Artificial Intelligence

- Artificial intelligence is the study of how to make computers do things which, at moment people do better.
- Artificial intelligence can be viewed from a variety of perspectives.
- From the perspective of intelligence, artificial intelligence is making machines "intelligent" -- acting as we would expect people to act.
    - The inability to distinguish computer responses from human responses is called the Turing test.
    - Intelligence requires knowledge.
- 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.
    - Typically AI programs focus on symbols rather than numeric processing.
    - Problem solving i.e. to achieve a specific goal.
    - Search - rarely access a solution directly. Search may include a variety of techniques.
- It is the science and engineering of making intelligent machines, especially intelligent computer programs.

AI Problems

- Much of the early work in the field of AI focused on formal tasks, such as game playing and theorem proving.
- Game playing and theorem proving share the property that people who do them well are considered to be displaying *Intelligence*.
- Initially computers could perform well at those tasks simply by being fast at exploring a large number of solution paths and then selecting the best one.
- Humans learn mundane (ordinary) tasks since their birth. They learn by perception, speaking, using language, and training. They learn Formal Tasks and Expert Tasks later.
- Another early foray into AI focused on commonsense reasoning, which includes reasoning about physical objects and their relationship to each other, as well as reasoning about actions and their consequences.
- As AI research progressed, techniques for handling large amount of world knowledge were developed.
- New tasks reasonably attempted such as perception, natural language understanding and problem solving in specialized domains.
- Some of the task domains of artificial intelligence are presented in table I.
- Earlier, all work of AI was concentrated in the mundane task domain.

Mundane tasks	Formal tasks	Expert tasks
Perception – Computer Vision – Speech, Voice	Games – Go – Chess (Deep Blue) – Ckeckers	Engineering – Design – Fault Finding – Manufacturing – Monitoring
Natural Language Processing – Understanding – Language Generation – Language Translation	Mathematics – Geometry – Logic – Integration and Differentiation	Scientific Analysis
Common Sense Reasoning	Theorem Proving	Financial Analysis
Planning		Medical Diagnosis
Robot Control		

Table I Task Domains of AI

- Later, it turned out that the machine requires more knowledge, complex knowledge representation, and complicated algorithms for handling mundane tasks.
- This is the reason why AI work is more flourishing in the Expert Tasks domain now, as the expert task domain needs expert knowledge without common sense, which can be easier to represent and handle.

What is an AI technique?

- Artificial intelligence problems span a very broad spectrum. They appear to have very little in common except that they are hard.
- AI Research of earlier decades results into the fact that intelligence requires knowledge.
- Knowledge possess following properties:
    - It is voluminous.
    - It is not well-organized or well-formatted.
    - It is constantly changing.
    - It differs from data. And it is organized in a way that corresponds to its usage.
- AI technique is a method that exploits knowledge that should be represented in such a way that:
    - Knowledge captures generalization. Situations that share common properties are grouped together. Without this property, inordinate amount of memory and modifications will be required.
    - It can be understood by people who must provide it. Although bulk of data can be acquired automatically, in many AI domains most of the knowledge must ultimately be provided by people in terms they understand.

- It can easily be modified to correct errors and to reflect changes in the world.
- It can be used in many situations even though it may not be totally accurate or complete.
- It can be used to reduce its own volume by narrowing range of possibilities.

- There are three important AI techniques:
  1. Search –
     a. Provides a way of solving problems for which no direct approach is available.
     b. It also provides a framework into which any direct techniques that are available can be embedded.
  2. Use of knowledge –
     a. Provides a way of solving complex problems by exploiting the structure of the objects that are involved.
  3. Abstraction –
     a. Provides a way of separating important features and variations from many unimportant ones that would otherwise overwhelm any process.

Classification of AI

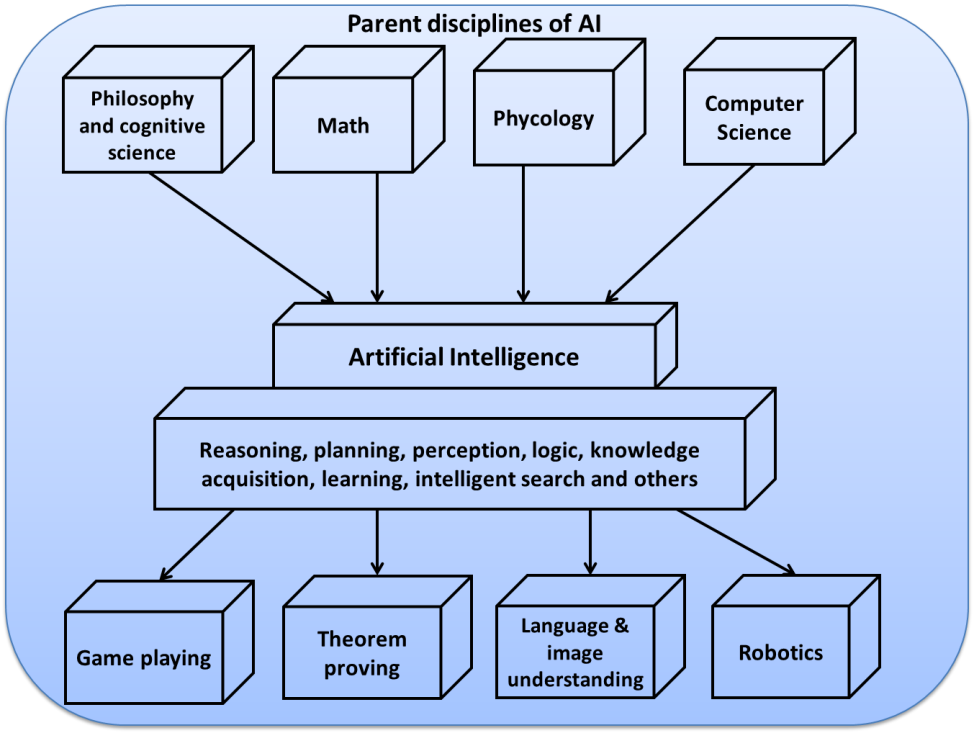
1. Weak AI: The study and design of machines that perform intelligent tasks.
   - Not concerned with how tasks are performed, mostly concerned with performance and efficiency, such as solutions that are reasonable for NP-Complete problems. E.g., to make a flying machine, use logic and physics, don't mimic a bird.
2. Strong AI: The study and design of machines that simulate the human mind to perform intelligent tasks.
   - Borrow many ideas from psychology, neuroscience. Goal is to perform tasks the way a human might do them – which makes sense, since we do have models of human thought and problem solving.
   - Includes psychological ideas in STM, LTM, forgetting, language, genetics, etc. Assumes that the physical symbol hypothesis holds.
3. Evolutionary AI. The study and design of machines that simulate simple creatures, and attempt to evolve and have higher level emergent behavior. For example, ants, bees, etc.

Applications of AI

- AI has been dominant in various fields such as −
  1. Gaming − AI plays vital role in strategic games such as chess, poker, tic-tac-toe, etc., where machine can think of large number of possible positions based on heuristic knowledge.
  2. Natural Language Processing − It is possible to interact with the computer that

understands natural language spoken by humans.

3. Expert Systems − There are some applications which integrate machine, software, and special information to impart reasoning and advising. They provide explanation and advice to the users.
4. Computer Vision Systems − These systems understand, interpret, and comprehend visual input on the computer.
5. Speech Recognition − Some intelligent systems are capable of hearing and comprehending the language in terms of sentences and their meanings while a human talks to it. It can handle different accents, slang words, noise in the background, change in human's noise, etc.
6. Handwriting Recognition − The handwriting recognition software reads the text written on paper by a pen or on screen by a stylus. It can recognize the shapes of the letters and convert it into editable text.
7. Intelligent Robots − Robots are able to perform the tasks given by a human. They have sensors to detect physical data from the real world such as light, heat, temperature, movement, sound, bump, and pressure. They have efficient processors, multiple sensors and huge memory, to exhibit intelligence. In addition, they are capable of learning from their mistakes and they can adapt to the new environment.



Introduction

- Problem solving is the major area of concern in Artificial Intelligence.
- It is the process of generating solution from given observed data.
- To solve a particular problem, we need to build a system or a method which can generate required solution.
- Following four things are required for building such system.
    1. Define the problem precisely.
        - This definition must precisely specify the initial situation (input).
        - What final situation (output) will constitute the acceptable solution to the problem.
    2. Analyze the problem.
        - To identify those important features which can have an immense impact on the appropriateness of various possible techniques for solving the problem.
    3. Isolate and represent the task knowledge that is necessary to solve the problem.
    4. Choose the best problem solving technique and apply it to the particular problem.

Defining the Problem as a State Space Search

1. Defining Problem & Search

- A problem is described formally as:
    1. Define a state space that contains all the possible configurations of relevant objects.
    2. Specify one or more states within that space that describe possible situations from which the problem solving process may start. These states are called initial states.
    3. Specify one or more states that would be acceptable as solutions to the problem. These states are called goal states.
    4. Specify a set of rules that describe the actions available.
- The problem can then be solved by using the rules, in combination with an appropriate control strategy, to move through the problem space until a path from an initial state to a goal state is found.
- This process is known as search.
- Search is fundamental to the problem-solving process.
- Search is a general mechanism that can be used when more direct method is not known.
- Search also provides the framework into which more direct methods for solving subparts of a problem can be embedded.

2. Defining State & State Space

- A ***state*** is a representation of problem elements at a given moment.
- ***A State space is the set of all states reachable from the initial state.***

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

- ***The state space representation forms the basis of most of the AI method*s.**
- Its structure corresponds to the structure of problem solving in two important ways:
  1. It allows for a formal definition of a problem as per the need to convert some given situation into some desired situation using a set of permissible operations.
  2. It permits the problem to be solved with the help of known techniques and control strategies to move through the problem space until goal state is found.

3. Define the Problem as State Space Search

Ex.1:- Consider the problem of Playing Chess

- To build a program that could play chess, we have to specify:
  - The starting position of the chess board,
  - The rules that define legal moves, and
  - The board position that represents a win.
- The starting position can be described by an 8 X 8 array square in which each element square (x, y), (x varying from 1 to 8 & y varying from 1 to 8) describes the board position of an appropriate piece in the official chess opening position.
- The goal is any board position in which the opponent does not have a legal move and his or her "king" is under attack.
- The legal moves provide the way of getting from initial state of final state.
- The legal moves can be described as a set of rules consisting of two parts: A left side that gives the current position and the right side that describes the change to be made to the board position.
- An example is shown in the following figure.

Current Position

While pawn at square (5 , 2), AND Square (5 , 3) is empty, AND Square (5 , 4) is empty.

Changing Board Position

Move pawn from Square (5 , 2) to Square (5 , 4) .

- The current position of a chess coin on the board is ***its state*** and the set of all possible states is ***state space***.
- One or more states where the problem terminates are goal states.
- Chess has approximately 10^{120} game paths. These positions comprise the problem search space.
- Using above formulation, the problem of playing chess is defined as a problem of moving around in a state space, where each state corresponds to a legal position of the

board.

- State space representation seems natural for play chess problem because the set of states, which corresponds to the set of board positions, is well organized.

Ex.2:- Consider Water Jug problem

- A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?
- Here the initial state is (0, 0). The goal state is (2, n) for any value of n.
- **State Space Representation:** we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note that $0 \le x \le 4$, and $0 \le y \le 3$.
- To solve this we have to make some assumptions not mentioned in the problem. They are:
  - We can fill a jug from the pump.
  - We can pour water out of a jug to the ground.
  - We can pour water from one jug to another.
  - There is no measuring device available.
- Operators – we must define a set of operators that will take us from one state to another.

Sr.	Current state	Next State	Descriptions
1	(x, y) if x < 4	(4,y)	Fill the 4 gallon jug
2	(x, y) if y <3	(x,3)	Fill the 3 gallon jug
3	(x, y) if x > 0	(x-d, y)	Pour some water out of the 4 gallon jug
4	(x, y) if y > 0	(x, y-d)	Pour some water out of the 3 gallon jug
5	(x, y) if x>0	(0, y)	Empty the 4 gallon jug
6	(x, y) if y >0	(x,0)	Empty the 3 gallon jug on the ground
7	(x, y) if x+y >= 4 and y > 0	(4, y-(4-x))	Pour water from the 3 gallon jug into the 4 gallon jug until the 4 gallon jug is full
8	(x, y) if x+y >= 3 and x>0	(x-(3-y), 3)	Pour water from the 4 gallon jug into the 3-gallon jug until the 3 gallon jug is full
9	(x, y) if x+y <=4 and y>0	(x+y, 0)	Pour all the water from the 3 gallon jug into the 4 gallon jug

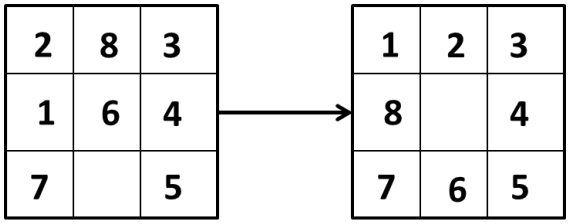
10	(x, y) if x+y <= 3 and x>0	(0, x+y)	Pour all the water from the 4 gallon jug into the 3 gallon jug
11	(0,2)	(2,0)	Pour the 2 gallons from 3 gallon jug into the 4 gallon jug
12	(2,y)	(0,y)	Empty the 2 gallons in the 4 gallon jug on the ground

- There are several sequences of operators that will solve the problem.
- One of the possible solutions is given as:

Gallons in the 4-gallon jug	Gallons in the 3-gallon jug	Rule applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5 or 12
0	2	9 0r 11
2	0	--

Ex.3:- Consider 8 puzzle problem

- The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.



Initial State **Goal State**

- The program is to change the initial configuration into the goal configuration.
- A solution to the problem is an appropriate sequence of moves, such as “move tiles 5 to the right, move tile 7 to the left ,move tile 6 to the down” etc…

- To solve a problem, we must specify the global database, the rules, and the control strategy.
- For the 8 puzzle problem that correspond to three components.
- These elements are the problem states, moves and goal.
- In this problem each tile configuration is a state.
- The set of all possible configuration in the problem space, consists of 3,62,880 different configurations of the 8 tiles and blank space.
- For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers. Initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states.
- A move transforms one problem state into another state.

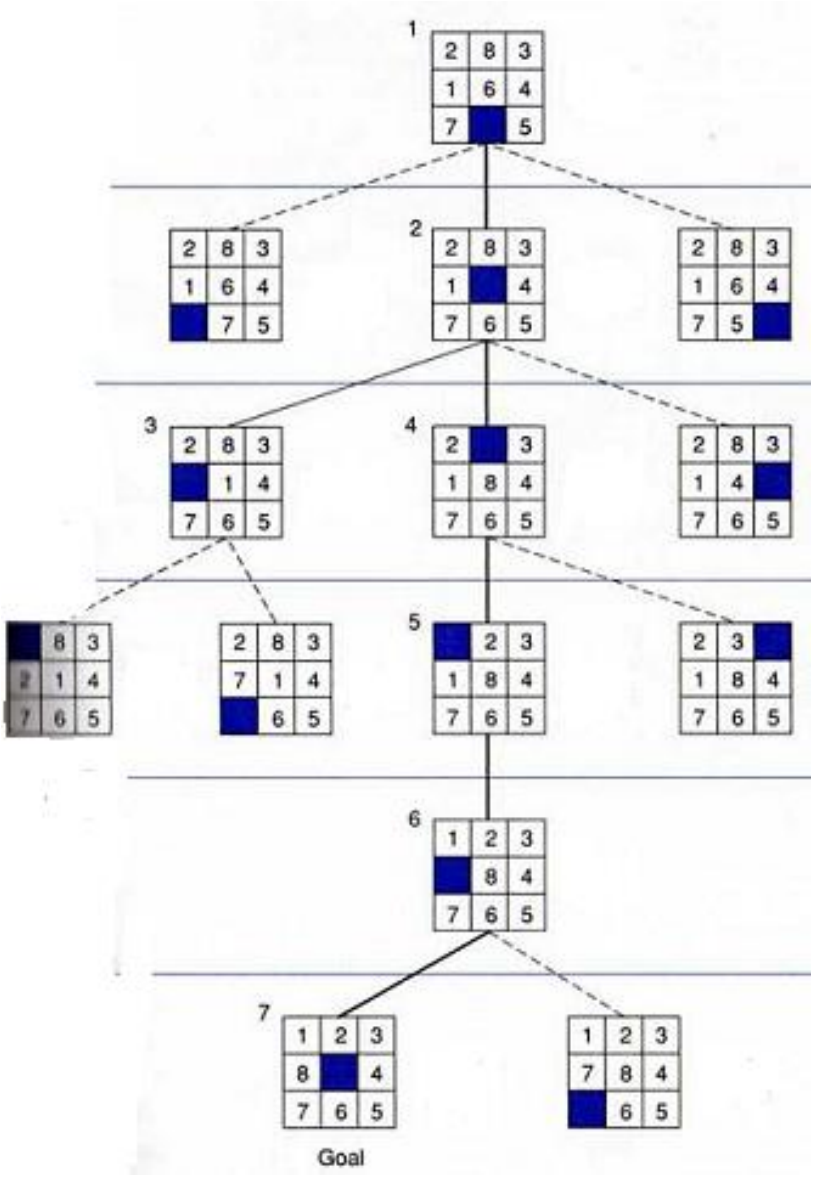


Figure 1: Solution of 8 Puzzle problem

- The 8-puzzle is conveniently interpreted as having the following for moves.

  - Move empty space (blank) to the left, move blank up, move blank to the right and move blank down.
  - These moves are modeled by production rules that operate on the state descriptions in the appropriate manner.
- The goal condition forms the basis for the termination.
- The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced.
- It also keeps track of rules that have been applied so that it can compose them into sequence representing the problem solution.
- A solution to the 8-puzzle problem is given in fig. 1.

Production System

- Search process forms the core of many intelligence processes.
- So, it is useful to structure AI programs in a way that facilitates describing and performing the search process.
- Production system provides such structures.
- A production system consists of:
  1. **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 that rule is applied.
  2. **One or more knowledge/databases** that contain whatever information is appropriate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem.
  3. **A control strategy** that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.
  4. **A rule applier** which is the computational system that implements the control strategy and applies the rules.
- In order to solve a problem:
  - We must first reduce it to the form for which a precise statement can be given. This can be done by defining the problem's state space (start and goal states) and a set of operators for moving that space.
  - The problem can then be solved by searching for a path through the space from an initial state to a goal state.
  - The process of solving the problem can usefully be modeled as a production system.

Benefits of Production System

1. Production systems provide an excellent tool for structuring AI programs.
2. Production Systems are highly modular because the individual rules can be added, removed or modified independently.
3. The production rules are expressed in a natural form, so the statements contained in the

knowledge base should be easily understandable.

Production System Characteristics

1. Monotonic Production System: the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected. i.e., rules are independent.
2. Non-Monotonic Production system is one in which this is not true.
3. Partially commutative Production system: a production system with the property that if application of a particular sequence of rules transforms state x to state y, then allowable permutation of those rules, also transforms state x into state y.
4. Commutative Production system: A Commutative production system is a production system that is both monotonic and partially commutative.

Control Strategies

- Control strategies help us decide which rule to apply next during the process of searching for a solution to a problem.
- Good control strategy should:
    1. It should cause motion
    2. It should be Systematic
- Control strategies are classified as:
    1. Uninformed/blind search control strategy:
        - Do not have additional information about states beyond problem definition.
        - Total search space is looked for solution.
        - Example: Breadth First Search (BFS), Depth First Search (DFS), Depth Limited Search (DLS).
    2. Informed/Directed Search Control Strategy:
        - Some information about problem space is used to compute preference among the various possibilities for exploration and expansion.
        - Examples: Best First Search, Problem Decomposition, A*, Mean end Analysis

Breadth-First Search Strategy (BFS)

- This is an exhaustive search technique.
- The search generates all nodes at a particular level before proceeding to the next level of the tree.
- ***The search systematically proceeds testing each node that is reachable from a parent node before it expands to any child of those nodes.***
- Search terminates when a solution is found and the test returns true.

Algorithm:

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

i. *Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit.*
ii. *For each way that each rule can match the state described in E do:*
    a. *Apply the rule to generate a new state.*
    b. *If the new state is a goal state, quit and return this state.*
    c. *Otherwise, add the new state to the end of NODE-LIST.*

Depth-First Search Strategy (DFS)

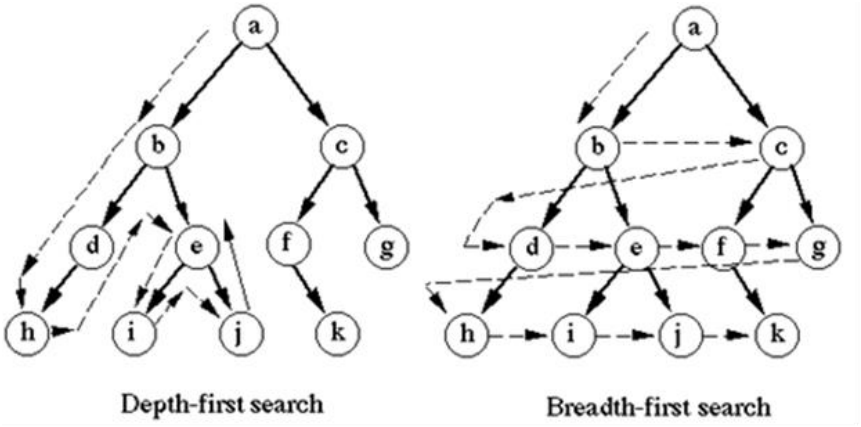
- Here, the search systematically proceeds to some depth *d*, before another path is considered.
- If the maximum depth of search tree is reached and if the solution has not been found, then the search backtracks to the previous level and explores any remaining alternatives at this level, and so on.

Algorithm:

1. *If the initial state is a goal state, quit and return success*
2. *Otherwise, do the following until success or failure is signaled:*
    a. *Generate a successor, E, of 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.*

Comparison: DFS & BFS

Depth First Search	Breath First Search
DFS requires less memory since only the nodes on the current path are stored.	BFS guarantees that the space of possible moves is systematically examined; this search requires considerable memory resources.
By chance, DFS may find a solution without examining much of the search space at all. Then it finds solution faster.	The search systematically proceeds testing each node that is reachable from a parent node before it expands to any child of those nodes.
If the selected path does not reach to the solution node, DFS gets stuck into a blind alley.	BFS will not get trapped exploring a blind alley.
Does not guarantee to find solution. Backtracking is required if wrong path is selected.	If there is a solution, BFS is guaranteed to find it.

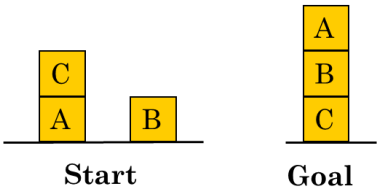


Problem Characteristics

- In order to choose the most appropriate problem solving method, it is necessary to analyze the problem along various key dimensions.
- These dimensions are referred to as problem characteristics discussed below.

1. Is the problem decomposable into a set of independent smaller or easier sub-problems?

- A very large and composite problem can be easily solved if it can be broken into smaller problems and recursion could be used.
- For example, we want to solve :- $\int x^2 + 3x + \sin 2x \cos 2x \, dx$
- This can be done by breaking it into three smaller problems and solving each by applying specific rules. Adding the results we can find the complete solution.
- But there are certain problems which cannot be decomposed into sub-problems.
- For example Blocks world problem in which, start and goal state are given as,



- Here, solution can be achieved be moving blocks in a sequence such that goal state can be derived.
- Solution steps are interdependent and cannot be decomposed in sub problems.
- These two examples, symbolic integration and the blocks world illustrate the difference between decomposable and non-decomposable problems.

2. Can solution steps be ignored or at least undone if they prove unwise?

- Problem fall under three classes, (i) ignorable, (ii) recoverable and (iii) irrecoverable.
- This classification is with reference to the steps of the solution to a problem.
- Consider theorem proving. We may later find that it is of no use. We can still proceed further, since nothing is lost by this redundant step. This is an example of ignorable solutions steps.
- Now consider the 8 puzzle problem tray and arranged in specified order.
- While moving from the start state towards goal state, we may make some stupid move

but we can backtrack and undo the unwanted move. This only involves additional steps and the solution steps are recoverable.

- Lastly consider the game of chess. If a wrong move is made, it can neither be ignored nor be recovered. The thing to do is to make the best use of current situation and proceed. This is an example of an irrecoverable solution steps.
- Knowledge of these will help in determining the control structure.
  - Ignorable problems can be solved using a simple control structure that never backtracks.
  - Recoverable problems can be solved by a slightly more complicated control strategy that allows backtracking.
  - Irrecoverable problems will need to be solved by a system that expends a great deal of effort making each decision since decision must be final.

3. Is the problem's universe predictable?

- Problems can be classified into those with certain outcome (eight puzzle and water jug problems) and those with uncertain outcome (playing cards).
- In certain – outcome problems, planning could be done to generate a sequence of operators that guarantees to lead to a solution.
- Planning helps to avoid unwanted solution steps.
- For uncertain outcome problems, planning can at best generate a sequence of operators that has a good probability of leading to a solution.
- The uncertain outcome problems do not guarantee a solution and it is often very expensive since the number of solution paths to be explored increases exponentially with the number of points at which the outcome cannot be predicted.
- Thus one of the hardest types of problems to solve is the irrecoverable, uncertain – outcome problems (Ex:- Playing cards).

4. Is a good solution to the problem obvious without comparison to all other possible solutions?

- There are two categories of problems - Any path problem and Best path problem.
- In any path problem, like the water jug and 8 puzzle problems, we are satisfied with the solution, irrespective of the solution path taken.
- Whereas in the other category not just any solution is acceptable but we want the best path solution.
- Like that of traveling sales man problem, which is the shortest path problem.
- In any – path problems, by heuristic methods we obtain a solution and we do not explore alternatives.
- Any path problems can often be solved in a reasonable amount of time by using heuristics that suggest good paths to explore.
- For the best-path problems all possible paths are explored using an exhaustive search until the best path is obtained.
- Best path problems are computationally harder.

