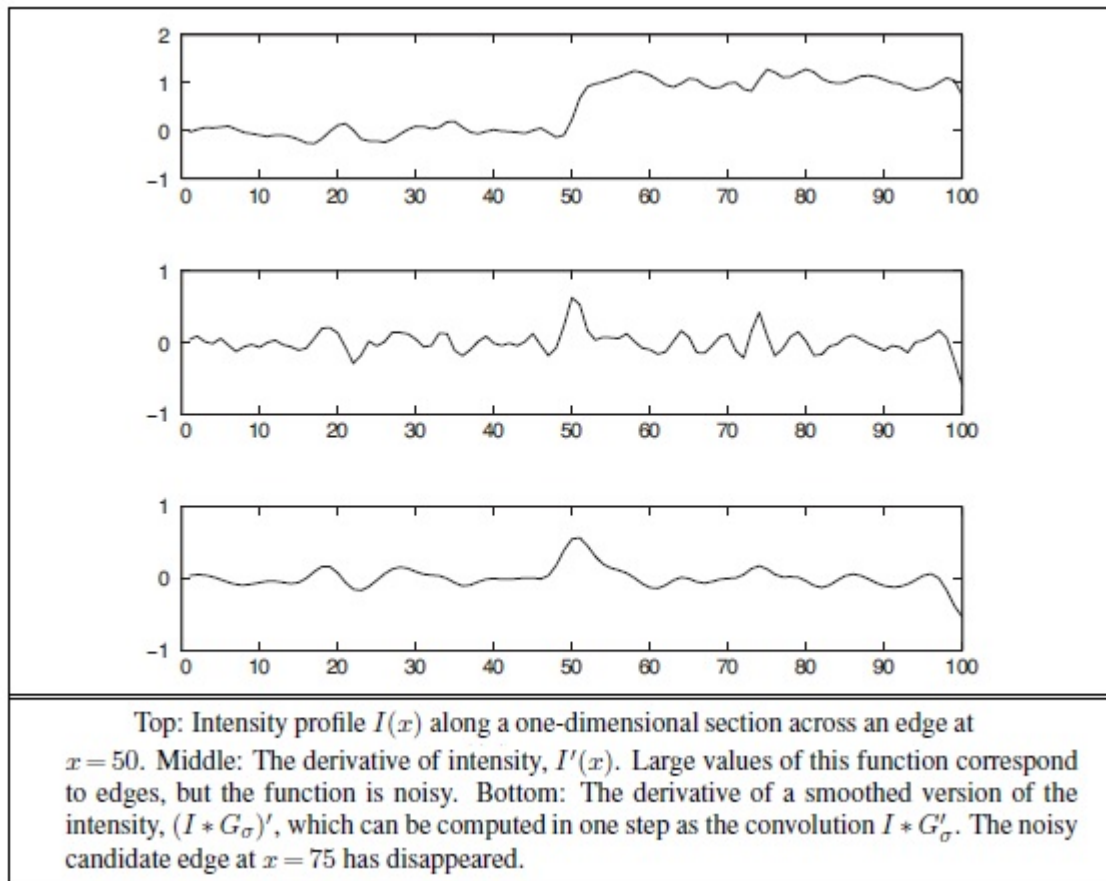


Edge detection

Edges are straight lines or curves EDGE in the image plane across which there is a “significant” change in image brightness. The goal of edge detection is to abstract away from the messy, multimega byte image and toward a more compact, abstract representation, the motivation is that edge contours in the image correspond to important scene contours. In the figure we have three examples of depth discontinuity, labelled 1; two surface-normal discontinuities, labelled 2; a reflectance discontinuity, labelled 3; and an illumination discontinuity (shadow), labelled 4. Edge detection is concerned only with the image, and thus does not distinguish between these different types of scene discontinuities; later processing will.



(a) shows an image of a scene containing a stapler resting on a desk, and
(b) shows the output of an edge-detection algorithm on this image. As you can see, there is a difference between the output and an ideal line drawing. There are gaps where no edge appears, and there are “noise” edges that do not correspond to anything of significance in the scene. Later stages of processing will have to correct for these errors.



How do we detect edges in an image? Consider the profile of image brightness along a one-dimensional cross-section perpendicular to an edge—for example, the one between the left edge of the desk and the wall. Edges correspond to locations in images where the brightness undergoes a sharp change, so a naive idea would be to differentiate the image and look for places where the magnitude of the derivative $I'(x)$ is large.

One good answer is a weighted average that weights the nearest pixels the most, then gradually decreases the weight for more distant pixels. The **Gaussian filter** does just that. (Users of Photoshop recognize this as the *Gaussian blur* operation.) Recall that the Gaussian function with standard deviation σ and mean 0 is

$$N_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} \quad \text{in one dimension, or}$$

$$N_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \quad \text{in two dimensions.}$$

The application of the Gaussian filter replaces the intensity $I(x_0, y_0)$ with the sum, overall (x, y) pixels, of $I(x, y)N_\sigma(d)$, where d is the distance from (x_0, y_0) to (x, y) . This kind of weighted sum is so common that there is a special name and notation for it. We say that the function h is the **convolution** of two functions f and g (denoted $f \otimes g$) if we have

$$h(x) = (f * g)(x) = \sum_{u=-\infty}^{+\infty} f(u) g(x - u) \quad \text{in one dimension, or}$$

$$h(x, y) = (f * g)(x, y) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u, v) g(x - u, y - v) \quad \text{in two.}$$

There is a natural generalization of this algorithm from one-dimensional cross sections to general two-dimensional images. In two dimensions edges may be at any angle θ . Considering the image brightness as a scalar function of the variables x, y , its gradient is a vector

$$\nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$$

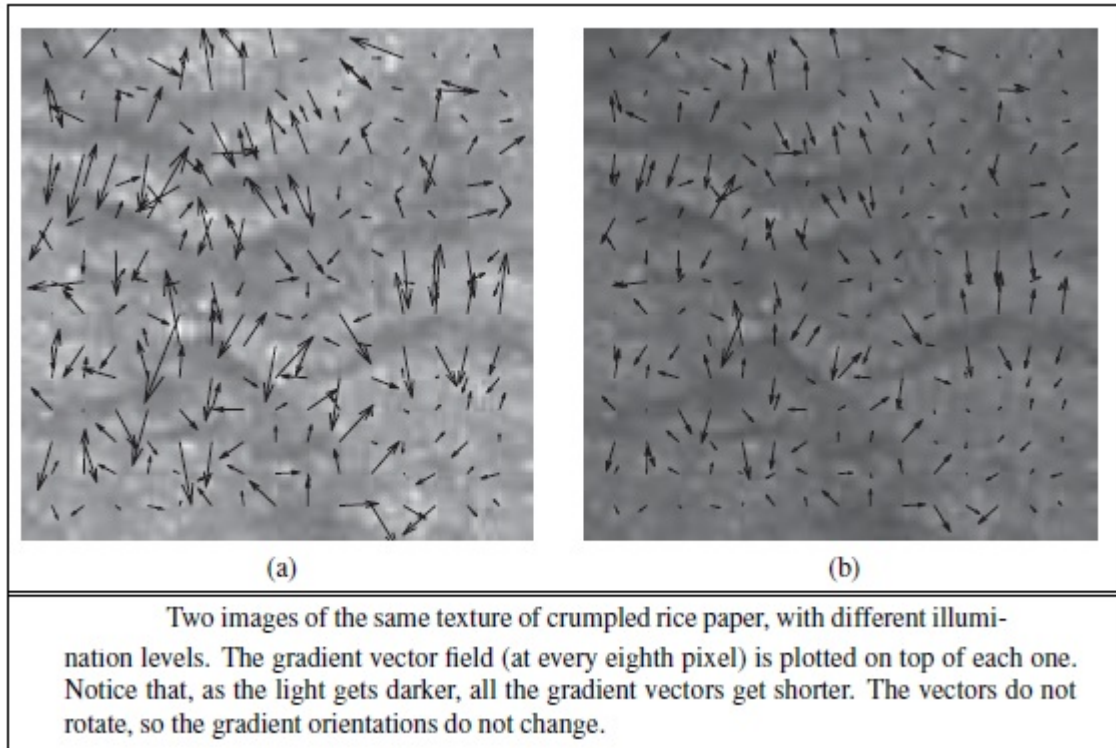
This gives us a $\theta = \theta(x, y)$ at every pixel, which defines the edge **orientation** at that pixel.

$$\frac{\nabla I}{\|\nabla I\|} = \begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix}$$

Once we have marked edge pixels by this algorithm, the next stage is to link those pixels that belong to the same edge curves. This can be done by assuming that any two neighbouring edge pixels with consistent orientations must belong to the same edge curve.

Texture

In everyday language, **texture** is the visual TEXTURE feel of a surface—what you see evokes what the surface might feel like if you touched it (“texture” has the same root as “textile”). In computational vision, texture refers to a spatially repeating pattern on a surface that can be sensed visually. Examples include the pattern of windows on a building, stitches on a sweater, spots on a leopard, blades of grass on a lawn, pebbles on a beach, and people in a stadium. Sometimes the arrangement is quite periodic, as in the stitches on a sweater; in other cases, such as pebbles on a beach, the regularity is only statistical.



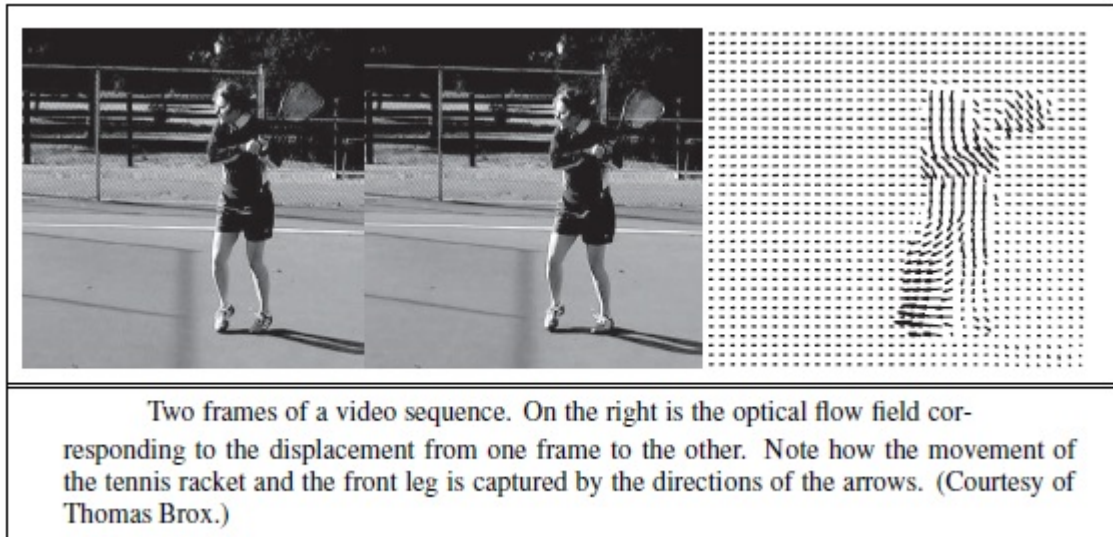
In images of textured objects, edge detection does not work as well as it does for smooth objects. This is because the most important edges can be lost among the texture elements. Quite literally, we may miss the tiger for the stripes. The solution is to look for differences in texture properties, just the way we look for differences in brightness. A patch on a tiger and a patch on the grassy background will have very different orientation histograms, allowing us to find the boundary curve between them.

Optical flow

Next, let us consider what happens when we have a video sequence, instead of just a single static image. When an object in the video is moving, or when the camera is moving relative to an object, the resulting apparent motion in the image is called **optical flow**. Optical flow describes the direction and speed of motion of features in the image—the optical flow of a video of a race car would be measured in pixels per second, not miles per hour. The optical flow encodes useful information about scene structure. For example, in a video of scenery taken from a moving train, distant objects have slower apparent motion than close objects; thus, the rate of apparent motion can tell us something about distance. Optical flow also enables us to recognize actions.

This block of pixels is to be compared with pixel blocks cantered at various candidate pixels at $(x_0 + D_x, y_0 + D_y)$ at time $t_0 + D_t$. One possible measure of similarity is the **sum of squared differences** (SSD):

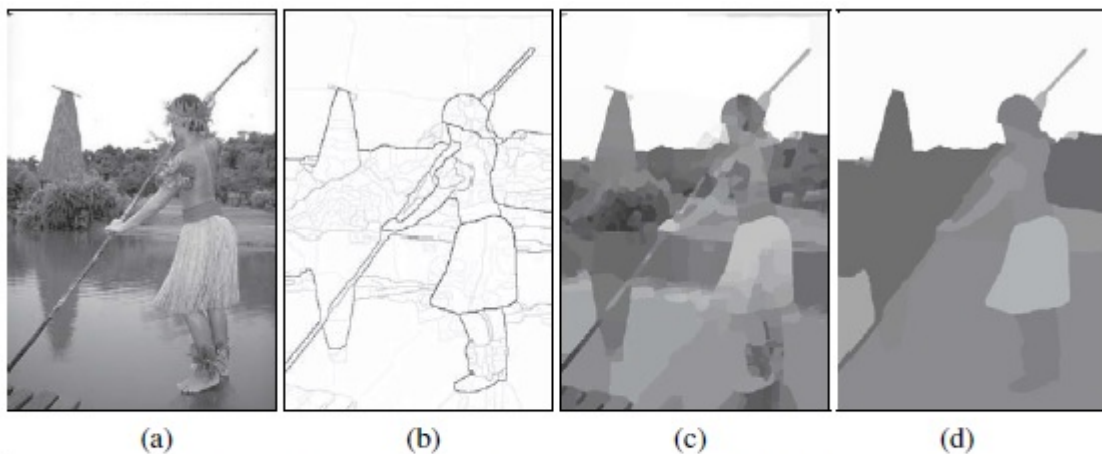
$$SSD(D_x, D_y) = \sum_{(x,y)} (I(x, y, t) - I(x + D_x, y + D_y, t + D_t))^2.$$



Segmentation of images

Segmentation is the process of breaking an image into **regions** of similar pixels. Each image pixel can be associated with certain visual properties, such as brightness, colour, and texture. Within an object, or a single part of an object, these attributes vary relatively little, whereas across an inter-object boundary there is typically a large change in one or more of these attributes.

Boundaries detected by this technique turn out to be significantly better than those found using the simple edge-detection technique described previously. But still there are two limitations. (1) The boundary pixels formed by thresholding $P_b(x, y, \theta)$ are not guaranteed to form closed curves, so this approach doesn't deliver regions, and (2) the decision making exploits only local context and does not use global consistency constraints.



(a) Original image. (b) Boundary contours, where the higher the P_b value, the darker the contour. (c) Segmentation into regions, corresponding to a fine partition of the image. Regions are rendered in their mean colors. (d) Segmentation into regions, corresponding to a coarser partition of the image, resulting in fewer regions. (Courtesy of Pablo Arbelaez, Michael Maire, Charles Fowlkes, and Jitendra Malik)

Segmentation based purely on low-level, local attributes such as brightness and colour cannot be expected to deliver the final correct boundaries of all the objects in the scene. To reliably find object boundaries we need high-level knowledge of the likely kinds of objects in the scene. Representing this knowledge is a topic of active research. A popular strategy is to produce an over-segmentation of an image, containing hundreds of homogeneous regions known as **super pixels**. From there, knowledge-based algorithms can take over; they will find it easier to deal with hundreds of super pixels rather than millions of raw pixels. How to exploit high-level knowledge of objects is the subject of the next section.

