The Reversi (Othello)

Arunothia Marappan

13378

(images)

With

Vineet Hingorani

Aim

To implement the game Othello with the Artificial Intelligence challenges using C++.

The Game Othello

Othello is a well-known two player board game. The following section describes the rules and popular strategies of this game.

The Rules

Othello is played by two players, Black and White, on an 8×8 board. On this board, so-called discs are placed. Discs have two sides: a black one and a white one. If a disc on the board has its black side flipped up, it is owned by player Black and if it has its white side up, it belongs to player White. The game starts with four discs on the board, as shown in figure1.

Black always starts the game, and the players make moves alternately. When it is a player's turn he has to place a disc on the board in such a way that he captures at least one of the opponent's discs. A disc is captured when it lies on a straight line between the placed disc and another disc of the player making the move. Such a straight line may not be interrupted by an empty square or a disc of the player making the move. All captured discs are flipped and the turn goes to the other player.

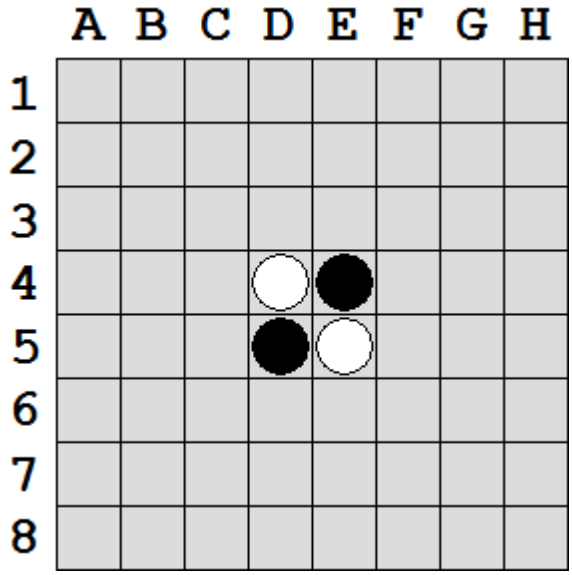


Figure 1: The initial position.

If a player cannot make a legal move, he has to pass. If both players have to pass, the game is over. The player, who has the most discs with his color flipped up, wins the game

Strategies

Othello is considered to be a game that is ‘a minute to learn, a lifetime to master’. This slogan accurately captures the game, because the rules of the game are very simple, but playing the game well is very difficult. For a computer player, it is important that it has some strategic knowledge about the game. We describe some commonly known strategies that are used by virtually every advanced Othello player.

Corners and Stable discs

During a game some discs, especially those in the middle, are flipped a lot of times. Discs that still can be flipped during the current game are called unstable discs. Stable discs, on the other hand, cannot be flipped anymore during the current game. This means that players owning stable discs cannot lose them anymore. Discs placed in the corners of the board are always stable. Discs that are adjacent to stable discs can also be stable. This makes corners strategically very important. In the example in Figure 2, the dark shaded squares indicate the stable discs for Black.

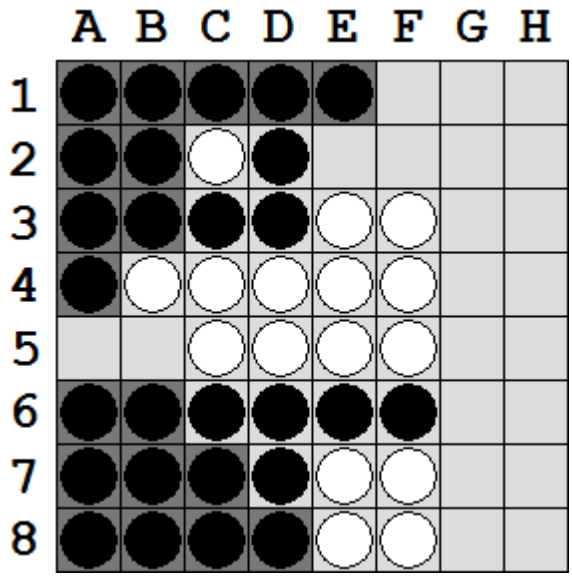


Figure 2: Stable discs for Black.

