T.Y.B.Sc. (C. S.)

SEMESTER - V (CBCS)

ARTIFICIAL INTELLIGENCE

SUBJECT CODE: USCS501

Prof. (Dr.) D. T. Shirke
Offg. Vice Chancellor
University of Mumbai, Mumbai

Prin. Dr. Ajay Bhamare
Offg. Pro Vice-Chancellor,
University of Mumbai

Prof. Prakash Mahanwar
Director,
IDOL, University of Mumbai

Programme Co-ordinator : **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology IDOL,
Univeristy of Mumbai - 400098

Course Co-ordinator : **Ms. Mitali Vijay Shewale**
Doctoral Research,
Veermata Jijabai Technological Institute
Matunga, Mumbai.

Editor : **Mr. Anish Raut**
Assistant Manager,
Dahua Technology India Pvt.Ltd., Mumbai.

Course Writers : **Dr. Rajeshri Pravin Shinkar**
Assistant Professor,
SIES (Nerul) College of Arts, Science &
Commerce.Navi Mumbai.

: **Dr. Rajendra Patil**
Principal, Bunts Sangha Anna Leela College of
Commerce and Economics, Mumbai.

: **Ms. Mitali Vijay Shewale**
Doctoral Research,
Veermata Jijabai Technological Institute
Matunga, Mumbai.

August 2023, Print - 1

Published by : Director,
Institute of Distance and Open Learning,
University of Mumbai,
Vidyanagari,Mumbai - 400 098.

DTP composed and Printed by: Mumbai University Press

CONTENTS

T.Y.B.Sc. (C. S.)

SEMESTER - V (CBCS)

ARTIFICIAL INTELLIGENCE

SYLLABUS

Course: USCS501	TOPICS (Credits : 03 Lectures/Week:03) Artificial Intelligence	
Objectives: Artificial Intelligence (AI) and accompanying tools and techniques bring transformational changes in the world. Machines capability to match, and sometimes even surpass human capability, make AI a hot topic in Computer Science. This course aims to introduce the learner to this interesting area. **Expected Learning Outcomes:** After completion of this course, learner should get a clear understanding of AI and different search algorithms used for solving problems. The learner should also get acquainted with different learning algorithms and models used in machine learning.		
Unit I	**What Is AI:** Foundations, History and State of the Art of AI. **Intelligent Agents:** Agents and Environments, Nature of Environments, Structure of Agents. **Problem Solving by searching:** Problem-Solving Agents, Example Problems, Searching for Solutions, Uninformed Search Strategies, Informed (Heuristic) Search Strategies, Heuristic Functions.	**15L**
Unit II	**Learning from Examples:** Forms of Learning, Supervised Learning, Learning Decision Trees, Evaluating and Choosing the Best Hypothesis, Theory of Learning, Regression and Classification with Linear Models, Artificial Neural Networks, Nonparametric Models, Support Vector Machines, Ensemble Learning, Practical Machine Learning	**15L**

Unit III	**Learning probabilistic models:** Statistical Learning, Learning with Complete Data, Learning with Hidden Variables: The EM Algorithm. Reinforcement learning: Passive Reinforcement Learning, Active Reinforcement Learning, Generalization in Reinforcement Learning, Policy Search, Applications of Reinforcement Learning.	**15L**

Textbook(s):

1) Artificial Intelligence: A Modern Approach, Stuart Russell and Peter Norvig,3rd Edition, Pearson, 2010.

Additional Reference(s):

1) Artificial Intelligence: Foundations of Computational Agents, David L Poole,Alan K. Mackworth, 2nd Edition, Cambridge University Press ,2017.
2) Artificial Intelligence, Kevin Knight and Elaine Rich, 3rd Edition, 2017
3) The Elements of Statistical Learning, Trevor Hastie, Robert Tibshirani and Jerome Friedman, Springer, 2013

1

ARTIFICIAL INTELLIGENCE

Unit Structure :

1.0 OBJECTIVES

After completing this chapter learner will able to:

- Understand What is Artificial Intelligence?
- Foundations of Artificial Intelligence
- Categories of Intelligent System
- Components of Artificial Intelligence.
- History of Artificial Intelligence.
- Applications of AI
- Problems of AI

1.1 WHAT IS AI?

It is a branch of Computer Science that pursues creating the computers or machines as intelligent as human beings.

It is the science and engineering of making intelligent machines, especially intelligent computer programs.

It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biological observable.

1.1.1 Introduction

Artificial intelligence (AI) is a relatively recent branch of science and engineering. Soon after World War II, work began in earnest, and the term was coined in 1956. In addition to molecular biology, AI is frequently mentioned by scientists from other fields as the "field I'd most like to be in."

A physics student may fairly believe that all of the good ideas have already been taken.

Galileo, Newton, and Einstein are three of the most famous scientists of all time.

Definition: Artificial Intelligence is the study of how to make computers do things, which, at the moment, people do better.
According to the father of Artificial Intelligence, John McCarthy, it is “The science and
engineering of making intelligent machines, especially intelligent computer programs”.

Artificial Intelligence is a way of making a computer, a computer-controlled robot, or a
software thinks intelligently, in the similar manner the intelligent humans think.

AI is accomplished by studying how human brain thinks and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

It has gained prominence recently due, in part, to big data, or the increase in speed, size and variety of data businesses are now collecting. AI can perform tasks such as identifying patterns in the data more efficiently than humans, enabling businesses to gain more insight out of their data.
From a business perspective AI is a set of very powerful tools, and methodologies for using those tools to solve business problems.
Intelligence

Because our intelligence is so vital to us, we call ourselves Homosapiens-man the wise. For thousands of years, scientists have attempted to

comprehend how we think: how a small amount of matter can see, comprehend, predict, and manage a world considerably larger and more sophisticated than itself. Artificial intelligence, or Al, is a field that goes even further.

1.2 FOUNDATIONS OF AI

Now we discuss the various disciplines that contribute ideas, viewpoints and techniques to AI.

Philosophy provide base to AI by providing theories of relationship between physical brain and mental mind, rules for drawing valid conclusions. It also provides information about knowledge origins and the knowledge needs to actions.

Mathematics gives strong base to AI to develop concrete and formal rules for drawing valid conclusions, various methods for date computation and techniques to deal with uncertain information.

Economics support AI to make decisions so as to maximum payoff and make decisions under certain circumstances.

Neuroscience gives information which is related to brain processing which helps AI to developed date processing theories.

Psychology provides strong concepts of how humans and animals act which helps AI for developing process of thinking and actions.

Historically there are four approaches to AI have been followed, each by different people with different methods. A rationalist approach involves a combination of mathematics and engineering. The various group have both disparaged and helped each other.

Intelligent Systems

In order to design intelligent systems, it is important to categorize them into four categories (Luger and Stubberfield 1993), (Russell and Norvig, 2003)

1. Systems that think like humans
2. Systems that think rationally
3. Systems that behave like humans
4. Systems that behave rationally

	Human-Like	**Rationally**
Think :	Cognitive Science Approach "Machines that think like humans"	Laws of thought Approach "Machines that think Rationally"
Act :	Turing Test Approach "Machines that behave like humans"	Rational Agent Approach "Machines that behave Rationally"

1.2.1 Acting Humanlay: The Turing Test Arrpoach

Turing test: a method of determining intellect. Turing Test was conceived by Alan Turing in 1950. He proposed a test based on common characteristics that can be matched to the most intelligent entity on the planet – humans.

Computer would need to process the following capabilities:

I) Natural language processing - In order for it to be able to communicate effectively in English.

II) Knowledge representation to store what it knows, what it hears.

III) Automated reasoning to make use of stored information to answer questions being asked and to draw conclusions.

IV) Machine learning to adapt to new circumstances and to detect and make new predictions by finding patterns.

V) Turing also proposed that the interrogator and the computers engage physically. The Turing test avoids this, but the Total Turing Assess includes a video signal to allow the interrogator to test the subject's perceptual abilities, as well as the ability to pass physical things "through the hatch."

VI) To pass total Turing test in addition, computer will need following capabilities.

VII) Computer vision to perceive objects.

VIII) Robotics to manipulate objects.

1.2.2 Thinking Rationally: The “Laws of Thought” Approach

Because we're claiming that the given software thinks like a human, we need to understand how humans think. The theory of human minds must be investigated in order to achieve this. There are two methods for doing so: introspection (trying to catch our own thoughts as they pass us by) and psychological experiments.

We can argue that some of the program's mechanisms are also operating in human mode if computer programmers’, input, output, and timing behaviors’ mirror similar human behaviors. Cognitive science is an interdisciplinary study that draws together computer models from AI and experimental approaches from psychology to try to build accurate and testable explanations of how the human mind works.

1.2.3 Thinking Rationally: The “Laws of Thought” Approach

“Right thinking” concept was introduced by Aristotle. Patterns for argument structures that always gives correct decisions when the premises are correct. It is known as the laws of thought approach.

"The study on mental faculties through the use of computational models. (Charmiakand McDemott, 1985)

"The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)

Law of thought were supposed to govern the operation in the mind; their study initiated the field called Logic which can be implemented to create the system which is known as intelligent system.

1.2.4 Acting Rationally: The Rational Agent Approach

Something that acts is called an agent (Latin agre-to-do). Computer agents, on the other hand, are intended to have additional characteristics that separate them from "programmes," such as independent control, time perception, adaptability to change, and the ability to take on new goals. When there is uncertainty, a rational agent is required to act in such a way that the best possible outcome is achieved. The laws of thought emphasis on correct inference which should be incorporated in rational agent.

“Computational Intelligence is the study of the design of intelligent agents.” By Poole et at, 1998

1.2.5 Categorization of Intelligent Systems

There are various types and forms of AI. The various categories of AI can be based on the capacity of intelligent program or what the program is able to do. Consideration of the above factors there are three main categories:

1) Weak AI (Artificial Narrow Intelligence)
2) Strong AI (Artificial General Intelligence)
3) Artificial Super Intelligence

1) **Weak AI :** Weak AI is AI that focuses on a single task. It isn't an intellect that can be used in a variety of situations. Narrow intelligence or weak AI refers to an intelligent agent that is designed to solve a specific problem or perform a certain task. For example, it took years of AI research to beat the chess grandmaster, and humans still haven't beaten the machines at chess since then. But that's all it can do, and it does it exceptionally well.

2) **Strong AI :** Strong AI, often known as general AI, refers to machine intelligence proven in the performance of any cognitive task that a person can execute. It is far more difficult to construct powerful AI than it is to develop weak AI. Artificial general intelligence machines can display human qualities such as reasoning, planning, problem solving, grasping complicated ideas, learning from personal experiences, and so on by using artificial general intelligence. Many corporations and companies are working on developing general intelligence, but they have yet to finish it.

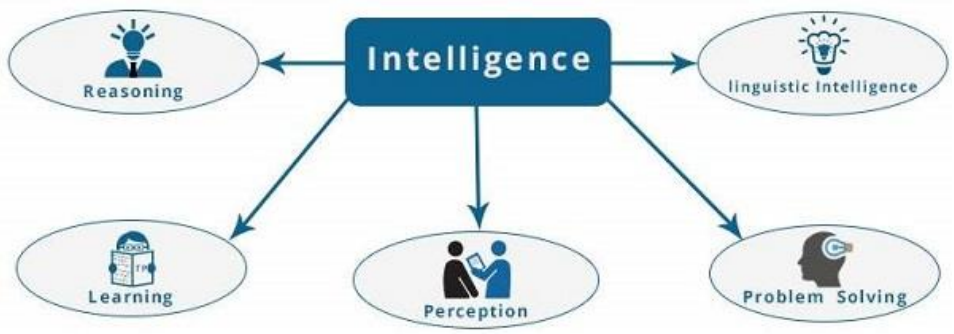
3) **Artificial Super-Intelligence :** AI thinker Nick Bostrom defined "Super intelligence is an intellect that is much smarter than the best human brains in practically every field, including scientific creativity, general wisdom and social skills." Super intelligence ranges from a machine which is just a little smarter than a human to a machine that is trillion times smarter. Artificial super intelligence is the ultimate power of AI.

Weak AI	**Strong AI**
It is a narrow application with a limited scope.	It is a wider application with a more vast scope.
This application is good at specific tasks.	This application has an incredible human-level intelligence.
It uses supervised and unsupervised learning to process data.	It uses clustering and association to process data.
Example Siri, Alexa	Example Advanced Robotics

1.2.6 Components of AI

The intelligence is intangible. It is composed of −

- Reasoning
- Learning
- Problem Solving
- Perception
- Linguistic Intelligence



Let us go through all the components briefly −

- Reasoning − It is the set of processes that enables us to provide basis for judgement, making decisions, and prediction. There are broadly two types −

Inductive Reasoning	Deductive Reasoning
It conducts specific observations to makes broad general statements.	It starts with a general statement and examines the possibilities to reach a specific, logical conclusion.
Even if all of the premises are true in a statement, inductive reasoning allows for the conclusion to be false.	If something is true of a class of things in general, it is also true for all members of that class.
Example − "Nita is a teacher. Nita is studious. Therefore, All teachers are studious."	Example − "All women of age above 60 years are grandmothers. Shalini is 65 years. Therefore, Shalini is a grandmother."

- Learning − It is the activity of gaining knowledge or skill by studying, practising, being taught, or experiencing something. Learning enhances the awareness of the subjects of the study.

  The ability of learning is possessed by humans, some animals, and AI-enabled systems. Learning is categorized as −

  - Auditory Learning − It is learning by listening and hearing. For example, students listening to recorded audio lectures.
  - Episodic Learning − To learn by remembering sequences of events that one has witnessed or experienced. This is linear and orderly.
  - Motor Learning − It is learning by precise movement of muscles. For example, picking objects, Writing, etc.
  - Observational Learning − To learn by watching and imitating others. For example, child tries to learn by mimicking her parent.
  - Perceptual Learning − It is learning to recognize stimuli that one has seen before. For example, identifying and classifying objects and situations.
  - Relational Learning − It involves learning to differentiate among various stimuli on the basis of relational properties, rather than absolute properties. For Example, Adding ‘little less’ salt at the time of cooking potatoes that came up salty last time, when cooked with adding say a tablespoon of salt.
  - Spatial Learning − It is learning through visual stimuli such as images, colors, maps, etc. For Example, A person can create roadmap in mind before actually following the road.

- Stimulus-Response Learning − It is learning to perform a particular behaviour when a certain stimulus is present. For example, a dog raises its ear on hearing doorbell.

- Problem Solving − It is the process in which one perceives and tries to arrive at a desired solution from a present situation by taking some path, which is blocked by known or unknown hurdles.

  Problem solving also includes decision making, which is the process of selecting the best suitable alternative out of multiple alternatives to reach the desired goal are available.

- Perception − It is the process of acquiring, interpreting, selecting, and organizing sensory information.

  Perception presumes sensing. In humans, perception is aided by sensory organs. In the domain of AI, perception mechanism puts the data acquired by the sensors together in a meaningful manner.

- Linguistic Intelligence − It is one's ability to use, comprehend, speak, and write the verbal and written language. It is important in interpersonal communication.

1.2.7 Computational Intelligence (CI) Vs Artificial Intelligence (AI)

Computational Intelligence (CI)	Artificial Intelligence (AI)
Computational Intelligence is the study of the design of intelligent agents.	Artificial Intelligence is the study of making machines which can do things which at presents human do better.
Involvement of numbers and computations.	Involvement of designs and symbolic knowledge representations.
CI constructs the system starting from the bottom level computations, hence follows bottom-up approach.	AI analyses the overall structure of an intelligent system by following top down approach.
CI concentrates on low level cognitive function implementation.	AI concentrates on high level cognitive structure design.

1.3 HISTORY OF ARTIFICIAL INTELLIGENE

John McCarthy in 1955 introduced the term Artificial Intelligence.

The early work of Artificial Intelligence was done in the period 1943 to 1955. The first AI thoughts were formally put by McCulloch & Walter Pitts in the year 1943. They introduced with the concept of AI was based on different three theories. First theory is based on phycology i.e. Neuron functions in the brain. Second theory is based on formal analysis of

propositional logic and third theory is based on Turing's theory of computations.

1956-61

The first year of this period gave rise to the terminology 'Artificial Intelligence' proposed by McCarthy & supposed by the participants in the conference. In the same year Samuel developed a program for chess playing which performed better than its creator.

Around 1956-57, Chomsky's grammar in NLP i.e. linguistic model processing was a remarkable event. In 1958, McCarthy made a very significant contribution, development of LISP, an AI programming language and advice taker which combined the method of knowledge representation and resoning. Herbert Gelerriter at IBM in 1959 designed the first written AI program for geometry theorem proving in quick succession of time. In 1960, Window alone & then with Hoff developed networks called 'Adaline', based on the concepts of Hebbian learning. In 1956-57, logic theorist (LT), a program for automatic theorem proving was developed.

1962-67

At the beginning of this period Frank Rosen blatt proposed the concept of 'perception' in the line of Window's concept for artificial neural networks (ANN), a biological model to incorporate computational rationality. In 1963. McCarthy developed a general purpose logical reasoning method and it was enhanced by the Robinson's 'Resolution principle' (Robinson, 1965). The logical neural model of McCulloch and Pitts was enhanced by Winograd & Cowan in 1963. James Slage's program was developed for the interpretation of calculus in 1963. In 1965, Hearsay was developed at CMU for natural language interpretation of subset language.

1968-73

In this period, some AI program for practical use were developed. In 1967, David Bobrow developed 'STUDENT' to solve algebra story problems. The first knowledge-based expert system DENDRAL was developed by J. Lederber, Edward Feigenbaum and Carl Djerassi in 1968, although the work had started in 1965. The program discovered the molecular structure of an organic compound based on the mass spectral data. Simon stated that within 10 years a computer would be chess champion, & a significant mathematical theorem would be proved by machine. These predictions came true or approximately true within 40 years rather than 10.

The new back-propagation learning algorithms for multilayer networks that were to cause an enormous resurgence in neural-net research in the late 1980's were actually discovered first in 1969.

1974- 1980

In 1969 Minsky and Papert's book Perceptron's proved that perceptrons could represent vary little. Although their result did not apply to more

complex, multilayer networks, research funding for neural-net research soon dwindled to almost nothing.

In 1973 Professor S ir James Lightill mentioned the problem of combinatorial explosion or intractability which implied that many of AI's most successful algorithms would grind to a halt on real world problems and were only suitable for solving "toy" versions.

1981-1985

In this period many expert system shells, expert system tools and expert system programs were developed. During 1984-85 the expert system shells came into picture was EMYCIN by Buchanan, a rule-based diagnostic consultant based on LISP, EXPERT by Weiss and KAS by DUDE are the rule-based model for classification using FORTRAN; a semantic network-based system using LISP, others are knowledge crafts by GILMORE using object-oriented programming (OGP), KL-ONE by Brakeman using LISP for automatic inheritance. The most important development was PROLOG as AI programming language by Clockcin 1984.

1986-91

In this period significant developments occurred in ANN model in particular, the appearance of error back propagation algorithm formulated by Rumelhart and Hinton in parallel distributed processing. The probabilistic reasoning method in intelligent system appeared in 1988 by the work of Pearl. The distributed artificial intelligence concepts were formally incorporated in the multi-agent systems. The complete agent-based architecture was first implemented in a model SOAR, designed by Newel, Laired and Rosenbloom. Hidden Markov Model (HMM) was also conceptualized for speech processing and natural language processing during this period.

1992-97

In this period full swing of rise to the agent-based technology and multi-agent system (MAS). In 1992, Nawana brought the concept of autonomous agent, capable of acting independently with rationality. Different kinds of agent were defined. In 1994, Jennings Yoam introduced social and responsible agents, Yoam Shoham in 1993, described the concept of agent-oriented programming with different components and modalities. Belief, desire and intention (BDI) theory was introduced by Cohen (1995) during this period. The concept of cooperation, coordination and conflict resolution in MAS was introduced in this period.

In the NLP, a mean X-project was developed at Zurich by Nobel laureate Gerd Binning, who emphasized the use of word 'knowledge' to achieve comprehension. Gordon made a model language for representing strategies on standard AI planning techniques. The ABLE (Agent Building & Learning Environment) was developed by Joe Bigns, which focused on building hybrid intelligent agents for both reasoning and learning.

In introduction, incorporation and integration of AI concepts, theories and algorithms in and with web technology for information retrieval, extraction and categorization, document summarization, machine translation (single or multilingual) discourse analysis were performed in this period. Courteous logic program in which users specify the scope of potential conflict by pairwise mutual exclusion is implemented in common rules, a Java library used for e-commerce, business and web intelligence emerged as the front of AI. Formula Augmented Network (FAN) was developed by Morgenstern and Singh, is a knowledge structure, which enabled efficient reasoning and about potentially conflicting business rules.

Heuristic search methods were devised for game playing such as chess, checkers, Rubik cube with the concept of MPC. Blue Deep was a chess-playing computer developed by IBM on May 11, 1997. Robotics in game playing and surgery marks the splendid achievements in AI based robotics. The Seoul Robotic Football Game from 2001 onwards regularly updated is a good example of the development and incorporation of AI search methodology in game playing.

2004- Future directions and dimensions

Communications of the ACM 2003, a certain direction and dimension of AI have been envisaged for the future. Shannon has given the frame qualification and ramification and learning from books, i.e. reading the text and extracting relevant information. Three-dimensional robot surrounding in distilling from the www, a huge knowledge base, the developed of semantic scrapes and computational engines such as human mind and brain. Knowledge discovery and vision system for biometric and automated object regulated in supermarkets are imported milestones. Intelligent interface design should be intuitive for the novice, efficient perception for expert and robust undermines which would facilitate recovery from cognitive and manipulative mistakes. Programs that are helpful for diagnosis of errors and suggestions for corrective actions are to be developed.

1.3.1 Applications of AI

Artificial Intelligent Systems

1. **Medical :** AI has applications in cardiology (CRG), neurology (MRI), Embryology (Sonography), and difficult internal organ procedures, among other fields.

2. **Education :** Training simulators can be built using artificial intelligence techniques. Software for pre-school children are developed to enable learning with fun games. Automated grading, Interactive tutoring, instructional theory are the current areas of application.

3. **Military :** When decisions have to be made quickly taking into account an enormous amount of information, and when lives are at stake, artificial intelligence can provide crucial assistance.

   Training simulators can be used in military applications for the purpose of difficult task which human can not do easily, Robots are also used in many situations. AI plays important role in modern military.

4. **Entertainment :** Playing different AI based games, where one side human and other side the player (machine) which works on AI technology. Many film industries use Robots to play a role for critical situations like fire, jump etc.

5. **Business and Manufacturing:** Robots are well equipped with the various task in business and manufacturing. Vehicle workshops Robots are useful for jack purpose, car painting etc.

1.3.2 Domains or sub areas of AI

AI applications can be roughly classified based on the type of tools used for inoculating intelligence in the system. Various sub domains and areas in intelligent systems can be given as follows:

Expert Systems

Natural Language Processing

Neural Networks

Robotics

Fuzzy Logic

Sr.No.	Research Areas	Example
1	Expert Systems Examples − Flight-tracking systems, Clinical systems.	
2	Natural Language Processing Examples: Google Now feature, speech recognition, Automatic voice output.	

Sr.No.	Research Areas	Example
3	Neural Networks Examples − Pattern recognition systems such as face recognition, character recognition, handwriting recognition.	
4	Robotics Examples − Industrial robots for moving, spraying, painting, precision checking, drilling, cleaning, coating, carving, etc.	
5	Fuzzy Logic Systems Examples − Consumer electronics, automobiles, etc.	

1.4 STATE OF ART OF AI

Artificial Intelligence has infiltrated every part of our daily lives. Everywhere, from washing machines to air conditioners to smart phones. AI is assisting us in making our lives easier. AI is also doing fantastic things in industries. In factories, sound work is done by robots. Self-driving cars are now a reality. Barbie, who is WiFi-enabled, uses speech recognition to converse with and listen to children. AI is being used by businesses to improve their products and increase sales. Machine learning has made considerable progress in AI.

Areas in which AI is showing significant advancements as follows:

1. Deep Learning
2. Machine Learning
3. AI replacing Workers
4. Internet of Things (IoT)
5. Emotional AI
6. AI in shopping and customer service
7. Ethical AI

1. **Deep Learning :** Deep learning has been successfully used to a variety of text analysis and understanding challenges in recent years. Document categorization, sentiment analysis, machine translation, and other similar techniques are used, and the results are frequently dramatic.

   Top Applications of Deep Learning Across Industries

   - Self Driving Cars.
   - News Aggregation and Fraud News Detection.
   - Natural Language Processing.
   - Virtual Assistants.
   - Entertainment.
   - Visual Recognition.
   - Fraud Detection.
   - Healthcare.

2. **Machine Learning :** Machine Learning is an artificial intelligence application in which a computer/machine learns from past experiences (input data) and predicts the future. The system's performance should be at least human-level. The system learns from the data set provided in order to complete task T.

   Top 10 real-life examples of Machine Learning

   - Image Recognition. Image recognition is one of the most common uses of machine learning.
   - Speech Recognition. Speech recognition is the translation of spoken words into the text.
   - Medical diagnosis.
   - Statistical Arbitrage.
   - Learning associations.
   - Classification.
   - Prediction.
   - Extraction.

3. **AI Replacing Workers :** Machines are already better than humans in physical jobs; they can move quicker, more precisely, and lift heavier loads. There will be almost nothing these machines can't accomplished or learn to do quickly. Once they are as sophisticated as we are.

4. **Internet of Things (IoT) :** AI-assisted The Internet of Things (IoT) develops intelligent machines that mimic smart behaviour and assist in decision-making with little or no human intervention. While IoT is concerned with devices connecting with each other over the internet, AI is concerned with devices learning from their data and experience.

5. **Emotional AI :** Emotion AI, often known as affective computing, is all about utilising artificial intelligence to identify emotions. Machines with this level of emotional intelligence can comprehend both cognitive and emotional channels of human communication.

6. **AI in shopping and customer service :** Voice detection technology powered by AI may enable customers to converse with digital assistants in order to get the most out of the products they purchase. Most consumers will benefit greatly from this virtual link.

   Artificial intelligence can help retailers decrease operational costs by automating in-store processes. It can assist customers in the store without the use of salespeople, reduce lines with cashier-less payments, refill stock with real-time stock monitoring, and digitise store displays and trial rooms.

7. **Ethical AI :** The notion of building artificially intelligent systems utilising norms of behaviour that ensure an automated system can respond to situations in an ethical manner is known as Roboethics, or robot ethics. To do so, we turn to machine ethics, which is concerned with the process of imbuing AI robots with moral characteristics.

1.5 PROBLEM SOLVING WITH ARTIFICIAL INTELLIGENCE

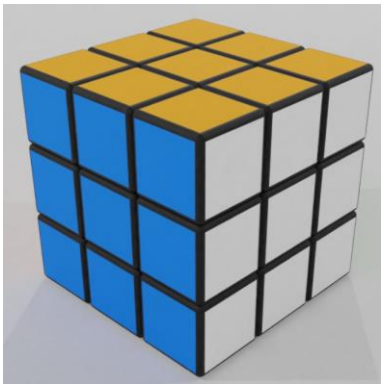
1.5.1 Problems

To identify desirable answers, problem-solving relates to artificial intelligence techniques such as building efficient algorithms, heuristics, and doing root cause analysis. The problem-solving agent can decide what to do by reviewing various possible sequences of actions that lead to states of known value and then selecting the best sequence. Search is the term for the process of looking for such a sequence.

1.5.1.1 Classic examples of Artificial Intelligence Search Problems

1. 3*3*3 Rubik’s cube problem
2. 8/15/24 -puzzle problem
3. N-queen problem
4. Water Jug problem

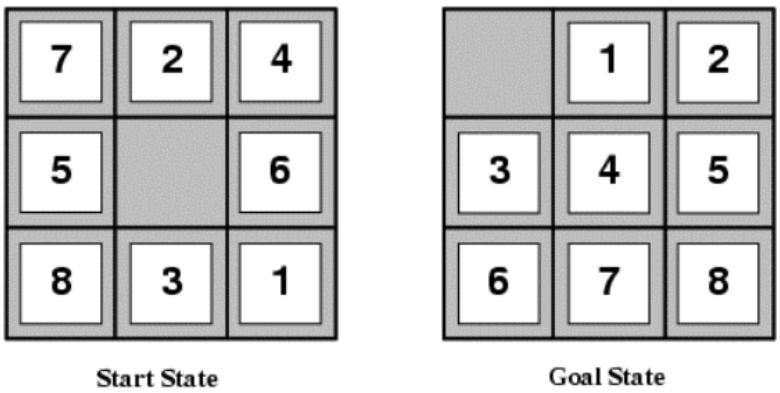
1. **3*3*3 Rubik’s cube problem :** A Rubik's cube is a three-dimensional puzzle with six faces, each of which has nine stickers in a three-dimensional (3x3) pattern. The goal of the puzzle is to solve it such that each face has just one colour.



Rubic's cube Problem

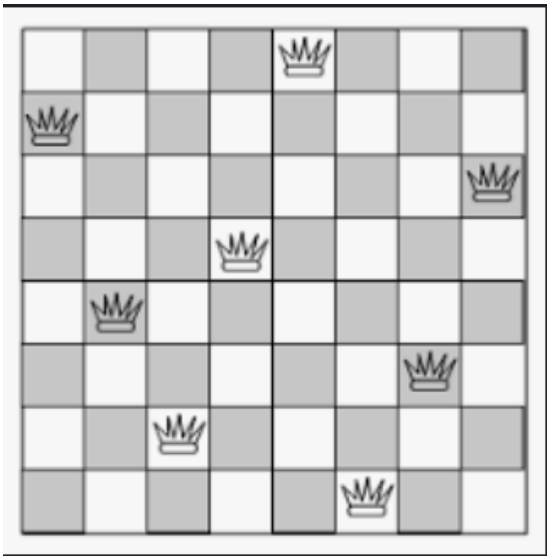
2. **8/15/24 -Puzzle Problem :** The 8 puzzle problem by the name of N puzzle problem or sliding puzzle problem. N-puzzle that consists of N tiles (N+1 titles with an empty tile) where N can be 8, 15, 24 and so on. In our example N = 8. (that is square root of (8+1) = 3 rows and 3 columns).

   The 8-puzzle is a sliding puzzle that is played on a 3-by-3 grid with 8 square tiles labelled from 1 through 8, plus a blank square. The goal is to rearrange the tiles so that they are in row-major order, using as few moves as possible. You are permitted to slide tiles either horizontally or vertically into the blank square.



8 – Puzzle Problem

3. **N – Queen Problems :** In N-Queen, the queens need to be placed on the n*n board, in such a way that no queen can dash the other queen, in any direction i.e. horizontally, vertically as well as diagonally.



N – Queen Problem

N=8

4. **Water Jug Problem :** In the Artificial Intelligence water jug problem, we are given two jugs, one of which can contain 3 gallons of water and the other of which can store 4 gallons of water. There is no additional measuring equipment accessible, and the jugs themselves are not marked in any way. The agent's job is to fill the 4-gallon jug with 2 gallons of water using only these two jugs and no additional materials. Both of our jugs are initially empty.

SUMMARY

This chapter gives the details about Artificial Intelligence, History of Artificial Intelligence with its applications, state of art of AI, Various problem related to Artificial Intelligence, Applications, Domains or sub Areas in which AI is showing significant advancements.

QUESTIONS

Q.1) What is Artificial Intelligence?

Q.2) What are the components of Artificial Intelligence?

Q.3) Explain various applications of Artificial Intelligence.

Q.4) Define Artificial Intelligence

Q.5) Discuss examples of problem solving with AI.

Q.6) Give the difference between Computational Intelligence (CI) and Artificial Intelligence (AI).

TEXT BOOK

1. Artificial Intelligence A Modern Approach, Third Edition, Stuart Russell and Peter Norvig, Pearson Education

2. Elaine Rich, Kevin Knight, & Shivashankar B Nair, Artificial Intelligence, McGraw Hill, 3rd ed.,2009

REFERENCES

1) Introduction to Artificial Intelligence & Expert Systems, Dan W Patterson, PHI.,2010

2) S Kaushik, Artificial Intelligence, Cengage Learning, 1st ed.2011

3) Artificial Intelligence, 3rd Edn., E. Rich and K. Knight (TMH)

4) Artificial Intelligence, 3rd Edn., Patrick Henny Winston, Pearson Education.

❄❄❄❄❄❄❄

2

INTELLIGENT AGENT

Unit Structure :

2.0 OBJECTIVES

In this chapter we are going to learn agent, intelligent agent, actuators, agent function, what is rationality, rational agent, structure of intelligent agent, various natures of environments, PEAS properties of agent as well as different types of agents.

2.1 INTRODUCTION

Agent is something that perceives its environment or surroundings with the help of sensors and act upon that environment with the help of actuators.

2.1.1 What Is Agent?

An agent is anything that can be thought of as sensing its surroundings through sensors and acting on them through actuators.

2.1.2 Actuators

A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators. A robotic agent might have cameras and

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

We use the term percept to refer to the agent's perceptual inputs at any given instant. An agent's percept sequence is the complete history of everything the agent has ever perceived. In general, an agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived. By specifying the agent's choice of action for every possible percept sequence,

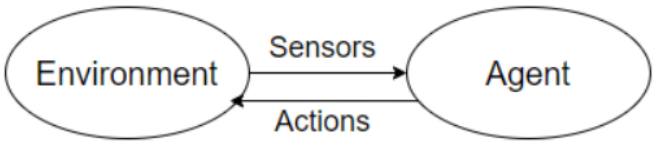


Fig: Environment and Agent

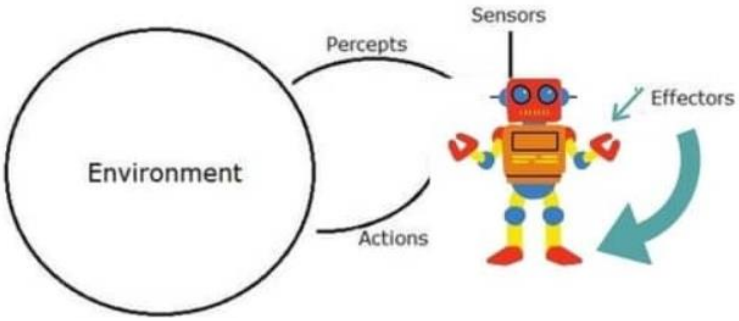


Fig: Generic Robotic Agent Architecture

As sensors, the robotic agent uses cameras, infrared range finders, scanners, and other devices, while actuators include various types of motors, screens, printing devices, and other devices.

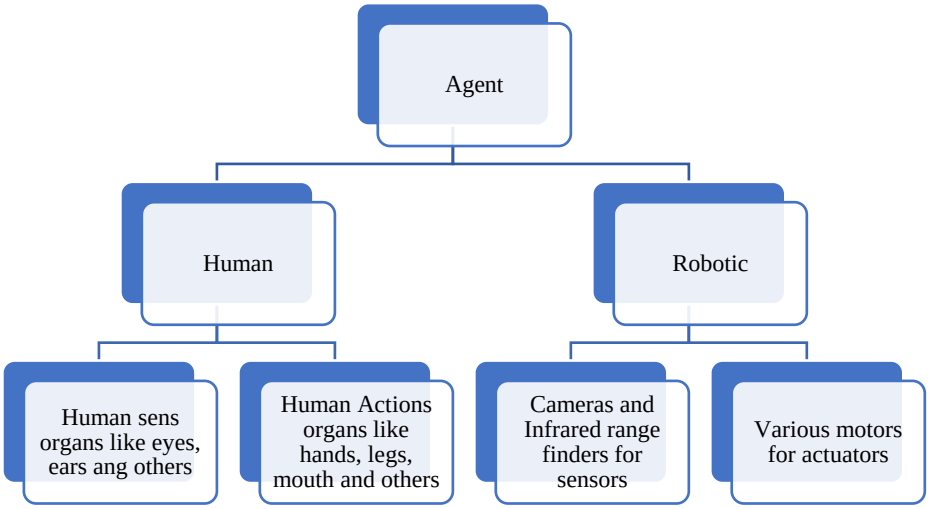


Fig : Sensors and Actuators in Human & Robotic Agent

2.2 AGENT FUNCTION

The agent function is the description of what all functionalities the agent is supposed to do. The agent function provides mapping between percept sequences to the desired actions.

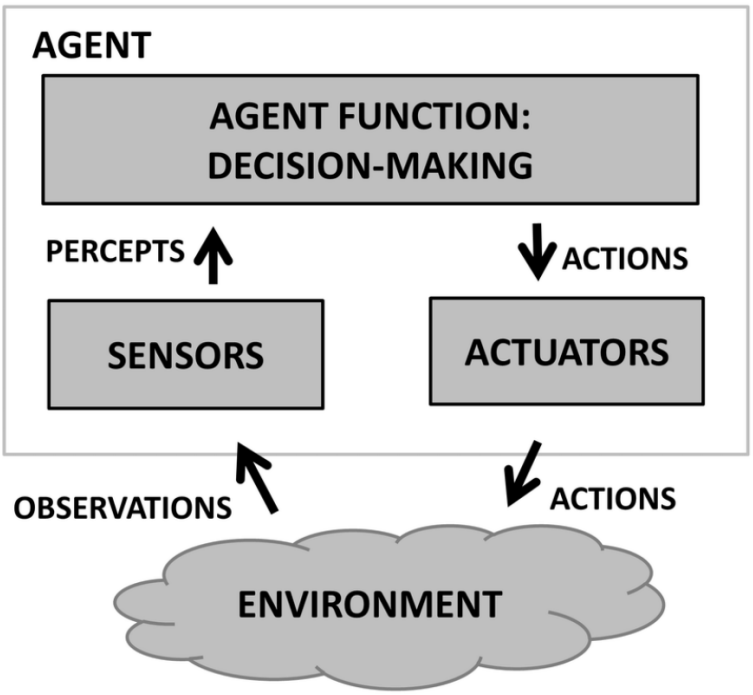


Figure : Agent Function

An agent is anything that can perceive its environment through sensors and acts upon that environment through effectors.