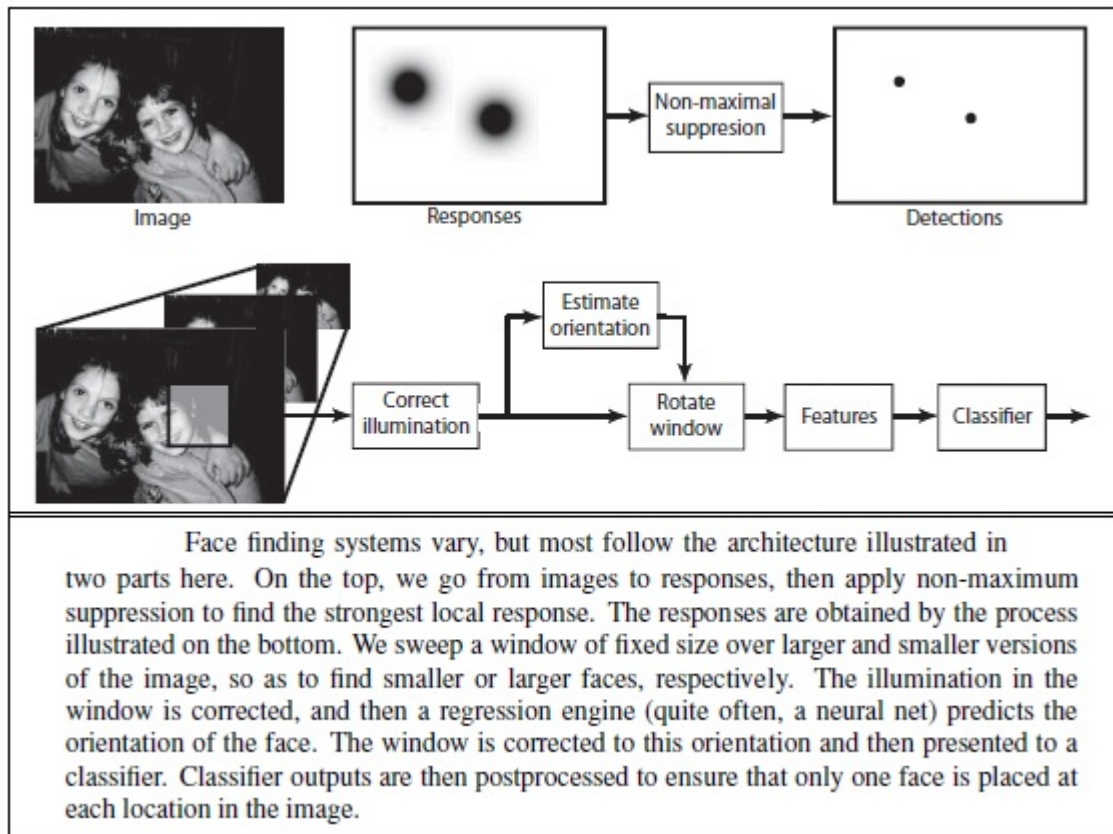
OBJECT RECOGNITION BY APPEARANCE

Appearance is shorthand for what an object tends to look like. Some object categories—for example, baseballs—vary rather little in appearance; all of the objects in the category look about the same under most circumstances. In this case, we can compute a set of features describing each class of images likely to contain the object, then test it with a classifier.

Testing each class of images with a learned classifier is an important general recipe. It works extremely well for faces looking directly at the camera, because at low resolution and under reasonable lighting, all such faces look quite similar. The face is round, and quite bright compared to the eye sockets; these are dark, because they are sunken, and the mouth is a dark slash, as are the eyebrows. Major changes of illumination can cause some variations in this pattern, but the range of variation is quite manageable. That makes it possible to detect face positions in an image that contains faces. Once a computational challenge, this feature is now commonplace in even inexpensive digital cameras.

For the moment, we will consider only faces where the nose is oriented vertically; we will deal with rotated faces below. We sweep a round window of fixed size over the image, compute features for it, and present the features to a classifier. This strategy is sometimes called the **sliding window**. Features need to be robust to shadows and to changes in brightness caused by illumination changes. One strategy is to build features out of gradient orientations. Another is to estimate and correct the illumination in each image window. To find faces of different sizes, repeat the sweep over larger or smaller versions of the image. Finally, we postprocess the responses across scales and locations to produce the final set of detections.

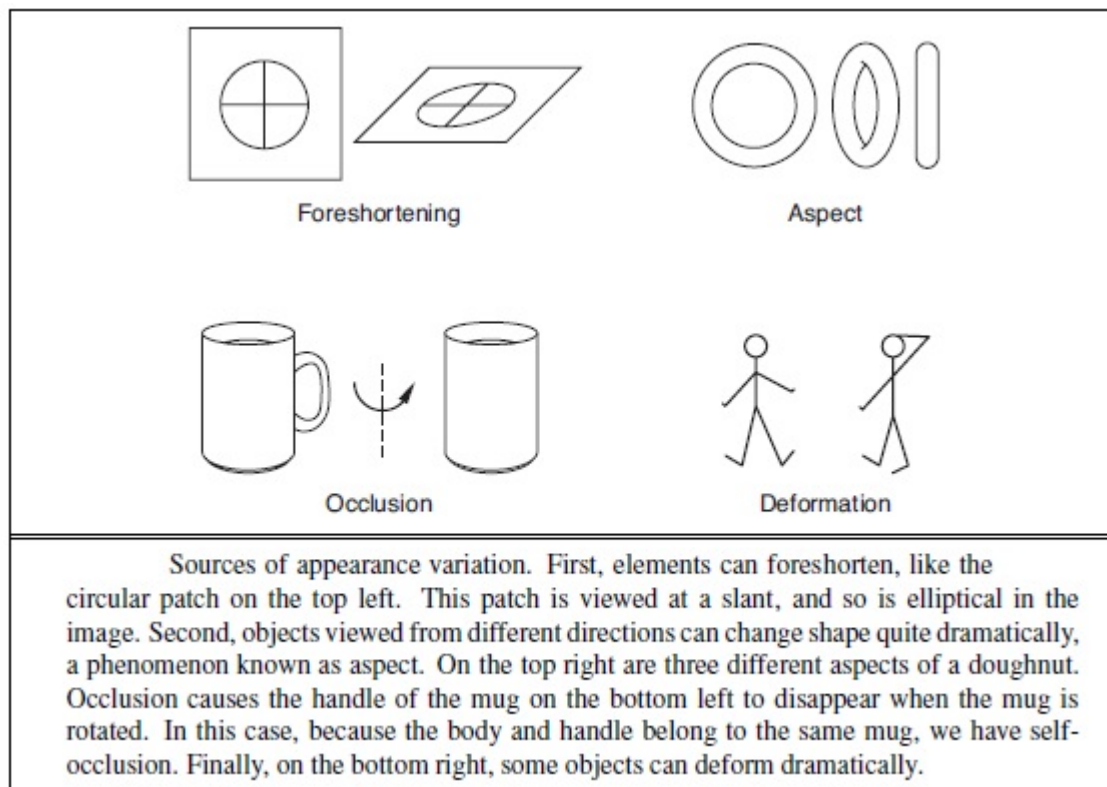
Training data is quite easily obtained. There are several data sets of marked-up face images, and rotated face windows are easy to build (just rotate a window from a training data set). One trick that is widely used is to take each example window, then produce new examples by changing the orientation of the window, the center of the window, or the scale very slightly. This is an easy way of getting a bigger data set that reflects real images fairly well; the trick usually improves performance significantly.



Complex appearance and pattern elements

Many objects produce much more complex patterns than faces do. This is because several effects can move features around in an image of the object.

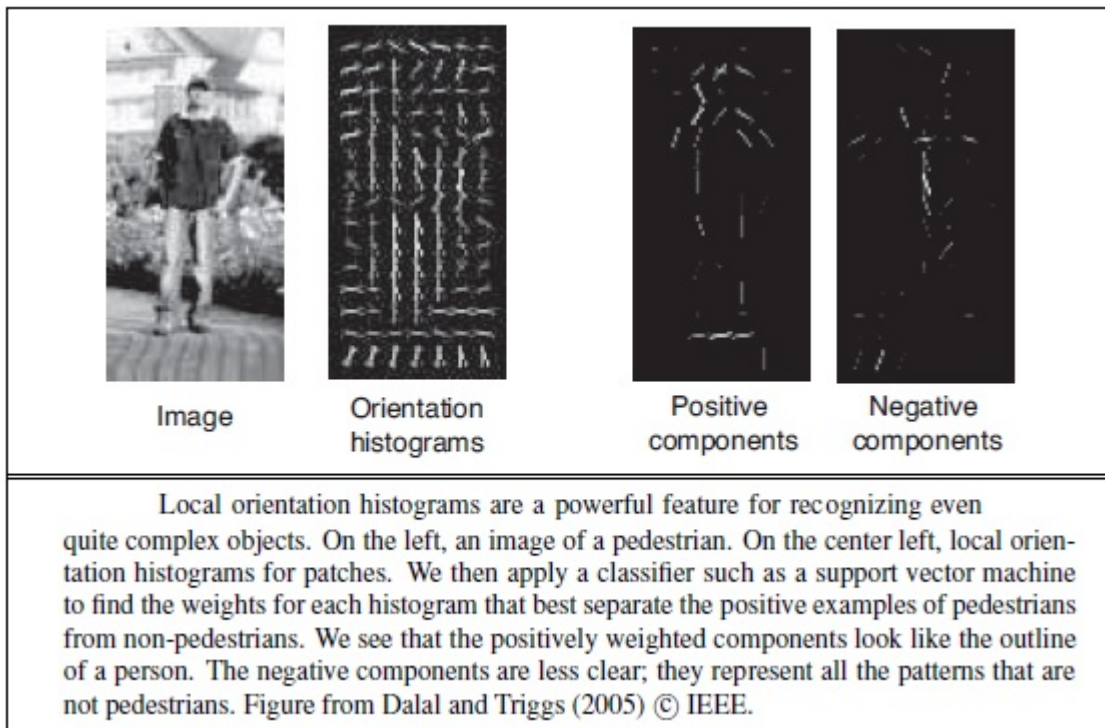
- **Foreshortening**, which causes a pattern viewed at a slant to be significantly distorted.
- **Aspect**, which causes objects to look different when seen from different directions. Even as simple an object as a doughnut has several aspects; seen from the side, it looks like a flattened oval, but from above it is an annulus.
- **Occlusion**, where some parts are hidden from some viewing directions. Objects can occlude one another, or parts of an object can occlude other parts, an effect known as self-occlusion.
- **Deformation**, where internal degrees of freedom of the object change its appearance. For example, people can move their arms and legs around, generating a very wide range of different body configurations.



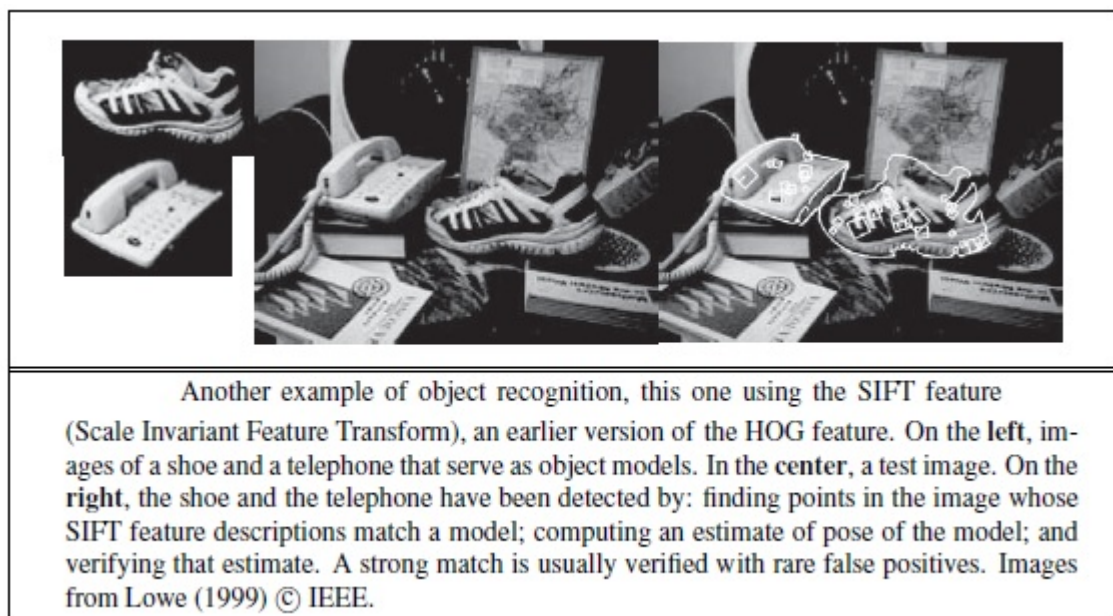
The most obvious approach is to represent the image window with a histogram of the pattern elements that appear there. This approach does not work particularly well, because too many patterns get confused with one another. For example, if the pattern elements are color pixels, the French, UK, and Netherlands flags will get confused because they have approximately the same color histograms, though the colors are arranged in very different ways. Quite simple modifications of histograms yield very useful features. The trick is to preserve some spatial detail in the representation; for example, headlights tend to be at the front of a car and wheels tend to be at the bottom. Histogram-based features have been successful in a wide variety of recognition applications; we will survey pedestrian detection.

Pedestrian detection with HOG features

The World Bank estimates that each year car accidents kill about 1.2 million people, of whom about two thirds are pedestrians. This means that detecting pedestrians is an important application problem, because cars that can automatically detect and avoid pedestrians might save many lives. Pedestrians wear many different kinds of clothing and appear in many different configurations, but, at relatively low resolution, pedestrians can have a fairly characteristic appearance. The most usual cases are lateral or frontal views of a walk. In these cases, we see either a “lollipop” shape — the torso is wider than the legs, which are together in the stance phase of the walk — or a “scissor” shape — where the legs are swinging in the walk. We expect to see some evidence of arms and legs, and the curve around the shoulders and head also tends to be visible and quite distinctive. This means that, with a careful feature construction, we can build a useful moving-window pedestrian detector.



One further trick is required to make a good feature. Because orientation features are not affected by illumination brightness, we cannot treat high-contrast edges specially. This means that the distinctive curves on the boundary of a pedestrian are treated in the same way as fine texture detail in clothing or in the background, and so the signal may be submerged in noise. We can recover contrast information by counting gradient orientations with weights that reflect how significant a gradient is compared to other gradients in the same cell.



orientation at \mathbf{x} for this cell. A natural choice of weight is

$$w_{x,c} = \frac{\|\nabla I_x\|}{\sum_{u \in \mathcal{C}} \|\nabla I_u\|} .$$

This compares the gradient magnitude to others in the cell, so gradients that are large compared to their neighbours get a large weight. The resulting feature is usually called a **HOG feature**

This feature construction is the main way in which pedestrian detection differs from face detection.

