UNIT II REPRESENTATION OF KNOWLEDGE

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

GAME PLAYING

Early work on AI focused on formal tasks such as game playing and theorem proving. In game playing to select the next state, search technique is used. There were two reasons that games appeared to be a good domain in which to explore machine intelligence.

(i) Games provide a structured task in which it is easy to measure success or failure
(ii) They did not require large amount of knowledge. They apply a straight forward search and provide a solution from the starting state to a winning state

Unfortunately, the second statement is not true for all. But, it is true for simple games.

- For e.g., consider chess
  - The average branching factor is around 35
  - Each player may make 50 moves
  - It takes 35^{100} positions to examine the complete tree

Thus, in this case, it is difficult to apply a simple straightforward search. Some kind of heuristic search procedure is necessary.

In generate and test procedure, the generator generates the entire proposed solutions and the tester then evaluates. To improve the effectiveness of the test procedure two things can be done.

- Improve the generate procedure so that only good moves are generated.
- Improve the test procedure, so that the best moves will be recognized and explored first.

Two types of generator

(i) Legal move generator – All the possible moves will be generated
(ii) Plausible move generator – Some smaller number of promising moves are generated.

MINIMAX SEARCH ALGORITHM

It is a depth first, depth limited search procedure. The idea is to start at the current position and uses a plausible move generator to generate the set of possible successor positions. Apply static evaluation function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent the evaluation of it. We assume that the static evaluation function returns large values to indicate good situation so our goal is to maximize the value of the static function for the next board position

Principle of Minimax

Given two players(x,y)- the chance of one person winning the game is possible at the expense of other person losing the game.

One ply search

- Maximize the value
- Selecting B's value to A, conclude that A's value is 8.

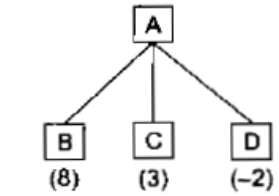
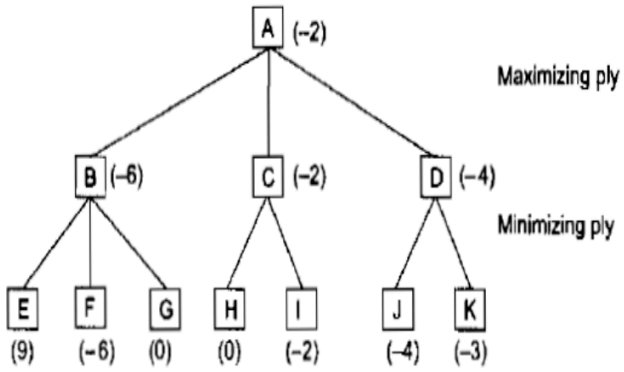


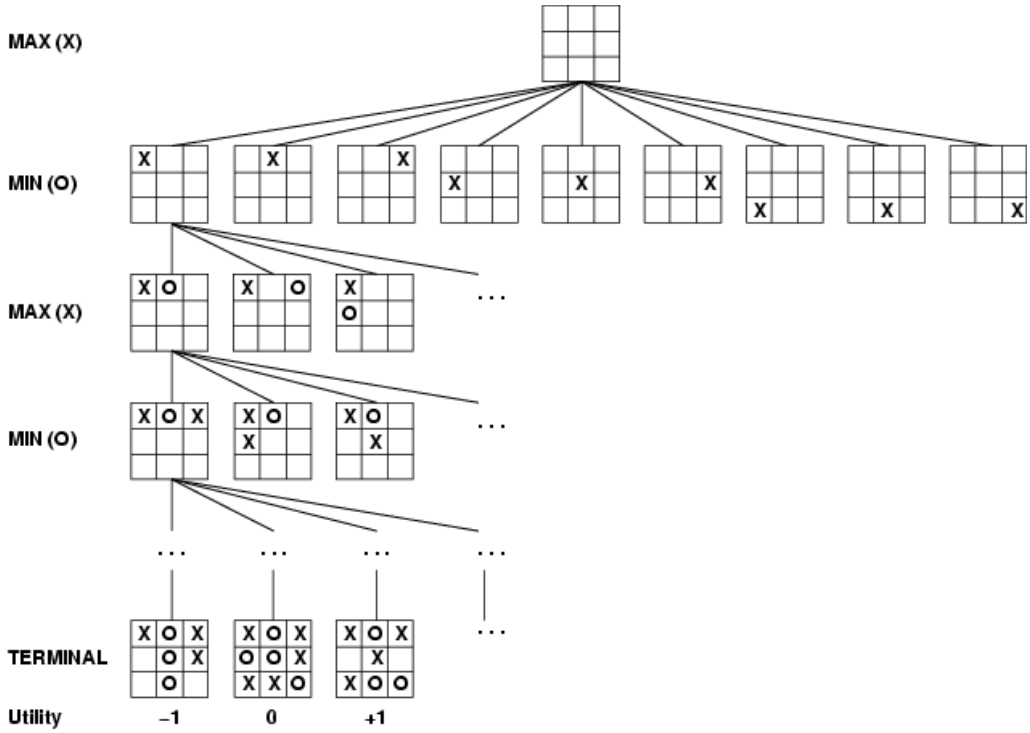
Fig. 12.1 *One-Ply Search*

Two ply search



Backing Up the Values of a Two-Ply Search

The alternation of maximizing and minimizing at alternate ply when evaluations being pushed back up correspond to the opposing strategies of two plays and gives this method a name minimax procedure.



The computer is **Max**.
The opponent is **Min**.

Minimax procedure

- Create start node as a MAX node with current board configuration
- Expand nodes down to some **depth** (a.k.a. **ply**) of lookahead in the game
- Apply the evaluation function at each of the leaf nodes
- "Back up" values for each of the non-leaf nodes until a value is computed for the root node
    - At MIN nodes, the backed-up value is the **minimum** of the values associated with its children.
    - At MAX nodes, the backed up value is the **maximum** of the values associated with its children.
- Pick the operator associated with the child node whose backed-up value determined the value at the root

The important functions in the minimax algorithm are

- MOVEGEN (position, player) – The plausible move generator which return a list of nodes representing the moves that can be made by player in position.
- STATIC(position, player) – Static evaluation function
- DEEP-ENOUGH (position, depth) – It evaluates and return TRUE if the search should be stopped at the current level and FALSE otherwise.

The MNIMAX procedure has to return two results

- The back up value of the path it chooses (VALUE)
- The path itself (PATH)

Algorithm

1. If DEEP-ENOUGH (position, depth), then return structure.

VALUE=STATIC (position, player)

PATH=Nil

This indicates that there is no path from this node.

2. Otherwise generate one more ply of the tree by calling the function MOVE-GEN (position, player) and setting SUCCESSORS to the list it returns.

3. If SUCCESSORS is empty, then there are no moves to be done, so return the same structure that would have been returned by DEEP_ENOUGH.

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.

Initialize BEST-SCORE to the minimum value.

For each element SUCC of SUCCESSORS do the following

(a) Set RESULT-SUCC to MINIMAX (SUCC, Depth+1, OPPOSITE (player)

(b) Set NEW-VALUE to VALUE (RSULT-SUCC)

(c) If NEW-VALUE > BEST-SCORE, then we have found the successor that is better than any that have been examined so far. Record this by doing the following.

(i) Set BEST-SCORE to NEW VALUE.

(ii) The best known path is now from CURRENT to SUCC and then on the appropriate path down from SUCC as determined by the recursive call to MINIMAX.

5. Now that all successors have been examined, we know value of position as well as which path to take from it. So return the structure.

VALUE = BEST-SCORE

PATH = BEST-PATH

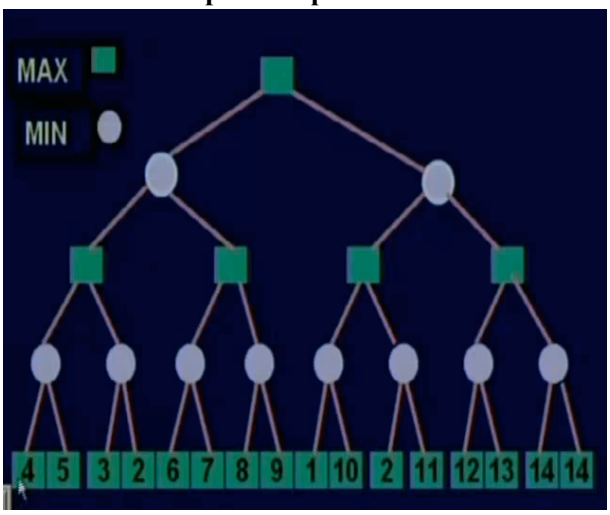
Performance

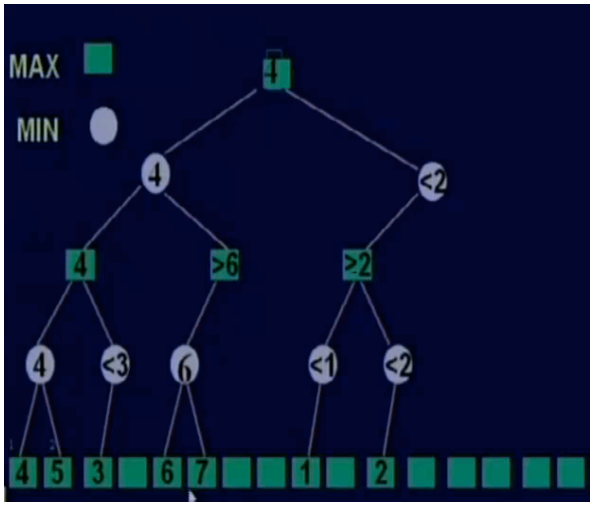
If the maximum depth of the tree is m, and there are b legal moves at each point, then the time complexity of the minimax algorithm is O (b^m).

