UNIT IV

1. Representing knowledge using rules: Procedural Vs Declarative knowledge

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

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

Procedural vs Declarative Knowledge

Procedural Knowledge

☞ A representation in which the control information that is necessary to use the knowledge is embedded in the knowledge itself for e.g. computer programs, directions, and recipes;

these indicate specific use or implementation;

☞ The real difference between declarative and procedural views of knowledge lies in where control information reside.

For example, consider the following

Man (Marcus)

Man (Caesar)

Person (Cleopatra)

$\forall x$*: Man(x)* → *Person(x)*

Now, try to answer the question. ?Person(y)

The knowledge base justifies any of the following answers.

Y=Marcus

Y=Caesar

Y=Cleopatra

☞ We get more than one value that satisfies the predicate.

☞ If only one value needed, then the answer to the question will depend on the order in which the assertions examined during the search for a response.

☞ If the assertions declarative then they do not themselves say anything about how they will be examined. In case of procedural representation, they say how they will examine.

Declarative Knowledge

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

2. Logic programming

Logic Programming – Representing Knowledge Using Rules

- Logic programming is a programming paradigm in which logical assertions viewed as programs.
- These are several logic programming systems, PROLOG is one of them.
- ***A PROLOG program consists of several logical assertions where each is a horn clause***

  i.e. a clause with at most one positive literal.
- Ex : P, P V Q, P → Q
- The facts are represented on Horn Clause for two reasons.
  1. Because of a uniform representation, a simple and efficient interpreter can write.
  2. The logic of Horn Clause decidable.
- Also, The first two differences are the fact that PROLOG programs are actually sets of

  Horn clause that have been transformed as follows:-
  1. If the Horn Clause contains no negative literal then leave it as it is.

2. Also, Otherwise rewrite the Horn clauses as an implication, combining all of the negative literals into the antecedent of the implications and the single positive literal into the consequent.

- ❖ Moreover, This procedure causes a clause which originally consisted of a disjunction of literals (one of them was positive) to be transformed into a single implication whose antecedent is a conjunction universally quantified.
- ❖ But when we apply this transformation, any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent are still universally quantified.

  For example the PROLOG clause P(x): – Q(x, y) is equal to logical expression $\forall x: \exists y: Q(x, y) \rightarrow P(x)$.
- ❖ The difference between the logic and PROLOG representation is that the PROLOG interpretation has a fixed control strategy. And so, the assertions in the PROLOG program define a particular search path to answer any question.
- ❖ But, the logical assertions define only the set of answers but not about how to choose among those answers if there is more than one.

Consider the following example:

1. Logical representation

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

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

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

   poodle (fluffy)

2. Prolog representation

   apartmentpet (x) : pet(x), small (x)

   pet (x): cat (x)

   pet (x): dog(x)

   dog(x): poodle (x)

   small (x): poodle(x)

   poodle (fluffy)

3. Forward Vs Backward reasoning

<u>Forward versus Backward Reasoning</u>

A search procedure must find a path between initial and goal states.

There are two directions in which a search process could proceed.

The two types of search are:

1. Forward search which starts from the start state
2. Backward search that starts from the goal state

The production system views the forward and backward as symmetric processes.

Consider a game of playing 8 puzzles. The rules defined are

Square 1 empty and square 2 contains tile n. →

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

Square 1 empty Square 4 contains tile n. →

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

We can solve the problem in 2 ways:

1. Reason forward from the initial state:

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

2. Reasoning backward from the goal states:

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

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

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

Example 1 of Forward versus Backward Reasoning

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

Example 2 of Forward versus Backward Reasoning

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

Example 3 of Forward versus Backward Reasoning

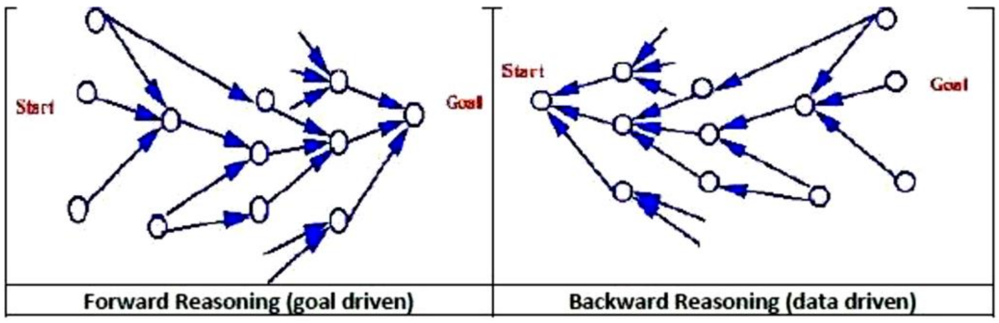
❖ The third factor is nothing but deciding whether the reasoning process can justify its reasoning. If it justifies then it can apply. For example, doctors are usually unwilling to accept any advice from diagnostics process because it cannot explain its reasoning.

Example 4 of Forward versus Backward Reasoning

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

Combining Forward and Backward Reasoning

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


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

Forward versus Backward Reasoning

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

4. Matching

Conflict resolution

The result of the matching process is a list of rules whose antecedents

- Preferences based on rules:
  - Specificity of rules
  - Physical order of ruIes
- Preferences based on objects:
  - Importance of objects

    - Position of objects
- Preferences based on action:
    - Evaluation of states

The individual preconditions of the rule can be matched

Son(x,y) A son(y,z) $\longrightarrow$ grandparent(x,z)

Can be matched, but not in a manner that satisfies the constraint imposed by the variable y.

5. Control knowledge.

- Knowledge about which paths are most likely to find the goal state is often called search control knowledge
    - Which states are more preferable to others.
    - Knowledge about which rules to apply in a given situation.
    - Knowledge about the order in which to pursue sub goals.
    - Knowledge about useful sequence of rules to apply.

  Search control knowledge = Meta knowledge

  1. Long term memory $\longrightarrow$ Rules
  2. Short term memory $\longrightarrow$ Working memory

UNIT V

1. Game playing

- **Computer programs which play 2-player games**
  - game-playing as search
  - with the complication of an opponent
- **General principles of game-playing and search**
  - evaluation functions
  - minimax principle
  - alpha-beta-pruning
  - heuristic techniques
- **Status of Game-Playing Systems**
  - in chess, checkers, backgammon, Othello, etc, computers routinely defeat leading world players
- **Applications?**
  - think of "nature" as an opponent
  - economics, war-gaming, medical drug treatment

Solving 2-players Games

- **Two players, perfect information**
- **Examples:**
  - e.g., chess, checkers, tic-tac-toe
- **configuration of the board = unique arrangement of "pieces"**
- **Statement of Game as a Search Problem**
  - **States** = board configurations
  - **Operators** = legal moves
  - **Initial State** = current configuration
  - **Goal** = winning configuration
  - **payoff function** = gives numerical value of outcome of the game
- **A working example: Grundy's game**
  - Given a set of coins, a player takes a set and divides it into two unequal sets. The player who plays last, looses.

Grundy's game - special case of nim

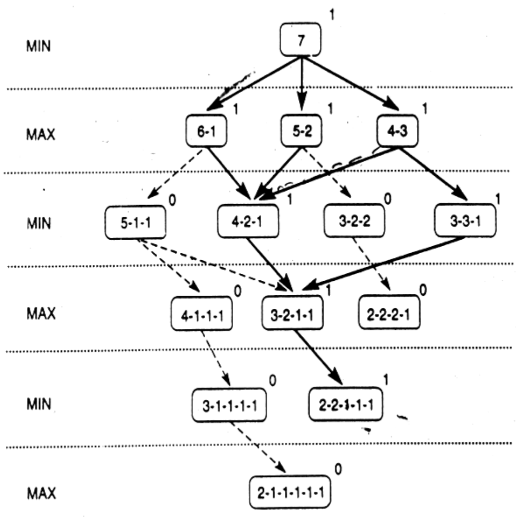
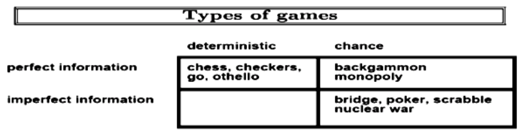


Figure 1. Exhaustive minimax for the game of nim. Bold lines indicate forced win for MAX. Each node is marked with its derived value (0 or 1) under minimax.

