Implementation of Minimax Alpha Beta Pruning Search using Tic-Tac-Toe as an example domain

Minimax is an algorithm for decision rule used in the decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario for any player. Originally formulated for two-player game theory, covering both the cases where players take alternate, it has also been extended to more complex games and to general decision-making in the presence of uncertainty about the future possibilities.

The Minimax algorithm is an algorithm for determining the optimal game strategy for finite two-person zero-sum games with perfect information about each and every step. These games are especially board games like Chess, tic tac toe in which both players always know the entire history of the game. Even for games with random influence as Backgammon allows the Minimax algorithm based on expected values ​​expand. Typically, but not exclusively, the minimax algorithm is applied to games with alternate train right.

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree by pruning all the unwanted sub trees. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, etc). 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 by proof. 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 thus saving time and space.

The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. This way, the search time can be limited to the more promising subtree, and a deeper search can be performed in the same time. Like its predecessor, it belongs to the branch and bound class of algorithms. The optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node).

With an (average or constant) branching factor of b, and a search depth of d plies, the maximum number of leaf node positions evaluated is O(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}). In the latter case, where the ply of a search is even, the effective branching factor is reduced to its square root, or, equivalently, the search can go twice as deep with the same amount of computation. The explanation of b\*1\*b\*1\*... is that all the first player's moves must be studied to find the best one, but for each, only the best second player's move is needed to refute all but the first (and best) first player move—alpha–beta ensures no other second player moves need be considered. When nodes are ordered at random, the average number of nodes evaluated is roughly O(b^{3d/4}).

Normally during alpha–beta, the subtrees are temporarily dominated by either a first player advantage. This advantage can switch sides many times during the search if the move ordering is incorrect, each time leading to inefficiency. As the number of positions searched decreases exponentially each move nearer the current position, it is worth spending considerable effort on sorting early moves. An improved sort at any depth will exponentially reduce the total number of positions searched, but sorting all positions at depths near the root node is relatively cheap as there are so few of them. In practice, the move ordering is often determined by the results of earlier, smaller searches, such as through iterative deepening.

The algorithm maintains two values, alpha and beta, which represent the maximum score that the maximizing player is assured of and the minimum score that the minimizing player is assured of respectively. Initially alpha is negative infinity and beta is positive infinity, i.e. both players start with their lowest possible score. 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.

A calculated with the Minimax algorithm strategy is minimax strategy called. It assures the player concerned the maximum commercial opportunities which can be achieved regardless of the playing style of the opponent. This is formed from the minimax strategies of both players strategy pair forms a Nash equilibrium .

For non-zero-sum games, in which the defeat of the enemy coincides not necessarily with their own profit, the Minimax algorithm does not necessarily provide an optimal strategy.

Variants of the Minimax algorithm form the core element of gambling programs like a chess program . The increasing processing power of computers has now meant that even in such complex games like chess, most people can be beaten by the computer without any trouble now.

In game theory, a game tree is a directed graph whose nodes are positions in a game and whose edges are moves. The complete game tree for a game is the game tree starting at the initial position and containing all possible moves from each position; the complete tree is the same tree as that obtained from the extensive-form game representation.

Initially, in our program we give a choice to the player to choose whom to start the game first. The computer takes the part of O and the player X.

-----Welcome to Tic Tac Toe game using minimax Aplha Beta pruning:------

You : X

Computer : O

Select the options:

1.You will start.

2.I will start

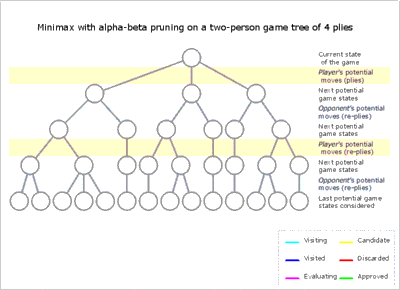
After giving the input "1" it asks for the coordinates from the player.

1

Enter the coordinates where you want to place X:

1 1

Then this move is placed on the board and the "applyMinimax Algorithm()" is called which creates the child nodes and displays it as it traverses through the tree.



The picture shows a simple tree with search depth 4. Player A is on the train.

