Project Report

On

Applications of Artificial Intelligence and Machine Learning in Othello

Submitted by:

B.Snehashriie (180001013)

M.Uday Kumar Reddy (180001030)

Computer Science and Engineering

2nd year

Under the Guidance of

Dr Kapil Ahuja

Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2020

Contents:

Overview :

As a part of our project, we have decided to develop an AI-based algorithm which enables the computer to find an optimal move. We analyse the performance of various search algorithms involved in the main algorithm, including naive min-max and min-max with alpha-beta pruning. ML algorithms to train the evaluation function by automatically optimizing the relative feature weights are also discussed.

Introduction :

Othello, a game that according to many takes a minute to learn and a lifetime to master. It is a strategy board game played between 2 players. One player plays black and the other white.
Black always starts the game. Then the game alternates between white and black until:

- one player can not make a valid move to outflank the opponent.
- both players have no valid moves.

A valid move is a move made by placing a disc of the player's colour on the board in a position that "out-flanks" one or more of the opponent's discs.

A disc or row of discs is outflanked when it is surrounded *at the ends* by discs of the opposite colour.

A disc may outflank any number of discs in one or more rows in any direction (horizontal, vertical, diagonal).

All the discs which are outflanked will be flipped. Discs may only be outflanked as a direct result of a move and must fall in the direct line of the disc being played.

If you can't outflank and flip at least one opposing disc, you must pass your turn. However, if a move is available to you, you can't forfeit your turn.

Once a disc has been placed on a square, it can never be moved to another square later in the game.

When a player has no valid moves, he passes his turn and the opponent continues. The game ends when both players cannot make a valid move.

Objectives :

1. Explore Artificial Intelligence techniques in the board game Othello
2. Comparison of various search algorithms (naive minimax, minimax with alpha-beta pruning etc)
3. Application of ML in training the static evaluation function

AI Algorithm Analysis (Search algorithms):

The main part of the AI algorithm is the search algorithm.We analyse two of the search algorithms:

- Naive Minimax
- Minimax with alpha-beta pruning

Minimax Algo :

It finds the optimal move for a player, assuming that your opponent also plays optimally.

It recursively evaluates a position by taking the best of the values for each child position. The best value is the maximum for one player and the minimum for the other player because positions that are good for one player are bad for the other.

Pseudocode :

```
function minimax(board, depth, isMaximizingPlayer):
  if current board state is a terminal state :
      return value of the board

   if isMaximizingPlayer :
      bestVal = -INFINITY
      for each move in board :
         value = minimax(board, depth+1, false)
         bestVal = max( bestVal, value)
      return bestVal
```

```
else :
    bestVal = +INFINITY
    for each move in board :
        value = minimax(board, depth+1, true)
        bestVal = min( bestVal, value)
    return bestVal
```

Alpha-Beta Pruning (Dynamic Programming Technique) :

Minimax can be made more efficient by using the techniques of dynamic programming. Minimax with alpha-beta (α-β) pruning is a depth-first search algorithm that prunes by passing pruning information down in terms of parameters α and β.

In this depth-first search, a node has a "current" value, which has been obtained from some of its descendants. This current value can be updated as it gets more information about the value of its other descendants. It cuts off branches in the game tree which need not be searched because there already exists a better move available.

Pseudocode :

