Artificial Intelligence Basics

A Self-Teaching Introduction

N. Gupta & R. Mangla

Artificial Intelligence Basics

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher*. Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Artificial Intelligence Basics

Neeru Gupta, PhD
&
Ramita Mangla

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Copyright ©2020 by Mercury Learning And Information LLC. All rights reserved.
Reprinted and revised with permission.

Original title and copyright: *Foundation of Artificial Intelligence and Expert Systems*.
Copyright ©2019 by Trinity Press (An imprint of Laxmi Publications Pvt. Ltd. All rights reserved.)

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai
Mercury Learning And Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
1-800-232-0223

N. Gupta & R. Mangla. *Artificial Intelligence Basics*.
ISBN: 978-1-68392-516-3

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2020931301

202122321 Printed on acid-free paper in the United States of America.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at *academiccourseware.com* and other digital vendors. The sole obligation of Mercury Learning And Information to the purchaser is to replace the book, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

CONTENTS

Acknowledgments .... *ix*

1 ARTIFICIAL INTELLIGENCE (AI)........................ 1–12

1.1 Computerized Reasoning 1
1.2 Turing Test 2
1.3 What is Intelligence? 3
1.4 Artificial Intelligence 4
1.5 Goals of Artificial Intelligence 4
1.6 History of Artificial Intelligence 5
1.7 Advantages of Artificial Intelligence 7
1.8 Application Areas of Artificial Intelligence 7
1.9 Components of Artificial Intelligence 10

2 PROBLEM REPRESENTATION 13–24

2.1 Introduction 13
2.2 Problem Characteristics 13
2.3 Problem Representation in AI 14
2.4 Production System 18
2.5 Conflict Resolution 22

3 THE SEARCH PROCESS 25–42

3.1 Search Process 25
3.2 Strategies for Search 26
3.3 Search Techniques 26

4 GAME PLAYING 43–52

4.1 Game Playing 43
4.2 Game Tree 44
4.3 Components of a Game Playing Program 44
4.4 Game Playing Strategies 45
4.5 Problems in Computer Game Playing Programs 50

5 KNOWLEDGE REPRESENTATION 53–70

5.1 Introduction 53
5.2 Definition of Knowledge 53
5.3 Importance of Knowledge 56
5.4 Knowledge-based Systems 56
5.5 Differences between Knowledge-based Systems and Database Systems 56
5.6 Knowledge Representation Scheme 57

6 EXPERT SYSTEMS 71–94

6.1 Introduction 71
6.2 Definition of an Expert System 71
6.3 Characteristics of an Expert System 72
6.4 Architectures of Expert Systems 72
6.5 Expert System Life Cycle 84
6.6 Knowledge Engineering Process 86
6.7 Knowledge Acquisition 87
6.8 Difficulties in Knowledge Acquisition 87
6.9 Knowledge Acquisition Strategies 88
6.10 Advantages of Expert Systems 89
6.11 Limitations of Expert Systems 90
6.12 Examples of Expert Systems 91

7 LEARNING 95–104

7.1 Learning 95
7.2 General Model for Machine Learning Systems 95
7.3 Characteristics of Machine Learning 97
7.4 Types of Learning 97
7.5 Advantages of Machine Learning 103
7.6 Disadvantages of Machine Learning 103

8 PROLOG 105–120

8.1 Preliminaries of Prolog 105
8.2 Milestones in Prolog Language Development 106
8.3 What is a Horn Clause? 106
8.4 Robinson's Resolution Rule 107
8.5 Parts of a Prolog Program 107
8.6 Queries to a Database 108
8.7 How does Prolog Solve a Query? 109
8.8 Compound Queries 109
8.9 The _ Variable 109
8.10 Recursion in Prolog 110
8.11 Data Structures in Prolog 111
8.12 Head and Tail of a List 111
8.13 Print all the Members of the List 112
8.14 Print the List in Reverse Order 112
8.15 Appending a List 113
8.16 Find Whether the Given Item is a Member of the List 113
8.17 Finding the Length of the List 113
8.18 Controlling Execution in Prolog 113
8.19 About Turbo Prolog 117

9 PYTHON 121–138

9.1 Languages Used for Building AI 121
9.2 Why Do People Choose Python? 121
9.3 Build AI Using Python 122
9.4 Running Python 124
9.5 Pitfalls 125

9.6 Features of Python 125
9.7 Useful Libraries 132
9.8 Utilities 134
9.9 Testing Code 137

10 ARTIFICIAL INTELLIGENCE MACHINES AND ROBOTICS 139–186

10.0 Introduction 139
10.1 History: Serving, Emulating, Enhancing, and Replacing Man 143
10.2 Technical Issues 160
10.3 Applications: Robotics in the Twenty-First Century 170
10.4 Summary 182

REVIEW QUESTIONS 187–192

INDEX 193

ACKNOWLEDGMENTS

While writing this book, many of our nears-and-dears helped us with their sincere suggestions and comments. We are thankful to them. We also express our gratitude to the Tata McGraw-Hills, Roger Schank, University of Texas, Rakesh Sharma, Intelligence Information Technology and Services. Besides, we also extend our thanks to the Dan W. Patterson, Electrical Engineering and Computer Science Department of the University of Texas at El Paso for the unrestrained use of their facilities.

Creative suggestions for improvement of the book shall be acknowledged gratefully.

—Authors

CHAPTER 1

ARTIFICIAL INTELLIGENCE (AI)

1.1 Computerized Reasoning

From the ancient times, human beings have tried to get their work done by using human strength or inanimate machines. One characteristic that is constant in the software industry today is that of “change.” Change is one the most critical aspects of software development and management. New tools and new approaches are announced almost every day. Thanks to the industrial revolution, many activities are carried out by machines. With the invention of digital computers, error-prone numerical problems and time-consuming tasks are done with accuracy and relative ease. Then, it struck people: “Why not seek the help of computers in the reasoning process?” How computers can help with the reasoning process was explained by Alan Turing. Before we explain the Turing Test, let’s look at Table 1.1, which presents the major differences between humans and computers.

Table 1.1 Differences between humans and computers

Humans	Computers
Have emotions	“Dumb” and have no emotions
Have a continuous nature	Discrete in nature
Have the capacity to learn	Must be programmed
Limited memory size	Unlimited memory size
Storage “devices” are electrochemical	Storage devices are electronic
in nature	and magnetic
Living “devices”	Non-living devices
Use Fuzzy logic	Use binary logic for computation

1.2 Turing Test

In 1950, Alan Turing wrote an article in *Mind Magazine* that considered the question "Can a machine think?" This article began the discussion that has become part of the philosophy of Artificial Intelligence (AI).

"Thinking" is difficult to define. There are two kinds of questions that philosophers have addressed:

a. Can a machine be intelligent? Can it solve all the problems that a human can solve by using intelligence?

b. Can a machine be built with a mind and experience a subjective consciousness [a quality of awareness]?

In the *Mind Magazine* article, Turing proposed the "imitation game," which was later known as the "Turing Test." The Turing Test measures the performance of a supposedly intelligent machine against that of human being. Turing's imitation game places a human and a machine counterpart in rooms apart from a second human being, who is referred to as the interrogator. A diagrammatic representation of the Turing Test is given in Figure 1.1.

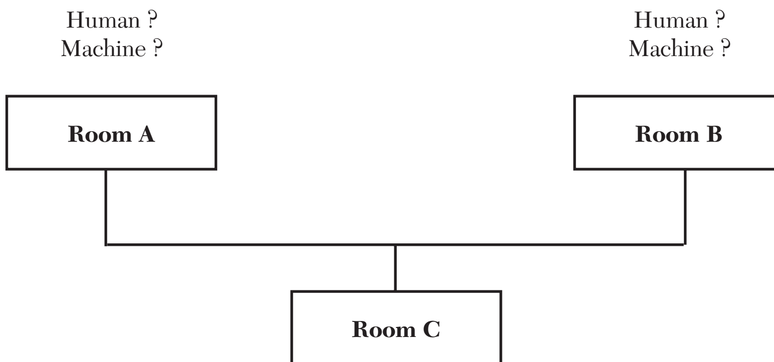


FIGURE 1.1 Representation of Turing Test.

Turing proposed that if the human interrogator in Room C is not able to identify who is in Room A or in Room B, then the machine possesses intelligence. Turing considered this test as sufficient for attributing thinking capacity to a machine.

However, the test is not easy as it seems. Humans are far superior to machines in regards to creativity, common sense, and reasoning. So humans

are sure to excel in these areas while machines are faster and more accurate for numerical computations. Machines can never be wrong in their computations, whereas there exists a probability that humans may give incorrect answers, even after taking a long time. Therefore, Turing argued the machine may be assumed to be intelligent.

1.2.1 Weakness of the Turing Test

Turing did not explicitly state that the Turing Test could be used as a measure of intelligence or any other human quality. However, the Turing Test has come under severe criticism because it has been proposed as a measure of a machine's "ability to think" or its "intelligence." This proposal has received criticism from both philosophers and computer scientists. It assumes that an interrogator can determine if a machine is "thinking" by comparing its behavior with human behavior. Every element of this assumption has been questioned, including the value of comparing only behavior and the value of comparing a machine with a human. The reliability of the interrogator's judgment has also been part of the discussion.

Because of these considerations, some AI researchers have questioned the relevance of the test, which brings us to the verge of a major question: "What is intelligence?"

1.3 What is Intelligence?

A typical definition of intelligence is "the ability to acquire and apply knowledge." Intelligence includes the ability to benefit from past experience, act purposefully to solve problems, and adapt to new situations.

1.3.1 Types of Intelligence

In the 1980s and 1990s, psychologist Howard Gardner proposed the idea of eight kinds of intelligence, which are relatively independent of one another. These eight types of intelligence are:

1. **Linguistic:** Spoken and written language skills
2. **Logical-Mathematical:** Number skills
3. **Musical:** Performance or composition skills
4. **Spatial:** Ability to evaluate and analyze the visual world
5. **Bodily-Kinesthetic:** Dance or athletic ability

6. **Interpersonal:** Skills in understanding and relating to others
7. **Intrapersonal:** Skills in understanding the self
8. **Nature:** Skills in understanding the natural world.

In the 1980s and 1990s, Robert Sternberg proposed the Triarchic Theory of Intelligence that distinguished among three aspects of intelligence:

Componential Intelligence: The ability assessed by intelligence tests;

Experimental Intelligence: The ability to adapt to new situations and produce new ideas;

Contextual Intelligence: The ability to function effectively in daily situations.

1.4 Artificial Intelligence

The term Artificial Intelligence (AI) was coined by John McCarthy in 1956. Numerous definitions for AI have been proposed by scientists and researchers such as:

Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better;

Artificial Intelligence is a part of computer science that is concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior;

Artificial Intelligence is the branch of computer science that deals with the way of representing knowledge using symbols rather than numbers and with rules-of-thumb or heuristic methods for processing information.

1.5 Goals of Artificial Intelligence

The goals of AI are as follows:

To create expert systems: These are systems that exhibit intelligent behavior, learn, demonstrate, explain, and advise users.

To implement human intelligence in machines: Researchers want to create systems that understand, think, learn, and behave like humans.

1.6 History of Artificial Intelligence

1941: The electronic computer was first developed in 1941, but evidence of AI can be traced back to ancient Egypt and Greece. Eventually, the technology become available to create machine intelligence.

1949: The stored computer program made the job of entering a program easier, and advancements in computer theory led to the creation of the fields of computer science and AI.

1950: Alan Turing proposed the Turing Test. Turing considered this as a sufficient test for attributing thinking capacity to a machine.

1955: Newell and Simon developed the Logic Theorist. The program represented each problem as a tree model, which it would attempt to solve by selecting the branch that would most likely result in the correct conclusion.

1956: The field of AI research was founded at a conference on the campus of Dartmouth College in the summer of 1956. The father of AI is John McCarthy. He wrote programs that solved word problems in algebra, proved logical theorems, and used English words.

1958: John McCarthy announced his new development, the LISP language. LISP stands for List Processing and was soon adopted as the language of choice among most AI developers.

Knowledge Expansion: The next few years would later be called an "AI winter," a period when funding for AI projects was hard to find.

1968: A multitude of programs, notably the SHRDLU and part of the MicroWorlds project, which consisted of research and programming using small words, were launched.

1970: In 1970, AI achieved commercial success with the "expert system," a form of AI program that simulated the knowledge and analytical skills of one or more human experts. Another development during this time was the Prolog language.

1985: By 1985, the market for AI had reached over a billion dollars. At the same time, Japan's fifth generation computer project inspired the U.S. and British governments to restore funding for academic research in the field.

1990: In the 1990s and early 21st century, AI achieved its greatest successes. Now AI is used for logistics, data mining, medical diagnosis, and many other areas throughout the technology industry.

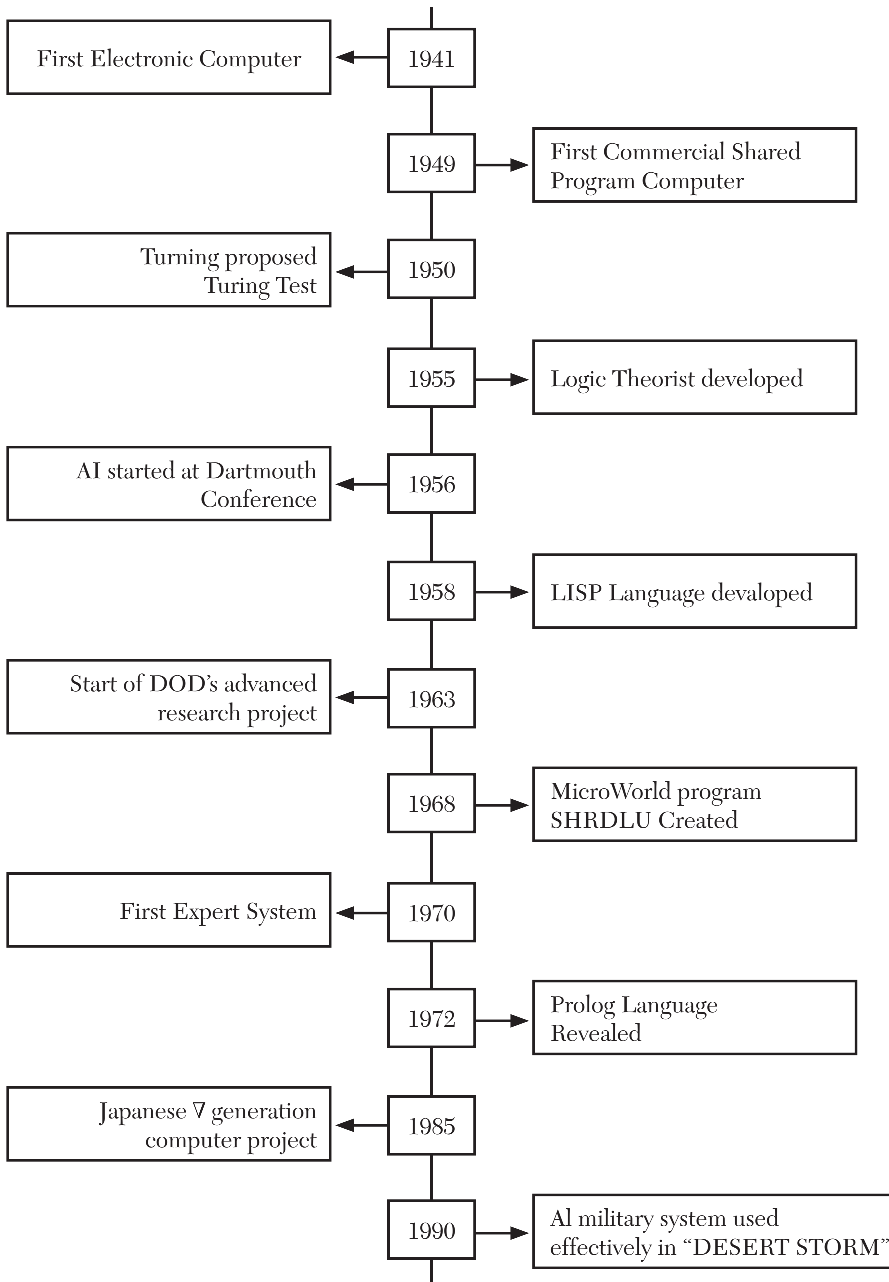


FIGURE 1.2 A brief history of Artificial Intelligence.

1.7 Advantages of Artificial Intelligence

It is evident that artificial intelligence will have to fill the gaps of human knowledge, and it will make man's work easier. Some advantages of AI are given below:

1. AI machines do not get sick. There is no need for sleep or breaks. AI can go, go, go. AI machines can definitely get a lot more work done than people can. Take the finance industry, for example, where there are numerous stories showing the value of AI.
2. AI techniques play a major role in science and medicine. AI methods have been employed recently to discover subtle interactions between medications that put patients at risk for serious side effects.
3. AI can help us to plan trips using GPS systems that rely on AI to cut through the complexity of millions of routes to find the best one to take.
4. AI can help perform calculations that are too challenging for humans.
5. AI algorithms detect faces as we take pictures with our phones and recognize the faces of individual people when we post those pictures to Facebook. With the help of AI machines, our smart phones can understand our speech.
6. AI provides accurate results with few errors and defects.
7. AI easily works in stressful and complex situations where humans may struggle or cannot accomplish the task.
8. AI can be used for longer problems, where more direct methods fail.

1.8 Application Areas of Artificial Intelligence

The concept of AI has been implemented in the following fields:

1. **Problem Solving:** This is first application area of AI research; the objective of this particular area of research is how to implement the procedures on AI Systems to solve problems like humans solve problems.
2. **Game Playing:** Much of early research in state space search was done using common board games such as checkers, chess, and 8-puzzle. The board configurations used in playing these games are easily

represented in computers, requiring no complex formalisms. For solving large and complex AI problems, many techniques, like heuristics, are required. Game playing in AI is important because:

The rules of the games are limited. Hence, extensive amounts of domain-specific knowledge are seldom needed.

Games provide a structured task where success or failure can be measured with the least effort.

Games visualize real life situations in a constricted fashion. Moreover, game playing permits one to simulate real life situations.

Unfortunately, developments in computer game playing programs are not that easy because of problems with the combinatorial explosion of solutions. In chess, the number of positions to be examined is about 35^{100}.

3. **Natural Language Processing [NLP]:** Natural language is the language of our routine; we speak it and understand it very well. The main goal of NLP is for people to ask questions to the computer in their mother tongue, and the computer will "understand" that particular language. The system will then give the response in the same language. Researchers are trying to make computers so intelligent that they can understand our natural language (such as English or any other language). NLP can be divided into two sub fields:
   a. **Natural language understanding:** NLP researchers investigate some methods of allowing the machine to improve instructions given in ordinary English so that the computers can understand people more easily.
   b. **Natural language generation:** This aims to have computers produce ordinary English so that people can understand the computers more easily.
4. **Robotics:** Robotics can be defined as the science or study of technology primarily associated with the design, fabrication, theory, and application of robots. The term "robot" is a Czech word meaning "slave."

   "Robots are machines that can be programmed to perform tasks."

   Many robots do jobs that are hazardous to people, such as defusing bombs and mines and exploring shipwrecks. Robots have electrical

components which power and control the machinery. The major components of a robot are:

Manipulator: The manipulator arm performs the job that has been assigned to it by the control unit.

Control unit: This provides the necessary control signals for activating the various parts for manipulation. It acts as an interface to various sensors which determine what the external environment is.

Power source unit: This provides the necessary energy to make the robot perform activities. Many different types of batteries can be used as a power source for robots. Designing a battery powered robot needs to take into account factors such as safety, lifetime cycle, and weight.

5. **Expert Systems:** Expert systems are one of first AI technologies to help people solve important problems, and they are very important.

   "Expert systems are comprised of knowledge based programs that can solve problems when technical expertise is required."

   Some examples of expert systems that are in use are as follows:

   MYCIN, which is used in the medical field to diagnose diseases and

   DENDRAL, which is used in life science to identify the structure of chemical molecules.

6. **Vision Systems:** These systems understand, interpret, and comprehend visual input on the computer. For example,

   A spy plane takes photographs, which are used to figure out spatial information or map the area.

   Police use computer software that can recognize the face of a criminal with a stored portrait created by a forensic artist.

7. **Speech Recognition:** Some intelligent systems are capable of hearing and comprehending language in terms of sentences and their meaning while a human talks to them. These systems can handle different accents, slang words, noise in the background, and even changes in a human's voice due to a cold.

1.9 Components of Artificial Intelligence

Any AI system consists chiefly of the following components, such as a learning AI programming language, knowledge representation, problem-solving (mainly by heuristic search), and AI hardware.

a. **Learning:** Learning means adding new knowledge to the knowledge base and improving or refining previous knowledge.

   The success of an AI program is based on the extent of knowledge it has and how frequently it acquires knowledge. Learning agents consists of four main components. They are the:

   Learning element, the part of the agent responsible for improving its performance;

   Performance element, the part that chooses the actions to take;

   Critics, which tells the learning element how the agent is doing;

   Problem generator, which suggests actions that could lead to new information experiences.

b. **AI Programming Language:** Today, just as we have specialized languages and programs for data processing and scientific applications, we have specialized languages and tools for AI programming using AI language programs and tools for the AI environment. LISP and Prolog are the primary languages used in AI programming.

   LISP (List Processing): LISP is an AI programming language developed by John McCarthy in 1950. LISP is a symbolic processing language that represents information in lists and manipulates lists to derive information.

   PROLOG (Programming in Logic): Prolog was developed by Alain Colmeraver and P. Roussel at Marseilles University in France in the early 1970's. Prolog uses the syntax of predicate logic to perform symbolic, logical computations.

c. **Knowledge Representation:** The quality of the result depends on how much knowledge the system possesses. The available knowledge must be represented in an efficient way. Hence, knowledge represen-

tation is a vital component of the system. The best known representations schemes are:

- Associative Networks or Semantic Networks
- Frames
- Conceptual Dependencies and
- Scripts

d. **Problem-solving:** The objective of this particular area of research is how to implement the procedures on AI systems to solve problems like humans do. The inference process should also be equally good to obtain satisfactory results. The inference process is broadly divided into the brute and heuristic search procedures.

e. **AI Hardware:** Today, most of the AI programs are implemented on Von Neumann machines only. However, dedicated workstations have emerged for AI programming. Computers are classified into one of following four categories:

   a) Single Instruction Single Data (SISD) Machines

   b) Single Instruction Multiple Data (SIMD) Machines

   c) Multiple Instruction Single Data (MISD) Machines

   d) Multiple Instruction Multiple Data (MIMD) Machines

In these machines, numeric computations occupy a substantial chunk of the processing time, followed by symbolic processing.

CHAPTER 2

PROBLEM REPRESENTATION

2.1 Introduction

We face so many problems in day-to-day life and want to find the solutions for them. Our goal in AI is to construct working programs that solve these problems. The steps that are required to build a system to solve a particular problem are

1. **Problem Definition:** This must include precise specifications of what the initial situation will be, as well as what final situations constitute acceptable solutions to the problem.
2. **Problem Analysis:** This can have an immense impact on the appropriateness of various possible techniques for solving the problem.
3. **Selection:** This involves choosing the best techniques for solving the particular problem.

2.2 Problem Characteristics

In order to choose the most appropriate method for a particular problem, it is necessary to check the problem in light of the following considerations:

1. Is the problem decomposable into smaller or easier sub-problems?

   A large and composite problem can be easily solved if it can be broken into smaller problems and recursion could be used. For example, let's consider $\int x^2 + 3x + \text{Sin } 2x \text{ Cos } 2x \, dx$.

   This can be done by breaking it into three problems and solving each by applying some specific rules. By adding the results of these individual solutions, the complete solution can be obtained.

2. Can the solution steps be ignored or undone?

   AI problems fall into three classes: ignorable, recoverable, and irrecoverable. These classifications are related to the steps of the solution to a problem. For example, consider the following:

   a. Theorem proving, in which solution steps can be ignored;

   b. The 8-puzzle game, in which solution steps can be undone;

   c. Chess, in which the solution steps cannot be undone.

3. AI programs use an internally consistent knowledge base.

4. In order to classify a system as an AI program, the fundamental criterion is that it must require a lot of knowledge or it uses knowledge to constrain solutions.

5. AI programs require periodic interactions between humans and computer, since the programs assist humans in making the right decisions. AI systems also have the capacity to handle uncertainty and incomplete and irrelevant information.

6. AI programs use the heuristic search to solve a large class of problems. This search includes a variety of techniques. Heuristics are also used for problems where no general algorithms are known.

7. A vital characteristic of an AI program is its ability to learn. Learning is an essential feature without which the modern systems could not have achieved their advanced technological level.

2.3 Problem Representation in AI

Before a solution can be found, the most important condition is that the problem must be very precisely defined. By defining it properly, it is easy to understand and we can abstract it into different states. These states are opened by a set of operators and the decisions of which operators should be applied. When and where are dictated by the overall control strategy. The most common methods of problem representation in AI are State Space Representation and Problem Reduction.

1. **State Space Representation:** In this method, the problem is defined in the form of states. The straight forward approach for planning an algorithm is the state space search because it takes into account

everything needed for finding a solution. Hence, "state" here means "the position at a certain time." For example, consider the following:

Water in a pan is in a liquid state, but when the stove is turned on, the state of water is changed: now it is boiled water (vapor). The state space search involves finding a path from the initial state of a search problem to a goal state. To do this, we first build a search graph starting from the initial state (or goal state). We expand the state by applying search operators to that state which generate all of its successor states. The problem must cross the following states:

a. Initial state or starting state

b. Rule applied or operator used

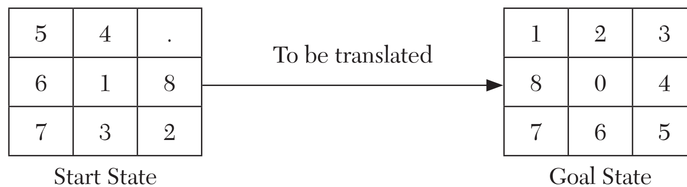
c. Goal state

When we define problems according to states, the problems become easier and more understandable because the process shows every aspect of the problem. There are two ways available to solve a state space search:

- The forward state space search, which is sometimes called *progression planning* because it moves in the forward direction; and
- The backward state space search, which is sometimes called *regression planning* because it finds the solution from the goal to the starting stage.

Here are some examples of state space representation:

A. The 8-puzzle: This is the 8-puzzle with a 3 × 3 grid with 8 consecutively numbered tiles arranged on it. Any tiles adjacent to the space can be moved. A number of different goal states are used.

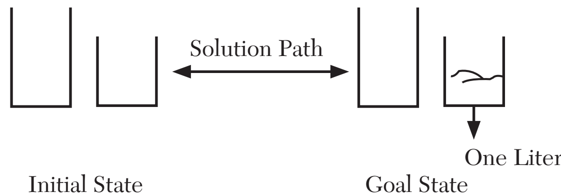


A state for this problem needs to keep track of the position of all the tiles on the game board, with 0 representing the blank position (space) on

the board. The initial state could be represented as (5, 4, 0), (6, 1, 8), (7, 3, 2). The final state could be represented as (1, 2, 3), (8, 0, 4), (7, 6, 5). The operators can be thought of in terms of direction: a blank space effectively moves up, down, left, or right.

B. **Water Jug Problem:** Imagine a 3-liter jug, a 5-liter jug, and an unlimited supply of water. The goal is to get exactly 1 liter of water into either jug. Either jug can be emptied or filled, or poured into the other.

A state in this problem could be represented with just a pair of numbers, the first representing the number of liters of water in the 5-liter (large) jug and the second representing the number of liters of water in the 3-liter (small) jug.



- The initial state would typically be (0, 0), representing the fact that both jugs start empty.
- The final state would be represented as (0, 1).
- The operators for this problem could include

  a. Fill the 5-liter jug to capacity from the water source.

  b. Fill the 3-liter jug to capacity from the water source.

  c. Empty the 5-liter jug into a drain.

  d. Pour water from the 3-liter jug into the 5-liter jug until its capacity is reached. So, there is space for 2 liters of water in the 5-liter jug.

  e. Again fill the 3-liter jug and put the water in the 5-liter jug. Now, the 5-liter jug is full, and there is exactly 1 liter of water remaining in the 3-liter jug.

  f. Empty the 5-liter jug into a drain.

  g. We have reached our goal state, i.e., (0,1).

Deficiencies of the state space representation:

a. It is not possible to display all states for a given problem.

b. This approach explores a monolithic (massive) model of the world, rather than applying a factored perspective.

c. The resources of the computer system are limited in handling this massive state space representation.

d. It is a time consuming process.

e. The program does not learn from mistakes, and hence tends to commit the same mistake repeatedly.

2. **Problem Reduction:** To overcome the problems of the state space method, the problem reduction technique is used. Problem reduction search is a basic problem-solving technique of AI involving reducing a problem to a set of easier sub-problems whose solutions, if found, can be combined to form a solution to the complex problem. Such a search is easily written as a recursive program. Problem reduction can be graphically represented with the help of AND and OR graphs or the AND-OR tree. The decomposition of the problem generates AND arcs. One AND may point to any number of successor nodes. All these must be solved so that the arc will give rise to many arcs, indicating several possible solutions.

   - The AND relationship solutions for a problem is obtained by solving all the sub-problems, like the AND gate is true if, and only if, all the inputs are true.
   - In the OR relationship, the solution for the problem is obtained by solving any of the sub-problems.

Figures 2.1 and 2.2 are helpful for understanding the AND-OR graph.

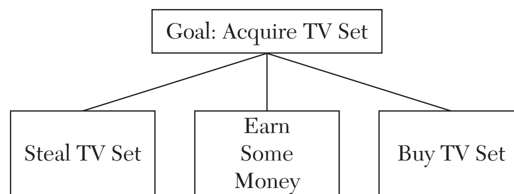


FIGURE 2.1 Problem Reduction Using the AND-OR Graph.

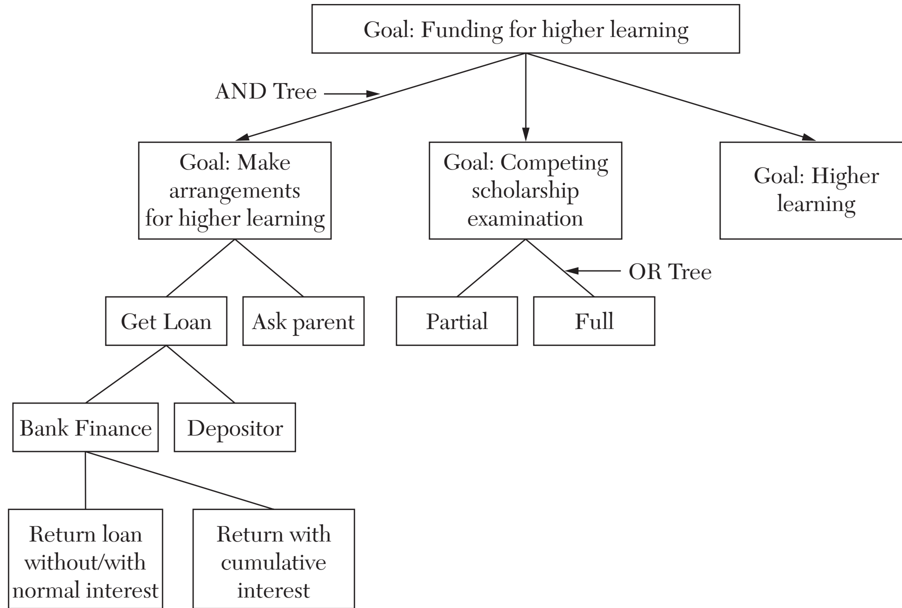


FIGURE 2.2 Problem Reduction Using the AND-OR Graph.

2.4 Production System

The production system is a mechanism that describes and performs the search process which consists primarily of a set of rules about behavior. These rules, called productions, are a basic representation found useful in automated planning, expert systems, and action selection. Production systems provide the mechanism necessary to execute productions in order to achieve some goal for the system. A production system consists of the following components:

- **A Set of Production Rules**

  A production system consists of two parts, a sensory precondition (or "IF" statement) and an action (or "THEN"). If the production system's precondition matches the current state of the world, then the production system is said to be triggered. If the production system's action is executed, it is said to have fired. So, a production system consists of a set of rules that are in the "if-then" form. That is, given a particular situation, what are the actions to be performed? For example: If it is raining, then take an umbrella.

- **Working Memory (A Global Database)**

  Working Memory (WM) is the central data structure used by an AI production system. The production rules operate using a global database. Each rule has a precondition that is either satisfied or not by the database. If the precondition is satisfied, then the rule can be applied. The application of the rule changes the database.

- **Control System**

  The control system chooses which applicable rule should be applied and ceases computation when a termination condition for the database is satisfied. If several rules are to fire at the same time, then the control system resolves the conflict.

- **A Rule Applier**

  The rule applier is the core unit of the production system. The rules are applied with the help of the rule applier.

2.4.1 Characteristics of Production Systems

We have argued that production systems are a good way to describe the operations that can be performed in a search for a solution to a problem. This leads us to several questions:

- Can production-system-like problems be described by a set of characteristics that shed some light on how they can easily be implemented?
- If so, what relationship is there between the problem types and types of production systems best suited for solving the problems?

2.4.2 Types of Production Systems

- **Monotonic Production System (MPS):** This is a system in which the application of a rule never prevents the later application of another rule that could also have applied at the time that the first rule was selected. Some production systems are monotonic, however, and only add elements to the working memory, never deleting or modifying knowledge through the action of the production rules. Such systems may be regarded as implicitly parallel. Since all rules that match will be fired regardless of which is first fired.

- **Non-Monotonic Production System (NMPS):** This is a system in which the application of a rule prevents the later application of a rule which may not have applied at the time when the first rule was selected, i.e., it is a system in which the above rule is not true and the monotonic production system in which elements may be added and deleted. The addition of new knowledge may obviate previous knowledge. The NMPS increases the significance of the conflict resolution scheme, since productions which match in one cycle may not match in the following because of the action of the intervening production.
- **Commutative Law Based Production System (CLBPS):** This is a system that satisfies both the monotonic and partially commutative conditions.
- **Partially Commutative Production System (PCPS):** This is a system with a property that if the application of those rules is allowable, it transforms state x to state y. We present two special types of production systems:
  i. Commutative Production System
  ii. Decomposable Production System

The special features of these production systems are outlined below.

i. **Commutative Production System:**

A production system is called commutative if, for a given a set of rules (R) and a working memory (WM), the following conditions are satisfied:

- **Freedom in the orderliness of rule firing:** The arbitrary order of the filing of the applicable rules selected from Set S will not make a difference in the content of the working memory. In other words, the working memory that results due to an application of a sequence of rules from Set S is invariant under the permutation of the sequence.
- **Invariance of the precondition of attaining the goal:** If the precondition of a goal is satisfied by the working memory before the firing of a rule, then it should remain satisfiable after the firing of the rule.
- **Independence of the rules:** The "firability" condition of a yet-unfired Rule R, with respect to the working memory remains unaltered, even after firing Rule R, for any condition.