Location

An Othello board consists of 64 squares. Strategically, each of those squares has a certain importance. As explained above, the four corner squares are very important. To make strategic discussions easier, most squares on an Othello board are given names. Besides the corners, an Othello board contains X-, A-, B-, and C-squares and the Sweet 16. In figure 3, the locations of these squares are shown. The 16 dark-shaded squares in the middle of the board belong to the Sweet 16.

Typically, playing X-squares or, to a lesser extent, C-squares are considered to be strategically bad moves. If a player owns such a square, he risks losing the corner. Placing a disc in the Sweet 16, however, often does not bear any serious risks and is often a good move.

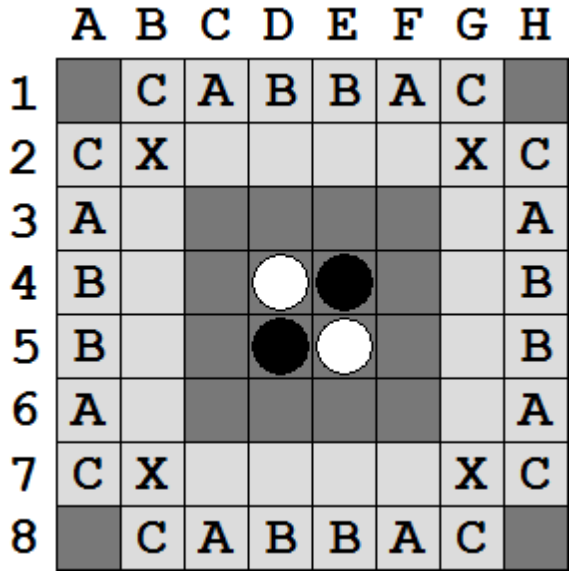


Figure 3: Types of squares on an Othello board.

Mobility and frontier

Each player is obliged, if possible, to play a move. Because of this rule, it is possible to force the opponent to make bad moves. This can be achieved by limiting the number of moves the opponent can do. Advanced players can place their discs in such a way to limit the possible moves of the opponent to only bad ones.

One way to increase your own and limit the opponent's mobility is to make your frontier as small as possible. All discs that are adjacent to at least one empty square belong to the frontier. A large frontier often means a large number of possible moves for the opponent, whereas a small frontier limits this number.

Parity

Playing the last move in a game is generally a major advantage. Discs that are flipped during the last move cannot be lost anymore and this feature gives the player doing the last move a considerable advantage. Typically, because Black begins the game, White can make the last move. However, when someone has to pass, the parity changes from one player to the other. So, in order to make the last move, Black should try to force an odd number of passes in the game to 'win parity'. White, on the other hand, tries to either avoid any passes or to force an even number of passes in the game.

What has been done in this project?

The user is given the choice to chose between two player game and user versus computer game. The AI part of the user versus computer game has been achieved with two algorithms: Min_Max Algorithm and Alpha_beta Pruning. There is also the option for the user to chose a random game generating function which would play with the computer in the chosen algorithm, with the mentioned tree depth and for the given number of times and display the respective results of % win and time taken.

The following global variables have been used to achieve the implementation of this program.

- A character 2-D Array named 'gamegrid' of size 8*8 which stores 'O' in the position occupied by player1 (user) and 'X' in the position occupied by player2 (computer) and '.' for unoccupied places.

- An integer 2-D Array named ‘pointsgrid’ of size 8*8 which stores the corresponding “value” of the location. This is done based on the strategies described earlier in my proposal. (refer the following box and fig 3 in the proposal)

Square Type (fig 3)	Owner of the square	
	Player	Opponent
Corner	+5	-5
X-square	-2	+2
C-square	-1	+1
Sweet 16	+2	-2
Others	+1	-1

- Integers ‘score1’ and ‘score 2’ to track the strategic score of the corresponding players as the game proceeds.

