

UNIT I

What is Artificial Intelligence?

1. INTELLIGENCE

- ❖ The capacity to learn and solve problems.
- ❖ In particular,
 - the ability to solve novel problems (i.e solve new problems)
 - the ability to act rationally (i.e act based on reason)
 - the ability to act like humans

1.1 What is involved in intelligence?

- **Ability to interact with the real world**
 - to perceive, understand, and act
 - e.g., speech recognition and understanding and synthesis
 - e.g., image understanding
 - e.g., ability to take actions, have an effect
- **Reasoning and Planning**
 - modeling the external world, given input
 - solving new problems, planning, and making decisions
 - ability to deal with unexpected problems, uncertainties
- **Learning and Adaptation**
 - we are continuously learning and adapting
 - our internal models are always being —updated||
 - e.g., a baby learning to categorize and recognize animals

2. ARTIFICIAL INTELLIGENCE

It is the study of how to make computers do things at which, at the moment, people are better.

The term AI is defined by each author in own ways which falls into 4 categories

1. The system that think like humans.
2. System that act like humans.
3. Systems that think rationally.
4. Systems that act rationally.

2.1 SOME DEFINITIONS OF AI

- **Building systems that think like humans**
 - The exciting new effort to make computers think ... machines with minds, in the full and literal sense|| -- Haugeland, 1985
 - The automation of activities that we associate with human thinking, ... such as decision-making, problem solving, learning, ...|| -- Bellman, 1978

- **Building systems that act like humans**
 - The art of creating machines that perform functions that require intelligence when performed by people|| -- Kurzweil, 1990
 - The study of how to make computers do things at which, at the moment, people are better|| -- Rich and Knight, 1991
- **Building systems that think rationally**
 - The study of mental faculties through the use of computational models|| -- Charniak and McDermott, 1985
 - The study of the computations that make it possible to perceive, reason, and act|| -Winston, 1992
- **Building systems that act rationally**
 - A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes|| -- Schalkoff, 1990
 - The branch of computer science that is concerned with the automation of intelligent behavior|| -- Luger and Stubblefield, 1993

2.1.1. Acting Humanly: The Turing Test Approach

- ❖ Test proposed by Alan Turing in 1950
- ❖ The computer is asked questions by a human interrogator.

The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Programming a computer to pass, the computer need to possess the following capabilities:

- ❖ **Natural language processing** to enable it to communicate successfully in English.
 - ❖ **Knowledge representation** to store what it knows or hears
 - ❖ **Automated reasoning** to use the stored information to answer questions and to draw new conclusions.
 - ❖ **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns.
- To pass the complete Turing Test, the computer will need
- ❖ Computer vision to perceive the objects, and
 - ❖ Robotics to manipulate objects and move about.

2.1.2 Thinking humanly: The cognitive modeling approach

We need to get inside actual working of the human mind:

- (a) Through introspection – trying to capture our own thoughts as they go by;
- (b) Through psychological experiments

Allen Newell and Herbert Simon, who developed GPS, the —General Problem Solver tried to trace the reasoning steps to traces of human subjects solving the same problems. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind

2.1.3 Thinking rationally : The “laws of thought approach”

The Greek philosopher Aristotle was one of the first to attempt to codify —right thinking that is irrefutable (ie. Impossible to deny) reasoning processes. His syllogism provided patterns for argument structures that always yielded correct conclusions when given correct premises—for example, Socrates is a man; all men are mortal; therefore Socrates is mortal. These laws of thought were supposed to govern the operation of the mind; their study initiated a field called logic.

2.1.4 Acting rationally : The rational agent approach

An agent is something that acts. Computer agents are not mere programs, but they are expected to have the following attributes also: (a) operating under autonomous control, (b) perceiving their environment, (c) persisting over a prolonged time period, (e) adapting to change. A rational agent is one that acts so as to achieve the best outcome.

3. HISTORY OF AI

- 1943: early beginnings
 - McCulloch & Pitts: Boolean circuit model of brain
- 1950: Turing
 - Turing's "Computing Machinery and Intelligence—
- 1956: birth of AI
 - Dartmouth meeting: "Artificial Intelligence—name adopted
- 1950s: initial promise
 - Early AI programs, including
 - Samuel's checkers program
 - Newell & Simon's Logic Theorist

- 1955-65: —great enthusiasm
 - Newell and Simon: GPS, general problem solver
 - Gelertner: Geometry Theorem Prover
 - McCarthy: invention of LISP
- 1966—73: Reality dawns
 - Realization that many AI problems are intractable
 - Limitations of existing neural network methods identified
 - Neural network research almost disappears
- 1969—85: Adding domain knowledge
 - Development of knowledge-based systems
 - Success of rule-based expert systems,
 - E.g., DENDRAL, MYCIN
 - But were brittle and did not scale well in practice
- 1986-- Rise of machine learning
 - Neural networks return to popularity
 - Major advances in machine learning algorithms and applications
- 1990-- Role of uncertainty
 - Bayesian networks as a knowledge representation framework
- 1995--AI as Science
 - Integration of learning, reasoning, knowledge representation
 - AI methods used in vision, language, data mining, etc

PROBLEMS, PROBLEM SPACES AND SEARCH

2. FORMULATING PROBLEMS

Problem formulation is the process of deciding what actions and states to consider, given a goal

Formulate Goal, Formulate problem



Search



Execute

2.1 WELL-DEFINED PROBLEMS AND SOLUTIONS

A problem can be defined formally by four components:

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

1. Initial State

The starting state which agent knows itself.

1. Successor Function

- A description of the possible actions available to the agent.
- State x , successor – FN (x) returns a set of $\langle \text{action}, \text{successor} \rangle$ ordered pairs, where each action is a legal action in a state x and each successor is a state that can be reached from x by applying that action.

2.1 State Space

The set of all possible states reachable from the initial state by any sequence of actions. The initial state and successor function defines the state space. The state space forms a graph in which nodes are state and axis between the nodes are action.

2.2 Path

A path in the state space is a sequence of state connected by a sequence of actions.

2. Goal Test

Test to determine whether the given state is the goal state. If there is an explicit set of possible goal states, then we can whether any one of the goal state is reached or not.

Example : In chess, the goal is to reach a state called —checkmate— where the opponent's king is under attack and can't escape.

3. Path cost

A function that assigns a numeric cost to each path. The cost of a path can be described as the sum of the costs of the individual actions along that path.

Step cost of taking an action a to go from one state x to state y is denoted by $C(x,a,y)$

C-Cost , x,y - states , Action , Step costs are non-negative

These 4 elements are gathered into a data structure that is given as input to problem solving algorithm. A solution quality is measured by path cost function. An optimal solution has lowest path cost among all solutions.

Total cost = Path cost + Search cost

Example: Route finding problem

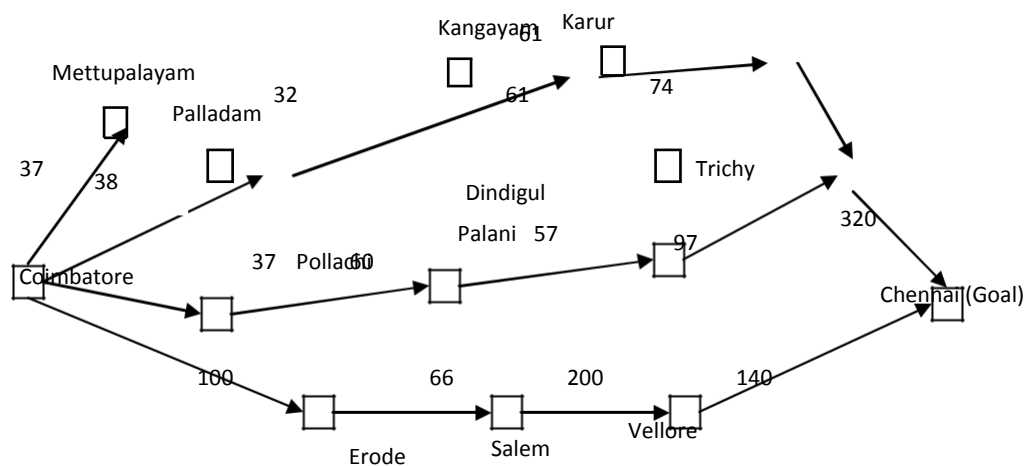


Fig: 1 Route Finding Problem

Initial State: In (Coimbatore)

Successor Function: {< Go (Pollachi), In (Pollachi)>

< Go (Erode), In (Erode)>

< Go (Palladam), In (Palladam)>

< Go (Mettupalayam), In (Mettupalayam)>}

Goal Test: In (Chennai)

Path Cost: {(In (Coimbatore),)}

{Go (Erode),} = 100 [kilometers]

{In (Erode)}

Path cost = 100 + 66 + 200 + 140 = 506

2.2 TOY PROBLEM

Example-1 : Vacuum World

Problem Formulation

- States
 - $2 \times 2^2 = 8$ states
 - Formula $n2^n$ states
- Initial State
 - Any one of 8 states
- Successor Function
 - Legal states that result from three actions (Left, Right, Absorb)
- Goal Test
 - All squares are clean
- Path Cost
 - Number of steps (each step costs a value of 1)

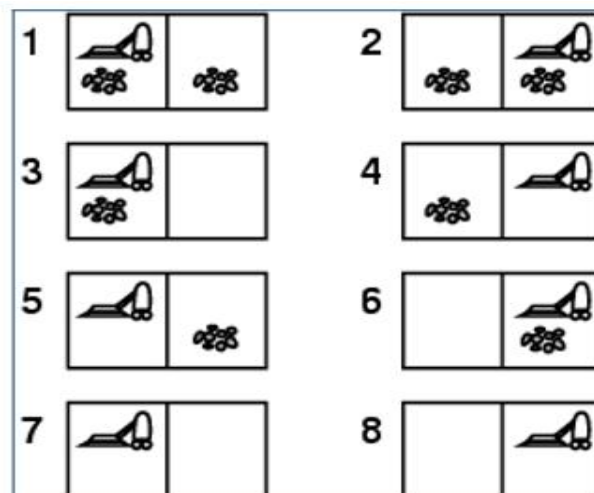


Fig 1.2 Vacuum World

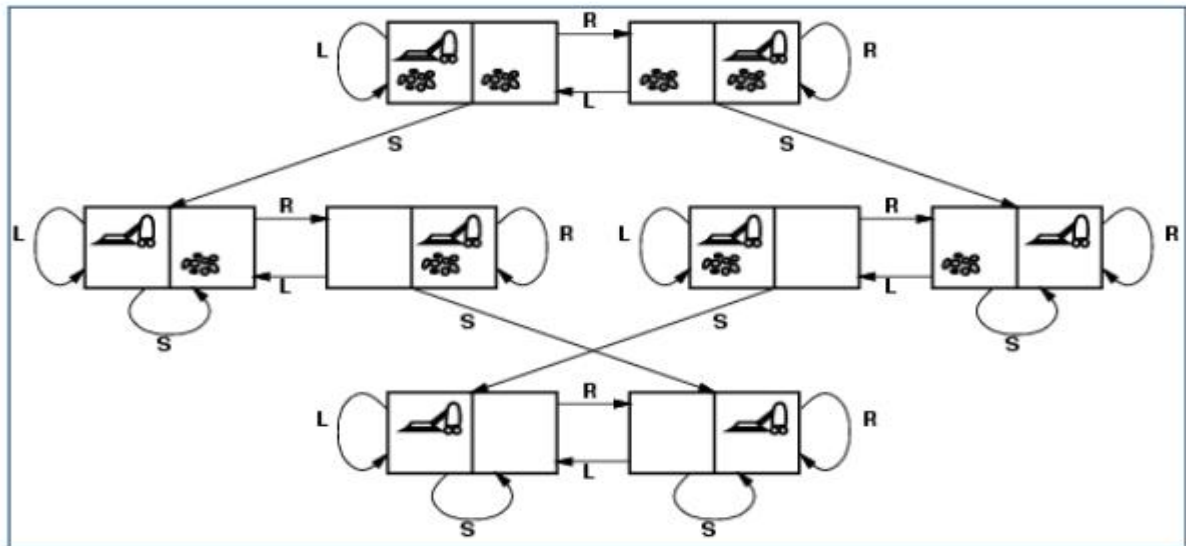


Fig: 1.3 State Space for the Vacuum World

State Space for the Vacuum World

Labels on Arcs denote L: Left, R: Right, S: Suck

Example 2: Playing chess

Initial State: Described as an 8 X 8 array where each positions contains a symbol standing for the appropriate piece in the official chess position.

Successor function: The legal states that results from set of rules.

They can be described easily by as a set of rules consisting of two parts: a left side that serves as a pattern to be matched against the current board position and a right side that describes the changes to be made to the board position to reflect the move. An example is shown in the following figure.

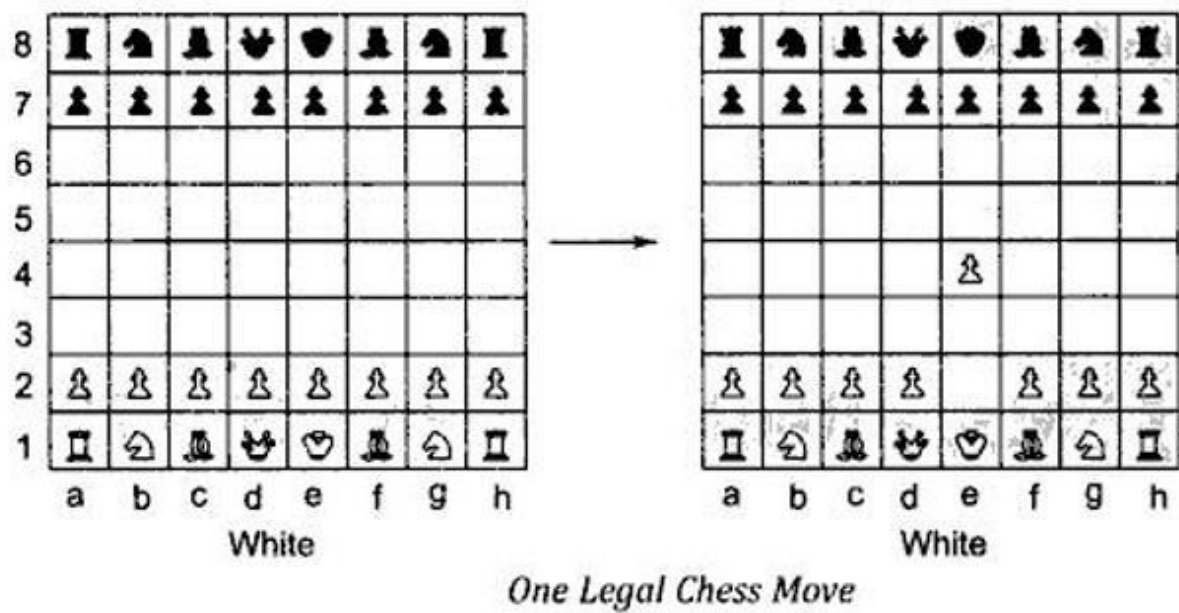


Fig 1.4: The legal states that results from set of rules

However if we write rules like the one above, we have to write a very large number of them since there has to be a separate set of rule for each of them roughly 10^{120} possible board positions.

Practical difficulties to implement large number of rules,

1. It will take too long to implement large number of rules and could not be done without mistakes.
2. No program could easily handle all those rules and storing it possess serious difficulties.

In order to minimize such problems, we have to write rules describing the legal moves in as a general way as possible. The following is the way to describe the chess moves.

Current Position

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

Changing Board Position

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

Some of the problems that fall within the scope of AI and the kinds of techniques will be useful to solve these problems.

GoalTest

Any position in which the opponent does not have a legal move and his or her king is under attack.

Example: 3 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?

State: (x, y) $x = 0, 1, 2, 3, \text{ or } 4$ $y = 0, 1, 2, 3$

x represents quantity of water in 4-gallon jug and y represents quantity of water in 3-gallon jug.

•**Start state:** $(0, 0)$.

•**Goal state:** $(2, n)$ for n . Attempting to end up in a goal state.(since the problem any doesn't specify the quantity of water in 3-gallon jug)

- | | | |
|---|--------------------------------|---|
| 1. (x, y)
If $x < 4$ | $\rightarrow (4, y)$ | Fill the 4-gallon jug |
| 2. (x, y)
If $y < 3$ | $\rightarrow (x, 3)$ | Fill the 3-gallon jug |
| 3. (x, y)
If $x > 0$ | $\rightarrow (x - d, y)$ | Pour some water out of the 4-gallon jug |
| 4. (x, y)
If $y > 0$ | $\rightarrow (x, y - d)$ | Pour some water out of the 3-gallon jug |
| 5. (x, y)
If $x > 0$ | $\rightarrow (0, y)$ | Empty the 4-gallon jug on the ground |
| 6. (x, y)
If $y > 0$ | $\rightarrow (x, 0)$ | Empty the 3-gallon jug on the ground |
| 7. (x, y)
If $x + y \geq 4, y > 0$ | $\rightarrow (4, y - (4 - x))$ | Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full |
| 8. (x, y)
If $x + y \geq 3, x > 0$ | $\rightarrow (x - (3 - y), 3)$ | Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full |
| 9. (x, y) | $\rightarrow (x + y, 0)$ | Pour all the water from the 3-gallon |

	If $x + y \leq 4, y > 0$		jug into the 4-gallon jug Pour all the water from the 4-gallon
10.	(x, y)	$\rightarrow (0, x + y)$	
	If $x + y \leq 3, x > 0$		jug into the 3-gallon jug Pour the 2 gallons from the 3-gallon
11.	$(0, 2)$	$\rightarrow (2, 0)$	
			Jug into the 4-gallon jug
12.	$(2, y)$	$\rightarrow (0, y)$	Empty the 2 gallons in the 4-gallon Jug on the ground

Production rules for the water jug problem

Trace of steps involved in solving the water jug problem First solution

Number of Steps	Rules applied	4-g jug	3-g jug
1	Initial state	0	0
2	R2 {Fill 3-g jug}	0	3
3	R7 {Pour all water from 3 to 4-g jug}	3	0
4	R2 {Fill 3-g jug}	3	3
5	R5 {Pour from 3 to 4-g jug until it is full}	4	2
6	R3 {Empty 4-gallon jug}	0	2
7	R7 {Pour all water from 3 to 4-g jug}	2	0

Goal State

Second Solution

Number of Steps	Rules applied	4-g jug	3-g jug
1	Initial state	0	0
2	R1 {Fill 4-gallon jug}	4	0
3	R6 {Pour from 4 to 3-g jug until it is full}	1	3
4	R4 {Empty 3-gallon jug}	1	0
5	R8 {Pour all water from 4 to 3-gallon jug}	0	1
6	R1 {Fill 4-gallon jug}	4	1
7	R6 {Pour from 4 to 3-g jug until it is full}	2	3
8	R4 {Empty 3-gallon jug}	2	0

Goal State

Example - 5 8-puzzle Problem

The 8-puzzle problem consists of a 3 x 3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state.

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

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

Successor function: This generates the legal states that result from trying the four actions (blank moves Left, Right, Up, or Down).

Goal test: This checks whether the state matches the goal configuration (Other goal configurations are possible.)

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

2	8	3
1	6	4
7		5

Initial State

	1	2
3	4	5
6	7	8

Goal State

Fig 1.5 8 Puzzle Problem

Exampe-6 8-queens problem

The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal).

States: Any arrangement of 0 to 8 queens on the board is a state.

Initial state: No queens on the board.

Successor function: Add a queen to any empty square.

Goal test: 8 queens are on the board, none attacked.

Path cost: Zero (search cost only exists)

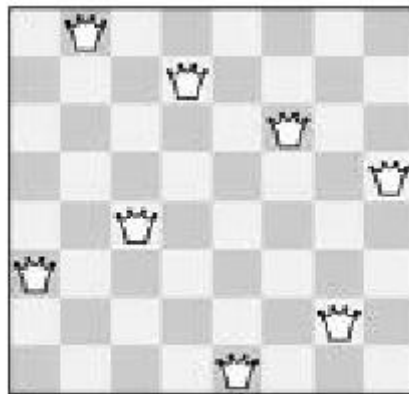


Fig 1.6 Solution to the 8 queens problem

Production system is a mechanism that describes and performs the search process.

A production system consists of four basic components:

1. A set of rules of the form $C_i \rightarrow A_i$ where C_i is the condition part and A_i is the action part. The condition determines when a given rule is applied, and the action determines what happens when it is applied.

(i.e A set of rules, each consist of left side (a pattern) that determines the applicability of the rule and a right side that describes the operations to be performed if the rule is applied)
2. One or more knowledge databases that contain whatever information is relevant for the given problem. Some parts of the database may be permanent, while others may temporary and only exist during the solution of the current problem. The information in the databases may be structured in any appropriate manner.
3. A control strategy that determines the order in which the rules are applied to the database, and provides a way of resolving any conflicts that can 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 one 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 be solved by searching for a path through the space from the initial state to a goal state.
- ❖ The process of solving the problem can be usefully modeled as a production system.

1 Control strategies

By considering control strategies we can decide which rule to apply next during the process of searching for a solution to problem.

The two requirements of good control strategy are that

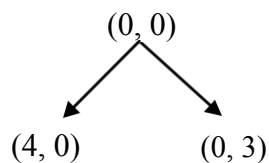
- ❖ **It should cause motion:** consider water jug problem, if we implement control strategy of starting each time at the top of the list of rules, it will never leads to solution. So we need to consider control strategy that leads to solution.
- ❖ **It should be systematic:** choose at random from among the applicable rules. This strategy is better than the first. It causes the motion. It will lead to the solution eventually. Doing like this is not a systematic approach and it leads to useless sequence of operators several times before finding final solution.

1.1 Systematic control strategy for the water jug problem

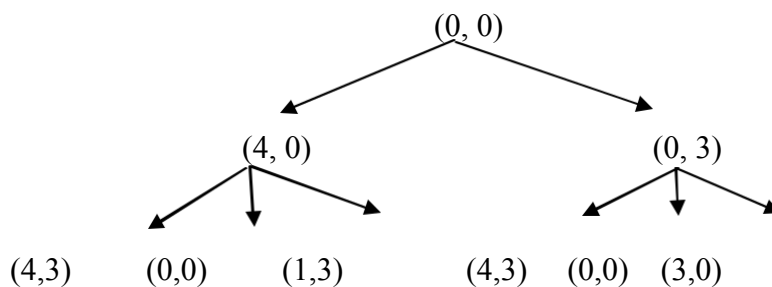
1.1.1 Breadth First Search (Blind Search)

Let us discuss these strategies using water jug problem. These may be applied to any search problem.

- ❖ Construct a tree with the initial state as its root.
- ❖ Generate all the offspring of the root by applying each of the applicable rules to the initial state.



- ❖ Now for each leaf node, generate all its successors by applying all the rules that are appropriate.



- ❖ Continue this process until some rule produces a goal state.

Algorithm

1. Create a variable called NODE-LIST and set it to initial state.
2. Unit a goal state is found or NODE-LIST is empty do
 - a. Remove the first element from NODE-LIST and call it E. if NODE-LIST is empty, quit.
 - b. For each way that each rule can match the state described in E do:
 - i. Apply the rule to generate a new state.
 - ii. If the new state is a goal state, quit and return this state.
 - iii. Otherwise, add the new state to the end of NODE-LIST.

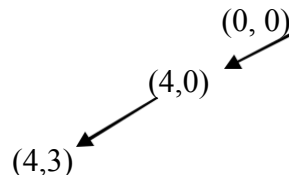
1.1.2 Depth First Search

Algorithm

1. If the initial state is the 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 successor, signal failure.
 - b. Call depth first search, with E as the initial state.
 - c. If the success is returned, signal success. Otherwise continue in this loop.

Backtracking

- ❖ In this search, we pursue a single branch of the tree until it yields a solution or until a decision to terminate the path is made.
- ❖ It makes sense to terminate a path if it reaches dead-end, produces a previous state. In such a state backtracking occurs.
- ❖ Chronological Backtracking: order in which steps are undone depends only on the temporal sequence in which steps were initially made.
- ❖ Specifically most recent step is always the first to be undone. This is also simple backtracking.



Advantages of Depth First search

- ❖ DFS requires less memory since only the nodes on the current path are stored.
- ❖ By chance DFS may find a solution without examining much of the search space at all.

Advantages of Breath First search

- ❖ BFS cannot be trapped exploring a blind alley.
- ❖ If there is a solution, BFS is guaranteed to find it.
- ❖ If there are multiple solutions, then a minimal solution will be found.

Traveling Salesman Problem (with 5 cities):

A salesman is supposed to visit each of 5 cities shown below. There is a road between each pair of cities and the distance is given next to the roads. Start city is A. The problem is to find the shortest route so that the salesman visits each of the cities only once and returns to back to A.

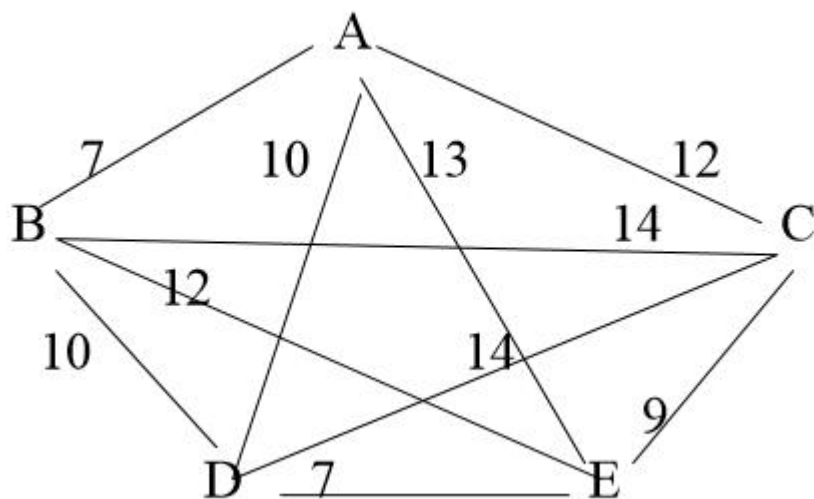


Fig Travelling Salesman Problem

- A simple, motion causing and systematic control structure could, in principle solve this problem.
- Explore the search tree of all possible paths and return the shortest path.
- This will require $4!$ paths to be examined.
- If number of cities grow, say 25 cities, then the time required to wait a salesman to get the information about the shortest path is of $O(24!)$ which is not a practical situation.
- This phenomenon is called combinatorial explosion.
- We can improve the above strategy as follows.

Branch and Bound

- ❖ Begin generating complete paths, keeping track of the shortest path found so far.
- ❖ Give up exploring any path as soon as its partial length become greater than the shortest path found so far.

* This algorithm is efficient than the first one, still requires exponential time \propto some number raised to N.

2 Heuristic Search

- Heuristics are criteria for deciding which among several alternatives be the most effective in order to achieve some goal.
- Heuristic is a technique that improves the efficiency of a search process possibly by sacrificing claims of systematic and completeness. It no longer guarantees to find the best answer but almost always finds a very good answer.
- Using good heuristics, we can hope to get good solution to hard problems (such as travelling salesman) in less than exponential time.
- There are **general-purpose** heuristics that are useful in a wide variety of problem domains.
- We can also construct **special purpose** heuristics, which are domain specific.

