UNIT-2

REPRESENTATION OF KNOWLEDGE

CHAPTER-1

GAME PLAYING

2.1 Introduction

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

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

Game playing is a search problem defined by:

- Initial state of the game
- Operators defining legal moves
- Successor function
- Terminal test defining end of game states
- Goal test
- Path cost/utility/payoff function

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

Characteristics of game playing

1. There are always an “unpredictable” opponent:
    - The opponent introduces uncertainty
    - The opponent also wants to win

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

2. Time limits:

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

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

Types of games

There are basically two types of games

- Deterministic games
- Chance games

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

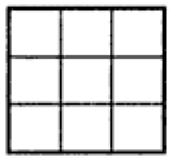
Minimize search procedure

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

In a game like tic-tac-toe

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

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



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

X	O	
	X	O
O		X

Another possibility is that no1 wins eg..

O	O	O
X	X	O
O	O	X

Or the third possibility is a draw case

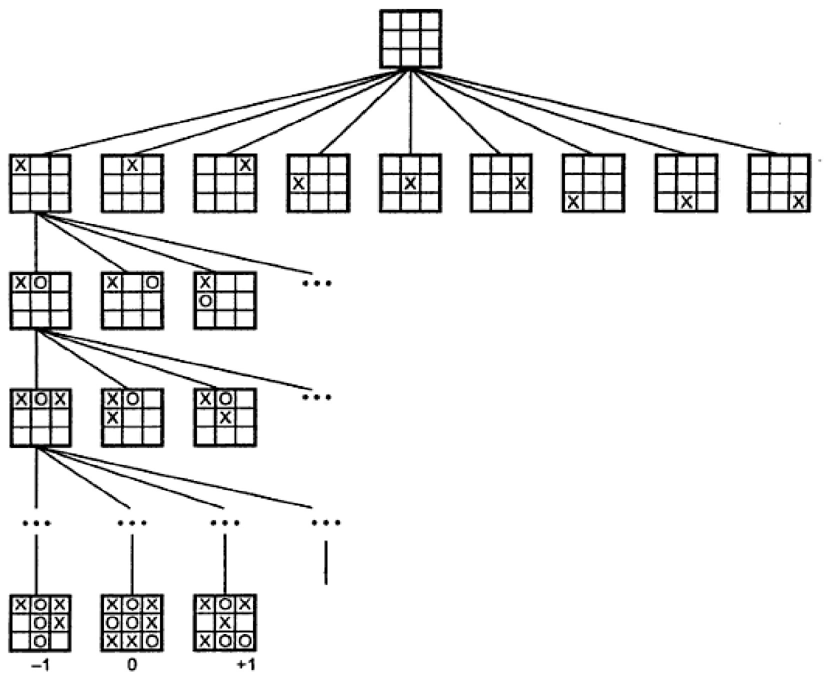
X	O	X
X	O	O
O	X	O

Search tree for tic-tac-toe

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

The trees representing games contain two types of nodes:

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



Search tree for tic-tac-toe

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

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

Minimax procedure

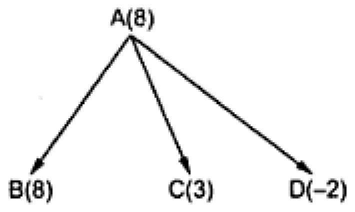
- Starting from the leaves of the tree (with final scores with respect to one player, MAX), and go backwards towards the root.
- At each step, one player (MAX) takes the action that leads to the highest score, while the other player(MIN) takes the action that leads to the lowest score.
- All the nodes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

The minimax procedure operates on a game tree and is recursive procedure where a player tries to minimize its opponent's advantage while at the same time maximize its own. The player hoping for positive number is called the maximizing player. His opponent is the minimizing player. If the player to move is the maximizing player, he is looking for a path leading to a large positive number and his opponent will try to force the play toward situation with strongly negative static evaluations. In game playing first construct the tree up till the depth-bound and then compute the evaluation function for the leaves. The next step is to propagate the values up to the starting.

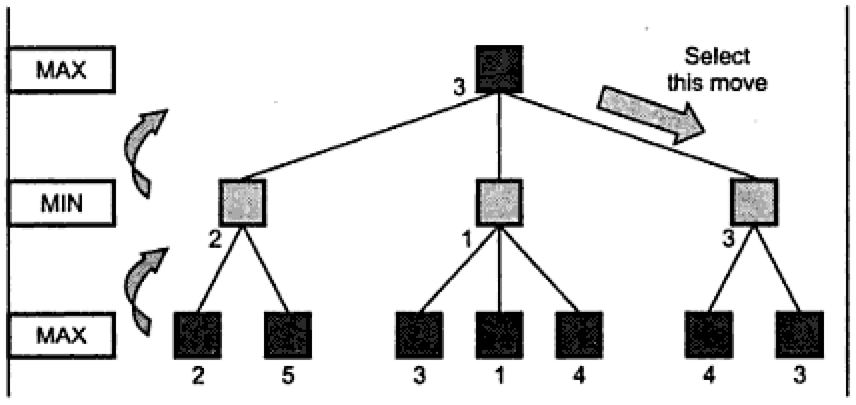
The procedure by which the scoring information passes up the game tree is called the MINIMAX procedures since the score at each node is either minimum or maximum of the scores at the nodes immediately below

One-ply search

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



Two-ply search



Static evaluation function

To play an entire game we need to combine search oriented and non-search oriented techniques. The idea way to use a search procedure to find a solution to the problem statement is to generate moves through the problem space until a goal state is reached. Unfortunately for games like chess even with a good plausible move generator, it is not possible to search until goal state is reached. In the amount of time available it is possible to generate the tree at the most 10 to 20 ply deep. Then in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous. This is done using the static evaluation function. The static evaluation function evaluates individual board positions by estimating how much likely they are eventually to lead to a win.

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

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

Apply MINIMAX to each of these successors. Return the maximum of the result.

The maximum algorithm uses the following procedures

1. MOVEGEN(POS)

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

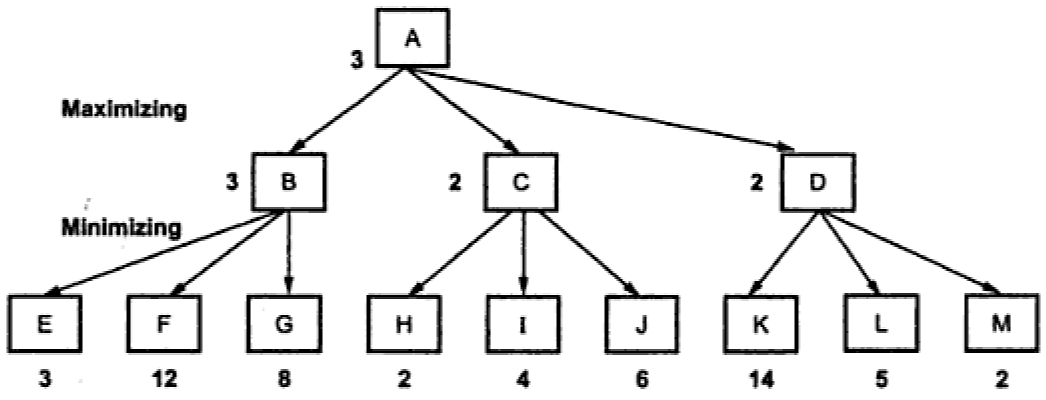
2. STSTIC (Pos, Depth)