- A human agent has sensory organs such as eyes, ears, nose, tongue and skin parallel to the sensors, and other organs such as hands, legs, mouth, for effectors.
- A robotic agent replaces cameras and infrared range finders for the sensors, and various motors and actuators for effectors.

Agent Terminology

- Performance Measure of Agent − It is the criteria, which determines how successful an agent is.
- Behavior of Agent − It is the action that agent performs after any given sequence of percepts.
- Percept − It is agent's perceptual inputs at a given instance.
- Percept Sequence − It is the history of all that an agent has perceived till date.
- Agent Function − It is a map from the precept sequence to an action.

2.3 RATIONALITY

Rationality is nothing but status of being reasonable, sensible, and having good sense of judgment.

Rationality is concerned with expected actions and results depending upon what the agent has perceived. Performing actions with the aim of obtaining useful information is an important part of rationality.

What is Ideal Rational Agent?

An ideal rational agent is the one, which is capable of doing expected actions to maximize its performance measure, on the basis of −

- Its percept sequence
- Its built-in knowledge base

Rationality of an agent depends on the following four factors −

- The performance measures, which determine the degree of success.
- Agent's Percept Sequence till now.
- The agent's prior knowledge about the environment.
- The actions that the agent can carry out.

2.3.1 Rational Agent

A rational agent always performs right action, where the right action means the action that causes the agent to be most successful in the given percept sequence. The problem the agent solves is characterized by Performance Measure, Environment, Actuators, and Sensors (PEAS).

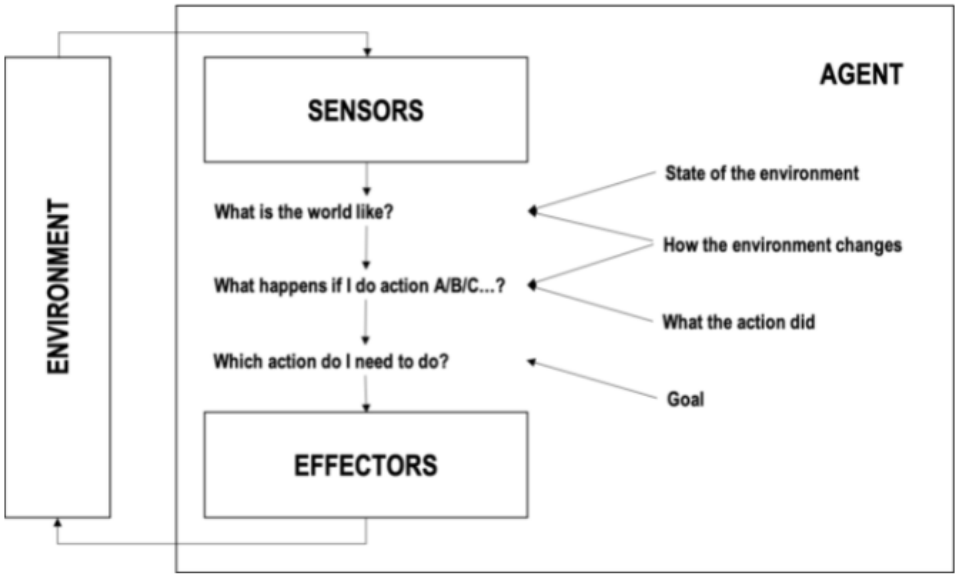


Figure : Rational Agent Work

2.4 INTELLIGENT AGENT

An intelligent agent is anything that perceives the environment through it's sensors and acts upon it through it's actuators (effectors). The actions of the agent are always directed towards the goal.

2.4.1 Structure of Intelligent Agents

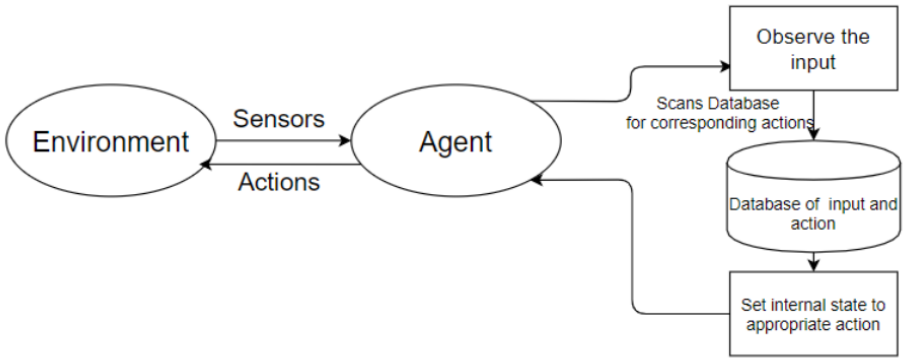


Figure : Intelligent Agent

The few types of agents are

- Human Agent : Sensors : Nose, Ears, Eyes, Tongue, Skin.
- Actuators : Hands, Legs, Mouth.
- Robotic Agent :
- Sensors : Camera, Infrared range finders.
- Actuators : Motors.
- Software agent : They have encoded bit strings as sensors and actuators.

Agent's structure can be viewed as −

- Agent = Architecture + Agent Program
- Architecture = the machinery that an agent executes on.
- Agent Program = an implementation of an agent function.

2.5 TYPES OF AGENTS

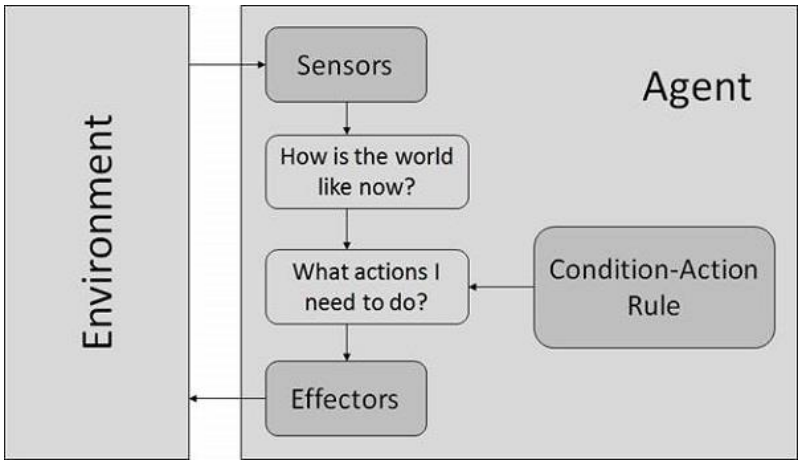
TYPES OF AGENTS

1. SIMPLE REFLEX AGENTS
2. MODEL-BASED RELEX AGENTS
3. GOAL BASED AGENTS
4. UTILITY BASED AGENTS
5. LEARNING AGENTS

2.5.1 Simple Reflex Agents

- They choose actions only based on the current percept.
- They are rational only if a correct decision is made only on the basis of current precept.
- Their environment is completely observable.

Condition-Action Rule − It is a rule that maps a state (condition) to an action.



2.5.2 Model Based Reflex Agents

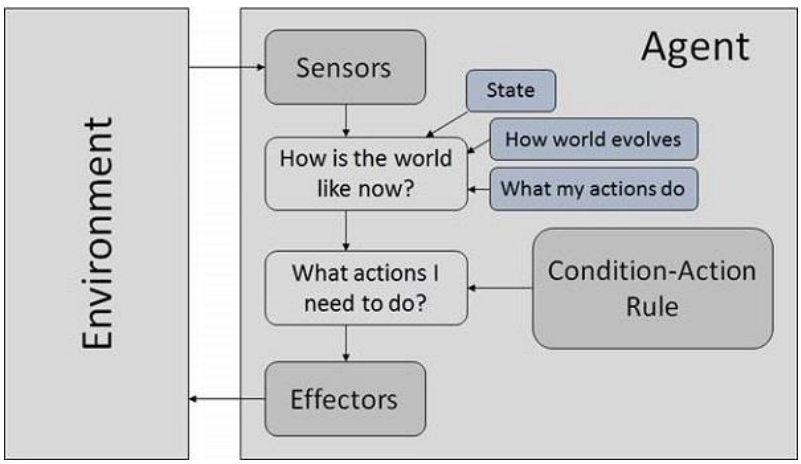
They use a model of the world to choose their actions. They maintain an internal state.

Model − The knowledge about "how the things happen in the world".

Internal State − It is a representation of unobserved aspects of current state depending on percept history.

Updating the state requires the information about −

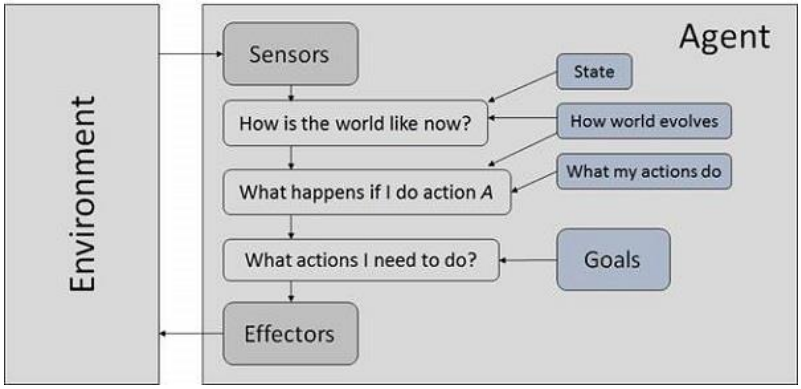
- How the world evolves.
- How the agent's actions affect the world.



2.5.3 Goal Based Agents

They choose their actions in order to achieve goals. Goal-based approach is more flexible than reflex agent since the knowledge supporting a decision is explicitly modeled, thereby allowing for modifications.

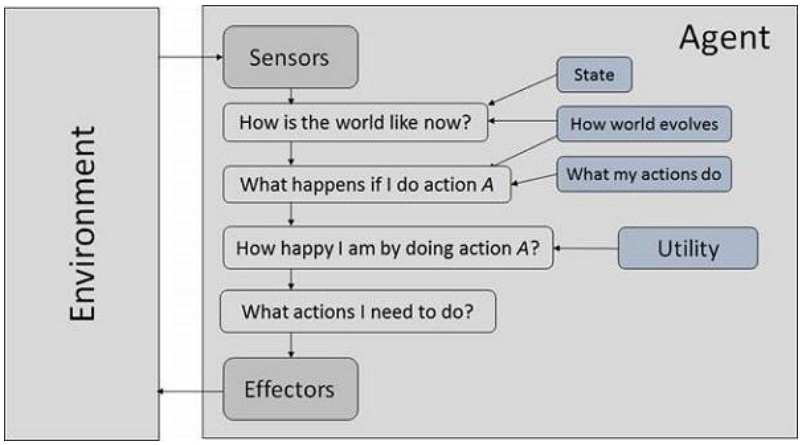
Goal − It is the description of desirable situations.



2.5.4 Utility Based Agents

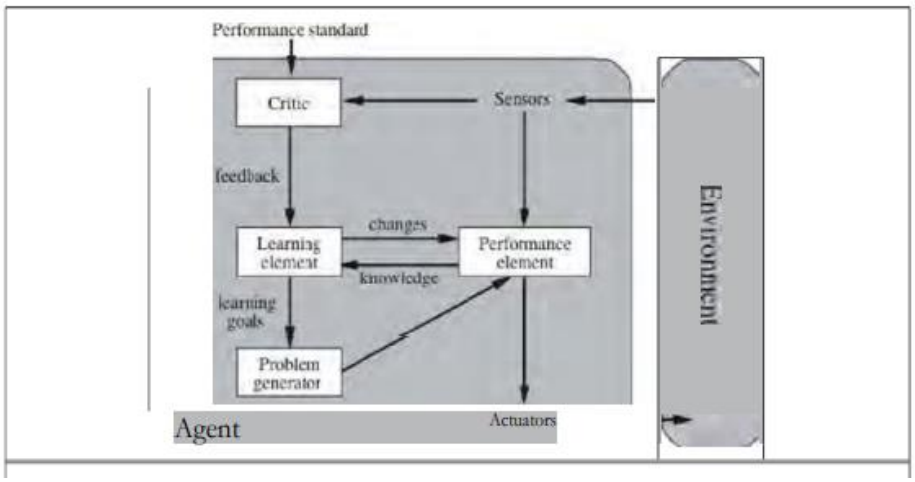
They choose actions based on a preference (utility) for each state. Goals are inadequate when −

- There are conflicting goals, out of which only few can be achieved.
- Goals have some uncertainty of being achieved and you need to weigh likelihood of success against the importance of a goal.



2.5.5 Learning Agents

We've gone over different approaches for selecting actions in agent programmes. So far, we haven't detailed how the agent programmes are created. Turing (1950) examines the possibility of programming his intelligent machines by hand in his famous early article.



- It calculates the amount of time this will take and concludes, "Some more expeditious method appears desired." Building learning machines and then teaching them is the way he advocates. This is now the favoured strategy for developing cutting-edge AI systems in many fields. Another advantage of learning is that, as previously said, it allows the agent to operate in previously unknown contexts and to grow more proficient than its starting understanding would allow. The core concepts of learning agents are briefly introduced in this section. Throughout the book, we discuss learning possibilities and strategies for various types of agents. Part V delves deeper into the learning algorithm.

- As indicated in Figure, a learning agent can be separated into four conceptual components. The most crucial contrast is between the learning element, which is in charge of improving, and the performance element, which is in charge of deciding on external activities. The performance element is what we previously thought of as the full agent: it processes information and makes decisions. The learning element takes the critic's comments on how the agent is performing and decides how the performance aspect should be tweaked in order for the agent to perform better in the future.

- The learning element's design is heavily influenced by the performance element's design. When trying to create an agent that learns a specific capacity, the first thing to ask is "What type of performance factor will my agent need to do this once it has learned how?" rather than "How am I going to get it to teamthis?" Learning methods can be built to improve any aspect of an agent given its design.

- The critic informs the learning aspect of the agent's performance against a predetermined benchmark. Because the percepts do not provide any evidence of the agent's success, the critic is required. A chess programme, for example, might receive a percept indicating that it has checkmated its opponent, but it would need a performance standard to know that this is a good thing because the percept does not state so.

- The learning agent's final component is the problem generator. It is in charge of recommending measures that will lead to new and educational experiences. The idea is that, given what it knows, if the performance factor had its way, it would continue to do the best activities. However, if the agent is prepared to experiment a little and take some potentially poor behaviours in the short term, it may uncover much better long-term activities. It is the role of the problem generator to suggest these exploratory actions. When scientists conduct experiments, they do them in this manner. Galileo did not consider dropping rocks from the top of a Pisa tower to be valuable in and of itself. He wasn't attempting to breach the law.

2.6 NATURE OF ENVIRONMENTS

Some programs operate in the entirely artificial environment confined to keyboard input, database, computer file systems and character output on a screen.

In contrast, some software agents (software robots or softbots) exist in rich, unlimited softbots domains. The simulator has a very detailed, complex environment. The software agent needs to choose from a long array of actions in real time. A softbot designed to scan the online preferences of the customer and show interesting items to the customer works in the real as well as an artificial environment.

The most famous artificial environment is the Turing Test environment, in which one real and other artificial agents are tested on equal ground. This is a very challenging environment as it is highly difficult for a software agent to perform as well as a human.

Turing Test

The success of an intelligent behavior of a system can be measured with Turing Test.

Two persons and a machine to be evaluated participate in the test. Out of the two persons, one plays the role of the tester. Each of them sits in different rooms. The tester is unaware of who is machine and who is a human. He interrogates the questions by typing and sending them to both intelligences, to which he receives typed responses.

This test aims at fooling the tester. If the tester fails to determine machine’s response from the human response, then the machine is said to be intelligent.

Properties of Environment

2.6.1 Natures of Environment

The environment has multifold properties −

- Discrete / Continuous − If there are a limited number of distinct, clearly defined, states of the environment, the environment is discrete

(For example, chess); otherwise it is continuous (For example, driving).

- Observable / Partially Observable − If it is possible to determine the complete state of the environment at each time point from the percepts it is observable; otherwise it is only partially observable.
- Static / Dynamic − If the environment does not change while an agent is acting, then it is static; otherwise it is dynamic.
- Single agent / Multiple agents − The environment may contain other agents which may be of the same or different kind as that of the agent.
- Accessible / Inaccessible − If the agent's sensory apparatus can have access to the complete state of the environment, then the environment is accessible to that agent.
- Deterministic / Non-deterministic − If the next state of the environment is completely determined by the current state and the actions of the agent, then the environment is deterministic; otherwise it is non-deterministic.
- Episodic / Non-episodic − In an episodic environment, each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself. Subsequent episodes do not depend on the actions in the previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

2.6.2 PEAS Properties of Agent

PEAS: Performance Measure, Environment, Actuators, & Sensors.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers

Agent Type	Performance Measure	Environment	Actuators	Sensors
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts: bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, beaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises. suggestions, corrections	Keyboard entry

Examples of agent types and their PEAS descriptions

SUMMARY

This chapter gives the details about Artificial Intelligence agent and how it works, agent types, use of agent function, structure of intelligent agents, nature of environments and PEAS properties, various types of agent.

QUESTIONS

Q.1) What Is Intelligent Agent?

Q.2) Define Agent

Q.3) Explain Different Types of Agents.

Q.4) Explain Peas

Q.5) Discuss Different Natures of Environments in Detail.

❄❄❄❄❄❄❄

3

PROBLEM SOLVING BY SEARCHING

Unit Structure :

3.0 OBJECTIVES

In this chapter we are going to learn about:

- what is problem, problem solving methods, problem solving agents. Study of Various searching techniques and types, comparison of searching techniques, Heuristic function etc.

AI and related fields

- **Logical AI :** What a program knows about the world in general the facts of the specific situation in which it must act, and its goals are all represented by sentences of some mathematical logical language. The program decides what to do by inferring that certain actions are appropriate for achieving its goals.

- **Search:** AI programs often examine large numbers of possibilities, e.g. moves in a chess game or inferences by a theorem proving program. Discoveries are continually made about how to do this more efficiently in various domains.

- **Pattern Recognition :** When a program makes observations of some kind, it is often programmed to compare what it sees with a pattern. For example, a vision program may try to match a pattern of eyes and a nose in a scene in order to find a face. More complex patterns, e.g. in a natural language text, in a chess position, or in the history of some event are also studied.

- **Representation :** Facts about the world have to be represented in some way. Usually, languages of mathematical logic are used.

- **Inference :** From some facts, others can be inferred. Mathematical logical deduction is adequate for some purposes, but new methods of non-monotonic inference have been added to logic since the 1970s. The simplest kind of non-monotonic reasoning is default reasoning in which a conclusion is to be inferred by default, but the conclusion can be withdrawn if there is evidence to the contrary. For example, when we hear of a bird, we man infer that it can fly, but this conclusion can be reversed when we hear that it is a penguin. It is the possibility that a conclusion may have to be withdrawn that constitutes the non-monotonic character of the reasoning. Ordinary logical reasoning is

monotonic in that the set of conclusions that can the drawn from a set of premises is a monotonic increasing function of the premises.

- **Common sense knowledge and reasoning :** This is the area in which AI is farthest from human-level, in spite of the fact that it has been an active research area since the 1950s. While there has been considerable progress, e.g. in developing systems of non-monotonic reasoning and theories of action, yet more new ideas are needed.

- **Learning from experience :** Programs do that. The approaches to AI based on connectionism and neural nets specialize in that. There is also learning of laws expressed in logic. Programs can only learn what facts or behaviors their formalisms can represent, and unfortunately learning systems are almost all based on very limited abilities to represent information.

- **Planning :** Planning programs start with general facts about the world (especially facts about the effects of actions), facts about the particular situation and a statement of a goal. From these, they generate a strategy for achieving the goal. In the most common cases, the strategy is just a sequence of actions.

- **Epistemology :** This is a study of the kinds of knowledge that are required for solving problems in the world.

- **Ontology :** Ontology is the study of the kinds of things that exist. In AI, the programs and sentences deal with various kinds of objects, and we study what these kinds are and what their basic properties are. Emphasis on ontology begins in the 1990s.

- **Heuristics :** A heuristic is a way of trying to discover something or an idea imbedded in a program. The term is used variously in AI. Heuristic functions are used in some approaches to search to measure how far a node in a search tree seems to be from a goal. Heuristic predicates that compare two nodes in a search tree to see if one is better than the other, i.e. constitutes an advance toward the goal, may be more useful.

- **Genetic Programming :** Genetic programming is a technique for getting programs to solve a task by mating random Lisp programs and selecting fittest in millions of generations.

3.1 INTRODUCTION (PROBLEM SOLVING)

Search and Control Strategies

Problem Solving

Problem solving is an important aspect of Artificial Intelligence. A problem can be considered to consist of a goal and a set of actions that can be taken to lead to the goal. At any given time, we consider the state of the search space to represent where we have reached as a result of the actions we have applied so far. For example, consider the problem of looking for a contact lens on a football field. The initial state is how we start out, which is to say

we know that the lens is somewhere on the field, but we don't know where. If we use the representation where we examine the field in units of one square foot, then our first action might be to examine the square in the top-left corner of the field. If we do not find the lens there, we could consider the state now to be that we have examined the top-left square and have not found the lens. After a number of actions, the state might be that we have examined 500 squares, and we have now just found the lens in the last square we examined. This is a goal state because it satisfies the goal that we had of finding a contact lens.

3.1.1 Search

Search is a method that can be used by computers to examine a problem space like this in order to find a goal. Often, we want to find the goal as quickly as possible or without using too many resources. A problem space can also be considered to be a search space because in order to solve the problem, we will search the space for a goal state. We will continue to use the term search space to describe this concept. In this chapter, we will look at a number of methods for examining a search space. These methods are called search methods.

3.1.2 Importance of Search in AI

- It has already become clear that many of the tasks underlying AI can be phrased in terms of a search for the solution to the problem at hand.
- Many goal based agents are essentially problem solving agents which must decide what to do by searching for a sequence of actions that lead to their solutions.
- For production systems, we have seen the need to search for a sequence of rule applications that lead to the required fact or action.
- For neural network systems, we need to search for the set of connection weights that will result in the required input to output mapping.

3.1.3 Problem Solving Agent

Problem formulation is the basic step of problem solving agent where problems defined by using five components.

3.1.3.1 Steps in Problem Solving

Problem can be defined formally using five components as follows:

1. Initial state
2. Actions
3. Successor function
4. Goal test
5. Path cost

1. **Initial state :** The initial state is the one in which the agent starts in.
2. **Actions:** It is the set of actions that can be executed or applicable in all possible states. A description of what each action does; the formal name for this is the transition model.
3. **Successor function :** It is a function that returns a state on executing an action on the current state.
4. **Goal test :** It's a test to see if the present state is a goal state or not. In some cases, a goal test can be performed simply by comparing the current state to the declared goal state, which is known as an explicit goal test. The implicit goal test is used when the state of a problem cannot be described directly and must be generated by doing some computations.

   Example: In Tic-Tac-Toe game making diagonal or vertical or horizontal combination declares the winning state which can be compared explicitly, but in case of chess game, the goal state cannot be predefined but it's a scenario called as "Checkmate" which has to be evaluated implicitly.
5. **Path cost :** It is simply the cost associated with each step to be taken to reach to the goal state. To determine the cost to reach to each state, there is a cost function, which is chosen by the problem solving agent.

3.1.3.2 Algorithm

Procedure or method : Problem solving agent, agent may be unknown, space, percept.

Result : An Action.

Input : P → percept (Environment perception)

Static :

1) A → Ab action sequence, initially with null value.
2) S → State – current state.
3) G → Goal – A goal initially null.
4) P → Problem – A real world situation.

State – update state (State, Percept)

If (s) is empty then do

g → Formulate goal/goals

P → Formulate problem (s,g)

S → Search (p)

G First (s)

First(s) → G

Rest(s) → S

Return a

Procedure

Which search algorithm one should use will generally depend on the problem domain? There are four important factors to consider:

- Completeness – Is a solution guaranteed to be found if at least one solution exists?
- Optimality – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?
- Time Complexity – The upper bound on the time required to find a solution, as a function of the complexity of the problem.
- Space Complexity – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem

3.1.3.3 Solutions to The Problem Solving

- **Problem Solution**

  A well -defined problem with specification of initial state, goal test, successor function, and path coast it can be represented as a data structure and used to implement a program which can search for a goal test. A solution to a problem is a sequence of action chosen by the problem solving agent that leads from the initial state to a goal state. Solution quality is measured by the path cost function.

- **Optimal Solution**

  An optimal solution is the solution with least path cost among all solutions. General sequence followed by a simple problem solving agent is, first it formulates the problem with the goal to be achieved, then it searches for a sequence of actions that would solve the problem, and then executes the actions one at a time.

3.1.3.3 EXAMPLE

Examples of Problem Solving

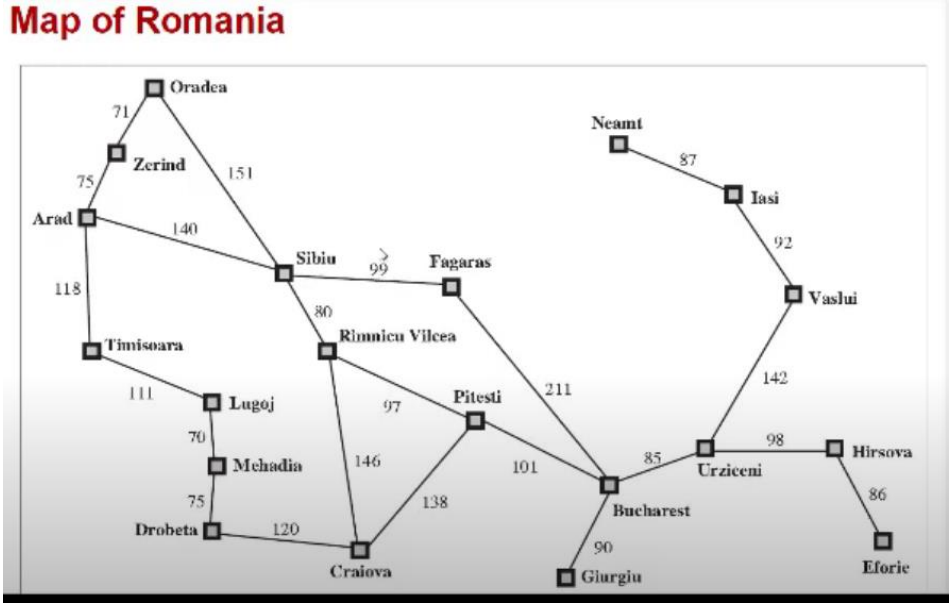
By using Solving Problems by Searching, there are five different components for defining a problem.

Components for defining a problem

1) Initial State
2) Actions Available
3) Transition Model
4) Path Cost
5) Goal Test

Example – 1) Romania Map Problem

Map of Romania : Romania is a country in Europe, following are the major cities.



Description of Map Problem

1) The initial state : The initial state for our agent in this case (Romania) might be described as In(Arad).

2) Actions : It gives the possible actions from the current state, in this case the possible actions are (Go(Sibiu), Go(Timisoara), Go(Zerind)}.

3) Transition Model : This is Specified by using a function RESULT(s,a), that returns the state that results from doing action a in state s.

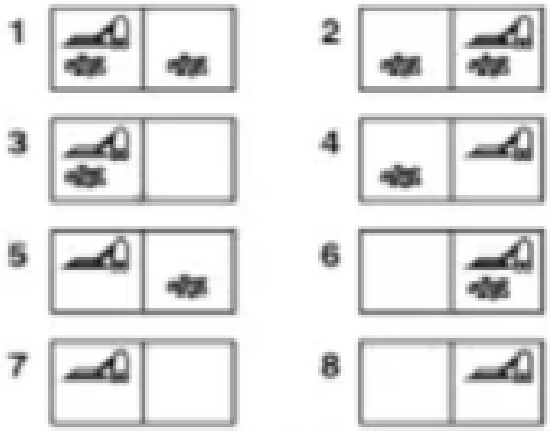
   RESULT(In(Arad),Go(Zerind)) = In(Zerind).

4) Path Cost : Path cost function assigns a numeric cost to each path, In the present case it is the distance in kilometers.

5) Goal Test : It determines whether the given state is Goal State.

Example – 2) Vacuum-Cleaner Problem

Vacuum World :



1) State : The state is determined by both the agent location and the dirt locations. There are 8 possible world states.

2) Initial state : Any state can be designated as the initial state.

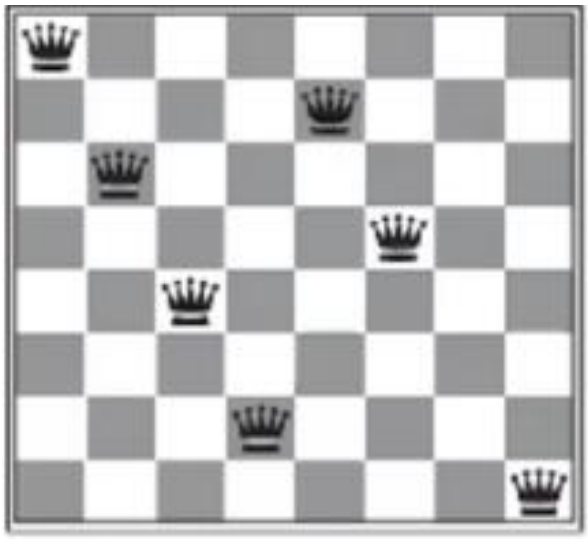
3) Actions : Each state has just three actions : 1. Left 2. Right 3. Suck.

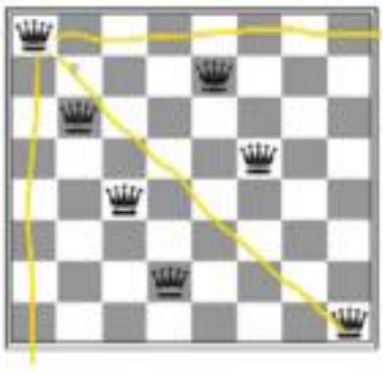
4) Transition model : Action left takes the agent to Leftmost square, Action Right takes the agent to Rightmost square and the Suck action cleans a dirty square and performs no action on clean square.

5) Goal test : This checks whether all the squares are clean.

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

Example – 3) 8 Queens Problem





1) States : Any arrangement of 0 to 8 queens on the boards is a state.

2) Initial state : Empty board i.e. No queens on the board.

3) Actions : Adding a queen to any empty square.

4) Transition model : Returns the board with a queen added to the specified square.

5) Goal test : 8 queen are on the board, in such a way that none is in attacking position.

Example – 4) Route-finding problem
It is same as Romania map problem

1) States : Each state includes a location, the current time, cost of an action (a flight segment)

2) Initial state : This is specified by the user (the starting point)

3) Actions : Take any flight from the current location, in any seat class, leaving after the current time.

4) Transition model : The state resulting from taking a flight will have the flight's destination as the current location and the destination time as the current time.

5) Goa test : Check if the Final destination has been reached

6) Path cost : This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane etc.

Example – 5) The Travelling Salesperson Problem (TSP)

1) States : All the cities which are to be visited by the salesman.

2) Initial State : Any city can be the initial state.

3) Actions : The agent can visit any city which is not yet visited from the current city.

4) Transition model : The next city becomes the current city

5) Goal test : Check if all the cities have been visited with minimum cost & also the final state is the initial state.

6) Path cost : This depends on the actions the agent has taken throughout the journey.

Measuring Problem Solving Performance

There are three possible outcomes for any problem.

1) We reach at failure state

2) We reach at solution state

3) Algorithm might get stuck in an infinite loop.

Problem solving performance can be evaluated with 4 various factors as follows:

1) Completeness

2) Optimality

3) Time complexity

4) Space complexity

There are two Basic Search Strategies

Figure : Types of search strategies

1) **Uninformed search :** It is also known as blind search. In this strategy only the initial and goal state is given and no additional information is available.

2) **Informed Search :** It is also known as Heuristic search. Some additional information apart from the initial and the goal state is given, through which we know which state out of the explored is more beneficial.

3. 2 UNINFORMED SEARCH STRATEGIES

3.2.1 Introduction (Uninformed Search)

- They do not have any additional information
- Information provided in the problem definition only
- To reach at goal state using different order or length of actions.
- It does not use knowledge in the processing of search.
- It is time consuming for getting the solution.
- It is always complete.
- It is expensive
- It requires moderate time
- It is lengthy for implementation
- Examples as follows
  1. Breadth First Search (BFS)
  2. Depth First Search (DFS)
  3. Uniform cost search (UCS)
  4. Depth Limited Search (DLS)
  5. Iterative deepening DFS (IDDFS)
  6. Bi-directional search

Informed Search Strategies / Heuristic Function

- It contains information on goal state.
- It helps in search efficiently.
- The information is obtained by a function which will helps to estimate how close a current state is from the goal state.
- It uses the knowledge in the processing of searching.
- It helps to find the solution quickly.
- It may be or may not be complete.
- It is inexpensive
- It requires less time

- It gives the direction about the solution.
- It is less lengthy to implement.
- Examples of Informed Search

  1) Best First Search

  2) Greedy best first Search

  4) A* Search

  5) Memory bounded heuristic Search

3.2.2 Breadth First Search (BFS)

Uninformed Search Strategies:

3.2.2.1 Concept

- Breadth First Search: The root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- The shallowest unexpanded node is chosen for expansion.

3.2.2.2 Implementation

- This is achieved very simply by using a FIFO (First In First Out) Queue for the frontier.
- The goal test is applied to each node when it is generated rather than when it is selected for expansion.

3.2.2.3 Algorithm

1. Put the root node on a queue
2. While (queue is not empty)

   a) Remove a node from the queue

   if node is a goal node

   then return success;

   put all children of node onto the queue;

3. Return failure;

3.2.2.4 Performance Evaluation

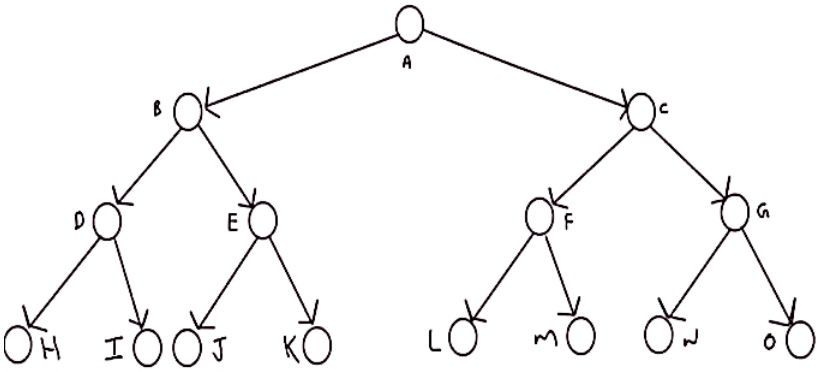
1) Completeness : It is complete, provided the shallowest goal node is at some finite-depth.

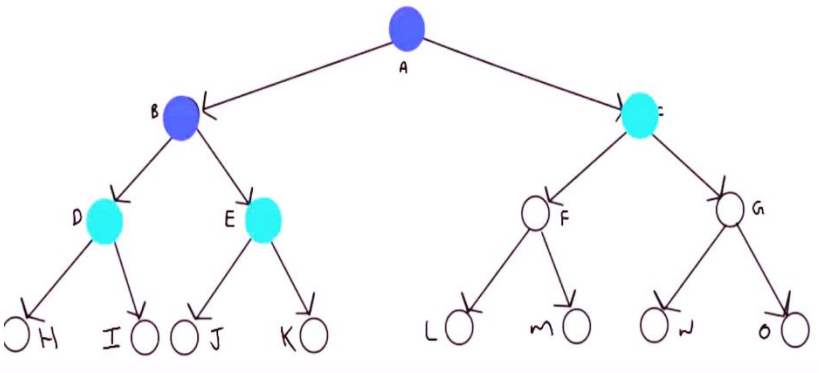
2) Optimality : It is optimal, as it always finds the shallowest solution.

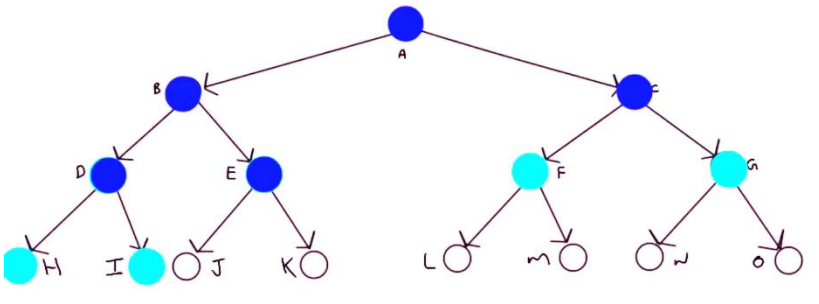
3) Time complexity : O(bd), number of nodes in the fringe.

