Adversarial Search

Slides mainly compiled from material by: C. R. Dyer
University of Wisconsin-Madison
(http://www.cs.wisc.edu/~dyer/cs540/notes/games.html)

Game Playing as Search

- For 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, the players alternate moves and there is no chance (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello, etc.
- 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.

Game Playing as Search

- Adversary methods are needed because alternate moves are made by an opponent who is trying to win. Therefore, we 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 a state. 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.

Game Playing as Search

- 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 can 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 one player and bad for the other and vice versa. f(n) near 0 means the board is a neutral position.
- Example of an Evaluation Function for Tic-Tac-Toe:
  f(n) = [number of 3lengths open for P-1] - [number of 3lengths open for P-2]
  where a 3length is a complete row, column, or diagonal.

Game Trees

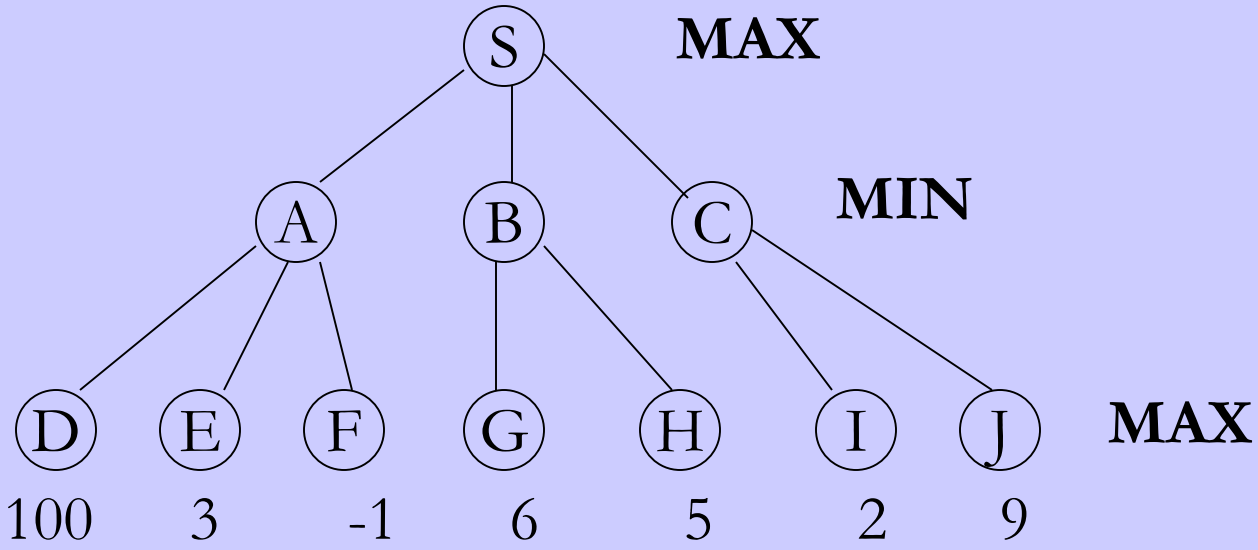
- Root node represents the configuration of the board at which a decision must be made as to what is the best single move to make next. If it is say P-1’s turn to move, then the root is labeled a "MAX" node otherwise it is labeled a "MIN" node to indicate it is the opponent's turn i.e. for P-2.
- Arcs 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

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.



Procedure

- 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).

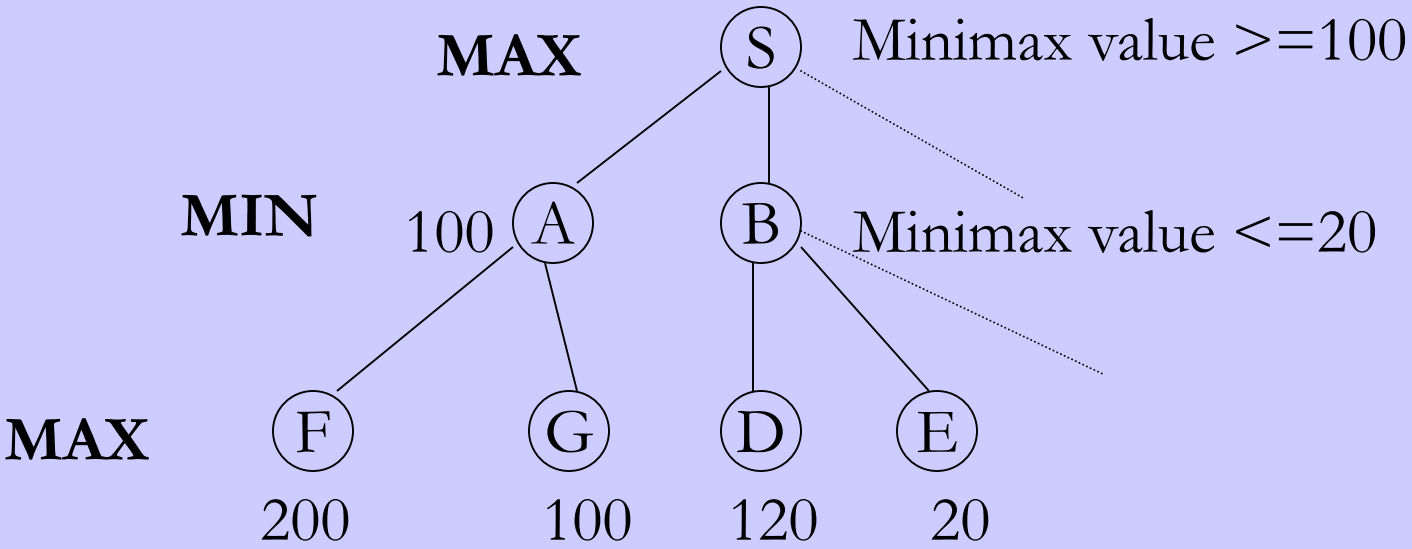
Procedure

- 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.