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

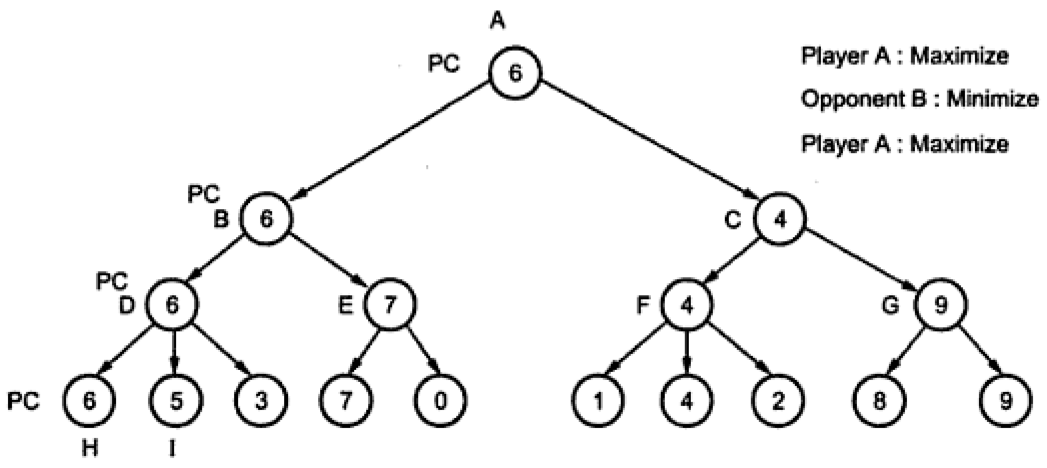
3. DEEP-ENOUGH

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

A MINIMAX example



Another example of minimax search procedure



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

Alpha-beta cutoffs

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

The general principle of alpha-beta pruning is

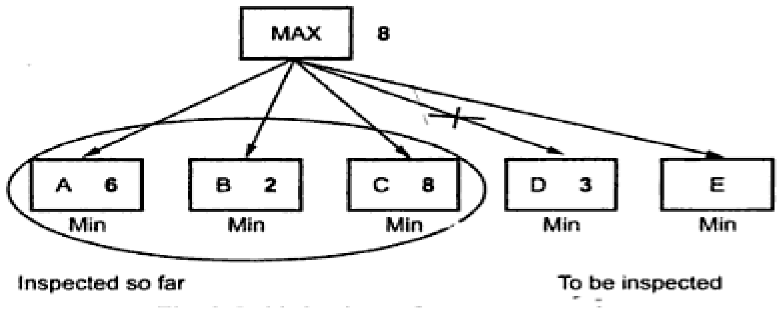
- Consider a node n somewhere in the tree, such that a player has a chance to move to this node.
- If player has a better chance m either at the parent node of n (or at any choice point further up) then n will never be reached in actual play.

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

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

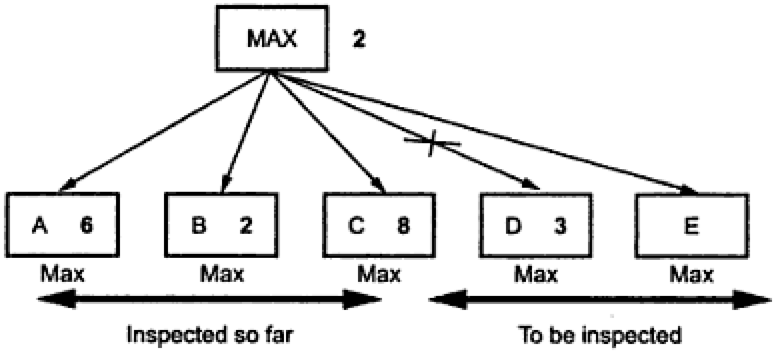
The maximize is constantly trying to push the alpha value up by finding better moves; the minimizer is trying to push the beta value down. If a node’s value is between alpha and beta, then the players might reach it. At the beginning, at the root of the tree, we don’t have any guarantees yet about what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

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



Alpha-beta for a max node

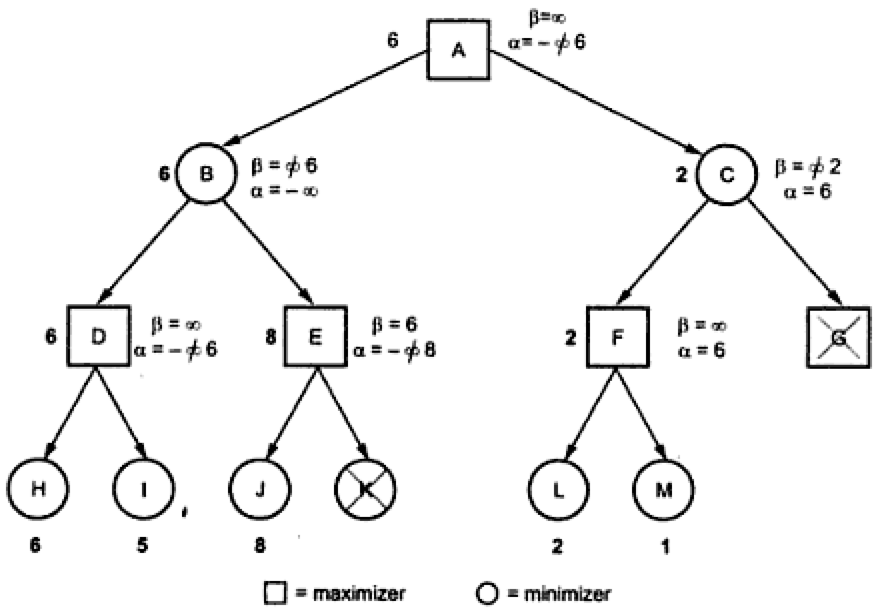
At this point the “tentative” value which is backed up so far of F is 8. MAX is not interested in any move which has a value of less than 8, since it is already known that 8 is the worst that MAX can do, so far. Thus the node D and all its descendent can be pruned or excluded from further exploration, since MIN will certainly go for a value of 3 rather than 8. Similarly for a MIN-node:



Alpha-beta for a min node

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

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



Tree with alpha-beta cut-offs

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

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

Algorithm-Alpha-beta

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

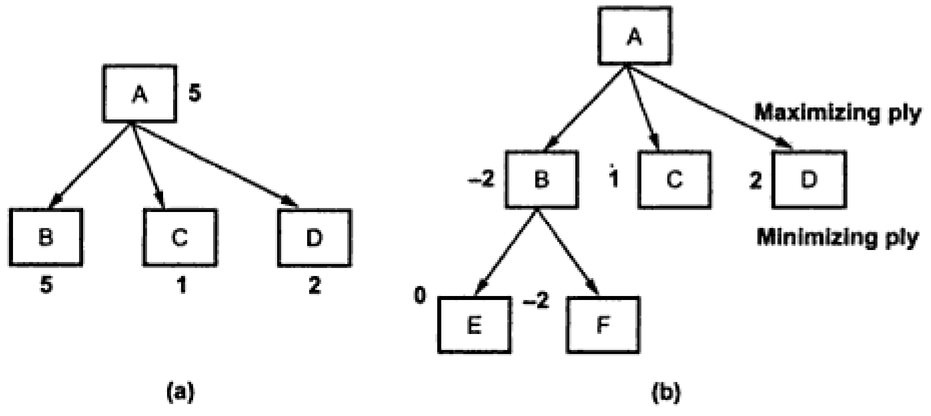
If the highlighted characters are removed, what is left is a min-max function. The function is passed the depth it should search, and –INIFINITY as alpha and +INIFINITY as beta.

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

This does a five-ply search

The Horizon effect

A potential problem in game tree search to a fixed depth is the horizon effect, which occurs when there is a drastic change in value immediately beyond the place where the algorithm stops searching. Consider the tree shown in the below fig.A. it has nodes A, B, C and D. at this level since it is a maximizing ply, the value which will passed up at A is 5.



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

Secondary search

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

Waiting for Quiescence

- If a position looks “dynamic”, don’t even bother to evaluate it.
- Instead, do a small secondary search until things calm down.
- E.g after capturing a piece, things look good, but this would be misleading if opponent was about to capture right back.
- In general such factors are called continuation heuristics

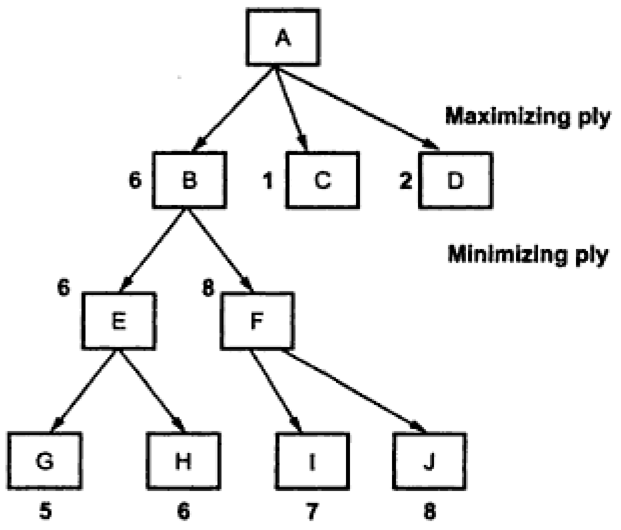


Fig above shows the further exploration of the tree in fig B. Here now since the tree is further explored, the value, which is passed to A is 6. Thus the situation calms down. This is called as waiting for quiescence. This helps in avoiding the horizon effect of a drastic change of values.