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
```

```
    value = minimax(node, depth+1, true, alpha, beta)
    bestVal = min( bestVal, value)
    beta = min( beta, bestVal)
    if beta <= alpha:
      break
  return bestVal
```

Complexity Analysis :

Minimax:
O(b^d)

All the nodes in the tree have to be generated once at some point, and the assumption is that it costs a constant time c to generate a node (constant times can vary, you can just pick c to be the highest constant time to generate any node). The order is determined by the algorithm and ensures that nodes don't have to be repeatedly expanded.

At the deepest level of the tree at depth d there will be b^dnodes, the work at that level therefor is c*b^d. The total amount of work done to this point is c*b^0 + c*b^1 + ... + c*b^d. For the complexity we only look at the fastest rising term and drop the constant so we get:

O(c + c*b + ... + c*b^d) = O(c*b^d) = O(b^d).

In essence: The time is a function f(d) = SUM(i=1..d){c*b^i}

and O(f(d)) = O(b^d).

Alpha-Beta Pruning:

O((b/n)^d) (assuming the search depth to be the same)

Where b is the average branching factor, d is the search depth and n is a constant >= 1 which depends on search depth, game state and the order of traversing the possible moves.
Alpha-beta search greatly reduces the number of nodes in the game tree that must be searched,especially in case of higher search depths.This can also be seen in the experimental results obtained over implementing the search algorithms for various search depths on two board states.

AI Algorithm Design:

AI-Based Algorithms to find Optimal move:

The basic idea is to consider the possible moves from the current game state, evaluate the position resulting from each move, and choose the one that appears best. The major component is the search algorithm, which more accurately evaluates a state by looking ahead at potential moves.

Main algorithms used

- Minimax
- Alpha-beta pruning

Supporting Functions

- Evaluate
- Possible moves
- Make move
- Best move
- Check winning

Static Evaluation Function : evaluate()

Based on certain parameters known as features, which are given corresponding weights based on the game position , we evaluate the child nodes and determine the game board state .

The features used here are:

1. Mobility:

   Mobility is a measure of the number of moves available to each player. The number of moves available to the player minus the number of moves available to the opponent determines mobility.

2. Stability :

   Pieces that are impossible to flip are called stable. These pieces are useful because they contribute directly to the final score.

   a. Corners: Corners are extremely valuable because corner pieces are immediately stable and can make adjacent pieces stable. They have high positive weight.