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 $38 + 38^2 + ... + 38^{12} = 38(38^{12}-1/(38-1)$ 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)?

Example of Alpha-Beta Pruning



- 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).

Example of Alpha-Beta Pruning

- 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(if any). 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).

Alpha-Beta Algorithm

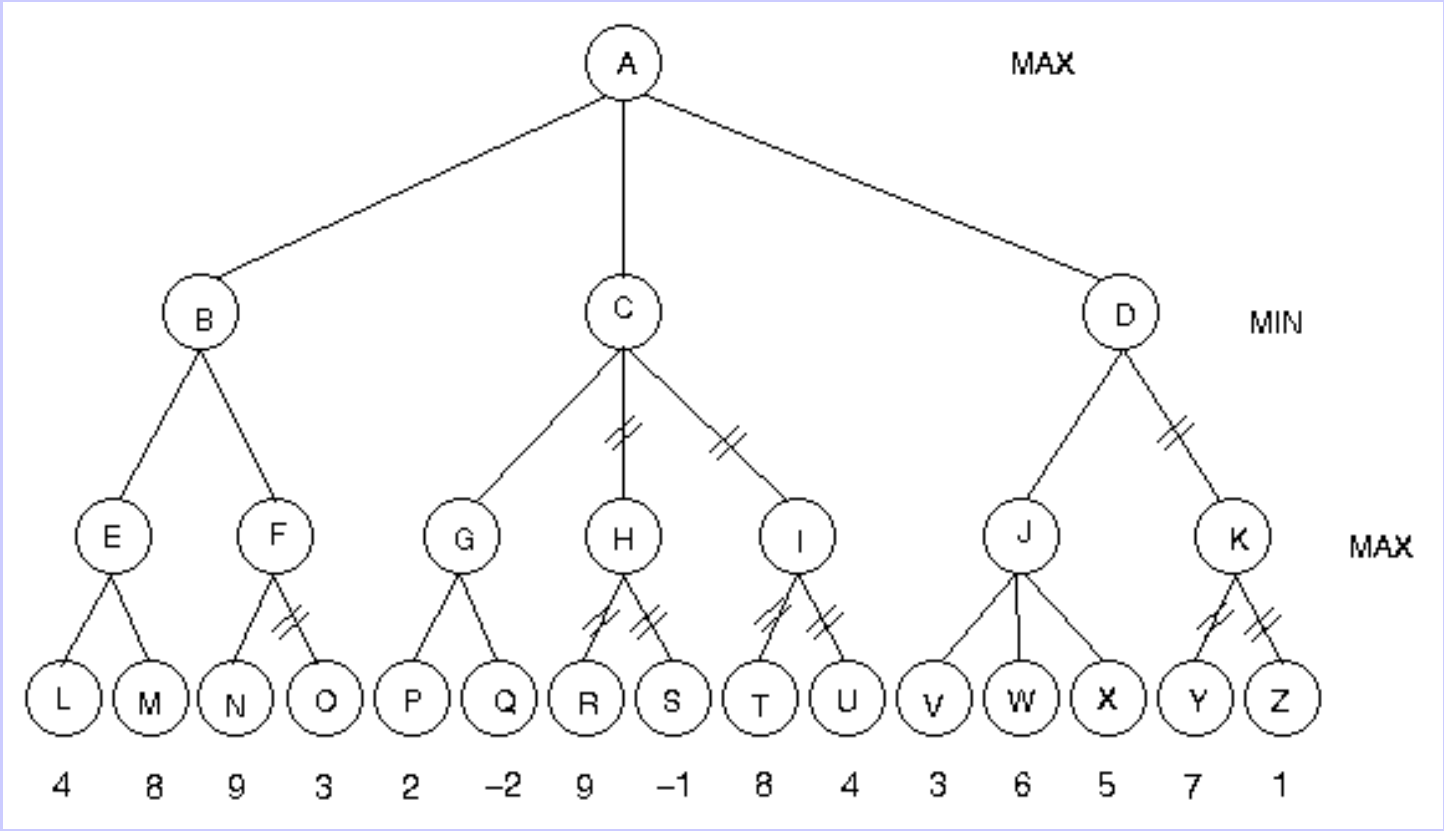
- The alpha value (of a MAX node) is *monotonically non-decreasing*
- The beta value (of a MIN node) is *monotonically non-increasing*
- Given a node *n*, cutoff the search below *n* (i.e., don't generate any more of *n*'s children) if
  - *n* is a MAX node and alpha(n) >= beta(i) for some MIN node ancestor *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**

Alpha-Beta Algorithm

- 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.
- Initially, assign to the root values of alpha = -infinity and beta = +infinity

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!

Effectiveness of Alpha-Beta

- 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, it was empirically found that Alpha-Beta pruning meant that the average branching factor at each node was about 6 instead of about 35-40