Artificial Intelligence and Expert Systems

I. Gupta & G. Nagpal

Artificial Intelligence and Expert Systems

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 and Expert Systems

Itisha Gupta
&
Garima Nagpal

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

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

I. Gupta & G. Nagpal. *Artificial Intelligence and Expert Systems.*
ISBN: 978-1-68392-507-1

Library of Congress Control Number: 2020935416

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

CONTENTS

Preface *xiii*

Chapter 1: Introduction to Artificial Intelligence **1**

1.1 The Turing Test 2
1.2 Intelligent Agents 5
1.2.1 Software Agents 5
1.2.2 Physical Agents 5
1.3 Approaches in Artificial Intelligence 7
1.3.1 Acting Humanly: The Turing Test Approach 7
1.3.2 Thinking Humanly: The Cognitive Modelling Approach 8
1.3.3 Thinking Rationally: The Laws of Thought Approach 8
1.3.4 Acting Rationally: The Rational Agent Approach 9
1.4 Definitions of Artificial Intelligence 10
1.4.1 Intelligent Behavior 12
1.4.2 Interpretations of Artificial Intelligence 12
1.5 AI Problems 13
1.5.1 Tasks Under Artificial Intelligence 14
1.5.2 Tasks Domains of Artificial Intelligence 14
1.6 Features of AI Programs 16
1.7 Importance of AI 17
1.8 What Can Artificial Intelligence Systems Do? 17
1.9 What Can Artificial Intelligence Systems Not Do Yet? 18
1.10 Advantages of AI 18
1.11 Disadvantages of Artificial Intelligence 19
Exercises 21

Chapter 2: Applications of Artificial Intelligence **23**

2.1 Finance 23
2.2 Hospitals and Medicine 23
2.3 Robotics 24
2.4 Expert Systems 24
2.5 Diagnosis 25
2.6 Pattern Recognition 25
2.7 Natural Language Processing 26
2.8 Game Playing 28
2.9 Image Processing 28
2.10 Data Mining 30
2.11 Big Data Mining 30
Exercises 31

Chapter 3: Introduction to the State Space Search **33**

3.1 State Space Search 34
3.1.1 The Search Problem 35
3.2 Search Techniques 38
3.2.1 Basic Search Algorithm 38
3.3 Types of Searching Techniques 39
3.3.1 Uninformed Search (Blind Search) 39
3.3.2 Avoiding Repeated States 50
Exercises 52

Chapter 4: Heuristic Search Strategies **53**

4.1 Types of Heuristic Search Techniques 54
4.1.1 Generate and Test 55
4.1.2 Best First Search 55
4.1.3 Hill Climbing Search 58
4.1.4 Simulated Annealing Search 61
4.1.5 A* Algorithm 62
4.1.6 AND-OR Graphs 64
4.2 Properties of the Heuristic Search Algorithm 65
4.3 Adversary Search 66
4.3.1 The MINIMAX Algorithm 67
Exercises 69

Chapter 5: Expert Systems **71**

5.1 Definitions of Expert Systems 71
5.2 Features of Good Expert Systems 72
5.3 Architecture and Components of Expert Systems 73

5.3.1 User Interface 74
5.3.2 Knowledge Base 75
5.3.3 Working Storage (Database) 79
5.3.4 Inference Engine 79
5.3.5 Explanation Facility 86
5.3.6 Knowledge Acquisition Facility 86
5.3.7 External Interface 86
5.4 Roles of the Individuals Who Interact with the System 86
5.4.1 Domain Expert 86
5.4.2 Knowledge Engineer 87
5.4.3 Programmer 87
5.4.4 Project Manager 88
5.4.5 User 88
5.5 Advantages of Expert Systems 89
5.6 Disadvantages of Expert Systems 90
Exercises 93

Chapter 6: The Expert System Development Life Cycle 95

6.1 Stages in the Expert System Development Life Cycle 96
6.1.1 Problem Selection 97
6.1.2 Conceptualization 98
6.1.3 Formalization 100
6.1.4 Prototype Construction 101
6.1.5 Implementation 106
6.1.6 Evaluation 107
6.2 Sources of Error in Expert System Development 109
6.2.1 Knowledge Errors 110
6.2.2 Syntax Errors 110
6.2.3 Semantic Errors 110
6.2.4 Inference Engine Errors 110
6.2.5 Inference Chain Errors 110
Exercises 111

Chapter 7: Knowledge Acquisition 113

7.1 Knowledge Basics 113
7.2 Knowledge Engineering 115
7.2.1 Knowledge Acquisition 116
7.2.2 Knowledge Engineer 117
7.2.3 Difficulties in Knowledge Acquisition 118
7.3 Knowledge Acquisition Techniques 120
7.3.1 Natural Techniques 121

7.3.2 Contrived Techniques 122
7.3.3 Modelling Techniques 126
Exercises 128

Chapter 8: Knowledge Representation 129

8.1 Definitions of Knowledge Representation 129
8.2 Characteristics of Good Knowledge Representation 130
8.3 Basics of Knowledge Representation 131
8.4 Properties of the Symbolic Representation of Knowledge 132
8.5 Properties for the Good Knowledge Representation Systems 133
8.6 Categories of Knowledge Representation Schemes 134
8.7 Types of Knowledge Representational Schemes 135
8.7.1 Formal Logic 135
8.7.2 Semantic Net 172
8.7.3 Frames 194
8.7.4 Scripts 213
8.7.5 Conceptual Dependency (CD) 225
Exercises 242

Chapter 9: Neural Networks 243

9.1 Neural Networks vs. Conventional Computers 244
9.2 Neural Networks 244
9.2.1 Neurons 245
9.2.2 Types of Neural Networks 245
9.2.3 Historical Background 246
9.3 Biological Neural Networks 247
9.3.1 Biological Neurons 249
9.4 Artificial Neural Networks 249
9.5 Differences Between Biological and Artificial Neural Networks 253
9.6 Architecture of a Neural Network 253
9.6.1 Single Layer Feed-Forward Networks 254
9.6.2 Multilayer Feed-Forward Network 255
9.6.3 Recurrent Networks 256
9.6.4 Feedback Networks 256
9.6.5 Network Layers 257
Exercises 258

Chapter 10: The Learning Process 259

10.1 Types of Learning in a Neural Network 259
10.1.1 Supervised Learning 259
10.1.2 Unsupervised Learning 261
10.1.3 Reinforcement Learning 262
10.2 Perceptron 262

10.2.1 The Representational Power of a Perceptron 263
10.3 Backpropagation Networks 264
10.4 Advantages of Neural Networks 264
10.5 Limitations of Neural Networks 265
10.6 Applications of Neural Networks 266
Exercises 269

Chapter 11: Fuzzy Logic 271

11.1 Introduction to Fuzzy Logic 271
11.1.1 Definition of Fuzzy Logic 273
11.1.2 Features of Fuzzy Logic 274
11.1.3 Advantages of Fuzzy Logic 275
11.1.4 Disadvantages of Fuzzy Logic 275
11.2 Crisp Set (Classical set) 276
11.3 Fuzzy Set 277
11.3.1 Linguistic Variables in a Fuzzy Set 281
11.4 Membership Function of Crisp Logic 287
11.5 Membership Function of the Fuzzy Set 287
11.6 Fuzzy Set Operations 291
11.6.1 Union 291
11.6.2 Intersection 291
11.6.3 Complement 292
11.6.4 Equality of Two Fuzzy Sets 293
11.6.5 Containment 293
11.6.6 Normal Fuzzy Set 294
11.6.7 Support of a Fuzzy Set 294
11.6.8 α-Cut or α-Level Set 294
11.6.9 Disjunctive Sum (Exclusive OR) 294
11.6.10 Disjoint Sum 296
11.6.11 Difference 296
11.6.12 The Bounded Difference 297
11.7 Properties of A Fuzzy Set 297
11.8 Differences Between a Fuzzy Set and A Crisp Set 298
11.9 Differences Between Boolean Logic and Fuzzy Logic 302
Exercises 305

Chapter 12: Fuzzy Systems 307

12.1 Fuzzy Rule 307
12.1.1 Fuzzy Rules as Relations 311
12.1.2 Interpretation of Fuzzy Rules 315
12.2 Fuzzy Reasoning 316
Exercises 320

Chapter 13: Fuzzy Expert Systems **321**

13.1 The Need for Fuzzy Expert Systems 321
13.2 Operations on a Fuzzy Expert System 324
13.2.1 Fuzzification (Fuzzy Input) 326
13.2.2 Fuzzy Operator 327
13.2.3 Fuzzy Inferencing (Implication) 327
13.2.4 Aggregate All Output 329
13.2.5 Defuzzification 330
13.3 Fuzzy Inference Systems 332
13.3.1 Mamdani Fuzzy Inference Method 332
13.3.2 Sugeno Inference Method (TSK Fuzzy Model of Takagi, Sugeno, and Kang) 337
13.3.3 Choosing the Inference Method 339
13.4 The Fuzzy Inference Process in a Fuzzy Expert System 340
13.4.1 Monotonic Inference 340
13.4.2 Non-Monotonic Inference 341
13.4.3 Downward Monotonic Inference 341
13.5 Types of Fuzzy Expert Systems 341
13.5.1 Fuzzy Control 341
13.5.2 Fuzzy Reasoning 342
13.6 Fuzzy Controller 342
13.6.1 Components of a Fuzzy Controller 344
13.6.2 Application Areas of Fuzzy Controller 356
Exercises 357

Chapter 14: Logic Programming **359**

14.1 Introduction 359
14.2 Difference Between C/C++ and Prolog 360
14.3 How Does Prolog Work? 361
14.4 A Little History 362
14.5 Converting English to Prolog 363
14.6 Goals 363
14.6.1 How Prolog Satisfies Goals 364
14.7 Queries 365
14.8 Clauses 367
14.8.1 Facts 367
14.8.2 Rules 368
14.9 Notation in Prolog for Building Blocks 371
14.9.1 Atoms 371
14.9.2 Variables 371
14.9.3 Data Types and Structures 372

14.10 Arithmetic Operations 379
14.11 Strings 381
Exercises 382

Chapter 15: Advanced Prolog 383

15.1 Input and Output Predicates 383
15.1.1 Terms and Character I/O 384
15.1.2 File I/O 385
15.2 Backtracking 386
15.2.1 Problems with Backtracking 389
15.3 Cut 390
15.4 Fail 393
15.4.1 Cut and Fail Combination 394
15.5 Recursion 394
15.6 Prolog Data Structure 397
15.6.1 Terms 397
15.6.2 Unification 398
15.7 Dynamic Database 401
15.8 Programs in Prolog 402
15.9 Problems with Prolog 404
Exercises 405

Index 407

PREFACE

Artificial Intelligence (AI) is a branch of computer and information science. The goal of AI is to create a machine that behaves like an ordinary human with an improved machine behavior in tackling complex tasks and to accomplish those tasks in such a way that they would be considered to display "intelligence."

Artificial Intelligence and Expert Systems is a book about the *science* of artificial intelligence. It is designed to help readers in learning about some of the current applications and techniques of AI as an aid to solving problems and accomplishing tasks. The book provides a general introduction to problems and techniques of AI. We have tried to explore the various branches of AI, which encompass formal logic, reasoning, knowledge engineering, expert system neural networks, fuzzy logic, etc. Thus, this book has been structured into parts that benefit the reader in choosing from a variety of paths to the chapters. This book is divided into five parts: problems and state space, knowledge engineering, neural networks, fuzzy logic, and Prolog.

Part I provides introductory concepts on various problems that AI seeks to solve. It introduces a coherent framework in which to understand AI. It also includes a detailed explanation of various state space search algorithms such as best first search, hill climbing, A* algorithms, and uniform search techniques.

Reasoning is one of the important fields of AI that requires a great deal of knowledge about the world in order to solve complex problems and

simulates the decision-making ability of man. Part II explores an important application of AI, i.e. the field of expert systems which was among the first truly successful forms of AI software, designed to solve complex problems by reasoning, like an expert solving complex tasks. This part also introduces the various methods of knowledge acquisition from human domain experts and explores various knowledge representational schemes like predicate logic, propositional logic, frames, scripts and semantic networks.

Part III describes another important branch of AI that is neural networks which are simplified models of the biological neuron system. Neural networks are a parallel distributed processing system that is made by highly interconnected computing elements, used to learn and thereby acquire knowledge. This part provides a detailed explanation of the various forms of artificial neural networks, e.g., single layer feed, forward neural networks, multilayer networks, feed backward networks and various learning methods including supervised, unsupervised, and reinforcement.

Part IV provides basic concepts on fuzzy logic introduced in 1930s by Jan Lukasiewicz. Fuzzy logic is a problem-solving, control system methodology that is used in systems ranging from simple, small, embedded micro-controllers to large, networked, multi-channel PC or workstation-based data acquisition and control systems. Fuzzy logic provides a simple way to arrive at a definite conclusion based upon vague, ambiguous, imprecise, noisy, or missing input information. It also explores the reasoning process in fuzzy logic to derive conclusions from known facts and rules. Further, it provides introductory concepts on a fuzzy expert system to deal with uncertainty and ambiguities that are difficult to deal with in conventional expert systems.

Part V describes Prolog which is a programming language of AI with an ultimate goal of developing code for solving AI problems. Prolog is a declarative (descriptive) language, non-procedural in nature. The programs are written in a way that not only defines how the computational process is to be carried out, but also consists of several declarations representing significant facts and rules. The solution to be mined is also expressed as a question to be answered and a goal to be achieved.

CHAPTER 1

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

We all know that computers are suitable for performing mechanical computations using fixed programmed rules that allow machines to perform simple monotonous tasks efficiently and reliably. Human beings get bored very quickly with monotonous tasks. A computer cannot reason and lacks common sense, and it is difficult for a computer to understand new situations and adapt itself. However, human beings can adapt themselves to new situations since they have the ability to reason. Human beings see through their eyes, and their brains interpret this input to extract the types of objects in the scene. A human being hears a set of voice signals through their ears, and the brain interprets it as a meaningful sentence. Thus, the goal of Artificial Intelligence (AI) is to create a machine that behaves like an ordinary human being and that is an improvement over current machine behavior for tackling complex tasks.

Much of AI research has allowed us to understand our intelligent behavior. Humans have an interesting approach to problem-solving that is based on abstract thought, high-level deliberative reasoning, and pattern recognition. AI can help us understand this process by recreating it, enabling us to enhance our abilities. AI currently includes a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks, such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually

into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives.

Although artificial intelligence as an independent field of study is relatively new, it has some roots in the past. We can say that it started 2,400 years ago when the Greek philosopher Aristotle invented the concept of logical reasoning. The effort to finalize the language of logic continued with Leibniz and Newton. George Boole developed Boolean algebra in the nineteenth century, which laid the foundation of computer circuits. However, the main idea of a thinking machine came from Alan Turing, who proposed the Turing test. In 1950, Alan Turing proposed the Turing test, which provides a definition of intelligence in a machine. The term "artificial intelligence" was first coined by John McCarthy in 1956.

1.1 The Turing Test

The English mathematician Alan M. Turing devised a test to determine whether a computer can be said to think like a human. The test was named after Turing, who founded artificial intelligence during the 1940s and 1950s. The original version of the test asked the question "Can machines think?" According to this test, a computer is deemed to have artificial intelligence if it can mimic human responses under specific conditions. In Turing's test, if the human conducting the test is unable to consistently determine whether an answer has been given by a computer or by another human being, then the computer is considered to have "passed" the test. In the basic Turing test, there are three terminals. Two of the terminals are operated by humans, and the third terminal is operated by a computer. Each terminal is physically separated from the other two. One human is designated as the questioner (interrogator). The other human and the computer are designated the respondents. The questioner interrogates both the human respondent and the computer according to a specified format, within a certain subject area and context, and for a pre-set length of time (such as 10 minutes). The test simply compares the intelligent behavior of a human being with that of a computer. An interrogator asks a set of questions that are forwarded to both the computer and the human. The interrogator receives two sets of responses, but does not know which set comes from the human and which set from the computer. After a careful examination of the two sets, if the interrogator cannot definitely tell which set has come from the computer

and which from the human, the computer has passed the Turing test for intelligent behavior. However, the test is not as straight-forward as it seems because humans are superior to computers in creativity, common sense, and reasoning. If the test uses any question that is related to these concepts, then the human is sure beat the computer. Computers are more accurate and faster at performing computations than humans. The Turing test is a test that a machine should pass in order to be called intelligent.

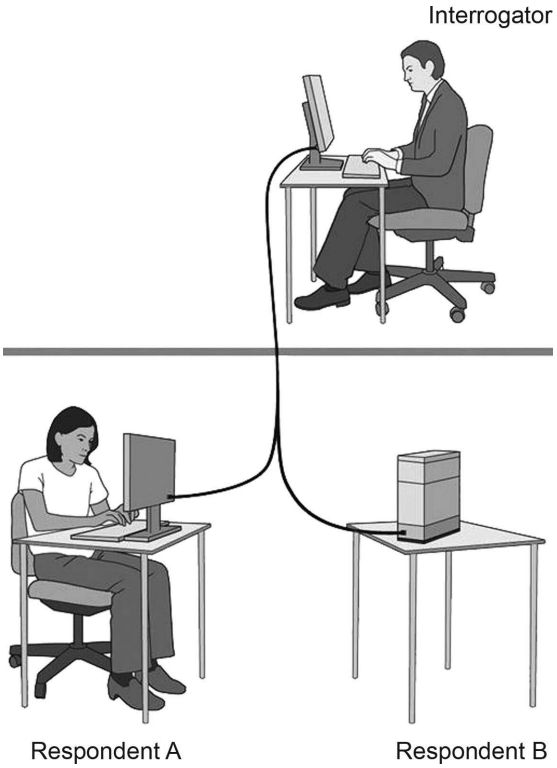


FIGURE 1.1 An example of the Turing test, in which the Interrogator must determine which respondent is the computer

There are some criticisms of the Turing test:

- A machine could pass the Turing test, but it is a different matter to know what the level of proficiency the machine actually has.
- Searle proposed an argument called the "Chinese room argument" to bring attention to a major flaw in the Turing test. According to this

argument, Searle did not know Chinese and was locked in room with set of Chinese letters. He was given some writing in Chinese with instructions in English that correlated to the first and second set of symbols. He was also given a set of questions for answering (which was supplemented by instructions in English). He claimed that he could manipulate the Chinese symbols in a formal way and provide a satisfactory answer to people outside the room that would create the illusion that he knew Chinese. Searle argued that a machine that passes the Turing test and is assumed to be intelligent actually behaves in the same fashion (it manipulates formal symbols with a lack of understanding).

- The Turing test has been criticized because the nature of the questioning must be limited in order for a computer to exhibit human-like intelligence. For example, a computer might score high when the questioner formulates the queries so they have "Yes" or "No" answers and pertain to a narrow field of knowledge, such as mathematical number theory. If the responses to the questions are of a broad-based, conversational nature, however, a computer would not be expected to perform like a human being. This is especially true if the subject is emotionally charged or socially sensitive.
- In some specialized instances, a computer may perform so much better and faster than a human that the questioner can easily tell which is which. Google and Yahoo are examples of computer applications that outperform a human in a Turing test based on information searches.

These arguments highlight the deficiencies of the Turing test and raise the question "What is intelligence?"

Intelligence: This is the ability to reason, develop new thoughts, perceive, and learn. Psychologists have proposed various definitions, but there is no consensus on any particular definition.

The term "thought" can be defined as a mechanism which

a) stimulates

- action
- information generation
- knowledge generation

b) is triggered by
- external stimulus
- internal stimulus

c) acts through
- present environment
- past memory

d) is stored as
- the charged/discharged state of neurons
- electromagnetic thought waves

1.2 Intelligent Agents

An intelligent agent is a system that perceives its environment, learns from it, and interacts with it intelligently. Intelligent agents can be divided into two broad categories: software agents and physical agents.

1.2.1 Software Agents

A software agent is a set of programs that is designed to do particular tasks. For example, a software agent can check the contents of received e-mails and classify them into different categories (junk, less important, important, very important, and so on). Another example of a software agent is a search engine used to search the World Wide Web and find sites that can provide information about a requested subject.

1.2.2 Physical Agents

A physical agent (robot) is a programmable system that can be used to perform a variety of tasks, e.g., simple robots can be used in manufacturing industries for performing various routine jobs such as assembling, welding, or painting. Some organizations use mobile robots for performing routine delivery jobs, such as distributing mail or correspondence to different rooms. Mobile robots are used underwater to prospect for oil.

AI is the branch of computer science which aims to make computers behave like human beings. The term was coined in 1956 by John McCarthy at the Massachusetts Institute of Technology. The various areas of artificial intelligence include

- **game playing:** programming computers to play games, such as chess and checkers
- **expert systems:** programming computers to make decisions in real-life situations (for example, some expert systems help doctors diagnose diseases based on symptoms)
- **natural language:** programming computers to understand natural human languages
- **neural networks:** systems that simulate intelligence by attempting to reproduce the types of physical connections that occur in animal brains
- **robotics:** programming computers to see and hear and react to other sensory stimuli.

There are many different approaches to artificial intelligence, none of which are completely right or wrong. Through the years, new techniques have emerged based on the state of mind of the researchers, funding opportunities, and the available computer hardware.

Over the past five decades, AI research has mostly focused on solving specific problems. Many solutions have been proposed, and there have been improvements in the efficiency and reliability of these solutions. AI is divided into many fields, ranging from pattern recognition to artificial life. AI is a broad discipline that simulates human skills such as automatic programming, case-based reasoning, neural networks, decision-making, expert systems, natural language processing, pattern recognition, and speech recognition. AI technologies bring more complex data analysis features to existing applications.

Currently, no computers exhibit full artificial intelligence (a simulation of human behavior). The greatest advances have occurred in the field of game playing. The best computer chess programs are now capable of beating humans. In May, 1997, an IBM super-computer called Deep Blue defeated world chess champion Gary Kasparov in a chess match. Computers are now widely used in assembly plants, but they are capable only of very limited tasks. Robots have great difficulty identifying objects based on appearance or feel, and they still move and handle objects clumsily.

There are several programming languages that are known as AI languages because they are used almost exclusively for AI applications. The two most common are LISP and Prolog.

1.0 Artificial Intelligence Characteristics	
Systems that act like humans	Systems that act rationally

1.3 Approaches in Artificial Intelligence

1.3.1 Acting Humanly: The Turing Test Approach

AI is a system that "thinks" like humans and can be explained using the Turing test. The Turing Test was designed to provide a satisfactory operational definition of intelligence. Turing defined intelligent behavior as the ability to achieve human-level performance in all cognitive tasks, sufficient to fool an interrogator. The test he originally proposed is that the computer should be interrogated by a human via a teletype, and the computer passes the test if the interrogator cannot tell if there is a computer or a human at the other end. The computer would need to possess the following capabilities to pass the Turing test:

- natural language processing to enable it to communicate successfully in English (or some other human language)
- knowledge representation to store information provided before or during the interrogation
- automated reasoning to use the stored information to answer questions and draw new conclusions
- machine learning to adapt to new circumstances and to detect and extrapolate patterns

Turing's test deliberately avoided direct physical interaction between the interrogator and the computer because the physical simulation of a person is unnecessary for intelligence. However, the Total Turing Test includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch." To pass the Total Turing Test, the computer will need

- computer vision to perceive objects
- robotics to move them about

Within AI, there has not been a big effort to try to pass the Turing test. The issue of acting like a human comes up primarily when AI programs have to interact with people, as when an expert system explains how it came to its diagnosis or a natural language processing system has a dialogue with a user. These programs must behave according to certain normal conventions of human interactions in order to make themselves understood. The underlying representation and reasoning in such a system may or may not be based on a human model.

1.3.2 Thinking Humanly: The Cognitive Modelling Approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get inside the actual workings of human minds. There are two ways to do this: through introspection—trying to catch our own thoughts as they go by—or through psychological experiments. Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input/output and timing behavior matches human behavior, that is evidence that some of the program's mechanisms may also be operating in humans. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind. Real cognitive science, however, is necessarily based on the experimental investigation of actual humans or animals, and we assume that the reader only has access to a computer for experimentation. We will simply note that AI and cognitive science continue to enrich each other, especially in the areas of vision, natural language, and learning.

1.3.3 Thinking Rationally: The Laws of Thought Approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking," that is, unquestionable reasoning processes. His famous syllogisms provided patterns for argument structures that always gave correct conclusions given correct premises. For example, "Socrates is a man; all men are mortal; therefore, Socrates is mortal." These laws of thought were supposed to govern the operation of the mind and initiated the field of logic.

The development of formal logic in the late nineteenth and early twentieth centuries, which we describe in more detail in the next chapters, provided a precise notation for statements about all kinds of things in the world and the relationships between them. By 1965, programs existed that could, given enough time and memory, take a description of a problem in logical notation and find the solution to the problem, if one existed. (If there is no solution, the program might never stop looking for it.) There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between being able to solve a problem in principle and doing so in practice.

1.3.4 Acting Rationally: The Rational Agent Approach

Acting rationally means acting so as to achieve one's goals given one's beliefs. An agent is just something that perceives and acts. (This may be an unusual use of the word, but you will get used to it.) In this approach, AI is viewed as the study and construction of rational agents.

In the "laws of thought" approach to AI, the whole emphasis was on correct inferences. Making correct inferences is sometimes part of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals, and then to act on that conclusion. On the other hand, correct inference is not all of rationality; because there are often situations where there is no provably correct thing to do, yet something must still be done. There are also ways of acting rationally that cannot be reasonably said to involve inference. For example, pulling one's hand off of a hot stove is a reflex action that is more successful than a slower action taken after careful deliberation.

All the "cognitive skills" needed for the Turing test are there to allow for rational actions. Thus, we need the ability to represent knowledge and reason with it because this enables us to reach good decisions in a wide variety of situations. We need to be able to generate comprehensible sentences in natural language because saying those sentences helps us get by in a complex society. We need learning not just for erudition, but because having a better idea of how the world works enables us to generate more effective strategies for dealing with it. We need visual perception not just because seeing is fun, but in order to get a better idea of what an action might achieve—for example, being able to see a tasty morsel helps one to move toward it.

The study of AI as the design of a rational agent therefore has two advantages. First, it is more general than the "laws of thought" approach because correct inference is only a useful mechanism for achieving rationality, and not a necessary one. Second, it is more amenable to scientific development than approaches based on human behavior or human thought because the standard of rationality is clearly defined and completely general. Human behavior, on the other hand, is well-adapted for one specific environment and is the product, in part, of a complicated and largely unknown evolutionary process that still may be far from achieving perfection.

1.4 Definitions of Artificial Intelligence

A number of definitions have been proposed for AI:

- AI is a technology and a branch of computer science that studies and develops intelligent machines and software.
- Software technologies that make a computer or robot perform equal or better than normal human computational ability in accuracy, capacity, and speed.
- Artificial intelligence is a branch of science which deals with helping machines find solutions to complex problems in a human-like fashion. This generally involves borrowing characteristics from human intelligence and applying them as algorithms in a computer-friendly way.
- Artificial intelligence is the study of programmed systems that can simulate, to some extent, human activities such as perceiving, thinking, learning, and acting.
- "The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990)
- "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)
- "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes." (Schalkoff, 1990)
- "The branch of computer science that is concerned with the automation of intelligent behavior." (Luger and Stubblefield, 1993).

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

There are two ideas in the definition.

1. Intelligence
2. Artificial device

What is Intelligence?

- Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals, and some machines.
- The capacity to learn and solve problems

In particular,

- the ability to solve novel problems
- the ability to act rationally
- the ability to act like humans
    - ♦ A system with intelligence is expected to behave as intelligently as a human
    - ♦ A system with intelligence is expected to behave in the best possible manner

What is involved in intelligence?

- The ability to interact with the real world
    - ♦ to perceive, understand, and act
    - ♦ e.g., speech recognition and understanding and synthesis
    - ♦ e.g., image understanding
    - ♦ e.g., the ability to take actions, to have an effect
- Reasoning and Planning
    - ♦ modeling the external world, given input
    - ♦ solving new problems, planning, and making decisions
    - ♦ the ability to deal with unexpected problems and uncertainties

- Learning and Adaptation
  - ♦ We are continuously learning and adapting.
  - ♦ Our internal models are always being "updated."
  - ♦ One example is a baby learning to categorize and recognize animals.

1.4.1 Intelligent Behavior

Tasks and applications that constitute intelligent behavior are

- perception involving image recognition and computer vision
- reasoning
- learning
- understanding language involving natural language processing and speech processing
- solving problems
- robotics

1.4.2 Interpretations of Artificial Intelligence

Different interpretations have been used by researchers for defining the scope and view of AI.

a) One view is that artificial intelligence is about designing systems that are as intelligent as humans. This view means that we should try to understand human thought and build machines that simulate the human thought process. This view is the cognitive science approach to AI.

b) The second approach is best defined by the concept of the Turing test. The Turing test is a kind of imitation game, in which a human being and a computer are interrogated under conditions where the interrogator does not know which is machine and which is human. The communications are carried out entirely via text messages. Turing argued that if the interrogator could not distinguish them by questioning, then it would be unreasonable not to call the computer intelligent. Turing's imitation game is the Turing test.

c) The third view of AI is that it is the study of rational agents. This view deals with building machines that act rationally. The focus is on how the system acts and performs, and not so much on the reasoning process. A rational agent is one that acts rationally, that is, is in the best possible manner.

1.5 AI Problems

While studying the typical range of tasks that we might expect an intelligent entity to perform, we need to consider both common-place tasks as well as expert tasks.

a) A lot of work in AI is focused on formal tasks such as game playing and theorem proving. Samuel wrote a checker playing program that not only plays the game with opponents, but also uses its experience to improve its later performance. Such types of tasks require intelligence, so people who do these tasks well are considered intelligent. It may seem like a computer could perform such tasks well by exploring a large number of solution paths fast and select best one. But this assumption is false, since no computer is fast enough to overcome the combinatorial explosion generated by most problems.

b) Another focus in AI is solving everyday tasks that require common sense reasoning. This includes reasoning about physical objects and their relationship to each other, as well as reasoning about actions and consequences.

c) As AI research has progressed, techniques have been developed for handling a large amount of knowledge. Progress was made in handling more complex tasks such as perception, natural language understanding, and diagnosis problems.

d) Animal have less intelligence than humans, but have more sophisticated visual perception. Perceptual tasks are difficult because they involve analog signals (noisy signals).

e) Natural language understanding is a problem. In addition to these mundane tasks, people may perform one or more specialized tasks that require expertise, such as engineering design tasks, medical diagnosis, and scientific discovery tasks.

1.5.1 Tasks Under Artificial Intelligence

There are tasks done routinely by humans and animals. Examples of common-place tasks include

- recognizing people and objects
- communicating (through natural language)
- navigating around obstacles on the streets

Examples of expert tasks include

- medical diagnosis
- mathematical problem solving
- playing games like chess

Expert tasks cannot be done by all people; they can only be performed by skilled specialists. Clearly tasks of the first type are easy for humans to perform, and almost all are able to master them. The second range of tasks requires skill development and/or intelligence. Only some specialists can perform them well. The achievements of computer systems include performing sophisticated tasks like making a medical diagnosis, performing symbolic integration, proving theorems, and playing chess.

However, it has proven to be very difficult to make computer systems perform many routine tasks that all humans and a lot of animals can do. Examples of such tasks include navigating our way without running into things, catching prey, and avoiding predators. Humans and animals are also capable of interpreting complex sensory information. We are able to recognize objects and people from the visual image that we receive. We are also able to perform complex social functions.

1.5.2 Tasks Domains of Artificial Intelligence

Mundane Tasks

1. Perception
   - Vision
   - Speech

2. Natural Language
   - Understanding
   - Generation
   - Translation
3. Common Sense Reasoning
4. Robot Control

Formal Tasks

1. Games
   - Chess
   - Backgammons
   - Checkers-go
2. Mathematics
   - Logic
   - Geometric
   - Integral calculus

Expert Tasks

1. Engineering
   - Design
   - Fault Finding
     - ♦ Medical Diagnosis
     - ♦ Financial Analysis
     - ♦ Scientific Analysis

The fields of AI are domains that require specialized expertise without the help of common sense reasoning.

Research into AI shows that intelligence requires knowledge. It is the basic thrust behind every intelligent system. The properties of knowledge are

1. It is voluminous.
2. It is hard to characterize accurately.
3. It is constantly changing.
4. It is well organized and corresponds to the way it will be used.

The AI technique is a method that exploits knowledge that should be represented in such a way that

- Knowledge captures generalization. It is not necessary to represent each individual situation separately. Situations that share important properties are grouped together. Otherwise, a lot of memory and updating would be required.
- It can be understood by the people who must provide it.
- It can be easily modified to correct errors.
- It can be used in many situations even if it is not totally accurate.

1.6 Features of AI Programs

a) AI problems have combinatorial explosions of solutions.

b) AI programs manipulate symbolic information to a large extent, in contrast to conventional programs that deal with numeric processing.

c) AI programs use heuristic search techniques to solve problems and prune search trees. One of the techniques for solving problems in artificial intelligence is searching. Searching can be described as solving a problem using a set of states (a situation). A search procedure starts from an initial state and goes through the intermediate states until finally reaching a target state. For example, in solving a puzzle, the initial state is the unsolved puzzle, the intermediate states are the steps taken to solve the puzzle, and the target state is the situation in which the puzzle is solved. The set of all states used by a searching process is referred to as the search space.

d) An AI program must have large quantities of knowledge that must be represented in a form such that a system working on the knowledge can easily manipulate it.

e) AI programs deal with real life problems. AI programs help people make the right decisions.

f) AI programs have the ability to learn.

1.7 Importance of AI

a) Organizations that use AI applications become more diverse because these applications provide the ability to analyze data across multiple variables, and can help with fraud detection and customer relationship management. All such things are very important from a competitive point of view.

b) AI is a branch of science that deals with helping machines that find solutions to complex problems in a human-like fashion by borrowing characteristics from human intelligence and applying them as algorithms in a computer-friendly way.

c) AI is generally associated with computer science, but it has its roots in variety of fields, such as math, psychology, cognition, biology, and philosophy. Thus, combining knowledge from all these fields benefits the development of an intelligent artificial being.

d) AI is a machine that can behave like an ordinary human. One of the meanings of the word "perception" is understanding what is received through the senses—sight, hearing, touch, smell, and taste. A human being sees a scene through the eyes, and the brain interprets it to extract the type of objects in the scene. A human hears a set of voice signals through the ears, and the brain interprets it as a meaningful sentence.

1.8 What Can Artificial Intelligence Systems Do?

Today's AI systems have been able to achieve limited success in some of these tasks:

- In computer vision, the systems are capable of facial recognition.
- In robotics, we have been able to make vehicles that are mostly autonomous.

- In natural language processing, we have systems that are capable of simple machine translation.
- Today's expert systems can carry out medical diagnoses in a narrow domain.
- Speech understanding systems are capable of recognizing several thousand words of continuous speech.
- Learning systems are capable of performing text categorization into about 1,000 topics.
- AI systems can play games at the Grand Master level in chess (world champion) and checkers.

1.9 What Can Artificial Intelligence Systems Not Do Yet?

- Understand natural language robustly (e.g., read and understand articles in a newspaper).
- Surf the web.
- Interpret an arbitrary visual scene.
- Learn a natural language.
- Construct plans in dynamic real-time domains.
- Exhibit true autonomy and intelligence.

1.10 Advantages of AI

a) AI is used in various areas like diagnosis, medicine, image processing, and game playing; complex tasks in such fields can be performed efficiently and reliably.

b) AI can perform multiple tasks at once, such as tasks that would be too difficult or time consuming when carried out by humans. These tasks include mathematical equations that are used to design and operate video games or autopilots used by airplanes to fly planes in normal situations and aid the crew in emergencies.

c) AI helps in the mass production of industrial parts to make sure parts are accurate and to specifications.

d) AI makes life safer and more pleasurable for people at every stage of modern life.

e) AI machines help in the continuity of work in various fields, as the machines can constantly monitor complex situations.

f) AI can take on stressful and complex work that humans may struggle with or cannot do. Machines have no need for sleep, they don't get ill, and there is no need for breaks. Doing tasks without getting tired is a significant benefit offered by artificial intelligence. AI can get a specific task finished without a coffee break or lunch break, unlike humans, who require a break. A machine can also complete a particular job almost instantly.

g) AI can replace human beings in some specific jobs at stores and perform some of a household's day-to-day activities, helping to address manpower problems.

h) AI can help hospitals providing food and medicines where humans may be exposed to disease. AI has its application in variety of fields, such as robotics. Robots can be used in manufacturing industries or other industries for doing tasks that are harmful to humans.

i) AI helps researchers in aeronautics better know the universe.

1.11 Disadvantages of Artificial Intelligence

a) It is true that AI has lot of advantages in various fields (such as in chess, where a computer can beat a human). Expert systems assist industry with a wide range of diagnostic software. Robots are used to perform complex and dangerous work. Optical character recognition and speech recognition have advanced enough to have many practical applications. AI has disadvantages, though. For example, without a massive amount of storage, the simultaneous real-time retrieval of multisensory data is out of reach. Natural language processing suffers from this problem. It requires an understanding of language, culture, history, and emotions to be able to translate a sentence. A robot does not have the common sense and the reasoning power to understand a word like "outside." Common sense requires a large amount of knowledge.

b) The main disadvantage of AI is that it lacks the pattern recognition tools needed to succeed. The study of AI began formally at Dartmouth College in 1956 as an effort by a group of scientists to evaluate and mechanically replicate human intelligence on the assumption that "every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it." Their objective was to write computer programs, which could finally create human level intelligence in computers and robots. Those early scientists failed to realize that the mind uses pattern recognition and not computation. They also underestimated the memory storage capacity required to achieve such an ambitious objective.

c) Nature has provided a large memory capacity in humans to sustain life on an unimaginable scale. The evolutionary process logically assembled these in cell memories. The DNA of every living thing on the planet has digital, error-correcting, and self-replicating codes. These vast blueprints improved with each generation across millions of years. AI's disadvantage is that it presently lacks the means to store a comparable size of memory and also lacks a clear strategy for instantly accessing this enormous memory store.

d) Nature has assembled ascending levels of knowledge in the immune system, the spinal cord, the reticular system, the limbic system, and the prefrontal regions. Millions of potentially pathogenic organisms and substances had to be neutralized. "Knowledge" in the spinal cord coordinates the movements of muscles millisecond by millisecond. Memories for myriad smells enabled the reptilian systems to distinguish between prey and predator. "Knowledge" in the limbic system responded suitably to a wide range of events, which trigger anger and fear, or jealousy and despair. The disadvantage of AI is that the computation capabilities of a computer cannot manage the pattern sensing responses of living things, who have assembled this knowledge over millions of years of history and a lifetime of experience, play, and imagination.

e) Human intelligence adds new levels to animal intelligence. The great achievements in science and art are based on the stored knowledge of millions of relationships between numerous fields. A work of art is only possible through an immense number of inherited skills and through practice, training, and experience. AI has barely touched on these complex pattern sensing tasks.

f) The possibility of a breakdown is one of the most infamous disadvantages of artificial intelligence. It is like spending much of your money on a car in order to get from one point to another and then needing to deal with the breakdown of the car shortly after buying it. It is the same way for artificial intelligence: it can easily perform a task, but a malfunction can turn the whole thing into nothing.

g) Aside from the possibility of a breakdown, there is also the possibility of losing your essential information. In some cases, because of the malfunction of specific parts, an artificial mind can fall short in keeping in its memory all the files that it must have. This can also occur with humans. If the person who is responsible for maintaining information and collecting data falls asleep on the job, it is accepted that the failure is that person's mistake. On the other hand, with an artificial mind, it is not assumed, and this really makes the entire difference. This then becomes an important issue. AI or computer systems must be switched off on a daily basis for maintenance. This could be a restraint to output and efficiency, as well as to the interests and benefits of the company in question.

h) AI fails in the speed of knowledge retrieval. An animal mind stores the equivalent of billions of pages of code. This data is evaluated and acted on within milliseconds. The unconscious processes of your immune system utilize internal code recognition systems to attack a detected invader. The olfactory system, using an inbuilt knowledge of smells, enables an animal to instantly recognize a scent and sense danger.

AI cannot compete in the field of real time information retrieval achieved by animals. To succeed, artificial intelligence requires myriad pattern sensing algorithms and the ability to extract contextual knowledge in real time from a vast amount of coded memories.

Exercises

Q1. What is Artificial Intelligence?

Q2. What are the various areas where AI (Artificial Intelligence) can be used?

Q3. Give an explanation of the difference between a strong AI and weak AI.

Q4. Is intelligence a single thing, so that one can ask the "yes or no" question "Is this machine intelligent or not?"

Q5. Is AI about simulating human intelligence?

Q6. What about other comparisons between human and computer intelligence?

Q7. What is the Turing test?

Q8. What are the task domains of AI?

Q9. What are intelligence and intelligent agents?

Q10. What is the cognitive modeling approach?

CHAPTER 2

APPLICATIONS OF ARTIFICIAL INTELLIGENCE

There are many applications of AI, as it can be used in a variety of fields for solving complex problems. Applications range from military uses (for autonomous control and target identification) to the entertainment industry (for computer games and robotic pets). AI has been used in medical diagnosis, stock trading, robot control, law, remote sensing, scientific discovery, and toys.

The various applications areas are as follows.

2.1 Finance

Banks use artificial intelligence systems for organizing operations, investing in stocks, and managing properties. Financial institutions use artificial neural network systems to detect charges or claims outside of the norm and identify these for human investigation.

2.2 Hospitals and Medicine

Hospitals use artificial intelligence systems to organize bed schedules, perform staff rotations, and provide medical information. Artificial neural networks are used for medical diagnosis.

Other tasks in medicine that can be performed by artificial intelligence include the following:

a) Artificial intelligence systems are used for analyzing medical images to detect diseases. Such systems help scan digital images, such as those from computed tomography. A typical application is the detection of tumors.

b) Heart sound analysis

2.3 Robotics

Robotics is the branch of technology that deals with the design, construction, operation, and application of robots, as well as computer systems for their control, sensory feedback, and information processing. Some robots are used in performing dangerous tasks in the manufacturing industry.

A robot is a mechanical or virtual agent, usually an electro-mechanical machine, that is guided by a computer program or electronic circuitry. A robot performs only those tasks for which it is programmed. However, an intelligent robot has sensors, such as cameras, which allow it to respond to changes in the environment.

2.4 Expert Systems

An expert system is a computer program that simulates the judgment and behavior of a human with expert knowledge and experience in a particular field. Expert systems contain a knowledge base of accumulated experience and a set of rules that are applied to each particular situation. Sophisticated expert systems can be enhanced with additions to the knowledge base or to the set of rules.

Expert systems are applications of AI that utilize human expertise. Expert systems are used to solve complex problems with help of the expertise stored in the database in rule form. To design an expert system, we need a knowledge engineer, an individual who studies how human experts make decisions and translates the rules into terms that a computer can understand. Expert systems are also known as knowledge-based systems, knowledge-based expert systems, and rule-based systems. They are considered to be "applied artificial intelligence." The process of developing with an expert system is knowledge engineering.

EMYCIN was one of the first "shells" for an expert system, which was created from the MYCIN medical diagnosis system. A production rule system is a rule engine that uses the rule-based approach to implement an expert system.

Expert systems use knowledge representation languages to perform tasks that normally need human expertise. For example, in medicine, an expert system can be used to narrow down a set of symptoms to a likely subset of causes, a task normally carried out by a doctor.

An expert system is built on predefined knowledge about a field of expertise. An expert system in medicine, for example, is built on the knowledge of a doctor specialized in the field for which the system is built: an expert system is supposed to do the same job as the human expert.

2.5 Diagnosis

Diagnosis deals with the development of algorithms and techniques that are able to determine whether the behavior of a system is correct. If the system is not functioning correctly, the algorithm should be able to determine, as accurately as possible, which part of the system is failing, and which kind of fault it is facing.

An example of making a diagnosis is the process a garage mechanic uses with an automobile. The mechanic will first try to detect any abnormal behavior based on the observations of the car and his knowledge of this type of vehicle. If he finds out that the behavior is abnormal, the mechanic will try to refine his diagnosis by using new observations and possibly testing the system until he discovers the faulty component. The mechanic plays an important role in the vehicle's diagnosis.

2.6 Pattern Recognition

Pattern recognition is the process of establishing a close match between new stimuli and a previously stored pattern. Pattern recognition systems are used to classify objects based on their attributes and attribute values.

In pattern recognition, a label is assigned to a given input value. An example of pattern recognition is classification, which attempts to assign each input value to one of a given set of classes (for example, determine

whether a given email is "spam" or "non-spam"). Pattern recognition algorithms generally aim to provide a reasonable answer for all possible inputs and to perform the "most likely" matching of the inputs, taking into account their statistical variation.

Pattern recognition is generally categorized according to the type of learning procedure used to generate the output value.

Supervised learning assumes that a set of training data (the training set) has been provided, and it consists of a set of instances that have been properly labeled by hand with the correct output.

Unsupervised learning assumes training data has not been hand-labeled, and attempts to find inherent patterns in the data that can then be used to determine the correct output value for new data instances. A combination of the two is called semi-supervised learning, which uses a combination of labeled and unlabeled data (typically a small set of labeled data combined with a large amount of unlabeled data).

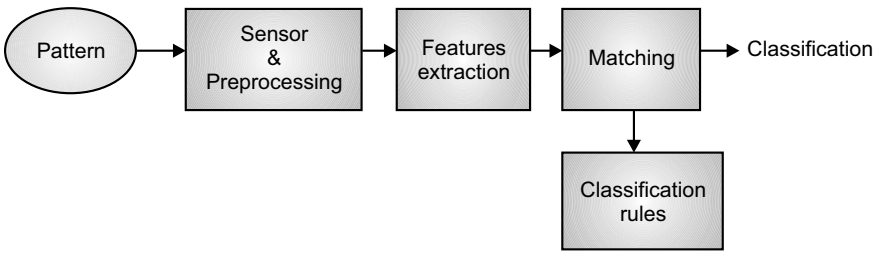


FIGURE 2.1 Pattern Recognition Process

2.7 Natural Language Processing

Natural language processing (NLP) is used for analyzing and representing natural text at one or more levels of linguistic analysis to obtain human-like language processing. NLP is related to human–computer interactions.

NLP is a branch of artificial intelligence that deals with analyzing, understanding, and generating the natural languages humans use. One of the challenges in NLP is teaching computers to understand the way humans learn and use language.

For example, consider the sentence "Baby swallows fly." This simple sentence has multiple meanings, depending on whether the word "swallows" or the word "fly" is used as the verb, which also determines whether "baby" is used as a noun or an adjective. In the case of human communication, the meaning of the sentence depends on the context in which it was communicated. This sentence presents problems for software, which must first be programmed to understand the context and linguistic structures.

Computers can't understand natural language, so researchers are trying to make them more intelligent. NLP is divided into following subfields:

a) natural language understanding

b) analysis of language to provide meaningful representation

c) natural language generation

d) production of language from representation

Steps in NLP:

a) The first step in natural language processing is speech recognition. In this step, a speech signal is analyzed and the sequence of words it contains is extracted. The input to the speech recognition subsystem is a continuous (analog) signal: the output is a sequence of words. The signal needs to be divided into different sounds, sometimes called phonemes. The sounds then need to be combined into words.

b) The syntactic analysis step is used to define how words are to be grouped in a sentence. This is a difficult task in a language like English, in which the function of a word in a sentence is not determined by its position in the sentence. For example, consider the following two sentences.

Mary rewarded John.

John was rewarded by Mary.

c) It is always John who is rewarded, but in the first sentence, John is in the last position and Mary is in the first position. A machine

that hears any of the above sentences needs to interpret them correctly and come to the same conclusion, no matter which sentence is heard.

d) The semantic analysis extracts the meaning of a sentence after it has been syntactically analyzed. This analysis creates a representation of the objects involved in the sentence, their relationships, and their attributes. The analysis can use any of the knowledge representation schemes. For example, the sentence "John has a dog" can be represented using predicate logic.

$$\exists x \, \mathbf{dog}(x) \ \mathbf{has}\ (\mathbf{John}, x)$$

The three previous steps—speech recognition, syntax analysis, and semantic analysis—can create a knowledge representation of a spoken sentence. In most cases, another step, pragmatic analysis, is needed to further clarify the purpose of the sentence and remove ambiguities.

2.8 Game Playing

In 1960, Arthur Samuel built the first game playing program, which learned from its mistakes and improved its performance. Game playing has great role in AI because of the following:

- Rules are limited and little knowledge is required.
- Games provide structural tasks that are easy to measure as a success or failure.
- Game playing simulates real-life situations.

2.9 Image Processing

Image processing is any form of signal processing for which the input is an image, such as photograph or video frame; the output of image processing may be either an image or a set of characteristics or parameters related to the image. Most image-processing techniques involve treating the image as a two-dimensional signal and applying standard signal-processing techniques to it.

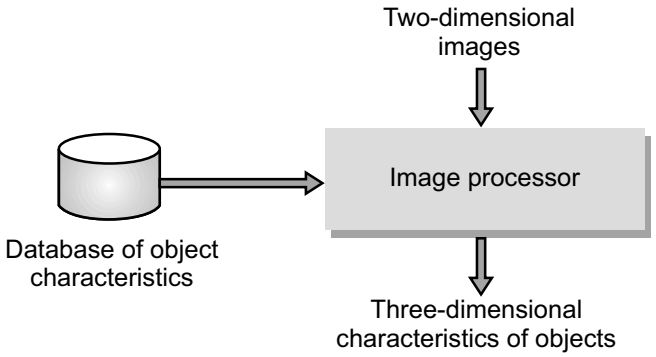


FIGURE 2.2 The Components of an Image Processor

Image processing, or computer vision, is an area of AI that deals with the perception of objects through the artificial eyes of an agent, such as a camera. An image processor takes a two-dimensional image from the outside world and tries to create a description of the three-dimensional objects present in the scene. This is an easy task for a human being, but a difficult task for an artificial agent. The processor uses a database containing the characteristics of objects for comparison.

Image processing usually refers to digital image processing, but optical and analog image processing also are possible.

Image processing basically includes the following three steps:

- Importing the image with an optical scanner or by digital photography
- Analyzing and manipulating the image, which includes data compression and image enhancement, and spotting patterns that are not obvious to the human eyes (like satellite photographs)
- Output is the last stage in which a result can be an altered image or report that is based on image analysis.

The purpose of image processing is

- **Visualization:** observe the objects that are not visible
- **Image sharpening and restoration:** to create a better image
- **Image retrieval:** seek for the image of interest
- **Measurement of a pattern:** measures various objects in an image

- **Image recognition:** distinguish the objects in an image
- **Color correction,** such as the brightness adjustment and contrast adjustments.

2.10 Data Mining

Data mining is a field of computer science that deals with the computational process of discovering patterns in large data sets. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use.

Data mining uses information from past data to analyze the outcome of a particular problem or situation that may arise. Data mining works to analyze the data stored in data warehouses, which are used to store the data for analysis. That particular data may come from all parts of a business, from production to management. Managers also use data mining to decide upon marketing strategies for their product. They can use data to make comparisons with competitors. Data mining interprets data using real-time analysis; the results of the analysis can be used to increase sales, promote new products, or eliminate products that are not adding value to the company.

2.11 Big Data Mining

Today's advancements in technology, like cloud computing, sensors, and data wireless networks, have dramatically increased the size of the Internet, and societal transformation makes it possible to capture a large amount of data ("big data") that has the potential to reveal meaningful and valuable information for fields like business, industry, healthcare, agriculture, finance, weather forecasting, scientific research, astronomy, transportation, and even societal development. There are a growing number of ways to generate this data, such as sensors, automated processes, multimedia, cameras, satellites, telescopes, transceivers, and mobile phones. The exponential growth in the generation of data is bolstered by the Internet, where everything is recorded. Each and every activity of a user on the Internet is generating data. For example, when you do any surfing, all of that activity is recorded. When shopping online, customer behaviors, buying patterns, items viewed by customers, and items discarded by

customers are recorded; each and every detail, whether small or large, is recorded. Through proper analysis, such captured data helps in predicting trends and buying patterns so that the appropriate decisions can be made and strategies can be developed. Google, Amazon, Twitter, and Facebook are the first companies to face the exponential growth of data from the Internet. They developed solutions for dealing with that enormous growth of data. Big data is collection of extremely large data sets that helps in decision making through the proper analysis of patterns and trends.

Exercises

Q1. What is the importance of artificial intelligence in expert systems?

Q2. Explain pattern recognition.

Q3. What is meant by image processing?

Q4. Explain role of AI in big data mining.

CHAPTER 3

INTRODUCTION TO THE STATE SPACE SEARCH

The state-space search paradigm received early attention from Newell and Simon, who developed a system called GPS (the General Problem Solver) in 1960. Search is an integral component of numerous computer programs, so search techniques are heavily studied in computer science. Search is often used in AI within the context of the state space search. Let's look at this technique in detail.

A search algorithm takes a problem as input and returns a solution in the form of an action sequence. The state space concept often provides the framework.

8 Puzzle Space

To generate an intuitive understanding of the state space concept, let's look at a fairly simple example, the 8 puzzle. The fairly simple 8 puzzle can be made more interesting by setting it in three dimensions instead of two. Since it is not easy to represent a three-dimensional puzzle and, even then, one cannot see all 27 blocks at once, it can be rendered as three two-dimensional puzzles.

8 Puzzle State Space

It is easy to see that each move changes the configuration of the tiles. Each such configuration is called a state. From any configuration, one can make either two, three, or four moves. Of course, each move leads to a new

configuration. Thus, the states (configurations) are related to each other by moves. At this point, we can build upon the concepts of discrete mathematics. Each state can be represented by a vertex and each move by an edge, giving us an undirected graph. The graph is undirected since each move is reversible.

The graph thus produced is called the state space. One further note: computer scientists usually call the vertices "nodes" and the edges "arcs."

3.1 State Space Search

We originate a problem as a state space search by viewing the legal problem states, legal operators, and initial and goal states.

- A state is defined by the requirement of the values of all the attributes of interest in the world.
- An operator changes one state into other; it has a pre-condition, which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator.
- The initial state is where you start.
- The goal state is the incomplete explanation of the solution.

State Space Search Representation

Let us begin by introducing certain terms.

Initial state: This is the description of the starting configuration of the agent (the agent selects its action based on the goal it has. The agent must choose a sequence of actions to achieve the desired goal).

Action or operator: This takes the agent from one state to another state, which is called a successor state. A state contains a number of successor states. A plan is a series of actions. The cost of a plan is referred to as the path cost. The cost of a plan is a positive number, and a frequent path cost may be the sum of the cost of the steps in the path.

Now let us look at the idea of a search problem. Problem revolution means choosing an applicable set of states to consider, and a feasible set of operators for moving from one state to another.

Search is the process of allowing for several possible sequences of operators to be applied to the initial state and finding a sequence that culminates in a goal state.

3.1.1 The Search Problem

We are now ready to define a search problem. Any search problem consists of the following:

S : full set of states

S0 : initial state

A : S $\rightarrow$ s is a set of operators

G : set of final states

Search problem: This involves finding a sequence of actions that transforms the agent from the initial state to a goal state. A search problem is represented by a 4 tuple {S,SO,A,G}. This sequence of actions is called a solution path. It is a path from the initial state to a goal state S. Plan P is a sequence of actions.

$P = \{a_0, a_1, a_2, \ldots aN\}$

which leads to traversing a number of states $\{s0, s1, s2, \ldots sN + 1\}$. A sequence of states is called a path. The cost of path is a positive number. In many cases, the path cost is computed by taking the sum of the cost of each action.

The representation of a search problem occurs when a search problem is represented using a directed graph. States are represented as nodes. The allowed actions are represented as arcs.

Searching process: This is the basic searching process. It can be very basically described in terms of the following steps, which are repeated until a solution is found or the state space is exhausted.

- Check the current node.
- Execute an allowable action to find the successor states.
- Pick one of the new states.
- Check if the new state is a solution state.
- If it is not, the new state becomes the current state and the process is repeated.

Let's take a look at an illustration of a search process We will now demonstrate the searching process with the help of an example. Consider the problem shown in the figure.

- S0 is the initial state
- The successor states are the adjacent states in the graph.
- There are three goal states.
- Two successor states of the initial state are generated.
- The successors of these states are picked and their successors are generated.
- The successors of all these states are generated.
- The successors are generated.
- A goal state has been found.

The above example illustrates how we can start from a given state and follow the successor to find the solution paths that lead to a goal state. The grey node defines the search tree. Typically, the search tree is extended one node at a time. The search strategy is used to find the order of the search tree.

Pegs and Disk

We will now illustrate the state space search with one more example, the pegs and disk problem. We will illustrate a solution sequence, which, when applied to the initial state, takes us to a goal state. Consider the following problem. We have three pegs and three disks.

Operators: One may move the topmost disk on any needle to the topmost position to any other needle. In the goal state, all the pegs are in needle B, as shown in figure below.

The initial state is illustrated below.

Now, we will describe a sequence of actions that can be applied on the initial state.

Step 1 : move $A \rightarrow C$

Step 2 : move $A \rightarrow B$

Step 3 : move $A \rightarrow C$

Step 4 : move $B \rightarrow A$

Step 5 : move $C \rightarrow B$

Step 6 : move $A \rightarrow B$

Step 7 : move $C \rightarrow B$

8 Puzzle

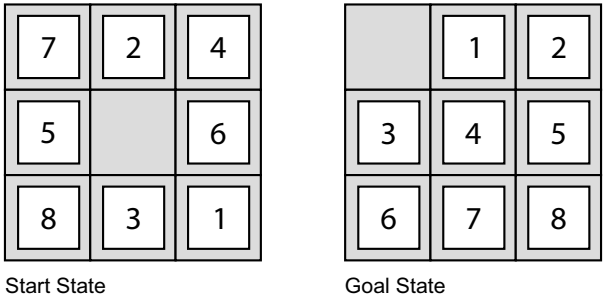


FIGURE 3.1 The Start and Goal States of the 8 Puzzle

In the 8 puzzle problem, we have a 3x3 square board and eight numbered tiles. The board has one blank position. The blocks can be slid to an adjacent blank position. We can alternatively and equivalently look upon this as the movement of the blank position up, down, left, or right. The objective of this puzzle is to move the tiles starting from the initial position and arriving at a given goal configuration.

The 15 puzzle problem is similar to the 8 puzzle problem. It has a 4x4 square board and 15 numbered tiles. The state space representation for this problem is summarized below:

State: A state is a description of each of eight tiles in each location that it can occupy.

Operators/actions: The blocks can be moved left, right, up, or down.

Goal test: The current state matches a certain state (for example, one of the moves shown on the previous slide). **Path cost:** This is each move of the blank cost.

An example of the state space of the 8 puzzle is shown. Note that we do not need to generate all the states before the search begins. The states can be generated when required.

3.2 Search Techniques

Every AI program has to go through the process of searching because the solution steps are not explicit in nature. The process of finding a solution for AI problems involves searching the path from the start state to the goal state. This is a very important aspect of problem solving because the search technique not only helps in finding most feasible path towards the goal state, but it also makes the entire process efficient and economical.

The search technique is an algorithm that takes problems as input and returns solution to the problem, usually after evaluating a number of possible solutions. The set of all possible solutions is called the search space.

Search: The searching mechanism through a state space involves the following:

- a set of states
- operators and their cost
- start state
- a test to check for the goal state

3.2.1 Basic Search Algorithm

Let *L* be a list containing the initial state

Loop

If *L* is empty, then return failure

Node ← select(L)

If node is a goal

Then return node

Else generate all successors of node and merge the newly generated state into *L*

End loop

The data structure for a node will keep track of not only the state, but also the present state or the operator that was applied to get this state. The search algorithm maintains a list of nodes.

Which path to choose?

The objective of a search problem is to find a path from the initial state to a goal state. If there are a number of paths, which path should be chosen? Our intention could be to find any path or we may need to find the shortest path.

What are the characteristics of different search algorithms and what is their efficiency? We will look at following three factors to measure this:

1. **Completeness:** This is the strategy that finds a solution if one exists.
2. **Optimality:** Does the solution have a low cost or the minimal cost?
3. What is the search cost related with the time and memory necessary to find a solution?

   a) **Time complexity:** time taken to find a solution

   b) **Space complexity:** space used by the algorithm.

3.3 Types of Searching Techniques

a) Uninformed or blind search

b) Informed or heuristic search

3.3.1 Uninformed Search (Blind Search)

The uninformed searches do not have any domain specific knowledge. All they need are the initial state, final state, and a set of legal operators. In this, they do not use any extra information about the problem domain (no extra information about the states).

Such a problem might relate to the problem space as a whole or to only some states. It may be accessible a priori or only after a node has been expanded.

In a worst case scenario, the only information available will be the ability to distinguish the goal from the non-goal nodes. When no further information is known, a priori, a search program must perform a blind or uninformed search. It proceeds in a systematic way, exploring nodes in some predetermined order or simply by selecting nodes at random.

Search programs may be required to return only a solution value when a goal is found or to record and return the solution path as well.

Uninformed searches are of several types:

a) breadth first search

b) depth first search

c) depth limited search

d) iterative deepening depth first search (DFID)

3.3.1.1 Breadth First Search

In the breadth first search (BFS), the searching process goes level by level. An operator is employed to generate all possible children of a node. In this, the root node is expanded first, then all the successors of the root node are expanded next. In the next step, all the successors of every node are expanded. This process continues until the goal state is achieved. BFS can be implemented by calling TREE-SEARCH with empty fringe that is a FIFO queue.

The node that is visited first will be expanded first. The FIFO queue puts all newly generated successors at the end of the queue

Complete: if shallowest goal node n is at some finite depth d

Optimal: if the path cost is a non-decreasing function of the depth of the node. Algorithm:

Step 1 : Set the initial node on a list START.

Step 2 : if START = empty or START = goal, terminate.

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

Step 4 : If *A* = goal, terminate the search with success

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

Step 6 : Go to Step 2.

Example:

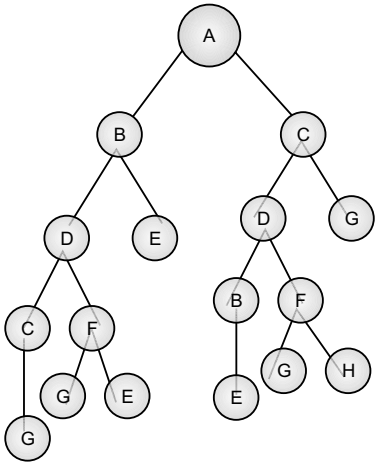


FIGURE 3.2 Breadth First Search

Step 1 : Initially, it contains only one node.

A

Step 2 : A is removed. The node is expanded, and its children B and C are generated.

B C

Step 3 : B is removed and is expanded; D and E are generated.

C D E

Step 4 : C is removed and expanded. Its children D and G are generated.

E D G

Step 5 : D is removed, and its children C and F are generated.

E D G C F

Step 6 : E is removed, and it has no children.

D G C F

Step 7 : D is expanded. B and F are added.

G C F B F

Step 8 : G is selected for growth. It is found to be a goal node. So, the algorithm returns the path A C G by following the pointer of the node corresponding to G.

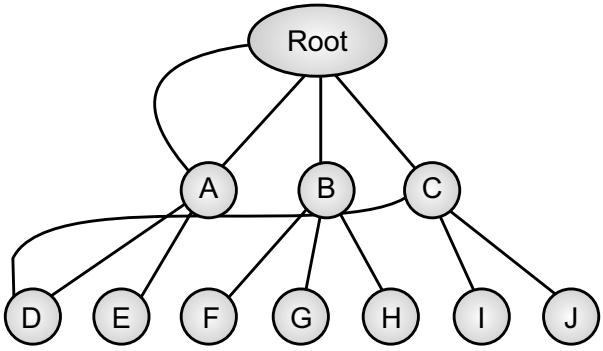


FIGURE 3.3 Order of Traversal in BFS

Properties: Complete Optimal

The algorithm has exponential time and space complexity:

$1 + b + b2 + b3 + ... + bd$

Time complexity = o(bd)

BFS has to remember every node it has generated. Since the procedure has to keep track of all the children it has generated, the space complexity is also function of d and b (the branching factor):

Space complexity = o(bd)

Memory requirements are a bigger problem with BFS than the execution time. Few computers have the terabyte of main memory it would take. However, there are other search strategies that require less memory. For example, if depth = 12, it would take 35 years for BFS to find it. When depth = 15, 0 to 14 have 10 children, and every node at depth 15 is a leaf node o(1015) node →: If BFS expands 10000 nodes per second and each node uses 1000 bytes of storage, BFS will take 3500 years to run, in the worst case, and it will use 11100 terabytes of memory. In this regard, the search space is quite small.

Advantages of BFS

- In the situation where there are multiple solutions, BFS finds the minimal solution (it requires a minimum number of steps).
- BFS is quite simple.
- It finds the path with the shortest length to the goal.

The travelling salesman problem can be solved by using BFS. With a small number of cities, it works well. If we have a large number of cities, it fails because the number of paths (and hence, the time taken to perform the search) becomes too big to be controlled by this method efficiently.

Disadvantage: BFS requires the generation and storage of a tree whose size is the exponent of the depth of the shallowest goal node.

Uniform cost search: This expands the node n with the lowest path cost. It does not care about the number of steps a path has, but only about the total cost.

The algorithm expands the nodes in the order of their cost from the source. The operator is associated with the cost. The path cost is usually taken to be sum of the step cost.

A newly generated node is put in the OPEN list according to the path costs. This ensures that when a node is selected for an expansion, it is the node with the cheapest cost.

Let $g(n)$ = the cost of a path from the start node to the current node. Sort the nodes by increasing the value of g.

- Complete
- Optimal

Exponential Time and Space Complexity: the Uninformed Cost Search

The BFS is optimal when all step costs are equal, as it expands the shallowest unexpanded node. BFS can be extended to mean "Instead of expanding the shallowest node, expand the node with lowest path cost." This extended BFS is called the uniform cost search.

If all step costs are equal, then the uniform cost search is identical to the BFS. The uniform cost search method expands the nodes in order of the increasing path cost. In fact, the tree search applies the goal test only to the nodes that are selected for expansion.

3.3.1.2 Depth First Search (DFS)

The depth first search is a very simple type of brute force search technique. The search begins by raising the initial node, i.e., by using an operator, and generating all the successors of the initial node and testing them.

This procedure finds whether the goal can be reached or not. However, the path it has to pursue has not been mentioned.

DFS expands the deepest node in the current fringe of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successor. As those nodes are expanded, they are dropped from the fringe.

This strategy can be implemented by TREE-SEARCH with LIFO, also known as STACK. It needs to store only a single path from the root to a leaf node, along with the remaining expanded sibling nodes for each node on the path. Once the node has been expanded, it can be removed from memory.

A variant of DFS called the backtracking search uses less memory. In backtracking, only one successor is generated at a time rather than all successors, and each moderately expanded node remembers which successor to generate next. In this, only o (m) memory is needed rather than o(bm). It is a memory-saving and time-saving technique.

The idea is to generate a successor by modifying the current state description directly rather than copying it first. This reduces the memory requirement to just one state description and o(m) action.

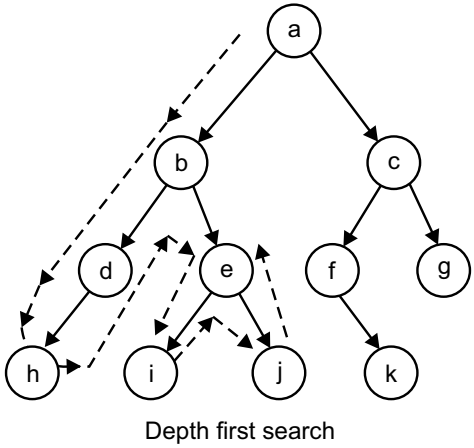


FIGURE 3.4 Depth First Search

Algorithm

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

Step 2 : If START = empty or START = goal, terminate search.

Step 3 : Remove the first node from START, call this node a.

Step 4 : If a= goal, terminate search with success.

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

Step 6 : Go to Step 2.

Example:

Step 1 : Initially the fringe contains only node A A

Step 2 : A is removed. A is expanded, and B and C are put in front of the fringe.

B C

Step 3 : B is removed, and D and E are pushed in front of the fringe.

D E C

Step 4 : D is removed, and C and F are pushed in front of the fringe.

C F E C

Step 5 : C is removed, and its child G is pushed in front of the fringe.

G F E C

Step 6 : G is expanded and found to be a goal node. Solution path A-B-D-C-G is returned.

Properties

The algorithm takes an exponential amount of time. If N is the maximum depth of a node in the search space, in the worst case, the algorithm will take o(bd). Space time is linear o(bN).

The time taken by an algorithm is related to the maximum depth of the search tree. If a search tree has infinite depth, the algorithm may not

terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycle. The latter case can be handled by checking for cycles in the algorithm. Then, DFS is not complete.

Advantages

- The DFS requires less space and less memory than the BFS since the nodes on the current path are stored.
- It may find the solution without examining much of the search space because we may get the desired solution in the very first try for a problem where only one solution is considered sufficient.

$$1 + b + b2 + b3 + bd$$

Time complexity = o(bd)

The DFS stores only the current path it is pursuing; the space complexity is a linear function of the depth o(d).

Disadvantages

The DFS is the determination of the depth unto which the search has to proceed. This depth is called the cut off depth. The DFS, unlike the BFS, may follow a single unfruitful path for a very long time. Theoretically, in the situation when there are no successors, it will stop searching. In the problem, this occurs when the production rule forms a loop.

For example, in the water jug problem, there are a number of production rules, and the problem can be solved by applying some rules in a particular sequence.

Suppose the DFS technique is applied to find the current path and solution of the problem. In the tree, it starts searching a branch having rules 1, 8, and 5. By applying these rules, a 4-liter jug will be filled, some water from a 4-liter jug will be put into a 3-liter jug, and a 4-liter jug will be emptied. The process is repeated, and we will never get a solution. The DFS may not find the optimal solution because as soon as a solution is found, it will stop the search. This solution may not be the optimal one. It may find the answer in a greater number of steps by unnecessarily exploring the wrong paths.

The BFS technique takes a lot of time. It is more suitable if a problem has more than one solution, and we have to find an optimal solution.

For the problem having a single solution, it may take unnecessary time in exploring the entire path in spite of getting the solution earlier.

A better approach requires the combination of the BFS and DFS. Some strategies have been used to accomplish this.

3.3.1.3 Depth Limited Search

The depth limited search is a combination of the BFS and DFS. The node at a certain level is considered as having no successor. Up to a certain depth, the tree is explored by the DFS method, and the rest of the tree is explored by the BFS. The DFS is a typical depth limited search method with a depth equal to infinity.

The depth limited search solves the infinite path problem. A variation of the DFS circumvents the above problem by keeping a depth bound. Nodes are only expanded if they have a depth less than the bound. This algorithm is known as the depth limited search.

Algorithm

Let the fringe(start) be a list containing the initial state

Loop

If the fringe is empty, then failure Node ← – – – – – remove-first(fringe) If node is a goal

Then return the path from the initial state to the node

Else if the depth of node = limit return cut off

Else add generated nodes to the front of the fringe

End loop

The nodes at depth l are treated as if they have no successor (l < d)

Time complexity = o(bl)

Space complexity = o(bl)

The DFS can be viewed as a depth limited search with l=infinity. Unfortunately, it also introduces an additional source of incompleteness. If we choose l<d, the shallowest goal is beyond the depth limit. The depth limited search will also non-optimal if we choose l>d.

l---- DFS with a predetermined depth limit l.

Sometimes the depth limit can be based on the knowledge of the problem. For example, there are 20 cities on the map of Romania. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so l=19 is a possible choice. But, in fact, if we have studied the map carefully, we would discover that any city can be reached from any other city in almost 9 steps. This number is known as the diameter of the state space, and it gives us a better depth limit, which leads to a more efficient depth limited search. For most problems, however, we will not know a good depth limit until we have solved the problem.

Iterative deepening DFS: first perform the DFS to depth 0 (treat the start node as having no successor), then if no solution is found, do the DFS to depth 1 (and so on).

3.3.1.4 Iterative Deepening Depth First Search (IDDFS)

Until solution found do

DFS with depth cutoff c

C = c+1

Iterative deepening (ID) is a general strategy often used in combination with the DFS that finds the best depth limit. It does this by gradually increasing the limit, first 0, then 1, and then 2, and so on, until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. ID combines the benefits of the DFS and BFS. Like the DFS, its memory requirements are very modest (o(bd), to be precise). Like the BFS, it is complete when the branching factor is finite and optimal when the path cost is a non-decreasing function of the depth of the node.

The ID search (IDS) may seem wasteful because the states are generated multiple times. It turns out this is not very costly. The reason is that in a search tree with the same branching factor at each level, most of the nodes are at the bottom level, so it does not matter much that the upper levels are generated multiple times. In the ID search, the nodes on the bottom level are generated twice and so on, up to the children of the root, which are generated d times.

The total number of nodes generated is

N(IDS) = (d)b+(d – 1)b2+_ _ _ _ _(bd)

Time complexity = o(bd)

N(BFS) = b + b2 + b3 + ... + bd + (bd + 1 – b)

The BFS generates some nodes at depth d + 1, but the IDS does not.

The IDS is actually faster than the BFS.

For example, if b=10, d=5

N(IDS) = 50 + 400 + 3000 + 20000 + 100,000 = 123450

N(BFS) = 10 + 100 + 1000 + 10,000 + 100,000 + 999,990 = 1,111,100

The IDS is analogous to the BFS in that it explores a complete layer of new nodes at each iteration before going on to the next layer.

ID searches are performed as a form of repetitive DFS. They begin by performing a DFS to a depth of one. It then discards all nodes generated and starts over, doing a search to a depth of 2. If no goal has been found, it discards all nodes generated and does a DFS to a depth of 3. This process continues until a goal node is found.

Advantages

- finds the shortest path
- has the linear memory requirements of the DFS
- guaranteed to find the goal node of the minimal path

Disadvantages: It performs wasted computations before reaching a goal depth.

Procedure: Successive DFSs are conducted, each with a depth bound increasing by 1.

Properties: For a large d, the ratio of the number of nodes expanded by the IDDFS compared to that of DFS is given by b/(b-1).

For BF = 10 and a deep goal, there is an 11% greater node expansion in ID searches than in the BFS. The algorithm is

- Complete
- **Optimal:** if all operators have the same cost

The time complexity is a little worse than that of the BFS and DFS because the nodes near the top of the search tree are guaranteed multiple times. In addition, almost all of the nodes are near the bottom of the tree. The worst case scenario for the time complexity is still the exponential o(bd) linear space complexity o(bd).

This algorithm is generally performed for a large state space where the solution depth is unknown.

BFS (completeness)

DFS (limited space and finds the longer path more quickly)

3.3.2 Avoiding Repeated States

In a search algorithm, there is always a possibility of wasting time by expanding states that have already been encountered and expanded before. For some problems, repeated states are unavoidable, i.e., where the actions are reversible, such as route finding problems and sliding block puzzles. The search trees for these problems are infinite.

By pruning some of the repeated states, the search tree can be cut down to a finite size. In an extreme case, a state space of size d+1 exists because of a tree with the second leaves.

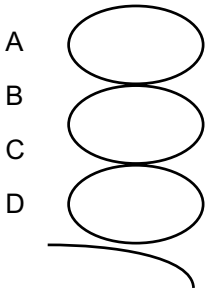


FIGURE 3.5 A State Space in Which There are Two Possible Actions Leading from A to B

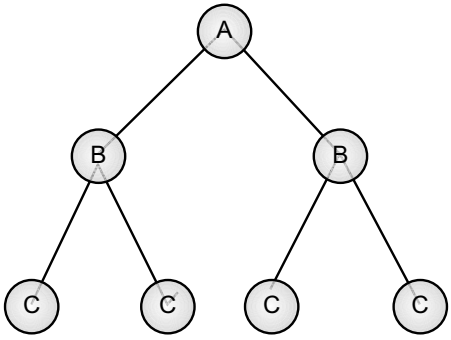


FIGURE 3.6 The Corresponding Search Tree

Algorithms that forget their history are doomed to repeat it.

- If an algorithm remembers every state that is has visited, then it can be viewed as exploring the state space graph directly.
- This can be done by including a data structure called a closed list, which stores every expanded node. The fringe of the unexpanded node is called the open list.
- Thus, if the current node matches a node on the closed list, it is disordered instead of being expanded.

A general graph search algorithm can be modified accordingly, as given below:

```
Function graph – search (problem, fringe)
Returns a solution or failure
Begin
Closed ← [ ]
Fringe ← insert (make node (initial_state [problem, fringe])
Loop do
        Begin
        If empty (fringe) then return failure
                Node ← remove first (fringe)
                If goal- TEST [problem] (state[node]) Then return
                solution (node)
                Is state[node] is not in closed then
                Add state[node] to closed
                Fringe ← insert_all (Expand(node, problem)fringe)
        End
End
```

Example:

Give the BFS, DFS, Depth Limited Search (limit=2), and Iterative Deepening Search results, and list the name of the nodes in order by the algorithm for the search of node C in the search space as given below.

Solution

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

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

DLS: A, B, D, E, C, F, G

IDDFS: It searches in the DLS manner until the goal is not reached.

Assuming the goal node is not specified in the given node tree, then the IDDFS will search in the following manner:

Limit = 0 A

Limit = 1 A, B, C

Limit = 2 A, B, D, E, C, F, G,

Limit = 3 A, B, D, H, I, E, J, K, C, F, L, M, G, N, O

Exercises

Q1. What is the search process? What is needed for a search method in AI?

Q2. Explain searching for solutions.

Q3. Explain the DFS and BFS in detail and differentiate between them.

Q4. Give the comparisons of the uninformed search strategies.

Q5. Show that the depth first technique is neither complete nor optimal.

Q6. What are different parameters that are used to evaluate search techniques?

CHAPTER 4

HEURISTIC SEARCH STRATEGIES

Heuristics are approximations used to minimize the searching process. Generally, there are two categories of problems:

- problems for which no exact algorithm is known and one needs to find an approximate and satisfying solution, e.g., computer vision and speech recognition
- problems for which an exact solution is known, but is computationally infeasible, e.g., chess. The heuristics needed for solving problems are generally represented as a heuristic function, which maps the problem state into numbers. These numbers are then approximately used to guide the search. Some simple heuristic functions are as follows:
    - ♦ In the famous 8 puzzle, the Hamming distance is a popular heuristic function. It is an indicator of the number of tiles in the positions they are in versus the goal positions.
    - ♦ In a game like chess, the material advantage one has over the opponent is an indicator.

A heuristic involves searching to find out. Heuristics are prescribed rules applied to discover those branches in a state space that are most likely to direct an acceptable and feasible solution of a given problem. To find the solution of the same problem, a heuristic search might reduce the time and effort needed to solve it.

Two applications of AI are game playing and theorem proving. These applications need heuristics to discover the state space for searching for the proper solution. When a human expert solves a problem, he or she definitely applies heuristics at some stage of the process. Experts normally use a rule of thumb, which is also heuristic in nature. These heuristics are also used in developing intelligent expert systems, which are extracted, coded, and represented by an expert system designer for a variety of real-world applications.

