

## CS6659-ARTIFICIAL INTELLIGENCE VI SEMESTER

### UNIT -1

### **SYLLABUS**

#### **INTRODUCTION TO AI AND PRODUCTION SYSTEMS**

Introduction to AI-Problem formulation, Problem Definition-Production systems, Control strategies, Search strategies. Problem characteristics, Production system characteristics-Specialized production system-Problem solving methods-Problem graphs, Matching, Indexing and Heuristic functions-Hill Climbing-Depth first and Breadth first, Constraints satisfaction-Related algorithms, Measure of performance and analysis of search algorithms.

#### **Introduction to AI**

What is artificial intelligence?

Artificial Intelligence (AI) is a branch of Science which deals with helping machines find solutions to complex problems in a more human like fashion.

Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an intelligent agent is a system that perceives its environment and takes actions which maximize its chances of success.

The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

Systems that think like humans "The exciting new effort to make computers think . machines with minds,in the full and literal sense."(Haugeland,1985)	Systems that think rationally "The study of mental faculties through the use of computer models." (Charniak and McDermont,1985)
Systems that act like humans The art of creating machines that perform functions that require intelligence when performed by people."(Kurzweil,1990)	Systems that act rationally "Computational intelligence is the study of the design of intelligent agents."(Poole et al.,1998)

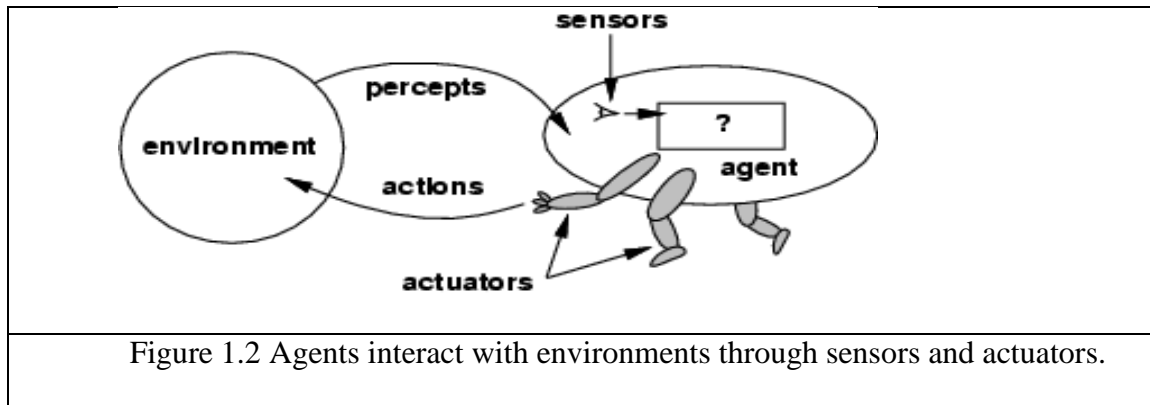
#### **INTELLIGENT AGENTS**

##### **Agents and environments**

An agent is anything that can be viewed as perceiving (aware of) its environment through sensors and SENSOR acting upon that environment through actuators. This simple idea is illustrated in Figure 1.2.

- A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.

- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.



### Percept

We use the term percept to refer to the agent's perceptual inputs at any given instant.

### Percept Sequence

An agent's percept sequence is the complete history of everything the agent has ever perceived.

### Agent function

Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any given percept sequence to an action.

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

### Agent program

Internally, The agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in Figure 1.3. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in Figure 1.4.

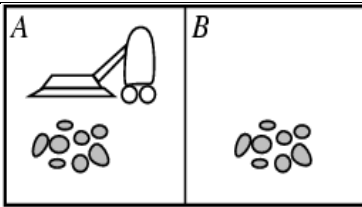


Figure 1.3 A vacuum-cleaner world with just two locations.

### Agent function

Percept Sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
...	

Figure 1.4 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 1.3.

### agent program

```
function REFLEX-VACUUM-AGENT([location, status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

### Rational Agent

A rational agent is one that does the right thing-conceptually speaking, every entry in the table for the agent function is filled out correctly. Obviously, doing the right thing is better than doing the wrong thing. The right action is the one that will cause the agent to be most successful.

### Performance measures

A performance measure embodies the criterion for success of an agent's behavior. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well.

- The performance measure that defines the criterion of success.

- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

### PEAS

All these are grouped together under the heading of the task environment.

We call this the PEAS (Performance, Environment, Actuators, Sensors) description.

In designing an agent, the first step must always be to specify the task environment as fully as possible.

Agent Type	Performance Measure	Environments	Actuators	Sensors
Taxi driver	Safe: fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, Signal, horn, display	Cameras, sonar, Speedometer, GPS, Odometer, engine sensors, keyboards, accelerometer

Figure 1.5 PEAS description of the task environment for an automated taxi.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Figure 1.6 Examples of agent types and their PEAS descriptions.

### Properties of task environments

- Fully observable vs. partially observable
- Deterministic vs. stochastic
- Episodic vs. sequential
- Static vs. dynamic
- Discrete vs. continuous
- Single agent vs. multiagent

Figure 1.7 lists the properties of a number of familiar environments.

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Stochastic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Figure 1.7 Examples of task environments and their characteristics.

### Agent programs

The agent programs all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuator. Notice the difference between the agent program, which takes the current percept as input, and the agent function, which takes the entire percept history. The agent program takes just the current percept as input because nothing more is available from the environment; if the agent's actions depend on the entire percept sequence, the agent will have to remember the percepts.

Function TABLE-DRIVEN-AGENT(*percept*) returns an action

```

static: percepts, a sequence initially empty
       table, a table of actions, indexed by percept sequence

