ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING (Effective from the academic year 2018 -2019) SEMESTER – VII			
Course Code	**18CS71**	**CIE Marks**	40
Number of Contact Hours/Week	4:0:0	**SEE Marks**	60
Total Number of Contact Hours	50	**Exam Hours**	03
CREDITS –4			

Course Learning Objectives: This course (18CS71) will enable students to:

- Explain Artificial Intelligence and Machine Learning
- Illustrate AI and ML algorithm and their use in appropriate applications

Module 1	**Contact Hours**
What is artificial intelligence?, Problems, problem spaces and search, Heuristic search techniques **Texbook 1: Chapter 1, 2 and 3** **RBT: L1, L2**	10
Module 2	
Knowledge representation issues, Predicate logic, Representaiton knowledge using rules. Concpet Learning: Concept learning task, Concpet learning as search, Find-S algorithm, Candidate Elimination Algorithm, Inductive bias of Candidate Elimination Algorithm. **Texbook 1: Chapter 4, 5 and 6** **Texbook2: Chapter 2 (2.1-2.5, 2.7)** **RBT: L1, L2, L3**	10
Module 3	
Decision Tree Learning: Introduction, Decision tree representation, Appropriate problems, ID3 algorith. Aritificil Nueral Network: Introduction, NN representation, Appropriate problems, Perceptrons, Backpropagation algorithm. **Texbook2: Chapter 3 (3.1-3.4), Chapter 4 (4.1-4.5)** **RBT: L1, L2, L3**	10
Module 4	
Bayesian Learning: Introduction, Bayes theorem, Bayes theorem and concept learning, ML and LS error hypothesis, ML for predicting, MDL principle, Bates optimal classifier, Gibbs algorithm, Navie Bayes classifier, BBN, EM Algorithm **Texbook2: Chapter 6** **RBT: L1, L2, L3**	10
Module 5	
Instance-Base Learning: Introduction, k-Nearest Neighbour Learning, Locally weighted regression, Radial basis function, Case-Based reasoning. Reinforcement Learning: Introduction, The learning task, Q-Learning. **Texbook 1: Chapter 8 (8.1-8.5), Chapter 13 (13.1 – 13.3)** **RBT: L1, L2, L3**	10

Course Outcomes: The student will be able to :

- Appaise the theory of Artificial intelligence and Machine Learning.
- Illustrate the working of AI and ML Algorithms.
- Demonstrate the applications of AI and ML.

Question Paper Pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks

- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Textbooks:

1. Tom M Mitchell,**"Machine Lerning",**1st Edition, McGraw Hill Education, 2017.
2. Elaine Rich, Kevin K and S B Nair, **"Artificial Inteligence",** 3rd Edition, McGraw Hill Education, 2017.

Reference Books:

1. Saroj Kaushik, Artificial Intelligence, Cengage learning
2. Stuart Rusell, Peter Norving , Artificial Intelligence: A Modern Approach, Pearson Education 2nd Edition
3. AurÈlienGÈron,"Hands-On Machine Learning with Scikit-Learn and Tensor Flow: Concepts, Tools, and Techniques to Build Intelligent Systems", 1st Edition, Shroff/O'Reilly Media, 2017.
4. Trevor Hastie, Robert Tibshirani, Jerome Friedman, h The Elements of Statistical Learning, 2nd edition, springer series in statistics.
5. Ethem Alpaydın, Introduction to machine learning, second edition, MIT press
6. Srinvivasa K G and Shreedhar, " Artificial Intelligence and Machine Learning", Cengage

ARTIFICIAL INTELLIGENCE

MODULE-1

You will be learning

- ❖What is artificial intelligence?
- ❖Problems
- ❖problem spaces and search
- ❖Heuristic search techniques

What is Artificial Intelligence?

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 biologically observable

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 think 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.

From a **programming** perspective, AI includes the study of symbolic programming, problem solving, and search.

AI Vocabulary

Intelligence relates to tasks involving higher mental processes, e.g. creativity, solving problems, pattern recognition, classification, learning, induction, deduction, building analogies, optimization, language processing, knowledge and many more. Intelligence is the computational part of the ability to achieve goals.

Intelligent behaviour is depicted by perceiving one's environment, acting in complex environments, learning and understanding from experience, reasoning to solve problems and discover hidden knowledge, applying knowledge successfully in new situations, thinking abstractly, using analogies, communicating with others and more.

Science based goals of AI pertain to developing concepts, mechanisms and understanding biological intelligent behaviour. The emphasis is on understanding intelligent behaviour.

Engineering based goals of AI relate to developing concepts, theory and practice of building intelligent machines. The emphasis is on system building.

AI Techniques depict how we represent, manipulate and reason with knowledge in order to solve problems. Knowledge is a collection of 'facts'. To manipulate these facts by a program, a suitable representation is required. A good representation facilitates problem solving.

Learning means that programs learn from what facts or behaviour can represent. Learning denotes changes in the systems that are adaptive in other words, it enables the system to do the same task(s) more efficiently next time.

Applications of AI refers to problem solving, search and control strategies, speech recognition, natural language understanding, computer vision, expert systems, etc.

Problems of AI:

Intelligence does not imply perfect understanding; every intelligent being has limited perception, memory and computation. Many points on the spectrum of intelligence versus cost are viable, from insects to humans. AI seeks to understand the computations required from intelligent behaviour and to produce computer systems that exhibit intelligence. Aspects of intelligence studied by AI include perception, communicational using human languages, reasoning, planning, learning and memory.
The following questions are to be considered before we can step forward:
1. What are the underlying assumptions about intelligence?
2. What kinds of techniques will be useful for solving AI problems?
3. At what level human intelligence can be modelled?
4. When will it be realized when an intelligent program has been built?

Branches of AI:

A list of branches of AI is given below. However some branches are surely missing, because no one has identified them yet. Some of these may be regarded as concepts or topics rather than full branches.

Logical AI — 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 — Artificial Intelligence programs often examine large numbers of possibilities – for example, moves in a chess game and inferences by a theorem proving program. Discoveries are frequently made about how to do this more efficiently in various domains.

Pattern Recognition — When a program makes observations of some kind, it is often planned 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 are like a natural language text, a chess position or in the history of some event. These more complex patterns require quite different methods than do the simple patterns that have been studied the most.

Representation — Usually languages of mathematical logic are used to represent the facts about the world.

Inference — Others can be inferred from some facts. Mathematical logical deduction is sufficient for some purposes, but new methods of *non-monotonic* inference have been added to the 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 divergent. For example, when we hear of a bird, we 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. Normal logical reasoning is monotonic, in that the set of conclusions can be drawn from a set of premises, i.e. monotonic increasing function of the premises. Circumscription is another form of non-monotonic reasoning.

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

Learning from experience — There are some rules expressed in logic for learning. Programs can only learn what facts or behaviour their formalisms can represent, and unfortunately learning systems are almost all based on very limited abilities to represent information.

Planning — Planning starts 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, planning programs 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. Ontology assumed importance from the 1990s.

Heuristics — A heuristic is a way of trying to discover something or an idea embedded in a program. The term is used variously in AI. *Heuristic functions* are used in some approaches to search or 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, and may be more useful.

Genetic programming — Genetic programming is an automated method for creating a working computer program from a high-level problem statement of a problem. Genetic programming starts from a high-level statement of 'what needs to be done' and automatically creates a computer program to solve the problem.

Applications of AI

AI has applications in all fields of human study, such as finance and economics, environmental engineering, chemistry, computer science, and so on. Some of the applications of AI are listed below:

- Perception
    - ■ Machine vision
    - ■ Speech understanding
    - ■ Touch (*tactile* or *haptic*) sensation
- Robotics
- Natural Language Processing
    - ■ Natural Language Understanding
    - ■ Speech Understanding
    - ■ Language Generation
    - ■ Machine Translation
- Planning
- Expert Systems
- Machine Learning
- Theorem Proving
- Symbolic Mathematics
- Game Playing

AI Technique:

Artificial Intelligence research during the last three decades has concluded that *Intelligence requires knowledge*. To compensate overwhelming quality, knowledge possesses less desirable properties.

A. It is huge.
B. It is difficult to characterize correctly.
C. It is constantly varying.
D. It differs from data by being organized in a way that corresponds to its application.
E. It is complicated.

An AI technique is a method that exploits knowledge that is represented so that:

- The knowledge captures generalizations that share properties, are grouped together, rather than being allowed separate representation.
- It can be understood by people who must provide it—even though for many programs bulk of the data comes automatically from readings.
- In many AI domains, how the people understand the same people must supply the knowledge to a program.
- It can be easily modified to correct errors and reflect changes in real conditions.
- It can be widely used even if it is incomplete or inaccurate.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must be usually considered.

In order to characterize an AI technique let us consider initially OXO or tic-tac-toe and use a series of different approaches to play the game.

The programs increase in complexity, their use of generalizations, the clarity of their knowledge and the extensibility of their approach. In this way they move towards being representations of AI techniques.

Example-1: Tic-Tac-Toe

1.1 The first approach (simple)

The Tic-Tac-Toe game consists of a nine element vector called BOARD; it represents the numbers 1 to 9 in three rows.

1	2	3
4	5	6
7	8	9

An element contains the value 0 for blank, 1 for X and 2 for O. A MOVETABLE vector consists of 19,683 elements (3^9) and is needed where each element is a nine element vector. The contents of the vector are especially chosen to help the algorithm.

The algorithm makes moves by pursuing the following:

1. View the vector as a ternary number. Convert it to a decimal number.
2. Use the decimal number as an index in MOVETABLE and access the vector.
3. Set BOARD to this vector indicating how the board looks after the move. This approach is capable in time but it has several disadvantages. It takes more space and requires stunning

effort to calculate the decimal numbers. This method is specific to this game and cannot be completed.

1.2 The second approach

The structure of the data is as before but we use 2 for a blank, 3 for an X and 5 for an O. A variable called TURN indicates 1 for the first move and 9 for the last. The algorithm consists of three actions:

MAKE2 which returns 5 if the centre square is blank; otherwise it returns any blank non-corner square, i.e. 2, 4, 6 or 8. POSSWIN (p) returns 0 if player p cannot win on the next move and otherwise returns the number of the square that gives a winning move.

It checks each line using products 3*3*2 = 18 gives a win for X, 5*5*2=50 gives a win for O, and the winning move is the holder of the blank. GO (n) makes a move to square n setting BOARD[n] to 3 or 5.

This algorithm is more involved and takes longer but it is more efficient in storage which compensates for its longer time. It depends on the programmer's skill.

1.3 The final approach

The structure of the data consists of BOARD which contains a nine element vector, a list of board positions that could result from the next move and a number representing an estimation of how the board position leads to an ultimate win for the player to move.

This algorithm looks ahead to make a decision on the next move by deciding which the most promising move or the most suitable move at any stage would be and selects the same.

Consider all possible moves and replies that the program can make. Continue this process for as long as time permits until a winner emerges, and then choose the move that leads to the computer program winning, if possible in the shortest time.

Actually this is most difficult to program by a good limit but it is as far that the technique can be extended to in any game. This method makes relatively fewer loads on the programmer in terms of the game technique but the overall game strategy must be known to the adviser.

Example-2: Question Answering

Let us consider Question Answering systems that accept input in English and provide answers also in English. This problem is harder than the previous one as it is more difficult to specify the problem properly. Another area of difficulty concerns deciding whether the answer obtained is correct, or not, and further what is meant by 'correct'. For example, consider the following situation:

2.1 Text

Rani went shopping for a new Coat. She found a red one she really liked.
When she got home, she found that it went perfectly with her favourite dress.

2.2 Question

1. What did Rani go shopping for?
2. What did Rani find that she liked?
3. Did Rani buy anything?
4.

Method 1

2.3 Data Structures

A set of templates that match common questions and produce patterns used to match against inputs. Templates and patterns are used so that a template that matches a given question is associated with the corresponding pattern to find the answer in the input text. For example, the template who did **x y** generates **x y z** if a match occurs and **z** is the answer to the question. The given text and the question are both stored as strings.

2.4 Algorithm

Answering a question requires the following four steps to be followed:

- Compare the template against the questions and store all successful matches to produce a set of text patterns.
- Pass these text patterns through a substitution process to change the person or voice and produce an expanded set of text patterns.
- Apply each of these patterns to the text; collect all the answers and then print the answers.

2.5 Example

In **question 1** we use the template WHAT DID X Y which generates Rani go shopping for **z** and after substitution we get Rani goes shopping for **z** and Rani went shopping for **z** giving **z** [equivalence] a new coat

In **question 2** we need a very large number of templates and also a scheme to allow the insertion of ‘find’ before ‘that she liked’; the insertion of ‘really’ in the text; and the substitution of ‘she’ for ‘Rani’ gives the answer ‘a red one’.

Question 3 cannot be answered.

2.6 Comments

This is a very primitive approach basically not matching the criteria we set for intelligence and worse than that, used in the game. Surprisingly this type of technique was actually used in ELIZA which will be considered later in the course.

Method 2

2.7 Data Structures

A structure called English consists of a dictionary, grammar and some semantics about the vocabulary we are likely to come across. This data structure provides the knowledge to convert English text into a storable internal form and also to convert the response back into English. The structured representation of the text is a processed form and defines the context of the input text by making explicit all references such as pronouns. There are three types of such *knowledge representation* systems: production rules of the form 'if x then y', slot and filler systems and statements in mathematical logic. The system used here will be the slot and filler system.

Take, for example sentence:

'She found a red one she really liked'.

Event2		**Event2**	
instance:	finding	instance:	liking
tense:	past	tense:	past
agent:	Rani	modifier:	much
object:	Thing1	object:	Thing1

Thing1	
instance:	coat
colour:	red

2.8 Algorithm

- Convert the question to a structured form using English know how, then use a marker to indicate the substring (like 'who' or 'what') of the structure, that should be returned as an answer. If a slot and filler system is used a special marker can be placed in more than one slot.
- The answer appears by matching this structured form against the structured text.
- The structured form is matched against the text and the requested segments of the question are returned.

2.9 Examples

Both questions 1 and 2 generate answers via a new coat and a red coat respectively. Question 3 cannot be answered, because there is no direct response.

2.10 Comments

This approach is more meaningful than the previous one and so is more effective. The extra power given must be paid for by additional search time in the knowledge bases. A warning must be given here: that is – to generate unambiguous English knowledge base is a complex task and must be left until later in the course. The problems of handling pronouns are difficult.

For example:

Rani walked up to the salesperson: she asked where the toy department was.
Rani walked up to the salesperson: she asked her if she needed any help.

Whereas in the original text the linkage of 'she' to 'Rani' is easy, linkage of 'she' in each of the above sentences to Rani and to the salesperson requires additional knowledge about the context via the people in a shop.

Method 3

2.11 Data Structures

World model contains knowledge about objects, actions and situations that are described in the input text. This structure is used to create integrated text from input text. The diagram shows how the system's knowledge of shopping might be represented and stored. This information is known as a script and in this case is a shopping script. (**See figure 1.1 next page)**

1.8.2.12 Algorithm

Convert the question to a structured form using both the knowledge contained in Method 2 and the World model, generating even more possible structures, since even more knowledge is being used. Sometimes filters are introduced to prune the possible answers.

To answer a question, the scheme followed is: Convert the question to a structured form as before but use the world model to resolve any ambiguities that may occur. The structured form is matched against the text and the requested segments of the question are returned.

2.13 Example

Both questions 1 and 2 generate answers, as in the previous program. Question 3 can now be answered. The shopping script is instantiated and from the last sentence the path through step 14 is the one used to form the representation. 'M' is bound to the red coat-got home. '**Rani buys a red coat'** comes from step 10 and the integrated text generates that she bought a red coat.

2.14 Comments

This program is more powerful than both the previous programs because it has more knowledge. Thus, like the last game program it is exploiting AI techniques. However, we are not yet in a position to handle any English question. The major omission is that of a general reasoning mechanism known as inference to be used when the required answer is not explicitly given in the input text. But this approach can handle, with some modifications, questions of the following form with the answer—Saturday morning Rani went shopping. Her brother tried to call her but she did not answer.

Question: Why couldn't Rani's brother reach her?

Answer: Because she was not in.

This answer is derived because we have supplied an additional fact that a person cannot be in two places at once. This patch is not sufficiently general so as to work in all cases and does not provide the type of solution we are really looking for.

Shopping Script: C - Customer, S - Salesperson

Props: M - Merchandize, D - Money-dollars, Location: L - a Store.

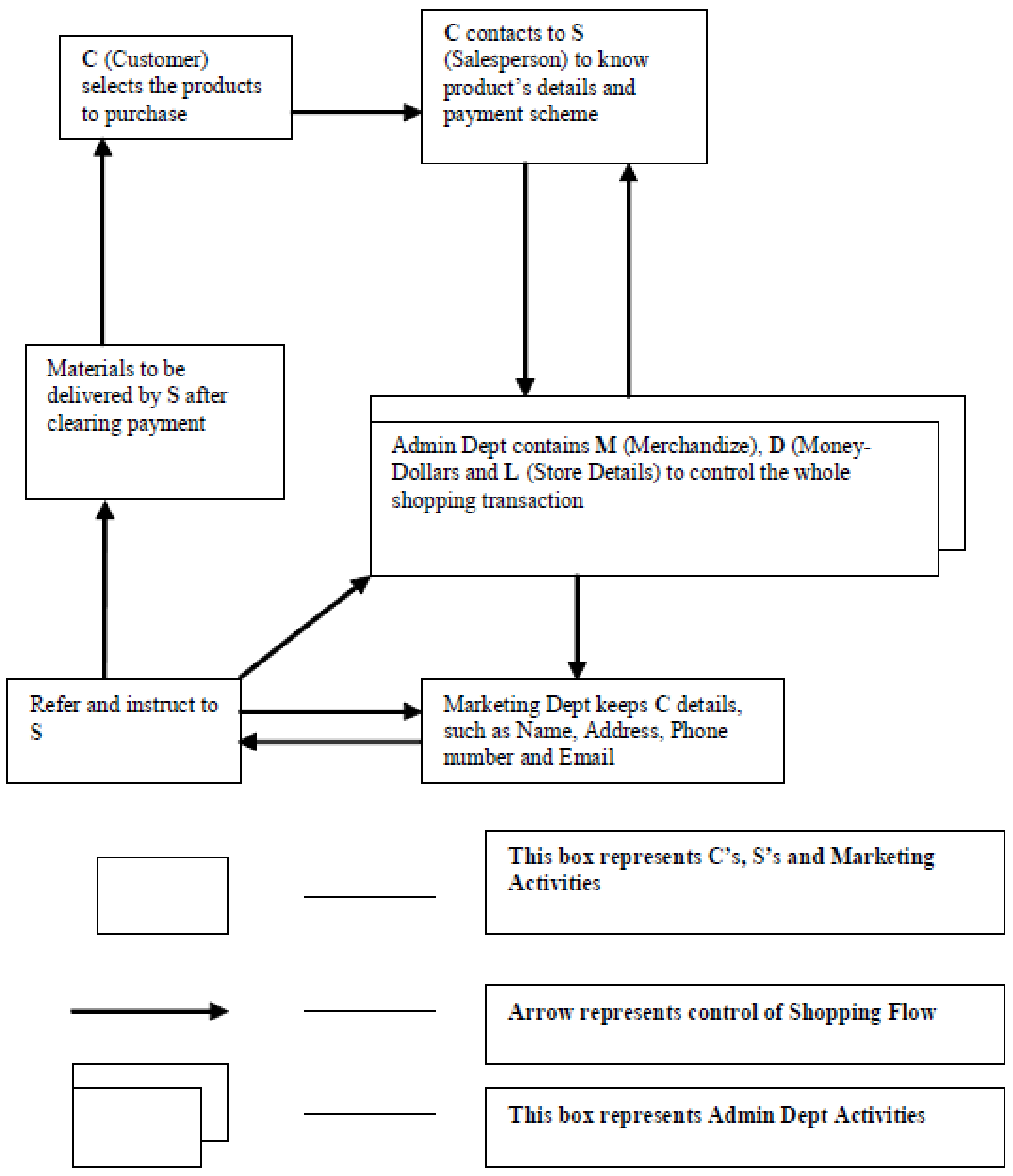

C (Customer) selects the products to purchase
C contacts to S (Salesperson) to know product's details and payment scheme
Materials to be delivered by S after clearing payment
Admin Dept contains M (Merchandize), D (Money-Dollars and L (Store Details) to control the whole shopping transaction
Refer and instruct to S
Marketing Dept keeps C details, such as Name, Address, Phone number and Email
This box represents C's, S's and Marketing Activities
Arrow represents control of Shopping Flow
This box represents Admin Dept Activities

Fig. 1.1 Diagrammatic Representation of Shopping Script

LEVEL OF THE AI MODEL

‘What is our goal in trying to produce programs that do the intelligent things that people do?’

Are we trying to produce programs that do the tasks the same way that people do?

OR

Are we trying to produce programs that simply do the tasks the easiest way that is possible?

Programs in the first class attempt to solve problems that a computer can easily solve and do not usually use AI techniques. AI techniques usually include a search, as no direct method is available, the use of knowledge about the objects involved in the problem area and abstraction on which allows an element of pruning to occur, and to enable a solution to be found in real time; otherwise, the data could explode in size. Examples of these trivial problems in the first class, which are now of interest only to psychologists are EPAM (Elementary Perceiver and Memorizer) which memorized garbage syllables.

The second class of problems attempts to solve problems that are non-trivial for a computer and use AI techniques. We wish to model human performance on these:

1. To test psychological theories of human performance. Ex. PARRY [Colby, 1975] – a program to simulate the conversational behavior of a paranoid person.
2. To enable computers to understand human reasoning – for example, programs that answer questions based upon newspaper articles indicating human behavior.
3. To enable people to understand computer reasoning. Some people are reluctant to accept computer results unless they understand the mechanisms involved in arriving at the results.
4. To exploit the knowledge gained by people who are best at gathering information. This persuaded the earlier workers to simulate human behavior in the SB part of AISB simulated behavior. Examples of this type of approach led to GPS (General Problem Solver).

Questions for Practice:

1. What is *intelligence*? How do we measure it? Are these measurements useful?
2. When the temperature falls and the thermostat turns the heater on, does it act because it *believes* the room to be too cold? Does it *feel* cold? What sorts of things can have beliefs or feelings? Is this related to the idea of consciousness?
3. Some people believe that the relationship between your mind (a non-physical thing) and your brain (the physical thing inside your skull) is exactly like the relationship between a computational process (a non-physical thing) and a physical computer. Do you agree?
4. How good are machines at playing chess? If a machine can consistently beat all the best human chess players, does this prove that the machine is *intelligent*?
5. What is AI Technique? Explain Tic-Tac-Toe Problem using AI Technique.

PROBLEMS, PROBLEM SPACES AND SEARCH

To solve the problem of building a system you should take the following steps:

1. Define the problem accurately including detailed specifications and what constitutes a suitable solution.
2. Scrutinize the problem carefully, for some features may have a central affect on the chosen method of solution.
3. Segregate and represent the background knowledge needed in the solution of the problem.
4. Choose the best solving techniques for the problem to solve a solution.

***Problem solving* is a process** of generating solutions from observed data.

- a *'problem'* is characterized by a set of *goals*,
- a set of *objects*, and
- a set of *operations*.

These could be ill-defined and may evolve during problem solving.

- A **'*problem space*'** is an abstract space.
  - ✓ A problem space encompasses all *valid states* that can be generated by the application of any combination of *operators* on any combination of *objects*.
  - ✓ The problem space may contain one or more *solutions*. A solution is a combination of *operations* and *objects* that achieve the *goals*.
- A '***search***' refers to the search for a solution in a problem space.
  - ✓ Search proceeds with different types of '*search control strategies*'.
  - ✓ The *depth-first search and breadth-first search* are the two common *search strategies.*

2.1 AI - General Problem Solving

Problem solving has been the key area of concern for Artificial Intelligence.

Problem solving is a process of generating solutions from observed or given data. It is however not always possible to use direct methods (i.e. go directly from data to solution). Instead, problem solving often needs to use indirect or modelbased methods.

General Problem Solver (GPS) was a computer program created in 1957 by Simon and Newell to build a universal problem solver machine. *GPS* was based on Simon and Newell's theoretical work on logic machines. *GPS* in principle can solve any formalized symbolic problem, such as theorems proof and geometric problems and chess playing. *GPS* solved many simple problems, such as the Towers of Hanoi, that could be sufficiently formalized, but ***GPS could not solve any real-world problems***.

To build a system to solve a particular problem, we need to:

- Define the problem precisely – find input situations as well as final situations for an acceptable solution to the problem

- Analyze the problem – find few important features that may have impact on the appropriateness of various possible techniques for solving the problem
- Isolate and represent task knowledge necessary to solve the problem
- Choose the best problem-solving technique(s) and apply to the particular problem

Problem definitions
A problem is defined by its '*elements*' and their '*relations*'. To provide a formal description of a problem, we need to do the following:

a. Define a *state space* that contains all the possible configurations of the relevant objects, including some impossible ones.
b. Specify one or more states that describe possible situations, from which the problem-solving process may start. These states are called *initial states*.
c. Specify one or more states that would be acceptable solution to the problem.

These states are called *goal states*.

Specify a set of *rules* that describe the actions (*operators*) available.

The problem can then be solved by using the *rules*, in combination with an appropriate *control strategy*, to move through the *problem space* until a *path* from an *initial state* to a *goal state* is found. This process is known as ***'search'***. Thus:

- *Search* is fundamental to the problem-solving process.
- *Search* is a general mechanism that can be used when a more direct method is not known.
- *Search* provides the framework into which more direct methods for solving subparts of a problem can be embedded. A very large number of AI problems are formulated as search problems.
- Problem space

A *problem space* is represented by a directed graph, where *nodes* represent search state and *paths* represent the operators applied to change the *state*.

To simplify search algorithms, it is often convenient to logically and programmatically represent a problem space as a **tree**. A *tree* usually decreases the complexity of a search at a cost. Here, the cost is due to duplicating some nodes on the tree that were linked numerous times in the graph, e.g. node ***B*** and node ***D.***

A *tree is a graph* in which any two vertices are connected by exactly one path. Alternatively, any connected *graph with no cycles is a tree*.

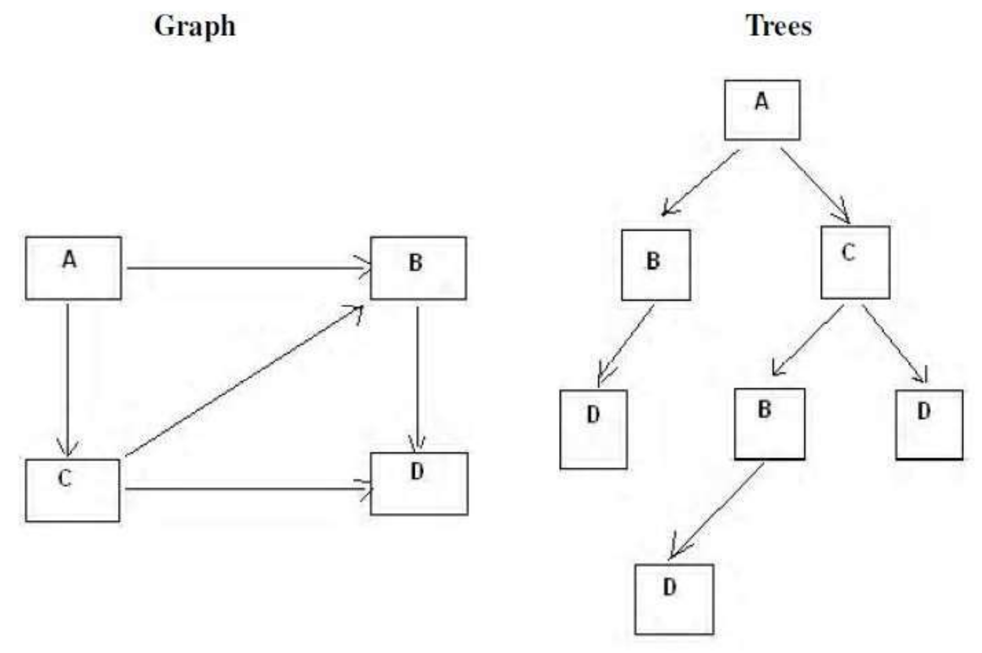


Fig. 2.1 Graph and Tree

• **Problem solving:** The term, Problem Solving relates to analysis in AI. characterized as a systematic search through a range of possible actions to

goal or solution. Problem-solving methods are categorized as *special purpose* and *general purpose*.

• A *special-purpose method* is tailor-made for a particular problem, often exploits very specific features of the situation in which the problem is embedded.

• A *general-purpose method* is applicable to a wide variety of problems. One General-purpose technique used in AI is *'means-end analysis':* a step-bystep, or incremental, reduction of the difference between current state and final goal.

2.3 DEFINING PROBLEM AS A STATE SPACE SEARCH

To solve the problem of playing a game, we require the rules of the game and targets for winning as well as representing positions in the game. The opening position can be defined as the initial state and a winning position as a goal state. Moves from initial state to other states leading to the goal state follow legally. However, the rules are far too abundant in most games— especially in chess, where they exceed the number of particles in the universe. Thus, the rules cannot be supplied accurately and computer programs cannot handle easily. The storage also presents another problem but searching can be achieved by hashing.

The number of rules that are used must be minimized and the set can be created by expressing each rule in a form as possible. The representation of games leads to a state space representation and it is common for well-organized games with some structure. This representation allows for the formal definition of a problem that needs the movement from a set of initial positions to one of a set of target positions. It means that the solution involves using known techniques and a systematic search. This is quite a common method in Artificial Intelligence.

2.3.1 State Space Search

A *state space* represents a problem in terms of *states* and *operators* that change states.
A state space consists of:

- A representation of the *states* the system can be in. For example, in a board game, the board represents the current state of the game.
- A set of *operators* that can change one state into another state. In a board game, the operators are the legal moves from any given state. Often the operators are represented as programs that change a state representation to represent the new state.
- An *initial state*.
- A set of *final states*; some of these may be desirable, others undesirable. This set is often represented implicitly by a program that detects terminal states.

2.3.2 The Water Jug Problem

In this problem, we use two jugs called **four** and **three;** four holds a maximum of four gallons of water and **three** a maximum of three gallons of water. How can we get two gallons of water in the **four** jug?

The state space is a set of prearranged pairs giving the number of gallons of water in the pair of jugs at any time, i.e., (**four, three**) where **fou**r = 0, 1, 2, 3 or 4 and **three** = 0, 1, 2 or 3.

The start state is (0, 0) and the goal state is (2, n) where n may be any but it is limited to **three** holding from 0 to 3 gallons of water or empty. Three and four shows the name and numerical number shows the amount of water in jugs for solving the water jug problem. The major production rules for solving this problem are shown below:

Initial condition	**Goal comment**
1. (four, three) if four < 4	(4, three) fill four from tap
2. (four, three) if three< 3	(four, 3) fill three from tap
3. (four, three) If four > 0	(0, three) empty four into drain
4. (four, three) if three > 0	(four, 0) empty three into drain
5. (four, three) if four + three<4	(four + three, 0) empty three into four
6. (four, three) if four + three<3	(0, four + three) empty four into three
7. (0, three) If three > 0	(three, 0) empty three into four
8. (four, 0) if four > 0	(0, four) empty four into three
9. (0, 2)	(2, 0) empty three into four
10. (2, 0)	(0, 2) empty four into three
11. (four, three) if four < 4	(4, three-diff) pour diff, 4-four, into four from three
12. (three, four) if three < 3	(four-diff, 3) pour diff, 3-three, into three from four and a solution is

given below four three rule

*(**Fig. 2.2** Production Rules for the Water Jug Problem)*

Gallons in Four Jug	Gallons in Three Jug	Rules Applied
0	0	-
	3	2
3	0	7
3	3	2
4	2	11
0	2	3
2	0	10

*(**Fig. 2.3** One Solution to the Water Jug Problem)*

The problem solved by using the production rules in combination with an appropriate control strategy, moving through the problem space until a path from an initial state to a goal state is found. In this problem solving process, search is the fundamental concept. For simple problems it is easier to achieve this goal by hand but there will be cases where this is far too difficult.

2.4 PRODUCTION SYSTEMS

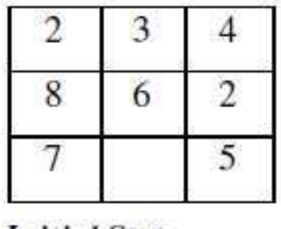
Production systems provide appropriate structures for performing and describing search processes. A production system has four basic components as enumerated below.

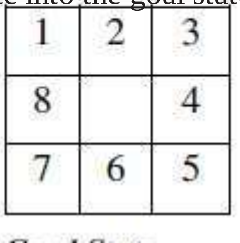
- A set of rules each consisting of a left side that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.
- A database of current facts established during the process of inference.
- A control strategy that specifies the order in which the rules will be compared with facts in the database and also specifies how to resolve conflicts in selection of several rules or selection of more facts.
- A rule firing module.

The production rules operate on the knowledge database. Each rule has a precondition—that is, either satisfied or not by the knowledge database. If the precondition is satisfied, the rule can be applied. Application of the rule changes the knowledge database. The control system chooses which applicable rule should be applied and ceases computation when a termination condition on the knowledge database is satisfied.

Example: Eight puzzle (8-Puzzle)

The 8-puzzle is a 3 × 3 array containing eight square pieces, numbered 1 through 8, and one empty space. A piece can be moved horizontally or vertically into the empty space, in effect exchanging the positions of the piece and the empty space. There are four possible moves, UP (move the blank space up), DOWN, LEFT and RIGHT. The aim of the game is to make a sequence of moves that will convert the board from the start state into the goal state:





This example can be solved by the operator sequence UP, RIGHT, UP, LEFT, DOWN.

Example: Missionaries and Cannibals

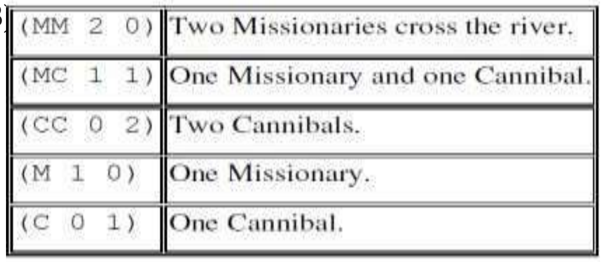
The Missionaries and Cannibals problem illustrates the use of state space search for planning under constraints:

Three missionaries and three cannibals wish to cross a river using a two person boat. If at any time the cannibals outnumber the missionaries on either side of the river, they will eat the missionaries. How can a sequence of boat trips be performed that will get everyone to the other side of the river without any missionaries being eaten?

State representation:

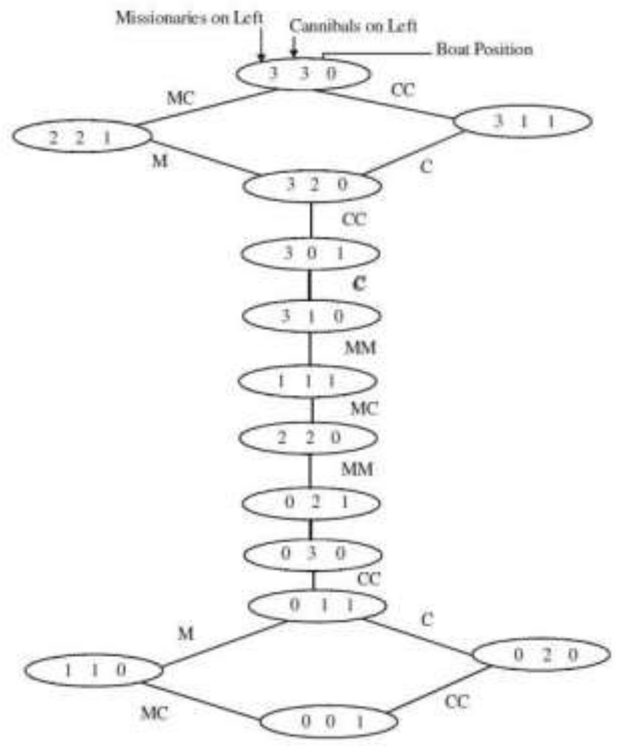
1. BOAT position: original (T) or final (NIL) side of the river.
2. Number of Missionaries and Cannibals on the original side of the river.
3. Start is (T 3 3)

Operators:



(MM 2 0)	Two Missionaries cross the river.
(MC 1 1)	One Missionary and one Cannibal.
(CC 0 2)	Two Cannibals.
(M 1 0)	One Missionary.
(C 0 1)	One Cannibal.

Missionaries/Cannibals Search Graph



2.4.1 Control Strategies

The word '***search***' refers to the search for a solution in a *problem space*.

- Search proceeds with different types of '***search control strategies***'.
- A strategy is defined by picking the order in which the nodes expand.

The Search strategies are evaluated along the following dimensions: Completeness, Time complexity, Space complexity, Optimality (the search- related terms are first explained, and then the search algorithms and control strategies are illustrated next).

Search-related terms

• **Algorithm's performance and complexity**

Ideally we want a common measure so that we can compare approaches in order to select the most appropriate algorithm for a given situation.

✓ *Performance* of an algorithm depends on internal and external factors.

Internal factors/ External factors

- ***Time*** required to run
- ***Size*** of input to the algorithm
- ***Space*** (memory) required to run
- ***Speed*** of the computer
- ***Quality*** of the compiler

✓ *Complexity* is a measure of the performance of an algorithm. *Complexity* measures the internal factors, usually in time than space.

• **Computational complexity\**

It is the measure of resources in terms of ***Time*** and ***Space***.

✓ If ***A*** is an algorithm that solves a decision problem ***f***, then run-time of ***A*** is the number of steps taken on the input of length ***n.***

✓ *Time Complexity* ***T(n)*** of a decision problem ***f*** is the run-time of the 'best' algorithm ***A*** for ***f.***

✓ *Space Complexity* ***S(n)*** of a decision problem ***f*** is the amount of memory used by the 'best' algorithm ***A*** for ***f.***

• '**Big - O**' **notation**

The ***Big-O***, theoretical *measure of the execution of an* algorithm, usually indicates the ***time*** or the ***memory*** needed, given the problem size ***n***, which is usually the number of items.

• ***Big-O*** notation

The ***Big-O*** notation is used to give an approximation to the *run-time- efficiency of an algorithm;* the letter '**O**' is for order of magnitude of operations or space at run-time.

• The ***Big-O*** of an Algorithm ***A***

✓ If an algorithm ***A*** requires time proportional to ***f(n)***, then algorithm ***A*** is said to be of order ***f(n)***, and it is denoted as ***O(f(n))***.

✓ If algorithm ***A*** requires time proportional to ***n2***, then the order of the algorithm is said to be ***O(n2).***

✓ If algorithm ***A*** requires time proportional to ***n***, then the order of the algorithm is said to be ***O(n).***

The function ***f(n)*** is called the algorithm's *growth-rate function*. In other words, if an algorithm has performance complexity ***O(n)***, this means that the run-time ***t*** should be directly proportional to ***n***, ie ***t • n or t = k n*** where ***k*** is constant of proportionality.
Similarly, for algorithms having performance complexity ***O(log2(n)), O(log N), O(N log N), O(2N)*** and so on.

Example 1:
Determine the ***Big-O*** of an algorithm:

Calculate the sum of the ***n*** elements in an integer array ***a[0..n-1]***.

Line no.	**Instructions**	**No of execution steps**
line 1	sum	1
line 2	for (i = 0; i < n; i++)	n + 1
line 3	sum += a[i]	n
line 4	print sum	1
	Total	**2n + 3**

Thus, the polynomial ***(2n + 3)*** is dominated by the 1st term as ***n*** while the number of elements in the array becomes very large.

• In determining the ***Big-O***, ignore constants such as ***2*** and ***3***. So the algorithm is of order ***n***.
• So the ***Big-O*** of the algorithm is ***O(n)***.
• In other words the run-time of this algorithm increases roughly as the size of the input data ***n***, e.g., an array of size ***n***.

Tree structure
Tree is a way of organizing objects, related in a hierarchical fashion.
- Tree is a type of data structure in which each *element* is attached to one or more elements directly beneath it.
- The connections between elements are called *branches*.
- Tree is often called *inverted trees* because it is drawn with the *root* at the top.
- The elements that have no elements below them are called *leaves*.
- A *binary tree* is a special type: each element has only two branches below it.

Properties
- Tree is a special case of a *graph*.
- The topmost node in a tree is called the *root node*.
- At root node all operations on the tree begin.
- A node has at most one parent.
- The topmost node (root node) has no parents.
- Each node has zero or more *child nodes*, which are below it .
- The nodes at the bottommost level of the tree are called *leaf nodes*.
Since *leaf nodes* are at the bottom most level, they do not have children.
- A node that has a child is called the child's *parent node*.
- The *depth of a node* **n** is the length of the path from the root to the node.
- The root node is at depth zero.