Games vs. search problems

- "Unpredictable" opponent → specifying a move for every possible opponent reply
- Time limits → unlikely to find goal, must approximate

Types of Games



Types of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war

Game Trees

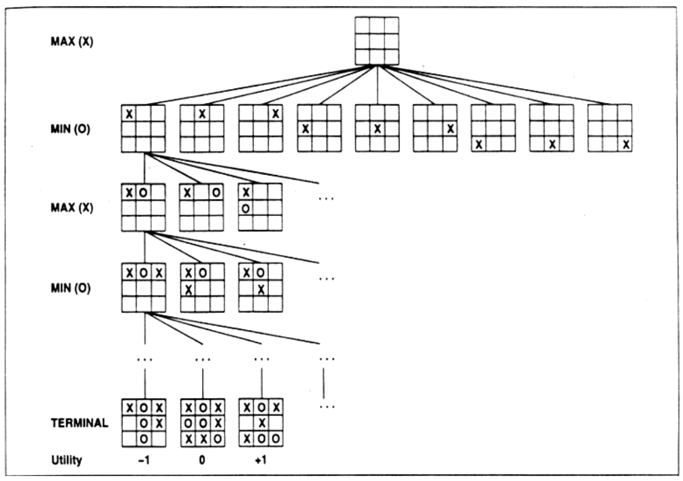


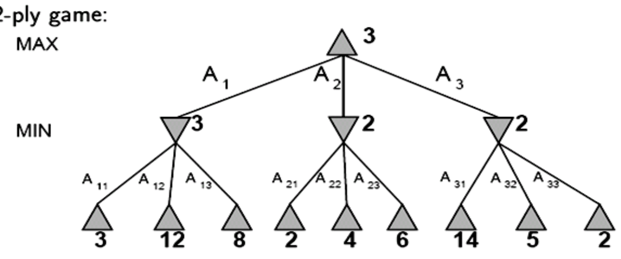
Figure 2. A (Partial) search tree for the game of Tic-Tac-Toe. The top node is the initial state, and MAX moves first, placing an X in some square. We show part of the search tree, giving alternating moves by MIN (O) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

2. The minimax search procedure

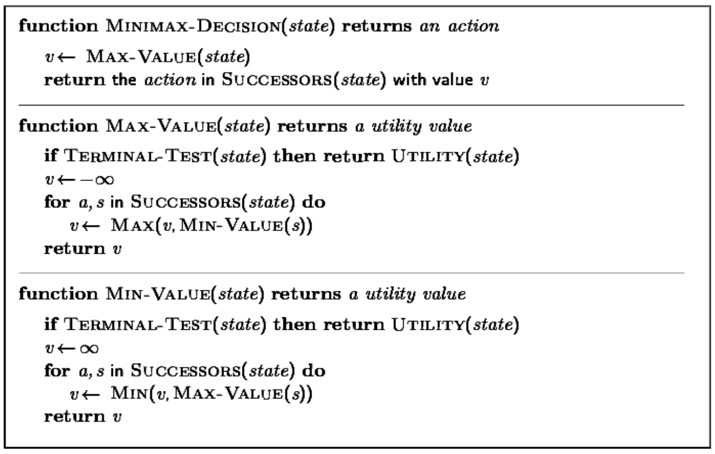
Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest *minimax value*
= best achievable payoff against best play

E.g., 2-ply game:



Minimax algorithm



```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

An optimal procedure: The Min-Max method

- **Designed to find the optimal strategy for Max and find best move:**
  - 1. Generate the whole game tree to leaves
  - 2. Apply utility (payoff) function to leaves
  - 3. Back-up values from leaves toward the root:
    - a Max node computes the max of its child values
    - a Min node computes the Min of its child values
  - 4. When value reaches the root: choose max value and the corresponding move.

Properties of minimax

- **Complete? Yes (if tree is finite)**
- **Optimal? Yes (against an optimal opponent)**
- **Time complexity? O(b^m)**

- **<u>Space complexity?</u> O(bm) (depth-first exploration)**

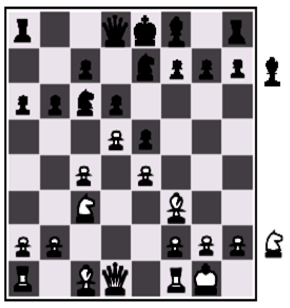
- **For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games**
  → exact solution completely infeasible

  - Chess:
    - b ~ 35 (average branching factor)
    - d ~ 100 (depth of game tree for typical game)
    - $b^d \sim 35^{100} \sim 10^{154}$ nodes!!
  - Tic-Tac-Toe
    - ~5 legal moves, total of 9 moves
    - 5^9 = 1,953,125
    - 9! = 362,880 (Computer goes first)
    - 8! = 40,320 (Computer goes second)

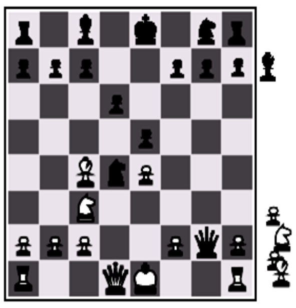
<u>Static (Heuristic) Evaluation Functions</u>

- **An Evaluation Function:**
  - estimates how good the current board configuration is for a player.
  - Typically, one figures how good it is for the player, and how good it is for the opponent, and subtracts the opponents score from the players
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces
- **Typical values from -infinity (loss) to +infinity (win) or [-1, +1].**
- **If the board evaluation is X for a player, it's -X for the opponent**
- **Example:**
  - Evaluating chess boards,
  - Checkers
  - Tic-tac-toe

<u>Evaluation functions</u>


Black to move

White slightly better


White to move

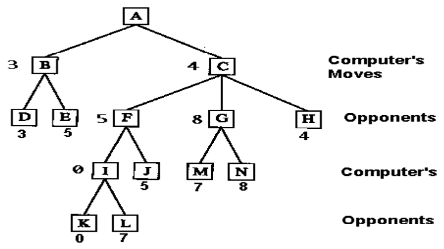
Black winning

For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s)$ = (number of white queens) – (number of black queens), etc.

Deeper Game Trees



Applying MiniMax to tic-tac-toe

- **The static evaluation function heuristic**

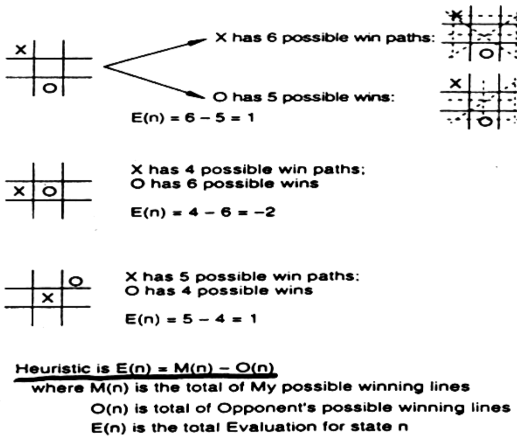

X has 6 possible win paths:
O has 5 possible wins:
E(n) = 6 – 5 = 1
X has 4 possible win paths;
O has 6 possible wins
E(n) = 4 – 6 = –2
X has 5 possible win paths;
O has 4 possible wins
E(n) = 5 – 4 = 1
Heuristic is E(n) = M(n) – O(n)
where M(n) is the total of My possible winning lines
O(n) is total of Opponent's possible winning lines
E(n) is the total Evaluation for state n

Figure. Heuristic measuring conflict applied to states of tic-tac-toe

Alpha Beta Procedure

- **Idea:**
  - Do Depth first search to generate partial game tree,
  - Give static evaluation function to leaves,
  - compute bound on internal nodes.
- **Alpha, Beta bounds:**
  - Alpha value for Max node means that Max real value is at least alpha.
  - Beta for Min node means that Min can guarantee a value below Beta.
- **Computation:**
  - Alpha of a Max node is the maximum value of its seen children.
  - Beta of a Min node is the lowest value seen of its child node.

3. Expert System

