**Intelligent Checkers Player using Alpha-Beta Depth-algorithm Documentation**

Team Members:

|  |  |  |  |
| --- | --- | --- | --- |
| Department (General) | Level | Team member name (**in Arabic**) | Team Member ID |
| CS | 3 | احمد سمير حشمت حافظ | 201900040 |
| CS | 3 | طارق مصطفى اسماعيل محمد | 201900387 |
| CS | 3 | سهيله محسن سعد احمد | 201900357 |
| CS | 3 | عبد الرحمن عماد عبدالله السيد | 201900425 |
| CS | 3 | ادهم سيد محمد توفيق | 201900117 |

**Contains:**

**1) Projects idea in detail**

**2) Similar applications in the market**

**3) An initial literature review of Academic publications (papers)**

**4) Main functionalities + 2 Heuristic**

**5) Details of the algorithm(s)/approach(es) used**

**6) Development platform**

**7) Analysis of the results, what are the insights?**

**8) What are the advantages / disadvantages?**

**9) Why did the algorithm behave in such a way? What might be the future modifications you’d like to try when solving this problem?**

**10) Diagrams**

**11) Member’s rules**

**( Introduction and Overview )**

1. **Project idea in details:**

**-** The classic board game of checkers (also referred to as 8 x 8 draughts in Britain) is played by two opponents, on opposite sides of the gameboard. One player has the dark pieces; the other has the light pieces. Players alternate turns. A player may not move an opponent's piece. A move consists of moving a piece diagonally to an adjacent unoccupied square. If the adjacent square contains an opponent's piece, and the square immediately beyond it is vacant, the piece may be captured (and removed from the game) by jumping over it. Only the dark squares of the checkered board are used. A piece may move only diagonally into an unoccupied square. When presented, capturing is mandatory in most official rules and where the player does not capture the opposing player can remove the opponent piece as penalty (or muffin) and where there are two or more such positions the player forfeits that/those he cannot move. Although some rule variations make capturing optional. In almost all variants, the player without pieces remaining, or who cannot move due to being blocked, loses the game. The objective behind this project idea is to investigate if it is possible to create a game playing system using Alpha-Beta Depth-First algorithm that would beat novices with ease and at least challenge advanced novice to intermediate-level players by designing a rules-based expert system whose knowledge base consists of nothing more than the rules of checkers, and a set of guidelines for gameplay based on good strategy. The game can be represented in terms of an initial state, intermediate states, and (possibly) a (set of) final state(s). For example, in the case of checkers, the initial state would be the starting board position, the intermediate states would be the board positions resulting from various moves made by the players as the game progresses, and the final state(s) would be board positions in which one player has emerged as the clear victor, or the game is drawn. Therefore, these states form a state-space. Evidently, for any one player, some of the final states are more desirable than the others. It could then be seen that games could be approached as a quest to traverse this state-space, starting at the initial state and ending in one of the desirable final states (for instance, win or draw). State-space search has been used quite successfully in the past to create game-playing programs.

1. **Similar applications in the market and what are the functionalities/features, and how they work (if that information is available):**
   1. [Real Checkers Pc game](https://www.gametop.com/download-free-games/checkers/)
      1. Description -> Move diagonally into unoccupied squares and capture opponent pieces by jumping over them. Win this game when no opponent pieces have no remaining pieces or unable to make any moves.
      2. Features
         * Great graphics and relaxing soundtrack
         * Helpful undo feature
         * Advanced checkers engine
         * Game statistics are available
         * Perfect for a quick break or a few hours of fun
      3. Tips and Tricks
         * Control the center
         * Do not play only defensively
         * Be willing to sacrifice pieces
         * Use forced moves to your advantage
         * Aim to get a checker to the end of the board
   2. [Online Checkers game](https://cardgames.io/checkers/)
      1. The game can end in four different ways:
         * If a player has lost all his pieces he loses.
         * If a player can't move at all, all his pieces are blocked, he loses.
         * The exact same board state has come up three times without any men captured in between. The game ends in a draw. This is to avoid situation with two pieces left just moving around never being able to capture each other.
         * There have been 100 moves (50 for each player) with no piece captured. The game ends in a draw.
   3. [Draughts Android game](https://play.google.com/store/apps/details?id=pl.lukok.draughts)
2. **An initial literature review of Academic publications (papers) relevant to the idea (at least 5 papers):**