• **Stacks and Queues**

The *Stacks* and *Queues* are data structures that maintain the order of *last-in, first-out* and *first-in, first-out* respectively. Both *stacks* and *queues* are often implemented as linked lists, but that is not the only possible implementation.

Stack - Last In First Out (LIFO) lists

- An ordered list; a sequence of items, piled one on top of the other.
- The ***insertions*** and ***deletions*** are made at one end only, called ***Top***.
- If Stack ***S = (a[1], a[2],........ a[n])*** then ***a[1]*** is bottom most element
- Any intermediate element ***(a[i])*** is on top of element ***a[i-1], 1 < i <= n.***
- In Stack all operation take place on ***Top***.

The ***Pop*** operation removes item from top of the stack.
The ***Push*** operation adds an item on top of the stack.

Queue - First In First Out (FIFO) lists

• An ordered list; a sequence of items; there are restrictions about how items can be added to and removed from the list. A queue has two ends.
• All ***insertions*** (enqueue) take place at one end, called ***Rear*** or ***Back***
• All ***deletions*** (dequeue) take place at other end, called ***Front***.
• If Queue has ***a[n]*** as rear element then ***a[i+1]*** is behind ***a[i] , 1 < i <= n.***
• All operation takes place at one end of queue or the other.

The ***Dequeue*** operation removes the item at ***Front*** of the queue.
The ***Enqueue*** operation adds an item to the ***Rear*** of the queue.

Search

Search is the systematic examination of *states* to find path from the *start / root state* to the *goal state*.

• Search usually results from a lack of knowledge.
• Search explores knowledge alternatives to arrive at the best answer.
• Search algorithm output is a solution, that is, a path from the initial state to a state that satisfies the goal test.

For general-purpose problem-solving – '***Search' is an approach***.

• Search deals with finding *nodes* having certain properties in a *graph* that represents search space.
• Search methods explore the search space 'intelligently', evaluating possibilities without investigating every single possibility.

Examples:
• For a Robot this might consist of PICKUP, PUTDOWN, MOVEFORWARD, MOVEBACK, MOVELEFT, and MOVERIGHT—until the goal is reached.
• Puzzles and Games have explicit rules: e.g., the '***Tower of Hanoi***' puzzle

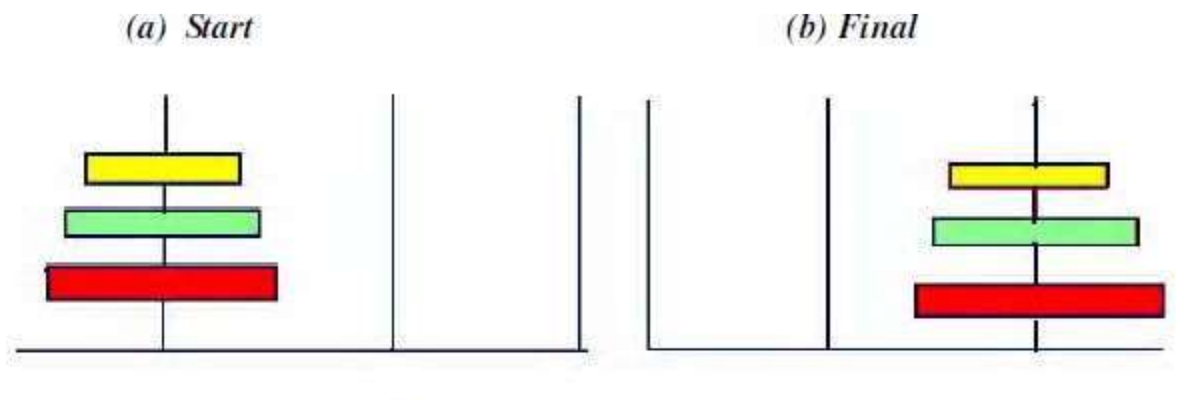


Fig. 2.4 Tower of Hanoi Puzzle

This puzzle involves a set of rings of different sizes that can be placed on three different pegs.
• The puzzle starts with the rings arranged as shown in Figure 2.4(a)
• The goal of this puzzle is to move them all as to Figure 2.4(b)
• Condition: Only the top ring on a peg can be moved, and it may only be placed on a smaller ring, or on an empty peg.

In this *Tower of Hanoi* puzzle:
• Situations encountered while solving the problem are described as *states*.
• Set of all possible configurations of rings on the pegs is called *'problem space'*.
• **States**
A *State* is a representation of elements in a given moment.
A problem is defined by its *elements* and their *relations*.
At each instant of a problem, the elements have specific descriptors and relations; the *descriptors* indicate how to select elements?
Among all possible states, there are two special states called:

- ✓ *Initial state* – the start point
- ✓ *Final state* – the goal state

• **State Change:** Successor Function
A '*successor function*' is needed for state change. The Successor Function moves one state to another state.
Successor Function:

- ✓ It is a description of possible actions; a set of operators.
- ✓ It is a transformation function on a state representation, which converts that state into another state.
- ✓ It defines a relation of accessibility among states.
- ✓ It represents the conditions of applicability of a state and corresponding transformation function.

• **State space**
A *state space* is the set of all *states* reachable from the *initial state*.

- ✓ A *state space* forms a *graph* (or map) in which the *nodes* are states and the *arcs* between nodes are actions.
- ✓ In a *state space*, a *path* is a sequence of states connected by a sequence of actions.
- ✓ The *solution* of a problem is part of the map formed by the *state space*.

• **Structure of a state space**
The *structures* of a *state space* are *trees* and *graphs***.**
- ✓ A *tree* is a hierarchical structure in a graphical form.
- ✓ A *graph* is a non-hierarchical structure.

• A *tree* has only one path to a given node;
i.e., a *tree* has one and only one path from any point to any other point.
• A *graph* consists of a set of nodes (vertices) and a set of edges (arcs). Arcs establish relationships (connections) between the nodes; i.e., a graph has several paths to a given node.
• The *Operators* are directed *arcs* between nodes.
A *search* process explores the *state space*. In the worst case, the search explores all possible *paths* between the *initial state* and the *goal state*.
• **Problem solution**
In the *state space*, a *solution* is a path from the *initial state* to a *goal state* or, sometimes, just a *goal state*.
- ✓ A *solution cost function* assigns a numeric cost to each *path*; it also gives the cost of applying the *operators* to the *states*.
- ✓ A *solution quality* is measured by the *path cost function*; and an optimal solution has the lowest path cost among all solutions.
- ✓ The *solutions* can be *any or optimal or all*.
- ✓ The importance of cost depends on the problem and the type of solution asked

• **Problem description**
A problem consists of the description of:
- ✓ The current state of the world,
- ✓ The actions that can transform one state of the world into another,
- ✓ The desired state of the world.

The following action one taken to describe the problem:
- ✓ *State space* is defined explicitly or implicitly

A *state space* should describe everything that is needed to solve a problem and nothing that is not needed to solve the problem.
- ✓ *Initial state* is start state
- ✓ *Goal state* is the conditions it has to fulfill.

The description by a desired state may be complete or partial.
- ✓ *Operators* are to change state
- ✓ Operators do actions that can transform one state into another;
- ✓ Operators consist of: Preconditions and Instructions;

Preconditions provide partial description of the state of the world that must be true in order to perform the action, and
Instructions tell the user how to create the next state.
- Operators should be as general as possible, so as to reduce their number.
- *Elements of the domain* has relevance to the problem
  - ✓ Knowledge of the starting point.
- *Problem solving* is finding a solution
  - ✓ Find an ordered sequence of operators that transform the current (start) state into a goal state.

- *Restrictions* are solution quality any, optimal, or all
    - ✓ Finding the shortest sequence, or
    - ✓ finding the least expensive sequence defining cost, or
    - ✓ finding any sequence as quickly as possible.

This can also be explained with the help of algebraic function as given below.

PROBLEM CHARACTERISTICS

Heuristics cannot be generalized, as they are domain specific. Production systems provide ideal techniques for representing such heuristics in the form of IF-THEN rules. Most problems requiring simulation of intelligence use heuristic search extensively. Some heuristics are used to define the control structure that guides the search process, as seen in the example described above. But heuristics can also be encoded in the rules to represent the domain knowledge. Since most AI problems make use of knowledge and guided search through the knowledge, AI can be described as *the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about problem domain*.

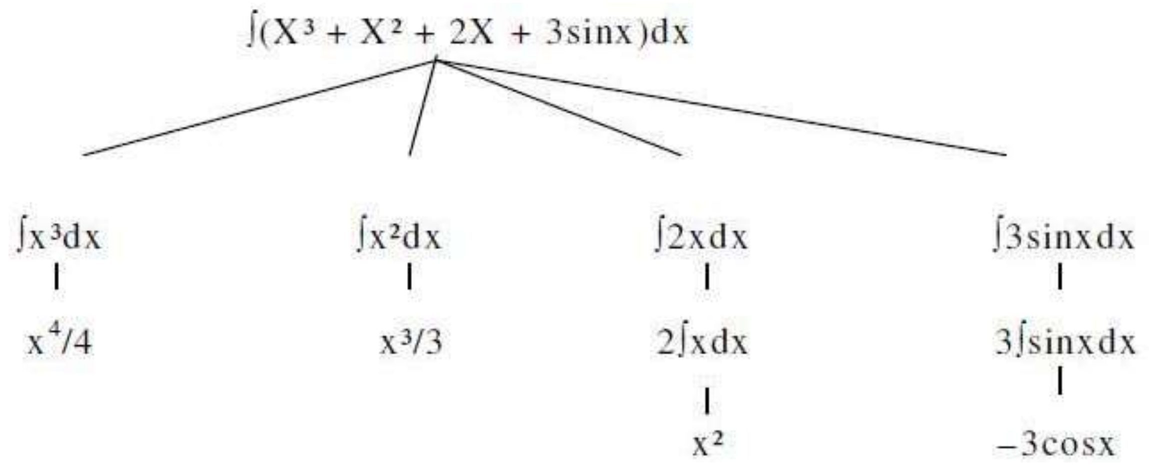
To use the heuristic search for problem solving, we suggest analysis of the problem for the following considerations:

- Decomposability of the problem into a set of independent smaller subproblems
- Possibility of undoing solution steps, if they are found to be unwise
- Predictability of the problem universe
- Possibility of obtaining an obvious solution to a problem without comparison of all other possible solutions
- Type of the solution: whether it is a state or a path to the goal state
- Role of knowledge in problem solving
- Nature of solution process: with or without interacting with the user

The general classes of engineering problems such as planning, classification, diagnosis, monitoring and design are generally knowledge intensive and use a large amount of heuristics. Depending on the type of problem, the knowledge representation schemes and control strategies for search are to be adopted. Combining heuristics with the two basic search strategies have been discussed above. There are a number of other general-purpose search techniques which are essentially heuristics based. Their efficiency primarily depends on how they exploit the domain-specific knowledge to abolish undesirable paths. Such search methods are called 'weak methods', since the progress of the search depends heavily on the way the domain knowledge is exploited. A few of such search techniques which form the centre of many AI systems are briefly presented in the following sections.

Problem Decomposition

Suppose to solve the expression is: + $\int(X^3 + X^2 + 2X + 3\sin x)\,dx$



This problem can be solved by breaking it into smaller problems, each of which we can solve by using a small collection of specific rules. Using this technique of problem decomposition, we can solve very large problems very easily. This can be considered as an intelligent behaviour.

Can Solution Steps be Ignored?

Suppose we are trying to prove a mathematical theorem: first we proceed considering that proving a lemma will be useful. Later we realize that it is not at all useful. We start with another one to prove the theorem. Here we simply ignore the first method.
Consider the 8-puzzle problem to solve: we make a wrong move and realize that mistake. But here, the control strategy must keep track of all the moves, so that we can backtrack to the initial state and start with some new move.
Consider the problem of playing chess. Here, once we make a move we never recover from that step. These problems are illustrated in the three important classes of problems mentioned below:

1. Ignorable, in which solution steps can be ignored. Eg: Theorem Proving
2. Recoverable, in which solution steps can be undone. Eg: 8-Puzzle
3. Irrecoverable, in which solution steps cannot be undone. Eg: Chess

Is the Problem Universe Predictable?

Consider the 8-Puzzle problem. Every time we make a move, we know exactly what will happen. This means that it is possible to plan an entire sequence of moves and be confident what the resulting state will be. We can backtrack to earlier moves if they prove unwise.

Suppose we want to play Bridge. We need to plan before the first play, but we cannot play with certainty. So, the outcome of this game is very uncertain. In case of 8-Puzzle, the outcome is very certain. To solve uncertain outcome problems, we follow the process of plan revision as the plan is carried out and the necessary feedback is provided. The disadvantage is that the planning in this case is often very expensive.

Is Good Solution Absolute or Relative?
Consider the problem of answering questions based on a database of simple facts such as the following:

1. Siva was a man.
2. Siva was a worker in a company.
3. Siva was born in 1905.
4. All men are mortal.
5. All workers in a factory died when there was an accident in 1952.
6. No mortal lives longer than 100 years.

Suppose we ask a question: 'Is Siva alive?'

By representing these facts in a formal language, such as predicate logic, and then using formal inference methods we can derive an answer to this question easily.

There are two ways to answer the question shown below:

Method I:

1. Siva was a man.
2. Siva was born in 1905.
3. All men are mortal.
4. Now it is 2008, so Siva's age is 103 years.
5. No mortal lives longer than 100 years.

Method II:

1. Siva is a worker in the company.
2. All workers in the company died in 1952.

Answer: So Siva is not alive. It is the answer from the above methods.

We are interested to answer the question; it does not matter which path we follow. If we follow one path successfully to the correct answer, then there is no reason to go back and check another path to lead the solution.

CHARACTERISTICS OF PRODUCTION SYSTEMS

Production systems provide us with good ways of describing the operations that can be performed in a search for a solution to a problem.

At this time, two questions may arise:

1. Can production systems be described by a set of characteristics? And how can they be easily implemented?
2. What relationships are there between the problem types and the types of production systems well suited for solving the problems?

To answer these questions, first consider the following definitions of classes of production systems:

1. A monotonic production system is a production system in which the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected.
2. A non-monotonic production system is one in which this is not true.
3. A partially communicative production system is a production system with the property that if the application of a particular sequence of rules transforms state P into state Q, then any combination of those rules that is allowable also transforms state P into state Q.
4. A commutative production system is a production system that is both monotonic and partially commutative.

Table 2.1 Four Categories of Production Systems

Production System	Monotonic	Non-monotonic
Partially Commutative	Theorem Proving	Robot Navigation
Non-partially Commutative	Chemical Synthesis	Bridge

Is there any relationship between classes of production systems and classes of problems? For any solvable problems, there exist an infinite number of production systems that show how to find solutions. Any problem that can be solved by any production system can be solved by a commutative one, but the commutative one is practically useless. It may use individual states to represent entire sequences of applications of rules of a simpler, non-commutative system. In the formal sense, there is no relationship between kinds of problems and kinds of production systems Since all problems can be solved by all kinds of systems. But in the practical sense, there is definitely such a relationship between the kinds of problems and the kinds of systems that lend themselves to describing those problems.

Partially commutative, monotonic productions systems are useful for solving ignorable problems. These are important from an implementation point of view without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed. Both types of partially commutative production systems are significant from an implementation point; they tend to lead to many duplications of individual states during the search process. Production systems that are not partially commutative are useful for many problems in which permanent changes occur.

Issues in the Design of Search Programs

Each search process can be considered to be a tree traversal. The object of the search is to find a path from the initial state to a goal state using a tree. The number of nodes generated might be huge; and in practice many of the nodes would not be needed. The secret of a good search routine is to generate only those nodes that are likely to be useful, rather than having a precise tree. The rules are used to represent the tree implicitly and only to create nodes explicitly if they are actually to be of use.

The following issues arise when searching:
• The tree can be searched forward from the initial node to the goal state or backwards from the goal state to the initial state.
• To select applicable rules, it is critical to have an efficient procedure for matching rules against states.
• How to represent each node of the search process? This is the knowledge representation problem or the frame problem. In games, an array suffices; in other problems, more complex data structures are needed.

Finally in terms of data structures, considering the water jug as a typical problem do we use a graph or tree? The breadth-first structure does take note of all nodes generated but the depth-first one can be modified.

Check duplicate nodes

1. Observe all nodes that are already generated, if a new node is present.
2. If it exists add it to the graph.
3. If it already exists, then
   a. Set the node that is being expanded to the point to the already existing node corresponding to its successor rather than to the new one. The new one can be thrown away.

   b. If the best or shortest path is being determined, check to see if this path is better or worse than the old one. If worse, do nothing.

Better save the new path and work the change in length through the chain of successor nodes if necessary.

Example: Tic-Tac-Toe

State spaces are good representations for board games such as Tic-Tac-Toe. The position of a game can be explained by the contents of the board and the player whose turn is next. The board can be represented as an array of 9 cells, each of which may contain an X or O or be empty.
• **State:**
- ✓ Player to move next: X or O.
- ✓ Board configuration:

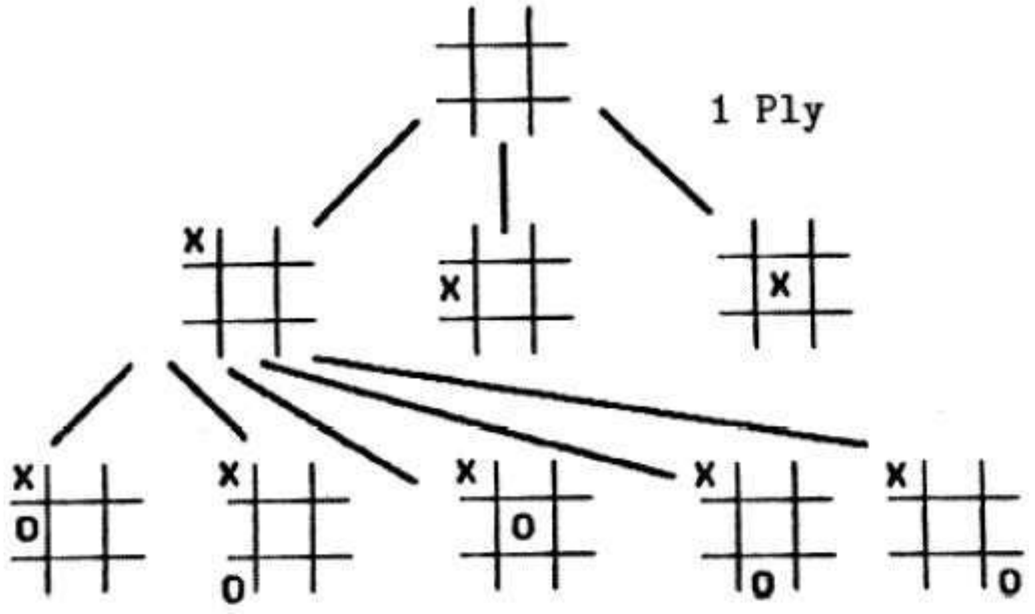
X		0
	0	
X		X

• **Operators:** Change an empty cell to X or O.
• **Start State:** Board empty; X's turn.
• **Terminal States:** Three X's in a row; Three O's in a row; All cells full.

Search Tree

The sequence of states formed by possible moves is called a *search tree*. Each level of the tree is called a *ply*.

Since the same state may be reachable by different sequences of moves, the state space may in general be a graph. It may be treated as a tree for simplicity, at the cost of duplicating states.



Solving problems using search

• Given an informal description of the problem, construct a formal description as a state space:

- ✓ Define a data structure to represent the *state*.
- ✓ Make a representation for the *initial state* from the given data.
- ✓ Write programs to represent *operators* that change a given state representation to a new state representation.
- ✓ Write a program to detect *terminal states*.

• Choose an appropriate search technique:

- ✓ How large is the search space?
- ✓ How well structured is the domain?
- ✓ What knowledge about the domain can be used to guide the search?

HEURISTIC SEARCH TECHNIQUES:

Search Algorithms
Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solutions within some practical time and space limits. Consequently, many special techniques are developed, using ***heuristic functions.***
The algorithms that use *heuristic functions* are called ***heuristic algorithms***.

• Heuristic algorithms are not really intelligent; they appear to be intelligent because they achieve better performance.
• Heuristic algorithms are more efficient because they take advantage of feedback from the data to direct the search path.
• ***Uninformed search algorithms*** or *Brute-force algorithms*, search through the search space all possible candidates for the solution checking whether each candidate satisfies the problem's statement.
• ***Informed search algorithms*** use heuristic functions that are specific to the problem, apply them to guide the search through the search space to try to reduce the amount of time spent in searching.

A good heuristic will make an informed search dramatically outperform any uninformed search: for example, the Traveling Salesman Problem (TSP), where the goal is to find is a good solution instead of finding the best solution.

In such problems, the search proceeds using current information about the problem to predict which path is closer to the goal and follow it, although it does not always guarantee to find the best possible solution. Such techniques help in finding a solution within reasonable time and space (memory). Some prominent intelligent search algorithms are stated below:

1. ***Generate and Test Search***
2. ***Best-first Search***
3. ***Greedy Search***
4. ***A* Search***
5. ***Constraint Search***
6. ***Means-ends analysis***

There are some more algorithms. They are either improvements or combinations of these.
• **Hierarchical Representation of Search Algorithms:** A Hierarchical representation of most search algorithms is illustrated below. The representation begins with two types of search:
• **Uninformed Search:** Also called blind, exhaustive or brute-force search, it uses no information about the problem to guide the search and therefore may not be very efficient.
• **Informed Search:** Also called heuristic or intelligent search, this uses information about the problem to guide the search—usually guesses the distance to a goal state and is therefore efficient, but the search may not be always possible.

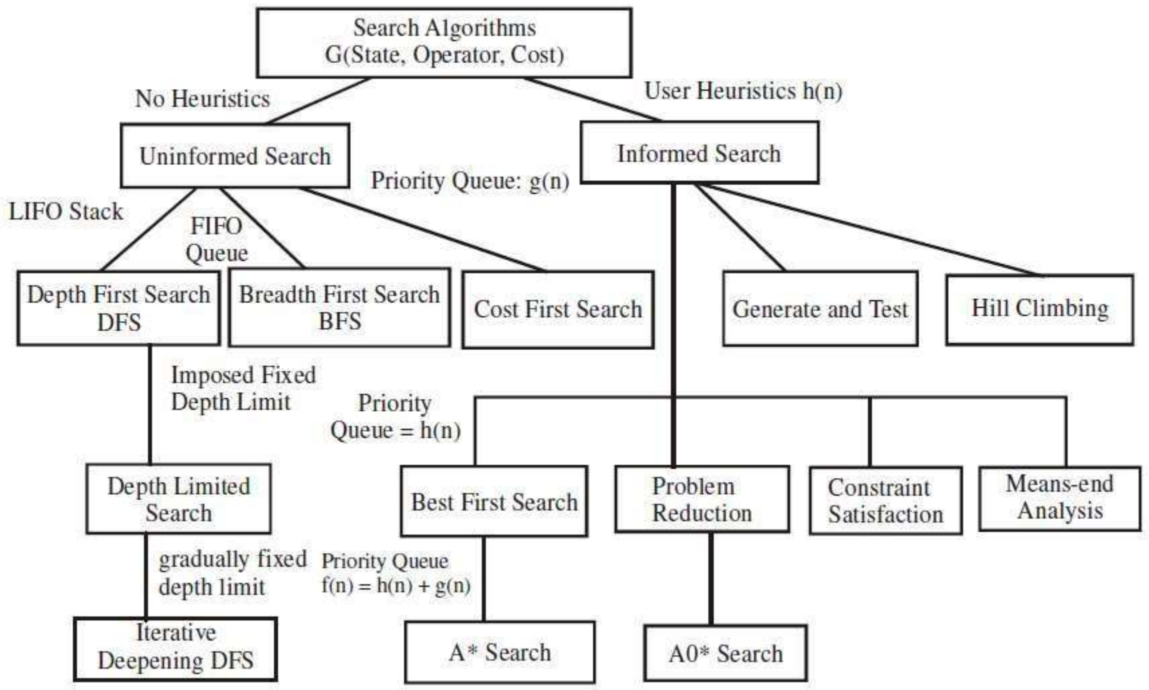
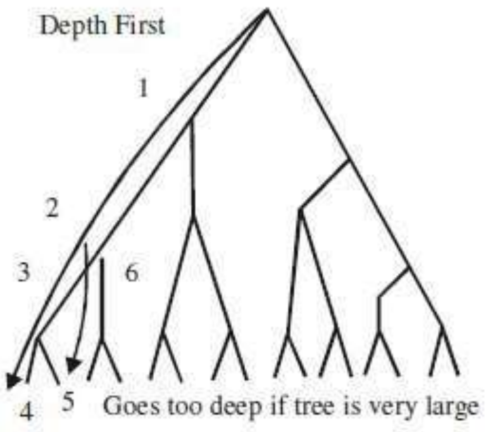
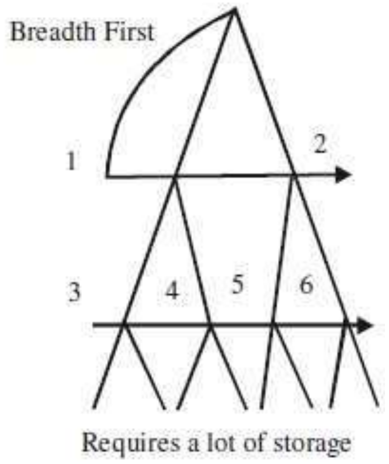
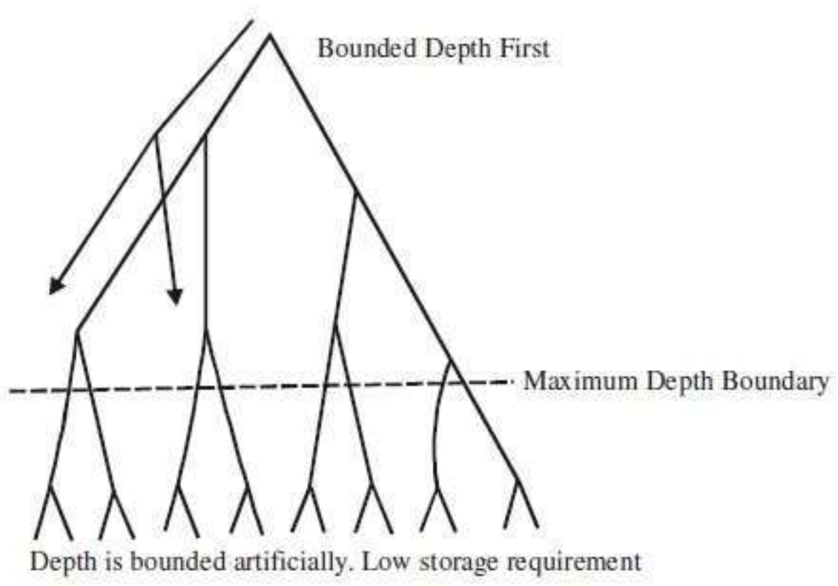


Fig. *Different Search Algorithms*

The first requirement is that it causes motion, in a game playing program, it moves on the board and in the water jug problem, filling water is used to fill jugs. It means the control strategies without the motion will never lead to the solution.
The second requirement is that it is systematic, that is, it corresponds to the need for global motion as well as for local motion. This is a clear condition that neither would it be rational to fill a jug and empty it repeatedly, nor it would be worthwhile to move a piece round and round on the board in a cyclic way in a game. We shall initially consider two systematic approaches for searching. Searches can be classified by the order in which operators are tried: depth-first, breadth-first, bounded depth-first.







Breadth-first search

A Search strategy, in which the highest layer of a decision tree is searched completely before proceeding to the next layer is called *Breadth-first search (BFS)*.

• In this strategy, no viable solutions are omitted and therefore it is guaranteed that an optimal solution is found.

• This strategy is often not feasible when the search space is large.

Algorithm

```
1. Create a variable called LIST and set it to be the starting state.
2. Loop until a goal state is found or LIST is empty, Do
a. Remove the first element from the LIST and call it E. If the LIST is empty, quit.
b. For every path each rule can match the state E, Do
(i) Apply the rule to generate a new state.
(ii) If the new state is a goal state, quit and return this state.
(iii) Otherwise, add the new state to the end of LIST.
```

Advantages
1. Guaranteed to find an optimal solution (in terms of shortest number of steps to reach the goal).
2. Can always find a goal node if one exists (complete).

Disadvantages
1. High storage requirement: *exponential* with tree depth.

Depth-first search
A search strategy that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path is called *Depth-first search (DFS)*.
• This strategy does not guarantee that the optimal solution has been found.
• In this strategy, search reaches a satisfactory solution more rapidly than breadth first, an advantage when the search space is large.

Algorithm
Depth-first search applies operators to each newly generated state, trying to drive directly toward the goal.
1. If the starting state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signalled:
a. Generate a successor E to the starting state. If there are no more successors, then signal failure.
b. Call Depth-first Search with E as the starting state.
c. If success is returned signal success; otherwise, continue in the loop.

Advantages
1. Low storage requirement: *linear* with tree depth.
2. Easily programmed: function call stack does most of the work of maintaining state of the search.

Disadvantages
1. May find a sub-optimal solution (one that is deeper or more costly than the best solution).
2. Incomplete: without a depth bound, may not find a solution even if one exists.

2.4.2.3 Bounded depth-first search
Depth-first search can spend much time (perhaps infinite time) exploring a very deep path that does not contain a solution, when a shallow solution exists. An easy way to solve this problem is to put a maximum depth bound on the search. Beyond the depth bound, a failure is generated automatically without exploring any deeper.
Problems:
1. It’s hard to guess how deep the solution lies.
2. If the estimated depth is too deep (even by 1) the computer time used is dramatically increased, by a factor of *bextra*.
3. If the estimated depth is too shallow, the search fails to find a solution; all that computer time is wasted.

Heuristics
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 entirety 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.

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.

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.

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.
  ✓ Heuristics might (for reasons) *underestimate* or *overestimate* the merit of a state with respect to goal.
  ✓ 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.

Heuristic search compared with other search
The Heuristic search is compared with Brute force or Blind search techniques below:

Comparison of Algorithms

Brute force / Blind search	**Heuristic search**
Can only search what it has knowledge about already	Estimates 'distance' to goal state through explored nodes
No knowledge about how far a node node from goal state	Guides search process toward goal
	Prefers states (nodes) that lead close to and not away from goal state

Example: Travelling salesman
A salesman has to visit a list of cities and he must visit each city only once. There are different routes between the cities. The problem is to find the shortest route between the cities so that the salesman visits all the cities at once.

Suppose there are N cities, then a solution would be to take N! possible combinations to find the shortest distance to decide the required route. This is not efficient as with N=10 there are 36,28,800 possible routes. This is an example of *combinatorial explosion*.

There are better methods for the solution of such problems: one is called *branch* and *bound*. First, generate all the complete paths and find the distance of the first complete path. If the next path is shorter, then save it and proceed this way avoiding the path when its length exceeds the saved shortest path length, although it is better than the previous method.

Generate and Test Strategy

Generate-And-Test Algorithm
Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution.
Algorithm: Generate-And-Test
1. Generate a possible solution.
2. Test to see if this is the expected solution.
3. If the solution has been found quit else go to step 1.

Potential solutions that need to be generated vary depending on the kinds of problems. For some problems the possible solutions may be particular points in the problem space and for some problems, paths from the start state.

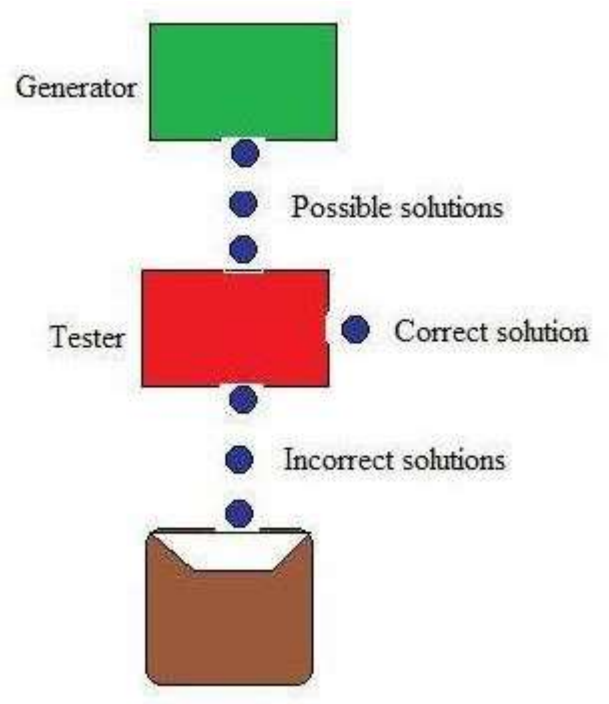


Figure: Generate And Test

Generate-and-test, like depth-first search, requires that complete solutions be generated for testing. In its most systematic form, it is only an exhaustive search of the problem space. Solutions can also be generated randomly but solution is not guaranteed. This approach is what is known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

Systematic Generate-And-Test

While generating complete solutions and generating random solutions are the two extremes there exists another approach that lies in between. The approach is that the search process proceeds systematically but some paths that unlikely to lead the solution are not considered. This evaluation is performed by a heuristic function.

Depth-first search tree with backtracking can be used to implement systematic generate-and-test procedure. As per this procedure, if some intermediate states are likely to appear often in the tree, it would be better to modify that procedure to traverse a graph rather than a tree.

Generate-And-Test And Planning

Exhaustive generate-and-test is very useful for simple problems. But for complex problems even heuristic generate-and-test is not very effective technique. But this may be made effective by combining with other techniques in such a way that the space in which to search is restricted. An AI program DENDRAL, for example, uses plan-Generate-and-test technique. First, the planning process uses constraint-satisfaction techniques and creates lists of recommended and contraindicated substructures. Then the generate-and-test procedure uses the lists generated and required to explore only a limited set of structures. Constrained in this way, generate-and-test proved highly effective. A major weakness of planning is that it often produces inaccurate solutions as there is no feedback from the world. But if it is used to produce only pieces of solutions then lack of detailed accuracy becomes unimportant.

Hill Climbing

Hill Climbing is heuristic search used for mathematical optimization problems in the field of Artificial Intelligence .

Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

- In the above definition, mathematical optimization problems implies that hill climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by salesman.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in reasonable time.
- A heuristic function is a function that will rank all the possible alternatives at any branching step in search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

Features of Hill Climbing

1. Variant of generate and test algorithm : It is a variant of generate and test algorithm. The generate and test algorithm is as follows :

1. Generate a possible solutions.
2. Test to see if this is the expected solution.
3. If the solution has been found quit else go to step 1.

Hence we call Hill climbing as a variant of generate and test algorithm as it takes the feedback from test procedure. Then this feedback is utilized by the generator in deciding the next move in search space.

2. Uses the Greedy approach : At any point in state space, the search moves in that direction only which optimizes the cost of function with the hope of finding the optimal solution at the end.

Types of Hill Climbing

1. Simple Hill climbing : It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node.
   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 state is a goal state, then stop and return success.
ii. If it is better than the 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 is found.

Step 3 : Exit.

2. Steepest-Ascent Hill climbing : It first examines all the neighboring nodes and then selects the node closest to the solution state as 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 examine all the neighboring nodes before deciding which node to select .It just selects a neighboring node at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

State Space diagram for Hill Climbing

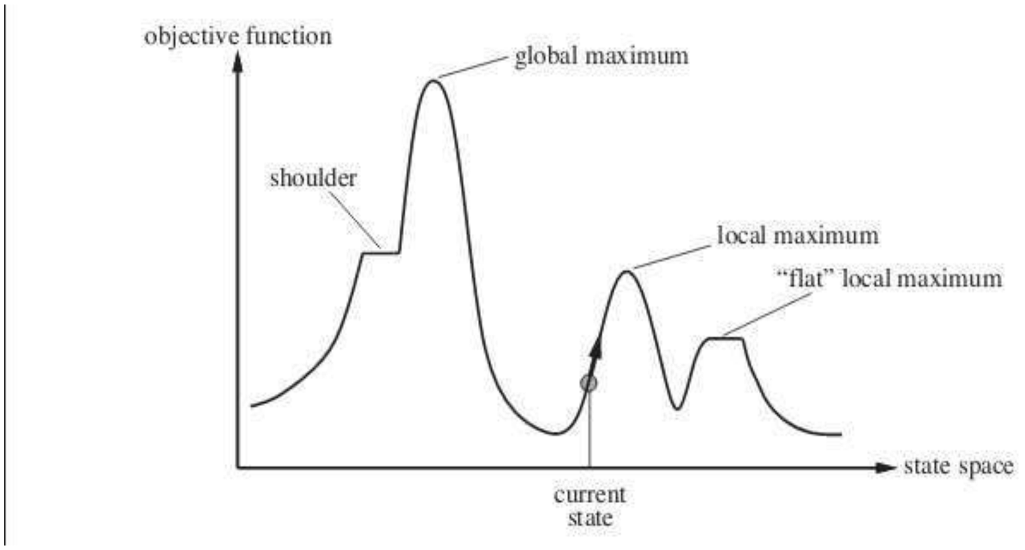
State space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).

X- axis : denotes the state space ie states or configuration our algorithm may reach.

Different regions in the State Space Diagram

1. Local maximum : It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here value of objective function is higher than its neighbors.

Y-axis : denotes the values of objective function corresponding to to a particular state.
maximu



2. Global maximum : It is the best possible state in the state space diagram. This because at this state, objective function has highest value.
3. Plateua/flat local maximum : It is a flat region of state space where neighboring states have the same value.
4. Ridge : It is region which is higher than its neighbours but itself has a slope. It is a special kind of local maximum.
5. Current state : The region of state space diagram where we are currently present during the search.
6. Shoulder : It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing
Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :

1. Local maximum : At a local maximum all neighboring states have a values which is worse than than the current state. Since hill climbing uses greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
   To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
2. Plateau : On plateau all neighbors have same value . Hence, it is not possible to select the best direction.

To overcome plateaus : Make a big jump. Randomly select a state far away from current state. Chances are that we will land at a non-plateau region

3. Ridge : Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
   To overcome Ridge : In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Best First Search (Informed Search)

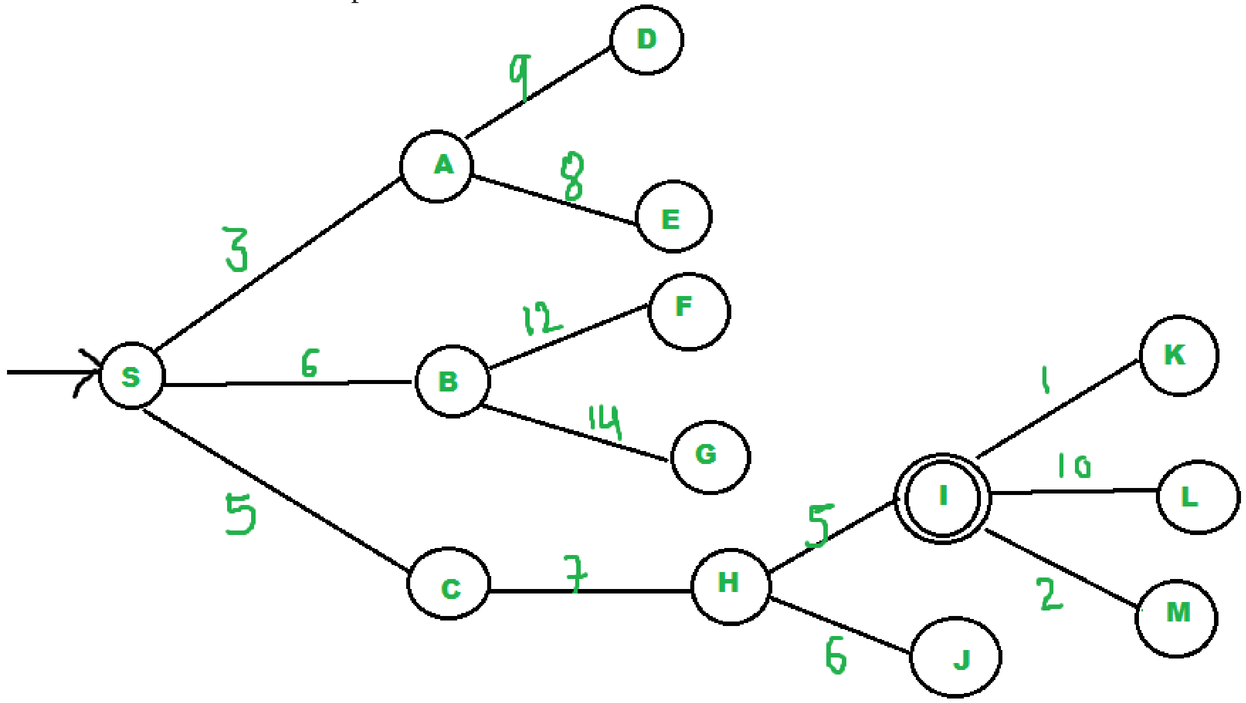
In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.
We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

Algorithm:

```
Best-First-Search(Grah g, Node start)
  1) Create an empty PriorityQueue
     PriorityQueue pq;
  2) Insert "start" in pq.
     pq.insert(start)
  3) Until PriorityQueue is empty
       u = PriorityQueue.DeleteMin
```

```
        If u is the goal
          Exit
        Else
          Foreach neighbor v of u
            If v "Unvisited"
                Mark v "Visited"
                pq.insert(v)
          Mark v "Examined"
End procedure
```

Let us consider below example.



