Overview of Artificial Intelligence and Expert Systems

Dr. Sudeep Marwaha

Intelligence should be placed in the context of biology: Intelligence connects perception to action to help an organism survive. Intelligence is computation in the service of life, just as metabolism is chemistry in the service of life. Intelligence does not imply perfect understanding; every intelligent being has limited perception, memory, and computation. Many points on the spectrum of intelligence-versus-cost are viable, from insects to humans.

AI seeks to understand the computations required for intelligent behavior and to produce computer systems that exhibit intelligence. Aspects of intelligence studied by AI include perception, motor control, communication using human languages, reasoning, planning, learning, and memory. AI seeks to understand the working of the mind in mechanistic terms, just as medicine seeks to understand the working of the body in mechanistic terms. According to Marvin Minsky “The mind is what the brain does”. The strong AI position is that any aspect of human intelligence could, in principle, be mechanized. The computer systems can be made more intelligent by using following concepts:

- Autonomy to perform tasks that currently require human operators without human intervention or monitoring.
- Flexibility in dealing with variability in the environment.
- Ease of use: computers that are able to understand what the user wants from limited instructions in natural languages.
- Learning from experience

Following are some of the important areas in which AI is used:

- Perception
- Machine vision
- Speech understanding
- Touch (tactile or haptic) sensation
- Robotics
- Natural Language Processing
- Natural Language Understanding
- Speech Understanding
- Language Generation
- Machine Translation
- Planning
- Expert Systems
- Machine Learning
- Theorem Proving
- Symbolic Mathematics
- Game Playing
Characteristics of A.I. Programs

Symbolic Reasoning: reasoning about objects represented by symbols, and their properties and relationships, not just numerical calculations. AI programs often do some numerical calculation, but focus Examples of symbolic processing:

- Understanding English:
  (show me a good chinese restaurant in los altos)
- Reasoning based on general principles:
  if: the patient is male
  then: the patient is not pregnant
- Symbolic mathematics:
  If y = m*x+b, what is the derivative of y with respect to x?
- on reasoning with symbols that represent objects and relationships in the real world.

Knowledge: General principles are stored in the program and used for reasoning about novel situations.
Flexible Control: Direction of processing can be changed by changing facts in the environment.

Knowledge Representation
It is necessary to represent the computer's knowledge of the world by some kind of data structures in the machine's memory. Traditional computer programs deal with large amounts of data that are structured in simple and uniform ways. A.I. programs need to deal with complex relationships, reflecting the complexity of the real world. The notion of knowledge representation can best be understood in terms of five distinct roles it plays, each crucial to the task at hand:

- A knowledge representation (KR) is most fundamentally a surrogate, a substitute for the thing itself, used to enable an entity to determine consequences by thinking rather than acting, i.e., by reasoning about the world rather than taking action in it.
- It is a set of ontological commitments, i.e., an answer to the question: In what terms should I think about the world?
- It is a fragmentary theory of intelligent reasoning, expressed in terms of three components: (i) the representation's fundamental conception of intelligent reasoning; (ii) the set of inferences the representation sanctions; and (iii) the set of inferences it recommends.
- It is a medium for pragmatically efficient computation, i.e., the computational environment in which thinking is accomplished. One contribution to this pragmatic efficiency is supplied by the guidance a representation provides for organizing information so as to facilitate making the recommended inferences.
- It is a medium of human expression, i.e., a language in which we say things about the world.
The above roles provide a framework useful for characterizing a wide variety of representations. The fundamental "mindset" of a representation can be captured by understanding how it views each of the roles, and that doing so reveals essential similarities and differences.

**Production System**

A production system (or production rule system) is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behavior. A production system provides the mechanism necessary to execute productions in order to achieve some goal for the system. Productions consist of two parts: a sensory precondition (or "IF" statement) and an action (or "THEN"). If a production's precondition matches the current state of the world, then the production is said to be triggered. If a production's action is executed, it is said to have fired. A production system also contains a database, sometimes called working memory, which maintains data about current state or knowledge, and a rule interpreter. The rule interpreter must provide a mechanism for prioritizing productions when more than one is triggered.

Rule interpreters generally execute a forward chaining algorithm for selecting productions to execute to meet current goals, which can include updating the system's data or beliefs. The condition portion of each rule (left-hand side or LHS) is tested against the current state of the working memory.

In idealized or data-oriented production systems, there is an assumption that any triggered conditions should be executed: the consequent actions (right-hand side or RHS) will update the agent's knowledge, removing or adding data to the working memory. The system stops processing either when the user interrupts the forward chaining loop; when a given number of cycles has been performed; when a "halt" RHS is executed, or when no rules have true LHSs. Real-time and expert systems, in contrast, often have to choose between mutually exclusive productions --- since actions take time, only one action can be taken, or (in the case of an expert system) recommended. In such systems, the rule interpreter, or inference engine, cycles through two steps: matching production rules against the database, followed by selecting which of the matched rules to apply and executing the selected actions.

Production systems may vary on the expressive power of conditions in production rules. Accordingly, the pattern matching algorithm which collects production rules with matched conditions may range from the naive -- trying all rules in sequence, stopping at the first match -- to the optimized, in which rules are "compiled" into a network of inter-related conditions.

The latter is illustrated by the RETE algorithm, designed by Charles L. Forgy in 1983, which is used in a series of production systems, called OPS and originally developed at Carnegie Mellon University culminating in OPS5 in the early eighties. OPS5 may be viewed as a full-fledged programming language for production system programming.

Production systems may also differ in the final selection of production rules to execute, or fire. The collection of rules resulting from the previous matching algorithm is called the conflict set, and the selection process is also called a conflict resolution strategy. Here again, such strategies may vary from the simple -- use the order in which production rules were written; assign weights or priorities to production rules and sort the conflict set...
accordingly -- to the complex -- sort the conflict set according to the times at which production rules were previously fired; or according to the extent of the modifications induced by their RHSs. Whichever conflict resolution strategy is implemented, the method is indeed crucial to the efficiency and correctness of the production system.

**Logic for Artificial Intelligence**

Mathematical logic is an important area of AI:

Logic is one of the major knowledge representation and reasoning methods. Logic serves as a standard of comparison for other representation and reasoning methods. Logic has a sound mathematical basis. The PROLOG language is based on logic. Those who fail to learn logic are doomed to reinvent it.

The form of logic most commonly used in AI is First-Order Predicate Calculus (FOPC or just PC). Facts can also be represented in a logical formalism. In principle, logic and semantic network representations can be equivalent.

\[(\text{huey huyey-457})\]

\[(\text{all x (if (huey x) (helicopter x)))}\]

\[(\text{all x (if (huey x) (payload x 4000)))}\]

\[(\text{all x (if (helicopter x) (can-fly x)))}\]

Mathematical logic formalizes certain kinds of reasoning in terms of operations on mathematical formulas. It is important for people working in A.I. to know logic for several reasons:

Theory: Logic has a sound mathematical foundation; things can be proved about it.
Applications: For certain classes of applications (e.g., proving correctness of programs) logic is the representation of choice.
Comparison with Other Methods: Other representation methods are often reducible to logic. Knowing logic helps in understanding other methods and may help prevent reinvention of old techniques.

**Semantic Networks**

A semantic network system represents knowledge by nodes and labeled arcs among nodes. A semantic net is really just a graph, where the nodes in the graph represent concepts, and the arcs represent binary relationships between concepts. The most important relations between concepts are subclass relations between classes and subclasses, and instance relations between particular objects and their parent class. However, any other relations are allowed, such as has-part, colour etc.

This network represents the fact that Huey and Huey-457 are helicopters, that helicopters have capability of flying. The subclass relations define a class hierarchy. The subclass and instance relations may be used to derive new information which is not explicitly represented. They inherit information from their parent classes. Semantic networks normally allow efficient inheritance-based inferences using special purpose algorithms.
Semantic nets are fine at representing relationships between two objects - but what if we want to represent a relation between three or more objects? Say we want to represent the fact that "John gives Mary the book" This might be represented in logic as gives(john, mary, book2) where book2 represents the particular book we are talking about. However, in semantic networks we have to view the fact as representing a set of binary relationships between a "giving" event and some objects:

When semantic networks became popular in the 1970s there was much discussion about what the nodes and relations really meant. People were using them in subtly different ways, which led to much confusion. For example, a node such as elephant might be used to represent the class of all elephants or just a typical elephant. Saying that an elephant has part head could mean that an every elephant has some particular head, that there exists some elephant that has a head, or (more reasonably in this case) that they all have some object belonging to the class head. Depending on what interpretation you choose for your nodes and links, different inferences are valid. For example, if it's just a typical elephant, then Clyde may have properties different from general elephant properties (such as being pink and not grey). The simplest way to interpret the class nodes is as denoting sets of objects. So, an elephant node denotes the set of all elephants. Nodes such as Clyde and Nellie denote individuals. So the instance relationship can be defined in terms of set membership, while the subclass relation can be defined in terms of a subset relation - the set of all elephants is a subset of the set of all mammals. Saying that elephants are grey means (in the simple model) that every individual in the set of elephants is grey (so Clyde can't be pink). If we interpret networks in this way we have the advantage of a clear, simple semantics, but the disadvantage of a certain lack of flexibility - maybe Clyde is pink!