Significance of the Commutative Production System

1. The rule can be fired in any order without having the risk of losing the goal, in case it is attainable.
2. An irrevocable control strategy can be designed for such systems, as an application of a rule to the WM never needs to be undone.

ii. **Decomposable Production System**

The commutative property of the production system facilitates a limited degree of flexibility in the sequence in which the applicable rules are fired. The decomposability property of a production system allows for some freedom in the ordering of rule application. A production system is called decomposable if the goal, G, and working memory, WM, can be partitioned into G_i and WM_i such that

$G = AND_i\ (G_i)$

$WM = U\ \{WM_i\}$

The rules are applied to each WM_i independently or concurrently to yield G_i.

Significance of the Decomposable Production System

- Decomposition allows the parallel firing of rules, without causing a difference in the context of the working memory.
- Decomposable production systems have been successfully used for the evaluation of symbolic integration. Here, an integral can be expressed as a sum of more than one integral, all of which can be executed independently.

2.4.3. Advantages of Production Systems

1. Production systems are highly modular because the individual rules can be added, removed, or modified independently.
2. The production rules are expressed in a natural form, so the statements contained in the knowledge base should be like a recording of an expert thinking out loud.
3. Production systems provide an excellent tool for structuring AI programs.

4. An important aspect of the production system model is the lack of any syntactic interactions between production rules. The syntactic independence supports the incremental development of expert systems by successively adding, deleting, or changing the knowledge (rules) of the system.

5. The production system is an elegant model of the separation of knowledge and the control system in a computer program. The advantage of this separation includes the ease of modifying the knowledge base without requiring a change in the code for program control.

6. One of the advantages of the production system is that the computational complexity of the matcher, while large, is deterministically finite and the conflict resolution scheme is trivial.

2.4.4. Limitations of Production Systems

1. Production rules lack expressive power for describing situations; while procedural knowledge can be implemented, it is not that easy to make use of the production rules for descriptions.

2. When the number of rules is large, it becomes difficult to check whether a new rule brought into the system is redundant or in conflict with the existing ones.

3. One important disadvantage is the fact that it may be very difficult to analyze the flow of control within a production system because the individual rules do not call each other.

2.5 Conflict Resolution

Conflict resolution is used in a production system to help in choosing which production rule to fire. The need for such a strategy arises when there is more than one rule that can be fired in a situation. The rule interpreter decides which rule to fire, what is the order of triggering, and whether to apply all rules that are applicable or to be selective about the rules.

Most conflict resolution schemes are very simple and are dependent on the number of conditions in the production or the time stamps (ages) of

the elements to which the conditions matched, or they may be completely random. Conflict resolution strategies fall into four main categories:

1. **Specificity:**

   If all of the conditions of two or more rules are satisfied, choose the rule with most specific condition. This is also referred to as the "degree of specialization."

   For example, consider the following two rules:

   a. "It is hot and smokey."

   b. "It is hot."

   The first rule (a) is more specific than the second rule (b). We choose the specific rule for the current situation.

2. **Recency:**

   Facts are usually tagged to show how recently they were added. It is generally believed that a newly-added rule contains more information than the existing ones. When two or more rules could be chosen, the system favors the one that matches the most currently relevant facts. However, there is a small challenge with using this strategy. The system has to keep track of which rule came in at what time and which rules were modified.

3. **Refraction:**

   Refraction specifies that once a rule has fired, it may not fire again until the working memory elements that match its conditions have been modified. This helps the system avoid entering infinite loops.

4. **Order:**

   Pick the first applicable rule in the order of presentation. This is the type of strategy that Prolog uses, and it is one of the most common ones.

CHAPTER 3

THE SEARCH PROCESS

3.1 Search Process

Searching is defined as a sequence of steps that transforms the initial state to the goal state. To perform a search, the following steps are needed:

- Initial state (I)
- Goal state (G)
- A set of legal operators that changes the state.

The following list shows some search terminology:

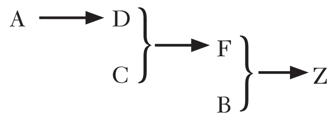
1. **Problem Space:** This is the environment in which the search takes place. It is a set of states and a set of operators to change those states.
2. **Problem Instance:** This is the initial state + goal state.
3. **Problem Space Graph:** This represents the state + problem state. States are shown by nodes, and operators are shown by edges.
4. **Depth of a Problem:** Length of the shortest path or the shortest sequence of operators from the initial state to the goal state
5. **Space Complexity:** The maximum number of nodes that are stored in the memory
6. **Time Complexity:** The maximum number of nodes that are created
7. **Admissibility:** The property of an algorithm to always find an optimal solution
8. **Branching Factor:** The average number of child nodes in a problem space graph

3.2 Strategies for Search

A search procedure must find a path between the initial and goal states. There are two directions in which a search process could proceed:

- **Forward Search:** Data-driven inference works from the initial state. By looking at the premises of the rules (IF-Part), it performs the action (THEN-Part), possibly updating the knowledge base or working memory. This continues until no more rules can be applied.

  For example:

  Disadvantage:

  Many rules may be applicable, so the whole process is not directed toward a goal.

- **Backward Search:** Goal driven inference works toward a final state by looking at the working memory to see if a goal is already there. If not, it looks at the action (THEN-Parts) of the rules that will establish the goal and sets up sub-goals for achieving the premises of the rules(IF-Part).This process continues until some rules can be applied to achieve the goal state.

  Advantage: Search is directed.

  Disadvantage: Goal has to be known.

3.3 Search Techniques

The search process in AI can be mainly classified into two types:

1. Uninformed search (also called a Blind Search or Brute Force search)
2. Informed search or heuristic search

1. **Uniformed Search:** A uniformed search algorithm does not have any domain specific knowledge. These algorithms use information like the initial state, final state, and a set of logical operators. This search should

proceed in a systematic way by exploring the nodes in a predetermined order. Uniformed search can be classified into two search technologies:

i. Depth-First Search
ii. Breadth-First Search

i. Depth-First Search: A depth-first search (DFS) is one of the main search processes. It starts off at the root of the tree and works its way down the left branch until it gets to the end. If this is not the goal state, then it backs up and tries the next branch. This continues until the goal state is reached. The algorithm tries to get as deep as possible as fast as possible. It is guaranteed to find a goal if one exists, but it does not always find the optimal path. The algorithm for the depth-first search is given in Figure 3.1.

Step 1: Put the initial node on a list (START).

Step 2: If (START is empty) or (START=GOAL), terminate the search.

Step 3: Remove the first node from START. Call this node Ⓐ.

Step 4: If (Ⓐ=GOAL), terminate the search with success.

Step 5: Else if node Ⓐ has successors, generate all of them and add them at the beginning of START.

Step 6: Go to Step 2.

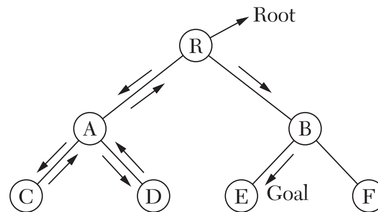


FIGURE 3.1 Algorithm for the Depth-First Search.

Look at the above tree, which has nodes starting from the root node R at the first level, A and B at the second level, and C, D, E, and F at the third level. If we want to search for node E, then the depth-first search will search for node E from left to right. First, it will check if E exists at root R. After that, it will check the nodes on the left side of the tree. Finally, it will check the nodes on the right side of the tree.

Advantages:

1. It stores only a stack of nodes on the path from the root to the current node, which is why less memory space is required.
2. If the depth-first search finds a solution without exploring much in a path, then it will use less time and space than it would otherwise.
3. The depth-first search may be useful for problems where any satisfactory solution will suffice (i.e., we are not looking for the optimal solution).

Disadvantages:

1. There is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree. This depth is called the cut-off depth. The value of the cut-off depth is essential because otherwise the search will go on and on. If the value of the cut-off is less than d, the algorithm will fail to find a solution, whereas, if the cut-off depth is greater than d, a large price is paid in terms of the execution time.
2. The depth-first search is not guaranteed to find the solution. If more than one solution exists, then the depth-first search is not guaranteed to find the minimal solution.
3. Its complexity depends on the number of paths. It cannot check duplicate nodes.

Performance of the Depth-First Search

Two important factors must be considered in any search procedure, the time complexity and space complexity.

- **Time Complexity:** The amount of time taken to generate the nodes is called the time complexity. The amount of time is proportional to the depth (d) and branching factor (the average number of child nodes for a given node) (b). The total number of nodes at level d = b^d.

  For the depth-first search, total amount of time needed is given by
  $1 + b + b^2 + \dots\dots\dots + b^{d.}$
  Thus, the time complexity = $O(b^d)$.

- **Space Complexity:** The depth-first search stores only the current path that it is pursuing. Hence, the space complexity is a linear function of the depth. Thus, the space complexity = $O(b^d)$.

ii. Breadth-First Search: The breadth-first search is another search process. It checks all of the nodes at one level starting on the left and working towards the right, before expanding the tree one level deeper. In other words, it moves back and forth through the search tree, only looking at the children of a node when all other nodes at a level have been examined. It finds the shallowest solution rather than the first solution it reaches. Therefore, it is useful when we want to find a solution with the minimum number of steps from the starting point. The algorithm for the breadth-first search is given in Figure 3.2.

Step 1: Put the initial node on a list (START).

Step 2: If (START is empty) or (START=GOAL), terminate the search.

Step 3: Remove the first node from START. Call this node Ⓐ.

Step 4: If (Ⓐ=GOAL), terminate the search with success.

Step 5: Else if node Ⓐ has successors, generate all of them and add them at the end of START.

Step 6: Go to Step 2.

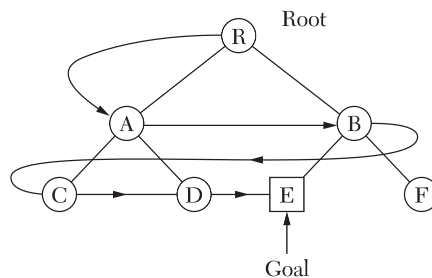


FIGURE 3.2 Algorithm for the Breadth-First Search.

Look at the above tree with nodes starting from root node R at the first level, A and B at the second level, and C, D, E, and F at the third level. If we want to search for node E, then the breadth-first search will search level by level. First, it will check if E exists at the root. Then, it will check the nodes at the second level. Finally, it will check node E at the third level.

Advantages:

1. The breadth-first search is an exhaustive search algorithm. It is simple to implement, and it can be applied to any search problem. If we

compare the breadth-first search to the depth-first search algorithm, the breadth-first search does not suffer from any potential infinite loop problem which may cause the computer to crash. So it will not go down a blind alley to find a solution.

2. If there is a solution, then the breadth-first search will definitely find it out. However, if there is more than one solution, the breadth-first search can find the minimal one that requires the smallest number of steps.

Disadvantages:

1. The main drawback of the breadth-first search is its memory requirement, since each level of the tree must be saved in order to generate the next level. The amount of memory used is proportional to the number of nodes stored. Hence, the space complexity of the breadth-first search is O (b^d). As a result, the breadth-first search is severely space-bound, so it will exhaust the memory available on a typical computer in a matter of minutes.
2. If the solution is further from the root, the breadth-first search will consume a lot of time.
3. Its complexity depends on the number of nodes.

Performance of the Breadth-First Search

Similar to the depth-first search, two important factors must be considered in the search procedure, i.e., the time complexity and space complexity.

- **Time Complexity:** This is the amount of time taken to generate the nodes. The amount of time needed is proportional to the depth (d) and branching factor (b). The total number of nodes at level d=bd. For the breadth-first search, the total amount of time needed is given by $1 + b + b^2 + b^3 + \ldots\ldots\ldots\ldots b^d$

  Hence, the time complexity = $O(b^d)$.

- **Space Complexity:** This refers to amount of memory needed. It keeps track of all children it has generated. The space complexity is also proportional to depth d and branching factor b. Thus, the space complexity becomes $1 + b + b^2 + \ldots\ldots\ b^d$. Hence, the space complexity = $O(b^d)$.

2. Informed Search or Heuristic Search

If we consider human problem solving, it is usually a combination of the depth-first search and breadth-first search. Blind searches are normally very inefficient. By adding domain-specific knowledge, we can improve the search process. The idea behind a heuristic search is that we explore the node that is most likely to be nearest to a goal state. So, heuristics are the "rules of thumb," almost like tour guides in that they are good at pointing in a general direction, but may miss certain paths. Heuristics are approximates used to reduce the search process. The following types of problems use a heuristic search:

- Problems for which no exact algorithms are known, and to find an approximate and satisfying solution. For example, speech recognition or computer vision;
- Problems for which exact solutions are known, but the computations for these problems are not feasible, e.g., a Rubik's Cube or chess.

Heuristic Function

Heuristic search uses a heuristic evaluation function which evaluates each state into numbers. On average, it improves the quality of the paths that are explored. It is a means by which humans can perform a more efficient search. A heuristic function is normally denoted h(n), that is h(n) = the estimated cost of the cheapest path from the state at node n to the goal state.

Heuristic Search Techniques

Heuristic techniques are called weak methods since they are vulnerable to the combinatorial explosion. Even then, these techniques continue to provide a framework into which domain specific knowledge can be placed. The following list includes some general purpose control strategies:

- **i.** Hill climbing
- **ii.** Best first search
- **iii.** A* Algorithm
- **iv.** AO*Algorithm
- **v.** Beam search
- **vi.** Constraint satisfaction

i. **Hill Climbing:** This is a search method for finding a maximum (or minimum) of an evaluation function. It considers the local neighborhood of a node, evaluating those nodes with the largest (or smallest) values and next examines those nodes with the largest (or smallest) values. Unlike other search strategies that use evaluation functions (like the uninformed depth-first search), hill climbing is an irrevocable scheme. It does not permit us to shift attention back to previously-suspended alternatives, even though they may have offered a better alternative than the one at hand. This property is the heart of both its computational simplicity and its shortcomings. It requires very little memory, since alternatives do not need to be retained for future consideration. However, it is not guaranteed to lead to a solution, since it can get struck on a local maximum or plateau or even wander or follow infinite uncontrolled paths, unless the guiding evaluation function is very informative. The algorithm for hill climbing is given in Figure 3.3.

Step 1: Put the initial node on a list (START).

Step 2: If (START is empty) or (START=GOAL), terminate the search.

Step 3: Remove the first node from START. Call this node **Ⓐ.**

Step 4: If (Ⓐ=GOAL), terminate the search with success.

Ste**p 5:** Else if node Ⓐ has successors, generate all of them. Find out how far they are from the goal node. Sort them by the remaining distance from the goal and add them to the beginning of START.

Step 6: Go to Step 2.

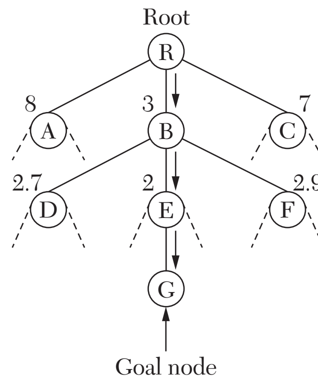


FIGURE 3.3 Algorithm for the Hill Climbing Search.

Drawbacks of the Hill Climbing Technique

This technique has three well known drawbacks:

- **Local maximum:** A local maximum is a peak that is lower than the highest peak in the state space, but it is better than all its neighbors. Once on a local maximum, hill climbing will halt, even though there is a better solution.
- **Plateau:** A plateau is an area of the state space where the evaluation function is nearly flat. Hill climbing will do a random walk in such an area.
- **Ridge:** A ridge is a curve in the search place that leads to a maximum, but the orientation of the high region (ridge) compared to the available moves that are used to climb is such that each move will lead to a smaller point. In other words, each point on a ridge looks like a local maximum, even though the point is part of a curve leading to a better optimum. So, it is an area in the path which must be traversed very carefully because movement in any direction might keep one at same level or result in a fast descent.

Figure 3.4 provides a pictorial representation of the local maximum, plateau, and the ridge.

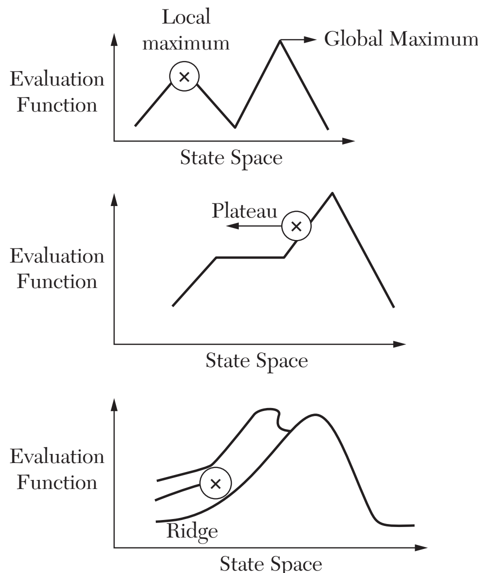


FIGURE 3.4 Problems Associated with Hill Climbing: The Local Maximum, Plateau, and Ridge.

In order to overcome these problems, we can

- Backtrack to some earlier node and try to go in a different direction.
- Make a big jump to try to get in a new section of the search space. A huge jump is recommended because in a plateau, all neighboring points have the same value.
- Move in several directions at once. This is a particularly good strategy for dealing with ridges.

Conclusion

1. Hill climbing is a local method. It decides what to do next by looking only at the "immediate" consequences of its choices.
2. Global information might be encoded in heuristic functions.
3. It can be very inefficient in a large, rough problem space.
4. Global heuristics may have to pay for the computational complexity. They are often useful when combined with other methods for getting the process started right in the correct general neighborhood.

ii. **Best-First Search:** The best-first search is another heuristic search technique and it is a way of uniting the advantages of the depth-first search and breadth-first search into a single method. One way of combining the two methods is to follow a single path at a time but switch paths whenever a rival path looks more promising. This is done through applying an appropriate heuristic evaluation function to the nodes we have generated so far. The algorithm is given in Figure 3.5.

Step 1: Put the initial node on a list (START).

Step 2: If (START is empty) or (START=GOAL), terminate the search.

Step 3: Remove the first node from START. Call this node Ⓐ.

Step 4: If (Ⓐ=GOAL), terminate the search with success.

Step 5: Else if node Ⓐ has successors, generate all of them. Find out how far they are from the goal node. Sort all the children generated so far by the remaining distance from the goal.

Step 6: Name this list as START 1.

Step 7: Replace START with START 1.

Step 8: Go to Step 2.

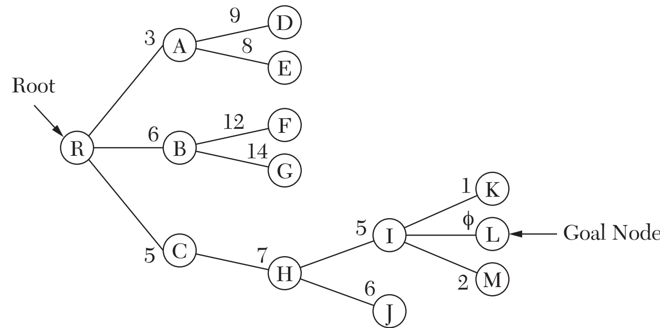


FIGURE 3.5 Algorithm for the Best-First Search.

iii. A* Algorithm: The A* Algorithm combines features of the uniform cost search and pure heuristic search to efficiently compute optimal solutions. In the best-first search, we use the evaluation function value (which estimates how far a particular node is from the goal), i.e., h(n). Apart from the evaluation function values, one can also use the cost function. The cost function indicates how much of the resources, like time, energy, and money, have been spent in reaching a particular node from the start, i.e., g(n).

So, A* Algorithm is a best-first search algorithm in which the cost associated with a node is $f(n) = g(n) + h(n)$.

The sum of the evaluation function value and cost along the path to that state is called the fitness number, i.e., f(n). Therefore, A* Algorithm guides an optimal path to a goal if the heuristic function h(n) is admissible, meaning it never overestimates the actual cost. For example, the distance a plane flies never overestimates the actual highway distance.

The algorithm is given in Figure 3.6.

Step 1: Put the initial node on a list (START).

Step 2: If (START is empty) or (START=GOAL), terminate the search.

Step 3: Remove the first node from START. Call this node Ⓐ.

Step 4: If (Ⓐ=GOAL), terminate the search with success.

Step 5: Else if node Ⓐ has successors, generate all of them. Estimate the fitness number of the successors by totaling the evaluation function value and the cost-function value. Sort the list by the fitness number.

Step 6: Name the new list as START 1.

Step 7: Replace START with START 1.

Step 8: Go to Step 2.

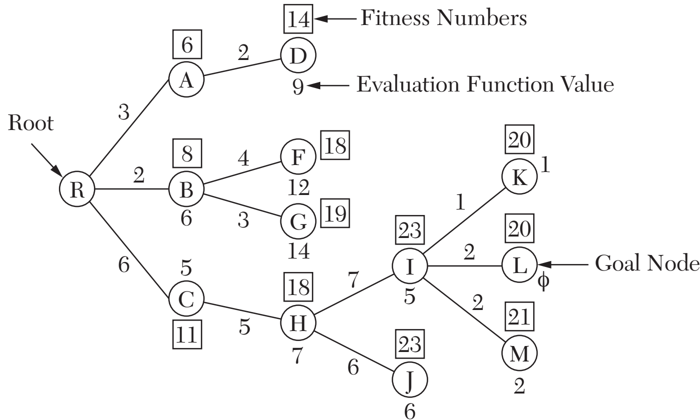


FIGURE 3.6 Algorithm for A* Algorithm.

Here, we associated each node with three numbers: The evaluation function value, the cost function value, and the fitness number.

The fitness number is the total of the evaluation function value, the cost function value. For example, for node K, the fitness number is 20, which is obtained as follows.

(Evaluation on function of K)+

(Cost function involved from start node R to node K)

= 1 + (Cost function from R to C + Cost function from C to H + Function from H + I + Cost function from I to K)

= 1 + 6 + 5 + 7 + 1

= 20

While best-first search uses the evaluation function value only for expanding the best node, A* Algorithm uses the fitness number for its computation.

iv. AO* Algorithm (Problem Reduction): When a problem can be divided into a set of sub-problems where each problem can be solved separately and a combination of these will be a solution, AND-OR graph or AND-

OR trees are used for representing the solution. The decomposition of the problem generates AND arcs. One AND arc may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. The AO* Algorithm cannot search AND-OR graphs efficiently because for the AND tree, all branches of it must be scanned to arrive at a solution. To highlight this idea, consider the small AND/Or tree shown in Figure 3.7.

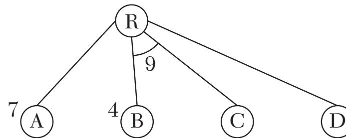


FIGURE 3.7 A Simple AND/OR Tree.

In Figure 3.7, we find the minimal is B, which has value of 4. But B is a part of the AND graph, and so we have to take into account the other branch of the AND tree. The estimate now has a value of 9. This forces us to rethink the options, and now we choose D because it has the lowest value.

The algorithm for the AO* Algorithm is given in Figure 3.8.

Step 1: Create an initial graph GRAPH with a single node NODE. Compute the evaluation function value of NODE.

Step 2: Repeat until NODE is solved or the cost reaches a very high value that cannot be expanded.

Step 2.1 Select a node NODE1 from NODE. Keep track of the path.

Step 2.2 Expand NODE1 by generating its children. For children that are not the ancestors of NODE1, evaluate the evaluation function value. If the child node is a terminal one, label it END_NODE.

Step 2.3 Generate a set of nodes DIFF_NODES having only NODE1.

Step 2.4: Repeat until DIFF_NODES is empty.

Step 2.4.1 Choose a node CHOOSE_NODE from DIFF_NODES such that none of the descendants of CHOOSE_NODE is in DIFF_NODES.

Step 2.4.2 Estimate the cost of each node emerging from CHOOSE_NODE. This cost is the total of the evaluation function value and the cost of the arc.

Step 2.4.3 Find the minimal value and mark a connector through which the minimum is achieved, overwriting the previous if it is different.

Step 2.4.4 If all the output nodes of the marked connector are marked END_NODE, label CHOOSE_NODE as OVER.

Step 2.4.5 If CHOOSE_NODE has been marked OVER or the cost has changed, add to set DIFF_NODES to all ancestors of CHOOSE_NODE.

FIGURE 3.8 The AO* Algorithm.

v. **Beam Search:** This is a search method in which heuristics are used to prune the search space to a small number of nearly optimal alternatives. This set comprises the "beam," and its members are then searched in parallel. Beam search uses the breadth-first search to build its search tree. At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of the heuristic cost. However, it only stores a predetermined number of the best states at each level (called the beam width) and only those states are expanded next.

The algorithm for the beam search is given in Figure 3.9.

Step 1: Let width_of_beam = w.

Step 2: Put the initial node on a list (START).

Step 3: If (START is empty) or (START=GOAL), terminate the search.

Step 4: Remove the first node from START. Call this node Ⓐ.

Step 5: If (Ⓐ=GOAL), terminate the search with success.

Step 6: Else if node Ⓐ has successors, generate all of them and add them at the end of START.

Step 7: Use a heuristic function to rank and sort all the elements of START.

Step 8: Determine the nodes to be expanded. The number of nodes should not be greater than w. Name these START1.

Step 9: Replace START with START1.

Step 10: Go to Step 2.

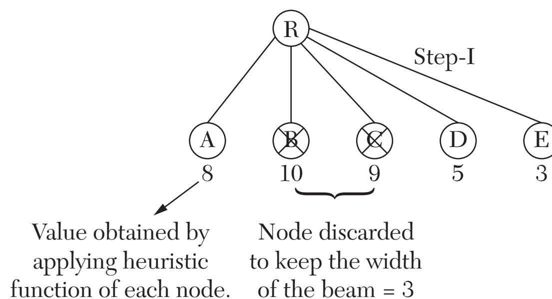


FIGURE 3.9 Algorithm for the Beam Search.

The greater the beam width, the fewer states are pruned. With an infinite beam width, no states are pruned and the beam search is identical to the breadth-first search. The beam width bounds the memory required to perform the search. The beam search is not optimal, but it returns the first solution found. Speech recognition and vision and learning applications use the beam search.

vi. Constraint Satisfaction: As can be inferred from the name, this set of problems deals with constraints. These constraints are no different from the ones that inhabit the real world. There are constraints all around us, such as temporal constraints (managing work and home life) or tangible constraints (making sure we do not go over budget), and we figure out ways to deal with them with varying degrees of success. For solving problems in this area, human beings use extensive domain-specific and heuristic knowledge. The following are examples where constraint programming has been successfully applied in various fields:

- Operations Research (scheduling, timetabling)
- Bioinformatics (DNA searches)
- Electrical Engineering (Circuit Layout)

So, a constraint satisfaction problem consists of

A set of variables: X_1, X_2...........,X_n

A set of domains: D_1, D_2............,D_n

Such that all variables X_i have a value in their respective domain D_i.

A set of constraints encompasses C_i, C_2,........C_m such that a constraint C_i restricts (imposes a constraint on) the possible values in the domain of some value in its domain so that every constraint is satisfied. Therefore, each assignment of a value to a variable must be a constraint. It must not violate any of the constraints.

Cryptarithmetic problems are typical constraint-satisfaction problems. To explain cryptarithmetic problems, consider the following example:

SEND + MORE = MONEY

Here, the constraints are

- No two digits can be assigned to the same letter. This means that only a single digit can be assigned to a letter, and all letters have different numeric values.

- Assumptions can be made at various levels so that they do not contradict each other.
- Any of the search techniques may be used.
- Backtracking may be performed as applicable for the applied search technique.
- The rules of arithmetic may be followed.

The solution is to find the value of the letters M, O, N, E, Y, S, R, and D. We consider the following:

5	4	3	2	1	column no. ____
	S	E	N	D	
+	M	O	R	E	
	c3	c2	c1		carry ____
M	O	N	E	Y	

1. From column 5, the initial guess is M = 1. Since it is the only carry-over possible, from the sum of the two single digit numbers in column 4.
2. To produce a carry-over from column 4 to column 5,
   "S + M" is at least 9 so
   "S = 8 or 9"
   "S + M" = "9 or 10," and so
   "O = 0 or 1," but "M = 1" so "O = 0"
3. If there is a carry-over from column 3 to 4, then
   "E = 9" and so "N = 0" but
   "O = 0" so there is no carry, and "S = 9" and "C 3 = 0"
4. If there is no carry-over from column 2 to column 3, then
   "E = N," which is impossible; there is a carry-over, and "N = E + 1" and "C2 =1"
5. If there is a carry-over from column 1 to column 2, then
   "N + R = E mod 10"....................a and "N = E + 1"..........................b
   So, put the value of equation b into equation a. Then,
   "E + 1 + R = E mod 10." So, "R = 9" but "s = 9" so, there must be a carry from column 1 to column 2. Therefore, "C1 = 1" and "R = 8"

6. To produce a carry-over from "C1 =1" from column 1 to column 2, we must have

"D + E = 10 + 4"

as 4 cannot be 0/1. So "D + E" is at least 12, as D is almost 7 and E is at least 5. (D cannot be 8 or 9, as it is already assigned.) N is almost 7 and "N = F + 1." So, "E = 5 or 6." Therefore, E is at least 5.

7. If E were 6 and "D + E" is at least 12, then D would be 7, but "N = E + 1," and N would also be 7, which is impossible. Therefore, "E = 5" and "N = 6"

8. "D + E" is at least 12. We then obtain "D = 7" and "Y = 2."

Then, the solution is

$$\begin{array}{r} 9\,5\,6\,7 \\ +\,1\,0\,8\,5 \\ \hline 1\,0\,6\,5\,2 \\ \hline \end{array}$$

Values:

S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2

In the AI literature, constraints satisfaction is characterized as a hill climbing technique with only a global maximum.

CHAPTER 4

GAME PLAYING

4.1 Game Playing

Game playing demonstrates several aspects of intelligence, particularly the ability to plan (at both the immediate tactical level and long-term strategic level) and the ability to learn. Successful gaming is generally deemed to require intelligence. Computers can be programmed to play games such as tic-tac-toe, checkers, and chess. The board configurations used in playing these games are easily represented in computers, requiring no complex formalisms. To solve large and complex AI problems, lots of techniques, like heuristics, are needed.

These are the following reasons for the importance of game playing in AI:

- The rules of games are limited. Hence, extensive amounts of domain-specific knowledge are seldom needed.
- Games provide a structured task where success or failure can be measured with the least amount of effort.
- Games visualize real life situations in a constricted fashion. Moreover, game playing permits the simulation of real life situations.

Unfortunately, the development of computer game programs is not that easy because of the problem of the combinatorial explosion of solutions. For example, in chess, the number of positions to be examined is about 35^{100}.

4.2 Game Tree

We can represent all possible games (of a given type) using a directed graph often called a "game tree." The nodes of the graph represent the states of the game. The arcs of the graph represent the possible moves by the players (+ and –). Consider the start of a game:

The game starts with some "start" rules and there is a set of possible moves:

m_1, m_2............., m_n

These give rise, respectively, to the states

S_1, S_2, ..., S_n

By considering the possible moves at any state S_i (recursively), we develop a game tree.

The leaves of this tree represent the state of play so far. The rules of the game assign a value to every terminal position:

W = Won,

L = Lost from the point of view of "+," and

D = Draw.

One way to guarantee a good game would be to completely analyze the game tree.

4.3 Components of a Game Playing Program

There are two major components of a game playing program: a plausible move generator and a static evaluation function generator.

- **Plausible Move Generator**

  In games where the number of legal moves is too high, it is not possible to perform a full-width search to a depth sufficient enough to have a good game. The plausible move generator is an important search alternative in such domains. It expands or generates only the selected moves. It is not possible for all moves to be examined because of the following:

1. The amount of time given for a move is limited.
2. The amount of computational power available at the disposal for examining the various states is also limited. However, further research is going on to enhance the computational power using parallel processing architectures.

- **Static Evaluation Function Generator**

  This is the most important component of a game playing program, and it is used to evaluate the positions at the leaves of the tree or every move that is being made. The static evaluation function generator occupies a crucial role in a game playing program because of the following factors:

  1. It utilizes heuristic knowledge for evaluating the static evaluation function value.
  2. The static evaluation faction generator acts like a pointer to point the way so the plausible move generator can generate future paths.

  Designing the static evaluation generator is an art. A good static evaluation generator should be very fast because it is the limiting factor in how quickly the search algorithm runs.

4.4 Game Playing Strategies

Games can be classified as either a single-person or multi-person. Games like the Rubik's Cube and 8-puzzle are single person games. For these, the search strategies such as the best-first or A* Algorithm can be used. These strategies help in identifying paths in a clear fashion.

On the other hand, in a two-person game, like chess or checkers, each player tries to outsmart the opponent. Each has their own way of evaluating the situation. Since each player tries to obtain the maximum benefits, the best-first search or A* Algorithm do not serve the purpose. The basic methods available for game playing are as follows:

1. minimax strategy
2. minimax strategy with alpha-beta cut-offs

1. Minimax Strategy

This is the most well known strategy for two player games. Here, one player is called a "maximizer" and other is called a "minimizer." The main objective of a player is to minimize the loss and maximize the profit. It is a type of mixed strategy. Both the maximizer and minimizer fight it out to see which opponent gets the minimum benefit while they get the maximum benefit.

Algorithm for the Minimax Strategy

The minimax search procedure is a depth-first, depth-limited search procedure. The idea is to start at the current position and use the plausible move generator to generate a set of possible successor positions. We apply the static evaluation function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent our evaluation of it. The starting position is exactly as good for us as the position generated by the best move we can make next. The algorithm for the minimax strategy is shown in Figure 4.1.

```
Function MINIMAX (N) is
Begin
   If N is a leaf then
      Return the estimated score of this leaf
   Else
      Let N1, N2, ........, Nn be the successor of N;
   If N is a MIN node then
      Return min{MINIMAX(N1),.....MINIMAX(Nm)}
   Else
      Return max{MINIMAX(N1),.....MINIMAX(Nm)}
   END MINIMAX;
```

FIGURE 4.1 Algorithm for Minimax.

We can explain the minimax strategy with the help of Figure 4.2. Let's assume that the maximizer will have to play first, followed by the minimizer. The search strategy here tries for only two moves, the root being M and the leaf nodes being Q, R, S, T, U, V, W, X, Y, and Z

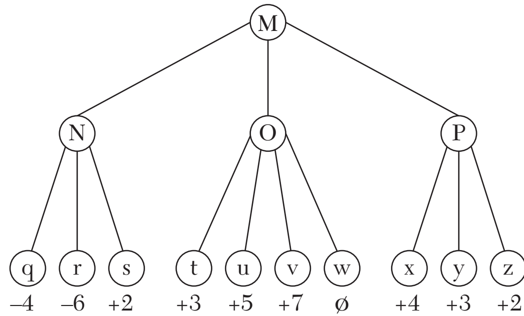


FIGURE 4.2 A Game Tree Expanded by Two Levels.

Before the maximizer moves to N, O, and P, he will think about which move would be highly beneficial to him. In order to evaluate the move, the children of the intermediate nodes N, O, and P are generated, and the static evaluation function value generator assigns values for all the leaf nodes.

If M moves to N, it is the minimizer who will have to play next. The minimizer always tries to give the minimum benefit to the other and hence he will move to R (static evaluation value = –6). This is backed up at N.

If M moves to O, then the minimizer will move to W (static evaluation function value = 0), which is the minimum of 3, +5, 7, and 0. So the value of 0 is backed up at O. On a similar line, the value that is backed up at P is 2. The tree now with backed up values is given in Figure 4.3.