append percept to the end of percepts
action ← LOOKUP(percepts, table)
return action
```

Figure 1.8 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time.

**Drawbacks:**

- Table lookup of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
- Problems
  - Too big to generate and to store (Chess has about  $10^{120}$  states, for example)
  - No knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
  - Looping: Can't make actions conditional
- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

**Some Agent Types**

- Table-driven agents
  - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) lookup table.
- Simple reflex agents
  - are based on condition-action rules, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.
- Agents with memory
  - have internal state, which is used to keep track of past states of the world.
- Agents with goals
  - are agents that, in addition to state information, have goal information that describes desirable situations. Agents of this kind take future events into consideration.
- Utility-based agents
  - base their decisions on classic axiomatic utility theory in order to act rationally.

**Simple Reflex Agent**

The simplest kind of agent is the simple reflex agent. These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure 1.10 is a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.

- Select action on the basis of *only the current* percept.  
E.g. the vacuum-agent
- Large reduction in possible percept/action situations(next page).
- Implemented through *condition-action rules*  
If dirty then suck

**Problem Definition :**

- The process of working through details of a problem to reach a solution.
- Problem solving may include mathematical or systematic operations.
- 4 necessity things to solve a problem
  1. Define the problem

The definition must include specification of the initial situations and also final situations.

2. Analyze the problem  
Apply the various techniques for solving the problems
3. Isolate and represent the knowledge to solve the problem
4. Choose the best problem.

### Example: Water Jug Problem

Consider the following problem:

A Water Jug Problem: 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?

Solutions:

- State:  $(x, y)$   
 $x = 0, 1, 2, 3, \text{ or } 4$        $y = 0, 1, 2, 3$
- Start state:  $(0, 0)$ .
- Goal state:  $(2, n)$  for any  $n$ .
- Attempting to end up in a goal state.

State Space Search: Water Jug Problem

1.  $(x, y) \rightarrow (4, y)$   
if  $x < 4$
2.  $(x, y) \rightarrow (x, 3)$   
if  $y < 3$
3.  $(x, y) \rightarrow (x - d, y)$   
if  $x > 0$
4.  $(x, y) \rightarrow (x, y - d)$   
if  $y > 0$
5.  $(x, y) \rightarrow (0, y)$   
if  $x > 0$
6.  $(x, y) \rightarrow (x, 0)$   
if  $y > 0$
7.  $(x, y) \rightarrow (4, y - (4 - x))$

- if  $x + y \geq 4, y > 0$
8.  $(x, y) \rightarrow (x - (3 - y), 3)$
- if  $x + y \geq 3, x > 0$
9.  $(x, y) \rightarrow (x + y, 0)$
- if  $x + y \leq 4, y > 0$
10.  $(x, y) \rightarrow (0, x + y)$
- if  $x + y \leq 3, x > 0$
11.  $(0, 2) \rightarrow (2, 0)$
12.  $(2, y) \rightarrow (0, y)$

Solving by production rules:

1. current state = (0, 0)
2. Loop until reaching the goal state (2, 0)
  - Apply a rule whose left side matches the current state
  - Set the new current state to be the resulting state

- 0: (4, 0) (3, 0)-----→ empty the 4 gallon jug & 3 gallon jug
- 1: (4, 3) (3, 0)-----→ fill the 4 gallon jug
- 2: (4, 1) (3, 3)----→ pour water from the 4 gallon jug into 3 gallon jug until 3 gallon jug is full
- 3: (4, 1) (3, 0)----→ empty the 3 gallon jug
- 4: (4, 0) (3, 1)----→ fill the 3 gallon jug with remaining water from the 4 gallon jug
- 5: (4, 4) (3, 1)----→ fill the 4 gallon jug
- 6: (4, 2) (3, 3)----→ pour water from the 4 gallon jug into 3 gallon jug until jug gets filled

**EXAMPLE 2:**

The 8-puzzle

An 8-puzzle consists of a 3x3 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 the goal state, as shown in figure 2.4

Example: The 8-puzzle



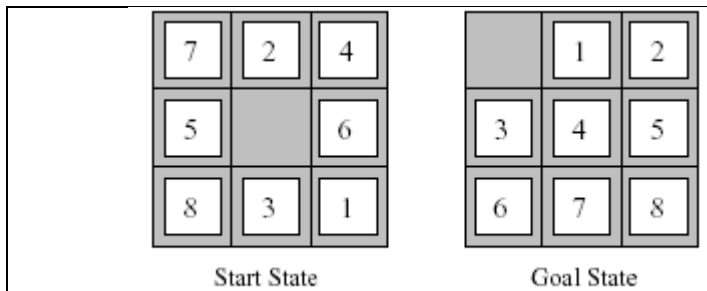


Figure 1.21 A typical instance of 8-puzzle.

The problem formulation is as follows :

- 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. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- 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 shown in figure 2.4.(Other goal configurations are possible)
- Path cost : Each step costs 1,so the path cost is the number of steps in the path.

### **EXAMPLE 3:**8-queens problem

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

Figure 2.5 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.

An Incremental formulation involves operators that augments the state description,starting with an empty state.for 8-queens problem,this means each action adds a queen to the state.

A complete-state formulation starts with all 8 queens on the board and move them around.

In either case the path cost is of no interest because only the final state counts.

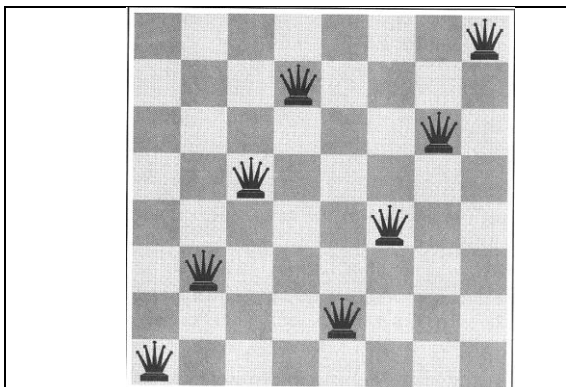


Figure 1.22 8-queens problem

The first incremental formulation one might try is the following :

- States : Any arrangement of 0 to 8 queens on board is a state.
- Initial state : No queen on the board.
- Successor function : Add a queen to any empty square.
- Goal Test : 8 queens are on the board, none attacked.

In this formulation, we have  $64 \times 63 \times \dots \times 57 = 3 \times 10^{14}$  possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked.

- States : Arrangements of  $n$  queens ( $0 \leq n \leq 8$ ), one per column in the left most columns, with no queen attacking another are states.
- Successor function : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queen state space from  $3 \times 10^{14}$  to just 2057, and solutions are easy to find. For the 100 queens the initial formulation has roughly  $10^{400}$  states whereas the improved formulation has about  $10^{52}$  states. This is a huge reduction, but the improved state space is still too big for the algorithms to handle.

### **AIRLINE TRAVEL PROBLEM**

The airline travel problem is specified as follows :

- States : Each is represented by a location (e.g., an airport) and the current time.
- Initial state : This is specified by the problem.
- Successor function : This returns the states resulting from taking any scheduled flight (further specified by seat class and location), leaving later than the current time plus the within-airport transit time, from the current airport to another.
- Goal Test : Are we at the destination by some prespecified time?
- Path cost : This depends upon the monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of air plane, frequent-flyer mileage awards, and so on.

### **Production system**

- Production system - uses knowledge in the form of rules to provide diagnoses or advice on the basis of input data.
- Parts
  - Database of rules (knowledge base)
  - Database of facts
  - Inference engine which reasons about the facts using the rules

(detail in III unit)

### **Problem solving methods**

### **Problem graphs:**

**DEFINITION****GRAPH**

A graph consists of:

A set of *nodes*  $N_1, N_2, N_3, \dots, N_n, \dots$ , which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc  $(N_3, N_4)$  connects node  $N_3$  to node  $N_4$ . This indicates a direct connection from node  $N_3$  to  $N_4$  but not from  $N_4$  to  $N_3$ , unless  $(N_4, N_3)$  is also an arc, and then the arc joining  $N_3$  and  $N_4$  is undirected.

If a directed arc connects  $N_j$  and  $N_k$ , then  $N_j$  is called the *parent* of  $N_k$  and  $N_k$ , the *child* of  $N_j$ . If the graph also contains an arc  $(N_j, N_l)$ , then  $N_k$  and  $N_l$  are called *siblings*.

A *rooted* graph has a unique node  $N_s$  from which all paths in the graph originate. That is, the root has no parent in the graph.

A *tip* or *leaf* node is a node that has no children.

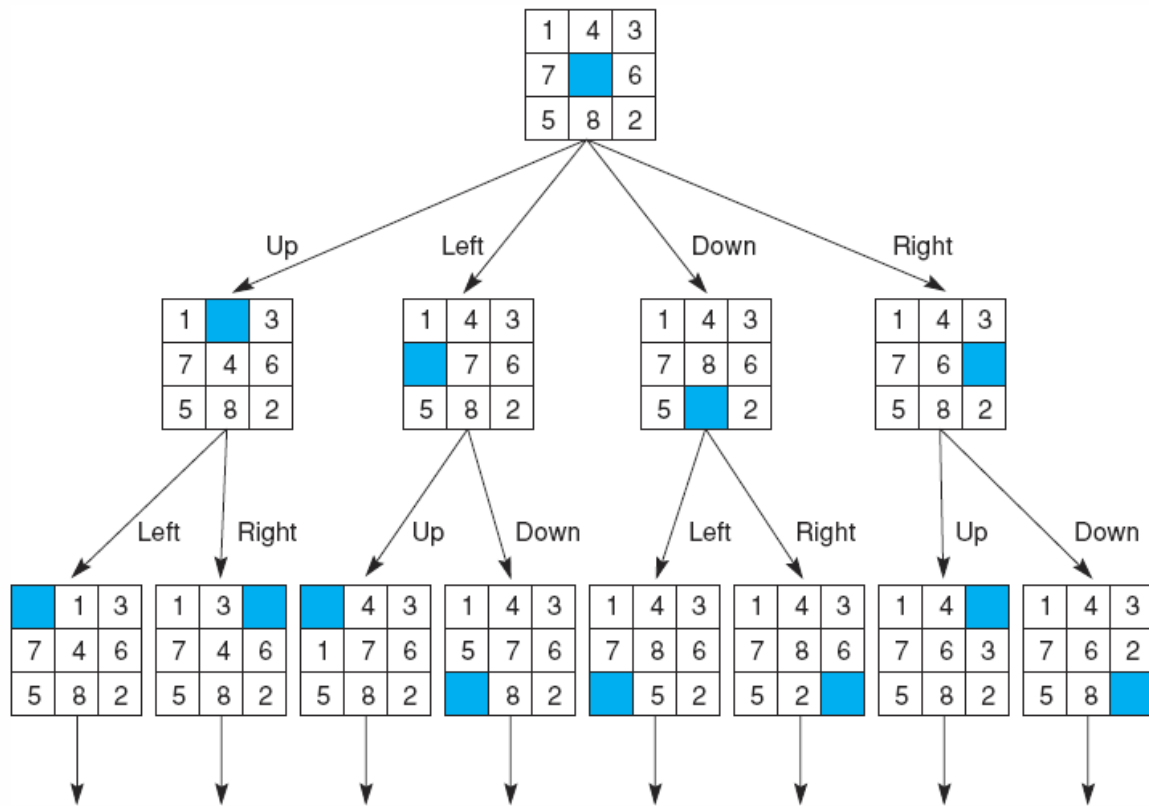
An ordered sequence of nodes  $[N_1, N_2, N_3, \dots, N_n]$ , where each pair  $N_i, N_{i+1}$  in the sequence represents an arc, i.e.,  $(N_i, N_{i+1})$ , is called a *path* of length  $n - 1$ .

On a path in a rooted graph, a node is said to be an *ancestor* of all nodes positioned after it (to its right) as well as a *descendant* of all nodes before it.

A path that contains any node more than once (some  $N_i$  in the definition of path above is repeated) is said to contain a *cycle* or *loop*.

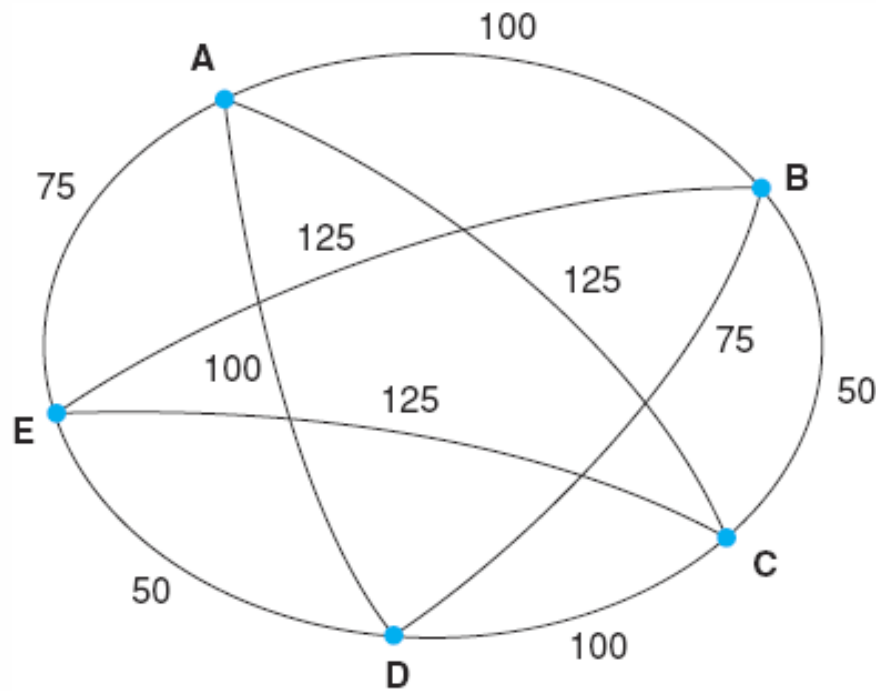
A *tree* is a graph in which there is a unique path between every pair of nodes. (The paths in a tree, therefore, contain no cycles.)

**Example 1: 8-puzzle problem (GRAPH)**

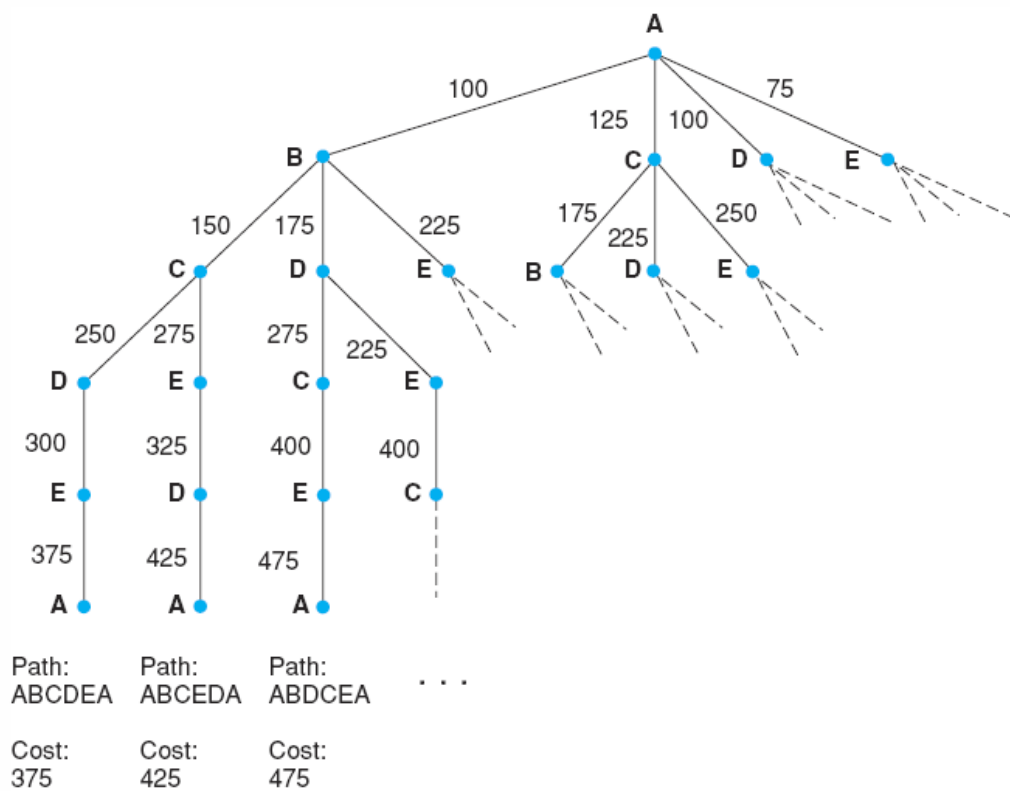


State space of the 8-puzzle generated by "move blank" operations

### Example 2 :travelling salesperson problem



**Fig An instance of the travelling salesperson problem**

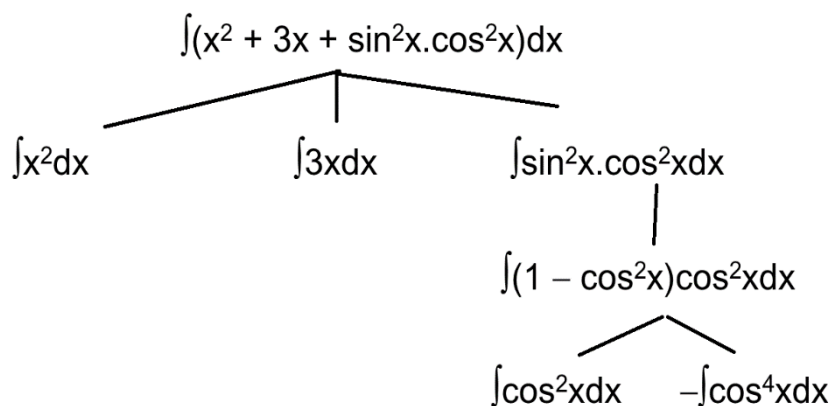


## **Problem Characteristics**

To choose an appropriate method for a particular problem:

- Is the problem decomposable?
- Can solution steps be ignored or undone?
- Is the universe predictable?
- Is a good solution absolute or relative?
- Is the solution a state or a path?
- What is the role of knowledge?
- Does the task require human-interaction?

### ➤ **Is the problem decomposable?**



### ➤ **Can solution steps be ignored or undone?**

- Ignorable problems can be solved using a simple control structure that never backtracks.
- Recoverable problems can be solved using backtracking.
- Irrecoverable problems can be solved by recoverable style methods via planning.

### ➤ **Is the universe predictable?**

The 8-Puzzle

Every time we make a move, we know exactly what will happen.

Certain outcome!

### ➤ **Is a good solution absolute or relative?**

The Travelling Salesman Problem

We have to try all paths to find the shortest one.

- Any-path problems can be solved using heuristics that suggest good paths to explore.  
For best-path problems, much more exhaustive search will be performed

### ➤ **Is the solution a state or a path?**

Finding a consistent interpretation

“The bank president ate a dish of pasta salad with

the fork”.

- “bank” refers to a financial situation or to a side of a river?
  - “dish” or “pasta salad” was eaten?
  - Does “pasta salad” contain pasta, as “dog food” does not contain “dog”?
  - Which part of the sentence does “with the fork” modify?
- What if “with vegetables” is there?

No record of the processing is necessary.

#### ➤ **Is the solution a state or a path?**

The Water Jug Problem

The path that leads to the goal must be reported.

- A path-solution problem can be reformulated as a state-solution problem by describing a state as a partial path to a solution.
- The question is whether that is natural or not.

#### ➤ **What is the role of knowledge**

Playing Chess

Knowledge is important only to constrain the search for a solution.

Reading Newspaper

Knowledge is required even to be able to recognize a solution.

## **SEARCHING FOR SOLUTIONS**

### **What is Search?**

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

The set of possible states, together with *operators* defining their connectivity constitute the *search space*.

The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

### **SEARCH TREE**

Having formulated some problems, we now need to solve them. This is done by a search through the state space. A search tree is generated by the initial state and the successor function that together define the state space. In general, we may have a *search graph* rather than a *search tree*, when the same state can be reached from multiple paths.

### **Types of Search**

There are three broad classes of search processes:

#### 1) Uninformed- Blind Search-

- There is no specific reason to prefer one part of the search space to any other, in finding a path from initial state to goal state.
- systematic, exhaustive search
  - depth-first-search
  - Breadth-first-search

#### 2) Informed – Heuristic search - there is specific information to focus the search.

- Hill climbing
  - Branch and bound
  - Best first
  - A\*
- 3)Game playing – there are at least two partners opposing to each other.
- Minimax (a, b pruning)
  - Means ends analysis

### **UNINFORMED SEARCH STRATGES**

Uninformed Search Strategies have no additional information about states beyond that provided in the problem definition.

Strategies that know whether one non goal state is “more promising” than another are called Informed search or heuristic search strategies.

There are five uninformed search strategies as given below.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

#### **Breadth-first search**

- Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breadth-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue, assuring that the nodes that are visited first will be expanded first.
- In other words, calling TREE-SEARCH(problem, FIFO-QUEUE()) results in breadth-first-search.
- The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.



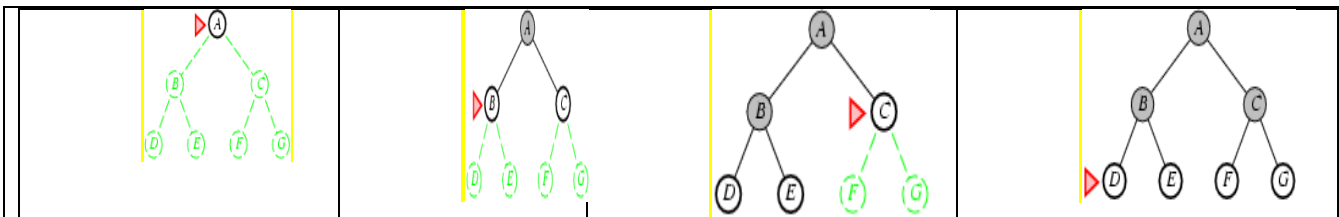


Figure 1.27 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

### Properties of breadth-first-search

Complete?? Yes (if  $b$  is finite)

Time??  $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$ , i.e., exp. in  $d$

Space??  $O(b^{d+1})$  (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec  
so 24hrs = 8640GB.

Figure 1.28 Breadth-first-search properties

### Time and Memory Requirements for BFS – $O(b^{d+1})$

#### Example:

- $b = 10$
- 10000 nodes/second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	.11 sec	1 meg
4	111,100	11 sec	106 meg
6	$10^7$	19 min	10 gig
8	$10^9$	31 hrs	1 tera
10	$10^{11}$	129 days	101 tera
12	$10^{13}$	35 yrs	10 peta
14	$10^{15}$	3523 yrs	1 exa

Figure 1.29 Time and memory requirements for breadth-first-search. The numbers shown assume branch factor of  $b = 10$ ; 10,000 nodes/second; 10 bytes/node

### Time complexity for BFS

Assume every state has  $b$  successors. The root of the search tree generates  $b$  nodes at the first level, each of which generates  $b$  more nodes, for a total of  $b^2$  at the second level. Each of these generates  $b$  more nodes, yielding  $b^3$  nodes at the third level, and so on. Now suppose, that the solution is at depth  $d$ . In the worst case, we would expand all but the last node at level  $d$ , generating  $b^{d+1} - b$  nodes at level  $d+1$ .