5. **Is the desired solution a state of the world or a path to a state?**
   - Consider the problem of natural language processing.
   - Finding a consistent interpretation for the sentence "The bank president ate a dish of pasta salad with the fork".
   - We need to find the interpretation but not the record of the processing by which the interpretation is found.
   - Contrast this with the water jug problem.
   - In water jug problem, it is not sufficient to report that we have solved, but the path that we found to the state (2, 0). Thus the statement of a solution to this problem must be a sequence of operations that produces the final state.
6. **What is the role of knowledge?**
   - Though one could have unlimited computing power, the size of the knowledge base available for solving the problem does matter in arriving at a good solution.
   - Take for example the game of playing chess, just the rules for determining legal moves and some simple control mechanism is sufficient to arrive at a solution.
   - But additional knowledge about good strategy and tactics could help to constrain the search and speed up the execution of the program. The solution would then be realistic.
   - Consider the case of predicting the political trend. This would require an enormous amount of knowledge even to be able to recognize a solution, leave alone the best.
7. **Does the task require interaction with a person?**

   The problems can again be categorized under two heads.

   i. Solitary in which the computer will be given a problem description and will produce an answer, with no intermediate communication and with the demand for an explanation of the reasoning process. Simple theorem proving falls under this category. Given the basic rules and laws, the theorem could be proved, if one exists.

   ii. Conversational, in which there will be intermediate communication between a person and the computer, either to provide additional assistance to the computer or to provide additional information to the user, or both, such as medical diagnosis fall under this category, where people will be unwilling to accept the verdict of the program, if they cannot follow its reasoning.

Problem Classification

- Actual problems are examined from the point of view of all these questions; it becomes apparent that there are several broad classes into which the problems fall.

Issues in the design of search programs

1. The direction in which to conduct the search (forward versus backward reasoning). If the search proceeds from start state towards a goal state, it is a forward search or we can also search from the goal.
2. How to select applicable rules (Matching). Production systems typically spend most of their time looking for rules to apply. So, it is critical to have efficient procedures for matching

rules against states.

3. How to represent each node of the search process (knowledge representation problem).

Heuristic Search Techniques

- In order to solve many hard problems efficiently, it is often necessary to compromise the requirements of mobility and systematicity and to construct a control structure that is no longer guaranteed to find the best answer but will always find a very good answer.
- Usually very hard problems tend to have very large search spaces. Heuristics can be used to limit search process.
- There are good general purpose heuristics that are useful in a wide variety of problem domains.
- Special purpose heuristics exploit domain specific knowledge.
- For example nearest neighbor heuristics for shortest path problem. It works by selecting locally superior alternative at each step.
- Applying nearest neighbor heuristics to Travelling Salesman Problem:
    1. Arbitrarily select a starting city
    2. To select the next city, look at all cities not yet visited and select the one closest to the current city. Go to next step.
    3. Repeat step 2 until all cities have been visited.
- This procedure executes in time proportional to N^2, where N is the number of cities to be visited.

Heuristic Function

- ***Heuristic function maps from problem state descriptions to measures of desirability, usually represented as numbers.***
- Which aspects of the 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 a solution.
- Well-designed heuristic functions can play an important part in efficiently guiding a search process toward a solution.
- Every search process can be viewed as a traversal of a directed graph, in which the nodes represent problem states and the arcs represent relationships between states.
- The search process must find a path through this graph, starting at an initial state and ending in one or more final states.
- Domain-specific knowledge must be added to improve search efficiency. Information about the problem includes the nature of states, cost of transforming from one state to another, and characteristics of the goals.
- This information can often be expressed in the form of heuristic evaluation function.

- In general, heuristic search improve the quality of the path that are exported.
- Using good heuristics we can hope to get good solutions to hard problems such as the traveling salesman problem in less than exponential time.

Heuristic Search Techniques

I. Generate-and-Test

- 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:

1. *Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others it means generating a path from a start stat.*
2. *Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.*
3. *If a solution has been found, quit, Otherwise return to step 1.*

- It is a depth first search procedure since complete solutions must be generated before they can be tested.
- In its most systematic form, it is simply an exhaustive search of the problem space.
- It operates by generating solutions randomly.

II. Simple Hill Climbing

- Hill climbing is a variant of generate-and test in which feedback from the test procedure is used to help the generator decide which direction to move in search space.
- The test function is augmented with a heuristic function that provides an estimate of how close a given state is to the goal state.
- Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available.
- ***The key difference between Simple Hill climbing and Generate-and-test is the use of evaluation function as a way to inject task specific knowledge into the control process.***

Algorithm:

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

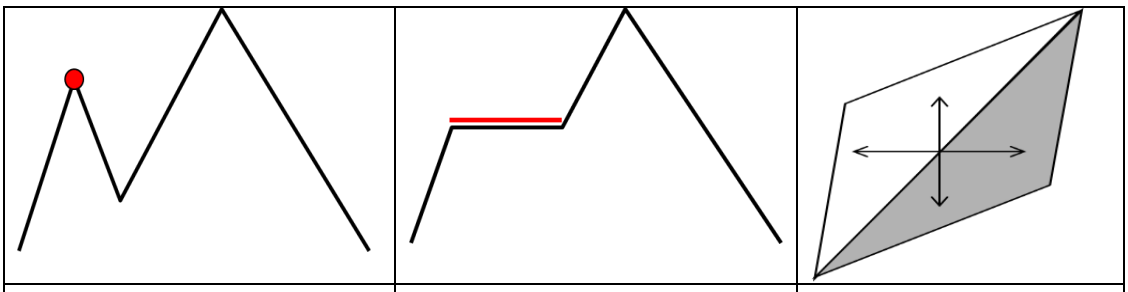
III. Steepest-Ascent Hill Climbing

- This is a variation of simple hill climbing which considers all the moves from the current state and selects the best one as the next state.
- At each current state we select a transition, evaluate the resulting state, and if the resulting state is an improvement we move there, otherwise we try a new transition from where we were.
- We repeat this until we reach a goal state, or have no more transitions to try.
- The transitions explored can be selected at random, or according to some problem specific heuristics.

Algorithm

1. *Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.*
2. *Loop until a solution is found or until a complete iteration produces no change to current state:*
    a. *Let S be a state such that any possible successor of the current state will be better than S.*
    b. *For each operator that applies to the current state do:*
        i. *Apply the operator and generate a new state*
        ii. *Evaluate the new state. If is is a goal state, then return it and quit. If not, compare it to S. If it is better, then set S to this state. If it is not better, leave S alone.*
    c. *If the S is better than the current state, then set current state to S.*

- Hill Climbing has three well-known drawbacks:
    i. **Local Maxima**: a local maximum is a state that is better than all its neighbors but is not better than some other states further away.
    ii. **Plateau**: a plateau is a flat area of the search space in which, a whole set of neighboring states have the same values.
    iii. **Ridge**: is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has slop.



Local Maxima	Plateau	Ridge

- In each of the previous cases (local maxima, plateaus & ridge), the algorithm reaches a

point at which no progress is being made.

- A solution is,
    i. Backtrack to some earlier node and try going in a different direction.
    ii. Make a big jump to try to get in a new section.
    iii. Moving in several directions at once.

IV. Best First Search

- DFS is good because it allows a solution to be found without expanding all competing branches. BFS is good because it does not get trapped on dead end paths.
- Best first search combines the advantages of both DFS and BFS into a single method.
- One way of combining BFS and DFS is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.
- At each step of the Best First Search process; we select the most promising of the nodes we have generated so far.
- This is done by applying an appropriate heuristic function to each of them.
- We then expand the chosen node by using the rules to generate its successors.
- If one of them is a solution, we can quit. If not, all those new nodes are added to the set of nodes generated so far.

OR Graphs

- It is sometimes important to search graphs so that duplicate paths will not be pursued.
- An algorithm to do this will operate by searching a directed graph in which each node represents a point in problem space.
- Each node will contain:
    - Description of problem state it represents
    - Indication of how promising it is
    - Parent link that points back to the best node from which it came
    - List of nodes that were generated from it
- Parent link will make it possible to recover the path to the goal, once the goal is found.
- The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors.
- This is called OR-graph, since each of its branches represents an alternative problem solving path.

Implementation of OR graphs

We need two lists of nodes:

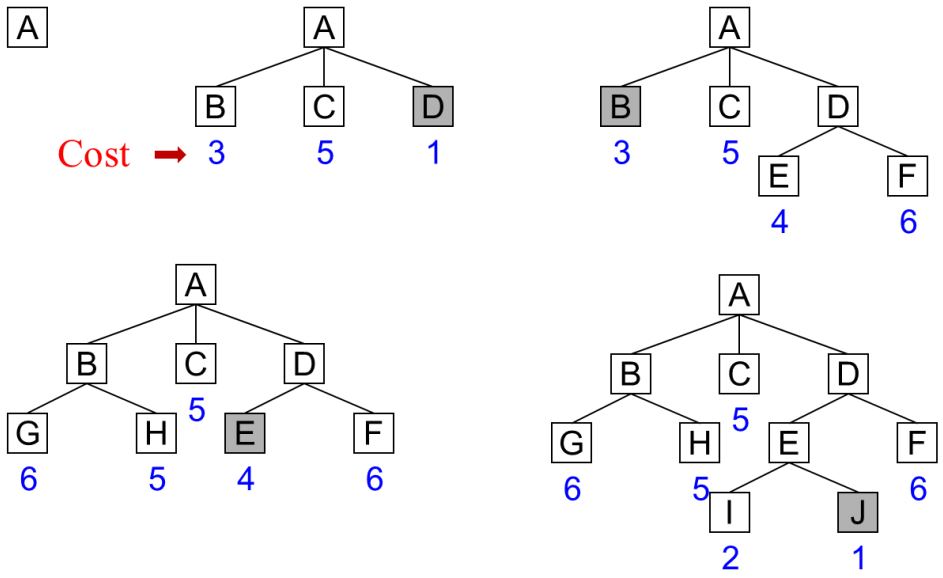
- OPEN – nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined. OPEN is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function.
- CLOSED- nodes that have already been examined. We need to keep these nodes in

memory if we want to search a graph rather than a tree, since whenever a new node is generated; we need to check whether it has been generated before.

Algorithm: Best First Search

1. *Start with OPEN containing just the initial state*
2. *Until a goal is found or there are no nodes left on OPEN do:*
   a. *Pick the best node on OPEN*
   b. *Generate its successors*
   c. *For each successor do:*
      i. *If it has not been generated before, evaluate it, add it to OPEN, and record its parent.*
      ii. *If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.*

Best First Search example



The A* Algorithm

- Best First Search is a simplification of A* Algorithm.
- This algorithm uses following functions:
  1. ***f':*** Heuristic function that estimates the merits of each node we generate. ***f' = g + h'.*** ***f'*** represents an estimate of the cost of getting from the initial state to a goal state along with the path that generated the current node.
  2. ***g:*** The function *g* is a measure of the cost of getting from initial state to the current node.
  3. ***h':*** The function *h'* is an estimate of the additional cost of getting from the current node to a goal state.
- The algorithm also uses the lists: OPEN and CLOSED

Algorithm: A*

1. *Start with OPEN containing only initial node. Set that node's g value to 0, its h' value to whatever it is, and its f' value to h'+0 or h'. Set CLOSED to empty list.*
2. *Until a goal node is found, repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise select the node on OPEN with the lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it in CLOSED. See if the BESTNODE is a goal state. If so exit and report a solution. Otherwise, generate the successors of BESTNODE but do not set the BESTNODE to point to them yet. For each of the SUCCESSOR, do the following:*
    a. *Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.*
    b. *Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR*
    c. *See if SUCCESSOR is the same as any node on OPEN. If so call the node OLD.*
        i. *Check whether it is cheaper to get to OLD via its current parent or to SUCESSOR via BESTNODE by comparing their g values.*
        ii. *If OLD is cheaper, then do nothing. If SUCCESSOR is cheaper then reset OLD's parent link to point to BESTNODE.*
        iii. *Record the new cheaper path in g(OLD) and update f '(OLD).*
    d. *If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors.*
    e. *If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE's successors. Compute f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR).*

Observations about A*

- Role of g function: This lets us choose which node to expand next on the basis of not only of how good the node itself looks, but also on the basis of how good the path to the node was.
- h', the distance of a node to the goal. If h' is a perfect estimator of h, then A* will converge immediately to the goal with no search.

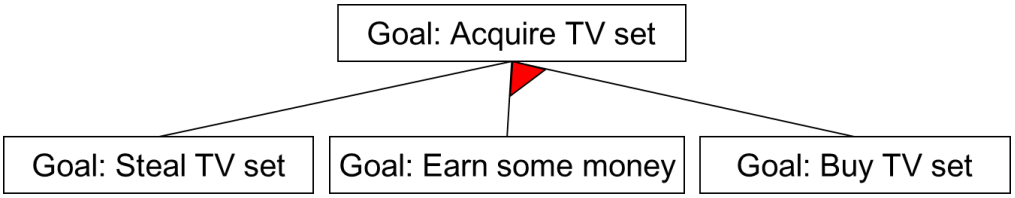
Admissibility of A*

- A heuristic function h'(n) is said to be admissible if it never overestimates the cost of getting to a goal state.
- i.e. if the true minimum cost of getting from node n to a goal state is C then h must satisfy: $h'(n) \leq C$
- If h' is a perfect estimator of h, then A* will converge immediately to the goal state with no search.
- If h' never overestimates h, then A* algorithm is guaranteed to find an optimal path if one exists.

V. Problem Reduction

AND-OR graphs

- AND-OR graph (or tree) is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved.
- This decomposition or reduction generates arcs that we call AND arcs.
- One AND arc may point to any numbers of successor nodes. All of which must then be solved in order for the arc to point solution
- In order to find solution in an AND-OR graph we need an algorithm similar to best –first search but with the ability to handle the AND arcs appropriately.
- ***We define FUTILITY, if the estimated cost of solution becomes greater than the value of FUTILITY then we abandon the search, FUTILITY should be chosen to correspond to a threshold***.
- Following figure shows AND arcs are indicated with a line connection all the components.



The AO* Algorithm

- Rather than the two lists, OPEN and CLOSED, that were used in the A* algorithm, the AO* algorithm will use a single structure GRAPH, representing the part of the search graph that has been explicitly generated so far.
- Each node in the graph will point both down to its immediate successors and up to its immediate predecessors.
- Each node in the graph will also have associated with it an *h'* value, an estimate of the cost of a path from itself to a set of solution nodes.
- We will not store *g* (the cost of getting from the start node to the current node) as we did in the A* algorithm.
- And such a value is not necessary because of the top-down traversing of the edge which guarantees that only nodes that are on the best path will ever be considered for expansion.

Algorithm: AO*

1. Let GRAPH consist only of the node representing the initial state. Call this node INIT, Compute VINIT.
2. Until INIT is labeled SOLVED or until INIT's h' value becomes greater than FUTILITY, repeat the following procedure:
    a. Trace the labeled arcs from INIT and select for expansion one of the as yet

unexpanded nodes that occurs on this path. Call the selected node NODE.

b. Generate the successors of NODE. If there are none, then assign FUTILITY as the h' value of NODE. This is equivalent to saying that NODE is not solvable. If there are successors, then for each one (called SUCCESSOR) that is not also an ancestor of NODE do the following:
    i. Add SUCCESSOR to GRAPH
    ii. If SUCCESSOR is a terminal node, label it SOLVED and assign it an h' value of 0
    iii. If SUCCESSOR is not a terminal node, compute its h' value
c. Propagate the newly discovered information up the graph by doing the following: Let S be a set of nodes that have been labeled SOLVED or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to NODE. Until S is empty, repeat the, following procedure:
    i. If possible, select from S a node none of whose descendants in GRAPH occurs in S. If there is no such node, select any node from S. Call this node CURRENT, and remove it from S.
    ii. Compute the cost of each of the arcs emerging from CURRENT. The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as CURRENT'S new h' value the minimum of the costs just computed for the arcs emerging from it.
    iii. Mark the best path out of CURRENT by marking the arc that had the minimum cost as computed in the previous step.
    iv. Mark CURRENT SOLVED if all of the nodes connected to it through the new labeled arc have been labeled SOLVED.
    v. If CURRENT has been labeled SOLVED or if the cost of CURRENT was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of CURRENT to S.

VI. Constraint Satisfaction

- Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description.
- A goal state is any state that has been constrained “enough” where “enough” must be defined for each problem.
- For example, in cryptarithmetic problems, enough means that each letter has been assigned a unique numeric value.
- Constraint Satisfaction problems in AI have goal of discovering some problem state that satisfies a given set of constraints.
- Design tasks can be viewed as constraint satisfaction problems in which a design must be

created within fixed limits on time, cost, and materials.

- Constraint Satisfaction is a two-step process:
    1. First constraints are discovered and propagated as far as possible throughout the system.
    2. Then if there is still not a solution, search begins. A guess about something is made and added as a new constraint.

Example: Cryptarithmetic Problem

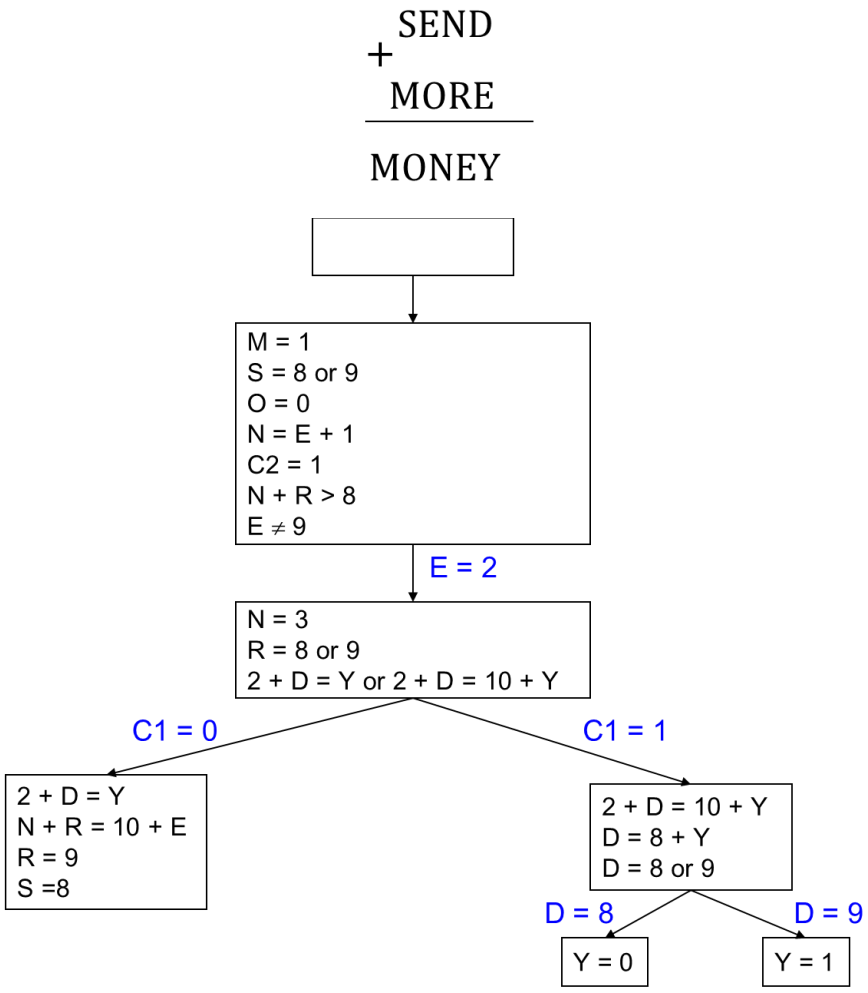
Constraints:

- No two letters have the same value
- The sums of the digits must be as shown in the problem

Goal State:

- All letters have been assigned a digit in such a way that all the initial constraints are satisfied

Input State



- The solution process proceeds in cycles. At each cycle, two significant things are done:
    1. Constraints are propagated by using rules that correspond to the properties of arithmetic.
    2. A value is guessed for some letter whose value is not yet determined.

Solution:

$$\begin{array}{r} 9567 \\ +\ 1085 \\ \hline 10652 \end{array}$$

Algorithm: Constraint Satisfaction

1. *Propagate available constraints. To do this first set OPEN to set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty:*
    a. *Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.*
    b. *If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraints with OB.*
    c. *Remove OB from OPEN.*
2. *If the union of the constraints discovered above defines a solution, then quit and report the solution.*
3. *If the union of the constraints discovered above defines a contradiction, then return the failure.*
4. *If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this loop until a solution is found or all possible solutions have been eliminated:*
    a. *Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.*
    b. *Recursively invoke constraint satisfaction with the current set of constraints augmented by strengthening constraint just selected.*

VII. Means-Ends Analysis

- Collection of strategies presented so far can reason either forward or backward, but for a given problem, one direction or the other must be chosen.
- A mixture of the two directions is appropriate. Such a mixed strategy would make it possible to solve the major parts of a problem first and then go back and solve the small problems that arise in “gluing” the big pieces together.
- The technique of Means-Ends Analysis (MEA) allows us to do that.
- MEA process centers around the detection of differences between the current state and the goal state.
- Once such a difference is isolated, an operator that can reduce the difference must be found.
- If the operator cannot be applied to the current state, we set up a sub-problem of

getting to a state in which it can be applied.

- The kind of backward chaining in which operators are selected and then sub-goals are set up to establish the preconditions of the operators is called operator sub-goaling

Algorithm: Means-Ends Analysis

1. *Compare CURRENT to GOAL. If there are no differences between them then return.*
2. *Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:*
    a. *Select an as yet untried operator O that is applicable to the current difference. If there are no such operators, then signal failure.*
    b. *Attempt to apply O to CURRENT. Generate descriptions of two states: O-START, a state in which O's preconditions are satisfied and O-RESULT, the state that would result if O were applied in O-START.*
    c. *If*

        (FIRST-PART ← MEA(CURRENT, O-START))

        and

        (LAST-PART ← MEA(O-RESULT, GOAL))

        are successful, then signal success and return the result of concatenating FIRST-PART, O, and LAST-PART.

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 intelligent system.
- ***Knowledge is a description of the world. It determines a system's competence by what it knows.***
- ***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.
- 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.
        - 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.
        - Can be shared, copied, processed and stored.
        - Easy to steal or copy
        - Drawn from artifact of some type as 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. Representation of facts in some chosen formalism. Things we will actually be able to manipulate.
- These entities are structured at two levels:
    1. The knowledge level, at which facts are described.

2. The symbol level, at which representation of objects are defined in terms of symbols that can be manipulated by programs

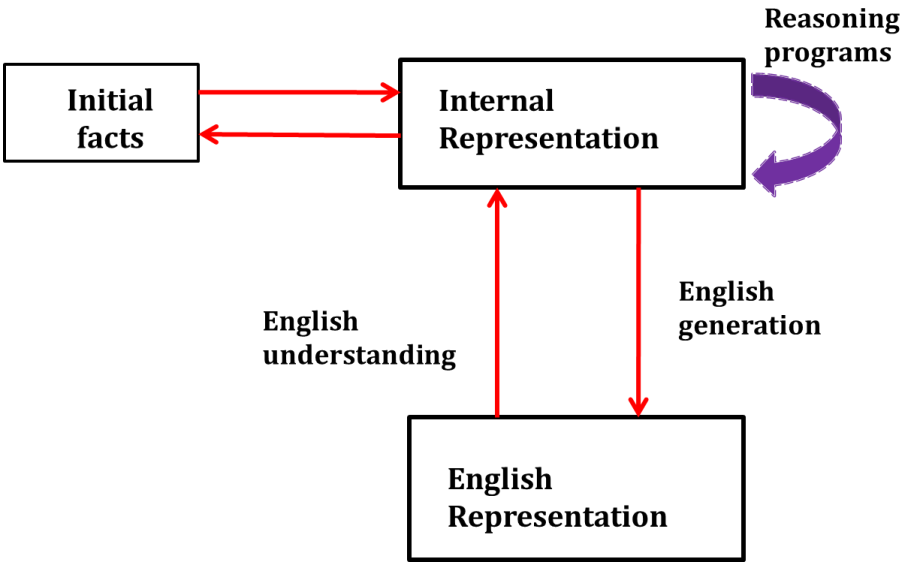


Fig. 3.1 Mapping between Facts and Representations

Framework of Knowledge Representation

- Computer requires a well-defined problem description to process and provide well-defined acceptable solution.
- 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.
- The computer can then use an algorithm to compute an answer. This process is illustrated as,

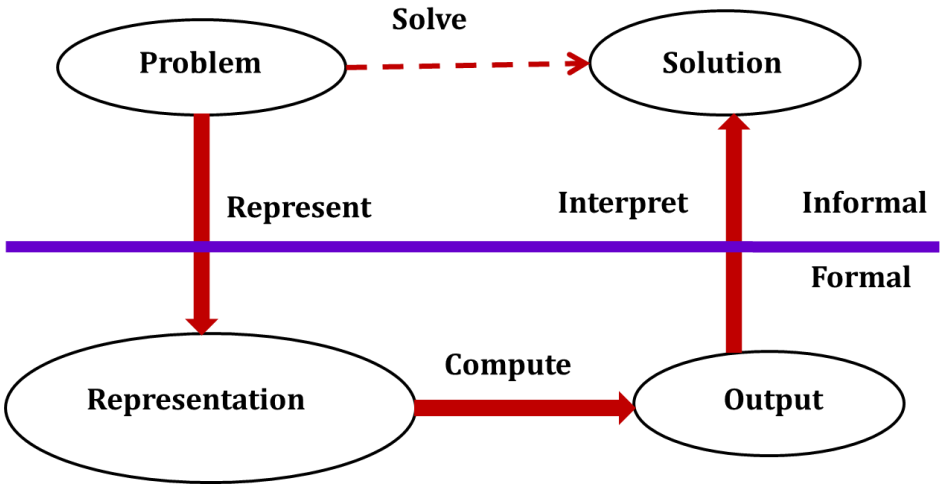


Fig. 3.2 Knowledge Representation Framework

- The steps are:
  - The informal formalism of the problem takes place first.
  - It is then represented formally and the computer produces an output.

