Lecture Notes in Artificial Intelligence

Dr V N Krishnachandran

Department of Computer Applications (MCA)
Vidya academy of Science and Technology
Thrissur – 680 501

This page is intentionally left blank.

LECTURE NOTES IN ARTIFICIAL INTELLIGENCE

Dr V N Krishnachandran

Department of Computer Applications (MCA)
Vidya Academy of Science & Technology
Thrissur - 680501

Title: LECTURE NOTES IN ARTIFICIAL INTELLIGENCE

Author's name:
DR. V. N. KRISHNACHANDRN

Published by author
Pblisher's address: Dr. V. N. Krishnachandran
Vidya Academy of Science & Technology
Thalakkottukara, Thrissur - 680501, Kerala

First Edition

The book was typeset by the author using the LaTeX document preparation system.

Cover design: Author

Price: Rs 0.00.

First printing: April 2021

ISBN: $\cdots\cdots\cdots$

Preface

The book is written primarily for the students pursuing the MCA programme of the APJ Abdul Kalam Technological University. The Curriculum for the programme offers a course on artificial intelligence as an elective course in the Second Semester with code and name "20MCA188 Artificial Intelligence". The selection of topics in the book was guided by the contents of the syllabus for the course and also by the contents in the books cited in the syllabus. The book will also be useful to faculty members who teach the course. The users of these notes are earnestly requested to bring to the notice of the author any errors they find so that a corrected and revised edition can be published at the earliest.

VAST Campus
April 2021

Dr V N Krishnachandran
Department of Computer Applications
Vidya Academy of Science & Technology, Thrissur - 680501
(email: `krishnachandran.vn@vidyaacademy.ac.in`)

This page is intentionally left blank.

Contents

Syllabus

0. Course name and code

20MCA 188	Artificial Intelligence	Category	L	T	P	Credit
		Elective	3	1	0	4

1. Preamble

This course introduces the techniques of Artificial Intelligence and analyses various methods of solving problems using it. The concept of expert system architecture and fuzzy operations are introduced. This course serves as a prerequisite for many advanced courses in Data Science areas.

2. Prerequisite

Mathematical foundations for computing, advanced data structures

3. Course outcomes

After the completion of the course the student will be able to:

CO 1	Apply the steps needed to provide a formal specification for solving the problem.
CO 2	Apply and analyze the different types of control and heuristic search methods to solve problems
CO 3	Understand various Game theory problems and Knowledge structures
CO 4	Formulate knowledge representation and examine resolution in predicate and propositional logic
CO 5	Apply feasible planning and learning techniques to solve non-trial problems
CO 6	Analyze expert systems and fuzzy operations to solve real life problems.

4. Mapping of course outcomes to programme outcomes

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	3	3							2			
CO 2	3	3							2			
CO 3	3	3							2			
CO 4	3	3							2			
CO 5	3	3							2			
CO 6	3	3	3				3		2		2	2

5. Assessment pattern

Bloom's Category	Continuous Assessment Tests		End Semester Examination
	1	2	
Remember(K1)	10	10	10
Understand(K2)	20	20	20
Apply(K3)	20	20	30
Analyse(K4)			
Evaluate(K5)			
Create(K6)			

6. Marks distribution

Total Marks	Continuous Internal Evaluation (CIE)	End Semester Examination (ESE)	ESE Duration
100	40	60	3 hours

7. Continuous internal evaluation pattern

Attendance	:	8 marks
Continuous Assessment Test (2 numbers)	:	20 marks
Assignment/Quiz/Course project	:	12 marks

8. Course level assessment questions

Course Outcome 1 (CO 1)

1. Describe the areas of Artificial intelligence. (K1)
2. List the problem formulations & production characteristics. (K1 & K2)
3. Solve the various problems such as 8 puzzle, Crypt arithmetic, etc. (K3)

Course Outcome 2 (CO 2)

1. Describe search strategies in solving problems. (K1 & K2)
2. List the disadvantages of hill climbing algorithm. (K1& K2).

3. Illustrate A* algorithm for the graph. (K3)

Course Outcome 3 (CO 3)

1. Demonstrate two player Zero sum game. (K3)
2. List and explain the knowledge representation methods in AI. (K1 & K2)
3. Explain how alpha-beta algorithm works in pruning of branches with an example. (K3)

Course Outcome 4 (CO 4)

1. Translate the following sentence to predicate logic: (K3)
   (a) "All pompeians were Romans."
   (b) "All Romans were either loyal to Caesar or hated him."
2. Explain the algorithm to convert WFF to clause. (K1 & K2)
3. Describe about resolution graph in predicate and propositional logic. (K1 & K2)

Course Outcome 5 (CO 5)

1. Differentiate between Goal stack and Hierarchical planning in AI. (K1 & K2)
2. Discuss about neural net learning. (K1 & K2)
3. List out the steps in genetic learning. (K1 & K2)

Course Outcome 6 (CO 6)

1. Specify the components in expert system. (K1 & K2)
2. Solve various fuzzy operations. (K3)
3. List out and explain various tools and languages in AI. (K1 & K2)

8. Model question paper

Part A

1. List the applications areas in AI.
2. Solve the following cryptarithmetic problem:

```
  SEND +
  MORE
 -----
 MONEY
 -----
```

3. Explain iterative deepening search.
4. List the disadvantages of hill climbing.

5. Solve a simple two player Zero sum game.
6. Explain about conceptual dependency.
7. Explain inference rules in FOPL.
8. List components of a planning system.
9. Give a short note on role of an expert system.
10. List various fuzzy operations.

(10 $\times$ 3 = 30 marks)

Part B

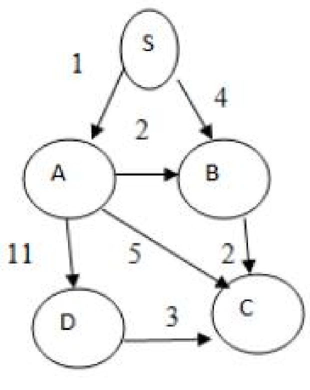
11. Consider a water jug problem. You are given two jugs, a 4 gallon and 3 gallons. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into 4-gallon jug. State the production rule for waterjug problem. (6)

OR

12. Solve missionaries and cannibals problem. (6)
13. Explain blind search strategies in detail. (6)

OR

14. Explain A* Algorithm for the given graph: (6)



S	A	B	C	D
6	2	3	0	2

15. List and explain the knowledge representation methods in AI. (6)

OR

16. Explain how alpha-beta algorithm works in pruning of branches with an example.(6)

17. Explain the algorithm to convert WFF to clause with an example. (6)

OR

18. Explain Neural net and Genetic learning methods in AI. (6)
19. Illustrate architecture of an expert system and mention its features. (6)

OR

20. Solve the following using various fuzzy set operations: (6)

$$A = \{0.1/1, 0.3/2, 0.45/3\}, \quad B = \{0.15/1, 0.34/2\}.$$

(5×6=30 Marks)

(**Note:** *Question 20 appears to be incomplete.*)

9. Syllabus

Module 1

Introduction to AI and production systems: AI Problem formulation, Problem definition, Production systems, Problem characteristics, Production system characteristics. **Example AI problems**: 8 puzzle problem, Missionary and cannibals problem, Cryptarithmetic problems, Block world problems.

Module 2

Search strategies: Blind search strategies: Depth first search, Breadth first search, Best first search, Iterative deepening search. Heuristic search strategies: Admissible heuristics and examples, Simple hill climbing and steepest ascending hill climbing, Simulated annealing, A* algorithm.

Module 3

Game playing: Two-player zero sum games, Modelling two-player zero sum games as search problems, Min-max algorithm, Optimizing Min-max algorithm using α - β cut off. **Knowledge representation structures**: Frames, Semantic networks, Conceptual dependencies.

Module 4

Knowledge representation using logic: First order predicate logic (FOPL), Well formed formula (WFF) in FOPL, Inference rules for FOPL, Clause form and conversion of WFFs to clause form, Resolution-refutation. **Planning**: Overview, components of a planning system, Goal stack planning, Hierarchical planning. **Learning**: Forms of learning, Neural net learning, Genetic learning.

Module 5

Expert systems: Architecture of expert systems, Roles of expert systems, Languages and tools - Typical expert system examples. **Fuzzy logic**: Fuzzy variables, Fuzzy sets and fuzzy set operations, Typical examples using fuzzy sets.

10. Text books

1. Kevin Night and Elaine Rich, "*Artificial Intelligence (SIE)*", McGraw Hill, 2008.
2. Stuart Russel and Peter Norvig, "*AI - A Modern Approach*", 2nd Edition, Pearson Education, 2007

11. Reference books

1. Peter Jackson, "*Introduction to Expert Systems*", 3rd Edition, Pearson Education, 2007.
2. Dan W. Patterson, "*Introduction to AI and ES*", Pearson Education, 2007.

12. Course contents and lecture schedule

No.	**Topic**	**Number of lectures**
1	**Module I: Introduction to AI**	**9 hours**
1.1	AI-Problem formulation, Problem Definition - Production systems	
1.2	Production system characteristics	
1.3	AI Problems	
2	**Module II: Search Strategies**	**9 hours**
2.1	Blind search strategies	
2.2	Heuristics search strategies	
2.3	Simple Hill Climbing and Steepest Ascending Hill Climbing,	
2.4	Simulated annealing	
2.5	A* algorithm	
3	**Module III: Game playing**	**9 hours**
3.1	Zero sum game	
3.2	Minimax algorithm	
3.3	Alpha beta pruning	
3.4	Knowledge representation structure	
4	**Module IV: Knowledge representation using Logic**	**12 hours**
4.1	First Order Predicate Logic (FOPL)	
4.2	Well Formed Formula(WFF) in FOPL, Inference rules for FOPL	
4.3	The Clause Form and conversion of WFFs to Clause Form	
4.4	Resolution	
4.5	Planning	
4.6	Learning	
5	**Module V: Applications**	**6 hours**
5.1	Expert system Architecture	
5.2	Fuzzy logic operations	
5.3	Languages and tools	

LECTURE NOTES IN
ARTIFICIAL INTELLIGENCE

This page is intentionally left blank.

Chapter 1

Introduction to artificial intelligence

In this introductory chapter, we shall see several definitions of artificial intelligence (the terminology is commonly abbreviated as AI), shall have a look at the various stages in the historical development of the discipline of artificial intelligence and shall have a glimpse of the many fields in which this discipline has been successfully applied.

1.1 What is artificial intelligence?

There is no universally accepted definition of artificial intelligence. The following are some of the well-known definitions of artificial intelligence:

1. "Artificial intelligence is the art of creating machines that perform functions that require intelligence when performed by people. (Kurzweil, 1990)"

2. "Artificial intelligence is the study of how to make computers do things at which, at the moment, people are better."

3. "An agent is something that acts. A rational agent is an agent that acts so as to achieve the best outcome or when there is uncertainty, the best expected outcome. Artificial intelligence is the study of the general principles of rational agents and of the components for constructing them."

1.2 History of artificial intelligence

The following are some of the important stages in the development of the study of artificial intelligence as the term is now generally understood.

1. **Work prior to the coining of the term "artificial intelligence" (1943 - 1955)**

   The first work that is now generally recognized as artificial intelligence was done by Warren McCulloch and Walter Pitts in around 1943. It was not then referred to as a work in artificial intelligence.

2. **The birth of artificial intelligence (1956)**

   The terminology "artificial intelligence" was coined by John McCarthy and used by him in a proposal for organising a two-month workshop in Dartmouth College in 1956. The year 1956 is generally considered as the year of birth of artificial intelligence.

3. **The period of early successes (1952 - 1969)**

   There were several successes in the early period. The General Problem Solver (a program that imitates human problem-solving protocols) by Allen Newell and Herbert Simon, Geometry Theorem Prover by Herbert Gelernter (1959), Arthur Samuel's programs for playing checkers (draughts), James Slagle's SAINT (Symbolic Automatic INTegrator) program (1963) to solve calculus integration problems, Daniel Bobrow's STUDENT program (1967) that solved algebra story problems, etc. were some of the notable achievements of this period.

4. **The period of knowledge-based systems (1969 - 1979)**

   This period saw the use of more powerful, domain-specific knowledge in developing programs that allow larger reasoning steps and can more easily handle typically occurring cases in narrow areas of expertise.

5. **The period when neural networks became important again(1986 - present)**

   Although neural networks had been in use in the early years of the development of artificial intelligence, due to the failure of researchers to meet exaggerated claims and expectations, funding for research in neural networks was halted till early 1980's. But due to a combination of several factors, interest in neural networks re-emerged in around 1986.

6. **The period when artificial intelligence began to adopt the scientific method (1987 - present)**

   This was the period when artificial intelligence came firmly under the scientific method. To be accepted, hypotheses must be subjected to rigorous empirical experiments, and the results must be analysed statistically for their importance. It is now possible to replicate experiments by using shared repositories of test data and code.

7. **The period when very large data sets began to be available (2001 - present)**

   Throughout the 60-year history of computer science, the emphasis has been on the algorithm as the main subject of study. For many problems, it makes more sense to worry about the data and be less concerned about what algorithm to apply. This is true because of the increasing availability of very large data sources: for example, trillions of words of English and billions of images from the Web; or billions of base pairs of genomic sequences.

1.3 Applications of artificial intelligence

Nowadays, the concepts, methods and tools of artificial intelligence are finding applications in more and more fields of human activity almost on a day by day basis (see Figure 1.1). So it is difficult to give a concise answer to the question "What are the applications of artificial intelligence?".

1.3.1 Some well known applications of artificial intelligence

As a sample, a few of the well known applications of artificial intelligence are listed below.

Figure 1.1: Applications of artificial intelligence

1. **Robotic vehicles**

   A self-driving car, also known as an autonomous vehicle (AV), driverless car, or robocar is a vehicle that is capable of sensing its environment and moving safely with little or no human input. Self-driving cars combine a variety of sensors to perceive their surroundings. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage.

2. **Speech recognition**

   Speech recognition is concerned with the recognition and translation of spoken language into text by computers. It is also known as automatic speech recognition, computer speech recognition or speech to text. Some speech recognition systems require "training" where an individual speaker reads text or isolated vocabulary into the system. The system analyses the person's specific voice and uses it to fine-tune the recognition of that person's speech, resulting in increased accuracy.

3. **Autonomous planning and scheduling**

   NASA's Remote Agent program was the first on-board autonomous planning program to control the scheduling of operations for a spacecraft. The Remote Agent generated plans from high-level goals specified from the ground and monitored the execution of those plans - detecting, diagnosing, and recovering from problems as they occurred. Successor program MAPGEN planned the daily operations for NASA's Mars Exploration Rovers, and MEXAR2 did mission planning for the European Space Agency's Mars Express mission in 2008.

4. **Game playing**

   IBM's DEEP BLUE became the first computer program to defeat the world champion in a chess match when it defeated Garry Kasparov in an exhibition match in 1997.

5. **Spam fighting**

   Each day, learning algorithms classify over a billion messages as spam, saving the recipient from having to waste time deleting them.

6. **Logistics planning**

   During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool (DART) to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The artificial intelligence planning techniques generated in hours a plan that would have taken weeks with older methods.

7. **Robotics**

   The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use. The company also deploys the more rugged PackBot to Iraq and Afghanistan, where it is used to handle hazardous materials, clear explosives, and identify the location of snipers.

8. **Machine translation**

   There are computer programs that can automatically translate from, say, Arabic to English. The program uses a statistical model built from examples of Arabic-to-English translations and from examples of English text totalling two trillion words.

1.3.2 Recent applications

The following are some of the recent applications of artificial intelligence.

1. **Agriculture**

   In agriculture new artificial intelligence advancements show improvements in gaining yield and to increase the research and development of growing crops. New artificial intelligence methods now predict the time it takes for a crop like a tomato to be ripe and ready for picking thus increasing efficiency of farming.

2. **Cybersecurity**

   The more advanced of these solutions use artificial intelligence and natural language processing (NLP) methods to automatically sort the data in networks into high risk and low-risk information. This enables security teams to focus on the attacks that have the potential to do real harm to the organization, and not become victims of attacks such as denial-of-service (DoS), malware and others.

3. **Finance**

   Banks use artificial intelligence systems today to organize operations, maintain bookkeeping, invest in stocks, and manage properties. Artificial intelligence can react to changes overnight or when business is not taking place.

4. **Healthcare**

   The primary aim of health-related artificial intelligence applications is to analyse relationships between prevention or treatment techniques and patient outcomes. Artificial intelligence programs are applied to practices such as diagnosis processes, treatment protocol development, drug development, personalized medicine, and patient monitoring and care. Artificial intelligence algorithms can also be used to analyse large amounts of data through electronic health records for disease prevention and diagnosis.

5. **. . . and many more**

   The tools of artificial intelligence are used in many, many more fields like virtual assistant or chatbots, retail, sports analytics, manufacturing and production, inventory management, etc.

1.4 Sample questions

(a) Short answer questions

1. Define artificial intelligence.
2. Why a particular year is considered as the year of birth of artificial intelligence?
3. What is “DEEP BLUE”?
4. Why artificial intelligence is now considered as a branch of science?
5. Briefly describe some recent application of artificial intelligence.
6. How artificial intelligence is used in healthcare?
7. List application areas of artificial intelligence.

(b) Long answer questions

1. Describe some of the important periods in the history of the development of artificial intelligence methods.
2. Describe some of the well known applications of artificial intelligence (at least five applications).

Chapter 2

Problem definition and examples

In this chapter we consider the formal definition of the concept of a "problem" and illustrate the concept with several well known examples. We also discuss different characteristics of a problem. The concept of a "production system" is also introduced in this chapter. We begin the chapter with a very brief discussion of a "rational agent" because the chapter is concerned with the behaviour of a special kind of rational agent known as "problem solving agent".

2.1 Rational agents

2.1.1 Agents

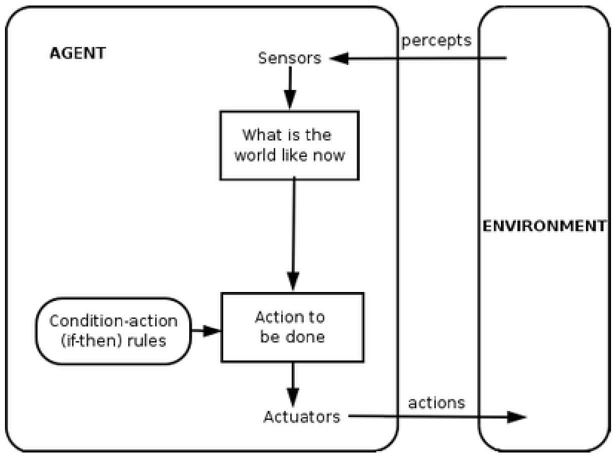


Figure 2.1: Schematic diagram of an agent

An **agent** is anything that can be viewed as perceiving (becoming aware of) its environment through sensors and acting upon that environment through actuators (or effectors) (see Fig 2.1).

Examples

- A **human agent** has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators.

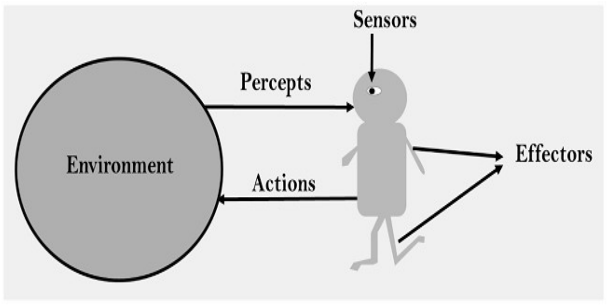


Figure 2.2: Schematic diagram of a human agent

- A **robotic agent** might have cameras and infrared range finders for sensors and various motors for actuators.
- A **software agent** receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

An agent's behaviour is described by the **agent function** that maps any given input sequence to an action. The agent function for an artificial agent is implemented by an **agent program**.

2.1.2 Rational agents

A **rational agent** is defined as follows:

> "A rational agent is an agent which, for each possible input sequence sensed by the sensors, selects an action that maximises its expected performance measure."

Throughout the rest of these notes, by an "agent" we shall always mean a "rational agent". Artificial intelligence is sometimes defined as a study of rational agents. A rational agent could be anything which makes decisions such as a person, firm, machine, or software.

This chapter describes the behaviour of a special kind of rational agent called a **problem-solving agent**. Our discussion of problem solving begins with precise definitions of problems and their solutions and give several examples to illustrate these definitions.

2.2 Definition of the concept of a problem in AI

In this section, we first consider an illustrative example to familiarise ourselves with the terminology associated with the formal definition of a problem. This is followed by the formal definition of a problem.

2.2.1 An illustrative example

We illustrate the the various components that constitute a problem by considering a well known problem called the "3–puzzle". The 3–puzzle is a special case of a more general puzzle known as the "$(n^2 - 1)$–puzzle", namely, the case where $n = 2$. The "8–puzzle", the $(n^2 - 1)$–puzzle with $n = 3$, is discussed in Section 2.3.2. Such problems are also referred to as "sliding-tile puzzles" or "sliding-tile problems".

Problem statement

Figure 2.3: 3-puzzle board

"The problem consists of a 2 × 2 board with three numbered tiles and a blank space (see Figure 2.3). A tile adjacent to the blank space can slide into the space. The objective is to reach the goal position shown in Figure 2.4 by sliding the tiles one at a time from start position as in Figure 2.4."

1	2
3	

Start

3	1
2	

Goal

Figure 2.4: Start position and goal position in a 3-puzzle

States of the problem

The "Start" and "Goal" shown in Figure 2.4 represent different states of the problem. As we move tiles to reach the goal from the start, the problem passes through several states. In fact, there are 24 different possible states for the problem as shown in Figure 2.5.

1	2
3	

s_1

1	2
	3

s_2

1	3
2	

s_3

1	3
	2

s_4

1	
2	3

s_5

1	
3	2

s_6

2	1
3	

s_7

2	1
	3

s_8

2	3
1	

s_9

2	3
	1

s_{10}

2	
1	3

s_{11}

2	
3	1

s_{12}

3	1
2	

s_{13}

3	1
	2

s_{14}

3	2
1	

s_{15}

3	2
	1

s_{16}

3	
1	2

s_{17}

3	
2	1

s_{18}

	1
2	3

s_{19}

	1
3	2

s_{20}

	2
1	3

s_{21}

	2
3	1

s_{22}

	3
1	2

s_{23}

	3
2	1

s_{24}

Figure 2.5: The different states of the 3-puzzle

Actions

To move from one state to another state we apply some action. In this problem an action can be thought of as moving the blank tile. Hence there are four possible actions that can be applied to any state: “move blank tile up”, “move blank tile down”, “move blank tile left” and “move blank tile right”. We may abbreviate these actions as “Up”, “Down”, “Left” and “Right”. Note that not all actions are applicable to all states.

The set of actions applicable to a state s is denoted by ACTIONS(s). Thus, from Figure 2.5, we can see that

$$\text{ACTIONS}(s_1) = \{\text{Up, Left}\}$$
$$\text{ACTIONS}(s_{19}) = \{\text{Right, Down}\}$$

Results of actions

The result of applying an action a to a state s is denoted by RESULT(s, a). For example, we have

$$\text{RESULT}(s_1, \text{Up}) = s_6$$
$$\text{RESULT}(s_{19}, \text{Down}) = s_8$$

A successor of a state is a state reachable by a single action. For example, s_8 is a successor of s_{19}. Also, s_5 is a successor of s_{19}.

State space

The initial state, the actions applicable to the initial states, the resulting states, the actions applicable to the resulting states and the resulting successors, and so on, all together constitute the state space for the problem. The state space can be represented in the form of a tree[1]. Figure 2.6 shows part of the state space assuming that the start state is as shown in Figure 2.4.

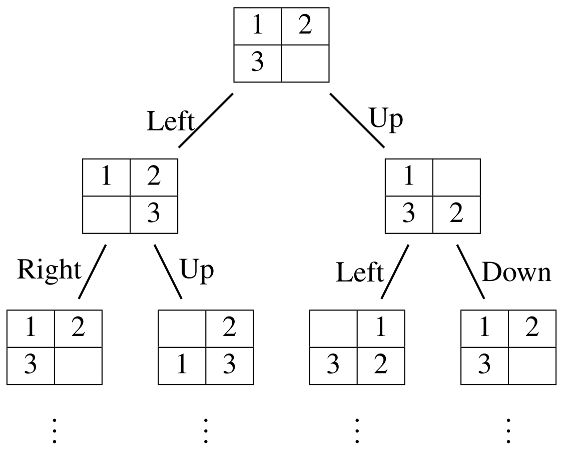


Figure 2.6: A part of the state space for the 3-puzzle

[1]For a precise definition of a tree, see Section 3.1.

Solution

A solution is an action sequence that leads from the initial state to a goal state. It can be easily verified that the following sequence of actions is a solution to the problem that we are talking about:

$$\text{Up} \rightarrow \text{Left} \rightarrow \text{Down} \rightarrow \text{Right}$$

The resulting states are shown in Figure 2.7.

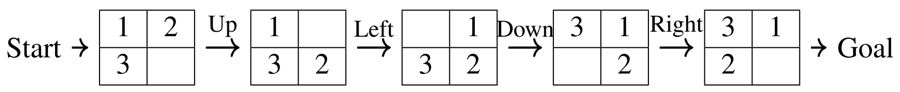


Figure 2.7: Solution to the 3-puzzle

Optimal solutions

We may assign a cost for executing each action in a problem. The costs depend on the problem. In the present problem, let us assign a cost of 1 unit for each action. Then, in the solution specified in Figure 2.7, the the total cost of arriving at the goal state is 4 units. In other words, the cost of the solution is 4 units. There may be solutions with lower costs. The solution with the least cost is referred to as the optimal solution.

The reader may verify that the solution given in Figure 2.7 is indeed an optimal solution for the problem under discussion.

2.2.2 Definition

A problem is defined formally by the following five components:

1. **Initial state**

   The **initial state** that the agent starts in.

2. **Actions**

   A description of the possible **actions** available to the agent. Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions are **applicable** in s.

3. **Transition model**

   A description of what each action does. The formal name for this is the **transition model**, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.

   The initial state, the actions, and the transition model all taken together implicitly define the **state space** of the problem – the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions. A **path** in the state space is a sequence of states connected by a sequence of actions.

4. **Goal test**

   The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

5. **Path cost**

   A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. We assume that the cost of a path can be described as the sum of the costs of the individual actions along the path.

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. The **optimal solution** is the solution having the lowest path cost among all solutions.

2.3 Example problems

2.3.1 The vacuum cleaner world

Description

This world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square.

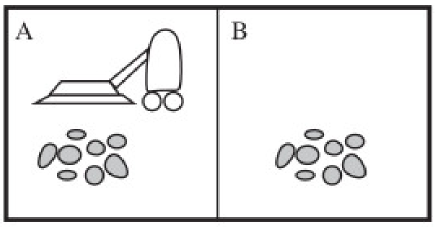


Figure 2.8: The Vacuum cleaner world

Standard formulation

- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states.

- **Initial state**: Any state can be designated as the initial state.

- **Actions**: In this simple environment, each state has just three actions: Left, Right, and Suck.

- **Transition model**: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in Figure 2.9.

- **Goal test**: This checks whether all the squares are clean.

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

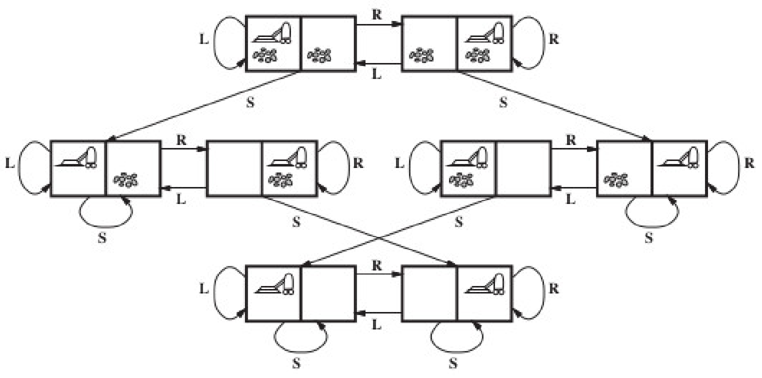


Figure 2.9: The state space for the vacuum world. Links denote actions: L = Left, R = Right, S = Suck.

2.3.2 8-puzzle problem

Recall the detailed discussion of the 3-puzzle in Section 2.2.1.

Description

The 8-puzzle consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The objective is to reach a specified goal state such as the one shown in Figure 2.10 by sliding the tiles one at a time, given some initial or start state as in Figure 2.10

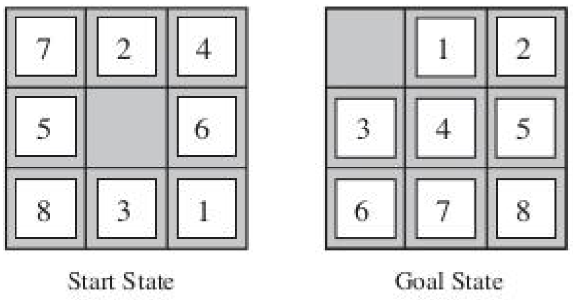


Figure 2.10: 8-puzzle

Standard formulation

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares. (The total number of states is $9! = 362880$.)
- **Initial state:** Any state can be designated as the initial state.
- **Actions:** Actions may be defined as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.

- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure 2.10, the resulting state has the 5 and the blank cell switched.
- **Goal test:** This checks whether the state matches the goal configuration shown in Figure 2.10.
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

2.3.3 Missionaries and cannibals problem

This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint.

Description

Three missionaries and three cannibals[2] are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. (See Figure 2.11.)

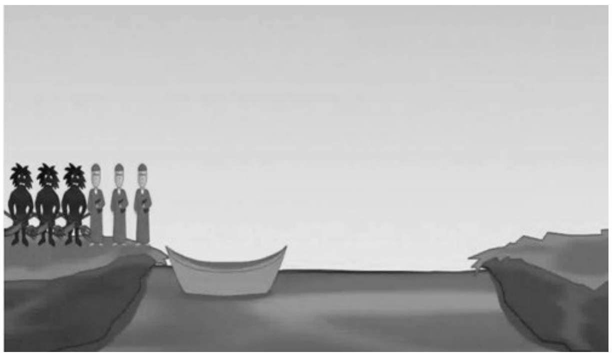


Figure 2.11: The missionaries and cannibals problem

Standard formulation

Ignore all irrelevant parts of the problem (e.g., weather conditions, possible presence of crocodiles in the river, etc.).

- **States:** Each state is represented by an ordered sequence of 3 numbers (x, y, z) where

x	:	number of missionaries on initial river bank
y	:	number of cannibals on initial river bank
z	:	number of boats on initial river bank

  For example, if triangles represent missionaries, circles represent cannibals and the left bank is the initial bank then the state shown in Figure 2.12 is specified by the ordered triple $(0, 1, 0)$. Note that the fact that the boat is at the destination bank is indicated by $z = 0$. Also it may be noted that not all such ordered triples represent

[2]A cannibal is a person who eats the flesh of other humans.

legal states. For example, the triple $(2, 3, 1)$ is not allowed because it represents a state where the number of cannibals outnumber the missionaries.

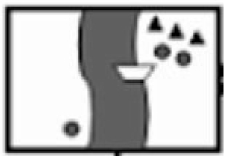


Figure 2.12: The state $(0, 1, 0)$ of the missionaries and cannibals problem

- **Initial state:** The ordered sequence $= (3, 3, 1)$ (see Figure 2.11).
- **Actions:** Take two missionaries, two cannibals, or one of each across in the boat. We have to take care to avoid illegal states.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply the action "Take 1 cannibal" to the state $(0, 2, 0)$ the resulting state is $(0, 3, 1)$.
- **Goal test:** We have reached state $(0, 0, 0)$.
- **Path cost:** Number of crossings.

Solution

Figure 2.13 shows the possible legal states reachable from the initial state applying only legal actions. In the figure, the notation " $\xrightarrow{\text{2c}}$ " indicates the operation of taking 2 cannibals from the bank where the boat is currently located to the other bank and the notation " $\xrightarrow{\text{2m}}$ " indicates taking two missionaries. From the figure we can easily get the solution to the problem as follows:

$$(3,3,1) \xrightarrow{\text{2c}} (3,1,0) \xrightarrow{\text{1c}} (3,2,1) \xrightarrow{\text{2c}} (3,0,0) \xrightarrow{\text{1c}} (3,1,1) \xrightarrow{\text{1m}} (1,1,0) \xrightarrow{\text{1m, 1c}}$$
$$(2,2,1) \xrightarrow{\text{2m}} (0,2,0) \xrightarrow{\text{1c}} (0,3,1) \xrightarrow{\text{2c}} (0,1,0) \xrightarrow{\text{1c}} (0,2,1) \xrightarrow{\text{2c}} (0,0,0)$$

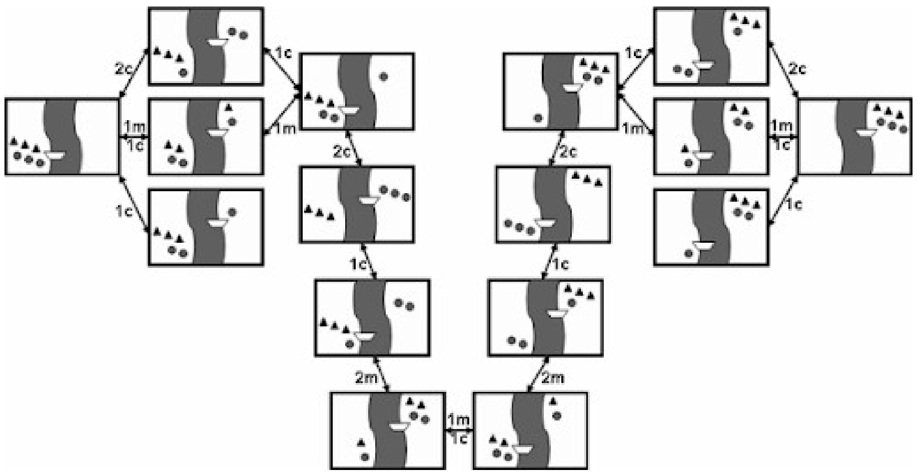


Figure 2.13: The state space of the missionaries and cannibals problem (with triangles representing missionaries and circles representing cannibals

2.3.4 Cryptarithmetic puzzle

This is an example for a special type of problem known as "constraint satisfaction problem". Constraint satisfaction problems are extensively studied and are the subject of intense research in artificial intelligence.

Description

A cryptarithmetic puzzle is a mathematical exercise where the digits of some numbers are represented by letters. Each letter represents a unique digit. It will be assumed that the leading digits of numbers are not zero. The problem is to find the digits such that a given mathematical equation is satisfied.

Illustrative example

Find the digits to be assigned to the letters such that the following equation is true (the digits assigned to T and F should not be 0):

```
  T W O +
  T W O
-------
F O U R
-------
```

Solution

With some trial and error we can get the following solution:

F = 1, O = 4, R = 8, T = 7, U = 6, W = 3.

That this is indeed a solution can be verified easily as follows.

```
  7 3 4 +
  7 3 4
-------
1 4 6 8
-------
```

Here is another solution.

```
  8 2 6 +
  8 2 6
-------
1 6 5 2
-------
```

There may still be other solutions.

Problem formulation

Using the idea of states the problem can be formulated as follows:

- **States:** Let there be n different letters where $n \leq 10$. A problem state is an ordered n-tuple of digits $(d_1, d_2, \ldots, d_n)$ representing the digits to be assigned to the integers.
- **Initial state:** The initial state can be considered as the ordered n-tuple all of whose elements are 0's.
- **Actions:** Increase the value assigned to a letter by 1.
- **Transition model:** Given a state and action, this returns the resulting state.
- **Goal test:** We have reached state $(d_1, d_2, \ldots, d_n)$ which satisfies the constraints of the problem.
- **Path cost:** Number of actions.

Solution

There are algorithms for solving the cryptarithmetic puzzle. Implementations of these algorithms by hand computations is difficult. The method of trial and error keeping in mind the constraints to be satisfied may be applied to obtain solutions. We illustrate the method by solving some simple cryptarithmetic problems.

Example problem 1

Solve the following cryptarithmetic puzzle:

```
   T O
+  G O
-----
O  U T
-----
```

Solution by hand computation

- Since leading digits are not 0, we have T $\neq$ 0, G $\neq$ 0, O $\neq$ 0.
- Since O is a non-zero carry digit when the two digits T an G are added, we must have O = 1.
- Since O = 1 and O + O = T, we must have T = 2.
- Since T = 2 and T +G $\geq$ 10, T + G must be either 10 or 11.
- If T + G = 10, then G = 8 and U = 0.
- This gives a solution to the problem, namely,

```
T = 2, O = 1, G = 8, U = 0.

  2 1
+ 8 1
-----
1 0 2
-----
```

- If T + G = 11, then U = 1 and in that case U = O = 1 which is not acceptable.
- Thus the problem has a unique solution as given above.

Example problem 2

Solve the following cryptarithmetic puzzle:

```
  S E N D
+ M O R E
---------
M O N E Y
---------
```

Solution

1. We number the columns as follows:

```
Col:5 4 3 2 1
    ---------
      S E N D
    + M O R E
    ---------
    M O N E Y
    ---------
```

2. From column 5, we must have $M = 1$ since it is the only carry possible from the sum of two single digit numbers.

```
Col:5 4 3 2 1
    ---------
      S E N D
    + 1 O R E
    ---------
    1 O N E Y
    ---------
```

3. To produce a carry from column 4 to column 5, $S + 1$ must be at least 9. We have $S + 1 = 9$ (if there is a carry over from column 3) or 10 (if there is no carry from column 3). So, we must have $S = 8$ or 9. Then $O = 0$ or 1. But we have already seen that $M = 1$. The same value cannot be assigned to O. Therefore we must have $O = 0$.

```
Col:5 4 3 2 1
    ---------
      S E N D
    + 1 0 R E
    ---------
    1 0 N E Y
    ---------
```

4. If there is carry from column 3 to 4 then $E = 9$ and so $N = 0$. But $O = 0$ so there is no carry and $S = 9$ and the carry-over from column 3 is 0.

```
Col:5 4 3 2 1
    ---------
      9 E N D
    + 1 0 R E
    ---------
    1 0 N E Y
    ---------
```

5. If there is no carry from column 2 to 3 then $E = N$ which is impossible, therefore there is carry and
$$N = E + 1$$
and the carry from column 2 to 3 is 1.

6. (a) If there is no carry from column 1 to 2 then
$$N + R = 10 + E$$
and since $N = E + 1$, we have
$$E + 1 + R = 10 + E.$$
Therefore $R = 9$. But $S = 9$ and so R cannot be 9.
   (b) So there must be carry from column 1 to 2 and hence $R = 8$.

```
Col:5 4 3 2 1
    ---------
      9 E N D
    + 1 0 8 E
    ---------
    1 0 N E Y
    ---------
```

7. (a) To produce a carry of 1 from column 1 to 2, we must have
$$D + E = 10 + Y.$$
   (b) Since 0 and 1 has already been assigned, Y cannot be 0 or 1. Hence $D + E$ is at least 12.

   (c) As 8 and 9 have already been assigned, D is at most 7 and so E is at least 5.
   (d) Also N is at most 7 and $N = E + 1$ so we must have $E = 5$ or 6.

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

```
Col:5 4 3 2 1
    ---------
      9 5 6 D
    + 1 0 8 5
    ---------
    1 0 6 5 Y
    ---------
```

9. $D + E$ is at least 12 for that we must have $D = 7$ and $Y = 2$.

```
Col:5 4 3 2 1
    ---------
      9 5 6 7
    + 1 0 8 5
    ---------
    1 0 6 5 2
    ---------
```

10. The problem has unique solution, namely,

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

2.3.5 Blocks world problem

Blocks world (or, the world of blocks) is a model domain used in artificial intelligence to explore different approaches to automated reasoning. This model is used to illustrate that a given algorithm can perform planning, or that it is efficient in terms of the number of calculations required to find a solution or in terms of the length of that solution.

Description

There is a table on which some uniform blocks (cubes) are placed. Some blocks may or may not be stacked on other blocks. We have a robot arm to pick up or put down the blocks. The robot arm can move only one block at a time, and no other block should be stacked on top of the block which is to be moved by the robot arm. The problem is to change the configuration of the blocks from a given initial state to a given goal state (see, for example, Figure 2.14).

Problem formulation

The problem can be formulated as follows:

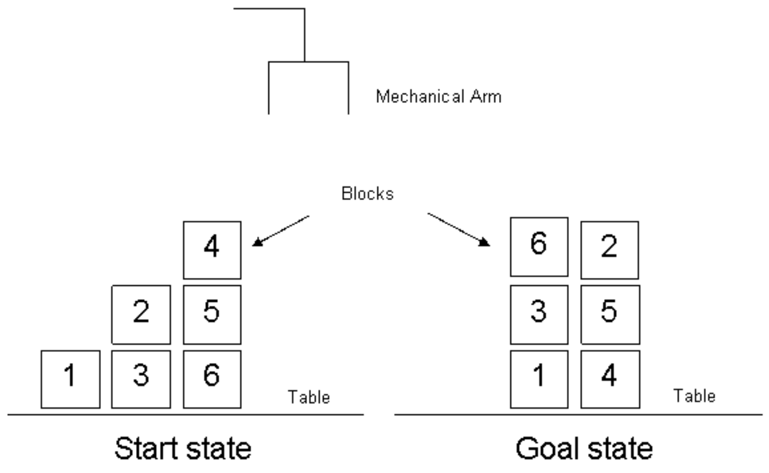


Figure 2.14: A blocks world problem

- **States:** Configurations of the blocks satisfying the conditions. The configurations can be specified using the following predicates where A and B denote arbitrary blocks:

ON(A,B)	:	block A is on block B.
ONTABLE(A)	:	block A is on the table.
CLEAR(A)	:	block A has nothing on it.
HOLDING(A)	:	the arm holds block A.
ARMEMPTY	:	the arm holds nothing.

- **Initial state:** The initial configuration of blocks.

- **Actions:**

UNSTACK(A,B)	:	pick up clear block A from block B and hold it in the arm
STACK(A,B)	:	place block A held in the arm onto clear block B
PICKUP(A)	:	lift clear block A with the empty arm
PUTDOWN(A)	:	place block A held in the arm onto a free space on the table.

- **Goal state:** A certain configuration of the blocks.

- **Path cost:** Number of actions taken to reach the goal state.

Solution

The blocks world problem has always a trivial solution: All blocks not already correctly positioned for the goal state be set off onto the table (one at a time with the mechanical arm), and then reassembled in the proper order on top of any blocks already correctly positioned.

To illustrate the idea consider the blocks world problem specified in Figure 2.14. A solution consists of the following 12 moves in that order:

1. UNSTACK(2, 3)
2. PUTDOWN(2)
3. UNSTACK(4, 5)
4. PUTDOWN(4)
5. UNSTACK(5, 6)
6. STACK(5, 4)
7. PICKUP(2)
8. STACK(2, 5)
9. PICKUP(3)
10. STACK(3, 1)
11. PICKUP(6)
12. STACK(6, 3)

2.3.6 Monkey and banana problem

The monkey and banana problem is a famous toy problem in artificial intelligence, particularly in logic programming and planning.

Description

A monkey is in a room. A bunch of bananas is hanging from the ceiling. The monkey cannot reach the bananas directly. There is box in the room. The ceiling is just the right height so that a monkey standing on the box can get hold of the bananas. The monkey knows how to move around and carry things around. How can the monkey get the bananas? What is the best sequence of actions for the monkey? (See Figure 2.15.)

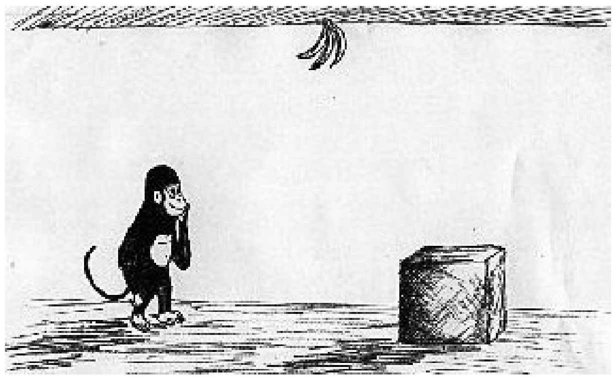


Figure 2.15: Monkey and banana

Remark

The purpose of the problem is to raise the question: Are monkeys intelligent? Both humans and monkeys have the ability to use mental maps to remember things like where to go to find shelter, or how to avoid danger. They can also remember where to go to gather food and water, as well as how to communicate with each other. Monkeys have the ability not only to remember how to hunt and gather but to learn new things, as is the case with the monkey and the bananas: despite the fact that the monkey may never have been in an identical situation, with the same artifacts at hand, a monkey is capable of concluding that it needs to make a ladder, position it below the bananas, and climb up to reach for them.

Problem formulation

Let the initial location of the monkey be A, that of the bananas be B and that of the box be C. Initially, the monley and box have height "low"; but when the monkey climbs on the box, it will have height "high".

- **States:** Each state is represented by the location and height of the monkey. bananas and the box.
- **Initial state:**

  Location(monkey, A), Location(bananas, B), Location(box, C),
  Position(monkey, low), Position(bananas, high), Position(box, low)
- **Actions:** Let x and y be locations. The following is a simplified description of the available actions. Clearly, to apply these actions certain preconditions are to be met.

  GO(x, y): Go from location x to y.

  PUSH(*object*, x, y): Push *object* from location x to y.

  CLIMBUP(*object*, x): Climb up on the *object* at location x.

  CLIMBDOWN(*object*, x): Climb down from the *object* at location x.

  GRASP(*object*, x, *height*): Grasp *object* located at x in position *height*)
- **Transition model:** Given a state and action, this returns the resulting state.
- **Goal test:** The goal state can be specified as follows:

  Location(monkey, A), Location(bananas, B), Location(box, C),
  Position(monkey, low), Position(bananas, low), Position(box, low)
- **Path cost:** Number of actions.

Solution

Step 1. GO(A,C)

Step 2. PUSH(box, C, B)

Step 3. CLIMBUP(box, B)

Step 4. GRASP(bananas, B, high)

Step 5. CLIMBDOWN(box, B)

Step 6. GO(B, A)

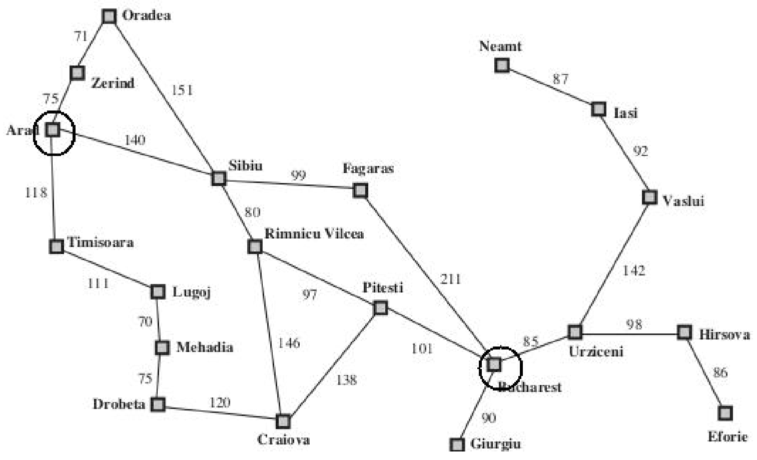


Figure 2.16: A road map of Romania

2.3.7 A map problem

Description

Given a road map of Romania (see Figure 2.16), find the shortest route from Arad to Bucharest. (In the figure, the numbers written by the sides of the line segments are the distances between the cities the line segment connects.)

Problem formulation

Using the idea of states the problem can be formulated as follows:

- **States:** Each city in the map represents a state.
- **Initial state:** The initial state is the city of Arad.
- **Actions:** Travel to a connected nearby city.
- **Transition model:** Given a state and action, this returns the resulting state.
- **Goal test:** The city of Bucharest is reached.
- **Path cost:** The total distance traveled in reaching the goal state.

2.3.8 Water jug problem

Problem statement

We have two jugs of capacity 4 liters and 3 liters, and a tap with an endless supply of water. The objective is to obtain 2 liters of water exactly in the 4-liter jug with the minimum steps possible.

Problem formulation

- **States:** Let x denote the number of liters of water in the 4-liter jug and y the number of liters of water in the 3-liter jug. Now $x = 0, 1, 2, 3$, or 4 and $y = 0, 1, 2$, or 3. The ordered pair (x, y) represents a state.