Iterative Deepening

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

2.2 Knowledge Representation

Representation and Mapping

- Problem solving requires large amount of knowledge and some mechanism for manipulating that knowledge.
- The Knowledge and the Representation are distinctentities, play a central but distinguishable roles in intelligent system.
    - **Knowledge** is a description of the world;

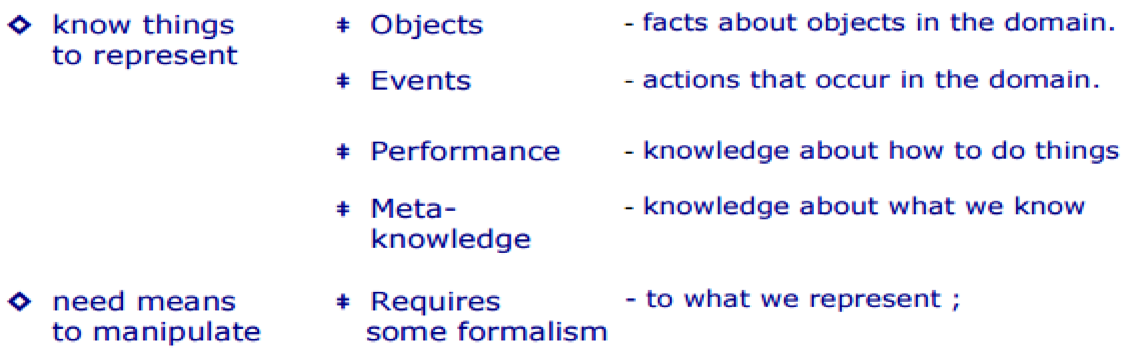
      it determines a *system's competence* by what it knows.
    - **Representation** is the way knowledge is encoded;

      it defines the *system's performance* in doing something.

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

➢ In simple words, we *:*

- need to know about *things we want to represent* , and
- need some means by which *things we can manipulate*.



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

Thus, knowledge representation can be considered at two levels :

➢ knowledge level at which facts are described, and

➢ symbol level at which the representations of the objects, defined in terms of symbols, can be manipulated in the programs.

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

Mapping between Facts and Representation

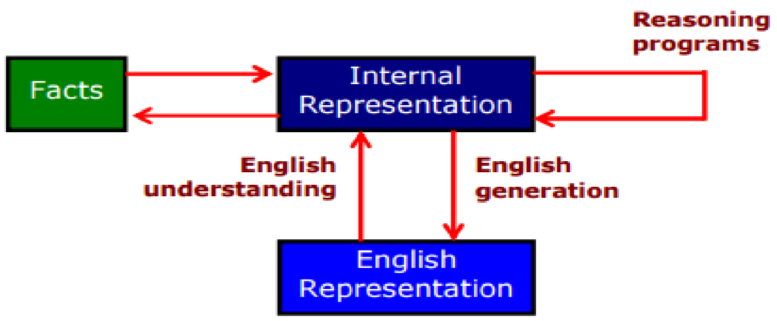
➢ Knowledge is a collection of *"facts"* from some domain.

➢ We need a representation of *"facts"* that can be manipulated by a program. Normal English is insufficient, too hard currently for a computer program to draw inferences in natural languages.

➢ Thus some symbolic representation is necessary.

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

Example : Consider an English sentence



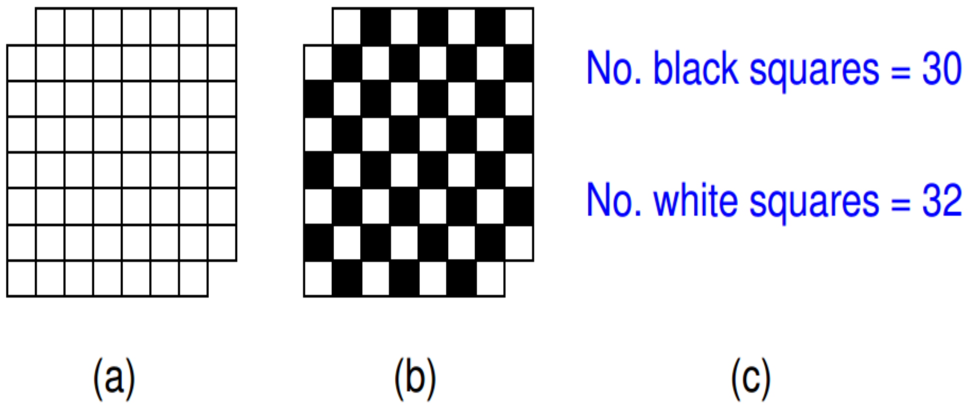
Facts	Representations
◇ **Spot is a dog**	A *fact* represented in *English sentence*
◇ **dog (Spot)**	Using *forward mapping function* the above *fact* is represented in *logic*
◇ **∀ x : dog(x) → hastail (x)**	A *logical representation* of the *fact* that *"all dogs have tails"*

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

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

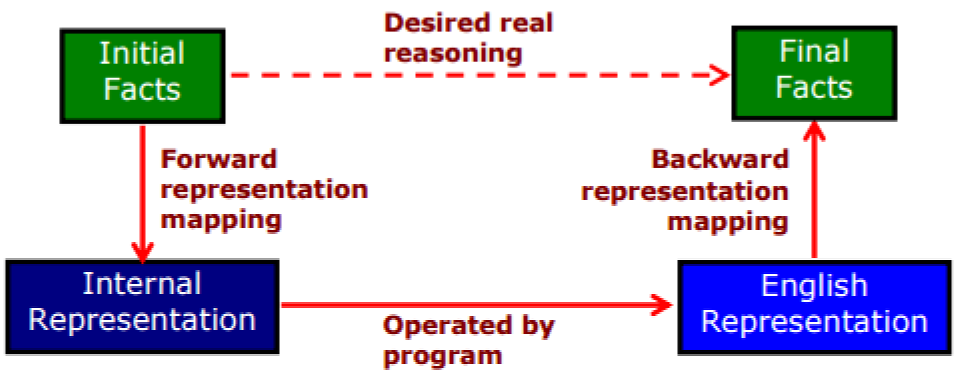
➢ Good representation can make a reasoning program trivial

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



- ➢ Forward and Backward Representation

  The forward and backward representations are elaborated below



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

KR System Requirements

- ➢ A good knowledge representation enables fast and accurate access to knowledge and understanding of the content.
- ➢ A knowledge representation system should have following properties.

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

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

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

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

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

2.3 Knowledge Representation Schemes

There are four types of Knowledge representation :

Relational, Inheritable, Inferential, and Declarative/Procedural.

- **Relational Knowledge :**
  - provides a framework to compare two objects based on equivalent attributes.
  - any instance in which two different objects are compared is a relational type of knowledge.
- **Inheritable Knowledge**
  - is obtained from associated objects.
  - it prescribes a structure in which new objects are created which may inherit all or a subset of attributes from existing objects.
- **Inferential Knowledge**
  - is inferred from objects through relations among objects.
  - e.g., a word alone is a simple syntax, but with the help of other words in phrase the reader may infer more from a word; this inference within linguistic is called semantics.
- **Declarative Knowledge**
  - a statement in which knowledge is specified, but the use to which that knowledge is to be put is not given.
  - e.g. laws, people's name; these are facts which can stand alone, not dependent on other knowledge;
- **Procedural Knowledge**

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

Relational Knowledge

This knowledge associates elements of one domain with another domain.

- Relational knowledge is made up of objects consisting of attributes and their corresponding associated values.
- The results of this knowledge type is a mapping of elements among different domains.

The table below shows a simple way to store facts.

- The facts about a set of objects are put systematically in columns.
- This representation provides little opportunity for inference.

Table - Simple Relational Knowledge

Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

- ✓ Given the facts it is not possible to answer simple question such as :

  " Who is the heaviest player ? ".

  but if a procedure for finding heaviest player is provided, then these facts will enable that procedure to compute an answer.
- ✓ We can ask things like who "bats – left" and "throws – right".

Inheritable Knowledge

- ➢ Here the knowledge elements inherit attributes from their parents.

- ➢ The knowledge is embodied in the design hierarchies found in the functional, physical and process domains. Within the hierarchy, elements inherit attributes from their parents, but in many cases not all attributes of the parent elements be prescribed to the child elements.
- ➢ The *inheritance* is a powerful form of inference, but not adequate. The basic KR needs to be augmented with inference mechanism.
- ➢ The KR in hierarchical structure, shown below, is called *"semantic network"* or a collection of *"frames" or "slot-and-filler structure"*. The structure shows property inheritance and way for insertion of additional knowledge.
- ➢ Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes. The classes are organized in a generalized hierarchy.