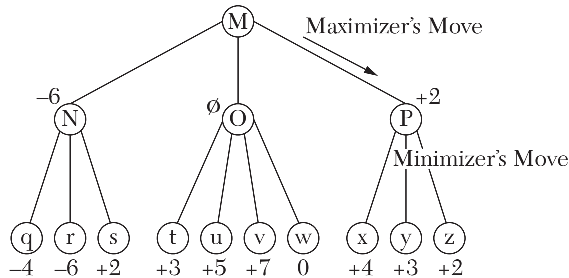


FIGURE 4.3 Maximizer's Move for the Tree Given in Figure 4.2.

The maximizer will now have to choose between N, O, or P with the values –6 and 2. Being a maximizer, he will choose node P because by doing so, he is sure of getting a value of 2, which is much better than 0 and –6.

What will be the move chosen if the minimizer has to make the first move?

Figure 4.4. shows this.

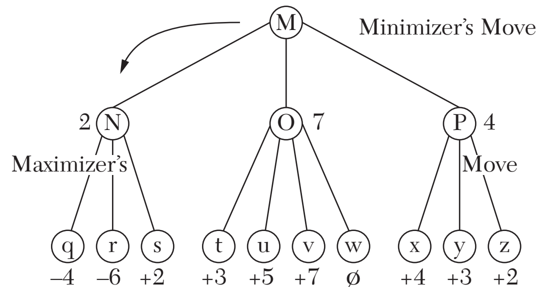


FIGURE 4.4 Maximizer's Move for the Tree Given in Figure 4.2.

This search has just stopped with two levels only. However, it is possible to consider more levels for accurate results. It depends on the following factors:

- time left forming
- the stage of the game
- number of pieces one has

2. Minimax Strategy with the Alpha-Beta Cut Off

It is necessary to modify the search procedure slightly to handle both the maximizing and minimizing players. It is also necessary to modify the branch and bound strategy to include two bounds, one for each player. This modified strategy is called "alpha-beta pruning." It requires the maintenance of two threshold values. One represents a lower bound on the value that a maximizing node may ultimately be assigned, called "alpha." The other represents the upper bound on the value that a minimizing node may be assigned, called "beta." For the MIN nodes, the score computed starts with (+) infinity and decreases with time. For the MAX nodes, the score computed starts with (-) infinity and increases with time. Alpha-beta pruning is strongly affected by the order in which branches are explored. The sooner the best moves are discovered, the sooner the worst branches can be explored.

Algorithm for the Alpha-Beta Cut Off

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and maximum score that the minimizing player is assured of, respectively. Initially, alpha is "negative" infinity and beta is "positive" infinity. As the recursion progresses, the "window" becomes smaller. When beta becomes less than alpha, the current position cannot be the result of the best play by both players, and hence, it need not be explored further.

The pseudocode for the alpha-beta algorithm is given in Figure 4.5.

```
Evaluate (node, alpha, beta)
  IF Node is a leaf
    Return the heuristic value of node
  IF node is a minimizing node
    For each child of node
      Beta =min (beta , evaluate (Child, alpha, beta))
      If beta <=alpha
      Return beta
Return beta
  If node is a maximizing node For
    each child of node
    alpha = max (alpha, evaluate (Child, alpha, beta))
    If beta <=alpha
      Return alpha
Return alpha
```

FIGURE 4.5 Pseudocode for the Alpha-Beta Cut Off.

For example,

- **MAX NODE:** This is generally drawn as a square or possibly an upward-pointing triangle, i.e., (□, △).
- **MIN NODE:** This is generally drawn as a circle or possibly downward-pointing triangle i.e., (○, ▽).

The alpha-beta values help develop the tree structure in Figure 4.6.

Alpha Max –≥ inf ≥–7

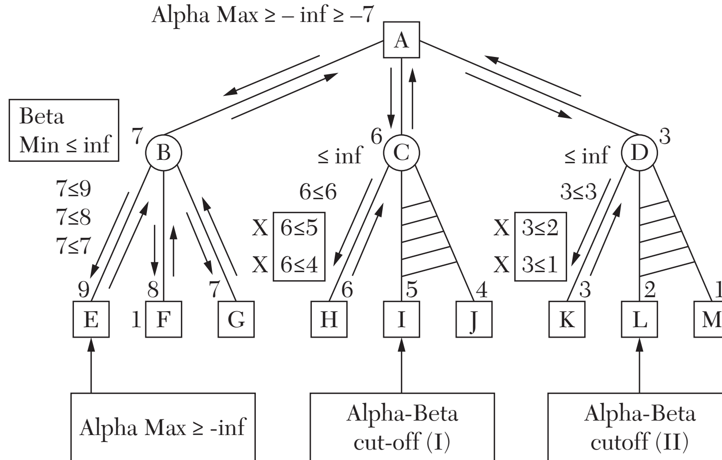


FIGURE 4.6 A Sample Tree to Explain the Alpha-Beta Search.

The maximizer has to play first, followed by the minimizer. Here, A is the maximizing player. A can branch to B, C, and D. The static evaluation function generator has assigned values which are given for the leaf nodes since E, F, G, H, I, J, K, L, and M are also maximizers.

Thus E, F, G, H, I, J, K, L, and M have the values 9, 8, 7, 6, 5, 4, 3, 2, and 1, respectively.

The preceding level (i.e., the nodes B, C, and D) is the minimizer's. Thus, B takes the minimum values of 9, 8, and 7; C takes the minimum values of 6, 5, and 4; and D takes the minimum values of 3, 2, and 1. Since A is the maximizer, then A will obviously opt for B first, then C and D.

Alpha-beta pruning is an improvement over the minimax algorithm. The problem with minimax is that the number of game states it has to examine is exponential in the number of moves. While it is impossible to eliminate the exponent completely, we are able to cut it in half.

4.5 Problems in Computer Game Playing Programs

Even though much has been said about the problem of the combinatorial explosion of solutions, computer game playing programs have still been

developed. These programs suffer from a few deficiencies, i.e., the horizon effect and optimal move question.

1. **Horizon Effect**

   This is a problem that occurs in many games where the number of possible states of positions is immense, and the computer can only feasibly search a small portion of them, typically a few levels down the game tree.

   When searching a game tree to depth n, the horizon effect occurs when the search goes to depth n+1, which would result in the evaluation of a move being drastically different. When evaluating a large game tree using techniques such as the minimax or alpha-beta pruning, the search depth is limited for feasibility reasons. However, evaluating a partial tree may give misleading results when a significant change exists over the horizon of the search depth.

   For example, in chess, assume a situation where the computer only searches the game tree to six plies (turns). From the current position, it determines that the queen is lost in the sixth ply, and suppose there is a move in the search depth where it may sacrifice a rook and the loss of the queen is pushed to the eighth ply. This is, of course, a worse move than sacrificing the queen, because it leads to losing both a queen and rook.

   However, because the loss of the queen was pushed over the horizon of the search, it is not discovered and evaluated by the search. We think "Losing the rook seems to be better than losing the queen," so the sacrifice is returned as the best option while delaying the sacrifice of queen. But the insertion of delaying moves causes an inevitable loss of material to occur beyond the program's horizon (maximum search depth). It weakens the computer's position. The effect is less apparent in a program with more knowledgeable quiescence searching. Beside an obligatory, quiescence search (the purpose of this search is to only evaluate "quiet" positions where there are no winning tactical moves to be made), extensions (especially check extensions) are designed to reduce the horizon effect.

2. **Optimal Move Question**

   The second major defect is that these programs expect the opponent to make the most optimal move, which cannot be expected in real games.

CHAPTER 5

KNOWLEDGE REPRESENTATION

5.1 Introduction

A knowledge-based management system (KBMS) is a computer system that manages the knowledge in a given domain of interest and exhibits reasoning power to the level of a human expert in this domain. AI is the part of computer science that designs intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior. Furthermore, operations in a knowledge-based system are more complex than those in a traditional database. When a rule is added, the system must check for contradictions and redundancy.

5.2 Definition of Knowledge

Knowledge can be defined as the body facts and principles accumulated by humankind or the act, fact, or state of knowing. Knowledge is having a familiarity with language, concepts, procedure, rules, ideas, abstraction, places, custom, facts, and associations, coupled with an ability to use these notions effectively in modelling different aspects of the world. Without this ability, the facts and concepts are meaningless and therefore worthless.

The meaning of knowledge is closely related to the meaning of intelligence. Intelligence requires the possession of and access to knowledge. A characteristic of intelligent people is that they possess a large amount of knowledge.

Human experts have two main types of knowledge:

1. **Domain Specific Knowledge**

   Domain specific knowledge refers to specialized knowledge required to perform a particular task. To acquire this knowledge, we have to be trained or study it.

2. **Commonsense Knowledge**

   All other pieces of knowledge that help in reasoning other than domain specific knowledge are called commonsense knowledge.

   The performance of the system increases when both types of knowledge are coupled. Associative literature has also classified knowledge as being either Declarative or Procedure.

5.2.1 Procedural Knowledge

Procedural knowledge is the compiled or processed form of information. It is related to the performance of some task. It gives knowledge/ information about how to achieve something. For example, a sequence of steps to solve a problem is procedural knowledge.

Advantages of procedural knowledge

1. **Meta knowledge:** It is knowledge about knowledge and how to gain and use pieces of information. It can be easily expressed in procedural form.
2. Procedural knowledge involves more senses, such as hands-on experience, practice at solving problems, and understanding the limitations of a specific solution. Thus, it can frequently eclipse theory.
3. Statements can be written without regard for the use that will be made of them later in the program, but in practice, the programmer will always have this in mind.

   Procedural knowledge is implemented via procedural or rule-based (production) systems.

   These are often structured as IF (condition) – THEN (action). For example, consider the following code:

```
procedure Carnivore(x);
If(x=cheetah) then return true
Else return false; END procedure Carnivore(x).
procedure sharp_teeth(x);
```

```
If Carnivore(x) then return true
Else return false END procedure sharp_teeth(x).
```

To see whether "cheetah" has sharp teeth, one should activate procedure sharp_teeth with variable x initialized to the value "cheetah." This procedure calls procedure Carnivore (x), and in turn, the value of (x = cheetah). Procedure Carnivore returns a true value and so does procedure sharp_teeth.

5.2.2 Declarative Knowledge

Passive knowledge includes statements of facts about the world. For example, the marked assignment of a student is declarative knowledge.

Declarative schemes include logic-based and relational approaches. A declarative representation declares every piece of knowledge and permits the reasoning system to use the rules of inference like modus pones, modus tollens, and the chain rule to come out with new pieces of information.

Advantages of Declarative Knowledge

1. The ability to use knowledge in ways that the system designer did not foresee.
2. A statement involving several variables needs only be written once in declarative form and can be used in different ways on different occasions according to the results sought.
3. A declarative structure is easy to modify, and new statements can be added easily.

   For example, consider the following statements:

   "All carnivores have sharp teeth" and

   "A cheetah is a carnivore."

   This can be represented using a declarative representation such as

   ```
   x (carnivore (x) —> sharp_teeth (x)
   Carnivore (cheetah)
   ```

   Using these two representations, it is possible to deduce that "A cheetah has sharp teeth."

So, it is enough that you represent the knowledge only once. In the example discussed above, the statement "`x (carnivore (x) —>`

`sharp_teeth (x)`" is made only once, and the variable x encompasses a wide variety of animals which are carnivorous in nature.

5.3 Importance of Knowledge

Intelligence requires knowledge. To exhibit intelligence, knowledge is required. Knowledge plays a major role in intelligent systems because they use the knowledge of syntax and meaning in order to understand sentences. They use knowledge to eliminate one useless or time-consuming search when solving a problem.

5.4 Knowledge-Based Systems

These are systems that use the knowledge provided to solve problems in specific domains. Much of work done in AI has been related to knowledge-based systems, including work in vision, learning, and general problem solving and natural language understanding. A knowledge-based system has two sub-systems:

- A knowledge base
- An inference engine

A knowledge base represents the facts about the world. The inference engine represents the logical assertions and conditions about the world, usually represented via IF-THEN rules.

A knowledge-based system may also incorporate an explanation facility so that user can determine whether the reasoning used by the system is consistent and complete. The reasoning facility also offers a form of tutoring to the uninitiated user.

5.5 Differences Between Knowledge-Based Systems and Database Systems

Knowledge Base	Database
It is any collection of information. It is used very broadly.	It is a collection of data organized in some form. A database is a software program that is used to create tables, queries, and views.

Knowledge Base	Database
It is significantly smaller than a database, and we change the knowledge base gradually.	It contains a large volume of data, and the facts change over time.
Knowledge-based systems are far more complex and require far greater computing capabilities. They essentially represent the attainment of artificial intelligence.	Databases are structured according to specific requirements, and enable users to access the desired information quickly and efficiently.
Updation is performed by domain experts.	Updation is performed by clerical personnel.
The demands of knowledge-based systems can be formidable.	Databases are expanded through the continuous inputting of names, places, and data. Any conclusions drawn on the basis of the searches of those databases are entirely dependent upon the skills and knowledge level of the people exploiting the data.
It operates on a class of objects.	It operates on single objects.
A DBMS provides the user with an integrated language, which serves the purpose of the traditional DML of the existing DBMS and has the power of a high-level application language.	A database can be viewed as a very basic knowledge-based system in so far as it manages facts. There will be a continuing need for a current DBMS and functionalities that co-exist with an integrated KBMS.

5.6 Knowledge Representation Scheme

A knowledge representation scheme is a set of syntactic and semantic conventions used to describe various objects. The syntax and arrangements of symbols form expressions. Mylopoulos and Levesque classified the schemes into four categories:

- **Logical Representation Scheme:** For example, first order predict logic [FOPL]
- **Procedural Representation Scheme:** For example, production rules
- **Network Representation Scheme:** For example, semantic networks
- **Structured Representation Scheme:** For example, scripts, frames, and conceptual dependencies.

A knowledge representation system should provide ways of representing complex knowledge. So, in this chapter, we discuss some of the widely known representation schemes. They are:

1. Semantic Networks or Associative Networks
2. Frames
3. Conceptual Dependency
4. Scripts

1. Semantic Networks of Associative Networks

A semantic network is a structure of representing knowledge as a pattern of interconnected nodes and arcs. It is also defined as a graphical representation of knowledge. The objects under consideration serve as nodes and the relationships with other nodes give the arcs. In a semantic network, information is represented as a set of nodes connected to each other by a set of labelled ones, which represent the relationship among the nodes. The network can include many different types of relationships. For example, "Is- a," "form," "has-attribute," "used-for," "adjacent-to," and "has-value."

The following are the rules about how nodes and arcs are applied in associative networks:

- Node in semantic network can be:
    - **a.** States
    - **b.** Attributes
    - **c.** Events
- Arcs in the network give the relationship between the nodes and the labels on the arcs show what type of relationship exists. Using a simple semantic network, it is possible to add more knowledge by linking other objects with different relationships. A simple semantic network is shown in Figure 5.1.

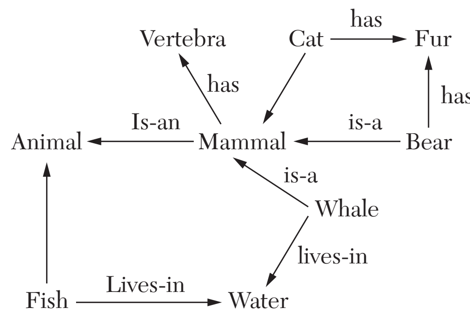
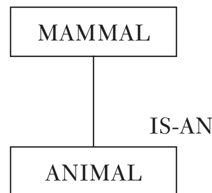


FIGURE 5.1 A Sample of a Semantic Network.

From this, it is possible for us to say that “Mammal” is an animal, and it is a “Fish” that lives in water. “Mammal” is also a “Bear” that “has Fur.”

Such a semantic network not only gives details about an object under consideration, but also provides facilities to represent variables. The semantic net shown in the above figure, cannot be represented like this on a computer. Every pair and its link are stored separately.

For example, IS-AN in Prolog represents the following:



The figure above is a One-Way Link Representation.

As with a knowledge representation scheme, the problem-solving power comes from the ability of the program. Inter-section search is used to find the relationship between two objects. But a major hurdle in utilizing semantic networks is that there is no standardization and formalization as far as notations and reasoning are concerned. However, the overall concepts of arcs and nodes in semantic networks have been standardized.

2. Frames

A principle for the large-scale organization of knowledge introduced by Minsky, originally in connection with vision, but more generally applicable, is called “frames.” Frames may be arbitrarily complex and have procedures attached to the slots. The default values for the slots are helpful when the frames are used in the absence of the full instantiation data. The character of frames suggests a hierarchical organization of sets of frames, but the non-hierarchical filling of one frame slot by another is possible. A frame is defined as a combination of declarative and operational knowledge. A frame is a data structure that has slots for various objects and a collection of frames consists of the expectations for a given situation. A frame structure provides facilities for describing objects, facts about situations, and procedures on what to do when a situation is encountered. Frames are also useful for representing commonsense knowledge. An example is presented below:

Slots	Fillers
Name:	CHAIR
Is-A:	FURNITURE
Color:	BROWN
MADE-OF:	WOOD
LEGS:	4
ARMS:	DEFAULT: 0
PRICE:	100

Types of Frames

There are two types of frames, i.e., the Declarative/Factual Frame and procedural frame.

Declarative Frame

A frame that merely contains a description about an object is called a Declarative Type/Factual/Situation frame (Figure 5.2).

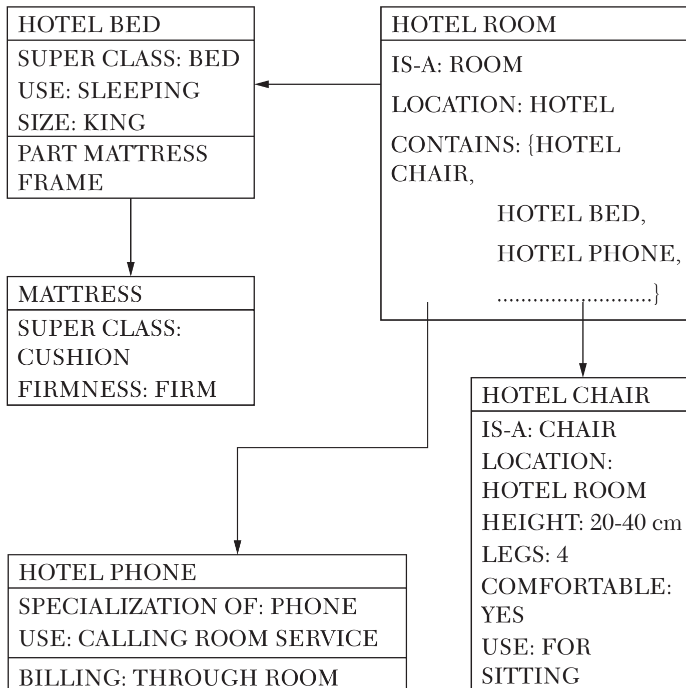


FIGURE 5.2 Frame Description of a Hotel Room.

Minsky (1975) developed the original idea of frames and defined them as "data-structures for representing stereotyped situations," such as going to a hotel room.

Procedural Frame

Apart from the declarative part in a frame, it is also possible to attach slots that explain how to perform things. In other words, it is possible to have procedural knowledge represented in a frame. Frames with procedural knowledge embedded in them are called "action procedure frames." The action frame has the following slots:

- **Objects Slots:** This frame has information about the item to be operated in.
- **Actor Slot:** This frame has information about who is performing the activity.
- **Source Slot:** This frame has information about where the action has to begin.
- **Destination Slot:** This frame has information about where the action has to end.
- **Task Slot:** This frame generates the sub-frame required for performing the operation.

With the help of Figure 5.3, we can clearly understand the procedure of fixing a punctured tire on a scooter.

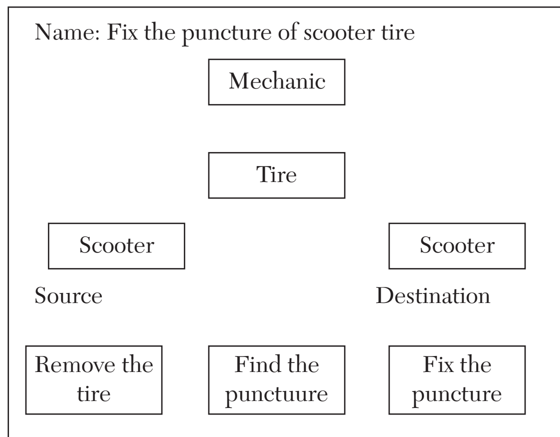


FIGURE 5.3 Procedural Frame.

Advantages of Frames

1. Frames add power and clarity to the semantic net by allowing complex objects to be represented as a single frame.
2. Frames provide an easier framework than semantic nets to organize information hierarchically.
3. Frames allow for procedural attachment, such as if-needed, if-deleted, and if-added, which run a demon (piece of code) as a result of another action in the knowledge base.
4. Frames support the class of inheritance

3. Conceptual Dependency

Conceptual dependency was originally developed to represent knowledge acquired from natural language input. The goals of this theory are as follows:

- To construct a computer program that can understand natural language
- To help in drawing inferences from sentences and also identify conditions in which sentences can have a similar meaning
- To be independent of words used in the original input. That is to say, for any two (or more) sentences that are identical in meaning, there should be only one representation of that meaning
- To provide a necessary platform so that sentences in one language can be easily translated into another language.

Conceptual dependency has been used by many programs that portend to understand English (such as MARGIE, PAM, and SAM).

There is a set of allowable dependencies among conceptualizations described in a sentence:

1. **ACTS (Actions):** These are equivalent to verbs or group of verbs.
2. **PPs (Picture Producer):** These are equivalent to nouns.
3. **AAs (Action Aider):** These are modifiers of actions (acts) and thus are equivalent to adverbs.

4. **PAs (Picture Aider):** These are modifiers of PPs and thus are equivalent to adjectives.
5. **Conceptual Cases:** There are different types of conceptual cases:
    - Objective case (o)
    - Directive case (d)
    - Instrumental case (i)
    - Recipient case (r)
6. **Conceptual Tenses:** Schank proposed a list of attachments to the relationship. A partial list of these is as follows:
    - Continuing (k)
    - Future (f)
    - Interrogative (?)
    - Past (p)
    - Present (nil)
    - Transition (t)
    - Transition start (t_s)
    - Transition finished (t_f)
    - Negative (/)
    - Conditional (c)
7. **Conceptual Dependencies:** These provide a structure into which nodes representing information can be placed with a specific set of primitives at a given level of granularity. The following are the conceptual dependency primitives:
    a. **PTRANS:** Physical transfer of location of an object [e.g., GO]
    Slots for PTRANS are:
        - ACTOR: A HUMAN (or animate object) that initiates the PTRANS;

- OBJECT: A physical object that is PTRANSed;
- FROM: A LOCATION at which PTRANS begins;
- TO: A LOCATION at which PTRANS ends.

b. **ATRANS:** Abstract transfer of ownership possession or control of an object (e.g., give)

c. **MTRANS:** Mental transfer of information between agents (e.g., tell)

d. **MBUILD:** Mental construction of a thought or new information between agents (e.g., decide)

e. **ATTEND:** Act of focusing attention of a sense organ toward an object (e.g., listen)

f. **GRASP:** Grasping of an object by an actor for manipulation (e.g., hold)

g. **PROPEL:** The application of physical force to an agent by that agent (e.g., throw)

h. **MOVE:** The movement of a body part of an agent by that agent (e.g., kick)

i. **INGEST:** Taking of an object (such as food, air, or water) by an animal (e.g., drink or eat)

j. **EXPEL:** The expulsion of an object by an animal (e.g., spit)

k. **SPEAK:** The act of producing sound, including non-communicative sounds.

There are semantic rules for the formation of the dependency structure:

Rule 1: PP ⟺ ACT

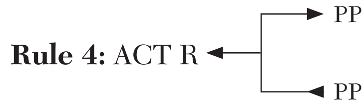
Indicates that an actor acts

Rule 2: PA ⟺ PA

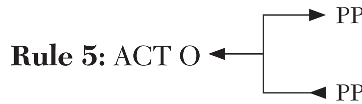
Indicates the object has certain attributes

Rule 3: ACT O ⟵ PP

Indicates the object of an action



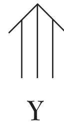
Indicates the recipient and the donor of an object within an action



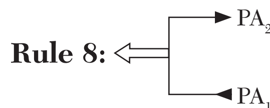
Indicates the direction of an object within an action

Indicates the instrument conceptualization of an action

Rule 7: X



Indicates the conceptualization where X caused the conceptualization Y; when written with a C, this form denotes that X COULD cause Y.



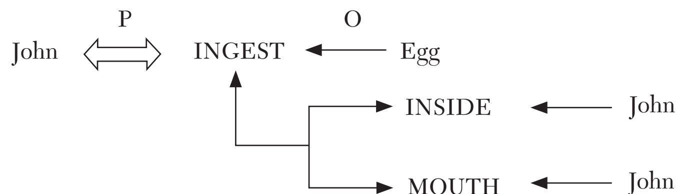
Indicates a state change of an object

Rule 9: $PP_1 \longleftarrow PP_2$

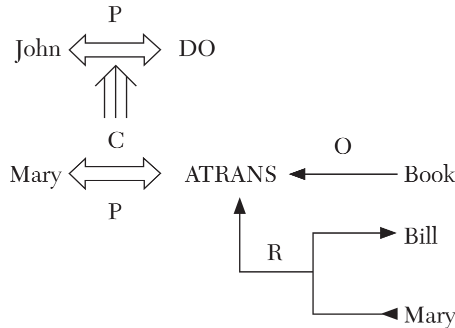
Indicates the PP is either PART-OF or the POSSESSOR of PP1

For example,

1. John ate the egg.



2. John prevented Mary from giving a book to Bill.



3. A boy is nice.

boy ⟵ nice

4. John pushed the bike.

John ⟺ PROPEL ⟵O— bike

Advantages of the Conceptual Dependency

1. Using these primitives involves fewer inference rules.
2. Many inference rules are already represented in the conceptual dependency structure.
3. The holes in the initial structure help to focus on the points still to be established.

4. Scripts

A script is a structured representation describing a stereotyped sequence of events in a particular context. Scripts are used in natural language understanding systems to organize a knowledge base using the terms of the situations that the system should understand. It could be considered to consist of a number of slots or frames, but with more specialized roles. The components of scripts include:

Entry conditions: These must be satisfied before events in the scripts can occur.

Results: Conditions that will be true after the events in the scripts occur

Props: Slots representing the object involved in events

Roles: People involved in the events

Track: Variations on the scripts. Different tracks may share components of the same script.

Scenes: The sequence of events that occurs. Events are represented in the conceptual dependency form.

Scripts are useful in describing certain situations, such as robbing a bank. This might involve the following steps:

- getting a gun
- holding up a bank
- escaping with the money

Here props might be

- gun, G
- loot, L
- bag, B
- get-away car, C

The Roles might be

- robber, R
- cashier, M
- bank manager, O
- policeman, P

The entry conditions might be

- R is poor
- R is destitute

The result might be

- R has more money;
- O is angry;
- M is in a state of shock;
- P is shot.

There are three scenes: obtaining the gun, robbing the bank, and making the getaway.

The full scripts are described in Figure 5.4.

Scripts: ROBBERY	
Props: G=Gun	Roles: R=Robber
Entry Conditions: R is poor	Results: R has more money
Scene 1: Getting a gun R PTRANS R into gun shop R MBUILD R Choice of G R MTRANS Choice R ATRANS buys G (Go to Scene 2)	
Scene 2: Holding up the bank R PTRANS R into bank R ATTEND eyes M,O, and P R MOVE R to M position R GRASP G R MOVE G to point to M R MTRANS “Give me the money or ELSE” to M P MTRANS “Hold it! Hands up!” to R R PROPEL shoots G P INGEST bullet from G M ATRANS L to M M ATRANS L puts in bag B M PTRANS exits O ATRANS raises the alarm (Go to Scene 3)	
Scene 3: The Getaway M PTRANS C	

FIGURE 5.4 Pseudocode For Robbing the Bank.

Advantages of scripts:

- Ability to predict events
- A single coherent interpretation may be built from a collection of observations.

Disadvantages:

- Less general than frames
- May not be suitable to represent all kinds of knowledge.

CHAPTER 6

EXPERT SYSTEMS

6.1 Introduction

An expert system is an artificial intelligence program that has expert level knowledge about a particular domain and knows how to use its knowledge to respond properly. "Domain" refers to the area within which the task is being performed. Ideally, the expert system should be a substitute for a human expert. Edward Feigenbaum of Stanford University defined an expert system as "an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions." It is a branch of AI introduced by researchers in the Stanford Heuristic Programming Project.

6.2 Definition of an Expert System

- An expert system is a computer program designed to act as an expert in a particular field of knowledge or area of expertise. Expert systems are also known as knowledge-based systems.
- Expert systems are sophisticated computer programs that manipulate knowledge to solve problems.
- An expert system is a system that offers intelligent advice or makes an intelligent decision about a processing function.
- An expert system is a computer program that contains a knowledge base and a set of algorithms or rules that infers new facts from knowledge and from incoming data.

The method used to construct such systems, knowledge engineering, extracts a set of rules and data from an expert or experts through extensive questioning. This material is then organized in a format suitable for inquiry, manipulation, and response. While such systems do not often replace the human experts, they can serve as useful assistants.

6.3 Characteristics of an Expert System

Expert systems should have the following characteristics:

1. the ability to solve complex problems with the same (or greater) solvency as a human expert
2. heuristic reasoning through empirical rules, which properly interacts with human experts
3. ability to work with data that contains errors, using uncertainty procedural rules
4. ability to consider multiple hypotheses simultaneously
5. perform at the level of a human expert
6. ability to respond in a reasonable amount of time. Time is crucial, especially for real-time systems.
7. be reliable and should not crash
8. not be a black box; instead, the expert system should be able to explain the steps of the reasoning process. It should justify its conclusions in the same way as a human expert explains why he arrived at a particular conclusion.
9. need heavy investment, and there should be a considerable Return on Investment (ROI).

6.4 Architectures of Expert Systems

There are two types of architectures in expert systems:

- Rule-based system architecture (Production Systems)
- Non-production system architecture

Rule-Based System Architecture

This is most common form of architecture used in expert and other knowledge-based systems. This type of system uses knowledge in the form of production rules, i.e., if.......then rules.

IF: Condition 1 and condition 2

THEN: Take action 3

Each rule represents a small chunk of knowledge related to a given domain of expertise. A number of related rules collectively may correspond to a chain of inferences which lead from some initially-known facts to some useful conclusions. Inference in a production system is achieved by a process of chaining through the rules recursively, either in a forward or backward direction, until a conclusion is reached or until a failure occurs.

Components of an Expert System

Figure 6.1 shows the complete structure of rule-based expert system.

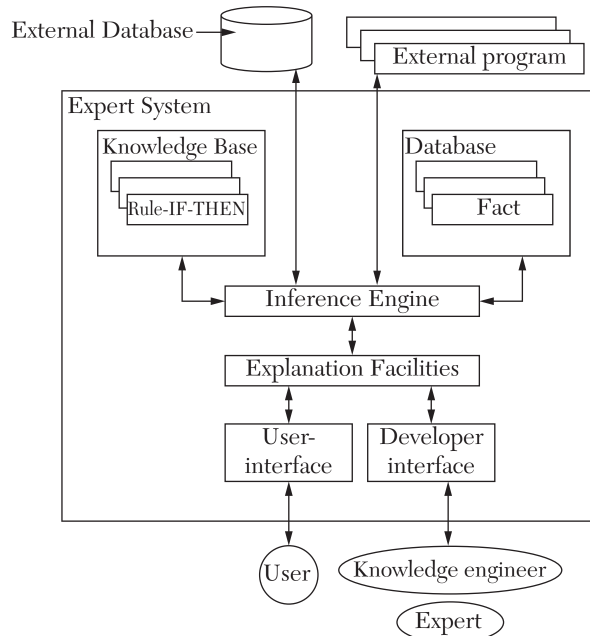


FIGURE 6.1 Complete Structure of a Rule-Based Expert System.

The fundamental modules of an expert system are the

1. knowledge base
2. inference engine
3. user interface
4. explanation facility
5. knowledge acquisition facility
6. external interface
7. database

1. Knowledge Base

A knowledge base is an organized collection of facts about the system's domain. Facts for a knowledge base must be acquired from human experts through interviews and observation. A knowledge base contains domain-specific and high quality knowledge. Knowledge is required to exhibit intelligence. The success of any expert system largely depends upon the collection of highly accurate and precise knowledge.

Components of a Knowledge Base

The knowledge base of an expert system is a store of both factual and heuristic knowledge.

- **Factual knowledge:** This is the information widely accepted by knowledge engineers and scholars in the task domain.
- **Heuristic knowledge:** This is about the practice of generating an accurate judgment, one's ability to make an evaluation, and guessing.

Knowledge Representation

This is a method used to organize and formalize the knowledge in the knowledge base. This knowledge is then usually represented in the form of "IF-THEN" rules (production rules). "If some condition is true, then the following inference can be made (or some action taken)." The knowledge base of a major expert system includes thousands of rules. A probability factor is often attached to the conclusion of each of the production rules because the conclusion is not a certainty. For example, a system for the diagnosis of eye diseases might indicate, based on the information supplied to it, a 90

percent probability that a person has glaucoma. It might also list conclusions with lower probabilities. An expert system may display the sequence of rules through which it arrived at its conclusion; tracing this flow helps the user to appraise the credibility of its recommendation. The knowledge base is formed by readings from various experts, scholars, and knowledge engineers. A knowledge engineer also monitors the development of the expert system.

2. Inference Engine

A very important element of the expert system is also called the inference engine. Knowledge of science must always be stored in the knowledge base in a formalized form that is understandable to the inference engine. The inference engine can be divided into following functional elements:

- **Control system:** This determines the order of testing in the knowledge base rules.
- **Rule-interpreter:** This defines the Boolean (the true, not true uncertainty factor) application rules.
- **Explanation mechanism:** This justifies the outcome to the user with the reasoning process and generates a report.

The inference engine repeatedly applies the rules to the working memory, adding new information (obtained from the rules' conclusions) to it until a goal state is produced or confirmed. One of several strategies can be employed by an inference engine to reach a conclusion. Inference engines for rule-based systems generally work by either the forward or backward chaining of rules. These two strategies are:

- **Forward chaining:** This is a data-driven strategy. The inference process moves from the facts of the case to a goal (conclusion). The strategy is thus driven by the facts available in the working memory and by the premises that can be satisfied. The inference engine attempts to match the condition (IF) part of each rule in the knowledge base with the facts currently available in the working memory.

  Forward-chaining systems are commonly used to solve the open-ended problems of a design or those that involve planning, such as establishing the configuration of a complex product.

- **Backward chaining:** The inference engine attempts to match the assumed conclusion to the goal or sub-goal state with the conclusion

(THEN) part of the rule. If such a rule is found, its premise becomes the new sub-goal. In an expert system with few possible goal states, this is a good strategy to pursue.

If the assumed goal state cannot be supported by the premises, the system will attempt to prove another goal state. Thus, a possible conclusion is to perform a review until a goal state that can be supported by the premises is encountered. Backward chaining is best suited for applications in which the possible conclusions are limited in number and well defined.

3. User Interface

A user must have a way to communicate with the system. The component of an expert system that helps its user to communicate with it is known as the "user interface." The function of a user interface is to provide a means for bi-directional communication in which the user describes the problem and the system responds with solutions or recommendations. The user interface helps to explain how the expert system has arrived at a particular recommendation. The explanation may appear in the following forms:

- natural language displayed on the screen
- verbal narrations in natural language
- a listing of rule numbers displayed on the screen.