ALPHA BETA CUT OFF

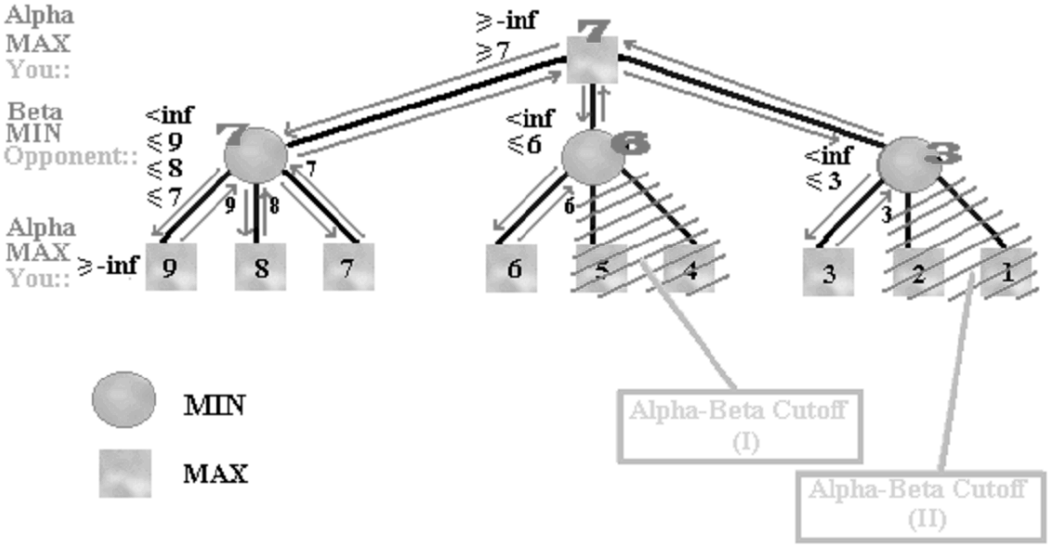
MINIMAX searches the entire tree, even if in some cases the rest can be ignored. But, this alpha beta cutoff returns appropriate minimax decision without exploring entire tree. The minimax search procedure is slightly modified by including branch and bound strategy one for each players. This modified strategy is known as alpha beta pruning. It requires maintenance of two threshold values, one representing a lower bound on the value that a maximizing node may ultimately be assigned (alpha) and another representing upper bound on the value that a minimizing node may be assigned (beta).

Here is an example of Alpha-Beta search:





Another example



Algorithm: MINIMAX-A-B(position, depth, player, Use-Thresh, Pass-Thresh)

1. If DEEP-ENOUGH (position, depth), then return structure.

VALUE=STATIC (position, player)

PATH=Nil

This indicates that there is no path from this node.

2. Otherwise generate one more ply of the tree by calling the function MOVE-GEN (position, player) and setting SUCCESSORS to the list it returns.

3. If SUCCESSORS is empty, then there are no moves to be done, so return the same structure that would have been returned by DEEP_ENOUGH.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.
For each element SUCC of SUCCESSORS do the following

(a) Set RESULT-SUCC to MINIMAX-A-B (SUCC, Depth+1, OPPOSITE (player), Pass-Thresh, Use-Thresh)

(b) Set NEW-VALUE to VALUE (RSULT-SUCC)

(c) If NEW-VALUE > Pass-Thresh, then we have found a successor that is better than any that have been examined so far. Record this by doing the following.

(i) Set Pass-Thresh to NEW VALUE.

(ii) The best known path is now from CURRENT to SUCC and then on the appropriate path down from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to the result of attaching SUCC to the front of PATH (RESULT-SUCC).

5. If Pass-Thresh is not better than Use-Thresh, then we should stop examining this branch. But, both thresholds and values have been inverted. So, if Pass-Thresh>=Use-Thresh, then return immediately with the value.

VALUE = Path-Thresh

PATH = BEST-PATH

5. So return the structure.

VALUE = Path-Thresh

PATH = BEST-PATH

Performance analysis

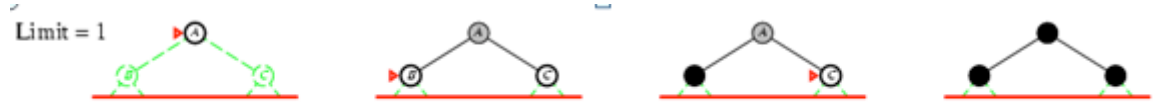
- Guaranteed to compute the same root value as MINIMAX.
- If we choose a best successor first, the need to examine $O(b^{m/2})$ nodes.
- If we choose a random successor, the need to examine $O(b^{3m/4})$ nodes.

ITERATIVE DEEPENING

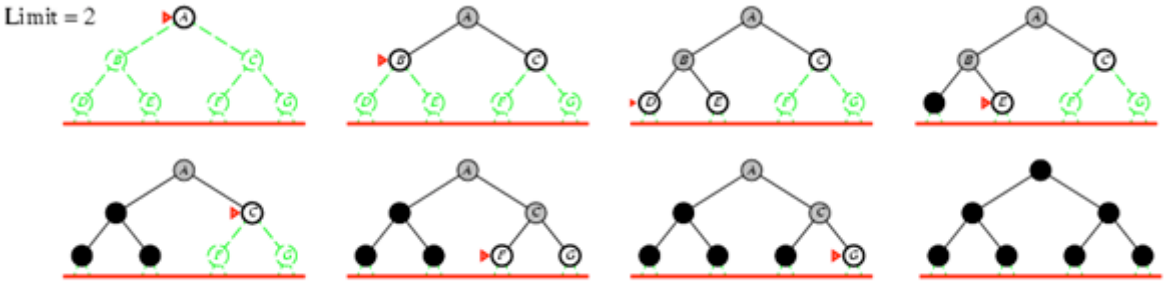
Depth first search was efficient in terms of space but required some cutoff depth in order to force backtracking when a solution was not found. Breadth first search was guaranteed to find the shortest solution path but required inordinate amount of space because all leaf nodes had to be kept in memory. An algorithm called depth first iterative deepening (DFID) combines the best aspects of depth-first and breadth-first search.

To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don't expand beyond depth L. This is called Depth-Limited Search. Suppose if the solution is deeper than L, then increase L iteratively. This is known as Iterative Deepening Search. This iterative deepening search inherits the memory advantage of Depth-First search.