**- Introduction**

Searching trees of possible alternatives is a task common to a wide range of programs. The efficiency with which these trees can be searched is of critical importance to such programs, since the trees are typically very big.

**This paper is concerned with measuring the efficiency of the minimax search of a game tree with alpha-beta pruning.**

**- Minimax**

The operation of the minimax search procedure and the alpha-beta pruning procedure are illustrated in the context of game playing programs. We give the name Max to the player whose turn it is to move and the name Min to his opponent.

Max tries to maximize the ultimate value of the game while Min tries to minimize the value. A lot of strategies exist to help a player in determining his next move, but the minimax procedure has received the most attention in programs which play games of perfect information.

In the tree, the nodes of the tree are interpreted as positions, and the arcs from each node are the legal moves from that position. The square nodes indicate it is Max's turn to move while the circles indicate it is Min's turn. (Figure 1)

In the minimax procedure the backed-up value of a Max position is the maximum of the values of its immediate successors and similarly, the backed-up value of a Min position is

the minimum of the values of its immediate successors, each node the player to move will choose the move which is most favorable to himself.

**- Alpha-beta and minimax**

The alpha-beta calculation is comparable to the minimax calculation in that the two of them observe a similar best move from position p and both will relegate a similar worth of anticipated that advantage should it. Alpha-beta is quicker than minimax because that it doesn't investigate some parts of the tree that won't influence on the esteem of value.

Alpha-beta algorithm is not altogether a different algorithm than mini-max, it’s basically an optimized version of it.

it optimizes mini-max by reducing the computing time by visiting fewer nodes in a decision tree than visited in mini-max.

in alpha-beta, there are two extra parameters passed to mini-max function.

We have two main factors in our algorithm:

1)Maximizing player that assured of a maximum score which is stored in alpha.

2)Minimizing player that assured of a minimum score which is stored in beta.

Alpha and Beta values initializing with the worth values:

Alpha: (∞ -) , Beta:) ∞(

\*The tracing will be terminating if (beta<=alpha) \*

The algorithm can be illustrated with the tree of depth three in Figure 1, If the searching proceeds in (DFS) a depth-first fashion from left to right and that the root node is a Max node.

**Consider**: is max.

is min.

**Diagram

Description automatically generated**

**#figure 1**

So now we can see how many nodes are pruned because it will not influence the algorithm’s decision.

COMPARISON BETWEEN MINI-MAX AND ALPHA-BETA PRUNING:

\*Note this screenshot does not belong to the tree above it’s for example\*

Table

Description automatically generated

In the same difficulty level, we see that alpha-beta in depth 4 takes 447 iterations with 6.00 ms for computation and regular mini-max takes 2799 with 33.00 ms, and in depth 8 alpha-beta takes 71773 iterations with 1009.00 ms vs 5847005 iterations with 55441.00 ms.

**- HEURISTIC FUNCTION:**

The application of heuristic function is to coordinate robot agents in adversarial environment.

We can say that heuristic function can calculate the value of the board depending on the placement of pieces on the board in games, **we can explain it as:**

The AI will evaluate the tree to a certain depth and then evaluation of the board is done using heuristic function. the AI finds who wins or loses confined to the specified depth and plays well.

The whole idea here is that AI can be guided to a winnable or losable position by heuristic function.

We can refer to this function as Evaluation Function or Heuristic Function.