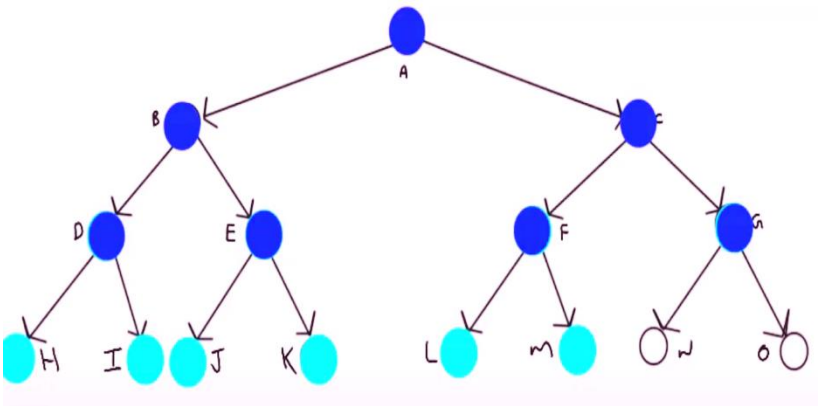
4) Space complexity : O(bd), total number of nodes explored.

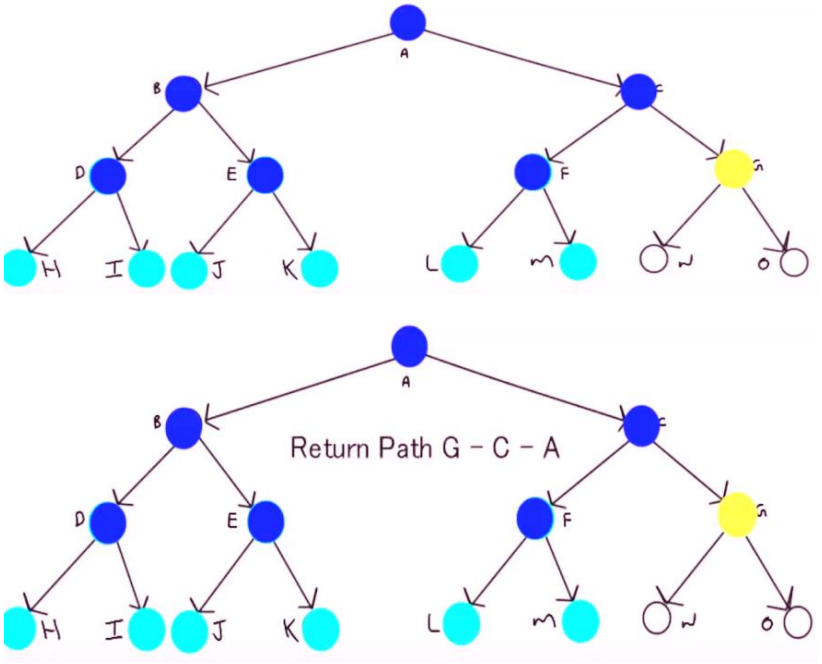
Process of Breadth First Search











3.2.3 Depth First Search (DFS)

3.2.3.1 Concept

- Depth first search always expands the deepest node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have to successors.
- As the nodes are expanded, they are dropped from the frontier, so then the search backs up the next deepest node that still has unexplored successors

3.2.3.2 Implementation

- Depth first search uses a LIFO (Last In First Out) queue.
- A LIFO (Last In First Out) queue means that the most recently generated node is chosen for expansion.

3.2.3.3 Algorithm

For Recursive implementation of DFS

DFS (c) :

Step 1. If node is a goal, return success;

Step 2. For each child c of node

If DFS (c) is successful,

Then return success;

Step 3. Return failure;

For non-recursive implementation of DFS

Step 1. Push the root node on a stack

Step 2. While (stack is not empty)

Pop a node from the stack;

If node is a goal node then return success;

Push all children of node onto the stack;

Step 3. Return failure

3.2.3.4 Performance Evaluation

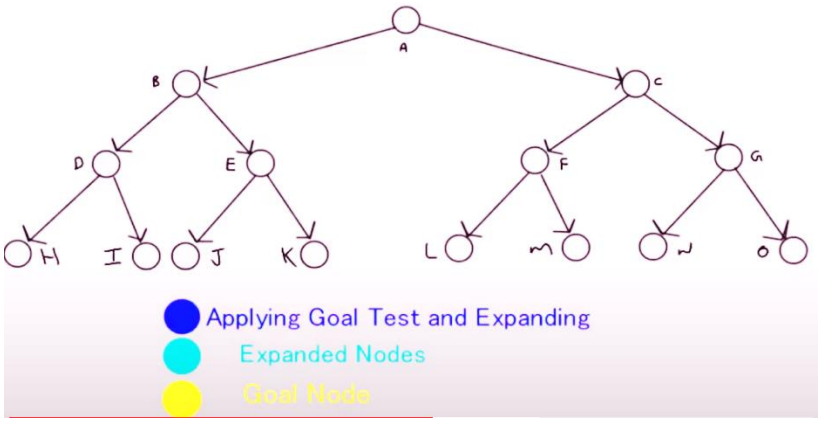
1) Completeness : Complete if DFS explores all the nodes hence it gives the assurance for solution.

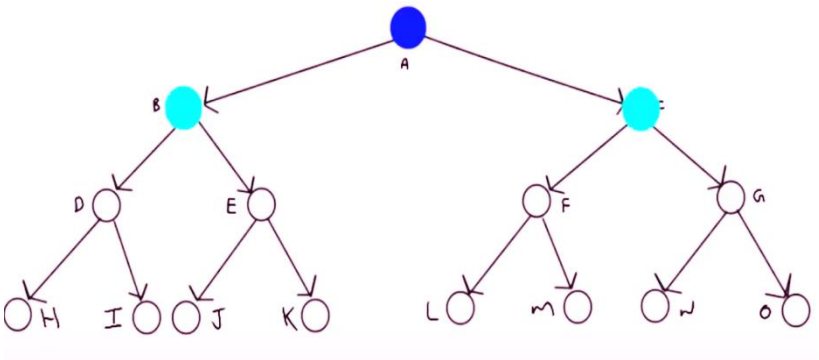
2) Optimality : No, as it cannot guarantee the shallowest solution.

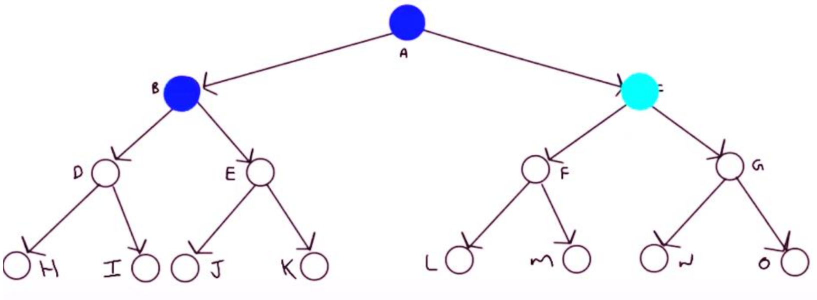
3) Time Complexity : DFS requires moderate amount of time to generate all the O(bm) nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space.

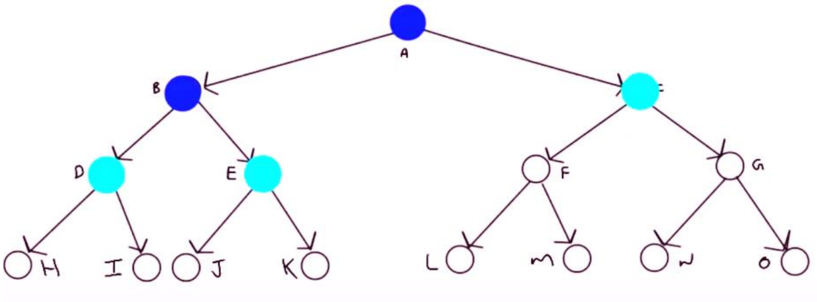
4) Space Complexity : To store single path from root to the some node to a particular level, along with unexpanded siblings, when same node gets fully explored, its complete branch i.e, descendants will get removed from the memory O(bm).

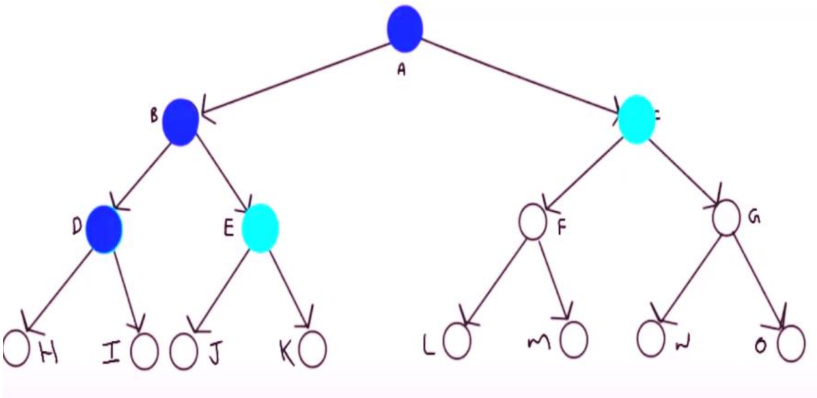
Process of Depth First Search

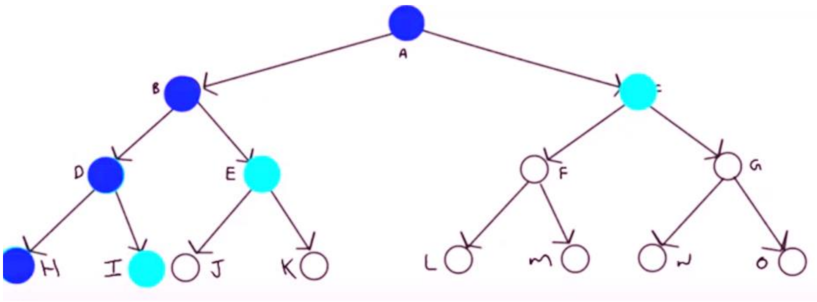


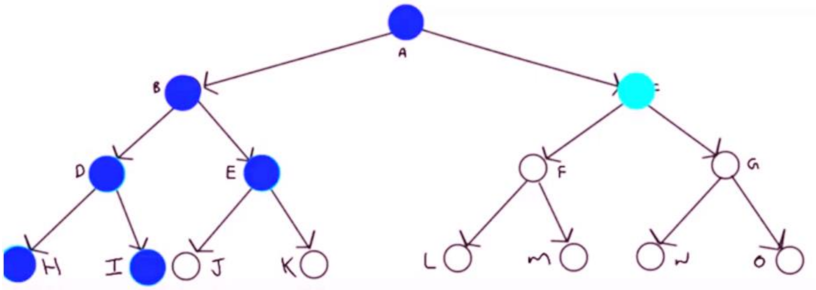


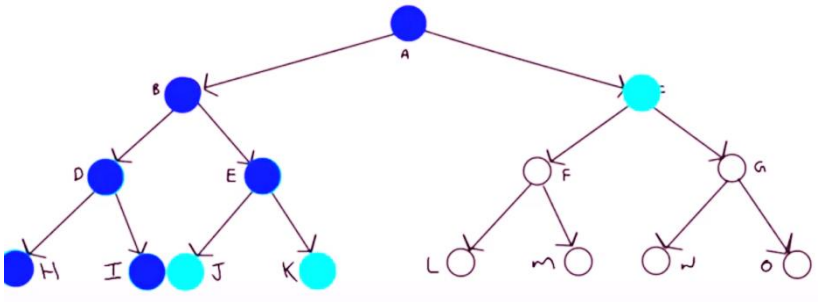

A
B
D
E
F
G
H
I
J
K
L
m
o

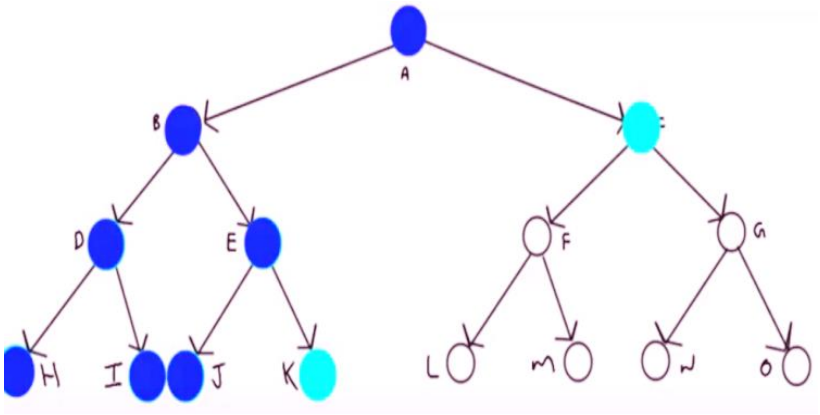

A
B
D
E
F
G
H
I
J
K
L
m
o

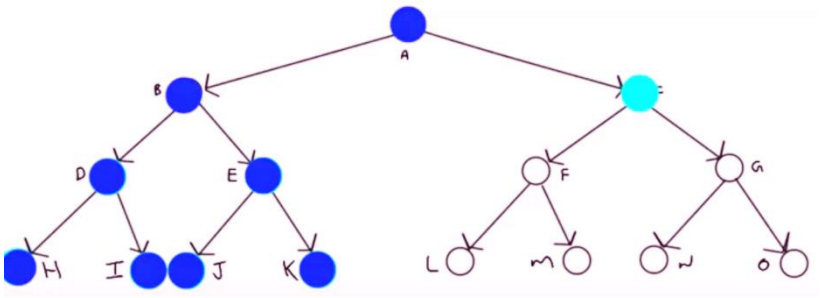

A
B
D
E
F
G
H
I
J
K
L
m
o

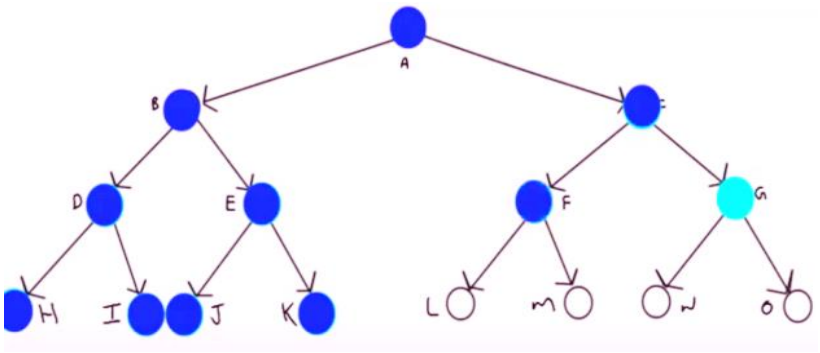

A
B
D
E
F
G
H
I
J
K
L
m
o

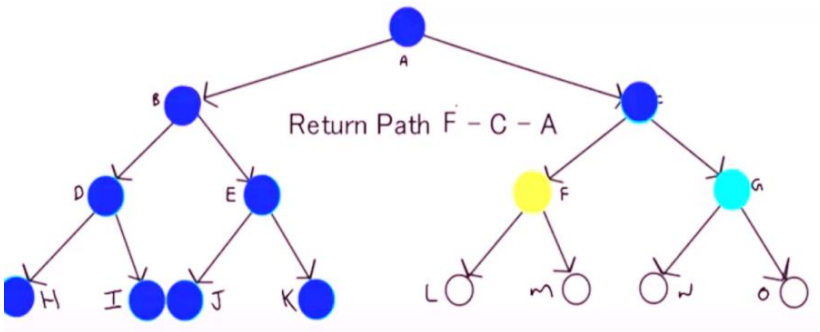

A
B
D
E
F
G
H
I
J
K
L
m
o


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


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


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


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


A
B
C
Return Path F - C - A
D
E
F
G
H
I
J
K
L
M
N
O

3.2.4 Uniform cost search (UCS)

3.2.4.1 Concept

- Uniform cost search is a breadth-first search with the same cost for all pathways. We can add a simple tweak to the basic implementation of BFS to make it work in real time. As a result, an algorithm with any path cost is optimal.
- Uniform cost search algorithm is used for visiting the weighted tree. The main aim or goal is to fetch a goal node and find the true path, including with its cumulative cost.
- With the help of BFS expansion of shallowest node first, but in uniform cost search, instead of expanding the shallowest node, the node with the lowest path cost will be expanded first.

3.2.4.2 Implementation

- Uniform cost search can be achieved by implementing the fringe as a priority queue ordered by path cost. It is same as BFS, except for the use of a priority queue and the additional check in case of the shortest path to any node if found.
- Algorithm will take care of the nodes which are inserted in the fringe for the exploration, with the help of priority queue and hash table data structures.
- Priority Queue will useful to count the total cost from the root to the node.
- UCS gives the minimum path cost with the maximum priority.

3.2.4.3 Algorithm

Step 1) Insert root node into the priority queue

Step 2) While the queue is not empty :

Dequeue the maximum priority node from the queue.

If priorities are equal then check the alphabets for smaller node

get selected

If the node is the goal node then print the path

Exit

Else

Insert all the children of the dequeued node, with their total costs as priority.

The algorithm returns the best cost path which is encountered first and will never go for other possible paths. The solution path is optimal in terms of cost.

Priority queue will count the total cost from root to the node path cost, it will take the shorter path cost.

The nodes in the priority queue have almost the same cost at a given time so the name given as Uniform Cost Search.

3.2.4.4 Performance Evaluation

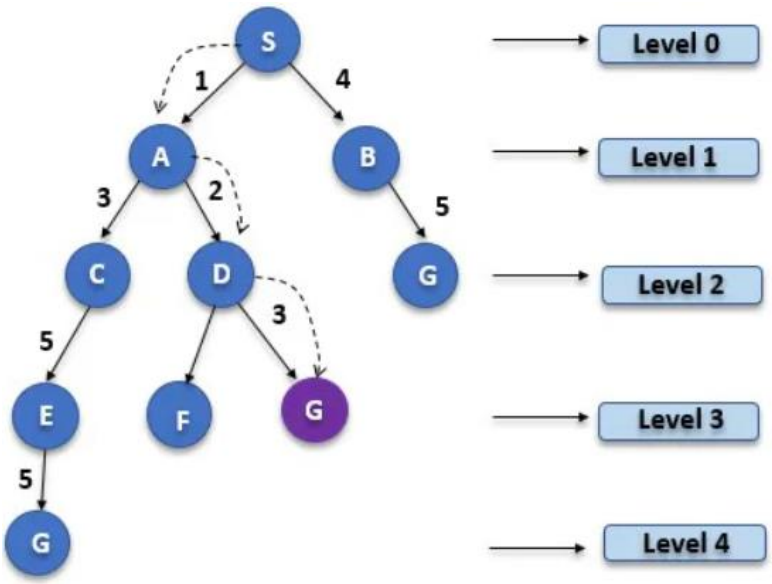
1) Completeness : UCS gives guaranteed cost of every step exceeds some small positive constant.

2) Optimality : It produces optimal solution as nodes are expanded in order of their path cost.

3) Time complexity : It considers path costs than the depths; so the complexity is not depends on b and d. Consider C^* be the cost of the optimal solution, we assume every action cost at least €. Then the algorithm's worst case time complexity will be $O(bC^*/€)$.

4) Space complexity : $O(bC^*/€)$ it indicates the number of node in memory at the time of execution.

Process



3.2.5 Depth Limited Search (DLS)

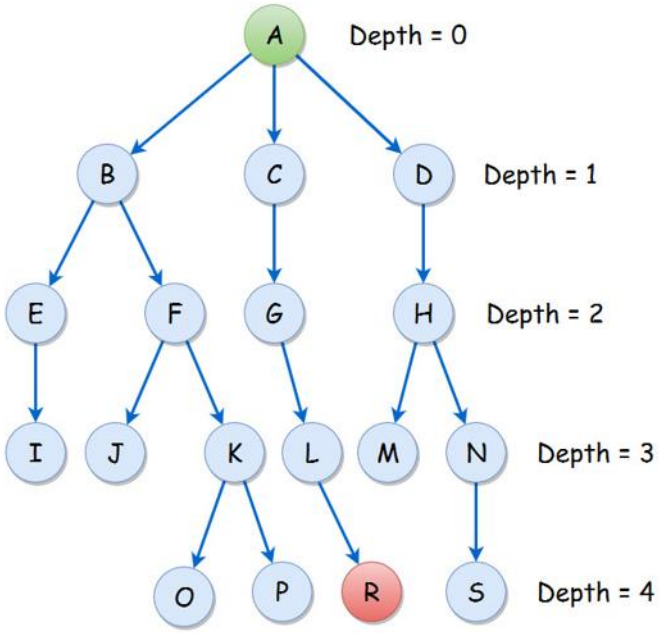
3.2.5.1 Concept

- In Depth limited search, DFS is performed upto a predefined depth
- It is a special case of DFS.
- It solves infinite path problems.

3.2.5.2 Implementation

- In case of DFS in DLS we can use the same fringe implemented is queue.

- Additionally the level of each node needs limited depth level l.
- DLS will terminate with two kinds of failure.
    - The standard failure value indicates no solution and the cut off value indicates no solution within the depth limit.



3.2.5.3 Algorithm

Step 1. Determine the start node and the search depth.

Step 2. Check if the current node is the goal node

If not then do nothing

If yes then return

Step 3. Check if the current node is within the specified search depth

If not then do nothing

If yes then expand the node and save all of its successors in a stack.

Call DLS function recursively for all nodes of the stack and go to step 2.

3.2.5.4 Performance Evaluation

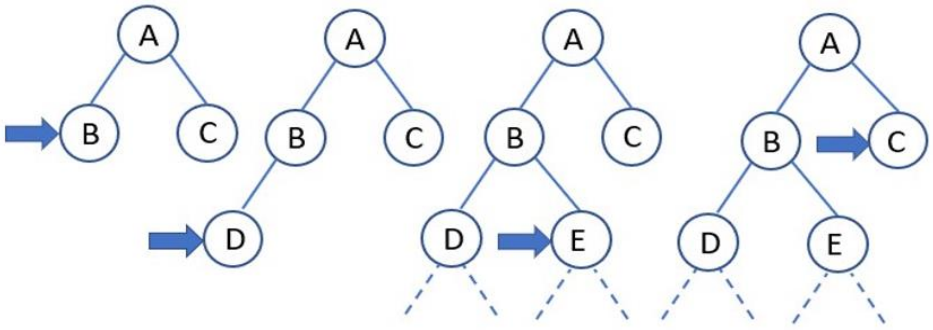
1) Completeness : If incomplete it shallowest goal is beyond the depth limit

2) Optimality: Non optimal, as the depth chosen can be greater than d.

3) Time complexity : It is same as DFS, O(bl), where l is the depth limit (specified)

4) Space complexity : It is same as DFS, O(bl), where l is the depth limit (specified)

Process



3.2.6 Iterative Deepening DFS (IDDFS)

3.2.6.1 Concept

- We know two common graph traverse methods BFS and DFS, Tree or graph with huge height and width, both BFS and DFS are not useful because

  1. DFS traverses adjacent of root then next adjacent…
  2. With this problem, DFS may not find shortest path to a node.
  3. While in BFS , it traverse level wise but requires high memory or space.
  4. With the help of IDDFS, to overcome DFS and BFS problems.
  5. It combines depth first search space for efficiency as well as breadth first search fast search (for nodes closer to root).

- Iterative Deepening Depth First Search (IDDFS) is an algorithm that is an important part of an Uniformed BFS and DFS.
- Keep on incrementing the depth limit by iterating the procedure unless we have reached to the goal node.
- Otherwise have to travel the whole tree which one is the easier.

3.2.6.2 Implementation

- In Iterative Deepening Depth First Search it works like BFS where queue concept uses, but more iterations are required to increment the depth limit by one in every recursive call of Depth Limit Search.
- IDDFS recursively calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. DFS as well as BFS both are implemented in IDDFS.

3.2.6.3 Algorithm

Step 1. Initial depth limit to zero.

Step 2. Repeat until the goal node is found

Step 3. Call Depth Limit Search with new depth limit.

Step 4. Increment depth limit to next level.

3.2.6.4 Performance Evaluation

1) Completeness : Iterative Deepening DFS is complete when the branching factor b is finite.

2) Optimality : It gives an optimal solution when the path cost is a non-decreasing function of the depth of the node.

3) Time complexity : It depends on branching factor (b) and the bottom most level that is depth (d). It is $O(b^d)$.

4) Space complexity : It has very moderate space complex which is O(bd).

Process

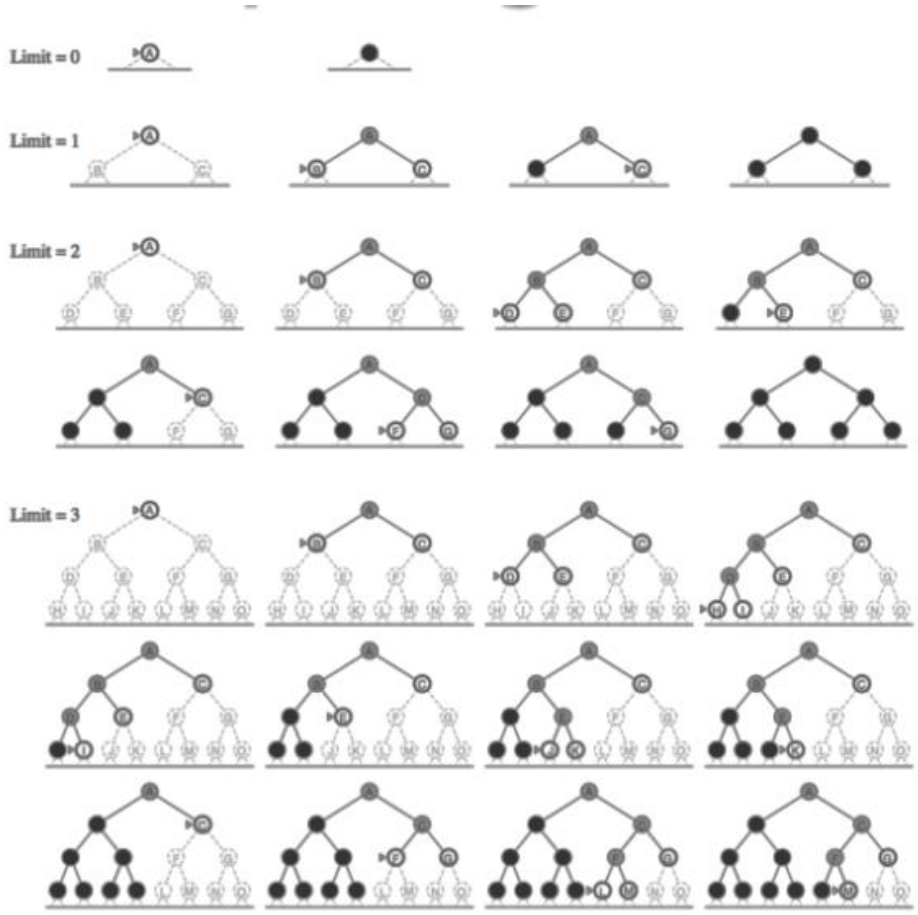


Figure: Search Process in IDDFS

3.2.7 Bidirectional Search

3.2.7.1 Concept

- Bidirectional search is a graph search algorithm, it finds a shortest path from an initial vertex to a goal vertex in a directed graph.
- Two graph traversals i.e. BFS take place at the same time and is used to find the shortest distance between a fixed start vertex and end vertex.
- It is a faster approach, it reduces the time required for the traversing the graph.
- It can be used for other applications as well.

3.2.7.2 Implementation

- It runs two simultaneous searches

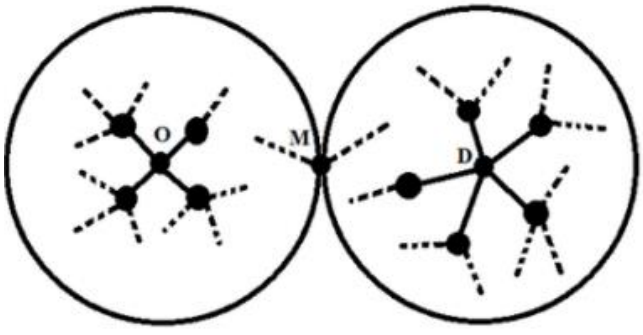
  One forward from the initial state and one backward from the goal, when both forward and backward state will meet that point it will stop.
- Two simultaneous searches are performed

3.2.7.3 Algorithm

- Undirected weighted graph, with all the edge weights assign to zero
- Bidirectional search from initial state and destination at the same time
- Stop both searches when a node in the middle (node M), becomes permanent in both directions.
- Suppose d(O,M)=X and d(D,M)=Y.
- Does the shortest path from O to D pass through M and is its cost equal to X+Y?

If so, then explain why and prove

If not, provide a counter example



3.2.7.4 Performance Evaluation

1) Completeness : Yes, if branching factor b is finite and both directions use breadth first search.

2) Optimality : Yes, if all costs are identical and both directions use breadth first search.

3) Time complexity : It uses BFS in both directions i.e from initial state and from goal state i.e. O(bd/2).

4) Space complexity : It requires at least one of the two fringes need to keep in a memory for checking common node, so the space complexity is O(bd/2).

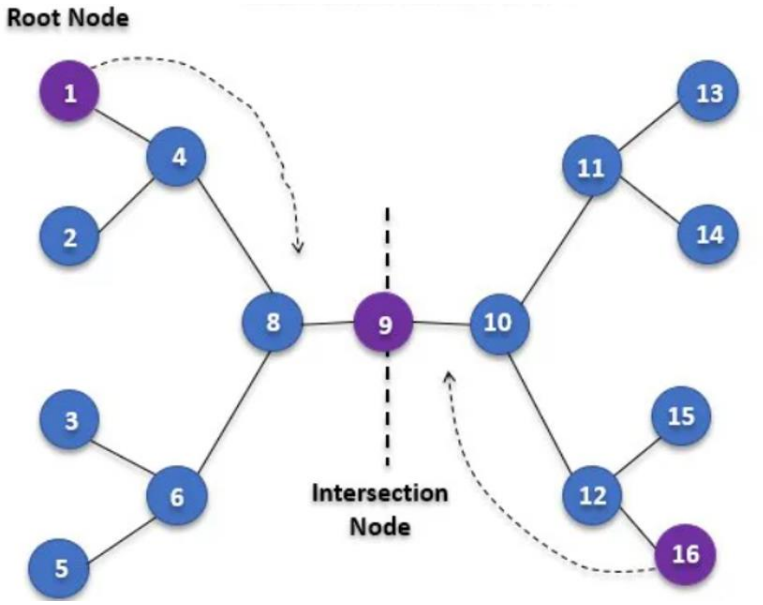
3.2.7.5 Advantages of Bidirectional Search

- Efficient
- It reduces space and time as it starts from both directions.
- If b=10 and d=6 then Breadth first search will require 106 = 1,000,000 nodes
- In bidirectional search requires 2 x 103 = 2,000 nodes.
- For better performance we can combine different search strategies in different directions.

3.2.7.6 Disadvantages of Bidirectional Search

- The search requires generating predecessors of states.
- Overhead of checking whether each node appears in the other search is involved.
- For larger depth d, it is still impractical.
- It uses both the directions i.e. initial and goal state directions with breadth first search algorithm with branching factor b and depth d of the solution for the memory requirement will be bd/2 for each search.

Process



3.2.8 Comparision of Searching Methods

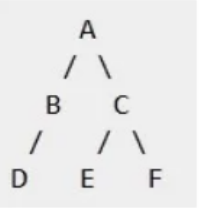
3.2.8.1 Unidirecional and Bidirectional Search

Unidirectional Search Method	Bidirectional Search Method
Unidirectional uses search tree, start node, goal node as input for starting search	Bidirectional have additional information about search trees nodes, along with the start and goal node.
They use only information from the problem definition.	They incorporate additional measure of a potential of a specific state to reach the goal.
Sometimes unidirectional search methods use past exploration.	All these methods use a potential of a state to reach a goal is measured through heuristic functions.
All these techniques are based on pattern of exploration of nodes in the search tree.	All these techniques totally depend on the evaluated value of each node generated by heuristic function.
In real time problems uninformed search techniques can be costly with respect to time and space.	In real time problems informed search techniques are cost effective with respective to time and space.
Comparatively more number of nodes will be explored in unidirectional search.	As compared to uninformed techniques less number of nodes are explored.
Examples: Breadth First Search Depth First Search Uniform Cost Search Depth Limited Search Iterative Depending Depth First Search.	Examples: Hill Climbing Search Best First Search A* Search IDA Search SMA* Search

3.2.8.2 Difference between BFS & DFS

KEY	**BFS**	**DFS**
Definition	BFS is Breadth First Search	DFS is Depth First Search
Implementation	BFS travels tree level wise. (Root node's nearer nodes will be visited first	DFS travels tree depth wise. (Depth-wise each node in a particular branch are

KEY	BFS	DFS
	from the left to right direction)	visited till the leaf node and then continues with searching branch by branch from left to right direction in a tree.
Data Structure	BFS uses the data structure queue with FIFO (First In First Out) list.	DFS uses the data structure stack with LIFO (Last In First Out) list.
No. of step algorithm	Single step algorithm. Visited vertices are removed from the queue and then displayed at once.	Two step algorithm. 1) Visited vertices are pushed onto the stack. 2) After pushing all vertices on stack if there are no vertex then visit one by one vertex and popped out.
Memory	Requires more memory as compare with DFS	Requires less memory as compare with BFS
Source	BFS is better when target is closer to source.	DFS is better when target is far from source.
Speed	BFS is slower than DFS	DFS is faster than BFS
Backtracking	BFS no need of backtracking.	DFS requires backtracking.
Loop	BFS can never get into infinite loops.	DFS generally gets trapped into infinite loops, so search tree are dense.
Optimality	BFS is optimal and complete if branching factor is finite.	DFS is neither complete nor optimal even in finite branching factor.
Suitable for decision tree	BFS considers all neighbour nodes so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the solution, then we won.

KEY	BFS	DFS
Time Complexity	O(V+E) where V = Vertices and E = Edges	O(V+E) V = Vertices and E = Edges
Applications	• To find shortest path • Single source and all pairs shortest paths • In Spanning tree • In Connectivity	• Useful in Cycle detection • In Connectivity testing • Finding a path between V and W in the graph • Useful in finding spanning trees and forest.
Tree	 A, B, C, D, E, F	 A, B, D, C, E, F

3.2.8.3 Comparison of tree search strategies basis on performance evaluation

The following table shows the comparison of all uninformed search techniques with respective their performance evaluation on the basis of completeness, optimality, time complexity, space complexity,

b : branching factor

l : depth limit

d : depth of the shallowest solution

m : maximum depth of the search tree

Parameters	BFS	Uniform Cost	DFS	DLS	IDDFS	Bidirectional
Completeness Optimality	Yes	Yes	No	No	Yes	Yes
Time Complexity	Yes	Yes	No	No	Yes	Yes
Space Complexity	O(bd)	O(b C*/€)	O(bm)	O(bl)	O(bd)	O(bd/2)
	O(bd)	O(b C*/€)	O(bm)	O(bl)	O(bd)	O(bd/2)

3.3 INFORMED SEARCH TECHNIQUES

Following search includes in Informed Search Strategies

Heuristic Search

Best First Search

Greedy best first Search

A* Search

Memory bounded heuristic Search

3.3.1 Heuristic

3.3.1.1 Introduction

A heuristic is a method that improves the efficiency of the search process. These are like tour guides. There are good to the level that they may neglect the points in general interesting directions. They are bad to the level that they may neglect points of interest to particular individuals. Some heuristics help in the search process without sacrificing any claims to entirety that the process might previously had. Others may occasionally cause an excellent path to be overlooked. By sacrificing entirely it increases efficiency. Heuristics may not find the best solution every time but guarantee that they find a good solution in a reasonable time. These are particularly useful in solving tough and complex problems, solutions of which would require infinite time, i.e. far longer than a lifetime for the problems which are not solved in any other way.

3.3.1.2 Heuristic Search

To find a solution in proper time rather than a complete solution in unlimited time we use heuristics. 'A heuristic function is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers'. Heuristic search methods use knowledge about the problem domain and choose promising operators first. These heuristic search methods use heuristic functions to evaluate the next state towards the goal state. For finding a solution, by using the heuristic technique, one should carry out the following steps:

1. Add domain—specific information to select what is the best path to continue searching along.

2. Define a heuristic function h(n) that estimates the 'goodness' of a node n.

   Specifically, h(n) = estimated cost(or distance) of minimal cost path from n to a goal state.

3. The term, heuristic means 'serving to aid discovery' and is an estimate, based on domain specific information that is computable from the current state description of how close we are to a goal.

Finding a route from one city to another city is an example of a search problem in which different search orders and the use of heuristic knowledge are easily understood.

1. State: The current city in which the traveller is located.
2. Operators: Roads linking the current city to other cities.
3. Cost Metric: The cost of taking a given road between cities.
4. Heuristic information: The search could be guided by the direction of the goal city from the current city, or we could use airline distance as an estimate of the distance to the goal.

3.3.1.3 Heuristic Search Techniques

For complex problems, the traditional algorithms, presented above, are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed, using heuristic functions.

- Blind search is not always possible, because it requires too much time or Space (memory).

Heuristics are rules of thumb; they do not guarantee a solution to a problem.

- Heuristic Search is a weak technique but can be effective if applied correctly; it requires domain specific information.

3.3.1.4 Characteristics of Heuristic Search

- Heuristics are knowledge about domain, which help search and reasoning in its domain.
- Heuristic search incorporates domain knowledge to improve efficiency over blind search.
- Heuristic is a function that, when applied to a state, returns value as estimated merit of state, with respect to goal.
    i) Heuristics might (for reasons) underestimate or overestimate the merit of a state with respect to goal.
    ii) Heuristics that underestimate are desirable and called admissible.
- Heuristic evaluation function estimates likelihood of given state leading to goal state.
- Heuristic search function estimates cost from current state to goal, presuming function is efficient.

