Game Playing

There were two reasons that games appeared to be a good domain in which to explore machine intelligence:

1. They provide a structured task in which it is very easy to measure success or failure.
2. They did not require large amount of knowledge. They were thought to be solvable by straight forward search from the starting state to a winning position.

Ex Chess

- The average branching factor is around 35.
- In an average game, each player might make 50 moves.
- One would have to examine 35^{100} positions.

Some kind of heuristic search procedure is necessary.

Generally Generate and test procedure is used, in which at one extreme the generator generates entire proposed solution, which the tester then evaluates.at the other extreme, the generator generates individual moves in the search space, each of which is then evaluated by the tetser and the most promising one is chosen.

To improve the effectiveness of a search-based problem –solving program 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.

Consider again the problem of playing chess. On the average, there are about 35 legal moves available at each turn. If a simple legal-move generator is used, then the test procedure will have to look at each of them. The test procedure must be fast and look at so many possibilities. So it cannot do a very accurate job.

On the other hand, instead of a legal move generator, a plausible move generator generates only some small amount of promising moves. As the legal moves increases, heuristics have to be applied to select only those that have some kind of promise. Thus by incorporating heuristic knowledge into both the generator and the tester the performance of the overall system can be improved.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached. The depth of the resulting tree and its branching factor are too great. It is possible to search a tree only ten or twenty moves deep called ply and to choose the best move, the resulting board positions must be compared to discover which is most advantageous. This is done using a static evaluation function which uses whatever information it has to evaluate individual board positions by estimating how likely they are to lead eventually to a win.

A good plausible-move generator and a good static evaluation function must both incorporate a great deal of knowledge about the particular game being played. But unless these functions are perfect, a search procedure is needed that makes it possible to look ahead as many moves as possible to see what may occur.

For a simple one-person game or puzzle, the A* algorithm can be used. But this procedure is inadequate for two-person games such as chess. There are several ways this can be done and the most commonly used method is the minimax procedure.

MIN-MAXSearch

Games have always been an important application area for heuristic algorithms. In playing games whose state space may be exhaustively delineated, the primary difficulty is in accounting for the actions of the opponent. This can be handled easily by assuming that the opponent uses the same knowledge of the state space as us and applies that knowledge in a

consistent effort to win the game. Minmax implements game search under referred to as MIN and MAX.

The min max search procedure is a depth first, depth limited search procedure. The idea is to start at the current position and use the plausible move generator to generate the set of possible successor positions. To decide one move, it explores the possibilities of winning by looking ahead to more than one step. This is called a ply. Thus in a two ply search, to decide the current move, game tree would be explored two levels farther.

Consider the below example

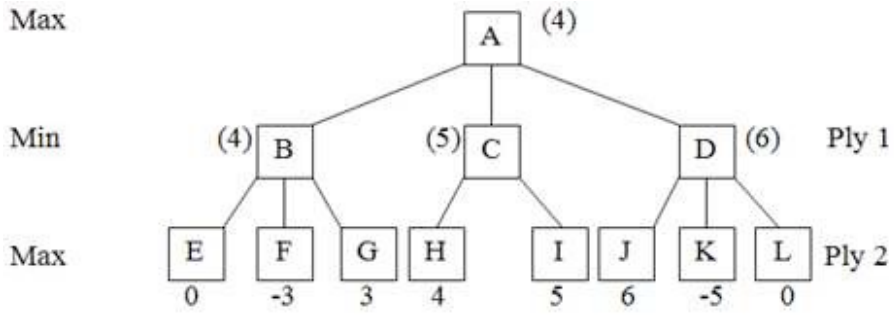


Figure Tree showing two ply search

In this tree, node A represents current state of any game and nodes B, C and D represent three possible valid moves from state A. similarly E, F, G represents possible moves from B, H, I from C and J, K, L, from D. to decide which move to be taken from A, the different possibilities are explored to two next steps. 0, -3, 3, 4, 5, 6, -5, 0 represent the utility values of respective move. They indicate goodness of a move. The utility value is back propagated to ancestor node, according to situation whether it is max ply or min ply. As it is a two player game, the utility value is alternatively maximized and minimized. Here asthe second player's move is maximizing, so maximum value of all children of one node will be back propagated to node. Thus, the nodes B, C, D, get the values 4, 5, 6 respectively. Again as ply 1 is minimizing, so the minimum value out of these i.e. 4 is propagated to A. then from A move will be taken toB.

MIN MAX procedure is straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game beingplayed.

representing the moves that can be made by player in position. We may have 2 players namely PLAYER-TWO in a chessproblem.

2. STATIC (position, player): the static evaluation function, which returns a number representing the goodness of position from the standpoint of player.

The issue in the MINIMAX procedure is when to stop the recursion and call the static evaluation function.