In a heuristic search, besides the normal production rules, additional information or knowledge about the problem is given in the form of a clue. All of these clues or extra information are also known as heuristics. The main aim of heuristics is to provide the goal to lead the search in a precise direction. Hence, it improves the quality of the path that is explored, reduces the search space, and efficiently obtains the solution of the problem.

Consider a situation in which one wants to search a particular house in a city. There is much information about the house, but the only available information that we know is the address of the house. To start the search, we need to look at the addresses of all houses until the required house is found. This search procedure is an example of a blind search. Now if the person has additional information or a clue about the house we are searching for (for example, it is a pink building), then finding the house will require checking only pink buildings. Now the person who sees all the houses and finds the house that matches the conditions will reject the houses that do not match the criteria. All additional information reduces the search space. Other clues that are given (such as "it is a pink, two-story building") will further reduce the search space. Heuristics are also used by humans, using the same criteria.

4.1 Types of Heuristic Search Techniques

- Generate and Test
- Best First Search
- Hill Climbing
- A*
- AO*

4.1.1 Generate and Test

The generate and test method is the simplest approach. As the name implies, this technique first generates a result and then tests whether it is the desired solution of the given problem.

Algorithm: Generate and Test

These steps are repeated until a satisfactory solution is obtained or no more solutions can be generated:

1. Generate a possible solution.
2. Test to see if this is a solution.
3. If a solution has been found, quit; otherwise, return to Step 1.

The generate and test approach can be compared with the DFS technique because we would have to generate a complete solution of the problem before it can be tested. In generate and test, two methods are used to complete its task.

The first method would be to generate a random solution and test it. If a solution is found, but it is wrong, create another solution and test it again. This method provides you with a solution that is a first trial; however, it means in this many chances, and there is a better chance that the solution is never found.

In second method, a systematic approach of generating a solution is applied, and we are sure of ultimately getting a right solution if it exists. The disadvantage of this method is that it might take lot of time if the problem space is large.

For straightforward problems having a limited problem space, the generate and test technique can be successfully deployed to be an exhaustive search; however, for problems with a bigger space, this method proves inefficient.

4.1.2 Best First Search

This search procedure is an evaluation function variant of the BFS. The heuristic function is used and is called an evaluation function; it is an indicator of how far the node is from the goal node. The goal node has an evaluation function value of zero. The evaluation function's value may be decided by cost or the distance of the current node from the goal node. The decision of which node should be expanded depends on the value of this evaluation function.

It is beneficial to search a graph instead of a tree to avoid the searching duplicate paths. In this process, searching is done in a direct graph, where each node represents a point in the problem space. This graph is known as an OR graph. Each of branch of an OR graph represents an alternative problem-solving path.

Two lists of nodes are used to implement a graph search procedure:

- **OPEN:** In open, there are only nodes that have been generated, but have not been examined yet. A heuristic function is also applied to these nodes.
- **CLOSED:** These are the nodes that have already been examined. If we want to search a graph rather than a tree, examined nodes always reside in memory because whenever a new node will be generated, we will have to check whether it has been generated earlier.

In this graph, S is the start node that is expanded first. It has three children, A, B, and C, with values 3, 6, and 5, respectively. These values just about specify how far they are from the goal node. Now we choose the node that has the minimum value, i.e., node A. Node A is expanded, and its children are generated, which are D and E with values 9 and 8, respectively. There are four nodes for searching, D, E, B, and C; D has a value of 9 and E has a value of 8, which are less than node B with a value of 6 and node C with a value of 5. In all of these nodes, node C has the minimal value, which is expanded to give node H with a value of 7. The available nodes for the search are (D : 9), (E : 8), (B : 6), and (H : 7), in which B is minimal. Hence, B is expanded: (F : 12) and (G :14).

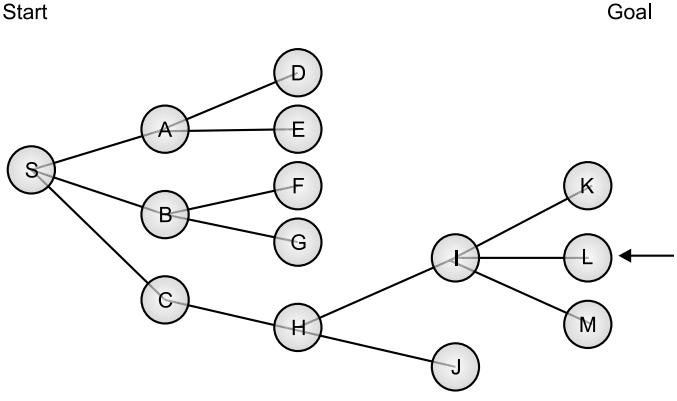


FIGURE 4.1 A Sample Tree for the Best First Search

At this juncture, the nodes available for the search are (D:9), (E:8), (H:7), (F:12), and (G:14), out of which (H:7) is minimal and is expanded to give (I:5) and (J:6). For searching, there are nodes now available for expansion: (D:9), (E:8), (F:12), (G:14), (I:5), and (J:6). We find that the node with the minimal value is (I: 5). Now this node is expanded to give the goal node.

Algorithm

Step 1 : Take an initial node and put this node on a list called START

Step 2 : Now check if START = empty or START = goal, then end (terminate) the search

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

Step 4 : If a = goal, terminate the search with success.

Step 5 : Else if the node has a successor, generate all of its children. Now from these nodes, find the goal node. When all the children are generated, then sort all the children so far by the remaining distance from the goal.

Step 6 : Name this list as START1.

Step 7 : Replace START with START1.

Step 8 : Go to Step 2.

The best first search combines the benefits of both the BFS and DFS. It uses the DFS because it allows a solution to be found without completing all the branches that have to be expanded. It uses the BFS because it does not get trapped on the dead ends of paths. Hence, at each step, we select the most promising node out of the successor nodes that have been generated so far.

Function of the Best First Search

- Here, a list is used, known as the open list, which contains just the initial node. The best first search maintains this list.
- Until a goal is found or there are no nodes left in the open list, do the following:
  - ♦ Pick the best node from the open list.
  - ♦ Many children are generated called successors, and for each successor

- Check all its successors, and if it has not been generated before, estimate (evaluate) it and add it to the open list and record its parent.
- If it has been generated before and the new path is better than the previous one, then change the parent.

A priority queue is used to implement the best first search.

4.1.3 Hill Climbing Search

The hill climbing search is another approach, and it is a discrete optimization algorithm. This search uses a simple heuristic function that uses the distance the node is from the goal node. The ordering of choices is a heuristic measure of the remaining distance one has to traverse to reach the goal node.

There is a category of problems where the path reporting is not important, and only reporting the final state is important. This kind of application includes circuit design, job shop scheduling, vehicle routing, and automatic programming. To solve this type of problem, the hill climbing technique is used. This type of technique works on the principle of the local search algorithm (i.e., the local search algorithm is used). It operates using a single current state, and it contains a loop that continuously moves in the direction of increasing the value of the objective function.

It is simply a loop that continuously moves in the direction of the increasing value; it is uphill, and it terminates when it reaches a peak where no neighbor has a higher value. Only the current node data structure needs to record the state and its objective function because this algorithm does not maintain a search tree. This algorithm does not look ahead further than the next neighbor of the current state.

The name of the hill climbing approach was derived from the simulation of the situation where a person is climbing a hill. The person makes every move towards the top of the hill. This person continues to move up until he reaches a peak at which the value is no higher than the other heuristic function, and that peak is known as the peak of the hill. Hill climbing is an alternate (variant) of the generate and test approach, in which feedback from a test procedure is used in deciding in which direction the search should proceed. At each point in the search path, a successor node that appears to reach the top of the hill most quickly

is selected for exploration. It does not keep a search tree, and the current node's data structure requires the recording of only the state and its objective function value. It does not look ahead further than its next neighbor. The hill climbing algorithm addresses the pure optimization problem, where the objective is to find the best state according to the objective function.

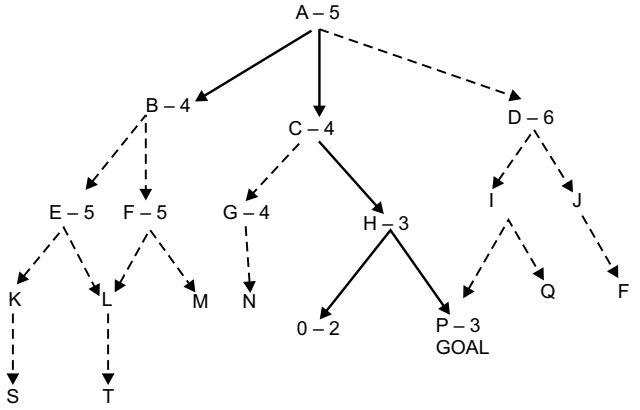


FIGURE 4.2 A Sample Tree for Hill Climbing

Algorithm

Step 1 : Put the initial node on a list START

Step 2 : If START=empty or START=goal, terminate

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

Step 4 : If a=goal, terminate the search with success.

Step 5 : Else if the node has a successor, generate all of them. Find out how far they are from the goal node. Sort them by the remaining distance from goal and add them to beginning of START.

Step 6 : Go to Step 2.

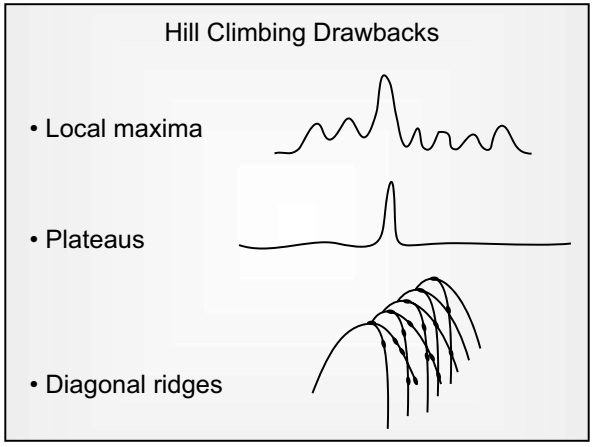
Limitations

1. **Local Maxima:** This is a peak that is higher than each of its neighboring states, but lower than the global maximum. In this situation, once point C is reached, the next calculation of the objective function is to

move downhill, so it will not be executed. Point C will be reported as the solution, which is actually not a global maximum. A distant peak with a greater height or higher value of the objective function is available. It is also a state that is better than all its neighbors, but not so when compared to states that are farther away.

2. **Ridge:** This is special kind of local maxima with a very steep slope that is difficult to trace in one calculation of the objective function (like D). This is also a narrow elevation or raised part running along a surface. This is an area in the path that must be traversed very carefully because movement in any direction might maintain one at the same level or result in a fast descent.

3. **Plateau:** This is a flat area of the search space in which all neighbors have the same value (like E) where the next move does not give a better solution than the present state. It becomes difficult to decide where to move. A hill climbing search might be unable to find its way off the plateau.



Solution to the Problem

1. Backtrack to some earlier node and try going in a different direction. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a good way of dealing with the local maxima.

2. Make a by jump in some direction to try to get a new selection of the search space. This is a good way of dealing with the plateau. It is the only rule available in single small steps, and you apply them several times in the same direction.

3. Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a good way of dealing with ridges.

4.1.4 Simulated Annealing Search

The problem of the local maxima is overcome in the simulated annealing search. In a normal hill climbing search, the movements may be downwards, and so the hill is never made. In such an algorithm, the search may get stuck at the local maxima. Thus, this search cannot guarantee a complete solution. In contrast, a random search towards a successor chosen randomly from the set of successors will be complete, but it will be extremely inefficient. The combination of hill climbing and random search, which yields both efficiency and completeness, is called simulated annealing.

Simulated annealing searches use the term "objective function" instead of heuristic function. If the move improves the situation, it is accepted. Otherwise, the algorithm accepts the move with some probability less than 1.

This probability is

$$P = e^{-E/Kt}$$

where

E = a positive change in the energy level

T = temperature

K = Boltzman constant

As indicated by the equation, the probability decreases with "badness" of the move. Annealing is a process used to harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low energy crystalline state.

Instead of picking the best move, this approach picks a random move. Simulated annealing was first used extensively to solve the VLSI layout problem in the early 1980s. It has been widely applied to factory scheduling and other large scale optimization tasks.

The process has the following differences from the hill climbing search:

- The annealing schedule is maintained.
- Moves to worse states are also accepted.
- In addition to the current state, the best state record is also maintained.

Algorithm

Step 1 : Evaluate the initial state. Mark it as the current state. Until the current state is not a goal state, initialize the best state to the current state. If the initial state is the best state, return it and quit.

Step 2 : Initialize T according to the annealing schedule.

Step 3 : Repeat the following until a solution is obtained or there are no operators left:

a) Apply the not-yet applied operators to produce a new state.

b) For a new state, compute E = value of the current state – the value of the new state. If the new state is the goal state, then stop, or if it is better than the current state, make it the current state and record it as the best state.

c) If *c* is not better than the current state, then make it the current state with a probability P.

d) Revise T according to the annealing schedule.

Step 4 : Return the best state as the answer.

4.1.5 A* Algorithm

The A* algorithm is a variation of the best first search. The most widely utilized form of the best first search is called the A* search. It evaluates the node by combining $g(n)$, the cost to reach the node $h(n)$, to get from the node to the goal.

$$F(n) = g(n) + h(n)$$

Here, $g(n)$ gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal.

$F(n)$ = estimated cost of cheapest solution through n.

If we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$.

This approach provides guidelines about how to estimate the goal distances for a general search graph. At each node along a path to the goal node, the A* algorithm generates all the successor nodes and computes an estimate of the distance (cost) from the start node to the goal node.

$G(n)$ = the cost of the current node from the start node

$H(n)$ = the cost of the current node from the goal node

Here, two functions are at work: the evaluation function and cost function. The cost function is how much of the resources, like time, energy, and money, have been spent in reaching a particular node from the start node. The evaluation function value deals with the future. The cost function value deals with the path.

The sum of the evaluation function value and cost along the path leading to that state is called the fitness number.

Now there are three numbers associated with each node: the evaluation function value, cost function value, and the fitness number. For example, consider node k, when the fitness number is 20.

(evaluation function of k) + (cost function involved s to k)

= 1 + (cost function from s to c^+ cost function from c to h + h to I + I to k)

= 1 + 6 + 5 + 7 + 1 = 20

Algorithm

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

Step 2 : If START is empty or START is the goal, terminate the search.

Step 3 : Remove the first node from START, and call this node a.

Step 4 : If a is the goal, terminate the search with success.

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

Step 6 : Name the new list as START1.

Step 7 : Replace START with START1.

Step 8 : Go to Step 2.

The A* algorithm maintains two lists. One stores the list of the open nodes, and the other maintains the list of the already expanded nodes. The A* search is both complete and optimal. A* is an example of an optimal search algorithm.

The optimality of A* is straightforward to analyze if it is used with the TREE-SEARCH. In this case, A* is optimal, if $h(n)$ is an admissible heuristic, that is provided that $h(n)$ never overestimates the cost to reach the goal. Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it is, actually n. Since $g(n)$ is the exact cost to reach n, we find, as an immediate consequence, that $f(n)$ never overestimates the true cost of a solution through n.

An example of an admissible heuristic is the straight line distance h_{SLD} that we used in getting to Bucharest. The straight line distance is admissible because the shortest path between any two points is a straight line, so a straight line cannot be an overestimation.

4.1.6 AND-OR Graphs

AND-OR graphs are a problem reduction technique. There are certain types of AI problems that can be decomposed into smaller problems. AND-OR graphs are useful for finding the solution to such problems. The problem is solved by breaking it into a set of smaller sub-problems, all of which must be solved in order to solve the complete problem. In the tree representation of the state space search, such sub-problems generated from a main problem create the "AND" arc. One arc may point to any number of successor nodes, all of which must be solved in order to get the complete solution. Here, the solution of any one sub-problem works as the solution of the complete problem. Representations of such problems are done using an OR graph.

The Function of an AND-OR Graph

AND-OR graphs are applicable for parse tree generation in English: type = "i">

(i) $S \leftarrow - - \rightarrow N_P V_P$ (ii) $N_P \leftarrow - - \rightarrow N$

(iii) $N_P \leftarrow - - \rightarrow \text{art } N$ (iv) $V_P \leftarrow - - \rightarrow V$

(*v*) $V_P \leftarrow - \rightarrow VN_P$ (*vi*) Art $\leftarrow - \rightarrow a$

(*vii*) Art $\leftarrow - \rightarrow$ then (*viii*) N $\leftarrow - \rightarrow$ man

(*ix*) $N \leftarrow - \rightarrow$ dog (*x*) $V \leftarrow - \rightarrow$ like

(*xi*) $V \leftarrow - \rightarrow$ bites

The method of finding a solution using AND-OR problem reduction is the AO* algorithm. It is simpler than the A* algorithm. As the AO* algorithm works for the AND-OR graph, it identifies all the sub-trees that must be solved, and it labels the traversed nodes in a tree as solved or unsolved. To account for the AND node arcs in the solution process, which requires the solution to all the successor nodes, the leveling of the nodes is required. The solution of the problem is found when the start node is labeled as solved.

Algorithm:

Step 1 : Place the start node *S* on an open list.

Step 2 : Using the search tree constructed so far, compute the most promising solution.

Step 3 : Select a node *n* and remove *n* from open and place it on closed.

Step 4 : If *n* is a terminal goal node, label *n* as solved if the solution of *n* results in any of *n*'s ancestors being solved. Label all the ancestors as solved if start node *S* is solved. Exit with success. Remove from open all nodes with a solved ancestor.

4.2 Properties of the Heuristic Search Algorithm

- **Admissibility conditions:** Any algorithm is called admissible if it is a guaranteed to return an optimal solution. When one exists, a solution is optimal if it generates the solution in the minimum number of steps. A* is an admissible algorithm. Admissible heuristics are optimistic because they assume the cost of solving a problem is less than the actual cost.
- **Completeness condition:** The A* algorithm is complete if it always terminates with a solution when one exists.

- **Dominance property:** This compares multiple algorithms created for the solution of the same problem. Let A*1 and A*2 be admissible algorithms with the heuristic estimation functions h1* and h2*. A*1 is said to dominate A*2 whenever h1*(n) > h2*(n) for all 1. A*1 is also said to be more informed than A*2. This indicates how good a heuristic function is.
- **Consistent heuristic:** A heuristic is said to be consistent if for every node *n* and its successor *nc* (generated by action a), the estimated cost of reaching the goal node from n is not greater than the step cost of getting to nc, plus the estimated cost of reaching the goal from nc.

$$H(n) \Leftarrow c(n, an) + h(n')$$

This is also known as monotonicity. It indicates that the search space is locally consistent everywhere with the heuristic employed.

A heuristic function h is monotone if it satisfies the following two properties:

- For all states n_i and n_j, where n_j is the descendent of n_i

$$H(n_i) - h(n_j) \Leftarrow \text{cost}(n_i, n_j)$$

Actual cost of going from n_i to n_j

- The heuristic evaluation of the goal state is zero, h(goal)=0.

Time and space complexities: The time complexity indicates how much time an algorithm takes to solve a problem. Space complexity indicates how much memory an algorithm takes to solve a problem.

They are dependent of the following factors:

- branching factor
- maximum length of any path in state space
- depth of the shallowest goal node.

4.3 Adversary Search

In this section, we will discuss a special type of search technique required in a game playing between two opponents. The state space in this case is represented by a tree or graph and includes the possible turns of both players. Each level of the search space in the present context denotes the possible turns of one player only. We start with a simple algorithm called MINIMAX.

4.3.1 The MINIMAX Algorithm

The MINIMAX algorithm considers the exhaustive possibility of the state transition from a given state and consequently covers the entire space. The algorithm thus is applicable to games having few possible state transitions from a given trial state. One typical example that can be simulated with MINIMAX is the NIM game. A NIM game is played between two players.

The game starts with an odd number of matchsticks, normally 7 or 9, placed on a single row, called a pile. Each player, in his turn, has to break a single pile into two piles of an unequal number of sticks greater than zero. The game will come to an end when either of the two players cannot make a successful move; the player who cannot make a successful first move will lose the game.

According to standard convention, we name the two players MINIMIZER and MAXIMIZER. NIM is a defensive game; consequently, the opening player here is called the MINIMIZER. For a game such as tic-tac-toe, where the opener always gets the benefit, the opening player is called the MAXIMIZER. A graph space for the NIM is represented in the figure below, demonstrating MAXIMIZER's move and MINIMIZER's move.

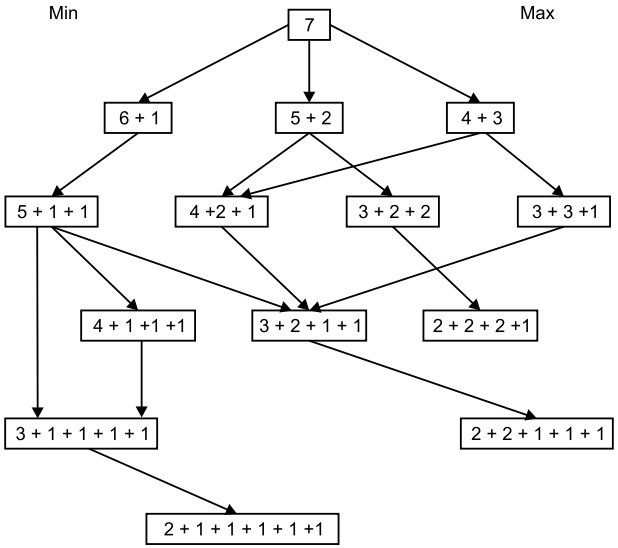


FIGURE 4.3 State Space for the NIM Game

In the MINIMAX algorithm, the following conventions will be used. The MAXIMIZER's success is denoted by +1, the MINIMIZER's success by -1, and a draw by 0. These values are attached to the moves of the players.

A question then normally arises: how do the players automatically learn about their success or failure until the game is over? This is realized in the MINIMAX algorithm by the following principle: assign a number from {+1,0,-1} at the leaves, depending on whether it is a success for the MAXIMIZER, MINIMIZER, or a draw. Now, propagate the values up by checking whether it is the MAXIMIZER's move or MINIMIZER's move. If it is the MAXIMIZER's move, then its value will be the maximum value possessed by its offspring. In case it is a MINIMIZER's move, then its value will presume the minimum of the values possessed by its offspring. If the values are propagated up to the root node by the above principle, then each player can select the better move in his turn. The computation process in a MINIMAX game is illustrated below:

Algorithm:

Begin

1. Expand the entire state space below the starting node.
2. Assign values to the terminals of the state space from {-1,0,+1}, depending on the success of the MINIMIZER, a draw, or the success of the MAXIMIZER.
3. For each node where all children possess values, do

Begin

If it is a MAXIMIZER node, then its value will be maximum of its children's value. If it is a MINIMIZER node, then its value will be the minimum of its children.

Table 4.1 Comparison of Search Techniques

Criterion	BFS	UCS	DFS	DLS	IDS	BS
Time	b^d	b^d	B^m	B^l	b^d	$b^{d/2}$
Space	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, If $l >= d$	Yes	Yes

Exercises

Q1. What are heuristics, and what is their importance?

Q2. Why is the heuristic search better than the blind search?

Q3. Distinguish between heuristics and algorithms.

Q4. Explain the best first search algorithm.

Q5. Describe the A* algorithm. Prove that the A* algorithm is complete and optimal.

Q6. Explain the AO* algorithm.

Q7. Describe the hill climbing algorithm.

Q8. What is meant by the local maxima with respect to search techniques?

Q9. When will the hill climbing search technique fail? Does the steepest ascent always find solutions? How can some problems be overcome in search?

Q10. How does the depth first search get converted into hill climbing?

CHAPTER 5

EXPERT SYSTEMS

The expert system is an application of AI that embodies human expertise. The field of expert systems addresses the solutions to complex problems. Artificial intelligence researchers achieved considerable success over the last 50 years in developing expert system technology to solve complex, real-time problems that are difficult to solve using traditional methods.

The first expert systems were created in the 1970s and then proliferated in the 1980s. Expert systems were among the first truly successful forms of AI software. Expert systems are the result of a novel approach of AI technology called the rule-based system. To design an expert system, one needs a knowledge engineer, an individual who studies how human experts make decisions and translates those rules into terms that a computer can understand. Expert systems are also known as knowledge-based systems, knowledge-based expert systems, and rule-based systems. They are considered to be "applied artificial intelligence." The process of developing with an expert system is called knowledge engineering. EMYCIN was one of the first "shells" for an expert system, which was created from the MYCIN medical diagnosis system. The production rule system is a rule engine that uses the rule-based approach to implement an expert system.

5.1 Definitions of Expert Systems

- An expert system is a computer system that simulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge like an expert.

- An expert system is a computer program that simulates the judgment and behavior of a human who has expert knowledge and experience in a particular field. Typically, such a system contains a knowledge base containing accumulated experience and a set of rules for applying the knowledge base to each particular situation that is described to the program. Sophisticated expert systems can be enhanced with additions to the knowledge base or to the set of rules.
- An expert system is a computer system that performs a task that would otherwise be performed by a human expert. For example, there are expert systems that can diagnose human illnesses, make financial forecasts, and schedule routes for delivery vehicles.
- Expert systems can also be defined as intelligent programs that have the ability to provide expertise in solving problems by using the domain-specific knowledge of a human expert.

5.2 Features of Good Expert Systems

All good expert systems

- should be useful: They should be developed to meet a specific need.
- should be usable: They should be designed so that even a novice computer user finds them easy to use.
- should be educational: An expert system may be used by non-experts who can then increase their own expertise by using it.
- should be able to explain the given advice: Expert systems should be able to explain the reasoning process.
- should be able to learn new knowledge: Expert systems should be able to ask questions to gain additional knowledge.
- should exhibit a high performance: Expert systems should provide high quality output, otherwise users will not be satisfied no matter how fast the output is produced.
- should make timely decisions: Expert systems must be able to produce decisions on time, otherwise, there is no point whether the output is right or wrong if timely output is not produced.

- should use heuristics: Expert systems must use heuristics to narrow the search area.

5.3 Architecture and Components of Expert Systems

The most common form of architecture of expert systems is the ruled-based system or production system. This type of system uses knowledge written in the form of production rules, that is, using "IF…THEN" rules. For example,

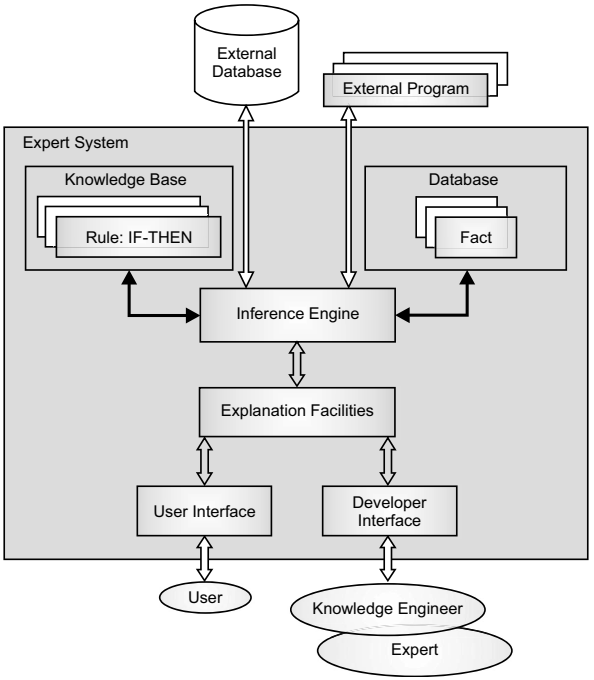


FIGURE 5.1 Expert System Components with Members of the Development Team

Syntax: If Condition THEN Action

Example: If it is cold THEN take umbrella

Each rule represents a small piece of knowledge relating to a given area of expertise. A set of related rules can lead from initially known facts to some useful conclusions. When the fact matches with left side of the rule, then the action is taken.

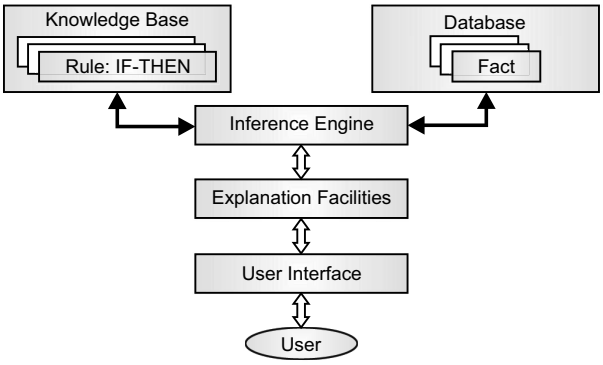


FIGURE 5.2 Components of Expert Systems

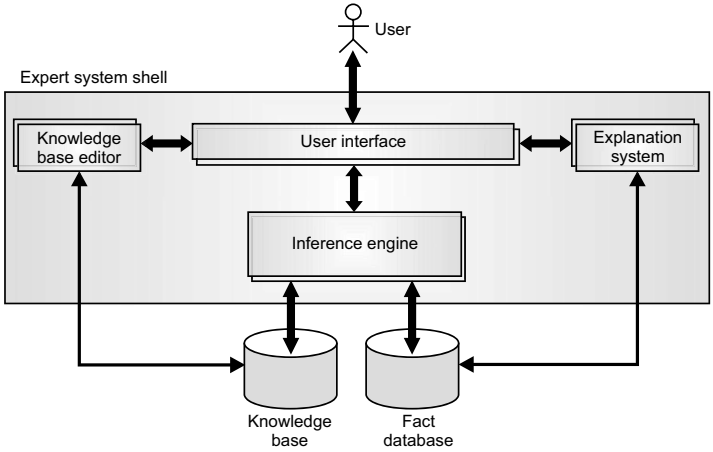


FIGURE 5.3 The Architecture of an Expert System

Various components of expert systems are given in the next sections.

5.3.1 User Interface

The acceptability of an expert system depends on the quality of the user interface. It provides for the communication exchange between the user and the system. The user interface can be a text-oriented interface or graphical interface, which is decided at the time when the expert system is designed. However, usually a graphical interface is chosen because it is more user friendly.

The user can enter commands and respond to questions. The system responds to commands, and asks questions during the inferencing process. Advanced interfaces make heavy use of various methods, such as pop-up

menus and windows. Through the user interface, the user can see into the system to determine what conclusion has been reached so far, why that conclusion was reached, what the system is doing now, and why it is doing it.

5.3.2 Knowledge Base

The knowledge base is the heart of an expert system. It contains all the knowledge of a domain expert obtained by the knowledge engineer through knowledge acquisition techniques. It contains expert-level knowledge on a particular subject stored in a knowledge representational form. It has been said that knowledge may be defined as factsor skills that are obtained through several years of experience.omain experts gather this knowledge, such as what is learned from school and from several years of experience. A domain expert is capable of expressing his knowledge in the form of rules for solving problems. The knowledge base contains rules and knowledge that are expressed in rule form.

An " IF ... THEN" rule has two parts:

a) the IF part, also called the antecedent or premise (condition part)

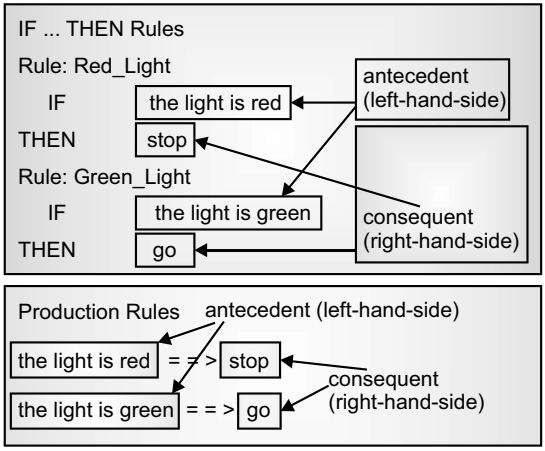
b) the THEN part, also called the consequent part or conclusion part (action part).

Structure of a rule:

IF <antecedent>

THEN <consequent>

For example:



Also, a rule can have multiple antecedent parts joined using connectives like AND or OR, and this is called a compound rule.

The structure of compound rule is as follows:

(a) IF <antecedent 1>
AND <antecedent 2>
•
•
•
AND <antecedent *n*>
THEN <consequent>

(b) IF <antecedent 1>
OR <antecedent 2>
•
•
•
OR <antecedent *n*>
THEN <consequent>

Expert systems can also use mathematical operators in place of AND and OR:

IF "age of the customer" < 18
AND "cash withdrawal" > 1000
THEN "signature of the parent" is required

Also, one important point to remember is that rules in the knowledge base can represent any relation, heuristic, suggestion, or recommendation.

- **Relation**

  IF the "fuel tank" is empty
  THEN the car is dead

- **Recommendation**

  IF the season is autumn

  AND the sky is cloudy

  AND the forecast is drizzle

  THEN the advice is "take an umbrella"

- **Directive**

  IF the car is dead

  AND the "fuel tank" is empty

  THEN the action is "refuel the car"

- **Heuristic**

  IF the spill is liquid

  AND the "spill pH" < 6

  AND the "spill smell" is vinegar

  THEN the "spill material" is "acetic acid"

Rules are the most common way of representing knowledge acquired from an expert. Rules also provide a description of how to solve a particular problem. The knowledge represented by the production rules (IF….THEN) is used for reasoning that is developed if the IF part of the rule is satisfied; consequently, the THEN part can be concluded, or its problem-solving action taken. Expert systems whose knowledge is represented in rule form are called rule-based systems. There are other ways also for representing knowledge, like frames and scripts, which we will discuss later.

A knowledge base also contains facts and questions, but a rule expresses the expertise. It is the warehouse of the domain specific knowledge captured from human experts through the knowledge acquisition module. The knowledge base is used by the inference engine component of an expert system for evaluating the rule and drawing conclusions from it.

The knowledge base of an expert system has various types of knowledge:

- **Procedural Knowledge:** The procedure refers to any task that is related to the performance of some task and a processed form of

information. For example, if we have step-by-step information for solving a problem, then it is called procedural knowledge.

- **Factual Knowledge:** This is the knowledge about the facts of a particular task domain that are found inside textbooks and journals. Such knowledge is shared widely.
- **Heuristic Knowledge:** This is the opposite of factual knowledge, as it is not widely shared but discussed rarely and is less rigorous, largely individualistic, and more experiential. Heuristic knowledge arises from good practice and good judgment.

Five processes are needed for building a knowledge base:

1. Knowledge acquisition
2. Knowledge analysis and representation
3. Knowledge validation
4. Inference design
5. Explanation and justification

These are not stages that have to follow each other — some of them will run concurrently, and this will be discussed later.

There are several advantages in representing knowledge through rules:

1. **Acquisition & Maintenance:** Since domain expert knowledge is encoded in the form of rules in the knowledge base, a domain expert can themselves define and maintain the rule.
2. **Explanation:** If knowledge can be represented in rule form, then it is also possible to explain to users about the conclusion reached. For example, consider a chain of inferences that led to a diagnosis. We can then use these facts to explain how such a diagnosis was reached.

Since the complexity of problems has increased, we need a complex knowledge base and sophisticated knowledge representation techniques, like the semantic net and frames.

5.3.3 Working Storage (Database)

The working storage contains facts that are used by the inference engine for matching facts with the antecedent part of a rule to find a conclusion. It contains the data that is specific to the problem being solved. It is a working store that the inference engine can use to hold data while it is working on problem. It holds all the data about the current task, such as

- user answers to questions
- any data from outside sources
- any intermediate result of reasoning
- any conclusions reached so far

There is difference between the knowledge base and the database. The knowledge base contains knowledge and is used over and over again, but a database contains data about a particular case.

5.3.4 Inference Engine

The inference engine is the reasoning component of an expert system, and it is a rule interpreter. The inference engine is a computer program that produces the reasoning for a rule. This engine is able to generate new information from the knowledge contained in the rule base and data to be processed. The brain of the rules system is an inference engine. The inference engine matches facts and data against the production rules — also called productions or just rules — to infer conclusions that result in actions. When the IF (condition) part of the rule matches with a fact, then the rule is said to be fired, and its THEN (action) part is executed.

This matching produces inference chains. The inference chain specifies how to apply the rules to reach a conclusion.

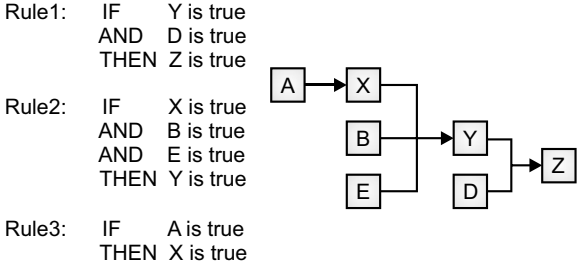


FIGURE 5.4 The Inference Chain

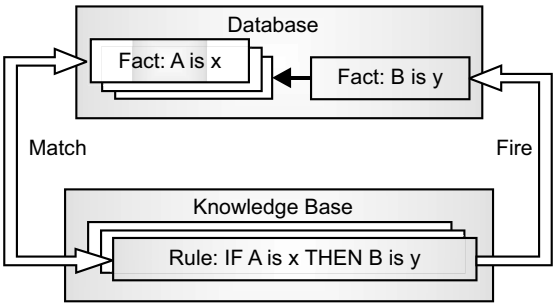


FIGURE 5.5 How the Inference Engine Works via the Match-Fire Procedure

The process of matching the new or existing facts against the production rules is called pattern matching, which is performed by the inference engine. It is the responsibility of the inference engine to prioritize the rules and fire rules having the highest priority. The rules are stored in the production memory and the facts that the inference engine matches against are kept in the working memory (database). Facts are declared into the working memory, where they may then be modified.

There are two methods of execution for a rule system: forward chaining and backward chaining. The systems that implement both are called hybrid chaining systems.

5.3.4.1 Forward Chaining

Forward chaining is "data-driven" reasoning, and we move from facts to conclusion. That is, we move forward with facts being asserted into the working memory, which results in one or more rules being concurrently true. In short, we start with a fact, it propagates, and we end in a conclusion. In forward chaining, only the topmost rules are evaluated, and every firing of a rule adds a new fact to the working memory. This process stops when no more rules are left for firing.

Forward-Chaining Algorithm

- while (no new assertion made) and (unresolved)
- for each rule

(And for each possible binding)

- try to support rule's conditions from known facts
- if all supported then assert consequent

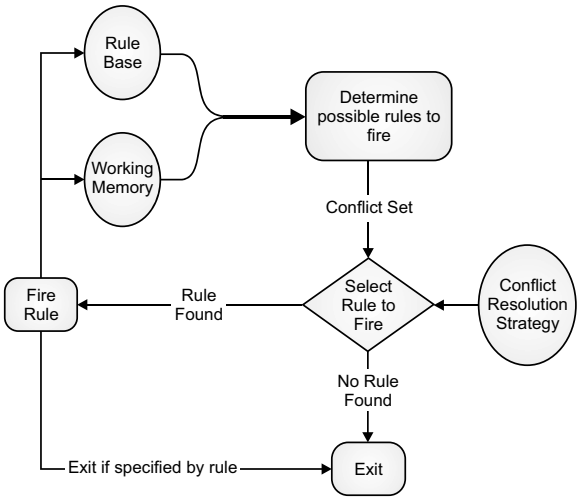


FIGURE 5.6 Forward Chaining

For example: Consider the below rule base of a simple vehicle recognizer.

R1: If ?x has wings

Then ?x is a plane

R2: If ?x flies

Then ?x is a plane

R3: If ?x runs on tracks

Then ?x is a train-or-tram

R4: If ?x is a plane

?x can take off vertically

?x has rotors

Then ?x is a helicopter

R5: If ?x is a train-or-tram

?x stays underground

Then ?x is a subway car

R6: If ?x is a helicopter

?x is made in South Africa

Then ?x is a Rooivalk

Now, from the above rule base, we move from fact to conclusion. That is, if one knows that a vehicle takes off vertically and has rotors, one can fire rule R4 to show that it is a helicopter.

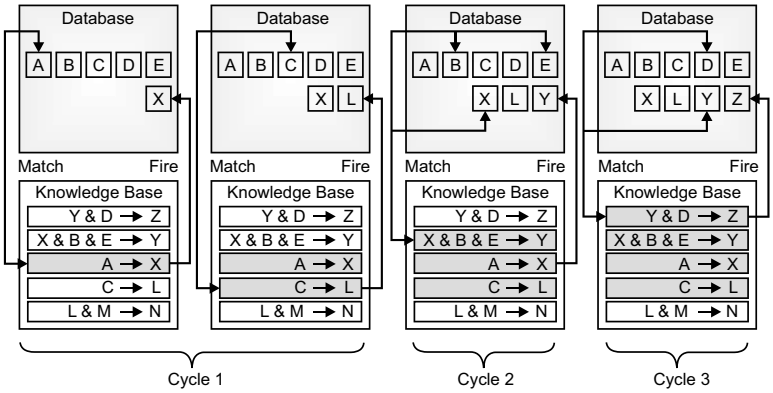


FIGURE 5.7 An Example of Forward Chaining

The problem with forward chaining is that some rules get executed even if they make no contribution to the goal. Thus, it is inefficient way of reasoning if we want to conclude only one fact.

5.3.4.2 Backward Chaining

Backward chaining is "goal-driven," that is, we start with a conclusion that the engine then tries to satisfy. Expert systems have a goal, and the inference engine in backward chaining tries to prove that goal. First, the knowledge base is searched for a rule having that goal. If such a rule is found, and it's IF (condition) part also matches with the data in the database, then that the rule is fired and the goal is proved. If it can't find such rule, then it searches for conclusions that it can satisfy; these are known as sub-goals that will help satisfy some unknown part of the current goal. Now, a new rule is searched in the knowledge base for proving that sub-goal. This process continues until either the initial conclusion is proven or there are no more sub-goals. Prolog is an example of a backward chaining engine.

Backward-Chaining Algorithm

- while (no untried hypothesis) and (unresolved)
- for each hypothesis
  - ♦ for each rule with the hypothesis as the consequent
  - ♦ try to support the rule's conditions from known facts or via recursion (trying all possible buildings)
  - ♦ if all are supported, then assert the consequent

For example, one may try to show that the vehicle is a Rooivalk by using rule R6. The one fact—South African—is known, but it is not known whether the vehicle is a helicopter. That may be done by rule R4. Two of the three facts there are known, but it is not known that the vehicle is a plane. We can then try rules R1 and R2. That is, we move backward.

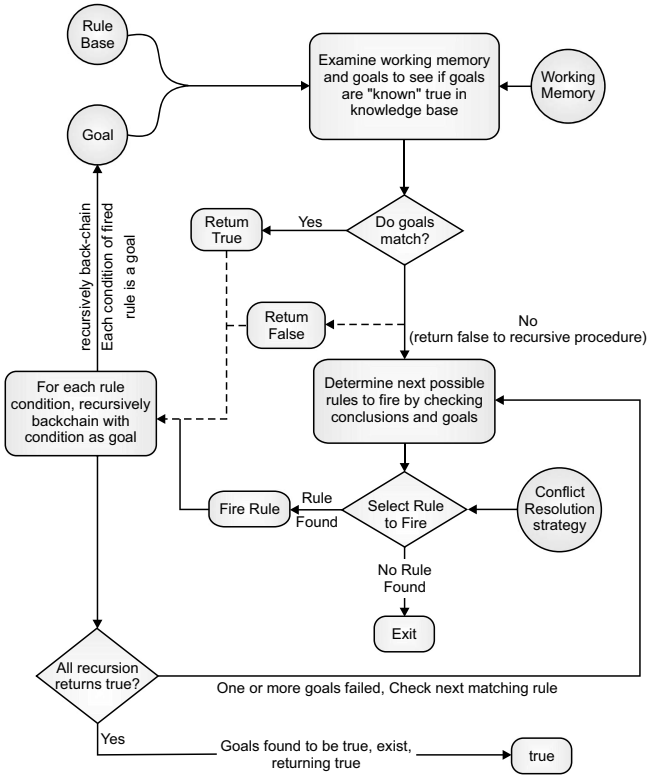


FIGURE 5.8 The Backward Chaining Process

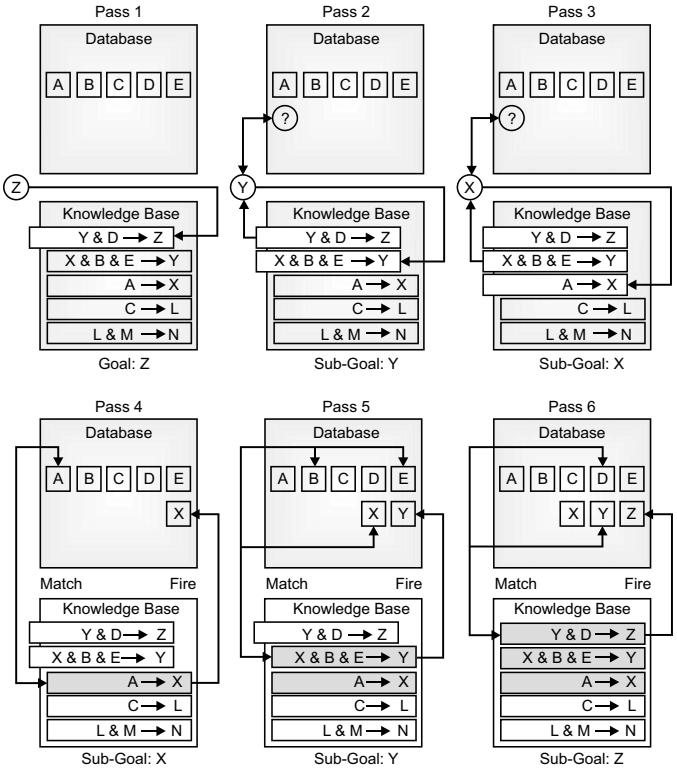


FIGURE 5.9 An Example of Backward Chaining

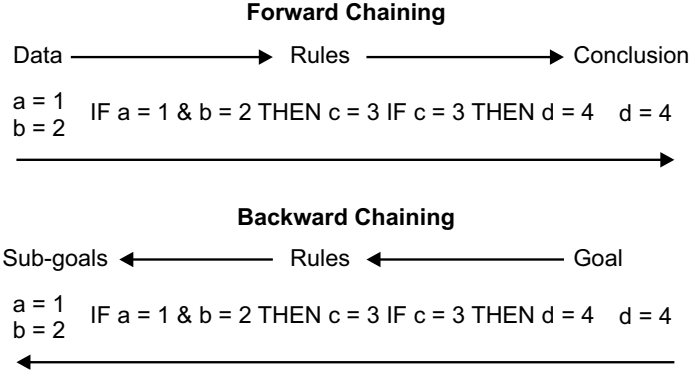


FIGURE 5.10 Difference Between Forward and Backward Chaining

The forward chaining system starts with the data of $a = 1$ and $b = 2$ and uses the rules to derive $d = 4$. The backward chaining system starts with the goal of finding a value for d and uses the two rules to reduce that to the problem of finding values for a and b.

Conflict resolution by the inference engine: Conflict occurs when two or more rules are satisfied by the same fact. A system with a large number of rules and facts may result in many rules being true for the same fact. These rules are said to be in conflict, and the inference engine has to use a conflict resolution strategy for managing the execution order of these conflicting rules.

Consider the following three rules in a knowledge base:

1. IF the "traffic light" is green
   THEN the action is go

2. IF the "traffic light" is red
   THEN the action is stop

3. IF the "traffic light" is red
   THEN the action is go

Now, it is clear the rules 2 and 3 have same conditional part, so if the conditional part is satisfied by a fact from the database, then both rules will be fired. This is called a conflict, and the set of conflict rules is called the conflict set. It is the responsibility of the inference engine to determine which rule to fire from the conflict set. The inference engine uses a conflict resolution strategy.

In the case of forward chaining, both rules would be fired. Since the rule of forward chaining dictates that the topmost rule will be fired, rule 2 is fired first. The action takes the value "stop." But rule 3 can also get fired after rule 2, because its condition also matches, so now the action takes on the new value "go."

There are two conflict resolution strategies used by the inference engine for conflict resolution:

- The first strategy is to assign priorities to each rule and store the rules in the knowledge base in order of their priority. In the case of conflict, the high priority rule is fired first.
- The second strategy is to fire the rule that possesses more information, that is, the most specific rule is fired.

5.3.5 Explanation Facility

It is important to explain the reasoning of the system to a user. It is possible for the system to provide those rules which were used during the inference process to the user as a means for explaining the results. This type of explanation can be very dramatic for some systems, such as the bird identification system.

At other times, however, the explanations are relatively useless to the user. This is because the rules of an expert system typically represent empirical knowledge, and not a deep understanding of the problem domain. For example, a car diagnostic system has rules that relate symptoms to problems, but no rules which describe why those symptoms are related to those problems.

Explanations are always of extreme value to the knowledge engineer. By looking at the explanations, the knowledge engineer can see how the system is behaving, and how the rules and data are interacting.

5.3.6 Knowledge Acquisition Facility

This refers to an automatic way for the expert to enter knowledge in the system rather than by having the knowledge engineer explicitly code the knowledge.

5.3.7 External Interface

The user interface handles all data going to and from users, but sometimes the expert system exchanges data with other sources, such as data files. The inference engine calls the external interface to get the input it needs and to transmit output to the proper destination.

5.4 Roles of the Individuals Who Interact with the System

There are five members on the development team of an expert system: the knowledge engineer, domain expert, end user, programmer, and project manager. The success of their expert system entirely depends on how well the members work together.

5.4.1 Domain Expert

A domain expert is an individual (or individuals) who is an expert at solving the problems that the system is intended to solve. Domain experts have a deep knowledge of facts and also have strong experience in a particular

domain. A domain expert is a skilled person with knowledge in a particular domain for solving problems in that domain. The expert knowledge of a domain expert is captured through knowledge acquisition techniques. The domain expert must be willing to participate in the knowledge acquisition process.

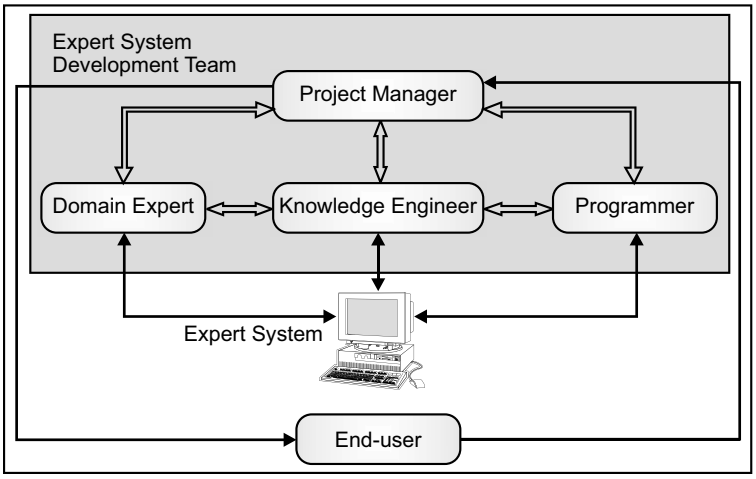


FIGURE 5.11 The Expert System Development Team

5.4.2 Knowledge Engineer

It is the knowledge engineer who obtains knowledge from various sources. The knowledge engineer's responsibility is to design, build, and test an expert system. It is the knowledge engineer who interacts with the expert and knowledge base.

The knowledge engineer, after acquiring knowledge, decides the representational scheme for that knowledge and also decides on the programming language for knowledge encoding. The knowledge engineer's role is not restricted to knowledge acquisition and representation. He is also responsible for testing and revising the expert system.

5.4.3 Programmer

The programmer is responsible for using the programming language chosen by knowledge engineer for encoding the knowledge. The AI programming languages are LISP and PROLOG, so the programmer must have strong skills in these languages, in addition with other languages like C, C++, and Java.

5.4.4 Project Manager

The project manager is responsible for managing the overall expert system's development. He keeps the project on track and coordinates with the rest of the team members.

5.4.5 User

The end user is the person who finally uses the developed expert system. He is the person who will be consulting with the system to get advice which would have been provided by the expert. The user must feel that the system is user friendly.

Many expert systems are built with products called expert system shells. The shell is a piece of software that contains the user interface, a format for the declarative knowledge in the knowledge base, and an inference engine. The knowledge engineer uses the shell to build a system for a particular problem domain.

Expert systems are also built with shells that are custom developed for particular applications. In this case, there is another key individual. The system engineer builds the user interface, designs the declarative format of the knowledge base, and implements the inference engine. Depending on the size of the project, the knowledge engineer and the system engineer might be the same person.

A major problem in building expert systems is the knowledge engineering process (the entire process of building an expert system is called knowledge engineering). The coding of the expertise into the declarative rule format can be a difficult and tedious task. One major advantage of a customized shell is that the format of the knowledge base can be designed to facilitate the knowledge engineering process.

Thus, an expert system shell is a special purpose tool that is designed using the requirements of its application. Users then supply the knowledge base to the shell. The shell accomplishes the input and output. It then processes the information which is given by the user, evaluates it according to the knowledge contained in the knowledge base, and then finally provides a solution for a particular problem.

5.5 Advantages of Expert Systems

- **Quick availability and opportunity to program itself**

  As the rule base is in everyday language, an expert system can be written much faster than a conventional program, by users or experts, bypassing professional developers and avoiding the need to explain the subject.

- **Ability to exploit a considerable amount of knowledge**

  The expert system uses a rule base, unlike conventional programs, which means that the volume of knowledge to program is not a major concern. Whether the rule base has 10 rules or 10,000, the engine operation is the same.

- **Reliability and consistency**

  The reliability of an expert system is the same as the reliability of a database. It also depends on the size of the knowledge base. If the knowledge base is set up with no ambiguity or subjectivity, then the expert system (with the same input criteria) will always deliver the same output. This is useful for expert systems used to make decisions that need to have no bias, such as a loan decision expert system. As such, the loan expert system will evaluate two different people with the same financial history in the same manner, ensuring equal opportunity.

- **Scalability**

  Evolving an expert system means to add, modify, or delete rules. Since the rules are written in plain language, it is easy to identify those that should be removed or modified.

- **Pedagogy**

  The engines that are run by true logic are able to explain to the user in plain language why they ask a question and how they arrived at each deduction. In doing so, they show the knowledge of the expert contained in the expert system. So, the user can learn this knowledge in its context. Moreover, they can communicate their deductions step by step. The user has information about their problem even before the final answer is given by the expert system.

- **Preservation and improvement of knowledge**

  Valuable knowledge can disappear with the death, resignation, or retirement of an expert. Recorded in an expert system, it becomes eternal. To develop an expert system involves interviewing an expert and making the system aware of their knowledge. In doing so, it reflects and enhances the expert knowledge.

- **Robust and effective**

  An expert system is robust so it can operate with incomplete knowledge, and it is effective because it can operate with reasonable performance in a complex domain.

- **Transparent**

  An expert system is transparent, meaning it can explain or justify its reasoning naturally. Knowledge can be represented in a declarative way without affecting its use.

5.6 Disadvantages of Expert Systems

- Every expert system has a major flaw, which explains their low success rate despite the fact that the principles such systems are based have existed for 70 years, such as knowledge collection and its interpretation into rules, or knowledge engineering. Most developers have no automated method to perform this task; instead they work manually, increasing the likelihood of errors. Expert knowledge is generally not well understood; for example, rules may not exist, be contradictory, or be poorly written and unusable. Most expert systems use a computational engine incapable of reasoning. As a result, an expert system will often work poorly, and the project will be abandoned. Correct development methodology can solve these problems.
- The disadvantages of using expert systems include that they usually only cover a narrow spectrum. They are also expensive. Many expert systems are menu-driven. This means they don't handle ambiguity well.
- Expert systems lack the common sense needed to make some decisions.

- They cannot respond creatively, like a human expert would in unusual circumstances.
- Domain experts are not always able to explain their logic and reasoning.
- Errors may occur in the knowledge base and lead to wrong decisions.
- They cannot adapt to changing environments unless the knowledge base is changed.
- Expert systems can't draw analogies from other sources to solve a newly encountered problems like a human would; in other words, they can't be creative.
- Human experts automatically adapt to changing environments; expert systems must be explicitly updated.
- Human experts have available to them a wide range of sensory experiences; expert systems are currently dependent on symbolic input.
- Although inexpensive to operate, expert systems are expensive to develop and maintain.

Table 5.1 A Comparison of Expert Systems with Conventional Systems and Human Experts

Human Experts	Expert Systems	Conventional Programs
Use knowledge in the form of rules of thumb or heuristics to solve problems in a narrow domain.	Process knowledge expressed in the form of rules and use symbolic reasoning to solve problems in a narrow domain.	Process data and use algorithms, a series of well-defined operations, to solve general numerical problems.
In the human brain, knowledge exists in a compiled form.	Provide a clear separation of knowledge from its processing.	Do not separate knowledge from the control structure to process this knowledge.
Capable of explaining a line of reasoning and providing the details.	Trace the rules fired during a problem-solving session and explain how a particular conclusion was reached and why specific data was needed.	Do not explain how a particular result was obtained and why input data was needed.

Use inexact reasoning and can deal with incomplete, uncertain and fuzzy information.	Permit inexact reasoning and can deal with incomplete, uncertain and fuzzy data.	Work only on problems where data is complete and exact.
Can make mistakes when information is incomplete or fuzzy.	Can make mistakes when data is incomplete or fuzzy.	Provide no solution at all, or a wrong one, when data is incomplete or fuzzy.
Enhance the quality of problem solving via years of learning and practical training. This process is slow, inefficient, and expensive.	Enhance the quality of problem solving by adding new rules or adjusting old ones in the knowledge base. When new knowledge is acquired, changes are easy to accomplish.	Enhance the quality of problem solving by changing the program code, which affects both the knowledge and its processing, making changes difficult.

Table 5.2 A Comparison of Conventional Programs and Expert Systems

Characteristics	Conventional Program	Expert System
Control by ...	Statement order	Inference engine
Control and Data	Implicit integration	Explicit separation
Control Strength	Strong	Weak
Solution by ...	Algorithm	Rules and Inference
Solution search	Small or none	Large
Problem solving	Algorithm	Rules
Input	Assumed correct	Incomplete, incorrect
Unexpected input	Difficult to deal with	Very responsive
Output	Always correct	Varies with the problem
Explanation	None	Usually
Applications	Numeric, file and text	Symbolic reasoning
Execution	Generally sequential	Opportunistic rules

Exercises

Q1. What is an expert system?

Q2. What are the advantages of expert systems over human experts?

Q3. Define an inference engine used as part of an expert system.

Q4. Describe expertise and its limits.

Q5. What rules do expert systems use?

Q6. What is a knowledge base?

Q7. Describe the difference between forward and backward chaining.

Q8. Explain the role of the various individuals who interact with an expert system.

CHAPTER 6

THE EXPERT SYSTEM DEVELOPMENT LIFE CYCLE

Expert systems are an application of AI that embodies human expertise. In recent years, expert systems have emerged as one of the most powerful tools from the field of AI. Expert systems are designed for solving complex decision–making problems that simulate a human expert's capability for making decisions. Rosenman defined expert systems as "An automated reasoning system that attempts to mimic the performance of the human expert."

Expert systems can also be defined as intelligent programs that have the ability to provide expertise in solving problems by using the domain-specific knowledge of a human expert. The architecture of expert systems was discussed in the previous chapter. As noted before, the following are the components of expert systems:

- user interface
- knowledge base
- database
- inference engine
- explanation facility
- knowledge acquisition facility

expert system = knowledge base + inference engine

There are many examples of expert systems. MYCIN was used for medical diagnosis (to identify a specific group of diseases). DENDRAL helped to identify a molecule's structure given its chemical formula and other data.

For conventional software, there is the software development life cycle (SDLC). Similarly, for there is an expert system development life cycle for developing expert systems. Most of the expert systems can be developed by using conventional software engineering techniques, but with some modifications in these techniques, since the conventional software development life cycle is somewhat inadequate in satisfying expert system requirements. Conventional techniques of software development are also based on the assumption that the requirements are well defined.

For expert system development, the evolutionary and prototype approaches must be used, where the system is developed in multiple stages through the continuous and proper interaction between the knowledge engineer and expert. The prototype approach is best suited for decision-oriented applications. The expert system development life cycle involves much more prototyping than conventional software development methods.

The expert system development life cycle goes through a number of stages, starting from the problem definition and ending with the implementation and operation. In between these two stages are all the activities that must follow an iterative cycle until the system's desired stability is achieved.

6.1 Stages in the Expert System Development Life Cycle

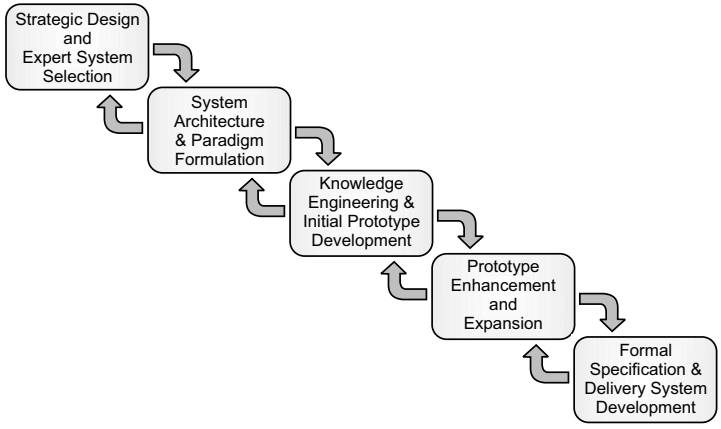


FIGURE 6.1 Stages in the Expert System Development Life Cycle (ESDLC)

The following are the stages in the expert system development life cycle.

6.1.1 Problem Selection

The first step is to identify the appropriate problem domain, which is to define the problem that the client expects the expert system to solve. In this stage, various features regarding the suitability of the expert system are assessed. The various activities in the problem definition stage are as follows:

- Determine the feasibility of the expert system.
- Determine whether an expert system is the right approach for solving that problem.
- Properly identify and define the problem domain that the client wants to solve.
- Identify domain experts (sources of knowledge) and the users of the expert system.
- The domain expert must have enough knowledge of the subject matter area and must have enough time for project completion.
- Identify the appropriate approach to the problem and discuss it with the domain experts and users to obtain a proper understanding of the purpose of the expert system.
- Interview experts and users and focus on their skills with respect to the problem domain.
- Prepare a proper plan of the expert system that includes the timing schedule and resource requirements.
- Since the prototype approach is used for expert system development, a sub-set of the whole problem is selected for which the initial prototype will be made for determining feasibility.
- Analyze various methods of knowledge representation to choose an appropriate tool for prototype development.
- Perform the cost-benefit analysis.

The outcome of this phase is the initial requirement review report (IRR). The IRR report must include following things:

- a description of problem, that is, what should be done by the expert system
- the specifications of the list of skills that users must have in order to work with the expert system
- a listing of the currently established requirements
- a tentative schedule for the initial prototype development
- time constraints

Table 6.1 Activities in the Problem Definition Stage

Task	Objective
Feasibility assessment	Determine if it is worthwhile to build the system and if so, whether the expert systems' technology should be used.
Resource management	Assess resources such as people, time, money, software, and hardware. Acquire and manage the required resources.
Task phasing	Specify the tasks and their order in the stages.
Schedules	Specify the starting and delivery dates of tasks in the stages.
Preliminary functional layout	Define what the system should accomplish by specifying the high-level functions of the system. This task specifies the purpose of the system.
High-level requirements	Describe in high-level terms how the functions of the system will be accomplished.

6.1.2 Conceptualization

In this stage, the characterization of the situation and design of the proposed program is done. That is, the intended system capability is described, and the expertise needed for solving the proposed problem is also determined. A discussion between the knowledge engineer and expert helps in clarifying the scope of the system and its details, and helps

in identifying sub-problems of the problem that can be implemented quickly in the form of a prototype.

In this stage, knowledge acquisition is done, that is, acquiring knowledge from the domain expert. Two tasks are done:

- knowledge source identification and selection
- knowledge acquisition, analysis, and extraction

Acquiring knowledge is the responsibility of the knowledge engineer. The knowledge engineer has to identify various knowledge sources and select the appropriate one. A knowledge engineer interviews the domain expert many times and asks a lot of questions.

The following questions may be used by the knowledge engineer to help understand what the expert does:

- Exactly what decisions does the expert make?
- What are the outcomes of the decisions?
- Which outcomes require greater exploration or interaction?
- What resources or inputs are required to reach a decision?
- What conditions are present when a particular outcome is decided?

Table 6.2 Knowledge Source Identification

Task	Objective
Source identification	Who and what are the knowledge sources, without regard to availability?
Source importance	Prioritized list of knowledge sources in order of importance to development.
Source availability	List of knowledge sources ranked in order of availability. The web, books, and other documents are generally much more available than human experts.
Source selection	Select the knowledge sources based on importance and availability.

After selecting the appropriate knowledge source, knowledge acquisition is done by knowledge acquisition methods.

6.1.3 Formalization

In this stage, the program logic is designed. Formalization means organizing the knowledge obtained from the previous stages and organizing the key concepts and sub-problems. Acquired knowledge is organized and classified into a hierarchical tree-like structure. Organizing knowledge using knowledge representing schemes as knowledge representation is important for a system's trustworthiness.

- Construct a knowledge base (with rules) by obtaining knowledge from domain experts.
- Identify and define the proper user interface between the expert system and other external sources.

Table 6.3 Knowledge Definition Tasks

Task	Objective
Knowledge representation	Specify how knowledge will be represented, such as the rules, frames, or logic. Dependent upon what the expert systems tool will support.
Detailed control structure	Specify three general control structures: 1. If the system is embedded in the procedural code, how it will be called; 2. Control of the related groups of rules within an executing system; 3. Meta-level control structures for rules.
Internal fact structure	Specify the internal structure of facts in a consistent manner to aid in understanding and good style.
Preliminary user interface	Specify a preliminary user interface. Get feedback from users about the interface.
Initial test plan	Specify how the code will be tested. Define the test data, test drivers, and how the test results will be analyzed.

Table 6.4 Detailed Design of the Knowledge Tasks

Task	Objective
Design structure	Specify how knowledge is logically organized in the knowledge base and what is in the knowledge base.
Implementation strategy	Specify how the system is to be implemented.

Detailed user interface	Specify the detailed user interface after receiving user feedback from the preliminary user interface design.
Design specifications and report	Document the design.
Detailed test plan	Specify exactly how the code will be tested and verified.

Thus, knowledge refinement in the hierarchy and its relationship are important in the formalization stage. The knowledge engineer also specifies inference rules and control strategies.

6.1.4 Prototype Construction

IEEE defines prototyping as "A type of development in which emphasis is placed on developing prototypes early in the development process to permit early feedback and analysis in support of the development process."

A prototype is just a working model or an early model that is built for demonstrating the proposed system's feasibility. The prototype just contains a sub-set of the functionality of the entire system and is built so that the domain expert can obtain feedback from its ideas. The basic idea behind the prototype approach is the development of a working model to understand requirements rather than freezing requirements before design or coding can proceed.

The rapid prototyping method is used for the development of an expert system because it's always better to find a proposed system's feasibility (economic, technical, and operational) by building a sub-problem prototype instead of rejecting the whole system after investing lot of money, time, and resources. It is cost-effective to make changes early in the expert system's development life cycle. Thus, prototyping is a risk reduction activity. Prototyping involves the iterative creation of a proposed system. The main focus of the prototyping approach is the quick implementation of ideas so that immediate feedback can be obtained either for modifying that prototype or moving forward. One more advantage of developing a prototype is that it helps the user get a better understanding of how the system will work because the prototype will help build the user interface.

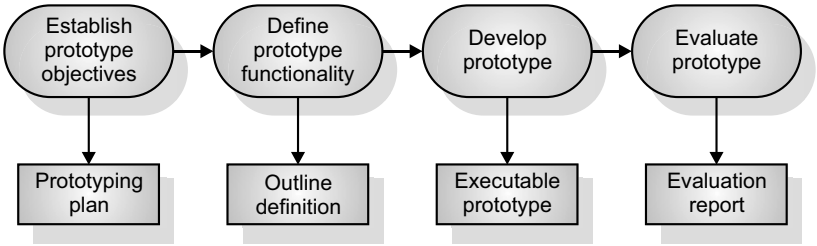


FIGURE 6.2 The Prototyping Process

Activities in this phase are as follows:

- To build a working model (prototype), formalized knowledge is mapped into the development tool framework.
- Implement the core of the prototype system (knowledge base, inference engine, and database).
- After performing the above tasks, the initial prototype (working model) is constructed. Since it's just a working model, all features are not included in it, so the knowledge and inferences are not fully developed and left for expansion in the future.
- At last, the prototype is demonstrated to the domain expert to obtain feedback and his ideas; it also helps to assess the feasibility of the system by testing it with the selected test cases. It also helps in better understanding the system. Errors are also corrected.
- Prototyping helps a domain expert in understanding what is expected from him and also provides the knowledge engineer with better insights of the whole procedure of the expert system development. Both get information about previously undiscovered possibilities with the help of the prototype.

Types of Prototypes

There are basically two types of prototypes.

6.1.4.1 Throw-Away Prototype

The throw-away prototype is built using little requirement analysis. As its name suggests, the throw-away prototype is discarded, or thrown away, after the needed requirements have been gathered and validated. This prototype does not become part of the final expert system. We can

say that the motive behind the throw-away prototype is to elicit and validate the requirements. This process continues until all the requirements are validated. When the domain expert comes to know about its expectations and has a better understanding of the system, then the prototype is thrown away and the system development starts based on the identified and validated requirements.

Thus, the throw-away prototype is used for reducing the risk of the requirements. When the risk requirements are acceptably low, the prototype is thrown away. The throw-away prototype cannot be a part of a full system because of the following:

- The throw-away prototype is unstructured and undocumented.
- It does not specify non-functional requirements.

There are various steps in throw-away prototyping:

- Gather the preliminary requirements.
- Design the prototype.
- Evaluate the prototype (users use the prototype and specify requirements).
- Repeat the process, if necessary.
- Write the final requirements and throw away the prototype.

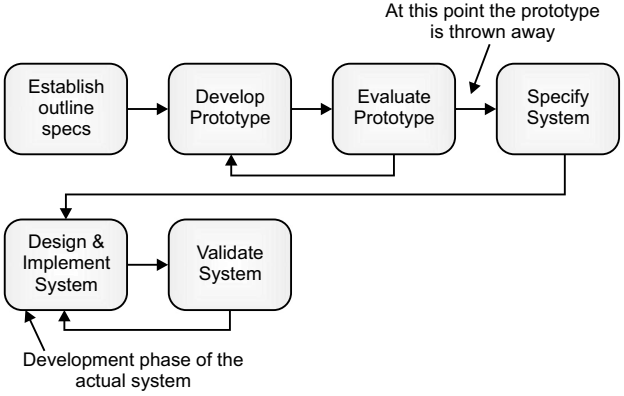


FIGURE 6.3 The Process Used with the Throw-Away Prototype

6.1.4.2 Evolutionary Prototype

The evolutionary prototype is completely different from the throw-away prototype, as it evolves into the final expert system through the iterative domain expert feedback. The evolutionary prototype is not discarded, as the main aim of the evolutionary prototype is to develop a robust prototype in a proper, structured manner so that it can finally evolve into the final expert system through refining and rebuilding it.

It is true that evolutionary prototypes do not have all the features needed, but the aim is to expand the prototype through continuous refinement by adding more knowledge until it becomes a final (finished) system knowledge base.

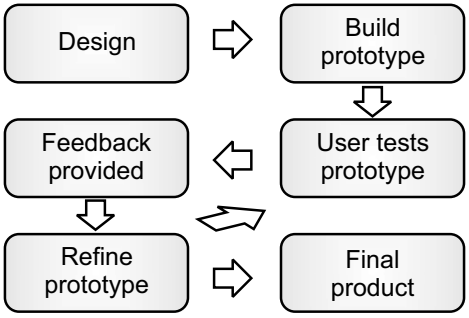


FIGURE 6.4 Evolutionary Prototyping

Where should the prototyping approach be used?

- when clients have a growing list of requirements
- web development
- application development
- when the software relies more heavily on the user input than on the background data processing task
- user interface

The advantages of the prototyping approach are as follows:

- The prototyping approach for developing an expert system is cost effective because making changes in the early stages is less costly than making changes in software after a lot of work has been done.

- This approach provides a better and clearer understanding of the proposed system, and the domain expert understands better what is expected from him.
- This approach prevents misunderstandings and miscommunication between the user and expert as both get a clear picture of what is desired so the final system will be more productive and have a good quality.
- The evaluation of the prototype gives an idea about the feasibility of the proposed expert system.
- This approach provides more user interaction and helps a client better understand the requirements of the proposed system.
- This approach also helps in finding incomplete and inconsistent requirements and allows for quicker feedback to give a better solution.
- It reduces the failure risk of the proposed expert system.

The disadvantages of the prototyping approach are as follows:

- The user sometimes gets confused between the prototype and final expert system. He starts thinking that a few modifications in the prototype will give them the final proposed system. So, the user starts expecting the developer to develop a prototype that accurately models the final system.
- In the prototyping approach, the main focus is on the development of the prototype, so a number of things are overlooked, like security issues, the system backup, and recovery.
- Similarly, the documentation of the system is absent or incomplete, as the focus is on prototype development.
- Since prototyping is a quick implementation of rough ideas, the quality of the overall system suffers due to the need to hurry in developing the prototype.
- The prototyping approach is a continuously iterative approach, but sometimes too many changes can disturb the rhythm of the development team.
- Developers may also get attached to the prototype, as they spend lot of effort in developing it. They sometimes want to see the prototype evolve into the final system.

6.1.5 Implementation

This stage is similar to the coding stage of conventional models. Implementation is just like expanded prototyping. It is a critical examination of the initial prototype that is done with repeated debugging and expanding the capability of the expert system to its full extent. This cycle is repeated until the knowledge base is finished. The initial prototype has only a sub-set of all the necessary features, so after the proper evaluation of the initial prototype, all the needed features are included in the system.

The knowledge base is expanded so as to cover whole aspects of a complete problem. That is, we get a full expert system at the end of this phase, with a proper documentation of the knowledge base as it is coded.

Activities are to

- ensure acceptance by users
- install, demonstrate, and deploy the system
- arrange the orientation and training for the users
- ensure security
- provide documentation
- arrange for integration and field testing

Table 6.5 Tasks in Implementation

Task	Objective
Coding	Implement coding.
Tests	Test code using test data, test drivers, and test analysis procedures.
Source listings	Produce commented, documented source code.
User manual	Produce the working user's manual so experts and users can provide feedback on the system.
Installation/operations guide	Document the installation/operation of the system for users.
System description document	Document the overall expert system functionality, limitations, and problems.

6.1.6 Evaluation

The final step in an expert system development life cycle is the evaluation of the complete expert system. In this stage, all features of the expert system, like the user interface and explanation facilities, are tested properly. The performance verification of the expert system is a must in this stage.

Testing (verification and validation) of an expert system is done by using a number of test cases to identify errors or weak points in the knowledge base and inference engine. It is necessary to generate a sufficient number of test cases for testing expert systems that cover the entire domain. In case errors (incompleteness and inconsistency) are present, then the knowledge base (structure and content both are refined) and inference rules are refined.

This stage includes the verification of the relationship, validation of expert system performance, and evaluation of the utility of the developed system, including evaluating the cost, benefit, accessibility, and acceptance of the system.

During validation, the following areas are checked:

- consistency and completeness of the rules
- validating rules thoroughly in order to avoid any unanticipated consequences of the interaction among the rules
- information appropriateness about how the conclusions are reached and why certain information is required
- agreement of the computer program output with the domain expert's corresponding solutions
- After passing all the tests, the delivery of the expert system is done, that is, connecting the expert system with the existing communication network (and protecting the system from unauthorized access). Then comes the maintenance stage. Maintenance tasks include correcting bugs and updating the knowledge base after its delivery.

Knowledge acquisition is a continuous process in the expert system development life cycle. This is required in all stages, but with a different purpose. For example, in the conceptualization stage, knowledge acquisition is done for gathering knowledge for building the database.

In the evaluation stage, knowledge acquisition is done for refining the knowledge base.

The following figures show the cyclic development of the ESDLC.

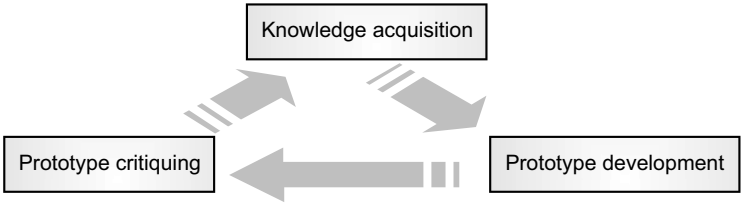


FIGURE 6.5 Cyclic Development of the ESDLC

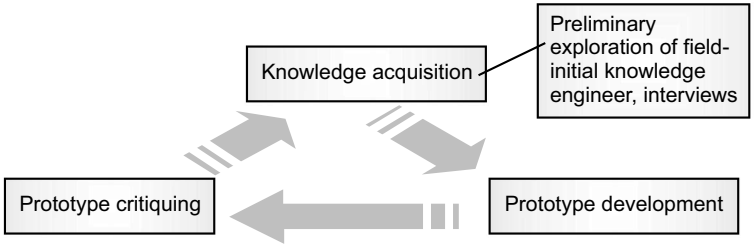


FIGURE 6.6 Cyclic Development of the ESDLC

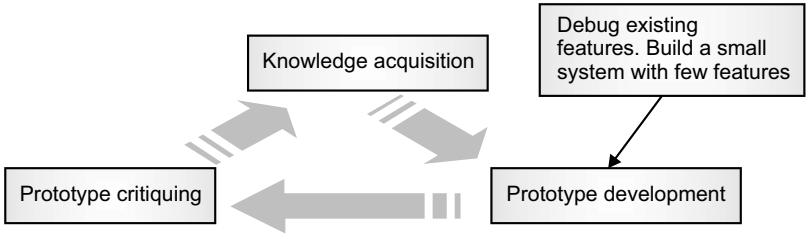


FIGURE 6.7 Cyclic Development of the ESDLC

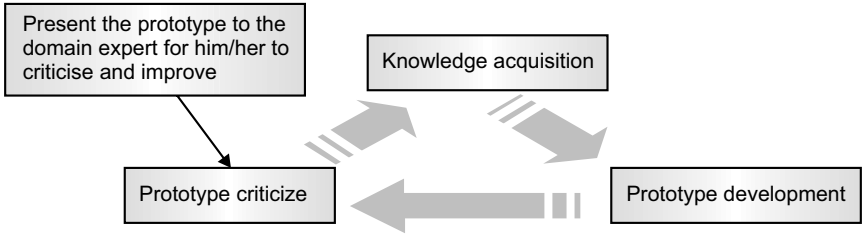


FIGURE 6.8 Cyclic Development of the ESDLC

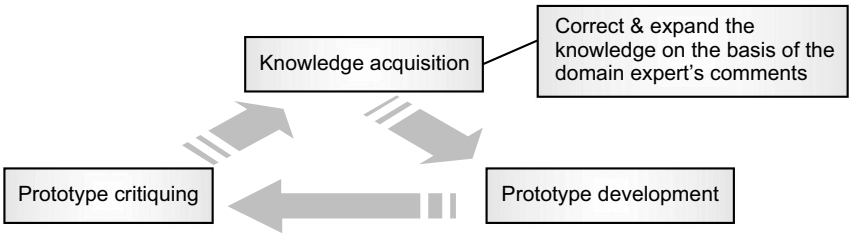


FIGURE 6.9 Cyclic Development of the ESDLC

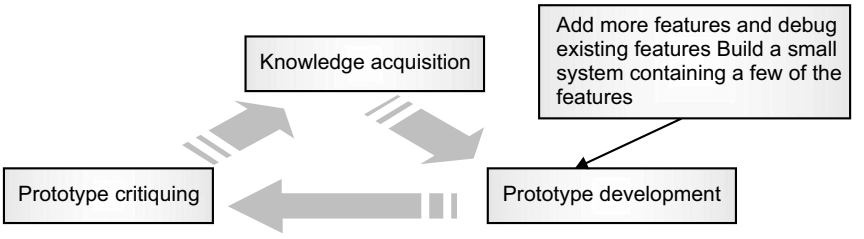


FIGURE 6.10 Cyclic Development of the ESDLC

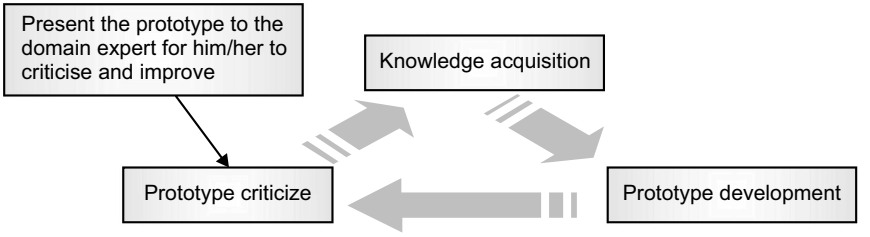


FIGURE 6.11 Cyclic Development of the ESDLC

6.2 Sources of Error in Expert System Development

- Knowledge Errors
- Syntax Errors
- Semantic Errors
- Inference Engine Errors
- Inference Chain Errors

6.2.1 Knowledge Errors

A knowledge error occurs when the knowledge obtained from the domain expert is incorrect. Since knowledge is the base of the expert system, it's necessary to provide a remedial measure for it. Otherwise, incorrect knowledge can have serious consequences. For example, incorrect knowledge will cause the wrong decision to be produced by the expert system. The verification and validation of expert knowledge must be done to scrutinize the expert knowledge.