- This output can then be represented in an informally described solution that user understands or checks for consistency.

- The Problem solving requires,
  - Formal knowledge representation, and
  - Conversion of informal knowledge to formal knowledge that is conversion of implicit knowledge to explicit knowledge.

Mapping between Facts and Representation

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

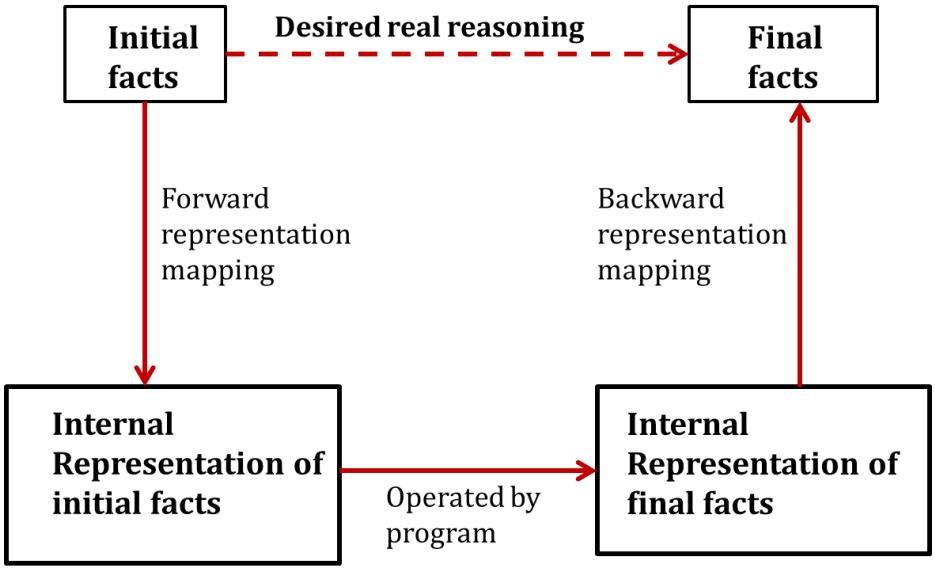


Fig. 3.3 Representation of Facts

Approaches to knowledge Representation

- 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
     - 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
    - The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Knowledge Representation Schemes

1. Relational Knowledge :

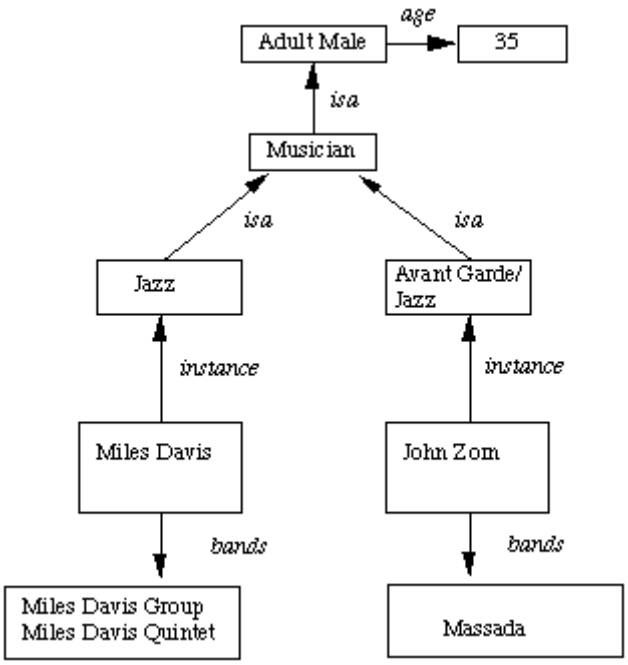
- The simplest way to represent declarative facts is as a set of relations of the same sort used in the database system.
- Provides a framework to compare two objects based on equivalent attributes.
- Any instance in which two different objects are compared is a relational type of knowledge.
- The table below shows a simple way to store facts.
    - 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 simple question such as :
  "Who is the heaviest player?"
- But if a procedure for finding heaviest player is provided, then these facts will enable that procedure to compute an answer.
- We can ask things like who "bats – left" and "throws – right".

2. Inheritable Knowledge

- Here the knowledge elements inherit attributes from their parents.
- The knowledge is 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 be prescribed to the child elements.
- The inheritance is a powerful form of inference, but not adequate.
- The basic KR (Knowledge Representation) needs to be augmented with inference mechanism.
- Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes.
- The classes are organized in a generalized hierarchy.



- Boxed nodes -- objects and values of attributes of objects.
- Arrows -- 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:
  i. Find the object in the knowledge base
  ii. If there is a value for the attribute report it
  iii. Otherwise look for a value of an instance, if none fail
  iv. Otherwise go to that node and find a value for the attribute and then report it
  v. Otherwise search through using isa until a value is found for the attribute.

3. 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) is used to infer from a set of attributes. Inference through predicate logic uses a set of logical operations to relate individual data.
- Represent knowledge as formal logic:

  All dogs have tails $\forall x: dog(x) \rightarrow hastail(x)$

- Advantages:
  - A set of strict rules.
  - Can be used to derive more facts.
  - Truths of new statements can be verified.
  - Guaranteed correctness.
- Many inference procedures available to implement standard rules of logic popular in AI

systems. e.g Automated theorem proving.

4. **Procedural Knowledge**
   - A representation in which the control information, to use the knowledge, is embedded in the knowledge itself. For example, computer programs, directions, and recipes; these indicate specific use or implementation;
   - Knowledge is encoded in some procedures, small programs that know how to do specific things, how to proceed.
   - Advantages:
     - Heuristic or domain specific knowledge can be represented.
     - Extended logical inferences, such as default reasoning facilitated.
     - Side effects of actions may be modeled. Some rules may become false in time. Keeping track of this in large systems may be tricky.
   - Disadvantages:
     - Completeness -- not all cases may be represented.
     - 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 is sacrificed. Changes in knowledge base might have far-reaching effects.
     - Cumbersome control information.

Issues in Knowledge Representation

- The fundamental goal of Knowledge Representation is to facilitate inference (conclusions) from knowledge.
- The issues that arise while using KR techniques are many. Some of these are explained below.

1. **Important Attributes :**
   - Any attribute of objects so basic that they occur in almost every problem domain?
   - There are two attributes "instance" and "isa", that are of general significance. These attributes are important because they support property inheritance.
2. **Relationship among attributes:**
   - Any important relationship that exists among object attributes?
   - The attributes we use to describe objects are themselves entities that we represent.
   - The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:
     i. Inverses - This is about consistency check, while a value is added to one attribute. The entities are related to each other in many different ways.
     ii. Existence in an *isa* hierarchy - This is about generalization-specialization, like, classes of objects and specialized subsets of those classes, there are attributes and specialization of attributes. For example, the attribute height is a specialization of general attribute physical-size which is, in turn, a specialization of physical-attribute. These generalization-specialization relationships are

important for attributes because they support inheritance.

iii. Techniques for reasoning about values - This is about reasoning values of attributes not given explicitly. Several kinds of information are used in reasoning, like,

height : must be in a unit of length,

Age: of person cannot be greater than the age of person's parents.

The values are often specified when a knowledge base is created.

iv. Single valued attributes - This is about a specific attribute that is guaranteed to take a unique value. For example, a baseball player can at time have only a single height and be a member of only one team. KR systems take different approaches to provide support for single valued attributes.

3. Choosing Granularity :

- At what level of detail should the knowledge be represented?
- Regardless of the KR formalism, it is necessary to know :
  - At what level should the knowledge be represented and what are the primitives?"
  - Should there be a small number or should there be a large number of low-level primitives or High-level facts.
  - High-level facts may not be adequate for inference while Low-level primitives may require a lot of storage.
- Example of Granularity :
  - Suppose we are interested in following facts:

    John spotted Sue.

    This could be represented as

    Spotted (agent(John), object (Sue))

    - Such a representation would make it easy to answer questions such are :
      - Who spotted Sue?

    Suppose we want to know :

      - Did John see Sue?

    - Given only one fact, we cannot discover that answer.
    - We can add other facts, such as

    Spotted (x , y) → saw (x , y)

    - We can now infer the answer to the question.

4. Set of objects :

- How should sets of objects be represented?
- There are certain properties of objects that are true as member of a set but not as individual;
  - Example : Consider the assertion made in the sentences :

    "there are more sheep than people in Australia", and

    "English speakers can be found all over the world."

- To describe these facts, the only way is to attach assertion to the sets representing people, sheep, and English.
- The reason to represent sets of objects is: If a property is true for all or most elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every elements of the set .
- This is done,
  - in logical representation through the use of universal quantifier, and
  - in hierarchical structure where node represent sets and inheritance propagate set level assertion down to individual.

5. Finding Right structure :

- Given a large amount of knowledge stored in a database, how can relevant parts are accessed when they are needed?
- This is about access to right structure for describing a particular situation.
- This requires, selecting an initial structure and then revising the choice.
- While doing so, it is necessary to solve following problems :
  - how to perform an initial selection of the most appropriate structure.
  - how to fill in appropriate details from the current situations.
  - how to find a better structure if the one chosen initially turns out not to be appropriate.
  - what to do if none of the available structures is appropriate.
  - when to create and remember a new structure.
- There is no good, general purpose method for solving all these problems. Some knowledge representation techniques solve some of these issues.

Logic

- The logical formalism of a language is useful because it immediately suggests a powerful way of deriving new knowledge from old using mathematical deduction.
- In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known.

Proposition

- A proposition is a statement, or a simple declarative sentence.
- For example, "the book is expensive" is a proposition.
- A proposition can be either true or false.

Propositional logic

- Logical constants: true, false
- Propositional symbols: P, Q, S,... (atomic sentences)
- Propositions are combined by connectives:

$\wedge$	*and* [conjunction]
$\vee$	*or* [disjunction]
$\Rightarrow$	*implies* [implication]
$\neg$	*not* [negation]
$\forall$	*For all*
$\exists$	*There exists*

- Propositional logic is a simple language useful for showing key ideas and definitions.
- User defines a set of propositional symbols, like P and Q.
- User defines the semantics of each propositional symbol:

  P means "It is hot"

  Q means "It is humid"

  R means "It is raining"

- A sentence (well-formed formula) is defined as follows:
  - A symbol is a sentence.
  - If S is a sentence, then $\neg$ S is a sentence.
  - If S is a sentence, then (S) is a sentence.
  - If S and T are sentences, then (S $\vee$ T), (S $\wedge$ T), (S $\rightarrow$ T), and (S $\leftrightarrow$ T) are sentences
  - A sentence results from a finite number of applications of the above rules.
- ***Real world facts can be represented by well-formed formulas (wffs) in propositional 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.
- 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 any 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?
- Propositional logic commits only to the existence of facts that may or may not be the case in the world being represented.
- 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. That is, all variables are "bound" by universal or existential quantifiers.

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

Quantifiers

- Universal quantification

  ($\forall$x)P(x) means that P holds for all values of x in the domain associated with that

variable

E.g., $(\forall x)$ dolphin(x) $\rightarrow$ mammal(x)

- Existential quantification

  $(\exists x)P(x)$ means that P holds for some value of x in the domain associated with that variable

  E.g., $(\exists x)$ mammal(x) $\wedge$ lays-eggs(x)

- 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. 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.

     $\forall x$: Pompeian(x) $\rightarrow$ Roman(x)
  4. Caesar was a ruler.

     ruler(Caesar)
  5. All Pompeians were either loyal to Caesar or hated him.

     inclusive-or

     $\forall x$: Roman(x) $\rightarrow$ loyalto(x, Caesar) $\vee$ hate(x, Caesar)

     exclusive-or

     $\forall x$: Roman(x) $\rightarrow$ (loyalto(x, Caesar) $\wedge \neg$hate(x, Caesar)) $\vee$
     ($\neg$loyalto(x, Caesar) $\wedge$ hate(x, Caesar))
  6. Every-one is loyal to someone.

     $\forall x$: $\exists y$: loyalto(x, y)
  7. People only try to assassinate rulers they are not loyal to.

     $\forall x$: $\forall y$: person(x) $\wedge$ ruler(y) $\wedge$ tryassassinate(x, y)
     $\rightarrow \neg$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?

- Now let's try to produce a formal proof, reasoning backward from the desired goal:

  ¬ loyalto(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 be transformed, and so on, until there are no unsatisfied goals remaining.

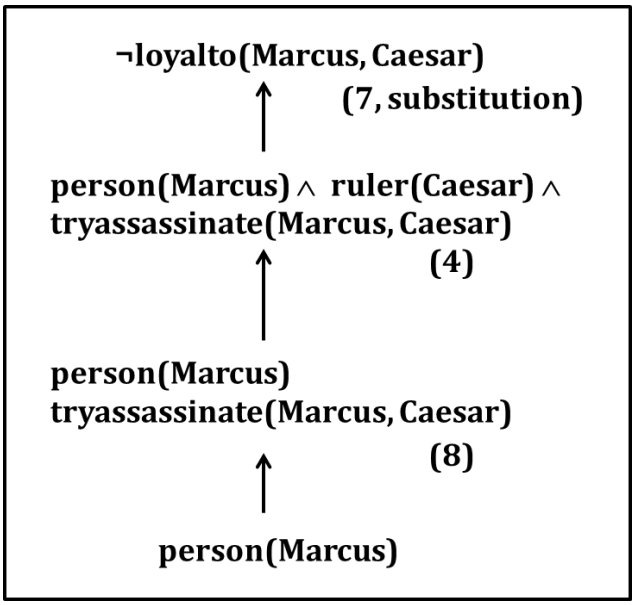


Figure 4.1 An attempt to prove ¬loyalto(Marcus, Caesar).

- 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. 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.
- From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:
  1. Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
  2. There is often a choice of how to represent the knowledge. Simple representations are desirable, but they may exclude certain kinds of reasoning.
  3. 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, 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 important role particularly in a useful form of reasoning called property inheritance.
- ***The predicates instance and isa explicitly captured the relationships they are used to express, namely class membership and class inclusion.***
- Fig. 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 is represented with unary predicates (such as Roman), each of which corresponds to a class.
- Asserting that P(x) is true is equivalent to asserting that x is an instance (or element) of P.
- The second part of the figure contains representations that use the ***instance*** predicate explicitly.

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

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

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

Figure 4.2 Three ways of representing class membership

- 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, are 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:

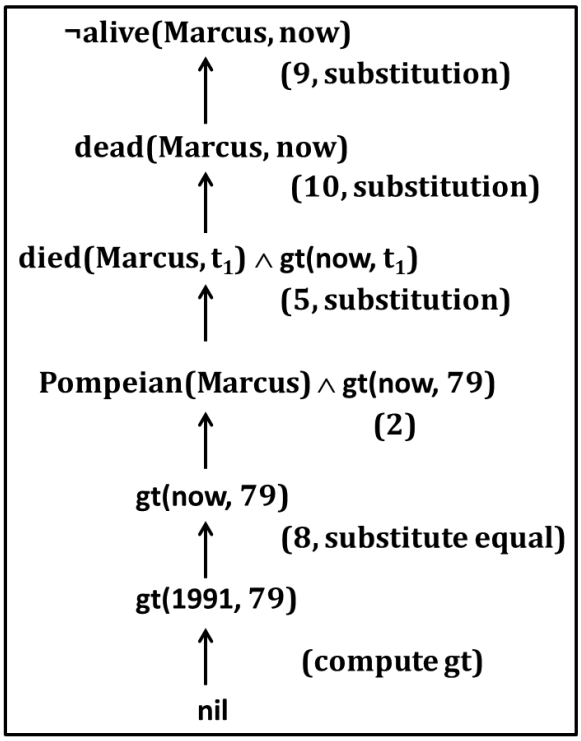
  gt(1,O) It(0,1)
  gt(2,1) It(1,2)
  gt(3,2) It(2,3)

- 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
- 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.
- 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.
- 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.
     $\forall x: \forall t: [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
  9. If someone dies, then he is dead at all later times.
     $\forall x: \forall t1: At2: died(x, t1) \wedge gt(t2, t1) \rightarrow dead(x, t2)$
- Now let's attempt to answer the question "Is Marcus alive?" by proving:
  ¬ alive(Marcus, now)



Resolution

- 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 into a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

  winter* V *summer

  ¬ winter* V *cold

  - Now we observe that precisely one of winter and ¬ winter will be true at any point.
  - If winter is true, then cold must be true to guarantee the truth of the second clause. If ¬ winter is true, then summer must be true to guarantee the truth of the first clause.
  - Thus we see that from these two clauses we can deduce

    summer V cold
  - This is the deduction that the resolution procedure will make.
  - Resolution operates by taking two clauses that each contains the same literal, in this example, ***winter***.
  - The literal must occur in positive form in one clause and in negative form in the other. The resolvent is obtained by combining all of the literals of the two parent clauses except the ones that cancel.
  - If the clause that is produced is the empty clause, then a contradiction has been found. *For example, the two clauses*

    winter

    ¬ winter

    will produce the empty clause.

Conversion to Clause form

- To apply resolution, we need to reduce a set of *wff's* to a set of clauses, where a clause is defined to be a *wff* in conjunctive normal form but with no instances of the connector A.
- We can do this by first converting each wff into conjunctive normal form and then breaking apart each such expression into clauses, one for each conjunct.
- All the conjuncts will be considered to be conjoined together as the proof procedure operates. To convert a *wff* into clause form, perform the following sequence of steps.

Algorithm: Convert to Clause Form

1. Eliminate →, using the fact that a → b is equivalent to ¬ a V b. Performing this transformation on the wff given above becomes

   ∀x: ¬ [Roman(x) ∧ know(x, Marcus)] V

   [hate(x, Caesar) V (∀y : ¬(∃z : hate(y, z)) V thinkcrazy(x, y))]

2. Reduce the scope of each ¬ to a single term, using the fact that ¬ (¬ p) = p, Performing this transformation on the wff from step 1 becomes

   ∀x: [¬ Roman(x) V ¬ know(x, Marcus)] V
   [hate(x, Caesar) V (∀y: ∀z: ¬ hate(y, z) V thinkcrazy(x, y))]

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff.

   For example, the formula ∀x: P(x) V ∀x: Q(x)
   would be converted to ∀x: P(x) V ∀y: Q(y)

4. Move all quantifiers to the left of the formula without changing their relative order. This is possible since there is no conflict among variable names. Performing this operation on the formula of step 2, we get

   ∀x: ∀y: Az: [¬ Roman(x) V ¬ know(x, Marcus)] V
   [hate(x, Caesar) V (¬ hale(y, z) V thinkcrazy(x, y))]

5. Eliminate existential quantifiers. So, for example, the formula

   ∃y : President(y)

   can be transformed into the formula

   President(S1)

   For example, in the formula

   ∀x: ∃y: father-of(y, x)

   would be transformed into

   ∀x: father-of(S2(x), x)

   These generated functions are called Skolem functions. Sometimes ones with no arguments are called Skolem constants.

6. Drop the prefix. At this point, all remaining variables are universally quantified, so the prefix can just be dropped and any proof procedure we use can simply assume that any variable it sees is universally quantified. Now the formula produced in step 4 appears as

   [¬ Roman(x) V ¬ know(x, Marcus)] V
   [hate(x, Caesar) V (¬ hate(y, z) V thinkcrazy(x, y))]

7. Convert the matrix into a conjunction of disjuncts. In the case or our example, since there are no and's, it is only necessary to exploit the associative property of or [i.e., (a ∧ b) V c = (a V c) ∧ (b ∧ c)] and simply remove the parentheses, giving

   ¬ Roman(x) V ¬ know(x, Marcus) V
   hate(x, Caesar) V ¬ hate(y, z) V thinkcrazy(x, y)

8. Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true.

9. Standardize apart the variables in the set of clauses generated in step 8. By this we mean rename the variables so that no two clauses make reference to the same variable. In making this transformation, we rely on the fact that

   (∀x: P(x) ∧ Q(x)) = ∀x: P(x) ∧ ∀x: Q(x)

Thus since each clause is a separate conjunct and since all the variables are universally

quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Algorithm: 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:
    a. Select two clauses. Call these the parent clauses.
    b. 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.
    c. If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

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 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:
    a. If L1 and L2 are identical, then return NIL.
    b. Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
    c. 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 number of arguments in L1 :

   a. Call Unify with the i^{th} argument of L1 and the i^{th} argument of L2, putting result in S.
   b. If S contains FAIL then return {FAIL}.
   c. If S is not equal to NIL then:
      i. Apply S to the remainder of both L1 and L2.
      ii. 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 is found, no progress can be made, or a predetermined amount of effort has been expended.
   a. Select two clauses. Call these the parent clauses.
   b. Resolve them together. The resolvent will be 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 be omitted from the resolvent.
   c. If the resolvent is an empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Example: Consider the following facts. Translate given sentences into formulas in predicate logic. Prove by resolution the given fact.

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All pompeians were Romans.
4. Caesar was a ruler.
5. All Romans 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.

Prove: hate (Marcus, Caesar)

Solution:

Predicate logic statements.

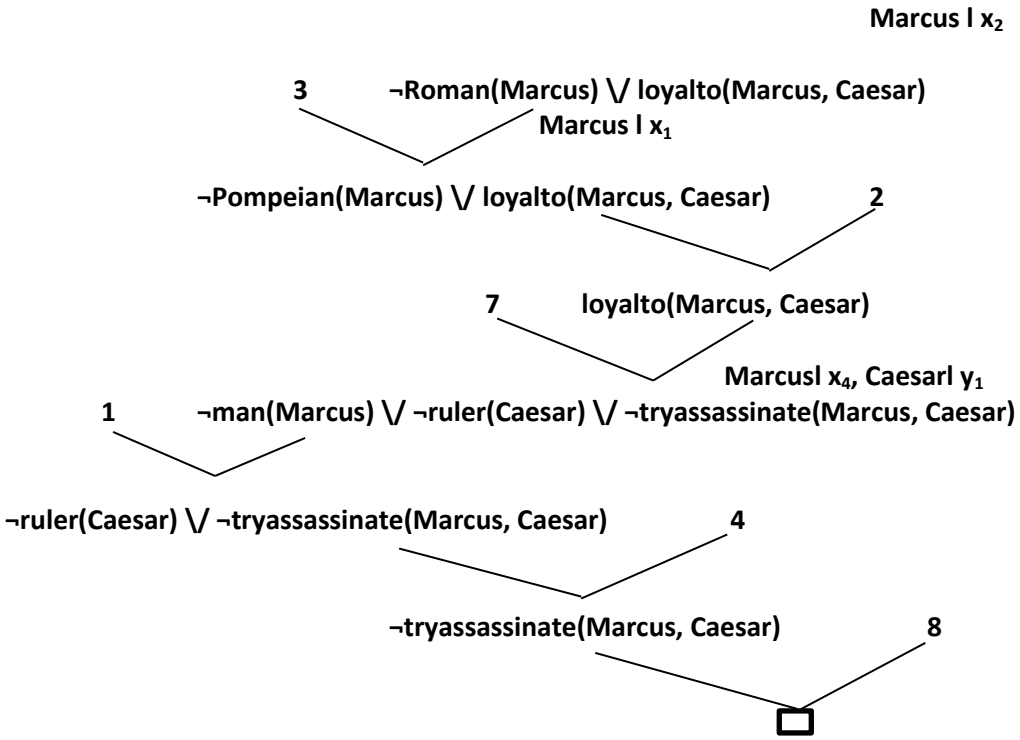
1. Marcus was a man.
   man(Marcus)
2. Marcus was a Pompeian.
   Pompeian(Marcus)
3. All Pompeians were Romans.
   $\forall x$: Pompeian(x) $\rightarrow$ Roman(x)
4. Caesar was a ruler.
   ruler(Caesar)
5. All Pompeians were either loyal to Caesar or hated him.
   $\forall x$: Roman(x) $\rightarrow$ loyalto(x, Caesar) $\vee$ hate(x, Caesar)
6. Every-one is loyal to someone.
   $\forall x$: $\exists y$: loyalto(x, y)
7. People only try to assassinate rulers they are not loyal to.
   $\forall x$: $\forall y$: person(x) $\wedge$ ruler(y) $\wedge$ tryassassinate(x, y)
   $\rightarrow \neg$loyalto(x, y)
8. Marcus tried to assassinate Caesar.
   tryassassinate(Marcus, Caesar)

Axioms in clause form –

1. man(Marcus)
2. Pompeian(Marcus)
3. ¬Pompeian(x1) \/ Roman(x1)
4. ruler(Caesar)
5. ¬Roman(x2) \/ loyalto(x2, Caesar) \/ hate(x2, Caesar)
6. loyalto(x3,fl(x3))
7. ¬man(x4) \/ ¬ruler(y1) \/ ¬tryassassinate(x4,y1) \/ loyalto(x4,y1)
8. trayassassinate(Marcus,Caesar)

- **To prove hate (Marcus, Caesar) we start with a negation of it as ¬hate (Marcus, Caesar).**

Prove: hate (Marcus, Caesar) **¬hate (Marcus, Caesar)** **5**



- **Final resolvent is an empty clause means that a contradiction is found in the initial assumption so, proved hate (Marcus, Caesar).**

Introduction

- We have discussed various search techniques in previous units. Now we would consider a set of rules that represents,
  i. Knowledge about relationships in the world and
  ii. Knowledge about how to solve problem using the content of the rules.