Otherwise, building a pedestrian detector is very like building a face detector. The detector sweeps a window across the image, computes features for that window, then presents it to a classifier. Non-maximum suppression needs to be applied to the output. In most applications, the scale and orientation of typical pedestrians is known. For example, in driving applications in which a camera is fixed to the car, we expect to view mainly vertical pedestrians, and we are interested only in nearby pedestrians. Several pedestrian data sets have been published, and these can be used for training the classifier.

RECONSTRUCTING THE 3D WORLD

In this section we show how to go from the two-dimensional image to a three-dimensional representation of the scene. The fundamental question is this: Given that all points in the scene that fall along a ray to the pinhole are projected to the same point in the image, how do we recover three-dimensional information? Two ideas come to our rescue

- If we have two (or more) images from different camera positions, then we can triangulate to find the position of a point in the scene.
- We can exploit background knowledge about the physical scene that gave rise to the image. Given an object model $\mathbf{P}(\text{Scene})$ and a rendering model $\mathbf{P}(\text{Image} \mid \text{Scene})$, we can compute a posterior distribution $\mathbf{P}(\text{Scene} \mid \text{Image})$.

There is as yet no single unified theory for scene reconstruction. We survey eight commonly used visual cues: **motion**, **binocular stereopsis**, **multiple views**, **texture**, **shading**, **contour**, and **familiar objects**.

Motion parallax

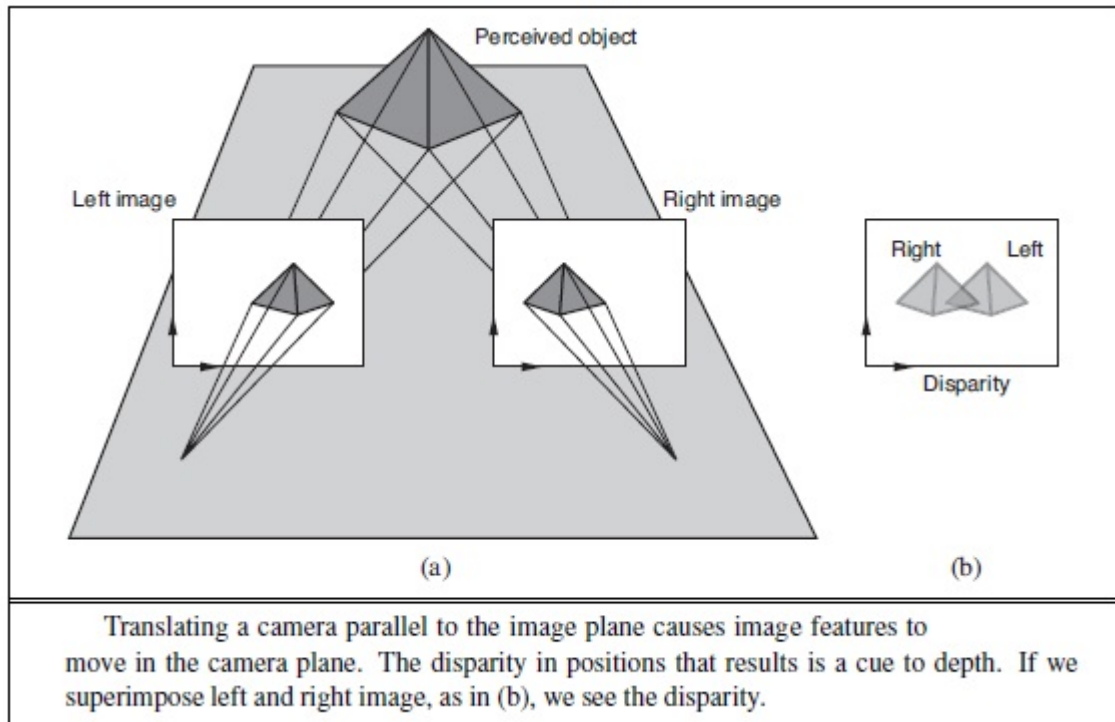
If the camera moves relative to the three-dimensional scene, the resulting apparent motion in the image, optical flow, can be a source of information for both the movement of the camera and depth in the scene. To understand this, we state (without proof) an equation that relates the optical flow to the viewer's translational velocity \mathbf{T} and the depth in the scene. The components of the optical flow field are

$$v_x(x, y) = \frac{-T_x + xT_z}{Z(x, y)}, \quad v_y(x, y) = \frac{-T_y + yT_z}{Z(x, y)},$$

where $Z(x, y)$ is the z-coordinate of the point in the scene corresponding to the point in the image at (x, y) .

Note that both components of the optical flow, $v_x(x, y)$ and $v_y(x, y)$, are zero at the point $x = T_x/T_z, y = T_y/T_z$. This point is called the **focus of expansion** of the flow field. Suppose we change the origin in the x - y plane to lie at the focus of expansion; then the expressions for optical flow take on a particularly simple form. Let (x', y') be the new coordinates defined by $x' = x - T_x/T_z, y' = y - T_y/T_z$. Then

$$v_x(x', y') = \frac{x'T_z}{Z(x', y')}, \quad v_y(x', y') = \frac{y'T_z}{Z(x', y')}.$$



OBJECT RECOGNITION FROM STRUCTURAL INFORMATION

Putting a box around pedestrians in an image may well be enough to avoid driving into them. We have seen that we can find a box by pooling the evidence provided by orientations, using histogram methods to suppress potentially confusing spatial detail. If we want to know more about what someone is doing, we will need to know where their arms, legs, body, and head lie in the picture. Individual body parts are quite difficult to detect on their own using a moving window method, because their color and texture can vary widely and because they are usually small in images. Often, forearms and shins are as small as two to three pixels wide. Body parts do not usually appear on their own, and representing what is connected to what could be quite powerful, because parts that are easy to find might tell us where to look for parts that are small and hard to detect.

Inferring the layout of human bodies in pictures is an important task in vision, because the layout of the body often reveals what people are doing. A model called a **deformable template** can tell us which configurations are acceptable: the elbow can bend but the head is never joined to the foot. The simplest deformable template model of a person connects lower arms to upper arms, upper arms to the torso, and so on.

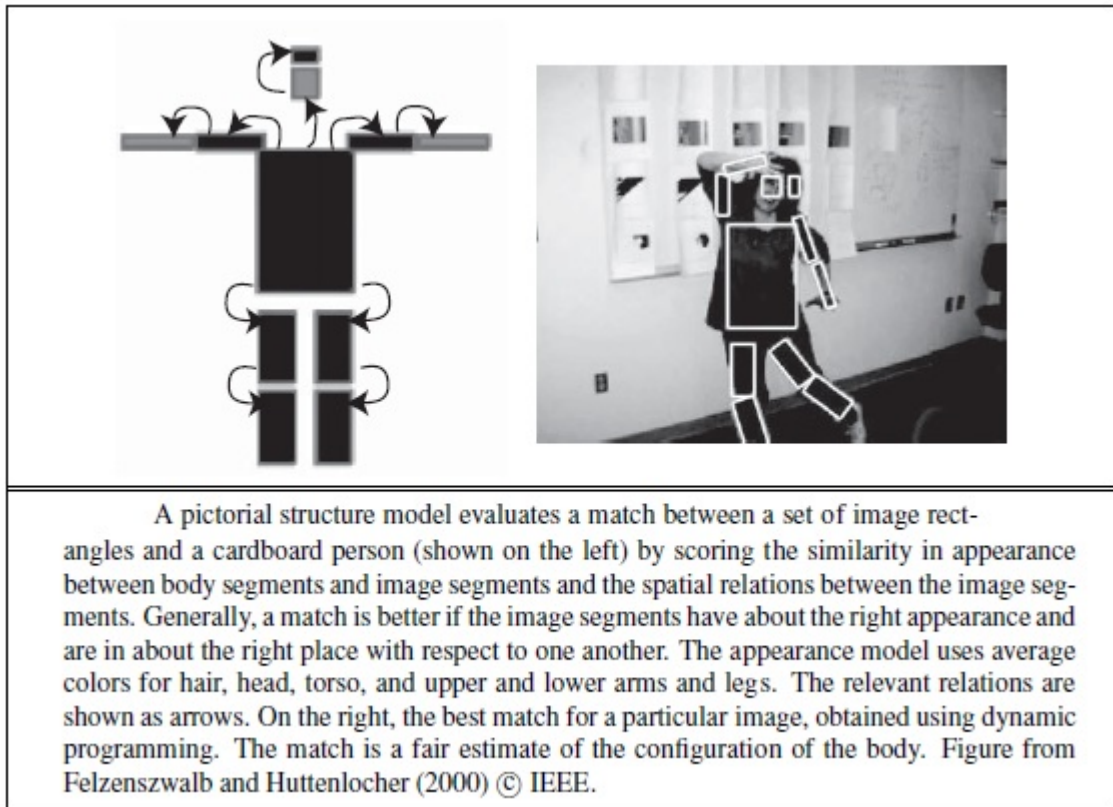
we could represent the fact that left and right upper arms tend to have the same color and texture, as do left and right legs.

The geometry of bodies: Finding arms and legs

For the moment, we assume that we know what the person's body parts look like (e.g., we know the color and texture of the person's clothing). We can model the geometry of the body as a tree of eleven segments (upper and lower left and right arms and legs respectively, a torso, a face, and hair on top of the face) each of which is rectangular. We assume that the position and orientation (**pose**) of POSE the left lower arm is independent of all other segments given the pose of the left upper arm; that the pose of the left upper arm is independent of all segments given the pose of the torso; and extend these assumptions in the obvious way to include the right arm and the legs, the face, and the hair. Such models are often called “cardboard people” models.

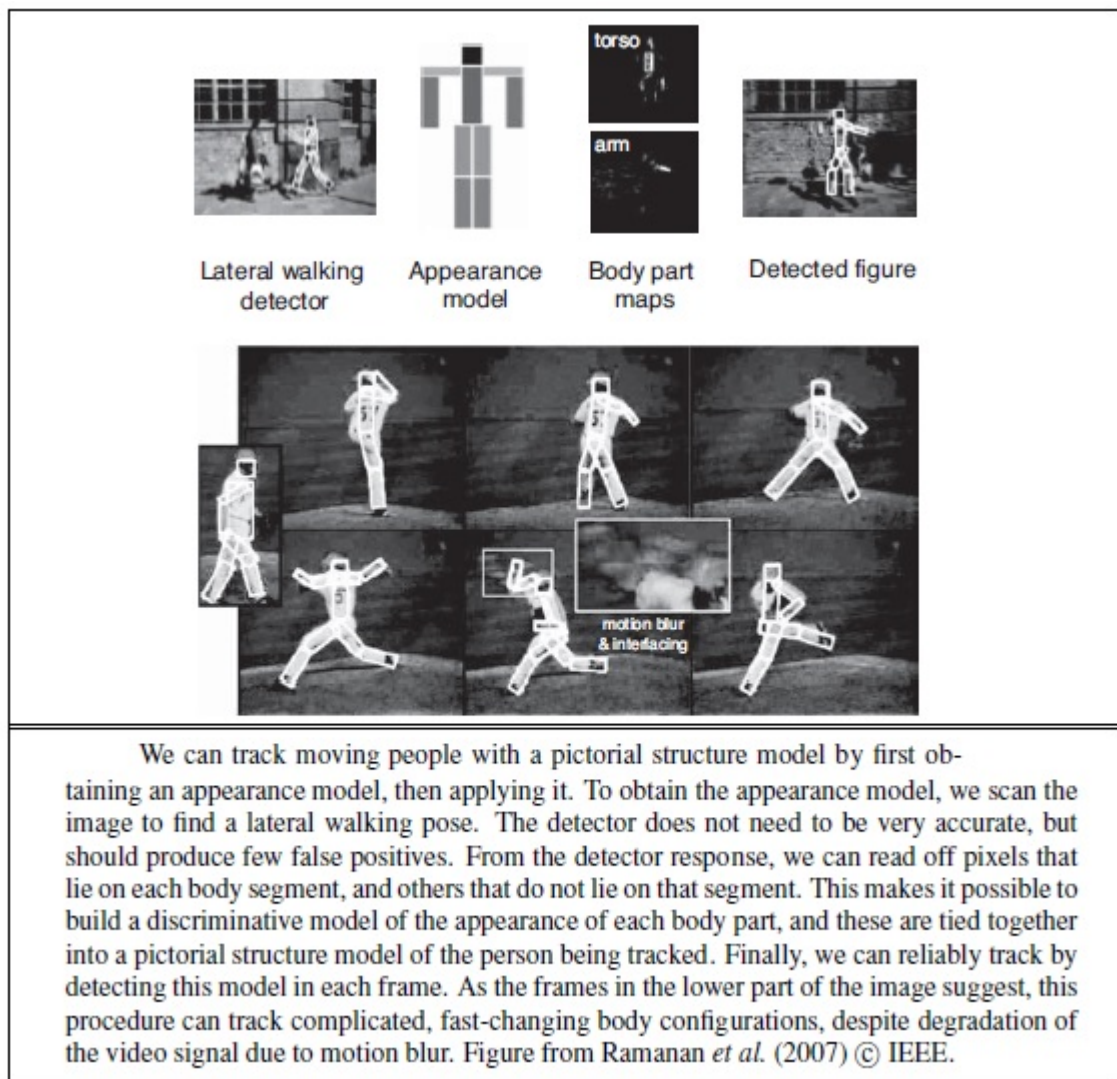
There are two criteria for evaluating a configuration. First, an image rectangle should look like its segment. For the moment, we will remain vague about precisely what that means, but we assume we have a function ϕ_i that scores how well an image rectangle matches a body segment. For each pair of related segments, we have another function ψ that scores how well relations between a pair of image rectangles match those to be expected from the body segments. The dependencies between segments form a tree, so each segment has only one parent, and we could write $\psi_{i,pa(i)}$.

$$\sum_{i \in \text{segments}} \phi_i(m_i) + \sum_{i \in \text{segments}} \psi_{i,pa(i)}(m_i, m_{pa(i)}) .$$



Coherent appearance: Tracking people in video

Tracking people in video is an important practical problem. If we could reliably report the location of arms, legs, torso, and head in video sequences, we could build much improved game interfaces and surveillance systems. Filtering methods have not had much success with this problem, because people can produce large accelerations and move quite fast. This means that for 30 Hz video, the configuration of the body in frame i doesn't constrain the configuration of the body in frame $i+1$ all that strongly. Currently, the most effective methods exploit the fact that appearance changes very slowly from frame to frame. If we can infer an appearance model of an individual from the video, then we can use this information in a pictorial structure model to detect that person in each frame of the video. We can then link these locations across time to make a track.



USING VISION

Some problems are well understood. If people are relatively small in the video frame, and the background is stable, it is easy to detect the people by subtracting a background image from the current frame. If the absolute value of the difference is large, this **background subtraction** declares the pixel to be a foreground pixel; by linking foreground blobs over time, we obtain a track.

Words and pictures

Many Web sites offer collections of images for viewing. How can we find the images we want? Let's suppose the user enters a text query, such as "bicycle race." Some of the images will have keywords or captions attached, or will come from Web pages that contain text near the image.