6.2.2 Syntax Errors

A syntactical error occurs when the syntax is not followed by the rule or facts required by the development tool because the knowledge engineer is not familiar with the tool.

6.2.3 Semantic Errors

A semantic error arises when the knowledge obtained from the domain expert is not properly and correctly encoded into the rules by the knowledge engineer, so the rules do not reflect what is stated by the domain expert.

There may also be the possibility that the expert misinterprets the questions asked by the knowledge engineer. Formal protocols must be used for knowledge elicitation.

6.2.4 Inference Engine Errors

Inference engine errors arise from the malfunctioning of the inference component of an expert system, that is, there is an error in its rule interactions or conflict resolution. Such errors have serious effects as the inference engine evaluates the rules and fire rules, so incorrect solutions may be provided by the expert system.

6.2.5 Inference Chain Errors

Such errors arise by erroneous knowledge, inappropriate overall conclusions, semantic errors, inference engine bugs, inappropriate priorities of rule, and strange interactions among rules.

Exercises

Q1. What is meant by the expert system development life cycle?

Q2. What are the various stages of the expert system development life cycle?

Q3. What is meant by formalization?

Q4. Explain prototype construction.

Q5. Explain the various sources of error in expert system development.

CHAPTER 7

KNOWLEDGE ACQUISITION

Before explaining knowledge acquisition, it is necessary to understand the meaning of knowledge. Knowledge evolves from data and information. Refined data is information, and refined information is knowledge. People who possess knowledge are called experts. Different people give different meanings for knowledge.

7.1 Knowledge Basics

- "Knowledge is processed information about a domain that is then used for solving problem related to that domain."
- "Knowledge may be defined as facts, information, or skills that are obtained through several years of experience and education."
- "Knowledge is the practical or theoretical understanding of a particular domain obtained by learning, discovery, or perceiving."
- "Knowledge is familiarity or awareness obtained by experience of facts or situations."

The following are some examples of pieces of knowledge:

- John is an employee of the ACTME Company.
- All employees of ACTME earn more than $25,000.
- All employees of ACTME know that they should have a good lifestyle.

- John doesn't think that he has a good life style.
- Everybody who knows that he should have a good lifestyle and does not think that he has a good lifestyle is disappointed.

The knowledge base of an expert system has various types of knowledge:

- **Procedural knowledge:** A procedure refers to any task, so it is related to the performance of some task and a processed form of information. For example, if we have step-by-step information for solving a problem, then it is called procedural knowledge.
- **Factual knowledge:** This type of knowledge contains facts of a particular task domain, and it is found inside textbooks and journals. Such knowledge is shared widely.
- **Heuristic knowledge:** This is the opposite to factual knowledge, as it is not widely shared and discussed rarely. It is less rigorous, largely individualistic, and more experiential. Heuristic knowledge arises from good practices and good judgment and through experience.

Knowledge plays an important role in expert systems, as the expert system's knowledge base stores domain expert knowledge in rule form. Such knowledge is used for making decisions, which is the purpose of the expert system (the decision-making capability). This means the expert system's decision-making capability is based on the quality of the knowledge acquired from a domain expert.

We can say that knowledge is used for building an intelligent system that can mimic human decision-making capabilities.

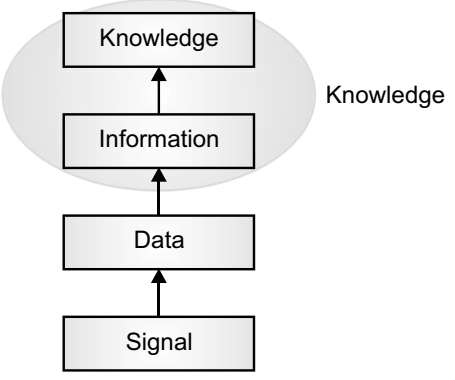


FIGURE 7.1 Knowledge = Facts + Rules + Control Strategy + (Sometimes) Faith

From Figure 7.1, it is clear that knowledge is derived from data and information. We can define "data" as "a representation of simple components of some basic element of discourse." That is, data is a special case of knowledge. Here are some examples of pieces of data:

- John is married to Sally.
- John works for the ACTME Company.
- The average salary of ACTME is $30,000.

The features of good knowledge are as follows:

- knowledge should be complete
- consistent
- voluminous
- correct and reliable

7.2 Knowledge Engineering

"Knowledge engineering is the process of building an entire knowledge base system from beginning to end." Thus, knowledge engineering includes the design, building, and then installation of an expert system. Knowledge engineering's entire process includes the following:

- acquisition of knowledge
  - ♦ general knowledge or meta knowledge
  - ♦ it comes from experts, books, documents, sensors, and files
- knowledge representation
  - ♦ organized knowledge using representation techniques like rules and frames
- knowledge validation and verification
- inferences
- explanation and justification capabilities

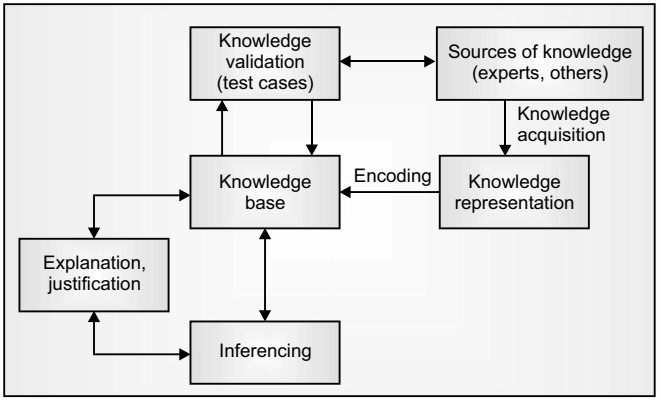


FIGURE 7.2 Knowledge Engineering Process

It is clear that knowledge acquisition is the first step of the knowledge engineering process. Knowledge representation will be explained in the next chapter.

7.2.1 Knowledge Acquisition

- Knowledge acquisition is the process of obtaining or gathering knowledge from a domain expert using various techniques.
- Knowledge acquisition is the process of obtaining knowledge and transforming that knowledge into a representational form that then can be used by the expert system for making decisions.
- Knowledge acquisition is the process of gathering knowledge for forming the knowledge base of an expert system.

Knowledge is an important part of an expert system because the expert system mimics the human decision-making capability through knowledge that is stored in its knowledge base. Knowledge acquisition is obviously the most important task in expert system development. However, acquiring knowledge is the most difficult phase in knowledge engineering, and it has what is known as the "knowledge acquisition bottleneck." This bottleneck during the development of an expert system is what causes most of the failures at the time of knowledge acquisition.

Sources of knowledge for acquisition include

- documented sources: textbooks, journals, historical records, and databases
- an undocumented source: human expert knowledge

Generally, knowledge acquired from documented sources is always insufficient for solving real-world problems because real-world problems require heuristic knowledge that can only be obtained from a human expert. The knowledge engineer is an important member of the development team of an expert system. The knowledge engineer plays an important role in knowledge acquisition. The thing that matters most in knowledge acquisition is the selection of the domain expert, because the domain expert is the source of knowledge for an expert system. A lot of time and effort are needed to select an appropriate domain expert. The following points must be kept in mind when a selecting domain expert for knowledge acquisition:

- Basically, the expert that has domain expertise and heuristic knowledge is selected for acquiring knowledge because acquiring heuristics knowledge (knowledge comes with experience) is the main goal of the knowledge acquisition process.
- The expert must have the capability of distributing his knowledge to the project team, who may have little or no idea about that domain.
- The expert must be able to communicate with his team properly and must be able to explain his reasoning process.
- Knowledge acquisition is not a one-day process; it requires lot of time, so the domain expert must be willing to devote his time and effort to the knowledge acquisition process.

After selecting an expert, that domain expert must give some information about the background of the domain to the knowledge engineer because it is the knowledge engineer who acquires the knowledge from the expert. The knowledge engineer must be able to understand the ways in which a domain expert relates various things like relationships, objects, and events.

7.2.2 Knowledge Engineer

The main task of a knowledge engineer is to acquire knowledge from a domain expert or from other sources by using acquisition techniques and then to map or encode the gathered knowledge into AI formalisms for computational purposes.

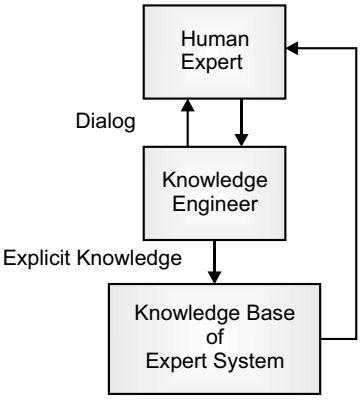


FIGURE 7.3 The Responsibility of the Knowledge Engineer

Knowledge engineers do the following tasks for developing systems:

- knowledge elicitation
- interviewing experts and creating knowledge bases
- knowledge fusion
- fusing individual knowledge bases
- coding knowledge base
- testing and evaluation of the system

The knowledge engineer obtains knowledge from various sources. The knowledge engineer's responsibility is to design, build, and test an expert system. It is the knowledge engineer who interacts between the expert and the knowledge base.

A knowledge engineer, after acquiring knowledge, decides the representational scheme for that knowledge and also decides on the programming language for knowledge encoding. The knowledge engineer's role is not restricted to knowledge acquisition and representation. He is also responsible for testing and revising the expert system.

7.2.3 Difficulties in Knowledge Acquisition

Knowledge acquisition is the most important task in knowledge engineering; it is also the most difficult one. Knowledge acquisition basically depends on the domain expert from whom the knowledge is acquired. This process depends on the expert's capability, availability, and willingness to cooperate.

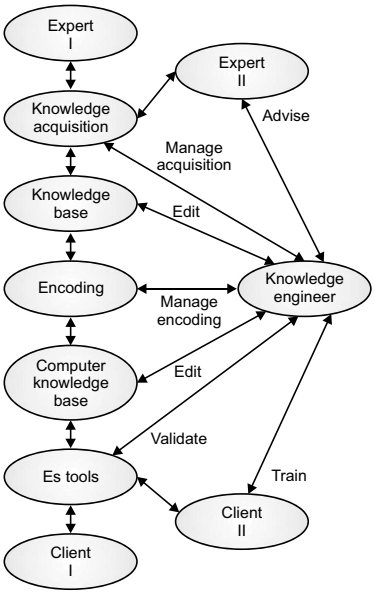


FIGURE 7.4 The Role of the Knowledge Engineer in Knowledge Acquisition

- The expert must be able to justify his reasoning or explain his reasoning, otherwise, it will become a bottleneck in the knowledge engineering process.
- The domain expert must be willing to cooperate with the team and must be able to devote his time because the knowledge obtained the from expert is the base of the expert system. An unwilling expert may be the reason for the failure of the knowledge acquisition process.
- The expert is not able to describe all that he knows about his subject domain.
- No expert knows everything.
- Experts have considerable implied knowledge that is difficult to describe.

Because of the above issues, it is necessary for knowledge acquisition techniques to do the following:

- Knowledge acquisition techniques should be able to capture tacit knowledge from the expert.
- They should be able to assemble knowledge from different experts.

- They should be able to validate the gathered knowledge.
- They should be able to focus on essential knowledge.
- They must allow non-experts to obtain an understanding of the knowledge.

7.3 Knowledge Acquisition Techniques

There are varieties of knowledge acquisition techniques, which are also called knowledge elicitation techniques. Knowledge elicitation means gathering or obtaining the knowledge from the human domain expert. That knowledge is then verified and validated for creating the knowledge base of the expert system. The knowledge elicitation tasks focus on finding at least one expert for the proposed domain who is

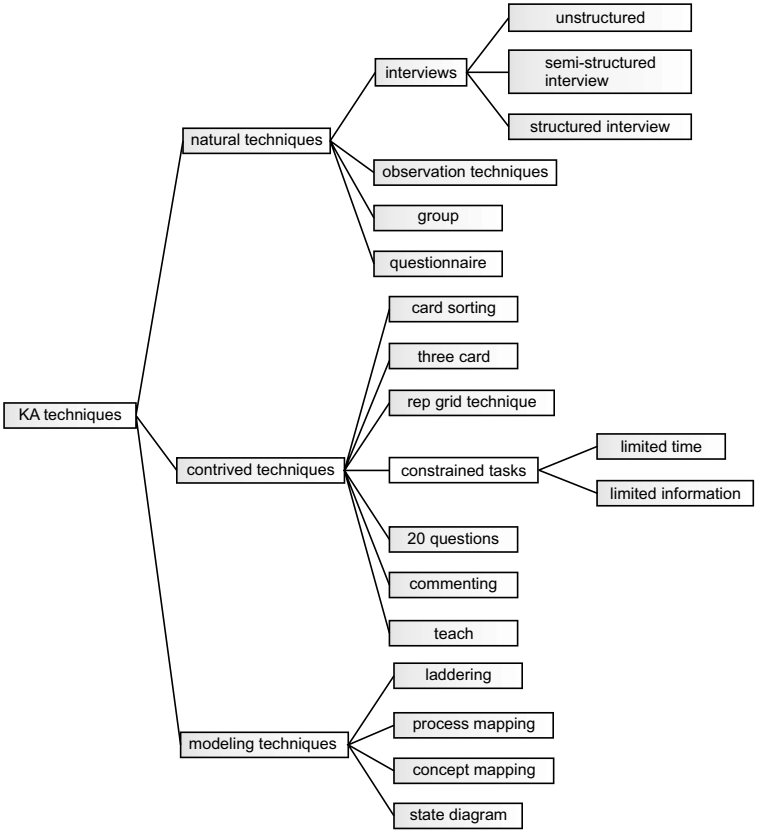


FIGURE 7.5 The Classification of Knowledge Acquisition Techniques

- able to justify his knowledge
- able to devote time for knowledge sharing
- is willing to provide his tasks.

Expert knowledge contains the following information:

- facts and principles about the domain area of the expert
- strategies for solving problems
- reasoning or justification of solutions
- **Meta knowledge**: This is knowledge about knowledge, that is, it is the knowledge of how to use particular knowledge.

Various knowledge acquisition techniques are discussed below. There are various ways of classifying knowledge acquisition techniques. **The direct acquisition method** focuses on explicit knowledge that the expert is able to communicate verbally. Interviews, questionnaires, and task observation are examples of the direct method.

Indirect methods of acquisition focus on the implicit knowledge of the expert that is difficult for expert to verbalize. Implicit knowledge is obtained through inference. Hierarchical clustering and multi-dimensional scaling are examples of the indirect acquisition method. Another form of classification depends on how the knowledge engineer interacts with the domain expert.

7.3.1 Natural Techniques

7.3.1.1 Interviews

Interviews are a direct method of knowledge acquisition; these involve face-to-face communication between the knowledge engineer and domain expert. Interviews must be held away from the expert's workplace to minimize interruptions. Usually, it takes one month or longer to interview an expert. The knowledge engineer asks questions from the expert relating to its domain and how they perform the tasks. The success of the interview depends on the quality of the questions asked and the answers given by the expert. It is difficult to extract tacit knowledge from an expert through the interview technique.

The types of interviews are

- **Structured interview:** In the structured interview, there is a pre-planned questionnaire that has to be completed by the domain expert. It is a goal-oriented technique, and it deals with some particular concept in the domain.
- **Unstructured interview:** This is just a random interview, as there are no pre-planned questions from the knowledge engineer. Both the knowledge engineer and domain expert can discuss the domain and explore it fully; this is an inefficient way of acquisition. This type of interview is done when the knowledge engineer has little knowledge of the domain.
- **Semi-structured interview:** This is a combination of the structured and unstructured types, because in it, some questions are pre-planned and the knowledge engineer can ask some random questions to clarify his understanding. This method is effective, as it helps the expert to avoid unnecessary details.
- **Problem solving interview:** In this, the domain expert is provided with a real-world problem. He is asked to solve it and describe each step and the justification of each step.
- An audio recording is done for the interview for future reference.

7.3.1.2 Observation

In this technique, expert activities for solving a real problem are observed, and a report is made based on the observations. It is good to make a video recording of the observing activities. The knowledge engineer can also ask questions after the completion of the observation task.

7.3.1.3 Questionnaire

The questionnaire method is basically used with other techniques, like the interview. In this method, a question list is prepared by a knowledge engineer when specific information is needed. This method is useful when knowledge has to be elicited from different experts.

7.3.2 Contrived Techniques

Contrived techniques are specialized techniques for capturing tacit knowledge. In such techniques, experts are asked to perform a task that they normally would not do in their job.

7.3.2.1 Concept (Card) Sorting

Concept sorting is a way of finding out or determining how an expert compares and orders concepts. It is a way of revealing tacit knowledge about classes, relations, and properties. In this method, the domain name concepts are written on cards for sorting by experts. After this, the cards are presented to the domain experts by spreading the cards on the table. The domain expert has to sort or arrange these cards so that each cluster of cards has something important in common. Then, the domain expert has to explain what principles he/she used for making such groupings or clusters of cards. The cards are collected, and the same process is repeated until the expert has nothing to sort.

7.3.2.2 The Three-Card Method

The three-card method is also used for capturing tacit knowledge and the way in which an expert explains a concept in a domain. In this method, experts are asked to select three cards randomly. He has to tell how two cards are similar and how they are different from the third one. This approach helps in determining the features of classes.

7.3.2.3 Repertory Grid Technique

The repertory grid technique is used to bring about attributes for a set of concepts and then rate the concepts against the attributes using a numerical scale. It then uses statistical analysis to arrange and group similar concepts and attributes. It allows the expert to provide a rating of each concept for an attribute in concept sorting.

Attribute	**Orientation**	**Ease of Programming**	**Training Time**	**Availability**
Trait Opposite	**Symbolic (3) Numeric (1)**	**High (3) Low (1)**	**High (1) Low (3)**	**High (3) Low (1)**
LISP	3	3	1	1
PROLOG	3	2	2	2
C++	3	2	2	3
COBOL	1	2	1	3

FIGURE 7.6 The Repertory Grid

The repertory grid technique works as follows:

1st stage

- Concepts are selected (between 6 and 15).
- The set of approximately the same number of attributes is also required.
- These should be such that the values can be rated on a continuous scale (e.g., small to large).
- These should be chosen from knowledge previously elicited.

2nd stage

- The expert provides a rating for each concept against each attribute.
- A numerical scale is used.

3rd stage

- Ratings are applied to the cluster analysis to create a visual representation of the ratings called a focus grid.
- Concepts with similar scores are grouped together, and attributes with similar scores are grouped.

4th stage

- The engineer walks the expert through the results to gain feedback and prompt further knowledge about the groupings.
- If needed, more concepts and attributes are rated and included in the grid.
- In Figure 7.7, the domain elements are certain types of crime: petty theft, burglary, drug-dealing, murder, mugging, and rape.
- This is one expert's view on the issue.
- Consider carefully whether any pair of attributes is very similar by comparing the horizontal lines in this grid.
- The closest is probably the personal/impersonal one and the major/petty one.
- Beware when making this comparison, because the expert may have inadvertently "inverted" the scale for just one of two similar constructs.

	petty theft	burglary	drug-dealing	murder	mugging	rape	
anybody	2	1	1	1	1	5	women only
long sentence	2	1	1	2	3	5	short sentence
sensational	2	5	1	1	4	5	common-place
premeditated	5	3	1	2	5	4	spur of the moment
OK for the victim	3	2	2	5	5	5	nasty for the victim
impersonal	2	2	1	5	4	5	personal
petty	1	3	1	5	4	5	major
non-violent	1	1	2	5	5	5	violent

FIGURE 7.7 A grid showing how various constructs are being ranked on a scale of 1–5. It is not always the value on the left-hand side of a scale that corresponds to the value 1.

- For example, in the example, the major/petty construct has a value of 5 for "major." If the expert had chosen 1 instead, and 5 for "petty," then this construct and the personal/impersonal one would look very different.
- Further analysis may lead you to omit one pairing of constructs.
- Following that, you would draw up a table showing how similar or dissimilar each domain element is from the others.

7.3.2.4 Constrained Task

In this method, the expert has to perform a task with constraints. This is useful for focusing the expert on essential knowledge and priorities.

7.3.2.5 20 Questions

In this method, the expert asks the knowledge engineer questions. The expert is asked to imagine that the knowledge engineer also has the same knowledge level of the concepts. The questions should be the "yes" or "no" type of question. The knowledge engineer then notes down the questions,

and there is no need for a deep knowledge of the domain by the knowledge engineer. He just has to answer "yes" or "no" randomly. This is a fast way of getting insights into key aspects and properties of the domain. This method is generally used for generating new concepts that are then added to the knowledge base.

7.3.2.6 Commentary

In this method, the expert gives a running commentary on the performance of a task that helps in the elicitation of valuable tacit knowledge.

7.3.3 Modelling Techniques

7.3.3.1 Laddering

The laddering method is used for acquiring concept knowledge. It finds similarities and differences between groups of things. It uses various trees, like the concept tree, attribute tree, and composition tree. The construction, validation, and modification of the trees are done in this method.

For example:

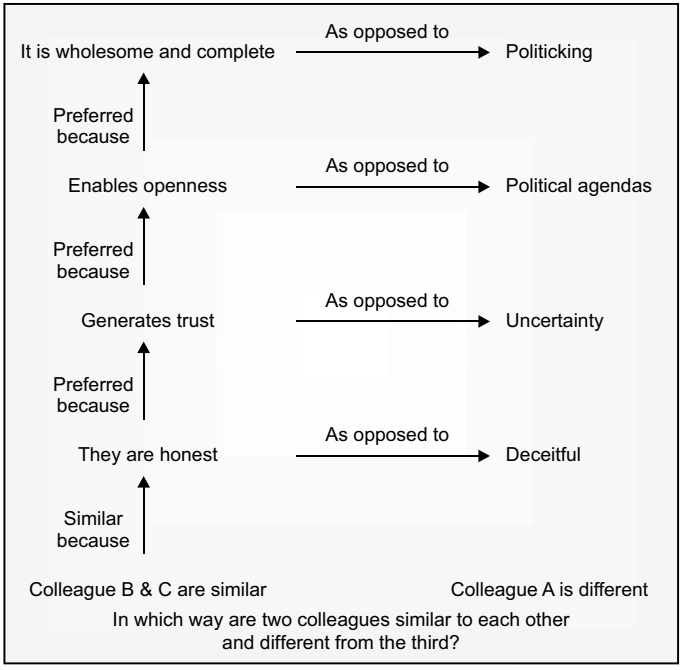


FIGURE 7.8 Laddering

7.3.3.2 Concept Tree

A concept tree is a hierarchical tree-like structure of concepts representing classes and members. The activities used to create it are to

- move nodes (concepts) around the tree
- add new nodes
- delete nodes
- re-name nodes

With the knowledge acquisition process, the knowledge analysis process occurs simultaneously. Knowledge analysis is the process of determining or identifying elements that are needed to build the knowledge base. These are

- concepts
- things that constitute a domain
- main elements of the k-base
- attributes
- qualities or features belonging to a class of concepts
- values
- specific qualities or features of a concept that differentiate it from other concepts
- relations
- the way in which concepts are associated with one another

Knowledge acquired from a domain expert is then represented using knowledge representing techniques like frames, semantic nets, and production rules, which we will study in the next chapter. These will improve the knowledge engineer's understanding of the subject domain.

Exercises

Q1. What is meant by knowledge?

Q2. Explain the various steps in knowledge engineering.

Q3. Explain the knowledge acquisition role in knowledge engineering.

Q4. Explain the various knowledge acquisition techniques.

CHAPTER 8

KNOWLEDGE REPRESENTATION

In the previous chapter, we discussed knowledge. Knowledge plays an important role in AI. There is no intelligence without knowledge. To solve the complex problems faced in artificial intelligence, a large amount of knowledge is required. There must also be some way of representing and manipulating the knowledge. Knowledge involves the fact or condition of being aware of something or knowing something gained through experience. We can say that knowledge may be defined as facts, information, or skills that are obtained through several years of experience and education.

8.1 Definitions of Knowledge Representation

- "Knowledge representation is a substitute or a surrogate for knowledge that helps in reasoning and deducing new facts and decision-making ability."
- Knowledge representation provides a computational medium or environment where reasoning can be accomplished.
- Knowledge representation is a way of organizing information so that the appropriate inference can be done.
- It is a set of ontological commitments, i.e., an answer to the question "In what terms should I think about the world?"

- Knowledge consists of models that attempt to represent the environment in such a way as to maximally simplify problem-solving. It is assumed that no model can ever hope to capture all relevant information.
- It is a fragmentary theory of intelligent reasoning, expressed in terms of three components:
  - ♦ the representation's fundamental conception of intelligent reasoning
  - ♦ the set of inferences the representation sanctions
  - ♦ the set of inferences it recommends
- Knowledge representation is an issue that arises in both cognitive science and AI.
  - ♦ In cognitive science, it is concerned with how people store and process information.
  - ♦ In AI, the primary aim is to store knowledge so that programs can process it and achieve the verisimilitude of human intelligence.
  - ♦ AI researchers have borrowed representation theories from cognitive science.

Knowledge representation (KR) is an important concern in both cognitive science and artificial intelligence.

- In cognitive science, it deals with way people store and process information.
- In AI, the main concern is about storing knowledge so that programs can process it and achieve human-like intelligence.

8.2 Characteristics of Good Knowledge Representation

- It should
  - ♦ be able to represent the knowledge important to the problem
  - ♦ reflect the structure of knowledge in the domain

- (Otherwise, our development is a constant process of distorting things to make them fit.)
  - ♦ capture knowledge at the appropriate level of granularity
  - ♦ support incremental, iterative development
- It should not
  - ♦ be too difficult to reason about
  - ♦ require that more knowledge be represented than is needed to solve the problem

8.3 Basics of Knowledge Representation

Knowledge should be represented in the proper format to get its benefits. Knowledge representation and its reasoning are central to AI. To represent knowledge, it is necessary to understand the following two entities:

- **Facts:** These contain the truth that we want to represent
- **Representation of Facts:** This refers to representing facts in some chosen format so that they can be used and manipulated easily.

These two entities can be structured at two levels:

- **Knowledge Level:** The description of facts occurs at this level
- **Symbol Level:** Facts are represented in terms of symbols

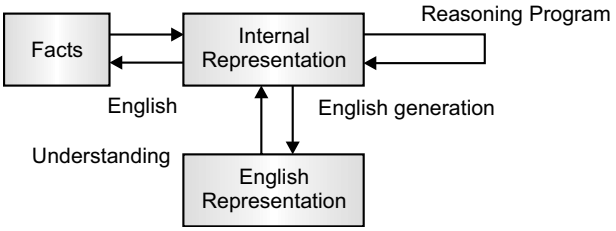


FIGURE 8.1 Mapping Between Facts and Representations

The above figure shows that the focus is on facts, representations, and two ways of mapping between them. Forward mapping is from fact to representation, and backward mapping is from representation to facts. Facts are represented in two ways in the above figure, that is, one is an internal

representation of fact that we use in programs and the other is a natural language representation of the facts. There is mapping from the natural to internal representation and vice-versa.

Consider the following example of an English sentence:

Dober is a dog.

We can represent this fact in logic as follows:

```
Dog(Dober)
```

Suppose also we have a logical representation of the fact: all dogs have tails, as shown on the next page.

```
"x: dog(x)→hastail(x)
```

Using the deductive mechanisms of the logic, we may generate the new representation object:

```
hastail(Dober)
```

Now using backward mapping, the English sentence can be generated:

Dober has a tail.

This newly generated fact can be used for deducing new facts or can be used for taking some actions. It is important to note that the above mapping is not a one-to-one relation; it is showing a many-to-many relation. Normally, many-to-many relations are found in the mapping of English representations of facts. For example, consider two English sentences:

- All dogs have tails.
- Every dog has a tail.

It is clear that both sentences represent the same fact ("Every dog has at least one tail."), and that is a many-to-one relation. The symbolic representation of knowledge should have the properties discussed in the next section.

8.4 Properties of the Symbolic Representation of Knowledge

- Make references explicit: Since natural language is ambiguous, it's necessary to explicitly refer to the entities.

  The stool was placed on the table.

  It was broken.

The stool (r1) was placed on the table (r2).

It (r1) was broken.

(Now it becomes obvious what was broken.)

- Referential uniqueness refers to removing all the ambiguities related to the entities in their internal representation.

  "David should do it."

  "David who?"

  Now instead of using the same name multiple times, a unique name must be given, like david-1 or david-2.

- Semantic uniqueness: the meaning of each symbol must be unique in its internal representation

  Jackie caught a ball. [Catch-object]

  Jackie caught a cold. [Catch-illness]

- Functional uniqueness: the functional aspect must be unique in the symbolic representation.

  Peta catches the ball.

  The ball Peta catches.

  The ball is caught by Peta.

Who is the catcher? Who or what is the caught object?

8.5 Properties for the Good Knowledge Representation Systems

- **Representational Adequacy:** The system should have the ability to represent all types of knowledge (such as procedural and factual knowledge).
- **Inferential Adequacy:** The system should have ability to infer or deduce new knowledge from old knowledge.
- **Inferential efficiency:** The system should have the ability to focus the inferential mechanism in the most promising direction to achieve good results.

- **Acquisitional efficiency:** This refers to the ability to acquire new information easily.

There are various ways of representing the knowledge, but it's necessary to choose the knowledge representation mechanism very carefully, otherwise project can fail. Knowledge representation schemes are divided into following categories.

8.6 Categories of Knowledge Representation Schemes

Logical Representation Schemes

This scheme of representations uses formal logic method to represent a knowledge base, for example, formal logic.

Network Representation Schemes

A network representation scheme uses a hierarchical structure for representing knowledge, such as a graph in which the nodes represent objects or concepts in the problem domain and the arcs represent the relations or associations between them. It is used to represent the "is-a" relationships, where a general type (for example, a ball) is linked to more specific types (i.e., rubber, golf, baseball, or football) that inherit the basic properties of the general type. Inheritance benefits in a compact representation of knowledge and an algorithm of reasoning can be applied at various levels of granularity or abstraction. Semantic networks and conceptual graph are examples of network representation schemes.

Structured Representation Schemes

A structured representation scheme is an extension of the network scheme. That is, each node can be a complex data structure consisting of named slots with attached values. Scripts and frames are examples of this scheme.

Procedural Representation Schemes

- Procedural schemes are used for representing knowledge as a set of instructions for solving a problem. This differs from the declarative representations provided by logic and semantic networks.
- A production rule system is an example of this approach in which the "IF….THEN" rule is used to represent knowledge. In the previous chapter, we already discussed how to represent knowledge using rules and how to do reasoning on that.

One of the simplest ways of representing declarative knowledge is through a database that shows a set of relations and objects and their attributes.

Table 8.1 A Database Representing Knowledge

Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right

The problem is that such a representation scheme has a weak reasoning capability. It cannot answer questions like "Who is the heaviest player?," but such a scheme may act as input to some powerful inference engine and can compute the answer if some procedure is provided.

8.7 Types of Knowledge Representational Schemes

There are various types of knowledge representational schemes. These are

- Formal Logic
- Semantic Net
- Frames
- Scripts
- Conceptual Dependency

8.7.1 Formal Logic

Formal logic is basically used for representing inferential knowledge, or knowledge from which new knowledge and the decision-making capability can be inferred. Formal logic is a language that has its own syntax and semantics, and it defines how to make a sentence and the conclusions that can be drawn from that. Formal logic may be defined as

- a language with concrete rules
- having no ambiguity in representation

- that which allows unambiguous communication and processing
- is very unlike natural languages, e.g., English

Let us understand meaning of the syntax and semantics of a sentence before going into further details of formal logic:

- Syntax
  - ♦ means rules that should be used for constructing legal sentences in the logic
  - ♦ which symbols we can use (English: letters, punctuation)
  - ♦ how we are allowed to combine symbols
- Semantics
  - ♦ refers to the meaning of a sentence, that is, how to interpret (read) sentences in the logic
  - ♦ assigns a meaning to each sentence

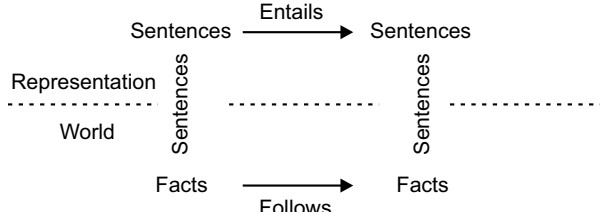


FIGURE 8.2 Mapping of sentences to facts

Figure 8.2 shows the mapping of sentences to facts; mapping determines which object is related to or referenced by which other object. The way one fact follows another should be mirrored by the way one sentence is entailed by another. For example, consider the sentence "All teachers are seven feet tall."

- This is a valid sentence (syntax).
- We can understand the meaning (semantics).

Formal Logic = Formal Language + Semantics

We can compare logic to natural languages (expressive, but context sensitive) and programming languages (good for concrete data structures, but

not expressive). Logic combines the advantages of natural languages and formal languages. Logic is

- concise
- unambiguous
- context insensitive
- expressive
- effective for inferences

Formal logic is used for representing facts in a precise and unambiguous manner. Formal logics provides a powerful structure in which the relationships among the values can be described. There are two types of formal logic.

8.7.1.1 Propositional Logic

Propositional logic is the simplest logic, but it is not very expressive. A proposition is a statement that is either true or false, but it cannot be both at same time, for example, "Ram is a boy" or "The sun rises in the east." Basically, we use the symbols *P* and *Q* for representing a proposition (i.e., "P: Ram is a boy."). The symbols are in uppercase letters. Proposition symbols are used for representing facts.

There are two types of proposition, simple and compound propositions.

Simple proposition: This cannot be further broken down (that is, it is "atomic"), for example, "P: Darbe is a dog."

Compound proposition: This contains another proposition as its part, for example, "Ram is a boy and he is clever." This statement contains more than one proposition.

Formulas are used for representing propositions (*P*, *Q*, *R*) in formal logic and some connectives can be used like a conjunction (AND), disjunction (OR), and implications (*P* implies *q*) for constructing a compound statement.

Types of rules for compound propositions:

- **Conjunction (and):** $(p \wedge q)$ indicates that p and q both must be true for getting a true result.
- **Disjunction (or):** $(p \vee q)$ indicates that either p or q or both must be true for getting true as a result.

- **Implication ($p \Rightarrow q$)** consists of a pair of sentences separated by the $\Rightarrow$ operator and enclosed in parentheses. The sentence to the left of the operator is called the *antecedent*, and the sentence to the right is called the *consequent*.
- **Equivalence ($p \Leftrightarrow q$)** is a combination of an implication and a reduction.
- **Negation:** ¬p indicates the opposite of p is a simple proposition (not compound) because it contains only a single statement. A literal is an atomic sentence or negated atomic sentence like p, $\neg p$.

Here are some examples of representing a sentence in proposition logic:

***P*:** It is humid

P $\Rightarrow$ ***Q*:** If it is humid, then it is hot.

P $\wedge$ ***Q*** $\Rightarrow$ ***R*:** If it is hot and humid, then it is raining.

Here are some formulas:

- If P is a sentence, then $\neg P$ is also a sentence (negation)
- If $P1$, $P2$ are sentences, then $P1 \wedge P2$ is also a sentence (conjunction)
- If $P1$, $P2$ are sentences, then $P1 \vee P2$ is also a sentence (disjunction)
- If $P1$, $P2$ are sentences, then $P1 \Rightarrow P2$ is also a sentence (implication)
- If $P1$, $P2$ are sentences, then $P1 \Leftrightarrow P2$ is also a sentence (biconditional)

Tautology: It is a statement that is true only in every possible interpretation.

For example, p V $\neg p$ is a tautology.

"Men are mortal."

Truth Table: This is a table that contains the symbols of propositions indicating propositions like P and Q; each symbol can either take a true or false value. Depending upon the formula used (that is, a conjunction or disjunction, or something else) a result is obtained. Truth table rules for compound propositions that specify truth values for a complex sentence can be expressed as shown in Table 8.2.

Table 8.2 The Truth Table of Five Logic Connectives

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

Table 8.3 Various Properties or Rules Applicable to Propositions

Commutative	$E \wedge F \Leftrightarrow F \wedge E$
	$E \vee F \Leftrightarrow F \vee E$
Distributive	$E \wedge (F \vee G) \Leftrightarrow (E \vee F) \vee (E \wedge G)$
	$E \vee (\mathrm{F} \wedge \mathrm{G}) \Leftrightarrow (E \vee F) \wedge (E \vee G)$
Associative	$E \wedge (F \wedge G) \Leftrightarrow (E \wedge F) \wedge G$
	$E \vee (F \vee G) \Leftrightarrow (E \vee F) \vee G$
De Morgan's	$C(E \wedge F) \Leftrightarrow \neg E \vee \neg F$
	$\neg(E \vee F) \Leftrightarrow \neg E \Leftrightarrow \neg F$
Negation	$\neg(\neg E) \Leftrightarrow E$

Deduction in Proposition Logic

There are a number of inference systems for propositional logic. Most of them are actually restrictions of inference systems for first-order logic. The Hilbert-style inference system consists of the following axiom schemes (Mendelson, 1997):

(A1) $A \Rightarrow (B \Rightarrow A)$

(A2) $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

(A3) $(\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B)$

and the inference rule *modus ponens*: $A, A \Rightarrow B \vdash B$.

A *proof* or a derivation in a Hilbert system is a finite sequence of formulas such that each element is either an axiom or follows from earlier formulas by the rule of inference.

A *proof* of a derivation from a set S of formulas is a finite sequence of formulas such that each term is either an axiom, or is a member of S, or follows from earlier formulas by the rule of inference.

If there is a proof for A, then A is a *theorem*, and we denote that by A. For example, it can be proved that $A \Rightarrow A$ is a theorem, as follows:

1. $(A \Rightarrow ((A \Rightarrow A) \Rightarrow A)) \Rightarrow ((A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A))$ (instance of $A2$)
2. $A \Rightarrow ((A \Rightarrow A) \Rightarrow A)$ (instance of A1)
3. $(A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A)$ (from 1 and 2, by MP)
4. $A \Rightarrow (A \Rightarrow A)$ (instance of A1)
5. $A \Rightarrow A$ (from 3 and 4, by MP)

This may seem like a lot of work to prove that "A implies A," but that's the nature of formal logic systems! Derivations are broken down into extremely small steps that are rigorously mathematically justified. In a commonsense inference, we tend to proceed in large leaps instead, at least on the conscious level. But unconsciously, our brains are carrying out multitudes of small steps, though the analogy between these small steps and the small steps in logical proofs is a subject of debate in the AI and cognitive science community.

Strengths and Weaknesses of Propositional Logic

- Propositional logic is an easy way to represent real-world knowledge that can be used for problem solving.
- Propositional logic is simple to use and to deal with, and it is declarative.
- Propositional logic permits conjunctive/disjunctive/partial/negative information.
- Real world facts can be written as well-formed formulas (well-formed formulas), e.g.,

  Socrates is a man. `SOCRATESMAN`

  Ramesh is a man. `RAMESHMAN`

  It is cold. `COLD`

- Propositional logic has a very limited expressive power. For example, "search for a candle in all local shops" has a clear meaning to search all

shops in the locality for a candle. But propositional logic will require a separate statement for each shop.

- Propositions can be deceptive or extremely difficult to use to draw a meaningful conclusion, e.g., `IRFANMAN` and `INZMAMMAN` produce totally different assertions.
- Propositional logic assumes the world is all full of facts, so it constitutes well-formed formulas.
- It cannot represent the properties of an object.
- The facts like "Peter is a man," "Paul is a man," and "John is a man" can be symbolized by P, Q, and R, respectively, in PL. We cannot draw any conclusions about similarities between P, Q, and R. Better representations of these facts can be done through predicate logic.
- Reasoning with propositional logic is difficult. For example, it is impossible to represent this categorical syllogism in propositional logic:

  Every person is mortal

  Tony Blair is a person

  Therefore, Tony Blair is mortal

8.7.1.2 Predicate Logic

The weakness of propositions can be removed by using predicate logic, as it is much more expressive than propositional logic. The components, elements, and terms used in predicate logic are

- **Constants:** Ramesh, Alysa, 2, 2013, March
- **Functions:** These are a total map that associates one single value to the ordered collection of its arguments. For example: brotherof, gt, and lt.
- **Variables:** x, y, z
- Predicates can have the values true or false.
- A predicate can take arguments, e.g., `man(Ramesh)` or `gt(3,2)`
- A predicate with one argument shows the property of the bracketed argument or object, e.g., `teacher(Mukesh)`

- A predicate with two arguments relates the arguments with each other, e.g., `Brother(Mukesh, Suresh)`
- A predicate without any argument is a proposition or zero order logic.
- **Quantifiers:** Universal ∀ (something is true for all objects) means for all; ∀*x*: gt (*y*, *x*) means for all values of *x*; *y* will be greater than *x*.
  - ♦ Remember that the upside-down A stands for "all." Thus, in the sentence "For all x, if x is a king, then x is a person," the symbol x is called a *variable*. By convention, variables are lowercase letters, A variable is a term all by itself, and as such, it can also serve as the argument of a function—for example, `LeftLey(x)`. A term with no variables is called a ground term.
  - ♦ Intuitively, the sentence ∀*x P*, where *P* is any logical expression, says that *P* is true for every object *x*. More precisely, ∀*x P* is true in a given model if *P* is true in all possible extended interpretations constructed from the interpretation given in the model.
- **Existential Quantifier** ∃ (something is true for at least one object) means there exists at least one *x*, e.g., ∃*x*: *eq*(*y*, *x*) means there exists at least one *x* for which *y* equals to *x*.
  - ♦ Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it by using an existential quantifier.
  - ♦ For example, "King John has a crown on his head."
  - ♦ This sentence can be written as: $Crown(x)∧OnHead(x, John)
  - ♦ ∃*x* is pronounced "There exists an x such that ..." or "For some x ..."
  - ♦ Intuitively, the sentence ∃*x P* says that *P* is true for at least one object *x*.
- **Nested Quantifiers:** For constructing complex sentences, multiple quantifiers can be used in same sentence.
  - ♦ The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

    ∀`x` ∀`y` Brothers (`x`, `y`)=Sibling (`z`, `y`)

- ♦ Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

  $\forall x, y$ Sibling $(x, y), \Leftrightarrow$ -Sibling(y, x).

- ♦ In other cases, we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves:

  $\forall x\ y$ Loves(x, y).

- ♦ On the other hand, to say "There is someone who is loved by everyone," we write

  $\exists x\ \forall y$ *Loves*(x, y).

- **Atomic Sentence:** This consists of a predicate symbol optionally followed by terms in parentheses. For example: Brother (Mukesh, Suresh). This predicate shows that Mukesh is the brother of Suresh. An atomic sentence can consist of complex terms as arguments. For example,

  Married(Father (Richard), Mother (John))

The above states that "Richard," a father, is married to "John's" mother (again, under a suitable interpretation).

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

Predicate logic uses the concept of quantifiers for referring to a set of objects. That is, it deals with objects, attributes of objects, and the relationship between objects, so it is a good way to represent almost any type of knowledge.

For example, consider following statements: "Ram likes apples" and "Today is wet."

This proposition can be represented using predicate logic:

Properties: is wet (today)

↓ ↓

predicate arguments

Relations: likes (Ram, apple)

That is "is wet" is an attribute or property and "like" is the relationship between "Ram" and the "apple" object.

Let us discuss another example: "Thailand is cold in the winter" can be represented in three single parameter using predicate logic: place (Thailand), temperature (cold), and season (winter). Or it can be represented a single relation: cold (Thailand, winter) and winter (Thailand, cold).

Some examples of predicate logic are

- "x loves y" is represented as LOVE(x, y), which maps it to true or false when x and y get instantiated to actual values.
- "John's father loves John" is represented as LOVE(father(John), John).
  - Here, "father" is a function that maps "John" to his father.
- x is greater than y is represented in predicate calculus as GT(x, y).
- It is defined as follows:

$$\text{GT}(x, y) = T, \text{ if } x > y$$
$$= F, \text{ otherwise}$$

- Symbols like GT and LOVE are called predicates.
  - Predicates have two terms and map to T or F depending upon the values of their terms.

Translate the sentence "Every man is mortal" into a predicate formula.

- Represent the statement in predicate form
  - "x is a man" and "MAN(x),
  - x is mortal" by MORTAL(x)
- Every man is mortal:

  $(\forall x)\,(\text{MAN}(x) \rightarrow \text{MORTAL}(x))$

Types of Predicate Logic: There are three types of predicate logic depending upon the number of arguments:

- ♦ **Zero Order Predicate Logic:** This does not have any arguments.

- **First Order Predicate Logic or First Order Predicate Calculus (FOPL or FOPC):** This is more expressive than proposition logic and has one argument only. For example, Man (Ram) is a representation of "Ram is a Man," a sentence in FOPL.

Some terms used in FOPL are

- constants, like the objects John, apples, and Ram
- predicates, which are properties and relations of objects like John and apple
- Likes (John, apples) is an example of a predicate showing a relationship and a mortal (person) showing the properties of the object "person." Other examples are person and king.
- Functions transform objects:

  Likes (john, fruit of (apple _tree))

- variables, which represent any object: likes(*X*, apples)
- quantifiers, which qualify the values of variables
- true for all objects (Universal):

  $\forall X$. likes(*X*, apples)

- having at least one object (existential): $\exists X$. likes(*X*, apples).

Consider the statement "Every Monday and Wednesday, I go to Ram's house for dinner."

In FOPL, or simply predicate logic, this statement can be written as

$\forall X$ ((day_of_week (*X*, Monday) $\wedge$day_of_week(*X*, weds)) (go_to (me, house_of (john) $\vee$eat (me, dinner))).

Here, the symbols Monday, Wednesday, me, dinner, and John are all constants, base-level objects in the world about which we want to talk. The symbols day_of_week, go_to, and eat_meal are predicates that represent relationships between the arguments that appear inside the brackets. For example, in eat_meal, the relationship specifies that a person (first argument) eats a particular meal (second argument). In this case, we have represented the fact that "me" eats dinner. The symbol X is a variable, which can take on a range of values. This enables us to be more expressive, and in particular, we can quantify X with the "for all" symbol,

so that our sentence of predicate logic talks about all possible X's. Finally, the symbol house_of is a function, and - if we can - we are expected to replace house_of (John) with the output of the function (John's house) given the input to the function (John).

A model in first-order logic consists of a set of objects and an interpretation that maps the constant symbols to objects, predicate symbols to relations on those objects, and function symbols to the functions on those objects.

Here is one more example of a statement written in predicate calculus:

Suppose that c stands for "the cat," m stands for "the mat," s stands for "sits on," b stands for "black," f stands for "fat," and h stands for "happy". The statement "If the fat black cat sits on the mat then it is happy" is written in predicate logic as

$$(f(c) \wedge b(c) \wedge s(c, m)) \Rightarrow h(c)$$

Example of a statement written in predicate calculus using quantifiers: Suppose that d stands for "is a day," p stands for "is a person," mo stands for "is mugged on," mi stands for "is mugged in," S stands for Soho, x stands for some unspecified day, and y stands for some unspecified person. Then the statement "Someone is mugged in Soho everyday" is written in predicate logic as

$$\text{“}\forall x(d(x) \Rightarrow \exists y(p(y) \wedge mo(y, x) \wedge mi(y, S)))$$

Note that sentence "Someone is mugged in Soho everyday" is unambiguous, but its corresponding predicate calculus representation is not ambiguous.

Using Predicates to Represent Relationships

In order to represent a relationship between individual objects, we can use a predicate specifying the objects as its arguments.

- "Alison likes Richard and chocolate."
- Representation using the predicate is likes(alison, richard) $\wedge$ likes(alison, chocolate)

Using Predicates within a Rule

"If Richard is a friend of Alison, then Alison likes Richard."

The above sentence is represented using a predicate like that:

friends(alison, richard) $\Rightarrow$ likes(alison, richard)

Using Variables with Predicates to Capture Generalizations

We can capture generalizations by asserting that any instance of a given class has the relevant property. For example,

- "Every elephant is grey."
- $\forall X$: elephant(X) $\Rightarrow$ grey(X)

Using Quantifiers and Variables

We can use quantification to distinguish general and specific assertions.

- "There is a white alligator:" $\exists X$: alligator(X) $\wedge$ white(X)
- "Alison eats everything that she likes:" $\forall X$: likes(alison, X) $\Rightarrow$ eats (alison, X)
- "There is some bird that doesn't fly:" $\exists X$: bird(X) $\Rightarrow \neg$flies(X).
- "Every person has something that they love:" X: person(X); Y: loves(X, Y)

Representation of the following sentences in predicate logic

Suppose we want to convert following sentences into predicate logic:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeiians were Romans.
4. Caesar was a ruler.
5. All Pompeiians were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The conversion is as follows:

1. Man (Marcus)
2. Pompeian (Marcus)
3. $\forall x$: Pompeian(x) $\Rightarrow$ Roman(x)
4. ruler(Caesar)

5. $\forall x$: Roman(x) $\wedge$ loyalto(x, Caesar) $\wedge$ hate(x, Caesar)
6. $\forall x$:$\exists$ y: loyalto (x, y)
7. $\forall x$: $\forall y$: person(x) $\wedge$ ruler(y) $\wedge$ tryassassinate(x, y) ¬loyalto(x, y)
8. tryassassinate (Marcus, Caesar)

In FOPL there is an instance (object of class) and is-a relationship (class inheritance). For example,

- Man (Marcus) shows an instance, that is, "Marcus" is an instance of the class "man." This can also be written like "instance (Marcus, Man)," showing the same instance relationship.
- Is-a (Pompeiian, Marcus) shows the is-a relationship between the Pompeiian and Marcus, that is, the Pompeiian subclass is derived from the Marcus superclass.

The main problem with FOPL is that its expressiveness complicates the process of inference. Also, in FOL you cannot construct sentences that make assertions about other sentences. For example, you cannot say things like "there exists a property such that..." For this task, you need higher-order logic.

- High order predicate logic: It is more expressive than the first order predicate logic, as it allows quantification over functions and predicates, as well as objects.

  For example, "All our polynomials have a zero at 17" yields

  $f\,(f(17)=0)$.

This is important to AI, but is not often used, as it is harder to reason through.

Deduction in predicate logic: Deduction in first-order logic is similar conceptually to its analogue in propositional logic, but more complex in detail due to the presence of quantified variables. There are several different deductive systems available; one of the first was developed by Hilbert in the early 20th century and we will describe it now. In Hilbert's system, formulas are built using only the connectives $\Rightarrow$ and ¬, and the quantifiers $\forall$("for all") and $\exists$("there exists"). The system consists of the following axiom schemes:

- (A1) $A \Rightarrow (B \Rightarrow A)$
- (A2) $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

- (A3) $(\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B)$
- (A4) $(\forall x)A \Rightarrow A[x \rightarrow t]$, while the term t is free for x in A
- (A5) $(\forall x)(A \Rightarrow B) \Rightarrow (A \Rightarrow (\forall x)\ B)$, while A does not involve free occurrences of x and the following inference rules

Modus ponens: A, A $\Rightarrow$ B ‖ B

Gen: $A \parallel (\forall x)A$

A proof or a derivation in a Hilbert system is a finite sequence of formulas such that each element is either an axiom or follows from earlier formulas by one of the rules of inference. A proof of a derivation from a set *S* of formulas is a finite sequence of formulas such that each formula is either an axiom, or is a member of *S*, or follows from earlier formulas by one of the rules of inference. If there is a proof for *A*, then *A* is a theorem and we denote that by *A*. There is a link between the semantics of first order logic and the above.

8.7.1.3 Introduction to Resolution

Resolution is a procedure or algorithm to produce proof for the facts (that are represented by sentence) by virtue of contradiction. Resolution takes two clauses having complementary literals (it is an atomic symbol or its negation P, $\neg P$) as an input and produces a new clause. The resolution rule was discovered by Alan Robinson in the mid-1960s. For example, if you want to prove that a certain theorem is true, then you have to prove that the negation of that theorem is not true.

Let us understand the resolution: Suppose that we know the following two facts:

- not feathers (Tweety) or bird (Tweety)
- feathers (Tweety)

Sentence 1 shows that either Tweety does not have feathers or else Tweety is a bird. Sentence 2 shows that Tweety has feathers. Now to prove that Tweety is a bird, first we have to prove an assumption that is the negation of that predicate, giving sentence 3: not bird (Tweety).

In sentences 1 and 2, "not feathers (Tweety)" and "feathers (Tweety)" cancel each other out. Resolving sentences 1 and 2 produces the resolvent, sentence 4, which is added to our fact set: "bird (Tweety)."

So it's clear that sentences 3 and 4 cannot both be true, either Tweety is a bird or it is not. Thus, we have a contradiction. We have just proved that our first assumption, "not bird (Tweety)," is false, and the alternative, "bird (Tweety)," must be true.

Resolution takes clauses as an input and produces a new clause.

8.7.1.4 Conjunctive Normal Form

A clause must be in the conjunctive normal form, that is, a conjunction of clauses where the clause is the disjunction of literals where the literal and its complement cannot appear in same clause, like p ∨q ∨ r is a disjunction of literals. A single literal can also be considered as a disjunction of one literal only that is called a unit clause. It is also important that the resolution algorithm can only be applied to a clause only so the resolution is relevant only for the knowledge base that has that clause only. If a clause is not in the conjunctive or disjunctive normal form, then before applying that resolution, it's necessary to convert these clauses into the conjunctive or disjunctive form.

Any formula can be converted into the conjunctive normal form using following equivalence:

- $P \Rightarrow Q = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
- $P \Rightarrow Q = \neg P \vee Q$
- $\neg(P \wedge Q) = \neg P \vee \neg Q$ (De Morgen's law)
- $\neg(P \vee Q) = \neg P \wedge \neg Q$ (De Morgen's law)
- $\neg \neg P = P$ (Negation rule)
- $P \vee Q \wedge R = (P \vee Q) \wedge (P \vee R)$ (Distributive law)

The following are the examples of clauses (disjunction of literals)

- p
- $\neg p$
- $p \vee q$
- $p \vee \neg r \vee \neg p$
- $\neg s \vee t \vee p$

The following are the examples of CNF

- $(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$
- $A \vee B$
- $A \wedge B$

Conversion of axiom to conjunctive normal form (clausal form):

Every propositional formula can be converted into its equivalent conjunctive normal form (CNF) by applying the rules of logical equivalence, like the negation rule, De Morgen's rule, and the distributive rule.

1. Eliminate implications.

Using the rule

$$A \rightarrow B \equiv \neg A \vee B$$

We may eliminate all occurrences of $\rightarrow$.

Example:

$$p \rightarrow ((q \rightarrow r) \vee \neg s) \equiv p \rightarrow ((\neg q \vee r) \vee \neg s)$$
$$\equiv \neg p \vee ((\neg q \vee r) \vee \neg s)$$

2. Move negations down to the atomic formulas.

Using De Morgan's Laws and the double negation rule,

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg\neg A \equiv A$$

We push the negations down towards the atoms until we obtain a formula that is formed from literals using only $\wedge$ and $\vee$. For example:

$$\neg(\neg p \wedge (q \vee \neg(r \wedge s)))$$
$$\equiv (\neg\neg p \vee \neg(q \vee \neg(r \wedge s)))$$
$$\equiv p \vee (\neg q \vee \neg\neg(r \wedge s))$$
$$\equiv p \vee (\neg q \vee (r \wedge s))$$

3. Remove existential quantifiers.

To eliminate an independent existential quantifier, replace the variable with a Skolem constant. This process is called Skolemization.

Example: $\exists y$: President (y)

Here "y" is an independent quantifier so we can replace "y" by any name (say, George Bush). So, $\exists y$: President (y) becomes President (George Bush).

To eliminate a dependent the existential quantifier, we replace its variable by a Skolem function that accepts the value of "x" and returns the corresponding value of "y".

Example: $\forall x : \exists y$: father_of (x, y)

Here, "y" is dependent on "x," so we replace "y" with $S(x)$.

So, $\forall x : \exists y$: father_of (x, y) becomes $\forall x : \exists y$: father_of $(x, S(x))$.

The Skolem function, or Skolemization

- Replace each occurrence of its existentially quantified variable by a Skolem function whose arguments are those universally quantified variable function symbols.
- Create a Skolem function of no arguments.
- **Skolem form:** To eliminate all of the existentially quantified variables from a well-formed formula, the proceeding procedure on each sub-formula is used in turn. Eliminating the existential quantifiers from a set of well-formed formulas produces what is called the Skolem form of the set of formulas.
- The Skolem form of a well-formed formula is not equivalent to the original well-formed formula.

What is true is that in a set of formulas, Δ is satisfiable if, and only if, the Skolem form of Δ is. Or more usefully for purpose of resolution refutations, Δ is unsatisfiable if and only if the Skolem form of Δ is unsatisfiable.

4. Rename variables if necessary.

For sentences like $(\forall x P(x)) \vee (\exists x Q(x))$, which use the same variable name twice, change the name of one of the variables. This avoids confusion later when dropping quantifiers later.

For example, $\forall x[\exists y\ Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y\ Loves(y, x)]$ is renamed to $\forall x[\exists y\ Animal(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists z\ Loves\ (z, x)]$.

5. Move the universal quantifiers to the left.

Move the quantifiers outwards: repeatedly replace $P \wedge (\forall x Q(x))$ with $\forall x(P \wedge Q(x))$;

replace $P \vee (\forall x Q(x))$ with $\forall x(P \vee Q(x))$;

replace $P \vee (\exists x Q(x))$ with $\exists x(P \vee Q(x))$;

replace $P \vee (\exists x Q(x))$ with $\exists x(P \vee Q(x))$.

These replacements preserve equivalence, since the previous variable standardization step ensures that x doesn't occur in P. After these replacements, a quantifier may occur only in the initial prefix of the formula, but never inside a $\neg$, $\wedge$, or $\vee$.

Repeatedly replace $\forall x_1 \ldots x_n \exists y\ P(y)$ with $\forall x_1 \ldots \forall x_n\ P(f(x_1, \ldots, x_n))$, where f is a new n-ary function symbol, a Skolem function. This is the only step that preserves only the satisfiability rather than the equivalence. It eliminates all existential quantifiers.

6. Move the disjunctions down to the literals.

7. Eliminate the conjunctions.

$a \wedge b$ splits the entire clause into two separate clauses, i.e., a and b.

$(a \vee b) \wedge c$ splits the entire clause into two separate clauses $a \vee b$ and c.

$(a \wedge b) \vee c$ splits the clause into two clauses *i.e.* $a \vee c$ and $b \vee c$.

To eliminate "$\wedge$," break the clause into two; if you cannot break the clause, distribute the OR "$\vee$" and then break the clause.

8. Rename the variables, if necessary.

9. Purge or drop the universal quantifiers.

Consider the following example. Suppose we assert that "All music lovers who enjoy Bach either dislike Wagner or think that anyone who dislikes any composer is a philistine." We shall use enjoy() for enjoying a composer, and similarly for dislike.

"$\forall x$[musiclover(x)enjoy(x, Bach) $\Rightarrow$ dislike(x, Wagner) $\vee$ ("$\forall y$[$\exists z$[dislike(y, z)] $\Rightarrow$ think-philistine(x, y)])]

We now examine the recipe to reach the conjunctive normal form needed in the resolution. It is rather long, but not difficult to follow, and it should give confidence in handling expressions.

Step 1: Recall that *E F E F*, and thereby filter the expression to remove the symbols.

"$\forall x$[musiclover(x) enjoy(x, Bach) $\Rightarrow$ dislike(x, Wagner) $\vee$ "$\forall y$["$\forall z$[dislike(y, z)] $\Rightarrow$ think-philistine(x, y)])]

Step 2: Filter using the following relationships:

¬(¬P)		⇔	P;
¬(	$a \wedge$ b	)	$\Leftrightarrow \neg a \vee \neg b$;
¬(	$a \vee$ b	)	$\Leftrightarrow \neg a \wedge \neg b$;
$\neg\forall x$		$P(x) \Leftrightarrow \exists x\ \neg P(x)$;	and
$\neg\exists x\ P(x) \Leftrightarrow \forall x\ \neg P(x)$			

Our expression becomes

$\forall x$[¬musiclover(x) ∨ ¬ enjoy(x, Bach) ∨ dislike(x, Wagner) ∨ "$\forall y$["$\forall z$[¬dislike(y, z)] ∨ think- philistine (x, y)]].

Step 3: Standardize the variables so that each quantifier binds a unique variable. This is already the case in our expression, but the following is an example.

$\forall x$	Pred1(x) ∨ $\forall x$	Pred2(x)

becomes

$\forall x$ Pred1(x) ∨ $\forall y$ Pred2(y).

Step 4: Step 3 allows us to move all the quantifiers to the left in Step 4. Our expression becomes

$\forall$x$\forall y\forall z$[¬musiclover(x) ∨¬enjoy(x, Bach) ∨

dislike(x, Wagner) ∨¬dislike(y, z) think-philistine(x, y)].

This is called the **prenex normal form**. A well-formed formula in prenex form consists of a string of quantifiers called a prefix followed by a quantifier-free formula called a matrix.

For example:

$(\forall x)(\forall y)\{\neg P(x) \vee \{[\neg P(y) \vee P(f(x, y))] \wedge [Q(x, h(x)) \wedge \neg P(h(x))]\}\}$

Step 5: This step will seem a bit of a fiddle. We eliminate the existential quantifiers, by arguing that if $\exists y$ Composer(y) then if we could actually find an object S1 to replace the variable x. This gets replaced simply by Composer(S1).

Now, if existential quantifiers exist within the scope of the universal quantifiers, we can't merely use an object, but rather a function that returns an object. The function will depend on the universal quantifier.

$\forall x \exists y$ tutor-of(y, x)

gets replaced by

$\forall x$ tutor-of(S2(x), x).

This process is called Skolemization, and S2 is a Skolem function.

Step 6: This step is merely to save writing. Any variable left must be universally quantified out on the left, so don't bother writing the quantifier. Our expression becomes

¬musiclover(x) ∨ ¬enjoy(x, Bach) ∨ dislike(x, Wagner) ∨ ¬dislike(y, z) ∨ think-philistine(x, y).

Step 7: Convert everything into a conjunction of disjunctions using the associative, commutative, and distributive laws. The form you want is like

$(a \vee b \vee c \vee d \vee ...) \wedge (p \vee q \vee ...) \wedge ...$

Step 8: Call each conjunction a separate clause. In order for the entire well-formed formula to be true, each clause must be true separately.

Step 9: Standardize the variables in the set of clauses generated in Steps 7 and 8. This requires renaming the variables so that no two clauses make a reference to the same variable. Remember that all variables are implicitly universally quantified to the left.

$\forall x\ P(x) \wedge Q(x) \Leftrightarrow \forall x P(x) \wedge \forall x Q(x) \Leftrightarrow \forall x P(x) \wedge \forall y Q(y)$

This completes the recipe. After application to a set of well-formed formulas, we end up with a set of clauses, each of which is a disjunction of literals.

Resolution is based on the principle of proof by contradiction that we already discussed above in the example. First, if a clause is not in the conjunctive normal form, then convert it into that. After that, apply the resolution to the resulting clause. Now we choose those pairs that have complementary literals from a set of clauses to produce a new clause called the resolvent clause and add that new one to the knowledge base if it is not present already. This process continues until one of two things happens:

- There are no new clauses that can be added to the knowledge base.
- The resolution of two clauses results in an empty one. An empty clause (disjunction of no disjuncts) is considered to be false.

Resolution steps

- a conversion of axioms or formulas representing a sentence in canonical or clause form
- use refutation, which means to show that the negation of the statement produces a contradiction with the known statement (which is a fact).

If clauses are resolved in a systematic way, then if a contradiction exists, the resolution is guaranteed to find the contradiction.

8.7.1.5 Resolution in Proposition Logic

Suppose a set of axioms (propositions) is given. Convert all propositions of this set to clause form. The algorithm for propositional resolution

- converts all propositions to clause form
- negates P and converts the result to clause form; it adds it to the set of clauses
- repeats until either a contradiction is found or no progress is possible
  - ♦ Takes any two clauses as parent clauses
  - ♦ Resolves these two clauses. The resulting clause is called the resolvent.
  - ♦ If the resolvent is an empty clause, then a contradiction is found. If not, then add the resolvent to the set of clauses.

Example 1: Suppose the following propositions are given and we have to prove R.

Now for proving R, we have to prove it by contradiction, and we start with ¬R.

Given axioms (Proposition)	Clause form	No
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
Separate (3) in CF	$\neg T \vee Q$	(4)
T	T	(5)

Now for proving R, we have to prove it by contradiction, and we start with $\neg R$.

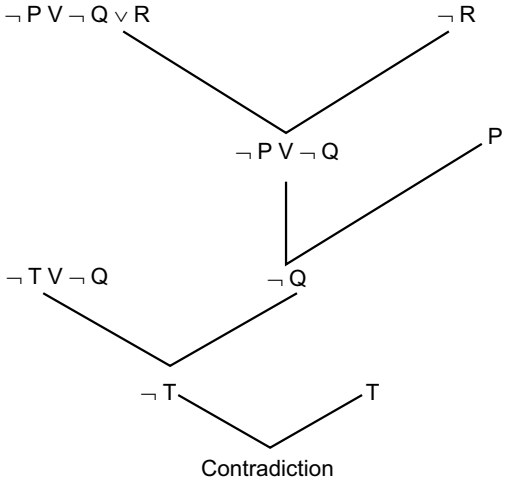


FIGURE 8.3 The Resolution of Clauses

We reach the contradiction, as the resolvent clause is now empty, so process is stopped here. We started with the negation R and after applying the resolution, we found our assumption is wrong because we got an empty clause at the end. R is true as $\neg R$ is false.

Example 2: Let's say I'm given "P or Q," "P implies R," and "Q implies R." I would like to conclude R from these three axioms.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	**Formula**	**Derivation**

Step 1: We start by converting this first sentence into the conjunctive normal form. We don't actually have to do anything. It's already in the right form.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given

Step 2: Now, "*P* implies *R*" turns into "not *P* or *R*."

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given

Step 3: Similarly, "*Q* implies *R*" turns into "not *Q* or *R*."

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given

Step 4: Now we want to add one more thing to our list of given statements. What's it going to be? Not *R*. Right? We're going to assert the negation of the thing we're trying to prove. We'd like to prove that *R* follows from these things. But what we're going to do instead is say not *R*, and now we're trying

to prove it false. If we manage to prove it false, then we will have a proof that R is entailed by the assumptions.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion

Step 5: We'll draw a blue line just to divide the assumptions from the proof steps. And now, we look for opportunities to apply the resolution rule.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion

Step 6: We can apply the resolution to lines 1 and 2, and get "Q or R" by resolving away P.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1, 2

Step 7: We can take lines 2 and 4, resolve away *R*, and get "not *P*."

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1, 2
6	$\neg P$	2, 4

Step 8: Similarly, we can take lines 3 and 4, resolve away *R*, and get "not *Q*."

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1, 2
6	$\neg P$	2, 4
7	$\neg Q$	3, 4

Step 9: By resolving away Q in lines 5 and 7, we get R.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1, 2
6	$\neg P$	2, 4
7	$\neg Q$	3, 4
8	R	5, 7

Step 10: Finally, resolving away R in lines 4 and 8, we get the empty clause, which is false. We'll often draw this little black box to indicate that we've reached the desired contradiction.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1, 2
6	$\neg P$	2, 4
7	$\neg Q$	3, 4
8	R	5, 7
9	*	4, 8

Step 11: How did I do this last resolution? Let's see how the resolution rule is applied to lines 4 and 8. The way to look at it is that *R* is really "false or *R*", and that "not *R*" is really "not *R* or false." (Of course, the order of the disjuncts is irrelevant, because the disjunction is commutative). So, now we resolve away *R*, getting "false or false," which is false.

Propositional Resolution Example

Prove *R*	
1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

False $\vee$ R

$$\frac{\neg R \vee false}{false \vee false}$$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1, 2
6	$\neg P$	2, 4
7	$\neg Q$	3, 4
8	R	5, 7
9	*	4, 8

8.7.1.6 Resolution Algorithm in Predicate Logic

- Convert all the propositions (axioms) of F to clause form.
- Negate P (conclusion) and convert the result to clause form. Add it to the set of clauses obtained in 1.
- Repeat until either a contradiction is found, no progress can be made, or a predetermined amount of effort has been expended.
  - ♦ Select two clauses. Call these the parent clauses.
  - ♦ Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with the appropriate substitutions performed and with the following exception: If there is one pair of literals $T1$ and $\neg\ T2$ such that one of the parent clauses contains $T1$ and the other contains $\neg\ T2$ and if $T1$ and $T2$ are unifiable, then neither $T1$ nor $\neg\ T2$ should appear in the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
  - ♦ If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Let us discuss some examples of resolutions in predicate logic.

Example 1: Consider the statement: "Every rich person owns a house. Susan is rich. Susan is a person. Therefore, Susan owns a house." So "Susan own a house" is a conclusion. Let us prove that Susan owns a house. We prove it by using a resolution algorithm.

1. First, convert the above sentence into predicate calculus.

$\forall x\ [(\text{person}(x) \wedge \text{rich}(x)) \rightarrow \exists y(\text{house}(y) \wedge \text{owns}(x, y))]$.

rich(Susan).

person(Susan).

The conclusion:

$\exists z(\text{house}(z) \wedge \text{owns}(\text{Susan}, z))$.

Here, x, y, and z are variables.

2. Negate the conclusion.

This becomes: $\neg\exists z(\text{house}(z) \wedge \text{owns}(\text{Susan}, z))$.

3. Since above predicate form is not in CNF, before beginning resolution, all clauses have to be converted into CNF.

a) Eliminate implications, using the logical equivalence that $a \rightarrow b \leftrightarrow \neg a \vee b$.

So the first statement becomes

$\forall x\ [\neg(\text{person}(x) \wedge \text{rich}(x)) \vee \exists y(\text{house}(y) \wedge \text{owns}(x, y))]$.

b) Move the negations inwards (i.e., ensure that no lines, or groups of terms, begin with ¬). Use suitable logical equivalences such as

$\neg(\neg a) \leftrightarrow a$, $\neg(a \vee b) \leftrightarrow \neg a \wedge \neg b$, $\neg(a \wedge b) \leftrightarrow \neg a \vee \neg b$, $\neg\forall x\ P(x) \leftrightarrow \exists x\ \neg P(x)$, $\neg\exists x\ P(x) \leftrightarrow \forall x\ \neg P(x)$. So the first statement becomes $\forall x\ [(\neg\text{person}(x) \vee \neg\text{rich}(x)) \vee \exists y(\text{house}(y) \wedge \text{owns}(x, y))]$. The conclusion becomes: $\forall z\ \neg(\text{house}(z) \wedge \text{owns}(\text{Susan}, z))$ then $\forall z\ \neg\text{house}(z) \vee \neg\text{owns}(\text{Susan}, z))$ becomes $\forall z\ \neg\text{house}(z) \vee \neg\text{owns}(\text{Susan}, z))$.

c) Standardize the variables so that different quantifiers refer to different variables.

d) Eliminate all existential quantifiers ("Skolemization"). This is done by substituting a different predicate name that is unique to the object in question, but which relates to the universally-quantified class in which it is found, rather than labelling it as an instance of a class of objects.

So the first statement becomes $\forall x\ [(\neg\text{person}(x) \vee \neg\text{rich}(x)) \vee (\text{house}(G(x)) \wedge \text{owns}(x, G(x))]$.

e) Eliminate all universal quantifiers by assuming that all variables are universally quantified.

The first statement becomes

$(\neg\text{person}(x) \vee \neg\text{rich}(x)) \vee (\text{house}(G(x)) \wedge \text{owns}(x, G(x))$.

The conclusion becomes $\neg\text{house}(z) \wedge \neg\text{owns}(\text{Susan}, z)$.

f) Rewrite in the conjunctive normal form. This means groups of terms joined by "and," the groups themselves being terms joined by "or." Use the logical equivalence that

$a \vee (b \wedge c) \leftrightarrow (a \vee b) \wedge (a \vee c)$.

The first statement becomes

$(\neg\text{person}(x) \vee \neg\text{rich}(x) \vee \text{house}\ (G(x))) \wedge (\neg\text{person}(x) \vee \neg\text{rich}(x) \vee \text{owns}(x, G(x)))$.

g) For statements produced as a result of (f), since the groups are joined by "and," they can become separate statements in their own right.

The first statement becomes

$\neg\text{person}(x) \vee \neg\text{rich}(x) \vee \text{house}(G(x))$

$\neg\text{person}(x) \vee \neg\text{rich}(x) \vee \text{owns}(x, G(x))$.

h) Change the variable names, so that each clause uses different variables.

We finish up with 5 clauses like this:

clause 1: $\neg\text{person}(x) \vee \neg\text{rich}(x) \vee \text{house}(G(x))$

clause 2: $\neg\text{person}(y) \vee \neg\text{rich}(y) \vee \text{owns}(y, G(y))$

clause 3: rich(Susan).

clause 4: person(Susan).

clause 5: $\neg\text{house}(z) \vee \neg\text{owns}(\text{Susan}, z)$).

4. Now all clauses are in CNF, so we apply the resolution algorithm now and pick up two clauses having complementary literals, and these two clauses are called parent clauses and resolved to give a third clause. If the resulting clause is empty, then the proof succeeded; otherwise, add the resulting clause to the set of clauses and repeat the same process.

Resolving clauses: Pick 2 clauses that contain the same term, negated in one case and not negated in the other.

Combine them to form a new clause, containing all the terms that were in both the old ones, except that the term that is present as a and ¬a is eliminated; however, if in one case it contains an argument (or arguments) that is a variable and in the other case a constant, substitute the constant for the variable everywhere so that the constant appears in the clause.

Empty clause: This is the result of resolving two clauses where each only contained one term, so that nothing remains.

In our example, the process is as follows:

Resolving clauses 1 and 3 gives

¬person(Susan) ∨ house(*G*(Susan)).

Add this to the clauses as number 6.

Resolve 6 and 4 to give

House(*G*(Susan)).

Add this to the clauses as number 7.

Resolve 2 and 3 to give

¬person(Susan) ∨ owns(Susan, G(Susan)).

Add this as number 8.

Resolve 8 and 4 to give

owns (Susan, *G*(Susan)).

Add this as number 9.

Resolve 7 and 5 to give

¬owns(Susan, *G*(Susan)).

Add this as number 10.

Resolve 10 and 9.

This gives an empty clause, so the proof has succeeded.

Example 2: Problem Statement:

1. Ravi likes all kind of food.
2. Apples and chicken are food.
3. Anything anyone eats and is not killed by is food.
4. Ajay eats peanuts and is still alive.
5. Rita eats everything that Ajay eats.

Prove by resolution that Ravi likes peanuts using resolution. Convert the given statements into predicate/propositional logic.

Solution:

Step 1: Convert the given statements into predicate/propositional logic.

(*i*) $\forall x$: food(x) $\rightarrow$ likes (Ravi, x)

(*ii*) food (Apple) $\wedge$ food (chicken)

(*iii*) $\forall a$: $\forall b$: eats (a, b) $\wedge$ killed (a) $\rightarrow$ food (b)

(*iv*) $\forall$eats (Ajay, peanuts) $\wedge$ alive (Ajay)

(*v*) $\forall c$: eats (Ajay, c) $\rightarrow$ eats (Rita, c)

(*vi*) $\forall d$: alive(d) $\rightarrow$ $\neg$ killed (d)

(*vii*) $\forall e$: $\neg$killed(e) $\rightarrow$ alive(e)

Conclusion: likes (Ravi, peanuts)

Step 2: Convert into CNF.

(*i*) $\neg$food(x) $\vee$ likes (Ravi, x)

(*ii*) Food (apple)

(*iii*) Food (chicken)

(*iv*) $\neg$ eats (a, b) $\vee$ killed (a) $\vee$ food (b)

(*v*) Eats (Ajay, peanuts)

(*vi*) Alive (Ajay)

(*vii*) $\neg$eats (Ajay, c) $\vee$ eats (Rita, c)

(*viii*) $\neg$alive (d) $\vee$ $\neg$ killed (d)

(*ix*) Killed (e) $\vee$ alive (e)

Conclusion: likes (Ravi, peanuts)

Step 3: Negate the conclusion.

$\neg$ likes (Ravi, peanuts)

Step 4: Resolve using a resolution tree.

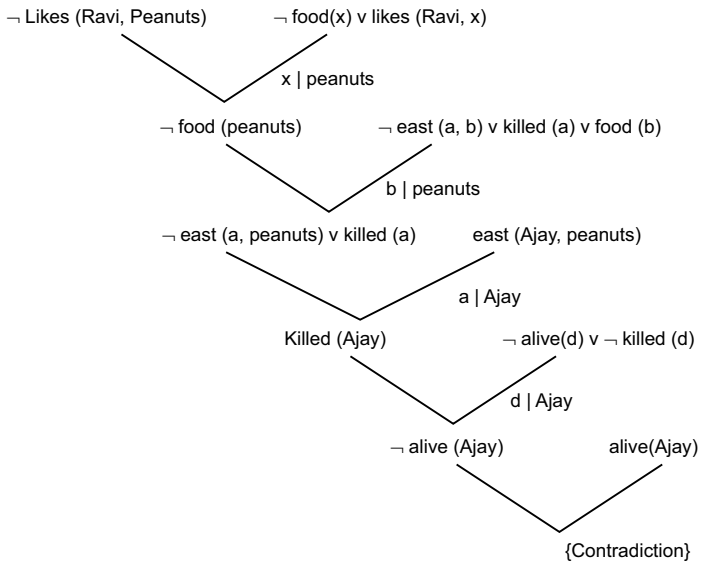


FIGURE 8.4 The Resolution Tree

Hence, we see that the negation of the conclusion has been proved as a complete contradiction with the given set of facts.