We assume that MIN MAX returns a structure containing both results and that we have two functions, VALUE and PATH that extract the separate components. A function LAST PLY is taken which is assumed to evaluate all of the factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise.

MIN MAX procedure takes three parameters like a board position, a current depth of the search and the players to move. So the initial call to compute the best move from the position CURRENT should be

MIN MAX (CURRENT, 0, PLAYER-ONE)

(If player one is tomove)

Or

MIN MAX (CURRENT, 0, PLAYER-TWO)

(If player two is to move)

Player(Position, Depth):

for each S € SUCCESSORS(Position) do

RESULT = Opponent(S, Depth + 1)

NEW-VALUE = PLAYER-

VALUE(RESULT)

if NEW-VALUE > MAX-SCORE, then

MAX-SCORE = NEW-VALUE

BEST-PATH = PATH(RESULT) + S

return

PATH=BEST=PATH

Opponent(Position, Depth):

for each S € SUCCESSORS(Position) do

RESULT = Player(S, Depth + 1)

NEW-VALUE = PLAYER-

VALUE(RESULT)

if NEW-VALUE < MIN-SCORE, then

MIN-SCORE = NEW-VALUE

BEST-PATH = PATH(RESULT) + S

return

VALUE = MIN-SCORE

PATH=BEST=PATH

Any-Player(Position, Depth):

for each S € SUCCESSORS(Position) do

RESULT = Any-Player(S, Depth + 1)

NEW-VALUE = - VALUE(RESULT)

if NEW-VALUE > BEST-SCORE, then

BEST-SCORE = NEW-VALUE

BEST-PATH = PATH(RESULT) + S

return

VALUE = BEST-SCORE

PATH=BEST-PATH.

MiniMax Procedure:

MAX

Algorithm: MINMAX (position, depth, player)

1. If DEEP_ENOUGH (Position,Depth,Player), then return the structure
   VALUE =Static(Position,Player);
   PATH=Nil;

   This denotes that there is no path from the node and its value is determined by the static evaluation function
2. Else, generate one more ply of the tree by calling the function MOVE_GEN (position, player) and set SUCCESORS to the list it returns.
3. If SUCESSORS is empty,THEN no moves to be made

   RETURN the same structure that would have been returned if LAST_PLY had returned TRUE.
4. If SUCCESORS is not empty,

   THEN examine each element in turn and keep track of the best one.

   For each element SUCC of SUCCESSORS do the following
   for each S € SUCCESSORS(Position) do

   RESULT-SUCC = MININMAX(SUCC, Depth + 1,OPPOSITE(PALYER))
   NEW-VALUE = - VALUE(RESULT-
   SUCC)
   if NEW-VALUE > BEST-SCORE, then
   BEST-SCORE = NEW-VALUE
   BEST-PATH = PATH(RESULT) + S
   return

5. After examining all the nodes,
   RETURN VALUE = BEST-SCORE
   PATH = BEST-PATH

When the initial call to MIN MAX returns, the best move from CURRENT is the first element in the PATH.

Alpha- Beta (α-β) Pruning

- At the player choice, maximize the static evaluation of the next position.>**α** threshold
- At the opponent choice, minimize the static evaluation of the next position. <**β** threshold

When a number of states of a game increase and it cannot be predicted about the states, then we can use the method pruning. Pruning is a method which is used to reduce the no. of states in a game. Alpha- beta is one such pruning technique. The problem with minmax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately we cannot eliminate the exponent, but we can effectively cut it in half.

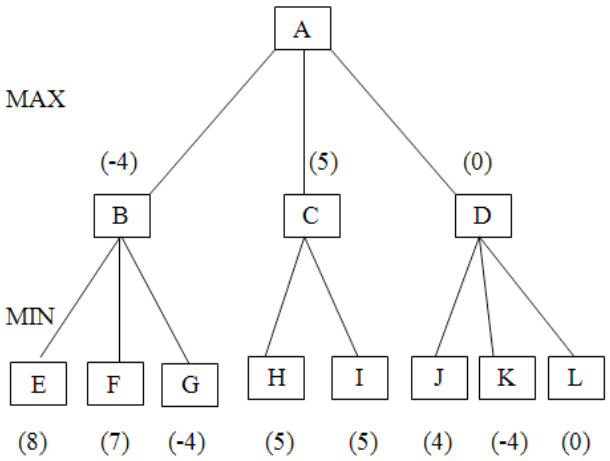
pruning is applied to a standard minmax tree, it returns the same move as minmax would, but prunes away branches that cannot possibly influence the final decision.

Alpha-Beta pruning requires the 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 an upper bound on the value that a minimizing node may be assigned (beta).