We start from source "S" and search for goal "I" using given costs and Best First search.

pq initially contains S
We remove s from and process unvisited neighbors of S to pq.
pq now contains {A, C, B} (C is put before B because C has lesser cost)

We remove A from pq and process unvisited neighbors of A to pq.
pq now contains {C, B, E, D}

We remove C from pq and process unvisited
neighbors of C to pq.
pq now contains {B, H, E, D}

We remove B from pq and process unvisited
neighbors of B to pq.
pq now contains {H, E, D, F, G}

We remove H from pq. Since our goal
"I" is a neighbor of H, we return.

Analysis :

- The worst case time complexity for Best First Search is O(n * Log n) where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take O(log n) time.
- Performance of the algorithm depends on how well the cost or evaluation function is designed.

A* Search Algorithm

A* is a type of search algorithm. Some problems can be solved by representing the world in the initial state, and then for each action we can perform on the world we generate states for what the world would be like if we did so. If you do this until the world is in the state that we specified as a solution, then the route from the start to this goal state is the solution to your problem.

In this tutorial I will look at the use of state space search to find the shortest path between two points (pathfinding), and also to solve a simple sliding tile puzzle (the 8-puzzle). Let's look at some of the terms used in Artificial Intelligence when describing this state space search.

Some terminology

A *node* is a state that the problem's world can be in. In pathfinding a node would be just a 2d coordinate of where we are at the present time. In the 8-puzzle it is the positions of all the tiles. Next all the nodes are arranged in a *graph* where links between nodes represent valid steps in solving the problem. These links are known as *edges*. In the 8-puzzle diagram the edges are shown as blue lines. See figure 1 below.
State space search, then, is solving a problem by beginning with the start state, and then for each node we expand all the nodes beneath it in the graph by applying all the possible moves that can be made at each point.

Heuristics and Algorithms

At this point we introduce an important concept, the *heuristic*. This is like an algorithm, but with a key difference. An algorithm is a set of steps which you can follow to solve a problem, which always works for valid input. For example you could probably write an algorithm yourself for

multiplying two numbers together on paper. A heuristic is not guaranteed to work but is useful in that it may solve a problem for which there is no algorithm.
We need a heuristic to help us cut down on this huge search problem. What we need is to use our heuristic at each node to make an estimate of how far we are from the goal. In pathfinding we know exactly how far we are, because we know how far we can move each step, and we can calculate the exact distance to the goal.
But the 8-puzzle is more difficult. There is no known algorithm for calculating from a given position how many moves it will take to get to the goal state. So various heuristics have been devised. The best one that I know of is known as the Nilsson score which leads fairly directly to the goal most of the time, as we shall see.

Cost

When looking at each node in the graph, we now have an idea of a heuristic, which can estimate how close the state is to the goal. Another important consideration is the cost of getting to where we are. In the case of pathfinding we often assign a movement cost to each square. The cost is the same then the cost of each square is one. If we wanted to differentiate between terrain types we may give higher costs to grass and mud than to newly made road. When looking at a node we want to add up the cost of what it took to get here, and this is simply the sum of the cost of this node and all those that are above it in the graph.

8 Puzzle
Let's look at the 8 puzzle in more detail. This is a simple sliding tile puzzle on a 3*3 grid where one tile is missing and you can move the other tiles into the gap until you get the puzzle into the goal position. See figure 1.

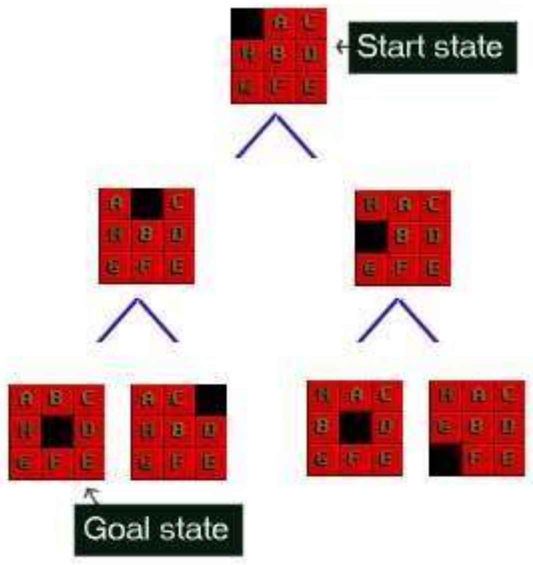


Figure 1 : The 8-Puzzle state space for a very simple example

There are 362,880 different states that the puzzle can be in, and to find a solution the search has to find a route through them. From most positions of the search the number of edges (that's the blue lines) is two. That means that the number of nodes you have in each level of the search is 2^d where d is the depth. If the number of steps to solve a particular state is 18, then that�s 262,144 nodes just at that level.

The 8 puzzle game state is as simple as representing a list of the 9 squares and what's in them. Here are two states for example; the last one is the GOAL state, at which point we've found the solution. The first is a jumbled up example that you may start from.

Start state SPACE, A, C, H, B, D, G, F, E
Goal state A, B, C, H, SPACE, D, G, F, E
The rules that you can apply to the puzzle are also simple. If there is a blank tile above, below, to the left or to the right of a given tile, then you can move that tile into the space. To solve the puzzle you need to find the path from the start state, through the graph down to the goal state.

There is example code to to solve the 8-puzzle on the github site.

Pathfinding
In a video game, or some other pathfinding scenario, you want to search a state space and find out how to get from somewhere you are to somewhere you want to be, without bumping into walls or going too far. For reasons we will see later, the A* algorithm will not only find a path, if there is one, but it will find the shortest path. A state in pathfinding is simply a position in the world. In the example of a maze game like Pacman you can represent where everything is using a simple 2d grid. The start state for a ghost say, would be the 2d coordinate of where the ghost is at the start of the search. The goal state would be where pacman is so we can go and eat him. There is also example code to do pathfinding on the github site.

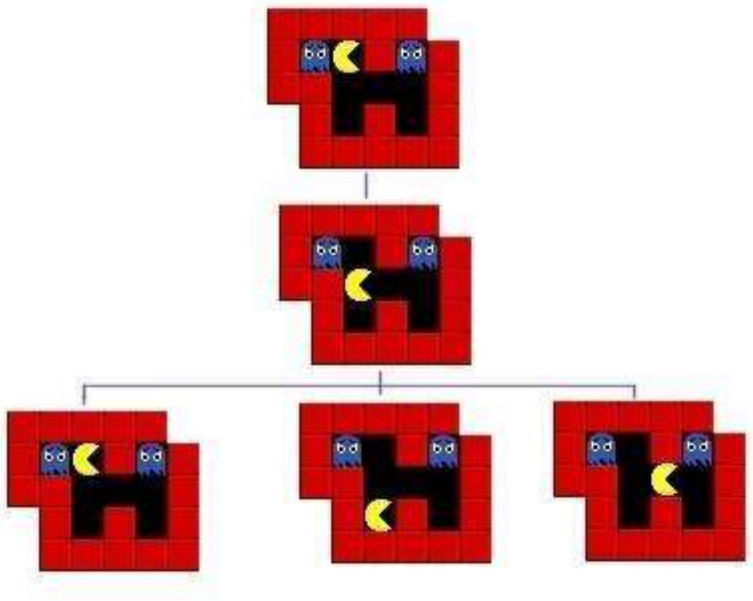


Figure 2 : The first three steps of a pathfinding state space

Implementing A*

We are now ready to look at the operation of the A* algorithm. What we need to do is start with the goal state and then generate the graph downwards from there. Let's take the 8-puzzle in figure 1. We ask how many moves can we make from the start state? The answer is 2, there are two directions we can move the blank tile, and so our graph expands.

If we were just to continue blindly generating successors to each node, we could potentially fill the computer's memory before we found the goal node. Obviously we need to remember the best nodes and search those first. We also need to remember the nodes that we have expanded already, so that we don't expand the same state repeatedly.

Let's start with the OPEN list. This is where we will remember which nodes we haven't yet expanded. When the algorithm begins the start state is placed on the open list, it is the only state we know about and we have not expanded it. So we will expand the nodes from the start and put those on the OPEN list too. Now we are done with the start node and we will put that on the CLOSED list. The CLOSED list is a list of nodes that we have expanded.

$f = g + h$

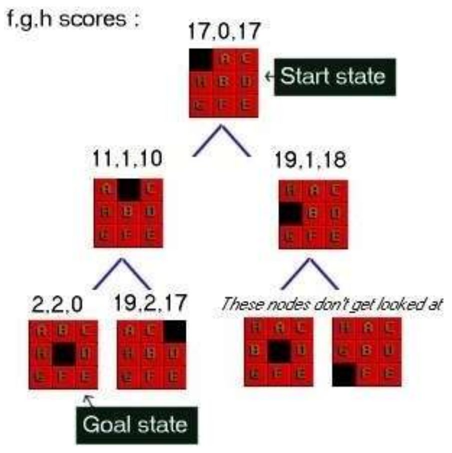
Using the OPEN and CLOSED list lets us be more selective about what we look at next in the search. We want to look at the best nodes first. We will give each node a score on how good we think it is. This score should be thought of as the cost of getting from the node to the goal plus the cost of getting to where we are. Traditionally this has been represented by the letters f, g and h. 'g' is the sum of all the costs it took to get here, 'h' is our heuristic function, the estimate of what it will take to get to the goal. 'f' is the sum of these two. We will store each of these in our nodes.

Using the f, g and h values the A* algorithm will be directed, subject to conditions we will look at further on, towards the goal and will find it in the shortest route possible.

So far we have looked at the components of the A*, let's see how they all fit together to make the algorithm :

Pseudocode

Hopefully the ideas we looked at in the preceding paragraphs will now click into place as we look at the A* algorithm pseudocode. You may find it helpful to print this out or leave the window open while we discuss it.



To help make the operation of the algorithm clear we will look again at the 8-puzzle problem in figure 1 above. Figure 3 below shows the f, g and h scores for each of the tiles.
Figure 3 : 8-Puzzle state space showing f,g,h scores

First of all look at the g score for each node. This is the cost of what it took to get from the start to that node. So in the picture the center number is g. As you can see it increases by one at each level. In some problems the cost may vary for different state changes. For example in pathfinding there is sometimes a type of terrain that costs more than other types.
Next look at the last number in each triple. This is h, the heuristic score. As I mentioned above I am using a heuristic known as Nilsson's Sequence, which converges quickly to a correct solution in many cases. Here is how you calculate this score for a given 8-puzzle state :

Advantages:

It is complete and optimal.

It is the best one from other techniques. It is used to solve very complex problems.

It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

Disadvantages:

This algorithm is complete if the branching factor is finite and every action has fixed cost.

The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n).

AO* Search: (And-Or) Graph

The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.

OPEN:

It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.

CLOSE:

It contains the nodes that have already been processed.

6 7: The distance from current node to goal node.

Algorithm

Step 1: Place the starting node into OPEN.

Step 2: Compute the most promising solution tree say T0.

Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in

CLOSE

Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.

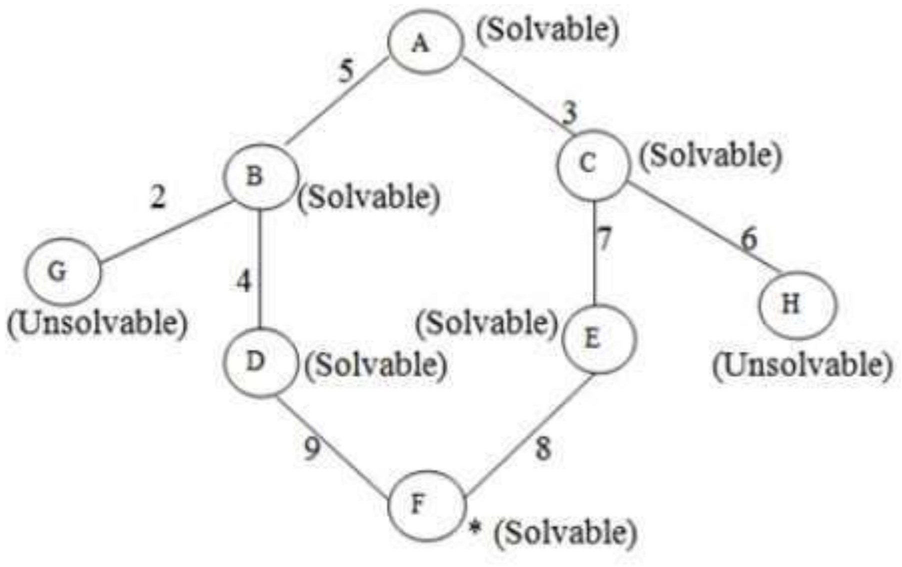
Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.

Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.

Step 7: Return to Step 2.

Step 8: Exit.

Implementation:



Figure

Let us take the following example to implement the AO* algorithm

Step 1:

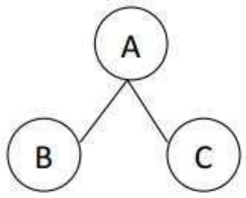
In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H. Take A as the starting node. So place A into OPEN.

i.e. OPEN $\boxed{A}$ $\boxed{\phi}$

Step 2:

The children of A are B and C which are solvable. So place them into OPEN and place A into the CLOSE.

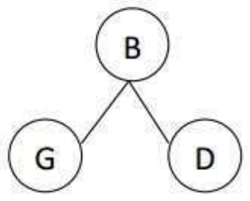
i.e. OPEN = $\boxed{B \mid C}$ CLOSE = $\boxed{A}$

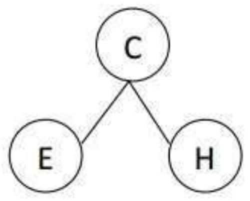


Step 3:

Now process the nodes B and C. The children of B and C are to be placed into OPEN. Also remove B and C from OPEN and place them into CLOSE.

So OPEN = $\boxed{G \mid D \mid E \mid \;}$ C $\boxed{A \mid B \mid C}$



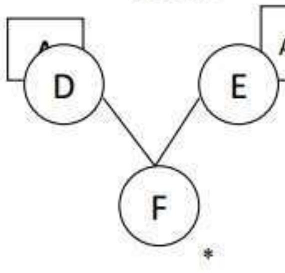


'O' indicated that the nodes G and H are unsolvable.

Step 4:

As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e. OPEN = CLOSE =



A	B	C		$G^{(O)}$	D E		$H^{(O)}$

Step 5:

Now we have been reached at our goal state. So place F into CLOSE.

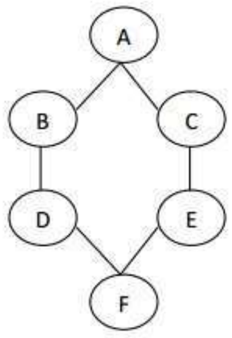
A	B	C		$G^{(O)}$	D E		$H^{(O)}$	F

i.e. CLOSE =

Step 6:

Success and Exit

AO* Graph:



Figure

Advantages:

It is an optimal algorithm.

If traverse according to the ordering of nodes. It can be used for both OR and AND graph.

Disadvantages:

Sometimes for unsolvable nodes, it can't find the optimal path. Its complexity is than other algorithms.

PROBLEM REDUCTION

Problem Reduction with AO* Algorithm.

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

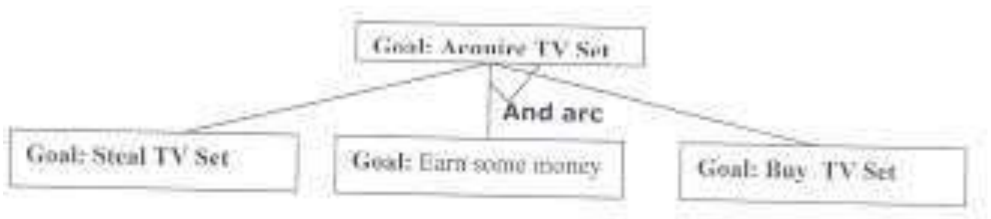


Figure shows AND - Or graph - an example.

An algorithm to find a solution in an AND - OR graph must handle AND area appropriately. A* algorithm can not search AND - OR graphs efficiently. This can be understand from the give figure.

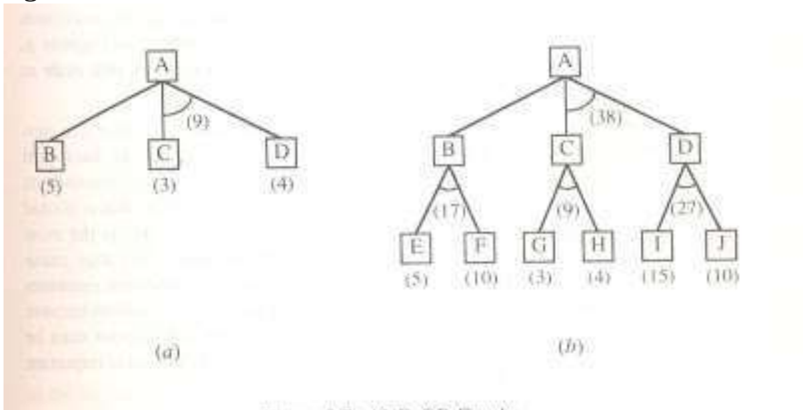


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D . the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components. With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

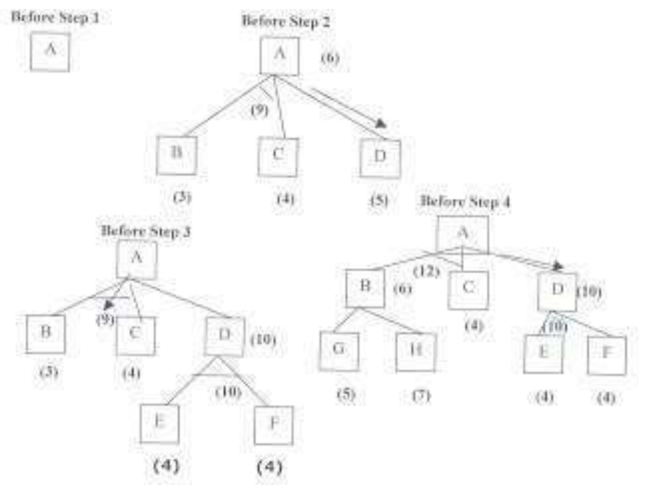
Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of (17+1=18). Thus we can see that to search an AND-OR graph, the following three things must be done.

1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.

2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.

3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

The propagation of revised cost estimation backward is in the tree is not necessary in A* algorithm. This is because in AO* algorithm expanded nodes are re-examined so that the current best path can be selected. The working of AO* algorithm is illustrated in figure as follows:



Referring the figure. The initial node is expanded and D is Marked initially as promising node. D is expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better . it is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution. An A* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes.

The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes. The cost of getting from the start nodes to the current node "g" is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO* algorithm serves as the estimate of goodness of a node. Also a there should value called FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical.
For representing above graphs AO* algorithm is as follows

AO* ALGORITHM:
1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT).

2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(I) Trace the marked arcs from INIT and select an unbounded node NODE.

(II) Generate the successors of NODE . if there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not also an ancester of NODE do the following

(a) add SUCCESSOR to graph G

(b) if successor is not a terminal node, mark it solved and assign zero to its h ' value.

(c) If successor is not a terminal node, compute it h' value.

(III) propagate the newly discovered information up the graph by doing the following . let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;

(a) select a node from S call if CURRENT and remove it from S.

(b) compute h' of each of the arcs emerging from CURRENT , Assign minimum h' to CURRENT.

(c) Mark the minimum cost path a s the best out of CURRENT.

(d) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labeled SOLVED.

(e) If CURRENT has been marked SOLVED or its h ' has just changed, its new status must be propagate backwards up the graph . hence all the ancestors of CURRENT are added to S.

(Refered From Artificial Intelligence TMH)

AO* Search Procedure.

1. Place the start node on open.

2. Using the search tree, compute the most promising solution tree TP .

3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.

4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.

5. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. Remove all nodes from open ,with unsolvable ancestors.

6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.

7. Go back to step(2)

Note: AO* will always find minimum cost solution.

CONSTRAINT SATISFACTION:-

Many problems in AI can be considered as problems of constraint satisfaction, in which the goal state satisfies a given set of constraint. constraint satisfaction problems can be solved by using any of the search strategies. The general form of the constraint satisfaction procedure is as follows:

Until a complete solution is found or until all paths have led to lead ends, do

1. select an unexpanded node of the search graph.

2. Apply the constraint inference rules to the selected node to generate all possible new constraints.

3. If the set of constraints contains a contradiction, then report that this path is a dead end.

4. If the set of constraints describes a complete solution then report success.

5. If neither a constraint nor a complete solution has been found then apply the rules to generate new partial solutions. Insert these partial solutions into the search graph.

Example: consider the crypt arithmetic problems.

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

Assign decimal digit to each of the letters in such a way that the answer to the problem is correct to the same letter occurs more than once , it must be assign the same digit each time . no two different letters may be assigned the same digit. Consider the crypt arithmetic problem.

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

CONSTRAINTS:-

1. no two digit can be assigned to same letter.

2. only single digit number can be assign to a letter.

1. no two letters can be assigned same digit.

2. Assumption can be made at various levels such that they do not contradict each other.

3. The problem can be decomposed into secured constraints. A constraint satisfaction approach may be used.

4. Any of search techniques may be used.

5. Backtracking may be performed as applicable us applied search techniques.

6. Rule of arithmetic may be followed.

Initial state of problem.
D=?
E=?
Y=?
N=?
R=?
O=?
S=?
M=?
C1=?
C2=?
C1 ,C 2, C3 stands for the carry variables respectively.

Goal State: the digits to the letters must be assigned in such a manner so that the sum is satisfied.

Solution Process:

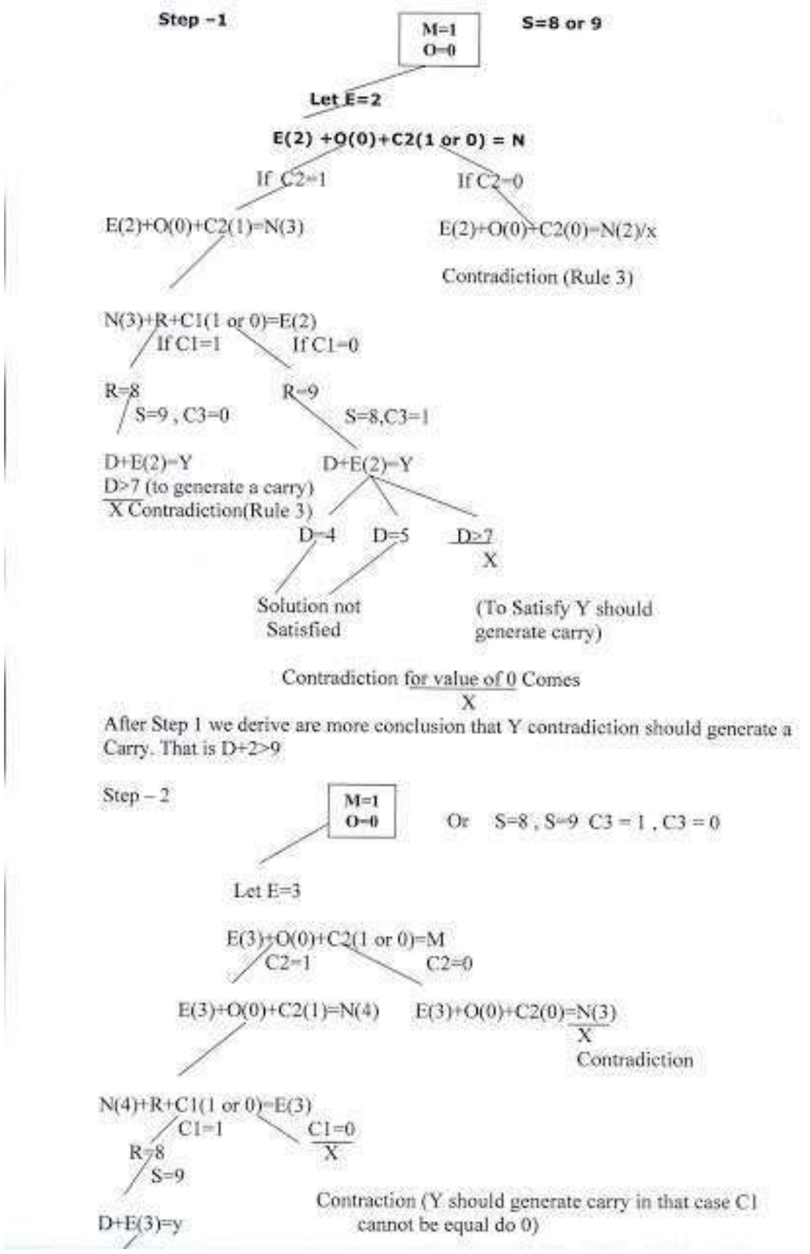
We are following the depth-first method to solve the problem.

1. initial guess m=1 because the sum of two single digits can generate at most a carry '1'.

2. When n=1 o=0 or 1 because the largest single digit number added to m=1 can generate the sum of either 0 or 1 depend on the carry received from the carry sum. By this we conclude that o=0 because m is already 1 hence we cannot assign same digit another letter(rule no.)

3. We have m=1 and o=0 to get o=0 we have s=8 or 9, again depending on the carry received from the earlier sum.

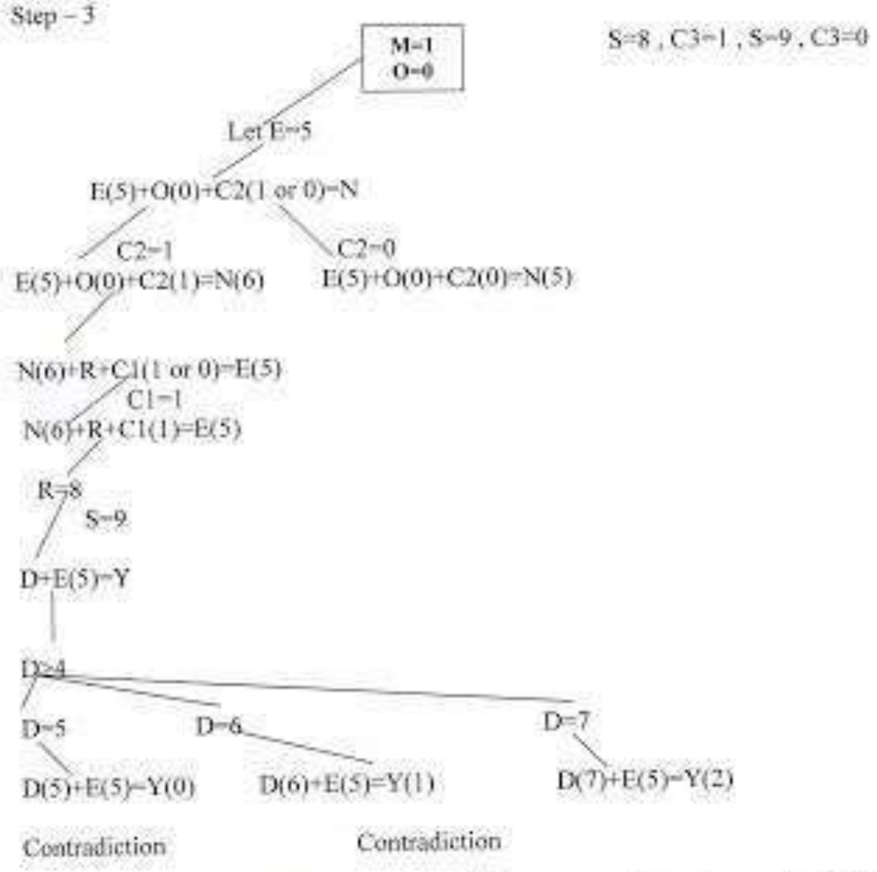
The same process can be repeated further. The problem has to be composed into various constraints. And each constraints is to be satisfied by guessing the possible digits that the letters can be assumed that the initial guess has been already made . rest of the process is being shown in the form of a tree, using depth-first search for the clear understandability of the solution process.



D>6 (Controduction)

After Step 2 , we found that C1 cannot be Zero, Since Y has to generate a carry to satisfy goal state. From this step onwards, no need to branch for C1=0.

Step – 3



At Step (4) we have assigned a single digit to every letter in accordance with the constraints & production rules.
Now by backtracking , we find the different digits assigned to different letters and hence reach the solution state.

Solution State:-

Y = 2
D = 7
S = 9
R = 8
N = 6
E = 5
O = 0
M = 1
C1 = 1
C2 = 0
C3 = 0

$$\begin{array}{cccccc} & & C3(0) & C2(1) & C1(1) & \\ & & S(9) & E(5) & N(6) & D(7) \\ + & & M(1) & O(0) & R(8) & E(5) \\ \hline & M(1) & O(0) & N(6) & E(5) & Y(2) \\ \hline \end{array}$$

MEANS - ENDS ANALYSIS:-

Most of the search strategies either reason forward of backward however, often a mixture o the two directions is appropriate. Such mixed strategy would make it possible to solve the major parts of problem first and solve the smaller problems the arise when combining them together. Such a technique is called "Means - Ends Analysis".

The means -ends analysis process centers around finding the difference between current state and goal state. The problem space of means - ends analysis has an initial state and one or more goal state, a set of operate with a set of preconditions their application and difference functions that computes the difference between two state a(i) and s(j). A problem is solved using means - ends analysis by

1. Computing the current state s1 to a goal state s2 and computing their difference D12.

2. Satisfy the preconditions for some recommended operator op is selected, then to reduce the difference D12.

3. The operator OP is applied if possible. If not the current state is solved a goal is created and means- ends analysis is applied recursively to reduce the sub goal.

4. If the sub goal is solved state is restored and work resumed on the original problem.

(the first AI program to use means - ends analysis was the GPS General problem solver)

means- ends analysis I useful for many human planning activities. Consider the example of planing for an office worker. Suppose we have a different table of three rules:

1. If in out current state we are hungry , and in our goal state we are not hungry , then either the "visit hotel" or "visit Canteen " operator is recommended.

2. If our current state we do not have money , and if in your goal state we have money, then the "Visit our bank" operator or the "Visit secretary" operator is recommended.

3. If our current state we do not know where something is , need in our goal state we do know, then either the "visit office enquiry" , "visit secretary" or "visit co worker " operator is recommended.

KNOWLEDGE REPRESENTATION

KNOWLEDGE REPRESENTATION:-

For the purpose of solving complex problems c\encountered in AI, we need both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. In all variety of knowledge representations , we deal with two kinds of entities.

A. Facts: Truths in some relevant world. These are the things we want to represent.

B. Representations of facts in some chosen formalism . these are things we will

actually be able to manipulate.

One way to think of structuring these entities is at two levels : (a) the knowledge level, at which facts are described, and (b) the symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

The facts and representations are linked with two-way mappings. This link is called representation mappings. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations One common representation is natural language (particularly English) sentences. Regardless of the representation for facts we use in a program , we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. We need mapping functions from English sentences to the representation we actually use and from it back to sentences.

Representations and Mappings

- In order to solve complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions.
- Knowledge and Representation are two distinct entities. They play central but distinguishable roles in the intelligent system.
- Knowledge is a description of the world. It determines a system’s competence by what it knows.
- Moreover, Representation is the way knowledge is encoded. It defines a system’s performance in doing something.
- Different types of knowledge require different kinds of representation.

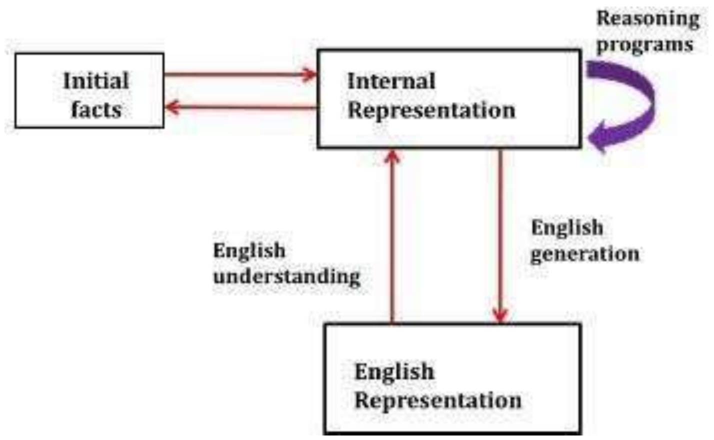


Fig: Mapping between Facts and Representations

The Knowledge Representation models/mechanisms are often based on:

- Logic
- Rules
- Frames
- Semantic Net

Knowledge is categorized into two major types:

1. Tacit corresponds to “informal” or “implicit“
    - Exists within a human being;
    - It is embodied.
    - Difficult to articulate formally.
    - Difficult to communicate or share.
    - Moreover, Hard to steal or copy.
    - Drawn from experience, action, subjective insight
2. Explicit formal type of knowledge, Explicit
    - Explicit knowledge
    - Exists outside a human being;
    - It is embedded.
    - Can be articulated formally.
    - Also, Can be shared, copied, processed and stored.
    - So, Easy to steal or copy
    - Drawn from the artifact of some type as a principle, procedure, process, concepts.

A variety of ways of representing knowledge have been exploited in AI programs.
There are two different kinds of entities, we are dealing with.

1. Facts: Truth in some relevant world. Things we want to represent.
2. Also, Representation of facts in some chosen formalism. Things we will actually be able to manipulate.

These entities structured at two levels:

1. The knowledge level, at which facts described.
2. Moreover, The symbol level, at which representation of objects defined in terms of symbols that can manipulate by programs

Framework of Knowledge Representation

- The computer requires a well-defined problem description to process and provide a well-defined acceptable solution.

- Moreover, To collect fragments of knowledge we need first to formulate a description in our spoken language and then represent it in formal language so that computer can understand.
- Also, The computer can then use an algorithm to compute an answer.

So, This process illustrated as,

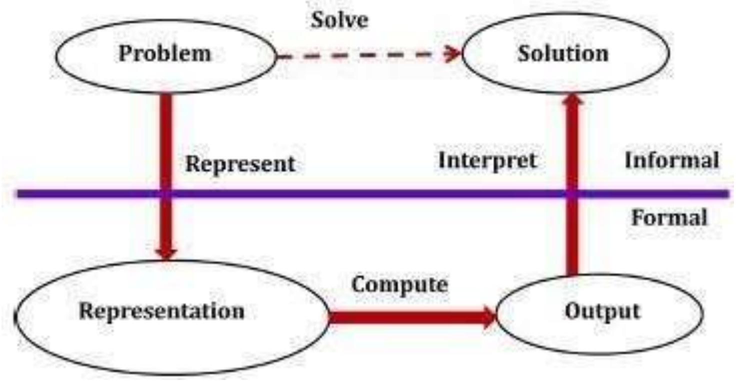


Fig: Knowledge Representation Framework

The steps are:

- The informal formalism of the problem takes place first.
- It then represented formally and the computer produces an output.
- This output can then represented in an informally described solution that user understands or checks for consistency.

The Problem solving requires,

- Formal knowledge representation, and
- Moreover, Conversion of informal knowledge to a formal knowledge that is the conversion of implicit knowledge to explicit knowledge.

Mapping between Facts and Representation

- Knowledge is a collection of facts from some domain.
- Also, We need a representation of "facts" that can manipulate by a program.
- Moreover, Normal English is insufficient, too hard currently for a computer program to draw inferences in natural languages.
- Thus some symbolic representation is necessary.

A good knowledge representation enables fast and accurate access to knowledge and understanding of the content.

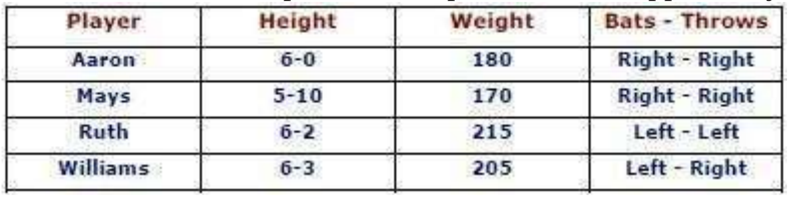
A knowledge representation system should have following properties.

1. Representational Adequacy
   - The ability to represent all kinds of knowledge that are needed in that domain.
2. Inferential Adequacy
   - Also, The ability to manipulate the representational structures to derive new structures corresponding to new knowledge inferred from old.
3. Inferential Efficiency
   - The ability to incorporate additional information into the knowledge structure that can be used to focus the attention of the inference mechanisms in the most promising direction.
4. Acquisitional Efficiency
   - Moreover, The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Knowledge Representation Schemes

Relational Knowledge

- The simplest way to represent declarative facts is a set of relations of the same sort used in the database system.
- Provides a framework to compare two objects based on equivalent attributes. o Any instance in which two different objects are compared is a relational type of knowledge.
- The table below shows a simple way to store facts.
    - Also, The facts about a set of objects are put systematically in columns.
    - This representation provides little opportunity for inference.

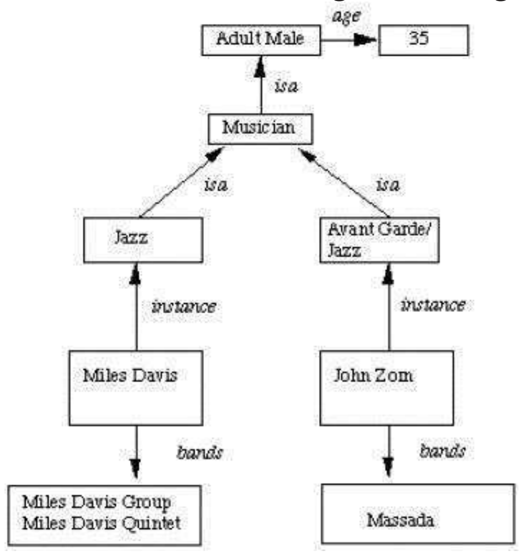


Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

- Given the facts, it is not possible to answer a simple question such as: "Who is the heaviest player?"
- Also, But if a procedure for finding the heaviest player is provided, then these facts will enable that procedure to compute an answer.
- Moreover, We can ask things like who "bats – left" and "throws – right".

Inheritable Knowledge

- Here the knowledge elements inherit attributes from their parents.
- The knowledge embodied in the design hierarchies found in the functional, physical and process domains.
- Within the hierarchy, elements inherit attributes from their parents, but in many cases, not all attributes of the parent elements prescribed to the child elements.
- Also, The inheritance is a powerful form of inference, but not adequate.
- Moreover, The basic KR (Knowledge Representation) needs to augment with inference mechanism.
- Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes.
- So, The classes organized in a generalized hierarchy.



- Boxed nodes — objects and values of attributes of objects.
- Arrows — the point from object to its value.
- This structure is known as a slot and filler structure, semantic network or a collection of frames.

The steps to retrieve a value for an attribute of an instance object:

1. Find the object in the knowledge base
2. If there is a value for the attribute report it
3. Otherwise look for a value of an instance, if none fail
4. Also, Go to that node and find a value for the attribute and then report it
5. Otherwise, search through using is until a value is found for the attribute.

Inferential Knowledge

- This knowledge generates new information from the given information.
- This new information does not require further data gathering form source but does require analysis of the given information to generate new knowledge.
- Example: given a set of relations and values, one may infer other values or relations. A predicate logic (a mathematical deduction) used to infer from a set of attributes. Moreover, Inference through predicate logic uses a set of logical operations to relate individual data.
- Represent knowledge as formal logic: All dogs have tails $\forall x: dog(x) \rightarrow hastail(x)$
- Advantages:
    - A set of strict rules.
    - Can use to derive more facts.
    - Also, Truths of new statements can be verified.
    - Guaranteed correctness.
- So, Many inference procedures available to implement standard rules of logic popular in AI systems. e.g Automated theorem proving.

Procedural Knowledge

- A representation in which the control information, to use the knowledge, embedded in the knowledge itself. For example, computer programs, directions, and recipes; these indicate specific use or implementation;
- Moreover, Knowledge encoded in some procedures, small programs that know how to do specific things, how to proceed.
- Advantages:
    - Heuristic or domain-specific knowledge can represent.
    - Moreover, Extended logical inferences, such as default reasoning facilitated.
    - Also, Side effects of actions may model. Some rules may become false in time. Keeping track of this in large systems may be tricky.
- Disadvantages:
    - Completeness — not all cases may represent.
    - Consistency — not all deductions may be correct. e.g If we know that Fred is a bird we might deduce that Fred can fly. Later we might discover that Fred is an emu.
    - Modularity sacrificed. Changes in knowledge base might have far-reaching effects.
    - Cumbersome control information.