3.3.1.5 Comparison of Blind Search and Heuristic Search

Blind Search	Heuristic Search
It Can only search what it already has knowledge.	It estimates distance to goal state through explored nodes.
Do not have knowledge about how far a node from goal state.	It gives the guidance for search process toward goal. It always prefers state (nodes) that lead close to and not away from goal state.

Example:

Travelling salesman

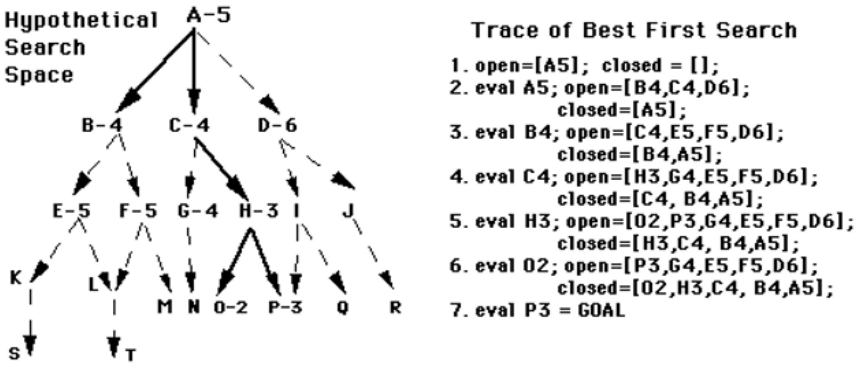
A salesperson must visit a list of cities, each of which must be visited just once. There are several routes that connect the two cities. The issue is determining the shortest path between cities so that the salesman can visit all of them at the same time.

If there are N cities, a solution would be to try N! different combinations to get the shortest distance and hence the required route. This is inefficient because there are 36,28,800 potential routes with N=10. Combinatorial explosion is demonstrated here.

There are better ways to solve such problems, one of which is termed branch and bound. To begin, produce all of the complete paths and calculate the distance between the first and last complete paths. If the next path is shorter, save it and continue in this manner, avoiding the path when its length exceeds the saved shortest path length, however this method is preferable to the prior method.

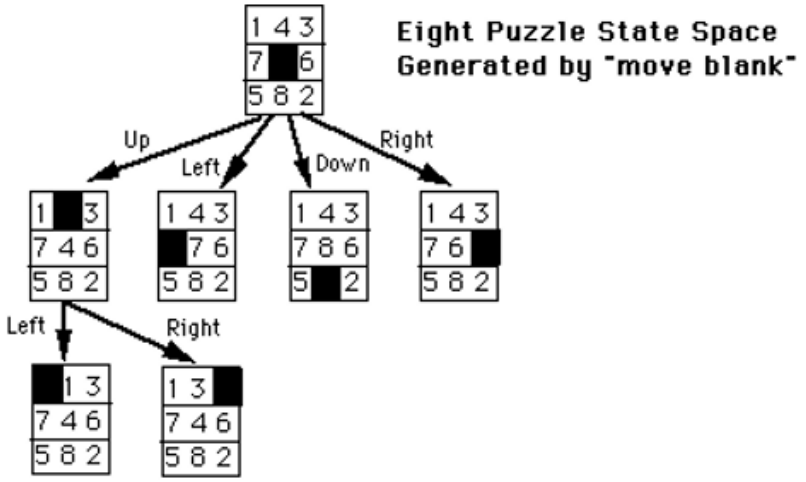
3.3.1.6 Heuristic Function

The heuristic function is a method of informing the search about a goal's direction. It allows you to make an educated prediction about which of a node's neighbours will go to a destination. A heuristic function isn't magical in any way. It can only use information about a node that is easily accessible.



Heuristic Evaluation Functions

Let us discuss 8-Puzzle Problem



- Heuristic 1 (H1) : Count the out-of-place tiles, as compared to the goal.
- Heuristic 2 (H2): Sum the distances by which each tile is out of place.
- Heuristic 3 (H3): Multiply the number of required tile reversals by 2.

Weaknesses

- H1 doesn't account for the distance that tiles have to move to get in place.
- H2 doesn't account for tile-reversals. It takes several moves to reverse two adjacent tiles.
- H3 often fails to distinguish between vastly different states.

Consider the following example:

Start	Goal	H1	H2	H3
2 8 3 1 6 4 - 7 5	1 2 3 8 - 4 7 6 5	5	6	0
2 8 3 1 - 4 7 6 5	1 2 3 8 - 4 7 6 5	3	4	0
2 8 3 1 6 4 7 5 -	1 2 3 8 - 4 7 6 5	5	6	0

Analysis of the Evaluation Function

For developing a good evaluation function for the states in a search space, we need

g(n) : how far is the state n from the start state?

h(n) : how far is state n from a goal state?

The first value, g(n), is important because you often want to find the shortest path. This value (distance from start) can be exactly measured by incorporating a depth count into the search algorithm.

The second value, h(n), is important for guiding the search toward the goal. It is an estimated value based on your heuristic rules.

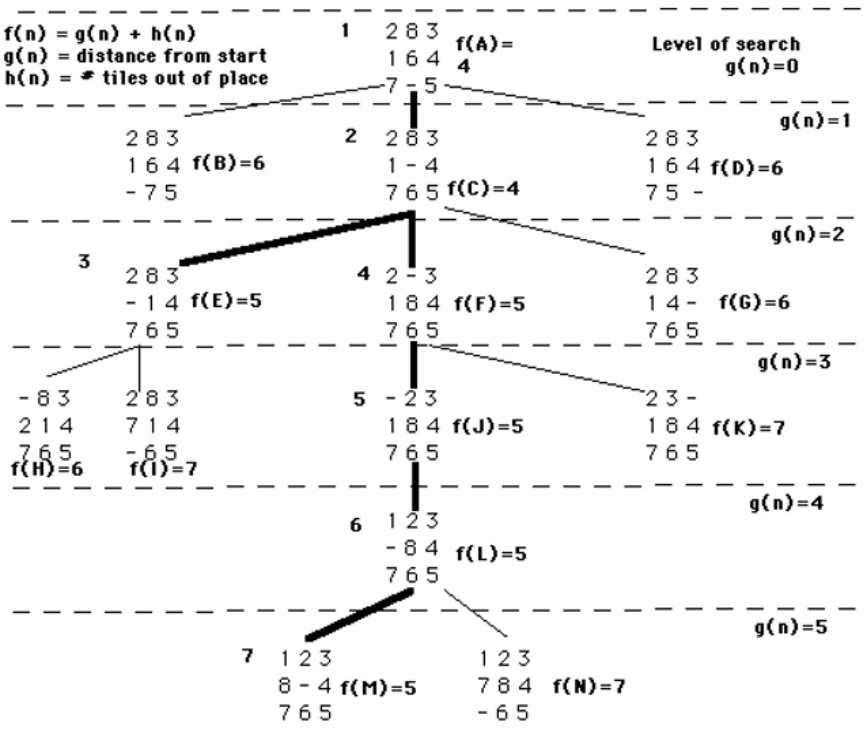
Evaluation function. This gives us the following evaluation function:

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

where g(n) measures the actual length of the path from the start state to the state n, and h(n) is a heuristic estimate of the distance from a state n to a goal state.

Heuristic Applied to the Eight Puzzle

The following figure shows the full best-first search of the eight puzzle graph using H1 -- that is, the value of h(n) is the number of tiles out of place. The number at the top of each state represents the order in which it was taken off of the open list. The levels of the graph are used to assign g(n). For each state, its heuristic value, f(n) is shown.



that occurs in step 3, where E and F have the save value. State E is examined first, which puts its children, H and I, on the open list. But since state f(F) has a lower value than f(H) and f(I), the search is prevented from going down to those children. In effect, the g(n)gives the state more of a breadth-first search, keeping it from going too deeply down paths that don't get near a goal.

- Operations on states generate children of the state currently being evaluated.
- To prevent loops, Each new state is checked to see whether it occurred before.
- Each state, n, is given an f value, using f(n) = g(n) + h(n), where h guides the search toward promising states and g prevents it from going too far on a dead end.
- States on open are sorted by their f values.
- Efficiency can be improved by efficient management of the open and closed lists.

3.3.2 Best First Search :

3.3.2.1 Concept :

- Search will start at root node
- The node to be expanded next is selected on the basis of an evaluation function, f(n).
- The node having lowest value for f(n) indicates that goal is nearest from this node i.e. f(n) indicates distance from current node to goal node.

3.3.2.2 Implementation

- Best first search uses two lists in order to record the path i.e. OPEN list and CLOSED list for implementation.
- OPEN list stores nodes that have been generated, but have not examined. It is organised by using the priority queue with the help of storing the nodes in increasing order of their heuristic value, for maximization heuristic. So that we can get the efficient selection of the current best candidate for extension.
- CLOSED list stores nodes that have already visited. It contains all nodes that have been evaluated and will not be looked at again.
- For new node generation, check whether, it has been generated before that, If it is already visited then check all its recorded value and then select the next node as a parent if this new node value is better than previous one.
- Because of this selection it will avoid any node being evaluated twice, and will never get stuck into an infinite loops.

3.3.2.3 Algorithm

Step 1) Two ordered list OPEN = [initial state]
Closed = [ ]

Step 2) If OPEN is empty then exit with failure.

Step 3) Select first node on OPEN. Remove it from OPEN and put it on CLOSED.

Call this node n.

Step 4) If 'n' is the goal node then exit.

Step 5) The solution obtained by tracing a path backward along the arc in the tree from 'n' to the initial node i.e. 'n1'

Step 6) Expand node 'n', it will generate successors.

Step 7) The set of successors generated be S then create arcs from 'n' to each number of S.

Step 8) Reorder the list OPEN, according to the heuristic and go back to step 2.

3.3.2.4 Performance Evaluation

1) Completeness : It is not complete, it may follow the infinite path and never return to try other possibilities which can give solution.

2) Optimality :It is not optimal as it can initially select low value h(n) node but it may happen that some greater value node in current fringe can lead to better solution. Greedy strategies suffers from this, "looking the current best they loose on future best and will give finally the best solution.

3) Time complexity : Worst case time complexity $O(b^m)$ where 'm' is the maximum depth of the search space. The complexity can be reduced by using good heuristic function.

4) Space complexity : $O(b^m)$ where 'm' is the maximum depth of the search space. The complexity can be reduced by using good heuristic function.

Process

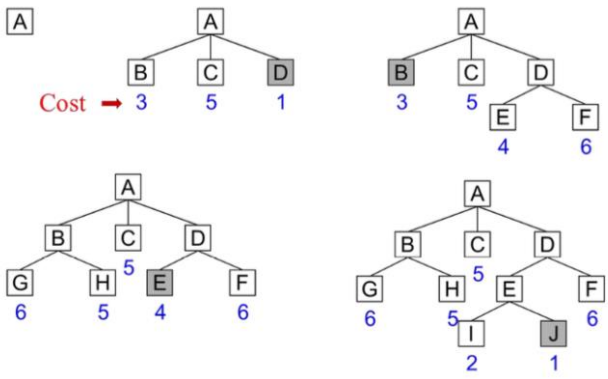


Figure : Best First Search

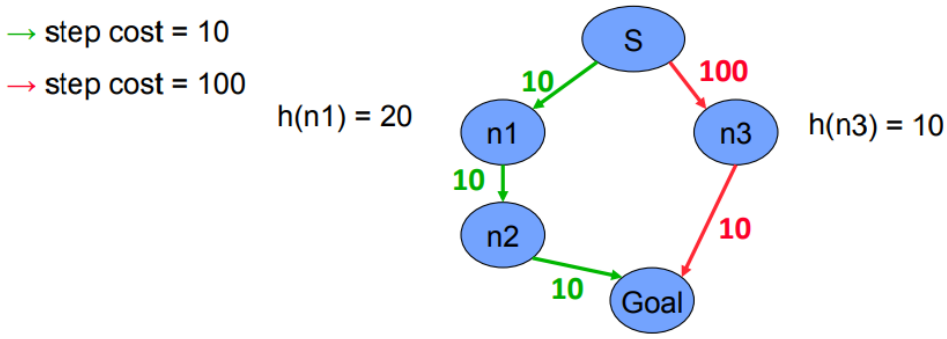
3.3.3 Greedy Best First Search

3.3.3.1 Concept

- We use h(n) to rank the nodes on OPEN
- Always expand node with lowest h-value.
- We are greedily trying to achieve a low cost solution.

3.3.3.2 Implementation

- In Greedy Best First Search method ignores the cost of getting to n, so it can be lead astray exploring nodes that cost a lot but near or close to the goal node.



3.3.3.3 Algorithm

Step 1) First successor of the parent is expanded. For the successor node

If the successor node's heuristic is better than its parent, the successor is set at the front of the queue, with the parent reinserted directly behind it, and the loop restarts.

Else

If The successor is inserted into the queue, in a location determined by its heuristic value.

The procedure will evaluate the remaining successors, if any of the parent.

In many cases, greedy best first search may not always produce an optimal solution, but the solution will be locally optimal, as it will be generated in comparatively less amount of time.

In mathematical optimization, greedy algorithm solve combinatorial problems.

3.3.3.4 Performance Evaluation

1) Completeness : It is not complete, it may get stuck in loops, or may be taken wrong start and quality of heuristic function.

2) Optimality : It is not optimal, as in selection of single path and never checks for other various possibilities.

3) Time complexity : $O(b^m)$ but with good heuristic, will give dramatic improvement.

4) Space complexity : $O(b^m)$, needed to keep all nodes in memory.

Process

Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

3.3.4 A* SEARCH

3.3.4.1 Concept

- The A* Search method is one of the most widely used path-finding and graph traversal techniques.
- Unlike other traversal strategies, A* Search algorithms have "brains." What this means is that it is a sophisticated algorithm that distinguishes itself from other algorithms. In the sections below, this truth is clarified in great depth.
- It's also worth noting that many games and web-based maps employ this approach to effectively locate the shortest path (approximation).

3.3.4.2 Implementation

- A * algorithm is a path-finding algorithm that looks for the shortest path between the starting and ending states. It's utilised in a variety of applications, including maps. The A* algorithm is used in maps to find the shortest distance between a source (initial state) and a destination (final state) (final state).
- At each step it picks the node according to a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell.
- We define 'g' and 'h' as simply as possible below
- g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

- h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate 'h'.

3.3.4.3Algorithm

Step 1) Initialize the open list

Step 2) Initialize the closed list

Step 3) Put the starting node on the open list (you can leave its f at zero)

Step 4) While the open list is not empty

a) find the node with the least f on the open list, call it "q"

b) pop q off the open list

c) generate q's 8 successors and set their parents to q

d) for each successor

i) if successor is the goal, stop search

successor.g = q.g + distance between successor and q

successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

successor.f = successor.g + successor.h

ii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor

iii) if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor

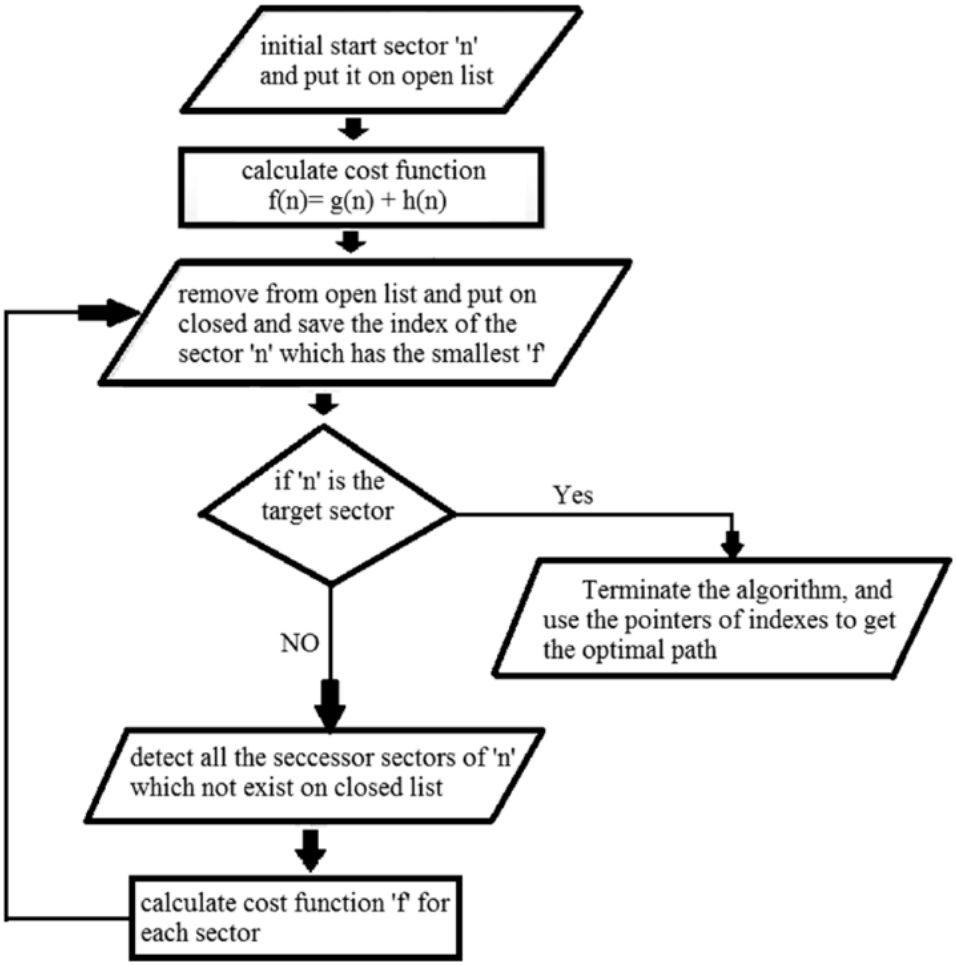
otherwise, add the node to the open list

end (for loop)

e) push q on the closed list

end (while loop)

3.3.4.4 Flow chart of A* search algorithm



3.3.4.5 Performance Evaluation

1) Completeness : It is complete, as it will always find solution if one exist.

2) Optimality : Yes, it is optimal.

3) Time Complexity : $O(b^m)$ the number of nodes grows exponentially with solution cost.

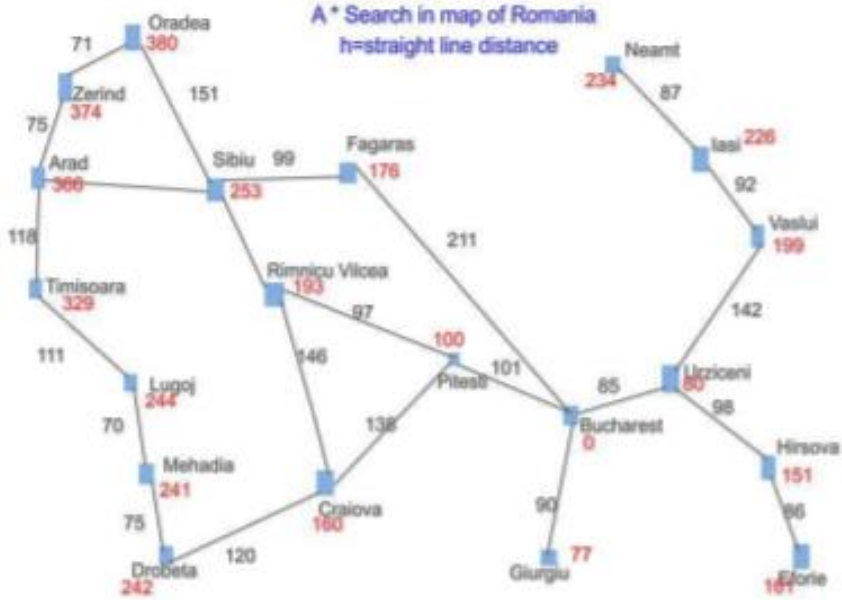
4) Space Complexity : $O(b^m)$ it keeps all nodes in memory.

Process

Search in map of Romania

- This figure represents the intial map of Romania.The values representing in red colour are heuristic values(i.e h(n)).
- The values representing in black colour are path cost values(i.e g(n)).
- The values representing in blue colour are f(n) values i.e.

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



After expanding Arad

- We have three nodes i.e Zerind, Sibiu and Timisoara.
  - f(n) = g(n) + h(n)
  - f(Sibiu)=f(n)=140+253=393(g(n)=140 and h(n)=253).
  - f(Zerind)=f(n)=75+374=449(g(n)=75 and h(n)=374).
  - f(Timisoara)=f(n)=118+329=447(g(n)=118 and h(n)=329).
- From these nodes we have to choose the least f(n) , so f(Sibiu) is least among these nodes.

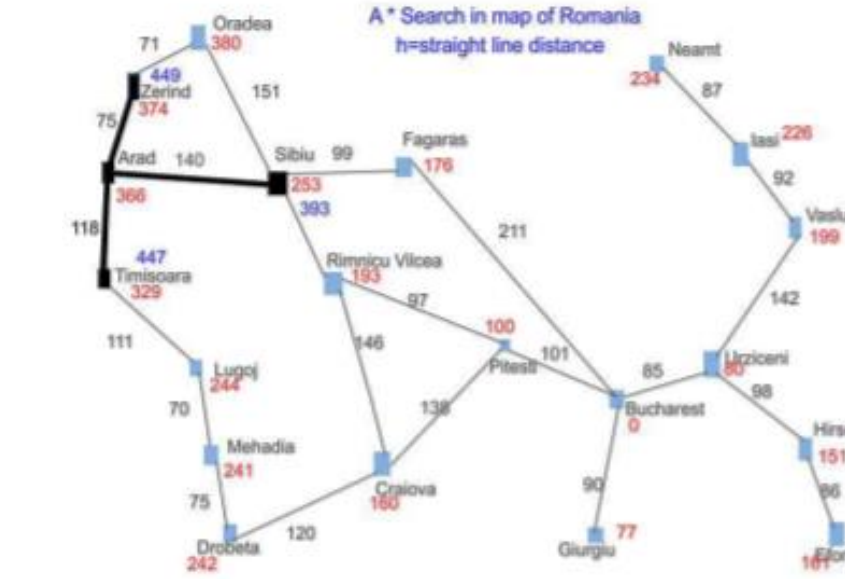


Figure : After expanding Arad

- After expanding Sibiu we have four nodes i.e Arad, Oradea, Fagaras and Rimnicu Vilcea.
- f(n) = g(n) + h(n)
- f(Arad) = f(n) = 280+366=646 g(n)=280 and h(n) =366
- f(Oradea)=f(n)=291+380=671(g(n)=291 and h(n)=380).
- f(Fagaras)=f(n)=239+176=415(g(n)=239 and h(n)=176).
- f(Rimnicu)=f(n)=220+193=413(g(n)=220 and h(n)=193).
- From these nodes we have to choose the least f(n) value, so f(Rimnicu) is

  Least among these nodes. f(Rimnicu)=413

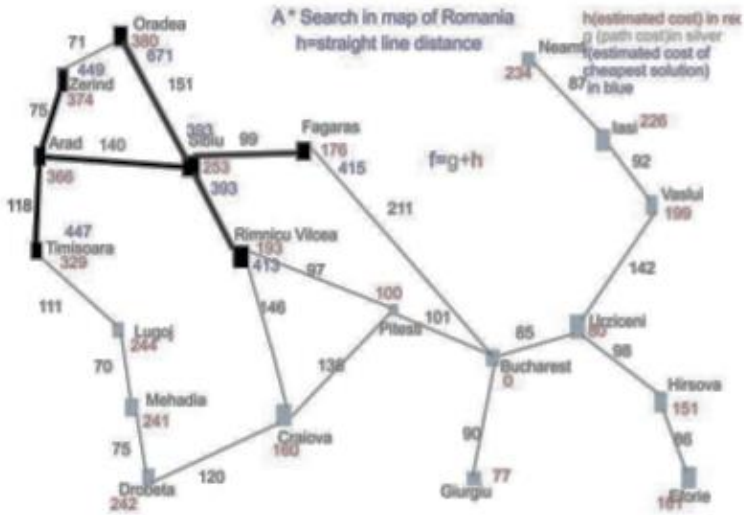


Figure After expanding Sibiu

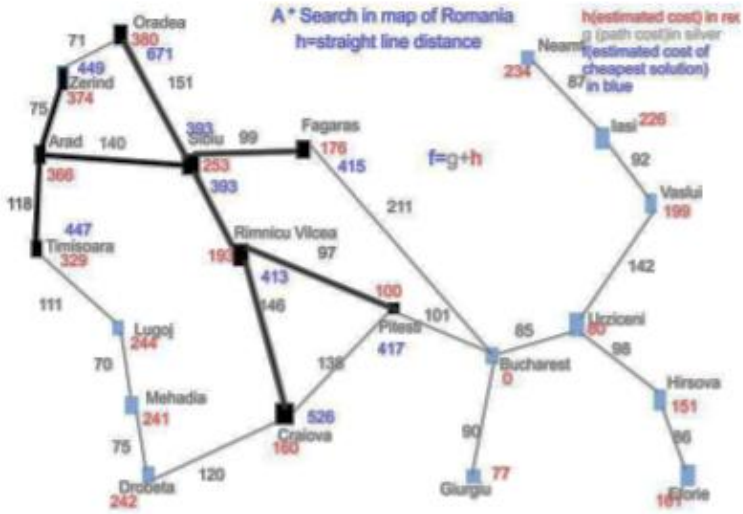


Figure : After expanding Rimnicu Vilcea

After exapanding Rimnicu

- After expanding Rimnicu node we have three nodes i.e Craiova, Pitesti and Sibiu.
- f(n0 = g(n) + h(n)
- f(Craiova)=f(n)=366+160=526(g(n)=366 and h(n)=160).
- f(Pitesti)=f(n)=317+100=417(g(n)=317 and h(n)=100).
- f(Sibiu)=f(n)=300+253=553(g(n)=300 and h(n)=253).
- From these three nodes we have to choose the least f(n) value, so f(Fagaras)
  is least among the nodes

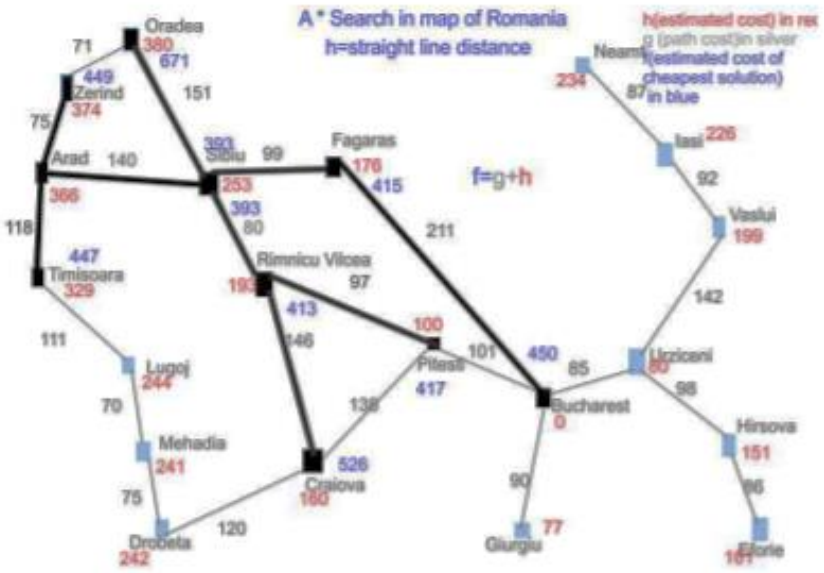


Figure After expanding Fagaras

After Expanding Fagaras

- After expanding Fagaras node we have two nodes i.e Sibiu and Bucharest.
- f(n) = g(n) + h(n)
- f(Sibiu)=f(n)=338+253=591(g(n)=338 and h(n)=253).
- f(Buchrest)=f(n)=450+0=450(g(n)=450 and h(n)=0).
- From these three nodes we have to choose the least f(n) value, so f(Fagaras) is least among the nodes.

After expanding Pitesti

- After expanding Pitesti node we have three nodes i.e Bucharest, Craiova and Rimnicu.

- f(n) = g(n) + h(n)
- f(Craiova)=f(n)=455+160=615(g(n)=455 and h(n)=160).
- f(Buchrest)=f(n)=418+0=418(g(n)=418 and h(n)=0).
- f(Rimnicu)=f(n)=414+193=607(g(n)=414 and h(n)=193).

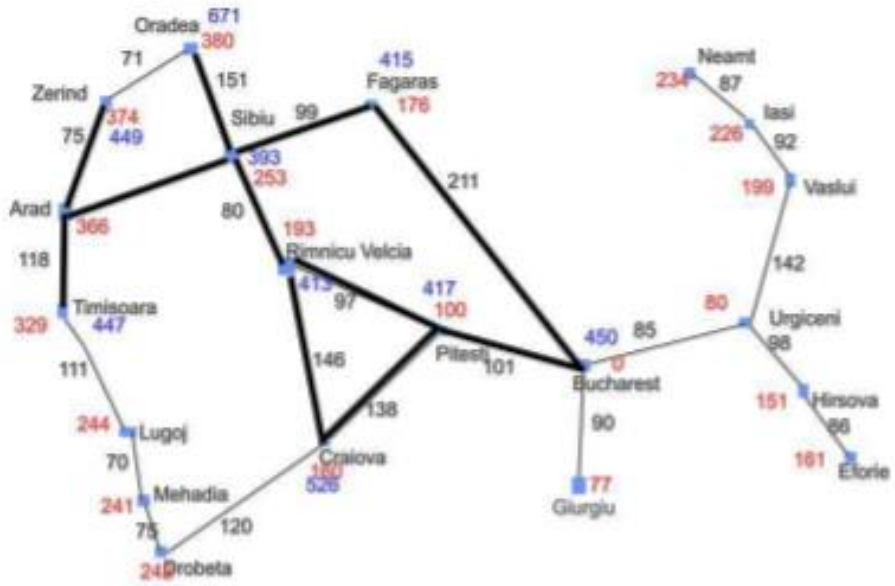


Figure After expanding Pitesti

- From these three nodes we have to choose the least f(n) value, so f(Bucharest)
  is least among the nodes
- Since Bucharest is the goal node, so h(n)=0 at Bucharest.

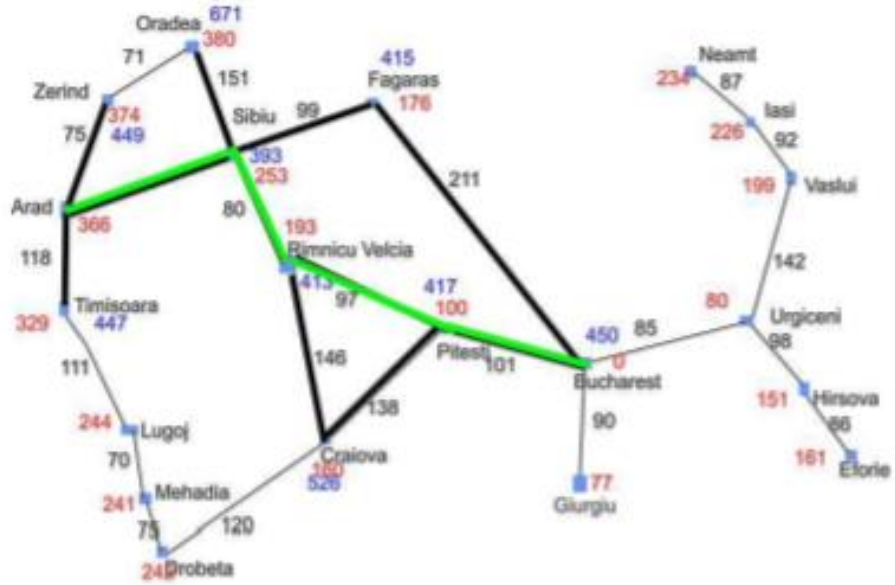


Figure : Shortest path from Arad to Bucharest

3.3.5 Memory Bounded Heuristic Search

3.3.5.1 Concept

- To reduce memory, we are using Iterative deepening to the heuristic search.
- There are two memory bonded algorithm :
  1) RBFS – Recursive best first search
  2) MA* - Memory bounded A* and
     SMA* - Simplified memory MA*

1) RBFS : Recursive Best First Search
   - It is same as BFS for mimic the various operations.
   - It replaces the f-value of each node along the path with the best f-value of its children.
   - Suffers from using very little memory.
   - Even more memory were available, but Recursive Best First Search has no way to make use of it.

3.3.5.2 SMA* : Simplified Memory Bounded A*

- Proceeds life A* expands best leaf until memory is full.
- Cannot add new node without dropping an old one.
- Always ready to drop worst node.
- Expand the leaf which is best and delete the node which are worst leaf.
- If all have same f-value then selection of same node for expansion as well for deletion.
- SMA* is complete if we received the final solution.

3.3.6 Local Search Algorithm and Optimization Problems

3.3.6.1 Hill Climbing

Hill Climbing is a type of heuristic search used in the field of Artificial Intelligence to solve mathematical optimization issues.

It seeks to discover a sufficiently good solution to the problem given a broad number of inputs and a good heuristic function. It's possible that this isn't the best option in the world.

- According to the definition above, mathematical optimization problems are those in which we must maximise or minimise a given real function by selecting values from provided inputs. For example, consider the travelling salesman dilemma, in which we must reduce the distance travelled by salespeople.

- The term "heuristic search" refers to the possibility that this search method will not identify the best answer to the problem.
- A heuristic function is a function that ranks all viable alternatives in a search algorithm at any branching step depending on the information provided. It aids the algorithm in selecting the optimum route from a list of options.

Features of Hill Climbing

1) Generate a possible solutions
2) Testing of expected solution.
3) If the solution found then quit

Else go to step 1.

a) Because it incorporates feedback from the test procedure, we refer to Hill Climbing as a version of the produce and test algorithm. The generator then uses this information to determine the next step in the search process.

b) Employs the Greedy approach: At any point in state space, the search moves only in the direction that optimises the cost of function, in the hopes of eventually discovering the best answer.

Types of Hill Climbing

Simple Hill Climbing: It checks each surrounding node individually and chooses the first neighbouring node that minimises the current cost as the next node.

1. Algorithm for Simple Hill climbing :

Step 1 : Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.

Step 2 : Loop until the solution state is found or there are no new operators present which can be applied to current state.

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

b) Perform these to evaluate new state

i) If the current sate is a goal state, then stop and return success.

ii) If it is better than current state, then make it current state and proceed further.

iii) If it is not better than the current state, then continue in the loop until a solution found.

Step 3 : Exit.

2. Steepest-Ascension Hill Climbing: It checks all nearby nodes first, then chooses the node that is closest to the solution state as the next node.

   Step 1 : Evaluate the initial state. If it is goal state then exit else make the current state as initial state

   Step 2 : Repeat these steps until a solution is found or current state does not change

   i. Let ‘target’ be a state such that any successor of the current state will be better than it;

   ii. for each operator that applies to the current state

      a. apply the new operator and create a new state

      b. evaluate the new state

      c. if this state is goal state then quit else compare with ‘target’

      d. if this state is better than ‘target’, set this state as ‘target’

      e. if target is better than current state set current state to Target

Step 3 : Exit

3. Stochastic hill climbing: It does not look at all of the nearby nodes before determining which one to choose. It just chooses an adjacent node at random and decides whether to go to that neighbour or inspect another based on the level of progress in that neighbour.

   State Space Diagram for Hill Climbing

   A space diagram depicts the collection of states that our search method can reach in relation to the value of our objective function (the function which we wish to maximize).

   The X-axis represents the state space, or the possible states or configurations that our algorithm could achieve.

   The values of the goal function corresponding to a specific state are represented on the Y-axis.

   The best solution will be found in the state space with the highest value of the objective function (global maximum).

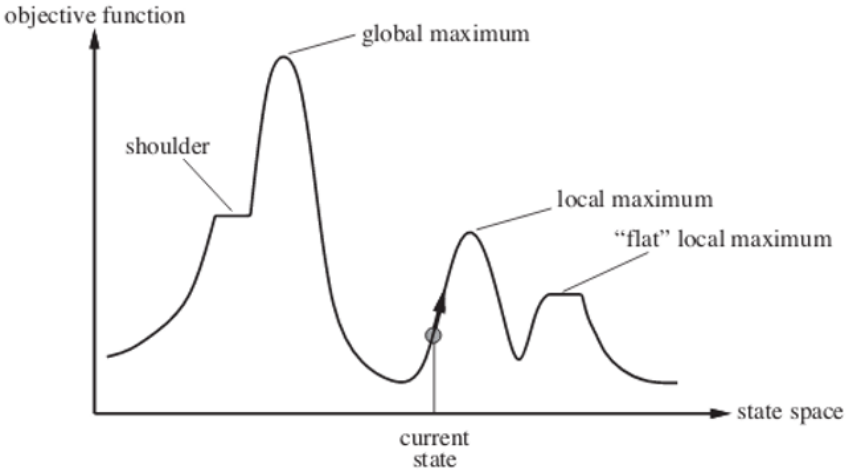


Figure : Different regions in the State Space Diagram

1. Local maximum: A state that is better than its neighbouring state, yet there is another state that is better (global maximum). This condition is preferable since the objective function value is higher than that of its neighbours.

2. Global maximum: In the state space diagram, this is the best possible state. This is because the objective function has the maximum value in this state.

3. Plateua/flat local maximum: This is a flat region of state space with the same value for nearby states.

4. Ridge: A ridge is a place that is higher than its neighbours but has a slope to it. It's a unique type of local maximum.

5. Current state: The area of the state space diagram in which we are currently situated during the search.

6. Shoulder: It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing

If hill climbing hits any of the following regions, it will be unable to attain the optimal/best state (global maximum):

1. Local maximum: All adjacent states have values that are worse than the current state at a local maximum. Because hill climbing has a greedy attitude, it will not deteriorate and eventually end. Even if a superior option exists, the process will come to an end.