   b. X Squares: X-squares are the squares diagonally adjacent to corners. X-squares are highly undesirable when the adjacent corner is unoccupied because they make the corner vulnerable to being taken by the opponent, so they have a negative weight.
   c. C squares: C-squares are the squares adjacent to corners and on an edge. C-squares adjacent to an unoccupied corner are somewhat undesirable, and hence are given a negative weight. C-squares adjacent to an occupied corner are desirable, and hence are given a positive weight

Possible Moves: pm()

The array M[64] is initialised to '0'. The function pm() modifies M[] such that the indexes corresponding to the possible moves available to the present player are given certain values. These values (2,3,5,7) are dependent on the alignment in which the possible move exists.
Here, the alignment refers to the direction in which the coins flip once the player plays that move.
For eg: M[4]=2 implies putting a coin at fifth position will flip the coins of the opposite player horizontally.
M[4]=6 implies putting a coin at fifth position will flip the coins of the opposite player vertically and horizontally.

make_move():
This function flips all the coins in accordance with the M[i] that is chosen.

best_move():
Returns the index corresponding to the optimal move for the player.

Check Winning : check_winning()

If the current player does not have any possible moves, this function checks whether the next player has any moves or not.
If the next player has any possible moves, the current player's turn is skipped.
If the next player also does not have any possible moves, a winner is declared based on which player has more coins on the board.

Corner Cases:

There are two such cases we've implemented:

1) ***When one player's turn is skipped***
   Here, the current player does not have any valid moves. Therefore, his turn is skipped and the next player gets to play.
2) ***When the game ends before the board is completely filled***
   In this case, both the players do not have any valid moves left despite the board not being completely filled. Since neither of them can move. The game is terminated and the winner is declared based on the number of coins of each player. I.e. the player with more number of pieces is declared as the winner.

Design and Analysis of ML Algorithms:

In order to compare the various moves available for a player, the child nodes of the search tree are to be evaluated. The static evaluation does this job.

In order to evaluate the nodes accurately, the weights of various features are to be chosen carefully.Moreover these weights change along with the game state.

We develop an algorithm which estimates the appropriate weights at a particular game

state based on the results from the previous games.

Algorithms used

- Interpolation
- Gradient descent

Interpolation() :

Interpolation is a method or process by which we can develop a function from a discrete set of points.

