**UNIT-04**

**Game Playing and Natural Language Processing**

**UNIT-04/LECTURE-01**

# Game Playing

## Formulating Game Playing as Search

- Consider 2-person, zero-sum, perfect information (i.e., both players have access to complete information about the state of the game, so no information is hidden from either player) games. Players alternate moves and there is no chance (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, and Othello
- **Iterative** methods apply here because search space is too large for interesting games to search for a "solution." Therefore, search will be done before EACH move in order to select the best next move to be made.
- **Adversary** methods needed because alternate moves are made by an opponent who is trying to win. Therefore must incorporate the idea that an adversary makes moves that are "not controllable" by you.
- **Evaluation function** is used to evaluate the "goodness" of a configuration of the game. Unlike in heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node, here the evaluation function, also called the *static evaluation function* estimates board quality in leading to a win for one player.
- Instead of modeling the two players separately, the zero-sum assumption and the fact that we don't have, in general, any information about how our opponent plays, means we'll use a single evaluation function to describe the goodness of a board with respect to BOTH players. That is, *f(n)* = large positive value means the board associated with node *n* is good for me and bad for you. *f(n)* = large negative value means the board is bad for me and good for you. *f(n)* near 0 means the board is a neutral position. *f(n)* = +infinity means a winning position for me. *f(n)* = -infinity means a winning position for you.
- Example of an Evaluation Function for Tic-Tac-Toe:
  *f(n)* = [number of 3-lengths open for me] - [number of 3-lengths open for you]
  where a 3-length is a complete row, column, or diagonal.
- Most evaluation functions are specified as a weighted sum of "features:" (w1 * feat1) + (w2 * feat2) + ... + (wn * featn). For example, in chess some features evaluate piece placement on the board and other features describe configurations of several pieces. Deep Blue has about 6000 features in its evaluation function.

## Game Trees

- Root node represents the configuration of the board at which a decision must be made as to what is the best single move to make next. If it is my turn to move, then the root is labeled a "MAX" node indicating it is my turn; otherwise it is labeled a "MIN" node to indicate it is my opponent's turn.
- Arcs represent the possible legal moves for the player that the arcs emanate from
- Each level of the tree has nodes that are all MAX or all MIN; since moves alternate, the nodes at level *i* are of the opposite kind from those at level *i*+1

## Searching Game Trees using the Minimax Algorithm

Steps used in picking the next move:

1. Create start node as a MAX node (since it's my turn to move) with current board configuration
2. Expand nodes down to some depth (i.e., ply) of lookahead in the game
3. Apply the evaluation function at each of the leaf nodes
4. "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.
5. Pick the operator associated with the child node whose backed up value determined the value at the root

Note: The above process of "backing up" values gives the optimal strategy that BOTH players would follow given that they both have the information computed at the leaf nodes by the evaluation function. This is implicitly assuming that your opponent is using the same static evaluation function you are, and that they are applying it at the same set of nodes in the search tree.

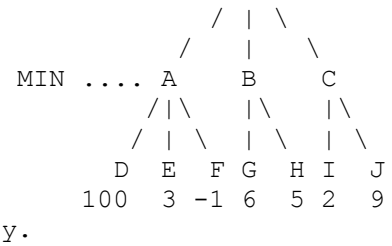
## Minimax Algorithm in Java

```
public int minimax(s)
{
  int [] v = new int[#ofSuccessors];
  if (leaf(s))
       return(static-evaluation(s));
  else
  {
    // s1, s2, ..., sk are the successors of s
    for (int i = 1; i < #ofSuccessors; i++)
    {
      v[i] = minimax(si);
    }
    if (node-type(s) = max)
      return max(v1, ..., vk);
      else return min(v1, ..., vk);
   }
}
```

## Example of Minimax Algorithm

For example, in a 2-ply search, the MAX player considers all (3) possible moves.

The opponent MIN also considers all possible moves. The evaluation function is applied to the leaf level only.




Once the static evaluation function is applied at the leaf nodes, backing up values can begin. First we compute the backed-up values at the parents of the leaves. Node A is a MIN node corresponding to the fact that it is a position where it's the opponent's turn to move. A's backed-up value is -1 (= min(100, 3, -

1), meaning that if the opponent ever reaches the board associated with this node, then it will pick the move associated with the arc from A to F. Similarly, B's backed-up value is 5 (corresponding to child H) and C's backed-up value is 2 (corresponding to child I).

Next, we backup values to the next higher level, in this case to the MAX node S. Since it is our turn to move at this node, we select the move that looks best based on the backed-up values at each of S's children. In this case the best child is B since B's backed-up value is 5 (= max(-1, 5, 2)). So the minimax value for the root node S is 5, and the move selected based on this 2-ply search is the move associated with the arc from S to B.

