CHAPTER 1: WHAT IS ARTIFICIAL INTELLIGENCE?

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

OR

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

OR

- 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".

- 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 behavior** 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 behavior. The emphasis is on understanding intelligent behavior.
- **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 behavior 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.

- 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.

The AI Problem

- **AI** seeks to understand the computations required from intelligent behavior 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 proceed:

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?

1.2 Underlying Assumption

Branches of AI

- Logical AI
- Search
- Pattern Recognition
- Representation
- Inference
- Common sense knowledge and Reasoning
- Learning from experience
- Planning
- Epistemology
- Ontology
- Heuristics

Genetic programming
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

What is 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.
- It is huge.
- It is difficult to characterize correctly.
- It is constantly varying.
- It differs from data by being organized in a way that corresponds to its application.
- 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.

The Level of Model

Example-1: Tic-Tac-Toe

1	2	3
4	5	6
7	8	9

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

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 center 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.

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.

CHAPTER 2

Problems, Problem Spaces, and Search

2.1 Defining the Problem as a State Space 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

- 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.

Problem Space

- 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.

Search

- 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.

✓ 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 solved many simple problems, such as the Towers of Hanoi, that could be sufficiently formalized, but GPS could not solve any real-world problems.

- 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.

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.

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

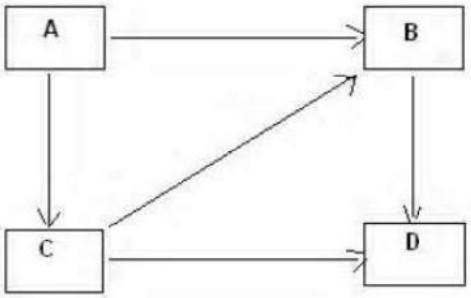
✓ To simplify search algorithms, it is often convenient to logically and programmatically represent a problem space as a tree.

✓ A tree usually decreases the complexity of a search at a cost.

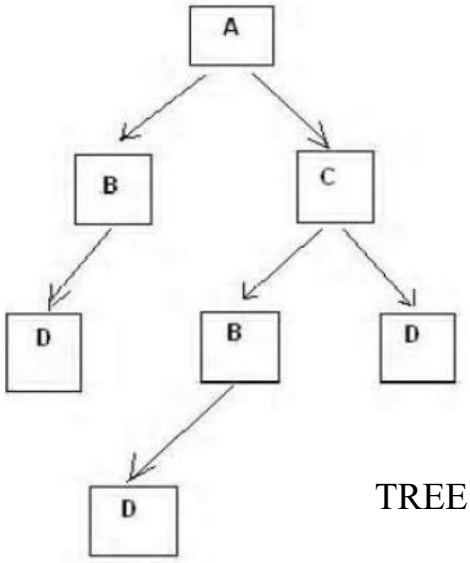
✓ Here, the cost is due to duplicating some nodes on the tree that were linked numerous times in the graph, e.g. node B and node D.

✓ 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.



GRAPH



TREE

Production Systems

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.

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 four = 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.

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

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

- Production systems provide appropriate structures for performing and describing search processes.
- A production system has four basic components :
- 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.

Eight Puzzle Problem

- 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.

2	3	4
8	6	2
7		5

Initial State

1	2	3
8		4
7	6	5

Goal State

Solution:

The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal state.

Rules of solving puzzle

Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile.

The empty space can only **move in four directions** (Movement of empty space)

1. Up
2. Down
3. Right or
4. Left

The empty space **cannot move diagonally** and can take **only one step at a time**.

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); Goal is (NIL 0 0).

(MM 2 0)	**2 Missionaries cross the river**
(MC 1 1)	1 Missionary and 1 Cannibal
(CC 0 2)	2 Cannibals
(M 1 0)	1 Missionary
(C 0 1)	1 Cannibal

State Representation

- <R,M,C>
- Initial state: <1,3,3> <2,0,0>
- <1,3,1> <2,0,2>
- <1,3,2> <2,0,1>
- <1,3,0> <2,0,3>
- <1,3,1> <2,0,2>
- <1,1,1> <2,2,2>
- <1,2,2> <2,1,1>
- <1,0,2> <2,3,1>
- <1,0,3> <2,3,0>
- <1,0,1> <2,3,2>
- <1,0,2> <2,3,1>

- Goal state: <1,0,0> <2,3,3>

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

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.

Algorithm BFS: Breadth First Search

1. Create a variable called NODE_LIST and set it to the initial state.
2. Until a goal state is found or NODE_LIST is empty:

a. Remove the first element from NODE_LIST and call it E. If NODE_LIST was empty, quit.

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

i. Apply the rule to generate a new state.

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

iii. Otherwise, add the new state to the end of NODE_LIST.

Algorithm DFS: Depth First Search

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled.

a. Generate successor, E of the initial state. If there are no more successors, signal failure.

b. Call Depth-First Search with E as the initial state.

c. If success is returned, signal success. Otherwise continue in this loop.

Advantages of BFS:

3. Used to find the shortest path between states.
4. Always finds optimal solutions.
5. There is nothing like useless path in BFS,since it searches level by level.
6. Finds the closest goal state in less time.

Disadvantages of BFS:

All of the connected vertices must be stored in memory. So consumes more memory

Advantages of DFS:

1. Consumes less memory
2. Finds the larger distant element(from initial state) in less time.

Disadvantages of DFS:

1. May not find optimal solution to the problem.

2. May get trapped in searching useless path.

Heuristic Search

- Heuristic is a technique that improves the efficiency of a search process.
- Eg:Nearest neighbour heuristic in travelling salesman problem.
- A heuristic function is a function that maps from problem state descriptions to measures of desirability.
- Which aspects of problem state are considered,
- How those aspects are evaluated , and
- the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to the solution.

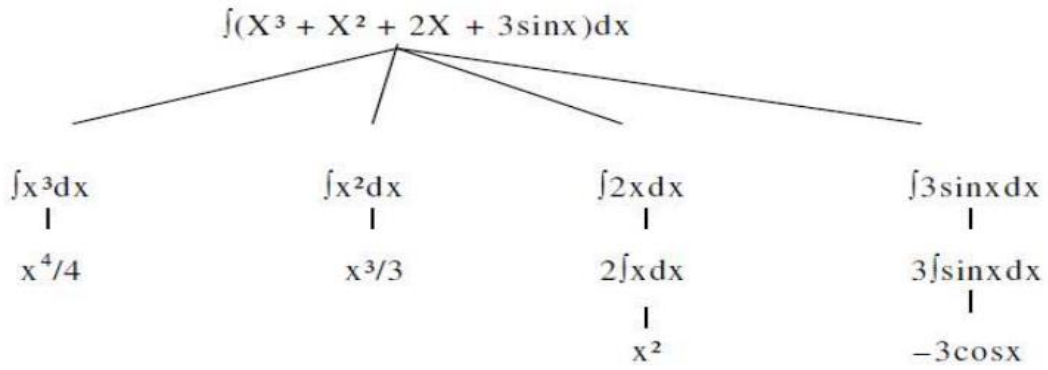
Problem Characteristics

- 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.
- 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:

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

1: Problem Decomposition:

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



1. Monotonic Production System
2. Non monotonic Production System
3. Partially commutative production system
4. Commutative production system:both monotonic and partially commutative.

The Four categories

Production System	**Monotonic**	**Non monotonic**
Partially Commutative	Theorem Proving	Robot Navigation
Not Partially Commutative	Chemical Synthesis	Bridge

Issues in Design of Search Programs

1. The direction in which to conduct the search.
2. How to select applicable rules.
3. How to represent each node of the search process.

Chapter 3

Heuristic Search Techniques

Generate and Test: 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.

• 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.

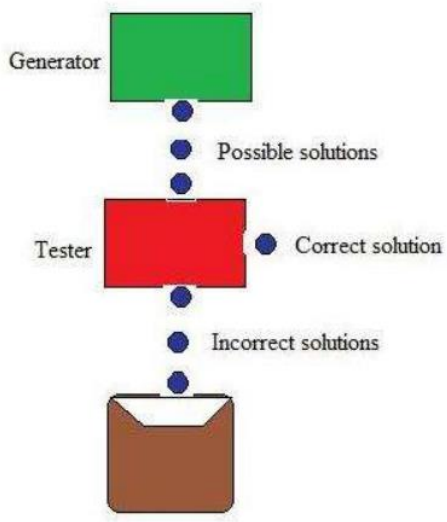


Figure: Generate and Test

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.
- 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.

3. No Backtracking: A hill-climbing algorithm only works on the current state and succeeding states (future). It does not look at the previous states.

4. Feedback mechanism: The algorithm has a feedback mechanism that helps it decide on the direction of movement (whether up or down the hill).

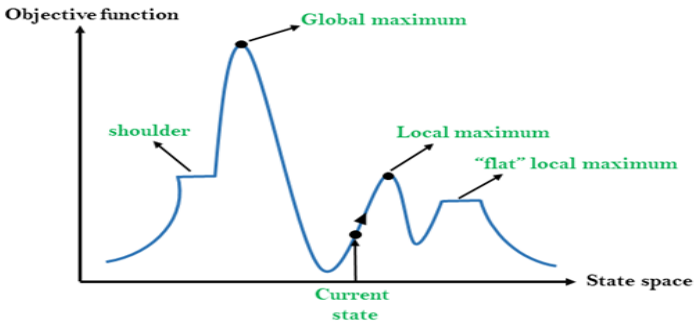
The feedback mechanism is enhanced through the generate-and-test technique.

5. Incremental change: The algorithm improves the current solution by incremental changes.

State Space diagram for Hill Climbing

- State space diagram is a graphical representation of the set of states our search algorithm can reach VS(versus) the value of our objective function(the function which we wish to maximize).
- X-axis : denotes the state space i.e., states or configuration our algorithm may reach.
- Y-axis : denotes the values of objective function corresponding to a particular state.

- The best solution will be that state space where objective function has maximum value(global maximum).



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.

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. Plateau/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 neighbors 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.
- 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.

PROBLEMS OF HILL CLIMBING

- A major problem of hill climbing strategies is their tendency to become stuck at foothills, a plateau or a ridge.
- If the algorithm reaches any of the above mentioned states, then the algorithm fails to find a solution.
- Foothills or **local maxima** is a state that is better than all its neighbors but is not better than some other states farther away.
- At a local maximum, all moves appear to make things worse.
- Foothills are potential traps for the algorithm.

It can be overcome by:

- 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.
- A **plateau** is a flat area of the search space in which a whole set of neighboring states have the same value.
- On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

How 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.
- A **ridge** is a special kind of local maximum.
- It is an area of the search space that is higher that the surrounding areas and that itself has a slope.
- But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.
- Any point on a ridge can look like peak because movement in all probe directions is downward.

It can be overcome by:

- In this kind of obstacle, use two or more rules before testing.
- It implies moving in several directions at once.

Types of Hill Climbing :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

Stochastic Hill Climbing Algorithm:

- 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.

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.

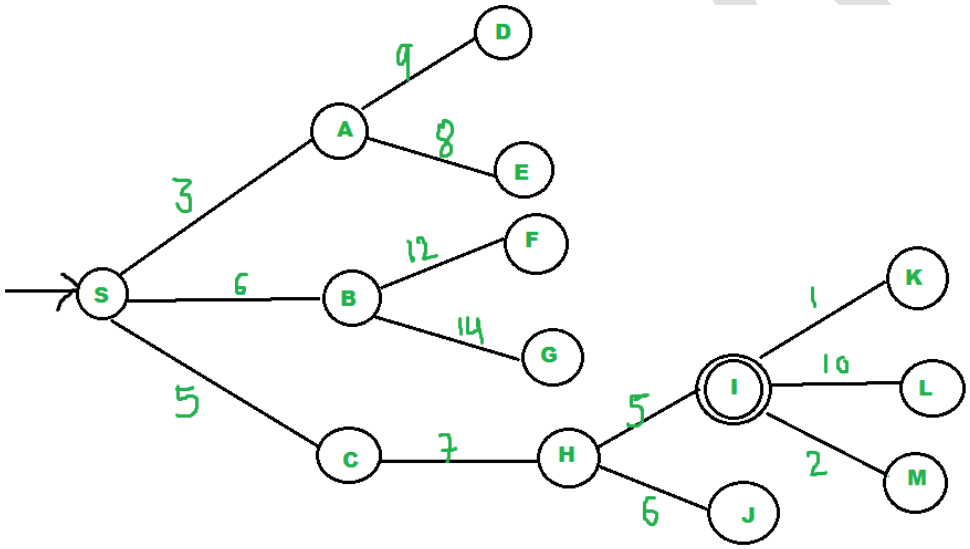
Algorithm: Best-First-Search(Grah g, Node start)

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} 43 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. 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. 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 44 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 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.

AO* Search: (And-Or) Graph

The Depth first search and Breadth first search given earlier for OR trees or graphs can be easilyadopted 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.

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 andplace it in CLOSE.

Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of nas 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 asunsolvable, then return failure and exit.

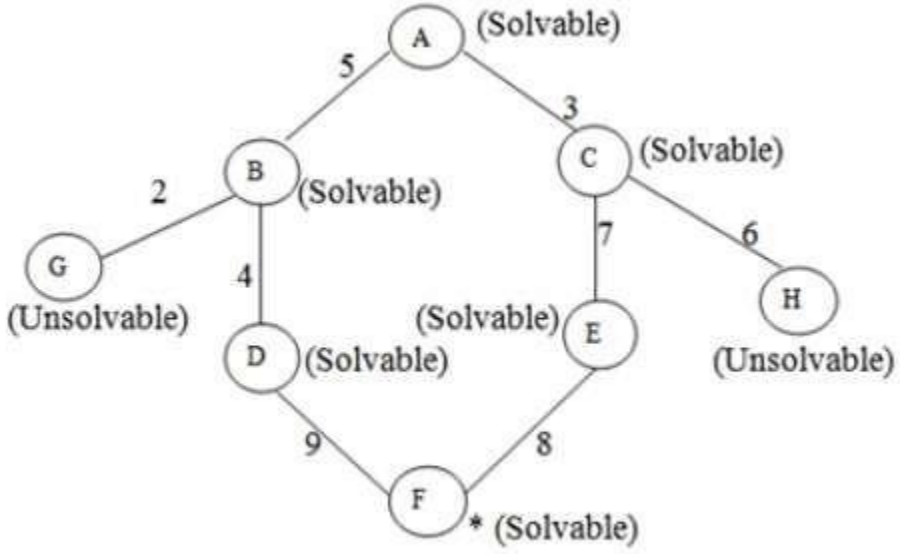
Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.

Step 7: Return to Step 2.

Step 8: Exit

Implementation:

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



Figure

Step 1:

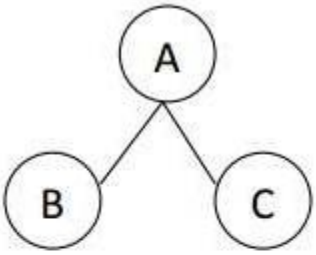
In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H.Take A as the starting node. So place A into OPEN.

i.e. OPEN = [A] CLOSE = (NULL) [ϕ] (A)

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 = | B | C | CLOSE = | A |

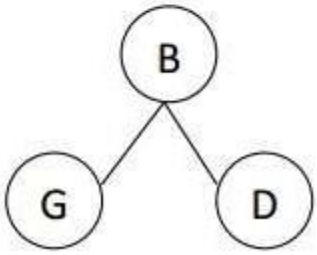


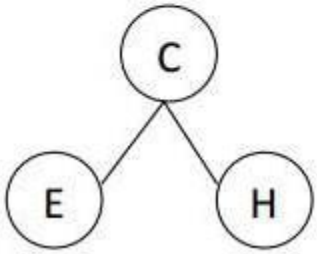
Step 3:

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 = | G | D | E | |

C | A | B | C |





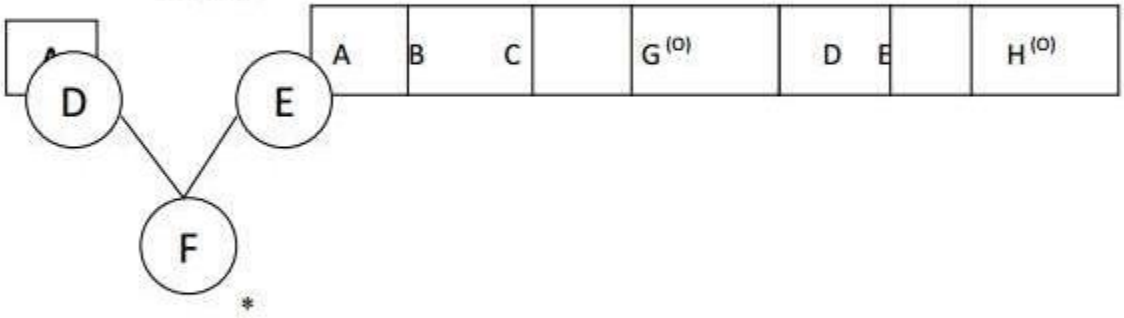
(O)

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

Step 4:

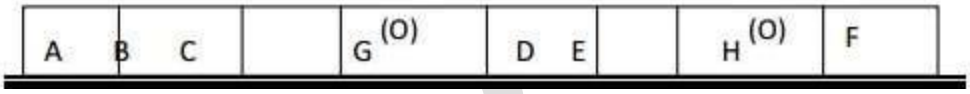
As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e. OPEN = CLOSE =



Step 5:

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

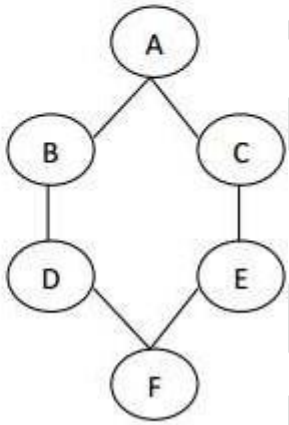


i.e. CLOSE =

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

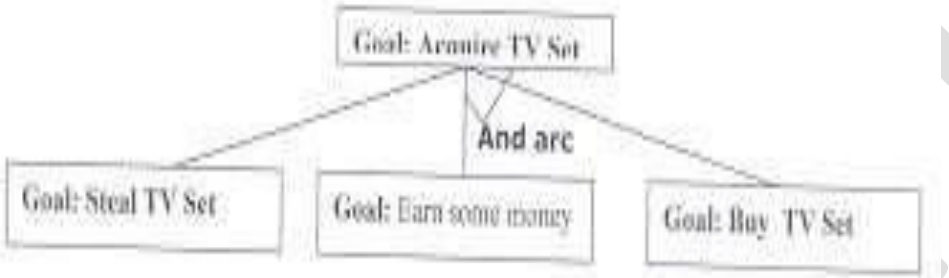


Figure shows AND - Or graph - an example.