In the debate about semantic nets, people were also concerned about their representational adequacy (ie, what sort of facts they were capable of representing). Things that are easy to represent in logic (such as "every dog in town has bitten the constable") are hard to represent in nets (at least, in a way that has a clear and well-defined interpretation). Techniques were developed to allow such things to be represented, which involved partitioning the net into sections, and having introducing a special relationship. These techniques didn't really catch on, so we won't go into them here, but can be found in many AI textbooks.
To summarise, nets allow us to simply represent knowledge about an object that can be expressed as binary relations. Subclass and instance relations allow us to use inheritance to infer new facts/relations from the explicitly represented one. However, early nets didn't have a very clear semantics (ie, it wasn't clear what the nodes and links really meant). It was difficult to use nets in a fully consistent and meaningful manner, and still use them to represent what you wanted to represent. Techniques evolved to get round this, but they are quite complex, and seem to partly remove the attractive simplicity of the initial idea.

**Frames**

Frames can also be regarded as an extension to Semantic nets. Indeed it is not clear where the distinction between a semantic net and a frame ends. Semantic nets initially we used to represent labelled connections between objects. As tasks became more complex the representation needs to be more structured. The more structured the system it becomes more beneficial to use frames. A frame is a collection of attributes or slots and associated values that describe some real world entity. Frames on their own are not particularly helpful but frame systems are a powerful way of encoding information to support reasoning. Set theory provides a good basis for understanding frame systems. Each frame represents:

- a class (set), or
- an instance (an element of a class).

Consider the example first:

<table>
<thead>
<tr>
<th>Class</th>
<th>ISA</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>isa: Mammal</td>
<td></td>
</tr>
<tr>
<td>Adult-Male</td>
<td>isa: Person</td>
<td></td>
</tr>
<tr>
<td>Rugby-Player</td>
<td>isa: Adult-Male</td>
<td></td>
</tr>
<tr>
<td>Rugby-Team</td>
<td>isa: Team</td>
<td></td>
</tr>
<tr>
<td>Mike-Hall</td>
<td>instance: Back</td>
<td></td>
</tr>
</tbody>
</table>

Consider the example first:

- Person
  - isa: Mammal
  - Cardinality: ...

- Adult-Male
  - isa: Person
  - Cardinality: ...

- Rugby-Player
  - isa: Adult-Male
  - Height: ...
  - Weight: ...
  - Position: ...
  - Team: ...
  - Team-Colours: ...

- Back
  - isa: Rugby-Player
  - Cardinality: ...
  - Tries: ...

- Mike-Hall
  - instance: Back
  - Height: 6-0
  - Position: Centre
  - Team: Cardiff-RFC
  - Team-Colours: Black/Blue

- Rugby-Team
  - isa: Team
  - Cardinality: ...
  - Team-size: 15
  - Coach: ...
Here the frames Person, Adult-Male, Rugby-Player and Rugby-Team are all classes and the frames Robert-Howley and Cardiff-RFC are instances. Following needs to be noted:

- The isa relation is in fact the subset relation.
- The instance relation is in fact element of.
- The isa attribute possesses a transitivity property. This implies: Robert-Howley is a Back and a Back is a Rugby-Player who in turn is an Adult-Male and also a Person.
- Both isa and instance have inverses which are called subclasses or all instances.
- There are attributes that are associated with the class or set such as cardinality and on the other hand there are attributes that are possessed by each member of the class or set.

**Distinction Between Sets And Instances**

It is important that this distinction is clearly understood. Cardiff-RFC can be thought of as a set of players or as an instance of a Rugby-Team. If Cardiff-RFC were a class then

- its instances would be players
- it could not be a subclass of Rugby-Team otherwise its elements would be members of Rugby-Team which we do not want.

Instead we make it a subclass of Rugby-Player and this allows the players to inherit the correct properties enabling us to let the Cardiff-RFC to inherit information about teams. This means that Cardiff-RFC is an instance of Rugby-Team.

BUT There is a problem here:

- A class is a set and its elements have properties.
- We wish to use inheritance to bestow values on its members.
- But there are properties that the set or class itself has such as the manager of a team.

This is why we need to view Cardiff-RFC as a subset of one class players and an instance of teams. We seem to have a CATCH 22. Solution: MetaClasses : A metaclass is a special class whose elements are themselves classes.

Now consider our rugby teams as:
The basic metaclass is `Class`, and this allows us to

- define classes which are instances of other classes, and (thus)
- inherit properties from this class.

Inheritance of default values occurs when one element or class is an instance of a class.

**Slots as Objects**

How can we to represent the following properties in frames?

- Attributes such as `weight`, `age` be attached and make sense.
- Constraints on values such as `age` being less than a hundred
- Default values
- Rules for inheritance of values such as children inheriting parent's names
- Rules for computing values
- Many values for a slot.
A slot is a relation that maps from its domain of classes to its range of values. A relation is a set of ordered pairs so one relation is a subset of another. Since slot is a set the set of all slots can be represent by a metaclass called Slot, say. Consider the following:

**SLOT**

<table>
<thead>
<tr>
<th>isa:</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance:</td>
<td>Class</td>
</tr>
<tr>
<td>domain:</td>
<td></td>
</tr>
<tr>
<td>range:</td>
<td></td>
</tr>
<tr>
<td>range-constraint:</td>
<td></td>
</tr>
<tr>
<td>definition:</td>
<td></td>
</tr>
<tr>
<td>default:</td>
<td></td>
</tr>
<tr>
<td>to-compute:</td>
<td></td>
</tr>
<tr>
<td>single-valued:</td>
<td></td>
</tr>
</tbody>
</table>

**Coach**

| instance: | SLOT |
| domain:    | Rugby-Team |
| range:     | Person |
| range-constraint: | \( \lambda x. (\text{experience } x.\text{manager}) \) |
| default:   |         |
| single-valued: | TRUE |

**Colour**

| instance: | SLOT |
| domain:    | Physical-Object |
| range:     | Colour-Set |
| single-valued: | FALSE |

**Team-Colours**

| instance: | SLOT |
| isa:      | Colour |
| domain:   | team-player |
| range:    | Colour-Set |
| range-constraint: | not Pink |
| single-valued: | FALSE |

**Position**

| instance: | SLOT |
| domain:   | Rugby-Player |
| range:    | \{ Back, Forward, Reserve \} |
| to-compute: | \( \lambda x. \text{position} \) |
| single-valued: | TRUE |

**NOTE** the following:

- Instances of SLOT are slots
- Associated with SLOT are attributes that each instance will inherit.
- Each slot has a domain and range.
- Range is split into two parts one the class of the elements and the other is a constraint which is a logical expression if absent it is taken to be true.
• If there is a value for default then it must be passed on unless an instance has its own value.
• The to-compute attribute involves a procedure to compute its value. E.g. in Position where we use the dot notation to assign values to the slot of a frame.
• Transfers through lists other slots from which values can be derived from inheritance.

Search
Search programs find a solution for a problem by trying different sequences of actions (operators) until a solution is found.

Advantage:
Many kinds of problems can be viewed as search problems. To solve a problem using search, it is only necessary to code the operators that can be used; search will find the sequence of actions that will provide the desired result. For example, a program can be written to play chess using search if one knows the rules of chess; it isn't necessary to know how to play good chess.