USING PREDICATE LOGIC

Representation of Simple Facts in Logic

Propositional logic is useful because it is simple to deal with and a decision procedure for it exists.

Also, In order to draw conclusions, facts are represented in a more convenient way as,

1. Marcus is a man.
   - man(Marcus)
2. Plato is a man.
   - man(Plato)
3. All men are mortal.
   - mortal(men)

But propositional logic fails to capture the relationship between an individual being a man and that individual being a mortal.

- How can these sentences be represented so that we can infer the third sentence from the first two?
- Also, Propositional logic commits only to the existence of facts that may or may not be the case in the world being represented.
- Moreover, It has a simple syntax and simple semantics. It suffices to illustrate the process of inference.
- Propositional logic quickly becomes impractical, even for very small worlds.

Predicate logic

First-order Predicate logic (FOPL) models the world in terms of

- Objects, which are things with individual identities
- Properties of objects that distinguish them from other objects
- Relations that hold among sets of objects
- Functions, which are a subset of relations where there is only one “value” for any given “input”

First-order Predicate logic (FOPL) provides

- Constants: a, b, dog33. Name a specific object.
- Variables: X, Y. Refer to an object without naming it.
- Functions: Mapping from objects to objects.
- Terms: Refer to objects
- Atomic Sentences: in(dad-of(X), food6) Can be true or false, Correspond to propositional symbols P, Q.

A well-formed formula (*wff*) is a sentence containing no “free” variables. So, That is, all variables are “bound” by universal or existential quantifiers.

(∀x)P(x, y) has x bound as a universally quantified variable, but y is free.

Quantifiers

Universal quantification

- (∀x)P(x) means that P holds for all values of x in the domain associated with that variable
- E.g., (∀x) dolphin(x) → mammal(x)

Existential quantification

- (∃ x)P(x) means that P holds for some value of x in the domain associated with that variable
- E.g., (∃ x) mammal(x) Λ lays-eggs(x)

Also, Consider the following example that shows the use of predicate logic as a way of representing knowledge.

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

The facts described by these sentences can be represented as a set of well-formed formulas (*wffs*) as follows:

1. Marcus was a man.
    - man(Marcus)
2. Marcus was a Pompeian.
    - Pompeian(Marcus)
3. All Pompeians were Romans.
    - ∀x: Pompeian(x) → Roman(x)
4. Caesar was a ruler.
    - ruler(Caesar)
5. All Pompeians were either loyal to Caesar or hated him.
    - inclusive-or
    - ∀x: Roman(x) → loyalto(x, Caesar) ∨ hate(x, Caesar)
    - exclusive-or
    - ∀x: Roman(x) → (loyalto(x, Caesar) ∧¬ hate(x, Caesar)) ∨
    - (¬loyalto(x, Caesar) ∧ hate(x, Caesar))
6. Everyone is loyal to someone.
    - ∀x: ∃y: loyalto(x, y)

7. People only try to assassinate rulers they are not loyal to.
    - ∀x: ∀y: person(x) ∧ ruler(y) ∧ tryassassinate(x, y)
    - →¬loyalto(x, y)
8. Marcus tried to assassinate Caesar.
    - tryassassinate(Marcus, Caesar)

Now suppose if we want to use these statements to answer the question: ***Was Marcus loyal to Caesar?***

Also, Now let’s try to produce a formal proof, reasoning backward from the desired goal: ¬ Ioyalto(Marcus, Caesar)

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can, in turn, transformed, and so on, until there are no unsatisfied goals remaining.

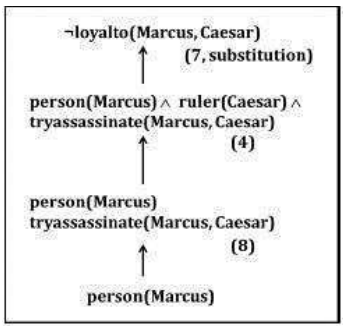
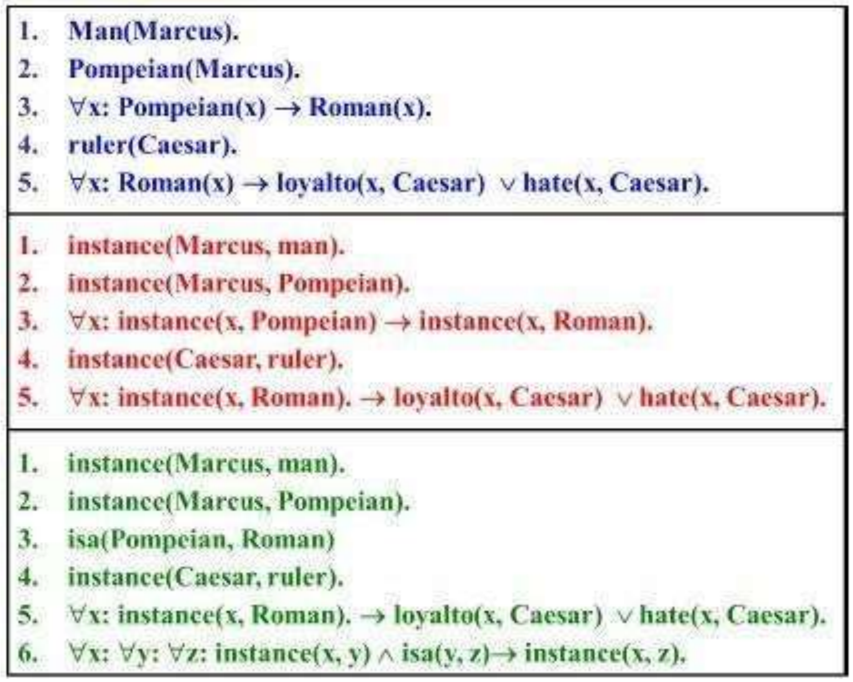

¬loyalto(Marcus, Caesar)
(7, substitution)
person(Marcus) ∧ ruler(Caesar) ∧
tryassassinate(Marcus, Caesar)
(4)
person(Marcus)
tryassassinate(Marcus, Caesar)
(8)
person(Marcus)

Figure: An attempt to prove ¬loyalto(Marcus, Caesar).

- The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. Also, We need to add the representation of another fact to our system, namely***:*** ∀ ***man(x) → person(x)***
- Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.
- Moreover, From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:
    1. Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
    2. Also, There is often a choice of how to represent the knowledge. Simple representations are desirable, but they may exclude certain kinds of reasoning.
    3. Similalry, Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively. Moreover, It is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention.

Representing Instance and ISA Relationships

- Specific attributes **instance** and **isa** play an important role particularly in a useful form of reasoning called property inheritance.
- The predicates instance and isa explicitly captured the relationships they used to express, namely class membership and class inclusion.
- 4.2 shows the first five sentences of the last section represented in logic in three different ways.
- The first part of the figure contains the representations we have already discussed. In these representations, class membership represented with unary predicates (such as Roman), each of which corresponds to a class.
- Asserting that P(x) is true is equivalent to asserting that x is an instance (or element) of P.
- The second part of the figure contains representations that use the ***instance*** predicate explicitly.



1. Man(Marcus).
2. Pompeian(Marcus).
3. $\forall x$: Pompeian(x) $\rightarrow$ Roman(x).
4. ruler(Caesar).
5. $\forall x$: Roman(x) $\rightarrow$ loyalto(x, Caesar) $\vee$ hate(x, Caesar).

1. instance(Marcus, man).
2. instance(Marcus, Pompeian).
3. $\forall x$: instance(x, Pompeian) $\rightarrow$ instance(x, Roman).
4. instance(Caesar, ruler).
5. $\forall x$: instance(x, Roman). $\rightarrow$ loyalto(x, Caesar) $\vee$ hate(x, Caesar).

1. instance(Marcus, man).
2. instance(Marcus, Pompeian).
3. isa(Pompeian, Roman)
4. instance(Caesar, ruler).
5. $\forall x$: instance(x, Roman). $\rightarrow$ loyalto(x, Caesar) $\vee$ hate(x, Caesar).
6. $\forall x$: $\forall y$: $\forall z$: instance(x, y) $\wedge$ isa(y, z)$\rightarrow$ instance(x, z).

Figure: Three ways of representing class membership: ISA Relationships

- The predicate ***instance*** is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs.
- But these representations do not use an explicit ***isa*** predicate.
- Instead, subclass relationships, such as that between Pompeians and Romans, described as shown in sentence 3.
- The implication rule states that if an object is an instance of the subclass Pompeian then it is an instance of the superclass Roman.
- Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship.
- The third part contains representations that use both the ***instance*** and ***isa*** predicates explicitly.
- The use of the ***isa*** predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

Computable Functions and Predicates

- To express simple facts, such as the following greater-than and less-than relationships: gt(1,O) It(0,1) gt(2,1) It(1,2) gt(3,2) It(2,3)
- It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of gt(2 + 3,1)
- To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to gt.

Consider the following set of facts, again involving Marcus:

1) Marcus was a man.
   man(Marcus)
2) Marcus was a Pompeian.
   Pompeian(Marcus)
3) Marcus was born in 40 A.D.
   born(Marcus, 40)
4) All men are mortal.
   x: man(x) → mortal(x)
5) All Pompeians died when the volcano erupted in 79 A.D.
   erupted(volcano, 79) ∧ ∀ x : [Pompeian(x) → died(x, 79)]
6) No mortal lives longer than 150 years.
   x: t1: At2: *mortal(x) born(x, t1) gt(t2 – t1,150) → died(x, t2)*
7) It is now 1991.
   now = 1991

So, Above example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

- So, Now suppose we want to answer the question "Is Marcus alive?"
- The statements suggested here, there may be two ways of deducing an answer.
- Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible.
- Also, As soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking.

So we add the following facts:

8) Alive means not dead.
   x: t: [alive(x, t) → ¬ dead(x, t)] [¬ dead(x, t) → alive(x, t)]
9) If someone dies, then he is dead at all later times.
   x: t1: At2*: died(x, t1) gt(t2, t1) → dead(x, t2)*

So, Now let's attempt to answer the question "Is Marcus alive?" by proving: **¬ *alive(Marcus, now)***

Resolution

Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
    1. Select two clauses. Call these the parent clauses.
    2. Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
    3. If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of classes available to the procedure.

The Unification Algorithm

- In propositional logic, it is easy to determine that two literals cannot both be true at the same time.
- Simply look for L and ¬L in predicate logic, this matching process is more complicated since the arguments of the predicates must be considered.
- For example, man(John) and ¬man(John) is a contradiction, while the man(John) and ¬man(Spot) is not.
- Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical.
- There is a straightforward recursive procedure, called the unification algorithm, that does it.

Algorithm: Unify(L1, L2)

```
1. If L1 or L2 are both variables or constants, then:
    1. If L1 and L2 are identical, then return NIL.
    2. Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return
       (L2/L1).
    3. Also, Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL}, else
       return (L1/L2). d. Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.
3. If LI and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the
   substitutions used to unify L1 and L2.)
5. For I ← 1 to the number of arguments in L1 :
    1. Call Unify with the i^th argument of L1 and the i^th argument of L2, putting the
       result in S.
    2. If S contains FAIL then return {FAIL}.
    3. If S is not equal to NIL then:
        2. Apply S to the remainder of both L1 and L2.
        3. SUBST: = APPEND(S, SUBST).
6. Return SUBST.
```

Resolution in Predicate Logic

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until a contradiction found, no progress can make, or a predetermined amount of effort has expanded.
    1. Select two clauses. Call these the parent clauses.
    2. Resolve them together. The resolvent will the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals T1 and ¬T2 such that one of the parent clauses contains T2 and the other contains T1 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 Complementary literals. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should omit from the resolvent.
    3. If the resolvent is an empty clause, then a contradiction has found. Moreover, If it is not, then add it to the set of classes available to the procedure.

Resolution Procedure

- Resolution is a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form.
- Resolution produces proofs by refutation.
- In other words, ***to prove a statement (i.e., to show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable).***
- The resolution procedure is a simple iterative process: at each step, two clauses, called the parent clauses, are compared (resolved), resulting in a new clause that has inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

winter* V *summer

¬ *winter* V *cold*

- Now we observe that precisely one of winter and ¬ winter will be true at any point.
- If winter is true, then cold must be true to guarantee the truth of the second clause. If ¬ winter is true, then summer must be true to guarantee the truth of the first clause.
- Thus we see that from these two clauses we can deduce ***summer V cold***
- This is the deduction that the resolution procedure will make.
- Resolution operates by taking two clauses that each contains the same literal, in this example, ***winter***.
- Moreover, The literal must occur in the positive form in one clause and in negative form in the other. The resolvent obtained by combining all of the literals of the two parent clauses except the ones that cancel.
- If the clause that produced is the empty clause, then a contradiction has found.

For example, the two clauses

winter

¬ winter

will produce the empty clause.

Natural Deduction Using Rules

Testing whether a proposition is a tautology by testing every possible truth assignment is expensive—there are exponentially many. We need a **deductive system**, which will allow us to construct proofs of tautologies in a step-by-step fashion.

The system we will use is known as **natural deduction**. The system consists of a set of **rules of inference** for deriving consequences from premises. One builds a proof tree whose root is the proposition to be proved and whose leaves are the initial assumptions or axioms (for proof trees, we usually draw the root at the bottom and the leaves at the top).

For example, one rule of our system is known as **modus ponens**. Intuitively, this says that if we know P is true, and we know that P implies Q, then we can conclude Q.

$$\frac{P \qquad P \Rightarrow Q}{Q}\ \text{(modus ponens)}$$

The propositions above the line are called **premises**; the proposition below the line is the **conclusion**. Both the premises and the conclusion may contain metavariables (in this case, P and Q) representing arbitrary propositions. When an inference rule is used as part of a proof, the metavariables are replaced in a consistent way with the appropriate kind of object (in this case, propositions).

Most rules come in one of two flavors: **introduction** or **elimination** rules. Introduction rules introduce the use of a logical operator, and elimination rules eliminate it. Modus ponens is an elimination rule for ⇒. On the right-hand side of a rule, we often write the name of the rule. This is helpful when reading proofs. In this case, we have written (modus ponens). We could also have written (⇒-elim) to indicate that this is the elimination rule for ⇒.

Rules for Conjunction

Conjunction (∧) has an introduction rule and two elimination rules:

$$\frac{P \qquad Q}{P \land Q}\ (\land\text{-intro}) \qquad \frac{P \land Q}{P}\ (\land\text{-elim-left}) \qquad \frac{P \land Q}{Q}\ (\land\text{-elim-right})$$

Rule for T

The simplest introduction rule is the one for T. It is called "unit". Because it has no premises, this rule is an **axiom**: something that can start a proof.

$$\frac{}{T}\ \text{(unit)}$$

Rules for Implication

In natural deduction, to prove an implication of the form P ⇒ Q, we assume P, then reason under that assumption to try to derive Q. If we are successful, then we can conclude that P ⇒ Q.

In a proof, we are always allowed to introduce a new assumption P, then reason under that assumption. We must give the assumption a name; we have used the name x in the example below. Each distinct assumption must have a different name.

$$\frac{}{[x : P]}\ \text{(assum)}$$

Because it has no premises, this rule can also start a proof. It can be used as if the proposition P were proved. The name of the assumption is also indicated here.
However, you do not get to make assumptions for free! To get a complete proof, all assumptions must be eventually *discharged*. This is done in the implication introduction rule. This rule introduces an implication P ⇒ Q by discharging a prior assumption [x : P]. Intuitively, if Q can be proved under the assumption P, then the implication P ⇒ Q holds without any assumptions. We write x in the rule name to show which assumption is discharged. This rule and modus ponens are the introduction and elimination rules for implications.

$$\frac{\begin{array}{c}[x : P]\\ \vdots\\ Q\end{array}}{P \Rightarrow Q}\ (\Rightarrow\text{-intro}/x) \qquad \frac{P \quad P \Rightarrow Q}{Q}\ (\Rightarrow\text{-elim, modus ponens})$$

A proof is valid only if every assumption is eventually discharged. This must happen in the proof tree below the assumption. The same assumption can be used more than once.

Rules for Disjunction

$$\frac{P}{P \vee Q}\ (\vee\text{-intro-left}) \qquad \frac{Q}{P \vee Q}\ (\vee\text{-intro-right}) \qquad \frac{P \vee Q \quad P \Rightarrow R \quad Q \Rightarrow R}{R}\ (\vee\text{-elim})$$

Rules for Negation

A negation ¬P can be considered an abbreviation for P ⇒ ⊥:

$$\frac{P \Rightarrow \bot}{\neg P}\ (\neg\text{-intro}) \qquad \frac{\neg P}{P \Rightarrow \bot}\ (\neg\text{-elim})$$

Rules for Falsity

$$\frac{\begin{array}{c}[x : \neg P]\\ \vdots\\ \bot\end{array}}{P}\ (\text{reductio ad absurdum, RAA}/x) \qquad \frac{\bot}{P}\ (\text{ex falso quodlibet, EFQ})$$

Reductio ad absurdum (RAA) is an interesting rule. It embodies proofs by contradiction. It says that if by assuming that P is false we can derive a contradiction, then P must be true. The assumption x is discharged in the application of this rule. This rule is present in classical logic but not in **intuitionistic** (constructive) logic. In intuitionistic logic, a proposition is not considered true simply because its negation is false.

Excluded Middle

Another classical tautology that is not intuitionistically valid is the **the law of the excluded middle**, P ∨ ¬P. We will take it as an axiom in our system. The Latin name for this rule is *tertium non datur*, but we will call it *magic*.

$$\frac{}{P \vee \neg P}\ (\text{magic})$$

Proofs

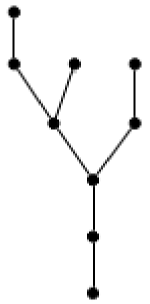
A proof of proposition P in natural deduction starts from axioms and assumptions and derives P with all assumptions discharged. Every step in the proof is an instance of an inference rule with metavariables substituted consistently with expressions of the appropriate syntactic class.

Example

For example, here is a proof of the proposition (A ⇒ B ⇒ C) ⇒ (A ∧ B ⇒ C).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[y : A \wedge B]}\,(A)}{A}\,(\wedge E) \quad \dfrac{}{[x : A \Rightarrow B \Rightarrow C]}\,(A)}{B \Rightarrow C}\,(\Rightarrow E) \quad \dfrac{\dfrac{}{[y : A \wedge B]}\,(A)}{B}\,(\wedge E)}{C}\,(\Rightarrow E)}{A \wedge B \Rightarrow C}\,(\Rightarrow I, y)}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)}\,(\Rightarrow I, x)$$

The final step in the proof is to derive (A ⇒ B ⇒ C) ⇒ (A ∧ B ⇒ C) from (A ∧ B ⇒ C), which is done using the rule (⇒-intro), discharging the assumption [x : A ⇒ B ⇒ C]. To see how this rule generates the proof step, substitute for the metavariables P, Q, x in the rule as follows: P = (A ⇒ B ⇒ C), Q = (A ∧ B ⇒ C), and x = x. The immediately previous step uses the same rule, but with a different substitution: P = A ∧ B, Q = C, x = y.
The proof tree for this example has the following form, with the proved proposition at the root and axioms and assumptions at the leaves.



A proposition that has a complete proof in a deductive system is called a **theorem** of that system.
Soundness and Completeness
A measure of a deductive system's power is whether it is powerful enough to prove all true statements. A deductive system is said to be **complete** if all true statements are theorems (have proofs in the system). For propositional logic and natural deduction, this means that all tautologies must have natural deduction proofs. Conversely, a deductive system is called **sound** if all theorems are true. The proof rules we have given above are in fact sound and complete for propositional logic: every theorem is a tautology, and every tautology is a theorem.
Finding a proof for a given tautology can be difficult. But once the proof is found, checking that it is indeed a proof is completely mechanical, requiring no intelligence or insight whatsoever. It is therefore a very strong argument that the thing proved is in fact true.
We can also make writing proofs less tedious by adding more rules that provide reasoning shortcuts. These rules are sound if there is a way to convert a proof using them into a proof using the original rules. Such added rules are called **admissible**.

Procedural versus Declarative Knowledge

We have discussed various search techniques in previous units. Now we would consider a set of rules that represent,

1. Knowledge about relationships in the world and
2. Knowledge about how to solve the problem using the content of the rules.

Procedural vs Declarative Knowledge
Procedural Knowledge

- A representation in which the control information that is necessary to use the knowledge is embedded in the knowledge itself for e.g. computer programs, directions, and recipes; these indicate specific use or implementation;
- The real difference between declarative and procedural views of knowledge lies in where control information reside.

For example, consider the following
Man (Marcus)
Man (Caesar)
Person (Cleopatra)
$\forall x$: *Man(x)* $\rightarrow$ *Person(x)*
Now, try to answer the question. ?Person(y)
The knowledge base justifies any of the following answers.
Y=Marcus
Y=Caesar
Y=Cleopatra

- We get more than one value that satisfies the predicate.
- If only one value needed, then the answer to the question will depend on the order in which the assertions examined during the search for a response.
- If the assertions declarative then they do not themselves say anything about how they will be examined. In case of procedural representation, they say how they will examine.

Declarative Knowledge

- A statement in which knowledge specified, but the use to which that knowledge is to be put is not given.
- For example, laws, people's name; these are the facts which can stand alone, not dependent on other knowledge;
- So to use declarative representation, we must have a program that explains what is to do with the knowledge and how.
- For example, a set of logical assertions can combine with a resolution theorem prover to give a complete program for solving problems but in some cases, the logical assertions can view as a program rather than data to a program.
- Hence the implication statements define the legitimate reasoning paths and automatic assertions provide the starting points of those paths.
- These paths define the execution paths which is similar to the 'if then else "in traditional programming.
- So logical assertions can view as a procedural representation of knowledge.
-

Logic Programming – Representing Knowledge Using Rules

- Logic programming is a programming paradigm in which logical assertions viewed as programs.
- These are several logic programming systems, PROLOG is one of them.
- ***A PROLOG program consists of several logical assertions where each is a horn clause i.e. a clause with at most one positive literal.***
- Ex : P, P V Q, P $\rightarrow$ Q
- The facts are represented on Horn Clause for two reasons.
    1. Because of a uniform representation, a simple and efficient interpreter can write.
    2. The logic of Horn Clause decidable.

- Also, The first two differences are the fact that PROLOG programs are actually sets of Horn clause that have been transformed as follows:-
    1. If the Horn Clause contains no negative literal then leave it as it is.
    2. Also, Otherwise rewrite the Horn clauses as an implication, combining all of the negative literals into the antecedent of the implications and the single positive literal into the consequent.
- Moreover, This procedure causes a clause which originally consisted of a disjunction of literals (one of them was positive) to be transformed into a single implication whose antecedent is a conjunction universally quantified.
- But when we apply this transformation, any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent are still universally quantified.

For example the PROLOG clause P(x): – Q(x, y) is equal to logical expression ∀x: ∃y: Q (x, y) → P(x).

- The difference between the logic and PROLOG representation is that the PROLOG interpretation has a fixed control strategy. And so, the assertions in the PROLOG program define a particular search path to answer any question.
- But, the logical assertions define only the set of answers but not about how to choose among those answers if there is more than one.

Consider the following example:
1. Logical representation

∀x : pet(x) ∧ small (x) → apartmentpet(x)
∀x : cat(x) ∧ dog(x) → pet(x)
∀x : poodle (x) → dog (x) ∧ small (x)
poodle (fluffy)

2. Prolog representation

apartmentpet (x) : pet(x), small (x)
pet (x): cat (x)
pet (x): dog(x)
dog(x): poodle (x)
small (x): poodle(x)
poodle (fluffy)

Forward versus Backward Reasoning

Forward versus Backward Reasoning
A search procedure must find a path between initial and goal states.
There are two directions in which a search process could proceed.
The two types of search are:
1. Forward search which starts from the start state
2. Backward search that starts from the goal state

The production system views the forward and backward as symmetric processes.
Consider a game of playing 8 puzzles. The rules defined are
Square 1 empty and square 2 contains tile n. →

- *Also, Square 2 empty and square 1 contains the tile n.*

Square 1 empty Square 4 contains tile n. →

- *Also, Square 4 empty and Square 1 contains tile n.*

We can solve the problem in 2 ways:

1. Reason forward from the initial state
   - Step 1. Begin building a tree of move sequences by starting with the initial configuration at the root of the tree.
   - Step 2. Generate the next level of the tree by finding all rules ***whose left-hand side matches*** against the root node. The right-hand side is used to create new configurations.
   - Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose left-hand side match.
2. Reasoning backward from the goal states:
   - Step 1. Begin building a tree of move sequences by starting with the goal node configuration at the root of the tree.
   - Step 2. Generate the next level of the tree by finding all rules ***whose right-hand side matches*** against the root node. The left-hand side used to create new configurations.
   - Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose right-hand side match.
   - So, The same rules can use in both cases.
   - Also, In forwarding reasoning, the left-hand sides of the rules matched against the current state and right sides used to generate the new state.
   - Moreover, In backward reasoning, the right-hand sides of the rules matched against the current state and left sides are used to generate the new state.

There are four factors influencing the type of reasoning. They are,

1. Are there more possible start or goal state? We move from smaller set of sets to the length.
2. In what direction is the branching factor greater? We proceed in the direction with the lower branching factor.
3. Will the program be asked to justify its reasoning process to a user? If, so then it is selected since it is very close to the way in which the user thinks.
4. What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new factor, the forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

Example 1 of Forward versus Backward Reasoning

- It is easier to drive from an unfamiliar place from home, rather than from home to an unfamiliar place. Also, If you consider a home as starting place an unfamiliar place as a goal then we have to backtrack from unfamiliar place to home.

Example 2 of Forward versus Backward Reasoning

- Consider a problem of symbolic integration. Moreover, The problem space is a set of formulas, which contains integral expressions. Here START is equal to the given formula with some integrals. GOAL is equivalent to the expression of the formula without any integral. Here we start from the formula with some integrals and proceed to an integral free expression rather than starting from an integral free expression.

Example 3 of Forward versus Backward Reasoning

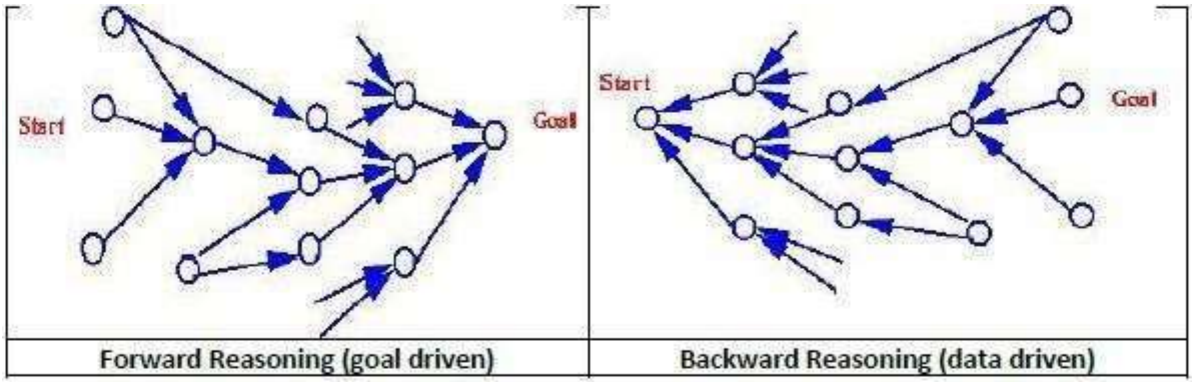
- The third factor is nothing but deciding whether the reasoning process can justify its reasoning. If it justifies then it can apply. For example, doctors are usually unwilling to accept any advice from diagnostics process because it cannot explain its reasoning.

Example 4 of Forward versus Backward Reasoning

- Prolog is an example of backward chaining rule system. In Prolog rules restricted to Horn clauses. This allows for rapid indexing because all the rules for deducing a given fact share the same rule head. Rules matched with unification procedure. Unification tries to find a set of bindings for variables to equate a sub-goal with the head of some rule. Rules in the Prolog program matched in the order in which they appear.

Combining Forward and Backward Reasoning

- Instead of searching either forward or backward, you can search both simultaneously.
- Also, That is, start forward from a starting state and backward from a goal state simultaneously until the paths meet.
- This strategy called Bi-directional search. The following figure shows the reason for a Bidirectional search to be ineffective.


Start
Goal
Forward Reasoning (goal driven)
Start
Goal
Backward Reasoning (data driven)

- Based on the form of the rules one can decide whether the same rules can apply to both forward and backward reasoning.
- Moreover, If left-hand side and right of the rule contain pure assertions then the rule can reverse.
- And so the same rule can apply to both types of reasoning.
- If the right side of the rule contains an arbitrary procedure then the rule cannot reverse.
- So, In this case, while writing the rule the commitment to a direction of reasoning must make.

Symbolic Reasoning Under Uncertainty

Symbolic Reasoning

- The reasoning is the act of deriving a conclusion from certain properties using a given methodology.
- The reasoning is a process of thinking; reasoning is logically arguing; reasoning is drawing the inference.
- *When a system is required to do something, that it has not been explicitly told how to do, it must reason. It must figure out what it needs to know from what it already knows.*

MODULE 2

INTRODUCTION

Ever since puters were invented, we have wondered whether they might be made to learn. If we could understand how to program them to learn-to improve automatically with experience-the impact would be dramatic.

- Imagine puters learning from medical records which treatments are most effective for new diseases
- Houses learning from experience to optimize energy costs based on the particular usage patterns of their occupants.
- Personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the online morning newspaper

A successful understanding of how to make puters learn would open up many new uses of puters and new levels of petence and customization

Some successful applications of machine learning

- Learning to recognize spoken words
- Learningtodriveanautonomousvehicle
- Learning to classify new astronomical structures
- Learning to play world-class backgammon

Why is Machine Learning Important?

- Some tasks cannot be defined well, except by examples (e.g., recognizing people).
- Relationships and correlations can be hidden within large amounts of data. Machine Learning/Data Mining may be able to find these relationships.
- Human designers often produce machines that do not work as well as desired in the environments in which they are used.
- The amount of knowledge available about certain tasks might be too large for explicit encoding by humans (e.g., medical diagnostic).
- Environments change over time.
- New knowledge about tasks is constantly being discovered by humans. It may be difficult to continuously re-design systems “by hand”.

CONCEPT LEARNING

- Learning involves acquiring general concepts from specific training examples. Example: People continually learn general concepts or categories such as "bird," "car," "situations in which I should study more in order to pass the exam," etc.
- Each such concept can be viewed as describing some subset of objects or events defined over a larger set
- Alternatively, each concept can be thought of as a Boolean-valued function defined over this larger set. (Example: A function defined over all animals, whose value is true for birds and false for other animals).

Definition: Concept learning - Inferring a Boolean-valued function from training examples of its input and output

A CONCEPT LEARNING TASK

Consider the example task of learning the target concept "Days on which ***Aldo*** enjoys his favorite water sport"

Example	**Sky**	**AirTemp**	**Humidity**	**Wind**	**Water**	**Forecast**	**EnjoySport**
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Table: Positive and negative training examples for the target concept ***EnjoySport.***

The task is to learn to predict the value of ***EnjoySport*** for an arbitrary day, based on the values of its other attributes?

What hypothesis representation is provided to the learner?

- Let's consider a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes.
- Let each hypothesis be a vector of six constraints, specifying the values of the six attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*.

For each attribute, the hypothesis will either

- Indicate by a "?' that any value is acceptable for this attribute,
- Specify a single required value (e.g., Warm) for the attribute, or
- Indicate by a "Φ" that no value is acceptable

If some instance ***x*** satisfies all the constraints of hypothesis ***h***, then ***h*** classifies ***x*** as a positive example (***h(x) = 1***).

The hypothesis that ***PERSON*** enjoys his favorite sport only on cold days with high humidity is represented by the expression

(?, Cold, High, ?, ?, ?)

The most general hypothesis-that every day is a positive example-is represented by (?, ?, ?, ?, ?, ?)

The most specific possible hypothesis-that no day is a positive example-is represented by (Φ, Φ, Φ, Φ, Φ, Φ)

<u>Notation</u>

- The set of items over which the concept is defined is called the ***set of instances***, which is denoted by X.

Example: X is the set of all possible days, each represented by the attributes: Sky, AirTemp, Humidity, Wind, Water, and Forecast

- The concept or function to be learned is called the ***target concept***, which is denoted by c. c can be any Boolean valued function defined over the instances X

$$c: X \rightarrow \{O, 1\}$$

Example: The target concept corresponds to the value of the attribute ***EnjoySport*** (i.e., c(x) = 1 if ***EnjoySport*** = Yes, and c(x) = 0 if ***EnjoySport*** = No).

- Instances for which c(x) = 1 are called ***positive examples***, or members of the target concept.
- Instances for which c(x) = 0 are called ***negative examples***, or non-members of the target concept.
- The ordered pair (x, c(x)) to describe the training example consisting of the instance x and its target ***concept value c(x).***
- ***D*** to denote the set of available training examples

- The symbol ***H*** to denote the set of all possible hypotheses that the learner may consider regarding the identity of the target concept. Each hypothesis ***h*** in ***H*** represents a Boolean-valued function defined over ***X***

$$h: X \rightarrow \{O, 1\}$$

The goal of the learner is to find a hypothesis h such that h(x) = c(x) for all x in X.

- Given:
  - Instances X: Possible days, each described by the attributes
    - *Sky* (with possible values Sunny, Cloudy, and Rainy),
    - *AirTemp* (with values Warm and Cold),
    - *Humidity* (with values Normal and High),
    - *Wind* (with values Strong and Weak),
    - *Water* (with values Warm and Cool),
    - *Forecast* (with values Same and Change).
  - Hypotheses *H*: Each hypothesis is described by a conjunction of constraints on the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The constraints may be "?" (any value is acceptable), “Φ” (no value is acceptable), or a specific value.
  - Targetconcept*c*:***EnjoySport***:X→{0,l}
  - Training examples *D*: Positive and negative examples of the target function
- Determine:
  - A hypothesis h in H such that *h(x) = c(x)* for all *x* in X.

Table: The ***EnjoySport*** concept learning task.

The inductive learning hypothesis

Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

CONCEPT LEARNING AS SEARCH

- Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation.
- The goal of this search is to find the hypothesis that best fits the training examples.

Example:
Consider the instances X and hypotheses H in the *EnjoySport* learning task. The attribute Sky has three possible values, and *AirTemp, Humidity, Wind, Water, Forecast* each have two possible values, the instance space X contains exactly

3.2.2.2.2.2 = 96 distinct instances

5.4.4.4.4.4 = 5120 syntactically distinct hypotheses within H.

Every hypothesis containing one or more "Φ" symbols represents the empty set of instances; that is, it classifies every instance as negative.

1 + (4.3.3.3.3.3) = 973. Semantically distinct hypotheses

General-to-Specific Ordering of Hypotheses

Considerthetwohypotheses h_1 = (Sunny, ?, ?, Strong, ?, ?)
h_2 = (Sunny, ?, ?, ?, ?, ?)

- Consider the sets of instances that are classified positive by h_l and by h_2.
- h_2 imposes fewer constraints on the instance, it classifies more instances as positive. So, any instance classified positive by h_l will also be classified positive by h_2. Therefore, h_2 is more general than h_l.

Given hypotheses h_j and h_k, h_j is more-general-than or- equal do h_k if and only if any instance that satisfies h_k also satisfies h_i

Definition: Let h_j and h_k be Boolean-valued functions defined over X. Then h_j is ***more general-than-or-equal-to*** h_k (written $h_j \geq h_k$) if and only if

$$(\quad x \quad X) [(h_k (x) = 1) \rightarrow (h_j (x) = 1)]$$

- In the figure, the box on the left represents the set X of all instances, the box on the right the set H of all hypotheses.
- Each hypothesis corresponds to some subset of X-the subset of instances that it classifies positive.
- The arrows connecting hypotheses represent the more - general -than relation, with the
- arrowpointingtowardthelessgeneralhypothesis. Note the subset of instances characterized by h2 subsumes the subset characterized by hl , hence h2 is more - general– than h1

FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

FIND-S Algorithm

```
1. Initialize $h$ to the most specific hypothesis in $H$
2. For each positive training instance $x$
     For each attribute constraint $a_i$ in $h$
        If the constraint $a_i$ is satisfied by $x$
              Then do nothing
        Else replace $a_i$ in $h$ by the next more general constraint that is
        satisfied by $x$
3. Output hypothesis $h$
```

To illustrate this algorithm, assume the learner is given the sequence of training examples from the ***EnjoySport*** task

Example	**Sky**	**AirTemp**	**Humidity**	**Wind**	**Water**	**Forecast**	**EnjoySport**
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

- The first step of FIND-S is to initialize h to the most specific hypothesis in H h - (Ø, Ø, Ø, Ø, Ø, Ø)

- Consider the first training example

  x_1 = <Sunny Warm Normal Strong Warm Same>, +

  Observing the first training example, it is clear that hypothesis *h* is too specific. None of the "Ø" constraints in h are satisfied by this example, so each is replaced by the next *more general constraint* that fits the example

  h_1 = <Sunny Warm Normal Strong Warm Same>

  Considerthesecondtrainingexample x_2 = <Sunny, Warm, High, Strong, Warm, Same>, +

  The second training example forces the algorithm to further generalize h, this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example

  h_2 = <Sunny Warm ? Strong Warm Same>

- Consider the third training example

  x3 = <Rainy, Cold, High, Strong, Warm, Change>, -

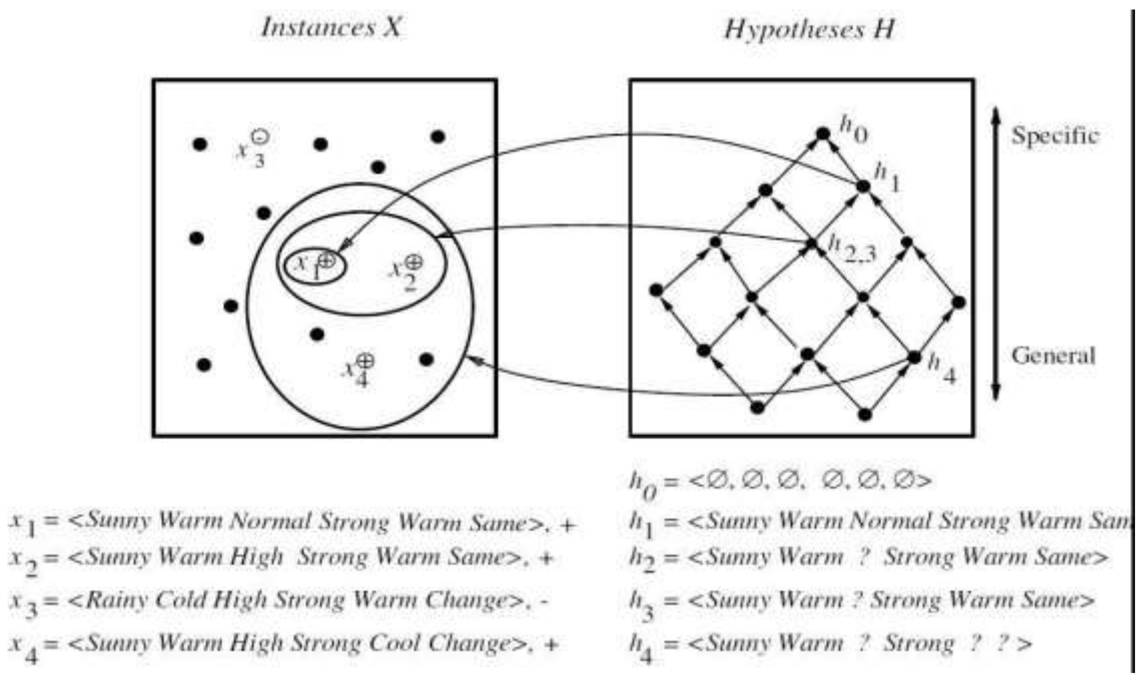
  Upon encountering the third training the algorithm makes no change to h. The FIND-S algorithm simply ignores every negative example.

  h_3 = < Sunny Warm ? Strong Warm Same>

- Consider the fourth training example

  x_4 = <Sunny Warm High Strong Cool Change>, +

  The fourth example leads to a further generalization of h h_4 = < Sunny Warm ? Strong ? ? >


Instances X
Hypotheses H
Specific
General
x_1 = <Sunny Warm Normal Strong Warm Same>, +
x_2 = <Sunny Warm High Strong Warm Same>, +
x_3 = <Rainy Cold High Strong Warm Change>, -
x_4 = <Sunny Warm High Strong Cool Change>, +
h_0 = <∅, ∅, ∅, ∅, ∅, ∅>
h_1 = <Sunny Warm Normal Strong Warm San
h_2 = <Sunny Warm ? Strong Warm Same>
h_3 = <Sunny Warm ? Strong Warm Same>
h_4 = <Sunny Warm ? Strong ? ? >

The key property of the FIND-S algorithm

FIND-S is guaranteed to output the most specific hypothesis within H that is consistent with the positive training examples

- FIND-S algorithm's final hypothesis will also be consistent with the negative examples provided the correct target concept is contained in H, and provided the training examples are correct.