The nodes of the levels 0 and 2 correspond to game situations where player A at the train. Here, the evaluation function of the child nodes of the budget for player A train will each maximized d. H. Selected and assigned the value of the parent node.

The nodes of level 1 and 3 correspond to game situations where player B on the train. Here, the evaluation function of the child nodes of the cheapest for player B train is respectively minimized, d. H. Selected and assigned the value of the parent node.

The algorithm starts at the bottom with the leaves, and then goes up to the hilt. In Level 3, the algorithm selects the minimum value of child nodes and assigns the parent node to (it is minimized). In Level 2 the largest respective child node is then the parent node assigned (it is maximized). This is performed alternately until the root is reached. The root is assigned the value of the biggest child node. It then is the train, which is to be played.

Board:

-------

|X|X|O|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|X|O|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|X|O|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|X|-|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|X|-|

|O|X|-|

|-|O|O|

-------

Board:

-------

|X|X|-|

|O|X|-|

|-|O|O|

-------

Board:

-------

|X|X|-|

|O|X|-|

|-|O|O|

-------

Board:

-------

|X|X|-|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|X|-|

|-|X|O|

|-|O|O|

-------

Board:

-------

|X|X|-|

|-|X|O|

|-|O|O|

-------

Board:

-------

|X|X|-|

|-|X|O|

|-|O|O|

-------

Board:

-------

|X|X|-|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|X|-|

|-|X|-|

|O|O|O|

-------

Board:

-------

|X|X|-|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|-|-|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|-|-|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|-|-|

|-|X|-|

|-|O|O|

-------

Board:

-------

|X|-|-|

|-|X|-|

|-|-|O|

-------

Board:

-------

|-|-|-|

|-|X|-|

|-|-|O|

-------

Board:

-------

|-|-|-|

|-|X|-|

|-|-|O|

-------

Board:

-------

|-|-|-|

|-|X|-|

|-|-|O|

-------

Board:

-------

|-|-|-|

|-|X|-|

|-|-|-|

-------

The time taken for this step is:203

The number of child nodes for this particular user input is3075

The current board after placing computers' move at:(0 0)

Board:

-------

|O|-|-|

|-|X|-|

|-|-|-|

-------

Enter the coordinates where you want to place X:

Also, the time taken for the computer to create these child nodes are printed using the:

**long** startTime = System.*currentTimeMillis*();

*ApplyAlphaBetaMinimax*(b,Integer.***MIN\_VALUE***, Integer.***MAX\_VALUE***, 0, 1);

**long** finishTime = System.*currentTimeMillis*();

**long** elapsedTime = finishTime - startTime;

The minimax alpha beta pruning algorithm is :

**public** **static** **int** ApplyAlphaBetaMinimax(TicTacToeBoard b,**int** AplhaValue, **int** BetaValue, **int** depth, **int** turn)

{

**if**(BetaValue<=AplhaValue)

{

**if**(turn == 1)

**return** Integer.***MAX\_VALUE***;

**else**

**return** Integer.***MIN\_VALUE***;

}

**int** flag=0;

**if**((depth == -1) || (flag=*HasAnybodyWon*(b))==0)

**return** *UtilityFunction*(b);

List<Coordinates> AvailableSpaces = **new** ArrayList<>();

AvailableSpaces=*FindEmptyCoordinates*(b);

**if**(AvailableSpaces.isEmpty())

**return** 0;

**if**(depth==0)

b.CoordinateClass.clear();

**int** HighestValue = Integer.***MIN\_VALUE***, LowestValue = Integer.***MAX\_VALUE***;

**for**(**int** i=0;i<AvailableSpaces.size(); i++)

{

Coordinates Coordinate = AvailableSpaces.get(i);

**int** Points = 0;

**if**(turn == 1)

{

b.Contents[Coordinate.x][Coordinate.y]=1;

Points = *ApplyAlphaBetaMinimax*(b,AplhaValue, BetaValue, depth+1, 2);

HighestValue = Math.*max*(HighestValue, Points);

**if**(HighestValue<Points)

{

HighestValue=Points;

}

**if**(AplhaValue<Points)

{

AplhaValue=Points;

}

**if**(depth == 0)

b.CoordinateClass.add(**new** CoordinateClass(Coordinate,Points));