   Use the backtracking approach to solve the local maximum problem. Keep track of the states you've visited. If the search hits an unfavourable point, it can revert to the original configuration and choose a different route.

2. Plateau: All neighbours on a plateau have the same value. As a result, choosing the ideal direction is impossible.

To overcome a plateau, take a major step forward. Choose a state that is far distant from your current location at random. It's likely that we'll land somewhere other than a plateau.

3. Ridge: Because movement is downhill in all directions, any point on a ridge can appear to be a summit. As a result, the algorithm comes to a halt when it reaches this point.

To overcome Ridge: Use two or more rules before testing in this type of obstacle. It entails travelling in multiple directions at the same time.

3.3.6.2 Local Beam Search

Introduction

A heuristic strategy is a set of criteria for assessing which of several alternatives will be most effective in reaching a specific goal. By giving up claims of systematic and completeness of the best, this technique improves the efficiency of a search process.

If we utilise proper heuristics, we can hope to solve complex tasks (such as the travelling salesman problem) in less than exponent time.

Beam Search

Beam search is a heuristic search algorithm that investigates a graph by extending the most promising node in a small set.

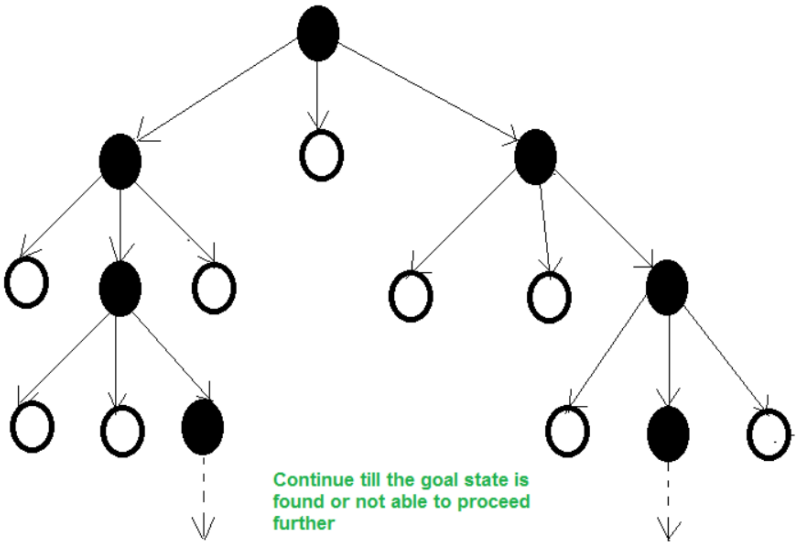
Beam search is a heuristic search approach that grows the W number of optimal nodes at each level at all times. It travels downhill exclusively from the best W nodes at each level as it develops level by level. Beam Search constructs its search tree using breadth-first search. Beam Search uses breadth-first search to build its search tree. At each level of the tree, it generates all the successors of the current level's state. However, it only assesses a W number of states at each level. Other nodes aren't taken into consideration.

The best nodes are chosen based on the heuristic cost associated with each node. W stands for the width of the beam search. If B is the branching factor, there will always be W B nodes under evaluation at every depth, but only W will be selected. When the beam width is shortened, more states are clipped.

When W = 1, the search is transformed into a hill-climbing search, with the best node always chosen from the successor nodes. If the beam width is limitless, no states are pruned, and the beam search is identified as a breadth-first search.

The beamwidth constrains the amount of memory required to finish the search, but at the expense of completeness and optimality (possibly that it will not find the best solution). This threat arises from the possibility that the desirable state has been trimmed.

Example: The following is the search tree generated by this algorithm with W = 2 and B = 3:



Beam Search

The black nodes are selected based on their heuristic values for further expansion.

Algorithm :

Input: Start & Goal States.

Local Variables: OPEN, NODE, SUCCS, W_OPEN, FOUND

Output: Yes or No (yes if the search is successfully done)

Start

Take the inputs

NODE = Root_Node & Found = False

If : Node is the Goal Node,

    Then Found = True,

Else :

    Find SUCCs of NODE if any, with its estimated cost&

    store it in OPEN List

While (Found == false & not able to proceed further), do

{

    Sort OPEN List

    Select top W elements from OPEN list and put it in

    W_OPEN list and empty the OPEN list.

    for each NODE from W_OPEN list

    {

```
        if NODE = Goal,
           then FOUND = true
        else
           Find SUCCs of NODE. If any with its estimated
           cost & Store it in OPEN list
     }
}
If FOUND = True,
   then return Yes
else
   return No
Stop
```

SUMMARY

This chapter gives the details about introduction of problem solving , search, importance of search in AI, Problem solving agent, steps in problem solving, solutions to the problem solving with examples, Uninformed and Informed Search strategies, difference between different search strategies as well as the comparison with respective to their performance evaluation.

QUESTIONS

Q1) Give the importance of Search in AI

Q2) Explain in Detail Problem Solving Agent

Q3) Differentiate Between BFS and DFS

Q4) Differentiate Between Unidirectional and Bidirectional Search

Q5) Write a Note on BFs and Uniform Cost Search

Q6) What is Heuristic Function? Explain with it's Characteristics

❄❄❄❄❄❄❄

4

LEARNING FROM EXAMPLES

Unit Structure :

4.0 OBJECTIVES

- To understand the concept of learning through different models
- To get familiar with networks, models and the best fit selection criteria for the model to get high performance

4.1 INTRODUCTION

After making observations about the outside environment, an agent learns if doing so enhances its performance on subsequent tasks. Learning can be simple, like writing down a phone number, or it can be profound, like when Albert Einstein deduced a new theory of the cosmos. This chapter will focus on one type of learning issue that appears to be limited but actually has a wide range of applications: develop a function from a set of input-output pairs that predicts the outcome for fresh inputs.

We need an agent to learn, but why? Why wouldn't the designers just programme in that enhancement from the start if the agent's design can be improved? The main causes are three. First of all, the designers are unable to foresee every scenario that an agent can encounter. For instance, a tunnel robot must memorize the layout of every new maze it meets. A programme created to forecast stock market prices for tomorrow must learn to adjust as conditions shift from a boom to a bust. This is because, second, the creators cannot foresee every change throughout time. Lastly, occasionally human programmers are unable to create a solution on their own. Even the best programmers cannot create a computer to recognise the faces of family members, for instance, unless they use learning algorithms. Most individuals are skilled at this task.

4.2 FORMS OF LEARNING

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

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

Components to be learned

The elements here can all be learned. Think of an agent who is learning how to drive a cab, for instance. The agent may pick up a condition-action rule for when to brake (component 1) every time the instructor yells, but they also pick it up every time the instructor doesn't. It can become familiar with buses by viewing numerous camera photos that are said to contain buses (2). It can discover the impact of its actions by attempting activities and tracking the outcomes, such as applying heavy braking on a slick surface (3). When passengers who have gone through a lot on the trip don't tip it, it can then learn a helpful aspect of its total utility function (4)

Representation and prior knowledge

We can use a variety of representations for agent components, such as propositional and first-order logical sentences for an agent's logical components, Bayesian networks for an agent's inferential components, etc. All of these representations have been given efficient learning methods. This chapter discusses inputs that consist of a vector of attribute values that have been factored and outputs that can either be continuous numerical values or discrete values.

There is an alternative perspective on the many learning modalities. Inductive learning is the process of inferring a general function or rule from a set of specific input-output pairings. We can also learn analytically or

deductively, which involves moving from a general rule that is understood to a new rule that is logically implied.

Feedback to learn from

The three primary types of learning are determined by three different sorts of feedback:

Unsupervised learning allows the agent to pick up patterns in the input without explicit feedback. The most typical unsupervised learning job is clustering, which involves finding groups of input samples that may be beneficial. For instance, a cab driver may gradually come to understand the difference between "good traffic days" and "poor traffic days" without ever receiving labelled instances of either from an instructor.

In reinforcement learning, the agent picks up new information from a succession of rewards or penalties. For instance, the taxi driver knows something went wrong when there is no tip left at the end of the trip. The two points for a win at the conclusion of a game of chess indicate to the agent that things went well. The agent must determine which of the earlier activities was most accountable for it before the reinforcement.

In supervised learning, the agent watches a few real-world input-output examples and picks up a function that converts input to output. In the aforementioned component 1, the percepts are the inputs, while the teacher's commands to "Brake!" or "Turn left" are the outputs. In component 2, camera photos provide the inputs, and an instructor again provides the outputs by stating, "That's a bus." The theory of braking is a function of states, braking actions, and stopping distance in feet in section 3. In this instance, the agent's perceptions can be used to determine the output value directly (post-hoc); the environment serves as the instructor.

These distinctions are not always clear in practise. We are only provided a small number of labelled examples in semi-supervised learning, and we must make the best use of a sizable pool of unlabeled data. The oracular truths we search for in the labels themselves may not even be true. Consider developing a technique to determine a person's age from a photo. By taking photographs of people and getting their ages, you can collect some labelled instances. Supervised learning is that. Nonetheless, several of the individuals actually lied about their age. The errors are systematic rather than merely random noise in the data, and finding them requires solving an unsupervised learning problem involving photos, self-reported ages, and true (unknown) ages. Thus, both noise and lack of labels create a continuum between supervised and unsupervised learning

4.3 SUPERVISED LEARNING

Supervised learning is a sort of machine learning in which the output is predicted by the machines using well-labeled training data that has been used to train the machines. The term "labelled data" refers to input data that has already been assigned the appropriate output.

In supervised learning, the training data that is given to the computers serves as the supervisor, instructing them on how to correctly predict the output. It employs the same idea that a pupil would learn under a teacher's guidance.

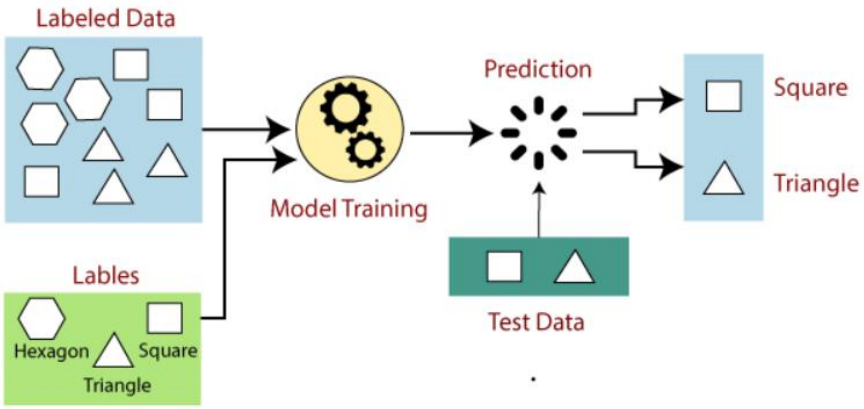
The method of supervised learning involves giving the machine learning model the right input data as well as the output data. Finding a mapping function to link the input variable (x) with the output variable is the goal of a supervised learning algorithm (y).

Supervised learning has applications in the real world, including risk assessment, image categorization, fraud detection, spam filtering, etc.

Working of Supervised Learning?

Models are trained using labelled datasets in supervised learning, where the model learns about various types of input. Following the completion of the training phase, the model is evaluated using test data (a subset of the training set), and it then makes output predictions.

The example and graphic following make it simple to understand how supervised learning operates:



Let's say we have a dataset of various forms, such as squares, rectangles, triangles, and polygons. The model must now be trained for each shape, which is the first stage.

- The given shape will be referred to as a Square if it has four sides and all of those sides are equal.
- A triangle will be designated as the provided shape if it has three sides.
- The given shape will be referred to be a hexagon if it has six equal sides.

The model's job is to recognise the shape when put to the test using the test set after training.

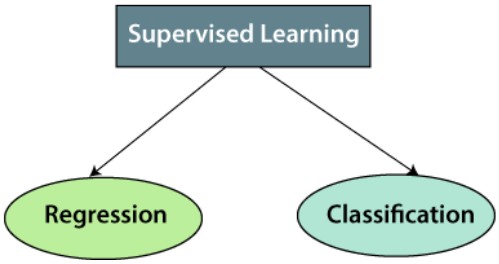
The computer has already educated on many forms, so when it encounters a new shape, it categorises it based on a number of its sides and forecasts the result.

Procedures for Supervised Learning:

- Identify the training dataset type first.
- Obtain the training data using labels.
- Create training, test, and validation datasets from the training dataset.
- Identify the training dataset's input features, which should have sufficient details to enable reliable output prediction.
- Choose the best method for the model, such as a decision tree or a support vector machine.
- Apply the algorithm to the practise data. Validation sets, a subset of training datasets, are occasionally required as control parameters.
- Use the test set to determine the model's correctness. If the model correctly predicts the outcome, then it is accurate.

Types of Supervised learning algorithms

Supervised learning can be further divided into two types of problems:



1. Regression

Regression algorithms are used if there is a relationship between the input variable and the output variable. It is used for the prediction of continuous variables, such as Weather forecasting, Market Trends, etc. Below are some popular Regression algorithms which come under supervised learning:

- Linear Regression
- Regression Trees
- Non-Linear Regression
- Bayesian Linear Regression
- Polynomial Regression

2. Classification

Classification algorithms are used when the output variable is categorical, which means there are two classes such as Yes-No, Male-Female, True-false, etc.

- Random Forest
- Decision Trees
- Logistic Regression
- Support vector Machines

Advantages

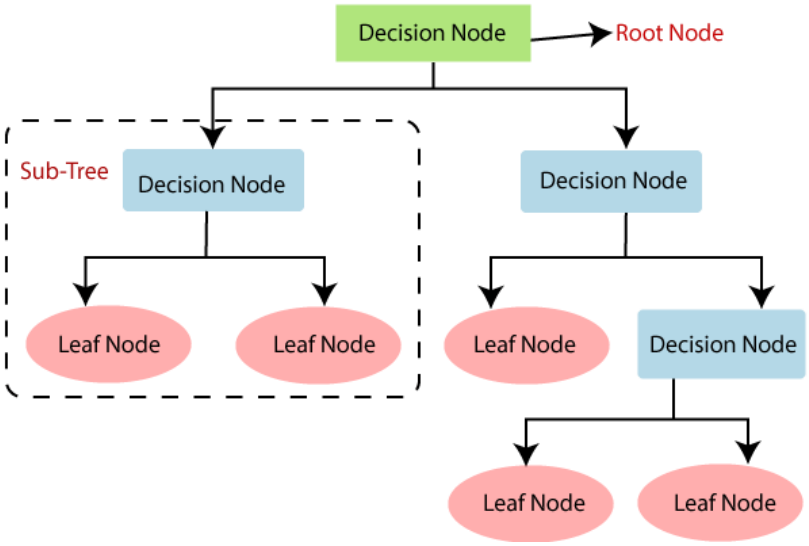
- The model can forecast the outcome based on prior experiences with the aid of supervised learning.
- With supervised learning, we can be certain of the object classes.
- We use the supervised learning model to address a variety of real-world issues, including spam filtering and fraud detection.

Disadvantages

- Models of supervised learning are inadequate for dealing with difficult tasks.
- If the test data and the training dataset are not the same, supervised learning cannot predict the right result.
- It took a long time to compute throughout training.
- In supervised learning, we require sufficient information of the object class.

4.4 LEARNING DECISION TREES

A supervised learning method called a decision tree can be used to solve classification and regression problems, but it is typically favored for doing so. It is a tree-structured classifier, where internal nodes stand in for a dataset's features, branches for the decision-making process, and each leaf node for the classification result. The Decision Node and Leaf Node are the two nodes of a decision tree. Whereas Leaf nodes are the results of decisions and do not have any more branches, Decision nodes are used to create decisions and have numerous branches. The given dataset's features are used to execute the test or make the decisions. The given dataset's features are used to execute the test or make the decisions. It is a graphical depiction for obtaining all feasible answers to a choice or problem based on predetermined conditions. It is known as a decision tree because, like a tree, it begins with the root node and grows on subsequent branches to form a structure resembling a tree. The CART algorithm, which stands for Classification and Regression Tree algorithm, is used to construct a tree. A decision tree simply asks a question, then based on the answer (Yes/No), it further split the tree into subtrees. The decision tree's general structure is shown in the diagram below:



Decision Tree Terminologies:

- Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.
- Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.
- Splitting: Splitting is the process of dividing the decision node/root node into sub-nodes according to the given conditions.
- Branch/Sub Tree: A tree formed by splitting the tree.
- Pruning: Pruning is the process of removing the unwanted branches from the tree.
- Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

Working of an algorithm:

In a decision tree, the algorithm begins at the root node and works its way up to forecast the class of the given dataset. This algorithm follows the branch and jumps to the following node by comparing the values of the root attribute with those of the record (real dataset) attribute.

The algorithm verifies the attribute value with the other sub-nodes once again for the following node before continuing. It keeps doing this until it reaches the tree's leaf node. The following algorithm can help you comprehend the entire procedure:

- **Step-1:** Begin the tree with the root node, says S, which contains the complete dataset.
- **Step-2:** Find the best attribute in the dataset using Attribute Selection Measure (ASM).

- **Step-3:** Divide the S into subsets that contains possible values for the best attributes.
- **Step-4:** Generate the decision tree node, which contains the best attribute.
- **Step-5:** Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

4.5 EVALUATING AND CHOOSING THE BEST HYPOTHESIS

We want to select the learning algorithm that best fits the common dataset when we train a model because there are many learning algorithms that may be employed to do so (training and testing dataset).

We contrast various learning algorithms in an effort to identify the optimal one. We'll look at various factors that we must consider while comparing learning algorithms.

Why use statistical techniques to assess learning algorithms?

K-fold cross-validation is a popular method for calculating the mean performance of machine learning models.

The algorithm that performs the best on average ought to outperform those that perform the worst. But what if a statistical anomaly caused the difference in average performance?

A statistical hypothesis test is used to examine whether the difference in mean performance between any two algorithms is real or not.

Learning algorithms comparison:

We are interested in finding out which learning algorithm, on average, is the most effective at teaching a given target function f. Across all the training sets of size n that were chosen from the instance distribution D, average the performance of these two techniques.

We use this formula to estimate the expected value of the difference in the errors.

$$\underset{S \subset \mathcal{D}}{E} [error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))] \quad (1)$$

Where L(S) signifies the hypothesis generated by learning technique L given a sample of training data of size S.

When comparing learning algorithms, we only have a small sample Do of data to work with.

Do can be divided into two sets: a training set So and a disjoint test set To. The test data can be used to assess the accuracy of the two hypotheses, while the training data can be used to train both LA and LB (the learning algorithms).

$$error_{T_0}(L_A(S_0)) - error_{T_0}(L_B(S_0)) \quad (2)$$

There are two major distinctions between this estimator and the quantity in Equation (1):

- To begin, we'll use error(h) to approximate error D(h).
- Second, rather than considering the anticipated value of this difference overall samples S chosen from the distribution D, we just measure the difference in errors for one training set S0.

To make the estimator in Equation better (2)

- split the data on a regular basis and take the mean of the test set errors for these individual studies and divide them into disjoint training and test sets.

1. Partition the available data D_0 into k disjoint subsets $T_1, T_2, \ldots, T_k$ of equal size, where this size is at least 30.
2. For i from 1 to k, do
   use T_i for the test set, and the remaining data for training set S_i
   - $S_i \leftarrow \{D_0 - T_i\}$
   - $h_A \leftarrow L_A(S_i)$
   - $h_B \leftarrow L_B(S_i)$
   - $\delta_i \leftarrow error_{T_i}(h_A) - error_{T_i}(h_B)$
3. Return the value $\bar{\delta}$, where

$$\bar{\delta} \equiv \frac{1}{k}\sum_{i=1}^{k}\delta_i$$

TABLE
A procedure to estimate the difference in error between two learning methods L_A and L_B. Approximate confidence intervals for this estimate are given in the text.

The $\bar{\delta}$ quantity returned by the procedure of Table can be taken as an estimate of the desired quantity from Equation 1. More appropriately, we can view $\bar{\delta}$ as an estimate of the quantity.

$$E_{S \subset D_0}[error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))] \quad (3)$$

Where S represents a random sample of size ((k-1)/k)*|D0| drawn uniformly from D0.

The approximate N% confidence interval for estimating the quantity in Equation 3 using $\bar{\delta}$ is given by

$$\bar{\delta} \pm t_{N,k-1}\ s_{\bar{\delta}} \tag{4}$$

Where tN, k-1 is a constant that plays a role analogous to that of Zn in our earlier confidence interval expressions, and where $s_{\bar{\delta}}$ is an estimate of the standard deviation of the distribution governing $\bar{\delta}$. In particular, $s_{\bar{\delta}}$ is defined as

$$s_{\bar{\delta}} \equiv \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^{k} (\delta_i - \bar{\delta})^2} \tag{5}$$

Notice the constant tN,k-1 in equation (4) has two subscripts. The first one specifies the desired confidence level, as it did for our earlier constant zN.

The second parameter called the number of degrees of freedom and usually denoted by v, is related to the number of independent random events that go into producing the value for the random variable $\bar{\delta}$.

4.6 REGRESSION AND CLASSIFICATION WITH LINEAR MODELS

4.6.1 Univariate linear regression

Univariate (or single-variable) linear regression refers to a linear regression model where we use only one independent variable *x* to learn a *linear* function that maps *x* to our dependent variable *y:*

$$y^i = \beta_0 + \beta_1 x^i + \epsilon^i$$

In the preceding equation, we have the following:

- y^i represents the *dependent* variable for the i^{th} observation
- x^i represents the single *independent* variable for the i^{th} observation
- εi represents the *error* term for the i^{th} observation
- β_0 is the intercept coefficient
- β_1 is the regression coefficient for the single independent variable

4.6.2 Multivariate linear regression

This is quite similar to the simple linear regression model we have discussed previously, but with multiple independent variables contributing to the dependent variable and hence multiple coefficients to determine and

complex computation due to the added variables. Jumping straight into the equation of multivariate linear regression,

$$Y_i = \alpha + \beta_1 x_i^{(1)} + \beta_2 x_i^{(2)} + \ldots + \beta_n x_i^{(n)}$$

Y_i is the estimate of i^{th} component of dependent variable y, where we have **n** independent variables and x_i^j denotes the i^{th} component of the j^{th} independent variable/feature. Similarly cost function is as follows,

$$E(\alpha, \beta_1, \beta_2, \ldots, \beta_n) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - Y_i)$$

where we have **m** data points in training data and y is the observed data of dependent variable. As per the formulation of the equation or the cost function, it is pretty straight forward generalization of simple linear regression. But computing the parameters is the matter of interest here.

4.6.3 Linear classification with logistic regression

Logistic regression is a supervised learning classification algorithm used to predict the probability of a target variable. The nature of target or dependent variable is dichotomous, which means there would be only two possible classes.

In simple words, the dependent variable is binary in nature having data coded as either 1 (stands for success/yes) or 0 (stands for failure/no).

Mathematically, a logistic regression model predicts P(Y=1) as a function of X. It is one of the simplest ML algorithms that can be used for various classification problems such as spam detection, Diabetes prediction, cancer detection etc.

Binary Logistic Regression model

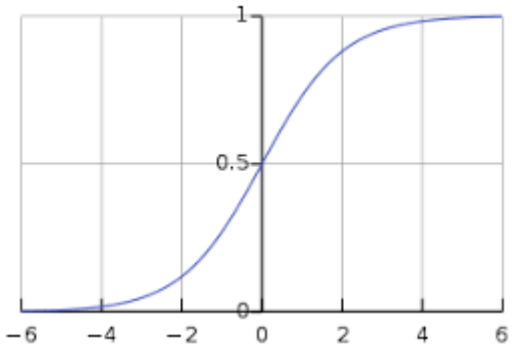
The simplest form of logistic regression is binary or binomial logistic regression in which the target or dependent variable can have only 2 possible types either 1 or 0. It allows us to model a relationship between multiple predictor variables and a binary/binomial target variable. In case of logistic regression, the linear function is basically used as an input to another function such as g in the following relation −

$$h_\theta(x) = g(\theta^T x) where 0 \leq h_\theta \leq 1$$

Here, g is the logistic or sigmoid function which can be given as follows –

$$g(z) = \frac{1}{1 + e^{-z}} where z = \theta^T x$$

To sigmoid curve can be represented with the help of following graph. We can see the values of y-axis lie between 0 and 1 and crosses the axis at 0.5.



The classes can be divided into positive or negative. The output comes under the probability of positive class if it lies between 0 and 1. For our implementation, we are interpreting the output of hypothesis function as positive if it is ≥0.5, otherwise negative.

We also need to define a loss function to measure how well the algorithm performs using the weights on functions, represented by theta as follows −

$$h=g(X\theta)$$

$$J(\theta) = \frac{1}{m}.(-y^T log(h) - (1-y)^T log(1-h))$$

Now, after defining the loss function our prime goal is to minimize the loss function. It can be done with the help of fitting the weights which means by increasing or decreasing the weights. With the help of derivatives of the loss function w.r.t each weight, we would be able to know what parameters should have high weight and what should have smaller weight.

The following gradient descent equation tells us how loss would change if we modified the parameters

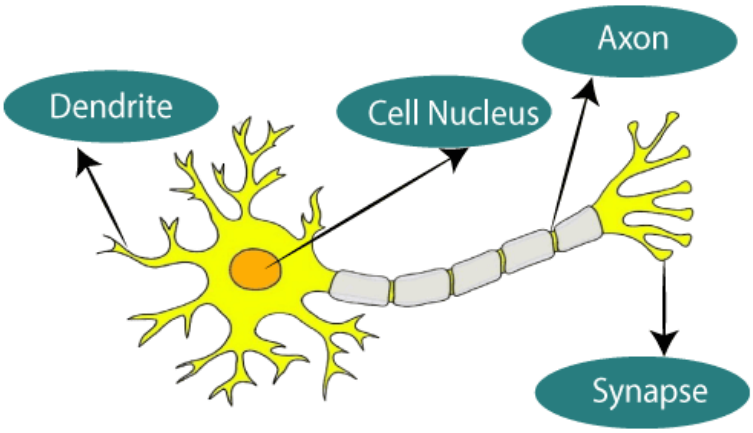
$$\frac{\delta J(\theta)}{\delta \theta_j} = \frac{1}{m} X^T (g(X\theta) - y)$$

Multinomial Logistic Regression Model

Another useful form of logistic regression is multinomial logistic regression in which the target or dependent variable can have 3 or more possible unordered types i.e. the types having no quantitative significance.

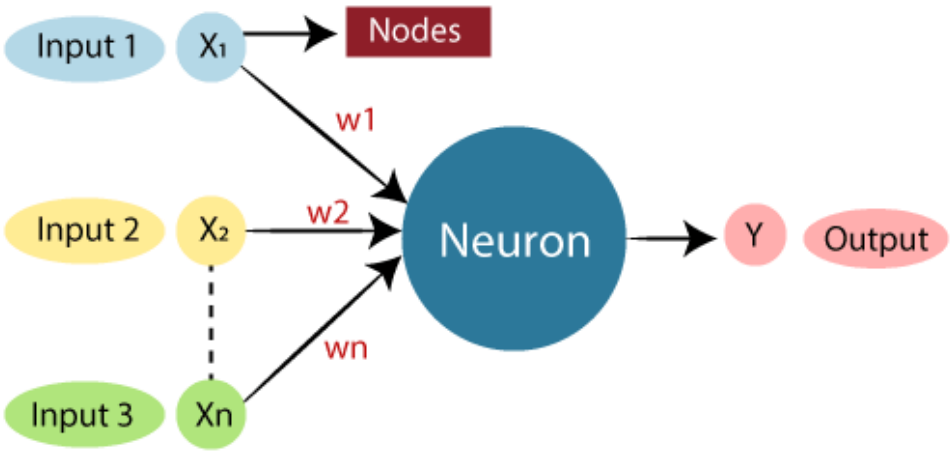
4.7 ARTIFICIAL NEURAL NETWORKS

The biological neural networks that shape the structure of the human brain are where the phrase "artificial neural network" originates. Artificial neural networks also feature neurons that are interconnected to one another in different levels of the networks, much like the human brain, which has neurons that are interconnected to one another. Nodes are the name for these neurons.



The given figure illustrates the typical diagram of Biological Neural Network.

The typical Artificial Neural Network looks something like the given figure.



Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

Relationship between Biological neural network and artificial neural network:

Biological Neural Network	**Artificial Neural Network**
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

An **Artificial Neural Network** (ANN) in the field of **Artificial intelligence** where it attempts to mimic the network of neurons makes up a human brain so that computers will have an option to understand things and make decisions in a human-like manner. The artificial neural network is

designed by programming computers to behave simply like interconnected brain cells.

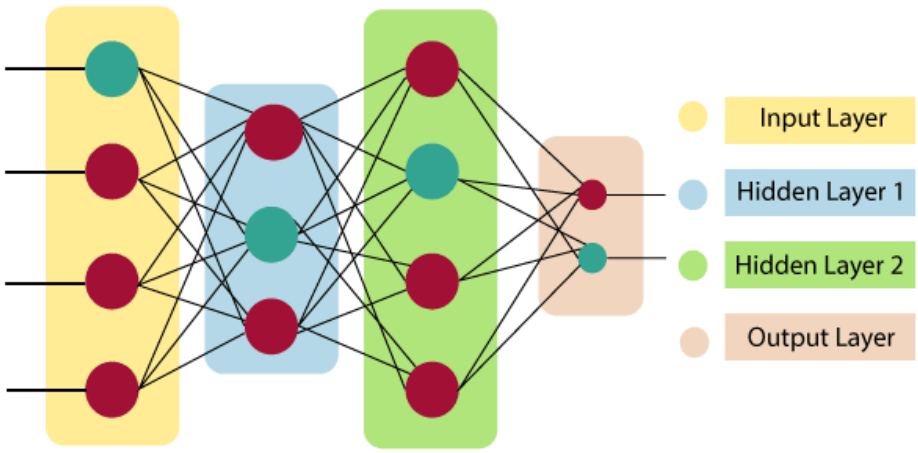
There are around 1000 billion neurons in the human brain. Each neuron has an association point somewhere in the range of 1,000 and 100,000. In the human brain, data is stored in such a manner as to be distributed, and we can extract more than one piece of this data, when necessary, from our memory parallelly. We can say that the human brain is made up of incredibly amazing parallel processors.

We can understand the artificial neural network with an example, consider an example of a digital logic gate that takes an input and gives an output. "OR" gate, which takes two inputs. If one or both the inputs are "On," then we get "On" in output. If both the inputs are "Off," then we get "Off" in output. Here the output depends upon input. Our brain does not perform the same task. The outputs to inputs relationship keep changing because of the neurons in our brain, which are "learning."

Architecture of ANN

Understanding the components of a neural network is necessary to comprehend the idea of the architecture of an artificial neural network. A vast number of artificial neurons, also known as units, are placed in a hierarchy of layers to form what is known as a neural network. Let's examine the many layers that can be found in an artificial neural network.

Artificial Neural Network primarily consists of three layers:



Input Layer:

As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

$$\sum_{i=1}^{n} Wi * Xi + b$$

It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

Advantages

1] Processing in parallel capability: Artificial neural networks have a numerical value that allows them to carry out multiple tasks at once.

2] Archiving data over the network: Traditional programming does not employ a database; instead, it stores data on the entire network. The network continues to function even if some data disappears from one location temporarily.

3] Ability to work with limited information: After ANN training, the data may still produce output even with insufficient data. The relevance of the missing data in this situation is what causes the performance loss.

4] Having a spread of memories: Determining the instances and motivating the network in accordance with the intended output by showing it these examples is crucial for ANN to be able to adapt. The network's output can be false if the event can't be represented by the network in all of its characteristics because the network's succession is directly proportional to the selected occurrences.

5] Fault-tolerant attitude: The network is fault-tolerant since expropriation of one or more ANN cells does not prevent the network from producing output.

Disadvantages

1] Guarantee of appropriate network architecture: The construction of artificial neural networks is not determined by any specific rules. With experience, trial, and error, the right network structure is achieved.

2] Unrecognized network behavior: That is the most important ANN issue. When an ANN generates a testing solution, it doesn't explain why or how. It erodes network confidence.

3] Hardware reliance: According to their structure, artificial neural networks require processors with parallel processing power. As a result, the equipment's realization is dependent.

4] Have trouble getting the network to see the problem: ANNs can process data that is numerical. Before using ANN, problems must be transformed into numerical values. The network's performance will be directly impacted by the presentation mechanism that must be decided here. It is dependent on the user's skills.

5] Unknown is the network's lifespan: The network is reduced to a particular error value, and this error value does not produce the best outcomes for us.

4.8 NONPARAMETRIC MODELS

The underlying premise of non-parametric models is that the data distribution cannot be described in terms of such a small number of parameters. However, by assuming an endless dimensional space θ, they are frequently defined. Typically, we consider θ to be a function.

Nonparametric machine learning algorithms are those that do not make any firm assumptions about the shape of the mapping function. They are allowed to learn any functional form from the training data because they are not making any assumptions.

Although keeping some ability to generalize to untried data, nonparametric approaches aim to develop the mapping function that best fits the training data. They can therefore fit a variety of practical forms.

The k-nearest neighbours approach, which generates predictions based on the k most comparable training patterns for a new data instance, is a simple nonparametric model. The method does not assume anything about the form of the mapping function other than patterns that are close are likely to have a similar output variable.

Some more examples of popular nonparametric machine learning algorithms are:

- k-Nearest Neighbors
- Decision Trees like CART and C4.5
- Support Vector Machines

Advantages

- Flexibility: Able to accommodate a wide range of useful shapes.

- Power: The underlying function is not assumed (or is assumed only loosely).
- Performance: May lead to prediction models with higher performance.

Limitations

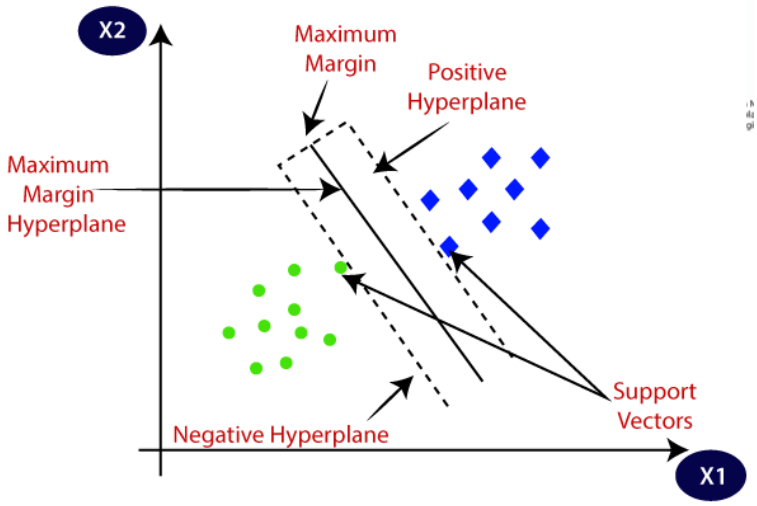
- Increased data: The mapping function estimation process calls for a large amount more training data.
- Slower: Training takes a lot longer because there are frequently more parameters to train.
- Overfitting: There is a greater chance of overfitting the training set, and it is more difficult to justify why certain predictions are made.

4.9 SUPPORT VECTOR MACHINES

One of the most well-liked supervised learning algorithms, Support Vector Machine, or SVM, is used to solve Classification and Regression problems. However, it is largely employed in Machine Learning Classification issues.

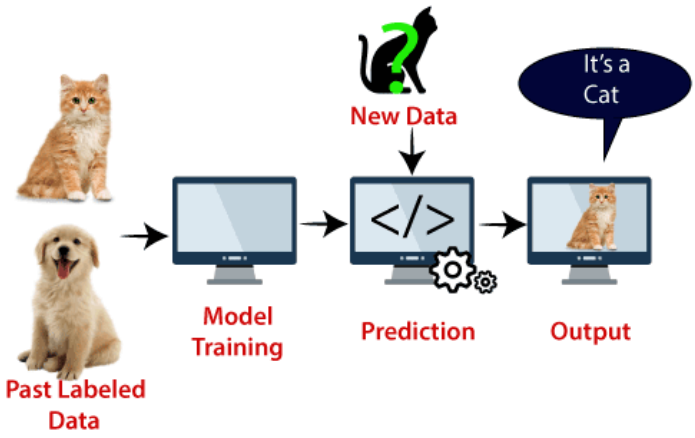
The SVM algorithm's objective is to establish the best line or decision boundary that can divide n-dimensional space into classes, allowing us to quickly classify fresh data points in the future. A hyperplane is the name given to this optimal decision boundary.

SVM selects the extreme vectors and points that aid in the creation of the hyperplane. Support vectors, which are used to represent these extreme instances, form the basis for the SVM method. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



Example: The example we used for the KNN classifier can be utilized to understand SVM. If we want a model that can correctly distinguish between a cat and a dog, let's say we observe an unusual cat that also resembles a

dog. We can build such a model by utilizing the SVM algorithm. Prior to testing it with this weird animal, we will first train our model with several photographs of cats and dogs so that it can become familiar with the various attributes of cats and dogs. As a result, the extreme cases of cats and dogs will be seen by the support vector when it draws a judgement border between these two sets of data (cat and dog). On the basis of the support vectors, it will classify it as a cat. Consider the below diagram:



SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.