2.1 General Purpose Heuristics

- A general-purpose heuristics for combinatorial problem is nearest neighbor algorithms which works by selecting the locally superior alternative.
- For such algorithms, it is often possible to prove an upper bound on the error which provide reassurance that one is not paying too high a price in accuracy for speed.
- In many AI problems, it is often hard to measure precisely the goodness of a particular solution.
- For real world problems, it is often useful to introduce heuristics based on relatively unstructured knowledge. It is impossible to define this knowledge in such a way that mathematical analysis can be performed.
- In AI approaches, behavior of algorithms are analyzed by running them on computer as contrast to analyzing algorithm mathematically.
- There are at least many reasons for the adhoc approaches in AI.
 - ❖ It is a lot more fun to see a program do something intelligent than to prove it.
 - ❖ AI problem domains are usually complex, so generally not possible to produce analytical proof that a procedure will work.
 - ❖ It is even not possible to describe the range of problems well enough to make statistical analysis of program behavior meaningful.
- But still it is important to keep performance question in mind while designing algorithm.
 - One of the most important analysis of the search process is straightforward i.e., —Number of nodes in a complete search tree of depth D and branching factor F is F^D .

•This simple analysis motivates to



Look for improvements on the exhaustive search.



Find an upper bound on the search time which can be compared with exhaustive search procedures.

Best-First Search:

It is a general heuristic based search technique. In best first search, in the graph of problem representation, one evaluation function (which corresponds to heuristic function) is attached with every node. The value of evaluation function may depend upon cost or distance of current node from goal node. The decision of which node to be expanded depends on the value of this evaluation function. The best first can understood from following tree. In the tree, the attached value with nodes indicates utility value. The expansion of nodes according to best first search is illustrated in the following figure.

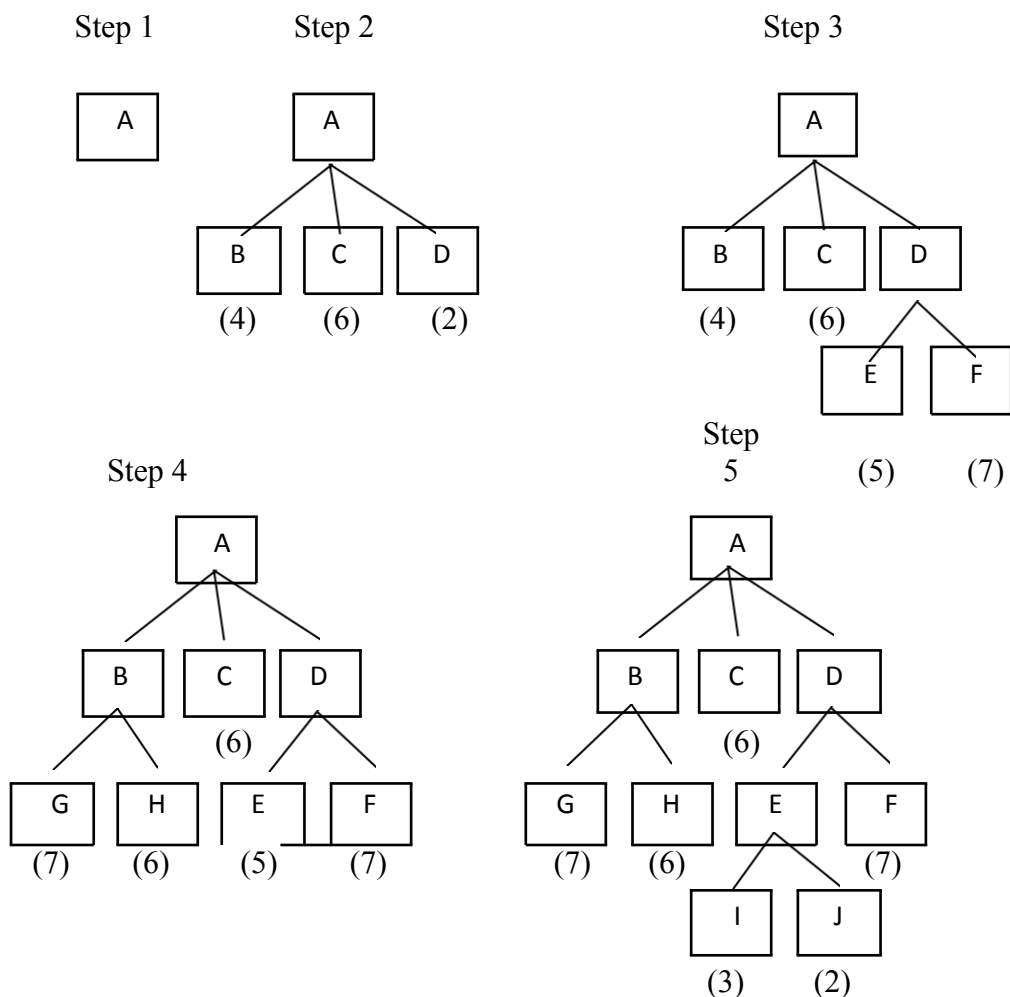


fig:1.10 Tree getting expansion according to best first search

Here, at any step, the most promising node having least value of utility function is chosen for expansion.

In the tree shown above, best first search technique is applied, however it is beneficial sometimes to search a graph instead of tree to avoid the searching of duplicate paths. In the process to do so, searching is done in a directed graph in which each node represents a point in the problem space. This graph is known as OR-graph. Each of the branches of an OR graph represents an alternative problem solving path.

Two lists of nodes are used to implement a graph search procedure discussed above. These are

- 1.OPEN: these are the nodes that have been generated and have had the heuristic function applied to them but not have been examined yet.
- 2.CLOSED: these are the nodes that have already been examined. These nodes are kept in the memory if we want to search a graph rather than a tree because whenever a node will be generated, we will have to check whether it has been generated earlier.

The best first search is a way of combining the advantage of both depth first and breadth first search. The depth first search is good because it allows a solution to be found without all competing branches have to be expanded. Breadth first search is good because it does not get trapped on dead ends of path. The way of combining this is to follow a single path at a time but switches between paths whenever some competing paths looks more promising than current one does. Hence at each step of best first search process, we select most promising node out of successor nodes that have been generated so far.

The functioning of best first search is summarized in the following steps:

- 1.It maintains a list open containing just the initial state.
- 2.Until a goal is found or there are no nodes left in open list do:
 - a. Pick the best node from open,
 - b. Generate its successor, and for each successor:
 - Check, and if it has not been generated before evaluate it and add it to open and record its parent.
 - If it has been generated before, and new path is better than the previous parent then change the parent.

The algorithm for best first search is given as follows:

Algorithm: Best first search

- 1.Put the initial node on the list say `_OPEN`.
- 2.If (OPEN = empty or OPEN= goal) terminate search, else
- 3.Remove the first node from open(say node is a)
- 4.If (a=goal) terminate search with success else

5. Generate all the successor nodes of $_a$. Send node $_a$ to a list called $_CLOSED$. Find out the value of heuristic function of all nodes. Sort all children generated so far on the basis of their utility value. Select the node of minimum heuristic value for further expansion.

6. Go back to step 2.

The best first search can be implemented using priority queue. There are variations of best first search. Examples of these are greedy best first search, A* and recursive best first search.

The A* Algorithm:

The A* algorithm is a specialization of best first search. Its most widely known form is best first search. It provides general guidelines about how to estimate goal distance for a general search graph. At each node along a path to the goal node, the A* algorithm generates all successor nodes and computes an estimate of distance (cost) from the start node to a goal node through each of

the successors. It then chooses the successor with the shortest estimated distance from expansion. It calculates the heuristic function based on the distance of the current node from the start node and the distance of the current node to the goal node.

The form of the heuristic estimation function for A* is defined as follows:

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

where $f(n)$ = evaluation function

$g(n)$ = cost (or distance) of current node from start node.

$h(n)$ = cost of current node from goal node.

In A* algorithm, the most promising node is chosen for expansion. The promising node is decided based on the value of the heuristic function. Normally, the node having the lowest value of $f(n)$ is chosen for expansion. We must note that the goodness of a move depends upon the nature of the problem; in some problems, the node having the least value of the heuristic function would be the most promising node, while in some situations, the node having the maximum value of the heuristic function is chosen for expansion. A* algorithm maintains two lists. One stores the list of open nodes and the other maintains the list of already expanded nodes. A* algorithm is an example of an optimal search algorithm. A search algorithm is optimal if it has an admissible heuristic. An algorithm has an admissible heuristic if its heuristic function $h(n)$ never overestimates the cost to reach the goal. Admissible heuristics are always optimistic because in them, the cost of solving the problem is less than what actually is. The A* algorithm works as follows:

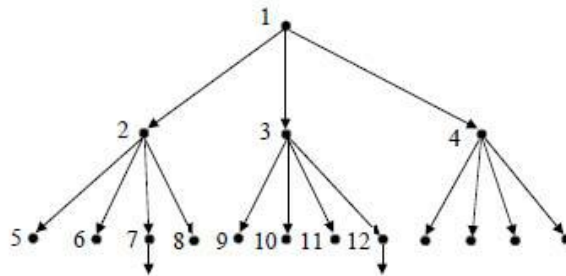
A* algorithm:

1. Place the starting node $_s$ on $_OPEN$ list.

- 2.If OPEN is empty, stop and return failure.
- 3.Remove from OPEN the node n' that has the smallest value of $f^*(n)$. if node n' is a goal node, return success and stop otherwise.
- 4.Expand n' generating all of its successors n'' and place n'' on CLOSED. For every successor n'' if n'' is not already OPEN , attach a back pointer to n' . compute $f^*(n)$ and place it on CLOSED.
- 5.Each n' that is already on OPEN or CLOSED should be attached to back pointers which reflect the lowest $f^*(n)$ path. If n' was on CLOSED and its pointer was changed, remove it and place it on OPEN.
- 6.Return to step 2.

Uniformed or Blind search

Breadth First Search (BFS): BFS expands the leaf node with the lowest path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple queue (“first in, first out”).



This is guaranteed to find an optimal path to a goal state. It is memory intensive if the state space is large. If the typical branching factor is b , and the depth of the shallowest goal state is d – the space complexity is $O(b^d)$, and the time complexity is $O(b^d)$.

BFS is an easy search technique to understand. The algorithm is presented below.

```

breadth_first_search ()
{
    store initial state in queue Q

    set state in the front of the Q as current state ;

    while (goal state is reached OR Q is empty)
    {
        apply rule to generate a new state from the current
        state ;

        if (new state is goal state) quit ;

        else if (all states generated from current states are
        exhausted)
        {
            delete the current state from the Q ;

            set front element of Q as the current state ;
        }
    }
}

```

```

else continue ;

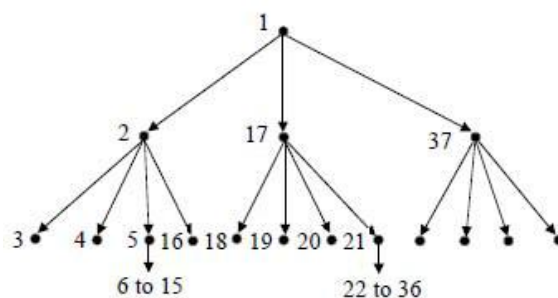
}

}

```

The algorithm is illustrated using the bridge components configuration problem. The initial state is PDFG, which is not a goal state; and hence set it as the current state. Generate another state DPDFG (by swapping 1st and 2nd position values) and add it to the list. That is not a goal state, hence; generate next successor state, which is FDPG (by swapping 1st and 3rd position values). This is also not a goal state; hence add it to the list and generate the next successor state GDFP. Only three states can be generated from the initial state. Now the queue Q will have three elements in it, viz., DPDFG, FDPG and GDFP. Now take DPDFG (first state in the list) as the current state and continue the process, until all the states generated from this are evaluated. Continue this process, until the goal state DGPF is reached. The 14th evaluation gives the goal state. It may be noted that, all the states at one level in the tree are evaluated before the states in the next level are taken up; i.e., the evaluations are carried out breadth-wise. Hence, the search strategy is called breadth-first search.

Depth First Search (DFS): DFS expands the leaf node with the highest path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple stack (“last in, first out”).



This is not guaranteed to find any path to a goal state. It is memory efficient even if the state space is large. If the typical branching factor is b , and the maximum depth of the tree is m – the space complexity is $O(bm)$, and the time complexity is $O(b^m)$. In DFS, instead of generating all the states below the current level, only the first state below the current level is generated and evaluated recursively. The search continues till a further successor cannot be generated.

Then it goes back to the parent and explores the next successor. The algorithm is given below.

```
depth_first_search ()
```

```
{
```

```
    set initial state to current state ;
```

```
    if (initial state is current state) quit ;
```

```

else
{
    if (a successor for current state exists)
    {
        generate a successor of the current state and
        set it as current state ;
    }
    else return ;
    depth_first_search (current_state) ;
    if (goal state is achieved) return ;
    else continue ;
}
}

```

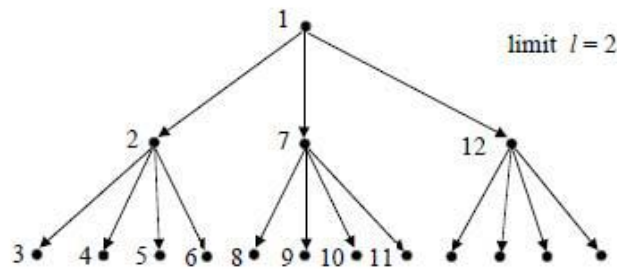
Since DFS stores only the states in the current path, it uses much less memory during the search compared to BFS.

The probability of arriving at goal state with a fewer number of evaluations is higher with DFS compared to BFS. This is because, in BFS, all the states in a level have to be evaluated before states in the lower level are considered. DFS is very efficient when more acceptable solutions exist, so that the search can be terminated once the first acceptable solution is obtained.

BFS is advantageous in cases where the tree is very deep.

An ideal search mechanism is to combine the advantages of BFS and DFS.

Depth Limited Search (DLS): DLS is a variation of DFS. If we put a limit l on how deep a depth first search can go, we can guarantee that the search will terminate (either in success or failure).



If there is at least one goal state at a depth less than l , this algorithm is guaranteed to find a goal state, but it is not guaranteed to find an optimal path. The space complexity is $O(bl)$, and the time complexity is $O(b^l)$.

Depth First Iterative Deepening Search (DFIDS): DFIDS is a variation of DLS. If the lowest depth of a goal state is not known, we can always find the best limit l for DLS by trying all possible depths $l = 0, 1, 2, 3, \dots$ in turn, and stopping once we have achieved a goal state.

This appears wasteful because all the DLS for l less than the goal level are useless, and many states are expanded many times. However, in practice, most of the time is spent at the deepest part of the search tree, so the algorithm actually combines the benefits of DFS and BFS.

Because all the nodes are expanded at each level, the algorithm is complete and optimal like BFS, but has the modest memory requirements of DFS. Exercise: if we had plenty of memory, could/should we avoid expanding the top level states many times?

The space complexity is $O(bd)$ as in DLS with $l = d$, which is better than BFS. The time complexity is $O(b^d)$ as in BFS, which is better than DFS.

Bi-Directional Search (BDS): The idea behind bi-directional search is to search simultaneously both forward from the initial state and backwards from the goal state, and stop when the two BFS searches meet in the middle.

This is not always going to be possible, but is likely to be feasible if the state transitions are reversible. The algorithm is complete and optimal, and since the two

search depths are $\sim d/2$, it has space complexity $O(b^{d/2})$, and time complexity $O(b^{d/2})$. However, if there is more than one possible goal state, this must be factored into the complexity.

Repeated States: In the above discussion we have ignored an important complication that often arises in search processes – the possibility that we will waste time by expanding states that have already been expanded before somewhere else on the search tree.

For some problems this possibility can never arise, because each state can only be reached in one way.

For many problems, however, repeated states are unavoidable. This will include all problems where the transitions are reversible, e.g.

$((1\ 2\ 3)) \rightarrow ((1)\ (2\ 3)) \rightarrow ((1\ 2\ 3)) \rightarrow ((1)\ (2\ 3)) \rightarrow ((1\ 2\ 3)) \rightarrow \dots$

The search trees for these problems are infinite, but if we can prune out the repeated states, we can cut the search tree down to a finite size. We effectively only generate a portion of the search tree that matches the state space graph.

Avoiding Repeated States: There are three principal approaches for dealing with repeated states:

- ❖ Never return to the state you have just come from

The node expansion function must be prevented from generating any node successor that is the same state as the node's parent.

- ❖ Never create search paths with cycles in them

The node expansion function must be prevented from generating any node successor that is the same state as any of the node's ancestors.

- ❖ Never generate states that have already been generated before

This requires that every state ever generated is remembered, potentially resulting in space complexity of $O(b^d)$.

Comparing the Uninformed Search Algorithms: We can now summarize the properties of our five uninformed search strategies:

Strategy	Complete	Optimal	Time Complexity	Space Complexity
BFS	Yes	Yes	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
DLS	If $l \geq d$	No	$O(b^l)$	$O(bl)$
DFIDS	Yes	Yes	$O(b^d)$	$O(bd)$
BDS	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

Simple BFS and BDS are complete and optimal but expensive with respect to space and time.

DFS requires much less memory if the maximum tree depth is limited, but has no guarantee of finding any solution, let alone an optimal one. DLS offers an improvement over DFS if we have some idea how deep the goal is.

The best overall is DFID which is complete, optimal and has low memory requirements, but still exponential time.

Informed search

Informed search uses some kind of evaluation function to tell us how far each expanded state is from a goal state, and/or some kind of heuristic function to help us decide which state is likely to be the best one to expand next.

The hard part is to come up with good evaluation and/or heuristic functions. Often there is a natural evaluation function, such as distance in miles or number objects in the wrong position.

Sometimes we can learn heuristic functions by analyzing what has worked well in similar previous searches.

The simplest idea, known as greedy best first search, is to expand the node that is already closest to the goal, as that is most likely to lead quickly to a solution. This is like DFS in that it attempts to follow a single route to the goal, only attempting to try a different route when it reaches a dead end. As with DFS, it is not complete, not optimal, and has time and complexity of $O(b^m)$. However, with good heuristics, the time complexity can be reduced substantially.

Branch and Bound: An enhancement of backtracking. Applicable to optimization problems.

For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution).

Uses the bound for:

- Ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far.
- Guiding the search through state-space.

The search path at the current node in a state-space tree can be terminated for any one of the following three reasons:

- The value of the node’s bound is not better than the value of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point and hence we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Best-First branch-and-bound:

- A variation of backtracking.
- Among all the nonterminated leaves, called as the live nodes, in the current tree, generate all the children of the most promising node, instead of generation a single child of the last promising node as it is done in backtracking.
- Consider the node with the best bound as the most promising node.

A* Search: Suppose that, for each node n in a search tree, an evaluation function $f(n)$ is defined as the sum of the cost $g(n)$ to reach that node from the start state, plus an estimated cost $h(n)$ to get from that state to the goal state. That $f(n)$ is then the estimated cost of the cheapest solution through n .

A* search, which is the most popular form of best-first search, repeatedly picks the node with the lowest $f(n)$ to expand next. It turns out that if the heuristic function $h(n)$ satisfies certain conditions, then this strategy is both complete and optimal.

In particular, if $h(n)$ is an admissible heuristic, i.e. is always optimistic and never overestimates the cost to reach the goal, then A* is optimal.

The classic example is finding the route by road between two cities given the straight line distances from each road intersection to the goal city. In this case, the nodes are the intersections, and we can simply use the straight line distances as $h(n)$.

Real-Time A*

Real-time A* is a variation of A*.

Search continues on the basis of choosing paths that have minimum values of $f(\text{node}) = g(\text{node}) + h(\text{node})$. However, $g(\text{node})$ is the distance of the node from the current node, rather than from the root node.

Hence, the algorithm will backtrack if the cost of doing so plus the estimated cost of solving the problem from the new node is less than the estimated cost of solving the problem from the current node.

Implementing real-time A* means maintaining a hash table of previously visited states with their $h(\text{node})$ values.

Iterative-Deepening A* (IDA*)

By combining iterative-deepening with A*, we produce an algorithm that is optimal and complete (like A*) and that has the low memory requirements of depth-first search.

IDA* is a form of iterative-deepening search where successive iterations impose a greater limit on $f(\text{node})$ rather than on the depth of a node.

IDA* performs well in problems where the heuristic value $f(\text{node})$ has relatively few possible values.

For example, using the Manhattan distance as a heuristic in solving the eight-queens problem, the value of $f(\text{node})$ can only have values 1, 2, 3, or 4.

In this case, the IDA* algorithm only needs to run through a maximum of four iterations, and it has a time complexity not dissimilar from that of A*, but with a significantly improved space complexity because it is effectively running depth-first search.

In cases such as the traveling salesman problem where the value of $f(\text{node})$ is different for every state, the IDA* method has to expand $1 + 2 + 3 + \dots + n$ nodes = $O(n^2)$ where A* would expand n nodes.

GAME PLAYING

Introduction

Game Playing is one of the oldest sub-fields in AI. Game playing involves abstract and pure form of competition that seems to require intelligence. It is easy to represent the states and actions. To implement the game playing very little world knowledge is required.

The most common used AI technique in game is search. Game playing research has contributed ideas on how to make the best use of time to reach good decisions.

Game playing is a search problem defined by:

- Initial state of the game

- Operators defining legal moves

- Successor function

- Terminal test defining end of game states

- Goal test

- Path cost/utility/payoff function

More popular games are too complex to solve, requiring the program to take its best guess. “ for example in chess, the search tree has 1040 nodes (with branching factor of 35). It is the opponent because of whom uncertainty arises.

Characteristics of game playing

3. There are always an “unpredictable” opponent:

- The opponent introduces uncertainty

- The opponent also wants to win

The solution for this problem is a strategy, which specifies a move for every possible opponent reply.

2. Time limits:
-

Game are often played under strict time constraints (eg:chess) and therefore must be very effectively handled.

There are special games where two players have exactly opposite goals. There are also perfect information games(sch as chess and go) where both the players have access to the same information about the game in progress (e.g. tic-tac-toe). In imoerfect game information games (such as bridge or certain card games and games where dice is used). Given sufficient time and space, usually an optimum solution can be obtained for the former by exhaustive search, though not for the latter.

Types of games

There are basically two types of games

Deterministic games

Chance games

Game like chess and checker are perfect information deterministic games whereas games like scrabble and bridge are imperfect information. We will consider only two player discrete, perfect information games, such as tic-tac-toe, chess, checkers etc... . Two- player games are easier to imagine and think and more common to play.

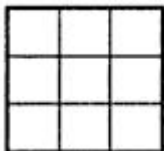
Minimize search procedure

Typical characteristic of the games is to look ahead at future position in order to succeed. There is a natural correspondence between such games and state space problems.

In a game like tic-tac-toe

- States-legal board positions
- Operators-legal moves
- Goal-winning position

The game starts from a specified initial state and ends in position that can be declared win for one player and loss for other or possibly a draw. Game tree is an explicit representation of all possible plays of the game. We start with a 3 by 3 grid..



Then the two players take it in turns to place a there marker on the board(one player uses the „X“ marker, the other uses the „O“ marker). The winner is the player who gets 3 of these markers in a row, eg.. if X wins

X	O	
	X	O
O		X

Another possibility is that no1 wins eg..

O	O	O
X	X	O
O	O	X

Or the third possibility is a draw case

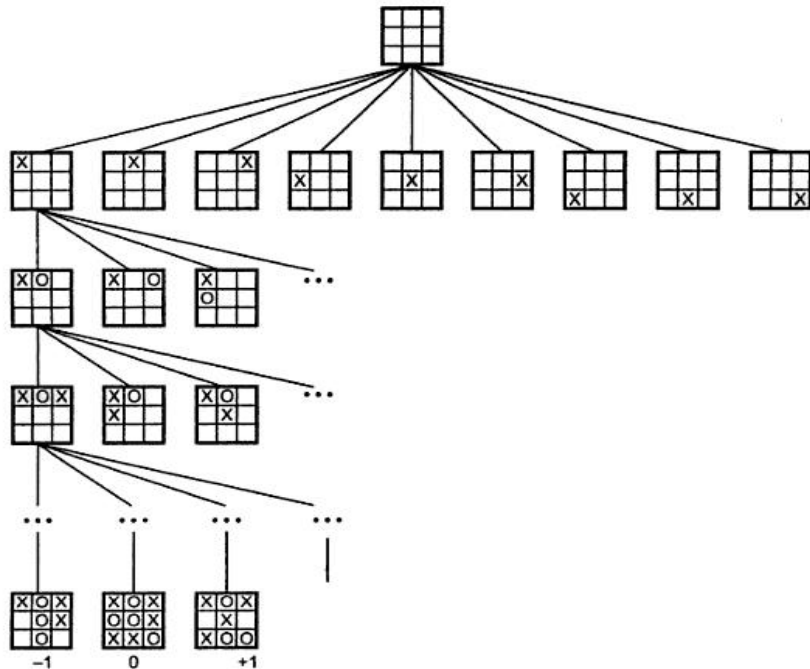
X	O	X
X	O	O
O	X	O

Search tree for tic-tac-toe

The root node is an initial position of the game. Its successors are the positions that the first player can reach in one move; their successors are the positions resulting from the second player's replies and so on. Terminal or leaf nodes are presented by WIN, LOSS or DRAW. Each path from the root ro a terminal node represents a different complete play of the game. The moves available to one player from a given position can be represented by OR links whereas the moves available to his opponent are AND links.

The trees representing games contain two types of nodes:

- MAX- nodes (assume at even level from root)
 - MIN - nodes [assume at odd level from root)
-



Search tree for tic-tac-toe

the leaves nodes are labeled WIN, LOSS or DRAW depending on whether they represent a win, loss or draw position from Max's viewpoint. Once the leaf nodes are assigned their WIN-LOSS or DRAW status, each nodes in the game tree can be labeled WIN, LOSS or DRAW by a bottom up process.

Game playing is a special type of search, where the intention of all players must be taken into account.

Minimax procedure

- Starting from the leaves of the tree (with final scores with respect to one player, MAX), and go backwards towards the root.
- At each step, one player (MAX) takes the action that leads to the highest score, while the other player(MIN) takes the action that leads to the lowest score.
- All the nodes in the tree will be scored and the path from root to the actual result is the one on which all node have the same score.

The minimax procedure operates on a game tree and is recursive procedure where a player tries to minimize its opponent's advantage while at the same time maximize its own. The player hoping for positive number is called the maximizing player. His opponent is the minimizing player. If the player to move is the maximizing player, he is looking for a path leading to a large positive number and his opponent will try to force the play toward situation with strongly

The procedure by which the scoring information passes up the game tree is called the MINIMAX procedures since the score at each node is either minimum or maximum of the scores at the nodes immediately below

In this fig since it is the maximizing search ply 8 is transferred upwards to A

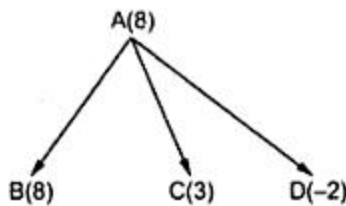


Diagram illustrating a minimax search tree. The tree structure is as follows:

- Level 0 (MAX): Root node labeled 3.
- Level 1 (MIN): Three child nodes, all labeled 1.
- Level 2 (MAX): Grandchild nodes.
 - From the first MIN node (1): two children labeled 2 and 5.
 - From the second MIN node (1): three children labeled 3, 1, and 4.
 - From the third MIN node (1): two children labeled 4 and 3.