Disadvantage:
Most problems have search spaces so large that it is impossible to search the whole space. Chess has been estimated to have $10^{120}$ possible games. The rapid growth of combinations of possible moves is called the combinatoric explosion problem. There is no model of the world that is complete, consistent, and computable. Any intelligent system must encounter surprises. Solutions to problems cannot be precomputed; many problems must be solved dynamically, starting from observed data. Flexibility to deal with a variable environment requires search. Ambiguity in interpretation of perceptual data requires search. Interpretation may be locally ambiguous, but global constraints may permit an unambiguous total interpretation. Creativity can result from searching through many possible designs

Search seems to arise in nearly every area of A.I. In part, this is because A.I. attempts to model human intelligence (which is highly parallel) on serial machines.
• Logic: A theorem prover searches for a sequence of proof steps that will prove a desired conclusion.
• Natural Language: A parser searches for the best ways of assigning structure and meaning to sentences that are ambiguous.
• Planning: A planner searches for a sequence of actions that will accomplish a goal.
• Perception: The raw input is often ambiguous; the perception program searches for a consistent set of interpretations of parts of the input.
• Learning: A learning program searches for a compact description of a set of training instances.
• Expert Systems: Search finds rules applicable to the current problem.
• State Space Search
• Combinatoric Explosion Problem
A state space represents a problem in terms of states and operators that change states. A state space consists of:

- A representation of the states the system can be in. In a board game, for example, the board represents the current state of the game.
- A set of operators that can change one state into another state. In a board game, the operators are the legal moves from any given state. Often the operators are represented as programs that change a state representation to represent the new state.
- An initial state.
- A set of final states; some of these may be desirable, others undesirable. This set is often represented implicitly by a program that detects terminal states.

**Tic-Tac-Toe as a State Space**

State spaces are good representations for board games such as Tic-Tac-Toe. The state of a game can be described by the contents of the board and the player whose turn is next. The board can be represented as an array of 9 cells, each of which may contain an X or O or be empty.

State:
Player to move next: X or O.

Board configuration:  

```
X | O |   
---|---|---
   | O |   
---|---|---
X | X |   
```
Operators: Change an empty cell to X or O.
Start State: Board empty; X's turn.
Terminal States:
Three X's in a row; Three O's in a row; All cells full.
Search Tree
The sequence of states formed by possible moves is called a search tree. Each level of the
tree is called a ply.
Since the same state may be reachable by different sequences of moves, the state space
may in general be a graph. It may be treated as a tree for simplicity, at the cost of
duplicating states.

Solving Problems Using Search
- Given an informal description of the problem, construct a formal description as a
  state space:
- Define a data structure to represent the state.
- Make a representation for the initial state from the given data.
- Write programs to represent operators that change a given state representation to a
  new state representation.
- Write program to detect terminal states.

Choose an appropriate search technique:
- How large is the search space?
- How well-structured is the domain?
- What knowledge about the domain can be used to guide the search?

Basic Recursive Algorithm
If the input is a base case, for which the solution is known, return the solution.
Otherwise,
Do part of the problem, or break it into smaller subproblems.
Call the problem solver recursively to solve the subproblems.
Combine the subproblem solutions to form a total solution.
In writing the recursive program:
Write a clear specification of the input and output of the program.
Assume it works already.
Write the program to use the input form and produce the output form.

**Basic Depth-first Search Algorithm**

```lisp
(defun search (state)
  (let (op oplist newstate)
    (if (terminalp state)
      (if (goalp state) '() 'failure)
      (progn
        (setq op (choose-op state))
        (setq newstate
          (funcall op state))
        (setq oplist (search newstate))
        (if (eq oplist 'failure)
          'failure
          (cons op oplist))))))
```

Complications:
We may have to try different operators (newstate might be a dead end). It may not be possible to apply op. Applying op might violate a constraint. We could get into a loop applying op and its inverse. The program continually goes deeper until it reaches a terminal state, which is either a goal or a failure. When the goal is found, search returns '() as its answer. This is an empty list of operators, since no operators are required to reach the goal.

At each level as the search unwinds, the operator used at that level is put onto the front of the operator list using cons. cons adds a new item onto the front of a list:

```
(cons 'a '(b c)) = (A B C)
```

**Recursive Depth-First Search**

Function sss(s prev ops):
If s is a goal, return '() as the answer. This is a list of no operators, since no operators are required to reach the goal state.
If s is a failure, or no operators remain, return 'failure.
If the next operator, op, is applicable to the input state s, compute new by applying it to s.
If new duplicates one of its ancestor states on prev, try the next operator.
If a search for the goal from the new state,

```
(sss new (cons s prev) *ops*)
```

succeeds, return the cons of op onto the front of its operator sequence, opseq.

```
s &rarr new &rarr ... &rarr goal
  op opseq
```

Else, try the next operator.
Else, try the next operator.
Search Order
The excessive time spent in searching is almost entirely spent on failures (sequences of operators that do not lead to solutions). If the computer could be made to look at promising sequences first and avoid most of the bad ones, much of the effort of searching could be avoided.

Blind search methods try operators in some fixed order, without knowing which operators may be more likely to lead to a solution. Such methods can succeed only for small search spaces.

Heuristic search methods use knowledge about the problem domain to choose more promising operators first.

Searches can be classified by the order in which operators are tried: depth-first, breadth-first, bounded depth-first.

![Fig. 4. Depth First Search Tree](image)

**Depth-First Search**
Depth-first search applies operators to each newly generated state, trying to drive directly toward the goal.

**Advantages:**
Low storage requirement: linear with tree depth.
Easily programmed: function call stack does most of the work of maintaining state of the search.

Disadvantages:
May find a sub-optimal solution (one that is deeper or more costly than the best solution).
Incomplete: without a depth bound, may not find a solution even if one exists.

**Breadth-First Search**
Breadth-first search generates new states in the order of their distance from the start state. All states at level i are examined before any states at level i+1 are examined.

**Advantages:**
Guaranteed to find an optimal solution (in terms of shortest number of steps to reach the goal).
Can always find a goal node if one exists (complete).

**Disadvantages:**
High storage requirement: exponential with tree depth.

**Using Heuristics to Guide Search**
We now turn to methods for using heuristic knowledge to make search more efficient. Finding a route from one city to another city is an example of a search problem in which different search orders and the use of heuristic knowledge are easily understood.

**State:** the current city in which the traveler is located.

**Operators:** roads linking the current city to other cities.

**Cost Metric:** the cost of taking a given road between cities (distance, time required, dollar cost, a weighted sum of costs, etc.).

**Heuristic Information:** the search could be guided by the direction of the goal city from the current city, or we could use airline distance as an estimate of the distance to the goal.

**Hill Climbing**
A strategy for climbing a hill in a fog is to move upward. A heuristic that estimates distance to the goal can be used to guide a hill-climbing search. A discrete depth-first search guided by such a heuristic is called greedy best-first search; it can be very efficient. For example, in route finding, hill climbing could be implemented by selecting the next city that is closest to the goal. Unfortunately, hill-climbing sometimes gets into trouble:

- **Local Maxima:**

- **Mesas:**

![Fig. 5. Local Maxima & Mesas](image)

Random restart is one way to recover from local maxima.
Simulated Annealing
Annealing is a technique in which a metal is heated to a high temperature, then allowed to cool slowly; this relieves internal stresses. Simulated annealing [S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", Science vol. 220, no. 4598 (13 May 1983), pp. 671-680.] is analogous to hill-climbing in which it is possible to make some moves that are locally non-optimal.

By analogy with statistical thermodynamics, it is assumed that the goal is to minimize the "energy" $E$ of a system. If a move causes a change $\Delta E$, if $\Delta E \leq 0$, the move is accepted.

If $\Delta E > 0$, the move is accepted with probability $P(\Delta E) = e^{-\Delta E k T}$, where $k$ is the Boltzmann constant and $T$ is temperature. (Compute a random number $r$ in $[0, 1]$ and accept the move if $P(\Delta E) > r$.)

In simulated annealing, an evaluation function $f$ replaces $E$. An artificial "temperature" is created, initially high enough for many moves to be accepted, and progressively reduced.

Use of Simulated Annealing
Simulated annealing is well suited for problems where:
- The number of variables, and thus possible states, is very large. This makes exhaustive search infeasible.
- There is "frustration" : it is not possible to optimize all cost measures simultaneously. Significant improvements from a random starting position are possible. There are many good near-optimal solutions. Hill-climbing is likely to get stuck on local maxima. Computation is proportional to $N$ or a small power of $N$, while finding the exact optimum is often NP-complete.

Knowledge Representation System
A knowledge representation system will include ways to store knowledge, ways to add new knowledge, and ways to query the knowledge. We can think of the interface as being two procedures:

```
Tell(fact)
Ask(question)
```

Tell and Ask will be front-end programs that access a database of facts in some knowledge representation language.

![Fig. 6. Knowledge Representation System](image)
Either Tell or Ask (or both) may do inference:
Ask(question) may do backward inference when the answer is implied but not explicit in the KB.
Tell(fact) may do forward inference to derive additional facts from what it is told.