++*Count*;

*DisplayBoard*(b);

}

**else** **if**(turn == 2)

{

b.Contents[Coordinate.x][Coordinate.y]=2;

Points = *ApplyAlphaBetaMinimax*(b,AplhaValue, BetaValue, depth+1, 1);

**if**(LowestValue>Points)

{

LowestValue=Points;

}

**if**(BetaValue>Points)

{

BetaValue=Points;

}

}

b.Contents[Coordinate.x][Coordinate.y] = 0;

++*Count*;

*DisplayBoard*(b);

**if**(Points == Integer.***MAX\_VALUE*** || Points == Integer.***MIN\_VALUE***)

**break**;

}

**if**(turn==1)

{

**return** HighestValue;

}

**return** LowestValue;

}

This makes use of various functions to check whether the current board position is won by anubody or not by checking diagonally, ertically and horizontaly.

**public** **static** **int** HasAnybodyWon(TicTacToeBoard b)

{

**if**(*HasComputerWon*(b))

**return** 0;

**if**(*HasHumanWon*(b))

**return** 0;

**if**(*HasDraw*(b))

**return** 0;

**return** -1;

}

**public** **static** **boolean** HasComputerWon(TicTacToeBoard b)

{

**for**(**int** i=0;i<3;i++)

**for**(**int** j=0;j<3;j++)

**if** ((b.Contents[0][0] == b.Contents[1][1] && b.Contents[0][0] == b.Contents[2][2] && b.Contents[0][0] == 1) || (b.Contents[0][2] == b.Contents[1][1] && b.Contents[0][2] == b.Contents[2][0] && b.Contents[0][2] == 1))

{

**return** **true**;

}

**for** (**int** i = 0; i < 3; ++i)

{

**if** (((b.Contents[i][0] == b.Contents[i][1] && b.Contents[i][0] == b.Contents[i][2] && b.Contents[i][0] == 1)|| (b.Contents[0][i] == b.Contents[1][i] && b.Contents[0][i] == b.Contents[2][i] && b.Contents[0][i] == 1)))

{

**return** **true**;

}

}

**return** **false**;

}

**public** **static** **boolean** HasHumanWon(TicTacToeBoard b)

{

**for**(**int** i=0;i<3;i++)

**for**(**int** j=0;j<3;j++)

**if** ((b.Contents[0][0] == b.Contents[1][1] && b.Contents[0][0] == b.Contents[2][2] && b.Contents[0][0] == 2)|| (b.Contents[0][2] == b.Contents[1][1] && b.Contents[0][2] == b.Contents[2][0] && b.Contents[0][2] == 2))

{

**return** **true**;

}

**for** (**int** i = 0; i < 3; ++i)

{

**if** (((b.Contents[i][0] == b.Contents[i][1] && b.Contents[i][0] == b.Contents[i][2] && b.Contents[i][0] == 2)|| (b.Contents[0][i] == b.Contents[1][i] && b.Contents[0][i] == b.Contents[2][i] && b.Contents[0][i] == 2)))

{

**return** **true**;

}

}

**return** **false**;

}

**public** **static** **boolean** HasDraw(TicTacToeBoard b)

{

**for**(**int** i=0;i<3;i++)

**for**(**int** j=0;j<3;j++)

{

**if**(b.Contents[i][j]==0)

**return** **false**;

}

**return** **true**;

}

An ideal evaluation function assigns a position the value +1 to when player A wins, and the value -1 if player B wins, and 0 in a draw. Can one of all game positions the search tree up to the maximum depth to build (up to the end position, where you can see who wins), the algorithm makes a perfect match. However, in practice the complete structure of a search tree only for very simple games like is Tic-Tac-Toe possible.

In almost all other games, this is too computationally expensive. Therefore, one is content to construct the search tree only to a search level (horizon). The evaluation function is modified, very good match for positions A obtained very high values, very good match for positions B get very low scores. To determine the values ​​one uses heuristics to estimate.

In our program we make use of an utility function which takes the tic tac toe board and returns the points acquired by mevalauting the board positions of both the players.