An arrow points to the third MIN node (1) with the text "Select this move".

To play an entire game we need to combine search oriented and non-search oriented techniques. The idea way to use a search procedure to find a solution to the problem statement is to generate moves through the problem space until a goal state is reached. Unfortunately for games like

chess even with a good plausible move generator, it is not possible to search until goal state is reached. In the amount of time available it is possible to generate the tree at the most 10 to 20 ply 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 the static evaluation function. The static evaluation function evaluates individual board positions by estimating how much likely they are eventually to lead to a win.

The minimax procedure is a depth-first, depth limited search procedure.

- If the limit of search has reached, compute the static value of the current position relative to the appropriate layer as given below (maximizing or minimizing player). Report the result (value and path).
- If the level is minimizing level(minimizer's turn)
- Generate the successors of the current position. Apply MINIMAX to each of the successors. Return the minimum of the result.
- If the level is a maximizing level. Generate the successors of current position

Apply MINIMAX to each of these successors. Return the maximum of the result. The maximum algorithm uses the following procedures

MOVEGEN(POS)

It is plausible move generator. It returns a list of successors of „Pos“.

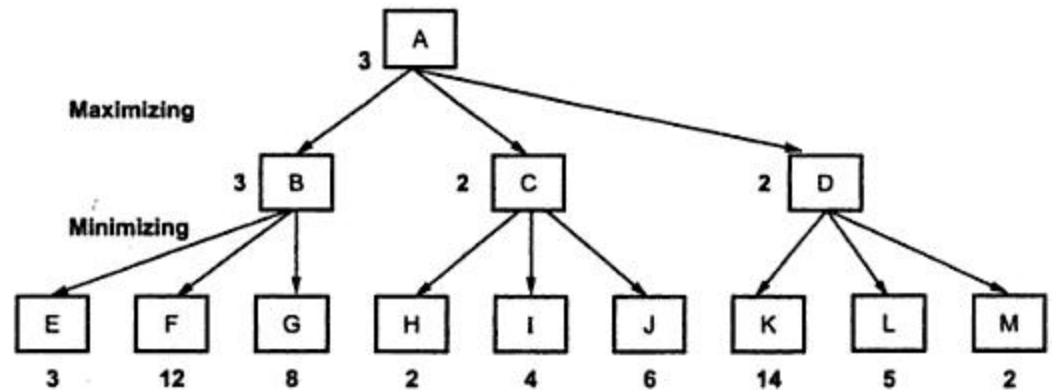
STSTIC (Pos, Depth)

The static evaluation function that returns a number representing the goodness of „pos“ from the current point of view.

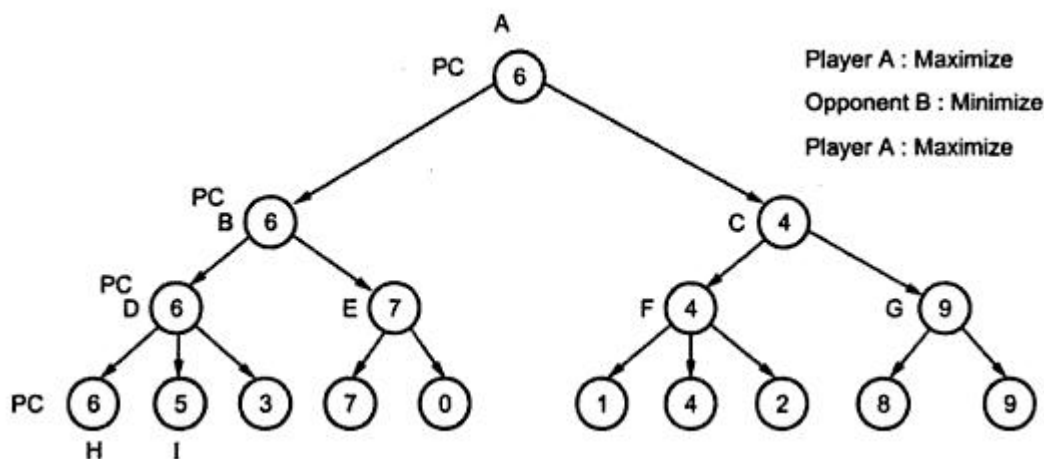
DEEP-ENOUGH

It returns true if the search to be stopped at the current level else it returns false.

A MINIMAX example



Another example of minimax search procedure



In the above example, a Minimax search in a game tree is simulated. Every leaf has a corresponding value, which is approximated from player A's view point. When a path is chosen, the value of the child will be passed back to the parent. For example, the value for D is 6, which is the maximum value of its children, while the value for C is 4 which is the minimum value of F and G. In this example the best sequence of moves found by the maximizing/minimizing procedure is the path through nodes A, B, D and H, which is called the principal continuation. The nodes on the path are denoted as PC (principal continuation) nodes. For simplicity we can modify the game tree values slightly and use only maximization operations. The trick is to maximize the scores by negating the returned values from the children instead of searching for minimum scores and estimate the values at leaves from the player's own viewpoint

Alpha-beta cutoffs

The basic idea of alpha-beta cutoffs is “It is possible to compute the correct minimax decision without looking at every node in the search tree”. This is called pruning (allow us to ignore portions of the search tree that make no difference to the final choice).

The general principle of alpha-beta pruning is

- ⊗ Consider a node n somewhere in the tree, such that a player has a chance to move to this node.
- ⊗ If player has a better chance m either at the parent node of n (or at any choice point further up) then n will never be reached in actual play.

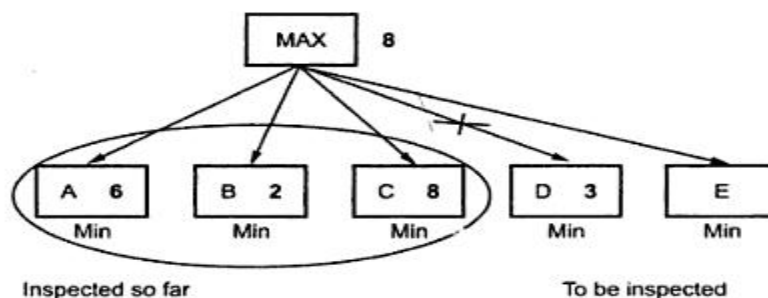
When we are doing a search with alpha-beta cut-offs, if a node's value is too high, the minimizer will make sure it's never reached (by turning off the path to get a lower value). Conversely, if a node's value is too low, the maximizer will make sure it's never reached. This gives us the following definitions

Alpha: the highest value that the maximize can guarantee himself by making some move at the current node OR at some node earlier on the path to this node.

Beta: the lowest value that the minimizer can guarantee by making some move at the current node OR at some node earlier on the path to this node.

The maximize is constantly trying to push the alpha value up by finding better moves; the minimizer is trying to push the beta value down. If a node's value is between alpha and beta, then the players might reach it. At the beginning, at the root of the tree, we don't have any guarantees yet about what values the maximizer and minimizer can achieve. So we set beta to ∞ and alpha to $-\infty$. Then as we move down the tree, each node starts with beta and alpha values passed down from its parent.

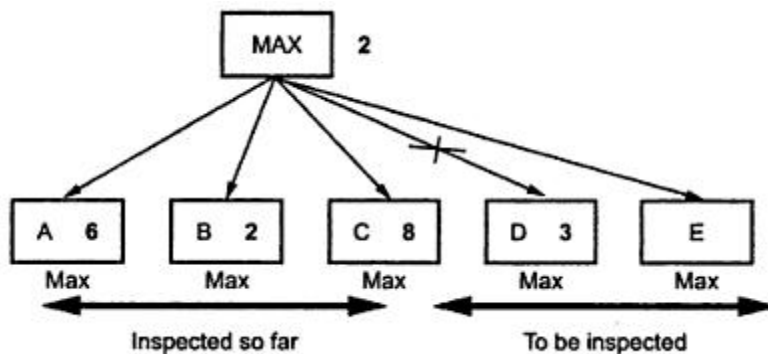
Consider a situation in which the MIN – children of a MAX-node have been partially inspected



Alpha-beta for a max node

At this point the “tentative” value which is backed up so far of F is 8. MAX is not interested in any move which has a value of less than 8, since it is already known that 8 is the worst that MAX

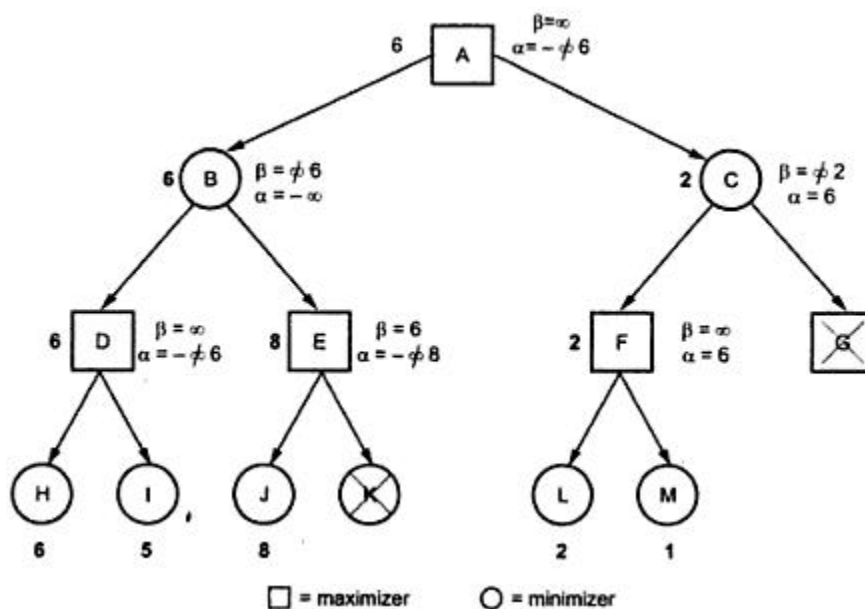
can do, so far. Thus the node D and all its descendent can be pruned or excluded from further exploration, since MIN will certainly go for a value of 3 rather than 8. Similarly for a MIN-node:



Alpha-beta for a min node

MIN is trying to minimize the game-value. So far, the value 2 is the best available from MIN's point of view. MIN will immediately reject node D, which can be stopped for further exploration.

In a game tree, each node represents a board position where one of the players gets to choose a move. For example, in the fig below look at the node C. As soon as we look at its left child, we realize that if the players reach node C, the minimizer can limit the utility to 2. But the maximize can get utility 6 by going to node B instead, so he would never let the game reach C. therefore we don't even have to look at C's other children



Tree with alpha-beta cut-offs

Initially at the root of the tree, there is no guarantee about what values the maximizer and minimizer can achieve. So beta is set to ∞ and alpha to $-\infty$. Then as we move down the tree, each node starts with beta and alpha values passed down from its parent. If it's a maximize node, and then alpha is increased if a child value is greater than the current alpha value. Similarly, at a minimizer node, beta may be decreased. This is shown in the fig.

At each node, the alpha and beta values may be updated as we iterate over the node's children. At node E, when alpha is updated to a value of 8, it ends up exceeding beta. This is a point where alpha beta pruning is required we know the minimizer would never let the game reach this node so we don't have to look at its remaining children. In fact, pruning happens exactly when the alpha and beta lines hit each other in the node value.

Algorithm-Alpha-beta

```
int AlphaBeta(int depth, int alpha, int beta)
{
    if (depth == 0)
        return Evaluate();
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = -AlphaBeta(depth - 1, -beta, -alpha);
        UnmakeMove();
        if (val >= beta)
            return beta;
        if (val > alpha)
            alpha = val;
    }
    return alpha;
}
```

If the highlighted characters are removed, what is left is a min-max function. The function is passed the depth it should search, and $-\text{INFINITY}$ as alpha and $+\text{INFINITY}$ as beta.

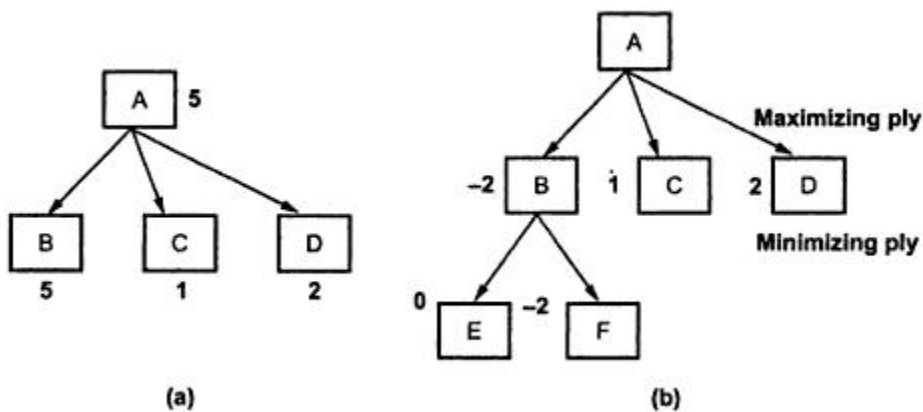
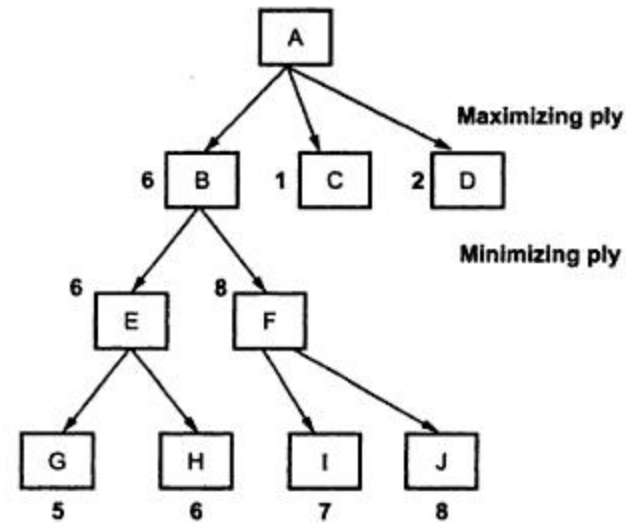
```
val = AlphaBeta(5, -INFINITY, INFINITY);
```

This does a five-ply search

The Horizon effect

A potential problem in game tree search to a fixed depth is the horizon effect, which occurs when there is a drastic change in value immediately beyond the place where the algorithm stops

searching. Consider the tree shown in the below fig.A. it has nodes A, B, C and D. at this level since it is a maximizing ply, the value which will be passed up at A is 5.



Suppose node B is examined one more level as shown in fig B. then we see because of a minimizing ply value at B is -2 and hence the value passed to A is 2. This results in a drastic change in the situation. There are two proposed solutions to this problem, neither very satisfactory.

Secondary search

One proposed solution is to examine the search beyond the apparently best one to see if something is looming just over the horizon. In that case we can revert to the second-best move. Obviously then the second-best move has the same problem and there is not time to search beyond all possible acceptable moves.

Waiting for Quiescence

If a position looks “dynamic”, don’t even bother to evaluate it.

Instead, do a small secondary search until things calm down.

E.g after capturing a piece, things look good, but this would be misleading if opponent was about to capture right back.

In general such factors are called continuation heuristics

Fig above shows the further exploration of the tree in fig B. Here now since the tree is further explored, the value, which is passed to A is 6. Thus the situation calms down. This is called as waiting for quiescence. This helps in avoiding the horizon effect of a drastic change of values.

Iterative Deepening

Rather than searching to a fixed depth in the game tree, first search only single ply, then apply MINMAX to 2 ply, further 3 ply till the final goal state is searched. This is called as iterative deepening. Besides providing good control of time, iterative deepening is usually more efficient than an equivalent direct search. The reason is that the results of previous iterations can improve the move ordering of new iteration, which is critical for efficient searching.

UNIT II Reasoning

Knowledge Representation

Representation and Mapping

Problem solving requires large amount of knowledge and some mechanism for manipulating that knowledge.

- The Knowledge and the Representation are distinct entities, play a 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 the *system's performance* in doing something.

Facts Truths about the real world and what we represent. This can be regarded as the knowledge level

In simple words, we :

need to know about *things we want to represent* , and

need some means by which *things we can manipulate*.

◇ know things to represent	‡ Objects	- facts about objects in the domain.
	‡ Events	- actions that occur in the domain.
	‡ Performance	- knowledge about how to do things
	‡ Meta- knowledge	- knowledge about what we know
◇ need means to manipulate	‡ Requires some formalism	- to what we represent ;

Thus, knowledge representation can be considered at two levels :

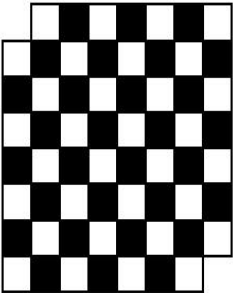
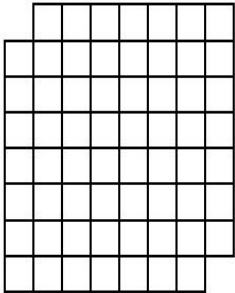
- knowledge level at which facts are described, and
- symbol level at which the representations of the objects, defined in terms of symbols, can be manipulated in the programs.

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

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.
 - Therefore, we must be able to map "*facts to symbols*" and "*symbols to facts*" using *forward and backward representation mapping*.
-

Example : Consider an English sentence



No. black squares = 30
No. white squares = 32

F

(a)

◆ Spot is a dog

◆ dog (Spot)

◆ $\forall x : \text{dog}(x) \rightarrow \text{hastail}(x)$

(b)

(c)

A fact represented in English sentence

Using forward mapping function the above fact is represented in logic

A logical representation of the fact that "all dogs have tails"

Now using deductive mechanism we can generate a new representation of object:

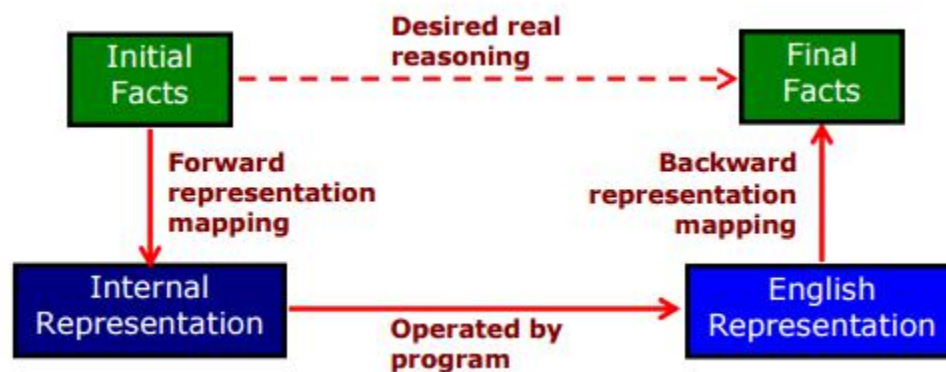
Hastail (Spot)	A new object representation
Spot has a tail [it is new knowledge]	Using backward mapping function to generate English sentence

Good representation can make a reasoning program trivial

The Mutilated Checkerboard Problem: “Consider a normal checker board from which two squares, in opposite corners, have been removed. The task is to cover all the remaining squares exactly with dominoes, each of which covers two squares. No overlapping, either of dominoes on top of each other or of dominoes over the boundary of the mutilated board are allowed. Can this task be done?”

Forward and Backward Representation

The forward and backward representations are elaborated below



- The dotted line on top indicates the abstract reasoning process that a program is intended to model.
- The solid lines on bottom indicate the concrete reasoning process that the program performs.

KR System Requirements

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

A knowledge representation system should have following properties.

Representational Adequacy the ability to represent all kinds of knowledge that are needed in that domain.

Inferential Adequacy the ability to manipulate the representational structures to derive new structure corresponding to new knowledge inferred from old.

Inferential Efficiency the ability to incorporate additional information into the knowledge structure that can be used to focus attention of the inference mechanisms in the most promising direction.

Acquisitional Efficiency the ability to acquire new knowledge using automatic methods whenever possible rather than reliance on human intervention

Note: To date no single system can optimize all of the above properties.

2.3 Knowledge Representation Schemes

There are four types of Knowledge representation :

Relational, Inheritable, Inferential, and Declarative/Procedural.

Relational Knowledge :

- 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.

Inheritable Knowledge

- is obtained from associated objects.
- it prescribes a structure in which new objects are created which may inherit all or a subset of attributes from existing objects.

Inferential Knowledge

- is inferred from objects through relations among objects.
- e.g., a word alone is a simple syntax, but with the help of other words in phrase the reader may infer more from a word; this inference within linguistic is called semantics.

Declarative Knowledge

- a statement in which knowledge is specified, but the use to which that knowledge is to be put is not given.
- e.g. laws, people's name; these are facts which can stand alone, not dependent on other knowledge;

Procedural Knowledge

- a representation in which the control information, to use the knowledge, is embedded in the knowledge itself.
- e.g. computer programs, directions, and recipes; these indicate specific use or implementation;

Relational Knowledge

This knowledge associates elements of one domain with another domain.

Relational knowledge is made up of objects consisting of attributes and their corresponding associated values.

The results of this knowledge type is a mapping of elements among different domains.

The facts about a set of objects are put systematically in columns.

This representation provides little opportunity for inference.

Table - Simple Relational Knowledge

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".

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 needs to be augmented with inference mechanism.

The KR in hierarchical structure, shown below, is called “*semantic network*” or a collection of “*frames*” or “*slot-and-filler structure*”. The structure shows property inheritance and way for insertion of additional knowledge.

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.

Baseball knowledge

– *isa* : show class inclusion

– *instance* : show class membership

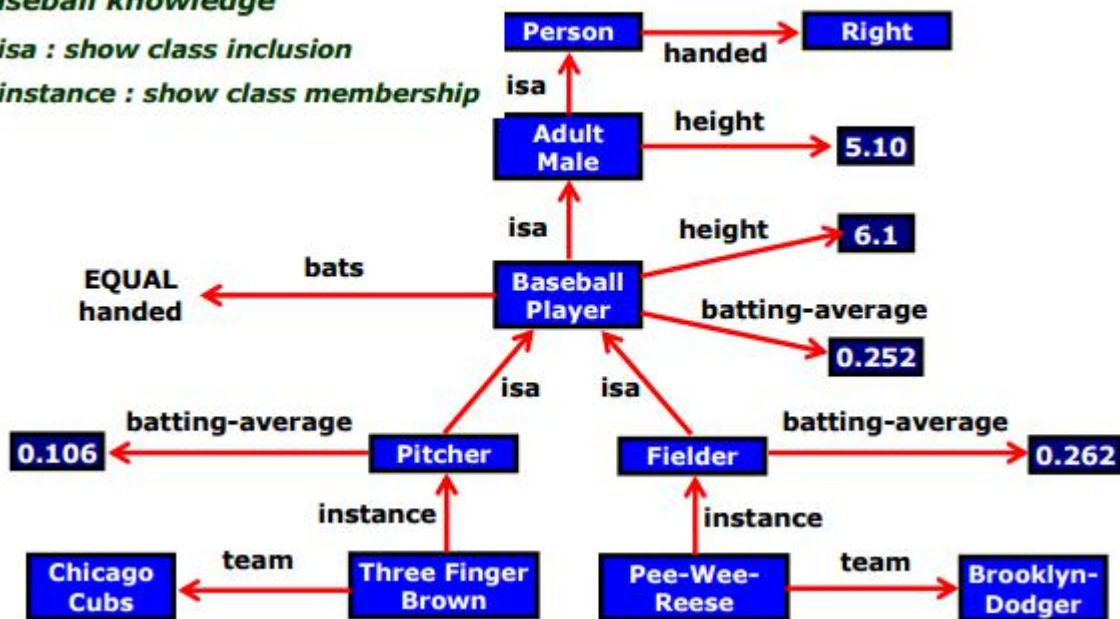


Fig. Inheritable knowledge representation (KR)

Ⓟ

The directed arrows represent *attributes* (*isa*, *instance*, *team*) originates at object being described and terminates at object or its value.

Ⓟ

The box nodes represents *objects* and *values* of the attributes.

Viewing a node as a frame

Example : Baseball-player

isa : Adult-Male

Bates : EQUAL handed

Height : 6.1

Batting-average : 0.252

Algorithm : Property Inheritance

Retrieve a value V for an attribute A of an instance object O

Steps to follow:

- < Find object O in the knowledge base.
- < If there is a value for the attribute A then report that value.
- < Else, if there is a value for the attribute instance; If not, then fail.
- < Else, move to the node corresponding to that value and look for a value for the attribute A ; If one is found, report it.
- 5. Else, do until there is no value for the “isa” attribute or until an answer is found :
 - Get the value of the “isa” attribute and move to that node.
 - See if there is a value for the attribute A ; If yes, report it.

This algorithm is simple. It describes the basic mechanism of inheritance. It does not say what to do if there is more than one value of the instance or “isa” attribute.

This can be applied to the example of knowledge base illustrated, in the previous slide, to derive answers to the following queries :

team (Pee-Wee-Reese) = Brooklyn–Dodger

batting–average(Three-Finger-Brown) = 0.106

height (Pee-Wee-Reese) = 6.1

bats (Three Finger Brown) = right

Inferential Knowledge

This knowledge generates new information from the given information.

This new information does not require further data gathering from 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.

the symbols used for the logic operations are :

" \rightarrow " (implication), " \neg " (not), " \vee " (or), " \wedge " (and),
" \forall " (for all), " \exists " (there exists).

Examples of predicate logic statements :

1. "Wonder" is a name of a dog : **dog (wonder)**
2. All dogs belong to the class of animals : **$\forall x : \text{dog}(x) \rightarrow \text{animal}(x)$**
3. All animals either live on land or in water : **$\forall x : \text{animal}(x) \rightarrow \text{live}(x, \text{land}) \vee \text{live}(x, \text{water})$**

From these three statements we can infer that :

" Wonder lives either on land or on water."

Note : If more information is made available about these objects and their relations, then more knowledge can be inferred.

Declarative/Procedural Knowledge

Differences between Declarative/Procedural knowledge is not very clear.

Declarative knowledge :

Here, the knowledge is based on declarative facts about *axioms* and *domains* .

axioms are assumed to be true unless a counter example is found to invalidate them.

domains represent the physical world and the perceived functionality.

- ✓ axiom and domains thus **simply exists** and serve as declarative statements that can stand alone.

Procedural knowledge:

Here, the knowledge is a mapping process between domains that specify "what to do when" and the representation is of "how to make it" rather than "what it is". The procedural knowledge :

- ✓ may have inferential efficiency, but no inferential adequacy and acquisitional efficiency.



are represented as small programs that know how to do specific things, how to proceed.

Example : A parser in a natural language has the knowledge that a noun phrase may contain articles, adjectives and nouns. It thus accordingly call routines that know how to process articles, adjectives and nouns.

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.

Important Attributes :

Any attribute of objects so basic that they occur in almost every problem domain ?

Relationship among attributes:

Any important relationship that exists among object attributes ?

Choosing Granularity :

At what level of detail should the knowledge be represented ?

Set of objects :

How sets of objects be represented ?

Finding Right structure :

Given a large amount of knowledge stored, how can relevant parts be accessed ?

- ❖ **Important Attributes**

There are attributes that are of general significance.

There are two attributes "instance" and "isa", that are of general importance. These attributes are important because they support property inheritance.

- ❖ **Relationship among Attributes**

The attributes to describe objects are themselves entities they represent.

The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:



- ❖ Inverses, existence in an isa hierarchy, techniques for reasoning about values and single valued attributes.

- ❖ **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. The figure shows attributes (isa, instance, and team), each with a directed arrow, originating at the object being described and terminating either at the object or its value.