**Expert Systems**

Expert Systems attempt to capture the knowledge of a human expert and make it available through a computer program. There have been many successful and economically valuable applications of expert systems. Following are some of the benefits of expert systems:

- Reducing skill level needed to operate complex devices.
- Diagnostic advice for device repair.
- Interpretation of complex data.
- “Cloning” of scarce expertise.
- Capturing knowledge of expert who is about to retire.
- Combining knowledge of multiple experts.
- Intelligent training.

The basic idea behind expert system is to simply transfer expertise that is the vast body of task-specific knowledge from a human to a computer. Knowledgebase and inference engine are the two most important components of an expert system. The basic principal of the separation of the knowledge from its treatment is of prime importance in the building of every expert system. The building and maintenance of an expert system is greatly facilitated by trying to adhere to this principal as closely as possible. Creation of knowledgebase is known as the most difficult phase in the life cycle of an expert system. Moreover in today’s dynamic world the solution loses its relevance if it is not based on the recent knowledge. So, there is need of the regular updating of knowledgebase. If knowledgebase can be built dynamically, then this problem can be solved more easily than the manual updating.

Research in the field of expert systems is started in 1960’s in chemistry and medical domains. Success achieved in these systems results in popularization of the expert systems in general and their application in wide range of domains including business. Initially, expert systems were built from scratch and there is no abstraction in terms of knowledgebase and inference engine. As the research in the field matured researchers have developed generic inference engines that can be applied to any knowledge domains.

Feigenbaum, [1982] defined an expert system as “an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions.” Hayes-Roth, [1983] linked the problem solving capability of expert systems with efficiency and performance as “expert system achieves high performance by using knowledge to make the best use of its time.” Jackson, [1988] defined an expert system as “a computer program that represents and reasons with knowledge of some specialist subject with a view to solving problems or giving advice”.
Expert systems incorporate concepts derived from experts in a field and use their knowledge to provide problem analysis to users of the software. Knowledge is a major factor in the performance of an expert system and is in two forms. The first is common facts consisting of widely shared knowledge that is accepted by the professional and other accepted sources of data. The second type of knowledge is called heuristic, which is the knowledge of good judgment and common good practice or "rule of thumb" in a field. Both facts and heuristic knowledge are required for development of an expert system. This section reviews some of the significant contributions by earliest expert systems, on which the recent expert systems stand.

The DENDRAL (1965-83) [Buchanan and Feigenbaum, 1978; Lindsay et al. 1980] was one of the earliest expert systems and demonstrated the power of rule-based reasoning. Using a plan-generate-test search paradigm and data from mass spectrometry and other sources, DENDRAL proposes plausible candidate structures for new or unknown chemical compounds. META-DENDRAL (1970-76) is an inductive program that automatically formulates new rules for DENDRAL to use in explaining data about unknown chemical compounds. Although META-DENDRAL is no longer an active program, its contributions to ideas about learning and discovery are being applied to new domains. Among these ideas are that induction can be automated as heuristic search; that, for efficiency, search can be broken into two steps—approximate and refined; that learning must be able to cope with noisy and incomplete data; and that learning multiple concepts at the same time is sometimes inescapable.

MYCIN (1972-80) [Shortliffe, 1974] is an interactive program that diagnoses certain infectious diseases, prescribes anti-microbial therapy, and can explain its reasoning in detail. MYCIN extended the notion that the knowledgebase should be separate from the inference engine, and its rule-based inference engine was built on a backward-chaining, or goal-directed, control strategy. Since, it was designed as a consultant for physicians, MYCIN was given the ability to explain both its line of reasoning and its knowledge. And because medical diagnosis often involves a degree of uncertainty, MYCIN's rules incorporated certainty factors to indicate the importance (i.e., likelihood and risk) of a conclusion. MYCIN has substantially influenced other AI research and led to work in EMYCIN [van Melle et al., 1981], PUFF [Aikins et al., 1983], CENTAUR [Aikins, 1984], VM [Fagan, 1980], and SACON [Bennett, 1978].

EMYCIN or "Essential MYCIN" (1974-79) was developed with the core inference engine of MYCIN, together with a knowledge engineering interface. It is a domain-independent framework that can be used to build rule-based expert systems for consultation problems such as those encountered in diagnosis or troubleshooting.

The CENTAUR (1977-80) system was designed to experiment with an expert system that combines both rule and frame-based approaches to represent and use knowledge about medicine and medical diagnostic strategies. CENTAUR demonstrated the effectiveness of this representation and control methodology.
The Ventilator Manager (VM) (1977-81) program interprets online quantitative data in the intensive care unit (ICU) and advises physicians on the management of post-surgical patients needing a mechanical ventilator to help them breathe. While based on the MYCIN architecture, VM was redesigned to allow for the description of events that change over time. Thus, it can monitor the progress of a patient, interpret data in the context of the patient's present and past condition, and suggest adjustments to therapy.

The AM (1974-80) [Lenat, 1976, Davis and Lenat 1982] program explored machine learning by discovery in the domain of elementary mathematics. Using a framework of 243 heuristic rules, AM successfully proposed plausible new mathematical concepts, gathered data about them, noticed regularities, and, completing this cycle, found ways of shortening the statement of those hypotheses by making new definitions. However, AM was not able to generate new heuristics. This failing was found to be inherent in the design of AM; related work on discovering new heuristics was done as part of EURISKO [Lenat, 1983]. In each domain, EURISKO has three levels of task to perform: working at the domain level to solve problems; inventing new domain concepts; and synthesizing new heuristics that are specific and powerful enough to aid in handling tasks in the domain.

INTERLISP [Teitelman & Masinter, 1981] is a programming language and has all the standard features of LISP [Siklossy, 1976] plus an elaborate support environment that includes sophisticated debugging facilities. Some of the above cited expert systems have their code written in INTERLIST.

GLISP [Novak, 1982] is a programming language that allows programs to be written in terms of objects and their properties and behavior. The GLisp compiler converts such programs into efficient Lisp code. The compiler has been released to outside users, along with the window-based data inspector, which displays data according to their GLisp description.

The research in the field of expert systems during mid 1980’s to mid 1990’s takes a leap forward in developing techniques like Fuzzy Systems, Artificial Neural Networks, Genetical Optimizers etc. These techniques introduced additional features in expert system’s reasoning mechanism and self-learning capability.

Wheat Management Expert System (NEPER): NEPER is an integrated bilingual (English and Arabic) system that addresses all the aspects of wheat management in Egypt. The main goal of the NEPER wheat expert system is to provide its users with the recommendation and advice concerning wheat production management in Egypt. Hence, the system is designed to cover the majority of the agriculture operations involved in the cultivation of this crop. There are two main components of NEPER Wheat a strategic part, and a tactical part.
1. Strategic Component
   Strategic component consists of six modules. Each module can run independently and produce multiple plans. A database has been integrated with the expert system to store the data of a specific field, fertilizers, and equipment.
The variety Selection module: This module identifies the best variety given a use. Variety is displayed with its growth period and the expected yield. Current sixteen varieties are included.

The planting module: The goal of this module is to determine the planting parameters for the field including: planting date, planting method, sowing method, and seeding rate.

The Land preparation Module: The objective of this module is to generate a schedule of operations for land preparation including tillage, leveling, weed removable, stone picking, and drainage system improvement. For each operation, the date, the tool and the method are given.

The Irrigation module: the amount of water and its application date. The goal of this module is to generate an irrigation schedule containing.

The Fertilizer Module: The goal of this module is to generate fertilization schedule containing nitrogen, phosphorus, and potassium needed by the growing plant throughout the cropping season.

The Harvest Module: The goal of the harvest system is to determine the suitable time and method of harvesting operation.

2. The Tactical component

The tactical component consists of two modules:

The Weeds Identification and Control Module: The goal of this module is to help its user to identify the weeds that present on his farm through images and sketches. The module also provide advice and control measures for identified weed.

Diagnosis and Treatment Module: The goal of this module is to identify and treat diseases, insects, and malnutrition. The system currently covers nineteen different diseases, eleven insects, and the deficiency and excess of eight different elements.

Weed Control Advisor for Rice (1995-1998): The Weed Control Advisor (WCA) is an expert system that is designed to help the user with weed control decisions throughout the year in the rice production areas in Texas. The weed complex considered in the WCA includes 23 of the most important weeds in Texas rice production. The WCA accounts for weed species, weed size, rice growth stage, rice size, water management practices, herbicide history and environmental conditions. The WCA provides a list of recommended herbicides, weed control strategies, and explanations of why herbicides are recommended or not recommended. WCA has been licensed by AgroEcoSystems Research Group, Texas A&M University.