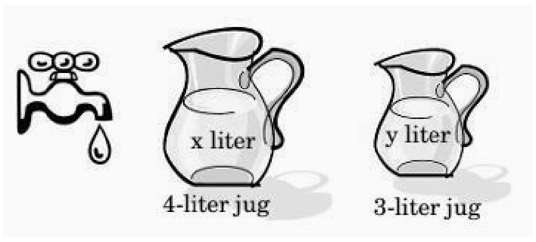


Figure 2.17: The water jug problem

- **Initial state:** The ordered pair $(0, 0)$.
- **Actions:** Each action is represented in the form "$(x, y) \to (u, v)$" where (x, y) represents the state before the application of the action and (u, v) represents the state after the application of the action. The possible actions are given in Table 2.1. The table also specifies the conditions under which the various actions can be applied.

Sl No.	State before action		State after action	Description of operation
1	(x, y) if $x < 4$	$\to$	$(4, y)$	Fill 4-liter jug
2	(x, y) if $y < 3$	$\to$	$(x, 3)$	Fill 3-liter jug
3	(x, y) if $x > 0$	$\to$	$(0, y)$	Empty 4-liter jug on the ground
4	(x, y) if $x > 0$	$\to$	$(x, 0)$	Empty 3-liter jug on the ground
5	(x, y) if $x + y \geq 4$ and $y > 0$	$\to$	$(4, y - (4 - x))$	Pour water from 3-liter jug into 4-liter jug until 4-liter jug is full
6	(x, y) if $x + y \geq 3$ and $x > 0$	$\to$	$(x - (3 - y), 3)$	Pour water from 4-liter jug into 3-liter jug until 3-liter jug is full
7	(x, y) if $x + y \leq 4$ and $y > 0$	$\to$	$(x + y, 0)$	Pour all water from 3-liter jug into 4-liter jug
8	(x, y) if $x + y \leq 3$ and $x > 0$	$\to$	$(0, x + y)$	Pour all water from 4-liter jug into 3-liter jug

Table 2.1: Actions in the water jug problem

- **Transition model:** The production system given in Table 2.1 also specifies the transition model.
- **Goal test:** The state $(2, n)$ where n is some integer.
- **Path cost:** Number of operations performed.

Solution

One solution to the water jug problem is given below:

$$(0,0) \xrightarrow{\text{Rule 2}} (0,3) \xrightarrow{\text{Rule 7}} (3,0) \xrightarrow{\text{Rule 2}} (3,3) \xrightarrow{\text{Rule 5}} (4,2) \xrightarrow{\text{Rule 3}} (0,2) \xrightarrow{\text{Rule 7}} (2,0).$$

2.4 Problem characteristics

2.4.1 Characteristics

The following are the important characteristics of a problem.

1. Problem is decomposable to smaller or easier problems.
2. Solution steps can be ignored or undone.
3. The problem universe is predictable.
4. There are obvious good solutions without comparisons to all other possible solutions.
5. Desired solution is a state of the universe or a path to a state.
6. Requires lots of knowledge; or, uses knowledge to constrain solutions.
7. Problem requires periodic interaction between humans and computer.

2.5 Examples illustrating problem characteristics

2.5.1 Problem is decomposable to smaller or easier problems

Example 1

Consider the problem: Evaluate the integral

$$\int (x^2 + \sin^2 x)\, dx.$$

We show that this problem is decomposable to smaller subproblems (see Figure 2.18). Since

$$\int (x^2 + \sin^2 x)\, dx = \int x^2\, dx + \int \sin^2 x\, dx,$$

the problem can be decomposed into two simpler subproblems.

- Problem (i): Evaluate $\int x^2\, dx$.
- Problem (ii): Evaluate $\int \sin^2 x\, dx$.

$$\begin{tikzcd}
 & \boxed{\int (x^2 + \sin^2 x)\, dx} \arrow[dl] \arrow[dr] & & \\
\boxed{\int x^2\, dx} \arrow[d] & & \boxed{\int \sin^2 x\, dx} \arrow[d] & \\
\boxed{\frac{1}{3}x^3} & & \boxed{\int \frac{1}{2}(1 - \cos 2x)\, dx} \arrow[dl] \arrow[dr] & \\
 & \boxed{\int \frac{1}{2}\, dx} \arrow[d] & & \boxed{\int -\frac{1}{2} \cos 2x\, dx} \arrow[d] \\
 & \boxed{\frac{1}{2}x} & & \boxed{-\frac{1}{4} \sin 2x}
\end{tikzcd}$$

Figure 2.18: Decomposing a problem into smaller subproblems

The first subproblem can be easily solved to get

$$\int x^2\, dx = \tfrac{1}{3}x^3.$$

Since

$$\begin{aligned}\int \sin^2 x\, dx &= \int \tfrac{1}{2}(1 - \cos 2x)\, dx \\ &= \int \tfrac{1}{2}\, dx - \int \cos 2x\, dx,\end{aligned}$$

Problem (ii) can be divided into two subproblems:

- Problem (iii): Evaluate $\int \frac{1}{2}\, dx$.
- Problem (iv): Evaluate $\int -\frac{1}{2} \cos 2x\, dx$.

These subproblems can be easily solved as

$$\begin{aligned}\int \tfrac{1}{2}\, dx &= \tfrac{1}{2}x \\ \int \tfrac{1}{2} \cos 2x\, dx &= -\tfrac{1}{2} \sin 2x\end{aligned}$$

Combining the solutions of the subproblems we get a solution of the given problem.

Example 2

Consider a simple blocks world problem specified by Figure 2.19.

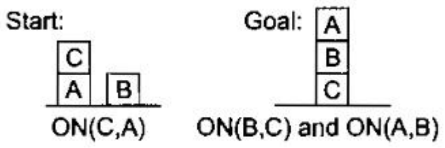


Figure 2.19: A simple blocks world problem

The following are the solution steps :

UNSTACK(C,A)

PUTDOWN(C)
PICKUP(B)
STACK(B,A)
STACK(C,B)

Since, the steps are interdependent the problem *cannot be decomposed* into two independent problems.

2.5.2 Solution steps can be ignored or undone.

Example 1

Suppose our goal is to prove a theorem in mathematics. On the way, we prove some preliminary results, say Result 1, Result 2, Result 3, and then finally we prove the theorem. The steps in the proof look like this:

Result 1
Result 2
Result 3
Theorem

Suppose, we later realize that Result 2 is not actually needed in proving the theorem. Then, we may *ignore* Result 2 and present the steps of the proof as follows:

Result 1
Result 3
Theorem

In this example, the solution steps *can be ignored.*

Example 2

Consider the 8-puzzle. The solution involves a sequence of moves. In the process of finding a solution, after some moves, we realize that a certain previous move has be reversed. The previous move can be *undone* by backtracking the moves. In this example, the *solution steps can be ignored.*

Example 3

Consider the problem of playing chess. If a chess playing programme makes a wrong move, the rules of the game do not permit undoing the move. In this example, the solution steps *cannot be undone or ignored.*

2.5.3 The problem universe is predictable.

Example 1

In the 8-puzzle, every time we make a move we know exactly what will happen. This means that it is possible to plan an entire sequence of moves. Thus in this problem, the universe is *predictable*.

Example 2

In a game like bridge, this is not possible because a player does not know where the various cards are and what the other players will do on their turns. In this problem, the universe is *unpredictable*.

2.5.4 There are obvious good solutions without comparison to all other possible solutions.

Example 1

Consider a mathematical problem. In general, there may be many methods for solving the problem. Any method is a good method without comparison to other methods provided it solves the problem. In general, any "any-path problem" is an example of a problem having obvious good solutions without comparison to other possible solutions.

Example 2

A "best-path problem" is a problem having no obvious good solutions. The travelling salesman problem is an example for a best-path problem. The travelling salesman problem can be formulated as follows: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

2.5.5 Desired solution is a state of the universe or a path to a state.

Example 1

In the missionaries and cannibals problem, if we organise the various states in the form of a tree, it can be seen that the solution to the problem is a path connecting the various states (see Figure 2.13).

Example 2

In the cryptarithmetic puzzle, the solution to the problem is a state of the problem, namely, the state representing the assignment of digits to the letters in the puzzle.

2.5.6 Requires lots of knowledge; or, uses knowledge to constrain solutions.

Example 1

Consider the problem of playing chess. The amount of knowledge required to play chess is very little: just the rules of the game! Additional knowledge about strategies may be used to make intelligent moves!!

Example 2

Consider the problem of scanning daily newspapers to decide which are supporting and which are opposing a political party in an upcoming election. It is obvious that a great deal of knowledge is required to solve this problem.

2.5.7 Problem requires periodic interaction between human and computer.

Example

Even in a so called "fully automated system" situations constantly arise that call for human intervention. When the machines get thrown off track, or become faulty, experts have to be summoned to step in and troubleshoot the problems.

2.6 Production systems

The term "production system" refers to many things. It may refer to a computer program which is used to provide a solution for a problem using a set of rules. It may also refer to a programming language for writing such programs. Further it can also be thought of as a model of computation that can be applied to implement search algorithms or as a model of human problem solving.

However, here, we define a production system as a structure having certain well defined components as described below.

2.6.1 Definition

A **production system** (also called *production rule system*) consists of the following:

1. **Set of rules**

   This is the most important component of a production system. Briefly, a production rule is a rule of the form "$C \to A$" which may be interpreted as "*given condition C, take action A*". Thus, each rule in the set of rules consists of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.

2. **Knowledge databases**

   The databases contain whatever information is appropriate for the particular task. Some parts of the database may be permanent.

   The database includes a **working memory** whose elements may pertain only to the solution of the current problem. Working memory contains a description of the current state of the world in the problem-solving process. The description is matched against the conditions of the production rules. When a condition matches, its action is performed. *Actions are designed to alter the contents of working memory.*

3. **Control strategy**

   The **control strategy** includes a specification of the order in which the rules are to be applied and a **conflict resolution strategy**.

   (a) **Order of application of rules**

   There are two methods of execution for a rule system:

   - **Forward chaining**
     In forward chaining systems, the execution of rules starts from facts in databases and moves towards some goal or conclusion (see Figure 2.20).

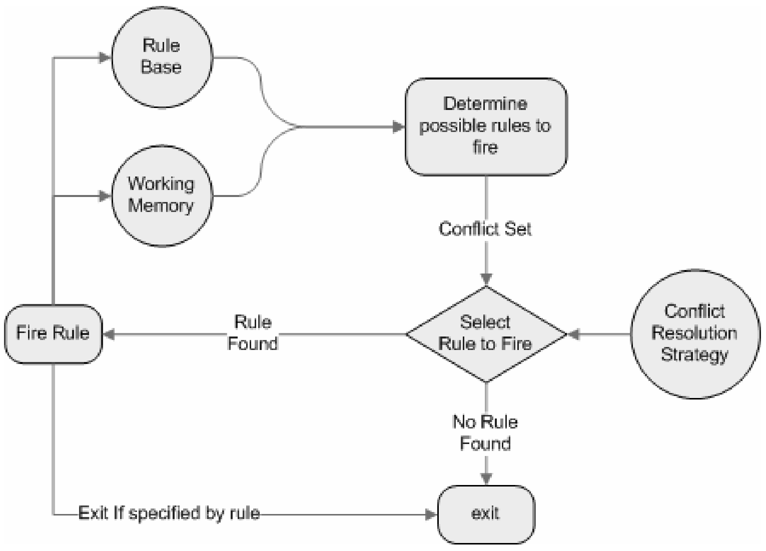


Figure 2.20: Recognise-act (or, select-execute) cycle: Forward chaining model

- **Backward chaining**
  In backward chaining, the execution of rules starts from the goals and conclusions and works backwards to see if there is any data from which the goals and conclusions can be derived (see Figure 2.21).

(b) **Conflict resolution strategy**

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. The conflict resolution strategy is a way of resolving the conflicts that manages the execution order of these conflicting rules. Some of the commonly used conflict resolution strategies are listed below:

- Choose arbitrarily.
- Choose the lowest numbered rule.
- Choose the rule containing the largest number of conditions.
- Choose the least recently used rule.
- Choose a rule where the condition is a new fact.
- Choose the rule with the highest priority (weight).

4. **Interpreter**

The interpreter repeats the following operations: all rules whose conditions are satisfied are found (rule matching), one of them is selected (conflict resolution), and its action is called (rule firing).

The interpreter (also referred to as rule applier) can be viewed as a “select-execute loop”, in which one rule applicable to the current state of the data base is chosen, and then executed. Its action results in a modified data base, and the select phase begins again. This cycle is also referred to as a “recognize-act cycle”.

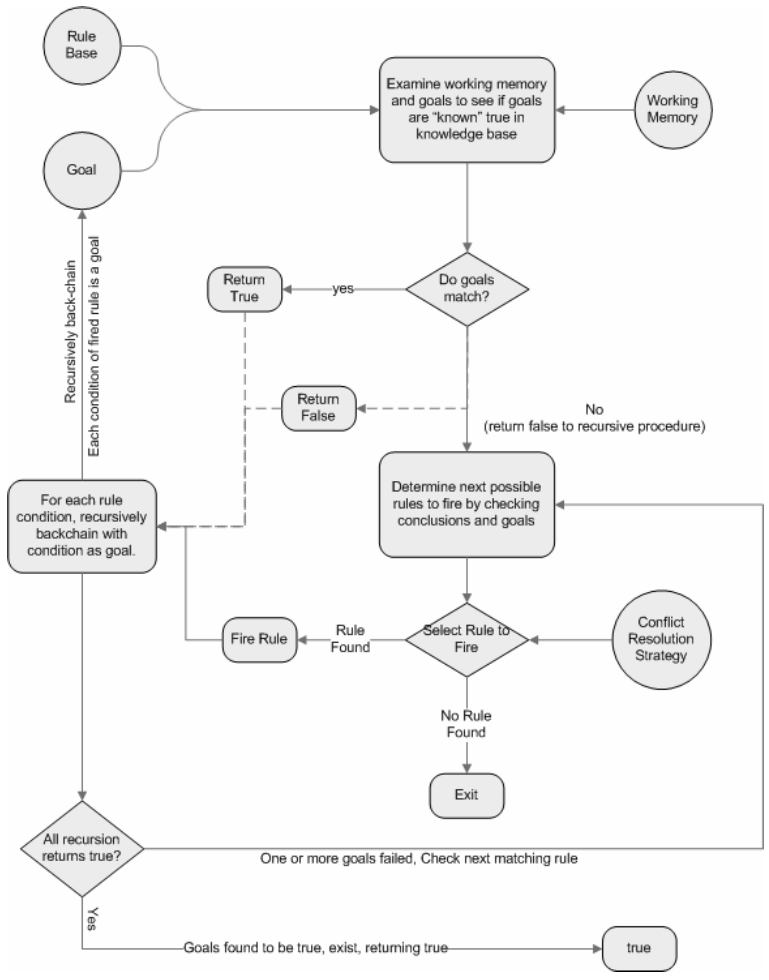

Rule Base
Goal
Examine working memory and goals to see if goals are "known" true in knowledge base
Working Memory
Recursively back-chain
Each condition of fired rule is a goal
Return True
yes
Do goals match?
Return False
No
(return false to recursive procedure)
For each rule condition, recursively backchain with condition as goal.
Determine next possible rules to fire by checking conclusions and goals
Fire Rule
Rule Found
Select Rule to Fire
Conflict Resolution Strategy
No Rule Found
Exit
All recursion returns true?
One or more goals failed, Check next matching rule
Yes
Goals found to be true, exist, returning true
true

Figure 2.21: Backward chaining model

2.6.2 Production system example: 3×3 knight's tour

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square exactly once. In the 3×3 knight's tour problem, instead of the usual 8×8 board in chess, we consider a 3×3 board as in Figure 2.22.

Problem statement

Using the production system model, determine whether a path exists from square 1 to square 2 in the 3×3 knight's tour problem (see Figure 2.22).

1	2	3
4	5	6
7	8	9

Figure 2.22: A 3×3 chess board

Solution

Step 1. Problem formulation as a production system

(i) **Knowledge database**

Since there is no information other than that contained in the statement of the problem, the database contains only the working memory. The working memory contains the following two facts:

$$\{\text{knight on square } 1, \text{knight on square } 2\}$$

We denote this initial working memory by the set $\{1, 2\}$.

(ii) **Production rules**

The set of production rules is given in Table 2.2.

Rule No.	Condition		Action
1	knight on square 1	$\rightarrow$	knight on square 8
2	knight on square 1	$\rightarrow$	knight on square 6
3	knight on square 2	$\rightarrow$	knight on square 9
4	knight on square 2	$\rightarrow$	knight on square 7
5	knight on square 3	$\rightarrow$	knight on square 4
6	knight on square 3	$\rightarrow$	knight on square 8
7	knight on square 4	$\rightarrow$	knight on square 9
8	knight on square 4	$\rightarrow$	knight on square 3
9	knight on square 6	$\rightarrow$	knight on square 1
10	knight on square 6	$\rightarrow$	knight on square 7
11	knight on square 7	$\rightarrow$	knight on square 2
12	knight on square 7	$\rightarrow$	knight on square 6
13	knight on square 8	$\rightarrow$	knight on square 3
14	knight on square 8	$\rightarrow$	knight on square 1
15	knight on square 9	$\rightarrow$	knight on square 2
16	knight on square 9	$\rightarrow$	knight on square 4

Table 2.2: The set of production rules for the 3×3 knight's tour problem

.

(iii) **Conflict resolution strategy**

Select and fire the first rule in the conflict set that does not lead to a repeated state.

(iv) **Interpreter**

Interpreter is the select-execute loop.

Step 2. Application of the rules

Table 2.3 shows the sequence of the applications of the rules starting from the state represented by "knight on square 1" in the working memory.

Step no.	Working memory	Current square	Conflict set	Fire rule	Remarks
0	$\{1, 2\}$	1	–	–	Initial state
1	$\{1, 2\}$	1	1, 2	1	–
2	$\{8, 2\}$	8	13, 14	13	–
3	$\{3, 2\}$	3	5, 6	5	–
4	$\{4, 2\}$	4	7, 8	7	–
5	$\{9, 2\}$	9	15, 16	15	–
6	$\{2, 2\}$	2	–	–	Halt

Table 2.3: Sequence of the applications of the production rules in the 3×3 knight's tour problem

Step 3. Conclusion

There exists a path from the initial state "knight on square 1" to the goal state "knight on square 2".

2.6.3 Production system example: The water jug problem

We have already seen the formulation of the water jug problem in the standard format of a problem in artificial intelligence (see Section 2.3.8). We now formulate the same problem in the format of a production system and then provide a solution to the problem using the solution method of production systems.

Problem statement

We have two jugs of capacity 4 liters and 3 liters, and a tap with an endless supply of water. The objective is to obtain 2 liters of water exactly in the 4-liter jug with the minimum steps possible.

Solution

Step 1. Problem formulation as a production system

As in section 2.3.8, we represent a state of the problem by an ordered pair (x, y) of integers where x denotes the number of liters of water in the 4-liter jug and y the number of liters of water in the 3-liter jug.

1. **Knowledge database**

   Since there is no information other than that contained in the statement of the problem, the database contains only the working memory. The working memory contains the following two facts:

$$\{(0,0),(2,0)\}$$

where $(0,0)$ is the initial state and $(2,0)$ is the goal state.

2. **Production rules**

   The set of production rules is given in Table 2.4. This is identical with Table 2.1 except for the differences in the column titles.

Sl No.	Condition		Action	Description
1	(x,y) if $x<4$	$\rightarrow$	$(4,y)$	Fill 4-liter jug
2	(x,y) if $y<3$	$\rightarrow$	$(x,3)$	Fill 3-liter jug
3	(x,y) if $x>0$	$\rightarrow$	$(0,y)$	Empty 4-liter jug on the ground
4	(x,y) if $x>0$	$\rightarrow$	$(x,0)$	Empty 3-liter jug on the ground
5	(x,y) if $x+y\geq 4$ and $y>0$	$\rightarrow$	$(4,y-(4-x))$	Pour water from 3-liter jug into 4-liter jug until 4-liter jug is full
6	(x,y) if $x+y\geq 3$ and $x>0$	$\rightarrow$	$(x-(3-y),3)$	Pour water from 4-liter jug into 3-liter jug until 3-liter jug is full
7	(x,y) if $x+y\leq 4$ and $y>0$	$\rightarrow$	$(x+y,0)$	Pour all water from 3-liter jug into 4-liter jug
8	(x,y) if $x+y\leq 3$ and $x>0$	$\rightarrow$	$(0,x+y)$	Pour all water from 4-liter jug into 3-liter jug

Table 2.4: Production rules of the water jug problem

3. **Conflict resolution strategy**

   Select and fire the first rule in the conflict set that does not lead to a repeated state.

4. **Interpreter**

   Interpreter is the select-execute loop.

Step 2. Application of the rules

The various steps in the solution procedure obtained by using the forward chaining method is shown in Table 2.5.

Step No.	Working memory	Current state	Conflict set	Fire rule	Remarks
0	$\{(0,0),(2,0)\}$	$(0,0)$	–	–	Initial state
1	$\{(0,0),(2,0)\}$	$(0,0)$	$1,2$	1	–
2	$\{(4,0),(2,0)\}$	$(4,0)$	2, 3, 4, 6	2	–
3	$\{(4,3),(2,0)\}$	$(4,3)$	3, 4, 5, 6	3	–
4	$\{(0,3),(2,0)\}$	$(0,3)$	1,7	7	–
5	$\{(3,0),(2,0)\}$	$(3,0)$	1, 2, 3, 4, 6, 8	2	–
6	$\{(3,3),(2,0)\}$	$(3,3)$	1, 3, 4, 5, 6	5	–
7	$\{(4,2),(2,0)\}$	$(4,2)$	2, 3, 4, 5, 6	3	–
8	$\{(0,2),(2,0)\}$	$(0,2)$	1, 2, 7	7	–
9	$\{(2,0),(2,0)\}$	$(2,0)$	–	–	Goal state

Table 2.5: Sequence of the applications of the production rules to reach the goal state in the water jug problem

Let us examine the working of the conflict resolution strategy in this example. Consider Step No. 6 in Table 2.5. The current state is $(3,3)$. If we examine the rules, it can be seen that this state satisfies the conditions of the Rules 1, 3, 4, 5, and 6 and hence the conflict set is $\{$Rule 1, Rule 3, Rule 4, Rule 5, Rule 6$\}$. The conflict resolution strategy we have adopted is "select and fire the first rule that does not lead to a repeated state". If we select Rule 1 and fire it, the resulting state is $(4,3)$ and we have obtained this state in Step No. 3. Similarly if we select Rule 3 and fire it the resulting state is $(0,3)$ which we have already obtained in Step No. 4. Proceeding like this, we see that that the first rule in the conflict set that does not lead to a repeated state is Rule 5 and we select it and fire it to obtain the non-repeated state $(4,2)$.

3. Conclusion

The operations specified in Table 2.5 specify a method of getting 2 liters of water in the 4-liter jug. However, this need not necessarily be the optimum solution in the sense that there may be a solution with less number of operations.

2.6.4 Production system characteristics (Classes of production systems)

Definitions

1. **Monotonic and nonmonotonic production systems**

   A monotonic production system is a production system in which the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected. A production system which is not monotonic is called a nonmonotonic production system.

2. **Partially commutative production systems**

A partially commutative production systems is a production system with the property that if the application of a particular sequence of rules transforms state x to state y then any allowable permutation of those rules also transforms state x into state y.

3. **Commutative production systems**

   A production system that is both monotonic and partially commutative is called a commutative production system.

Remarks

1. It can be shown that, depending on how the operators are chosen, the 8-puzzle and the blocks world problem are partially commutative systems.
2. Production systems that are not partially commutative are useful in which irreversible changes occur. For example, the process to produce a desired chemical compound may involve may irreversible steps.
3. Commutative production systems are useful for solving ignorable problems like the problem of proving a theorem in mathematics.
4. Nonmonotonic partially commutative production systems are useful for problems in which the order of operations is not important. For example, robot navigation is such a problem.

Table 2.6 summarises the characteristics of some of the well known production systems.

	Monotonic	Nonmonotonic
Partially commutative	Theorem proving	Robot navigation
Not partially commutative	Chemical synthesis	Bridge

Table 2.6: The four categories of production systems

2.6.5 Advantages and disadvantages of production systems

Advantages

Some of the advantages of production system in artificial intelligence are the following.

1. Provides excellent tools for structuring AI programs.
2. The system is highly modular because individual rules can be added, removed or modified independently.
3. There is separation of knowledge and control.
4. The system uses pattern directed control which is more flexible than algorithmic control.
5. Provides opportunities for heuristic control of the search.
6. Quite helpful in a real-time environment and applications.

Disadvantages

The following are some of the disadvantages.

1. There is a lot of inefficiency in production systems. For example, there may be situations where multiple rules get activated during execution and each of these rules may trigger exhaustive searches.
2. It is very difficult to analyze the flow of control within a production system.
3. There is an absence of learning due to a rule-based production system that does not store the result of the problem for future use.
4. The rules in the production system should not have any type of conflict operations. When a new rule is added to the database it should ensure that it does not have any conflict with any existing rule.

2.7 Sample questions

(a) Short answer questions

1. Describe the 8-puzzle problem.
2. State the missionaries and cannibals problem in AI.
3. What is a production system in AI?
4. Explain the blocks world problem.
5. What is a cryptarithmetic puzzle? Illustrate with an example.
6. Briefly explain the concepts of forward chaining and backward chaining in the context of a production system.
7. What is the role of a "conflict resolution strategy" in a production system?
8. What are the different classes of production systems?
9. What actions are available in the blocks world problem?
10. Illustrate with an example the concept of decomposability of a problem into smaller or easier problems.
11. Give an example of a problem in which some of the solution steps can be ignored or undone.
12. Define a monotonic production system.
13. Consider the following familiar set of production rules:

1	IF	*green*	THEN	*walk*
2	IF	*red*	THEN	*wait*
3	IF	*green* AND *blinking*	THEN	*hurry*
4	IF	*red* OR *green*	THEN	*traffic light works*

Which of the above rules will be put into a conflict set by the system if the working memory contains two facts: *green, blinking*? Apply some conflict resolution strategy to fire a rule.

(a) Long answer questions

1. Explain the components of a problem in AI.
2. State the missionaries and cannibals problem and describe its various components as a problem in AI.
3. State the monkey and banana problem and describe its various components as a problem in AI.
4. Briefly explain the characteristics of a problem in AI.
5. Explain the water jugs problem and give the system of production rules for the problem.
6. Describe a production system in AI. What are the merits and demerits of production systems?
7. Define a production rule system and illustrate it with an example.
8. Write a note on the water jug problem using production rules.
9. Solve the following cryptarithmetic problems:

   (a)
   ```
     A A
     B B
   + C C
   -----
   A B C
   -----
   ```
   (b)
   ```
       E A T
   + T H A T
   ---------
   A P P L E
   ---------
   ```
   (c)
   ```
     F O R T Y
         T E N
   +     T E N
   -----------
     S I X T Y
   -----------
   ```
   (d)
   ```
     C R O S S
   + R O A D S
   -----------
   D A N G E R
   -----------
   ```

10. Formulate the water jug problem as a production rule system.

Chapter 3

Blind search strategies

Search algorithms are one of the most important areas of artificial intelligence. Search is an important component of problem solving in artificial intelligence and, more generally, in computer science, engineering and operations research. Combinatorial optimization, decision analysis, game playing, learning, planning, pattern recognition, robotics and theorem proving are some of the areas in which search algorithms play a key role.

In this chapter we consider a certain class of search algorithms called blind search algorithms.

3.1 Preliminaries

In describing the various strategies discussed in this and the next chapter, we make extensive use the mathematical notions of graphs and trees. In this short section we quickly review the necessary concepts.

3.1.1 Graphs

A **graph** G is an ordered pair (V, E) where the elements of V are called **vertices** or **nodes** and the elements of E are called **edges** or **sides**. Each element of E can be thought of an arc joining two vertices in V. Sometimes the arcs may have specific directions in which case the graph is called a **directed graph**. There may be multiple edges joining two given edges. If there are more than one edge joining two vertices, the graph is called a **multigraph**. There may also be loops in graphs, a **loop** being an edge joining a vertex with itself. In the sequel, unless otherwise explicitly stated, by "graph" we shall always mean an undirected graph without loops and without multiple edges.

Graphs can be pictorially represented as in Figure 3.1. The set of vertices of the graph in the figure is $V = \{v_1, v_2, v_3, v_4, v_5\}$ and the set of edges is $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$.

3.1.2 Trees

A graph is said to be **connected** if there is a path along the edges joining any two vertices in the graph. A path which returns to the starting vertex is called a **cycle**. A **tree** is a connected graph without cycles.

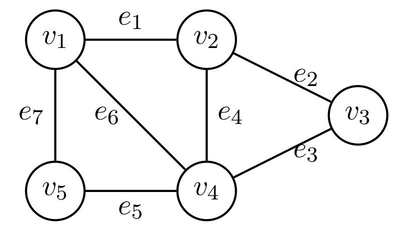


Figure 3.1: A graph without loops and without multiple edges

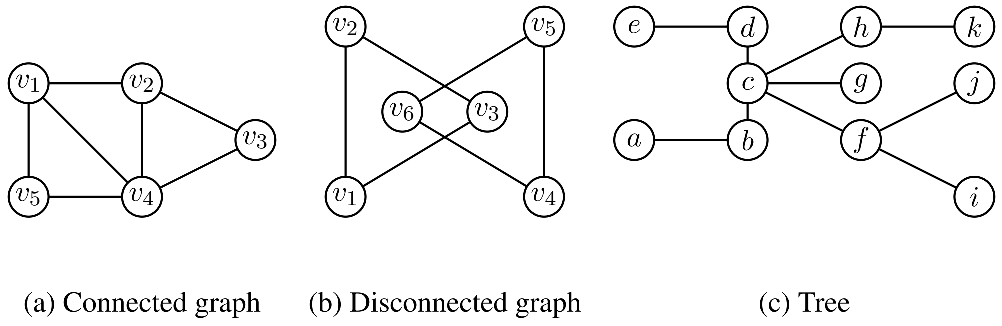


Figure 3.2: Different types of graphs

Examples

Rooted trees

In search strategies, we generally consider **rooted trees** where one vertex is designated as the **root** and all edges are directed away from the root. In pictorial representations of rooted trees, the root vertex is positioned near the top of a page and the remaining vertices are positions below the root vertex at different levels (see, for example, Figure 3.3).

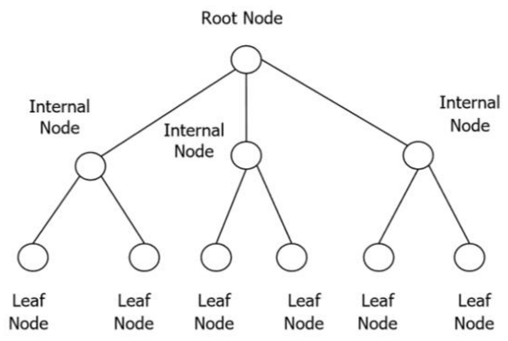


Figure 3.3: A rooted tree

3.2 Search in general

3.2.1 Solution as a sequence of actions

In Section 2.2.1 we discussed the following problem: “The problem consists of a 2×2 board with three numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The objective is to reach the goal shown in Figure 2.4 by sliding the tiles one at a time from start as in Figure 2.4."

We have also seen that the solution to the problem is the following sequence of actions:

$$\text{Up} \to \text{Left} \to \text{Down} \to \text{Right}$$

This, that the solution is a sequence of actions, is true in general. Assuming that the environment has certain properties[1], the solution to any problem (of the type considered in AI) is a fixed sequence of actions.

3.2.2 Search algorithm

The process of looking for a sequence of actions that reaches the goal is called **search**. A **search algorithm** takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

3.2.3 Search tree

Having formulated some problems, we need to solve them. Since a solution is an action sequence, search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the **branches** are actions and the **nodes** correspond to states in the state space of the problem. The **root node** of the tree corresponds to the initial state.

Example

Consider a person intending to travel from Arad to Bucharest in Romania (see Figure 3.4). Partial search trees for finding a route from Arad to Bucharest are shown in Figure 3.5.

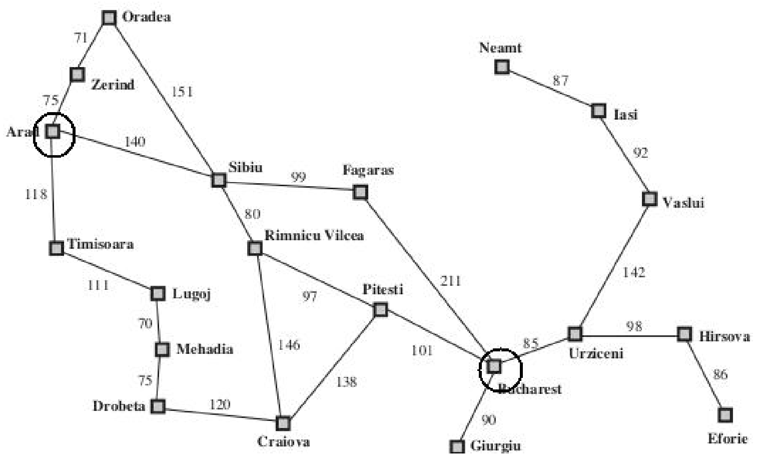


Figure 3.4: A road map of Romania

[1]The environment must be **fully observable**, **discrete**, **known** and **deterministic**.

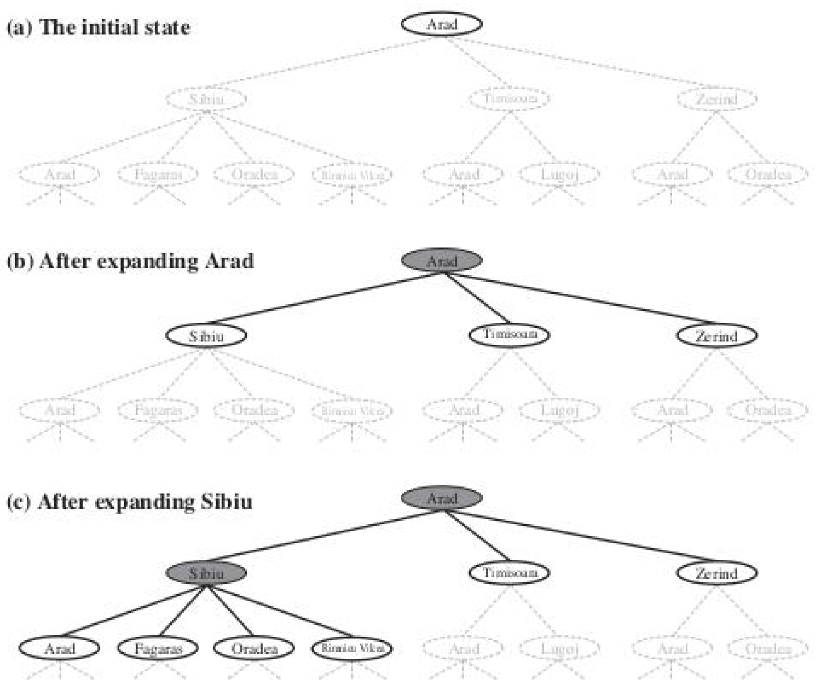


Figure 3.5: Partial search trees for finding a route from Arad to Bucharest

3.2.4 Search strategy

The first step is to test whether this is a goal state. Then we need to consider taking various actions. We do this by **expanding** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states. Now we must choose which of these possibilities to consider further. A **search strategy** specifies how to choose which state to expand next.

Figure 3.5 shows some of the early stages in the construction of the search tree for finding a path from Arid to Bucharest. In the figure, nodes that have been expanded are shaded, nodes that have been generated but not yet expanded are outlined in bold and nodes that have not yet been generated are shown in faint dashed lines.

3.3 Blind search

A **blind search** (also called an **uninformed search**) is a search that has no information about its domain. The only thing that a blind search can do is distinguish a non-goal state from a goal state. The blind search algorithms have no domain knowledge of the problem state. The only information available to blind search algorithms is the state, the successor function, the goal test and the path cost. Strategies that know whether one non-goal state is “more promising” than another are called **informed search** or **heuristic search** strategies.

Blind search is useful in situations where there may not be any information available to us. We might just be looking for an answer and won’t know we have found it until we see it. In real world, we will have millions of nodes and we will have no idea where to look for our target. It is also useful to study these blind searches as they form the basis for some of the intelligent searches.

3.3.1 Example

Figure 3.4 shows a map of Romania.

- Let us assume that we are currently at Arad and we want to get to Bucharest.
- To construct a search tree we start by designating the initial state as the root node. In the present problem, the initial state is the city of Arad and so we designate Arad as the root node (Figure 3.5 (a)).
- There are three possible actions that we can take while we are at Arad, namely, go to Sibiu, go to Timisoara and go to Zerind. These actions define three branches of the search tree emanating from the root node and ending at the nodes named as Sibiu, Timisoara and Zerind. These three branches form level 1 of the search tree (Figure 3.5 (b)). A blind search will have no preference as to which node it should explore first.
- Suppose we arbitrarily move to Sibiu. Then the possible actions are: go to Arad, go to Rimnicu Vicea, go to Fagaras and go to Oradea. These actions produce four branches from the node Sibiu (Figure 3.5 (c)).
- The process is continued.

Remarks

The search tree shown in Figure 3.5(c) includes the path from Arad to Sibiu and back to Arad again! We say that Arad is a **repeated state** in the search tree, generated in this case by a **loopy path**. Considering such loopy paths means that the complete search tree for Romania is infinite because there is no limit to how often one can traverse a loop. On the other hand, the map has only 20 states. However, there are mechanisms which ensure that such loopy paths are generated while creating a search tree.

3.4 Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Remark

Figure 3.6 shows the order in which the nodes in a tree are explored while applying the breadth-first search algorithm.

3.4.1 Algorithm

The algorithm returns the goal state or failure.

1. Create a variable called *NODE-LIST* and set it to the initial state.
2. Until a goal state is found or *NODE-LIST* is empty do:

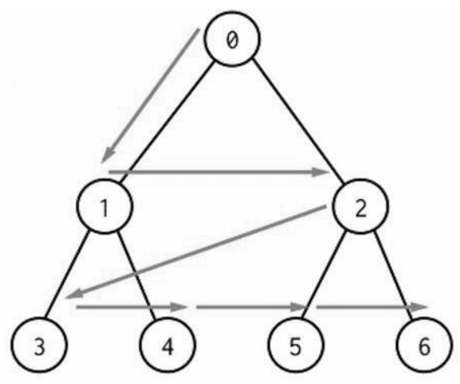


Figure 3.6: The order in which the nodes are explored while applying the BFS algorithm: $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$

(a) Remove the first element from *NODE-LIST* and call it E. If *NODE-LIST* was empty, quit and return failure.

(b) For each way that each rule can match the state described in E do:

   i. Apply the rule to generate a new state.

   ii. If the new state is a goal state, quit and return this state.

   iii. Otherwise, add the new state to the end of *NODE-LIST*.

3.4.2 Examples

Example 1

Let the search tree be as in Figure 3.7 the initial and goal states being 1 and 8 respectively. Apply the breadth-first search algorithm to find the goal state.

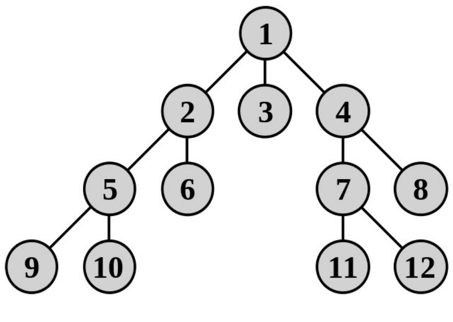


Figure 3.7: The search tree for a simple problem

Solution

The details of the various steps in finding a solution to the problem are depicted in Table 3.1. It can be seen that the nodes are visited in the following order:

$$1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8.$$

Step no.	*NODE-LIST*	Node E	Node generated from E	Comment
1	1	-	-	Initial state
2	-	1	-	-
3	-	1	2	-
4	2	1	-	-
5	2	1	3	-
6	2, 3	1	-	-
7	2, 3	1	4	-
8	2, 3, 4	1	-	-
9	3, 4,	2	-	-
10	3, 4	2	5	-
11	3, 4, 5	2	-	-
12	3, 4, 5	2	6	-
13	3, 4, 5, 6	2	-	-
13	4, 5, 6	3	-	No node generated from E
14	5, 6	4	-	-
15	5, 6	4	7	-
16	5, 6, 7	4	-	-
17	5, 6, 7	4	8	Goal state. Return 8 and quit.

Table 3.1: Table showing the details of the various steps in the solution to the problem specified in Figure 3.7

Example 2

Consider the water jug problem discussed in Section 2.6.3 and the associated production rules given Table 2.1. We now construct the search tree using the breadth-first search algorithm.

Stage 1. Construct a tree with the initial state $(0, 0)$ as its root.

$$\boxed{(0, 0)}$$

Figure 3.8: Stage 1 in the construction of the breadth-first search tree of the water jug problem

Stage 2. Construct all the children of the root by applying each of the applicable rules to the initial state. Then we get a tree as in Figure 3.9.

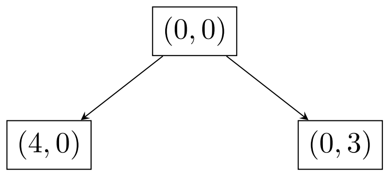


Figure 3.9: Stage 2 in the construction of the breadth-first search tree of the water jug problem

Stage 3. Now for each leaf node in Figure 3.9, generate all its successors by applying all the rules that are appropriate. The tree at this point is as shown in Figure 3.10.

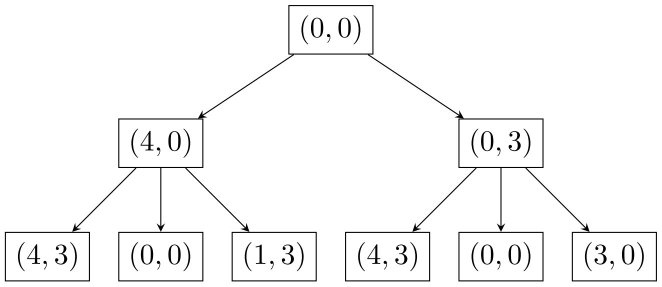


Figure 3.10: Stage 3 in the construction of the breadth-first the search tree of the water jug problem

Stage 4. The process is continued until we get a node representing the goal state, namely, $(2, 0)$.

3.5 Depth-first search

3.5.1 Algorithm

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:
   (a) Generate a successor E of the initial state. If there are no more successors, signal failure.
   (b) Call depth-first search with E as the initial state.
   (c) If success is returned, signal success. Otherwise continue in this loop.

Remark

Figure 3.11 shows the order in which the nodes in a tree are explored while applying the depth-first search algorithm.

3.5.2 Examples

Example 1

Let the search tree be as in Figure 3.12 the initial and goal states being 1 and 7 respectively. Apply the depth-first search algorithm to find the goal state.

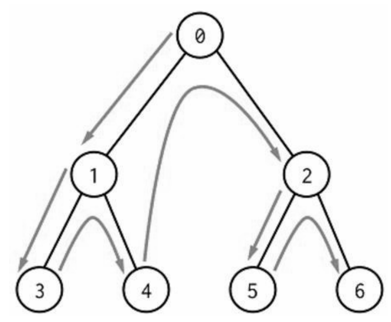


Figure 3.11: The order in which the nodes are explored while applying the DFS algorithm: $0 \to 1 \to 3 \to 4 \to 2 \to 5 \to 6$

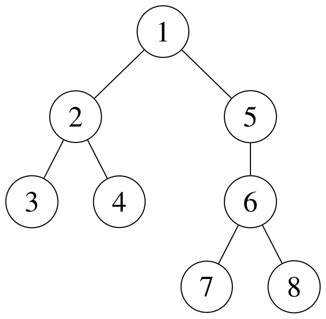


Figure 3.12: A graph

Solution

We denote by DFS(x) an invocation of the depth-first search algorithm with x as the initial vertex. The various steps in the implementation of the algorithm are shown in Figure 3.13. Note that the various nodes are visited in the following order:

$$1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7$$

```
Call DFS(1)
1. Node 1 is not a goal state.
2. Do until successor or failure is signaled.
    (a) Choose 2 as successor of node 1: $E = 2$.
    (b) Call DFS(2).
        1. Node 2 is not a goal state.
        2. Do until success or failure is signaled.
            (a) Choose 3 as successor of 2: $E = 3$
            (b) Call DFS(3).
                1. Node 3 is not a goal state.
                2. Do until successor or failure is signaled.
                    (a) Node 3 has no successor. Signal failure.
                DFS(3) ends.
            (a) Choose 4 as successor of 2: $E = 4$
            (b) Call DFS(4).
                1. Node 4 is not a goal state.
                2. Do until successor or failure is signaled.
                    (a) Node 4 has no successor. Signal failure.
                **DFS**$(4)$ ends.
        **DFS**$(2)$ ends.
    (a) Choose 5 as successor of node 1: $E = 5$
    (b) **Call DFS**$(5)$.
        1. Node 5 is not a goal state.
        2. Do until success or failure is signaled.
            (a) Choose 6 as successor of 5: $E = 6$
            (b) **Call DFS**$(6)$.
                1. Node 6 is not a goal state.
                2. Do until successor or failure is signaled.
                    (a) Choose 7 as a successor of 6: $E = 7$
                    (b) **Call DFS**$(7)$.
                        1. Node 7 is a goal state. Quit and return success.
                    **DFS**$(7)$ ends.
                **DFS**$(6)$ ends.
        **DFS**$(5)$ ends.
**DFS**$(1)$ ends.
```

Figure 3.13: Finding a path from node 1 to node 7 in Figure 3.12 using the DFS

Example 2

Consider the water jug problem discussed in Section 2.6.3 and the associated production rules given Table 2.1. We now construct the search tree using the depth-first search algorithm.

Stage 1. Construct a tree with the initial state $(0, 0)$ as its root. The initial state is not a good state.

$$\boxed{(0,0)}$$

Figure 3.14: Stage 1 in the construction of the depth-first search tree of the water jug problem

Stage 2. We generate a successor E for $(0, 0)$. We choose the successor obtained by the application of Rule 1 in the production rules. Then we get a tree as in Figure 3.15.

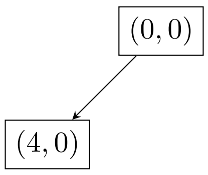


Figure 3.15: Stage 2 in the construction of the depth-first search tree of the water jug problem

Stage 3. Now for the leaf node in Figure 3.15, generate a successor by applying an applicable rule. The tree at this point is as shown in Figure 3.16.

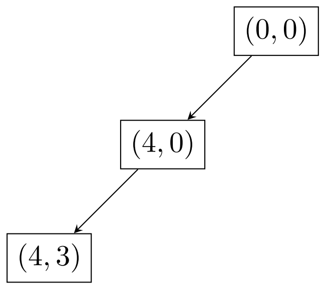


Figure 3.16: Stage 3 in the construction of the depth-first search tree of the water jug problem

Stage 4. The process is continued until we get a node representing the goal state, namely, $(2, 0)$.

3.6 Breadth-first search *vs.* depth-first search

The Table 3.2 summarises the main advantages and disadvantages of breadth-first search and depth-first search.

Table 3.2: Breadth-first search *vs.* depth-first search

Sl No	Breadth-first	Depth-first
1	Requires more memory because all of the tree that has so far been generated must be stored.	Requires less memory because only the nodes in the current path are stored.
2	All parts of the search tree must be examined at level n before any node on level $n + 1$ can be examined.	May find a solution without examining much of the search space. It stops when one solution is found.
3	Will not get trapped exploring a blind alley.	May follow an unfruitful path for a long time, perhaps for ever.
4	If there is a solution, Guaranteed to find a solution if there exists one. If there are multiple solutions then a minimal solution (that is, a solution that takes a minimum number of steps) will be found.	May find a long path to a solution in one part of the tree when a shorter path exists in some other unexplored path of the tree.
5	Breadth-first search uses queue data structure.	Depth-first search uses stack data structure.
6	More suitable for searching vertices which are closer to the given source.	More suitable when there are solutions away from source.