- Integers ‘count1’ and ‘count2’ to track the number of coins of the corresponding players on board as the game proceeds.

- Integers ‘color1’ and ‘color2’ to store the color values of the coins for the corresponding players for graphic display.

- An Integer ‘Winsize’ to decide the size of the window for graphic display.

- An integer ‘width’ to decide the width of each small square on the game grid for graphic display.

- Integers ‘leftend’ and ‘rightend’ to describe the start and end location of the game grid for graphic display.

- Integers ‘radius’ to describe coin radius and ‘font’ to describe the font size of texts for graphic display.

- Integer ‘D’ which decides as to how deep the Min_Max and Alpha_beta pruning algorithms should go down the tree.

- A class named ‘current_state’ (for Min_Max Algorithm implementation) with public elements as
    a. An integer ‘depth’ to say the depth of the tree at which the given node occurs.
    b. An integer ‘points’ to store the credits of that particular move.
    c. An integer ‘pl’ to denote the player who plays at this current state.
    d. Integers ‘posx’ and ‘posy’ to store the position being played at this current state.
    e. A container element list of type current_state named ‘branches’ to store the children of the given node. (cplusplus.com)
    f. An argument constructor to the given class which takes the player number, and position being played as arguments. The positions are given default values as ‘-1’ because the positions of the root current_state is decided only after all recursive calls.

- A self referencing class named ‘current_state_prune’ (for Alpha_beta Pruning implementation) with public elements as

a. An integer ‘v’ to store the current moves value.
b. Integers ‘posx’ and ‘posy’ to store positions of the current play.
c. A pointer to the same class named as ‘ptr’ to keep track of the path.
d. An integer ‘minmax’ to store -1 if the current node is a minimiser and 1 if it is a maximiser.
e. An integer ‘depth’ to store the depth of the current node.
f. Integer ‘alpha’ to store the best value achieved for the maximiser along the path back to the root.
g. Integer ‘beta’ to store the best value achieved for the minimiser along the path back to the root.
h. An argument constructor to the given class which takes the player number, positions and alpha, beta values as arguments. Alpha and beta are given default values to match their definition.

The following functions have been used to achieve the implementation of this program.

- **Void Createboard()**

  This function initializes the global variables gamegrid, pointsgrid, score1, score2, count1 and count2.

- **Void Print_circle_solid()**

  This function prints the coins in solid format for graphic display.

- **Void Print_circle_empty()**

  This function prints the coins in empty format for graphic display.

- **Void Create_window_box()**

  Create window and a square box for graphics display.

- **Void heading()**

  Print heading using graphics.

- **Void create_window_grids()**

  Create the grids on the board for graphics display.

- **Void print_coins()**

  Print coins of the players on window using graphics.

- **Void print_number()**

  Function used to number the grids (x and y axis).

- **Void print_score()**

  Function to print the score on window using graphics.

- **Void printboard()**

  Function to print complete game board using graphics.

- **void suggest(int)**

  Function to suggest moves for the player using graphics.

- **void print_board()**

  Prints board in grid format using 'x' and 'o' display on console.

- **void score()**

  This function updates the global variables score1, score2, count1, count2 as the game proceeds.

- **void change(int, int, int, int, int)**

  This function is called by flip function to flip the coins in the given direction.

- **void flip(int, int, int)**

  This function flips the coins given the new insertion position (x, y) and the player number.

- **int isvalid(int, int, int)**

  This function checks whether the given move is valid or not and takes positions and player number as arguments.

- **void suggest(int)**

  Suggest moves for given player number on console.

- **int possible(int)**

  Returns 0 or 1 depending upon whether moves are available for the player number.

- **void two_player()**

  This is the two player game function. It calls all the predefined functions required to run a two player game of Reversi.

- **bool compare(current_state, current_state)**

  Function used by min_element () and max_element ()

- **int count(current_state *)**

To count the coins of player on board after insertion of a given position or after the shift of current_state to the current node

- **current_state * max(current_state *)**

  Max function to find the best points for maximiser from its branches. This function is preferred over the predefined STL min_element () and max_element functions so as to distinguish between moves with equal scores.