There are two ways of realizing this:

- ⊗ first, represent two relationships in a single representation; e.g., a logical representation, team(Pee-Wee-Reese, Brooklyn–Dodgers), that can be interpreted as a statement about Pee-Wee-Reese or Brooklyn–Dodger.
- ⊗ second, use attributes that focus on a single entity but use them in pairs, one the inverse of the other; for e.g., one, team = Brooklyn– Dodgers , and the other, team = Pee-Wee-Reese,

This second approach is followed in semantic net and frame-based systems, accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slot by checking, each time a value is added to one attribute then the corresponding value is added to the inverse.

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.

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 for attributes are important because they support inheritance.

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 can not be greater than the age of person's parents.

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

Single valued attributes

This is about a specific attribute that is guaranteed to take a unique value.

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.



Choosing Granularity

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 as 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.



Set of Objects

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 different ways :

in logical representation through the use of universal quantifier, and

in hierarchical structure where node represent sets, the inheritance propagate set level assertion down to individual.

Example: assert large (elephant); Remember to make clear distinction between,

❖ whether we are asserting some property of the set itself, means, the set of elephants is large, or

❖ asserting some property that holds for individual elements of the set , means, any thing that is an elephant is large.

There are three ways in which sets may be represented :

❖ Name, as in the example. Inheritable KR, the node - Baseball- Player and the predicates as Ball and Batter in logical representation.

❖ Extensional definition is to list the numbers, and

❖ Intensional definition is to provide a rule, that returns true or false depending on whether the object is in the set or not.

❖ **Finding Right Structure**

Access to right structure for describing a particular situation.

It 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 them.

Knowledge Representation using predicate logic

Representing Simple Facts in Logic

AI system might need to represent knowledge. Propositional logic is one of the fairly good forms of representing the same because it is simple to deal with and a decision procedure for it exists. Real-world facts are represented as logical propositions and are written as well-formed formulas (wff's) in propositional logic, as shown in Figure below. Using these propositions, we may easily conclude it is not sunny from the fact that its raining. But contrary to the ease of using the propositional logic there are its limitations. This is well demonstrated using a few simple sentence like:

It is raining.
RAINING

It is sunny.
SUNNY

It is windy.
WINDY

If it is raining, then it is not sunny.
RAINING $\rightarrow \neg$ SUNNY

Some simple facts in Propositional logic

Socrates is a man.

We could write:

SOCRATESMAN

But if we also wanted to represent

Plato is a man.

we would have to write something such as:

PLATOMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between Socrates and Plato. It would be much better to represent these facts as:

MAN(SOCRATES)

MAN(PLATO)

since now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use predicates applied to arguments. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal.

We could represent this as:

MORTALMAN

But that fails to capture the relationship between any individual being a man and that individual being a mortal. To do that, we really need variables and quantification unless we are willing to write separate statements about the mortality of every known man.

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences:

1 . Marcus was a man.

- Marcus was a Pompeian.
- All Pompeians were Romans.
- Caesar was a ruler.
- All Romans were either loyal to Caesar or hated him.
- Everyone is loyal to someone.
- People only try to assassinate rulers they are not loyal to.
- Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

Marcus was a man.

$\text{man}(\text{Marcus})$

Although this representation fails to represent the notion of past tense (which is clear in the English sentence), it captures the critical fact of Marcus being a man. Whether this omission is acceptable or not depends on the use to which we intend to put the knowledge.

Marcus was a Pompeian.

$\text{Pompeian}(\text{Marcus})$

3. All Pompeians were Romans.

$\forall x : \text{Pompeian}(x) \rightarrow \text{Roman}(x)$

4. Caesar was a ruler.

ruler(Caesar)

Since many people share the same name, the fact that proper names are often not references to unique individuals, overlooked here. Occasionally deciding which of several people of the same name is being referred to in a particular statement may require a somewhat more amount of knowledge and logic.

5. All Romans were either loyal to Caesar or hated him.

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

Here we have used the inclusive-or interpretation of the two types of Or supported by English language. Some people will argue, however, that this English sentence is really stating an exclusive-or. To express that, we would have to write:

\rightarrow
 $\forall x: \text{Roman}(x) \rightarrow [(\text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})) \wedge$
 Not $(\text{loyalto}(x, \text{Caesar}) \wedge \text{hate}(x, \text{Caesar}))]$

6. Everyone is loyal to someone.

$\forall x: \exists y : \text{loyalto}(x, y)$

The scope of quantifiers is a major problem that arises when trying to convert English sentences into logical statements. Does this sentence say, as we have assumed in writing the logical formula above, that for each person there exists someone to whom he or she is loyal, possibly a different someone for everyone? Or does it say that there is someone to whom everyone is loyal?

$\neg \text{loyalto}(\text{Marcus}, \text{Caesar})$
 \uparrow (7, substitution)
 $\text{person}(\text{Marcus}) \wedge$
 $\text{ruler}(\text{Caesar}) \wedge$
 $\text{tryassassinate}(\text{Marcus}, \text{Caesar})$
 \uparrow (4)
 $\text{person}(\text{Marcus})$
 $\text{tryassassinate}(\text{Marcus}, \text{Caesar})$
 \uparrow (8)
 $\text{person}(\text{Marcus})$

An Attempt to Prove not loyal to(Marcus,Caesar)

7. People only try to assassinate rulers they are not loyal to.

$\forall x : \forall y : \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassasinate}(x, y) \rightarrow \neg \text{loyalto}(x, y)$

8. Like the previous one this sentence too is ambiguous which may lead to more than one conclusion. The usage of “try to assassinate” as a single predicate gives us a fairly simple

representation with which we can reason about trying to assassinate. But there might be connections as try to assassinate and not actually assassinate could not be made easily.

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

• Marcus tried to assassinate Caesar.
 tryassassinate (Marcus, Caesar)

now, say suppose we wish to answer the following question:

Was Marcus loyal to Caesar?

What we do is start reasoning backward from the desired goal which is represented in predicate logic as:

$\neg loyalto(Marcus, Caesar)$

Figure 4.2 shows an attempt to produce a proof of the goal by reducing the set of necessary but as yet unattained goals to the empty sets. The attempts fail as we do not have any statement to prove person(Marcus). But the problem is solved just by adding an additional statement i.e.

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. $\forall x: \forall y: \forall z: instance(x, y) \wedge isa(y, z) \rightarrow instance(x, z)$

10. All men are people.

$\forall x: man(x) \rightarrow person(x)$

Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.

Representing Instance and isa relationships

Knowledge can be represented as classes, objects, attributes and Super class and sub class relationships.

Knowledge can be inference using property inheritance. In this elements of specific classes inherit the attributes and values.

Attribute instance is used to represent the relationship “Class membership ” (element of the class)

Attribute isa is used to represent the relationship “Class inclusion” (super class, sub class relationship)

Three ways of representing class membership

These examples illustrate two points. The first is fairly specific. It is that, although class and superclass memberships are important facts that need to be represented those memberships need not be represented with predicates labelled instance and isa. In fact, in a logical framework it is usually unwieldy to do that, and instead unary predicates corresponding to the classes are often used. The second point is more general. There are usually several different ways of representing a given fact within a particular representational framework, be it logic or anything else. The choice depends partly on which deductions need to be supported most efficiently and partly on taste. The only important thing is that within a particular knowledge base consistency of representation is critical. Since any particular inference rule is designed to work on one particular form of representation, it is necessary that all the knowledge to which that rule is intended to apply be in the form that the rule demands. Many errors in the reasoning performed by knowledge-based programs are the result of inconsistent representation decisions. The moral is simply to be careful.

Computable functions and predicates

Some of the computational predicates like Less than, Greater than used in knowledge representation. It generally return true or false for the inputs.

Examples: Computable predicates

gt(1,0) or lt(0,1)

gt(5,4) or gt(4,5)

Computable functions: gt(2+4, 5)

Resolution

- ❖ Resolution is a proof procedure by refutation.
- ❖ To prove a statement using resolution it attempt to show that the negation of that statement.

2.5.1 Resolution in Propositional Logic

In propositional logic, the procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following.

ALGORITHM: PROPOSITIONAL RESOLUTION

Convert all the propositions of F to clause form

Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.

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 $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.

- 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.

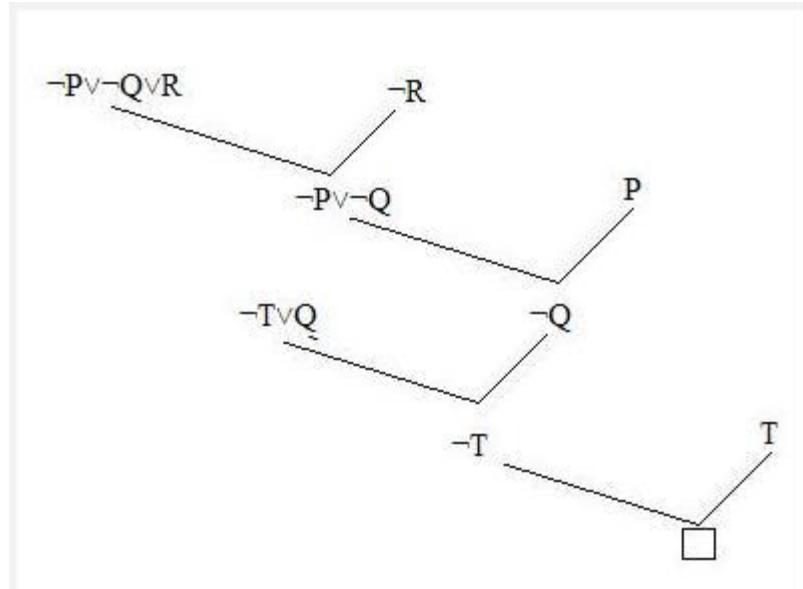


Figure : Resolution In Propositional Logic

2.5.3 UNIFICATION ALGORITHM

In propositional logic it is easy to determine that two literals can not both be true at the same time. Simply look for L and $\sim L$. In predicate logic, this matching process is more complicated, since bindings of variables must be considered.

For example $\text{man}(\text{john})$ and $\text{man}(\text{john})$ is a contradiction while $\text{man}(\text{john})$ and $\text{man}(\text{Himalayas})$ is not. Thus in order to determine contradictions we need a matching procedure that compares two literals and discovers whether there exist a set of substitutions that makes them identical. There is a recursive procedure that does this matching. It is called Unification algorithm.

In Unification algorithm each literal is represented as a list, where first element is the name of a predicate and the remaining elements are arguments. The argument may be a single element (atom) or may be another list. For example we can have literals as

(tryassassinate Marcus Caesar)

(tryassassinate Marcus (ruler of Rome))

To unify two literals, first check if their first elements are same. If so proceed. Otherwise they can not be unified. For example the literals

(tryassassinate Marcus Caesar)

(hate Marcus Caesar)

Can not be Unified. The unification algorithm recursively matches pairs of elements, one pair at a time. The matching rules are :

28. Different constants , functions or predicates can not match, whereas identical ones can.
29. A variable can match another variable , any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).
30. The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent. (a substitution y for x written as y/x)

The Unification algorithm is listed below as a procedure UNIFY (L1, L2). It returns a list representing the composition of the substitutions that were performed during the match. An empty list NIL indicates that a match was found without any substitutions. If the list contains a single value F, it indicates that the unification procedure failed.

The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(L1, L2)

I. If L1 or L2 are both variables or constants, then:

If L1 and L2 are identical, then return NIL.

Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).

Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL} , else return (L1/L2).

Else return {FAIL}.

9. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.

10. If L1 and L2 have a different number of arguments, then return {FAIL}.

11. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2.)

12. For $i \leftarrow 1$ to number of arguments in L1 :

Call Unify with the ith argument of L1 and the ith argument of L2, putting result in S.

If S contains FAIL then return {FAIL}.

If S is not equal to NIL then:

Apply S to the remainder of both L1 and L2.

(ii) SUBST: = APPEND(S, SUBST).

6. Return SUBST.

Resolution in Predicate Logic

ALGORITHM: RESOLUTION IN PREDICATE LOGIC

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 step 1.
3. Repeat until either a contradiction is found or 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 resulting clause, called the resolvent, will be the disjunction of all of the literals of both the 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 T1 and the other contains T2 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 one pair of complementary literals, only one such pair should be omitted 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.

If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably as given below.

Procedural v/s Declarative Knowledge

- A Declarative representation is one in which knowledge is specified but the use to which that knowledge is to be put in, is not given.
- A Procedural representation is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.
- To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.
- The difference between the declarative and the procedural views of knowledge lies in where control information resides.

man(Marcus)

man(Caesar)

person(Cleopatra)

$\forall x: \text{man}(x) \rightarrow \text{person}(x)$

person(x)?

Now we want to extract from this knowledge base the ans to the question:

$\exists y$: person (y)

Marcus, Ceaser and Cleopatra can be the answers

- As there is more than one value that satisfies the predicate, but only one value is needed, the answer depends on the order in which the assertions are examined during the search of a response.
- If we view the assertions as declarative, then we cannot depict how they will be examined. If we view them as procedural, then they do.
- Let us view these assertions as a non deterministic program whose output is simply not defined, now this means that there is no difference between Procedural & Declarative Statements. But most of the machines don't do so, they hold on to what ever method they have, either sequential or in parallel.
- The focus is on working on the control model.
man(Marcus)
man (Ceaser)
 $\forall x$: man(x)
person(x)

Person(Cleopatra)

- If we view this as declarative then there is no difference with the previous statement. But viewed procedurally, and using the control model, we used to get Cleopatra as the answer, now the answer is marcus.
- The answer can vary by changing the way the interpreter works.
- The distinction between the two forms is often very fuzzy. Rather than trying to prove which technique is better, what we should do is to figure out what the ways in which rule formalisms and interpreters can be combined to solve problems.

Logic Programming

- Logic programming is a programming language paradigm in which logical assertions are viewed as programs, e.g : PROLOG
 - A PROLOG program is described as a series of logical assertions, each of which is a Horn Clause.
 - A Horn Clause is a clause that has at most one positive literal.
-

- Eg $p, \neg p \vee q$ etc are also Horn Clauses.
- The fact that PROLOG programs are composed only of Horn Clauses and not of arbitrary logical expressions has two important consequences.
Because of uniform representation a simple & effective interpreter can be written.
- The logic of Horn Clause systems is decidable.
- Even PROLOG works on backward reasoning.
- The program is read top to bottom, left to right and search is performed depth-first with backtracking.
- There are some syntactic difference between the logic and the PROLOG representations as mentioned
- The key difference between the logic & PROLOG representation is that PROLOG interpreter has a fixed control strategy, so assertions in the PROLOG program define a particular search path to answer any question. Whereas Logical assertions define set of answers that they justify, there can be more than one answers, it can be forward or backward tracking.
- Control Strategy for PROLOG states that we begin with a problem statement, which is viewed as a goal to be proved.
- Look for the assertions that can prove the goal.
- To decide whether a fact or a rule can be applied to the current problem, invoke a standard unification procedure.
- Reason backward from that goal until a path is found that terminates with assertions in the program.
- Consider paths using a depth-first search strategy and use backtracking.
- Propagate to the answer by satisfying the conditions.

Frame Based System

A frame is a data structure with typical knowledge about a particular object or concept. Frames, first proposed by Marvin Minsky in the 1970s.

Example : Boarding pass frames

QANTAS BOARDING PASS		AIR NEW ZEALAND BOARDING PASS	
Carrier:	<i>QANTAS AIRWAYS</i>	Carrier:	<i>AIR NEW ZEALAND</i>
Name:	<i>MR N BLACK</i>	Name:	<i>MRS J WHITE</i>
Flight:	<i>QF 612</i>	Flight:	<i>NZ 0198</i>
Date:	<i>29DEC</i>	Date:	<i>23NOV</i>
Seat:	<i>23A</i>	Seat:	<i>27K</i>
From:	<i>HOBART</i>	From:	<i>MELBOURNE</i>

Each frame has its own name and a set of attributes associated with it. Name, weight, height and age are slots in the frame Person. Model, processor, memory and price are slots in the frame Computer. Each attribute or slot has a value attached to it.

Frames provide a natural way for the structured and concise representation of knowledge.

A frame provides a means of organising knowledge in slots to describe various attributes and characteristics of the object.

Frames are an application of object-oriented programming for expert systems.

Object-oriented programming is a programming method that uses objects as a basis for analysis, design and implementation.

In object-oriented programming, an object is defined as a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand. All objects have identity and are clearly distinguishable. Michael Black, Audi 5000 Turbo, IBM Aptiva S35 are examples of objects.

An object combines both data structure and its behaviour in a single entity. This is in sharp contrast to conventional programming, in which data structure and the program behaviour have concealed or vague connections.

When an object is created in an object-oriented programming language, we first assign a name to the object, then determine a set of attributes to describe the object's characteristics, and at last write procedures to specify the object's behaviour.

A knowledge engineer refers to an object as a frame (the term, which has become the AI jargon).

Frames as a knowledge representation technique

The concept of a frame is defined by a collection of slots. Each slot describes a particular attribute or operation of the frame.

Slots are used to store values. A slot may contain a default value or a pointer to another frame, a set of rules or procedure by which the slot value is obtained.

Typical information included in a slot

Frame name.

Relationship of the frame to the other frames. The frame IBM Aptiva S35 might be a member of the class Computer, which in turn might belong to the class Hardware.

Slot value. A slot value can be symbolic, numeric or Boolean. For example, the slot Name has symbolic values, and the slot Age numeric values. Slot values can be assigned when the frame is created or during a session with the expert system.

Default slot value. The default value is taken to be true when no evidence to the contrary has been found. For example, a car frame might have four wheels and a chair frame four legs as default values in the corresponding slots.

Range of the slot value. The range of the slot value determines whether a particular object complies with the stereotype requirements defined by the frame. For example, the cost of a computer might be specified between \$750 and \$1500.

Procedural information. A slot can have a procedure attached to it, which is executed if the slot value is changed or needed.

Most frame based expert systems use two types of methods:

WHEN CHANGED and WHEN NEEDED

A WHEN CHANGED method is executed immediately when the value of its attribute changes.

A WHEN NEEDED method is used to obtain the attribute value only when it is needed.

A WHEN NEEDED method is executed when information associated with a particular attribute is needed for solving the problem, but the attribute value is undetermined.

Most frame based expert systems allow us to use a set of rules to evaluate information contained in frames.

How does an inference engine work in a frame based system?

In a rule based system, the inference engine links the rules contained in the knowledge base with data given in the database.

When the goal is set up, the inference engine searches the knowledge base to find a rule that has the goal in its consequent.

If such a rule is found and its IF part matches data in the database, the rule is fired and the specified object, the goal, obtains its value. If no rules are found that can derive a value for the goal, the system queries the user to supply that value.

In a frame based system, the inference engine also searches for the goal. But

In a frame based system, rules play an auxiliary role. Frames represent here a major source of knowledge and both methods and demons are used to add actions to the frames.

Thus the goal in a frame based system can be established either in a method or in a demon.

Difference between methods and demons:

A demon has an IF-THEN structure. It is executed whenever an attribute in the demon's IF statement changes its value. In this sense, demons and methods are very similar and the two terms are often used as synonyms.

However, methods are more appropriate if we need to write complex procedures. Demons on the other hand, are usually limited to IF-THEN statements.

Inference

Two control strategies: forward chaining and backward chaining

Forward chaining:

Working from the facts to a conclusion. Sometimes called the datadriven approach. To chain forward, match data in working memory against 'conditions' of rules in the rule-base. When one of them fires, this is liable to produce more data. So the cycle continues

Backward chaining:

Working from the conclusion to the facts. Sometimes called the goal-driven approach.

To chain backward, match a goal in working memory against 'conclusions' of rules in the rule-base.

When one of them fires, this is liable to produce more goals. So the cycle continues.

The choice of strategy depends on the nature of the problem. Assume the problem is to get from facts to a goal (e.g. symptoms to a diagnosis).

Backward chaining is the best choice if:

The goal is given in the problem statement, or can sensibly be guessed at the beginning of the consultation; or:

The system has been built so that it sometimes asks for pieces of data (e.g. "please now do the gram test on the patient's blood, and tell me the result"), rather than expecting all the facts to be presented to it.

This is because (especially in the medical domain) the test may be expensive, or unpleasant, or dangerous for the human participant so one would want to avoid doing such a test unless there was a good reason for it.

Forward chaining is the best choice if:

All the facts are provided with the problem statement; or:

There are many possible goals, and a smaller number of patterns of data; or:

There isn't any sensible way to guess what the goal is at the beginning of the consultation.

Note also that a backwards-chaining system tends to produce a sequence of questions which seems focussed and logical to the user, a forward-chaining system tends to produce a sequence which seems random & unconnected.

If it is important that the system should seem to behave like a human expert, backward chaining is probably the best choice.

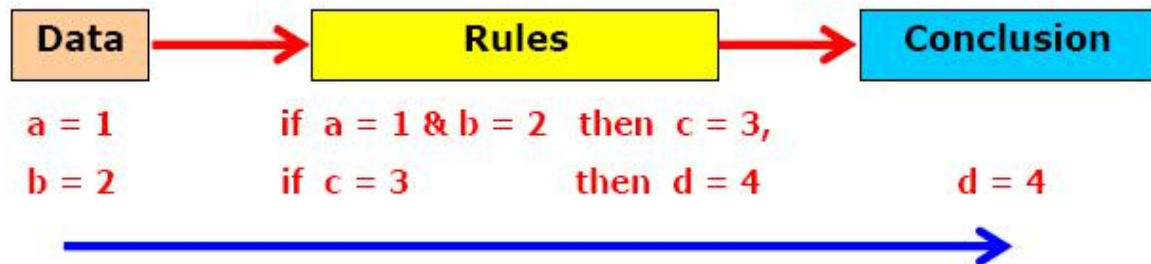
Forward Chaining Algorithm

Forward chaining is a techniques for drawing inferences from Rule base. Forward-chaining inference is often called data driven.

The algorithm proceeds from a given situation to a desired goal,adding new assertions (facts) found.

A forward-chaining, system compares data in the working memory against the conditions in the IF parts of the rules and determines which rule to fire.

Data Driven



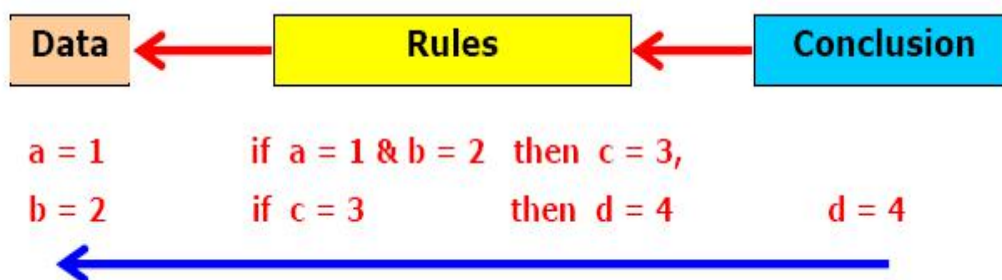
3.3.2 Backward Chaining Algorithm

Backward chaining is a techniques for drawing inferences from Rule base. Backward-chaining inference is often called goal driven.

The algorithm proceeds from desired goal, adding new assertions found.

A backward-chaining, system looks for the action in the THEN clause of the rules that matches the specified goal.

Goal Driven



Planning

Planning With State Space Search

The agent first generates a goal to achieve and then constructs a plan to achieve it from the Current state.

Problem Solving To Planning

Representation Using Problem Solving Approach

- ✓ Forward search
- ✓ Backward search
- ✓ Heuristic search

Representation Using Planning Approach

- ✓ STRIPS-standard research institute problem solver.
- ✓ Representation for states and goals
- ✓ Representation for plans
- ✓ Situation space and plan space
- ✓ Solutions

Why Planning?

Intelligent agents must operate in the world. They are not simply passive reasons (Knowledge Representation, reasoning under uncertainty) or problem solvers (Search), they must also act on the world.

We want intelligent agents to act in “intelligent ways”. Taking purposeful actions, predicting the expected effect of such actions, composing actions together to achieve complex goals. E.g. if we have a robot we want robot to decide what to do; how to act to achieve our goals.

Planning Problem

How to change the world to suit our needs

Critical issue: we need to reason about what the world will be like after doing a few actions, not just what it is like now

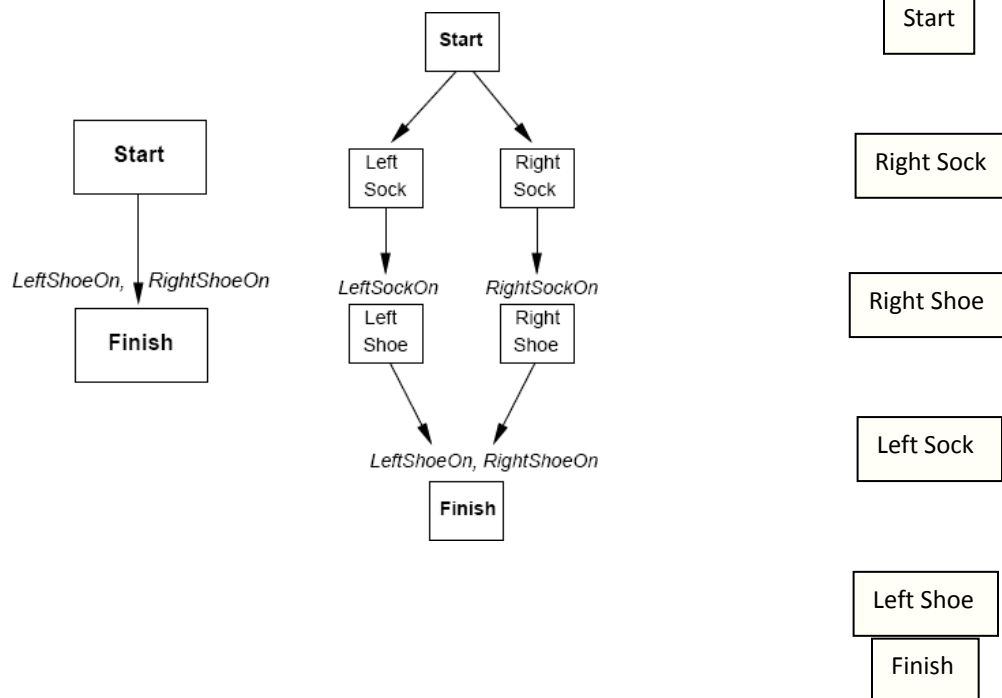
GOAL: Craig has coffee

CURRENTLY: robot in mailroom, has no coffee, coffee not made, Craig in office etc.

TO DO: goto lounge, make coffee

Partial Order Plan

- A partially ordered collection of steps
 - Start step has the initial state description and its effect
 - Finish step has the goal description as its precondition
 - Causal links from outcome of one step to precondition of another step
 - Temporal ordering between pairs of steps
- An open condition is a precondition of a step not yet causally linked
- A plan is **complete** if every precondition is achieved
- A precondition is **achieved** if it is the effect of an earlier step and no possibly intervening step undoes it




```

procedure CHOOSE-OPERATOR(plan, operators, Sneed, c)
  choose a step Sadd from operators or STEPS(plan) that has c as an effect
  if there is no such step then fail
  add the causal link  $S_{add} \xrightarrow{c} S_{need}$  to LINKS(plan)
  add the ordering constraint  $S_{add} \prec S_{need}$  to ORDERINGS(plan)
  if Sadd is a newly added step from operators then
    add Sadd to STEPS(plan)
    add  $Start \prec S_{add} \prec Finish$  to ORDERINGS(plan)



---


procedure RESOLVE-THREATS(plan)
  for each Sthreat that threatens a link  $S_i \xrightarrow{c} S_j$  in LINKS(plan) do
    choose either
      Demotion: Add  $S_{threat} \prec S_i$  to ORDERINGS(plan)
      Promotion: Add  $S_j \prec S_{threat}$  to ORDERINGS(plan)
    if not CONSISTENT(plan) then fail
  end

```

4.2 Stanford Research Institute Problem Solver (STRIPS)

STRIPS is a classical planning language, representing plan components as states, goals, and actions, allowing algorithms to parse the logical structure of the planning problem to provide a solution.

