**Adversarial Search**

Mathematical game theory, a branch of economics, views any multi-agent environment as a game, provided that the impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive.1 In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, zero-

sum games of perfect information (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents’ utility functions that makes the situation adversarial.

Games, unlike most of the toy problems studied previously are interesting because they are too hard to solve. For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35100 or 10154 nodes (although the search graph has “only” about 1040 distinct nodes). Games, like the real world, therefore require the ability to make some decision even when calculating the optimal decision is infeasible. Games also penalize inefficiency severely. Whereas an implementation of A∗ search that is half as efficient will simply take twice as long to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal. Game-playing research has therefore spawned a number of interesting ideas on how to make the best possible use of time.

We begin with a definition of the optimal move and an algorithm for finding it. We then look at techniques for choosing a good move when time is limited.

**Pruning** allows us to ignore portions of the search tree that make no difference to thefinal choice, and heuristic evaluation functions allow us to approximate the true utility of a state without doing a complete search.

We first consider games with two players, whom we call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following elements:

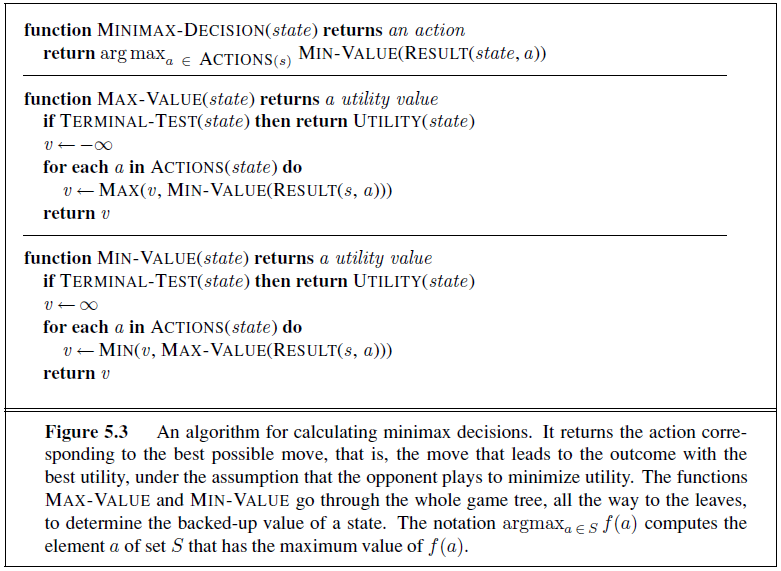
* S0: The initial state, which specifies how the game is set up at the start.
* PLAYER(s): Defines which player has the move in a state.
* ACTIONS(s): Returns the set of legal moves in a state.
* RESULT(s, a): The transition model, which defines the result of a move.
* TERMINAL-TEST(s): A terminal test, which is true when the game is over and false
* TERMINAL STATES otherwise. States where the game has ended are called terminal states.
* UTILITY(s, p): A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state *s* for a player *p*. In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2. Some gameshave a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192. A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or 1/2 + 1/2. “Constant-sum” would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of 1/2.

**The minimax algorithm**

Minimax is a kind of [backtracking](https://www.geeksforgeeks.org/tag/backtracking/) algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

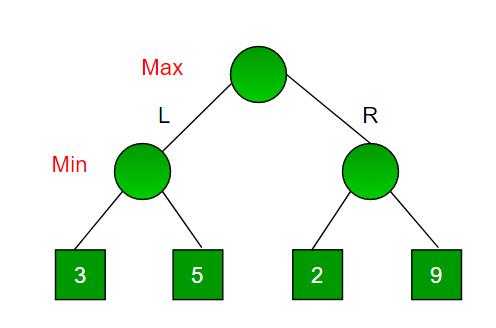
Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.



**Time Complexity:** O(bd)

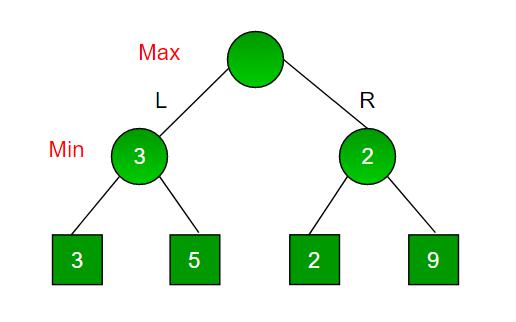
**Space Complexity:** O(bd)

**Example:** Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e., you are at the root and your opponent at next level. **Which move you would make as a maximizing player considering that your** **opponent also plays optimally?**

****

Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

* Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the **minimizer** it will definitely choose the least among both, that is 3
* Maximizer goes RIGHT: It is now the **minimizers** turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.Now the game tree looks like below :



Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