Procedural versus Declarative Knowledge

Procedural Knowledge

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

  Man (Marcus)
  Man (Caesar)
  Person (Cleopatra)
  ∀x: Man(x) → 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 is needed, then the answer to the question will depend on the order in which the assertions are examined during the search for a response.
- If the assertions are declarative then they do not themselves say anything about how they will be examined. In case of procedural representation, they say how they will be examined.

Declarative Knowledge

- A statement in which knowledge is 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 be done to the knowledge and how.
- For example, a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems but in some cases the logical assertions can be viewed 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 be viewed as a procedural representation of knowledge.

Differences Between Declarative knowledge and procedural knowledge

Procedural knowledge	Declarative knowledge
High efficiency	Higher level of abstraction
Low modifiability	Good modifiability and good readability
Low perceptive adequacy (better for knowledge engineers)	Suitable for independent facts
Produces creative, reflective thought and promoters critical thinking and independent decision making	Good cognitive matching (better for domain experts and end-users) and low computational efficiency.

Logic Programming

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

- For example the PROLOG clause P(x): - Q(x, y) is equal to logical expression ∀x: ∃y: Q (x, y) → P(x).
- The difference between the logic and PROLOG representation is that the PROLOG interpretation has a fixed control strategy and so the assertions in the PROLOG program define a particular search path to answer to 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:

  A. Logical representation

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

  B. 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

- A search procedure must find a path between initial and goal states.
- There are two directions in which a search process could proceed.
- The two types of search are:
  1. Forward search which starts from the start state
  2. Backward search that starts from the goal state
- The production system views the forward and backward as symmetric processes.
- Consider a game of playing 8 puzzles. The rules defined are
  - *Square 1 empty and square 2 contains tile n.* →
    - *Square 2 empty and square 1 contains the tile n.*
  - *Square 1 empty Square 4 contains tile n.* →
    - *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 tree by finding all rules ***whose left hand side matches*** against the root node. The right hand side is used to create new

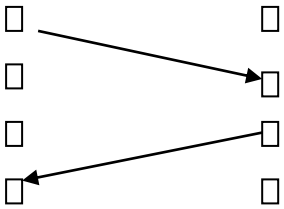
configurations.
    - Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose left hand side match.
2. Reasoning backward from the goal states:
    - Step 1. Begin building a tree of move sequences by starting with the goal node configuration at the root of the tree.
    - Step 2. Generate the next level of tree by finding all rules ***whose right hand side matches*** against the root node. The left 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 right hand side match.

- The same rules can be used in both cases.
- In forward reasoning the left hand sides of the rules are matched against the current state and right sides are used to generate the new state.
- In backward reasoning the right hand sides of the rules are 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 longer.
    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, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.
- Example 1: It is easier to drive from an unfamiliar place from home, rather than from home to an unfamiliar place. If you consider a home as starting place an unfamiliar place as a goal then we have to back track from unfamiliar place to home.
- Example 2: Consider a problem of symbolic integration. 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: The third factor is nothing but deciding whether the reasoning process can justify its reasoning. If it justifies then it can be applied. For example, doctors are usually unwilling to accept any advice from diagnostics process because it cannot explain its reasoning.
- Example 4: Prolog is an example of backward chaining rule system. In Prolog rules are restricted to Horn clauses. This allows for rapid indexing because all the rules for deducing a

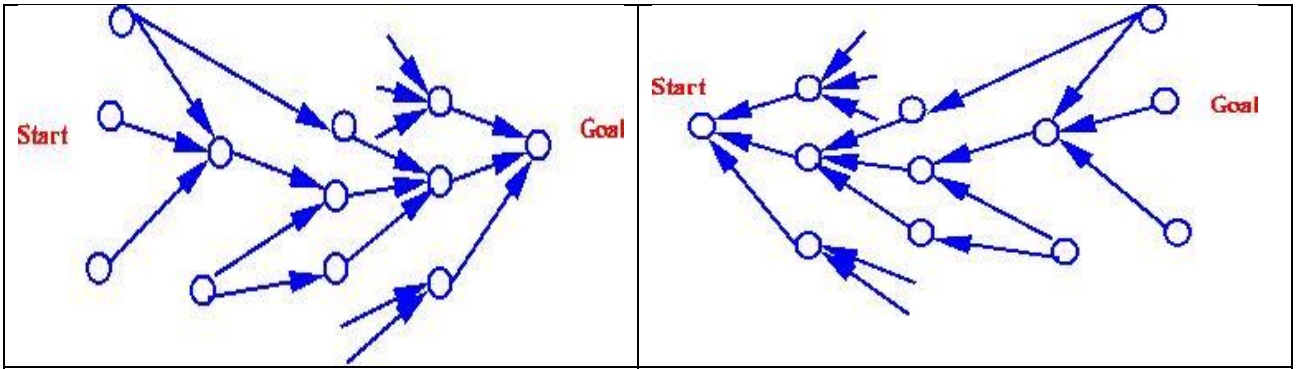
given fact share the same rule head. Rules are matched with unification procedure. Unification tries to find a set of bindings for variables to equate a sub-goal with the head of some rule. Rules in the Prolog program are matched in the order in which they appear.

Combining Forward and Backward Reasoning

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



- 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 be applied for both forward and backward reasoning.
- If left hand side and right of the rule contain pure assertions then the rule can be reversed. And so the same rule can be applied for both types of reasoning.
- If the right side of the rule contains an arbitrary procedure then the rule cannot be reversed.
- In this case while writing the rule the commitment to direction of reasoning must be made.
-



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

What is Reasoning

- Reasoning is the act of deriving a conclusion from certain properties using a given methodology.
- Reasoning is a process of thinking; reasoning is logically arguing; reasoning is drawing inference.
- ***When a system is required to do something, that it has not been explicitly told how to do, it must reason. It must figure out what it needs to know from what it already knows.***
- Many types of Reasoning have been identified and recognized, but many questions regarding their logical and computational properties still remain controversial.
- The popular methods of Reasoning include abduction, induction, model-based, explanation and confirmation. All of them are intimately related to problems of belief revision and theory development, knowledge absorption, discovery and learning.

Logical Reasoning

- Logic is a language for reasoning. It is a collection of rules called Logic arguments, we use when doing logical reasoning.
- Logic reasoning is the process of drawing conclusions from premises using rules of inference.
- The study of logic is divided into formal and informal logic. The formal logic is sometimes called symbolic logic.
- Symbolic logic is the study of symbolic abstractions (construct) that capture the formal features of logical inference by a formal system.
- Formal system consists of two components, a formal language plus a set of inference rules.
- The formal system has axioms. Axiom is a sentence that is always true within the system.
- Sentences are derived using the system's axioms and rules of derivation are called theorems.
- The Logical Reasoning is of our concern in AI.

Approaches to Reasoning

- There are three different approaches to reasoning under uncertainties.
  1. Symbolic reasoning
  2. Statistical reasoning
  3. Fuzzy logic reasoning

Symbolic Reasoning

- The basis for intelligent mathematical software is the integration of the "power of symbolic mathematical tools" with the suitable "proof technology".
- Mathematical reasoning enjoys a property called monotonicity, that says, "If a conclusion follows from given premises A, B, C, … then it also follows from any larger set of premises, as long as the original premises A, B, C, … are included."
- Human reasoning is not monotonic.
- People arrive to conclusions only tentatively; based on partial or incomplete information,

reserve the right to retract those conclusions while they learn new facts. Such reasoning is non-monotonic, precisely because the set of accepted conclusions have become smaller when the set of premises is expanded.

Formal Logic

- The Formal logic is the study of inference with purely formal content, i.e. where content is made explicit.
- **Examples - Propositional logic and Predicate logic.**
- Here the logical arguments are a set of rules for manipulating symbols. The rules are of two types,
    1. Syntax rules: say how to build meaningful expressions.
    2. Inference rules: say how to obtain true formulas from other true formulas.
- Logic also needs semantics, which says how to assign meaning to expressions.

Uncertainty in Reasoning

- The world is an uncertain place; often the Knowledge is imperfect which causes uncertainty. Therefore reasoning must be able to operate under uncertainty.
- AI systems must have ability to reason under conditions of uncertainty.

Uncertainties	Desired action
Incompleteness of Knowledge	Compensate for lack of knowledge
Inconsistencies of Knowledge	Resolve ambiguities and contradictions
Changing Knowledge	Update the knowledge base over time

Monotonic Reasoning

- A reasoning process that moves in one direction only.
- The number of facts in the knowledge base is always increasing.
- The conclusions derived are valid deductions and they remain so.

A monotonic logic cannot handle,

a) Reasoning by default: because consequences may be derived only because of lack of evidence of the contrary.
b) Abductive reasoning: because consequences are only deduced as most likely explanations.
c) Belief revision: because new knowledge may contradict old beliefs.

Introduction to Non-monotonic Reasoning

- Non-monotonic reasoning (NMR) is based on supplementing absolute truth with beliefs.
- These tentative beliefs are generally based on default assumptions that are made in light of lack of evidence.
- A non-monotonic reasoning (NMR) system tracks a set of tentative beliefs and revises those beliefs when knowledge is observed or derived.
- The reason is, the human reasoning is non-monotonic in nature.
- This means, we reach to conclusions from certain premises that we would not reach if

certain other sentences are included in our premises.

- Conventional reasoning systems such as first order predicate logic are designed to work with information that has three important properties.
  1. It is complete with respect to the domain of interest.
  2. It is consistent.
  3. The only way it can change is that new facts can be added as they become available. If the new facts are consistent with all other facts that have already been asserted, then nothing will be ever retracted from the set of facts that are known to be true. This property is called monotonicity.
- Non-monotonic reasoning systems are designed to be able to solve problems in which above three properties may be missing.
- In order to do this, following key issues must be addressed.
  1. How can knowledge base be extended to allow inferences to be made on the basis of lack of knowledge as well as on the presence of it?
  2. How can the knowledge base be updated properly when a new fact is added to the system or when an old one is removed?
  3. How can knowledge be used to help resolve conflicts when there are several in consistent non-monotonic inferences that could be drawn?

Logics for Non-monotonic Reasoning

- A non-monotonic logic is a formal logic whose consequence relation is not monotonic.
- Logic is non-monotonic if the truth of a proposition may change when new information (axioms) is added.
- Allows a statement to be retracted.
- Used to formalize plausible (believable) reasoning.

Example 1 :

Birds typically fly. Tweety is a bird.
Tweety (presumably) flies.

- Conclusion of non-monotonic argument may not be correct.

Example-2 : (Ref. Example-1)

If Tweety is a penguin, it is incorrect to conclude that Tweety flies.

(Incorrect because, in example-1, default rules were applied when case-specific information was not available.)

- All non-monotonic reasoning are concerned with consistency.
- Inconsistency is resolved, by removing the relevant conclusion(s) derived by default rules, as shown in the example below.

Example -3 :

- The truth value (true or false), of propositions such as "Tweety is a bird" accepts default that is normally true, such as "Birds typically fly".

A conclusion derived was *"Tweety flies".*

- When an inconsistency is recognized, only the truth value of the last type is changed.

Methods of Reasoning

- Generally there are three kinds of logical reasoning: Deduction, Induction, Abduction.
    1. Deduction
       Example: "When it rains, the grass gets wet. It rains. Thus, the grass is wet."
        - This means in determining the conclusion; it is using rule and its preconditioned to make a conclusion.
        - Applying a general principle to a special case.
        - Using theory to make predictions
        - Usage: Inference engines, Theorem provers, planning.
    2. Induction
       Example: "The grass has been wet every time it has rained. Thus, when it rains, the grass gets wet."
        - This means in determining the rule; it is learning the rule after numerous examples of conclusion following the precondition.
        - Deriving a general principle from special cases
        - From observations to generalizations to knowledge
        - Usage: Neural nets, Bayesian nets, Pattern recognition
    3. Abduction
       Example: "When it rains, the grass gets wet. The grass is wet, it must have rained."
        - Means determining the precondition; it is using the conclusion and the rule to support that the precondition could explain the conclusion.
        - Guessing that some general principle can relate a given pattern of cases
        - Extract hypotheses to form a tentative theory
        - Usage: Knowledge discovery, Statistical methods, Data mining.

Default Reasoning

- This is a very common form of non-monotonic reasoning. The conclusions are drawn based on what is most likely to be true.
- There are two approaches; both are logic type, to Default reasoning: one is Non-monotonic logic and the other is Default logic.
    1. Non-monotonic logic
        - It has already been defined. It says, "the truth of a proposition may change when new information (axioms) are added and a logic may be built to allow the statement to be retracted."
        - Non-monotonic logic is predicate logic with one extension called modal operator M which means “consistent with everything we know”.
        - The purpose of M is to allow consistency.

- A way to define consistency with PROLOG notation is :

  To show that fact P is true, we attempt to prove ¬P.

  - If we fail we may say that P is consistent since ¬P is false.

  Example :

  $\forall x : plays\_instrument(x) \wedge M\ manage(x) \rightarrow jazz\_musician(x)$

  - States that for all x, the x plays an instrument and if the fact that x can manage is consistent with all other knowledge then we can conclude that x is a jazz musician.

2. Default Logic
   - Default logic initiates a new inference rule

$$\frac{A : B}{C}$$

   - Where, A is known as the prerequisite, B as the justification, and C as the consequent.
   - Read the above inference rule as: " if A, and if it is consistent with the rest of what is known to assume that B, then conclude that C ".
   - The rule says that given the prerequisite, the consequent can be inferred, provided it is consistent with the rest of the data.

   Example :

   Rule that "birds typically fly" would be represented as,

$$\frac{bird(x) : flies(x)}{flies(x)}$$

   - Which says, "If x is a bird and the claim that x flies is consistent with what we know, then infer that x flies".
   - The idea behind non-monotonic reasoning is to reason with first order logic, and if an inference cannot be obtained then use the set of default rules available within the first order formulation.

Introduction

- We have described several representation techniques that can be used to model belief systems in which, at any given point in time, a particular fact is believed to be true, believed to be false or not considered to be either.
- But for some kinds, it is useful to be able to describe beliefs that are not certain but for which there is some supporting evidence.
- For example, problems that contain genuine randomness. E.g. Card Games
- Problems that could, in principle be modeled using the techniques that we used in uncertainty.
- For such problems, statistical measures may serve a very useful function as summaries of the world; rather than looking for all possible exceptions we can use a numerical summary that tells us how often an exception of some sort can be expected to occur.
- This is useful for dealing with problems where there is randomness and unpredictability (such as in games of chance) and also for dealing with problems where we could, if we had sufficient information, work out exactly what is true.
- To do all this in a principled way requires techniques for probabilistic reasoning.

Statistical Reasoning

Probability & Bayes Theorem

- An important goal for many problem solving systems is to collect evidence as the system goes along and to modify its behavior on the basis of evidence.
- To model this behavior we need a statistical theory of evidence.
- Bayesian statistics is such a theory. The fundamental notion of Bayesian statistics is that of condition probability.
- Read the expression as the probability of Hypothesis H given that we have observed evidence E.
- To compute this, we need to take into account the prior probability of H (the probability that we would assign to H if we had no evidence) & the extent to which E provides evidence of H.
- To do this we need to define a universe that contains an exhaustive, mutually exclusive set of Hi's, among which we are trying to discriminate.

  P(Hi\E) = The probability that hypothesis Hi is true given evidence E

  P(E\Hi) = The probability that we will observe evidence E given that hypothesis i is true.

  P(Hi) = The a priori probability that hypothesis *i* is true in the absence of any specific evidence. These probabilities are called prior probabilities.

  K = The number of possible hypothesis.

  Bayers' theorem states then that,

$$P(H_i/E) = \frac{P(E/H_i) \cdot P(H_i)}{\sum_{n=1}^{k} P(E/H_n) \cdot P(H_n)}$$

Certainty Factors & Rule Based Systems

- In this we are trying to describe a practical way of compromising on a pure Bayesian System.
- We will use MYCIN as an example here which uses LISP and acts as an Expert System.
- MYCIN uses the rules to reason Backwards to the clinical data available from its goal of finding significant disease-causing organisms.
- Each rule has a certainty attached to it. Once the identities of the organism are found, it then attempts to select a therapy by which the disease can be treated.
- A certainty factor (CF [h, e])is defined in terms of two components:
  1. MB[h, e]- a measure (between 0 and 1) of belief in hypothesis "h" given the evidence "e".
     - MB measures the extent to which the evidence supports the hypothesis.
     - It is zero if the evidence fails to support the hypothesis.
  2. MD[h, e]- a measure (between 0 and 1) of disbelief in hypothesis "h" given the evidence "e".
     - MD measures the extent to which the evidence supports the negation of the hypothesis. It is zero if the evidence support the hypothesis.

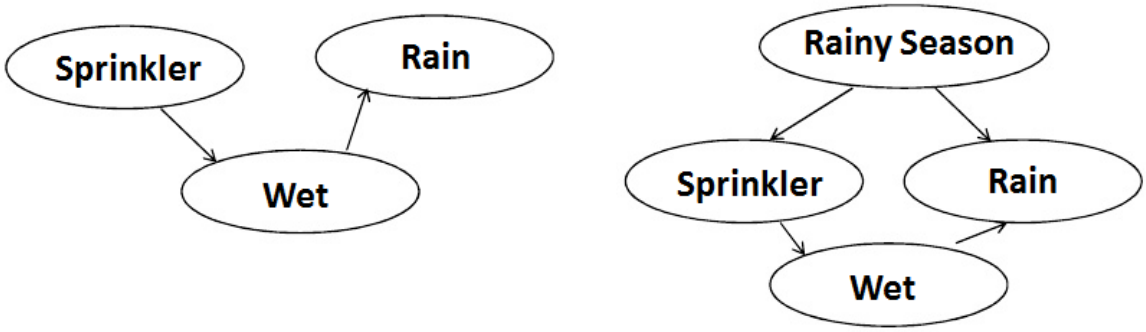
CF[h, e] = MB[h, e]- MD[h, e]

- Since any particular piece of evidence either supports or denies a Hypothesis (but not both) and since each MYCIN rule corresponds to one piece of evidence (can be compound piece of evidence) a single number suffices for each rule to define both MB and MD and thus the CF.
- The CFs are provided by the experts who write the rules. They reflect the experts' assessments of the strength of the evidence in support of the Hypothesis.
- At times CFs need to be combined to reflect the operation of multiple pieces of evidence and multiple rules applied to a problem.
- The combining functions should satisfy the following properties :
- Since the order in which evidence is collected is arbitrary, the combining functions should be commutative and associative.
- Until certainty is reached, additional confirming evidence should increase MB and similarly for disconfirming evidence, MD.
- If uncertain inference are chained together, then the result should be less certain that either of the inferences alone.
- By use of CFs we reduce the complexity of a Bayesian reasoning system by making some approximations to the formalism.

Bayesian Networks

- It is an alternative approach to what we did in the previous section.
- The idea is to describe the real world, it is not necessary to use a huge joint probability table in which we list the probabilities of all combinations, because most events are independent of each other, there is no need to consider the interactions between them.
- We will use a more local representation in which we will describe clusters of events that interact.
- Let us consider an example as,

  S: sprinkler was on last night
  W: grass is wet
  R: it rained last night



- There are two different ways that propositions can influence the likelihood of each other.
  1. The first is that causes influence to the likelihood of their symptoms.
  2. The second is that observing a symptom affects the likelihood of all its possible causes.
- The main idea behind the Bayesian network structure is to make a clear distinction between these two kinds of influences.
- The Graph in above figure is known as DAG (Directed A cyclic Graph) that represents causality relationships among the variables.
- However, we need more information. In particular, we need to know, for each value of a parent node, what evidence is provided about the values that the child node can take on.
- We create a probability table for that.
- When the most genuine reasoning does not seem true then we need to use an undirected graph in which the arcs can be used to transmit probabilities in either direction, depending upon where the evidence came from.
- We have to ensure that there is no cycle that exists between the evidences.
- Three algorithms are available for doing these computations
  1. A message-passing method.
  2. A cliché triangulation method.

3. A variety of stochastic algorithms.

- The message-passing approach is based on the observation that to compute the probability of a node A given what is known about other nodes in the network.
- It is necessary to know three things.
  - The total support arriving at A from its parents.
  - The total support arriving at A from its children.
  - The entry in the fixed condition probability matrix that relates A to its causes.
- Cliché triangulation method.
  - Explicit arcs are introduced between pair of nodes that share a common descendent.
- Stochastic Algorithm or Randomized Algorithms
  - The idea is to shield a given node probabilistically from most of the other nodes in the network. These algorithms run faster but may not give correct results.

Dempster – Shafer Theory

- Till now, we have talked about individual propositions and assign to each of them a single number of the degree of belief that is warranted given the evidence.
- In Dempster-Shafer Theory we consider sets of propositions and assign to each of them an interval in which the degree of belief must lie.

[Belief, Plausibility]

- Belief (denoted as Bel) measures the strength of the evidence in favor of a set of propositions.
- It ranges from 0 (no evidence) to 1 (definite certainty)
- Plausibility (Pl) is,

Pl(s) = 1- Bel(¬s)

- It also ranges from 0 to 1 and measures the extent to which evidence in favor of ¬s leaves room for belief in s.
- In short if we have certain evidence in favor of ***(¬s)***, then Bel(not(s)) will be 1 and Pl(s) will be 0.
- This tells us that the only possible value for Bel(s) is also 0.
- The interval, also tells about the amount of information that we have.
- If we have no evidence we say that the hypothesis is in the range of [0, 1].
- As we gain more evidence, this interval can be expected to shrink, and giving confident answers. This is different from Bayesian as in that we would probably begin by distributing the probability among the hypotheses (.33 in case of 3) and we may end up with the same probability at the end, but in Dempster we state that we have no information at the start and everything is in the range of 0 to 1.
- Let's take an example where we have some mutually exclusive hypothesis.
- {Allergy, Flu, Cold, Pneumonia}, The set is denoted by θ.

- We want to attach some measure of belief to elements of θ.
- But all evidences are not directly supportive of individual elements.
- The key function we use here is a Probability Density Function, denoted by m.
- The function m, is not only defined for elements of θ but also all subsets of it.
- The quantity m(p) measures the amount of belief that is currently assigned to exactly to the set "p" of hypothesis.
- If θ contains n elements then there are 2^n subsets of θ.
- We must assign m so that the sum of all the m values assigned to subsets of θ is 1.
- At the beginning we have m as under θ = (1.0)
- If we get an evidence of .6 magnitude that the correct diagnosis is in the set {Flu, Cold, Pneu} then,

  {Flu, Cold, Pneu} = (0.6)

  θ = (0.4)
- This does not lead us anywhere, we need more information.
- Now to move further, let's consider we have two belief function m_1 and m_2.
- Let X be the set of subsets of θ to which m_1 assigns a nonzero value and let Y be the corresponding set for m_2.
- We define m_3, as a combination of the m_1 and m_2 to be,

$$\mathbf{m_3(Z)} = \frac{\sum_{X \cap Y = Z} \mathbf{m_1(X) \cdot m_2(Y)}}{\mathbf{1} - \sum_{X \cap Y = \emptyset} \mathbf{m_1(X) \cdot m_2(Y)}}$$

- For example, suppose m_1 corresponds to our belief after observing fever:

  m_1= { F, C, P} = 0.6

  θ = (0.4)
- suppose m_2 corresponds to our belief after observing runny nose:

  m2= { A ,F, C} =0.8

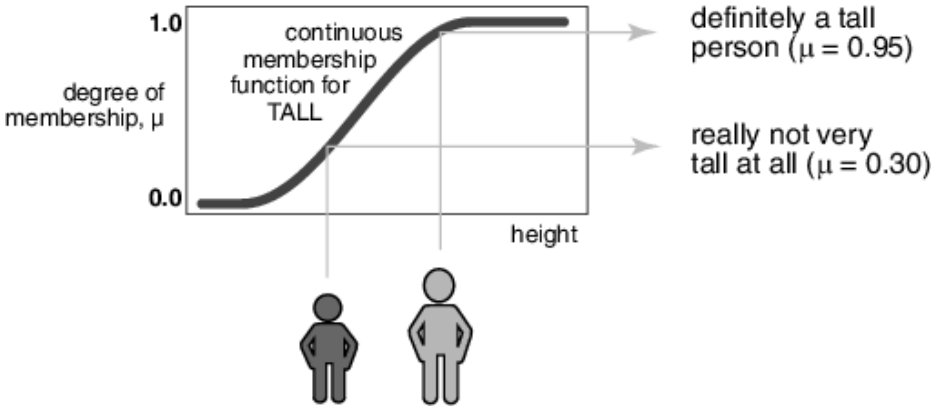
  θ = 0.2
- Then we can compute their combination m_3 using the following table.

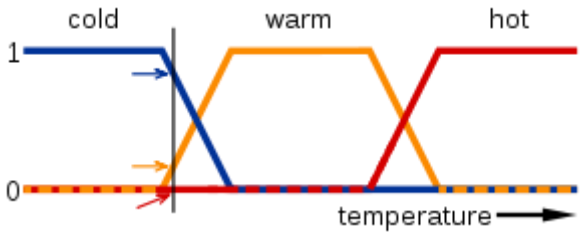
		{A, F, C}	(0.8)	Θ	(0.2)
{F, C, P}	(0.6)	{F, C}	(0.48)	{F, C, P}	(0.12)
Θ	(0.4)	{A, F, C}	(0.32)	Θ	(0.08)

Fuzzy Logic

- Fuzzy logic is a form of many-valued logic in which the truth values of variables may be any real number between 0 and 1.

- By contrast, in Boolean logic, the truth values of variables may only be the integer values 0 or 1.
- Fuzzy logic has been employed to handle the concept of partial truth, where the truth value may range between completely true and completely false.
- Furthermore, when linguistic variables are used, these degrees may be managed by specific (membership) functions.
- In the standard Boolean definition for tall people are either tall or not and there must be a specific height that defines the boundary.
- Once set membership has been redefined in this way, it is possible to define a reasoning system based on the techniques for combining distributions.
- Such methods have been used in control systems for devices like trains and washing machines.