The idea of alpha beta pruning is very simple. Alpha beta search proceeds in a depth first fashion rather than searching the entire space. Generally two values, called alpha and beta, are created during the search. The alpha value is associated with MAX nodes and the beta value is with MIN values. The value of alpha can never decrease; on the other hand the value of beta never increases. Suppose the alpha value of A MAX node is 5. The MAX node then need not consider any transmitted value less than or equal to 5 which is associated with any MIN node below it. Alpha is the worst that MAX can score given that MIN will also do its best. Similarly, if a MIN has a beta value of 5, it need not further consider any MAX node below it that has a value of 6 or more.

The general principal is that: consider a node η somewhere in the search tree, such that player has a choice of moving to that node. If player has a better choice K either at the parent node of η or at any choice point further up, then η will never be reached in actual play. So once we have found out enough about η (by examining some of its descendents) to reach this conclusion, we can prune it.

We can also say that "α" is the value of the best choice we have found so far at any choice point along the path for MAX. Similarly "β" is the value of the best choice we have found so far at any choice point along the path for MIN. Consider the following example



Figure

Here at MIN ply, the best value from three nodes is - 4, 5, 0. These will be back propagated towards root anc maximizing move 5 will be taken. Now the node E has the value 8 is far more, then accepted as it is minimizi ply. So, further node E will not be explored. In the situation when more plies are considered, whole sub tı below E will be pruned. Similarly if α=0, β=7, all the nodes and related sub trees having value less than 0 maximizing ply and more than 7 at minimizing ply will be pruned.

Alpha beta search updates the value of α and β as it goes along and prunes the remaining branches at a node soon as the value of the current node is known to be worse than the current α and β value for MAX or M respectively. The effectiveness of alpha- beta pruning is highly dependent on the order in which the successc are examined suppose in a search tree the branching factor is x and depth d. the α-β search needs examining or $x^{d/2}$ nodes to pick up best move, instead of x^{d} for MINMAX.

Consider the example below: after examining node F, its known that the opponent is guaranteec score of -5 or less at C(Since the opponent is the minimizing player). A score of 3 or greater is guaranteed node A,when a move is made to B. Any other move that produces a score of less than 3 is worse than the mo to B and it can be ignored. After examining only F, it is sure that a move to C is worse regardless of the score G.