Then the total number of nodes generated is

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space complexity is, therefore, the same as the time complexity.

## UNIFORM-COST SEARCH

Instead of expanding the shallowest node, uniform-cost search expands the node  $n$  with the lowest path cost. uniform-cost search does not care about the number of steps a path has, but only about their total cost.

Expand least-cost unexpanded node

Implementation:

*fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost  $\geq \epsilon$

Time?? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$   
where  $C^*$  is the cost of the optimal solution

Space?? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of  $g(n)$

Figure 1.30 Properties of Uniform-cost-search

## DEPTH-FIRST-SEARCH

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in figure 1.31. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search “backs up” to the next shallowest node that still has unexplored successors.

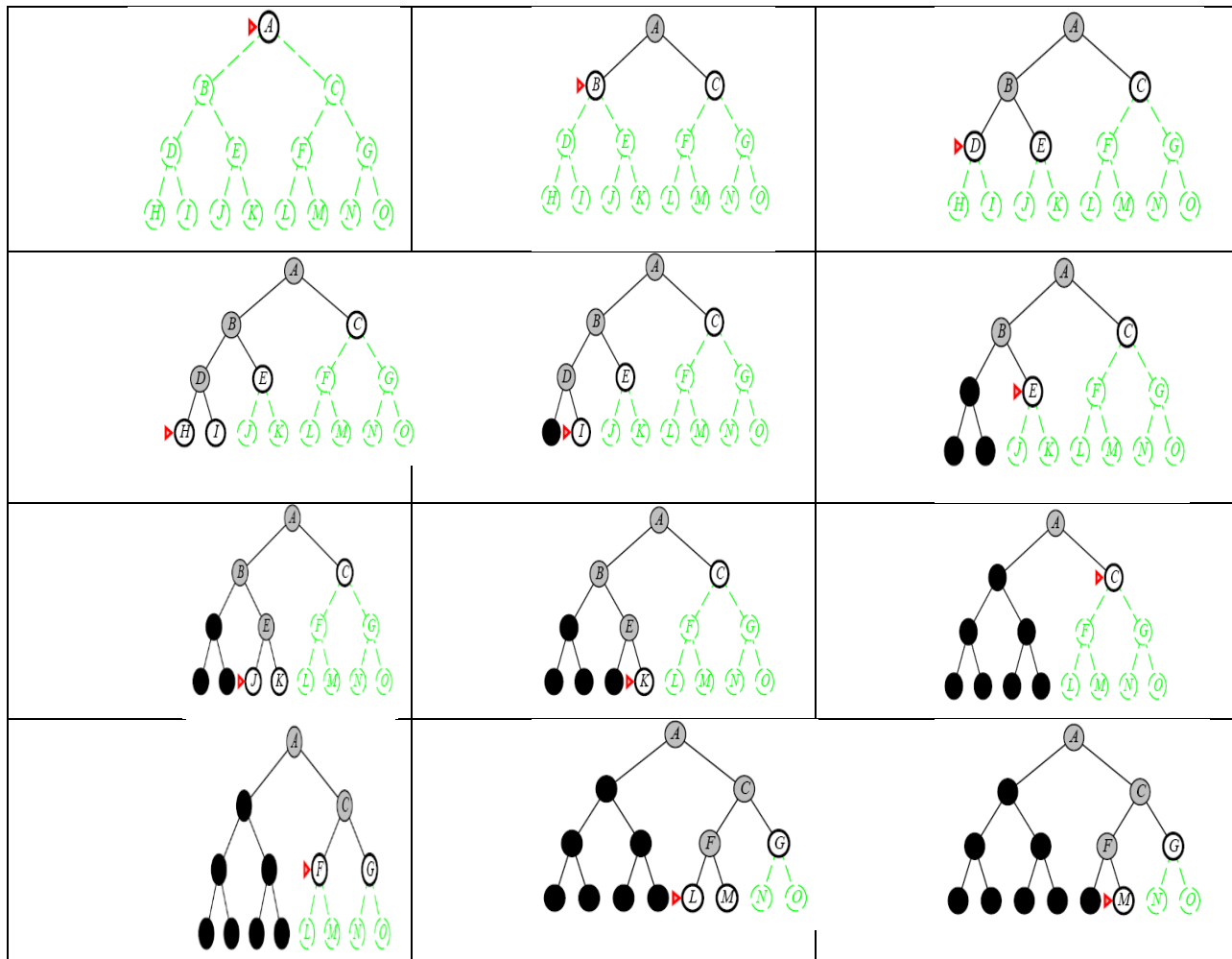


Figure 1.31 Depth-first-search on a binary tree. Nodes that have been expanded and have descendants in the fringe can be removed from the memory; these are shown in black. Node at depth 3 are assumed to have no successors and M is the only goal node.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

Depth-first-search has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored (Refer Figure 2.12).

For a state space with a branching factor  $b$  and maximum depth  $m$ , depth-first-search requires storage of only  $bm + 1$  nodes.

**Drawback of Depth-first-search**

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

**BACKTRACKING SEARCH**

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only  $O(m)$  memory is needed rather than  $O(bm)$

**Heuristic Search**

Reasons for heuristics

- impossible for exact solution,
- heuristics lead to promising path
- no exact solution but an acceptable one
- fallible due to limited information

*Intelligence for a system with limited processing resources consists in making wise choices of what to do next*

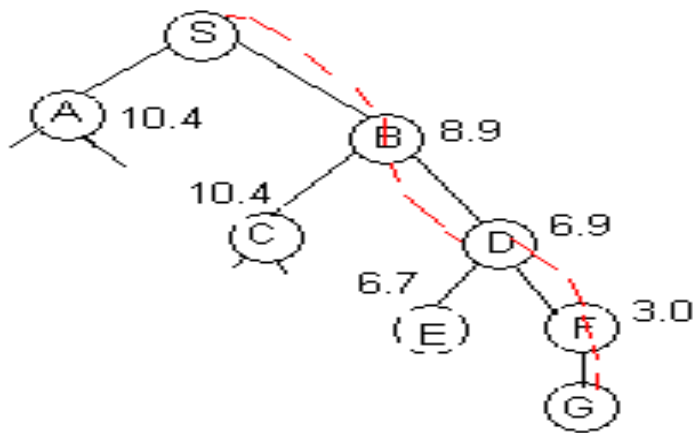
Heuristics = Search Algorithm + Measure

**Hill Climbing**

Hill climbing is depth first search with a heuristic measurement that orders choices as nodes are expanded. The algorithm is the same only 2b differs slightly.

1. Form a one element queue consisting of the root node
2. Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.
  - a) If the first element is the goal do nothing.
  - b) If the first element is not the goal node remove the first element from the queue, sort the first element's children, if any by estimated remaining distance, and add the first element's children if any to the front of the queue.
3. If the goal node has been found announce success, otherwise announce failure.

**Hill climbing example**



Problems that may arise:

- A local maximum, is a state that is better than all its neighbors, but is not better than some other states farther away. At a local maximum, all moves appear to make things worse.
- A plateau, A whole set of neighboring states have the same value. It is not possible to determine the best direction.
- A ridge, Higher than surrounding area but can not be traversed by single move in any one direction.

Some ways of dealing with these:

- Backtrack: local maximum
- Make a big jump in one direction to try to get to new section of search space (plateau)
- Apply two or more rules before doing the test. This corresponds to moving in several directions at once (ridges).

### Best-first Search

Best-first search is a combination of depth-first and breadth-first search algorithms. Forward motion is from the best open node (most promising) so far, no matter where it is on the partially developed tree.

The second step of the algorithm changes as:

2. Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.

a) If the first element is the goal do nothing.

b) If the first element is not the goal node remove the first element from the queue and add the first element's children if any to the queue and sort the entire queue by estimated remaining distance..

procedure best\_first\_search;

begin

    open := [Start]; closed = [];

while open <> [] do

    remove leftmost state from open, call it X;

    if X = goal then return path from Start to X

    else begin

        generate children of X; for each child of X do

### Example of best-first search



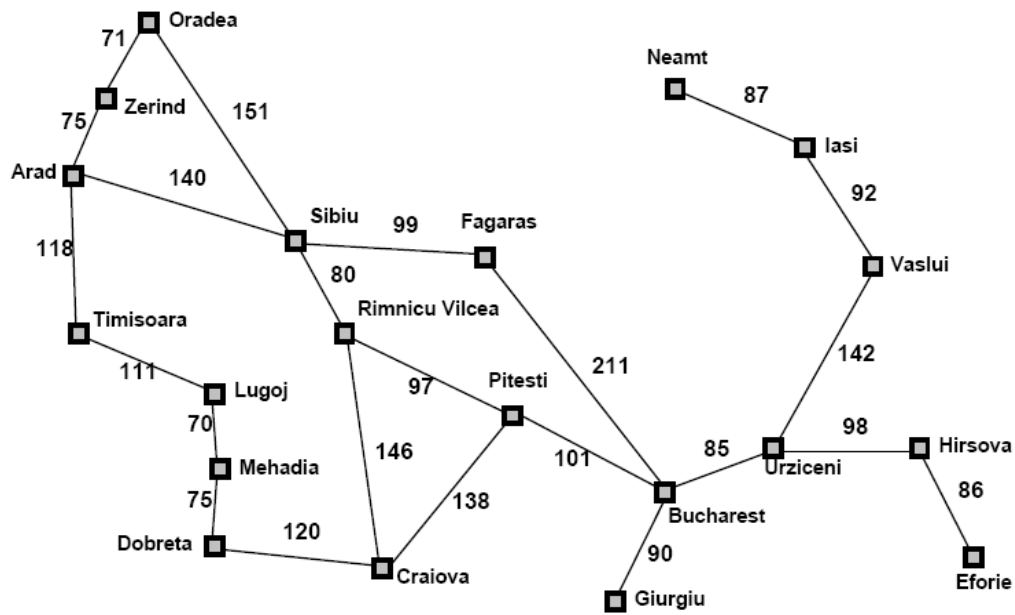
1. open = [A5]; closed = []
2. eval A5; open = [B4, C4, D6];  
closed = [A5]
3. eval B4; open = [C4, E5, F5, D6]; closed = [B4, A5]
4. eval C4; open = [H3, G4, E5, F5, D6] closed = [C4, B4, A5]
5. eval H3; open = [O2, P3, G4, E5, F5, D6]; closed = [H3, C4, B4, A5]
6. eval O2; open = [P3, G4, E5, F5, D6]  
closed = [O2, H3, C4, B4, A5]
7. eval P3; the solution is found!

### A\* Search

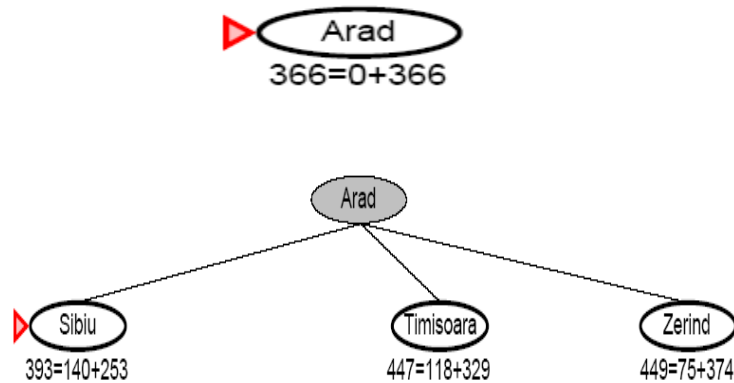
- A\* (A star) is the most widely known form of Best-First search
  - It evaluates nodes by combining  $g(n)$  and  $h(n)$
  - $f(n) = g(n) + h(n)$
  - Where
    - $g(n)$  = cost so far to reach  $n$
    - $h(n)$  = estimated cost to goal from  $n$
    - $f(n)$  = estimated total cost of path through  $n$
- When  $h(n)$  = actual cost to goal
  - Only nodes in the correct path are expanded
  - Optimal solution is found
- When  $h(n) <$  actual cost to goal
  - Additional nodes are expanded
  - Optimal solution is found
- When  $h(n) >$  actual cost to goal
  - Optimal solution can be overlooked
- A\* is optimal if it uses an admissible heuristic
  - $h(n) \leq h^*(n)$  the true cost from node  $n$
  - if  $h(n)$  never overestimates the cost to reach the goal
- Example
  - $h_{SLD}$  never overestimates the actual road distance

### Greedy Best-First Search

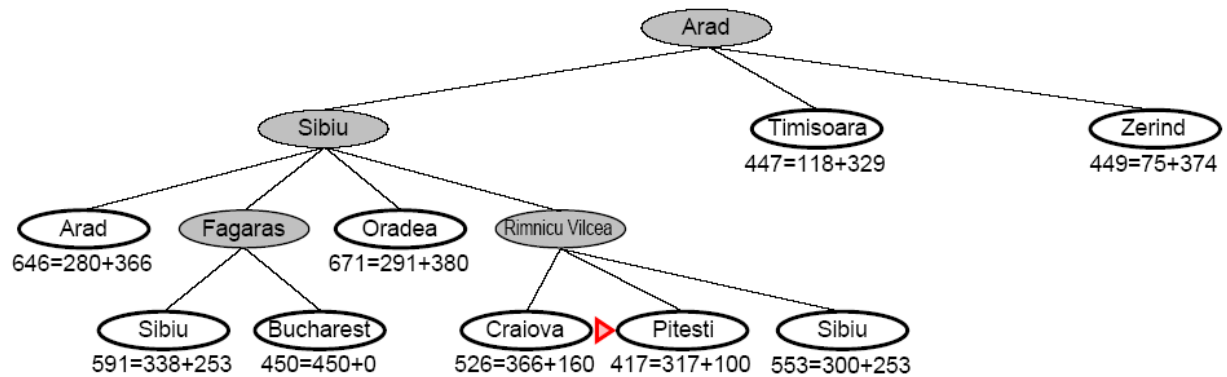
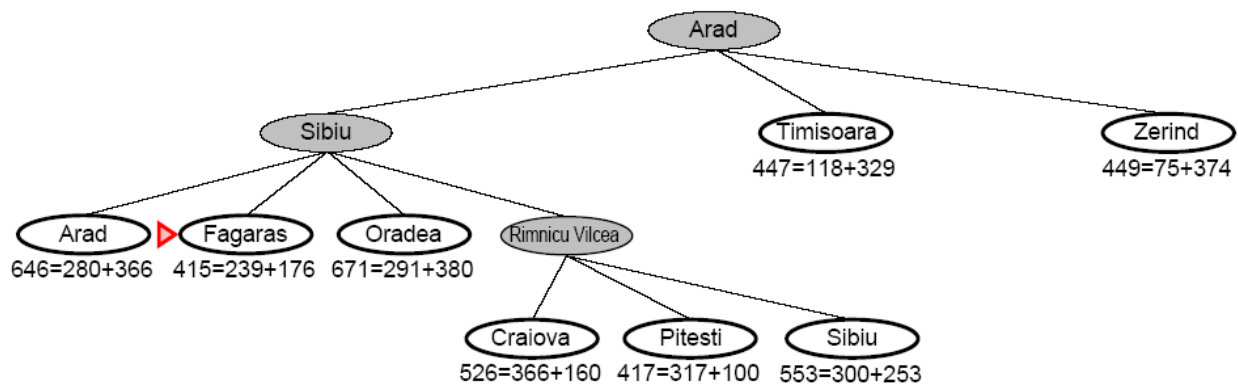
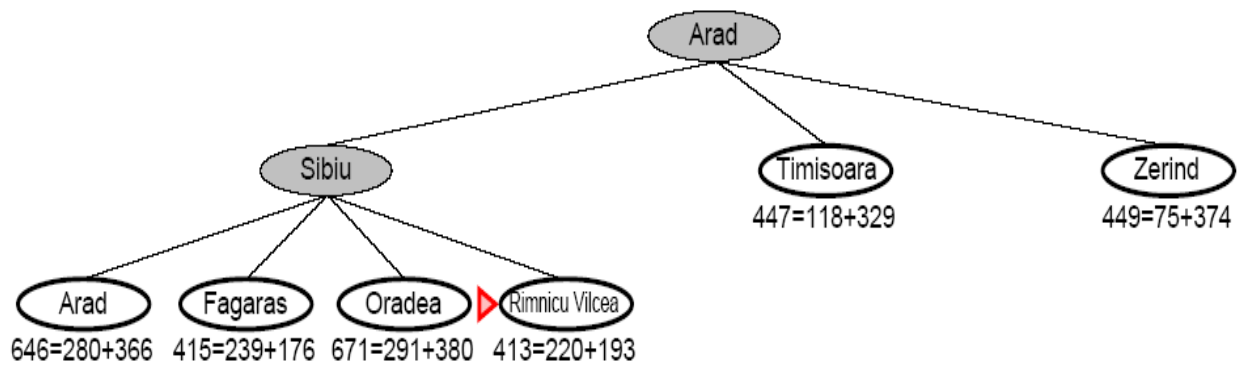
UNIT-3/PIT/IT/Artificial Intelligence

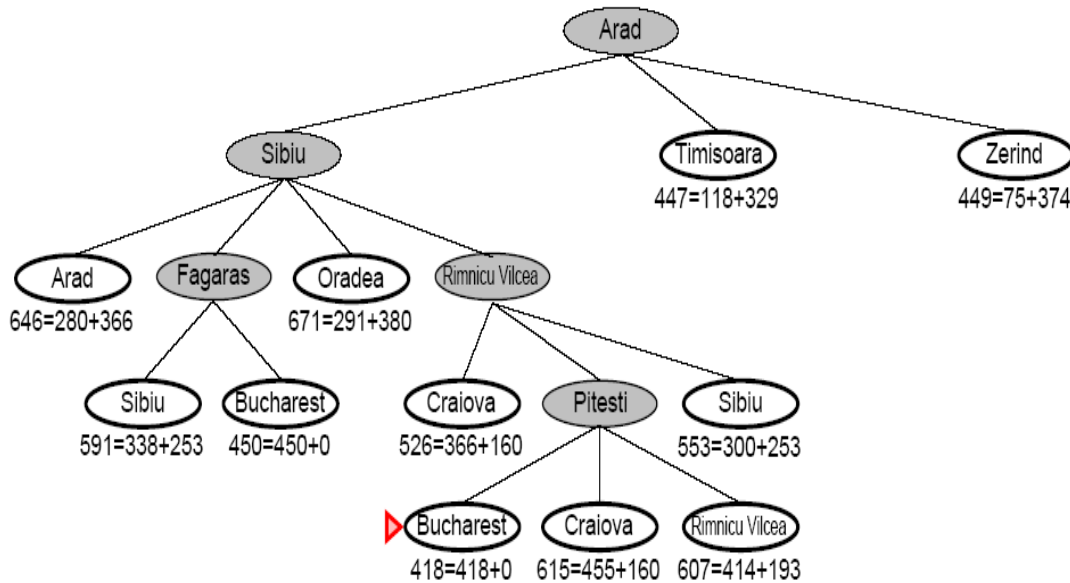


A\* Search









- Complete
  - Yes, unless there are infinitely many nodes with  $f \leq f(G)$
- Time
  - Exponential in [relative error of  $h \times$  length of soln]
  - The better the heuristic, the better the time
    - Best case  $h$  is perfect,  $O(d)$
    - Worst case  $h = 0$ ,  $O(b^d)$  same as BFS
- Space
  - Keeps all nodes in memory and save in case of repetition
  - This is  $O(b^d)$  or worse
  - A\* usually runs out of space before it runs out of time
- Optimal
  - Yes, cannot expand  $f_{i+1}$  unless  $f_i$  is finished

### Admissible heuristics(Heuristics function)

A heuristic  $h(n)$  is admissible if for every node  $n$ ,

$$h(n) \leq h^*(n), \text{ where } h^*(n) \text{ is the true cost to reach the goal state from } n.$$

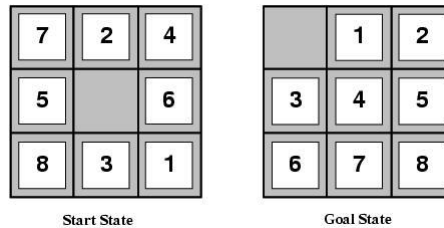
An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

### Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$  = number of misplaced tiles

$h_2(n)$  = total Manhattan distance (no. of squares from desired location of each tile)



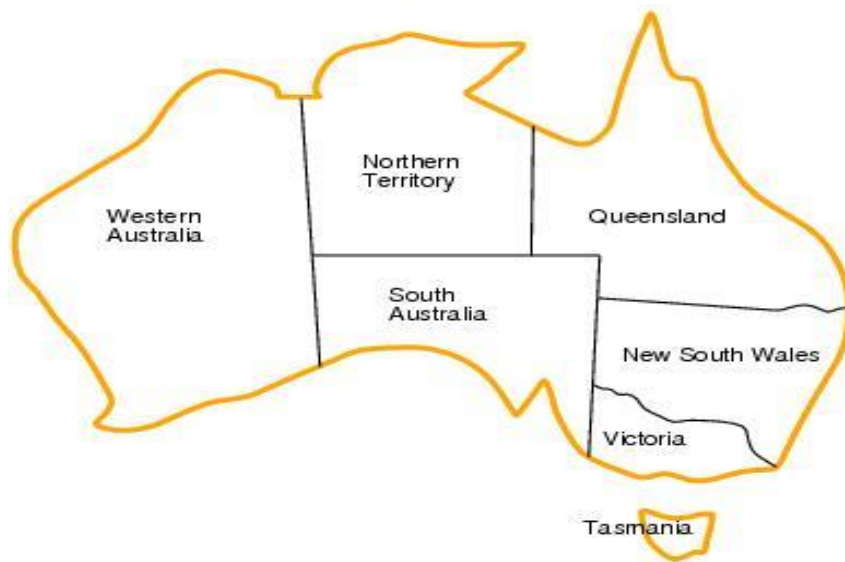
$$h_1(S) = 8$$

$$h_2(S) = 3+1+2+2+2+3+3+2 = 18$$

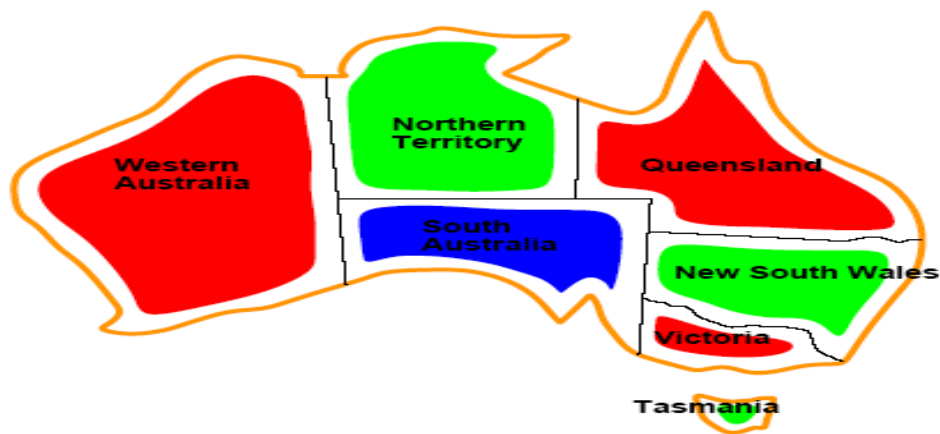
### constraints satisfaction

- What is a CSP
  - Finite set of variables  $V_1, V_2, \dots, V_n$ 
    - Nonempty domain of possible values for each variable  
 $D_{V_1}, D_{V_2}, \dots, D_{V_n}$
  - Finite set of constraints  $C_1, C_2, \dots, C_m$ 
    - Each constraint  $C_i$  limits the values that variables can take,
      - e.g.,  $V_1 \neq V_2$
  - A *state* is defined as an *assignment* of values to some or all variables.
- *Consistent assignment*
  - assignment does not violate the constraints
  - CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain specific expertise).
- An assignment is *complete* when every variable is mentioned.
- A *solution* to a CSP is a complete assignment that satisfies all constraints.
- Some CSPs require a solution that maximizes an *objective function*.
- Examples of Applications:
  - Scheduling the time of observations on the Hubble Space Telescope
  - Airline schedules
  - Cryptography
  - Computer vision -> image interpretation
  - Scheduling your MS or PhD thesis exam ☺

CSP example: map coloring

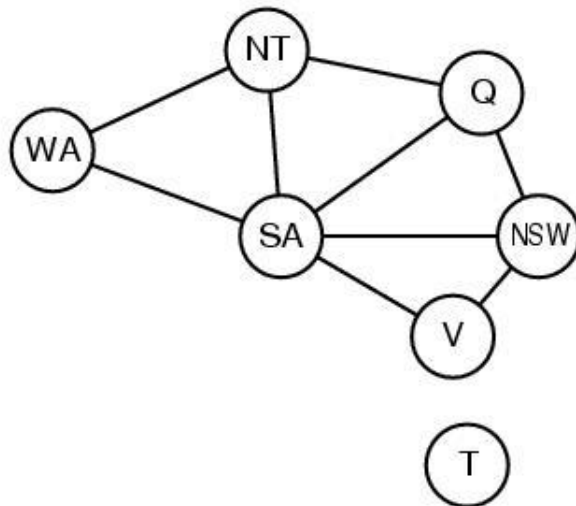


- Variables:  $WA, NT, Q, NSW, V, SA, T$
- Domains:  $D_i = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors.
  - E.g.  $WA \neq NT$



- Solutions are assignments satisfying all constraints, e.g.  
 $\{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green\}$
- More general problem than map coloring
  - Planar graph = graph in the 2d-plane with no edge crossings

- Guthrie's conjecture (1852)  
*Every planar graph can be colored with 4 colors or less*
  - Proved (using a computer) in 1977 (Appel and Haken)



- Constraint graph:
  - nodes are variables
  - arcs are binary constraints
- Graph can be used to simplify search  
 e.g. Tasmania is an independent subproblem  
 (will return to graph structure later)

CSP Example: Cryptarithmic puzzle

### Crypt-Arithmetic puzzle

#### ■ Problem Statement:

- ☐ Solve the following puzzle by assigning numeral (0-9) in such a way that each letter is assigned unique digit which satisfy the following addition.
- ☐ Constraints : No two letters have the same value. (The constraints of arithmetic).

$$\begin{array}{r}
 \phantom{+} \quad \text{S} \quad \text{E} \quad \text{N} \quad \text{D} \\
 + \quad \text{M} \quad \text{O} \quad \text{R} \quad \text{E} \\
 \hline
 \text{M} \quad \text{O} \quad \text{N} \quad \text{E} \quad \text{Y} \\
 \hline
 \end{array}$$

#### ■ Initial Problem State

- ☐ S = ? ; E = ? ; N = ? ; D = ? ; M = ? ; O = ? ; R = ? ; Y = ?

Carries :

$$C_4 = ? ; C_3 = ? ; C_2 = ? ; C_1 = ?$$

$C_4$	$C_3$	$C_2$	$C_1$	$\longleftarrow$	Carry
	S	E	N	D	
+	M	O	R	E	
M	O	N	E	Y	

Constraint equations:

$$\begin{array}{rcl}
 Y & = & D + E & C_1 \\
 E & = & N + R + C_1 & C_2 \\
 N & = & E + O + C_2 & C_3 \\
 O & = & S + M + C_3 & C_4 \\
 M & = & C_4 & 
 \end{array}$$

We can easily see that M has to be non zero digit, so the value of  $C_4 = 1$

$$M = C_4 \Rightarrow M = 1$$

$$O = S + M + C_3 \rightarrow C_4$$

For  $C_4 = 1$ ,  $S + M + C_3 > 9 \Rightarrow$

$$S + 1 + C_3 > 9 \Rightarrow S + C_3 > 8.$$

If  $C_3 = 0$ , then  $S = 9$  else if  $C_3 = 1$ ,  
then  $S = 8$  or  $9$ .

We see that for  $S = 9$

- $C_3 = 0$  or  $1$
- It can be easily seen that  $C_3 = 1$  is not possible as  $O = S + M + C_3 \Rightarrow O = 11 \Rightarrow O$  has to be assigned digit 1 but 1 is already assigned to M, so not possible.
- Therefore, only choice for  $C_3 = 0$ , and thus  $O = 10$ . This implies that O is assigned 0 (zero) digit.

Therefore,  $O = 0$

$$M = 1, O = 0$$

$C_4$	$C_3$	$C_2$	$C_1$	$\leftarrow$ Carry
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

$$\begin{aligned}
 Y &= D + E && \rightarrow C_1 \\
 E &= N + R + C_1 && \rightarrow C_2 \\
 N &= E + O + C_2 && \rightarrow C_3 \\
 O &= S + M + C_3 && \rightarrow C_4 \\
 M &= C_4
 \end{aligned}$$

3. Since  $C_3 = 0$ ;  $N = E + O + C_2$  produces no carry.

■ As  $O = 0$ ,  $N = E + C_2$ .

■ Since  $N \neq E$ , therefore,  $C_2 = 1$ .

**Hence  $N = E + 1$**

- Now  $E$  can take value from 2 to 8 {0,1,9 already assigned so far}

□ If  $E = 2$ , then  $N = 3$ .

□ Since  $C_2 = 1$ , from  $E = N + R + C_1$ , we get  $12 = N + R + C_1$

■ If  $C_1 = 0$  then  $R = 9$ , which is not possible as we are on the path with  $S = 9$

■ If  $C_1 = 1$  then  $R = 8$ , then

■ From  $Y = D + E$ , we get  $10 + Y = D + 2$ .

■ For no value of  $D$ , we can get  $Y$ .

□ Try similarly for  $E = 3, 4$ . We fail in each case.

$C_4$	$C_3$	$C_2$	$C_1$	← Carry
	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y
<hr/>				

$$Y = D + E \rightarrow C_1$$

$$E = N + R + C_1 \rightarrow C_2$$

$$N = E + O + C_2 \rightarrow C_3$$

$$O = S + M + C_3 \rightarrow C_4$$

$$M = C_4$$

**If  $E = 5$ , then  $N = 6$**

□ Since  $C_2 = 1$ , from  $E = N + R + C_1$ , we get  $15 = N + R + C_1$ ,

□ If  $C_1 = 0$  then  $R = 9$ , which is not possible as we are on the path with  $S = 9$ .

□ If  $C_1 = 1$  then  $R = 8$ , then

■ From  $Y = D + E$ , we get  $10 + Y = D + 5$  i.e.,  $5 + Y = D$ .

■ If  $Y = 2$  then  $D = 7$ . These values are possible.

Hence we get the final solution as given below and on backtracking, we may find more solutions.

**$S = 9$  ;  $E = 5$  ;  $N = 6$  ;  $D = 7$  ;  
 $M = 1$  ;  $O = 0$  ;  $R = 8$  ;  $Y = 2$**

$C_4$	$C_3$	$C_2$	$C_1$	← Carry
	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y
<hr/>				

$$Y = D + E \rightarrow C_1$$

$$E = N + R + C_1 \rightarrow C_2$$

$$N = E + O + C_2 \rightarrow C_3$$

$$O = S + M + C_3 \rightarrow C_4$$

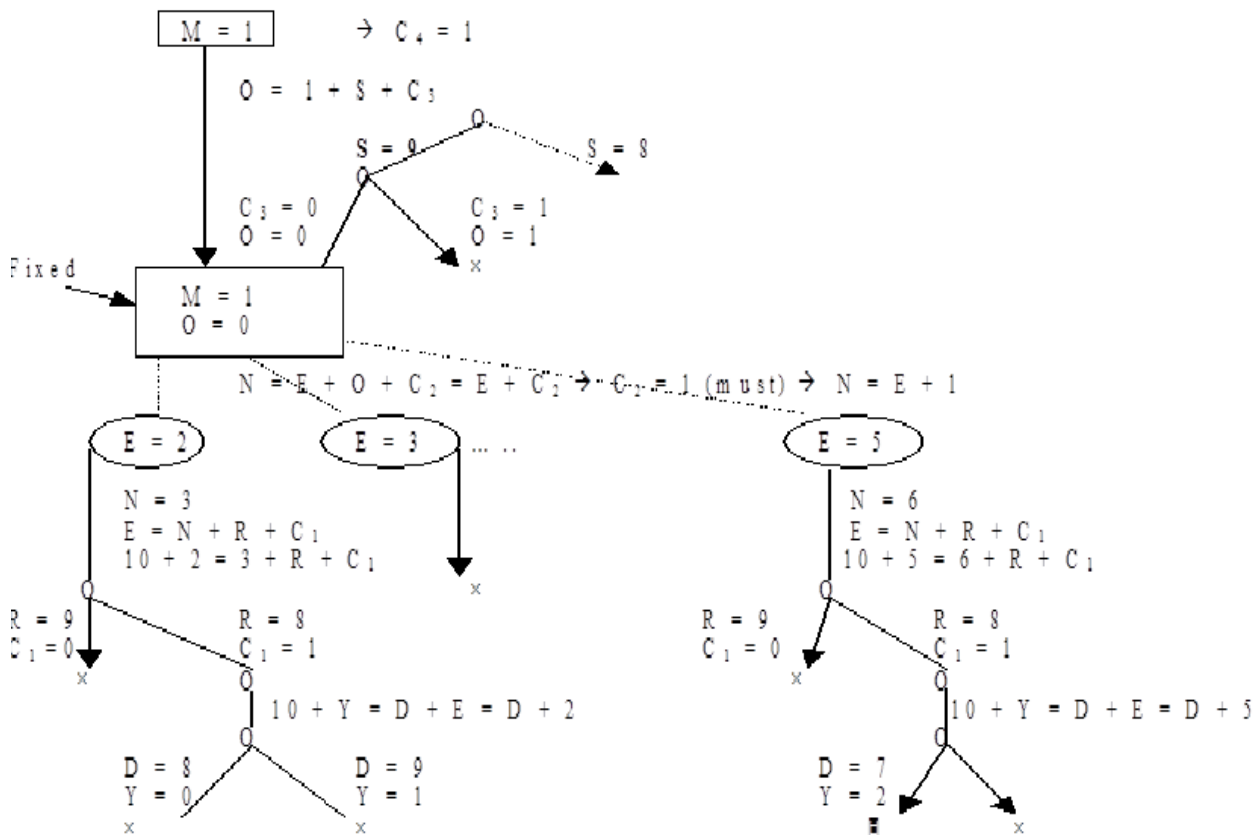
$$M = C_4$$



Constraints:

$$\begin{aligned} Y &= D + E && \longrightarrow C_1 \\ E &= N + R + C_1 && \longrightarrow C_2 \\ N &= E + O + C_2 && \longrightarrow C_3 \\ O &= S + M + C_3 && \longrightarrow C_4 \\ M &= C_4 \end{aligned}$$

Initial State



The first solution obtained is:

$$M = 1, O = 0, S = 9, E = 5, N = 6, R = 8, D = 7, Y = 2$$

CSP as a standard search problem

- A CSP can easily be expressed as a standard search problem.
- Incremental formulation
  - *Initial State*: the empty assignment  $\{\}$
  - *Successor function*: Assign a value to any unassigned variable provided that it does not violate a constraint
  - *Goal test*: the current assignment is complete  
(by construction its consistent)
  - *Path cost*: constant cost for every step (not really relevant)

- Can also use complete-state formulation
- Local search techniques (Chapter 4) tend to work well
- Solution is found at depth  $n$  (if there are  $n$  variables).
- Consider using BFS
  - Branching factor  $b$  at the top level is  $nd$
  - At next level is  $(n-1)d$
  - ....
  - end up with  $n/d^n$  leaves even though there are only  $d^n$  complete assignments!

#### Backtracking search

- Similar to Depth-first search
  - Chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
  - Uninformed algorithm
- No good general performance (see table p. 143)