Through this function we can obtain values for continuous arguments.

Lagrange's interpolation:

- A set of n data points {(ti,yi)}. 1<=i<=n.
- Lagrange's basis functions Lj(t) 1<=j<=n.

$$L_j(t) = \frac{\prod_{i=1, i\neq j}^{n} (t - t_i)}{\prod_{i=1, i\neq j}^{n} (t_j - t_i)}, \quad j = 1(1)n.$$

Polynomial obtained :

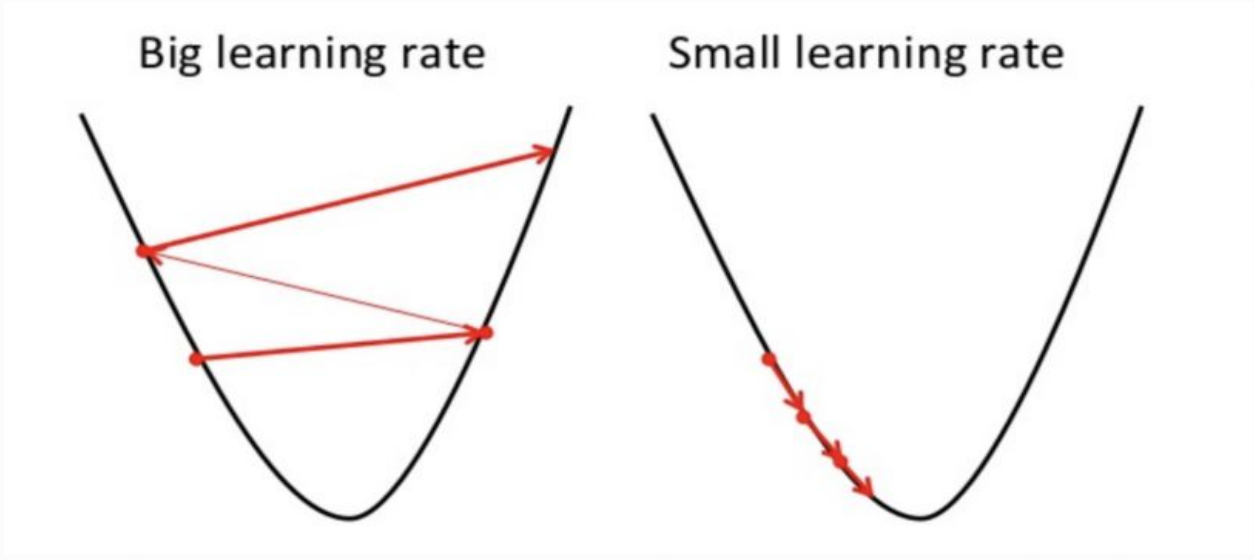
$$\mathbf{p(t) = y_1.L_1(t)+y_2.L_2(t)+y_3.L_3(t)+\ldots\ldots\ldots\ldots+y_n.L_n(t)}$$

Gradient descent():

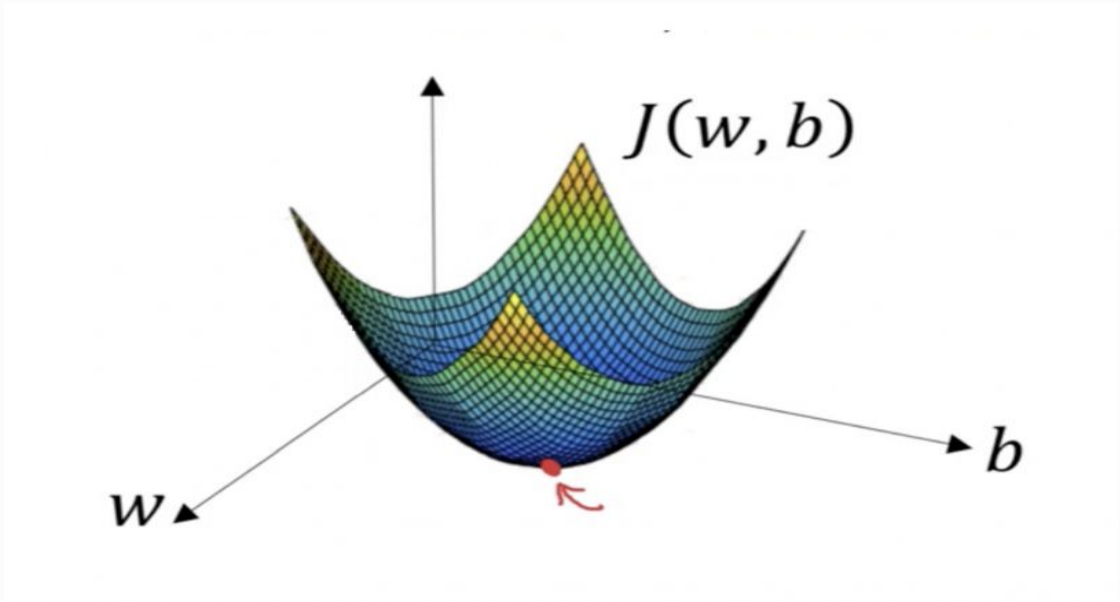
Gradient descent is an optimization algorithm that finds a local minimum of a function by taking steps in the direction of steepest descent, proportional to the negative gradient of the function.

As applied to static evaluation training, the function to minimize is a measure of the error for a given set of weights, based on the difference between the target value for each position and the value produced by the static evaluation function with the given weights. The gradient descent method starts with an arbitrary set of weights and then repeatedly takes steps in the direction that reduces error most until it reaches convergence at a local minimum.

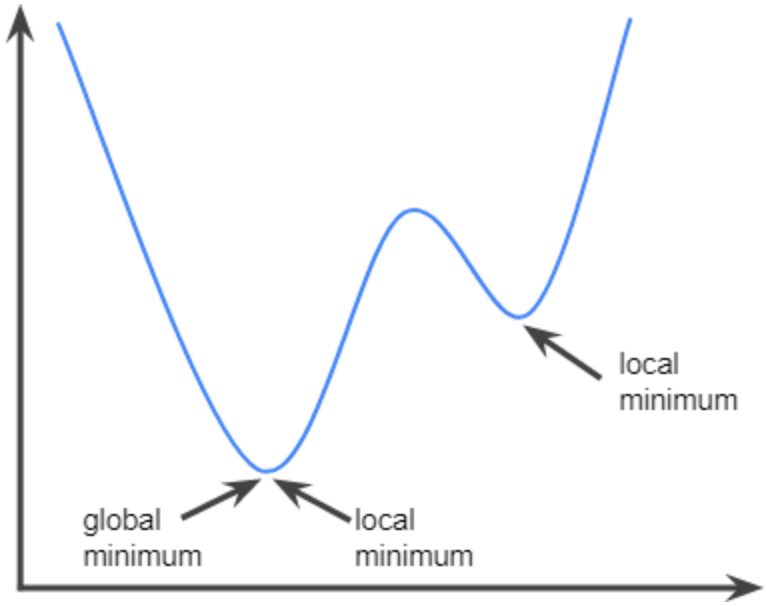
A basic form of gradient descent takes steps of size directly proportional to the magnitude of the gradient, with a fixed constant of proportionality called the learning rate. However, a learning rate that is too small may result in extremely slow convergence, while a learning rate that is too large may converge to a poor solution or fail to converge at all.



The level of accuracy that is achievable by this method, depends on the number of iterations and the learning rate.



We keep updating the parameters W and B till we reach at the global minimum.
This method suits best when the function that is to be minimised is a convex function.
In case of non convex functions we end up arriving at a local minima which might not be the global minimum (in this the initial values of the parameters chosen decides the point to which the function converges).



Pseudo code:

Gradient descent for m examples:
Let the input set be $\{(t_1,y_1),(t_2,y_2),.....(t_m,y_m)\}$.
Let the parameters be W and B.
Let the learning rate be alpha.
Initialize the parameters to 0.

For each iteration

Compute the loss function J(W,B) (square error in estimating the weights)
$J(W,B) = (1/m)\Sigma(W.x_i + B - y_i)^2$

Let the differential of the loss function wrt W and B be dw and db respectively.
W:= W - (alpha)*(dw);
B:=B-(alpha)*(db);

Implementation and Results:

ML Algorithms

Training Set:

a	b	c	d	e	f	g
5	7	3	0.27	-0.75	-1	1.5
9	7	3.1	0.277	-0.753	-1.09	1.5
13	6.75	3.2	0.286	-0.756	-1.1	8 1.5
16	6.7	3.3	0.30	-0.8	-1.22	1.75
19	6.6	3.4	0.31	-0.83	-1.27	1.75
25	6.35	3.7	0.329	-0.85	-1.36	1.75
29	6.2	4.3	0.339	-0.867	-1.39	1.75
32	6	4.6	0.357	-0.889	-1.45	2