- In above image, the meanings of the expressions cold, warm, and hot are represented by functions mapping a temperature scale.
- A point on that scale has three "truth values"—one for each of the three functions.
- The vertical line in the image represents a particular temperature that the three arrows (truth values) gauge.
- Since the red arrow points to zero, this temperature may be interpreted as "not hot". The orange arrow (pointing at 0.2) may describe it as "slightly warm" and the blue arrow (pointing at 0.8) "fairly cold".

Introduction

- Inheritance property can be represented using ***isa*** and ***instance*** links.
- Monotonic Inheritance can be performed substantially more efficiently with such structures than with pure logic, and non-monotonic inheritance is also easily supported.
- The reason that makes Inheritance easy is that the knowledge in slot and filler systems is structured as a set of entities and their attributes.
- These structures turn out to be useful as,
    - It indexes assertions by the entities they describe. As a result, retrieving the value for an attribute of an entity is fast.
    - It makes easy to describe properties of relations. To do this in a purely logical system requires higher-order mechanisms.
    - It is a form of object-oriented programming and has the advantages that such systems normally include modularity and ease of viewing by people.
- Here we would describe two views of this kind of structure - Semantic Nets & Frames.

Semantic Nets

- There are different approaches of knowledge representation include semantic net, frames and script.
- The semantic net describes both objects and events.
- In a semantic net, information is represented as a set of nodes connected to each other by a set of labeled arcs, which represents relationships among the nodes.
- It is a directed graph consisting of vertices which represent concepts and edges which represent semantic relations between the concepts.
- It is also known as associative net due to the association of one node with other.
- The main idea is that the meaning of the concept comes from the ways in which it is connected to other concepts.
- We can use inheritance to derive additional relations.

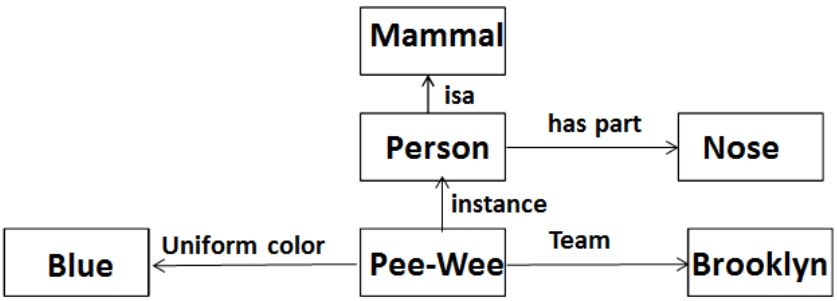


Figure 8.1 A Semantic Network

Intersection Search

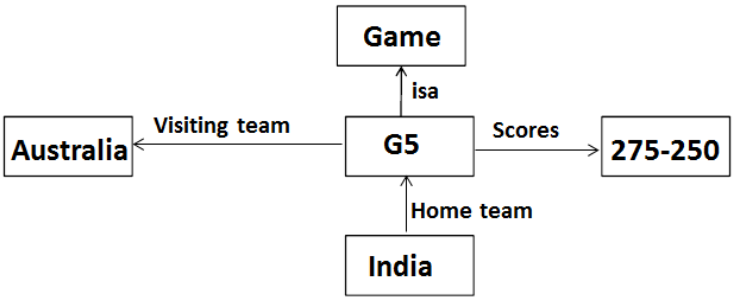
- We try to find relationships among objects by spreading activation out from each of two nodes and seeing where the activation meets.

- Using this we can answer the questions like, *what is the relation between India and Blue*.
- It takes advantage of entity based organization of knowledge that slot and filler representation provides.

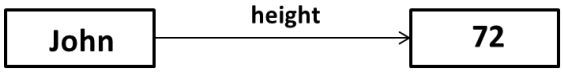
Representing Non-binary Predicates

- Simple binary predicates like isa(Person, Mammal) can be represented easily by semantic nets but other non-binary predicates can also be represented by using general-purpose predicates such as ***isa*** and ***instance***.
- Three or even more place predicates can also be converted to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe relationship to this new object.

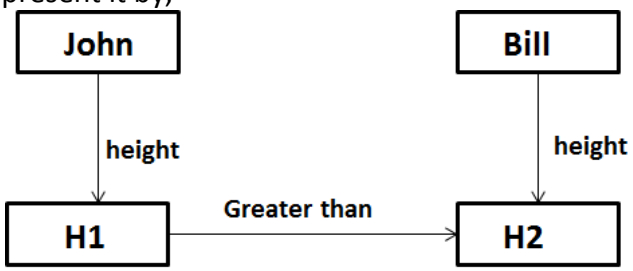
Example score(Australia, India, 250-275)



- Making important distinctions



- These entities are independent of each other, now we want to prove that John is taller than Bill then we can represent it by,



Partitioned Semantic Nets

- If we want to represent simple quantified expressions in semantic nets then it can be done with the help of partitioning the semantic net into a hierarchical set of spaces, each of which corresponds to the scope of one or more variable.
- For example,

The dog bit the mail carrier.

- This is simple and can be easily represented as shown in figure 8.2 (a).
- In this figure d, b and m represent a particular dog, a particular biting and a particular mail

carrier.

- Now suppose we want to represent following fact,

Every dog has bitten a mail carrier.

- To represent this fact, it is necessary to encode the scope of the universally quantified variable x. this can be done using partitioning as shown in figure 8.2 (b).
- The node g is given for the assertion. Node g is an instance of the special case GS of general statements about the world.
- Every element of GS has at least two attributes: a form and one or more ∀ connections.

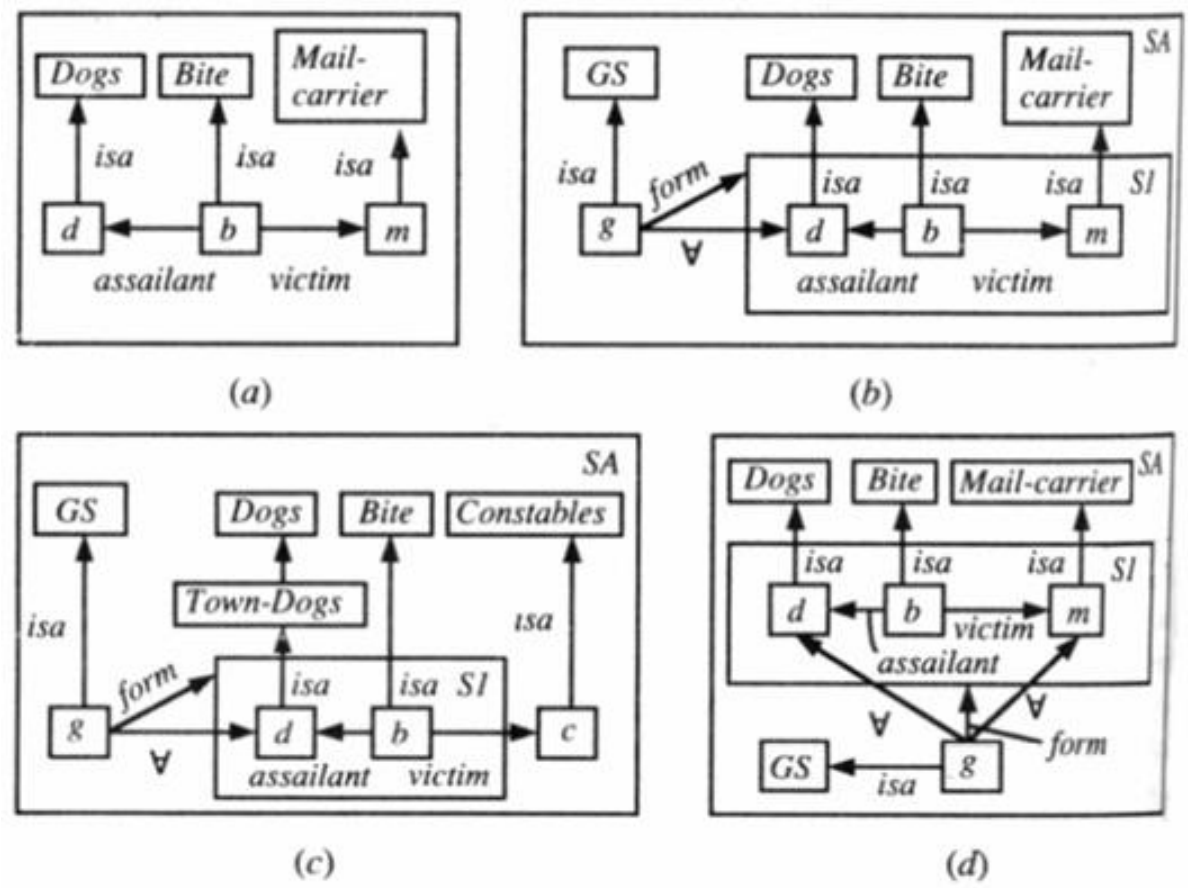


Figure 8.2 Partitioned Semantic Net

- Figure 8.2 (c) represents the similar sentences,

Every dog in town has bitten the constable.

- Figure 8.2 (d) represents,

Every dog has bitten every mail carrier.

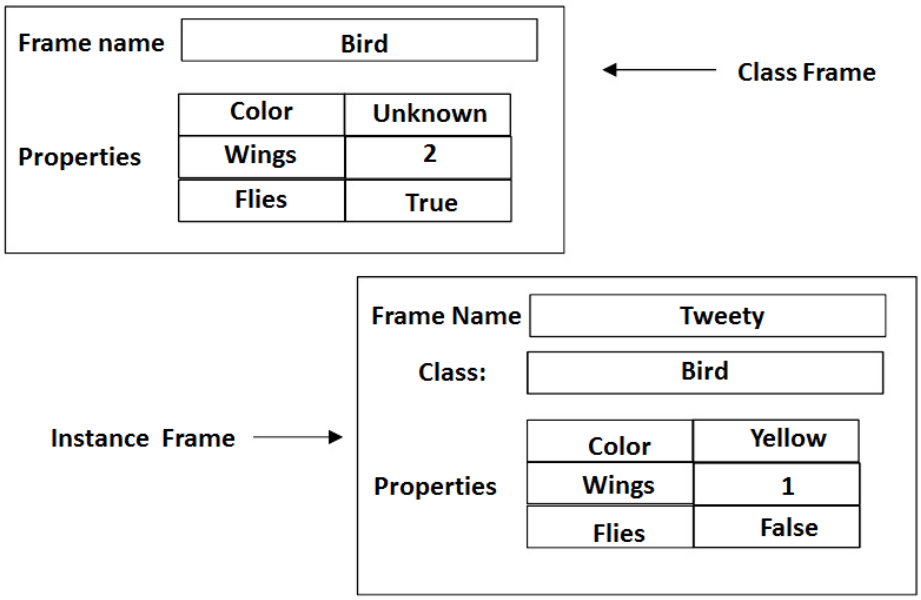
Evolution of Frames

- As seen in the previous example, there are certain problems which are difficult to solve with Semantic Nets.
- Although there is no clear distinction between a semantic net and frame system, but more structured the system is, more likely it is to be termed as a frame system.

- A frame is a collection of attributes (called slots) and associated values that describe some entities in the world. Sometimes a frame describes an entity in some absolute sense;
- Sometimes it represents the entity from a particular point of view only.
- A single frame taken alone is rarely useful; we build frame systems out of collections of frames that are connected to each other by virtue of the fact that the value of an attribute of one frame may be another frame.

Frames as Sets and Instances

- The set theory is a good basis for understanding frame systems.
- Each frame represents either a class (a set) or an instance (an element of class)
- Both ***isa*** and ***instance*** relations have inverse attributes, which we call subclasses & all-instances.
- As a class represents a set, there are 2 kinds of attributes that can be associated with it.
    1. Its own attributes &
    2. Attributes that are to be inherited by each element of the set.



Frames as Sets and Instances

- Sometimes, the difference between a set and an individual instance may not be clear.
- Example: Team India is an instance of class of Cricket Teams and can also be thought of as set of players.
- Now the problem is if we present Team India as a sub class of Cricket teams, then Indian players automatically become part of all the teams, which is not true.
- So, we can make Team India a sub class of class called Cricket Players.
- To do this we need to differentiate between regular classes and meta-classes.
- Regular Classes are those whose elements are individual entities whereas Meta-classes are those special classes whose elements are themselves classes.

- The most basic meta-class is the class *CLASS*.
- It represents the set of all classes.
- All classes are instances of it, either directly or through one of its subclasses.
- The class *CLASS* introduces the attribute cardinality, which is to be inherited by all instances of CLASS. Cardinality stands for number.

Other ways of Relating Classes to Each Other

- We have discussed that a class1 can be a subset of class2.
- If Class2 is a meta-class then Class1 can be an instance of Class2.
- Another way is ***mutually-disjoint-with*** relationship, which relates a class to one or more other classes that are guaranteed to have no elements in common with it.
- Another one is, ***is-covered-by*** which relates a class to a set of subclasses, the union of which is equal to it.
- If a class is-covered-by a set S of mutually disjoint classes, then S is called a partition of the class.

Slots as Full-Fledged Objects (Frames)

- Till now we have used attributes as slots, but now we will represent attributes explicitly and describe their properties.
- Some of the properties we would like to be able to represent and use in reasoning include,
    - The class to which the attribute can be attached.
    - Constraints on either the type or the value of the attribute.
    - A default value for the attribute. Rules for inheriting values for the attribute.
    - To be able to represent these attributes of attributes, we need to describe attributes (slots) as frames.
    - These frames will be organized into an ***isa*** hierarchy, just as any other frames are, and that hierarchy can then be used to support inheritance of values for attributes of slots.
    - Now let us formalize what is a slot. A slot here is about a relation.
    - It maps from elements of its domain (the classes for which it makes sense) to elements of its range (its possible values).
    - A relation is a set of ordered pairs.
    - Thus it makes sense to say that relation R1 is a subset of another relation R2.
    - In that case R1 is a specialization of R2. Since a slot is a set, the set of all slots, which we will call SLOT, is a meta-class.
    - Its instances are slots, which may have sub slots.

Frame Example

- In this example, the frames Person, Adult-Male, ML-Baseball-Player (corresponding to major league baseball players), Pitcher, and ML-Baseball-Team (for major league baseball team) are all classes.

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

- The frames Pee-Wee-Reese and Brooklyn-Dodgers are instances.
- The ***isa*** relation that we have been using without a precise definition is in fact the subset relation. The set of adult males is a subset of the set of people.
- The set of major league baseball players is a subset of the set of adult males, and so forth.
- Our instance relation corresponds to the relation element-of Pee Wee Reese is an element of the set of fielders.
- Thus he is also an element of all of the supersets of fielders, including major league baseball players and people. The transitivity of *isa* follows directly from the transitivity of the subset relation.
- Both the isa and instance relations have inverse attributes, which we call subclasses and all-instances.
- Because a class represents a set, there are two kinds of attributes that can be associated with it.
- Some attributes are about the set itself, and some attributes are to be inherited by each element of the set.
- We indicate the difference between these two by prefixing the latter with an asterisk (*).

- For example, consider the class ML-Baseball-Player, we have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624.
- We have listed five properties that all major league baseball players have (height, bats, batting-average, team, and uniform-color), and we have specified default values for the first three of them.
- By providing both kinds of slots, we allow a both class to define a set of objects and to describe a prototypical object of the set.
- Frames are useful for representing objects that are typical to stereotypical situations.
- The situation like structure of complex physical objects, visual scenes, etc.
- A commonsense knowledge can be represented using default values if no other value exits. Commonsense is generally used in the absence of specific knowledge.

Introduction

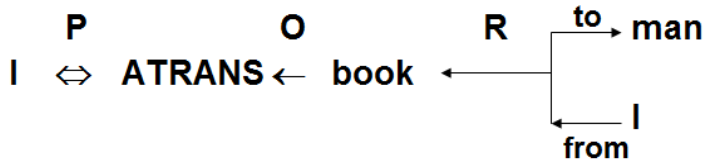
- The main problem with semantic networks and frames is that they lack formality; there is no specific guideline on how to use the representations.
- In frame when things change, we need to modify all frames that are relevant – this can be time consuming.
- Strong slot and filler structures typically represent links between objects according to more rigid rules, specific notions of what types of object and relations between them are provided and represent knowledge about common situations.
- We have types of strong slot and filler structures:
  1. Conceptual Dependency (CD)
  2. Scripts
  3. Cyc

1. Conceptual Dependency (CD)

- Conceptual Dependency originally developed to represent knowledge acquired from natural language input.
- The goals of this theory are:
  - To help in the drawing of inference from sentences.
  - To be independent of the words used in the original input.
  - That is to say: For any 2 (or more) sentences that are identical in meaning there should be only one representation of that meaning.
- It has been used by many programs that portend to understand English (MARGIE, SAM, PAM).
- Conceptual Dependency (CD) provides:
  - A structure into which nodes representing information can be placed.
  - A specific set of primitives.
  - A given level of granularity.
- Sentences are represented as a series of diagrams depicting actions using both abstract and real physical situations.
  - The agent and the objects are represented.
  - The actions are built up from a set of primitive acts which can be modified by tense.
- CD is based upon events and actions. Every event (if applicable) has:
  - an ACTOR
  - an ACTION performed by the Actor
  - an OBJECT that the action performs on
  - a DIRECTION in which that action is oriented
- These are represented as slots and fillers. In English sentences, many of these attributes are left out.

A Simple Conceptual Dependency Representation

- For the sentences, "I gave a book to the man" CD representation is as follows:



- Where the symbols have the following meaning.
  - Arrows indicate directions of dependency.
  - Double arrow indicates two way link between actor and action.
  - O -- for the object case relation
  - R – for the recipient case relation
  - P – for past tense
  - D - destination

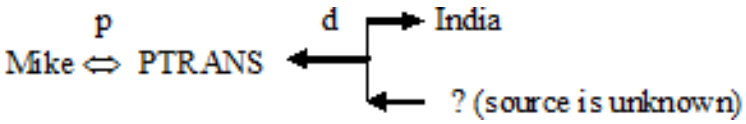
Primitive Acts of CD theory

ATRANS	Transfer of an abstract relationship (i.e. give)
PTRANS	Transfer of the physical location of an object (e.g., go)
PROPEL	Application of physical force to an object (e.g. push)
MOVE	Movement of a body part by its owner (e.g. kick)
GRASP	Grasping of an object by an action (e.g. throw)
INGEST	Ingesting of an object by an animal (e.g. eat)
EXPEL	Expulsion of something from the body of an animal (e.g. cry)
MTRANS	Transfer of mental information (e.g. tell)
MBUILD	Building new information out of old (e.g decide)
SPEAK	Producing of sounds (e.g. say)
ATTEND	Focusing of a sense organ toward a stimulus (e.g. listen)

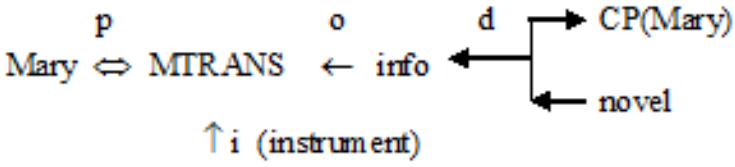
- There are four conceptual categories. These are,

ACT	Actions {one of the CD primitives}
PP	Objects {picture producers}
AA	Modifiers of actions {action aiders}
PA	Modifiers of PP's {picture aiders}

Example 1: Mike went to India



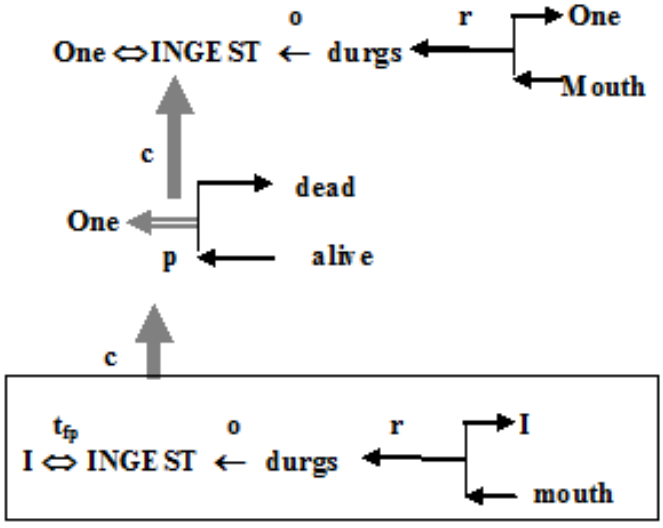
Example 2: Mary read a novel



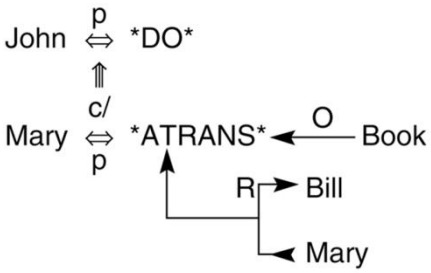
- The primitives are used to define conceptual dependency relationships or Conceptual syntax rules.

PP ⇔ ACT	indicates that an actor acts.
PP ⇔ PA	indicates that an object has a certain attribute.
ACT ←(O) PP	indicates the object of an action.
ACT ←(R) PP, PP	indicates the recipient and the donor of an object within an action.
ACT ←(D) PP, PP	indicates the direction of an object within an action.
ACT ←(1) ⇕	indicates the instrumental conceptualization for an action.
X ⇑ Y	indicates that conceptualization X caused conceptualization Y. When written with a C this form denotes that X COULD cause Y.
PP ⇐ PA2, PA1	indicates a state change of an object.
PP1 ← PP2	indicates that PP2 is either PART OF or the POSSESSOR OF PP1.

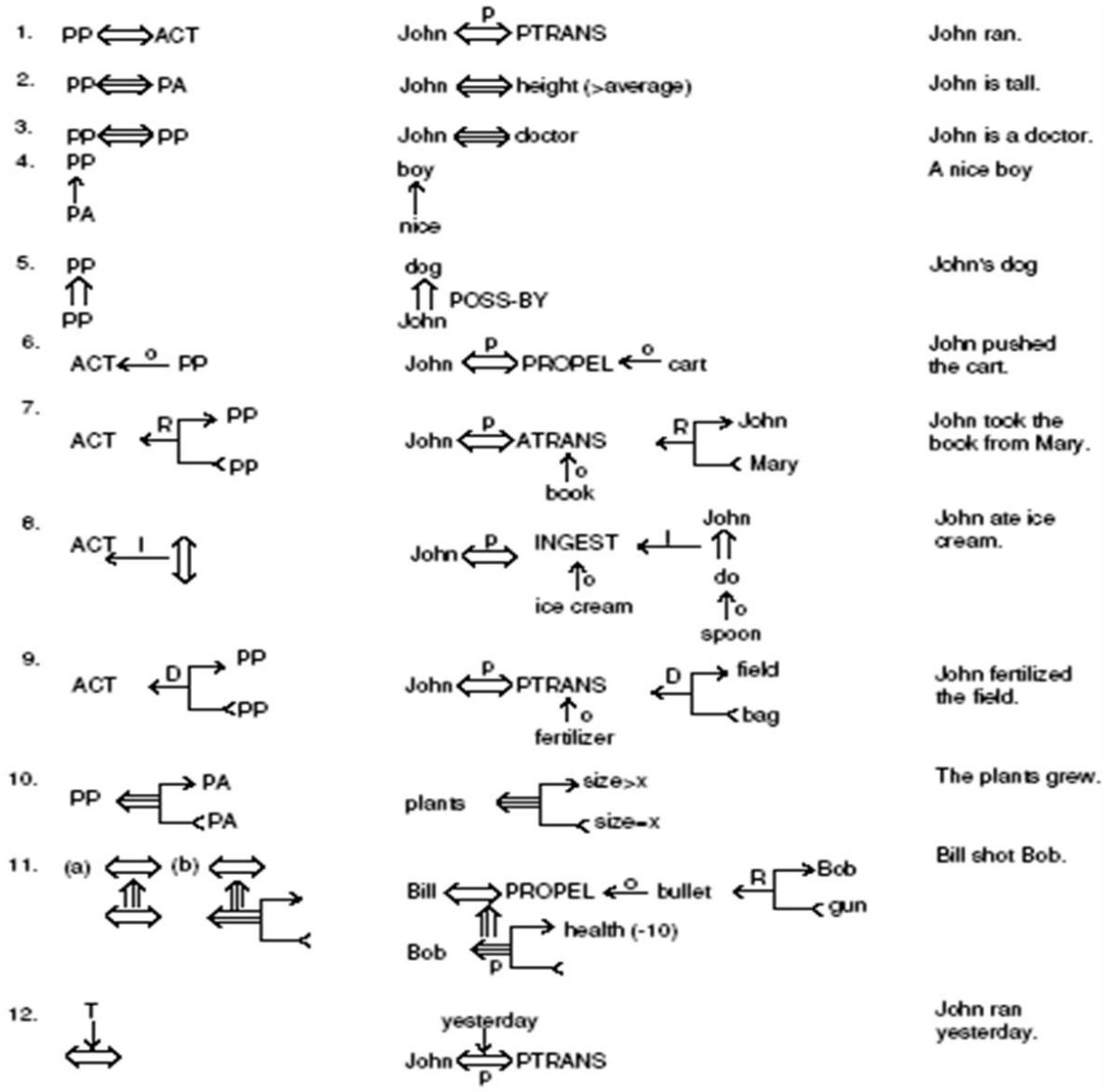
Example 3: Since drugs can kill, I stopped.



Example 4: John prevented Mary from giving a book to Bill



- Some basic conceptual dependencies and their use in representing more complex English sentences.