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.

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

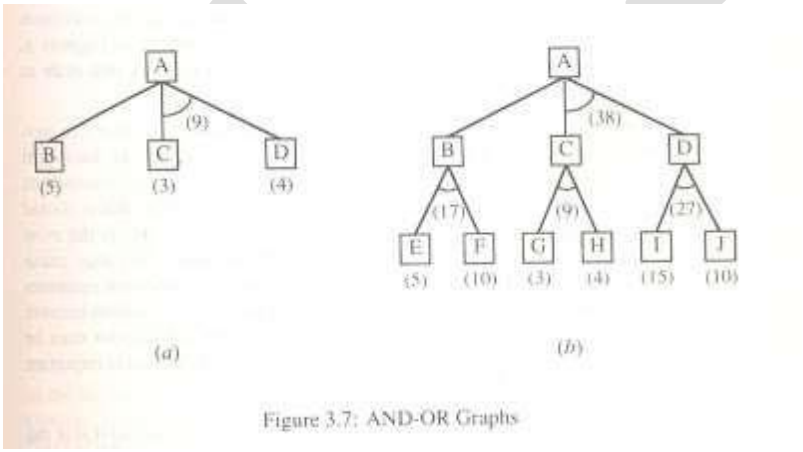


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

In figure (a) the top node A has been expanded producing two area one leading to B and leadingto 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).

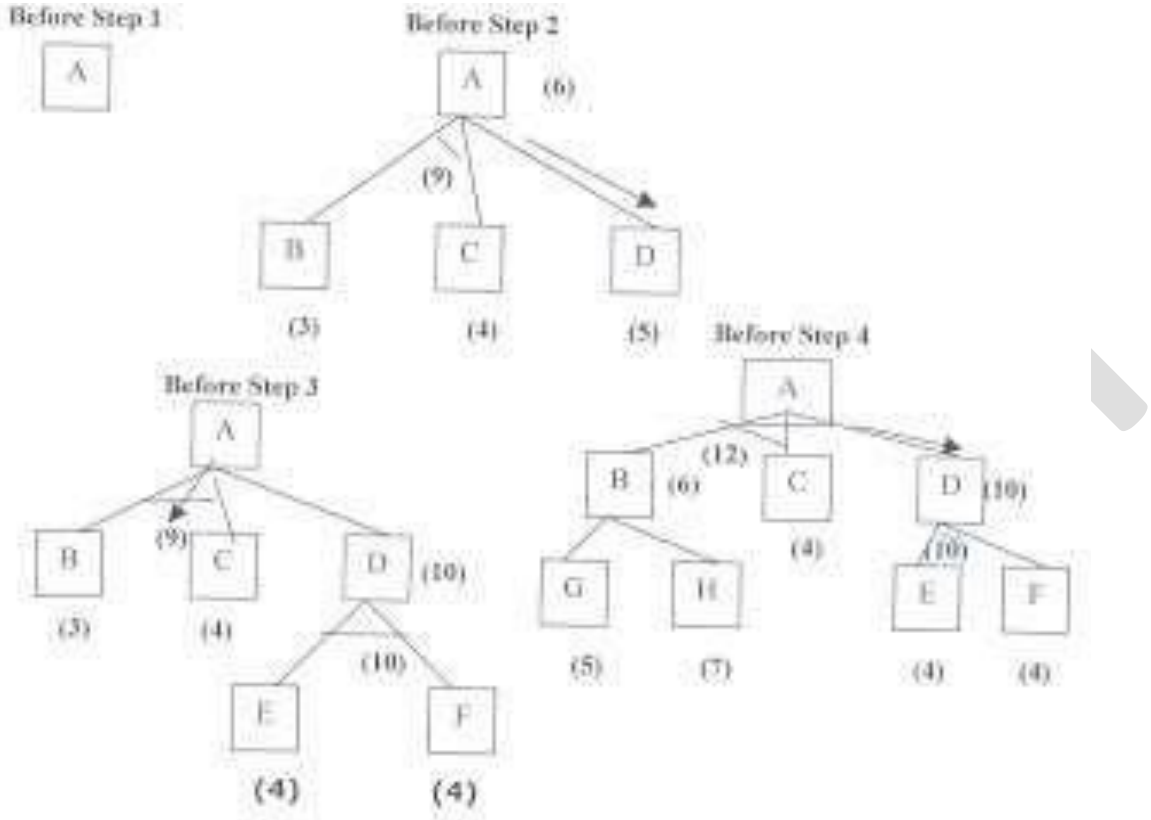
Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figurethe node G appears to be the most promising node, with the least f ' value. But G is not on the current beat 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, andaccumulate 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 andcomputer 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 producedby 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 currentbest 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. Dis expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we cansee 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 isabandoned 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. Computeh' (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 ash' (NODE). This means that NODE is not solvable. If there are successors then for each
on
e 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 aset 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' toCURRENT.

(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 markedare 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 addedto 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 isthe 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, exitwith 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 goalstate 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 newconstraints.

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 generatenew 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 correctto 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 approachmay 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:

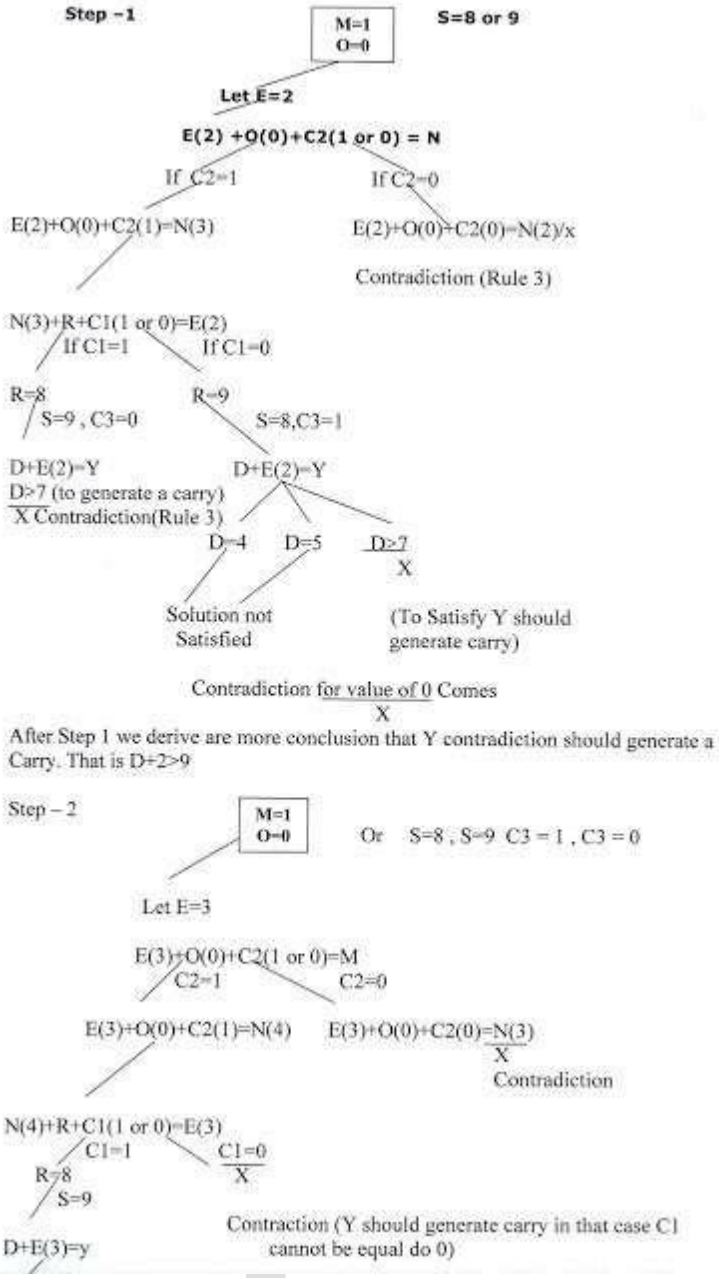
We are following the depth-first method to solve the problem.

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 thato=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 receivedfrom the earlier sum.

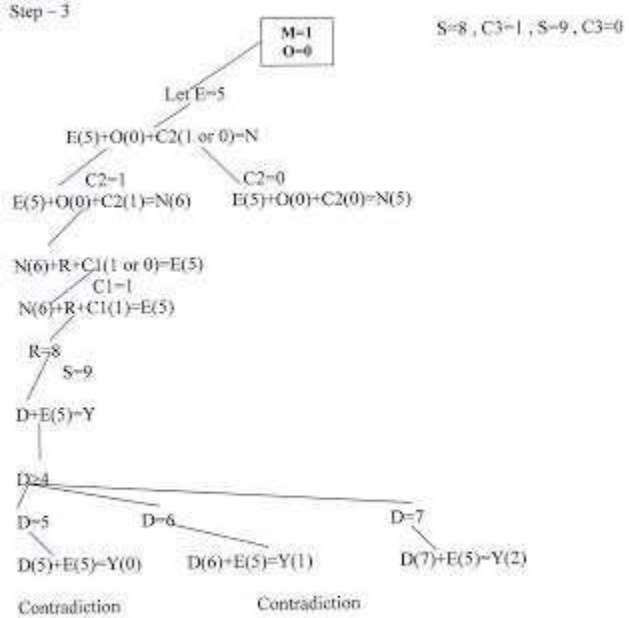
The same process can be repeated further. The problem has to be composed into various constraints. And each constraints is to be satisfied by guessing the possible digits that the letterscan be assumed that the initial guess has been already made . rest of the process is being shownin the form of a tree, 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

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

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.

CHAPTER 4
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 to facts.

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 useand 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.

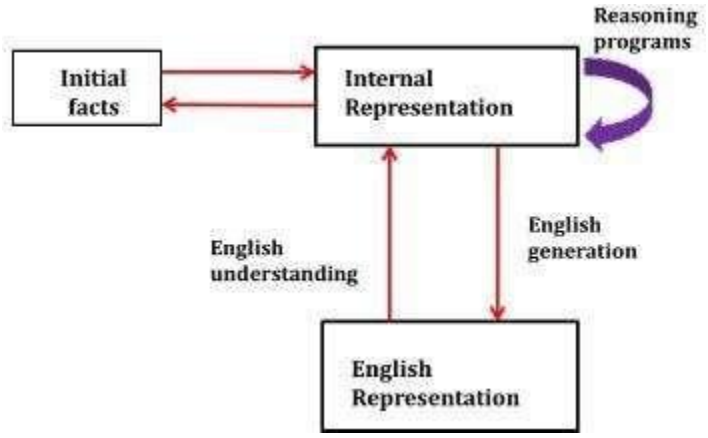


Fig: Mapping between Facts and Representations

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,

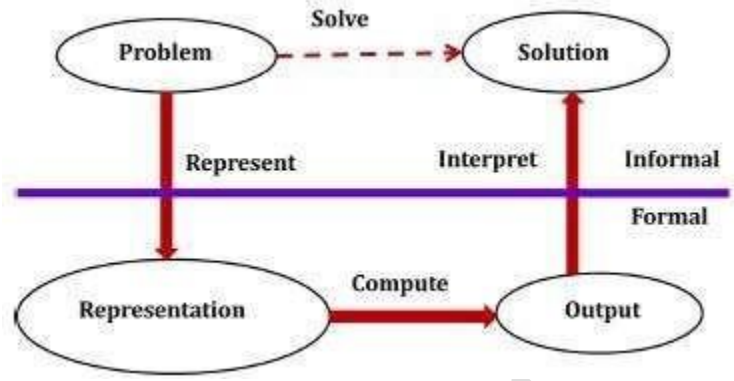


Fig: Knowledge Representation Framework

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 be represented in an informally described solution that user understandsor 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

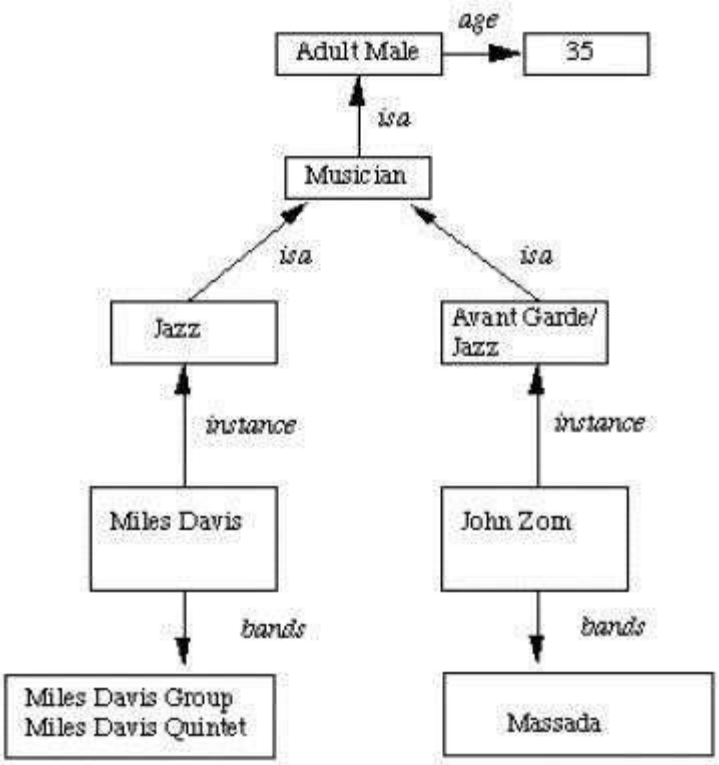
- The simplest way to represent declarative facts is a set of relations of the same sort used in the database system.
- Provides a framework to compare two objects based on equivalent attributes.
- Any instance in which two different objects are compared is a relational type of knowledge.
- The table 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 ∀x*: dog(x) → 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.

CHAPTER 4

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, allvariables 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) A 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) A¬ hate(x, Caesar)) ∨
    - (¬loyalto(x, Caesar) A 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) A ruler(y) A 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.

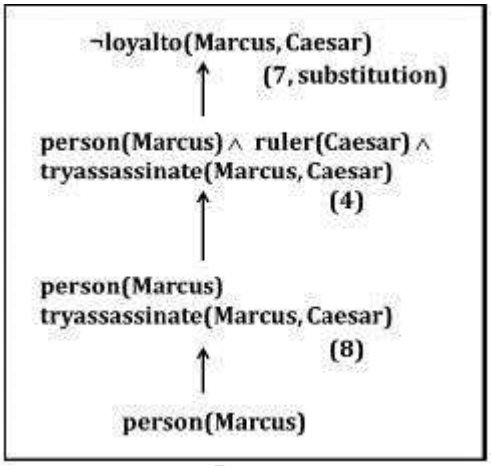


Figure: An attempt to prove ¬loyalto(Marcus, Caesar).

- 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 thenusing 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. ∀x: Pompeian(x) → Roman(x).
4. ruler(Caesar).
5. ∀x: Roman(x) → loyalto(x, Caesar) ∨ hate(x, Caesar).
1. instance(Marcus, man).
2. instance(Marcus, Pompeian).
3. ∀x: instance(x, Pompeian) → instance(x, Roman).
4. instance(Caesar, ruler).
5. ∀x: instance(x, Roman). → loyalto(x, Caesar) ∨ hate(x, Caesar).
1. instance(Marcus, man).
2. instance(Marcus, Pompeian).
3. isa(Pompeian, Roman)
4. instance(Caesar, ruler).
5. ∀x: instance(x, Roman). → loyalto(x, Caesar) ∨ hate(x, Caesar).
6. ∀x: ∀y: ∀z: instance(x, y) ∧ isa(y, z)→ 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 ¬ *L* such that one of the parent clauses contains *L* and the other contains ¬*L*, then select one such pair and eliminate both *L* and ¬ *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 eachother. 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 (A) has an introduction rule and two elimination rules:

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

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.

[x : P] (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 $P \Rightarrow Q$ by discharging a prior assumption [x : P]. Intuitively, if Q can be proved under the assumption P, then the implication $P \Rightarrow Q$ 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 $\neg P$ can be considered an abbreviation for $P \Rightarrow \bot$:

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

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).

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

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**.

CHAPTER 6
REPRESENTING KNOWLEDGE USING RULES

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 knowledgeis 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 → 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. Also, The first two differences are the fact that PROLOG programs are actually sets ofHorn 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 inthe 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

$\forall x : pet(x) \wedge small(x) \rightarrow apartmentpet(x)$

$\forall x : cat(x) \wedge dog(x) \rightarrow pet(x)$

$\forall x : poodle(x) \rightarrow dog(x) \wedge small(x)$

$poodle(fluffy)$

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

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

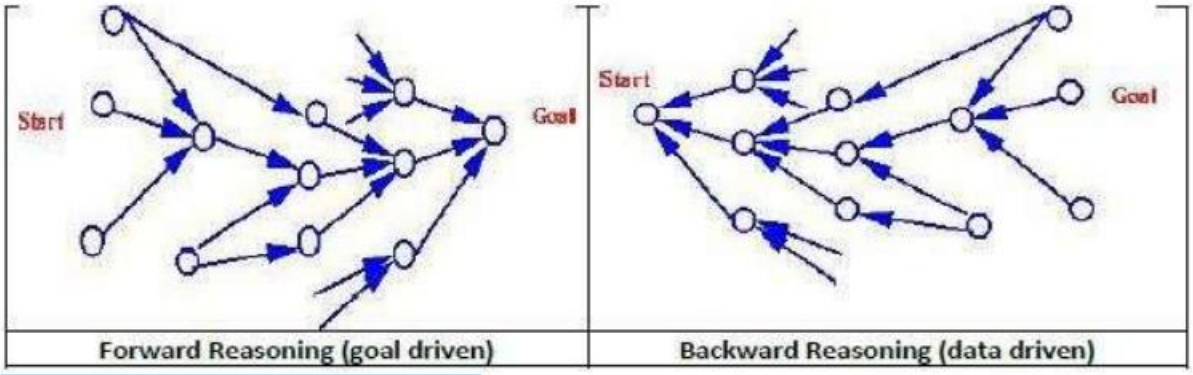
• Prolog is an example of backward chaining rule system. In Prolog rules restricted to Horn clauses. This allows for rapid indexing because all the rules for deducing a given fact share the same rule head. Rules matched with unification procedure. Unification tries to find a set of bindings for variables 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.



