Module 3

Problem Solving using Search- (Two agent)

3.1 Instructional Objective

- The students should understand the formulation of multi-agent search and in detail two-agent search.
- Students should b familiar with game trees.
- Given a problem description, the student should be able to formulate it in terms of a two-agent search problem.
- The student should be familiar with the minimax algorithms, and should be able to code the algorithm.
- Students should understand heuristic scoring functions and standard strategies for generating heuristic scores.
- Students should understand alpha-beta pruning algorithm, specifically its
    - Computational advantage
    - Optimal node ordering
- Several advanced heuristics used in modern game playing systems like detection of quiescent states, lengthening should be understood.
- A chess playing program will be analyzed in detail.

At the end of this lesson the student should be able to do the following:

- Analyze a given problem and formulate it as a two-agent search problem
- Given a problem, apply possible strategies for two-agent search to design a problem solving agent.

Lesson 7

Adversarial Search

3.2 Adversarial Search

We will set up a framework for formulating a multi-person game as a search problem. We will consider games in which the players alternate making moves and try respectively to maximize and minimize a scoring function (also called utility function). To simplify things a bit, we will only consider games with the following two properties:

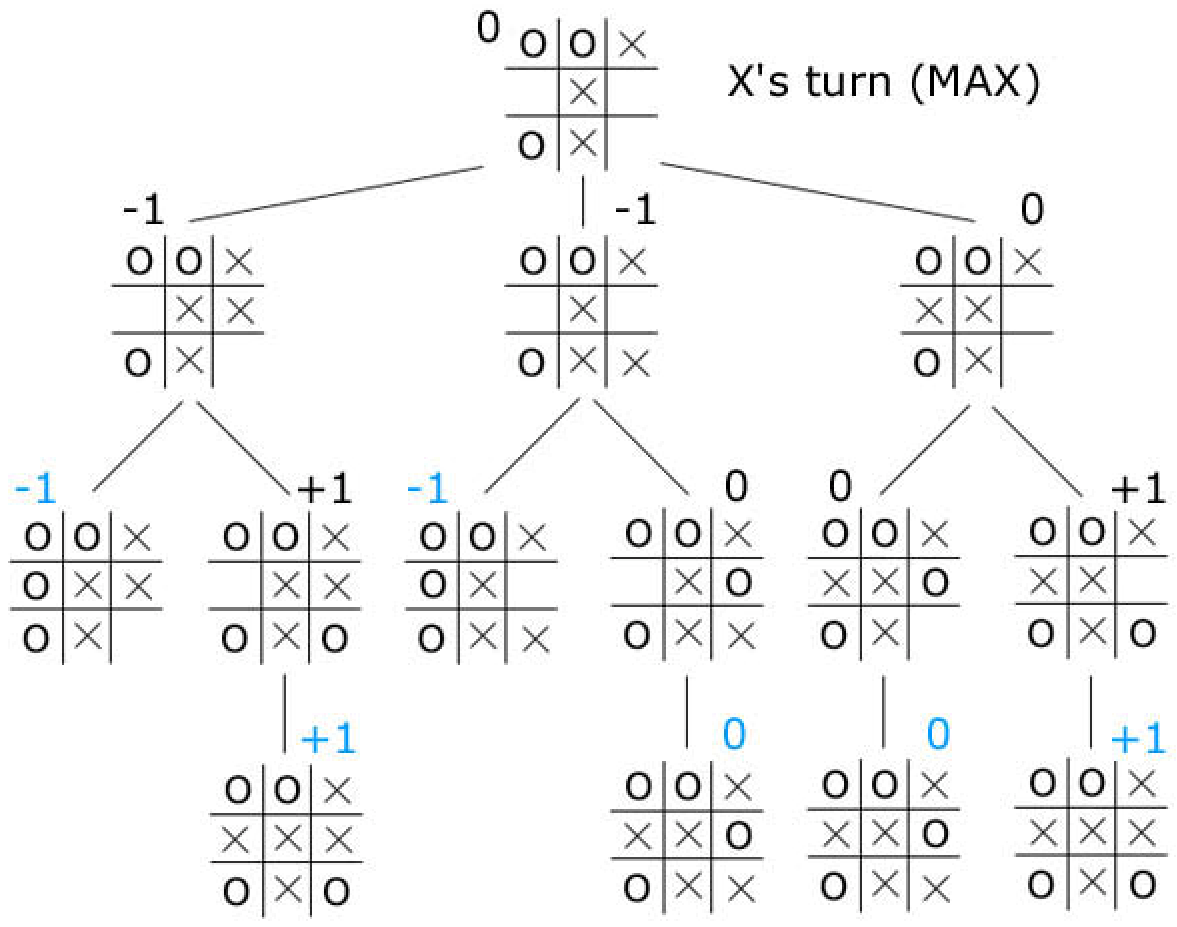
- Two player - we do not deal with coalitions, etc.
- Zero sum - one player's win is the other's loss; there are no cooperative victories