The user interface makes it easy to trace the credibility of the deductions.

Advantages of the User Interface

1. It should help users to accomplish their goals in the shortest possible way.
2. It should be designed to work for users existing or desired work practices.
3. Its technology should be adaptable to the user's requirements, not the other way around.
4. It should make efficient use of the user's input.

4. Explanation Facility

This is a part of the user interface. It enables the user to ask the expert system how a particular conclusion is reached and why a specific task (fact) is needed.

An expert system must be able to explain its reasoning and justify its advice, analysis, or conclusion. Hence, the explanation facility must be superb.

5. Knowledge Acquisition Facility

The major bottleneck in expert system development is knowledge acquisition. This includes the elicitation, collection, analysis, modeling, and validation of knowledge for knowledge engineering and knowledge management projects. Various techniques of knowledge acquisition and the inherent problems associated with that will be discussed in later topics.

6. External Interface

This allows an expert system to work with external files using programs written in conventional programming languages such as C, Pascal, Fortran, and Basic. It provides the communication link between the expert system and external environment. When there is a formal consultation, it is done via the user interface. In real-time expert systems where they form a part of the closed loop system, it is not proper to expect human intervention every time conditions must be fed in to get remedies. Moreover, the time-gap is too narrow in real-time systems. The external interface with its sensors gets minute-by-minute information about the situation and acts accordingly. Such real-time expert systems are of tremendous value in industrial process controls, nuclear power plants, and supersonic jets.

The communication subsystem is part of the external interface that permits the system to communicate with a global database for its operation.

7. Database

A database is a collection of information that is organized so that it can easily be accessed, managed, and updated. It is working storage, a "notepad" that the inference engine can use to hold data while it is working on a problem. It holds all the data about the current task, including:

- the user's answers to questions
- any data from outside sources
- any intermediate results of the reasoning
- any conclusions reached so far.

There is a clear distinction between the knowledge base and the database. The knowledge base contains know-how, and it can be applied to many different cases. Once built, a knowledge base will be saved and used many times over. The database contains data about the particular case that is being run at the time. For example, the knowledge base about sales problems might be applicable to any small manufacturing business. During a run, the database would contain data about a specific company and its trading performance. The database can also be called "the world model."

Non-Production System Architecture

Instead of rules, these systems employ more structured representation schemes like the semantic (associative) network, frames, tree structure (decision trees), or even neural networks.

1. Associative (Semantic) Network

Semantic network representation schemes are networks made up of nodes connected by directed arcs. The nodes represent object, attributes, concepts, or other basic entities, and the arcs, which are labelled, describe the relationship between the two nodes they connect. Special network links include the IS-A and HAS-PART links, which designate an object as being a certain type of object (belonging to a class of the object) and being a subpart of another object, respectively.

Associative network representations are especially useful in depicting a hierarchical knowledge structure where property inheritance is common. Objects belonging to a class of other objects may inherit many of the characteristics of the class. Inheritance can also be treated as a form of default reasoning. This facilitates the storage of information when shared by many objects, as well as the inferencing process.

For example, one expert system based on the use of an associative network representation is CASNET (Casual Associative Network), which was developed at Rutgers University during the 1970s (Weiss et al., 1978). CASNET is used to diagnose and recommend treatment for glaucoma, one of the leading causes of blindness.

The network in CASNET is divided into three types of knowledge:

- **Patient observations (tests, symptoms, and other signs):** These are provided by the user during an interactive session with the system. The system is presented to the user during an interactive session. The system presents menu type queries, and the user selects one of several possible choices.
- **Pathophysiological states:** These observations help to establish the abnormal condition caused by the disease process. The condition is established through the casual network model as part of the cause and effect relationship relating the symptoms and other signs to diseases.
- **Disease categories:** Inference is accomplished by traversing the network, following the most possible paths of causes and effects. Once a sufficiently strong path is determined through the network, the diagnostic conclusions are inferred using classification tables that interpret the patterns of the casual network. These tables are similar to rule interpretations.

2. Frame Architecture

Frames are structured sets of closely-related knowledge, such as an object or concept name, the object's main attributes, its corresponding values, and possibly some attached procedure (if-added, if-needed, or if-removed procedures). The attribute's values and procedure are stored in specified slots and slot facts of the frame. Individual frames are usually linked together as a network, and, much like the nodes, this is an associative network, including property inheritance and default reasoning. Several expert systems have been constructed with frame architecture, and a number of building tools which create and manipulate frame structured systems have been developed.

For example, PIP (Present Illness Program) was used to diagnose patients using low cost, easily-obtained information. The medical knowledge in PIP is organized in frame structures, where each frame is composed of the categories of slots with names such as:

- typical findings
- logical decision criteria
- complimentary relations to other frames
- differential diagnosis
- scoring.

A special IS-sufficient slot is used to confirm the presence of a disease when key findings correlate with the slot contents.

3. Decision-Tree Architecture

When knowledge can be structured in a top-to-bottom manner, it may be stored in the form of a decision tree. For example, the identification of objects (equipment faults, physical object diseases, and the like) can correspond to an object's attribute, and the terminal nodes can correspond to the identities of objects. A decision tree takes input from an object given by a set of properties and outputs a Boolean value (yes/no decision). Each internal node in the tree corresponds to a test of one property. Branches are labelled with possible values of the test.

For example, assume the problem is waiting for a table at a restaurant. A decision tree decides whether to wait (or not) in a given situation. Here are some of the following attributes:

- **Alternative:** Alternative restaurant nearby
- **Bar:** Bar area to wait
- **Fri/Sat:** True on Fridays and Saturdays
- **Hungry:** Whether we are hungry
- **Patrons:** How many people are in the restaurant (none, some, or full)
- **Price:** Price range ($, $$, or $$$)
- **Raining:** Raining outside
- **Reservation:** Whether we made a reservation
- **Type:** Kind of restaurant (French, Italian, Thai, or Burger)
- **Wait Estimate:** Estimated wait time (<10, 10–30, 30–60, or >60).

The above problem can be explained with the help of a decision tree.

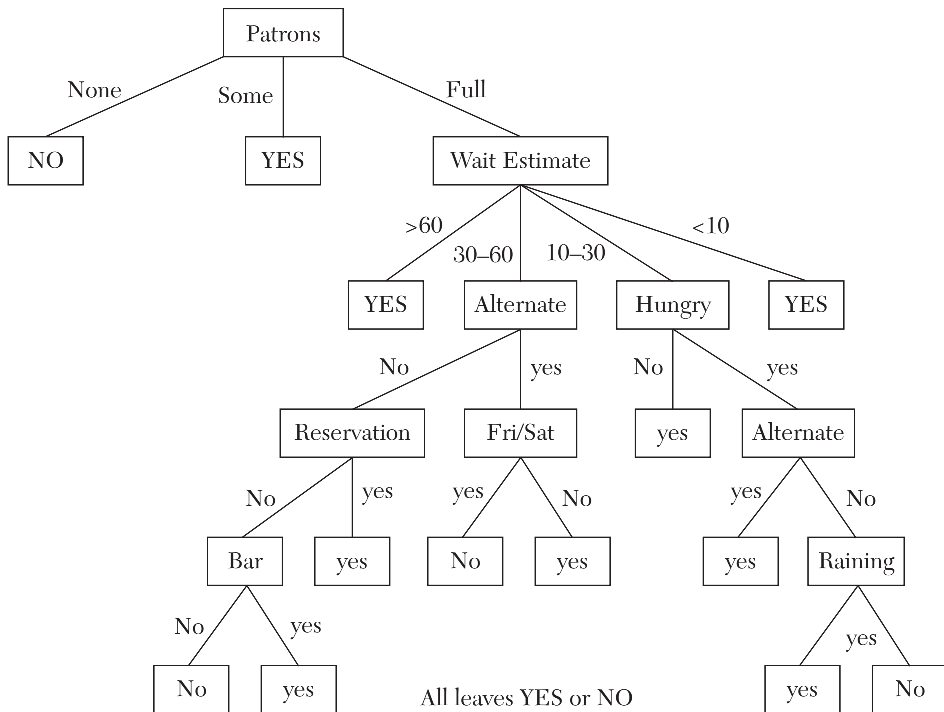


FIGURE 6.2 A Decision Tree.

Advantages of the Decision Tree Architecture

1. Decision trees are open systems, as it is easy to link the end of a pathway within a decision tree to start another decision tree.
2. Decision trees are simple natural programs that can adopt to complexity and chaotic conditions.
3. Decision trees are a "white box," meaning they are transparent and simple to understand and interpret. People are able to understand decision trees and therefore, they are designed for the organized retention of knowledge.
4. Decision trees can have value very quickly, even with a small number of nodes. Important insights can be gained from their usage that often stimulates ideas for knowledge evolution that were not obvious at first.

5. Decision trees can be created by subject matter experts without the need for software specialists.
6. Decision tree development enriches inductive and deductive reasoning, as they focus on the pathways and outcomes. This value is largely diluted with an expert system, as it is in the hands of the knowledge engineer and not the subject matter experts.
7. The construction of the decision tree is not just focused on business logic, but on a good dialogue and choices that influence behavior and decision making. The automated analysis of behavior enables the decisions tree to adopt certain behavioral dynamics.
8. New nodes and branches can be added to the tree when additional attributes are needed to further discriminate among new objects. As it gains experience, the value associated with the branches can be modified or the system can return more accurate results.

4. Blackboard Architecture

Blackboard system architecture refers to a special type of knowledge-based system which uses a form of opportunistic reasoning. It uses both forward and backward chaining and chooses them dynamically at each stage in the problem-solution process. The blackboard system architecture is composed of three functional components:

a. blackboard
b. knowledge source
c. control information/scheduler.

Figure 6.3 shows the architecture of a Blackboard system.

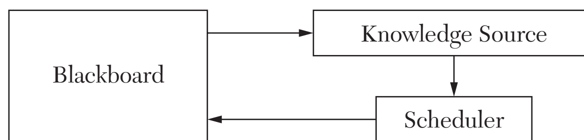
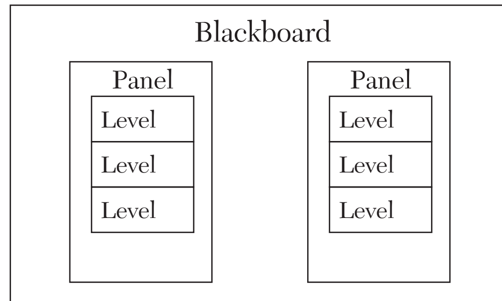


FIGURE 6.3 A Blackboard System Architecture.

a. **Blackboard:** The blackboard is the common data structure of knowledge sources. The blackboard is able to represent all states of some problem space. The blackboard contains several levels of description with respect to the problem space. These levels may have several relationships with each other, like IS-PART-OF.

The levels are parts of the same data structure. If separate data structures are needed, the blackboard is added into panels. Each panel, in turn, may contain several levels.



b. **Knowledge source:** This is a component that adds to the solution of the problem. It may be anything that reads from some level of the blackboard and suggests some change to parts of the blackboard. Its most common form is the production rule. Knowledge sources are completely unconnected to other knowledge sources.

c. **The scheduler:** This determines which knowledge source gets the chance to change the blackboard. Every execution cycle, it notices changes to the blackboard, activates the appropriate knowledge source, and selects one of these and executes it.

   For example, Hearsay-II is a speech recognition program. Speech can be recognized at several levels.

5. Neural Network Architecture

Neural networks are computing systems modeled on the human brain's mesh-like network of interconnected processing elements called neurons. Of course, neural networks are much simpler than the human brain (which is estimated to have more than 100 billion neurons). Like the brain, however, such networks can process many pieces of information simultaneously and can learn to recognize patterns and programs themselves to solve related problems on their own. A neural network is an array of inter-connected processing elements, each of which can accept inputs, process them, and produce a single output with the objective of imitating the operation human brain. Knowledge is represented in neural networks by the pattern of connections formed during the processing of elements and by adjusting

the weights of these connections. The strength of neural networks is in the applications that require sophisticated pattern recognition. The greatest weakness of neural networks is that they do not furnish an explanation for the conclusions they make. A neural network can be trained to recognize certain patterns and then apply what it learned to new cases where it can discern the patterns.

6.5 Expert System Life Cycle

A life cycle for an expert system is discussed here, and we outline the tasks and activities to be performed at each stage of development. The life cycle highlights the role of alternative development paradigms and the importance of social and organization characteristics in the system's transfer to users. There are five major stages in the development of an expert system. Each stage has its own unique features and correlation with the other stages.

1. **Identification Stage:** The first step in acquiring knowledge for an expert system is to characterize the important aspects of the problem. This involves identifying the participants, problems characteristics, resources, and goals.

   a. **Participants' Identification and Roles**

   Before we begin the knowledge acquisition, we must select the participants and their roles. Usually, this is the interaction between a single domain expert and a single knowledge engineer. The knowledge acquisition process can also include other participants. They may be multiple domain experts and multiple knowledge engineers.

   b. **Problem Identification**

   After we have chosen the knowledge engineer and the domain expert, they can proceed towards identifying the problem under consideration. This involves an informal exchange of views on various aspects of the problem, its definition, characteristics, and sub-problems. The objective is to characterize the problem and its supporting knowledge structure so that the development of the knowledge base may begin.

c. Resource Identification

Resources are needed for acquiring the knowledge implemented in the system and testing it. Typical resources are knowledge sources, time, computing facilities, and money.

d. Goal Identification

Most likely, the domain expert will identify the goals or objectives of building the expert system in the course of identifying the problem. It is helpful to separate the goals from the specific tasks of the problem.

2. **Conceptualization Stage:** The key concepts and relationships mentioned during the identification stage are made explicit during the conceptualization stage. It may be useful for the knowledge engineer to diagram those concepts and relationships. The following questions need to be answered before proceeding with the conceptualization process:

   - What types of data are available?
   - What is given and what is inferred?
   - Do the sub-tasks have names?
   - Do the strategies have name?
   - Are there identifiable partial hypotheses that are commonly used?
   - How are objects in the domain related?
   - What processes are involved in the problem's solution?
   - What are the constraints on these processes?
   - What is the information flow?

3. **Formalization Stage:** The formalization process involves mapping the key concepts, sub-problems, and information flow characteristics related during conceptualization into more formal representations based on various knowledge engineering tools or frameworks. The knowledge engineer now takes a more active role, telling the domain expert about the existing tool representations and the problem types that seem to match the problem at hand if, as a result of an informal

experiment with a preliminary prototype, the knowledge engineer believes there is a close fit with an existing tool or framework.

4. **Implementation Stage:** Implementation involves mapping the formalized knowledge from the previous stage into the representational framework associated with the tool chosen for the problem. As the knowledge in this framework is made consistent and compatible, and is organized to define a particular control and information flow, it becomes an executable program. The knowledge engineer evolves a useful representation for the knowledge and uses it to develop a prototype expert system. The prototype knowledge base is implemented by using whatever knowledge engineering aids are available for the representation (editors, intelligent editors, or acquisition programs). When the existing aids are inadequate, the knowledge engineer must develop new ones.

5. **Testing Stage:** The testing stage involves evaluating the prototype system and the representation forms used to implement it. Once the prototype system runs from start to finish on two or three examples, it should be tested with a variety of examples to determine the weaknesses in the knowledge base and inference structure. The elements that are usually found to cause poor performance because of faulty adjustments are the input/output characteristics, inference rules, control strategies, and test examples. Testing provides an opportunity to identify the weaknesses in the structure and implementation of the system and to make appropriate corrections.

6.6 Knowledge Engineering Process

The process of building an expert system goes through a number of stages. It is similar in many ways to the software engineering life cycle:

- **Requirements analysis:** Customer requirements are ascertained.
- **Knowledge-Acquisition:** Problem solving expertise is transferred from some knowledge source to a program.
- **Architectural Design:** High level organization of the system
- **System Design:** Detailed design of the (sub) system
- **Implementation:** Coding
- **Deployment:** Installation, operation, and maintenance.

Each of these phases includes the appropriate validation, verification, and quality assurance tests.

6.7 Knowledge Acquisition

Knowledge acquisition can be regarded as a method in which a knowledge engineer gathers information mainly from experts, but also from textbooks, technical manuals, research papers, and other authoritative sources, and translates this information into a knowledge base that is understandable to both machines and humans.

The person undertaking the knowledge acquisition (the knowledge engineer) must convert the acquired knowledge into an electronic format that a computer program can use.

In the process of knowledge acquisition for an expert system project, the knowledge engineer basically performs four major tasks in sequence:

- First, the engineer ensures that he or she understands the aim and objective of the proposed expert system to get a feeling for the potential scope of the project.
- Second, the engineer develops a working knowledge of the problem domain by mastering its terminology by looking up definitions in technical dictionaries and terminology databases. For this task, the key sources of knowledge are identified such as textbooks, papers, technical reports, manuals, code of practice, users, and domain experts.
- Third, the knowledge engineer interacts with experts via meetings or interviews to acquire, verify, and validate their knowledge.
- Fourth, the knowledge engineer produces a document or a group of documents (nowadays, in electronic format) which forms an intermediate stage in the translation of knowledge from the source to the computer program.

6.8 Difficulties in Knowledge Acquisition

Acquiring knowledge from experts is not an easy task. The following list includes some factors that add to the complexity of knowledge acquisition from experts and the knowledge transfer to a computer:

- Experts may not know how to articulate their knowledge or may be unable to do so.
- Experts may lack time or may be unwilling to co-operate.
- Testing and refining knowledge are complicated.
- Methods for knowledge elicitation may be poorly defined.
- System builders tend to collect knowledge from one source, but the relevant knowledge may be scattered across several sources.
- System builders may attempt to collect documented knowledge rather than use experts. The knowledge collected may be incomplete.
- It is difficult to recognize specific knowledge when it is mixed up with irrelevant data.
- Experts may change their behavior when they are observed or interviewed.
- Problematic interpersonal communication factors may affect the knowledge engineer and the experts.

6.9 Knowledge Acquisition Strategies

There are several ways by which knowledge is acquired. Some of the prominent methods are discussed below:

1. **Protocol Analysis:** This is the set of techniques known as the verbal protocol analysis. It is a method by which the knowledge engineer acquires detailed knowledge from the expert. A protocol is a record or documentation of the expert's step-by-step information processing and decision-making behavior. The expert is asked to talk about a thing out loud while performing the task or solve the problem under observation. In this method, the knowledge engineer does not interrupt while the expert is working.
2. **Observations:** In many ways, this is most obvious and straightforward approach to knowledge acquisition. In this method, the knowledge engineer observes the expert performing a task. This prevents the knowledge engineer from inadvertently interfering in the process, but does not provide any insight into why decisions are made.

3. **Interview Analysis:** This is an explicit technique that appears in several variations. It involves a direct dialog between the expert and the knowledge engineer. The interview process can be tedious. It places great demands on the domain expert, who must be able not only to demonstrate expertise but also to express it. Interviews can be unstructured, semi-structured, or structured. The success of an interview session is dependent on the questions asked and ability of the expert to articulate their knowledge.

4. **Introspection:** The expert becomes a knowledge engineer and then relies on a combination of introspection and knowledge of the expert system's architecture to convert know-how into the knowledge base.

5. **Teach back:** The knowledge engineer attempts to teach the information back to the expert, who then provides corrections and fills in the gaps.

These are some methods to help acquire knowledge from experts. Generally, no knowledge engineer sticks to one method, but adopts a combination of these methods. The knowledge engineer must aim to extract more of the deep knowledge that will help in understanding the fundamentals of the domain.

6.10 Advantages of Expert Systems

1. **Availability:** Expert systems are easily available due to vast production of software.

2. **Speed:** Expert systems offer great speed. They reduce the amount of work an individual puts in.

3. **Low Error Rate:** Their error rate is lower than that of humans.

4. **Steady Response:** They work steadily without getting emotional, tensed, or fatigued, whereas human experts, under stress, in a bad mood, or when time is limited, either make faulty assumptions or forget relevant factors.

5. **Reproducibility:** Many copies of an expert system can be made, but training new human experts is time-consuming and expensive, whereas the time for the duplication of an expert system is very short.

6. **Recovery:** Expert systems can be combined with other systems or database knowledge to address more complicated situations.
7. **Reducing Risk:** They can work in environments that are dangerous to humans.
8. **Consistency:** With expert systems, similar transactions are handled in the same way. The system will make comparable recommendations for like situations.
9. **Multi-Dimensional:** An expert system plays three major roles: the role of a problem solver, a tutor, and an archive. Even though natural language system interfaces are very primitive, expert systems of today serve these roles very well. A human expert who is a good problem solver need not be a good tutor.
10. **Efficiency:** An expert system makes things more efficient by reducing the time needed to solve problems. Expert systems provide strategic and comparative advantages that may create problems for competitors.

6.11 Limitations of Expert Systems

1. The different types of multidimensional problems that are faced by various users while performing activities cannot be efficiently tackled by expert systems.
2. Expert systems do not respond well to situations outside their range of the expertise.
3. Some of the typical expert system at times are not able to make available commonsense knowledge and broad-ranging contextual information.
4. There is no flexibility or ability to adapt to changing environments.
5. The construction process of an expert system is a laborious one. Currently, a lot of resources are required.
6. Expert systems focus on very specific topics, like computer faults, radiology, and diagnostic skills. The major reason for this situation is the difficulty in extracting knowledge, and building and maintaining a large knowledge base.
7. The verification of the correctness of any large computer system is difficult to prove, and expert systems are particularly difficult to verify.

This is a serious problem, as expert system technology is being applied to critical applications such as air traffic control, nuclear reactor operations, and weapon systems.

8. There is little learning from experience. Current expert systems are handcrafted; once the system is completed, its performance will not improve without further attention from its programmers.

6.12 Examples of Expert Systems

The following are some successful expert systems in different domains that not only helped pioneer the development of new techniques and tools, but also proved that AI systems can be terribly successful in select areas of expertise.

1. **DENDRAL:** This was developed at Stanford in the 1960s. DENDRAL was one of the first systems to rival the performance of domain experts. DENDRAL stored and reasoned with knowledge from the field of organic chemistry using a planned generate-test search paradigm. Specifically, the task was to determine the molecular structure of an organic molecule. It receives as its input a molecular formula with a set of constraints which serve to restrict the possible interconnections among atoms. A list of all possible ways of assembling the atoms into molecules is generated. These are ordered using the knowledge base to make testable predictions about candidate molecules.

   This enables the pruning of the candidate list. Because organic molecules tend to be very large, the number of possible structures for these molecules tends to be huge. DENDRAL addresses the problem of this large search space by applying the heuristic knowledge of expert chemists to the structure elucidation problem. DENDRAL's methods proved remarkably effective. META-DENDRAL added a machine-learning capability in the form of an inductive rule learner based on a hill climbing algorithm using raw mass spectrographic data. The new heuristics were then used in deducing the structure of unknown molecules from their mass spectra. Although META-DENDRAL is no longer an active program, its contributions to ideas about learning and discovery are being applied to new domains.

2. **MYCIN:** Stanford was also the home of another influential expert system called MYCIN. MYCIN established the methodology of contem-

porary expert systems. MYCIN was originally written in INTERLISP, a dialect of the LISP programming language.

MYCIN is a medical expert system that assists a physician who is not an expert in the field of antibiotics with the treatment of blood infections. Figure 6.4 shows the structure of MYCIN.

MYCIN consists of five modules:

a. knowledge base

b. a patient database

c. a consultation program

d. an explanation program

e. a knowledge acquisition program.

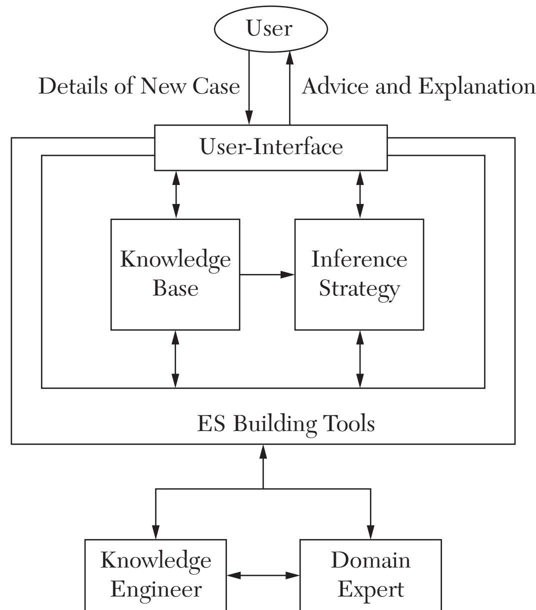


FIGURE 6.4 The Structure of MYCIN.

The knowledge is organized as a series of IF-THEN rules. Certain factors can be associated with the knowledge. Patient information is stored in a contextual form. This includes data such as blood samples, recent operative procedures, and drugs. The selection takes place after

the diagnosis. It consists of selecting candidate drugs and then choosing a preferred antibiotic. MYCIN itself has never been used in a clinical setting, but descendants of the program have.

3. **EMYCIN (Empty MYCIN):** This system allows the MYCIN architecture to be applied to another medical domain besides blood diseases. It is not a general-purpose problem solving architecture, rather, it is more suited to diagnostic tasks in medicine. The PUFF system was the first program built using EMYCIN. PUFF's domain is the interpretation of pulmonary functioning tests for patients with lung disease. The program can diagnose the presence and severity of lung disease and produce reports for the patient's file. The knowledge acquisition program TEIRSIAS was built to assist domain experts in refining the EMYCIN knowledge base. TEIRSIAS developed the concept of meta-level knowledge, i.e., knowledge by which a program cannot only use its knowledge directly, but can examine it, reason about it, and direct its use.

4. **PROSPECTOR:** A classic expert system is the PROSPECTOR program, which determines the probable location and type of ore deposits based on geological information about a site. PROSPECTOR attempts to predict the minerals to be found there. Like MYCIN, PROSPECTOR is a rule-based system that uses certainty factors to represent the strengths of the rules. PROSPECTOR deals with geologic settings, structural controls, and the kinds of rocks, minerals, and alternate products present or suspected. It compares observations with stored models of ore deposits, notes the similarities, differences, and missing information, asks for additional information if necessary, and then assesses the mineral potential of the prospect.

CHAPTER 7

LEARNING

7.1 Learning

Although everyone seems to know what it is, learning is actually very difficult to precisely define. Roughly, any system that improves its performance in response to internal changes caused by experience can be said to learn. This definition can be related to human beings. In psychology, various generalized definitions of learning have been proposed and many of them interpret learning as the change in the behavior of a being, subject to a given situation or a sequence of his or her repeated experiences in that situation.

In AI, machine learning can be defined as the capability of an AI system to improve its performance over a period of time. This, of course, assumes the capability of the system to acquire new knowledge and skills, as well as its capability to recognize the existing knowledge based on the newly acquired knowledge. Machine learning has grown into a widespread research field devoted to the search for new learning methods and/or learning algorithms, as well as their implementations.

7.2 General Model for Machine Learning Systems

Machine learning usually starts with some knowledge and the corresponding knowledge organization so that a system can interpret, analyze, and test the knowledge acquired. Figure 7.1 is a model of a machine learning system.

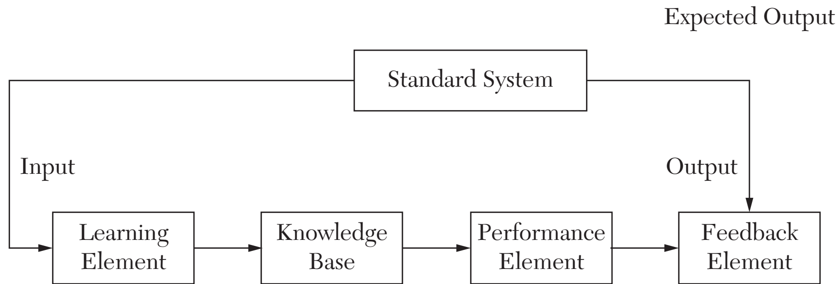


FIGURE 7.1 Learning System Model.

The figure shown above is a typical learning system model. It consists of the following components:

1. Learning Element
2. Knowledge Base
3. Performance Element
4. Feedback Element
5. Standard System

1. **Learning Element:** This receives and processes the input obtained from a person (i.e., a teacher) from reference materials like magazines, journals, or from the environment at large.

2. **Knowledge Base:** This is somewhat similar to the database. Initially it may contain some basic knowledge. Therefore, it may receive more knowledge which may be new and so it can be added as it is or it may replace the existing knowledge.

3. **Performance Element:** This uses the updated knowledge base to perform some tasks or solves some problems and produces the corresponding output.

4. **Feedback Element:** This receives two inputs, one from the learning element and one from the standard (or idealized) system. The feedback element identifies the differences between the two inputs. The feedback is used to determine what should be done in order to produce the correct output.

5. **Standard System:** This is a trained person or a computer program that is able to produce the correct output. In order to check whether the machine learning system has learned well, the same input is given to the

standard system. The output of a standard system and that of the performance element are given as inputs to the feedback element for the comparison. The standard system is also called an "idealized system."

7.3 Characteristics of Machine Learning

There are several characteristics of machine learning:

1. highly accurate predictions using test data (the goal is not to uncover the underlying "truth")
2. methods should be general-purpose, fully automatic, and "off-the-shelf" (however, in practice, the incorporation of prior, human knowledge is crucial)
3. rich interplay between theory and practice
4. emphasis on methods that can handle large data sets.

7.4 Types of Learning

The definition of learning is too broad and too vague to be of much use. Cognitive scientists have given the forms of learning various names: rote learning, direct instruction, learning by analogy, learning by deduction, learning by induction (also called "learning from examples"), failure-driven learning, learning by being told (also called "learning by instruction"), and learning by exploration, to name just a few. Although each of these forms of learning emphasize a different aspect of learning, they all involve a change to an internal, persistent memory of the system.

7.4.1 Rote Learning or Memorization

Rote learning is known as learning by repetition. It is a method of learning that involves memorization. This memorization is usually achieved through the repetition of activities, such as reading or recitation and the use of flashcards and other learning aids. The theory behind this learning technique is that students will commit facts to memory after repeated study and will then be able to retrieve those facts whenever necessary.

7.4.2 Direct Instruction

This type of learning is different from rote learning. It is the use of straightforward, explicit teaching techniques, usually to teach a specific skill. It is

a teacher-directed method, meaning that the teacher stands in front of a classroom and presents the information. For example, the teacher might give a lesson that very clearly outlines the order of all the planets in the solar system.

7.4.3 Learning by Analogy

Learning by analogy is the process of learning a new concept or solution through the use of similar known concepts or solutions. We use this type of learning when solving problems on an exam where previously learned examples serve as a guide or when we make frequent use of analogical learning. This form of learning uses more inferring than either of the previous forms, since difficult transformations must be made between known and unknown situations.

7.4.4 Learning by Deduction

Deduction means to draw conclusions from given facts. Deduction is applied to obtain a generalization from a domain theory, a solved example, and its explanation. It is a logical process in which a conclusion is based on the concordance of multiple premises that are generally assumed to be true. It is sometimes referred to as top-down logic.

For example,

1. All men are mortal.
2. Socrates is a man.
3. Therefore, Socrates is mortal.

The first premise states that all objects classified as a "men" have the attribute "mortal." The second premise states that "Socrates" is classified as a "man," a member of the set "men." The conclusion then states that "Socrates" must be "mortal" because he inherits this attribute from his classification as a "man."

7.4.5 Learning by Induction (Learning by Examples)

This is a process of learning by example. The system tries to induce a general rule from a set of observed instances. The learning method extract rules and patterns out of massive data sets. The learning process belongs to supervised learning and unsupervised learning, does classification, and constructs class definitions, called induction or concept learning.

- **Supervised Learning:**

  The program is "trained" via past experiences based on a predefined set of "training examples," which then facilitate its ability to reach an accurate conclusion when given new data. In supervised learning, the output datasets are provided, and these are used to train the machine and get the desired outputs.

  For example, in facial recognition, the system learns by examples as to what a face is in terms of the structure and color, so that after several iterations, it learns to define a face.

- **Unsupervised Learning:**

  No data sets are provided. Instead, the data is clustered into different classes.

  For example, in facial recognition, since there is no desired output (in this case, that is provided) the categorization is done so that the algorithm correctly differentiates between the faces of a horse, cat, or human (clustering of data).

  So, learning by induction is a method that is used frequently by humans. It is a powerful form of learning, like analogical learning, which also requires more inferring than other methods. We use inductive learning of instances or examples of a concept. For example, we learn the concepts of color or a sweet taste after experiencing the sensations associated with several examples of colored objects or sweet food.

7.4.6 Failure-Driven Learning

Failure-driven learning is based on creating a program that will learn by making mistakes and then finding a solution so that the mistake does not happen again. This is similar to the way humans learn. If we make a mistake, we usually try to learn from that mistake to improve ourselves so we do not make it again.

Sussman's Hacker is an example of a failure-driven learning system that operates in the block world. It solves problems by looking up plan schemes in its Plan Library and fitting them together. There is a "gallery" of "critics" that do the plan criticism. Hacker analyzes problems in terms close to standard computer programming. The plan it is to execute looks like a program that is a linear object containing conditionals and loops. However, the program is still hierarchical: problems are solved by programs, whose steps

become new problems. Once the program has been completed, Hacker executes it. If a bug halts execution, then Hacker corrects the program that led to it. The next time this program is used, that particular bug will not occur again.

Suppose a programmer has been given the problem to put the "A" block on the top of the "B" block in the situation shown in Figure 7.2.

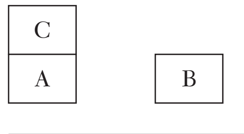


FIGURE 7.2 Failure-Driven Learning by the Problem That Puts "A" on the Top of "B."

At first, the program cannot execute because the "C" block is on the top of the "A" block. The program now has to figure a solution to why it cannot lift the "A" block. It devises a solution to move the "C" block off the "A" block. Once the "C" block has been moved, it can place the "A" block on the top of "B" block, and its objective is completed.

The sample code to do this would be as follows (in the example below, x is block "A," y is block "B," and z is block

"C" or any block that is on top of the block being moved).

Original Code:

`[to do ? task ( achieve (on ? x ?y)):` To get something on something else
`(move ? x ? y)] :` Use the move operation

Program-altered Code:

`[to do ? task (achieve (on ? x ? y)) :` To get something on something else
`(prog (for each ? z (on ? z ? x)`
`(get-ritdof ? z))`
`(move ? x ? y)];` Then do the move.

The "program altered code" is code that the program created so it could remove any block on top of the block that it was originally supposed to move.

The problem with the above example is that the program cannot necessarily know why it cannot lift block "A." It could just have another box on

the top of it, or it could be glued to the ground, or both. It would need some way of determining the physical situation of block "A."

Inside a computer program, it is reasonable to assume that one has complete knowledge of the reasons for a bug. However, this assumption is actually not reasonable, since there are intermittent bugs in complex software, but automatic programming is hard enough without worrying about such bugs.

Now take the example of a restaurant. Suppose you go to an ordinary restaurant and eat a meal with your fingers. Now, you expect a conflict with the scheme, in that normally you would use a knife and fork. This kind of contradiction is evident with the current hypothesis, which you want to be changed. Next time, you go to another reputed restaurant and you eat with a knife and fork or other utensils which were not available at the previous restaurant. This will make you recall the past scenario and generate new or revised plans to include in your schemes. We learn from failures of expectations about what will happen when schemes are used for planning or understanding. However, in a sense, all learning is failure-driven. We must change the rules whenever something happens that should not have happened or fails to happen when it should have happened.