RiceIPM – An interactive information and identification system for Integrated Pest Management of riceRiceIPM is an interactive training and resource package for researchers, advisers and students. Focusing on tropical rice, this system provides a comprehensive source of information and training material for improving the management of rice-pests, including insect pests, diseases and weeds. A major feature of this interactive knowledge management tool is that users can navigate through the content in any way they want to meet their own, specific information and learning needs. The combination of video, images, hypertext links and interactive keys provides a unique way
of accessing the wealth of material. A diagnostic key assists users to short list the likely causes of observed rice disorders. A series of interactive, multimedia lucid keys provides help to users in identifying insects found in rice. A customized search engine provides a rapid means of directing the user to specific topics to be found in RiceIPM. The system contains section on Pest ecology; crop checking; fact sheets on major insect pests, rats, diseases, weeds, nutrient deficiency and toxicity; crop growth and pest damage; pest management options and decision-making and economics. A separate section provides material for researchers and advisers on various aspects of implementing IPM, including Farmer Field Schools, Multimedia campaigns, and stakeholder workshops. Other components include frequently asked questions, a multiple-choice quiz, a ‘slide-show’ tutorial on biological control, and an information sources section, including references, rice statistics and a glossary. The software was developed by University of Queensland Australia.

Vidur- A Case Based Reasoning (CBR) based Advisory System for Farmers: CBR relies on solving problems based on past solutions, just as humans use experience to solve problems. Experiences are stored as cases in a case base. Each case has a problem description part and a solution part. The problem to be solved is a case, without a solution. Solving a problem reduces to searching similar cases from the case base and uses the solution of the similar retrieved case(s). The simplicity in storing experiences allows anyone associated with the domain share their experiences. The solution strategy being general and has been applied successfully in many domains ranging from medical to agriculture. Vidur is an Advisory System for farmers developed using CBR. Vidur is developed to be used by farmers in Manipur state in some aspects of a farming process. At present, the system can be used for Paddy Variety Selection and Weed Control. The system was developed with the knowledge fed by agricultural experts from CAU, Imphal. With this advisory system, farmers now have a virtual expert available as and when needed.

A number of ontology based expert systems are reported in literature. Ontology-based expert system for database design [Storey et al. 1998] is based on the ontology that can be used as a surrogate for the meaning of words in a database design system to simulate the contributions that a designer would make based on his or her general knowledge. The ontology classifies a term into one or more categories such as person, abstract good or tradable document. The expert system is comprised of a semantic network, a knowledgebase containing information on the meaning of terms that have been classified, an expert system knowledge-acquisition component, and a distance measure for assessing the distance between the meanings of terms.

Ontology-based Knowledge Management System for the Metal Industry [Li et al. 2003] employs information ontology and domain ontology to manage complexity and diversity of metal industry knowledge. The system used the KAON [Volz et al. 2003] API environment and is built upon the Java J2EE distributed component environment. It provides the capability of semantic match, assist engineers in designing a blueprint and determining what formula can be applied to, who has the better solution, and what materials can be used, etc.
Liao, [2005] presents a decade review (1994 to 2003) of research in expert systems. He divided the advances in research in many categories according to the major type of techniques or paradigms. These categories are Rule-based systems, Knowledge-based systems, Neural networks, Fuzzy expert systems, Object-oriented methodology, Case-based reasoning, Modeling, Database methodology, System architecture, Intelligent agents, and Ontology.

Lee & Wanga, [2007] developed an ontology-based intelligent healthcare agent for the respiratory waveform recognition to assist the medical staff in judging the meaning of the graph reading from ventilators. The intelligent healthcare agent contains three modules, including the respiratory waveform ontology, ontology construction mechanism, and fuzzy recognition agent, to classify the respiratory waveform. The respiratory waveform ontology represents the respiratory domain knowledge, which is utilized to classify and recognize the respiratory waveform by the intelligent healthcare agent. The ontology construction mechanism infers the fuzzy numbers of each respiratory waveform from the patient or respiratory waveform repository. Next, the fuzzy recognition agent classifies and recognizes the respiratory waveform into different types of respiratory waveforms. Finally, after the confirmation of medical experts, the classified and recognized results are stored in the classified waveform repository. The experimental results show that their approach can classify and recognize the respiratory waveform effectively.

**Rule-based Systems**
A typical rule-based system has four basic components:
1. A list of rules or rule base, which is a specific type of knowledge base.
2. An inference engine or semantic reasoner, which infers information or takes action based on the interaction of input and the rule base. The interpreter executes a production system program by performing the following match-resolve-act cycle:
   a. Match: In this first phase, the left-hand sides of all productions are matched against the contents of working memory. As a result a conflict set is obtained, which consists of instantiations of all satisfied productions. An instantiation of a production is an ordered list of working memory elements that satisfies the left-hand side of the production.
   b. Conflict-Resolution: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.
   c. Act: In this third phase, the actions of the production selected in the conflict-resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, execution returns to the first phase.
3. Temporary working memory.
4. A user interface or other connection to the outside world through which input and output signals are received and sent.
Case-based reasoning (CBR) Systems

Case-based reasoning (CBR) is the process of solving new problems based on the solutions of similar past problems. An auto mechanic who fixes an engine by recalling another car that exhibited similar symptoms is using case-based reasoning. A lawyer who advocates a particular outcome in a trial based on legal precedents or a judge who creates case law is using case-based reasoning. So, too, an engineer copying working elements of nature (practicing biomimicry), is treating nature as a database of solutions to problems. Case-based reasoning is a prominent kind of analogy making.

It has been argued that case-based reasoning is not only a powerful method for computer reasoning, but also a pervasive behavior in everyday human problem solving; or, more radically, that all reasoning is based on past cases personally experienced. This view is related to prototype theory, which is most deeply explored in cognitive science.

Reasoning is often modeled as a process that draws conclusions by chaining together generalized rules, starting from scratch. Case-based reasoning (CBR) takes a very different view. In CBR, the primary knowledge source is not generalized rules but a memory of stored cases recording specific prior episodes. In CBR, new solutions are generated not by chaining, but by retrieving the most relevant cases from memory and adapting them to fit new situations. Thus in CBR, reasoning is based on remembering. As the passages starting this section illustrate, remindings facilitate human reasoning in many contexts and for many tasks, ranging from children's simple reasoning to expert decision-making. Much of the original inspiration for the CBR approach came from the role of reminding in human reasoning.

The CBR approach is based on two tenets about the nature of the world. The first tenet is that the world is regular: similar problems have similar solutions. Consequently, solutions for similar prior problems are a useful starting point for new problem-solving. The second tenet is that the types of problems an agent encounters tend to recur. Consequently, future problems are likely to be similar to current problems. When the two tenets hold, it is worthwhile to remember and reuse current reasoning: case-based reasoning is an effective reasoning strategy.

CBR can also be beneficial, however, when a reasoner must solve problems that are quite different from prior experiences. As a case-based reasoner applies cases to increasingly novel problems, the CBR process changes from simple reuse to more creative problem-solving. Regardless of whether a case-based reasoner solves a routine or novel problem, and of whether the problem-solving outcome is success or failure, the case-based reasoner learns from its experience. Complementary with the principle of reasoning by remembering is the principle that reasoning is remembered—that reasoning and learning are intimately connected. The knowledge of a case-based reasoner is constantly changing as new experiences give rise to new cases which are stored for future use. A case-based reasoner learns from experience to exploit prior successes and avoid prior failures. Humans are robust problem-solvers; they routinely solve hard problems despite limited and uncertain knowledge, and their performance improves with experience. All of these qualities are desirable for real-world AI systems. Consequently, it is natural to ask how
CBR can advance AI technology. Discussions of this question have identified five main problems that can be ameliorated by case-based reasoning:

1. **Knowledge acquisition:** A classic problem in traditional knowledge-based systems is how to provide the rules on which the systems depend. The rule acquisition process can be laborious and unreliable: it may be difficult to elicit rules, and there is no assurance that those rules will actually be sufficient to characterize expert performance. In some domains, rules may be difficult to formalize or the number of rules required may be unmanageably large. Because case-based reasoners reason from complete specific episodes, CBR makes it unnecessary to decompose experiences and generalize their parts into rules. Some task domains are especially natural for CBR, with cases that are suitable for CBR already collected as part of standard problem-solving procedures. In those domains, the cost of knowledge acquisition for CBR is very low.