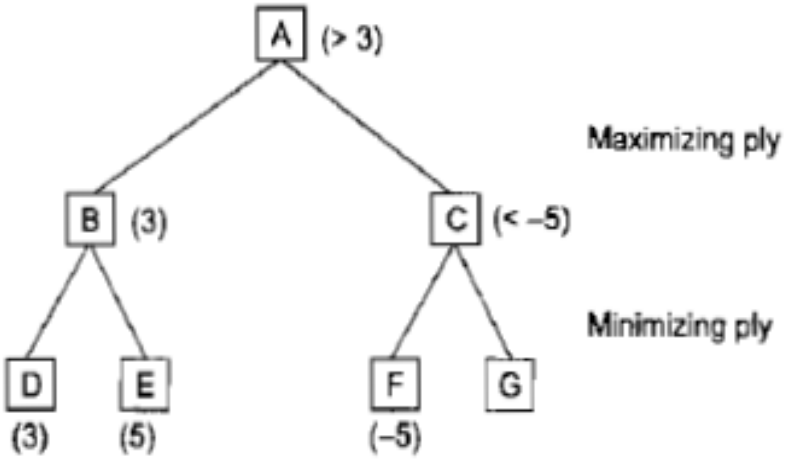


Fig. 12.4 *An Alpha Cutoff*

Usage of both Alpha and Beta

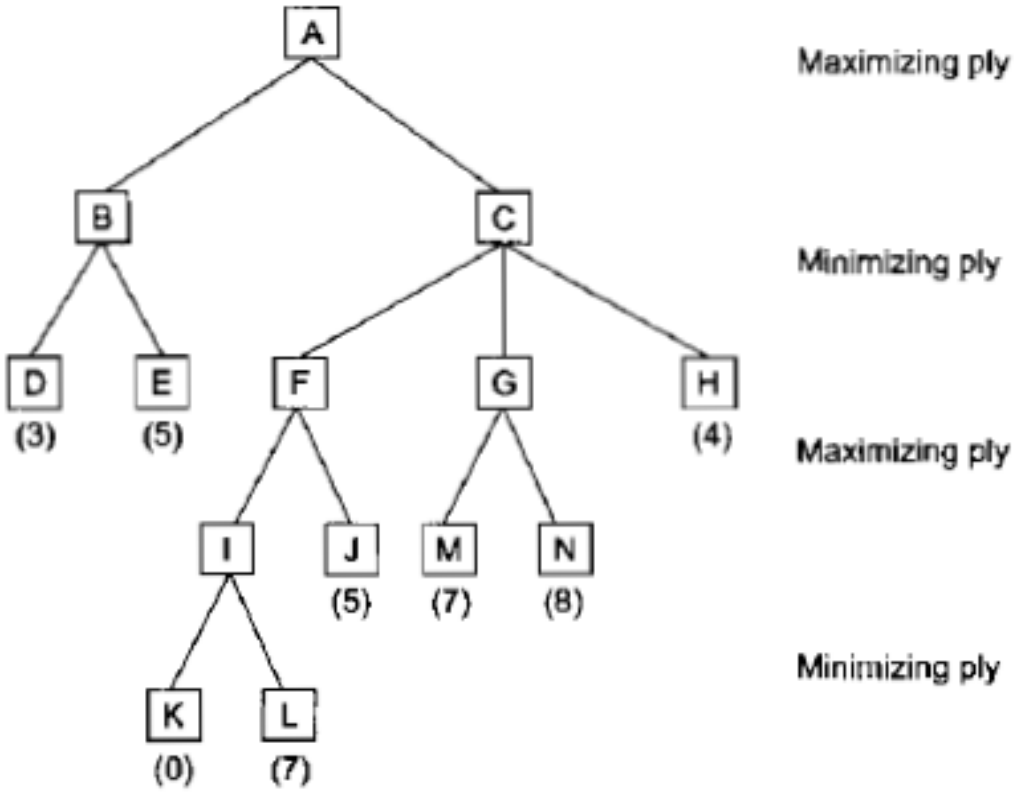
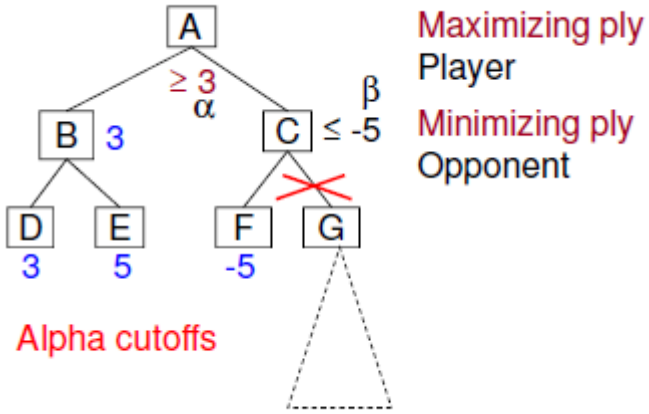
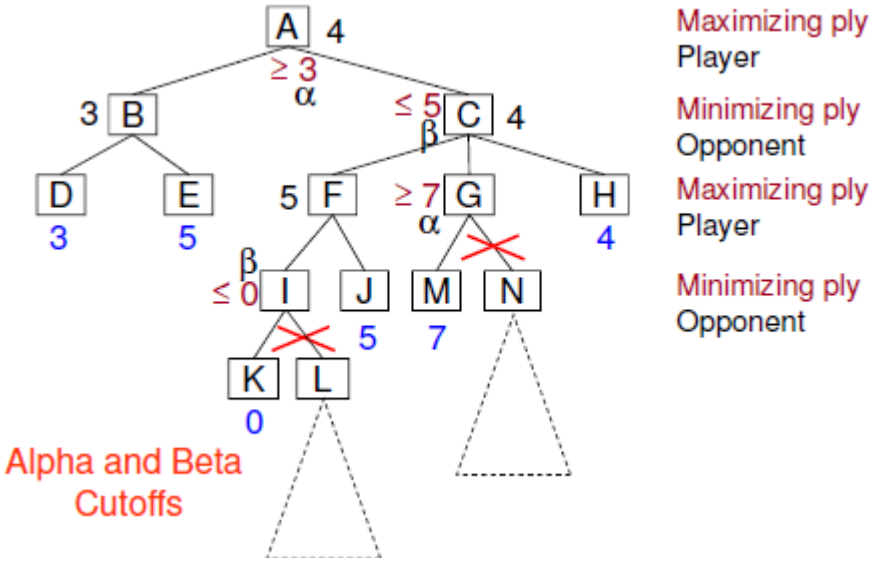


Fig. 12.5 *Alpha and Beta Cutoffs*

In searching this tree, the entire subtree headed by B is searched and its found that A can expect a score at least 3. When this alpha value is passed down to F,it will enable us to skip the exploration of L. The reason After K is examined, I is guaranteed a maximum score of 0 which means that F is guaranteed a minimum of But this is less than alpha's value of 3, so no more branches of I is considered. the maximizing player alrea knows not to choose to move to C and then to I since , if that move is made, the resulting score will be no bet than 0 and a score of 3 can be achieved by moving to B instead.

After cutting off further exploration of I, J is examined, yielding a value of 5 which is assigned as t value of F (since it is the maximum of 5 and 0).this value becomes the value of beta at node C. It indicates that is guaranteed to get a 5 or less. Now expand G.First M is examined and it has a value of 7 which is passed ba to G as its tentative value. Now 7 is compared to beta (5). it is greater and the player and the player whose turn at node C is trying to minimize. So this player will not choose G, which would lead to a score of at least since there is an alternative move to F which will lead to a score of 5. Thus it is not necessary to explore any the other branches of G.