Unanswered by FIND-S

1. Has the learner converged to the correct target concept?
2. Why prefer the most specific hypothesis?
3. Are the training examples consistent?
4. What if there are several maximally specific consistent hypotheses?

VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM

The key idea in the CANDIDATE-ELIMINATION algorithm is to output a description of the set of all ***hypotheses consistent with the training examples***

Representation

Definition: consistent- A hypothesis h is **consistent** with **a** set of training examples ***D*** if **and** only if ***h(x)*** = ***c(x)*** for each example ***(x, c(x))*** in ***D.***

$$Consistent\ (h, D) \quad (\quad x, c(x)\ D)\ h(x) = c(x))$$

Note difference between definitions of *consistent* and *satisfies*

- An example *x* is said to ***satisfy*** hypothesis ***h*** when ***h(x)*** = 1, regardless of whether *x* is a positive or negative example of the target concept.
- An example *x* is said to ***consistent*** with hypothesis ***h*** iff ***h(x)*** = ***c(x)***

Definition: version space- The **version space,** denoted $VS_{H,D}$ with respect to hypothesis space ***H*** and training examples D, is the subset of hypotheses from ***H*** consistent with the training examples in D

$$VS_{H,D}\{hH|Consistent(h, D)\}$$

The LIST-THEN-ELIMINATION algorithm

The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in H and then eliminates any hypothesis found inconsistent with any training example.

```
1. VersionSpace c a list containing every hypothesis in H
2. For each training example, (x, c(x))
       remove from VersionSpace any hypothesis h for which h(x) ≠ c(x)
3. Output the list of hypotheses in VersionSpace
```

The LIST-THEN-ELIMINATE Algorithm

- List-Then-Eliminate works in principle, so long as version space is finite.
- However, since it requires exhaustive enumeration of all hypotheses in practice it is not feasible.

A More pact Representation for Version Spaces

The version space is represented by its most general and least general members. These members form general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.

Definition: The **general boundary** G, with respect to hypothesis space ***H*** and training data *D*, is the set of maximally general members of *H* consistent with *D*

$$G\ \{g\quad H \mid Consistent\ (g, D)\ (\quad g' \quad H)[(g'\ _{g}\ g) \quad Consistent(g', D)]\}$$

Definition: The **specific boundary** S, with respect to hypothesis space *H* and training data *D*, is the set of minimally general (i.e., maximally specific) members of *H* consistent with *D.*

$$S\ \{s \quad H \mid Consistent\ (s, D)\ (\quad s' \quad H)[(s\ _{g}\ s') \quad Consistent(s', D)]\}$$

Theorem: Version Space representation theorem

Theorem: Let X be an arbitrary set of instances and Let H be a set of Boolean-valued hypothesesdefinedoverX.Letc:X→{O,1}beanarbitrary targetconceptdefinedover X, and let D be an arbitrary set of training examples {(x, c(x))). For all X, H, c, and D such that S and G are well defined,

$$VS_{H,D} = \{ h \quad H \mid (\; s \quad S) (\; g \quad G) (g \;\; _{g} h \;\; _{g} s) \}$$

To Prove:

1. Every h satisfying the right hand side of the above expression is in $\boldsymbol{VS_{H,D}}$

2. Every member of $\boldsymbol{VS_{H,D}}$ satisfies the right-hand side of the expression

Sketch of proof:

1. let g, h, s be arbitrary members of G, H, S respectively with $g \;\; _g h \;\; _g s$

- By the definition of **S, s** must be satisfied by all positive examples in D. Because $h \; _g s$, h must also be satisfied by all positive examples in D.
- By the definition of ***G,*** g cannot be satisfied by any negative example in D, and because $g \; _g h$ h cannot be satisfied by any negative example in D. Because h is satisfied by all positive examples in D and by no negative examples in D, h is consistent with D, and therefore h is a member of $VS_{H,D}$.

2. It can be proven by assuming some h in $VS_{H,D}$,that does not satisfy the right-hand side of the expression, then showing that this leads to an inconsistency

CANDIDATE-ELIMINATION Learning Algorithm

The CANDIDATE-ELIMINTION algorithm putes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples.

```
Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do
  • If d is a positive example
     • Remove from G any hypothesis inconsistent with d
     • For each hypothesis s in S that is not consistent with d
        • Remove s from S
        • Add to S all minimal generalizations h of s such that
           • h is consistent with d, and some member of G is more general than h
        • Remove from S any hypothesis that is more general than another hypothesis in S

  • If d is a negative example
     • Remove from S any hypothesis inconsistent with d
     • For each hypothesis g in G that is not consistent with d
       •RemovegfromG
        • Add to G all minimal specializations h of g such that
           • h is consistent with d, and some member of S is more specific than h
        • Remove from G any hypothesis that is less general than another hypothesis in G
```

CANDIDATE- ELIMINTION algorithm using version spaces

An Illustrative Example

Example	**Sky**	**AirTemp**	**Humidity**	**Wind**	**Water**	**Forecast**	**EnjoySport**
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

CANDIDATE-ELIMINTION algorithm begins by initializing the version space to the set of all hypotheses in H;

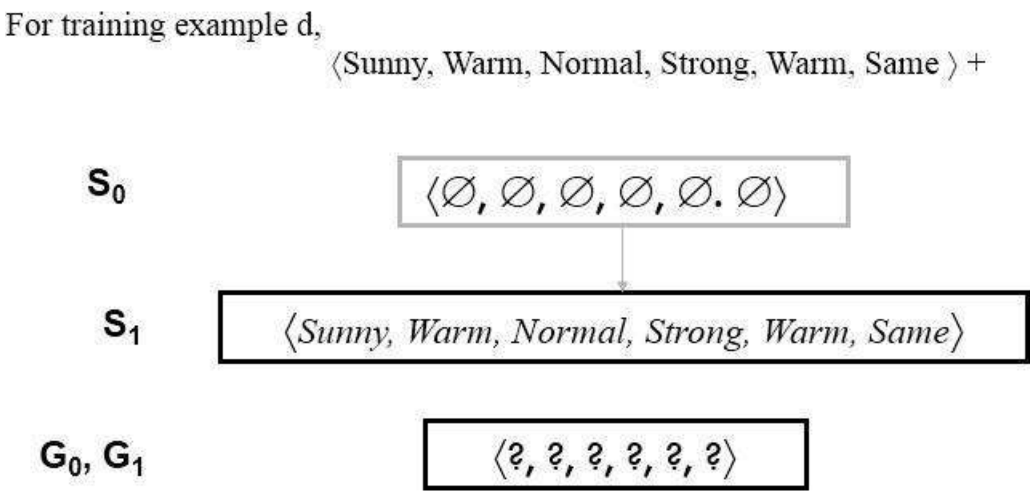
Initializing the G boundary set to contain the most general hypothesis in H

G_0 ?, ?, ?, ?, ?,?

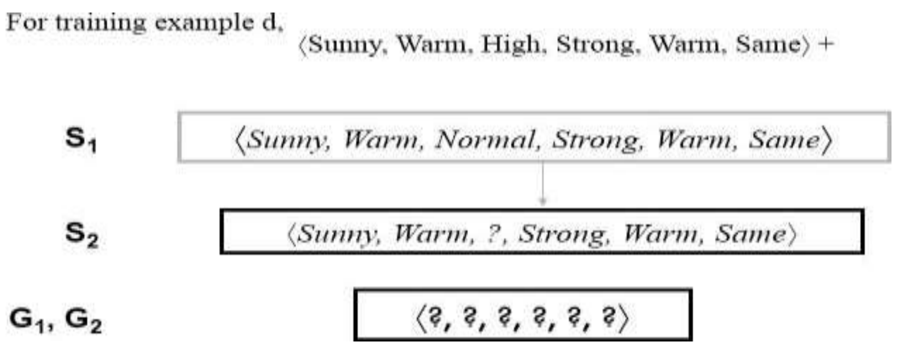
Initializing the S boundary set to contain the most specific (least general) hypothesis

S_0,,,,,

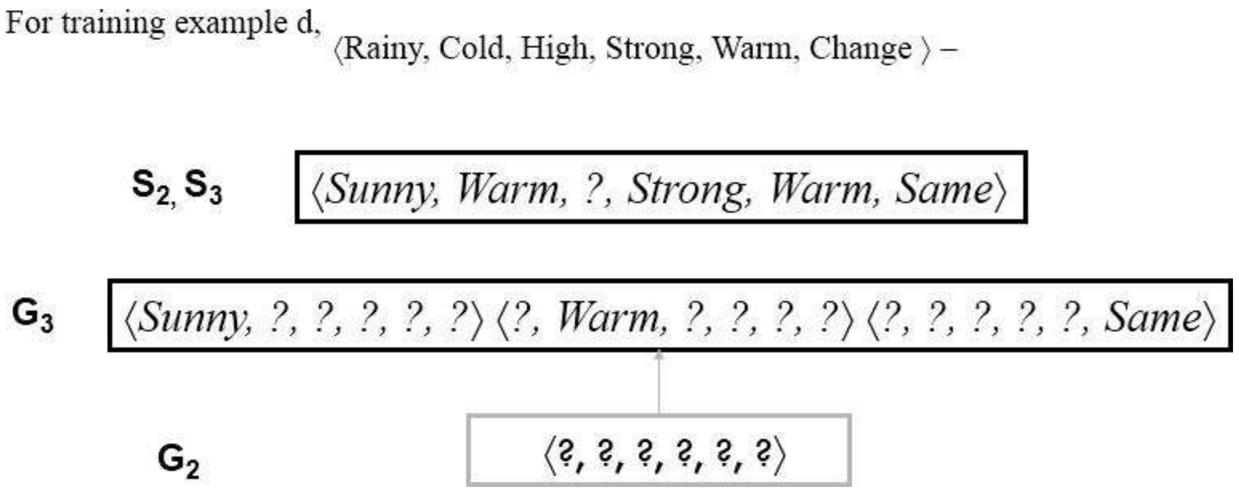
- When the first training example is presented , the CANDIDATE-ELIMINTION algorithm checks the S boundary and finds that it is overly specific and it fails to cover the positive example.
- The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example
- No update of the G boundary is needed in response to this training example because G_o correctly covers this example


For training example d,
⟨Sunny, Warm, Normal, Strong, Warm, Same ⟩ +
S0
⟨∅, ∅, ∅, ∅, ∅. ∅⟩
S1
⟨Sunny, Warm, Normal, Strong, Warm, Same⟩
G0, G1
⟨?, ?, ?, ?, ?, ?⟩

- When the second training example is observed, it has a similar effect of generalizing S further to S_2, leaving G again unchanged i.e., $G_2 = G_1 = G_0$


For training example d, ⟨Sunny, Warm, High, Strong, Warm, Same⟩ +
S_1 ⟨Sunny, Warm, Normal, Strong, Warm, Same⟩
S_2 ⟨Sunny, Warm, ?, Strong, Warm, Same⟩
G_1, G_2 ⟨?, ?, ?, ?, ?, ?⟩

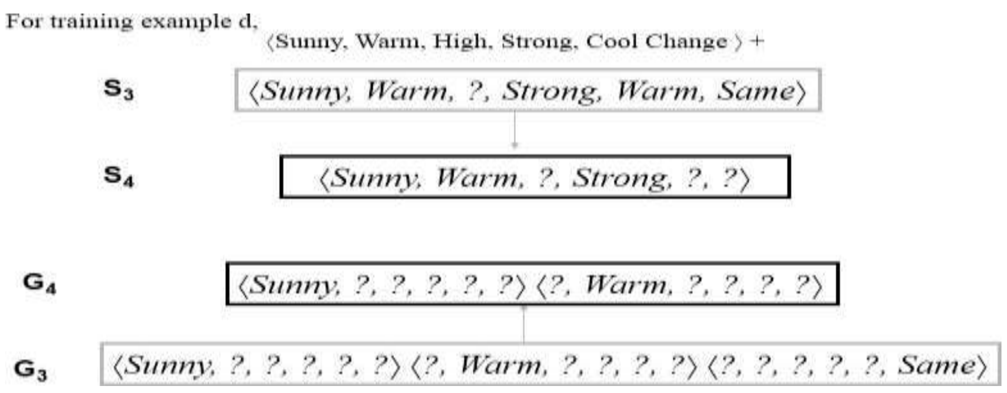
- <u>Consider the third training example</u>. This negative example reveals that the G boundary of the version space is overly general, that is, the hypothesis in G incorrectly predicts that this new example is a positive example.
- The hypothesis in the G boundary must therefore be specialized until it correctly classifies this new negative example


For training example d, ⟨Rainy, Cold, High, Strong, Warm, Change ⟩ –
S_2, S_3 ⟨Sunny, Warm, ?, Strong, Warm, Same⟩
G_3 ⟨Sunny, ?, ?, ?, ?, ?⟩ ⟨?, Warm, ?, ?, ?, ?⟩ ⟨?, ?, ?, ?, ?, Same⟩
G_2 ⟨?, ?, ?, ?, ?, ?⟩

<u>Given that there are six attributes that could be specified to specialize G2, why are there only three new hypotheses in G3?</u>

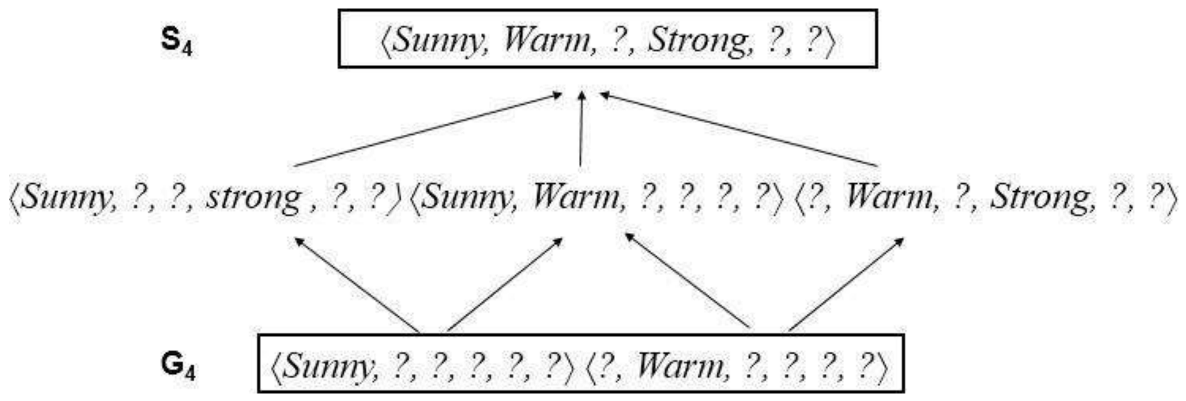
For example, the hypothesis h = (?, ?, Normal, ?, ?, ?) is a minimal specialization of G2 thatcorrectlylabelsthenewexampleasanegativeexample,butitisnotincluded in G3. The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples

- <u>Consider the fourth training example</u>.


For training example d, ⟨Sunny, Warm, High, Strong, Cool Change ⟩ +
S_3 ⟨Sunny, Warm, ?, Strong, Warm, Same⟩
S_4 ⟨Sunny, Warm, ?, Strong, ?, ?⟩
G_4 ⟨Sunny, ?, ?, ?, ?, ?⟩ ⟨?, Warm, ?, ?, ?, ?⟩
G_3 ⟨Sunny, ?, ?, ?, ?, ?⟩ ⟨?, Warm, ?, ?, ?, ?⟩ ⟨?, ?, ?, ?, ?, Same⟩

- This positive example further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example

After processing these four examples, the boundary sets S_4 and G_4 delimit the version space of all hypotheses consistent with the set of incrementally observed training examples.


S4
⟨Sunny, Warm, ?, Strong, ?, ?⟩
⟨Sunny, ?, ?, strong , ?, ?⟩
⟨Sunny, Warm, ?, ?, ?, ?⟩
⟨?, Warm, ?, Strong, ?, ?⟩
G4
⟨Sunny, ?, ?, ?, ?, ?⟩ ⟨?, Warm, ?, ?, ?, ?⟩

INDUCTIVE BIAS

The fundamental questions for inductive inference

1. What if the target concept is not contained in the hypothesis space?
2. Can we avoid this difficulty by using a hypothesis space that includes every possible hypothesis?
3. How does the size of this hypothesis space influence the ability of the algorithm to generalize to unobserved instances?
4. How does the size of the hypothesis space influence the number of training examples that must be observed?

These fundamental questions are examined in the context of the CANDIDATE-ELIMINTION algorithm

<u>A Biased Hypothesis Space</u>

- Suppose the target concept is not contained in the hypothesis space *H*, then obvious solution is to enrich the hypothesis space to include every possible hypothesis.
  - Consider the ***EnjoySport*** example in which the hypothesis space is restricted to include onlyconjunctionsofattributevalues.Becauseofthisrestriction,thehypothesisspace is unable to represent even simple disjunctive target concepts such as

"Sky = Sunny or Sky = Cloudy."

  - The following three training examples of disjunctive hypothesis, the algorithm would find that there are zero hypotheses in the version space

Sunny Warm Normal Strong Cool Change	Y
Cloudy Warm Normal Strong Cool Change	Y

Rainy Warm Normal Strong Cool Change N

- If Candidate Elimination algorithm is applied, then it end up with empty Version Space. After first two training example

  S= ? Warm Normal Strong Cool Change

- This new hypothesis is overly general and it incorrectly covers the third negative training example! So H does not include the appropriate c.
- In this case, a more expressive hypothesis space is required.

An Unbiased Learner

- The solution to the problem of assuring that the target concept is in the hypothesis space H is to provide a hypothesis space capable of representing every teachable concept that is representing every possible subset of the instances X.
- The set of all subsets of a set X is called the power set of X

- In the ***EnjoySport*** learning task the size of the instance space X of days described by the six attributes is 96 instances.
- Thus, there are 2^{96} distinct target concepts that could be defined over this instance space and learner might be called upon to learn.
- The conjunctive hypothesis space is able to represent only 973 of these - a biased hypothesis space indeed

- Let us reformulate the ***EnjoySport*** learning task in an unbiased way by defining a new hypothesis space H' that can represent every subset of instances
- The target concept "Sky = Sunny or Sky = Cloudy" could then be described as

(Sunny, ?, ?, ?, ?, ?) v (Cloudy, ?, ?, ?, ?, ?)

The Futility ofBias-FreeLearning

Inductive learning requires some form of prior assumptions, or inductive bias

Definition:

Consider a concept learning algorithm L for the set of instances X.

- Let c be an arbitrary concept defined over X
- Let D_c = {(x , c(x))} be an arbitrary set of training examples of c.
- Let L (x_i, D_c) denote the classification assigned to the instance x_i by L after training on the data D_c.
- The inductive bias of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D_c

- $(\ \ x_i\ \ X)\ [(B\ \ D_c\ \ x_i) \vdash L\ (x_i, D_c)]$

The below figure explains

- Modelling inductive systems by equivalent deductive systems.

- The input-output behavior of the CANDIDATE-ELIMINATION algorithm using a hypothesis space H is identical to that of a deductive theorem prover utilizing the assertion "H contains the target concept." This assertion is therefore called the inductive bias of the CANDIDATE-ELIMINATION algorithm.
- Characterizing inductive systems by their inductive bias allows modelling them by their equivalent deductive systems. This provides a way to pare inductive systems according to their policies for generalizing beyond the observed training data.

MODULE 3

DECISION TREE LEARNING

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree.

DECISION TREE REPRESENTATION

- Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance.
- Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute.
- An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

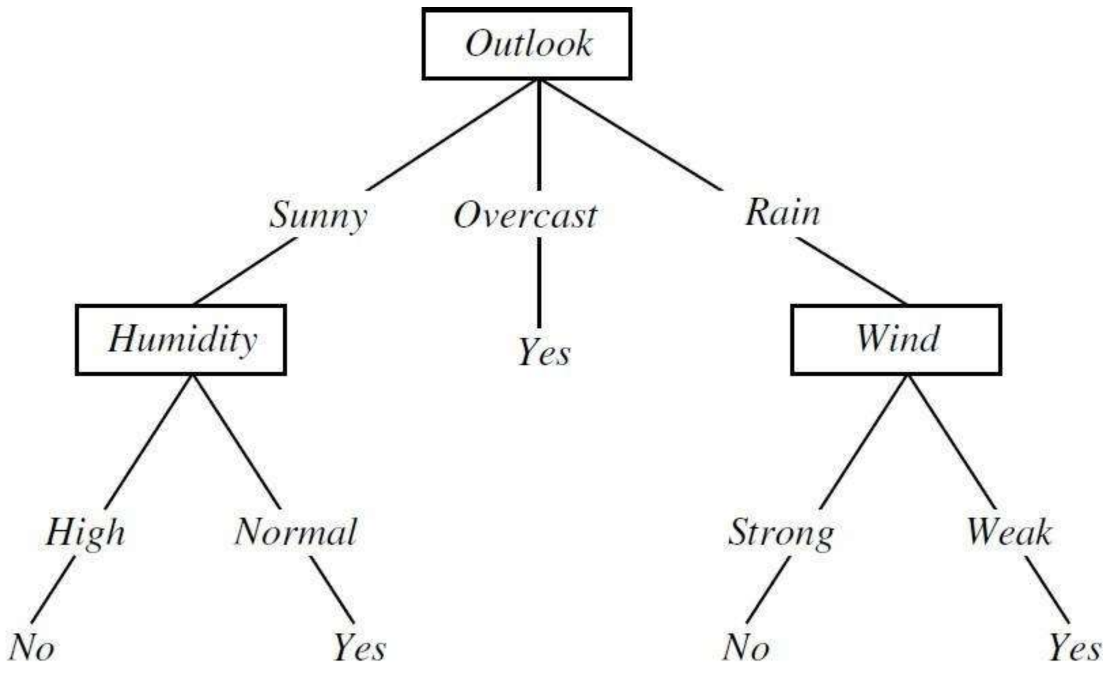


FIGURE: A decision tree for the concept ***PlayTennis***. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf

- Decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances.

- Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions

For example, the decision tree shown in above figure corresponds to the expression

$$(Outlook = Sunny \wedge Humidity = Normal)$$
$$\vee \quad (Outlook = Overcast)$$
$$\vee \quad (Outlook = Rain \wedge Wind = Weak)$$

APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Decision tree learning is generally best suited to problems with the following characteristics:

1. ***Instances are represented by attribute-value pairs*** – Instances are described by a fixed set of attributes and their values

2. ***The target function has discrete output values*** – The decision tree assigns a Boolean classification (e.g.,yesorno) to each example.Decision tree methodseasilyextend to learning functions with more than two possible output values.

3. ***Disjunctive descriptions may be required***

4. ***The training data may contain errors*** – Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.

5. ***The training data may contain missing attribute values*** **–** Decision tree methods can be used even when some training examples have unknown values

THE BASIC DECISION TREE LEARNING ALGORITHM

The basic algorithm is ID3 which learns decision trees by constructing them top-down

ID3(Examples, Target_attribute, Attributes)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a Root node for the tree
- If all Examples are positive, Return the single-node tree Root, with label = +
- If all Examples are negative, Return the single-node tree Root, with label = -
- If Attributes is empty, Return the single-node tree Root, with label = most common value of Target_attribute in Examples

```
• Otherwise Begin
    • A ← the attribute from Attributes that best* classifies Examples
    • ThedecisionattributeforRoot←A
    • For each possible value, $v_i$, of A,
        • Add a new tree branch below *Root*, corresponding to the test A = $v_i$
        • Let *Examples* $_{vi}$, be the subset of Examples that have value $v_i$ for *A*
        • If *Examples* $_{vi}$ , is empty
            • Then below this new branch add a leaf node with label = most
              common value of Target_attribute in Examples
            • Else below this new branch add the subtree
                ID3(*Examples* $_{vi}$, Targe_tattribute, Attributes – {A}))
• End
• Return Root
```

* The best attribute is the one with highest information gain

TABLE: Summary of the ID3 algorithm specialized to learning Boolean-valued functions. ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used

Which Attribute Is the Best Classifier?

- The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree.
- A statistical property called ***information gain*** that measures how well a given attribute separates the training examples according to their target classification.
- ID3 uses ***information gain*** measure to select among the candidate attributes at each step while growing the tree.

ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

To define information gain, we begin by defining a measure called entropy. *Entropy measures the impurity of a collection of examples.*

Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this Boolean classification is

$$Entropy\ (S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

Where, p_+ istheproportionofpositiveexamplesinS p_- is the proportion of negative examples in S.

<u>Example:</u>

Suppose S is a collection of 14 examples of some boolean concept, including 9 positive and 5 negative examples. Then the entropy of S relative to this boolean classification is

$$Entropy([9+, 5-]) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.940$$

- The entropy is 0 if all members of S belong to the same class
- The entropy is 1 when the collection contains an equal number of positive and negative examples
- If the collection contains unequal numbers of positive and negative examples, the entropy is between 0 and 1

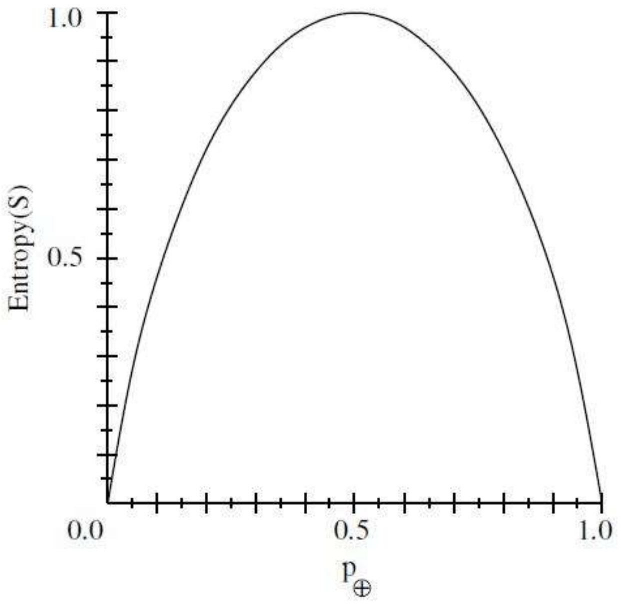


FIGURE The entropy function relative to a boolean classification, as the proportion, $p_\oplus$, of positive examples varies between 0 and 1.

INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

- ***Informationgain,***istheexpectedreductioninentropy.causedcombypartitioningthe examples according to this attribute.
- The information gain, Gain(S, A) of an attribute A, relative to a collection of examples S, is defined as

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

<u>**Example:**</u> Information gain
Let, *Values*(*Wind*) = {*Weak*, *Strong*}
S = [9+, 5−]

$S_{Weak} = [6+, 2-]$

$S_{Strong} = [3+, 3-]$

Information gain of attribute *Wind*:

$$Gain(S, Wind) = Entropy(S) - 8/14\ Entropy\ (S_{Weak}) - 6/14\ Entropy\ (S_{Strong})$$

$$= 0.94 - (8/14)^{*}\ 0.811 - (6/14)\ {}^{*}1.00$$

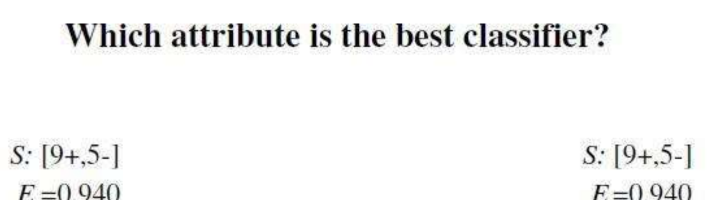
$$= 0.048$$

An Illustrative Example

- To illustrate the operation of ID3, consider the learning task represented by the training examples of below table.
- Here the target attribute ***PlayTennis***, which can have values ***yes*** or ***no*** for different days.
- Consider the first step through the algorithm, in which the topmost node of the decision tree is created.

Day	***Outlook***	***Temperature***	***Humidity***	***Wind***	***PlayTennis***
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

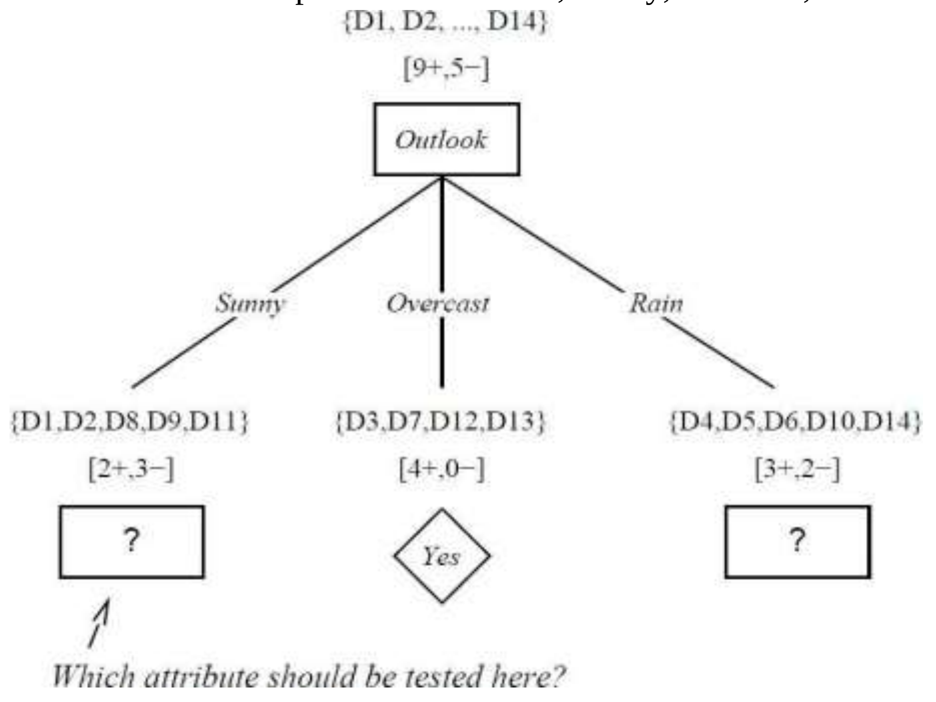
- ID3 determines the information gain for each candidate attribute (i.e., Outlook, Temperature, Humidity, and Wind), then selects the one with highest information gain.



- The information gain values for all four attributes are

Gain(S, Outlook)	= 0.246
Gain(S, Humidity)	= 0.151
Gain(S, Wind)	= 0.048
Gain(S, Temperature)	= 0.029

- According to the information gain measure, the ***Outlook*** attribute provides the best prediction of the target attribute, ***PlayTennis***, over the training examples. Therefore, ***Outlook*** is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values i.e., Sunny, Overcast, and Rain.



S_{sunny} = {D1,D2,D8,D9,D11}

$Gain\ (S_{sunny}, Humidity)$ = .970 − (3/5) 0.0 − (2/5) 0.0 = .970

$Gain\ (S_{sunny}, Temperature)$ = .970 − (2/5) 0.0 − (2/5) 1.0 − (1/5) 0.0 = .570

SRain = { D4, D5, D6, D10, D14}

Gain (SRain , Humidity) = 0.970 – (2/5)1.0 – (3/5)0.917 = 0.019

Gain (SRain , Temperature) =0.970 – (0/5)0.0 – (3/5)0.918 – (2/5)1.0 = 0.019

Gain (SRain , Wind) =0.970 – (3/5)0.0 – (2/5)0.0 = 0.970

HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

- ID3 can be characterized as searching a space of hypotheses for one that fits the training examples.
- The hypothesis space searched by ID3 is the set of possible decision trees.
- ID3 performs a simple-to complex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data

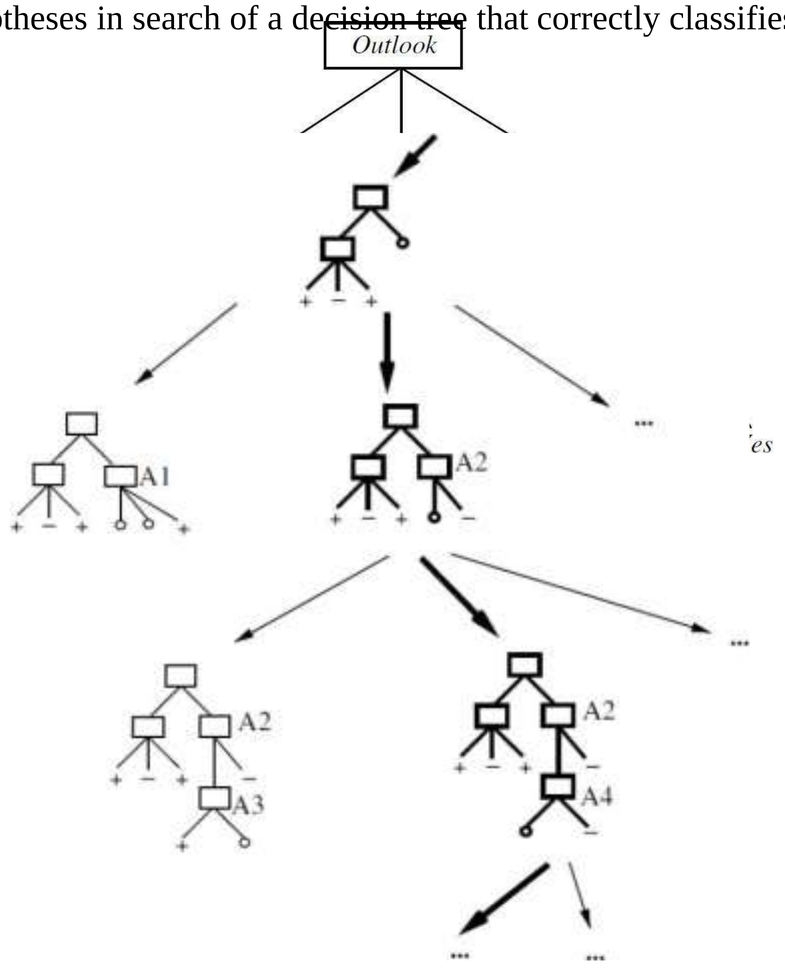


Figure: Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

<u>By viewing ID3 in terms of its search space and search strategy, there are some insight into its capabilities and limitations</u>

1. *ID3's hypothesis space of all decision trees is a complete space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree*

   ID3 avoids one of the major risks of methods that search incomplete hypothesis spaces: that the hypothesis space might not contain the target function.
2. *ID3 maintains only a single current hypothesis as it searches through the space of decision trees.*

   For example, with the earlier version space candidate elimination method, which maintains the set of all hypotheses consistent with the available training examples.

   By determining only a single hypothesis, ID3 loses the capabilities that follow from explicitly representing all consistent hypotheses.

   For example, it does not have the ability to determine how many alternative decision trees are consistent with the available training data, or to pose new instance queries that optimally resolve among these competing hypotheses

3. *ID3 in its pure form performs no backtracking in its search. Once it selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice.*

   In the case of ID3, a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search.

4. *ID3 uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis*.

   One advantage of using statistical properties of all the examples is that the resulting search is much less sensitive to errors in individual training examples.

   ID3 can be easily extended to handle noisy training data by modifying its termination criterion to accept hypotheses that imperfectly fit the training data.

INDUCTIVE BIAS IN DECISION TREE LEARNING

Inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances

Given a collection of training examples, there are typically many decision trees consistent with these examples. Which of these decision trees does ID3 choose?

ID3 search strategy

- Selects in favour of shorter trees over longer ones
- Selects trees that place the attributes with highest information gain closest to the root.

Approximate inductive bias of ID3: Shorter trees are preferred over larger trees

- Consider an algorithm that begins with the empty tree and searches breadth first through progressively more complex trees.
- First considering all trees of depth 1, then all trees of depth 2, etc.
- Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes).
- Let us call this breadth-first search algorithm BFS-ID3.
- BFS-ID3 finds a shortest decision tree and thus exhibits the bias "shorter trees are preferred over longer trees.

A closer approximation to the inductive bias of ID3: Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

- ID3 can be viewed as an efficient approximation to BFS-ID3, using a greedy heuristic search to attempt to find the shortest tree without conducting the entire breadth-first search through the hypothesis space.
- Because ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3.
- In particular, it does not always find the shortest consistent tree, and it is biased to favour trees that place attributes with high information gain closest to the root.

Restriction Biases and Preference Biases

Difference between the types of inductive bias exhibited by ID3 and by the CANDIDATE-ELIMINATION Algorithm.

ID3:

- ID3 searches a complete hypothesis space
- It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met
- Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias

CANDIDATE-ELIMINATION Algorithm:

- The version space CANDIDATE-ELIMINATION Algorithm searches an incomplete hypothesis space
- It searches this space completely, finding every hypothesis consistent with the training data.
- Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias

***Preference bias* –** The inductive bias of ID3 is a preference for certain hypotheses over others (e.g., preference for shorter hypotheses over larger hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is called a preference bias or a search bias.

Restriction bias – The bias of the CANDIDATE ELIMINATION algorithm is in the form of a categorical restriction on the set of hypotheses considered. This form of bias is typically called a restriction bias or a language bias.

Which type of inductive bias is preferred in order to generalize beyond the training data, a preference bias or restriction bias?

- A preference bias is more desirable than a restriction bias, because it allows the learner to work within a complete hypothesis space that is assured to contain the unknown target function.
- In contrast, a restriction bias that strictly limits the set of potential hypotheses is generally less desirable, because it introduces the possibility of excluding the unknown target function altogether.

Why PreferShortHypotheses?Occam's razor

- Occam's razor: is the problem-solving principle that the simplest solution tends to be the right one. When presented with competing hypotheses to solve a problem, one should select the solution with the fewest assumptions.

- Occam's razor: “Prefer the simplest hypothesis that fits the data”.

Argument in favour of Occam’s razor:

- Fewer short hypotheses than long ones:
  - Short hypotheses fits the training data which are less likely to be coincident
  - Longer hypotheses fits the training data might be coincident.

- Many complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data.

Argument opposed:

- There are few small trees, and our priori chance of finding one consistent with an arbitrary set of data is therefore small. The difficulty here is that there are very many small sets of hypotheses that one can define but understood by fewer learner.
- The size of a hypothesis is determined by the representation used internally by the learner. Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners, both justifying their contradictory conclusions by Occam's razor. On this basis we might be tempted to reject Occam's razor altogether.

ISSUES IN DECISION TREE LEARNING

Issues in learning decision trees include

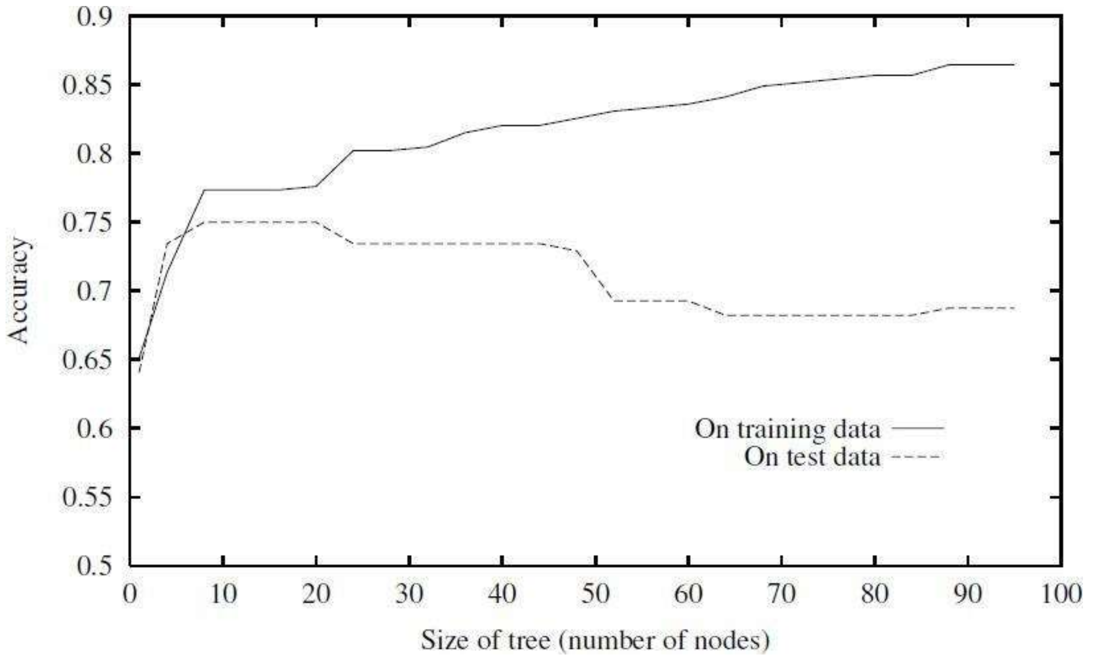
1. Avoiding Overfitting the Data
   Reduced error pruning
   Rule post-pruning
2. Incorporating Continuous-Valued Attributes
3. Alternative Measures for Selecting Attributes
4. HandlingTrainingExampleswithMissingAttributeValues
5. Handling Attributes with Differing Costs

1. Avoiding Overfitting the Data

- The ID3 algorithm grows each branch of the tree just deeply enough to perfectly classify the training examples but it can lead to difficulties when there is noise in the data, or when the number of training examples is too small to produce a representative sample of the true target function. This algorithm can produce trees that overfit the training examples.