7.4.7 Learning by Being Told or Getting Advice (Learning by Instruction)

Learning by being told is another area of AI learning. It allows a system to improve its task performance by the repetition of this task, like an athlete who trains himself by the execution of the same movement several times. During these repetitions, the system gains know-how and is progressively improved. It requires that the learning system select and transform the knowledge into a usable form and then integrate it into the existing knowledge of the system. It includes learning from teachers and learning by using books, publications, and other types of instruction.

Learning by being told is simply the interaction of a teacher (human) and an AI student. The teacher is there to teach the AI how to do things in the real world. Because the teacher has a grasp of real-world situations, it virtually eliminates the need for induction by the AI. The only problem is the communication between the teacher and the AI student. Preferably, the teacher would want to teach in English, but the AI does not understand English. There is not sufficient English to code a translator.

- One solution is for the teacher is to use limited English. This reduces the need to interpret the unnecessary parts of the sentence, such as the pronunciation and articles (i.e., instead of saying "It is easier to move the little boxes first," the teacher could say "Move little boxes first." This reduces the commands down to verbs, adjectives, nouns, and words telling the program in what order to move the boxes.
- Another solution is for the teacher to actually put the instructions into code. This is not preferable, since one of the many goals for AI is to get it to interpret English commands, sometimes on the fly. Short instructions are no problem to put into code, but should the instruction set be lengthy, the teacher will spend a lot time coding the set of instructions, instruction-by-instruction, until the AI understands the way the teacher is teaching it. This can be time consuming, especially if the AI does not learn the set of instructions, or learns it incorrectly and new instructions need to be created to nullify what the program has learned.

7.4.8 Learning by Exploration

Learning by exploration is a little different from the other ways of learning. It is a restricted form of learning. The purpose of learning to explore is to just gather information and not really pursue a goal. All the system tries to do is find interesting information it can store and learn from it. But it does not explore until it has nothing left to explore. The system will follow a series of tasks. It will perform one task, which may add more tasks, and then move onto the next task. This causes the database of concepts to continue to grow.

The program will organize the tasks in order of "interestingness." The program will also not always look at each task. Sometimes, it needs to determine what would be a waste of time exploring. This causes a problem because the program needs some way of determining what task is worth exploring, and should it choose not to explore a task, it has to make sure it is not missing out on anything by ignoring it.

Sometimes the program will find that the tasks it has left are not interesting enough to explore. If this happens, it will go through all its tasks and explore a "suggestions" slot so it can make the tasks more interesting. This way, the program will more than likely not run out of tasks to explore.

The program should also be able to generate concepts from what it already contains in its database. This way, it can generate more tasks to explore or just create new concepts that may have a purpose in the real world.

7.5 Advantages of Machine Learning

1. Often, machine learning is much more accurate than human-crafted rules since it is data-driven.
2. Humans are often incapable of expressing what they know (e.g., the rules of English or how to recognize letters), but can easily classify examples.
3. Machine learning does not need a human expert or programmer.
4. The programs use automatic methods to search for hypotheses explaining data.
5. Machine learning is cheap and flexible, and it can be applied to any learning task.

7.6 Disadvantages of Machine Learning

1. Machine learning needs a lot of labeled data.
2. Machine learning is error-prone (it is usually impossible to get perfect accuracy).

CHAPTER 8

PROLOG

8.1 Preliminaries of Prolog

This chapter deals with the preliminaries of Prolog, an AI programming language. Specialized languages exist for the majority of activities in data processing. For example, COBOL is for business applications, FORTRAN is for scientific computations, and BASIC is for general purpose computing. In a similar way, LISP and Prolog are the two major languages used for majority of the AI problems. We discussed LISP earlier. In this chapter, we discuss the preliminaries of Prolog.

Prolog has been successful as an AI programming language for the following reasons:

- The syntax and semantics of Prolog are very close to formal logic. By this time, it must be clear to you that most AI programs reason using logic.
- The Prolog language has a built-in inference engine and automatic backtracking facility. This helps in the efficient implementations of various search strategies.
- This language has a high productivity and allows for easy program maintenance.
- Prolog is based on the universal formalism of "Horn clauses." The positive feature of this is its immunity to implementation dependencies, and programs tend to be uniform.
- Because of the inherent AND parallelism, Prolog can be implemented with ease on parallel machines.
- The clauses of Prolog have a procedural and declarative meaning. Because of this, understanding the language is easy.

- In Prolog, each clause can be executed separately as though it is a separate program. Hence, modular programming and testing are possible.
- Prolog's free data structure is amenable to complex data structures.
- As an interpreter, Prolog is suitable for quick prototyping and incremental system development.
- Program tracing during development is possible with modest debugging efforts in Prolog.

Logic programming is an approach to computer science in which the Horn clause form of first order logic is used as a high level programming language. Logic programming allows the programmer to describe a situation with formulae in predicate logic and use a mechanical problem solver to make inferences from the formulae.

8.2 Milestones in Prolog Language Development

1965 Robinson develops the resolution procedure.

1973 Colmeraur at Marseilles develops the Prolog Language in FORTRAN

1974 Kowlaski's work on predicate logic as a programming language

1977 The University of Edinburgh develops the Prolog interpreter on the DEC10 machine.

1980 The Imperial College develops micro-Prolog for personal computers.

1981 Japanese Fifth Generation Computer Systems adopts Prolog as its main programming language.

8.3 What is a Horn Clause?

In a Horn clause, one condition is followed by zero or more conditions. It is represented as follows:

```
conclusion:
   condition_1,
   condition_2,
   condition_3, .......
   condition_n.
```

The conclusion is true if, and only if, condition_1 is true and condition_2 is true and condition_3 is true and so on until condition_n is true.

In simple terms, a Horn clause consists of a set of statements joined by logical ANDs.

The basis of Prolog is formed by Horn clauses and Robinson's resolution rule.

8.4 Robinson's Resolution Rule

The principle of the resolution is as follows.

Two clauses can be resolved with one another if one of them contains a positive literal and the other contains a corresponding negative literal with the same predicate symbols and the same number of arguments. Consider the following clauses:

– X (a) V Y (p, q) ...(1)

– Y (p, q) V T (r, s) ...(2)

These two clauses can be unified to give

– T (r, s) V – X (a) ...(3)

Now (1) – (3) can be used for future computations.

8.5 Parts of a Prolog Program

A Prolog program consists of a set of clauses. A clause is either a fact or a rule. A fact is used to indicate a simple data relationship between the elements called objects.

For example, "Kumar likes toffees" is represented as

```
likes (kumar, toffees).
                objects/items
```

The word "likes" is a relation that links the objects together.

A predicate is the abstract sense of the relation that holds true between a certain number of arguments. A predicate is identified by the predicate name and its arity (number of arguments). In the example given above, "likes" is the predicate name and its arity is 2.

A predicate can have any number of arguments.

The simplest Prolog program is a set of facts, referred to as a database. Here is a database of "likes" facts:

```
likes(kumar, toffees).
likes(ram, aircrafts).
likes(mani, toffees).
likes(ram, cars).
```

8.6 Queries to a Database

Once a database has been created, one can make queries to it. A simple query consists of a predicate name and its arguments.

For instance, for the "likes" database created, the query

```
likes(ram, cars)
```

would return the value "True."

For the query

```
likes(murali, jeeps)
```

the system would return the value "False."

It is also possible that one can have a variable for an argument. If the query has a variable, then the system will try to evaluate those predicates for which the variable is "True." Normally, variables will start with an uppercase letter. For the query

```
like(murali, jeeps)
```

the system would return the value "False."

It is also possible that one can have a variable for an argument. If the query has a variable, then the system will try to evaluate those predicates for which the variable is "True." Normally, variables start with an uppercase letter. The query

```
likes(ram, What)
would have the answer
What = aircrafts
What = cars
```

8.7 How Does Prolog Solve a Query?

Prolog tries to match (this process is called “unification”) the arguments of the query with the facts in the database. If the unification succeeds, the variable is said to be instantiated. It is also possible that one can have variables for all the arguments.

The query

```
likes(Who, What)
```

would result in

```
Who = kumar,          What = toffees
Who = ram,            What = aircrafts
Who = mani,           What = toffees
Who = ram,            What = cars
```

The sequence adopted for this is the same as the sequence in the database.

8.8 Compound Queries

The queries that were posed to the system were simple ones. It is also possible to pose compound queries to the system. For this, consider the “likes” database again.

The query

```
likes(mani, What),likes(kumar, What)
```

has the meaning “Is there an item which Kumar and Mani like?” In Prolog, the comma symbol represents logical ANDs.

The system will respond

```
What = toffees.
```

8.9 The _ Variable

This is a special variable, the anonymous variable, that instructs the system to ignore the value of an argument. It unifies with anything but does not print.

The query

```
likes(ram,_)
```

will return the value "True" because the system can match from the database the predicate name and the argument. The anonymous variable is ignored.

8.10 Recursion in Prolog

If a function during execution calls itself again, then such a function is said to be recursive in nature.

To explain a recursion set of instructions, consider the evaluation of N answer books.

To evaluate N answer books,

If N = 0, then stop correction.

If N > 0, value one answer book, then evaluate N-1 answer books.

Recursion is a major built-in function in Prolog.

Let's discuss how recursion in Prolog works.

Consider the program that finds the "ancestor." The Prolog program for this is as follows.

```
ancestor(A, B) :/* Clause 1 */
      parent(A, B)
ancestor(A, B) :/* Clause 2 */
      parent(C, B),
      ancestor(A, C).
```

Together, these rules define two ways of how a person can be the ancestor of the other.

Clause 1 states A is an ancestor of B, when A is a parent of B.

Clause 2 states A is an ancestor of B, when C is a parent of B and A is an ancestor of C.

To verify how this works, consider the following database.

```
parent(person_1, person_2).
parent(person_1, person_3).
parent(person_3, person_4).
```

The query

```
ancestor(person_1, Whom)
```

will have the answers

```
Whom = person_2.
Whom = person_3.
Whom = person_4.
```

Any recursive procedure has to have

- a non-recursive clause to indicate when the recursive has to stop
- a recursive rule.

In the example given, Clause 1 will serve to stop the recursion.

8.11 Data Structures in Prolog

The list structure is an important data structure in Prolog. This is nothing but a collection of ordered sequences of terms. The elements of the list are written between the square brackets separated by the commas. For example, `[apple, orange mango, grapes]` is a list of fruits. Since a list is an ordered sequence, the list `[apple, grapes, orange, mango]` is not the same as the first list, even though they both have only four elements and the members of the list are the same.

8.12 Head and Tail of a List

The symbol "|" divides the list into two parts, the head of the list and the tail of the list, respectively.

In the fruit list,

```
[apple|Rest]
```

would give the result

```
Rest = [orange, mango, grapes].
```

An empty list (a list with no elements) is represented as [ ].

Example for List Unification

(a) `[H|T] = [1,2,3,4]` (b) `[H|T] =[a]`

```
H = 1                 H =[a]
T = [2,3,4].          T = [ ]
```

(c) `[H1, H2, H3]   |T] = [a, ,b, c, d, e]`

```
[H1, H2,  |  T] = [a]
H1 = a, H2 = b, H3 = c
T = [d, e]
```

This is false because there is an element for T.

Some of the operations possible on the list are given below.

8.13 Print all the Members of the List

The members of a list cannot be written using the write statement available in Prolog. For this purpose, one has to write a clause that uses to recursion for this purpose. This clause is

```
writelist([ ]).        /* If list is empty, stop
                       recursion */tv
writelist([H|T] :-
write([H]),            /* Write the first element of
                       the list*/
write([T]).            /* Recursive call of the clause
                       */
```

8.14 Print the List in Reverse Order

This is similar to the writelist clause discussed above. The modification is done in the ordering of the sub-goals. The clause is as follows.

```
   rev_print([ ]). /* If the list is empty, stop
recursion */
   rev_print([H|T]):-
        rev_print(T),
   write(H)
```

8.15 Appending a List

In this, all the arguments are lists. The first are appended together and returned in the third argument.

```
append([ ]), List, List)
append([H|List_1], List_2, [H|List_3]):
      append(List_1, List_2, List_3).
```

8.16 Find Whether the Given Item is a Member of the List

```
member (X, [X|Rest]). /* X is the first element of
the list */
/* X is not the first element of the list. So look
for X in the rest of the list.  */
member(X, [Y|Rest]):
member(X, Rest).
```

8.17 Finding the Length of the List

```
has_length([ ], 0).
has_length([H|T], N):
has_length(T, N1),
N = N1+1.
```

8.18 Controlling Execution in Prolog

The two major ways of controlling execution in Prolog is through fail and cut (represented as "!") predicates.

Fail Predicate

The fail predicate will make a clause fail during execution. In order to force backtracking, this predicate is useful. The purpose of this predicate and its importance is discussed using the following Prolog program.

```
clause 1:
person(Name, Designation),
```

```
write(Name),
write(Designate),
fail.
person(raman, researcher).
person(kumar, manager).
person(ravi, accountant).
person(selvan, partner).
```

When this program is executed, the system will bind "raman" to "Name" and "researcher" to "Designation" and print them. This clause deliberately fails using the fail predicate. This forces backtracking and the system instantiates another value to be the variable. Thus, the system will print all the names and designations, and it will fail because of the fail predicate.

In order to make the clause succeed, all that has to be done is to make the clause true. This is done by adding the clause without any conditions to it. This is done in the following program.

```
clause 1:
person(Name, Designation),
write(Name),
write(Designation),
fail.
clause 1. /* This clause will make clause 1 succeed */
person(raman, researcher).
person(kumar, manager).
person(ravi, accountant).
person(selvan, partner).
```

The point to be noted here is that the variables in the clause lose their bindings every time the rule fails. Backtracking forces a new binding.

However, the fail predicate is not sufficient to achieve total control over execution. The necessity of some other predicate is explained with

the following example. Consider the previous example of `person(Name, Designation)`.

Here, we do not want to print the name of the person whose designation is "accountant." The program for that is to check the designation and if the designation is "accountant," the program should not print the name. The program for that is shown below.

```
clause 1:-
    person(Name, Designation),
    check_designation(Designation),
    write(Name),
    write(Designation),
    fail.
clause 1:/* This clause will make clause 1  suc-
ceed */
    check_designation(accountant):fail
    check_designation(_). /* This clause will make
 the predicate  check_designation succeed */
    person(raman, researcher).
    person(kumar, manager).
    person(ravi, accountant).
    person(selvan, partner).
```

When the program is executed, and when `check_designation(accountant)` fails, the system checks `check_designation(_)` and succeeds. The anonymous variable binds any value to the variable. So "ravi" and "accountant" will also be printed, which are not the solution. The solution for this problem is achieved using the cut predicate.

Cut Predicate

The cut predicate is a built-in predicate that instructs the interpreter not to backtrack beyond the point at which it occurs. This is primarily used to prune the search space.

To explain the concept of a cut predicate, consider the following program with the facts and clause.

```
state(tamilnadu).
state(kerala).
state(andhra_pradesh).
state(uttar_pradesh).
state(karnataka).
state(madhya_pradesh).
state(S):
      write("Are you from") write(S), write("?"),
      reading(Reply), Reply = "yes",
      !
      write("So, you are from"), write(S).
```

Consider what will be the output when a person from Uttar Pradesh answers.

Are you from tamilnadu?
no

Are you from kerala?
no

Are you from andhra_pradesh?
yes

So, you are from uttar_pradesh.

In fact, the system reads the user's variable in `Reply`. If it is "no," then the `Reply` sub-goal fails and the system backtracks to get a new variable for S. When the user from Uttar Pradesh types "yes," the system allows the program to proceed beyond the cut. The cut will see to it that the query will end after the first "yes" answer and will not permit it to backtrack. This is the reason the system will not ask about Karnataka and Madhya Pradesh.

This is what is called backtracking because the system backtracks whenever the reply is "no." There are two ways to get out of this.

- Exhaust all of the state database.

- Allow the system to pass through the cut. The cut will prevent backtracking.

The cut predicate must be used with extreme caution. Otherwise, it is likely to disrupt the normal execution of the program by pruning needed states.

Here is the solution for the `person(Name, Designation)` problem.

```
clause 1:-
    person(Name, Designation), check_designation
  (Designation),
    write(Name),
    write(Designation),
clause 1./* This clause will make clause 1 suc-
ceed*/ check_designation(accountant):-
    !,
    fail.
check_designation(_). /* This clause will make
the predicate
check_designation succeed */
person(raman, researcher).
person(kumar, manager).
person(ravi, accountant).
person(selvan, partner).
```

Here, when `check_designation(accountant)` fails, Prolog backtracks to the next person's `(Name, Designation)` and the next variable binding is tried. Thus, "ravi" and "accountant" are not printed.

8.19 About Turbo Prolog

One of the most commonly available types of Prolog is Turbo Prolog, developed by Borland International. This Prolog runs on IBM compatible PCs in a DOS environment.

Turbo Prolog is a compiler. The general form of the program is as follows.

```
trace                      /* optional     */
project   "project_        /* optional     */
name"
include                    /* optional     */
"other_source_file"
domains                    /* This section defines      */
person = symbol            /* the domain used    */
shift = symbol
database
  works(person,            /* This section declares */
  shift)
                           /* the predicates that are to be
                           stored in the dynamic database
                           */

predicates
  known(person,            /*  This  section  declares  the
  person)                  domains of each argument */


goal
  knows(A, B).             /* optional. This is needed when
                           one likes to have an .EXE file
                           */
clauses
                           /*actual program starts here*/
      works(magesh, day).
      works(senthil, day).
  knows(X,Y):-
  works(X, S),
  works(Y, S),
  X <> Y.
```

Turbo Prolog expects that the domain type of each of its arguments in the predicates will be defined. For this purpose, the domains available are char, integer, real, string, symbol, and file.

Lists are declared in the domains using `*`. For example, a list of integers `(int_list)` is declared in the domains as follows:

```
domains
      int_list = integer*
```

Turbo Prolog's development environment is user-friendly, with windows for editing, dialogue, messages, and tracing.

The Turbo Prolog debugger is invoked using the "trace" command. This option will trace the execution of the complete program. If one wants to trace only certain predicates, then the option "shorttrace" is used.

CHAPTER 9

PYTHON

9.1 Languages Used for Building AI

LISP is one of the most popular languages for creating AI. Its best features include garbage collection, uniform syntax, dynamic typing, and an interactive environment. LISP code is written as s-expressions and consists of lists.

Another popular AI programming language is Prolog. The best thing about this language is a built-in unifier. Its main disadvantage is that this language is difficult to learn.

C/C++ is used for building simple AI programs in a short period of time. Java is not as fast as C, but its portability and built-in types make Java the choice of many developers. Finally, there is Python. As many developers have noted, Python is similar to LISP. It is one of the most popular AI languages. Why is this so? Why do developers code AI with Python? Let's check it out.

9.2 Why Do People Choose Python?

Python was created at the end of the 1980s. Its implementation started in 1989. Python's philosophy is very interesting, as it includes several aphorisms: it is explicit rather than implicit, simple rather than complex. Python creators value its beautiful design and look. They prefer the complex to the complicated, and they stated that readability counts. Python has a clean grammar and syntax. It is natural and fluent. Python's developers said that the language's goal is to be "cool" to use. Since it was named after Monty Python, a British comedy group, the language's users have a playful approach to writing many tutorials and other materials.

Developers have said that they enjoy the variety and quality of Python's features. Though it is not the perfect scientific programming language, its features are efficient:

- data structure
- classes
- flexible function calling syntax
- iterators
- nested functions
- kitchen-sink-included standard library
- great scientific libraries
- "cool" open source libraries (Numpy, Cython, IPython, and MatPlotLib).

Other features developers like about Python are as follows: the holistic language design, thought-out syntax, language interoperability, balance of high-level and low-level programming, documentation generation system, modular programming, correct data structures, numerous libraries, and testing frameworks. One of the disadvantages is the need for programmers to be good at MATLAB, as it is common in general scientific coding. That is why many developers publish open research code in MATLAB.

Compared to other OOP languages, Python is relatively easy to learn. It has a bunch of image intensive libraries, such as VTK, Maya 3D Visualization Toolkits, Scientific Python, Numeric Python, and Python Imaging Library. These tools are perfect for numeric and scientific applications.

Python is used everywhere and by everyone: in simple terminal commands, in vitally important scientific projects, and in big enterprise apps. This language is well designed and fast. It is scalable, open source, and portable.

9.3 Build AI Using Python

The first step is to get started. Though it sounds a bit stressful and hard, you should understand that building an AI in Python will take some time. The amount of time needed depends on your motivation, skills, and your level of programming experience.

In order to build an AI program with Python, you need to have some basic understanding of this language. This is not just a popular general-purpose programming language. It is also widely used for machine learning and computing. First of all, install Python. You may do that by installing Anaconda, the open source analytics platform. Include the needed packages for machine learning: NumPy, scikit-learn, iPython Notebook, and matplotlib.

The next step is to boost your machine learning skills. Of course, it is almost impossible to reach the ultimate understanding of machine learning in a short period of time (unless you are a genius or a machine like IBM's Watson). That is why it is better to start with gaining basic machine learning knowledge or improving your understanding with the help of the following courses: Andrew Ng's *Machine Learning course* and Tom Mitchell's *Machine Learning* lectures. You need a basic understanding of machine learning's theoretical aspects.

We have already mentioned Python's scientific libraries. These Python libraries will be useful when you build an AI. For example, you will use NumPy as a container of generic data. Since it contains an N-dimensional array object, tools for integrating C/C++ code, Fourier transform, random number capabilities, and other functions, NumPy is one of the most useful packages for scientific computing.

Another important tool is pandas, an open source library that provides users with easy-to-use data structures and analytic tools for Python. Matplotlib is another service you will like. It is a 2D plotting library that creates publication-quality figures. Among the best matplotlib advantages is the availability of 6 graphical user interface toolkits, web application servers, and Python scripts. Scikit-learn is an efficient tool for data analysis. It is open source and commercially usable. It is the most popular general-purpose machine learning library.

After you work with scikit-learn, you may take your AI programming using Python to the next level and explore k-means clustering. You should also read about decision trees, continuous numeric prediction, and logistic regression. If you want to learn more about Python in AI, read about the deep learning framework Caffe and a Python library Theano.

There are Python AI libraries, such as AIMA, pyDatalog, SimpleAI, and EasyAi. There are also Python libraries for machine learning, such as PyBrain, MDP, scikit, and PyML. If you are searching for natural language and text processing libraries, check out NLTK.

As you see, the importance of Python for AI is obvious. Any machine learning project will benefit from using Python. As AI needs a lot of research, programming artificial intelligence using Python is efficient – you may validate almost every idea with up to thirty code lines.

9.4 Running Python

We assume that everything is done with an interactive Python shell. You can either do this with an IDE, such as IDLE1, that comes with standard Python distributions, or just run iPython3 (or perhaps just ipython) from a shell.

Here we describe the simplest version that uses no IDE. If you download the zip file and change the directory to the "aipython" folder where the .py files are, you should be able to do the following, with the associated user input.

The first iPython3 command is in the operating system shell (note that the -i is important to enter interactive mode).

```
$ ipython3 -i searchAStar.py

Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
```

Type "copyright", "credits," or "license" for more information.

`IPython 5.1.0 – An enhanced Interactive Python.`

`?` –> Introduction and overview of IPython's features.

`%quickref` –> Quick reference.

`help` –> Python's own help system.

`object?` –> Details about "object", use `'object??'` for extra details.

```
In [1]: import searchProblem

In [2]: searcher1 = Searcher(searchProblem.acyclic_delivery_problem)

In [3]: print(searcher1.search()) # find first path
```

Sixteen paths have been expanded and 5 nodes remain in the frontier.

```
o103 –> o109 –> o119 –> o123 –> r123
```

```
In [4]: print(searcher1.search()) # find next path
```

Twenty-one paths have been expanded and 6 nodes remain in the frontier.

```
o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123
```

9.5 Pitfalls

It is important to know when side effects occur. Often, AI programs consider what would happen or what may have happened. In many such cases, we do not want side effects. When an agent acts in the world, side effects are common.

In Python, you need to be careful to understand the side effects. For example, the inexpensive function for adding an element to a list, namely `append`, changes the list. In a functional language like LISP, adding a new element to a list, without changing the original list, is a cheap operation. For example, if x is a list containing n elements, adding an extra element to the list in Python (using `append`) is fast, but it has the side effect of changing the list x. To construct a new list that contains the elements of x plus a new element, without changing the value of x, entails copying the list, or using a different representation for lists. In the search code, we will use a different representation for lists for this reason.

9.6 Features of Python

9.6.1 Lists, Tuples, Dictionaries, and Conditionals

Lists

- Python has a flexible and powerful list structure.
- Lists are mutable sequences – they can be changed in place.
- They are denoted with square brackets. `l1 = [1, 2, 3, 4]`
- You can create nested sub-lists. `l2 = [1, 2, [3, 4, [5], 6], 7]`
- You can use concatenation. `l1 + l2`
- You can use repetition. `l1 * 4`
- You can use slices. `l1[3:5], l1[:3], l1[5:]`

- `append`, `extend`, `sort`, and `reverse` are built in.
- You can create a list of integers with `range`.

Tuples

- Tuples are like immutable lists.
- Nice for dealing with enumerated types
- Can be nested and indexed.

  Ø `t1 = (1,2,3), t2 = (1,2,(3,4,5))`
- Can use index, slice, and length, just like lists.

  Ø `t1[3], t1[1:2], t1[-2]`
- Tuples are mostly useful when you want to have a list of a predetermined size/length.
- Tuples have constant-time access to elements (fixed memory locations).
- Tuples are very useful as keys for dictionaries.

Dictionaries

- A dictionary is a Python hash table (or associative list)
- They are unordered collections of arbitrary objects.

`d1 = {} – new hashtable d2 = {'spam' : 2, 'eggs', 3}`

- Can be indexed by key: `d2['spam']`
- Keys can be any immutable object.
- Can have nested hash tables

  Ø `d3 = {'spam' : 1, 'other' :{'eggs' :2, 'spam' : 3}}`

  Ø `d3['other']['spam']`
- have `_key, keys(), and values() for k in keys()`
- Typically, you will insert/delete dictionaries with the following:

  Ø `d3['spam'] = 'delicious!' Ø del d3['spam']`

Conditionals

The general format for an if statement is as follows.

```
if<test1> :
<statement1>
<statement2>
elseif:<test2> :
<statement3>
else:
<statement>
```

- Notice the colons after the conditionals.
- Compound statements consist of the colon, followed by an indented block.
- Logical tests return 1 for "True" and 0 for "False."
- "True" and "False" are shorthand
- `and`, `or`, and `not` are available for compound tests.

One of the nice features of Python is the use of list comprehensions (and also tuple, set, and dictionary comprehensions).

`(fe for e in iter if cond)` enumerates the values fe for each `e in iter` for which `cond` is true. The "`if cond`" part is optional, but the "`for`" and "`in`" are not optional. Here, `e` has to be a variable, and iter is an iterator, which can generate a stream of data, such as a list, a set, a range object, or a file. `cond` is an expression that evaluates to either True or False for each `e`, and `fe` is an expression that will be evaluated for each value of `e` for which `cond` returns True.

Python for Artificial Intelligence

This can go in a list, but it can be called directly using `next`. The following shows a simple example, where user input is prepended with>>>.

```
>>> [e*e for e in range(20) if e%2==0]
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

```
>>> a = (e*e for e in range(20) if e%2==0)
>>> next(a)
0
>>> next(a)
4
>>> next(a)
16
>>> list(a)
[36, 64, 100, 144, 196, 256, 324]
>>> next(a)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Notice how `list(a)` continued on the enumeration, and got to the end of it.

Comprehensions can also be used for dictionaries. The following code creates an index for list a:

```
>>> a = ["a","f","bar","b","a","aaaaa"]
>>> ind = {a[i]:i for i in range(len(a))}
>>> ind
{'a': 4, 'f': 1, 'bar': 2, 'b': 3, 'aaaaa': 5}
>>> ind['b'] 3
```

which means that `b` is the 3rd element of the list.

The assignment of `ind` could have also been written as `>>> ind = {val:i for (i,val) in enumerate(a)}`, where `enumerate` returns an iterator of the (index, value) pairs.

9.6.2 Functions as Rst-Class Objects

Python can create lists and other data structures that contain functions. There is an issue that tricks many newcomers to Python. A function uses

the last value of a variable when the function is called, not the value of the variable when the function was defined (this is called “late binding”). This means if you want to use the value a variable has when the function is created, you need to save the current value of that variable. Python uses “late binding” by default, but the alternative that newcomers often expect is “early binding,” where a function uses the value a variable had when the function was defined; this approach can be easily implemented.

Features of Python 11

Consider the following programs designed to create a list of 5 functions, where the ith function in the list is meant to add i to its argument: `2 pythonDemo.py |`. Some tricky examples are as follows.

```
11 fun_list1 = []
12 for i in range(5):
13 def fun1(e):
14 return e+i
15fun_list1.append(fun1)
16 17 fun_list2 = [] 18 for i in range(5):
19 def fun2(e,iv=i):
20 return e+iv
21 fun_list2.append(fun2)
22
23 fun_list3 = [lambda e: e+i for i in range(5)]
24
25 fun_list4 = [lambda e,iv=i: e+iv for i in
range(5)]
26
27 i=56
```

Try to predict, and then test, the output of the following calls, remembering that the function uses the latest value of any variable that is not bound in the function call.

```
pythonDemo.py | (continued)
29 # in Shell do
30## ipython -i pythonDemo.py
31 # Try these (copy text after the comment
symbol and paste in the Python prompt):
32 # print([f(10) for f in fun_list1])
33 # print([f(10) for f in fun_list2])
34 # print([f(10) for f in fun_list3])
35 # print([f(10) for f in fun_list4])
```

In the first for-loop, the function `fun` uses `i`, whose value is the last value it was assigned. In the second loop, the function `fun2` uses `iv`. There is a separate `iv` variable for each function, and its value is the value of `i` when the function was defined. Thus, `fun1` uses late binding, and `fun2` uses early binding. `fun list3` and `fun list4` are equivalent to the first two (except `fun list4` uses a different `i` variable).

One of the advantages of using the embedded definitions (as in `fun1` and `fun2` above) over the lambda is that is it possible to add a `__doc__` string, which is the standard for documenting functions in Python, to the embedded definitions.

9.6.3 Generators and Coroutines

Python has generators which can be used as a form of coroutines.

The yield command returns a value that is obtained with `next`. It is typically used to enumerate the values for a for loop or in generators.

A version of the built-in range with 2 or 3 arguments (and positive steps) can be implemented as follows.

```
pythonDemo.py | (continued)
37 def myrange(start, stop, step=1):
38 """ enumerates the values from start in steps
of size step that are 39 less than stop.
40 """
```

```
41 assert step>0, "only positive steps imple-
mented in myrange"
42 i = start
43 while i<stop:
44 yield i
45 i += step
46
47print("myrange(2,30,3):",list(myrange(2,30,3)))
```

Note that the built-in range is unconventional in how it handles a single argument, as the single argument acts as the second argument of the function.

Note also that the built-in range also allows for indexing (e.g., `range(2, 30, 3)[2]` returns `8`), which the above implementation does not. However, `myrange` also works for floats, while the built-in range does not.

Exercise 1.1. Implement a version of `myrange` that acts like the built-in version when there is a single argument. (Hint: Make the second argument have a default value that can be recognized in the function.)

```
pythonDemo.py | (continued) 49 def ga(n):
50 """generates the square of even nonnegative
integers less than n"""
51 for e in range(n): 52 if e%2==0:
53 yield e*e
54 a = ga(20)
```

The sequence of `next(a)` and `list(a)` gives exactly the same results as the comprehension.

It is straightforward to write a version of the built-in enumerate.

Let's call it

`myenumerate:`

```
pythonDemo.py | (continued)
56 def myenumerate(enum):
57 for i in range(len(enum)):
58 yield i,enum[i]
```

9.7 Useful Libraries

9.7.1 Timing Code

In order to compare algorithms, we often want to compute how long a program takes; this is called the **runtime** of the program. The most straightforward way to compute runtime is to use `time.perf counter()`.

```
import time
start_time = time.perf_counter()
compute_for_a_while()
end_time = time.perf_counter()
print("Time:", end_time - start_time, "seconds")
```

If this time is very small (say less than 0.2 seconds), it is probably very inaccurate, and it may be better to run your code many times to get a more accurate time. For this, you can use timeit (*https://docs.python.org/3/library/timeit.html*). To use timeit to time the call `foo.bar(aaa)` use the following:

```
import timeit
time = timeit.timeit("foo.bar(aaa)",
setup="from __main__ import foo,aaa", number=100)
```

The setup is needed so that Python can find the meaning of the names in the string that is called. This returns the number of seconds to execute `foo.bar(aaa)`100 times. The variable number should be set so that the runtime is at least 0.2 seconds.

You should not trust a single measurement, as that can be confounded by interference from other processes. `timeit.repeat` can be used for running timit a few (say 3) times. Usually, the minimum time is the one to report, but you should be explicit and explain what you are reporting.

9.7.2 Plotting: Matplotlib

The standard plotting for Python is performed with matplotlib (*http://matplotlib.org/*). We will utilize the most basic plotting feature, the pyplot interface.

Here is a simple example that has everything we will use.

```
pythonDemo.py | (continued)
60 import matplotlib.pyplot as plt
61
62 def myplot(min,max,step,fun1,fun2):
63 plt.ion() # make it interactive
64 plt.xlabel("The x axis")
65 plt.ylabel("The y axis")
14 1.  Python for Artificial Intelligence
66 plt.xscale('linear') # Makes a 'log' or 'lin-
ear' scale
67 xvalues = range(min,max,step)
68 plt.plot(xvalues,[fun1(x) for x in xvalues],
69 label="The first fun")
70 plt.plot(xvalues,[fun2(x) for x in xvalues],
linestyle='—',color='k',
71 label=fun2.__doc__) # use the doc string of
the function
72 plt.legend(loc="upper right") # display the
legend
73
74 def slin(x):
75 """y=2x+7"""
76 return 2*x+7 77 def sqfun(x):
78 """y=(x-40)^2/10-20"""
79 return (x-40)**2/10-20
```

```
80
81 # Try the following:
82 # from pythonDemo import myplot, slin, sqfun
83 # import matplotlib.pyplot as plt
84 # myplot(0,100,1,slin,sqfun)
85 # plt.legend(loc="best")
86 # import math
87 # plt.plot([41+40*math.cos(th/10) for th in
range(50)],
88 # [100+100*math.sin(th/10) for th in range(50)])
89 # plt.text(40,100,"ellipse?")
90 # plt.xscale('log')
```

At the end of the code are some commented-out commands you should try in interactive mode. Cut these commands from the file and paste them into Python (and remember to remove the comments symbol and leading space).

9.8 Utilities

9.8.1 Display

In this distribution, to keep things simple and to only use standard Python, we use a text-oriented tracing of the code. A graphical depiction of the code could override the definition of `display` (but we leave it as a project).