36	5.7	5.4	0.380	-0.945	-1.57	2
39	5.3	6.3	0.40	-1.011	-1.68	2
45	4.3	7.4	0.63	-1.208	-1.99	2
48	3.4	8.3	0.701	-1.233	-2.13	2.5
52	3.2	8.9	0.725	-1.24	-2.21	2.5
55	3.1	9.4	0.74	-1.245	-2.25	2.5
57	3.05	10	0.75	-1.25	-2.25	2.7

a) *State (total no.of pieces)*
b) *Mobility*
c) *Corners*
d) *C with corners occupied*
e) *C corners unoccupied*
f) *X pieces with corners unoccupied*
g) *Pieces*

Denotations:

a. *error in mobility weight estimation = e_mob;*
b. *error in corners weight estimation = e_cor;*
c. *error in c squares with corners occupied weight estimation = e_c_occ;*
d. *error in c squares with corners unoccupied weight estimation = e_c_uno;*
e. *error in x squares with corners unoccupied weight estimation = e_x_uno;*
f. *error in piece weight estimation = e_p;*

Results(Accuracy):

Learning Rate : 0.0003						
No. of iterations	e_mob	e_cor	e_c_occ	e_c_uno	e_x_uno	e_p
1	0.275712	0.416967	0.00163419	0.000377976	0.00428231	0.00699192
12	0.796761	0.0104084	6.73529e-06	0.00431185	0.00627991	0.00783709
We see that the errors in estimated weights of mobility, c squares with corners unoccupied,x squares with corners unoccupied, pieces have increased with the number of iterations. This occurs due to high learning rate.The learning has to be reduced inorder to arrive at a minimum (local/global).						

Learning Rate : 0.0001						
No. of iterations	e_mob	e_cor	e_c_occ	e_c_uno	e_x_uno	e_p
1	0.122484	3.7503	0.0203275	0.0491271	0.167372	0.242685
12	0.719629	0.0284009	7.09384e-06	0.00305744	0.00379366	0.00456353
We see that the error in the estimated weight of mobility has increased with the number of iterations. Hence the learning rate has to be further reduced inorder to arrive at minimum(local/global)						

Learning Rate : 0.00005						
No. of iterations	e_mob	e_cor	e_c_occ	e_c_uno	e_x_uno	e_p
1	0.323467	5.10434	0.0282616	0.0740914	0.245054	0.353887
12	0.334138	0.322013	0.00117183	5.16522e-05	0.00198985	0.00346095
We see that the error in the estimated weight of mobility has increased with the number of iterations. The learning rate has to be further reduced inorder to arrive at a minimum value(global/local).						

Learning Rate : 0.00003						
No. of iterations	e_mob	e_cor	e_c_occ	e_c_uno	e_x_uno	e_p
1	0.430661	5.70428	0.0318004	0.0855082	0.280261	0.404223
12	0.0675182	1.07524	0.00509557	0.00651658	0.0287632	0.0430897
30	0.557747	0.101356	0.000210839	0.00101302	0.000471149	0.000403522
40	0.686204	0.0391766	2.49039e-05	0.00256926	0.00289055	0.00339149
We see that the error in the estimated weight of mobility has increased with the number of iterations (12,30,40). Hence the learning rate has to be further reduced inorder to arrive at a minimum value(local/global).						

Learning Rate : 0.00001						
No. of iterations	e_mob	e_cor	e_c_occ	e_c_uno	e_x_uno	e_p
1	0.553169	6.33754	0.035548	0.0977428	0.31783	0.457904
12	0.109874	3.64529	0.019716	0.0472495	0.161477	0.234236