Forward versus Backward Reasoning

• Also, The two searches may pass each other resulting in more work.

• 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.

MACHINE LEARNING

INTRODUCTION

Ever since computers 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. This is machine learning.

Examples: Computers 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.

WELL-POSED LEARNING PROBLEMS

Definition: ***A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.***

For example, a computer program that learns to play checkers might improve its performance as ***measured by its ability to win (P)*** at the class of tasks involving ***playing checkers games (T),*** through experience ***obtained*** by ***playing games against itself (E).***

In general, to have a well-defined learning problem, we must identity these three features: the class of tasks, the measure of performance to be improved, and the source of experience.**A checkers learning proble m**:

- Task T: playing checkers
- Performance measure P: percent of games won against opponents
- Training experience E: playing practice games against itself

We can specify many learning problems in this fashion, such as learning to recognizehandwritten words, or learning to drive a robotic automobile autonomously.

A handwriting recognition learning problem:

- Task T: recognizing and classifying handwritten words within images
- Performance measure P : percent of words correctly classified
- Training experience E: a database of handwritten words with givenclassifications

A robot driving learning problem:

- Task T: driving on public four- lane highways using vision sensors
- Performance measure P: average distance traveled before an error
- Training experience E: a sequence of images and steering commands recorded whileobserving a human driver

Some successful applications of machine learning

- ✓ **Learning to recognize spoken words.**

Neural network learning methods and methods for learning are effective for automatically customizing to, individual speakers, vocabularies, microphone characteristics, background noise, etc. Similar techniques have potential applications in many signal-interpretation problems

- ✓ **Learning to drive an autonomous vehicle.**

Machine learning methods have been used to train computer- controlled vehicles to steer correctly when driving on a variety of road types. For example, the ALVINN system (Pomerleau 1989) has used its learned strategies to drive unassisted at 70 miles per hour for90 miles on public highways among othercars.

- ✓ **Learning to classify ne w astronomical structures.**

Machine learning methods have been applied to a variety of large databases to learn general regularities implicit in the data. For Example:decision tree learning algorithms

- ✓ **Learning to play world-class backgammon.**

The most successful computer programs for playing games such as backgammon are based on machine learning algorithms.

Some disciplines and examples of their influence on machine learning.

1. Artificial intelligence: Using prior knowledge together with training data to guide learning.
2. Bayesian methods: Bayes' theorem is the basis for calculating probabilities of hypotheses. The naive Bayes classifier Algorithms for estimating values of unobserved variables.
3. Computational complexity theory: Complexity of different learning tasks, measured in terms of the computational effort, number of training examples, number of mistakes, etc. required in order to learn.
4. Control theory: Procedures that learn to control processes in order to optimize predefined objectives and that learn to predict the next state of the process they are controlling.
5. Information theory: Measures of entropy and information content.

6. Philosophy: Occam's razor, suggesting that the simplest hypothesis is the best. Analysis of the justification for generalizing beyond observed data.

7. Psychology and neurobiology: Neurobiological studies motivating artificial neural network models of learning.

8. Statistics: Characterization of errors (e.g., bias and variance) and Confidence intervals, statistical tests.

DESIGNING A LEARNING SYSTEM

In order to illustrate some of the basic design issues and approaches to machine learning, let us consider designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament. We adopt the obvious performance measure: the percent of games it wins in this world tournament.

Choosing the Training Experience

- The first design choice to choose the type of training experience from which our system will learn.
- The type of training experience available can have a significant impact on success or failure of the learner.
- One key attribute is whether the training experience provides direct or indirect feedback regarding the choices made by the performance system.
- For example, in learning to play checkers, the system might learn from direct training examples consisting of individual checkers board states and the correct move for each. Alternatively, it might have available only indirect information consisting of the move sequences and final outcomes of various games played.
- A second important attribute of the training experience is the degree to which the learner controls the sequence of training examples
- A third important attribute of the training experience is how well it represents the distribution of examples over which the final system performance P must be measured.

A checkers learning problem:

Task T: playing checkers

Performance measure P: percent of games won in the world tournament Training experience E: games played against itself

In order to choose the correct type follow the rules below:

1. The exact type of knowledge to be learned
2. A representation for this target knowledge
3. A learning mechanism

Choosing the Target Function

The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program. A checkers-playing program that can generate the legal moves from any board state.

The program needs only to learn how to choose the best move from among these legal moves.

Let us call this function ChooseMove and use the notation ChooseMove : B->M to indicate that this function accepts as input any board from the set of legal board states B and produces as output some move from the set of legal moves M. It is useful to reduce the problem of improving performance P at task T to the problem of learning some particular arget function such as Choose Move. The choice of the target function will therefore be a key design choice.

Let us therefore define the target value V (b) for an arbitrary board state b in B , as follows:

1. if b is a final board state that is won, then V (b)= 100
2. if b is a final board state that is lost, then V (b)= -100
3. if b is a final board state that is drawn, then V (b)= 0
4. if b is a not a final state in the game, the n V(b) = V(b'), where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game (assuming the opponent plays optimally, as well).

We say that it is a nonoperational definition. The goal of learning in this case is to discoveran operational description of V that is a description that can be used by the checkers-playing program to evaluate states and select moves within realistic time bounds.

In fact, we often expect learning algorithms to acquire only some approximation to the target function, and for this reason the process of learning the target function is often called function approximation. In the current discussion we will use the symbol $V^{\wedge}$ to refer to the function that is actually learned by our program, to distinguish it from the ideal target function V.

Choosing a Representation for the Target Function

Now that we have to keep the discussion brief, let us choose a simple representation: for any given board state,the function c will be calculated as a linear combination of the following board features:

x_1 : the number of black pieces on the board

x_2 : the number of white pieces on the boardx_3 : the number of black kings on the board x4: the number of white kings on the board

x5: the number of black pieces threatened by white (i.e., which can be captured on

white'snext turn)

x_6 : the number of white pieces threatened by black

Thus, our learning program will represent c(b) as a linear function of the form

Partial design of a checkers learning program:

Task T: playing checkers

Performance measure P: percent of games won in the world tournamentTraining experience E: games played against itself

Target function: Target function representation

Choosing a Function Approximation Algorithm

In order to learn the target function f we require a set of training examples, each describing aspecific board state b and the training value for b.

In other words, each training example is an ordered pair of the form (b,).

For instance, the following training example describes a board state b in which black has won the game (note x2 = 0 indicates that red has no remaining pieces) and for which the target function value Vtrain (b) is therefore +100.

<<x1=3, x2=0, x3=1, x4=0, x5=0, x6=0>, +100)

Estimating Training Values

Recall that according to our formulation of the learning problem, the only training information available to our learner is whether the game was eventually won or lost.

On the other hand, we require training examples that assign specific scores to specific board states. While it is easy to assign a value to board states that correspond to the end of the game, it is less obvious how to assign training values to the more numerous intermediate board states that occur before the game's end. For example, even if the program loses the game, it may still be the case that board states occurring early in the game should be rated very highly and that the cause of the loss was a subsequent poor move.

Despite the ambiguity inherent in estimating training values for intermediate boa rd states, one simple approach has been found to be surprisingly successful. This approach is to assign the training value of V_{train} (b) for any intermediate board state b to be

(successor (b)),where is the learner's current approximation to V and where S uccessor (b) denotes the next board state following b for which it is again the program's turn to move .This rule for estimating training values can be summarized as

Adjusting the Weights

The algorithm can be viewed as performing a stochastic grad ient-descent search through thespace of possible hypotheses (weight values) to minimize the squared error E.
The LMS algorithm is defined as follows:

LMS weight update rule.

- For each training example (b, (b))
- Use the current weights to calculate (b)
- For each weight, update it.

Here η is a small constant (e.g., 0.1) that moderates the size of the weight update. Notice that when the error (Vtrain(b)- (b)) is zero, no weights are changed. When (Vtrain(b) - (b)) is positive (i.e., when f(b) is too low), then each weight is increased in proportion to the value of its corresponding feature. This will raise the value of (b), reducing the error.

The Final Design

The final design of our checkers learning system can be naturally described by four distinct program modules that represent the central components in many learning systems. These four modules, summarized in Figure 1.1, are as follows:

- The Performance System is the module that must solve the given performance task,in this case playing checkers, by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output.
- The Critic takes as input the history or trace of the game and produces as output a set of training examples of the target function. As shown in the diagram, each training examplein this case corresponds to some game state in the trace, along with an estimate Vtrain,f the target function value for this example.

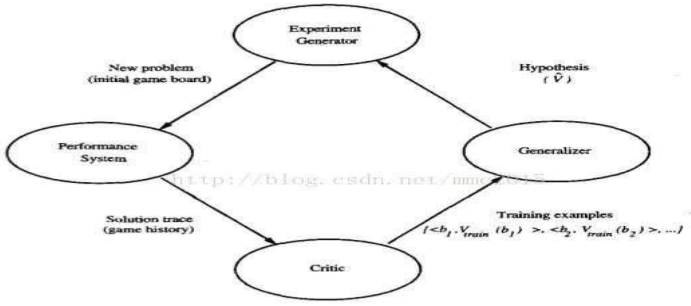


FIGURE 1.1
Final design of the checkers learning program.

- The Generalizer takes as input the training examples and produces an output hypothesis that is its estimate of the target function. It generalizes from the specific training examples, hypothesizing a general function that covers these examples and other cases beyond the training examples. In our example, the generalize corresponds to the LMS algorithm, and the output hypothesis is the function f described by the learned weights wo, . .

. , W6.

- The Experiment Generator takes as input the current hypothesis (currently learned function) and outputs a new problem (i.e., initial board state) for the Performance System to explore. Its role is to pick new practice problems that will maximize the learning rate of the overall system.
- In our example, the Experiment Generator follows a very simple strategy. It always proposes the same initial game board to begin a new game.

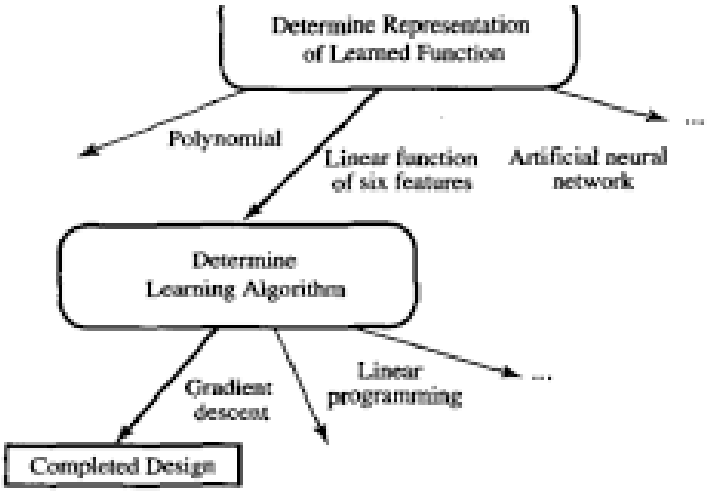


Figure 1.2 Summary of choices in designing the checkers learning program.

PERSPECTIVES AND ISSUES IN MACHINE LEARNING

- One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner.
- The hypothesis space consists of all evaluatio n functions that can be represented by some choice of values for the weights w_o through w_6. The learner's task is thus to search through this vast space to locate the hypothesis that is most consistent with the available training examples.
- The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value.

Issues in Machine Learning

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space?
- When and how can prior knowledge held by the learner guide the process of generalizing from examples?
  - Can prior knowledge be helpful even when it is only approximately correct?
  - What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?
- What is the best way to reduce the learning task to one or more function approximationproblems?
- How can the learner automatically alter its representation to improve its ability torepresent and learn the target function?

INTRODUCTION

- Much of learning involves acquiring general concepts from specific training examples. People, for example, continually learn general concepts or categories suchas "bird", "car", etc.
- Each such concept can be viewed as describing some subset of objects or events defined over a larger set (e.g., the subset of animals that constitute birds).
- Each concept can be thought of as a boolean-valued function defined over this larger set (e.g., a function defined over all animals, whose value is true for birds and false for other animals).
- **Concept learning.** Inferring a boolean-valued function from training examples of its input and output.

CONCEPT LEARNING TASK

- Consider the example task of learning the target concept "days on which a person enjoys his favourite water sport." Table 2.1 describes a set of example days, each represented by a set of ***attributes.*** The attribute ***EnjoySport*** indicates whether or not a person enjoys his favourite water sport on this day.
- The task is to learn to predict the value of ***EnjoySport*** for an arbitrary day, based on the values of its other attributes.
- Let us begin by considering a simple representation in which each hypothesisconsists of a conjunction of constraints on the instance attributes.
- In particular, let each hypothesis be a vector of six constraints, specifying the valuesof 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 *"∅" phi* that no value isacceptable.
- If some instance x satisfies all the constraints of hypothesis h, then h classifies x asa positive example (h(x) = 1).

To illustrate, the hypothesis that Aldo enjoys his favorite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression

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

• The most general hypothesis-that every day is a positive example- is represented by

<?, ?, ?, ?, ?, ?>

• And the most specific possible hypothesis-that no day is a positive example-is represented by <□ , □ , □, □, □, □ >

To summarize, the EnjoySport concept learning task requires learning the set of days for which EnjoySport = yes, describing this set by a conjunction of constraints over the instance attributes.

In general, any concept learning task can be described by the set of instances over which the target function is defined, the target function, the set of candidate hypotheses considered by the learner, and the set of available training examples.

Notation

The EnjoySport concept learning task

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), and 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), "0 (no value is acceptable), or a specific value.

Target concept c: EnjoySport: X → {0,l)}

Training examples D: Positive and negative examples of the target function (see Table 2.1).

DETERMINE:

A hypothesis h in H such that h(x) = c(x) for all x in X.

The Inductive Learning Hypothesis

The inductive learning hypothesis is any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well ove r 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. It is important to note that by selecting a hypothesis representation, the designer of the learning algorithm implicitly defines the space of all hypotheses that the program can ever represent and therefore can ever learn.
- Consider, for example, the instances X and hypotheses H in the EnjoySport learning task.
- Given that the attribute Sky has three possible values, and that AirTemp, Humidity, Wind, Water, and Forecast each have two possible values, the instance space X contains exactly

  3.2 .2 .2 2 .2 = 96 distinct instances.

- A similar calculation shows that there are

  5.4.4.4.4.4 = 5120 syntactically distinct hypotheses within H.

- However, that every hypothesis containing one or more "□" symbols represents the empty set of instances; that is, it classifies every instance as negative. Therefore, the number of semantically distinct hypotheses is only 1 + (4.3.3.3.3.3) = 973.

General-to-Specific Ordering of Hypotheses

• A general-to-specific ordering of hypotheses. By taking advantage of this naturally occurring structure over the hypothesis space, we can design learning algorithms that exhaustively search even infinite hypothesis spaces without explicitly enumerating every hypothesis.

• To illustrate the general-to-specific ordering, consider the two hypotheses.

h1 = <Sunny, ?, ?, Strong, ?, ?>

h2 = <Sunny, ?, ?, ?, ?, ?>

• Now consider the sets of instances that are classified positive by hl and by h2. Because h2imposes fewer constraints on the instance, it classifies more instances as positive.

In fact, any instance classified positive by hl will also be classified positive by h2. Therefore,we say that h2 is more general than h1.

We will also find it useful to consider cases where one hypothesis is strictly more general than the other.

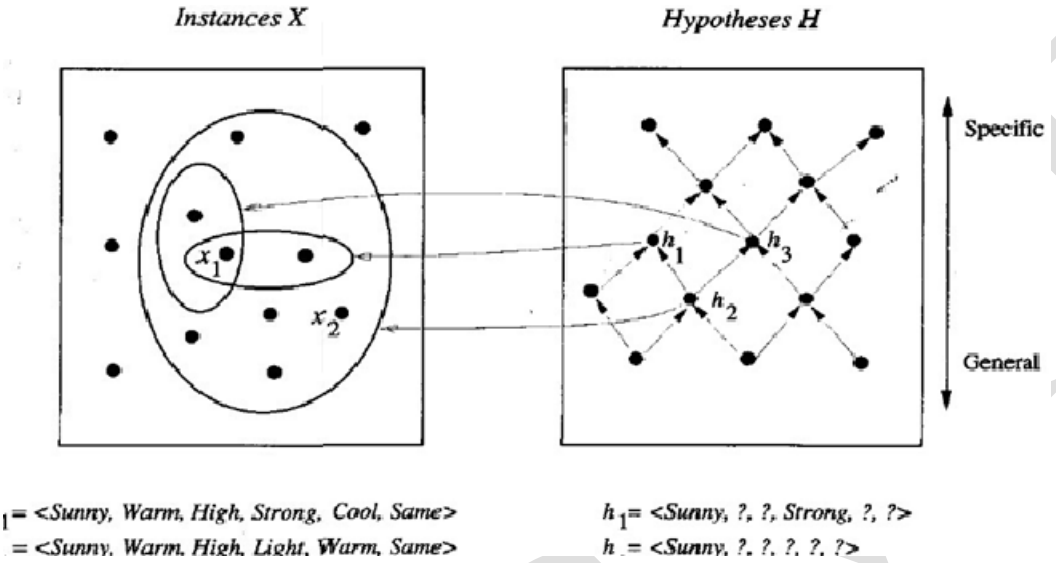


Figure 2.1

Therefore, we will say that $\mathbf{h_j}$ is (strictly) more_general_than $\mathbf{h_k}$ (written $\mathbf{h_j} >_g \mathbf{h_k}$) if andonly if $(\mathbf{h_j} \geq_g \mathbf{h_k}) \wedge (\mathbf{h_k} \geq_g \mathbf{h_j})$.