The method **`self .display`** is used to trace the program. Any call **`self .display`** (level, to print . . .) where the level is less than or equal to the value for the max display level will be printed. The "to print . . ." part can be anything that is accepted by the built-in `print` (including any keyword arguments). The definition of display is as follows.

```
utilities.py | AIFCA utilities
11 class Displayable(object):
12 max_display_level = 1 # can be overridden in
subclasses
```

```
1.7. Utilities 15 13
14 def display(self,level,*args,**nargs):
15 """print the arguments if the level is less than or equal to the 16 current max_display_level.
17 level is an integer.
18 the other arguments are whatever arguments print can take.
19 """
20 if level <= self.max_display_level:
21 print(*args, **nargs) ##if error you are using Python2, not Python3
```

Note that `args` gets a tuple of the positional arguments, and `nargs` gets a dictionary of the keyword arguments). This will not work in Python 2; it will give an error.

Any class that wants to use `display` can be made a sub-class of Displayable.

To change the maximum display level to say 3, for a class do `Classname.max display level = 3,` which will make calls to `display` in that class print when the value of level is less than or equal to 3. The default display level is 1. It can also be changed for individual objects (the object value overrides the class value).

The values of the max display level by convention are as follows:

0 display nothing,

1 display solutions,

2 also display the values as they change, and

3 also display more details.

9.8.2 Argmax

Python has a built-in max function that takes a generator (or a list or set) and returns the maximum value. The `argmax` method returns the index of an element that has the maximum value. If there are multiple elements with the maximum value, one of the indexes to that value is returned at

random. This assumes a generator of (element, value) pairs, as for example is generated by the built-in enumerate.

```
utilities.py | (continued)
23 import random
24
25 def argmax(gen):
26 """gen is a generator of (element,value) pairs, where value is a real number.
27 argmax returns an element with maximal value.
28 If there are multiple elements with the max value, one is returned at random.
29 """
30 maxv = float('-Infinity') # negative infinity
16 1. Python for Artificial Intelligence 31 maxvals = [] # list of maximal elements 32 for (e,v) in gen:
33 if v>maxv: 34 maxvals,maxv = [e], v 35 elif v==maxv:
36 maxvals.append(e)
37 return random.choice(maxvals)
38
39 # Try:
40 # argmax(enumerate([1,6,3,77,3,55,23]))
```

Exercise 1.3. Change `argmax` to have an optimal argument that specifies whether you want the "first," "last," or a "random" index of the maximum value returned.

If you want the first or the last, you do not need to keep a list of the maximum elements.

9.8.3 Probability

For many of the simulations, we want to make a variable True with some probability.

flip(p) returns True with a probability p, and otherwise returns False.

```
utilities.py | (continued)
42 def flip(prob):
43 """return true with probability prob"""
44 return random.random() < prob
```

9.8.4 Dictionary Union

The function `dict union(d1, d2)` returns the union of dictionaries d1 and d2. If the values for the keys conflict, the values in d2 are used. This is similar to `dict(d1, _ _ d2)`, but that only works when the keys of d2 are strings.

```
utilities.py | (continued)
def dict_union(d1,d2):
""" returns a dictionary that contains the keys of d1 and d2.

    The value for each key that is in d2 is the value from d2 (49), otherwise, it is the value from d1.

    This does not have side effects.
"""
d = dict(d1) # copy d1
d.update(d2)
return d
```

9.9 Testing Code

It is important to test code early and test it often. We include here a simple form of **unit tests**. The value of the current module is in `__name__` and Testing Code 17 runs at the top-level. Its value is `__main__`.

The following code tests `argmax` and `dict_union`, but only when if the utilities are loaded in the top level. If they are loaded in a module, the test code is not run.

In your code, you should do more substantial testing than we do here (in particular, testing the boundary cases).

```
utilities.py | (continued)
56 def test():
57 """Test part of utilities"""
58 assert argmax(enumerate([1,6,55,3,55,23])) in [2,4]
59 assert dict_union({1:4, 2:5, 3:4},{5:7, 2:9}) == {1:4, 2:9, 3:4, 5:7}
60 print("Passed unit test in utilities")
61
62 if __name__ == "__main__":
63 test()
```

CHAPTER 10

ARTIFICIAL INTELLIGENCE MACHINES AND ROBOTICS[1]

10.0 Introduction

This chapter introduces the subject of robotics, which is no longer just a look into the future, but has been developing for many years, is happening now, and will continue to emerge as a part of human life for the unforeseeable future. First, we present the philosophical and pragmatic issues of the field; then we review the history of man trying to create machines that emulate what he does, or recreate himself. There follows a discussion of the technical issues that must be addressed when robots are built. Then a number of applications of robotics are presented. The chapter concludes with a presentation and discussion of the future from the perspective of the "Singularity" as proposed by the great AI inventor, Raymond Kurzweil.

"*In the Year 2525 (Exordium et Terminus)*" was the title of the number one hit song by Zager and Evans in 1969. The song projects what may happen to mankind in the coming millennia. Its thesis is the premise that man will continue to dehumanize himself in the coming years as he succumbs to technological advances.

That is not the subject of this chapter, but it sets the tone for the kinds of considerations for the future of mankind that we are required to look into when seeking advances in robotics. Here, we will guess, dream, imagine, or "look into the crystal ball" to consider how our lives will change. Robots are no longer just a futuristic topic as they were in the early history of AI: They

---

[1] This chapter appeared as Chapter 15 in Artificial Intelligence in the 21st Century, Second Edition by S. Lucci and D. Kopec. Revised and reprinted with permission. ©2016 Mercury Learning and Information. All rights Reserved.

are a reality of life and becoming a greater part of everyday life. Advances in robotics are integrally tied to advances in AI. Let us consider now a small, future robot scene in a middle-class American home. Let us consider what this dialogue entails and what kinds of information, knowledge, and state of the art/technological advances this dialogue entails. Every sentence by both five-year-old Bobby and MrTomR gives a significant clue to the state of the world when this dialogue could take place.

MrTomR is a robot whose task is similar to that of a butler or nanny who must take care of a five-year-old. The parents of Bobby are away at work or on a weekend vacation. MrTomR is doing what he can to simulate the interactions that might take place. Let us analyze what kinds of intelligence MrTomR must have to be able to conduct this dialogue.

- **First, MrTomR suggests that Bobby should have breakfast at a particular time**. That is not a difficult programming task. The only thing that is sophisticated about this is the robot's ability to speak a sentence that is understandable. The sentence can be constructed from a *menu* of commands that MrTomR is programmed to speak in certain trigger situations. Those triggers are that Bobby is home alone being cared for by MrTomR and it is time for breakfast, which Bobby has not yet received (Bobby never gets his own breakfast).
- **MrTomR tells Bobby to sit down**. This indicates that MrTomR understands what it means to be standing, that it has some sense of locomotion. In order to eat breakfast "civilly," Bobby should be sitting at the breakfast table. Furthermore, MrTomR is able to point and understands where Bobby should be sitting. That is already quite a bit of advanced intelligence that MrTomR is demonstrating.
- **MrTomR announces the breakfast menu**. This indicates that MrTomR understands the question from Bobby and can articulately state the answer to it. Bobby asks MrTomR for toast and coffee. MrTomR knows that Bobby is not allowed coffee (although it recognizes that toast was one of the items which comprises part of the menu). As children will do, Bobby is trying to see how far he can go with his caretaker. MrTomR is intelligent enough to be aware of the rules. He responds as an intelligent, experienced human butler or nanny might.

Every chapter and topic in our text to this point is or could be related to the field of robotics. Whether we are delving into search, games, logic,

knowledge representation, production and expert systems, or neural networks, genetic algorithms, language, planning, there are easy and natural connections to robotics. They are not far-fetched or remote. We now consider some of these connections in more detail.

- **Robotics and Search:** From the early days of robotics (in the sense of a machine serving man by trying to accomplish a task), search has been integral to robotics. For example, the kinds of search problems that we addressed in earlier chapters, including, for example, breadth-first search and depth-first search, heuristic search, and search in games, are all typical problems that roboticists must address when building a system. That is, a robot must be programmed to get from point A to point B in the most efficient way, or it must get around some obstacles to reach a destination or goal, akin to dealing with certain kinds of maze problems.
- **Robotics, Logic, and Knowledge Representation:** It goes without saying that robots and logic go hand-in-hand. The kinds of logical problems presented earlier are the foundations of robotics, and the methods are the building blocks for constructing sound robotic systems. Before any AI system is built, consideration must be made of how the elements of that system will be represented. Whether an agent-based approach will be used, swarm intelligence, trees, graphs, networks, or other approaches, these considerations are fundamental in robotic systems.
- **Production Systems and Expert Systems:** Production systems as the foundations of expert systems are closely tied to control systems, which are the basic foundation of robotic systems. Tasks such as directing a robot across a factory floor or getting a robot to pick up packages in an Amazon factory show what kind of tasks need to be accomplished in order to be able to accomplish a bigger task (hierarchy). These are examples of how robots may depend on production systems and expert systems. Furthermore, the expertise that humans have developed in various spheres (e.g., machinist tools, factory assembly lines, blending of colors for paint generation, or choosing the right packaging) are natural arenas for production systems comprising expert systems.
- **Fuzzy Logic:** Even in the robotic world, there are outcomes that are not only black and white or “yes” and “no,” but “to a certain degree

of." For example, a robot may encounter resistance along its path to a goal, and thereby stumble. The robot must persist in its goal of accomplishing an objective. In other words, even the robot world is not just discrete, but it depends on certain "degrees of freedom" with variations on the degrees of attributes, rather than outcomes which are just "on" or "off" or "yes" or "no."

- **Machine Learning and Neural Networks:** As the sophistication of these AI methods has improved, opportunities for their use in robotics have emerged. The Google Car comes to mind as a premier example.
- **Techniques such as Genetic Algorithms, Tabu Search, and Swarm Intelligence:** These techniques are naturally utilized by robotic systems, especially when they must work in groups. For example, these techniques are important for the simulation of crowd behavior or walking on New York City streets. Robots use these techniques for simulating people rushing to their commutes while avoiding other people who are approaching them or are otherwise in their paths.
- **Natural Language Understanding and Speech Understanding:** We continually see improvements in how machines (robots) will replace humans in ever-more advanced tasks which involve language and speech understanding. Hence, progress in these disciplines is integral and important to robotics. The issues and factors involved (for example, semantics, syntax, accent, and inflection) are enormous.
- **Planning:** This has always been a subfield of AI that is strongly associated with robotics. We have discussed planning in robotics, which involves how a program should proceed in accomplishing a task or set of tasks.

We will now discuss some of the challenges for robotics and why it is both a promising and very difficult field. In constructing robots, we are addressing the issues that make mankind unique. The challenges are dependent on how ambitious we want to be. That is, do we only wish the robot to be mobile? Do we wish the robot to perform tasks akin to the original definition of the word from the play by the Czech playwright Karel Čapek entitled *R.U.R.* (1921) where it was first introduced? In the Czech language *robota* means "labor" or "work," but in the context of the play it meant "slavery" or forced "labor." Or do we have much greater ambitions for robots: that they not only be able to aid man, but emulate

him, enhance him, and be recreated/replaced in his image? Hence, we have robots performing mundane tasks that not only people have to do (e.g., vacuum, as with the IROBOT Roomba), but also performing surgery, entering dangerous places, carrying heavy loads, and even driving cars safely without humans! In the new millennium, robots are starting to perform such difficult tasks better than humans can, that is, more accurately, more quickly, and more efficiently, thereby freeing people from the dangers and challenges of such tasks. Robots are taking on more tasks that for hundreds of years humans had customarily performed themselves. Robots are even being built to simulate recreational tasks, such as playing bridge and soccer.

These advances have been enabled by improvements in locomotion, machine vision, machine learning, planning, and problem solving. In the future, we will likely entrust robots with an increasing number of decisions of a vital nature to humans. Some argue that there are limitations to what robots will be able to accomplish until we understand ourselves better. Marvin Minsky discussed this perspective in his relatively early work on robotics. For nearly thirty years he, Doug Lenat, and others have been trying to address the problem of *common sense knowledge.* He addresses questions such as *How do children really learn?*, *What turns short-term memories into long-term memories?*, and *How is knowledge organized for people?* During the past 25 years, it has become evident that robots are and will continue to be able to take advantage of tremendous advances in natural language processing and speech understanding. As already mentioned, such advances, along with the possibility that machines will be built with intelligence on a par with or beyond our own, will pose difficult philosophical and practical questions. One thing is clear: despite the recognizable pros and cons of building highly intelligent robotic systems, there is no turning back.

10.1 History: Serving, Emulating, Enhancing, and Replacing Man

The history in "*Man Makes Man*," by T. A. Heppinger, is much richer and longer than one might imagine. We will consider the historical aspects of robotics from a number of perspectives, including:

- Robot Lore
- Early Mechanical Robots

- Robots in Film and Literature
- Early Twentieth-Century Robots.

10.1.1 Robot Lore

One of the earliest examples of robot lore is the story of the brilliant thirteenth-century English clergyman-scientist-philosopher, Friar Roger Bacon, who wanted to build a wall of brass to protect England against invaders. To accomplish this, he proposed a "brass head" to explain how such a wall should be built. That head was watched for three weeks, and it was only after the friars had watched carefully over the head that it spoke, "Time is." A half hour later, it said "Time was." Another half hour later, "Time is past." Certainly, it is just a tale, but it may have been the inspiration for the leading medieval physician Paracelsus to suggest how an entire living being, a "homunculus," could be built:

> Let the semen of a man putrefy by itself in a hermetically sealed glass with the highest putrefaction of horse manure for forty days, or until it begins at last to live, move and be agitated, which can easily be seen. After this time it will be in some degree like a human being…If now after this, it will be every day nourished and fed cautiously and prudently with the Arcanum of human blood, and kept for forty days in the perpetual and equal heat of horse manure, … This we call a homunculus and it should afterwards be educated with the greatest care and zeal, until it grows and begins to display intelligence.

Although this idea was based on "alchemical lore," the story reminds us of the vast advancements science and the medical profession have been made through the centuries.

Another legend of a man-made man is the lore of golem from the sixteenth century, several decades after Paracelsus. In the Talmud, the word "golem" means "incomplete" or "malformed," such as an embryo or the shapeless mass of dust from which Adam was created. It is said that around the year 1550, Elijah of Chelm created an artificial man, called a golem, with the Name of God corresponding to the four letters YHWH. This golem became a monster that threatened the world until the sacred name was removed.

Thirty years, later there was another golem story. This one centered around the Rabbi Judah ben Loew, Chief Rabbi of Prague. The Rabbi was known as a sober figure who was friends with the famous astronomers Tycho Brahe and Johannes Kepler. To protect his people, the Rabbi is said to have gone to the River Moldau with two assistants where they fashioned from clay a human figure (see Figure 10.1).

FIGURE 10.1 Clay Golem

The story continues:

> One assistant circled the figure seven times from left to right. Loew pronounced an incantation, and the golem began to shine like fire. The other assistant then began his own incantation which circling seven times from right to left. The fire went out, hair grew on the figure's head, and nails developed on its fingers. Now it was Loew's turn to circle seven times, as the three of them chanted words from Genesis. When Loew implanted the Holy Name upon its forehead, the golem opened its eyes and came to life…

Although the golem was unable to speak, it had superhuman power, and thus was useful in defending the Jews of Prague against the Gentiles. The golem was also Loew's servant and worked as a janitor within the temple, with an allowance for rest on the Sabbath. Only Rabbi Loew was able to control the golem, but eventually it ran amok, attacking its creator. The golem's reign

of destruction ended when Rabbi Loew tricked it into kneeling before him and plucked the sacred name from its forehead—and magically the golem was again reduced to clay. These three legends—the brass head of Bacon, the homunculus of Paracelsus, and the golem of Rabbi Loew—share in common the notion of a savant (a respected, accomplished man of intelligence) creating something in the form of a man that will have the power of a man. The famous story *Frankenstein*, authored by Mary Shelley in 1817, is actually a statement on the dangers of letting technology run amok; it is noteworthy that the story, by analogy, is quite consistent with the story of the golem some four centuries earlier.

10.1.2 Early Mechanical Robots

Perhaps the first accepted mechanical representation of man was the Strasbourg cock, a cast-iron rooster built in 1574, intended to be a reminder of St. Peter's denial of Jesus (Figure 10.2). At noon daily, it opened its beak, stretched out its tongue, flapped its wings, spread out its feathers, raised its head, and crowed three times. Used until 1789, it served as an inspiration to Hobbes, Descartes, and Boyle as an example of what might someday be achievable by machinery.

FIGURE 10.2 Strasbourg Cock

In the mid-eighteenth century, there followed the inventions of Jacques de Vaucanson, who created various artificial humans and animals that were quite realistic. One of his most famous inventions was a 1738 mechanical duck which amazed in its ability to quack, splash around in water, eat, drink,

and excrete (Figure 10.3). Vaucanson also built two androids in human form that played musical instruments (Figure 10.3). One played the flute and the other the drums. What most impressed people was that the flutist was actually playing, rather than producing sounds from a hidden place. The flutist's breath came directly from its mouth by means of a set of bellows. Lip movements were controlled by a mechanism. The flute, a standard instrument, made sounds via finger motions over holes—as would be performed by a human. Hence, in the early history of robotics, this was a considered a landmark, in that the flute was considered an instrument of skill that only a small number of people could play well. Here, we had the first mechanical device that performed a learned skill better than most people.

FIGURE 10.3 Vaucanson's Duck, Flutist, and Drummer

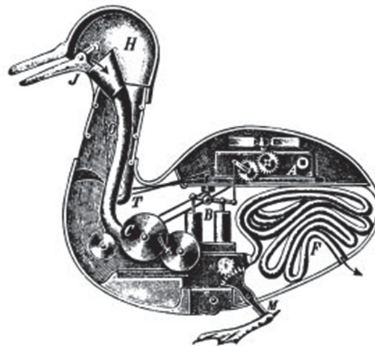


FIGURE 10.3(a) Vaucanson's Duck with Internal Mechanisms

The next rather well-known example of a man emulating a man was somewhat of a hoax that fooled Europeans for many years. The Turk was a contraption built by Baron Wolfgang von Kempelen in the Austro-Hungarian Court in 1769. Purportedly, a midget Polish chess master was inside a box with gears and cogs which played chess. It featured "a mannequin in the form of a Turk, with turban and handlebar mustache, seated behind a wooden cabinet" (Figure 10.4). The Turk wowed audiences across Europe for many years in that it played chess well and could not be fooled with illegal moves. It was also impressive in the fact that it was the first time that people believed that the distinction between man and machine had been blurred. Eventually the Turk was safely transported to a Philadelphia museum, which, in the mid-twentieth century, unfortunately burned down.

FIGURE 10.4 Baron von Kempelen's "The Turk"

Between 1770 and 1773, the father and son pair, Pierre and Henri-Louis Jaquet-Drov, developed and demonstrated three amazing human-like

figures known as the Scribe, the Draftsman, and the Musician (Figure 10.5). All three operated via clockwork using an intricate array of cams. The Scribe and the Draftsman were in the shape of young boys, elegantly dressed. The Scribe was capable of dipping a quill pen in an inkwell and then writing up to forty letters. The Scribe's hand, controlled by a cam, could move in any of three directions to form one letter. Levers on a disk were used for control, and the Scribe could then write any desired text. His brother, the Draftsman, could produce drawings of Louis XV and similar figures including, for example, a battleship. The eyes of these androids demonstrated an attentive attitude while at work by moving their eyes accordingly.

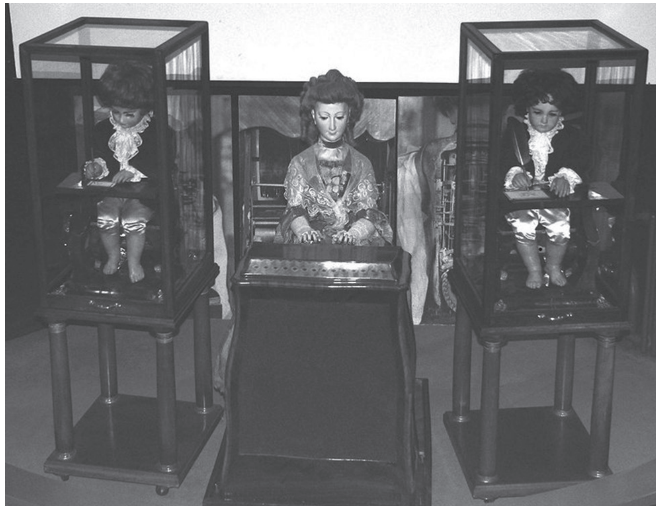


FIGURE 10.5 The Scribe, the Draftsman, and the Musician Developed By Pierre and Henri Louis Jaquet-Drov

The Musician, another Jaquet-Droz android, resembled a girl of 16, wearing a powdered wig and a dress appropriate for the court of Vienna. She played the organ well, with convincing eye and body movements that made her seem alive. The end of a performance was accompanied by a bow. The Jaquet-Droz androids found permanent homes in the Muséed'Art et d'Histoire in Neuchatel, Switzerland. The Draftsman, with its design of a battleship, found its way into the Franklin Institute in Philadelphia. In each android, one can see the innovation and engineering which led to modern industrial robots. The differences are in form and the modern use of hydraulics and programming instead of springs, cams, and clockwork mechanisms.

There followed the industrial revolution, and one of its artifacts was a mechanism devised by James Watt (credited with the development of the first practical steam engine circa 1783). In 1788 Watt devised a "flywheel governor" featuring two whirling balls that were able to swing outward via centrifugal force. It was linked to a steam engine whereby the outward swing of the flyballs measured the engine's speed; furthermore, using another linkage, the outward swing controlled a value that maintained its present speed. In essence, this comprised the world's first feedback-control mechanism. In 1868 James Clerk Maxwell (who discovered Maxwell's equations in electromagnetism) published "On Governors," the first systematic study of feedback control. This turned out to be an essential element of robots in the twentieth century.

In 1912, the automatic, mechanical, chess-playing machine built of gears and cogs by Leonardo Torres y Quevedo could play the elementary endgame King and Rook vs. King via an explicit set of rules to deliver checkmate in a limited number of moves regardless of the starting position. This was believed to be the first machine capable of not only handling information but being able to make decisions based on this information.

10.1.3 Robots in Film and Literature

The play *R.U.R.* ("Rossum's Universal Robots") is about robots who have been designed and used as general purpose laborers. They are devoid of human feelings and emotions, but are used as soldiers in war. In the play, it turns out that an associate at R.U.R. discovers how to add pain and emotions to the robots. Hence, the robots rebel against their human masters, virtually exterminating them. However, they are unable to maintain the level of production of themselves. A final touch is when two robots fall in love, suggesting the coming of a new Adam and Eve.

We must bear in mind the time when *R.U.R.* appeared, which was just after the end of World War I. It was also a statement on the dangers of technology which, with the invention of machine guns, submarines, and poison gas, had turned the war into a bloodbath with mass carnage and massacre. Another work in the same vein was the 1926 classic movie *Metropolis* by Fritz Lang, a very popular and highly respected German filmmaker. It was based on a book written by his wife Thea Harbou. *Metropolis* focuses on the wretched lives of workers who live beneath a city. Its robot is a labor agitator, Maria, who assumes the appearance of a leader whom the workers can

trust. It turns out that Maria leads the robots to self-destruction, and they burn her at the stake, where she turns to metal.

Regarding contributions to robotics in film, arts, and literature, the work of Isaac Asimov must be introduced. In 1942 as a young science fiction writer, he contributed to *Galaxy Science Fiction* the story "The Caves of Steel," where he first presented the oft-repeated *Three Laws of Robots*:

1. A robot may not injure a human being, or through inaction allow a human being to come to harm.
2. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Many decades passed before Asimov's ideas captivated the world in such films as *Forbidden Planet* (1956) and the *Star Wars* Trilogy (1977, *Star Wars*; 1980, *The Empire Strikes Back*; and 1983, *The Return of the Jedi*).

10.1.4 Twentieth-Century Robots

In the twentieth century, a number of robotic systems were built. Many were successful. In the 1980s, robots started to become commonplace in factories and industrial settings. Here, we limit our discussion to robots that were particularly instrumental to research and progress in the field.

10.1.4.1 Biomimetic Systems

In this section, we present two biomimetic systems that were very important to the progress in robotics research. One field that has not been discussed in our text to this point, considered an early forerunner to AI, is the field of cybernetics, which is the study and comparison of communication and control processes in biological and artificial systems. The person most credited for defining and doing seminal research in this field is Norbert Wiener at MIT. This field combined theories and principles from neuroscience and biology with those from engineering, with the goal of finding common properties and principles in animals and machines. Matarić notes that "a key concept of cybernetics focuses on the coupling, combining, and interaction between the mechanism or organism and its environment." Such interactions are necessarily complex, as we shall soon

see. Her definition of a robot is as follows: "an autonomous system which exists in the physical world, can sense its environment, and can act on it to achieve some goals." [2]

Given this definition, Prof. Matarić calls William Grey Walter's *Tortoise* the first robot that was built with the underlying goals of cybernetics. Walter (1910–1977) was born in Kansas City but lived and was educated in Great Britain. He was a neurophysiologist with a strong interest in how the brain works, and he discovered the theta and delta waves that are produced during sleep. He built machines with animal-like behavior to study how the brain works. Walter was convinced that even organisms with very simple nervous systems could exhibit complex and unexpected behavior. Walter's robots were distinct from the robots that preceded them in that they behaved in unpredictable ways, had reflexes, and in their environments were able to avoid repetitious behaviors. The tortoise consisted of a hard plastic shell with three wheels (Figure 10.6). Two wheels were for forward and backward motion, while the third was for steering. Its "sense organs" were extremely simple, consisting of only a photoelectric cell to provide sensitivity to light and surface electric contacts that served as touch sensors. A telephone battery provided power, while the shell provided some degree of protection against physical damage.

FIGURE 10.6 Grey Walter's Tortoise, the First Recognized Robot

[2] Note that an autonomous robot acts on the basis of its own decisions, and is not controlled by a human.

With these simple components and a few others, Grey Walter's *Machina Speculatrix* ("machine that thinks") exhibited the following behaviors:

- find the light
- head toward the light
- back away from bright light
- turn and push to avoid obstacles
- recharge its battery.

The turtles were the earliest examples of *artificial life* or "Alife." Their variety of complex, unprogrammed behaviors were early examples of what we now call *emergent behavior*.

Valentino Braitenberg was a German scientist who was inspired by Grey Walter's work. In 1984, he published a book entitled *Vehicles,* long after the idea of cybernetics was developed and was considered a separate discipline of study. The book presents a series of ideas (or thought experiments) demonstrating how simple robots (which he called "vehicles") can produce behaviors which appear very human and lifelike. Although Braitenberg's vehicles were never built, they proved inspirational for roboticists.

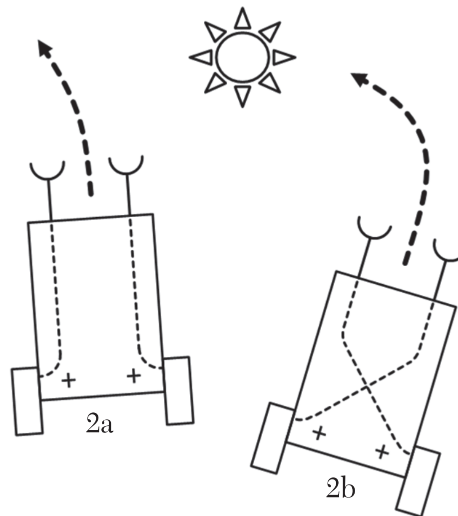


FIGURE 10.7 Example of Braitenberg's Vehicles. Vehicle 2a moves toward a source of light while vehicle 2b moves away from a source of light.

These started with a single motor and light sensor. Gradually, they increased in complexity to several motors and sensors, and the exploration of the various permutations of sensors between them. The sensors were connected to the motors. Therefore, a light sensor could be connected directly to the wheels of a vehicle; as the light became stronger, the robot would move faster toward the light. This is called *photophilic* attraction or "loving light." The connections could be reversed so that the robot would move more slowly and hence be *photophobic*, or exhibit a "fear of light."

Furthermore, akin to the concept of neural networks, the connections between the sensors and motors, whereby stronger sensor input produced stronger output, were called *excitatory connections*. Conversely, sensory inputs that weakened the motor as they got stronger were called *inhibitory connections*. Again, the inspiration came from biological neurons and their excitatory and inhibitory connections. Continuing with this analogy, it is fairly evident how variations in these connections between sensors and motors can result in a variety of behaviors. Braitenberg's book describes how such simple mechanisms can be used to store information, build a memory, and even achieve learning.

10.1.4.2 Recent Systems

Artificial Intelligence research progressed in many arenas during the twentieth century, a point we have described throughout this text. Research incorporating what had and was being learned in the various disciplines of AI was focused at three institutions: The Massachusetts Institute of Technology (MIT), Stanford, and SRI International (then known as the Stanford Research Institute).

Shakey, at SRI (1966–1972), was the first general-purpose mobile robot able to reason about its own actions. Shakey (Figure 10.8) was designed to analyze commands and break them into a series of actions necessary to perform. Its basis was research in computer vision and natural language processing. Charles Rosen was the project manager; contributors included Nils Nilsson, Alfraed Brain, Sven Wahlstrom, Bertram Raphael, and others. *STRIPS* (Stanford Research Institute Problem Solver) is a premier example of an automated planning robot system. It was developed by Richard Fikes and Nils Nilsson in 1971 at SRI International. MIT has a long history of research and contributions to the field of AI and robotics, including robots in many environments such as space and sea, and those that exhibit locomotion.

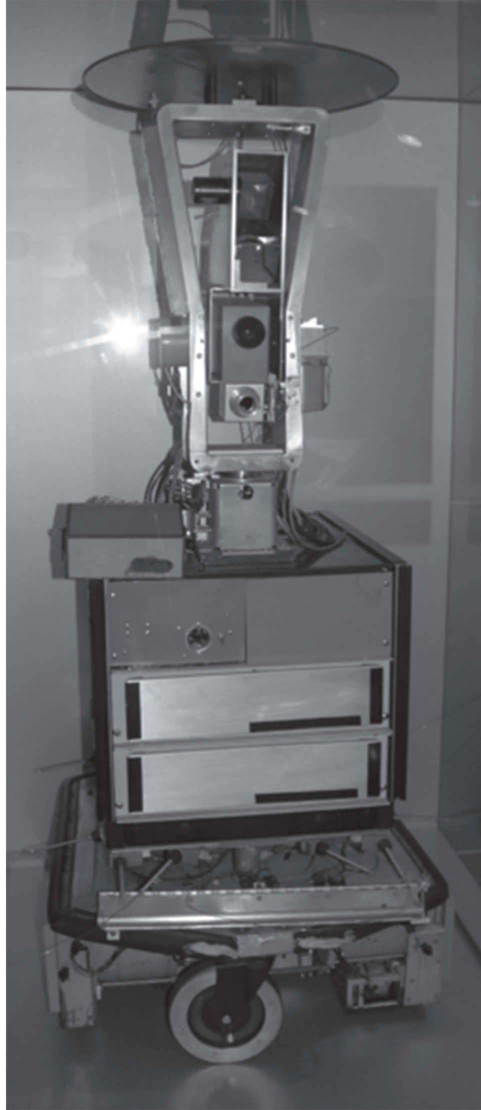


FIGURE 10.8 SRI's Shakey

There are many more examples than we can give justice to here. Table 1 presents diverse robot systems that have been built during the past 55 years or so. Their increasing sophistication, capabilities, and purpose are noteworthy. Problems which involve locomotion in open terrain are much harder to solve than those in well-defined spaces or environments.

Table 1 Summary of Robotics Projects from 1960–2010

	SYSTEM NAME	YEAR	CREATOR	INSTITUTION / COMPANY	FEATURES	FOOTNOTE
1	Stanford Cart	1960–1980	James Adams	Stanford University	Able to move around obstacles using a camera	[9]
2	Freddy	1969–1971	Donald Michie	University of Edinburgh	Assembles blocks by using its camera	[10]
3	WABOT-1	1970–1973	Waseda University	Waseda University	First full-scale anthropomorphic robot. Able to communicate with a person in Japanese. Could measure distances with receptors.	[11]
4	FAMULUS	1973	KUKA Robotics	KUKA Robotics	Material handling, i.e., moving parts and materials in factories	[1]
5	Silver Arm	1974	David Silver	MIT	Small parts assembler that reacts to feedback from touch and pressure sensors.	[2]
6	WABOT-2	1980–1984	Waseda University	Waseda University	Able to read a musical score and play the organ, and speak to people	[11]

	SYSTEM NAME	YEAR	CREATOR	INSTITUTION / COMPANY	FEATURES	FOOTNOTE
7	Omnibot	1980s–2000	Tomy	Tomy	Carry light objects with arms, had a tray to carry objects	[12]
8	Direct Drive Arm	1981	Takeo Kanade	Carnegie Mellon University	Robotic arm that could move more freely and smoothly	[3]
9	Modulus Robot	1984–1990s	Massimo Giuliana	Sirius	Domestic household robot, household applications	[13]
10	Big Dog	1986–Present	Buehler, Martin	Boston Dynamics	Quadruped walking, pack mule	[7]
11	Kismet	1990s	Cynthia Breazeal	MIT	Low-level feature extraction system, motivation system, motor system	[30]
12	COG	1993–Present	Rodney Brooks	MIT	Humanoid, emulates human thought	[4]
13	The Walking Forest Machine	1995	PlusTech Ltd.	PlusTech Ltd.	Walking backwards, forwards, sideways, and diagonally in uneven terrain	[5]
14	Scout II	1998	Ambulatory Robotic Laboratory	Ambulatory Robotic Laboratory	Quadruped walking	[5]

	SYSTEM NAME	YEAR	CREATOR	INSTITUTION / COMPANY	FEATURES	FOOTNOTE
15	AIBO	1999	Sony	Sony	Quadruped walking, pet	[6]
16	Hiro	1999–2010	Kawada KK	Kawada Industries INC.	Runs real time Linux QNX	[14]
17	CosmoBot	1999–Present	Dr. Corinna Lathan with Jack Vice	AnthroTronix, Inc.	Live Play, Simon Says, playback	[15]
9	ASIMO	2000–Present	Honda	Honda	Humanoid upright, two-legged walking	[5]
20	Anybots	2001–Present	Trevor Blackwell	ANYBOTS	Virtual presence systems	[16]
21	Inkha	2002–2006	mat and mrplong	King's College London	Camera to track Human movement, speaks periodically about facts	[17]
22	Domo	2004–Present	Jeff Weber and Aaron Edsinger	MIT	Perception, learning, manipulation	[18]
23	Seropi	2005–Present	KITECH	KITECH	Human-friendly working space guidance	[19]
24	Wakamaru	2005–Present	Mitsubishi Heavy Industries	Mitsubishi Heavy Industries	Reminder, emergency call, Linux operating system and connects to the internet	[20]

	SYSTEM NAME	YEAR	CREATOR	INSTITUTION / COMPANY	FEATURES	FOOTNOTE
25	Enon	2005–Present	Fujitsu	Fujitsu Corporation	Self-guiding, limited speech recognition and synthesis	[21]
26	MUSA	2005–Present	Young Bong Bang	Seoul National University	Fight using kendo	[22]
28	BEAR	2005–Present	Vecna Technologies	Vecna Technologies	Six feet tall, hydraulic upper body lifts 500 lbs, steel torso, maximum hydraulic exertion of 3000 psi	[23]
29	Issac	2006–Present	IssacTeam	Politecnico di Torino	Offers many solutions oriented to automation industry	[24]
30	Willow Garage	2006–Present	Scott Hassan	Willo Garage Inc.	ROS (Robot Operating System) developing hardware and software for robotics applications	[25]
31	RuBot II	2006–Present	Pete Redmond	Mechatrons.com	Solves Rubik's Cube	[26]
32	KeepOn	2007	Kozima, Hideki	Miyagi University	Responds to emotions and dances	[8]

	SYSTEM NAME	YEAR	CREATOR	INSTITUTION / COMPANY	FEATURES	FOOTNOTE
33	Topio Dio	2008–2010	TOSY Robotics JSC	Automatica	Remote control via wireless, integrate 3D vision via 2 cameras, 3D operation space, processes pre-defined images, detects obstacles by ultrasonic sensor, three-wheeled base with omni-directional and balanced motion	[27]
34	Phobot	2008–Present	Students	University of Amsterdam	Exhibits behavior that mimics fear and overcoming it by graded exposure	[28]
35	Salvius	2008–Present	Gunther Cox	Salvius Robot	Modular design, constructed using recycled materials and open source	[29]
36	ROBOTY	2010–Present	Hamdi M. Sahloul	Engineering University of Sana	Robot capable of playing chess	[30]

10.2 Technical Issues

As we alluded to at the beginning of this chapter, the technical issues for developing robots are immense, and in one way or another, they depend on

how ambitious and sophisticated one's goals are for a robot's capabilities. In essence, working in robotics is a multifaceted form of problem solving.