Hence, the negation is completely invalid or false, or the assertion is completely valid or true.

The Explanation of the Resolution Tree

- In the first step of the resolution tree, ¬ likes (Ravi, peanuts) and likes (Ravi, *x*) getresolved (cancelled). So, we are only left with ~food (peanuts). In this, "*x*" is replaced by peanuts, i.e., "*x*" is bound to peanuts.
- In the second step of the resolution tree, ¬food(peanuts) and food (*b*) get resolved, so we are left with ¬eats (a, peanuts) ∨ killed(*a*). In this, "*b*" is bound to peanuts, thus, we replace every instance of "*b*" by peanuts in that particular clause. Thus, now we are left with eats(a, peanuts) ∨ killed (*a*).
- In the third step of the resolution tree, ¬eats (*a*, peanuts) and eats (Ajay, peanuts) gets resolved. In this "*a*" is bound to "Ajay." So, we replace every instance of "*a*" by "Ajay." Thus, we are now left with killed (Ajay).

- In the fourth step of the resolution tree, killed (Ajay) and ¬killed (d) get resolved. In this, "d" is bound to "Ajay," thus every instance of "d" is replaced by "Ajay." Now we are left with ¬alive(Ajay).
- In the fifth step of the resolution tree, ¬Alive(Ajay) and Alive(Ajay) get resolved, and we are only left with a null set.

Example 3. Suppose following axioms in clause form are given:

1. man(Marcus)
2. Pompeian(Marcus)
3. ¬ *Pompeian*($x1$) ∨ Roman($x1$)
4. Ruler(Caesar)
5. ¬ Roman($x2$) ∨ loyalto ($x2$, Caesar) ∨ hate($x2$, Caesar)
6. loyalto ($x3$, $f1$($x3$))
7. ¬ man($x4$) ∨ ¬ ruler($y1$) ∨ ¬ tryassassinate ($x4$, $y1$) ∨ ¬ loyalto ($x4$, $y1$)
8. tryassassinate (Marcus, Caesar)

We have to prove that Marcus hated Caesar.

Variables in 3, 5, 6, and 7 ($x1$, $x2$, $x3$, and $x4$ y, respectively) have been used to discriminate them from each other.

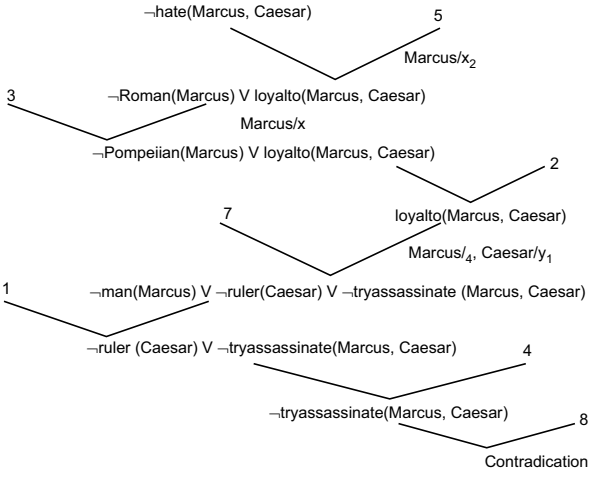


FIGURE 8.5 A Proof by Resolution: Hate (Marcus, Caesar)

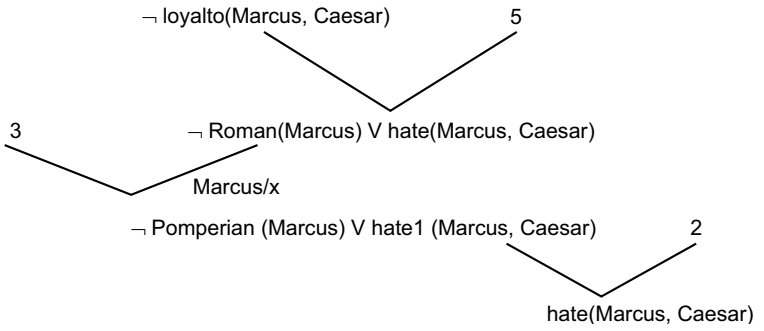


FIGURE 8.6 The Unsuccessful Attempt at Resolution of Marcus is Loyal to Caesar: loyalto(Marcus, Caesar)

Uses of Resolution in Today's World

- helps in the development of computer programs to automate reasoning and theorem proving
- used widely in AI
- helps in forward and backward reasoning
- resolution is a proof by contradiction, which is even used in math problems

8.7.1.7 Unification

Unification is a technique for taking two sentences in predicate logic and finding a substitution that makes them look the same.

- A variable can be replaced by a constant.
- A variable can be replaced by another variable.
- A variable can be replaced with a predicate, as long as the predicate does not contain that variable.

Let us understand unification through examples:

- We know that dog(Boxer) and ¬ dog(Boxer) is a contradiction, as both cannot be true at the same time. However, dog(Boxer) and ¬dog(Jackie) is not a contradiction. To check a contradiction, there must be some procedure to match literals and the possibility to make them identical. This recursive procedure is called the unification algorithm.

- We know that classmates(Ram, Ramesh) and beats(Ram, Ramesh) cannot be unified, as both have different initial predicate symbols ("classmates" and "beats," which differ). If both predicate symbols match, then we can only use the unification procedure.

Some simple examples are as follows:

- Unify $Q(x)$ and $P(x)$ fails as literals are different and cannot be unified
- Unify $Q(x)$ and $Q(x)$ nil as literals are identical so no there is no scope of unification
- Unify $P(x)$ and $P(x, y)$ fails as both literals have a different number of arguments
- Unify $P(x, x)$ and $P(y, z)$

Here, both initial predicate symbols are identical, *P,* so we check number of arguments, which is also same. This means we can apply the unification procedure to that. Substitute y/x to get $P(y, y)$ and $P(y, z)$ then take z/y, which produces $P(z, z)$, thus $(z/y)(y/x)$ is the total substitution applied to unify the two literals. Avoid a substitution like $(x/y)(x/z)$, as they cause inconsistency.

Given the following set of predicates, let's explore how they can be unified:

1. Hates (*X*, *Y*)
2. Hates (George, broccoli)
3. Hates (Alex, spinach)

We could unify sentence 2 with 1 by binding George to variable *X*, and broccoli to variable *Y*. Similarly, we could bind Alex to *X* and spinach to *Y*. Note that if the predicate names were different, we could not unify these predicates.

If we introduce a few more predicates, we can explore more complex unifications:

4. hates(*X*, vegetable(*Y*))
5. hates(George, vegetable(*Y*))
6. hates(Z, broccoli)

We could unify sentence 6 with sentence 1 by replacing variable *X* with variable *Z* and variable *Y* with the constant broccoli. Sentences 4 and 5 could be unified with George bound to *X*, and broccoli to variable *Y*.

Unification Algorithm

Unify(*L*1, *L*2) // unifies two literals, *L*1 and *L*2.

The steps are

1. If *L*1 or *L*2 is a variable or constant, then
   - **a)** If *L*1 and *L*2 are identical, then return NIL.
   - **b)** Else if *L*1 is a variable, then if *L*1 occurs in *L*2, then return FAIL, else return {(*L*2/*L*1)}.
   - **c)** Else if *L*2 is a variable, then if *L*2 occurs in *L*1, then return FAIL, else return {(*L*1/*L*2)}.
   - **d)** Else return FAIL.
2. If the initial predicate symbols in *L*1 and *L*2 are not identical, then return FAIL.
3. If *L*1 and *L*2 have a different number of arguments, then return FAIL.
4. Set *SUBST* to NIL.
5. For *i* 1 to a number of arguments in L1,
   - **a)** Call Unify with the *i*th argument of *L*1 and the *i*th argument of *L*2, putting the result in *S*.
   - **b)** If *S* = FAIL then return FAIL.
   - **c)** If *S* is not equal to NIL then

   Apply *S* to the remainder of both *L*1 and *L*2.

   SUBST: = APPEND(*S*, SUBST).
6. Return *SUBST*.

Another example is that hate(*x*, *y*) and hate (Marcus, *z*) can be unified using (Marcus/*x*, *z*/*y*) or (Marcus/*x*, *y*/*z*):

- Unify(Knows (John, *x*), Knows (John, Jane)) = {*x*/Jane}
- Unify(Knows (John, *x*), Knows(*y*, Bill)) = {*x*/Bill, *y*/John}

- Unify(Knows(John, x), Knows(y, Mother(y))) = {y/John, x/Mother (John)}
- Unify (Knows (John, x), Knows(x, Eliza)) = fail

The last unification fails because x cannot take on the values "John" and "Eliza" simultaneously.

Because the variables are universally quantified, Knows(x, Eliza) means that everyone knows Eliza. In that sense, we should be able to infer that John knows Eliza.

Important Properties of Logical Systems

- **Consistency:** no theorem of the system contradicts another
- **Soundness:** The system's rules of proof will never allow a false inference from a true premise. If a system is sound and its axioms are true, then its theorems are also guaranteed to be true.
- **Completeness:** There are no true sentences in the system that cannot, at least in principle, be proved in the system.

Some logical systems do not have all three properties. Kurt Godel's incompleteness theorems show that no standard formal system of arithmetic can be consistent and complete.

8.7.2 Semantic Net

Quillian devised the semantic net in 1968 as a model of the human memory. This technique offered the possibility that computers might be able to use words like humans did, following the failure of early machine translators. The various definitions of semantic nets are

- It is one of the network representational schemes that use hierarchical representation for knowledge. A semantic net is a graphical representation of knowledge. Semantic nets are used to define the meaning of a concept by its relationships to other concepts. A graph data structure is used, with nodes used to hold concepts and links with natural language labels used to show the relationships.
- A semantic network is often used as a form of knowledge representation by connecting concepts together. It is a directed graph consisting of vertices that represent concepts and edges that represent the semantic relations between the concepts. In a semantic net,

knowledge is represented as a collection of concepts, represented by nodes and connected together by relationships, represented by arcs.

- A semantic network or net is a graphic notation for representing knowledge in patterns of interconnected nodes and arcs.
- Semantic networks are systems specially designed for organizing and reasoning with categories.
- They provide
  - ♦ graphical aids for visualizing a knowledge base
  - ♦ efficient algorithms for inferring the properties of an object on the basis of its category membership

A semantic network (Quillian, 1967) is a knowledge representation schema that captures knowledge as a **graph**. The nodes denote **objects** or concepts, their **properties,** and corresponding **values**. The arcs denote **relationships** between the nodes. Both nodes and arcs are generally **labelled** (arcs have weights).

- A semantic net has a binary relation.
- Concepts are represented by nodes.
- Links between nodes represent the relationships.
- Examples of relationship labeled on arcs (notice that there is an underscore) are as follows:
  - ♦ is_a
  - ♦ has_a
  - ♦ has_part

Examples of concepts (nodes)

- bird
- person
- book
- famous
- intelligent

Let us discuss some examples and the semantic net in detail.

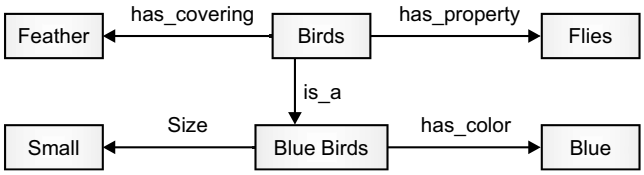


FIGURE 8.7 The Semantic Net Representing a Bird's Properties

Question: Consider the below sentences and draw a semantic network for them.

Lab is a room. Lab has a door. Lab has many computers. Printer is in lab. Laser printer is a Printer.

Answer:

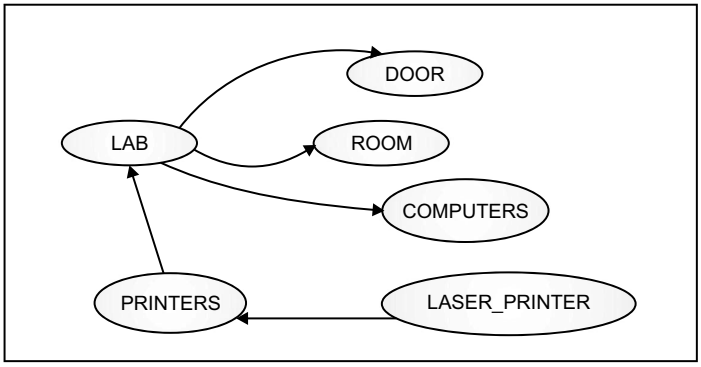


FIGURE 8.8 The Sematic Network for the Above Sentences

- Semantic nets are useful for representing inheritable knowledge. Inheritable knowledge is the most useful for property inheritance, in which elements of specific classes inherit attributes and values from more general classes in which they are included. In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy. Inheritance provides the cognitive economy, but there is a storage-space/processing-time trade-off.
- This means that, if you adopt this technique, you will use less storage space than if you don't, but your system will take longer to find the answers to questions.

Example 1

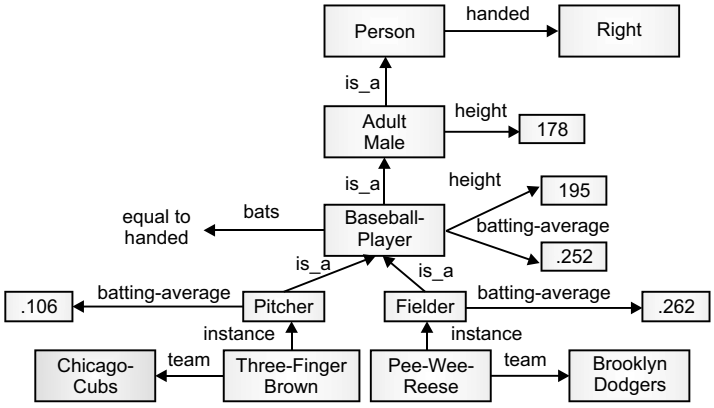


FIGURE 8.9 Baseball Knowledge Represented through the Semantic Net

Inheritance is a process by which the local information of a superclass node is assumed by a class node, a subclass node, and an instance node. The advantages of inheritance are as follows:

- It provides a natural tool for representing taxonomically structured knowledge.
- It provides an economical means of expressing properties common to a class of objects.
- It reduces the size of a knowledge base.
- It provides more compact code.

In these figures, nodes represent concepts (for example, person or right), and each concept may be an object or class (collection of objects). Each class has some properties and also a relation "is-a" showing inheritance, that is, one class deriving features and properties from other class.

- **Lines represent attributes:** Boxed nodes represent objects and values of attributes of objects. For example, the Adult Male class is derived from the Person class, and the Pitcher class is derived from Baseball Player.
- The arc labelled with the instance showing the object of a class that is Three-Finger-Brown is an object or instance of the Pitcher class.
- Other than instance and the is-a labelled arc, the rest of the arcs are labelled with the properties of the class. For example, the Adult Male object has a property height whose value is 195.

The correct deduction from Figure 8.9 could be that the height of Three-Finger Brown is 195 cm. An incorrect deduction would be that the height of Three-Finger Brown is 178 cm.

- Specific reasoning mechanisms can be established that allow us to answer questions about the representation:
  - ♦ Are two concepts related?
  - ♦ What relates two concepts?
  - ♦ Which is the closest concept that relates two other concepts?
- If richer semantics are defined for relations, more complex questions can be answered about
  - ♦ the taxonomy among the concepts (class and subclass and instances)
  - ♦ generalization/specialization

A semantic net should make a distinction between types and tokens. This is why the diagram above uses "instance" arcs as well as "is-a" arcs. Individual instances of objects have a token node. Categories of objects have a type node. There is always at least one type node above a token node. The information needed to define an item is (normally) is found attached to the type nodes above it.

- **Example 2:**

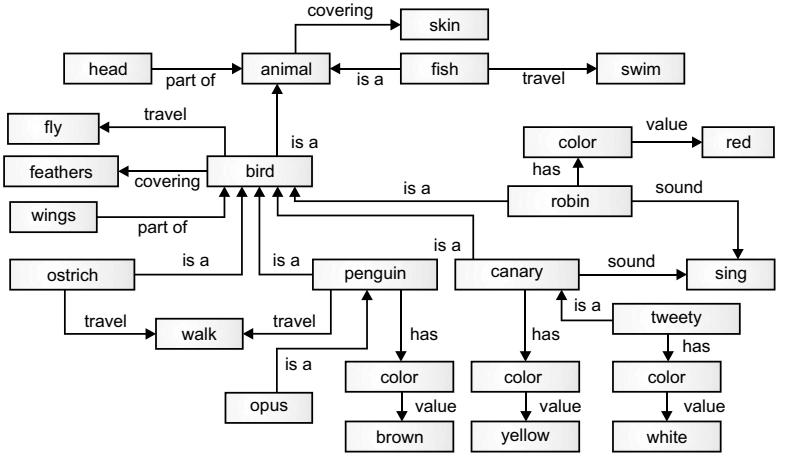


FIGURE 8.10 Semantic Net with Various Relationships

- **Example 3:**

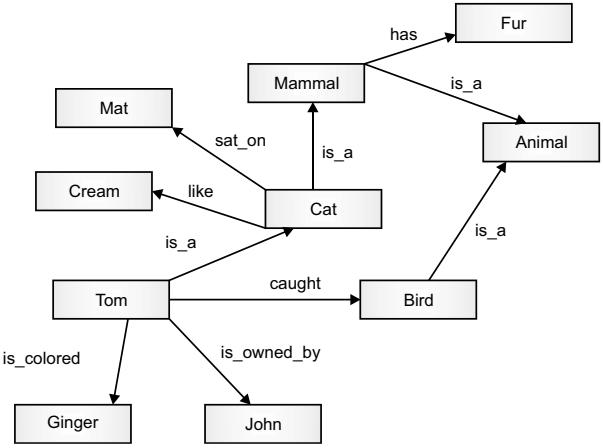


FIGURE 8.11 The Relationships in a Semantic Net

FIGURE 8.11 represents the following data:

Tom is a cat.

Tom caught a bird.

Tom is owned by John.

Tom is ginger in color.

Cats like cream.

The cat sat on the mat.

A cat is a mammal.

A bird is an animal.

All mammals are animals.

Mammals have fur.

It is argued that this form of representation is closer to the way a human structures knowledge by building mental links between things rather than the predicate logic we studied earlier. Note in particular how all the information about a particular object is concentrated on the node representing that object, rather than being scattered around several clauses in the logic.

There is, however, some confusion here that comes from the imprecise nature of semantic nets. A particular problem is that we haven't distinguished

between the nodes representing classes of things, and the nodes representing individual objects. So, for example, the node labelled "Cat" represents both the single (nameless) cat who sat on the mat, and the whole class of cats to which "Tom" belongs, which are mammals and which like cream. The is_a link has two different meanings – it can mean that one object is an individual item from a class, for example, Tom is a member of the class of cats, or that one class is a subset of another, for example, the class of cats is a subset of the class of mammals. This confusion does not occur in logic, where the use of quantifiers, names and predicates makes it clear what we mean, so

"Tom is a cat" is represented by Cat(Tom).

"The cat sat on the mat" is represented by $\exists x \exists y$ (Cat(x) $\wedge$ Mat(y) $\wedge$ sat_on(x, y)).

"A cat is a mammal" is represented by $\forall x$ (Cat(X) $\rightarrow$ Mammal(x)).

We can clean up the representation by distinguishing between the nodes representing the individuals or instances, and nodes representing classes. The is_a link will only be used to show an individual belonging to a class. The link representing one class being a subset of another will be labelled a_kind_of, or ako for short. The names instance and subclass are often used in the place of is_a and ako, but we will use these terms with a slightly different meaning in the coming section on frames.

Various types of relationships in a sematic net are (depicted by arc)

- IS-A
- PART-OF
- HAS
- VALUE
- LINGUISTIC

IS-A

Supertype – type (superclass – class)

Type – subtype (class – subclass)

Subtype – instance (subclass – instance)

PART-OF

Supertype – type (superclass – class)

Type – subtype (class –subclass)

HAS

Object – property

VALUE

Property –value

LINGUISTIC

Examples: likes, owns, travel, made of

8.7.2.1 Inheritance in Semantic Nets

Semantic networks can show and capture inheritance. Inheritance is a process by which the local information of a superclass node is assumed by a class node, a subclass node, and an instance node. The idea of this is that if an object belongs to a class (indicated by an `is_a` link), it inherits all the properties of that class. So, for example, as we have a `likes` link between cats and cream, meaning "all cats like cream," we can infer that any object that has an is_a link to cats will like cream. So, both `Tom` and `Cat1` like cream. However, the `is_colored` link is between Tom and ginger, not between cats and ginger, indicating that being ginger is a property of Tom as an individual, and not of all cats. We cannot say that `Cat1` is ginger; for example, if we wanted to, we would have to put another `is_colored` link between `Cat1` and ginger.

Inheritance also applies across the `a_kind_of` links. For example, any property of mammals or animals will automatically be a property of cats. So, we can infer, for example, that Tom has fur, since Tom is a cat, a cat is a kind of mammal, and mammals have fur. If, for example, we had another subclass of mammals, say dogs, and we had, say, Fido `is_a` dog, Fido would inherit the property "has fur" from mammals, but not the property "likes cream," which is specific to cats. This situation is shown in the diagram below:

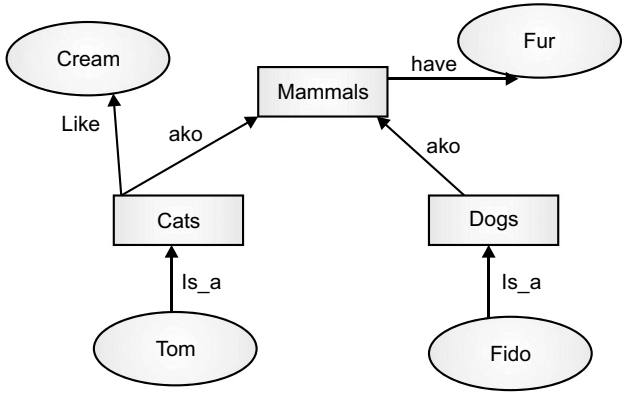


FIGURE 8.12 Inheritance in a Semantic Net

Let us discuss another example of inheritance.

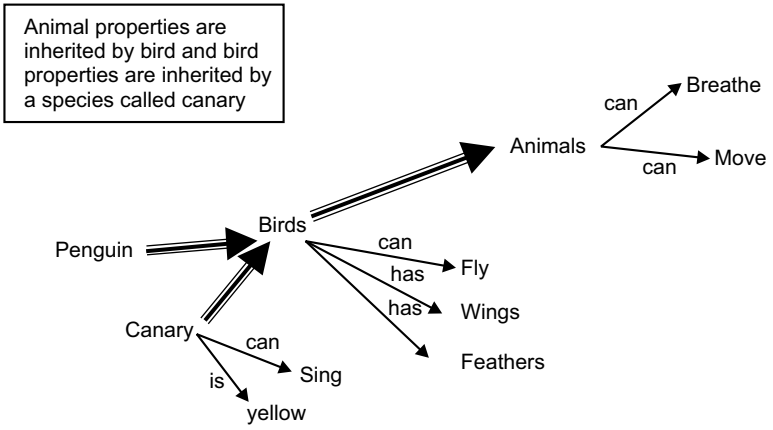


FIGURE 8.13 Inheritance in a Semantic Net

Sometimes, inheritance may cause problems. "Penguin" through inheritance gets the property "fly" (in practice, it cannot).

To avoid this situation, all the specific properties of a node must be attached to it through local nodes, so that when an answer is needed, it will search all the local nodes first. If the answer is not available in the local nodes, then the general nodes will be used.

For example, if we ask "How does a penguin travel?," the reply will be "it walks" (supposedly, that is already stored in the local node).

Example 1:

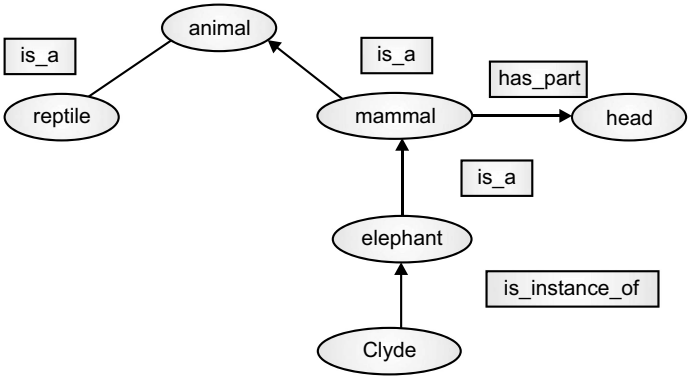


FIGURE 8.14 A Semantic Net Depicting Inheritance and Properties

Example 2:

"Is-a" shows a subset relation and the instance shows the membership relation.

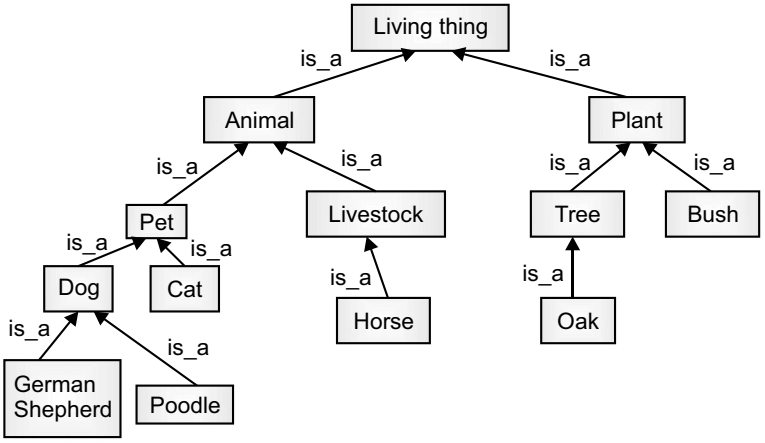


FIGURE 8.15 The "Is-A" Hierarchy

Example 3:

We can take the hierarchy all the way down to the atomic level with the "is part" hierarchy.

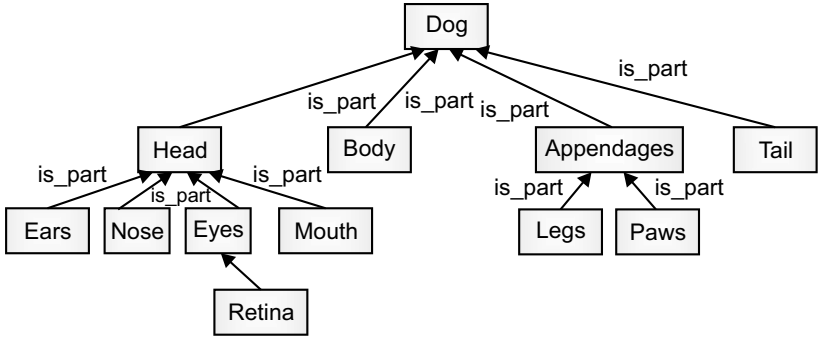


FIGURE 8.16 The "Is Part" Hierarchy

Example 4:

Semantic networks are very good at representing simple events and declarative sentences by basing them around the "event node."

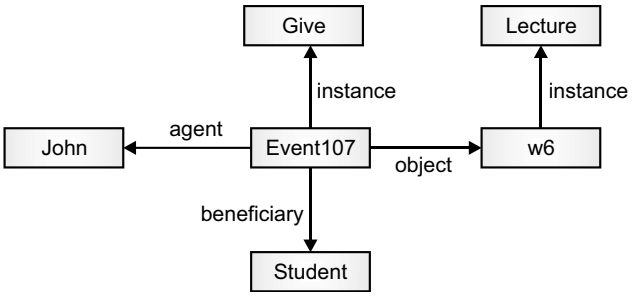


FIGURE 8.17 A Semantic Net Representing Events and Declarative Sentences

Example 5:

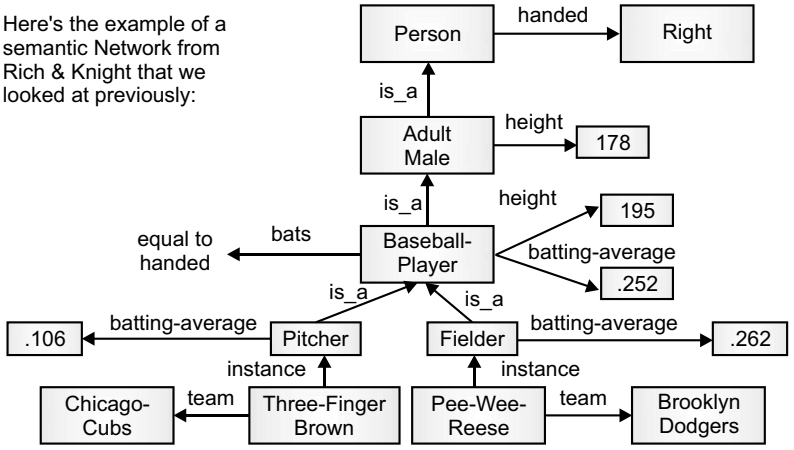


FIGURE 8.18 A Mixed Semantic Network

8.7.2.2 Inference in the Semantic Net

There are two ways of using an inference mechanism in a semantic net.

Intersection search: This is the earliest way that a semantic network can use to find a relationship between objects, which is by spreading the activation out of two nodes and finding their intersection; it finds relationships among objects. This is achieved by assigning a special tag to each visited node.

This search has many advantages, including entity-based organization and fast parallel implementation. However, very structured questions need highly structured networks.

Let us discuss some examples.

1. **Question:** "What is the relation between the 1950 Chicago Cubs and Brooklyn Dodgers?"
   Answer: "They are teams of baseball players."

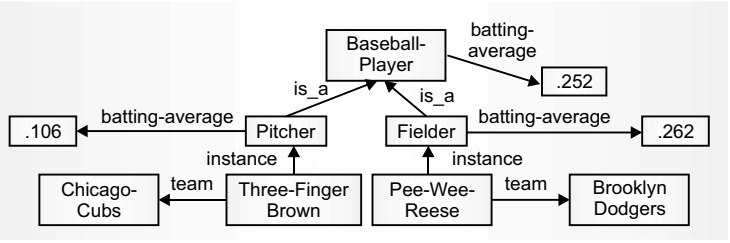


FIGURE 8.19 The Intersection Search in the Semantic Net

2. **Question:** "What is the relation between Liverpool and red?"
   Answer: "Liverpool is a team name whose uniform color is red."

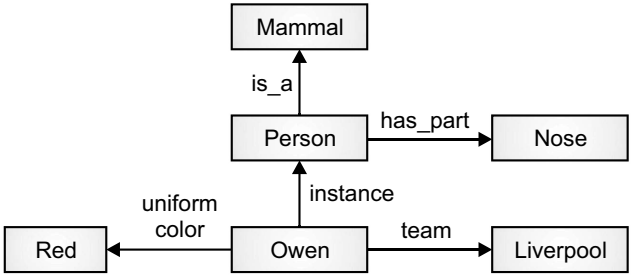


FIGURE 8.20

Inheritance: The concept of inheritance was discussed above. The inheritance mechanism allows knowledge to be stored at the highest possible level of abstraction, which reduces the size of the knowledge base.

- It facilitates inferencing of the information associated with semantic nets.
- It is a natural tool for representing taxonomically structured information and ensures that all the members and sub-concepts of a concept share common properties.
- It also helps us to maintain the consistency of the knowledge base by adding new concepts and members of existing ones.

- Properties attached to a particular object (class) are inherited by all subclasses and members of that class.

The is-a and instance representation provide a mechanism to implement this. Inheritance also provides a means of dealing with default reasoning.

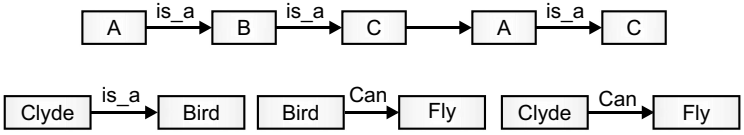


FIGURE 8.21 Inheritance

For example, we could represent the following sentences as a semantic net:

- Emus are birds.
- Typically, birds fly and have wings.
- Emus run.

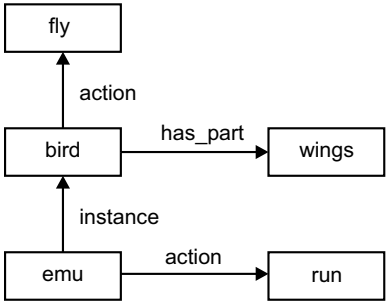


FIGURE 8.22 Default Reasoning within Inheritance through a Semantic Net

Two important features of the semantic net are the default values of the attributes and inheritance. Consider the following semantic net.

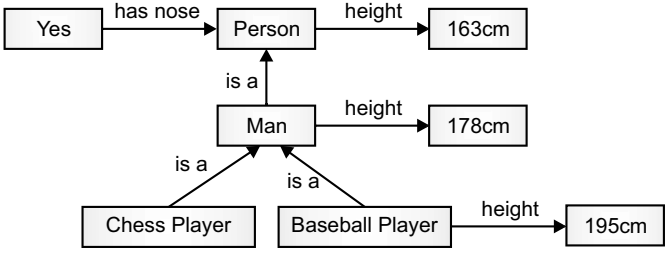


FIGURE 8.23 A Semantic Net with Default Values

We can assign the expected default values of parameters (e.g., height, has nose) and inherit them from higher up in the hierarchy. This is more efficient than listing all the details at each level. We can also override the defaults. For example, baseball players are taller than average, so their default height overrides the default height for men.

8.7.2.3 Multiple Inheritances in a Semantic Net

With simple trees, inheritance is straightforward. However, when multiple inheritances are allowed, problems can occur. For example, consider this famous example.

Question: "Is Nixon a pacifist?"

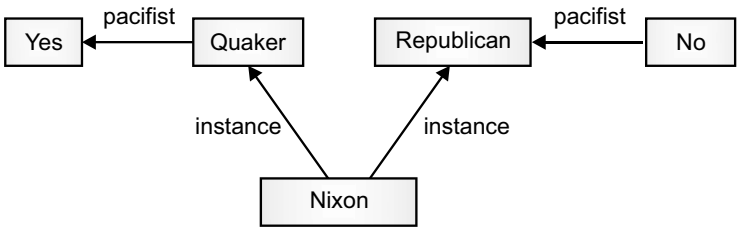


FIGURE 8.24 Multiple Inheritance

Conflicts like this are common in the real world. It is important that the inheritance algorithm reports the conflict, rather than just traversing the tree and reporting the first answer it finds. In practice, we aim to build semantic networks in which all such conflicts are either overridden, or resolved appropriately.

Advantages of Semantic Nets

- Easy to visualize and understand
- The knowledge engineer can arbitrarily define the relationships.
- Related knowledge is easily categorized.
- Efficient in terms of space requirements
- Node objects are represented only once.
- Standard definitions of semantic networks have been developed.

Limitations of Semantic Nets

- Different formalisms exist, with different capabilities.
- There is no standard reasoning model.
- Many believe that the basic notion is a powerful one and has to be complemented by, for example, logic, to improve the notion's expressive power and robustness.
- Others believe that the notion of semantic networks can be improved by incorporating the reasoning used to describe events.
- Difficulties exist associated with the mechanisms of the property inheritance: the values of properties inherited by different parent nodes can be in conflict.
- It is difficult (but not impossible) to express disjunctions (and therefore implications) and negations.
- Logical inadequacy – vagueness about what types and tokens really mean
- Heuristic inadequacy – finding a specific piece of information could be chronically inefficient
- Establishing negation is likely to lead to a combinatorial explosion.
- The "spreading activation" search is very inefficient because it is not knowledge-guided.
- Binary relations are usually easy to represent, but sometimes can be difficult. For example, try to represent the sentence "John caused trouble at the party."

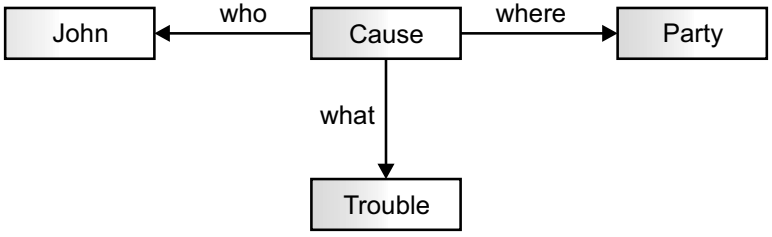


FIGURE 8.25

- Other problematic statements. . .
  - ♦ negation: "John does not go fishing."
  - ♦ disjunction: "John eats pizza or fish and chips."
- Quantified statements are very hard for semantic nets, e.g.:
  - ♦ "Every dog has bitten a postman."
  - ♦ "Every dog has bitten every postman."
  - ♦ **Solution:** Partitioned semantic networks can represent quantified statements

Now we are able to make a semantic network on our own. Let's see how to represent some sentences in a semantic network.

- Try to represent the following two sentences in the appropriate semantic network diagram:

1. isa (person, mannal)
instance (Mike-Hall, person)
team (Mike-Hall, Cardiff) } all in graph
2. score (Cardiff, Llanelli, 23-6)
3. John gave Mary the book.

- **Solution for Problem 1:** is_a(person, mammal), instance(Mike-Hall, person), team(Mike- Hall, Cardiff)

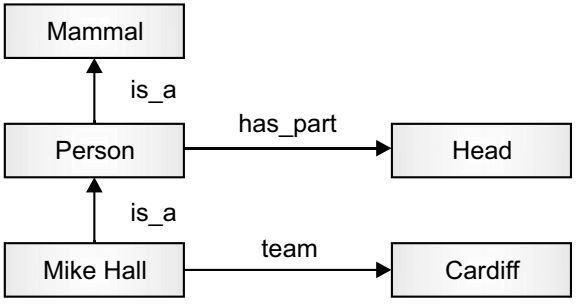


FIGURE 8.26

- **Solution for Problem 2:** score(Spurs, Norwich, 3-1)

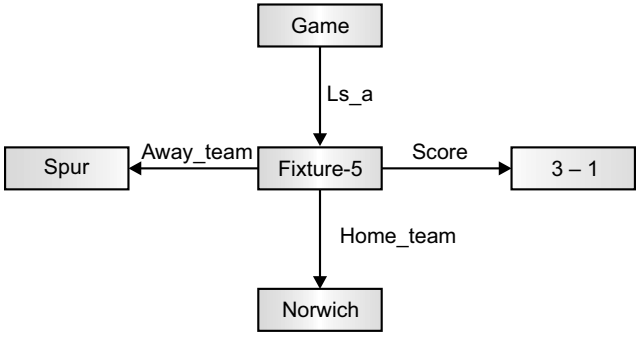


FIGURE 8.27

- **Solution for Problem 3:** John gave Mary the book.

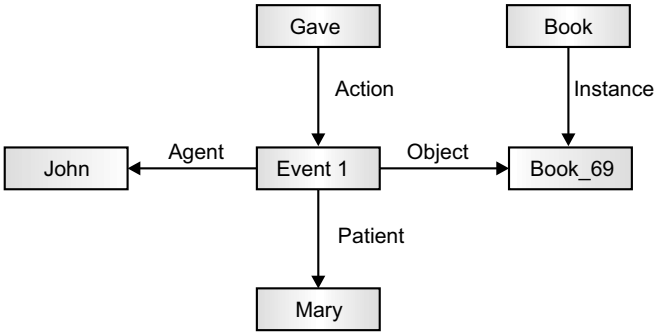


FIGURE 8.28

8.7.2.4 Partitioned Semantic Network

One of the limitations of the semantic network is that it cannot be used to represent quantified statements (for all ∀ and there exist ∃) like

"Every dog has bitten a postman."

"Every dog has bitten every postman."

The solution to this problem is to use a partitioned semantic network. Hendrix (1976: 21-49, 1979: 51-91) developed the so-called partitioned semantic network to represent the difference between the description of an individual object or process and the description of a set of objects. The set description involves quantification.

Hendrix partitioned a semantic network; a semantic network, loosely speaking, can be divided into one or more networks for the description of an individual.

Partitioned semantic networks have the expressive power of predicate calculus that we discussed previously. That is negation, conjunction, disjunction, and implication all can be represented using a partitioned semantic network which otherwise is not possible with a semantic network. But we cannot neglect the advantages of a semantic network, like two-way indexing, direct set-subset element representation, and variable classifications according to types.

Partitioned Semantic Networks Allow For

- propositions to be made without commitment to truth
- expressions to be quantified

In it, the network is broken into **spaces,** which consist of groups of nodes and arcs and regard each space as a node. Therefore, the central idea of partitioning is to allow groups, nodes, and arcs to be bundled together into units called spaces – fundamental entities in partitioned networks, on the same level as nodes and arcs (Hendrix 1979:59).

- Every node and every arc of a network belongs to (or lies in/on) one or more spaces.
- Some spaces are used to encode "background information" or generic relations; others are used to deal with specifics called "scratch" space.

In a partitioned semantic network, the partition is used as a barrier. Since the network is a sub-graph, a partitioned semantic network creates a subgraph that has two important advantages:

- **Syntactic:** It is useful to delimit that part of the network that represents the results of specific inferences.
- **Semantic:** It is useful to delimit that part of the network that represents knowledge about specific objects. Partitioning may then be used to impose a hierarchy upon a flat structure of nodes.

Consider the following:

"Andrew believes that the earth is flat." We can encode the proposition "the earth is flat" in a space and within it have nodes and arcs to represent

the fact (see Figure 8.29). We can have nodes and arcs to link this space the rest of the network to represent Andrew's belief.

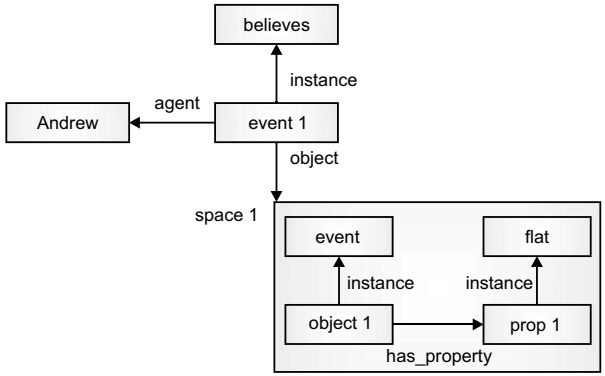


FIGURE 8.29 The Partitioned Semantic Net for Andrew's Proposition

Figure 8.29 shows a simple partitioned semantic network that does not have any quantification, but in many cases, the sentences may not be as straight and simple as seen in previous examples, especially when the scope of certain literals of the sentence is quantified by ∀ (for all) and ∃ (at least one). In these situations, the sentences may be partly broken into segments so that each segment has a specific scope. The networks where the scope of the variables in a sentence are separately shown are called partitioned semantic networks.

Let us discuss some examples of how to make partitioned semantic network for quantification.

a) Suppose that we want to represent the following sentence using a network scheme:

- "The dog bites the mail carrier."
- Since there is not more than one dog/mail carrier, there is no need of partitioning, and hence, that sentence can be represented using an ordinary semantic network.

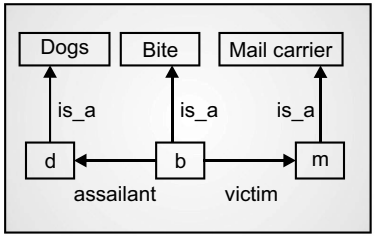


FIGURE 8.30 Semantic Network

b) "Every dog has bitten a mail carrier."

In the above sentence, there is the use of every word that indicates a need for all quantifiers ∀, so we need partitioning as there is a statement that cannot be represented by a semantic network.

Every means ∀ (thus ∀d: ∃ m: dog (*d*) ∧ *m*(mail carrier → bit(d, m)).

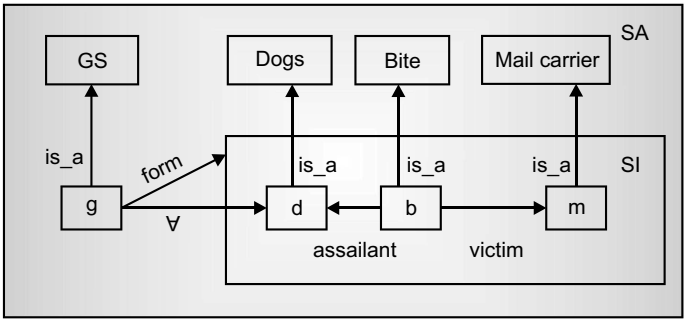


FIGURE 8.31 A Partitioned Semantic Network

In Figure 8.31, the partitioned semantic network comprises two partitions, SA and S1. Node *g* is an **instance** of the special class of general statements about the world comprising link statement, **form**, and one **universal quantifier** ∀.

c) "Every dog has bitten the constable."

Every dog bites the same constable, so the sentence has a generic scope for dogs and not for constables (i.e., ∀*d*: dog (*d*) → bit (*d*, constable)).

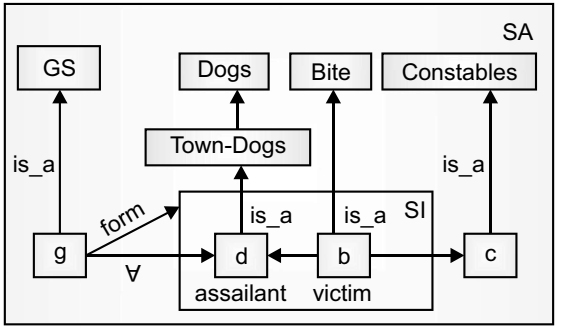


FIGURE 8.32 A Partitioned Semantic Network

d) "Every dog has bitten every mail carrier."

In the above sentence, the scope of the sentence is for dogs as well as for the mail carrier, so the ∀quantifier is connected to both, dogs and the mail carrier $(\forall d\colon \forall\, m\text{:dog}(d) \wedge m(\text{mail carrier} \rightarrow \text{bit}(d, m))$.

The sentences can have a form to envelop the scope of it and the variables that are attached to quantifiers.

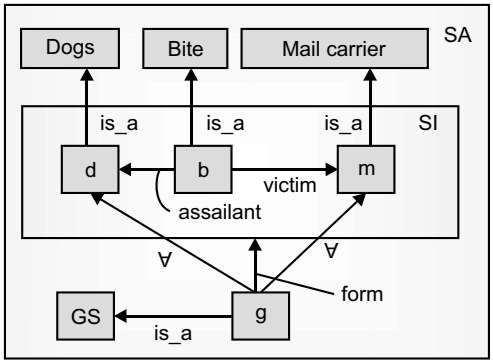


FIGURE 8.33 A Partitioned Semantic Network

e) "Every parent loves their child."

To represent this, we

- Create a general statement, *GS*, a special class.
- Make node *g* an instance of *GS*.
- Every element will have at least 2 attributes:
  - ♦ a form that states which relation is being asserted
  - ♦ one or more for all ∀ or exists ∃ connections—these represent universally quantifiable variables in such statements *e.g.*, x, y in $\forall x\ \text{parent}(x) \rightarrow \exists y\text{: child}(y) \wedge \text{loves}\ (x, y)$.

Here, we have to construct two spaces, one for each x, y.

NOTE: We can express variables as existentially qualified variables and express the event of "love" having an agent p and receiver b for every parent p, which could simplify the network.

If we change the sentence to **"Every parent loves their child,"** then the node of the object being acted on (the child) lies outside the form of

the general statement. Thus, it is not viewed as an existentially qualified variable whose value may depend on the agent. So, we could construct a partitioned network as shown in Figure 8.34.

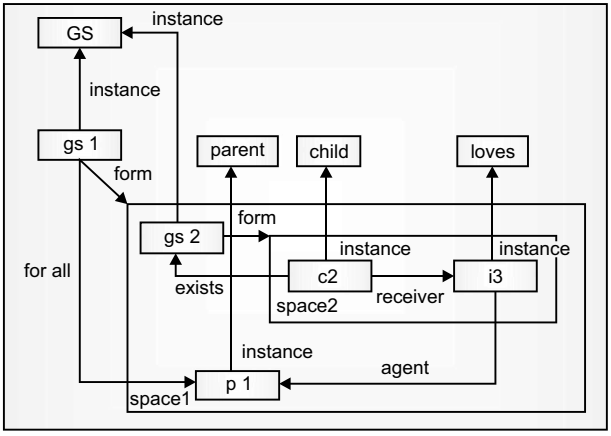


FIGURE 8.34 A Partitioned Semantic Network

f) "John believes that pizza is tasty."

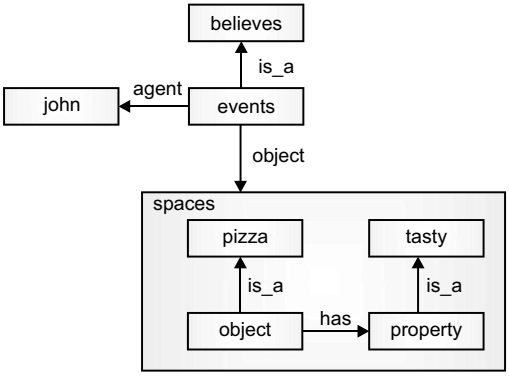


FIGURE 8.35

Thus, the partitioning of a semantic network is

- logically adequate, in that one can distinguish between individuals and sets of individuals,
- indirectly more heuristically adequate by way of controlling the search space by explaining semantic networks.

8.7.3 Frames

The term "frame" was coined by Minsky in 1975. Frames are a more structured form of packaging knowledge. They are a version of the semantic network that was proposed by Minsky. Frames can be viewed as a structural representation of the semantic network. Frames organize our own knowledge of the world. We adjust to new situations by calling up information structured by past experience. We revise the details of the past experience to represent individual differences for the new situation.

A frame is used to represent stereotypical information about an object or concept, with descriptors (slots) and relations to other objects or concepts. A frame is a data structure for representing a stereotyped situation, like going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed. Relations and slots have a structure, too, which helps to describe their semantics. All the information relevant to a particular concept is stored in a single complex entity, the frame. A single frame alone is rarely useful, so we have to build a frame system from a collection of frames that are connected with each other. Thus, frames are organized into hierarchies or a network of frames.

Frames also support inheritance, that is, low level frames can inherit information from upper level frames.

"A frame is a collection of attributes (called slots) and associated values (numeric, symbolic, and Boolean), and it is also called frame identification information." The frame describes some entity in the world, that is, it is the object's description.

Slots are similar to attributes in the object-oriented approach, but it can contain declarative and procedural information.

The sources of attribute values are as follows:

- Initialize
- Database
- Procedure
- Expert System

- User
- Inheritance
- Other Frame (object)

A frame defines the state of an object and its relationship to other frames (objects). But a frame is much more than just a record or data structure containing data. The information is stored in frames with slots. Some of the slots trigger actions, causing new situations.

Frames are templates

- that are to be filled-in in a situation
- filling them in causes an agent to undertake actions and retrieve other frames

In AI, frames are called slot-filler data representations. A frame may have any number of slots needed for describing an object, e.g., the faculty frame may have the name, age, address, and qualification as slot names.

Each frame includes two basic elements: slots and facets.

Frame slots: These are slots with labels describing the attributes and possible values for each attribute. Slots describe the characteristics of frames. Their facets (e.g., the domain, range, cardinality, and default value) define their semantics. They are where the demons are defined. Each slot may contain one or more facets (called fillers or slot descriptors).

Slots in a frame can contain following:

- Frame identification information
  - ♦ **For example,** a frame that stores knowledge about cars can have a name "Car."
- Relationships between this and other frames
  - ♦ **For example,** a superclass of a frame "Car" is a frame "Vehicle."
- Knowledge about an attribute of an object and its value
  - ♦ A frame "Car" can have an attribute "Number of wheels" with a value of 4.
- Procedures to carry out after various slots are filled

- Default information to use where input is missing.
  - ♦ In situations where certain information required for the frame is missing, the defaults can be specified. For instance, a table may be assumed to be wooden until this information can be ascertained. Default information is used in choosing actions until more specific information is found.
  - ♦ A frame "Car" can have a slot "Number of doors" with value 4, however, there are cars with only 2 doors.
- Blank slots – left blank unless required for a task
  - ♦ A frame "Car" can contain a slot "Color", however, this slot will be empty, because there are many different colors. When knowledge about a particular car, for example, John's car, will be represented, then the slot "Color" will have a certain value.
- Other frames, which creates a hierarchy

  A frame can have slots with
  - ♦ **Static values:** the value of the slot does not change during the operation of a system
  - ♦ **Dynamic values:** the value of the slot changes during system operation.

The General Structure of a Frame

Slot Name	Slot value

For example,

BOOK	
Title	: Qualitative Reasoning
Author: Ken D.	: Forbus
Publisher	: Prentice-Hall
Year	: 2000

FIGURE 8.36 A Frame Book with 4 Slots

Above figure shows a frame whose name is "book" as it stores knowledge about a book. It has 4 slots and each slot represents an attribute and value of each attribute, like the title, author, publisher, and year, are the attributes or slot names. 2000 is the value of the attribute (slot) year.

- **Frame facets:** A facet is extended knowledge about a frame's property. Facets provide additional control over the slot value by using procedures and other things. Facets include following information:
    - ♦ TYPE – Defines a type of value that can be associated with the attribute
    - ♦ DEFAULT – Defines a default value, i.e., an initial value for the attribute
    - ♦ CONSTRAINT – Defines the allowable value
    - ♦ MINIMUM CARDINALITY – Establishes the minimum number of values
    - ♦ MAXIMUM CARDINALITY – Establishes maximum number of values
    - ♦ RANGE (indicates the range of integers or enumerated values a slot can have),
    - ♦ DEMONS (allow to obtain information about a frame or to make operations about its characteristics and its relations. Demons are attached to slots and invoked automatically when a slot is accessed)

Demons can be of the following types:

- ♦ if-needed (They activate when the slot is read.)
- ♦ if-added (They activate when a value is added to the slot.)
- ♦ if-removed (They activate when a value is removed from the slot.)
- ♦ if-modified (if-changed) (They activate when a slot's value is modified.)

– Other (may contain rules, other frames, semantic net or any type of other information).

Name of a frame		
Slots	Slot values	Facets

FIGURE 8.37 Structure of a Frame with Slots and Facets

Demons in a Frame

Demons are the methods attached to slots. A method is a procedure attached to an object that will be executed whenever requested. We already explained the types of demons and the methods that are executed whenever the slot is accessed.

Example (*a*)

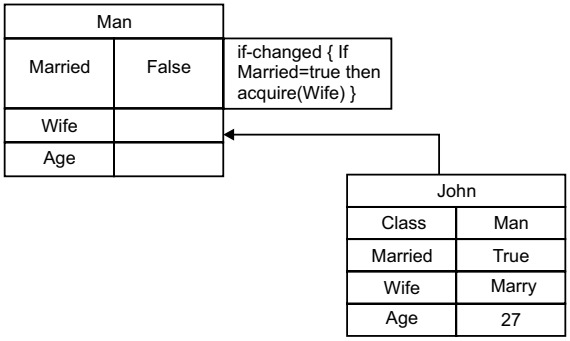


FIGURE 8.38 A Frame System Having "If-Changed" Demons

Example (*b*)

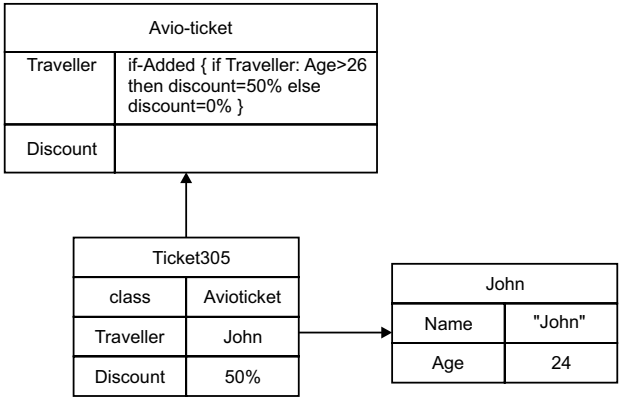


FIGURE 8.39 A Frame System with "If-Added" Demons

A frame consists of a selection of slots that can be filled by values, or procedures for calculating values, or pointers to other frames. Each frame describes an object by embedding all the information about that object in slots.

- Slots are commonly known in programming terms as fields or attributes with an associated value.
- A frame is similar to a database record.
- A frame describes typical instances of the concepts it represents.

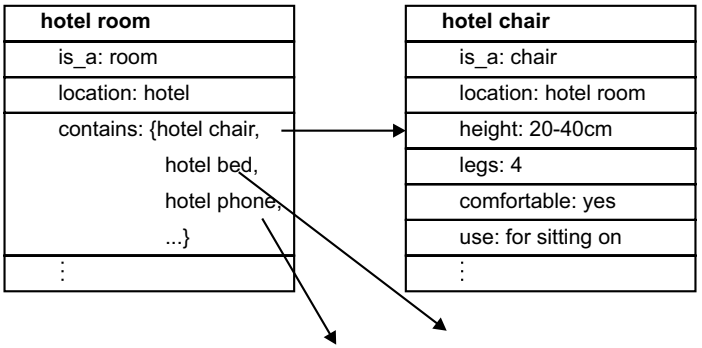


FIGURE 8.40 Frames with Slots

The above figure shows that a hotel room is a frame having three slots: is-a, location, and contains. The "contains" slot has pointers to other frames, like a hotel chair, hotel bed, and hotel phone. Thus, the hotel chair and hotel bed are the constituent parts of the frame "hotel room."

Let us discuss an example of using frames. Suppose an agent is taking notes at a lecture and wants to decide how much attention it should pay and determine any other ways in which it should behave. It searches for frames that match the given situation. It is in a meeting of some kind, so it retrieves that frame. In the specializations slot of meetings is another frame, lecture, which is more appropriate because the context for that is a large number of students. It retrieves the lecture frame and starts filling in slots.

The first slot is the name of the course, which in this case, is the operating system. The next slot is the level of the course, and it's difficult. This fires the procedural rule: "If it's a difficult course, pay attention," so the agent begins to pay more attention. The next slot is "lecturer," and this is a frame in itself, so the agent retrieves the lecturer frame and starts filling in the slots on that frame. The first slot is "tolerance," and this lecturer is

not tolerant. This fires more procedural rules, such as "If it's an intolerant lecturer, then turn off your mobile phone," so the agent turns its phone off. Having dealt with the lecturer frame, it returns to the lecture frame and looks at the next slot, which is the room number. This is flagged to be not important for the task of taking notes, so the agent doesn't fill it in. The frames in this example are portrayed in Figure 8.41.

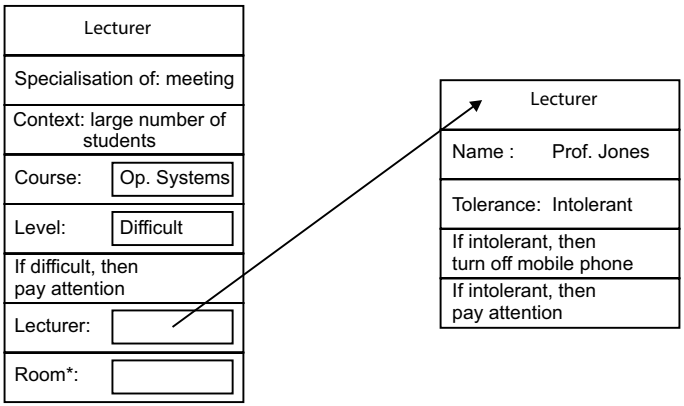


FIGURE 8.41 A Frame System

We can see how this scheme of retrieving frames, filling in slots, and reacting to production rules in the slots can be used to make an agent act rationally. Note that a search may be involved in order to use the frame representation, both in order to find the correct frames for a situation and as part of the procedures carried out when filling the slots.

So, the names of slots correspond to the links in semantic nets and the values of the slots correspond to the nodes. Hence, each slot can be another frame. Frames are often linked into a hierarchy to represent the has-part and is-a relationships. Thus, in this view, each frame represents a class, subclass, or instance of class.

- **Class:** This is a collection of objects that share some common properties (attributes).

A class frame contains

- ♦ a descriptive name of the concept
- ♦ a set of attributes that are characteristic of all its associated objects
- ♦ attribute values that are considered common to these objects

It may contain

1. an explicit reference to all of its associated subclasses
2. information describing the behavior of the concept

- **Sub-class:**

Sub-classes are classes that represent sub-sets of higher level classes. Subclasses inherit the attributes of high level classes and also have their own attributes.

There are three kinds of class relationships:

Generalization – "Kind of" relationship

Aggregation – "Part of" relationship

Association – "Semantic" relationship

- **Instance:** The instance is a specific object from a class of objects. For example, "Ram" is an instance of class "Person."
  - ♦ Describes
- A specific object from its related class.
  - ♦ Contains
- All of the characteristics of the class frame as well as a specific information (specific features and property values).

So it's clear now that we have three types of frames.

8.7.3.1 Class Frame

A class frame includes slots describing an attribute of a class of objects. Typically, the slots of such frames have default information or unspecified values that can be redefined at a lower level, that is, in the subclass frame. If the class frame has an actual value facet, then decedent frames cannot modify that value. The value remains unchanged for subclasses and instances. Slots of the class-level frame represent attributes that are common to all members of that class. For example, class "vehicle" has 4 attributes and is common for all classes, that is, members of that class (that is "car") is a subclass of "vehicle," so all attributes of the "vehicle" frame are the same for the "car" frame.

Class: Vehicle	
Regno	
Model	
Producer	
Owner	

FIGURE 8.42 A Class Frame

8.7.3.2 Subclass Frames

A subclass derives attributes of the superclass and also has its own attributes.

Car	
Class: Vehicle	
Regno	
Model	
Producer	
No. of doors	4
Owner	

FIGURE 8.43 A Subclass Frame

Figure 8.43 shows a subclass "Car" that derives the attributes of the class "Vehicle" and the subclass "Car"'s own attribute is "No. of doors."

8.7.3.3 Instance Frames

The value of the attributes of the instance frame varies among all instances of the class.

John's Car	
Class: Car	
Regno	123
Model	La78
Producer	Toyota
No of door	4
Owner	John

FIGURE 8.44 The Instance Frame

Figure 8.44 shows an instance frame: "John's Car" is an instance or object of the class "Car." Each instance takes different values for the attribute. The attribute is the same for all instances of the class "Car," but the value of the attribute is different for each instance.

Here is an example showing the class and instance frame:

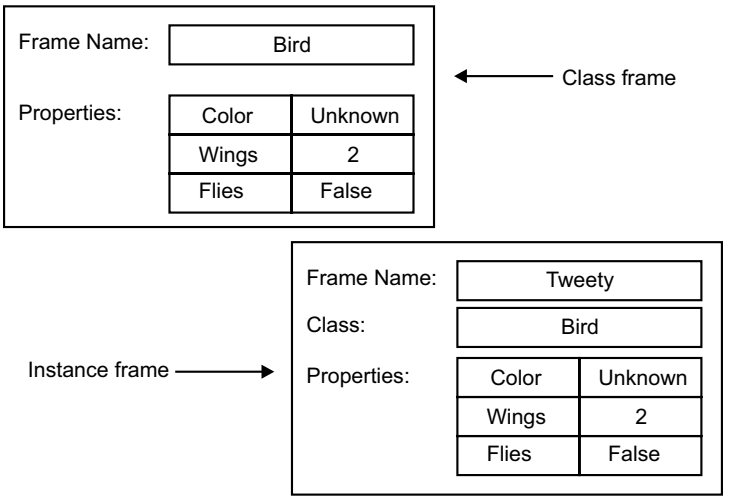


FIGURE 8.45 The Class and Instance Frames

An example showing all the frames that we discussed above:

Mammal

Sub-class:	Animal
Warm_blooded:	yes

Elephant

Sub-class:	Mammal
*Color:	grey
*Size:	large

Clyde

Instance:	Elephant
Color:	pink
Owner:	Fred

FIGURE 8.46 A Frame System with Different Types of Frames

Figure 8.46 shows that "Mammal" is a subclass of the class "Animal," so to derive the attributes of the class "Animal" and "Mammal," the subclasses must have their own attribute called warm_blooded with the value "yes." Similarly, "Elephant" is a subclass of "Mammal" and "Clyde" is an instance frame, that is, "Clyde" is an instance of the "Elephant" subclass.

One important point to remember that there are two kinds of attributes that are associated with a class. The first type of attribute belongs to only that class itself, and the second type of attribute is that which is inherited by all the instances of that class. The second type is indicated by an asterisk (*). For example, in Figure 8.46, class "Elephant" has three attributes: subclass, color, and size. Out of these three, the color and size attributes are marked with an asterisk, so these two attributes are inherited by all the instances of the class "Elephant."

8.7.3.4 Relationships in Frames

An individual frame is of little use; there is a need of a frame system with a number of frames that have some relationship with each other. There can be any kind of relationship between the frames, such as a superclass-subclass relationship or instance relationship. Relations allow connections between frames. Relations have a description that defines their semantics and functioning. Frames also support inheritance and slot inheritance is based on relations. Three types of relations connect frames within a frame system. Relationships are divided into two categories: taxonomic and non-taxonomic. Is-a, instance, and a-part-of are taxonomic relations (predefined).

- "Is-a" relationship: This relates a subclass frame with a class frame or an instance frame with a subclass or class frame. "Is-a" relation correspond to the "subset" relation and "instance" relation is part of the "element of" relation. In this case, a subclass frame or an instance frame inherits all the slots from a class frame, but it can include a new slot.

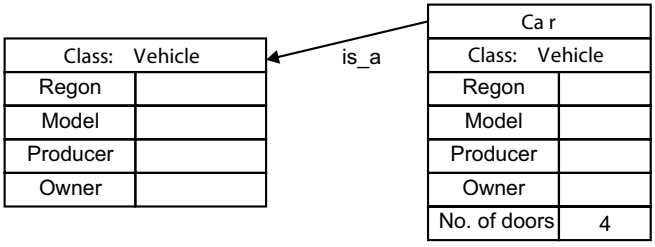


FIGURE 8.47 A Frame System with the Is-A Relationship

Figure 8.47 shows two frames, "Vehicle" and "Car." "Car" is a subclass derived from the "Vehicle" superclass, so we derive all attributes of the "Vehicle" class and have one slot of its own called "No. of doors."

- **"A-Part-Of" relationship:** Relates the whole with its constituent parts. That is, one frame becomes the values of the slot of another frame.

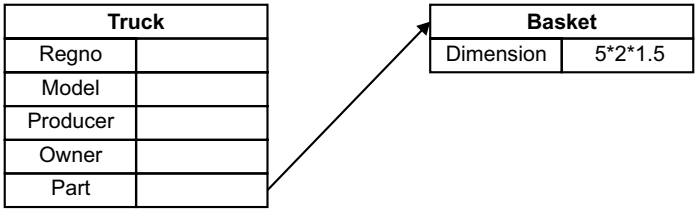


FIGURE 8.48 A-Part-Of Relationship

Figure 8.48 shows that the frame "Basket" is part of the frame "Truck." The frame "Truck" has a slot name "Part" and the value of "Part" is a frame itself. So, both frames are connected by the "a-part-of" relationship.

We now look at one more example of a frame system that shows the "a-part-of" relationship.

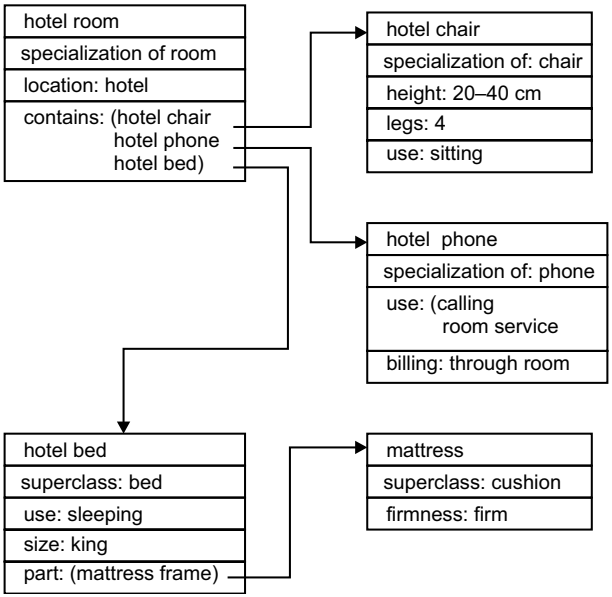


FIGURE 8.49 A Frames System with Pointers to Other Frames

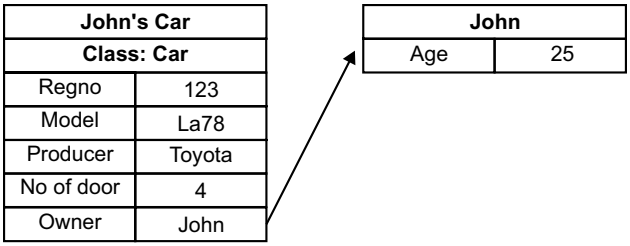


FIGURE 8.50 A Semantic Relationship

A complete frame-based representation will consist of a whole hierarchy or network of frames connected together by the appropriate links pointers.

Thus, the nodes in a frame system are connected using links viz.

- **ako:** links two class frames, one of which is a subclass of the other, e.g., the science_faculty class is ako of faculty class.
- is_a (subclass/class)/instance (instance/class) connects a particular instance of a class frame, e.g., Renuka is_a science_faculty.
- a_part_of connects two class frames, one of which is contained in the other, e.g., the faculty class a_part_of department class.
- The property link of the semantic net is replaced by SLOT fields.

Let us see various examples of the frame system with relationships.

Example (*a*)

In our example we have a general class of birds, and all birds have the attributes "flying," "feathered," and "color." The attributes "flying" and "feathered" are Boolean values and are fixed to true at this level, which means that for all birds, the attribute "flying" is true and the attribute "feathered" is true. The attribute "color," though defined at this level, is not filled, which means that although all birds have a color, their color varies. Two subclasses of birds, `pet_canaries` and `ravens` are defined. Both have the "color" slot filled in, `pet_canaries` with yellow, `ravens` with black. The class `pet_canaries` has an additional slot, `owner`, meaning that all pet canaries have an owner, though it is not filled at this level since it is obviously not the case that all pet canaries have the same owner. We can therefore say that any instance of the class `pet_canary` has the attributes `color yellow`, `feathered true`,

flying true, and owner, the last of these varying among instances. Any instance of class raven has color black, feathered true, and flying true, but no attribute owner. The two instances of pet_canary shown, Tweety and Cheepy, have the owners John and Mary, who are separate instances of the class Person, for simplicity, no attributes have been given for the class Person. The instance of pet_canary Cheepy has an attribute that is restricted to itself, Vet (since not all pet canaries have their own vet), which is a link to another "Person" instance, but in this case, we have a subclass of Person, Vet. The frame diagram for this is as follows.

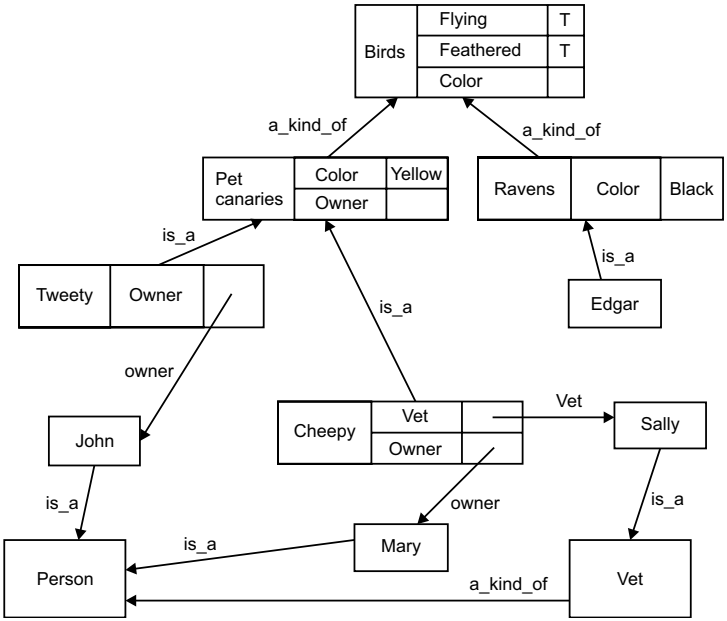


FIGURE 8.51 Frame System for Birds

Example (*b*)

In this frame system, there is a superclass called "Vehicle" with a slot named "Wheels" and a value of the slot is "yes." Automobile and cycle are subclasses inherited from "Vehicle." The "Sports Car" class is inherited from "Automobile" and the "Bicycle" class is a subclass of "Cycle" so the relationship link is "is-a." "Corvette" is an instance of the class "Sports Car," so the link is an "instance" link.

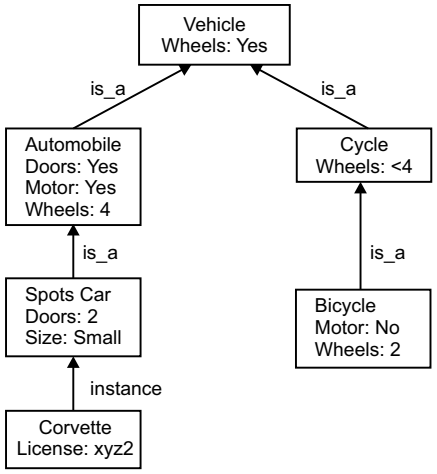


FIGURE 8.52 A Frame System with Different Types of Relationships

Example (*c*)

In this example, "Animals" is a class frame with two slots, "Alive" and "Flies" with the values T and F, respectively. "Birds" and "Mammals" are the subclasses of "Animals" as the link is "is-a" between the nodes. "Opus" is an object or instance of the class "Penguin" as the link between them is "instance."

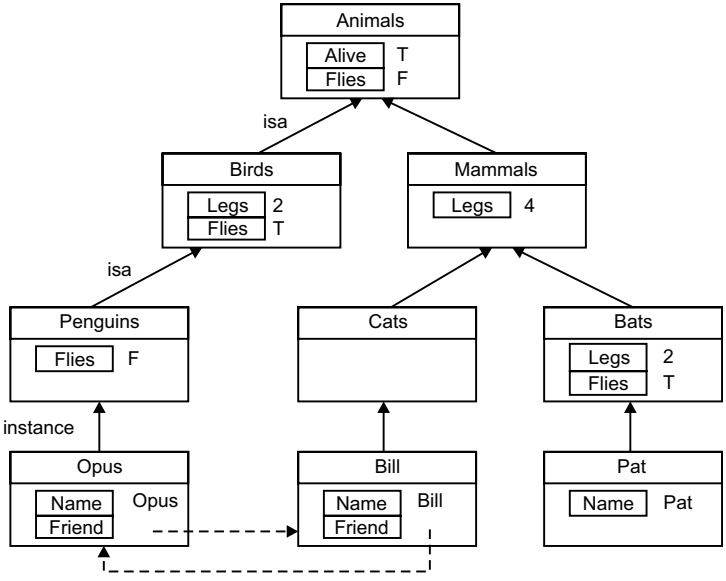


FIGURE 8.53 Frame System for Animals

8.7.3.5 Inheritance in Frames

Frames also support inheritance like the semantic network because frames in a frame system are connected by links, like is_a, instance, and kind_of, that show inheritance.

- **Definition:** Inheritance is the process by which the characteristics of a parent frame are assumed by its child frame, and the child frame has its own attribute also.
- **Note:** In general, a child frame will inherit information from its parents, grandparents, and great-grandparents.

For example, in Figure 8.53, the frame "Car" is a child frame and a subclass of the class "Vehicle" (the parent frame), so all the attributes of the "Vehicle" class belong to the class "Car" also and it has its own attributes. "No. of doors" is an attribute of the frame "Car" and the rest of the attributes of the frame "Car" are inherited from the "Vehicle" class.

Some points to remember about frame inheritance:

- If a slot is not defined for a given frame, we can look at the parent-class slot with the same name. The class frame generally has default values that can be re-defined at lower levels.
- If the class frame has actual value facets, then the decedent frames (child frames) cannot modify that value.
- The value remains unchanged for subclasses and instances.
- In the case of taxonomic relations, inheritance (of slots and values) is given by default.
- In the rest of the relations (non-taxonomic), it has to be explicitly defined.
- There are slots inherently non-inheritable, e.g., the has-instance slot can never be inherited.

In Figure 8.54, "department" and "hostel" are frames that are contained within the frame "university" as the value of one of the slots of the "university" frame as the relationship is a_part_of. "Nilgiri hostel" is a child frame of the frame "hostel" as the relationship is is_a. Let us see how to represent that frame system properly with slots and values as shown below.

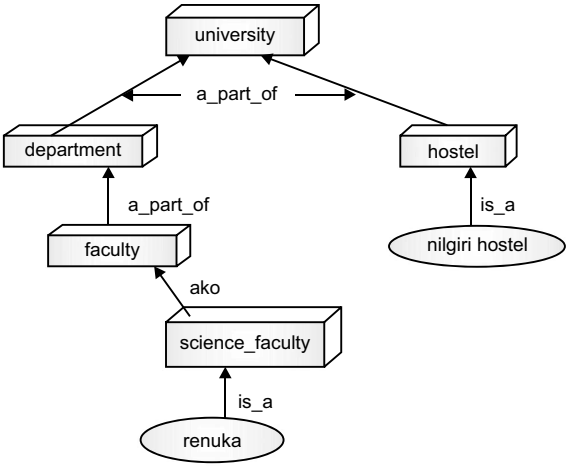


FIGURE 8.54 Example of a Frame Network

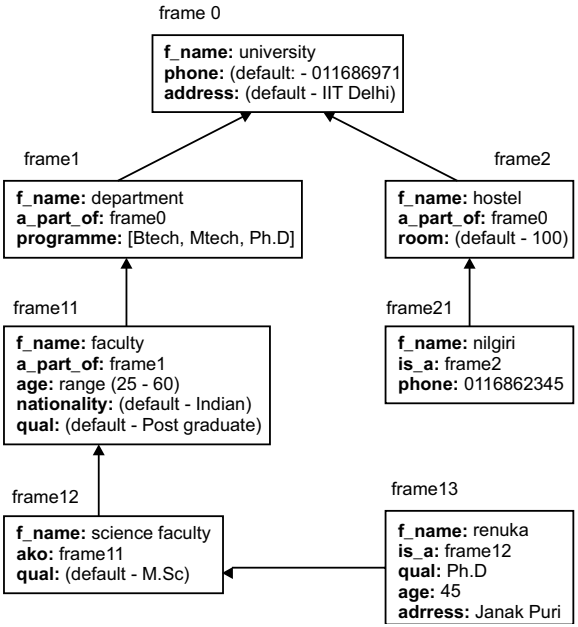


FIGURE 8.55 Detailed Representation of the Above Frame Network

From Figure 8.55, we can infer some knowledge is not given explicitly in the frame network. Suppose we want to know the nationality or phone number of an instance-frame frame 13 of Renuka.