Finally, we will sometimes find the inverse useful and will say that **hj** is

more_specific_than $\mathbf{h_k}$ when $\mathbf{h_k}$ is **more_general-than** $\mathbf{h_j}$.

To illustrate these definitions, consider the three hypotheses h_l, h_2, and h_3 from our EnjoySport shown in Figure 2.1

Hypothesis $\mathbf{h_2}$ is more general than $\mathbf{h_l}$ because every instance that satisfies $\mathbf{h_l}$ also satisfies $\mathbf{h_2}$. Similarly, $\mathbf{h_2}$ is more general than $\mathbf{h_3}$. Note that neither $\mathbf{h_l}$ nor $\mathbf{h_3}$ is more general than the other; although the instances satisfied by these two hypotheses intersect, neither set subsumes the another.

Formally, the $\geq_g$, relation defines a partial order over the hypothesis space H (the relation is reflexive, anti-symmetric, and transitive). Informally, when we say the structure is a partial(as opposed to total) order, we mean there may be pairs of hypotheses such as $\mathbf{h_l}$ and $\mathbf{h_3}$, suchthat $\mathbf{h_l} \geq_g \mathbf{h_3}$ and $\mathbf{h_3} \geq_g \mathbf{h_l}$.

The $\geq_g$ relation is important because it provides a useful structure over the hypothesis spaceH for *any* concept learning problem.

FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

• How can we use the more- general-than partial ordering to organize the search for a hypothesis consistent with the observed training examples? One way is to begin with the most specific possible hypothesis in H, then generalize this hypothesis each time it fails to cover an observed positive training example. (We say that a hypothesis "covers" a positive example if it correctly classifies the example as positive.)

• FIND-S algorithm defined above, assume the learner is given the sequence of training examples from for the EnjoySport task.

• The first step of FIND-S is to initialize h to the most specific hypothesis in H Upon observing the first training example from Table 2.1, which happens to be a positive example, it becomes clear that our hypothesis is too specific.

• In particular, none of the "□ " constraints in h are satisfied by this example, so each is replaced by the next more general constraint that fits the example; namely, the attribute values for this training example.

h (Sunny, Warm, Normal, Strong, Warm, Same)

• This h is still very specific; it asserts that all instances are negative except for the single positive training example we have observed. Next, the second training example (also positive in this case) forces the algorithm to further generalize h, this time substituting a "?'

Pseudo code of algorithm :

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

in place of any attribute value in h that is not satisfied by the new example. The refined hypothesis in this case is

h *(Sunny, Warm, ?, Strong, Warm, Same)*

• Upon encountering the third training example- in this case a negative example the algorithm makes no change to h. In fact, the FIND-S algorithm simply ignores every negative example!

• To complete our trace of FIND-S, the fourth (positive) example leads to a further generalization of h

h *(Sunny, Warm, ?, Strong, ?, ?)*

• The FIN D-S algorithm illustrates one way in which the more-general- than partial ordering can be used to organize the search for an acceptable hypothesis.

• The search moves from hypothesis to hypothesis, searching from the most specific to progressively more general hypotheses along one chain of the partial ordering.

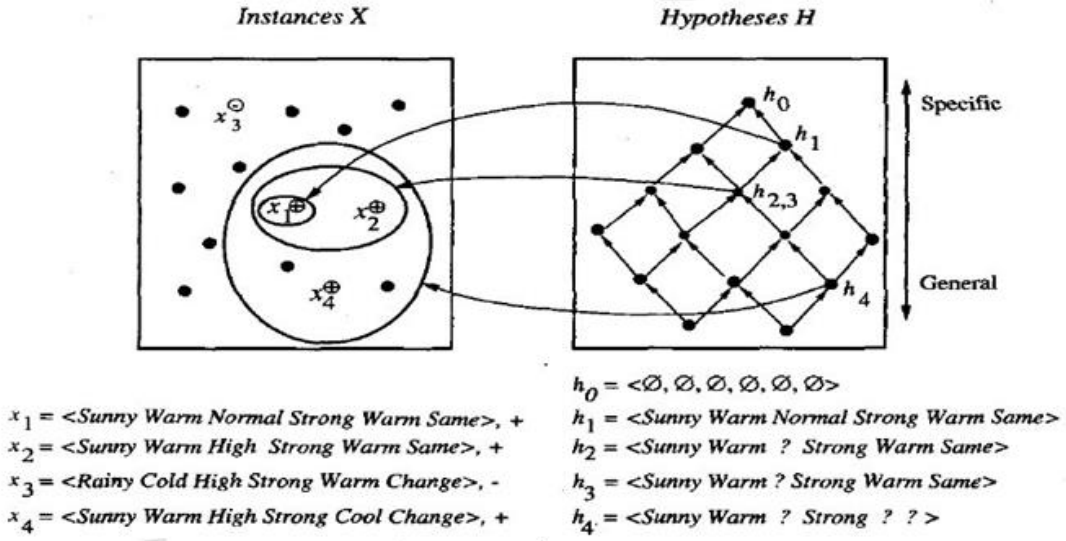


FIGURE 2.2: The hypothesis space search performed by FIND-S.

Figure 2.2: The Hypothesis space search performed by Find-S.

The search begins (h_o) with the most specific hypothesis in H, then considers increasingly general hypotheses (h_l through h_4) as mandated by the training examples. In the instance space diagram, positive training examples are denoted by "+" and negative by "-," and instances that have not been presented as training examples are denoted by a solid circle.

Complaints about Find-S

- Has the learner converged to the correct target concept? Although FIND-S will find a

hypothesis consistent with the training data, it has no way to determine whether it has found the ***only*** hypothesis in ***H*** consistent with the data (i.e., the correct target concept), or whether there are many other consistent hypotheses as well

- Why prefer the most specific hypothesis? In case there are multiple hypotheses consistent with the training examples, FIND-S will find the most specific. It is unclear whether we should prefer this hypothesis over, say, the most general, or some other hypothesis of intermediate generality.
  - Are the training examples consistent? In most practical learning problems there is some chance that the training examples will contain at least some errors or noise. Such inconsistent sets of training examples can severely mislead FIND-S, given the fact that it ignores negative examples.
  - What if there are several maximally specific consistent hypotheses? There can be several maximally specific hypotheses consistent with the data. In this case, FIND-S must be extended to allow it to backtrack on its choices.

VERSION SPACES AND THECANDIDATE-ELIMINATION ALGORITHM

This section describes a second approach to concept learning, the CANDIDATE ELIMINATION algorithm that addresses several of the limitations of FIND-S.

- Notice that although FIND-S outputs a hypothesis from H, that is consistent with the training examples, this is just one of many hypotheses from H that might fit the training data equally well.
- The key idea in the CANDIDATE- ELIMINATION ALGORITHM is to output a description of the set ofall hypotheses consistent with the training examples.

Surprisingly, the CANDIDATE-ELIMINATION ALGORITHM computes the description of this set without explicitly enumerating all of its members.

- This is accomplished by again using the more-general-than partial ordering, this time to maintain a compact representation of the set of consistent hypotheses and to incrementally refine this representation as each new training example is encountered.

Representation

The CANDIDATE-ELIMINATION ALGORITHM finds all desirable hypotheses that are consistent with the observed training examples.

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

$$Consistent(h, D) \equiv (\forall \langle x, c(x)\rangle \in D)\ h(x) = c(x)$$

Definition: **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.

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

The LIST-THEN-ELIMINATE Algorithm

The LIST-THEN-ELIMINATE Algorithm

1. *VersionSpace* ← a list containing every hypothesis in H
2. For each training example, $\langle x, c(x)\rangle$
   remove from *VersionSpace* any hypothesis h for which $h(x) \neq c(x)$
3. Output the list of hypotheses in *VersionSpace*

- First initializes the version space to contain all hypotheses in H, then eliminates any hypothesis found inconsistent with any training example. The version space of candidate hypotheses thus shrinks as more examples are observed.
- Unfortunately, it requires exhaustively enumerating all hypotheses in H-an unrealistic requirement for all but the most trivial hypothesis spaces.

A More Compact Representation for Version Spaces

To illustrate this representation for version spaces, consider again the Enjoysport concept learning problem described in "**Notations**". Recall that given the four training examples from Table 2.1, FIND-S outputs the hypothesis

h = <Sunny, Warm, ?, Strong, ?, ?>

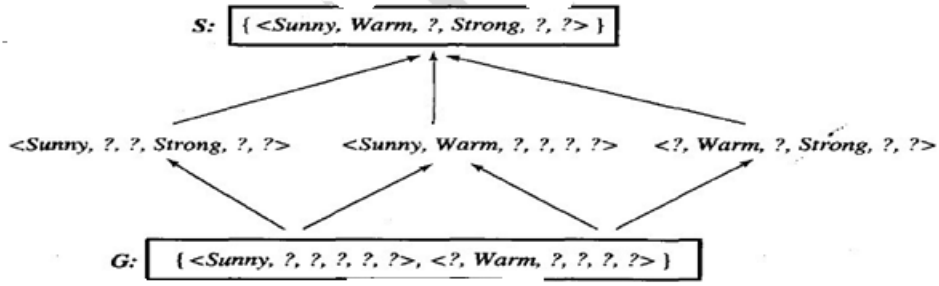


FIGURE 2.3: A version space with its general and specific boundary sets.

The version space includes all six hypotheses shown here, but can be represented more simply by S and G. Arrows indicate instances of the more- general-than relation. This is the version space for the Enjoysport concept learning problem and training examples describedin Table 2.1.

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.

$$G \equiv \{g \in H | Consistent(g, D) \wedge (\neg \exists g' \in H)[(g' >_g g) \wedge Consistent(g', D)]\}$$

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*

$$S \equiv \{s \in H | Consistent(s, D) \wedge (\neg \exists s' \in H)[(s >_g s') \wedge Consistent(s', D)]\}$$

CANDIDATE-ELIMINATION LEARNING ALGORITHM

- The CANDIDATE-ELIMINATION ALGORITHM computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples.
- It begins by initializing the version space to the set of all hypotheses in H; that is, by initializing the G boundary set to contain the most general hypothesis in H

  G_o {(?, ?, ?, ?, ?, ?)}

And initializing the S boundary set to contain the most specific (least general hypothesis S_o {0,0,0,0, 0,0)

These two boundary sets delimit the entire hypothesis space, because every other hypothesisin H is both more general than S_o and more specific than G_o. As each training example is

considered, the S and G boundary sets are generalized and specialized, respectively, to

eliminate from the version space any hypotheses found inconsistent with the new trainingexample.

CANDIDATE-ELIMINATION Algorithm using version space 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 anot her 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

    Remove g from G

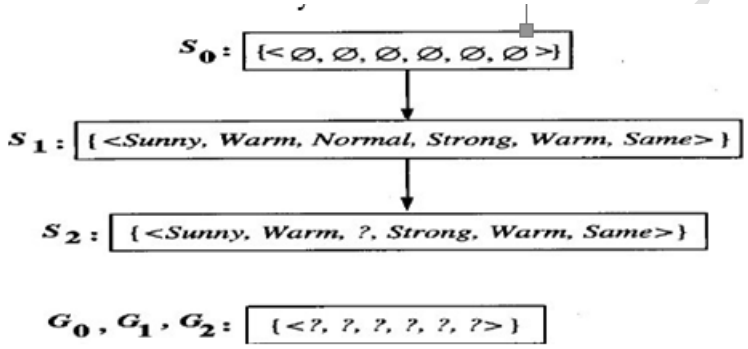
Add to G all minimal specializations h of g such that

- 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

AN ILLUSTRATIVE EXAMPLE

CANDIDATE-ELIMINATION Trace 1: S_o and G_o are the initial boundary sets corresponding to the most specific and most general hypotheses. Training examples 1 and 2force the S boundary to become more general, as in the FIND-S algorithm. They have no effect on the G boundary.



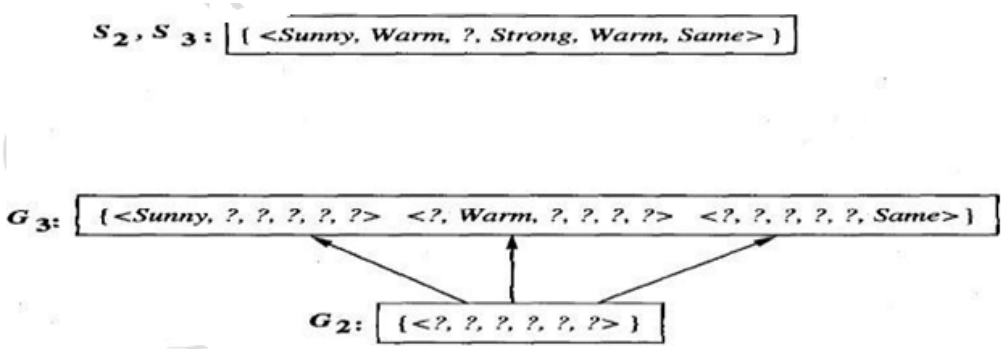
Training examples:

1 <Sunny, Warm, Normal, Strong, Warm, Same>, Enjoy Sport = Yes

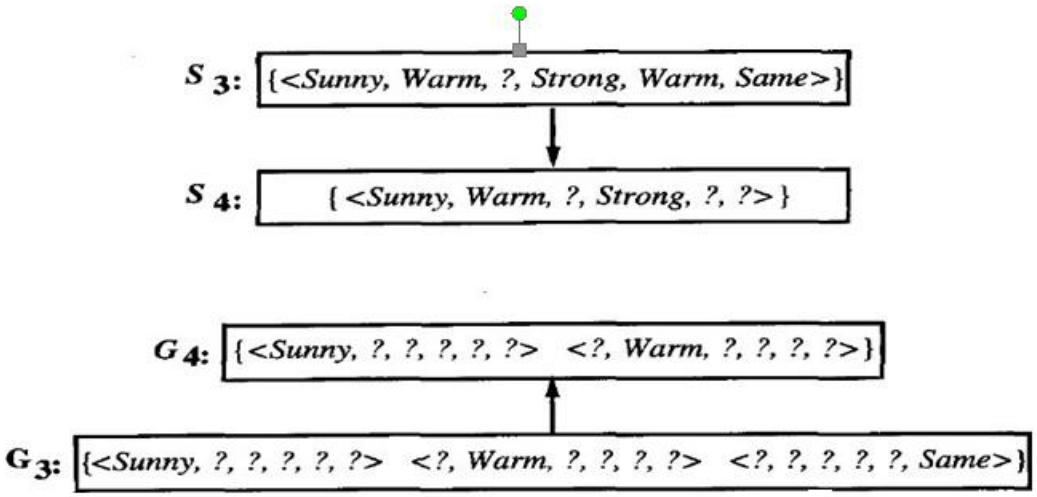
2 <Sunny, Warm, High, Strong, Warm, Same>, Enjoy Sport = YesTraining Example:

3 <Rainy, Cold, High, Strong, Warm, Change>, Enjoy Sport = No

CANDIDATE-ELMN ATION Trace 2: Training example 3 is a negative example that forces the G2 boundary to be specialized to G3. Note several alternative maximally general hypotheses are included in G_3.



Training Example: 4. <Sunny, Warm, High, Strong, Cool, Change>, Enjoy Sport = Yes CANDIDATE-ELIMINATIO Trace 3: The positive training example generalizes the S boundary, from S3 to S4. O ne member of G3 must also be deleted, because it is no longermore general than the S4 boundary.



2.7 INDUCTIVE BIAS

The CANDIDATE-ELIMINATION Algorithm will converge toward the true target concept provided it is given accurate training examples and provided its initial hypothesis space contains the target concept.

2.7.1 A Biased Hypothesis Space

• Suppose we wish to assure that the hypothesis space contains the unknown target concept.The obvious solution is to enrich the hypothesis space to include every possible hypothesis.

• To illustrate, consider again the EnjoySport example in which we restricted the hypothesisspace to include only conjunctions of attribute values. Because of this restriction, the

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

hypothesis space is unable to represent even simple disjunctive target concepts such as "Sky

= Sunny or Sky = Cloudy." In fact, given the following three training examples of this disjunctive hypothesis, our algorithm would find that there are zero hypotheses in the version space.

• To see why there are no hypotheses consistent with these three examples, note that the most specific hypothesis consistent with the first two examples and representable in the given hypothesis space H is

S2 : (?, Warm, Normal, Strong, Cool, Change)

This hypothesis, although it is the maximally specific hypothesis from H that is consistent with the first two examples, is already overly general: it incorrectly covers the third (negative) training example.

• The problem is that we have biased the learner to consider only conjunctive hypotheses. In this case we require a more expressive hypothesis space.

An UN-Biased Learner

• Idea: Choose H that expresses every teachable concept (i.e., H is the power set of X)

• Consider = disjunctions, conjunctions, negations over previous H.

E.g.,

• $\langle Sunny\ Warm\ Normal\ ?\ ?\ ?\rangle \vee \neg\langle ?\ ?\ ?\ ?\ ?\ Change\rangle$

• To see why, suppose we present three positive examples (xl,x2, x3)and two negative examples(x4, x5) to the learner. At this point, the S boundary of the version space will contain the hypothesis which is just the disjunction of the positive examples because this is the most specific possible hypothesis that covers these three examples.

$$S : \{(x_1 \vee x_2 \vee x_3)\}$$

Similarly, the G boundary will consist of the hypothesis that rules out only the observed negative examples.

$$G : \{\neg(x_4 \vee x_5)\}$$

• The problem here is that with this very expressive hypothesis representation, the S boundary will always be simply the disjunction of the observed positive examples, while the G boundary will always be the negated disjunction of the observed negative examples.