❑ Player(Position, Depth, α, β):

```
for each S ∈ SUCCESSORS(Position) do
    RESULT = Opponent(S, Depth + 1, α, β)
    NEW-VALUE = VALUE(RESULT)
    if NEW-VALUE > α, then
        α = NEW-VALUE
        BEST-PATH = PATH(RESULT) + S
    if α ≥ β then return
        VALUE = α
        PATH = BEST-PATH
return
    VALUE = α;    PATH = BEST-PATH
```

❑ Opponent(Position, Depth, α, β):

```
for each S ∈ SUCCESSORS(Position) do
    RESULT = Player(S, Depth + 1, α, β)
    NEW-VALUE = VALUE(RESULT)
    if NEW-VALUE < β, then
        β = NEW-VALUE
        BEST-PATH = PATH(RESULT) + S
    if β ≤ α then return
        VALUE = β
        PATH = BEST-PATH
return
    VALUE = β;    PATH = BEST-PATH
```

❑ Any-Player(Position, Depth, α, β):

```
for each S ∈ SUCCESSORS(Position) do
    RESULT = Any-Player(S, Depth + 1, - β, - α)
    NEW-VALUE = - VALUE(RESULT)
    if NEW-VALUE > α, then
        α = NEW-VALUE
        BEST-PATH = PATH(RESULT) + S
    if α ≥ β then return
        VALUE = α
        PATH = BEST-PATH
return
    VALUE = α;    PATH = BEST-PATH
```

Algorithm

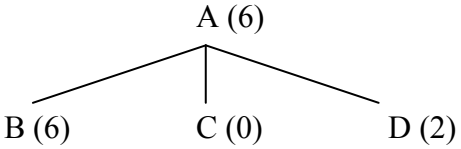
- MINIMAX-A-B(Position, Depth, Player, UseTd, PassTd):
  - UseTd: checked for cutoffs.
  - PassTd: current best value

1. if DEEP-ENOUGH(Position, Depth), then return:
   VALUE = STATIC(Position, Player)
   PATH = nil
2. SUCCESSORS = MOVE-GEN(Position, Player)
3. if SUCCESSORS is empty, then do as in Step 1
4. if SUCCESSORS is not empty:
   for each SUCC ∈ SUCCESSORS do
   RESULT-SUCC = MINIMAX-A-B(SUCC, Depth + 1, Opp(Player), - PassTd, - UseTd)
   NEW-VALUE = - VALUE(RESULT-SUCC)
   if NEW-VALUE > PassTd, then:
   PassTd = NEW-VALUE
   BEST-PATH = PATH(RESULT-SUCC) + SUCC
   if PassTd ≥ UseTd, then return:
   VALUE = PassTd
   PATH = BEST-PATH
5. Return: VALUE = PassTd; PATH = BEST-PATH

Additional Refinements

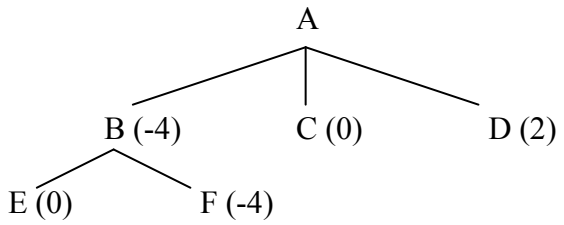
- In addition to α-β pruning, there are variety of other modifications to MINMAX procedure, which c improve its performance. One of the factors is that when to stop going deeper in the search tree.

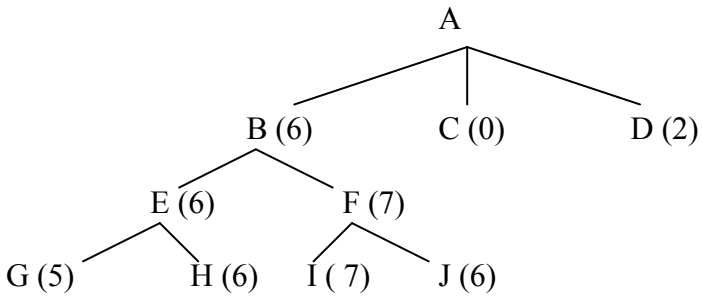
Waiting for Quiescence



- Suppose node B is expanded one more level and the result is as shown below.

- The estimate of worth of B has changed. This may happen if opponent has significantly improved.
- If we stop exploring the tree at this level and assign -4 to B and therefore decide that B is not a go move.
- To make sure that such short term measures don't unduly influence our choice of move, we shou continue the search until no such drastic change occurs from one level to the next or till the condition stable. This is called waiting for **quiescence**.
- Go deeper till the condition is stable before deciding the move.
- Now B is again looks like a reasonable move. Waiting for quiescence helps in avoiding the horizon effe in which an inevitable bad event can be delayed . The effect may make a move look good despite the fa that the move might be better if delayed past the horizon.