In STRIPS, state is represented as a conjunction of positive literals. Positive literals may be a propositional literal (e.g., Big ^ Tall) or a first-order literal (e.g., At(Billy, Desk)). The positive literals must be grounded – may not contain a variable (e.g., At(x, Desk)) – and must be function-free – may not invoke a function to calculate a value (e.g., At(Father(Billy), Desk)). Any state conditions that are not mentioned are assumed false.

The goal is also represented as a conjunction of positive, ground literals. A state satisfies a goal if the state contains all of the conjuncted literals in the goal; e.g., Stacked ^ Ordered ^ Purchased satisfies Ordered ^ Stacked.

Actions (or operators) are defined by action schemas, each consisting of three parts:

- The action name and any parameters.
- Preconditions which must hold before the action can be executed. Preconditions are represented as a conjunction of function-free, positive literals. Any variables in a precondition must appear in the action's parameter list.
- Effects which describe how the state of the environment changes when the action is executed. Effects are represented as a conjunction of function-free literals. Any

variables in a precondition must appear in the action's parameter list. Any world state not explicitly impacted by the action schema's effect is assumed to remain unchanged.

The following, simple action schema describes the action of moving a box from location x to location y :

Action: *MoveBox*(x, y)

Precond: *BoxAt*(x)

Effect: *BoxAt*(y), \neg *BoxAt*(x)

If an action is applied, but the current state of the system does not meet the necessary preconditions, then the action has no effect. But if an action is successfully applied, then any positive literals, in the effect, are added to the current state of the world; correspondingly, any negative literals, in the effect, result in the removal of the corresponding positive literals from the state of the world.

For example, in the action schema above, the effect would result in the proposition *BoxAt*(y) being added to the known state of the world, while *BoxAt*(x) would be *removed* from the known state of the world. (Recall that state only includes positive literals, so a negation effect results in the *removal* of positive literals.) Note also that positive effects can not get duplicated in state; likewise, a negative of a proposition that is not currently in state is simply ignored. For example, if *Open*(x) was not previously part of the state, \neg *Open*(x) would have no effect.

A STRIPS problem includes the complete (but relevant) initial state of the world, the goal state(s), and action schemas. A STRIPS algorithm should then be able to accept such a problem, returning a solution. The solution is simply an action sequence that, when applied to the initial state, results in a state which satisfies the goal.

4.2.1 STRIPS Planning Algorithm

As previously referenced, STRIPS began as an automated planning algorithm and has double-meaning to describe the language (described above) used to provide input to that algorithm. While the algorithm does not scale well to real-world problems, it, like the language, serves as a foundational starting point to developing and understanding more powerful automated planning algorithms. The STRIPS algorithm [3] is found below, followed by a brief commentary:

STRIPS(A, s, g)

p = empty plan

loop...

if s satisfies g then return p

a = [an applicable action in A , relevant for g]

if a = null, then return failure

$p' = \text{STRIPS}(A, s, \text{precond}(a))$

if $p' = \text{failure}$, then return failure

$s = \text{apply } p' \text{ to } s$

$s = \text{apply } a \text{ to } s$
 $p = p + p' + a$

In the above STRIPS algorithm, A represents all of the possible, grounded actions (i.e., action schemas with variables replaced with values), while s is the current state, and g is the goal state.

Critically important to note is that this algorithm is a *backward-search* algorithm. In other words, the goal state of the planning problem is fed into the algorithm as s , while the initial state of the planning problem is provided as g . The returned solution, if any, simply needs to be reversed to provide the solution to the planning problem. Simply put, it works backwards from the goal to figure out a logical path to get there.

4.3 Explanation

Most expert systems have explanation facilities that allow the user to ask questions- why and how it reached some conclusion

The questions are answered by referring to the system goals, the rules being used, and existing problem solving. The rules typically reflect empirical or “compiled” knowledge. They are codes of an expert’s rule of thumb, not the expert’s deeper understanding.

Example:

Dialog with an expert system designed to give advice on car problems.

System	Does the car start?
User	No.
System	Does the engine turn over?
User	yes
System	Do you smell gas?
User	yes
System	Recommendation-wait 5 minutes and try again.
User	Why?
System	I used the rule: If not start, and engine_turn_over, and smell_gas Then recommendation is „wait 5 minutes and try again“

User

Note: The rule gives the correct advice for a flooded car. And knows the questions to be ask to determine if the car is flooded, but it does not contain the knowledge of what a flooded car is and why waiting will help.

Types of Explanation

There are four types of explanations commonly used in expert systems.

- ✓ Rule trace reports on the progress of a consultation;
- ✓ Explanation of how the system reached to the give conclusion;
- ✓ Explanation of why the system did not give any conclusion.
- ✓ Explanation of why the system is asking a question;

UNIT 3

EXPERT SYSTEMS

An expert system is a computer program that represents and reasons with knowledge of some specialist subject with a view to solving problems or giving advice.

To solve expert-level problems, expert systems will need efficient access to a substantial domain knowledge base, and a reasoning mechanism to apply the knowledge to the problems they are given. Usually they will also need to be able to explain, to the users who rely on them, how they have reached their decisions.

They will generally build upon the ideas of knowledge representation, production rules, search, and so on, that we have already covered.

Often we use an expert system shell which is an existing knowledge independent framework into which domain knowledge can be inserted to produce a working expert system. We can thus avoid having to program each new system from scratch.

Typical Tasks for Expert Systems

There are no fundamental limits on what problem domains an expert system can be built to deal with. Some typical existing expert system tasks include:

1. The interpretation of data

Such as sonar data or geophysical measurements

2. Diagnosis of malfunctions

Such as equipment faults or human diseases

3. Structural analysis or configuration of complex objects

Such as chemical compounds or computer
systems

4. Planning sequences of actions

Such as might be performed by robots

5. Predicting the future

Such as weather, share prices, exchange rates

However, these days, “conventional” computer systems can also do some of these things

Characteristics of Expert Systems

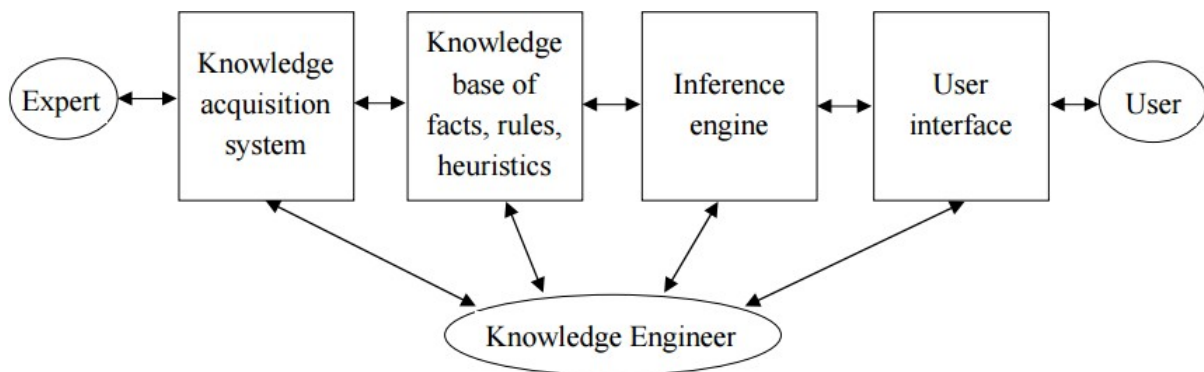
Expert systems can be distinguished from conventional computer systems in that:

1. They simulate human reasoning about the problem domain, rather than simulating the domain itself.
2. They perform reasoning over representations of human knowledge, in addition to doing numerical calculations or data retrieval. They have corresponding distinct modules referred to as the inference engine and the knowledge base.
3. Problems tend to be solved using heuristics (rules of thumb) or approximate methods or probabilistic methods which, unlike algorithmic solutions, are not guaranteed to result in a correct or optimal solution.
4. They usually have to provide explanations and justifications of their solutions or recommendations in order to convince the user that their reasoning is correct.

Note that the term Intelligent Knowledge Based System (IKBS) is sometimes used as a synonym for Expert System.

The Architecture of Expert Systems

The process of building expert systems is often called knowledge engineering. The knowledge engineer is involved with all components of an expert system:



Building expert systems is generally an iterative process. The components and their interaction will be refined over the course of numerous meetings of the knowledge engineer with the experts and users. We shall look in turn at the various components.

Knowledge Acquisition

The knowledge acquisition component allows the expert to enter their knowledge or expertise into the expert system, and to refine it later as and when required.

Historically, the knowledge engineer played a major role in this process, but automated systems that allow the expert to interact directly with the system are becoming increasingly common.

The knowledge acquisition process is usually comprised of three principal stages:

1. Knowledge elicitation is the interaction between the expert and the knowledge engineer/program to elicit the expert knowledge in some systematic way.
2. The knowledge thus obtained is usually stored in some form of human friendly intermediate representation.
3. The intermediate representation of the knowledge is then compiled into an executable form (e.g. production rules) that the inference engine can process.

In practice, much iteration through these three stages is usually required!

Knowledge Elicitation

The knowledge elicitation process itself usually consists of several stages:

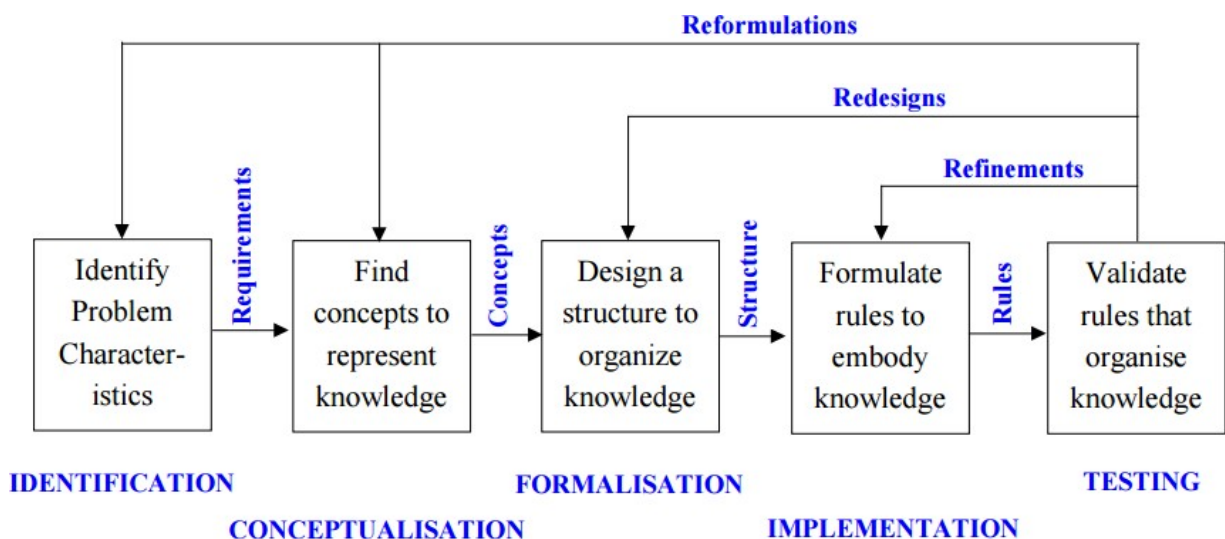
1. Find as much as possible about the problem and domain from books, manuals, etc. In particular, become familiar with any specialist terminology and jargon.
2. Try to characterize the types of reasoning and problem solving tasks that the system will be required to perform.
3. Find an expert (or set of experts) that is willing to collaborate on the project. Sometimes experts are frightened of being replaced by a computer system!

4. Interview the expert (usually many times during the course of building the system). Find out how they solve the problems your system will be expected to solve. Have them check and refine your intermediate knowledge representation.

This is a time intensive process, and automated knowledge elicitation and machine learning techniques are increasingly common modern alternatives.

5.3.1 Stages of Knowledge Acquisition

The iterative nature of the knowledge acquisition process can be represented in the following diagram.



Levels of Knowledge Analysis

Knowledge identification: Use in depth interviews in which the knowledge engineer encourages the expert to talk about how they do what they do. The knowledge engineer should understand the domain well enough to know which objects and facts need talking about.

Knowledge conceptualization: Find the primitive concepts and conceptual relations of the problem domain.

Epistemological analysis: Uncover the structural properties of the conceptual knowledge, such as taxonomic relations (classifications).

Logical analysis: Decide how to perform reasoning in the problem domain. This kind of knowledge can be particularly hard to acquire.

Implementation analysis: Work out systematic procedures for implementing and testing the system.

Capturing Tacit/Implicit Knowledge

One problem that knowledge engineers often encounter is that the human experts use tacit/implicit knowledge (e.g. procedural knowledge) that is difficult to capture.

There are several useful techniques for acquiring this knowledge:

1. **Protocol analysis:** Tape-record the expert thinking aloud while performing their role and later analyze this. Break down their protocol/account into the smallest atomic units of thought, and let these become operators.
2. **Participant observation:** The knowledge engineer acquires tacit knowledge through practical domain experience with the expert.
3. **Machine induction:** This is useful when the experts are able to supply examples of the results of their decision making, even if they are unable to articulate the underlying knowledge or reasoning process.

Which is/are best to use will generally depend on the problem domain and the expert.

Representing the Knowledge

We have already looked at various types of knowledge representation. In general, the knowledge acquired from our expert will be formulated in two ways:

1. **Intermediate representation** – a structured knowledge representation that the knowledge engineer and expert can both work with efficiently.
2. **Production system** – a formulation that the expert system's inference engine can process efficiently.

It is important to distinguish between:

1. **Domain knowledge** – the expert's knowledge which might be expressed in the form of rules, general/default values, and so on.
2. **Case knowledge** – specific facts/knowledge about particular cases, including any derived knowledge about the particular cases.

The system will have the domain knowledge built in, and will have to integrate this with the different case knowledge that will become available each time the system is used.

Meta Knowledge

Knowledge about knowledge

- Meta knowledge can be simply defined as knowledge about knowledge.
- Meta knowledge is knowledge about the use and control of domain knowledge in an expert system.

Roles in Expert System Development

Three fundamental roles in building expert systems are:

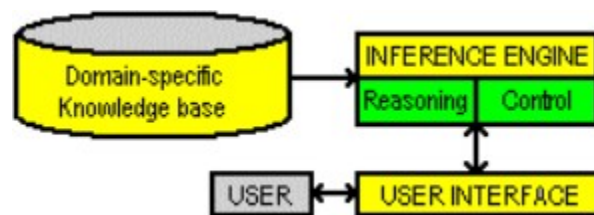
1. **Expert** - Successful ES systems depend on the experience and application of knowledge that the people can bring to it during its development. Large systems generally require multiple experts.
2. **Knowledge engineer** - The knowledge engineer has a dual task. This person should be able to elicit knowledge from the expert, gradually gaining an understanding of an area of expertise. Intelligence, tact, empathy, and proficiency in specific techniques of knowledge acquisition are all required of a knowledge engineer. Knowledge-acquisition techniques include conducting interviews with varying degrees of structure, protocol analysis, observation of experts at work, and analysis of cases.

On the other hand, the knowledge engineer must also select a tool appropriate for the project and use it to represent the knowledge with the application of the knowledge acquisition facility.

3. User - A system developed by an end user with a simple shell, is built rather quickly and inexpensively. Larger systems are built in an organized development effort. A prototype-oriented iterative development strategy is commonly used. ESs lend themselves particularly well to prototyping.

Typical Expert System

1. A problem-domain-specific knowledge base that stores the encoded knowledge to support one problem domain such as diagnosing why a car won't start. In a rule-based expert system, the knowledge base includes the if-then rules and additional specifications that control the course of the interview.



2. An inference engine a set of rules for making deductions from the data and that implements the reasoning mechanism and controls the interview process. The inference engine might be generalized so that the same software is able to process many different knowledge bases.

3. The user interface requests information from the user and outputs intermediate and final results. In some expert systems, input is acquired from additional sources such as data bases and sensors.

An expert system shell consists of a generalized inference engine and user interface designed to work with a knowledge base provided in a specified format. A shell often includes tools that help with the design, development and testing of the knowledge base. With the shell approach, expert systems representing many different problem domains may be developed and delivered with the same software environment. .

There are special high level languages used to program expert systems egg PROLOG

The user interacts with the system through a user interface which may use menus, natural language or any other style of interaction). Then an inference engine is used to reason with both the expert knowledge (extracted from our friendly expert) and data specific to the particular problem being solved. The expert knowledge will typically be in the form of a set of IF-THEN rules. The case specific data includes both data provided by the user and partial conclusions

(along with certainty measures) based on this data. In a simple forward chaining rule-based system the case specific data will be the elements in working memory.

How an expert system works Car engine diagnosis

1. IF

engine_getting_petrol

AND engine_turns_over

THEN problem_with_spark_plugs

2. IF NOT engine_turns_over

AND NOT lights_come_on

THEN problem_with_battery

3. IF NOT engine_turns_over

AND lights_come_on THEN

problem_with_starter

4. IF petrol_in_fuel_tank THEN

engine_getting_petrol

There are three possible problems with the car:

➤ problem_with_spark_plugs,

➤ problem_with_battery,

➤

problem_with_starteThe

system will ask the user:

- Is it true that there's petrol in the fuel tank?

Let's say that the answer is yes. This answer would be recorded, so that the user doesn't get asked the same question again. Anyway, the system now has proved that the engine is getting petrol, so now wants to find out if the engine turns over. As the system doesn't yet know whether this is the case, and as there are no rules which conclude this, the user will be asked:

- Is it true that the engine turns over?

Lets say this time the answer is no. There are no other rules which can be used to prove ``problem_with_spark_plugs" so the system will conclude that this is not the solution to the problem, and will consider the next hypothesis: problem_with_battery. It is true that the engine does not turn over (the user has just said that), so all it has to prove is that the lights don't come on. It will ask the user

- Is it true that the lights come on?

Suppose the answer is no. It has now proved that the problem is with the battery. Some systems might stop there, but usually there might be more than one solution, (e.g., more than one fault with the car), or it will be uncertain which of various solutions is the right one. So usually all hypotheses are considered. It will try to prove ``problem_with_starter", but given the existing data (the lights come on) the proof will fail, so the system will conclude that the problem is with the battery. A complete interaction with our very simple system might be:

- System: Is it true that there's petrol in the fuel tank?

User: Yes.

System: Is it true that the engine turns over?

User: No.

System Is it true that the lights come on?

User: No.

System: I conclude that there is a problem with battery.

Note that in general, solving problems using backward chaining involves searching through all the possible ways of proving the hypothesis, systematically checking each of them.

Questions

1. ``Briefly describe the basic architecture of a typical expert system, mentioning the function of each of the main components."
2. ``A travel agent asks you to design an expert system to help people choose where to go on holiday. Design a set of decisions to help you give advice on which holiday to take.

Expert System Use

Expert systems are used in a variety of areas, and are still the most popular developmental approach in the artificial intelligence world.

The table below depicts the percentage of expert systems being developed in particular areas:

Area	Percentage
Production/Operations Mgmt	48%
Finance	17%
Information Systems	12%
Marketing/Transactions	10%
Accounting/Auditing	5%
International Business	3%
Human Resources	2%
Others	2%

- Medical screening for cancer and brain tumours
- Matching people to jobs
- Training on oil rigs
- Diagnosing faults in car engines
- Legal advisory systems
- Mineral prospecting

MYCIN

Tasks and Domain

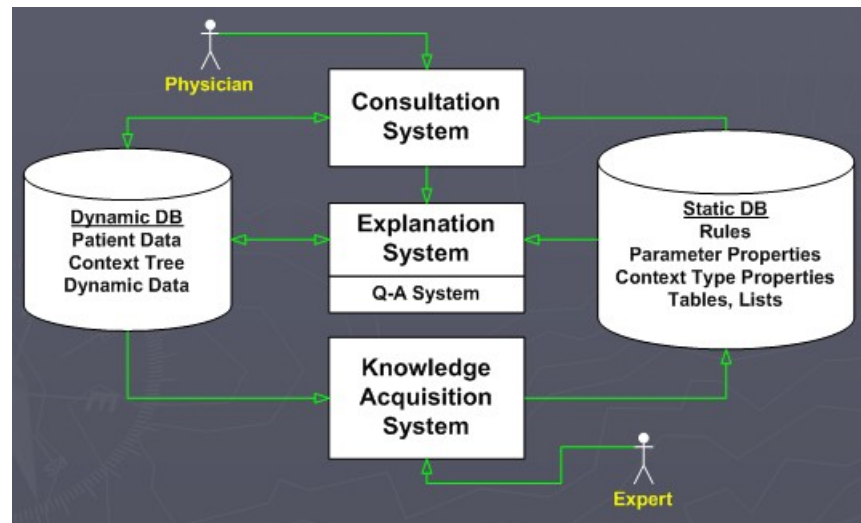
- Disease DIAGNOSIS and Therapy SELECTION
- Advice for non-expert physicians with time considerations and incomplete evidence on:

- Bacterial infections of the blood
- Expanded to meningitis and other ailments

System Goals

- ▶ Utility
 - Be useful, to attract assistance of experts
 - Demonstrate competence
 - Fulfill domain need (i.e. penicillin)
- ▶ Flexibility
 - Domain is complex, variety of knowledge types
 - Medical knowledge rapidly evolves, must be easy to maintain K.B.
- ▶ Interactive Dialogue
 - Provide coherent explanations (symbolic reasoning paradigm)
 - Allow for real-time K.B. updates by experts
- ▶ Fast and Easy
 - Meet time constraints of the medical field

Architecture



Consultation System

- ▶ Performs Diagnosis and Therapy Selection
- ▶ Control Structure reads Static DB (rules) and read/writes to Dynamic DB (patient, context)
- ▶ Linked to Explanations
- ▶ Terminal interface to Physician

DART

The Dynamic Analysis and Replanning Tool, commonly abbreviated to DART, is an artificial intelligence program used by the U.S. military to optimize and schedule the transportation of supplies or personnel and solve other logistical problems.

DART uses intelligent agents to aid decision support systems located at the U.S. Transportation and European Commands. It integrates a set of intelligent data processing agents and database management systems to give planners the ability to rapidly evaluate plans for logistical feasibility. By automating evaluation of these processes DART decreases the cost and time required to implement decisions.

DART achieved logistical solutions that surprised many military planners. Introduced in 1991, DART had by 1995 offset the monetary equivalent of all funds DARPA had channeled into AI research for the previous 30 years combined

Development and introduction

DARPA funded the MITRE Corporation and Carnegie Mellon University to analyze the feasibility of several intelligent planning systems. In November 1989, a demonstration named The Proud Eagle Exercise indicated many inadequacies and bottlenecks within military support systems. In July, DART was previewed to the military by BBN Systems and Technologies and the ISX Corporation (now part of Lockheed Martin Advanced Technology Laboratories) in conjunction with the United States Air Force Rome Laboratory. It was proposed in November 1990, with the military immediately demanding that a prototype be developed for testing. Eight weeks later, a hasty but working prototype was introduced in 1991 to the USTRANSCOM at the beginning of Operation Desert Storm during the Gulf War.

Impact

Directly following its launch, DART solved several logistical nightmares, saving the military millions of dollars. Military planners were aware of the tremendous obstacles facing moving military assets from bases in Europe to prepared bases in Saudi Arabia, in preparation for Desert Storm. DART quickly proved its value by improving upon existing plans of the U.S. military. What surprised many observers were DART's ability to adapt plans rapidly in a crisis environment.

DART's success led to the development of other military planning agents such as:

- RDA - Resource Description and Access system
- DRPI - Knowledge-Based Planning and Scheduling Initiative, a successor of DART

Expert Systems shells

Initially each expert system is build from scratch (LISP). Systems are constructed as a set of declarative representations (mostly rules) combined with an interpreter for those representations. It helps to separate the interpreter from domain-specific knowledge and to create a system that could be used construct new expert system by adding new knowledge corresponding to the new problem domain. The resulting interpreters are called shells. Example of shells is EMYCIN (for Empty MYCIN derived from MYCIN).

Shells – A shell is nothing but an expert system without knowledge base. A shell provides the developers with knowledge acquisition, inference engine, user interface, and explanation facility. For example, few shells are given below –

- Java Expert System Shell (JESS) that provides fully developed Java API for creating an expert system.
- Vidwan, a shell developed at the National Centre for Software Technology, Mumbai in 1993. It enables knowledge encoding in the form of IF-THEN rules.

Shells provide greater flexibility in representing knowledge and in reasoning than MYCIN. They support rules, frames, truth maintenance systems and a variety of other reasoning mechanisms.

Fuzzy Logic

Fuzzy Logic (FL) is a method of reasoning that resembles human reasoning. The approach of FL imitates the way of decision making in humans that involves all intermediate possibilities between digital values YES and NO.

The conventional logic block that a computer can understand takes precise input and produces a definite output as TRUE or FALSE, which is equivalent to human's YES or NO.

The inventor of fuzzy logic, Lotfi Zadeh, observed that unlike computers, the human decision making includes a range of possibilities between YES and NO, such as –

CERTAINLY YES
POSSIBLY YES
CANNOT SAY
POSSIBLY NO
CERTAINLY NO

The fuzzy logic works on the levels of possibilities of input to achieve the definite output.

Implementation

- It can be implemented in systems with various sizes and capabilities ranging from small micro-controllers to large, networked, workstation-based control systems.
- It can be implemented in hardware, software, or a combination of both.

Why Fuzzy Logic?

Fuzzy logic is useful for commercial and practical purposes.

- It can control machines and consumer products.
- It may not give accurate reasoning, but acceptable reasoning.
- Fuzzy logic helps to deal with the uncertainty in engineering.