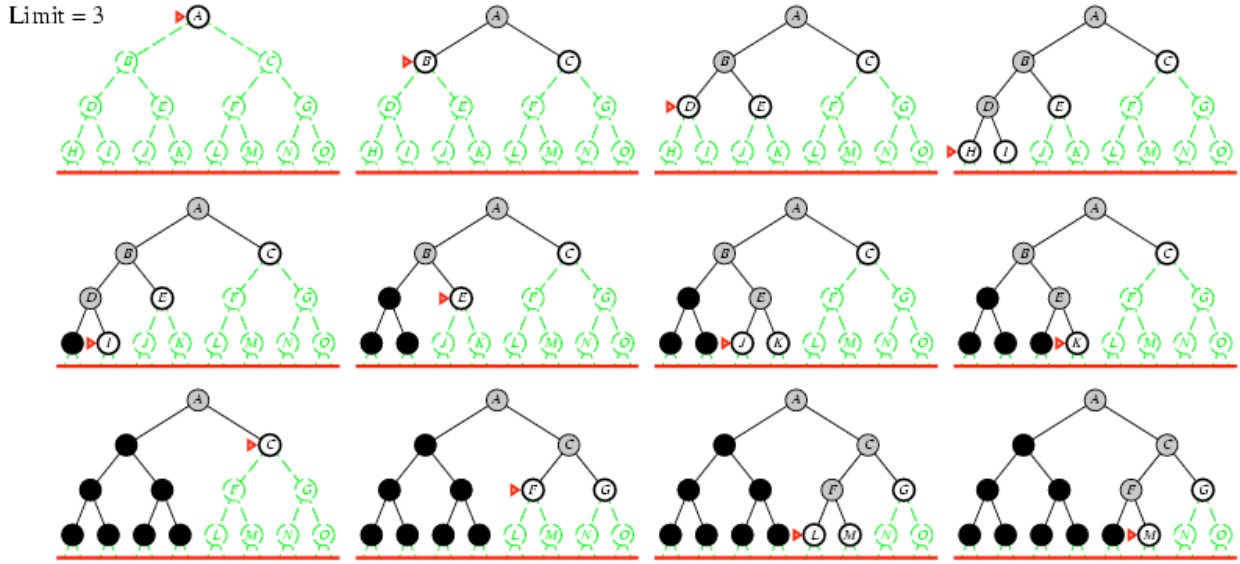
Iterative deepening search *L*=1



Iterative deepening search *L*=2



Iterative deepening search *L*=3



Algorithm

1. Set SEARCH-DEPTH=1.
2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution path is found, then return it.
3. Otherwise, increment SEARCH-DEPTH by 1 and go to step2.

Disadvantage

Iterative deepening looks inefficient because so many states are expanded multiple times. In practice this is not that bad, because by far most of the nodes are at the bottom level.

- For a branching factor b of 2, this might double the search time.
- For a branching factor b of 10, this might add 10% to the search time.

Advantage

- Avoids the problem of choosing cutoffs without sacrificing efficiency.
- DFID is the optimal algorithm for uniformed search.

Performance Analysis

Completeness - Yes

Time Complexity- $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space Complexity - $O(bd)$

Optimality - Yes, if step cost = 1 or increasing function of depth.

ITERATIVE DEEPENING A*

1. Set THRESHOLD=the heuristic evaluation of the start state.
2. Conduct a depth-first search, pruning any branch when its total cost function (g+h) exceeds a THRESHOLD. If a solution path is found during the search, return it.
3. Otherwise, Increment THRESHOLD by the minimum amount. It was exceeded during the previous step, and the go to step 2.

Iterative Deepening A* is guaranteed to find optimal solution, provided that h is a n admissible heuristic. Because of its depth-first search technique, IDA* is very efficient with respect to space. IDA* is the first heuristic search algorithm to find optimal paths within reasonable time and space constraints.

KNOWLEDGE REPRESENTATION

In order to solve complex problems in artificial intelligence, we need large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems. Consider dealing with two different kinds of entities.

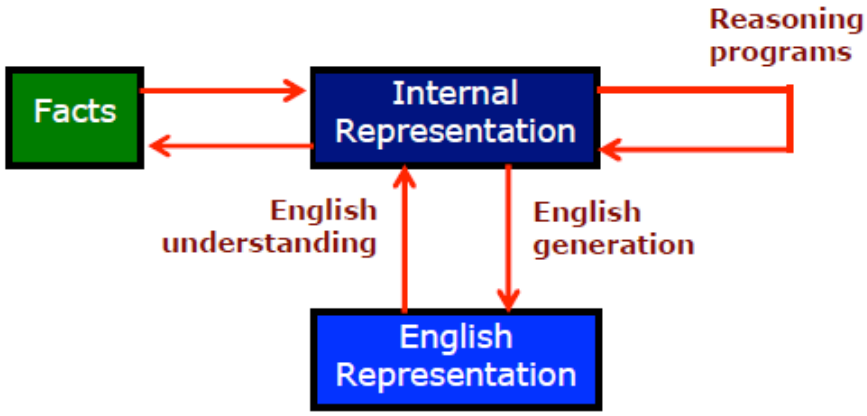
- Facts- truths in some relevant world. These are the things we want to represent.
- Representation- These are the things we will actually be able to manipulate.

One way to think of structuring these entities is as two levels.

- Knowledge level, at which the facts are represented.
- Symbol level, at which representations of objects at the knowledge level are defined in terms of symbols

There are two way mapping exists between facts and representation.

- Forward representation mapping – maps from facts to representation
- Backward representation mapping – maps from representation to facts



Mapping between facts and representation

English representation of these facts in order to facilitate getting information in to or out of the system.

Consider the English sentence

Spot is a dog

The fact represented by that English sentence can be represented in logic as

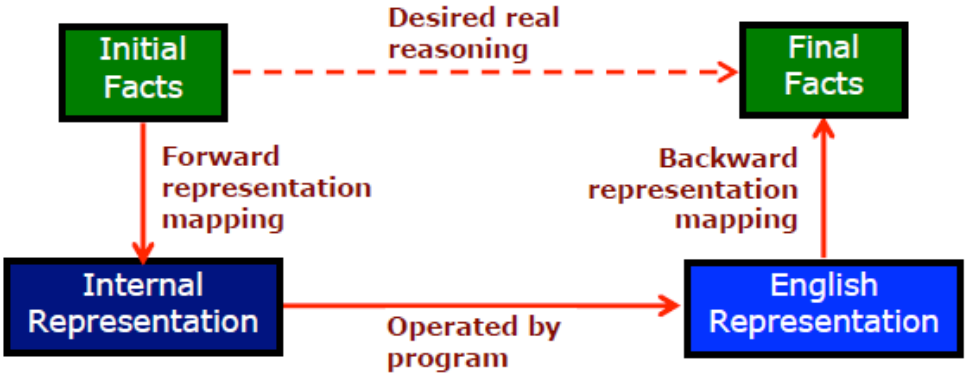
Dog(Spot)

Suppose we have logical representation of the fact that “All dogs have tail”

∀x:dog(x)→hastail(x)

Using appropriate backward mapping function, we could generate English sentence

Spot has a tail



Representation of facts (Expanded view)

APPROACHES TO KNOWLEDGE REPRESENTATION

A good system for the representation of structured knowledge in a particular domain should possess 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 to derive new structures corresponding to the new knowledge inferred from old..

(iii) **Inferential Efficiency**:- The ability to incorporate additional information into the knowledge structure that can be used to focus the attention of the inference mechanisms in the most promising directions.

(iv) **Acquisitional Efficiency** :- The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

KNOWLEDGE REPRESENTATION SCHEMES

There are four types of knowledge representation

- Relational Knowledge
- Inheritable Knowledge
- Inferential Knowledge
- Procedural Knowledge

(i) Relational Knowledge

Relational knowledge provides a frame work to compare two objects based on equivalent attributes. This knowledge associates element of one domain with another domain. Relational knowledge is made up of objects consisting of attributes as their corresponding associated values. The results of this knowledge type is mapping of elements among different domains.

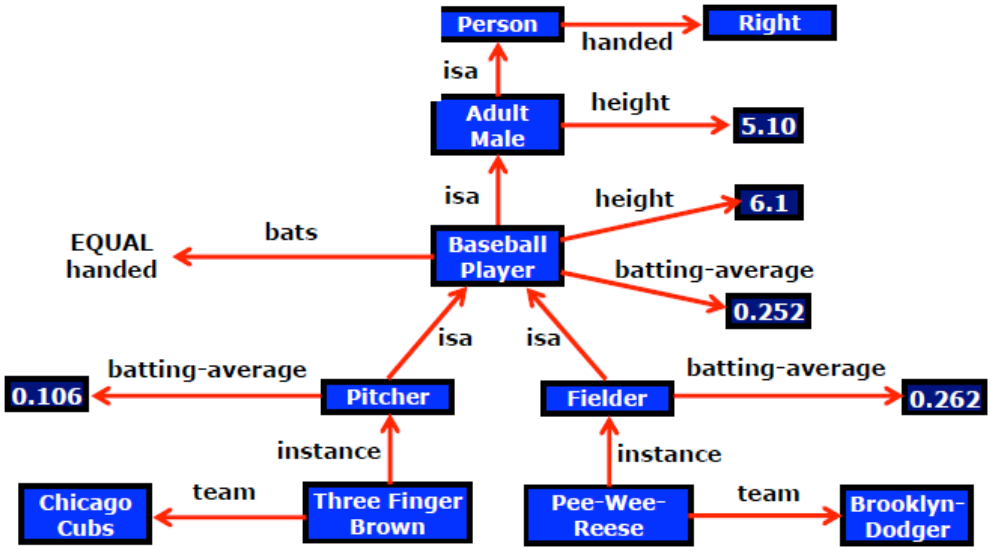
The table below shows a simple way to store facts. The facts about the set of objects are put systematically in columns.