- **current_state * min(current_state *)**

  This is the Min function to find the best points for minimiser from its branches. Its functioning is similar to max function.

- **void makebestmove(current_state *)**

  This function gives the best move for computer in one player game using minmax algorithm.

- **void best_move(current_state_prune)**

  This function gives the best move for computer in one player game using alpha beta pruning.

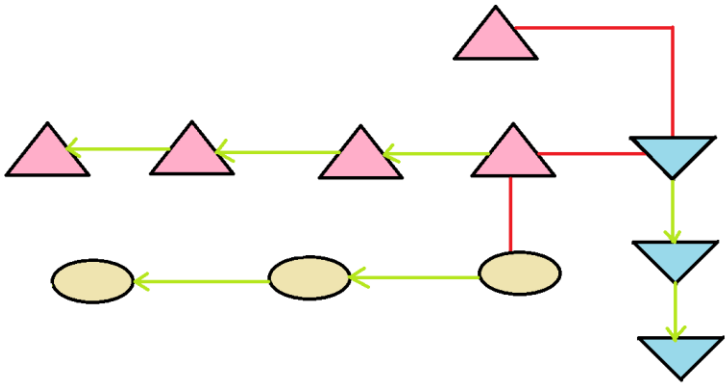
- **int random_generator (int)**

  It is the function that generates random moves and helps in game analysis by playing against computer. It takes the algorithm choice as its argument. It uses rand () and srand () functions to generate random moves. The header <time.h> has been used to seed srand () function.

- **void one_player()**

  This is the user versus computer game function.

Min_Max Algorithm Implementation (Wikipedia, Mini max theory)