Fuzzy Logic Systems Architecture

It has four main parts as shown –

Fuzzification Module – It transforms the system inputs, which are crisp numbers, into fuzzy sets. It splits the input signal into five steps such as –

LP x is Large Positive

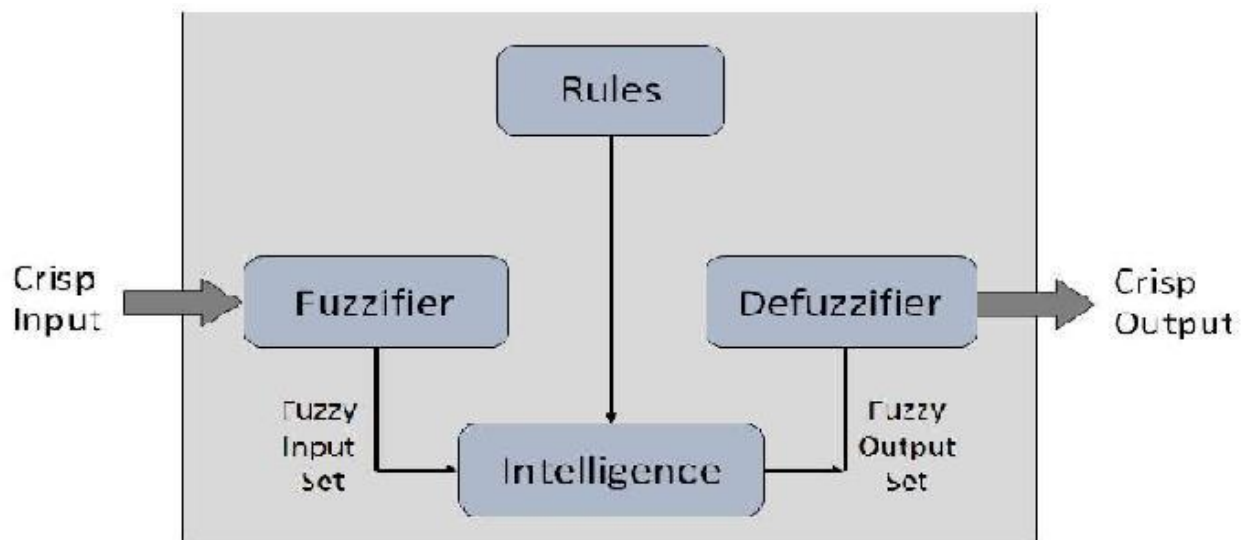
MP x is Medium Positive

S x is Small

MN x is Medium Negative

LN x is Large Negative

- Knowledge Base – It stores IF-THEN rules provided by experts.
- Inference Engine – It simulates the human reasoning process by making fuzzy inference on the inputs and IF-THEN rules.
- Defuzzification Module – It transforms the fuzzy set obtained by the inference engine into a crisp value.



The membership functions work on fuzzy sets of variables.

Membership Function

Membership functions allow you to quantify linguistic term and represent a fuzzy set graphically. A membership function for a fuzzy set A on the universe of discourse X is defined as $\mu_A: X \rightarrow [0,1]$.

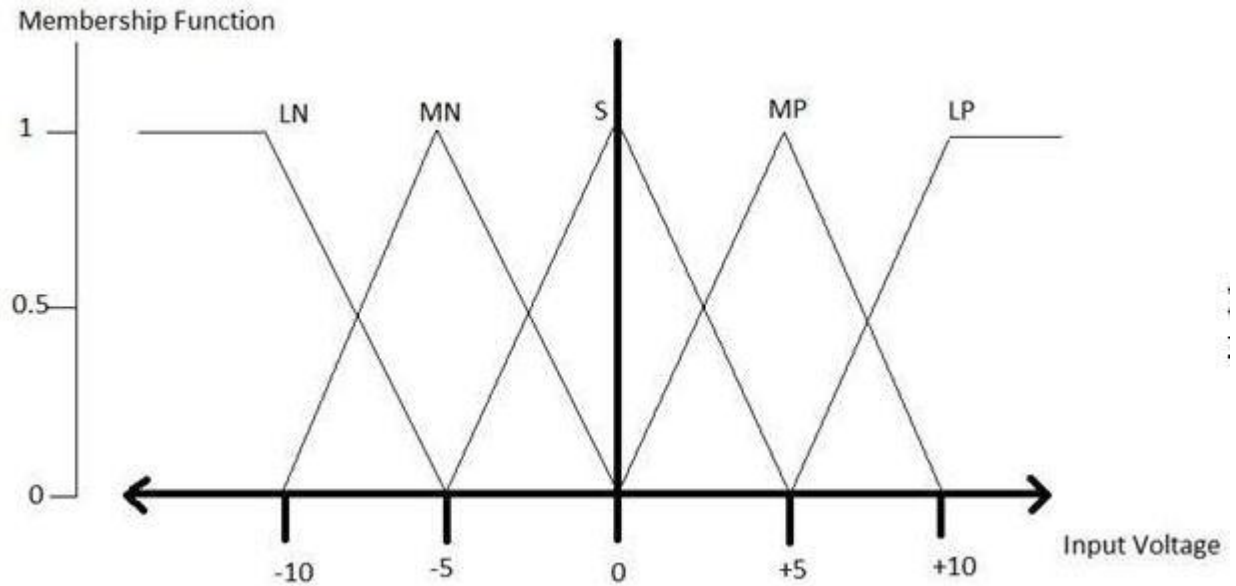
Here, each element of X is mapped to a value between 0 and 1. It is called membership value or degree of membership. It quantifies the degree of membership of the element in X to the fuzzy set A.

x axis represents the universe of discourse.

y axis represents the degrees of membership in the [0, 1] interval.

There can be multiple membership functions applicable to fuzzify a numerical value. Simple membership functions are used as use of complex functions does not add more precision in the output.

All membership functions for LP, MP, S, MN, and LN are shown as below –

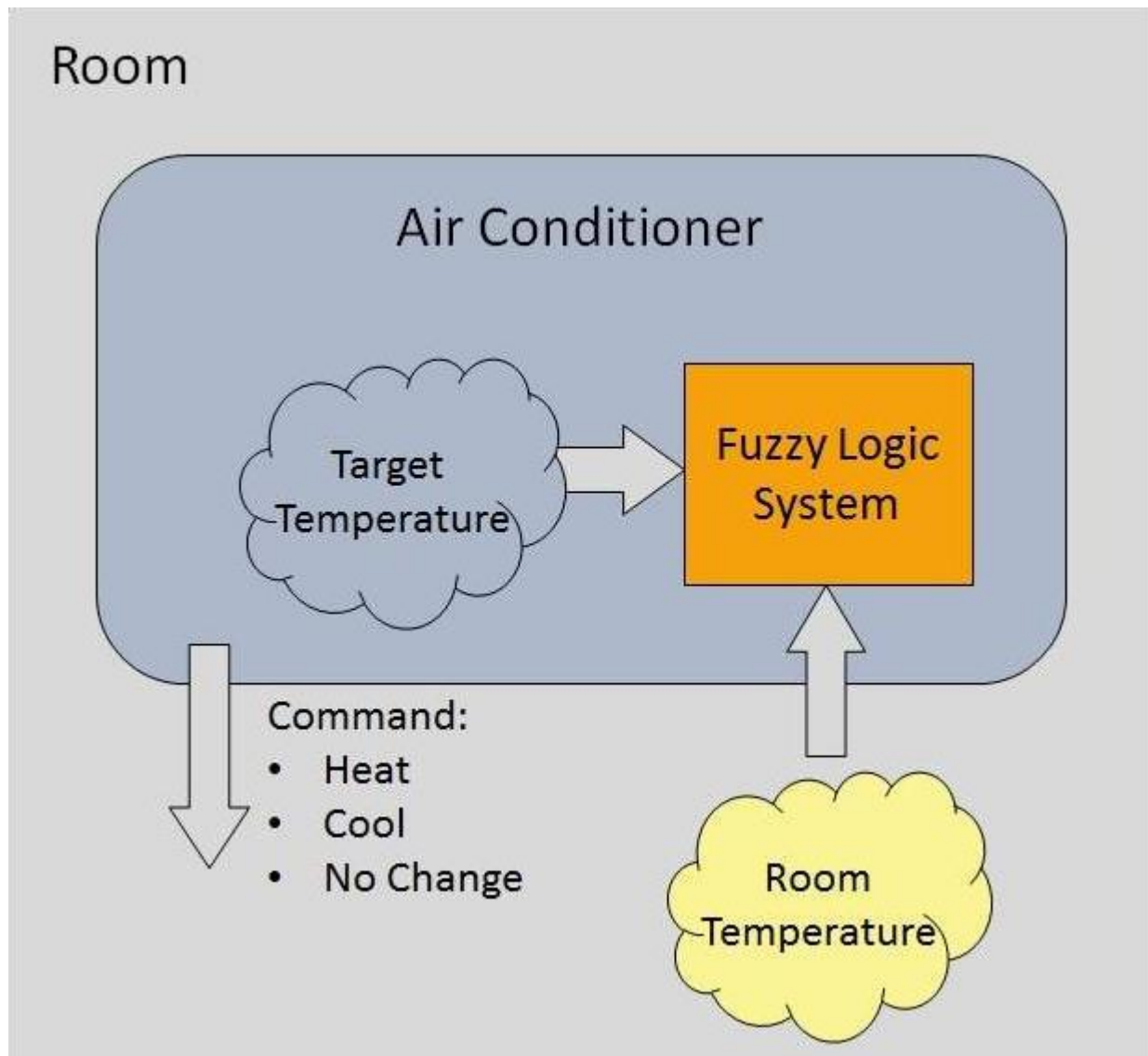


The triangular membership function shapes are most common among various other membership function shapes such as trapezoidal, singleton, and Gaussian.

Here, the input to 5-level fuzzifier varies from -10 volts to +10 volts. Hence the corresponding output also changes.

Example of a Fuzzy Logic System

Let us consider an air conditioning system with 5-level fuzzy logic system. This system adjusts the temperature of air conditioner by comparing the room temperature and the target temperature value.



Algorithm

-
- Define linguistic variables and terms.
 - Construct membership functions for them.
 - Construct knowledge base of rules.
 - Convert crisp data into fuzzy data sets using membership functions. (fuzzification)

Evaluate rules in the rule base. (interface engine)
Combine results from each rule. (interface engine)
Convert output data into non-fuzzy values. (defuzzification)

Logic Development

Step 1: Define linguistic variables and terms

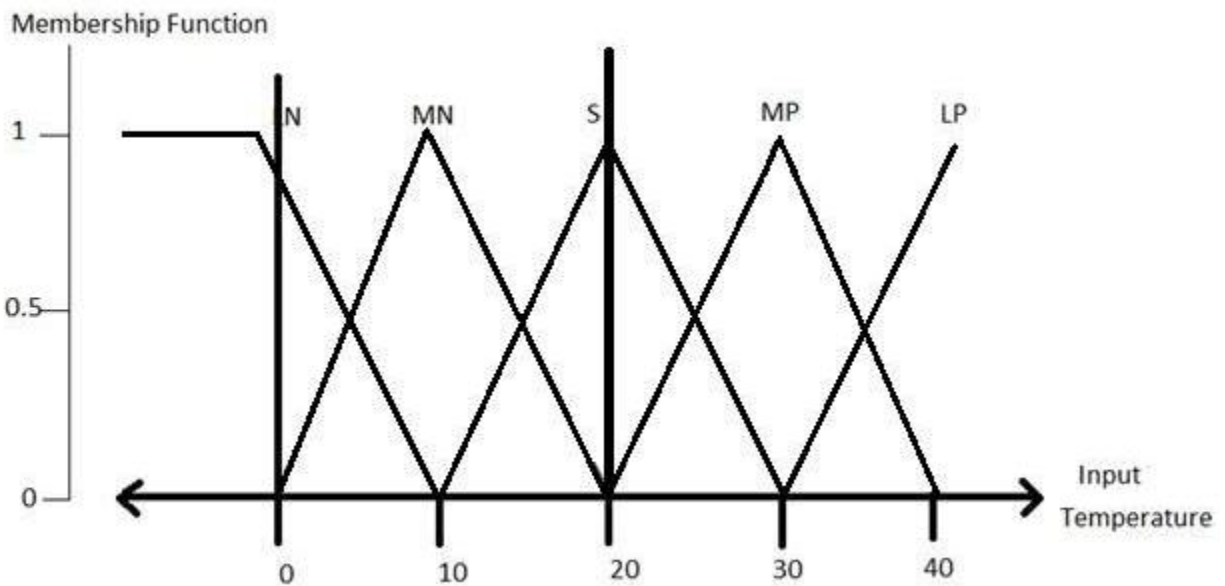
Linguistic variables are input and output variables in the form of simple words or sentences. For room temperature, cold, warm, hot, etc., are linguistic terms.

Temperature (t) = {very-cold, cold, warm, very-warm, hot}

Every member of this set is a linguistic term and it can cover some portion of overall temperature values.

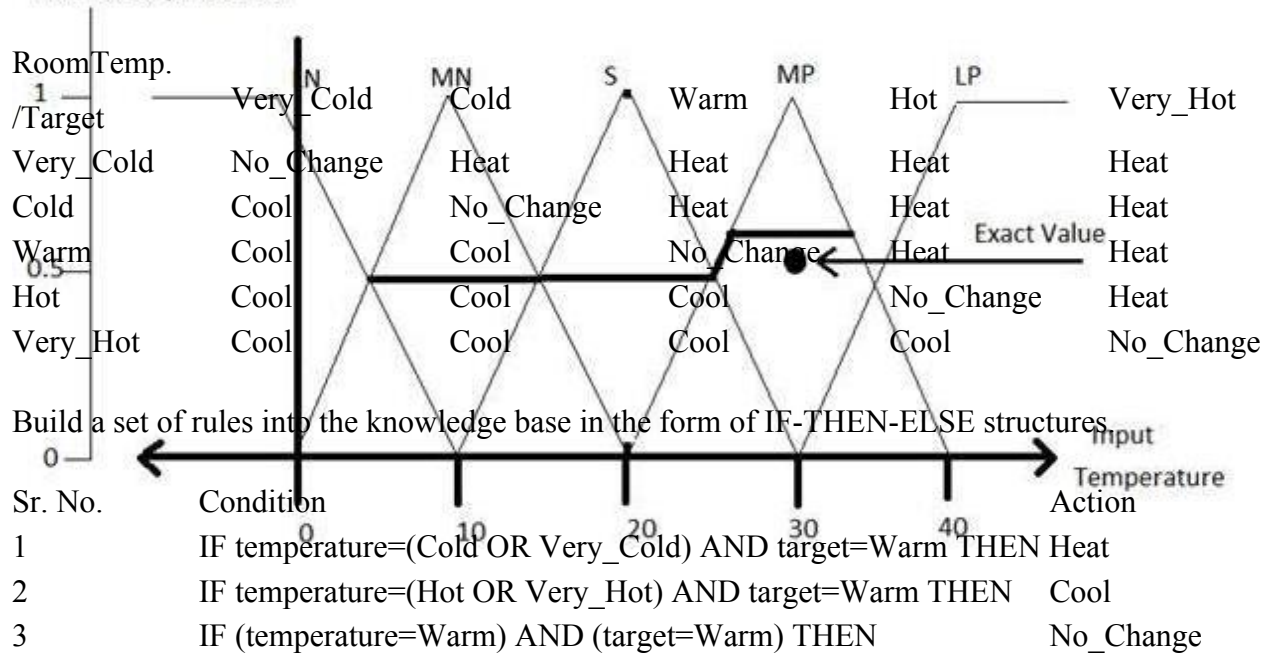
Step 2: Construct membership functions for them

The membership functions of temperature variable are as shown –



Step3: Construct knowledge base rules

Create a matrix of room temperature values versus target temperature values that an air conditioning system is expected to provide.



Step 4: Obtain fuzzy value

Fuzzy set operations perform evaluation of rules. The operations used for OR and AND are Max and Min respectively. Combine all results of evaluation to form a final result. This result is a fuzzy value.

Step 5: Perform defuzzification

Defuzzification is then performed according to membership function for output variable.

Application Areas of Fuzzy Logic

The key application areas of fuzzy logic are as given –

Automotive Systems

- Automatic Gearboxes
- Four-Wheel Steering
- Vehicle environment control

Consumer Electronic Goods

- Hi-Fi Systems
- Photocopiers
- Still and Video Cameras
- Television

Domestic Goods

- Microwave Ovens
- Refrigerators
- Toasters
- Vacuum Cleaners
- Washing Machines

Environment Control

- Air Conditioners/Dryers/Heaters
- Humidifiers

Advantages of FLSs

- Mathematical concepts within fuzzy reasoning are very simple.
- You can modify a FLS by just adding or deleting rules due to flexibility of fuzzy logic.
- Fuzzy logic Systems can take imprecise, distorted, noisy input information.
- FLSs are easy to construct and understand.
- Fuzzy logic is a solution to complex problems in all fields of life, including medicine, as it resembles human reasoning and decision making.

Disadvantages of FLSs

- There is no systematic approach to fuzzy system designing.
- They are understandable only when simple.
- They are suitable for the problems which do not need high accuracy.

Certainty Factor

A certainty factor (CF) is a numerical value that expresses a degree of subjective belief that a particular item is true. The item may be a fact or a rule. When probabilities are used attention must be paid to the underlying assumptions and probability distributions in order to show validity. Bayes' rule can be used to combine probability measures.

Suppose that a certainty is defined to be a real number between -1.0 and +1.0, where 1.0 represents complete certainty that an item is true and -1.0 represents complete certainty that an item is false. Here a CF of 0.0 indicates that no information is available about either the truth or the falsity of an item. Hence positive values indicate a degree of belief or evidence that an item is true, and negative values indicate the opposite belief. Moreover it is common to select a positive number that represents a minimum threshold of belief in the truth of an item. For example, 0.2 is a commonly chosen threshold value.

Form of certainty factors in ES

IF <evidence>
THEN <hypothesis> {cf }

cf represents belief in hypothesis H given that evidence E has occurred

It is based on 2 functions
< Measure of belief MB(H, E)
< Measure of disbelief MD(H, E)

Indicate the degree to which belief/disbelief of hypothesis H is increased if evidence E were observed

Total strength of belief and disbelief in a hypothesis:

$$cf = \frac{MB(H, E) - MD(H, E)}{1 - \min[MB(H, E), MD(H, E)]}$$

3.6 Bayesian networks

Represent dependencies among random variables

- Give a short specification of conditional probability distribution
- Many random variables are conditionally independent
- Simplifies computations
- Graphical representation

- DAG – causal relationships among random variables
- Allows inferences based on the network structure

Definition of Bayesian networks

A BN is a DAG in which each node is annotated with quantitative probability information, namely:

Nodes represent random variables (discrete or continuous)

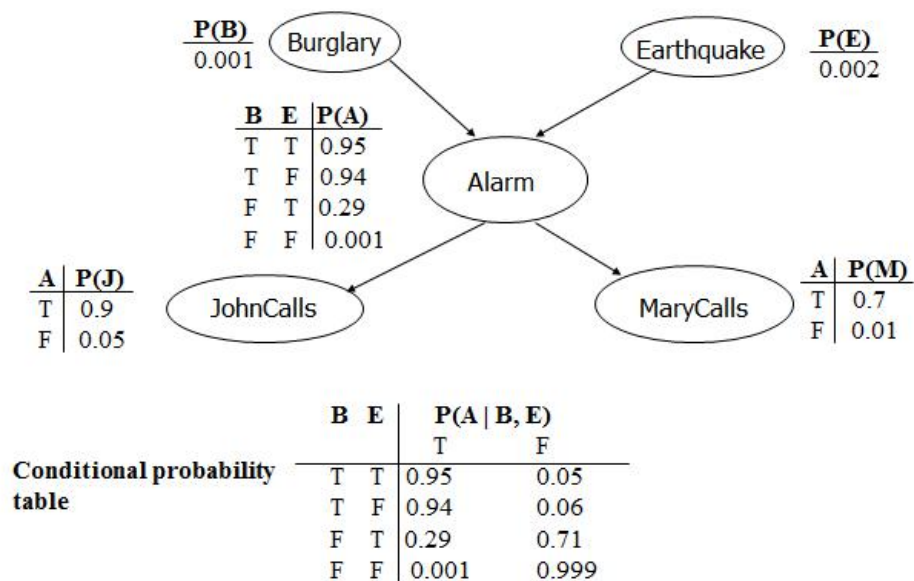
Directed links $X \rightarrow Y$: X has a direct influence on Y, X is said to be a parent of Y

each node X has an associated conditional probability table, $P(X_i | \text{Parents}(X_i))$ that quantify the effects of the parents on the node

Example: Weather, Cavity, Toothache, Catch

Weather, Cavity \rightarrow Toothache, Cavity \rightarrow Catch

Example



Bayesian network semantics

Represent a probability distribution

Specify conditional independence – build the network

A) each value of the probability distribution can be computed as:

$$P(X_1=x_1 \wedge \dots \wedge X_n=x_n) = P(x_1, \dots, x_n) = \prod_{i=1, n} P(x_i | \text{Parents}(x_i))$$

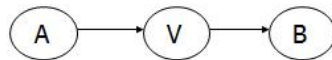
where $\text{Parents}(x_i)$ represent the specific values of $\text{Parents}(X_i)$

Building the network

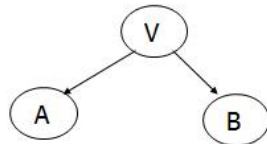
$$P(X_1=x_1 \wedge \dots \wedge X_n=x_n) = P(x_1, \dots, x_n) =$$

$$P(x_n | x_{n-1}, \dots, x_1) * P(x_{n-1}, \dots, x_1) = \dots =$$

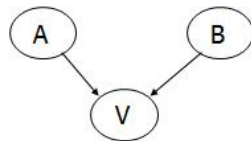
$$P(x_n | x_{n-1}, \dots, x_1) * P(x_{n-1} | x_{n-2}, \dots, x_1) * \dots * P(x_2 | x_1) * P(x_1) = \prod_{i=1, n} P(x_i | x_{i-1}, \dots, x_1)$$



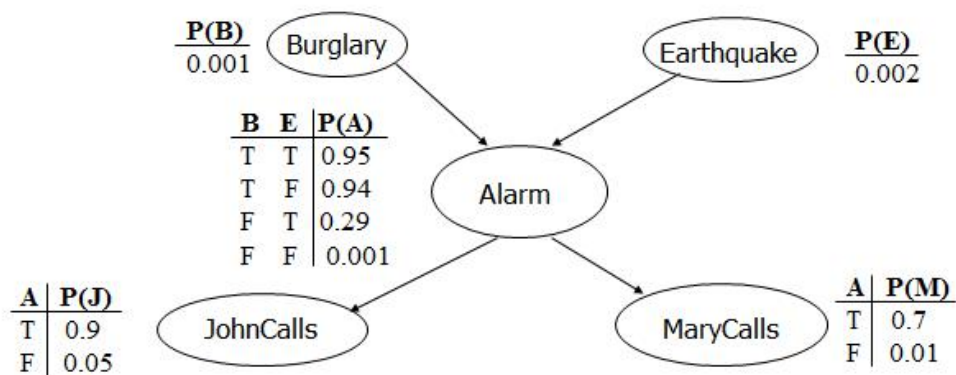
$$P(A \wedge V \wedge B) = P(A) * P(V|A) * P(B|V)$$



$$P(A \wedge V \wedge B) = P(V) * P(A|V) * P(B|V)$$



$$P(A \wedge V \wedge B) = P(A) * P(B) * P(V|A, B)$$



We can see that $P(X_i | X_{i-1}, \dots, X_1) = P(x_i | \text{Parents}(X_i))$ if $\text{Parents}(X_i) \subseteq \{X_{i-1}, \dots, X_1\}$

The condition may be satisfied by labeling the nodes in an order consistent with a DAG

Intuitively, the parents of a node X_i must be all the nodes X_{i-1}, \dots, X_1 which have a direct influence on X_i .

Pick a set of random variables that describe the problem

Pick an ordering of those variables

while there are still variables **repeat**

 choose a variable X_i and add a node associated to X_i

 assign $\text{Parents}(X_i) \leftarrow$ a minimal set of nodes that already exists in the network such that the conditional independence property is satisfied

 define the conditional probability table for X_i

Because each node is linked only to previous nodes \rightarrow DAG

$P(\text{MaryCalls} | \text{JohnCalls}, \text{Alarm}, \text{Burglary}, \text{Earthquake}) = P(\text{MaryCalls} | \text{Alarm})$

Compactness of node ordering

- ❖ Far more compact than a probability distribution
- ❖ Example of **locally structured system** (or **sparse**): each component interacts directly only with a limited number of other components
- ❖ Associated usually with a linear growth in complexity rather than with an exponential one
- ❖ *The order of adding the nodes is important*
- ❖ The correct order in which to add nodes is to add the “root causes” first, then the variables they influence, and so on, until we reach the leaves

Probabilistic Interfaces

$$P(J \wedge M \wedge A \wedge \sim B \wedge \sim E) =$$

$$P(J|A) * P(M|A) * P(A|\sim B \wedge \sim E) * P(\sim B) \wedge P(\sim E) = 0.9 * 0.7 * 0.001 * 0.999 * 0.998 = 0.00062$$

$$P(A|B) = P(A|B,E) * P(E|B) + P(A|B,\sim E) * P(\sim E|B) = P(A|B,E) * P(E) + P(A|B,\sim E) * P(\sim E)$$

$$= 0.95 * 0.002 + 0.94 * 0.998 = 0.94002$$

Different types of inferences

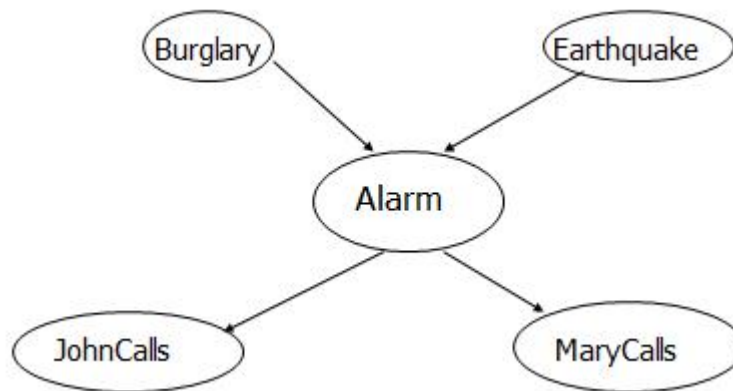
Diagnosis inferences (effect \rightarrow cause)

$$P(\text{Burglary} | \text{JohnCalls})$$

Causal inferences (cause \rightarrow effect)

$$P(\text{JohnCalls} | \text{Burglary}),$$

$$P(\text{MaryCalls} | \text{Burglary})$$



Intercausal inferences (between cause and common effects)

$$P(\text{Burglary} | \text{Alarm} \wedge \text{Earthquake})$$

Mixed inferences

$$P(\text{Alarm} | \text{JohnCalls} \wedge \sim \text{Earthquake}) \rightarrow \text{diag} + \text{causal}$$

$$P(\text{Burglary} | \text{JohnCalls} \wedge \sim \text{Earthquake}) \rightarrow \text{diag} + \text{intercausal}$$

3.7 Dempster-Shafer Theory



Dempster-Shafer theory is an approach to combining evidence

- ❖ Dempster (1967) developed means for combining degrees of belief derived from independent items of evidence.
- ❖ His student, Glenn Shafer (1976), developed method for obtaining degrees of belief for one question from subjective probabilities for a related question

- ❖ People working in Expert Systems in the 1980s saw their approach as ideally suitable for such systems.

- ❖ Each fact has a degree of support, between 0 and 1:

0 No support for the fact

1 full support for the fact

- ❖ Differs from Bayesian approach in that:

Belief in a fact and its negation need not sum to 1.

Both values can be 0 (meaning no evidence for or against the fact)

Set of possible conclusions: Θ

$$\Theta = \{ \theta_1, \theta_2, \dots, \theta_n \}$$

Where:

- ❖ Θ is the set of possible conclusions to be drawn
- ❖ Each θ_i is mutually exclusive: at most one has to be true.
- ❖ Θ is Exhaustive: At least one θ_i has to be true.