3.7 Depth-first iterative deepening (DFID) search

Iterative deepening combines the benefits of depth-first and breadth-first searches. Like breadth-first search, the iterative deepening search is guaranteed to find a solution if one exists. In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

3.7.1 Algorithm

1. Set SEARCH-DEPTH = 1.
2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution is found, then return it.
3. Otherwise, increment SEARCH-DEPTH by 1 and go to step 2.

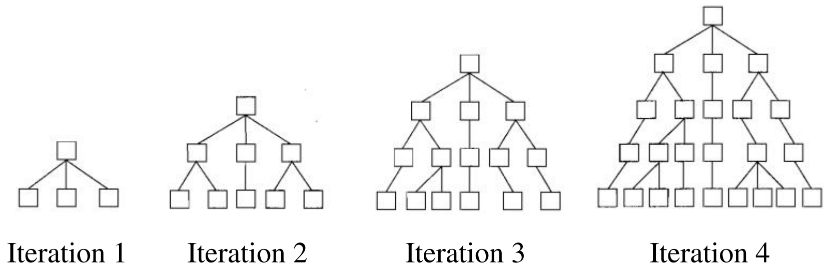


Figure 3.17: Iterative deepening

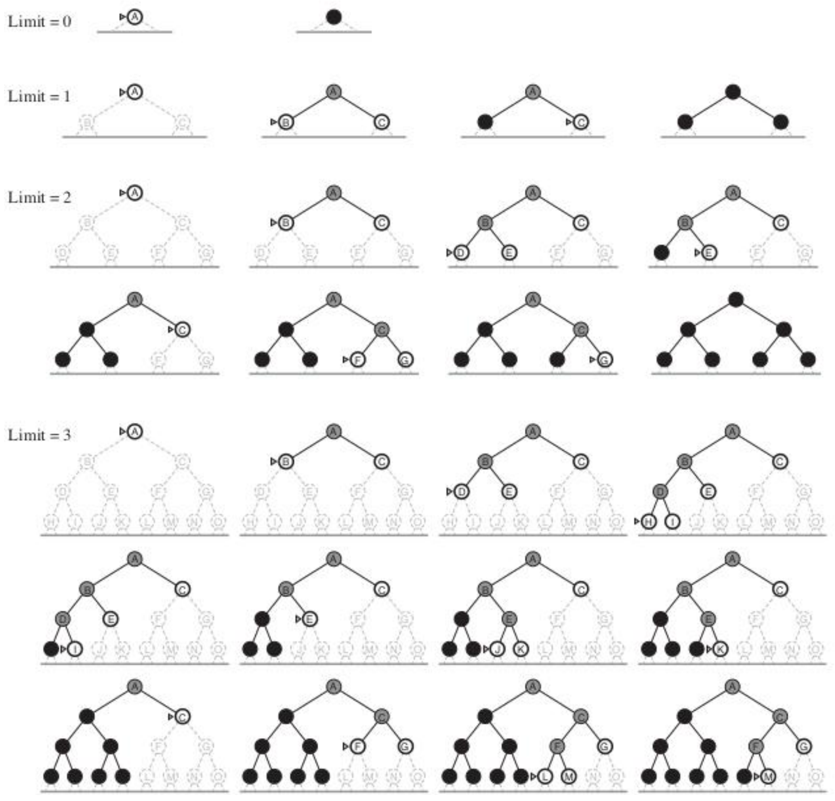


Figure 3.18: The order in which the nodes are explored in iterative deepening

3.7.2 Advantages and disadvantages

1. The DFID will always find the shortest solution path to the goal state if it exists.
2. The maximum amount of memory used is proportional to the number of nodes in the solution path.
3. All iterations but the final one are wasted.
4. The DFID is slower than the depth-first search only by a constant factor.
5. DFID is the optimal algorithm in terms of space and time for uninformed search.

3.8 Blind search in graphs

The algorithms we have discussed in the previous sections of this chapter can only be applied for searching the nodes of a tree. Since a tree is a graph with some special properties, these algorithms cannot be directly applied to search the nodes of a graph. However, the breadth-first and depth-first search algorithms can be modified to create algorithms for searching the nodes in a graph.

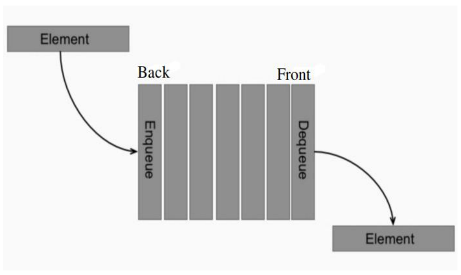
Suppose we are searching a graph G. The set of all leaf nodes available for expansion at any given point is called the **frontier**. (Many authors call it the **open list**.)

3.8.1 Graph-search algorithm

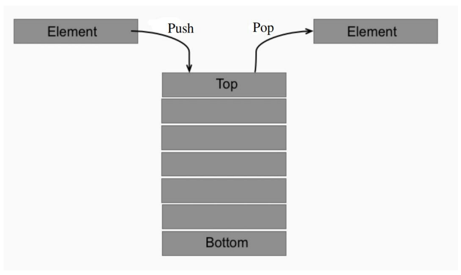
1. Initialize the frontier using the initial state of problem.
2. Initialize the explored set to be empty.
3. Do the following until success or failure is returned:
   (a) If the frontier is empty then return failure.
   (b) Choose a leaf node and remove it from the frontier.
   (c) If the node contains a goal state then return the corresponding solution.
   (d) Add the node to the explored set.
   (e) Expand the chosen node, adding the resulting nodes to the frontier only if not in the frontier or explored set.

3.8.2 Breadth-first and depth-first search in graphs

In the graph-search algorithm given in Section 3.8.1, we may use different data structures to store the nodes in the frontier. If we use the queue data structure (first-in first-out (FIFO) queue) the resulting algorithm is the breadth-first search algorithm where as if we use the stack data structure (last-in first-out (LIFO) queue) the resulting algorithm is the depth-first search algorithm.



(a) Queue data structure



(b) Stack data structure

Figure 3.19: Queue and stack data structures

3.8.3 Example

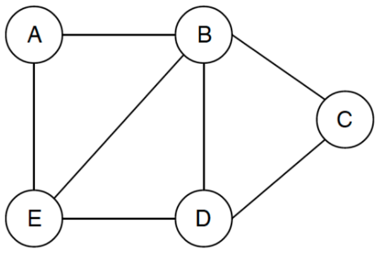


Figure 3.20: A graph

Consider the graph shown in Figure 3.20. Let us explore the graph starting from the node A. If we use the queue data structure to store the frontier set, then one of the orders in which the nodes are explored is shown in Table 3.3. A different order in which the nodes are explored is shown in Table 3.4.

Iteration No.	Frontier set (Queue (FIFO))					Explored set
Initial					A	∅
Iteration 1				E	B	{ A }
Iteration 2			D	C	E	{A, B}
Iteration 3				D	C	{A, B, E}
Iteration 4					D	{A, B, E, C}
Iteration 5						{A, B, E, C, D}

Table 3.3: Frontier set using queue data structure, and explored set

Iteration No.	Frontier set (Queue (FIFO))					Explored set
Initial					A	∅
Iteration 1				B	E	{ A }
Iteration 2				D	B	{A, E}
Iteration 3				C	D	{A, E, B}
Iteration 4					C	{A, E, B, E, D}
Iteration 5						{A, E, B, D, C}

Table 3.4: Frontier set using queue data structure, and explored set

If we use the stack data structure (LIFO) to store the frontier set, then one of the orders in which the nodes are explored is shown in Table 3.5.

Iteration No.	Frontier set (Stack (LIFO))	Explored set
Initial	A	∅
Iteration 1	E B	{ A }
Iteration 2	D B	{ A, E }
Iteration 3	C B	{ A, E, D }
Iteration 4	B	{ A, E, D, C }
Iteration 5		{ A, E, D, C, B}

Table 3.5: Frontier set using stack data structure, and explored set

---

3.9 Sample questions

(a) Short answer questions

1. Define search tree and illustrate with an example.

2. What is meant by blind search? Give examples of blind-search methods.

3. Compare and contrast unformed and uninformed search methods.

4. List some of the uninformed search methods.

5. State the advantages of the breadth-first search method.

6. Will the breadth-first search method always find the optimal solution? Why? Illustrate with an example.

7. Compare and contras BFS and DFS methods.

8. Give an outline of the DFID search algorithm.

9. Let the search tree be as in Figure 3.21. If the BFS algorithm is used, what is the order in which the various nodes are explored starting from node 0.

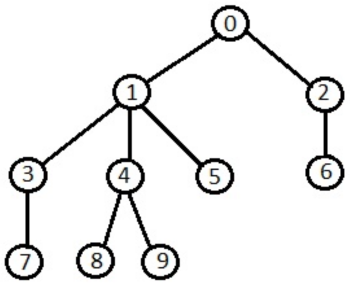


Figure 3.21: A search tree

10. Let the search tree be as in Figure 3.21. If the DFS algorithm is used, what is the order in which the various nodes are explored starting from node 0.

11. Consider the following graph:

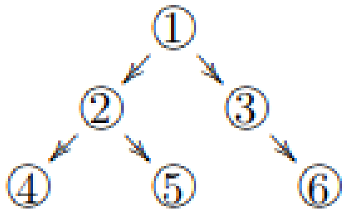


Figure 3.22: A search tree

Starting at root node 1, give the order in which the nodes will be visited by the breadth-first and depth-first algorithms.

12. If the graph in Figure 3.23 is explored using the breadth first algorithm starting from node Q, what is the order in which the nodes will be explored? what will be the order if it is explored using the depth first algorithm?

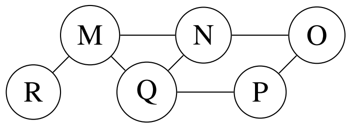


Figure 3.23: A graph

13. If the graph in Figure 3.24 is explored using the breadth first algorithm starting from node 0, what is the order in which the nodes will be explored? what will be the order if it is explored using the depth first algorithm?

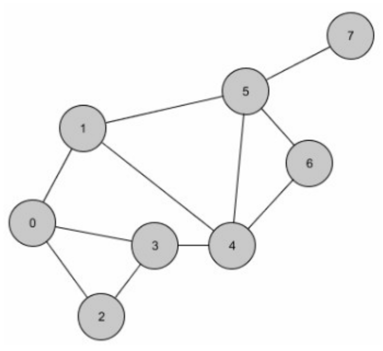


Figure 3.24: A graph

(b) Long answer questions

1. Distinguish breadth-first and depth-first search methods.
2. Briefly describe the breadth-first and the depth-first search for trees.
3. Write the breadth-first search algorithm and illustrate it with an example.
4. Write the depth-first search algorithm and illustrate it with an example.
5. Explain a graph-search algorithm.
6. Explain various uninformed search strategies.

Chapter 4

Heuristic search strategies

In computer science and artificial intelligence, a **heuristic**[1] is a technique designed for solving a problem more quickly when classic (that is, well known) methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.

The objective of a heuristic is to produce a solution in a reasonable time that is good enough for solving the problem at hand. This solution may not be the best of all the actual solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time.

In this chapter we consider informed (heuristic) search strategies, that is, strategies that use problem-specific knowledge beyond the definition of the problem itself, strategies that can find solutions more efficiently than an uninformed strategy.

4.1 Heuristic function

A **heuristic function**, also called simply a **heuristic**, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow.

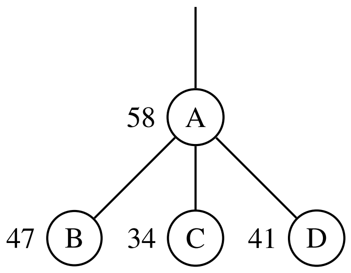


Figure 4.1: Values of a heuristic function

For example, let us assume that we are at node A in the search tree of some search problem (see Figure 4.1). Let us also assume that there are three different branches, say

[1]The word heuristics comes from Greek word *heuriskein* which means "to find, to discover".

AB, AC and AD, along which we can continue the search for the goal node. We have to decide which one of these nodes is to be selected. From the circumstances of the problem let it be possible to obtain an *estimate* of the cheapest path from the nodes to the goal node. Let these estimates be as shown alongside each of the nodes in the figure. For example, the estimate of the cheapest cost from node C to the goal node is 34.

One way of choosing the next node is to chose that node for which the estimated cost of the cheapest path to goal is least. As per this criterion, we choose to proceed along the branch AC. (Of, course, there are other ways in which the estimates can be used to choose the next branch.) These estimated costs define a heuristic function for the problem. The methods of constructing such functions from the circumstances of a problem are illustrated in Section 4.1.4.

4.1.1 Definition

The **heuristic function** for a search problem is a function $h(n)$ defined as follows:

$$h(n) = \text{estimated cost of the cheapest path from the state at node } n \text{ to a goal state.}$$

Remark

Unless otherwise specified, it will be assumed that a heuristic function $h(n)$ has the following properties:

1. $h(n) \geq 0$ for all nodes n.
2. If n is the goal node, then $h(n) = 0$.

4.1.2 Admissible heuristic functions

A heuristic function is said to be **admissible** if it never overestimates the cost of reaching the goal, that is, the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.

Let $h(n)$ be a heuristic function for a search problem. $h(n)$ is the estimated cost of reaching the goal from the state n. Let $h^*(n)$ be the optimal cost to reach a goal from n. We say that $h(n)$ is admissible if

$$h(n) \leq h^*(n) \quad \text{for all } n$$

4.1.3 Remarks

1. There are no limitations on the function $h(n)$. Any function of our choice is acceptable. As an extreme case, the function $h(n)$ defined by

   $$h(n) = c \text{ for all } n$$

   where c is some constant can also be taken as a heuristic function. The heuristic function defined by

   $$h(n) = 0 \text{ for all } n$$

   is called the **trivial heuristic** function.

2. The heuristic function for a search problem depends on the problem in the sense that the definition of the function would make use of information that are specific to the circumstances of the problem. Because of this, there is no one method for constructing heuristic functions that are applicable to all problems.

3. However, there is a standard way to construct a heuristic function: It is to find a solution to a simpler problem, which is one with fewer constraints. A problem with fewer constraints is often easier to solve (and sometimes trivial to solve). An optimal solution to the simpler problem cannot have a higher cost than an optimal solution to the full problem because any solution to the full problem is a solution to the simpler problem.

4.1.4 Examples

In the following subsections we present several examples of heuristic functions. In each of these examples, the heuristic function is defined using knowledge about the circumstances of the particular problem. Also, these functions are solutions of problems obtained by relaxing some of the constraints in the problem. Moreover, it can be seen that all the presented heuristic functions are admissible.

1. $h(n)$ in path search in maps

Consider the problem of finding the shortest path from Arid to Bucharest in Romania (see Figure 3.4).

In the problem, the traveler is required to travel by roads shown in the map. Let us consider a problem obtained by relaxing this condition. Let the traveler be allowed to travel by any method. In this relaxed problem, obviously, the shortest path from from a city n to Bucharest is the air distance between n and Bucharest. We may use this solution as the definition of a heuristic function for the original problem.

If n is any of the cities in Romania, we may define the heuristic function $h(n)$ as follows:

$$h(n) = \text{The straight line distance (air distance) from } n \text{ to Bucharest.}$$

Table 4.1 shows the values of $h(n)$. Note that $h(n) \geq 0$ for all n and $h(\text{Bucharest}) = 0$.

Now $h^*(n)$ is the the length of the shortest road-path from n to Bucharest. Since, the air distance never exceeds the road distance we have

$$h(n) \leq h^*(n) \quad \text{for all } n.$$

Thus, $h(n)$ as defined above is an admissible heuristic function for the problem

2. $h(n)$ in missionaries and cannibals problem

Recall the description of the missionaries and cannibals problem given in Section 2.3.3.

To define a heuristic function for the missionaries and cannibals problem, let us consider a simpler problem wherein we ignore the condition that at any location (on either bank of the river or in the boat) the number of cannibals should not exceed the number of missionaries. Let k be the number of persons at the initial side of the river. If $k = 2$ we need only 1 river crossing to take both persons to the other side of the river. If $k = 3$ we need 3 river

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

Table 4.1: Heuristic function for Romania travel problem

crossings, if $k = 4$ also we need only 3 river crossings to take all people to the other side of the river. If $k = 5$ or $k = 6$ we need 5 river crossings to take all people to the other side of the river. See below for details of the crossings in the case when $k = 6$.

Crossing 1	Take 2 to destination side
Crossing 2	Return to initial side
Crossing 3	Take 2 to destination side
Crossing 4	Return to initial side
Crossing 5	Take 2 to destination side

In general, in the simplified scenario, it takes k or $k - 1$ river crossings depending on whether k is odd or even to take k persons to the other side of the river.

We may choose the following function as a heuristic function for the missionaries and cannibals problem. If (x, y, z) is a state of the problem (see Section 2.3.3), we define

$$h(x,y,z) = \begin{cases} 0 & \text{if } x+y=0 \\ x+y & \text{if } x+y \text{ is odd} \\ x+y-1 & \text{if } x+y>0 \text{ and } x+y \text{ is even} \end{cases}$$

Note that $x + y$ is the number of people on the initial side of the river.

Note that $h(n) \geq 0$ for all n and $h(\text{Goal state}) = 0$. Also, it is obvious that this is an admissible heuristic function for the problem.

Another admissible heuristic function for the problem is given below:

$$h(n) = \tfrac{1}{2}(x + y).$$

3. $h(n)$ in the 8-puzzle problem

There are several restrictions, though not explicitly stated, on the movements of the tiles in the 8-puzzle problem like the following (see Section 2.3.2):

1. A tile can be moved only to a nearby vacant slot.
2. Tiles can only be moved in the horizontal or vertical direction.
3. One tile cannot slide over on another tile.

We can get different admissible heuristic functions for the 8-puzzle problem by solving simpler problems by relaxing or removing some of these restrictions. We define two different heuristic functions $h_1(n)$ and $h_2(n)$ for the 8-puzzle problem. $h_1(n)$ is obtained by solving the problem without all the three restrictions given above. $h_2(n)$ is obtained by solving the problem obtained by relaxing first restriction to the restriction that "a tile may be moved to any nearby slot and deleting the third restriction".

- **Hamming distance[2] heuristic**

  $h_1(n) =$ Number of tiles out of position.

  Example

5		8
4	2	1
7	3	6

(a) Given state (n)

1	2	3
4	5	6
7	8	

(b) Goal state

Figure 4.2: An initial state and a goal state of 8-puzzle

  Consider the state shown in Figure 4.2(a) and compare the positions of the tiles with their positions in the goal state shown in Figure 4.2(b). It can be seen that only two tiles, namely 4 and 7 (ignoring the empty cell), are in their positions. Hence, for the state we have

$$h_1(n) = 6.$$

- **Manhattan distance heuristic**

  We first consider the Manhattan distance between two cells in a grid of cells. Figure 4.3 illustrates the concept using 8×8 grid.

[2]The terminology comes from information theory. In information theory, the Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. In other words, it measures the minimum number of substitutions required to change one string into the other. For example, the Hamming distance between the strings "1011011010" and "1010011110" is 2.

(C) 0	1	2	(D) 3				
					(B) 7		
					6		
					5		(E) 0
	(A) 0	1	2	3	4		1
							2
							(F) 3

Figure 4.3: In the 8×8 grid, the Manhattan distance between the cells (A) and (B) is 7, between (C) and (D) is 3 and between (E) and (F) is 3.

The Manhattan distance heuristic function is defined as follows:

$h_2(n) =$ Sum of the Manhattan distances of every numbered tile to its goal position.

Example

For the initial state n shown in Figure 4.2 we have

$$\begin{aligned} h_2(n) =& \text{Manhattan distance of 1 from goal position} + \\ & \text{Manhattan distance of 2 from goal position} + \\ & \cdots + \\ & \text{Manhattan distance of 2 from goal position} \\ =& 3 + 1 + 3 + 0 + 2 + 1 + 0 + 3 \\ =& 13. \end{aligned}$$

4. $h(n)$ in blocks world problem

We define a heuristic function for the blocks world problem (see Section 2.3.5) as follows:

$h(n) =$ Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

This is an example of a heuristic function not satisfying the conditions $h(n) \geq 0$ and $h(\text{Goal node}) = 0$.

Example

To illustrate the idea, consider a blocks world problem with the initial and goal states as in Figure 4.4. In the initial state, only block numbered "1" is resting on the thing it is supposed to be resting on. All other blocks are sitting on the wrong things. Hence we have:

$$h(\text{Initial state}) = +1 - 1 - 1 - 1 - 1 - 1 = -4.$$

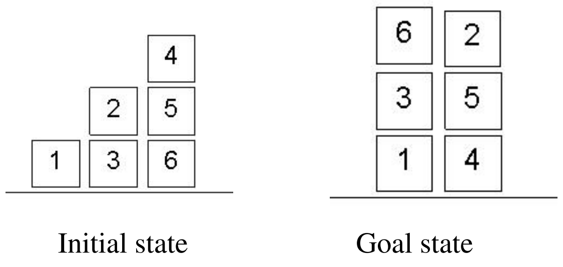


Figure 4.4: A blocks world problem

Since in the goal state every block is sitting on the right thing, we have

$$h(\text{Goal state}) = 1 + 1 + 1 + 1 + 1 + 1 = 6 \neq 0.$$

5. $h(n)$ in water jug problem

For the water jug problem given in Section 2.6.3, we may choose the following as a heuristic function. For a state specified by the ordered pair (x, y), we define

$$h(x, y) = |x - 2| + |y - 2|.$$

This function satisfies the condition $h(n) \geq 0$ for all n. It does not satisfy the condition $h(\text{Goal state}) = 0$, because as a goal state is specified by a pair of the form $(2, y)$, we have

$$h(\text{Goal state}) = h(2, y) = |y - 2|.$$

4.2 Best first search

If we consider searching as a form of traversal in a graph, an uninformed search algorithm blindly traverses to the next node in a given manner without considering the cost associated with that step. An informed search, on the other hand, uses an **evaluation function** $f(n)$ to decide which among the various available nodes is the most promising (or "best") before traversing to that node. An algorithm that implements this approach is known as a best first search algorithm. It should be emphasised that, in general, the evaluation function is not the same as the heuristic function.

There are different best first search algorithms depending on what evaluation function is used to determine the most promising node. One commonly used evaluation function is the heuristic function. The corresponding algorithm is sometimes called the the greedy best first search algorithm. Another variant known as the $A^\star$ best search algorithm uses the sum of the heuristic function and the function that specifies the cost of traversing the path from the start node to a particular node as the evaluation function.

4.3 Greedy best first search

4.3.1 The algorithm

The greedy best first search algorithm uses the following notations and conventions:

OPEN	:	A list which keeps track of the current "immediate" nodes available for traversal.
CLOSED	:	A list that keeps track of the nodes already traversed.
$h(n)$	:	The heuristic function used as the evaluation function $f(n)$, that is, $f(n) = h(n)$.

Algorithm

1. Create two empty lists: OPEN and CLOSED.
2. Start from the initial node (say N) and put it in the ordered OPEN list.
3. Repeat the next steps until goal node is reached.
   (a) If OPEN list is empty, then exit the loop returning FALSE.
   (b) Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also record the information of the parent node of N.
   (c) If N is a goal node, then move the node to the CLOSED list and exit the loop returning TRUE. The solution can be found by backtracking the path.
   (d) If N is not the goal node, generate the immediate successors of N, that is, the immediate next nodes linked to N and add all those to the OPEN list.
   (e) Reorder the nodes in the OPEN list in ascending order according to the values of the evaluation function $f(n)$ at the nodes.

4.3.2 Examples

Example 1

Use the greedy best first search algorithm to find a path from A to C in the graph shown in Figure 4.5.

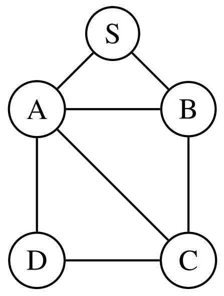


Figure 4.5: An example graph

Use the following heuristic function:

Node n	S	A	B	C	D
$h(n)$	6	2	3	0	2

Solution

Table 4.2 shows the various stages in the execution of the algorithm with details of the current node, children of the current node, the contents of the OPEN list and the contents of the CLOSED list.

Table 4.2: Various stages in the execution of the greedy best first search algorithm for the graph in Figure 4.5

Current (N)	Children of N	OPEN	CLOSED (node, parent)	Remarks
S	–	–	–	Initial state
–	–	S	–	Initial state added to OPEN.
S	–	–	(S, –)	S made current node and added to CLOSED.
S	A, B	–	(S, –)	Children of S
S	–	A, B	(S, –)	Add children to OPEN and order according to values of heuristic function.
A	–	B	(S, –), (A, S)	A made current node and added to CLOSED.
A	B, D	B	(S, –) (A, S)	Children of A
A	–	D, B	(S, –) (A, S)	Add children to OPEN and order according to values of heuristic function.
D	–	B	(S, –), (A, S), (D, A)	D made current node and added to CLOSED.
D	A, C	B	(S, –), (A, S), (D, A)	Children of D
D	–	C, B	(S, –), (A, S), (D, A)	Add children to OPEN and order according to values of heuristic function.

C	–	B	(S, –), (A, S), (D, A), (C, D)	C made current node and added to CLOSED.
C	–	–	–	C is goal state.

From the list of nodes in the CLOSED list at the stage when the goal state has been reached we see that the path from S to C is

$$\text{“C} \leftarrow \text{D} \leftarrow \text{A} \leftarrow \text{S”,}$$

or equivalently,

$$\text{“S} \rightarrow \text{A} \rightarrow \text{D} \rightarrow C\text{”.}$$

Example 2

Using the greedy best first search algorithm, find an optimal path from A to F in the search graph shown in Figure 4.6. In the figure, the numbers written alongside the nodes are the values of the heuristic function and the numbers written alongside the edges are the costs associated with the edges.

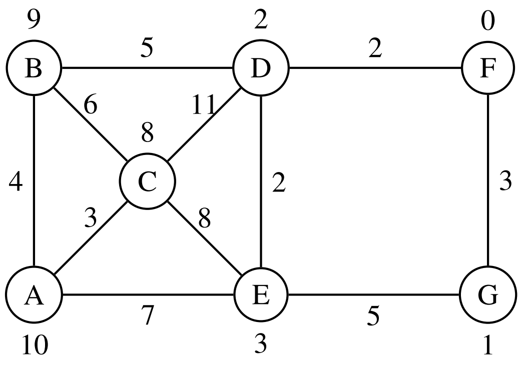


Figure 4.6: An example graph

Solution

Table 4.3 shows the various stages in the execution of the algorithm with details of the current node, children of the current node, the contents of the OPEN list and the contents of the CLOSED list.

Table 4.3: Various stages in the execution of the greedy best first search algorithm for the search tree in Figure 4.6

Current (N)	Children of N	OPEN	CLOSED (node, parent)	Remarks
A	–	–	–	Initial state

–	–	A	–	Initial state added to OPEN.
A	–	–	(A, –)	A made current and added to CLOSED.
A	B, C, E	–	(A, –)	Children of A
A	–	E, C, B	(A, –)	Add children to OPEN and order according to values of heuristic function.
E	–	C, B	(A, –), (E, A)	E made current and added to CLOSED
E	C, D, G	C, B	(A, –), (E, A)	Children of E
E	–	G, D, C, B	(A, –), (E, A)	Add children to OPEN and order according to values of heuristic function.
G	–	D, C, B	(A, –), (E, A), (G, E)	G made current and added to CLOSED.
G	F	D, C, B	(A, –), (E, A), (G, E)	E is child of G, but is in CLOSED and so not considered.
G	–	F, D, C, B	(A, –), (E, A), (G, E)	Add children to OPEN and order according to values of heuristic function.
F	–	D, C, B	(A, –), (E, A), (G, E), (F, G)	F made current and added to CLOSED.
F	–	–	–	Goal state reached.

From the list of nodes in the CLOSED list at the stage when the goal state has been reached we see that the path from A to F is “F ← G ← E ← A”, or equivalently, “A → E → G → F”.

Example 3

Using the greedy best first search algorithm find the optimal path from Arid to Bucharest in Figure 4.7 using the heuristic function defined by Table 4.1.

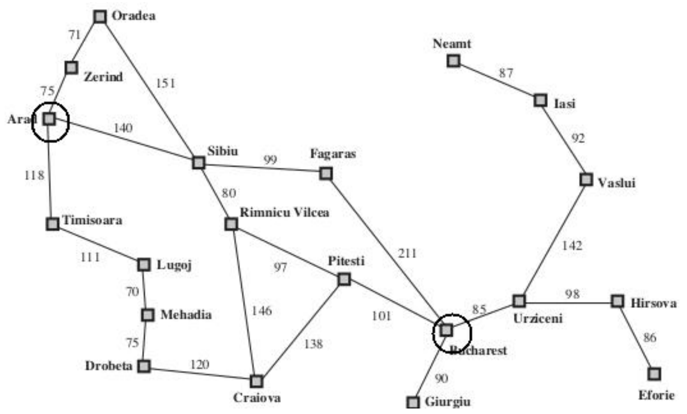


Figure 4.7: A road map of Romania

Solution

Current	Children	OPEN	CLOSED (node, parent)
Arad-366	-	-	
-	-	Arad-366	-
Arad-366	-	-	(Arad-366,-)
Arad-366	Timisoara-329, Sibiu-253, Zerind-374	-	(Arad-366,-)
Arad-366	-	Timisoara-329, Sibiu-253, Zerind-374	(Arad-366,-)
Arad-366	-	Sibiu-253, Timisoara-329, Zerind-374	(Arad-366,-)
Sibiu-253	-	Timisoara-329, Zerind-374	(Arad-366,-), (Sibiu-253, Arad-366)
Sibiu-253	Oradea-380, Fagaras-176, Rimnicu Vicea-193	Timisoara-329, Zerind-374	(Arad-366,-), (Sibiu-253, Arad-366)
Sibiu-253	-	Oradea-380, Fagaras-176, Rimnicu Vicea-193, Timisoara-329, Zerind-374	(Arad-366,-), (Sibiu-253, Arad-366)

Sibiu-253	-	Fagaras-176, Rimnicu Vicea-193, Timisoara-329, Zerind-374, Oradea-380	(Arad-366,-), (Sibiu-253, Arad-366)
Fagaras-176	-	Rimnicu Vicea-193, Timisoara-329, Zerind-374, Oradea-380	(Arad-366,-), (Sibiu-253, Arad-366), (Fagaras-176, Sibiu-253)
Fagaras-176	Bucharest-0	Rimnicu Vicea-193, Timisoara-329, Zerind-374, Oradea-380	(Arad-366,-), (Sibiu-253, Arad-366), (Fagaras-176, Sibiu-253)
Fagaras-176	-	Bucharest-0, Rimnicu Vicea-193, Timisoara-329, Zerind-374, Oradea-380	(Arad-366,-), (Sibiu-253, Arad-366), (Fagaras-176, Sibiu-253)
Fagaras-176	-	Bucharest-0, Rimnicu Vicea-193, Timisoara-329, Zerind-374, Oradea-380	(Arad-366,-), (Sibiu-253, Arad-366), (Fagaras-176, Sibiu-253)
Bucharest-0	-	Rimnicu Vicea-193, Timisoara-329, Zerind-374, Oradea-380	(Arad-366,-), (Sibiu-253, Arad-366), (Fagaras-176, Sibiu-253), (Bucharest-0, Fagaras-176)
Goal node			

The greedy best first search algorithm yields the following path from Arad to Bucharest:

$$\text{Arad} \to \text{Sibiu} \to \text{Fagaras} \to \text{Bucharest}$$

Remarks

It may be noted that the above solution is not the optimal path from Arad to Bucharest. The cost of the above path is

$$140 + 99 + 211 = 450$$

where as there is path with lower cost, namely,

$$\text{Arad} \to \text{Sibiu} \to \text{Rmnicu Vilcea} \to \text{Pitest} \to \text{Bucharest}$$

whose total cost is only

$$140 + 80 + 97 + 101 = 418.$$

4.4 A^* best first search

Let $h(n)$ be a heuristic function associated with the search problem and $g(n)$ be the cost of traversing the path from the start node to the node n. In the A^* best first search[3], we use the following evaluation function:

$$f(n) = g(n) + h(n).$$

4.4.1 Examples

The following examples illustrate the basic idea of the A^* algorithm.

Example 1

Use the A^* best first search algorithm to find a path from A to C in the graph shown in Figure 4.8.

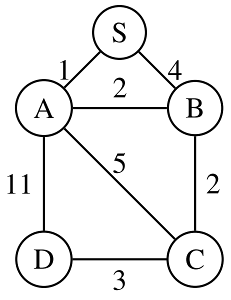


Figure 4.8: An example graph

Use the following heuristic function:

Node n	S	A	B	C	D
$h(n)$	6	2	3	0	2

Solution

Step 1. We start with S.

Nodes A and B are immediate successors of S. We have

$$\begin{aligned} f(A) &= g(A) + h(A) = 1 + 2 = 3 \\ f(B) &= g(B) + h(B) = 4 + 3 = 7 \\ \min\{f(A), f(B)\} &= \min\{3, 7\} = 3 = f(A) \end{aligned}$$

So we move to A and the path is S $\to$ A.

[3]The notation A^* is borrowed from the statistical literature. Statisticians use a hat to indicate an estimate for a quantity, and often use a star to indicate an estimate that's optimal with respect to some criterion.

Step 2. Nodes B and D can be reached form A. We have

$$\begin{aligned} f(B) &= g(B) + h(B) = (1+2) + 3 = 6 \\ f(D) &= g(D) + h(D) = (1+11) + 2 = 14 \\ \min\{f(B), f(D)\} &= \min\{6, 14\} = 6 = f(B) \end{aligned}$$

So we move to B and the path so far is S $\rightarrow$ A $\rightarrow$ B.

Step 3. C can be reached from D and C is the goal state.

Hence the required path is S $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C.

Example 2

Consider the graph shown in Figure 4.9. The numbers written on edges represent the distance between the nodes. The numbers written on nodes represent the heuristic value. Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

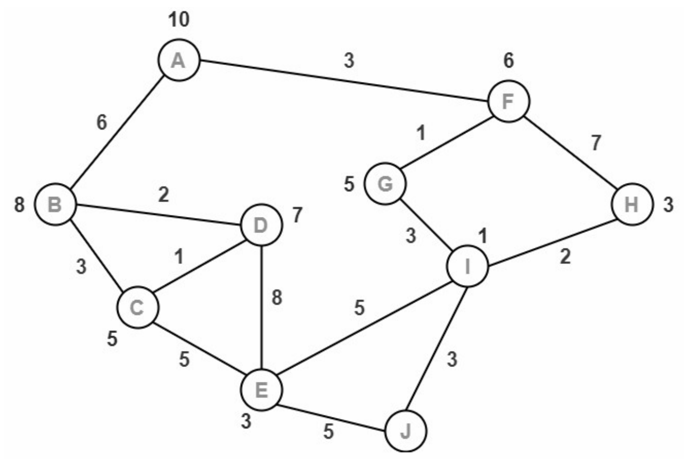


Figure 4.9: A search graph

Solution

Step 1. We start with node A.

Node B and Node F can be reached from node A.

A* Algorithm calculates $f(B)$ and $f(F)$.

$$\begin{aligned} f(B) &= 6 + 8 = 14 \\ f(F) &= 3 + 6 = 9 \end{aligned}$$

Since $f(F) < f(B)$, so it decides to go to node F.

Path: $A \rightarrow F$

Step 2. Node G and Node H can be reached from node F.

A* Algorithm calculates $f(G)$ and $f(H)$.

$$f(G) = (3+1)+5 = 9$$
$$f(H) = (3+7)+3 = 13$$

Since $f(G) < f(H)$, so it decides to go to node G.

Path: $A \to F \to G$

Step 3. Node I can be reached from node G.

A* Algorithm calculates $f(I)$.

$$f(I) = (3+1+3)+1 = 8$$

It decides to go to node I.

Path: $A \to F \to G \to I$

Step 4. Node E, Node H and Node J can be reached from node I.

A* Algorithm calculates $f(E)$, $f(H)$ and $f(J)$.

$$f(E) = (3+1+3+5)+3 = 15$$
$$f(H) = (3+1+3+2)+3 = 12$$
$$f(J) = (3+1+3+3)+0 = 10$$

Since $f(J)$ is least, so it decides to go to node J.

Path: $A \to F \to G \to I \to J$

Step 5. This is the required shortest path from node A to node J.

4.4.2 A^* best first search algorithm

Algorithm

We use the following notations:

OPEN	:	Nodes that have been generated, but not yet examined
CLOSED	:	Nodes that have already been examined
$g(n)$	:	The cost of getting from start node to node n
$h(n)$	:	The heuristic function
$f(n)$	:	$g(n) + h(n)$
start	:	The initial node
BESTNODE	:	Node in OPEN with lowest f value
SUCCESSOR	:	A successor of BESTNODE
OLD	:	A node in OPEN/CLOSED
cost(X,Y)	:	Cost of getting from node X to successor node Y.

1. (a) Set *start* as the only member of OPEN.
   (b) Set $g(start) = 0$.
   (c) Set $f(start) = h(start)$.
2. Until a the goal node is found, repeat the following procedure:
   (a) If there are no nodes in OPEN report *failure*.
   (b) Otherwise do the following:
      i. Pick the node in OPEN with the lowest f value. Call it BESTNODE.
      ii. Remove BESTNODE from OPEN.
      iii. If BESTNODE is a goal, exit and report a solution.
      iv. If BESTNODE is not a goal, generate the successors of BESTNODE. Let SUCCESSOR denote a successor of BESTNODE. For each SUCCESSOR do the following:
         A. Set SUCCESSOR to point back to BESTNODE.
         B. Set $g(\text{SUCCESSOR})$ equal to

$$g(\text{BESTNODE}) + cost(\text{BESTNODE}, \text{SUCCESSOR}).$$

         C. Examine whether SUCCESSOR is in the list OPEN.
            - Let SUCCESSOR be in OPEN.
              Call that node OLD. Delete SUCCESSOR from the list of BESTNODE's successors and add OLD. If $g(\text{OLD})$ via its current parent is greater than or equal to $g(\text{SUCCESSOR})$ via BESTNODE then reset OLD's parent link to point to BESTNODE, record new cheaper path in $g(\text{OLD})$ and update $f(\text{OLD})$.
            - Let SUCCESSOR be not in OPEN and let SUCCESSOR be in CLOSED. If so, call the node in CLOSED as OLD and add OLD to the list of BESTNODE's successors. Check to see if the new path or the old path is better (as in) and reset the parent link and g and f. If we have found a better path to old, the improvements should be propagated[4] to OLD's successors.
            - Let SUCCESSOR be not in OPEN and not in CLOSED. Then put it in OPEN and add it the list of BESTNODE's successors. Compute

$$f(\text{SUCCESSOR}) = g(\text{SUCCESSOR}) + h(\text{SUCCESSOR}).$$

4.5 Greedy best first *vs* A^* best first

4.6 Simple hill climbing

Hill climbing is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found. The solution obtained by

[4]How exactly the improvements can be propagated is a a little tricky. We omit the details.

Sl. No.	Greedy best first	A^* best first
1	The greedy best first is not complete, that is, the algorithm may not find the goal always.	If the heuristic function is admissible, the A^* best first is complete.
2	In general, the greedy best first may not find the optimal path from the initial to the goal state.	If the heuristic function is admissible, the A^* best first is always yields the optimal path from the initial state to goal state.
3	In general, greedy best first uses less memory than A^* best first.	In general, A* best first uses more memory than greedy best first.

Table 4.5: Greedy best first *vs* A^* best first

this method may not be the global optimum; it will only be a local optimum. In this sense, it is a local search method. The difference between a local and a global maxima is illustrated in Figure 4.10. In the figure, the x-axis denotes the nodes in the state space and the y-axis denotes the values of the objective function corresponding to particular states.

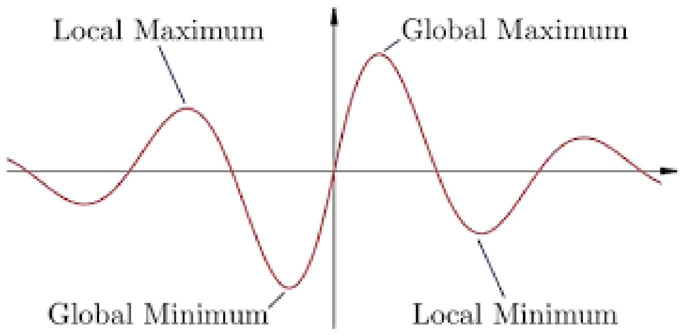


Figure 4.10: Local and global maxima and minima

When we perform hill-climbing, we are short-sighted. We cannot really see far. We make local best decisions with the hope of finding a global best solution. See Figure 4.11

Figure 4.11: Hill climbing

4.6.1 Simple hill climbing algorithm

In the following the word “operator” refers to some applicable action. It usually includes information about the preconditions for applying the action and also about the effect of the action.

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with the initial state as the current state.
2. Loop until a solution is found or until there are no new operators left to be applied in the current state:
   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
   (b) Evaluate the new state.
      i. If it is a goal state then return it and quit.
      ii. If it is not a goal state but if it is better than the current state then make it the current state.
      iii. If it is not better than the current state then continue in the loop.

4.6.2 Examples

Example 1

Apply the hill climbing algorithm to solve the blocks world problem shown in Figure 4.12:

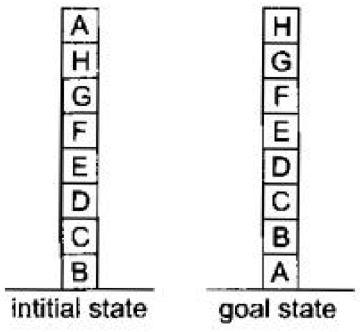


Figure 4.12: A blocks world problem

Solution

To use the hill climbing algorithm we need an evaluation function or a heuristic function. We consider the following evaluation function:

$h(n) =$ Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

We call "initial state" "State 0" and "goal state" "Goal". Then considering the blocks A, B, C, D, E, F, G, H in that order we have

$$\begin{aligned} h(\text{State 0}) &= -1-1+1+1+1+1+1+1 \\ &= 4 \\ h(\text{Goal}) &= +1+1+1+1+1+1+1+1 \\ &= 8 \end{aligned}$$

There is only one move from State 0, namely, to move block A to the table. This produces the State 1 with a score of 6:

$$h(\text{State 1}) = 1-1+1+1+1+1+1+1 = 6.$$

There are three possible moves form State 1 as shown in Figure 4.13. We denote these three

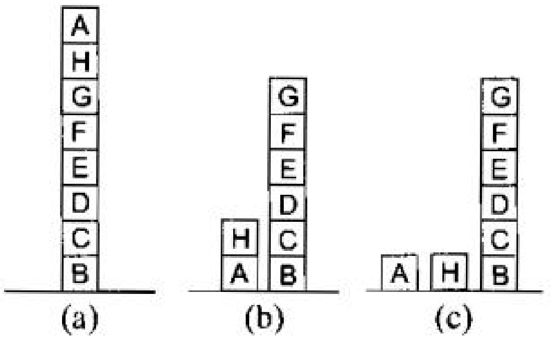


Figure 4.13: A blocks world problem

states State 2(a), State 2(b) and State 2 (c). We also have

$$\begin{aligned} h(\text{State 2(a)}) &= -1-1+1+1+1+1+1+1 = 4 \\ h(\text{State 2(b)}) &= +1-1+1+1+1+1+1-1 = 4 \\ h(\text{State 2(c)}) &= +1-1+1+1+1+1+1-1 = 4 \end{aligned}$$

Hill climbing will halt because all these states have lower scores than the current state. The process has reached a local maximum that is not a global maximum. We have reached such a situation because of the particular choice of the heuristic function. A different choice of heuristic function may not produce such a situation.

Example 2

Given the 8-puzzle shown in Figure 4.14, use the hill-climbing algorithm with the Manhattan distance heuristic to find a path to the goal state.

Solution

By definition, the Manhattan distance heuristic is the sum of the Manhattan distances of tiles from their goal positions. In Figure 4.14, only the tiles 5, 6 and 8 are misplaced and their distances from the goal positions are respectively 2, 2 and 1. Therefore,

$$h(\text{Initial state}) = 2+2+1 = 5.$$

1	2	3
4	8	
7	6	5

Initial state

1	2	3
4	5	6
7	8	

Goal state

Figure 4.14: An 8-puzzle

Figure 4.15 shows the various states reached by applying the various operations specified in the hill-climbing algorithm. The numbers written alongside the grids are the values of the heuristic function.

Note that the problem is a minimisation problem. The value of the heuristic function at the goal state is 0 which is the minimum value of the function. Note further that once we find an operation which produces a better value than the value in current state we do proceed to that state without considering whether there is any other move which produces a state having a still better value.

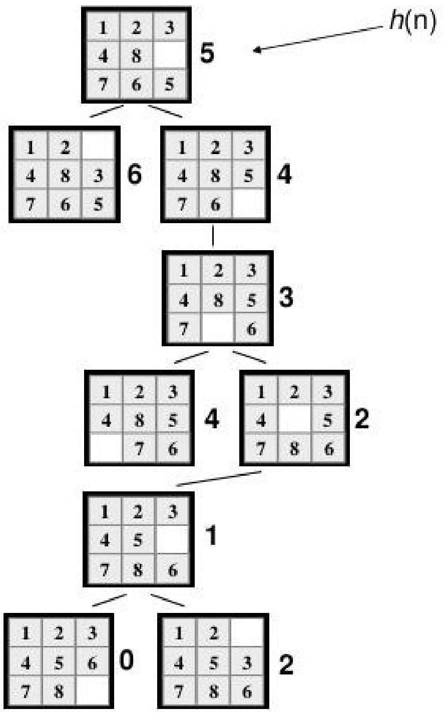


Figure 4.15: The search path obtained by applying the hill-climbing algorithm to an 8-puzzle

4.7 Steepest-ascent hill climbing

In steepest-ascent hill climbing, we consider all the moves from the current state and selects the best as the next state. In the basic hill climbing, the first state that is better than the current state is selected.

4.7.1 Algorithm

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with the initial state as the current state.

2. Loop until a solution is found or until a complete iteration produces no change to current state:

   (a) Let $SUCC$ be a state such that any possible successor of the current state will be better than $SUCC$.

   (b) For each operator that applies to the current state do:

      i. Apply the operator and generate a new state.

      ii. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to $SUCC$. If it is better then set $SUCC$ to this state. If it is not better, leave $SUCC$ alone.

   (c) If $SUCC$ is better than current state, then set current state to $SUCC$.

4.8 Advantages and disadvantages of hill climbing

Advantages

The following are some of the advantages of the hill climbing algorithms.

- Hill climbing is very useful in routing-related problems like travelling salesmen problem, job scheduling, chip designing, and portfolio management.
- It is good in solving optimization problems while using only limited computation power.
- It is sometimes more efficient than other search algorithms.
- Even though it may not give the optimal solution, it gives decent solutions to computationally challenging problems.

Disadvantages

Both the basic hill climbing and the steepest-ascent hill climbing may fail to produce a solution. Either the algorithm terminates without finding a goal state or getting into a state from which no better state can be generated. This will happen if the programme has reached either a local maximum, a ridge or a plateau.

- **Local optima (maxima or minima)**: A local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum (see Figure 4.10). A solution to this problem is to backtrack to an earlier solution and try going in a different direction.
- **Ridges**: A ridge is a sequence of local maxima. Ridges are very difficult to navigate for a hill-climbing algorithm (see Figure 4.17). This problem may be overcome by moving in several directions at once.

- **Plateaus**: A plateau is an area of the state space where the evaluation function is flat. It can be a flat local maximum, from which no uphill path exists, or a shoulder, from which progress is possible (see Figure 4.16). One solution to handle this situation is to make a big jump in some direction to get to a new part of the search space.