This information is not given in this frame. The search will start from frame 13 in the upward direction until we get our answer or have reached the root frame.

Multiple inheritance: This means the child frame can inherit information from more than one parent class frame. So, in multiple inheritance, there is more than one parent frame for a child frame, which sometimes causes ambiguity.

Some points to remember about multiple inheritance:

- If the taxonomy is a tree, the inheritance is simple.
- The inheritance is multiple if
  - ♦ the taxonomy is a graph
  - ♦ there are other (non-taxonomic) relations, which allow inheritance
  - ♦ If there is inheritance of the slots and values, there may be a conflict of values.
- needs an algorithm for traversing the is_a hierarchy that guarantees that specific knowledge will always dominate more general facts
- needs an inheritance algorithm that reports the ambiguity

Examples of multiple inheritance: (*a*)

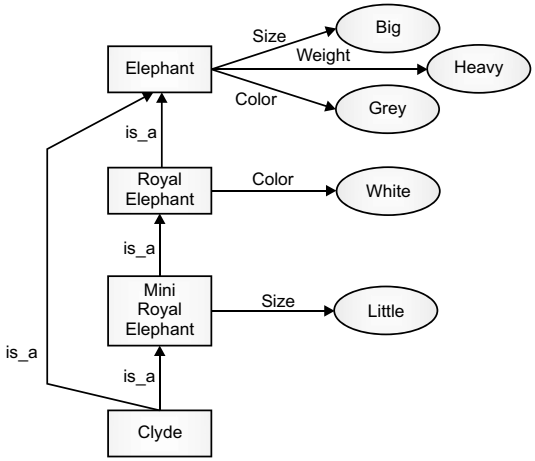


FIGURE 8.56 Multiple Inheritance

In Figure 8.56, the confusion is that "Clyde" inherits information from both "Mini Royal Elephant" and "Elephant," so there are two parent classes for it. So, "Clyde" becomes both an elephant and mini royal elephant.

Example (*b*)

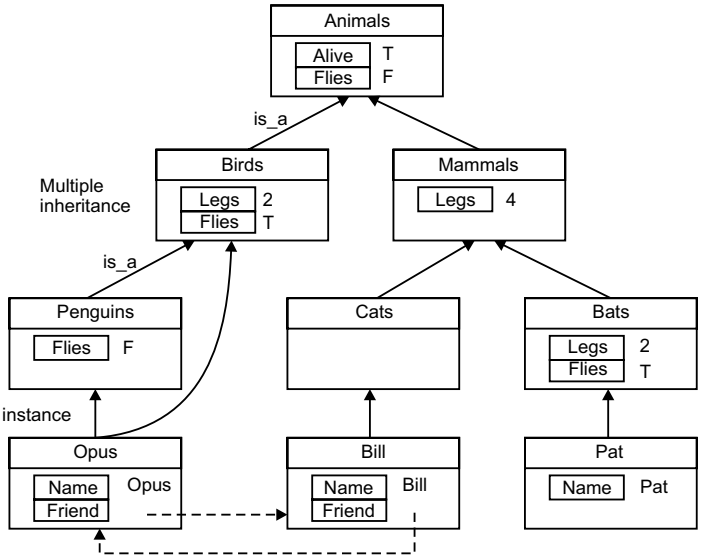


FIGURE 8.57

To deal with multiple inheritance, an inferential distance algorithm is required. The inferential distance algorithm allows us to define from which other frame a given frame inherits.

1. Create the set of frames from which the value of the slot can be explicitly inherited → Candidates.
2. Remove from "Candidates" all frames that are parents of other frames of the set.
3. If the resulting number of candidates is
   - ♦ 0 → The slot cannot be inherited.
   - ♦ 1 → This is the value to be inherited.
   - ♦ $N > 1$ → There is a multiple-inheritance problem if the slot's cardinality is not at least N.

8.7.3.6 Advantages of Frames

- A frame collects information about an object in a single place in an organized fashion.
- By relating slots to other kinds of frames, a frame can represent the typical structures involving an object; these can be very important for reasoning based on limited information.
- Frames provide a way of associating knowledge with objects (via the slot procedures).
- Frames may be a relatively efficient way of implementing AI applications (direct procedure invocation versus search in a logic system).
- Frames allow data that are stored and computed to be treated in a uniform manner.(e.g., AGE might be stored, or might be computed from BIRTHDAY.)
- Object-oriented programming has much in common with frames.
- Expressive power
- Easy to set up slots for new properties and relations
- Easy to include default information

8.7.3.7 Disadvantages

- Difficult to program
- Difficult for inference
- Lack of inexpensive software
- Slot fillers must be "real" data.
- It is not possible to quantify over slots. For example, there is no way to represent "Some student earned 100% on the exam."
- It is necessary to repeat the same information to make it usable from different viewpoints, since the methods are associated with slots or particular object types.

8.7.4 Scripts

Knowledge representation researchers, particularly Roger Schank and his associates, devised some interesting variations on the theme of structured objects. In particular, they invented the idea of scripts (1973). Schank and his co-workers developed a technique for reducing a story or a newspaper report to conceptual primitives and their interrelations. The conceptual primitives

specify certain basic actions that people and objects can perform, for instance, transferring a physical thing from one location to another, transferring a mental idea from one mind to another, building new information from old, grasping objects, focusing attention of a sense-organ on some occurrence, ingesting some form of nourishment, and so on. Schank showed how complex representations of the meaning of individual sentences could probably be built up from these conceptual primitives. To represent a narrative composed of a series of linked sentences, Schank proposed using frame-like structures, called scripts, which record the normal sequence of events for a given type of occurrence. A script is a way of representing specific knowledge, that is, detailed knowledge about an event or situation. Scripts are used in natural language understanding systems to organize a knowledge base in terms of the situations that the system should understand. Script theory is primarily intended to explain language processing and higher thinking skills.

Frames and scripts offer extremely rich and versatile methods for representing organized clusters of knowledge about every day or specialized occurrences. They reproduce a powerful feature of our own thinking processes: the fact that our understanding of new situations is often driven by stereotypes, which can be applied in a rough-and-ready way, avoiding the need for extensive inferential processes in order to build up an understanding from scratch. Often our initial attempts to make sense of a situation are quite inappropriate, and we have to make improvisations and revisions as we go along. Frame and script systems are able to incorporate such flexibility. Although frame and scripts were developed independently (both originating in the early 1970s), and are different in important ways, they have sufficient similarities to be considered together. One influential proponent of frame-based systems is Marvin Minsky (1975); a champion of script-based systems was Roger Schank (see Schank and Abelson, 1977). The key idea involved in both frames and scripts is that our knowledge of concepts, events, and situations is organized around the expectations of the key features of those situations.

A script is a knowledge representation technique that is somewhat similar to frames: it uses inheritance and slots and portrays stereotyped situation (i.e., if the system isn't told some detail of what's going on, it assumes the "default" information is true), but it is used for describing events in terms of contexts, participants, and sub-events rather than just an object. It is similar to a thought sequence or a chain of situations that could be anticipated. It could be considered to consist of a number of slots or frames

but with more specialized roles. Scripts are used for interpreting stories. Popular examples are script-driven systems that can interpret and extract facts from newspaper stories.

Definition of a Script

- A script is a remembered precedent, consisting of tightly coupled, expectation-suggesting primitive-action and state-change frames [Winston, 1992]
- A script is a structured representation describing a stereotyped sequence of events in a particular context [Luger and Stubblefield, 1998]
- A script is a structure that describes a stereotyped sequence of events in a particular context
    - ♦ closely resembles a frame, but with additional information about the expected sequence of events and the goals/motivations of the actors involved
    - ♦ the elements of the script are represented using conceptual dependency relationships (as when the actions are reduced to conceptual primitives)
- A script is a data structure used to represent a sequence of events.
- A script is a structure that prescribes a set of circumstances which could be expected to follow on from one another.

Why We Use Scripts

Real-world events follow stereotyped patterns. Human beings use previous experiences to understand verbal accounts; computers can use scripts instead. When relating events, people use large amounts of assumed detail out of their accounts. People don't find it easy to converse with a system that can't fill in missing conversational details. A script predicts unobserved events and can build a coherent account from disjointed observations.

Scripts have been used to

- ♦ interpret, understand, and reason about stories
- ♦ understand and reason about observed events
- ♦ reason about observed actions
- ♦ plan actions to accomplish tasks

Components of Scripts

- Entry Conditions

  What are the descriptors of the world that must be true for the script to be called?

- Props

  What objects make up the content of the script?

- Roles

  What actions are performed by the participants in the script?

- Scenes

  Temporal decomposition of script into meaningful episodes

- Results

  What are the outcomes following termination of the script?

Special symbols of actions are used for scripts.

Table 8.3 The Symbols for Script Actions

Symbol	Meaning	Example
ATRANS	transfer a relationship	give
PTRANS	transfer the physical location of an object	go
PROPEL	apply physical force to an object	push
MOVE	move body part by owner	kick
GRASP	grab an object by an actor	grasp
INGEST	ingest an object by an animal	eat
EXPEL	expel from an animal's body	cry
MTRANS	transfer mental information	tell
MBUILD	mentally make new information	decide
CONC	conceptualize or think about an idea	think
SPEAK	produce sound	say
ATTEND	focus sense organ	listen

In each scene, one or more actors perform actions. The actors act with the props. The script can be represented as a tree or network of states driven by events. As with frames, scripts drive interpretation by telling the system what to look for and where to look next. The script can predict events.

Let us discuss an example of a restaurant script.

Script: RESTAURANT

Props:	Tables	Results:
	Menu	*S* has less money
	F = Food	*O* has more money
	Bill	*S* is not hungry
	Money	*S* is pleased (optional)
Roles:	*S* = Customer	Scenes:
	W = Waiter	Entering
	C = Cashier	Ordering
	M = Cashier	Eating
	O = Owner	Exiting

Entry conditions:

S is hungry

S has money

Scene 1: Entering

`S PTRANS S` into restaurant, `S ATTEND` eyes to tables, `S MBUILD <` where to sit, `S PTRANS S` to table, `S MOVE S` to sitting position

Scene 2: Ordering

`S PTRANS<` menu to S (menu already on table), `S MBUILD<` choice of food, `S MTRANS<` signal to waiter, waiter `PTRANS` to table, `S MTRANS< "I want food"` to waiter, waiter `PTRANS` to cook.

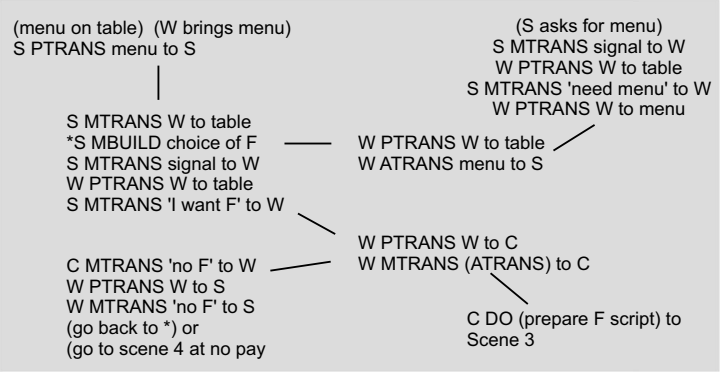


FIGURE 8.58 An Ordering Scene

Besides the actions PTRANS (transfer of location) and ATRANS (transfer of possession), this script uses two more of their primitive actions, namely, MTRANS (transfer of information) and MBUILD (creating or combining thoughts). CP(S) stands for S's "conceptual processor" where the thought takes place, and DO stands for a "dummy action," d. The lines in the diagram show possible alternative paths through the script. So, for example, if the menu is already on the table, the script begins at the upper left-hand corner; otherwise it begins at the upper right-hand corner. I believe most of the script is self-explanatory, but I'll explain what goes on in the middle. S brings the "food list" into its central processor where it is able to mentally decide (build) a choice of food. S then transfers information to the waiter to come to the table, which the waiter does. Then, S transfers the information about his or her choice of food to the waiter. This continues until either the cook tells the waiter that he does not have the food that is ordered or the cook prepares the food.

Scene 3: Eating

Cook ATRANS food to waiter, waiter PTRANS food to S, S INGEST food

C ATRANS F to W
W ATRANS F to S
S INGEST F
(Option: Return to Scene 2 to order more;
otherwise go to Scene 4)

FIGURE 8.59 An Eating Scene

Scene 4: Exiting

waiter `MOVE` write check, waiter `PTRANS` to `S`, waiter `ATRANS` check to `S`, `S ATRANS` money to waiter, `S PTRANS` out of restaurant

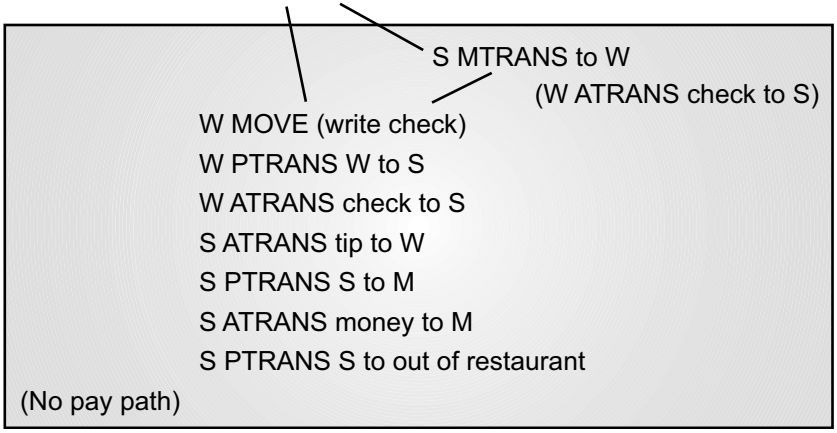


FIGURE 8.60 An Exiting Scene

There are many variations possible on this general script having to do with different types of restaurants or procedures. For example, the script above assumes that the waiter takes the money; in some restaurants, the check is paid to a cashier. Such variations are opportunities for misunderstandings or incorrect inferences.

Scripts help explain some of the reasoning we do automatically when we hear a story. For example, if we hear that John went to a coffee shop and ordered lasagna, we can reasonably assume that lasagna was on the menu. If we later learn that John had to order something else instead, we can assume that the coffee shop was out of lasagna. Schank and Abelson gave unreliable evidence that even small children build such scripts and that people must have a great number of scripts to enable them to navigate through and reason about the situations they encounter.

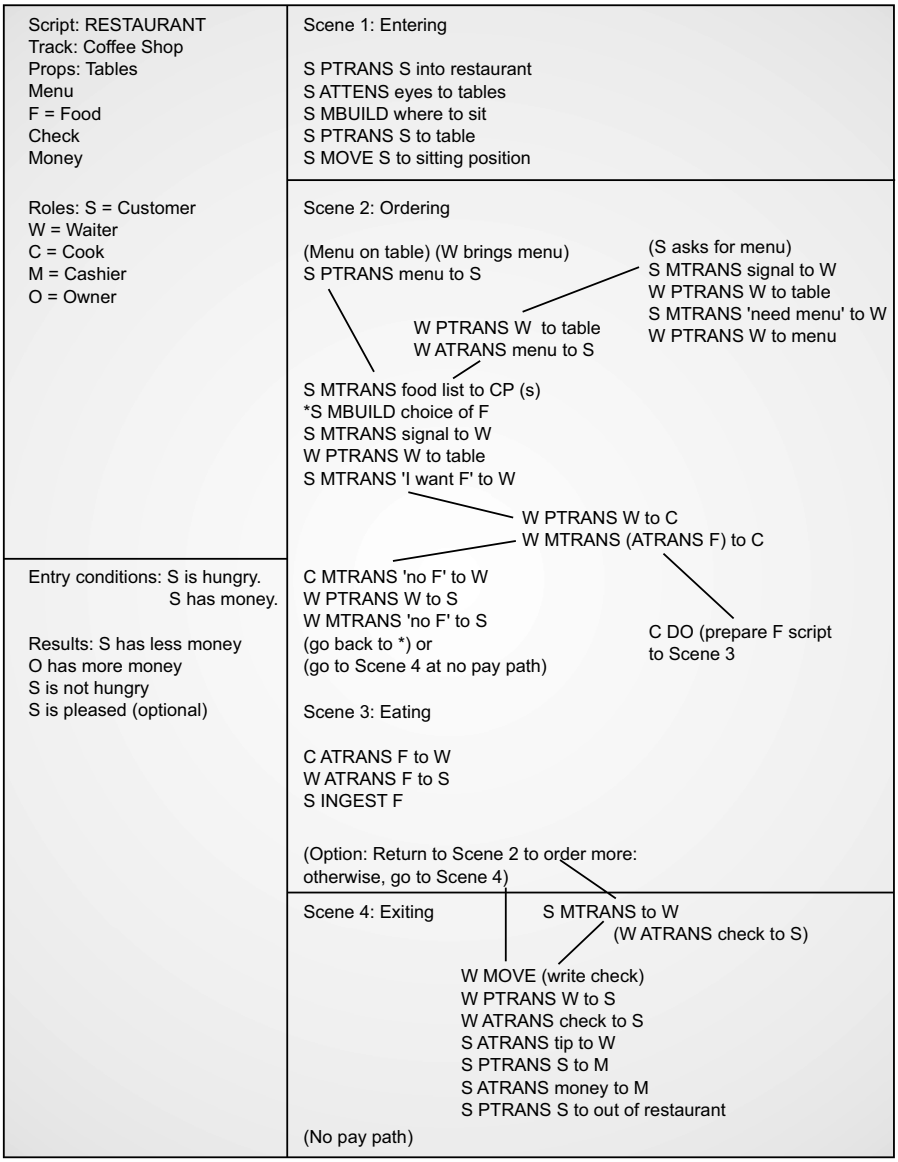


FIGURE 8.61 A Complete Restaurant Script

Let us discuss one more example of a script.

Scripts are useful in describing certain situations such as robbing a bank. This might involve

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

Here, the *Props* might be

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

The *Roles* might be

- robber, *S*
- cashier, *M*
- bank manager, *O*
- policeman, *P*

The Entry Conditions might be

- *S* is poor.
- *S* is destitute.

The Results might be

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

There are 3 scenes: obtaining the gun, robbing the bank, and making the getaway.

Script: ROBBERY	*Track: Successful Snatch*
Props:	*Roles:*
G = Gun,	*R* = Robber,
L = Loot,	*M* = Cashier,
B = Bag,	*O* = Bank Manager,
C = Get away car.	*P* = Policeman.
Entry Conditions:	*Results:*
R is poor.	*R* has more money.
R is destitute.	*O* is angry.
	M is in a state of shock.
	P is shot.

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 exit
O ATRANS raises the alarm
(go to scene 3)
Scene 3: Holding up the bank
M PTRANS C

FIGURE 8.62 A Complete Script for a Bank Robbery

The following table shows an example of a script.

Script: Play in theater	Various Scenes
Track: Play in Theater **Props:**	*Scene 1: Going to theater* • P PTRANS P into theater • P ATTEND eyes to ticket counter
• Tickets • Seat • Play **Roles:** • Person (who wants to see a play) – P • Ticket distributor – TD • Ticket checker – TC **Entry Conditions:** • P wants to see a play • P has money **Results:** • P saw a play • P has less money • P is happy (optional if he liked the play)	*Scene 2: Buying ticket* • P PTRANS P to ticket counter • P MTRANS (need a ticket) to TD • TD ATRANS ticket to P
	Scene 3: Going inside hall of theater and sitting on a seat • P PTRANS P into Hall of theater. • TC Attend eyes on ticket POSS_BY P • TC MTRANS (showed seat) to P • P PTRANS P to sear • P Moves P to sitting position
	Scene 4: Watching a play • P ATTEND eyes on play • P MBUILD (good moments) from play
	Scene 5: Exiting • P PTRANS P out of Hall and theater

Script Invocation

- It must be activated based on its significance.
- If the topic is important, then the script should be opened.
- If a topic is just mentioned, then a pointer to that script could be held.
- For example, given "John enjoyed the play in theater," a script "Play in Theater" suggests the above is invoked.
- All implicit questions can be answered correctly.

- Here, the significance of this script is high.
  - ♦ Did John go to theater?
  - ♦ Did he buy a ticket?
  - ♦ Did he have money?
- If we have a sentence like "John went to theater to pick his daughter," then invoking this script will lead to many wrong answers.
  - ♦ Here, the significance of the script theater is less.
- Getting significance from the story is not straightforward. However, some heuristics can be applied to get the value.

Some additional points to note on scripts:

- If a particular script is to be applied, it must be activated and the activation depends on its significance.
- If a topic is mentioned in passing, then a pointer to that script could be held.
- If the topic is important then the script should be opened.
- The danger lies in having too many active scripts much as one might have too many windows open on the screen or too many recursive calls in a program.
- Provided events follow a known trail, we can use scripts to represent the actions involved and use them to answer detailed questions.
- Different trails may be allowed for different outcomes of scripts (e.g., The bank robbery goes wrong).

8.7.4.1 Advantages of Scripts

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

8.7.4.2 Disadvantages of Scripts

- less general than frames
- may not be suitable to represent all kinds of knowledge

8.7.5 Conceptual Dependency (CD)

Conceptual Dependency (CD) theory was developed by Schank in 1973 to 1975 to represent the meaning of natural language sentences.

- It helps in drawing inferences.
- It is independent of the language.
- CD representation of a sentence is not built using words in the sentence, rather, it is built using conceptual primitives which give the intended meanings of words.

The main goal behind CD was to develop a representation of the conceptual base that underlies all natural languages. This goal required a small set of primitive actions and a set of dependencies that connected the primitive actions with each other and with their actors, objects, and instruments. The claim was that this small set of representational elements could be used to produce a canonical form representation for English sentences and other natural languages. CD provides structures and a specific set of primitives from which representation can be built.

Thus, CD is a theory of how to represent the meaning of natural language sentences in a way so that

- it facilitates drawing inferences from the sentences
- the representation (CD) is independent of the language in which the sentences were originally stated

Schank's claim was that sentences can be translated into basic concepts expressed as a small set of semantic primitives. Conceptual dependency allows these primitives, which signify meanings, to be combined to represent more complex meanings.

Semantic Nets vs. Conceptual Dependency

- Semantic nets only provide a structure into which nodes representing information can be placed.
- Conceptual Dependency representation, on the other hand, provides both a structure and a specific set of primitives out of which representations of particular pieces of information can be constructed.

Building Blocks of Conceptual Dependency

- Primitive conceptualizations (conceptual categories)
- Conceptual dependencies (diagrammatic conventions)
- Conceptual cases
- Primitive acts
- Conceptual tenses

8.7.5.1 Primitive Conceptualization

Primitive conceptualization means analyzing a sentence at the conceptual level. There are four primitive conceptualizations (conceptual categories):

- actions (ACT: actions)
- objects (PP: picture producers)
- modifiers of actions (AA: action aiders): "quickly" is a AA modifier (e.g., He quickly runs.)
- modifiers of objects (PA picture aiders): "blue" is a PA modifier (e.g., a blue car)

First, it is necessary to understand what a concept in CD is used for in regards to representing concepts.

Concept: a concept can be abstract or concrete

- **Concrete concepts** (objects), for instance, a cat, telephone, or book. These concepts are characterized by our ability to form an image of them in our minds. Concrete concepts include generic concepts such as a cat or book along with concepts of specific cats and books.
- abstract concepts, for instance, beauty, loyalty, and love that do not correspond to images in our minds

8.7.5.2 Conceptual Dependencies

Conceptual dependencies refer to relationships between conceptual categories (AA, ACT, PP, PA).

- In a dependency relation, one partner or item is dependent, and the other is dominant or governing.

- A governing dependent is a partially ordered relationship
  - ♦ A dependent must have a governor and it is understood in terms of the governor.
  - ♦ A governor may or may not have a dependent(s) and has an independent existence.
  - ♦ A governor can be a dependent.
  - ♦ PP and ACT are inherently governing categories.
- PA and AA are inherently dependent.
- For a conceptualization to exist, there must be at least two governors:

For example, consider "Sally stroked her fat cat."

PP :	Sally, cat, her [Sally]
ACT :	stroke
PA :	fat
Governors :	Sally, stroke, cat
Dependent :	PP (cat) on ACT (stroke)
	PA (fat) on PP (cat)
	PP (cat) on PP (her[Sally])

Before going forward, we need to understand the conceptual graph. In 1984, John Sowa published his conceptual graph approach. This is something like the semantic net approach described earlier. It is more sophisticated in the way in which it represents concepts. It is much more precise regarding what the objects in the graphs mean in real-world terms.

Some conventions are

- Arrows indicate directions of dependency.
- Double arrow indicates two-way links between the actor and action.

Conceptual graph: A conceptual graph is a finite, connected, bipartite graph. The nodes of the graph denote either concepts or conceptual relations. Conceptual graphs do not use labelled arcs (a semantic network uses labelled arcs). Instead, the conceptual relation nodes represent relations between concepts. Concepts can only have arcs to conceptual

relations, and vice versa. Concepts are represented as boxes and conceptual relations as ellipses.

- Conceptual relation nodes indicate a relation involving one or more concepts. Some special relation nodes, namely, agent, recipient, object, and experiencer, are used to link a subject and the verb.
- Conceptual graphs can represent the relations of any arity.
- A relation of arity n is represented by a conceptual relation node having n arcs.
- Each conceptual graph represents a **single proposition**
- The knowledge base contains **a set** of conceptual graphs.
- Graphs may be **arbitrarily complex,** but must be finite.

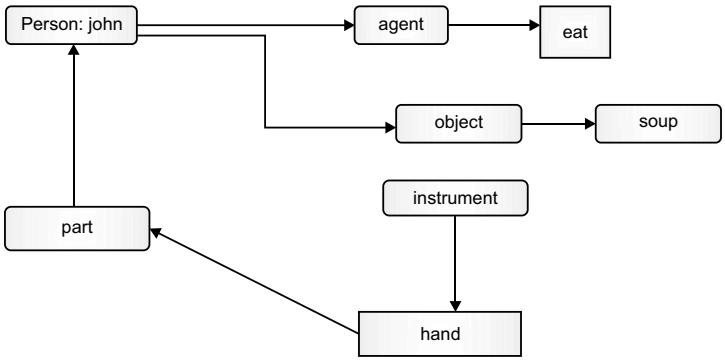


FIGURE 8.63 Conceptual Graph

Let us take an example that shows how to build a CD graph.

Consider the statement "Sally stroked her fat cat."

This sentence can be represented in a CD graph by using various building blocks of CD.

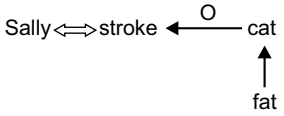
- "Sally" and "stroking" are necessary for conceptualization: there is a two-way dependency between each other:

  Sally $\Leftrightarrow$ stroke

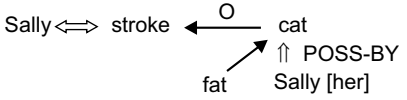
- "Sally's cat" cannot be conceptualized without the ACT (set of primitives) stroke $\Rightarrow$ it has an objective dependency on stroke.

  Sally $\Leftrightarrow$stroke $\underleftarrow{o}$ cat

- The concept "cat" is the governor for the modifier "fat:"



- The concept PP(cat) is also governed by the concept PP(Sally) through a prepositional dependency.



PP ⇔ ACT	Indicates that an actor acts.
PP ⇔ PA	indicates that an object has a certain attribute.
ACT ←O PP	Indicates the object of an action.
ACT ← R PP, PP	Indicates the direction of an object within an action.
ACT ←I ⇑	Indicates the instrumental conceptualization for an action.
X ⇑ Y	Indicates that conceptualization X caused conceptualization Y. When written with a C this form denotes that X COULD cause Y.
PP ⇐ PA2, PA1	Indicates a state change of an object.
PP1 ← PP2	Indicates that PP2 is either PART OF or the POSSESSOR OF PP1.

FIGURE 8.64 Various Dependency Rules

Let us explain some of the dependency rules.

Rule 1: PP ⇔ ACT

- It describes the relationship between an actor and the event he or she causes.
  - ♦ This is a two-way dependency, since neither actor nor event can be considered primary.
  - ♦ The letter P in the dependency link indicates the past tense.
  - ♦ Example: John ran.

$$\text{John} \quad \Leftrightarrow \quad \overset{p}{\texttt{PTRANS}}$$

Rule 2: ACT ← PP

- It describes the relationship between the ACT and a PP (object) of ACT.
  - ♦ The direction of the arrow is toward the ACT since the context of the specific ACT determines the meaning of the object relation.
  - ♦ Example: John pushed the bike.

    John ⇔ PROPEL ←° bike

Rule 3: PP ↔ PP

- This describes the relationship between two PPs, one of which belongs to the set defined by the other.
- Example: John is a doctor.

  John ↔ doctor

Rule 4: PP ← PP

- This describes the relationship between two PPs, one of which provides a particular kind of information about the other.
  - ♦ The three most common types of information to be provided in this way are possession (shown as POSS-BY), location (shown as LOC), and physical containment (shown as CONT).
  - ♦ The direction of the arrow is again toward the concept being described.
  - ♦ Example: John's dog

    dog ←*poss-by*— John

Rule 5: PP ⇔ PA

- It describes the relationship between a PP and a PA that is asserted to describe it.
- PA represents states of PP such as height and health.
- Example: John is fat.

  John ⇔ weight (> 80)

Rule 6: PP ← PA

- This describes the relationship between a PP and an attribute that already has been predicated.
  - ♦ The direction is towards the PP being described.
  - ♦ Example: Smart John

John ← smart

Rule 7: ACT ← R → PP (to); R → PP (from)

- This describes the relationship between an ACT and the source and the recipient of the ACT.
- Example: John took the book from Mary.

John ⇔ ATRANS ← R → John; R → Mary; ATRANS ↑ book

Rule 8: PP ← PA; PA

- This describes the relationship that describes the change in state.
- Example: Tree grows

Tree ← size > C; size = C

Rule 9: {x} ⇔ {x}; ⇑; {y} ⇔ {y}

- This describes the relationship between one conceptualization and another that causes it.
  - ♦ Here, {x} is causes {y} i.e., if x then y
  - ♦ Example: Bill shot Bob.

$\{x\}$: Bill shot Bob

$\Uparrow$

$\{y\}$: Bob's health is poor.

Rule 10: $\Leftrightarrow \{x\}$ $\downarrow$ $\Leftrightarrow \{y\}$

- This describes the relationship between one conceptualization and another that is happening at the time of the first.
  - ♦ Here, $\{y\}$ is happening while $\{x\}$ is in progress.
  - ♦ Example: While going home, I saw a snake.

    I am going home.

    $\downarrow$

    I saw a snake.

8.7.5.3 Conceptual Cases

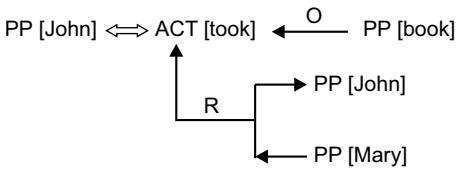
Dependents that are required by ACT are called conceptual cases:

There are four main conceptual cases:

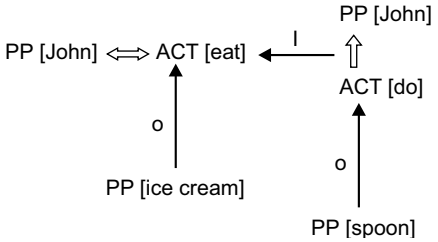
- Objective Case (O)
- Recipient Case (R)
- Instrumental Case (I)
- Directive Case Relation (D)
  - ♦ **Objective Case (O):** "John took the book."

PP [John] $\Leftrightarrow$ ACT [took] $\xleftarrow{O}$ PP [book]

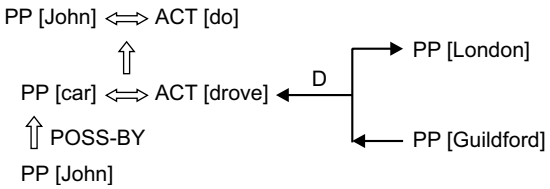
  - ♦ **Recipient Case (R):** "John took the book from Mary."



♦ **Instrumental Case (I):** "John ate the ice cream with a spoon."



♦ **Directive Case Relation (D):** "John drove his car to London from Guildford."



8.7.5.4 Prepositional Dependency

- Possession

  e.g., "This is Sally's cat."

  Cat

  ⇑ POSS-BY

  Sally

- Location

  e.g., "Sally is in London."

  London

  ⇑ LOC

  Sally

- Containment

  e.g., "The glass contains water."

  Water

  ⇑ CONT

  Glass

8.7.5.5 Primitive Acts

These are used to represent action in the world.

Table 8.4 The Primitives Set

Primitive Act	Elaboration
ATRANS	Transfer of an abstract relationship such as possession, ownership, or control (give)
PTRANS	Transfer of the physical location of an object (go)
PROPEL	Application of a physical force to an object (push)
MOVE	Movement of a body part of an animal by that animal (kick)
Grasp	Grasping of an object by an actor (grasp)
INGEST	Taking in of an object by an animal to the inside of that animal (eat)
EXPEL	Expulsion of an object from the object of an animal into the physical world (cry)
MTRANS	Transfer of mental information between animals or within an animal (tell)
MBUILD	Construction by an animal of new information of old information (decide)
CONC	Conceptualize or think about an idea (think)
SPEAK	Action of producing sounds (say)
ATTEND	Action of attending or focusing a sense organ towards a stimulus (listen)

For example: "I gave a book to Sally."

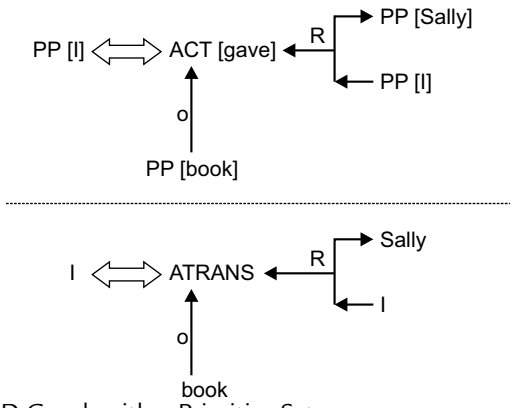


FIGURE 8.65 A CD Graph with a Primitive Set

8.7.5.6 Conceptual Tenses

Any conceptualization can be modified as a whole by a conceptual tense.

For example: John took the book. (John ⇔ took) can be denoted by looking at the lemma "take" (from which the past tense took was derived):

$$\text{John} \overset{p}{\Leftrightarrow} \text{ATRANS}$$

Table 8.5 The Symbols for Conceptual Tenses

Symbol	Elaboration
`p`	Past
`f`	Future
`t`	Transition
`ts`	Start Transition
`tf`	Finished Transition
`k`	Continuing
`?`	Interrogative
`/`	Negative
`Nil`	Present
`delta`	Timeless
`c`	Conditional

Various examples of conceptual tenses are

- "John will be taking the book."

  John ⇔ taking

  or

  f

  John ⇔ ATRANS

- "John is taking the book."

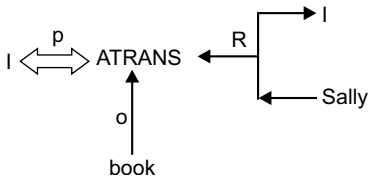
  John ⇔ taking

  or

  k

  John ⇔ ATRANS

Now, after understanding all the building blocks of CD, let us represent a sentence using all building blocks of CD: "I took a book from Sally."



Here is an explanation of the above CD graph:

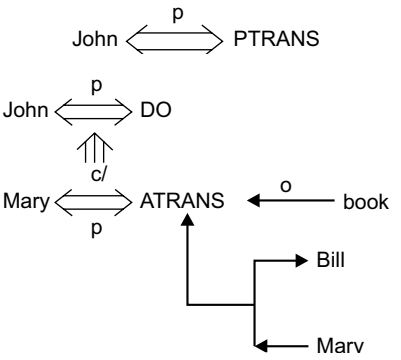
- Primitive conceptualizations (conceptual categories):
  - ♦ Objects (Picture Producers: PP): Sally, I, book
- Conceptual dependencies (diagrammatic conventions):
  - ♦ Arrows indicate the direction of dependency
  - ♦ The double arrow indicates a two-way link between the actor and action.
- Conceptual cases
  - ♦ "O" indicates object case relation
  - ♦ "R" indicates recipient case relation

- Primitive acts
  - ♦ ATRANS indicates transfer (of possession)
- Conceptual tenses

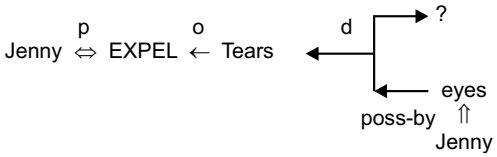
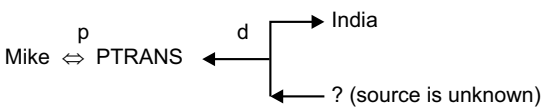
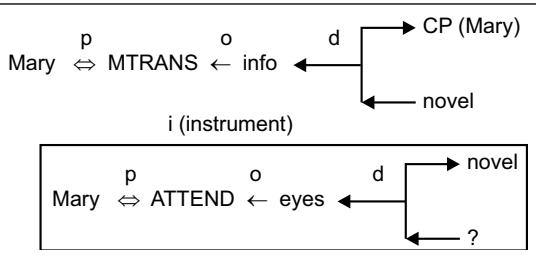
"*p*" indicates that the action was performed in the past.

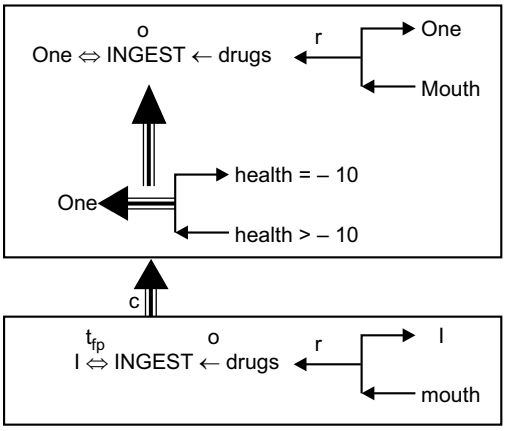
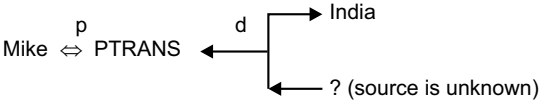
Let us represent some sentences using conceptual dependency graph

a) "John ran."

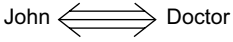


Let us see some examples of CD.

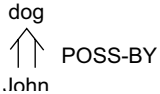
Sentences	CD Representations
Jenny cried.	
Mike went to India.	
Mary read a novel.	

Sentences	CD Representations
Since drugs can kill, I stopped taking them.	
Mike went to India.	

b) "John is a doctor."

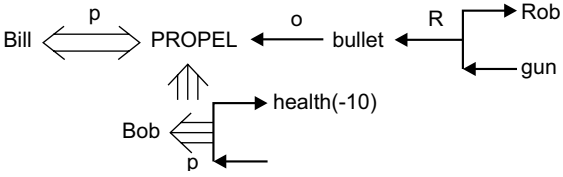


c) "John's dog"

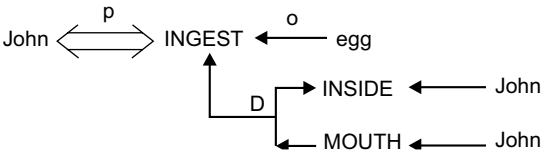


d) "John pushed the cart."

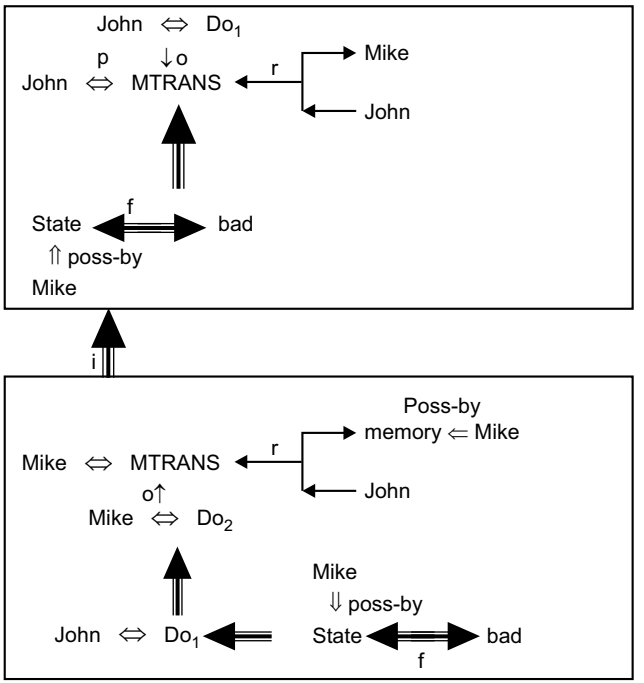
e) "Bill shot Bob."



f) "John ate the egg."



g) "John prevented Mary from giving a book to Bill."

Sentence	CD Representations
John warned Mike about the dire consequences.	

Inferences Associated with the Primitive Acts

- General inferences are stored with each primitive act, thus reducing the number of inferences that need to be stored explicitly with each concept.
- For example, from the sentence "John killed Mike," we can infer that "Mike is dead."

- Let us take another example of the primitive act INGEST.
- The following inferences can be associated with it.
  - ♦ The object ingested is no longer available in its original form.
  - ♦ If the object is eatable, then the actor has less hunger.
  - ♦ If the object is toxic, then the actor's health is bad.
  - ♦ The physical position of object has changed. So PTRANS is inferred.

Example: The verbs {give, take, steal, donate} involve a transfer of the ownership of an object.

- ♦ If any of them occurs, then inferences about who now has the object and who once had the object may be important.
- ♦ In a CD representation, these possible inferences can be stated once and associated with the primitive act ATRANS.

- Consider another sentence: "Bill threatened John with a broken nose."
  - ♦ Sentence interpretation is that Bill informed John that he (Bill) will do something to break John's nose.
  - ♦ Bill did (said) so in order that John will believe that if he (John) does some other thing (different from what Bill wanted), then Bill will break John's nose.

8.7.5.7 Advantages of CD

- The organization of knowledge in terms of the primitives (or "primitive acts") leads to fewer inference rules.
- Many inferences are already contained in the representation itself.
- The initial structure that is built to represent the information contained in one sentence will have holes in it that has to be filled in.
  - ♦ Holes serve as attention focusers for subsequent sentences.

8.7.5.8 Disadvantages of CD

- CD requires all knowledge to be broken down into the 12 primitives, which is sometimes inefficient and sometimes impossible.

- CD is essentially a theory of the representation of events: though it is possible to have an event-centered view of knowledge, it is not a practical proposition for storing and retrieving knowledge.
- It may be difficult or impossible to design a program that will reduce sentences to canonical form. (Probably not possible for monoids, which are simpler than natural language.)
- It is computationally expensive to reduce all sentences to the 12 primitives.
- It is difficult to construct the original sentence from its corresponding CD representation.
- CD representation can be used as a general model for knowledge representation, because this theory is based on a representation of events as well as all the information related to the events.
- Rules are to be carefully designed for each primitive action in order to obtain a semantically correct interpretation.
- It is difficult to
    - ♦ construct the original sentence from its corresponding CD representation.
    - ♦ CD representation can be used as a general model for knowledge representation, because this theory is based on representation of events as well as all the information related to events.
- Many verbs may fall under different primitive acts, and it becomes difficult to find the correct primitive act in the given context.
- The CD representation becomes complex, requiring a lot of storage for many simple actions.
- For example, the sentence "John bet Mike that their team will win the upcoming World Cup" would require an enormous CD structure.

Exercises

Q1. What is the relationship between belief and knowledge?

Q2. What does first-order logic (predicate logic) consist of?

Q3. What are the quantifiers in predicate logic and their meanings?

Q4. Distinguish predicate logic from propositional logic.

Q5. How would an agent use the expression `Dog(fido)` to solve a problem?

Q6. What do semantic networks represent? How?

Q7. How is knowledge given to and received from expert systems?

Q8. Name the relationships that categories may have. How may several category relationships be organized?

Q9. What is a validity-maintaining procedure for deriving sentences from other sentences in first-order logic?

Q10. What is unification used for?

Q11. Explain the different types of frames.

Q12. What is a script and what are its components?

Q13. What is meant by Conceptual Dependency?

CHAPTER 9

NEURAL NETWORKS

A conventional computer asks a programmer what it is to do, and it can be used for fast calculations. Conventional computers are not used for assisting with noisy data, managing fault tolerance or huge parallelisms, or adapting to conditions. A neural network system is used to formulate algorithmic solutions or when we need to use lots of examples of the behavior we require. Neural networks follow different examples for computing. The von Neumann machine is based on the processing/memory abstraction of human information processing. Neural networks are based on the parallel architecture of biological neurons. Neural networks are an outline of a multi-processor computer system with single processing elements, a high degree of interconnection, and communication between the elements.

Neural networks are used for complicated or imprecise data to derive the meaning. They are also used in mining patterns and engaging in learning that is too complex to be observed by either humans or other computer techniques. A trained neural network can be considered an "expert" when it comes to analyzing a category of information. This expert can then be used to provide projections, given new situations of interest, and answer "what if" questions.

It provides other features such as

- knowledge acquisition under noise and uncertainty
- flexible knowledge representation

- efficient knowledge processing
- fault tolerance
- a learning capability

9.1 Neural Networks vs. Conventional Computers

Neural networks and conventional computers complement each other. There are many different tasks that are suitable for an algorithmic approach, like arithmetic operations, and other tasks that are more suited to neural networks. Conventional computers use an algorithmic approach, but neural networks work like the human brain and learn by example.

Computer	Neural Network
algorithmic approach	learning approach
They are programmed for a specific task.	They are not programmed for a specific task.
work on a pre-defined set of instructions	used in decision-making
operations are predictable	operation is unpredictable

9.2 Neural Networks

A neural network is a parallel distributed processing system that is made by highly interconnected computing elements that are used for their ability to learn and acquire knowledge and make it available for use. It is a simplified model of the biological neuron system. A neural network is a powerful data modeling tool that is able to capture and represent complex input/output relationships. The technology that is used in a neural network performs "intelligent" tasks that are also performed by the human brain. Several classes of neural networks are used. The learning process is referred to as training and the ability to solve a problem using the knowledge acquired is called an inference.

Neural networks are simplified simulations of the central nervous system and clearly require the kind of computing that is performed by the human brain. The structural components of a human brain, the neurons,

perform computations, such as pattern recognition cognition, logical inference, and so on. Hence, the technology that is based on a simplified simulation of computing by the neurons of a brain has been defined as Artificial Neural System (ANS) technology, Artificial Neural Network (ANN), or simply, a neural network.

9.2.1 Neurons

A neuron may be defined as a device that receives many inputs and produces one output. The neuron has two modes of operation, the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire.

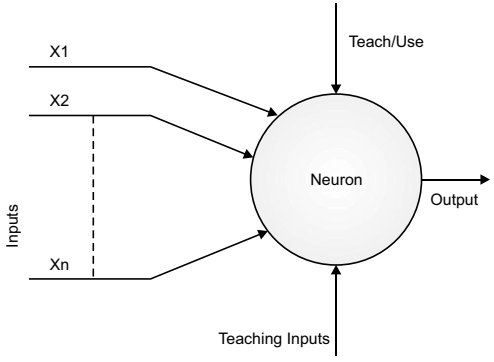


FIGURE 9.1 How a Neuron Works

Figure 9.1 represents a neuron that receives several inputs, such as $x1, x2 \ldots\ldots x_n$. When the input patterns are applied to this neuron, it takes action and produces only one output.

9.2.2 Types of Neural Networks

- A fixed network is one in which the weights cannot be changed; we can say that $dw/dt = 0$, where dw is the change in the weight with respect to time. In such a network, the weights are fixed a priori according to the problem that must be solved.
- An adaptive network is able to change its weight, i.e., dw/dt!=0.

9.2.3 Historical Background

In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons might work. They found how neurons work in the brain and are now recognized as the designers of the first neural network. They modeled a simple neural network using electrical circuits. They defined the original way in which an ANN operates and used a fixed set of weights.

In 1949, Donald Hebb wrote the paper "The Organization of Behavior." It defined the concept about how humans learn. It is the first learning rule about how humans easily learn. If two nerves fire at the same time, the connection between them is enhanced.

In 1959, Bernard Widrow and Marcian Hoff of Stanford developed models called ADALINE and MADALINE. These names come from their use of the Multiple Adaptive Linear Elements. When reading streaming bits from a phone line, it is necessary to predict the next bit. ADALINE was developed for this purpose. MADALINE was the first neural network for solving a real-world problem; it used an adaptive filter to eliminates noises on phone lines.

In 1962, Widrow and Hoff developed a learning procedure. This learning procedure was used to examine the value before the weight adjusts it according to the following rule: Weight Change = (Pre-Weight Line Value) * (Error/(Number of Inputs)). The weight can be adjusted by the value 0 or 1. Many researchers have also worked on Perceptron. In this algorithm, the neural network model needs to verify and converge to the correct weights in order to solve the problem. The weight adjustment (learning algorithm) used in Perceptron was found to be more powerful than the learning rules used by Hebb.

In the same time period, a paper was written that suggested there could not be an extension from the single-layered neural network to a multilayered neural network. In addition, many people in the field were using a learning function that was fundamentally defective because it was not differentiable across the entire line. As a result, research and funding went down drastically.

In 1972, Kohonen and Anderson developed networks that were independent of one another. To solve the problem and describe the ideas, they used the matrix concept of mathematics. However, they did not conclude the how to create an array of analog ADALINE circuits. In these networks,

neurons are used, but these neurons are supposed to activate a set of outputs instead of just one.

In 1975, the first multilayered network was developed. This network was called an unsupervised network.

In 1982, John Hopfield of Caltech presented a paper to the National Academy of Sciences. In this paper, he developed more useful machines by using bidirectional lines. In these machines, the connection between neurons was via two directions, but in earlier machines, the connections between neurons was only one way.

Also in 1982, there was a joint US-Japan conference on Cooperative/Competitive Neural Networks. Japan publicized a new Fifth Generation effort on neural networks. This fifth-generation computing involves artificial intelligence. The first generation used switches and wires, the second generation used the transistor, the third generation used solid-state technology like integrated circuits and higher level programming languages, and the fourth generation involves using code generators.

In 1986, with multiple layered neural networks in the news, the problem was how to extend the Widrow-Hoff rule to multiple layers. Three groups of researchers worked independently but came up with similar ideas and at last developed networks now called back propagation networks. Hybrid networks use just two layers, while these back-propagation networks use many.

9.3 Biological Neural Networks

A biological neural network may be defined as a collection of connected biological nerve cells. An example of a biological neural network is your brain.

The brain is mainly composed of about 10 billion neurons; each neuron is connected to about 10,000 other neurons. All of these neurons are used in processing information. It is a collection of several nerve cells. Each cell works like a simple processor. In the brain, there is significant interaction between these cells and their parallel processing, which makes the brain's abilities possible. Our entire brain is composed of these interconnected electro-chemical transmitting neurons. When these simple processing units are combined together to form extremely long units, then the brain can perform complex tasks. These processing units are used in determining the weighted sum of its input; if these inputs exceed a certain level, then a binary signal is fired to perform the task.

The neural system of the human body has three stages: receptors, a neural network, and effectors. The receptors receive the stimuli either internally or from the external world, and then pass the information to the neurons in the form of electrical impulses. The neural network then processes the inputs and makes the proper decision about outputs. Finally, the effectors translate the electrical impulses from the neural network into responses to the outside environment. Figure 9.2 shows the bidirectional communication between the stages for feedback.

Biological neurons and their components

- The primary element of the neural network is called a neuron.
- The neuron's *cell body (soma)* receives the incoming activations, processes them, and then converts them into output activations.
- The neuron's *nucleus* contains the genetic material in the form of DNA. This exists in most types of cells, not just neurons.
- Dendrites are the tree-like structures that receive the signal from the surrounding neurons, where each line is connected to one neuron. Dendrites receive activations from other neurons. The soma processes the incoming activations and converts them into output activations.
- The axon is a thin cylinder that transmits the signal from one neuron to others. At the end of axon, the contact to the dendrites is made through a synapse. The axon acts as a transmission line to send activations to other neurons.
- The junctions that allow signal transmission between the axons and dendrites are called *synapses*. Synapses are the junctions that allow signal transmissions between the axon and dendrites.
- The process of transmission is by the diffusion of chemicals called *neurotransmitters* across the synaptic cleft.

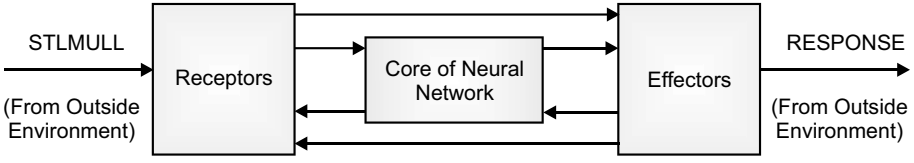
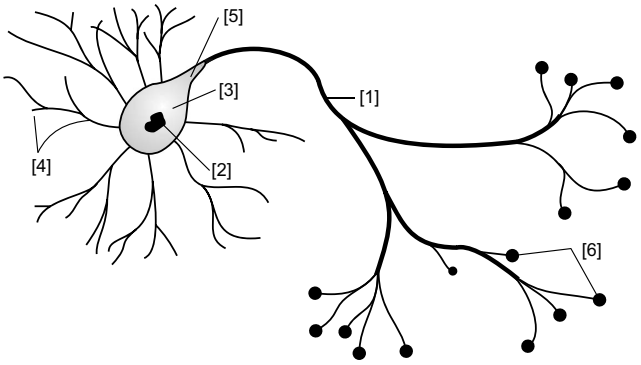


FIGURE 9.2 Three Stages of a Biological Neural System



1. Axon, 2. Nucleus, 3. Soma (Body), 4. Dendrite, 5. Axon Hillock, 6. Terminals (Synapses)

FIGURE 9.3 Biological Neural Network

9.3.1 Biological Neurons

Now, how can we explain how information flows in a neural cell? The diagram in Figure 9.4 shows the input/output and propagation flow in a neural cell. This is the structure of a neural cell in the human brain.

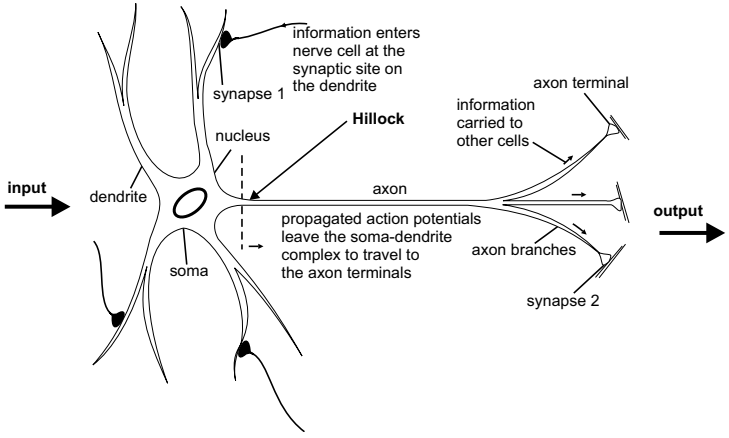


FIGURE 9.4 A Biological Neural Network

9.4 Artificial Neural Networks

A set of input connections brings in activation from other neurons. A processing unit sums the input and then applies a non-linear activation function. An output line transmits the result to other neurons.

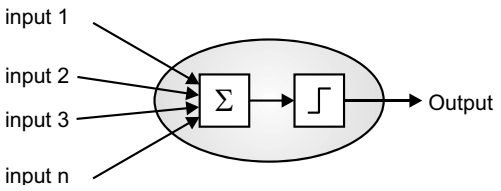


FIGURE 9.5 Artificial Neural Network

Artificial neural networks (ANNs) are based on mathematical concepts and try to imitate the structure and functionalities of biological neurons. In artificial neural network processing, elements are used besides neurons, for example, summing its inputs and applying a threshold to the result to determine the output of that "neuron." Such a model follows three simple sets of rules: multiplication, summation, and activation. When artificial neurons are entered, the input values associated with every input are multiplied with their individual weights. In the middle section of an artificial neuron, the sum function is used, which sums all the weighted inputs and bias. When artificial neurons exit, then the activation function is used, in which the sum of the previously weighted inputs and bias is passed through the activation function. This activation function is also called a transfer function. A set of input connections brings in activations from other neurons.

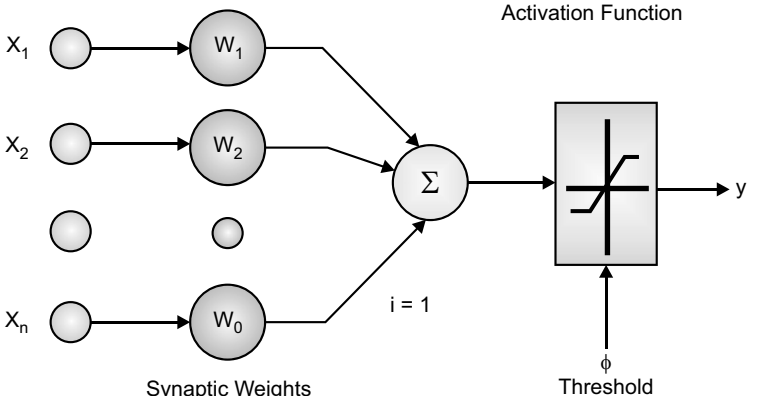


FIGURE 9.6 Basic Elements of an Artificial Linear Neuron

Functions: The function $y = f(x)$ describes a relationship and input output mapping from x to y.

Activation function controls when the unit is active or inactive. The threshold or sign function sgn(x) is defined as

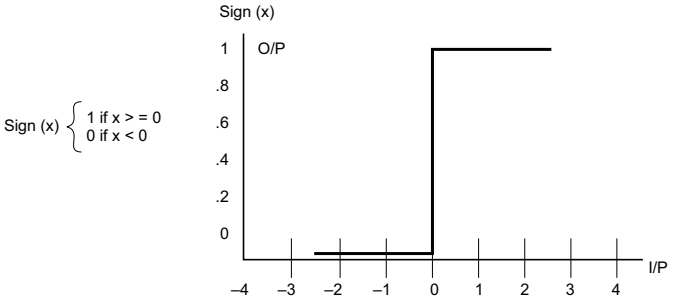


FIGURE 9.7 Sign Function

The threshold or sigmoid function sigmoid(x) is defined as a smoothed (differentiable) form of the threshold function.

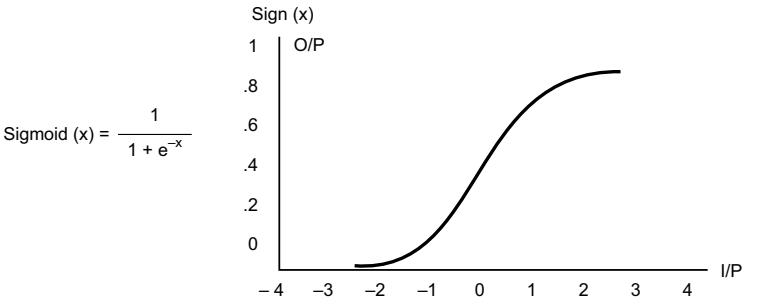


FIGURE 9.8 Sigmoid Function

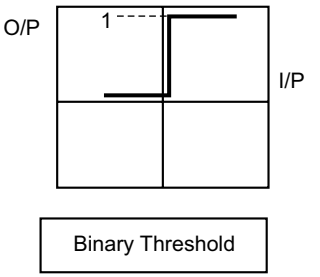
The threshold activation function is either a binary type or a bipolar type.

The output of a binary threshold will produce

- 1 if the weighted sum of the input is positive
- 0 if the weighted sum of the input is negative

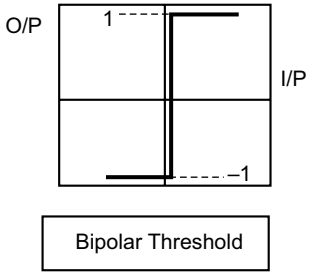
$$Y = f(I) = \{1 \text{ if } I >= 0$$

$$\{0 \text{ if } I < 0$$



Bipolar Threshold: The output of the bipolar threshold function produces

- 1 if the weighted sum of the inputs is positive
- –1 if the weighted sum of the inputs is negative



$Y = f(I) = \{1 \text{ if } I >= 0$

$\{-1 \text{ if } I < 0$

Output = sgn($\sum_{i=1}^{n}$ input $i - \phi$)

where ϕ is the neuron's activation threshold.

If $\sum_{i=1}^{n}$ input $i > \phi$ then Output = 1

if $\sum_{i=1}^{n}$ input $i < \phi$ then Output = 0

Now, we can explain an artificial neural network using an example. In this example, we have four inputs with their associated weights.

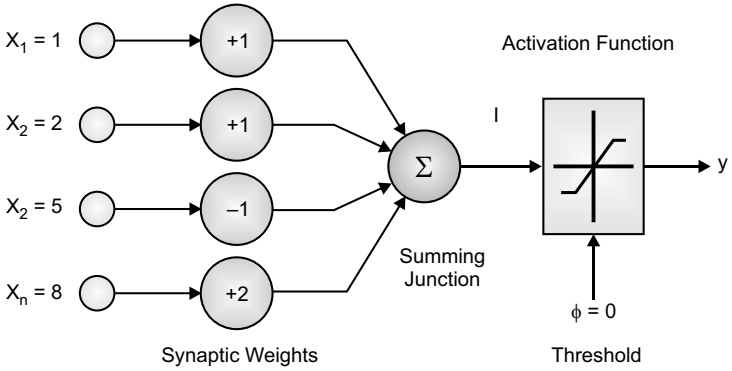


FIGURE 9.9 An Artificial Neural Network

Explanation

The output I of the network prior to the activation function stage is

$$I = X^{T}.W = [1\ 2\ 5\ 8].\begin{pmatrix} +1 \\ +1 \\ -1 \\ +2 \end{pmatrix} = 14$$

The binary activation function the output of the neuron is

$$Y(\text{threshold}) = 1$$

9.5 Differences Between Biological and Artificial Neural Networks

Human (Biological NN)	Artificial NN
Neuron	Processing Element
Dendrites	Combining Function
Cell Body	Transfer Function
Axons	Element Output
Synapses	Weights

9.6 Architecture of a Neural Network

A neural network is a data processing system that consists of a large number of processing elements that are highly interconnected with each other, such as artificial neurons in a network structure that can be represented using a directed graph G, an ordered 2-tuple (V, E) consisting of a set V of vertices, and a set E of edges.

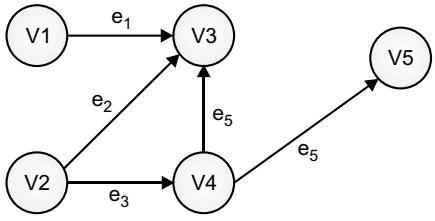


FIGURE 9.10 Directed Graph

The vertices may represent neurons (input/output) and the edges may represent synaptic links labeled by the weights attached.

Figure 9.10 shows a directed graph with the following:

Vertices $V = \{v1, v2, v3, v4, v5\}$

Edges $E = \{e1, e2, e3, e4, e5\}$

A neural network can be classified according to the following architectures:

- **Connection types:** The neural network architecture can consist of two types
  - ♦ static (feed-forward)
  - ♦ dynamic (feed-backward)
- **Topology:** Three types of neural network architecture are as follows:
  - ♦ single layer
  - ♦ multilayer
  - ♦ recurrent
- **Learning methods:** The three types of neural network architecture are
  - ♦ supervised
  - ♦ unsupervised
  - ♦ reinforcement

9.6.1 Single Layer Feed-Forward Networks

A single-layer feed-forward network consists of a single layer of weights, where the inputs are directly connected to the outputs using a series of weights. In this network, synaptic links that represent the edges are used to carry the weights from every input to every output, but not every output to every input. This way, it is considered a network of the feed-forward type. At each neuron node, three sets of rules are applied. First, the sum of the products of the weights and the inputs is calculated. Then, all values are checked to determine the activation function. If the value is above some threshold (0), the neuron fires and takes the activated value (typically 1); otherwise, it takes the deactivated value (typically -1).

Feed-forward ANNs permit signals to travel one way only, from input to output. There is no feedback (loops), i.e., the output of any layer does not affect that same layer. Feed-forward ANNs are likely to be straight forward networks that correlate inputs with outputs. This type of organization is also referred to as bottom-up or top-down.

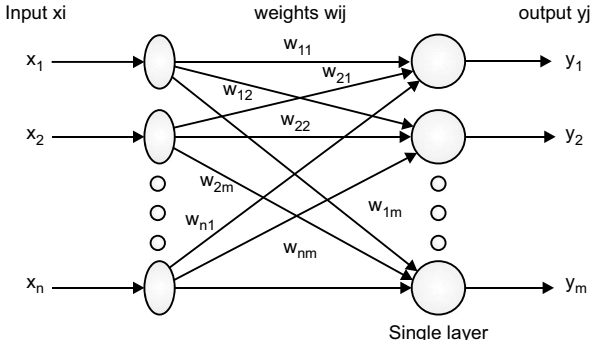


FIGURE 9.11 A Single-Layer Feed-Forward Network

9.6.2 Multilayer Feed-Forward Network

As the name suggests, a multilayer feed-forward network consists of multiple layers. The architecture of this class of network, besides having the input and the output layers, and also has one or more intermediary layers called *hidden layers*. In this network, input layers are connected with hidden layers and then these feed into several output layers. The computational units of the hidden layers are known as *hidden neurons*.

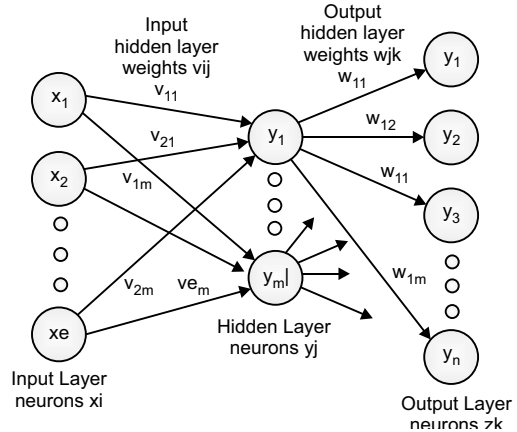


FIGURE 9.12 Multilayer Feed-Forward Network

9.6.3 Recurrent Networks

A recurrent network differs from the feed-forward architecture. A recurrent network has at least one feedback loop. There could be neurons with self-feedback links; that is, the output of a neuron is the feedback into itself as input. In a recurrent network, each processing element may be any one of two states, either black active or white inactive. A positive weighted connection indicates that two units tend to activate each other. A negative connection allows an active unit to deactivate from a neighboring unit.

Example:

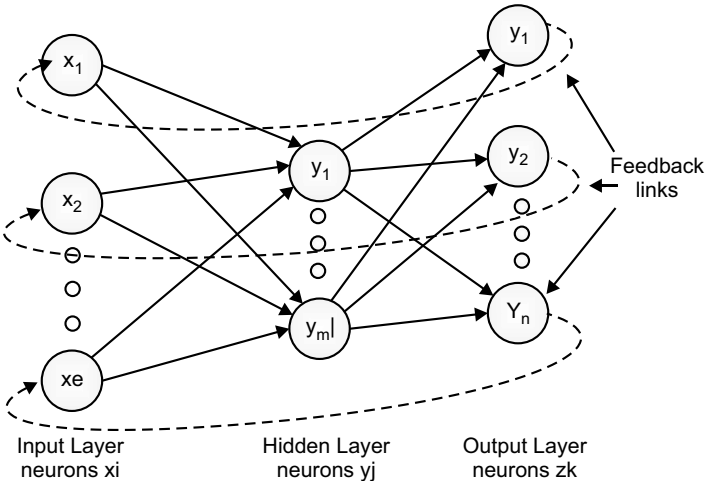


FIGURE 9.13 A Recurrent Network

9.6.4 Feedback Networks

Feed-forward networks are used when the signal travels only in one direction. When the signal needs to travel in both directions, feedback networks are used by introducing loops in the network. Feedback networks are very powerful and can get vastly complicated. Feedback networks are dynamic; their "state" is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote the feedback connections in single-layer organizations.

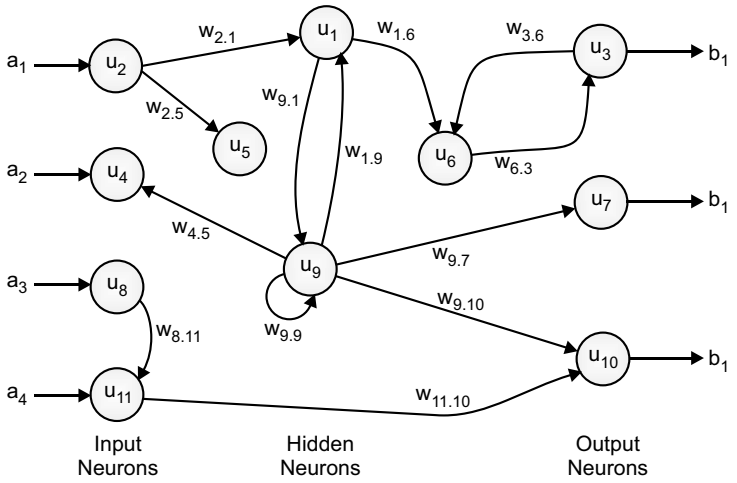


FIGURE 9.14 An Example of a Complicated Network

9.6.5 Network Layers

There are three types of network layers used in network architecture. These layers are also known as groups or units. The first layer is the input layer, which is connected to a layer of hidden units, which is connected to a layer of output units. The processing of these layers is as follows:

The movement of the input units represents the raw information that is fed into the network. The movement of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units. The behavior of the output units depends on the activity of the hidden units and the weights between the hidden and output units. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish between single-layer and multilayer architectures. The single-layer organization, in which all units are connected to one another, constitutes the most general case and has a greater potential computational power than hierarchically structured multi-layer organizations. In multilayer networks, units are often numbered by layer, instead of following global numbering.

Exercises

Q1. What is the use of neural networks in AI?

Q2. What are the advantages of neural networks over conventional computers?

Q3. Explain the neural network in detail.

Q4. Discuss a neuron using a diagram.

Q5. What are the types of neural networks?

Q6. Explain the workings of a biological neural network.

Q7. How does information flow in an artificial neural network?

Q8. Differentiate between a biological neural network and an artificial neural network.

Q9. Explain all the types of architecture of neural networks.

Q10. Define complicated neural networks using a suitable diagram.

CHAPTER 10

THE LEARNING PROCESS

There are various types of learning in neural networks that allow neural networks to predict relationships and patterns.

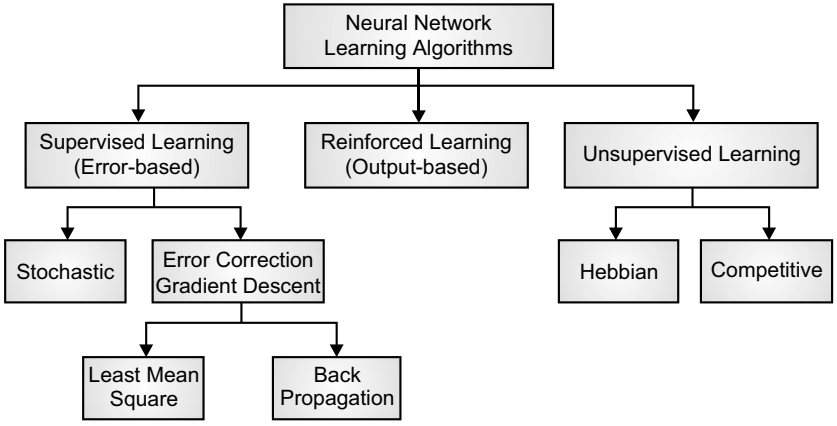


FIGURE 10.1 Learning Algorithm Classification

10.1 Types of Learning in a Neural Network

10.1.1 Supervised Learning

In supervised learning, a teacher is present during the learning process. This machine learning technique sets the parameters of an artificial neural network from the training data. The function of this learning technique

is to set the values for any valid input value after having seen the output value. Each output unit is told what its desired response to the input signal ought to be by an external teacher. During the learning process, global information may be required. The errors that are generated are used to change the network parameters, resulting in improved performance. Paradigms of supervised learning include error correction learning, reinforcement learning, and stochastic learning. In supervised learning, an important issue is the minimization of the error between the desired and computed unit values, referred to as the problem of error convergence. The aim is to determine a set of weights that minimize the error. One well known method, which is common to many learning paradigms, is the Least Mean Square (LMS) convergence. Supervised learning can also be referred to as classification, where we have a broad range of classifiers, such as the artificial neural networks' Methodological Advances and Biomedical Applications suitable classifiers (like the multilayer perceptron, Support Vector Machines, k-nearest neighbor algorithm, Gaussian mixture model, Gaussian, naive Bayes, decision tree, and radial basis function classifiers). In order to solve a given problem in supervised learning, various steps must be considered.

In the first step, we have to determine the type of training examples. In the second step, we need to gather a training data set that satisfactory describes a given problem. In the third step, we need to describe the training data set in a form that is understandable to the chosen artificial neural network. In the fourth step, we do the learning, and after the learning, we can test the performance of the learned artificial neural network with the test (validation) data set. The test data set consists of data that has not been introduced to the artificial neural network while learning.

10.1.1.1 Stochastic

In the stochastic method, the weights are adjusted using a probabilistic method. An example is simulated annealing, which is a learning mechanism employed by Boltzman.

10.1.1.2 Error Correction Gradient Descent

The error correction gradient descent is based on the minimization of the error, E, defined in terms of weights and the activation function of the network. The activation function of the network must be differentiable,

because the update of the weights is dependent on the gradient of the error, E. If *wij* is the weight update of the link connecting to the *i*th and *j*th neuron of the two neighboring layers, then it is defined as

$$\Delta wij = \eta(\partial E/\partial wij)$$

where ή the is learning rate parameter and the error gradient with reference to the weight *wij*.

10.1.2 Unsupervised Learning

In unsupervised learning, no teacher is present. The expected or desired output is not presented to the network. Unsupervised learning is a machine learning technique that sets the parameters of an artificial neural network based on the given data and a cost function, which is to be minimized. The cost function can be any function, and it is determined by the task formulation. It uses no external teacher and is based upon only local information. It is also referred to as being self-organizing in the sense that it organizes data presented to the network and detects their developing collective properties. The problem with unsupervised learning is that it is mostly used in applications that are related to estimation problems, such as statistical modeling, compression, filtering, blind source separation, and clustering. In unsupervised learning, we seek to determine how the data is organized. It differs from supervised learning and reinforcement learning in that the artificial neural network is given only unlabeled examples. One common form of unsupervised learning is clustering, where we try to categorize data in different clusters by their similarity.

Hebbian Learning: Hebb proposed a rule based on the correlative weight adjustment. In this rule, the input/output pattern pairs (*Xi*, *Yi*) are associated by the weight matrix w, known as the correlation matrix, and are computed as

$$\mathrm{w} = \sum_{i=0}^{n} X_i Y_i$$

where Yi^T is the transpose of the associated output factory.

Competitive Learning: In this method, those neurons that respond strongly to the input stimuli have their weights updated. When an input pattern is presented, all neurons in the layer compete, and the winning neuron undergoes a weight adjustment. This strategy is called the "winner-takes-all."

10.1.3 Reinforcement Learning

Reinforcement learning is a machine learning technique that sets the parameters of an artificial neural network, where the data is usually not given, but is instead generated by interactions with the environment. Reinforcement learning is concerned with how an artificial neural network ought to take action in an environment so as to maximize some notion of a long-term reward. Reinforcement learning is frequently used as a part of an artificial neural network's overall learning algorithm.

After the return function that needs to be maximized is defined, reinforcement learning uses several algorithms to find the policy that produces the maximum return. A naive brute force algorithm in the first step calculates the return function for each possible policy and chooses the policy with the largest return. The obvious weakness of this algorithm occurs when there is an extremely large or even infinite number of possible policies. This weakness can be overcome by the value function approaches or direct policy estimation. The value function approaches attempt to find a policy that maximizes the return by maintaining a set of estimates of expected returns for one policy, usually either the current or the optimal estimates. These methods converge to the correct estimates for a fixed policy and can also be used to find the optimal policy.

Like the value function approaches, the direct policy estimation can also find the optimal policy. It finds it by searching it directly in the policy space, which greatly increases the computational cost.

Reinforcement learning is particularly suited to problems that include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including the introduction to the artificial neural networks of robotic control, telecommunications, games such as chess, and other sequential decision-making tasks.

10.2 Perceptron

One type of ANN is based on the unit called a perceptron. It takes a vector of real-value inputs, calculates a linear combination of these inputs, and then outputs 1 if the result is greater than some threshold and - 1 otherwise. Perceptron is a term invented by Frank Rosenblatt. The perceptron is a model in which a neuron with weighted inputs is used with some additional, fixed pre-processing. This model is known as the MVC. In the perceptron

model, units labelled A1, A2, Aj, and Ap are called association units and their task is to extract a specific feature from the input images. They are mainly used in pattern recognition, even though their capabilities extend to a lot more.

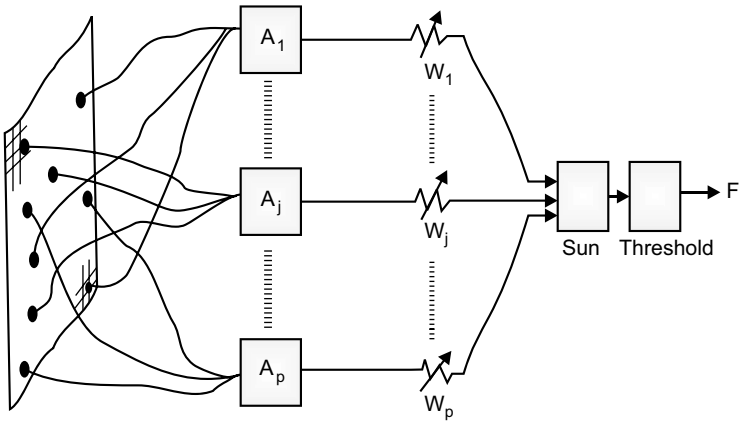


FIGURE 10.2 How a Perceptron Works

10.2.1 The Representational Power of a Perceptron

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances. The perceptron outputs a 1 for points lying on one side of the hyperplane and outputs a -1 for points lying on the other side. The equation for this hyperplane is w.x=0. Some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called linearly separable sets of examples.

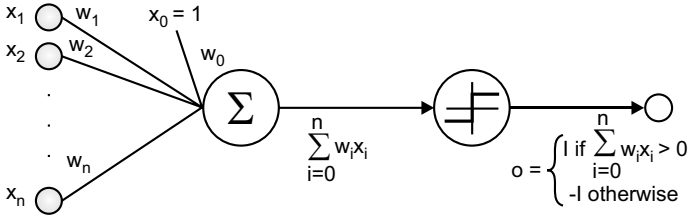


FIGURE 10.3 The Power of a Perceptron

Perceptron Rules: How can we learn the weight for a single perceptron? There are many learning approaches to help determine the weight vector that causes a perceptron to produce the correct +_1 output for each member of a given training example.