Types of SVM: SVM are categorized into following two types-

- Linear SVM: Linear SVM is used for data that can be divided into two classes using a single straight line. This type of data is called linearly separable data, and the classifier employed is known as a Linear SVM classifier.
- Non-linear SVM: Non-Linear SVM is used for non-linearly separated data. If a dataset cannot be classified using a straight line, it is considered non-linear data, and the classifier employed is referred to as a Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

Hyperplane:

In n-dimensional space, there may be several lines or decision boundaries used to divide classes; however, the optimal decision boundary for classifying the data points must be identified. The hyperplane of SVM is a name for this optimal boundary.

The dataset's features determine the hyperplane's dimensions, therefore if there are just two features (as in the example image), the hyperplane will be a straight line. Moreover, if there are three features, the hyperplane will only have two dimensions.

We always build a hyperplane with a maximum margin, or the greatest possible separation between the data points.

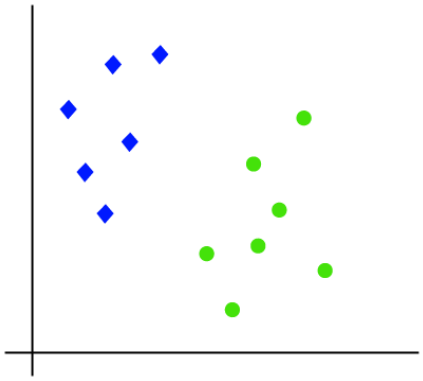
Support Vectors:

Support vectors are the data points or vectors that are closest to the hyperplane and have the greatest influence on where the hyperplane is located. These vectors are called support vectors because they support the hyperplane.

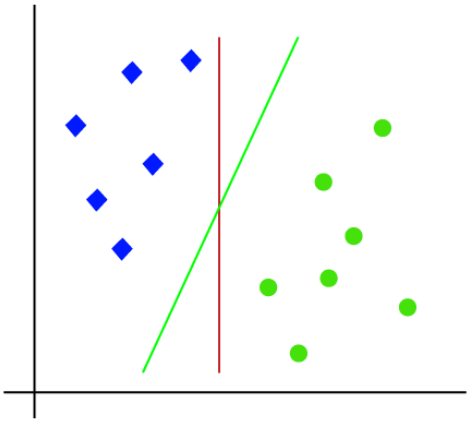
Working of SVM:

1] Linear SVM

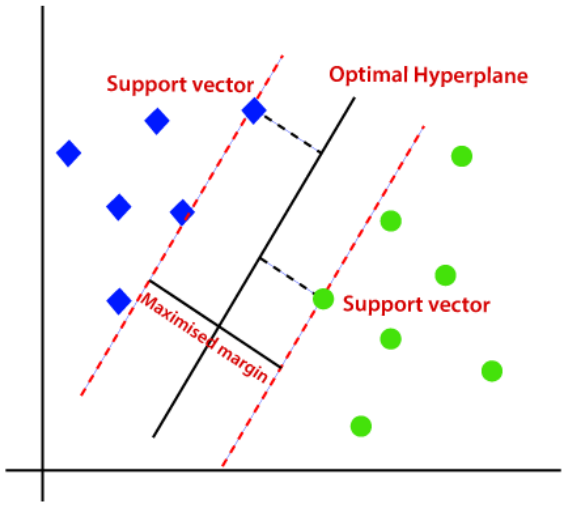
By presenting an example, the SVM algorithm's operation can be better understood. Consider a dataset with two tags (green and blue), two features (x1 and x2), and two tags. We need a classifier that can identify whether the pair of coordinates (x1, x2) is blue or green. Consider the image below:



So, as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:

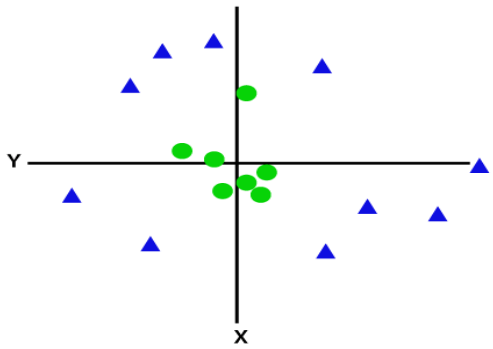


Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a hyperplane. SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called as margin. And the goal of SVM is to maximize this margin. The hyperplane with maximum margin is called the optimal hyperplane.



2] Non-Linear SVM

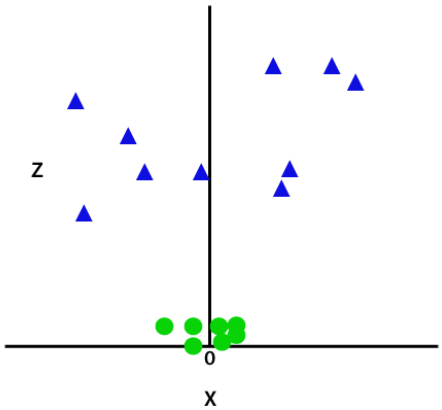
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we annot draw a single straight line. Consider the below image:



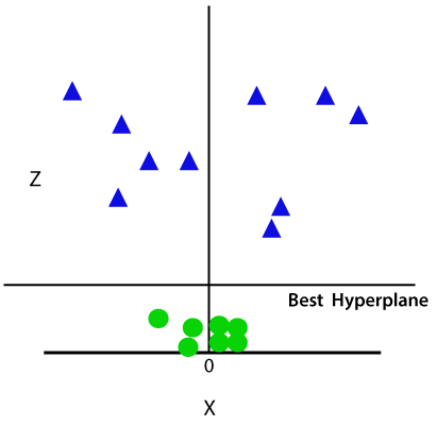
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:

$$z=x^2 +y^2$$

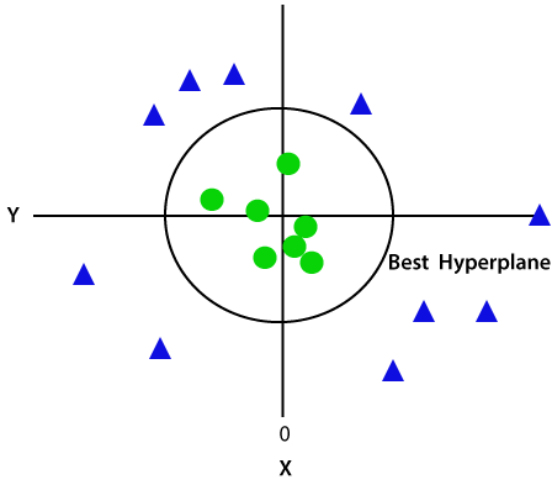
By adding the third dimension, the sample space will become as below image:



So now, SVM will divide the datasets into classes in the following way. Consider the below image:



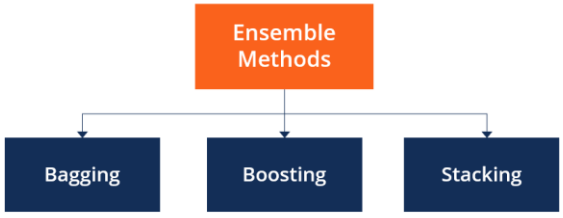
Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as:



Hence, we get a circumference of radius 1 in case of non-linear data.

4.10 ENSEMBLE LEARNING

A machine learning technique called ensemble techniques combines multiple base models to create a single, ideal predictive model. By mixing numerous models rather than relying just on one, ensemble approaches seek to increase the accuracy of findings in models. The integrated models considerably improve the results' accuracy. Due of this, ensemble approaches in machine learning have gained prominence.



Categories of ensemble methods

Sequential ensemble techniques and parallel ensemble techniques are the two main categories into which ensemble methods belong. Base learners are produced via sequential ensemble approaches, such as adaptive boosting (AdaBoost). The dependency between the base learners is encouraged by their consecutive generation. The model's performance is then enhanced by giving previously misrepresented learners more weight.

Base learners are created in a parallel fashion, such as random forest, in parallel ensemble approaches. To promote independence among the basis learners, parallel techniques make use of parallel generations of base learners. The mistake resulting from the use of averages is greatly decreased by the independence of base learners.

The majority of ensemble techniques only use one algorithm for base learning, which makes all base learners homogeneous. Base learners who have comparable traits and are of the same type are referred to as homogenous base learners. Some approaches create heterogeneous ensembles by using heterogeneous base learners. Many sorts of learners make up heterogeneous base learners.

Main types of ensemble methods

1] **Bagging :** Bootstrap aggregating is commonly used in classification and regression, and also known as bagging. Using decision trees, it improves the models' accuracy, greatly reducing variation. Many prediction models struggle with overfitting, which is eliminated by reducing variation and improving accuracy.

Bootstrapping and aggregation are the two categories under which bagging is categorized. Bootstrapping is a sampling strategy where samples are taken utilizing the replacement procedure from the entire population (set). The sampling with replacement method aids in the randomization of the selection process. The process is finished by applying the base learning algorithm to the samples.

In bagging, aggregation is used to include all potential outcomes of the prediction and randomize the result. Predictions made without aggregation won't be accurate because all possible outcomes won't be taken into account. As a result, the aggregate is based either on all of the results from the predictive models or on the probability bootstrapping techniques.

Bagging is useful because it creates a single strong learner that is more stable than individual weak base learners. Moreover, it gets rid of any variance, which lessens overfitting in models. The computational cost of bagging is one of its drawbacks. Hence, ignoring the correct bagging technique can result in higher bias in models.

2] **Boosting:** Boosting is an ensemble strategy that improves future predictions by learning from previous predictor errors. The method

greatly increases model predictability by combining numerous weak base learners into one strong learner. Boosting works by placing weak learners in a sequential order so that they can learn from the subsequent learner to improve their predictive models.

There are many different types of boosting, such as gradient boosting, Adaptive Boosting (AdaBoost), and XGBoost (Extreme Gradient Boosting). AdaBoost employs weak learners in the form of decision trees, the majority of which include a single split known as a decision stump. The primary decision stump in AdaBoost consists of observations with equal weights.

Gradient boosting increases the ensemble's predictors in a progressive manner, allowing earlier forecasters to correct later ones, improving the model's accuracy. To offset the consequences of errors in the earlier models, new predictors are fitted. The gradient booster can identify and address issues with learners' predictions thanks to the gradient of descent.

Decision trees with boosted gradients are used in XGBoost, which offers faster performance. It largely depends on the goal model's efficiency and effectiveness in terms of computing. Gradient boosted machines must be implemented slowly since model training must proceed sequentially.

3] **Stacking:** Another ensemble method called stacking is sometimes known as layered generalization. This method works by allowing a training algorithm to combine the predictions of numerous different learning algorithms that are similar. Regression, density estimations, distance learning, and classifications have all effectively used stacking. It can also be used to gauge the amount of inaccuracy that occurs when bagging.

4.11 PRACTICAL MACHINE LEARNING

We look at two features of real-world machine learning in this section. Finding algorithms that can recognize handwritten digits and extracting every last bit of predicted performance from them constitutes the first step. The second includes anything but, namely highlighting that data collection, cleansing, and representation might be at least as crucial as algorithm design

1] Case study: Handwritten digit recognition

With numerous applications, such as automatic mail sorting by postal code, automated reading of cheques and tax returns, and data entry for handheld computers, reading handwritten digits poses a significant challenge. The field has advanced quickly, in part because to improved learning algorithms and in part due to the accessibility of richer training sets. The National Institute of Science and Technology (NIST) of the United States has preserved a database of 60,000 labelled numbers, each measuring 20 by 20 pixels (or 400 pixels) and having 8-bit grayscale values. It is now considered

to be one of the common benchmark issues for contrasting new learning algorithms. Following figure displays a few sample digits.

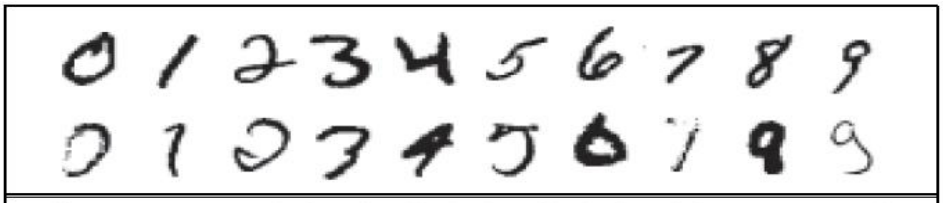


Figure Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

There have been many different learning strategies tested. The 3-nearest-neighbor classifier is one of the first and most likely the simplest. It also has the benefit of requiring no training time. Nevertheless, because it is a memory-based technique, it has a poor run-time performance and must keep all 60,000 photos. A test error rate of 2.4% was attained.

For this issue, a single-hidden-layer neural network with 400 input units (one per pixel) and 10 output units was created (one per class). It was discovered through cross-validation that about 300 concealed units performed the best. There were 123,300 weights in total with full linkages between levels. A 1.6% error rate was attained by this network.

A series of specialized neural networks called LeNet were devised to take advantage of the structure of the problem—that the input consists of pixels in a two–dimensional array, and that small changes in the position or slant of an image are unimportant. Each network had an input layer of 32 × 32 units, onto which the 20 × 20 pixels were centered so that each input unit is presented with a local neighborhood of pixels. This was followed by three layers of hidden units. Each layer consisted of several planes of n × n arrays, where n is smaller than the previous layer so that the network is down-sampling the input, and where the weights of every unit in a plane are constrained to be identical, so that the plane is acting as a feature detector: it can pick out a feature such as a long vertical line or a short semi-circular arc. The output layer had 10 units. Many versions of this architecture were tried; a representative one had hidden layers with 768, 192, and 30 units, respectively. The training set was augmented by applying affine transformations to the actual inputs: shifting, slightly rotating, and scaling the images. (Of course, the transformations have to be small, or else a 6 will be transformed into a 9!) The best error rate achieved by LeNet was 0.9%.

A boosted neural network comprised three copies of the LeNet architecture, the third of which was trained on patterns for which the first two disagreed and the second of which was trained on a mixture of patterns that the first one got 50% wrong. The three nets followed the majority decision while testing. The rate of test mistake was 0.7%.

1.1% error rate was attained using a support vector machine with 25,000 support vectors. It is notable that despite requiring essentially no thought or iterated experimentation on the side of the developer, the SVM strategy, like the straightforward nearest neighbour approach, was nevertheless able to match the performance of LeNet, which had undergone years of research.

A virtual support vector machine starts with a regular SVM and then improves it with a technique that is designed to take advantage of the structure of the problem. Instead of allowing products of all pixel pairs, this approach concentrates on kernels formed from pairs of nearby pixels. It also augments the training set with transformations of the examples, just as LeNet did. A virtual SVM achieved the best error rate recorded to date, 0.56%. Shape matching is a technique from computer vision used to align corresponding parts of two different images of objects (Belongie et al., 2002). The idea is to pick out a set of points in each of the two images, and then compute, for each point in the first image, which point in the second image it corresponds to. From this alignment, we then compute a transformation between the images. The transformation gives us a measure of the distance between the images. This distance measure is better motivated than just counting the number of differing pixels, and it turns out that a 3–nearest neighbor algorithm using this distance measure performs very well. Training on only 20,000 of the 60,000 digits, and using 100 sample points per image extracted from a Canny edge detector, a shape matching classifier achieved 0.63% test error.

On this issue, it is predicted that human error rates will be around 0.2%. Because humans have not been subjected to as many tests as machine learning algorithms, this statistic is somewhat shaky. Human errors accounted for 2.5% of a similar data set of numbers from the USPS.

The error rates, run-time efficiency, memory needs, and training time for the seven algorithms we've covered are all summarized in the following graph. It also introduces a new metric, the proportion of rejected digits needed to achieve a 0.5% error rate. For example, if the SVM is allowed to reject 1.8% of the inputs that is, pass them on for someone else to make the final judgment then its error rate on the remaining 98.2% of the inputs is reduced from 1.1% to 0.5%. The following table summarizes the error rate and some of the other characteristics of the techniques we have discussed

	3 NN	300 Hidden	LeNet	Boosted LeNet	SVM	Virtual SVM	Shape Match
Error rate (pct.)	2.4	1.6	0.9	0.7	1.1	0.56	0.63
Run time (millisec/digit)	1000	10	30	50	2000	200	
Memory requirements (Mbyte)	12	.49	.012	.21	11		
Training time (days)	0	7	14	30	10		
% rejected to reach 0.5% error	8.1	3.2	1.8	0.5	1.8		

2] Case study: Word senses and house prices

To convey our concepts in a book, we must work with basic, toy data a small data set, typically in two dimensions. Yet, the data set is typically big, complex, and chaotic in machine learning applications that are used in the real world. The analyst must go out and gather the proper data; it is not provided to him or her as a readymade set of (x, y) values. There is a task that needs to be completed, and the majority of the technical challenge is determining which data are required to complete the goal; a minor portion is selecting and putting into practice the best machine learning technique to

process the data. Following figure shows a typical real-world example, comparing five learning algorithms on the task of word-sense classification (given a sentence such as "The bank folded," classify the word "bank" as "money-bank" or "river-bank").

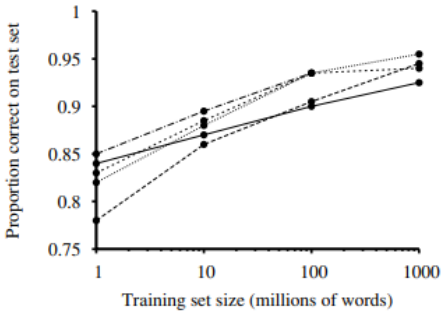


Figure Learning curves for five learning algorithms on a common task. Note that there appears to be more room for improvement in the horizontal direction (more training data) than in the vertical direction (different machine learning algorithm). Adapted from Banko and Brill (2001).

It's important to note that machine learning researchers have primarily concentrated on the vertical direction: Is it possible to create a new learning algorithm that outperforms previously published algorithms on a typical training set of 1 million words? The graph, however, demonstrates that there is greater space for advancement in the horizontal direction: rather than creating a new algorithm, all I need to do is collect 10 million training words; even the worst algorithm at 10 million words performs better than the best algorithm at 1 million. The curves continue to grow as we collect more data, dwarfing the variations in techniques.

SUMMARY

The focus of this chapter has been on learning functions inductively from examples. These were the main ideas:

- Depending on the agent's nature, the component that needs to be improved, and the feedback that is provided, learning can take many different forms.
- If the feedback at hand offers the right response for hypothetical inputs, then the learning issue is referred to as supervised learning. The assignment is to learn the y = h (x). Learning a continuous function is known as regression, while learning a discrete-valued function is known as classification.
- Any Boolean functions can be represented using decision trees. The information-gain heuristic offers a quick way to locate a straightforward, reliable decision tree.
- The learning curve, which displays the prediction accuracy on the test set as a function of the training-set size, is used to evaluate the performance of a learning algorithm.

- One often used model is linear regression. A gradient descent search can be used to find the ideal linear regression model parameters, or they can be calculated precisely.
- The hard threshold of the perceptron is replaced by a soft threshold determined by a logistic function in logistic regression. Even with noisy data that are not linearly separable, gradient descent performs well.
- Neural networks use a network of linear-threshold units to express complex nonlinear functions.
- Instead, then attempting to initially summarize the data with a few parameters, nonparametric models use all the data to make each prediction.
- machines that support vectors locate linear separators with the highest margin to enhance the classifier's generalization performance.
- Ensemble techniques, like boosting, frequently outperform standalone techniques.

LIST OF REFERENCES

1. Artificial Intelligence: A Modern Approach, Stuart Russell and Peter Norvig,3rd Edition, Pearson, 2010
2. Artificial Intelligence: Foundations of Computational Agents, David L Poole, Alan K. Mackworth, 2nd Edition, Cambridge University Press ,2017
3. Artificial Intelligence, Kevin Knight and Elaine Rich, 3rd Edition, 2017
4. The Elements of Statistical Learning, Trevor Hastie, Robert Tibshirani and Jerome Friedman, Springer, 2013

UNIT END EXERCISES

1] Explain the forms of Learning.

2] What is Supervised Learning?

3] Describe the concept of Learning Decision Trees.

4] How will you evaluate and choose the best hypothesis.

5] Explain Regression and Classification with Linear Models.

6] Describe Artificial Neural Networks.

7] Write a note on Nonparametric Models.

8] Explain Support Vector Machines.

9] Describe Ensemble Learning.

10] Illustrate the case studies of practical Machine Learning.

❄❄❄❄❄❄❄

5

LEARNING PROBABILISTIC MODELS

Unit Structure :

5.0 OBJECTIVES

A learner will about:

- the probability theory used for handling uncertainties
- the concepts like bayes theorem and other probability methods
- supervised and unsupervised learning
- some advanced algorithms like EM with Bayesian approach

5.1 INTRODUCTION

Agents can handle uncertainty by using the methods of probability and decision theory, but first they must learn their probabilistic theories of the world from experience. This chapter explains how they can do that, by formulating the learning task itself as a process of probabilistic inference. We will see that a Bayesian view of learning is extremely powerful, providing general solutions to the problems of noise, overfitting, and optimal prediction. It also takes into account the fact that a less-than-omniscient agent can never be certain about which theory of the world is correct, yet must still make decisions by using some theory of the world.

Here, the data are evidence—that is, instantiations of some or all of the random variables describing the domain. The hypotheses in this chapter are probabilistic theories of how the domain works, including logical theories as a special case. Consider a simple example. Our favorite Surprise candy comes in two flavors: cherry (yum) and lime (ugh). The manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

h1: 100% cherry,

h2: 75% cherry + 25% lime,

h3: 50% cherry + 50% lime,

h4: 25% cherry + 75% lime,

h5: 100% lime .

Given a new bag of candy, the random variable H (for hypothesis) denotes the type of the bag, with possible values h1 through h5. H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed—D_1, D_2, ..., D_N , where each Di is a random variable with possible values cherry and lime. The basic task faced by the agent is to predict the flavor of the next piece of candy.1 Despite its apparent triviality, this scenario serves to introduce many of the major issues. The agent really does need to infer a theory of its world, albeit a very simple one.

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using all the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis. In this way, learning is reduced to probabilistic inference. Let D represent all the data, with observed value d; then the probability of each hypothesis is obtained by Bayes' rule:

$$P(h_i \mid d) = \alpha P(d \mid h_i) P(h_i) . \qquad (5.1)$$

Now, suppose we want to make a prediction about an unknown quantity X. Then we have

$$\mathbf{P}(X \mid \mathbf{d}) = \sum_i \mathbf{P}(X \mid \mathbf{d}, h_i)\mathbf{P}(h_i \mid \mathbf{d}) = \sum_i \mathbf{P}(X \mid h_i) P(h_i \mid \mathbf{d}) , \qquad (5.2)$$

where we have assumed that each hypothesis determines a probability distribution over X.

This equation shows that predictions are weighted averages over the predictions of the individual hypotheses. The hypotheses themselves are essentially "intermediaries" between the raw data and the predictions. The key quantities in the Bayesian approach are the hypothesis prior, P(hi), and the likelihood of the data under each hypothesis, P(d | hi). For our candy example, we will assume for the time being that the prior distribution over h1,...,h5 is given by 0.1, 0.2, 0.4, 0.2, 0.1, as advertised by the anufacturer. The likelihood of the data is calculated under the assumption that the observations are i.i.d.

$$P(\mathbf{d} \mid h_i) = \prod_j P(d_j \mid h_i) \ . \tag{5.3}$$

For example, suppose the bag is really an all-lime bag (h5) and the first 10 candies are all lime; then P(d | h3) is 0.510, because half the candies in an h3 bag are lime.2 Figure 5.1(a) shows how the posterior probabilities of the five hypotheses change as the sequence of 10lime candies is observed. Notice that the probabilities start out at their prior values, so h3 is initially the most likely choice and remains so after 1 lime candy is unwrapped. After 2 lime candies are unwrapped, h4 is most likely; after 3 or more, h5 (the dreaded all-lime bag) is the most likely. After 10 in a row, we are fairly certain of our fate. Figure 5.1(b) shows the predicted probability that the next candy is lime, based on Equation (5.2). As we would expect, it increases monotonically toward 1.

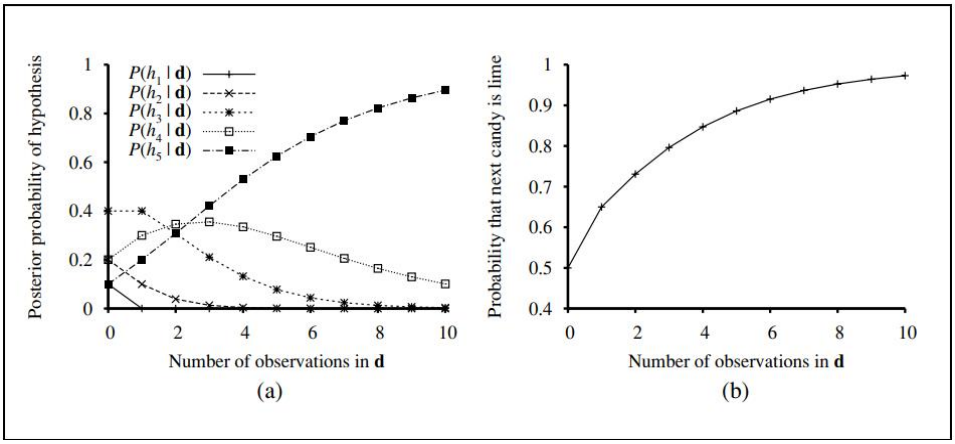


Figure 5.1. (a) Posterior probabilities P(hi | d1,...,dN) from Equation (20.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy.

(b) Bayesian prediction P(dN+1 = lime | d1,...,dN)

The example shows that the Bayesian prediction eventually agrees with the true hypothesis. This is characteristic of Bayesian learning. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will, under certain technical conditions, eventually vanish. This happens simply because the probability of generating "uncharacteristic" data indefinitely is vanishingly small. More important, the Bayesian prediction is optimal, whether the data set be small or large. Given the hypothesis prior, any other prediction is expected to be correct less often.

A very common approximation—one that is usually adopted in science—is to make predictions based on a single most probable hypothesis—that is, an hi that maximizes P(h_i | d). This is often called a maximum a posteriori or MAP (pronounced "em-ay-pee") hypothesis. Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that P(X | d) ≈ P(X | h_{MAP}). In our candy example, h_{MAP} = h5 after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with probability 1.0—a much more dangerous prediction than the Bayesian

prediction of 0.8 shown in Figure 5.1(b). As more data arrive, the MAP and Bayesian predictions become closer, because the competitors to the MAP hypothesis become less and less probable.

Although our example doesn't show it, finding MAP hypotheses is often much easier than Bayesian learning, because it requires solving an optimization problem instead of a large summation (or integration) problem. We will see examples of this later in the chapter.

In both Bayesian learning and MAP learning, the hypothesis prior $P(h_i)$ plays an important role. We saw in pervious chapter that overfitting can occur when the hypothesis space is too expressive, so that it contains many hypotheses that fit the data set well. Rather than placing an arbitrary limit on the hypotheses to be considered, Bayesian and MAP learning methods use the prior to penalize complexity. Typically, more complex hypotheses have a lower prior probability—in part because there are usually many more complex hypotheses than simple hypotheses. On the other hand, more complex hypotheses have a greater capacity to fit the data. (In the extreme case, a lookup table can reproduce the data exactly with probability 1.) Hence, the hypothesis prior embodies a tradeoff between the complexity of a hypothesis and its degree of fit to the data.

We can see the effect of this tradeoff most clearly in the logical case, where H contains only deterministic hypotheses. In that case, $P(d \mid h_i)$ is 1 if h_i is consistent and 0 otherwise. Looking at Equation , we see that h_{MAP} will then be the simplest logical theory that is consistent with the data. Therefore, maximum a posteriori learning provides a natural embodiment of Ockham's razor. Another insight into the tradeoff between complexity and degree of fit is obtained by taking the logarithm of Equation(5.1). Choosing h_{MAP} to maximize $P(d \mid h_i)P(h_i)$ is equivalent to minimizing

$$-\log_2 P(d \mid h_i) - \log_2 P(h_i) .$$

Using the connection between information encoding and probability that we introduced in Chapter, we see that the $-\log_2 P(hi)$ term equals the number of bits required to specify the hypothesis hi. Furthermore, $-\log_2 P(d \mid h_i)$ is the additional number of bits required to specify the data, given the hypothesis. (To see this, consider that no bits are required if the hypothesis predicts the data exactly—as with h5 and the string of lime candies—and $\log_2 1=0$.) Hence, MAP learning is choosing the hypothesis that provides maximum compression of the data. The same task is addressed more directly by the minimum description length, or MDL, learning method. Whereas MAP learning expresses simplicity by assigning higher probabilities to simpler hypotheses, MDL expresses it directly by counting the bits in a binary encoding of the hypotheses and data.

A final simplification is provided by assuming a uniform prior over the space of hypotheses. In that case, MAP learning reduces to choosing an h_i that maximizes $P(d \mid h_i)$.This is called a maximum-likelihood (ML) hypothesis, hML. Maximum-likelihood learning is very common in statistics, a discipline in which many researchers distrust the subjective nature of hypothesis priors. It is a reasonable approach when there is no

reason to prefer one hypothesis over another a priori—for example, when all hypotheses are equally complex. It provides a good approximation to Bayesian and MAP learning when the data set is large, because the data swamps the prior distribution over hypotheses, but it has problems (as we shall see) with small data sets.

5.2 LEARNING WITH COMPLETE DATA

The general task of learning a probability model, given data that are assumed to be generated from that model, is called density estimation. (The term applied originally to probability density functions for continuous variables, but is used now for discrete distributions too.) This section covers the simplest case, where we have complete data. Data are complete when each data point contains values for every variable in the probability model being learned. We focus on parameter learning—finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. We will also look briefly at the problem of learning structure and at nonparametric density estimation.

5.2.1. Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime–cherry proportions are completely unknown; the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The parameter in this case, which we call θ, is the proportion of cherry candies, and the hypothesis is h_θ. (The proportion of limes is just $1 - \theta$.) If we assume that all proportions are equally likely a priori, then a maximum likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, Flavor (the flavor of a randomly chosen candy from the bag). It has values cherry and lime, where the probability of cherry is θ. Now suppose we unwrap N candies, of which c are cherries and $l = N - c$ are limes. According to Equation , the likelihood of this particular data set is

$$P(\mathbf{d} \mid h_\theta) = \prod_{j=1}^{N} P(d_j \mid h_\theta) = \theta^c \cdot (1-\theta)^\ell \, .$$

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. The same value is obtained by maximizing the log likelihood,

$$L(\mathbf{d} \mid h_\theta) = \log P(\mathbf{d} \mid h_\theta) = \sum_{j=1}^{N} \log P(d_j \mid h_\theta) = c \log \theta + \ell \log(1-\theta) \, .$$

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of θ, we differentiate L with respect to θ and set the resulting expression to zero:

$$\frac{dL(\mathbf{d} \mid h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \qquad \Rightarrow \qquad \theta = \frac{c}{c+\ell} = \frac{c}{N} \, .$$

In English, then, the maximum-likelihood hypothesis h_{ML} asserts that the actual proportion of cherries in the bag is equal to the observed proportion in the candies unwrapped so far! It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning, a method with broad applicability:

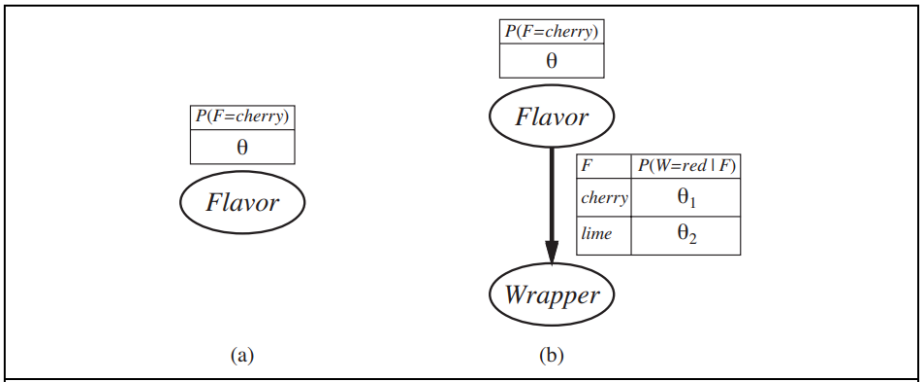


Figure 5.2. Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

1. Write down an expression for the likelihood of the data as a function of the parameter(s).

2. Write down the derivative of the log likelihood with respect to each parameter.

3. Find the parameter values such that the derivatives are zero.

The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Chapter. The example also illustrates a significant problem with maximum-likelihood learning in general: when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum-likelihood hypothesis assigns zero probability to those events. Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of 0.

Let us look at another example. Suppose this new candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green. The Wrapper for each candy is selected probabilistically, according to some unknown conditional distribution, depending on the flavor. The corresponding probability model is shown in Figure. Notice that it has three parameters: θ, θ1, and θ2. With these parameters, the likelihood of seeing, say, a cherry candy in a green wrapper can be obtained from the standard semantics for Bayesian networks.

$$
\begin{aligned}
&P(Flavor = cherry, Wrapper = green \mid h_{\theta,\theta_1,\theta_2}) \\
&= P(Flavor = cherry \mid h_{\theta,\theta_1,\theta_2}) P(Wrapper = green \mid Flavor = cherry, h_{\theta,\theta_1,\theta_2}) \\
&= \theta \cdot (1 - \theta_1) \, .
\end{aligned}
$$

Now we unwrap N candies, of which c are cherries and are limes. The wrapper counts are as follows: r_c of the cherries have red wrappers and g_c have green, while rl of the limes have red and gl have green. The likelihood of the data is given by

$$
\boldsymbol{P(\mathbf{d} \mid h_{\theta,\theta_1,\theta_2}) = \theta^c (1 - \theta)^\ell \cdot \theta_1^{r_c} (1 - \theta_1)^{g_c} \cdot \theta_2^{r_\ell} (1 - \theta_2)^{g_\ell} \, .}
$$

This looks pretty horrible, but taking logarithms helps:

$$
L = [c \log \theta + \ell \log(1 - \theta)] + [r_c \log \theta_1 + g_c \log(1 - \theta_1)] + [r_\ell \log \theta_2 + g_\ell \log(1 - \theta_2)] \, .
$$

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set them to zero, we get three independent equations, each containing just one parameter:

$$
\begin{array}{lcll}
\frac{\partial L}{\partial \theta} & = & \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 & \Rightarrow \quad \theta = \frac{c}{c+\ell} \\
\frac{\partial L}{\partial \theta_1} & = & \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 & \Rightarrow \quad \theta_1 = \frac{r_c}{r_c+g_c} \\
\frac{\partial L}{\partial \theta_2} & = & \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 & \Rightarrow \quad \theta_2 = \frac{r_\ell}{r_\ell+g_\ell} \, .
\end{array}
$$

The solution for θ is the same as before. The solution for θ_1, the probability that a cherry candy has a red wrapper, is the observed fraction of cherry candies with red wrappers, and similarly for θ_2.

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables. The most important point is that, with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter.

The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

5.2.3. Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the naive Bayes model first introduced on. In this model, the “class” variable C (which is to be predicted) is the root and the “attribute” variables Xi are the leaves. The model is “naive” because it assumes that the attributes are conditionally independent of each other, given the class. (The model in Figure is a naive Bayes model with class Flavor and just one attribute, Wrapper.) Assuming Boolean variables, the parameters are

$$\theta = P(C = true), \theta_{i1} = P(X_i = true \mid C = true), \theta_{i2} = P(X_i = true \mid C = false).$$

The maximum-likelihood parameter values are found in exactly the same way as for Figure. Once the model has been trained in this way, it can be used to classify new examples for which the class variable C is unobserved. With observed attribute values $x_1,...,x_n$, the probability of each class is given by

$$\mathbf{P}(C \mid x_1, \ldots, x_n) = \alpha \, \mathbf{P}(C) \prod_i \mathbf{P}(x_i \mid C) \, .$$

A deterministic prediction can be obtained by choosing the most likely class. Figure 5.3 shows the learning curve for this method when it is applied to the restaurant problem. The method learns fairly well but not as well as decision-tree learning; this is presumably because the true hypothesis—which is a decision tree—is not representable exactly using a naive Bayes model. Naive Bayes learning turns out to do surprisingly well in a wide range of applications; the boosted version is one of the most effective

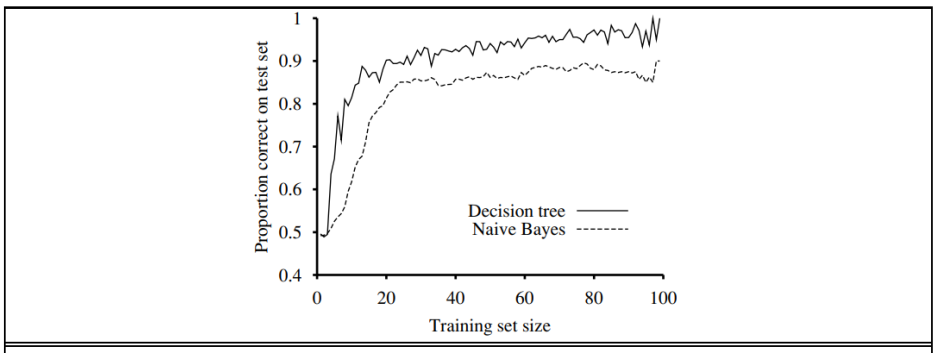


Figure 5.3: The learning curve for naive Bayes learning applied to the restaurant problem; the learning curve for decision-tree learning is shown for comparison. general-purpose learning algorithms. Naive Bayes learning scales well to very large problems: with n Boolean attributes, there are just 2n + 1 parameters, and no search is required to find h_{ML}, the maximum-likelihood naive Bayes hypothesis. Finally, naive Bayes learning systems have no difficulty with noisy or missing data and can give probabilistic predictions when appropriate.