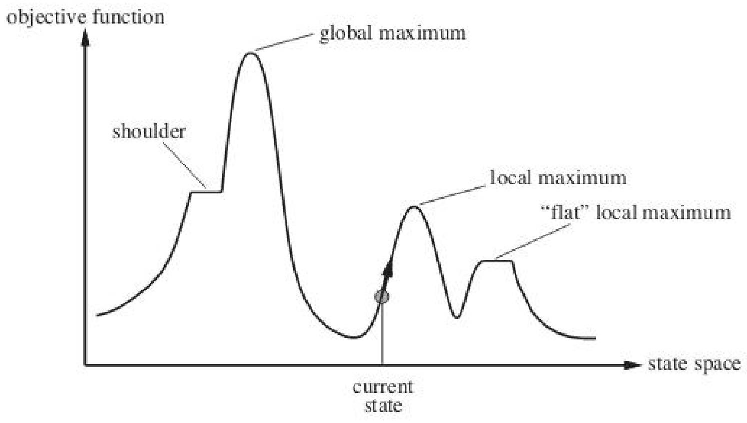


Figure 4.16: Local and global maxima

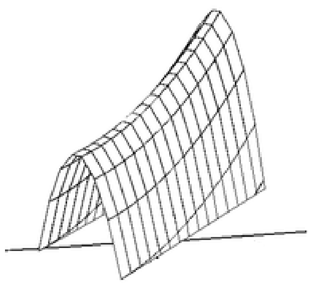


Figure 4.17: A ridge: The ridge is the curved line at the top joining the local maxima

4.9 Simulated annealing

Simulated annealing is a variation of hill climbing. In simulated annealing, some down-hill moves may be made at the beginning of the process. This to explore the whole space at the beginning so that we do not land in a local maximum, or plateau or ridge instead of the global maximum. The algorithm is somewhat analogous to a process known as "annealing" in metallurgy.

the general principle of simulated annealing is illustrated in Figure 4.18

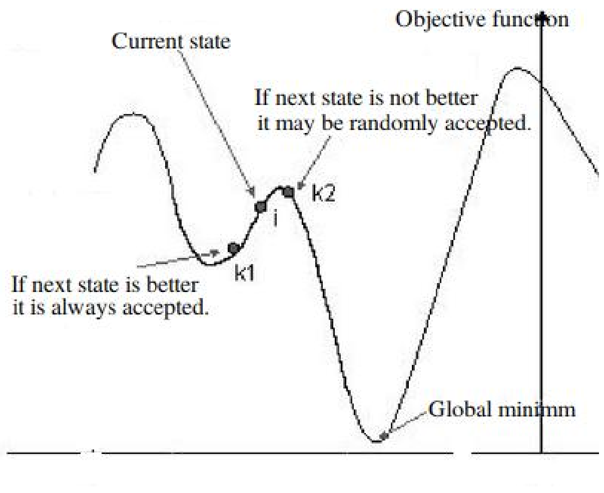


Figure 4.18: General principle of simulated annealing

4.9.1 Annealing

The name of the algorithm comes from annealing in metallurgy. Annealing is a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable. It involves heating a material above a certain temperature, maintaining a suitable temperature for an appropriate amount of time and then cooling.

4.9.2 Algorithm

Notations and conventions

The following notations and conventions are used in the algorithm.

- A variable *BEST-SO-FAR* which takes a problem state as a value.
- A certain function of time denoted by $T(t)$ or simply T. This function is called the *annealing schedule*.
- AS per standard usage, in the context of the simulated annealing algorithm, the heuristic function will be called an objective function.
- It will be assumed that the problem is a minimisation problem

Algorithm

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialise *BEST-SO-FAR* to the current state.
3. Initialise T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.

(a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(b) Evaluate the new state. Compute

$$\Delta E = \text{(value of current state)} - \text{(value of new state)}.$$

- If the new state is a goal state, then return it and quit.
- If it is not a goal state but is better than the current state, then make it the current state. Also set *BEST-SO-FAR* to this new state.
- If it is not better than the current state then make it the current state with probability $e^{-\Delta E/T}$.

5. Return *BEST-SO-FAR* as the answer.

4.10 Sample questions

(a) Short answer questions

1. Define a heuristic function and an admissible heuristic function and illustrate it with an example.

2. Give a heuristic function for the problem of path search in maps. Is it admissible? Why?

3. Give an admissible and nontrivial heuristic for the missionaries and cannibals problem.

4. Give a nontrivial heuristic for the blocks world problem. Is it admissible?

5. Define the Hamming distance and Manhattan distance heuristic functions for the 8-puzzle.

6. Given the 8-puzzle specified in Figure 4.19, what are the values of the Hamming distance and Manhattan distance heuristic functions at the initial state?

8	7	
1	3	2
6	5	4

(a) Initial state

1	2	3
4	5	6
7	8	

(b) Goal state

Figure 4.19: An initial state and a goal state of 8-puzzle

7. Define a nontrivial heuristic function for the blocks world problem and compute its value at the initial state for the problem shown in Figure 4.20.

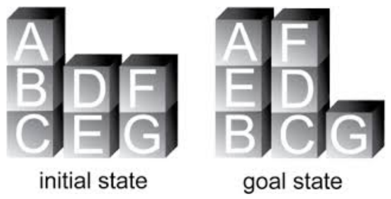


Figure 4.20: A block world problem

(b) Long answer questions

1. Compare and contrast the greedy best first and the A^*- best first algorithms.
2. Write the simple hill climbing algorithm and illustrate it with an example.
3. Write the steepest ascent hill climbing algorithm.
4. Explain the simulated annealing algorithm.
5. Explain the following types of hill climbing:
   (a) Simple hill climbing.
   (b) Steepest-ascent hill climbing
6. What are the advantages and disadvantages of the hill climbing algorithm?
7. Consider a blocks world with 4 blocks A, B, C and D with the start and goal states given in Figure 4.21. Solve the problem using the hill climbing algorithm and a suitable heuristic function. Show the intermediate decisions and states.

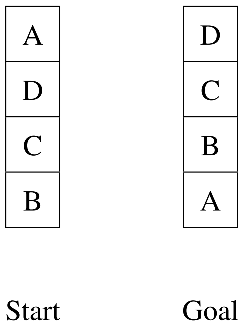


Figure 4.21: A block world problem

8. Using the greedy best first search algorithm, find an optimal path from S to G in the search graph shown in Figure 4.22. Show in a tabular form the various stages in the execution of the algorithm with details of the current node, children of the current node, the contents of the OPEN list and the contents of the CLOSED list.

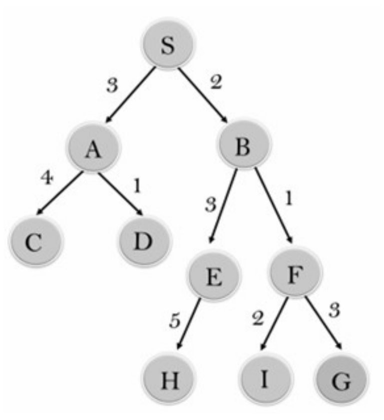


Figure 4.22: A search graph

Use the heuristic function:

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

9. Solve the previous problem using the A^* best first search algorithm.

10. Consider the graph shown in Figure 4.23. In the graph, "S" is the start state and "G1, G2, G3" are three goal states. When traversing the graph, one can move only in the direction indicated by the arrows. The numbers on the edges are the step-cost of traversing that edge. the numbers in the nodes are estimated cost to the nearest goal state. Apply each of the algorithms below to find a solution path from the start state to a goal state and to calculate the total cost to the goal.

    (a) Depth-first search

    (b) Greedy best-first search

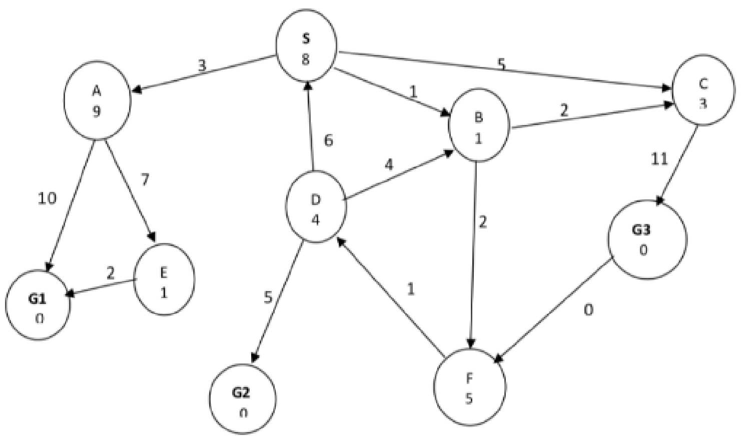


Figure 4.23: A search graph

11. Use the graph in Figure 4.24 to find the path using A^* (S is the start state and G!, G2 are the goal states). Find the path from to start to goal using the greedy best first

search also.

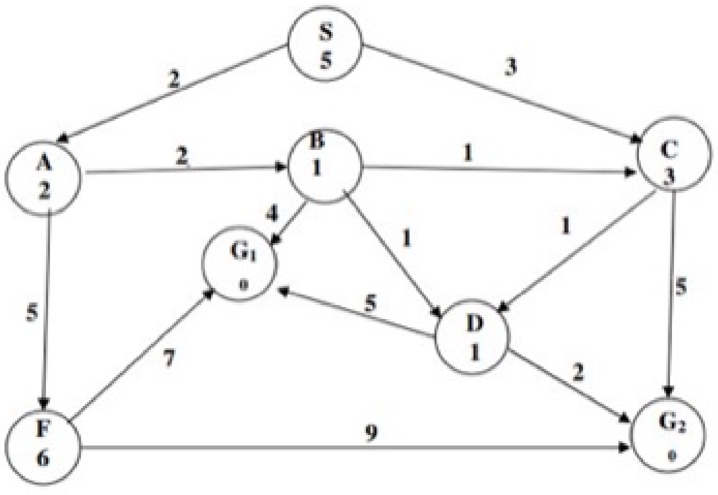


Figure 4.24: A search graph

Chapter 5

Game playing

A multiagent environment is an environment in which each agent needs to consider the actions of other agents and how they affect its own welfare. In this chapter we cover competitive environments, in which the agents' goals are in conflict, giving rise to **adversarial search problems** which are often known as **games**.

5.1 Games considered in AI

In artificial intelligence, the most commonly considered games are those that have the following properties:

- **Deterministic**: A game is deterministic if an action of a player leads to completely predictable outcomes. Chess is an example of a determinist game: the rules allow for no variation of outcome and there are no physical factors involved. Football, on the other hand, is an indeterministic game: players cannot reliably kick the ball exactly the same way each time, and the small variations of that action lead to chaotic outcomes.
- **Turn-taking**: A turn-taking game is one where the players play one at a time in alternating turns.
- **Two-player**
- **Zero-sum**: A zero-sum game[1] is defined as one where the total payoff to all players is the same for every instance of the game.
- **Perfect information**: Perfect information implies knowledge of each player's utility functions, payoffs and strategies. Chess is an example of a game with perfect information as each player can see all the pieces on the board at all times. Other examples of games with perfect information include tic-tac-toe, checkers, and Go.

Example 1: Chess

The best known example of a game considered in AI is chess. The reader is assumed to be familiar with how the game of chess is played. The initial position is shown in Figure 5.1.

[1]The terminology of *zero-sum game* comes from mathematical game theory a branch of applied mathematics extensively used in economics.

Figure 5.1: Chess board in the initial position

In chess, the outcome is a win, loss, or draw, with values $+1$, 0, or $1/2$. It is a zero-sum game because every game has the total payoff of either $0 + 1$, $1 + 0$ or $1/2 + 1/2$.

Example 2: Tic-tac-toe

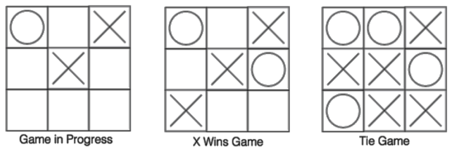


Figure 5.2: Tic-tac-toe

Tic-tac-toe or noughts and crosses is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of his/her marks in a diagonal, horizontal, or vertical row is the winner (see Figure 5.2).

5.2 Game tree

We illustrate the idea with an example. Figure 5.3 shows part of the **game tree** for the tic-tac-toe game.

Example

Consider the game of tic-tac-toe and let the players be named “MAX” and “MIN”. MAX plays by placing an X and MIN plays by placing an O in a cell in the grid.

In the figure the initial state is represented by a blank grid at the top. From the initial state, MAX has nine possible moves. These are indicated in the figure by the nine children of the initial state. Play alternates between MAX’s placing an X and MIN’s placing an O. This is continued until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

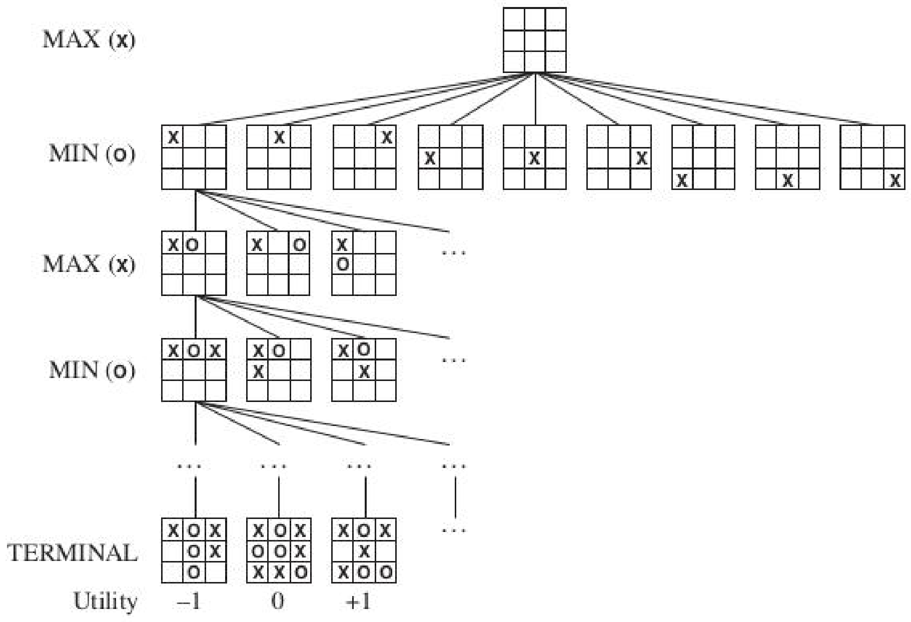


Figure 5.3: A partial game tree for the game of tic-tac-toe

5.3 Solving a game

By solving a two-player zero sum game we mean determining what actions are to be taken by the players when they are at particular states in playing the game. We assume that the players are “rational” by which we mean that each player would play to secure the optimal advantage. Since it is a zero-sum game, one player’s gain is opponent’s loss. As a convention, the player who is trying to maximize the gain is called the maximizing player and is MAX and the player who is trying to minimize the loss is called the minimizing player and is MIN.

The general method to solve such game problems is based on the “minimax strategy” whereby the player MAX chooses a move that would maximize his minimum gain and player MIN chooses a move which would minimize his maximum loss. The proceudre based on this strategy known as the “Minimax algorithm” is described in detail in Section 5.6 below.

Example 1

Solve a zero-sum two-player game represented by the game tree shown in Figure 5.4.

Solution

The steps in arriving at a solution are described below.

1. The first player is MAX and he is at state A.

2. There are three possible moves for MAX now: a_1, a_2, a_3.

3. Suppose A chooses a_1. The resulting state is B.

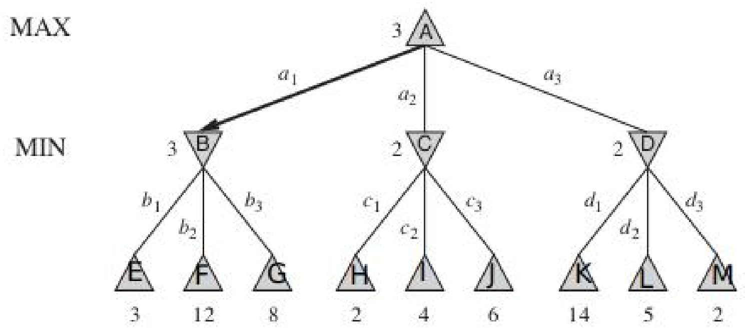


Figure 5.4: A two-ply game

   (a) Then the next move is of MIN. There are are three possible moves for MIN from B, namely, b_1, b_2, b_3. The respective losses are 3, 12, 8.
   (b) Being rational, MIN tries to minimize his loss and co chooses b_1 with a possible loss of 3. This results in a gain of 3 for MAX.

4. Suppose A chooses a_2. The resulting state is C.
   (a) Then the next move is of MIN. There are are three possible moves for MIN from B, namely, c_1, c_2, c_3. The respective losses are 2, 4, 6.
   (b) Being rational, MIN tries to minimize his loss and co chooses c_1 with a possible loss of 2. This results in a gain of 2 for MAX.

5. Suppose A chooses a_3. The resulting state is D.
   (a) Then the next move is of MIN. There are are three possible moves for MIN from B, namely, d_1, d_2, d_3. The respective losses are 14, 5, 2.
   (b) Being rational, MIN tries to minimize his loss and co chooses d_3 with a possible loss of 2. This results in a gain of 2 for MAX.

6. Now, if MAX chooses a_1 the minimum gain is 3, if MAX chooses a_2 the minimum gain is 2 and a_3 is chosen the minimum gain is 2.

7. Hence to maximize the minimum gain, MAX chooses a_1.

The solution can be stated as follows:

> "Initially, MAX chooses a_1 and then MIN chooses b_1. This guarantees a minimum gain of 3 for MAX and a maximum loss of 3 for MIN."

Example 2

Solve the game represented by the game tree in Figure 5.5. Assume that the maximising player plays first.

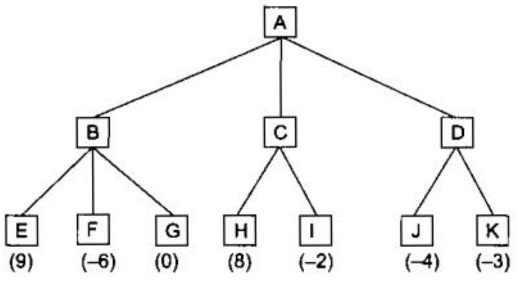


Figure 5.5: A two-ply game

Solution

We have:

$$\begin{aligned}
\text{Minimum loss for MIN from move from B} &= \min\{9, -6, 0\} = -6\\
\text{Minimum loss for MIN from move from C} &= \min\{8, -2\} = -2\\
\text{Minimum loss for MIN from move from D} &= \min\{-4, -3\} = -4\\
\text{Maximum gain for MAX from move from A} &= \max\{-6, -2, -4\} = -2.
\end{aligned}$$

The solution can be stated as follows:

> "Initially, MAX chooses the move AC and then MIN chooses CI. This guarantees a minimum gain of -2 for MAX and a maximum loss of -2 for MIN."

5.4 Formulation of a game as a search problem

We consider games with two players whom we call MAX and MIN. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser.

A game can be formally defined as a kind of search problem with the following elements:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state s.
- ACTIONS(s): Returns the set of legal moves in the state s.
- RESULT(s, a): The **transition model**, which defines the result of a move a in state s
- TERMINAL-TEST(s): A boolean function of the state s. It's value is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p.

A **zero-sum game** is defined as one where the total payoff to all players is the same for every instance of the game.

The initial state, ACTIONS function, and RESULT function define the game tree for the game-a tree where the nodes are game states and the edges are moves.

Example

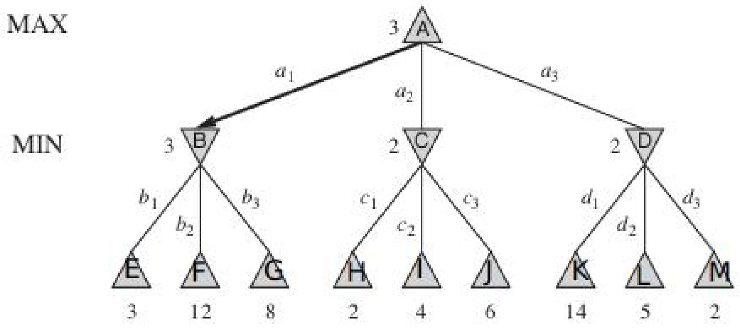


Figure 5.6: A two-ply game

To digest the above notations, let us examine what the above elements are in the game represented by the tree in Figure 5.6.

S_0	A
PLAYER(B)	MIN
ACTIONS(B)	$\{b_1, b_2, b_3\}$
RESULT(C, c_2)	I
TERMINAL-TEST(B)	False
TERMINAL-TEST(K)	True
UTILITY(E, MIN)	3

5.5 The minimax value

The minimax value at a state s, denoted by MINIMAX(s) is defined as follows:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX(RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX(RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Example

Consider a two-player game with the game tree as shown in Figure 5.6. The Δ-nodes are “MAX nodes” in which it is MAX’s turn to play and the ∇-nodes are the “MIN nodes”. The terminal nodes sho the utility values for MAX. The minimax values of the other nodes are calculated as follows:

$$\begin{aligned}\text{MINIMAX(B)} &= \min_{a\in\{b_1,b_2,b_3\}} \text{MINIMAX(RESULT}(B,a))\\ &= \min\{\text{MINIMAX(E), MINIMAX(F), MINIMAX(G)}\}\\ &= \min\{3,12,8\}\\ &= 3.\end{aligned}$$

$$\begin{aligned}\text{MINIMAX(C)} &= \min_{a\in\{c_1,c_2,c_3\}} \text{MINIMAX(RESULT}(C,a))\\ &= \min\{\text{MINIMAX(H), MINIMAX(I), MINIMAX(J)}\}\\ &= \min\{2,4,6\}\\ &= 2.\end{aligned}$$

$$\begin{aligned}\text{MINIMAX(D)} &= \min_{a\in\{d_1,d_2,d_3\}} \text{MINIMAX(RESULT}(D,a))\\ &= \min\{\text{MINIMAX(K), MINIMAX(L), MINIMAX(M)}\}\\ &= \min\{14,5,2\}\\ &= 2.\end{aligned}$$

$$\begin{aligned}\text{MINIMAX(A)} &= \max_{a\in\{a_1,a_2,a_3\}} \text{MINIMAX(RESULT}(A,a))\\ &= \min\{\text{MINIMAX(B), MINIMAX(C), MINIMAX(D)}\}\\ &= \min\{3,2,2\}\\ &= 3.\end{aligned}$$

5.6 The minimax algorithm

The minimax algorithm computes the minimax decision from the current state.

5.6.1 The algorithm

```
function MINIMAX-DECISION (state) returns an action
    return arg max_{$a\in$ACTIONS($s$)} MIN-VALUE (RESULT (state, $a$))
```

```
function MAX-VALUE (state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for each $a$ in ACTIONS(state) do
        $v \leftarrow$ MAX ($v$, MIN-VALUE (RESULT ($s, a$)))
    return $v$
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow \infty$
    for each $a$ in ACTIONS(state) do
        $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT ($s, a$)))
    return $v$
```

Remarks

1. The algorithm is presented using the pseudocode employed by Russel and Norvig in their book "*Articial Intelligence: A Modern Approach*".
2. The algorithm determines the action to be taken when the game is at *state*. To determine the action to be taken at the *initial state* the function value MINIMAX-DECISION(*initial state*) has to be computed.
3. In the algorithm the notation "←" denotes the assignment operator: "$x \leftarrow y$" means assign y to x.
4. The notation arg $\max_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$. For example, let
$$S = \{1, 2, 3, 4, 5\}$$
$$f(1) = 28, f(2) = 39, f(3) = 38, f(4) = 32, f(5) = 30$$
then
$$\text{arg max}_{a \in S} f(a) = 2.$$

5.6.2 Example

Problem

Consider a perfect two player game with the game tree shown in Figure 5.7. Assuming that the root is a maximizing node, apply the minimax algorithm to determine the optimal initial action for the MAX player.

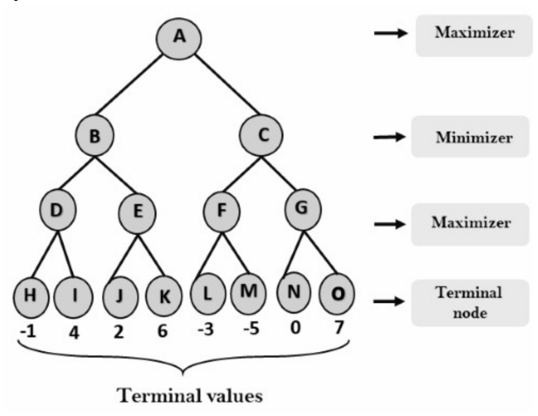


Figure 5.7: A game tree

Solution

We are required to compute MINIMAX-DECISION(A). According to the minimax algorithm, this is given by

$$\text{MINIMAX-DECISION(A)} = \underset{a \in \text{ACTIONS(A)}}{\text{arg max}} \text{ MIN-VALUE(RESULT(A,}a\text{))}.$$

Now from the game tree we can see that

$$\text{ACTIONS(A)} = \{\text{AB, AC}\}.$$

and also that

$$\text{RESULT(A, AB)=B}, \qquad \text{RESULT(A, AC)=C}.$$

Thus

$$\text{MINIMAX-DECISION(A)} = \underset{a\in\text{ACTIONS(A)}}{\text{arg max}} \{\text{MIN-VALUE(B), MIN-VALUE(C)}\}.$$

1. Next we compute MIN-VALUE(B). Since B is not a terminal state and since ACTIONS(B)= $\{$BD, BE$\}$, MIN-VALUE(B) is computed as follows:

   $v \leftarrow \infty$

   $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(B, BD))

   $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(B, BE))

   MIN-VALUE(B) $\leftarrow v$.

   (a) $v \leftarrow \infty$

   (b) We have to compute
   MAX-VALUE(RESUL(B,BD))=MAX-VALUE(D).
   Since ACTIONS(D)= $\{$ DH, DI$\}$ and since D is not a terminal state MAX-VALUE(D) is computed as follows:
   $w \leftarrow -\infty$.
   $w \leftarrow$ MAX(w, MIN-VALUE(RESULT(D, DH))
   $w \leftarrow$ MAX(w, MIN-VALUE(RESULT(D, DI))
   MAX-VALUE(D) $\leftarrow w$.

      (i) $w \leftarrow -\infty$.

      (ii) We compute
      MIN-VALUE(RESUL(D, DH))=MAX-VALUE(H).
      Since H is a terminal state, we have
      MAX-VALUE(H) = UTILITY(H) = -1.
      Thus we have
      $w \leftarrow$ MAX($-\infty$, -1)
      $w \leftarrow -1$

      (iii) We compute
      MIN-VALUE(RESUL(D, DI))=MAX-VALUE(I).
      Since I is a terminal state, we have
      MAX-VALUE(I) = UTILITY(I) = 4.
      Thus we have
      $w \leftarrow$ MAX(-1, 4)
      $w \leftarrow 4$

      (iv) Finally we get
      MAX-VALE(D)$\leftarrow$ 4.

   $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(B, BD))).
   that is, $v \leftarrow \text{MIN}(\infty, 4)$, that is $v \leftarrow 4$.

   (c) We compute
   MAX-VALUE(RESUL(B,BE))=MAX-VALUE(E).

Since E is not a terminal state, and since ACTIONS(E)= {EJ, EK}, MAX-VALUE(E) is computed as follows:
$w \leftarrow -\infty$
$w \leftarrow$ MAX(w, MIN-VALUE(RESULT(E, EJ))
$w \leftarrow$ MAX(w, MIN-VALUE(RESULT(E, EK))
MAX-VALUE(E) $\leftarrow w$.

(i) $w \leftarrow -\infty$
(ii) Let us compute the assignment
We compute
MIN-VALUE(RESUL(E, EJ))=MAX-VALUE(J).
Since J is a terminal state, we have
MAX-VALUE(J) = UTILITY(J) = 2.
Thus we have
$w \leftarrow$ MAX($-\infty$, 2)
$w \leftarrow 2$
(iii) We compute
MIN-VALUE(RESUL(E, EK))=MAX-VALUE(K).
Since K is a terminal state, we have
MAX-VALUE(K) = UTILITY(K) = 6.
Thus we have
$w \leftarrow$ MAX(2, 6)
$w \leftarrow 6$

$v \leftarrow$ MIN(v, MAX-VALUE(RESULT(B, BE)))
that is, $v \leftarrow \text{MIN}(4, 6)$, that is $v \leftarrow 4$.

(d) We now get
MIN-VALUE(B) $\leftarrow$ 4.

2. In a similar way we can compute MIN-VALUE(C) $\leftarrow -3$.

Finally

$$\begin{aligned}\text{MINIMAX-DECISION(A)} &= \underset{a \in \text{ACTIONS(A)}}{\text{arg max}} \{\text{MIN-VALUE(B), MIN-VALUE(C)}\} \\ &= \underset{a \in \text{ACTIONS(A)}}{\text{arg max}} \{4, -3\} \\ &= 4.\end{aligned}$$

The action in the set ACTIONS(A) which produces this minimax value is AB and so the player must start with the action AB.

5.7 Minimax with alpha-beta pruning

AlphaâĂŞbeta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further.

5.7.1 Basic idea

The algorithm maintains two values, α and β, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially we set $\alpha = -\infty$ and $\beta = +\infty$. Whenever the maximum score that the minimizing player (that is, the β-player) is assured of becomes less than the minimum score that the maximizing player (that is, the α-player) is assured of (that is, $\beta < \alpha$), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

5.7.2 Example

Example 1

Figure 5.8 represents a game tree showing which nodes can be pruned. Suppose the values

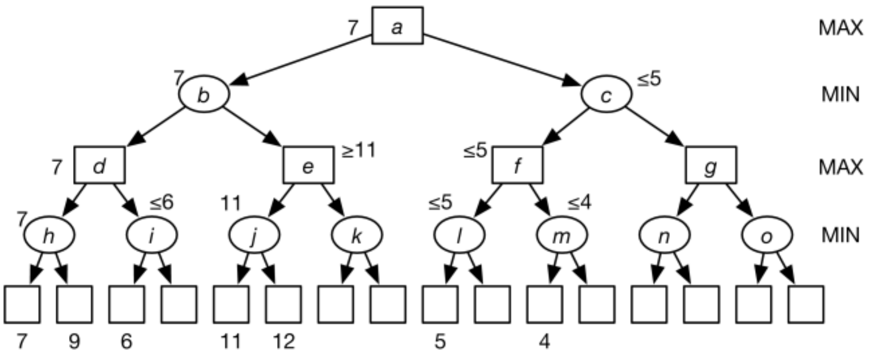


Figure 5.8: Alpha-beta pruning: Example

of the leaf nodes are given or are computed given the definition of the game. The numbers at the bottom show some of these values. The other values are irrelevant, as we show here.

- The value of node h is 7, because it is the minimum of 7 and 9.
- Just by considering the leftmost child of i with a value of 6, we know that the value of i is less than or equal to 6.
- Therefore, at node d, the maximizing agent will go left. We do not have to evaluate the other child of i.
- Similarly, the value of j is 11, so the value of e is at least 11, and so the minimizing agent at node b will choose to go left.
- The value of l is less than or equal to 5, and the value of m is less than or equal to 4; thus, the value of f is less than or equal to 5, so the value of c will be less than or equal to 5.
- So, at a, the maximizing agent will choose to go left.

Notice that this argument did not depend on the values of the unnumbered leaves. Moreover, it did not depend on the size of the subtrees that were not explored.

Example 2

Consider the game tree shown in Figure 5.6. We show below why the leave c_2 and c_3 can be pruned without affecting the final decision.

Figure 5.9 shows the various stages in the calculation of the optimal decision for the game tree in Figure 5.6. At each point, we also show the range of possible values for each node.

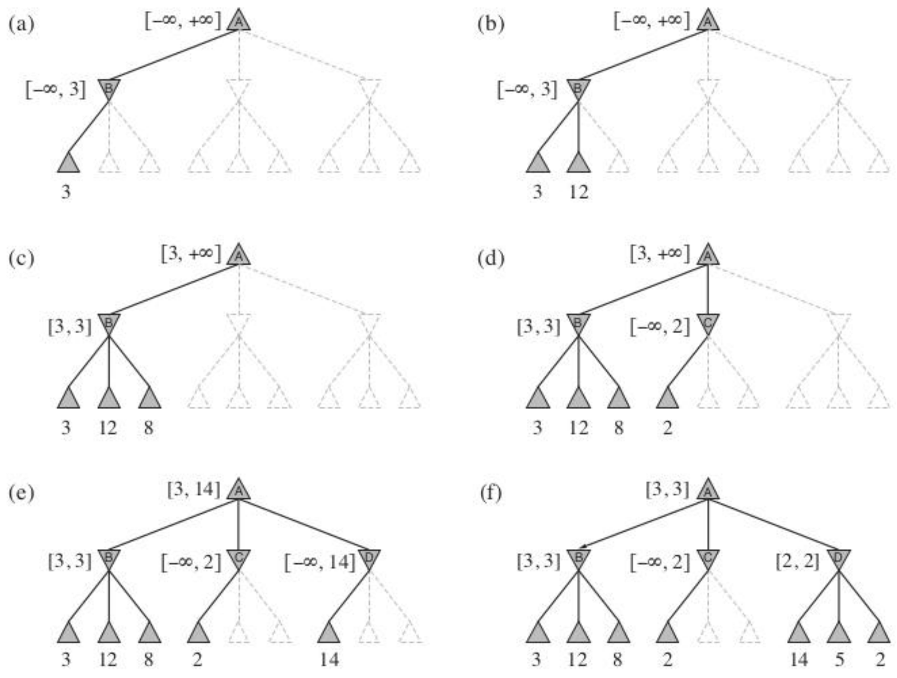


Figure 5.9: Stages in the calculation of the optimal decision for the game tree in Figure 5.6

(a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3.

(b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.

(c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.

(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. **This is an example of alpha-beta pruning.**

(e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (that is, 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.

(f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Example 3

Figure 5.10 illustrates alpha-beta pruning. The grayed-out subtrees are not explored (when moves are evaluated from left to right). The max and min levels represent the turn of the player and the adversary, respectively.

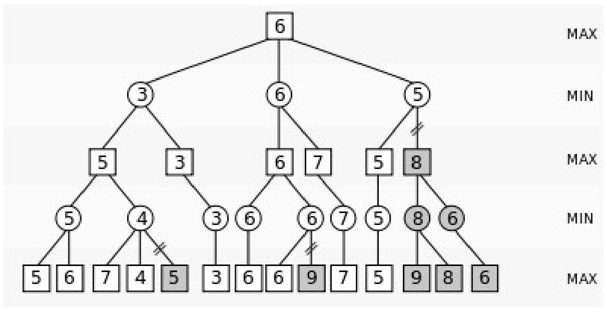


Figure 5.10: Alpha-beta pruning: Example

5.7.3 Algorithm

The minimax with alpha-beta pruning algorithm is given below.

Algorithm

```
01. function alphabeta(node, $\alpha$, $\beta$, Player)
02.     if node is a terminal node then
03.         return the value of node
04.     if Player == MAX then
05.         value := $-\infty$
06.         for each child of node do
07.             value := max(value, alphabeta(child, $\alpha$, $\beta$, MIN))
08.             $\alpha$ := max($\alpha$, value)
09.             if $\alpha \geq \beta$ then
10.                 break (* $\beta$ cutoff *)
11.         return value
12.     else
13.         value := $+\infty$
14.         for each child of node do
15.             value := min(value, alphabeta(child, $\alpha$, $\beta$, MAX))
16.             $\alpha$ := min($\beta$, value)
17.             if $\alpha \geq \beta$ then
18.                 break (* $\alpha$ cutoff *)
19.         return value
```

The algorithm is initially invoked by issuing the command:

alphabeta(*InitialState*, $-\infty$, $+\infty$, MAX)

5.7.4 Additional example

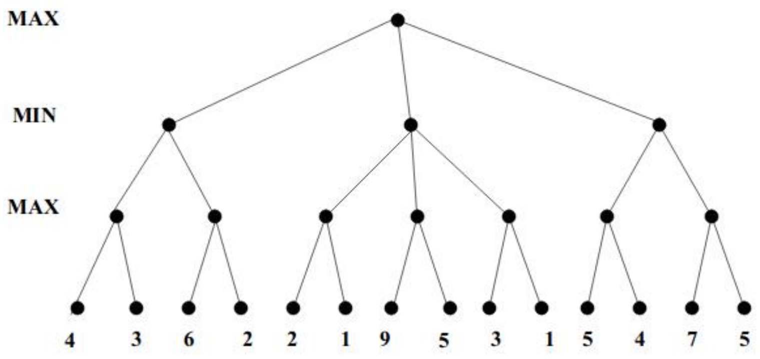


Figure 5.11: Alpha-beta pruning: Example

Solution

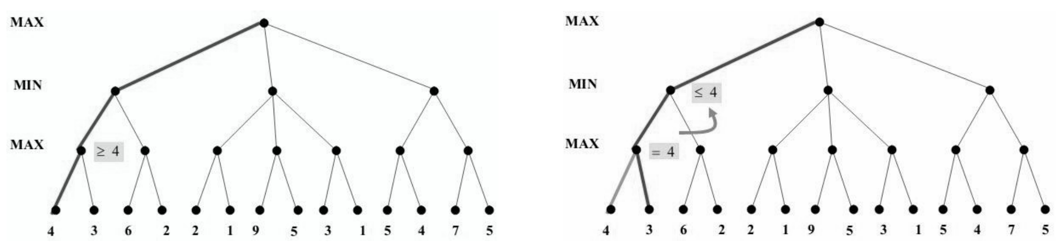


Figure 5.12: Alpha-beta pruning: Stages (1) and (2)

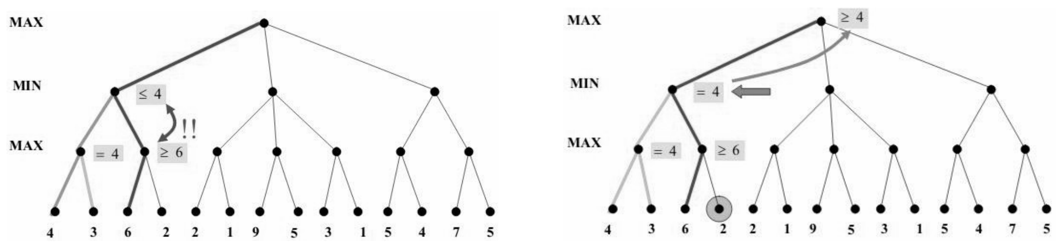


Figure 5.13: Alpha-beta pruning: Stages (3) and (4)

5.8 Sample questions

(a) Short answer questions

1. Explain the concept of a game tree with an example.

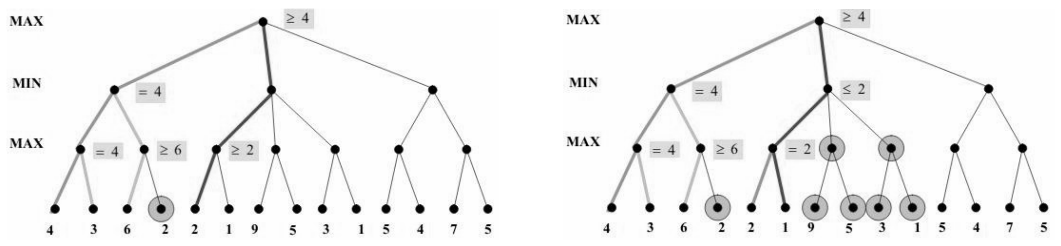


Figure 5.14: Alpha-beta pruning: Stages (5) and (6)

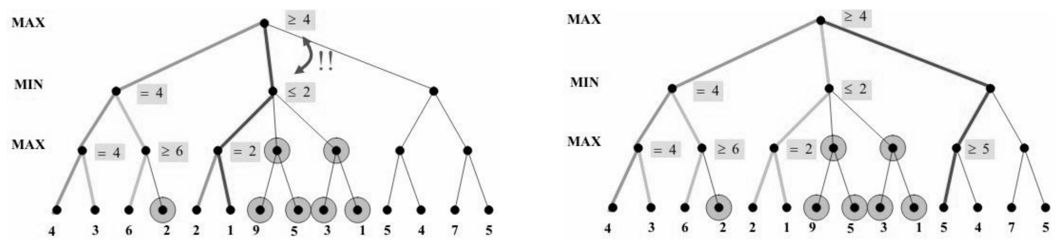


Figure 5.15: Alpha-beta pruning: Stages (7) and (8)

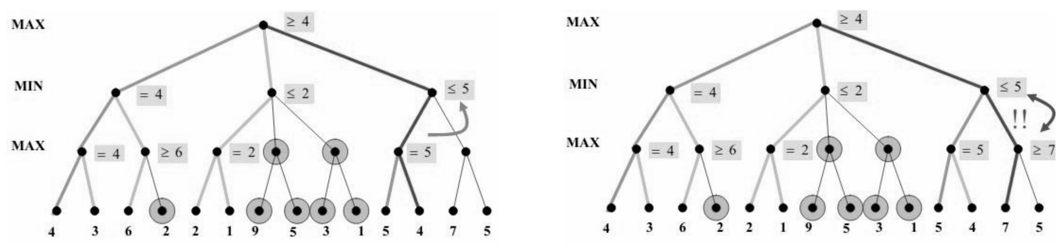


Figure 5.16: Alpha-beta pruning: Stages (9) and (10)

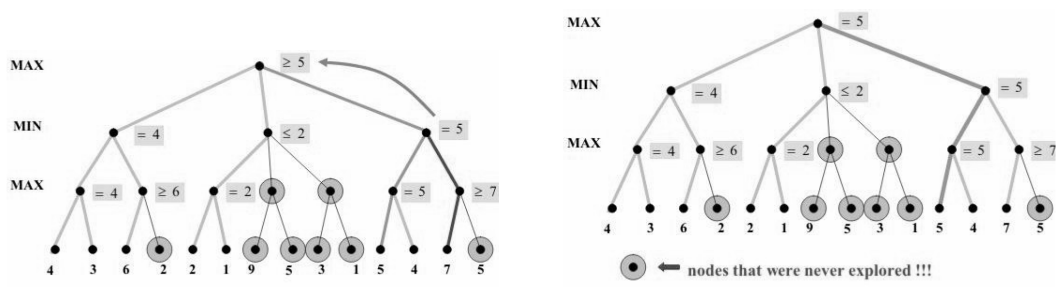


Figure 5.17: Alpha-beta pruning: Stages (11) and (12)

2. Define a zero-sum game.

3. Define the minimax value at a state s in a game.

4. Compute MINIMAX(S) in the game tree shown in Figure 5.18.

5. Why the alpha-beta pruning method is better than the minimax search method in

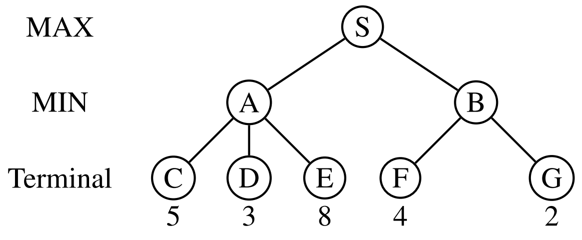


Figure 5.18: A game tree

solving a game?

(b) Long answer questions

1. Explain the minimax algorithm for computing the minimax decision.
2. Explain with an example the alpha-beta pruning algorithm.
3. Write the alpha-beta pruning algorithm.
4. Consider a two-player game in which the minimax search procedure is used to compute the best move for the first player. Assume the following game tree in which the scores are from the first player's point of view. Suppose the first player is the maximising player and needs to make the first move. What move should be chosen at this point?

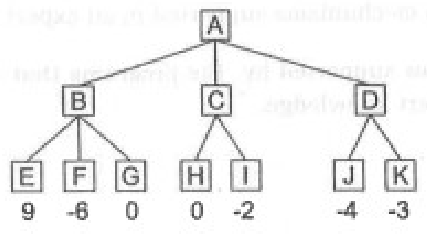


Figure 5.19: A game tree

5. Determine which of the branches in the game tree shown in Figure 5.20 will be pruned if we apply alpha-beta pruning to solve the game.

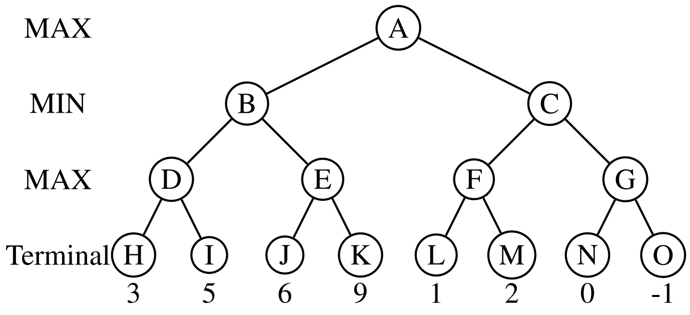


Figure 5.20: A game tree

Chapter 6

Knowledge representation

6.1 Knowledge

Knowledge is awareness or familiarity gained by experiences of facts, data, and situations. Knowledge is a familiarity, awareness, or understanding of someone or something, such as facts (propositional knowledge), skills (procedural knowledge), or objects (acquaintance knowledge). By most accounts, knowledge can be acquired in many different ways and from many sources, including but not limited to perception, reason, memory, testimony, scientific inquiry, education, and practice. The philosophical study of knowledge is called epistemology.

6.1.1 Knowledge in AI

The following are the kinds of knowledge which need to be represented in AI systems:

- **Object**:

  All the facts about objects in our world domain. E.g., Guitars contains strings, trumpets are brass instruments.

- **Events**:

  Events are the actions which occur in our world.

- **Performance**:

  It describe behavior which involves knowledge about how to do things.

- **Meta-knowledge**:

  It is knowledge about what we know.

- **Facts**:

  Facts are the truths about the real world and what we represent.

- **Knowledge base**:

  It is the totality of information, structured or unstructured, stored in a system.

6.1.2 Different types of knowledge

1. **Declarative knowledge**

   Declarative knowledge, also known as descriptive knowledge, is to know about something such as concepts, facts, and objects. It is expressed in declarative sentences.

2. **Procedural knowledge**

   Procedural knowledge, also known as imperative knowledge, is a type of knowledge which is responsible for knowing how to do something. It can be directly applied to any task and it includes rules, strategies, procedures, agendas, etc.

3. **Meta-knowledge**

   Knowledge about the other types of knowledge is called meta-knowledge.

4. **Heuristic knowledge**

   Heuristic knowledge is knowledge of experts in a field or subject. Heuristic knowledge is a "rule of thumb" knowledge based on previous experiences, awareness of approaches, and which are good to work but not guaranteed.

5. **Structural knowledge**

   It describes relationships between various concepts such as kind of, part of, and grouping of something. It describes the relationship that exists between concepts or objects.

6.2 Representation

By "representation" we mean a relationship between two domains where the first is meant to "stand for" or take place of the second. Usually the first domain is more concrete, or accessible in some way than the second. For example, a drawing of a milkshake and a hamburger on a sign might stand for a less visible fast food restaurant.

6.3 Knowledge representation

Knowledge representation is a field of artificial intelligence that focuses on designing computer representations that capture information and knowledge about the world that can be used to solve complex problems. Knowledge representation is not just storing data into some database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human. Virtually all knowledge representation languages have a reasoning or inference engine as part of the system.

A **knowledge representation structure** (or, **knowledge representation system**) is a particular set of definitions, rules and procedures for setting up a representation that captures information and knowledge about the world.

6.4 Desirable properties of knowledge representation systems

The list below gives the important desirable properties of knowledge representation systems. Unfortunately there is no single knowledge representation systems that that have all the desired properties. Because of this, there are different knowledge representations systems having varying capabilities.

1. **Representational adequacy**

   This the ability to represent all kinds of knowledge that are needed in a domain.

2. **Inferential adequacy**

   This is the ability to derive new knowledge from old.

3. **Inferential efficiency**

   This is the ability to incorporate into the knowledge base additional information.

4. **Acquisitional efficiency**

   This is the ability to acquire new knowledge easily.

6.5 Some knowledge representation systems

There are several knowledge representation systems such as the following.

- Semantic networks
- Frames
- Conceptual dependencies
- System based on logic

6.6 Semantic networks

A network is an interconnected group or system. The word "semantic" means "related to meaning, especially meaning in language". Thus a "semantic network" is an interconnected system or group related to meaning. Such a system can be represented by a directed labelled graph. Semantic networks are a logic-based formalism for knowledge representation.

6.6.1 Definition

A **semantic network** is a graph constructed from a set of vertices (or nodes) and a set of directed and labelled edges. The vertices or nodes represent concepts or objects, and the edges represent relations between the nodes.

6.6.2 Examples

Example 1

Consider the knowledge contained in the following sentence:

S: "Sparrow is a bird."

There are two concepts in the sentence, namely, “Sparrow” and “Bird”. The relation between these concepts is indicated by “is a”. We represent the two concepts by two nodes in a graph and the relation between them by a directed edge with the label “is-a”. (We have represented the “is a” relation by “is-a” as it is the convention in the theory of semantic networks.) The knowledge contained in S can now be represented by the graph shown in Figure 6.1. This is the semantic network representation of the knowledge contained in S.