In the most straightforward version of this task, we have a set of correctly tagged example images, and we wish to tag some test images. This problem is sometimes known as auto-annotation. The most accurate solutions are obtained using nearest-neighbours' methods. One finds the training images

that are closest to the test image in a feature space metric that is trained using examples, then reports their tags.

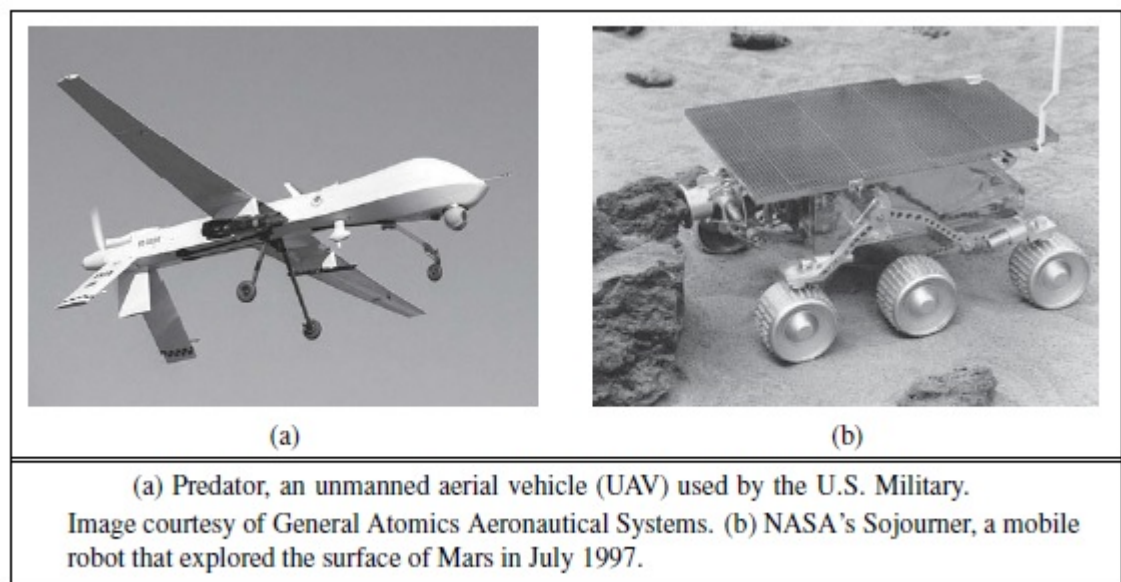
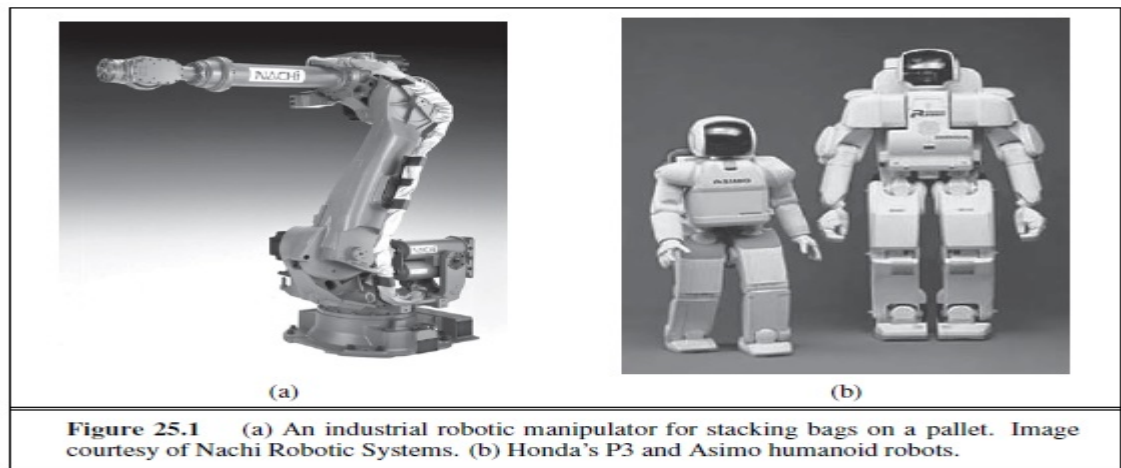
UNIT-V

Robotics: Introduction, Robot Hardware, Robotic Perception, planning to move, planning uncertain movements, Moving, Robotic software architectures, application domains

Philosophical foundations: Weak AI, Strong AI, Ethics and Risks of AI, Agent Components, Agent Architectures, are we going in the right direction, what if AI does succeed.

Introduction

- **Robots** are physical agents that perform tasks by manipulating the physical world. To do so they are equipped with **effectors** such as legs, wheels, joints, and grippers. Effectors have a single purpose: to assert physical forces on the environment. Robots are also equipped with **sensors**, which allow them to perceive their environment.
- Present day robotics employs a diverse set of sensors, including cameras and lasers to measure the environment, and gyroscopes and accelerometers to measure the robot's own motion.
- Most of today's robots fall into one of three primary categories. **Manipulators**, or robot arms, are physically anchored to their workplace, for example in a factory assembly line or on the International Space Station. Manipulator motion usually involves a chain of controllable joints, enabling such robots to place their effectors in any position within the workplace. Manipulators are by far the most common type of industrial robots, with approximately one million units installed worldwide. Some mobile manipulators are used in hospitals to assist surgeons.
- The second category is the **mobile robot**. Mobile robots move about their environment using wheels, legs, or similar mechanisms.
- They have been put to use delivering food in hospitals, moving containers at loading docks, and similar tasks. **Unmanned ground vehicles**, or UGVs, drive autonomously on streets, highways, and off-road.
- The **planetary rover** shown in Figure 25.2(b) explored Mars for a period of 3 months in 1997.
- Other types of mobile robots include **unmanned air vehicles** (UAVs), commonly used for surveillance, crop-spraying,

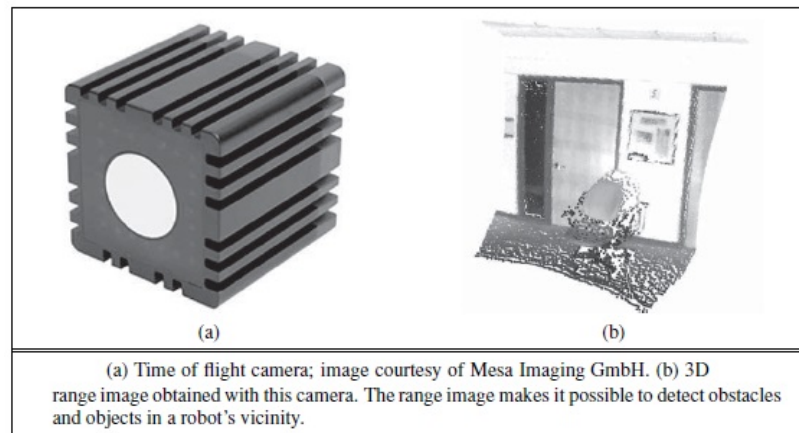


- UAV commonly used by the U.S. military. **Autonomous underwater vehicles** (AUVs) are used in deep sea exploration. Mobile robots deliver packages in the workplace and vacuum the floors at home.
- The third type of robot combines mobility with manipulation, and is often called a **mobile manipulator**. **Humanoid robots** mimic the human torso. shows two early humanoid robots, both manufactured by Honda Corp. in Japan. Mobile manipulators can apply their effectors further afield than anchored manipulators can, but their task is made harder because they don't have the rigidity that the anchor provides.

ROBOT HARDWARE

- **Sensors:** Sensors are the perceptual interface between robot and environment

- **Passive sensors**, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment.
- **Active sensors**, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor. Active sensors tend to provide more information than passive sensors, but at the expense of increased power consumption and with a danger of interference when multiple active sensors are used at the same time. Whether active or passive, sensors can be divided into three types, depending on whether they sense the environment, the robot's location, or the robot's internal configuration.
- **Range finders** are sensors that measure the distance to nearby objects. In the early days of robotics, robots were commonly equipped with **sonar sensors**. Sonar sensors emit directional sound waves, which are reflected by objects, with some of the sound making it back into the sensor. The time and intensity of the returning signal indicates the distance to nearby objects. Sonar is the technology of choice for autonomous underwater vehicles



- **Stereo vision** relies STEREO VISION on multiple cameras to image the environment from slightly different viewpoints, analysing the resulting parallax in these images to compute the range of surrounding objects. For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate.
- Most ground robots are now equipped with optical range finders. Just like sonar sensors, optical range sensors emit active signals (light) and measure the time until a reflection of this signal arrives back at the sensor. shows a **time of flight camera**. This camera acquires range images like the one at up to 60 frames per second.

- Other range sensors use laser beams and special 1-pixel cameras that can be directed using complex arrangements of mirrors or rotating elements. These sensors are called **scanning lidars** (short for *light detection and ranging*). Scanning lidars tend to provide longer ranges than time of flight cameras, and tend to perform better in bright daylight.
- Other common range sensors include radar, which is often the sensor of choice for UAVs. Radar sensors can measure distances of multiple kilometres. On the other extreme end of range sensing are **tactile sensors** such as whiskers, bump panels, and touch-sensitive skin. These sensors measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.
- A second important class of sensors is **location sensors**. Most location sensors use range sensing as a primary component to determine location. Outdoors, the **Global Positioning System** (GPS) is the most common solution to the localization problem. GPS measures the distance to satellites that emit pulsed signals. At present, there are 31 satellites in orbit, transmitting signals on multiple frequencies. GPS receivers can recover the distance to these satellites by analysing phase shifts. By triangulating signals from multiple satellites, GPS receivers can determine their absolute location on Earth to within a few meters.
- **Differential GPS** involves a second ground receiver with known location, providing millimetre accuracy under ideal conditions. Unfortunately, GPS does not work indoors or underwater. Indoors, localization is often achieved by attaching beacons in the environment at known locations. Many indoor environments are full of wireless base stations, which can help robots localize through the analysis of the wireless signal.
- The third important class is **proprioceptive sensors**, which inform the robot of its own motion. To measure the exact configuration of a robotic joint, motors are often equipped with **shaft decoders** that count the revolution of motors in small increments.
- On robot arms, shaft decoders can provide accurate information over any period of time. On mobile robots, shaft decoders that report wheel revolutions can be used for **odometry**—the measurement of distance travelled.
- **Inertial sensors**, such as gyroscopes, rely on the resistance of mass to the change of velocity. They can help reduce uncertainty.
- Other important aspects of robot state are measured by **force sensors** and **torque sensors**. These are indispensable when robots handle fragile objects or objects whose exact shape and location is unknown. Imagine a one-ton robotic manipulator screwing in a light bulb. It would be all too easy to

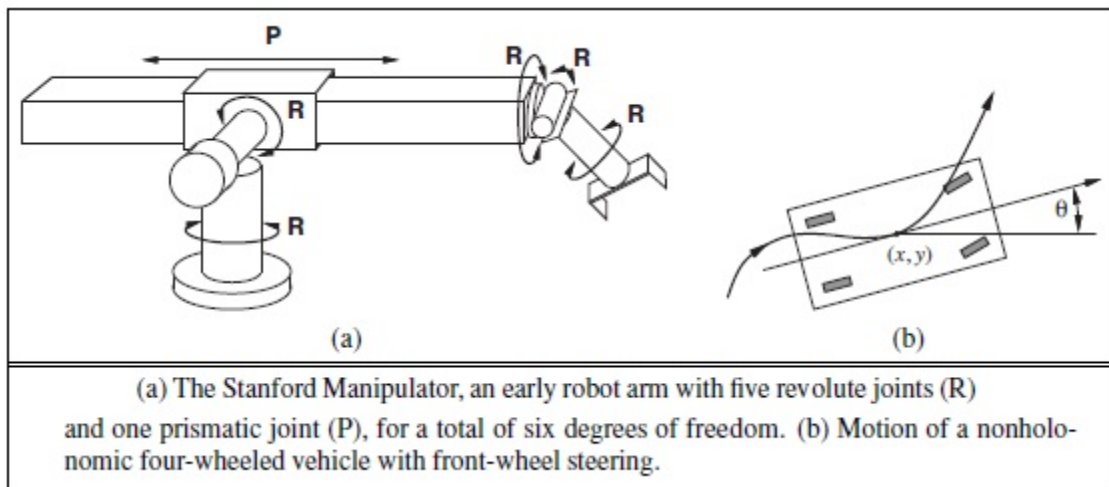
apply too much force and break the bulb. Force sensors allow the robot to sense how hard it is gripping the bulb, and torque sensors allow it to sense how hard it is turning. Good sensors can measure forces in all three translational and three rotational directions. They do this at a frequency of several hundred times a second, so that a robot can quickly detect unexpected forces and correct its actions before it breaks a light bulb.

25.2.2 EFFECTORS

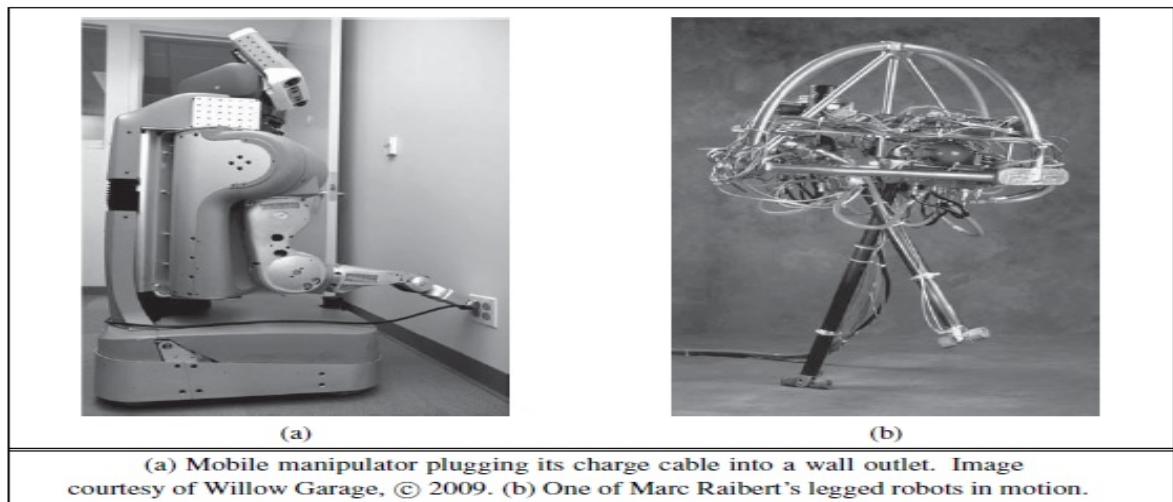
Effectors are the means by which robots move and change the shape of their bodies. To understand the design of effectors, it will help to talk about motion and shape in the abstract, using the concept of a **degree of freedom** (DOF). We count one degree of freedom for each independent direction in which a robot, or one of its effectors, can move. For example, a rigid mobile robot such as an AUV has six degrees of freedom, three for its (x, y, z) location in space and three for its angular orientation, known as *yaw*, *roll*, and *pitch*. These six degrees define the **kinematic state**² or **pose** of the robot. The **dynamic state** of a robot includes these six plus an additional six dimensions for the rate of change of each kinematic dimension, that is, their velocities.