- Advantages of CD:
  - Using these primitives involves fewer inference rules.
  - Many inference rules are already represented in CD structure.
  - The holes in the initial structure help to focus on the points still to be established.
- Disadvantages of CD:

- Knowledge must be decomposed into fairly low level primitives.
- Impossible or difficult to find correct set of primitives.
- A lot of inference may still be required.
- Representations can be complex even for relatively simple actions.
- Consider: Dave bet Frank five pounds that Wales would win the Rugby World Cup.
- Complex representations require a lot of storage

2. Scripts

- A script is a structure that prescribes a set of circumstances which could be expected to follow on from one another.
- It is similar to a thought sequence or a chain of situations which could be anticipated.
- It could be considered to consist of a number of slots or frames but with more specialized roles.
- Scripts are beneficial because:
  - Events tend to occur in known runs or patterns.
  - Causal relationships between events exist.
  - Entry conditions exist which allow an event to take place
  - Prerequisites exist upon events taking place. E.g. when a student progresses through a degree scheme or when a purchaser buys a house.

Script Components

- Each script contains the following main components.
  - Entry Conditions: Must be satisfied before events in the script can occur.
  - Results: Conditions that will be true after events in script occur.
  - Props: Slots representing objects involved in the events.
  - Roles: Persons involved in the events.
  - Track: Specific variation on more general pattern in the script. Different tracks may share many components of the same script but not all.
  - Scenes: The sequence of events that occur. Events are represented in conceptual dependency form.

Advantages / Disadvantages of Script

- Advantages
  - Capable of predicting implicit events
  - Single coherent interpretation may be build up from a collection of observations.
- Disadvantage
  - More specific (inflexible) and less general than frames.
  - Not suitable to represent all kinds of knowledge.
- To deal with inflexibility, smaller modules called memory organization packets (MOP) can be combined in a way that is appropriate for the situation.

Script Example

Script : Play in theater	Various Scenes
Track: Play in Theater **Props:** • Tickets • Seat • Play **Roles:** • Person (who wants to see a play) – P • Ticket distributor – TD • Ticket checker – TC **Entry Conditions:** • P wants to see a play • P has a money **Results:** • P saw a play • P has less money • P is happy (optional if he liked the play)	***Scene 1: Going to theater*** • P PTRANS P into theater • P ATTEND eyes to ticket counter ***Scene 2: Buying ticket*** • P PTRANS P to ticket counter • P MTRANS (need a ticket) to TD • TD ATRANS ticket to P ***Scene 3: Going inside hall of theater and sitting on a seat*** • P PTRANS P into Hall of theater • TC ATTEND eyes on ticket POSS_by P • TC MTRANS (showed seat) to P • P PTRANS P to seat • P MOVES P to sitting position ***Scene 4: Watching a play*** • P ATTEND eyes on play • P MBUILD (good moments) from play ***Scene5: Exiting*** • P PTRANS P out of Hall and theater

- It must be activated based on its significance.
- If the topic is important, then the script should be opened.
- If a topic is just mentioned, then a pointer to that script could be held.
- For example, given "John enjoyed the play in theater", a script "Play in Theater" suggested above is invoked.
- All implicit questions can be answered correctly.
- Here the significance of this script is high.
    - Did john go to theater?
    - Did he buy ticket?
    - Did he have money?
- If we have a sentence like "John went to theater to pick his daughter", then invoking this script will lead to many wrong answers.
    - Here significance of the script theater is less.
- Getting significance from the story is not straightforward. However, some heuristics can be applied to get the value.

3. CYC

- What is CYC?
  - An ambitious attempt to form a very large knowledge base aimed at capturing commonsense reasoning.
  - Initial goals to capture knowledge from a hundred randomly selected articles in the EnCYClopedia Britannica.
  - Both Implicit and Explicit knowledge encoded.
  - Emphasis on study of underlying information (assumed by the authors but not needed to tell to the readers.
- Example: Suppose we read that Wellington learned of Napoleon's death
- Then we (humans) can conclude Napoleon never new that Wellington had died.
- How do we do this?
- We require special implicit knowledge or commonsense such as:
  - We only die once.
  - You stay dead.
  - You cannot learn of anything when dead.
  - Time cannot go backwards.
- Why build large knowledge bases:
  1. Brittleness
     - Specialised knowledge bases are brittle. Hard to encode new situations and non-graceful degradation in performance. Commonsense based knowledge bases should have a firmer foundation.
  2. Form and Content
     - Knowledge representation may not be suitable for AI. Commonsense strategies could point out where difficulties in content may affect the form.
  3. Shared Knowledge
     - Should allow greater communication among systems with common bases and assumptions.
- How is CYC coded?
  - By hand.
  - Special CYCL language:
  - LISP like.
  - Frame based
  - Multiple inheritance
  - Slots are fully fledged objects.
  - Generalized inheritance -- any link not just ***isa*** and ***instance***.

Introduction to Game playing

- Charles Babbage, the nineteenth century computer architect thought about programming his analytical engine to play chess and later of building a machine to play tic-tac-toe.
- There are two reasons that games appeared to be a good domain.
    1. They provide a structured task in which it is very easy to measure success or failure.
    2. They are easily solvable by straightforward search from the starting state to a winning position.
- The first is true is for all games bust the second is not true for all, except simplest games.
- For example consider chess.
- The average branching factor is around 35. In an average game, each player might make 50.
- So in order to examine the complete game tree, we would have to examine 35^{100} positions.
- Thus it is clear that a simple search is not able to select even its first move during the lifetime of its opponent.
- It is clear that to improve the effectiveness of a search based problem solving program two things can be done.
    1. Improve the generate procedure so that only good moves are generated.
    2. Improve the test procedure so that the best move will be recognized and explored first.
- If we use legal-move generator then the test procedure will have to look at each of them, because the test procedure must look at so many possibilities, it must be fast.
- Instead of legal-move generator we can use plausible-move generator in which only some small numbers of promising moves are generated.
- As the number of legal available moves increases it becomes increasingly important in applying heuristics to select only those moves that seem more promising.
- The performance of overall system can be improved by adding heuristic knowledge into both the generator and the tester.
- In game playing, a goal state is one in which we win but the game like chess, it is not possible, even we have good plausible move generator.
- The depth of the resulting tree or graph and its branching factor is too great.
- It is possible to search tree only ten or twenty moves deep then in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous.
- This is done using static evolution function, which uses whatever information it has to evaluate individual board position by estimating how likely they are to lead eventually to a win.
- Its function is similar to that of the heuristic function h’ in the A* algorithm: in the absence of complete information, choose the most promising position.

The MINIMAX search procedure

- The minimax search is a depth first and depth limited procedure.
- The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions.
- Now we can apply the static evolution function to those positions and simply choose the best one.
- After doing so, we can back that value up to the starting position to represent our evolution of it.
- Here we assume that static evolution function returns larger values to indicate good situations for us.
- So our goal is to maximize the value of the static evaluation function of the next board position.
- The opponents' goal is to minimize the value of the static evaluation function.
- ***The alternation of maximizing and minimizing at alternate ply when evaluations are to be pushed back up corresponds to the opposing strategies of the two players is called MINIMAX.***
- It is recursive procedure that depends on two procedures
    - MOVEGEN(position, player)— The plausible-move generator, which returns a list of nodes representing the moves that can be made by Player in Position.
    - STATIC(position, player)-- static evaluation function, which returns a number representing the goodness of Position from the standpoint of Player.
- With any recursive program, we need to decide when recursive procedure should stop.
- There are variety of factors that may influence the decision they are,
    - Has one side won?
    - How many ply have we already explored? Or how much time is left?
    - How stable is the configuration?
- We use DEEP-ENOUGH which is assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise.
- It takes two parameters, position and depth, it will ignore its position parameter and simply return TRUE if its depth parameter exceeds a constant cut off value.
- One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results.
    - The backed-up value of the path it chooses.
    - The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.
- We assume that MINIMAX returns a structure containing both results and we have two functions, VALUE and PATH that extract the separate components.
- Initially, It takes three parameters, a board position, the current depth of the search, and the player to move,

- MINIMAX(current,0,player-one) If player –one is to move
- MINIMAX(current,0,player-two) If player –two is to move

Algorithm: MINIMAX(position, depth, player)

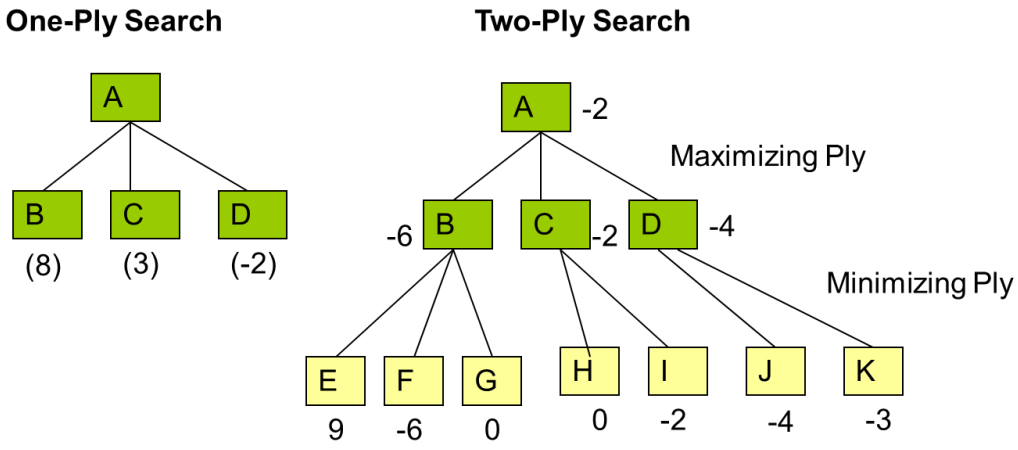
1. If DEEP-ENOUGH(Position, Depth), then return the structure
   VALUE = STATIC(Position, Player);
   PATH = nil
   This indicates that there is no path from this node and that its value is that determined by the static evaluation function
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN (Position Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.
   Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS.
   For each element SUCC of SUCCESSORS, do the following:
   a. Set RESULT-SUCC to **MINIMAX(SUCC, Depth + 1, OPPOSITE(Player)).**
      This recursive call to MINIMAX will actually carry out the exploration of SUCC
   b. Set NEW-VALUE to - VALUE (RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level
   c. If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
      i. Set BEST-SCORE to NEW-VALUE
      ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH (RESULT-SUCC).
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure
   VALUE = BEST-SCORE
   PATH = BEST-PATH

- When the initial call to MINIMAX returns, the best move from CURRENT is the first element on PATH. Performance can be improved significantly with a few refinements.

Example

- We use static evaluation function that returns value ranging from -10 to 10, with 10 indicating a win for us,-10 indicating win for opponent, 0 indicating an even match.
- As shown in fig 10.1, we move to B. Backing B's value up to A, we can conclude that A's

value is 8.



Backing up the values of a Two-Ply Search

Static Evaluation function ranges from -10 to +10, with 10 indicating win for us

Figure 10.1 backing Up the values of a two –play search

- But since we know that the static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply.
- After one move the situation would appear to be very good, but if we move ahead, situation may not be favorable.
- We first apply plausible move generator, generating a set of successor positions for each position, then we apply static evolution function to each of these position shown in fig 10.1.
- But now we must take into account that the opponent gets to choose which successor moves minimize the ply.
- This mean that if we make move B, the actual position in which we end is -6, because opponent selects it, also there is an E which is very good for us but opponent task is to minimize play shown in the figure.
- The minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen.
- The process can be repeated as many ply as time allows and more accurate evaluations that are produced can be used to choose the correct move at the top level.
- The alternation of maximizing and minimizing at alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players.
- This method is called ***Minimax.***

Adding alpha – beta cutoffs

- Minimax procedure is depth first process. One path is explored as far as time allows, the

static evolution function is applied to the game positions at the last step of the path.

- The efficiency of the depth first search can be improved by branch and bound technique in which partial solutions that are clearly worse than known solutions can be abandoned early.
- It is necessary to modify the branch and bound strategy to include two bounds, one for each of the players.
- This modified strategy is called alpha–beta pruning.
- It requires to maintain of two threshold values, one representing a lower bound on that a maximizing node may ultimately be assigned (we call this alpha).
- And another representing an upper bound on the value that a minimizing node may be assigned (this we call beta).
- Each level must receive both the values, one to use and one to pass down for the next level to use.
- The MINIMAX procedure as it stands does not need to treat maximizing and minimizing levels differently since it simply negates evaluation each time it changes levels.
- Instead of referring to alpha and beta, MINIMAX uses two values, USE-THRESH and PASS-THRESH.
- USE-THRESH is used to compute cutoffs. PASS-THRESH is passed to next level as its USE-THRESH.
- USE-THRESH must also be passed to the next level, but it will be passed as PASS-THRESH so that it can be passed to third level down as USE-THRESH again, and so forth.
- Just as values had to be negated each time they were passed across levels.
- Still there are no difference between the code required at maximizing levels and that required at minimizing levels.
- PASS-THRESH should always be the maximum of the value it inherits from above and the best move found at its level.
- If PASS-THRESH is updated the new value should be propagated both down to lower levels and back up to higher ones so that it always reflect the best move found anywhere in the tree.
- The MINIMAX-A-B requires five arguments, position, depth, player, Use-thresh, and pass-Thresh.
- MINIMAX-A-B(current,0,player-one,maximum value static can compute, minimum value static can compute)

Algorithm: MINIMAX-A-B(Position, depth, player, use-thresh, pass-thresh)

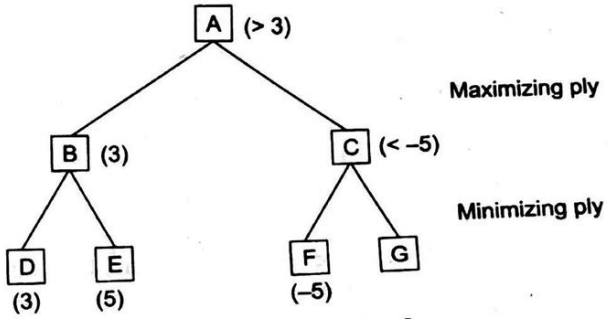
1. If DEEP-ENOUGH(Position, Depth), then return the structure
   VALUE = STATIC(Position, Player);
   PATH = nil
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position, Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, there are no moves to be made; return the same structure that

would have been returned if DEEP-ENOUGH had returned TRUE.

4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the best one. This is done as follows.
   For each element SUCC of SUCCESSOR:
   a. Set RESULT-SUCC to
      MINIMAX-A-B(SUCC, Depth + 1, OPPOSITE(Player), —Pass-Thresh, —Use-Thresh).
   b. Set NEW-VALUE to —VALUE (RESULT-SUCC).
   c. If NEW-VALUE > Pass-Thresh, then we have found a successor that is better than any that have been examined so far. Record this by doing the following.
      i. Set Pass-Thresh to NEW-VALUE.
      ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path from SUCC as determined by the recursive calls to MINIMAX-A-B. So set BEST-PATH to the result of attaching SUCC to the front of PATH (RESULT-SUCC).
   d. If Pass-Thresh Is not better then Use-Thresh, then we should stop examining this branch. But both thresholds and value been inverted. So if Pass-Thresh>=Use-Thresh, then return immediately with the value.
5. Return the structure
   VALUE=pass-Thresh
   PATH=BEST-PATH

Example

- To see how the alpha-beta procedure works, consider the example shown in Fig below.
- After examining node F, we know that the opponent is guaranteed a score of -5 or less at C (since the opponent is the minimizing player).
- But we also know that we are guaranteed a score of 3 or greater at node A, which we can achieve if we move to B.
- Any other move that produces a score of less than 3 is worse than the move to B.
- After examining only F, we are sure that a move to C is worse Maximizing ply (it will be less than or equal to -5) regardless of the score of node G.
- So we need not to explore node G.
- So not only one node but if we were exploring this tree to six ply, then we would have eliminated not a single node but an entire tree three ply deep.



- To see how the two thresholds, alpha and beta, can both be used, consider the example shown in Fig. 10.2.
- The entire sub tree headed by B is searched and we discover that at A, we can accept a score of at least 3.
- When this alpha value is passed down to F, it will enable us to skip the exploration of L let's see how.
- After K is examined, we see that I is guaranteed a maximum score of 0, which means that F is guaranteed a minimum of 0.
- But this is less than alpha's value of 3.
- Because we can get score of 3 by moving to B rather then moving to I.
- Now we examined J and we yield 5 because it is grater than 0.
- Now this value becomes the value of beta at node C.
- It indicates that C is guaranteed to get a 5 or less.
- Now we must expand G, first M is examined and it has a value 7 which is passed to G as the tentative value. But now 7 is compared to beta (5).
- It is greater and the player whose turn it is at C is trying to minimize.
- So player will not choose G which would lead to a score of at least 7, since there is an alternative move to F which will lead to a score of 5.
- So it is not necessary to explore any of other branches of G.

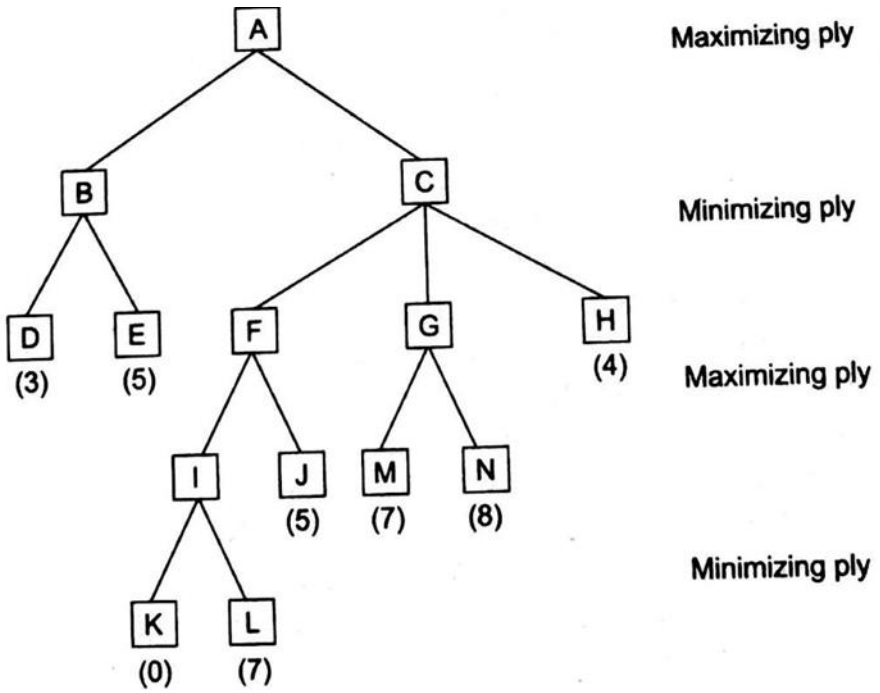


Figure 10.2 alpha and beta cutoffs

Planning

- Methods which focus on ways of decomposing the original problem into appropriate subparts and on ways of recording and handling interactions among the subparts as they are detected during the problem-solving process are often called as planning.
- Planning refers to the process of computing several steps of a problem-solving procedure before executing any of them.

Components of a planning system

- Choose the best rule to apply next, based on the best available heuristic information.
    - The most widely used technique for selecting appropriate rules to apply is first to isolate a set of differences between desired goal state and then to identify those rules that are relevant to reduce those differences.
    - If there are several rules, a variety of other heuristic information can be exploited to choose among them.
- Apply the chosen rule to compute the new problem state that arises from its application.
    - In simple systems, applying rules is easy. Each rule simply specifies the problem state that would result from its application.
    - In complex systems, we must be able to deal with rules that specify only a small part of the complete problem state.
    - One way is to describe, for each action, each of the changes it makes to the state description.
- Detect when a solution has been found.
    - A planning system has succeeded in finding a solution to a problem when it has found a sequence of operators that transforms the initial problem state into the goal state.
    - How will it know when this has been done?
    - In simple problem-solving systems, this question is easily answered by a straightforward match of the state descriptions.
    - One of the representative systems for planning systems is predicate logic. Suppose that as a part of our goal, we have the predicate P(x).
    - To see whether P(x) is satisfied in some state, we ask whether we can prove P(x) given the assertions that describe that state and the axioms that define the world model.
- Detect dead ends so that they can be abandoned and the system's effort directed in more fruitful directions.
    - As a planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect when it is exploring a path that can never lead to a solution.
    - The same reasoning mechanisms that can be used to detect a solution can often be used for detecting a dead end.

  - If the search process is reasoning forward from the initial state, it can prune any path that leads to a state from which the goal state cannot be reached.
  - If search process is reasoning backward from the goal state, it can also terminate a path either because it is sure that the initial state cannot be reached or because little progress is being made.
- Detect when an almost correct solution has been found and employ special techniques to make it totally correct.
  - The kinds of techniques discussed are often useful in solving nearly decomposable problems.
  - One good way of solving such problems is to assume that they are completely decomposable, proceed to solve the sub-problems separately, and then check that when the sub-solutions are combined, they do in fact give a solution to the original problem.

Goal Stack Planning

- In this method, the problem solver makes use of a single stack that contains both goals and operators that have been proposed to satisfy those goals.
- The problem solver also relies on a database that describes the current situation and a set of operators described as PRECONDITION, ADD, and DELETE lists.
- The goal stack planning method attacks problems involving conjoined goals by solving the goals one at a time, in order.
- A plan generated by this method contains a sequence of operators for attaining the first goal, followed by a complete sequence for the second goal etc.
- At each succeeding step of the problem solving process, the top goal on the stack will be pursued.
- When a sequence of operators that satisfies it, is found, that sequence is applied to the state description, yielding new description.
- Next, the goal that is then at the top of the stack is explored and an attempt is made to satisfy it, starting from the situation that was produced as a result of satisfying the first goal.
- This process continues until the goal stack is empty.
- Then as one last check, the original goal is compared to the final state derived from the application of the chosen operators.
- If any components of the goal are not satisfied in that state, then those unsolved parts of the goal are reinserted onto the stack and the process is resumed.

Nonlinear Planning using Constraint Posting

- Difficult problems cause goal interactions.
- The operators used to solve one sub-problem may interfere with the solution to a previous

sub-problem.

- Most problems require an intertwined plan in which multiple sub-problems are worked on simultaneously.
- Such a plan is called nonlinear plan because it is not composed of a linear sequence of complete sub-plans.

Constraint Posting

- The idea of constraint posting is to build up a plan by incrementally hypothesizing operators, partial orderings between operators, and binding of variables within operators.
- At any given time in the problem-solving process, we may have a set of useful operators but perhaps no clear idea of how those operators should be ordered with respect to each other.
- A solution is a partially ordered, partially instantiated set of operators to generate an actual plan, and we convert the partial order into any number of total orders.

Constraint Posting versus State Space search

- State Space Search
  - Moves in the space: Modify world state via operator
  - Model of time: Depth of node in search space
  - Plan stored in: Series of state transitions
- Constraint Posting Search
  - Moves in the space:
    Add operators
    Oder Operators
    Bind variables
    Or Otherwise constrain plan
  - Model of Time: Partially ordered set of operators
  - Plan stored in: Single node

Algorithm: Nonlinear Planning (TWEAK)

1. Initialize S to be the set of propositions in the goal state.
2. Remove some unachieved proposition P from S.
3. Achieve P by using step addition, promotion, DE clobbering, simple establishment or separation.
4. Review all the steps in the plan, including any new steps introduced by step addition, to see if any of their preconditions are unachieved. Add to S the new set of unachieved preconditions.
5. If S is empty, complete the plan by converting the partial order of steps into a total order, instantiate any variables as necessary.
6. Otherwise go to step 2.

Hierarchical Planning

- In order to solve hard problems, a problem solver may have to generate long plans.
- It is important to be able to eliminate some of the details of the problem until a solution that addresses the main issues is found.
- Then an attempt can be made to fill in the appropriate details.
- Early attempts to do this involved the use of macro operators, in which larger operators were built from smaller ones.
- In this approach, no details were eliminated from actual descriptions of the operators.

ABSTRIPS

- A better approach was developed in ABSTRIPS systems which actually planned in a hierarchy of abstraction spaces, in each of which preconditions at a lower level of abstraction were ignored.
- ABSTRIPS approach is as follows:
    - First solve the problem completely, considering only preconditions whose criticality value is the highest possible.
    - These values reflect the expected difficulty of satisfying the precondition.
    - To do this, do exactly what STRIPS did, but simply ignore the preconditions of lower than peak criticality.
    - Once this is done, use the constructed plan as the outline of a complete plan and consider preconditions at the next-lowest criticality level.
    - Augment the plan with operators that satisfy those preconditions.
    - Because this approach explores entire plans at one level of detail before it looks at the lower-level details of any one of them, it has been called length-first approach.
- The assignment of appropriate criticality value is crucial to the success of this hierarchical planning method.
- Those preconditions that no operator can satisfy are clearly the most critical.
- Example, solving a problem of moving robot, for applying an operator, PUSH-THROUGH DOOR, the precondition that there exist a door big enough for the robot to get through is of high criticality since there is nothing we can do about it if it is not true.

Reactive Systems

- The idea of reactive systems is to avoid planning altogether, and instead use the observable situation as a clue to which one can simply react.
- A reactive system must have access to a knowledge base of some sort that describes what actions should be taken under what circumstances.
- A reactive system is very different from the other kinds of planning systems we have discussed because it chooses actions one at a time.
- It does not anticipate and select an entire action sequence before it does the first thing.

- Example is a Thermostat. The job of the thermostat is to keep the temperature constant inside a room.
- Reactive systems are capable of surprisingly complex behaviors.
- The main advantage reactive systems have over traditional planners is that they operate robustly in domains that are difficult to model completely and accurately.
- Reactive systems dispense with modeling altogether and base their actions directly on their perception of the world.
- Another advantage of reactive systems is that they are extremely responsive, since they avoid the combinatorial explosion involved in deliberative planning.
- This makes them attractive for real time tasks such as driving and walking.