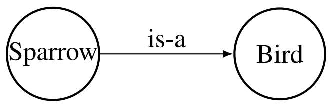


Figure 6.1: Semantic network representation of “Sparrow is a bird”

Example 2

Consider the knowledge contained in the following sentences:

S: “Tweety is a bird.”, “Birds are animals.”

The sentences has three objects “Tweety”, “Birds” and ”Animals”. Tweety is the name of a particular bird. So the sentence “Tweety is a bird” means that the bird Tweety is an instance of the class of things indicated by the word “Birds”. The sentence “Birds are animals” means that all members of the class of things indicated by ”Birds” are also members of the class of things indicated by “Animals”.

To represent the knowledge contained in S by a semantic network, we need three vertices to denote the three objects. These vertices are to be connected by directed edges with appropriate labels. The knowledge contained in S is represented by the graph exhibited in Figure 6.2.

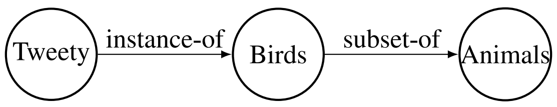


Figure 6.2: Semantic network representation of the sentences “Tweety is a bird.” and “Birds are animals.”

We may represent the nodes as rectangles as in Figure 6.3.

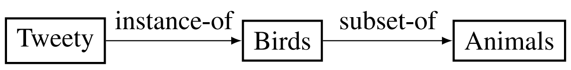


Figure 6.3: Semantic network representation of the sentences “Tweety is a bird.” and “Birds are animals.”

Example 3

Consider the sentence:

S: “A bird has feathers.”

This sentence means that the object “Feather” is part of the the object “Bird”. In a semantic network representation of the sentence, the edge joining the nodes is labelled “HAS-A” (or “HASA”, or “PART-OF”) as in Figure 6.4. Such a relation is referred to as a *part-whole relation*. The link or edge is from whole to part.

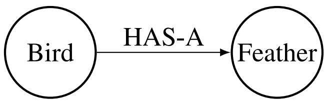


Figure 6.4: Semantic network representation of “A bird has feathers”

Example 4

Consider the sentence:

S: “Bill is taller than John.”

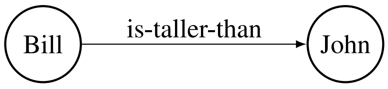


Figure 6.5: Inappropriate semantic network representation of “Bill is taller than John”

This representation of the sentence does not represent the knowledge expressed by the sentence because the relation “is-taller-than” cannot be taken as a basic relation. What the sentence actually means is that the height of Bill is greater than than the height of John. Thus the proper semantic representation of the sentence “Bill is taller than John” is the one given Figure 6.6. In this figure we have used the basic relation “is-greater-than” and also the relation “height”.

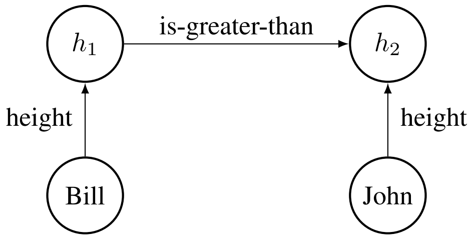


Figure 6.6: Semantic network representation of “Bill is taller than John”

Example 5

Consider the sentence:

S: “Sara has brown eyes.”

This sentence has three concepts “Sara”, “Brown” and “Eyes”. “Brown” is a property of “Eyes” and, in a semantic representation of the sentence, the property can be denoted by the label “Property” or more specifically by the label “Colour”. The semantic network is shown in Figure 6.7.

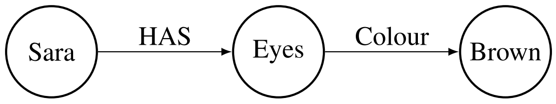


Figure 6.7: Semantic network representation of the sentence “Sara has brown eyes.”

Example 6

Consider the knowledge represented by the following sentences:

- Tweety and Sweety are birds.
- Tweety has a red beak.
- Sweety is Tweety's child.
- A crow is a bird.
- Birds can fly.

The semantic network representation of the sentences is given in Figure 6.8. Note the use of the labels "Property", "Child-of" and "Color" in the network.

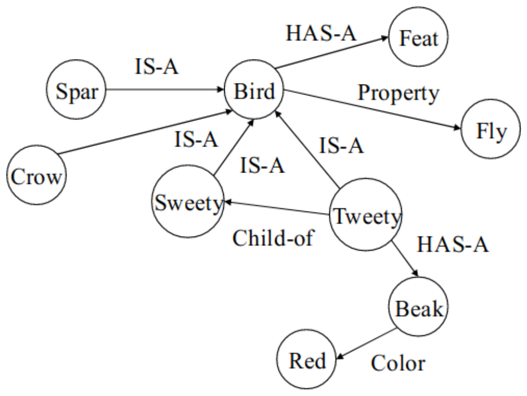


Figure 6.8: A semantic network

Example 7

Consider the following sentences:

- Motor bike is a two wheeler.
- Scooter is a two wheeler.
- Two wheeler is a moving vehicle.
- Moving vehicle has a brake.
- Moving vehicle has a engine.
- Moving vehicle has electrical system.
- Moving vehicle has fuel system.

The semantic network representation of the sentences is given in Figure 6.9.

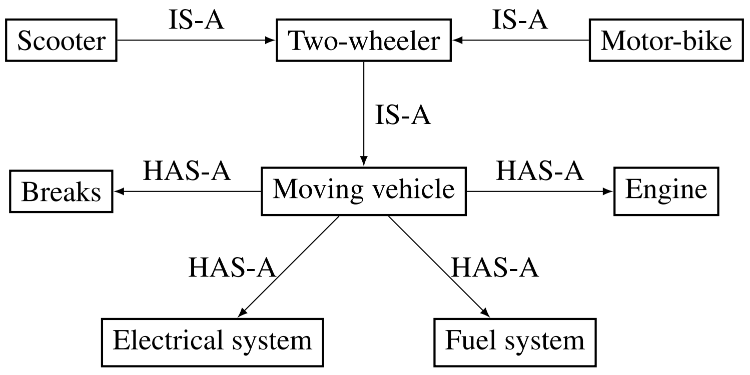


Figure 6.9: A semantic network

Example 8

Figure 6.10 shows another simple semantic network. Note the use of the label "is" instead of "IS-A". This is perfectly permitted as there are no standardisation of the notations for labels. Also note the labels "lives in" and "works in".

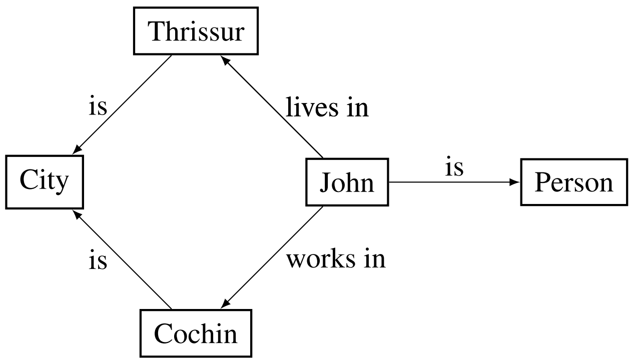


Figure 6.10: A simple semantic network

Example 9

Figure 6.11 shows a semantic network with several objects and categories.

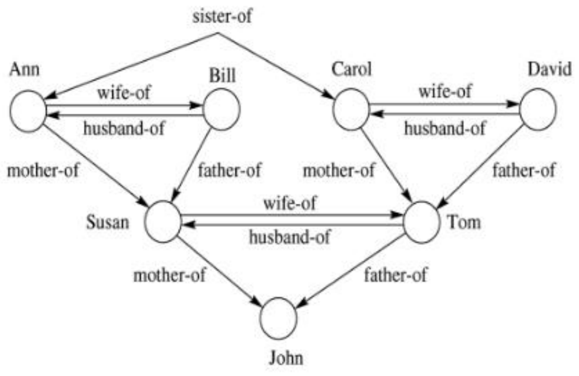


Figure 6.11: A semantic network

Example 10

Figure 6.12 shows a semantic network with two objects (John and Mary) and four categories (Mammals, Persons, FemalePersons and MalePersons).

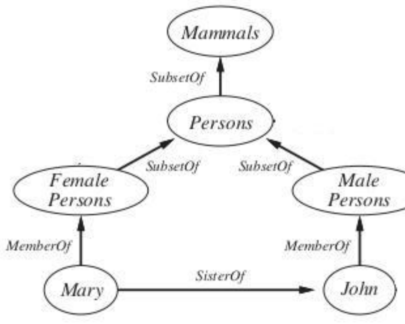


Figure 6.12: A semantic network

Example 11

Figure 6.13 shows a semantic network with "IS-A" and "AKO" (A-KIND-OF) links. The link AKO is used to relate one class (or category) to another. AKO relates generic nodes to generic nodes while the IS-A relates an instance or individual to a generic class.

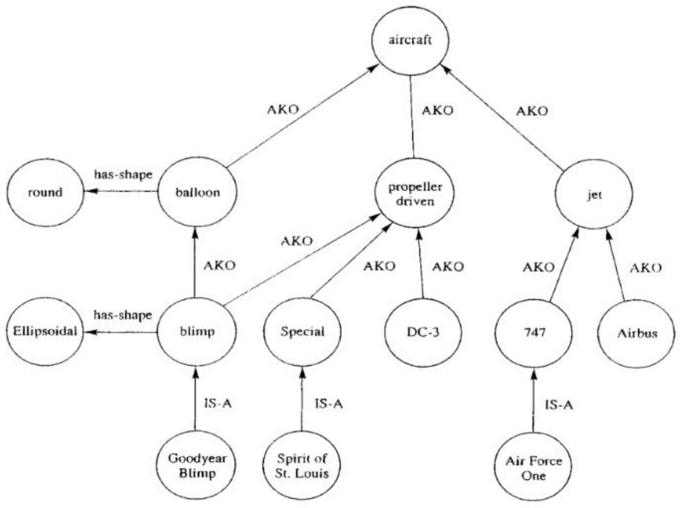


Figure 6.13: A semantic network

Example 12

Figure 6.14 illustrates a semantic network where the principal concept is "financial product," which can be a loan or an investment. An example of a long-term loan is a mortgage, and an example of a low-risk investment is a government bond. Note the variety of labels used to represent the various relations. Note also the presence of two-way links connecting nodes.

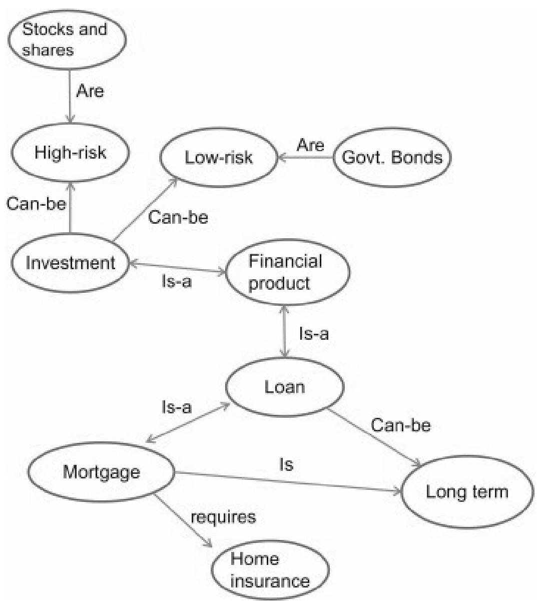


Figure 6.14: A semantic network

6.6.3 Remarks

It appears that there are no universally agreed conventions on the use of geometrical shapes to represent nodes. Ovals, circles and rectangles are all used to represent nodes. Also, there is no complete set of standard terminology to label the edges in semantic graphs. However, the following are some of the most common semantic relations used in semantic networks.

- IS-A (ISA)
- IS-PART-OF
- IS-SUBSET-OF
- A-KIND-OF (AKO)

6.6.4 Different kinds of semantic networks

The following are six of the most common kinds of semantic networks:

1. **Definitional networks**

   These emphasize the subtype relation between a concept type and a newly defined subtype.

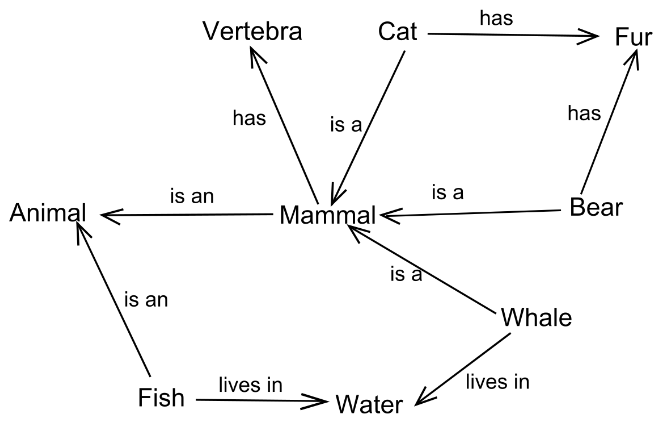


Figure 6.15: A definitional semantic network

2. **Assertional networks**

   Assertional semantic networks, also known as **propositional semantic networks** are designed to represent assertions or propositions. Figure 6.16 shows a semantic network representation of the proposition "All men are mortal". In the figure, node M1 corresponds to the proposition that "All men are mortal." Node V1 is the variable corresponding to "all men." Node M2 is the proposition that corresponds to the arbitrary man being a member of the class of men.

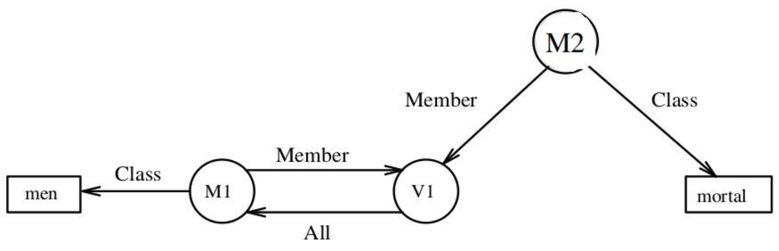


Figure 6.16: An assertional semantic network

3. **Implicational networks**

   These use implication as the primary relation for connecting nodes.

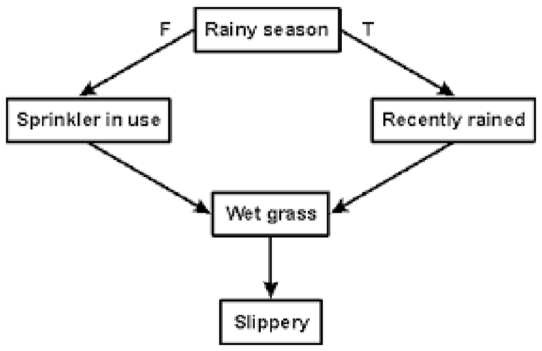


Figure 6.17: An implicational semantic network

4. **Executable networks**

   These include some mechanism which can perform inferences, pass messages, or search for patterns and associations.

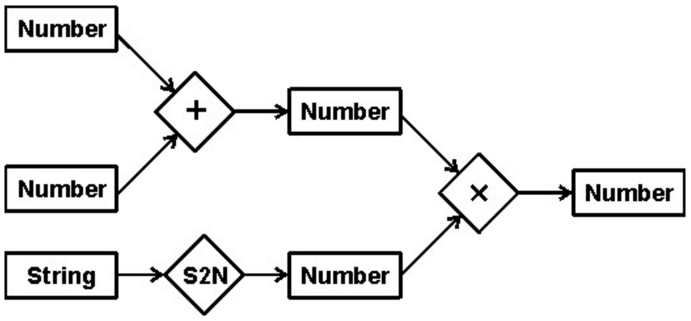


Figure 6.18: An executable semantic network

5. **Learning networks**

   These build or extend their representations by acquiring knowledge from examples.

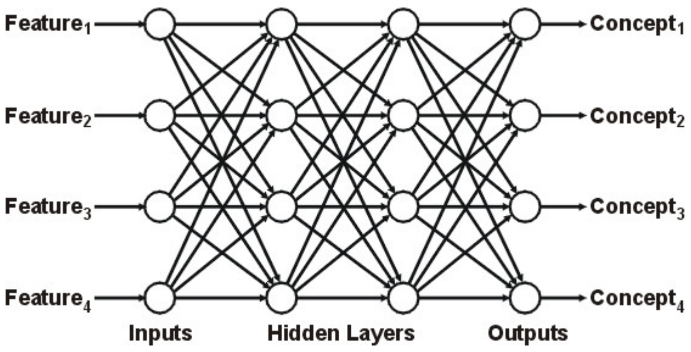


Figure 6.19: A learning semantic network

6. **Hybrid networks**

   These combine two or more of the previous techniques.

6.6.5 Intersection search

One of the earliest ways that semantic networks were used was to find relationships between objects by spreading **activation** from each of two nodes and seeing where the activations met. This process is called **intersection search**.

For example, given the semantic network shown in Figure 6.20 consider the question: "What is the relation between Chicago Cubs and Brooklyn Dodgers?"

By spreading activations from the nodes Chicago Cubs and Brooklyn Dodgers, we see that they meet at the node Baseball Players. Thus the question can be answered as "They are both teams of baseball players"

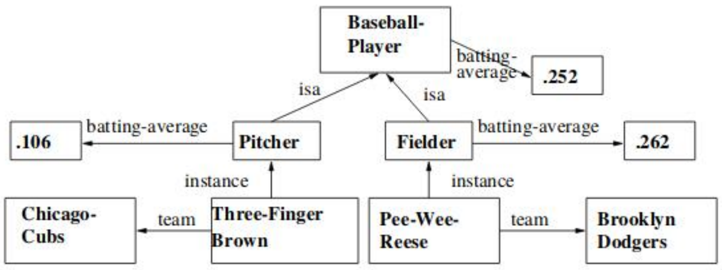


Figure 6.20: A learning semantic network

6.6.6 Representing nonbinary predicates

To represent a proposition involving a nonbinary relation in semantic nets, we create a node to representing the particular proposition and then relate the elements of the proposition to the newly created node.

Example 1

Consider the proposition: "John gave the book to Mary." This is a proposition involving three elements.

This proposition can be considered as an instance of the category of events called "Give". Let us call this instance as EV7. There are three components to this event, namely, an agent (the giver), a beneficiary and an object. The object "the book" is a particular instance of the the category "Book". Let us denote this particular instance of Book by BK23. The proposition can b now be represented by the semantic net shown in Figure 6.21.

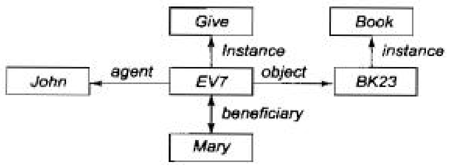


Figure 6.21: A semantic network representing the sentence: "John gave the book to Mary."

Example 2

Consider the proposition: "The score in Cubs *vs.* Dodgers game was 5 - 3." This is also a proposition involving three elements. This may be represented as a semantic network as in Figure 6.22. In the figure, "G23" is the particular instance of the category "Game" played by Cubs and Dodgers.

6.7 Advantages and disadvantages of semantic networks

6.7.1 Advantages

1. They give an adaptable method of representing knowledge because many different types of object can be included in the network.

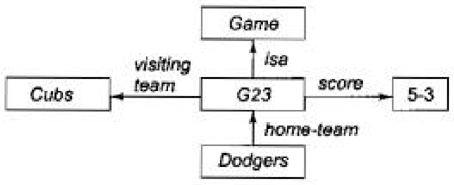


Figure 6.22: A semantic network representing the sentence: “The score in Cubs *vs.* Dodgers game was 5 - 3.”

2. The network is graphical and therefore relatively easy to understand.
3. Can be used as a common communication tool between the knowledge engineer and the human expert.
4. Efficient in space requirement.
5. Easily clusters related knowledge.

6.7.2 Disadvantages

1. There is no standard definition of link names.
2. Semantic networks are not intelligent, dependent on creator.
3. Links are not all alike in function or form.
4. Links on objects represent only binary options.
5. Undistinguished nodes that represent classes and that represent individual objects.
6. Processing is inefficient for large networks.
7. Do not represent performances effectively.
8. It is difficult to express some properties using semantic networks, like negation, disjunction, etc.

6.8 Frames

6.8.1 Definition

A **frame** is a collection of attributes and possible values that describe some entity in the world. A frame system is a collection of frames that are connected to each other by the fact that the value of an attribute in one frame may be another frame.

Remarks

1. The basic characteristic of a frame is that it represents related knowledge about a narrow subject.
2. A frame system is a good choice for describing a mechanical device, for example a car.

3. Just as with semantic nets, *there are no standards for defining frame-based systems*.
4. A frame is analogous to a record structure, corresponding to the fields and values of a record are the slots and slot fillers of a frame.
5. A frame is basically a group of slots and fillers that defines a stereotypical object.
6. A frame is also known as **slot-filter knowledge representation** in artificial intelligence.

6.8.2 Frame structure

The frame contains information on how to use the frame, what to expect next, and what to do when these expectations are not met. Some information in the frame is generally unchanged while other information, stored in "terminals", usually change. Terminals can be considered as variables. Top level frames carry information, that is always true about the problem in hand, however, terminals do not have to be true. Their value might change with the new information encountered. Different frames may share the same terminals.

Each piece of information about a particular frame is held in a slot. The information can contain:

- Facts or data
  - Values (called facets)
- Procedures (also called procedural attachments)
  - IF-NEEDED: deferred evaluation
  - IF-ADDED: updates linked information
- Default values
  - For data
  - For procedures
- Other frames or subframes

6.8.3 Examples

Example 1

Table 6.1 is a frame for a book.

Slot	Value
Title	Artificial Intelligence
Genre	Computer Science
Author	Peter Norvig
Edition	Third Edition
Year	1996
Page	1152

Table 6.1: A frame for a book

Example 2

Table 6.2 is the frame representation of the statement: “Peter is an engineer as a profession, and his age is 25. He lives in city London, and the country is England.”

Slot	Value
Name	Peter
Profession	Doctor
Age	25
Marital status	Single
Weight	78

Table 6.2: A frame

Example 3

Table 6.3 displays another frame. In this frame, he car is the object, the slot name is the attribute, and the filler is the value.

Slots	Fillers
Manufacturer	General Motors
Model	Chevrolet Caprice
Year	1979
Transmission	Automatic
Tyres	4
Colour	Blue
engine	Petrol

Table 6.3: A frame

Example 4

Table 6.4 defines a frame with name ALEX. The various slots in the frame, their names, their values and their types are all specified in the table.

Slot	Value	Type
ALEX	-	(This Frame)
NAME	Alex	(key value)
ISA	BOY	(parent frame)
SEX	Male	(inheritance value)
AGE	IF-NEEDED:	(procedural
	Subtract(current,BIRTHDATE);	attachment)
HOME	100 Main St.	(instance value)
BIRTHDATE	8/4/2000	(instance value)
FAVORITE_FOOD	Spaghetti	(instance value)
CLIMBS	Trees	(instance value)
BODY_TYPE	Wiry	(instance value)
NUM_LEGS	1	(exception)

Table 6.4: Example frame

The parent frame BOY of the frame named ALEX is shown in Table 6.5.

Slot	Value	Type
BOY	-	(This Frame)
ISA	Person	(parent frame)
SEX	Male	(instance value)
AGE	Under 12 yrs.	(procedural attachment
		- sets constraint)
HOME	A Place	(frame)
NUM_LEGS	Default = 2	(default, inherited
		from Person frame)

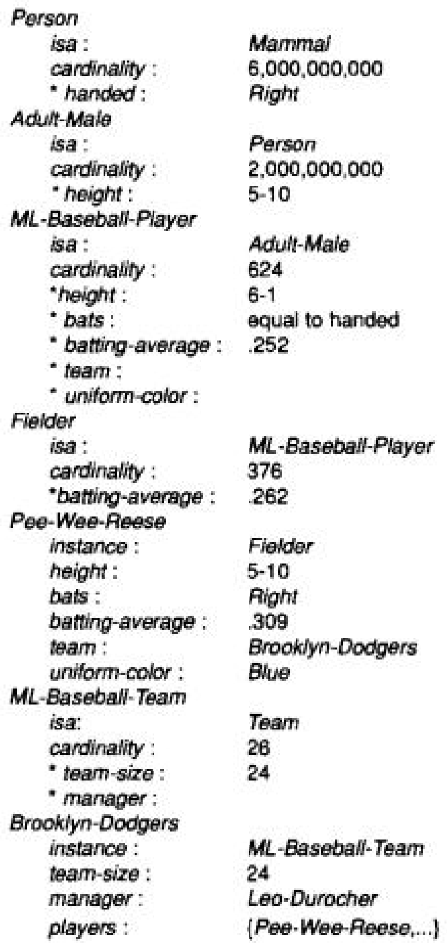
Table 6.5: Example frame

Example 5

Figure 6.23 shows a simplified frame system. There are two types of attributes that can be associated with a class or set: those that are about the set itself and those inherited by each element of the set. In the figure, the latter type of attributes are prefixed by an asterisk (*).

6.8.4 Frame languages

A **frame language** is a technology used for knowledge representation in artificial intelligence. The earliest Frame based languages were custom developed for specific research projects and were not packaged as tools to be re-used by other researchers. One of the first general purpose frame languages was KRL (Knowledge Representation Language). One of



Person	
isa :	*Mammal*
cardinality :	6,000,000,000
** handed :*	*Right*
Adult-Male	
isa :	*Person*
cardinality :	2,000,000,000
** height :*	5-10
ML-Baseball-Player	
isa :	*Adult-Male*
cardinality :	624
**height :*	6-1
** bats :*	equal to handed
** batting-average :*	.252
** team :*	
** uniform-color :*	
Fielder	
isa :	*ML-Baseball-Player*
cardinality :	376
**batting-average :*	.262
Pee-Wee-Reese	
instance :	*Fielder*
height :	5-10
bats :	*Right*
batting-average :	.309
team :	*Brooklyn-Dodgers*
uniform-color :	*Blue*
ML-Baseball-Team	
isa:	*Team*
cardinality :	26
** team-size :*	24
** manager :*	
Brooklyn-Dodgers	
instance :	*ML-Baseball-Team*
team-size :	24
manager :	*Leo-Durocher*
players :	{*Pee-Wee-Reese,...*}

Figure 6.23: A frame system

the most influential early Frame languages was KL-ONE. KL-ONE spawned everal subsequent Frame languages. One of the most widely used successors to KL-ONE was the Loom language developed by Robert MacGregor at the Information Sciences Institute.

6.9 Conceptual dependency

Conceptual dependency (CD) is a theory of knowledge representation developed by Roger Schank and his teammates at Yale University in the 1970's. It is a theory to represent natural language sentences in such way that:

- It is independent of the language in which the sentences are stated.
- It facilitates drawing inferences from sentences.

6.9.1 Simple examples of representations of sentences in CD theory

Example 1

Consider the English sentence:

"John ran".

Let us analyse this sentence.

This sentence has tow words "John" and "ran". The first word refers to a particular person or a particular object. The second word is a verb and hence denotes an action. The action is running, that is, moving from one place to another. In the terminology of CD theory, we say that it is an example of a physical transfer of an object from one place to another. We also say that it is an action of the PTRANS type. (PTRANS stands for physical transfer.) In the sentence, "ran" depends on "John" and "John" depends on "ran". There is a mutual dependency.

In the notations and terminology of CD theory, we represent the sentence "John ran." in the following form:

$$\text{John} \overset{\text{p}}{\Longleftrightarrow} \text{PTRANS}$$

Here, the two-sided double-line arrow indicates mutual dependency and the letter "p" above the arrow indicates that the action is in the past tense.

Further, in CD theory, we think of "John" as a member of the category of "real world objects" which is denoted by "PP" which is an abbreviation for "picture provider". Also, PTRANS is a member of the category of real world actions which is denoted by "ACT". The representation of the sentence "John ran." given above can be thought of as obtained by an application of the following rule of dependency in CD theory:

$$\text{PP} \Longleftrightarrow \text{ACT}$$

Example 2

Consider the English sentence:

"John gave Mary a book."

Let us analyse this sentence.

There are four concepts in the sentence: "John", "gave", "Mary", "book". In the sentence the actor is "John", the action is "gave" and the object of the action is "book". The actor and the action are mutually dependent, but the action and object are not. Action depends on the object, but the object does not depend on the action. So, the sentence "John gave a book" is represented as follows:

$$\text{John} \overset{\text{P}}{\Longleftrightarrow} \text{gave} \overset{\text{O}}{\longleftarrow} \text{book}$$

Now, consider the action "gave". It has two meanings: simply a physical transfer, or an ownership transfer. In the former case it is an example of PTRANS. In the latter case it is an abstract transfer which in CD theory is indicated by ATRANS. Assuming that it is the latter, we may represent "John gave a book." as

$$\text{John} \overset{\text{P}}{\Longleftrightarrow} \text{ATRANS} \overset{\text{O}}{\longleftarrow} \text{book}$$

Next, let us consider the dependency of "Mary" on the other concepts in the sentence. The dependency of "Mary" on "John" is one of a "recipient-donor" relationship and the "book" is dependent on this relation. So the dependency can be represented as follows:

$$\begin{tikzcd} & & \text{Mary} \\ \text{book} & {} \arrow[l, "R"'] \arrow[ur, "\text{to}"] & \\ & & \text{John} \arrow[ul, "\text{from}"] \end{tikzcd}$$

In the diagram, R indicates a recipient-donor relationship. Putting all the components together, we obtain the CD representation of the sentence "John gave Mary a book." as shown below:

$$\begin{tikzcd} & & & & \text{Mary} \\ \text{John} \arrow[r, Leftrightarrow, "p"] & \text{ATRANS} & \text{book} \arrow[l, "O"'] & {} \arrow[l, "R"'] \arrow[ur, "\text{to}"] & \\ & & & & \text{John} \arrow[ul, "\text{from}"] \end{tikzcd}$$

Example 3

Consider the English sentence:

"John ate the ice cream with a spoon."

The verb "ate" which is the past tense of "eat" is conceptualised as "ingesting" or "taking in" and is indicated by the primitive action INGEST. In the sentence, the spoon is the instrument of the action. In CD theory, an instrument of action is indicated by the letter "I". The sentence can be represented in the following form.

$$\begin{tikzcd} & & & \text{John} \arrow[dd, Leftrightarrow] \\ \text{John} \arrow[r, Leftrightarrow, "p"] & \text{INGEST} & \text{ice cream} \arrow[l, "O"'] & {} \arrow[l, "I"'] \\ & & & \text{do} \\ & & & \text{spoon} \arrow[u, "O"'] \end{tikzcd}$$

Remark

In any domain in which we build a conceptual representation system, we will have to choose an appropriate level for primitive actions.

6.9.2 The building blocks of conceptual dependency

In the examples of conceptual representations discussed above, we have seen several primitive actions like PTRANS, ATRANS and INGEST, the primitive category PP, the symbol for the past tense, symbols indicating relations like o for object relation and R for recipient-donor relation. In this section we have collected all these building blocks of the conceptual dependency theory.

1. Primitives conceptual categories

The primitive conceptual categories available in the CD theory are given Table 6.6.

Primitive	Meaning
ACT	Real world actions. There are only eleven of these ACTs (see Table 6.7).
PP	Real world objects (picture producers). Only physical objects are PPs.
AA	Modifiers of actions (action aiders) (Attributes of actions)
PA	Attributes of an object (picture aiders). PAs take the form: STATE(VALUE). That is, a PA is an attribute characteristic (like color or size) plus a value for that characteristic (red or 10 cm).
T	Times
LOC	Location

Table 6.6: Primitive conceptual categories in CD theory

2. The primitive acts of CD

The CD theory recognises 11 primitive acts. Table 6.7 gives these primitive acts, their meanings and examples.

Table 6.7: Primitives of conceptual dependency theory

Primitive	Meaning	Example
ATRANS	Transfer of an abstract relationship (Abstract TRANSfer)	give
PTRANS	Transfer of the physical location of an object (Physical TRANSfer)	go
PROPEL	Application of physical force to an object (PROPELling an object by applying physical force)	push
MOVE	Movement of a body part by it's owner (MOVEment)	kick
GRASP	Grasping of an object by an actor (GRASPing)	throw
INGEST	Ingesting of an object by an animal (INGESTing)	eat
EXPEL	Expulsion of something from the body of an animal (EXPElling)	cry
MTRANS	Transfer of mental information (Mental TRANSfer)	tell
MBUILD	Building new information out of old (Mentally BUILDing)	decide

SPEAK	Production of sounds (SPEAKing by producing sounds)	say
ATTEND	Focusing of sense organ toward a stimulus (ATTENDing to a stimulus)	listen

3. Conceptual tenses

There are definite symbols in CD theory to indicate the tenses of verbs like, for example, "p" for past tense. The available symbols are given in Table 6.8.

Symbol	Tense	Symbol	Tense
p	Past	?	Interrogative
f	Future	/	Negative
t	Transition	nil	Present
t_s	Start transition	delta	Timeless
t_f	Finished transition	c	Conditional
k	Continuing		

Table 6.8: Conceptual tenses in conceptual dependency theory

4. Indicators of dependencies (arrows)

Arrow type	Meaning
$\longrightarrow, \longleftarrow$	Direction of dependency
$\Longrightarrow$	Two-way links between the actor (PP) and the action (ACT)
$\Rrightarrow$	Dependency between PP and PA

Table 6.9: Meanings of arrows in conceptual dependency theory

5. Symbols for conceptual cases

Symbol	Meaning
o	Objective case
R	Recipient case
I	Instrumental case (e.g., eat with a spoon)
D	Directive case (e.g., going home)

Table 6.10: Symbols above arrows

6. States

States of objects are described by scales which have numerical values. For example, the state of health of a person is represented by a numerical value which goes from -10 to

+10, "−10" indicating "dead" and "+10" indicating "perfect health". The intermediate values may represent various states of health as indicated below:

State of health	**Numerical value**
dead	−10
gravely ill	−9
sick	−9 to −1
all right	0
tip top	+7
perfect health	+10

Various other states like the states of "fear", "anger", "mental state" (sad, happy, etc.), hunger, etc. have also been defined the CD theory with appropriate numerical values.

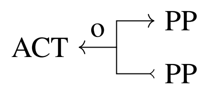
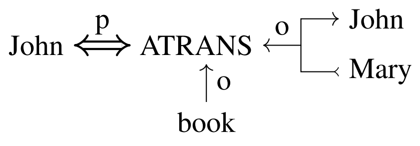
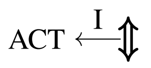
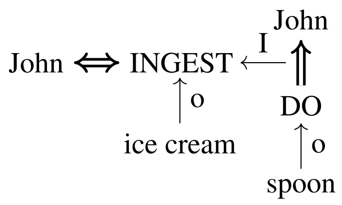
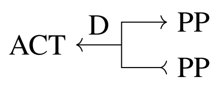
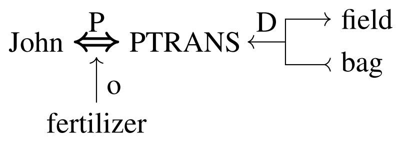
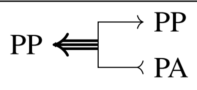
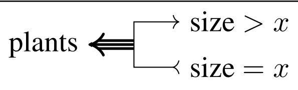
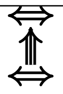
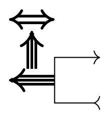
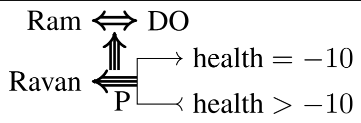
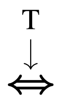
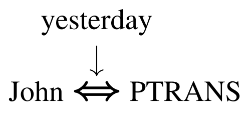
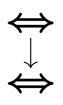
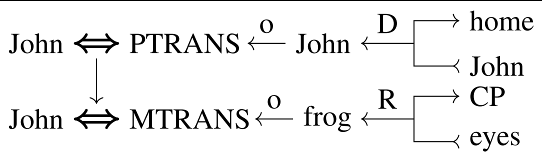
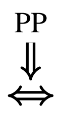
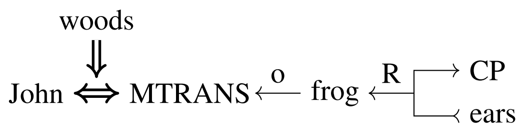
There are also states that have absolute measures like, "length", "mass", "weight", "speed", "size", etc. Further, there are also states which indicate relationships between objects like "possession" indicated by "POSS" in CD theory.

6.9.3 Rules

The conceptual categories combine in certain specified ways. The rules for combination of conceptual categories are called the conceptual syntax rules. There are used to represent natural language statements. The various rules are summarised in Table 6.11.

Table 6.11: Rules in CD theory

Sl. No.	**Rule**	**Usage**	**Meaning**
1	$\text{PP} \Longleftrightarrow \text{ACT}$	$\text{John} \overset{\text{P}}{\Longleftrightarrow} \text{PTRANS}$	John ran.
2	$\text{PP} \Lleftarrow\!\!\Rrightarrow \text{PA}$	$\text{John} \Lleftarrow\!\!\Rrightarrow \text{height} (> \text{average}$	John is tall.
3	$\text{PP} \Lleftarrow\!\!\Rrightarrow \text{PA}$	$\text{John} \Lleftarrow\!\!\Rrightarrow \text{doctor}$	John is a doctor.
4	$\begin{array}{c}\text{PP}\\ \uparrow \\ \text{PA}\end{array}$	$\begin{array}{l}\text{boy}\\ \uparrow \\ \text{nice}\end{array}$	A nice boy
5	$\begin{array}{c}\text{PP}\\ \Uparrow \\ \text{PP}\end{array}$	$\begin{array}{l}\text{dog}\\ \Uparrow\text{POSS-BY} \\ \text{John}\end{array}$	John's dog (POSS-BY denotes "possession by".)
6	$\text{ACT} \overset{\text{o}}{\longleftarrow} \text{PP}$	$\text{John} \overset{\text{p}}{\Longleftrightarrow} \text{PROPEL} \overset{\text{o}}{\longleftarrow} \text{cart}$	John pushed the cart.

7			John took the book from Mary.
8		 John INGEST I John DO o ice cream o spoon	John ate ice cream with a spoon.
9			John fertilized the field ("D" denotes the directive case.)
10			The plants grew.
11			Bill shot Bob.
12			Ram killed Ravan. (Here "DO" an unspecified activity.)
13			John ran yesterday. ("T" denotes time.)
14		 John PTRANS o John D home John John MTRANS o frog R CP eyes	While going home, John saw a frog. (Here "CP" denotes "Conceptual Processor" which is a symbol for an unspecified object.)
15			John heard a frog in the woods.

6.9.4 Advantages and disadvantages of the CD theory

Advantages

1. CD theory gives a representation of sentences which is independent of the words used in the statement.
2. CD can act as Interlingua and can facilitate translation from one language to another based on the idea of the statement and not just translation of the words used in the statement.
3. We can fill missing pieces in the fixed structures by fetching them from the context.

Disadvantages

1. It is only a theory of representation of events.
2. CD theory requires that all knowledge be decomposed into low-level primitives. This may be impossible in certain situations.
3. The CD representation is sometimes very inefficient. To represent a sentence it may take several pages of CD diagrams.
4. Complex representations require a lot of storage.
5. There are no representations of relationships which can help us make inferences from the links.
6. The set of 11 primitive acts is incomplete. All knowledge should be expressed into these acts which is sometimes inefficient and even impossible.
7. Many concepts are not recognized in CD.
8. In the generalized representation, the finer meaning is lost. For example there is a difference between giving and gifting something which is not captured by CD.
9. It is difficult to construct the original sentence from its CD representation.

6.10 Sample questions

(a) Short answer questions

1. Define a semantic network and give an example.
2. What are the different types of knowledge?
3. What is meant by a knowledge representation system? Name some knowledge representation systems.
4. State some of the desirable properties of knowledge representation systems.
5. Explain the intersection search method.

6. Define a frame and give an example.
7. What makes the conceptual dependency theory important?
8. Give a simple example for representation of sentences in CD theory.
9. Give three primitive acts in CD theory with meanings and examples.
10. How are states of objects represented in CD theory?
11. Give two examples of rules in CD theory.
12. Construct semantic net representations of each of the following:
    (a) Raja is a bank manager.
    (b) Raja works in SBI located in Defence Colony.
    (c) Raja is 26 years old.
    (d) Raja has blue eyes.
    (e) Raja is senior in service to John.
13. Construct semantic net representations of each of the following:
    (a) Dave is Welsh, Dave is a lecturer.
    (b) Paul leant his new Compact Disc to his best friend.
14. Give a CD representation of the sentence: “I gave a book to the man.”
15. Give a CD representation of the sentence: “Mohan took the book from Laila.”
16. Express the statement “He gave the pen” through conceptual dependency.
17. What are the kinds of knowledge represented in AI systems?
18. List some frame languages in AI.

(b) Long answer questions

1. Describe the building blocks of the CD theory.
2. Draw a semantic network representing the following knowledge:

   “Every vehicle is a physical object. Every car is a vehicle. Every car has four wheels. Electrical system is a part of car. Battery is a part of electrical system. Pollution system is a part of every vehicle. Vehicle is used in transportation. Swift is a car.”
3. Describe the different types of semantic networks with examples.
4. What are the advantages and disadvantages of the CD theory.
5. Represent the relationships between quadrangle, parallelogram, rhombus, rectangle and square in the form of a semantic network.

6. Represent the following knowledge in a semantic network:

Dogs are Mammals	Birds have Wings	Mammals are Animals
Bats have Wings	Birds are Animals	Bats are Mammals
Fish are Animals	Dogs chase Cats	Worms are Animals
Cats eat Fish	Cats are Mammals	Birds eat Worms
Cats have Fur	Fish eat Worms	Dogs have Fur

7. Represent the following sentences by a semantic network:

Birds are animals.	Birds have feathers, fly and lay eggs.
Albatros is a bird.	Donald is a bird.
Tracy is an albatros.	

8. Represent the following sentences by a semantic network:

Palco is a calico.	Herb is a tuna.
Charlie is a tuna.	All tunas are fishes.
All calicos are cats.	All cats like to eat all kinds of fishes.

9. Represent the following sentences by a semantic network:

Circus elephants are elephants.	
Elephants have heads.	Elephants have trunks.
Heads have mouths.	Elephants are animals.
Animals have hearts.	Circus elephants are performers.
Performers have costumes.	Costumes are cloths.
Horatio is a circus elephant.	

10. Represent the following sentences by a semantic network:

    Every truck is a vehicle.

    Every trailer truck is a truck that has

      as part a trailer,

      an unloaded weight, which is a weight measure,

      a maximum gross weight, which is a weight measure,

      a cargo capacity, which is a volume measure,

      and a number of wheels, which is the integer 18.

11. What is conceptual dependency? How do you represent conceptual dependency?

12. What is CD? List primitive acts with meaning.

13. What do you understand by conceptual dependency? Give the conceptual dependency representations of the following sentences:
    (a) John shot Mary.
    (b) John fertilized the field.
    (c) Mary drove her car to the office.
14. Represent the following sentences using conceptual dependency diagrams.
    (a) John pushed the cart.
    (b) The plants grew.
    (c) John ran yesterday.
    (d) John is a teacher.
    (e) While going home, I saw a frog.
15. Explain the primitive conceptual categories in CD theory.

Chapter 7

Knowledge representation using logic

In the last chapter we discussed knowledge representation using semantic networks and conceptual dependency diagrams. In the present chapter we consider a third method of knowledge representation, namely, representation using logic, especially first order predicate logic.

In the first section we give a quick review of the basic ideas of propositional logic as a preparation for studying first order predicate logic. The chapter concludes with a summary of the various knowledge representation methods.

7.1 Propositional logic

7.1.1 Propositions

A **proposition** is a statement which is either true or false but not both. If a proposition is true we say that its **truth value** is "True" and if the proposition is false, we say its truth value is "False". The truth value "True" is also denoted by T or 1 and the truth value "False" is sometimes denoted by F or 0.

Examples

1. The statement "New Delhi is the capital of India." is a proposition and its truth value is True,
2. The statement "India became a republic in 1947." is a proposition and its truth value is False.
3. The sentence "It may rain tomorrow." is not a proposition because it has no definite truth value.

Propositions are denoted by capital letters like A, P, X, etc.

7.1.2 Operations on propositions: Logical operators

Complex statements can be constructed by applying operations on propositions. Such statements are called propositional formulas or propositional expressions. A table giving the truth values of a propositional expression is called the **truth table** of the expressions. The basic operations on propositions are described below.

1. Negation

The **negation** of a proposition P is denoted by $\neg P$. By definition, the truth value of $\neg P$ is False if the truth value of P is True and the truth value of $\neg P$ is True if the truth value of P is false. The truth table of $\neg P$ is exhibited in Table 7.1.

P	$\neg P$
T	F
F	T

Table 7.1: Truth table of negation

2. Disjunction

The **disjunction** of two propositions P and Q is denoted by $P \vee Q$. The truth value of $P \vee Q$ is True if P is True, or Q is True or both P and Q are True. The truth value of $P \vee Q$ is False in all other cases. The truth values are given Table 7.2.

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

Table 7.2: Truth table of disjunction

3. Conjunction

The **conjunction** of two propositions P and Q is denoted by $P \wedge Q$. The truth value of $P \wedge Q$ is True only if both P and Q are True. The truth value of $P \vee Q$ is False in all other cases. The truth values are given Table 7.3.

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Table 7.3: Truth table of conjunction

4. Implication

The **implication** of proposition Q from P is denoted by $P \Rightarrow Q$. The truth value of $P \Rightarrow Q$ is False only if P is True and Q is False. The truth value of $P \Rightarrow Q$ is True in all other cases. The truth values are given Table 7.4.

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

Table 7.4: Truth table of implication

5. Two-way implication (or, biconditional)

The **two-way implication** of P from Q is denoted by $P \Leftrightarrow Q$. The truth value of $P \Leftrightarrow Q$ is True only if both P and Q have the same truth values. The truth values are given Table 7.5.

P	Q	$P \Leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

Table 7.5: Truth table of implication

7.1.3 Tautologies and contradictions

A logical expression is called a **tautology** if its truth value is always True whatever be the truth values of its component propositions. A logical expression is called a **contradiction** if its truth value is always False whatever be the truth values of its component propositions.

Examples

For example it can be seen that the expression $(\neg P) \vee (P \vee Q)$ is a tautology. We prove this by constructing a truth table of the expression.

P	Q	$\neg P$	$P \vee Q$	$(\neg P) \vee (P \vee Q)$
T	T	F	T	T
T	F	F	T	T
F	T	T	T	T
F	F	T	F	T

Table 7.6: Truth table of $(\neg P) \vee (P \vee Q)$

Also it can be seen that $\neg P \wedge (P \wedge Q)$ is a contradiction from Table 7.7.

P	Q	$\neg P$	$P \wedge Q$	$\neg P \wedge (P \wedge Q)$
T	T	F	T	F
T	F	F	F	F
F	T	T	F	F
F	F	T	F	F

Table 7.7: Truth table of $(\neg P) \wedge (P \wedge Q)$

7.1.4 Identities

Two logical expressions $\mathbf{X}$ and $\mathbf{Y}$ are said to be **logically equivalent** if the expression $\mathbf{X} \Leftrightarrow \mathbf{Y}$ is a tautology. If $\mathbf{X}$ and $\mathbf{Y}$ are logically equivalent we write $\mathbf{X} \equiv \mathbf{Y}$ and say that $\mathbf{X} \equiv \mathbf{Y}$ is a logical identity.

Example

Prove the identity:

$$(P \Rightarrow Q) \equiv ((\neg P) \vee Q)$$

We have to show that the expression

$$(P \Rightarrow Q) \Leftrightarrow ((\neg P) \vee Q)$$

is a tautology. This we do by constructing the truth table.

P	Q	$P \Rightarrow Q$	$\neg P$	$(\neg P \vee Q)$	$(P \Rightarrow Q) \Leftrightarrow ((\neg P) \vee Q)$
T	T	T	F	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

Table 7.8: Truth table of $(P \Rightarrow Q) \Leftrightarrow ((\neg P) \vee Q)$

Some important identities

Let P, Q, R, S be arbitrary propositions and let $\mathbf{T}$ denote a proposition whose truth value is True and $\mathbf{F}$ one whose truth value is False. Then we have the following identities.