10.3 Backpropagation Networks

A backpropagation network is a general-purpose learning algorithm, which is powerful but expensive in terms of its computational requirements for training. To train a network using backpropagation involves three stages:

- feed forward of the input training network
- back propagation of the associated error
- adjustments of weight

In the feed-forward stage, every input unit receives an input signal and sends the signals to each of the hidden units. Each output unit computes its activation to form the response of the network for the given input pattern.

While training, each output unit compares its computed activation with its target value to determine the error associated with the pattern with that unit. The mathematical basis for the backpropagation algorithm is the optimization technique known as the gradient descent.

Choice of Activation Functions

An activation function for a backpropagation network has several important characteristics. It should be continuous, differentiable, and monotonically non-decreasing. It is very advantageous if its derivative is easy to compute.

10.4 Advantages of Neural Networks

- A neural network can perform tasks that a linear program cannot.
- A neural network learns by example and does not need to be reprogrammed.
- It can be implemented in any application and without any problems.
- When an element of a neural network fails, it can continue functioning because of its parallel nature.
- Neural networks are powerful so they can model complex functions.
- They can handle noisy and missing data.

- They provide general solutions with good predictive accuracy.
- They involve human-like thinking.
- They can work with large numbers of variables or parameters.
- They deal with non-linearity in the world in which we live.

10.5 Limitations of Neural Networks

- **Neural networks are too much of a black box.**

  They are considered black box technology, since the knowledge of their internal workings is never known. It is challenging to determine how they are solving a problem, because they are opaque. Neural networks are difficult to troubleshoot when they don't work as you expect, and when they do work, you will never really feel confident that they will generalize well to data not included in your training set because, fundamentally, you don't understand how your network is solving the problem.

- **Neural networks are not probabilistic.**

  For the most part, neural networks have few, if any, probabilistic underpinnings, unlike their more statistical or Bayesian counterparts. It's extremely useful to know how confident your classifier is about its answers because that information allows you to better manage the cost of making errors by tuning your classifier. A neural network might give you a continuous number as its output (e.g., a score), but translating that into a probability is often difficult.

- **Neural networks are not a substitute for understanding the problem deeply.**

  Applying a neural network for a human-related problem requires time. You need to invest extra time in studying. You must analyze your data first and then choose the best techniques that will allow you to have the confidence that they will work well for your problem, instead of investing your time throwing a neural net at the problem.

- They are just approximation of a desired solution and errors are expected.

10.6 Applications of Neural Networks

- **Character Recognition**

  Character Recognition is used in mapping the matrix of pixels into characters and words. Artificial neural network theories are used in performing character recognition. Character recognition is important for handheld devices. Several types of characters can be used, but neural networks can be used to recognize handwritten characters. In a neural network, expert knowledge can be defined into the architecture to reduce the number of parameters determined by the training by examples. Neural networks provide this type of flexibility.

- **Image Compression**

  Neural networks are also used in image compression because they can receive and process huge amounts of information at once. With the explosion of the Internet and more sites using more images, using neural networks for image compression is worth a look.

  In images, some redundant information is used. Data compression techniques are used to remove this information. These images then require less storage space and less time to transmit, so neural nets can be used.

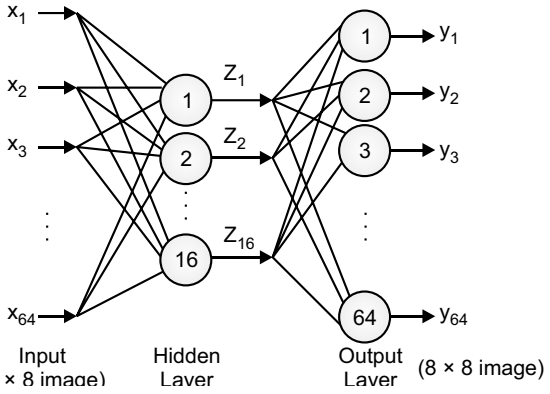


FIGURE 10.4 A Neural Network using Image Compression

A neural net architecture can be used to solve the image compression problem. This architecture is represented in Figure 10.5. In this type of structure, many input layers are feeding into a small hidden layer,

which then feeds into a large output layer. This type of structure is referred to as a bottleneck type network. The idea behind the design of this type of architecture is as follows. Suppose that it had been trained to implement an identity map. Then, a tiny image existing on the network as input would appear exactly the same at the output layer.

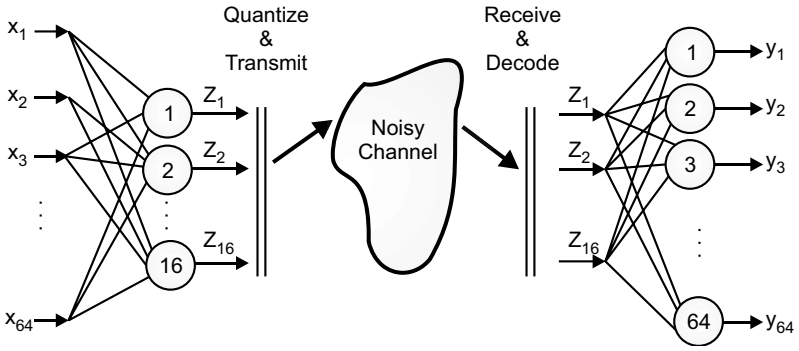


FIGURE 10.5 Neural Net Archeticture for the Image Compression Problem.

Now this neural net architecture can be broken into two parts so that image compression techniques can be used. The first one is the transmitter and the second one is the decoder. The use of a transmitter encodes and then transmits the output of the hidden layer (only 16 values, as compared to the 64 values of the original image). The receiver receives and decodes the 16 hidden outputs and generates the 64 outputs. We already explained that this neural net architecture is used to implement an identity map, and the output at the receiver is an exact reconstruction of the original image.

- **Stock Market Predictions**

  Predicting the stock market is enormously complicated. There are many factors that affect the stock market: sometimes it rises and sometimes it falls. Neural networks can be used to predict stock prices by examining a lot of information quickly. This problem can be sorted out quickly via neural network.

- **Traveling Salesman's Problem**

  Neural networks can solve the traveling salesman problem, but only to a certain degree of approximation.

- **Business Applications**

  Neural networks are used in a variety of business applications. Neural networks are used in major fields of business applications, such as financial operations, enterprise planning, trading, business analytics, and product maintenance. Neural networks are also used in forecasting and marketing research solutions. Neural networks can be applied gainfully by all kinds of traders, so if you're a trader and you haven't yet been introduced to neural networks, we'll take you through this method of technical analysis and show you how to apply it to your trading style.

- **Diagnostics Systems**

  In diagnostic systems, ANNs are used to detect heart problems and cancer. Artificial neural networks are used because they are not affected by factors such as fatigue, working conditions, and emotional state.

- **Biochemical Analysis**

  Neural networks are also used in an extensive variety of critical chemistry applications. In medicine, ANNs have been used to analyze blood and urine samples, track glucose levels in diabetics, determine ion levels in body fluids, and detect pathological conditions, such as tuberculosis.

- **Image Analysis**

  Artificial neural networks are used to analyze medical images. The use of image analysis includes many areas, such as MRI (Magnetic Resonance Images), classification of x-rays, tumor detection, determination of skeletal age, and the determination of brain maturation.

- **Other Applications (Medicine, Electronic Nose, Security, and Loan Applications)**

  There are some applications that are in their proof-of-concept stage, with the exception of a neural network that can decide whether to grant a loan. The neural network is making this decision more successfully than many humans.

- **Pattern Recognition**

  Pattern recognition involves categorizing input data into certain classes by the use of important feature attributes of the data (sample),

where the feature attributes are extracted from a background of irrelevant detail. Neural networks are used in pattern recognition because of their ability to learn and store knowledge.

Clustering: A neural network is used in clustering techniques. It is used in grouping the same types of clusters. The best applications include data compression and data mining.

Function Approximation: The function approximation is used to find an estimate of unknown functions subject to noise. It is used in several scientific and engineering regulations.

Exercises

Q1. What is the transfer function in an artificial neural network? Explain the types of ANNs.

Q2. Discuss the learning process in a neural network.

Q3. Write the back propagation algorithm in detail.

Q4. What are the pros and cons of neural networks?

Q5. Discuss the various applications of neural networks.

CHAPTER 11

FUZZY LOGIC

11.1 Introduction to Fuzzy Logic

Fuzzy or multivalued logic was introduced in the 1930s by Jan Lukasiewicz. This kind of logic extended the range of truth values to all real numbers in the interval between 0 and 1, e.g., it addressed ideas such as the possibility that a man 181 cm tall (a "tall" man) might to be set to a value of 0.86. This led to the inexact reasoning theory called the possibility theory.

Jan Lukasiewicz was a Polish logician and philosopher (Lukasiewicz, 1930). He studied the mathematical representation of fuzziness based on terms such as "tall," "old," and "hot." While classical logic operates with only two values, 1 (true) and 0 (false), Lukasiewicz introduced logic that extended the range of truth values to all real numbers in the interval between 0 and 1. He used a number in this interval to represent the possibility that a given statement was true or false. For example, the possibility that a man 181 cm tall is really tall might be set to a value of 0.86. (In this case, it is likely that the man is considered tall.)

Later, in 1937, Max Black, a philosopher, published a paper called "Vagueness: An Exercise in Logical Analysis" (Black, 1937). In this paper, he argued that a continuum implies degrees. Imagine, he said, a line of countless chairs. At one end is a Chippendale. Next to it is a near-Chippendale, in fact, one that is indistinguishable from the first item. Succeeding chairs are less and less chair-like, until the line ends with a log. When does a chair become a log? The concept "chair" does not permit us to draw a clear line distinguishing "chair" from "not-chair." Black stated that if a continuum is

discrete, a number can be allocated to each element. This number indicates a degree. But the question is "A degree of what?" Black used the number to show the percentage of people who would call an element in a line of chairs a chair; in other words, he accepted vagueness as a matter of probability. However, Black's most important contribution was in the paper's appendix. There, he defined the first simple fuzzy set and outlined the basic ideas of fuzzy set operations.

In 1965, Lotfi Zadeh published his famous paper "Fuzzy Set." Zadeh extended the work on possibility theory into a formal system of mathematical logic and introduced new logic. This new logic for representing and manipulating fuzzy terms was called fuzzy logic. The concept of fuzzy logic (FL) was conceived by Zadeh while he was a professor at the University of California at Berkley, and it was presented not as a control methodology, but as a way of processing data by allowing partial set membership rather than crisp set membership or non-membership. This approach to set theory was not applied to control systems until the 1970s due to insufficient computer capabilities prior to that time. Professor Zadeh reasoned that people do not require precise, numerical information input, and yet they are capable of highly adaptive control. If feedback controllers could be programmed to accept noisy, imprecise input, they would be much more effective and perhaps easier to implement. Unfortunately, US manufacturers have not been so quick to embrace this technology while the Europeans and Japanese have been aggressively building real products around it.

Fuzzy logic provides mathematical rules and functions that permit natural language queries. Fuzzy logic provides a means of calculating the intermediate values between the absolute true and absolute false, with the resulting values ranging between 0.0 and 1.0. Fuzzy logic calculates the shades of gray between black/white and true/false.

Fuzzy logic is a super set of conventional (or Boolean) logic that has been extended to handle the concept of partial truth and contains similarities and differences with Boolean logic. Fuzzy logic is similar to Boolean logic in that Boolean logic results are returned by fuzzy logic operations when all fuzzy memberships are restricted to 0 and 1. Fuzzy logic differs from Boolean logic in that it is permissive of natural language queries and is more like human thinking; it is based on degrees of truth.

Truth values (in fuzzy logic) or membership values (in fuzzy sets) belong to the range [0, 1], with 0 being absolute falseness and 1 being absolute truth. Values between 0 and 1 deal with real world vagueness.

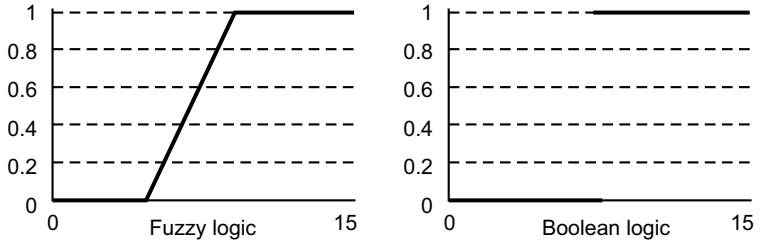


FIGURE 11.1 Difference between Fuzzy and Boolean Logic

A fuzzy set is a set whose elements have degrees of membership. That is, a member of a set can be a full member (100% membership status) or a partial member (e.g., less than 100% membership and greater than 0% membership). To fully understand fuzzy sets, one must first understand traditional sets.

Fuzzy logic deals with those imprecise conditions about which a true/false value cannot be determined. Much of this has to do with the vagueness and ambiguity that can be found in everyday life. For example, the question "Is it hot outside?" probably would lead to a variety of responses from those asked. These are often labeled as subjective responses, where no one answer is exact. Subjective responses are relative to an individual's experience and knowledge. Human beings are able to exert this higher level of abstraction during the thought process. For this reason, fuzzy logic has been compared to the human decision-making process. Conventional logic (and computing systems, for that matter) is by nature related to Boolean conditions (true/false). What fuzzy logic attempts to encompass is that area where a partial truth can be established, that is, a gradient within the true/false realm.

11.1.1 Definition of Fuzzy Logic

Fuzzy logic is a set of mathematical principles for knowledge representation based on the degrees of membership rather than on the crisp membership of classical binary logic. Fuzzy logic is a logic that is used to describe fuzziness or vagueness. Fuzzy logic is based on the idea that all things can be viewed as having degrees, for example, "The motor is running really hot."

Fuzzy logic is a problem-solving control system methodology that is used in systems ranging from simple, small, embedded micro-controllers to large, networked, multi-channel PC or workstation-based data acquisition and control systems. It can be implemented in hardware, software, or a combination of both. Fuzzy logic provides a simple way to arrive at a definite conclusion based upon vague, ambiguous, imprecise, noisy, or missing input information. Fuzzy logic's approach to control problems mimics how a person would make decisions, only much faster.

Fuzzy logic is a form of multi-valued logic that deals with inexact reasoning. Computers can apply this logic to represent vague and imprecise ideas, such as "hot," "tall," or "balding." Fuzzy logic variables have range between 0 and 1, and they are based on the degree of truth rather than the true-or-false Boolean logic. This range includes 0 and 1 as the extreme cases of truth, but also includes various states of truth in between. For example, the result of a comparison between two things would not be tall or short, but 0.38 of tallness.

Fuzzy theory provides a mechanism for representing linguistic constructs such as many, low, medium, or often.

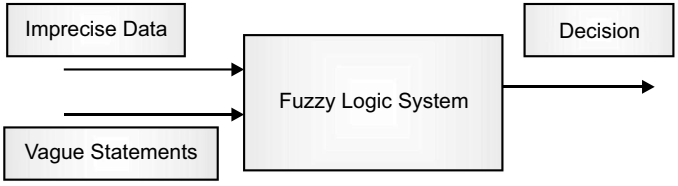


Figure 11.2 A Fuzzy Logic System that Accepts Imprecise Data and Vague Statements such as Low or Medium

11.1.2 Features of Fuzzy Logic

- In fuzzy logic, exact reasoning is viewed as a limiting case of approximate reasoning.
- In fuzzy logic, everything is a matter of degree.
- It is suitable for approximate reasoning.
- It allows decision-making with estimated values using uncertain information.

- It is robust as it does not require precise, noise-free inputs and can be programmed to fail safely if a feedback sensor is destroyed.
- Any reasonable number of inputs can be processed and numerous outputs can be generated.
- Fuzzy logic is not limited to a few feedback inputs. Any sensor data that provides some indication of a system's action and reaction is sufficient. This allows sensors to be inexpensive and imprecise, which makes the overall system cost and complexity low.
- Fuzzy logic can control non-linear systems that would be difficult or impossible to model mathematically.

11.1.3 Advantages of Fuzzy Logic

- Fuzzy logic is easy to understand, test, and maintain.
- It is flexible, meaning it is easy to layer on more functionality without starting from scratch.
- It can tolerate imprecise data.
- It can model the non-linear functionality of arbitrary complexity.
- It is based on natural language.
- It is robust, meaning can operate when there is a lack of rules.
- It allows for rapid computation due to its parallel evaluation nature.

11.1.4 Disadvantages of Fuzzy Logic

Fuzzy logic has disadvantages:

- It needs a lot of tests and evaluation.
- It does not learn easily.
- It is highly abstract and heuristic.
- It has no precise mathematical model.
- It lacks the self-organization and self-tuning mechanism of a neural network.
- It is difficult to establish correct rules.

11.2 Crisp Set (Classical set)

The concept of a set is fundamental to mathematics.

Let X be the universe of discourse, and its element is denoted as x. In classical set theory, a crisp set A of X is defined as the function $F_{A(x)}$, that is

$$F_{A(x)} : X \rightarrow \{0,1\} \text{ where } \begin{cases} F_{A(x)} = 1 \; if & x \in A \\ 0 \; if & x \notin A \end{cases}$$

For any element x of X, $F_{A(x)}$ is equal to 1 if x is an element of set A and equal to 0 if x is not an element of

A. Classical sets are called crisp sets: either an element belongs to a set or not, i.e.,

$x \in A$ OR $x \notin$ A

In classical set theory, the membership of elements in a set is assessed in binary terms according to a bivalent condition: an element either belongs or does not belong to the set. For example, for the set of integers, either an integer is even or it is not (it is odd). Classical sets are also called crisp (sets).

Lists A = {apples, oranges, cherries, mangoes}

$A = \{a_1, a_2, a_3\}$

A = {2, 4, 6, 8, …}

Formulas A = {x | x is an even natural number}

A = {x | $x = 2n$, n is a natural number}

One more example is as follows:

P : the set of all people

Y : the set of all young people

Young=$\{y|y = \text{age}(x) \leq 25, x \in P\}$

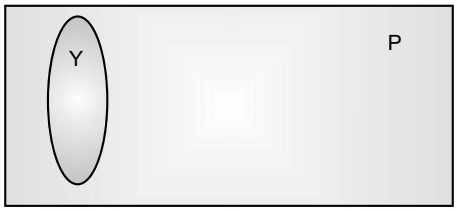


FIGURE 11.3 A Pictorial Representation of a Classical Set

11.3 Fuzzy Set

Fuzzy set theory is different from classical set theory, as the elements of a fuzzy set admit some degree of membership, meaning how much an element belongs to set (for example, the set of tall men). What does it mean to be tall? Height is all relative. As a descriptive term, "tall" is very subjective and relies on the context in which it is used. Even a 5' 7" man can be considered "tall" when he is surrounded by people shorter than he is. It is impossible to give a classical definition for the subset of tall men. However, we could establish to which degree a man can be considered tall. This can be done using membership functions ($\mu A(x)$):

- $\mu A(x) = y$
  - ♦ an individual x belongs to some extent ("y") to subset A
  - ♦ y is the degree to which the individual x is tall
- $\mu A(x) = 0$
  - ♦ Individual x does not belong to subset A
- $\mu A(x) = 1$
  - ♦ Individual x definitely belongs to subset A

Some Points About Fuzzy Sets

- Fuzzy sets are sets whose elements have degrees of membership.
- Fuzzy sets were introduced by Lotfi A. Zadeh (1965) as an extension of the classical notion of a set.
- Fuzzy set theory permits the gradual assessment of the membership of elements in a set; this is described with the aid of a membership function valued in the real unit interval [0,1].
- The fuzzy set theory can be used in a wide range of domains in which information is incomplete or imprecise, such as bioinformatics.
- They are extensions of classical sets.
- They use not just the membership values of "in the set" and "out of the set," 1 and 0, but also partial membership values between 1 and 0.

A fuzzy set can be defined mathematically by assigning to each possible individual in the universe of discourse a value representing its grade of membership in the fuzzy set.

For example, a fuzzy set representing our concept of "sunny" might assign a degree of membership of 1 to a cloud cover of 0%, 0.8 to a cloud cover of 20%, 0.4 to a cloud cover of 30%, and 0 to a cloud cover of 75%.

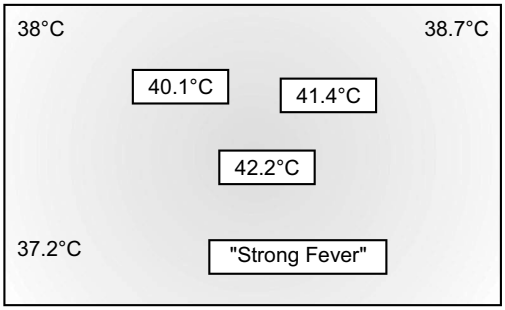


FIGURE 11.4 A Fuzzy Set

Definition of a Fuzzy Set

A fuzzy set is a pair (U, m) where U is a set and $m : U \rightarrow [0, 1]$.

For each $x \in U$, the value $m(x)$ is called the **grade** of membership of x in (U, m).

For a finite set $U = \{x_1, ..., x_n\}$, the fuzzy set (U, m) is often denoted by

$\{m(x_1)/x_1, ..., m(x_n)/x_n\}$.

Let $x \in U$. This called **not included** in the fuzzy set (U, m). If $m(x) = 0$, x is called **fully included.** If $m(x) = 1$, it is called a fuzzy member if $0 < m(x) < 1$.

The set $\{x \in U | m(x) > 0\}$ is called the **support** of (U, m), and the set $\{x \in U \mid m(x) = 1\}$ is called its **kernel**. The function m is called the **membership function** of the fuzzy set (U, m).

OR

Fuzzy set A of universe X is defined by $\mu_A(x)$, called the membership function of set A.

$\mu A(x) : X \rightarrow \{0, 1\}$ where $\mu A(x) = 1$ if x is totally in A,

$\mu_A(x) = 0$ if x is not in A, and

$0 < \mu A(x) < 1$ if x is partly in A.

For any element x of universe X, the membership function $\mu_{A(x)}$ equals the degree to which x is element of set A. These values between 0 and 1 represent degree of membership.

OR

If U is a collection of objects denoted generically by x, then a fuzzy set A in U is defined as a set of ordered pairs:

$$A = \{(x, \mu_A(x)) | \, x \in U\}$$

where $\mu_A : U \rightarrow [0, 1]$

We have seen that a fuzzy set A of a set $X(\neq \phi)$ is characterized by the membership function:

$$\mu A : X \rightarrow [0, 1]$$

Thus, for any $x \in X$, the degree of belongingness of it in the fuzzy set A is $\mu_A(x)$, when $(0 < \mu_A(x) < 1)$.

If $X = \{x_1, x_2, ..., x_n\}$, then a fuzzy set A of X could be written as

$$A = \{(x_1, \mu_A(x_1)), (x_2, \mu_2(x_2)) ..., (x_n, \mu_A(x_n))\}$$

which sometimes we write in the following way symbolically:

$$A = \left\{ \frac{\mu_A(x_1)}{x_1}, \frac{\mu_A(x_2)}{x_2} ..., \frac{\mu_A(x_n)}{x_n} \right\}$$

For example, consider a set of tall men. The elements of the fuzzy set "tall men" are all men, but their degrees of membership depend on their height, as shown in Table 11.1. Suppose, for example, Mark at 205 cm tall is given a degree of 1, and Peter at 152 cm is given a degree of 0. All the men of intermediate height have intermediate degrees of being tall. They are partly tall. Obviously, different people may have different views as to whether a given man should be considered as tall. However, our candidates for *tall* men could have the memberships as shown in the table. It can be seen that the crisp set asks the question "Is the man tall?" and draws a line at, say, 180 cm. *Tall men* are above this height and *not tall men* below. In contrast, the fuzzy set asks "How tall is the man?" The answer is given in the partial membership in the fuzzy set (for example, Tom is 0.82 tall).

Table 11.1: The Degree of Membership for Crisp and Fuzzy Sets Based on Height

Name	Height, cm	Degree of Membership	
		Crisp	*Fuzzy*
Chris	208	1	1.00
Mark	205	1	1.00
John	198	1	0.98
Tom	181	1	0.82
David	179	0	0.78
Mike	167	0	0.15
Bob	167	0	0.15
Steven	158	0	0.06
Bill	152	0	0.00
Peter	152	0	0.00

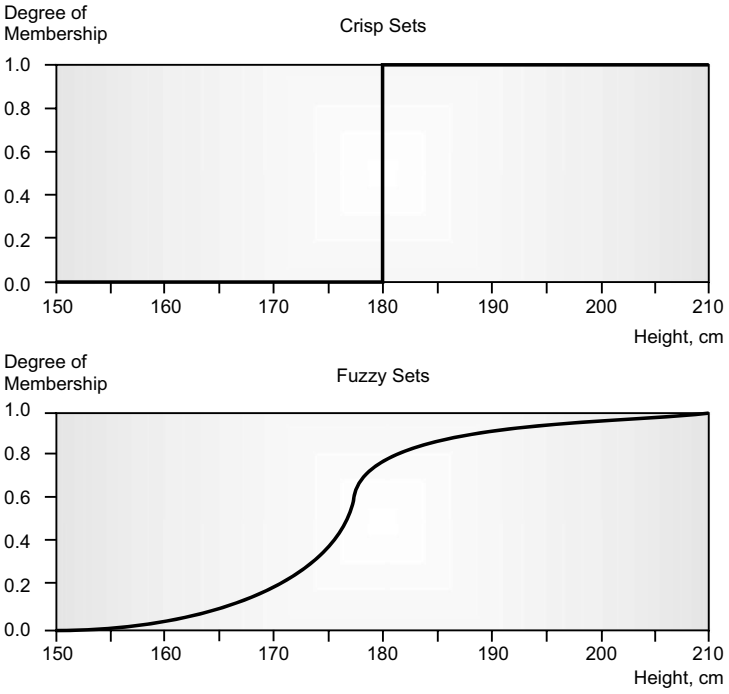
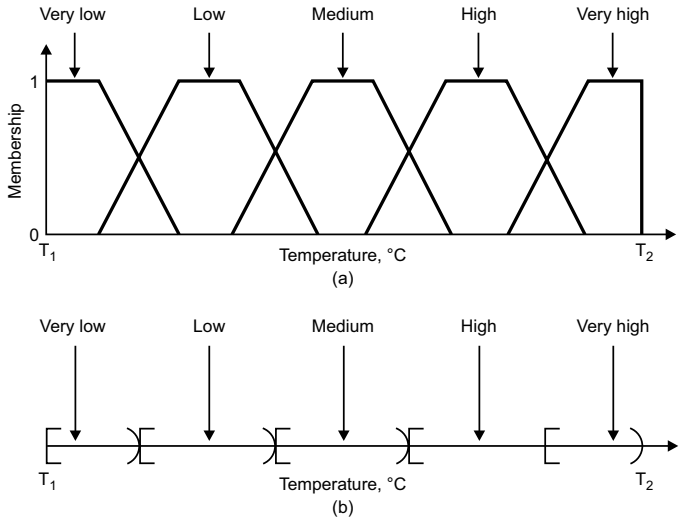


FIGURE 11.5 Crisp and Fuzzy Set Pictorial Representations

In Figure 11.5, the horizontal axis represents the universe of discourse, i.e., the range of all possible values applicable to a chosen variable. In our case, the variable is human height. According to this representation, the universe of men's heights consists of all tall men.

Now let us discuss fuzzy variables or linguistic variables.



Temperature in the range $\{T_1, T_2\}$ conceived as: (a) a fuzzy variable; (b) a traditional (crisp) variable.

FIGURE 11.6 Traditional and Fuzzy Variables

11.3.1 Linguistic Variables in a Fuzzy Set

- A linguistic variable may be defined as variables whose values can be phrases, words, etc. in a natural or artificial language.
- Fuzzy sets and linguistic variables can be used to quantify the meaning of natural language, which can then be manipulated.
- Linguistic variables must have a valid syntax and semantics.
- Each linguistic variable may be assigned one or more linguistic values, which are in turn connected to a numeric value through the mechanism of membership functions.
- Linguistic variables are fuzzy variables.

For example, "John is tall" means the linguistic variable "John" takes the linguistic value "tall."

The range of all possible values of linguistic variables represent the universe of discourse of that variable. For the example, for the universe of discourse of the linguistic variable speed, we must include fuzzy subsets such as "very slow," "slow," "medium," and "fast". A linguistic variable carries with it the concept of fuzzy set qualifiers called hedges. Hedges modify shape of the fuzzy set. They include adverbs such as "very," "quiet," "more," and "less."

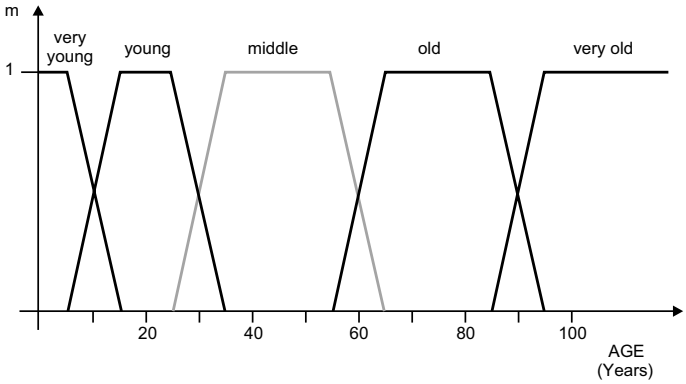


FIGURE 11.7 An Example of Hedges and Linguistic Variables

A linguistic variable is characterized by a quintuple:

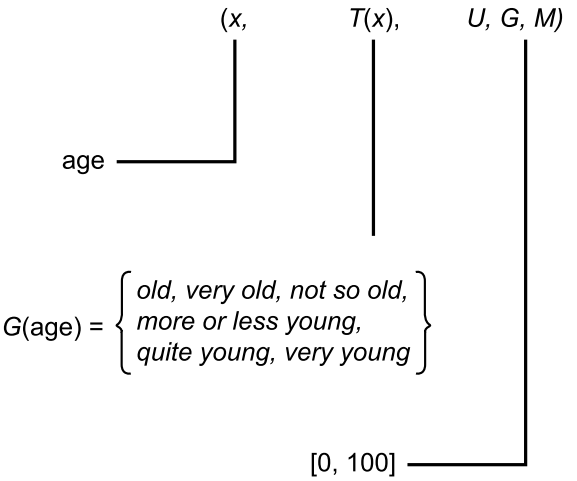
$(x, U, T(x), G, M)$ in which

- x is the name of the variable.
- U is the universe of discourse.
- $T(x)$ is the term set of x, that is, the set of names of linguistic values of x with each value being a fuzzy number defined on U.
- G is a syntactic rule for generating the names of values of x.
- M is a semantic rule for associating with each value its meaning.

For example,

- age = {very young; young; middle; old; very old}
- blood glucose level = {slightly increased; increased; significantly increased; stronglyincreased}
- insulin doses = {none; low; medium; high}

For example,



An Example of a Semantic Rule

$$M(\text{old}) = \{(u, \mu_{\text{old}}(u)) | u \in [0, 100]\}$$

$$\mu_{\text{old}}(u) = \begin{cases} 0 & u \in [50, 100] \\ \left[1 + \left(\frac{u-50}{5}\right)^{-2}\right]^{-1} & u \in [50, 100] \end{cases}$$

Fuzzy partitions formed by the linguistic values:
"young", "middle aged", and "old"

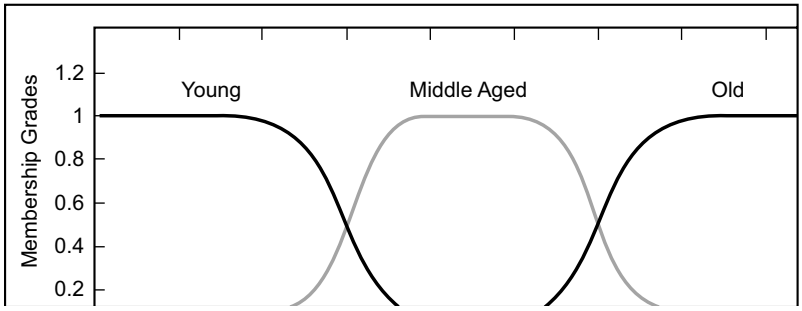


FIGURE 11.8 A Fuzzy Set Partition

Fuzzy Set with a Discrete Universe:

Fuzzy set A = "sensible number of children"

X = {0, 1, 2, 3, 4, 5, 6} (discrete universe)

A = {(0, 1), (1, 3), (2, 7), (3, 1), (4, 6), (5, 2), (6, 1)}

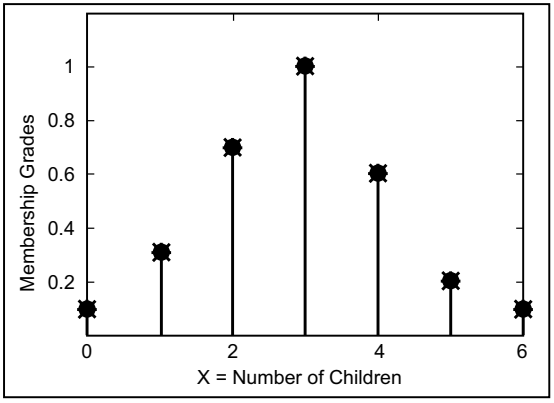


FIGURE 11.9 A Fuzzy Set with a Discrete Universe

Fuzzy Sets with Continuous Universes

Fuzzy set B = "about 50 years old"

X = Set of positive real numbers (continuous)

$B = \{(x, \mu_A(x) | x \text{ in } X)\}$

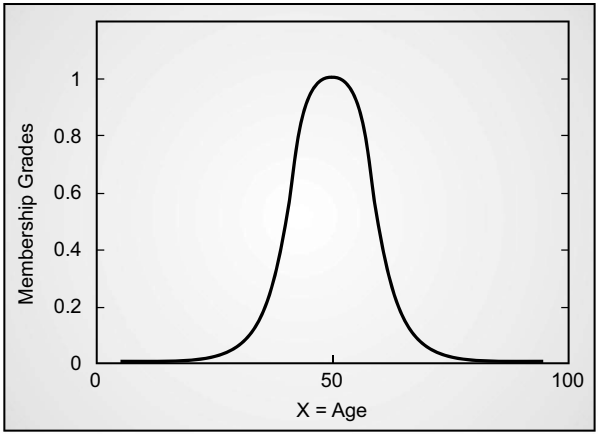


FIGURE 11.10 A Fuzzy Set with a Continuous Universe

Various examples for the illustration of fuzzy logic concepts are as follows.

Example 1

The whole concept can be illustrated with this example. Let's talk about people and "youthfulness." In this case, the set S (the universe of discourse) is the set of people. A fuzzy subset YOUNG is also defined, which answers the question "to what degree is person x young?" To each person in the universe of discourse, we have to assign a degree of membership in the fuzzy subset YOUNG. The easiest way to do this is with a membership function based on the person's age.

$$\text{Young}(x) = \begin{Bmatrix} 1, if\ age\,(x) <= 20 \\ (30 - age\,(x)/10, if\ 20 < age\,(x) <= 30,) \\ 0, if\ age\,(x) > 30 \end{Bmatrix}$$

A graph of this appears in Figure 11.11.

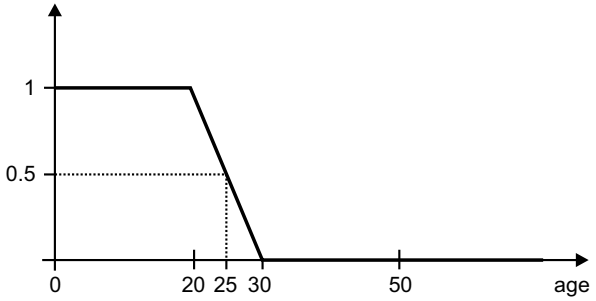


FIGURE 11.11 Graph Showing the Membership Function of a Fuzzy Set

Given this definition, here are some example values:

Person	Age	Degree of Youth
Johan	10	1.00
Edwin	21	0.90
Parthiban	25	0.50
Arosha	26	0.40
Chin Wei	28	0.20
Rajkumar	83	0.00

So given this definition, we'd say that the degree of truth of the statement "Parthiban is YOUNG" is 0.50.

Example 2

For a set of tall men, we shall say that people taller than or equal to 6 feet are tall. This set can be represented graphically as shown in Figure 11.12.

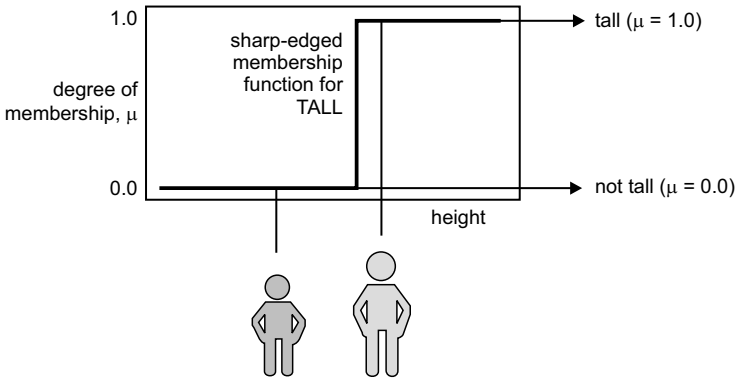


FIGURE 11.12 Crisp Set

The function shown above describes the membership of the "tall" set: you are either in it or you are not in it. This sharp-edged membership function works nicely for binary operations and mathematics, but it does not work as nicely in describing the real world. The membership function makes no distinction between somebody who is 6'1" and someone who is 7'1", they are both simply tall. Clearly there is a significant difference between the two heights.

The fuzzy set approach to the set of tall men provides a much better representation of the tallness of a person. The set, shown below, is defined by a continuously inclining function.

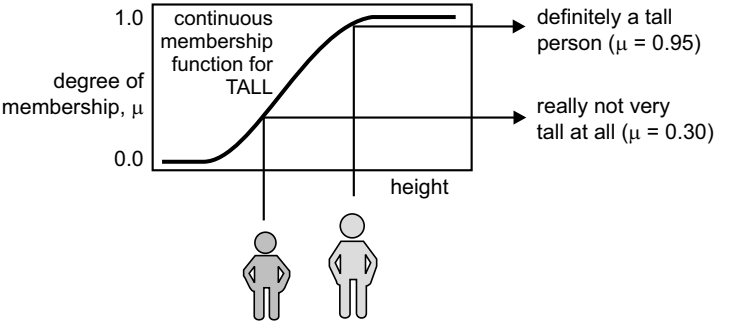


FIGURE 11.13 The Fuzzy Set for Tall Men

The membership function defines the fuzzy set for the possible values underneath it on the horizontal axis. The vertical axis, on a scale of 0 to 1, provides the membership value of the height in the fuzzy set. So, for the two people shown in Figure 11.13, the first person has a membership of 0.3 and so he is not very tall. The second person has a membership of 0.95 and so he is definitely tall. He does not, however, belong to the set of tall men in the way that bivalent sets work; he has a high degree of membership in the fuzzy set of tall men.

Now, let us discuss what the membership function is in crisp logic and fuzzy logic.

11.4 Membership Function of Crisp Logic

Crisp logic is concerned with absolutes: something is either true or false, and there is no in-between. For example

- Rule:
  - If the temperature is higher than 80°F, it is hot; otherwise, it is not hot.
- Cases:
  - Temperature = 100°F Hot
  - Temperature = 80.1°F Hot
  - Temperature = 79.9°F Not Hot
  - Temperature = 50°F Not Hot

Hence, the classification of individuals can be done using an indicator or characteristic function $\mu A(x)=0$ or 1

such that $\mu A(x)=0$ or 1.

11.5 Membership Function of the Fuzzy Set

The membership function of a fuzzy set is a generalization of the indicator function in classical sets. In fuzzy logic, it represents the degree of truth as an extension of the valuation. Degrees of truth are often confused with probabilities, although they are conceptually distinct, because fuzzy truth represents membership in vaguely defined sets, not the likelihood of some

event or condition. Membership functions were introduced by Zadeh in the first paper on fuzzy sets (1965).

For any set X, a membership function on X is any function from X to the real unit interval [0,1]. Membership functions on X represent fuzzy subsets of X. The membership function that represents a fuzzy set $\tilde{A}$ is usually denoted by μA. For an element x of X, the value $\mu_A(x)$ is called the *membership degree* of x in the fuzzy set $\tilde{A}$. The membership degree $\mu_A(x)$ quantifies the grade of the membership of the element x to the fuzzy set $\tilde{A}$. The value 0 means that x is not a member of the fuzzy set; the value 1 means that x is fully a member of the fuzzy set. The values between 0 and 1 characterize fuzzy members, which belongs to the fuzzy set only partially.

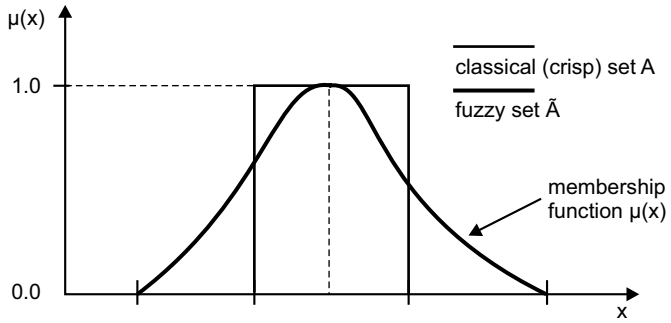


FIGURE 11.14 The Membership Function of Crisp and Fuzzy Logic

Definition: The membership function for a fuzzy set A on the universe of discourse X is defined as $\mu_A: X \rightarrow [0,1]$, where each element of X is mapped to a value between 0 and 1. This value, called the membership value or degree of membership, quantifies the grade of membership of the element in X to the fuzzy set A.

Membership functions allow us to graphically represent a fuzzy set. The x axis represents the universe of discourse, whereas the y axis represents the degrees of membership in the [0,1] interval. Simple functions are used to build membership functions. Because we are defining fuzzy concepts, using more complex functions does not add more precision.

- The membership function fully defines the fuzzy set.
- A membership function provides a measure of the degree of similarity of an element to a fuzzy set.

- Membership functions can take any form, but there are some common examples that appear in real applications.
- Membership functions represent distributions of possibility rather than probability. For instance, the fuzzy set "Young" expresses the possibility that a given individual is young. Membership functions often overlap with each other. A given individual may belong to different fuzzy sets (with different degrees).
- Membership functions can
  - ➢ either be chosen by the user arbitrarily, based on the user's experience (membership functions chosen by two users could be different depending upon their experiences and perspectives)
  - ➢ be designed using machine learning methods (e.g., artificial neural networks or genetic algorithms)

For example, the fuzzy set approach to the set of tall men provides a much better representation of the tallness of a person. The set, shown below, is defined by a continuously inclining function.

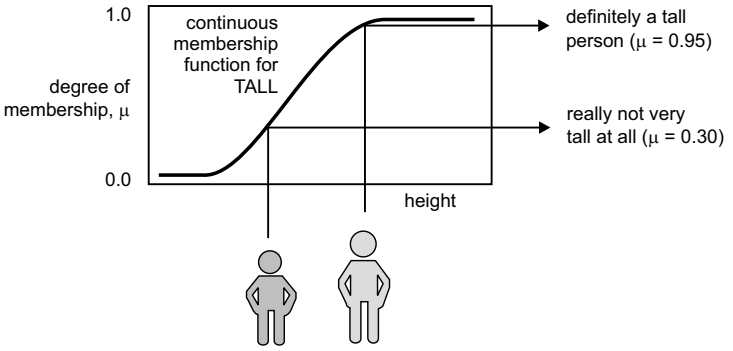


FIGURE 11.15 Fuzzy Set Tall Men Representation

The membership function defines the fuzzy set for the possible values underneath it on the horizontal axis. The vertical axis, on a scale of 0 to 1, provides the membership value of the height in the fuzzy set. So, for the two people shown in Figure 11.15, the first person has a membership of 0.3 and is not very tall. The second person has a membership of 0.95, and he is definitely tall. He does not, however, belong to the set of tall men in the way that bivalent sets work; he has a high degree of membership in the fuzzy set of tall men.

Here is another example.

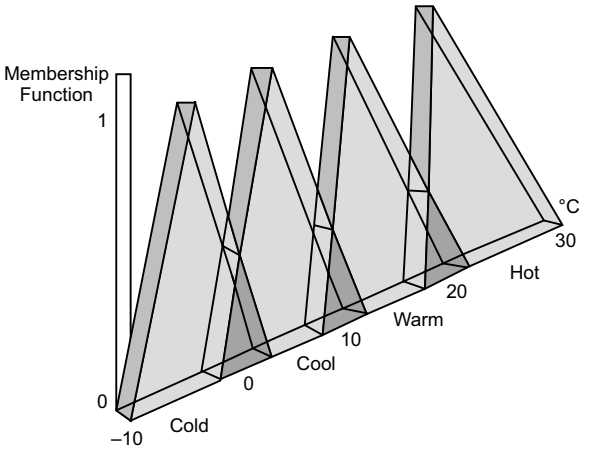


FIGURE 11.16 Fuzzy Set Characterizing the Temperature of a Room

There are different shapes of membership functions, such as triangular, trapezoidal, piecewise-linear, Gaussian, and bell-shaped.

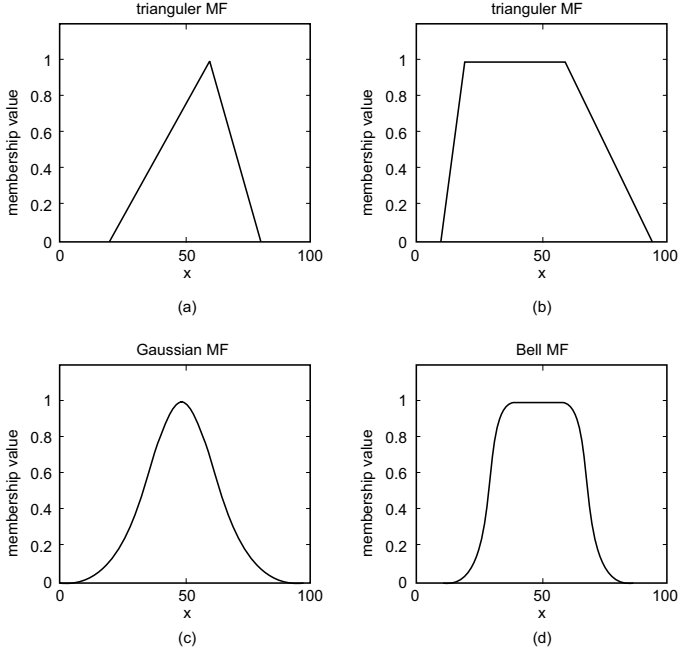


FIGURE 11.17 Different Shapes of the Fuzzy Membership Function

11.6 Fuzzy Set Operations

A **fuzzy set operation** is an operation on fuzzy sets. These operations are a generalization of crisp set operations. There is more than one possible generalization. The most widely used operations are called standard fuzzy set operations. In fuzzy logic, there are three basic operations on fuzzy sets: union, intersection, and complement.

11.6.1 Union

Let μ_A and μ_B be membership functions that define the fuzzy sets A and B, respectively, on the universe X. The union of fuzzy sets A and B is a fuzzy set defined by the membership function:

$$\mu_{AUB}(x) = \text{Max}\,(\mu_A(x),\mu_B(x))$$

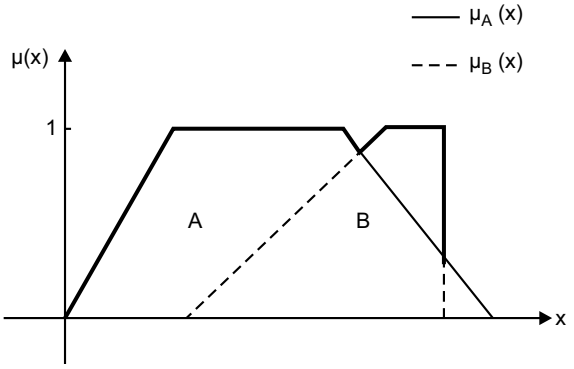


FIGURE 11.18 A Union Fuzzy Operation

For example, given two fuzzy sets A and B,

$A = 0.4/1+0.6/2+0.7/3+0.8/4$

$B = 0.3/1+0.65/2+0.4/3+0.1/4$

The union of the fuzzy sets A and B

$= 0.4/1+0.65/2+0.7/3+0.8/4$

11.6.2 Intersection

Let μ_A and μ_B be the membership functions that define the fuzzy sets A *y* B, respectively, on the universe X. The intersection of fuzzy sets A and B is a fuzzy set defined by the membership function

$$\mu_{A\cap B}(x) = \text{Min}\,(\mu_A(x),\mu_B(x))$$

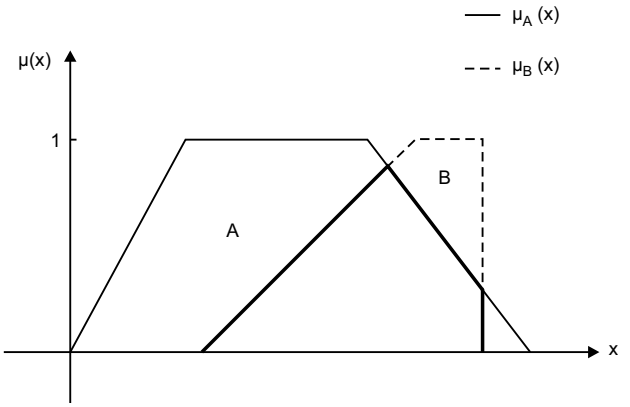


FIGURE 11.19 An Intersection Fuzzy Operation

For example, given two fuzzy sets *A* and *B*

$A = 0.4/1+0.6/2+0.7/3+0.8/4$

$B = 0.3/1+0.65/2+0.4/3+0.1/4$

The intersection of the fuzzy sets A and B = 0.3/1+0.6/2+0.4/3+0.1/4.

11.6.3 Complement

Let μ_A be a membership function that defines the fuzzy set *A*, on the universe *X*. The complement of *A* is a fuzzy set defined by the following membership function:

$$\mu_{\tilde{A}}(x) = 1 - \mu_A(x)$$

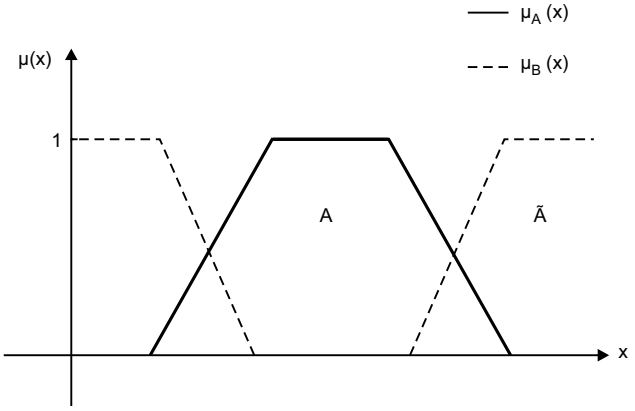


FIGURE 11.20 A Complement Operation

For example, given fuzzy set A

A = 0.4/1+0.6/2+0.7/3+0.8/4

The complement of the fuzzy set = 0.6/1+0.4/2+0.3/3+0.2/4.

Other operations in the fuzzy set are discussed next.

11.6.4 Equality of Two Fuzzy Sets

Let A and B be two fuzzy sets of X ($\neq \phi$) with membership functions of μ_A and μ_B. We say that A and B are equally written by $A = B$, if and only if

$\mu_A(x) = \mu_B(x) \forall x \in X$

Example

Suppose X = {1,2,3}. Consider the fuzzy sets A, B, and C of X given by

$$A = \left\{\frac{1}{2}, \frac{2}{7}, \frac{3}{0}\right\},$$

$$B = \left\{\frac{2}{4}, \frac{1}{2}, \frac{3}{0}\right\}, \text{ and}$$

$$C = \left\{\frac{2}{7}, \frac{3}{0}, \frac{1}{2}\right\}$$

and $\quad A = C, A \neq B, B \neq C$

11.6.5 Containment

Let X be a set ($\neq \phi$) and A and B are two fuzzy sets of X with membership function μ_A and μ_B respectively. We say that the fuzzy set A is contained in the fuzzy set B if and only if

$\mu_A(x) \leq \mu_B(x) \qquad \forall x \in X.$

We may say that A is a fuzzy subset of B, denoted by $A \subseteq B$.

For example, if X = {1, 2, 3} and A, B, and C are three fuzzy sets given by

$$A = \left\{\frac{1}{1}, \frac{5}{2}, \frac{1}{3}\right\}, B = \left\{\frac{1}{1}, \frac{4}{2}, \frac{9}{3}\right\}, \text{ and}$$

$$C = \left\{\frac{1}{1}, \frac{6}{2}, \frac{1}{3}\right\}, \text{ then } B \subseteq A, C \subseteq A.$$

11.6.6 Normal Fuzzy Set

A fuzzy set A of set X is called a normal set if and only if

$$\max_{x \in X} \mu_A(x) = 1$$

i.e., $\mu_A(x) = 1$ for the least one $x \in X$.

For example, if x = {1,2,3} and A = $\left\{\frac{1}{2}, \frac{2}{1}, \frac{3}{0}\right\}$ is a fuzzy set of *X*, then *A* is normal fuzzy.

11.6.7 Support of a Fuzzy Set

The support of a fuzzy set *A* of the set *X* is the classical set

$$\{x \in X : \mu_A(x) > 0\}.$$

denoted by support (*A*). For example, in the previous example, support (*A*) = {1, 2}.

11.6.8 α-Cut or α-Level Set

The α-cut or α-level set of fuzzy set A of the set X is the following crisp (i.e., conventional) set given by

$$A_\alpha = \{x \in X : \mu_A(x) \geq \alpha\}$$

For example, suppose X = {2, 1, 4, 3}. Consider the fuzzy set *A* of *X* given by

$$A = \left\{\frac{5}{2}, \frac{1}{1}, \frac{1}{4}, \frac{8}{3}\right\}$$

Clearly, $A_1 = \{1, 2, 3, 4\}$.

$A_6 = \{3, 4\}$, $A_0 = \{1, 2, 3, 4\}$

$A_1 = \{4\}$, $A_3 = \phi$

Clearly, if $\alpha \geq \beta$, A_α A_β. Also, $A_0 = X$ and $A_\alpha = \phi$, $\forall \alpha > 1$.

11.6.9 Disjunctive Sum (Exclusive OR)

$$A \oplus B = \left(A \cap \overline{B}\right) \cup \left(\overline{A} \cap B\right)$$

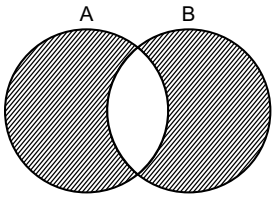


FIGURE 11.21 A Disjunctive Sum

$$\mu_{\overline{A}}(x) = 1 - \mu_A(x),\ \mu_{\overline{B}}(x) = 1 - \mu_B(x)$$

$$\mu_{A \cap B}(x) = \text{Min}[\mu_A(x), 1 - \mu_B(x)]$$

$$\mu_{\overline{A} \cap B}(x) = \text{Min}[1 - \mu_A(x), \mu_B(x)]$$

$A \oplus B = \left(A \cap \overline{B}\right) \cup \left(\overline{A} \cap B\right)$, then

$\mu A \oplus B\ (x) =$ Max {Min[$\mu A(x)$, 1 – $\mu B(x)$], Min[1 – $\mu A(x)$, $\mu B(x)$]}

$A =$ $\{(x_1, 0.2), (x_2, 0.7), (x_3, 1), (x_4, 0)\}$

$B =$ $\{(x_1, 0.5), (x_2, 0.3), (x_3, 1), (x_4, 0.1)\}$

$\overline{A} =$ $\{(x_1, 0.8), (x_2, 0.3), (x_3, 0), (x_4, 1)\}$

$\overline{B} =$ $x_1, 0.5), (x_2, 0.7), (x_3, 0), (x_4, 0.9)\}$

$A \cap \overline{B} = \{(x_1, 0.2), (x_2, 0.7), (x_3, 0), (x_4, 0)\}$

$\overline{A} \cap B = \{(x_1, 0.5), (x_2, 0.3), (x_3, 0), (x_4, 0.1)\}$

and as a consequence,

$A \oplus B = \left(A \cap \overline{B}\right) \cup \left(\overline{A} \cap B\right) = \{(x_1, 0.5), (x_2, 0.7), (x_3, 0), (x_4, 0.1)\}$

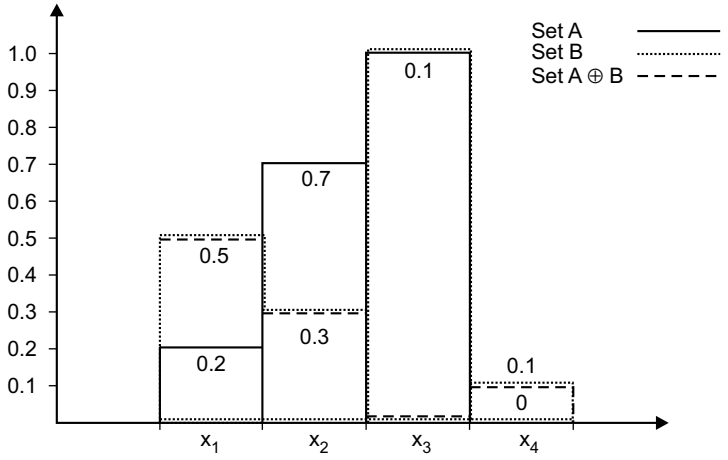


FIGURE 11.22 A Disjunctive Sum

11.6.10 Disjoint Sum

$$\mu_{A\Delta}B(x) = |\mu_A(x) - \mu_B(x)|$$

$$A = \{(x_1, 0.2), (x_2, 0.7), (x_3, 1), (x_4, 0)\}$$

$$b = \{(x_1, 0.5), (x_2, 0.3), (x_3, 1), (x_4, 0.1)\}$$

$$A \Delta B = \{(x_1, 0.3), (x_2, 0.4), (x_3, 0), (x_4, 0.1)\}$$

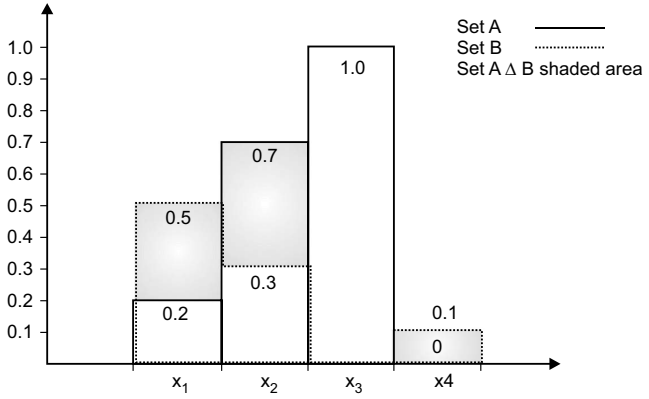


FIGURE 11.23 A Disjoint Sum

11.6.11 Difference

$$A = \{(x_1, 0.2), (x_2, 0.7), (x_3, 1), (x_4, 0)\}$$

$$B = \{(x_1, 0.5), (x_2, 0.3), (x_3, 1), (x_4, 1)\}$$

$$\overline{B} = \{(x_1, 0.5), (x_2, 0.7), (x_3, 0), (x_4, 0.9)\}$$

$$A - B = A \cap \overline{B} = \{(x_1, 0.2), (x_2, 0.7), (x_3, 0), (x_4, 0)\}$$

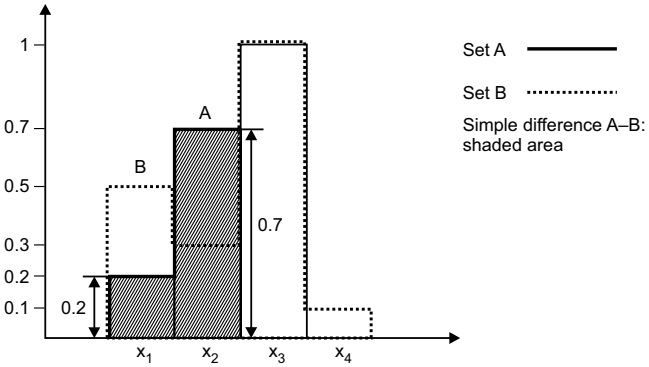


FIGURE 11.24 A Difference of Two Fuzzy Sets

11.6.12 The Bounded Difference

$$\mu_{A\theta}B(x) = \text{Max}[0, \mu_A(x)] - \mu_B(x)$$

$$A = \{(x_1, 0.2), (x_2, 0.7), (x_3, 1), (x_4, 0)\}$$

$$B = \{(x_1, 0.5), (x_2, 0.3), (x_3, 1), (x_4, 0.1)\}$$

$$A\ \theta\ B = \{(x_1, 0), (x_2, 0.4), (x_3, 0), (x_4, 0)\}$$

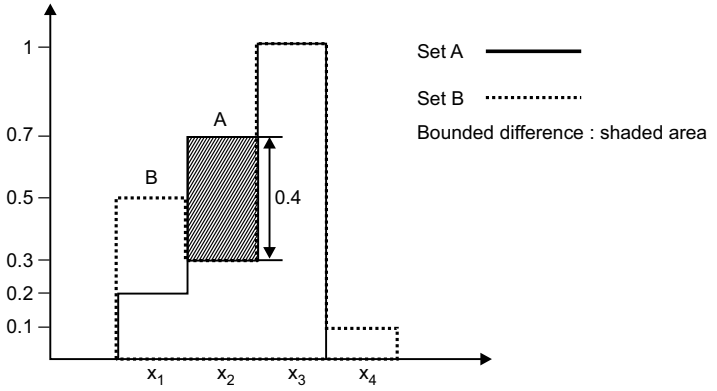


FIGURE 11.25 The Bounded Difference of Fuzzy Sets

11.7 Properties of A Fuzzy Set

- Involution $\overline{A} = A$
- Commutativity $A \cup B = B \cup A$

  $A \cap B = B \cap A$
- Associativity $(A \cup B) \cup C = A \cup (B \cup A)$

  $(A \cap B) \cap C = A \cap (B \cap A)$
- Distributivity $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

  $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
- Idempotent $A \cup A = A$

  $A \cap A = A$
- Absorption $A \cup (A \cap B) = A$

  $A \cap (A \cup B) = A$
- De Morgan's Law $\overline{A \cap B} = \overline{A} \cup \overline{B}$

  $\overline{A} \cap \overline{B} = \overline{A} \cup \overline{B}$

11.8 Differences Between a Fuzzy Set and A Crisp Set

1.

- A **crisp set** is also called a classical set. In the case of a crisp set, either an element belongs to the set or it does not. For example, for the set of integers, either an integer is even or it is not (it is odd).
- For example A = {apples, oranges, cherries, mangoes} is a crisp set because each element of set A either belongs to A or not.
- **Fuzzy set** theory is different from classical set theory as the elements of a fuzzy set admit some degree of membership, which means how much an element belongs to a set. In fuzzy set theory, we assume that all are members, e.g., all belong to the set up to a certain extent. Some elements may belong at 80% and some 30%, which gives the measure of belongingness or the degree of belongingness.

2.

- **Crisp set:** Let X be the universe of discourse, and its element is denoted as x. In classical set theory, crisp set A of X is defined as function $F_A(x)$ that is

$$F_A(x) : \mathrm{X} \rightarrow \{0,1\} \text{ where } F_A(x) = \begin{matrix} 1 \, if & x \in A \\ 0 \, if & x \notin A \end{matrix}$$

For any element x of X, $F_A(x)$ is equal to 1 if x is an element of set A and equal to 0 if x is not an element of A. Classical sets are called crisp sets: either

OR

$x \in A \;\; x \notin A$

- **Fuzzy set:** Fuzzy set A of universe X is defined by $\mu_A(x)$, which is called a membership function of set A.

$\mu_A(x) : x \rightarrow \{0,1\}$ where $\mu_A(x) = 1$ if x is totally in A

$\mu_A(x) = 0$ if x is totally in A

$0 < \mu_A(x) < 1$ if x is partly in A

For any element x of universe X, the membership function equals the degree to which x is an element of set A. These values between 0 and 1 represent the degrees of membership.

For example, fuzzy set A = "sensible number of children"

$X = \{0, 1, 2, 3, 4, 5, 6\}$ (discrete universe)

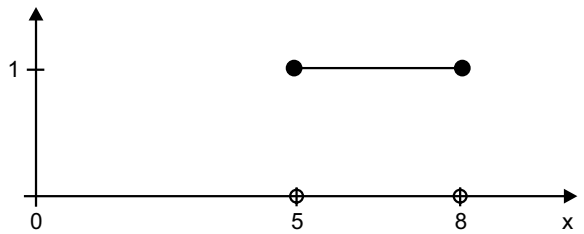
$A = \{(0, .1), (1, .3), (2, .7), (3, 1), (4, .6), (5, .2), (6, .1)\}$

3.

- Crisp set: In classical set theory, the membership of elements in a set is assessed in binary terms according to a bivalent condition: an element either belongs or does not belong to the set.
- Fuzzy set: The fuzzy set membership of an element is not assessed in binary terms, as fuzzy logic is a form of multi-valued logic since membership of an element of a fuzzy set can be 0,1, or between 0 and 1.

4.

- Crisp Set: Consider a set X that contains all the real numbers between 0 and 10 and a subset A of the set X that contains all the real numbers between 5 and 8. Subset A is represented in the figure below.



In the figure, the interval on the x-axis between 5 and 8 has the y-value of 1. This indicates that any number in this interval is a member of the subset A. Any number that has a y-value of 0 is considered to be a non-member of the subset A.

Fuzzy Set

- Here is an example describing a set of young people using fuzzy sets. In general, young people range in age from 0 to 20. But, if we use this strict interval to define young people, then a person on his 20th birthday is still young (still a member of the set). But on the day after his 20th birthday, this person is now old (not a member of the

young set). However, we can relax the boundary between the strict separation of young and old. The figure below graphically illustrates a fuzzy set of young and old people.

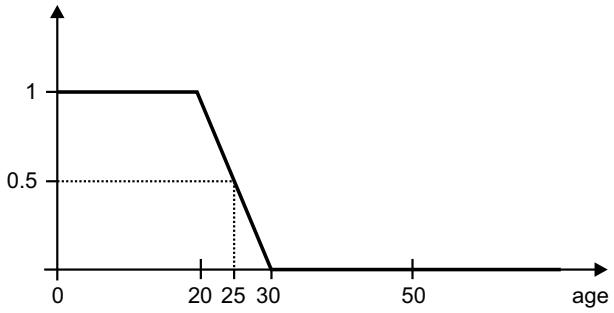


FIGURE 11.26 A Fuzzy Set of Young and Old People

Notice in the figure that people whose ages are >= 0 and <= 20 are complete members of the young set (that is, they have a membership value of 1). Also note that people whose ages are > 20 and < 30 are partial members of the young set. For example, a person who is 25 would be young to the degree of 0.5. Finally, people whose ages are >= 30 are non-members of the young set.

5.

For example, consider a set of tall men. The elements of the fuzzy set "tall men" are all men, but their degrees of membership depend on their height, as shown in Table 11.2. Suppose, for example, Mark at 205 cm tall is given a degree of 1, and Peter at 152 cm is given a degree of 0. All men of intermediate height have intermediate degrees. They are partly tall. Obviously, different people may have different views as to whether a given man should be considered as tall. However, our candidates for tall men could have the memberships presented in table below.

It can be seen that the crisp set asks the question "Is the man tall?" and draws a line at, say, 180 cm. Tall men are above this height and not tall men are below it. In contrast, the fuzzy set asks "How tall is the man?" The answer is the partial membership in the fuzzy set, for example, Tom is 0.82 tall.

Table 11.2 Degree of Membership of Tall Men

Name	Height, cm	Degree of Membership	
		Crisp	*Fuzzy*
Chris	208	1	1.00
Mark	205	1	1.00
John	198	1	0.98
Tom	181	1	0.82
David	179	0	0.78
Mike	172	0	0.24
Bob	167	0	0.15
Steven	158	0	0.06
Bill	155	0	0.01
Peter	152	0	0.00

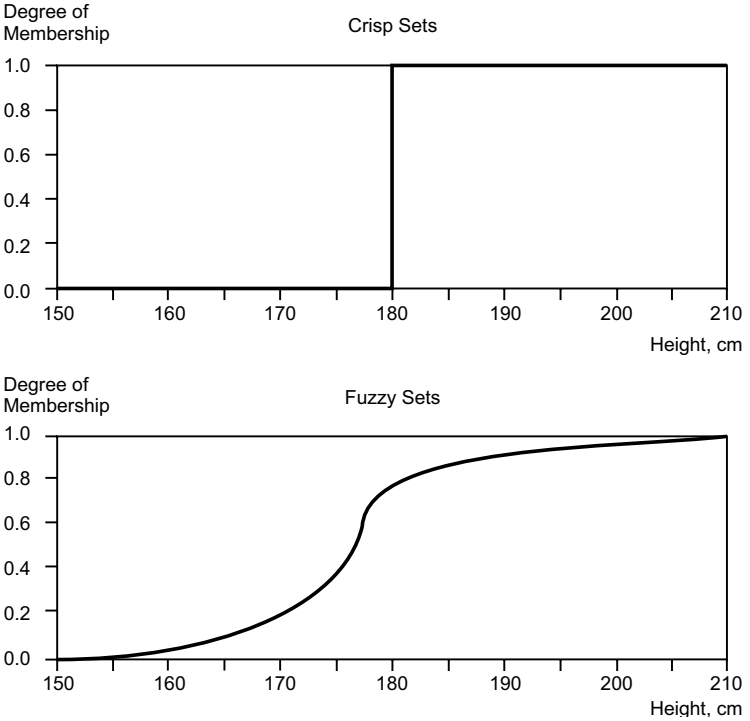


FIGURE 11.27 The Difference Between Crisp and Fuzzy Logic

In this figure, the horizontal axis represents the universe of discourse, that is, the range of all possible values applicable to a chosen variable. In our case, the variable is the human height. According to this representation, the universe of men's heights consists of all tall men.

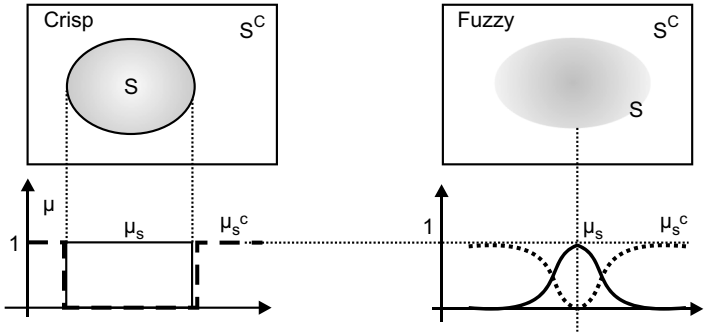


FIGURE 11.28 Fuzzy and Crisp Sets

11.9 Differences Between Boolean Logic and Fuzzy Logic

1.

- **Boolean Logic:** This is a kind of logic with two values, 0 and 1.
- **Fuzzy Logic:** This is a form of multi-valued logic that can take the values 0, 1, or between 0 and 1.

2.

- **Boolean Logic:** Here, variables may take on true or false values. Boolean logic only allows true or false values. An example of this could be a computer game. A person is standing in a doorway while a thing explodes. The character is hit or not hit if Boolean logic is used. For example, for the set of integers, either an integer is even or it is not (it is odd). Either you are in the US or you are not.
- **Fuzzy logic:** Fuzzy logic allows all things in between. In it, variables may have a truth value that ranges in degree between 0 and 1. Fuzzy logic has been extended to handle the concept of partial truth, where the truth value may range between completely true and completely false. Furthermore, when linguistic variables are used, these degrees may be managed by specific functions. Fuzzy logic variables have a

range between 0 and 1. Fuzzy logic is based on the degree of truth rather than true or false Boolean logic. It includes 0 and 1 as extreme cases of truth, but it also includes various states of truth in between those values.

- For example, let a 100 ml glass contain 30 ml of water. We may consider two concepts: empty and full. The meaning of each of them can be represented by a certain fuzzy set. One might define the glass as being 0.7 empty and 0.3 full. Note that the concept of emptiness would be subjective and thus would depend on the observer or designer. Another designer might equally design a set membership function where the glass would be considered full for all values down to 50 ml.

3.

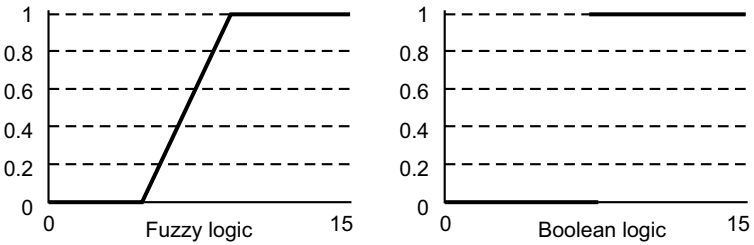


FIGURE 11.29 Differences Between Fuzzy and Boolean Logic

4.

- **Boolean Logic:** In this, a crisp set is a well-defined collection of elements in which an element either belongs to a set or not. For example, A = {x | x is an even natural number}, so the element is either even or not.
- **Fuzzy Logic:** Here, a fuzzy set provides the means to model the uncertainty associated with vagueness, imprecision, and a lack of information regarding a problem. Fuzzy sets contain elements that have degrees of membership. For example,
  - ♦ The motor is running really hot.
  - ♦ Tom is a very tall guy.
  - ♦ Electric cars are not very fast.

5.

- **Boolean Logic:** This cannot represent vague concepts, and therefore fails to give the answers for paradoxes.
- **Fuzzy Logic:** This refers to inexact reasoning. It is used to describe fuzziness and represent vague concepts.

6.

- **Boolean Logic:** It asks the question whether an element belongs to set or not. It can be seen that the crisp set asks the question "Is the man tall?" and draws a line at, say, 180 cm. *Tall men* are above this height and *not tall men* are below it.
- **Fuzzy Logic:** It asks how much an element belongs to a set. The classical example in the fuzzy set theory is that of tall men. The elements of the fuzzy set "tall men" are all men, but their degrees of membership depend on their height. Different people may have different views as to whether a given man should be considered tall.

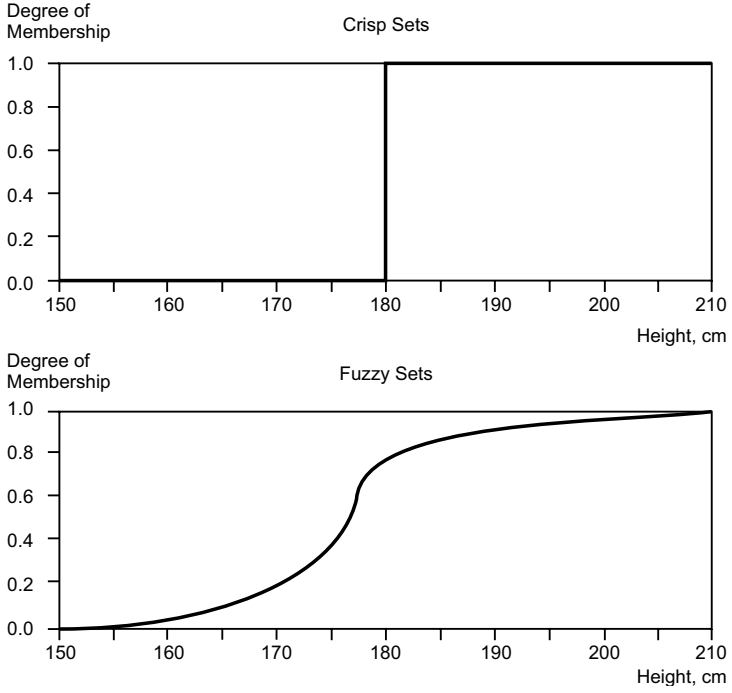


FIGURE 11.30 Fuzzy Logic and Boolean Logic Differences in Graphical Form

7.

- **Boolean Logic:** Boolean or conventional logic uses sharp distinctions. It forces us to draw lines between members of a class and non-members. It makes us draw lines in the sand. For instance, we may say, "The maximum range of an electric vehicle is short," regarding a range of 300 km or less as short, and a range greater than 300 km as long. By this standard, any electric vehicle that can cover a distance of 301 km (or 300 km and 500 m or even 300 km and 1 m) would be described as long-range.
- **Fuzzy Logic:** Fuzzy logic reflects how people think. It attempts to model our sense of words, our decision making, and our common sense. As a result, it is leading to new, more human-like, intelligent systems. For example, we say "Tom is tall" because his height is 181 cm. If we drew a line at 180 cm, we would find that David, who is 179 cm, is small. Is David really a small man or have we just drawn an arbitrary line in the sand? Fuzzy logic makes it possible to avoid such absurdities.

Exercises

Q1. Differentiate between fuzzy and Boolean logic.

Q2. What is a membership function of a fuzzy set?

Q3. Can a fuzzy membership be true and false at the same time?

Q4. Name three strengths and three weaknesses of fuzzy expert systems.

Q5. What is a linguistic variable?

Q6. What is the main difference between probability and fuzzy logic?

Q7. What types of operations can be performed on or with fuzzy sets?

Q8. Differentiate between a fuzzy set and a crisp set.

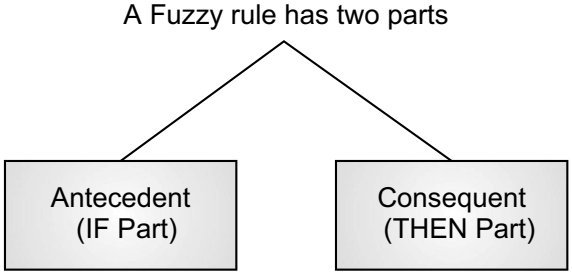
CHAPTER 12

FUZZY SYSTEMS

A fuzzy system is also called a rule-based system that is constructed from a collection of linguistic rules. It involves non-linear mapping from input to output. A fuzzy-rule-based system is most useful in modeling a complex system that can be observed by humans.

12.1 Fuzzy Rule

The fuzzy rule is the basic unit for capturing knowledge in a fuzzy system. It is the backbone of a fuzzy inference system, and it is the most important modeling tool based on fuzzy set theory. In 1973, Lotfi Zadeh published his second most influential paper (Zadeh, 1973). This paper outlined a new approach to the analysis of complex systems, in which Zadeh suggested capturing human knowledge in fuzzy rules.



The antecedent describes the conditions and consequents that describe the conclusion drawn when a condition gets satisfied. A fuzzy rule can be defined as a conditional statement in the following form:

IF x is A

THEN y is B

Here, "x *is* A" is called the antecedent or premise.

"y is B" is called the consequence or conclusion.

In this example, x and y are linguistic variables; A and B are the linguistic values determined by fuzzy sets on the universe of discourses X and Y, respectively. An example is as follows:

Rule 1:

IF height is tall

THEN weight is heavy

Rule 2:

IF pressure is high

THEN volume is small

The value of the output or a truth membership grade of the rule consequent can be estimated directly from a corresponding truth membership grade in the antecedent. The antecedent of a fuzzy rule can have multiple parts. As a production rule, a fuzzy rule can have multiple antecedents, combined using the connectives AND, OR, and NOT. For example,

Rule 1:

IF project_duration is long

AND

project_staffing is large

AND

project_funding is inadequate

THEN risk is high

Rule 2:

IF service is excellent

OR food is delicious

THEN tip is generous

All parts of the antecedent are calculated simultaneously and resolved in a single number using fuzzy set operations.