Other Planning Techniques

- Triangle tables
  - Provides a way of recording the goals that each operator is expected to satisfy as well as the goals that must be true for it to execute correctly.
- Meta-planning
  - A technique for reasoning not just about the problem being solved but also about the planning process itself.
- Macro-operators
  - Allow a planner to build new operators that represent commonly used sequences of operators.
- Case based planning:
  - Re-uses old plans to make new ones.

Blocks World Problem

- In order to compare the variety of methods of planning, we should find it useful to look at all of them in a single domain that is complex enough that the need for each of the mechanisms is apparent yet simple enough that easy-to follow examples can be found.
  - There is a flat surface on which blocks can be placed.
  - There are a number of square blocks, all the same size.
  - They can be stacked one upon the other.
  - There is robot arm that can manipulate the blocks.

Actions of the robot arm

1. UNSTACK(A,B): Pick up block A from its current position on block B.
2. STACK(A,B): Place block A on block B.
3. PICKUP(A): Pick up block A from the table and hold it.
4. PUTDOWN(A): Put block A down on the table.

- Notice that the robot arm can hold only one block at a time.

Predicates

- In order to specify both the conditions under which an operation may be performed and the results of performing it, we need the following predicates:
  1. ON(A,B): Block A is on Block B.
  2. ONTABLES(A): Block A is on the table.
  3. CLEAR(A): There is nothing on the top of Block A.
  4. HOLDING(A): The arm is holding Block A.
  5. ARMEMPTY: The arm is holding nothing.

Robot –problem solving system (STRIPS)

- List of new predicates that the operator causes to become true is ADD List
- List of old predicates that the operator causes to become false is DELETE List
- PRECONDITIONS list contains those predicates that must be true for the operator to be applied.

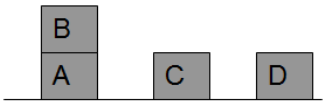
STRIPS style operators for BLOCKs World

STACK(x, y)
P: CLEAR(y)^HOLDING(x)
D: CLEAR(y)^HOLDING(x)
A: ARMEMPTY^ON(x, y)
UNSTACK(x, y)
PICKUP(x)
P: CLEAR(x) ^ ONTABLE(x) ^ARMEMPTY
D: ONTABLE(x) ^ ARMEMPTY
A: HOLDING(x)
PUTDOWN(x)

Goal Stack Planning

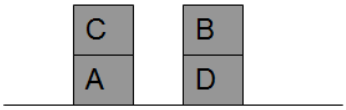
- To start with goal stack is simply:

  ON(C,A)^ON(B,D)^ONTABLE(A)^ONTABLE(D)
- This problem is separate into four sub-problems, one for each component of the goal.
- Two of the sub-problems ONTABLE(A) and ONTABLE(D) are already true in the initial state.



Start:

ON(B,A)^ONTABLE(A) ^ ONTABLE(C) ^ONTABLE(D) ^ARMEMPTY



Goal: ON(C,A)^ON(B,D)^ ONTABLE(A)^ONTABLE(D)

- Alternative 1: Goal Stack:
  ON(C,A)
  ON(B,D)
  ON(C,A)^ON(B,D)^OTAD

- Alternative 2: Goal stack:
  ON(B,D)
  ON(C,A)
  ON(C,A)^ON(B,D)^OTAD

Exploring Operators

- Pursuing alternative 1, we check for operators that could cause ON(C,A)
- Out of the 4 operators, there is only one STACK. So it yields:
  - STACK(C,A)
  - ON(B,D)
  - ON(C,A)^ON(B,D)^OTAD
- Preconditions for STACK(C,A) should be satisfied, we must establish them as sub-goals:
  - CLEAR(A)
  - HOLDING(C)
  - CLEAR(A)^HOLDING(C)
  - STACK(C,A)
  - ON(B,D)
  - ON(C,A)^ON(B,D)^OTAD
- Here we exploit the Heuristic that if HOLDING is one of the several goals to be achieved at once, it should be tackled last.

Goal stack Planning

- Next we see if CLEAR(A) is true. It is not. The only operator that could make it true is UNSTACK(B,A). This produces the goal stack:
  - ON(B,A)
  - CLEAR(B)
  - ON(B,A)^CLEAR(B)^ARMEMPTY
  - UNSTACK(B,A)
  - HOLDING(C)

- CLEAR(A)^HOLDING(C)
- STACK(C,A)
- ON(B,D)
- ON(C,A)^ON(B,D)^OTAD

- We see that we can pop predicates on the stack till we reach HOLDING(C) for which we need to find suitable operator.
- The operators that might make HOLDING(C) true : PICKUP(C) and UNSTACK(C, x). Without looking ahead, since we cannot tell which of these operators is appropriate, we create two branches of the search tree corresponding to the following goal stacks:

ALT1:	ALT2:
ONTABLE(C)	ON(C, x)
CLEAR(C)	CLEAR(C)
ARMEMPTY	ARMEMPTY
ONTABLE(C) ^CLEAR(C)^ARMEMPTY	ON(C,x)^CLEAR(C)^ARMEMPTY
PICKUP(C)	UNSTACK(C,x)
CLEAR(A)^HOLDING(C)	CLEAR(A)^HOLDING(C)
STACK(C,A)	STACK(C,A)
ON(B,D)	ON(B,D)
ON(C,A)^ON(B,D)^OTAD	ON(C,A)^ON(B,D)^OTAD

Complete plan

1. UNSTACK(C,A)
2. PUTDOWN(C)
3. PICKUP(A)
4. STACK(A,B)
5. UNSTACK(A,B)
6. PUTDOWN(A)
7. PICKUP(B)
8. STACK(B,C)
9. PICKUP(A)
10. STACK(A,B)

Heuristics for Planning using Constraint Posting (TWEAK)

1. Step addition – creating new steps for a plan.
2. Promotion – Constraining one step to come before another in a final plan.
3. Declobbering – Placing one (possibly new) step S2 between two old steps S1 and S3 such that S2 reasserts some precondition of S3 that was neglected (or “clobbered”) by S1.
4. Simple establishment – Assigning a value to a variable, in order to ensure the preconditions of some step.
5. Separation – Preventing the assignment of certain values to a variable.

Two steps with respect to ON(A,B) and ON(B,C)

- Each step is written with its preconditions above it and its post-conditions below it.
- Delete post-conditions are marked with a negation symbol (¬)
- Neither can be executed right away because some of their preconditions are not satisfied.
- An unachieved precondition is marked with *.

CLEAR(B) *HOLDING(A)	CLEAR(C) *HOLDING(B)
STACK(A,B)	STACK(B,C)
ARMEMPTY ON(A,B) ¬CLEAR(B) ¬HOLDING(A)	ARMEMPTY ON(B,C) ¬CLEAR(C) ¬HOLDING(B)

Step Addition

- Introducing new steps to achieve goals or preconditions is called Step addition.
- It is one of the heuristics used in generating nonlinear plans.
- Step addition is a very basic method where Means-Ends Analysis was used to pick operators with post-conditions corresponding to desired states.
- To achieve the preconditions of the two steps, we can use step addition again:

*CLEAR(A) ONTABLE(A) *ARMEMPTY	*CLEAR(B) ONTABLE(B) *ARMEMPTY
PICKUP(A)	PICKUP(B)
¬ONTABLE(A) ¬ARMEMPTY HOLDING(A)	¬ONTABLE(B) ¬ARMEMPTY HOLDING(B)

Partial Ordering and Promotion

- Adding PICKUP steps is not enough to satisfy the *HOLDING preconditions of the STACK steps.
- If in the eventual plan, the PICKUP steps were to follow the STACK steps, then the *HOLDING preconditions would need to be satisfied by some other set of steps.
- In this case we want each PICKUP step should precede its corresponding STACK step
  - PICKUP(A) ← STACK(A,B)
  - PICKUP(B) ← STACK(B,C)
- We now have four partially ordered steps and four unachieved preconditions.
- To achieve precondition CLEAR(B), we use Heuristic known as PROMOTION.
- Promotion amounts to posting a constraint that one step must precede another in eventual plan.
- We can achieve CLEAR(B) by stating that the PICKUP(B) step must come before the

STACK(A,B) step:

- PICKUP(B) ← STACK(A,B)

Declobbering

- A third heuristic called Declobbering can help achieve *ARMEMPTY precondition in the PICKUP(A) step.
- PICKUP(B) asserts ¬ARMEMPTY, but if we can insert another step between PICKUP(B) and PICKUP(A) to reassert ARMEMPTY, then the precondition will be achieved. The STACK(B,C) does the trick, so we post another constraint:
- PICKUP(B) ← STACK(B,C) ←PICKUP(A)
- The step PICKUP(B) is said to "clobber" PICKUP(A)'s precondition. STACK(B,C) is said to "Declobber" it.

Simple Establishment

- *ON(x, A)
- *CLEAR(x)
- *ARMEMPTY
- ----------
- UNSTACK(x, A)
- ----------
- ¬ARMEMPTY
- CLEAR(A)
- HOLDING(A)
- ¬ON(x, A)

- A variable x is introduced because the only precondition we are interested in is CLEAR(A). Whatever block is on top of A is irrelevant.
- Variables allow us to avoid committing to particular instantiations of operators.
- We have three unachieved preconditions. We can achieve ON(x, A) easily by constraining the value of x to be block C. This works because block C is on block A in the initial state.
- This is called Simple establishment and allows us to state that two different propositions must be ultimately instantiated to the same proposition.
- x = C in step UNSTACK(x, A)

Plan ordering and Variable Binding

1. UNSTACK(C,A)
2. PUTDOWN(C)
3. PICKUP(B)
4. STACK(B,C)
5. PICKUP(A)
6. STACK(A,B)

- We used four different Heuristics to synthesize it: Step addition, promotion, declobbering and simple establishment.

Introduction

- Language is meant for communicating about the world.
- By studying language, we can come to understand more about the world.
- If we can succeed at building computational mode of language, we will have a powerful tool for communicating about the world.
- We look at how we can exploit knowledge about the world, in combination with linguistic facts, to build computational natural language systems.
- Natural Language Processing (NLP) problem can be divided into two tasks:
    1. Processing written text, using lexical, syntactic and semantic knowledge of the language as well as the required real world information.
    2. Processing spoken language, using all the information needed above plus additional knowledge about phonology as well as enough added information to handle the further ambiguities that arise in speech.

Steps in Natural Language Processing

1. Morphological Analysis:
    - Individual words are analyzed into their components and non-word tokens such as punctuation are separated from the words.
2. Syntactic Analysis:
    - Linear sequences of words are transformed into structures that show how the words relate to each other.
    - Some word sequences may be rejected if they violate the language's rule for how words may be combined.
3. Semantic Analysis:
    - The structures created by the syntactic analyzer are assigned meanings.
    - A mapping is made between the syntactic structures and objects in the task domain.
    - Structures for which no such mapping is possible may be rejected.
4. Discourse integration:
    - The meaning of an individual sentence may depend on the sentences that precede it and may influence the meanings of the sentences that follow it.
5. Pragmatic Analysis:
    - The structure representing what was said is reinterpreted to determine what was actually meant.

1. Morphological Analysis

- Suppose we have an English interface to an operating system and the following sentence is typed:

I want to print Bill's .init file.

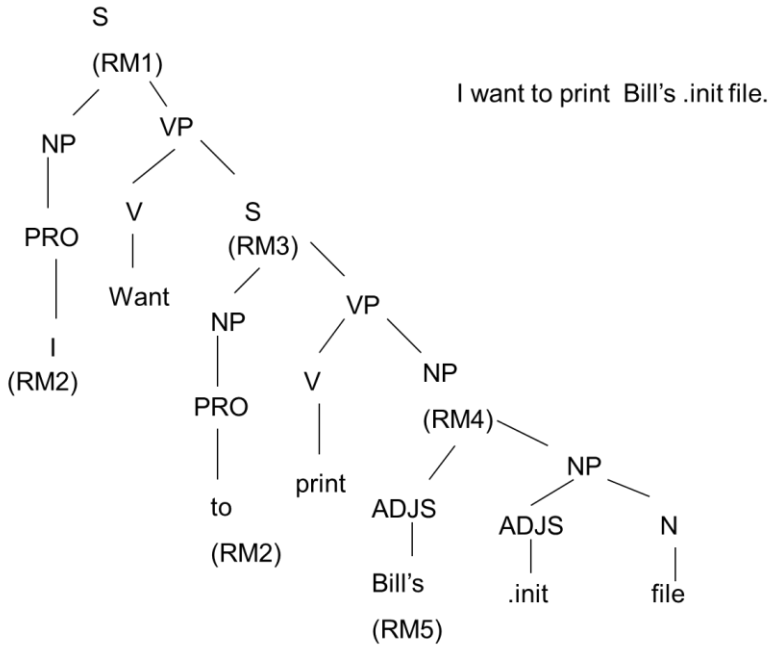
- Morphological analysis must do the following things:

- Pull apart the word "Bill's" into proper noun "Bill" and the possessive suffix "'s"
- Recognize the sequence ".init" as a file extension that is functioning as an adjective in the sentence.

- This process will usually assign syntactic categories to all the words in the sentence.

2. Syntactic Analysis

- Syntactic analysis must exploit the results of morphological analysis to build a structural description of the sentence.
- The goal of this process, called parsing, is to convert the flat list of words that forms the sentence into a structure that defines the units that are represented by that flat list.
- The important thing here is that a flat sentence has been converted into a hierarchical structure and that the structure corresponds to meaning units when semantic analysis is performed.
- Reference markers (set of entities) are shown in the parenthesis in the parse tree.
- Each one corresponds to some entity that has been mentioned in the sentence.
- These reference markers are useful later since they provide a place in which to accumulate information about the entities as we get it.



3. Semantic Analysis

- Semantic analysis must do two important things:
  a. It must map individual words into appropriate objects in the knowledge base or database.
  b. It must create the correct structures to correspond to the way the meanings of the individual words combine with each other.

4. Discourse Integration

- Specifically we do not know whom the pronoun "I" or the proper noun "Bill" refers to.
- To pin down these references requires an appeal to a model of the current discourse

context, from which we can learn that the current user is USER068 and that the only person named "Bill" about whom we could be talking is USER073.

- Once the correct referent for Bill is known, we can also determine exactly which file is being referred to.

5. Pragmatic Analysis

- The final step toward effective understanding is to decide what to do as a result.
- One possible thing to do is to record what was said as a fact and be done with it.
- For some sentences, whose intended effect is clearly declarative, that is precisely correct thing to do.
- But for other sentences, including this one, the intended effect is different.
- We can discover this intended effect by applying a set of rules that characterize cooperative dialogues.
- The final step in pragmatic processing is to translate, from the knowledge based representation to a command to be executed by the system.

Summary

- Results of each of the main processes combine to form a natural language system.
- All of the processes are important in a complete natural language understanding system.
- Not all programs are written with exactly these components.
- Sometimes two or more of them are collapsed.
- Doing that usually results in a system that is easier to build for restricted subsets of English but one that is harder to extend to wider coverage.

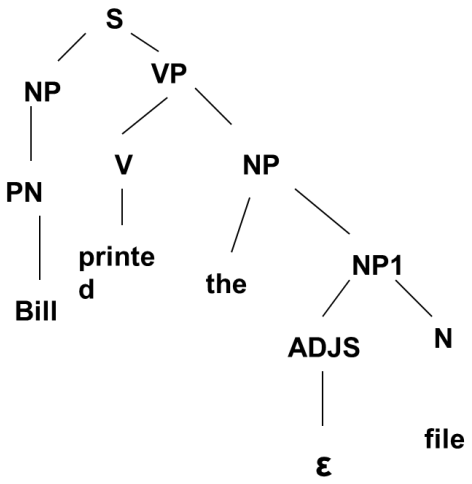
Syntactic Processing

- Syntactic Processing is the step in which a flat input sentence is converted into a hierarchical structure that corresponds to the units of meaning in the sentence. This process is called parsing.
- It plays an important role in natural language understanding systems for two reasons:
  i. Semantic processing must operate on sentence constituents. If there is no syntactic parsing step, then the semantics system must decide on its own constituents. If parsing is done, on the other hand, it constrains the number of constituents that semantics can consider.
  ii. Syntactic parsing is computationally less expensive than is semantic processing. Thus it can play a significant role in reducing overall system complexity.
- Although it is often possible to extract the meaning of a sentence without using grammatical facts, it is not always possible to do so.
- Almost all the systems that are actually used have two main components:
  1. A declarative representation, called a grammar, of the syntactic facts about the language.
  2. A procedure, called parser that compares the grammar against input sentences to produce parsed structures.

Grammars and Parsers

- The most common way to represent grammars is as a set of production rules.
- First rule can be read as "A sentence is composed of a noun phrase followed by Verb Phrase"; Vertical bar is OR; ε represents empty string.
- Symbols that are further expanded by rules are called non-terminal symbols.
- Symbols that correspond directly to strings that must be found in an input sentence are called terminal symbols.
- Grammar formalism such as this one underlies many linguistic theories, which in turn provide the basis for many natural language understanding systems.
- Pure context free grammars are not effective for describing natural languages.
- NLPs have less in common with computer language processing systems such as compilers.
- Parsing process takes the rules of the grammar and compares them against the input sentence.
- The simplest structure to build is a Parse Tree, which simply records the rules and how they are matched.
- Every node of the parse tree corresponds either to an input word or to a non-terminal in our grammar.
- Each level in the parse tree corresponds to the application of one grammar rule.

Example: A Parse tree for a sentence: Bill Printed the file



- Grammar specifies two things about a language:
  1. Its weak generative capacity, by which we mean the set of sentences that are contained within the language. This set is made up of precisely those sentences that can be completely matched by a series of rules in the grammar.
  2. Its strong generative capacity, by which we mean the structure to be assigned to each grammatical sentence of the language.

Augmented transition network (ATN)

- An augmented transition network is a top down parsing procedure that allows various kinds

of knowledge to be incorporated into the parsing system so it can operate efficiently.

- ATNs build on the idea of using finite state machines (Markov model) to parse sentences.
- Instead of building an automaton for a particular sentence, a collection of transition graphs are built.
- A grammatically correct sentence is parsed by reaching a final state in any state graph.
- Transitions between these graphs are simply subroutine calls from one state to any initial state on any graph in the network.
- A sentence is determined to be grammatically correct if a final state is reached by the last word in the sentence.
- The ATN is similar to a finite state machine in which the class of labels that can be attached to the arcs that define transition between states has been augmented.
- Arcs may be labeled with:
    - Specific words such as “in’.
    - Word categories such as noun.
    - Procedures that build structures that will form part of the final parse.
    - Procedures that perform arbitrary tests on current input and sentence components that have identified.

Semantic Analysis

- Producing a syntactic parse of a sentence is only the first step toward understanding it.
- We must produce a representation of the meaning of the sentence.
- Because understanding is a mapping process, we must first define the language into which we are trying to map.
- There is no single definitive language in which all sentence meaning can be described.
- The choice of a target language for any particular natural language understanding program must depend on what is to be done with the meanings once they are constructed.
- Choice of target language in semantic Analysis
    - There are two broad families of target languages that are used in NL systems, depending on the role that the natural language system is playing in a larger system:
    - When natural language is being considered as a phenomenon on its own, as for example when one builds a program whose goal is to read text and then answer questions about it, a target language can be designed specifically to support language processing.
    - When natural language is being used as an interface language to another program (such as a db query system or an expert system), then the target language must be legal input to that other program. Thus the design of the target language is driven by the backend program.

Lexical processing

- The first step in any semantic processing system is to look up the individual words in a dictionary (or lexicon) and extract their meanings.
- Many words have several meanings, and it may not be possible to choose the correct one just by looking at the word itself.
- The process of determining the correct meaning of an individual word is called word sense disambiguation or lexical disambiguation.
- It is done by associating, with each word in lexicon, information about the contexts in which each of the word's senses may appear.
- Sometimes only very straightforward info about each word sense is necessary.
- Some useful semantic markers are :
    - PHYSICAL-OBJECT
    - ANIMATE-OBJECT
    - ABSTRACT-OBJECT

Sentence-Level Processing

- Several approaches to the problem of creating a semantic representation of a sentence have been developed, including the following:
    - Semantic grammars, which combine syntactic, semantic and pragmatic knowledge into a single set of rules in the form of grammar.
    - Case grammars, in which the structure that is built by the parser contains some semantic information, although further interpretation may also be necessary.
    - Conceptual parsing, in which syntactic and semantic knowledge are combined into a single interpretation system that is driven by the semantic knowledge.
    - Approximately compositional semantic interpretation, in which semantic processing is applied to the result of performing a syntactic parse.

Semantic Grammar

- A semantic grammar is a context-free grammar in which the choice of non-terminals and production rules is governed by semantic as well as syntactic function.
- There is usually a semantic action associated with each grammar rule.
- The result of parsing and applying all the associated semantic actions is the meaning of the sentence.
- Advantages of Semantic grammars
    - When the parse is complete, the result can be used immediately without the additional stage of processing that would be required if a semantic interpretation had not already been performed during the parse.
    - Many ambiguities that would arise during a strictly syntactic parse can be avoided since some of the interpretations do not make sense semantically and thus cannot be generated by a semantic grammar.
    - Syntactic issues that do not affect the semantics can be ignored.

- The drawbacks of use of semantic grammars are:
    - The number of rules required can become very large since many syntactic generalizations are missed.
    - Because the number of grammar rules may be very large, the parsing process may be expensive.

Case grammars

- Case grammars provide a different approach to the problem of how syntactic and semantic interpretation can be combined.
- Grammar rules are written to describe syntactic rather than semantic regularities.
- But the structures the rules produce correspond to semantic relations rather than to strictly syntactic ones.

Conceptual Parsing

- Conceptual parsing is a strategy for finding both the structure and meaning of a sentence in one step.
- Conceptual parsing is driven by dictionary that describes the meaning of words in conceptual dependency (CD) structures.
- The parsing is similar to case grammar.
- CD usually provides a greater degree of predictive power.

Discourse and Pragmatic processing

- To understand a single sentence, it is necessary to consider the discourse and pragmatic context in which the sentence was uttered.
- There are a number of important relationships that may hold between phrases and parts of their discourse contexts, including:
    1. Identical entities. Consider the text:
        - Bill had a red balloon.
        - John wanted it.
        - The word “it” should be identified as referring to red balloon. These types of references are called anaphora.
    2. Parts of entities. Consider the text:
        - Sue opened the book she just bought.
        - The title page was torn.
        - The phrase “title page” should be recognized as part of the book that was just bought.
    3. Parts of actions. Consider the text:
        - John went on a business trip to New York.
        - He left on an early morning flight.
        - Taking a flight should be recognized as part of going on a trip.
    4. Entities involved in actions. Consider the text:

- My house was broken into last week.
- They took the TV and the stereo.
- The pronoun “they” should be recognized as referring to the burglars who broke into the house.

5. Elements of sets. Consider the text:
   - The decals we have in stock are stars, the moon, item and a flag.
   - I’ll take two moons.
   - Moons mean moon decals.
6. Names of individuals:
   - Dev went to the movies.
7. Causal chains
   - There was a big snow storm yesterday.
   - The schools were closed today.
8. Planning sequences:
   - Sally wanted a new car
   - She decided to get a job.
9. Implicit presuppositions:
   - Did Joe fail CS101?

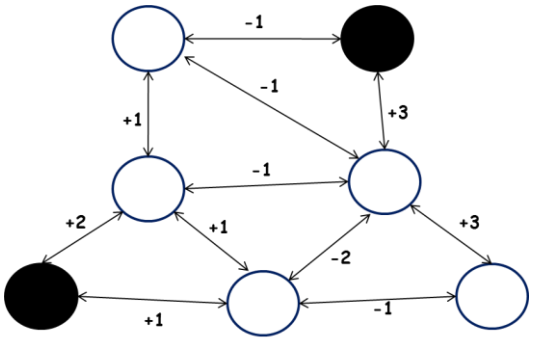
- The major focus is on using following kinds of knowledge:
  - The current focus of the dialogue.
  - A model of each participant’s current beliefs.
  - The goal-driven character of dialogue.
  - The rules of conversation shared by all participants.

Introduction

- Neural network architectures have been called connectionist architecture.
- They are characterized by having:
  - A very large number of simple neuron-like processing elements.
  - A large number of weighted connections between the elements. The weights on the connections encode the knowledge of a network.
  - Highly parallel, distributed control.
  - An emphasis on learning internal representations automatically.

Hopfield Networks

- Hopfield [1982]: introduced a neural network as a theory of memory: Model of content addressable memory.
- Features of a Hopfield Network:
  - Distributed representation
  - A memory is stored as a pattern of activation across a set of processing elements.
  - Distributed, asynchronous control
  - Content-addressable memory
  - A number of patterns can be stored in a network. To retrieve a pattern, a specific portion of it is specified and the network automatically finds the closest match.
  - Fault tolerance: If a few processing elements misbehave or fail completely, the network will still function properly.
  - Each processing element makes decisions based only on its own local situation.
- A simple Hopfield Network is shown below:



Black → active
White → inactive

- Processing elements or units are always in one of two states, active or inactive.
- Units are connected to each other with weighted symmetric connection a positive weighted connection indicates that the two units tend to activate each other.
- A negative weighted connection allows an active unit to deactivate a neighboring unit.

Parallel relaxation algorithm

- The network operates as follows:
  1. A random unit is chosen
  2. If any of it neighbors are active, the unit computes the sum of the weights on the connections to those active neighbors.
  3. If the sum is positive, the unit becomes active,
     Otherwise it becomes inactive.
  4. Another random unit is chosen, and the process repeats until the network reaches a stable state. (e.g. until no more unit can change state)
- black and positive → will attempt to activate the unit connected to it
- Network can be thought of as storing the patterns, given any set of weights and any initial state, the parallel relaxation algorithm will eventually steer the network into a stable state.
- Problem: sometimes the network cannot find global solution because the network sticks with the local minima as nodes settle into stable states via a completely distributed algorithm.