1. $P \vee (\neg P) \equiv \mathbf{T}$
2. $P \wedge (\neg P) \equiv \mathbf{F}$
3. $P \vee P \equiv P$
4. $P \wedge P \equiv P$
5. $P \vee Q \equiv Q \vee P$
6. $P \wedge Q \equiv Q \wedge P$
7. $P \vee (Q \vee R) \equiv (P \vee Q) \vee R$
8. $P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$
9. $(P \Rightarrow Q) \equiv ((\neg P) \vee Q)$
10. $\neg(P \vee Q) \equiv (\neg P) \wedge (\neg Q)$ (De Morgan's law)
11. $\neg(P \wedge Q) \equiv (\neg P) \vee (\neg Q)$ (De Morgan's law)

7.1.5 Inferences

Arguments

An **argument** in propositional logic is a sequence of propositions. All but the final proposition are called premises and the final proposition is called the conclusion. An argument is valid if the truth of all its premises implies that the conclusion is true.

An example of an argument is given in Figure 7.1. It means that if the statements $P \vee Q$, $P \Rightarrow R$ and $Q \Rightarrow R$ are true then R is also true.

$$\begin{array}{lcl} & & P \vee Q \\ \text{Premises} & & P \Rightarrow R \\ & & \underline{Q \Rightarrow R} \\ \text{Conclusion} & \therefore & R \end{array}$$

Figure 7.1: An argument in propositional logic

Proof of validity of an argument

A **proof of validity** of a given argument is a sequence of statements, each of which is either a premise of that argument or follows from preceding statements of the sequence by an elementary valid argument, such that the last statement in the sequence is the conclusion of the argument whose validity is being proved.

7.1.6 Rules of inference

The elementary valid arguments that are used in constructing proofs of validity of arguments are called the rules of inference. Rules of inference are templates for building valid arguments.

Let us create a rule of inference. Constructing truth tables, one can prove that the following is a tautology:

$$(P \wedge (P \Rightarrow Q)) \Rightarrow Q$$

This means that if P and $P \Rightarrow Q$ are true then Q is also true which in turn implies that the following argument is valid:

$$\begin{array}{rl} & P \\ & P \Rightarrow Q \\ \hline \therefore & Q \end{array}$$

This is a well known rule of inference is known as “modus-ponens”. Table 7.9 lists the various rules of inference and their names.

Example of proof of validity

Consider the argument given in Figure 7.1. The proof of the validity of this argument is shown in Table 7.10.

Step no.	Formula	Derivation
1	$P \vee Q$	Premise
2	$P \Rightarrow R$	Premise
3	$Q \Rightarrow R$	Premise
4	$\neg P \vee R$	Step 2, logical identity
5	$Q \vee R$	Steps 1,4 (Resolution)
6	$\neg Q \vee R$	Step 3, logical identity
7	$R \vee R$	Steps 5, 6 (Resolution)
8	R	Step 7, logical identity

Table 7.10: Proof of validity of the argument in Figure 7.1

7.1.7 Conjuctive normal form

The atomic sentences in propositional logic consist of a single proposition symbol. A literal is either an atomic sentence (a positive literal) or a negated atomic sentence (a negative literal). A clause is a disjunction of literals. A sentence expressed as a conjunction of clauses is said to be in conjunctive normal form (CNF) or clause form.

Sl. No.	Inference rule	Name
1	$\begin{array}{rl} & P \\ & P \Rightarrow Q \\ \hline \therefore & Q \end{array}$	Modus ponens
2	$\begin{array}{rl} & \neg Q \\ & P \Rightarrow Q \\ \hline \therefore & \neg P \end{array}$	Modus tollens
3	$\begin{array}{rl} & P \Rightarrow Q \\ & Q \Rightarrow R \\ \hline \therefore & P \Rightarrow R \end{array}$	Hypothetical syllogism
4	$\begin{array}{rl} & P \vee Q \\ & \neg P \\ \hline \therefore & Q \end{array}$	Disjunctive syllogism
5	$\begin{array}{rl} & P \\ \hline \therefore & P \vee Q \end{array}$	Addition
6	$\begin{array}{rl} & P \wedge Q \\ \hline \therefore & P \end{array}$	Simplification
7	$\begin{array}{rl} & P \\ & Q \\ \hline \therefore & P \wedge Q \end{array}$	Conjunction
8	$\begin{array}{rl} & P \vee Q \\ & \neg P \vee R \\ \hline \therefore & Q \vee R \end{array}$	Resolution

Table 7.9: Rules of inference

Examples

Sentences	Examples
Atomic sentence	$P, Q, R, \ldots$
Literals	$P, \neg P, Q, \neg Q, R, \neg R, \ldots$
Clauses	$P \wedge Q \wedge \neg R,\ \neg Q \wedge R, \ldots$
CNF	$(P \wedge Q \wedge \neg R) \vee (\neg P \wedge Q)$

Conversion to CNF

Every sentence of propositional logic is logically equivalent to a conjunction of clauses. We now describe a procedure for converting to CNF.

1. Eliminate "⇔" by replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow) \wedge (\beta \Rightarrow \alpha)$.
2. Eliminate "⇒" by replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
3. We "move ¬ inwards" by repeated application of the following equivalences:
$$\begin{aligned}\neg(\neg\alpha) &\equiv \alpha\\ \neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta)\\ \neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta)\end{aligned}$$
4. Apply the distributivity laws distributing $\vee$ over $\wedge$ wherever possible, that is, we apply the equivalence:
$$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma).$$

Example

Transform the following sentence into CNF:

$$P \Leftrightarrow (Q \vee R)$$

Solution

Given sentence:

$$P \Leftrightarrow (Q \vee R)$$

Eliminating "⇔" we get the equivalent sentence:

$$(P \Rightarrow (Q \vee R)) \wedge ((Q \vee R) \Rightarrow P)$$

Eliminating "⇒" we have

$$(\neg P \vee (Q \vee R)) \wedge (\neg(Q \vee R) \vee P)$$

Moving ¬ inwards in $\neg(Q \vee R)$ we have

$$(\neg P \vee (Q \vee R)) \wedge ((\neg Q \wedge \neg R) \vee P)$$

Using distributivity of $\vee$ over $\wedge$ in $((\neg Q \wedge \neg R) \vee P)$ we have

$$(\neg P \vee (Q \vee R)) \wedge ((\neg Q \vee P) \wedge (\neg R \vee P))$$

By associativity we have

$$(\neg P \vee Q \vee R) \wedge (\neg Q \vee P) \wedge (\neg R \vee P)$$

This is the CNF of the given sentence.

7.1.8 Resolution-refutation in propositional logic

A **proof technique** is a procedure for showing that a statement logically follows from a set of premises. **Resolution** is a proof technique that works on conjunctive normal form expression. The proof technique known as resolution depends on the the rule of inference known as "resolution" (see Table 7.9). The basic idea of resolution can be stated thus:

1. Select two clauses that contain conflicting terms (that, one term in one of the clauses is the negation of a term in the other clause).
2. Combine those two clauses.
3. Cancel out the conflicting terms.

What makes the resolution proof technique so important is that this one simple technique is capable of performing the same kinds of reasoning tasks as modus ponens, modus tollens, syllogisms and many other logical operations.

A **resolution refutation proof** is proof by contradiction using resolution. Like every proof by contradiction, we begin by assuming the opposite of what we wish to prove, and then show that this "fact" would lead to a contradiction.

The algorithm of the resolution-refutation proof technique is given below.

Algorithm: Propositional resolution

Let be required to produce a proof of a proposition P from a set of premises F.

Step 1. Convert all propositions in F to clause form.

Step 2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in Step 1.

Step 3. Repeat until either a contradiction is found or no progress can be made.

(a) Select two clauses. Call these the parent clauses.

(b) Resolve them together. The resulting clause is called the resolvent.
(The resolvent is the disjunction of all literals of both the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, select one such pair and eliminate both L and $\neg L$ from the resolvent.)

(c) If the resolvent is the empty clause then a contradiction has been found.

(d) If it is not, then add it to the set of clauses available to the procedure.

If a contradiction has been found, we conclude that P is a valid conclusion from the premises F. Otherwise, that is if the algorithm terminates in such a status that no progress can be made, we conclude that P is not a valid conclusion from F.

Example

Using resolution-refutation determine whether the argument shown in Figure 7.1 is valid.

Solution

The premises are

$$P \vee Q, \quad P \Rightarrow R, \quad Q \Rightarrow R,$$

Converting these to the clause form we get

$$P \vee Q, \quad \neg P \vee R, \quad \neg Q \vee R.$$

We now add the negation of the conclusion to the premises:

$$P \vee Q, \quad \neg P \vee R, \quad \neg Q \vee R, \quad \neg R.$$

The various stages in the resolution are shown in Table 7.11.

Step no.	Formula	Derivation
1	$P \vee Q$	Premise
2	$\neg P \vee R$	Premise
3	$\neg Q \vee R$	Premise
4	$\neg R$	Negated conclusion
5	$Q \vee R$	Steps 1, 2 (Resolution)
6	$\neg Q$	Steps 3, 4 (Resolution)
7	R	Steps 5, 6 (Resolution)
8	Empty clause	Steps 4, 7 (Resolution)

Table 7.11: Proof by resolution-refutation of the argument in Figure 7.1

7.2 First order predicate logic (FOPL)

In propositional logic we are concerned with the truth values of stand alone statements like “New Delhi is capital of India” and “105 is not a prime number” and of complex statements obtained by applying various logical connectives to such sentences. We are also concerned with the different inference rules which will help us derive new sentences from given premises.

In **first order predicate logic**, also called simply **predicate logic**, we are concerned with statements which will be applicable to the elements in some set of objects and about the truth values of such statements. For example, consider the set of children shown in the Figure 7.2. With reference to this set of children, we may may like to know the truth values of the following statements:

“All children wear boots.”

“All children are boys.”

“There is a girl among the children.”

The predicate logic is a theory for studying such problems.

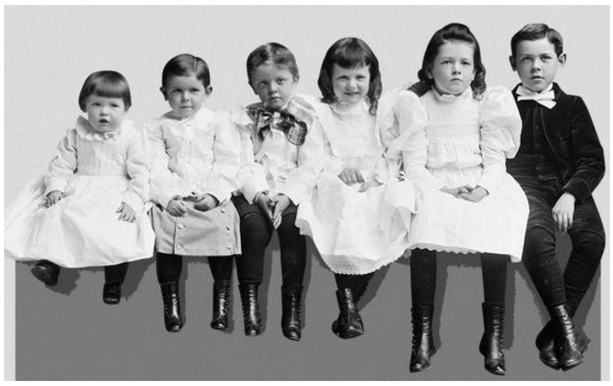


Figure 7.2: A domain whose objects are children

To study the first of the three sentences given above, we consider the sentence "x wears boots". Here we assume that x is the name of an unspecified arbitrary child in the photograph. As such it has no truth value because we have not specified what x is. If we replace x by the name of a child in the photograph, say "John", we get the sentence "John wears boots." which has a truth value namely True. The statement "All children wear boots" is true if all sentences obtained by replacing x in "x wears boots" by the names of each and every child in the photograph is true. This is indeed the case and so we conclude that the statement "All children wear boots" is true.

To discuss the the third of the three statements, we consider the sentence "x is a girl" For the statement "There is a girl among the children" to be true it is enough that there is at least one name say "Sara" among the set of names of children which makes the sentence "Sara is a girl" true. This is indeed the case and so the truth value of the sentence "Thers is a girl among the children" is True.

Sentences like "x wears boots" and "x is a girl" are called **predicates**. The predicate "x wears boots" may be symbolically represented as "$wearsboots(x)$", or "$WearsBoots(x)$" or simply "$W(x)$". Similarly, the sentence "x is a girl" may be represented as "$girl(x)$" or "$Girl(x)$" or simply "$G(x)$".

Why *first order?*

The "first order predicate logic" is so called only to distinguish it from the "second order predicate logic". Second-order logic was introduced by Frege in 1879 who also coined the term "second order". In first order predicate logic, we study truth values of predicates which are essentially statements of properties of elements of sets. In the second order logic, we study properties of sets of such properties of sets. To see an example of such a property, let $P(x)$ be some arbitrary predicate. Then the following is a statement in second order logic:

"For all predicates P, and for each object x,
either $P(x)$ is true or $\neg\,(Px)$ is true."

In these notes we are concerned with only first order logic.[1]

[1]For a detailed introduction to second order logic, one may refer to "*Second-order and Higher-order Logc*", Stanford Encyclopedia of Philosophy, available ar `https://plato.stanford.edu/entries/logic-higher-order/`.

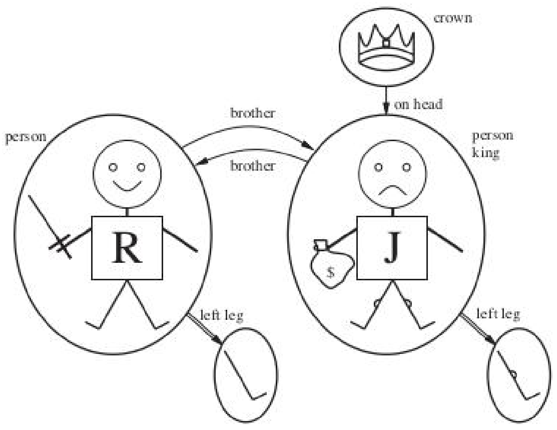


Figure 7.3: A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

7.2.1 Domain of discourse

In discussing FOPL, we assume a certain **domain of discourse**, or simply a domain (also called a "**world**"or a "**universe**"). The domain contains all the objects under consideration. The predicate logic requires that the domain be non-empty, that is, that it contains at least one object.

The domain is completely arbitrary. For example, Figure 7.3 shows a domain with five objects, namely, King Richard, King John, the left legs of Richard and John and a crown.

7.2.2 Predicates

A **predicate** is a statement (specifying some relation) that contains variables (predicate variables), and they may be true or false depending on the values of these variables. The domain is the set of values that variables can take.

Definition

A **predicate** is a sentence that contains a finite number of variables and becomes a statement when specific values are substituted for the variables.

Examples

The following are predicates in appropriate domains:

1. "x is a boy."
2. "x wears boots."
3. "x is a king."

7.2.3 Basic elements

- The basic elements of first-order logic are the symbols that stand for **objects**, **relations**, and **functions**.

- The symbols are of three kinds: constant symbols, which stand for objects; predicate symbols, which stand for relations; and function symbols, which stand for functions.
- Each predicate and function symbol comes with an **arity** that fixes the number of arguments.

A predicate with arity 1 is called a **unary** predicate. The predicate “x is a king” is unary predicate. A predicate with arity 2 is called a **binary predicate**. The predicate “x is a brother of y” is a binary predicate. In general, a predicate with arity n is called a n-ary predicate.

Notation

We adopt the convention that the symbols begin with uppercase letters. For example, for the world specified in Figure 7.3, we might use the constant symbols *Richard* and *John*; the predicate symbols *Brother*, *OnHead*, *Person*, *King*, and *Crown*; and the function symbol *LeftLeg*. The choice of names is entirely up to the user.

Terms and atomic sentences

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms. An atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as $Brother(Richard, John)$.

7.3 Quantifiers

7.3.1 Universal quantification

Consider the statement:

“All kings are persons."

To write this in the notations of FOPL, we introduce two predicates as follows:

“$King(x)$” denoting “x is a king.”
“$Person(x)$” denoting “x is a person.”

In the notations of first-order logic we write the sentence “All kings are persons.” as follows:

$$\forall x\ King(x) \rightarrow Person(x)$$

The symbol $\forall$ is usually pronounced “For all . . .” and is called the **universal quantifier**. Thus, the above sentence says:

“For all x, if x is a king, then x is a person.âĂİ

The symbol x is called a variable. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function - for example, $LeftLeg(x)$. A term with no variables is called a ground term.

Intuitively, the sentence $\forall x\ P$, where P is any logical expression, says that P is true for every object x.

7.3.2 Existential quantification

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 consider the statement:

King John has a crown on his head.

We need two predicates:

$Crown(x)$	:	"x is a crown"
$OnHead(x, John)$	:	"x is on the head of John."

To say that "King John has a crown on his head", we write

$$\exists\, x\ Crown(x) \land OnHead(x, John)$$

$\exists x$ is pronounced "There exists an x such that . . ." or "For some x . . .". The symbol "$\exists$" is called the **existential quantifier**. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence "King John has a crown on his head."

Intuitively, the sentence $\exists\, x\ P$ says that P is true for at least one object x.

7.3.3 Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. We illustrate the idea by examples:

- "Brothers are siblings" can be written as
$$\forall x\, \forall y\ Brother(x, y) \Rightarrow Sibling(x, y)$$
- "Everybody loves somebody" means that for every person, there is someone that person loves:
$$\forall x \exists y\, Loves(x, y)$$
- The statement "There is someone who is loved by everyone" we write as
$$\exists y \forall x\ x\ Loves(x, y).$$

The order of quantification is very important.

7.3.4 Connections between $\forall$ and $\exists$

The two quantifiers $\forall$ and $\exists$ are intimately connected with each other, through negation. We have the following logical equivalences:

1. $\forall x\ \neg P \equiv \neg\exists x\ P$
2. $\neg\forall x\ P \equiv \exists x\ \neg P$
3. $\forall x\ P \equiv \neg\exists x\ \neg P$
4. $\exists x\ P \equiv \neg\forall x\ \neg P$

7.3.5 Examples of quantified statements

We now consider several examples of quantified statements. In these examples, the meanings of the predicates are obvious from the contexts.

1. "If something is a cube, it is not a tetrahedron."

   $\forall x\ Cube(x) \Rightarrow \neg\, Tetrahedron(x)$

2. "Everybody loves a lover."

   $\forall x\ \forall y\ [Person(x) \wedge Lover(y) \Rightarrow L(x,y)]$

   where $L(x,y)$ denotes x loves y.

3. "If two persons speak the same language, they understand each other."

   $\forall x, y, l\ SpeaksLanguage(x,l) \wedge SpeaksLanguage(y,l)$

   $\Rightarrow Understands(x,y) \wedge Understands(y,x)$

4. "There is a barber who shaves all men in town who do not shave themselves."

   $\exists x\ \big(Barber(x) \wedge (\forall y\ Man(y) \wedge \neg\, Shaves(y,y) \Rightarrow Shaves(x,y))\big)$

5. "Every person who buys a policy is smart."

   $\forall x\ Person(x) \wedge (\exists y, z\ Policy(y) \wedge Buys(x,y,z)) \Rightarrow Smart(x)$

   ($Buys(x,y,z)$ denotes x buys y from z.)

6. "Only one student took French as elective."

   $\exists x\ Student(x) \wedge Takes(x) \wedge (\forall y\ (y \neq x) \Rightarrow \neg\, Takes(y)$

   ($Takes(x)$ means x takes French as elective.)

7. "The best score in French is always higher than the best score in German in all semesters."

   $\forall s\ \exists x\ \forall y\ Score(x, \text{French}, s) > Score(y, \text{German}, s)$

   ($Score(x,g,s)$ denotes the score obtained by student x in subject g in semester s.)

8. "A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth."

   $\forall x\ \Big(Person(x) \wedge Born(x) \wedge$

   $\big(\forall y\ Parent(y,x) \Rightarrow ((\exists r\ Citizen(y,r) \vee Resident(y)))\Big) \Rightarrow Citizen(x, \text{Birth})$

   ($Parent(x,y)$ denotes x is a parent of y and $Citizen(x,r)$ denotes x is a citizen of UK by reason r.)

9. "Everyone who loves all animals is loved by someone"

   $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$

10. "Every farmer who owns a donkey beats it."

    $\forall x\ \big(Farmer(x) \Rightarrow \forall y\ ((Donkey(y) \wedge Owns(x,y)) \Rightarrow Beats(x,y))\big)$

7.4 Well-formed formulas in FOPL

7.4.1 Well-formed formulas

Well-formed formulas (wffs) in FOPL are constructed using the following rules:

1. True and False are wffs.
2. Each propositional constant (i.e. specific proposition) is a wff
3. Each atomic formula (i.e. a specific predicate with variables) is a wff.
4. If A and B are wffs, then so are $\neg A$, $(A \vee B)$, $(A \wedge B)$, $(A \Rightarrow B)$, and $A \Leftrightarrow B$.
5. If x is a variable (representing objects of the universe of discourse), and A is a wff, then so are $\forall x \ A$ and $\exists x \ A$.

7.4.2 Clausal forms (also called conjuctive normal forms (CNFs))

1. A literal is either an atomic formula or the negation of an atomic formula.
2. A clause is the disjunction of literals.
3. A formula in clausal form (or, CNF) consists of a conjunction of clauses.

7.4.3 Algorithm to convert a wff to the clausal form (or, CNF)

1. **Eliminate the two-way implication (biconditional) $\Leftrightarrow$**

   Replace $a \Leftrightarrow b$ by $(a \Rightarrow b) \wedge (b \Rightarrow a)$.

2. **Eliminate implications**

   Replace $a \Rightarrow b$ by $\neg a \vee b$.

3. **Move $\neg$ inwards**

   Make replacements as follows:

   $\neg(a \vee b)$ by $\neg a \wedge \neg b$

   $\neg(a \wedge b)$ by $\neg a \vee \neg b$

   $\neg \forall x \ P$ by $\exists x \ \neg P$

   $\neg \exists x \ P$ by $\forall x \ \neg P$

4. **Standardize variables**

   Change variables so that each quantifier binds a unique variable.

   For example, for sentences like $(\exists x \ P(x)) \vee (\exists x \ Q(x))$ which use the same variable name twice, change the name of one of the variables.

5. **Skolemize**[2]

   Skolemization is the process of removing existential quantifiers by elimination.

[2]The terminology is in honour of Thoralf Albert Skolem (1887 - 1963), a Norwegian mathematician who worked in mathematical logic and set theory.

(a) Existentially quantified variables which are not inside the scope of a universal quantifier are replaced by creating new constants. For example, $\exists\, x\ P(x)$ is changed to $P(c)$, where c is a new constant.

(b) Replace every existentially quantified variable y with a term $f(x_1, \ldots, x_n)$ whose function symbol f is new. The variables of this term are the variables that are universally quantified and whose quantifiers precede that of y.

- For example, in the formula $\forall\, x\, \exists\, y\, \forall\, z\ P(x, y, z)$, Skolemization replaces y with $f(x)$, where f is a new function symbol, and removes the quantification over y. The resulting formula is $\forall x\, \forall z\ P(x, f(x), z)$. The Skolem term $f(x)$ contains x but not z, because the quantifier to be removed $\exists\, y$ is in the scope of $\forall\, x$, but not in that of $\forall z$
- As a second example, when we Skolemize

$$\forall\, x\, \forall\, y\, \exists\, z\, \forall\, u\, \exists\, v\ P(x, y, z, u, v)$$

we get

$$\forall\, x\, \forall\, y\, \forall\, u\ P(x, y, F(x, y), u, G(x, y, u))$$

where $F(x, y)$ and $G(x, y, u)$ are Skolem functions.

The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears.

6. **Remove universal quantifiers**

   At this point, all remaining variables will be universally quantified. We now drop the universal quantifiers.

7. **Distribute $\vee$ over $\wedge$**

   Convert the resulting expression into a conjunction of disjunctions by using distributivity.

   Replace $a \vee (b \wedge b)$ by $(a \vee b) \wedge (a \vee b)$.

8. **CNF of the given wff**

   The sentence is now in CNF.

7.4.4 Examples

Example 1

Convert the following wff to the clausal form:

$$\forall\, x\ [\forall\, y\ A(y) \Rightarrow L(x, y)] \Rightarrow [\exists\, y\ L(y, x)]$$

Solution

Step 1. Eliminate implications

$\forall\, x\ \neg[\forall\, y\ \neg A(y) \vee L(x, y)] \vee [\exists\, y\ L(y, x)]$

that is,

$\forall\, x\ [\neg\forall\, y\ \neg A(y) \vee L(x, y)] \vee [\exists\, y\ L(y, x)]$

Step 2. Move $\neg$ inwards

The sentence goes through the following transformations:

$\forall x\ [\exists y\ \neg(\neg A(y) \vee L(x,y))] \vee [\exists y\ L(y,x)]$

$\forall x\ [\exists y\ \neg\neg A(y) \wedge \neg L(x,y)] \vee [\exists y\ L(y,x)]$

$\forall x\ [\exists y\ A(y) \wedge \neg L(x,y)] \vee [\exists y\ L(y,x)]$

Step 3. Standardize variables

$\forall x\ [\exists y\ A(y) \wedge \neg L(x,y)] \vee [\exists z\ L(z,x)]$

Step 4. Skolemize

$\forall x\ [A(F(x)y) \wedge \neg L(x, F(x))] \vee [L(G(x), x)]$

(F and G are Skolem functions.)

Step 5. Drop universal quantifiers

$[A(F(x)y) \wedge \neg L(x, F(x))] \vee [L(G(x), x)]$

Step 6. Distribute $\vee$ over $\wedge$

$[A(F(x) \vee L(G(x), x)] \wedge [L(x, F(x)) \vee L(G(x), x)]$

Step 7. This the CNF of the given sentence.

Example 2

Convert the following wff to the clausal form:

$$\forall x\ \Big[[E(x)] \Leftrightarrow [\forall y\ E(T(x,y))]\Big]$$

Solution

Step 1. Remove biconditional

$\forall x\ ([E(x)] \Rightarrow [\forall y\ E(T(x,y))]) \wedge ([\forall y\ E(T(x,y))] \Rightarrow [E(x)])$

Step 2. Remove implications

$\forall x\ (\neg[E(x)] \vee [\forall y\ E(T(x,y))]) \wedge (\neg[\forall y\ E(T(x,y))] \vee [E(x)])$

Step 3. Move $\neg$ inwards

$\forall x\ (\neg[E(x)] \vee [\forall y\ E(T(x,y))]) \wedge ([\exists y\ \neg E(T(x,y))] \vee [E(x)])$

Step 4. Standardize variables

$\forall x\ (\neg[E(x)] \vee [\forall y\ E(T(x,y))]) \wedge ([\exists z\ \neg E(T(x,z))] \vee [E(x)])$

Step 5. Skolemize

$\forall x\ (\neg[E(x)] \vee [\forall y\ E(T(x,y))]) \wedge ([\neg E(T(x, F(x)))] \vee [E(x)])$

where $F(x)$ is a Skolem function.

Step 6. Drop universal quantifiers

$(\neg[E(x)] \vee [E(T(x,y))]) \wedge ([\neg E(T(x, F(x)))] \vee [E(x)])$

Step 7. This is the CNF of the given sentence.

Example 3

Transform to the CNF the following wff:

$$\begin{aligned}&\forall\, x\ [Romans(x) \wedge know(x, \text{Marcus})] \Rightarrow \\ &\quad [hate(x, \text{Caesar}) \vee (\forall\, y\ \exists\, z\ hate(y, z) \Rightarrow thinkcrazy(x, y))]\end{aligned}$$

This is a symbolic representation of the sentence: "All Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy." (see RKN, p.109) To simplify the writing of the various steps we shall write R for "Romans", K for "know", H for "hate", T for "thinkcrazy", M for "Marcus" and C for "Caesar".

In the simplified notations, the given sentence can be written as

$$\forall\, x\ [R(x) \wedge K(x, M)] \Rightarrow [H(x, C) \vee (\forall\, y\ (\exists\, z\ H(y, z)) \Rightarrow T(x, y))]$$

Solution

Step 1. Remove implications

$\forall\, x\ \neg[R(x) \wedge K(x, M)] \vee [H(x, C) \vee (\forall\, y\ \neg\, (\exists\, z\ H(y, z)) \vee T(x, y))]$

Step 2. Move $\neg$ inwards

$\forall\, x\ [\neg R(x) \vee \neg K(x, M)] \vee [H(x, C) \vee (\forall\, y\ (\forall\, z\ \neg H(y, z)) \vee T(x, y))]$

Step 3. Standardise variables

The variables are already standardised.

Step 4. Skolemize

No change.

Step 5. Remove universal quantifiers

$[\neg R(x) \vee \neg K(x, M)] \vee [H(x, C) \vee ((\neg H(y, z)) \vee T(x, y))]$

Step 6. This is the CNF of the given sentence.

7.5 Inference rules in FOPL

7.5.1 Rules borrowed from propositional logic

All the inference rules for propositional logic (see Tables 7.9) like modus ponens, modus tollens, resolution, etc. can be applied in FOPL as well.

7.5.2 Universal instantiation (UI)

The rule of Universal Instantiation (UI for short) says that we can infer any sentence obtained by substituting a ground term (a term without variables) for the variable in a predicate formula involving universal quantifiers.

In short, UI says that if $\forall\, x\ P(x)$ is true then $P(c)$ is true, where c is a constant in the domain.

Example

Consider the following predicate formula:

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$

If this is true then we can infer that the following are all true.

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

7.5.3 Existential instantiation (EI)

The rule for Existential Instantiation (EI for short) says that in a predicate formula involving existential quantifiers, the variable may be replaced by a single new constant symbol.

In short, UI says that if $\exists x\ P(x)$ is true then $P(k)$ is true, where k is a new constant.

Example

For example, from the sentence

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

we can infer the sentence

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

as long as C does not appear elsewhere in the knowledge base.

7.5.4 Generalised modus ponens (GMP)

Given atomic sentences $P_1, P_2, \ldots, P_N$, and implication sentence

$$(Q_1 \wedge Q_2 \wedge \cdots \wedge Q_N) \Rightarrow R,$$

where $Q_1, Q_2, \ldots, Q_N$ and R are atomic sentences, and

$$subst(\theta, P_i) = subst(\theta, Q_i) \text{ for } i = 1, ..., N,$$

we can derive the new sentence:

$$Subst\,(\theta, R)$$

where $Subst\,(\theta, R)$ denotes the result of applying a set of substitutions defined by θ to the sentence R.

Remarks

1. Recall the modus ponens inference rule of propositional logic:

$$\begin{array}{ll} & P \\ & P \Rightarrow Q \\ \hline \therefore & Q \end{array}$$

2. A substitution list is usually written in the following form:

$$\theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$$

   This means that we are required to replace all occurrences of variable symbol v_i by term t_i. Substitutions are made in left-to-right order in the list.

   For example, let the substitutions be

$$\theta = \{x/\text{IceCream}, y/\text{Ziggy}\}$$

   and the statement be

$$R = eats\,(x, y)$$

   then the substitution $Subst\,(\theta, R)$ yields

$$eats\,(\text{Ziggy}, \text{IceCream}).$$

7.5.5 The resolution inference rule

To state the resolution inference rule in FOPL. we require the concept of unification.

Unification

Unification is the process of finding substitutions that make different logical expressions look identical.

For example, consider the logical expressions $P(x, y)$ and $P(a, f(z))$. If we substitute x by a and y by $f(z)$ in the first expression, the first expression will be identical to the second expression. We say that the unifier of the expressions is the set $\theta = \{a/x, f(z)/y\}$.

As another example, consider the sentences:

$$Knows(\text{John}, x), \qquad Knows(x, \text{Elizabeth})$$

Since both sentences contain the same variable, we apply standardisation and change the x in $Knows(x, \text{Elizabeth})$ to y and get the following sentences:

$$Knows(\text{John}, x), \qquad Knows(y, \text{Elizabeth})$$

Now, it is easy to see that the two sentences are unified by the substitutions

$$\theta = \{x/\text{Elizabeth}, y/\text{John}\}.$$

For unification of two sentences, the predicate symbol must be same and number of arguments must also be the same.

Outline of a procedure for unification

1. Predicate symbols must match.
2. For each pair of predicate arguments:
   (a) Different constants cannot match.
   (b) A variable may be replaced by a constant.
   (c) A variable may be replaced by another variable.
   (d) A variable may be replaced by a function as long as the function does not contain an instance of the variable.

The resolution inference rule

Given sentence $P_1 \vee \ldots \vee P_n$ and sentence $Q_1 \vee \ldots \vee Q_m$ where each P_i and Q_j is a literal, if P_j and $\neg Q_k$ unify with substitution list θ then we can derive the resolvent sentence:

$$Subst\,(\theta, P_1 \vee \ldots \vee P_{j-1} \vee P_{j+1} \vee \ldots \vee P_n \vee Q_1 \vee \ldots Q_{k-1} \vee Q_{k+1} \vee \ldots \vee Q_m)$$

Example

Consider the clauses:

$$P(x, f(a)) \vee P(x, f(y)) \vee Q(y), \quad \text{and} \quad \neg P(z, f(a)) \vee \neg Q(z).$$

Let us denote

$$P_1 = P(x, f(a)), \quad Q_1 = \neg\, P(z, f(a)).$$

We have

$$\neg, Q_1 = \neg(\neg\, P(z, f(a))) = P(z.f(a)).$$

Clearly, P_1 and $\neg\, Q_1$ unify under the substitution

$$\theta = \{x/z\}.$$

Hence using the resolution inference rule we get the resolvent clause

$$P(z, f(y)) \vee Q(y) \vee \neg Q(z).$$

7.6 Inference methods in FOPL

7.6.1 Propositionalisation: Reduction to propositional inference

The following are the various steps of the method.

1. Using EI rule, we replace an existentially quantified statement by one instantiation.
2. Using UI rule, we replace a universally quantified statement by all possible instantiations.
3. We apply the methods of propositional logic to make valid inferences.

This procedure cannot always be applied because it may produce infinite sets of propositions.

7.6.2 Resolution-refutation method in FOPL

This is very similar to the resolution-refutation method of proof in propositional logic.

Algorithm: Propositional resolution

Let be required to produce a proof of a statement P from a set of statements F.

Step 1. Convert all statements in F to clause form.

Step 2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in Step 1.

Step 3. Repeat until either a contradiction is found, no progress can be made, or a pre-determined effort has been expended.

- (a) Select two clauses. Call these the parent clauses.
- (b) Resolve them together. The resulting clause is called the resolvent.

  (The resolvent is the disjunction of all literals of both the parent clauses with the following exception: If there are any pairs of literals T_1 and $\not T_2$ such that one of the parent clauses contains T_1 and the other contains T_2 and if T_1 and T_2 are unifiable, then neither T_1 T_2 should appear in the resolvent. We call T_1 and T_2 complementary literals. Use the substitution produced by unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.)
- (c) If the resolvent is the empty clause then a contradiction has been found.
- (d) If it is not, then add it to the set of clauses available to the procedure.

If a contradiction has been found, we conclude that P is a valid conclusion from the premises F. Otherwise, that is if the algorithm terminates in such a status that no progress can be made, we conclude that P is not a valid conclusion from F.

Remark

If the clauses to be resolved are chosen in certain systematic ways, the resolution procedure will find a contradiction if one exists. However it may take a long time.

7.6.3 Examples

The various steps in applying the above algorithm to a given simple problem can be summarised as follows:

1. Convert the facts into statements in FOPL.
2. Convert FOPL statements into CNF.
3. Negate the statement which needs to be proved and to the premises.
4. Draw the resolution graph showing the various unifications.

Example 1

Prove the validity of the following argument:

1. Cats like fish.
2. Cats eat everything they like.
3. Josephine is a cat.
4. Therefore, Josephine eats fish.

Solution

First, we express the given sentences and the negated goal G in first-order logic in the CNF.

1. $\neg\, cat(x) \vee likes(fish)$
2. $\neg cat(y) \vee \neg likes(y, z) \vee eats(y, z)$
3. $cat(jo)$
4. $eats(jo, fish)$ (conclusion)

The resolution proof that Josephine eats fish is given in Figure 7.4.

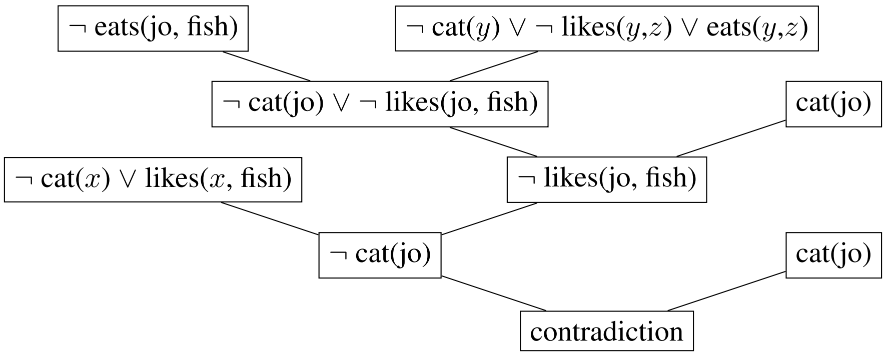

¬ eats(jo, fish)
¬ cat(y) ∨ ¬ likes(y,z) ∨ eats(y,z)
¬ cat(jo) ∨ ¬ likes(jo, fish)
cat(jo)
¬ cat(x) ∨ likes(x, fish)
¬ likes(jo, fish)
¬ cat(jo)
cat(jo)
contradiction

Figure 7.4: An example of a resolution-refutation proof

Example 2

Prove that the conclusion

$$hate(Marcus, Caesar)$$

can be derived from the following premises:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\neg\, Pompeian(x_1) \vee Roman(x_1)$
4. $ruler(Caesar)$
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$

6. $loyalto(x_3, f(x_3))$

7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$

8. $tryassassinate(Marcus, Caesar)$

Solution

The various stages in the derivation are shown in Figure 7.5.

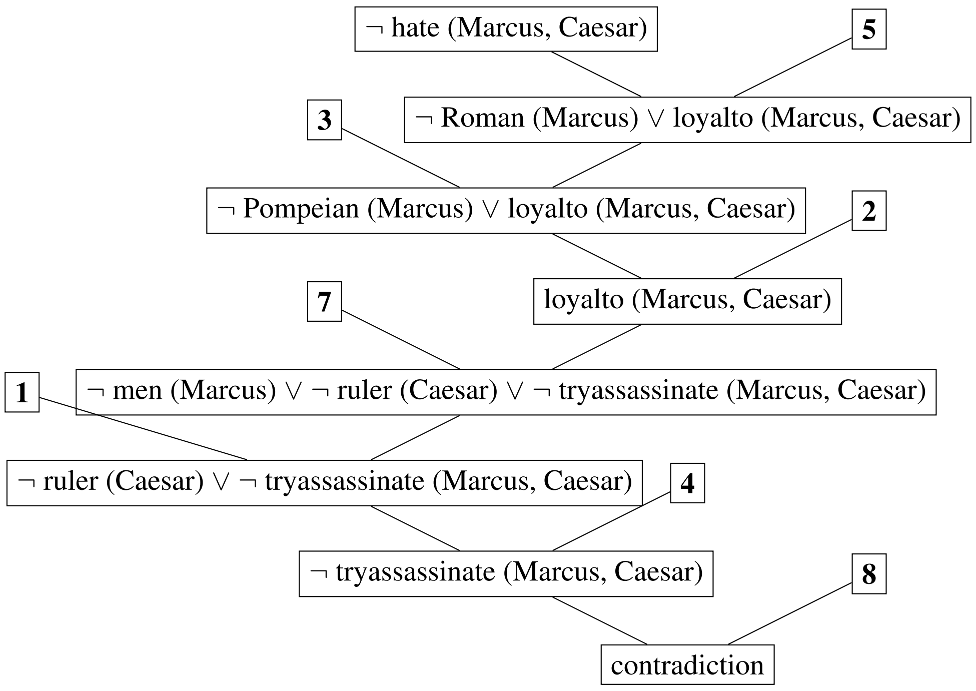

¬ hate (Marcus, Caesar)
5
3
¬ Roman (Marcus) ∨ loyalto (Marcus, Caesar)
¬ Pompeian (Marcus) ∨ loyalto (Marcus, Caesar)
2
7
loyalto (Marcus, Caesar)
1
¬ men (Marcus) ∨ ¬ ruler (Caesar) ∨ ¬ tryassassinate (Marcus, Caesar)
¬ ruler (Caesar) ∨ ¬ tryassassinate (Marcus, Caesar)
4
¬ tryassassinate (Marcus, Caesar)
8
contradiction

Figure 7.5: An example of a resolution-refutation proof

Example 3

Use resolution-refutation method to check the validity of the following argument:

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Therefore, Curiosity killed the cat.

Solution

First, we express the given sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ Loves(y, x)]$

B. $\forall x\ [\exists z\ Animal(z) \wedge Kills(x, z)] \Rightarrow [\forall y\ \neg Loves(y, x)]$

C. $\forall x\ Animal(x) \Rightarrow Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x\ Cat(x) \Rightarrow Animal(x)$

G. $\neg Kills(Curiosity, Tuna)$ (negation of conclusion)

Now we apply the conversion procedure to convert each sentence to CNF:

1. A1: $Animal(F(x)) \vee Loves(G(x), x)$
2. A2: $\neg Loves(x, F(x)) \vee Loves(G(x), x)$
3. B: $\neg Loves(y, x) \vee \neg Animal(z) \vee \neg Kills(x, z)$
4. C: $\neg Animal(x) \vee Loves(Jack, x)$
5. D: $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
6. E: $Cat(Tuna)$
7. F: $\neg Cat(x) \vee Animal(x)$ ÂňG.
8. G: $\neg Kills(Curiosity, Tuna)$

The resolution proof that Curiosity killed the cat is given in Figure 7.6.

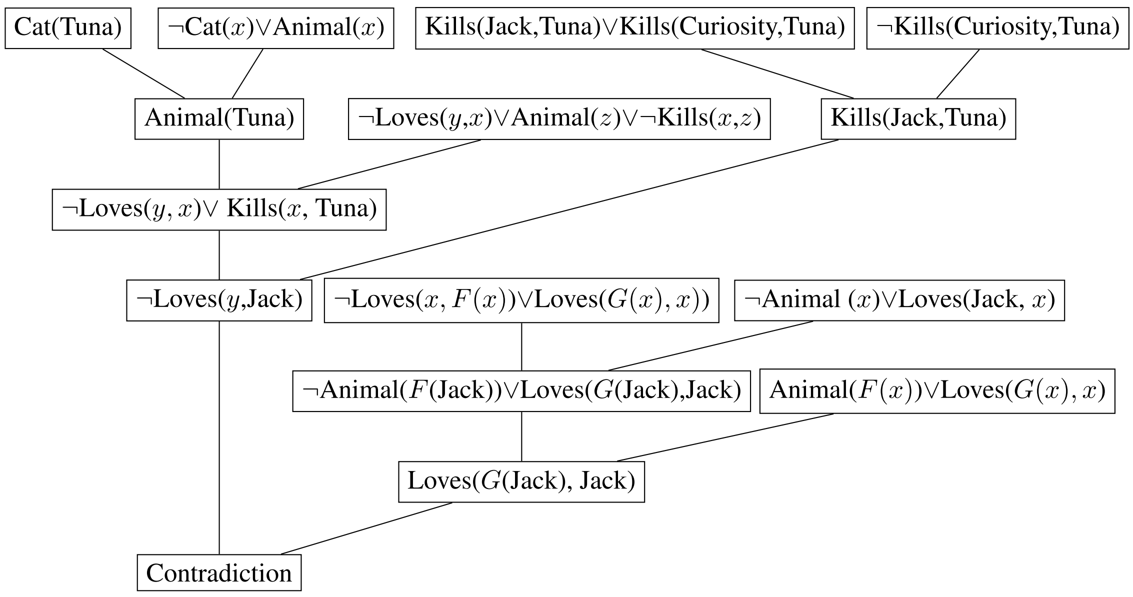

Cat(Tuna)
¬Cat(x)∨Animal(x)
Kills(Jack,Tuna)∨Kills(Curiosity,Tuna)
¬Kills(Curiosity,Tuna)
Animal(Tuna)
¬Loves(y,x)∨Animal(z)∨¬Kills(x,z)
Kills(Jack,Tuna)
¬Loves(y, x)∨ Kills(x, Tuna)
¬Loves(y,Jack)
¬Loves(x, F(x))∨Loves(G(x), x))
¬Animal (x)∨Loves(Jack, x)
¬Animal(F(Jack))∨Loves(G(Jack),Jack)
Animal(F(x))∨Loves(G(x), x)
Loves(G(Jack), Jack)
Contradiction

Figure 7.6: An example of a resolution-refutation proof

Example 4

Given the premises:

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

prove that West is a criminal.

Solution

First, we represent the facts as first-order clauses.

1. ". . . it is a crime for an American to sell weapons to hostile nations":
   $American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$
2. "Nono . . . has some missiles."
   $\exists x\ Missile(x) \wedge Owns(Nono, x)$
3. "All of its missiles were sold to it by Colonel West":
   $Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$
4. We also know that missiles are weapons:
   $Missile(x) \Rightarrow Weapon(x)$
5. We must know that an enemy of America counts as "hostile":
   $Enemy(x, America) \Rightarrow Hostile(x)$
6. "West, who is American . . .":
   $American(West)$
7. "The country Nono, an enemy of America . . .":
   $Enemy(Nono, America)$
8. "West is a criminal."
   $Criminal(West)$

We next write these sentences in the CNF as follows:

1. $\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \not Hostile(z) \vee Criminal(x)$
2. $\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono)$
3. $\neg Enemy(x, America) \vee Hostile(x)$
4. $\neg Missile(x) \vee Weapon(x)$
5. $Missile(M1)$
6. $Owns(Nono, M1)$

7. $American(West)$
8. $Enemy(Nono, America)$
9. $Criminal(West)$

The various satges of the resolution proof are shown in Figure 7.7

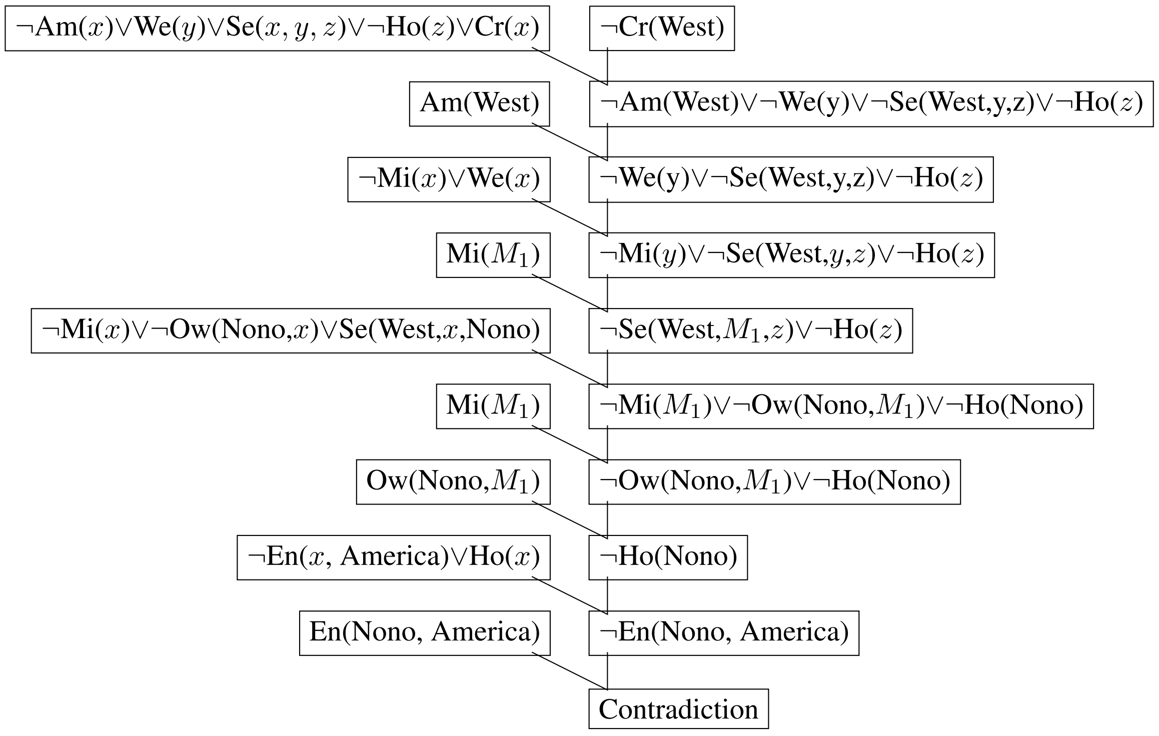

¬Am(x)∨We(y)∨Se(x, y, z)∨¬Ho(z)∨Cr(x)
¬Cr(West)
Am(West)
¬Am(West)∨¬We(y)∨¬Se(West,y,z)∨¬Ho(z)
¬Mi(x)∨We(x)
¬We(y)∨¬Se(West,y,z)∨¬Ho(z)
Mi(M1)
¬Mi(y)∨¬Se(West,y,z)∨¬Ho(z)
¬Mi(x)∨¬Ow(Nono,x)∨Se(West,x,Nono)
¬Se(West,M1,z)∨¬Ho(z)
Mi(M1)
¬Mi(M1)∨¬Ow(Nono,M1)∨¬Ho(Nono)
Ow(Nono,M1)
¬Ow(Nono,M1)∨¬Ho(Nono)
¬En(x, America)∨Ho(x)
¬Ho(Nono)
En(Nono, America)
¬En(Nono, America)
Contradiction

Figure 7.7: An example of a resolution-refutation proof

7.7 Summary of knowledge representation methods

In this and the previous chapter we have seen several knowledge representation methods that are extensively used in artificial intelligence. In this section we make a brief summary of these methods.