40	0.0949561	0.93251	0.00432217	0.00483876	0.022687	0.0342243
60	0.300398	0.374122	0.00142349	0.000199084	0.00317801	0.00530465
80	0.475626	0.162197	0.000449307	0.000346448	5.7059e-07	4.41245e-05
We see that the error in weight of mobility increased with the number of iterations (40,60,80). The learning has to be further reduced in order to arrive at the minimum values(global/local).						

Learning Rate : 0.000005						
No. of iterations	e_mob	e_cor	e_c_occ	e_c_uno	e_x_uno	e_p
1	0.586189	6.50106	0.0365175	0.100929	0.327591	0.471848
12	0.294834	4.9341	0.0272597	0.0708868	0.235141	0.339708
40	0.0103876	2.46687	0.0129086	0.0270024	0.0971473	0.141889
80	0.0924547	0.944186	0.0043851	0.00497099	0.023173	0.0349345
We see that the error in estimated weight of mobility increased with the number of iterations. Hence the learning has to be further reduced inorder to arrive at a minimum value(local/global).						

Conclusion:

From the above results we can conclude that very low learning rates are to be considered in order to ensure that the loss function corresponding to each feature converges to a minimum value(local/global).

This can be resolved by considering different learning rates for each loss function as we can see the loss functions corresponding to all the features except mobility converge for learning rate<=0.0003.

But since the loss functions considered are not convex ,we cannot guarantee that the error or the loss function converges to its global minimum (depends on the initial values of the parameters).

Hence it's better to use interpolation technique to obtain the weights since it's more flexible (non linear) and the error or the loss function equals to zero(more accurate) .

AI Algorithms

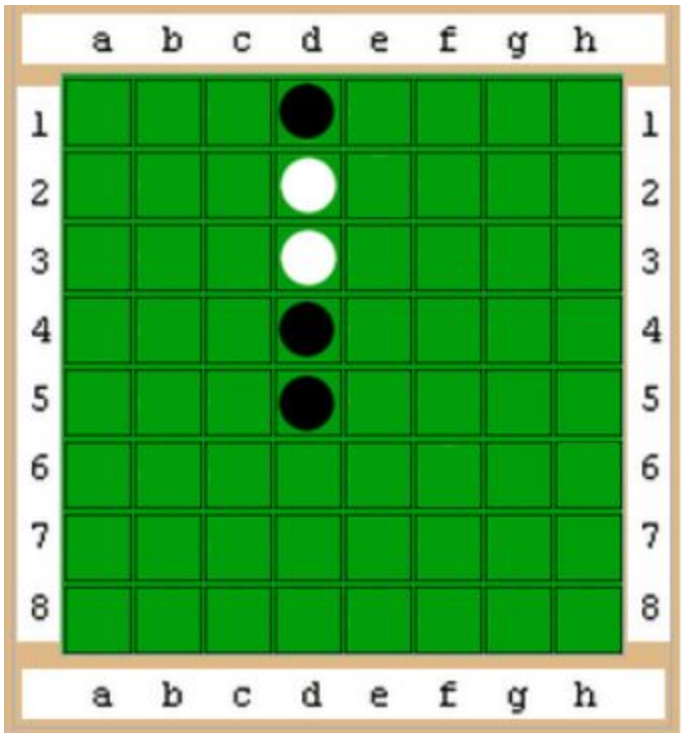
Interpolation technique is used for evaluating the weights of various features at various game states.

Corner Cases:

Case 1: No moves for the active player
In this case, the active player's turn is skipped and the game continues. The evaluation function has been designed such that these skips are also considered while evaluating a board state because if a player's turn is skipped it is beneficial for the opponent (locally i.e. greedy approach).

Generally, the bestmove function evaluates the moves of the active player
(assumed to be black) and does not operate when there are no moves left for the active player. In this case, the function assumes white (opponent player) to be the active player and finds the best move corresponding to it.Then, the game state obtained on making this bestmove is passed as an argument to the function which then evaluates the best move for black(the actual active player) in the final board state.

Example:



Output:

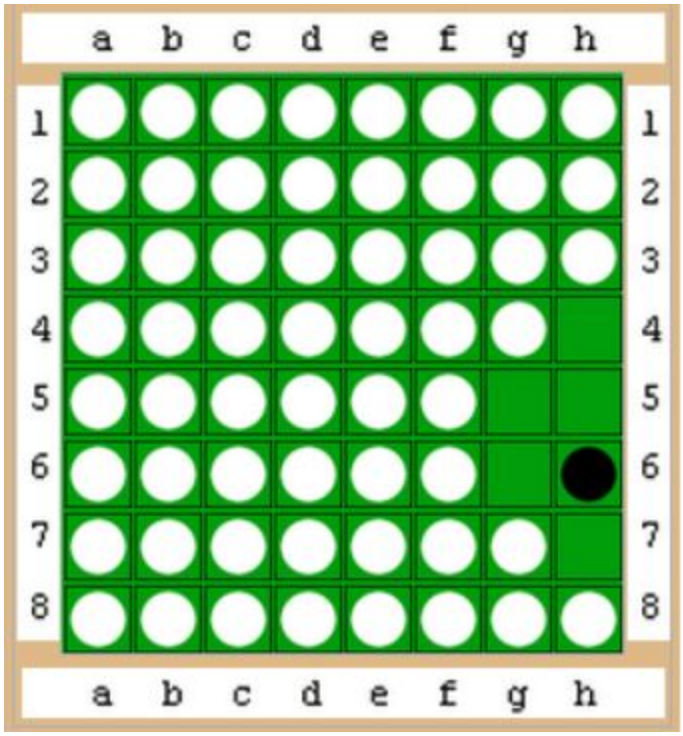
```
entered the if condition
10
00000000000000000001000000010000000100000001000000010000000000
depth :2
true returned 1000