For nonrigid bodies, there are additional degrees of freedom within the robot itself. For example, the elbow of a human arm possesses two degrees of freedom. It can flex the upper arm towards or away, and can rotate right or left. The wrist has three degrees of freedom. It can move up and down, side to side, and can also rotate.

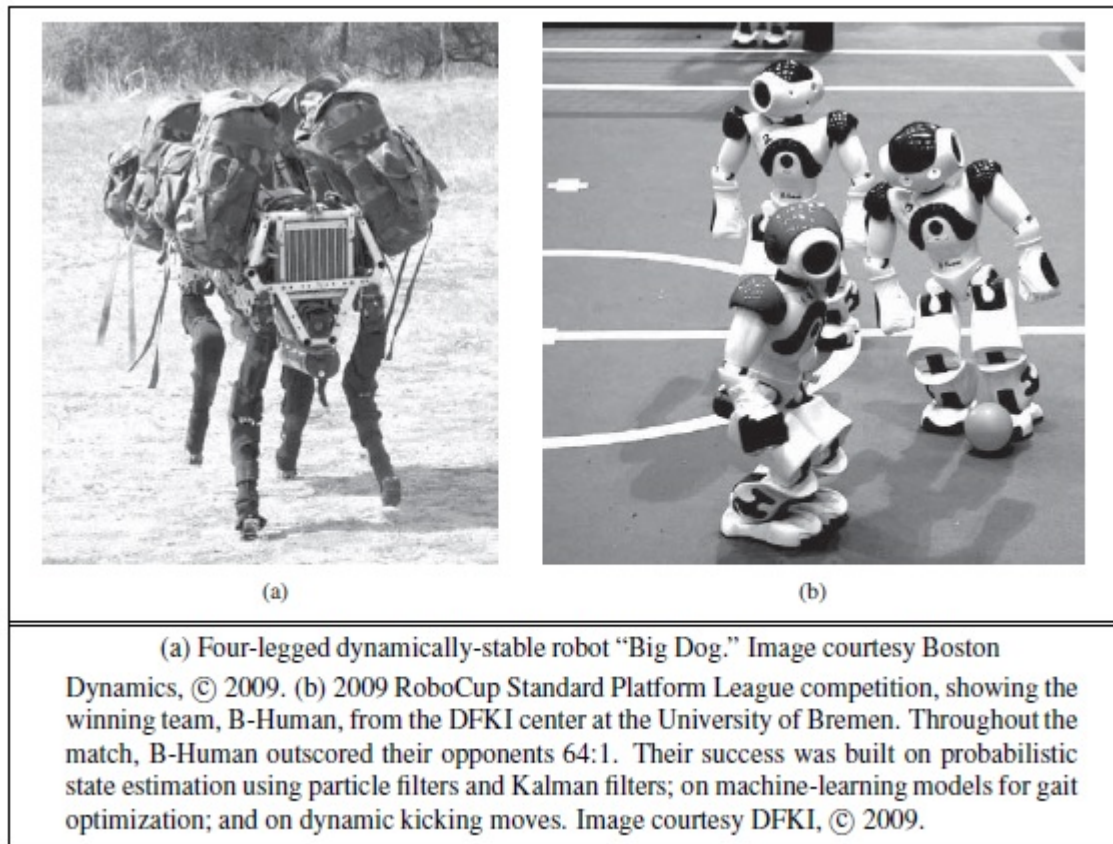
created by five **revolute joints** that generate rotational motion and one **prismatic joint** that generates sliding motion. You can verify that the human arm as a whole has more than six degrees of freedom by a simple experiment: put your hand on the table and notice that you still have the freedom to rotate your elbow without changing the configuration of your hand. Manipulators that have extra degrees of freedom are easier to control than robots with only the minimum number of DOFs. Many industrial manipulators therefore have seven DOFs, not six.



For mobile robots, the DOFs are not necessarily the same as the number of actuated elements. Consider, for example, your average car: it can move forward or backward, and it can turn, giving it two DOFs. In contrast, a car's kinematic configuration is three-dimensional: on an open flat surface, one can easily maneuver a car to any (x, y) point, in any orientation. Thus, the car has three **effective degrees of freedom** but two **control label degrees of freedom**. We say a robot is **nonholonomic** if it has more effective DOFs than controllable DOFs and **holonomic** if the two numbers are the same. Holonomic robots are easier to control—it would be much easier to park a car that could move sideways as well as forward and backward—but holonomic robots are also mechanically more complex. Most robot arms are holonomic, and most mobile robots are nonholonomic. Mobile robots have a range of mechanisms for locomotion, including wheels, tracks, and legs. **Differential drive** robots possess two independently actuated wheels (or tracks), one on each side, as on a military tank. If both wheels move at the same velocity, the robot moves on a straight line. If they move in opposite directions, the robot turns on the spot. An alternative is the **synchro drive**, in which each wheel can move and turn around its own axis. To avoid chaos, the wheels are tightly coordinated. When moving straight, for example, all wheels point in the same direction and move at the same speed. Both differential and synchro drives are nonholonomic. Some more expensive robots use holonomic drives, which have three or more wheels that can be oriented and moved independently external forces.



Legs, unlike wheels, can handle rough terrain. However, legs are notoriously slow on flat surfaces, and they are mechanically difficult to build. Robotics researchers have tried designs ranging from one leg up to opens of legs. Legged robots have been made to walk, run, and even hop—as we see with the legged robot. This robot is **dynamically stable**, meaning that it can remain upright while hopping around. A robot that can remain upright without moving its legs is called **statically stable**. A robot is statically stable if its center of gravity is above the polygon spanned by its legs. The quadruped (four-legged) robot may appear statically stable. However, it walks by lifting multiple legs at the same time, which renders it dynamically stable. The robot can walk on snow and ice, and it will not fall over even if you kick it (as demonstrated in videos available online). Two-legged robots are dynamically stable.



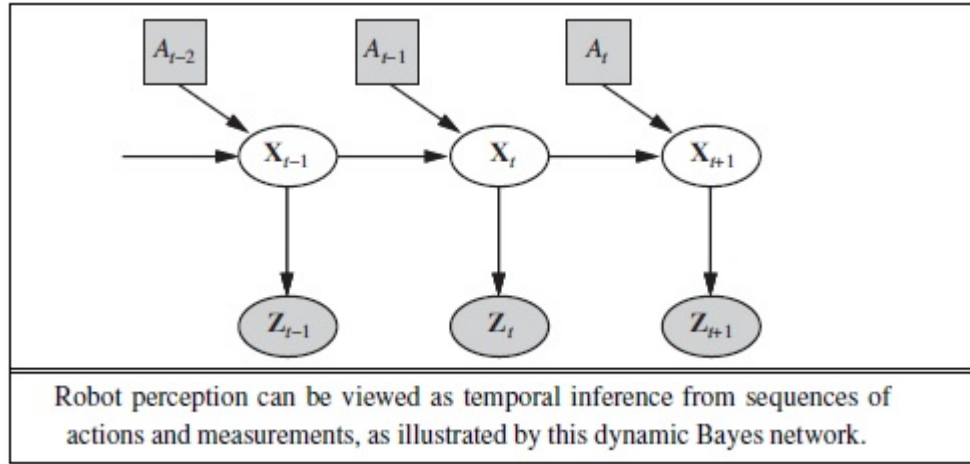
The **electric motor** is the most popular mechanism for both manipulator actuation and locomotion, but **pneumatic actuation** using compressed gas and **hydraulic actuation** using pressurized fluids also have their application niches.

ROBOTIC PERCEPTION

Perception is the process by which robots map sensor measurements into internal representations of the environment. Perception is difficult because sensors are noisy, and the environment is partially observable, unpredictable, and often dynamic. In other words, robots have all the problems of **state estimation** (or **filtering**). As a rule of thumb, good internal representations for robots have three properties: they contain enough information for the robot to make good decisions, they are structured so that they can be updated efficiently, and they are natural in the sense that internal variables correspond to natural state variables in the physical world.

we saw that Kalman filters, HMMs, and dynamic Bayes nets can represent the transition and sensor models of a partially observable environment, and we described both exact and approximate algorithms for updating the **belief state**—the posterior probability distribution over the environment state variables. Several dynamic Bayes net models. For robotics problems, we include the robot’s own past actions as observed

variables in the model. the notation used in this chapter: \mathbf{X}_t is the state of the environment (including the robot) at time t , \mathbf{Z}_t is the observation received at time t , and A_t is the action taken after the observation is received.



We would like to compute the new belief state, $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, \mathbf{a}_{1:t})$, from the current belief state $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, \mathbf{a}_{1:t-1})$ and the new observation \mathbf{z}_{t+1} . We did this in Section 15.2, but here there are two differences: we condition explicitly on the actions as well as the observations, and we deal with *continuous* rather than *discrete* variables.

$$\begin{aligned} & \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, \mathbf{a}_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1}) \int \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, \mathbf{a}_t) P(\mathbf{x}_t \mid \mathbf{z}_{1:t}, \mathbf{a}_{1:t-1}) d\mathbf{x}_t . \end{aligned}$$

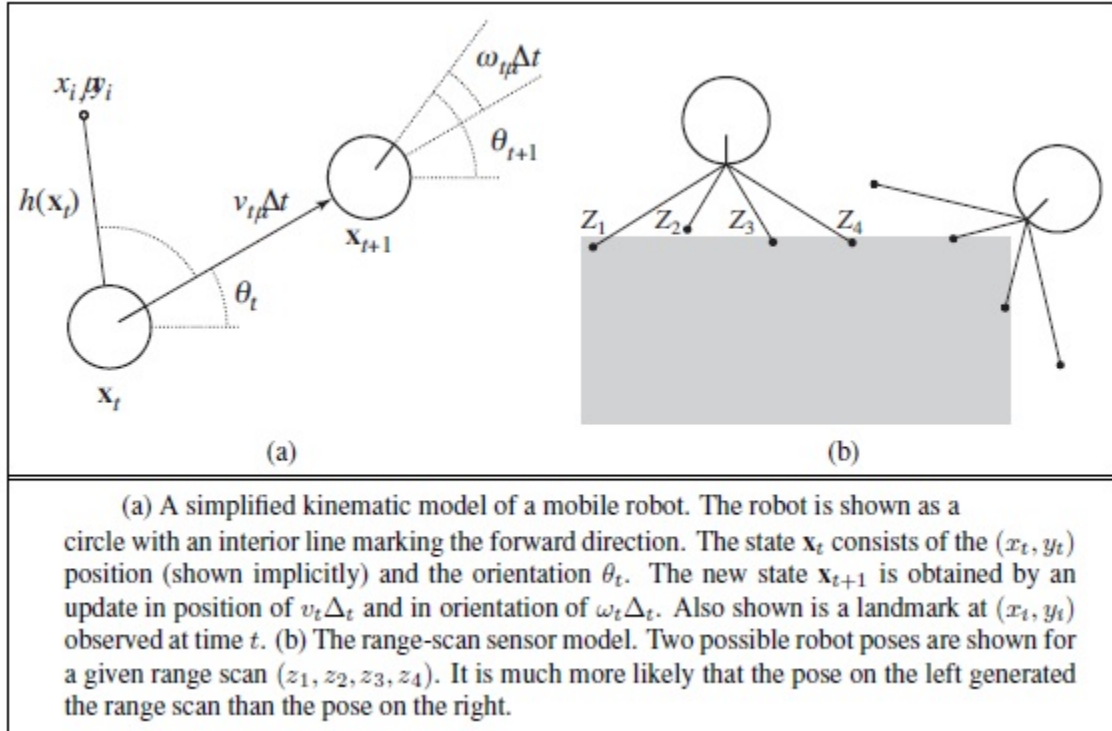
This equation states that the posterior over the state variables \mathbf{X} at time $t + 1$ is calculated recursively from the corresponding estimate one-time step earlier. This calculation involves the previous action \mathbf{a}_t and the current sensor measurement \mathbf{z}_{t+1} . For example, if our goal is to develop a soccer-playing robot, \mathbf{X}_{t+1} might be the location of the soccer ball relative to the robot. The posterior $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, \mathbf{a}_{1:t-1})$ is a probability distribution over all states that captures what we know from past sensor measurements and controls. how to recursively estimate this location, by incrementally folding in sensor measurements (e.g., camera images) and robot motion commands. The probability $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, \mathbf{a}_t)$ is called the **transition model** or **motion model**, and $\mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1})$ is the **sensor model**.

25.3.1 LOCALIZATION AND MAPPING

Localization is the problem of finding out where things are—including the robot itself. Knowledge about where things are is at the core of any successful physical interaction with the environment. For

example, robot manipulators must know the location of objects they seek to manipulate; navigating robots must know where they are to find their way around.

To keep things simple, let us consider a mobile robot that moves slowly in a flat 2D world. Let us also assume the robot is given an exact map of the environment. (An example of such a map appears in Figure 25.10.) The pose of such a mobile robot is defined by its two Cartesian coordinates with values x and y and its heading with value θ , as illustrated. If we arrange those three values in a vector, then any particular state is given by $\mathbf{X}_t = (x_t, y_t, \theta_t)$.



In the kinematic approximation, each action consists of the “instantaneous” specification of two velocities—a translational velocity v_t and a rotational velocity ω_t . For small time intervals Δt , a crude deterministic model of the motion of such robots is given by

$$\hat{\mathbf{X}}_{t+1} = f(\mathbf{X}_t, \underbrace{v_t, \omega_t}_{a_t}) = \mathbf{X}_t + \begin{pmatrix} v_t \Delta t \cos \theta_t \\ v_t \Delta t \sin \theta_t \\ \omega_t \Delta t \end{pmatrix}$$

The notation $\hat{\mathbf{X}}$ refers to a deterministic state prediction. Of course, physical robots are somewhat unpredictable. This is commonly modelled by a Gaussian distribution with mean $f(\mathbf{X}_t, v_t, \omega_t)$ and covariance Σ_x . (See Appendix A for a mathematical definition.)