We also consider only perfect information games.

3.3 Game Trees

The above category of games can be represented as a tree where the nodes represent the current state of the game and the arcs represent the moves. The game tree consists of all possible moves for the current players starting at the root and all possible moves for the next player as the children of these nodes, and so forth, as far into the future of the game as desired. Each individual move by one player is called a "ply". The leaves of the game tree represent terminal positions as one where the outcome of the game is clear (a win, a loss, a draw, a payoff). Each terminal position has a score. High scores are good for one of the player, called the MAX player. The other player, called MIN player, tries to minimize the score. For example, we may associate 1 with a win, 0 with a draw and -1 with a loss for MAX.

Example : Game of Tic-Tac-Toe



Above is a section of a game tree for tic tac toe. Each node represents a board position, and the children of each node are the legal moves from that position. To score each position, we will give each position which is favorable for player 1 a positive number (the more positive, the more favorable). Similarly, we will give each position which is favorable for player 2 a negative number (the more negative, the more favorable). In our tic tac toe example, player 1 is 'X', player 2 is 'O', and the only three scores we will have are +1 for a win by 'X', -1 for a win by 'O', and 0 for a draw. Note here that the blue scores are the only ones that can be computed by looking at the current position.

3.4 Minimax Algorithm

Now that we have a way of representing the game in our program, how do we compute our optimal move? We will assume that the opponent is rational; that is, the opponent can compute moves just as well as we can, and the opponent will always choose the optimal move with the assumption that we, too, will play perfectly. One algorithm for computing the best move is the minimax algorithm:

```
minimax(player,board)
    if(game over in current board position)
        return winner
    children = all legal moves for player from this board
    if(max's turn)
        return maximal score of calling minimax on all the children
    else (min's turn)
        return minimal score of calling minimax on all the children
```

If the game is over in the given position, then there is nothing to compute; minimax will simply return the score of the board. Otherwise, minimax will go through each possible child, and (by recursively calling itself) evaluate each possible move. Then, the best possible move will be chosen, where ‘best’ is the move leading to the board with the most positive score for player 1, and the board with the most negative score for player 2.

How long does this algorithm take? For a simple game like tic tac toe, not too long - it is certainly possible to search all possible positions. For a game like Chess or Go however, the running time is prohibitively expensive. In fact, to completely search either of these games, we would first need to develop interstellar travel, as by the time we finish analyzing a move the sun will have gone nova and the earth will no longer exist. Therefore, all real computer games will search, not to the end of the game, but only a few moves ahead. Of course, now the program must determine whether a certain board position is 'good' or 'bad' for a certainly player. This is often done using an *evaluation function*. This function is the key to a strong computer game. The depth bound search may stop just as things get interesting (e.g. in the middle of a piece exchange in chess. For this reason, the depth bound is usually extended to the end of an exchange to an quiescent state. The search may also tend to postpone bad news until after the depth bound leading to the *horizon effect*.