Name: Kenny Patel
Roll No. 19074010

Tic-Tac-Toe Lab Assignment 1

This is an implementation of the "Human vs Computer" game of Tic-Tac-Toe.

Here is the brief of all the four approaches, their advantages and disadvantages along with some observations.

Solution 1

In this Solution, I have started with a completely naive approach. A 9-element vector represents the board and the positions are numbered from 1 to 9. 0 represents blank, 1 for 'X' and 2 for 'O'. A large vector consisting of 3^9 vector elements is created(known as Movetable) which consists of all board positions.

Using the current board state specified by the ternary nine-digit number, it is converted to Decimal number. This decimal number is used as an index into the movetable and vector stored there is accessed. This accessed vector represents the correct move. It specifies what the board will look after the move. The board is then set equal to this vector.

Advantages:

1) This approach is efficient in time.
2) optimal in theory

Disadvantages:

1) Lots of storage space taken
2) Specifying all the 3^9 entries is tedious and time taking as well as error prone
3) To extend to 3 dimensions, we need to start from scratch and need too much space. So rendering is infeasible.

Solution 2

This approach is quite efficient in terms of space than the first one but less efficient in time. The game board consists of nine elements positioned from 1 to 9. Value 2: blank

Value 3: X

Value 4: O

Initially the board consists of 2's with the first element being blank so as to start indexing from one onwards.

Make 2 tries to make 2 in a row. It first tries to play in the centre and if it is already occupied, then tries non corner squares. Posswin() checks that if player p cannot win in the next move, it returns zero else returns the number of squares that constitutes a winning move.

Initially we call posswin(us), if we can win we make the winning move. If we cannot win we call posswin(opponent) and block opponents move if he/she can win. posswin checks each row, column, and diagonal as follows:

If (product == 18) X can win ((3 * 3 * 2) = 18) If (product == 50) O can win ((5 * 5 * 2) = 50) Hence, it Scans row or column or diagonal to find blank space to move to.

Go(n) function make a move to square n

For eg., it goes like this:

turn = 1: Go(1) starts from upper left corner

turn = 2: checks if board[5] is blank or not, if blank, Go(5) else Go(1)

turn = 3: checks if board[9] is blank or not, if yes, Go(9) else Go(3)

turn = 4: if posswin(X) is not equal to zero, block opponent else go(make2) and so on...

Advantages:

1) This algorithm is efficient in space as compared to the first algo which consumes lots of space
2) Time takes is not that long
3) It's easy to understand this strategy and also at any time, we can change it too.

Disadvantages:

1) Not efficient in time
2) Total strategy is already figured out by the programmer
3) Not generalizable to three dimensions

Solution 3

This is a more efficient approach to Solution 2, the difference being that the board will be a magic square whose all the rows, columns and diagonals add up to 15. To determine if a player has won, we have to find a combination of players' numbers from the magic square where the sum equals 15.

Observations:

1) Row scan is easier for humans.
2) Take into consideration how humans approach a problem and how computers do the same, the conclusion being that the human brain works in parallel and can see the complete board and all the possible moves at the same time whereas computers consider a move one at a time.
3) More efficient than solution 2 because more logic is poured in this approach.

Solution 4

This solution uses *Minimax algorithm* applied with Alpha-Beta Pruning. Minimax is a kind of 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. Alpha-Beta Pruning is an optimization technique for the Minimax algorithm.

In Minimax, the two players are called: Minimizer and maximizer. Maximizer tries to get the highest score possible while the minimizer tries to get the lowest score possible.
Alpha is the best value that the maximizer currently can guarantee at that level or above.
Beta is the best value that the minimizer currently can guarantee at that level or above.

The initial value of alpha here is -10 and the value of beta is +10. These values are passed down to subsequent nodes in a tree. At the root level, the maximizer must choose max of its child nodes. Now that child node is the minimizer and hence will choose minimum value from its child nodes.

It can happen that when choosing a certain branch of a certain node the minimum score that the minimizing player is assured of becomes less than the maximum score that the maximizing player is assured of (beta <= alpha). If this is the case, the parent node should not choose this node, because it will make the score for the parent node worse. Therefore, the other branches of the node do not have to be explored.

Advantages:

Allows elimination of search tree branches.
Memory wise, highly efficient.
Reduces search time and compilation during minimax approach.
Prevents the use of additional computation time, making the process fast.

Disadvantages:

Though designed to calculate a good move, it also calculates the value of all possible moves.