5.2.4. Maximum-likelihood parameter learning: Continuous models

Continuous probability models such as the linear Gaussian model were introduced in previous Section Because continuous variables are ubiquitous in real-world applications, it is important to know how to learn the parameters of continuous models from data. The principles for maximum-likelihood learning are identical in the continuous and discrete cases. Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated as follows:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \, .$$

The parameters of this model are the mean μ and the standard deviation σ. (Notice that the normalizing "constant" depends on σ, so we cannot ignore it.) Let the observed values be $x_1,...,x_N$. Then the log likelihood is

$$L = \sum_{j=1}^{N} \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log\sqrt{2\pi} - \log\sigma) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2} .$$

Setting the derivatives to zero as usual, we obtain

$$\begin{aligned} \frac{\partial L}{\partial \mu} &= -\frac{1}{\sigma^2}\sum_{j=1}^{N}(x_j-\mu) = 0 \quad &\Rightarrow \quad \mu = \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{j=1}^{N}(x_j-\mu)^2 = 0 \quad &\Rightarrow \quad \sigma = \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}} . \end{aligned} \tag{5.4}$$

That is, the maximum-likelihood value of the mean is the sample average and the maximum likelihood value of the standard deviation is the square root of the sample variance. Again, these are comforting results that confirm "commonsense" practice.

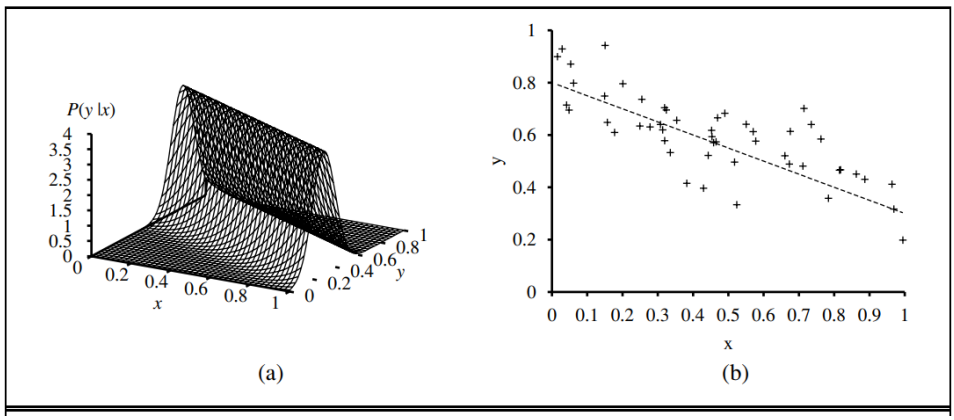


Figure 5.4.: (a) A linear Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b)A set of 50 data points generated from this model.

Now consider a linear Gaussian model with one continuous parent X and a continuous child Y ., Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional distribution P(Y | X), we can maximize the conditional likelihood

$$P(y \mid x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-(\theta_1 x+\theta_2))^2}{2\sigma^2}} . \tag{5.5}$$

Here, the parameters are θ_1, θ_2, and σ. The data are a collection of (x_j , y_j) pairs, as illustrated in Figure 5.4 Using the usual methods ,we can find the maximum-likelihood values of the parameters. The point here is different. If we consider just the parameters θ1 and θ2 that define the linear relationship between x and y, it becomes clear that maximizing the log likelihood with respect to these parameters is the same as minimizing the numerator $(y - (\theta_1 x + \theta_2))^2$ in the exponent of Equation. This is the L_2 loss, the squared error between the actual value y and the prediction $\theta_1 x + \theta_2$.

This is the quantity minimized by the standard linear regression procedure described. Now we can understand why: minimizing the sum of squared errors gives the maximum-likelihood straight-line model, provided that the data are generated with Gaussian noise of fixed variance.

5.2.5. Bayesian parameter learning

Maximum-likelihood learning gives rise to some very simple procedures, but it has some serious deficiencies with small data sets. For example, after seeing one cherry candy, the maximum-likelihood hypothesis is that the bag is 100% cherry (i.e., θ = 1.0). Unless one's hypothesis prior is that bags must be either all cherry or all lime, this is not a reasonable conclusion. It is more likely that the bag is a mixture of lime and cherry. The Bayesian approach to parameter learning starts by defining a prior probability distribution over the possible hypotheses. We call this the hypothesis prior. Then, as data arrives, the posterior probability distribution is updated.

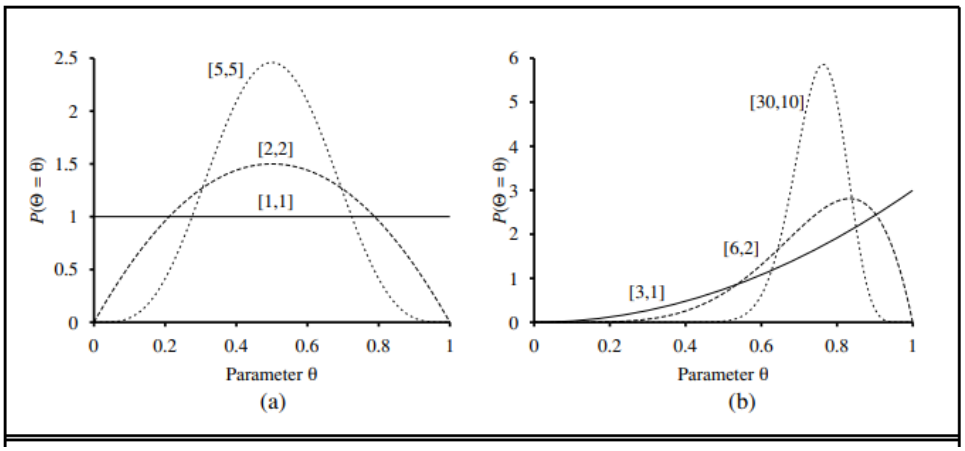


Figure 5.5: Examples of the beta[a, b] distribution for different values of [a, b]

The candy example in Figure (a) has one parameter, θ: the probability that a randomly selected piece of candy is cherry-flavored. In the Bayesian view, θ is the (unknown) value of a random variable Θ that defines the hypothesis space; the hypothesis prior is just the prior distribution P(Θ). Thus, P(Θ = θ) is the prior probability that the bag has a fraction θ of cherry candies. If the parameter θ can be any value between 0 and 1, then P(Θ) must be a continuous distribution that is nonzero only between 0 and 1 and that integrates to 1. The uniform density P(θ) = Uniform[0, 1](θ) is one candidate. It turns out that the uniform density is a member of the family of beta distributions. Each beta distribution is defined by two hyperparameters[3] a and b such that

$$\text{beta}[a, b](\theta) = \alpha\, \theta^{a-1}(1-\theta)^{b-1}\,, \tag{5.6}$$

for θ in the range [0, 1]. The normalization constant α, which makes the distribution integrate to 1, depends on a and b. Figure shows what the distribution looks like for various values of a and b. The mean value of the

distribution is a/(a + b), so larger values of a suggest a belief that Θ is closer to 1 than to 0. Larger values of a + b make the distribution more peaked, suggesting greater certainty about the value of Θ. Thus, the beta family provides a useful range of possibilities for the hypothesis prior.

Besides its flexibility, the beta family has another wonderful property: if Θ has a prior beta[a, b], then, after a data point is observed, the posterior distribution for Θ is also a beta distribution. In other words, beta is closed under update. The beta family is called the conjugate prior for the family of distributions for a Boolean variable.[4] Let's see how this works. Suppose we observe a cherry candy; then we have

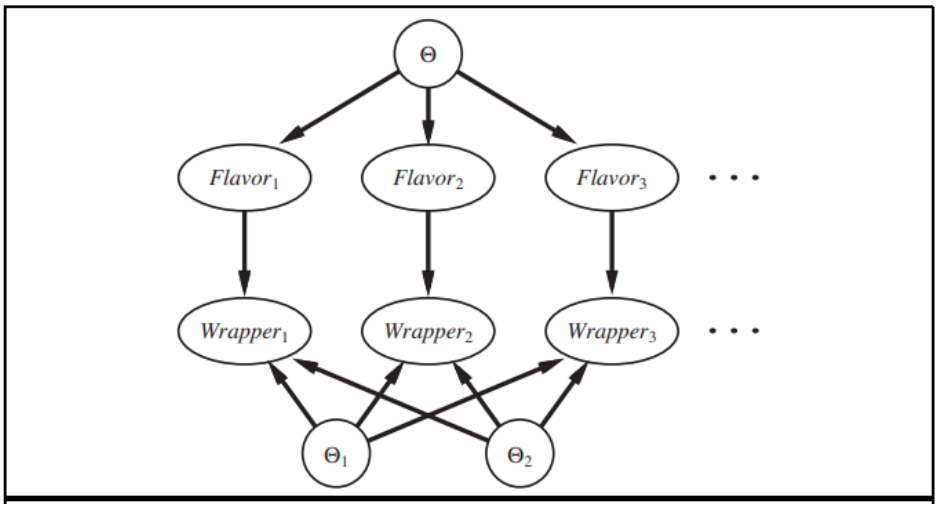


Figure 5.6: A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ, Θ_1, and Θ_2 can be inferred from their prior distributions and the evidence in the Flavor $_i$ and Wrapper $_i$ variables.

$$\begin{aligned} P(\theta \mid D_1 = cherry) &= \alpha\, P(D_1 = cherry \mid \theta) P(\theta) \\ &= \alpha'\, \theta \cdot \text{beta}[a, b](\theta) = \alpha'\, \theta \cdot \theta^{a-1}(1-\theta)^{b-1} \\ &= \alpha'\, \theta^{a}(1-\theta)^{b-1} = \text{beta}[a+1, b](\theta)\ . \end{aligned}$$

Thus, after seeing a cherry candy, we simply increment the a parameter to get the posterior; similarly, after seeing a lime candy, we increment the b parameter. Thus, we can view the a and b hyper parameters as virtual counts, in the sense that a prior beta[a, b] behaves exactly as if we had started out with a uniform prior beta[1, 1] and seen a − 1 actual cherry candies and b − 1 actual lime candies.

By examining a sequence of beta distributions for increasing values of a and b, keeping the proportions fixed, we can see vividly how the posterior distribution over the parameter Θ changes as data arrive. For example, suppose the actual bag of candy is 75% cherry. Figure 5.5(b) shows the sequence beta[3, 1], beta[6, 2], beta[30, 10]. Clearly, the distribution is converging to a narrow peak around the true value of Θ. For large data sets,

then, Bayesian learning (at least in this case) converges to the same answer as maximum-likelihood learning.

Now let us consider a more complicated case. The network in Figure 5.2(b) has three parameters, θ, θ_1, and θ_2, where θ_1 is the probability of a red wrapper on a cherry candy and θ_2 is the probability of a red wrapper on a lime candy. The Bayesian hypothesis prior must cover all three parameters—that is, we need to specify $P(\Theta, \Theta_1, \Theta_2)$. Usually, we assume parameter independence:

$$P(\Theta, \Theta_1, \Theta_2) = P(\Theta)P(\Theta_1)P(\Theta_2) .$$

With this assumption, each parameter can have its own beta distribution that is updated separately as data arrive. Figure 5.6 shows how we can incorporate the hypothesis prior and any data into one Bayesian network. The nodes Θ, Θ_1, Θ_2 have no parents. But each time we make an observation of a wrapper and corresponding flavor of a piece of candy, we add a node Flavor i, which is dependent on the flavor parameter Θ:

$$P(\text{Flavor}_i = \text{cherry} \mid \Theta = \theta) = \theta .$$

We also add a node Wrapper_i, which is dependent on Θ_1 and Θ_2:

$$P(\text{Wrapper}_i = \text{red} \mid \text{Flavor}_i = \text{cherry}, \Theta_1 = \theta_1) = \theta_1$$

$$P(\text{Wrapper}_i = \text{red} \mid \text{Flavor}_i = \text{lime}, \Theta_2 = \theta_2) = \theta_2 .$$

Now, the entire Bayesian learning process can be formulated as an inference problem. We add new evidence nodes, then query the unknown nodes (in this case, Θ, Θ_1, Θ_2). This formulation of learning and prediction makes it clear that Bayesian learning requires no extra "principles of learning." Furthermore, there is, in essence, just one learning algorithm —the inference algorithm for Bayesian networks. Of course, the nature of these networks is somewhat different from those of Chapter because of the potentially huge number of evidence variables representing the training set and the prevalence of continuous-valued parameter variables.

5.2.6. Learning Bayes net structures

So far, we have assumed that the structure of the Bayes net is given and we are just trying to learn the parameters. The structure of the network represents basic causal knowledge about the domain that is often easy for an expert, or even a naive user, to supply. In some cases, however, the causal model may be unavailable or subject to dispute—for example, certain corporations have long claimed that smoking does not cause cancer—so it is important to understand how the structure of a Bayes net can be learned from data. This section gives a brief sketch of the main ideas.

The most obvious approach is to search for a good model. We can start with a model containing no links and begin adding parents for each node, fitting the parameters with the methods we have just covered and measuring the accuracy of the resulting model. Alternatively, we can start with an initial guess at the structure and use hill-climbing or simulated annealing search to

make modifications, retuning the parameters after each change in the structure. Modifications can include reversing, adding, or deleting links. We must not introduce cycles in the process, so many algorithms assume that an ordering is given for the variables, and that a node can have parents only among those nodes that come earlier in the ordering. For full generality, we also need to search over possible orderings.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data. For example, the use of a naive Bayes model for the restaurant problem assumes that

P(Fri/Sat, Bar |WillWait) = P(Fri/Sat | WillWait)P(Bar |WillWait)

and we can check in the data that the same equation holds between the corresponding conditional frequencies. But even if the structure describes the true causal nature of the domain, statistical fluctuations in the data set mean that the equation will never be satisfied exactly, so we need to perform a suitable statistical test to see if there is sufficient evidence that the independence hypothesis is violated. The complexity of the resulting network will depend on the threshold used for this test—the stricter the independence test, the more links will be added and the greater the danger of overfitting.

An approach more consistent with the ideas in this chapter is to assess the degree to which the proposed model explains the data (in a probabilistic sense). We must be careful how we measure this, however. If we just try to find the maximum-likelihood hypothesis, we will end up with a fully connected network, because adding more parents to a node cannot decrease the likelihood We are forced to penalize model complexity in some way. The MAP (or MDL) approach simply subtracts a penalty from the likelihood of each structure (after parameter tuning) before comparing different structures. The Bayesian approach places a joint prior over structures and parameters. There are usually far too many structures to sum over (super exponential in the number of variables), so most practitioners use MCMC to sample over structures. Penalizing complexity (whether by MAP or Bayesian methods) introduces an important connection between the optimal structure and the nature of the representation for the conditional distributions in the network. With tabular distributions, the complexity penalty for a node's distribution grows exponentially with the number of parents, but with, say, noisy-OR distributions, it grows only linearly. This means that learning with noisy-OR (or other compactly parameterized) models tends to produce learned structures with more parents than does learning with tabular distributions.

5.2.7. Density estimation with nonparametric models

It is possible to learn a probability model without making any assumptions about its structure and parameterization by adopting the nonparametric methods of previous Section. The task of nonparametric density estimation is typically done in continuous domains, such as that shown in Figure 5.7(a).

The figure shows a probability density function on a space defined by two continuous variables. In Figure 5.7(b) we see a sample of data points from this density function. The question is, can we recover the model from the samples?

First we will consider k-nearest-neighbors models Given a sample of data points, to estimate the unknown probability density at a query point x we can simply measure the density of the data points in the neighborhood of x. Figure 5.7(b) shows two query points (small squares). For each query point we have drawn the smallest circle that encloses 10 neighbors—the 10-nearest-neighborhood. We can see that the central circle is large, meaning there is a low density there, and the circle on the right is small, meaning there is a high density there. In Figure 5.8 we show three plots of density estimation using k-nearest-neighbors, for different values of k. It seems clear that (b) is about right, while (a) is too spiky (k is too small) and (c) is too smooth (k is too big).

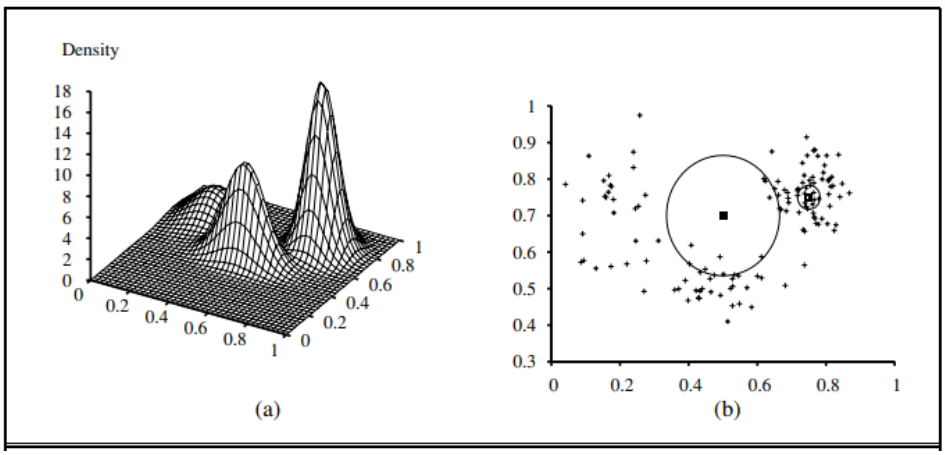


Figure 5.7.: (a) A 3D plot of the mixture of Gaussians from Figure 20.11(a). (b) A 128- point sample of points from the mixture, together with two query points (small squares) and their 10-nearest-neighborhoods (medium and large circles).

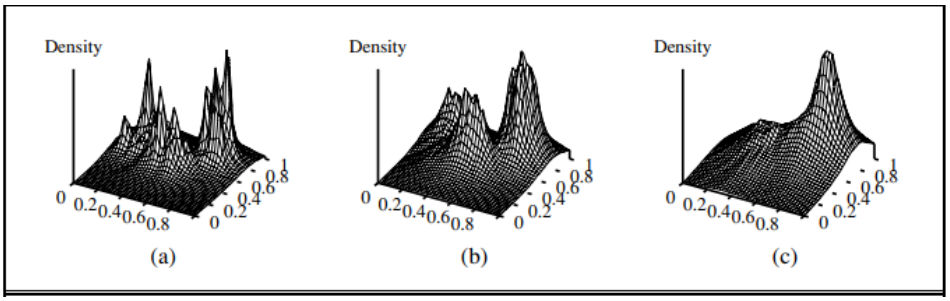


Figure 5.8.: Density estimation using k-nearest-neighbors, applied to the data in Figure 20.7(b), for k = 3, 10, and 40 respectively. k = 3 is too spiky, 40 is too smooth, and 10 is just about right. The best value for k can be chosen by cross-validation.

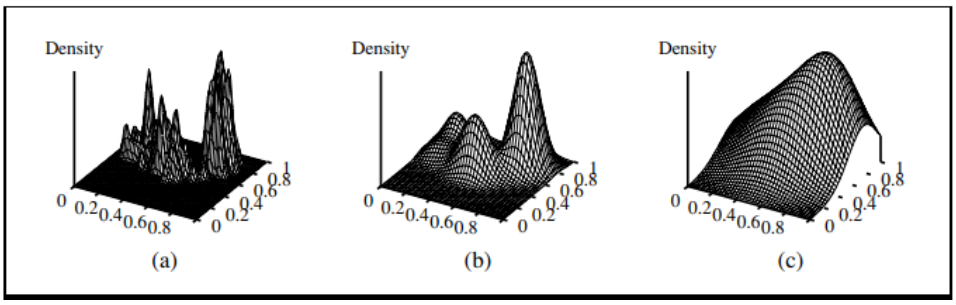


Figure 5.9.: Kernel density estimation for the data in Figure 20.7(b), using Gaussian kernels with w = 0.02, 0.07, and 0.20 respectively. w = 0.07 is about right.

Another possibility is to use kernel functions, as we did for locally weighted regression. To apply a kernel model to density estimation, assume that each data point generates its own little density function, using a Gaussian kernel. The estimated density at a query point x is then the average density as given by each kernel function:

$$P(\mathbf{x}) = \frac{1}{N}\sum_{j=1}^{N} \mathcal{K}(\mathbf{x}, \mathbf{x}_j) \ .$$

We will assume spherical Gaussians with standard deviation w along each axis:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}_j) = \frac{1}{(w^2\sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x},\mathbf{x}_j)^2}{2w^2}} \ ,$$

where d is the number of dimensions in x and D is the Euclidean distance function. We still have the problem of choosing a suitable value for kernel width w; Figure 5.9 shows values that are too small, just right, and too large. A good value of w can be chosen by using cross-validation.

5.3 LEARNING WITH HIDDEN VARIABLES: THE EM ALGORITHM.

The preceding section dealt with the fully observable case. Many real-world problems have hidden variables (sometimes called latent variables), which are not observable in the data that are available for learning. For example, medical records often include the observed symptoms, the physician's diagnosis, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself! (Note that the diagnosis is not the disease; it is a causal consequence of the observed symptoms, which are in turn caused by the disease.) One might ask, "If the disease is not observed, why not construct a model without it?" The answer appears in Figure 5.10, which shows a small, fictitious diagnostic model for heart disease. There are three observable predisposing factors and three observable symptoms (which are too depressing to name). Assume that each variable has three possible values (e.g., none, moderate, and severe).

Removing the hidden variable from the network in (a) yields the network in (b); the total number of parameters increases from 78 to 708. Thus, latent variables can dramatically reduce the number of parameters required to specify a Bayesian network. This, in turn, can dramatically reduce the amount of data needed to learn the parameters.

Hidden variables are important, but they do complicate the learning problem. In Figure 5.10(a), for example, it is not obvious how to learn the conditional distribution for HeartDisease, given its parents, because we do not know the value of HeartDisease in each case; the same problem arises in learning the distributions for the symptoms. This section describes an algorithm called expectation–maximization, or EM, that solves this problem in a very general way. We will show three examples and then provide a general description. The algorithm seems like magic at first, but once the intuition has been developed, one can find applications for EM in a huge range of learning problems.

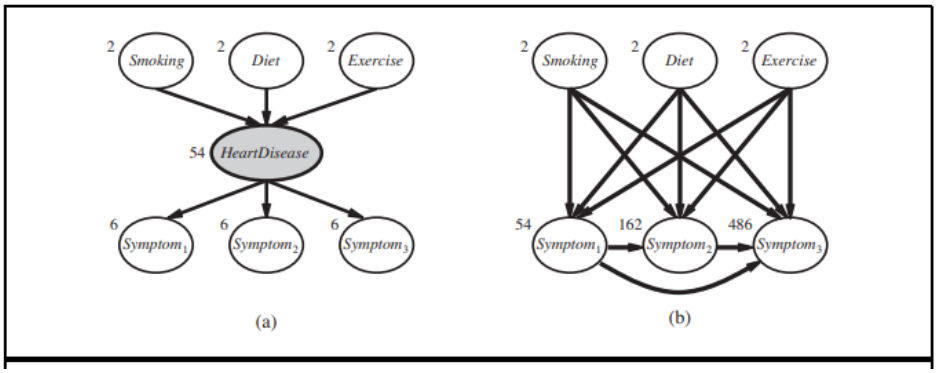


Figure 5.10.: (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with HeartDisease removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.

5.3.1. Unsupervised clustering: Learning mixtures of Gaussians

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given. For example, suppose we record the spectra of a hundred thousand stars; are there different types of stars revealed by the spectra, and, if so, how many types and what are their characteristics? We are all familiar with terms such as “red giant” and “white dwarf,” but the stars do not carry these labels on their hats—astronomers had to perform unsupervised clustering to identify these categories. Other examples include the identification of species, genera, orders, and so on in the Linnæan taxonomy and the creation of natural kinds for ordinary objects.

Unsupervised clustering begins with data. Figure 5.11(b) shows 500 data points, each of which specifies the values of two continuous attributes. The

data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies. Next, we need to understand what kind of probability distribution might have generated the data. Clustering presumes that the data are generated from a mixture distribution, P. Such a distribution has k components, each of which is a distribution in its own right. A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable C denote the component, with values 1,...,k; then the mixture distribution is given by

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)\, P(\mathbf{x} \mid C=i)\ ,$$

where x refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the multivariate Gaussian, which gives the so-called mixture of Gaussians family of distributions. The parameters of a mixture of Gaussians are

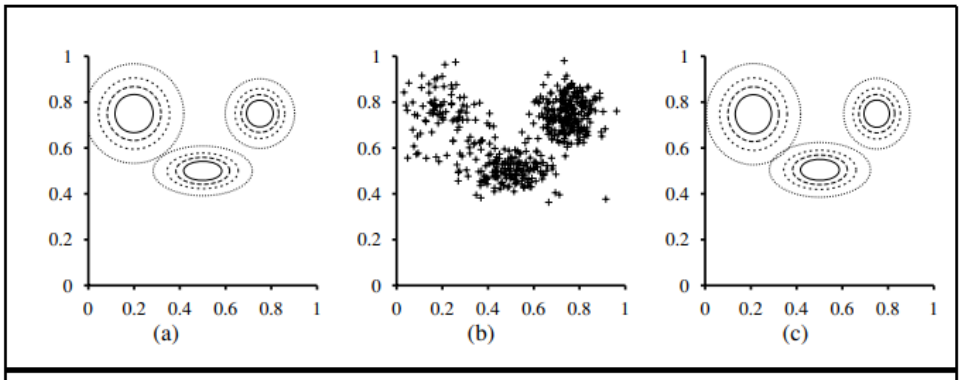


Figure 5.11 (a) A Gaussian mixture model with three components; the weights (left-toright) are 0.2, 0.3, and 0.5. (b) 500 data points sampled from the model in (a). (c) The model reconstructed by EM from the data in (b).

w_i = P(C = i) (the weight of each component), μ_i (the mean of each component), and Σi (the covariance of each component). Figure 5.11(a) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (b) as well as being the model shown in Figure 5.7(a)

The unsupervised clustering problem, then, is to recover a mixture model like the one in Figure5.11(a) from raw data like that in Figure 5.11(b). Clearly, if we knew which component generated each data point, then it would be easy to recover the component Gaussians: we could just select all the data points from a given component and then apply (a multivariate version of) Equation (5.4) for fitting the parameters of a Gaussian to a set of data. On the other hand, if we knew the parameters of each component, then we could, at least in a probabilistic sense, assign each data point to a component. The problem is that we know neither the assignments nor the parameters.

The basic idea of EM in this context is to pretend that we know the parameters of the model and then to infer the probability that each data point belongs to each component. After that, we refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence. Essentially, we are "completing" the data by inferring probability distributions over the hidden variables—which component each data point belongs to—based on the current model. For the mixture of Gaussians, we initialize the mixture-model parameters arbitrarily and then iterate the following two steps:

1. **E-step:** Compute the probabilities $p_{ij} = P(C = i| x_j)$, the probability that datum x_j was generated by component i. By Bayes' rule, we have $p_{ij} = \alpha P(x_j | C = i)P(C = i)$. The term $P(xj | C = i)$ is just the probability at xj of the ith Gaussian, and the term $P(C = i)$ is just the weight parameter for the ith Gaussian. Define $ni = \Sigma_j p_{ij}$, the effective number of data points currently assigned to component i.

2. **M-step:** Compute the new mean, covariance, and component weights using the following steps in sequence:

$$\boldsymbol{\mu}_i \leftarrow \sum_j p_{ij}\mathbf{x}_j / n_i$$
$$\boldsymbol{\Sigma}_i \leftarrow \sum_j p_{ij}(\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^\top / n_i$$
$$w_i \leftarrow n_i / N$$

where N is the total number of data points. The E-step, or expectation step, can be viewed as computing the expected values p_{ij} of the hidden indicator variables Z_{ij}, where Z_{ij} is 1 if datum xj was generated by the i^{th} component and 0 otherwise. The M-step, or maximization step, finds the new values of the parameters that maximize the log likelihood of the data, given the expected values of the hidden indicator variables.

The final model that EM learns when it is applied to the data in Figure 5.11(a) is shown in Figure 2511(c); it is virtually indistinguishable from the original model from which the data were generated. Figure 5.12(a) plots the log likelihood of the data according to the current model as EM progresses. here are two points to notice. First, the log likelihood for the final learned model slightly exceeds that of the original model, from which the data were generated. This might seem surprising, but it simply reflects the fact that the data were generated randomly and might not provide an exact reflection of the underlying model. The second point is that EM increases the log likelihood of the data at every iteration. This fact can be proved in general.

Furthermore, under certain conditions (that hold in ost cases), EM can be proven to reach a local maximum in likelihood. (In rare cases, it could reach a saddle point or even a local minimum.) In this sense, EM resembles a gradient-based hill-climbing algorithm, but notice that it has no "step size" parameter.

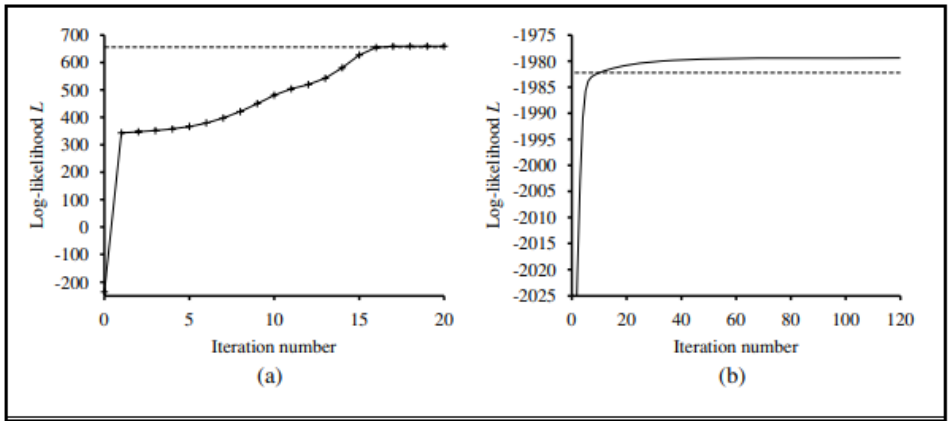


Figure 5.12: Graphs showing the log likelihood of the data, L, as a function of the EM

iteration. The horizontal line shows the log likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 5.11. (b) Graph for the Bayesian network in Figure 5.13(a).

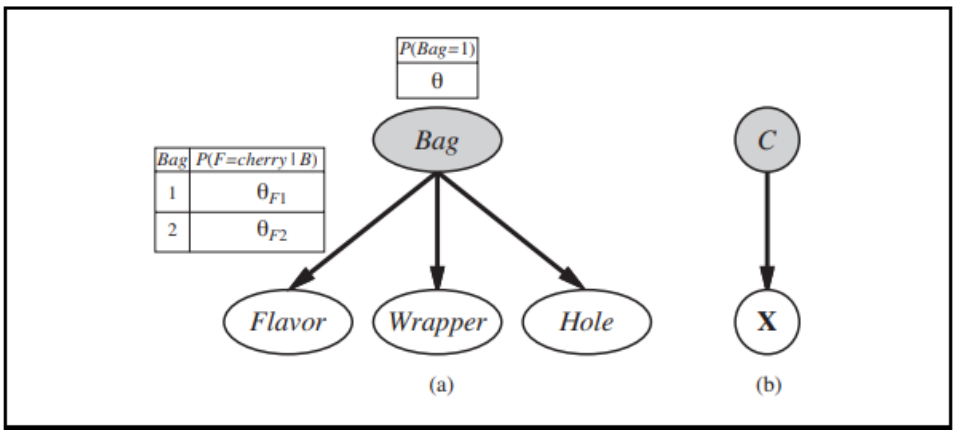


Figure 5.13: (a) A mixture model for candy. The proportions of different flavors, wrappers, presence of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables X depend on the component C.

Things do not always go as well as Figure 5.12(a) might suggest. It can happen, for example, that one Gaussian component shrinks so that it covers just a single data point. Then its variance will go to zero and its likelihood will go to infinity! Another problem is that two components can "merge," acquiring identical means and variances and sharing their data points. These kinds of degenerate local maxima are serious problems, especially in high dimensions. One solution is to place priors on the model parameters and to apply the MAP version of EM. Another is to restart a component with new random parameters if it gets too small or too close to another component. Sensible initialization also helps.

5.3.2. Learning Bayesian networks with hidden variables

To learn a Bayesian network with hidden variables, we apply the same insights that worked for mixtures of Gaussians. Figure 5.13 represents a situation in which there are two bags of candies that have been mixed together. Candies are described by three features: in addition to the Flavor and the Wrapper , some candies have a Hole in the middle and some do not. The distribution of candies in each bag is described by a naive Bayes model: the features are independent, given the bag, but the conditional probability distribution for each feature depends on the bag. The parameters are as follows: θ is the prior probability that a candy comes from Bag 1; θ_{F1} and θ_{F2} are the probabilities that the flavor is cherry, given that the candy comes from Bag 1 or Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{H1} and θ_{H2} give the probabilities that the candy has a hole. Notice that the overall model is a mixture model. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 5.13(b).) In the figure, the bag is a hidden variable because, once the candies have been mixed together, we no longer know which bag each candy came from. In such a case, can we recover the descriptions of the two bags by observing candies from the mixture? Let us work through an iteration of EM for this problem. First, let's look at the data. We generated 1000 samples from a model whose true parameters are as follows:

$$\theta = 0.5, \theta_{F1} = \theta_{W1} = \theta_{H1} = 0.8, \theta_{F2} = \theta_{W2} = \theta_{H2} = 0.3 . \qquad (5.7)$$

That is, the candies are equally likely to come from either bag; the first is mostly cherries with red wrappers and holes; the second is mostly limes with green wrappers and no holes. The counts for the eight possible kinds of candy are as follows:

	$W = red$		$W = green$	
	$H = 1$	$H = 0$	$H = 1$	$H = 0$
$F = cherry$	273	93	104	90
$F = lime$	79	100	94	167

We start by initializing the parameters. For numerical simplicity, we arbitrarily choose5

$$\theta^{(0)} = 0.6, \quad \theta_{F1}^{(0)} = \theta_{W1}^{(0)} = \theta_{H1}^{(0)} = 0.6, \quad \theta_{F2}^{(0)} = \theta_{W2}^{(0)} = \theta_{H2}^{(0)} = 0.4 .$$

First, let us work on the θ parameter. In the fully observable case, we would estimate this directly from the observed counts of candies from bags 1 and 2. Because the bag is a hidden variable, we calculate the expected counts instead. The expected count $\hat{N}(Bag = 1)$ is the sum, over all candies, of the probability that the candy came from bag 1:

$$\theta^{(1)} = \hat{N}(Bag = 1)/N = \sum_{j=1}^{N} P(Bag = 1 \mid flavor_j, wrapper_j, holes_j)/N . \qquad (5.8)$$

These probabilities can be computed by any inference algorithm for Bayesian networks. For a naive Bayes model such as the one in our example, we can do the inference "by hand," using Bayes' rule and applying conditional independence:

$$\theta^{(1)} = \frac{1}{N}\sum_{j=1}^{N} \frac{P(flavor_j \mid Bag=1)P(wrapper_j \mid Bag=1)P(holes_j \mid Bag=1)P(Bag=1)}{\sum_i P(flavor_j \mid Bag=i)P(wrapper_j \mid Bag=i)P(holes_j \mid Bag=i)P(Bag=i)} .$$

Applying this formula to, say, the 273 red-wrapped cherry candies with holes, we get a contribution of

$$\frac{273}{1000} \cdot \frac{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)}}{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)} + \theta_{F2}^{(0)}\theta_{W2}^{(0)}\theta_{H2}^{(0)}(1-\theta^{(0)})} \approx 0.22797 .$$

Continuing with the other seven kinds of candy in the table of counts, we obtain $\theta^{(1)} = 0.6124$.

Now let us consider the other parameters, such as θ_{F1}. In the fully observable case, we would estimate this directly from the observed counts of cherry and lime candies from bag 1.

The expected count of cherry candies from bag 1 is given by

$$\sum_{j:Flavor_j = cherry} P(Bag=1 \mid Flavor_j = cherry, wrapper_j, holes_j) .$$

Again, these probabilities can be calculated by any Bayes net algorithm. Completing this

process, we obtain the new values of all the parameters:

$$\theta^{(1)} = 0.6124,\ \theta_{F1}^{(1)} = 0.6684,\ \theta_{W1}^{(1)} = 0.6483,\ \theta_{H1}^{(1)} = 0.6558,$$
$$\theta_{F2}^{(1)} = 0.3887, \theta_{W2}^{(1)} = 0.3817,\ \theta_{H2}^{(1)} = 0.3827 . \qquad (5.9)$$

The log likelihood of the data increases from about −2044 initially to about −2021 after the first iteration, as shown in Figure 5.12(b). That is, the update improves the likelihood itself by a factor of about $e^{23} \approx 10^{10}$. By the tenth iteration, the learned model is a better fit than the original model (L = −1982.214). Thereafter, progress becomes very slow. This is not uncommon with EM, and many practical systems combine EM with a gradient-based algorithm such as Newton–Raphson for the last phase of learning.

The general lesson from this example is that the parameter updates for Bayesian network learning with hidden variables are directly available from the results of inference on each example. Moreover, only local posterior probabilities are needed for each parameter. Here, "local" means that the CPT for each variable Xi can be learned from posterior probabilities involving just Xi and its parents U_i. Defining θ_{ijk} to be the CPT parameter $P(X_i = x_{ij} \mid U_i = u_{ik})$, the update is given by the normalized expected counts as follows:

$$\theta_{ijk} \leftarrow \ddot{N}(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik}) / \ddot{N}(\mathbf{U}_i = \mathbf{u}_{ik}) \ .$$

The expected counts are obtained by summing over the examples, computing the probabilities

$P(X_i = x_{ij}, U_i = u_{ik})$ for each by using any Bayes net inference algorithm. For the exact algorithms—including variable elimination—all these probabilities are obtainable directly as a by-product of standard inference, with no need for extra computations specific to learning. Moreover, the information needed for learning is available locally for each parameter.