```

function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-
    VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
  
```

### MEASURING PROBLEM-SOLVING PERFORMANCE:

The output of problem-solving algorithm is either failure or a solution. (Some algorithms might stuck in an infinite loop and never return an output.

The algorithm's performance can be measured in four ways :

- Completeness : Is the algorithm guaranteed to find a solution when there is one?
- Optimality : Does the strategy find the optimal solution
- Time complexity : How long does it take to find a solution?
- Space complexity : How much memory is needed to perform the search?

ANIMALAR INSTITUTE OF TECHNOLOGY  
DEPARTMENT OF INFORMATION TECHNOLOGY  
**REGULATION 2013**  
**CS6659-ARTIFICIAL INTELLIGENCE**  
**VI SEMESTER**  
**UNIT –II**

**SYLLABUS:****REPRESENTATION OF KNOWLEDGE**

Game playing - Knowledge representation, Knowledge representation using Predicate logic, Introduction to predicate calculus, Resolution, Use of predicate calculus, Knowledge representation using other logic- Structured representation of knowledge.

## 1. Introduction

Game playing has been a major topic of AI since the very beginning. Beside the attraction of the topic to people, it is also because its close relation to "intelligence", and its well-defined states and rules.

**Mathematical Game Theory**, a branch of economics, views any multiagent environment as a game provided that the impact of each agent on the other is "significant", regardless of whether the agents are cooperative or competitive. In AI, "games" are deterministic, turn-taking, two-player, zero-sum games of perfect information. This means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins the game of chess (+1), the other player necessarily loses (-1). It is this opposition between the agents' utility functions that makes the situation adversarial.

## Formal Definition of Game

We will consider games with two players, whom we will call MAX and MIN. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a search problem with the following components :

- The initial state, which includes the board position and identifies the player to move.
- A successor function, which returns a list of (*move, state*) pairs, each indicating a legal move and the resulting state.
- A terminal test, which describes when the game is over. States where the game has ended are called terminal states.
- A utility function (also called an objective function or payoff function), which give a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values +1, -1, or 0. In backgammon, the range is from +192 to -192.

**Game as Search Problem**

- **Initial State:** board position and player to move
- **Successor Function:** returns a list of legal (*move, state*) pairs
- **Terminal Test:** determines when the game is over
- **Utility function:** Gives a numeric value for the terminal state

**Game Tree**

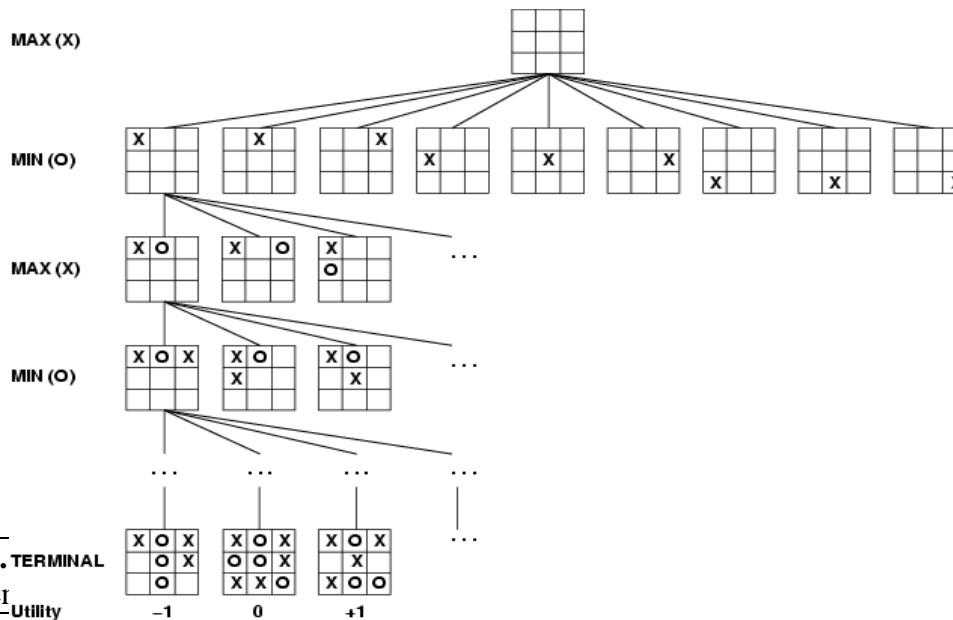
Game trees are used to represent two-player games.

Alternate moves in the game are represented by alternate levels in the tree (plies).

Nodes in the tree represent positions.

Edges between nodes represent moves.

Ex: Game Trees



Assumptions

first, placing an

In talking about game

playing systems, we make a number of assumptions:

The opponent is rational – will play to win.

The game is zero-sum – if one player wins, the other loses.

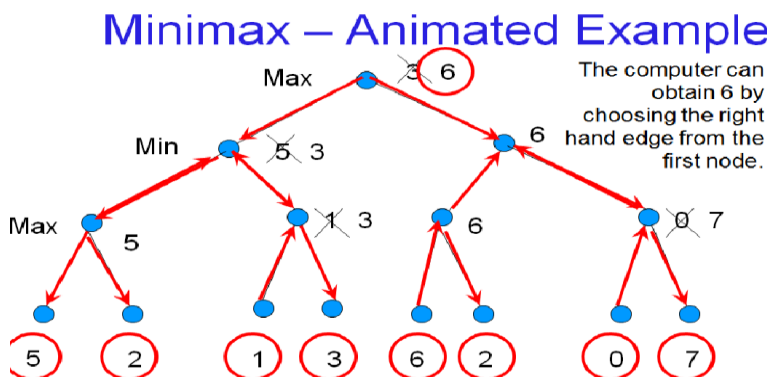
Usually, the two players have complete knowledge of the game. For games such as poker, this is clearly not true.

Minimax

Minimax is a method used to evaluate game trees.

A static evaluator is applied to leaf nodes, and values are passed back up the tree to determine the best score the computer can obtain against a rational opponent

### MINIMAX – ANIMATED EXAMPLE



## MINIMAX FUNCTION

- $\text{MINIMAX-VALUE}(n) =$ 
  - $\text{UTILITY}(n)$  if  $n$  is a terminal state
  - $\max_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s)$  if  $n$  is a MAX node
  - $\min_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s)$  if  $n$  is a MIN node

## Searching Game Trees

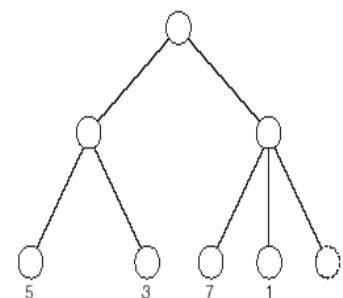
- Exhaustively searching a game tree is not usually a good idea.
- Even for a game as simple as tic-tac-toe there are over 350,000 nodes in the complete game tree.
- An additional problem is that the computer only gets to choose every other path through the tree – the opponent chooses the others

## Alpha-beta Pruning

- A method that can often cut off a half the game tree.
- Based on the idea that if a move is clearly bad, there is no need to follow the consequences of it.
- alpha – highest value we have found so far
- beta – lowest value we have found so far

## Alpha-beta Pruning – Example.

- In this tree, having examined the nodes with values 7 and 1 there is no need to examine the final node.



Alpha Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

- $\alpha$  : the value of the best(i.e.,highest-value) choice we have found so far at any choice point along the path of MAX.
- $\beta$ : the value of best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of  $\alpha$  and  $\beta$  as it goes along and prunes the remaining branches at a node(i.e.,terminates the recursive call) as soon as the value of the current node is known to be worse than the current  $\alpha$  and  $\beta$  value for MAX and MIN,respectively. The complete algorithm is given in Figure 2.21.

**function** ALPHA-BETA-SEARCH(*state*) *returns an action*

**inputs:** *state*, current state in game

$v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$

**return** the *action* in SUCCESSORS(*state*) with value *v*

---

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) *returns a utility value*

**inputs:** *state*, current state in game

$\alpha$ , the value of the best alternative for MAX along the path to *state*

$\beta$ , the value of the best alternative for MIN along the path to *state*

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

**for** *a, s* in SUCCESSORS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$

**if**  $v \geq \beta$  **then return** *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

**return** *v*