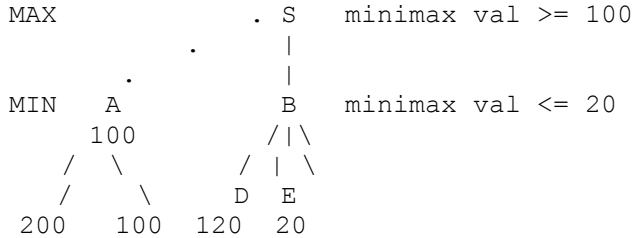
It is important to notice that the backed-up values are used at nodes A, B, and C to evaluate which is best for S; we do *not* apply the static evaluation function at any non-leaf node. Why? Because it is assumed that the values computed at nodes farther ahead in the game (and therefore lower in the tree) are more accurate evaluations of quality and therefore are preferred over the evaluation function values if applied at the higher levels of the tree.

Notice that, in general, the backed-up value of a node changes as we search more plies. For example, A's backed-up value is -1. But if we had searched one more ply, D, E and F will have their own backed-up values, which are almost certainly going to be different from 100, 3 and -1, respectively. And, in turn, A will likely not have -1 as its backed-up value. We are implicitly assuming that the deeper we search, the better the quality of the final outcome.

## Alpha-Beta Pruning

- Minimax computes the optimal playing strategy but does so inefficiently because it first generates a complete tree and then computes and backs up static-evaluation-function values. For example, from an average chess position there are 38 possible moves. So, looking ahead 12 plies involves generating $1 + 38 + 38^2 + ... + 38^{12} = (38^{12}-1)/(38-1)$ nodes, and applying the static evaluation function at $38^{12}$ = 9 billion billion positions, which is far beyond the capabilities of any computer in the foreseeable future. Can we devise another algorithm that is guaranteed to produce the same result (i.e., minimax value at the root) but does less work (i.e., generates fewer nodes)? Yes---Alpha-Beta.
- Basic idea: "If you have an idea that is surely bad, don't take the time to see how truly awful it is." -- Pat Winston

Example of how to use this idea for pruning away useless work:




In the above example we are performing a depth-first search to depth (ply) 2, where children are generated and visited left-to-right. At this stage of the search we have just finished generating B's second child, E, and computed the static evaluation function at E (=20). Before generating B's third child notice the current situation: S is a MAX node and its left child A has a minimax value of 100, so S's minimax value **must** eventually be some number >= 100. Similarly, B has generated two children, D and E, with values 120 and 20, respectively, so B's final minimax

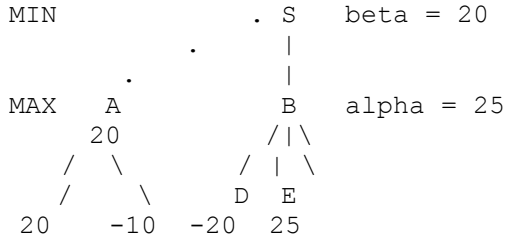
value must be <= min(120, 20) = 20 since B is a MIN node.

The fact that S's minimax value must be at least 100 while B's minimax value must be no greater than 20 means that no matter what value is computed for B's third child, S's minimax value will be 100. In other words, S's minimax value does not depend on knowing the value of B's third child. Hence, we can cutoff the search below B, ignoring generating any other children after D and E.

## Alpha-Beta Algorithm

- Traverse the search tree in depth-first order
- Assuming we stop the search at ply *d*, then at each of these nodes we generate, we apply the static evaluation function and return this value to the node's parent
- At each non-leaf node, store a value indicating the best backed-up value found so far. At MAX nodes we'll call this **alpha**, and at MIN nodes we'll call the value **beta**. In other words, alpha = best (i.e., maximum) value found so far at a MAX node (based on its descendant's values). Beta = best (i.e., minimum) value found so far at a MIN node (based on its descendant's values).
- The alpha value (of a MAX node) is *monotonically non-decreasing*
- The beta value (of a MIN node) is *monotonically non-increasing*
- Given a node *n*, cutoff the search below *n* (i.e., don't generate any more of *n*'s children) if
    - *n* is a MAX node and alpha(n) >= beta(i) for some MIN node ancestor *i* of *n*. This is called a **beta cutoff**.
    - *n* is a MIN node and beta(n) <= alpha(i) for some MAX node ancestor *i* of *n*. This is called an **alpha cutoff**.
- In the example shown above an alpha cutoff occurs at node B because beta(B) = 20 < alpha(S) = 100

An example of a beta cutoff at node B (because alpha(B) = 25 > beta(S) = 20) is shown below:




- To avoid searching for the ancestor nodes in order to make the above tests, we can carry *down* the tree the best values found so far at the ancestors. That is, at a MAX node *n*, beta = minimum of all the beta values at MIN node ancestors of *n*. Similarly, at a MIN node *n*, alpha = maximum of all the alpha values at MAX node ancestors of *n*. Thus, now at each non-leaf node we'll store both an alpha value and a beta value. https://www.rgpvonline.com
- Initially, assign to the root values of alpha = -infinity and beta = +infinity
- See the text for a pseudocode description of the full Alpha-Beta algorithm

## Example of Alpha-Beta Algorithm on a 3-Ply Search Tree

Below is a search tree where a beta cutoff occurs at node F and alpha cutoffs occur at nodes C and D. In this case we've pruned 10 nodes (O,H,R,S,I,T,U,K,Y,Z) from the 26 that are generated by Minimax.

## Effectiveness of Alpha-Beta

- Alpha-Beta is guaranteed to compute the same minimax value for the root node as computed by Minimax
- In the worst case Alpha-Beta does NO pruning, examining *b^d* leaf nodes, where each node has *b* children and a *d*-ply search is performed
- In the best case, Alpha-Beta will examine only *(2b)^(d/2)* leaf nodes. Hence if you hold fixed the number of leaf nodes (as a measure of the amount of time you have allotted before a decision must be made), then you can search twice as deep as Minimax!
- The best case occurs when each player's best move is the leftmost alternative (i.e., the first child generated). So, at MAX nodes the child with the largest value is generated first, and at MIN nodes the child with the smallest value is generated first.
- In the chess program Deep Blue, they found empirically that Alpha-Beta pruning meant that the average branching factor at each node was about 6 instead of about 35-40

## Cutting off Search (or, when to stop and apply the evaluation function)

So far we have assumed a fixed depth *d* where the search is stopped and the static evaluation function is applied. But there are variations on this that are important to note:

- Don't stop at **non-quiescent nodes**. If a node represents a state in the middle of an exchange of pieces, then the node is not quiescent and therefore the evaluation function may not give a reliable estimate of board quality. Or, another definition for chess: "a state is non-quiescent if any piece is attacked by one of lower value, or by more pieces than defenses, or if any check exists on a square controlled by the opponent." In this case, expand more nodes and only apply the evaluation function at quiescent nodes.
- The identification of non-quiescent nodes partially deals with the **horizon effect**. A negative horizon is where the state seen by the evaluation function is evaluated as better than it really is because an undesirable effect is just beyond this node (i.e., the search horizon). A positive horizon is where the evaluation function wrongly underestimates the value of a state when positive actions just over the search horizon indicate otherwise.
- **Iterative Deepening** is frequently used with Alpha-Beta so that searches to successively deeper plies can be attempted if there is time, and the move selected is the one computed by the

deepest search completed when the time limit is reached.

# Natural Language Processing

Natural Language Processing (NLP) is the process of computer analysis of input provided in a human language (natural language), and conversion of this input into a useful form of representation.

The field of NLP is primarily concerned with getting computers to perform useful and interesting tasks with human languages. The field of NLP is secondarily concerned with helping us come to a better understanding of human language.

- The input/output of a NLP system can be: –
  **written text**
  – **speech**
- We will mostly concerned with written text (not speech).
- To process written text, we need:
  – **lexical, syntactic, semantic knowledge about the language** – **discourse information, real world knowledge**
- To process spoken language, we need everything required to process written text, plus the challenges of speech recognition and speech synthesis.

There are two components of NLP.

1. **Natural Language Understanding**
   – Mapping the given input in the natural language into a useful representation.
   – Different level of analysis required: ***morphological analysis,*** ***syntactic analysis, semantic analysis, discourse analysis,*** …
2. **Natural Language Generation**
   – Producing output in the natural language from some internal representation.
   – Different level of synthesis required: ***deep planning*** (what to say), ***syntactic generation***
3. NL Understanding is much harder than NL Generation. But, still both of them are hard.

The difficulty in NL understanding arises from the following facts:

- Natural language is extremely rich in form and structure, and **very ambiguous**. – How to represent meaning,
  – Which structures map to which meaning structures.

- One input can mean many different things. Ambiguity can be at different levels.
  - Lexical (word level) ambiguity -- different meanings of words
  - Syntactic ambiguity -- different ways to parse the sentence
  - Interpreting partial information -- how to interpret pronouns
  - Contextual information -- context of the sentence may affect the meaning of that sentence.
- Many input can mean the same thing.
- Interaction among components of the input is not clear.