hello
 whites best moves index : 19  move value : 1000
correspondingly,optimal move for black :
depth :1
true returned 1000
index :11  move_value :1000
best_value : 1000
corresponding index : 11

Time taken by program is : 0.000292 sec
total child nodes_: 2
```

Case 2 : the game ends before the board is completely filled

Example:



Output:

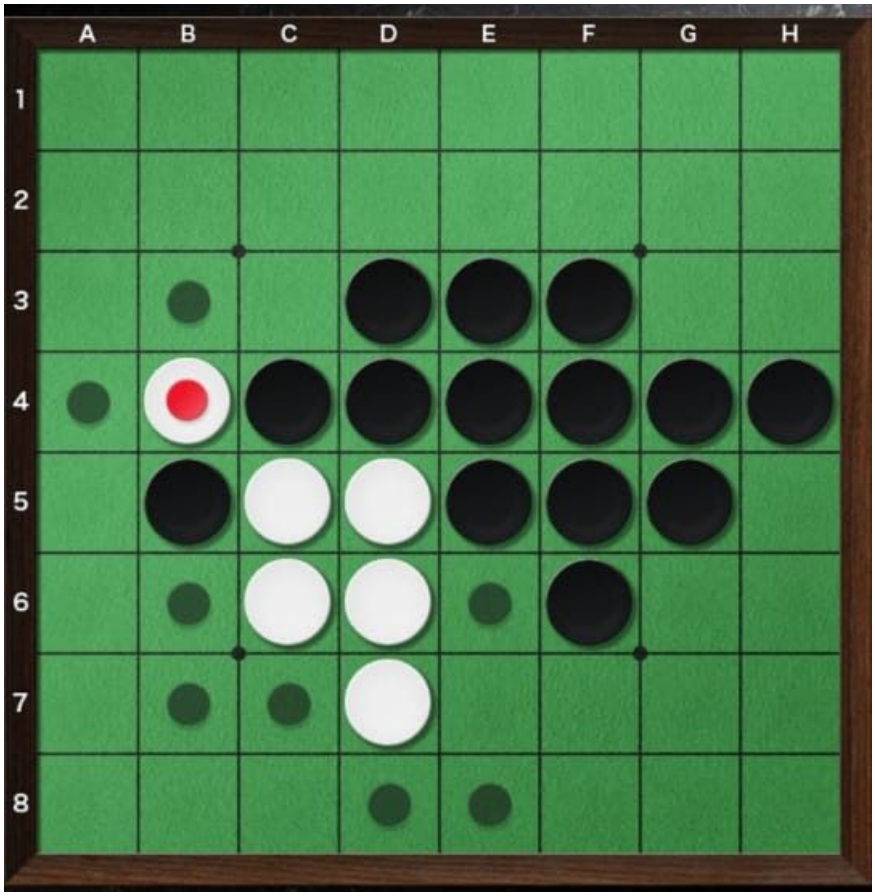
```
0000000000000000000000000000000000000000000000000000000000000000
entered the if condition
-1000
game over
white won
-1000