• Therefore, the only examples that will be unambiguously classified by S and G are the observed training examples themselves. In order to converge to a single, final target concept, we will have to present every single instance in X as a training example.

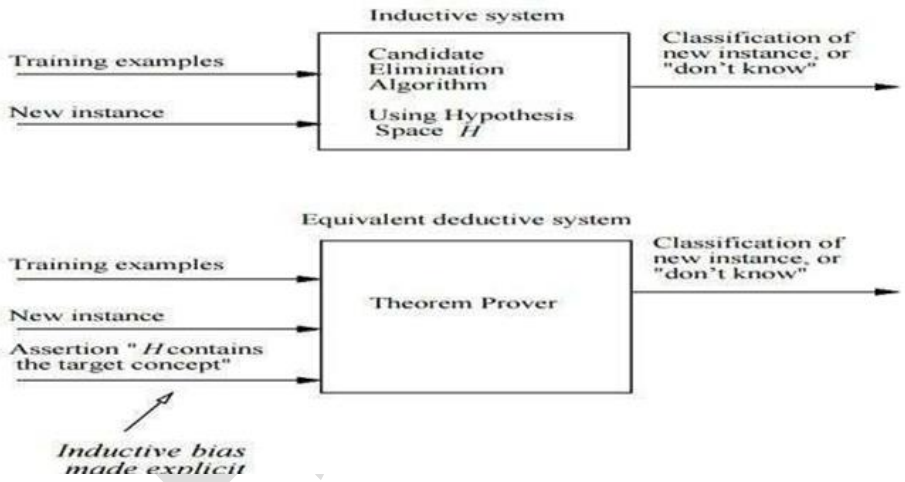
Definition: Consider a concept learning algorithm L for the set of instances X. Let c be an arbitrary concept defined over X, and let $D_c = \{\langle x, c(x)\rangle\}$ be an arbitrary set of training examples of c. Let $L(x_i, D_c)$ denote the classification assigned to the instance x_i by L after training 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

$$(\forall x_i \in X)[(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)] \quad (2.1)$$

Inductive Systems and Equivalent Deductive Systems

• Modeling 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 assertio n "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 modeling them by their equivalent deductive systems. This provides away to compare inductive systems according to their policies for generalizing beyond the observed training data.



• One advantage of viewing inductive inference systems in terms of their inductive bias is that it provides a nonprocedural means of characterizing their policy for generalizing beyond the observed data.

• A second advantage is that it allows comparison of different learners according to the strength of the inductive bias they employ. Consider, for example, the following three learning algorithms, which are listed from weakest to strongest bias.

• ROTE-LEARNER: Learning corresponds simply to storing each observed training example in memory. Subsequent instances are classified by looking them up in memory. If the instance is found in memory, the stored classification is returned. Otherwise, the system refuses to classify the new instance.

• CANDIDATE-ELIMINATION ALGORITHM: New instances are classified only in the case where all members of the current version space agree on the classification. Otherwise, the system refuses to classify the new instance.

• FIND-S: This algorithm, described earlier, finds the most specific hypothesis consistent with the training examples. It then uses this hypothesis to classify all subsequent instances.

• The ROTE-LEARNER has no inductive bias. The classifications it provides for new instances follow deductively from the observed training examples, with no additional assumptions required.

• The CANDIDATE-ELIMINATION ALGORITHM has a stronger inductive bias: that the target concept can be represented in its hypothesis space. Because it has a stronger bias, it will classify some instances that the ROTE-LEARNER will not.

• Of course the correctness of such classifications will depend completely on the correctness of this inductive bias. The FIND-S algorithm has an even stronger inductive bias. In addition to the assumption that the target concept can be described in its hypothesis space, it has an additional inductive bias assumption.

Module3

DECISION TREE LEARNING

3.1 INTRODUCTION

- Decision tree learning is one of the most widely used and practical methods for inductive inference.
- Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability.

3.2 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.

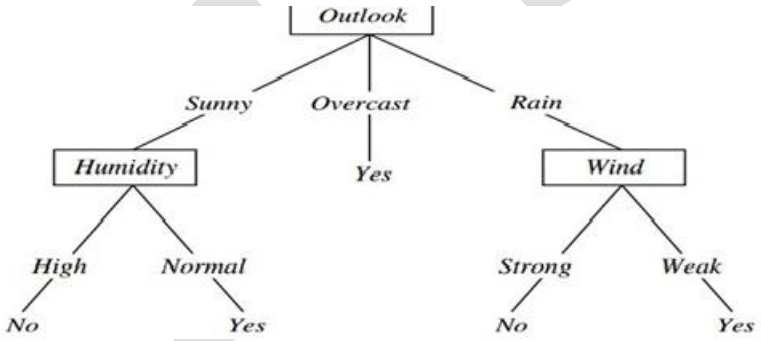


FIGURE 3.1

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 (in this case, *Yes* or *No).*

This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis. For example, the instance would be sorted down the left-most branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that ***PlayTennis*** = ***no).***

(Outlook = ***Sunny, Temperature*** = ***Hot, Humidity*** = ***High, Wind*** = ***Strong)*** In general, 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 Figure 3.1 corresponds to the expression

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

3.3 APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

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

- ***Instances are represented by attribute-value pairs.*** Instances are described by a fixed set of attributes (e.g., ***Temperature)*** and their values (e.g., ***Hot).*** The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., ***Hot, Mild, and Cold). Decision tree a***llow handling real-valued attributes as well (e.g., representing ***Temperature*** numerically).
- ***The target function has discrete output values.*** The decision assigns a boolean classification (e.g., ***yes*** or ***no)*** to each example. Decision tree methods easily extend to learning functions with more than two possible output values. A more substantial extension allows learning target functions with

real-valued outputs, though the application of decision trees in this setting is less common.

- ***Disjunctive descriptions may be required.*** As noted above, decision trees naturally represent disjunctive expressions.
- ***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.
- ***The training data may contain missing attribute values.*** Decision tree methods can be used even when some training examples have unknown values (e.g., if the ***Humidity*** of the day is known for only some of the training examples).

3.3 THE BASIC DECISION TREE LEARNING ALGORITHM

- Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees.
- The basic algorithm for decision tree learning, corresponding approximately to the ID3 algorithm.

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. We would like to select the attribute that is most useful for classifying examples.
- We will define a statistical property, called ***information gain that*** measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among the candidate attributes at each step while growing the tree.

ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

- In order to define information gain precisely, we begin by defining a measure commonly used in information theory, called ***entropy,*** that characterizes the (im)purity of an

arbitrary 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 the proportion of positive examples in S and, is the proportion of negative examples in S.
- In all calculations involving entropy we define 0 log 0 to be 0.

ID3 ALGORITHM

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.

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
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have value v_i for A
    - If $Examples_{v_i}$ 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($Examples_{v_i}$, *Target_attribute*, *Attributes* − {*A*}))
- End
- Return *Root*

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

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

- Notice that the entropy is 0 if all members of S belong to the

same class. For example, if all members are positive ($p_{\oplus} = 1$), then p is 0, and Entropy(S)=-1.$\log_2$1 -0.$\log_2$0= -1.0-0.$\log_2$ 0 = 0.

- Note the entropy is 1 when the collection contains an equal number of positive and negative examples.

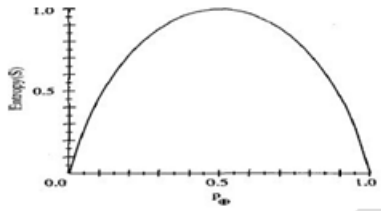


FIGURE 3.2
The entropy function relative to a boolean classification, as the proportion, $p_{\oplus}$, of positive examples varies between 0 and 1.

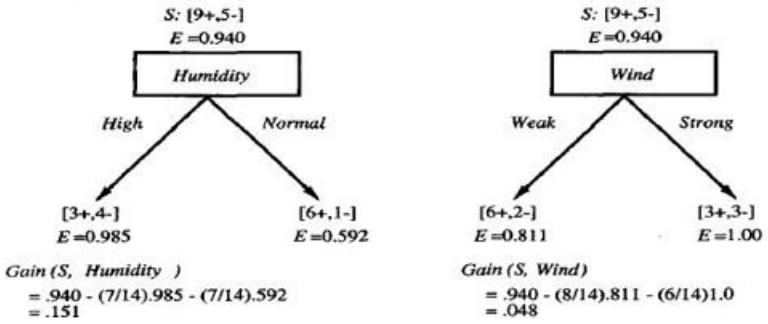
$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

- If the collection contains unequal numbers of positive and negative examples, the
- For example, suppose ***S*** is a collection of training-example days described by attributes including ***Wind,*** which can have the values ***Weak*** or ***Strong.***
- As before, assume ***S*** is a collection containing ***14*** examples, [9+, 5-]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have ***Wind*** = ***Weak,*** and the

remainder have ***Wind*** = ***Strong.***

- The information gain due to sorting the original ***14*** examples by the attribute ***Wind*** may then be calculated as

$$
\begin{aligned}
Values(Wind) &= Weak, Strong \\
S &= [9+, 5-] \\
S_{Weak} &\leftarrow [6+, 2-] \\
S_{Strong} &\leftarrow [3+, 3-] \\
Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\
&= Entropy(S) - (8/14) Entropy(S_{Weak}) \\
&\quad - (6/14) Entropy(S_{Strong}) \\
&= 0.940 - (8/14)0.811 - (6/14)1.00 \\
&= 0.048
\end{aligned}
$$



Which attribute is the best classifier?

Figure 3.3

Humidity provides greater information gain than ***Wind,*** relative to the target classification. Here, E stands for entropy and ***S*** for the original

collection of examples. Given an initial collection ***S*** of 9 positive and 5 negative examples, **[9+,** 5-], sorting these by their ***Humidity*** produces collections of [3+, 4-] ***(Humidity*** = ***High)*** and **[6+,** 1-] ***(Humidity*** = ***Normal).*** The information gained by this partitioning is 0.151,

compared to a gain of only **.048** for the attribute ***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

Table 3.2: Training examples for the target concept ***PlayTennis***

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

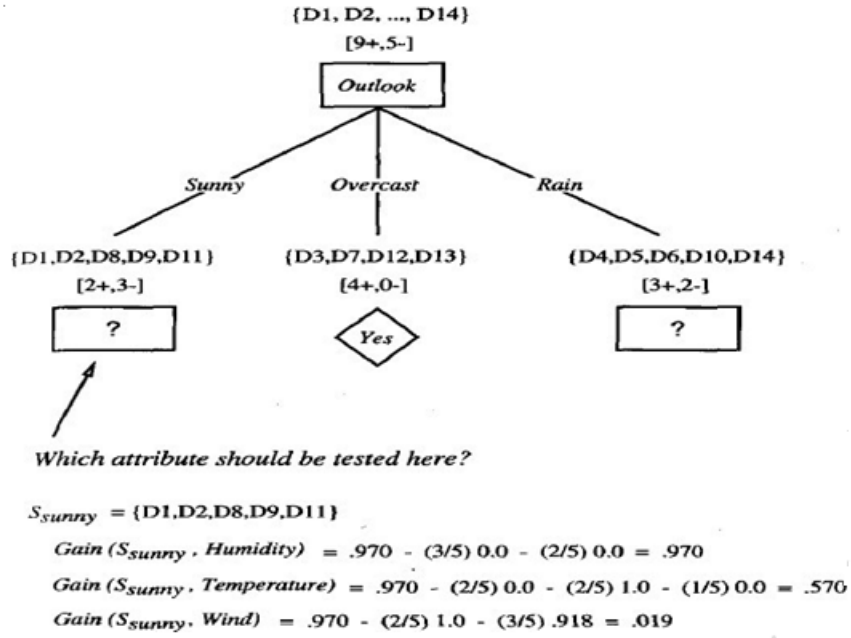


Fig 3.4: The partially learned decision tree resulting from the first step of ID3.

The training examples are sorted to the corresponding descendant nodes. The *Overcast* descendant has only positive examples and therefore becomes a leaf node with classification *Yes*. The other two nodes will be further expanded, by selecting the attribute with highest information gain relative to the new subsets of examples.

3.5 HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

- **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. Therefore, it is susceptible to the usual risks of hill-climbing search without

backtracking: converging to locally optimal solutions that are not globally optimal.

- Locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search.

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

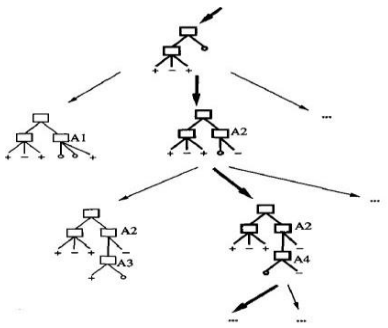


FIGURE 3.5
Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the *information gain heuristic.*

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

3.7 ISSUES IN DECISION TREE LEARNING

- Incorporating Continuous-Valued Attributes
- Alternative Measures for Selecting Attributes
- Handling Training Examples with Missing Attribute Values
- Handling Attributes with Differing Costs.
- Avoiding Over-fitting the Data
  - Reduced Error Pruning
  - Rule Post-Pruning

Incorporating Continuous-Valued Attributes

In particular, for an attribute A that is continuous-valued, the algorithm can dynamically create a new boolean attribute A, that is true if A < c and false otherwise. The only question is how to select the best value for the threshold c.

Suppose further that the training examples associated with a particular node in the decision tree have the following values for ***Temperature*** and the target attribute ***PlayTennis.***

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

Clearly, we would like to pick a threshold, ***c,*** that produces the greatest information gain. By sorting the examples according to the continuous attribute **A,** then identifying adjacent examples that differ in their target classification, we can generate a set of candidate thresholds midway between the corresponding values of ***A.***

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, $Temperature_{>54}$ $Temperature_{>85}$ the best can be selected ($Temperature_{>54}$)

Alternative Measures for Selecting Attributes

There is a natural bias in the information gain measure that favors attributes with many values over those with few values.

One way to avoid this difficulty is to select decision attributes based on some measure other than information gain. One alternative measure that has been used successfully is the gain ratio. The gain ratio measure a term, called split information, that is sensitive to how broadly and uniformly the attribute splits the data:

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

where S1 through S, are the ***c*** subsets of examples resulting from partitioning S by the c-valued attribute A.

The Gain Ratio measure is defined in terms of the earlier Gain measure, as well as this Split information, as follows

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

Handling Training Examples with Missing Attribute Values

In certain cases, the available data may be missing values for some attributes.

One strategy for dealing with the missing attribute value is to assign it the value that is most common among training examples at node ***n.***

Alternatively, we might assign it the most common value among examples at node ***n*** that have the classification ***c(x).***

A second, more complex procedure is to assign a probability to each of the possible values of ***A*** rather than simply assigning the most common value to ***A(x).*** These probabilities can be estimated again based on the observed frequencies of the various values for ***A*** among the examples at node ***n.***

Handling Attributes with Differing Costs.

In some learning tasks the instance attributes may have associated costs. For example, in learning to classify medical diseases we might describe patients in terms of attributes such **as** Temperature, BiopsyResult, Pulse, BloodTestResults, etc.

These attributes vary significantly in their costs, both in terms of monetary cost and cost to patient comfort. In such tasks, we would prefer decision trees that use low-cost attributes where possible, relying on high-cost attributes only when needed to produce reliable classifications.

More efficient recognition strategies are learned, without sacrificing classification accuracy, by replacing the information gain attribute selection measure by the following measure

Avoiding Over-fitting the Data

We will say that a hypothesis overfits the training examples if some other hypothesis that fits the training examples less well actually performs better over the entire distribution of instances (i.e., including instances beyond the training set).

> **Definition: Given a hypothesis space H, a hypothesis h ∈ H is said to over-fit 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.

There are several approaches to avoiding overfitting in decision tree learning. These can be grouped into two classes:

- Approaches that stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data,
- Approaches that allow the tree to over-fit the data, and then post-prune the tree.

Regardless of whether the correct tree size is found by stopping early or by post- pruning, a key question is what criterion is to be used to determine the correct final tree size. Approaches include:

- 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 an explicit measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized.

REDUCED ERROR PRUNING

Pruning a decision node consists of removing the sub tree 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.

RULE POST-PRUNING

Rule post-pruning involves the following steps:

1. Infer the decision tree from the training set, growing the tree until the trainingdata is fit as well as possible and allowing overfitting to occur.

2. Convert the learned tree into an equivalent set of rules by creating one rulefor each path from the root node to a leaf node.

3. Prune (generalize) each rule by removing any preconditions that result inimproving its estimated accuracy.

4. Sort the pruned rules by their estimated accuracy, and consider them in thissequence when classifying subsequent instances.

MODULE - 3

CHAPTER 4

INTRODUCTION

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

Historically, two groups of researchers have worked with artificial neural networks. One group has been motivated by the goal of using ANNs to study and model biological learning processes. A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes.

NEURAL NETWORK REPRESENTATIONS

A prototypical example of ANN learning is provided by Pomerleau's (1993) 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 30 x 32 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.

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. ~t is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks. It is appropriate for problems with the following characteristics:

- ***Instances are represented by many attribute-value pairs.*** The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- ***The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.*** For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering

command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.

- ***The training examples may contain errors.*** ANN learning methods are quite robust to noise in the training data.
- ***Long training times are acceptable.*** Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- ***Fast evaluation of the learned target function may be required.*** Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
- ***The ability of humans to understand the learned target function is not important.*** The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

PERCEPTRONS

One type of ANN system is based on a unit called a perceptron, illustrated in Figure 4.2. 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. More precisely, given inputs x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

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

where

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

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$.