1. Knowledge and knowledge representation

Knowledge is a familiarity, awareness, or understanding of someone or something, such as facts, skills, or objects. Knowledge can be acquired in many different ways and from many sources, including perception, reason, memory, testimony, scientific inquiry, education, and practice. There are different types of knowledge like declarative knowledge, procedural knowledge, heuristic knowledge, structural knowledge, etc.

Knowledge representation is a field of artificial intelligence that deals with designing computer representations that capture information and knowledge about the world that can be used to solve problems. Knowledge representation is not just storing data into some database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human.

A knowledge representation structure (or, knowledge representation system) is a particular set of definitions, rules and procedures for setting up a representation that captures information and knowledge about the world.

There are several knowledge representation systems such as the following:

(a) Semantic networks

(b) Frames

(c) Conceptual dependencies

(d) System based on logic

2. Semantic networks

Semantic networks are graphs in which vertices or nodes represent concepts or objects, and the edges represent relations between the nodes. It is a knowledge representation format that can be used to store the "meanings" of words so that human like use of these meanings is possible.

Figure 7.8 shows a semantic network with four objects (John, Mary, 1 and 2) and four categories (Mammals, Persons, FemalePersons and MalePersons).

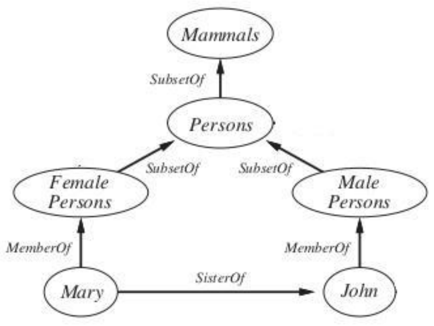


Figure 7.8: A semantic network

3. Frames

The frame contains information on how to use the frame, what to expect next, and what to do when these expectations are not met. Some information in the frame may remain unchanged while others may change. The changing information are stored in "terminals". Terminals can be considered as variables. Different frames may share the same terminals.

Table 7.12 defines a frame with name ALEX. The various slots in the frame, their names, their values and their types are all specified in the table.

4. Conceptual dependency

Conceptual dependency (CD) is a theory of knowledge representation to represent natural language sentences in such way that the representation is independent of the language in which the sentences are stated and it facilitates drawing inferences from sentences.

Slot	Value	Type
ALEX	-	(This Frame)
NAME	Alex	(key value)
ISA	BOY	(parent frame)
SEX	Male	(inheritance value)
AGE	IF-NEEDED:	(procedural
	Subtract(current,BIRTHDATE);	attachment)
BIRTHDATE	8/4/2000	(instance value)
CLIMBS	Trees	(instance value)

Table 7.12: Example frame

The principal conceptual categories in CD theory are “real world actions” (denoted by ACT), “real world objects” (PP), “modifiers of actions” (AA), “attributes of objects” (PA), “times” (T) and “location” (LOC). The CD theory recognises 11 primitive ACTs like “MOVE” (movement of a body part by its owner), “PROPEl” (application of physical force to an object), etc.

The English sentence:

“John ate the ice cream with a spoon.”

is represented in CD theory as in Figure 7.9.

$$\begin{tikzcd}
 & & & \text{John} \arrow[dd, Leftrightarrow] \\
\text{John} \arrow[r, Leftrightarrow, "p"] & \text{INGEST} & \text{ice cream} \arrow[l, "O"'] & {} \arrow[l, "I"'] \\
 & & & \text{do} \\
 & & & \text{spoon} \arrow[u, "O"']
\end{tikzcd}$$

Figure 7.9: Representation of the sentence ”John ate the ice cream with a spoon.” in CD theory

5. Knowledge representation using logic

Logic is the systematic study of valid rules of inference, that is, the relations that lead to the acceptance of one proposition (the conclusion) on the basis of a set of other propositions (premises). Knowledge representation schemes are making use of logic to construct computable models for given domains.

Now let us examine how logic can be used to form representations of the world and how a process of inference can be used to derive new representations about the world and how these can be used by an intelligent agent to deduce what to do.

For this we require a formal language to represent knowledge in a computer tractable form and a reasoning process to manipulate this knowledge to deduce non-obvious facts.

Logic makes statements about the world which are true (or false) if the state of affairs it represents is the case (or not the case). Compared to natural languages and programming languages logic combines the advantages of natural languages and formal languages. Logic is concise, unambiguous, context insensitive, expressive and effective for inferences.

A logic is defined by the syntax that describes the possible configurations that constitute sentences, semantics that determines what facts in the world the sentences refer to and a proof theory which is a set of rules for generating new sentences that are necessarily true given that the old sentences are true.

There are two kinds of logic: propositional logic and first-order logic. For example, knowledge contained in the following sentences can be represented by propositional logic:

> If the program is efficient, it executes quickly. Either the program is efficient, or it has a bug. However, the program does not execute quickly. Therefore it has a bug.

We write
E: the program is efficient
Q: the program executes quickly
B: the program has a bug
The given argument can be represented as follows:

$$(E \Rightarrow Q) \wedge (\neg E \Rightarrow B) \wedge \neg Q \Rightarrow B$$

The knowledge contained in the following statements can be represented using predicate logic.

1. Cats like fish.
2. Cats eat everything they like.
3. Josephine is a cat.
4. Therefore, Josephine eats fish.

This argument can be represented using the notations of predicate logic as follows:

1. $\forall x \; cat(x) \Rightarrow likes(fish)$
2. $\forall y \forall z \neg cat(y) \vee \neg likes(y, z) \vee eats(y, z)$
3. $cat(jo)$
4. $eats(jo, fish)$ (conclusion)

7.8 Sample questions

(a) Short answer questions

1. Define the domain of discourse in predicate logic and illustrate with an example.
2. Explain the existential and universal quantifiers.

3. Explain the sentences: “$\forall x\ :\ x$ is a boy” and “$\exists y\ :\ y$ has legs”.
4. Define a well-formed formula in FOPL.
5. When do we say that a logical statement is in clausal form?
6. Explain universal and existential instantiations in FOPL.
7. Explain the generalised modes ponens.
8. Explain the concept of unification in predicate logic.
9. Give a procedure for unification.
10. What is the resolution inference rule in FOPL?
11. Given the following statements:

    “If it is a pleasant day you will do strawberry picking.”

    “If you are doing strawberry picking you are happy.”

    Use refutation resolution to conclude that “If it is pleasant day you are happy.”

(b) Long answer questions

1. Convert the following sentences to CNF: $\forall x\, P(x) \Rightarrow \forall y\, \exists x\, Q(Y, x)$.
2. From the sentence “Heads I win, tails you lose,” prove using resolution refutation that “I win.”
3. Translate the following sentences into predicates in first order logic.
    (a) Every gardener likes the sun.
    (b) You can fool some of the people all of the time.
    (c) You can fool all of the people some of the time.
    (d) All purple mushrooms are poisonous.
    (e) No purple mushroom is poisonous.
    (f) There are exactly two purple mushrooms.
    (g) There is one and only one Pope.
4. Translate the following into FOPL:
    (a) No student loves Bill.
    (b) Bill has at least one sister.
    (c) Bill has at most one sister.
    (d) Bill has exactly one sister.
    (e) Bill has at least two sisters.
    (f) Only one student failed History.
    (g) No student failed Chemistry but at least one student failed History.
5. Prove the following argument by resolution-refutation:

- John likes all kind of food.
- Apple and vegetable are food.
- Anything anyone eats and not killed is food.
- Anil eats peanuts and is still alive.
- Harry eats everything that Anil eats.
- Therefore, John likes peanuts.

6. Give an algorithm to convert a well-formed formula to clausal form.

7. Describe the resolution-refutation algorithm in predicate logic.

8. Tony, Shi-Kuo and Ellen belong to the Hoofers Club. Every member of the Hoofers Club is either a skier or a mountain climber or both. No mountain climber likes rain, and all skiers like snow. Ellen dislikes whatever Tony likes and likes whatever Tony dislikes. Tony likes rain and snow. Is there a member of the Hoofers Club who is a mountain climber but not a skier?

9. Anyone passing his history exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but John is lucky. Anyone who is lucky wins the lottery. Is John happy?

10. Consider the following axioms:

    - All hounds howl at night.
    - Anyone who has any cats will not have any mice.
    - Light sleepers do not have anything which howls at night.
    - John has either a cat or a hound.

    Use resolution refutation to show that if John is a light sleeper, then John does not have any mice.

Chapter 8

Planning

Planning is the task of finding a procedural course of action for a system to reach its goals while optimizing overall performance measures. It is the first and foremost activity to achieve desired results. Planning is a fundamental property of intelligent behavior. This makes planning an important sub-area of artificial intelligence.

Planning techniques have been applied in a variety of tasks including robotics, process planning, web-based information gathering, autonomous agents and spacecraft mission control.

8.1 Overview

8.1.1 Decomposition of problems

When trying to solve a complicated problem, it may be necessary to work on small pieces of the problem separately and then combine the partial solutions at the end into a complete problem solution.

The ability to decompose a problem into small pieces to find a solution to the problem is important because of the following reasons.

1. The decomposition may help us to avoid having to recompute the entire problem state when we move from one state to next. This in turn help us to consider only that part of the state that may have changed.

    - For example, if we move from one room to another, it does not affect the locations of the doors and the windows in the two rooms.
    - Consider the problem of guiding a robot in an ordinary house. The description of a single state is very large since it must describe the locations of each and every object in the house as well as that of the robot. A single action by the robot will only change a small part of the total state. Instead of writing rules that describe transformations one entire state into another, we would like to write rules that describe only the affected parts of the state description.

2. The decomposition may divide a single difficult problem into several easier subproblems.

8.1.2 Nearly decomposable problems

It may not be always possible to divide a problem into completely separate subproblems. However, problems may be *nearly decomposable* by which we mean that they can be divided into subproblems that have only a small amount of interaction.

For example, consider the problem of moving all furniture out of a room. This can be divided into smaller problems each involving moving one piece of furniture. But if there is a bookcase behind a couch, we must move the couch first before moving the book case. Thus the subproblem of moving the couch and moving the bookcase interact.

8.1.3 Planning

Planning is the process of decomposing a problem into appropriate subparts and of recording and handling interactions among the subparts as they are detected during the problem solving process.

The word **planning** also refers to the process of computing several steps of a problem-solving procedure before executing them.

8.1.4 Impact of predictability on planning

The success of the planning approach depends critically on the predictability of the problem's domain or universe. If the universe is unpredictable we cannot be sure of the outcome of a solution step. In such cases we can consider only a set of possible outcomes possibly in some order according to the likelihood of the outcome occurring. Possibly the plan have to include paths for all possible outcomes of each step. But there may a great many outcomes for each step. In such situations it would be a great waste of effort to formulate plans for all possibilities.

If the universe is unpredictable, we may take one step at a time and not really try to plan ahead. Or, we we may try to produce a plan that is *likely* to succeed. If the plan fails, we may start the planning process over again!

8.2 Components of a planning system

The following the components of a planning system:

1. Choosing rules to apply
2. Applying rules
3. Detecting a solution
4. Detecting dead ends
5. Repairing an almost correct solution

8.2.1 Choosing rules to apply

The commonly used technique can be described thus:

1. Isolate a set of differences between the desired goal state and the current state.

2. Identify those rules that reduce the differences.

3. If several rules are hound, use heuristic methods to chose one.

8.2.2 Applying rules

In simple systems application of rules is easy. Each rule simply specified the problem state that would result from the application of the rule.

In complex systems rules specify only a small part of the problem state. In such cases there are several methods for the application of rules.

1. Describe each of the changes to the state description the application of the rule makes. Add a new rule that everything else is remaining unchanged. To do this, we may require the explicit statements of a large number of premises or axioms.

2. Describe each operator by a list of new predicates (called ADD) that the operator causes to become true and a list of old predicates (called DELETE) that the operator causes to become false. There is also a third list (called PRECONDITION) which specifies the predicates that must be true for the operator to be applied. Any predicate not included in either ADD or DELETE is is assumed to be unaffected by the rule.

8.2.3 Detecting a solution

A planning system can be considered to have succeeded in finding a solution to a problem, if it has found a sequence of operators that transforms the initial problem state into the goal state. How does the system know whether it has succeeded in finding a solution? In simple problems the system can easily determine whether it has succeeded in finding a solution by checking whether the problem state and goal state match.

But in complex problems, in a description of a problem state, the entire state may not be explicitly described; the state may be specified by a description of a smaller set of relevant properties. So in such cases there should be a method to determine whether two problem states match. This method will depend on how the various states are represented in the planning system.

For example, suppose that a planning system is represented using predicate logic. Let there be a predicate $P(x)$ as part of the goal. To see whether $P(x)$ has been satisfied at any stage, it is enough to prove that we can derive $P(x)$ from the assertions describing that particular state. If we can construct such a proof, the problem-solving process can be ended.

8.2.4 Detecting dead-ends

The planning system must be able to detect dead-end, that is, it must be able to determine whether a path it is exploring can never lead to a solution or appears unlikely to lead to a solution.

The mechanisms for detecting a solution are used to detect dead-ends also.

8.2.5 Repairing an almost correct solution

Consider the problem of solving a nearly decomposable problem.

We assume that the problem is completely decomposable and solve each subproblem separately and then combine the solutions of the subproblems to obtain a solution of the original problem. Since the problem is only *nearly* decomposable, the combined solution may not be the correct solution; it will only be an *almost correct solution*. The following are some of the ways in which an almost correct solution may be repaired.

1. Throw out the almost correct solution and look for another one and hope that it is better!
2. Consider the situation arising from the execution of the sequence of operations specified in the proposed almost correct solution. The difference between this and the goal will much less than the difference between the initial state and the goal. The problem solving system may be applied again to eliminate the difference.
3. Find exactly what went wrong and try a direct patch.
4. Apply the *least commitment strategy*. This means, leave the almost correct solutions incompletely specified until the last possible moment and when as much information as possible is available complete the specifications in such a way that no conflicts arise.

8.3 Goal stack planning

8.3.1 Stacks

As the name indicates, in goal stack planning, we make use the *stack* data structure. A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example âĂŞ a stack of books or a pile of plates, etc.

Figure 8.1: A stack of books and a pile of plates

A real-world stack allows operations at one end only. For example, we can place or remove a book or a plate from the top of the stack only. Likewise, the Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

8.3.2 Goal stack

In goal stack planning, we make use of stack called a **goal stack** usually denoted by GS. The goal stack GS contains both subgoals and actions that have been proposed to satisfy those subgoals. It relies on a database, denoted by DB, that describes the current situation, and a set of actions described by PRECONDITION, ADD and DELETE lists.

8.3.3 Basic idea

At each succeeding step of the problem solving process, a subgoal of the stack is pursued. An action that could cause it to be true is sought. If an appropriate action is found, its precondition is established as the top subgoal of the stack in order for that action to be applicable. When a sequence of actions that satisfies a subgoal is found, that sequence is applied to the current situation, yielding a new current situation. Then, all the subgoals that are satisfied by the new current situation are popped off the stack. Next, another subgoal of the stack is explored and an attempt is made to satisfy it, starting from the situation that was produced as a result of satisfying the first subgoal.

This process continues until the goal stack is empty. Then, as one last check, the original goal is compared to the final situation derived from the application of the chosen actions. If any conditions on the goal are not satisfied in that situation (which they might not be if they were achieved at one point and then undone later), then those unsolved subgoals are reinserted onto the stack and the process resumed.

8.3.4 The STRIPS planner

The reasoning strategy used by STRIPS is goal stack planning. STRIPS (Stanford Research Institute Problem Solver) is an automated planner developed by Richard Fikes and Nils Nilsson in 1971. The same name was later used to refer to the formal language of the inputs to this planner. This language is the base for most of the languages for expressing automated planning problem instances in use today; such languages are commonly known as action languages.

8.3.5 Algorithm

```
1. Push the compound predicate describing the goal state in to the Stack.
2. Push the individual predicates of the goal state in to the Stack.
3. Loop till the Stack is empty
   (a) Pop an element $E$ from the Stack.
   (b) If $E$ is a predicate
       i. If $E$ is True
          A. Do nothing.
       ii. Else
          A. Push the relevant action into the Stack.
          B. Push the individual predicates of the precondition of the action into the
             Stack.
   (c) Else if $E$ is an action "$a$"
       i. Apply the action "$a$" to the current state.
       ii. Add the action "$a$" to the plan.
```

8.3.6 Illustration: Blocks world

To illustrate the goal stack planning approach, we consider the blocks world domain (see Section 2.3.5 also).

1. The objects of the blocks world

- A flat surface on which blocks can be placed.
- A number of square blocks, all of the same size.
- A robot arm that can manipulate the blocks.
- Blocks can be stacked one upon another.

2. Predicates and relations to describe the state of the blocks world

Predicate	**Meaning**
ON(x, y)	Block x is on block y
ONTABLE(x)	Block x is on table.
CLEAR(x)	There is nothing on top of block x
HOLDING(x)	The arm is holding block x
ARMEMPTY	The arm is holding nothing.

3. Actions

The following are the available actions in the blocks world.

Action	**Meaning**
UNSTACK(x, y)	Pick up block x from its current position on block y and hold it in the arm
STACK(x, y)	Place block x held in the arm on block y.
PICKUP(x)	Pick up block x from table and hold it.
PUTDOWN(x)	Put block x held in the arm down on the table.

The PRECONDITION list (the list of predicates that should be true to apply the action), ADD list (the list of predicates that become true after the action) and DELETE list (the list of predicates that become false after the action) associated with the various actions are given below.

Action	**PRECONDITION list**	**ADD list**	**DELETE list**
UNSTACK(x, y)	ARMEMPTY, CLEAR(x), ON(x, y)	HOLDING(x), CLEAR(y)	ARMEMPTY
STACK(x, y)	HOLDING(x), CLEAR(y)	ARMEMPTY, ON(x, y), CLEAR(x)	
PICKUP(x)	ARMEMPTY, CLEAR(x).	HOLDING(x)	ARMEMPTY
PUTDOWN(x)	HOLDING(x).	ARMEMPTY, ONTABLE(x), CLEAR(x)	

4. Problem

Given the initial and the final states of a blocks world with four blocks as shown in Figure 8.2, use goal stack planning algorithm to obtain a plan for achieving the goal state.

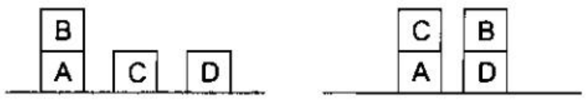


Figure 8.2: A simple blocks world problem: Initial state on the left and the goal state on the right

5. Solution

Step 1. The initial state can be described by the statement:

$$ON(B, A) \wedge ONTABLE(A) \wedge ONTABLE(C) \wedge ONTABLE(D) \wedge ARMEMPTY$$

The goal state can be specified by the statement:

$$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$$

Step 2. We create a stack and call it GS. We initialise GS to be empty.

(empty)

Step 3. We push the goal state to the stack:

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Step 4. The goal state can be divided into four components, namely,

$$ON(C, A), \quad ON(B, D), \quad ONTABLE(A), \text{and} \quad ONTABLE(D)$$

Of these the last two are true in the initial state. We push the first two components to the stack GS.

$ON(C, A)$

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Note: The choice of the order in which the components are stacked is indeed important. For example, the two new components may be stacked in the following order also. We have chosen the order as above in a purely arbitrary manner. However, to make an informed choice, we may need additional knowledge about the world in which we are operating.

$ON(B, D)$

$ON(C, A)$

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Step 5. Consider the top item in GS which is $ON(C, A)$. Check whether it is true in current state. It is not. Find an action which makes it true. the action is $STACK(C, A)$. We replace $ON(C, A)$ by $STACK(C, A)$ to get the following GS.

$STACK(C, A)$

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Step 6. To apply the action $STACK(C, A)$ the preconditions $CLEAR(A) \wedge HOLDING(C)$ must be true. This precondition and also its components are also pushed to the stack GS.

$CLEAR(A)$

$HOLDING(C)$

$CLEAR(A) \wedge HOLDING(C)$

$ON(C, A)$

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Note: As in Step 4, here also the order in which the components $CLEAR(A)$ and $HOLDING(C)$ are stacked has been chosen arbitrarily. The command $HOLDING(C)$ may be stacked above $CLEAR(A)$.

Step 7. We now check whether $CLEAR(A)$ is true. It is not. the only operator that make it true is $UNSTACK(B, A)$. So we replace $CLEAR(A)$ by $UNSTACK(B, A)$.

$UNSTACK(B, A)$

$HOLDING(C)$

$CLEAR(A) \wedge HOLDING(C)$

$STACK(C, A)$

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Step 8. To apply $UNSTACK(B, A)$, the precondition $ON(B, A) \wedge CLEAR(B) \wedge ARMEMPTY$ must be true. We add this precondition and its components to the stack GS.

$ON(B, A)$

$CLEAR(B)$

$ARMEMPTY$

$ON(B, A) \wedge CLEAR(B) \wedge ARMEMPTY$

$UNSTACK(B, A)$

$HOLDING(C)$

$CLEAR(A) \wedge HOLDING(C)$

$STACK(C, A)$

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Step 9. The top element $ON(B, A)$ is true and so we pop it off from the stack. The next elemt $CLEAR(B)$ is also true and so we pop it off also. At the current state, the arm is not holding anything and so $ARMEMPTY$ is also true and we pop it off also. Sine each of the components of the next statement is true, the compounded statement is also true. Thu it is also popped off from the stack GS. At this stage the stack is as follows.

$UNSTACK(B, A)$

$HOLDING(C)$

$CLEAR(A) \wedge HOLDING(C)$

$STACK(C, A)$

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

Step 10. The top element of the stack is $UNSTACK(B, A)$. The preconditions for applying this action are satisfied.

(a) We now apply this operator to get a new world state.

(b) We record that $UNSTACK(B, A)$ is the first operator of the solution sequence.

(c) The state of the world is specified by the following statements:

$$ONTABLE(A), ONTABLE(C), ONTABLE(D),$$
$$HOLDING(B), CLEAR(A)$$

(d) The goal stack GS is now

$$
\begin{array}{l}
HOLDING(C) \\
CLEAR(A) \wedge HOLDING(C) \\
STACK(C, A) \\
ON(B, D) \\
ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D) \\
\hline
\end{array}
$$

Step 11. The process is now repeated.

8.4 Hierarchical planning

8.4.1 The basic approach

In order to solve real world problems, the problem solver may have to generate long plans. To do this efficiently, the problem solver may ignore some of the details of the problem until a solution to the main issues have been found. This can be done by creating a hierarchy of of the tasks to be performed and then attempting to find a solution starting from the tasks at the highest level of the hierarchy. Creating a plan in this way is known as hierarchical planning.

Example

- For example, let it be required to travel at short notice to Mumbai with minimum expenses for an interview. The task of the highest priority is to check the availability of an airline at affordable cost. After finding a plan to solve this problem, the person may find plans for solving the next level problems like preparing baggage and hiring a taxi. Attempts to solve these problems lead to problems at the third level: detailed plans for preparing the baggage and hiring a taxi. These will in turn lead to problems at the next lower level.

8.4.2 The ABSTRIPS planner system

ABSTRIPS, a planning system developed in 1974 and built on top of the STRIPS planning system, is a planning system has a method for the implementation of hierarchical planning. In ABSTRIPS, the hierarchies are constructed by assigning *criticalities* which are numbers indicating relative difficulty to the preconditions of the operator. First a plan is found that satisfies only the preconditions of the the operators of the highest criticality values. The plan is then refined by considering the preconditions at the next level of criticality and inserting steps into the plan to achieve these preconditions. The process is repeated to the lowest level of criticality. For this system to work, there must be a scheme to assign the appropriate critcality values to the various terms in the precondition. Because this process explores entire plans at one level of detail before looking any of the lower details, it has been called *length-first search*.

8.5 Sample questions

(a) Short answer questions

1. Define planning in AI.
2. List the components of planning.
3. give examples of planning strategies.

(b) Long answer questions

1. Explain the components of planning.
2. Explain goal stack planning.
3. Describe the goal stack planning algorithm.
4. Explain hierarchical planning.

Chapter 9

Learning

9.1 What is learning?

Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.[1]

9.2 Forms of learning based on feedback

Consider a class of problems in which from a collection of input-output pairs a machine learns a function that predicts the output for new inputs.

1. Unsupervised learning

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

2. Reinforcement learning

In reinforcement learning the agent learns from a series of reinforcements - rewards or punishments. For example, the lack of a tip at the end of the journey gives the taxi agent an indication that it did something wrong. The two points for a win at the end of a chess game tells the agent it did something right.

3. Supervised learning

In supervised learning the agent observes some example input-output pairs and learns a function that maps from input to output.

[1] H A Simon, ”Why Should Machines Learn?”, Chapter 2 in *Machine Learning, An Artificial Intelligence Approach*, Tioga Publishing Company, 1983.

4. Semi-supervised learning

In semi-supervised learning we are given a few labeled examples and must make what we can of a large collection of unlabeled examples. Imagine that you are trying to build a system to guess a person's age from a photo. We gather some labeled examples by snapping pictures of people and asking their age. That is supervised learning. But in reality some of the people lied about their age and to uncover them is an unsupervised learning problem.

9.3 Forms of learning based on different ways of acquiring knowledge

9.3.1 Rote learning

Rote learning is simply storing data in memory. This helps programmes to perform better in the future. We store computed values so that we do not have to recompute them later. In complex learning system this requires capabilities for organised storage of information and generalisation.

Example learning system

A programme to play checkers developed by Arthur Samuel[2] in 1963 used the rote learning technique to improve its performance.

9.3.2 Learning by taking advice

The idea of advice taking in AI based learning was proposed as early as 1958 by John McCarthy[3]. However very few attempts were made in creating such systems until the late 1970s.

There are two basic approaches to advice taking:

- Take high level, abstract advice and convert it into rules that can guide performance elements of the system. Automate all aspects of advice taking.
- Develop sophisticated tools such as knowledge base editors and debugging. These are used to aid an expert to translate his expertise into detailed rules. Here the expert is an integral part of the learning system.

Example learning system

FOO (First Operational Operationaliser) is a programme developed in 1983 which accepts advice for playing hearts, a card game. It tries to convert high level advice (principles, problems, methods) into effective executable (LISP) procedures. Hearts is a game of partial information with no known algorithm for winning.

[2]Arthur Lee Samuel (1901 âĂŞ 1990) was an American pioneer in the field of computer gaming and artificial intelligence. He popularized the term "machine learning" in 1959. The Samuel Checkers-playing Program was among the world's first successful self-learning programs, and as such a very early demonstration of the fundamental concept of artificial intelligence (AI).

[3]John McCarthy (1927 âĂŞ 2011) was an American computer scientist and cognitive scientist. McCarthy was one of the founders of the discipline of artificial intelligence. He co-authored the document that coined the term "artificial intelligence" (AI), and was very influential in the early development of AI.

General advice can be given such as “Avoid taking points”. A human must convert into a FOO representation of the form

```
    (avoid (take-points me) (trick))
```

FOO operationalises the advice by translating it into expressions it can use in the game.

```
(achieve (not (during
           (scenario
              (each p1 (players) (play-card p1))
              (take-trick (trick-winner)))
              (take-points me))))
```

9.3.3 Learning in problem solving

There are three basic methods in which a system can learn from its own experiences.

1. Learning by parameter adjustment

Many programs rely on an evaluation procedure to summarise the state of search etc. Game playing programs provide many examples of this. However, many programs have a static evaluation function.

So the basic idea of idea of parameter adjustment is to:

- Start with some estimate of the correct weight settings.
- Modify the weight in the program on the basis of accumulated experiences.
- Features that appear to be good predictors will have their weights increased and bad ones will be decreased.

Example learning system

Samuel’s Checkers programs employed used a static function of the form

$$c_1t_1 + c_2t_2 + \cdots + c_{16}t_{16}$$

where the t terms are values of the 16 features and the c terms are weights. As learning progresses, the c values change.

2. Learning with macro-operators

The basic idea here is similar to rote learning: "Avoid expensive recomputation". Sequences of actions that can be treated as a whole are called *macro-operators*. For example making dinner can be described by the macro-actions lay the table, cook dinner, serve dinner. We could treat laying the table as on action even though it involves a sequence of actions.

Example learning system

The STRIPS problem-solving system employed macro-operators in it's learning phase. The STRIPS system has a learning component: After each problem solving episode, the system stores the computed plan as a macro-operator.

Consider a blocks world example in which ON(C,B) and ON(A,TABLE) are true. STRIPS can achieve ON(A,B) in four steps:

$$UNSTACK(C, B), PUTDOWN(C), PICKUP(A), STACK(A, B)$$

STRIPS now builds a macro-operator MACROP with preconditions ON(C,B), ON(A,TABLE), postconditions ON(A,B), ON(C,TABLE) and the four steps as its body. MACROP can now be used in future operation. But it is not very general. The above can be easily generalised with variables used in place of the blocks.

3. Learning by chunking

Chunking involves ideas similar to macro-operators and originates from psychological ideas on memory and problem solving. Its computational basis is in production systems.

Example learning system

The idea of learning by chunking has been used in the SOAR system which uses production rules to represent its knowledge. SOAR is an architecture for general intelligence developed by John E Laird and others in 1987.

9.3.4 Inductive learning

This involves the process of learning by example – where a system tries to induce a general rule from a set of observed instances.

Inductive learning involves classification, that is, assigning to a particular input the name of a class to which it belongs. Classification is important to many problem solving tasks. A learning system has to be capable of evolving its own class descriptions.But it is sometimes difficult to construct class definitions by hand. This is particularly true in domains that are not well understood or that change rapidly. The task of constructing class definitions is called *induction* or *concept learning*.

Example learning systems

The following are examples of concept learning programmes.

- Winston's learning programme: This programme operated in the blocks world domain. Its goal was to construct representations of concepts in the blocks domain.
- The version space approach introduced by Mitchell in 1977 attempts to produce a description that is consistent with all positive examples but no negative examples in the training set.
- A third approach to concept learning is the induction of *decision trees* as exemplified by the ID3 programme of Quinlan (1986). ID3 is a program that can build trees automatically from given positive and negative instances. Basically each leaf of a

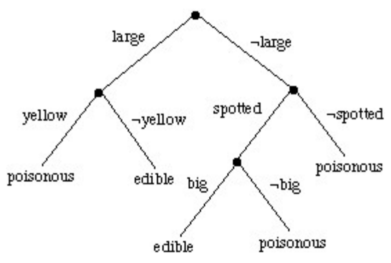


Figure 9.1: A decision tree

decision tree asserts a positive or negative concept. To classify a particular input we start at the top and follow assertions down until we reach an answer (see Figure 9.1).

9.3.5 Explanation based learning

Humans appear to learn quite a lot from one example. The basic idea of explanation based learning is to use results from one example problem solving effort the next time around.

An explanation based learning accepts four kinds of input:

- **A training example**: what the learning program "sees" in the world.
- **A goal concept**: A high level description of what the program is supposed to learn.
- **An operational criterion**: A description of which concepts are usable.
- **A domain theory**: A set of rules that describe relationships between objects and actions in a domain.

From this, EBL computes a *generalisation* of the training example that is sufficient to describe the goal concept and also satisfies the operationality criterion.

9.3.6 Discovery

Discovery is a restricted form of learning in which one entity acquires knowledge without the help of a teacher. Discovery can be of two types:

- Theory driven discovery
- Data driven discovery

Theory driven discovery: Example learning system

The Automated Mathematician (AM) is one of the earliest successful discovery systems. It was created by Douglas Lenat in 1970's.

AM has two inputs: One, a description of some concepts of set theory, for example, set union, intersection, the empty set; two, information on how to perform mathematics.

Given the above information AM discovered:

- Integers: it is possible to count the elements of a set.

- Addition: The union of two disjoint sets and their counting function.
- Multiplication: Having discovered addition and multiplication as laborious set-theoretic operations more effective descriptions were supplied by hand.
- Prime Numbers: Factorisation of numbers and numbers with only one factor were discovered.
- Golbach's Conjecture: Even numbers can be written as the sum of 2 primes. E.g. 28 = 17 + 11.

Data driven discovery: Example learning system

Many discoveries are made from observing data obtained from the world and making sense of it, for example, discovery of planets, quantum mechanics, discovery of sub-atomic particles.

BACON[4] is an attempt to provide such an AI system. BACON was able able to derive the ideal gas law. BACON has also been applied to Kepler's third law, Ohm's law, conservation of momentum and Joule's law.

9.3.7 Analogy

Analogy involves a mapping between what might appear to be two dissimilar concepts.

For example, consider the following statement: "Finding a good man is like finding a needle in a haystack." This involves a mapping between two worlds: One, a world of men, good men and searching for good men; two, a world of small objects, a haystack and searching for a needle.

There are two methods of analogical problem solving in AI: transformational analogy and derivational analogy.

Transformational analogy

Carbonell (1983) describes a T-space method to transform old solutions into new ones.

- Whole solutions are viewed as states in a problem space – the T-space.
- T-operators prescribe methods of transforming existing solution states into new ones.
- Reasoning by analogy becomes a search in T-space – means-end analysis.

Derivational analogy

Derivational analogy is a method of solving problems based upon the transfer of past experience to new problem situations. The experience transfer process consists of recreating lines of reasoning, including decision sequences and accompanying justifications, that proved effective in solving particular problems requiring similar initial analysis. The derivational analogy approach is advocated as a means of implementing reasoning from individual cases in expert systems.

[4]Named after the philosopher of science Sir Francis Bacon (1561 - 1626).

9.4 Neural net learning

9.4.1 Neurons and perceptrons

An Artificial Neuron Network (ANN), popularly known as Neural Network is a computational model based on the structure and functions of biological neural networks. It is like an artificial human nervous system for receiving, processing, and transmitting information. Figure 9.2 shows the structure of a neuron in the human nervous system.

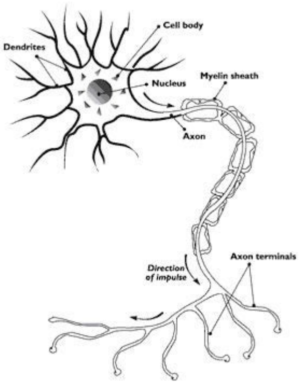


Figure 9.2: The structure of a neuron

In an ANN, each neuron receives one or more inputs and produces one or more outputs. In its simplest form known as a *perceptron* a neuron takes a weighted sum of its inputs and produces an output 1 if the sum is greater than a certain pre-defined threshold value and 0 otherwise. The relation between the inputs and the output of a perceptron can be represented as in Figure 9.3. In the figure the output is defined as follows:

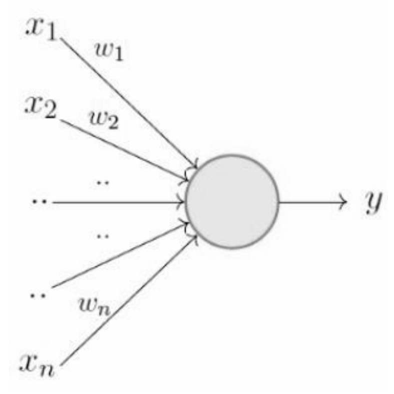


Figure 9.3: The model of a perceptron

$$y = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + \cdots + w_nx_n > w \\ 0 & \text{otherwise} \end{cases}$$

where w is some threshold value.

9.4.2 Neural networks

A neural networks consists of a number of neurons structured into different layers as in Figure 9.4 where each circle represents a neuron.

Basically, there are three different layers in a neural network:

1. Input layer: All the inputs are fed in the model through this layer.
2. Hidden layers: There can be more than one hidden layers which are used for processing the inputs received from the input layers.
3. Output layer: The data after processing is made available at the output layer.

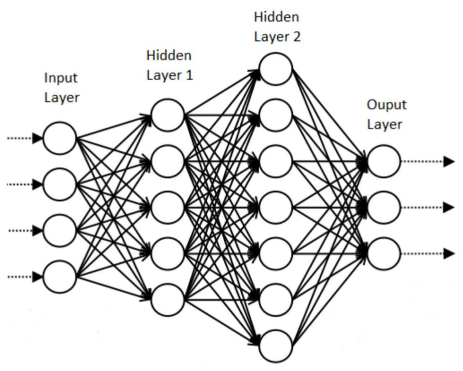


Figure 9.4: The different layers of a neural network

9.4.3 Learning in neural networks

In neural networks, learning is the process of modifying the values of the weights and the threshold. The learning takes place through an iterative process of "going and return" by the layers of neurons. The "going" is a forward propagation of information and the "return" is a backward propagation of information.

9.4.4 Applications

Artificial neural networks have found applications in many disciplines. Application areas include:

1. vehicle control, trajectory prediction
2. natural resource management
3. pattern recognition (radar systems, face identification, signal classification, 3D reconstruction, object recognition and more)
4. sequence recognition (gesture, speech, handwritten and printed text recognition)
5. medical diagnosis
6. finance (e.g. automated trading systems)

7. data mining
8. visualization
9. machine translation
10. e-mail spam filtering

9.5 Genetic algorithms

A genetic algorithm is an algorithm inspired by the process of natural selection in the evolution of organisms. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems.

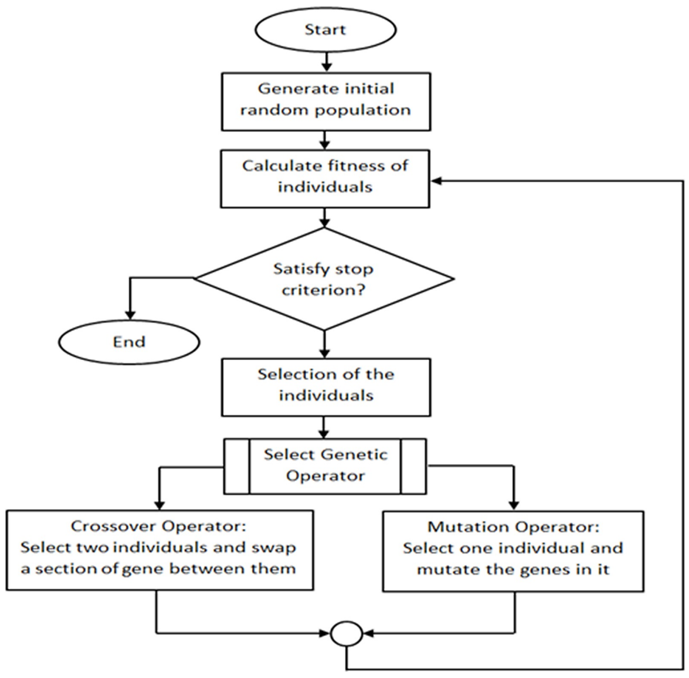

Start
Generate initial random population
Calculate fitness of individuals
Satisfy stop criterion?
End
Selection of the individuals
Select Genetic Operator
Crossover Operator: Select two individuals and swap a section of gene between them
Mutation Operator: Select one individual and mutate the genes in it

Figure 9.5: A genetic algorithm flowchart

9.5.1 Outline

In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (called its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s.

The evolution usually starts from a population of randomly generated individuals, and is an iterative process. The population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are randomly selected from the current population, and each individual's genome (the set of chromosomes) is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

The chromosome representation, selection, crossover (or, recombination), mutation, and fitness function computation are the key elements of the genetic algorithm.

A genetic algorithm flowchart is shown in Figure 9.5.

9.6 Sample questions

(a) Short answer questions

1. What is learning?
2. Describe supervised, unsupervised and reinforcement learning.
3. List the different types of learning based on feedback.
4. Explain inductive learning.
5. Explain explanation based learning.
6. Explaining discovery learning.
7. Describe the use of analogy in learning.

(b) Long answer questions

1. Explain the different types of learning based on different ways of acquiring knowledge.
2. Explain different types of learning in problem solving.
3. Describe the basic ideas of neural net learning.
4. With the help of a flowchart, explain the basic concepts of the genetic algorithm.

Chapter 10

Expert systems

10.1 Introduction

An **expert system** is a computer program that uses artificial intelligence methods to solve problems within a specialized domain that ordinarily requires human expertise. The first expert system was developed in 1965 by Edward Feigenbaum and Joshua Lederberg of Stanford University. Their system known as Dendral was designed to analyze chemical compounds. Expert systems now have commercial applications in fields as diverse as medical diagnosis, petroleum engineering, and financial investing.

In order to accomplish feats of apparent intelligence, an expert system relies on two components: a knowledge base and an inference engine. A knowledge base is an organized collection of facts about the system's domain. An inference engine interprets and evaluates the facts in the knowledge base in order to provide an answer. Typical tasks for expert systems involve classification, diagnosis, monitoring, design, scheduling, and planning for specialized endeavours.

Facts for a knowledge base must be acquired from human experts through interviews and observations. This knowledge is then usually represented in the form of "if-then" rules. The knowledge base of a major expert system includes thousands of rules. A probability factor is often attached to the conclusion of each rule and to the ultimate recommendation, because the conclusion is not a certainty.

10.2 Architecture of an expert system

Figure 10.1 shows the typical architecture of an expert system. The various components are explained below.

10.2.1 User interface

The user interacts with the system through a user interface which may use menus, natural language or any other style of interaction.

The user interface is closely linked to the explanation component described below. This is the dialogue component of the system. One side of the dialogue involves the user questioning the system and on the other side the system must be able to question the user to establish the existence of evidence. The dialogue component has two functions; determines which question to ask next and keep record of the previous questions.

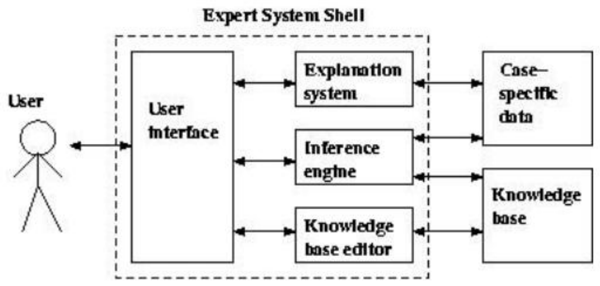


Figure 10.1: Typical architecture of an expert system

The dialogue could be one of three styles:

- System controlled, where the system drives the dialogue through questioning the user.
- User controlled, where the user drives the consultation by providing information to the system.
- Mixed control, where both user and system can direct the consultation.

10.2.2 Knowledge base

This element of an expert system system is so critical to the way the most expert system are constructed that they are also popularly known as knowledge based systems.

A knowledge base contains both declarative knowledge (facts about objects, events and situations) and procedural knowledge (information about course of action). Although many knowledge representation techniques have been used in expert system the most common form is the rule-based production system approach.

In rule-based systems the procedural knowledge, in the form of heuristic "if- then" rules, is completely integrated with the declarative knowledge. However, not all rules pertain to the system's domain. There are rules called meta-rules which pertain to other rules (or even to themselves). Here is an example of a meta-rule from MYCIN:

"If there are rules which do not mention the current goal in their premise or if there are rules which mention the current goal in their premise, then it is definite that the former should be done before the latter."

10.2.3 Inference engine

By simply having access to a great deal of knowledge does not make us an expert, we must know how or when to apply the appropriate knowledge. Similarly just having a knowledge base does not make an expert system intelligent. The system must have another component which directs the implementation of knowledge base. This component of the system is the inference engine.

The inference engine uses the information provided to it by the knowledge base and the user to infer new facts. It also decides which heuristic search techniques are to be used in determining how the rules in the knowledge are to be applied to the problem. In fact an inference engine "runs" an expert system, determining which rules are to be invoked, accessing the appropriate rules, executing the rules and determining when an acceptable solution has been found.

The reasoning process can be simple or complicated, depending on the structure of knowledge base. If the knowledge base consists of simple rules and facts, forward chaining suffices. However, for a knowledge which consists of rules and unstructured logic (facts, data and variables), both sophisticated forward and backward chaining with well thought-out search strategies may be required. Other methods such as problem reduction, pattern matching, unification etc. are also used to implement the reasoning process.

Mostly, the knowledge is not inter twined with the control structure. This has a value-added advantage, the same inference engine which works well in one expert system may work just as well with a different knowledge base to make another expert system, thus reducing expert system developing time.

For example, inference engine of MYCIN is available separately as EMYCIN (essential MYCIN). EMYCIN has been used with different knowledge base to create many new expert system, eliminating the need to develop a new inference engine. This will be taken up after having studied expert system shells.

10.2.4 Case specific data

The case specific data includes both data provided by the user and partial conclusions (along with certainty measures) based on this data. In a rule-based system the case specific data will be the elements in working memory.

10.2.5 Explanation system

The explanation system allows the program to explain its reasoning to the user. It is not acceptable for an expert system to take decisions without being able to provide an explanation for the decisions it has taken. Users using these expert systems need to be convinced of the validity of the conclusion drawn before applying it to their domain. They also need to be convinced that the solution is appropriate and applicable in their circumstances.

Knowledge engineers building the expert system also need to examine the reasoning behind decisions in order to access and evaluate the mechanisms being used. If explanation component is not provided it would not be possible to judge whether the expert system is working as desired or intended.

10.2.6 Knowledge base editor

Most expert systems provide a mechanism for editing the knowledge base. In the simplest case, this is just a standard text editor for modifying rules and data by hand. But many tools include other facilities in their support environment. For example, EMYCIN uses automatic book-keeping.

Another common facility in knowledge base editors is syntax checking, where the editor uses knowledge about the grammatical structure of the expert system language to help the user input rules with the correct spelling and format. When the user enters an ungrammatical rule or command, the editor catches it and explains what is wrong.

An extremely useful but generally unavailable facility for knowledge base editors is consistency checking where the system checks the semantics or meanings of the rules and data being entered to see if they conflict with existing knowledge in the system. When a conflict occurs, the editor helps the user resolve the conflict by explaining what caused it and describing ways to remove it.

Another potentially useful but generally unavailable facility for knowledge base editors is knowledge extraction where the editor helps the user enter new knowledge into the system. It combines syntax and consistency checking with sophisticated prompting and explanation to let even naive users add or modify rules.

10.2.7 Expert shell

One important feature of expert systems is the way they separate domain specific knowledge from more general purpose reasoning and representation techniques. The general purpose bit (in the dotted box in figure 10.1) is referred to as an **expert system shell**. The shell provides the inference engine, the user interface, an explanation system and sometimes a knowledge base editor. There are numerous commercial expert system shells, each one appropriate for a slightly different range of problems.

10.3 Roles of individuals who interact with expert systems

- **Domain expert** The domain experts are the individuals who are currently experts in solving the problem which the system is intended to solve.
- **Knowledge engineer**

  The knowledge engineer is the person who encodes the expert's knowledge in a declarative form that can be used by the expert system.
- **User**

  The user is the person who will be consulting with the system to get advice which would have been provided by the expert.
- **System engineer**

  The system engineer builds the user interface, designs the declarative format of the knowledge base, and implements the inference engine.

10.4 Languages and tools

Expert system tools are programming systems which simplify the job of constructing an expert system. They range from very high-level programming language to low-level support facilities. We divide expert system tools into four major categories as shown in Figure 10.2. We will describe these tools and explain briefly how they are used.