```

function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: state, current state in game
            $\alpha$ , the value of the best alternative for MAX along the path to state
            $\beta$ , the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$ 
  for  $a, s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return  $v$ 

```

**Figure 2.21** The alpha beta search algorithm. These routines are the same as the minimax routines in figure 2.20, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain  $\alpha$  and  $\beta$

### Bounded Lookahead

For trees with high depth or very high branching factor, minimax cannot be applied to the entire tree.

In such cases, bounded lookahead is applied:

- search is cut off at specified depth
- static evaluator applied.
- Horizon effect

### Knowledge

A variety of ways of knowledge(facts) have been exploited in AI programs. Facts : truths in some relevant world. These are things we want to represent.

### Knowledge Representation

- AI agents deal with knowledge (data)
  - Facts (believe & observe knowledge)
  - Procedures (how to knowledge)
  - Meaning (relate & define knowledge)
- Right representation is crucial
  - Early realisation in AI

- Wrong choice can lead to project failure
- Active research area

## Knowledge Representation using Logic

The aim of this section is to show how logic can be used to form representations of the world and how a process of inference can be used to derive new representations about the world and how these can be used by an intelligent agent to deduce what to do.

We require:

- A *formal language* to represent knowledge in a computer tractable form.
- *Reasoning* - Processes to manipulate this knowledge to deduce non-obvious facts.

Why logic?

**The challenge is to design a language which allows one to represent all the necessary knowledge. We need to be able to make statements about the world such as describing things - people, houses, theories etc; relations between things and properties of things.**

**Logic** makes statements about the world which are true (or false) if the state of affairs it represents is the case (or not the case). Compared to natural languages (expressive but context sensitive) and programming languages (good for concrete data structures but not expressive) logic combines the advantages of natural languages and formal languages.

Logic is:

- concise
- unambiguous
- context insensitive
- expressive
- effective for inferences

A logic is defined by the following:

1. **Syntax** - describes the possible configurations that constitute sentences.
2. **Semantics** - determines what facts in the world the sentences refer to i.e. the interpretation. Each sentence makes a claim about the world.
3. **Proof theory** - set of rules for generating new sentences that are necessarily true given that the old sentences are true. The relationship between sentences is called **entailment**. The semantics link these sentences (representation) to facts of the world. The proof can be used to determine new facts which follow from the old.



**Logic consists of**

- A language
  - which tells us how to build up sentences in the language (i.e., *syntax*), and
  - and what the sentences mean (i.e., *semantics*)
- An inference procedure
  - which tells us which sentences are valid inferences from other sentences

**Propositional logic**

The symbols of propositional calculus are

the propositional symbols:

P, Q, R, S,

the truth symbols:

true, false

and connectives:

$\wedge, \vee, \neg, \rightarrow, \equiv$

**Propositional Calculus Sentences**

- Every propositional symbol and truth symbol is a *sentence*.
  - Examples: true, P, Q, R.
- The *negation* of a sentence is a sentence.
  - Examples:  $\neg P$ ,  $\neg$  false.
- The *conjunction*, or *and*, of two sentences is a sentence.
  - Example:  $P \wedge \neg P$
- The *disjunction*, or *or*, of two sentences is a sentence.
  - Example:  $P \vee \neg P$
- The *implication* of one sentence from another is a sentence.
  - Example:  $P \rightarrow Q$
- The *equivalence* of two sentences is a sentence.
  - Example:  $P \vee Q \equiv R$
- Legal sentences are also called well-formed formulas or *WFFs*.

**Propositional calculus semantics**

An *interpretation* of a set of propositions is the assignment of a truth value, either T or F to each propositional symbol.

The symbol true is always assigned T, and the symbol false is assigned F.

The truth assignment of *negation*,  $\neg P$ , where P is any propositional symbol, is F if the assignment to P is T, and is T if the assignment to P is F.

The truth assignment of *conjunction*,  $\wedge$ , is T only when both conjuncts have truth value T; otherwise it is F.

The truth assignment of *disjunction*,  $\vee$ , is F only when both disjuncts have truth value F; otherwise it is T.

The truth assignment of *implication*,  $\rightarrow$ , is F only when the premise or symbol before the implication is T and the truth value of the consequent or symbol after the implication F; otherwise it is T.

The truth assignment of *equivalence*,  $\equiv$ , is T only when both expressions have the same truth assignment for all possible interpretations; otherwise it is F.

### **FOR PROPOSITIONAL EXPRESSIONS P, Q, R**

$$\neg(\neg P) \equiv P$$

$$(P \vee Q) \equiv (\neg P \rightarrow Q)$$

$$\text{the contrapositive law: } (P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$$

$$\text{de Morgan's law: } \neg(P \vee Q) \equiv (\neg P \wedge \neg Q) \text{ and } \neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$$

$$\text{the commutative laws: } (P \wedge Q) \equiv (Q \wedge P) \text{ and } (P \vee Q) \equiv (Q \vee P)$$

$$\text{the associative law: } ((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$$

$$\text{the associative law: } ((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$$

$$\text{the distributive law: } P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$

$$\text{the distributive law: } P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

### **Truth table for the operator $\wedge$**

P	Q	$P \wedge Q$
T	T	T
T	F	F
T	T	F
T	F	F

**Truth table demonstrating the equivalence of  $\neg P \vee Q$  and  $P \rightarrow Q$**

P	Q	$\neg P$	$\neg P \vee Q$	$P \Rightarrow Q$	$(\neg P \vee Q) = (P \Rightarrow Q)$
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

**Proofs in propositional calculus**

If it is sunny today, then the sun shines on the screen. If the sun shines on the screen, the blinds are brought down. The blinds are not down.

Is it sunny today?

P: It is sunny today.

Q: The sun shines on the screen.

R: The blinds are down.

Premises:  $P \rightarrow Q$ ,  $Q \rightarrow R$ ,  $\neg R$

Question: P

Represent in predicate calculus

If it is sunny on a particular day, then the sun shines on the screen [on that day]. If the sun shines on the screen on a particular day, the blinds are down [on that day]. The blinds are not down today.

Is it sunny today?

Premises:

$\forall D \text{ sunny}(D) \rightarrow \text{screen-shines}(D)$

$\forall D \text{ screen-shines}(D) \rightarrow \text{blinds-down}(D)$

$\neg \text{blinds-down}(\text{today})$

Question:  $\text{sunny}(\text{today})$

**Predicate calculus symbols**

The set of letters (both uppercase and lowercase): A ... Z, a ... z.

The set of digits: 0 ... 9

The underscore: \_

Note: Needs to start with a letter.

**Symbols and terms**

1. **Truth symbols** true and false (these are reserved symbols)

2. **Constant symbols** are symbol expressions having the first character lowercase.

E.g., today, fisher

3. **Variable symbols** are symbol expressions beginning with an uppercase character.

E.g., X, Y, Z, Building

4. **Function symbols** are symbol expressions having the first character lowercase.

Arity: number of elements in the domain

E.g., mother-of (bill); maximum-of (7,8)

A *function expression* consists of a function constant of arity  $n$ , followed by  $n$  terms,  $t_1, t_2, \dots, t_n$ , enclosed in parentheses and separated by commas.

E.g., mother-of(mother-of(joe))

maximum(maximum(7, 18), add\_one(18))

A *term* is either a constant, variable, or function expression.

E.g. color\_of(house\_of(neighbor(joe)))

house\_of(X)

### Predicates and atomic sentences

*Predicate symbols* are symbols beginning with a lowercase letter. Predicates are special functions with true/false as the range.

Arity: number of arguments

An *atomic sentence* is a predicate constant of arity  $n$ , followed by  $n$  terms,  $t_1, t_2, \dots, t_n$ , enclosed in parentheses and separated by commas.

The truth values, true and false, are also atomic sentences.

### Predicate calculus sentences

Every atomic sentence is a sentence.

1. If  $s$  is a sentence, then so is its negation,  $\neg s$ .

If  $s_1$  and  $s_2$  are sentences, then so is their

2. Conjunction,  $s_1 \wedge s_2$ .

3. Disjunction,  $s_1 \vee s_2$ .

4. Implication,  $s_1 \rightarrow s_2$ .

5. Equivalence,  $s_1 \equiv s_2$ .

If  $X$  is a variable and  $s$  is a sentence, then so are

6.  $\forall X s$ .

7.  $\exists X s$ .

### EXAMPLE:

A simple logic statement is "If  $P$ , then  $Q$ ". For example, "If it rains for an hour, then the ground will be wet". Note that there are two logic errors that people often make from this:

1) "If Q, then P". In our example, this would mean "If the ground is wet, then it has been raining for an hour". However, the ground could be wet for other reasons, like if snow has melted or a water main has broken.

2) "If not P, then not Q". In our example, this would mean "If it has not been raining for an hour, then the ground is not wet". However, the ground could still be wet for the reasons listed above.

The only correct inference that can be made is "If not Q, then not P". This means, in our example, "If the ground is not wet, then it has not been raining for an hour".

### Order of quantifiers matters

- **Everybody likes some food.**
- **There is a food that everyone likes.**
- **Whenever someone likes at least one spicy dish, they're happy.**
  
- **Everybody likes some food.**  
 $\forall X \exists F \text{ food}(F) \wedge \text{likes}(X, F)$
- **There is a food that everyone likes.**  
 $\exists F \forall X \text{ food}(F) \wedge \text{likes}(X, F)$
- **Whenever someone eats a spicy dish, they're happy.**  
 $\forall X \exists F \text{ food}(F) \wedge \text{spicy}(F) \wedge \text{eats}(X, F) \rightarrow \text{happy}(X)$

### **Examples**

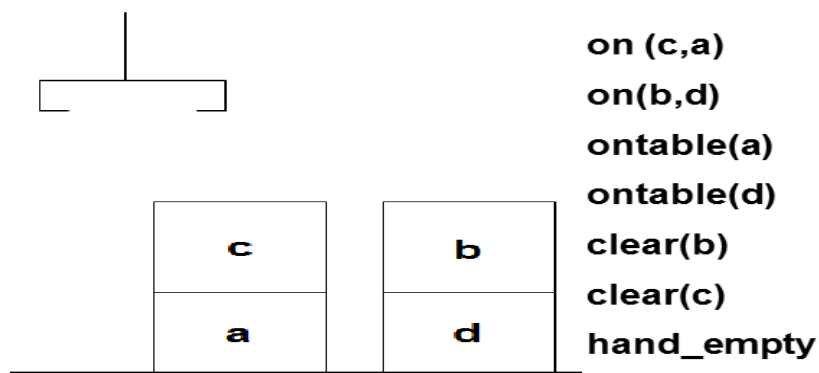
**John's meals are spicy.**

$\forall X \text{ meal-of}(\text{John}, X) \rightarrow \text{spicy}(X)$

**Every city has a dogcatcher who has been bitten by every dog in town.**

$\forall C \exists D \forall Z \text{ city}(C) \rightarrow (\text{dogcatcher}(D, C) \wedge$   
 $(\text{dog}(Z) \wedge \text{lives-in}(Z, C) \rightarrow \text{bit}(Z, D)))$

### Blocks world problem



**All blocks on top of blocks that have been moved or that are attached to blocks that have been moved have also been moved.**

$\forall X \forall Y (\text{block}(X) \wedge \text{block}(Y) \wedge$   
 $(\text{on}(X,Y) \vee \text{attached}(X,Y)) \wedge \text{moved}(Y)) \rightarrow$   
 $\text{moved}(X)$

## Quantifiers in Predicate Calculus

$P(x)$  represents a property of objects in a given set (the domain of  $x$ :  $D$ )

Naturally we are interested if there are objects in  $D$  for which  $P(x)$  is true.

To express the truth value of a predicate with respect to a set of objects, we use **quantifiers**. Quantifiers refer to quantities such as some, all, none. They tell us for how many elements a given predicate is true.

Here 'how many' is not a particular number. A quantified statement tells us if the predicate is true for all the elements in the set, or for some of the elements, or for none of the elements.

### 1. Universal quantifiers (" )

Consider the statement: All human beings are mortal.

Here, the property "being mortal" refers to **all** human beings.

The symbol " is used to denote the quantity "all" of objects for which a given predicate is true. Since it represents quantity, it is called **quantifier**. Since the quantity is "all" (the property is true for all objects) it is called **universal quantifier**.

We write:

$\forall x \in S, x \text{ is mortal, or } \forall x \in S \text{ mortal}(x)$   
 where  $S$  is the set of all human beings.

In English the quantity **all** can be represented in several ways: "for all", "all", "each", "every".

When we **quantify universally** a predicate, we **get a statement** (can be true or false)

**Definition:** A **universal statement** is a statement of the form

$\forall x \in D, P(x) \text{ or } \forall x, D(x) \rightarrow P(x)$

It is defined to be **true if and only if  $P(x)$  is true for every  $x$  in  $D$ .**

It is **false if and only if there exist at least one element in  $D$ , for which  $P(x)$  is false.**

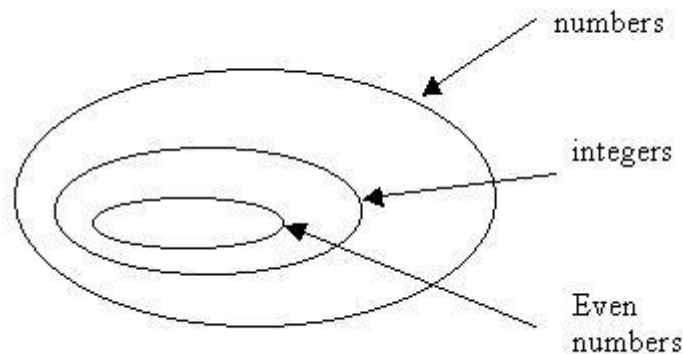
**Including the domain in the expression:**  $\forall x, D(x) \rightarrow P(x)$

For all  $x$ , if  $D(x)$  is true, i.e.  $x$  is in  $D$ , then  $P(x)$  is also true, i.e.  $x$  has the property  $P$ .

**Important:** How to understand the statement  $\forall x, P(x) \rightarrow Q(x)$

The first predicate  $P(x)$  defines some set of objects, e.g. even numbers, the second predicate  $Q(x)$  states a property for all objects in this set, e.g. integers - (all even numbers are integers)

The second predicate by itself defines a larger set, that contains the first set, e.g. integers contain all prime numbers, they also contain other numbers that are not even.



## 2. Existential quantifiers ( $\exists$ )

Consider the statements:

Some people are students at Simpson College.  
 There are some students at Simpson.  
 There exists at least one student at Simpson College.

All these statements are equivalent in that they claim the existence of at least one object for which the predicate  $\text{simpson\_student}(x)$  is true.

The symbol  $\exists$  is used to denote the quantity "at least one" of objects for which a given predicate is true.

Since it claims the existence of an object with a given property, it is called **existential quantifier**.

We write:

$\exists$  a person  $s$ , such that  $s$  is a student in Simpson College.

or:

$\exists s \in S$ , such that  $Q(s)$ ,  
 where  $S$  is the set of all people,  $Q(s)$  is "being a student in Simpson College."

In English the meaning of the existential quantifier is contained in the words *there is, we can find, there is at least one, for some, some, for at least one, at least one*

**Definition:** Existential statement is a statement of the form:

$$\begin{aligned} & \exists x \in D, \text{ such that } Q(x), \\ & \exists x \in D, Q(x) \text{ (more compact form)} \\ & \exists x, D(x) \wedge Q(x) \text{ (domain is included in the expression)} \end{aligned}$$

(The words "such that" may be omitted when we write the statement, but should be pronounced when we read the statement.)

It is defined to be **true if and only if**  $Q(x)$  is true for at least one element in  $D$ .

It is **false if and only if**  $Q(x)$  is false for all elements in  $D$ .

**Including the domain in the expression:**  $\exists x, D(x) \wedge Q(x)$

There is some  $x$  in  $D$  with the property  $Q$ , i.e.  $D(x)$  is true and  $Q(x)$  is true.

### 3. Implicit quantification

Consider the statements:

1. Prime numbers are integers.
2. If a number is a prime number then it is an integer.
3. All prime numbers are integers
4. Any prime number is an integer

These statements are universally quantified statements. (1) and (2) are implicitly quantified - using plural without a determiner, or singular with indefinite article "a".

If we use determiners (*the, this, these*, etc) the meaning (in English) is not "all". Instead we refer to some particular object(s).

For example, if we say: "This file is corrupted", we refer to a particular file, and then in logic the statement would be e.g.

$\text{corrupted}(\text{my\_file})$ .

We can write also

$\exists x, \text{file}(x) \wedge \text{corrupted}(x)$

### 4. Exercise

Consider the following statement: "All basketball players are tall".

This statement uses the following predicates:



basketball\_player(x)  
tall(x)

In predicate logic the statement can be represented as:

**(1)  $\forall x, \text{basketball\_player}(x) \rightarrow \text{tall}(x)$ .**

Consider now the following sentences and determine which of them are equivalent to (1):

**a. Every basketball player is tall:**

$\forall x, \text{basketball\_player}(x) \rightarrow \text{tall}(x)$

This is equivalent to (1).

**b. Among all the basketball players, some are tall:**

$\exists x, \text{basketball\_player}(x) \wedge \text{tall}(x)$ .

This sentence is **not equivalent to (1)**.

It claims the property "tall" only for some of the basketball players, so we may assume that there are basketball players that are not tall.

**c. Some of all the tall people are basketball players:**

$\exists x, \text{tall}(x) \wedge \text{basketball\_player}(x)$

This sentence is **not equivalent to (1)**.

It only claims that some persons are both tall and basketball players, however there may be basketball players that are not tall.

**d. Anyone who is tall is a basketball player:**

$\forall x, \text{tall}(x) \rightarrow \text{basketball\_player}(x)$ .

This sentence is **not equivalent to (1)**.

It claims that all tall persons are basketball players, while (1) does not say anything about the other tall persons - so according to (1) there may be other tall persons that are not basketball players.

**e. All people who are basketball players are tall:**

$\forall x, \text{basketball\_player}(x) \rightarrow \text{tall}(x)$ .

This is equivalent to (1)

f. Anyone who is a basketball player is a tall person.

"  $x$ , basketball\_player( $x$ )  $\rightarrow$  tall( $x$ )

This is equivalent to (1). Here the universal quantifier is expressed by the word "anyone"

## Structured Knowledge Representation

### Kinds of Knowledge

Cognitive Psychologists identify different categories of knowledge:

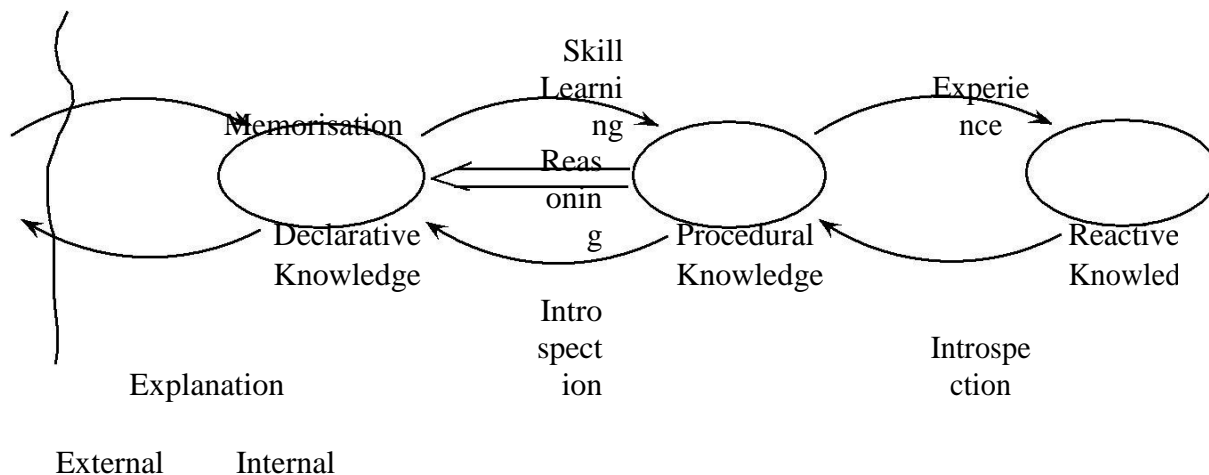
**Declarative:** A symbolic expression of competence. Declarative knowledge is abstract

Declarative knowledge is used to communicate and to reason.

**Procedural:** A series of steps to solve a problem.

A compiled expression of knowledge

**Reactive:** stimulus - response.



Physical Symbol Systems are a form of Declarative representation.

Declarative representations are useful for Communication and for reasoning about knowledge (meta-knowledge).

Structure Knowledge Representations were explored as a general representation for symbolic representation of declarative knowledge. One of the results was a theory for Schema Systems

## Structured Knowledge Representation

### Schema Systems

The concept for Schema dates back to I. Kant (In Critique of Pure Reason (Kritik der reinen Vernunft) 1781.

Kant wanted to show that not all truths are grounded in empirical observation.

Kant proposed that knowledge was grounded (based) on perception. He proposed the use of "schemata" as a means of organizing and interpreting perceptual phenomena.

Kant distinguished "Phenomena" from "Noumena". Noumena is the underlying unknowable reality of the universe. Noumena can never be completely known, but only partially observed through the senses.

Phenomena provide partial observation of Noumena. Thus Phenomena are necessarily imperfect.

Kant proposed that schemata are constructed to represent "typical" or prototype phenomena. Prototypes then are used to represent the entire set of observed phenomena for interpreting and organizing perceptions.

Kant's approach was popular in Cognitive Psychology in the 1930's and were adopted by early computer science researchers. Popular programming systems based on Schema included application in:

Computer Vision: Frames

Story Understanding: Scripts

Context Aware Systems: Situation Models,

Language translation and understanding: Semantic Nets

These tools can be unified under the term "Schema Systems". The key idea in all of these is to express relations between entities.

Entity: literals, constants, observables, primitives.

### Relations as N-Ary Predicates

Relations are a key concept in structured knowledge representations.

Relations express structure of entities.

## Structured Knowledge Representation

Examples include temporal relations, spatial relations, Family relations, Social relations, administrative

organizations, military hierarchies, etc

Relations are formalized as N-Ary Predicates.

The valence or Arity of a relation is the number of entities that it associates.

Unary or Monadic: Man(Bob)

Binary or dyadique: Brother(Charlie, Bob)

Ternary or triadique: Action(Jim, Talks-To, Bob)

Unary relations represent properties for entities

Binary relations associate entities.

Examples From Blocks world Arity0 :

(HandEmpty) Unary : (OnTable A)

Binary: (On A B)

Ternary: (Over A B C) ;; Block A is a bridge over B et C.

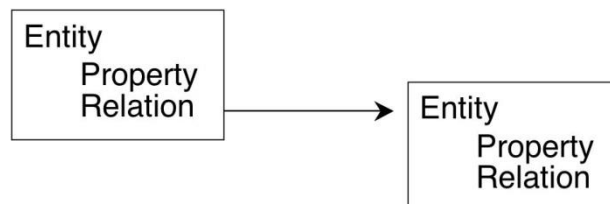
A symbol is a triadic relation between a sign, a thing and a agent. The agent interprets the sign to represent the thing.