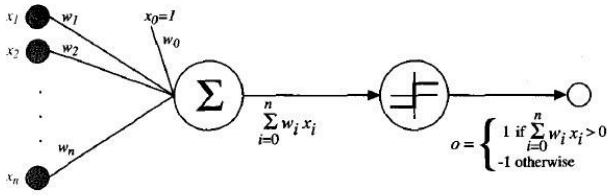


FIGURE 4.2
A perceptron.

Representational Power of Perceptrons

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights w_o = -.8, and w_1 = w_2 = .5. This perceptron can be made to represent the OR function instead by altering the threshold to w_o = -.3.

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND, and NOR. Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function.

The Perceptron Training Rule

Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct output for each of the given training examples. Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule.

One way to learn an acceptable weight vector is to 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

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

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

Here ***t*** is the target output for the current training example, ***o*** is the output generated by the perceptron, and n is a positive constant called the ***learning rate.*** The role of the learning rate is to moderate the degrees to which weights are changed at each step. It is usually set to some small value (e.g., 0.1)

To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the perceptron. In this case, ***(t - o)*** is zero, making Δw_i zero, so that no weights are updated.

Suppose the perceptron outputs a -1, when the target output is +1. To make the perceptron output a +1 instead of -1 in this case, the weights must be altered to increase the value of $\vec{w} \cdot \vec{x}$.

For example, if $x_i = .8$, $n = 0.1$, $t = 1$, and $o = -1$, then the weight update will be $\Delta w_i = n(t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$. On the other hand, if $t = -1$ and $o = 1$, then weights associated with positive x_i will be decreased rather than increased.

Gradient Descent and the Delta Rule

A second training rule, called the ***delta rule,*** is designed to overcome this difficulty. 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.

The delta training rule is best understood by considering the task of training an ***unthresholded*** perceptron; that is, a ***linear unit*** for which the output ***o*** is given by

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is:

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

where D is the set of training examples, t_d is the target output for training example d, and o_d is the output of the linear unit for training example d. By this definition, is simply half the squared difference between the target output t_d and the hear unit output o_d, summed over all training examples.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

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 (e.g., the weights in a linear unit), and

(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 (i.e., it can require many thousands of gradient descent steps), and

(2) If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

GRADIENT-DESCENT(*training_examples*, η)

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.* η *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 Δw_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

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \quad \text{(T4.1)}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \quad \text{(T4.2)}$$

TABLE 4.1
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

One common variation on gradient descent intended to alleviate these difficulties is called ***incremental gradient descent,*** or alternatively ***stochastic gradient descent.*** Whereas the gradient descent training rule presented in Equation ***(4.7)*** computes weight updates after summing over ***a22*** 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. The modified training rule is like the training rule given by Equation ***(4.7)*** except that ***as*** we iterate through each training example we update the weight according to

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.

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

A typical multilayer network and decision surface is depicted in Figure **4.5.** Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.). The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space.

A Differentiable Threshold Unit

What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a continuous function of its input. More precisely, the sigmoid unit computes its output ***o*** .

The BACKPROPAGATION Algorithm

The BACKPROPAGATaIlOgNor ithm 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.

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \qquad (4.13)$$

where ***outputs*** is the set of output units in the network, and t_{kd} and O_{kd} are the target and output values associated with the kth output unit and training example d.

The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by **all** possible weight values for all the units in the network.

The BACKPROPAGATaIlgOoNri thm is presented in Table 4.2. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} 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_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \qquad (T4.3)$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \qquad (T4.4)$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \qquad (T4.5)$$

TABLE 4.2

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- x_{ji} denotes the input from node i to unit j, and w_{ji} denotes the corresponding weight.
- δ_n denotes the error term associated with unit n. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Notice the algorithm in Table 4.2 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Giventhis fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

ADDING MOMENTUM

Because **BACKPROPAGATisIO** suNc h a widely used algorithm, many variations have been developed. Perhaps the most common is to alter the weight-update rule in Equation (T4.5) in the algorithm by making the weight update on the nth iteration depend partially on the update that occurred during the (n - 1)th iteration, as follows:

Notice the first term on the right of this equation is just the weight-update rule of Equation (T4.5) in the **BACKPROPAGATION Algorithm**.

The second term on the right is new and is called the momentum term. To see the effect ofthis momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of α is to add momentum that tends to keep the ball rolling in the same direction from one iteration to the next.

This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum.

It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence.

EVALUATING HYPOTHESIS

Chapter 5

Empirically evaluating the accuracy of hypotheses is fundamental to machine learning. Statistical methods for estimating hypothesis accuracy, focuses on three questions. First, given the observed accuracy of a hypothesis over a limited sample of data, how well does this estimate its accuracy over additional examples? Second, given that one hypothesis outperforms another over some sample of data, how probable is it that this hypothesis is more accurate in general? Third, when data is limited what is the best way to use this data to both learn a hypothesis and estimate its accuracy?

MOTIVATION

Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:

Bias in the estimate. First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. To obtain an unbiased estimate of future accuracy, we typically test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis

Variance in the estimate. Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

ESTIMATING HYPOTHESIS ACCURACY

Consider the following setting for the learning problem. There is some space of possible instances X, over which various target functions may be defined. We assume that different instances in X may be encountered with different frequencies. A convenient way to model this is to assume there is some unknown probability distribution ***D*** that defines the probability of encountering each instance in X.

Sample Error and True Error

Sample Error and True Error are the two notions of accuracy or, equivalently, error. One is the error rate of the hypothesis over the sample of data that is available.

The ***sample error*** of a hypothesis with respect to some sample *S* of instances drawn from X is the fraction of *S* that it misclassifies:

Definition: The **sample error** (denoted **$error_s(h)$**) of hypothesis **h** with respect to target function f and data sample **S** is

$$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

Where ***n*** is the number of examples in **S,** and the quantity δ(f ***(x), h(x))*** is ***1*** if f ***(x)*** ≠ ***h(x),*** and ***0*** otherwise.

The ***true error*** of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution ***D.***

Definition: The **true error** (denoted **$error_D(h)$**) of hypothesis **h** with respect to target function f and distribution **D,** is the probability that **h** will misclassify **an** instance drawn at random according to **D.**

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

What we usually wish to know is the true error ***$error_D(h)$*** of the hypothesis, because this is the error we can expect when applying the hypothesis to future examples. All we can measure, however, is the sample error ***$error_s(h)$*** of the hypothesis for the data sample ***S*** that we happen to have in hand. How good an estimate of ***errorD(h)*** is provided by ***errors (h)?***

Confidence Intervals for Discrete-Valued Hypotheses

More specifically, suppose we wish to estimate the true error for some discrete valued hypothesis ***h,*** based on its observed sample error over a sample S, where

- The sample ***S*** contains ***n*** examples drawn independent of one another, and independent of ***h,*** according to the probability distribution D
- ***n ≥ 30***
- Hypothesis ***h*** commits ***r*** errors over these ***n*** examples (i.e., ***$error_s(h)$*** = ***r|n).***

Under these conditions, statistical theory allows us to make the following assertions:

1. Given no other information, the most probable value of ***$error_D(h)$*** is ***$error_s(h)$***

2. With approximately ***95%*** probability, the true error ***$error_D(h)$*** lies in the interval

$$error_S(h) \pm 1.96\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

To illustrate, suppose the data sample S contains n = 40 examples and that hypothesis h commits r = 12 errors over this data. In this case, the sample error $error_s(h)$ = 12/40 = .30. Given no other information, the best estimate of the true error $error_D(h)$ is the observed

sample error .30. For the given confidence interval of 95% the true error will be ***0.30*** ± ***(1.96 - .07)*** = ***0.30*** ± .14.

The above expression for the ***95%*** confidence interval can be generalized to any desired confidence level. The constant ***1.96*** is used in case we desire a ***95%*** confidence interval. ***A*** different constant, Z_N, is used to calculate the ***N%*** confidence interval. The general expression for approximate ***N%*** confidence intervals for ***error***$_V$***(h)*** is

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \qquad (5.1)$$

Where the constant Z_N is chosen depending on the desired confidence level, using the values of Z_N given in Table *5.1*.

Confidence level N%:	50%	68%	80%	90%	95%	98%	99%
Constant z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

TABLE 5.1
Values of z_N for two-sided N% confidence intervals.

INSTANCE BASED LEARNING

Chapter 8

Instance-based learning methods such as nearest neighbour and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions.

k - NEAREST NEIGHBOUR LEARNING

The most basic instance-based method is the k-NEARESNT NEIGHBOUR algorithm. The nearest neighbours of an instance are defined in terms of the standard Euclidean distance.

More precisely, let an arbitrary instance ***x*** be described by the feature vector

$$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle$$

where $a_r(x)$ denotes the value of the rth 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-neighbour learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions. If we choose k = 1, then the 1-NEAREST NEIGHBOUR algorithm assigns to $\hat{f}(x_q)$ the value $f(x_i)$ where x_i is the training

instance nearest to x_q. For larger values of k, the algorithm assigns the most common value among the k nearest training examples. The k-NEARESNT NEIGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1.

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

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

TABLE 8.1
The k-NEAREST NEIGHBOR algorithm for approximating a discrete-valued function $f : \Re^n \rightarrow V$.

The k-NEAREST NEIGHBOR algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have the algorithm calculate the mean value of the k nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function we replace the final line of the above algorithm by the line

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k} \qquad (8.1)$$

Distance-Weighted NEAREST NEIGHBOR Algorithm

One obvious 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 algorithm of Table 8.1, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q. This can be accomplished by replacing the final line of the algorithm by

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i)) \qquad (8.2)$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2} \qquad (8.3)$$

We can distance-weight the instances for real-valued target functions in a similar fashion, replacing the final line of the algorithm in this case by

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \qquad (8.4)$$

where w_i is as defined in Equation (8.3).

Note all of the above variants of the k-NEAREST NEIGHBOR algorithm consider only the k nearest neighbours to classify the query point.

The only disadvantage of considering all examples is that our classifier will run more slowly. If all training examples are considered when classifying a new query instance, we call the algorithm a *global* method. If only the nearest training examples are considered, we call it a *local* method. When the rule in Equation (8.4) is applied as a global method, using all training examples, it is known as Shepard's method

Remarks on k-NEAREST NEIGHBOR algorithm

One practical issue in applying k-NEAREST NEIGHBOR algorithms is that the distance between instances is calculated based on *all* attributes of the instance. This lies in contrast to methods such as rule and decision tree learning systems that select only a subset of the instance attributes when forming the hypothesis.

- To see the effect of this policy, consider applying k-NEAREST NEIGHBOR algorithm to a problem in which each instance is described by 20 attributes, but where only 2 of these attributes are relevant to determining the classification for the particular target function.
- In this case, instances that have identical values for the 2 relevant attributes may nevertheless be distant from one another in the 20-dimensional instance space.
- As a result, the similarity metric used by k-NEAREST NEIGHBOR algorithm depending on all 20 attributes-will be misleading. The distance between neighbors will be dominated by the large number of irrelevant attributes.
- This difficulty, which arises when many irrelevant attributes are present, is sometimes referred to as the ***curse of dimensionality.*** Nearest-neighbor approaches are especially sensitive to this problem.

One interesting approach to overcoming this problem is to weight each attribute differently when calculating the distance between two instances.

An even more drastic alternative is to completely eliminate the least relevant attributes from the instance space.

One additional practical issue in applying k-NEAREST NEIGHBOR is efficient memory indexing. Because this algorithm delays all processing until a new query is received, significant computation can be required to process each new query. Various methods have been developed for indexing the stored training examples so that the nearest neighbours can

be identified more efficiently at some additional cost in memory. One such indexing method is the kd-tree.

Note: For example problem on k nearest neighbour algorithm refer class notes.

A Note on Terminology

Much of the literature on nearest-neighbour methods and weighted local regression uses a terminology that has arisen from the field of statistical pattern recognition. In reading that literature, it is useful to know the following terms:

- ***Regression*** means approximating a real-valued target function.
- ***Residual*** is the error $\hat{f}(x) - f(x)$ in approximating the target function.
- ***Kernel function*** is the function of distance that is used 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 nearest-neighbour approaches can be thought of as approximating the target function $f(x)$ at the single query point $x = x_q$.
- Locally weighted regression is a generalization of this approach. It constructs an explicit approximation to f over a local region surrounding x_q.
- Given a new query instance x_q, the general approach in locally weighted regression is to construct an approximation $\hat{f}$ that fits the training examples in the neighborhood surrounding x_q.
- This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance. The description of $\hat{f}$ may then be deleted, because a different local approximation will be calculated for each distinct query instance.

Locally Weighted Linear Regression

Let us consider the case of locally weighted regression in which the target function f is approximated near x, using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

Procedure to derive local approximation.

The simple way is to redefine the error criterion E to emphasize fitting the local training examples. Three possible criteria are given below. Note we write the error $E(x_q)$ to emphasize the fact that now the error is being defined as a function of the query point x_q.

1. Minimize the squared error over just the k nearest neighbours:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$

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.

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

Remarks on Locally Weighted Regression

We considered using a linear function to approximate f in the neighbourhood of the query instance x_q. In most cases, the target function is approximated by a constant, linear, or quadratic function. More complex functional forms are not often found because

(1) The cost of fitting more complex functions for each query instance is prohibitively high,

(2) These simple approximations model the target function quite well over a sufficiently small sub region of the instance space.

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

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \qquad (8.8)$$

where each x_u is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance $d(x_u, x)$ increases. Here k is a user provided constant that specifies the number of kernel functions to be included.

Even though $\hat{f}(x)$ is 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. It is common to choose each function $K_u(d(x_u, x))$ to be a Gaussian function centered at the point x_u with some variance σ^2_u.

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

An example radial basis function (RBF) network is illustrated in Figure 8.2. Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ^2_u that define its kernel function $K_u(d(x_u, x))$. Second, the weights w_u are trained to maximize the fit of the network to the training data.

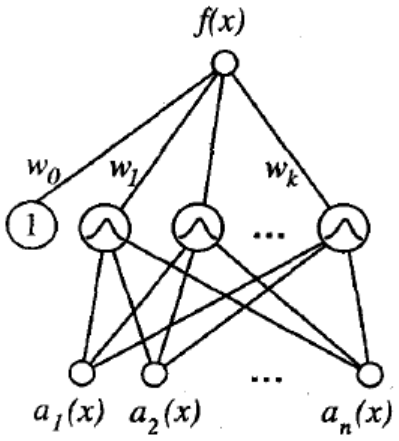


FIGURE 8.2
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u. Therefore, its activation will be close to zero unless the input x is near x_u. The output unit produces a linear combination of the hidden unit activations. Although the network shown here has just one output, multiple output units can also be included.

Methods 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 σ^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 neighbourhood of x_i. One advantage of this choice of kernel functions is that it allows the RBF network to fit the training data exactly.

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. The set of kernel functions may be distributed with centres spaced uniformly throughout the instance space X.

CASE-BASED REASONING

Instance-based methods such as k-NEARESTN NEIGHBOUR and locally weighted regression share three key properties.

1. They are lazy learning methods in that they defer the decision of how to generalize beyond the training data until a new query instance is observed.

2. They classify new query instances by analyzing similar instances while ignoring instances that are very different from the query.

3. They represent instances **as** real-valued points in an n-dimensional Euclidean space.

Case-based reasoning (CBR) is a learning paradigm based on the first two of these principles, but not the third.

- In CBR, instances are typically represented using more rich symbolic descriptions, and the methods used to retrieve similar instances are correspondingly more elaborate.

Let us consider a prototypical example of a case-based reasoning system "CADET system".

- 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 instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function.
- New design problems are then presented by specifying the desired function and requesting the corresponding structure.
- This problem setting is illustrated in Figure 8.3. The top half of the figure shows the description of a typical stored case called a T-junction pipe. Its 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 at its right, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail.
- For example, the output water flow Q_3 increases with increasing input water flow Q_1. Similarly a "-" label indicates that the variable at the head decreases with the variable at the tail.
- The bottom half of this figure depicts a new design problem described by its desired function. This particular function describes the required behavior of one type of water faucet.
- 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.
- Similarly, 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 water flow.
- Given this functional specification for the new design problem, **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. In Figure 8.3, for example, the T-junction function matches a sub graph of the water faucet function graph.

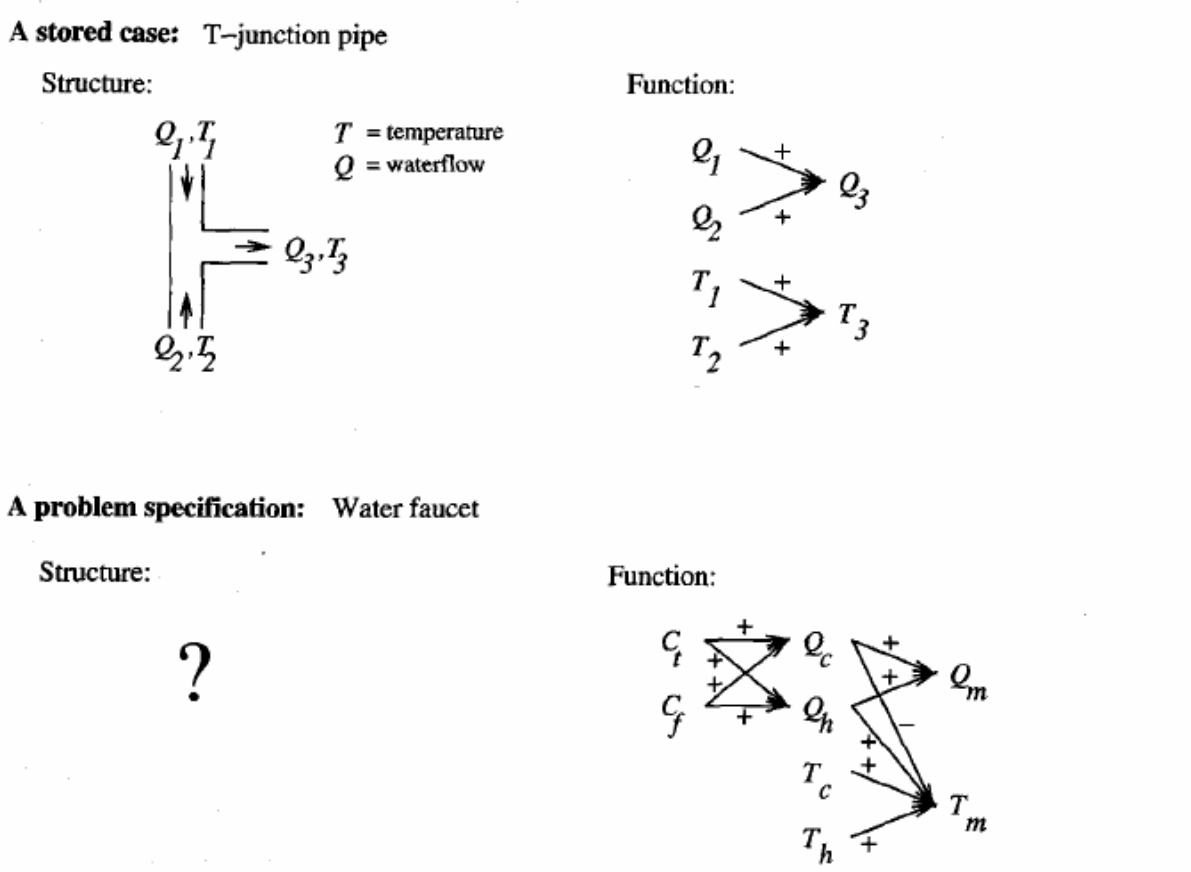


FIGURE 8.3
A stored case and a new problem. The top half of the figure describes a typical design fragment in the case library of CADET. The function is represented by the graph of qualitative dependencies among the T-junction variables (described in the text). The bottom half of the figure shows a typical design problem.