Frame of discernment

$$\Theta = \{ \theta_1, \theta_2, \dots, \theta_n \}$$

- ❖ Bayes was concerned with evidence that supported single conclusions (e.g., evidence for each outcome θ_i in Θ):
- ❖ $p(\theta_i | E)$
- ❖ D-S Theory is concerned with evidences which support
- ❖ subsets of outcomes in Θ , e.g., $\theta_1 \vee \theta_2 \vee \theta_3$ or $\{\theta_1, \theta_2, \theta_3\}$
- ❖ The “frame of discernment” (or “Power set”) of Θ is the set of all possible subsets of Θ :
E.g., if $\Theta = \{\theta_1, \theta_2, \theta_3\}$

Then the frame of discernment of Θ is:

(\emptyset , θ_1 , θ_2 , θ_3 , $\{\theta_1, \theta_2\}$, $\{\theta_1, \theta_3\}$, $\{\theta_2, \theta_3\}$, $\{\theta_1, \theta_2, \theta_3\}$)

\emptyset , the empty set, has a probability of 0, since one of the outcomes has to be true.

Each of the other elements in the power set has a probability between 0 and 1.

The probability of $\{\theta_1, \theta_2, \theta_3\}$ is 1.0 since one has to be true.

Mass function $m(A)$:

(where A is a member of the power set) = proportion of all evidence that supports this element of the power set.

“The mass $m(A)$ of a given member of the power set, A , expresses the proportion of all relevant and available evidence that supports the claim that the actual state belongs to A but to no particular subset of A .”

“The value of $m(A)$ pertains only to the set A and makes no additional claims about any subsets of A , each of which has, by definition, its own mass.
Each $m(A)$ is between 0 and 1.

All $m(A)$ sum to 1.

$m(\emptyset)$ is 0 - at least one must be true.

Interpretation of $m(\{A \vee B\}) = 0.3$

Means there is evidence for $\{A \vee B\}$ that cannot be divided among more specific beliefs for A or B .

Example

4 people (B, J, S and K) are locked in a room when the lights go out.

When the lights come on, K is dead, stabbed with a knife.

Not suicide (stabbed in the back)

No-one entered the room.

Assume only one killer.

- ❖ $\Theta = \{B, J, S\}$
- ❖ $P(\Theta) = (\emptyset, \{B\}, \{J\}, \{S\}, \{B, J\}, \{B, S\}, \{J, S\}, \{B, J, S\})$
- ❖ Detectives, after reviewing the crime-scene, assign mass probabilities to various elements of the power set.

Event	Mass
No-one is guilty	0
B is guilty	0.1
J is guilty	0.2
S is guilty	0.1
either B or J is guilty	0.1
either B or S is guilty	0.1
either S or J is guilty	0.3
One of the 3 is guilty	0.1

Belief in A:

The belief in an element A of the Power set is the sum of the masses of elements which are subsets of A (including A itself).

E.g., given $A = \{q_1, q_2, q_3\}$

$$\text{Bel}(A) = m(q_1) + m(q_2) + m(q_3) + m(\{q_1, q_2\}) + m(\{q_2, q_3\}) + m(\{q_1, q_3\}) + m(\{q_1, q_2, q_3\})$$

Example

- ❖ Given the mass assignments as assigned by the detectives:

A	{B}	{J}	{S}	{B,J}	{B,S}	{J,S}	{B,J,S}
m(A)	0.1	0.2	0.1	0.1	0.1	0.3	0.1

- ❖ $\text{bel}(\{B\}) = m(\{B\}) = 0.1$
- ❖ $\text{bel}(\{B, J\}) = m(\{B\}) + m(\{J\}) + m(\{B, J\}) = 0.1 + 0.2 + 0.1 = 0.4$
- ❖ Result:

A	{B}	{J}	{S}	{B,J}	{B,S}	{J,S}	{B,J,S}
m(A)	0.1	0.2	0.1	0.1	0.1	0.3	0.1
bel(A)	0.1	0.2	0.1	0.4	0.3	0.6	1.0

Plausibility of A: $pl(A)$

The plausibility of an element A, $pl(A)$, is the sum of all the masses of the sets that intersect with the set A:

$$\text{E.g. } pl(\{B,J\}) = m(B) + m(J) + m(B,J) + m(B,S) + m(J,S) + m(B,J,S) = 0.9$$

All Results:

A	{B}	{J}	{S}	{B,J}	{B,S}	{J,S}	{B,J,S}
m(A)	0.1	0.2	0.1	0.1	0.1	0.3	0.1
pl(A)	0.4	0.7	0.6	0.9	0.8	0.9	1.0

Disbelief (or Doubt) in A: $dis(A)$

The disbelief in A is simply $bel(\neg A)$.

It is calculated by summing all masses of elements which do not intersect with A.

The plausibility of A is thus $1 - dis(A)$:

$$pl(A) = 1 - dis(A)$$

A	{B}	{J}	{S}	{B,J}	{B,S}	{J,S}	{B,J,S}
m(A)	0.1	0.2	0.1	0.1	0.1	0.3	0.1
dis(A)	0.6	0.3	0.4	0.1	0.2	0.1	0
pl(A)	0.4	0.7	0.6	0.9	0.8	0.9	1.0

Belief Interval of A:

The certainty associated with a given subset A is defined by the belief interval:

$$[bel(A) \ pl(A)]$$

E.g. the belief interval of {B,S} is: [0.1 0.8]

A	{B}	{J}	{S}	{B,J}	{B,S}	{J,S}	{B,J,S}
m(A)	0.1	0.2	0.1	0.1	0.1	0.3	0.1
bel(A)	0.1	0.2	0.1	0.4	0.3	0.6	1.0
pl(A)	0.4	0.7	0.6	0.9	0.8	0.9	1.0

Belief Intervals & Probability

The probability in A falls somewhere between $\text{bel}(A)$ and $\text{pl}(A)$.

- ❖ $\text{bel}(A)$ represents the evidence we have for A directly So $\text{prob}(A)$ cannot be less than this value.
- ❖ $\text{pl}(A)$ represents the maximum share of the evidence we could possibly have, if, for all sets that intersect with A, the part that intersects is actually valid. So $\text{pl}(A)$ is the maximum possible value of $\text{prob}(A)$.

A	{B}	{J}	{S}	{B,J}	{B,S}	{J,S}	{B,J,S}
$m(A)$	0.1	0.2	0.1	0.1	0.1	0.3	0.1
$\text{bel}(A)$	0.1	0.2	0.1	0.4	0.3	0.6	1.0
$\text{pl}(A)$	0.4	0.7	0.6	0.9	0.8	0.9	1.0

Belief Intervals:

Belief intervals allow Dempster-Shafer theory to reason about the degree of certainty or certainty of our beliefs.

- ❖ A small difference between belief and plausibility shows that we are certain about our belief.
- ❖ A large difference shows that we are uncertain about our belief.

However, even with a 0 interval, this does not mean we know which conclusion is right. Just how probable it is!

A	{B}	{J}	{S}	{B,J}	{B,S}	{J,S}	{B,J,S}
$m(A)$	0.1	0.2	0.1	0.1	0.1	0.3	0.1
$\text{bel}(A)$	0.1	0.2	0.1	0.4	0.3	0.6	1.0
$\text{pl}(A)$	0.4	0.7	0.6	0.9	0.8	0.9	1.0

UNIT 4

4.4 Learning

Machine Learning

- Like human learning from past experiences, a computer does not have “experiences”.
- A computer system learns from data, which represent some “past experiences” of an application domain.
- Objective of machine learning : learn a target function that can be used to predict the values of a discrete class attribute, e.g., approve or not-approved, and high-risk or low risk.
- The task is commonly called: **Supervised learning, classification, or inductive learning**

Supervised Learning

Supervised learning is a machine learning technique for learning a function from training data. The training data consist of pairs of input objects (typically vectors), and desired outputs. The output of the function can be a continuous value (called regression), or can predict a class label of the input object (called classification). The task of the supervised learner is to predict the value of the function for any valid input object after having seen a number of training examples (i.e. pairs of input and target output). To achieve this, the learner has to generalize from the presented data to unseen situations in a "reasonable" way.

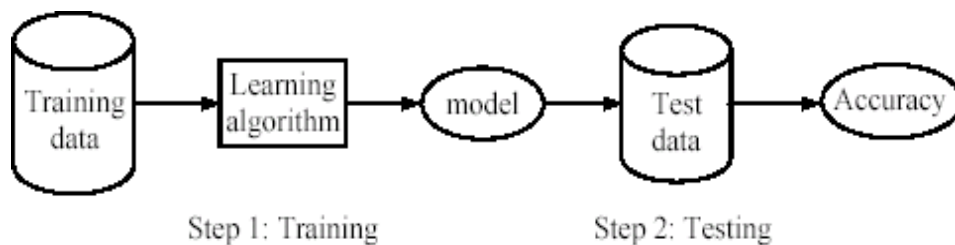
Another term for supervised learning is classification. Classifier performance depend greatly on

the characteristics of the data to be classified. There is no single classifier that works best on all given problems. Determining a suitable classifier for a given problem is however still more an art than a science. The most widely used classifiers are the Neural Network (Multi-layer Perceptron), Support Vector Machines, k-Nearest Neighbors, Gaussian Mixture Model, Gaussian, Naive Bayes, Decision Tree and RBF classifiers.

Supervised learning process: two steps

- **Learning** (training): Learn a model using the training data
- **Testing**: Test the model using unseen test data to assess the model accuracy

$$Accuracy = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}}$$



Supervised vs. unsupervised Learning

➤ **Supervised learning:**

classification is seen as supervised learning from examples.

- ✓ **Supervision:** The data (observations, measurements, etc.) are labeled with pre-defined classes. It is like that a “teacher” gives the classes (supervision).
- ✓ **Test data** are classified into these classes too.

➤ **Unsupervised learning** (clustering)

- ✓ Class labels of the data are unknown
- ✓ Given a set of data, the task is to establish the existence of classes or clusters in the data

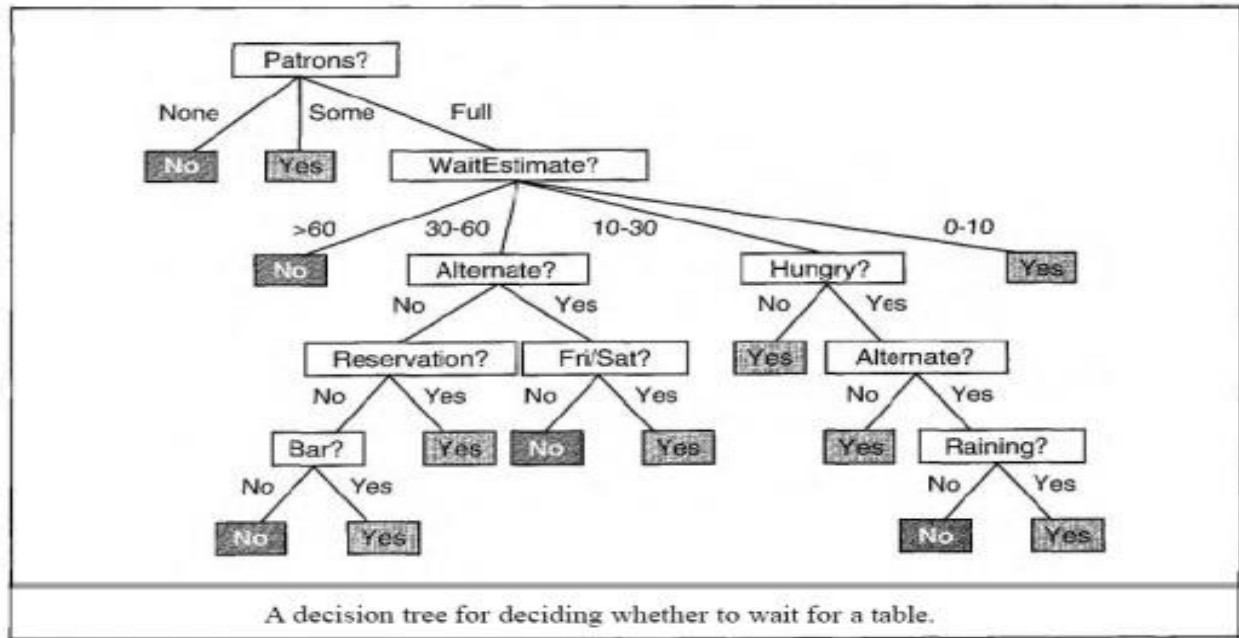
Decision Tree

- A decision tree takes as input an object or situation described by a set of attributes and returns a “decision” – the predicted output value for the input.
- A decision tree reaches its decision by performing a sequence of tests. Example : “HOW TO” manuals (for car repair)

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached. The decision tree representation seems to be very natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

A somewhat simpler example is provided by the problem of whether to wait for a table at a restaurant. The aim here is to learn a definition for the goal predicate Will Wait. In setting this up as a learning problem, we first have to state what attributes are available to describe examples in the domain. we will see how to automate this task; for now, let's suppose we decide on the following list of attributes:

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. Hungry: whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant's price range (\$, \$\$, \$\$\$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai, or burger).
10. Wait Estimate: the wait estimated by the host (0-10 minutes, 10-30, 30-60, >60).



Decision tree induction from examples

An example for a Boolean decision tree consists of a vector of input attributes, X , and a single Boolean output value y . A set of examples $(X_1, Y_1) \dots, (X_2, y_2)$ is shown in Figure. The positive examples are the ones in which the goal *Will Wait* is true (X_1, X_3, \dots); the negative examples are the ones in which it is false (X_2, X_5, \dots). The complete set of examples is called the **training set**.

Example	Attributes										Goal
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	WillWait
X_1	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0-10	Yes
X_2	Yes	No	No	Yes	Full	\$	No	No	Thai	30-60	No
X_3	No	Yes	No	No	Some	\$	No	No	Burger	0-10	Yes
X_4	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10-30	Yes
X_5	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	No
X_6	No	Yes	No	Yes	Some	\$\$	Yes	Yes	Italian	0-10	Yes
X_7	No	Yes	No	No	None	\$	Yes	No	Burger	0-10	No
X_8	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	0-10	Yes
X_9	No	Yes	Yes	No	Full	\$	Yes	No	Burger	>60	No
X_{10}	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10-30	No
X_{11}	No	No	No	No	None	\$	No	No	Thai	0-10	No
X_{12}	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30-60	Yes

Examples for the restaurant domain.

Decision Tree Algorithm

The basic idea behind the Decision-Tree-Learning-Algorithm is to test the most important attribute first. By "most important," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.

```
function DECISION-TREE-LEARNING(examples, attrs, default) returns a decision tree
  inputs: examples, set of examples
           attrs, set of attributes
           default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attrs is empty then return MAJORITY-VALUE(examples)
  else
    best  $\leftarrow$  CHOOSE-ATTRIBUTE(attrs, examples)
    tree  $\leftarrow$  a new decision tree with root test best
    m  $\leftarrow$  MAJORITY-VALUE(examples)
    for each value  $v_i$  of best do
      examplesi  $\leftarrow$  {elements of examples with best =  $v_i$ }
      subtree  $\leftarrow$  DECISION-TREE-LEARNING(examplesi, attrs - best, m)
      add a branch to tree with label  $v_i$  and subtree subtree
    return tree
```

The decision tree learning algorithm.

Reinforcement Learning

- Learning what to do to maximize reward
 - ✓ Learner is not given training
 - ✓ Only feedback is in terms of reward
 - ✓ Try things out and see what the reward is
- Different from Supervised Learning
 - ✓ Teacher gives training examples

Examples

- Robotics: Quadruped Gait Control, Ball Acquisition (Robocup)
- Control: Helicopters
- Operations Research: Pricing, Routing, Scheduling
- Game Playing: Backgammon, Solitaire, Chess, Checkers
- Human Computer Interaction: Spoken Dialogue Systems
- Economics/Finance: Trading

Markov decision process VS Reinforcement Learning

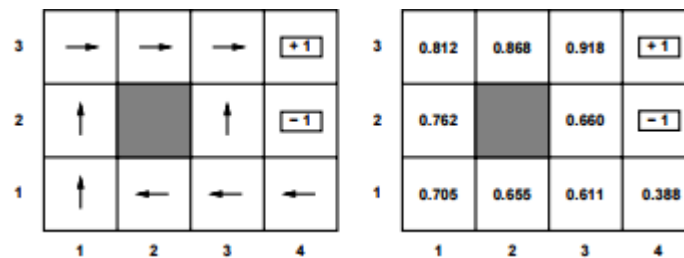
- Markov decision process
 - ✓ Set of state S , set of actions A
 - ✓ Transition probabilities to next states $T(s, a, a')$
 - ✓ Reward functions $R(s)$
- RL is based on MDPs, but
 - ✓ Transition model is not known
 - ✓ Reward model is not known
- MDP computes an optimal policy
- RL learns an optimal policy

Types of Reinforcement Learning

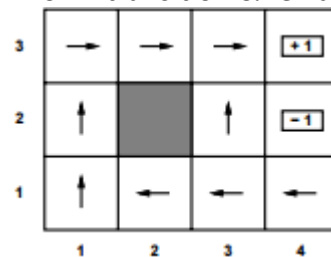
- Passive Vs Active
 - ✓ Passive: Agent executes a fixed policy and evaluates it
 - ✓ Active: Agents updates policy as it learns
- Model based Vs Model free
- Model-based: Learn transition and reward model, use it to get optimal policy

- Model free: Derive optimal policy without learning the model

Passive Learning



- Evaluate how good a policy π is
- Learn the utility $U^\pi(s)$ of each state
- Same as policy evaluation for known transition & reward models



Agent executes a sequence of trials:

$(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$
 $(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 2) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$
 $(1, 1) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (3, 2) \rightarrow (4, 2)_{-1}$

Goal is to learn the expected utility $U^\pi(s)$

$$U^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s \right]$$

Direct Utility Estimation

- Reduction to inductive learning
 - ✓ Compute the empirical value of each state
 - ✓ Each trial gives a sample value

- ✓ Estimate the utility based on the sample values
- Example: First trial gives
 - ✓ State (1,1): A sample of reward 0.72
 - ✓ State (1,2): Two samples of reward 0.76 and 0.84
 - ✓ State (1,3): Two samples of reward 0.80 and 0.88
- Estimate can be a running average of sample values
- Example: $U(1, 1) = 0.72, U(1, 2) = 0.80, U(1, 3) = 0.84, \dots$
- Ignores a very important source of information
- The utility of states satisfy the Bellman equations

$$U^\pi(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U^\pi(s')$$

- Search is in a hypothesis space for U much larger than needed
- Convergence is very slow
- Make use of Bellman equations to get $U^\pi(s)$
- Need to estimate $T(s, \pi(s), s')$ and $R(s)$ from trials
- Plug-in learnt transition and reward in the Bellman equations
- Solving for U^π : System of n linear equations
- Estimates of T and R keep changing
- Make use of modified policy iteration idea
 - ✓ Run few rounds of value iteration
 - ✓ Initialize value iteration from previous utilities
 - ✓ Converges fast since T and R changes are small
- ADP is a standard baseline to test „smarter“ ideas
- ADP is inefficient if state space is large

- ✓ Has to solve a linear system in the size of the state space
- ✓ Backgammon: 10^{50} linear equations in 10^{50} unknowns

Temporal Difference Learning

➤ Best of both worlds

- ✓ Only update states that are directly affected
- ✓ Approximately satisfy the Bellman equations
- ✓ Example:

$(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$

$(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 2) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$

$(1, 1) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (3, 2) \rightarrow (4, 2)_{-1}$

- After the first trial, $U(1, 3) = 0.84, U(2, 3) = 0.92$
- Consider the transition $(1, 3) \rightarrow (2, 3)$ in the second trial
- If deterministic, then $U(1, 3) = -0.04 + U(2, 3)$
- How to account for probabilistic transitions (without a model)

➤ TD chooses a middle ground

$$U^\pi(s) \leftarrow (1 - \alpha)U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s'))$$

➤ Temporal difference (TD) equation, α is the learning rate

➤ The TD equation

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s))$$

➤ TD applies a correction to approach the Bellman equations

- ✓ The update for s'' will occur $T(s, \pi(s), s'')$ fraction of the time
- ✓ The correction happens proportional to the probabilities
- ✓ Over trials, the correction is same as the expectation

➤ Learning rate α determines convergence to true utility

- ✓ Decrease α_s proportional to the number of state visits
- ✓ Convergence is guaranteed if

$$\sum_{m=1}^{\infty} \alpha_s(m) = \infty \quad \sum_{m=1}^{\infty} \alpha_s^2(m) < \infty$$

- ✓ Decay $\alpha_s(m) = 1/m$ satisfies the condition
 - TD is model free
- TD Vs ADP**
- TD is model free as opposed to ADP which is model based
 - TD updates observed successor rather than all successors
 - The difference disappears with large number of trials
 - TD is slower in convergence, but much simpler computation per observation

Active Learning

- Agent updates policy as it learns
- Goal is to learn the optimal policy
- Learning using the passive ADP agent
- ✓ Estimate the model $R(s), T(s, a, s')$ from observations
- ✓ The optimal utility and action satisfies

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U(s')$$

- ✓ Solve using value iteration or policy iteration
- Agent has “optimal” action
- Simply execute the “optimal” action

Exploitation vs Exploration

- The passive approach gives a greedy agent

- Exactly executes the recipe for solving MDPs
- Rarely converges to the optimal utility and policy
 - ✓ The learned model is different from the true environment
- Trade-off
 - ✓ Exploitation: Maximize rewards using current estimates
 - ✓ Agent stops learning and starts executing policy
 - ✓ Exploration: Maximize long term rewards
 - ✓ Agent keeps learning by trying out new things
- Pure Exploitation
 - ✓ Mostly gets stuck in bad policies
- Pure Exploration
 - ✓ Gets better models by learning
 - ✓ Small rewards due to exploration
- The multi-armed bandit setting
 - ✓ A slot machine has one lever, a one-armed bandit
 - ✓ n-armed bandit has n levers
- Which arm to pull?
 - ✓ Exploit: The one with the best pay-off so far
 - ✓ Explore: The one that has not been tried

Exploration

- Greedy in the limit of infinite exploration (GLIE)
 - ✓ Reasonable schemes for trade off
- Revisiting the greedy ADP approach
 - ✓ Agent must try each action infinitely often

- ✓ Rules out chance of missing a good action
- ✓ Eventually must become greedy to get rewards
- Simple GLIE
 - ✓ Choose random action $1/t$ fraction of the time
 - ✓ Use greedy policy otherwise
- Converges to the optimal policy
- Convergence is very slow

Exploration Function

- A smarter GLIE
 - ✓ Give higher weights to actions not tried very often
 - ✓ Give lower weights to low utility actions
- Alter Bellman equations using optimistic utilities $U^+(s)$

$$U^+(s) = R(s) + \gamma \max_a f \left(\sum_{s'} T(s, a, s') U^+(s'), N(a, s) \right)$$

- The exploration function $f(u, n)$
 - ✓ Should increase with expected utility u
 - ✓ Should decrease with number of tries n
- A simple exploration function

$$f(u, n) = \begin{cases} R^+, & \text{if } n < N \\ u, & \text{otherwise} \end{cases}$$

- Actions towards unexplored regions are encouraged
- Fast convergence to almost optimal policy in practice

Q-Learning

- Exploration function gives a active ADP agent
- A corresponding TD agent can be constructed

- ✓ Surprisingly, the TD update can remain the same
- ✓ Converges to the optimal policy as active ADP
- ✓ Slower than ADP in practice
- Q-learning learns an action-value function $Q(a; s)$
 - ✓ Utility values $U(s) = \max_a Q(a; s)$
- A model-free TD method
 - ✓ No model for learning or action selection
- Constraint equations for Q-values at equilibrium

$$Q(a, s) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(a', s')$$

- Can be updated using a model for $T(s; a; s')$
- The TD Q-learning does not require a model

$$Q(a, s) \leftarrow Q(a, s) + \alpha (R(s) + \gamma \max_{a'} Q(a', s') - Q(a, s))$$

- Calculated whenever a in s leads to s'
- The next action $a_{\text{next}} = \operatorname{argmax}_{a''} f(Q(a''; s''); N(s''; a''))$
- Q-learning is slower than ADP
- Trade-off: Model-free vs knowledge-based methods

UNIT 5

Natural Language Processing :

Developing programs to understand natural language is important in AI because a natural form of communication with systems is essential for user acceptance. One of the most critical tests for intelligent behavior is the ability to communicate effectively. This was the test proposed by Alan Turing. AI programs must be able to communicate with their human counterparts in a natural way, and natural language is one of the most important mediums for that purpose. A program understands a natural language if it behaves by taking a correct or acceptable action in response to the input. For example, we say a child demonstrates understanding if it responds with the correct answer to a question. The action taken need not be the external response. It may be the creation of some internal

data structures. The structures created should be meaningful and correctly interact with the world model representation held by the program. In this chapter we explore many of the important issues related to natural language understanding and language generation.

This chapter explores several techniques that are used to enable humans to interact with computers via natural human languages. Natural languages are the languages used by humans for communication (among other functions). They are distinctly different from formal languages, such as C++, Java, and PROLOG. One of the main differences, which we will examine in some detail in this chapter, is that natural languages are ambiguous, meaning that a given sentence can have more than one possible meaning, and in some cases the correct meaning can be very hard to determine. Formal languages are almost always designed to ensure that ambiguity cannot occur. Hence, a given program written in C++ can have only one interpretation. This is clearly desirable because otherwise the computer would have to make an arbitrary decision as to which interpretation to work with. It is becoming increasingly important for computers to be able to understand natural languages. Telephone systems are now widespread that are able to understand a narrow range of commands and questions to assist callers to large call centers, without needing to use human resources. Additionally, the quantity of unstructured textual data that exists in the world (and in particular, on the Internet) has reached unmanageable proportions. For humans to search through these data using traditional techniques such as Boolean queries or the database query language SQL is impractical. The idea that people should be able to

pose questions in their own language, or something similar to it, is an increasingly popular one. Of course, English is not the only natural language. A great deal of research in natural language processing and information retrieval is carried out in English, but many human languages differ enormously from English. Languages such as Chinese, Finnish, and Navajo have almost nothing in common with English (although of course Finnish uses the same alphabet). Hence, a system that can work with one human language cannot necessarily deal with any other human language. In this section we will explore two main topics. First, we will examine natural language processing, which is a collection of techniques used to enable computers to “understand” human language. In general, they are concerned with extracting grammatical information as well as meaning from human utterances but they are also concerned with understanding those utterances, and performing useful tasks as a result. Two of the earliest goals of natural language processing were automated translation (which is explored in this chapter) and database access. The idea here was that if a user wanted to find some information from a database, it would

make much more sense if he or she could query the database in her language, rather than needing to learn a new formal language such as SQL. Information retrieval is a collection of techniques used to try to match a query (or a command) to a set of documents from an existing corpus of documents. Systems such as the search engines that we use to find data on the Internet use information retrieval (albeit of a fairly simple nature).

Overview of linguistics

In dealing with natural language, a computer system needs to be able to process and manipulate language at a number of levels.

- Phonology. This is needed only if the computer is required to understand spoken language. Phonology is the study of the sounds that make up words and is used to identify words from sounds. We will explore this in a little more detail later, when we look at the ways in which computers can understand speech.
- Morphology. This is the first stage of analysis that is applied to words, once they have been identified from speech, or input into the system. Morphology looks at the ways in which words break down into components and how that affects their

grammatical status. For example, the letter “s” on the end of a word can often either indicate that it is a plural noun or a third-person present-tense verb.