By analogy, let us consider the problems a human faces when entering a shopping mall and attempting to find a particular store in that mall. For a human, there are fairly straightforward steps and questions to ask in order to find the store you are looking for. You might look for the mall directory, ask people at information desks, see store managers who might be familiar, or use information sources, such as the internet and phone apps. If we have previously visited the store, we may even have some memory of where this store is located in the mall, i.e., which floor, neighboring stores, and special features. Now let us consider what the challenges would be for a mobile robot to find a particular store in the mall. One solution would be for the robot to simply follow locomotion directions, for example, go straight for .2 miles, turn left, and go .1 miles. Or, it may be told to take an elevator up a floor. The means of communicating directions to the robot could vary in format. The directions could be sensory, auditory, written, or visual. The differences in how diverse robots could handle this problem and related problems is the subject of this section. It is important to bear in mind that whatever the solution method chosen for a robot to find the goal store in question, every aspect of the solution must be considered by the robot's developers and programmers. Its locomotion, its perception of obstacles, landmarks, and goal points, must all be considered in detail by human developers. That is why the possibility of employing machine learning in robots represents such an important advance in the field. If a robot can learn, then almost anything seems possible.

The early history of robotics focused on locomotion and vision (known as machine vision). Closely aligned to the discipline were problems of computational geometry and planning. In the past few decades, the possibilities for robots have become more of a reality, with domains such as linguistics, neural networks, and fuzzy logic being more integral to the research and progress in robotics.

10.2.1 Robot Components

Before we delve into the typical problems facing roboticists, we feel it is important to consider the components which comprise a typical robot. These include:

1. the physical body or embodiment
2. sensors for perceiving the environment

3. effectors and actuators to enable action
4. controller(s) to enable autonomous behavior.

We will consider the requirements for each of these four components one by one.

1. Having a **physical body** means that a robot may conceivably develop a sense of self; that is, it can consider such questions as *Where am I?*, *What is my state (or condition)?*, and *Where am I trying to go?* This also means that it is subject to the same physical laws that we live by, it takes up a certain amount of space, and also needs energy to perform functions, such as sensing and thinking.[3]
2. **Sensory perception** is a requirement for a real robot. It must be able to perceive the environment, react to it, and act on it. Usually such reactions involve movement, and that is a fundamental task for robots. As is common in computer sciencehardware, states of electronic systems are often represented by 1s and 0s (binary digits). Depending on the number of these sensors involved, there are 2^N combinations of perceptions (sensor states) that a robot can have. The sensors are used to represent the internal and external state of a robot. The internal world refers to the robot's own state as it perceives it. The external state refers to how the robot perceives the world it is interacting with. Representation of internal and external states (or ***internal models***) of robots is an important design issue.
3. ***Effectors and Actuators:*** Effectors are the components that enable a robot to take action. They use underlying mechanisms, such as muscles and motors, to perform various functions, but mainly use them for locomotion and manipulation. Locomotion and manipulation comprise two major subfields of robotics. The former is concerned with movement (i.e., the legs of robots), while the latter is concerned with handling things (i.e., the arms of a robot).
4. ***Controllers*** are the hardware and/or software that enable a robot to be autonomous and hence are the devices that control their decisions (or their "brains"). If robots are partially or fully controlled by humans, then they are not autonomous.

[3] It seems worthwhile mentioning that one of the basic elements of life is considered to be motion, or the ability to move. So when considering the possibility of machines moving, we are anointing them with one of the most basic accepted ingredients of being alive.

It is noteworthy that there are a number of important analogies between power supplies for robots and people. Humans need food and water to provide energy for their bodies, for locomotion, and for brain functioning. Robots' brains are not *presently* so developed and therefore need power (usually provided by batteries) for locomotion and manipulation. Now consider what happens when *our* power supply goes down (i.e., when we are hungry or require rest). We become incapable of making good decisions, make mistakes, and may act poorly or strangely. The same thing can happen to robots. Hence, their power supply must be isolated, protected, and efficient, and they should ***degrade gracefully***. That is, robots should be able to replenish their power autonomously and without totally breaking down.

Effectors are any device on a robot that has an effect on the environment. In the world of robotics, they may be arms, legs, or wheels, that is, any robot component that can be used to have an effect on the environment. Actuators are the mechanisms that enable effectors to perform their tasks. Actuators may include electric motors, hydraulic or pneumatic cylinders, or temperature-sensitive or chemically-sensitive materials. Such actuators may be used to activate wheels, arms, grippers, legs, and other effectors. Actuators may be passive or active. Although all actuators require energy, some may be passive and require direct power to operate, while others may be passive and use physical laws of motion to conserve energy. The most common actuators are motors, but there may also be hydraulics using fluid pressure, pneumatics using air pressure, photoreactive material (responding to light), chemically reactive materials, thermally reactive materials, or piezoelectric materials (materials, usually crystals, that create electric charges when pushed or pressed).

10.2.1.1 Motors and Gears

The invention of the electromagnet by Joseph Henry in 1831 is considered by many the greatest invention since man created the wheel. Closely tied to this, and of equal significance, is the invention of the electric motor in 1861 by Etienne Lenoir. The association and significance of motors to power for affecting motion is paramount. Equally significant, therefore, is the importance of motors to robotics.

Robots will typically use DC motors comprised of electromagnets and current to produce magnetic fields which turn the shafts of the motors. Motors must be run by a voltage appropriate for the task(s) so as not to wear them down. DC motors are preferred, as they provide constant voltage, drawing

current at an amount proportional to the work being done. Motors which run into high resistance (e.g., a robot runs into a wall that does not move) will eventually stall after running out of power. Recall from physics that

V (voltage) = I (current) × R (resistance).

Hence **V/I = R or voltage** is proportional to the resistance. However, **work = force x distance**. In the case of the robot stuck against a wall, the distance becomes very small (or zero) and thus, despite a high power (voltage), the work actually performed is very little or none at all. Perhaps an easy analogy to demonstrate this idea is a car that is stuck in the snow with its motor revved up and its wheels spinning. If this goes on for too long, the car too will eventually stall.

The more current (electrons transferred per unit of time, measured in Amperes) that a motor produces, the more torque (rotational force) is produced by the motor shaft. Hence, the power of a motor is the product of its torque and the rotational speed of the shaft.[4] Most DC motors operate at the speed of 3,000 – 9,000 revolutions per minute (rpm). This means they produce high speeds but low torque. However, robots are usually required to perform tasks that require little rotational speed and more torque, such as turning wheels, transporting loads, and lifting.

The problem with robot motors' need for more torque rather than rotational speed is alleviated by understanding and cleverly applying the theory of how gears work. As with robotics in general, simple ideas that are well-understood can be compounded to develop more complex working systems. Small gears will turn more quickly, but are less powerful. Larger gears turn more slowly but are more powerful. This is the principle of gears on which multi-gear / multi-speed bicycles are based. So if a smaller gear drives a larger gear, more torque is created in the ratio of the size of the smaller gear to the larger gear (in terms of the number of teeth). Such paired gears are called ***ganged gears***. Figure 10.9 illustrates this principle with ganged gears called a ***compound gear train***. For example, if the input-output ratio of one axle is 40 to 8, it would be reduced to 5 to 1. A second pair of meshed

[4] A colleague of the authors was known to have purchased a 1999 Cadillac in 2004. Shortly after he purchased it, a check engine error came up on the dashboard. It was identified as a problem with the torque converter, which is part of the transmission. The transmission was rebuilt, and this problem was allayed for some 100,000 miles before the torque converter problem did actually present itself after some 15 miles of continuous driving, when the car could not maintain its highway speed.

gears could have the input of an 8-tooth gear to drive a 24-tooth gear. This converts to a 3-to-1 ratio. Notice that the 8-tooth gear of the second axle may be on the same axle as the 40-tooth gear of the first pair. This gives a ganged gear ratio of 5 to 1 × 3 to 1, which is 15 to 1. Hence, the first axle (with smaller gears) must turn 15 times for the second axle to turn once. Therefore, more torque (in the ratio of 15:1) has been created for the second axle.

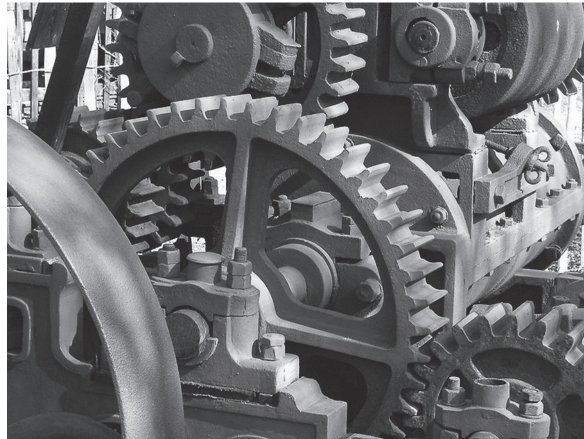


FIGURE 10.9 Ganged Gears

Another concept in robot motors is the servo motor. These kinds of motors (or "servos" for short) are motors that can rotate in such a way that their shaft reaches a specific position. They are common in toys, and are used for adjusting steering in remote control cars or wing positions in remote control planes. Servo motors are made from DC motors with the following additional components:

1. gear reduction for torque
2. a position sensor for the motor shaft to tell how much the motor is turning and in what direction
3. an electronic circuit to control the motor, telling it how much to turn and in what direction .

Electronic signals in the form of a series of pulses will tell the motor shaft how much to turn, typically within a range of 180 degrees. Pulse-width modulation is a method of controlling the amount that the motor's shaft will turn by the length of the pulse; the larger the pulse, the larger the turn

angle of the shaft. This is usually measured in units of microseconds and therefore quite precise. Between pulses, the shaft is stopped.

10.2.1.2 Degrees of Freedom

A common notion in the field of robotics is the concept of the degrees of motion for an object. These are a means of expressing the various types of motion available to a robot. As an example, consider the degrees of freedom of motion (called ***translational degrees of freedom***) of a helicopter. There are six degrees of freedom (DOF) which are usually used to describe the possible motions of a helicopter: the roll, pitch, and yaw (Figure 10.10). Roll means rolling from side to side, pitch means angling up or down, and yaw means turning left or right. An object like a car (or a helicopter on the ground) has only three DOF (the vertical motion is lost), but only two are controllable. That is, a car on the ground can only move forward and backward (via the wheels) and turn left or right via its steering wheel. If a car could move directly left or right (say by turning each of its wheels 90 degrees), that would add another DOF. Hence, with more complicated robot motions, such as arms or legs trying to move in various directions (as is possible in human arms with a rotator cuff), the number of DOF is an important issue.

FIGURE 10.10 A Helicopter and Its Degrees of Freedom (Source: *http://commons.wikimedia.org/wiki/Helicopter#mediaviewer/File:Bell_407_(D-HBEN).jpg*)

10.2.2 Locomotion

This is probably the oldest problem in robotics. Whether you are trying to get a robot to play soccer, land on the moon, or work under the ocean, the most fundamental issue is locomotion. How does the robot move? What are its capabilities? The typical actuators which come to mind include:

- wheels for rolling
- legs enabling walking, crawling, running, climbing, and jumping
- arms for grabbing hold, swinging, and climbing
- wings for flying
- flippers for swimming.

As soon as you start considering movement, you must also think about stability. After all, it typically takes a child at least a year before it can learn how to walk. For people and robots, there is also the notion of the center of gravity, which is some point above the ground where we are walking and able to stay balanced. Too low a center of gravity means that we are dragged down to the ground, while one that is too high means instability. Hand-in-hand with this concept is the notion of a ***polygon of support***. This is the platform that must support a robot to enforce stability. Humans have such a support platform as well, somewhere up in our torsos, only we are not usually aware of it. For a robot, as it attains more legs (that is, three, four, or six), this becomes less of an issue. For example, Figure 10.11 depicts NASA's Jet Propulsion Lab Spiderbot.

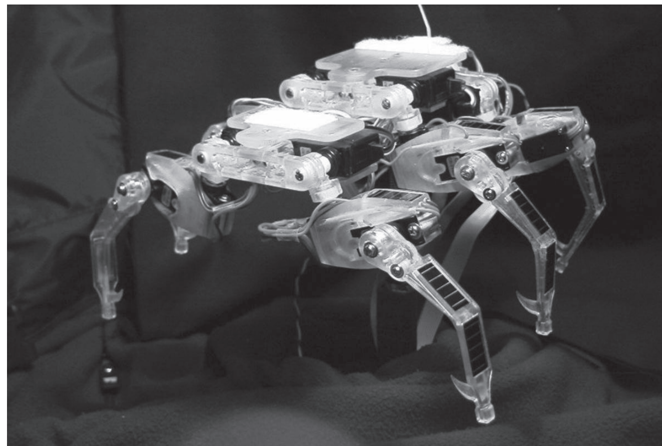


FIGURE 10.11 The Jet Propulsion Lab's "Spiderbot," Circa 2002

NASA's Spiderbot

Spiderbot was the first in a line of robots called "Spiderbot" for its spider-like appearance. This first MRE was a proof-of-concept to represent a

node in a mobile network of sensors for solid surface exploration. The JPL describes it further:

> Large robots use large actuators to build large structures. Fine work requires small, precise actuators and often small robots that can fit into confined spaces. Spiderbots can provide the small chassis and the mobility to support this second type of work. The Spiderbot is designed to develop and demonstrate hexapods that can walk on flat surfaces, crawl on meshes, and assemble simple structures. The task's current mission is to demonstrate complex mobility behaviors, including maneuvering (i.e., mesh crawling) in a space analog environment (i.e., micro-gravity). *http://www.robotics.jpl.nasa.gov/tasks/showTask.cfm?FuseAction=ShowTask&TaskID=30&tdaID=2585*

10.2.3 Path Planning for a Point Robot

A point robot is the simple notion of an autonomous robot as a single point operating in some well-defined environment, typically a Cartesian plane. Hence, the point (x,y) will be sufficient to describe the robot's state.

The fundamental problem is to find a path for the robot at some starting configuration, S = (a,b), to some goal state, T = (c,d). How can such a continuous path be found, if it exists? The most basic solution to this problem is known as the Bug2 Algorithm.

The algorithm is fairly straightforward. If a direct, straight-line path between S and T exists in the free space between S and T, the robot should use it. If the path is obstructed, then the robot uses the path until it encounters the obstacle (point P). The robot should then circumnavigate the obstacle until it can rejoin the line ST moving towards the goal T. If it encounters another obstacle, it should once again circumnavigate it until it finds another point on the obstacle on the line ST from which it can leave the obstacle in the direction of T that is closer to T than the point P at which it started circumnavigating the obstacle. If no such point exists, then the robot determines that no path exists from S to T.

Although the Bug2 Algorithm is known to be complete and certain to find a path to a goal if such a path exists, there is no guarantee that the path will be efficient. In order to be aware of the robot's position at all times

and plan appropriately, sensors must continuously refine their map of the environment and update their estimation of its position. In the world of robotics this is known as ***SLAM***, the simultaneous localization and mapping algorithm.

10.2.4 Mobile Robot Kinematics

Kinematics is the most basic study of how mechanical systems behave. In mobile robotics, this is a bottom-up technique that necessarily entails the worlds of physics, mechanics, software, and control. As such, it quickly gets rather complex because it requires software to control hardware at every moment.

For this purpose, much knowledge about kinematics was attained from the early programming of robot manipulators. The task was primarily to control a robot's arm. Consideration of the dynamics (force and mass) of such situations was important when built into the constraints on workspace and trajectory. We introduced the concept of locomotion in the previous section. Here we consider further factors which are integral to ***position estimation*** and ***motion estimation***, which are in themselves very challenging tasks.

Integral to considering the position and motion of a mobile robot is the position and angle of every wheel. Each wheel is considered for its contribution to the robot's motion, and these kinematic constraints are combined to express the entire robot's kinematic constraints.

The starting point is the robot's position in a simple X-Y plane. Consider its angle Θ which helps to create a reference point for the robot's direction of motion. That direction is represented with respect to the X-axis by the angle of Θ.

Hence the robot's global reference can be expressed by

$$\mathrm{I} = \begin{bmatrix} X \\ |Y| \\ \Theta \end{bmatrix}$$

This vector, comprised of X, Y, and Θ, defines what is called the "pose" of a robot. From this equation, all movements of the robot in the global plane $\{X_1, Y_1\}$ can be represented with respect to the local reference frame $\{X_R, Y_R\}$ using an ***orthogonal rotation matrix***.

Thus, instantaneous changes in the robot's position can be represented by matrix manipulations representing changes in the robot's wheel angles. Naturally, modeling of this kind is necessary and gets increasingly complicated. Adding more wheels and notions of velocity and diverse motions, possibly in different directions and dimensions, adds further complexity, which is beyond our purpose here. An excellent reference source for the further investigation of the technical details of kinematics, robot perception, mobile robot localization, and planning and navigation is the text by Siegwart, Nourbakhsh, and Scaramuzza.

10.3 Applications: Robotics in the Twenty-First Century

This section presents three major robotic systems that were developed in the twenty-first century: Big Dog, Asimo, and Cog. Each project represents a major effort that has been ongoing for several decades, starting in the late twentieth century. Each addresses complex and sophisticated technical issues and problems in robotics introduced in the previous section. Big Dog is mainly concerned with locomotion and conveyance of heavy loads, particularly for military purposes. Asimo displays diverse aspects of locomotion with a strong emphasis on anthropomorphic elements, that is, understanding how humans move. Cog is more about thinking, which is also considered to be special to humans, distinguishing us from other living beings.

10.3.1 BigDog

In 1986, Marc Raibert, Kevin Blankespoor, Gabriel Nelson, and Rob Playter, leaders of the ***BigDog*** Team at MIT, wanted to achieve animal-like mobility on rough terrain that people and vehicles have difficulties navigating. This effort was motivated by the fact that less than half of the earth's land is navigable by wheeled and tracked vehicles. The goal was to develop mobile robots that could perform on a par with humans and animals in terms of mobility, autonomy, and speed. Typical challenges included terrain that is steep, rutted, rocky, wet, muddy, and covered with snow. The team developed a series of robots that had up to four legs to perform movements of which humans and animals are capable. These multi-legged robots were developed to study dynamic control and the challenges of maintaining balance for robots on diverse terrain. Dynamically balanced legged systems were needed, hence BigDog was invented.

BigDog is a legged robot developed by Boston Dynamics (c. 1996) and was funded by DARPA (Defense Advanced Research Projects Agency). It is the size of a large dog, about 3 feet long, 2.5 feet tall, and weighs around 240 lbs. The goal of the BigDog project was to create an unmanned legged robot that could travel anywhere a person or an animal could go. This robot has built-in systems for power, actuation, sensing, control, and communication. Ideally, the system would be able to travel anywhere, run for consecutive hours, and carry its fuel and weight without trouble.

A human being employs an operator control unit (or OCU) connected to an IP radio to control BigDog's actions. A human employs a controller to provide steering and speed parameters to guide the robot through diverse terrains. The controller can also start and stop the robot as needed. The controller can also direct BigDog to walk, jog, or trot. The data is displayed and input. Then the robot's AI system takes over and operates on its own to make sure it stays upright or mobile.

BigDog employs AI for the coordination of its basic posture and to prevent falls, enabling it to learn to distribute weight amongst its four legs. This allows BigDog to carry heavy loads and to maneuver through diverse and rough terrain with little human support. The goal is to develop a system with auto-control. The robot has to be smart enough to navigate with little or minimal human guidance or intervention. The robot has 50 sensors which feed information to the onboard computer that monitors how BigDog is moving and where it is, and provide data from the field. Future projects seek further independence from human control, particularly in areas where there is limited human access.

There are high-level and low-level control systems which help maintain the robot's balance. The high-level system coordinates how the legs move as well as the speed and height of the body during movement, and the low-level system positions and moves the joints. This control system also helps it learn to adjust to maintain balance through slopes and climbs. It also controls ground actions to help maintain support of the robot's movements and keep it from slipping. If it falls, it learns to get back up and stand on all four legs, continuing with its movement through the terrain. The system also allows BigDog to have a variety of movement behaviors, including standing up on all four legs, squatting down, walking normally, or crawling by moving one leg forward at a time or in a diagonal action.

BigDog's power supply consists of water cooled by a two-stroke internal combustion engine, and the engine delivers high-pressure oil into the robot's leg actuators. Each leg has four hydraulic actuators that power BigDog's joints as well as a passive fifth degree of freedom. These actuators have sensors for the joint position, with a heat exchanger mounted on the body to stop it from overheating the engine. BigDog's 50 sensors include itertial sensors that measure the attitude and acceleration of the body and joint sensors for the actuators that help it move. These features enabled and facilitated BigDog through its longest movement of 6.2 consecutive miles. It can carry up to 154 kilograms on a flat terrain, but normal loads are usually 50 kilograms on a normal day. BigDog also has a visual system and a LIDAR, which is a pair of cameras, a computer, and visual software (Figure 10.12). These components help point out the terrain that BigDog is navigating and assist it in finding a clear path forward. The LIDAR system is for the sole purpose of ignoring a human operator and enabling the robot to use its sensors to follow a human leader out in the field.

FIGURE 10.12 BigDog Carrying Its Weight in Supplies

BigDog has a quadrupedal walking algorithm for sloped and tough terrains. It can walk on sloped pathways of up to 60 degrees but can also take into account unexpected or irregular terrain with the assistance of its control system. BigDog adapts to different changes in two ways: It fixes itself

according to the height and elevation of the terrain and footfall placement so that it will not go lopsided and fall over on its side, and it also looks at shadows for changes to make its own adjustments in posturewhile traveling through diverse terrain. BigDog's control system is coordinated with kinematics and ground reaction forces so that it can optiminze the amount it can carry. The control system optimizes the load by splitting it equally among the robot's legs.

Future Outlook: There are many plans for the future of BigDog. The team wants to make it possible for BigDog to move through rougher and steeper terrain and have it be able to carry more and heavier loads. The team wants to upgrade its engine and system to make it quieter, as its motors and system are extremly noisy. They also want BigDog to be less reliant on humans and employ computer vision to allow it to navigate entirely on its own. So far, new items include a head, arm, torso, and various other parts to increase versatility. These additions have given BigDog the ability to use its entire body to throw heavy objects around or lift and move heavy objects aside if they become obstructions.

FIGURE 10.13 BigDog Robots Trot Around in the Shadows

10.3.2 Asimo

Next, we present another robotics project that has been ongoing for many years: the Honda Asimo robot. Asimo moves in a very human-like way and was designed to be particularly helpful to people.

FIGURE 10.14 Honda's ASIMO

Imagine a world where humans and machine live together, aiding and supporting each other in all tasks ranging from carrying the everyday grocery shopping bags to helping firefighters rescue people trapped in flaming houses or fallen structures. This is a world envisioned by the Honda engineers who conceived Asimo in Japan in 1986. Asimo is a two-legged humanoid robot created in Honda's research lab after two decades of research and development. The objective of creating a humanoid robot that resembles and duplicates the complex structure of a human being is so that it is able to aide people with various activities for the advancement of scientific development.

Creating a humanoid robot was not an easy task. However, Honda has embraced this challenge by envisioning a world where robots and humans interact harmoniously. Having a valuable partner with great mobility and ability to maneuver who can interact with humans would be a great support for people who need an extra set of helping hands without the expense of another human.

Asimo's design concept was to make it into a people-friendly robot that is both lightweight and flexible. The Asimo is compact: 120 cm or 4 feet tall and weighing approximately 52 kgs or 115 lbs. The engineers chose this size to allow Asimo to operate freely and efficiently in a human living space. Based on their research, this height allows Asimo to "operate light switches and door knobs, and work at tables and work benches."

After collecting various data about human mobility and locomotion, including walking and other forms of human movement, Honda developed Asimo to walk in a very similar way to how humans walk. The two-legged

walking concept includes the operation and movement on different surfaces. Asimo can perform everyday tasks, such as walking from one point to another while avoiding obstacles, climbing or descending stairs, pushing a cart, passing through doorways, and carrying things while walking. These advanced physical capabilities are achieved by a number of sensors placed to determine the leg's joint angle and speed to mimic humans' center of gravity. These sensors collect data and interpret it into information to be processed for the next movement.

Asimo's second most prominent feature is its ability to interact with humans. Asimo must be able to approach and communicate with them. It achieves this by processing information that it captures through replicating humans' five senses. Asimo captures video input through the two cameras mounted in its head, which allow it to recognize moving objects and facial features on humans for limited facial recognition. It also creates a map of the surrounding environment with the visual information that helps for the purpose of collision prevention and object positioning.

Asimo is able to distinguish and interpret sounds and voice commands that are captured by the microphones installed in its head. Asimo processes audio input, enabling it to "recognize when its name is called, and then turn to the source of a sound," as well as react to "unusual sounds, such as those of an object falling or a collision, and face in that direction." Audio processing enables Asimo to engage in conversations with humans through its abilities in speech and natural language understanding. It is possible for Asimo to carry out orders and respond to them with specific feedback; the robot has internet connectivity, which enables it to access information via the internet to provide answers, such as news and weather conditions.

Future Outlook: Asimo's prospects for meeting its original goal—to be a helper to people in need—seem to be very bright. With all the capabilities that Asimo has, it would be able to not only support the sick and elderly, but also provide help for situations where it would be dangerous for humans to function, such as cleaning a toxic spill or putting out a blazing fire without risking lives. Furthermore, Asimo can provide a sense of companionship to people. Although it is not currently available for sale or lease in the United States, Asimo is featured in Japanese science museums and is "being used by a few high-tech companies to welcome guests to their facilities."

Although Asimo is a robot, it has traveled to many countries and landmarks around the world, ranging from the Brooklyn Bridge all the way to

Europe and Switzerland. It was also featured as a guest in Disney Land, and played soccer with President Barack Obama. Its popularity is increasing as it keeps encouraging and inspiring young people around the world to study the sciences via robotics and AI.

Jaemi the Humanoid Robot

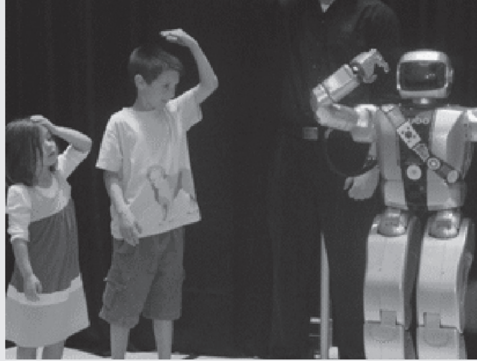


FIGURE 10.15 Jaemi the Humanoid Robot

Children play "Simon Says" with Jaemi, a humanoid robot (HUBO), during its visit to the Please Touch Museum in Philadelphia, PA. Jaemi was created by a team from Drexel University working in collaboration with Korean researchers. The project was supported by the National Science Foundation Partnership for International Research and Education (PIRE) program.

This image accompanied the NSF press release "U.S. and Korean Researchers Unveil Newest Research Team Member: Jaemi the Humanoid."

Credit: *Lisa-Joy Zgorski, National Science Foundation*.

Next, we present another long-term project that attempts to fulfill some of the early original aspirations for robotics discussed in previous sections, that is, to be able to mimic how people learn to interact as children and to develop cognitive skills.

10.3.3 Cog

In 1993, a team at MIT headed by Rodney Brooks started to construct a robot named Cog, which is short for "cognition." Cog was built based on the theory that "humanoid intelligence requires humanoid interactions with

the world," which would have necessitated the construction of a robot that would think and experience the world in the same way that a human would. Cog is made of actuators and motors that work similarly to humans' bones, joints, and movements. The MIT team built a robot that has a human-like intelligence, mimicking the human body and its behaviors. Nonetheless, there are some important aspects of the human body that cannot be mimicked by a robot. The team also wanted to be able to use this robot to interact with others as humans would. For the "training," Cog would interact with humans. What better way is there to learn human behaviors than to interact with them?

Cog was designed to simulate the same environments and physical constraints that adult humans encounter. Although it does not have legs, it does have a pair of symmetrical arms, a body, and a head. The lower part of its body, beyond the waist, is just a stand. Cog "sees" with two pairs of cameras mounted on its head with two DOF, and two microphones enable it to hear. Each eye also has its own pair of cameras for wide view and far range. The motor system has sensors indicating where the joints are and gives information on their current status, as well as if there are any issues or problems with them. Cog's arm also provides feedback by having an electric motor there to operate the arm and provide torque feedback information. The robot has a total of 22 DOF. It has six degrees in its arms, four degrees for its neck, three in its eyes, two degrees in its waist, and one in its torso enabling twisting motions.

Cog has a diverse network with many different processors operating at different control levels. Devices range from small microcontrollers for joint-level control to digital signal processors. The brain controls have been revised many times to help improve the way Cog acts like a human. The first network contained 16 megahertz Motorola 68332 microcontrollers with custom boards connected through dual port RAM. The current version of Cog consists of a network of 200 megahertz industrial personal computers running the QNX real-time operating system connected to a 100 VG Ethernet. This network currently has 4 nodes, but more can be added if desired.

The robot has a pair of electret condenser microphones mounted on its head close to where human ears would be. The microphone is similar in functionality to what a hearing aid is to a human. Cog includes a stereo system that amplifies the audio system and connects to a C40 DSP system. The team wanted to use these hearing systems to allow the robot to be aware of sounds that it hears in the same environment that humans do.

They also wanted to do the same with the robot's vision. Each of the robot's eyes rotates in a vertical and horizontal axis. In order to get a better resolution and view of the environment, Cog takes the visual information and processes the image in its network for a better image.

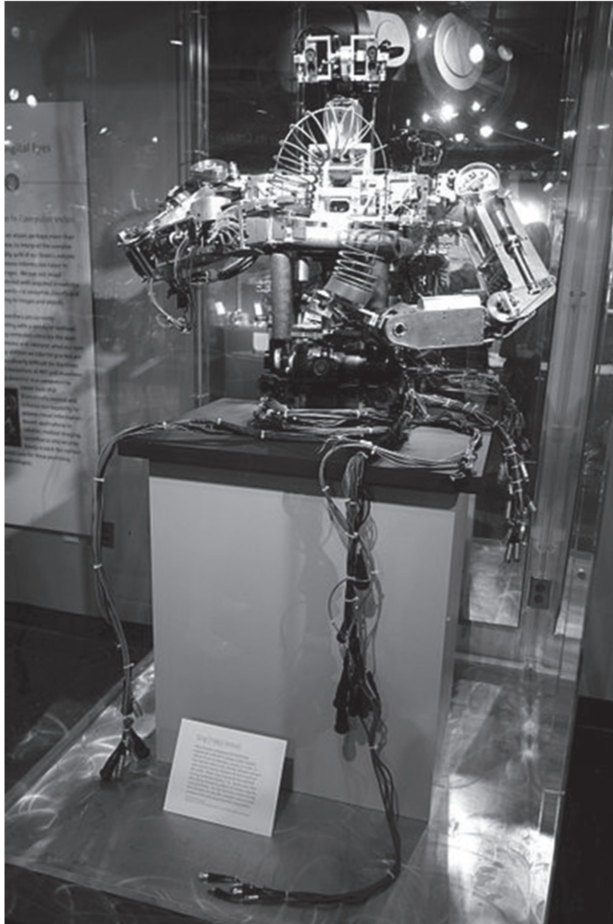


FIGURE 10.16 Image of Cog at the MIT Museum

Humans have a vestibular system which they use for movement and a sense of balance. Without it, people would fall over or would stay stationary. The brain takes information from this system and helps human beings coordinate everyday activities, such as walking and keeping themselves upright. The human system has three sensory organs with a semicircular passage. The team at MIT wanted to copy this idea for Cog. Cog includes three

rate gyroscopes placed on an orthogonal axis and two linear accelerometers. They put these devices below the eye so it can imitate sensory information for balance. The robot amplifies, processes, and converts these sensory devices for its computer "brain."

The team at MIT has created a pointing action that allows Cog to extend its arm and point at whatever is there. This action was tested many times, even without having the team observe its performance. During these actions, Cog's neck was still and it pointed at a target. In the initial stages of experimentation, Cog would perform these actions rather primitively, akin to a human infant or someone who is inexperienced at a certain task. However, in the process of "maturing," Cog seemed to learn and become more accurate in locating the target. In some sense, Cog became more human-like through its ability to mimic human actions; it learned and then began to practice achieving perfection in performing actions.

Future Outlook: Cog's developers seek to continually make improvements that will enable it to behave more like humans (for better or worse!), including the manipulation of its facial features. Cog currently does not have a face, but in the future, MIT roboticists will try to give Cog organic features akin to humans. Researchers also tried to replicate the behavior and thought processes of humans. Objectives included getting Cog to learn the relationship between motor commands and sensory inputs so it can observe and learn through its own actions. The team at MIT will try to get the neck and body to fully rotate as much as possible to simulate the way a human body rotates. The robot's front torso feedback was tested by using resistive force sensors. One experiment involved applying considerable force to a surface sensor, enabling the simulation of the robot's perception of forces.

The MIT team's plans for Cog include a greater number of sensors, motors, cameras, and joints so that it will have more DOF. This would allow Cog to become more human-like. Cog has learned to adapt to the way humans do things, but there are still some actions that it needs to learn and adapt to. A big challenge for Cog is to be able to adapt to new environments as a human infant might. Nonetheless, Cog has a long way to go before it becomes a full human simulation with thoughts, human-like movements, and interactions.

One of the main questions perplexing scientists and philosophers is how to determine whether a machine, robot, or an artificial creation possesses any sort of intelligence or conscience at the level of human intelligence. However, in order to compare the level of intelligence of different agents

we have to define what intelligence, or an intelligent being, means. Humans are intelligent beings because they are ***capable of thinking, rationalizing, learning, and conceptualizing information in their brains***. Can robots with algorithms that possess sufficient case scenarios be able to exhibit some form of intelligence? Certainly, that is a very plausible scenario, since nowadays robots can look, sound, and act like a person. They are capable of learning and storing information in their memory and processing it into logical cases. They are able to analyze a given sentence based on its semantics and syntax and come up with a credible and logical answer—but does that qualify these machines as intelligent? Is being able to effectively and continuously respond correctly the equivalent of understanding?