Generic properties of case-based reasoning systems that distinguish them from approaches such as k-NEAREST NEIGHBOUR:

- Instances or cases may be represented by rich symbolic descriptions, such as the function graphs used in CADET. This may require a similarity metric different from Euclidean distance, such as the size of the largest shared sub graph between two function graphs.

- Multiple retrieved cases may be combined to form the solution to the newproblem. This is similar to the k-NEAREST NEIGHBOUR approach, in that multiple similar cases are used to construct a response for the new query.
- There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving.

REINFORCEMENT LEARNING
Chapter 13

INTRODUCTION

Consider building a learning robot. The robot, or *agent*, has a set of sensors (such as a camera and sonars) to observe the *state* of its environment, and a set of *actions* (such as "move forward" and "turn") it can perform to alter this state. Its task is to learn a control strategy, or policy, for choosing actions that achieve its goals. For example, the robot may have a goal of docking onto its battery charger whenever its battery level is low.

We assume that the goals of the agent can be defined by a reward function that assigns a numerical value-an immediate payoff-to each distinct action the agent may take from each distinct state. The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent. This general setting for robot learning is summarized in Figure 13.1.

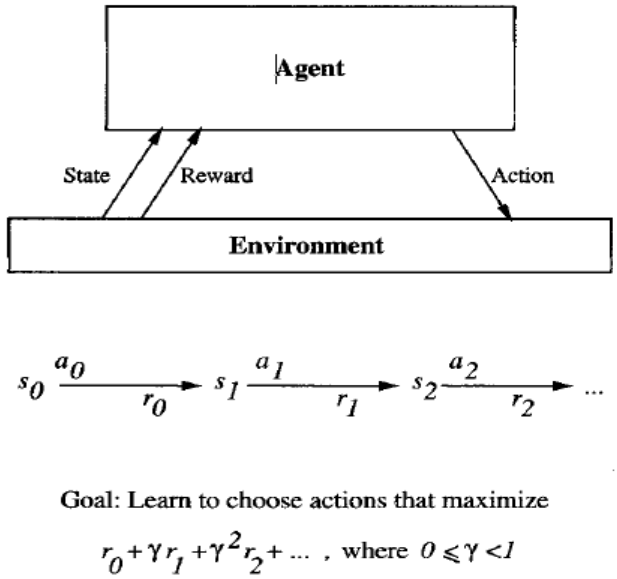


FIGURE 13.1
An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. It can perform any of a set of possible actions A. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the 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, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

Reinforcement learning problem differs from other function approximation tasks in several important respects.

1. Delayed reward. The task of the agent is to learn a target function π that maps from the current state ***s*** to the optimal action **a** = **π(s). *E***arlier we have always assumed that when learning some target function such as **π,** each training example would be a pair of the form **(s, π(s)).** In reinforcement learning, however, training information is not available in this form. 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 tradeoff in choosing whether to favor ***exploration*** of unknown states and actions (to gather new information), or ***exploitation*** of states and actions that it has already learned will yield high reward (to maximize its cumulative reward).

3. Partially observable states. Although it is convenient to assume that 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.

4. Life-long learning. Unlike isolated function approximation tasks, robot learning often requires that the robot 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

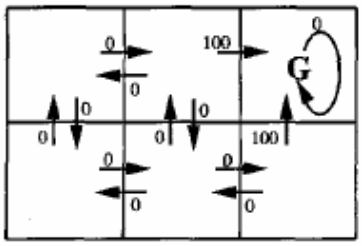
Consider a simple grid-world environment is depicted in the topmost diagram of Figure 13.2. The six grid squares in this diagram represent six possible states, or locations, for the agent. Each arrow in the diagram represents a possible action the agent can take to move from one state to another. The number associated with each arrow represents the immediate reward r(s, a) the agent receives if it executes the corresponding state-action transition. Note in this particular environment, the only action available to the agent once it enters the state G is to remain in this state. For this reason, we call G an absorbing state.

Once the states, actions, and immediate rewards are defined, and once we choose a value for the discount factor γ, we can determine the optimal policy π* and its value function V*(s). In

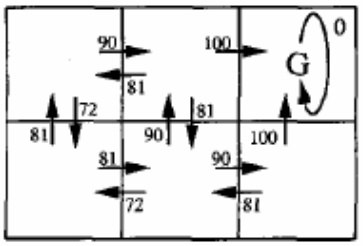
this case, let us choose $\gamma = 0.9$. The diagram at the bottom of the figure shows one optimal policy for this setting (there are others as well). The optimal policy directs the agent along the shortest path toward the state G.

The diagram at the right of Figure 13.2 shows the values of V* for each state. For example, consider the bottom right state in this diagram. The value of V* for this state is 100 because the optimal policy in this state selects the "move up" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards. Similarly, the value of V* for the bottom centre state is 90. This is because the optimal policy will move the agent from this state to the right (generating an immediate reward of zero), then upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom centre state is

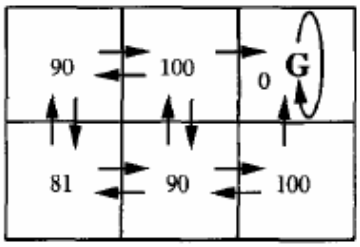
$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \ldots = 90$$



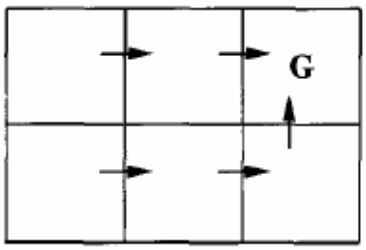
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

FIGURE 13.2
A simple deterministic world to illustrate the basic concepts of Q-learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s, a)$ gives reward 100 for actions entering the goal state G, and zero otherwise. Values of $V^*(s)$ and $Q(s, a)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal Q values, is also shown.

Q LEARNING

How can an agent learn an optimal policy n* for an arbitrary environment? It is difficult to learn the function $\pi^*: S \rightarrow A$ directly, because the available training data does not provide training examples of the form *(s, a).* Instead, the only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2, \ldots$ .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.

The Q Function

Let us define the evaluation function *Q(s, a)* so that the value of *Q* is the reward received immediately upon executing action *a* from state *s*, plus the value (discounted by y) of following the optimal policy thereafter.

$$Q(s,a) - r(s,a) + \gamma V^*(\delta(s,a)) \qquad 13.4$$

To illustrate, Figure 13.2 shows the *Q* values for every state and action in the simple grid world. Notice that the *Q* value for each state-action transition equals the *r* value for this transition plus the *V** value for the resulting state discounted by y. Note also that the optimal policy shown in the figure corresponds to selecting actions with maximal *Q* values.

An Algorithm for Learning Q

Learning the *Q* function corresponds to learning the optimal policy. How can *Q* be learned? The key problem is finding a reliable way to estimate training values for *Q*, given only a sequence of immediate rewards *r* spread out over time. This can be accomplished through iterative approximation. To see how, notice the close relationship between *Q* and *V**,

$$V^*(s) = \max_{a'} Q(s, a')$$

which allows rewriting Equation (13.4) as

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \qquad (13.6)$$

This recursive definition of Q provides the basis for algorithms that iteratively approximate Q. Q^ to refer to the learner's estimate, or hypothesis, of the actual ***Q*** function. In this algorithm the learner represents its hypothesis Q^ by a large table with a separate entry for each state-action pair. The table entry for the pair (s, a) stores the value for Q^(s, a) the, learner's current hypothesis about the actual but unknown value Q(s, a). The table can be initially filled with random values. The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward r = r(s, a) and the new state s' = δ(s, a). It then updates the table entry for Q^(s, ***a)*** following each such transition, according to the rule:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \qquad (13.7)$$

The above Q learning algorithm for deterministic Markov decision processes is described more precisely in Table 13.1. Using this algorithm the agent's estimate Q^ converges in the limit to the actual Q function, provided the system can be modelled as a deterministic Markov decision process, the reward function r is bounded, and actions are chosen so that every state-action pair is visited infinitely often.

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:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

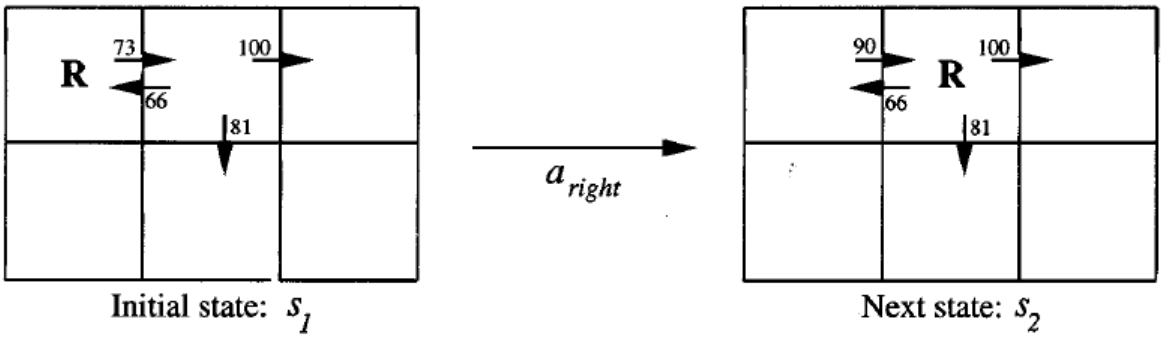
- $s \leftarrow s'$

TABLE 13.1
Q learning algorithm, assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$.

An Illustrative Example

To illustrate the operation of the ***Q*** learning algorithm, consider a single action taken by an agent, and the corresponding refinement to Q^ shown in Figure 13.3. In this example, the agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition. It then applies the training rule of Equation (13.7) to refine its estimate Q^ for the state-action transition it just executed. According to the training rule, the new Q^ estimate for this transition is the sum of the received reward (zero) and the highest Q^ value associated with the resulting state (l00), discounted by γ (.9).

Each time the agent moves forward from an old state to a new one, Q learning propagates Q^ estimates ***backward*** from the new state to the old. At the same time, the immediate reward received by the agent for the transition is used to augment these propagated values of Q^.



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$
$$\leftarrow 0 + 0.9 \max\{66, 81, 100\}$$
$$\leftarrow 90$$

FIGURE 13.3
The update to $\hat{Q}$ after executing a single action. The diagram on the left shows the initial state s_1 of the robot (**R**) and several relevant $\hat{Q}$ values in its initial hypothesis. For example, the value $\hat{Q}(s_1, a_{right}) = 72.9$, where a_{right} refers to the action that moves **R** to its right. When the robot executes the action a_{right}, it receives immediate reward $r = 0$ and transitions to state s_2. It then updates its estimate $\hat{Q}(s_1, a_{right})$ based on its $\hat{Q}$ estimates for the new state s_2. Here $\gamma = 0.9$.

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 our study of machine learning for two different reasons.

- 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.
- Bayesian methods are important to our study of machine learning is that they pro vide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

Features of Bayesian learning methods include:

- 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 (e.g., hypotheses such as "this pneumonia patient has a 93% chance of complete recovery").
- 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 whichother practical methods can be measured.

BAYES THEOREM

from some space H, given the observed training data D. One way to specify what we mean by the ***best*** hypothesis is to say that we demand the ***most probable*** hypothesis, given the data ***D*** plus any initial knowledge about the prior probabilities of the various hypotheses in H. Bayes theorem provides a direct method for calculating such probabilities. More precisely, 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.

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) is the prior probability of h*
- *P(D) to denote the prior probability that training data D will be observed (i.e.,the probability of D given no knowledge about which hypothesis holds).*
- *P(D|h) to denote the probability of observing data D given some world in which hypothesis h holds.*
- *P (h1 D) is called the posterior probability of h, because it reflects our confidence that h holds after we have seen the training data D.*

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*** (or at least one of the maximally probable if there are several).

Any such maximally probable hypothesis is ca lled a ***maximum*** a ***posteriori*** (MAP) hypothesis. We can determine the MAP hypotheses by using Bayes theorem to calculate the posterior probability of eachcandidate hypothesis.

More precisely, we will say that h_{MAP} is a MAP hypothesis provided:

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\operatorname{argmax}} P(h|D) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \end{aligned}$$

Notice in the final step above we dropped the term *P(D)* because it is a constant independent of *h.*

In some cases, we will assume that every hypothesis in H is equally probable a priori ($P(h_i) = P(h_j)$ for all h_i and h_j in H). In this case we can further simplify the above equation and need to only consider the term P(D|h) to find the most probable hypothesis. 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, h_{ML}.

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h)$$

An Example

To illustrate Bayes rule, consider a medical diagnosis problem in which there are two alternative hypotheses:

(1) That the patient; has a particular form of cancer. and

(2) That the patient does not cancer.

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 has this 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. Suppose we now observe a new patient for whom the lab test returns a positive result. Should we diagnose the patient as having cancer or not?

The above situation can be summarized by the following probabilities:

$$\begin{aligned} P(cancer) &= .008, & P(\neg cancer) &= .992 \\ P(\oplus|cancer) &= .98, & P(\ominus|cancer) &= .02 \\ P(\oplus|\neg cancer) &= .03, & P(\ominus|\neg cancer) &= .97 \end{aligned}$$

The maximum a posteriori hypothesis can be found using

$$\begin{aligned} P(\oplus|cancer)P(cancer) &= (.98).008 = .0078 \\ P(\oplus|\neg cancer)P(\neg cancer) &= (.03).992 = .0298 \end{aligned}$$

Thus, h_{MAP} = patient does not cancer.

The result of Bayesian inference depends strongly on the prior probabilities. also that in this example the hypotheses are not completely accepted or rejected, but rather become more or less probable as more data is observed.

BAYES THEOREM AND CONCEPT LEARNING

What is the relationship between Bayes theorem and the problem of concept learning? This section considers such a brute- force Bayesian concept learning algorithm, then compares it to concept learning algorithms.

Brute-Force Bayes Concept Learning

BRUTE-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

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(h|D)$$

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). Here let us choose them to be consistent with the following assumptions:*

1. *The training data D is noise free (i.e., $d_i = c(x_i)$).*
2. *The target concept c is contained in the hypothesis space H*
3. *We have no a priori reason to believe that any hypothesis is more probable than any other.*

Given these assumptions, what values should we specify for P(h).

According to the assumption 2 and 3

$$P(h) = \frac{1}{|H|} \quad \text{for all } h \text{ in } H$$

What choice shall we make for P(D|h)? 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,

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases} \qquad (6.4)$$

In other words, the probability of data D given hypothesis h is 1 if D is consistent with h, and 0 otherwise. Let us consider the first step of this algorithm, Recalling Bayes theorem, we have

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

First consider the case where h is inconsistent with the training data D. Since Equation (6.4) defines *P(D)h)* to be *0* when *h* is inconsistent with *D,* we have

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0 \text{ if } h \text{ is inconsistent with } D$$

Now consider the case where *h* is consistent with *D.* Since Equation *(6.4)* defines *P(D|h)* to be 1 when *h* is consistent with *D,* we have

$$P(h|D) = \frac{1 \cdot \frac{1}{|H|}}{P(D)}$$

$$= \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}}$$

$$= \frac{1}{|VS_{H,D}|} \text{ if } h \text{ is consistent with } D$$

where $VS_{H,D}$, is the subset of hypotheses from H that are consistent with *D.*

We can derive *P(D)* from the theorem of total probability and the fact that the hypotheses are mutually exclusive

$$P(D) = \sum_{h_i \in H} P(D|h_i) P(h_i)$$

$$= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} + \sum_{h_i \notin VS_{H,D}} 0 \cdot \frac{1}{|H|}$$

$$= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|}$$

$$= \frac{|VS_{H,D}|}{|H|}$$

To summarize, Bayes theorem implies that the posterior probability *P(h|D)* under our assumed *P(h)* and *P(D|h)* is

$$P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{if } h \text{ is consistent with } D \\ 0 & \text{otherwise} \end{cases} \qquad (6.5)$$

where $|VS_{H,D}|$ is the number of hypotheses from H consistent with ***D.*** The evolution of probabilities associated with hypotheses is depicted schematically in Figure *6.1.*

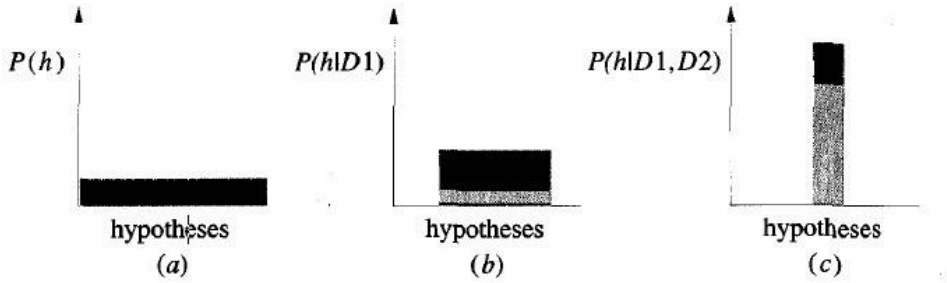


FIGURE 6.1
Evolution of posterior probabilities $P(h|D)$ with increasing training data. (*a*) Uniform priors assign equal probability to each hypothesis. As training data increases first to $D1$ (*b*), then to $D1 \wedge D2$ (*c*), the posterior probability of inconsistent hypotheses becomes zero, while posterior probabilities increase for hypotheses remaining in the version space.