```
(defclass symbol (is-a USER) (slot signe)
  (slot chose) (slot agent)
  )
```

Relations can be represented "implicitly" or "explicitly".

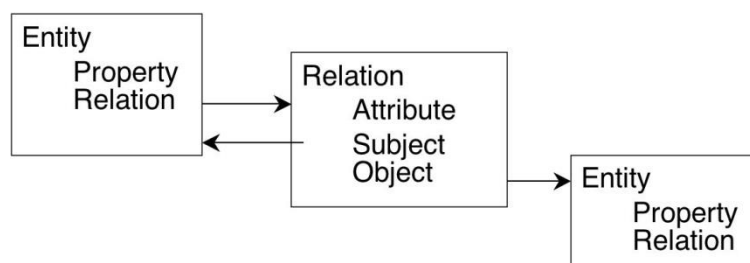
In an implicit representation, the thing is represented as a value of a slot. In an explicit representation, the ternary structure is expressed as a schema.

Implicit representation:



Explicit:

### Structured Knowledge Representation



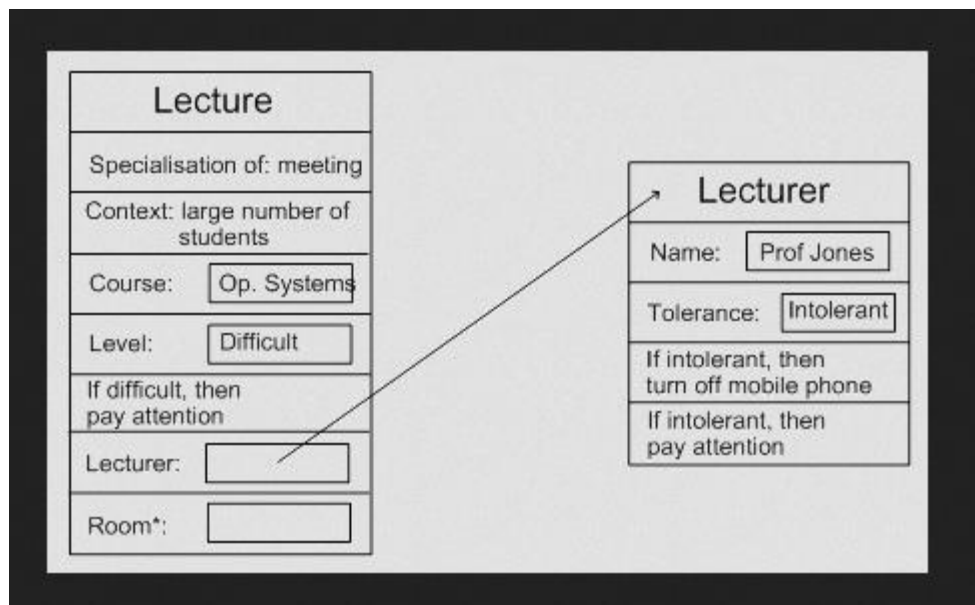
Implicit representations are simpler and slightly more efficient in computing and memory.

Explicit representations allows a system to reason about relations.

This is yet another example of the power of representing program as data.

## Frames

- Frame: A Structured Representation to provide context for focusing visual interpretation of scenes.
- A frame is an [artificial intelligence data structure](#) used to divide [knowledge](#) into substructures by representing "[stereotyped](#) situations." Frames are the primary data structure used in artificial intelligence [Frame languages](#).
- Frames are also an extensive part of [knowledge representation and reasoning](#) schemes
- Frames were originally derived from semantic networks and are therefore part of structure based knowledge representations.



A Frame is composed of a set of "slots" and "methods".

A slot is a named place-holder for a pointer. The slots point to other frames that represent entities that are described (or interpreted) by the frame. Ultimately, some slots point to raw perceptions. When a slot points to an entity it is said to play a "role" in the frame. Frames typically come with methods (procedures) for searching for the entities that can play roles in the frame.

Frames can be formalized as a set of relations between entities having certain properties.

## Scripts

Schank and Abelson Scripts, Plans, Goals and Understanding, Erlbaum, 1977.

A script is a data structure used to represent a sequence of events. Scripts are used for interpreting stories. Popular examples have been script driven systems that can interpret and extract facts from Newspaper Stories.

Scripts have been used to

- 1) Interpret, understand and reason about stories,
- 2) Understand and reason about observed events
- 3) Reason about observed actions
- 4) Plan actions to accomplish tasks.

A script is composed of

- 1) A scene (division)
- 2) Props (objects manipulated in the script)
- 3) The actors (agents that can change the state of the world).
- 4) Events
- 5) Acts: A set of actions by the actors.

In each scene, one or more actors perform actions. The actors act with the props. The script can be represented as a tree or network of states, driven by events.

As with Frames, scripts drive interpretation by telling the system what to look for and where to look next. The script can predict events.

### *Example of a script*

The classic example is the restaurant script:

Scene: A restaurant with an entrance and tables. Actors: The diners, servers, chef and Maitred'Hotel. Props: The table setting, menu, table, chair.

Acts: Entry, Seating, Ordering a meal, Serving a meal, Eating the meal, requesting the check, paying, leaving.

Schema systems capture Frames, Scripts, and semantic network as networks of schema. Typically, schema represent relations between entities playing roles:

Structured Knowledge Representation

### **Situation Models**

P. Johnson-Laird 1983 - Mental Models.

Situations models are used in cognitive psychology to express the mental models that people use to understand.

Situation: Relations between entities

Entities: People and things;

Relations: An N-ary predicate ( $N=1,2,3 \dots$ )

Example: John is facing Mary. John is talking to Mary.

Situation models can be easily expressed as schema.

A situation graph describes a state space of situations

A Situation determines:

System Attention: entities and relations for the system to observe System Behaviours:

List of actions that are allowed or forbidden Default Values: Expectations for entities, relations, and properties

Behaviours include methods for perception, and methods for interaction with the external world

Each situation indicates:

Transition probabilities for accessible situations

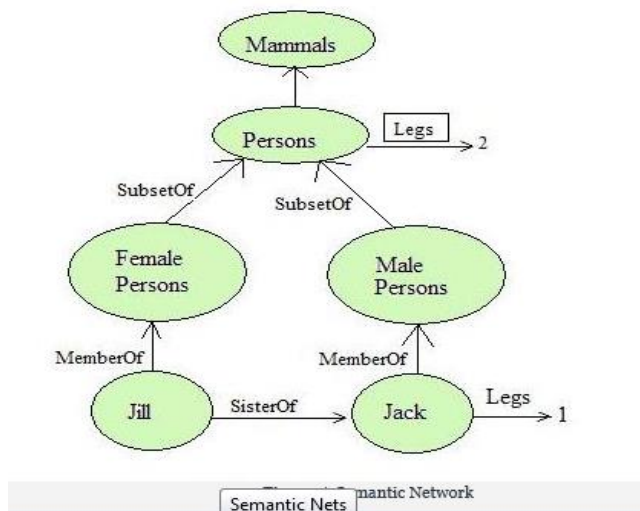
The appropriateness or inappropriateness of actions.

Structured Knowledge Representation

### **Semantic Nets**

A **semantic net** (or semantic network) is a [knowledge representation technique](#) used for propositional information. So it is also called a propositional net. Semantic nets convey meaning. They are two dimensional representations of [knowledge](#). Mathematically a *semantic net* can be defined as a labelled directed graph.

Semantic nets consist of nodes, links (edges) and link labels. In the semantic network diagram, nodes appear as circles or ellipses or rectangles to represent objects such as physical objects, concepts or situations. Links appear as arrows to express the relationships between objects, and link labels specify particular relations. Relationships provide the basic structure for organizing [knowledge](#). The objects and relations involved need not be so concrete. As nodes are associated with other nodes semantic nets are also referred to as associative nets.



In the above figure all the objects are within ovals and connected using labelled arcs. Note that there is a link between *Jill* and *FemalePersons* with label *MemberOf*. Similarly there is a *MemberOf* link between *Jack* and *MalePersons* and *SisterOf* link between *Jill* and *Jack*. The *MemberOf* link between *Jill* and *FemalePersons* indicates that *Jill* belongs to the category of female persons.

#### Inheritance Reasoning

Unless there is a specific evidence to the contrary, it is assumed that all members of a class (category) will inherit all the properties of their superclasses. So *semantic network* allows us to perform inheritance reasoning. For example *Jill* inherits the property of having two legs as she belongs to the category of *FemalePersons* which in turn belongs to the category of *Persons* which has a boxed *Legs* link with value 2. Semantic nets allows multiple inheritance. So an object can belong to more than one category and a category can be a subset of more than one another category.

#### Inverse links

Semantic network allows a common form of inference known as inverse links. For example we can have a *HasSister* link which is the inverse of *SisterOf* link. The inverse links make the job of inference algorithms



much easier to answer queries such as who the sister of *Jack* is. On discovering that *HasSister* is the inverse of *SisterOf* the inference algorithm can follow that link *HasSister* from *Jack* to *Jill* and answer the query.

#### Disadvantage of Semantic Nets

One of the drawbacks of semantic network is that the links between the objects represent only binary relations. For example, the sentence *Run(ChennaiExpress, Chennai,Bangalore,Today)* cannot be asserted directly.

There is no standard definition of link names.

#### Advantages of Semantic Nets

Semantic nets have the ability to represent default values for categories. In the above figure Jack has one leg while he is a person and all persons have two legs. So persons have two legs has only default status which can be overridden by a specific value.

## KNOWLEDGE INFERENCE

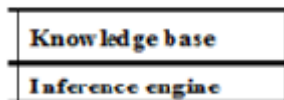
Knowledge representation -Production based system,Frame based system. Inference -Backward chaining,Forwardchaining,Rulevalueapproach,Fuzzyreasoning-Certainty factors,Bayesian Theory-BayesianNetwork-Dempster Shafertheory.

Inference(interpretation)

### Knowledge Representation is divided into 2 types:

- I. Production systems.
- II. Frame-based systems.

### Knowledge-based system



- **Knowledge base:**
  - A set of sentences that describe the world in some formal (representational) language (e.g. first-order logic)
  - Domain specific knowledge
- **Inference engine:**
  - A set of procedures that work upon the representational language and can infer new facts or answer KB queries (e.g. resolution algorithm, forward chaining)
  - Domain independent

### **Automated reasoning systems**

- **Theorem provers**

– Prove sentences in the first-order logic. Use inference rules, resolution rule and resolution refutation.

- **Deductive retrieval systems**

- Systems based on rules (KBs in Horn form)
- Prove theorems or infer new assertions

- **Production systems**

- Systems based on rules with actions in antecedents
- Forward chaining mode of operation

- **Semantic networks**

- Graphical representation of the world, objects are nodes in the graphs, relations are various links

- **Frames:**

- object oriented representation, some procedural control of inference

### **PRODUCTION BASED SYSTEMS**

Based on rules, but different from KBs in the Horn form Knowledge base is divided into:

- A Rule base (includes rules)
- A Working memory (includes facts)

**Rules: a special type of if – then rule**  $p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow a_1, a_2, \dots, a_k$

Antecedent	Consequent
A conjunction of condition	a sequence of action

#### **Basic operation:**

- Check if the antecedent of a rule is satisfied
- Decide which rule to execute (if more than one rule is satisfied)
- Execute actions in the consequent of the rule

#### **Working memory**

- Consists of a set of facts – statements about the world but also can represent various data structures
- The exact syntax and representation of facts may differ across different systems

- **Examples:**

- **predicates**

- such as Red(car12)
- but only ground statements

**or**

- **(type attr1:value1 attr2:value2 ...) objects** such as: (person age 27 home Toronto)

The type, attributes and values are all atoms

**Rules**

$$p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow a_1, a_2, \dots, a_k$$

- **Antecedents: conjunctions of conditions**

- **Examples:**

- a conjunction of literals  $A(x) \wedge B(x) \wedge C(y)$
- simple negated or non-negated statements in predicate logic

- or

- conjunctions of conditions on objects/object
- (type attr1 spec1 attr2 spec2 ...)
- Where specs can be an atom, a variable, expression, condition (person age  $[n+4]$  occupation  $x$ )
- (person age  $\{< 23 \wedge > 6\}$ )

**Production systems**

$$p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow a_1, a_2, \dots, a_k$$

- **Consequent:** a sequence of actions
- An action can be:
  - **ADD** the fact to the working memory (WM)
  - **REMOVE** the fact from the WM
  - **MODIFY** an attribute field
  - **QUERY** the user for input, etc ...

- **Examples:**

$$A(x) \wedge B(x) \wedge C(y) \Rightarrow \text{add } D(x)$$

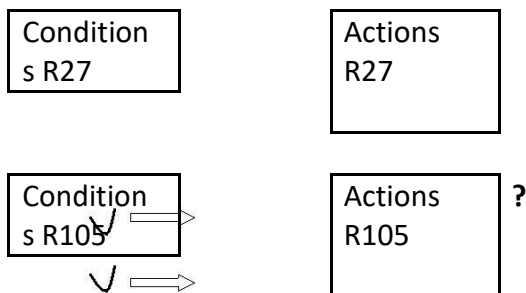
- **Or**

$$(\text{Student name } x) \Rightarrow \text{ADD (Person name } x)$$

- Use **forward chaining to do reasoning:**

- If the antecedent of the rule is satisfied (rule is said to be “active”) then its consequent can be executed (it is “fired”)

- **Problem:** Two or more rules are active at the same time. Which one to execute next?



- Strategy for selecting the rule to be fired from among possible candidates is called **conflict resolution**

- Why is conflict resolution important? Or, Why do we care about the order?
- Assume that we have two rules and the preconditions of both are satisfied:

**R1:**  $A(x) \wedge B(x) \wedge C(y) \Rightarrow \text{add } D(x)$

**R2:**  $A(x) \wedge B(x) \wedge E(z) \Rightarrow \text{delete } A(x)$

- What can happen if rules are triggered in different order?
- Why is conflict resolution important? Or, Why do we care about the order?
- Assume that we have two rules and the preconditions of both are satisfied:

**R1:**  $A(x) \wedge B(x) \wedge C(y) \Rightarrow \text{add } D(x)$

**R2:**  $A(x) \wedge B(x) \wedge E(z) \Rightarrow \text{delete } A(x)$

- What can happen if rules are triggered in different order?
  - If R1 goes first, R2 condition is still satisfied and we infer  $D(x)$
  - If R2 goes first we may never infer  $D(x)$
- **Problems with production systems:**
  - Additions and Deletions can change a set of active rules;
  - If a rule contains variables testing all instances in which the rule is active may require a large number of unifications.
  - Conditions of many rules may overlap, thus requiring to repeat the same unifications multiple times.
- **Solution: Rete algorithm**
  - gives more efficient solution for managing a set of active rules and performing unifications
  - Implemented in the system **OPS-5** (used to implement XCON – an expert system for configuration of DEC computers)

### Rete algorithm

- Assume a set of rules:

$A(x) \wedge B(x) \wedge C(y) \Rightarrow \text{add } D(x)$

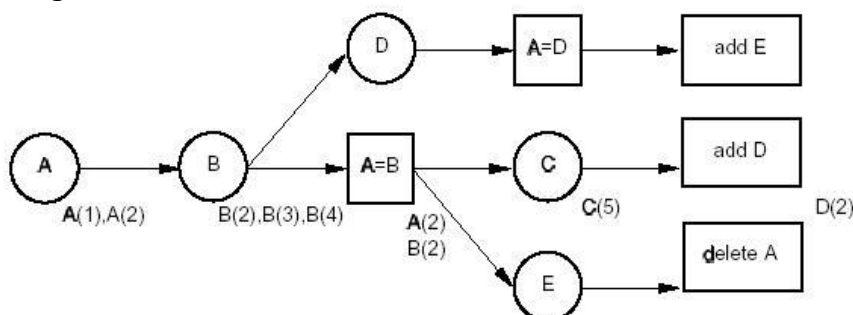
$A(x) \wedge B(y) \wedge D(x) \Rightarrow \text{add } E(x)$   $A(x) \wedge B(x) \wedge E(z) \Rightarrow \text{delete } A(x)$

• And facts:  $A(1), A(2), B(2), B(3), B(4), C(5)$

### • Rete:

- Compiles the rules to a network that merges conditions of multiple rules together (avoid repeats)
- Propagates valid unifications
- Reevaluates only changed conditions

### Rete algorithm.Network.



Rules:	$A(x) \wedge B(x) \wedge C(y) \Rightarrow add$	$D(x)$	
	$A(x) \wedge B(y) \wedge D(x) \Rightarrow add$	$E(x)$	
	$A(x) \wedge B(x) \wedge E(z) \Rightarrow delete$	$A(x)$	
Facts:	$A(1), A(2), B(2), B(3), B(4), C(5)$		

### Conflict resolution strategies

- **Problem:** Two or more rules are active at the same time. Which one to execute next?
- **Solutions:**
  - **No duplication** (do not execute the same rule twice)
  - **Regency.** Rules referring to facts newly added to the working memory take precedence
  - **Specificity.** Rules that are more specific are preferred.
  - **Priority levels.** Define priority of rules, actions based on expert opinion. Have multiple priority levels such that the higher priority rules fire first.

### FRAME-BASED REPRESENTATION

- Frames – semantic net with properties
- A frame represents an entity as a set of slots (attributes) and associated values
- A frame can represent a specific entry, or a general concept
- Frames are implicitly associated with one another because the value of a slot can be another frame

### 3 components of a frame

- frame name
- attributes (slots)
- values (fillers: list of values, range, string, etc.)

Book Frame
Slot → <i>Filler</i>

- Title → **AI. A modern Approach**
- Author → **Russell & Norvig**
- Year → **2003**

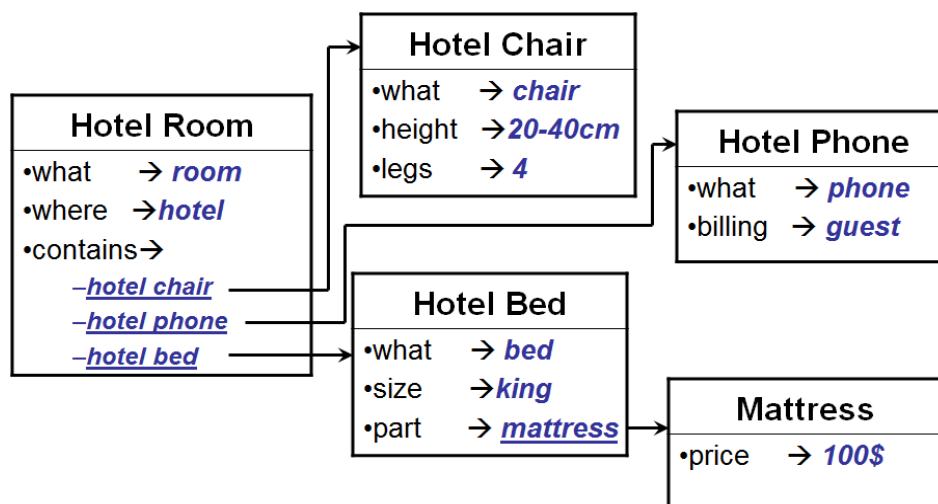
### Features of Frame Representation

- More natural support of values than semantic nets (each slots has constraints describing legal values that a slot can take)
- Can be easily implemented using object-oriented programming techniques
- Inheritance is easily controlled

### Inheritance

## Inheritance

- Similar to Object-Oriented programming paradigm



### Benefits of Frames

- Makes programming easier by grouping related knowledge
- Easily understood by non-developers
- Expressive power
- Easy to set up slots for new properties and relations

- Easy to include default information and detect missing values

### 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.

- 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 :

"					
→	" (implication),	" ¬ "	(not),	" ∨ " (or),	" ∧ " (and),
"		" ∃ "			
∀	" (for all),	"	(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 :  $\forall x : \text{animal}(x) \rightarrow (\text{live}(x, \text{land}) \vee \text{live}(x, \text{water}))$

### 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 exist 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 nounphrase may contain articles, adjectives and nouns. It thus accordingly call routines that know how to process articles, adjectives and nouns.

### ■ Inference Procedural Rules



These rules advise on how to solve a problem, while certain facts are known.

Example :

IF the data needed is not in the system THEN request it from the user.

These rules are part of the inference engine.

The logic model of computation is based on relations and logical inference.

Programs consist of definitions of relations. Computations are inferences (is a proof).

#### **Example 1 :Linear function**

A linear function  $y = 2x + 3$  can be represented as :

**f (X , Y) if Y is  $2 * X + 3$ .**

Here the function represents the relation between X and Y.

#### **Example 2: Determine a value for Circumference.**

The circumference computation can be represented as:

**Circle (R , C) if  $Pi = 3.14$  and  $C = 2 * pi * R$ .**

Here the function is represented as the relation between radius R and circumference C.

#### **Forward versus Backward Reasoning**

Rule-Based system architecture consists a *set of rules*, a *set of facts*, and an *inference engine*. The need is to find what new facts can be derived. Given a set of rules, there are essentially two ways to generate new Knowledge: one, forward chaining and the other, backward chaining.

**Forward chaining** : also called data driven.

It starts with the facts, and sees what rules apply.

**Backward chaining** : also called goal driven.

It starts with something to find out, and looks for rules that will help in answering it.

**KR – forward-backward reasoning**

#### **Example 1**

Rule	R1 :	IF hot AND smoky	THEN fire
Rule	R2 :	IF alarm_beeps	THEN smoky
Rule	R3 :	IF fire	THEN switch_on_sprinklers
Fact	F1 :	alarm_beeps [Given]	

Fact F2 : hot [Given]

### Example 2

Rule R1 : IF hot AND smoky THEN ADD fire  
 Rule R2 : IF alarm\_beeps THEN ADD smoky  
 Rule R3 : IF fire THEN ADD switch\_on\_sprinklers

Fact F1 : alarm\_beeps [Given]

Fact F2 : hot [Given]

### *KR – forward-backward reasoning*

#### Example 3 : A typical Forward Chaining

Rule R1 : IF hot AND smoky THEN ADD fire  
 Rule R2 : IF alarm\_beeps THEN ADD smoky  
 Rule R3 : If fire THEN ADD switch\_on\_sprinklers

Fact F1 : alarm\_beeps [Given]

Fact F2 : hot [Given]

Fact F4 : smoky [from F1 by R2]

Fact F2 : fire [from F2, F4 by R1]

Fact F6 : switch\_on\_sprinklers [from F2 by R3]

**Example 4 : A typical Backward Chaining**

Rule R1 : IF hot AND smoky THEN fire  
 Rule R2 : IF alarm\_beeps THEN smoky  
 Rule R3 : If \_fire THEN switch\_on\_sprinklers

Fact F1 : hot [Given]  
 Fact F2 : alarm\_beeps [Given]

Goal : Should I switch sprinklers on?

**Properties of Forward Chaining**

- all rules which can fire do fire.
- can be inefficient - lead to spurious rules firing, unfocused problem solving
- set of rules that can fire known as conflict set.
- decision about which rule to fire is conflict resolution.

**KR – forward chaining****Forward chaining algorithm - I**

Repeat  
 Collect the rule whose condition matches a fact in WM.  
 Do actions indicated by the rule.  
 (add facts to WM or delete facts from WM)  
 Until problem is solved or no condition match

**Backward chaining system**

Backward chaining means reasoning from goals back to facts. The idea is to focus on the search. Rules and facts are processed using backward chaining interpreter.

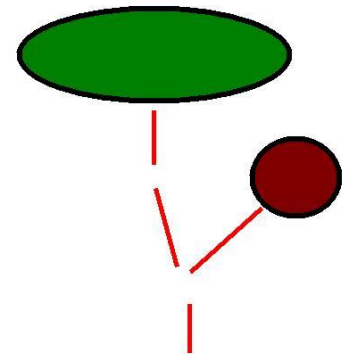
Checks hypothesis, e.g. "*should I switch the sprinklers on?*"

**Backward chaining algorithm**

Prove goal G

If **G** is in the initial facts , it is proven.  
Otherwise, find a rule which can be used to conclude **G**, and try to prove each of that rule's conditions.

**alarm\_beeps**  
**Smoky**  
**hot**  
**fire**  
**switch\_on\_sprinklers**



### Encoding of rules

Rule R1 : IF hot AND smoky THEN fire  
Rule R2 : IF alarm\_beeps THEN smoky  
Rule R3 : IF fire THEN switch\_on\_sprinklers

Fact F1 : hot [Given]  
Fact F2 : alarm\_beeps [Given]

Goal : Should I switch sprinklers on?

### Forward vs Backward Chaining

Depends on problem, and on properties of rule set.

Backward chaining is likely to be better if there is clear hypotheses. Examples : Diagnostic problems or classification problems, Medical expert systems

Forward chaining may be better if there is less clear hypothesis and want to see what can be concluded from current situation;

Examples : Synthesis systems - design / configuration.

### What are Bayesian nets?

- Bayesian nets (BN) are a network-based framework for representing and analyzing models involving uncertainty;
- BN are different from other knowledge-based systems tools because uncertainty is handled in mathematically accurate yet efficient and simple way
- BN are different from other probabilistic analysis tools because of network representation of problems, use of Bayesian statistics, and the synergy between these

### Definition of a Bayesian Network

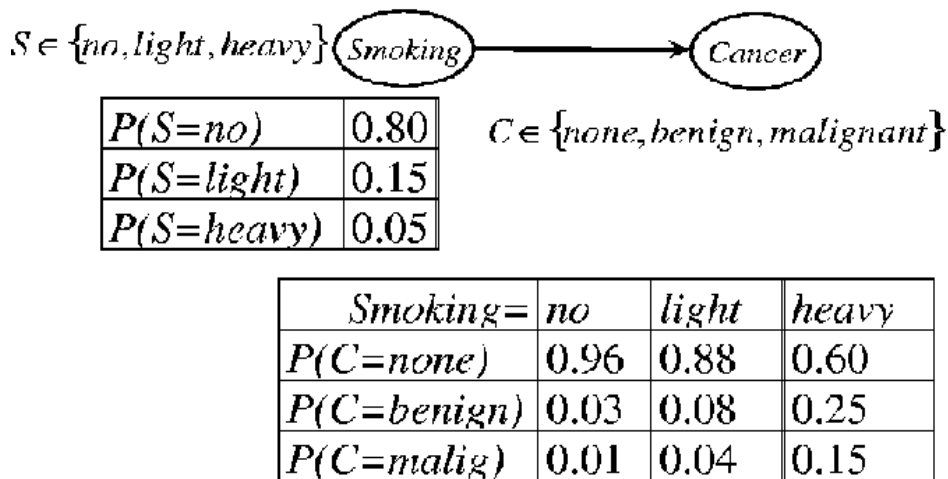
### Knowledge structure:

- variables are nodes
- arcs represent probabilistic dependence between variables
- conditional probabilities encode the strength of the dependencies

**Computational architecture:**

- computes posterior probabilities given evidence about some nodes
- exploits probabilistic independence for efficient computation

## Bayesian Networks




## Product Rule

■  $P(C,S) = P(C|S) P(S)$

$S \downarrow C \Rightarrow$	<i>none</i>	<i>benign</i>	<i>malignant</i>
<i>no</i>	0.768	0.024	0.008
<i>light</i>	0.132	0.012	0.006
<i>heavy</i>	0.035	0.010	0.005

# Marginalization

$S \downarrow C \Rightarrow$	<i>none</i>	<i>benign</i>	<i>malig</i>	total	
<i>no</i>	0.768	0.024	0.008	.80	$P(\text{Smoke})$
<i>light</i>	0.132	0.012	0.006	.15	
<i>heavy</i>	0.035	0.010	0.005	.05	
total	0.935	0.046	0.019		


  
 $P(\text{Cancer})$

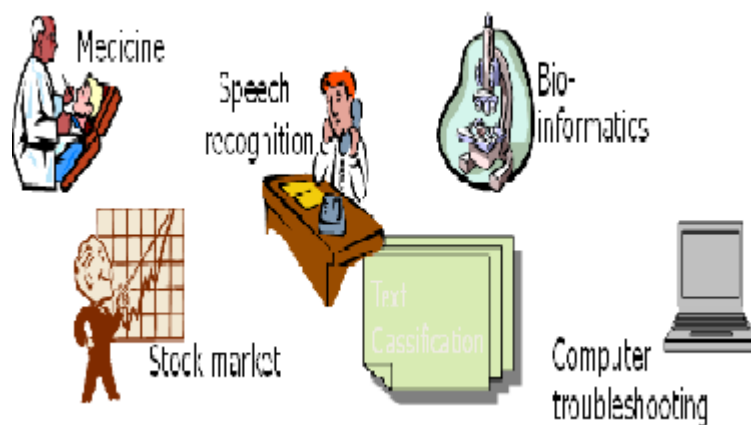
# Bayes Rule Revisited

$$P(S|C) = \frac{P(C|S)P(S)}{P(C)} = \frac{P(C,S)}{P(C)}$$

$S \Downarrow C \Rightarrow$	<i>none</i>	<i>benign</i>	<i>malig</i>
<i>no</i>	0.768/.935	0.024/.046	0.008/.019
<i>light</i>	0.132/.935	0.012/.046	0.006/.019
<i>heavy</i>	0.030/.935	0.015/.046	0.005/.019

<i>Cancer=</i>	<i>none</i>	<i>benign</i>	<i>malignant</i>
$P(S=no)$	0.821	0.522	0.421
$P(S=light)$	0.141	0.261	0.316
$P(S=heavy)$	0.037	0.217	0.263

What Bayesian Networks are good for?



Why learn Bayesian networks?

- Combining domain expert knowledge with data
- Efficient representation and inference



- **Incremental learning**
- **Handling missing data:** <1.3 2.8 ?? 0 1 >
- **Learning causal relationships:**

### Bayes rule

True Bayesians actually consider conditional probabilities as more basic than joint probabilities. It is easy to define  $P(A|B)$  without reference to the joint probability  $P(A,B)$ . To see this note that we can rearrange the conditional probability formula to get:

$$P(A|B) P(B) = P(A,B)$$

but by symmetry we can also get:

$$P(B|A) P(A) = P(A,B)$$

It follows that:

$$P(A|B) = \frac{P(B|A) P(A)}{P(B)}$$

which is the so-called **Bayes Rule**.

### Bayes Rule Example

**Question:** Suppose that we have two bags each containing black and white balls. One bag contains three times as many white balls as blacks. The other bag contains three times as many black balls as white. Suppose we choose one of these bags at random. For this bag we select five balls at random, replacing each ball after it has been selected. The result is that we find 4 white balls and one black. What is the probability that we were using the bag with mainly white balls?

**Solution.** Let  $A$  be the random variable "bag chosen" then  $A=\{a_1, a_2\}$  where  $a_1$  represents "bag with mostly white balls" and  $a_2$  represents "bag with mostly black balls". We know that  $P(a_1)=P(a_2)=1/2$  since we choose the bag at random.

Let  $B$  be the event "4 white balls and one black ball chosen from 5 selections".

Then we have to calculate  $P(a_1|B)$ . From Bayes' rule this is:

$$P(a_1|B) = \frac{P(B|a_1) \cdot P(a_1)}{P(B|a_1) \cdot P(a_1) + P(B|a_2) \cdot P(a_2)}$$

Now, for the bag with mostly white balls the probability of a ball being white is  $\frac{3}{4}$  and the probability of a ball being black is  $\frac{1}{4}$ . Thus, we can use the Binomial Theorem, to compute  $P(B|a_1)$  as:

$$P(B|a_1) = \binom{5}{1} \left(\frac{3}{4}\right)^4 \left(\frac{1}{4}\right)^1 = \frac{405}{1024}$$

Similarly

$$P(B|a_2) = \binom{5}{1} \left(\frac{1}{4}\right)^4 \left(\frac{3}{4}\right)^1 = \frac{15}{1024}$$

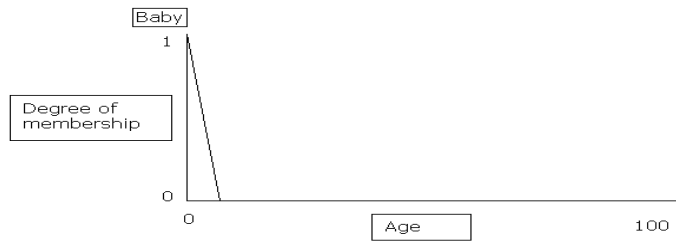
hence

$$P(a_1|B) = \frac{405/1024}{405/1024 + 15/1024} = \frac{405}{420} = 0.964$$

### **Fuzzy Reasoning**

#### **Cntents are:**

- **Bivalent and Multivalent Logics**
  - **Linguistic Variables**
  - **Fuzzy Sets**
  - **Membership Functions**
  - **Fuzzy Set Operators**
  - **Hedges**
  - **Fuzzy Logic**
  - **Fuzzy Rules**
  - **Fuzzy Inference**
  - **Fuzzy Expert Systems**
  - **Neuro-Fuzzy Systems**
- 
- **Bivalent (Aristotelian) logic :**
    - uses two logical values – true and false.
  - **Multivalent logics:**
    - use many logical values – often in a range of real numbers from 0 to 1.
  - **Difference between these two:**
    - Important to note the difference between multivalent logic and probability –  $P(A) = 0.5$  means that A may be true or may be false – a logical value of 0.5 means both true and false at the same time.
  - **Linguistic Variables:**
    - Variables used in fuzzy systems to express qualities such as height, which can take values such as “tall”, “short” or “very tall”.
    - These values define subsets of the universe of discourse.
  - **Fuzzy Sets**
    - A crisp set is a set for which each value either is or is not contained in the set.
    - For a fuzzy set, every value has a membership value, and so is a member to some extent.
    - The membership value defines the extent to which a variable is a member of a fuzzy set.
    - The membership value is from 0 (not at all a member of the set) to 1.
  - **Membership Functions**



- **Crisp Set Operators**

- Not A – the complement of A, which contains the elements which are not contained in A.
- $A \cap B$  – the intersection of A and B, which contains those elements which are contained in both A and B.
- $A \cup B$  – the union of A and B which contains all the elements of A and all the elements of B.
- Fuzzy sets use the same operators, but the operators have different meanings.

- **Fuzzy Set Operators**

- **Fuzzy set operators can be defined by their membership functions**
- $M_{\neg A}(x) = 1 - M_A(x)$
- $M_{A \cap B}(x) = \text{MIN}(M_A(x), M_B(x))$
- $M_{A \cup B}(x) = \text{MAX}(M_A(x), M_B(x))$
- **We can also define containment (subset operator):**
- $B \subset A \text{ iff } \forall x (M_B(x) \leq M_A(x))$

- **Hedges**

- A hedge is a qualifier such as “very”, “quite”, “somewhat” or “extremely”.
- When a hedge is applied to a fuzzy set it creates a new fuzzy set.
- Mathematic functions are usually used to define the effect of a hedge.
- For example, “Very” might be defined as:
- $M_{VA}(x) = (M_A(x))^2$

- **Fuzzy Logic**

- A nonmonotonic logical system that applies to fuzzy variables.
- We use connectives defined as:
- $A \vee B \equiv \text{MAX}(A, B)$
- $A \wedge B \equiv \text{MIN}(A, B)$
- $\neg A \equiv 1 - A$
- We can also define truth tables:

A	B	$A \vee B$
0	0	0
0	0.5	0.5
0	1	1
0.5	0	0.5
0.5	0.5	0.5
0.5	1	1
1	0	1
1	0.5	1
1	1	1

- **Fuzzy Rules**

- A fuzzy rule takes the following form:
- IF A op x then B = y
  - op is an operator such as >, < or =.
  - For example:
- IF temperature > 50 then fan speed = fast
- IF height = tall then trouser length = long
- IF study time = short then grades = poor

- **Fuzzy Inference**

- Inference is harder to manage.
- Since:
- $A \rightarrow B \equiv \neg A \vee B$
- Hence, we might define fuzzy inference as:
- $A \rightarrow B \equiv \text{MAX}((1 - A), B)$
- This gives the unintuitive truth table shown on the right.
- This gives us  $0.5 \rightarrow 0 = 0.5$ , where we would expect  $0.5 \rightarrow 0 = 0$
- An alternative is Gödel implication, which is defined as:
- $A \rightarrow B \equiv (A \leq B) \vee B$
- This gives a more intuitive truth table.

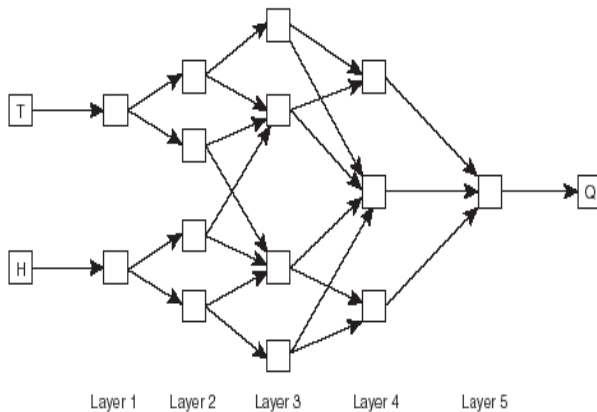
- **Fuzzy Expert Systems**

- A fuzzy expert system is built by creating a set of fuzzy rules, and applying fuzzy inference.
- In many ways this is more appropriate than standard expert systems since expert knowledge is not usually black and white but has elements of grey.
- The first stage in building a fuzzy expert system is choosing suitable linguistic variables.
- Rules are then generated based on the expert's knowledge, using the linguistic variables.

- **Neuro-Fuzzy Systems**

- 1 A fuzzy neural network is usually a feed-forward network with five layers:

1. Input layer – receives crisp inputs
2. Fuzzy input membership functions
3. Fuzzy rules
4. Fuzzy output membership functions
5. Output layer – outputs crisp values



### ➤ Dempster-Shafer Theory

- 5) Dempster-Shafer theory is an approach to combining evidence
- 6) Dempster (1967) developed means for combining degrees of belief derived from independent items of evidence.
- 7) His student, Glenn Shafer (1976), developed method for obtaining degrees of belief for one question from subjective probabilities for a related question
- 8) 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)

Frame of discernment :

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

- Bayes was concerned with evidence that supported single conclusions (e.g., evidence for each outcome  $\theta_i$  in  $\Theta$ ):

$p(\theta_i | E)$

- D-S Theory is concerned with evidences which support subsets of outcomes in  $\Theta$ , 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  $\Theta$  is the set of all possible subsets of  $\Theta$ :  
– E.g., if  $\Theta = \{\theta_1, \theta_2, \theta_3\}$
- Then the frame of discernment of  $\Theta$  is:  
(  $\emptyset, \theta_1, \theta_2, \theta_3, \{\theta_1, \theta_2\}, \{\theta_1, \theta_3\}, \{\theta_2, \theta_3\}, \{\theta_1, \theta_2, \theta_3\}$  )
- $\emptyset$ , 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  $A$ .” (wikipedia)

“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.

Mass function  $m(A)$ :

- 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  $B$ .

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  $A = \{q_1, q_2, q_3\}$

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

$$+ m(\{q_1, q_2\}) + m(\{q_2, q_3\}) + m(\{q_1, q_3\}) + m(\{q_1, q_2, q_3\})$$

## 3/PIT/IT/Artificial Intelligence

Belief in A: 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 0.3 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 0.3 0.1

bel(A) 0.1 0.2 0.1 0.4 0.3 0.6 1.0



**UNIT IV PLANNING AND MACHINE LEARNING****9**

Basic plan generation systems - Strips -Advanced plan generation systems – K strips -Strategic explanations -Why, Why not and how explanations. Learning- Machine learning, adaptive Learning.

**Planning:**

The task of coming up with a sequence of actions that will achieve a goal is called **planning**

**Advanced Problem Solving Approaches**

- In order to solve nontrivial problems, it is necessary to combine
  - Basic problem solving strategies
  - Knowledge representation mechanisms
  - Partial solutions and at the end combine into complete problem solution (decomposition)
- Planning refers to the process of computing several steps of a problem solving before executing any of them.
- Planning is useful as a problem solving technique for non decomposable problem.

**Components of a Planning System**

- In any general problem solving systems, elementary techniques to perform following functions are required
- Choose the best rule (based on heuristics) to be applied
- Apply the chosen rule to get new problem state
- Detect when a solution has been found
- Detect dead ends so that new directions are explored.

**Choose Rules to apply**

- Most widely used technique for selecting appropriate rules is to
  - first isolate a set of differences between the desired goal state and current state,
  - identify those rules that are relevant to reducing these difference,
  - if more rules are found then apply heuristic information to choose out of them.

**Apply Rules**

- In simple problem solving system,
  - applying rules was easy as each rule specifies the problem state that would result from its application.
  - In complex problem we deal with rules that specify only a small part of the complete problem state.

***Example: Block World Problem***

- Block world problem assumptions
  - Square blocks of same size
  - Blocks can be stacked one upon another.
  - Flat surface (table) on which blocks can be placed.
  - Robot arm that can manipulate the blocks. It can hold only one block at a time.
- In block world problem, the state is described by a set of predicates representing the facts that were true in that state.
- One must describe for every action, each of the changes it makes to the state description.
- In addition, some statements that everything else remains unchanged is also necessary.

### **Actions (Operations) done by Robot**

UNSTACK (X, Y) :      [US (X, Y)]

Pick up X from its current position on block Y. The arm must be empty and X has no block on top of it.

STACK (X, Y):      [S (X, Y)]

Place block X on block Y. Arm must holding X and the top of Y is clear.

PICKUP (X):      [PU (X) ]

Pick up X from the table and hold it. Initially the arm must be empty and top of X is clear.

PUTDOWN (X):      [PD (X)]

Put block X down on the table. The arm must have been holding block X.

Predicates used to describe the state

ON(X, Y)      -      Block X on block Y.

ONT(X)      -      Block X on the table.

CL(X)      -      Top of X clear.

HOLD(X)      -      Robot-Arm holding X.

AE      -      Robot-arm empty.

Logical statements true in this block world.

Holding X means, arm is not empty

$(\exists X) \text{ HOLD } (X) \rightarrow \sim \text{AE}$

X is on a table means that X is not on the top of any block

$(\forall X) \text{ ONT } (X) \rightarrow \sim (\exists Y) \text{ ON } (X, Y)$

Any block with no block on has clear top

$(\forall X) (\sim (\exists Y) \text{ ON } (Y, X)) \rightarrow \text{CL } (X)$

### **Effect of Unstack operation**

- The effect of US(X, Y) is described by the following axiom

▪  $[\text{CL}(X, \text{State}) \wedge \text{ON}(X, Y, \text{State})] \rightarrow$

•  $[\text{HOLD}(X, \text{DO}(\text{US}(X, Y), \text{State})) \wedge$

$\text{CL}(Y, \text{DO}(\text{US}(X, Y), \text{State}))]$

- DO is a function that generates a new state as a result of given action and a state.
- For each operator, set of rules (called frame axioms) are defined where the components of the state are
- affected by an operator

- If  $US(A, B)$  is executed in state  $S_0$ , then we can infer that  $HOLD(A, S_1) \wedge CLEAR(B, S_1)$  holds true, where  $S_1$  is new state after Unstack operation is executed.
  - not affected by an operator
    - If  $US(A, B)$  is executed in state  $S_0$ ,  $B$  in  $S_1$  is still on the table but we can't derive it. So frame rule stating this fact is defined as  $ONT(Z, S) \rightarrow ONT(Z, DO(US(A, B), S))$
- Advantage of this approach is that
  - simple mechanism of resolution can perform all the operations that are required on the state descriptions.
- Disadvantage is that
  - number of axioms becomes very large for complex problem such as COLOR of block also does not change.
  - So we have to specify rule for each attribute.
    - $COLOR(X, red, S) \rightarrow$ 
      - $COLOR(X, red, DO(US(Y, Z), s))$
- To handle complex problem domain, there is a need of mechanism that does not require large number of explicit frame axioms.

### ***STRIPS Mechanism***

- One such mechanism was used in early robot problem solving system named STRIPS (developed by Fikes, 1971).
- In this approach, each operation is described by three lists.
  - Pre\_Cond list contains predicates which have to be true before operation.
  - ADD list contains those predicates which will be true after operation
  - DELETE list contain those predicates which are no longer true after operation
- Predicates not included on either of these lists are assumed to be unaffected by the operation.
- Frame axioms are specified implicitly in STRIPS which greatly reduces amount of information stored.

### ***STRIPS – Style Operators***

- $S(X, Y)$ 
  - Pre:  $CL(Y) \wedge HOLD(X)$
  - Del:  $CL(Y) \wedge HOLD(X)$
  - Add:  $AE \wedge ON(X, Y)$
- $US(X, Y)$ 
  - Pre:  $ON(X, Y) \wedge CL(X) \wedge AE$
  - Del:  $ON(X, Y) \wedge AE$
  - Add:  $HOLD(X) \wedge CL(Y)$

- PU (X)
  - Pre:  $ONT(X) \wedge CL(X) \wedge AE$
  - Del:  $ONT(X) \wedge AE$
  - Add:  $HOLD(X)$
- PD (X)
  - Pre:  $HOLD(X)$
  - Del:  $HOLD(X)$
  - Add:  $ONT(X) \wedge AE$

### ***Simple Planning using a Goal Stack***

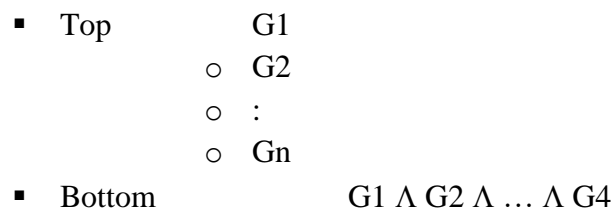
- One of the earliest techniques is planning using goal stack.
- Problem solver uses single stack that contains
  - sub goals and operators both
  - sub goals are solved linearly and then finally the conjoined sub goal is solved.
- Plans generated by this method will contain
  - complete sequence of operations for solving one goal followed by complete sequence of operations for the next etc.
- Problem solver also relies on
  - A database that describes the current situation.
  - Set of operators with precondition, add and delete lists.

### **Algorithm**

- Let us assume that the goal to be satisfied is:

$$\bullet \text{ GOAL} = G1 \wedge G2 \wedge \dots \wedge Gn$$

- Sub-goals  $G1, G2, \dots, Gn$  are stacked with compound goal  $G1 \wedge G2 \wedge \dots \wedge Gn$  at the bottom.



- At each step of problem solving process, the top goal on the stack is pursued.
- Find an operator that satisfies sub goal  $G1$  (makes it true) and replace  $G1$  by the operator.
  - If more than one operator satisfies the sub goal then apply some heuristic to choose one.
- In order to execute the top most operation, its preconditions are added onto the stack.

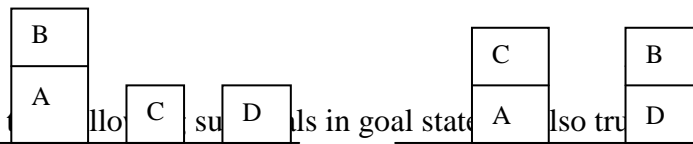
- Once preconditions of an operator are satisfied, then we are guaranteed that operator can be applied to produce a new state.
- New state is obtained by using ADD and DELETE lists of an operator to the existing database.
- Problem solver keeps track of operators applied.
  - This process is continued till the goal stack is empty and problem solver returns the plan of the problem.

**Goal stack method – Example**

- Logical representation of Initial and Goal states:
  - Initial State:  $ON(B, A) \wedge ONT(C) \wedge ONT(A) \wedge ONT(D) \wedge CL(B) \wedge CL(C) \wedge CL(D) \wedge AE$
  - Goal State:  $ON(C, A) \wedge ON(B, D) \wedge ONT(A) \wedge ONT(D) \wedge CL(C) \wedge CL(B) \wedge AE$

Initial State

Goal State



- We notice that blocks A, C, and D are in both the initial and goal states. Also, blocks B and D are in the goal state but not in the initial state.
- $ONT(A) \wedge ONT(D) \wedge CL(C) \wedge CL(B) \wedge AE$
- Represent for the sake of simplicity - **TSUBG**.
- Only sub-goals  $ON(C, A)$  &  $ON(B, D)$  are to be satisfied and finally make sure that TSUBG remains true.
- Either start solving first  $ON(C, A)$  or  $ON(B, D)$ . Let us solve first  $ON(C, A)$ .
- To solve  $ON(C, A)$ , operation  $S(C, A)$  could only be applied.
- So replace  $ON(C, A)$  with  $S(C, A)$  in goal stack.

**Goal Stack:**

$S(C, A)$   
 $ON(B, D)$   
 $ON(C, A) \wedge ON(B, D) \wedge TSUBG$

- $S(C, A)$  can be applied if its preconditions are true. So add its preconditions on the stack.

**Goal Stack:**

$CL(A)$   
 $HOLD(C)$       Preconditions of STACK

$CL(A) \wedge HOLD(C)$

$S(C, A)$  Operator

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge TSUBG$

■ Next check if  $CL(A)$  is true in State\_0.

■ Since it is not true in State\_0, only operator that could make it true is  $US(B, A)$ .

■ So replace  $CL(A)$  with  $US(B, A)$  and add its preconditions.

**Goal Stack:**  $ON(B, A)$

$CL(B)$  Preconditions of UNSTACK

$AE$

$ON(B, A) \wedge CL(B) \wedge AE$

$US(B, A)$  Operator

$HOLD(C)$

$CL(A) \wedge HOLD(C)$

$S(C, A)$  Operator

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge TSUBG$

■  $ON(B, A)$ ,  $CL(B)$  and  $AE$  are all true in initial state, so pop these along with its compound goal.

■ Next pop top operator  $US(B, A)$  and produce new state by using its ADD and DELETE lists.

■ Add  $US(B, A)$  in a queue of sequence of operators.

**SQUEUE =  $US(B, A)$**

State\_1:

$ONT(A) \wedge ONT(C) \wedge ONT(D) \wedge HOLD(B) \wedge ACL(A) \wedge CL(C) \wedge ACL(D)$

**Goal Stack:**

$HOLD(C)$

$CL(A) \wedge HOLD(C)$

$S(C, A)$  Operator

$ON(B, D)$

$ON(C, A) \wedge ON(B, D) \wedge TSUBG$

■  $ON(B, A)$ ,  $CL(B)$  and  $AE$  are all true in initial state, so pop these along with its compound goal.

■ Next pop top operator  $US(B, A)$  and produce new state by using its ADD and DELETE lists.

■ Add  $US(B, A)$  in a queue of sequence of operators.

**SQUEUE =  $US(B, A)$**

State\_1:

$ONT(A) \wedge ONT(C) \wedge ONT(D) \wedge HOLD(B) \wedge ACL(A) \wedge CL(C) \wedge ACL(D)$

**Goal Stack:**

HOLD(C)  
 CL(A) )  $\wedge$  HOLD(C)  
 S (C, A) Operator  
 ON(B, D)  
 ON(C, A)  $\wedge$  ON(B, D)  $\wedge$  TSUBG

- To satisfy the goal HOLD(C), two operators can be used e.g., PU(C ) or US(C, X), where X could be any block. Let us choose PU(C ) and proceed further.
- Repeat the process. Change in states is shown below.

**State\_1:**

$ONT(A) \wedge ONT(C) \wedge ONT(D) \wedge HOLD(B) \wedge CL(A) \wedge CL(C) \wedge CL(D)$   
 SQUEUE = US (B, A)

- Next operator to be popped of is S(B, D). So

**State\_2:**

$ONT(A) \wedge ONT(C) \wedge ONT(D) \wedge ON(B, D) \wedge CL(A) \wedge CL(C) \wedge CL(B) \wedge AE$   
 SQUEUE = US (B, A), S(B, D)

**State\_3:**

$ONT(A) \wedge HOLD(C) \wedge ONT(D) \wedge ON(B, D) \wedge CL(A) \wedge CL(B)$   
 SQUEUE = US (B, A), S(B, D), PU(C )

**State\_4:**

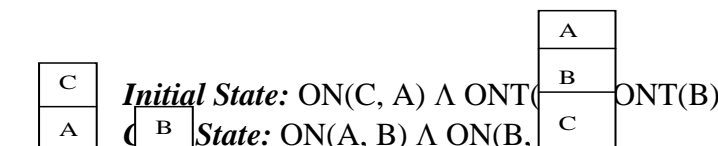
$ONT(A) \wedge ON(C, A) \wedge ONT(D) \wedge ON(B, D) \wedge CL(C) \wedge CL(B) \wedge AE$   
 SQUEUE = US (B, A), S(B, D), PU(C ), S(C, A)

**Difficult Problem**

- The Goal stack method is not efficient for difficult problems such as Sussman anomaly problem.
- It fails to find good solution.
- Let us consider the Sussman anomaly problem

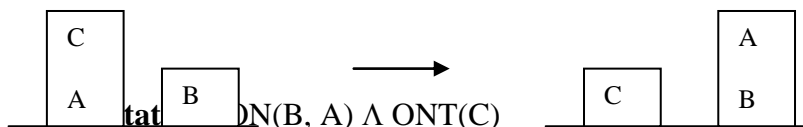
Initial State (State0)

Goal State

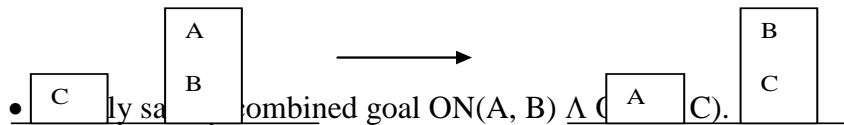


- Remove CL and AE predicates for the sake of simplicity.
- To satisfy ON(A, B), following operators are applied

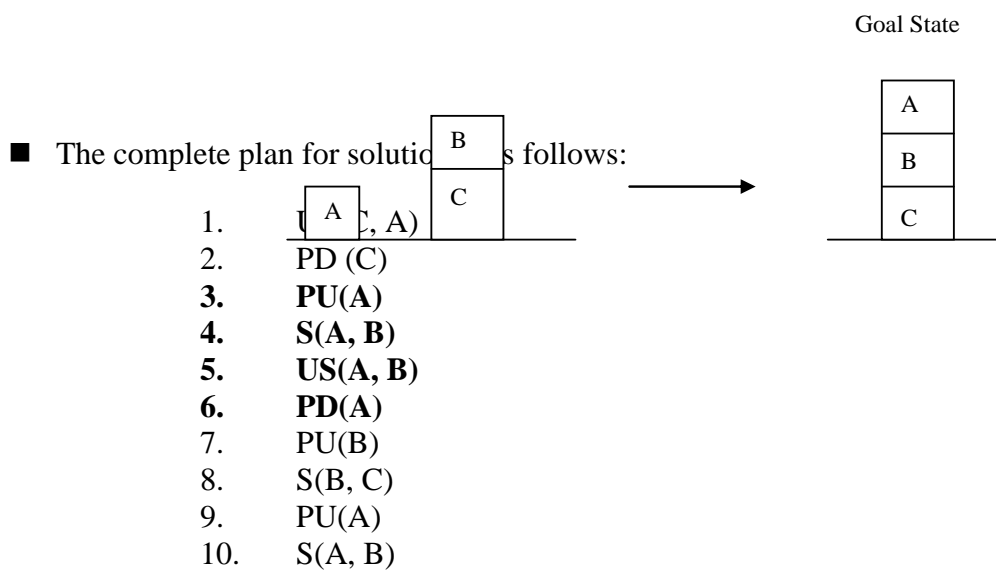
$US(C, A), PD(C), PU(A) \text{ and } S(A, B)$



- To satisfy  $ON(B, C)$ , following operators are applied
- $US(A, B)$ ,  $PD(A)$ ,  $PU(B)$  and  $S(B, C)$
- **State\_2:**  $ON(B, C) \wedge ONT(A)$



- Combined goal fails as while satisfying  $ON(B, C)$ , we have undone  $ON(A, B)$ .
- Difference in goal and current state is  $ON(A, B)$ .
- Operations required are  $PU(A)$  and  $S(A, B)$



- Although this plan will achieve the desired goal, but it is not efficient.
- In order to get efficient plan, either repair this plan or use some other method.
- Repairing is done by looking at places where operations are done and undone immediately, such as  $S(A, B)$  and  $US(A, B)$ .
- By removing them, we get

1.  $US(C, A)$
2.  $PD(C)$
3.  $PU(B)$
4.  $S(B, C)$
5.  $PU(A)$
6.  $S(A, B)$

### STRIPS planning



STRIPS stands for Standard Research Institute Problem Solver. It is an [automated planner](#) developed by [Richard Fikes](#) and [Nils Nilsson](#) in 1971 at [SRI International](#).

**Definition:**

A STRIPS instance is composed of:

- An initial state;
- The specification of the goal states – situations which the planner is trying to reach;
- A set of actions. For each action, the following are included:
  - preconditions (what must be established before the action is performed);
  - postconditions (what is established after the action is performed).

Mathematically, a STRIPS instance is a quadruple  $\langle S, O, I, G \rangle$ , in which each component has the following meaning:

1.  $S$  is a set of *conditions* (i.e., [propositional variables](#));
2.  $O$  is a set of *operators* (i.e., actions); each operator is itself a quadruple  $\langle P, A, E, R \rangle$ , each element being a set of conditions. These four sets specify, in order, which conditions must be true for the action to be executable, which ones must be false, which ones are made true by the action and which ones are made false;
3.  $I$  is the initial state, given as the set of conditions that are initially true (all others are assumed false);
4.  $G$  is the specification of the goal state; this is given as a pair  $\langle N, M \rangle$ , which specify which conditions are true and false, respectively, in order for a state to be considered a goal state.

**STRIPS problem**

A monkey is at location A in a lab. There is a box in location C. The monkey wants the bananas that are hanging from the ceiling in location B, but it needs to move the box and climb onto it in order to reach them.

Initial state: At(A), Level(low), BoxAt(C), BananasAt(B)

Goal state: Have(Bananas)

Actions:

// move from X to Y

\_Move(X, Y)\_

Preconditions: At(X), Level(low)

Postconditions: not At(X), At(Y)

// climb up on the box

\_ClimbUp(Location)\_

Preconditions: At(Location), BoxAt(Location), Level(low)

Postconditions: Level(high), not Level(low)

// climb down from the box

\_ClimbDown(Location)\_

Preconditions: At(Location), BoxAt(Location), Level(high)

Postconditions: Level(low), not Level(high)

// move monkey and box from X to Y

\_MoveBox(X, Y)\_

Preconditions: At(X), BoxAt(X), Level(low)

Postconditions: BoxAt(Y), not BoxAt(X), At(Y), not At(X)

// take the bananas

\_TakeBananas(Location)\_

Preconditions: At(Location), BananasAt(Location), Level(high)

Postconditions: Have(bananas)

## CS6659-ARTIFICIAL INTELLIGENCE

### VI SEMESTER

### UNIT -V

#### SYLLABUS:

#### **EXPERT SYSTEMS**

Expert systems - Architecture of expert systems, Roles of expert systems - Knowledge Acquisition – Meta knowledge, Heuristics. Typical expert systems - MYCIN, DART, XOON, Expert systems shells.

#### Expert Systems:

- The expert systems are the computer applications developed to solve complex problems in a particular domain, at the level of extra-ordinary human intelligence and expertise.
- An Expert System is a system that employs human knowledge captured in a computer to solve problems that ordinarily require human expertise
- ES imitate the expert's reasoning processes to solve specific problems

#### History of Expert Systems

##### 1. Early to Mid-1960s

- One attempt: the General-purpose Problem Solver (GPS)
- General-purpose Problem Solver (GPS)
- A procedure developed by Newell and Simon [1973] from their Logic Theory Machine -
  - Attempted to create an "intelligent" computer
  - general problem-solving methods applicable across domains
  - Predecessor to ES
  - Not successful, but a good start

##### 2. Mid-1960s: Special-purpose ES programs

- DENDRAL
- MYCIN
- Researchers recognized that the problem-solving mechanism is only a small part of a complete, intelligent computer system
- General problem solvers cannot be used to build high performance ES

- Human problem solvers are good only if they operate in a very narrow domain
- Expert systems must be constantly updated with new information
- The complexity of problems requires a considerable amount of knowledge about the problem area

### 3. Mid 1970s

- Several Real Expert Systems Emerge
  - Recognition of the Central Role of Knowledge
  - AI Scientists Develop
  - Comprehensive knowledge representation theories
  - General-purpose, decision-making procedures and inferences
  - Limited Success Because
  - Knowledge is Too Broad and Diverse
  - Efforts to Solve Fairly General Knowledge-Based Problems were Premature
- Several knowledge representations worked
- Key Insight
- The power of an ES is derived from the specific knowledge it possesses, not from the particular formalisms and inference schemes it employs

### 4. Early 1980s

- ES Technology Starts to go Commercial
  - XCON
  - XSEL
  - CATS-1
- Programming Tools and Shells Appear
  - EMYCIN
  - EXPERT
  - META-DENDRAL
  - EURISKO
- About 1/3 of These Systems Are Very Successful and Are Still in Use

### Characteristics of Expert Systems

- High performance
- Understandable
- Reliable
- Highly responsive
- 

### Capabilities of Expert Systems

The expert systems are capable of –

- Advising
- Instructing and assisting human in decision making
- Demonstrating
- Deriving a solution
- Diagnosing

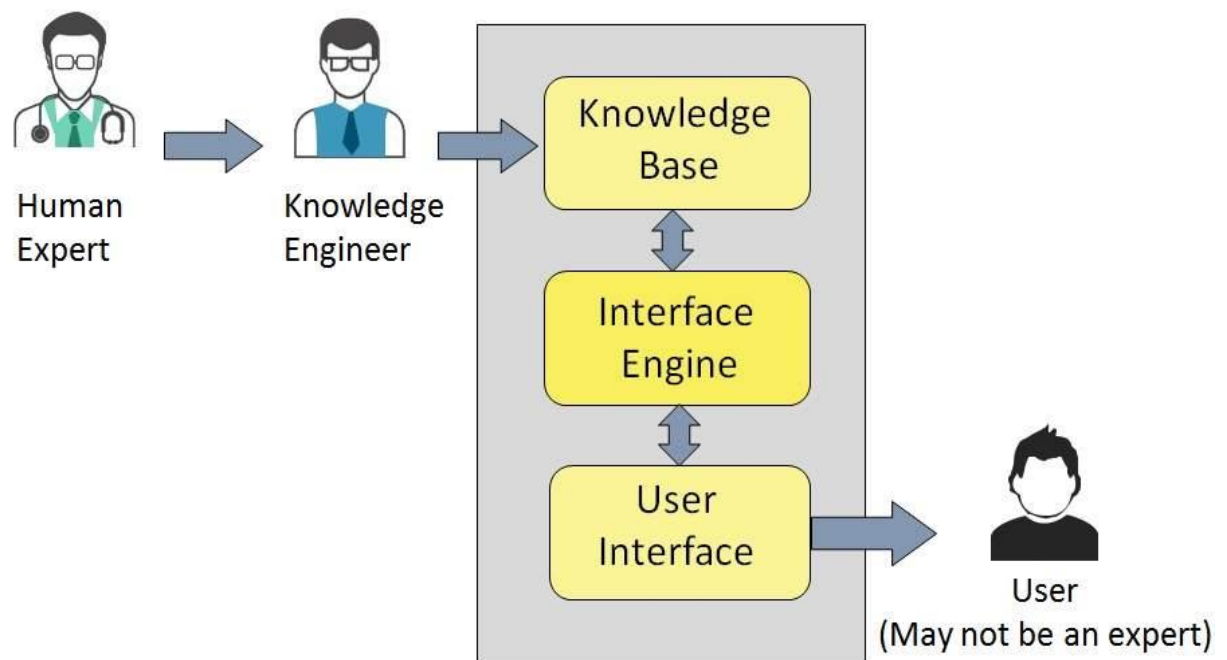
- Explaining
- Interpreting input
- Predicting results
- Justifying the conclusion
- Suggesting alternative options to a problem

### Components of Expert Systems

The components of ES include –

- Knowledge Base
- Interface Engine
- User Interface

Let us see them one by one briefly –



### Requirements of Efficient ES User Interface

- It should help users to accomplish their goals in shortest possible way.
- It should be designed to work for user's existing or desired work practices.
- Its technology should be adaptable to user's requirements; not the other way round.
- It should make efficient use of user input.

### Expert Systems Limitations

No technology can offer easy and complete solution. Large systems are costly, require significant development time, and computer resources. ESs have their limitations which include

- Limitations of the technology
- Difficult knowledge acquisition
- ES are difficult to maintain
- High development costs

### **Applications of Expert System**

The following table shows where ES can be applied.

Application	Description
Design Domain	Camera lens design, automobile design.
Medical Domain	Diagnosis Systems to deduce cause of disease from observed data, conduction medical operations on humans.
Monitoring Systems	Comparing data continuously with observed system with prescribed behavior such as leakage monitoring on long petroleum pipeline.
Process Control Systems	Controlling a physical process based on monitoring.
Knowledge Domain	Finding out faults in vehicles, computers.
Finance/Commerce	Detection of possible fraud, suspicious transaction in stock market trading, Airline scheduling, cargo scheduling.

### **Example:**

#### Backward Chaining in Expert Systems

- Backward chaining is often used in expert systems that are designed for medical diagnosis:
- For each hypothesis, H:
  - If H is in the facts database, it is proved.
  - Otherwise, if H can be determined by asking a question, then enter the user's answer in the facts database. Hence, it can be determined whether H is true or false, according to the user's answer.
  - Otherwise, find a rule whose conclusion is H. Now apply this algorithm to try to prove this rule's antecedents.

- If none of the above applies, we have failed to prove H.
- Usually backward chaining is used in conjunction with forward chaining.

### **Simple Medical Expert System**

- Rules
  1. If headache then prescribe pain killer
  2. If headache and sore throat and coughing than diagnose flu
  3. If tired and headache then diagnose glandular fever
  4. If tired and sore throat then diagnose tonsillitis
  5. If tired than diagnose stress
- User interaction
  1. Patient: I have a headache
  2. System: I suggest you take a pain killer. Do you have a sore throat?
  3. Patient: Yes, I do.
  4. System: Have you been coughing?
  5. Patient: No, why do you ask?
  6. System: If you had a head ache and a sore throat and had been coughing, I would have to conclude you might have the flu. It doesn't sound as though you have the flu? Have you been feeling tired recently?
  7. Patient: Yes, I have.
  8. System: It sounds as though you may have glandular fever or tonsillitis.
- Expert System vs. knowledge-based system
- An Expert System is a system that employs human knowledge captured in a computer to solve problems that ordinarily require human expertise
- ES imitate the expert's reasoning processes to solve specific problems

### **Expert System Technology**

There are several levels of ES technologies available. Expert systems technologies include –

- Expert System Development Environment – The ES development environment includes hardware and tools. They are –
  - Workstations, minicomputers, mainframes.
  - High level Symbolic Programming Languages such as LIStProgramming (LISP) and PROgrammation en LOGique (PROLOG).
  - Large databases.
- Tools – They reduce the effort and cost involved in developing an expert system to large extent.
  - Powerful editors and debugging tools with multi-windows.
  - They provide rapid prototyping
  - Have Inbuilt definitions of model, knowledge representation, and inference design.
- 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.

### **Development of Expert Systems: General Steps**

The process of ES development is iterative. Steps in developing the ES include –  
Identify Problem Domain

- The problem must be suitable for an expert system to solve it.
- Find the experts in task domain for the ES project.
- Establish cost-effectiveness of the system.

Design the System

- Identify the ES Technology
- Know and establish the degree of integration with the other systems and databases.
- Realize how the concepts can represent the domain knowledge best.

Develop the Prototype

From Knowledge Base: The knowledge engineer works to –

- Acquire domain knowledge from the expert.
- Represent it in the form of If-THEN-ELSE rules.

Test and Refine the Prototype

- The knowledge engineer uses sample cases to test the prototype for any deficiencies in performance.
- End users test the prototypes of the ES.

Develop and Complete the ES

- Test and ensure the interaction of the ES with all elements of its environment, including end users, databases, and other information systems.
- Document the ES project well.
- Train the user to use ES.

Maintain the ES

- Keep the knowledge base up-to-date by regular review and update.
- Cater for new interfaces with other information systems, as those systems evolve.

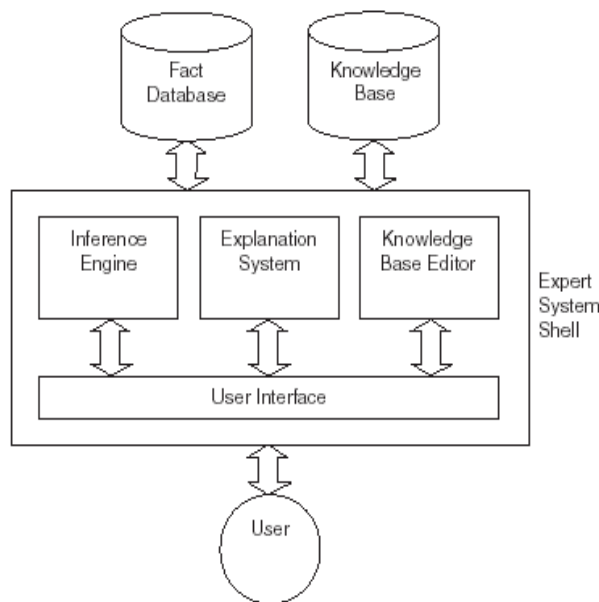
### **Benefits of Expert Systems**

- Availability – They are easily available due to mass production of software.
- Less Production Cost – Production cost is reasonable. This makes them affordable.
- Speed – They offer great speed. They reduce the amount of work an individual puts in.
- Less Error Rate – Error rate is low as compared to human errors.
- Reducing Risk – They can work in the environment dangerous to humans.
- Steady response – They work steadily without getting motional, tensed or fatigued.

### **The Architecture of Expert Systems**

- Expert knowledge derived from human experts
- Purpose:
  - Diagnose illnesses

- Provide recommendations
- Solve other problems
- Knowledge base: database of rules (domain knowledge).
- Explanation system: explains the decisions the system makes.
- User Interface: the means by which the user interacts with the expert system.
- Knowledge base editor: allows the user to edit the information in the knowledge base.



### Expert System Shells

- The part of an expert system that does not contain any domain specific or case specific knowledge is the expert system shell.
- A single expert system shell can be used to build a number of different expert systems.
- An example of an expert system shell is CLIPS.

### Knowledge Engineering

- Takes knowledge from experts and inputs it into the expert system.
- Usually choose which expert system shell to use.
- Responsible for entering meta-rules.

### CLIPS

- CLIPS is C Language Integrated Production System – an expert system shell.
- CLIPS uses a LISP-like notation to enter rules.