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{X}_t, v_t, \omega_t) = N(\hat{\mathbf{X}}_{t+1}, \Sigma_x).$$

Next, we need a sensor model. We will consider two kinds of sensor model. The first assumes that the sensors detect stable, recognizable features of the environment called **landmarks**. For each landmark, the range and bearing are reported. Suppose the robot's state is $\mathbf{x}_t = (x_t, y_t, \theta_t)$ and it senses a landmark whose location is known to be (x_i, y_i) . Without noise, the range and bearing can be calculated by simple geometry.

$$\hat{\mathbf{z}}_t = h(\mathbf{x}_t) = \left(\begin{array}{c} \sqrt{(x_t - x_i)^2 + (y_t - y_i)^2} \\ \arctan \frac{y_i - y_t}{x_i - x_t} - \theta_t \end{array} \right)$$

A somewhat different sensor model is used for an array of range sensors, each of which has a fixed bearing relative to the robot. Such sensors produce a vector of range values $\mathbf{z}_t = (z_1, \dots, z_M)$. Given a pose \mathbf{x}_t , let \hat{z}_j be the exact range along the j th beam direction from \mathbf{x}_t to the nearest obstacle. As before, this will be corrupted by Gaussian noise. Typically, we assume that the errors for the different beam directions are independent and identically distributed, so we have

$$P(\mathbf{z}_t | \mathbf{x}_t) = \alpha \prod_{j=1}^M e^{-(z_j - \hat{z}_j)^2 / 2\sigma^2}.$$

```

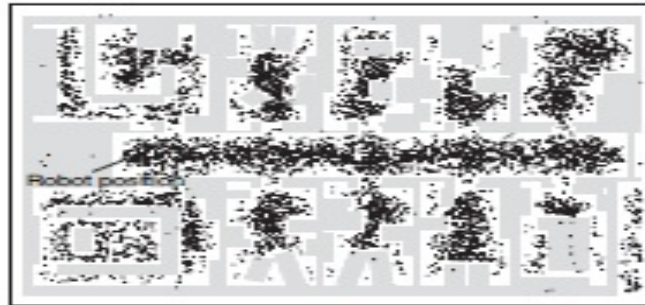
function MONTE-CARLO-LOCALIZATION( $a, z, N, P(X'|X, v, \omega), P(z|z^*), m$ ) returns
a set of samples for the next time step
inputs:  $a$ , robot velocities  $v$  and  $\omega$ 
           $z$ , range scan  $z_1, \dots, z_M$ 
           $P(X'|X, v, \omega)$ , motion model
           $P(z|z^*)$ , range sensor noise model
           $m$ , 2D map of the environment
persistent:  $S$ , a vector of samples of size  $N$ 
local variables:  $W$ , a vector of weights of size  $N$ 
                    $S'$ , a temporary vector of particles of size  $N$ 
                    $W'$ , a vector of weights of size  $N$ 

if  $S$  is empty then      /* initialization phase */
  for  $i = 1$  to  $N$  do
     $S[i] \leftarrow$  sample from  $P(X_0)$ 
  for  $i = 1$  to  $N$  do    /* update cycle */
     $S'[i] \leftarrow$  sample from  $P(X'|X = S[i], v, \omega)$ 
     $W'[i] \leftarrow 1$ 
    for  $j = 1$  to  $M$  do
       $z^* \leftarrow$  RAYCAST( $j, X = S'[i], m$ )
       $W'[i] \leftarrow W'[i] \cdot P(z_j | z^*)$ 
     $S \leftarrow$  WEIGHTED-SAMPLE-WITH-REPLACEMENT( $N, S', W'$ )
return  $S$ 

```

A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

Localization using particle filtering MONTE CARLO is called **Monte Carlo localization**, or MCL. The MCL algorithm is an instance of the particle-filtering algorithm of All we need to do is supply the appropriate motion model and sensor model. shows one version using the range-scan model. The operation of the algorithm is illustrated in as the robot finds out where it is inside an office building. In the first image, the particles are uniformly distributed based on the prior, indicating global uncertainty about the robot's position. In the second image, the first set of measurements arrives and the particles form clusters in the areas of high posterior belief. In the third, enough measurements are available to push all the particles to a single location.



(a)

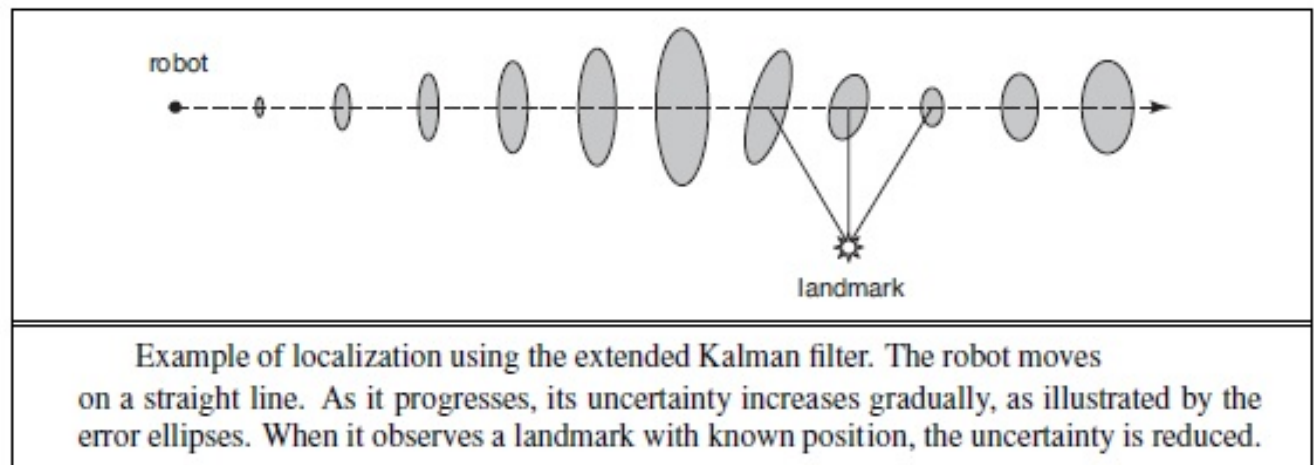
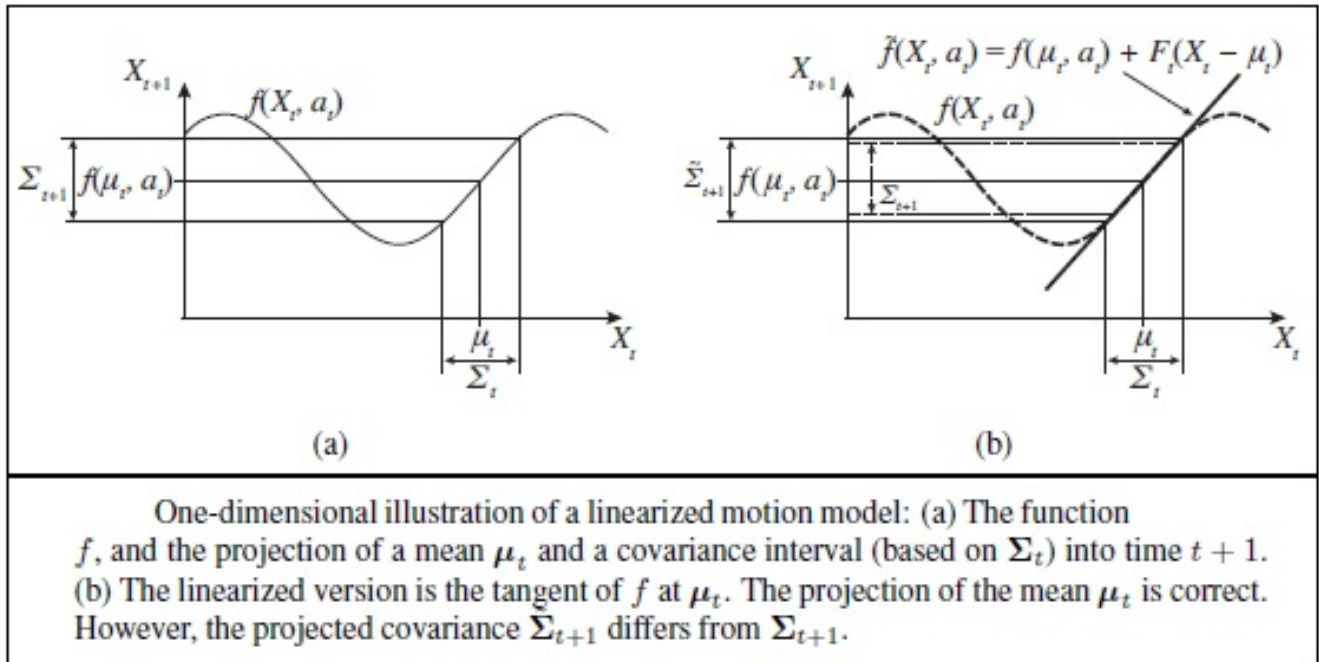


(b)



(c)

Monte Carlo localization, a particle filtering algorithm for mobile robot localization. (a) Initial, global uncertainty. (b) Approximately bimodal uncertainty after navigating in the (symmetric) corridor. (c) Unimodal uncertainty after entering a room and finding it to be distinctive.



the state vector to include the locations of the landmarks in the environment. Luckily, the EKF update scales quadratically, so for small maps (e.g., a few hundred landmarks) the computation is quite feasible. Richer maps are often obtained using graph relaxation methods, similar to the Bayesian network inference techniques.

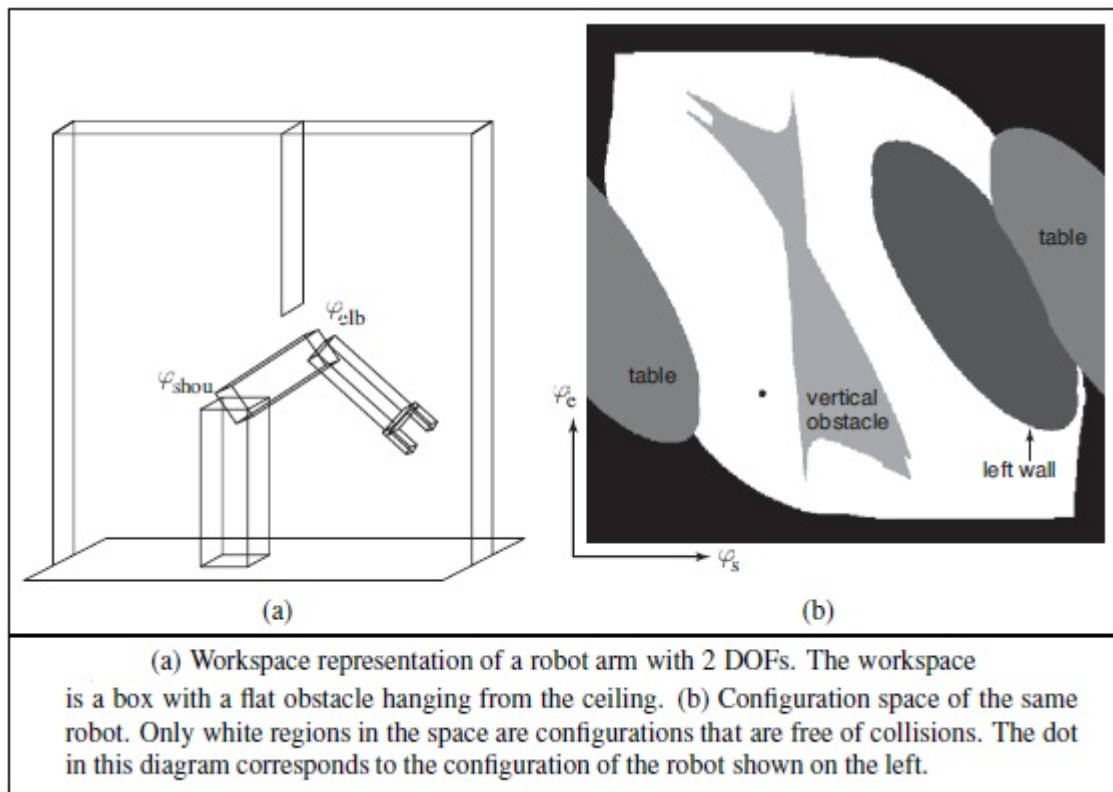
PLANNING TO MOVE

All of a robot's deliberations ultimately come down to deciding how to move effectors. The **point-to-point motion** problem is to deliver the robot or its end effector to a designated target location. A greater challenge is the **compliant motion** problem, in which a robot moves while being in physical contact with an

obstacle. An example of compliant motion is a robot manipulator that screws in a light bulb, or a robot that pushes a box across a table top. We begin by finding a suitable representation in which motion-planning problems can be described and solved. It turns out that the **configuration space**—the space of robot states defined by location, orientation, and joint angles—is a better place to work than the original 3D space. The **path lanning** problem is to find a path from one configuration to another in configuration space. We have already encountered various versions of the path-planning problem throughout this book; the complication added by robotics is that path planning involves continuous spaces. There are two main approaches: **cell decomposition** and **skeletonization**. Each reduces the continuous path-planning problem to a discrete graph-search problem. In this section, we assume that motion is deterministic and that localization of the robot is exact. Subsequent sections will relax these assumptions.

25.4.1 Configuration space

We will start with a simple representation for a simple robot motion problem. Consider the robot arm shown in Figure 25.14(a). It has two joints that move independently. Moving the joints alters the (x, y) coordinates of the elbow and the gripper. (The arm cannot move in the z direction.) This suggests that the robot's configuration can be described by a four-dimensional coordinate: (x_e, y_e) for the location of the elbow relative to the environment and (x_g, y_g) for the location of the gripper. Clearly, these four coordinates characterize the full state of the robot. They constitute what is known as **workspace representation**, since the coordinates of the robot are specified in the same coordinate system as the objects it seeks to manipulate (or to avoid). Workspace representations are well-suited for collision checking, especially if the robot and all objects are represented by simple polygonal models. The problem with the workspace representation is that not all workspace coordinates are actually attainable, even in the absence of obstacles. This is because of the **linkage constraints** on the space of attainable workspace coordinates. For example, the elbow position (x_e, y_e) and the gripper position (x_g, y_g) are always a fixed distance apart, because they are joined by a rigid forearm. A robot motion planner defined over workspace coordinates faces the challenge of generating paths that adhere to these constraints. This is particularly tricky

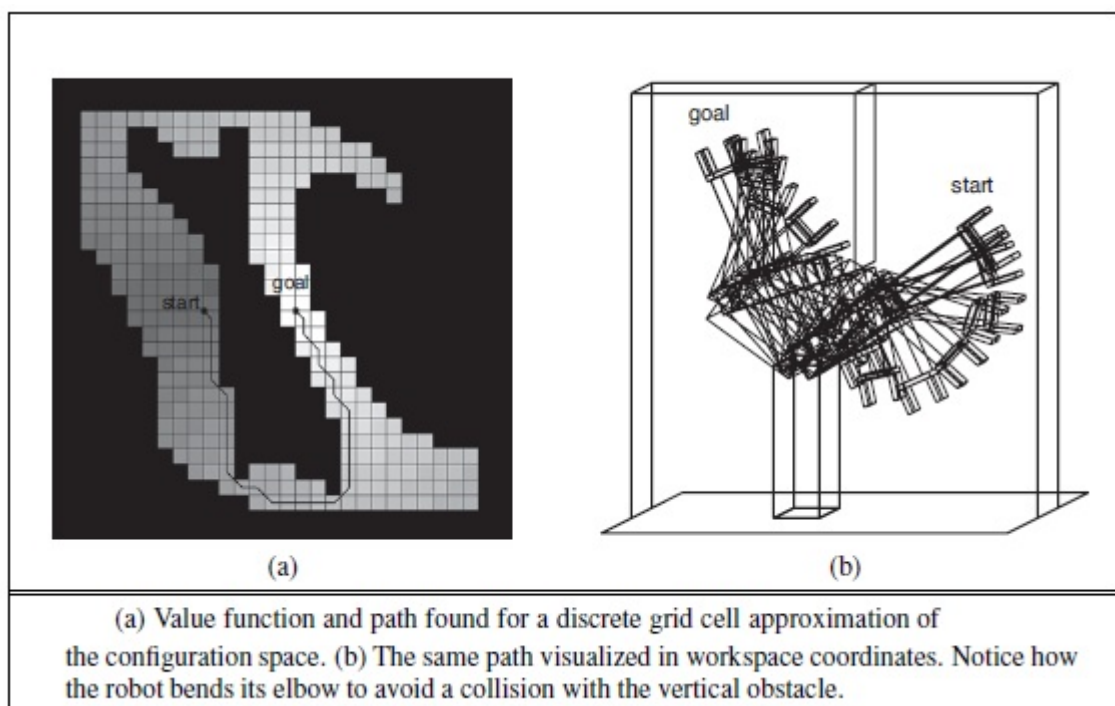


Unfortunately, configuration spaces have their own problems. The task of a robot is usually expressed in workspace coordinates, not in configuration space coordinates. This raises the question of how to map between workspace coordinates and configuration space. Transforming configuration space coordinates into workspace coordinates is simple: it involves a series of straightforward coordinate transformations. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as **kinematics**.