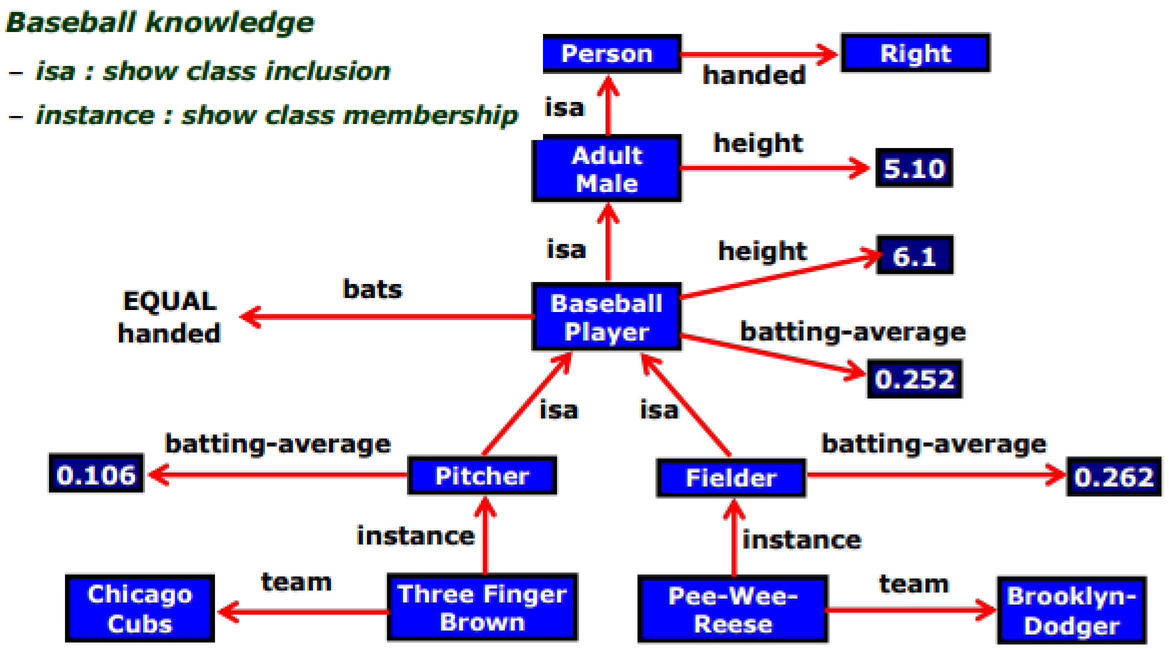


Fig. Inheritable knowledge representation (KR)

- ✓ The directed arrows represent *attributes* (*isa, instance, team*) originates at object being described and terminates at object or its value.
- ✓ The box nodes represents *objects* and *values* of the attributes.

- ➢ Viewing a node as a frame

Example :	Baseball-player
isa :	Adult-Male

Bates :	EQUAL handed
Height :	6.1
Batting-average :	0.252

- ➢ Algorithm : Property Inheritance

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

Steps to follow:

1. Find object **O** in the knowledge base.
2. If there is a value for the attribute **A** then report that value.
3. Else, if there is a value for the attribute instance; If not, then fail.
4. Else, move to the node corresponding to that value and look for a value for the attribute **A**; If one is found, report it.
5. Else, do until there is no value for the "**isa**" attribute or until an answer is found :

   (a) Get the value of the "**isa**" attribute and move to that node.

   (b) See if there is a value for the attribute **A**; If yes, report it.

- ➢ This algorithm is simple. It describes the basic mechanism of inheritance. It does not say what to do if there is more than one value of the instance or "isa" attribute.
- ➢ This can be applied to the example of knowledge base illustrated, in the previous slide, to derive answers to the following queries :
  - ✓ team (Pee-Wee-Reese) = Brooklyn–Dodger
  - ✓ batting–average(Three-Finger-Brown) = 0.106
  - ✓ height (Pee-Wee-Reese) = 6.1
  - ✓ bats (Three Finger Brown) = right

Inferential Knowledge

- ➢ This knowledge generates new information from the given information.
- ➢ This new information does not require further data gathering form source, but does require analysis of the given information to generate new knowledge.

Example :

– given a set of relations and values, one may infer other values or relations.

– a predicate logic (a mathematical deduction) is used to infer from a set of attributes.

– inference through predicate logic uses a set of logical operations to relate individual data.

– the symbols used for the logic operations are :

" → " (implication), " ¬ " (not), " V " (or), " Λ " (and),
" ∀ " (for all), " ∃ " (there exists).

Examples of predicate logic statements :

1. *"Wonder"* is a name of a dog :	**dog (wonder)**
2. All dogs belong to the class of animals :	**∀ x : dog (x) → animal(x)**
3. All animals either live on land or in water :	**∀ x : animal(x) → live (x, land) V live (x, water)**

From these three statements we can infer that :

" Wonder lives either on land or on water."

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

Declarative/Procedural Knowledge

Differences between Declarative/Procedural knowledge is not very clear.

Declarative knowledge :

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

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

✓ domains represent the physical world and the perceived functionality.

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

Procedural knowledge:

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

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

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

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

Issues in Knowledge Representation

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

    Any attribute of objects so basic that they occur in almost every problem domain ?
  - ✓ Relationship among attributes:

    Any important relationship that exists among object attributes ?
  - ✓ Choosing Granularity :

    At what level of detail should the knowledge be represented ?
  - ✓ Set of objects :

    How sets of objects be represented ?
  - ✓ Finding Right structure :

    Given a large amount of knowledge stored, how can relevant parts be accessed ?
- **Important Attributes**
  - ✓ There are attributes that are of general significance.
  - ✓ There are two attributes "instance" and "isa", that are of general importance. These attributes are important because they support property inheritance.
- **Relationship among Attributes**
  - ✓ The attributes to describe objects are themselves entities they represent.
  - ✓ The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:

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

  - ▪ **Inverses :**

  This is about consistency check, while a value is added to one attribute. The entities are related to each other in many different ways. The figure shows attributes (isa, instance, and team), each with a directed arrow, originating at the object being described and terminating either at the object or its value.

  There are two ways of realizing this:

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

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

  - ▪ Existence in an "isa" hierarchy

    This is about generalization-specialization, like, classes of objects and specialized subsets of those classes. There are attributes and specialization of attributes.

    Example: the attribute "height" is a specialization of general attribute "physical-size" which is, in turn, a specialization of "physical-attribute". These generalization-specialization relationships for attributes are important because they support inheritance.

  - ▪ Techniques for reasoning about values

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

    height : must be in a unit of length,

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

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

  - ▪ Single valued attributes

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

Example : A baseball player can at time have only a single height and be a member of only one team. KR systems take different approaches to provide support for single valued attributes.

- ➢ Choosing Granularity

  What level should the knowledge be represented and what are the primitives ?

  - ✓ Should there be a small number or should there be a large number of low-level primitives or High-level facts.
  - ✓ High-level facts may not be adequate for inference while Low-level primitives may require a lot of storage.

    Example of Granularity :

    - Suppose we are interested in following facts

      John spotted Sue.
    - This could be represented as

      Spotted (agent(John), object (Sue))
    - Such a representation would make it easy to answer questions such are

      Who spotted Sue ?
    - Suppose we want to know

      Did John see Sue ?
    - Given only one fact, we cannot discover that answer.
    - We can add other facts, such as

      Spotted (x , y) -> saw (x , y)
    - We can now infer the answer to the question.
- ➢ Set of Objects
  - ✓ Certain properties of objects that are true as member of a set but not as individual;

    Example : Consider the assertion made in the sentences "there are more sheep than people in Australia", and "English speakers can be found all over the world."

- ✓ To describe these facts, the only way is to attach assertion to the sets representing people, sheep, and English.
- ✓ The reason to represent sets of objects is :

  If a property is true for all or most elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every elements of the set.
- ✓ This is done in different ways :
  - in logical representation through the use of universal quantifier, and
  - in hierarchical structure where node represent sets, the inheritance propagate set level assertion down to individual.

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

- ✓ whether we are asserting some property of the set itself, means, the set of elephants is large, or
- ✓ asserting some property that holds for individual elements of the set , means, any thing that is an elephant is large.

There are three ways in which sets may be represented :

- ✓ Name, as in the example. Inheritable KR, the node - Baseball- Player and the predicates as Ball and Batter in logical representation.
- ✓ Extensional definition is to list the numbers, and
- ✓ In tensional definition is to provide a rule, that returns true or false depending on whether the object is in the set or not.

- ➢ **Finding Right Structure**
  - ✓ Access to right structure for describing a particular situation.
  - ✓ It requires, selecting an initial structure and then revising the choice. While doing so, it is necessary to solve following problems :
    - how to perform an initial selection of the most appropriate structure.
    - how to fill in appropriate details from the current situations.
    - how to find a better structure if the one chosen initially turns out not to be appropriate.
    - what to do if none of the available structures is appropriate.