It was claimed that a chatbot program called Eugene Goostman fooled judges into believing that the program was actually a thirteen-year-old Ukranian boy, thus passing the Turing Test. The chatbot program fooled the judges by avoiding questions that it did not have a concrete answer to, much like how a thirteen-year-old boy would act. Therefore, it is disputed amongst various scientists that the Turing Test only works with low-level intelligent (low AI) machines and can in those cases distinguish between machine and humans. However, in the case of the new highly intelligent (AI) machines developed today, the Turing Test fails to separate the two. In addition, a number of new "Turing Tests" have been proposed.

10.3.4 The Lovelace Project

The Lovelace Test: In order to design a test capable of distinguishing strong AI, the ***Lovelace Test*** was proposed by Bringjord, Bello, and Ferrucci to set a new bar for determining intelligent beings. It requires the machine to create something original, something that even the creator cannot explain how it was created, such as a poem, story, music, or painting—or any creative act that requires the cognitive capabilities of humans. These creative acts would then be evaluated by a human being in order to determine whether the creation passes a set of criteria.

Lovelace vs. Lovelace 2.0: Mark O. Riedl enhanced the Lovelace Test by proposing the ***Lovelace Test 2.0,*** stating that "the artificial agent passes if it develops a creative artifact from a subset of artistic genres deemed to require human-level intelligence, and the artifact meets certain creative constraints given by a human evaluator." The Lovelace Test 2.0 evaluates the creativity instead of only the intelligence of a machine.

The Lovelace 2.0 Test is as follows: artificial agent α passes the Lovelace Test if and only if:

- α creates an artifact o of type t,
- o conforms to a set of constraints C where $c_i \in C$ is any criterion expressible in natural language,
- a human evaluator h, having chosen t and C, is satisfied that o is a valid instance of t and meets C, and
- a human referee r determines the combination of t and C to not be impossible.

FIGURE 10.17 Robot at the Royal Australian Mint, and a Canberra Watercolor Painting (*www.kopecart.com*)

Riedl believed that a "computational system can originate a creative artifact," for example, when creating a fictional story, a machine requires common knowledge, planning, reason, language processing, familiarity with the subject, and a cultural artifact. However, no story generation system can pass the Lovelace 2.0 Test because most story generation systems require *a priori* (knowledge, or an argument independent of experience) domain descriptions.

Thus, although robots and machines have greatly advanced in the field of AI, there is a fundamental difference between humans, who possess creativity, and machines, which still follow a set program or rationalized path.

10.4 Summary

Robotics was once a rather distinct field which was closely related to AI via computational geometry and vision. Today, we can see many aspects of AI in robotics, especially as embedded systems. This includes search algorithms, logic, expert systems, fuzzy logic, machine learning, neural networks, genetic algorithms, planning, and games. Robots do not navigate stating, "I have AI," but it is clear that robotics as a field would not be where it is without employing AI. We discussed examples of how and where robotics is and how it will be used. Let us not forget the effect that advances in natural language and speech understanding have had on improving robotics.

The history of robotics and man is much richer than one might imagine. It starts with notions of robot lore, and the early mechanical systems such Vaucanson's duck and von Kempelen's Turk from the eighteenth century. Robots in film and literature are well-known via Mary Shelley's *Frankenstein* (1817), Karel Čapek's *R.U.R.* (1921), and Fritz Lang's *Metropolis* (1926), all of which pose a rather grim picture of the future impact of technology on man's life. In the first half of the twentieth century, science fiction hero Isaac Asimov already had the vision to develop the *Three Laws of Robotics*. More recent systems and their capabilities were presented. Technical details were presented, as well as some of the standard and more challenging issues. We discussed the various applications of robotics, focusing on BigDog, Asimo, and Cog, as well as new tests for AI via the Lovelace Project. (The Application Boxes on BigDog and Cog were contributed by Peter Tan. Application Boxes on Asimo and Lovelace were contributed by Mimi Lin Gao.)

References for Chapter 10.:

References for Table 10.1

1. RobotWorx. The History of...KUKA Robotics. December 9, 2014. Retrieved from *http://www.used-robots.com/articles/viewing/the-history-of-kuka-robotics.*
2. Nocks, L. 2007. *The Robot: The Life Store of Technology*. Westport: Greenwood Publishing Group.
3. Williams, J. D. Direct Drive Robotic Arms. December 9, 2014. Retrieved from *http://diva.library.cmu.edu/Kanade/kanadearm.html.*
4. Ahmad, N. 2003. The humanoid robot Cog. *Crossroads* 10 (2): 3.
5. Carbone, G. and Ceccarelli, M. Legged Robotic Systems. December 9, 2014. Retrieved from *http://cdn.intechopen.com/pdfs-wm/33.pdf.*
6. Sony. ERS-1010. December 9, 2014. Retrieved from *http://www.sony.net/Fun/design/history/product/1990/ers-110.html.*
7. Buehler, M. 2006. BigDog – a dynamic quadruped robot. Robotics Institute Seminar. Boston Dynamics. BigDog - The Most Advanced Rough-Terrain Robot. December 9, 2014. Retrieved from *http://www.bostondynamics.com/robot_bigdog.html.*
8. Cox, W. Top 10 Robots of the Past 10 Years – Robots of the Decade Awards. 4 January 2010. December 9, 2014. Retrieved from *http://www.robotshop.com/blog/en/top-10-robots-of-the-past-10-years-robots-of-the-decade-awards-3743.*
9. Earnest, L. December 2012. Stanford Cart. December 9, 2014. Retrieved from *http://web.stanford.edu/~learnest/cart.htm.*
10. Tate, A. December 14, 2012. Edinburgh Freddy Robot. December 9, 2014. Retrieved from *http://www.aiai.ed.ac.uk/project/freddy/.*
11. Humanoid Robotics Institute, Waseda University. Humanoid History -WABOT-. December 9, 2014. Retrieved from *http://www.humanoid.waseda.ac.jp/booklet/kato_2.html.*
12. Tomy. *http://www.theoldrobots.com/omnibot.html*
13. Sirius. *http://www.megadroid.com/Robots/mody.htm*

14. Kawada Industries. *http://global.kawada.jp/mechatronics/*

15. AnthroTronics Inc. *http://www.anthrotronix.com/?option=com_content&view=article&id=81&Itemid=144*

16. Anybots. *http://www.anybots.com/*

17. King's College London. *http://www.whoosh.co.uk/inkha/TextLifeStory.htm*

18. MIT. *http://people.csail.mit.edu/edsinger/domo_research.htm*

19. KITECH. *http://www.plasticpals.com/?p=12155*

20. Mitsubishi Heavy Industries. *https://www.mhi-global.com/products/detail/wakamaru_about.html*

21. Fjitsu. *http://thefutureofthings.com/5191-fujitsus-enon-robot/*

22. MUSA. *http://www.technovelgy.com/ct/Science-Fiction-News.asp?NewsNum=423*

23. Vecna Technologies. *http://www.vecna.com/labs; http://www.gizmag.com/battlefield-extraction-assist-robot/17059/*

24. ISAAC Team. *http://www.isaacrobot.it/*

25. Willow Garage. *http://www.willowgarage.com/*

26. Mechatrons. *http://mechatrons.com/rubot-ii/*

27. Automatica. *http://techcrunch.com/2010/06/18/topio-dio-meet-vietnams-first-robot/; http://en.akihabaranews.com/51330/robot/meet-topio-dio-vietnams-first-humanoid-service-robot*

28. University of Amsterdam. *http://www.foxnews.com/story/2008/03/17/cowardly-phobot-steals-show-at-amsterdam-robot-conference/*

29. Salvius Robot. *http://salviusrobot.blogspot.com/*

30. Engineering University of Sana. *http://www.scribd.com/doc/57089754/Roboty*

1. Heppenheimer, T. A. 1985. Man makes man. In *Robotics*, edited by M. L. Minsky. Omni Press: New York.

2. Minsky, M. L. 1985. Ch 1, *Introduction*. In *Robotics*, edited by M. L. Minsky. Omni Press: New York.

3. Wiener, N. 1948. *Cybernetics: Or Control and Communication in the Animal and the Machine.* Paris (Hermann & Cie) & Cambridge, MA: MIT Press. 2nd revised ed. 1961.

4. Mataric, M. 2007. *The Robotics Primer*. Cambridge, MA: MIT Press.

5. Levy, D. N. L. 2006. *Robots Unlimited*. A.K. Peters, Ltd: Wellesley, MA.

6. Dudek, G. and Jenkin, M. 2010. *Computational Principles of Mobile Robotics*, 2nd edition. Cambridge, England: Cambridge University Press.

7. Siegwart, R., Nourbaksh, I, and Scaramuzza, D. 2011. *Introduction to Autonomous Mobiles Robots*, 2nd ed. Cambridge, MA: MIT Press.

Big Dog References

Raibert, M. 1986. *Legged Robots that Balance*. MIT Press. Retrieved from *http://www.bostondynamics.com/img/BigDog_IFAC_Apr-8-2008.pdfhttp://phys.org/news/2013-03-boston-dynamics-bigdog-toss-video.html*

Asimo References

1. *http://asimo.honda.com/*

2. *http://asimo.honda.com/downloads/pdf/asimo-technical-faq.pdf*

3. *http://asimo.honda.com/downloads/pdf/asimo-technical-information.pdf*

Cog References

1. Overview of the Cog project. Retrieved from *http://www.ai.mit.edu/projects/cog/OverviewOfCog/cog_overview.html*

2. Naveed, Ahmad. The Humanoid Robot Cog (page 2)

Lovelace References

1. Cole, D. The Chinese Room Argument. The Stanford Encyclopedia of Philosophy (Summer 2014 Edition), Edited by Edward N. Zalta. Retrieved from *http://plato.stanford.edu/archives/sum2014/entries/chinese-room*

2. Amlen, D.2014. Our Interview with Turing Test Winner Eugene Goostman. Retrieved from *https://www.yahoo.com/tech/our-interview-with-turing-test-winner-eugene-goostman-88482732919.html*

3. Bringsjord, S.; Bello, P.; and Ferrucci, D. 2001. Creativity, the Turing Test, and the (better) Lovelace Test. *Minds and Machines* 11: 3–27.

4. Riedl, M. O. 2014. The Lovelace 2.0 Test of Artificial Creativity and Intelligence. Retrieved from *http://arxiv.org/pdf/1410.6142v1.pdf*

APPENDIX

REVIEW QUESTIONS

Chapter 1

Q1. Explain the term AI. Differentiate between machine and human intelligence.

Q2. Define AI and briefly explain the history of AI.

Q3. Explain the various areas where the concept of AI is used.

Q4. Describe the components of AI.

Q5. What are the advantages of AI?

Q6. What is the Turing Test? How is it helpful in concluding that the machine can think?

Chapter 2

Q1. What is a problem in AI? How can you solve it with different representation approaches?

Q2. What are the characteristics of a problem in AI? How can we represent a problem in AI?

Q3. What is state space representation? Explain it using an example.

Q4. How is state space representation helpful in representing problems? Explain the Water Jug problem.

Q5. What is "conflict resolution"? How can we address it?

Q6. What do you understand about production systems? Explain different types of production systems.

Q7. What are the advantages and disadvantages of production systems?

Q8. Can a problem be reduced using a graphical method? Show this using an example.

Q9. Write short notes on the following:

1. Conflict Resolution
2. 8-Puzzle Game
3. Water Jug Problem

Chapter 3

Q1. What is meant by search and control strategies? How are these useful in AI?

Q2. Explain the different steps involved in search techniques.

Q3. What is meant by search strategies? Explain data-driven search techniques.

Q4. Explain goal-driven search techniques.

Q5. Compare and explain the forward search and backward search.

Q6. Discuss search techniques by explaining any one of your own choice (such as the uniformed searching technique).

Q7. Write an algorithm for the depth-first or breadth-first search technique.

Q8. Explain the different factors affecting search techniques. Compare the depth-first and breadth-first searches.

Q9. What are heuristics? Explain a heuristic search technique.

Q10. Write an algorithm using the hill-climbing method and explain it using an example.

Q11. Explain the different problems of the hill-climbing method.

Q12. What is the best first search? Explain it using an example.

Q13. Explain the A^* algorithm.

Q14. What is a beam search? Explain it using an example.

Q15. Write about constraint satisfaction.

Chapter 4

Q1. What is game playing in AI? Explain the components of a game playing program.

Q2. Describe the basic methods used for game playing programs.

Q3. Explain one of the procedures given below:

1. Minimax
2. Alpha-Beta

Q4. Which problems occur in computer game playing programs?

Chapter 5

Q1. Define knowledge and discuss the types of knowledge.

Q2. What are the differences between a knowledge-based system and database system?

Q3. What are the desirable characteristics of knowledge representation schemes? Discuss their advantages and disadvantages.

Q4. What is knowledge representation? Explain various techniques.

Q5. How are semantic networks helpful in representations of knowledge? In what way are they better than others?

Q6. Explain the concept of conceptual dependency.

Q7. Explain how knowledge is represented with the help of frames.

Q8. What is a script? Explain using an example.

Chapter 6

Q1. Define an expert system. Explain the different characteristics of expert systems.

Q2. What is the rule-based system architecture? Explain.

Q3. Describe the different components of an expert system.

Q4. What is a knowledge base? How does an interface engine work?

Q5. What is non-production system architecture? Explain one of these types of architecture. How is it better than the others?

Q6. Explain the different stages of the development of an expert system.

Q7. What is knowledge acquisition?

Q8. Explain the advantages and limitations of an expert system.

Q9. Explain MYCIN and EMYCIN.

Chapter 7

Q1. What is learning? Explain the different sources of learning.

Q2. Describe the components of a machine learning system.

Q3. What are the advantages and disadvantages of machine learning systems?

Q4. Explain the different types of learning.

Q5. What is learning as induction? Explain.

Q6. What is failure-driven learning? How does it work? Explain.

Q7. What is learning by being told or given advice? Explain.

Q8. What is learning by exploration? Explain.

Chapter 8

Q1. What is Prolog? What are the different reasons for using Prolog?

Q2. What is a Horn clause? How is it used in Prolog?

Q3. How can variables be declared in Prolog?

Q4. How is a query solved in Prolog?

Q5. What is recursion? Explain using an example.

Q6. What is a control predicate? Explain.

Q7. Explain 5 predicates used in Prolog.

Q8. Write down the different steps for creating a program.

Chapter 9

Q1. Which languages are used for building AI? Why do developers code AI programs with Python?

Q2. Describe the features of Python.

Q3. How could we know the side effects occurring in an AI program?

Q4. Name the most well-known libraries and modules in Python.

Q5. What are the supported data types in Python?

Q6. What are the differences between list and tuples?

Q7. What is a dictionary in Python?

Q8. Explain four major utilities used in Python.

Chapter 10

Q1. Discuss five areas of AI presented in previous chapters and their relationship to robotics.

Q2. In the Story Box of MrTomR and Bobby, explain how today's robots may or may not be able to perform the functions of MrTomR.

Q3. Describe some of the early myths about robotics that were presented in the chapter, including The Brass Head, the Homunculus, and the golem.

Q4. Describe the inventions of the father-son team Pierre and Henri-Louis Jaquet-Drov. When did they occur?

Q5. Name and describe two chess-related automata that were built in prior centuries.

Q6. Describe the literary works of Karel Čapek, Mary Shelley, and Isaac Asimov and how they projected concerns and developments in robotics.

Q7. Consider Asimov's Three Laws of Robotics—are they still valid?

Q8. Describe the purpose of the field of cybernetics.

Q9. Discuss the purpose and capabilities of the Tortoise by Grey Walters.

Q10. Describe the purposes and capabilities of the three significant modern-day robot projects presented in Section 15.3—Big Dog, Asimo, and Cog.

Q11. What is the Lovelace Project about? Do you believe it is sound and appropriate?

INDEX

A

A* Algorithm
  algorithm, 35–36
  fitness number, 35–36
AAs (action aider), 62
action procedure frames, 61
actor slots, 61
ACTS (actions), 62
admissibility, 25
AI, *see* artificial intelligence (AI)
AIBO, 158
AIMA, 123
alpha-beta cut off, minimax strategy with
  algorithm for, 49
  alpha-beta pruning, 48–49
  MAX NODE, 49
  MIN NODE, 49
  tree structure, 50
Anybots, 158
AO* Algorithm (problem reduction), 36–37
  algorithm, 37
  AND-OR graphs, 36–37
  AND/OR Tree, 37
architectural design, 86
architectures, expert systems
  non-production system architecture, 78–84
  rule-based system architecture (production systems), 73–78
argmax, 135–136
artificial intelligence (AI)
  advantages, 7
  application areas of, 7–9
    expert systems, 9
    game playing, 7–8
    natural language processing (NLP), 8
    problem solving, 7
    robotics, 8–9
    speech recognition, 9
    vision systems, 9
  components, 10–11
  computerized reasoning, 1
  definitions, 4
  goals, 4
  hardware, 11
  history, 5–6
  human intelligence in machines, 4
  intelligence, 3–4
  knowledge representation, 10–11
  machines and robotics. *see* machines and robotics, AI
  problem-solving, 11
  programming language, 10
  Python for, 127–128
  turing test, 2–3
Asimo, 158, 173–176
  audio processing, 175
  design concept, 174
  future, 175–176
  interaction capability, 175
  two-legged, 174–175
Asimov, Isaac, 151, 182
associative (semantic) network, 78–79
ATRANS, 64
ATTEND, 64

B

backtracking forces, 114
backward chaining, 75–76
backward search, 26
Bacon, Friar Roger, 144
beam search, 38–39
  algorithm for, 38
  learning applications, 39
  speech recognition and vision, 39
beam width, 38
BEAR, 159
best-first search, 34–35
  algorithm for, 35
beta, 48
BigDog, 157, 170–173
  AI and, 171
  control systems, 171
  future, 173
  goal, 171
  multi-legged robots, 170
  power supply, 172
  quadrupedal walking algorithm, 172–173
  sensors, 172
biomimetic systems, twentieth-century robots, 151–154
blackboard, 82–83
  architecture, 82–83
Blankespoor, Kevin, 170
Blind Search, *see* uniformed search
Boston Dynamics, 171
Brahe, Tycho, 145
Braitenberg, Valentino, 153
branching factor, 25
breadth-first search
  advantages, 29–30
  algorithm for, 29
  disadvantages, 30
  performance of, 30
Brooks, Rodney, 176–177
Brute Force search, *see* uniformed search
Bug2 Algorithm, 168

C

Canberra Watercolor Painting, 181
Čapek, Karel, 182
CASNET (Casual Associative Network), 78–79
clay golem, 145
CLBPS, *see* commutative law based production system (CLBPS)
COBOL, 105
Cog, 157, 176–180
  actuators and motors, 177
  chatbot program, 180
  Eugene Goostman, 180
  future, 179–180
  gyroscopes, 179
  level of intelligence, 179–180
  microphones, 177
  network, 177
  QNX real-time operating system, 177
  version, 177
commonsense knowledge, 54
commutative law based production system (CLBPS), 20
commutative production system, 20–21
componential intelligence, 4
compound gear train, 164
computer game playing programs, problems in
  horizon effect, 51
  optimal move question, 51
concept learning, 98
conceptual cases, 63
conceptual dependencies, 62–66
conceptualization stage, expert systems, 85
conceptual tenses, 63
conditionals, Python, 127

conflict resolution, 22–23
constraint satisfaction, 39–41
  cryptarithmetic problems, 39
  problem, 39
constructing robots, 142–143
contextual intelligence, 4
controllers, robot, 162
control system, 19, 75
CosmoBot, 158
critics, 10
cut-off depth, 28
cut predicate, 115–117
cybernetics, 151–153

D

DARPA (Defense Advanced Research Projects Agency), 171
database, 77–78
data-driven inference, 26
DC motors, robots, 163–164
decision-tree architecture, 80–82
declarative frame, 60–61
declarative knowledge, 55–56
decomposable production system, 21
degrees of freedom (DOF), 166
DENDRAL, 9
deployment, 86
depth-first search (DFS)
  advantages, 28
  algorithm for, 27
  disadvantages, 28
  performance of, 28
depth of problem, 25
destination slot, 61
DFS, *see* depth-first search (DFS)
dictionaries, Python, 126
Direct Drive Arm, 157
direct instruction, 97–98
DOF, *see* degrees of freedom (DOF)
domain, 71
  specific knowledge, 54
Domo, 158
Draftsman, 148–149

E

early binding, 129
EasyAi, 123
effectors and actuators, robot, 162–163
8-puzzle, 15–16
Elijah of Chelm, 144
The Empire Strikes Back (1980) (film), 151
Enon, 159
Eugene Goostman, 180
excitatory connections, robot, 154
EXPEL, 64
experimental intelligence, 4
expert systems, 9
  advantages, 89–90
  architectures
    non-production system architecture, 78–84
    rule-based system architecture (production systems), 73–78
  characteristics, 72
  definition, 71–72
  DENDRAL, 91
  EMYCIN, 93
  knowledge acquisition, 87
    strategies, 88–89
  knowledge engineering process, 86–87
  life cycle, 84–86
    conceptualization stage, 85
    formalization stage, 85–86
    identification stage, 84–85
    implementation stage, 86
    testing stage, 86
  limitations, 90–91
  MYCIN, 91–93
  PROSPECTOR, 93
explanation

facility, 76–77
mechanism, inference engine, 75
external interface, 77

F

factual knowledge, 74
fail predicate, 113–115
failure-driven learning, 99–101
FAMULUS, 156
feedback element, 96
fitness number, 35
Forbidden Planet (1956) (film), 151
formalization stage, expert systems, 85–86
FORTRAN, 105
forward-chaining systems, 75
forward search, 26
frames, 59–62
advantages, 62
architecture, 79–80
declarative frame, 60–61
definition, 59
procedural frame, 61
types, 59
Frankenstein, 146, 182
Freddy, 156
fuzzy logic, 141–142

G

Galaxy Science Fiction, 151
game playing, 7–8
in AI, 43
computer game playing programs, problems in
horizon effect, 51
optimal move question, 51
game tree, 44
program, components
plausible move generator, 44–45
static evaluation function generator, 45
strategies
minimax strategy, 46–48
minimax strategy with the alpha-beta cut off, 48–50
game tree, 44
ganged gears, 164–165
generators and coroutines, Python, 130–132
genetic algorithms, 142
goal identification, 85
GRASP, 64

H

HAS-PART links, 78
Henry, Joseph, 163
Heppinger, T. A., 143
heuristic function, 31
heuristic knowledge, 74
heuristic search techniques, 31
A* Algorithm, 35–36
AO* Algorithm (problem reduction), 36–37
beam search, 38–39
best-first search, 34–35
constraint satisfaction, 39–41
hill climbing, 32–34
hill climbing
algorithm for, 32
drawbacks, 33–34
local maximum, 33
plateau, 33
ridge, 33
problems associated with, 33
Hiro, 158
Honda Asimo robot, 173–176
horizon effect, 51
Horn clause, 106–107
humanoid intelligence, 176–177

I

idealized system, 97
IF-THEN rules, 74, 92
implementation stage, expert systems, 86
induction, 98
inference engine, 75–76
informed search, 31–41
  heuristic function, 31
  heuristic search techniques, 31
    A* Algorithm, 35–36
    AO* Algorithm (problem reduction), 36–37
    beam search, 38–39
    best-first search, 34–35
    constraint satisfaction, 39–41
    hill climbing, 32–34
  problems, 31
INGEST, 64
inheritance, 78
inhibitory connections, 154
Inkha, 158
intelligence, 53
  definition, 3
  knowledge, 56
  types, 3–4
inter-section search, 59
iPython Notebook, 123
IS-A link, 78
Issac, 159

J

Jacques de Vaucanson, 146–147
Jaemi, a humanoid robot (HUBO), 176
Jaquet-Drov, Henri-Louis, 148–149
Jaquet-Drov, Pierre, 148–149
Jaquet-Droz android, 149
Jet Propulsion Lab Spiderbot, 167
Judah ben Loew, 145

K

KeepOn, 159
Kempelen, Wolfgang von, 148
Kepler, Johannes, 145
kinematics, 169–170
Kismet, 157
knowledge
  base, 74–75
  commonsense knowledge, 54
  declarative knowledge, 55–56
  definition, 53
  domain specific knowledge, 54
  engineering process, 86–87
  importance, 56
  knowledge-based systems, 56
    and database systems, 56–57
  procedural knowledge, 54–55
  representation, 74–75
    associative networks, semantic networks of, 58–59
    conceptual dependency, 62–66
    frames, 59–62
    script, 66–69
  source, 83
  types, 54
knowledge acquisition, 86–87
  difficulties in, 87–88
  facility, 77
  process, 87
  strategies
    interview analysis, 89
    introspection, 89
    observations, 88
    protocol analysis, 88
    teach back, 89
knowledge-based management system (KBMS), 53
knowledge-based systems, 56
  and database systems, 56–57

L

Lang, Fritz, 150, 182
languages used for building AI, 121
late binding, 129
learning, 10
  element, 10, 96
  from examples, 97
  by instruction, 97
  machine learning
    advantages, 103
    characteristics, 97
    disadvantages, 103
    systems, 95–97
  types of, 97–102
    direct instruction, 97–98
    failure-driven learning, 99–101
    learning by analogy, 98
    learning by being told or getting advice (learning by instruction), 101–102
    learning by deduction, 98
    learning by exploration, 102
    learning by induction (learning by examples), 98–99
    rote learning or memorization, 97
Lenoir, Etienne, 163
level of intelligence, 179–180
LISP (list processing), 10, 105
lists, Python, 125–126
local maximum, 33
locomotion, 166–168
logical representation scheme, 57
Logic programming, 106
lore of golem, 144
Lovelace Project, 180–182
Lovelace 2.0 Test, 180–182

M

Machina Speculatrix (“machine that thinks”), 153
machine learning and neural networks, 142
machines and robotics, AI; *see also* robots
  applications
    Asimo, 173–176
    BigDog, 170–173
    Cog, 176–180
    Lovelace Project, 180–182
  constructing robots, 142–143
  fuzzy logic, 141–142
  genetic algorithms, 142
  history, 143–160
    early mechanical robots, 146–150
    Robot Lore, 144–146
    robots in film and literature, 150–151
    twentieth-century robots, 151–155
  logic and knowledge representation, 141
  machine learning and neural networks, 142
  MrTomR, 140
  natural language understanding and speech understanding, 142
  planning, 142
  production systems and expert systems, 141
  search, 141
  swarm intelligence, 142
  tabu search, 142
  technical issues, 160–170
machine vision, 161
matplotlib, 123, 133–134
maximizer, 46
Maxwell, James Clerk, 150
MBUILD, 64
MDP, 123
mechanical robots, 146–150
meta knowledge, 54
Metropolis (movie), 150, 182
Mind Magazine article, 2

minimax strategy, 46–48
  algorithm for, 46
  with alpha-beta cut off, 48–50
  game tree expanded by two levels, 47
  maximizer's move, 48
minimizer, 46
mobile robot kinematics, 169–170
Modulus Robot, 157
monotonic production system (MPS), 19
motion estimation, kinematics, 169
motors and gears, robot, 163–166
MOVE, 64
MPS, *see* monotonic production system (MPS)
MrTomR, 140
MTRANS, 64
multi-legged robots, *see* BigDog
Multiple Instruction Multiple Data (MIMD) Machines, 11
Multiple Instruction Single Data (MISD) Machines, 11
MUSA, 159
Musician, 148–149
MYCIN, 9

N

NASA's Spiderbot, 167–168
natural language
  generation, 8
  understanding, 8
    and speech understanding, 142
natural language processing (NLP), 8
Nelson, Gabriel, 170
network representation scheme, 57
neural network architecture, 83–84
NLP, *see* natural language processing (NLP)
NMPS, *see* non-monotonic production system (NMPS)
non-monotonic production system (NMPS), 20
non-production system architecture, 78–84
  associative (semantic) network, 78–79
  blackboard architecture, 82–83
  decision-tree architecture, 80–82
  frame architecture, 79–80
  neural network architecture, 83–84
NumPy, 123

O

objects slots, 61
Omnibot, 157
One-Way Link Representation, 59
operator control unit (OCU), 171
optimal move question, 51
order, 23
orthogonal rotation matrix, 169

P

pandas, 123
partially commutative production system (PCPS), 20
participants' identification and roles, 84
PAs (picture aider), 63
passive knowledge, 55
PCPS, *see* partially commutative production system (PCPS)
performance element, 10, 96
Phobot, 160
photophilic attraction, 154
physical body, robot, 162
PIP (Present Illness Program), 79
plateau, 33
plausible move generator, 44–45
Playter, Rob, 170
point robot, path planning for, 168–169
polygon of support, 167
position estimation, kinematics, 169
PPs (picture producer), 62
probability, Python, 137

problem
  generator, 10
  identification, 84
  instance, 25
  reduction, 17–18
    using AND-OR graph, 17–18
  solving, 7
  space, 25
problem representation
  in AI, 14–18
    problem reduction, 17–18
    state space representation, 14–17
  characteristics, 13–14
  conflict resolution, 22–23
  production system, 18–22
    advantages, 21–22
    characteristics, 19
    control system, 19
    limitations, 22
    rule applier, 19
    set of production rules, 18
    special features, 20–21
    types, 19–20
    working memory (WM), 19
procedural frame, 61
procedural knowledge, 54–55
procedural representation scheme, 57
production rules, 73
production systems, 18–22
  advantages, 21–22
  characteristics, 19
  control system, 19
  and expert systems, robots, 141
  limitations, 22
  rule applier, 19
  set of production rules, 18
  special features, 20–21
  types, 19–20
  working memory (WM), 19
program altered code, 100
progression planning, 15
prolog
  AI programming language, 105
  appending a list, 113
  clauses, 105
  compound queries, 109
  controlling execution in, 113–117
    cut predicate, 115–117
    fail predicate, 113–115
  data structures in, 111
  element of the list, 113
  free data structure, 106
  head and tail of a list, 111–112
  Horn clause, 106–107
  language, 105–106
  length of the list, 113
  members write statement, 112
  preliminaries, 105–106
  print the list in reverse order, 112
  program, parts of, 107–108
  queries to a database, 108
  recursion in, 110–111
  Robinson's resolution rule, 107
  solution to query, 109
  syntax and semantics of, 105
  Turbo Prolog, 117–119
  _ variable, 109–110
PROLOG (programming in logic), 10
PROPEL, 64
PTRANS, 63–64
pulse-width modulation, 165–166
PyBrain, 123
pyDatalog, 123
PyML, 123
Python, 121–122
  build AI using, 122–124
  features
    for artificial intelligence, 127–128
    conditionals, 127
    dictionaries, 126
    functions as Rst-class objects,
      128–130

generators and coroutines, 130–132
lists, 125–126
tuples, 126
languages used for building AI, 121
pitfalls, 125
running, 124–125
testing code, 137–138
useful libraries
matplotlib, 133–134
timing code, 132
utilities
argmax, 135–136
display, 134–135
probability, 137
union of dictionaries, 137
Python 11, features, 129–130

Q

QNX real-time operating system, 177
quadrupedal walking algorithm, 172–173

R

Raibert, Marc, 170
recency, 23
refraction, 23
regression planning, 15
requirements analysis, 86
resource identification, 85
Return of the Jedi (1983) (film), 151
Return on Investment (ROI), 72
ridge, 33
Riedl, Mark O., 180
Robinson's resolution rule, 107
robotics, 8–9; *see also* machines and robotics, AI
Robot Lore, 144–146
robots; *see also* machines and robotics, AI
applications
Asimo, 173–176
BigDog, 170–173
Cog, 176–180
Lovelace Project, 180–182
components
controllers, 162
degrees of freedom, 166
effectors and actuators, 162–163
motors and gears, 163–166
physical body, 162
sensory perception, 162
definition, 152
in film and literature, 150–151
global reference, 169
history, 143–160
early mechanical robots, 146–150
Robot Lore, 144–146
robots in film and literature, 150–151
twentieth-century robots, 151–155
inhibitory connections, 154
locomotion, 166–168
mobile robot kinematics, 169–170
neural networks, 154
photophilic attraction, 154
photophobic, 154
point robot, path planning for, 168–169
ROBOTY, 160
"Rossum's Universal Robots" *(R.U.R)*, 150, 182
rote learning or memorization, 97
Royal Australian Mint, 181
Rst-class objects, functions, 128–130
RuBot II, 159
rule applier, 19
rule-based system architecture (production systems), 73–78
database, 77–78
explanation facility, 76–77
external interface, 77
inference engine, 75–76

knowledge acquisition facility, 77
knowledge base, 74–75
production rules, 73
structure, 73
user interface, 76
rule-interpreter, 75
running, Python, 124–125
runtime, 132

S

Salvius, 160
scheduler, blackboard system architecture, 83
scikit, 123
scikit-learn, 123
Scout II, 157
Scribe, 148–149
script, 66–69
advantages, 69
components, 66–67
disadvantages, 69
search process, 25
strategies, 26
techniques
informed search or heuristic search, 31–41
uniformed search, 26–30
self .display, 134
semantic networks of associative networks, 58–59
sensory perception, robot, 162
Seropi, 158
servo motor, 165
Shakey, mobile robot, 154–155
Shelley, Mary, 146, 182
Silver Arm, 156
SimpleAI, 123
Single Instruction Multiple Data (SIMD) Machines, 11
Single Instruction Single Data (SISD) Machines, 11
SLAM (simultaneous localization and mapping algorithm), 169
source slot, 61
space complexity, 25, 28, 30
SPEAK, 64
specificity, 23
speech recognition, 9
Spiderbot, 167–168
standard system, 96–97
Stanford Cart, 156
Stanford Heuristic Programming Project, 71
Star Wars (1977) (film), 151
state space representation, 14–17
deficiencies, 17
static evaluation function generator, 45
Strasbourg cock, 146
STRIPS (Stanford Research Institute Problem Solver), 154
structured representation scheme, 57
supervised learning, 99
swarm intelligence, 142
system design, 86

T

tabu search, 142
Tan, Peter, 182
task slot, 61
TEIRSIAS, 93
testing code, Python, 137–138
testing stage, expert systems, 86
text-oriented tracing, 134
Three Laws of Robotics, 182
time complexity, 25, 28, 30
timing code, Python, 132
top-down logic, 98
Topio Dio, 160
Torres y Quevedo, Leonardo, 150
Tortoise, 152
translational degrees of freedom, 166
tuples, Python, 126

Turbo Prolog, 117–119
Turing Test, 2, 180
  representation, 2
  weakness, 3
The Turk, 148
twentieth-century robots, 151–155
  biomimetic systems, 151–154
  recent systems, 154–155

U

unification, 109
uniformed search
  algorithm, 26–27
  breadth-first search, 29–30
  depth-first search (DFS), 27–28
union of dictionaries, 137
unit tests, 137–138
unprogrammed behaviors, robots, 153
unsupervised learning, 99
user interface, 76

V

Vehicles (book), 153
vehicles, robots, 153
vision systems, 9
voltage, 164

W

WABOT-1, 156
WABOT-2, 156
Wakamaru, 158
The Walking Forest Machine, 157
Walter, William Grey, 152–153
water jug problem, 16
Watt, James, 150
Wiener, Norbert, 151
Willow Garage, 159
working memory (WM), 19