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 fact that it is not [possible to answer a simple question such as “Who is the heaviest player?”. But, if the procedure for finding the heaviest player is provided, then these facts will be enable that procedure to compute an answer. This representation provides only little opportunity for inference.

(ii) Inheritable Knowledge

Inheritable knowledge is obtained from associated objects. It describes a structure in which new objects are created which may inherit all or subset of attributes from existing objects.



Inheritable knowledge representation

The directed arrow represent attributes originates at object being described and terminates at object or its value. The box nodes represent objects and values of the attributes.

Viewing node as a frame: Baseball Player

isa :Adult-Male
bats : Equal to handed
height : 6.1
bating average:0.252

Algorithm: Property of Inheritance

The steps for retrieving a value v for an attribute A of an instance object O

i) Find object O in the knowledge base.

ii) If there is a value for the attribute A, then report that value.

iii) Otherwise, see if there is a value for the attribute instance. If not then fail.

iv) Otherwise, move to the node corresponding to that value and look for a value for the attribute A. If one is found, report it.

v) Otherwise, do until there is no value for the isa attribute or until an answer is found.

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

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

Team(Pee-Wee-Reese)=Brooklyn Dodger
Batting average(Three finger Brown)=0.106
Height(Pee-Wee-Reese)=6.1
Bats(Three finger Brown)=right

iii) Inferential Knowledge

Inferential knowledge is inferred from objects through relations among objects. 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 with in linguistic is called semantics.

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

Example

Given a set of values and relations, one may infer other values or relations. Predicate logic is used to infer from a set of attributes. Inference from a predicate logic uses a set of logical operations to relate individual data.

i) Wonder is a name of a dog.

Dog(Wonder)

ii) All dogs belong to the class of animals.

$\forall x: dog(x) \rightarrow animal(x)$

iii) All animals either live on land or in water.

$\forall x: animal(x) \rightarrow live(x, land) \vee live(x, water)$

From these three statements we infer that

Wonder lives either on land or on water

iv) Procedural Knowledge

Procedural Knowledge specifies what to do when. The procedural knowledge is represented in program in many ways. The machine uses the knowledge when it executes the code to perform a task.

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

ISSUES IN KNOWLEDGE REPRESENTATION

The fundamental goal of knowledge representation is to facilitate inferencing from knowledge. The issues that arise while using knowledge representation techniques are many. Some of these are explained below.

- Any attributes of objects so basic that they appear in almost every problem domain?. If there are we need to make sure that they are handled appropriately in each of the mechanisms we propose.
- Any important relationship that exists among object attributes?
- At what level of detail should the knowledge be represented?
- How sets of objects be represented?
- Given a large amount of knowledge stored in a database, how can relevant parts be accessed?

i) Important Attributes

There are two attributes that are of very general significance

- Isa
- Instance

These attributes are important because they support property inheritance. They represent class membership and class inclusion and the class inclusion is transitive

ii) Relationships among Attributes

The attributes that are used to describe objects are themselves entities. The relationship between the Each entity have four properties independent of the specific knowledge they encode. They are:

(a) Inverses

(b) Existence in an isa hierarchy

(c) Techniques for reasoning about values

(d) Single-valued attributes

(a) Inverses:

Entities in the world are related to each other in many different ways. Relationships are attributes such as **isa, instance and team.** There are two good ways to represent the relationship.

The first is to represent both relationships in a single representation

team(Pee-Wee-Reese, Brooklyn-Dodgers) can equally easily be represent as a statement about Pee-Wee-Reese or about the Brooklyn-Dodgers.

The second approach is to use attributes that focus on a single entity but to use them in pairs, one the inverse of the other. In this approach, represent the team information with two attributes:

team = Brooklyn-Dodgers

team-members = Pee-Wee-Reese;…

This is the approach that is taken in semantic net and frame-based systems. Each time a value is added to one attribute then the corresponding value is added to the inverse.

(b) Existence in an isa hierarchy

The attribute height is the specialization of general attribute physical size which is in turn a specialization of physical attribute. The generalization specialization attributes are important because they support inheritance.

(c) Techniques for reasoning about values

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

- Information about the type of value. Eg height must be a unit of length.
- Constraints on the value. Eg age of a person cannot be greater than the age of persons parents.
- Rules for computing the value when it is needed. Eg bats attribute. These rules are called backward rules. Such rules are also called if-needed rules.
- Rules that describe actions that should be taken if a value ever becomes known. These rules are called forward rules or sometimes if-added rules.

(d) Single-valued attributes

This is about specific attribute that is guaranteed to take unique value. Eg. A baseball player can at a time have only a single height and be a member of only one team.

(iii) Choosing the granularity of representations

It deals with what level should the knowledge be represented. The primitives are

- 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, Consider the following fact

John spotted Smith

This could be represented as

Spotted(John, Smith)

Or

Spotted(agent(John), object(smith))

Such representation would make it easy to answer question such as

Who spotted Smith?

Suppose we want to know

Did John see smith?

Given only one fact, but we cannot discover the answer. So we can add another fact

Spotted(x, y)→saw(x, y)

We can now infer the answer to the question

(iv)Representing set of objects

Certain properties of objects that is true as member of a set but not as individual.
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 to attach assertion to the sets representing people, sheep and English. The reason to represent sets of object 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 explicitly with every elements of the set.This is done in different ways:

- In logical representation through the use of universal quantifier,
- In hierarchical structure where node represents 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.
- Asserting some property that holds for individual elements of the set, means, anything that is an elephant is large.

There are three ways in which sets may be represented

- Names
- Extensional definition
- Intentional definition

(v) Finding the right structures as needed

To access the right structure for describing a particular situation, it is necessary to solve all the following problems,

- How to perform an initial selection of the most appropriate structure
- How to fill in appropriate details from the current situation
- 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 structure is appropriate
- When to create and remember a new structure
    - Selecting an Initial structure
        - Index the structure
        - Pointer to all the structures
        - Locate one major due in the problem description
    - Revising the choice when necessary

KNOWLEDGE REPRESENTATION USING OTHER LOGIC
PROPOSITIONAL LOGIC

A *proposition* is a declarative statement that's either TRUE or FALSE (but not both).

Propositions	Not Propositions
3 + 2 = 32	Bring me coffee!
CS173 is Bryan's favorite class.	CS173 is her favorite class.
Every cow has 4 legs.	3 + 2
There is other life in the universe.	Do you like Cake?

- **The symbols of the language:**
    - Propositional symbols (Prop): A, B, C,…
    - Connectives:
        - ∧ and
        - ∨ or
        - ¬ not

- → implies
- ↔ equivalent to
- ⊗ xor (different than)
- ⊥, > False, True

- Parenthesis :(,).

Propositional Logic Semantics

Negation

- Truth tables define the semantics (=meaning) of the operators

Suppose *p* is a proposition.

The *negation* of *p* is written ¬*p* and has meaning:
"It is not the case that *p*."

- Ex. TFCS is NOT Bryan's favorite class.

p	¬*p*
T	F
F	T

Conjunction

Conjunction corresponds to English "and."

- p ∧ q is true exactly when p and q are both true.
- Ex. Amy is curious AND clever.

p	q	*p* ∧ *q*
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction

Disjunction corresponds to English "or."

- p ∨ q is when p or q (or both) are true.
- Ex. Michael is brave OR nuts.

p	q	*p* ∨ *q*
T	T	T
T	F	T
F	T	T
F	F	F

Implication

Implication: p → q corresponds to English "if p then q," or "p implies q."

- If it is raining then it is cloudy.
- If there are 200 people in the room, then I am the Easter Bunny.

- If p then 2+2=4.

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Propositional Logic: Truth Tables

Truth table for different connectives for Negation, Disjunction (AND), Conjunction (OR), Condition, Bicondition, NAND, NOR, XOR

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$	$p \uparrow q$	$p \downarrow q$	$p \oplus q$
T	T	F	T	T	T	T	F	F	F
T	F	F	F	T	F	F	T	F	T
F	T	T	F	T	T	F	T	F	T
F	F	T	F	F	T	T	T	T	F

Propositional Logic - special definitions

Contrapositives: $p \rightarrow q$ and $\neg q \rightarrow \neg p$

- Ex. "If it is noon, then I am hungry."

"If I am not hungry, then it is not noon."

Converses: $p \rightarrow q$ and $q \rightarrow p$

- Ex. "If it is noon, then I am hungry."

"If I am hungry, then it is noon."

Inverses: $p \rightarrow q$ and $\neg p \rightarrow \neg q$

- Ex. "If it is noon, then I am hungry."

"If it is not noon, then I am not hungry."