The fundamental thought behind the heuristic function is to assign a high value for a board if maximisers turn is being played or a low value for the board if minimizers turn is being played.

It is to be kept in mind that the AI is the maximizer and human is the minimizer, Alpha is the best value that the maximizer can ensure at that level or above. Beta is the best value that the minimizer can ensure at that level or above. The first thing that is done is the computation of the score (value) of the current board. Next check if the current state is a terminal state or not.

The subsequent stage is to make different game states for the next cycle. For each game state made call the minimizer function. The minimizer function is basically same as the maximizer function. The board score is calculated first. Then it is checked if the game state is a terminal state or not.

A loop is run to generate the next iteration of game states. For each game state produced the expanding move for the AI is calculated. The principle of alpha beta pruning is applied to reduce the number of game states that should be checked. This function returns the minimum score of the human player.

**References:**

[1)International Research Journal of Engineering and Technology (IRJET)( Alpha-Beta Pruning in Mini-Max Algorithm –An Optimized Approach for a Connect-4 Game)](https://d1wqtxts1xzle7.cloudfront.net/56816350/IRJET-V5I4366-with-cover-page-v2.pdf?Expires=1640229250&Signature=TqZOAhNNZrtPeiPhEs7JWnkqKGEjEREQDMYtUBpI~UhU5aTVlXKE7sCAR~1F~eFKIHFz6Xs85ZUKpQRnl0Je-yeaZbX0Lh34UtashAE6HgIGBjgFGOVqYZ7HqDCslmeiBGgBf5aFpX9H44dcktfTqfzOjc2rNelMk~3ESw-xugduPnXkOTgGRWd11mBVrbATLNJ0VnRSYFcRhEXfgo2UaTHHQyWPJ~HzNV55v8IzxVT9QVkENDALrOqwz~GtEfBDACnJcE~NLJIfmJLWbFnJOTREkj7sNbOtCgoZLL14y~3sJT9Uw6QQ-J7hCuXIzFy8oJlYZ1XziM-KMYhRdOMm~Q__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA)

2) ANALYSIS OF THE ALPHA-BETA PRUNING ALGORITHM S. H. Fuller, J. G. Gaschnig and J. J. Gillogly.

**( Proposed Solution )**

**(4) Main functionalities + 2 Heuristic functions comparison:**

**Alpha\_beta\_algorithm:**

def alpha\_beta\_algo(board, depth, alpha, beta, max\_player, game):

this function is an algorithm of code that controls how the computer will play and imagine the next steps to win player,

it takes the current shape of the board, depth of the tree that draws the game, alpha, beta, max player = (turning variable), then

if depth == 0 or board.winner() != None:

we will check if depth == 0 or there is a winner in the game, then return board evaluate (which is the heuristic value) and the board then check if this is AI turn (which mean we want to maximize the winning score)

if max\_player == BLUE:

if it’s max\_player => (AI turn)

for move in get\_all\_moves(board, BLUE, game):

the algorithm loops on all moves in this state(board) and for every move, the algorithm compares two values (evaluation value and the maxEval (-inf) and choose the max between them.

evaluation = alpha\_beta\_algo(move, depth-1, alpha, beta, False, game)[0]

And then it checks the alpha-beta pruning :

Start to compare the alpha value with the evaluation value and also choose the max between them.

if beta <= alpha:

Then check alpha and beta condition if beta <= alpha (which mean this road will not influence the decision don’t investigate it ). It will break the iteration and move on next move

if maxEval == evaluation:

If not, it will check if the maxEval value == Evaluation value then it means this move is the best move and return the maxEval value and this move which lead him to win.

This is the player turn algorithm that will generate or simulate how the human player will play and make all these steps but in a way that will be minimizing the winning score for AI, so it chooses the min value instead of the max and returns the best move to achieve that.

**Board class:**

1. def \_\_init\_\_(self):

This function used to initialize the number of pieces of the player and AI model and create them (12 for each one), and the number of kings for each the played and AI model (0 at the start of the game).