- when to create and remember a new structure.

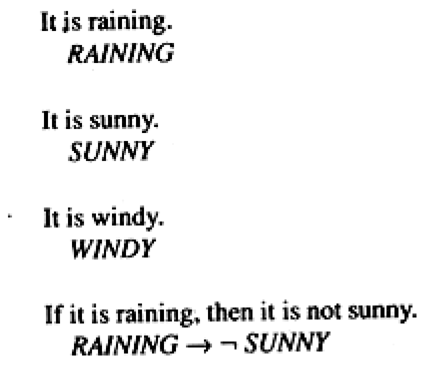
✓ There is no good, general purpose method for solving all these problems. Some knowledge representation techniques solve some of them.

CHAPTER 2

Knowledge Representation using predicate logic

2.4 Representing Simple Facts in Logic

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



It is raining.
RAINING

It is sunny.
SUNNY

It is windy.
WINDY

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

Some simple facts in Propositional logic

Socrates is a man.

We could write:

SOCRATESMAN

But if we also wanted to represent

Plato is a man.

we would have to write something such as:

PLATOMAN

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

MAN(SOCRATES)

MAN(PLATO)

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

All men are mortal.

We could represent this as:

MORTALMAN

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

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

1 . Marcus was a man.

2. Marcus was a Pompeian.

3. All Pompeians were Romans.

4. Caesar was a ruler.

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

6. Everyone is loyal to someone.

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

8. Marcus tried to assassinate Caesar.

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

1. Marcus was a man.

man(Marcus)

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

2. Marcus was a Pompeian.

Pompeian(Marcus)

3.All Pompeians were Romans.

$\forall x$: Pompeian(x)$\rightarrow$ Roman(x)

4.Caesar was a ruler.

ruler(Caesar)

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

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

∀x: Roman(x)→ loyalto(x, Caesar) V hate(Caesar)

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

∀x: Roman(x) → [(loyalto(x, Caesar) V hate(x, Caesar))∧

Not (loyalto(x, Caesar) ∧hate(x, Caesar))]

6. Everyone is loyal to someone.

∀x:∃y : loyalto(x,y)

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

$$\begin{tikzcd}
\neg loyalto(Marcus, Caesar) \\
\begin{array}{l} person(Marcus) \land \\ ruler(Caesar) \land \\ tryassassinate(Marcus, Caesar) \end{array} \arrow[u, "\text{(7, substitution)}"'] \\
\begin{array}{l} person(Marcus) \\ tryassassinate(Marcus, Caesar) \end{array} \arrow[u, "(4)"'] \\
person(Marcus) \arrow[u, "(8)"']
\end{tikzcd}$$

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

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

∀x : ∀y : person(x) ∧ ruler(y) ∧ tryassasinate(x,y) → ¬loyalto(x,y)

8.Like the previous one this sentence too is ambiguous which may lead to more than one conclusion. The usage of “try to assassinate” as a single predicate gives us a fairly simple representation with which we can reason about trying to assassinate. But there might be connections as try to assassinate and not actually assassinate could not be made easily.

9. Marcus tried to assassinate Caesar.

tryassasinate (Marcus,Caesar)

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

Was Marcus loyal to Caesar?

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

$\neg$loyalto(Marcus, Caesar)

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

10. All men are people.

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

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

2.4.1 Representing Instance and isa relationships

- Knowledge can be represented as classes, objects, attributes and Super class and sub class relationships.
- Knowledge can be inference using property inheritance. In this elements of specific classes inherit the attributes and values.
- Attribute instance is used to represent the relationship “Class membership ” (element of the class)
- Attribute isa is used to represent the relationship “Class inclusion” (super class, sub class relationship)

Three ways of representing class membership

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

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

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

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

2.4.2 Computable functions and predicates

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

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

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

Computable functions: gt(2+4, 5)

Consider the following set of facts, again involving Marcus:

1. marcus was a man

man(Marcus)

2. Marcus was a pompeian

Pompeian(Marcus)

3. Marcus was born in 40 A.D

born(marcus, 40)

4. All men are mortal

$\forall$x: men(x)$\rightarrow$ mortal(x)

5. All Pompeians died when the volcano erupted in 79 A.D

erupted(volcano,79) & x :pompeian(x)$\rightarrow$died(x, 79)

6. No mortal lives longer than150 years

$\forall$x: $\forall$t1: $\forall$t2: mortal(x) & born(x,t1) & gt(t2-t1,150)$\rightarrow$ dead(x,t1)

7. It is Now 1991

Now=1991

8. Alive means not dead

$\forall$x: $\forall$t: [alive(x,t) $\rightarrow$~dead(x,t)] & [~dead(x,t)$\rightarrow$alive(x,t)]

9. If someone dies then he is dead at all later times

$\forall$x: $\forall$t1: $\forall$t2: died(x,t1) & gt(t2,t1)$\rightarrow$ dead(x1,t2)

This representation says that one is dead in all years after the one in which one died. It ignores the question of whether one is dead in the year in which one died.

1. man(Marcus)

2. Pompeian(Marcus)

3. born(marcus, 40)

4. $\forall$x: men(x)$\rightarrow$ mortal(x)

5. ∀:pompeian(x)→died(x, 79)

6. erupted(volcano,79)

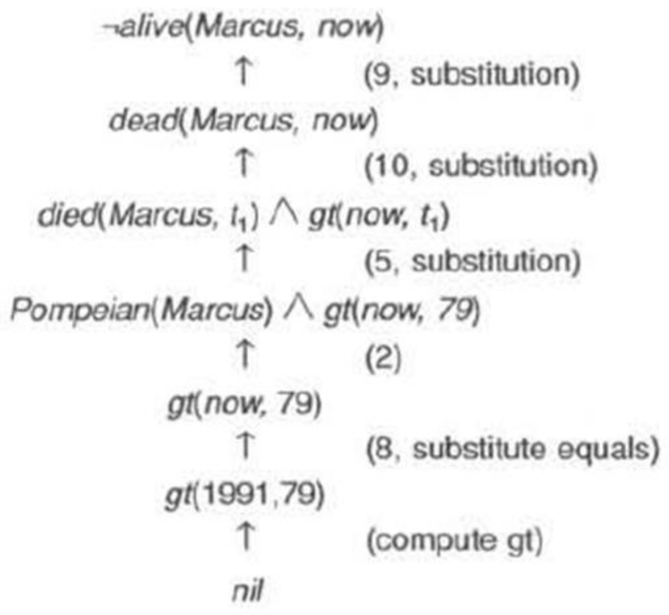
7. ∀ x: ∀t1: ∀t2: mortal(x) & born(x,t1) & gt(t2-t1,150)→

dead(x,t1)

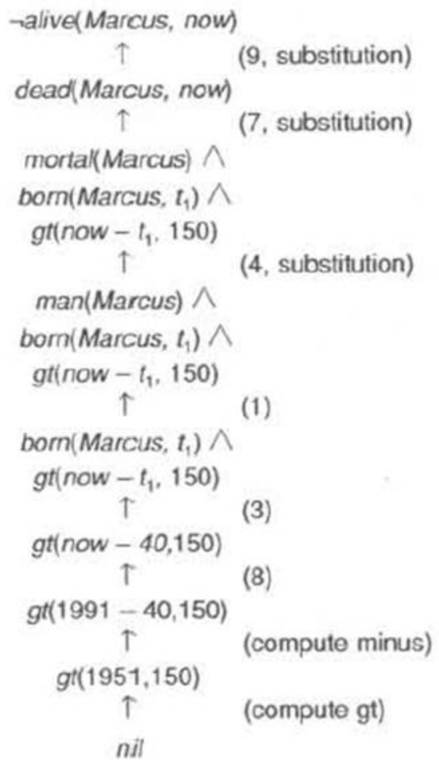
8. Now=1991

9. ∀x: ∀ t: [alive(x,t) →~dead(x,t)] &[~dead(x,t)→alive(x,t)]

10. ∀ x: ∀t1: ∀t2: died(x,t1) & gt(t2,t1)→ dead(x1,t2)



One Way of Proving That Marcus Is Dead



$$\begin{array}{ll}
\neg alive(Marcus, now) & \\
\uparrow & (9,\ \text{substitution}) \\
dead(Marcus, now) & \\
\uparrow & (7,\ \text{substitution}) \\
mortal(Marcus) \wedge & \\
born(Marcus, t_1) \wedge & \\
gt(now - t_1, 150) & \\
\uparrow & (4,\ \text{substitution}) \\
man(Marcus) \wedge & \\
born(Marcus, t_1) \wedge & \\
gt(now - t_1, 150) & \\
\uparrow & (1) \\
born(Marcus, t_1) \wedge & \\
gt(now - t_1, 150) & \\
\uparrow & (3) \\
gt(now - 40, 150) & \\
\uparrow & (8) \\
gt(1991 - 40, 150) & \\
\uparrow & (\text{compute minus}) \\
gt(1951, 150) & \\
\uparrow & (\text{compute gt}) \\
nil &
\end{array}$$

Another Way of Proving That Marcus is Dead

Two things should be clear from the proofs we have just shown:

- Even very simple conclusions can require many steps to prove.
- A variety of processes, such as matching, substitution, and application of modus ponens are involved in the production of a proof, This is true even for the simple statements we are using, It would be worse if we had implications with more than a single term on the right or with complicated expressions involving ands and ors on the left.