5.3.3. Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from previous Section that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 5.14. Each data point consists of an observation sequence of finite length, so the problem is to learn the transition probabilities from a set of observation sequences (or from just one long sequence).

We have already worked out how to learn Bayes nets, but there is one complication: in Bayes nets, each parameter is distinct; in a hidden Markov model, on the other hand, the individual transition probabilities from state i to state j at time t, $\theta_{ijt} = P(X_t+1 = j \mid X_t = i)$, are repeated across time—that is, $\theta_{ijt} = \theta_{ij}$ for all t. To estimate the transition probability from state i to state j, we simply calculate the expected proportion of times that the system undergoes a transition to state j when in state i:

$$\theta_{ij} \leftarrow \sum_t \hat{N}(X_{t+1} = j, X_t = i) / \sum_t \hat{N}(X_t = i) \ .$$

The expected counts are computed by an HMM inference algorithm. The forward–backward

algorithm shown in Figure 15.4 can be modified very easily to compute the necessary probabilities. One important point is that the probabilities required are obtained by smoothing

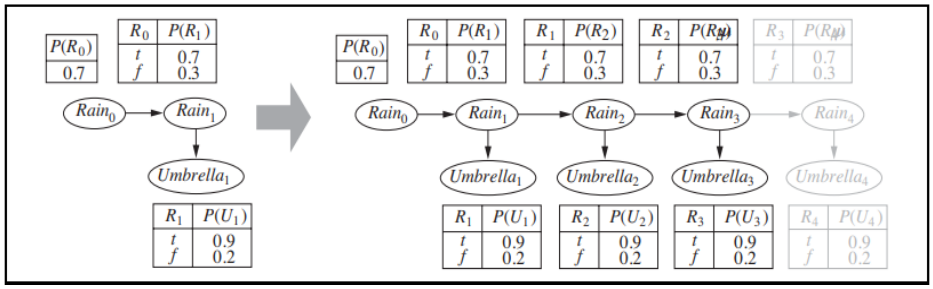


Figure 5.14.: An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 5.16). rather than filtering; that is, we need to pay attention to subsequent evidence in estimating the probability that a particular transition occurred. The evidence in a murder

case is usually obtained after the crime (i.e., the transition from state i to state j) has taken place.

5.4 THE GENERAL FORM OF THE EM ALGORITHM

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let x be all the observed values in all the examples, let Z denote all the hidden variables for all the examples, and let θ be all the parameters for the probability model. Then the EM algorithm is

$$\boldsymbol{\theta}^{(i+1)} = \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{\mathbf{z}} P(\mathbf{Z}=\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}^{(i)}) L(\mathbf{x}, \mathbf{Z}=\mathbf{z} \mid \boldsymbol{\theta}) \ .$$

This equation is the EM algorithm in a nutshell. The E-step is the computation of the summation, which is the expectation of the log likelihood of the "completed" data with respect to the distribution $P(Z = z \mid x, \theta^{(i)})$, which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden variables are the Z_{ij} s, where Z_{ij} is 1 if example j was generated by component i. For Bayes nets, Z_{ij} is the value of unobserved variable X_i in example j. For HMMs, Z_{jt} is the state of the sequence in example j at time t. Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of posteriors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an approximate E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC, the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated exactly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational and loopy methods, have also proved effective for learning very large networks.

SUMMARY

Statistical learning methods range from simple calculation of averages to the construction of complex models such as Bayesian networks. They have applications throughout computer science, engineering, computational biology, neuroscience, psychology, and physics. This chapter has presented some of the basic ideas and given a flavor of the mathematical underpinnings. The main points are as follows: Bayesian learning, Maximum a posteriori, Maximum-likelihood, Bayesian networks, model selection.

Statistical learning continues to be a very active area of research. Enormous strides have been made in both theory and practice, to the point where it is possible to learn almost any model for which exact or approximate inference is feasible.

EXERCISE

1. What is probability? How it is derived.
2. Explain bayes theorem with suitable illustration.
3. Give the outline of Bayesian classification.
4. Explain supervised and unsupervised algorithm.
5. Write a note on Naïve Bayes Model.
6. Give the outline of EM algorithm.

REFERENCE

- Artificial Intelligence: A Modern Approach Third Edition
- Artificial Intelligence: Foundations of Computational Agents, David L Poole,Alan K. Mackworth, 2nd Edition, Cambridge University Press ,2017.
- Artificial Intelligence, Kevin Knight and Elaine Rich, 3rd Edition, 2017
- The Elements of Statistical Learning, Trevor Hastie, Robert Tibshirani and Jerome Friedman, Springer, 2013

❄❄❄❄❄❄❄

6

REINFORCEMENT LEARNING

Unit Structure :

6.0 OBJECTIVES

A learner will learn about:

- Concepts of reinforcement learning
- Adaptive dynamic programming
- Implementation of reinforcement learning concepts
- Applications of reinforcement learning

6.1 INTRODUCTION

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

reward, or reinforcement. In games like chess, the reinforcement is received only at the end of the game. In other environments, the rewards come more frequently. In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement. Our framework for agents regards the reward as part of the input percept, but the agent must be "hardwired" to recognize that part as a reward rather than as just another sensory input. Thus, animals seem to be hardwired to recognize pain and hunger as negative rewards and pleasure and food intake as positive rewards. Reinforcement has been carefully studied by animal psychologists for over 60 years.

Rewards were introduced where they served to define optimal policies in Markov decision processes (MDPs). An optimal policy is a policy that maximizes the expected total reward. The task of reinforcement learning is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment. the agent has a complete model of the environment and knows the reward function, here we assume no prior knowledge of either. Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose." This is reinforcement learning in a nutshell.

In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels. For example, in game playing, it is very hard for a human to provide accurate and consistent evaluations of large numbers of positions, which would be needed to train an evaluation function directly from examples. Instead, the program can be told when it has won or lost, and it can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position. Similarly, it is extremely difficult to program an agent to fly a helicopter; yet given appropriate negative rewards for crashing, wobbling, or deviating from a set course, an agent can learn to fly by itself.

Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein. To keep the chapter manageable, we will concentrate on simple environments and simple agent designs. For the most part, we will assume a fully observable environment, so that the current state is supplied by each percept. On the other hand, we will assume that the agent does not know how the environment works or what its actions do, and we will allow for probabilistic action outcomes. Thus, the agent faces an unknown Markov decision process. We will consider three of the agent designs.

- A utility-based agent learns a utility function on states and uses it to select actions that maximize the expected outcome utility.
- A Q-learning agent learns an action-utility function, or Q-function, giving the expected utility of taking a given action in a given state.
- A reflex agent learns a policy that maps directly from states to actions.

A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead. For example, in order to make use of a backgammon evaluation function, a backgammon program must know what its legal moves are and how they affect the board position. Only in this way can it apply the utility function to the outcome states. A Q-learning agent, on the other hand, can compare the expected utilities for its available choices without needing to know their outcomes, so it does not need a model of the environment. On the other hand, because they do not know where their actions lead, Q-learning agents cannot look ahead; this can seriously restrict their ability to learn, as we shall see.

We begin in Section with passive learning, where the agent's policy is fixed and the task is to learn the utilities of states (or state–action pairs); this could also involve learning a model of the environment. The principal issue is exploration: an agent must experience as much as possible of its environment in order to learn how to behave in it. discusses how an agent can use inductive learning to learn much faster from its experiences. covers methods for learning direct policy representations in reflex agents. An understanding of Markov decision processes is essential for this chapter.

6.2. PASSIVE REINFORCEMENT LEARNING

To keep things simple, we start with the case of a passive learning agent using a state-based representation in a fully observable environment. In passive learning, the agent's policy π is fixed: in state s, it always executes the action π(s). Its goal is simply to learn how good the policy is—that is, to learn the utility function $U^{\pi}(s)$. a policy for that world and the corresponding utilities. Clearly, the passive learning task is similar to the policy evaluation task, part of the policy iteration algorithm described in previous Section. The main difference is that the passive learning agent does not know the transition model P(s | s, a), which specifies the probability of reaching state s from state s after doing action a; nor does it know the reward function R(s), which specifies the reward for each state.

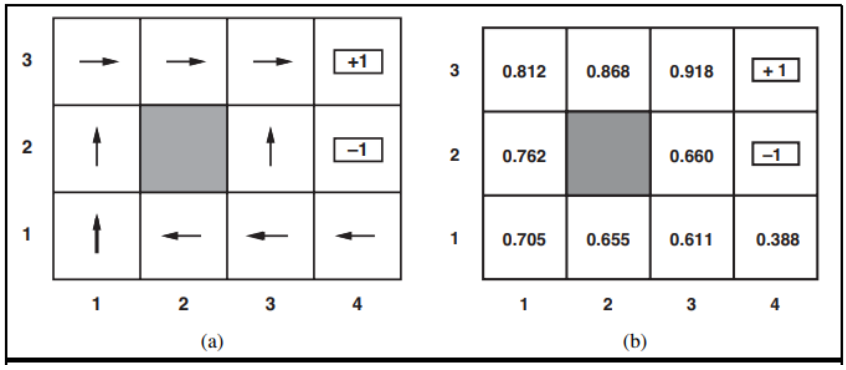


Figure 6.1 (a) A policy π for the 4 × 3 world; this policy happens to be optimal with rewards of R(s) = − 0.04 in the nonterminal states and no discounting. (b) The utilities of the states in the 4 ×3 world, given policy π.

The agent executes a set of trials in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state. Typical trials might look like this:

$$(1,1)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (2,3)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (4,3)_{\textbf{+1}}$$
$$(1,1)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (2,3)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (3,2)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (4,3)_{\textbf{+1}}$$
$$(1,1)_{\textbf{-.04}} \rightsquigarrow (2,1)_{\textbf{-.04}} \rightsquigarrow (3,1)_{\textbf{-.04}} \rightsquigarrow (3,2)_{\textbf{-.04}} \rightsquigarrow (4,2)_{\textbf{-1}} .$$

Note that each state percept is subscripted with the reward received. The object is to use the information about rewards to learn the expected utility $U^{\pi}(s)$ associated with each nonterminal states. The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed.

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right] \qquad (6.1)$$

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

6.2.1 Direct utility estimation

A simple method for direct utility estimation was invented in the late 1950s in the area of adaptive control theory by Widrow and Hoff (1960). The idea is that the utility of a state is the expected total reward from that state onward (called the expected reward-to-go), and each trial provides a sample of this quantity for each state visited. For example, the first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1), two samples of 0.76 and 0.84 for (1,2), two samples of 0.80 and 0.88 for (1,3), and so on. Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table. In the limit of infinitely many trials, the sample average will converge to the true expectation in Equation.

It is clear that direct utility estimation is just an instance of supervised learning where each example has the state as input and the observed reward-to-go as output. This means that we have reduced reinforcement learning to a standard inductive learning problem, as discussed in previous Chapter. in previous Section discusses the use of more powerful kinds of representations for the utility function. Learning techniques for those representations can be applied directly to the observed data.

Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known. Unfortunately, it misses a very important source of information, namely, the fact that the utilities of states are not independent! The utility of each state

equals its own reward plus the expected utility of its successor states. That is, the utility values obey the Bellman equations for a fixed policy.

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s') \; . \tag{6.2}$$

By ignoring the connections between states, direct utility estimation misses opportunities for learning. For example, the second of the three trials given earlier reaches the state (3,2), which has not previously been visited. The next transition reaches (3,3), which is known from the first trial to have a high utility. The Bellman equation suggests immediately that (3,2) is also likely to have a high utility, because it leads to (3,3), but direct utility estimation learns nothing until the end of the trial. More broadly, we can view direct utility estimation as searching for U in a hypothesis space that is much larger than it needs to be, in that it includes many functions that violate the Bellman equations. For this reason, the algorithm often converges very slowly.

function PASSIVE-ADP-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, discount γ
    U, a table of utilities, initially empty
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
    for each t such that $N_{s'|sa}[t, s, a]$ is nonzero **do**
      $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \; / \; N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION(π, U, *mdp*)
  if s'.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$
  return a

Figure 6.2 A passive reinforcement learning agent based on adaptive dynamic programming. The POLICY-EVALUATION function solves the fixed-policy Bellman equations.

6.2.2 Adaptive dynamic programming

An adaptive dynamic programming (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method. For a passive learning agent, this means plugging the learned transition model P(s | s, π(s)) and the observed rewards R(s) into the Bellman equations (6.2) to calculate the utilities of the states. As we remarked in our discussion of policy iteration in previous Chapter, these equations are linear (no maximization involved) so they can be solved using any linear algebra package. Alternatively, we can adopt the approach of modified policy iteration using a simplified value iteration process to update the utility estimates after each change to the learned

model. Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.

The process of learning the model itself is easy, because the environment is fully observable. This means that we have a supervised learning task where the input is a state–action pair and the output is the resulting state. In the simplest case, we can represent the transition model as a table of probabilities. We keep track of how often each action outcome occurs and estimate the transition probability P(s | s, a) from the frequency with which s is reached when executing a in s. For example, in the three trials given, Right is executed three times in (1,3) and two out of three times the resulting state is (2,3), so P((2, 3)| (1, 3), Right) is estimated to be 2/3.

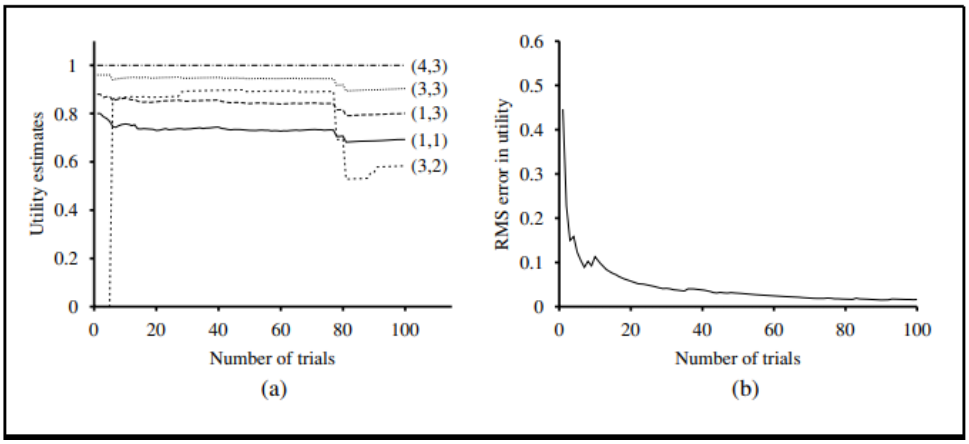


Figure 6.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 6.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the −1 terminal state at (4,2). (b) The root-mean-square error (see Appendix A) in the estimate for U(1, 1), averaged over 20 runs of 100 trials each.

The full agent program for a passive ADP agent is shown in Figure 6.2. Its performance on the 4 × 3 world is shown in Figure 6.3. In terms of how quickly its value estimates improve, the ADP agent is limited only by its ability to learn the transition model. In this sense, it provides a standard against which to measure other reinforcement learning algorithms. It is, however, intractable for large state spaces. In backgammon, for example, it would involve solving roughly 1050 equations in 1050 unknowns.

A reader familiar with the Bayesian learning ideas of previous Chapter will have noticed that the algorithm in Figure 6.2 is using maximum-likelihood estimation to learn the transition model; moreover, by choosing a policy based solely on the estimated model it is acting as if the model were correct. This is not necessarily a good idea! For example, a taxi agent that didn't know about how traffic lights might ignore a red light once or twice without no ill effects and then formulate a policy to ignore red lights from then on. Instead, it might be a good idea to choose a policy that, while not optimal for the model estimated by maximum likelihood, works reasonably well for

the whole range of models that have a reasonable chance of being the true model. There are two mathematical approaches that have this flavor.

P(h) for each hypothesis h about what the true model is; the posterior probability P(h | e) is

obtained in the usual way by Bayes' rule given the observations to date. Then, if the agent has

decided to stop learning, the optimal policy is the one that gives the highest expected utility.

Let u^{π}_{h} be the expected utility, averaged over all possible start states, obtained by executing

policy π in model h. Then we have

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_h P(h \mid \mathbf{e}) u_h^{\pi} .$$

In some special cases, this policy can even be computed! If the agent will continue learning in the future, however, then finding an optimal policy becomes considerably more difficult, because the agent must consider the effects of future observations on its beliefs about the transition model. The problem becomes a POMDP whose belief states are distributions over models. This concept provides an analytical foundation for understanding the exploration problem described in previous Section

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

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \min_h u_h^{\pi} .$$

Often, the set H will be the set of models that exceed some likelihood threshold on P(h | e), so the robust and Bayesian approaches are related. Sometimes, the robust solution can be computed efficiently. There are, moreover, reinforcement learning algorithms that tend to produce robust solutions, although we do not cover them here.

6.2.3 Temporal-difference learning

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

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

so $U^{\pi}(1, 3)$ would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state's to state's, we apply the following update to $U^{\pi}(s)$:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)) \,. \qquad (6.3)$$

Here, α is the learning rate parameter. Because this update rule uses the difference in utilities

between successive states, it is often called the temporal-difference, or TD, equation.

All temporal-difference methods work by adjusting the utility estimates towards the ideal equilibrium that holds locally when the utility estimates are correct. In the case of passive learning, the equilibrium is given by Equation (6.2). Now Equation (6.3) does in fact cause the agent to reach the equilibrium given by Equation (6.2), but there is some subtlety involved. First, notice that the update involves only the observed successor s , whereas the actual equilibrium conditions involve all possible next states. One might think that this causes an improperly large change in $U^{\pi}(s)$ when a very rare transition occurs; but, in fact, because rare transitions occur only rarely, the average value of $U^{\pi}(s)$ will converge to the correct value. Furthermore, if we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $U^{\pi}(s)$ itself will converge to the correct value.1 This gives us the agent program shown in Figure 6.4. Figure 6.5 illustrates the performance of the passive TD agent on the 4 × 3 world. It does not learn quite as fast as the ADP agent and shows much higher variability, but it is much simpler and requires much less computation per observation. Notice that TD does not need a transition model to perform its updates. The environment supplies the connection between neighboring states in the form of observed transitions.

```
function PASSIVE-TD-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
              U, a table of utilities, initially empty
              N_s, a table of frequencies for states, initially zero
              s, a, r, the previous state, action, and reward, initially null

  if s' is new then U[s'] ← r'
  if s is not null then
      increment N_s[s]
      U[s] ← U[s] + α(N_s[s])(r + γ U[s'] − U[s])
  if s'.TERMINAL? then s, a, r ← null else s, a, r ← s', π[s'], r'
  return a
```

Figure 6.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function α(n) is chosen to ensure convergence, as described in the text.

The ADP approach and the TD approach are actually closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One difference is that TD adjusts a state to agree with its observed successor (Equation (6.3)), whereas ADP adjusts

the state to agree with all of the successors that might occur, weighted by their probabilities (Equation (6.2)). This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability. A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the environment model P. Although the observed transition makes only a local change in P, its effects might need to be propagated throughout U. Thus, TD can be viewed as a crude but efficient first approximation to ADP.

Each adjustment made by ADP could be seen, from the TD point of view, as a result of a "pseudoexperience" generated by simulating the current environment model. It is possible to extend the TD approach to use an environment model to generate several pseudoexperiences—transitions that the TD agent can imagine might happen, given its current model. For each observed transition, the TD agent can generate a large number of imaginary transitions. In this way, the resulting utility estimates will approximate more and more closely those of ADP—of course, at the expense of increased computation time.

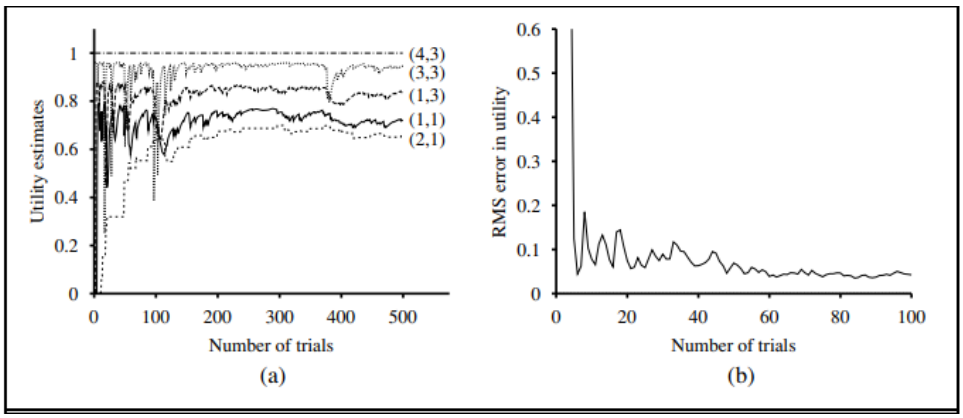


Figure 6.5 The TD learning curves for the 4 × 3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials. (b) The root-mean-square error in the estimate for U(1, 1), averaged over 20 runs of 500 trials each. Only the first 100 trials are shown to enable comparison with Figure 6.3.

In a similar vein, we can generate more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Even though the value iteration algorithm is efficient, it is intractable if we have, say, 10^{100} states. However, many of the necessary adjustments to the state values on each iteration will be extremely tiny. One possible approach to generating reasonably good answers quickly is to bound the number of adjustments made after each observed transition. One can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones. The prioritized sweeping heuristic prefers to make adjustments to states whose likely successors have just undergone a large adjustment in their own utility estimates. Using heuristics like this,

approximate ADP algorithms usually can learn roughly as fast as full ADP, in terms of the number of training sequences, but can be several orders of magnitude more efficient in terms of computation. This enables them to handle state spaces that are far too large for full ADP. Approximate ADP algorithms have an additional advantage: in the early stages of learning a new environment, the environment model P often will be far from correct, so there is little point in calculating an exact utility function to match it. An approximation algorithm can use a minimum adjustment size that decreases as the environment model becomes more accurate. This eliminates the very long value iterations that can occur early in learning due to large changes in the model.

6.3 ACTIVE REINFORCEMENT LEARNING

A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take. Let us begin with the adaptive dynamic programming agent and consider how it must be modified to handle this new freedom.

First, the agent will need to learn a complete model with outcome probabilities for all actions, rather than just the model for the fixed policy. The simple learning mechanism used by PASSIVE-ADP-AGENT will do just fine for this. Next, we need to take into account the fact that the agent has a choice of actions. The utilities it needs to learn are those defined by the optimal policy; they obey the Bellman equations given, which we repeat here for convenience:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' \mid s, a) U(s') \,. \qquad (6.4)$$

These equations can be solved to obtain the utility function U using the value iteration or policy iteration algorithms. The final issue is what to do at each step. Having obtained a utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look-ahead to maximize the expected utility; alternatively, if it uses policy iteration, the optimal policy is already available, so it should simply execute the action the optimal policy recommends. Or should it?

6.4 GENERALIZATION IN REINFORCEMENT LEARNING

So far, we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple. Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger. With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more. This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question. Backgammon and chess are tiny subsets of the real world, yet their state spaces contain on the order of 10^{20} and 10^{40} states, respectively. It would be absurd to suppose that one must visit all these states many times in order to learn how to play the game!

One way to handle such problems is to use function approximation, which simply means using any sort of representation for the Q-function other than a lookup table. The representation is viewed as approximate because it might not be the case that the true utility function or Q-function can be represented in the chosen form. For example, in previous Chapter we described an evaluation function for chess that is represented as a weighted linear function of a set of features (or basis functions) $f_1,...,f_n$:

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

A reinforcement learning algorithm can learn values for the parameters $\theta = \theta_1,...,\theta_n$ such

that the evaluation function $U\hat{}_\theta$ approximates the true utility function. Instead of, say, 1040

values in a table, this function approximator is characterized by, say, n = 20 parameters an enormous compression. Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers. If the approximation is good enough, however, the agent might still play excellent chess.3 Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit. The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited. That is, the most important aspect of function approximation is not that it requires less space, but that it allows for inductive generalization over input states. To give you some idea of the power of this effect: by examining only one in every 10^{12} of the possible backgammon states, it is possible to learn a utility function that allows a program to play as well as any human (Tesauro, 1992).

On the flip side, of course, there is the problem that there could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well. As in all inductive learning, there is a tradeoff between the size of the hypothesis space and the time it takes to learn the function. A larger hypothesis space increases the likelihood that a good approximation can be found, but also means that convergence is likely to be delayed. Let us begin with the simplest case, which is direct utility estimation. (See Section 21.2.) With function approximation, this is an instance of supervised learning. For example, suppose we represent the utilities for the 4 × 3 world using a simple linear function. The features of the squares are just their x and y coordinates, so we have

$$U\hat{}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y . \qquad (6.5)$$

Thus, if $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $U\hat{}_\theta(1, 1) = 0.8$. Given a collection of trials, we obtain a set of sample values of $U\hat{}_\theta(x, y)$, and we can find the best fit, in the sense of minimizing the squared error, using standard linear regression.

For reinforcement learning, it makes more sense to use an online learning algorithm that updates the parameters after each trial. Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that

$U\hat{}_\theta(1, 1)$, currently 0.8, is too large and must be reduced. How should the parameters be adjusted to achieve this? As with neuralnetwork learning, we write an error function and compute its gradient with respect to the parameters. If $u_j(s)$ is the observed total reward from state s onward in the j^{th} trial, then the error is defined as (half) the squared difference of the predicted total and the actual total: $E_j(s)=(U\hat{}_\theta(s) - u_j(s))2/2$. The rate of change of the error with respect to each parameter θ_i is $\partial E_j/\partial\theta_i$, so to move the parameter in the direction of decreasing the error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha\,(u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}\,. \qquad (6.6)$$

This is called the Widrow–Hoff rule, or the delta rule, for online least-squares. For the linear function approximator $U\hat{}_\theta(s)$ in Equation ,we get three simple update rules:

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha\,(u_j(s) - \hat{U}_\theta(s))\,, \\
\theta_1 &\leftarrow \theta_1 + \alpha\,(u_j(s) - \hat{U}_\theta(s))x\,, \\
\theta_2 &\leftarrow \theta_2 + \alpha\,(u_j(s) - \hat{U}_\theta(s))y\,.
\end{aligned}$$

We can apply these rules to the example where $U\hat{}_\theta(1, 1)$ is 0.8 and $u_j(1, 1)$ is 0.4. θ_0, θ_1, and θ_2 are all decreased by 0.4α, which reduces the error for (1,1). Notice that changing the parameters θ in response to an observed transition between two states also changes the values of $U\hat{}_\theta$ for every other state! This is what we mean by saying that function approximation allows a reinforcement learner to generalize from its experiences.

We expect that the agent will learn faster if it uses a function approximator, provided that the hypothesis space is not too large, but includes some functions that are a reasonably good fit to the true utility function. asks you to evaluate the performance of direct utility estimation, both with and without function approximation. The improvement in the 4 × 3 world is noticeable but not dramatic, because this is a very small state space to begin with. The improvement is much greater in a 10 × 10 world with a +1 reward at (10,10). This world is well suited for a linear utility function because the true utility function is smooth and nearly linear. If we put the +1 reward at (5,5), the true utility is more like a pyramid and the function approximator in Equation (21.10) will fail miserably. All is not lost, however! Remember that what matters for linear function approximation is that the function be linear in the parameters—the features themselves can be arbitrary nonlinear functions of the state variables. Hence, we can include a term such as $\theta_3 f_3(x, y) = \theta_3\sqrt{(x - x_g)^2 + (y - y_g)^2}$ that measures the distance to the goal. We can apply these ideas equally well to temporal-difference learners. All we need do is adjust the parameters to try to reduce the temporal difference between successive states. The new versions of the TD and Q-learning equations are given by

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\hat{U}_\theta(s') - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\max_{a'}\hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)]\frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i} \qquad (6.7\ \&6.8)$$

for Q-values. For passive TD learning, the update rule can be shown to converge to the closest

possible4 approximation to the true function when the function approximator is linear in the

parameters. With active learning and nonlinear functions such as neural networks, all bets are off: There are some very simple cases in which the parameters can go off to infinity even though there are good solutions in the hypothesis space. There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.

Function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an observable environment is a supervised learning problem, because the next percept gives the outcome state. Any of the supervised learning methods in Chapter 18 can be used, with suitable adjustments for the fact that we need to predict a complete state description rather than just a Boolean classification or a single real value. For a partially observable environment, the learning problem is much more difficult. If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters, as was described in previous Chapter. Inventing the hidden variables and learning the model structure are still open problems.

6.5 POLICY SEARCH

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

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

$$\pi(s) = \max_a \hat{Q}_\theta(s, a) \ . \tag{6.9}$$

Each Q-function could be a linear function of the parameters θ, as in Equation or it could be a nonlinear function such as a neural network. Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q functions, then policy search results in a process that learns Q-functions. This process is not the same as Q-learning! In Q-learning with function approximation, the algorithm finds a value of θ such that $\hat{Q}_\theta$ is "close" to Q_*, the optimal Q-function. Policy search, on the other hand, finds a value of θ that results in good performance; the values

found by the two methods may differ very substantially. (For example, the approximate Q-function defined by $\hat{Q}_\theta(s, a) = Q^*(s, a)/10$ gives optimal performance, even though it is not at all close to Q^*.) Another clear instance of the difference is the case where $\pi(s)$ is calculated using, say, depth-10 look-ahead search with an approximate utility function $\hat{U}_\theta$. A value of θ that gives good results may be a long way from making $\hat{U}_\theta$ resemble the true utility function.

One problem with policy representations of the kind given in Equation is that the policy is a discontinuous function of the parameters when the actions are discrete. (For a continuous action space, the policy can be a smooth function of the parameters.) That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another. This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult. For this reason, policy search methods often use a stochastic policy representation $\pi_\theta(s, a)$, which specifies the probability of selecting action a in state s. One popular representation is the softmax function

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

Softmax becomes nearly deterministic if one action is much better than the others, but it always gives a differentiable function of θ; hence, the value of the policy (which depends in a continuous fashion on the action selection probabilities) is a differentiable function of θ. Softmax is a generalization of the logistic function to multiple variables.

Now let us look at methods for improving the policy. We start with the simplest case: a deterministic policy and a deterministic environment. Let $\rho(\theta)$ be the policy value, i.e., the expected reward-to-go when $\pi\theta$ is executed. If we can derive an expression for $\rho(\theta)$ in closed form, then we have a standard optimization problem. We can follow the policy gradient vector $\nabla_\theta\rho(\theta)$ provided $\rho(\theta)$ is differentiable. Alternatively, if $\rho(\theta)$ is not available in closed form, we can evaluate $\pi\theta$ simply by executing it and observing the accumulated reward. We can follow the empirical gradient by hill climbing—i.e., evaluating the change in policy value for small increments in each parameter. With the usual caveats, this process will converge to a local optimum in policy space.

When the environment (or the policy) is stochastic, things get more difficult. Suppose we are trying to do hill climbing, which requires comparing $\rho(\theta)$ and $\rho(\theta + \Delta\theta)$ for some small $\Delta\theta$. The problem is that the total reward on each trial may vary widely, so estimates of the policy value from a small number of trials will be quite unreliable; trying to compare two such estimates will be even more unreliable. One solution is simply to run lots of trials, measuring the sample variance and using it to determine that enough trials have been run to get a reliable indication of the direction of improvement for $\rho(\theta)$. Unfortunately, this is impractical for many real problems where each trial may be expensive, time-consuming, and perhaps even dangerous.

For the case of a stochastic policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of the gradient at θ, $\nabla_\theta\rho(\theta)$, directly from the results of trials executed at θ. For simplicity, we will derive this estimate for the simple case of a nonsequential environment in which the reward R(a) is obtained immediately after doing action a in the start state s0. In this case, the policy value is just the expected value of the reward, and we have

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

Now we perform a simple trick so that this summation can be approximated by samples

generated from the probability distribution defined by $\pi_\theta(s_0, a)$. Suppose that we have N

trials in all and the action taken on the jth trial is a_j . Then

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

Thus, the true gradient of the policy value is approximated by a sum of terms involving the gradient of the action-selection probability in each trial. For the sequential case, this generalizes to

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

for each state s visited, where a_j is executed in s on the j^{th} trial and $R_j(s)$ is the total reward received from state s onwards in the j^{th} trial. The resulting algorithm is called REINFORCE (Williams, 1992); it is usually much more effective than hill climbing using lots of trials at each value of θ. It is still much slower than necessary, however.

Consider the following task: given two blackjack5 programs, determine which is best. One way to do this is to have each play against a standard "dealer" for a certain number of hands and then to measure their respective winnings. The problem with this, as we have seen, is that the winnings of each program fluctuate widely depending on whether it receives good or bad cards. An obvious solution is to generate a certain number of hands in advance and have each program play the same set of hands. In this way, we eliminate the measurement error due to differences in the cards received. This idea, called correlated sampling, un-derlies a policy-search algorithm called PEGASUS (Ng and Jordan, 2000). The algorithm is applicable to domains for which a simulator is available so that the "random" outcomes of actions can be repeated. The algorithm works by generating in advance N sequences of random numbers, each of which can be used to run a trial of any policy. Policy search is carried out by evaluating each candidate policy using the same set of random sequences to determine the action outcomes. It can be shown that the number of random sequences required

to ensure that the value of every policy is well estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain.

6.6 APPLICATIONS OF REINFORCEMENT LEARNING

The first significant application of reinforcement learning was also the first significant learning program of any kind—the checkers program written by Arthur Samuel (1959, 1967).Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation (21.12) to update the weights. There were some significant differences, however, between his program and current methods. First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree. This works fine, because it amounts to viewing the state space at a different granularity. A second difference was that the program did not use any observed rewards! That is, the values of terminal states reached in self-play were ignored. This means that it is theoretically possible for Samuel's program not to converge, or to converge on a strategy designed to lose rather than to win. He managed to avoid this fate by insisting that the weight for material advantage should always be positive. Remarkably, this was sufficient to direct the program into areas of weight space corresponding to good checkers play.

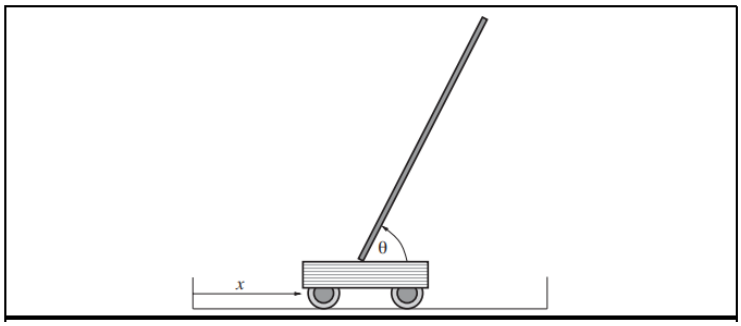


Figure 6.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

Gerry Tesauro's backgammon program TD-GAMMON (1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work (Tesauro and Sejnowski, 1989), Tesauro tried learning a neural network representation of Q(s, a) directly from ex amples of moves labeled with relative values by a human expert. This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts. The TD-GAMMON project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game. The evaluation function was represented by a fully connected neural network with a single hidden layer containing 40 nodes. Simply by repeated application of Equation, TD-GAMMON learned to play considerably better than NEUROGAMMON, even though the input representation contained just

the raw board position with no computed features. This took about 200,000 training games and two weeks of computer time. Although that may seem like a lot of games, it is only a vanishingly small fraction of the state space. When precomputed features were added to the input representation, a network with 80 hidden nodes was able, after 300,000 training games, to reach a standard of play comparable to that of the top three human players worldwide. Kit Woolsey, a top player and analyst, said that "There is no question in my mind that its positional judgment is far better than mine."

6.6.1 Application to robot control

The setup for the famous cart–pole balancing problem, also known as the inverted pendulum, is shown in Figure 21.9. The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown. Several thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem. The cart–pole problem differs from the problems described earlier in that the state variables x, θ, $\dot{x}$, and $\dot{\theta}$ are continuous. The actions are usually discrete: jerk left or jerk right, the so-called bang-bang control regime.

The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only about 30 trials. Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation. The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track. Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence. It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect. Improved generalization and faster learning can be obtained using an algorithm that adaptively partitions the state space according to the observed variation in the reward, or by using a continuous-state, nonlinear function approximator such as a neural network. Nowadays, balancing a triple inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans

Figure 6.10 Superimposed time-lapse images of an autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is

under the control of a policy developed by the PEGASUS policy-search algorithm. A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

SUMMARY

This chapter has examined the reinforcement learning problem: how an agent can become proficient in an unknown environment, given only its percepts and occasional rewards. Reinforcement learning can be viewed as a microcosm for the entire AI problem, but it is studied in a number of simplified settings to facilitate progress.

EXERCISE

1. What is reinforcement learning? State its applications
2. Give illustration of adaptive reinforcement learning.
3. Write a note on Adaptive dynamic programming.
4. Give the illustration of passive reinforcement learning.
5. Write a note on policy search.

REFERENCE

- Artificial Intelligence: A Modern Approach Third Edition
- Artificial Intelligence: Foundations of Computational Agents, David L Poole,Alan K. Mackworth, 2nd Edition, Cambridge University Press ,2017.
- Artificial Intelligence, Kevin Knight and Elaine Rich, 3rd Edition, 2017
- The Elements of Statistical Learning, Trevor Hastie, Robert Tibshirani and Jerome Friedman, Springer, 2013

❄❄❄❄❄❄❄