Secondary search

- To provide a double check, explore a game tree to an average depth of more ply and on the basis that, chose a particular move.
- Here chosen branch is further expanded up to two levels to make sure that it still looks good.
- This technique is called secondary search.

Singular Extension:

Success form of secondary search is called singular extensions.

The leaf node is be far superior to its siblings and if the value of the entire search depends critically on correctness of that node's value, then the node is expanded one extra ply.

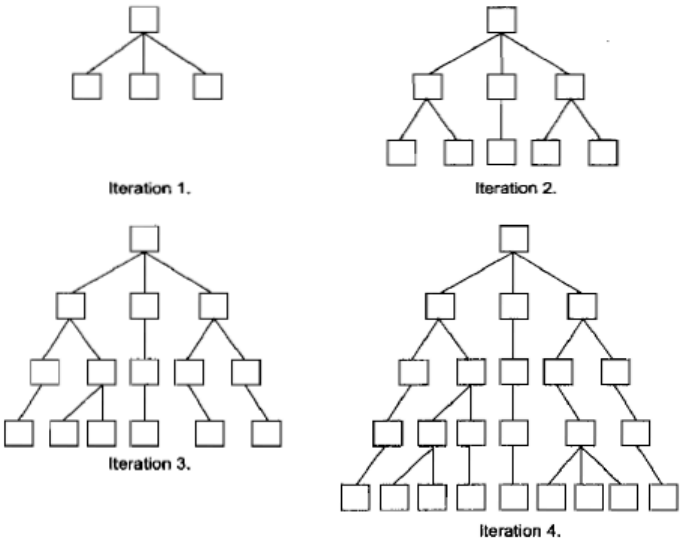
Using Book Moves

- For complicated games taken as wholes, it is not feasible to select a move by simply looking up the curr game configuration in a catalogue and extracting the correct move.
- But for some segment of some games, this approach is reasonable.
- For example in Chess, both opening and endgame sequences are highly stylized.
- In these situations, the performance of a program can be considerably enhanced if it is provided with a of moves(called book moves) that should be made.

- The use of book moves in the opening and end game sequences combined with the use of the minir search procedure for the midgame, provides a good example of the way that knowledge and search car combined in a single program to produce more effective results.

Alternative to MINMAX

- Even with refinements, MINMAX still has some problematic aspects.
- It relies heavily on the assumption that the opponent will always choose an optimal move. This assumpt is acceptable in winning situations.
- But in losing situation it might be better to take risk that opponent will make a mistake.
- Suppose we have to choose one move between two moves, both of which if opponent plays perfectly, l to situation that are very bad for us but one is slightly less bad than other.
- Further less promising move could lead to a very good situation for us if the opponent makes a sir mistake.
- MINMAX would always choose the bad move. We instead choose the other one.
- Similar situation occurs when one move appears to be only slightly more advantageous then another.
- It might be better to choose less advantageous move.
- To implement such system we should have model of individual opponents playing style. **Iterat Deepening**
- Rather than searching to a fixed depth in the game tree, first search only single ply, then apply MINM to 2 ply, further 3 ply till the final goal state is searched [CHESS 5 is based on this].
- There is a good reason why iterative deepening is popular for chess-playing. In competition play there is average amount of time allowed per move.
- The idea that capitalizes on this constraint is to do as much look-ahead as can be done in the available ti If we use iterative deepening, we can keep on increasing the look-ahead depth until we run out of time.
- We can arrange to have a record of the best move for a given look-ahead even if we have to interrupt attempt to go one level deeper.



- Depth-First Iterative Deepening (DFID)

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

- This will find the shortest solution path to the goal state. The amount of memory used is proportional to

number of nodes in that solution path.

- The problem in DFS is that there is no way to know in advance how deep the solution lies in the sea space.
- DFID avoid the problem of choosing cutoffs without sacrificing efficiency and in fact DFID is the opti algorithm for uninformed search.

- For informed heuristic search, iterative deeping can be used to improve the performance of the algorithm.
- Since the major difficulty is the large amount of memory it requires to maintain the search node l iterative deepening can be of considerable service.

❑ Iterative-Deepening-A* (IDA*)

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

- IDA* is guaranteed to find a optimal solution.

- **Knowledge and Representation**

  Problem solving requires large amount of knowledge and some mechanism for manipulating that knowledge.

  The Knowledge and the Representation are distinct entities, play a central but distinguishable roles in intelligent system.

  - **Knowledge** is a description of theworld;
    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.