Disadvantage:

- ➢ Many steps required to prove simple conclusions
- ➢ Variety of processes such as matching and substitution used to prove simple conclusions

2.5 Resolution

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

Algorithm: Convert to Clause Form

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

∀x: ¬ [Roman(x) ∧ know(x, Marcus)] V

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

2. Reduce the scope of each ¬ to a single term, using the fact that ¬ (¬ p) = p, deMorgan's laws [which say that ¬ (a ∧ b) = ¬ a V ¬ b and ¬ (a V b) = ¬ a ∧ ¬ b], and the standard correspondences between quantifiers [¬ ∀x: P(x) = ∃x: ¬ P(x) and ¬ ∃x: P(x) = ∀ x: ¬P(x)]. Performing this transformation on the wff from step 1 yields

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

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

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

∀x: P(x) V ∀x: Q(x)

would be converted to

∀x: P(x) V ∀y: Q(y)

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

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

[hate(x, Caesar) V (¬ hale(y, z) V thinkcrazy(x, y))]

At this point, the formula is in what is known as prenex normal form. It consists of a prefix of quantifiers followed by a matrix, which is quantifier-free.

5. Eliminate existential quantifiers. A formula that contains an existentially quantified variable asserts that there is a value that can be substituted for the variable that makes the formula true. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. Since we do not necessarily know how to produce the value, we must create a new function name for every such replacement. We make no assertions about these functions except that they must exist. So, for example, the formula

∃y : President(y)

can be transformed into the formula

President(S1)

where S1 is a function with no arguments that somehow produces a value that satisfies President. If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example, in the formula

∀x: ∃y: father-of(y, x)

the value of y that satisfies father-of depends on the particular value of x. Thus we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this example would be transformed into

∀x: father-of(S2(x), x)

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

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

[¬ Roman(x) V ¬ know(x, Marcus)] V

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

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

¬ Roman(x) V ¬ know(x, Marcus) V

hate(x, Caesar) V ¬ hate(y, z) V thinkcrazy(x, y)

However, it is also frequently necessary to exploit the distributive property [i.e. , (a ∧ b) V c = (a V c) ∧ (b V c)]. For example, the formula

(winter ∧ wearingboots) V (summer ∧ wearingsandals)

Becomes, after one application of the rule

[winter V (summer ∧ wearingsandals)]

∧ [wearingboots V (summer ∧ wearingsandals)]

And then, after a second application, required since there are still conjuncts joined by OR's, (winter V summer) ∧

(winter V wearingsandals) ∧

(wearingboots V summer) ∧

(wearingboots V wearingsandals)

8. Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the same set of facts as were represented by the original wff's.

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

$(\forall x: P(x) \wedge Q(x)) = \forall x: P(x) \wedge \forall x: Q(x)$

Thus since each clause is a separate conjunct and since all the variables are universally quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Performing this final step of standardization is important because during the resolution procedure it is sometimes necessary to instantiate a universally quantified variable (i.e., substitute for it a particular value). But, in general, we want to keep clauses in their most general form as long as possible. So when a variable is instantiated, we want to know the minimum number of substitutions that must be made to preserve the truth value of the system.

After applying this entire procedure to a set of wff's, we will have a set of clauses, each of which is a disjunction of *literals.* These clauses can now be exploited by the resolution procedure to generate proofs.

2.5.1 Resolution in Propositional Logic

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

ALGORITHM: PROPOSITIONAL RESOLUTION

1. Convert all the propositions of F to clause form

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

3. Repeat until either a contradiction is found or no progress can be made:

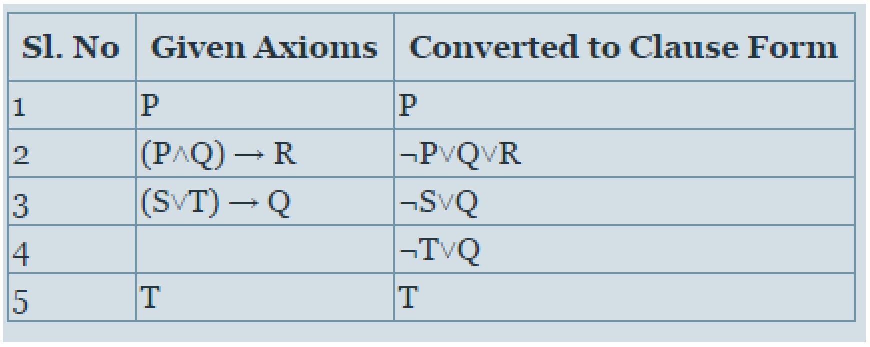
a) Select two clauses. Call these the parent clauses.

b) Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and ¬L such that one of the parent clauses contains L and the other contains ¬L, then select one such pair and eliminate both L and ¬L from the resolvent.

c) If the resolvent is the empty clause, then a contradiction has been found. If it is not then add it to the set of clauses available to the procedure.

Suppose we are given the axioms shown in the first column of Table 1 and we want to prove R.First we convert the axioms to clause which is already in clause form. Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of sequence of resolvents shown in figure 1. We begin by resolving with the clause ɷR since that is one of the clauses that must be involved in the contradiction we are trying to find.

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause.



Sl. No	Given Axioms	Converted to Clause Form
1	P	P
2	(P∧Q) → R	¬P∨Q∨R
3	(S∨T) → Q	¬S∨Q
4		¬T∨Q
5	T	T

Table 1 Some Facts in Propositional Logic

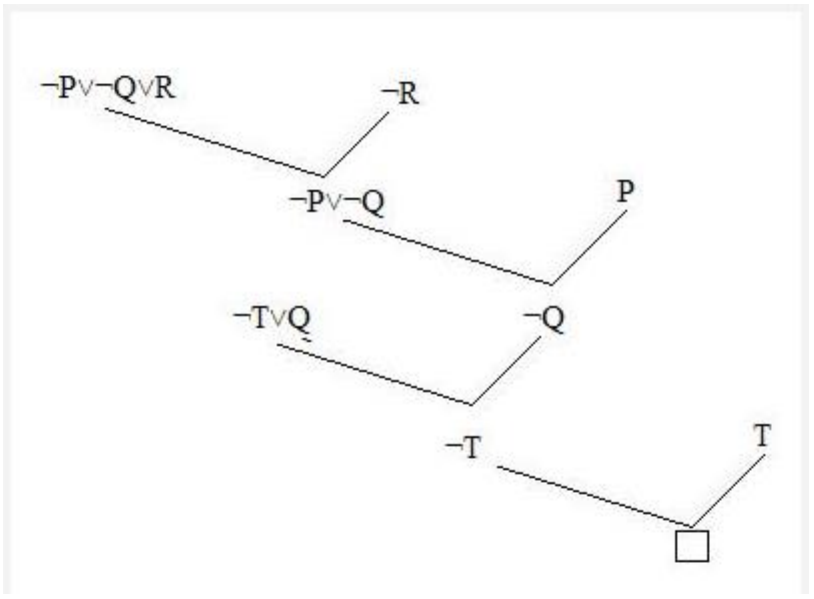


Figure : Resolution In Propositional Logic

2.5.3 UNIFICATION ALGORITHM

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

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

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

(tryassassinate Marcus Caesar)

(tryassassinate Marcus (ruler of Rome))

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

(try assassinate Marcus Caesar)

(hate Marcus Caesar)

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

i) Different constants , functions or predicates can not match, whereas identical ones can.

ii) A variable can match another variable , any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).

iii) The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent. (a substitution y for x written as y/x)

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

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

Algorithm: Unify(L1, L2)

```
I. If L1 or L2 are both variables or constants, then:
    (a) If L1 and L2 are identical, then return NIL.
    (b) Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
    (c) Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL} , else return (L1/L2).
    (d) Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.
3. If LI and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2.)
5. For i ← 1 to number of arguments in L1 :
    (a) Call Unify with the ith argument of L1 and the ith argument of L2, putting result in S.
    (b) If S contains FAIL then return {FAIL}.
    (c) If S is not equal to NIL then:
        (i) Apply S to the remainder of both L1 and L2.
```

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

6. Return SUBST.

2.5.4 Resolution in Predicate Logic

ALGORITHM: RESOLUTION IN PREDICATE LOGIC

1. Convert all the statements of F to clause form

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

3. Repeat until either a contradiction is found or no progress can be made or a predetermined amount of effort has been expended:

a) Select two clauses. Call these the parent clauses.

