**Tic Tac Toe Game**

I have choosen MiniMax algorithm with Alpha Beta Pruning for implementing Tic Tac Toe game. Following is general Minimax algorithm with Alpha Beta Pruning.

**function** alphabeta(node, depth, α, β, maximizingPlayer)

**if** depth = 0 **or** node is a terminal node

**return** the heuristic value of node

**if** maximizingPlayer

**for each** child of node

α := max(α, alphabeta(child, depth - 1, α, β, FALSE))

**if** β ≤ α

**break** *(\* β cut-off \*)*

**return** α

**else**

**for each** child of node

β := min(β, alphabeta(child, depth - 1, α, β, TRUE))

**if** β ≤ α

**break** *(\* α cut-off \*)*

**return** β

*(\* Initial call \*)*

alphabeta(origin, depth, -[∞](http://en.wikipedia.org/wiki/Infinity), +[∞](http://en.wikipedia.org/wiki/Infinity), TRUE)

Source : Wikipedia

Outline:

1. Each player minimizes the maximum payoff possible for the other—since the game is zero-sum, he also minimizes his own maximum loss (i.e. maximize his minimum payoff)
2. The AI tries to maximise the score possible (1) while assuming his opponent plays his best move achieving best score of -1. When there is no result possible the score is 0.
3. Alpha-Beta Pruning tries to decrease the number of nodes that are evaluated by the [minimax algorithm](http://en.wikipedia.org/wiki/Minimax" \l "Minimax_algorithm_with_alternate_moves" \o "Minimax) in its [search tree](http://en.wikipedia.org/wiki/Game_tree). It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
4. Applying Alpha Beta Pruning to Tic Tac Toe game reduces number of states to explore further , when we calculate beta score, less than or equal to alpha i.e. we already have a move for AI which has higher probability of AI win or draw (or lesser probability of AI loosing)
5. If player A *can* win in one move, his best move is that winning move. If player B knows that one move will lead to the situation where player A *can* win in one move, while another move will lead to the situation where player A can, at best, draw, then player B's best move is the one leading to a draw. Late in the game, it's easy to see what the "best" move is. The Minimax algorithm helps find the best move, by working backwards from the end of the game. At each step it assumes that player A is trying to **maximize** the chances of A winning, while on the next turn player B is trying to **minimize** the chances of A winning (i.e., to maximize B's own chances of winning).

Reason for choosing Minimax algorithm with alpha beta pruning:

1. In Tic Tac Toe game, the goal is to achieve best score for each move i.e. choose best possible move that will result in player win. Minimax algorithm is to minimise the maximum loss and also minimizing the maximum payoff possible for other player.
2. A minimax algorithm is a recursive [algorithm](http://en.wikipedia.org/wiki/Algorithm) for choosing the next move in a two-player game usually. A value is associated with each position or state of the game. This value is computed by means of a [position evaluation function](http://en.wikipedia.org/wiki/Evaluation_function) and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. So this algorithm is best suited for Tic Tac Toe game where AI tries to achive best score (win) by minimizing its loss (draw or human win).
3. Alpha–beta pruning is a [search algorithm](http://en.wikipedia.org/wiki/Search_algorithm) that seeks to decrease the number of nodes that are evaluated by the [minimax algorithm](http://en.wikipedia.org/wiki/Minimax" \l "Minimax_algorithm_with_alternate_moves" \o "Minimax) in its [search tree](http://en.wikipedia.org/wiki/Game_tree). It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
4. With an (average or constant) [branching factor](http://en.wikipedia.org/wiki/Branching_factor) of b, and a search depth of d [plies](http://en.wikipedia.org/wiki/Ply_(game_theory)), the maximum number of leaf node positions evaluated (when the move ordering is [pessimal](http://en.wiktionary.org/wiki/pessimal" \o "wiktionary:pessimal)) is [O](http://en.wikipedia.org/wiki/Big_O_notation)(b\*b\*...\*b) = O(bd) – the same as a simple minimax search. If the move ordering for the search is optimal (meaning the best moves are always searched first), the number of leaf node positions evaluated is about O(b\*1\*b\*1\*...\*b) for odd depth and O(b\*1\*b\*1\*...\*1) for even depth, or O(b^{{d/2}})=O({\sqrt  {b^{d}}})

**Generalization for Minimax algorithm:**

Minimax (sometimes MinMax or MM[[1]](http://en.wikipedia.org/wiki/Minimax#cite_note-1)) is a decision rule used in [decision theory](http://en.wikipedia.org/wiki/Decision_theory), [game theory](http://en.wikipedia.org/wiki/Game_theory), [statistics](http://en.wikipedia.org/wiki/Statistics) and [philosophy](http://en.wikipedia.org/wiki/Philosophy) for minimizing the possible [loss](http://en.wikipedia.org/wiki/Loss_function) for a worst case (maximum loss) scenario.

**Minimax algorithm is suitable for following type of environment:**

1. Fully Observable
2. Strategic
3. Static
4. Sequential
5. Discrete
6. Multi-agent (considering human as an agent)