Propositional Logic: Logical Equivalences

- ***Identity***

$p \wedge T \equiv p$

$p \vee F \equiv p$

- ***Domination***

$p \vee T \equiv T$

$p \wedge F \equiv F$

- ***Idempotence***

$p \vee p \equiv p$

$p \wedge p \equiv p$

- ***Double negation***

$\neg \neg p \equiv p$

- ***Commutativity:***

$p \vee q \equiv q \vee p$

$$p \wedge q \equiv q \wedge p$$

- ***Associativity:***

$$(p \vee q) \vee r \equiv p \vee (q \vee r)$$
$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$

- ***Distributive*:**

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$
$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

- ***De Morgan's*:**

$$\neg(p \wedge q) \equiv \neg p \vee \neg q \qquad \textit{(De Morgan's I)}$$
$$\neg(p \vee q) \equiv \neg p \wedge \neg q \qquad \textit{(De Morgan's II)}$$

- ***Excluded Middle:***

$$p \vee \neg p \equiv \mathrm{T}$$

- ***Uniqueness*:**

$$p \wedge \neg p \equiv \mathrm{F}$$

- **A useful Logical Equivalence (LE) involving →:**

$$p \rightarrow q \equiv \neg p \vee q$$

Clause- a special form

Literal

A single proposition and its negation. eg., p, $\neg p$

Clause

A clause is a disjunction of literals.

Eg., P∨Q∨¬R

Converting a compound proposition to the clausal form

Consider the sentence

¬(A→B) ∨(C→A)

1. Eliminate implication sign

¬(¬A∨B) ∨(¬C∨A)

2. Eliminate the double negation and reduce scope of "not sign" (De-Morgans Law)

(A∧ ¬B) ∨(¬C∨A)

3. Convert to conjunctive normal form by using distributive and associative laws

(A∨¬C ∨A) ∧ (¬B∨¬C∨A)

(A∨¬C) ∧ (¬B∨¬C∨A)

4. Get the set of clauses

(A∨¬C)

(¬B∨¬C∨A)

RESOLUTION

Resolution is a procedure used in proving that arguments which are expressible in predicate logic are correct. Resolution is a procedure that produces proofs by refutation or contradiction. Resolution leads to refute a theorem-proving technique for sentences in propositional logic and first order logic.

- Resolution is a rule of inference
- Resolution is a computerized theorem prover

Algorithm: Propositional Resolution

Let F be a set of axioms and P the theorem to prove.

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 pair 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.

Example 1

1. If a triangle is equilateral then it is isosceles.
2. If a triangle is isosceles the two sides AB and AC are equal.
3. If AB and AC are equal then angle B and angle C are equal.
4. ABC is an Equilateral triangle.

Prove "Angle B is equal to angle C"

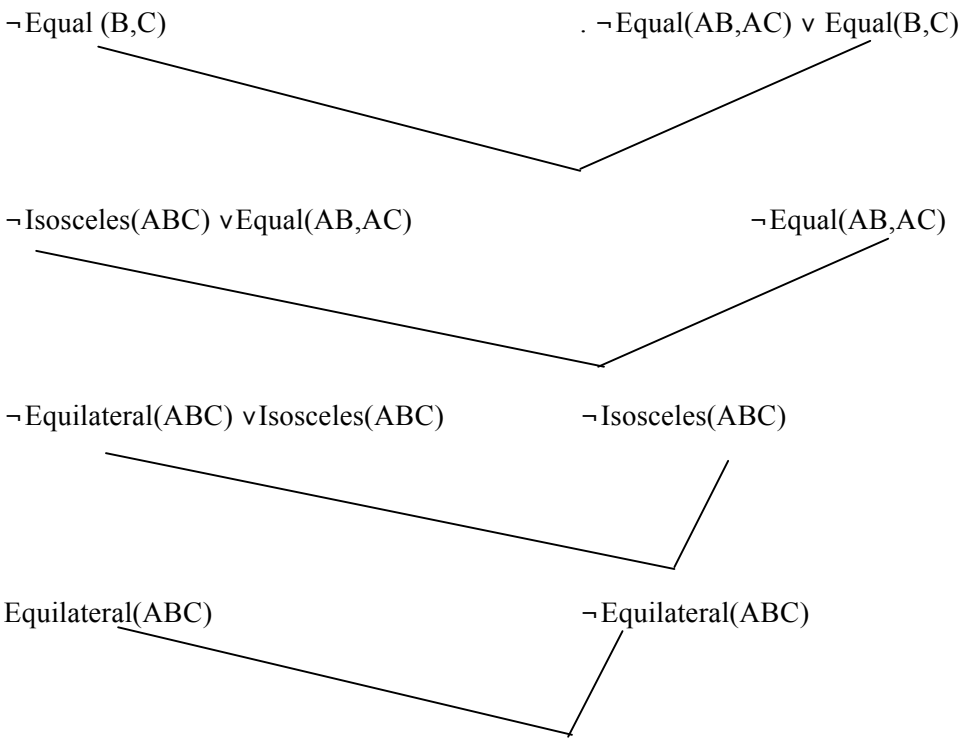
Propositional Logic

4. Equilateral(ABC)
1. Equilateral(ABC)→Isosceles(ABC)
2. Isosceles(ABC)→Equal(AB,AC)
3. Equal(AB,AC)→ Equal(B,C)

Convert to clausal form

1. ¬Equilateral(ABC) ∨Isosceles(ABC)
2. ¬Isosceles(ABC) ∨Equal(AB,AC)
3. ¬Equal(AB,AC) ∨ Equal(B,C)
4. Equilateral(ABC)

To prove " Equal (B,C)". **Let us disprove "Not equal B and C ¬Equal (B,C)"**



None

Thus we proved that angle B is equal to angle C

Example 2

1. Mammals drink milk.
2. Man is mortal.
3. Man is mammal.
4. Tom is a man.

Prove “Tom drinks milk” and Prove “Tom is mortal”

Propositional Logic

1. Mammal(Tom)→drink(Tom, Milk)
2. Man(Tom)→Mortal(Tom)
3. Man(Tom)→Mammal(Tom)
4. Man(Tom)

Convert to clausal form

1. ¬Mammal(Tom) ∨drink(Tom, Milk)
2. ¬Man(Tom) ∨Mortal(Tom)
3. ¬Man(Tom) ∨Mammal(Tom)
4. Man(Tom)

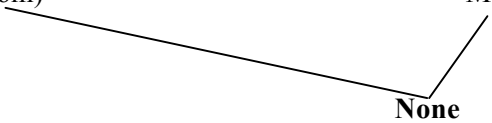
To prove ” Tom drinks milk”

Let us disprove “Not Tom drinks milk”➔ “¬ Drink (Tom, Milk)”

¬Mammal(Tom) ∨drink(Tom, Milk) . ¬Drink (Tom, Milk)

¬Man(Tom) ∨Mammal(Tom) ¬Mammal(Tom)

Man(Tom) ¬Man(Tom)



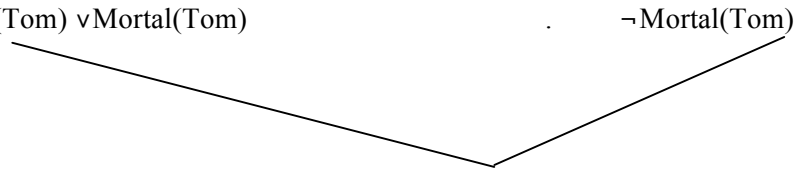
None

Thus we proved “Tom drinks Milk”

ii) To prove ” Tom is Mortal”➔Mortal(Tom)

Let us disprove “Not Tom is Mortal” ➔¬Mortal(Tom)

¬Man(Tom) ∨Mortal(Tom) . ¬Mortal(Tom)



Man(Tom) ¬Man(Tom)

None

Thus we proved “Tom is Mortal”

Disadvantages of propositional logic

Propositional logic only deals with finite number of propositions.

PREDICATE LOGIC OR FIRST ORDER LOGIC

FOL or predicate logic is a generalization of propositional logic that allows us to express and infer arguments in infinite models like

- All men are mortal
- Some birds cannot fly

Predicates are like functions except that their return type is true or false.
A predicate with no variable is a proposition.