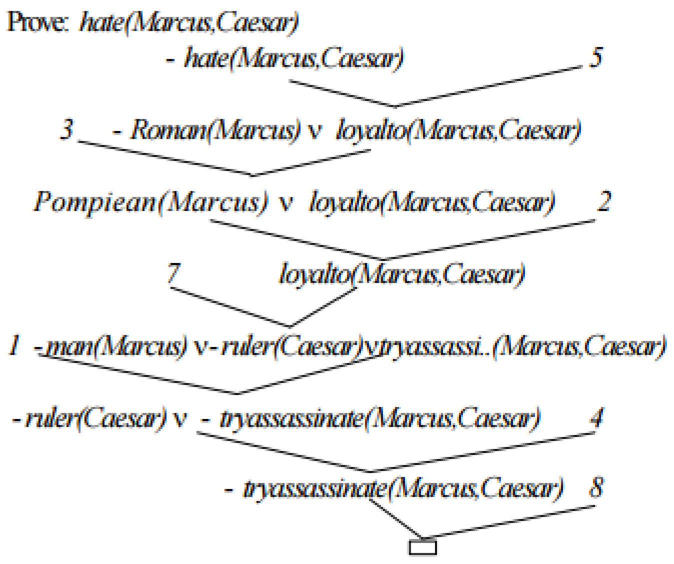
b) Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both the parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals T1 and T2 such that one of the parent clauses contains T1 and the other contains T2 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 complementary literals. Use the substitution produced by the unification to create the resolvent. If there is one pair of complementary literals, only one such pair should be omitted from the resolvent.

c) If the resolvent is the empty clause, then a contradiction has been found. If it is not then add it to the set of clauses available to the procedure.

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

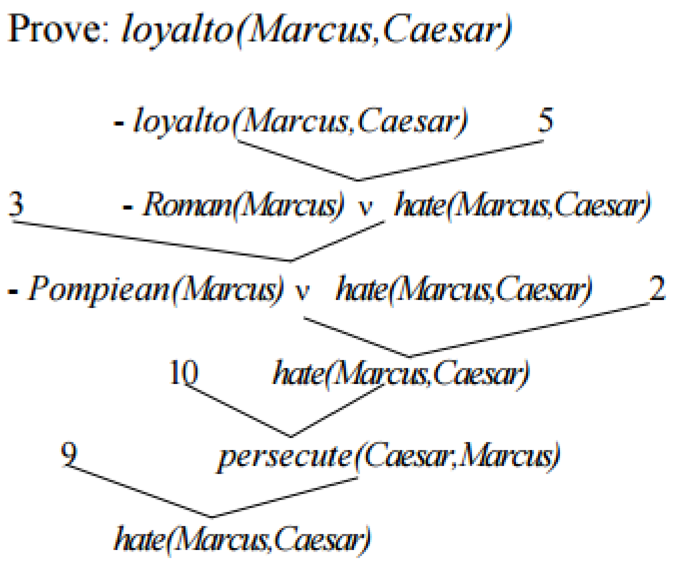
Axioms in clause form

1.*man(Marcus)*
2.*Pompiean(Marcus)*
3.*- Pompiean(x1) v Roman(x1)*
4.*ruler(Caesar)*
5.*- Roman(x2) v loyalto(x2,Caesar) v hate(x2,Caesar)*
6. *loyal(x3,f(x3))*
7.*-man(x4) v -ruler(y1) v - tryassassinate(x4,y1) v loyalto(x4,y1)*
8.*tryassassinate(Marcus,Caesar)*



Prove: hate(Marcus,Caesar)
- hate(Marcus,Caesar)
5
3
- Roman(Marcus) ∨ loyalto(Marcus,Caesar)
Pompiean(Marcus) ∨ loyalto(Marcus,Caesar)
2
7
loyalto(Marcus,Caesar)
1 -man(Marcus) ∨ -ruler(Caesar) ∨ tryassassi..(Marcus,Caesar)
-ruler(Caesar) ∨ - tryassassinate(Marcus,Caesar)
4
- tryassassinate(Marcus,Caesar)
8

A Resolution Proof



Prove: loyalto(Marcus,Caesar)
- loyalto(Marcus,Caesar)
5
3
- Roman(Marcus) ∨ hate(Marcus,Caesar)
- Pompiean(Marcus) ∨ hate(Marcus,Caesar)
2
10
hate(Marcus,Caesar)
9
persecute(Caesar,Marcus)
hate(Marcus,Caesar)

An Unsuccessful Attempt at Resolution

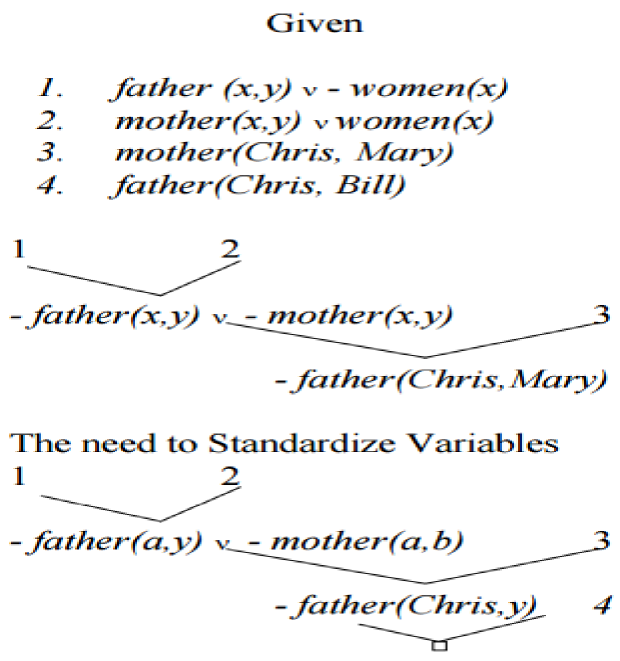
9. persecute(x, y) → hate(y, x)

10. hate(x,y) → persecute(y, x)

Converting to clause form, we get

9. $\neg$ persecute(x_5,y_2) $\vee$ hate(y_2,x_5)

10. $\neg$hate(x_6,y_3) $\vee$ persecute(y_3,x_6)


Given
1. father (x,y) v - women(x)
2. mother(x,y) v women(x)
3. mother(Chris, Mary)
4. father(Chris, Bill)
1
2
- father(x,y) v - mother(x,y)
3
- father(Chris,Mary)
The need to Standardize Variables
1
2
- father(a,y) v - mother(a,b)
3
- father(Chris,y)
4

2.5.5 Procedural v/s Declarative Knowledge

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

  man(Marcus)

  man(Caesar)

  person(Cleopatra)

  ∀x: man(x)→ person(x)

  person(x)?

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

Ǝy : person (y)

Marcus, Ceaser and Cleopatra can be the answers

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

man(Marcus)

man (Ceaser)

Vx : man(x)

person(x)

Person(Cleopatra)

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

2.5.6 Logic Programming

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

- Eg p, ¬ p V q etc are also Horn Clauses.
- The fact that PROLOG programs are composed only of Horn Clauses and not of arbitrary logical expressions has two important consequences.
- Because of uniform representation a simple & effective interpreter can be written.
- The logic of Horn Clause systems is decidable.
- Even PROLOG works on backward reasoning.
- The program is read top to bottom, left to right and search is performed depth-first with backtracking.
- There are some syntactic difference between the logic and the PROLOG representations as mentioned
- The key difference between the logic & PROLOG representation is that PROLOG interpreter has a fixed control strategy, so assertions in the PROLOG program define a particular search path to answer any question. Whereas Logical assertions define set of answers that they justify, there can be more than one answers, it can be forward or backward tracking.
- Control Strategy for PROLOG states that we begin with a problem statement, which is viewed as a goal to be proved.

- Look for the assertions that can prove the goal.
- To decide whether a fact or a rule can be applied to the current problem, invoke a standard unification procedure.
- Reason backward from that goal until a path is found that terminates with assertions in the program.
- Consider paths using a depth-first search strategy and use backtracking.
- Propagate to the answer by satisfying the conditions.