  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 :

(a) *knowledge level* at which facts are described, and

(b) *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 Factsand Representation**

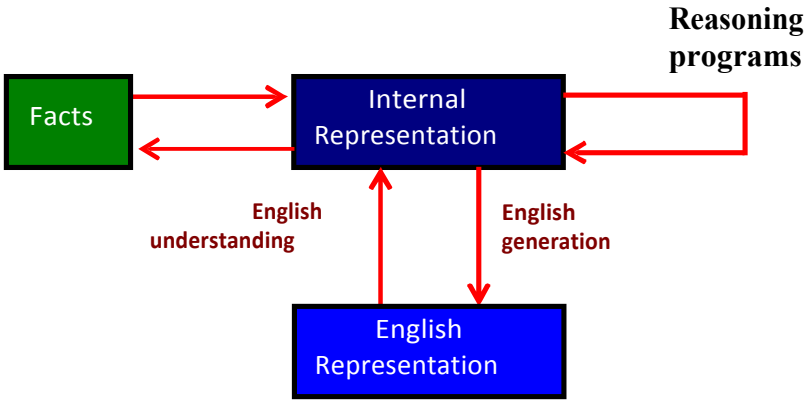
Knowledge is a collection of "*facts*" from some domain.

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



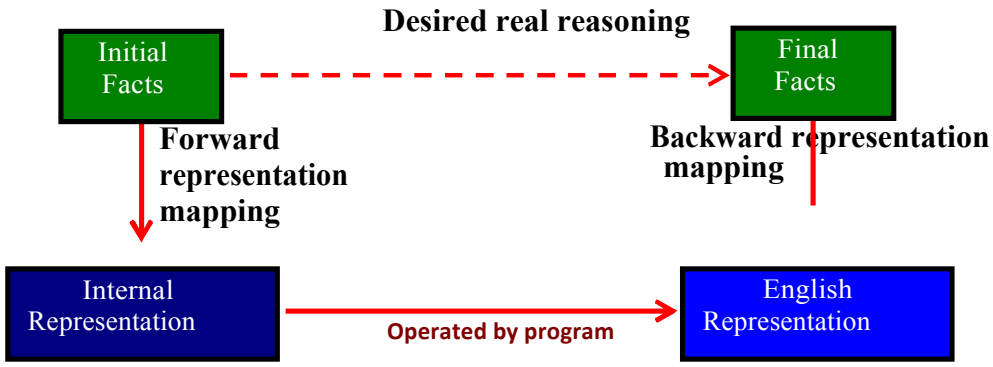
Facts	**Representations**
◊ **Spot is a dog**	A *fact* represented in *English sentence*
◊ **dog (Spot)**	Using *forward mapping function* the above *fact* is represented in *logic*
◊ **" x : dog(x) ® hastail (x)**	A *logical representation* of the *fact* that *"all dogs havetails"*

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

◊ **hastail (Spot)**	A new object representation
◊ **Spot has a tail** ***[it is newknowledge]***	Using *backward mapping function* to generate English sentence

- **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 indicates 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 the attention of the inference mechanisms in the most promising direction.
◊ Acquisitional Efficiency	The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

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

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;

These KR schemes are detailed in next few slides

- **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 attributesof 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.