Syntax of First-Order Logic

- Constants — KingJohn, 2, …
- Predicates/Relation — Brother, >, …
- Functions — Sqrt, LeftArmOf, …
- Variables — x, y, a, b, …
- Connectives — ∧ ∨ ¬ ⇒ ⇔
- Equality — =
- Quantifiers — $ "

Components of First-Order Logic

- **Term**
  - Constant, e.g. Red
  - Function of constant, e.g. Color(Block1)
- **Atomic Sentence**
  - Predicate relating objects (no variable)
    - Brother (John, Richard)
    - Married (Mother(John), Father(John))
- **Complex Sentences**
  - Atomic sentences + logical connectives
    - Brother (John, Richard) ∧¬Brother (John, Father(John))
- **Quantifiers**
  - Each quantifier defines a variable for the duration of the following expression, and indicates the truth of the expression…
- **Universal quantifier "for all" ∀**
  - The expression is true for every possible value of the variable
- **Existential quantifier "there exists" ∃**
  - The expression is true for at least one value of the variable

Examples

1. Some dogs bark.

∃x dog(x) ∧ bark(x)

2. All dogs have four legs.

∀x(dog(x)→have_four_legs(x))

(or)

∀x(dog(x)→legs(x,4))

3. All barking dogs are irritating

∀x(dog(x) ∧barking(x)→irritating(x))

4. No dogs purr.

¬∃x (dog(x) ∧ purr(x))

5. Fathers are male parents with children.

$\forall x(father(x) \rightarrow male(x) \wedge haschildren(x))$

6. Students are people who are enrolled in courses.

$\forall x(student(x) \rightarrow enrolled(x, courses))$

ALGORITHM: RESOLUTION-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

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

   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exceptions: If there is one pair of literals T1 and ¬T2 such that one of the parent clause contains T2 and the other contains t1 and if t1 and t2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 as complementary literals. Use the substitution produced by the unificationto create the resolvent. If there is more than pair of complementary literals, only one 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.

There exists a procedure for making the choice that can speed up the process considerably

- Only resolve pairs of clauses that contain complementary literals
- Eliminate certain clauses as soon as they are generated so that they cannot participate in later resolutions.
- Whenever possible resolve either with one of the clauses that is part of the statement we are trying to refute or with clause generated by a resolution with such clause. This is called set of support strategy
- Whenever possible resolve with clauses that have a single literal. Such resolution generate new clauses with fewer literals. This is called unit preference strategy.

Problem 1

1. All people who are graduating are happy.

2. All happy people smile.

3. Someone is graduating.

4. Conclusion: Is someone smiling?

Solution:

Convert in to predicate Logic

1. $\forall x[graduating(x) \rightarrow happy(x)$
2. $\forall x(happy(x) \rightarrow smile(x)$
3. $\exists x\ graduating(x)$
4. $\exists x\ smile(x)$

Convert to clausal form

(i) Eliminate the → sign

1. $\forall x \neg graduating(x) \vee happy(x)$
2. $\forall x \neg happy(x) \vee smile(x)$
3. $\exists x\ graduating(x)$
4. $\neg \exists x\ smile(x)$

(ii) Reduce the scope of negation

1. ∀x¬graduating(x) ∨happy(x)
2. ∀x¬happy(x) ∨smile(x)
3. ∃x graduating(x)
4. ∀x ¬ smile(x)

(iii) Standardize variables apart

1. ∀x¬graduating(x) ∨happy(x)
2. ∀y¬happy(y) ∨smile(y)
3. ∃x graduating(z)
4. ∀w ¬ smile(w)

(iv) Move all quantifiers to the left

1. ∀x¬graduating(x) ∨happy(x)
2. ∀y¬happy(y) ∨smile(y)
3. ∃x graduating(z)
4. ∀w ¬ smile(w)

(v) Eliminate ∃

1. ∀x¬graduating(x) ∨happy(x)
2. ∀x¬happy(y) ∨smile(y)
3. graduating(name1)
4. ∀w ¬ smile(w)

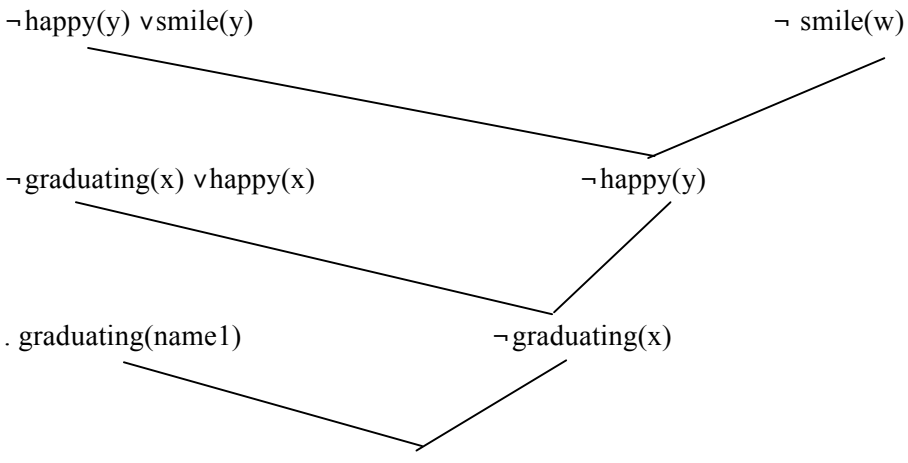
(vi) Eliminate ∀

1. ¬graduating(x) ∨happy(x)
2. ¬happy(y) ∨smile(y)
3. graduating(name1)
4. ¬ smile(w)

(vii) Convert to conjunct of disjuncts form.

(viii) Make each conjunct a separate clause.

(ix) Standardize variables apart again.



None

Thus, we proved someone is smiling.

Problem 2

Explain the unification algorithm used for reasoning under predicate logic with an example. Consider the following facts

a. Team India

b. Team Australia

c. Final match between India and Australia

d. India scored 350 runs, Australia scored 350 runs, India lost 5 wickets, Australia lost 7 wickets.

e. The team which scored the maximum runs wins.

f. If the scores are same the team which lost minimum wickets wins the match.

Represent the facts in predicate, convert to clause form and prove by resolution "India wins the match".

Solution:

Convert in to predicate Logic

(a) team(India)

(b) team(Australia)

(c) team(India) ∧ team(Australia)→final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ∃x team(x) ∧ wins(x)→score(x, max_runs))

(f) ∃xy score(x,equal(y)) ∧ wicket(x, min) ∧final_match(x,y)→win(x)

Convert to clausal form

(i) Eliminate the → sign

(a) team(India)

(b) team(Australia)

(c) ¬ (team(India) ∧ team(Australia)) ∨ final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ∃x ¬ (team(x) ∧ wins(x)) ∨score(x, max_runs))

(f) ∃xy ¬ (score(x,equal(y)) ∧ wicket(x, min) ∧final_match(x,y)) ∨win(x)

(ii) Reduce the scope of negation

(a) team(India)

(b) team(Australia)

(c) ¬ team(India) ∨ ¬team(Australia) ∨ final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ∃x ¬ team(x) ∨ ¬wins(x) ∨score(x, max_runs))

(f) ∃xy ¬ score(x,equal(y)) ∨¬wicket(x, min_wicket) ∨¬final_match(x,y)) ∨win(x)

(iii) Standardize variables apart

(iv) Move all quantifiers to the left

(v) Eliminate ∃

(a) team(India)

(b) team(Australia)

(c) ¬ team(India) ∨ ¬team(Australia) ∨ final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ¬ team(x) ∨ ¬wins(x) ∨score(x, max_runs))

(f) ¬ score(x,equal(y)) ∨¬wicket(x, min_wicket) ∨¬final_match(x,y)) ∨win(x)

(vi) Eliminate ∀

(vii) Convert to conjunct of disjuncts form.

(viii) Make each conjunct a separate clause.

(a) team(India)

(b) team(Australia)

(c) ¬ team(India) ∨ ¬team(Australia) ∨ final_match(India, Australia)

(d) score(India,350)

score(Australia,350)

wicket(India,5)

wicket(Australia,7)