```

CLIPS> (defrule rule1
(elevator ?floor_now)
(button ?floor_now)
=>
(assert (open_door)))
CLIPS> (defrule rule2
(elevator ?floor_now)
(button ?other_floor)
=>
(assert (goto ?other_floor)))
CLIPS> (assert (elevator floor1))
==> f-0 (elevator floor1)
<Fact-0>
CLIPS> (assert (button floor3))
==> f-1 (button floor3)
<Fact-1>
<CLIPS> (run)
==>f-2 (goto floor3)

```

### Comparison of expert systems with conventional systems and human experts

<i>Human Experts</i>	<i>Expert Systems</i>	<i>Conventional Programs</i>
Use knowledge in the form of rules of thumb or heuristics to solve problems in a narrow domain.	Process knowledge expressed in the form of rules and use symbolic reasoning to solve problems in a <i>narrow domain</i> .	Process data and use algorithms, a series of well-defined operations, to solve general numerical problems.
In a human brain, knowledge exists in a compiled form.	Provide a <i>clear separation of knowledge from its processing</i> .	Do not separate knowledge from the control structure to process this knowledge.
Capable of explaining a line of reasoning and providing the details.	<i>Trace the rules fired</i> during a problem-solving session and <i>explain how</i> a particular conclusion was reached and <i>why</i> specific data was needed.	Do not explain how a particular result was obtained and why input data was needed.

### Knowledge acquisition

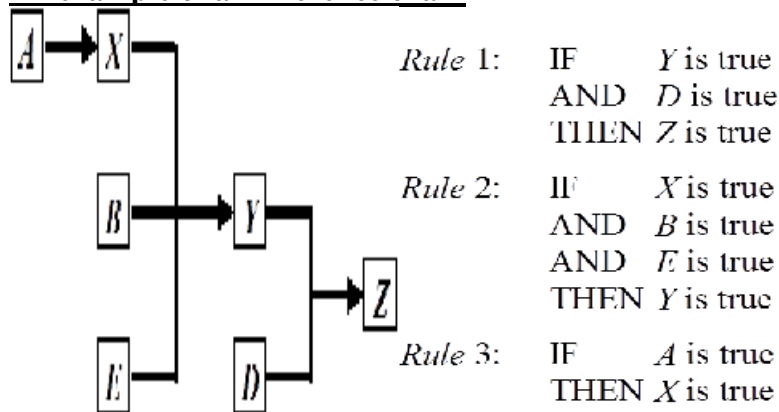
- Knowledge acquisition can be regarded as a method by which a knowledge engineer gathers information mainly from experts, but also from text books, technical manuals, research papers and other authoritative sources for ultimate translation into a knowledge base, understandable by both machines and humans.
- The person undertaking the knowledge acquisition, the knowledge engineer, must convert the acquired knowledge into an electronic format that a computer program can use.

### The Process of Knowledge Acquisition

- In the process of Knowledge Acquisition for an Expert System Project, the knowledge engineer basically performs four major tasks in sequence:

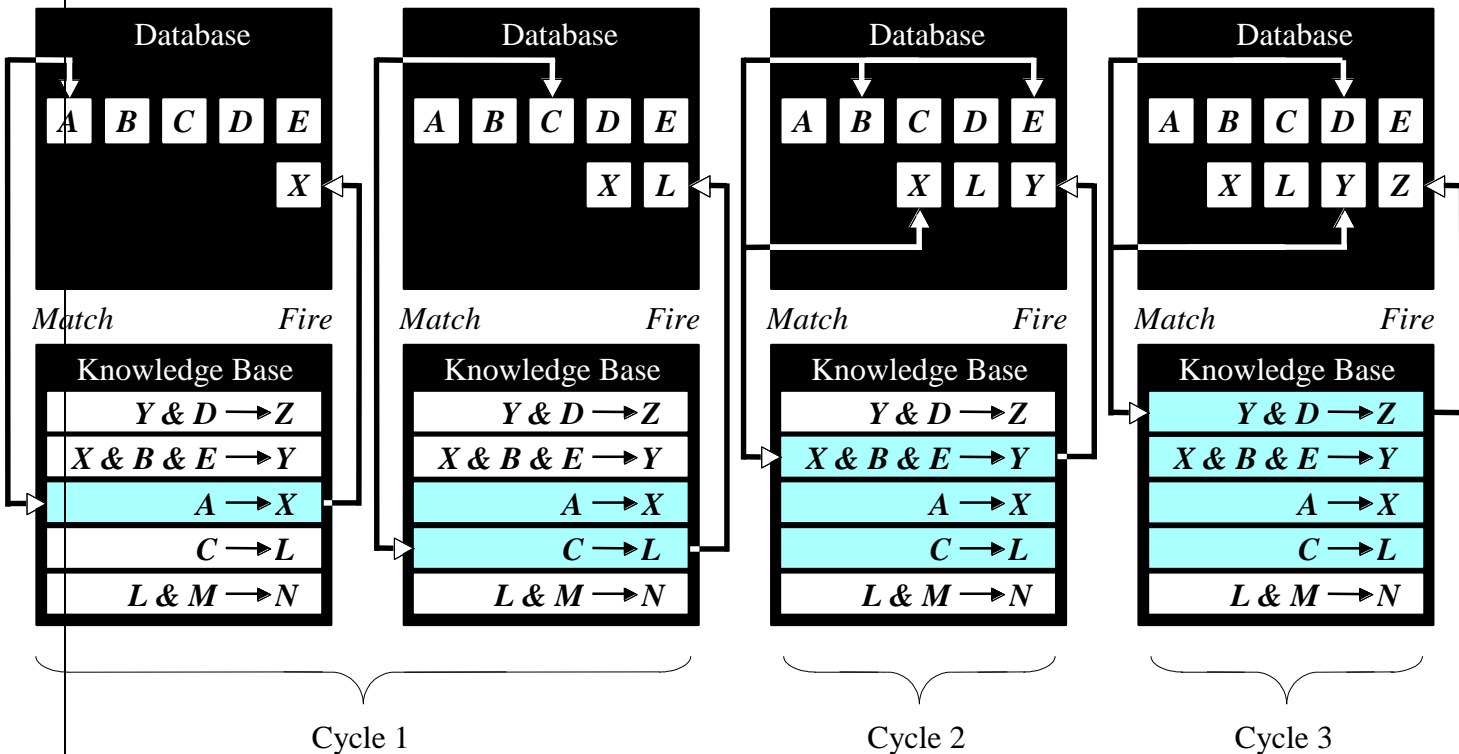
- **First**, the engineer ensures that he or she understands the aims and objectives of the proposed expert system to get a feeling for the potential scope of the project.
- **Second**, the engineer develops a working knowledge of the problem domain by mastering it's terminology by looking up technical dictionaries and terminology data bases. For this task the key sources of knowledge are identified: textbooks, papers, technical reports, manuals, codes of practice, users and domain experts.
- **Third**, the knowledge engineer interacts with experts via meetings or interviews to acquire, verify and validate their knowledge.
- **Fourth**, the knowledge engineer produces a "document knowledge base"; a document or group of documents (nowadays in electronic format) which form an intermediate stage in the translation of knowledge from source to computer program. This comprises:
  - the interview transcripts,
  - the analysis of the information they contain
  - and a full description of the major domain entities (e.g. tasks, rules and objects).

#### An example of an inference chain



**Forward chaining**

- Forward chaining is the **data-driven reasoning**.
- The reasoning starts from the known data and proceeds forward with that data.
- Each time only the topmost rule is executed.
- When fired, the rule adds a new fact in the database.
- Any rule can be executed only once. T
- he match-fire cycle stops when no further rules can be fired.



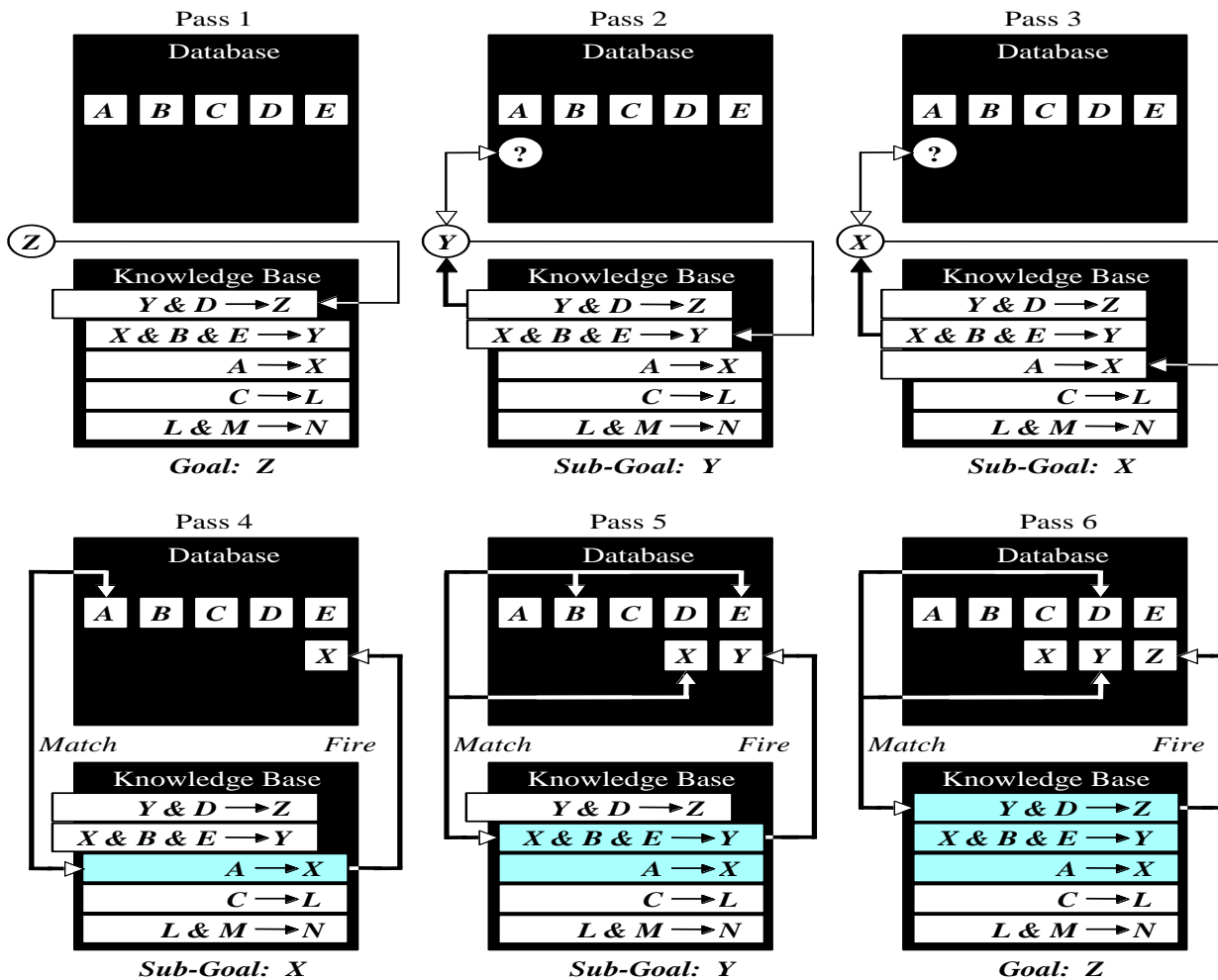
Forward chaining is a technique for gathering information and then inferring from it whatever can be inferred.

However, in forward chaining, many rules may be executed that have nothing to do with the established goal.

Therefore, if our goal is to infer only one particular fact, the forward chaining inference technique would not be efficient.

**Backward chaining**

- Thus the inference engine puts aside the rule it is working with (the rule is said to *stack*) and sets up a new goal, a subgoal, to prove the IF part of this rule.
- Then the knowledge base is searched again for rules that can prove the subgoal.
- The inference engine repeats the process of stacking the rules until no rules are found in the knowledge base to prove the current subgoal.



### How do we choose between forward and backward chaining?

- If an expert first needs to gather some information and then tries to infer from it whatever can be inferred, choose the forward chaining inference engine.
- However, if your expert begins with a hypothetical solution and then attempts to find facts to prove it, choose the backward chaining inference engine.

### Conflict resolution

Earlier we considered two simple rules for crossing a road. Let us now add third rule:

Rule 1:

IF the 'traffic light' is green  
THEN the action is go

Rule 2:

IF the 'traffic light' is red  
THEN the action is stop

Rule 3

IF the 'traffic light' is red  
 THEN the action is go

We have two rules, *Rule 2* and *Rule 3*, with the same IF part. Thus both of them can be set to fire when the condition part is satisfied. These rules represent a conflict set. The inference engine must determine which rule to fire from such a set. A method for choosing a rule to fire when more than one rule can be fired in a given cycle is called **conflict resolution**.

**Methods used for conflict resolution**

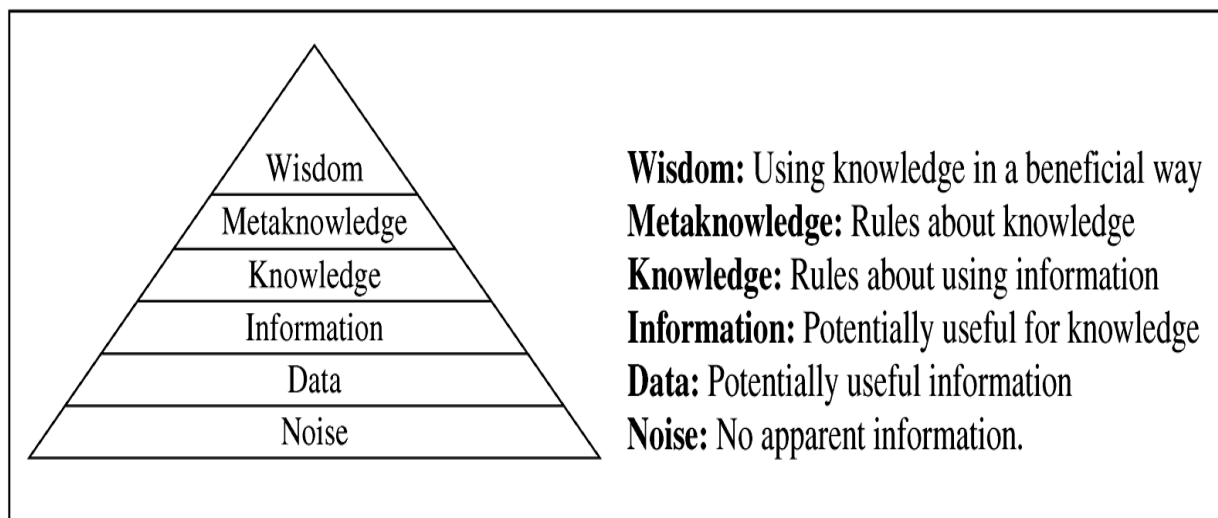
- Fire the rule with the **highest priority**. In simple applications, the priority can be established by placing the rules in an appropriate order in the knowledge base. Usually this strategy works well for expert systems with around 100 rules.
- Fire the **most specific rule**. This method is also known as the **longest matching strategy**. It is based on the assumption that a specific rule processes more information than a general one.

Fire the rule that uses the **data most recently entered** in the database. This method relies on time tags attached to each fact in the database. In the conflict set, the expert system first fires the rule whose antecedent uses the data most recently added to the database

**METAKNOWLEDGE**

- Metaknowledge can be simply defined as **knowledge about knowledge**. Metaknowledge is knowledge about the use and control of domain knowledge in an expert system.
- In rule-based expert systems, metaknowledge is represented by **metarules**. A metarule determines a strategy for the use of task-specific rules in the expert system.

**Figure 2.2 The Pyramid of Knowledge**

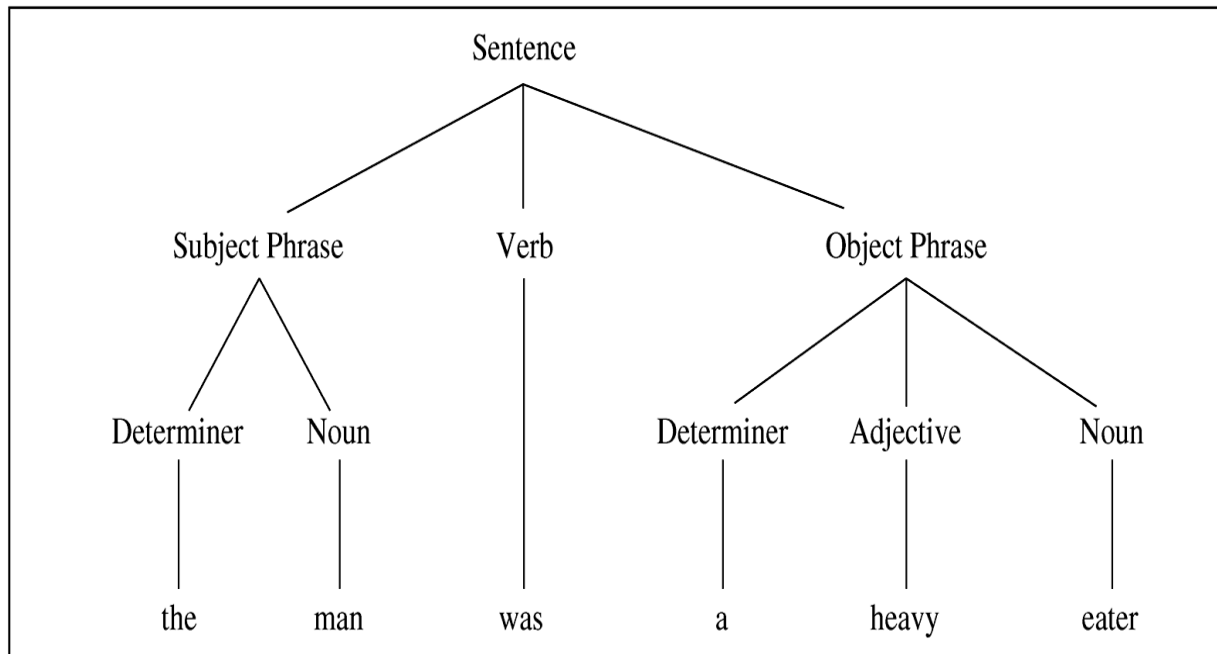
**Productions**

A number of knowledge-representation techniques have been devised:

- Rules

- Semantic nets
- Frames
- Scripts
- Logic
- Conceptual graphs

**Figure 2.3 Parse Tree of a Sentence**



### Metarules

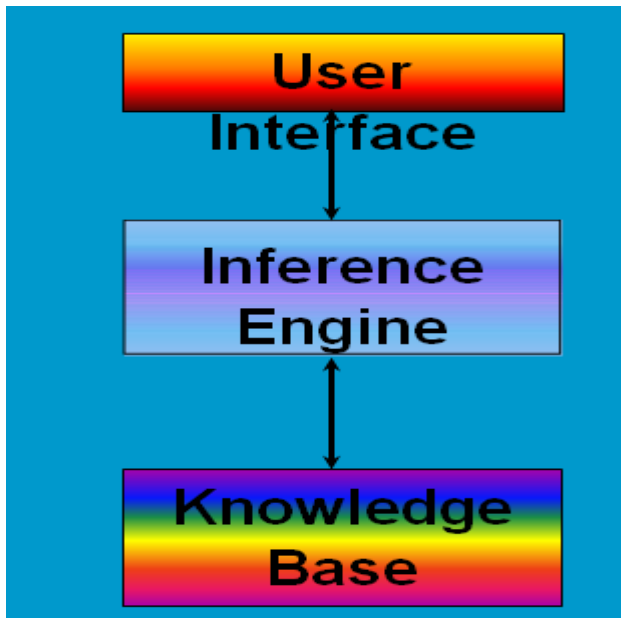
#### *Metarule 1:*

Rules supplied by experts have higher priorities than rules supplied by novices.

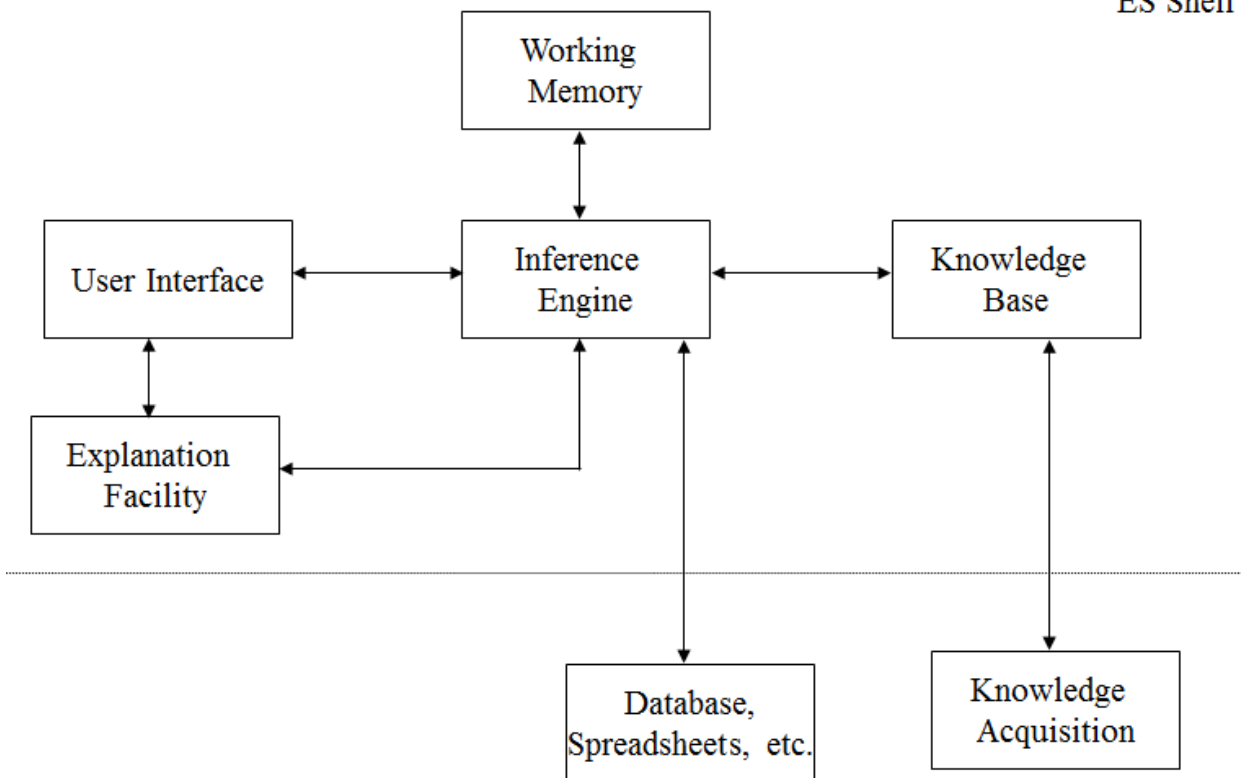
#### *Metarule 2:*

Rules governing the rescue of human lives have higher priorities than rules concerned with clearing overloads on power system equipment.

### Three Major ES Components



ES Shell



Basic ES Structure

All ES Components

- Knowledge Acquisition Subsystem
- Knowledge Base
- Inference Engine
- User Interface
- Blackboard (Workplace)
- Explanation Subsystem (Justifier)
- Knowledge Refining System
- User
- Most ES do not have a Knowledge Refinement Component
- (See Figure 10.3)
- Knowledge Base
- The knowledge base contains the knowledge necessary for understanding, formulating, and solving problems
- Two Basic Knowledge Base Elements
  - Facts
  - Special heuristics, or rules that direct the use of knowledge
  - Knowledge is the primary raw material of ES
  - Incorporated knowledge representation
- Inference Engine
- The brain of the ES
- The control structure (rule interpreter)
- Provides methodology for reasoning
- The Human Element in Expert Systems
- Builder and User
- Expert and Knowledge engineer.
- The Expert
  - Has the special knowledge, judgment, experience and methods to give advice and solve problems
  - Provides knowledge about task performance
- The Knowledge Engineer
  - Helps the expert(s) structure the problem area by interpreting and integrating human answers to questions, drawing analogies, posing counterexamples, and bringing to light conceptual difficulties
  - Usually also the System Builder
- How Expert Systems Work
  - Development
  - Consultation
  - Improvement
- Problem Areas Addressed by Expert Systems
  - Interpretation systems
  - Prediction systems
  - Diagnostic systems
  - Design systems
  - Planning systems
  - Monitoring systems
  - Debugging systems



- Repair systems
- Instruction systems
- Control systems
- Expert Systems Benefits
- Improved Decision Quality
- Increased Output and Productivity
- Decreased Decision Making Time
- Increased Process(es) and Product Quality
- Capture Scarce Expertise
- Can Work with Incomplete or Uncertain Information
- Enhancement of Problem Solving and Decision Making
- Improved Decision Making Processes
- Knowledge Transfer to Remote Locations

### **Expert Systems Types**

- Expert Systems Versus Knowledge-based Systems
- Rule-based Expert Systems
- Frame-based Systems
- Hybrid Systems
- Model-based Systems
- Ready-made (Off-the-Shelf) Systems
- Real-time Expert Systems

### **ES on the Web**

- ☐ Provide knowledge and advice
- ☐ Help desks
- ☐ Knowledge acquisition
- ☐ Spread of multimedia-based expert systems (Intelimedia systems)
- ☐ Support ES and other AI technologies provided to the Internet/Intranet

### **MYCIN**

**MYCIN** was an early [expert system](#) that used [artificial intelligence](#) to identify bacteria causing severe infections, such as bacteremia and meningitis, and to recommend [antibiotics](#), with the dosage adjusted for patient's body weight — the name derived from the antibiotics themselves, as many antibiotics have the suffix "-mycin". The Mycin system was also used for the diagnosis of blood clotting diseases.

### **Tasks and Domain**

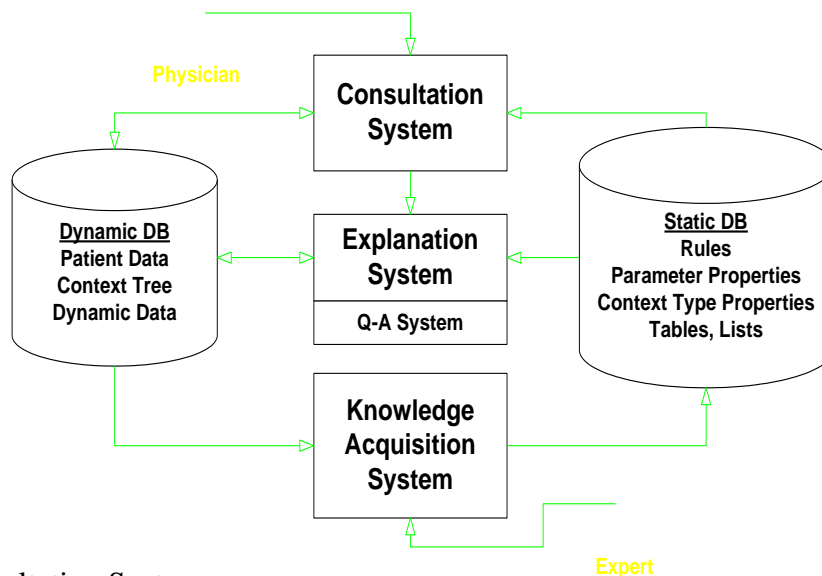
- Disease DIAGNOSIS and Therapy SELECTION
- Advice for non-expert physicians with time considerations and incomplete evidence on:
  - Bacterial infections of the blood
  - Expanded to meningitis and other ailments

### **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

### MYCIN 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

#### Consultation “Control Structure”

- ▶ Goal-directed Backward-chaining Depth-first Tree Search
- ▶ High-level Algorithm:
  1. Determine if Patient has significant infection
  2. Determine likely identity of significant organisms
  3. Decide which drugs are potentially useful
  4. Select best drug or coverage of drugs

#### Static Database

- ▶ Rules

- ▶ Meta-Rules
- ▶ Templates
- ▶ Rule Properties
- ▶ Context Properties
- ▶ Fed from Knowledge Acquisition System

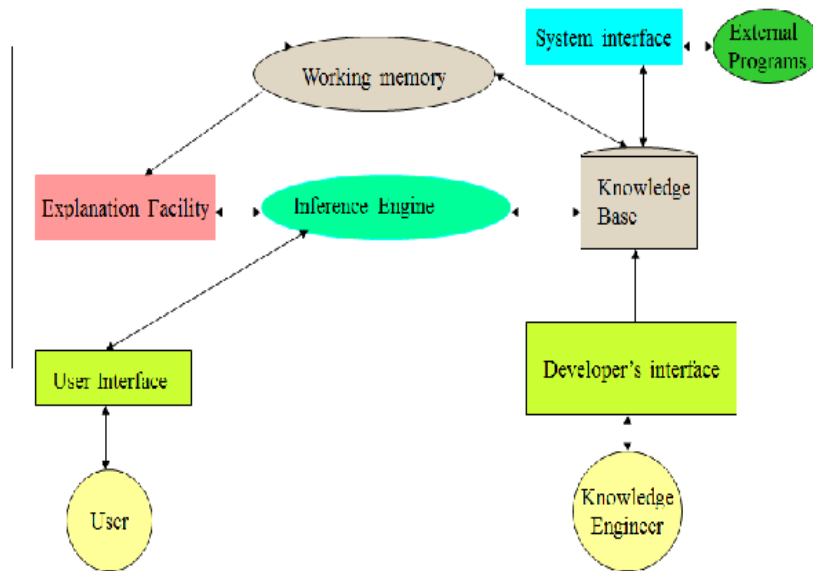
#### Production Rules

- ▶ Represent Domain-specific Knowledge
- ▶ Over 450 rules in MYCIN
- ▶ Premise-Action (If-Then) Form:  
     <predicate function><object><attrib><value>
- ▶ Each rule is completely modular, all relevant context is contained in the rule with explicitly stated premises

### **EXPERT SYSTEM SHELLS**

- MYCIN (Shortliffe, 1976) was developed at Stanford University
- MYCIN took approximately 20 person-years to complete. The system was written in INTERLISP, a dialect of the LISP programming language.
- During the work on MYCIN, a large amount of LISP code was written for different modules:
  - Knowledge base
  - Inference engine
  - Working memory
  - Explanation facility
  - End-user interface
- Toward the end of the project, the MYCIN developers realized that because the knowledge on infectious diseases was separate from its control, then the code written for the other modules should be portable to other applications.
- By removing the knowledge about infectious blood diseases, a system known as EMYCIN (van Melle, 1979) was formed. EMYCIN is a domain-independent version of MYCIN that contains all of MYCIN except its knowledge about infectious blood disease.
- EMYCIN facilitated the development of other expert systems, such as PUFF (Aikens et al., 1983), an application for the diagnosis of pulmonary problems.
- PUFF was produced in about 5 person-years.

### **EXPERT SYSTEM SHELL ARCHITECTURE**



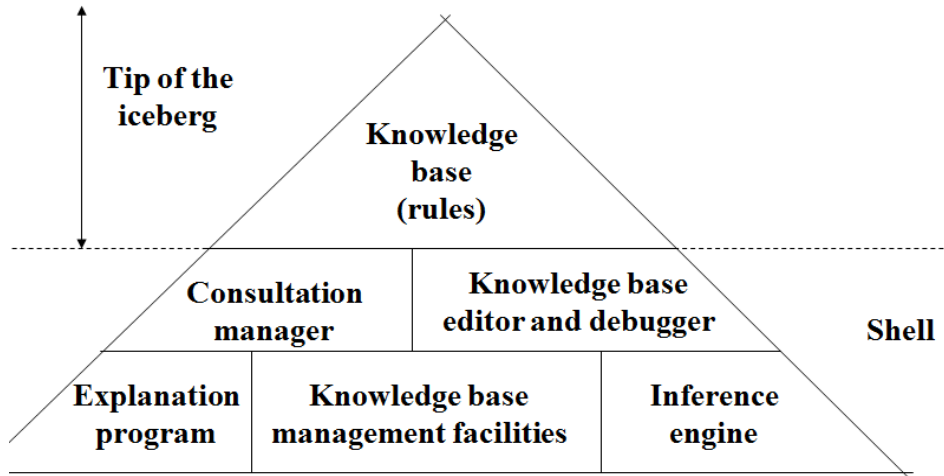
### Shells and Environments

#### **Expert systems components**

- 1. Knowledge acquisition subsystems**
- 2. Inference engine**
- 3. Explanation facility**
- 4. Interface subsystem**
- 5. Knowledge base management facility**
- 6. Knowledge base**

#### Shell Concept for Building Expert Systems (Figure 14.3)

# Shell Concept for Building Expert Systems (Figure 14.3)



Decision Support Systems and Intelligent Systems, Efraim Turban and Jay E. Aronson  
6th ed, Copyright 2001, Prentice Hall, Upper Saddle River, NJ

## Classification of Tools

### Rule-Based Shells

- Exsys
- InstantTea
- XpertRule KBS
- G2
- Guru
- K-Vision
- CLIPS
- JESS
- ESTA

## STEPS TO CREATE AN EXPERT SYSTEM

The process of creation of an expert system requires careful planning. It is common to acquire an expert systems tool, i.e., shell, instead of developing the inference engine from the scratch.

The steps involved in the creation of expert system are listed below.

Step 1: Select a domain and a particular task

a) Choose a task that an expert can do well.

b) The performance of the task should be related to both breadth and depth of knowledge.

c) The facts and rules should be stable.

Step 2: Select the expert system shell for implementation

a) Choose the type of inference control required.

b) Choose the type of pattern-matching capability required.

c) Decide whether certainty factors are necessary

d) Start building a prototype system

Step 3: Acquire initial knowledge about the domain and the task

a) Identify the knowledge experts

b) Select particular problems associated with each task

c) Obtain, record and cross-check factual knowledge from both reference material and experts

d) Obtain and record task-related rules from the experts and confirm them as far as possible

e) Prepare a set of test cases

Step 4: Encode the knowledge using the appropriate representation

a) Factual knowledge

b) Inference knowledge

c) Control knowledge

Step 5: Execute and test the knowledge

a) Evaluate the test cases

b) Be alert for problems with consistency and completeness

Step 6: Refine the current knowledge and acquire additional knowledge

a) Revise the rules as necessary

b) Modify any facts that need revision

c) Augment the system with information on additional domain tasks and test again

d) Repeat as often as necessary

Step 7: Complete any necessary interface code

a) Demonstrate the system

b) Make the system user-friendly

Step 8: Document the expert system

a) Provide on-line and hard-copy documentation as necessary

b) Document the consultation portion especially well

c) Document the knowledge portion to the degree necessary