In the function makebestmove (current_state *), the Min_Max algorithm has been used to find the best move. The function is a recursive function with base cases as –

- If maximum depth of search is reached then the data member ‘points’ of the current_state class is initialized with the difference of score of computer and score of user.
- If there is no valid move available for the player of current_state to play, then the data member ‘points’ of current_state class is initialized with score difference of computer and user and a + or - 5 is added depending upon whose move is being skipped. This follows from the logic that a move which can lead to the skipping of chances of the opponent is advantageous.

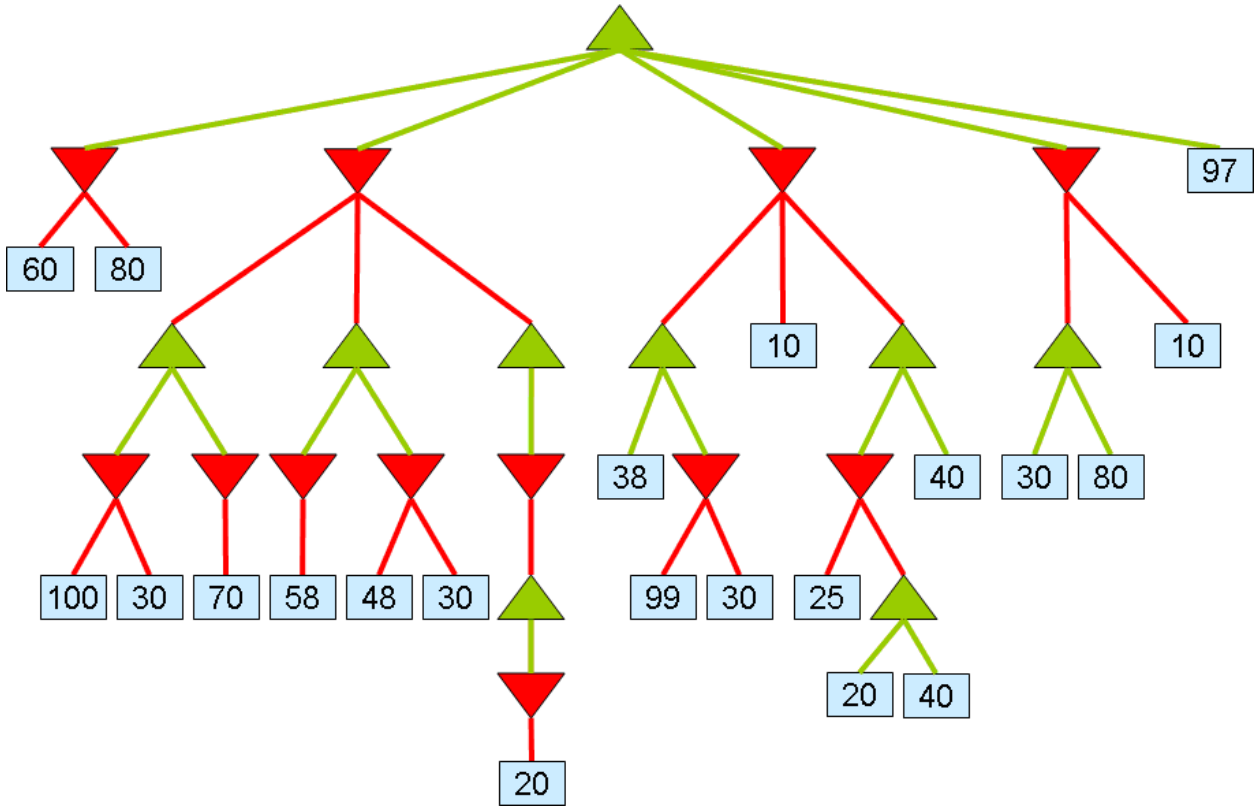
The even depth nodes act as maximisers and odd depth nodes act as minimisers. Therefore, the points in these cases are calculated as:

Even depth: max of points of its children

Odd depth: min of points of its children

The stack arrangement has been done using the class current_state. As each object of this class have a list of its own type as a member, the list acts as the children for each node.

Alpha_Beta Pruning Algorithm Implementation (Alpha beta pruning example)



In the function best_move (current_state_prune *), the alpha-beta pruning algorithm has been used to find the best move. The function is again recursive with similar base cases with member ‘points’ being replaced with ‘v’, in this case.

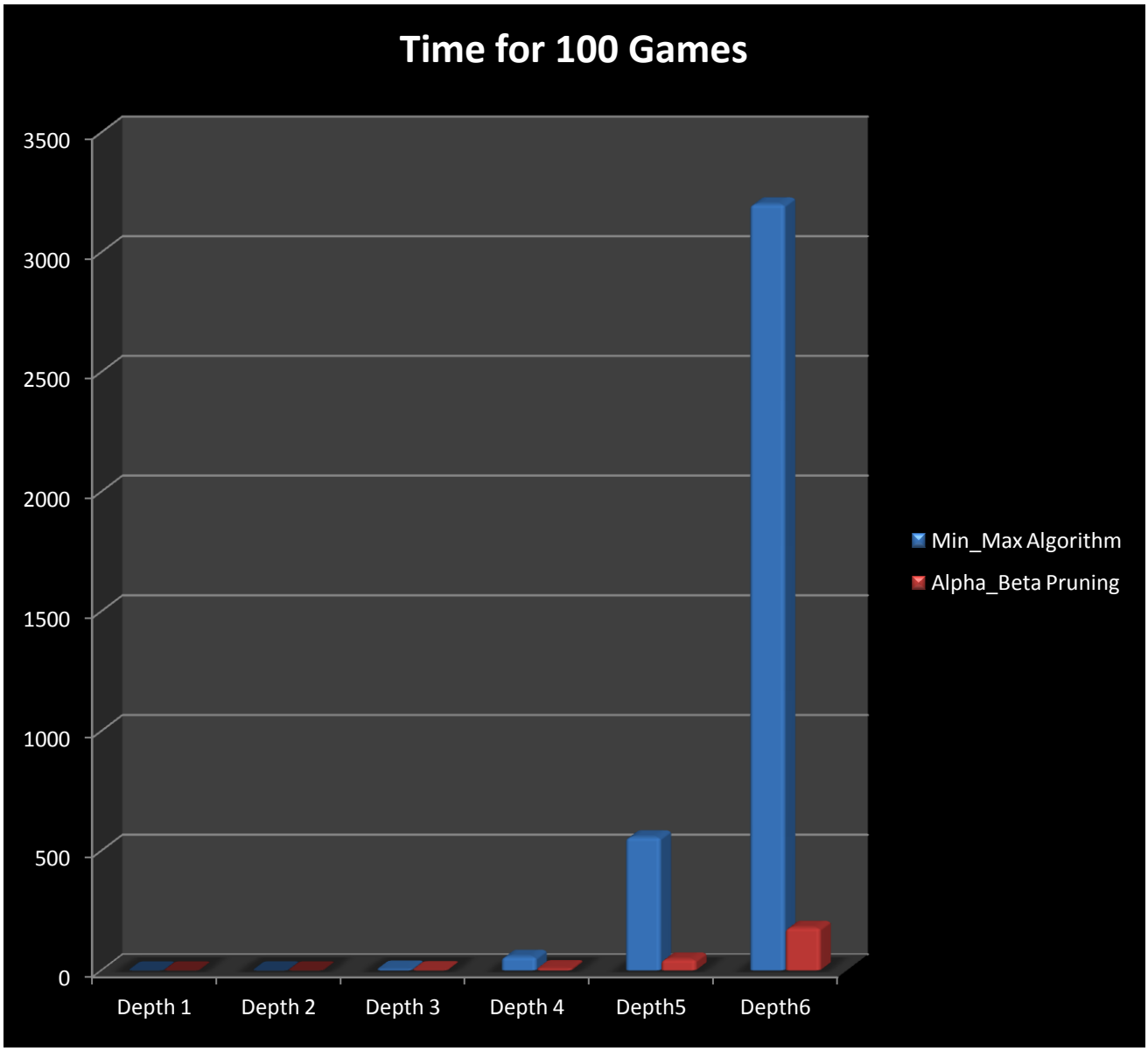
This algorithm is based on min-max but, the difference is that, the time of execution of alpha-beta is much lesser than pure Min-Max because, here, we don’t go into those branches which are not fruitful for our search. That is here we use the extra variable called alpha (beta) which

track the best value achieved by the maximiser (minimiser) along the way back to the root. The moment we get to know that the value of maximiser can get greater than beta, we prune away from that branch because the minimiser will never allow this to happen. Similarly, when we get to know that the value of minimiser can get lesser than alpha, we prune away from that branch because again maximiser will never allow this to happen.