The consequent of a fuzzy rule can have multiple parts. The consequent of a fuzzy rule can also include multiple parts, for instance,

IF temperature is hot

THEN hot_water is reduced;

cold_water is increased

In this case, all parts of the consequent are affected equally by the antecedent. The linguistic knowledge for a problem is given in the form of fuzzy rules.

- If the blood pressure is above the target and decreasing slowly, then reduce the drug infusion.
- If the pressure is high, then the volume is small.
- If the road is slippery, then driving is dangerous.
- If a tomato is red, then it is ripe.

Fuzzy rules are represented in matrix form, e.g., consider the following fuzzy rules:

- If Angle is Zero and **Angular vel is Zero**
  - ♦ then output Zero velocity
- If Angle is SP and **Angular vel is Zero**
  - ♦ then output SN velocity
- If Angle is SN and **Angular vel is Zero**
  - ♦ then output SP velocity

Table 12.1 A Representation of Fuzzy Rules in Matrix Form

Angle Vel Angle	LN	SN	ZE	SP	LP
LN			MP		
SN			SP		
ZE			ZE		
SP			SN		
LP			MN		

What is the difference between classical and fuzzy rules?

A classical IF-THEN rule uses binary logic, for example,

Rule: 1

IF speed is > 100

THEN stopping_distance is long

Rule: 2

IF speed is < 40

THEN stopping_distance is short

The variable speed can have any numerical value between 0 and 220 km/h, but the linguistic variable *stopping_distance* can take either the value "long" or "short." In other words, classical rules are expressed in the black-and-white language of Boolean logic.

However, we can also represent the stopping distance rules in a fuzzy form:

Rule: 1

IF speed is fast

THEN stopping_distance is long

Rule: 2

IF speed is slow

THEN stopping_distance is short

Here, the linguistic variable *speed* also has the range (the universe of discourse) between 0 and 220 km/h, but this range includes fuzzy sets, such as *slow*, *medium,* and *fast*. The universe of discourse of the linguistic variable *stopping_distance* can be between 0 and 300 m and may include such fuzzy sets as *short, medium, and long*. Thus, fuzzy rules relate to fuzzy sets. Fuzzy expert systems merge the rules and consequently cut the number of rules by at least 90%.

Before employing fuzzy rules to a model and analyzing a system, first we have to formalize what is meant by the expression "if x is A, then y is B." This is sometimes abbreviated as

$$\text{A} \quad \rightarrow \quad \text{B}$$

A fuzzy implication like A → B describes a relation between two variables, x and y. This suggests that a fuzzy IF-THEN rule be defined as a binary fuzzy relation R on the product space X × Y.

12.1.1 Fuzzy Rules as Relations

A fuzzy rule can be defined as a binary relation with a membership function.

$$\underbrace{A \rightarrow B}_{R} = \text{If x is A, then y is B.}$$

$$\underbrace{\mu_R(x,y) = \mu_{A\rightarrow B}(x,y)}_{\substack{\textit{Depends on how} \\ \textit{to interpret } A\rightarrow B}}$$

Fuzzy relation R is a 2D membership function, A, and the binary fuzzy relation R is an extension of the classical Cartesian product, where each element $(x, y) \varepsilon X \times Y$.

$$R = \left\{((x,y), \mu_R(x,y)) \mid (x,y)\varepsilon X \times Y\right\}$$

Examples

- x is close to y (x and y are numbers)
- x depends on y (x and y are events)
- x and y look alike (x and y are persons or objects)
- If x is large, then y is small (x is an observed instrument reading and y is a corresponding control action)

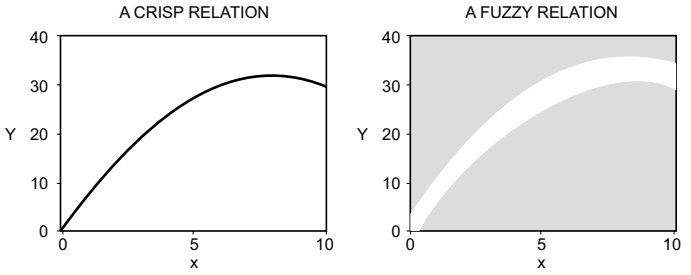


FIGURE 12.1 Crisp and Fuzzy Relations Showing X Close to Y

12.1.1.1 The Max-Min Composition

The max-min composition of two fuzzy relations R_1 (defined on X and Y) and R_2 (defined on Y and Z) are as follows:

R : fuzzy relation defined on X and Y.

S : fuzzy relation defined on Y and Z.

$\underbrace{R.S.}$: the composition of R and S.

A fuzzy relation defined on X and Z

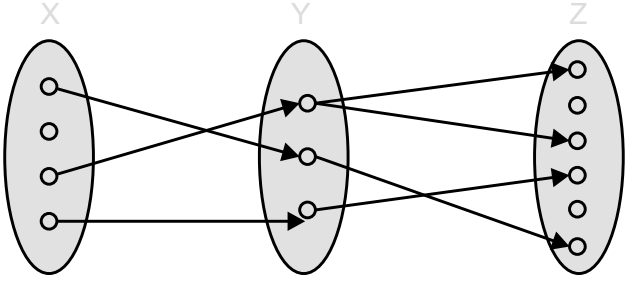


Figure 12.2 Showing Max-min Composition of Relations R_1 and R_2

To compute the max-min composition,

$$\mu_{Ras}(x,z) = \max_y \min(\mu_R(x,y), \mu_s(y,z))$$
$$= v_y(\mu_R(x,y) \wedge \mu_s(\mu, z))$$

Here is an example of the max-min composition.

R	a	b	c	d
1	0.1	0.2	0.0	1.0
2	0.3	0.3	0.0	0.2
3	0.8	0.9	1.0	0.4

S	a	β	y
a	0.9	0.0	0.3
b	0.2	1.0	0.8
c	0.8	0.0	0.7
d	0.4	0.2	0.3

	0.1	0.2	0.0	1.0
min	0.9	0.2	0.8	0.4
max	0.1	0.2	0.0	0.4

$R.S$	a	β	Y
1	0.4	0.2	0.3
2	0.3	0.3	0.3
3	0.8	0.9	0.8

Properties of **the M**ax-**M**in **C**omposition

- Associativity:

$$R \circ (S \circ T) = (R \circ S) \circ T$$

- Distributivity over union:

$$R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$$

- Weak distributivity over the intersection:

$$R \circ (S \cap T) \subseteq (R \circ S)\ (R \circ S) \cap (R \circ T)$$

- Monotonicity:

$$S \subseteq T \Rightarrow (R \circ S) \subseteq (R \circ T)$$

The max-min composition is not mathematically tractable; therefore, other compositions, such as the max-product composition, have been proposed.

12.1.1.2 Max-Product Composition

R : a fuzzy relation defined on X and Y

S : a fuzzy relation defined on Y and Z

$\underbrace{R.S.}$: The composition of R and S.

A fuzzy relation defined on X and Z.

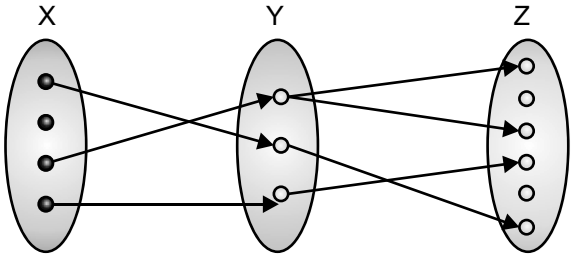


FIGURE 12.3 The Max-Product Composition

μRas (x,z) = max y (μr(x,y)μs (y,z))

In general, we have max * compositions:

$\mu R_1.R_2$ (x,z) = $\overset{V}{}_{y}[\mu_{R1}(x,y)^{*}\mu_{R2}(y,z)]$

Example – Max * Compositions

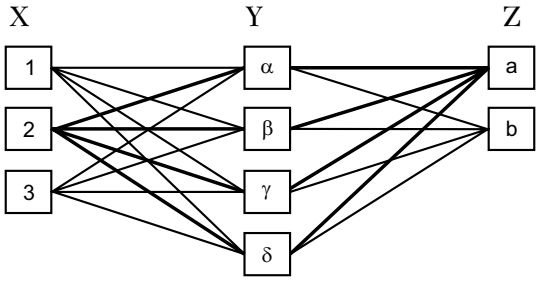
R_1: x is relevant to y

	$y = a$	$y = \beta$	$y = y$	$y = \delta$
$x = 1$	0.1	0.3	0.5	0.7
$x = 2$	0.4	0.2	0.8	0.9
$x = 3$	0.6	0.8	0.3	0.2

R_2: y is relevant to z

	$z = a$	$z = b$
$y = a$	0.9	0.1
$y = \beta$	0.2	0.3
$y = y$	0.5	0.6
$y = \delta$	0.7	0.2

How relevant is $x = 2$ to $z = a$?



$\mu R_1 R_2 (R,a) = 0.7$ (max - min composition)

$\mu R_1 R_2 (2,a) = 0.63$ (max - product composition)

12.1.2 Interpretation of Fuzzy Rules

There are two ways to interpret fuzzy rules, that is, one is coupling and the other is entailing.

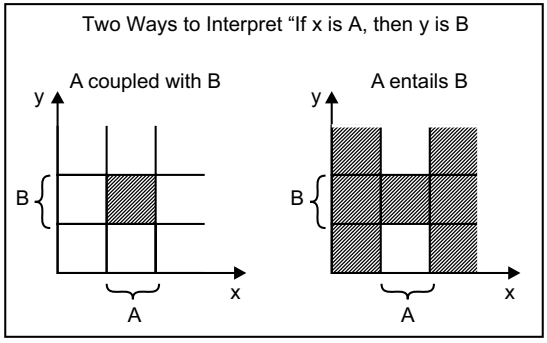


FIGURE 12.4 Coupling and Entailing

Here is an example showing the difference between coupling and entailing. Consider the fuzzy rule

If (profession is athlete) then (fitness is high).

Coupling: Athletes, and only athletes, have high fitness.

The "if" statement (antecedent) is a necessary and sufficient condition.

Entailing: Athletes have high fitness, and non-athletes may or may not have high fitness.

The "if" statement (antecedent) is a sufficient but not necessary condition.

12.1.2.1 Coupling (A with B)

If we interpret a fuzzy implication $F = A \rightarrow B$ as "A coupled B" then

$$R = A \rightarrow B = A \times B = \int \mu_A(x) * \mu_B(y) \mid (x, y)$$

where * is a t-norm.

For example,

$$\mu_R(x,y) = \min(\mu_A(x), \mu_B(y))$$

12.1.2.2 Entails (Not A or B)

If we interpret a fuzzy implication $F = A \rightarrow B$ as "A entails B," then it can be written as four different formulas:

- Material implication

  $R = A \rightarrow B = -A \cup B$ $\quad \mu_R(x, y) = \max(1-\mu_A(x), \mu_B(x))$

- Propositional calculus

  $R = A \rightarrow B = -A \cup (A \cap B)$ $\quad \mu_R(x, y) = \max(1-\mu_A(x), \min(\mu_A(x), \mu_B(x)))$

- Extended propositional calculus

  $R = A \rightarrow B = (-A \cap -B) \cup B$ $\quad \mu_R(x, y) = \max(1-\max(\mu_A(x), \mu_B(x)), \mu_B(x))$

- Generalization of modus ponens

12.2 Fuzzy Reasoning

Fuzzy reasoning is an inference procedure that derives conclusions from a set of fuzzy rules and known facts. For fuzzy reasoning, we have to first study the compositional rule of inference, which plays key role in fuzzy reasoning. Using the compositional rule of inference, we can formalize an inference procedure, called fuzzy reasoning, upon a set (bank) of fuzzy IF-THEN rules.

The basic rule of inference in traditional two-valued logic is the modus ponens, according to which we can infer the truth of a proposition *B* from the truth of *A* and the implication *A* ® *B*.

For example (modus ponus),

Premise 1 (fact):x is *A*,

Premise 2 (rule):if x is *A*, then y is *B*,

Consequence (Conclusion):y is *B*.

If A is identified with "the tomato is red" and B with "the tomato is ripe," then if it is true that "the tomato is red," it is also true that "the tomato is ripe."

But in the case of human reasoning or fuzzy logic, the modus ponus is applied in an approximate manner. For example, if we have same implication rule "IF tomato is red THEN it is ripe" and we know that "tomato is more or less red," then we may infer that "tomato is more or less ripe." This can be illustrated with help of the generalized modus ponus (it has the modus ponus as a special case).

The generalized modus ponens (GMP) or fuzzy reasoning or approximate reasoning is as follows:

Premise 1 (fact) : x is *A*, then y is *B*,

Premise 2 (rule) : if x is *A*, then y is *B*,

Consequence (Conclusion) : y is *B*'

where A' is close to A and B' is close to *B*.

Let us discuss rules with different parts and how they are reasoned.

A Single Rule with a Single Antecedent

Rule : if x is *A*, then y is *B*

Fact : x is *A*'

Conclusion : y is *B*'

Here, there is only one rule with one antecedent part so the membership function of the conclusion is calculated from the fact and rule. The procedure is described below.

Max-Min Composition

$$\begin{aligned}\mu B'(y) &= \max_x \min(\mu_{A'}(x), \mu_R(x, y)) \\ &= \vee_x (\mu_A(x) \wedge \mu_R(x, y)) \\ &= \vee_x (\mu_{A'}(x) \wedge \mu_A(x) \wedge \mu_B(y)) \\ &= [\vee_x (\mu_{A'}(x) \wedge \mu_A(x))] \wedge \mu_B(y)\end{aligned}$$

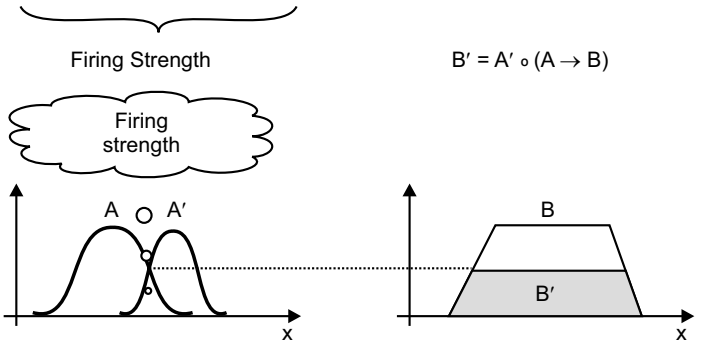


FIGURE 12.5 Approximate Reasoning for a Single Rule with a Single Antecedent

A Single Rule with Multiple Antecedents

Rule : if x is A and y is B, then z is C

Fact : x is A' and y is B'

Conclusion : z is C'

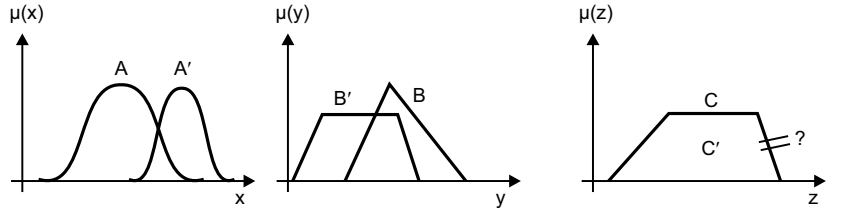


FIGURE 12.6 A Single Rule with Multiple Antecedents

$$R = A \times B \rightarrow C$$

$$\mu_R(x,y,z) = \mu_{(A \times B) \times C}(x, y, z)$$
$$= \mu_A(x) \wedge \mu_B(y) \wedge \mu_c(z)$$

The resulting expression is $C' = (A' \times B') \circ (A \times B \rightarrow C)$

Thus the max-min composition is

$$\mu_{C'}(y) = \max_{x,y} \min(\mu_{A',B'}(x, y), \mu_R(x, y, z))$$
$$= \vee_{x,y}(\mu_{A',B'}(x, y) \wedge \mu_R(x, y, z))$$
$$= \vee_{x,y}(\mu_{A'}(x) \wedge \mu_{B'}(y) \wedge \mu_A(x) \wedge \mu_B(y) \wedge \mu_C(z))$$
$$= [\vee_x(\mu_A(x) \wedge \mu_{A'}(x))] \wedge [\vee_y(\mu_B(y) \wedge \mu_{B'}(y))] \wedge \mu_C(z)$$

Firing Strength

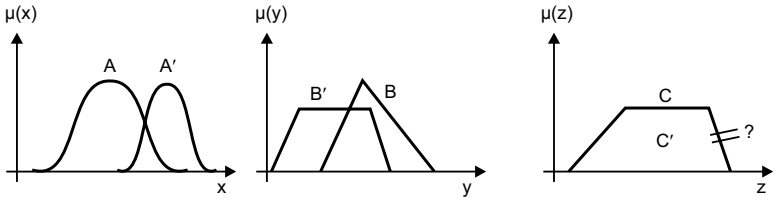


FIGURE 12.7 Approximate Reasoning for a Single Rule with Multiple Antecedents

Multiple Rules with Multiple Antecedents

The interpretation of multiple rules is usually taken as the union of the fuzzy relations corresponding to the fuzzy rules.

Rule 1:

if x is A_1 and y is B_1, then z is C_1

Rule 2:

if x is A_2 and y is B_2, then z is C_2

Fact : x is A' and y is B'

Conclusion : z is C'

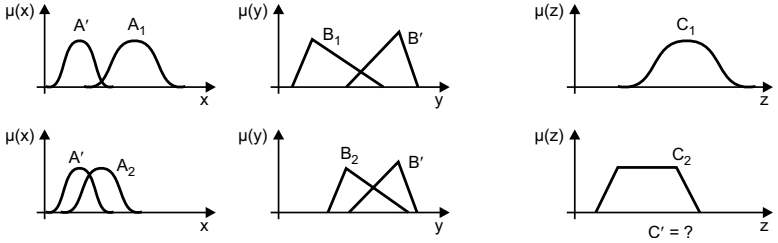


FIGURE 12.8 Multiple Rules with Multiple Antecedents

We can employ fuzzy reasoning as an inference procedure to derive the resulting output fuzzy set.

$$C' = (A' \times B') \circ (R_1 \cup R_2)$$
$$= [(A' \times B') \circ R_1] \cup [(A' \times B') \circ R_2]$$
$$= C'_1 \cup C'_2$$

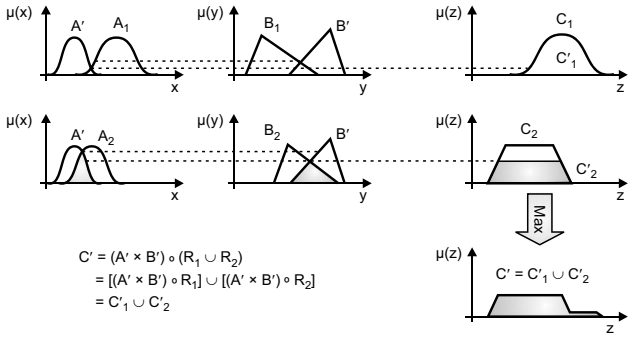


FIGURE 12.9 Approximate Reasoning for Multiple Rules with Multiple Antecedents

Thus, the overall process of fuzzy reasoning is divided into four steps:

- Degree of compatibility

Compare the known facts with the antecedent of the rule to find the degree of compatibility with respect to each antecedent's membership function.

- Firing strength

Combine the degree of compatibility with respect to the membership function in a rule using the fuzzy AND/OR operator to form a firing strength that indicates the degree to which the antecedent part of the rule is satisfied.

- Qualified (induced) consequent membership functions

Apply the firing strength to the consequent membership function of a rule to generate a qualified consequent membership function.

- Overall output membership function

Aggregate all qualified consequent membership functions to obtain the overall output membership function.

Exercises

Q1. What is a fuzzy rule?

Q2. What are fuzzy relations, and does fuzzy logic support them?

Q3. What are two ways of interpreting fuzzy relations?

Q4. Explain fuzzy reasoning.

Q5. What is entailment?

CHAPTER 13

FUZZY EXPERT SYSTEMS

13.1 The Need for Fuzzy Expert Systems

Expert systems are computer programs that are designed to make use some of the skills of an expert to non-experts. Such programs attempt to emulate, in some way, an expert's thinking patterns. For example, DENDRAL was the first expert system developed in 1965, and it determined the molecular structure from mass spectrometer data. MYCIN is an expert system used for medical diagnosis. There are a number of approaches for simulating the expertise of an expert: the rule-based approach, semantic nets, frames, and neural networks. Of these, the most important are the rule-based system and neural network.

In neural nets, there is no need to explicitly specify the thinking patterns of an expert. Instead, two sets of data are required from the real world. These data include all the inputs to the system, and the correct outputs correspond to these input values. The first set of data, the training set, is used to train the neural network so that the correct outputs are produced for each set of input values. The second set of data, the validation set, is used after the neural net has been trained so that the correct answers are produced using different input data. A disadvantage of the neural network approach is that a lot of training sets are required.

In rule-based systems, it is necessary to explicitly specify the expert's knowledge and thinking patterns. Two people are needed for developing a rule-based system. The first is the domain expert, who has the knowledge of how to solve problems, but little idea of computer programming. The second

is a knowledge engineer, who is trained on the computer technology involved and expert systems, but has little or no knowledge of the problem at hand. Obtaining such knowledge and writing that knowledge in proper rules is called the knowledge acquisition phase. The advantage of rule-based systems (expert systems) is that there is no need for a large training set. However, fuzzy expert systems have overtaken conventional expert systems.

Fuzzy expert systems are needed because of the flaws in conventional expert systems, as a conventional expert system is insufficient in emulating human thought patterns. Humans are able to deal with uncertainty and ambiguities, but it is difficult for conventional expert systems to deal with such things. Fuzzy logic (a fuzzy set) deals with imprecise data, vagueness, and uncertainties. Fuzzy expert systems can replace conventional expert systems by incorporating fuzzy concepts into the expert system. A fuzzy system can emulate an expert's rigid thinking patterns better than a conventional expert system.

A fuzzy expert system is used under following circumstances:

- when there is no large training set
- when we are interested in how the output is derived from the input
- when we have a domain expert and knowledge engineer

A fuzzy expert system is a collection of membership functions, facts, and rules (instead of Boolean logic) that are used to reason about data. A fuzzy expert system is also called a fuzzy inference system (FIS). A fuzzy expert system uses fuzzy logic concepts for making decisions rather than Boolean logic concepts.

In the case of a fuzzy expert system, the knowledge engineer should be familiar with fuzzy logic concepts and data-driven non-procedural languages. Conventional languages like C and Fortran are procedural languages, that is, the execution of the program statements occur in the order in which they are written unless explicit transfers of control are executed. In data-driven non-procedural rule-based programs, the firing of the rules does not follow a sequence: the rules are fired when the data permits it. If the data satisfy more than one rule at once, then a rule-conflict algorithm decides which rule should be fired first. If some fireable rules are not picked for firing, then such rules are placed on a stack for firing later. That is, a selected rule is first fired, and then the program goes back looking for newly fireable rules. One important advantage of a fuzzy expert system is that a fuzzy rule can be writ-

ten in natural language that is easily understandable by an expert. That is, we can write a fuzzy rule in a fuzzy expert system using words like "very tall" or "somewhat." A fuzzy expert system provides a proper structured way to deal with uncertainties and ambiguities by using these types of words in fuzzy rules. For example, a fuzzy rule might include the idea "if height is very tall."

The syntax of fuzzy rules in a fuzzy expert system is as follows:

IF x is A and y is B, THEN z is C

Here, x is A, y is B, and z is C; they are fuzzy statements. x and y are input variables; z is an output variable; and A, B, and C are fuzzy sets. A,B, and C are the linguistic values determined by the fuzzy sets on the universe of discourses that were described in a previous chapter.

- The IF part of the rule "x is A and y is B" is called the antecedent or premise, while the THEN part of the rule "z is C" is called the consequent or conclusion. The antecedent describes the conditions and the consequents describe the conclusion drawn when the condition gets satisfied, that is, the antecedent part of the rule describes to what degree the rule applies.
- Mostly, there is more than one conclusion per rule in a fuzzy expert system, and the set of rules in a fuzzy expert system is known as the *knowledge base* of the fuzzy expert system.
- One important feature of a fuzzy expert system is the concurrent execution of fireable rules that causes a parallel operation. A parallel system runs faster than a sequential system, but a sequential system also has some advantages, so it's necessary that fuzzy expert system should utilize both modes, i.e., the parallel and sequential modes.

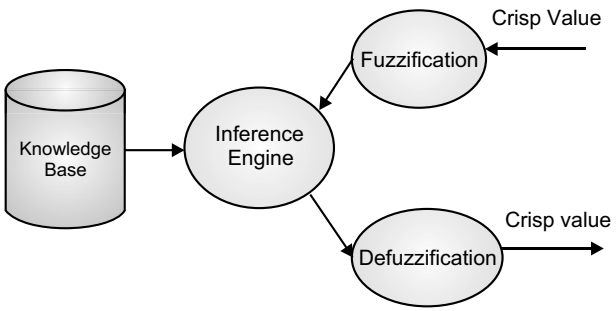


FIGURE 13.1 Structure of a Fuzzy Expert System

A fuzzy expert system requires two types of knowledge in its knowledge base:

- **Domain Knowledge:** This is the knowledge of the human expert that is stored in the knowledge base in the form of fuzzy rules and facts.
- **Meta Knowledge:** This is the knowledge of how to use that domain knowledge, that is, how to use methods like the type of t-norm, s-norm, implication and inference engine, and method of defuzzification.

There are three ways of obtaining domain knowledge:

- **Incorporating human expert knowledge:** The first method of obtaining domain knowledge is incorporating human expert knowledge into a knowledge base in the form of fuzzy IF….THEN rules.
- **Learning from examples:** In this, a set of input–output data pairs are used to train a system and then the resulting fuzzy system is used as a fuzzy model of the main system. It is expected to have the same input-output mapping. There are various ways of learning, such as fuzzy clustering and table look-up schemes.
- **Automatic optimization methods:** In these methods, we obtain domain knowledge by a search and optimization method, that is, we search the available knowledge and then use optimization techniques to find the appropriate knowledge for the knowledge base. There are various optimization techniques, such as the genetic algorithm and genetic programming.

13.2 Operations on a Fuzzy Expert System

Here, we are going to explain the whole process of a fuzzy inference system by using an example. There are three rules in the knowledge base of a fuzzy expert system.

- Rules
  - 1. If the service is **poor** or food is **bad**, then the tip is **cheap.**
  - 2. If the service is **good**, then the tip is **average.**
  - 3. If the service is **excellent** or the food is **delicious**, then the tip is **generous.**

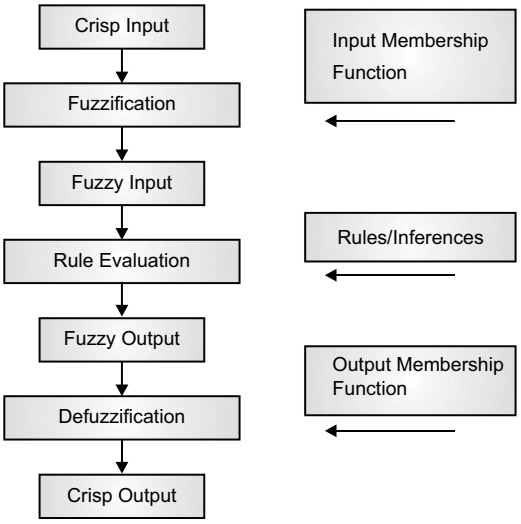


FIGURE 13.2 How a Fuzzy Expert System Works

- Input variables
  - ♦ Service: represented by poor, good, or excellent
  - ♦ Food: represented by bad or delicious
- Output Variable:
  - ♦ Tip: represented by cheap, average, or generous

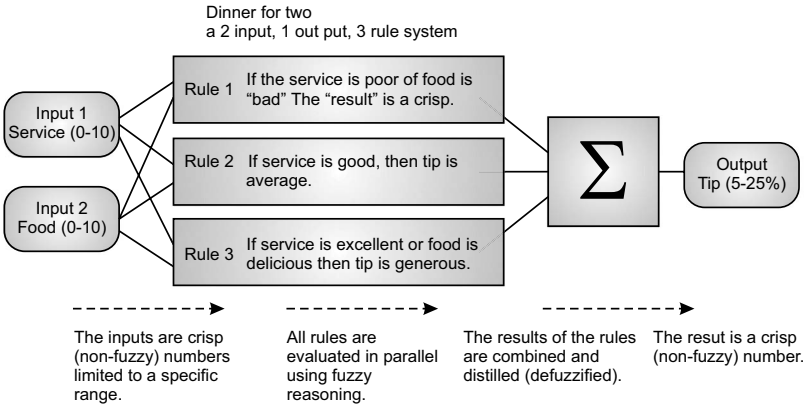


FIGURE 13.3 A Complete Description of a Fuzzy Expert System

13.2.1 Fuzzification (Fuzzy Input)

Fuzzification is the first step for fuzzy reasoning in a fuzzy expert system, and it is the base of the fuzzy system. In this step, the input and output of the system are identified, the appropriate fuzzy rules are defined, and raw data is used for deriving a membership function.

- The first step is to provide crisp inputs (non-fuzzy numbers), and, by applying the appropriate membership functions, determine the degree to which an input belongs to the fuzzy set.
- In the antecedent part of a fuzzy rule, the fuzzy statements are resolved to a degree of membership between 0 and 1, that is, the output of the fuzzification step is a fuzzy membership function that becomes the input to the fuzzy inference engine.
    - ♦ In the case of multiple parts of the antecedent, fuzzy operators are applied and the antecedent is resolved to a single number between 0 and 1.
- The antecedent can be joined using fuzzy operators like OR and AND.
    - ♦ For OR – max
    - ♦ For AND – min

For example, let the crisp input be "food=8;" then, this crisp input is fuzzified to each member of the linguistic set, like the "service is good," "food is bad," and "service is poor." That is, fuzzification returns to what extent the food is delicious.

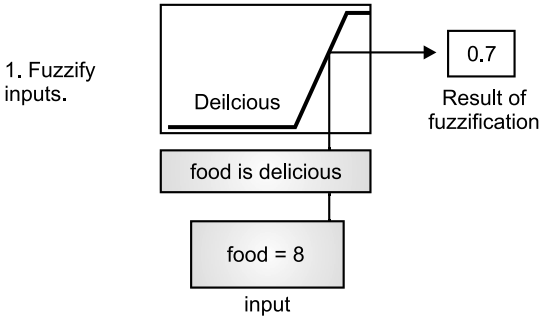


FIGURE 13.4 The Fuzzification of Input

The output of this fuzzification step is μ = 0.7 for the "delicious" membership function.

13.2.2 Fuzzy Operator

If the antecedent part of a fuzzy rule has more than one part, then we apply the fuzzy operator to the result of the fuzzification, that is, we apply the fuzzy operator to the membership function. After fuzzification, we came to know about the degree to which each rule antecedent part has been satisfied. So, after applying the fuzzy operator, we get a single truth value for an entire antecedent of each rule. There are varieties of fuzzy operators, like AND (min), OR (max), and complement.

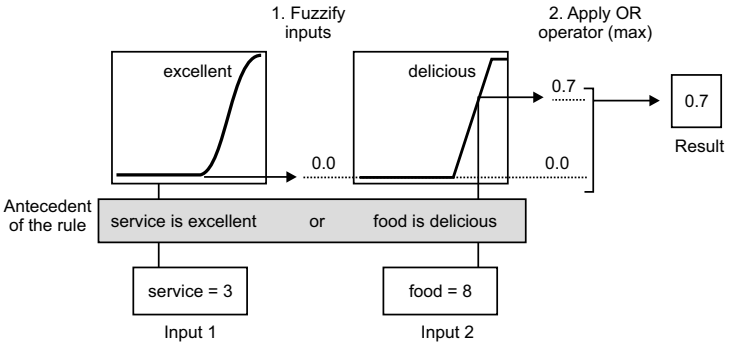


FIGURE 13.5 Application of a Fuzzy Operator

In this example, two crisp inputs are provided.

- Input 1

  Service=3, and the fuzzification step gives an output of 0.0 (membership function values) for the "service is excellent" part of the antecedent.

- Input 2

  Food=8, and the fuzzification step gives an output of 0.7 (membership function value) for the "food is delicious" part of the antecedent.

- Since both parts of the antecedent are joined using OR, the fuzzy logic OR operator is applied to the values of 0.0 and 0.7, and it selects the maximum of the two values (that is, 0.7). The final rule for the third part of the entire antecedent gives a value of 0.7, which is then applied to the consequent part of the rule in further steps.

13.2.3 Fuzzy Inferencing (Implication)

In fuzzy inferencing, the first thing that we need to keep in mind is that there is a weight of every rule. In general, the weight is 1, but there is a need to revise the weight of the rule from time to time to make it other than

1 relative to other rules. Inferencing is only done after the weight of each rule is properly assigned.

In the inference method,

- The truth value of the entire antecedent (0.7 in the previous step) of each rule is computed, and that is applied to the conclusion part of each rule (that is, the rules are evaluated).
- This results in one fuzzy set for each rule. The output fuzzy set is truncated depending on the degree of the "truth" of the antecedent part of rule by using the implication method.
- The output of the implication method is a fuzzy set, and it is done for each rule.
- If the consequent of a rule has many parts, then all parts of the consequents are equally affected by the result of the antecedent.
- Either we scale the fuzzy set or truncate it. For the scaling prod (PRODUCT), the implication method is used and for truncating, the min (minimum) implication method is used.
- The MIN and PRODUCT are two most important inference methods or inference rules. In the MIN method of inferencing, we clip off the output membership function at a height according to the premise's rule computed degree of truth. In the PRODUCT method of inferencing, we scaled the output membership function according to the premise's rule computed degree of truth.

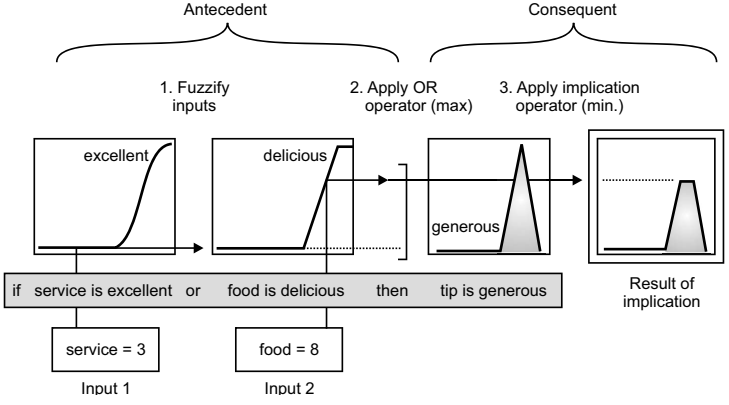


FIGURE 13.6 Fuzzy Implication (Inference)

In Figure 13.6, Rule 3 is evaluated, that is, the result of antecedent of rule (0.7) is applied to the consequent part of the rule ("tip is generous"). The obtained result is then truncated using the min implication method, and this results in a truncated fuzzy set.

13.2.4 Aggregate All Output

In this step, we combine the result of the implication of each rule. Aggregation is the process of combining the output of each rule (fuzzy set) into a single fuzzy set. The aggregation step is done once only before the final step (defuzzification). The input into the aggregation process is the list of truncated or scaled output functions (output of the implication process for each rule) and the output of the aggregation is the fuzzy set for each output variable.

The most commonly used aggregation operators are

- the maximum: point-wise maximum overall of the fuzzy sets
- the sum: (point-wise sum overall of the fuzzy sets)
- the probabilistic sum.

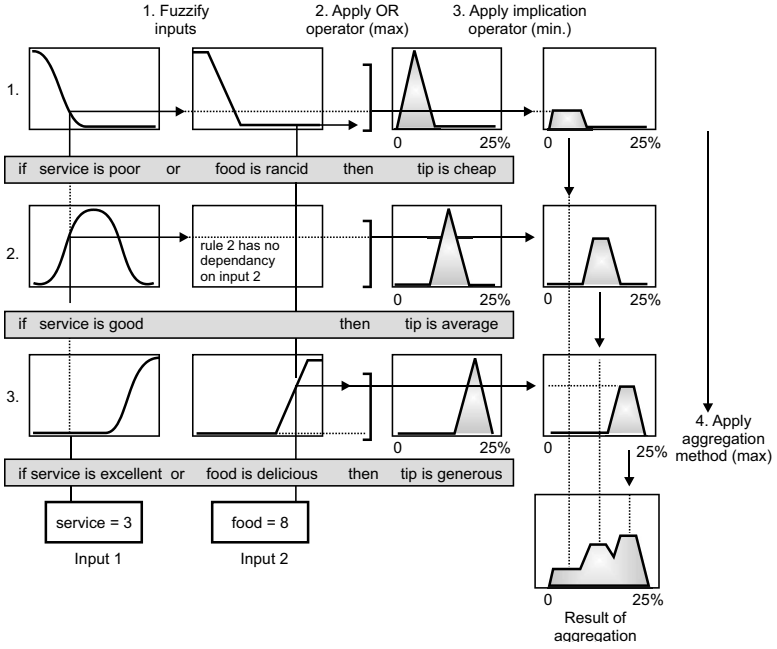


FIGURE 13.7 An Aggregation of a Fuzzy Set of Each Rule to a Single Fuzzy Set

13.2.5 Defuzzification

After the completion of the reasoning step, the defuzzification step is used to present the output of fuzzy reasoning in a form understandable to humans. There are basically two categories of defuzzification: arithmetic defuzzification and linguistic approximation.

In arithmetic defuzzification, we use a mathematical method to extract a single crisp value in the universe of discourse. This category of defuzzification is used in areas of control engineering where there is a need for a crisp result.

The second category of defuzzification is linguistic approximation, in which the consequent variable's term set is compared against the actual output set in a variety of combinations until the "best" representation is obtained in natural language. Linguistic approximation defuzzification is used in expert system advisory applications where human users view the output. Mostly, we use arithmetic defuzzification.

In arithmetic defuzzification, the output of the aggregation step acts as the input (aggregated fuzzy set) and gives a single crisp number as an output. This process is complex since it is not possible to directly translate a fuzzy set into a crisp value.

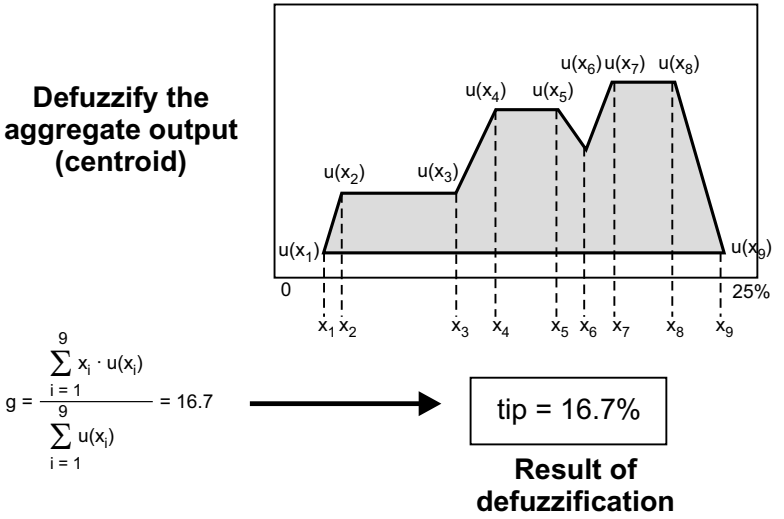


FIGURE 13.8 The Centroid Defuzzification Method

- Commonly used defuzzification methods are the centroid and maximum.
  - ♦ In the centroid method, the crisp value of the output variable is computed by finding the variable value of the center of gravity of the membership function for the fuzzy value.
  - ♦ In the maximum method, one of the variable values at which the fuzzy set has its maximum truth value is chosen as the crisp value for the output variable.
- Some other methods for defuzzification are the bisector, middle of maximum (mom) (the average of the maximum value of the output set), largest of maximum (lom), and smallest of maximum (som).

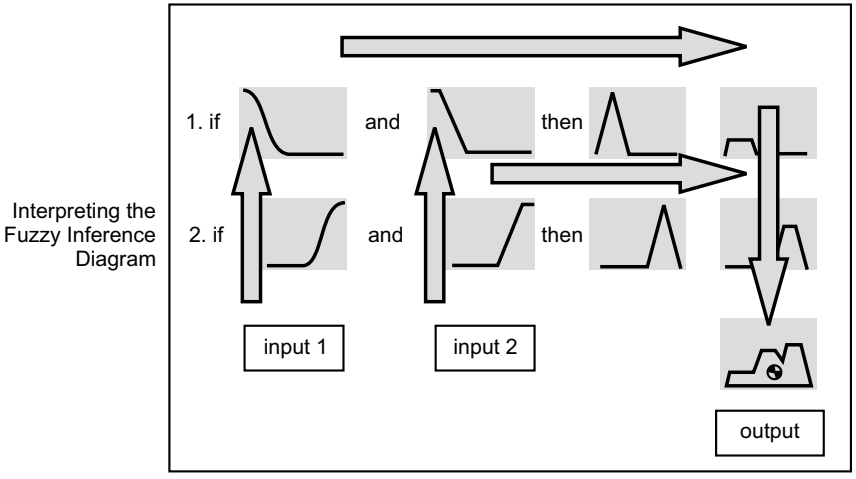


FIGURE 13.9 Fuzzy Inference Diagram Showing the Whole Inference Process

In Figure 13.9, the flow of data goes up from the inputs on the lower left, then across each row, or rule, and then down the rule outputs to finish on the lower right.

Fuzzy inference systems have a variety of applications, such as in automatic control, medical applications, science, data classification, decision analysis, expert systems, and computer vision.

13.3 Fuzzy Inference Systems

There are two types of fuzzy inference systems. Both methods have the same steps that were explained above.

13.3.1 Mamdani Fuzzy Inference Method

In the Mamdani inference method, the rules are of the following form: IF x is A and y is B, and THEN z is C.

Here, A, B, and C are fuzzy sets, x and y are input variables, and y is the output variable. The Mamdani system finally output one or more fuzzy sets, which are then defuzzified to obtain the crisp output.

Mamdani's fuzzy inference technique is the most commonly used technique. Mamdani's method was proposed in 1975 by Ebrahim Mamdani for controlling a steam engine and boiler combination by developing a set of linguistic control rules. The rules were obtained from the human operators' experiences.

The Mamdani inference method is used in applications where the fuzzy rules are a direct result of the human expert's advice, which are then expressed in the form of fuzzy rules.

Steps in the Mamdani Inference Method

- Fuzzification
- Rule evaluation
- Aggregation of rule output
- Defuzzification

Let us explain whole process using three fuzzy rules:

- Rule: 1

  IF x is $A3$ OR y is $B1$ THEN z $C1$

  Rule: 1

  IF project _funding is adequate OR

  project_staffing is small THEN risk is low

- Rule: 2

  IF x is $A2$ AND y is $B2$ THEN z is $C2$

  Rule: 2

  IF project_funding is marginal AND project_staffing is large THEN risk is normal

- Rule: 3
  IF x is *A*1 THEN z is C3

  Rule: 3
  IF project _funding is inadequate THEN risk is high

- Rules

  1. IF project_funding is adequate OR project_staffing is small, THEN risk is low.
  2. IF project_funding is marginal AND project_staffing is large, THEN risk is normal.
  3. IF project_funding is inadequate, THEN risk is high,

- Input variables
  - ♦ Project_funding: represented by adequate, inadequate, and marginal
  - ♦ Project_staffing: represented by small and large
- Output Variable:
  - ♦ Risk: represented by low, normal, and high

Fuzzification

The first step is to provide crisp inputs (non-fuzzy numbers) and by applying the appropriate

- membership functions, determine the degree to which an input belongs to a fuzzy set.

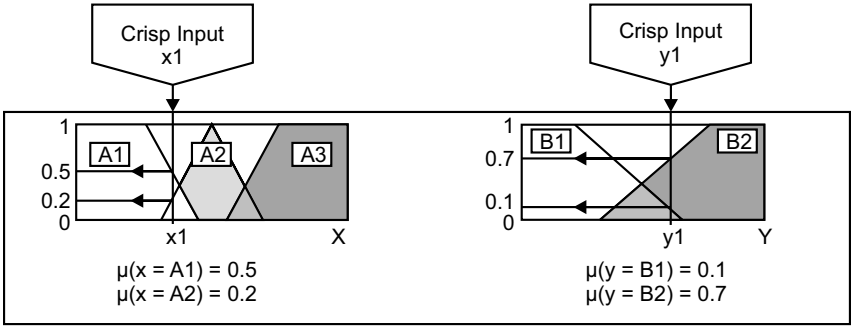


FIGURE 13.10 Fuzzification

In Figure 13.10, two crisp inputs are provided, $x1$ and $x2$ (project_funding, project_staffing), to determine how much these inputs belong to fuzzy sets $A1$, $A2$ and $B1$, $B2$. The outputs of the fuzzification step are the membership function values (0.5, 0.2, 0.1, and 0.7 are all membership function values obtained from the crisp inputs).

Apply Fuzzy Logic Operators

- Fuzzy logic operators are only applied if a fuzzy rule has multiple parts of antecedents. Since the above fuzzy rules have multiple antecedents, there is a need to apply a fuzzy operator (AND or OR) to obtain a single number.

The union of the fuzzy sets A and B (using the OR operator) is a fuzzy set defined by the membership function: $\mu_{AUB}(x) = \text{Max}\,(\mu_A(x),\mu_B(x))$. The intersection of fuzzy sets A and B(AND operator) is a fuzzy set defined by the membership function:

$$\mu_{A \cap B}(x) = \text{Min}(\mu_A(x),\mu_B(x))$$

- In Rule 1, the operator OR is used and in Rule 2, the operator AND is used. In Rule 1, we find maximum of the membership function values and in Rule 2, we find minimum of the membership function values to obtain a single number that represents the entire antecedent result.
- The second step is to take the fuzzified inputs, $\mu_{(x=A1)} = 0.5$, $\mu_{(x=A2)} = 0.2$, $\mu_{(y=B1)} = 0.1$, and $\mu_{(y=B\,2)} = 0.7$, and apply operators to obtain a single number.

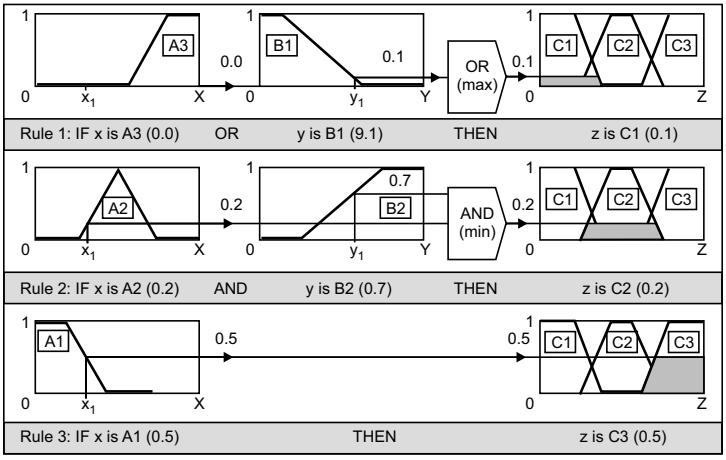


FIGURE 13.11 Application of Fuzzy Operators

Fuzzy Implication (Inference)

- The truth value of the entire antecedent of each rule is computed and that is applied to the conclusion part of each rule (that is, the rules are evaluated).
- This results in one fuzzy set for each rule. The output fuzzy set is truncated depending on the degree of the "truthfulness" of the antecedent part of the rule by using the implication method.
- The output of the implication method is a fuzzy set, and it is done for each rule.
- If the consequent of a rule has many parts, then all parts of the consequents are equally affected by the result of the antecedent.
- Either we scale the fuzzy set or truncate (clipping) it. For scaling prod (Product), the implication method is used and for truncating, the MIN (minimum) implication method is used.
- In case of clipping, the clipped fuzzy set loses some information but clipping is still preferred because it is less complex.

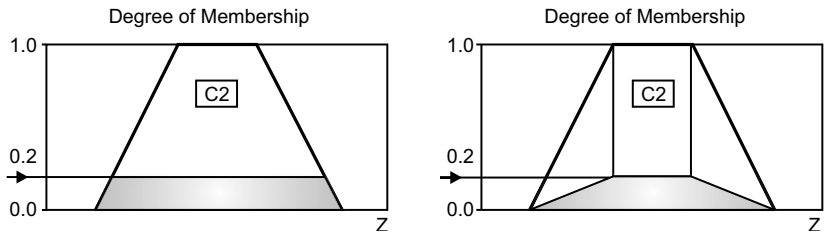


FIGURE 13.12

Aggregation of All Output

In this step, we combine result of the implication of each rule. Aggregation is the process of combining the output of each rule (fuzzy set) into a single fuzzy set. The aggregation step is done once only before the final step (defuzzification). The input to the aggregation process is the list of truncated or scaled output functions (the output of the implication process for each rule) and the output of the aggregation is a fuzzy set for each output variable.

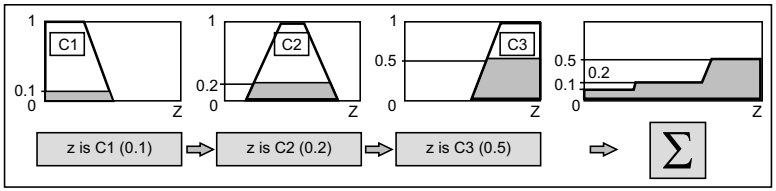


FIGURE 13.13 Aggregation of the Rule Output

Defuzzification

In defuzzification, the output of the aggregation step acts as input (aggregated fuzzy set) and gives a single crisp number as an output. This process is complex, since it is not possible to directly translate a fuzzy set into a crisp value.

There are several defuzzification methods, but the most popular one is the centroid technique. It finds the point where a vertical line would slice the aggregate set into two equal masses. The center of gravity (COG) can be calculated as follows:

$$\text{COG} = \frac{\int_a^b \mu_{\dot{A}}(x)_{xd}}{\int_a^b \mu_{\dot{A}}(x)d}$$

Center of Gravity (COG)

$$\text{COG} = \frac{(0+10+20)\times 0.1+(30+40+50+60)\times 0.2+(70+80+90+100)\times 0.5}{0.1+0.1+0.1+0.2+0.2+0.2+0.2+0.5+0.5+0.5+0.5} = 67.4$$

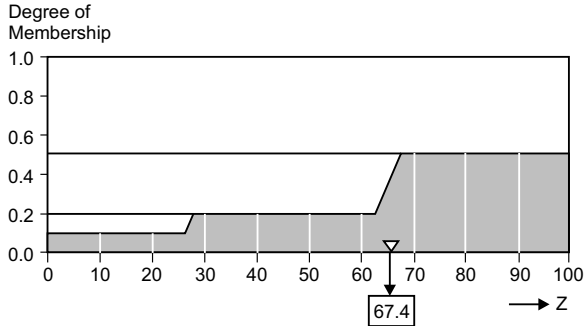


FIGURE 13.14 The Calculation of the COG

In Figure 13.14, the crisp output is obtained by applying the Mamdani method of inference to the above defined three rules.

13.3.2 Sugeno Inference Method (TSK Fuzzy Model of Takagi, Sugeno, and Kang)

In Mamdani-style inference, there is a need to apply the centroid method of defuzzification to find the centroid of a two-dimensional shape, but this is not efficient from a computation point of view. Michio Sugeno suggested another method of inference. A single spike, called a *singleton*, is used as the membership function of the rule consequent. In the Sugeno method, the rule is in the following form:

IF x is A and y is B THEN $z = f(x,y)$

Here, A and B are the fuzzy set in the antecedent and x and y are the input variables (linguistic variables), while $z = f(x, y)$ is a crisp function in the consequent part of the rule.

The Sugeno method and Mamdani method rules differ in their consequent part. In the Mamdani method, the rules have a fuzzy set in the consequent part of rule, but in the Sugeno method, a mathematical function of the input variables is used in the rule's consequent part.

In the Sugeno method, $z = f(x, y)$ is a polynomial in x and y. The order of the TSK model is defined by the order of the polynomial.

- In zero-order TSK models, z is a constant.

  Sugeno Type 1

- If X is small and Y is small, then $z = -x + y + 1$ (Rule)

  Sugeno Type 0

- In a zero-order Sugeno fuzzy model, the fuzzy rule is in the following form:

  IF x is A AND y is B, THEN z is k

  where k is a constant.

In this case x, y, and z are linguistic variables and A and B are fuzzy sets; the output of each fuzzy rule is constant. All consequent membership functions are represented by singleton spikes. For example,

IF X is small and Y is small, THEN $z = 5$ (rule)

In the Sugeno method, the central average defuzzifier is used.

Steps in the Sugeno Inference Method

- Fuzzification
- Rule evaluation
- Aggregation of rule output
- Defuzzification

That means the Sugeno method also follows four steps, as in the Mamdani method, for inferencing. Let us explain the Sugeno method of inference by taking an example of the rules.

Rule 1: IF x is $A3$ OR y is $B1$ THEN z is $k1$

Rule 2: IF x is $A2$ AND y is $B2$ THEN z is $k2$

Rule 3: IF x is $A1$ THEN z is $k3$

Here, x and y are input variables, $A1$, $A2$, $A3$, $B1$, and $B2$ are fuzzy sets.

Z is the output variable and $k1$, $k2$, and $k3$ are constants.

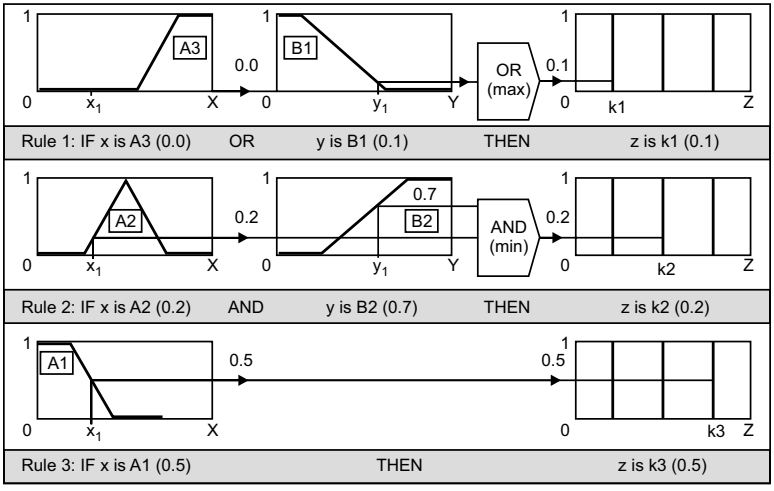


FIGURE 13.15 The Sugeno Style Rule Evaluation

Figure 13.15 shows the fuzzification of the crisp input, fuzzy operator, and fuzzy implications, but in the Mamdani method, the fuzzy set is the output of the fuzzy implication step.

In the Sugeno method, the output of the fuzzy implication step is not a fuzzy set.

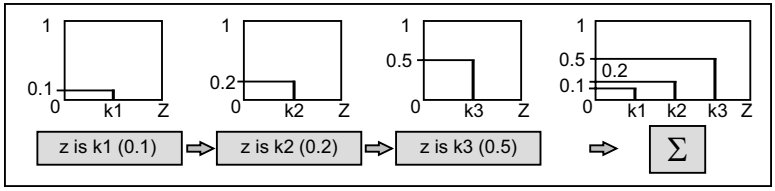


FIGURE 13.16 The Aggregation of All Rules' Output

$$\text{WA} = \frac{\mu(k1)\times k1 + \mu(k2)\times k2 + \mu(k3)\times k3}{\mu(k1) + \mu(k2) + \mu(k3)}$$

$$= \frac{0.1\times 20 + 0.2\times 50 + 0.5\times 80}{0.1 + 0.2 + 0.5} = 65$$

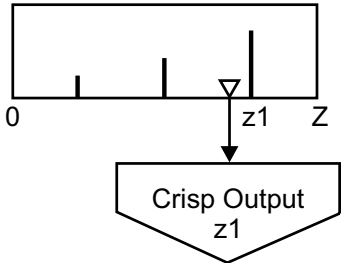


FIGURE 13.17 The Weighted Average Calculation and Defuzzification Step for Obtaining Crisp Output

13.3.3 Choosing the Inference Method

If non-numeric information is required in the result, then we use the Mamdani method. The Mamdani method is the most commonly used method, as fuzzy rules are the direct result of the human expert's advice, which are then expressed in the form of fuzzy rules. That is, human expertise is described in a human understandable manner. However, the Mamdani method is not computationally effective, that is, it has the burden of computation.

The Sugeno method is a good choice for control problems as it is a computationally effective method of inference. If processing speed and memory usage matter, then the Sugeno method is a good choice.

13.4 The Fuzzy Inference Process in a Fuzzy Expert System

The general meaning of inferences is to extract or draw some conclusions from the existing data. Fuzzy inference also means the same thing, that is, it is about drawing some conclusion from the existing data (input values and truth values) by using fuzzy logic concepts under the condition of uncertainties and ambiguities. Such inferred conclusions are then used in making decisions. So, fuzzy inference means mapping from a given input to an output using fuzzy logic concepts. Fuzzy inference in a fuzzy expert system differs from conventional logical inference, as in fuzzy inference, the modification of data is done by using fuzzy rules.

e.g., IF (P) THEN ($B' = B$)

Here, P is the antecedent part of the rule, B is the existing data; B' is the revised data, with a different truth value.

There are possible three types of inference in a fuzzy expert system.

13.4.1 Monotonic Inference

In monotonic inference, the truth value of the conclusion part may increase or remain the same, but it can't decrease, e.g., let the grade of membership of a fuzzy set B be 0.9. Now, a rule fires with the antecedent confidence of 0.4. In monotonic inference, this new information should be discarded and the rule would fail. The grade of membership of B would remain at 0.9.

The formula for B', the new truth value of B', using monotonic inference is $B' = P$ OR B, where P is the antecedent truth value.

- If A and A' represent single-valued data (integers, floats, strings), with the same value but different truth values A and A', then

  $$P = \max(A, A')$$

- If A and A' are discrete fuzzy sets, fuzzy numbers, or membership functions,

  $$P = \max(A(x), A'(x)) \text{ for every } x \text{ in } A, A'$$

Monotonic reasoning is useful when modifying values of scalar data or grades of membership of discrete fuzzy sets.

13.4.2 Non-Monotonic Inference

In non-monotonic inference, the truth value of the conclusion part may increase, decrease, or remain unchanged. For example, let the grade of membership of a fuzzy set *B* be 0.9. Now, a rule fires with the antecedent confidence of 0.4. In non-monotonic inference, the rule would not fail and the grade of membership of *B* would decrease to 0.4. But if the new rule fires with a confidence of 1.0, we would increase the truth value of B to 1.0

- For single-valued data, a formula for *B*', using non-monotonic inference is

  $$B' = P$$

- If B and B' are not single-valued data, the formula for B', using non-monotonic inference is

  $$B'j(x) = A'(x) \text{ AND } A(x)$$

Non-monotonic reasoning is useful when modifying truth values directly.

13.4.3 Downward Monotonic Inference

In downward monotonic inference, the truth value of the conclusion part may decrease or remain the same, but it can't increase. A formula for *B*', the new truth value of *B*, using downward monotonic inference is

$$B' = P \text{ AND } B$$

Downward monotonic reasoning is useful when combining the grade of membership of a linguistic variable with its membership function prior to defuzzification.

13.5 Types of Fuzzy Expert Systems

13.5.1 Fuzzy Control

Fuzzy control has been widely accepted, first in Japan and then throughout the world, after fuzzy process control was first successfully achieved by Mamdani. A fuzzy control system follows the same steps of inference as described above (fuzzification, fuzzy implications, aggregation of rule output, and defuzzification), that is, it accepts crisp numbers as input, then the fuzzification step translates the input numbers into linguistic terms such as

slow, medium, and fast (fuzzification). The rules then map the input linguistic terms onto similar linguistic terms describing the output. Finally, the output linguistic terms are translated into a crisp output number (defuzzification). A typical fuzzy control rule might be

IF input1 is High AND input2 is Low THEN output is Zero

A fuzzy control system only deals with numeric data because the domain of a fuzzy control system is well defined.

13.5.2 Fuzzy Reasoning

Fuzzy reasoning can deal with both numeric and non-numeric data, as the domain is not clearly defined. The fuzzy reasoning system also has the same four steps of reasoning. Here is the syntax for rules of the fuzzy reasoning system:

IF symptom is Depressive and duration is about 6 THEN diagnosis is Major_depression

This rule is different from fuzzy control rules, as in fuzzy control rules, the input symbol can only be numeric. In the case of fuzzy reasoning, the symptom is a set of linguistic terms. "Depressive" is a member of the linguistic term set. Similarly, "diagnosis" in the fuzzy control rule must have a scalar value, but in fuzzy reasoning, "diagnosis" is a set of linguistic terms.

13.6 Fuzzy Controller

A **fuzzy control system** is that system in which fuzzy logic is embedded into a control system that makes it different from conventional control systems. We know that human beings have the knowledge of how to control a control system, so by converting human knowledge into rules, a fuzzy control system can be made. Fuzzy control systems provide a technique for representing, manipulating, and implementing a human's heuristic knowledge about how to control a system. In a fuzzy control system, human expert knowledge in the form of fuzzy rules is used in designing the controller to control a complex process. That is, fuzzy rules, approximate reasoning, and fuzzy linguistic variables are used in designing a fuzzy controller. The concepts of fuzzy logic provide a way that is efficient and resourceful to solve complex control processes.

In fuzzy control, fuzzy linguistic variables are combined with fuzzy logic. A fuzzy controller is strongly based on fuzzy logic concepts and also uses differential equations, while differential equations are only the base of a conventional control system. In fuzzy control, more focus is on using heuristics, that is, a focus on the use of rules to represent how to control a plant rather than ordinary differential equations (ODE). This approach has some advantages in that the representation of knowledge in rules seems more eloquent and natural to some people. A fuzzy controller is fuzzy code that is designed for controlling complex real time processes. Fuzzy controllers can be incorporated into anything, from small circuits to large mainframe computers. Fuzzy controllers are used for temperature control and in anti-brake systems.

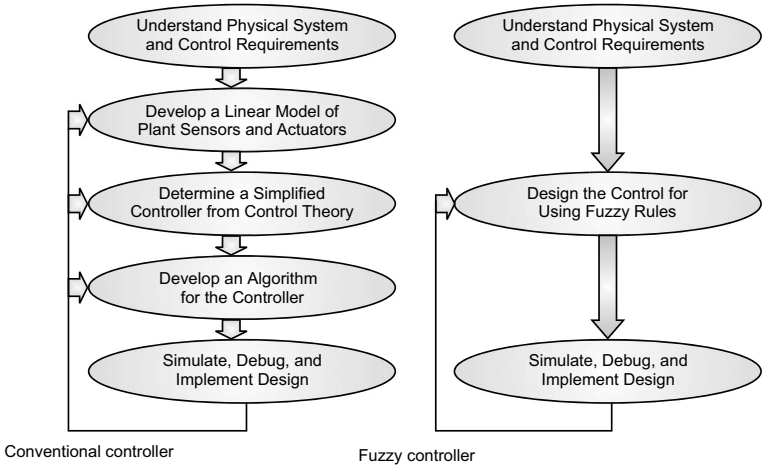


FIGURE 13.18 The Difference Between a Conventional Control System and Fuzzy Controller

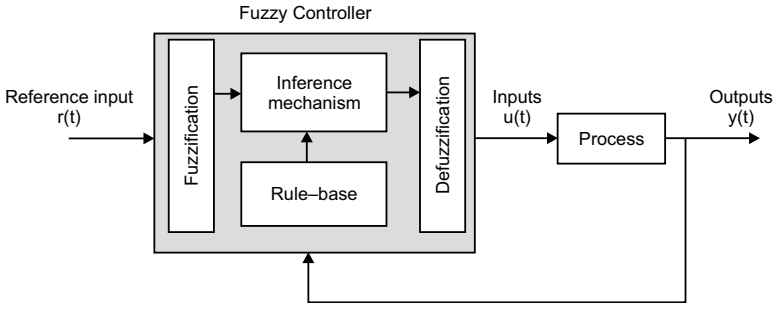


FIGURE 13.19 Fuzzy Controller Architecture

$y(t)$ is the plant output, $u(t)$ is the process input, and the reference input to the fuzzy controller is denoted by $r(t)$.

Figure 13.19 clearly shows that a fuzzy controller works in a closed loop system. That is, the fuzzy controller gathers plant output data $y(t)$, compares it to the reference input $r(t)$, and then decides what the plant input $u(t)$ should be to ensure that the performance objectives will be met.

It is the control engineer who gathers information on how the fuzzy controller should act in the closed-loop system. There are two ways of getting information: either obtain the needed information from a human expert or, sometimes, the control engineer will write down the rules for the fuzzy controller without any expert help. These "rules" basically say, "If the plant output and reference input are behaving in a certain manner, then the plant input should be some value." Then, a whole set of such "IF-THEN" rules are loaded into the rule-base, and an inference strategy is chosen. The system is then ready to be tested to see if the closed-loop specifications are met.

13.6.1 Components of a Fuzzy Controller

There are four components of fuzzy controllers: rule base, inference mechanism, fuzzification interface, and defuzzification interface.

13.6.1.1 Rule Base

The rule base is a depository of the fuzzy IF–THEN rules that are used for making decisions. Knowledge of how to control a process is stored in the rule base in the form of rules. That knowledge can be obtained from a human expert by the control engineer or the control engineer can develop control rules by studying the plant's dynamics. For example, in the cruise control problem, knowledge can be obtained from a human expert, or if someone has experience in driving, then he can write the rule himself without expert help. For instance, one rule that a human driver may use from his driving experience is "If the speed is lower than the set-point, then press down further on the accelerator pedal." A rule that would represent even more detailed information about how to regulate the speed would be "If the speed is lower than the set-point AND the speed is approaching the set-point very fast, then release the accelerator pedal by a small amount."

This second rule characterizes our knowledge about how to make sure that we do not overshoot our desired goal (the set-point speed). It is better to load very detailed expertise into the rule base to enhance our chances of

obtaining better system performance. To write rules for the rule base, the control engineer will use a linguistic description of natural language that is used by an expert for describing how to best control the plant. After writing all the rules using linguistic variables, these are loaded into the rule base.

The syntax of a control rule is as follows:

IF x is A and y is B THEN z is C

Here x, y, and z are linguistic variables representing process state variables and control variables, and A, B, and C are linguistic values of the linguistic variables.

For example, "IF angle is Z and angular velocity is NL THEN speed is NL."

Another form of the control rule is as follows:

Ri: IF x is *Ai*, …. AND y is *Bi* THEN z = fi (x, …,y)

where *fi* (x, …,y) is a function of the process state variables x, …, y.

Both fuzzy control rules have linguistic values as inputs and either linguistic values or crisp values as the output.

Consider the inverted pendulum control problem. Here, y denotes the angle that the pendulum makes with the vertical (in radians), l is the half-pendulum length (in meters), and u is the force input that moves the cart (in Newtons). We have the user denote the desired angular position of the pendulum. The goal is to balance the pendulum in the upright position (i.e., r = 0), when it initially starts with some non-zero angle off the vertical (i.e., $y \neq 0$).

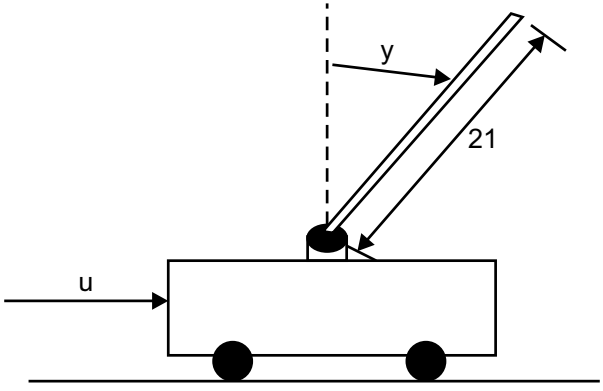


FIGURE 13.20 An Inverted Pendulum

Let us assume that the expert says that she or he will use

$$e(t) = r(t) - y(t) \text{ and } \frac{d}{dt}\, e(t)$$

as the variables on which to base decisions for controlling the pendulum on the cart.

Now, the control engineer, by using linguistic description, can convert these equations into rules. For the inverted pendulum,

"error" describes $e(t)$

"change-in-error" describes $\frac{d}{dt}\, e(t)$

"force" describes $u(t)$.

Now, let us assume for the pendulum example that "error," "change-in-error," and "force" take on the following values, called linguistic values.

"neglarge"

"negsmall"

"zero"

"possmall"

"poslarge"

We are using "negsmall" as an abbreviation for "negative small in size" and so on for the other variables. Such abbreviations help us keep the linguistic descriptions short yet precise. For an even shorter description, we could use integers:

"-2" to represent "neglarge"

"-1" to represent "negsmall"

"0" to represent "zero"

"1" to represent "possmall"

"2" to represent "poslarge"

Now we will use the above linguistic quantification to specify a set of rules (a rule-base) that captures the expert's knowledge about how to control the plant. For example,

IF error is neglarge and change-in-error is neglarge THEN force is poslarge

A convenient way to list all possible rules for the case where there are not too many inputs to the fuzzy controller is to use a tabular representation.

13.6.1.2 Fuzzification

This interface modifies the inputs so that they can be interpreted and compared to the rules in the rule-base. The *fuzzifier* transforms crisp measured data (e.g., the speed is 10 mph) into suitable linguistic values (i.e., in the fuzzy sets, for example, the speed is too slow).

It converts the controller inputs into a fuzzy set that the inference mechanism can easily use to activate and apply rules. In this step, we have to use fuzzified the data and create membership values for that data and put them into the fuzzy set. For example, in the case of the inverted pendulum problem, fuzzification is defined as follows:

The fuzzification process is the act of obtaining a value of an input variable (*e.g.*, $e(t)$) and finding the numeric values of the membership function(s) that are defined for that variable.

For example, if $e(t) = \pi/4$ and $\frac{d}{dt}\ e(r) = \pi/16$

then the fuzzification process is used to find how much these inputs belong to their fuzzy set. It finds out the membership function values, which are then used in the fuzzy inference process for rule evaluation.

13.6.1.3 Inference Mechanism (Inference Engine)

In the inference mechanism, the control rules stored in the rule base are evaluated to determine which control rules are relevant at the current time and what should be the input to the plant. In this, the mechanism's expert decision-making capability for controlling a system is emulated. In approximate reasoning, the generalized modus pones plays an important role. The generalized modus pones can be rewritten as

Premise 1: IF x is A, THEN y is B. Premise 2: x is A'

Conclusion: y is B'

where A, A', B, and B' are fuzzy predicates (fuzzy sets or relations) in the universal sets U, U', V

and V', respectively. In general, a fuzzy control rule (e.g., Premise 1) is a fuzzy relation.

According to the compositional rule of the inference conclusion, B' can be obtained by taking the composition of fuzzy set A' and the fuzzy relation (here, the fuzzy relation is a fuzzy implication) A B.

How Do We Decide Which Rules to Use?

The inference process generally involves two steps:

- The premises or antecedents of all the rules are compared to the controller inputs to determine which rules apply to the current situation. Basically, the rules that are more certain to apply to the current situation are chosen.
- After choosing the rules, a conclusion is determined. The conclusions are characterized with a fuzzy set (or sets) that represents the certainty that the input to the plant should take on various values.
  - ♦ For doing the inference, we need to quantify the premise of each of the rules with fuzzy logic.
  - ♦ We use fuzzy logic to quantify the meaning of the linguistic variables, linguistic values, and linguistic rules that are specified by the expert. A rule can have multiple parts in the premise, so we need to quantify the meaning of the premises of the rules that are composed of several parts, each of which involves a fuzzy controller's input.

For example: IF error is zero **AND** change-in-error is possmall **THEN** force is negsmall.

In the above rule, there are two parts of the antecedent combined using the AND operator. By applying fuzzy logic and the operator, we first quantify the meaning of the whole antecedent.

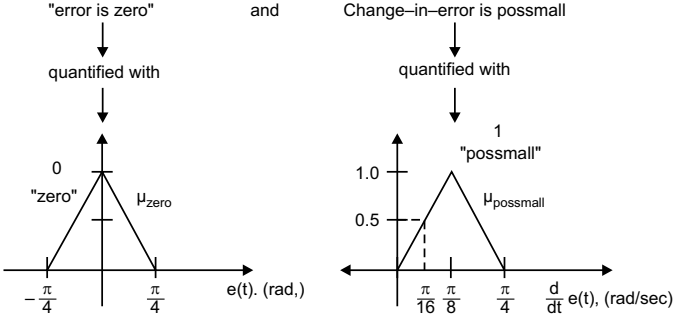


FIGURE 13.21 Membership Functions of the Premise Terms

Figure 13.21 clearly shows that we have quantified the meaning of the linguistic terms "error is zero" and "change-in-error is possmall" via the membership functions. Then, after quantifying the meaning of each part of the antecedent, we apply fuzzy logic and the operator to the membership function values of both terms that we obtained in order to get a single value.

Let us take an example to show how to quantify the AND operation, by supposing that

$$e(t) = \pi/8 \text{ and } \frac{d}{dt}\, e(r) = \pi/32$$

so that using above figure we see that

$$\mu_{\text{zero}}(e(t)) = 0.5$$

$$\mu_{\text{posmall}}\left(\frac{d}{dt}e(t)\right) = 0.25$$

Now, let us denote the premise of rule "error is zero and change-in-error is possmall" by $\mu_{\text{premise.}}$

After quantifying each antecedent, the next thing to do is to apply the AND operator. There are actually several ways to define it:

- **Minimum:** Define $\mu_{\text{premise}} = \min\{0.5, 0.25\} = 0.25$, that is, using the minimum of the two membership values.
- **Product:** Define $\mu_{\text{premise}} = (0.5)(0.25) = 0.125$, that is, using the product of the two membership values.

Determine Which Rules Are On

The next step after quantifying the meaning of the antecedent of the rule is to determine which rule is on.

We say that a rule is "on at time t" if its premise membership is as follows:

$$\text{function } \mu_{\text{premise}}\ \mu_{\text{premise}}\left(e(t), \frac{d}{dt}e(t)\right) > 0$$

Hence, the inference mechanism determines which rules are on to find out which rules are relevant to the current situation. In the next step, the inference mechanism will seek to combine the recommendations of all the rules to come up with a single conclusion to get the implied fuzzy set. Let us look at an example for showing how to reach a conclusion.

Consider the conclusion reached by the following rule:

IF error is zero and change-in-error is zero **THEN** force is zero

Using the minimum to represent the premise, we have

$$\mu_{\text{premise}(1)} = \min \{0.25, 1\} = 0.25$$

This means we are 0.25 certain that this rule applies to the current situation. The rule indicates that if its premise is true, then the action indicated by its consequent should be taken.

For the above rule, the consequent is "force is zero." The membership function for the conclusion reached by rule, which we denote by $\mu_{(1)}$ is as follows:

$$\mu_{(1)} = \min \{0.25, \mu_{\text{zero}}(u)\}$$

Notice that the membership function $\mu_{(1)}(u)$ is a function of u and that the minimum operation will generally "chop off the top" of the $\mu_{\text{zero}}(u)$ membership function to produce $\mu_{(1)}(u)$.

We will do the same for every rule, that is, go on and find out the conclusions reached so far.

Thus, the input to the inference process is the set of rules that are on, and its output is the set of implied fuzzy sets that represent the conclusions reached by all the rules that are on.

13.6.1.4 Defuzzification

Defuzzification is the interface that is the last and final component of a fuzzy controller. It used the conclusions obtained from the inference mechanism (the implied fuzzy set) and converts the conclusions reached by the inference mechanism into the inputs to the plant.

Thus, we can say like that defuzzification works on the implied fuzzy sets that are produced by the inference mechanism and combines their effects to provide the "most certain" controller output (plant input). In defuzzification control, the output refers to the control action.

There are many defuzzification methods, out of which the COG (Center of Gravity) and mean of the maxima (MOM) are the most commonly used methods of defuzzification.

Center of Gravity

$$\mu_{\text{crisp}} = \frac{\sum_i b_i \int \mu_{(i)}}{\sum_i \int \mu_{(i)}}$$

This is the formula for computing the center of gravity.

$\int \mu_{(i)}$ denotes the area under the membership function. b^i denotes the center of the membership function.

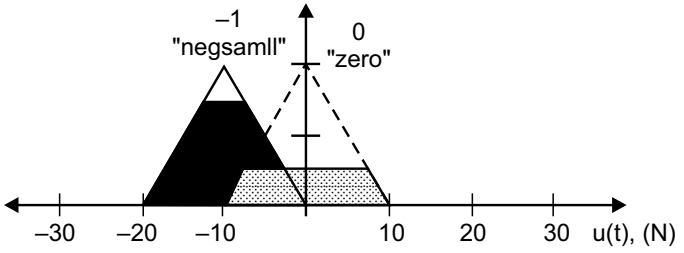


FIGURE 13.22 The Implied Fuzzy Set Produced by the Inference Mechanism

Now, by putting values into the above equation, we get the crisp output:

$$\mu^{\text{crisp}} = \frac{(0)(4.375)+(-10)(9.375)}{4.375+9.375} = -6.81$$

This value of – 6.81 acts as input to the pendulum for the given $e(t)$ and $\left(\frac{d}{dt}e(t)\right)$.

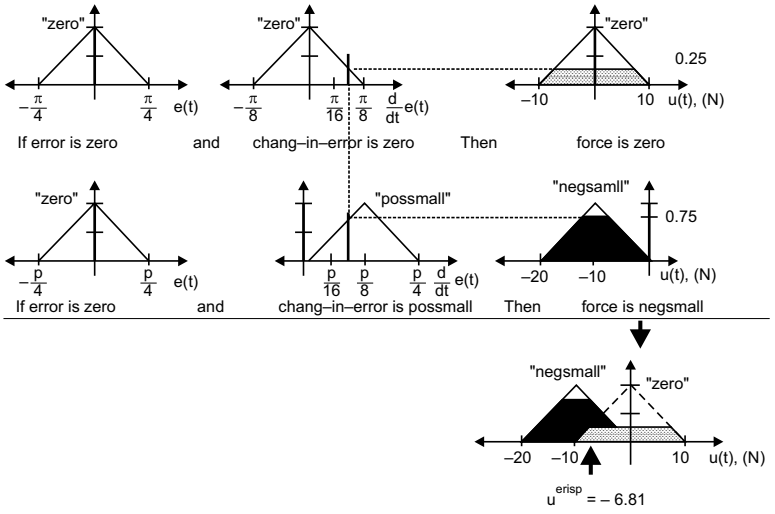


FIGURE 13.23 Graphic Representation of Fuzzy Controller Operations

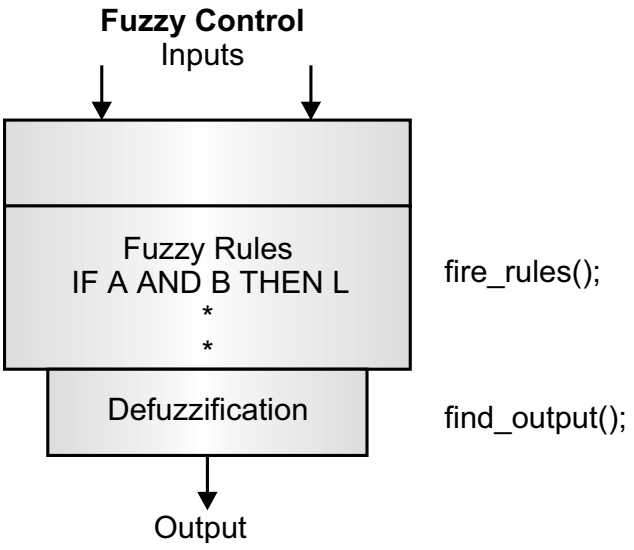


FIGURE 13.24

Steps in Building a Fuzzy Controller

- Define the input and output variables.
- Decide on the fuzzy partition of the input and output spaces and choose the membership functions for the input and output linguistic variables.
- Decide on the types and the derivation of the fuzzy control rules.
- Design the inference mechanism, which includes a fuzzy implication and a compositional operator, and the interpretation of the sentence connectives (AND).
- Choose the defuzzification method.
- Let us take an example that shows how to build a fuzzy controller:
- The temperature of a room equipped with a fan/air-conditioner should be controlled by adjusting the motor speed of the fan/air-conditioner.