- **Syntax.** This stage involves applying the rules of the grammar from the language being used. Syntax determines the role of each word in a sentence and, thus, enables a computer system to convert sentences into a structure that can be more easily manipulated.
- **Semantics.** This involves the examination of the meaning of words and sentences. As we will see, it is possible for a sentence to be syntactically correct but to be semantically meaningless. Conversely, it is desirable that a computer system be able to understand sentences with incorrect syntax but that still convey useful information semantically.
- **Pragmatics.** This is the application of human-like understanding to sentences and discourse to determine meanings that are not immediately clear from the semantics. For example, if someone says, “Can you tell me the time?”, most people know that “yes” is not a suitable answer. Pragmatics enables a computer system to give a sensible answer to questions like this.
- In addition to these levels of analysis, natural language processing systems must apply some kind of world knowledge. In most real-world systems, this world knowledge is limited to a specific domain (e.g., a system might have detailed knowledge about the Blocks World and be able to answer questions about this world). The ultimate goal of natural language processing would be to have a system with enough world knowledge to be able to engage a human in discussion on any subject. This goal is still a long way off.
- **Morphological Analysis**
 - In studying the English language, morphology is relatively simple. We have endings such as -ing, -s, and -ed, which are applied to verbs; endings such as -s and -es, which are applied to nouns; we also have the ending -ly, which usually indicates that a word is an adverb.
 - We also have prefixes such as anti-, non-, un-, and in-, which tend to indicate negation, or opposition.
 - We also have a number of other prefixes and suffixes that provide a variety of semantic and syntactic information.

- In practice, however, morphologic analysis for the English language is not terribly complex, particularly when compared with agglutinative languages such as German, which tend to combine words together into single words to indicate combinations of meaning.
- Morphologic analysis is mainly useful in natural language processing for identifying parts of speech (nouns, verbs, etc.) and for identifying which words belong together.
- In English, word order tends to provide more of this information than morphology, however. In languages such as Latin, word order was almost entirely superficial, and the morphology was extremely important. Languages such as French, Italian, and Spanish lie somewhere between these two extremes.
- As we will see in the following sections, being able to identify the part of speech for each word is essential to understanding a sentence. This can partly be achieved by simply looking up each word in a dictionary, which might contain for example the following entries:

(swims, verb, present, singular, third person)

(swimmer, noun, singular)

(swim, verb, present, singular, first and second persons)

(swim, verb, present plural, first, second, and third persons)

(swimming, participle)

(swimmingly, adverb)

(swam, verb, past)

- Clearly, a complete dictionary of this kind would be unfeasibly large. A more practical approach is to include information about standard endings, such as:

(-ly, adverb)

(-ed, verb, past)

(-s, noun, plural)

- This works fine for regular verbs, such as walk, but for all natural languages there are large numbers of irregular verbs, which do not follow these rules. Verbs such as to be and to do are particularly difficult in English as they do not seem to follow any morphologic rules.
- The most sensible approach to morphologic analysis is thus to include a set of rules that work for most regular words and then a list of irregular words.
- For a system that was designed to converse on any subject, this second list would be extremely long. Most natural language systems currently are designed to discuss fairly limited domains and so do not need to include over-large look-up tables.
- In most natural languages, as well as the problem posed by the fact that word order tends to have more importance than morphology, there is also the difficulty of ambiguity at a word level.
- This kind of ambiguity can be seen in particular in words such as trains, which could be a plural noun or a singular verb, and set, which can be a noun, verb, or adjective.

- **BNF**

- Parsing involves mapping a linear piece of text onto a hierarchy that represents the way the various words interact with each other syntactically.
- First, we will look at grammars, which are used to represent the rules that define how a specific language is built up.
- Most natural languages are made up of a number of parts of speech, mainly the following:
 - Verb
 - Noun
 - Adjective
 - Adverb
 - Conjunction
 - Pronoun
 - Article
- In fact it is useful when parsing to combine words together to form syntactic groups. Hence, the words, a dog, which consist of an article and

anoun, can also be described as a noun phrase.

- A noun phrase is one or more words that combine together to represent an object or thing that can be described by a noun. Hence, the following are valid noun phrases: christmas, the dog, that packet of chips, the boy who had measles last year and nearly died, my favorite color
- A noun phrase is not a sentence—it is part of a sentence.
- A verb phrase is one or more words that represent an action. The following are valid verb phrases: swim, eat that packet of chips, walking
- A simple way to describe a sentence is to say that it consists of a noun phrase and a verb phrase. Hence, for example: That dog is eating my packet of chips.
- In this sentence, that dog is a noun phrase, and is eating my packet of chips is a verb phrase. Note that the verb phrase is in fact made up of a verb phrase, is eating, and a noun phrase, my packet of chips.
- A language is defined partly by its grammar. The rules of grammar for a language such as English can be written out in full, although it would be a complex process to do so.
- To allow a natural language processing system to parse sentences, it needs to have knowledge of the rules that describe how a valid sentence can be constructed.
- These rules are often written in what is known as Backus–Naur form (also known as Backus normal form—both names are abbreviated as BNF).
- BNF is widely used by computer scientists to define formal languages such as C++ and Java. We can also use it to define the grammar of a natural language.
- A grammar specified in BNF consists of the following components:
 - Terminal symbols. Each terminal symbol is a symbol or word that appears in the language itself. In English, for example, the terminal symbols are our dictionary words such as the, cat, dog, and so on. In formal languages, the terminal symbols include variable names such as x, y, and so on, but for our purposes we will consider the terminal symbols to be the words in the language.
 - Nonterminal symbols. These are the symbols such as noun, verb

phrase, and conjunction that are used to define words and phrases of the language. A nonterminal symbol is so-named because it is used to represent one or more terminal symbols.

- The start symbol. The start symbol is used to represent a complete sentence in the language. In our case, the start symbol is simply sentence, but in first-order predicate logic, for example, the start symbol would be expression.
 - Rewrite rules. The rewrite rules define the structure of the grammar. Each rewrite rule details what symbols (terminal or nonterminal) can be used to make up each nonterminal symbol.
- Let us now look at rewrite rules in more detail. We saw above that a sentence could take the following form: noun phrase verb phrase
- We thus write the following rewrite rule: $\text{Sentence} \rightarrow \text{NounPhrase VerbPhrase}$ This does not mean that every sentence must be of this form, but simply that a string of symbols that takes on the form of the right-hand side can be rewritten in the form of the left-hand side. Hence, if we see the words

The cat sat on the mat

- we might identify that the cat is a noun phrase and that sat on the mat is a verb phrase. We can thus conclude that this string forms a sentence.
- We can also use BNF to define a number of possible noun phrases.
- Note how we use the “|” symbol to separate the possible right-hand sides in BNF:

$\text{NounPhrase} \rightarrow \text{Noun}$

| Article Noun

| Adjective Noun

| Article Adjective Noun

- Similarly, we can define a verb phrase:

$\text{VerbPhrase} \rightarrow \text{Verb}$

| Verb NounPhrase

| Adverb Verb NounPhrase

- The structure of human languages varies considerably. Hence, a set of rules like this will be valid for one language, but not necessarily for any other language.
- For example, in English it is usual to place the adjective before the noun (black cat, stale bread), whereas in French, it is often the case that the adjective comes after the noun (moulin rouge). Thus far, the rewrite rules we have written consist solely of nonterminal symbols.
- Rewrite rules are also used to describe the parts of speech of individual words (or terminal symbols):

Noun → cat

| dog

| Mount Rushmore

| chickens

Verb → swims

| eats

| climbs

Article → the

| a

Adjective → black

| brown

| green

| stale

Grammars and Languages

- The types of grammars that exist are Noam Chomsky invented a hierarchy of grammars.

- The hierarchy consists of four main types of grammars.
- The simplest grammars are used to define regular languages.
- A regular language is one that can be described or understood by a finite state automaton. Such languages are very simplistic and allow sentences such as “aaaaabbbbb.” Recall that a finite state automaton consists of a finite number of states, and rules that define how the automaton can transition from one state to another.
- A finite state automaton could be designed that defined the language that consisted of a string of one or more occurrences of the letter a. Hence, the following strings would be valid strings in this language:

aaa

a

aaaaaaaaaaaaaaaa

- Regular languages are of interest to computer scientists, but are not of great interest to the field of natural language processing because they are not powerful enough to represent even simple formal languages, let alone the more complex natural languages.
- Sentences defined by a regular grammar are often known as regular expressions.
- The grammar that we defined above using rewrite rules is a context-free grammar.
- It is context free because it defines the grammar simply in terms of which word types can go together—it does not specify the way that words should agree with each.

A stale dog climbs Mount Rushmore.

- It also, allows the following sentence, which is not grammatically correct: Chickens eats.
- A context-free grammar can have only at most one terminal symbol on the right-hand side of its rewrite rules.
- Rewrite rules for a context-sensitive grammar, in contrast, can have more than one terminal symbol on the right-hand side. This enables the grammar to specify number, case, tense, and gender agreement.
- Each context-sensitive rewrite rule must have at least as many symbols on the right-

hand side as it does on the left-hand side.

- Rewrite rules for context-sensitive grammars have the following form:

$$A X B \rightarrow A Y B$$

which means that in the context of A and B, X can be rewritten as Y.

- Each of A, B, X, and Y can be either a terminal or a nonterminal symbol.
- Context-sensitive grammars are most usually used for natural language processing because they are powerful enough to define the kinds of grammars that natural languages use. Unfortunately, they tend to involve a much larger number of rules and are a much less natural way to describe language, making them harder for human developers to design than context free grammars.
- The final class of grammars in Chomsky's hierarchy consists of recursively enumerable grammars (also known as unrestricted grammars).
- A recursively enumerable grammar can define any language and has no restrictions on the structure of its rewrite rules. Such grammars are of interest to computer scientists but are not of great use in the study of natural language processing.
- Parsing: Syntactic Analysis
 - As we have seen, morphologic analysis can be used to determine to which part of speech each word in a sentence belongs. We will now examine how this information is used to determine the syntactic structure of a sentence.
 - This process, in which we convert a sentence into a tree that represents the sentence's syntactic structure, is known as parsing.
 - Parsing a sentence tells us whether it is a valid sentence, as defined by our grammar
 - If a sentence is not a valid sentence, then it cannot be parsed. Parsing a sentence involves producing a tree, such as that shown in Fig 10.1, which shows the parse tree for the following sentence:

The black cat crossed the road.

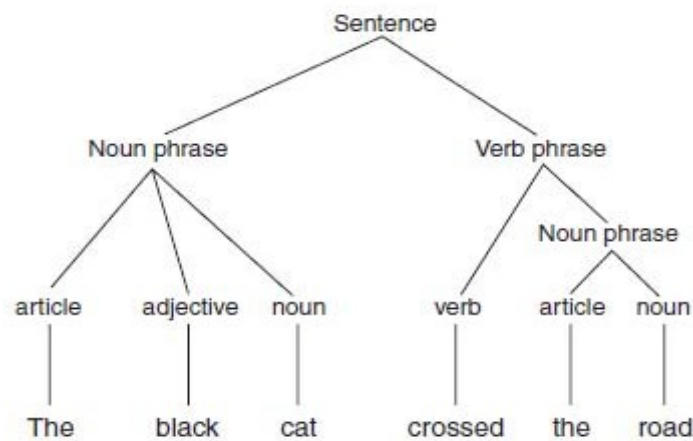


Fig 10.1

- This tree shows how the sentence is made up of a noun phrase and a verb phrase.
- The noun phrase consists of an article, an adjective, and a noun. The verb phrase consists of a verb and a further noun phrase, which in turn consists of an article and a noun.
- Parse trees can be built in a bottom-up fashion or in a top-down fashion.
- Building a parse tree from the top down involves starting from a sentence and determining which of the possible rewrites for Sentence can be applied to the sentence that is being parsed. Hence, in this case, Sentence would be rewritten using the following rule:

Sentence → NounPhrase VerbPhrase

- Then the verb phrase and noun phrase would be broken down recursively in the same way, until only terminal symbols were left.
- When a parse tree is built from the top down, it is known as a derivation tree.
- To build a parse tree from the bottom up, the terminal symbols of the sentence are first replaced by their corresponding nonterminals (e.g., cat is replaced by noun), and then these nonterminals are combined to match the right-hand sides of rewrite rules.
- For example, the and road would be combined using the following rewrite rule: NounPhrase → Article Noun

Basic parsing techniques

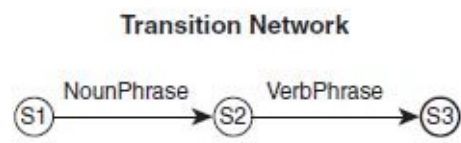
- Transition Networks
 - A transition network is a finite state automaton that is used to represent a part of a grammar.
 - A transition network parser uses a number of these transition networks to represent its entire grammar.
 - Each network represents one nonterminal symbol in the grammar. Hence, in the grammar for the English language, we would have one transition network for Sentence, one for Noun Phrase, one for Verb Phrase, one for Verb, and so on.
 - Fig 10.2 shows the transition network equivalents for three production rules.
 - In each transition network, S1 is the start state, and the accepting state, or final state, is denoted by a heavy border. When a phrase is applied to a transition network, the first word is compared against one of the arcs leading from the first state.
 - If this word matches one of those arcs, the network moves into the state to which that arc points. Hence, the first network shown in Fig 10.2, when presented with a Noun Phrase, will move from state S1 to state S2.
 - If a phrase is presented to a transition network and no match is found from the current state, then that network cannot be used and another network must be tried. Hence, when starting with the phrase the cat sat on the mat, none of the networks shown in Fig 10.2 will be used because they all have only nonterminal symbols, whereas all the symbols in the cat sat on the mat are terminal. Hence, we need further networks, such as the ones shown in Figure 10.2, which deal with terminal symbols.

Production Rule

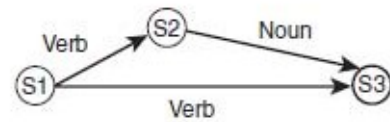
Sentence \rightarrow NounPhrase VerbPhrase

VerbPhrase \rightarrow Verb
 | Verb Noun

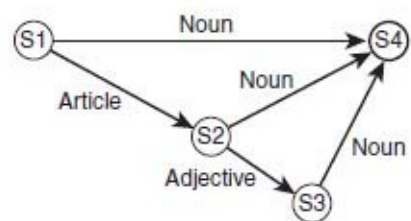
NounPhrase \rightarrow Noun
 | Article Noun
 | Article Adjective Noun



Sentence



VerbPhrase



NounPhrase

Production Rule

Noun \rightarrow cat
 | mat

Article \rightarrow the
 | a

Verb \rightarrow sat

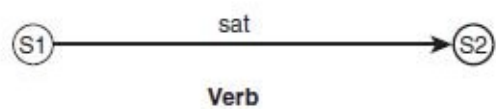
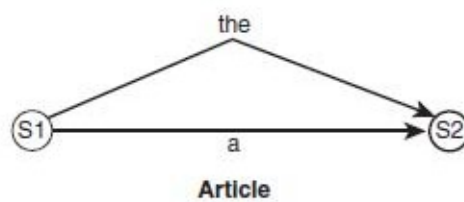
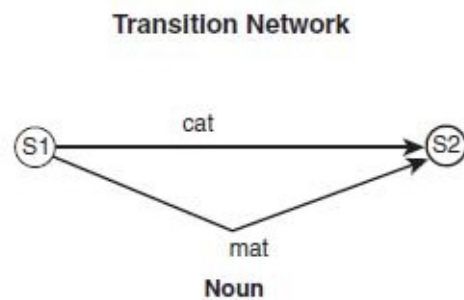


Fig 10.2

- Transition networks can be used to determine whether a sentence is grammatically correct, at least according to the rules of the grammar the networks represent.
- Parsing using transition networks involves exploring a search space of possible parses in a depth-first fashion.
- Let us examine the parse of the following simple sentence: A cat sat.
- We begin in state S1 in the Sentence transition network. To proceed, we must follow the arc that is labeled NounPhrase. We thus move out of the Sentence network and into the NounPhrase network.
- The first arc of the NounPhrase network is labeled Noun. We thus move into the Noun network. We now follow each of the arcs in the Noun network and discover that our first word, A, does not match any of them. Hence, we backtrack to the next arc in the NounPhrase network.
- This arc is labeled Article, so we move on to the Article transition network. Here, on examining the second label, we find that the first word is matched by the terminal symbol on this arc.
- We therefore consume the word, A, and move on to state S2 in the Article network. Because this is a success node, we are able to return to the NounPhrase network and move on to state S2 in this network. We now have an arc labeled Noun.
- As before, we move into the Noun network and find that our next word, cat, matches. We thus move to state S4 in the NounPhrase network. This is a success node, and so we move back to the Sentence network and repeat the process for the VerbPhrase arc.
- It is possible for a system to use transition networks to generate a derivation tree for a sentence, so that as well as determining whether the sentence is grammatically valid, it parses it fully to obtain further information by semantic analysis from the sentence.

This can be done by simply having the system build up the tree by noting which arcs it successfully followed. When, for example, it successfully follows the NounPhrase arc in the Sentence network, the system generates a root node labeled Sentence and an arc leading from

that node to a new node labeled NounPhrase. When the system follows the NounPhrase network and identifies an article and a noun, these are similarly added to the tree.

- In this way, the full parse tree for the sentence can be generated using transition networks. Parsing using transition networks is simple to understand, but is not necessarily as efficient or as effective as we might hope for. In particular, it does not pay any attention to potential ambiguities or the need for words to agree with each other in case, gender, or number.

- **Augmented Transition Networks**

- An augmented transition network, or ATN, is an extended version of a transition network.
- ATNs have the ability to apply tests to arcs, for example, to ensure agreement with number. Thus, an ATN for Sentence would be as shown in Figure 10.2, but the arc from node S2 to S3 would be conditional on the number of the verb being the same as the number for the noun.
- Hence, if the noun phrase were three dogs and the verb phrase were is blue, the ATN would not be able to follow the arc from node S2 to S3 because the number of the noun phrase (plural) does not match the number of the verb phrase (singular).
- In languages such as French, checks for gender would also be necessary. The conditions on the arcs are calculated by procedures that are attached to the arcs. The procedure attached to an arc is called when the network reaches that arc. These procedures, as well as carrying out checks on agreement, are able to form a parse tree from the sentence that is being analyzed.

- **Chart Parsing**

- Parsing using transition networks is effective, but not the most efficient way to parse natural language. One problem can be seen in examining the following two sentences: 1. Have all the fish been fed? , Have all the fish.
- Clearly these are very different sentences—the first is a question, and the second is an instruction. In spite of this, the first three words of each sentence are the same.

- When a parser is examining one of these sentences, it is quite likely to have to backtrack to the beginning if it makes the wrong choice in the first case for the structure of the sentence. In longer sentences, this can be a much greater problem, particularly as it involves examining the same words more than once, without using the fact that the words have already been analyzed.



Fig 10.3

- Another method that is sometimes used for parsing natural language is chart parsing.
- In the worst case, chart parsing will parse a sentence of n words in $O(n^3)$ time. In many cases it will perform better than this and will parse most sentences in $O(n^2)$ or even $O(n)$ time.
- In examining sentence 1 above, the chart parser would note that the words two children form a noun phrase. It would note this on its first pass through the sentence and would store this information in a chart, meaning it would not need to examine those words again on a subsequent pass, after backtracking.
- The initial chart for the sentence The cat eats a big fish is shown in Fig 10.3. It shows the chart that the chart parse algorithm would start with for parsing the sentence.
- The chart consists of seven vertices, which will become connected to each other by edges. The edges will show how the constituents of the sentence combine together.
- The chart parser starts by adding the following edge to the chart: $[0, 0, \text{Target} \rightarrow \bullet \text{Sentence}]$
- This notation means that the edge connects vertex 0 to itself (the first two numbers in the square brackets show which vertices the edge connects). Target is the target that we want to find, which is really just a placeholder to enable us to have an edge that requires us to find a whole sentence. The arrow indicates that in order to make what is on its left-hand side (Target)

we need to find what is on its right-hand side (Sentence). The dot (•) shows what has been found already, on its left-hand side, and what is yet to be found, on its right-hand side. This is perhaps best explained by examining an example.

- Consider the following edge, which is shown in the chart in Figure 10.4:

[0, 2, Sentence → NounPhrase • VerbPhrase]

- This means that an edge exists connecting nodes 0 and 2. The dot shows us that we have already found a NounPhrase (the cat) and that we are looking for a VerbPhrase.



Fig 10.4

- Once we have found the VerbPhrase, we will have what is on the left-hand side of the arrow—that is, a Sentence.
- The chart parser can add edges to the chart using the following three rules:
 - If we have an edge $[x, y, A \rightarrow \square B \bullet C]$, which needs to find a C, then an edge can be added that supplies that C (i.e., the edge $[x, y, C \rightarrow \bullet E]$), where E is some sequence of terminals or nonterminals which can be replaced by a C).
 - If we have two edges, $[x, y, A \rightarrow \square B \bullet C D]$ and $[y, z, C \rightarrow \square E \bullet]$, then these two edges can be combined together to form a new edge: $[x, z, A \rightarrow B C \bullet D]$.
 - If we have an edge $[x, y, A \rightarrow \square B \bullet C]$, and the word at vertex y is of type C, then we have found a suitable word for this edge, and so we extend the edge along to the next vertex by adding the following edge: $[y, y + 1, A \rightarrow B C \bullet]$.

- **Semantic Analysis**

- Having determined the syntactic structure of a sentence, the next task of natural language processing is to determine the meaning of the sentence.
- Semantics is the study of the meaning of words, and semantic analysis is the analysis we use to extract meaning from utterances.

- Semantic analysis involves building up a representation of the objects and actions that a sentence is describing, including details provided by adjectives, adverbs, and prepositions. Hence, after analyzing the sentence The black cat sat on the mat, the system would use a semantic net such as the one shown in Figure 10.5 to represent the objects and the relationships between them.

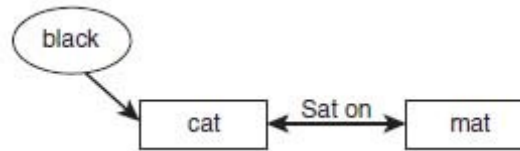


Fig 10.5

- A more sophisticated semantic network is likely to be formed, which includes information about the nature of a cat (a cat is an object, an animal, aquadruped, etc.) that can be used to deduce facts about the cat (e.g., that it likes to drink milk).
- Ambiguity and Pragmatic Analysis
 - One of the main differences between natural languages and formal languages like C++ is that a sentence in a natural language can have more than one meaning. This is ambiguity—the fact that a sentence can be interpreted in different ways depending on who is speaking, the context in which it is spoken, and a number of other factors.
 - The more common forms of ambiguity and look at ways in which a natural language processing system can make sensible decisions about how to disambiguate them.
 - Lexical ambiguity occurs when a word has more than one possible meaning. For example, a bat can be a flying mammal or a piece of sporting equipment. The word set is an interesting example of this because it can be used as a verb, a noun, an adjective, or an adverb. Determining which part of speech is intended can often be achieved by a parser in cases where only one analysis is possible, but in other cases semantic disambiguation is needed to determine which meaning is intended.
 - Syntactic ambiguity occurs when there is more than one possible parse of a sentence. The sentence Jane carried the girl with the spade could be

interpreted in two different ways, as is shown in the two parse trees in Fig 10.6. In the first of the two parse trees in Fig 10.6, the prepositional phrase with the spade is applied to the noun phrase the girl, indicating that it was the girl who had a spade that Jane carried. In the second sentence, the prepositional phrase has been attached to the verb phrase carried the girl, indicating that Jane somehow used the spade to carry the girl.

- Semantic ambiguity occurs when a sentence has more than one possible meaning—often as a result of a syntactic ambiguity. In the example shown in Fig 10.6 for example, the sentence Jane carried the girl with the spade, the sentence has two different parses, which correspond to two possible meanings for the sentence. The significance of this becomes clearer for practical systems if we imagine a robot that receives vocal instructions from a human.

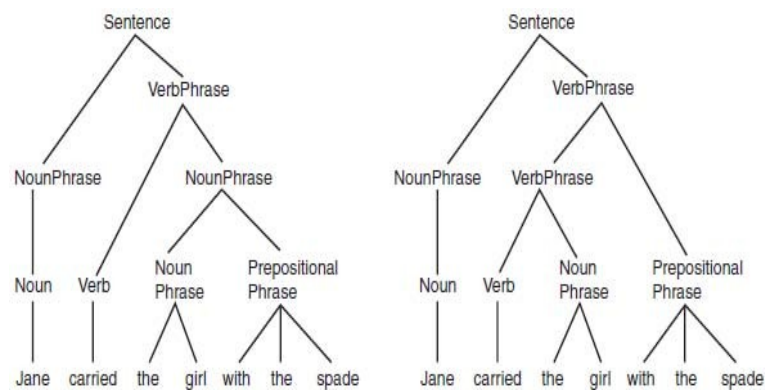


Fig 10.6

- Referential ambiguity occurs when we use anaphoric expressions, or pronouns to refer to objects that have already been discussed. An anaphora occurs when a word or phrase is used to refer to something without naming it. The problem of ambiguity occurs where it is not immediately clear which object is being referred to. For example, consider the following sentences:

John gave Bob the sandwich. He smiled.

- It is not at all clear from this who smiled—it could have been John or Bob. In general, English speakers or writers avoid constructions such as this to avoid humans becoming confused by the ambiguity. In spite of this, ambiguity can also occur in a similar way where a human would not have a problem, such as

John gave the dog the sandwich. It wagged its tail.

- In this case, a human listener would know very well that it was the dog that wagged

its tail, and not the sandwich. Without specific world knowledge, the natural language processing system might not find it so obvious.

- A local ambiguity occurs when a part of a sentence is ambiguous; however, when the whole sentence is examined, the ambiguity is resolved. For example, in the sentence There are longer rivers than the Thames, the phrase longer rivers is ambiguous until we read the rest of the sentence, than the Thames.
- Another cause of ambiguity in human language is vagueness. we examined fuzzy logic, words such as tall, high, and fast are vague and do not have precise numeric meanings.
- The process by which a natural language processing system determines which meaning is intended by an ambiguous utterance is known as disambiguation.
- Disambiguation can be done in a number of ways. One of the most effective ways to overcome many forms of ambiguity is to use probability.
- This can be done using prior probabilities or conditional probabilities. Prior probability might be used to tell the system that the word bat nearly always means a piece of sporting equipment.
- Conditional probability would tell it that when the word bat is used by a sports fan, this is likely to be the case, but that when it is spoken by a naturalist it is more likely to be a winged mammal.
- Context is also an extremely important tool in disambiguation. Consider the following sentences:

I went into the cave. It was full of bats.

I looked in the locker. It was full of bats.

- In each case, the second sentence is the same, but the context provided by the first sentence helps us to choose the correct meaning of the word “bat” in each case.
- Disambiguation thus requires a good world model, which contains knowledge about the world that can be used to determine the most likely meaning of a given word or sentence. The world model would help the system to understand that the sentence Jane carried the girl with the spade is unlikely to mean that Jane used the spade to carry the girl because spades are usually used to carry smaller things than girls. The challenge, of course, is to encode this knowledge in a

way that can be used effectively and efficiently by the system.

The world model needs to be as broad as the sentences the system is likely to hear. For example, a natural language processing system devoted to answering sports questions might not need to know how to disambiguate the sporting bat from the winged mammal, but a system designed to answer any type of question would.