MAP Hypotheses and Consistent Learners

A learning algorithm is a consistent learner provided it outputs a hypothesis that commits zero errors over the training examples. Given the above analysis, we can conclude that every consistent learner outputs a MAP hypothesis, if we assume a uniform prior probability distribution over H (i.e., $P(h_i) = P(h_j)$ for all i, j), and if we assume deterministic, noise free training data (i.e., P(D|h) = 1 if D and h are consistent, and 0 otherwise).

The Bayesian framework allows one way to characterize the behavior of learning algorithms (e.g., FIND-S), even when the learning algorithm does not explicitly manipulate probabilities. By identifying probability distributions ***P(h)*** and ***P(Dlh)*** under which the algorithm outputs optimal (i.e., MAP) hypotheses, we can characterize the implicit assumptions, under which this algorithm behaves optimally.

NAIVE BAYES CLASSIFIER

One highly practical Bayesian learning method is the naive Bayes learner, often called the naive Bayes classifier. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values (al, a2.. .a,). 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 $(a_1, a_2 \ldots a_n)$ that describe the instance.

$$v_{MAP} = \underset{v_j \in V}{\text{argmax}}\ P(v_j \mid a_1, a_2 \ldots a_n)$$

We can use Bayes theorem to rewrite this expression as

$$v_{MAP} = \underset{v_j \in V}{\text{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

$$= \underset{v_j \in V}{\text{argmax}}\ P(a_1, a_2 \ldots a_n | v_j) P(v_j) \qquad (6.19)$$

The assumption is that given the target value of the instance, the probability of observing the conjunction **al, a2.. .a,** is just the product of the probabilities for the individual attributes: $P(a_1, a_2 \ldots a_n \mid v_j) = \Pi_i\ P(a_i|v_j)$. Substituting this into Equation (6.19), we have the approach used by the naive Bayes classifier.

Naive Bayes classifier:

$$v_{NB} = \underset{v_j \in V}{\text{argmax}}\ P(v_j) \prod_i P(a_i | v_j) \qquad (6.20)$$

Where V_{NB} denotes the target value output by the naive Bayes classifier.

An Illustrative Example

Consider the dataset of 14 instances and 4 attributes that we have used in Decision tree learning module.

Here we use the naive Bayes classifier and the training data from this table to classify the following novel instance:

(Outlook = s unny, Te mpe rature = 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. Instantiating Equation (6.20) to fit the current task, the target value v_{NB} is given by

$$\begin{aligned} v_{NB} &= \underset{v_j \in \{yes,no\}}{\text{argmax}} P(v_j) \prod_i P(a_i|v_j) \\ &= \underset{v_j \in \{yes,no\}}{\text{argmax}} P(v_j) \quad P(Outlook = sunny|v_j) P(Temperature = cool|v_j) \\ &\qquad P(Humidity = high|v_j) P(Wind = strong|v_j) \end{aligned} \quad (6.21)$$

To calculate VNB we now require 10 probabilities that can be estimated from the training data. First, the probabilities of the different target values can easily be estimated based on their frequencies over the **14** training examples

$$P(PlayTennis = yes) = 9/14 = .64$$
$$P(PlayTennis = no) = 5/14 = .36$$

Similarly, we can estimate the conditio nal probabilities. For example, those for *Wind = trong* are

$$P(Wind = strong|PlayTennis = yes) = 3/9 = .33$$
$$P(Wind = strong|PlayTennis = no) = 3/5 = .60$$

Using these probability estimates and similar estimates for the remaining attribute values, we calculate ***VNB*** according to Equation ***(6.21)*** as follows

$$P(yes)\ P(sunny|yes)\ P(cool|yes)\ P(high|yes)\ P(strong|yes) = .0053$$
$$P(no)\ P(sunny|no)\ P(cool|no)\ P(high|no)\ P(strong|no) = .0206$$

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.

Estimating Probabilities

Up to this point 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, = 3 is the number of these for which Wind = strong. it provides poor estimates when $\mathbf{n_c}$ is very small. This raises two difficulties. First, n_c/n produces a biased underestimate of the probability. Second, when this probability estimate is zero, this probability term will dominate the Bayes classifier if the future query contains Wind = strong. 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:**

$$\frac{n_c + mp}{n + m} \qquad (6.22)$$

Here, *n*, and *n* are defined as before, 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.

BAYESIAN BELIEF N ETWORKS

- 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.
- In contrast to the naive Bayes classifier, which assumes that ***all*** the variables are conditionally independent given the value of the target variable, Bayesian belief networks allow stating conditional independence assumptions that apply to ***subsets*** of the variables.
- Thus, Bayesian belief networks provide an intermediate approach that is less constraining than the global assumption of conditional independence made by the naive Bayes classifier, but more tractable than avoiding conditional independence assumptions altogether.

Conditional Independence

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 $P(A_1, A_2 \mid V)$ in Equation (6.20) as follows

$$P(A_1, A_2|V) = P(A_1|A_2, V)P(A_2|V) \qquad (6.23)$$

$$= P(A_1|V)P(A_2|V) \qquad (6.24)$$

Equation (6.23) is just the general form of the product rule of probability. Equation (6.24) follows because if A_1 is conditionally independent of A_2 given V, then by our definition of conditional independence $P(A_1|A_2, V) = P(A_1|V)$.

Representation

A Bayesian belief network (Bayesian network for short) represents the joint probability distribution for a set of variables. In general, a Bayesian network represents the joint probability distribution by specifying a set of conditional independence assumptions (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. For each variable two types of information are specified. First, the network arcs represent the assertion that the variable is conditionally independent of its nondescendants in the network given its immediate predecessors in the network. Second, a conditional probability table 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 $(Y_1 \ldots Y_n)$ 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. To illustrate, the Bayesian network in Figure 6.3 represents the joint probability distribution over the boolean variables Storm, Lightning, Thunder, Forest Fire, Campfire, and BusTourGroup. Consider the node Campfire. The network nodes and arcs represent the assertion that Campfire is conditionally independent of its nondescendants Lightning and Thunder, given its immediate parents Storm and BusTourGroup. 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 right side of the figure shows the conditional probability table associated with the variable Campfire. The top left entry in this table, for example, expresses the assertion that

P(Campfire = True|Storm = True, BusTourGroup = True) = 0.4

this table provides only the conditional probabilities of *Campfire* given its parent variables *Storm* and *BusTourGroup*. The set of local conditional probability tables for all the variables, together with the set of conditional independence assumptions described by the network, describe the full joint probability distribution for the network.

One attractive feature of Bayesian belief networks is that they allow a convenient way to represent causal knowledge such as the fact that *Lightning* causes *Thunder*. In the terminology of conditional independence, we express this by stating that *Thunder* is conditionally independent of other variables in the network, given the value of *Lightning*.

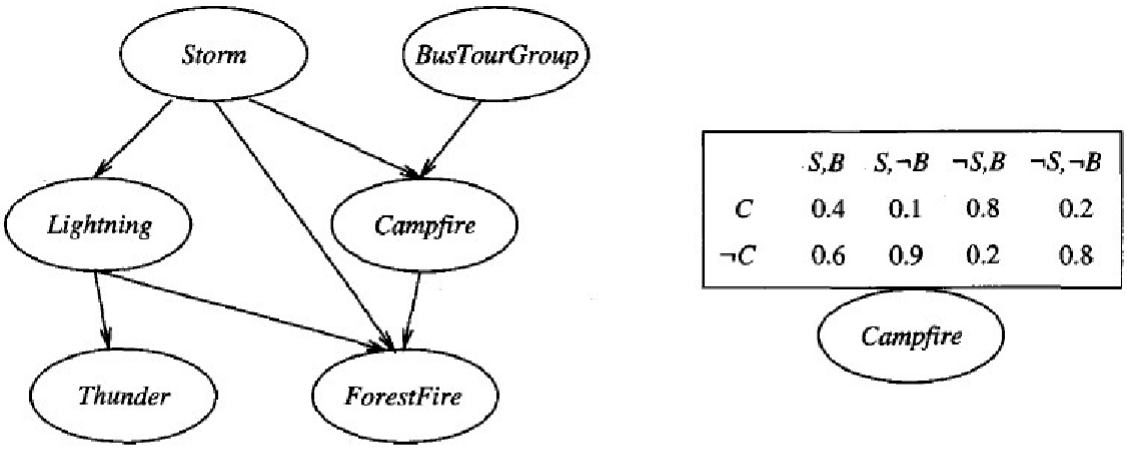


FIGURE 6.3
A Bayesian belief network. The network on the left represents a set of conditional independen assumptions. In particular, each node is asserted to be conditionally independent of its nondesce dants, given its immediate parents. Associated with each node is a conditional probability tabl which specifies the conditional distribution for the variable given its immediate parents in the grap The conditional probability table for the *Campfire* node is shown at the right, where *Campfire* abbreviated to *C*, *Storm* abbreviated to *S*, and *BusTourGroup* abbreviated to *B*.

Infe rence

We might wish to use a Bayesian network to infer the value of some target variable (e.g., *ForestFire)* given the observed values of the other variables. This inference step can be straightforward if values for all of the other variables in the network are known exactly. In the more general case we may wish to infer the probability distribution for some variable (e.g., ***ForestFire)*** given observed values for only a subset of the other variables e.g., ***Thunder*** and ***BusTourGroup*** may be the only observed values available). In general, 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. Exact inference of probabilities in general for an arbitrary Bayesian network is known to be NP-hard (Cooper 1990). N umerous methods have been proposed for probabilistic inference in Bayesian networks, including exact inference methods and approximate inference methods that sacrifice precision to gain efficiency.

Learning Bayesian Belief Networks

Can we devise effective algorithms for learning Bayesian belief networks from training data?. Several different settings for this learning problem can be considered. 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. We simply estimate the conditional probability table entries just as we would for a naive Bayes classifier.

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. This problem is somewhat analogous to learning the weights for the hidden units in an artificial neural network. In fact, Russell et al. (1995) propose a similar gradient ascent procedure that learns the entries in the conditional probability tables.

Learning the Structure of Bayesian Networks

Learning Bayesian networks when the network structure is not known in advance is also difficult. Cooper and Herskovits (1992) present a Bayesian scoring metric for choosing among alternative networks. They also present a heuristic search algorithm called **K2** for learning network structure when the data is fully observable. Like most algorithms for learning the structure of Bayesian networks, K2 performs a greedy search that trades off network complexity for accuracy over the training data. Constraint-based approaches to learning Bayesian network structure have also been developed. These approaches infer independence and dependence relationships from the data, and then use these relationships to construct Bayesian networks.

THE EM ALGORITHM

In many practical learning settings, only a subset of the relevant instance features might be observable. EM algorithm is widely used approach to learning in the presence of unobserved variables. 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. The EM algorithm is also the basis for many unsupervised clustering algorithms.

Estimating Means of k Gaussians

The easiest way to introduce the EM algorithm is via an example. 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 for the case where k = 2. The learning task is to output a hypothesis $\mathbf{h = (\mu_1, \ldots \mu_k)}$ that describes the means of each of the k distributions. We would like to find a maximum likelihood hypothesis for these means; It is easy to calculate the maximum likelihood hypothesis for the mean of a single Normal distribution.

The maximum likelihood hypothesis is the one that minimizes the sum of squared errors over the m training instances. We have

$$\mu_{ML} = \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{m} (x_i - \mu)^2 \qquad (6.27)$$

In this case, the sum of squared errors is minimized by the sample mean.

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^{m} x_i \qquad (6.28)$$

Our problem here, however, involves a mixture of k different Normal distributions, and we cannot observe which instances were generated by which distribution. Thus, we have a prototypical example of a problem involving hidden variables. We can think of the full description of each instance as the triple (x_i, z_{i1}, z_{i2}), where x_i is the observed value of the i^{th} instance and where z_{il} and z_{i2} 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.

Applied to the problem of estimating the two means the EM algorithm first initializes the hypothesis to $\mathbf{h} = (\mu_1, \mu_k)$, where μ_1 and μ_k are arbitrary initial values. It then iteratively re-estimates h by repeating the following two steps until the procedure converges to a stationary value for h.

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.

$$E[z_{ij}] = \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}}$$

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.

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}]\ x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

General Statement of EM Algorithm

➔ More generally, the EM algorithm can be applied in many settings where we wish to estimate some set of parameters **θ** that describe an underlying probability distribution,

given only the observed portion of the full data produced by this distribution. In the above two- means example the parameters of interest were $\theta = (\mu_1, \mu_2)$, and the full data were the triples (x_i, z_{i1}, z_{i2}) of which only the x_i were observed.

➔ In general let $X = \{x_1, \ldots, x_m\}$ denote the observed data in a set of m independently drawn instances, let $Z = \{z_1, \ldots, z_m\}$ denote the unobserved data in these same instances, and let $Y = X \cup Z$ denote the full data.

➔ We use h to denote the current hypothesized values of the parameters θ, and h' to denote the revised hypothesis that is estimated on each iteration ofthe EM algorithm.

➔ The EM algorithm searches for the maximum likelihood hypothesis **h'** by seeking the **h'** that maximizes E[ln P(Y|h')].

➔ Let us define a function Q(h'|h) that gives E[ln P(Y|h')] as a function of h', under the assumption that θ = h and given the observed portion X of the full data Y.

$$Q(h'|h) = E[\ln p(Y|h')|h, X]$$

➔ In its general form, the EM algorithm repeats the following two steps until convergence:

Step 1: ***Estimation*** **(E)** ***step:*** Calculate $Q(h'|h)$ using the current hypothesis h and the observed data X to estimate the probability distribution over Y.

$$Q(h'|h) \leftarrow \text{E}\,[\ln P(Y|h') \mid h, \text{X}]$$

Step 2: ***Maximization (M) step:*** Replace hypothesis h by the hypothesis h' that maximizes this Q function.

$$h \leftarrow \underset{h'}{\text{argmax}}\, Q(h'|h)$$

When the function Q is continuous, the EM algorithm converges to a stationary point of the likelihood function $P(Y|h')$.

In this respect, EM shares some of the same limitations as other optimization methods such as gradient descent, line search, and conjugate gradient.

Derivation of the k Means Algorithm

Let us use EM Algorithm to derive the algorithm for estimating the means of a mixture of k Normal distributions. To apply EM we must derive an expression for $Q(h|h')$ that applies to our k- means problem.

First, let us derive an expression for $\ln p(Y|h')$. Note the probability $p(y_i|h')$ of a single instance $y_i = (x_i, z_{i1}, \ldots z_{ik})$ of the full data can be written

$$p(y_i|h') = p(x_i, z_{i1}, \ldots, z_{ik}|h') = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\sum_{j=1}^{k} z_{ij}(x_i-\mu'_j)^2}$$

Given this probability for a single instance $p(y_i|h')$, the logarithm of the probability In $P(Y|h')$ for all m instances in the data is

$$\ln P(Y|h') = \ln \prod_{i=1}^{m} p(y_i|h')$$
$$= \sum_{i=1}^{m} \ln p(y_i|h')$$
$$= \sum_{i=1}^{m} \left(\ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{j=1}^{k} z_{ij}(x_i - \mu'_j)^2 \right)$$

Note the above expression for In P(Y|h') is a linear function of these z_{ij}. In general, for any function f (z) that is a linear function of z, the following equality holds

$$E[f(z)] = f(E[z])$$

This general fact about linear functions allows us to write

$$E[\ln P(Y|h')] = E\left[\sum_{i=1}^{m} \left(\ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{j=1}^{k} z_{ij}(x_i - \mu'_j)^2 \right) \right]$$
$$= \sum_{i=1}^{m} \left(\ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2 \right)$$

To summarize, the function Q(h'|h) for the k means problem is

$$Q(h'|h) = \sum_{i=1}^{m} \left(\ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2 \right)$$

where $h' = \langle \mu'_1, \ldots, \mu'_k \rangle$ and where $E[z_{ij}]$ is calculated based on the current hypothesis h and observed data X. As discussed earlier

$$E[z_{ij}] = \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^{k} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}} \qquad (6.29)$$

Thus, the first (estimation) step of the EM algorithm defines the Q function based on the estimated $E[z_{ij}]$ terms. The second (maximization) step then finds the values $\mu'_1, \ldots, \mu'_k$ that maximize this Q function. In the current case

$$\underset{h'}{\operatorname{argmax}}\, Q(h'|h) = \underset{h'}{\operatorname{argmax}} \sum_{i=1}^{m} \left(\ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2 \right)$$
$$= \underset{h'}{\operatorname{argmin}} \sum_{i=1}^{m} \sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2 \qquad (6.30)$$

Thus, the maximum likelihood hypothesis here minimizes a weighted sum of squared errors, where the contribution of each instance x_i to the error that defines μ'_j is weighted by $E[z_{ij}]$. The quantity given by Equation (6.30) is minimized by setting each μ'_j to the weighted sample mean

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}]\ x_i}{\sum_{i=1}^{m} E[z_{ij}]} \qquad (6.31)$$