25.4.2 Cell decomposition methods

The first CELL approach to path planning uses **cell decomposition**—that is, it decomposes the free space into a finite number of contiguous regions, called cells. These regions have the important property that the path-planning problem within a single region can be solved by simple means (e.g., moving along a straight line). The path-planning problem then becomes a discrete graph-search problem, very much like the search problems introduced in Chapter 3. The simplest cell decomposition consists of a regularly spaced grid. shows a square grid decomposition of the space and a solution path that is optimal for this grid size. Grayscale shading indicates the *value* of each free-space grid cell—i.e., the cost of the shortest path from that cell to the goal. (These values can be computed by a deterministic form of the VALUE-ITERATION shows the corresponding workspace trajectory for the arm. Of course, we can also use the A*

algorithm to find a shortest path. Such a decomposition has the advantage that it is extremely simple to implement, but it also suffers from three limitations. First, it is workable only for low-dimensional configuration spaces, because the number of grid cells increases exponentially with d , the number of dimensions. Sounds familiar? This is the curse! Dimensionality @of dimensionality. Second, there is the problem of what to do with cells that are “mixed”—that is, neither entirely within free space nor entirely within occupied space. A solution path that includes such a cell may not be a real solution, because there may be no way to cross the cell in the desired direction in a straight line. This would make the path planner *unsound*. On the other hand, if we insist that only completely free cells may be used, the planner will be *incomplete*, because it might



Cell decomposition methods can be improved in a number of ways, to alleviate some of these problems. The first approach allows *further subdivision* of the mixed cells—perhaps using cells of half the original size. This can be continued recursively until a path is found that lies entirely within free cells. (Of course, the method only works if there is a way to decide if a given cell is a mixed cell, which is easy only if the configuration space boundaries have relatively simple mathematical descriptions.) This method is complete provided there is a bound on the smallest passageway through which a solution must pass. Although it focuses most of the computational effort on the tricky areas within the configuration space, it still fails to scale well to high-dimensional problems because each recursive splitting of a cell creates $2d$ smaller cells. A second way to obtain a complete algorithm is to insist on an **exact cell decomposition** of the free space. This

EXACT CELL method must allow cells to be irregularly shaped where they meet the boundaries of free space, but the shapes must still be “simple” in the sense that it should be easy to compute a traversal of any free cell. This technique requires some quite advanced geometric ideas, so we shall not pursue it further here.

The exact, continuous state that was attained with the cell was first expanded in the search. Assume further, that when propagating information to nearby grid cells, we use this continuous state as a basis, and apply the continuous robot motion model for jumping to nearby cells. In doing so, we can now guarantee that the resulting trajectory is smooth and can indeed be executed by the robot. One algorithm that implements this is **hybrid A***.

25.4.3 Modified cost functions

This problem can be solved by introducing a **potential field**. A potential field is a function defined over state space, whose value grows with the distance to the closest obstacle. shows such a potential field—the darker a configuration state, the closer it is to an obstacle. The potential field can be used as an additional cost term in the shortest-path calculation. This induces an interesting trade off. On the one hand, the robot seeks to minimize path length to the goal. On the other hand, it tries to stay away from obstacles by virtue of minimizing the potential function. With the appropriate weight balancing the two objectives, a resulting path may look like the one shown in Figure 25.17(b). This figure also displays the value function derived from the combined cost function, again calculated by value iteration. Clearly, the resulting path is longer, but it is also safer. There exist many other ways to modify the cost function. For example, it may be desirable to *smooth* the control parameters over time. For example, when driving a car, a smooth path is better than a jerky one. In general, such higher-order constraints are not easy to accommodate in the planning process, unless we make the most recent steering command a part of the state. However, it is often easy to smooth the resulting trajectory after planning, using conjugate gradient methods. Such post-planning smoothing is essential in many real world applications.

PLANNING UNCERTAIN MOVEMENTS

Most of today’s robots use deterministic algorithms for decision making, such as the path-planning algorithms of the previous section. To do so, it is common practice to extract **MOST LIKELY STATE** the **most likely state** from the probability distribution produced by the state estimation algorithm. The advantage of this approach is purely computational. Planning paths through configuration space is already a challenging problem; it would be worse if we had to work with a full probability distribution over states.

Ignoring uncertainty in this way works when the uncertainty is small. In fact, when the environment model changes over time as the result of incorporating sensor measurements, many robots plan paths online during plan execution. **ONLINE REPLANNING** This is the **online re-planning** technique.

The field of robotics has adopted a range of techniques for accommodating uncertainty. Some are derived from the algorithms given in for decision making under uncertainty. If the robot faces uncertainty only in its state transition, but its state is fully observable, the problem is best modeled as a Markov decision process (MDP). The solution of an MDP is an optimal **policy**, which tells the robot what to do in every possible state. In this way, it can handle all sorts of motion errors, whereas a single-path solution from a deterministic planner would be much less robust. In robotics, policies are called **navigation NAVIGATION functions**. The value FUNCTION

Just as in, partial observability makes the problem much harder. The resulting robot control problem is a partially observable MDP, or POMDP. In such situations, the robot maintains an internal belief state, like the ones discussed. The solution to a POMDP is a policy defined over the robot's belief state. Put differently, the input to the policy is an entire probability distribution. This enables the robot to base its decision not only on what it knows, but also on what it does not know. For example, if it is uncertain INFORMATION about a critical state variable, it can rationally invoke an **information gathering action**. This GATHERING ACTION is impossible in the MDP framework, since MDPs assume full observability. Unfortunately, techniques that solve POMDPs exactly are inapplicable to robotics—there are no known techniques for high-dimensional continuous spaces. Discretization produces POMDPs that are far too large to handle. One remedy is to make the minimization of uncertainty a control object COASTAL For example, the **coastal navigation** heuristic requires the robot to stay near known NAVIGATION landmarks to decrease its uncertainty. Another approach applies variants of the probabilistic roadmap planning method to the belief space representation. Such methods tend to scale better to large discrete POMDPs.

MOVING

So far, we have talked about how to *plan* motions, but not about how to *move*. Our plans particularly those produced by deterministic path planners—assume that the robot can simply follow any path that the algorithm produces. In the real world, of course, this is not the case. Robots have inertia and cannot execute arbitrary paths except at arbitrarily slow speeds. In most cases, the robot gets to exert forces rather than specify positions. This section discusses methods for calculating these forces.

25.6.1 Dynamics and control

Section 25.2 introduced the notion of **dynamic state**, which extends the kinematic state of a robot by its velocity. For example, in addition to the angle of a robot joint, the dynamic state also captures the rate of change of the angle, and possibly even its momentary acceleration. The transition model for a dynamic state representation includes the effect of forces on this rate of change. Such models are typically DIFFERENTIAL expressed via **differential equations**, which are equations that relate a quantity (e.g., a kinematic state) to the change of the quantity over time (e.g., velocity). In principle, we could have chosen to plan robot motion using dynamic models, instead of our kinematic models. Such a methodology would lead to superior robot performance, if we could generate the plans. However, the dynamic state has higher dimension than the kinematic space, and the curse of dimensionality would render many motion planning algorithms inapplicable for all but the most simple robots. For this reason, practical robot systems often rely on simpler kinematic path planners. A common technique to compensate for the limitations of kinematic plans is to use a CONTROLLER separate mechanism, a **controller**, for keeping the robot on track. Controllers are techniques for generating robot controls in real time using feedback from the environment, so as to achieve a control objective. If the objective is to keep the robot on a preplanned path, it is often referred to as a **reference controller** and the path is called a **reference path**. Controllers that optimize a global cost function are known as **optimal controllers**. Optimal policies for continuous MDPs are, in effect, optimal controllers. On the surface, the problem of keeping a robot on a prespecified path appears to be relatively straightforward. In practice, however, even this seemingly simple problem has its pitfalls. illustrates what can go wrong; it shows the path of a robot that attempts to follow a kinematic path. Whenever a deviation occurs—whether due to noise or to constraints on the forces the robot can apply—the robot provides an opposing force whose magnitude is proportional to this deviation. Intuitively, this might appear plausible, since deviations should be compensated by a counterforce to keep the robot on track. However, illustrates, our controller causes the robot to vibrate rather violently. The vibration is the result of a natural inertia of the robot arm: once driven back to its reference position the robot then overshoots, which induces a symmetric error with opposite sign. Such overshooting may continue along an entire trajectory, and the resulting robot motion is far from desirable.

ROBOTIC SOFTWARE ARCHITECTURES

A methodology for structuring algorithms is SOFTWARE called a **software architecture**. An architecture includes languages and tools for writing programs, as well as an overall philosophy for how programs can be brought together. Modern-day software architectures for robotics must decide how to combine reactive control and model-based deliberative planning. In many ways, reactive and deliberate techniques have orthogonal strengths and weaknesses. Reactive control is sensor-driven and appropriate for making low-level decisions in real time. However, it rarely yields a plausible solution at the global level, because global control decisions depend on information that cannot be sensed at the time of decision making. For such problems, deliberate planning is a more appropriate choice.

Consequently, most robot architectures use reactive techniques at the lower levels of control and deliberative techniques at the higher levels. We encountered such a combination in our discussion of PD controllers, where we combined a (reactive) PD controller with a (deliberate) path planner. Architectures that combine reactive and deliberate techniques are called **hybrid architectures**.

25.7.1 Sub Sumption architecture

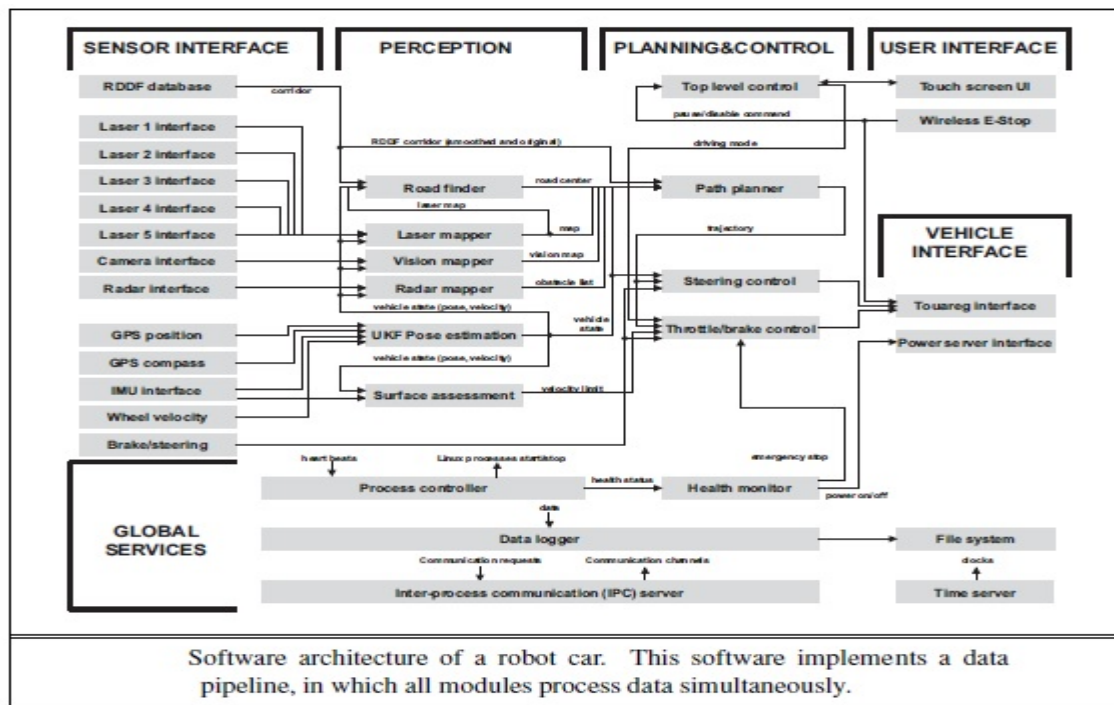
The **sub sumption architecture** (Brooks, 1986) is a framework for assembling reactive controllers out of finite state machines. Nodes in these machines may contain tests for certain sensor variables, in which case the execution trace of a finite state machine is conditioned on the outcome of such a test. Arcs can be tagged with messages that will be generated when traversing them, and that are sent to the robot's motors or to other finite state machines. Additionally, finite state machines possess internal timers (clocks) that control the time it takes to traverse an arc. The resulting machines are referred to as **augmented finite state machines**, or AFSMs, where the augmentation refers to the use of clocks.

An example of a simple AFSM is the four-state machine which generates cyclic leg motion for a hexapod walker. This AFSM implements a cyclic controller, whose execution mostly does not rely on environmental feedback. The forward swing phase, however, does rely on sensor feedback. If the leg is stuck, meaning that it has failed to execute the forward swing, the robot retracts the leg, lifts it up a little higher, and attempts to execute the forward swing once again. Thus, the controller is able to *react* to contingencies arising from the interplay of the robot and its environment.

The sub sumption architecture offers additional primitives for synchronizing AFSMs, and for combining output values of multiple, possibly conflicting AFSMs. In this way, it enables the programmer to compose increasingly complex controllers in a bottom-up fashion.

25.7.2 Three-layer architecture

Hybrid architectures combine reaction with deliberation. The most popular hybrid architecture is the **three-layer architecture**, which THREE-LAYER consists of a reactive layer, an executive layer, and a deliberative layer. The **reactive layer** provides low-level control to the robot. It is characterized by a tight sensor–action loop. Its decision cycle is often on the order of milliseconds. The **executive layer** (or sequencing layer) serves as the glue between the reactive layer and the deliberative layer. It accepts directives by the deliberative layer, and sequences them for the reactive layer. For example, the executive layer might handle a set of via-points generated by a deliberative path planner, and make decisions as to which reactive behaviour to invoke. Decision cycles at the executive layer are usually in the order of a second. The executive layer is also responsible for integrating sensor information into an internal state representation. For example, it may host the robot’s localization and online mapping routines.