2.5.7 Forward v/s Backward Reasoning

- The objective of any search is to find a path through a problem spacefrom the initial to the final one.
- There are 2 directions to go and find the answer

  - ✓ Forward
  - ✓ Backward
- ➢ 8-square problem
- ➢ Reason forward from the initial states: Begin building a tree of move sequences that might be solution by starting with the initialconfiguration(s) at the root of the tree. Generate the next level of tree by finding all the rules whose left sides match the root node and use the right sides to create the new configurations. Generate each node by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue.
- ➢ Reason backward from the goal states: Begin building a tree of move sequences that might be solution by starting with the goal configuration(s) at the root of the tree. Generate the next level of tree by finding all the rules whose right sides match the root node and use the left sides to create the new configurations. Generate each node by taking each node generated at the previous level and applying to it all of the rules whose right sides match it. Continue. This is also called Goal-Directed Reasoning.
- ➢ To summarize, to reason forward, the left sides (pre-conditions) are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached.
- ➢ To reason backwards, the right sides are matched against the current node and the left sides are used to generate new nodes.
- ➢ Factors that influence whether to choose forward or backward reasoning:
  - ✓ Are there more possible start states or goal states? We would like to go from smaller set of states to larger set of states.
  - ✓ In which direction is the branching factor (the average number of nodes that can be reached directly from a single node) greater? We would like to proceed in the direction with the lower branching factor.
  - ✓ Will the program be asked to justify its reasoning process to the user? It so, it is important to proceed in the direction that corresponds more closely with the way user will think.
  - ✓ What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact , forward reasoning should be used. If it a query to which response is desired, use backward reasoning.
- ➢ Home to unknown place example.
- ➢ MYCIN

- Bidirectional Search (The two searches must pass each other)
- Forward Rules: which encode knowledge about how to respond to certain input configurations.
- Backward Rules: which encode knowledge about how to achieve particular goals.
- Backward- Chaining Rule Systems
  - ✓ PROLOG is an example of this.
  - ✓ These are good for goal-directed problem solving.
  - ✓ Hence Prolog & MYCIN are examples of the same.
- Forward - Chaining Rule Systems
  - ✓ We work on the incoming data here.
  - ✓ The left sides of rules are matched with against the state description.
  - ✓ The rules that match the state dump their right side assertions into the state.
  - ✓ Matching is more complex for forward chaining systems.
  - ✓ OPS5, Brownston etc. are the examples of the same.
- Combining Forward v/s Backward Reasoning
  - ✓ Patients example of diagnosis.
  - ✓ In some systems, this is only possible in reversible rules.

Matching

- Till now we have used search to solve the problems as the application of appropriate rules.
- We applied them to individual problem states to generate new states to which the rules can then be applied, until a solution is found.
- We suggest that a clever search involves choosing from among the rules that can be applied at a particular point, but we do not talk about how to extract from the entire collection of rules those that can be applied at a given point.
- To do this we need matching.

Indexing

- Do a simple search through all the rules, comparing each one's precondition to the current state and extracting all the ones that match.
- But this has two problems
- In order to solve very interesting problems, it will be necessary to use a large number of rules, scanning through all of them at every step of the search would be hopelessly inefficient.
- It is not always immediately obvious whether a rule's preconditions are satisfied by a particular state.
- To solve the first problem, use simple indexing. E.g. in Chess, combine all moves at a particular board state together.

Matching with Variables

- The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties that the situations must have.
- Then we need to match a particular situation and the preconditions of a given situation.
- In many rules based systems, we need to compute the whole set of rules that match the current state description. Backward Chaining Systems usually use depth-first backtracking to select individual rules, but forward chaining systems use Conflict Resolution Strategies.
- One efficient many to many match algorithm is RETE

Complex &Approximate Matching

- A more complex matching process is required when the preconditions of a rule specify required properties that are not stated explicitly in thedescription of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others.
- An even more complex matching process is required if rules should be applied if their preconditions approximately match the current situation. Example of listening to a recording of a telephonic conversation.
- For some problems, almost all the action is in the matching of the rulesto the problem state. Once that is done, so few rules apply that theremaining search is trivial. Example ELIZA

Conflict Resolution

- The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable binding were generated by the matching process.
- It is the job of the search method to decide on the order in which the rules will be applied. But sometimes it is useful to incorporate some of the decision making into the matching process. This phase is called conflict resolution.
- There are three basic approaches to the problem of conflict resolution in the production system
  - ✓ Assign a preference based on the rule that matched.
  - ✓ Assign a preference based on the objects that matched.
  - ✓ Assign a preference based on the action that the matched rule would perform.

STRUCTURED REPRESNTATION OF KNOWLEDGE

- Representing knowledge using logical formalism, like predicate logic, has several advantages. They can be combined with powerful inference mechanisms like resolution, which makes reasoning with facts easy. But using logical formalism complex structures of the world, objects and their relationships, events, sequences of events etc. cannot be described easily.
- A good system for the representation of structured knowledge in a particular domain should posses the following four properties:

  (i) Representational Adequacy:- The ability to represent all kinds of knowledge that are needed in that domain.

  (ii) Inferential Adequacy :- The ability to manipulate the represented structure and infer new structures.

  (iii) Inferential Efficiency:- The ability to incorporate additional information into the knowledge structure that will aid the inference mechanisms.

  (iv) Acquisitional Efficiency :- The ability to acquire new information easily, either by direct insertion or by program control.
- The techniques that have been developed in AI systems to accomplish these objectives fall under two categories:

1. Declarative Methods:- In these knowledge is represented as static collection of facts which are manipulated by general procedures. Here the facts need to be stored only one and they can be used in any number of ways. Facts can be easily added to declarative systems without changing the general procedures.

2. Procedural Method:- In these knowledge is represented as procedures. Default reasoning and probabilistic reasoning are examples of procedural methods. In these, heuristic knowledge of "How to do things efficiently "can be easily represented.

- In practice most of the knowledge representation employ a combination of both. Most of the knowledge representation structures have been developed to handle programs that handle natural language input. One of the reasons that knowledge structures are so important is that they provide a way to represent information about commonly occurring patterns of things . such descriptions are some times called schema. One definition of schema is
- "Schema refers to an active organization of the past reactions, or of past experience, which must always be supposed to be operating in any well adapted organic response".
- By using schemas, people as well as programs can exploit the fact that the real world is not random. There are several types of schemas that have proved useful in AI programs. They include

  (i) Frames:- Used to describe a collection of attributes that a given object possesses (eg: description of a chair).

  (ii) Scripts:- Used to describe common sequence of event (eg:- a restaurant scene).

  (iii) Stereotypes :- Used to described characteristics of people.

  (iv) Rule models:- Used to describe common features shared among a set of rules in a production system.
- Frames and scripts are used very extensively in a variety of AI programs. Before selecting any specific knowledge representation structure, the following issues have to be considered.

  (i) The basis properties of objects , if any, which are common to every problem domain must be identified and handled appropriately.

  (ii) The entire knowledge should be represented as a good set of primitives.

  (iii) Mechanisms must be devised to access relevant parts in a large knowledge base.

PART – A

1. How is predicate logic helpful in knowledge representation?
2. Define semantic networks.
3. What is the need of facts and its representation?
4. What is property inheritance?
5. Discuss in brief about ISA and Instance classes.
6. Give some use of conceptual dependency.
7. Define inference.
8. Define logic.
9. Write short notes on uniqueness quantifier.
10. Write short notes on uniqueness operator.
11. Define WWF with an example.
12. Define FOL with an example.
13. Difference between propositional and FOL logic.
14. Define forward chaining and backward chaining.
15. Define Horn clause.
16. Define Canonical horn clause.
17. Write notes on long term and short term memory.
18. Name any 3 frame languages.
19. Write in short about iterative deepening..
20. Is minimax depth fist search or Breadth first search.

PART – B

1. Issues in knowledge representation
2. State Representation of facts in predicate logic.
3. How will you represent facts in propositional logic with an example?
4. Explain Resolution in brief with an example.
5. Write algorithm for propositional resolution and Unification algorithm.

6. Explain in detail about forward and backward chaining with suitable example.

7. Explain steps involved in Matching.

8. Explain the different logics used for knowledge representation.

9. How will you represent facts in Proportional logic with an example.

10. Explain resolution in brief with an example.

11. Explain in detail about minimax procedure.

12. Explain the effect of Alpha beta cut off over minimax.

13. How would the minimax procedure have to be modified to be used by a program playing 3 or 4 persons instead of 2 persons.