- ***Definition - Overfit:*** Given a hypothesis space H, a hypothesis h ∈ H is said to overfit the training data if there exists some alternative hypothesis h' ∈ H, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

The below figure illustrates the impact of overfitting in a typical application of decision tree learning.



- *The horizontal axis* of this plot indicates the total number of nodes in the decision tree, as the tree is being constructed. The vertical axis indicates the accuracy of predictions made by the tree.

  The solid line shows the accuracy of the decision tree over the training examples. The broken line shows accuracy measured over an independent set of test example

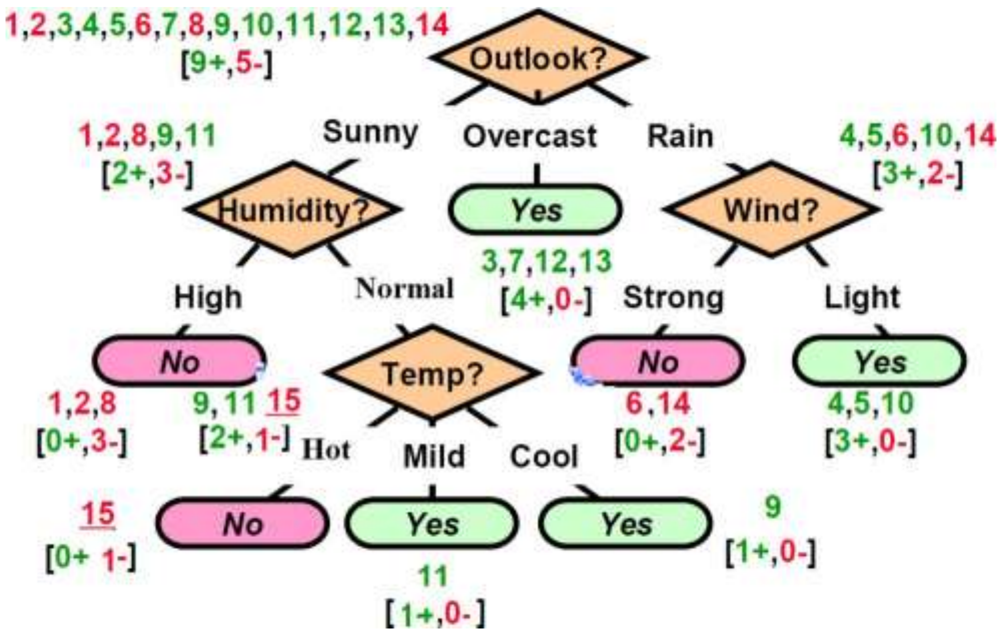
- The accuracy of the tree over the training examples increases monotonically as the tree is grown. The accuracy measured over the independent test examples first increases, then decreases.

How can it be possible for tree h to fit the training examples better than h', but for it to perform more poorly over subsequent examples?

1. Overfitting can occur when the training examples contain random errors or noise
2. When small numbers of examples are associated with leaf nodes.

<u>Noisy Training Example</u>

- Example 15: <Sunny, Hot, Normal, Strong, ->
- Example is noisy because the correct label is +
- Previously constructed tree misclassifies it



<u>Approaches to avoiding overfitting in decision tree learning</u>

- Pre-pruning (avoidance): Stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data
- Post-pruning(recovery):Allowthetreetooverfitthe.data,COMandthenpost-prunethe tree

<u>Criterion used to determine the correct final tree size</u>

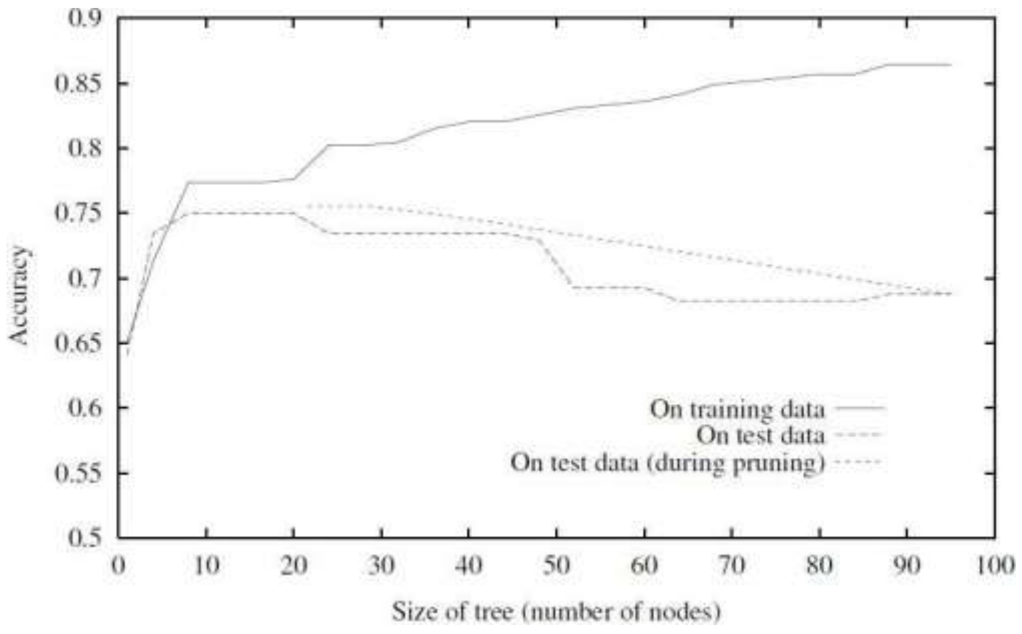
- Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree
- Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond the training set

- Use measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized. This approach is called the Minimum Description Length
  MDL – Minimize : size(tree) + size (misclassifications(tree))

Reduced-Error Pruning

- Reduced-error pruning, is to consider each of the decision nodes in the tree to be candidates for pruning
- ***Pruning*** a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node
- Nodes are removed only if the resulting pruned tree performs no worse than-the original over the validation set.
- Reduced error pruning has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set

The impact of reduced-error pruning on the accuracy of the decision tree is illustrated in below figure



- The additional line in figure shows accuracy over the test examples as the tree is pruned. When pruning begins, the tree is at its maximum size and lowest accuracy over the test set. As pruning proceeds, the number of nodes is reduced and accuracy over the test set increases.
- The available data has been split into three subsets: the training examples, the validation examples used for pruning the tree, and a set of test examples used to provide an unbiased estimate of accuracy over future unseen examples. The plot shows accuracy over the training and test sets.

<u>Pros and Cons</u>

Pro: Produces smallest version of most accurate *T* (subtree of *T*)

Con: Uses less data to construct *T*

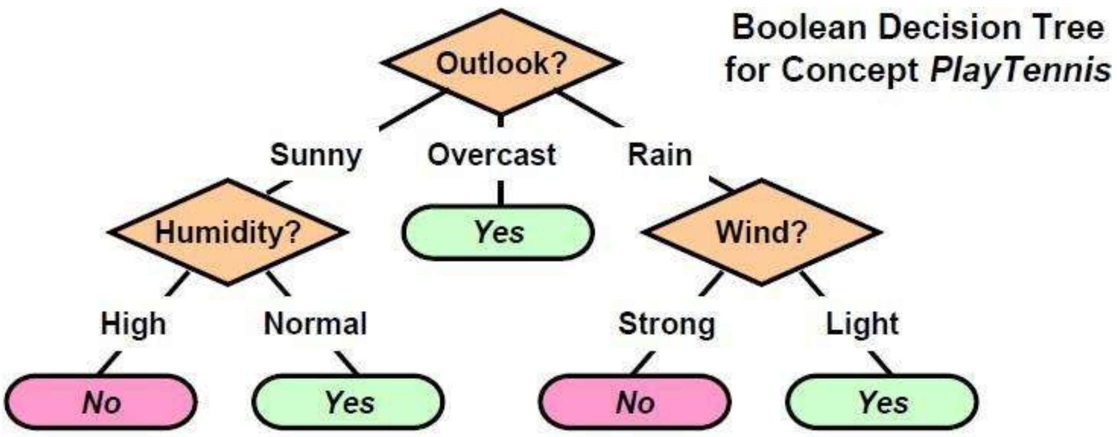
Can afford to hold out $D_{validation}$?. If not (data is too limited), may make error worse (insufficient D_{train})

Rule Post-Pruning

<u>Rule post-pruning is successful method for finding high accuracy hypotheses</u>

- Rule post-pruning involves the following steps:
- Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.
- Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.
- Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.
- Sort the pruned rules by their estimated accuracy, and consider them in this sequence whenclassifyingsubsequentinstances

<u>Converting a Decision Tree into Rules</u>



Boolean Decision Tree for Concept *PlayTennis*

Example

- IF (*Outlook* = *Sunny*) ∧ (*Humidity* = *High*) THEN *PlayTennis* = *No*
- IF (*Outlook* = *Sunny*) ∧ (*Humidity* = *Normal*) THEN *PlayTennis* = *Yes*
- …

For example, consider the decision tree. The leftmost path of the tree in below figure is translated into the rule.

IF (Outlook = Sunny) ^ (Humidity = High)
THEN PlayTennis = No

Given the above rule, rule post-pruning would consider removing the preconditions (Outlook = Sunny) and (Humidity = High)

- It would select whichever of these pruning steps produced the greatest improvement in estimated rule accuracy, then consider pruning the second precondition as a further pruning step.
- No pruning step is performed if it reduces the estimated rule accuracy.

There are three main advantages by converting the decision tree to rules before pruning

1. Converting to rules allows distinguishing among the different contexts in which a decision node is used. Because each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path.
2. Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves. Thus, it avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the sub tree below this test.
3. Converting to rules improves readability. Rules are often easier for to understand.

2. Incorporating Continuous-Valued Attributes

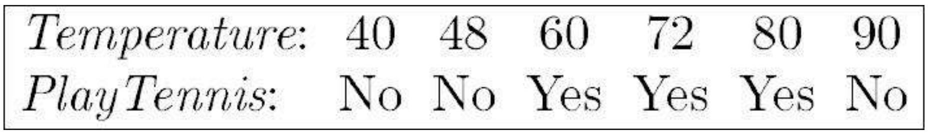
Continuous-valued decision attributes can be incorporated into the learned tree.

There are two methods for Handling Continuous Attributes

1. Define new discrete valued attributes that partition the continuous attribute value into a discrete set of intervals.
E.g., {high ≡ Temp > 35º C, med ≡ 10º C < Temp ≤ 35º C, low ≡ Temp ≤ 10º C}

2. Using thresholds for splitting nodes
e.g., A ≤ a produces subsets A ≤ a and A > a

What threshold-based Boolean attribute should be defined based on Temperature?



Temperature:	40	48	60	72	80	90
PlayTennis:	No	No	Yes	Yes	Yes	No

- Pick a threshold, c, that produces the greatest information gain
- In the current example, there are two candidate thresholds, corresponding to the values of Temperature at which the value of PlayTennis changes: (48 + 60)/2, and (80 + 90)/2.
- The information gain can then be computed for each of the candidate attributes, Temperature $_{>54}$, and Temperature $_{>85}$ and the best can be selected (Temperature $_{>54}$)

3. Alternative Measures for Selecting Attributes

- The problem is if attributes with many values, Gain will select it ?

- Example: consider the attribute Date, which has a very large number of possible values. (e.g., March 4, 1979).
- If this attribute is added to the Play Tennis data, it would have the highest information gain of any of the attributes .This is because Date alone perfectly predicts the target attribute over the training data. Thus, it would be selected as the decision attribute for the root node of the tree and lead to a tree of depth one, which perfectly classifies the training data.
- This decision tree with root node Date is not a useful predictor because it perfectly separates the training data, but poorly predict on subsequent examples.

One Approach: Use GainRatio instead of Gain

The gain ratio measure penalizes attributes by incorporating a split information, that is sensitive to how broadly and uniformly the attribute splits the data

$$GainRatio(S, A) \equiv \frac{Gain(S, A)}{SplitInformation(S, A)}$$

$$SplitInformation(S, A) \equiv - \sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Where, Si is subset of S, for which attribute A has value vi

4. Handling Training Examples with Missing Attribute Values

The data which is available may contain missing values for some attributes Example: Medical diagnosis

- <Fever = true, Blood-Pressure = normal, …, Blood-Test = ?, …>
- Sometimes values truly unknown, sometimes low priority (or cost too high)

Strategies for dealing with the missing attribute value

- If node n test A, assign most common value of A among other training examples sorted to node n
- Assign most common value of A among other training examples with same target value
- Assign a probability pi to each of the possible values vi of A rather than simply assigning the most common value to A(x)

5. Handling Attributes with Differing Costs

- In some learning tasks the instance attributes may have associated costs.
- For example: In learning to classify medical diseases, the patients described in terms of attributes such as Temperature, Biopsy Result, Pulse, BloodTestResults, etc.
- These attributes vary significantly in their costs, both in terms of monetary cost and cost to patient comfort
- Decision trees use low-cost attributes where possible, depends only on high-cost attributes only when needed to produce reliable classifications

How to Learn A Consistent Tree with Low Expected Cost?
One approach is replace Gain by Cost-Normalized-Gain
Examples of normalization functions

- Tan and Schlimmer

$$\frac{Gain^2(S, A)}{Cost(A)}.$$

- Nunez

$$\frac{2^{Gain(S,A)} - 1}{(Cost(A) + 1)^w}$$

where $w \in [0, 1]$ determines importance of cost

ARTIFICIAL NEURAL NETWORKS

INTRODUCTION

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued target functions.

Biological Motivation

- The study of artificial neural networks (ANNs) has been inspired by the observation that biological learning systems are built of very complex webs of interconnected ***Neurons***
- Human information processing system consists of brain ***neuron***: basic building block cell that communicates information to and from various parts of body

Facts of Human Neurobiology

- Number of neurons ~ 10^{11}
- Connection per neuron ~ 10^{4-5}
- Neuron switching time ~ 0.001 second or 10^{-3}
- Scenerecognitiontime~0.1second
- 100 inference steps doesn't seem like enough
- Highly parallel computation based on distributed representation

Properties of Neural Networks

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Input is a high-dimensional discrete or real-valued (e.g, sensor input)
-

NEURAL NETWORK REPRESENTATIONS

- A prototypical example of ANN learning is provided by Pomerleau's system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways
- The input to the neural network is a 30x32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- The network output is the direction in which the vehicle is steered

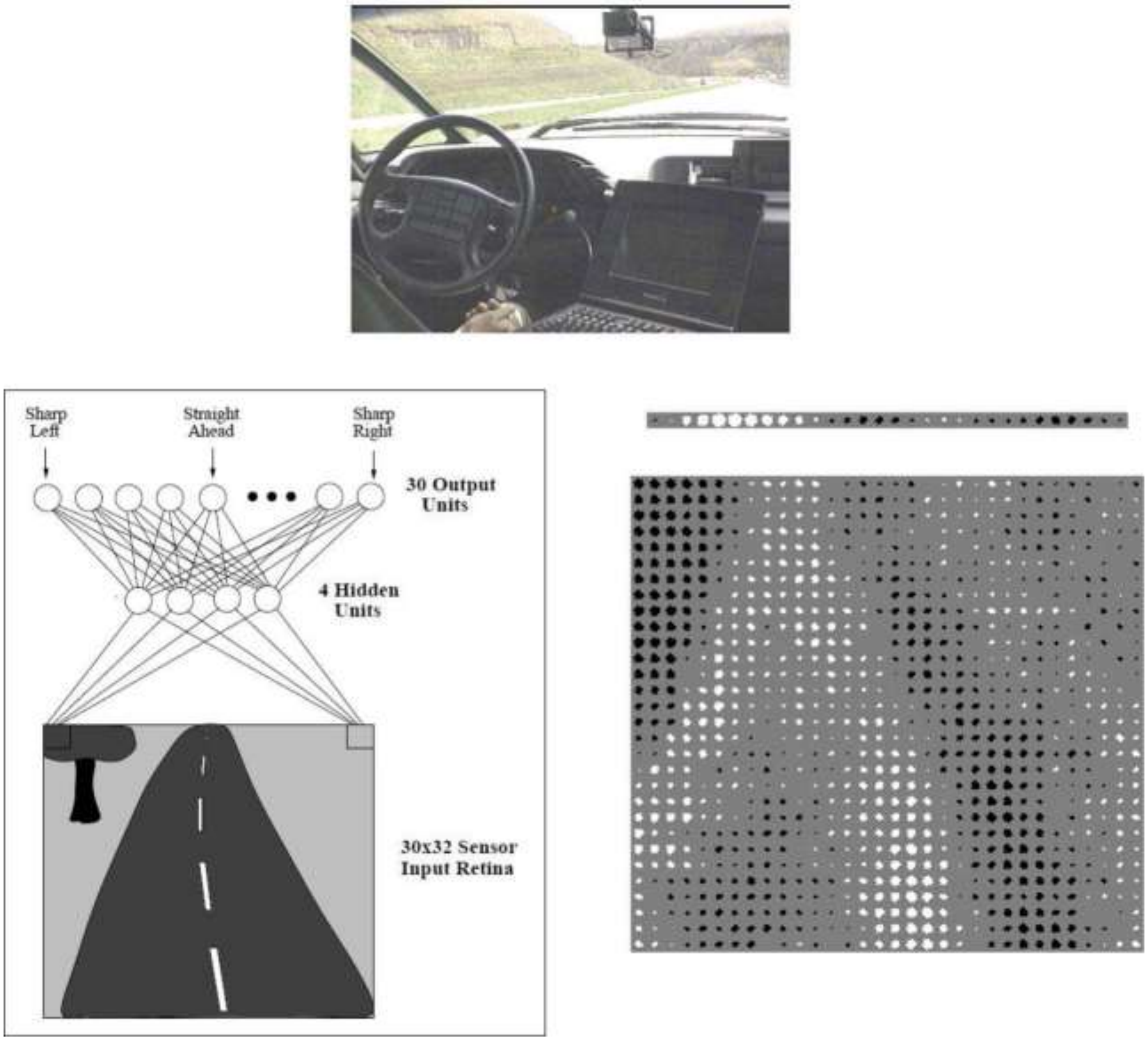


Figure: Neural network learning to steer an autonomous vehicle.

- Figure illustrates the neural network representation.
- The network is shown on the left side of the figure, with the input camera image depicted below it.
- Each node (i.e., circle) in the network diagram corresponds to the output of a single network unit, and the lines entering the node from below are its inputs.

- There are four units that receive inputs directly from all of the 30 x 32 pixels in the image. These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs
- These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.
- Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.
- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

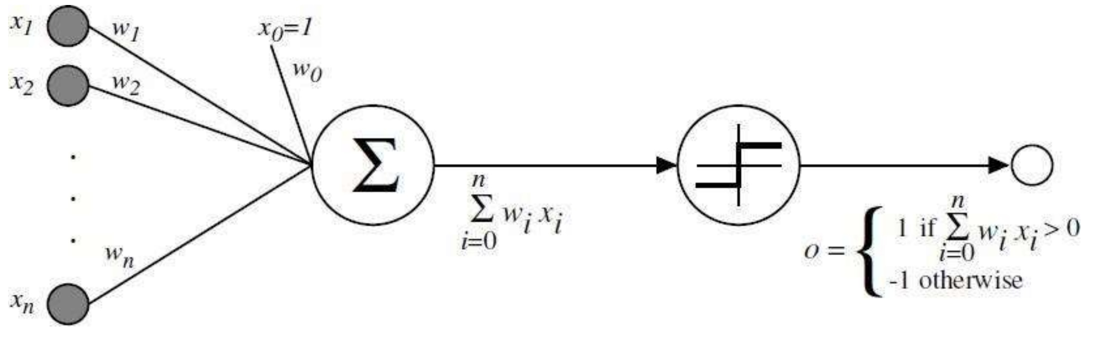
ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.

ANN is appropriate for problems with the following characteristics:

1. Instances are represented by many attribute-value pairs.
2. The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.
3. The training examples may contain errors.
4. Long training times are acceptable.
5. Fast evaluation of the learned target function may be required
6. The ability of humans to understand the learned target function is not important

PERCEPTRON

- One type of ANN system is based on a unit called a perceptron. Perceptron is a single layer neural network.



x_1
w_1
x_2
w_2
$x_0=1$
w_0
w_n
x_n
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure: A perceptron

- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.
- Given inputs ***x*** through ***x,*** the output **O(*x*₁, . . . , *x*ₙ*)*** computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

- Where, each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- $-w_0$ is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ must surpass in order for the perceptron to output a 1.

Sometimes, the perceptron function is written as,

$$O(\vec{x}) = \text{sgn}(\vec{w} \cdot \vec{x})$$

Where,

$$\text{sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

Representational Power of Perceptrons

- The perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points)
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in below figure

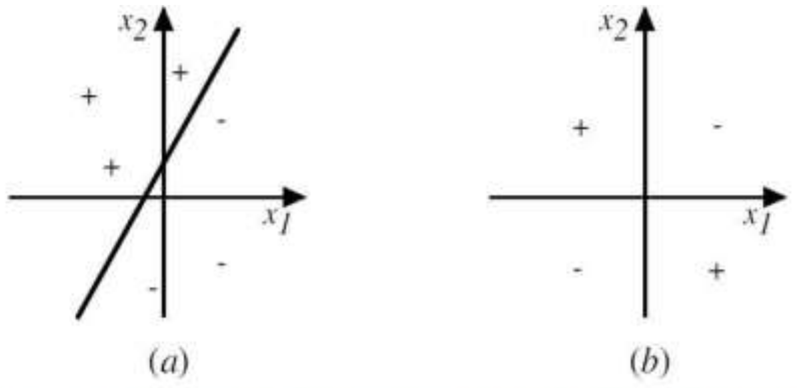


Figure : The decision surface represented by a two-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable.
x_1 and x_2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-".

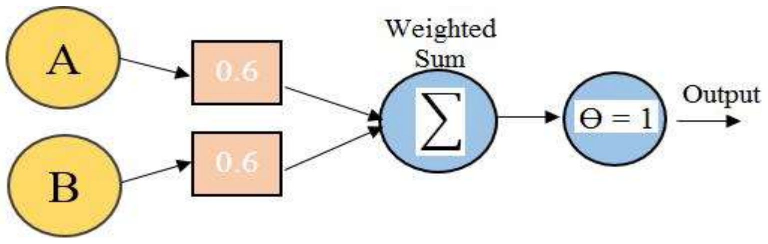
Perceptrons can represent all of the primitive Boolean functions AND, OR, NAND (~ AND),

and NOR(~OR)

Some Boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$

Example: Representation of AND functions

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1



If A=0 & B=0 → 0*0.6 + 0*0.6 = 0.

This is not greater than the threshold of 1, so the output = 0.

If A=0 & B=1 → 0*0.6 + 1*0.6 = 0.6.

This is not greater than the threshold, so the output = 0.

If A=1 & B=0 → 1*0.6 + 0*0.6 = 0.6.

This is not greater than the threshold, so the output = 0.

If A=1 & B=1 → 1*0.6 + 1*0.6 = 1.2.

This exceeds the threshold, so the output = 1.

Drawback of perceptron

- The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable

The Perceptron Training Rule

The learning problem is to determine a weight vector that causes the perceptron to produce the correct + 1 or - 1 output for each of the given training examples.

To learn an acceptable weight vector

- Begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input xi according to the rule.

$$w_i \leftarrow w_i + \Delta w_i$$

Where,

$$\Delta w_i = \eta(t - o)x_i$$

Here,
t is the target output for the current training example
o is the output generated by the perceptron
η is a positive constant called the ***learning rate***

- The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases

Drawback:
The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.

Gradient Descent and the Delta Rule

- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

To understand the delta training rule, consider the task of training an unthresholded perceptron. That is, a linear unit for which the output ***O*** is given by

$$o = w_0 + w_1x_1 + \cdots + w_nx_n$$

$$O(\vec{x}) = (\vec{w} . \vec{x}) \qquad \text{equ. (1)}$$

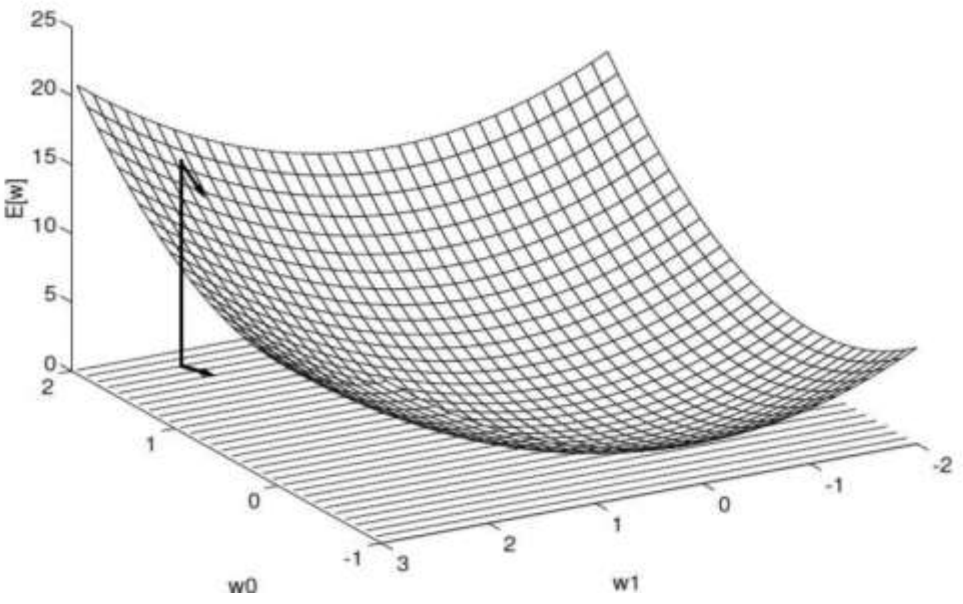
To derive a weight learning rule for linear units, specify a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples.

$$E[\vec{w}] \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \qquad \text{equ. (2)}$$

- Is the set of training examples, td is the target output for training example d,
- od is the output of the linear unit for training example d
- E ($\vec{w}$) is simply half the squared difference between the target output td and the linear unit output od, summed over all training examples. Where,•D

Visualizing the Hypothesis Space

- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values as shown in below figure.
- Here the axes w_0 and w_1 represent possible values for the two weights of a simple linear unit. The w_0, w_1 plane therefore represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.
- The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.
- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space



- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.
At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in above figure. This process continues until the global
Minimum error is reached.

Derivation of the Gradient Descent Rule

How to calculate the direction of steepest descent along the error surface?

The direction of steepest can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$. This vector derivative is called the gradient of E with respect to $\vec{w}$, written as

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right] \qquad \text{equ. (3)}$$

The gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

Where,

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \qquad \text{equ. (4)}$$

- Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search.
- The negative sign is present because we want to move the weight vector in the direction that decreases E.

This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

Where,

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \qquad \text{equ. (5)}$$

Calculate the gradient at each step. The vector of **derivatives that form the**

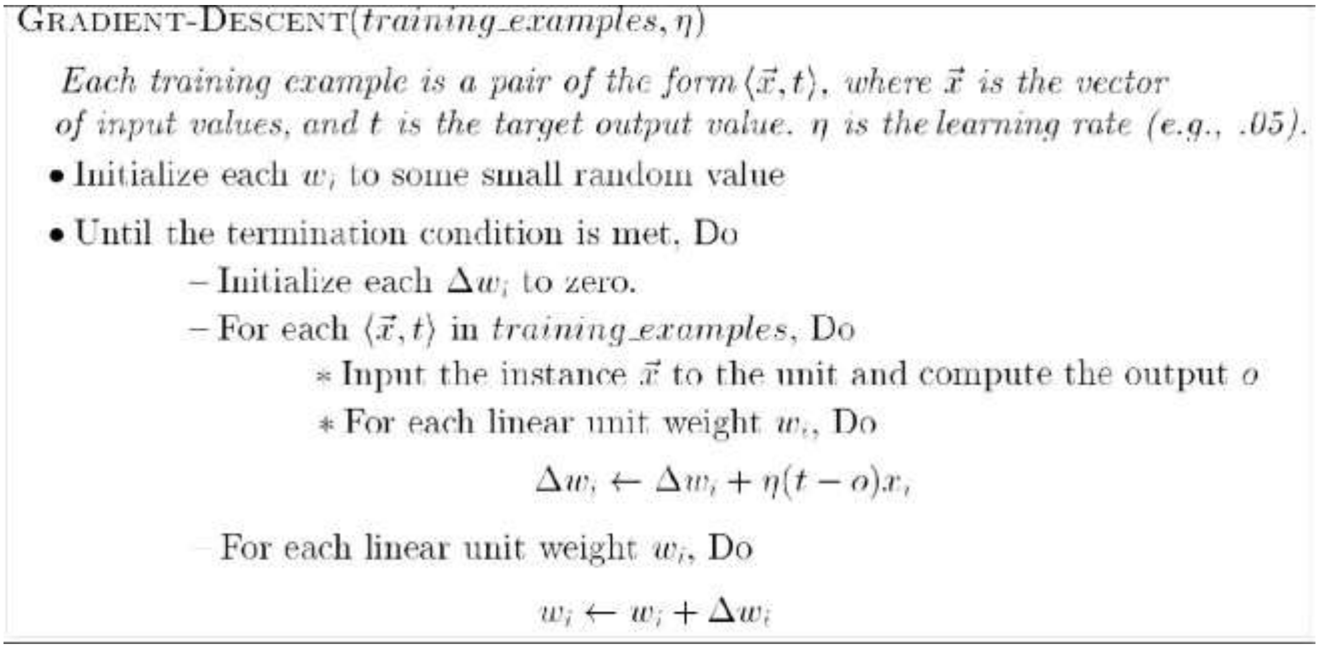
gradient can be obtained by differentiating E from Equation (2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \qquad \text{equ. (6)}
\end{aligned}$$

Substituting Equation (6) into Equation (5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta\sum_{d\in D}(t_d - o_d)\, x_{id} \qquad \text{equ. (7)}$$

GRADIENT DESCENT algorithm for training a linear unit



```
GRADIENT-DESCENT(training_examples, $\eta$)

  Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector
  of input values, and $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
  • Initialize each $w_i$ to some small random value
  • Until the termination condition is met, Do
        – Initialize each $\Delta w_i$ to zero.
        – For each $\langle \vec{x}, t \rangle$ in training_examples, Do
              * Input the instance $\vec{x}$ to the unit and compute the output $o$
              * For each linear unit weight $w_i$, Do
                    $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
        – For each linear unit weight $w_i$, Do
                    $w_i \leftarrow w_i + \Delta w_i$
```

To summarize, the gradient descent algorithm for training linear units is as follows:

- Pick an initial random weight vector.
- Apply the linear unit to all training examples ,then compute Δwi for each weight according to Equation (7).
- Update each weight w_i by adding Δw_i, then repeat this process

Issues in Gradient Descent Algorithm

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

1. The hypothesis space contains continuously parameterized hypotheses
2. The error can be differentiated with respect to these hypothesis parameters

The key practical difficulties in applying gradient descent are

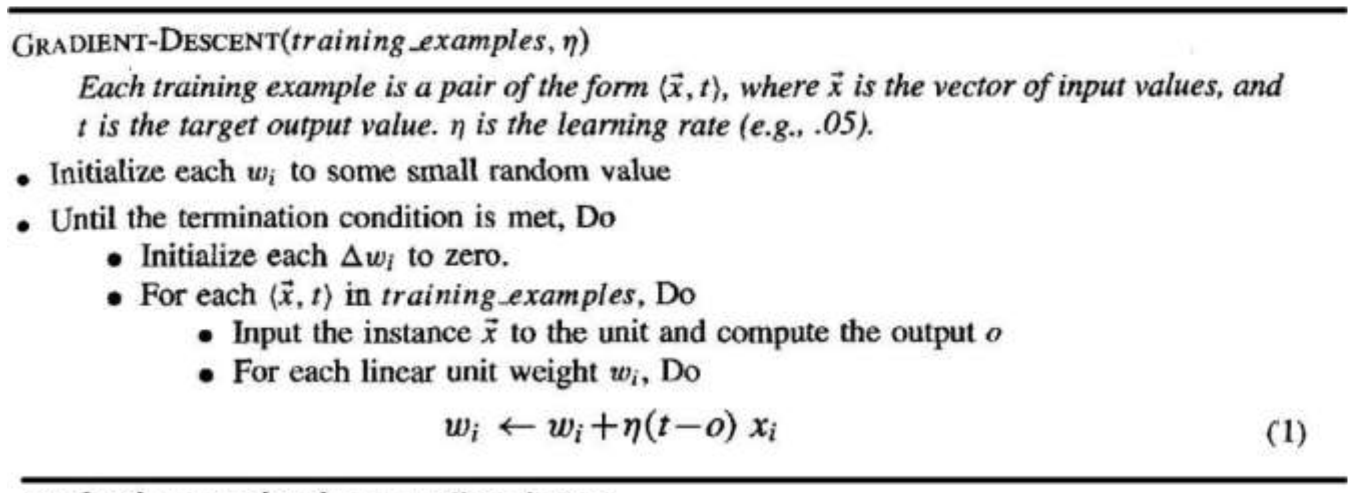
1. Converging to a local minimum can sometimes be quite slow
2. If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

Stochastic Approximation to Gradient Descent

- The gradient descent training rule presented in Equation (7) computes weight updates after summing over all the training examples in D
- The idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example

$$\Delta w_i = \eta\,(t - o)\,x_i$$

- where t, o, and x_i are the target value, unit output, and ith input for the training example in question



```
GRADIENT-DESCENT(training_examples, $\eta$)
    Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and
    $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
• Initialize each $w_i$ to some small random value
• Until the termination condition is met, Do
    • Initialize each $\Delta w_i$ to zero.
    • For each $\langle \vec{x}, t \rangle$ in training_examples, Do
        • Input the instance $\vec{x}$ to the unit and compute the output $o$
        • For each linear unit weight $w_i$, Do
                $w_i \leftarrow w_i + \eta(t - o)\, x_i$                    (1)
```

stochastic approximation to gradient descent

One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

- Where, t_d and o_d are the target value and the unit output value for training example d.
- Stochastic gradient descent iterates over the training examples d in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$
- The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E_d(\vec{w})$
- By making the value of η sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely

The key differences between standard gradient descent and stochastic gradient descent are

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true

gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.

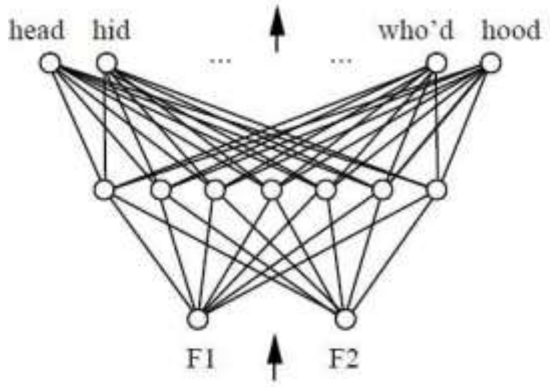
- In cases where there are multiple local minima with respect to stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Multilayer networks learned by the BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

Consider the example:

- Here the speech recognition task involves distinguishing among10possiblevowels, all spoken in the context of "h d" (i.e., "hid," "had," "head," "hood," etc.).
- The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest.
- The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network.



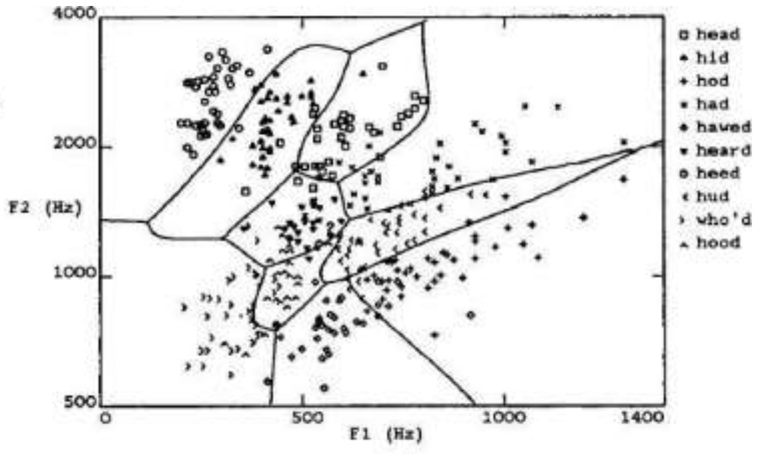
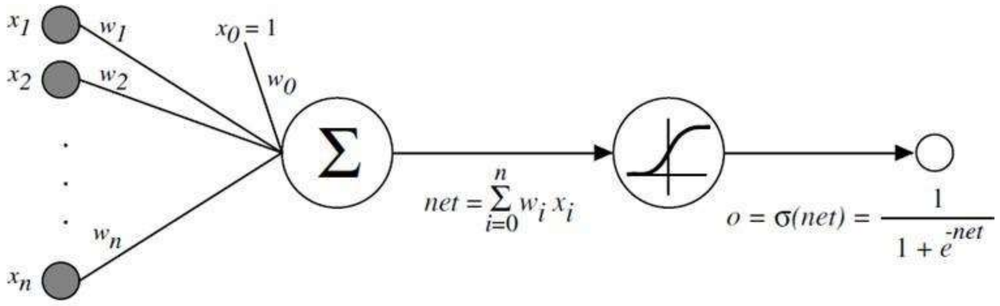


Figure: Decision regions of a multilayer feedforward network.

A Differentiable Threshold Unit (Sigmoid unit)

- Sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.



x_1
w_1
x_2
w_2
$x_0 = 1$
w_0
Σ
w_n
x_n
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

Figure: A Sigmoid Threshold Unit

- The sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result and the threshold output is a continuous function of its input.
- More precisely, the sigmoid unit computes its output O as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

Where,

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

σ is the sigmoid function

The BACKPROPAGATION Algorithm

- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- In BACKPROPAGATION algorithm, we consider networks with multiple output units rather than single units as before, so we redefine E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{........equ. (1)}$$

where,

- ***outputs*** - is the set of output units in the network
- t_{kd} and O_{kd} - the target and output values associated with the ***k****th* output unit
- ***d*** - training example

Algorithm:

BACKPROPAGATION (*training_example, η, n_{in}, n_{out}, n_{hidden}*)

Each training example is a pair of the form (→ ,), where ( ) is the vector of network input values, ( ) and is the vector of target network output values.

η is the learning rate (e.g., .05). n_i, is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.
The input from unit i into unit j is denoted x_{ji} , and the weight from unit i to unit j is denoted w_{ji}

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.

- Initialize all network weights to small random numbers

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

3. For each hidden unit *h*, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = \eta \delta_j x_{i,j}$$

- Until the termination condition is met, Do
  - For each ($\vec{}$,), in training examples, Do *Propagate the input forward through the network:*
  1. Input the instance $\vec{}$, to the network and compute the output o_u of every unit u in the network.
  2.

  Propagate the errors backward through the network:

Adding Momentum

Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. The most common is to alter the weight-update rule the equation below

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

by making the weight update on the nth iteration depend partially on the update that occurred during the (n - 1)th iteration, as follows:

$$\Delta w_{ji}(n) = \eta\, \delta_j\, x_{ji} + \alpha \Delta w_{ji}(n-1)$$

Learning in arbitrary acyclic networks

- BACKPROPAGATION algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule is retained, and the only change is to the procedure for computing δ values.
- In general, the ***δ***, value for a unit ***r*** in layer ***m*** is computed from the ***δ*** values at the next deeper layer m + 1 according to

$$\delta_r = o_r\,(1-o_r) \sum_{s \in layer\, m+1} w_{sr}\, \delta_s$$

- The rule for calculating ***δ*** for any internal unit

$$\delta_r = o_r (1 - o_r) \sum_{s \in Downstream(r)} w_{sr} \delta_s$$

Where, Downstream(r) is the set of units immediately downstream from unit ***r*** in the network: that is, all units whose inputs include the output of unit ***r***

Derivation of the BACKPROPAGATION Rule

- Deriving the stochastic gradient descent rule: Stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example
- For each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad \text{.........equ. (1)}$$

where, E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in output} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, t_k is the target value of unit ***k*** for training example ***d***, and o_k is the output of unit ***k*** given training example ***d***.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \Sigma_i w_{ji}x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- outputs = the set of units in the final layer of the network
- Downstream(j)=these to f units whose immediate inputs include the output of unit j

derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule

seen in Equation $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

notice that weight w_{ji} can influence the rest of the network only through net_j.

Use chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$
$$= \frac{\partial E_d}{\partial net_j} x_{ji} \qquad \text{......equ(2)}$$

Derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

<u>Consider two cases:</u> The case where unit j is an output unit for the network, and the case where j is an internal unit (hidden unit).