Time taken by program is : 0.000081 sec
total child nodes_: 0
```

Results:

Result 1:

Considered Board State

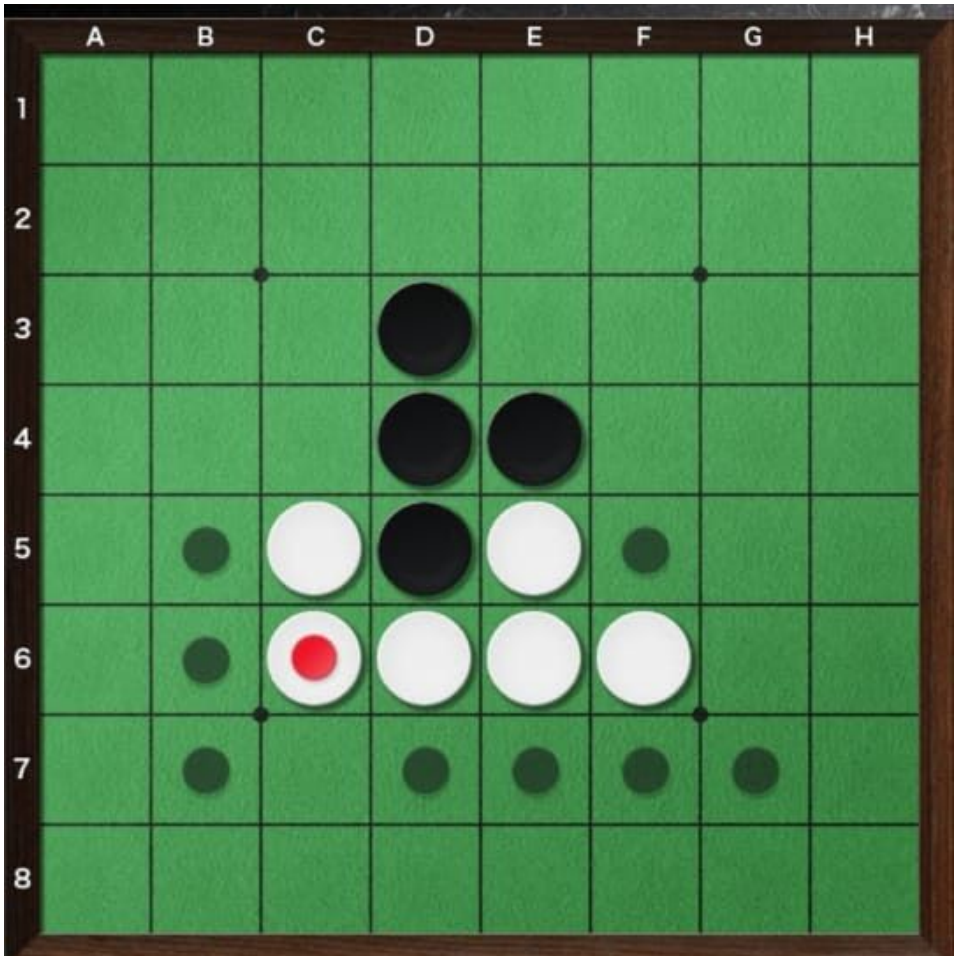


Time Taken (in sec)		
Search Depth	Naive	Alpha Beta
2	0.001044	0.000644
3	0.003479	0.002487
4	0.026015	0.006593
5	0.176897	0.044556
6	1.742132	0.061688

Number of Child nodes		
Search Depth	Naive	Alpha Beta
2	67	28
3	552	235
4	5601	983
5	51810	6121
6	616993	13756

Average Branching Factor			Ratio of Branching Factor
Search Depth	Naive	Alpha Beta	
2	8.185352	5.291502	1.547
3	8.20313185	6.17100579	1.329
4	8.65100161	5.59935985	1.545
5	8.76764018	5.71958443	1.533
6	9.22670650	4.89495489	1.885

Result 2: *Considered Board State*



Time Taken (in sec)		
Search Depth	Naive	Alpha Beta
2	0.000873	0.0006112
3	0.003926	0.0021402
4	0.0159218	0.0040167
5	0.100108	0.0174512
6	0.7351102	0.0471277

Number of Child nodes		
Search Depth	Naive	Alpha Beta
2	51	24
3	417	171
4	3286	525
5	28341	3437
6	251100	8887

Average Branching Factor			Ratio of Branching Factor
Search Depth	Naive	Alpha Beta	
2	7.14143	4.89898	1.458
3	7.47110	5.550499	1.346
4	7.57124	4.78673985	1.582
5	7.77111	5.09607612	1.525
6	7.942815	4.55119939	1.745

Conclusion:

Based on the above results we can conclude that alpha-beta pruning technique reduces the time complexity to a great extent. It is also observed that with the increase in the search depth alpha beta pruning algorithm becomes much more efficient than the naive algorithm.

References:

1. https://pdfs.semanticscholar.org/d94a/c5293e18aeed42615b81db44f9424ffe0027.pdf
2. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/amp/?ref=rp
3. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-2-evaluation-function/
4. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/amp/?ref=rp
5. https://developers.google.com/machine-learning/glossary