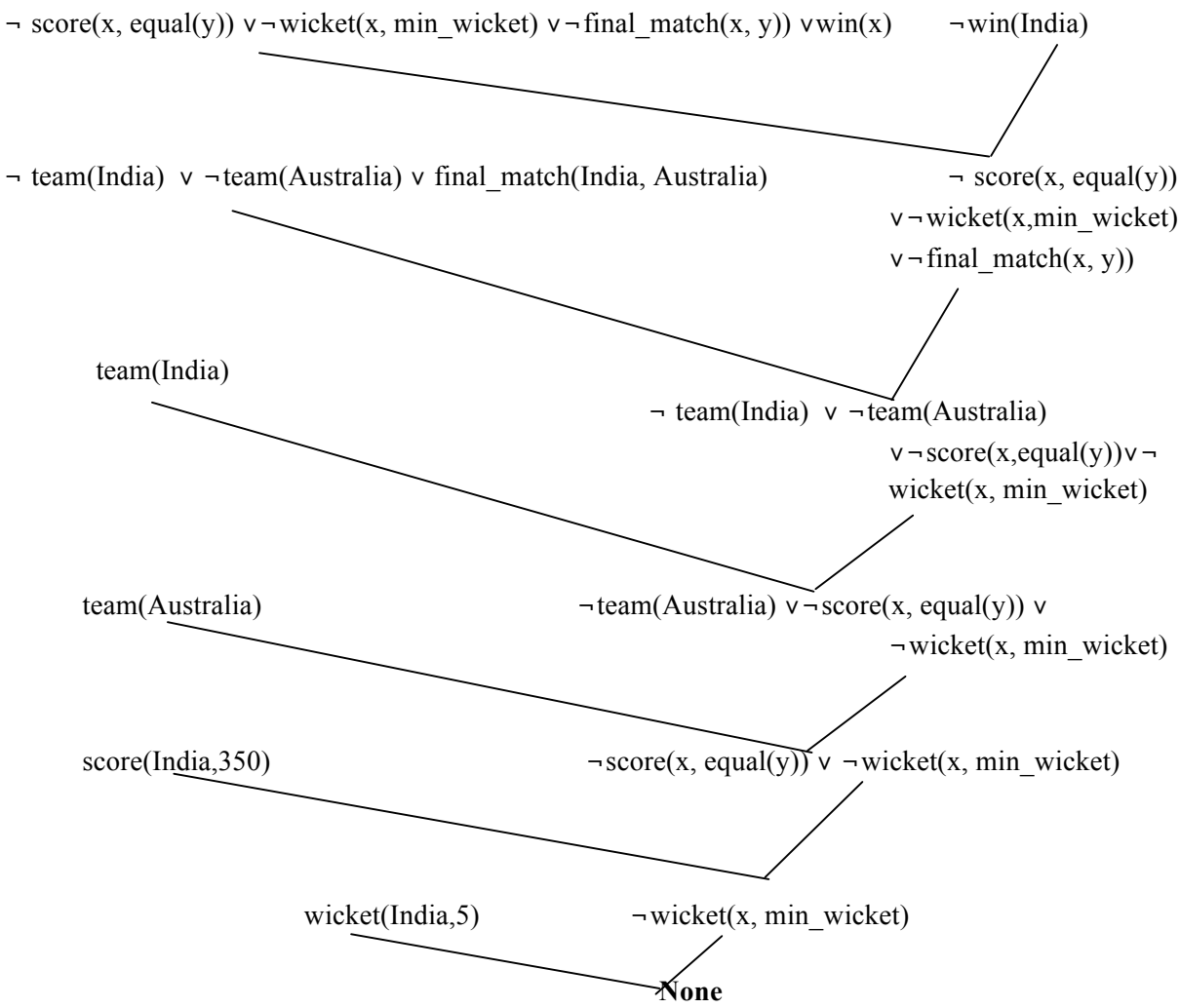
(e) ¬ team(x) ∨ ¬wins(x) ∨score(x, max_runs))

(f) ¬ score(x, equal(y)) ∨¬wicket(x, min_wicket) ∨¬final_match(x, y)) ∨win(x)

(ix) Standardize variables apart again.

To prove: win(India)

Disprove: ¬win(India)



Thus, proved India wins match.

Problem 3

Consider the following facts and represent them in predicate form:

F1. There are 500 employees in ABC company.

F2. Employees earning more than Rs. 5000 pay tax.

F3. John is a manager in ABC company.

F4. Manager earns Rs. 10,000.

Convert the facts in predicate form to clauses and then prove by resolution: "John pays tax".

Solution:

Convert in to predicate Logic

1. company(ABC) ∧employee(500,ABC)
2. ∃ x company(ABC) ∧employee(x, ABC) ∧earns(x,5000)→pays(x, tax)
3. manager(John, ABC)
4. ∃ x manager(x, ABC)→earns(x,10000)

Convert to clausal form

(i) Eliminate the → sign

1. company(ABC) ∧employee(500,ABC)
2. ∃ x¬ (company(ABC) ∧employee(x, ABC) ∧earns(x,5000)) ∨ pays(x, tax)
3. manager(John, ABC)
4. ∃ x ¬manager(x, ABC) ∨ earns(x, 10000)

(ii) Reduce the scope of negation

1. company(ABC) ∧employee(500,ABC)
2. ∃ x¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ∃ x ¬manager(x, ABC) ∨ earns(x, 10000)

(iii) Standardize variables apart

1. company(ABC) ∧employee(500,ABC)
2. ∃ x¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ∃ y ¬manager(x, ABC) ∨ earns(y, 10000)

(iv) Move all quantifiers to the left

(v) Eliminate ∃

1. company(ABC) ∧employee(500,ABC)
2. ¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ¬manager(x, ABC) ∨ earns(y, 10000)

(vi) Eliminate ∀

(vii) Convert to conjunct of disjuncts form.

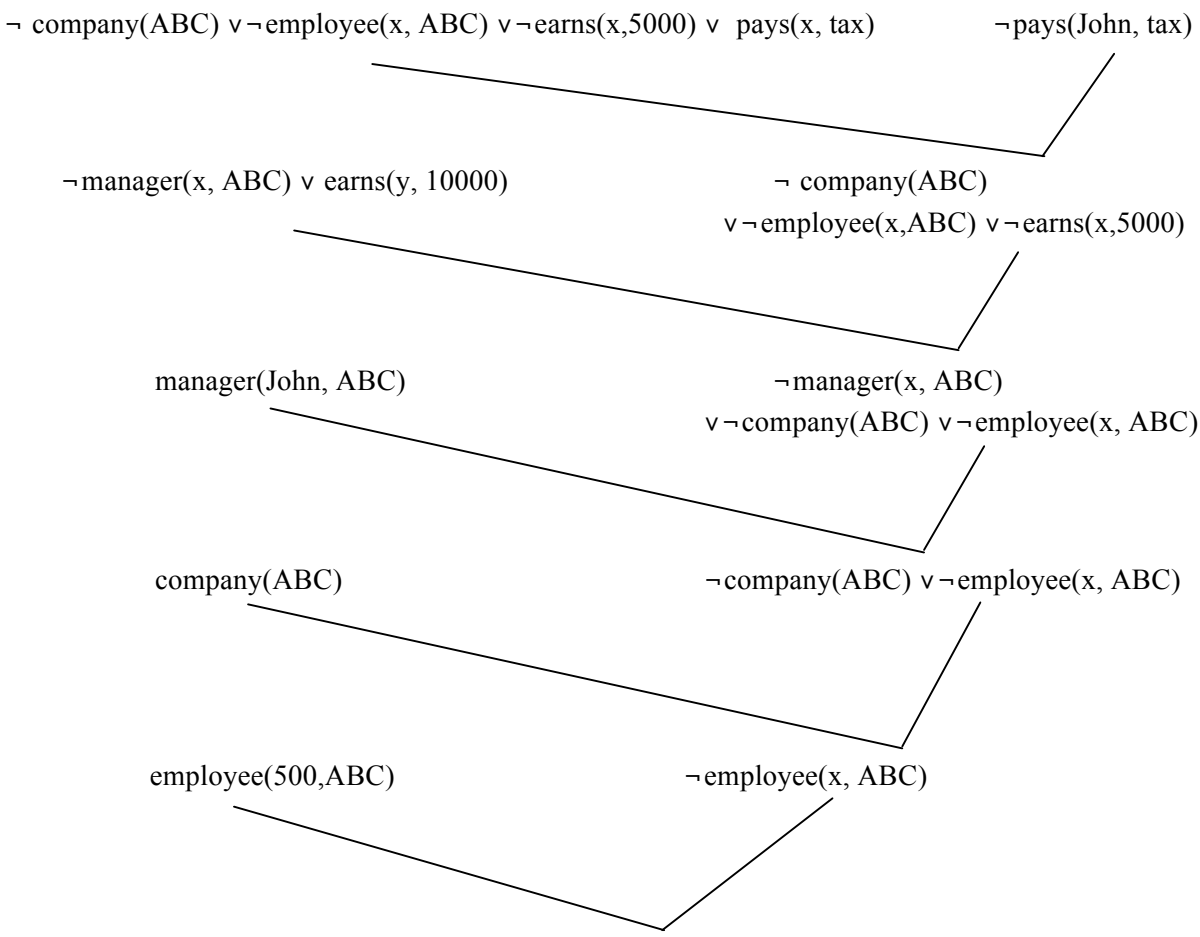
(viii) Make each conjunct a separate clause.

1. (a)company(ABC)
   (b)employee(500,ABC)
2. ¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ¬manager(x, ABC) ∨ earns(y, 10000)

(ix) Standardize variables apart again.

Prove :pays(John, tax)

Disprove: ¬pays(John, tax)



None

Thus, proved john pays tax.