Case 1: Training Rule for Output Unit Weights.

w_{ji} can influence the rest of the network only through net_j , net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \qquad \text{.....equ(3)}$$

To begin, consider just the first term in Equation (3)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. We therefore drop the summation over output units and simply set $k = j$.

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2}(t_j - o_j)^2$$
$$= \frac{1}{2} 2(t_j - o_j) \frac{\partial (t_j - o_j)}{\partial o_j}$$
$$= -(t_j - o_j) \qquad \text{......equ (4)}$$

Next consider the second term in Equation (3). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1-\sigma(net_j))$. Therefore,

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

$$= o_j(1-o_j) \qquad \text{.......equ(5)}$$

Substituting expressions (4) and (5) into (3), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1-o_j) \qquad \text{.......equ(6)}$$

and combining this with Equations (1) and (2), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1-o_j)x_{ji} \qquad \text{.......equ(7)}$$

Case 2: Training Rule for Hidden Unit Weights.

- In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d.
- For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network and denoted this set of units by Downstream(j).
- net_j can influence the network outputs only through the units in Downstream(j). Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj}\ o_j(1-o_j) \qquad \text{............equ (8)}
\end{aligned}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1-o_j) \sum_{k \in Downstream(j)} \delta_k\ w_{kj}$$

and

$$\Delta w_{ji} = \eta\ \delta_j\ x_{ji}$$

REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

- The BACKPROPAGATION multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.
- Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice.
- Local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases.

Common heuristics to attempt to alleviate the problem of local minima include:

1. Add a momentum term to the weight-update rule. Momentum can sometimes carry the gradient descent procedure through narrow local minima
2. Use stochastic gradient descent rather than true gradient descent
3. Train multiple networks using the same data, but initializing each network with different random weights

2. Representational Power of Feedforward Networks

*What setoffunctionscanberepresentedbyfeed-forwardnetworks?*The answer depends on the width and depth of the networks. There are three quite general
results are known about which function classes can be described by which types of Networks

1. Boolean functions – Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs

2. Continuous functions – Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units

3. Arbitrary functions – Any function can be approximated to arbitrary accuracy by a network with three layers of units.

3. Hypothesis Space Search and Inductive Bias

- Hypothesis space is the n-dimensional Euclidean space of the ***n*** network weights and hypothesis space is continuous.

- As it is continuous, E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis.
- It is difficult to characterize precisely the inductive bias of BACKPROPAGATION algorithm, because it depends on the interplay between the gradient descent search and

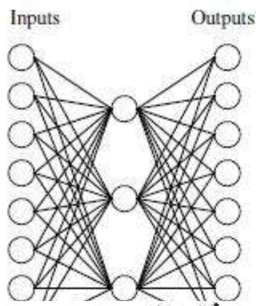
the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as smooth interpolation between data points.

4. Hidden Layer Representations

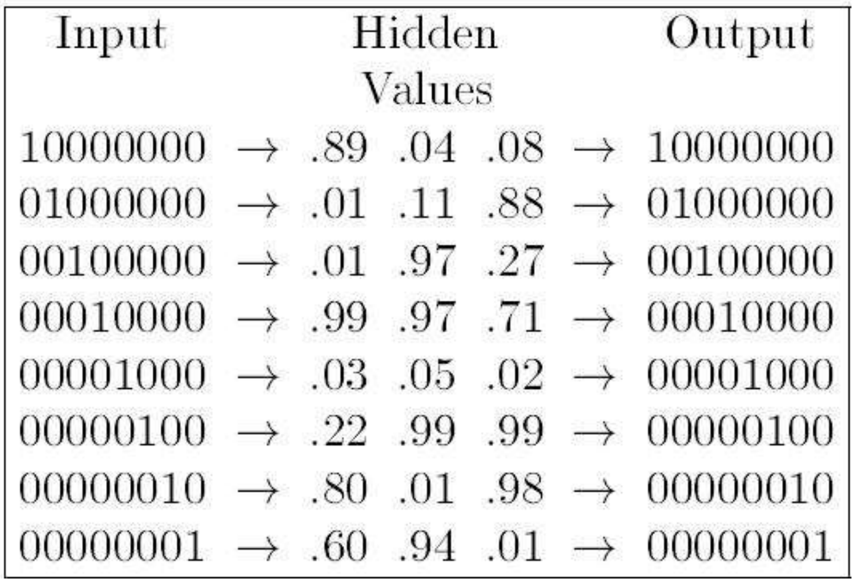
BACKPROPAGATION can define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider example, the network shown in below Figure

A network:



Learned hidden layer representation:



Input		Hidden Values				Output
10000000	$\rightarrow$	.89	.04	.08	$\rightarrow$	10000000
01000000	$\rightarrow$	.01	.11	.88	$\rightarrow$	01000000
00100000	$\rightarrow$	.01	.97	.27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99	.97	.71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03	.05	.02	$\rightarrow$	00001000
00000100	$\rightarrow$	.22	.99	.99	$\rightarrow$	00000100
00000010	$\rightarrow$	.80	.01	.98	$\rightarrow$	00000010
00000001	$\rightarrow$	.60	.94	.01	$\rightarrow$	00000001

- Consider training the network shown in Figure to learn the simple target function f (x) = x, where x is a vector containing seven 0's and a single 1.
- The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.
- When BACKPROPAGATION applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar

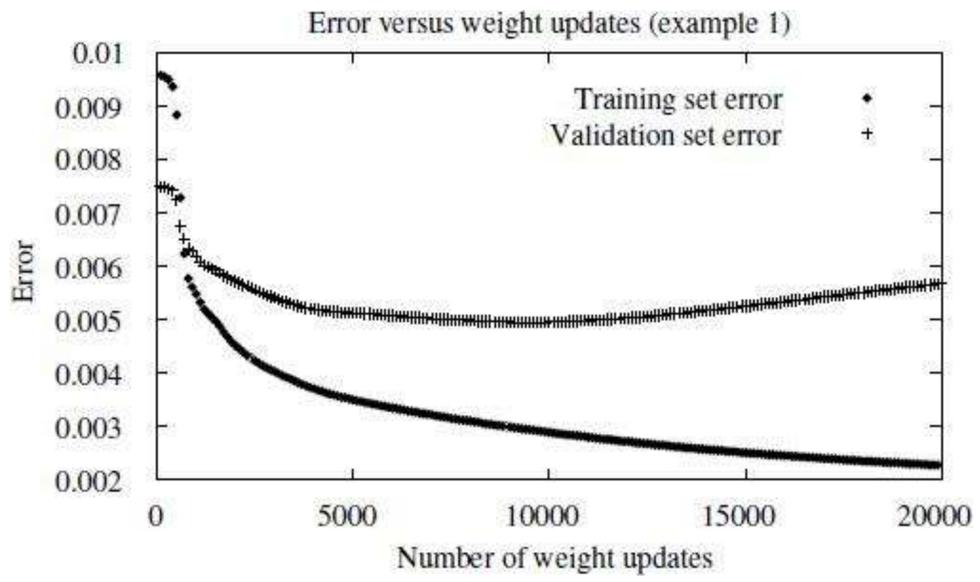
standard binary encoding of eight values using three bits (e.g., 000,001,010,. . . , 111). The exact values of the hidden units for one typical run of shown in Figure.

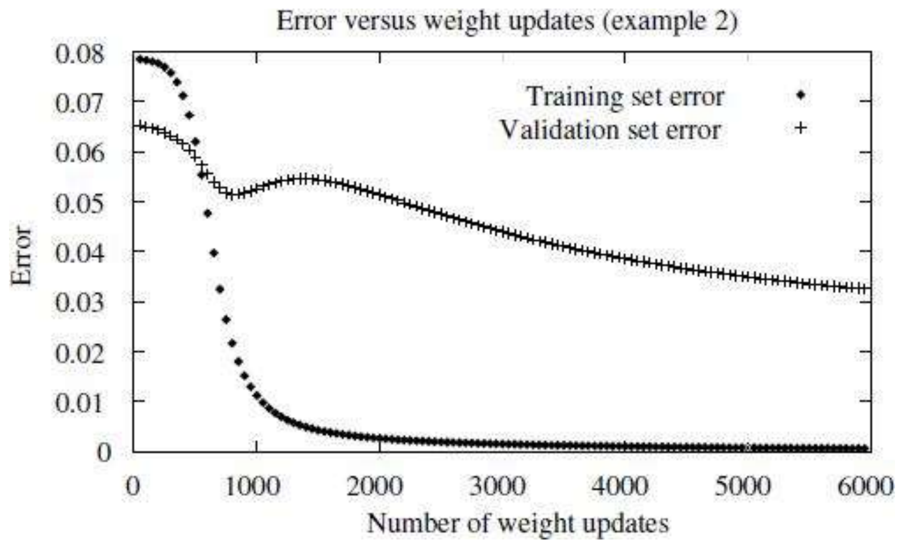
- This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning

5. Generalization, Overfitting, and Stopping Criterion

What is an appropriate condition for terminating the weight update loop? One choice is to continue training until the error E on the training examples falls below some predetermined threshold.

To see the dangers of minimizing the error over the training data, consider how the error E varies with the number of weight iterations





- Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples. This line measures the generalization accuracy of the network-the accuracy with which it fits examples beyond the training data.
- The generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies

- Why does overfitting tend to occur during later iterations, but not during earlier iterations?
  By giving enough weight-tuning iteration, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.

MODULE 4
BAYESIAN LEARNING

Bayesian reasoning provides a probabilistic approach to inference. It is based on the assumption that the quantities of interest are governed by probability distributions and that optimal decisions can be made by reasoning about these probabilities together with observed data

INTRODUCTION

Bayesian learning methods are relevant to study of machine learning for two different reasons.

1. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems
2. The second reason is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

Features of Bayesian Learning Methods

- Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct. This provides a more flexible approach to learning than algorithms that completely eliminate a hypothesis if it is found to be inconsistent with any single example
- Prior knowledge can be combined with observed data to determine the final probability of a hypothesis. In Bayesian learning, prior knowledge is provided by asserting (1) a prior probability for each candidate hypothesis, and (2) a probability distribution over observed data for each possible hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions
- New instances can be classified by combining the predictions of multiple hypotheses, weighted by their probabilities.
- Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

Practical difficulty in applying Bayesian methods

1. One practical difficulty in applying Bayesian methods is that they typically require initial knowledge of many probabilities. When these probabilities are not known in advance they are often estimated based on background knowledge, previously available data, and assumptions about the form of the underlying distributions.
2. A second practical difficulty is the significant computational cost required to determine the Bayes optimal hypothesis in the general case. In certain specialized situations, this computational cost can be significantly reduced.

BAYES THEOREM

Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

Notations

- P(h) prior probability of h, reflects any background knowledge about the chance that h is correct
- P(D)prior probability of D,probability that D will be observed
- P(D|h) probability of observing D given a world in which h holds
- P(h|D) posterior probability of h, reflects confidence that h holds after D has been observed

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability P(h|D), from the prior probability P(h), together with P(D) and P(D|h).

Bayes Theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- P(h|D) increases with P(h) and with P(D|h) according to Bayes theorem.
- P(h|D) decreases as P(D) increases, because the more probable it is that D will be observed independent of h, the less evidence D provides in support of h.

Maximum a Posteriori (MAP) Hypothesis

- In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis h ∈ H given the observed data D. Any such maximally probable hypothesis is called a maximum a posteriori (MAP) hypothesis.
- Bayes theorem to calculate the posterior probability of each candidate hypothesis is h_{MAP} is a MAP hypothesis provided

$$\begin{aligned} h_{MAP} &= \underset{h \in H}{argmax}\ P(h|D) \\ &= \underset{h \in H}{argmax}\ \frac{P(D|h)P(h)}{P(D)} \\ &= \underset{h \in H}{argmax}\ P(D|h)P(h) \end{aligned}$$

- P(D) can be dropped, because it is a constant independent of h

Maximum Likelihood (ML) Hypothesis

- In some cases, it is assumed that every hypothesis in H is equally probable apriori (P(hi) = P(hj) for all hi and hj in H).
- In this case the below equation can be simplified and need only consider the term P(D|h) to find the most probable hypothesis.

$$h_{MAP} = \underset{h \in H}{argmax}\ P(D|h)P(h)$$

the equation can be simplified

$$h_{ML} = \underset{h \in H}{argmax}\ P(D|h)$$

P(D|h) is often called the likelihood of the data D given h, and any hypothesis that maximizes P(D|h) is called a maximum likelihood (ML) hypothesis

Example

- Consider a medical diagnosis problem in which there are two alternative hypotheses: (1) that the patient has particular form of cancer, and (2) that the patient does not. The available data is from a particular laboratory test with two possible outcomes: + (positive) and - (negative).
- We have prior knowledge that over the entire population of people only .008 have this disease. Furthermore, the lab test is only an imperfect indicator of the disease.
- The test returns a correct positive result in only 98% of the cases in which the disease is actually present and a correct negative result in only 97% of the cases in which the disease is not present. In other cases, the test returns the opposite result.
- The above situation can be summarized by the following probabilities:

$$P(cancer) = .008 \qquad P(\neg cancer) = 0.992$$
$$P(\oplus|cancer) = .98 \qquad P(\ominus|cancer) = .02$$
$$P(\oplus|\neg cancer) = .03 \qquad P(\ominus|\neg cancer) = .97$$

Suppose a new patient is observed for whom the lab test returns a positive (+) result. Should we diagnose the patient as having cancer or not?

$$P(\oplus|cancer)P(cancer) = (.98).008 = .0078$$
$$P(\oplus|\neg cancer)P(\neg cancer) = (.03).992 = .0298$$
$$\Rightarrow h_{MAP} = \neg cancer$$

The exact posterior probabilities can also be determined by normalizing the above quantities so that they sum to 1

$$P(cancer|\oplus) = \frac{0.0078}{0.0078 + 0.0298} = 0.21$$

$$P(\neg cancer|\oplus) = \frac{0.0298}{0.0078 + 0.0298} = 0.79$$

Basic formulas for calculating probabilities are summarized in Table

- ***Product rule***: probability $P(A \wedge B)$ of a conjunction of two events A and B

$$P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$$

- ***Sum rule***: probability of a disjunction of two events A and B

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

- ***Bayes theorem***: the posterior probability $P(h|D)$ of h given D

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- ***Theorem of total probability***: if events $A_1, \ldots, A_n$ are mutually exclusive with $\sum_{i=1}^{n} P(A_i) = 1$, then

$$P(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i)$$

BAYES THEOREM AND CONCEPT LEARNING

What is the relationship between Bayes theorem and the problem of concept learning?

Since Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, and can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most probable.

Brute-Force Bayes Concept Learning

Consider the concept learning problem

- Assume the learner considers some finite hypothesis space H defined over the instance space X, in which the task is to learn some target concept c : X → {0,1}.
- Learner is given some sequence of training examples ((x_1, d_1) . . . (x_m, d_m)) where x_i is some instance from X and where d_i is the target value of x_i (i.e., d_i = c(x_i)).
- The sequence of target values are written as D = (d_1 . . . d_m).

We can design a straightforward concept learning algorithm to output the maximum a posteriori hypothesis, based on Bayes theorem, as follows:

<u>BRUTE</u>-FORCE MAP LEARNING algorithm:

1. For each hypothesis h in H, calculate the posterior probability

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

2. Output the hypothesis h_{MAP} with the highest posterior probability

$$h_{MAP} = \underset{h \in H}{argmax}\ P(h|D)$$

In order specify a learning problem for the BRUTE-FORCE MAP LEARNING algorithm we must specify what values are to be used for P(h) and for P(D|h) ?

Let’s choose P(h) and for P(D|h) to be consistent with the following assumptions:

- The training data D is noise free (i.e., d_i = c(x_i))
- The target concept c is contained in the hypothesis space H
- Do not have a priori reason to believe that any hypothesis is more probable than any other.

What values should we specify for P(h)?

- Given no prior knowledge that one hypothesis is more likely than another, it is reasonable to assign the same prior probability to every hypothesis h in H.
- Assume the target concept is contained in H and require that these prior probabilities sum to 1.

$$P(h) = \frac{1}{|H|} \text{ for all } h \in H$$

What choice shall we make for P(D|h)?

- P(D|h) is the probability of observing the target values D = (d_1 . . .d_m) for the fixed set of instances (x_1 . . . x_m), given a world in which hypothesis h holds

- Since we assume noise-free training data, the probability of observing classification d_i given h is just 1 if $d_i = h(x_i)$ and 0 if $d_i \neq h(x_i)$. Therefore,

$$P(D|h) = \begin{cases} 1 & \text{if } d_i = h(x_i) \text{ for all } d_i \in D \\ 0 & \text{otherwise} \end{cases}$$

Given these choices for P(h) and for P(D|h) we now have a fully-defined problem for the above BRUTE-FORCE MAP LEARNING algorithm.

Recalling Bayes theorem, we have

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Consider the case where h is inconsistent with the training data D

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0$$

The posterior probability of a hypothesis inconsistent with D is

zero *Consider the case where h is consistent with D*

$$P(h|D) = \frac{1 \cdot \frac{1}{|H|}}{P(D)} = \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}} = \frac{1}{|VS_{H,D}|}$$

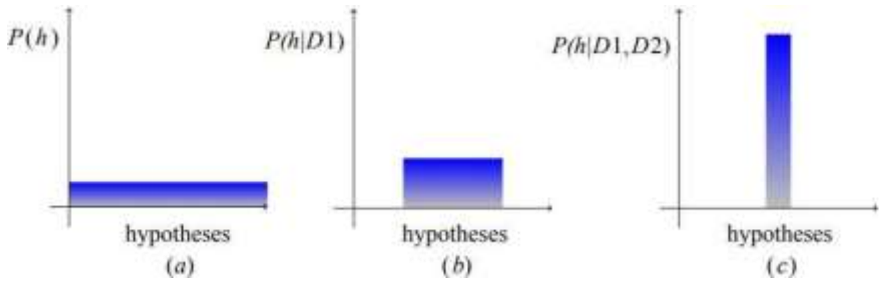
Where, $VS_{H,D}$ is the subset of hypotheses from H that are consistent with D

To summarize, Bayes theorem implies that the posterior probability P(h|D) under our assumed P(h) and P(D|h) is

$$P(D|h) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{if h is consistent with D} \\ 0 & \text{otherwise} \end{cases}$$

The Evolution of Probabilities Associated with Hypotheses

- Figure (a) all hypotheses have the same probability.
- Figures (b) and (c), As training data accumulates, the posterior probability for inconsistent hypotheses becomes zero while the total probability summing to 1 is shared equally among the remaining consistent hypotheses.



MAP Hypotheses and Consistent Learners

- A learning algorithm is a consistent learner if it outputs a hypothesis that commits zero errors over the training examples.
- Every consistent learner outputs a MAP hypothesis, if we assume a uniform prior probability distribution over H ($P(h_i) = P(h_j)$ for all i, j), and deterministic, noise free training data (P(D|h) =1 if D and h are consistent, and 0 otherwise).

Example:

- FIND-S outputs a consistent hypothesis, it will output a MAP hypothesis under the probability distributions P(h) and P(D|h) defined above.
- Are there other probability distributions for P(h) and P(D|h) under which FIND-S outputs MAP hypotheses? Yes.
- Because FIND-S outputs a maximally specific hypothesis from the version space, its output hypothesis will be a MAP hypothesis relative to any prior probability distribution that favours more specific hypotheses.

Note

- Bayesian framework is a way to characterize the behaviour of learning algorithms
- By identifying probability distributions P(h) and P(D|h) under which the output is a optimal hypothesis, implicit assumptions of the algorithm can be characterized (Inductive Bias)
- Inductive inference is modelled by an equivalent probabilistic reasoning system based on Bayes theorem

MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

Consider the problem of learning a *continuous- valued target function* such as neural network learning, linear regression, and polynomial curve fitting

A straightforward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a *maximum likelihood (ML) hypothesis*

- Learner L considers an instance space X and a hypothesis space H consisting of some class of real-valued functions defined over X, i.e., $(\forall\ h \in H)[\ h : X \rightarrow R]$ and training examples of the form <xi,di>
- The problem faced by L is to learn an unknown target function $f : X \rightarrow R$
- A set of m training examples is provided, where the target value of each example is corrupted by random noise drawn according to a Normal probability distribution with zero mean ($d_i = f(x_i) + e_i$)
- Each training example is a pair of the form (x_i, d_i) where $d_i = f(x_i) + e_i$.
  - Here $f(x_i)$ is the noise-free value of the target function and e_i is a random variable representing the noise.
  - It is assumed that the values of the e_i are drawn independently and that they are distributed according to a Normal distribution with zero mean.
- The task of the learner is to *output a maximum likelihood hypothesis* or a *MAP hypothesis assuming all hypotheses are equally probable a priori*.

Using the definition of h_{ML} we have

$$h_{ML} = \underset{h \in H}{argmax}\ p(D|h)$$

Assuming training examples are mutually independent given h, we can write P(D|h) as the product of the various (d_i|h)

$$h_{ML} = \underset{h \in H}{argmax} \prod_{i=1}^{m} p(d_i|h)$$

Given the noise e_i obeys a Normal distribution with zero mean and unknown variance σ^2 , each d_i must also obey a Normal distribution around the true targetvalue $f(x_i)$. Because we are writing the expression for P(D|h), we assume h is the correct description of f.

Hence, $\mu = f(x_i) = h(x_i)$

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - \mu)^2}$$

$$h_{ML} = \underset{h \in H}{argmax} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - h(x_i))^2}$$

Maximize the less complicated logarithm, which is justified because of the monotonicity of function p

$$h_{ML} = \underset{h \in H}{argmax} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

The first term in this expression is a constant independent of h, and can therefore be discarded, yielding

$$h_{ML} = \underset{h \in H}{argmax} \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Maximizing this negative quantity is equivalent to minimizing the corresponding positive quantity

$$h_{ML} = \underset{h \in H}{argmin} \sum_{i=1}^{m} \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Finally,discardconstantsthatareindependentofh.

$$h_{ML} = \underset{h \in H}{argmin} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

Thus, above equation shows that the maximum likelihood hypothesis h_{ML} is the one that minimizes the sum of the squared errors between the observed training values d_i and the hypothesis predictions $h(x_i)$

Note:

Why is it reasonable to choose the Normal distribution to characterize noise?

- Good approximation of many types of noise in physical systems
- Central Limit Theorem shows that the sum of a sufficiently large number of independent, identically distributed random variables itself obeys a Normal

distribution Only noise in the target value is considered, not in the attributes describing the instances themselves

MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES

- Consider the setting in which we wish to learn a nondeterministic (probabilistic) function f : X → {0, 1}, which has two discrete output values.
- We want a function approximator whose output is the probability that f(x) = 1. In other words, learn the target function f ` : X → [0, 1] such that f ` (x) = P(f(x) = 1)

How can we learn f ` using a neural network?

- Use of brute force way would be to first collect the observed frequencies of 1's and 0's for each possible value of x and to then train the neural network to output the target frequency for each x.

What criterion should we optimize in order to find a maximum likelihood hypothesis for f' in this setting?

- First obtain an expression for P(D|h)
- Assume the training data D is of the form D = {(x_1, d_1) . . . (x_m, d_m)}, where d_i is the observed 0 or 1 value for f (x_i).
- Both x_i and d_i as random variables, and assuming that each training example is drawn independently, we can write P(D|h) as

Applyingtheproductrule

$$P(D \mid h) = \prod_{i=1}^{m} P(x_i, d_i \mid h) \qquad \text{equ (1)}$$

$$P(D \mid h) = \prod_{i=1}^{m} P(d_i \mid h, x_i) P(x_i) \qquad \text{equ (2)}$$

The probability P(d_i|h, x_i)

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases} \qquad \text{equ (3)}$$

Re-express it in a more mathematically manipulable form, as

$$P(d_i|h, x_i) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \qquad \text{equ (4)}$$

Equation (4) to substitute for P(di |h, xi) in Equation (5) to obtain

$$P(D|h) = \prod_{i=1}^{m} h(x_i)^{d_i}(1-h(x_i))^{1-d_i} P(x_i) \qquad \text{equ (5)}$$

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1-h(x_i))^{1-d_i} P(x_i)$$

We write an expression for the maximum likelihood hypothesis
The last term is a constant independent of h, so it can be dropped

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1-h(x_i))^{1-d_i} \qquad \text{equ (6)}$$

It easier to work with the log of the likelihood, yielding

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} d_i \ln h(x_i) + (1-d_i)\ln(1-h(x_i)) \qquad \text{equ (7)}$$

Equation (7) describes the quantity that must be maximized in order to obtain the maximum likelihood hypothesis in our current problem setting

Gradient Search to Maximize Likelihood in a Neural Net

- Derive a weight-training rule for neural network learning that seeks to maximize G(h,D) using gradient ascent
- The gradient of G(h,D) is given by the vector of partial derivatives of G(h,D) with respect to the various network weights that define the hypothesis h represented by the learned network
- In this case, the partial derivative of G(h, D) with respect to weight w_{jk} from input k to unit j is

$$\begin{aligned}\frac{\partial G(h,D)}{\partial w_{jk}} &= \sum_{i=1}^{m} \frac{\partial G(h,D)}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^{m} \frac{\partial(d_i \ln h(x_i) + (1-d_i)\ln(1-h(x_i)))}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^{m} \frac{d_i - h(x_i)}{h(x_i)(1-h(x_i))} \frac{\partial h(x_i)}{\partial w_{jk}} \qquad \text{equ (1)}\end{aligned}$$

- Suppose our neural network is constructed from a single layer of sigmoid units. Then,

$$\frac{\partial h(x_i)}{\partial w_{jk}} = \sigma'(x_i)x_{ijk} = h(x_i)(1-h(x_i))x_{ijk}$$

  where x_{ijk} is the kth input to unit j for the i th training example, and d(x) is the derivative of the sigmoid squashing function.
- Finally, substituting this expression into Equation (1), we obtain a simple expression for the derivatives that constitute the gradient

$$\frac{\partial G(h,D)}{\partial w_{jk}} = \sum_{i=1}^{m} (d_i - h(x_i))\, x_{ijk}$$

Because we seek to maximize rather than minimize P(D|h), we perform gradient ascent rather than gradient descent search. On each iteration of the search the weight vector is adjusted in the direction of the gradient, using the weight update rule

$$w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$$

Where,

$$\Delta w_{jk} = \eta \sum_{i=1}^{m} (d_i - h(x_i))\, x_{ijk} \qquad \text{equ (2)}$$

Where, η is a small positive constant that determines the step size of the i gradient ascent search

MINIMUM DESCRIPTION LENGTH PRINCIPLE

- A Bayesian perspective on Occam's razor
- Motivated by interpreting the definition of h_{MAP} in the light of basic concepts from information theory.

$$h_{MAP} = \underset{h \in H}{argmax}\ P(D|h)P(h)$$

which can be equivalently expressed in terms of maximizing the $\log_2$ Or alternatively, minimizing the negative of this quantity

$$h_{MAP} = \underset{h \in H}{argmax}\ \log_2 P(D|h) + \log_2 P(h)$$
$$h_{MAP} = \underset{h \in H}{argmin}\ -\log_2 P(D|h) - \log_2 P(h) \qquad \text{equ (1)}$$

This equation (1) can be interpreted as a statement that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data

- $-\log_2 P(h)$: the description length of h under the optimal encoding for the hypothesis space H, $L_{C_H}(h) = -\log_2 P(h)$, where C_H is the optimal code for hypothesis space H.
- $-\log_2 P(D | h)$: the description length of the training data D given hypothesis h, under the
  optimal encoding from the hypothesis space H: $L_{C_H}(D|h) = -\log_2 P(D|h)$, where $C_{D|h}$ is the optimal code for describing data D assuming that both the sender and receiver know the hypothesis h.
- Rewrite Equation (1) to show that h_{MAP} is the hypothesis h that minimizes the sum given by the description length of the hypothesis plus the description length of the data given the hypothesis.

$$h_{MAP} = \underset{h \in H}{argmin}\ L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

Where, C_H and $C_{D|h}$ are the optimal encodings for H and for D given h

The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths of equ.

$$h_{MAP} = \underset{h \in H}{argmin}\ L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

Minimum Description Length principle:

$$h_{MDL} = \underset{h \in H}{\operatorname{argmin}} L_{C_1}(h) + L_{C_2}(D \mid h)$$

Where, codes C_1 and C_2 to represent the hypothesis and the data given the hypothesis

The above analysis shows that if we choose C_1 to be the optimal encoding of hypotheses C_H, and if we choose C_2 to be the optimal encoding $C_{D|h}$, then $h_{MDL} = h_{MAP}$

Application to Decision Tree Learning

Apply the MDL principle to the problem of learning decision trees from some training data. *What should we choose for the representations C1 and C2 of hypotheses and data?*

- For C_1: C_1 might be some obvious encoding, in which the description length grows with the number of nodes and with the number of edges
- ForC2: Suppose that the sequence of instances(x1...xm) is already known to both the transmitter and receiver, so that we need only transmit the classifications (f (x1) . . . f (xm)).
- Now if the training classifications (f (x_1) . . .f(x_m)) are identical to the predictions of the hypothesis, then there is no need to transmit any information about these examples. The description length of the classifications given the hypothesis ZERO
- If examples are misclassified by h, then for each misclassification we need to transmit a message that identifies which example is misclassified as well as its correct classification
- The hypothesis h_{MDL} under the encoding C_1 and C_2 is just the one that minimizes the sum of these description lengths.

NAIVE BAYES CLASSIFIER

- The naive Bayes classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target function f (x) can take on any value from some finite set V.
- A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values (a_l, a_2.. .a_m).
- The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach to classifying the new instance is to assign the most probable target value, V_{MAP}, given the attribute values (a_l, a_2.. .a_m) that describe the instance

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

Use Bayes theorem to rewrite this expression as

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

- $$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j) \qquad \textbf{equ (1)}$$

The naive Bayes classifier is based on the assumption that the attribute values are conditionally independent given the target value. Means, the assumption is that given the target value of the instance, the probability of observing the conjunction (a_1, a_2.. .a_m), is just the product of the probabilities for the individual attributes:

$$P(a_1, a_2 \ldots a_n | v_j) = \prod_i P(a_i | v_j)$$

Substituting this into Equation (1),

Naive Bayes classifier:

$$V_{NB} = \underset{v_j \in V}{\text{argmax}}\, P(v_j) \prod_i P(a_i | v_j) \qquad \text{equ (2)}$$

Where, V_{NB} denotes the target value output by the naive Bayes classifier

An Illustrative Example

- Let us apply the naive Bayes classifier to a concept learning problem i.e., classifying days according to whether someone will play tennis.
- The below table provides a set of 14 training examples of the target concept ***PlayTennis***, where each day is described by the attributes Outlook, Temperature, Humidity, and Wind

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

- Use the naive Bayes classifier and the training data from this table to classify the following novel instance:
  < Outlook = sunny, Temperature = cool, Humidity = high, Wind = strong >

- Our task is to predict the target value ***(yes or no)*** of the target concept ***PlayTennis*** for this new instance

$$V_{NB} = \underset{v_j \in \{yes, no\}}{\text{argmax}}\, P(v_j) \prod_i P(a_i|v_j)$$

The probabilities of the different target values can easily be estimated based on their frequencies over the 14 training examples

- P(P1ayTennis = yes) = 9/14 = 0.64
- P(P1ayTennis = no) = 5/14 = 0.36

Similarly, estimate the conditional probabilities. For example, those for Wind = strong

- P(Wind = strong | PlayTennis = yes) = 3/9 = 0.33
- P(Wind = strong | PlayTennis = no) = 3/5 = 0.60

Calculate V_{NB} according to Equation (1)

$$P(yes)\ P(sunny|yes)\ P(cool|yes)\ P(high|yes)\ P(strong|yes) = .0053$$

$$P(no)\ P(sunny|no)\ P(cool|no)\ P(high|no)\ P(strong|no) = .0206$$

Thus, the naive Bayes classifier assigns the target value ***PlayTennis = no*** to this new instance, based on the probability estimates learned from the training data.

By normalizing the above quantities to sum to one, calculate the conditional probability that the target value is ***no***, given the observed attribute values

$$\frac{.0206}{(.0206 + .0053)} = .795$$

Estimating Probabilities

- We have estimated probabilities by the fraction of times the event is observed to occur over the total number of opportunities.
- For example, in the above case we estimated P(Wind = strong | Play Tennis = no) by the fraction n_c /n where, n = 5 is the total number of training examples for which PlayTennis = no, and n_c = 3 is the number of these for which Wind = strong.
- When n_c = 0, then n_c /n will be zero and this probability term will dominate the quantity calculated in Equation (2) requires multiplying all the other probability terms by this zero value
- To avoid this difficulty we can adopt a Bayesian approach to estimating the probability, using the ***m-estimate*** defined as follows

  ***m* -estimate of probability:** $$\frac{n_c + mp}{n + m}$$

- p is our prior estimate of the probability we wish to determine, and m is a constant called the equivalent sample size, which determines how heavily to weight p relative to the observed data

- Method for choosing p in the absence of other information is to assume uniform priors; that is, if an attribute has k possible values we set p = 1 /k.

BAYESIAN BELIEF NETWORKS

- The naive Bayes classifier makes significant use of the assumption that the values of the attributes $a_1 \dots a_n$ are conditionally independent given the target value v.
- This assumption dramatically reduces the complexity of learning the target function

A Bayesian belief network describes the probability distribution governing a set of variables by specifying a set of conditional independence assumptions along with a set of conditional probabilities

Bayesian belief networks allow stating conditional independence assumptions that apply to subsets of the variables

Notation

- Consider an arbitrary set of random variablesY1...Yn, where each variable Yi can take on the set of possible values V(Yi).
- The joint space of the set of variables Y to be the cross product $V(Y_1) \times V(Y_2) \times \dots V(Y_n)$.
- In other words, each item in the joint space corresponds to one of the possible assignments of values to the tuple of variables $(Y_1 \dots Y_n)$. The probability distribution over this joint' space is called the joint probability distribution.
- The joint probability distribution specifies the probability for each of the possible variable bindings for the tuple $(Y_1 \dots Y_n)$.
- A Bayesian belief network describes the joint probability distribution for a set of variables.

Conditional Independence

Let X, Y, and Z be three discrete-valued random variables. X is conditionally independent of Y given Z if the probability distribution governing X is independent of the value of Y given a value for Z, that is, if

$$(\forall x_i, y_j, z_k)\ P(X = x_i | Y = y_j, Z = z_k) = P(X = x_i | Z = z_k)$$

Where,

$$x_i \in V(X),\ y_j \in V(Y), \text{ and } z_k \in V(Z).$$

The above expression is written in abbreviated form as

$$P(X|Y,Z)=P(X|Z)$$

Conditional independence can be extended to sets of variables. The set of variables $X_1 \dots X_l$ is conditionally independent of the set of variables $Y_1 \dots Y_m$ given the set of variables $Z_1 \dots Z_n$ if

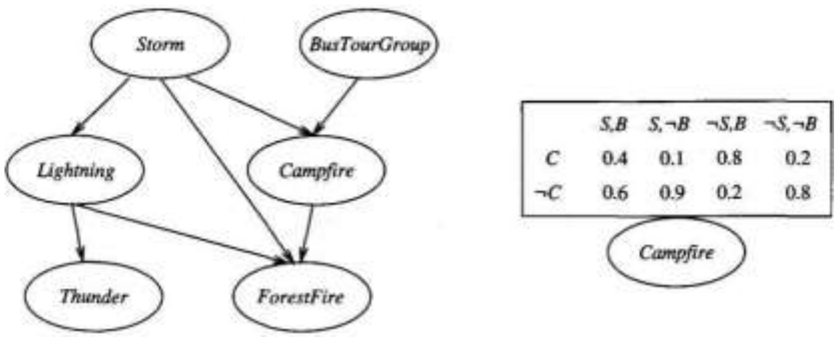
$$P(X_1 \dots X_l | Y_1 \dots Y_m, Z_1 \dots Z_n) = P(X_1 \dots X_l | Z_1 \dots Z_n)$$

The naive Bayes classifier assumes that the instance attribute A_1 is conditionally independent of instance attribute A_2 given the target value V. This allows the naive Bayes classifier to calculate $P(A_1, A_2 \mid V)$ as follows,

$$\begin{aligned} P(A_1, A_2|V) &= P(A_1|A_2, V)P(A_2|V) \\ &= P(A_1|V)P(A_2|V) \end{aligned}$$

Representation

A Bayesian belief network represents the joint probability distribution for a set of variables. Bayesian networks (BN) are represented by directed acyclic graphs.



	S,B	S,¬B	¬S,B	¬S,¬B
C	0.4	0.1	0.8	0.2
¬C	0.6	0.9	0.2	0.8

The Bayesian network in above figure represents the joint probability distribution over the boolean variables *Storm*, *Lightning*, *Thunder*, *ForestFire*, *Campfire*, and *BusTourGroup*

A Bayesian network (BN) represents the joint probability distribution by specifying a set of *conditional independence assumptions*

- BN represented by a directed acyclic graph, together with sets of local conditional probabilities
- Each variable in the joint space is represented by a node in the Bayesian network
- The network arcs represent the assertion that the variable is conditionally independent of its non-descendants in the network given its immediate predecessors in the network.
- A ***conditional probability table*** **(CPT)** is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors

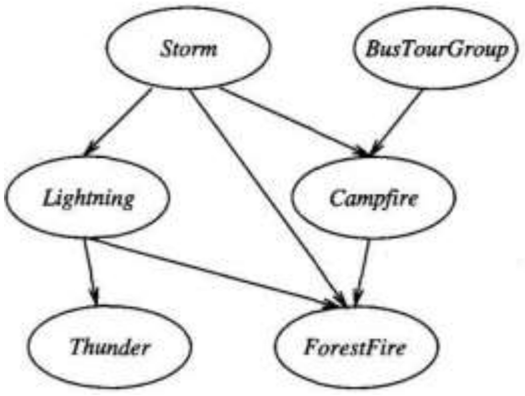
The joint probability for any desired assignment of values $(y_1, \ldots, y_n)$ to the tuple of network variables $(Y_1 \ldots Y_m)$ can be computed by the formula

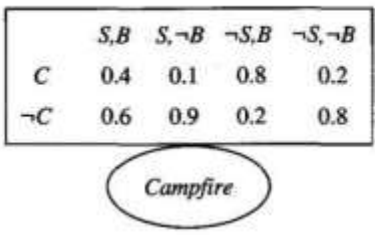
$$P(y_1, \ldots, y_n) = \prod_{i=1}^{n} P(y_i | Parents(Y_i))$$

Where, Parents(Y_i) denotes the set of immediate predecessors of Y_i in the network.

Example:

Consider the node ***Campfire***. The network nodes and arcs represent the assertion that ***Campfire*** is conditionally independent of its non-descendants ***Lightning*** and ***Thunder***, given its immediate parents Storm and ***BusTourGroup***.





	S,B	S,¬B	¬S,B	¬S,¬B
C	0.4	0.1	0.8	0.2
¬C	0.6	0.9	0.2	0.8

This means that once we know the value of the variables ***Storm*** and ***BusTourGroup*** , the variables ***Lightning*** and ***Thunder*** provide no additional information about ***Campfire***

The conditional probability table associated with the variable ***Campfire.*** The assertion is

P(Campfire = True | Storm = True, BusTourGroup = True) = 0.4

Inference

- Use a Bayesian network to infer the value of some target variable (e.g., ForestFire) given the observed values of the other variables.
- Inference can be straightforward if values for all of the other variables in the network are known exactly.
- A Bayesian network can be used to compute the probability distribution for any subset of network variables given the values or distributions for any subset of the remaining variables.
- An arbitrary Bayesian network is known to be NP-hard

Learning Bayesian Belief Networks

Affective algorithms can be considered for learning Bayesian belief networks from training data by considering several different settings for learning problem

➢ First, the network structure might be given in advance, or it might have to be inferred from the training data.

➢ Second, all the network variables might be directly observable in each training example, or some might be unobservable.

- In the case where the network structure is given in advance and the variables are fully observable in the training examples, learning the conditional probability tables is straightforward and estimate the conditional probability table entries
- In the case where the network structure is given but only some of the variable values are observable in the training data, the learning problem is more difficult. The learning problem can be compared to learning weights for an ANN.

Gradient Ascent Training of Bayesian Network

The gradient ascent rule which maximizes P(D|h) by following the gradient of ***ln P(D|h)*** with respect to the parameters that define the conditional probability tables of the Bayesian network.