Figure 13.25 describes the control of the room temperature. In this example, the goal is to design a motor speed controller for a fan.

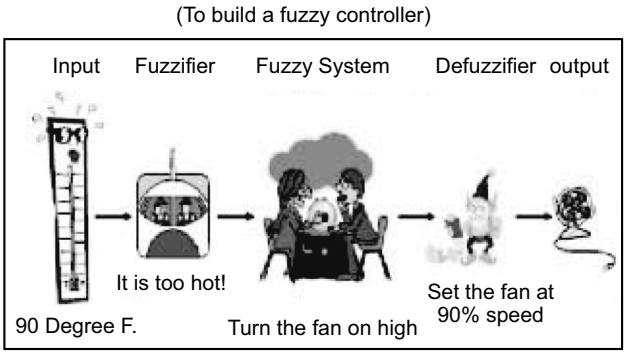


FIGURE 13.25 Controlling the room temperature.

Building a Fuzzy Controller

- **Step 1: Assign input and output variables.**

Let X be the temperature in Fahrenheit and Y be the motor speed of the fan.

- **Step 2: Pick fuzzy sets (Fuzzification).**

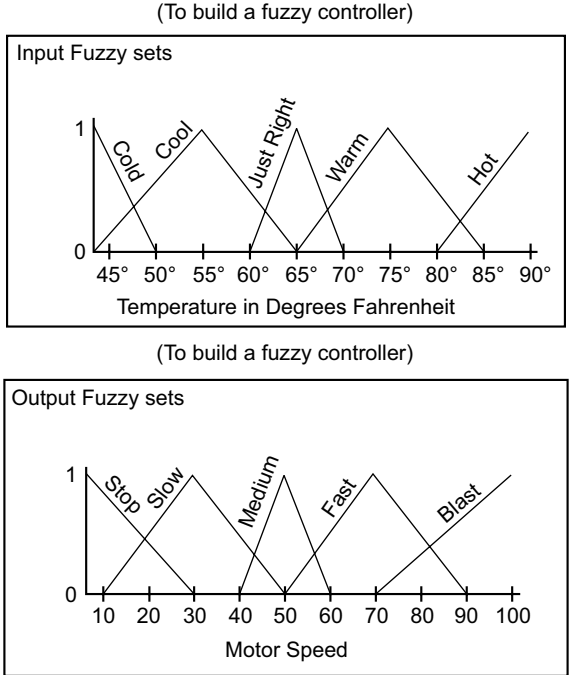


FIGURE 13.26 Choosing fuzzy sets(A).

Define the linguistic terms of the linguistic variables temperature (X) and motor speed (Y) and associate them with fuzzy sets. For example, five linguistic terms/fuzzy sets on X may be "Cold," "Cool," "Just Right," "Warm," and "Hot." Let the five linguistic terms/fuzzy sets on Y be "Stop," "Slow," "Medium," "Fast," and "Blast."

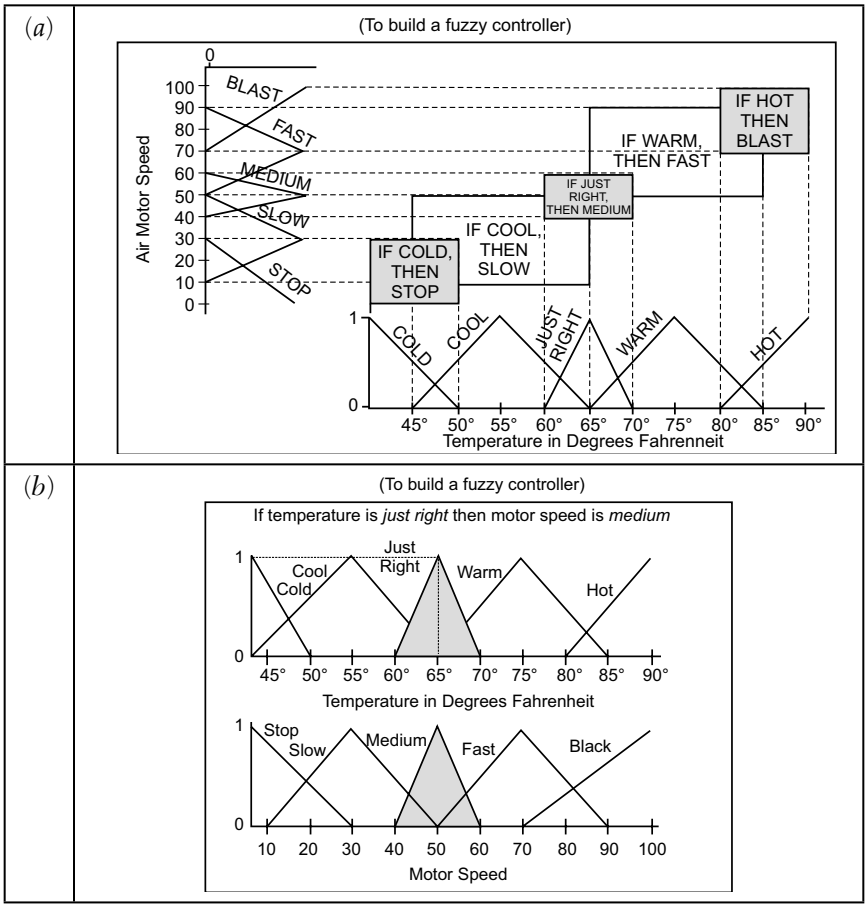
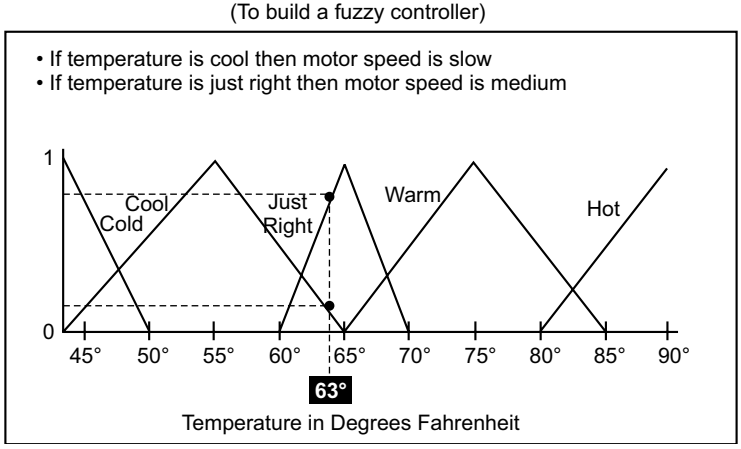


FIGURE 13.27 Motor speed as "blast." Building a Fuzzy Controller

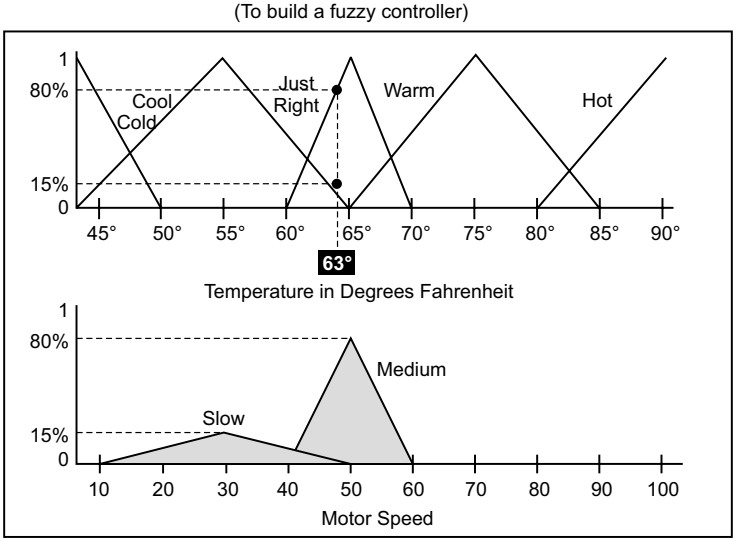
Step 3: Assign a motor speed set to each temperature set (rule or fuzzy controller)

- If the temperature is cold, then the motor speed is ***stop.***
- If the temperature is cool, then the motor speed is ***slow.***

- If the temperature is just ***right,*** then the motor speed is ***medium.***
- If the temperature is warm, then the motor speed is ***fast.***
- If the temperature is hot, then motor speed is ***blast.***



(**a**)



(**b**)

FIGURE 13.28 Choosing fuzzy sets(B).

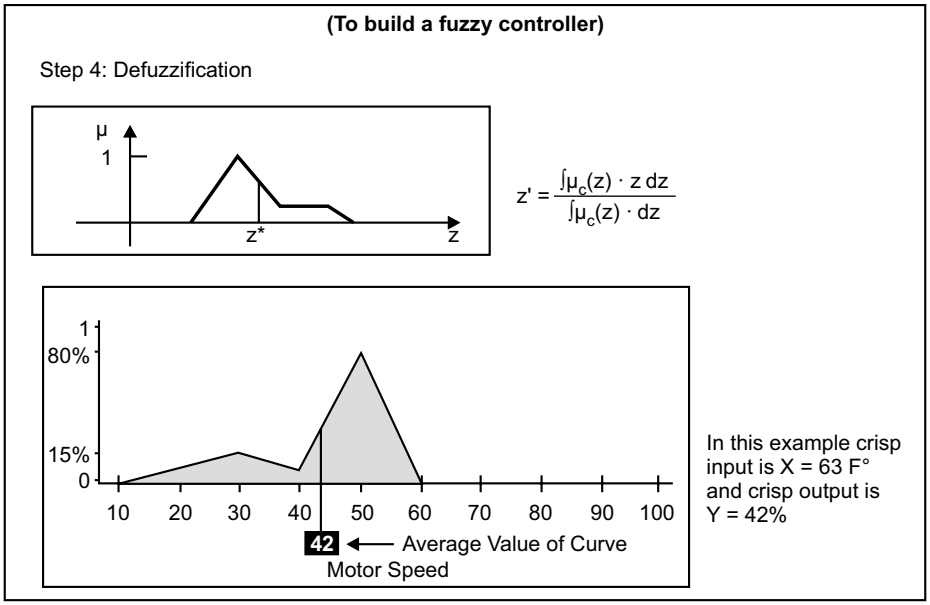


FIGURE 13.29 Choosing fuzzy sets(C).

13.6.2 Application Areas of Fuzzy Controller

Fuzzy systems have been used in a wide variety of applications in engineering, science, business, medicine, psychology, and other fields. In engineering some potential application areas include the following:

- *Aircraft/spacecraft:* Flight control, engine control, avionics systems, failure diagnosis, navigation, and satellite attitude control
- *Automated highway systems:* Automatic steering, braking, and throttle control for vehicles
- *Automobiles:* Brakes, transmission, suspension, and engine control
- *Autonomous vehicles:* Ground and underwater
- *Power industry:* Motor control, power control/distribution, and load estimation
- *Process control:* Temperature, pressure, and level control, failure diagnosis, distillation column control, and desalination processes
- *Robotics:* Position control and path planning

Why We Should Use Fuzzy Controllers

- can be easily modified
- can use multiple input and output sources
- much simpler than their predecessors (linear algebraic equations)
- very quick and cheap to implement

Exercises

Q1. Provide examples of how fuzzy systems are used?

Q2. What are two types of knowledge in a fuzzy systems knowledge base?

Q3. Define "defuzzification."

Q4. What are some practical applications of fuzzy controllers?

CHAPTER 14

LOGIC PROGRAMMING

14.1 Introduction

To represent problem operations, symbol patterns are used so that an intelligent action is executed. When these patterns are used, several potential solutions are generated, and among these solutions, a specific search is selected. The language that is used for an AI representation must

- handle qualitative knowledge
- permit new knowledge to be contingent on facts and rules
- allow the illustration of all-purpose principles
- capture complex semantic meaning.

In the study of AI , two programming languages are mainly used,

- **LISP (List Processing)**: This is the basic language that is used for AI problems, and it is used in a functional manner.
- **Prolog (Programming in Logic)**: This is used in a declarative manner to solve AI problems.

Prolog allows for programming in a logical language that utilizes symbolic or non-numeric computation. It is often used in AI where the exploitation of symbols and deductions can be manipulated in a simple way. The main aim of this this language is to explain the characteristics of objects and correlations between objects, and define the IF-THEN rules related with the properties and relations. In artificial intelligence, a number of problems

can be addressed when the solution of these problems involves the implementation of Prolog. The language can help achieve a certain goal in a certain situation and specifies what the situation and the goal are.

14.2 Difference Between C/C++ and Prolog

C/C++, Java, and Pascal are imperative languages in which a program is a specification of a sequence of instructions to be executed one after the other to solve a problem. The explanation of the problem is integrated completely with this specification., Typically, it is not used to distinguish between the clarification of the problem and the technique used for its solution. However, in logic programming, the clarification of the problem and the method for solving it are obviously separated from each other. Kowalski proposed an equation to define this separation and solve the problem. This equation can be expressed as

Algorithm = Logic + Control

In this equation, the term "logic" specifies the descriptive component of the algorithm, i.e., the explanation of the problem, and the term "control" specifies the component that tries to provide a solution with the help of the description of the problem. The logic component identifies what the algorithm is supposed to do; the control component indicates how it should be done.

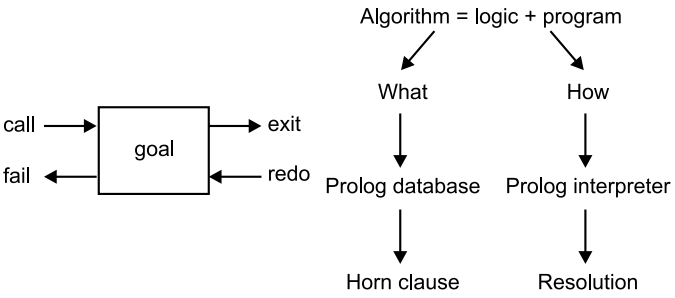


FIGURE 14.1 The Relationship between Prolog and Logic Programming

The difference between C and Prolog is that in C, the programmer tells the computer what to do, but in Prolog, the programmer tells the computer how to do it. Prolog is based on facts and rules: the programmer might start by using the facts (telling the computer the facts) and providing the rules. A Prolog program could then be used to ask the computer about the facts

already given and the computer would be able to give answers in the form of rules. When executing the program, the user asks a question and the solution is obtained. When the user asks a question to the computer, the runtime system searches through the database of facts and rules to determine (by logical deduction) the answer.

Prolog is a declarative (descriptive) language. A descriptive language is nonprocedural in its logic. A program written in this way not only defines exactly how the computational process is to be carried out, but consists of several declarations representing significant facts and rules. The solution to be mined is also expressed as a question to be answered and a goal to be achieved.

14.3 How Does Prolog Work?

To do work with Prolog, a Prolog program is run. It uses a database of facts and rules that define the relations between objects. Prolog programs are based on the Horn clause, which is a theory written in a subset of a predicate (or we can say that it is first order logic). A Horn clause consists of a result (headH, consequent) and a body (termsBi):

$$\mathbf{H} \leftarrow \boldsymbol{B}_1, \boldsymbol{B}_2, \ldots, \boldsymbol{B}_n$$

The deduction in Prolog is based on the modus ponens syllogism (if-then)

If $P(x)$ then $Q(x)$

$P(a)$

Therefore $Q(a)$

For example,

- If human(X) then mortal (X).
- human (socrates)
- Therefore: mortal(socrates)
- A pie is good = good (pie)

In Prolog, a relation's name is called a functor. In this example, "good" is a functor. A relation may include many arguments after the functor. All of these objects and relations are applicable for this language. When they are identified, they must be made precise by the facts and rules used for the objects and their interrelationships. When all the facts and rules are known, then a

definite problem may be turned into an ASA query relating to the objects and relationships between objects. We specify all the facts and rules to obtain this solution (for the objects and relationship between the objects).

Steps To Write a Prolog Program

- The program is a text file, and it is also sometimes called a database. This text file contains the facts, rules, and relations needed to describe the problem. The extension of the text file is *.pl.
- A window appears for you to run the facts and rules in a program; this is known as the query mode window. This window is represented by the "? – prompt." In this window, you ask questions related to the problem that describe the relations.
- When Prolog is started, the query mode window (?-) appears, indicating that you are in query mode. Now, you can load the program by writing the commands in Prolog. This is referred to as a file. This file is represented by file.pl, which contains your programs that are executable. When all of these are done, you can use all the facts and rules that are described in the program.

14.4 A Little History

In 1970, Colmerauer, Kowalski, Van Emden, and Marseille invented the logic programming language. In 1972, a professor of computer science at the University of Aix-Marseille in France, Alain Colmerauer, invented the first Prolog interpreter, which was used to compile Prolog programs. After the invention of the interpreter, a number of software products that used in Prolog were launched in 1982. All of these were insufficient for Prolog programming, so an extended version of Prolog was developed, known as CLP (Constraint Logic Programming) in 1995. CLP is defined as a language that represents knowledge in the form of facts and rules. Until the end of the 1970s, there was a very limited use of Prolog, but now it is more common. However, it was not often used in the academic world. After developing the compiler and interpreter, this programming language became more influential throughout the world, as well as in academia. The Prolog interpreter and compiler were invented by Warren and Pereira at the University of Edinburgh. Researchers mainly use Prolog because it permits the development of complex and general (extensive) programs in a shorter period of time than that needed to develop a C or Java program with a comparable functionality. Prolog is now used in many fields.

Main Applications

- Artificial intelligence: Prolog is used for expert systems and natural language processing.
- Databases: Prolog is used in databases for query languages and data mining.
- Mathematics: theorem proving and symbolic packages
- Compiler construction
- Work in the area of computer algebra
- The development of (parallel) computer architectures

14.5 Converting English to Prolog

Now, we will discuss how to convert a simple English statement into Prolog. In English statements, a simple sentence includes a noun, verb, and object. In Prolog only the facts, rules, and relationships among objects are used. To explain the relationship between an English statement and Prolog, we can say that a verb or adjective is replaced by the facts used in Prolog and the related noun written in parenthesis. Now, we let the computer know about these facts. English statements can be converted into Prolog.

14.6 Goals

A goal is a statement starting with a predicate, most likely followed by its arguments. A goal is valid when a number of arguments is the same as the fact and rules (means predicate) that appear in the consulted program. A goal can be an atom or a functor pursued by some arguments. All atoms are enclosed in parentheses and a comma is used to separate two or more than two atoms. For example, a(X, Y).

When a Prolog interpreter is running, you may see the prompt "?-" on the screen. The user types the goal or query at this prompt, which is totally based on the facts and rules that are used in the program. The main aim of this step to find out whether the statement represented by the goal is true according to the facts and rules that are used in the knowledge database (i.e., the consulted program).

14.6.1 How Prolog Satisfies Goals

When a user asks questions to a computer, it looks in the database where the facts are stored. Note that the computer knows only the information loaded into it. Goals are used to find a match. Prolog will give a list of facts that are already loaded into it. It generates the goals to satisfy a question. To explain how goals work, let's use an example. We have a Prolog statement "eats(naman, pizza)." This statement is already loaded in the database (as mentioned above), and when it is fulfilled by finding it as a fact, Prolog will give the answer "yes" or "true." Now suppose this statement is provided in query mode; then, the sentence is written as "eats(X, pizza)." It is also fulfilled by finding facts about someone who eats pizza, The Prolog answer to this query relates to the who satisfies it and provides the name of the variable X. Now, suppose another statement is "Delhi is a city," and the user asks it for this query. First, the program looks in the database, and matches this statement with all other statements that are loaded into the database. If no match is found, it will report "fail" or "no."

A goal is a combination of several terms, such as a1, a2, ... to the a*n* term. The Prolog inference engine first selects the leftmost term in the goal to satisfy the goal. In this case, a1 is the leftmost term, and this term matches with the clause that is used in the database. It then checks the database to find the clause. If this clause is found, then the leftmost term (a1) in the goal is substituted with the body of the clause. Now, the final goal becomes b1, b2, b3, a2, ... b*n*.

A Prolog goal is also used to represent the flow of control. It has four ports that are used in the representation of the goal. The four ports are call, exit, redo, and fail. First, the goal is called. When this goal is matched in our database, i.e., a successful match occurs, then it is in the "exit" state. If this match is not found, i.e., a failure occurs, then it is in the "fail" state. When a semicolon is used, then the goal is retired and it enters the "redo" state. A goal and its ports are represented in Figure 14.2.

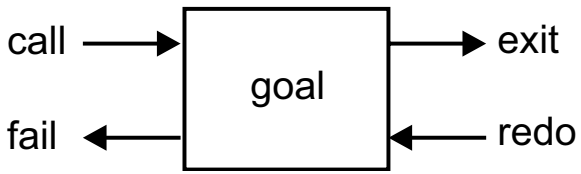


FIGURE 14.2 The Ports of a Prolog Goal

Call: This is the initial port in which the searching is done. To search the clause, a goal is called.

Exit: When the goal is satisfied, then the exit port is defined. It sets a place indicator at the clause and combines the variables properly.

Redo: When the goal is retired, then the redo port is used. It unbinds the variables and restarts the search at the place indicator.

Fail: When no match is found, i.e., a clause matches with the goal, then this state occurs.

14.7 Queries

English	Prolog
Ram is the father of Mohan.	father(ram, mohan).
Seeta is the wife of Ram.	wife(seeta, ram).
Sohan eats an apple.	eats(sohan,apple).
Naman eats pizza.	eats(naman, pizza).
Ram bought pizza for Mohan.	bought (ram,pizza,mohan).
Ram is tired.	tired(ram).

Notice that we aren't using capital letters to start the names; we reserve capital letters or terms starting with them for variables.

Now, we write these facts and load them into Prolog. The Prolog language uses a query mode window known as a "prompt," and it is represented by "?-." We know that Prolog provides question/answer statements, so on this prompt, the user can ask any question related to the problem. Note that some statements are in question form in English. In these statements, the user can ask a question to Prolog. All the statements (questions) asked by user are matched with the database (Prolog program), which contains these facts (already in the form of a statement); if a match is found, then the answer given by Prolog is "true," otherwise, it is "fail."

English question	Prolog (at query mode, prompt ?-)	Prolog responds
Is Ram the father of Mohan?	father(ram, mohan).	yes (or true)
Is Ram tired?	tired(ram).	yes
Is Seeta tired?	tired(seeta).	no (or fail)
Who is tired?	tired(X).	X = ram
Who is the wife of Ram?	husband (X,ram).	X = seeta

All these English statements are loaded into the database when we ask Prolog about Ram being the father of Mohan, and then it searches the database of the above statement. We do not have to change the form of the Prolog statement we used to tell Prolog that Ram is the father of Mohan. The statement containing "who" means that the user asks a question to the computer. For this, a variable is used, which is represented by "X." The variable X is used to define the "who" statement.

In Prolog, a query mode window is used to write the query. It is a prompt and represented by "?." When the user asks a query, this query is related with the facts and rules. Prolog can provide a solution to this query. This process is known as querying the system. A query is also written as a statement initialized with a predicate and followed by its arguments. To represent a query, a variable X is used that defines "who." A query is valid when the number of arguments and at least one fact or rule in the query is the same as that from the consulted program. A query refers to asking a question about what values are used to make the given statements true.

The following examples represent how goals and queries are evaluated.

```
?- parent (ram,mohan)
```

This is a goal proving that "Ram is the parent of Mohan." Now this Prolog statement is checked in the database where all the facts are stored (as mentioned above). These facts are searched in the database when a match is found, and then Prolog will respond "true."

```
?-parent (ram,mary)
```

This is a goal showing that "Ram is the parent of Mary". Now this type of fact is searched in the database. But in our database, this type of fact is

not found because we have a database that does not contain this type of statement, so Prolog responds "no," i.e., "fail."

```
?-parent (X, mohan)
```

This is a query defined as asking for the person who is the parent of Mohan. For this query, the fact and rule related with this query are stored in our database, i.e., the statement "Ram is the parent of Mohan" is already loaded in the database. Now this query is compared with this fact and it is matched, so Prolog will give an answer this query. Prolog will report "yes."

```
?-mama (shyam,X)
```

In this Prolog statement also, the X variable is used. It means this is a query asking for the person who calls Shyam "mama." To match this query with any facts or rules already stored in our database, Prolog checks the database. If a match is found, then the response will be "yes." But in our database, no such query is found, so it will report "fail" or "no."

14.8 Clauses

There are two types of clauses:

- Facts
- Rules

14.8.1 Facts

A fact consists of a particular item or relation between items. To represent a fact, a predicate is used in Prolog. It is written in the form of its atomic value, where the predicate is a name that is given to the relation and the atom is a constant value (written in lowercase letters). For example, *Pari eats pizza*, would be "eats (pari, pizza)."

A relationship is written first (typically, the predicate of the sentence), and it is always written in lowercase letters. To represent several objects in Prolog, a comma is inserted between two or more objects, such as *pari* and *pizza* (the two objects in the above example). It is always written in small brackets (round brackets). To end the statement (fact), a full stop (.) is used at the end. Facts also contain simple rules of syntax. Facts consist of a letter or number combination, and a special character underscore (_).

14.8.2 Rules

To infer facts from other facts, rules are used. The programmer describes rules similarly to the facts. Rules are also represented in the form of predicates, such as predicate(Var1,…):- predicate1(…), predicate2(…), … where Var1 is a variable, usually beginning with an uppercase letter.

For example, "Vishal likes bikes if they are blue" = likes(vishal, bikes):- blue(bikes). In Prolog, ":-" is pronounced "if."

Now, we will discuss how to represent facts and to query them. We will also discuss rules. Rules allow us to make conditional statements about our world. Each rule can have many variations, known as clauses. These clauses are used to provide us with different choices about how to perform inferences about our world. Let's take an example that tells how to represent facts and rules.

"All men are mortal."

This fact can be stated by the following Prolog rule: mortal(X) :-human(X).

The clause can be read in two ways (called either a declarative or a procedural interpretation). The declarative explanation is "For a given X, X is mortal if X is human." The procedural explanation is "prove both the goal and subgoal. The main goal is X is mortal and the subgoal is X is human."

Rule 1:

To explain another rule in Prolog language with the same example used above, now consider the fact "Ashoka is human;" our program now looks as follows:

```
mortal (X) : human(X).human (ashoka).
```

If we now create the question in Prolog

```
?- mortal(ashoka)
```

The Prolog interpreter would give the answer as follows:

yes

Rule 2:

To solve the query "?-mortal(ashoka)," a rule has to be defined that proves someone is mortal, so we had to prove them to be human. Now our aim is to find the subgoal also, so Prolog produces the subgoal, such as

"human(ashoka)." Facts and rules are generated by a matching process. In the matching process, facts are matched with the database that stores some statements. If we have facts, it means the matching is done (found), and then Prolog will generate the answer "yes," otherwise, it will report "fail."

Rule 3

Rule 3 explains the query. A query is defined by a variable that means "who." In a query, X is always used to represent the "who" statement. To explain the above example, we might want to see if there is somebody who is mortal. This query is represented by the following line.

```
?- mortal(X)
```

The Prolog interpreter responds.

X = ashoka yes

This line shows that Prolog has proved the goal by binding the variable X to ashoka. This also proves the goal and subgoal. The goal is proved by someone being mortal by proving, in the subgoal, that they are human. Prolog will respond to the answer by asking either if there was a human or not. All of these actions are done by the matching process. This process matches the clause "human(ashoka)" if it is found in our database, and then a variable X is bound with ashoka. This is the parent goal, which defines the binding, and the response is written in the form of a printout.

Rule 4

Sometimes, we may want to identify abnormal (unusual) ways of proving a particular thing. This is done by using different rules and facts with the same name. Let's take an example that defines Rule 4. We can represent the sentence "Something is fun if it's a green toy or a cherry car or it is ice cream" as follows:

```
fun(X) :
green(X),   toy(X).
fun(X) :
cherry(X),
car(X).
fun(ice_cream).
```

This is a rule in which we have three ways of finding out if something is fun. This is done if it is a green and a toy or cherry and a car, or if it is ice cream. All facts (options) are represented by different clauses in Prolog. All of these clauses are done by predicates, such as "fun." Prolog will start from the first clause (be it a rule or fact) of "fun" and try that. If that does not succeed, it will try the next clause. A "fail" will be generated when there is no more success.

Rule 5

All identically-named variables within a particular rule (e.g., all occurrences of, say, X in the first "fun" rule below) are constrained to have one and the same instantiation for each solution to a particular query. The name of the same variable in separate rules is not completely dependent on the others, so different variable names have been used. Let us consider an example program:

```
fun(X) :-
red(X), car(X).
fun(X) :- blue(X),
bike(X).
Looks to Prolog Like
fun(X_1) :-
      red(X_1),
      car(X_1).
fun(X_2) :-
      blue(X_2),
      bike(X_2).
```

The scope of the variable name is the per-individual rule, which is known as a clause. In this program, the same variable emerges in different clauses of a rule. These rules are used with different names. These rules are treated as something specific to some situation each time. In this example, only the X variable occurs several times, but it is used as a different requirement.

14.9 Notation in Prolog for Building Blocks

Several notations are used to design the building blocks of Prolog, such as atoms and variable structures. These basic building blocks of Prolog make it easy to write programs in Prolog that users can easily understand.

14.9.1 Atoms

Atoms are the fundamental building blocks of Prolog. Atoms are represented as character strings syntactically but as integer values internally. That's why Prolog can be used in the unification (comparing) of atom values. Constants are used that characterize particular objects and particular relationships in Prolog. Two types of these constraints are used: numbers and atoms.

Numbers: These are defined by how numbers are represented in Prolog. They are demonstrated through the following examples of the representation of numbers: 2, 8, 0, 100.4e2, and -7.58.

Atoms: In Prolog, atoms are illustrated by

- character values or a collection of characters (string) plus the symbol, digit, underscore, and dollar sign.
- math symbols and graphics symbols, such as symbols like "?–" and ":–," are also atoms in Prolog.

For example, ram, monkey, ram$seeta, cats_and_dogs, ;; , +, and america are valid atoms in Prolog. But some notations are not valid atoms (such as -ram-and-seeta, ;; $ = , Ram, ram and seta) because – and $ are not graphic symbols, and a capital letter and space are not allowed in Prolog.

Atoms are also used as part of the query format.

atom(ram)	yes
atom(123)	no
atomic (45)	yes (*because it is an integer*)
atomic ("45")	yes (*because is an atom*)

14.9.2 Variables

The variables used in Prolog are referred to as logical variables. These logical variables are totally different from those variables that are used in conventional programs. Logical variables may be defined as wild cards for

pattern matching (unification). These variables are also used in combination with other Prolog terms that take other values.

To understand variables in Prolog, let's use an example: born(—, delisha).

In Prolog, variables are represented by a string and sequence of letters, digits, and an underscore. Only the special symbol underscore (_) is used in variables, which is known as an anonymous variable.

For example, let us consider a fact. In this fact, we want to know who likes Ram. A special symbol is used, which is known as the underscore variable:

```
?-like(_, mohan)
```

In Prolog, how can we determine a query ("who")? Two special types of variables are used for determining this. The first one is variable X and other one is the _ (underscore) variable. These two types of variables are differentiated by using two distinct queries.

First, a query is given in the form of variable X:

```
?-like(X,X)
```

The two occurrences of X denote the same person, hence, if the first X occurs with any name associated with Rakesh, then the second occurrence of X is automatically associated with Rakesh(any name). Thus, the above declaration in this case says "Rakesh like himself."

Second, a query is given in the form of the _ (underscore) variable

```
?-likes(-,-)
```

This query clarifies the search is for who likes someone. In this query, two underscores are used for different occurrences, i.e., the second occurrence (someone) is dissimilar from the first occurrence. The result will be provided by Prolog as "Rakesh likes Rajiv," where the first someone is Rakesh and the second someone is Rajiv.

14.9.3 Data Types and Structures

There are two elementary data types in Prolog: atoms and numbers, as noted earlier. We have also already discussed how atoms and numbers are represented in Prolog.

Now we will discuss the relations between these numbers, the operations on numbers, and the statements about numbers. All discussions are

represented with symbolic processing instead of numeric processing, based on the facts used in Prolog. This is the most important aim of Prolog, which is used to explain how symbolic data is structured using two elementary data types.

The data structures are of two types:

- Structures
- Lists

14.9.3.1 Structures

Structures are the primary data types of Prolog. A structure is defined by its name and is also known as a functor. Its arguments are where the functor is an atom and the arguments, which include other structures that may be any Prolog terms. Structures are written in the following form:

name (arg1, arg2, ... , argn)

There are some restrictions for writing structures, such as the fact that there is no space between the name and the opening parenthesis "(." In this example, several arguments are used. The arguments are used in a structure called an arity. An atom is actually a degenerate structure of arity 0. The range of the arity in a structure is 0 to 4095, which is the maximum arity.

Here are some examples of structures whose names include the likes and the number of arguments is two, which defines the arity.

```
likes(ram, cake)

likes(pari, pizza)

likes(Everyone, biscuits)
```

In Prolog, structures are used as the head and goal of the bodies of a Prolog clause. Let's examine an example:

```
brothers (X, Y) likes(X, Something), likes (Y, Some-
thing)
```

A fact is represented in Prolog by just putting a full stop at the end of a suitable structure. Similarly, a rule or a query can be obtained from a suitable structure. A list of terms is described in a structure in which a term itself may be a structure. A structure is used for simple as well as complex data,

in which the record structure of other programming structures are used. Several terms are defined in a structure, such as the following:

```
Employee (name, age, designation, address, gross-pay)
```

The name designation and age are also structures. The name term is used as a complex structure which may become further structures, such as

```
Name (first-name, middle-name, last-name)
```

Further, first-name may be described as

```
(first-name, tarun)
```

You can write any name with first-name. All of these structures are complex and contain information about an employee whose name is tarun, which may be written as a fact in Prolog.

```
Employee ( name ( first-name, tarun), (middle-name, kumar), Last-name, sharma)), (designation, assistant-registrar). (age, 28), (gross-pay, 45000), (address, Bombay).
```

Similarly, the address may be represented as a complex structure, which is further structured as

```
address(h.no 3638), (street no 8) (area chandni-chowk) (city delhi)).
```

In the structure, a query is defined to find the gross pay of employee whose name is tarun in the form as:

```
?_employee(name(tarun),_,_),_,_) (gross-pay X)).
```

This will return 45000. In the above query, the underscores are used to represent various variables.

Example 1: Give the information about the book *Artificial Intelligence,* Third Edition, by Elaine Rich Knight by McGraw Hill Education in the year 2009 as a structure in Prolog so we can know the name of the author, assuming the author's name is not given.

Solution:

```
book (title artificial_intelligence), (edition third), (author ( first_name elaine), (middle_initial rich), (last_name knight)) (year 2009), (publisher mcgraw_hill_education)).
```

For the following query, write:

```
?_book ((title artificial_intelligence), (edition third), X, (year 2009), (publisher mcgraw_hill_education)).
```

The system returns the name of the author as `(author ((first_name elaine), (middle_initial rich), (last_name knight)).`

Example 2: Consider a simple sentence: "Monika eats an apple."

This may be represented in Prolog as

```
Sentence((noun monika), (verb_phrase (verb eats), (noun apple))).
```

In general, the structure of a sentence of the form given above may be expressed in Prolog as

```
Sentence((noun N1), (verbphrase ( verb v), (noun N2)).
```

It is represented in the form of a query as

```
?_sentence (( nounmonika), (verbphrase (verb eats), (noun X))
```

`Which` tells us about what Monika eats.

Example 3

```
Is_small(vishal) ^ is_richer( shyam, bhim)
```

Now this statement has to be expressed as two different Prolog statements, defined as `Is_small(vishal).`

```
Is_richer(shyam,bhim).
```

Note that at the end of every statement, there is a period (.).

14.9.3.2 Lists

The other data types in Prolog are lists. The list is a common data structure that is built into almost all programming languages that represents nonnumeric processing. A list is an ordered sequence of elements, and it can have any length. The difference between an array and a list is that an array has a fixed length, but a list has any length. In a list, terms are used to define the elements (such as constants and variables). Thus, a list is a recursive concept. Lists are represented by square brackets [].

A list in Prolog may be defined recursively as follows:

- [], representing an empty list, is a list.
- [e1, e2—en] is a list where e1, e2 and the others are several terms

Examples of Lists

- [] – empty list
- [1, 2, 3] – single element in a list
- [1,[2, 3]]– a list with two elements
- [3, X,[a, [b, X + Y]]] – is a list, provided that X + Y is defined. For example, X and Y are numbers, and then, as we shall see later, X + Y is a valid expression (it is a sequence of terms and operators formed according to the syntactic rules of the language, X-Y/Z).
- [_,_,Y] – This represents a list of three elements, out of which first two are "don't care" or anonymous variables.
- [f(1), [2, a], X]– This is a list that contains three elements in which the first is a structure (arity 1), the second is a sublist containing two elements, and the last element is a variable, X.

Logically, a list can be measured to have two elements:

- HEAD – the first element of the list
- TAIL – a list of the remaining elements in the list

At each level of recursion, the HEAD can be used, and the TAIL passed down to the next level of recursion. It is vital to keep in mind that the TAIL is always used as another list.

To represent a pattern in Prolog, a vertical bar is used: [HEAD | TAIL]. To unify this pattern, we can use a list. Now, let's see how it works:

```
?- [HEAD|TAIL] = [a, b, c, d].

HEAD = a

TAIL = [b, c, d]
```

In this, only one element is specified in the head, but more than one element is also specified, such as

```
?- [FIRST, SECOND|TAIL] = [a, b, c, d].
```

```
FIRST = a

SECOND = b

TAIL = [c, d]
```

This represents a list in which there is no tail.

```
?- [W, X, Y, Z|TAIL] = [a,b,c,d].

      W = a

      X = b

      Y = c

      Z = d

TAIL = []
```

[] is then a useful element for recognizing that a recursive list predicate has reached the end of the list. Now we want to represent each element of a list on a new line. To represent a new line, the "nl" term is used. For example,

```
write_list([]). empty list, end.

write_list([A|Z]) :-

write(A), write the head nl,

write_list(Z). recurse with the tail
```

Using it:

```
?- write_list([mango,banana,litchi,grapes]).
```

mango

banana

litchi

grapes

yes

While lists are stored more efficiently than structures, lists are also used as a nested structure of arity two or more. If a list has a structure of arity two in which first argument is the head, the second argument is the same structure representing the rest of the list. In this, a special atom is used, represented by [], which indicates the end of the nesting. This character (nature) of lists

can be observed using the display/1 predicate, where a period (.) is used as the functor of the structure.

```
?- display( [a, b, c] ).

.(a, .(b, .(c, [])))
```

Character Lists

Character lists are those lists whose elements are character codes. Character lists are mainly used in parsing applications. Prolog identifies a particular syntax to make this use more expedient.

A string of characters enclosed in double quotes (“) is converted into a list of character codes. Let us take an example (using member/2 from the list library) that illustrates how to use the character list and shows the predicates for converting between the character lists and atoms and strings.

For example:

```
        ?- X = “abc”.

      X = [0w0061, 0w0062, 0w0063]

      yes

?- member(0’b, “abc”).

yes

?- atom_codes(abc, X).

      X = [0w0061, 0w0062, 0w0063]

yes

?- atom_codes(A, “abc”). A = abc

yes

?- string_list(S, “abc”). S= abc

yes

?- string_list(`abc`, L).

      L = [0w0061, 0w0062, 0w0063]

yes
```

14.10 Arithmetic Operations

Prolog is not used for only symbolic representations of AI applications, but also used for numeric representations. In Prolog, there are several arithmetic predicates that take a number of arguments on which many operations are performed as in the usual mathematical expressions, such as "+" (for addition), "-" (for subtraction), "*" (for multiplication), and "/" (for division). Frequently, before executing an arithmetical predicate, every variable that is used in the expressions on its left-hand side and right-hand side has to be instantiated to terms only containing numbers and operators. The arguments will be evaluated before the test specified by the predicate is performed.

In Prolog, the following arithmetical relational predicates are most often used:

X > Y.

X < Y.

X > = Y.

X = < Y. X = : = Y. (equality)

X = \ = Y. (inequality)

It may be noted that in Prolog, the expression "3 + 7" will not give the result of addition (that is, "10"). The operation of "+" does not execute automatically. To execute an arithmetic operation, Prolog provides a special type of operator, "is": `? -X is 3 + 7`. This will give a result of 10, which is the same as the result of the arithmetic operation.

```
?- X is Y.
```

Only the right side Y has to be instantiated to an arithmetical expression. Note that the difference between the predicate "=:=" and the matching predicate "=;" In case of "=:=," both X and Y have to be instantiated to arithmetical expressions. In the case of a matching predicate, neither X nor Y has to be instantiated.

Example: Consider the following queries and answers that illustrate the differences and similarities between the predicates =, =:=, and is:

(i)

```
? - 8 = 5 + 3
```

no

(ii)
```
? - 8 is 5 + 3.
```
yes

(iii)
```
? - 8 = : = 5 + 3.
```
yes

(iv)
```
? - 3 + 4 = 3 + 4.
```
yes

(v)
```
? - 3 + 4 = : = 3 + 4.
```
yes

(vi)
```
? - 3 + 4 is 3 + 4.
```
no

(vii)
```
? - 4 + 3 = 3 + 4.
```
no

(viii)
```
? - 4 + 3 = : = 3 + 4.
```
yes

The following examples describe the behavior of these predicates when the left side is an uninstantiated variable. Prolog shows the computed instantiation:

```
? - X is 5 + 3.
X = 8
? - X = 5 + 3.
X = 5 + 3
```

We have left out the example of `? - X = : = 5 + 3`, since it is not permitted to have a uninstantiated variable as an argument to = : =. 8 The predicates are = : = . These may only be useful to arithmetical arguments. The predicate =, however, also applies to non-arithmetical arguments,

For example, `? - X = [a,b].` directs to the instantiation of the variable X to the list [a, b]. when we use is predicate and = : =predicate in Prolog language then Prolog interpreter will return a signaled error.

To understand this, let us explain a Prolog program for a factorial:

```
Factorial (0,1)
```

```
Factorial (Number, Result):-Number > 0, new is number
-1, factorial (new, partial), Result is Number * partial.
```

Here is an explanation of the Prolog program for a factorial. If the given number is 0, then its factorial is 1. Additionally, the result of the computation for a factorial of any number, Number, will be associated with the variable, Result, and the goal can be achieved through the following four subgoals:

- Number > 0 should be true.
- In the second subgoal, the operator "is" is used to calculate the number "Number –1" and is associated with the variable "New."
- The factorial of the number "Number -10" through the operator "is," i.e., of the number "New," is recursively calculated and the result of the calculation is associated with the variable partial.
- Finally, the operator "is" is used as the product of the number with a partial, which is the result of the factorial of "New" (i.e., to calculate the number –1; it is associated with "Result").

14.11 Strings

A string is an exchange way to represent a text. To represent an integer (terms), a string is also used. A string represents an integer in the form of a table, and a string can also be represented as a string itself. Strings are useful for textual information that is for display purposes only. To combine strings, the system uses a character-by-character basis and with atoms. A string is signified by the text enclosed in matching backquotes (`). Strings may also have fixed (embedded) formatting characters, like atoms. For example:

```
`This is a long string used for\ndisplay purposes`
```

Two backquotes are used for representing this string.

Strings are mainly used to represent text that is being used for I/O, and they are not used for unification. To understand this, let us look at an example of "customers," which is represented as a clause that might have the customer name as an atom. Now, the string is used for fast unification, such as the customer's address information as a string just for output purposes.

Internally, all strings are stored as Unicode (wide) character strings. In Prolog source code, when reading and displaying any information, this full wide (Unicode) character set can be used.

Strings do not occupy space in the atom table, and the space they occupy is automatically collected and reused by the system once the string is no longer needed. As a result, strings can be more memory-efficient for large quantities of textual information.

Exercises

Q1. What is Prolog? How it is different from other languages?

Q2. How do we differentiate between a Prolog and a simple English sentence?

Q3. Define facts and rules.

Q4. How many types of rules are there? Explain them.

Q5. What are the notations for building blocks in Prolog?

Q6. How does Prolog satisfy goals?

Q7. Write a program in Prolog using structures.

Q8. Write a program in Prolog using lists.

Q9. Define a data structure for Prolog.

Q10. Write a program in Prolog using arithmetic operations.

CHAPTER 15

ADVANCED PROLOG

15.1 Input and Output Predicates

In Prolog, there are mainly two types of input and output predicates. The input and output predicates are the most important predicates, in which a user reads from files and writes to files. The basic input and output predicates are shown in Table 15.1.

Table 15.1 I/O Predicates

Predicate	Explanation
write(X)	Write the term X on the current output stream.
nl	Start a new line on the current output stream.
read(X)	Read a term (finished by a full stop) from the current input stream and unify it with X.
put(N)	Write the ASCII character code N. N can be a string of length one.
get(N)	Read the next character code and unify its ASCII code with N.
see(File)	File becomes the current input stream.
seeing(File)	The current input stream is File.
Seen	Close the current input stream.
tell(File)	File becomes the current output stream.
telling(File)	The current output stream is File.
Told	Close the current output stream.

15.1.1 Terms and Character I/O

We can read input and write output in two ways: terms and characters. A term for input and output predicates is defined as being delimited by a full stop. It is used in the ordinary Prolog sense. ASCII characters are used for defining a character. The predicates used are demonstrated in the table below.

Input	Output
Term	read/1write/1
Character	get0/1put/1 get/1

What happens when we write predicates like this? Let's use an example from a Prolog terminal to help us understand how these predicates work.

```
| ? – write(hello).
hello
yes
| ? – write('hello world').
hello world
yes
| ? – X = hello, write(X).
hello
X = hello ?
yes
```

This example shows how different Prolog is from other languages! In every programming language, the first program you write is the "Hello World" program. With read/1, Prolog goes somewhere else (by default to the terminal, in which case the |: prompt appears (also by default) to tell the user where to type the term) to find something that ends with a full stop. Anything that it finds is unified with the variable you give it.

```
| ?- read(X).
|: 'hello ana'.
'hello ana'.
X = 'hello ana' ?
yes
```

If we want to write a term with input as spaces from the terminal, then we must use quotes to write it. You can use the value of what Prolog reads in (*X* in the example above) in a while goal, such as

```
| ?- read(X),name(X,L).
|: hello.
hello.
L = [104,101,108,108,111], X =
hello ?
yes
```

Put and get0 are the same as write and read, but they use a single ASCII character for writing or reading. The ASCII character is used so its argument becomes a list of single ASCII numbers. get0 and get are used, in which get0 reads everything and get binds its argument to the next printed character, spaces are removed, and new lines and control characters are found in the input stream.

Another procedure useful for producing output is nl/0. It creates a new line and always accomplishes this exactly once.

15.1.2 File I/O

By default, input is taken from the keyboard and output is sent to the screen. These include a virtual file, which is known as user (you can read more about this in the manual). In spite of these files, we can also handle other files using predefined commands to

- ask which file you are currently taking input from/sending output to
- tell Prolog to take input from/send output to a specific file
- tell Prolog to stop taking input/sending output to the file it is currently dealing with

These predicates are shown in the table below.

Input File	Output File
Determine current	seeing/1telling/1
Change to new	see/1tell/1
Close current	seen/0told/0

If you want to save the output of a program to a file, you can use tell/1 and told/0. Say that you want to analyze the data generated by the file pari.pl, and save the output to pari.dat. Then you can use the following code (the command which generates the data is called run):

```
| ?- consult([pari]).
{consulting /homedir/pari/Prolog/pari.pl...}
{/homedir/pari/Prolog/pari.pl consulted, 150 msec 13776 bytes}
yes
| ?- tell('pari.txt'), run, told.
Yes`
```

15.2 Backtracking

Backtracking is defined as the process to re-satisfy a goal. It is used by Prolog. It works when a goal cannot be satisfied; then, Prolog makes an effort to discover an alternative clause for that goal. When a subgoal or goal fails, it makes an effort to re-satisfy the goal. Variables, which are used in facts or rules in Prolog, are also used in backtracking. These variables are used to find their original values. In backtracking, when a previous search has been stopped, then a new solution can be found for the goal by beginning another search.

We can say that backtracking refers to going back over the steps you followed. In the search tree, you go back up the branches you followed, back up to the previously untraversed branches, so no untraversed branches are left. All branches must be visited once.

To understand the concept of backtracking, let's look at an example:

```
f(x) :- fail ? f(1)
f(1) ∧
fail Yes
```

Steps followed:

- Go from f(1) to fail.
- Go back from fail to f(1). This is the backtracking step.
- Go from f(1) to yes.

Backtracking is an influential tool in searching option directions when one of the directions being followed for finding a solution leads to a failure. This is of great importance in Prolog. When a query is entered and many solutions are provided at one time, backtracking is also useful. First, we demonstrate the concept through an example and then explain the general idea of backtracking.

Example

Consider the following query for the database:

?-is_sister (aditi), is_sister(Y, riddhi)

First, the query is translated into a simple (English) sentence and it becomes the following: find the names (denoted by Y) of all those persons for whom Aditi is a sister and who (denoted by Y) is a sister of Riddhi.

In order to answer the above query, the, first solution the Prolog system comes up with after searching the database is as follows: "Associate Aditi with Y," i.e., Aditi is one possible answer to this query (which is the conjunct of two proposition). In other words, associate Y with Aditi, which is, according to the facts and rules given in the database, satisfies is _ sister (Aditi, Aditi) and is_sister (Aditi, Riddhi). If we are interested in more than one answer, which is possible in this case. After obtaining the answer "Aditi," the user should type the symbol ";" (i.e., type a semi-colon). Typing a semi-colon followed by a return serves as a direction to the Prolog system to search the database from the beginning once again for an alternative solution.

In order to prevent the Prolog system from attempting to find the same answer (Aditi) again, the system puts marker-one on the rule and after that on the fact.

Once the instruction from the user through the semi-colon is received to find another solution, the system proceeds to satisfy the rules from the facts. Then, through the facts for the first occurrence of Y, Aman is associated. The variable X is already associated with constant Aditi. Then Aman replaces Y in the rule. Next, the Prolog system attempts to satisfy the second subgoal which, at present, is of the form

is _ sister (aman x z, riddhi)

To satisfy this goal, the Prolog system searches the database from the top again. Again, a rule is used. To satisfy the left-hand side of the subgoal, the right-hand side needs to be satisfied. The first subgoal on the right-hand side is to satisfy the fact (female(aman)), which is not satisfied.

At this stage, the Prolog system goes back to the association, i.e., Aman to Y and removes this association of Y with Aman. Next, the Prolog system attempts to associate some other value to Y, further from the point where Y was associated with Aman. This is what is meant by backtracking.

Next, through a fact, Riddhi is associated with Y and Aditi is already associated with X, thus, the subgoal to be searched becomes is _sister (riddhi, riddhi). This goal can be satisfied because the three subgoals for this goal, viz., female (riddhi), parents (riddhi, M, F) (where Riddhi is associated with X), and parents (riddhi, M, F) (where Riddhi is associated with Y), can be easily seen to be satisfiable from the database. Hence, the second answer the system gives is "Riddhi."

Again, the user may seek for another answer to the query by typing ";." The system has already marked the fact in the database while finding the answer "Riddhi." Therefore, for another answer, the Prolog system starts from the next statement, i.e., from fact to search. It can be easily seen that the Prolog system will not find any more answers. Hence, it returns "fail" or "no."

Another use of backtracking is to compute the permutation of a given list. For this, we want to write a predicate that is built in. The implementation of a permutation uses a built-in predicate, which takes a list as its second argument and matches the first argument with an element from that list. In the third argument, the variable position will then be matched with the rest of the list after having removed the chosen element.

Now, to find a possible solution of the permutation, there is one possible way: an empty list. An empty list is one possible permutation. In this, if the input list has obtained elements, then the goal or subgoal that has been selected will be successful. After this, it combines the variable element to an element that is used in an input list. After the combination, this element becomes the head of the output list and recursively calls the permutation again with the rest of the elements (those elements that are not used in the output list) of the input list. Now many answers are generated, but the first answer of the query will be used to reproduce the input list because the rest of the elements will be assigned to the value of the head of the output list. If further substitutions are requested, then backtracking is used, where the selected goal or subgoal takes place. Backtracking is used because it will

generate all possible orders of selecting the elements from the input list, in other words, it will generate all permutations of the input list. How can the input list generate the output list using backtracking in a permutation? Let's look at an example:

? - permutation([1, 2, 3], X).

X = [1, 2, 3] ;

X = [1, 3, 2] ;

X = [2, 1, 3] ;

X = [2, 3, 1] ;

X = [3, 1, 2] ;

X = [3,2,1];

No

15.2.1 Problems with Backtracking

Backtracking is one of the most qualitative features of Prolog. Some of the problems in backtracking can lead to inefficiency. For example, Prolog can waste time discovering possibilities that lead nowhere, and we can say that controlling backtracking is very difficult. This can be explained by an example.

Example

member(X, [X|_]).

member(X, [_|T]) :- member(X, T).

?- member(fred, [chirag, fred, kushal, fred]).

yes

?- member(X, [chirag, fred, kushal, fred]).

X = chirag;

X = fred;

X = kushal;

X = fred;

no

The Problem of Controlling Backtracking

color (cherry, red).

color (banana, yellow).

color (apple, red).

color (apple, green).

color (orange, orange).

color (X, unknown).

?- color (banana, X).

X = yellow

?- color(physalis, X).

X = unknown

?- color(cherry, X).

X = red;

X = unknown;

no

Now, there are some solutions that can control this problem. It would be pleasant to have some control over this quality of its behavior. Two possible ways to control the problem are shifting the order of rules and changing the order of the conjuncts in the body of rules.

Another possible solution of this problem is the built-in predicate known as "cut." This is the best solution for controlling the problem of backtracking, and is better than the two ways mentioned above.

15.3 Cut

"Cut" is basically a special atom that we can use when writing clauses. It used for logical properties as well as used for its effects. It always succeeds. It is represented by the exclamation mark (!). Let's look at an example.

p(*X*) : b(*X*), c(*X*),!, d(*X*), e(*X*).

This example explains the absolute rule of Prolog that defines cut. What does cut refer to? It is a goal, or we can say that it is a parent goal that always succeeds. Let us assume that this goal makes use of the clause. Cut is used

to assign any choices to Prolog that were made, since the parent goal was unified with the left-hand side of the rule.

If p(X) matches, goals b(X) and c(X) may backtrack among themselves. If it is p(X), another goal will be attempted. But as soon as the cut is crossed, Prolog commits to the current choice. All other choices are discarded.

To understand how cut works, let's look at an example that defines a (cut-free) predicate max/3, which takes integers as arguments and succeeds if the third argument is the maximum of the first two. For example, the queries

max (2, 3, 3)

and

max (3, 2, 3)

and

max (3, 3, 3)

should succeed, and the queries

max (2, 3, 2)

and

max(2, 3, 5)

should fail. Of course, we also want the program to work when the third argument is a variable. That is, we want the program to be able to find the maximum of the first two arguments for us:

? - max (2,3, Max).

Max = 3

Yes

?- max (2, 1, Max).

Max = 2

Yes

Now, it is easy to write a program that does this. Here's a first attempt:

max (X, Y, Y) :- X = < Y.

max (X, Y, X) :- X > Y.

This program works well, but it's not good enough. What's the problem? There is a possible inefficiency. Suppose this definition is used as part of a larger program, and somewhere along the way, max (3, 4, Y) is called. The program is correct when Y = 4. But now consider what happens if, at some stage, backtracking is forced. The program will try to re-satisfy max (3, 4, Y) using the second clause. And of course, this is completely pointless: the maximum of 3 and 4 is 4, and that's that. There is no second solution to find. To put it another way, the two clauses in the above program are mutually exclusive. If the first succeeds, the second must fail and vice versa. Attempting to re-satisfy this clause is a complete waste of time.

With the help of cut, this is easy to fix. We need to insist that Prolog should never try both clauses, and the following code does this:

```
max (X, Y, Y) :- X = < Y,!.

max (X, Y, X) :- X >Y.
```

Note how this works. Prolog will reach the cut if max (X, Y, Y) is called and X = < Y succeeds. In this case, the second argument is the maximum, and that's that, and the cut commits us to this choice. On the other hand, if X = < Y fails, then Prolog goes onto the second clause instead.

Note that this cut does not change the meaning of the program. Our new code gives exactly the same answers as the old one, it's just a bit more efficient. In fact, the program is exactly the same as the previous version, except for the cut, and this is a pretty good sign that the cut is a sensible one. Cuts like this, which don't change the meaning of a program, have a special name: they're called *green cuts*.

But there is another kind of cut: cuts that do change the meaning of a program. These are called *red cuts*, and these are usually best avoided. Here's an example of a red cut. Yet another way to write the max predicate is as follows:

```
max (X, Y, Y) :- X = < Y,!.

max (X, Y, X).
```

This is the same as our earlier green cut max, except that we got rid of the > test in the second clause. This is bad sign: it suggests that we're changing the underlying logic of the program. And indeed we are: this program works by relying on cut. How good is it?

Well, for some kinds of queries, it's fine. In particular, it answers correctly when we pose queries in which the third argument is a variable. For example:

? - max (100,101,X).

X = 101

Yes

and

? - max (3,2,X).

X = 3

Yes

Nonetheless, it's not the same as the green cut program: the meaning of max has changed. Consider what happens when all three arguments are instantiated. For example, consider the query max (2, 3, 2).

Obviously, this query should fail. But in the red cut version, it will succeed! Why? Well, this query simply won't match the head of the first clause, so Prolog goes straight to the second clause. The query will match with the second clause, and (trivially) the query succeeds! Oops! Getting rid of that > test wasn't quite so smart after all...

This program is a classic red cut. It does not truly define the max predicate, rather, it changes its meaning and only gets things right for certain types of queries.

15.4 Fail

When a rule is used in Prolog but fails due to some inefficiency, the fail predicate is used for forced backtracking. Its subgoal is found but all of the subgoals defined after fail will never be executed. Hence, a predicate fail should always be used as the last subgoal in a rule. It is to be noted that a rule containing a fail predicate will not produce any solution.

Consider an example to demonstrate the use of a fail predicate.

listing(Name, Address) :- emp(Name, Address).

emp (divya, maths).

emp (monika, cse).

emp (seeta, mechanical).

Goal: ?- listing(Name, Address).

When above goals are executed by backtracking, then all possible solutions are attained. All of these solutions are described as follows:

Name= divya, Address = maths; Name = monika , Address = cse;

Name= seeta, Address = mechanical;

listing:- write('Name'), write(' Address'), nl,

emp (Name, Address), write (Name), write (") write ('Address') nl, fail.

emp (divya, maths).

emp(monika, cse).

emp(seeta, mechanical).

Goal: ?- listing

Name Address

divya maths

monika cse

seeta mechanical

15.4.1 Cut and Fail Combination

The cut and fail combination is useful for expressing negative facts. Consider the example "Vishal does not like lions," which could be expressed by the following rule and fact.

like(vishal, lions):- !, fail.

like(vishal, X).

This rule and fact define that "Vishal likes everything except lions," which means that "Vishal does not like lions".

Goal: ?- like(vishal, lion). Answer: no

Goal: ?- like(vishal, snake). Answer: yes

15.5 Recursion

Recursion is used where many problems (class of problems) are generated and solving them in some way involves using parameters. Usually, parameters are used to determine the complexity of a problem that are numbers

or items in a Prolog list (rather than length). Now, we must make sure that every recursion step will really transform the problem into the next simpler case and that the base case will ultimately be reached.

Let's think about an example of how recursion is used in the factorial of a number.

The first argument is striving towards 1; in the len/2-example, the first argument is striving towards the empty list. The recursion principle itself is very simple and is applicable to many problems. Despite the simplicity of the principle, the actual execution tree of a recursive program might become rather complicated. This example recursion is performed in Java and Prolog. To find the factorial of a number in Java by recursion, we write the following code:

```
public int factorial(int n) {
if (n == 1) {
return 1; // base case
} else {
return factorial(n-1) * n; // recursion step
}
}
```

Now the recursion is used in Prolog to find the factorial of a number "Recursion" in Prolog. We include a definition of a Prolog predicate to compute factorials:

```
factorial(1, 1). % base case
factorial(N, Result) :- % recursion step
N > 1,
N1 is N - 1, factorial(N1, Result1), Result is Result1 * N.
```

Recursion 1

As is usually the case in many programming tasks, we frequently wish to repeatedly perform some operation either over a whole data structure or until a certain point is reached. This is done by recursion in Prolog because it provides the best way to perform this action. Recursion can be defined as a program that calls itself, usually until some final point is reached. But in

Prolog, it is defined in terms of the facts or rules. Suppose that we have a first fact that acts as some stopping condition, followed up by some rule(s) that performs some operation before re-invoking itself.

Recursion 2

```
on_route (miami).

on_route (Place):- move (Place,Method,New Place),

on_route (New Place).

move(home,car,railway station).

move(railway station, train, airport).

move(airport, plane, miami).
```

on_route is a recursive predicate. This program sees if it is possible to travel to Miami from a particular place. The first clause sees if we have already reached Miami, in which case, we stop. The second clause sees if there is a move from the current place to somewhere new, and then the recursive sees if the New Place is on_route to Miami. The database of moves that we can make is on the right.

Let's now consider what happens when we pose the query ?- on_ route(home). This matches clause two of on_route (it can't match clause one because home and Miami don't unify). The second on_route clause consists of two subgoals. The first asks whether you can move from home to some new location, i.e., move(home,Method,New Place). This succeeds with Method = taxi, New Place = railway station. This says that yes, we can move from home by taking a taxi to the railway station. Next, we recursively see if we can find a route from railway station to Miami by doing the same thing again. This is done by executing the new subgoal on_route(railway station).

Recursion 3

```
on_route (miami).

on_route (Place):- move (Place,Method, New Place),

on_route (New Place).

move(home,taxi,railway station).

move(railway station,train,airport).

move(airport,plane,miami).
```

The goal on_route (railway station) will fail to unify on clause one, so again we'll use the recursive clause two and find some new place to go to. Hence, we try the goal move(railway station,Method,New Place). This succeeds because we can catch a train from a railway station to the airport. Hence, Method = train, and New Place = airport. As a result, we then try the recursive call on_route(airport), i.e., we see if there is a move from the airport which will get us to Miami.

We now try on_route(airport), and again this only unifies with the second clause. As a result, we try the move clause again, this time with the "Place" bound to "airport." This query will match the third clause of the move database. The results in Method = plane, New Place = miami. Next, we try the recursive goal on_route(miami). This now matches clause one of on_route. This is just a fact and succeeds. As a result, all the other on_route goals in turn succeed. Thus, finally our first goal ?- on_route(home) succeeds and Prolog responds "yes."

15.6 Prolog Data Structure

A data structure is defined as the process of organizing the facts or rules. But in Prolog, it is of three types: terms, unification, and operators.

[terms] [unification] [operators]

15.6.1 Terms

A term is a basic data structure in Prolog. A term is defined as everything that is used in a program and the data. Everything can be represented in the form of terms. There are four basic types of terms in Prolog: variables, compound terms, atoms, and numbers.

term

|– **var** (X, Y)

| – **nonvar** (a, 1, f(a), f(X))

|– **compound** (f(a), f(X))

| – **atomic** (a,1)

|– **atom** (a)

| – **number** (1)

15.6.2 Unification

Another type of data structure is unification. Unification is an engine of Prolog. One powerful feature of Prolog is pattern matching, in which unification is used. It tries to find most general substitution of variables in two terms such that after applying this substitution to both terms, the terms become the same. If we want to unify two terms, A and B, one can easily invoke the built-in unification A = B. This is defined as a recursive definition to handle data structures. To understand the concept of unification, we will introduce the built-in predicate =, which means when two arguments unify, then it will be successful (yes), otherwise it will fail when the two arguments do not unify. It can be written in operator syntax as follows.

arg1 = arg2

which is equivalent to

= (arg1, arg2)

Note that the equal sign (=) does not define as assignment operator as in most programming languages. It causes Prolog unification.

The unification between two sides of an equal sign (=) is exactly the same as the unification that occurs when Prolog tries to match goals with the heads of clauses. When backtracking, the variable bindings are undone, just as they are when Prolog backtracks through clauses.

Unification occurs in its simplest form when no variable is used. It occurs only between two structures. It works as if these structures are identical; then, it will respond "true" (succeeds), otherwise, it return "fail," meaning that the unification failed.

```
?- a = a. yes
?- a = b. no
?- location (knife, kitchen)=location(knife kitchen).
                    yes
?- location(knife, kitchen)=location(knife, room).
                    no
?- a(b,c(d,e(f,g))) = a(b,c(d,e(f,g))).
                    yes
?- a(b,c(d,e(f,g))) = a(b,c(d,e(g,f))).
                    no
```

Unification occurs in another form also, between a variable and a structure (primitive). Unification succeeds when the variable takes on a value.

```
?- X = a.

X = a

?- 4 = Y.

Y = 4

?- location(knife, kitchen)=location(knife, X). X = kitchen
```

In other cases, multiple variables are simultaneously bound to values.

```
?- location (X, Y) = location (apple, kitchen).

X = apple

Y = kitchen

?- location(apple, X) = location (Y, kitchen).

X = kitchen

Y = apple
```

Prolog remembers the fact that the variables are bound together and will reflect this if either is later bound.

```
? - X = Y, Y = hello.

X = hello

Y = hello

? - X = Y, a (Z) = a(Y), X = hello.

X = hello

Y = hello

Z = hello
```

The last example is critical to a good understanding of Prolog and illustrates a major difference between unification with Prolog variables and the assignment with variables found in most other languages. Note carefully the behavior of the following queries.

```
?- X = Y, Y = 3, write(X).

        3
```

X = 3

Y = 3

?- X = Y, tastes _ yucky(X), write(Y).

broccoli

X = broccoli

Y = broccoli

Even in these more complex examples, the relationships between the variables are remembered and updated as new variable bindings occur.

? - a (b,X) = a (b,c (Y,e)), Y = hello.

X = c (hello, e)

Y = hello

?- food (X,Y)= Z, write(Z), nl, tastes _ yucky(X), edible(Y), write(Z).

food(_01,_02)

food(broccoli, apple)

X = broccoli

Y = apple

Z = food(broccoli, apple)

If a new value assigned to a variable in later goals conflicts with the pattern set earlier, the goal fails.

?- a(b,X) = a(b,c(Y,e)), X = hello.

no

The second goal failed, since there is no value of Y that will allow "hello" to unify with c(Y,e). The following will succeed.

?- a(b,X) = a(b,c(Y,e)), X = c(hello, e).

X = c(hello, e)

Y = hello

If there is no possible value the variable can take on, the unification fails.

?- a(X) = a(b,c).

no

```
?- a(b,c,d)        =        a(X,X,d).
                   no
```

The last example failed because the pattern asks that the first two arguments be the same, and they aren't.

```
?- a(c,X,X)        =        a(Y,Y,b). no
```

Did you understand why this example fails? Matching the first argument binds Y to c. The second argument causes X and Y to have the same value, in this case, c. The third argument asks that X bind to b, but it is already bound to c. No value of X and Y will allow these two structures to unify.

The anonymous variable (_) is a wild variable, and it does not bind to values. Multiple occurrences of it do not imply equal values.

```
?- a(c,X,X) = a(_,_,b).
X = b
```

Unification occurs explicitly when the equal (=) built-in predicate is used, and implicitly when Prolog searches for the head of a clause that matches a goal pattern.

15.7 Dynamic Database

Databases are of two types: the static database and dynamic database. If programmer wants to modify the content of the database at runtime, then a Prolog system provides a dynamic database.

The main feature of this database is the addition to clauses of two types of predicates, asserta and assertz. Both predicates take one argument. If this argument has been instantiated to a term before the procedure call is executed, asserta adds its argument as a clause to the database before all (possibly) present clauses that specify the same functor in their conclusions. On the other hand, assertz adds its argument as a clause to the database just after all other clauses concerning the functor.

Example Consider the Prolog database containing the following clauses.

```
fact(a).

fact(b).

yet _ another _ fact(c).
```

and _ another _ fact(d).

We enter the following query to the system:

?- asserta (yet _ another _ fact(e)).

After execution of the query the database will have been modified as follows:

fact(a).

fact(b).

yet _ another _ fact(e).

yet _ another _ fact(c).

and _ another _ fact(d).

The execution of the procedure call is as follows:

?- assertz(fact(f)).

which modifies the contents of the database as follows:

fact(a).

fact(b).

fact(f).

yet _ another _ fact(e).

yet _ another _ fact(c).

and _ another _ fact(d).

By means of the one-placed predicate take back, the first clause having both the conclusion and conditions matching with the argument is removed from the database.

15.8 Programs in Prolog

- Example 1

If John feels hungry, then he eats quickly. If he eats quickly, he gets heartburn. If he gets heartburn, he takes medicine. John feels hungry.

Conclusion: Given the above facts, what can we conclude about John taking medicine? Does he or doesn't he?

The above argument is a sorites argument, or it can be interpreted as a series of modus ponens arguments. This one is easy! Of course, John takes medicine. Now, can you get Prolog to tell you this?

Answer : Give Prolog the following facts:

```
eats(john,quickly) :- feels (john,hungry).
gets(john,heartburn) :- eats (john,quickly).
takes(john,medicine) :- gets(john,heartburn).
feels(john,hungry).
```

Ask Prolog "takes(john,medicine)." The answer should be "yes."

- Example 2

John is dating Nancy, but right now he is wondering if he must leave the country. Now John will take a vacation in either of two circumstances: if the IRS is after him or if he is doomed. He is also dating Susan. If he is dating both of them, then Nancy knows this. Now John is, in fact, doomed if both Nancy and Susan know he is dating both of them. If John will take a vacation, then he must leave the country. So, must John leave the country?

Conclusion:

What do you think? This one is a little more difficult.

Answer: Give Prolog the following facts:

```
dates(john, nancy).
takes(john,vacation) :- (after(irs,john) ; is(john,doomed)).
dates(john, susan).
knows(nancy, (dates(john,nancy) , dates(john,susan))):- (dates(john,nancy),dates(john,susan)).
is(john,doomed):-
(knows(nancy, (dates(john, nancy) , dates(john, susan))) , knows(susan, (dates(john, nancy) , dates(john,susan)))). leaves(john, country) :- takes(john, country).
```

Ask Prolog "leaves(john, country)." The answer should be "no" because the computer hasn't been told that Susan knows that John is dating both

Nancy and Susan, and so it interprets this statement as "fail" (unproved) and so the inference doesn't go through.

Can you get Prolog to compute these answers? 4 + 5 * 10

(4 + 5) * 10

Answers: Type

X is 4 + 5 * 10.

Prolog responds: X is 54

Type

X is (4 + 5) * 10.

Prolog responds:

X 90

15.9 Problems with Prolog

- To be purely declarative, a programmer should not be required to affect the control flow for program success.
  - ♦ It should be a consequent of the DFS search strategy.
  - ♦ A left recursion often leads to incorrect results.
- Prolog uses a closed world assumption (and nonmonotonic reasoning); Prolog is a true-fail system, not a true-false system. There is no mechanism in Prolog by which to assert facts that are assumed to be "fails." Prolog relies on pure positive logic. This is another reason why there is no logical "not." As a result, not(P) can succeed simply because Prolog cannot prove P is true.
- Horn clauses are not expressive enough to capture all knowledge in first-order predicate calculus (e.g., the propositions in clausal form with a disjunction of more than one non-negated term, such as every "positive natural number is either even or odd," even(N) ∨ odd(N) Ì natural(N), negation as failure (a reflection of Prolog's limitation to Horn clauses) odd(N) :- natural(N), not(even(N)). even(N) :- natural(N), not(odd(N)). More negation issues are as follows: not(transmission(X,manual)) means ¬ $X (transmission (X,manual)) or "There are no cars with manual transmissions" rather than

- $X (¬transmission(X,manual)) or "Not all cars have a manual transmission" (example inspired by [PLP] p. 582). As a result, the goalnot(transmission(X,manual)) fails even if we have transmission (accord,manual) in our database. This causes the occurs-check problem.

Exercises

Q1. What are input output predicates in Prolog?

Q2. Differentiate between file input output and character input output.

Q3. Write a program in Prolog using any type of predicate.

Q4. What is backtracking?

Q5. What are the disadvantages of backtracking?

Q6. Why are cut and fail used in Prolog?

Q7. What are the advantages and disadvantages of Prolog?

Q8. Why is Prolog used in artificial intelligence?

INDEX

A

Add new node, 127
Antecedent, 75, 76, 79, 138, 308, 309, 315, 317-319, 326, 327, 349
Arbitrary visual scene, 18
Arithmetic defuzzification, 330
Artificial intelligence, 1, 2, 5-7, 10, 11, 12, 14, 16-19, 21, 23, 24, 26, 31, 71, 129, 130, 247, 359, 363, 374, 375, 405
Artificial neural network, 23, 245, 249, 250, 252, 253, 258-262, 266, 268, 269, 289
Automatic optimization methods, 324

B

Backward chaining, 80, 82, 83, 85, 93
Biomedical applications, 260
Bisector, 331
Boolean algebra, 2
Boolean logic, 272, 274, 302-305, 310, 322

C

Center of gravity, 331, 336, 350, 351
Centroid, 330, 331, 336, 337
Centroid defuzzification method, 330
Character code, 378, 383
Class frame, 200-202, 204, 206, 209, 211
Cognitive modelling approach, 8
Concept (card) sorting, 123
Concept tree, 126, 127
Conceptual dependency (CD), 225
Consequence or conclusion, 308
Consequent, 67, 75-77, 80, 83, 138, 308, 309, 311, 320, 323, 327, 328, 330, 335, 337, 350, 361, 404
Crisp value, 330, 331, 336, 345

D

Debug, 106,
Defuzzification, 324, 329-332, 335-339, 341, 342, 344, 350, 352, 357
Depth first search (DFS), 44
Depth limited search (DLS), 40, 47, 48, 51
Discourses x, 308
Domain expert, 75, 78, 86-88, 91, 97, 99-105, 107, 110, 114, 116-123, 127, 321, 322
Domain knowledge, 324

E

Error correction gradient descent, 260
Expert systems, 6, 18, 19, 24, 25, 31, 54, 71-74, 76, 77, 82, 88-93, 95, 96, 107, 114, 242, 305, 311, 321-324, 331, 341, 363,

F

Feedback networks, 256
Finance field, 23, 30
Firing strength, 320
Formal logic, 9, 134-137, 140
Forward chaining, 80, 82, 85
Frames, 77, 78, 115, 127, 134, 135, 178, 194-197, 199, 200, 201, 202, 203, 204, 205, 206, 209, 212, 213,214, 215, 217, 224, 242, 321
Fuzzification, 324, 326, 327, 329, 330-338, 341, 342, 344, 347, 350, 352, 353, 357
Fuzzy expert systems, 305, 311, 321, 322, 341
Fuzzy input, 326
Fuzzy logic, 271-275, 285, 287, 291, 299, 302, 303-305, 317, 320, 322, 327, 334, 340,

342, 343, 348, 349
Fuzzy reasoning, 316, 317, 319, 320, 326, 330, 342
Fuzzy set, 272, 273, 277-279, 281-284, 286-289, 291-294, 297-300, 303-305, 307-309, 311, 319, 322, 323, 326, 328, 329-332, 333, 334-341, 347, 348-350, 353, 354

G

Game playing, 6, 13, 18, 28, 54, 66

I

Inference chain errors, 109, 110
Inference engine, 77, 79, 80, 82, 85, 86, 88, 93, 95, 102, 107, 109, 110, 135, 324, 326, 347, 364,
Inference engine errors, 109, 110
Input, 1, 8, 11, 25-28, 33, 38, 86, 88, 89, 99, 104, 149, 196, 244, 250, 275, 307, 321, 326, 331, 340, 383, 384, 388, 389, 405
Instance frame, 202-204, 210
Intelligent agent, 5, 22
Iterative deepening depth first search, 40, 48

K

Knowledge base, 24, 72, 75, 77-79, 82, 85, 87-89, 91, 95, 100, 102, 104, 106-108, 114-116, 118, 120, 126-127, 134, 150, 155, 173, 175, 183, 214, 228, 323, 324
Knowledge engineer, 24, 71, 75, 86-88, 90, 96, 98, 99, 101-102, 110, 117, 118, 121, 122, 125, 126, 185, 322
Knowledge error, 110

L

Laddering, 126
Least mean squares, 260
Linguistic rules, 307, 348
Linguistic variable in fuzzy set, 281
Linguistic variables, 281-282, 302, 308, 337, 342-343, 345, 348, 352, 354

M

Mamdani inference, 332
Meta knowledge, 115, 121, 324
Methodological advances, 260
Monotonic inference, 340, 341
Move nodes, 127

N

Natural language, 6-9, 12-15, 18-19, 26, 27, 132, 136, 137, 172, 214, 225, 241, 272, 275, 281, 323, 330, 345, 363
Network layers, 257
Neural network, 6, 23, 243-255
Neuron, 244, 245, 250, 253-254, 256, 261, 262
Non-linear mapping, 307

O

Ordering, 58, 217
Output predicate, 383, 384

P

Partitioned semantic network, 187-193
Pattern recognition, 1, 6, 20, 25-26, 31, 245, 255, 263, 268, 269
Predicate logic, 28, 141, 143-148, 162, 168, 177
Premise, 75, 172, 308, 317, 323, 347-350
Primitive conceptualization, 226, 236
Propositional logic, 136, 139-141, 148, 166

R

Resolution, 149-150, 152-153, 155-164, 166-169
Robotics, 6, 8, 12, 17, 19, 24, 356
Rule evaluation, 332, 338, 347
Rule-based systems 24, 71, 77

S

Script action, 216
Scripts, 77, 134, 135, 213-217, 219, 221, 224

Search problem, 34, 35, 39
Semantic errors, 109, 110
Semantic net, 78, 127, 135, 172-174, 176, 182, 184, 185, 225
Simulate, 6, 10, 12, 20, 95
Skolem form, 152
Skolem function, 152, 153, 155
Software agent, 5
State space search, 33-37, 64
Stochastic, 260
Subclass frame, 201, 202, 204
Sugeno inference method, 337-338
Syntax errors, 109-110

T

Thinking machine, 2
Three-card method, 123
Turing test, 2-4, 7-9, 12