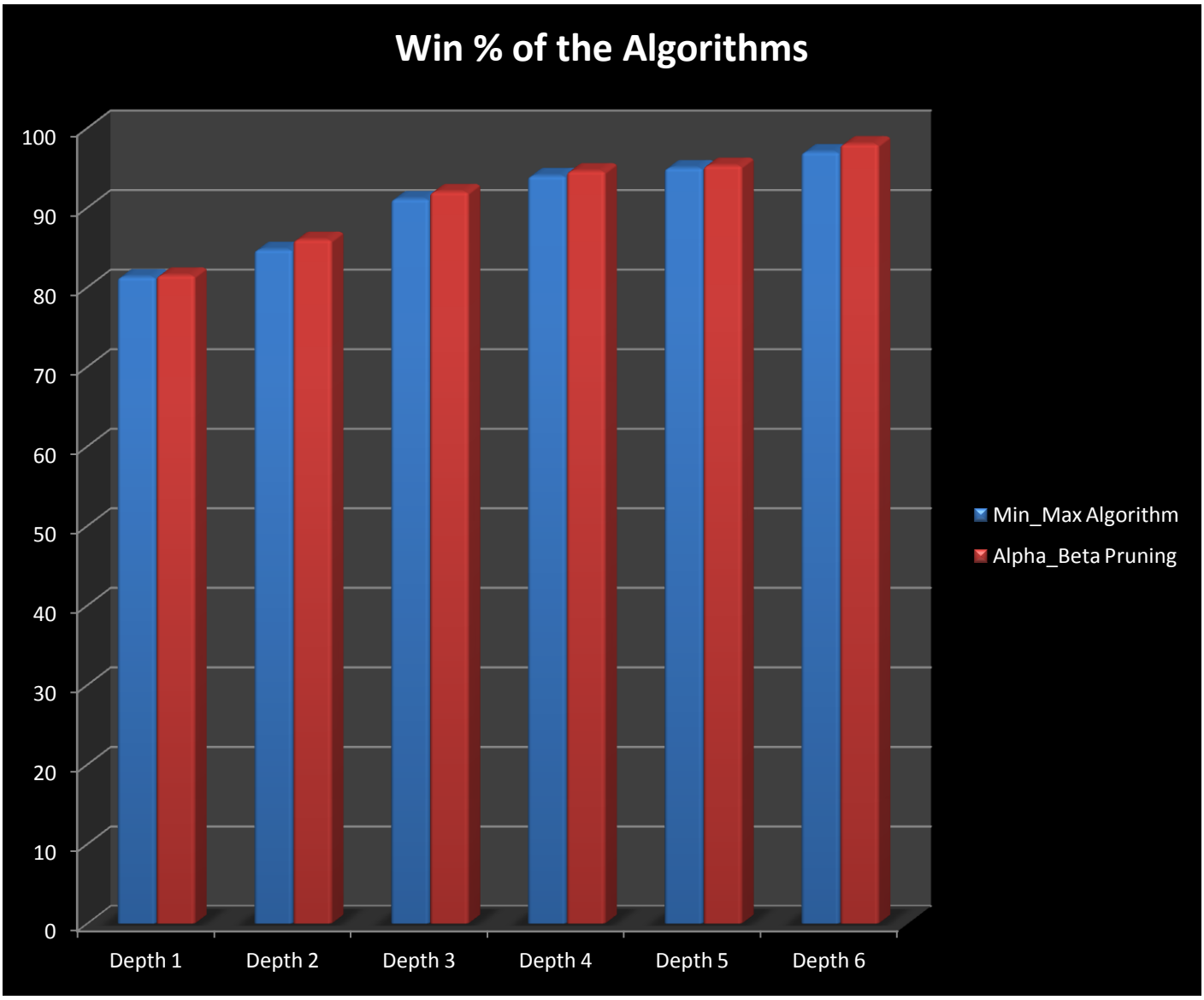
Thus, this saves a lot of unnecessary travel into the branches.

To achieve this algorithm the class current_state_prune has been used. This is a self referential class and this class has been used such that at any instant there is just one linked list maintained, the list from root to the current branch. This is done so to achieve the definitions of ‘alpha’ and ‘beta’ in the pruning algorithm.

The bar graph below has been obtained by using the random game generating function. The y-axis is time axis measured in seconds.



The bar graph below has been obtained by using the random game generating function. The y-axis shows the percentage of win.



DEPTH	Min_Max Algorithm		Alpha – Beta Pruning	
	Time	Win %	Time	Win %
1	0.12	81.3	0.07	81.5
2	0.74	84.73	0.36	85.99
3	6.3	91.1	1.9	92
4	57	94	8.8	94.6
5	556	95	46	95.3

6	3200	97	178	98

References

- Wikipedia. (n.d.). *alpha beta pruning.*

  http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

- Wikipedia. (n.d.). *Mini max theory.*

  http://en.wikipedia.org/wiki/Minimax

- cplusplus.com. (n.d.). *Standard libraries.*

  www.cplusplus.com/reference

- Alpha-Beta pruning Examples

  http://www.youtube.com/watch?v=xBXHtz4Gbdo

- Google Images

  https://www.google.co.in/search?q=reversi&source=lnms&tbm=isch&sa=X&ei=J7h_Ur-mIYbMrQfQ_ICQCA&ved=0CAcQ_AUoAQ&biw=1600&bih=799