10.4.1 Programming languages

Most important programming languages used for expert system applications are generally either problem-oriented languages such as FORTRAN and PASCAL, or symbol-manipulation

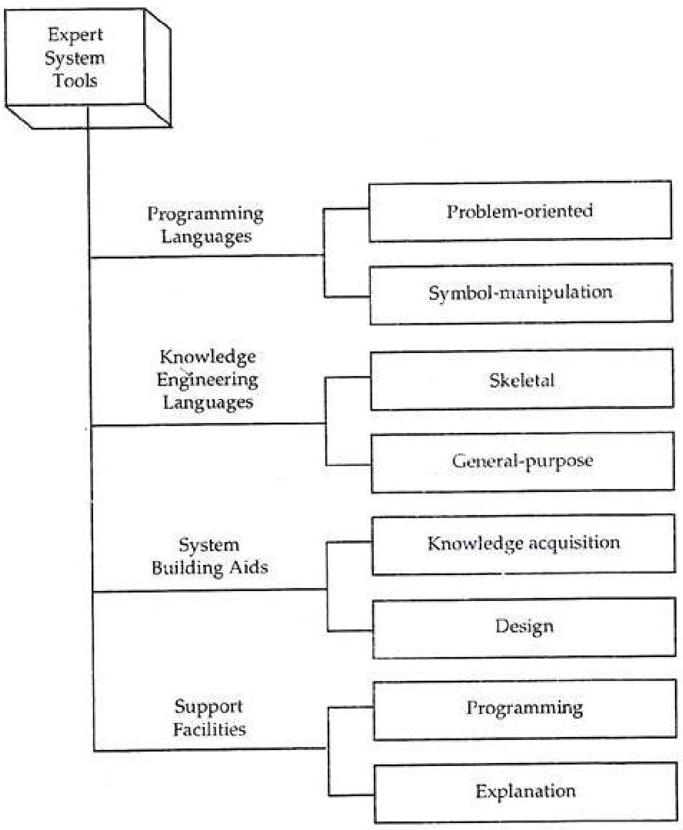


Figure 10.2: Types of expert systems

languages such as LISP and PROLOG. Symbol-manipulation languages are designed for artificial intelligence applications; for example, LISP has mechanism for manipulating symbols in the form of list structures. A list is simply a collection of items enclosed by parenthesis, where each item can be either a symbol or another list. List structures are useful building blocks for representing complex concepts.

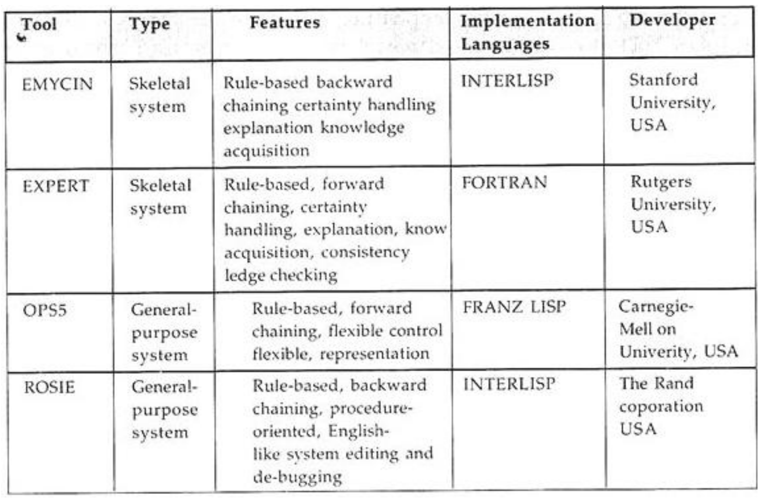
The most popular and widely used programming language for expert system applications are LISP and PROLOG. Symbol-manipulation languages like these are more suitable for work in artificial intelligence, although a few expert systems have been written in problem-oriented languages like FORTRAN and PASCAL.

Expert systems are also written in CLIPS (C Language Integrated Production System developed in mid 1980s). The major advantages of these languages as compared to conventional programming languages, is the simplicity of the addition, elimination or substitution of new rules and memory management capabilities.

10.4.2 Knowledge engineering languages

A knowledge engineering language is a sophisticated tool for developing expert systems. It consists of an expert system building language integrated into an extensive support environment. A knowledge engineering language is a type of programming language designed to construct and debug expert system. Knowledge engineering languages can be categorized as either skeletal systems or general-purpose systems.

A skeletal knowledge engineering language is a stripped down expert system, that is, an expert system with its domain-specific knowledge removed leaving only the inference engine and support facilities. The general-purpose knowledge engineering language can handle many different problem areas such as knowledge extraction, giving inference or making user interface though its use is rather tedious. These languages have a range of generality and flexibility. Table 10.1 shows some well known knowledge engineering languages.



Tool	Type	Features	Implementation Languages	Developer
EMYCIN	Skeletal system	Rule-based backward chaining certainty handling explanation knowledge acquisition	INTERLISP	Stanford University, USA
EXPERT	Skeletal system	Rule-based, forward chaining, certainty handling, explanation, know acquisition, consistency ledge checking	FORTRAN	Rutgers University, USA
OPS5	General-purpose system	Rule-based, forward chaining, flexible control flexible, representation	FRANZ LISP	Carnegie-Mell on Univerity, USA
ROSIE	General-purpose system	Rule-based, backward chaining, procedure-oriented, English-like system editing and de-bugging	INTERLISP	The Rand coporation USA

Table 10.1: Knowledge engineering languages for building expert systems

10.4.3 System-building aids

The system-building aids consist of programs which help acquire and represent the domain expert's knowledge and programs which help design the expert system under construction. These programs address very difficult tasks; many are research tools just beginning to evolve into practical and useful aids, although a few are offered as full-blown commercial systems.

Compared with programming and knowledge engineering languages, relatively few system-building aids have been developed. Those which exist fall into two major categories; design aids and knowledge acquisition aids. The AGE system exemplifies design aids, while TEIRSIAS, MOLE and SALT exemplify knowledge acquisition, TIMM system construction and SEEK knowledge refinement aids.

10.4.4 Tool support environment facilities

These are software packages which come with each tool to make it more user friendly and more efficient. The different components of a tool support environment are shown in Figure 10.3.

10.5 Typical expert systems

1. **Dendral**

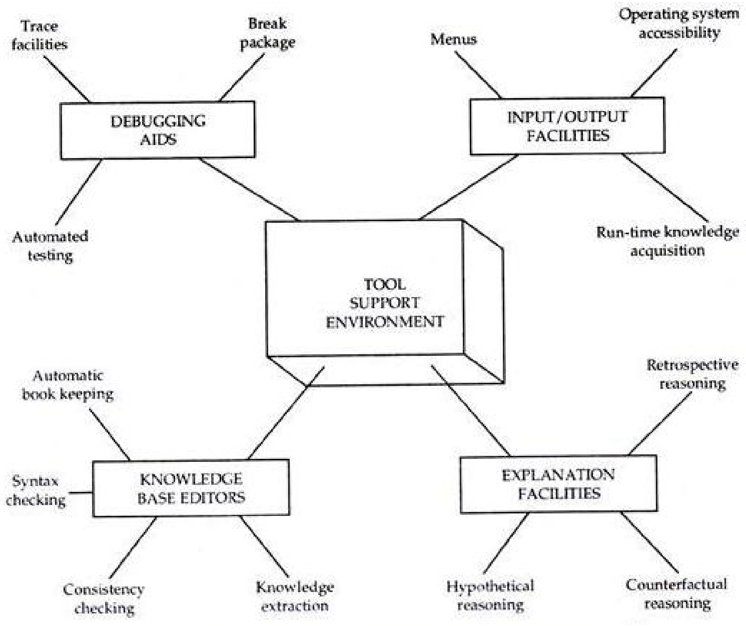


Figure 10.3: Components of a support environment for expert system tools

Dendral was a project in artificial intelligence of the 1960s. Its primary aim was to study hypothesis formation and discovery in science. For that, a specific task in science was chosen: help organic chemists in identifying unknown organic molecules, by analyzing their mass spectra and using knowledge of chemistry. It began at Stanford University in 1965 and spans approximately half the history of AI research. Dendral is considered the first expert system because it automated the decision-making process and problem-solving behavior of organic chemists.

2. **MYCIN**

   MYCIN was an early backward chaining expert system that used artificial intelligence to identify bacteria causing severe infections, such as bacteremia and meningitis, and to recommend antibiotics, with the dosage adjusted for patient's body weight. The name was derived from the antibiotics themselves, as many antibiotics have the suffix "-mycin". The Mycin system was also used for the diagnosis of blood clotting diseases. MYCIN was developed over five or six years in the early 1970s at Stanford University. It was written in Lisp.

   MYCIN operated using a fairly simple inference engine and a knowledge base of approximately 600 rules. It would query the physician running the program via a long series of simple yes/no or textual questions. At the end, it provided a list of possible culprit bacteria ranked from high to low based on the probability of each diagnosis, its confidence in each diagnosis' probability, the reasoning behind each diagnosis, and its recommended course of drug treatment.

3. **DART** (Diagnostic Assistance Reference Tool)

   DART is a joint project of the Stanford University and IBM that explores the application of artificial intelligence techniques to the diagnosis of computer faults. It assists a technician in finding the faults in a computer system. The primary goal of the DART Project was to develop programs that capture the special design knowledge and diagnostic abilities of the experts and to make them available to field engineers. The practical goal is the construction of an automated diagnostician capable of pinpointing the functional units responsible for observed malfunctions in arbitrary system configurations.

4. **XCON**

   XCON (eXpert CONfigure) is a system with the ability to select specific software to generate a computer system as per user preference, written in 1978.

5. **PXDES**

   This system could easily determine the type and the degree of lung cancer in patients based on limited data.

6. **CaDet**

   This is a clinical support system that identifies cancer in early stages.

7. **PIP**

   PIP (Process Invention Procedure) is an hierarchical expert system for the synthesis of chemical process flowsheets. It uses a combination of qualitative knowledge, that is, heuristics and quantitative knowledge, that is, design and cost calculations, arranged in a hierarchic manner. A hybrid, expert system control architecture was developed for PIP that allows these two types of knowledge-bases to interact with each other.

8. **INTERNIST**

   INTERNIST-I was designed between 1972 and 1973 to provide computer assisted diagnosis in general internal medicine by attempting to model the reasoning of clinicians.

10.6 Benefits and limitations

Some problems like scheduling manufacturing can not be adequately dealt with using mathematical algorithms. Such problems can be dealt with in an intelligent way using expert systems.

Many expert systems require large, lengthy and expensive development efforts. Expert systems lack the breadth of knowledge and the understanding of fundamental principles of a human expert. In fast moving fields such as medicine or computer science, keeping the knowledge base up to date is a critical problem. Expert systems can only represent limited forms of IF-THEN type of knowledge. There are no adequate representations for deep causal models or temporal trends. Expert systems cannot yet replicate knowledge which is intuitive, based on analogy or on common sense.

Expert system do best in automating lower-end clerical functions. They can provide electronic check lists for lower-level employees in service bureaucracies such as banking, insurance, sales and welfare agencies. The applicability of expert system to managerial problems generally involve drawing facts and interpretations from divergent sources, evaluating facts and comparing one interpretation of the facts with another and do not involve analysis or simple classification. They typically perform very limited tasks which can be performed by professional, in a few minutes and hours.

10.7 Sample questions

(a) Short answer questions

1. What is an expert system?
2. Give a definition of "Expert System".
3. List the languages and tools used in developing expert systems.
4. Give the names of some of the programming languages used to develop expert systems.
5. Give the names of some of the knowledge engineering languages used in developing expert systems.

(b) Long answer questions

1. Outline how Expert Systems can be distinguished from more conventional computer systems.
2. With the help of a neat diagram, describe the architecture of an expert system.
3. Briefly describe some of the well known expert systems.
4. What are the benefits and limitations of expert systems.

Chapter 11

Fuzzy logic

11.1 What is fuzzy logic?

The phrase "fuzzy logic" has two meanings.

- **As multivalued logic**

  In one sense the term "fuzzy logic" means the extension of ordinary logic, in which the set of truth values is the two-element set $\{0, 1\}$, to the case where the set of truth values is any finite or infinite subset of the closed interval $[0, 1]$ (the set of all numbers between 0 and 1 including both). In this sense, fuzzy logic is sometimes referred to "many-valued logic" or "multivalued logic". There are many ways in which this extension can be carried out and consequently there are many different multivalued logics.

- **As fuzzy sets**

  In the second sense, the term "fuzzy logic" refers to the use of fuzzy sets in the representation and manipulation of vague information for the purpose of making decisions or taking actions. It is the theory of fuzzy sets. A fuzzy set is a generalisation of the concept of a set.

In Section 11.2 of this chapter we discuss fuzzy logic in the first sense and in the remaining sections we discuss fuzzy logic in the second sense.

11.2 Fuzzy logic as multivalued logic

As a typical multivalued logic, we consider a three-valued logic.

11.2.1 Three-valued logic

Thinking of 0 as representing "false" and 1 as representing "true", we add a third truth value u representing "undecided" or "unknown". It is common to use $\frac{1}{2}$ instead of u. Thus, in the three-valued logic the set of truth values is the three-element set $\{0, u, 1\}$.

11.2.2 Logical connectives in three valued logic

As in classical two-valued logic, in multivalued logic also propositions are combined using the logical connectives "AND", "OR" and "NOT" (denoted respectively by $\wedge$, $\vee$ and $\neg$) to form compound propositions. The truth values of such compound propositions are determined by reference to truth tables. In the case of three-valued logic and in general in any multivalued logic different researchers have defined the truth tables is different ways. In these logics, the implication "$\Rightarrow'$" and the biconditional "$\Leftrightarrow$" are also defined by truth tables.

Lukasiewicz's definition of connectives

Here is the specification for a famous three-valued logic, that of Lukasiewicz.

$\vee$	0	u	1
0	0	u	1
u	u	u	1
1	1	1	1

$\wedge$	0	u	1
0	0	0	0
u	0	u	u
1	0	u	1

	$\neg$
0	1
u	u
1	u

$\Rightarrow$	0	u	1
0	1	1	1
u	u	1	1
1	0	u	1

$\Leftrightarrow$	0	u	1
0	1	u	0
u	u	1	u
1	0	u	1

Table 11.1: Logical connectives in Lukasiewicz logic

Kleene logic

The Kleene logic has the same tables for AND, OR, and NOT as the Lukasiewicz logic given above, but differs in its definition of implication.

$\Leftrightarrow$	0	u	1
0	1	1	1
u	u	u	1
1	0	u	1

Table 11.2: Definition of implication in Kleene logic

The only difference between the two definitions is that in Lukasiewicz logic "undecided implies undecided" is "true" where as in Kleene logic is "unknown".

11.2.3 Compound propositions in three-valued logic

Starting with a set of primitive or atomic propositions V we can build up compound propositions using this set and logical connectives. The set F of such compound propositions is the same as in classical two-valued logic. The truth values of such propositions can be computed using the truth tables of the logical connectives.

Example

Using the definitions of logical connectives in Lukasiewicz logic, construct the truth table of the compound proposition $(P \wedge Q) \Rightarrow P$.

Solution

P	Q	$P \wedge Q$	$(P \wedge Q) \Rightarrow P$
0	0	0	1
0	u	0	1
0	1	0	1
u	0	0	1
u	u	u	1
u	1	u	1
1	0	0	1
1	u	u	1
1	1	1	1

11.2.4 Other multivalued logics

There are several multivalued logics like the following:

1. Bochvar's three-valued logic
2. Belnap's four-valued logic
3. Godel's family G_k of multivalued logics where k is a positive integer greater than 2
4. A four-valued logic established by IEEE with the standard IEEE-1364
5. A nine-valued logic established by IEEE with the standard IEEE 1164

11.3 Fuzzy sets

11.3.1 Introduction

Fuzziness or vagueness can be found in many areas of daily life. This is more frequently seen in which human judgment, evaluation, and decisions are important. The reason for this is that our daily communication uses "natural languages" and a good part of our thinking is done in it. In these natural languages the meaning of words is very often vague. Examples are words such as “birds” (how about penguins, bats, etc.?), “red roses”, but also terms such as “tall men”, “beautiful women”, etc. The term “tall men” is fuzzy because the meaning of “tall” is fuzzy and dependent on the context (height of observer, culture, etc.).

11.3.2 Example

Imagine the set of all young men residing in a small town. Since the number of people residing in the town is not very large it is indeed possible to prepare a list of all persons in the town. But can we prepare a list of all young men in the town? It is not possible because the attribute of being “young” is not well defined, is vague and is imprecise. We say that he attribute is a “fuzzy” attribute. However, all of us will agree that a person aged 80 years cannot be a member of the “set of young men”! Similarly a three-year old boy cannot also be a member of the set. Also, all of us would agree that a 25-year old man residing in the city should be a member of the set.

To describe the set of young men in the town, we may arbitrarily assign a number to each person which may indicate the “youngness” of the person. We assign the smallest number to a person having the least “youngness” property and the highest value to those who have the highest “youngness” property. The assignment of numbers is purely arbitrary. However it should reflect a human being’s understanding the notion of “youngness”: a higher value must indicate a higher level of “youngness”. A set to whose elements are assigned numbers like this is called a fuzzy set. We shall consider a formal definition later. In practice, however, we usually assign values in the range 0 to 1, including both.

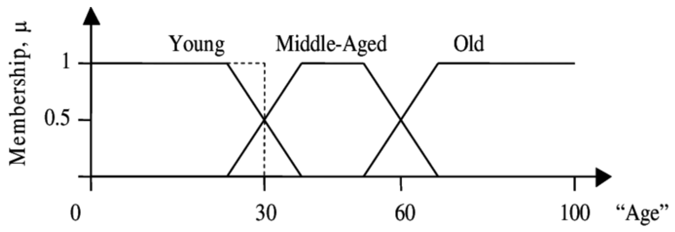


Figure 11.1: The graph of the measure “youngness”, “middle-agedness” and “oldness” vs. “age”

11.4 Crisp sets

In contrast to fuzzy sets, we shall refer to sets studied in classical set theory as “crisp sets”. In the case of crisp sets, given an object, we can say unambiguously whether the object is in the crisp set or not. In the case of crisp sets also, we can assign numbers to objects: Assign 1 if the object is in the set and assign 0 if it is not in the set. In this sense, crisp sets, that is sets considered in elementary set theory can also be regarded as special fuzzy sets.

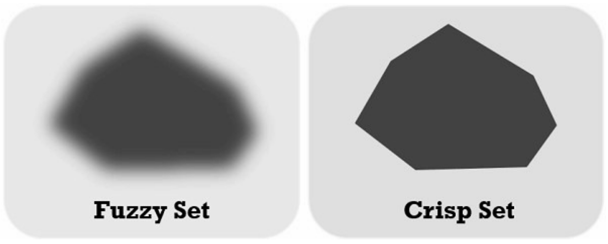


Figure 11.2: Fuzzy set vs. crisp set

11.5 Definition and examples

11.5.1 Definition

A **fuzzy set** is a pair (U, m) where U is a set (assumed to be non-empty) and $m\colon U \to [0,1]$ a function. The set U is called *universe of discourse*. The function m is called the *membership function*. For each $x \in U$, the value $m(x)$ is called the *grade of membership* of x in (U, m).

The fuzzy set (U, m) is often denoted by a single letter A. In such a case, the membership function is denoted by μ_A.

Types of membership

Let $x \in U$. Then x is said to be

- *not included* in the fuzzy set (U, m) if $m(x) = 0$ (no member),
- *fully included* if $m(x) = 1$ (full member),
- *partially included* if $0 < m(x) < 1$ (fuzzy member).

Support, kernel, α-cut

Let $A = (U, m)$ be a fuzzy set. The support of A is the crisp set

$$\text{Supp}\,(A) = \{x \in U : m(x) > 0\}.$$

The kernel of A is the crisp set

$$\text{Kern}\,(A) = \{x \in : m(x) = 1\}.$$

The α-cut (also called the α-level cut of A), denoted by $A^{\geq\alpha}$ or A_α, is defined as the crisp set

$$A^{\geq\alpha} = \{x \in U : \mu_A(x) \leq \alpha\}.$$

Notation

Let $A = (U, m)$ be a fuzzy set and let U be a finite set. Let the support of A be $\text{Supp}\,(U) = \{x_1, x_2, \ldots, x_n\}$. Then the fuzzy set A is sometimes written as a set of ordered pairs as follows:

$$A = \{(x_1, m(x_1)), (x_2, m(x_2)), \cdots, (x_n, m(x_n))\}$$

The following notation

$$A = \{m(x_1)/x_1, m(x_2)/x_2, \ldots, m(x_n)/x_n\}$$

as well as the notation

$$A = m(x_1)/x_1 + m(x_2)/x_2 + \ldots + m(x_n)/x_n$$

are also used to denote the fuzzy set A. Still others write

$$A = \left\{ \frac{m(x_1)}{x_1}, \frac{m(x_2)}{x_2}, \ldots, \frac{m(x_n)}{x_n} \right\}.$$

Note that in these representations, the symbols "/" and "+" are just symbols and they do not represent the corresponding arithmetical operations.

Crisp set as a fuzzy set

Let A be a crisp subset of the universe of discourse U. A can be treated as a fuzzy set $A = (U, m)$ where the membership function m is defined as

$$m(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

The membership function of a crisp set is called the characteristic function of the set.

11.5.2 Examples

Example 1

Consider the set

$$U = \{5, 10, 15, 20, 30, 35, 40, 45, 60, 70\}$$

and the function $m : U \to [0, 1]$ defined by the following table:

x	5	10	15	20	30	35	40	45	60	70
$m(x)$	1	1	1	1	0.6	0.5	0.4	0.2	0	0

$A = (U, m)$ is a fuzzy set. The support of A is

$$\text{Supp}\,(A) = \{5, 10, 15, 20, 30, 35, 40, 45\}$$

and the kernel of A is

$$\text{Kern}\,(A) = \{5, 10, 15, 20\}.$$

The fuzzy set may be represented as

$$A = \{1/5, 1/10, 1/15, 1/20, 0.6/30, 0.5/35, 0.4/40, 0.2/45\}$$

or as

$$A = 1/5 + 1/10 + 1/15 + 1/20 + 0.6/30 + 0.5/35 + 0.4/40 + 0.2/45.$$

Example 2

A realtor wants to classify the houses he offers to his clients. One indicator of comfort of these houses is the number of bedrooms in it. Let

$$U = \{1, 2, 3, 4, ..., 10\}$$

be the set of available types of houses described by $x =$ number of bedrooms in a house. Then the fuzzy set "comfortable type of house for a 4-person family" may be described as

$$A = \{0.2/1, 0.5/2, 0.8/3, 1.0/3, 0.7/5, 0.3/6)\}.$$

Example 3

The fuzzy set defined as "integers close to 10" may be specified by

$$A = 0.1/7 + 0.5/8 + 0.8/9 + 1/10 + 0.8/11 + 0.5/12 + 0.1/13$$

Example 4

Let the universe $U = \mathbb{R}$ be the set of all real numbers. Then the fuzzy set A described as "the set of real numbers close to 10" can be specified by the membership function

$$\mu_A(x) = \frac{1}{1 + (x - 10)^2}.$$

11.5.3 Some special membership functions

Let the universe of discourse $U = \mathbb{R}$ be the set of all real numbers. The following membership functions define certain special fuzzy sets $A = (U, m)$ which are useful in some applications.

1. **Triangular membership function**

$$\mu_A(x) = \begin{cases} 0 & x \leq a \\ \dfrac{x-a}{b-a} & a < x \leq b \\ \dfrac{x-b}{c-b} & b < x \leq c \\ 0 & c < x \end{cases}$$

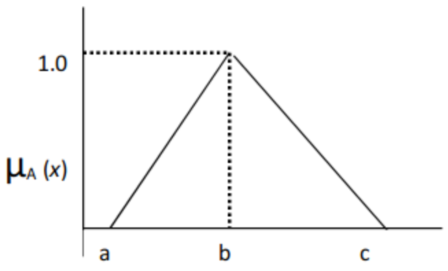


Figure 11.3: Graph of the triangular membership function

2. **Trapezoidal membership function**

$$\mu_A(x) = \begin{cases} 0 & x \leq a \\ \dfrac{x-a}{b-a} & a < x \leq b \\ 1 & b < x \leq c \\ \dfrac{x-c}{d-b} & c < x \leq d \\ 0 & d < x \end{cases}$$

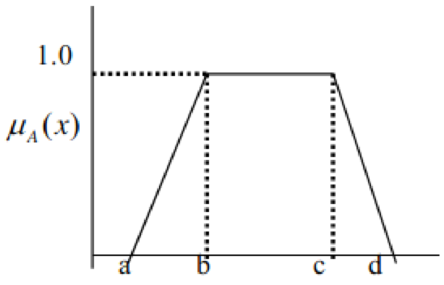


Figure 11.4: Graph of the trapezoidal membership function

3. **Gaussian membership function**

$$\mu_A(x) = \exp\left(-\frac{1}{2}\left|\frac{x-m}{s}\right|^k\right)$$

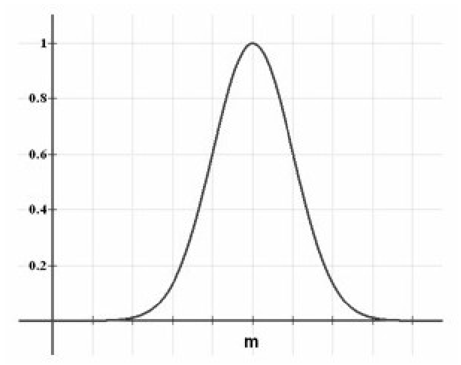


Figure 11.5: Graph of the Gaussian membership function

4. **Sigmoid membership function**

$$\mu_A(x) = \frac{1}{1+e^{-a(x-m)}}$$

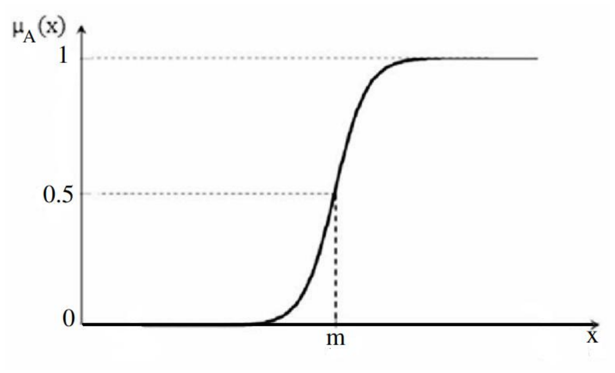


Figure 11.6: Graph of the sigmoid membership function

11.6 Set-theoretic operations for fuzzy sets

11.6.1 Definitions

Let A and B be two fuzzy sets in a universe U.

1. **Equality**

   A and B are said to be equal, denoted by $A = B$, if $\mu_A(x) = \mu_B(x)$ for all x in U.

2. **Union**

   The union of A and B, denoted by $A \cup B$, is the fuzzy set C whose membership function is defined by

$$\mu_C(x) = \max\{\mu_A(x), \mu_B(x)\}, \quad x \in U.$$

3. **Intersection**

   The intersection of A and B, denoted by $A \cap B$, is the fuzzy set D whose membership function is defined by

$$\mu_D(x) = \min\{\mu_A(x), \mu_B(x)\}, \quad x \in U.$$

4. **Complement**

   The complement of A, denoted by $\overline{A}$, is the fuzzy set E whose membership function is defined by

$$\mu_E(x) = 1 - \mu_A(x), \quad x \in U.$$

Remarks

1. The definitions of the membership functions of the union, intersection and complement of fuzzy sets may be given in the following forms also:

$$\begin{aligned}\mu_{A\cup B}(x) &= \max\{\mu_A(x), \mu_B(x)\}\\ \mu_{A\cap B}(x) &= \min\{\mu_A(x), \mu_B(x)\}\\ \mu_{\overline{A}}(x) &= 1 - \mu_A(x)\end{aligned}$$

2. The support of a fuzzy set is a crisp set. We have

$$\begin{aligned}\text{Supp}\,(A \cup B) &= \text{Supp}\,(A) \cup \text{Supp}\,(B)\\ \text{Supp}\,(A \cap B) &= \text{Supp}\,(A) \cap \text{Supp}\,(B)\\ \text{Supp}\,(\overline{A}) &= \overline{\text{Supp}\,(A)}\end{aligned}$$

3. If A is a fuzzy set in a universe U then in general we have

$$A \cup \overline{A} \neq U, \qquad A \cap \overline{A} \neq \varnothing.$$

   However, equalities hold if A is a crisp set considered as a fuzzy set.

11.6.2 Examples

Example 1

Given the fuzzy sets

$$\begin{aligned}A &= \{0.3/2, 0.4/3, 0.1/4, 0.8/5, 1.0/6\}\\ B &= \{0.7/4, 0.5/5, 1.0/6, 0.02/7, 0.75/8\}\end{aligned}$$

find $A \cup B$ and $A \cap B$.

Solution

We have:

$$\begin{aligned}\text{Supp}\,(A) &= \{2, 3, 4, 5, 6\}\\ \text{Supp}\,(B) &= \{4, 5, 6, 7, 8\}\end{aligned}$$

Let us compute $A \cup B$.

$$\begin{aligned}\text{Supp}\,(A \cup B) &= \text{Supp}\,(A) \cup \text{Supp}\,(B)\\ &= \{2, 3, 4, 5, 6, 7, 8\}\end{aligned}$$

We have

$$\mu_{A \cup B}(x) = \max\{\mu_A(x), \mu_B(x)\}.$$

Hence

$$\begin{aligned}
\mu_{A\cup B}(2) &= \max\{\mu_A(2), \mu_B(2)\} \\
&= \max\{0.3, 0\} \quad (2 \notin \text{Supp}\,(B)) \\
&= 0.3 \\
\mu_{A\cup B}(3) &= \max\{\mu_A(3), \mu_B(3)\} \\
&= \max\{0.4, 0\} \quad (3 \notin \text{Supp}\,(B)) \\
&= 0.4 \\
\mu_{A\cup B}(4) &= \max\{\mu_A(4), \mu_B(4)\} \\
&= \max\{0.1, 0.7\} \\
&= 0.7 \\
\cdots &= \cdots \\
\mu_{A\cup B}(8) &= \max\{\mu_A(8), \mu_B(8)\} \\
&= \max\{0, 0.75\} \quad (8 \notin \text{Supp}\,(A)) \\
&= 0.75
\end{aligned}$$

Therefore

$$A \cup B = \{0.3/2, 0.4/3, 0.7/4, 0.8/5, 1.0/6, 0.02/7, 0.75/8\}.$$

Now, let us calculate $A \cap B$.

$$\begin{aligned}
\text{Supp}\,(A \cap B) &= \text{Supp}\,(A) \cap \text{Supp}\,(B) \\
&= \{4, 5, 6\}
\end{aligned}$$

We have

$$\mu_{A\cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}.$$

Hence

$$\begin{aligned}
\mu_{A\cap B}(4) &= \min\{\mu_A(4), \mu_B(4)\} \\
&= \min\{0.1, 0.7\} \\
&= 0.1 \\
\mu_{A\cap B}(5) &= \min\{\mu_A(5), \mu_B(5)\} \\
&= \max\{0.8, 0.5\} \\
&= 0.5 \\
\mu_{A\cap B}(6) &= \min\{\mu_A(6), \mu_B(6)\} \\
&= \max\{1, 0, 1, 0\} \\
&= 1.0
\end{aligned}$$

Therefore

$$A \cap B = \{0.1/4, 0.5/5, 1.0/6\}$$

Example 2

Let the universe be $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and let

$$A = \{0.2/1, 0.4/2, 0.6/3, 0.8/4, 1.0/5\}.$$

Compute $\bar{A}$.

Solution

By definition

$$\mu_{\overline{A}}(x) = 1 - \mu_A(x), \text{ for all } x \in U.$$

Therefore, we have:

$$\overline{A} = \{0.8/1, 0.6/2, 0.4/3, 0.2/4, 1.0/6, 1.0/7, 1.0/8, 1.0/9, 1.0/10\}.$$

11.7 Some more definitions

1. **Fuzzy empty set**

   A fuzzy set A is said to be empty if $\mu_A(x) = 0$ for all $x \in U$, that is, if Supp $(A) = \varnothing$.

2. **Fuzzy subset**

   A fuzzy set A is said to be a subset of a fuzzy set B, denoted by $A \subseteq B$, if $\mu_A(x) \leq \mu_B(x)$ for all $x \in U$.

3. **Disjoint fuzzy sets**

   Two fuzzy sets A and B are said to be disjoint if $\mu_A(x) = 0$ or $\mu_B(x) = 0$ for all $x \in U$, that is, if Supp $(A \cap B) = \varnothing$.

Examples

1. Let

   $$A = \{0.12/2, 0.23/4\}, \quad B = \{0.15/2, 0.37/4, 0.15/6\},$$

   Then $A \subseteq B$.

2. Let $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and let

   $$A = \{0.1/1, 0.2/2, 0.3/3\}, \quad B = \{0.2/6, 0.4/7, 0.6/8\}.$$

   Then

   $$\begin{aligned} \text{Supp}\,(A) &= \{1, 2, 3\} \\ \text{Supp}\,(B) &= \{6, 7, 8\} \\ \text{Supp}\,(A \cap B) &= \text{Supp}\,(A) \cap \text{Supp}\,(B) \\ &= \varnothing \end{aligned}$$

   Hence A and B are disjoint fuzzy sets.

11.8 Properties of set operations on fuzzy sets

Let A, B, C be fuzzy sets in some universe X. Then we have:

1. **Commutativity**

   $A \cup B = B \cup B$

   $A \cap B = B \cap A.$

2. **Associativity**
   $A \cup (B \cup C) = (A \cup B) \cup C)$
   $A \cap (B \cap C) = (A \cap B) \cap C$
3. **Distributivity**
   $A \cup (B \cap C) = (A \cup B) \cap (A \cup C),$
   $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
4. **Identity**
   $A \cup \varnothing = A, A \cup X = A$
   $A \cap \varnothing = \varnothing, A \cap X = A$
5. **Involution**
   $\overline{(\overline{A})} = A.$
6. **De Morgan's laws**
   $\overline{A \cup B} = \overline{A} \cap \overline{B}$
   $\overline{A \cap B} = \overline{A} \cup \overline{B}.$

Example

Given the universe $U = \{a, b, c, d, e, f, g, h\}$ and the fuzzy sets

$$\begin{aligned} A &= \{0.1/a, 0.2/b, 0.3/c, 1.0/d, 0.1/e, 0.5/h\} \\ B &= \{0.75/b, 0.2/c, 0.35/d, 0.4/e, 0.5/f\} \\ C &= \{1.0/e, 0.9/f, 0.8/g, 0.7/h\} \end{aligned}$$

verify the distributivity properties.

Solution

$$\begin{aligned} B \cup C &= \{0.75/b, 0.2/c, 0.35/d, 1.0/e, 0.9/f, 0.8/g, 0.7/h\} \\ A \cap (B \cup C) &= \{0.2/b, 0.2/c, 0.35/d, 0.1/e, 0.5/h\} \\ A \cap B &= \{0.2/b, 0.2/c, 0.35/d, 0.1/e\} \\ A \cap C &= \{0.1/e, 0.5/h\} \\ (A \cap B) \cup (A \cap C) &= \{0.2/b, 0.2/c, 0.35/d, 0.1/e, 0.5/h\} \end{aligned}$$

Hence we have

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

In a similar way, the other distributivity property can be verified.

11.9 Fuzzy variables

An ordinary variable, or a crisp variable, is a variable whose possible values are elements of a given set. For example, a real variable is a variable whose possible values are elements of the set of real numbers.

11.9.1 Definition

A fuzzy variable x is a variable whose possible values are fuzzy sets in some universe of discourse.

11.9.2 Examples

Example 1

We define a fuzzy variable x as follows. Let the universe of discourse be

$$U = \{1, 2, 3, 4, 5\}.$$

The possible values of x are fuzzy sets in U. A possible value of x is

$$A = \{0.1/1, 0.7/2, 0.8/3, 0.2/4, 0/5\},$$

or

$$B = \{1.0/1, 0.8/2, 0.6/3, 0.4/4, 0.2/5\}.$$

Example 2

Consider a variable which denotes the age of a person in years. We may denote this variable by *age*. As a crisp variable, *age* can assume nonnegative integers as a values.

We may take *age* as a fuzzy variable. To do this, we have to define a universe of discourse U which we may take as the set of all nonnegative integers. As a fuzzy variable the possible values can that can be assigned to *age* are fuzzy sets in U. These fuzzy sets may denoted by linguistic terms indicative of the age of a person, such as *infant*, *young*, *middle-aged*, *old*, etc. Thus the fuzzy variable *age* can be thought of as a variable whose possible values are such linguistic terms. However, it is to be emphasised that the value is not the linguistic term but the fuzzy set represented by the term.

11.10 Example for application of fuzzy sets

We explain the working of a conventional fuzzy room cooler shown schematically in Figure 11.7.

The device

The cooler is implemented using a fan encased in a box with wool or hay that is continuously moistened with a trickle of water. A motorised pump controls the rate of flow of water. Two sensors measures the fan motor speed and the temperature.

Objective

The basic aim is to achieve a smooth control and to save water. We have to control the rate of flow of water based on fan motor speed (measured in rpm's) and temperature (measured in $^\circ$C). To make the problem precise, let it be required to find the rate of flow of water if the temperature is 42°C and fan motor speed is 31 rpm.

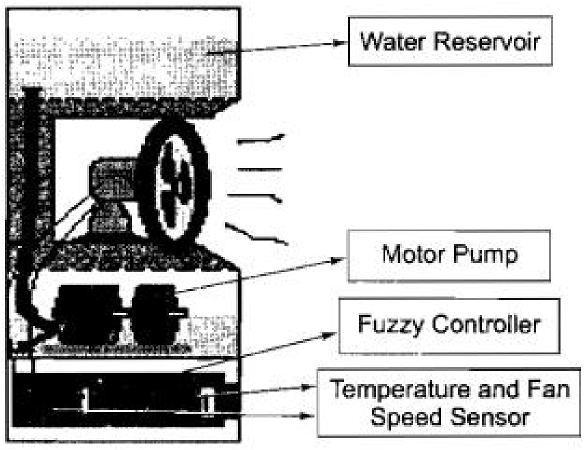


Figure 11.7: A sectional view of a fuzzy room cooler

Variables

The input variables of the system are "*temperature*" and "*fan-motor-speed*". We take them as fuzzy variables. We assign the fuzzy sets denoted by "*Cold, Cool, Moderate, Warm, Hot*" as the values of the variable *temperature* and the fuzzy sets denoted by "*Slack, Low, medium, Brisk, Fast*" as the values of *fan-motor-speed*. The output variable is "*water-flow-rate*" and it may take the fuzzy sets "*Strong-Negative (SN), Negative (N), Low-Negative (LN), Medium (M), Low-Positive (LP), Positive (P), High-Positive*" as its values.

Membership functions

Based on observations and experimentation we have to construct the membership functions to the various fuzzy sets introduced above. We assume that the membership functions are as shown in Figure 11.8.

Rules for action

The values for the output variable depends on the values of the input variable. In logic controllers, the values of the output variable are computed based on certain rules. These rules are formulated by the designer based on real life experiences. For example, for the room cooler the rules could be the following and probably many more such rules.

R.1 If *temperature* is *Hot* **and** *fan-motor-speed* is *Slack* then *water-flow-rate* is *HP*.

R.2 If *temperature* is *Hot* **and** *fan-motor-speed* is *Low* then *water-flow-rate* is *HP*.

R.3 If *temperature* is *Hot* **and** *fan-motor-speed* is *Medium* then *water-flow-rate* is *P*.

R.4 If *temperature* is *Hot* **and** *fan-motor-speed* is *Brisk* then *water-flow-rate* is *HP*.

R.5 If *temperature* is *Warm* **and** *fan-motor-speed* is *Medium* then *water-flow-rate* is *LP*.

R.6 If *temperature* is *Warm* **and** *fan-motor-speed* is *Brisk* then *water-flow-rate* is *P*.

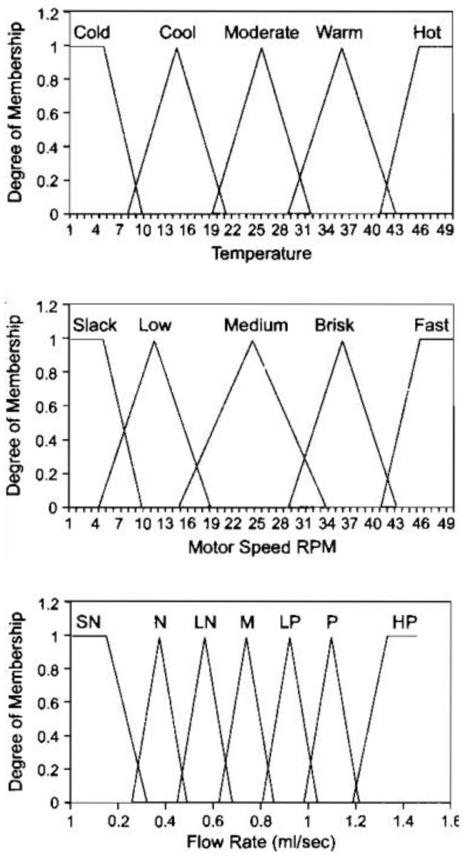


Figure 11.8: Graphs of membership functions

R.7 If *temperature* is *Cool* **and** *fan-motor-speed* is *Low* then *water-flow-rate* is *N*.

R.8 If *temperature* is *Moderate* **and** *fan-motor-speed* is *Low* then *water-flow-rate* is *M*.

Fuzzification

We have to express the observed values of 42°C and 31 rpm as values of the fuzzy variables *temperature* and *fan-motor-speed*.

From the graphs of the membership functions of the values of the fuzzy variable *temperature*, we see that a value of 42°C corresponds to the fuzzy set *Warm* with a membership grade of 0.142 and also to the fuzzy set *Hot* with a membership grade of 0.2.

Similarly from From the graphs of the membership functions of the values of the fuzzy variable *fan-motor-speed*, we see that a value of 31 rpm corresponds to the fuzzy set *Medium* with a membership grade of 0.25 and also to the fuzzy set *Brisk* with a membership grade of 0.286.

This process of expressing crisp values as values of fuzzy variables is known as fuzzification.

Application of rules

Since we have two possible fuzzy values for *temperature* and two possible fuzzy values for *fan-motor-speed* we have the combinations of values as shown Table 11.3. The table also shows the applicable rules, the value of the output variable obtained by applying the rule and further the membership grade of the output value. Note that the membership grade of the output variable *water-flow-rate* is taken as the minimum of the membership grades of the two input variables.

temperature	*fan-motor-speed*	Applicable rule	*water-flow-rate*
Hot/0.2	*Medium*/0.25	R3	*P*/0.2
Hot/0.2	*Brisk*/0.286	R4	*HP*/0.2
Warm/0.142	*Medium*/0.25	R5	*LP*/0.142
Warm/0.142	*Brisk*/0.286	R6	*LP*/0.142

Table 11.3: Output values and their membership grades

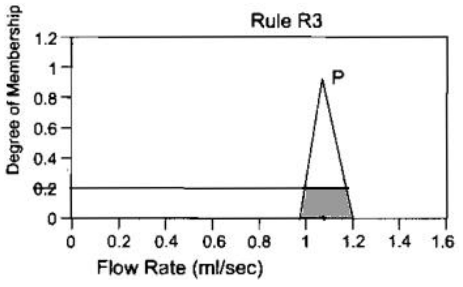


Figure 11.9: Defuzzification

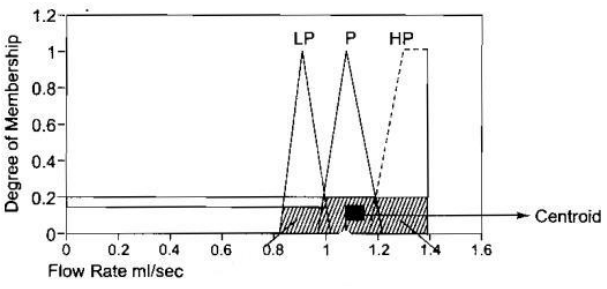


Figure 11.10: Defuzzification

Defuzzification

From the above set of possible fuzzy values for the output variable, we have to derive a crisp value for the output variable. This process is known as defuzzification. One method for doing this known as the "centre of gravity method".

To obtain the crisp value, we consider value P with membership grade 0.2 and consider the graph of the membership function of the fuzzy variable P. In this graph, we consider the shaded region shown in Figure 11.9. In a similar way, by considering the the other possible output values and their member ship grades, we construct shaded regions in the graphs of the respective membership functions. In this way, we get four shaded regions. We next construct in one graph a composite region consisting of all the four shaded regions as in Figure 11.10.

The x-coordinate of the centre of gravity, or the centroid, of this composite region is the defuzzified crisp value of the output variable *water-flow-rate*.

11.11 Sample questions

(a) Short answer questions

1. What is fuzzy logic?
2. Give an example of a three-valued logic.
3. Define fuzzy set and give an example.
4. What is a membership function of a fuzzy set?
5. Define the support and kernel of a fuzzy set and illustrate with examples.
6. Explain why a crisp set can also be considered as a fuzzy set.
7. Define the union and intersection of fuzzy sets. Illustrate with examples.
8. Define the complement of a fuzzy set. Illustrate with an example.
9. Define disjoint fuzzy sets and give an example.
10. Using an example of a fuzzy set A verify that $\overline{(\overline{A})} = A$.
11. Define a fuzzy variable and give an example.
12. What is the purpose of defuzzycation? Name at least one method used for defuzzycation.

(b) Long answer questions

1. Given the following fuzzy sets

$$A = \{0.1/1, 0.3/2, 0.45/3\}, \quad B = \{0.15/1, 0.34/2\}$$

find $A \cup B$ and $A \cap B$.

2. Let
$$A = 0.5/1 + 0.9/2 + 1/5, \quad B = 0.7/2 + 0.9/3 + 0.1/4.$$
Compute $A \cup B$ and $A \cap B$.

3. Suppose $U = \{1, 2, 3, 4, 5\}$ and $A = 0.5/1 + 0.3/3 + 1/5$. Compute:
   (a) $\overline{A}$
   (b) $\overline{A} \cap A$
   (c) $\overline{A} \cup A$

4. Let $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and let
$$A = \{0.4/1, 0.7/3, 0.5/5, 0.8/7\}, B = \{0.5/1, 0.2/2, 1.0/6, 0.1/8\}.$$
Verify De Morgan's laws.

5. Consider two fuzzy subsets of the set $X = \{a, b, c, d, e\}$ defined by
$$A = \{1/a, 0.3/b, 0.2/c0.8/d\}, \quad B = \{0.6/a, 0.9/b, 0.1/c, 0.3/d, 0.2/e\}.$$
Compute the following: Supp (A), Supp(B), $A \cup B$, $A \cap B$, $\overline{A}$ and $\overline{B}$.

6. Let
$$A = 0.2/1 + 0.5/2 + 0.7/3 + 1.0/4 + 0.8/5 + 0.4/6 + 0.2/7.$$
Compute the α-level set of A for $\alpha = 0.5$.

7. Given the fuzzy sets
$$C = \left\{\frac{0.2}{1}, \frac{0.5}{2}, \frac{0.8}{3}, \frac{1.0}{4}, \frac{0.7}{5}, \frac{0.3}{6}\right\}, \quad D = \left\{\frac{0.2}{3}, \frac{0.4}{4}, \frac{0.6}{5}, \frac{0.8}{6}, \frac{1.0}{8}, \frac{1.0}{9}, \frac{1.0}{10}\right\}$$
compute $C \cap D$ and $C \cup D$.

8. Let
$$A = 0.1/2 + 0.5/3 + 0.4/4 + 0.5/5, \quad B = 0.1/2 + 0.6/3 + 1/4 + 0.6/5 + 0.1/6.$$
Is it true that $B \not\subset A$?

9. Consider these (very subjective) membership functions for the height of a person:

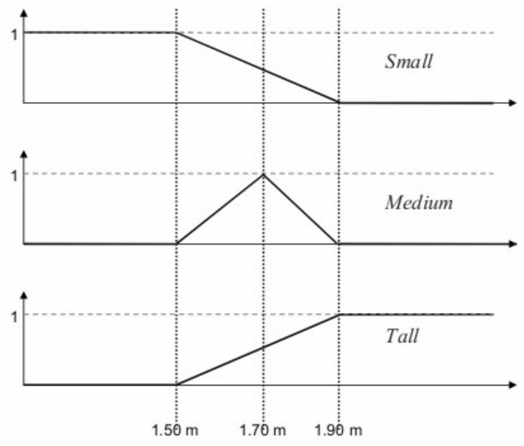


Figure 11.11: Graphical representations of membership functions

Compute the graphical representation of the membership function of the following fuzzy sets:

(a) $Tall \cap Small$

(b) $(Tall \cup Medium) - Tall$

10. Construct a fuzzy set (universal set and membership function) for the concept "early in the morning". (Hint: Choose the universal set as $\{1, 2, \ldots, 23, 24\}$ (times of a day) and define a membership function to indicate "early in the morning".)

Index

This page is intentionally left blank.

Lecture Notes in Artificial Intelligence

Dr V N Krishnachandran

Department of Computer Applications (MCA)
Vidya academy of Science and Technology
Thrissur – 680 501