Expert system = knowledge + problem-solving methods. ... A knowledge base that captures the domain-specific knowledge and an inference engine that consists of algorithms for manipulating the knowledge represented in the knowledge base to solve a problem presented to the system.

Expert systems (ES) are one of the prominent research domains of AI. It is introduced by the researchers at Stanford University, Computer Science Department.

DEFINITION

An expert system is a computer program that simulates the judgement and behavior of a human or an organization that has expert knowledge and experience in a particular field.

What are 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.

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

They are incapable of −

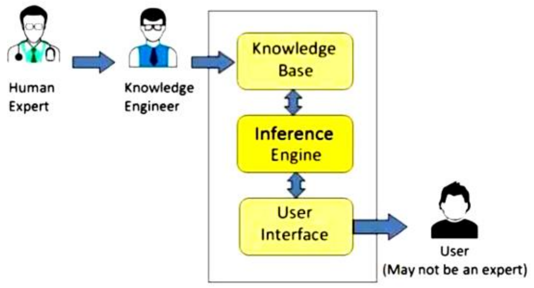
- Substituting human decision makers
- Possessing human capabilities
- Producing accurate output for inadequate knowledge base
- Refining their own knowledge

Components of Expert Systems

The components of ES include −

- Knowledge Base
- Inference Engine
- User Interface

Let us see them one by one briefly



Knowledge Base

It contains domain-specific and high-quality knowledge. Knowledge is required to exhibit intelligence. The success of any ES majorly depends upon the collection of highly accurate and precise knowledge.

What is Knowledge?

The data is collection of facts. The information is organized as data and facts about the task domain. Data, information, and past experience combined together are termed as knowledge.

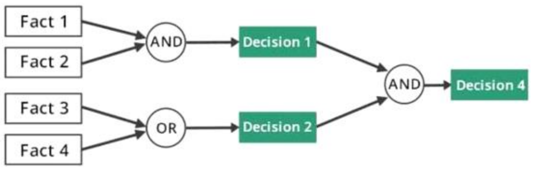
To recommend a solution, the Inference Engine uses the following strategies −

- Forward Chaining
- Backward Chaining

Forward Chaining

It is a strategy of an expert system to answer the question, "What can happen next?" Here, the Inference Engine follows the chain of conditions and derivations and finally deduces the outcome. It considers all the facts and rules, and sorts them before concluding to a solution. This strategy is followed for working on conclusion, result, or effect.

For example, prediction of share market status as an effect of changes in interest rates.

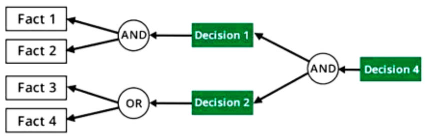


Backward Chaining

With this strategy, an expert system finds out the answer to the question, "Why this happened?" On the basis of what has already happened, the Inference Engine tries to find out which conditions cou

ld have happened in the past for this result. This strategy is followed for finding out cause or reason.

For example, diagnosis of blood cancer in humans.



User Interface

User interface provides interaction between user of the ES and the ES itself. It is generally Natural Language Processing so as to be used by the user who is well-versed in the task domain. The user of the ES need not be necessarily an expert in Artificial Intelligence. It explains how the ES has arrived at a particular recommendation. The explanation may appear in the following forms

- Natural language displayed on screen.
- Verbal narrations in natural language.
- Listing of rule numbers displayed on the screen.

The user interface makes it easy to trace the credibility of the deductions.