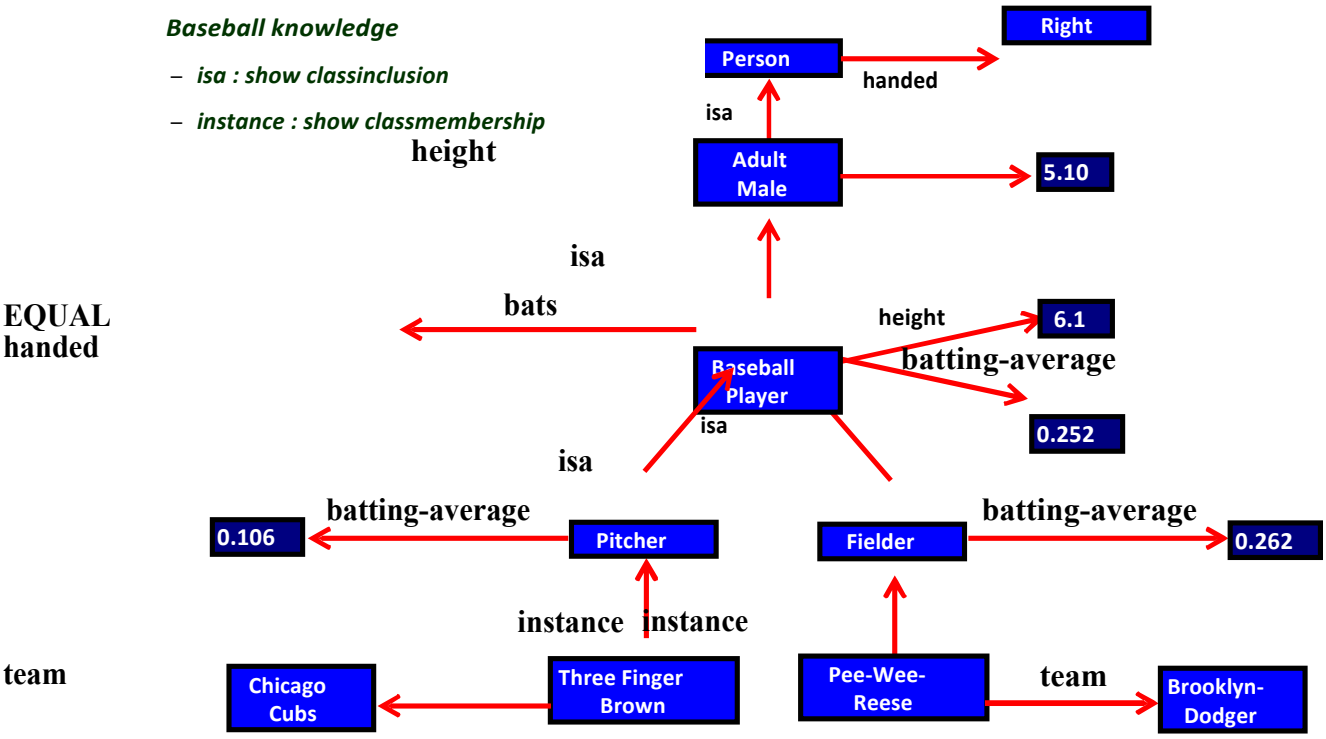


Fig. Inheritable knowledge representation (KR)

‡ 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, see 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 This attribute had a value stored explicitly the knowledge base.
- batting–average(Three-Finger-Brown) = 0.106.Since there is no value for batt average stored explicitly for Three-finger-brown, instance attribute to Pitcher is follov and the value stored there is extracted.
- 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.

Inference procedure implements the standard logical rules of inference.

- Some procedures reason forward from given facts to conclusion.
- Some procedures reason backward from desired conclusion to given facts.
- One of the most commonly used of these procedures is Resolution which exploits a proof by contradiction strategy.

- **Declarative/Procedural Knowledge**

Differences between Declarative/Procedural knowledge are 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.

```
Baseball-Player
    isa:            Adult-Male
    bats:           (lambda (x)
                        (prog ()
                            L1
                          (cond ((caddr x) (return (caddr x)))
                                (t (setq x (eval (cadr x)))
                                   (cond (x (go L1))
                                         (t (return nil)))))))
    height:         6-1
    batting-average: .252
```

- LISP will work given a particular way of sorting attributes and values in a list, it does not lend itself to being reasoned.
- LISP representation is powerful since it makes explicit use of the name of the node whose value for handed is to be found.
- Because of the difficulty in reasoning with LISP, attempts have been made to find other ways of representing procedural knowledge so that it can relatively easily be manipulated both by programs and by people.

- The most commonly used technique is the use of production rules.

If: ninth inning, and
score is close, and
less than 2 outs, and
first base is vacant, and
batter is better hitter than next batter,
Then: walk the batter.

Fig. 4.9 *Procedural Knowledge as Rules*

Issues in Knowledge Representation

The fundamental goal of Knowledge Representation is to facilitate inference(conclusions)fromknowledge.

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 we 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*
- *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 to represent other views of relationship:

‡ 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 associated with Pee-Wee-Reese,

team = Brooklyn– Dodgers ,

and the other associated with Brooklyn Dodgers ,

team-members = Pee-Wee-Reese, ...

This second approach is followed in semantic net and frame-based systems. It is accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slot by forcing them to be declared and then 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 :**

Sometimes value of attributes are specified explicitly when a knowledge base is created. Often the values are not specified explicitly. Several kinds of information can play a role in this reasoning including,

- Information about the type of the value. Example height : must be in a unit of length,
- Constraints on the value, often stated in terms of related entities. Example age: of person cannot be greater than the age of person's parents.
- Rules for computing the value when it is needed. Some rules are called backward rules or if-needed rules.
- Rules that describe action that should be taken if a value ever becomes known. These rules are called forward rules or sometimes if-added rules.

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.

- Introduce an explicit notation for temporal interval.if two different values are asserted for the same temporal interval, signal a contradiction automatically.
- Assume that the only temporal interval that is of interest is now. So if a new value is asserted replace the old value.

- **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.

- An alternative solution to this problem is represent the fact that spotting is really a spe type of seeing explicitly in the representation of the fact.
- this can be rewritten as

  saw(agent,(John),
  Object(sue),
  timespan(briefly)).

From this representation, the fact that John saw Sue is immediately accessible. but the fact that he spotted is more difficult to get to.

- **Set ofObjects**

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: **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 :

(a) Name, as in the example – Ref Fig. Inheritable KR, the node - Baseball- Player in semantic Net and the predicates as Ball and Batter in logical representation. This simple representation does make it possible to associate predicates with sets. But it does not provide information about the set it represents.

(b) Extensional definition is to list the numbers, and

(c) 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.