Let w denote a single entry in one of the conditional probability tables. In particular w_{ijk} ijk denote the conditional probability that the network variable Y_i will take on the value y_i, given that its immediate parents U_i take on the values given by u_{ik}.
The gradient of ***ln P(D|h)*** is given by the derivatives

$$\frac{\partial \ln P(D|h)}{\partial w_{ijk}}$$

As shown below, each of these derivatives can be calculated as

$$\frac{\partial \ln P(D|h)}{\partial w_{ij}} = \sum_{d\in D} \frac{P(Y_i = y_{ij}, U_i = u_{ik}|d)}{w_{ijk}} \qquad \textbf{equ(1)}$$

Derive the gradient defined by the set of derivatives for all ***i, j,*** and ***k***. Assuming the training examples ***d*** in the data set D are drawn independently, we write this derivative as

$$\frac{\partial P_h(D)}{\partial w_{ijk}}$$

$$\begin{aligned}\frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \frac{\partial}{\partial w_{ijk}} \ln \prod_{d\in D} P_h(d)\\ &= \sum_{d\in D} \frac{\partial \ln P_h(d)}{\partial w_{ijk}}\\ &= \sum_{d\in D} \frac{1}{P_h(d)} \frac{\partial P_h(d)}{\partial w_{ijk}}\end{aligned}$$

We write the abbreviation $P_h(D)$ to represent P(D|h).

This last step makes use of the general equality $\frac{\partial \ln f(x)}{\partial x} = \frac{1}{f(x)}\frac{\partial f(x)}{\partial x}$. We can now introduce the values of the variables Y_i and $U_i = Parents(Y_i)$, by summing over their possible values $y_{ij'}$ and $u_{ik'}$.

$$\begin{aligned}\frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \sum_{d\in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} \sum_{j',k'} P_h(d|y_{ij'}, u_{ik'}) P_h(y_{ij'}, u_{ik'})\\ &= \sum_{d\in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} \sum_{j',k'} P_h(d|y_{ij'}, u_{ik'}) P_h(y_{ij'}|u_{ik'}) P_h(u_{ik'})\end{aligned}$$

This last step follows from the product rule of probability . Now consider the rightmost sum in the final expression above. Given that $w_{ijk} \equiv P_h(y_{ij}|u_{ik})$, the only term in this sum for which $\frac{\partial}{\partial w_{ijk}}$ is nonzero is the term for which $j' = j$ and $i' = i$. Therefore

$$\begin{aligned}\frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \sum_{d\in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} P_h(d|y_{ij}, u_{ik}) P_h(y_{ij}|u_{ik}) P_h(u_{ik})\\ &= \sum_{d\in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} P_h(d|y_{ij}, u_{ik}) w_{ijk} P_h(u_{ik})\\ &= \sum_{d\in D} \frac{1}{P_h(d)} P_h(d|y_{ij}, u_{ik}) P_h(u_{ik})\end{aligned}$$

Applying Bayes theorem to rewrite $P_h(d|y_{ij}, u_{ik})$, we have

$$\frac{\partial \ln P_h(D)}{\partial w_{ijk}} = \sum_{d \in D} \frac{1}{P_h(d)} \frac{P_h(y_{ij}, u_{ik}|d) P_h(d) P_h(u_{ik})}{P_h(y_{ij}, u_{ik})}$$
$$= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d) P_h(u_{ik})}{P_h(y_{ij}, u_{ik})}$$
$$= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{P_h(y_{ij}|u_{ik})}$$
$$= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{w_{ijk}} \qquad \text{equ (2)}$$

Thus, we have derived the gradient given in Equation (1). There is one more item that must be considered before we can state the gradient ascent training procedure. In particular, we require that as the weights w_{ijk} are updated they must remain valid probabilities in the interval [0,1]. We also require that the sum $\sum_j w_{ijk}$ remains 1 for all i, k. These constraints can be satisfied by updating weights in a two-step process. First we update each w_{ijk} by gradient ascent

$$w_{ijk} \leftarrow w_{ijk} + \eta \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{w_{ijk}}$$

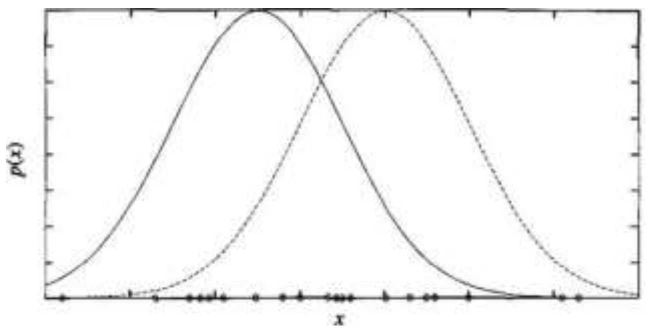
where η is a small constant called the learning rate. Second, we renormalize the weights w_{ijk} to assure that the above constraints are satisfied.
this process will converge to a locally maximum likelihood hypothesis for the conditional probabilities in the Bayesian network.

THE EM ALGORITHM

The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.

EstimatingMeansofkGaussians

Consider a problem in which the data D is a set of instances generated by a probability
distribution that is a mixture of k distinct Normal distributions.



- This problem setting is illustrated in Figure for the case where k = 2 and where the instances are the points shown along the x axis.
- Each instance is generated using a two-step process.
  - First, one of the k Normal distributions is selected at random.
  - Second, a single random instance x_i is generated according to this selected distribution.
- This process is repeated to generate a set of data points as shown in the figure.

- To simplify, consider the special case
  - The selection of the single Normal distribution at each step is based on choosing each with uniform probability
  - Each of the k Normal distributions has the same variance σ^2, known value.
- The learning task is to output a hypothesis $h = (\mu_1, \ldots, \mu_k)$ that describes the means of each of the k distributions.
- We would like to find a maximum likelihood hypothesis for these means; that is, a hypothesis ***h*** that maximizes p(D |h).

$$\mu_{ML} = \operatorname*{argmin}_{\mu} \sum_{i=1}^{m} (x_i - \mu)^2 \qquad (1)$$

In this case, the sum of squared errors is minimized by the sample mean

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^{m} x_i \qquad (2)$$

- Our problem here, however, involves a mixture of k different Normal distributions, and we cannot observe which instances were generated by which distribution.
- Consider full description of each instance as the triple (x_i, z_{i1}, z_{i2}), where xi is the observed value of the ith instance and where zi1 and zi2 indicate which of the two Normal distributions was used to generate the value x_i
- In particular, z_{ij} has the value 1 if x_i was created by the j^{th} Normal distribution and 0 otherwise.
- Here x_i is the observed variable in the description of the instance, and z_{il} and z_{i2} are hidden variables.
- If the values of z_{il} and z_{i2} were observed, we could use following Equation to solve for the means p_1 and p_2
- Because they are not, we will instead use the EM algorithm

EM algorithm

Step 1: Calculate the expected value $E[z_{ij}]$ of each hidden variable z_{ij}, assuming the current hypothesis $h = \langle \mu_1, \mu_2 \rangle$ holds.

Step 2: Calculate a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$, assuming the value taken on by each hidden variable z_{ij} is its expected value $E[z_{ij}]$ calculated in Step 1. Then replace the hypothesis $h = \langle \mu_1, \mu_2 \rangle$ by the new hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$ and iterate.

MODULE 5

INSTANCE BASED LEARNING

INTRODUCTION

- Instance-based learning methods such as nearest neighbor and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions.
- Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the new query instance
- Instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified

Advantages of Instance-based learning

1. Training is very fast
2. Learn complex target function
3. Don'tloseinformation

Disadvantages of Instance-based learning

- The cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered.
- In many instance-based approaches, especially nearest-neighbor approaches, is that they typically consider all attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most "similar" may well be a large distance apart.

k- NEAREST NEIGHBOR LEARNING

- The most basic instance-based method is the K- Nearest Neighbor Learning. This algorithm assumes all instances correspond to points in the n-dimensional space R^n.
- The nearest neighbors of an instance are defined in terms of the standard Euclidean distance.
- Let an arbitrary instance x be described by the feature vector

  $$((a_1(x), a_2(x), \ldots\ldots\ldots, a_n(x))$$

  Where, $a_r(x)$ denotes the value of the r^{th} attribute of instance x.

- Then the distance between two instances x_i and x_j is defined to be $d(x_i, x_j)$ Where,

- $$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n}(a_r(x_i) - a_r(x_j))^2}$$
- In nearest-neighbor learning the target function may be either discrete-valued or real-valued.

$$f : \Re^n \rightarrow V.$$

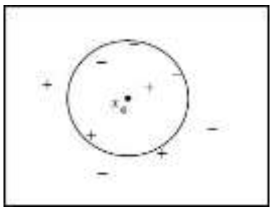
Let us firstconsiderlearning***discrete-valuedtargetfunctions***oftheform Where, V is the finite set {v1, . . . vs }

<u>The k- Nearest Neighbor algorithm for approximation a **discrete-valued target function** is given below:</u>

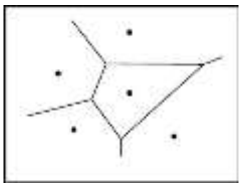
```
Training algorithm:
    • For each training example ⟨x, f(x)⟩, add the example to the list training_examples

Classification algorithm:
    • Given a query instance $x_q$ to be classified,
        • Let $x_1 \dots x_k$ denote the k instances from training_examples that are nearest to $x_q$
        • Return
              $\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$
          where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.
```

- The value ̂(xq) returned by this algorithm as its estimate of f(xq) is just the most common value of f among the k training examples nearest to xq.
- If k = 1, then the 1- Nearest Neighbor algorithm assigns to ̂(xq) the value f(xi). Where xi is the training instance nearest to xq.
- For larger values of k, the algorithm assigns the most common value among the k nearest training examples.

- Below figure illustrates the operation of the k- Nearest Neighbor algorithm for the case where the instances are points in a two-dimensional space and where the target function is Boolean valued.



- The positive and negative training examples are shown by "+" and "-" respectively. A query point x_q is shown as well.
- The 1-Nearest Neighbor algorithm classifies x_q as a positive example in this figure, whereas the 5-Nearest Neighbor algorithm classifies it as a negative example.
- Belowfigureshowstheshapeofthis**decisionsurface**inducedby1-NearestNeighbor over the entire instance space. The decision surface is a combination of convex polyhedral surrounding each of the training examples.



- For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the ***Voronoi diagram*** of the set of training example

The K- Nearest Neighbor algorithm for approximation a **real-valued target function** is given belo $f : \Re^n \rightarrow \Re$

```
Training algorithm:
    • For each training example $\langle x, f(x) \rangle$, add the example to the list training_examples
Classification algorithm:
    • Given a query instance $x_q$ to be classified,
        • Let $x_1 \ldots x_k$ denote the k instances from training_examples that are nearest to $x_q$
        • Return
              $\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$
```

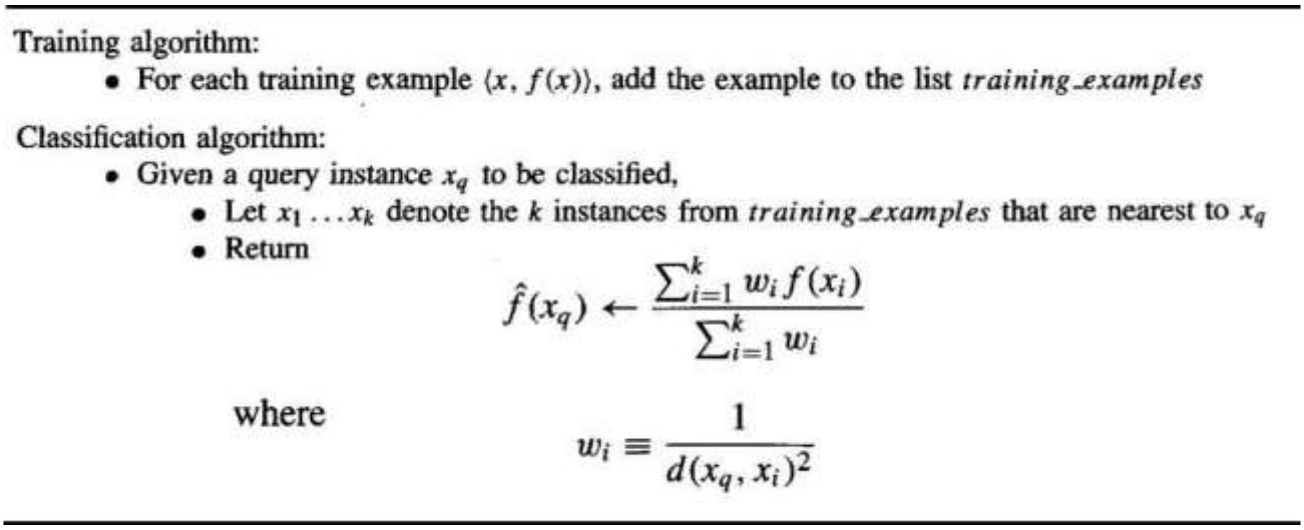
Distance-Weighted Nearest Neighbor Algorithm

- The refinement to the k-NEAREST NEIGHBOR Algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight to closer neighbors.
- For example, in the k-Nearest Neighbor algorithm, which approximates discrete-valued targetfunctions,wemightweightthevoteofeachneighboraccordingtotheinverse square of its distance from xq

Distance-Weighted Nearest Neighbor Algorithm for approximation a discrete-valued target functions

```
Training algorithm:
    • For each training example $\langle x, f(x) \rangle$, add the example to the list training_examples
Classification algorithm:
    • Given a query instance $x_q$ to be classified,
        • Let $x_1 \ldots x_k$ denote the k instances from training_examples that are nearest to $x_q$
        • Return
              $\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$
        where
              $w_i \equiv \frac{1}{d(x_q, x_i)^2}$
```

Distance-Weighted Nearest Neighbor Algorithm for approximation a Real-valued target functions



```
Training algorithm:
    • For each training example ⟨x, f(x)⟩, add the example to the list training_examples
Classification algorithm:
    • Given a query instance x_q to be classified,
        • Let x_1 ... x_k denote the k instances from training_examples that are nearest to x_q
        • Return
```

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

Terminology

- **Regression** means approximating a real-valued target function.
- **Residual** is the error ˆ(x) - f (x) in approximating the target function.
- **Kernel function** is the function of distance that issued to determine the weight of each training example. In other words, the kernel function is the function K such that
  W $_i$ = K(d(x$_i$, x $_q$))

LOCALLY WEIGHTED REGRESSION

- The phrase "**locally weighted regression**" is called **local** because the function is approximated based only on data near the query point, **weighted** because the contribution of each training example is weighted by its distance from the query point, and **regression** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

- Given a new query instance xq, the general approach in locally weighted regression is to construct an approximation ˆthat fits the training examples in the neighborhood
  surrounding xq. This approximation is then used to calculate the value ˆ(xq), which is output as the estimated target value for the query instance.

Locally Weighted Linear Regression

- Consider locally weighted regression in which the target function ***f*** is approximated near x$_q$ using a linear function of the form
  Where, a$_i$(x) denotes the value of the ith attribute of the instance x
- Derived methods are used to choose weights that minimize the squared error summed over the set D of training examples using gradient descent
  Which led us to the gradient descent training rule
  Where, η is a constant learning rate

- Need to modify this procedure to derive a local approximation rather than a global one. Thesimplewayistoredefinetheerrorcriterion E toemphasizefittingthelocaltraining examples. Three possible criteria are given below.

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in \, k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \qquad \textbf{equ(1)}$$

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from x_q :

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x)) \qquad \textbf{equ(2)}$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in \, k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x)) \qquad \textbf{equ(3)}$$

If we choose criterion three and re-derive the gradient descent rule, we obtain the following training rule

$$\Delta w_j = \eta \sum_{x \in \, k \text{ nearest nbrs of } x_q} K(d(x_q, x)) \; (f(x) - \hat{f}(x)) \; a_j(x)$$

The differences between this new rule and the rule given by Equation (3) are that the contribution of instance x to the weight update is now multiplied by the distance penalty **K(d(x_q, x)),** and that the error is summed over only the k nearest training examples.

RADIAL BASIS FUNCTIONS

- One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions
- In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \qquad \textbf{equ (1)}$$

- Where, each $x_u$isaninstancefromXandwherethekernelfunction$K_u(d(x_u,x))$is defined so that it decreases as the distance d(xu, x) increases.
- Here k is a user provided constant that specifies the number of kernel functions to be included.
- ˆis a global approximation to f (x), the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point x_u .

Choose each function $K_u(d(x_u, x))$ to be a Gaussian function centered at the point x_u with some variance $_u^2$

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

- The functional form of equ(1) can approximate any function with arbitrarily small error, provided a sufficiently large number k of such Gaussian kernels and provided the width ^{2} of each kernel can be separately specified
- The function given by equ(1) can be viewed as describing a two layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values

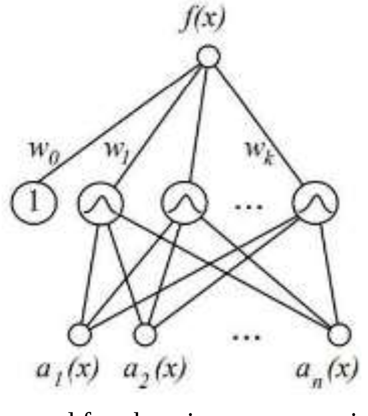
Example: Radial basis function (RBF) network

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process.

1. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and $_u^2$ that define its kernel function $K_u(d(x_u, x))$
2. Second, the weights w, are trained to maximize the fit of the network to the training data, using the global error criterion given by

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2$$

Because the kernel functions are held fixed during this second stage, the linear weight values w, can be trained very efficiently



Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions.

- One approach is to allocate a Gaussian kernel function for each training example $(x_i, f(x_i))$, centering this Gaussian at the point x_i.

  Each of these kernels may be assigned the same width ^{2}. Given this approach, the RBF network learns a global approximation to the target function in which each training example $(x_i, f(x_i))$ can influence the value of ***f*** only in the neighborhood of x_i.
- A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large.

Summary

- Radial basis function networks provide a global approximation to the target function, represented by a linear combination of many local kernel functions.
- The value for any given kernel function is non-negligible only when the input x falls into the region defined by its particular center and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function.
- One key advantage to RBF networks is that they can be trained much more efficiently than feed forward networks trained with BACKPROPAGATION.

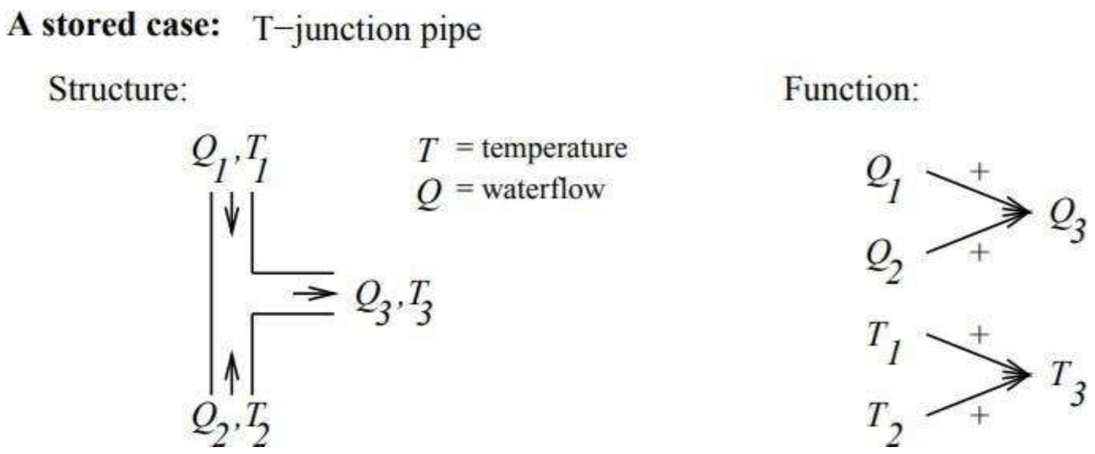
CASE-BASED REASONING

- Case-based reasoning (CBR) is a learning paradigm based on lazy learning methods and they classify new query instances by analyzing similar instances while ignoring instances that are very different from the query.
- In CBR represent instances are not represented as real-valued points, but instead, they use a ***rich symbolic*** representation.
- CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs, reasoning about new legal cases based on previous rulings, and solving planning and scheduling problems by reusing and combining portions of previous solutions to similar problems

A <u>prototypical example of a case-based reasoning</u>

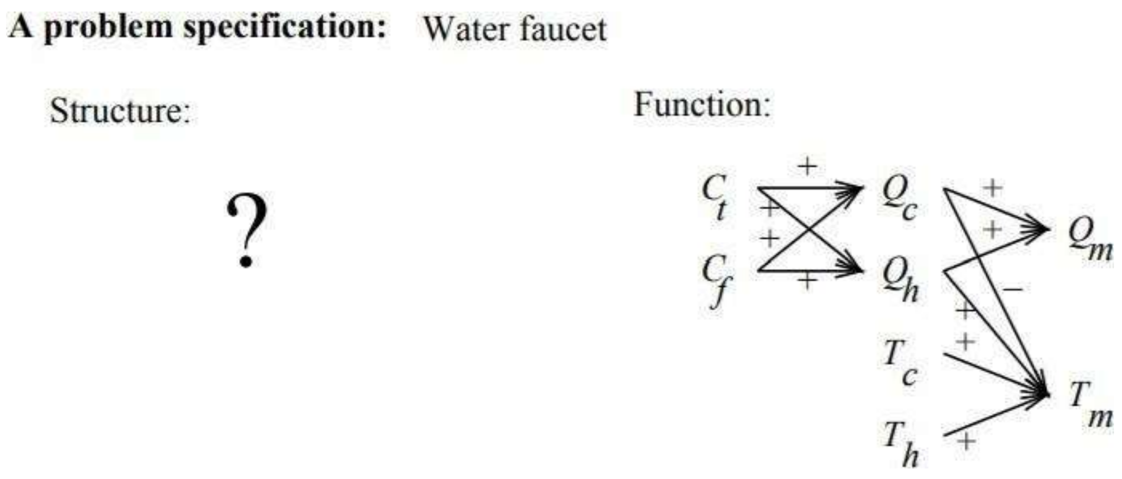
- The CADET system employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets.
- It uses a library containing approximately 75 previous designs and design fragments to suggest conceptual designs to meet the specifications of new design problems.
- Each instancestoredinmemory(e.g.,awaterpipe)isrepresentedbydescribingboth its structure and its qualitative function.
- New design problems are then presented by specifying the desired function and requesting the corresponding structure.

<u>The problem setting is illustrated in below figure</u>



- The function is represented in terms of the qualitative relationships among the water-flow levels and temperatures at its inputs and outputs.
- In the functional description, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail. A "-" label indicates that the variable at the head decreases with the variable at the tail.
- Here Q_c refers to the flow of cold water into the faucet, Q_h to the input flow of hot water, and Q_m to the single mixed flow out of the faucet.

- T_c, T_h, and T_m refer to the temperatures of the cold water, hot water, and mixed water respectively.
- The variable C_t denotes the control signal for temperature that is input to the faucet, and C_f denotes the control signal for waterflow.
- The controls C_t and C_f are to influence the water flows Q_c and Q_h, thereby indirectly influencing the faucet output flow Q_m and temperature T_m.



- CADET searches its library for stored cases whose functional descriptions match the design problem. If an exact match is found, indicating that some stored case implements exactly the desired function, then this case can be returned as a suggested solution to the design problem. If no exact match occurs, CADET may find cases that match various subgraphs of the desired functional specification.

REINFORCEMENT LEARNING

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.

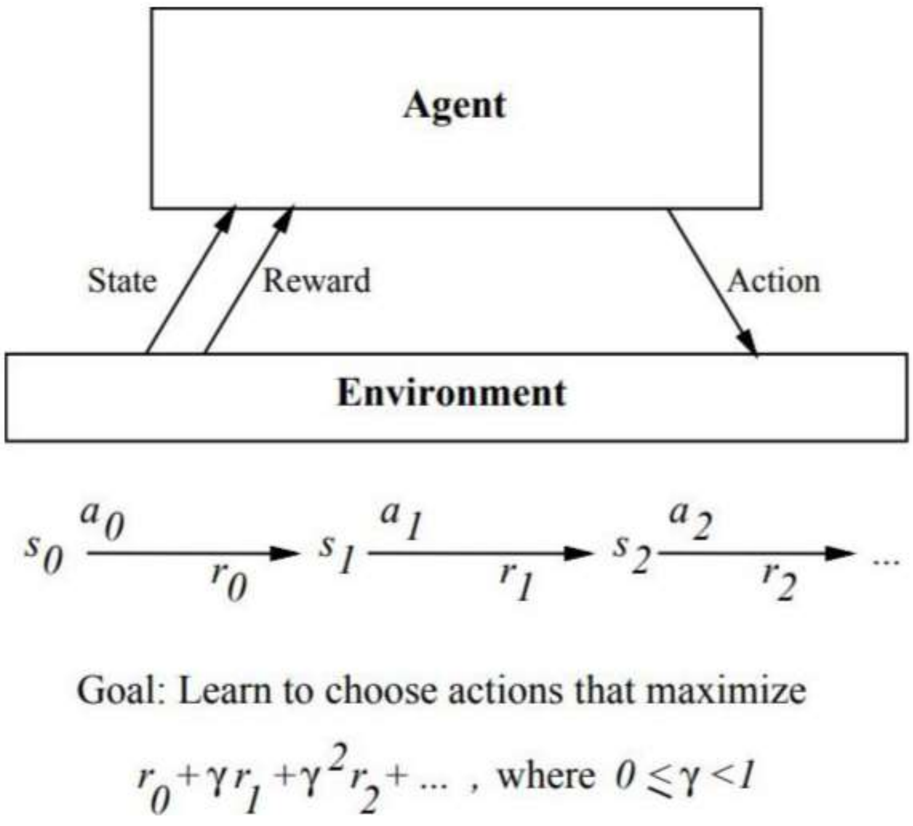
INTRODUCTION

- Consider building a **learning robot**. The robot, or ***agent***, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.
- Its task is to learn a control strategy, or ***policy***, for choosing actions that achieve its goals.
- The goals of the agent can be defined by a ***reward function*** that assigns a numerical value to each distinct action the agent may take from each distinct state.
- This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot.
- The **task** of the robot is to perform sequences of actions, observe their consequences, and learn a control policy.
- The control policy is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.

- Amobilerobotmayhavesensorssuchasacameraandsonars,andactionssuch as "move forward" and "turn."
- The robot may have a goal of docking onto its battery charger whenever its battery level is low.
- The goal of docking to the battery charger can be captured by assigning a positive reward (Eg., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition.

Reinforcement Learning Problem

- An agent interacting with its environment. The agent exists in an environment described by some set of possible states S.
- Agent perform any of a set of possible actions A. Each time it performs an action a, in some state s_t the agent receives a real-valued reward r, that indicates the immediate value of this state-action transition. This produces a sequence of states s_i, actions a_i, and immediate rewards r_i as shown in the figure.
- The agent's task is to learn a control policy, **: S → A**, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.


Agent
State
Reward
Action
Environment
$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots$
Goal: Learn to choose actions that maximize
$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$, where $0 \leq \gamma < 1$

Reinforcement learning problem characteristics

1. **Delayed reward**: The task of the agent is to learn a target function that maps from the current states to the optimal action a = (s). In reinforcement learning, training information is not available in (s, (s)). Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of ***temporal credit assignment***: determining which of the actions in its sequence are to be credited with producing the eventual rewards.

2. **Exploration:** In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a trade-off in choosing whether to favor exploration of unknown states and actions, or exploitation of states and actions that it has already learned will yield high reward.
3. **Partially observable states:** The agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. In such cases, the agent needs to consider its previous observations together with its current sensor data when choosing actions and the best policy may be one that of chooses actions specifically to improve the observability the environment.
4. **Life-long learning:** Robot requires to learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.
5. **Life-long learning:** Robot requires to learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

THE LEARNING TASK

- Consider Markov decision process (MDP) where the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.
- At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it.
- The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+l} = \delta(s_t, a_t)$. Here the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions.

The task of the agent is to learn a policy, **$: S \rightarrow A$**, for selecting its next action a, based on the current observed state s_t; that is, **$(s_t) = a_t$**.

How shall we specify precisely which policy π we would like the a gentto learn?

1. One approach is to require the policy that produces the greatest possible ***cumulative reward*** for the robot over time.

- To state this requirement more precisely, define the cumulative value $V^{\pi}(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad \text{equ (1)}$$

- Where, the sequence of rewards r_{t+i} is generated by beginning at state s_t and by repeatedly using the policy π to select actions.
- Here $0 \leq \gamma \leq 1$ is a constant that determines the relative value of delayed versus immediate rewards. if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward.
- The quantity V^{π} (s_t) is called the ***discounted cumulative reward*** achieved by policy π from initial state ***s***. It is reasonable to discount future rewards to obtain the reward sooner rather than later.

2. Other definitions of total reward is ***finite horizon reward,***

$$\sum_{i=0}^{h} r_{t+i}$$

Considers the undiscounted sum of rewards over a finite number ***h*** of steps

3. Another approach is ***average reward***

$$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$$

Considers the average reward per time step over the entire lifetime of the agent.

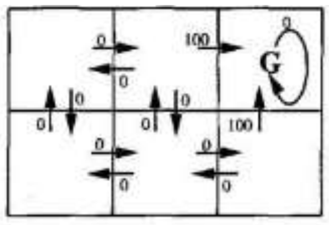
We require that the agent learn a policy π that maximizes V^{π} (s_t) for all states s. such a policy is called an ***optimal policy*** and denote it by π*

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s) \qquad \text{equ (2)}$$

Refer the value function V^{π}*(s) an optimal policy as V*(s). V*(s) gives the maximum discounted cumulative reward that the agent can obtain starting from state***s***.

Example:

A simple grid-world environment is depicted in the diagram

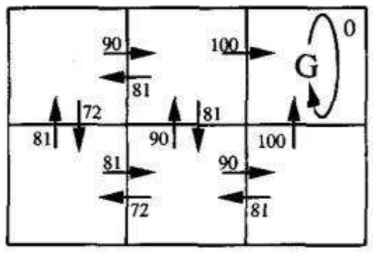


$r(s, a)$ (immediate reward) values

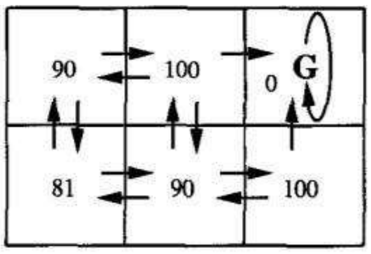
- The six grid squares in this diagram represent six possible states, or locations, for the agent.
- Each arrow in the diagram represents a possible action the agent can take to move from one state to another.
- The number associated with each arrow represents the immediate reward ***r(s, a)*** the agent receives if it executes the corresponding state-action transition
- The immediate reward in this environment is defined to be zero for all state-action transitions except for those leading into the state labeled G. The state G as the goal state, and the agent can receive reward by entering this state.

Once the states, actions, and immediate rewards are defined, choose a value for the discount factor γ, determine the optimal policy π * and its value function V*(s).

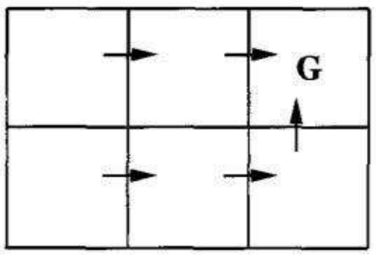
Let's choose γ = 0.9. The diagram at the bottom of the figure shows one optimal policy for this setting.



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

Values of V*(s) and Q(s, a) follow from r(s, a), and the discount factor γ = 0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown.

The discounted future reward from the bottom center state is $0+ \gamma\, 100+ \gamma^2\, 0+ \gamma^3\, 0+... = 90$

Q LEARNING

How can an agent learn an optimal policy π * for an arbitrary environment?

The training information available to the learner is the sequence of immediate rewards $r(s_i,a_i)$ for i = 0, 1,2, ... Given this kind of training information it is easier to learn a numerical Evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

What evaluation function should the agent attempt to learn?

One obvious choice is V*. The agent should prefer state s_1 over state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from s_1

The optimal action in state ***s*** is the action ***a*** that maximizes the sum of the immediate reward r(s, a) plus the value V* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))] \qquad \textbf{equ (3)}$$

The *Q* Function

The value of Evaluation function ***Q(s, a)*** is the reward received immediately upon executing action a from state ***s***, plus the value (discounted by ***γ***) of following the optimal policy thereafter

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \qquad \textbf{equ (4)}$$

Rewrite Equation (3) in terms of ***Q(s, a)*** as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \qquad \textbf{equ (5)}$$

Equation (5) makes clear, it need only consider each available action ***a*** in its current state ***s*** and choose the action that maximizes ***Q(s, a)***.

An Algorithm for Learning *Q*

- Learning the Q function corresponds to learning the **optimal policy**.

  Rewriting Equation

$$V^*(s) = \max_{a'} Q(s,a')$$

- **Q learning algorithm:**

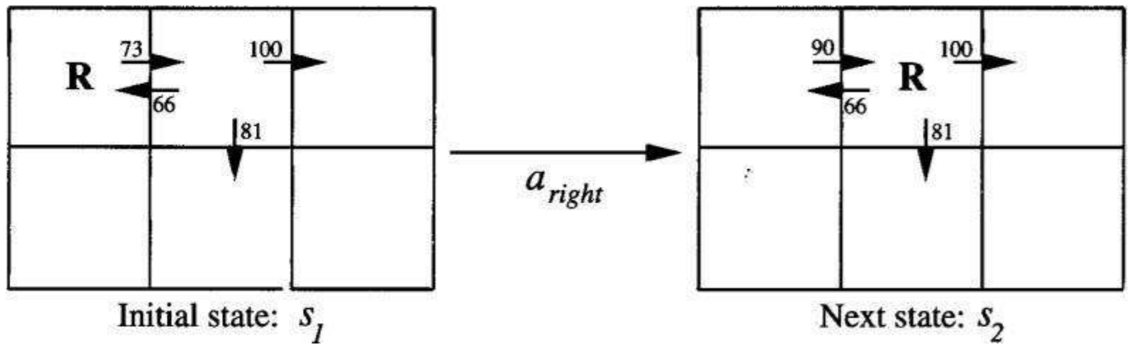
$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a')$$

```
Q learning algorithm
For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.
Observe the current state s
Do forever:
    • Select an action a and execute it
    • Receive immediate reward r
    • Observe the new state s'
    • Update the table entry for $\hat{Q}(s,a)$ as follows:
          $\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$
    • $s \leftarrow s'$
```

- Q learning algorithm assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \le \gamma < 1$
- ˆ to refer to the learner's estimate, or hypothesis, of the actual Q function

An Illustrative Example

- To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement toˆ shown in below figure



- Theagentmovesonecelltotherightinitsgridworldandreceivesanimmediate reward of zero for this transition.
- Apply the training rule of Equation

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

to refine its estimate Q for the state-action transition it just executed.

- According to the training rule, the newˆ estimate for this transition is the sum of the received reward (zero) and the highestˆ value associated with the resulting state (100), discounted by γ (.9).

$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \ \max\{66, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

Convergence

Will the Q Learning Algorithm converge toward a Q equal to the true Q function?

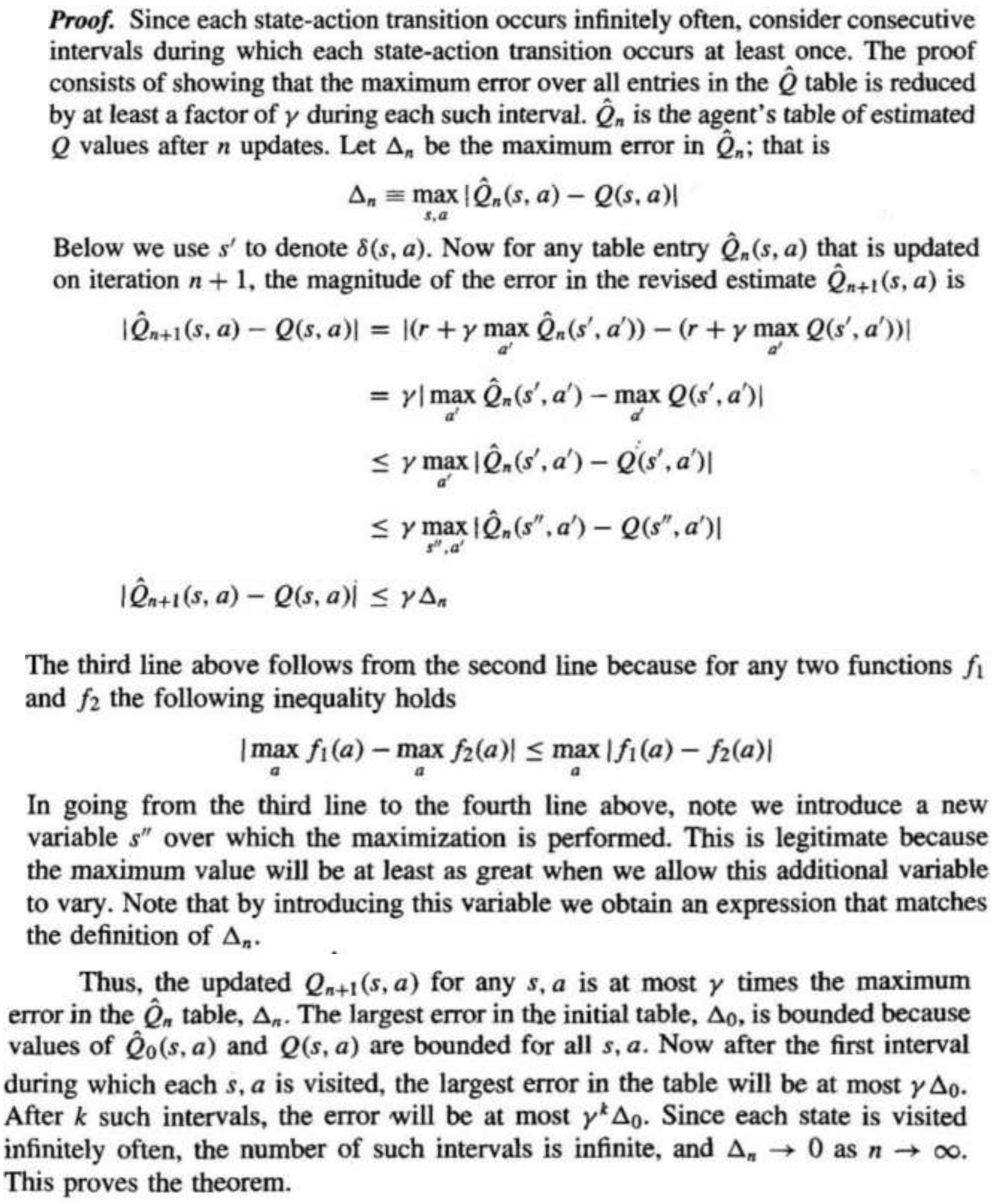
Yes, under certain conditions.

1. Assume the system is a deterministic MDP.
2. Assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, **| r(s, a)| < c**
3. Assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often

Theorem Convergence of Q learning for deterministic Markov decision processes.

Consider a Q learning agent in a deterministic MDP with bounded rewards $(\forall s, a) |r(s, a)| \leq c$.

The Q learning agent uses the training rule of Equation $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$ initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the nth update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \rightarrow \infty$, for all s, a.



Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of γ during each such interval. $\hat{Q}_n$ is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)|$$

Below we use s' to denote $\delta(s,a)$. Now for any table entry $\hat{Q}_n(s,a)$ that is updated on iteration $n+1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s,a)$ is

$$\begin{aligned}
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) - (r + \gamma \max_{a'} Q(s',a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s',a') - \max_{a'} Q(s',a')| \\
&\le \gamma \max_{a'} |\hat{Q}_n(s',a') - Q(s',a')| \\
&\le \gamma \max_{s'',a'} |\hat{Q}_n(s'',a') - Q(s'',a')|
\end{aligned}$$

$$|\hat{Q}_{n+1}(s,a) - Q(s,a)| \le \gamma \Delta_n$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \le \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable s'' over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of Δ_n.

Thus, the updated $Q_{n+1}(s,a)$ for any s, a is at most γ times the maximum error in the $\hat{Q}_n$ table, Δ_n. The largest error in the initial table, Δ_0, is bounded because values of $\hat{Q}_0(s,a)$ and $Q(s,a)$ are bounded for all s, a. Now after the first interval during which each s, a is visited, the largest error in the table will be at most $\gamma\Delta_0$. After k such intervals, the error will be at most $\gamma^k\Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \to 0$ as $n \to \infty$. This proves the theorem.

Experimentation Strategies

The. Q learning algorithm does not specify how actions are chosen by the agent. One

obvious strategy would be for the agent in state ***s*** to select the action ***a*** that

maximizes ̃(s, a), thereby exploiting its current approximation ̂.

- However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high Q values, while failing to explore other actions that have even higher values.

- For this reason, Q learning uses a probabilistic approach to selecting actions. Actions with higherˆ values are assigned higher probabilities, but every action is assigned a nonzero probability.
- One way to assign such probabilities is

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

Where, **P(a$_i$ |s)** is the probability of selecting action **a**$_i$, given that the agent is in state **s**, and **k** > **0** is a constant that determines how strongly the selection favors actions with highˆ values