**public** **static** **int** UtilityFunction(TicTacToeBoard b)

{

**int** UtilityValue = 0;

**for** (**int** i = 0; i < 3; i++)

{

**int** dash = 0,X=0,O=0;

**for** (**int** j = 0; j < 3; j++)

{

**if** (b.Contents[i][j] == '-')

{

dash++;

}

**else** **if** (b.Contents[i][j] == 1)

{

X++;

}

**else**

{

O++;

}

}

UtilityValue=UtilityValue+*PriorityExchange*(X, O);

}

**for** (**int** j = 0; j < 3; j++)

{

**int** DashCount = 0;

**int** XCount = 0;

**int** OCount = 0;

**for** (**int** i = 0; i < 3; i++)

{

**if** (b.Contents[i][j] == '-')

{

DashCount++;

}

**else** **if** (b.Contents[i][j] == 1)

{

XCount++;

}

**else**

{

OCount++;

}

}

UtilityValue+=*PriorityExchange*(XCount, OCount);

}

**int** DashCount=0;

**int** XCount = 0;

**int** OCount = 0;

**for**(**int** i=0;i<3;i++)

**for**(**int** j=0;j<3;j++)

{

**if** (b.Contents[i][j] == 1)

{

XCount++;

}

**else** **if** (b.Contents[i][j] == 2)

{

OCount++;

}

**else**

{

DashCount++;

}

}

UtilityValue=UtilityValue+*PriorityExchange*(XCount, OCount);

DashCount=XCount=OCount=0;

**for** (**int** i = 2, j = 0; i > -1; --i, j++)

{

**if** (b.Contents[i][j] == 1)

{

XCount++;

}

**else** **if** (b.Contents[i][j] == 2)

{

OCount++;

}

**else**

{

DashCount++;

}

}

UtilityValue+=*PriorityExchange*(XCount, OCount);

**return** UtilityValue;

}

The above function makes use of Priority exchange function to calculate the points:

**public** **static** **int** PriorityExchange(**int** X, **int** O)

{

**int** c;

**if** (X == 3)

{

c = 100;

}

**else** **if** (X == 2 && O == 0)

{

c = 10;

}

**else** **if** (X == 1 && O == 0)

{

c = 1;

}

**else** **if** (O == 3)

{

c = -100;

}

**else** **if** (O == 2 && X == 0)

{

c = -10;

}

**else** **if** (O == 1 && X == 0)

{

c = -1;

}

**else**

{

c = 0;

}

**return** c;

}

All the scores calculated will be stored in an object of class "b.CoordinateClass" along with the coordinates on the tic tac toe board. It takes the coordinate which will minimize the players' points and places its move at that particular point. The computer tries ti minimize the players' points and the player inturn will be trying to maximize his points.

**for** (**int** i=0;i<b.CoordinateClass.size(); i++)

{

**if** (n < b.CoordinateClass.get(i).Score)

{

n=b.CoordinateClass.get(i).Score;

efficient=i;

}

}

b.Contents[NewC.x][NewC.y]=1;

System.***out***.println("\nThe current board after placing computers' move at:("+NewC.x+" "+NewC.y+")");

*DisplayBoard*(b);

In this way the game moves on until all the positions on the board are filled resulting in either a win for the computer or numan,or a draw match.

The time taken for this step is:0

The number of child nodes for this particular user input is3

The current board after placing computers' move at:(0 1)

Board:

-------

|O|O|O|

|X|X|O|

|X|-|X|

-------

Yayy..!!! I won.

Wanna play another game???

1. Yes

2. No

Board:

-------

|O|X|O|

|O|X|X|

|X|O|X|

-------

The time taken for this step is:0

The number of child nodes for this particular user input is0

Match is a draw! Nobody wins...

Wanna play another game???

1. Yes

2. No

The corresponding code for the above mechanism is:

**if** (*HasComputerWon*(b))

{

System.***out***.println("Yayy..!!! I won.");

}

**else** **if** (*HasHumanWon*(b))

{

System.***out***.println("Winner:Human");

}

**else** **if**(*HasDraw*(b))

{

System.***out***.println("Match is a draw! Nobody wins... ");

}

}

By comparing the milliseconds taken by the minimax and this algorithm, it can be concluded that pruning is better. And also, the number of nodes that are created is very much lesser than the minimax.