2. **Knowledge maintenance:** Defining an initial knowledge base is generally only the first step towards a successful AI application. Initial understanding of the problem is often imperfect, requiring system knowledge to be refined. Likewise, changes in task requirements and circumstances may render existing knowledge obsolete. Although refinement of case representations and indexing schemes may be required as a task becomes better understood, CBR offers a significant benefit for knowledge maintenance: a user may be able to add missing cases to the case library without expert intervention. Also, because CBR systems do incremental learning, they can be deployed with only a limited set of "seed cases," to be augmented with new cases if (and only if) the initial case library turns out to be insufficient in practice. A CBR system needs only to handle the types of problems that actually occur in practice, while generative systems must account for all problems that are possible in principle.

3. **Increasing problem-solving efficiency:** People achieve satisfactory problem-solving performance despite the fact that commonplace problems in everyday reasoning, such as explanation and planning, are NP-hard. Reuse of prior solutions helps increase problem-solving efficiency by building on prior reasoning rather than repeating prior effort. In addition, because CBR saves failed solutions as well as successes, it can warn of potential problems to avoid.

4. **Increasing quality of solutions:** When the principles of a domain are not well understood, rules will be imperfect. In that situation, the solutions suggested by cases may be more accurate than those suggested by chains of rules, because cases reflect what really happens (or fails to happen) in a given set of circumstances. In medical reasoning, for example, anecdotes about specific cases go beyond codified knowledge, serving as "the as-yet-unorganized evidence at the forefront of clinical medicine".

User acceptance: A key problem in deploying successful AI systems is user acceptance: no system is useful unless its users accept its results. To trust the system's conclusions, a user may need to be convinced that they are derived in a reasonable way. This is a problem for other approaches: neural network systems cannot provide explanations of their decisions, and rule-based systems must explain their decisions by reference to their rules, which the user may not fully understand or accept. On the other hand, the results of
CBR systems are based on actual prior cases that can be presented to the user to provide compelling support for the system's conclusions.

**Online Expert System Architecture**
There are four essential component of a full-fledged expert system:
1. The knowledge base.
2. The inference engine.
3. The knowledge acquisition module.
4. The Explanatory interface.

All four modules shown in figure are critical, and while a knowledge-base system may lack one or two of them, a truly ‘expert system’ should not.

**The Knowledge Acquisition Module**
Knowledge in terms of a computer program is information that makes computer to behave intelligently. It may be in the form of facts, beliefs, and heuristic rules. An integrated collection of facts and relationships that when exercised, produces competent performance. The quantity and quality of knowledge possessed by a person or a computer can be judged by the variety of situations in which the person or program can obtain successful results.

Knowledge Acquisition is the process of extracting domain knowledge from domain experts. The process of incorporating domain knowledge into an expert system by extracting it from domain experts and encoding the information into an internal representation, such as rules. An automated process by which a program accepts knowledge from domain experts and incorporates it into an existing expert system learning. The process of extracting, structuring, and organizing knowledge from some source, usually human experts, so that it can be used in a program. The person undertaking the knowledge acquisition must convert the acquired knowledge into a form that a computer program can use. Essentially, the knowledge acquisition is the technique by which a knowledge engineer obtains information from experts, textbooks, and other authentic sources for ultimate translation into a machine language and knowledge base.

![Knowledge Acquisition Module](image)

**The Knowledgebase**
Although the term “knowledge representation” can be used in the general sense to refer to all types of representation - of problems, of solutions, and of knowledge in a rule base, it is this last use that most closely captures the meaning that we associate with the word
“knowledge”. A knowledge base for the programming field captures two fundamental types of knowledge - the connection from source code fragment to an abstract description of what the source code fragment “causes”; it also captures causal connections from one state (or possibly multiple states) to another (result) state. This is the classic “antecedent-consequent” form of a rule. A rule base for the expert system needs to provide a set of tables allowing clear distinction between rule antecedent and rule consequent; therefore a rule base structure involving a separate table for antecedent source code fragments is preferred; this table will exist to provide antecedents to the abstract state descriptions contained in the main consequent table or tables. Finally, intermediate tables may exist to show one-to-many relationships from result-states to prior-states, as well as one-to-many relationships from antecedents to consequents where an antecedent source code fragment or an antecedent state has application to multiple effect situations.

The Inference Engine
The heart of an expert system is the reasoning engine or “inference engine”. The reasoning engine processes the input statements of the problem statement document; this input is usually presented to the reasoning engine in an in-memory tree structure, having first been processed by a lexical analyzer and parser. The chief task of the reasoning engine is to take a problem statement, for instance a goal statement, and to match the set of attributes of the goal against a set of attributes in the rule base in a rule consequent table. The reasoning engine uses the results of this match to start retrieving the corresponding rule antecedents: there may be multiple “parallel” antecedents for a given goal, and/or there may be multiple antecedents that will form a “chain” of state description records as they are retrieved. The reasoning engine works together with the code generator to the effect that all necessary source code fragments are retrieved and combined to form a larger target program fragment, which is written to an output program file. An important part of this process is that of performing substitutions - the generator will in most cases be able to take values directly from the input specification statements and use them in a function that substitutes actual values for attribute value placeholders.

More sophisticated logic may exist in the engine, especially logic that is involved in determining “already happened” states. (The term “happened” is used to apply to states, even though it might more naturally be used to describe “actions”). An example of an “already happened” state is the state that results from the registration of a particular window class with its accompanying procedure. It is important to record such states since they may participate in multiple goals. For instance, one registered window class may be used to create multiple windows of that class. If the reasoning engine has logic to record “already happened” states and to determine states that have been recorded in this way, it greatly enhances its power and flexibility. Many situations occur in the programming domain that can benefit from the use of such an algorithm.

Explanatory Interface
Explanatory Interface allows the user to track the route followed to reach to the goal state. It also builds the confidence in the user that the results obtained are based on the reasoning and provides insight of the inference procedure. It also allows user to go back to a particular level and take another route if one is not satisfied with the given result.
Fig. 8. General Design of the Expert System
Implementation Details

1. Explanatory Interface
Explanatory Interface will be implemented in HTML (Hypertext Markup Language), JavaScript and CSS (Cascaded Style Sheets). It contains forms for presenting the data to the user in the form of tables and accepting information and allowing control functionality to the user.

HTML & CSS
To publish information for global distribution, one needs a universally understood language, a kind of publishing mother tongue that all computers may potentially understand. The publishing language used by the World Wide Web is HTML. User friendly explanatory interface will be built in HTML. CSS will be used for maintaining the uniform format and look of the different pages of the system.

JavaScript
JavaScript will be used for client-side scripting. The script will be included in or referenced by an HTML document, for the code to be interpreted by the browser. The script will be used for checking client side validations in forms and for implementing other user interface logic.
2. Application Logic Layer

In practice, the complicated part of creating a Web based system is getting the information into the database. Once the information has been stored, it is relatively straightforward to search and manipulate using standard database techniques. In the present study Application Logic Layer will be implemented in Java Server Pages. Its purpose is:
- Providing services to the Explanatory Interface for the user
- Hiding all the interactions with the inference engine & knowledge base
- Implementing all the data acquisition & knowledge management rules.

3. Inference Engine Layer

The inference engine layer will be implemented using the Java Expert System Shell built by the Sandia National Laboratories, Livermore, CA, USA. IASRI has acquired R&D license for the JESS under this project. Jess's rule engine uses an improved form of a well-known algorithm called Rete (Latin for "net") to match rules against the knowledge base. Jess is actually faster than some popular expert system shells written in C, especially on large problems, where performance is dominated by algorithm quality. Rete is an algorithm that explicitly trades space for speed, so Jess' memory usage is not inconsiderable. Jess does contain some commands, which allows one to sacrifice some performance to decrease memory usage. Jess can be used in two overlapping ways. First, it can be a rule engine - a special kind of program that very efficiently applies rules to data. A rule-based program can have hundreds or even thousands of rules, and Jess will continually apply them to data in the form of a knowledge base. Often the rules will represent the heuristic knowledge of a human expert in some domain, and the knowledge base will represent the state of an evolving situation (an interview, an emergency). In this case, they are said to constitute an expert system. Expert systems are widely used in many domains. But the Jess language is also a general-purpose programming language, and furthermore, it can directly access all Java classes and libraries. For this reason, Jess is also frequently used as a dynamic scripting or rapid application development environment. While Java code generally must be compiled before it can be run, a line of Jess code is executed immediately upon being typed. This allows you to experiment with Java APIs interactively, and build up large programs incrementally. It is also very easy to extend the Jess language with new commands written in Java or in Jess itself, and so the Jess language can be customized for specific applications. Jess is therefore useful in a wide range of situations.