Learning in Neural Network

Perceptron

- The perceptron an invention of (1962) Rosenblatt was one of the earliest neural network models.
- It models a neuron by taking a weighted sum of its inputs and sending the output 1 if the sum is greater than some adjustable threshold value (otherwise it sends 0).

$g(x) = \sum_{k=0}^{n} WkXk$	$\text{Output}(x) = \begin{cases} 1 & \text{if } g(x) > 0 \\ 0 & \text{if } g(x) > 0 \end{cases}$

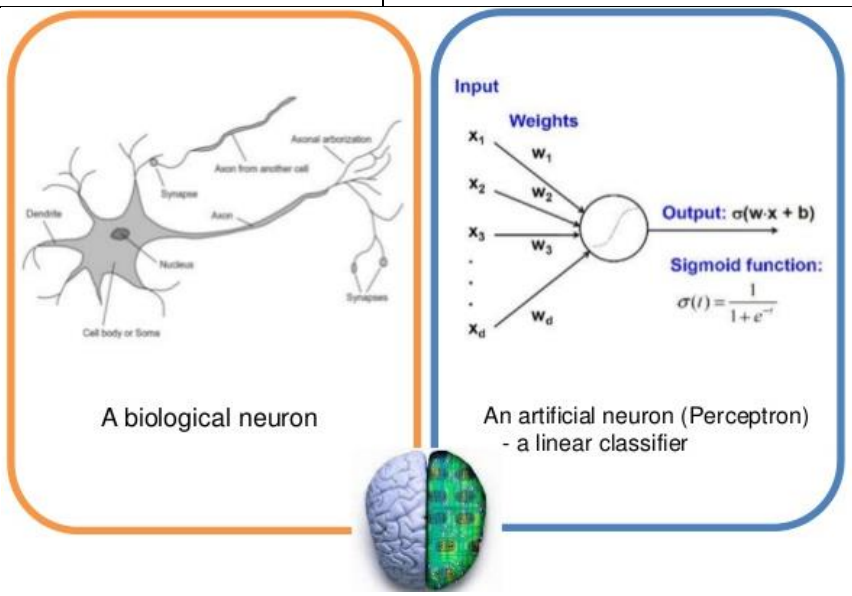


Figure 13.1 A neuron & a Perceptron

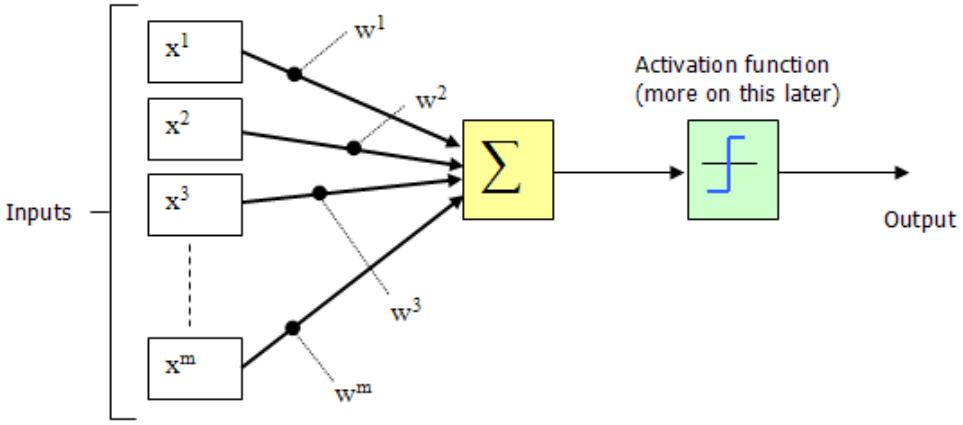


Figure 13.2 Perceptron with adjustable threshold

- In case of zero with two inputs

$$g(x) = w0 + w1x1 + w2x2 = 0$$

$$x2 = -(w1/w2)x1 - (w0/w2) \rightarrow \text{equation for a line}$$

- the location of the line is determine by the weight w0 w1 and w2
- if an input vector lies on one side of the line, the perceptron will output 1
- if it lies on the other side, the perceptron will output 0
- Decision surface: a line that correctly separates the training instances corresponds to a perfectly function perceptron.

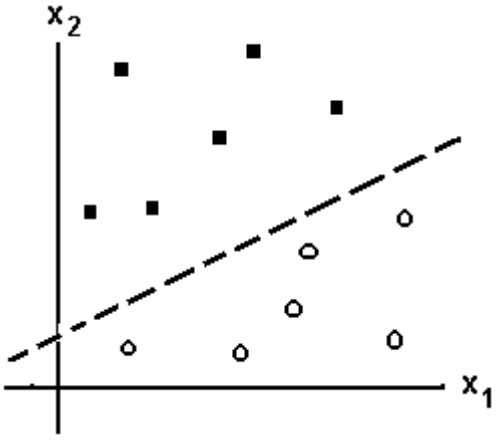


Figure 13.3 Linearly Separable Pattern Classification Problem

Perceptron Learning Algorithm

Given: A classification problem with n input feature (x_1, x_2, ..., x_n) and two output classes.

Compute: A set of weights (w_0, w_1, w_2,....,w_n) that will cause a perceptron to fire whenever the input falls into the first output class.

1. Create a perceptron with n+ 1 input and n+ 1 weight, where the x_0 is always set to 1.
2. Initialize the weights (w_0, w_1,...., w_n) to random real values.
3. Iterate through the training set, collecting all examples *misclassified* by the current set of

weights.

4. If all examples are classified correctly, output the weights and quit.
5. Otherwise, compute the vector sum S of the misclassified input vectors where each vector has the form (x_0, x_1, …, x_n). In creating the sum, add to S a vector $\vec{x}$ if $\vec{x}$ is an input for which the perceptron incorrectly fails to fire, but - $\vec{x}$ if $\vec{x}$ is an input for which the perceptron incorrectly fires. Multiply sum by a scale factor η.
6. Modify the weights (w_0, w_1, …, w_n) by adding the elements of the vector S to them. Go to step 3.

- The perceptron learning algorithm is a search algorithm. It begins in a random initial state and finds a solution state. The search space is simply all possible assignments of real values to the weights of the perceptron, and the search strategy is gradient descent.
- The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as a solution exists.
- This brings us to an important question. What problems can a perceptron solve? Recall that a single-neuron perceptron is able to divide the input space into two regions.
- The perceptron can be used to classify input vectors that can be separated by a linear boundary. We call such vectors linearly separable.
- Unfortunately, many problems are not linearly separable. The classic example is the XOR gate. **It was the inability of the basic perceptron to solve such simple problems that are not linearly separable or non-linear.**

Artificial neural network

- An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain process information.
- Typically, neurons are five to six orders of magnitude slower than silicon logic gates; events in a silicon chip happen in the nanosecond (10-9 s) range, whereas neural events happen in the millisecond (10-3 s) range.
- The brain is a highly complex, nonlinear, and parallel information-processing system.
- It has the capability of organizing neurons so as to perform certain computations (e.g. pattern recognition, perception, and motor control) many times faster than the fastest digital computer.
- A brain has great structure and the ability to build up its own rules through what we usually refer to as experience.
- During this early stage of development, about one million synapses are formed per second.
- Synapses are elementary structural and functional units that mediate the interactions between neurons.
- A developing neuron is synonymous with a plastic brain: Plasticity permits the developing nervous system to adapt to its surrounding environment.
- Axons act as transmission lines, and dendrites represent receptive zones. Neurons come in a

wide variety of shapes and sizes in different parts of the brain. A pyramidal cell can receive 10,000 or more synaptic contacts and it can project onto thousands of target cells.

- In its most general form, a neural network is a machine that is designed to model the way in which the brain performs a particular task or function of interest.
- The network is usually implemented using electronic components or simulated in software on a digital computer.
- To achieve good performance, neural networks employ a massive interconnection of simple computing cells referred to as neurons or processing units.
- A neural network is a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects.
  1. Knowledge is acquired by the network through a learning process.
  2. Interneuron connection strengths known as synaptic weights are used to store the Knowledge.
- Neural networks are also referred to as neuro-computers, connectionist networks, parallel Distributed processors, etc.

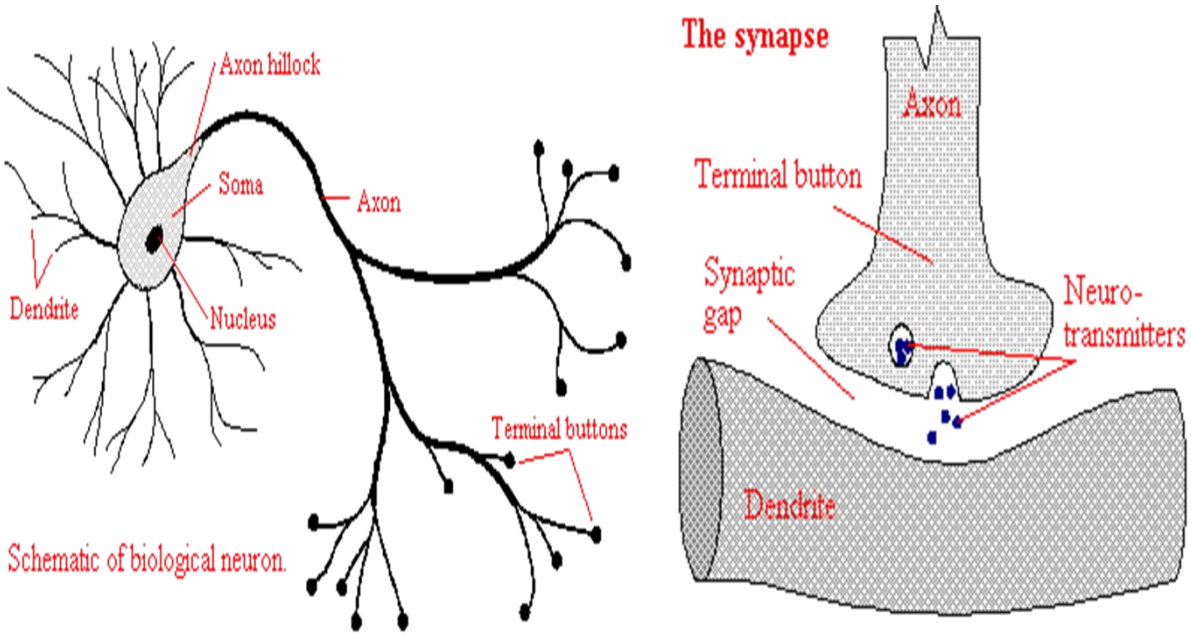


Figure 13.4 Biological neuron

Activation functions

- A neuron is an information-processing unit that is fundamental to the operation of a neural network.
- An activation function is used for limiting the amplitude of the output of a neuron. The activation function is also referred to in the literature as a squashing function in that it squashes (limits) the permissible amplitude range of the output signal to some finite value.
- Typically, the normalized amplitude range of the output of a neuron is written as the closed unit interval [0, 1] or alternatively [-1, 1].

- There are many activation functions.

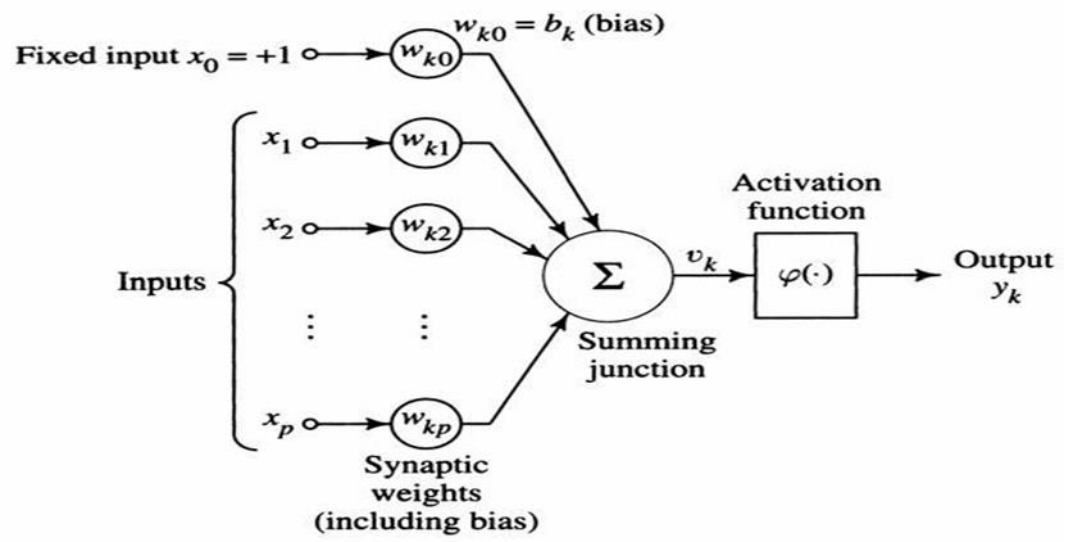
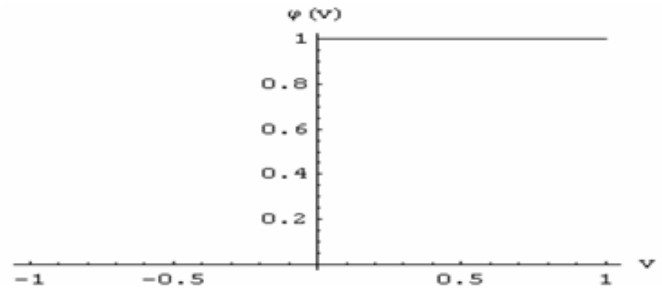


Figure 13.5 Nonlinear model of a neuron.

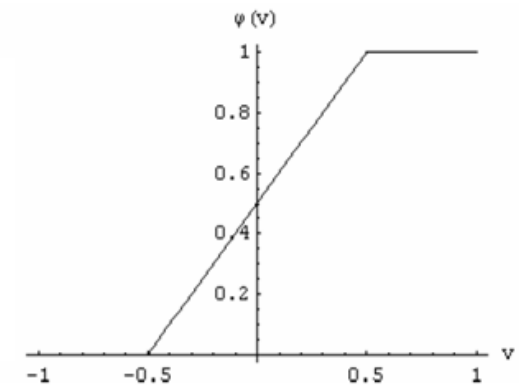
1. **Threshold activation function (McCulloch–Pitts model)**
   - In this model, the output of a neuron takes on the value of 1 if the total internal activity level of that neuron is nonnegative and 0 otherwise.

$$\varphi(v) = \begin{cases} 1, & v \geq 0 \\ 0, & v < 0 \end{cases}$$



2. **Piecewise-linear activation function**

$$\varphi(v) = \begin{cases} 1, & v \geq \frac{1}{2} \\ v + \frac{1}{2}, & -\frac{1}{2} < v < \frac{1}{2} \\ 0, & v \leq -\frac{1}{2} \end{cases}$$



   - The amplification factor inside the linear region is assumed to be unity.
   - A linear combiner arises if the linear region of operation in maintained without running

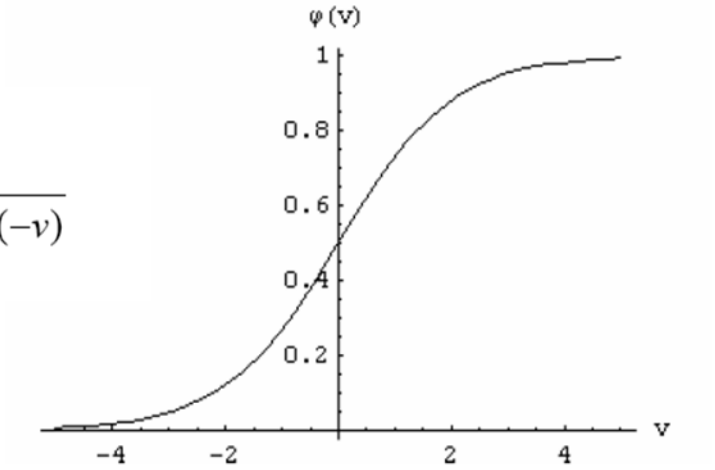
into saturation.

- The piecewise-linear function reduces to a threshold function if the amplification factor of the linear region in made infinitely large.

3. Sigmoid (logistic) activation function

- The sigmoid function is the most common form of activation function used in the construction of artificial neural networks.
- Whereas a threshold function assumes the value of 0 or 1, a sigmoid function assumes a continuous range of values form 0 and 1.
- Note also that the sigmoid function is differentiable, which is an important feature of neural network theory.

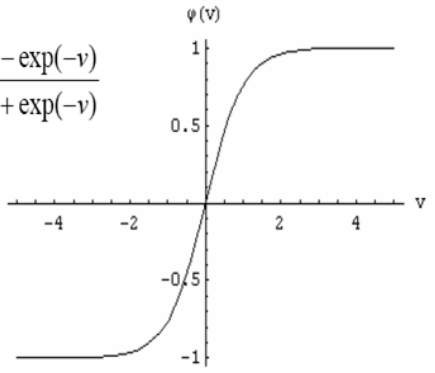
$$\varphi(v) = \frac{1}{1+\exp(-v)}$$



4. Hyperbolic tangent function

The hyperbolic tangent function can be easily expressed in terms of the logistic function: (2 × logistic function – 1).

$$\varphi(v) = \tanh(\frac{v}{2}) = \frac{1-\exp(-v)}{1+\exp(-v)}$$



5. Softmax activation function

- One approach toward approximating probabilities is to choose the output neuron nonlinearity to be exponential rather than sigmoidal and for each pattern to normalize the outputs to sum to 1.
- Let c be the number of output neurons.
- Each output is generated by the following activation function:

$$y_i = \frac{e^{v_i}}{\sum_{i=1}^{c} e^{v_i}}, \quad i = 1, 2, \ldots, c$$

- The softmax activation function is a smoothed version of a winner take- all nonlinearity in which the maximum output is transformed to 1, and all others reduced to 0.

Supervised and Unsupervised Learning

Supervised learning

- An essential ingredient of supervised learning is the availability of an external teacher, which is able to provide the neural network with a desired or target response.
- The network parameters are adjusted under the combined influence of the training vector and the error signal.
- This adjustment is carried out iteratively in a step-by-step fashion with the aim of eventually making the neural network emulate the teacher.
- This form of supervised learning is in fact an error-correction learning, which was already described.

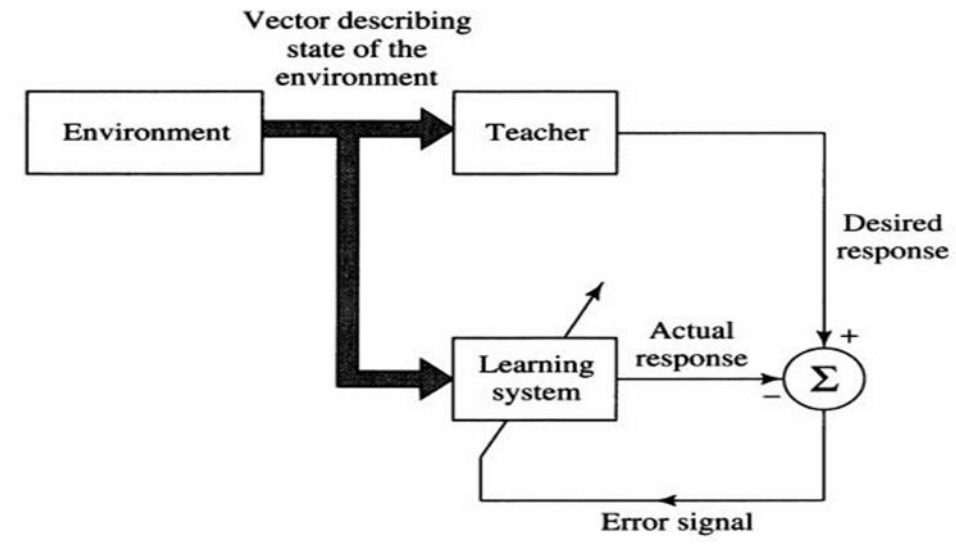


Figure 13.6 Supervised learning

Unsupervised learning

- In unsupervised or self-organized learning there is no external teacher to oversee the learning process.
- In other words, there are no specific samples of the function to be learned by the network.
- Rather, provision is made for a task-independent measure of the quality of

representation that the network is required to learn and the free parameters of the network are optimized with respect to that measure.

- Once the network has become tuned to the statistical regularities of the input data, it develops the ability to form internal representations for encoding features of the input and thereby creates new classes automatically.

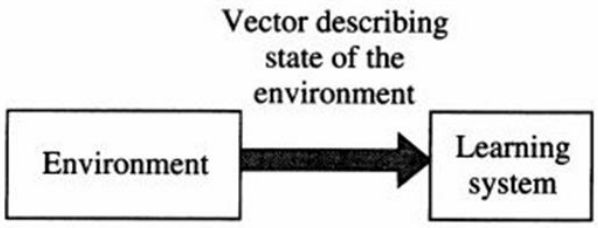


Figure 13.7 Unsupervised learning

Applications of neural networks

1. **Neural Networks in Practice**
   - Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:
     - Sales forecasting , industrial process control, customer research, data validation, risk management, target marketing.
   - ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis of hepatitis; recovery of telecommunications from faulty software; interpretation of multi-meaning Chinese words; undersea mine detection; texture analysis; three-dimensional object recognition; hand-written word recognition; and facial recognition.
2. **Neural networks in medicine**
   - Artificial Neural Networks (ANN) are currently a promising research area in medicine and it is believed that they will receive extensive application to biomedical systems in the next few years.
   - At the moment, the research is mostly on modeling parts of the human body and recognizing diseases from various scans.
   - Modeling and Diagnosing the Cardiovascular System
     - At the moment, the research is mostly on modeling parts of the human body and recognizing diseases from various scans
     - Diagnosis can be achieved by building a model of the cardiovascular system of an individual and comparing it with the real time physiological measurements taken from the patient.
   - Electronic noses
     - ANNs are used experimentally to implement electronic noses

- Electronic noses have several potential applications in telemedicine.
- The electronic nose would identify odors in the remote surgical environment.
- These identified odors would then be electronically transmitted to another site where a door generation system would recreate them.

- Instant Physician
  - An application developed in the mid-1980s called the "instant physician" trained an auto associative memory neural network to store a large number of medical records, each of which includes information on symptoms, diagnosis, and treatment for a particular case.

3. Neural Networks in business

- Business is a diverted field with several general areas of specialization such as accounting or financial analysis.
- Almost any neural network application would fit into one business area or financial analysis.
- There is some potential for using neural networks for business purposes, including resource allocation and scheduling.
- There is also a strong potential for using neural networks for database mining, which is, searching for patterns implicit within the explicitly stored information in databases.
- Marketing
  - There is a marketing application which has been integrated with a neural network system.
  - The Airline Marketing Tactician (a trademark abbreviated as AMT) is a computer system made of various intelligent technologies including expert systems.
  - A feedforward neural network is integrated with the AMT and was trained using back-propagation to assist the marketing control of airline seat allocations.
  - The adaptive neural approach was amenable to rule expression.
  - Additionally, the application's environment changed rapidly and constantly, which required a continuously adaptive solution.
- Credit Evaluation
  - The HNC Company, founded by Robert Hecht-Nielsen, has developed several neural network applications.
  - One of them is the Credit Scoring system which increases the profitability of the existing model up to 27%.
  - The HNC neural systems were also applied to mortgage screening.
  - A neural network automated mortgage insurance underwriting system was developed by the Nestor Company.

- Apart from that ANN is also used for,
  - **Classification**
    - ✓ in marketing: consumer spending pattern classification
    - ✓ In defense: radar and sonar image classification

- ✓ In agriculture & fishing: fruit and catch grading
- ✓ In medicine: ultrasound and electrocardiogram image classification, EEGs, medical diagnosis

- **Recognition and identification**
  - ✓ In general computing and telecommunications: speech, vision and handwriting recognition.
  - ✓ In finance: signature verification and bank note verification.
- **Assessment**
  - ✓ In engineering: product inspection monitoring and control
  - ✓ In defense: target tracking
  - ✓ In security: motion detection, surveillance image analysis and fingerprint matching
- **Forecasting and prediction**
  - ✓ In finance: foreign exchange rate and stock market forecasting
  - ✓ In agriculture: crop yield forecasting
  - ✓ In marketing: sales forecasting
  - ✓ In meteorology: weather prediction

Recurrent Networks

- Recurrent Networks are used in temporal AI task such as planning, natural language processing, etc…
- A recurrent neural network (RNN) is a class of neural network where connections between units form a directed cycle.
- This creates an internal state of the network which allows it to exhibit dynamic temporal behavior.
- Unlike feed-forward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs.
- This makes them applicable to tasks such as un-segmented connected handwriting recognition, where they have achieved the best known results.
- This is the basic architecture developed in the 1980s: a network of neuron-like units, each with a directed connection to every other unit.
- Each unit has a time-varying real-valued activation. Each connection has a modifiable real-valued weight. Some of the nodes are called input nodes, some output nodes, the rest hidden nodes.

Connectionist AI and Symbolic AI

- Different architectures make different assumptions about the content they will process and about the types of problems they will solve.

- Connectionism applied very successfully to classification problems and low-level processing. Strengths include integrity in face of uncertain and incorrect data, and a natural gradation in match.
- But they require large amounts of training data and the knowledge in the final net which is usually opaque.
- Symbolic systems have largely complementary strengths/weaknesses.
- Human-like cognition probably requires both.

  Connectionist
  - Search : parallel relaxation
  - Knowledge Representation: very large number of real-valued connection strengths. Structure often stored as distributed patterns of activation.
  - Learning : Back-propagation, Boltzmann machines, reinforcement learning, unsupervised learning

  Symbolic
  - Search: state space traversal.
  - Knowledge Representation: Predicate logic, semantic networks, frames, scripts.
  - Learning: Macro-operators, version spaces, explanation-based learning, discovery.