The following language related information are useful in NLP:

- **Phonology** – concerns how words are related to the sounds that realize them.
- **Morphology** – concerns how words are constructed from more basic meaning units called morphemes. A morpheme is the primitive unit of meaning in a language.
- **Syntax** – concerns how can be put together to form correct sentences and determines what structural role each word plays in the sentence and what phrases are subparts of other phrases.
- **Semantics** – concerns what words mean and how these meaning combine in sentences to form sentence meaning. The study of context-independent meaning.
- **Pragmatics** – concerns how sentences are used in different situations and how use affects the interpretation of the sentence.
- **Discourse** – concerns how the immediately preceding sentences affect the interpretation of the next sentence. For example, interpreting pronouns and interpreting the temporal aspects of the information.
- **World Knowledge** – includes general knowledge about the world. What each language user must know about the other's beliefs and goals.

## Natural Language Understanding

The steps in natural language understanding are as follows:

Words

↓

Morphological Analysis

↓

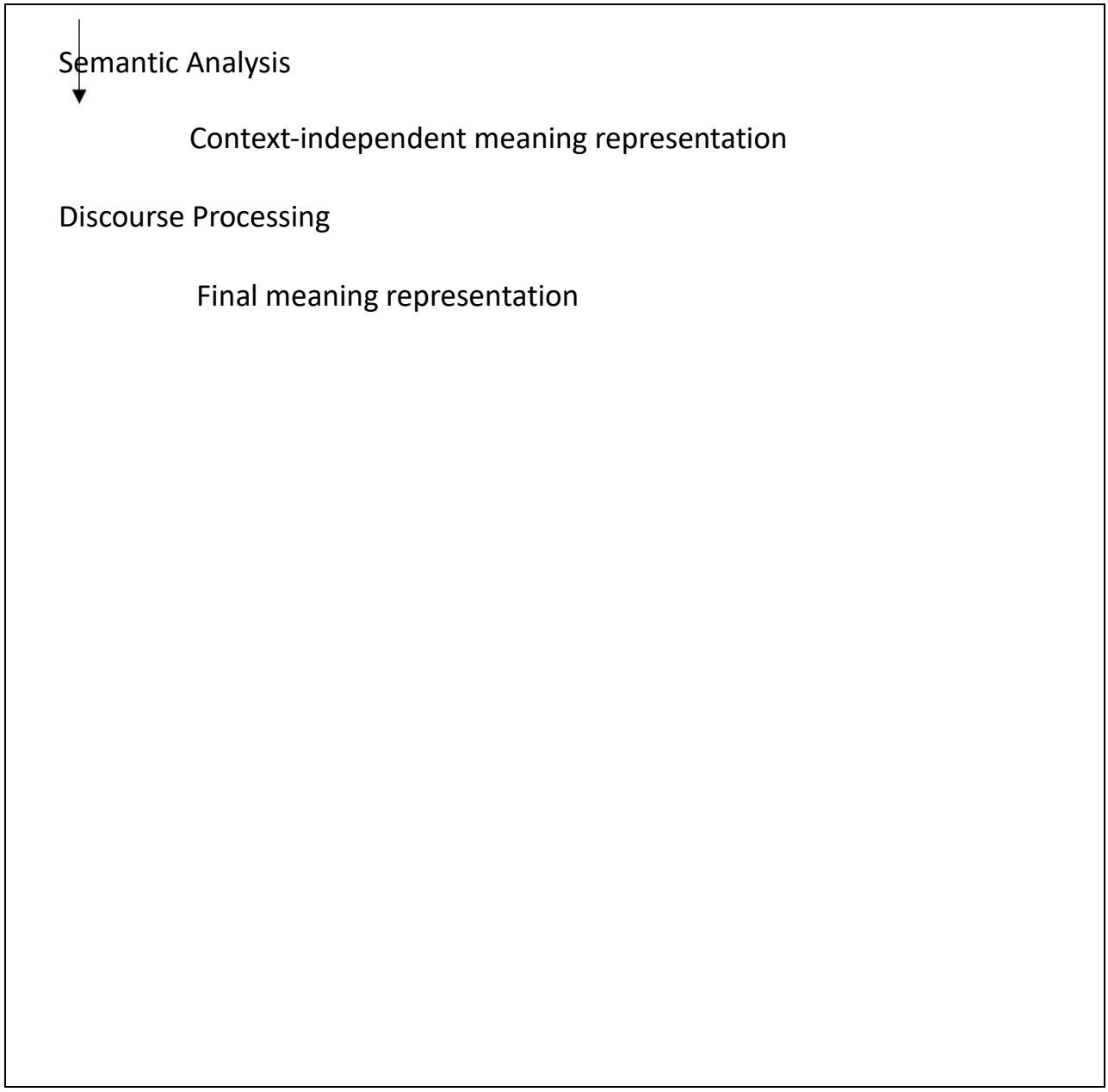
Morphologically analyzed words *(another step: POS tagging)*

↓

Syntactic Analysis

↓

Syntactic Structure


Semantic Analysis
Context-independent meaning representation
Discourse Processing
Final meaning representation