Problem 4

If a perfect square is divisible by a prime p then it is also divisible by square of p.
Every perfect square is divisible by some prime.
36 is a perfect square.

Convert in to predicate Logic

1. ∀xy perfect_sq(x) ∧prime(y) ∧divides(x,y)→divides(x,square(y))
2. ∀x∃y perfect_sq(x) ∧prime(y) ∧divides(x,y)
3. perfect_sq(36)

Problem 5

1. Marcus was a man

man(Marcus)

2. Marcus was a Pompeian

Pompeian(Marcus)

3. All Pompeians were Romans

∀x (Pompeian(x)→ Roman(x))

4. Caesar was a ruler

ruler(Caesar)

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

∀x (Roman(x) →loyalto(x,Caesar) ∨ hate(x,Caesar))

6. Everyone is loyal to someone

$\forall x \ \exists y$ (person(x) → person(y) ∧ loyalto(x,y))

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

$\forall x \ \forall y$ (person(x) ∧ ruler(y) ∧ tryassassinate(x,y)→ ¬loyalto(x,y))

8. Marcus tried to assassinate Caesar

tryassassinate(Marcus, Caesar)

9. All men are persons

$\forall x$ (man(x) → person(x))

UNIFICATION ALGORITHM

When attempting to match 2 literals, all substitutions must be made to the entire literal. There may be many substitutions that unify 2 literals, the most general unifier is always desired

1. Initial predicate symbols must match.
2. For each pair of predicate arguments:

– different constants cannot match.

– a variable may be replaced by a constant.

– a variable may be replaced by another variable.

– a variable may be replaced by a function as long as the function does not contain an instance of the variable

Algorithm :Unify(L1,L2)

1. 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 the return {FAIL}, else return (L2/L1).
   (c) Else if L2 is a variable, then if L2 occurs in L1 the return {FAIL}, else return (L1/L2).
   (d) Else return {FAIL}
2. If the initial predicate symbols in L1 and L2 are not identical, the return {FAIL}.
3. If L1 and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL.
5. For i←1 to number of arguments in L1:
   (a) Call Unify with the i^{th} argument of L1 and the i^{th} 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.

Example

P(x) and P(y) :substitution = (x/y)

• P(x,x) and P(y,z) : (z/y)(y/x)

• P(x,f(y)) and P(Joe,z) : (Joe/x, f(y)/z)

• P(f(x)) and P(x) : can't do it!

• P(x) Ú Q(Jane) and P(Bill) ∨ Q(y): (Bill/x, Jane/y)

DIFFERENTIATE PREDICATE AND PROPOSITIONAL LOGIC.

Ans:

Sl.No	Predicate logic	Propositional logic
1.	Predicate logic is a generalization of propositional logic that allows us to express and infer arguments in infinite models.	A proposition is a declarative statement that's either TRUE or FALSE (but not both).
2.	Predicate logic (also called predicate calculus and first-order logic) is an extension of propositional logic to formulas involving terms and predicates. The full predicate logic is undecidable	Propositional logic is an axiomatization of Boolean logic. Propositional logic is decidable, for example by the method of truth table
3.	Predicate logic have variables	Propositional logic has variables. Parameters are all constant
4.	A predicate is a logical statement that depends on one or more variables (not necessarily Boolean variables)	Propositional logic deals solely with propositions and logical connectives
5.	Predicate logic there are objects, properties, functions (relations) are involved	Proposition logic is represented in terms of Boolean variables and logical connectives
6.	In predicate logic, we symbolize subject and predicate separately. Logicians often use lowercase letters to symbolize subjects (or objects) and uppercase letter to symbolize predicates.	In propositional logic, we use letters to symbolize entire propositions. Propositions are statements of the form "x is y" where x is a subject and y is a predicate.
7.	Predicate logic uses quantifiers such as universal quantifier ("∀"), the existential quantifier ("∃")	Prepositional logic has no quantifiers.
8.	**Example** Everything is green" as "∀x Green(x)" or "Something is blue" as "∃x Blue(x)".	**Example** Everything is green" as "G(x)" or "Something is blue" as "B(x)".

STRUCTURED REPRESENTATIONS OF KNOWLEDGE

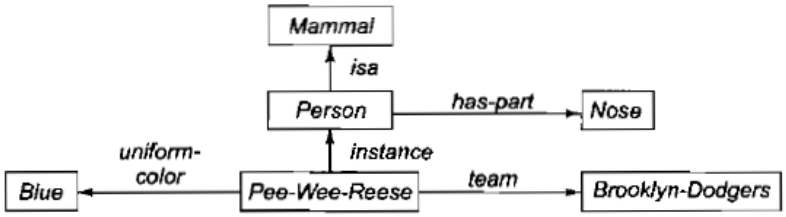
Modeling-based representations reflect the structure of the domain, and then reason based on the model.

- **Semantic Nets**
- **Frames**
- **Conceptual Dependency**
- **Scripts**
- **CYC**

(i) Semantic Networks

In semantic net, information is represented as a set of nodes connected to each other by a set of labelled arcs, which represent relationship among the nodes. In this network, inheritance is used to derive the additional relation.

Eg has_part(Pee-Wee-Reesse, nose)



Intersection search

To find the relationship among objects by spreading activation out from each of two nodes and seeing where the activation function met. This process is called intersection search.

Example:

What is the connection between the Brooklyn dodger and blue?

The answer is Pee-Wee-Reese

Representing non binary predicates

Semantic nets are a natural way to represent relationships that would appear as binary predicates.

instance(pee-Wee-Reese, Person)

team(Pee-Wee-Reese, Brooklyn-Dodger)

Unary predicates can also be represented as binary predicates

Unary Predicate: man(Marcus)

Binary Predicate: instance(Marcus, man)

(ii) Frame

A frame is a collection of attributes and associated values that describe some entity in the world. Sometimes a frame describes an entity in some absolute sense; sometimes it represents the entity from a particular point of view. A single frame taken alone is rarely useful. Instead build frame systems out of collections of frames that are connected to each other by a virtue of the fact that the value of an attribute of one frame may be another frame. Frame system can be used to encode knowledge and support reasoning. Each frame represents either a class or an instance.

Example of a frame

Pee-wee-Reese

instance:	:Fielder
height	:5.10
bats	:Right
batting-average	:0.309
team	:Brooklyn Dodger
uniform color	:Blue

ML-Baseball-Team

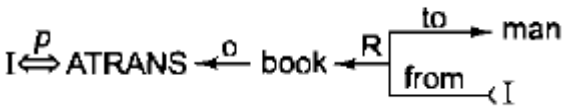
isa	:Team
cardinality	:26
team-size	:24

(iii) Conceptual Dependency

Conceptual dependency is a theory of how to represent the kind of knowledge about events that is usually contained in natural language sentence. The goal is to represent the knowledge in a way that

- Facilitates drawing inferences from the sentences
- Is independent of the language in which the sentence were originally stated

Conceptual dependency has been implemented in variety of programs that read and understand natural language text. Unlike semantic nets, which provide only a structure in to which nodes representing information at any level can be placed, conceptual dependency provides both a structure and a specific set of primitives, at a particular level of granularity, out of which representations of particular pieces of information can be constructed.



I gave the man a book

- Arrows indicate direction of dependency
- Double arrow indicate double two way link between actor and action
- P indicate past tense
- ATRANS is one of the primitive act used by the theory. It indicates transfer of possession.
- O indicates object case generation
- R indicate recipient case relation

(iv) Scripts

Script is a structure that describes a stereotyped sequence of events in a particular context. A script consists of a set of slots. Associated with each slot some information about what kind of values it may contain as well as a default value to be used if no other information is available. The important components of the scripts are

Entry conditions-Conditions that must be satisfied before the events described in a script can occur.
Result- Event happened after the script has occurred
Props- Objects involved in the event
Roles-People who are involved in the event
Track-The specific variation on a general more pattern
Scenes-The actual sequence of events that occur

Example: Restaurant
Track- Coffee Shop
Prop- Table, Menu, food, Money
Roles- S Customer
W Waiter
C cook
M Cashier
O Owner
Entry Condition
S is hungry
S has money

Results

S has less money
O has more money
S is not hungry

Scene-Eating

C ATRANS F to W
W WATRANS F to S
S INGEST F

(v) CYC

CYC is a very large knowledge base project aimed at capturing human common sense knowledge. The goal of CYC is to encode the large body of knowledge. Such a knowledge base could then be combined with specialized knowledge bases to produce systems.

CYC contains representations of events and objects. CYC is particularly concerned with the issue of scale, that is, what happens when we build knowledge bases that contain millions of objects.

Advantages of structured representation

- Retrieves the values for an attributes in a fast manner.
- Easy to describe the properties of relations.
- It is a form of object oriented programming