4. Database Layer

The knowledge base will be stored in form of a relational database. The database layer will provide the necessary rules to the inference engine and knowledge information to be presented by the explanatory interface through application logic layer. MS SQL server will be used for implementing this layer.
Some Guidelines For Deciding Whether To Use A Rules Engine

1. Is your algorithm primarily compute-intensive or is there significant decision-making capability involved?
   - If your basic algorithm is compute-intensive or a table-lookup, without much conditional branching or decision-making involved, then don't use a rules engine.
   - If however, the algorithm involves significant conditional branching or decision-making, then consider using a rules engine.

2. Once you've determined that your algorithm involves significant decision-making capability, you ought to be able to write some rules specifying the decisions that need to be made. Are the decisions that need to be made relatively simple, or potentially complex?
   - If you find that you can't write the decision rules, for whatever reason, then stop here until you can, or use some other tool instead that will help you discover the rules you need. Put another way, if you can't state some rules, don't use a rules engine.
   - If you have 2, or fewer, conditions in your rules (or, for example, a block with 2 nested if-statements or less), don't use a rules engine-it's probably overkill.
   - If you have 3 or more conditions in your rules (or, for example, a block with 3 or more nested if-statements in pseudo-code), then consider using a rules engine.

3. Once you've determined that the decisions are complex enough, is your algorithm basically static, or are the rules likely to change reasonably often over time?
   - If the rules/logic are well-defined and static, then don't use a rules engine-you probably don't need the overhead or flexibility.
   - If the rules are likely to change over time due to the nature of the application, then consider using a rules engine-the flexibility is worth the overhead.

4. Once you've determined that rules may need to be flexible, are you going to maintain the code or finished product over time, or is this effort a one-shot effort?
   - If the code is not going to be maintained over time, then don't use a rules engine-you probably won't gain any significant advantage from it.
   - If the code is going to be maintained over time, consider using a rules engine-the ROI will be worth it (see question #6).

5. Rules engines continue to get faster and faster, but some applications simply require every bit of speed and performance optimization you can reasonably give. Does your customer require a custom solution or do you need to hard-wire the algorithm for absolute high-end performance, or can the solution accommodate a rules engine?
   - If you need to optimize for speed and memory, or your customer requires a custom solution, then don't use a rules engine.
   - If the performance requirements will accommodate a rules engine solution, then consider using one.
6. If you answered all the other questions appropriately, can the project/product line afford the overall cost of using a rules engine over the project/product lifecycle?
   o There are a number of costs typically associated with using rules engine tools:
     - Licensing fees for development and deployment of the engine
     - Training developers and (if necessary) end-users (time and money)
     - ROI (return on investment)-financial analysts have shown that you don't begin to break even monetarily on the typical investment in rules engine technology until at least 1 year after deployment to your customer. However, at that point, the flexibility and ease-of-maintenance begin to show significant returns.
   o If the project schedule or product lifecycle can't accommodate the cost of a rules engine, in terms of time and money, then don't use one.
   o If you can't afford to wait at least a year to break even and begin to see a significant ROI, don't use a rules engine.
   o If you can't afford to train your developers and end-users, and you can't afford to hire a consultant, don't use a rules engine.
   o If, however, your project/product lifecycle costs can accommodate the use of a rules engine, it would be well worth the investment, so use one.

A rules engine tool can be very helpful during software development, regardless of other considerations:

- For simulation and prototyping.
- In cases where you find you may not really know or understand the rules you are trying to encode in your algorithm, a rules engine can provide a flexible way to encode and modify the rules over time as they are discovered.
- A rules engine architecture also provides a convenient structure for separating "business logic" from the rest of the system, aiding in the effort to clearly "separate concerns".

**Jess & Rete Algorithm**

Jess is a rule engine. In the simplest terms, this means that Jess's purpose it to continuously apply a set of if-then statements (rules) to a set of data (the working memory). You define the rules that make up your own particular rule-based system. Jess rules look something like this:

```
Jess> (defrule library-rule-1
   (book (name ?X) (status late) (borrower ?Y))
   (borrower (name ?Y) (address ?Z))
=>
   (send-late-notice ?X ?Y ?Z))
```

This rule might be translated into pseudo-English as follows:
Library rule #1:
If
  a late book exists, with name X, borrowed by someone named Y
and
  that borrower's address is known to be Z
then
  send a late notice to Y at Z about the book X.

The book and borrower entities would be found on the working memory. The working memory is therefore a kind of database of bits of factual knowledge about the world. The attributes (called slots) that things like books and borrowers are allowed to have are defined in statements called deftemplates. Actions like send-late-notice can be defined in user-written functions in the Jess language (deffunctions) or in Java (Userfunctions).

Each Jess rule engine holds a collection of knowledge nuggets called facts. This collection is known as the working memory. Working memory is important because rules can only react to additions, deletions, and changes to working memory. You can't write a Jess rule that will react to anything else.

Some facts are pure facts defined and created entirely by Jess. Other facts are shadow facts connected to Java objects you provide. Shadow facts act as "bridges" that let Jess reason about things that happen outside of working memory.

Every fact has a template. The template has a name and a set of slots, and each fact gets these things from its template. This is the same structure that JavaBeans -- plain old Java objects, or POJOs -- have, and it's also similar to how relational databases are set up. The template is like the class of a Java object, or like a relational database table. The slots are like the properties of the JavaBean, or the columns of a table. A fact is therefore like a single JavaBean, or like a row in a database table. You can think of it either way. In Jess, there are three kinds of facts: unordered facts, shadow facts and ordered facts.

The typical rule-based program has a fixed set of rules while the working memory changes continuously. However, it is an empirical fact that, in most rule-based programs, much of the working memory is also fairly fixed from one rule operation to the next. Although new facts arrive and old ones are removed at all times, the percentage of facts that change per unit time is generally fairly small. For this reason, the obvious implementation for the rule engine is very inefficient. This obvious implementation would be to keep a list of the rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the working memory and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration. However, since the working memory is stable, most of the tests will be repeated. You might call this the rules finding facts approach and its computational complexity is of the order of $O(RF^P)$, where $R$ is the number of rules, $P$ is the average number of patterns per rule LHS, and $F$ is the number of facts on the working memory. This escalates dramatically as the number of patterns per rule increases.

Jess instead uses a very efficient method known as the Rete (Latin for net) algorithm. The classic paper on the Rete algorithm ("Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19 (1982), 17-37) became the basis for a whole generation of fast rule engines: OPS5, its descendant ART, CLIPS, and of course Jess. In the Rete algorithm, the inefficiency described above
is alleviated (conceptually) by remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. Additionally, as will be described below, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops to something more like O(RFP), or linear in the size of working memory. The Rete algorithm is implemented by building a network of nodes, each of which represents one or more tests found on a rule LHS. Facts that are being added to or removed from the working memory are processed by this network of nodes. At the bottom of the network are nodes representing individual rules. When a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule and this set becomes an activation. The associated rule may have its RHS executed (fired) if the activation is not invalidated first by the removal of one or more facts from its activation set.

Within the network itself there are broadly two kinds of nodes: one-input and two-input nodes. One-input nodes perform tests on individual facts, while two-input nodes perform tests across facts and perform the grouping function. Subtypes of these two classes of node are also used and there are also auxiliary types such as the terminal nodes mentioned above.

For example: The following rules:

```
(defrule example-2
  (x)
  (y)
  (z)
  =>)
```

```
(defrule example-3
  (x)
  (y)
  =>)
```

might be compiled into the following network:

```
Fig. 10.
```
The nodes marked x?, etc., test if a fact contains the given data, while the nodes marked + remember all facts and fire whenever they've received data from both their left and right inputs. To run the network, Jess presents new facts to each node at the top of the network as they added to the working memory. Each node takes input from the top and sends its output downwards. A single input node generally receives a fact from above, applies a test to it, and, if the test passes, sends the fact downward to the next node. If the test fails, the one-input nodes simply do nothing. The two-input nodes have to integrate facts from their left and right inputs, and in support of this, their behavior must be more complex. First, note that any facts that reach the top of a two-input node could potentially contribute to an activation: they pass all tests that can be applied to single facts. The two input nodes therefore must remember all facts that are presented to them, and attempt to group facts arriving on their left inputs with facts arriving on their right inputs to make up complete activation sets. A two-input node therefore has a left memory and a right memory. It is here in these memories that the inefficiency described above is avoided. A convenient distinction is to divide the network into two logical components: the single-input nodes comprise the pattern network, while the two-input nodes make up the join network.