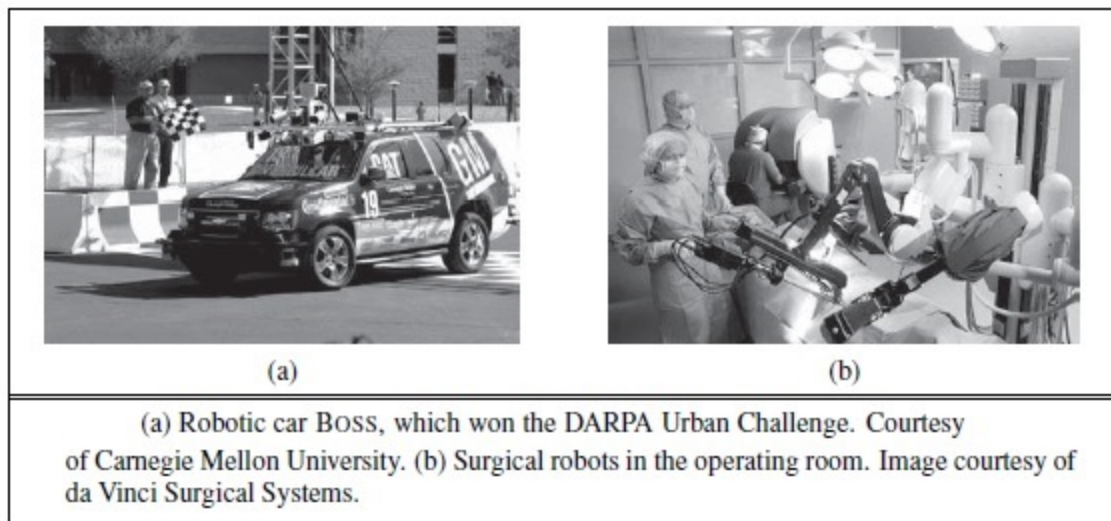
25.7.3 Pipeline architecture

Another architecture for robots is known as the **pipeline architecture**. Just like the sub sumption architecture, the pipeline architecture executes multiple process in parallel. However, the specific modules in this architecture resemble those in the three-layer architecture. pipeline architecture, which is used to control an autonomous car. Data enters this pipeline at the **sensor interface layer**. The **perception layer**

APPLICATION DOMAINS

Here are some of the prime application domains for robotic technology. **Industry and Agriculture.** Traditionally, robots have been fielded in areas that require difficult human labor, yet are structured enough

to be amenable to robotic automation. The best example is the assembly line, where manipulators routinely perform tasks such as assembly, part placement, material handling, welding, and painting. In many of these tasks, robots have become more cost-effective than human workers. Outdoors, many of the heavy machines that we use to harvest, mine, or excavate earth have been turned into robots. For example, a project at Carnegie Mellon University has demonstrated that robots can strip paint off large ships about 50 times faster than people can, and with a much-reduced environmental impact. Prototypes of autonomous mining robots have been found to be faster and more precise than people in transporting ore in underground mines. Robots have been used to generate high-precision maps of abandoned mines and sewer systems. While many of these systems are still in their prototype stages, it is only a matter of time until robots will take over much of the semi mechanical work that is presently performed by people.



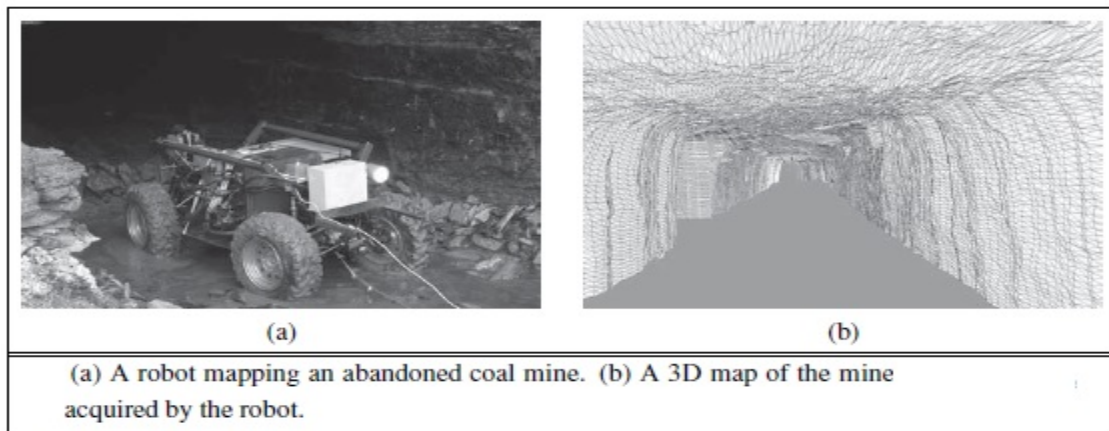
Transportation. Robotic transportation has many facets: from autonomous helicopters that deliver payloads to hard-to-reach locations, to automatic wheelchairs that transport people who are unable to control wheelchairs by themselves, to autonomous straddle carriers that outperform skilled human drivers when transporting containers from ships to trucks on loading docks. A prime example of indoor transportation robots, or gofers, is the Helpmate robot. This robot has been deployed in dozens of hospitals to transport food and other items. In factory settings, autonomous vehicles are now routinely deployed to transport goods in warehouses and between production lines. Many of these robots require environmental modifications for their operation. The most common modifications are localization aids such as inductive loops in the floor, active beacons, or barcode tags. An open challenge in robotics is the design of robots that can use natural cues, instead of artificial devices, to navigate, particularly in environments such as the deep ocean where GPS is unavailable.

Robotic cars. Most of use cars every day. Many of us make cell phone calls while driving. Some of us even text. The sad result: more than a million people die every year in traffic accidents. Robotic cars like BOSS and STANLEY offer hope: Not only will they make driving much safer, but they will also free us from the need to pay attention to the road during our daily commute.

Progress in robotic cars was stimulated by the DARPA Grand Challenge, a race over 100 miles of rehearsed desert terrain, which represented a much more challenging task than had ever been accomplished before. Stanford's STANLEY vehicle completed the course in less than seven hours in 2005, winning a \$2 million prize and a place in the National Museum of American History. Figure 25.28(a) depicts BOSS, which in 2007 won the DARPA Urban Challenge, a complicated road race on city streets where robots faced other robots and had to obey traffic rules.

Health care. Robots are increasingly used to assist surgeons with instrument placement when operating on organs as intricate as brains, eyes, and hearts.. Robots have become indispensable tools in a range of surgical procedures, such as hip replacements, thanks to their high precision. In pilot studies, robotic devices have been found to reduce the danger of lesions when performing colonoscopy. Outside the operating room, researchers have begun to develop robotic aides for elderly and handicapped people, such as intelligent robotic walkers and intelligent toys that provide reminders to take medication and provide comfort. Researchers are also working on robotic devices for rehabilitation that aid people in performing certain exercises.

Hazardous environments. Robots have assisted people in cleaning up nuclear waste, most notably in Chernobyl and Three Mile Island. Robots were present after the collapse of the World Trade Center, where they entered structures deemed too dangerous for human search and rescue crews. Some countries have used robots to transport ammunition and to defuse bombs—a notoriously dangerous task. A number of research projects are presently developing prototype robots for clearing minefields, on land and at sea. Most existing robots for these tasks are teleoperated—a human operates them by remote control. Providing such robots with autonomy is an important next step.



Entertainment. Robots have begun to conquer the entertainment and toy industry. we see **robotic soccer**, a competitive game very much like human soccer, but played with autonomous mobile robots. Robot soccer provides great opportunities for research in AI, since it raises a range of problems relevant to many other, more serious robot applications. Annual robotic soccer competitions have attracted large numbers of AI researchers and added a lot of excitement to the field of robotics.

Human augmentation. A final application domain of robotic technology is that of human augmentation. Researchers have developed legged walking machines that can carry people around, very much like a wheelchair. Several research efforts presently focus on the development of devices that make it easier for people to walk or move their arms by providing additional forces through extra skeletal attachments.

What is strong AI?

Strong artificial intelligence (AI), also known as artificial general intelligence (AGI) or general AI, is a theoretical form of AI used to describe a certain mindset of AI development. If researchers are able to develop Strong AI, the machine would require an intelligence equal to humans; it would have a self-aware consciousness that has the ability to solve problems, learn, and plan for the future.

Strong AI aims to create intelligent machines that are indistinguishable from the human mind. But just like a child, the AI machine would have to learn through input and experiences, constantly progressing and advancing its abilities over time.

While AI researchers in both academia and private sectors are invested in the creation of artificial general intelligence (AGI), it only exists today as a theoretical concept versus a tangible reality. While some individuals, like Marvin Minsky, have been quoted as being overly optimistic in what we could accomplish in a few decades in the field of AI; others would say that Strong AI systems cannot even be developed. Until

the measures of success, such as intelligence and understanding, are explicitly defined, they are correct in this belief. For now, many use the Turing test to evaluate intelligence of an AI system.

Tests of Strong AI

Turing Test

Alan Turing developed the Turing Test in 1950 and discussed it in his paper, [“Computing Machinery and Intelligence”](#) (PDF, 566 KB) (link resides outside IBM). Originally known as the Imitation Game, the test evaluates if a machine’s behavior can be distinguished from a human. In this test, there is a person known as the “interrogator” who seeks to identify a difference between computer-generated output and human-generated ones through a series of questions. If the interrogator cannot reliably discern the machines from human subjects, the machine passes the test. However, if the evaluator can identify the human responses correctly, then this eliminates the machine from being categorized as intelligent.

While there are no set evaluation guidelines for the Turing Test, Turing did specify that a human evaluator will only have a 70% chance of correctly predicting a human vs computer-generated conversation after 5 minutes. The Turing Test introduced general acceptance around the idea of machine intelligence. However, the original Turing Test only tests for one skill set — text output or chess as examples. Strong AI needs to perform a variety of tasks equally well, leading to the development of the Extended Turing Test. This test evaluates textual, visual, and auditory performance of the AI and compares it to human-generated output. This version is used in the famous Loebner Prize competition, where a human judge guesses whether the output was created by a human or a computer.

Chinese Room Argument (CRA)

The Chinese Room Argument was created by John Searle in 1980. In his paper, he discusses the definition of understanding and thinking, asserting that computers would never be able to do this. In this excerpt from his paper, from [Stanford’s website](#) (link resides outside IBM), summarizes his argument well, “Computation is defined purely formally or syntactically, whereas minds have actual mental or semantic contents, and we cannot get from syntactical to the semantic just by having the syntactical operations and nothing else...A system, me, for example, would not acquire an understanding of Chinese just by going through the steps of a computer program that simulated the behavior of a Chinese speaker (p.17).”

The Chinese Room Argument proposes the following scenario:

Imagine a person, who does not speak Chinese, sits in a closed room. In the room, there is a book with Chinese language rules, phrases and instructions. Another person, who is fluent in Chinese, passes

notes written in Chinese into the room. With the help of the language phrasebook, the person inside the room can select the appropriate response and pass it back to the Chinese speaker.

While the person inside the room was able to provide the correct response using a language phrasebook, he or she still does not speak or understand Chinese; it was just a simulation of understanding through matching question or statements with appropriate responses. Searle argues that Strong AI would require an actual mind to have consciousness or understanding. The Chinese Room Argument illustrates the flaws in the Turing Test, demonstrating differences in definitions of [artificial intelligence](#).

Strong AI vs. weak AI

Weak AI, also known as narrow AI, focuses on performing a specific task, such as answering questions based on user input or playing chess. It can perform one type of task, but not both, whereas Strong AI can perform a variety of functions, eventually teaching itself to solve for new problems. Weak AI relies on human interference to define the parameters of its learning algorithms and to provide the relevant training data to ensure accuracy. While human input accelerates the growth phase of Strong AI, it is not required, and over time, it develops a human-like consciousness instead of simulating it, like Weak AI. Self-driving cars and virtual assistants, like Siri, are examples of Weak AI.

Strong AI trends

While there are no clear examples of strong artificial intelligence, the field of AI is rapidly innovating. Another AI theory has emerged, known as artificial superintelligence (ASI), super intelligence, or Super AI. This type of AI surpasses strong AI in human intelligence and ability. However, Super AI is still purely speculative as we have yet to achieve examples of Strong AI.

With that said, there are fields where AI is playing a more important role, such as:

Cybersecurity: Artificial intelligence will take over more roles in organizations' cybersecurity measures, including breach detection, monitoring, threat intelligence, incident response, and risk analysis.

Entertainment and content creation: Computer science programs are already getting better and better at producing content, whether it is copywriting, poetry, video games, or even movies. OpenAI's GPT-3 text generation AI app is already creating content that is almost impossible to distinguish from copy that was written by humans.

Behavioral recognition and prediction: Prediction algorithms will make AI stronger, ranging from applications in weather and stock market predictions to, even more interesting, predictions of human

behavior. This also raises the questions around implicit biases and ethical AI. Some AI researchers in the AI community are pushing for a set of anti-discriminatory rules, which is often associated with the hashtag #responsibleAI.

Strong AI terms and definitions

The terms artificial intelligence, machine learning and deep learning are often used in the wrong context. These terms are frequently used in describing Strong AI, and so it's worth defining each term briefly:

[Artificial intelligence](#) defined by [John McCarthy](#) (PDF, 109 KB) (link resides outside IBM), is "the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable."

[Machine learning](#) is a sub-field of artificial intelligence. Classical (non-deep) machine learning models require more human intervention to segment data into categories (i.e. through feature learning).

[Deep learning](#) is also a sub-field of machine learning, which attempts to imitate the interconnectedness of the human brain using neural networks. Its artificial neural networks are made up layers of models, which identify patterns within a given dataset. They leverage a high volume of training data to learn accurately, which subsequently demands more powerful hardware, such as GPUs or TPUs. Deep learning algorithms are most strongly associated with human-level AI.

To read more about the nuanced differences between these technologies, read "[AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?](#)"

Deep learning applications

Deep learning can handle complex problems well, and as a result, it is utilized in many innovative and emerging technologies today. Deep learning algorithms have been applied in a variety of fields. Here are some examples:

Self-driving cars: Google and Elon Musk have shown us that self-driving cars are possible. However, self-driving cars require more training data and testing due to the various activities that it needs to account for, such as giving right of way or identifying debris on the road. As the technology matures, it'll then need to get over the human hurdle of adoption as polls indicate that many drivers are not willing to use one.

Speech recognition: Speech recognition, like [AI chatbots](#) and [virtual agents](#), is a big part of natural language processing. Audio-input is much harder to process for an AI, as so many factors, such as background noise, dialects, speech impediments and other influences can make it much harder for the AI to convert the input into something the computer can work with.

Pattern recognition: The use of deep neural networks improves pattern recognition in various applications. By discovering patterns of useful data points, the AI can filter out irrelevant information, draw useful correlations and improve the efficiency of big data computation that may typically be overlooked by human beings.

Computer programming: Weak AI has seen some success in producing meaningful text, leading to advances within coding. Just recently, OpenAI released GPT-3, an open-source software that can actually write code and simple computer programs with very limited instructions, bringing automation to program development.

Image recognition: Categorizing images can be very time consuming when done manually. However, special adaptations of deep neural networks, such as DenseNet, which connects each layer to every other layer in the neural network, have made image recognition much more accurate.

Contextual recommendations: Deep learning apps can take much more context into consideration when making recommendations, including language understanding patterns and behavioral predictions.

Fact checking: The University of Waterloo recently released a tool that can detect fake news by verifying the information in articles by comparing it with other news sources.