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.

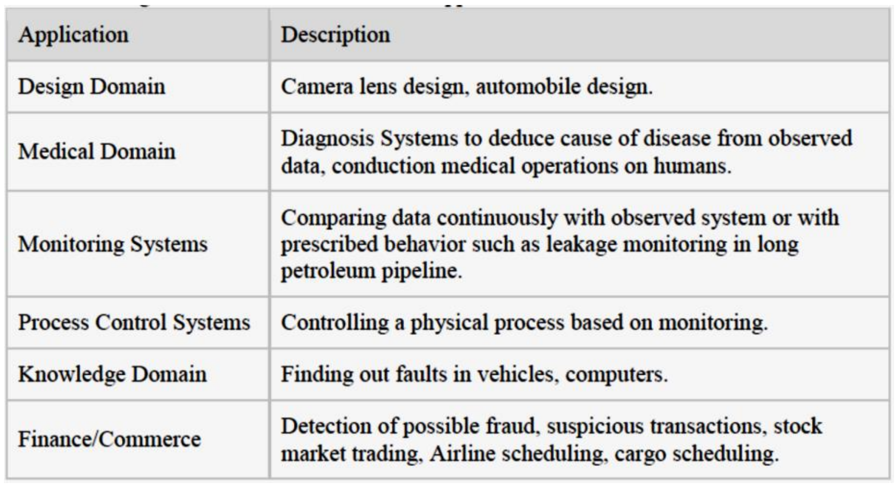
Expert Systems Limitations

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 or with prescribed behavior such as leakage monitoring in 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 transactions, stock market trading, Airline scheduling, cargo scheduling.

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 LISt Programming (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.

4. Perception and Action

Perception

Perception is the process of acquiring, interpreting, selecting, and organizing sensory information

Organisms in the real world have to do two basic things in order to survive:

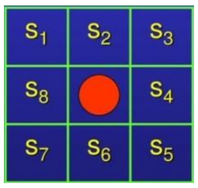
- They have to gather information about their environment (perception) and
- Based on this information the have to manipulate their environment (including themselves) in a way that is advantageous to them (action).

The action in turn may cause a change in the organism's perception, which can lead to a different type of action.

We call this the perception-action cycle

The robot is supposed to find a cell next to a boundary or object and then follow that boundary forever.

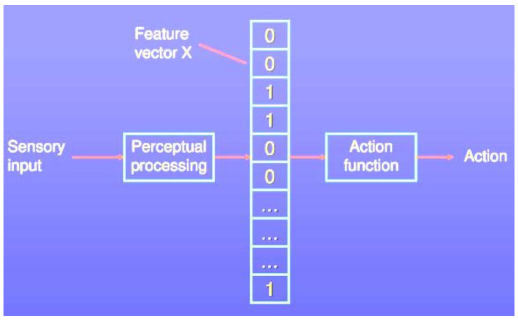
As we said, the robot can perceive the state of its neighboring cells:



The robot can move to a free adjacent cell in its column or row. Consequently, there are four possible actions that it can take:

- North: moves the robot one cell up
- East: moves the robot one cell right
- South: moves the robot one cell down
- West: moves the robot one cell to the left

- Now that we specified the robot's capabilities, its environment, and its task, we need to give "life" to the robot.
- In other words, we have to specify a function that maps sensory inputs to movement actions so that the robot will carry out its task.
- Since we do not want the robot to remember or learn anything, one such function would be sufficient.
- However, it is useful to decompose it in the following way



The functional decomposition has two advantages:

- Multiple action functions can be added that receive the same feature vector as their input,
- It is possible to add an internal state to the system to implement memory and learning.

Perception

- For the robot task, there are four binar -valued features of the sensory values that are useful for computing an appropriate action X1, X2, X,3, X4
- Perceptual processing might occasionally give erroneous, ambiguous, or incomplete information about the robot's environment
  - ♦ Such errors might evoke inappropriate actions
- For robots with more complex sensors and tasks, designing appropriate perceptual processing can be challenging

Action

- Specifying a function that selects the appropriate boundary-following action

$$\begin{cases} \text{if } x_1=1 \text{ and } x_2=0, & \textit{move east} \\ \text{if } x_2=1 \text{ and } x_3=0, & \textit{move south} \\ \text{if } x_3=1 \text{ and } x_4=0, & \textit{move west} \\ \text{if } x_4=1 \text{ and } x_1=0, & \textit{move north} \end{cases}$$

♦ None of the features has value 1, the robot can move in any direction until it encounters a boundary