There are two simple optimizations that can make Rete even better. The first is to share nodes in the pattern network. In the network above, there are five nodes across the top, although only three are distinct. We can modify the network to share these nodes across the two rules (the arrows coming out of the top of the x? and y? nodes are outputs):

![Diagram of Rete network with shared nodes]

But that's not all the redundancy in the original network. Now we see that there is one join node that is performing exactly the same function (integrating x,y pairs) in both rules, and we can share that also:

![Diagram of Rete network with shared join node]
The pattern and join networks are collectively only half the size they were originally. This kind of sharing comes up very frequently in real systems and is a significant performance booster!

One can see the amount of sharing in a Jess network by using the watch compilations command. When a rule is compiled and this command has been previously executed, Jess prints a string of characters something like this, which is the actual output from compiling rule example-2, above:

example-2: +1+1+1+1+1+1+2+2+t

Each time +1 appears in this string, a new one-input node is created. +2 indicates a new two-input node. Now watch what happens when example-3 is compiled:

example-3: =1=1=1=1=2+t

Here, =1 is printed whenever a pre-existing one-input node is shared; =2 is printed when a two-input node is shared. +t represents the terminal nodes being created. (Note that the number of single-input nodes is larger than expected. Jess creates separate nodes that test for the head of each pattern and its length, rather than doing both of these tests in one node, as we implicitly do in our graphical example.) No new nodes are created for rule example-3. Jess shares existing nodes very efficiently in this case.

Jess's Rete implementation is very literal. Different types of network nodes are represented by various subclasses of the Java class jess.Node: Node1, Node2, NodeNot2, NodeJoin, and NodeTerm. The Node1 class is further specialized because it contains a command member which causes it to act differently depending on the tests or functions it
needs to perform. For example, there are specializations of Node1 which test the first field (called the head) of a fact, test the number of fields of a fact, test single slots within a fact, and compare two slots within a fact. There are further variations which participate in the handling of multifields and multislots. The Jess language code is parsed by the class jess.Jesp, while the actual network is assembled by code in the class jess.ReteCompiler. The execution of the network is handled by the class Rete. The jess.Main class itself is really just a small demonstration driver for the jess package, in which all of the interesting work is done.

The view command is a graphical viewer for the Rete network itself; I have used this as a debugging tool for Jess, but it may have educational value for others, and it may help you to design more efficient systems of rules in Jess. Issuing the view command after entering the rules example-2 and example-3 produces a very good facsimile of the drawing (although it correctly shows the larger number of one-input nodes). The various nodes are color-coded according to their roles in the network; Node1 nodes are red; Node2 nodes are green; NodeNot2 nodes are yellow; and Defrule nodes are blue. The orange node in the figure is a "right-to-left adapter" node; one of these is always used to connect the first pattern on a rule's LHS to the network. Passing the mouse over a node displays information about the node and the tests it contains; double-clicking on a node brings up a dialog box containing the same information (for join nodes, the memory contents are also displayed, while for Defrule nodes, a pretty-print representation of the rule is shown). See the description of the view function for important information before using it.

Fig. 13. Graphic Representation of Nodes in Jess

Jess's rule engine uses an improved form of a well-known algorithm called Rete (Latin for "net") to match rules against the working memory. Jess is actually faster than some
popular rule engines written in C, especially on large problems, where performance is
dominated by algorithm quality. Rete is an algorithm that explicitly trades space for
speed, so Jess' memory usage is not inconsiderable. Jess does contain some commands
which will allow you to sacrifice some performance to decrease memory usage.
Nevertheless, Jess' memory usage is not ridiculous, and moderate-sized programs will fit
easily into Java's default 16M heap.

The single biggest determinant of Jess performance is the number of partial matches
generated by your rules. You should always try to obey the following (sometimes
contradictory) guidelines while writing your rules:

- Put the most specific patterns near the top of each rule's LHS.
- Put the patterns that will match the fewest facts near the top of each rule's LHS.
- Put the most transient patterns (those that will match facts that are frequently
  retracted and asserted) near the bottom of a LHS.

Content Based Image Retrieval Expert System

Farmers need satisfactory, timely and easy advice from experts. In a vast country like
India, it is not feasible for experts to reach each and every farmer. Online expert systems
have the capability to accomplish this goal. In order to deliver good advice the expert
system should have sufficient knowledge about the domain. Acquiring sufficient
knowledge and representing it in a machine understandable format is time consuming and
difficult job. Also, representing each and every kind of knowledge is still an open
research problem. Since, a single picture is worth a thousand words, it is a good idea to
acquire knowledge in terms of images rather than only text. Image is an easy way of
communication without any boundary of language. Hence, there is a need for building a
Content Based Image Retrieval (CBIR) Expert System that could deliver the knowledge
by searching the image having the similar features that is searched by the user. In the
presented work, an attempt is made to diagnose diseases in crops using the CBIR
technique. The knowledgebase of the CBIR Expert System contains a large pool of
images already annotated by the experts as per the diseases in crops. In order to get an
advice from the system, farmer uploads an image of the infested plant from his field. The
system matches the uploaded image with the annotated images stored in the
knowledgebase and if the match is found, the system identifies the infested disease and
provides control measures needed to be adopted by the farmer. The system has a
capability to learn from its experience as it stores the uploaded image with the diagnosed
disease information in the auto annotated images section. The image is transferred from
Auto-annotated to Annotated section after verification and correction from the experts.

The architecture of the system is presented in Fig.14. The architecture of the
system enables its usage from a standard web browser by connecting it to the system
website. The user after the authorization check is presented with an interface wherein he
can upload the image.
The application logic layer is developed using Java Server Pages (JSP) and can be deployed on any Web server that supports JSP. The application logic layer handles interaction among the user and various tiers of the system. The system logic uses classes and interfaces of JAVA implementation of content based image retrieval method and are accessed in application logic layer. JDOM library is used for parsing and reading XML (MPEG-7 Files) and lucene-core Library is used for creating indexes and search them. Here, Apache Tomcat Server is used as web server to deploy the system. The System can also deploy as webservice.

The database layer provides the storage capability to the system and has tables for image name, image ID, crop name, Crop disease symptom, user name, district name, state name. The database also stores the user information for their authorization and access rights. The system has a repository for images and extracted features of MPEG-7 images. Logically system works in two phases viz. indexing and searching. Indexing phase is divided into two steps.
In first phase of the system, image is browsed and annotated and uploaded. During the upload, java LIRE API used to annotate the image and extract the image’s low level features like; metadata, color, histogram, texture and space information using CEDD, Color correlogram, FCTH algorithms and convert into XML format. The image and user field is stored in data base and indexing of the image folder.

In second phase of the system is for searching the similar image. LIRE API provides specialized methods to query the image repository having the similar features. It may be image annotation, color or texture based search.

The core task of this system is to compare and match two or more images for finding similarities. The presented Java LIRE API compares two or more images to find any similarity between them. This is performed by identifying the key values of given image’s descriptors, histogram color and texture information, and is matched with other stored images.

This system supports manual annotation of digital images and automatic extraction of existing metadata and low level features. Metadata is stored and exchanged using MPEG-7 XML.

To enhance visualisation thumbnails are created and referenced in the MPEG-7 file. Existing IPTC and EXIF metadata is extracted and transformed to appropriate MPEG-7 descriptors. Manual annotation can be done in multiple ways. This system supports:

- Crop Name
- Disease Symptoms
- Crop Disease
- Advice

To support content-based image retrieval (CBIR) in the Multimedia enabled expert system; three index searching techniques are used viz. Auto Color Correlograms, Edge Directivity Descriptor (CEDD) and Fuzzy Color and Texture Histogram (FCTH).

The system allows to search in MPEG-7 (image) document collections and support:

- Content-based image retrieval based on the aforementioned low-level features using query-by-example.

- Search in text based MPEG-7 descriptors based on keywords, quality assessments and XPath statements.

The system supports linear search in MPEG-7 file directories on the file system and indexing of selected descriptors in either a database or a Lucene index. Retrieval results are presented in list format. Snapshot of “Multimedia Enabled Expert System for Maize” is shown in Fig. 15.
The advantages of using CBIR expert system include language independent, crop independent, ease of acquiring knowledge, ability to learn and easy to use from other devices such as mobiles.