1. def create\_squares(self, win):

this function used to draw squares at the board and handle to start each row with different color (start with black and then white).

1. def evaluate(self):

**Heuristic Function (1):**

return self.blue\_left - self.white\_left

For our implementation, it was evident that we would need to use an adversarial search. Additionally, we are planning on implementing an Alpha-Beta Depth-First algorithm into our artificial intelligence since this is a turn-based game. Now that we decided on those two elements of our AI, we needed to think of a good approach to our heuristic function. For this, we initially will use a simple function that will take the total count of the AI's chips minus the total amount of the opponent chips (i.e.: (# my pieces) - (# opponents’ pieces)).

**Heuristic Function (2):**

return self.blue\_left - self.white\_left + (self.blue\_kings \* 0.5 - self.white\_kings \* 0.5)

We quickly realized that we would eventually have to modify our heuristic function to add weight to acquiring a king since that is a very advantageous move for the player.

The "evaluation" of our project is simply how well it plays (against other AI and against humans). Obviously, we would like to maximize the number of games that our AI wins. To do this, we are going to modify our heuristic to include a weight for a king piece, as having a king makes more states available with one move and thus allows for more chances to take the opponent's pieces. (i.e.: (# my pieces) - (# opponents’ pieces) + (# my king pieces \* 0.5 - # opponents’ king pieces \* 0.5)). In the function we multiply the number of kings \* 0.5 to make the result of all function a small number to treat with it easily.

1. def get\_all\_pieces(self, color):

this function used to return a list of all pieces of the certain player (player or AI model).

1. def move(self, piece, row, col):

this function used to make the piece move from a position to another (by making the new square has a piece and the old one is empty), and make sure that the new position make the piece be a king or not if it makes the piece be a king, so the piece become a king piece if not there is no change.

1. def get\_piece(self, row, col):

this function used to give us the piece in a specific row and column from the list of pieces.

1. def create\_pieces(self):

this function used to put the pieces at its position in the board at the first of the game.

1. def draw(self, win):

this function used to draw the board (squares and pieces).

1. def remove\_piece(self, pieces):

this function used to remove the piece that crashed.

1. def winner(self):

this function used to return the winner player (played or AI model) by checking if the number of pieces of the player <= 0 or not if <= 0 then the player loses the game.

1. def get\_valid\_moves(self, piece):

this function used to give us all the available moves of the pieces if the piece is white, it can move only to up, but if the piece is blue, it’s can only move to down, if the piece is a king piece (white or blue) it can move up and down.

((we do this function by the help of the next two functions))

1. def \_traverse\_left(self, start, stop, step, color, left, skipped=[]):

this function used to determine the valid moves for the piece from the left side, and if there is a jump it add it also to the valid moves.

1. def \_traverse\_right(self, start, stop, step, color, right, skipped=[]):

this function used to determine the valid moves for the piece from the right side, and if there is a jump it add it also to the valid moves.

**- Game class:**

This class create to interface game’s movement and pieces, control board and update of board, and predict valid moves.

**Functions:**

def \_\_init\_\_(self, win):

this function to initiate the shape of board.

def \_init(self):

this function initializes all properties for the game to start, it initiates selected =none, object of board from class board, initiates variable (turn) by white and it makes list for the available moves

def winner(self):

it’s winner method to return shape of board in game.

def update(self):

it’s update method that draw and update board, first it draws board, draws valid moves on board and display this method at the end.

def reset(self):

it’s reset method that return board’s shape to reset that there is no selected and list of valid moves is zero.

def select(self, row, col):

select method to try move the selected piece, the method takes parameters (row, col) and send them to \_move method to try to move piece to this position and return if this valid position or not if it is not valid position put on selected variable and recursively this method again until find correct selected value, then put in piece the valid row and column and if selected piece is valid put it in list valid moves.

def \_move(self, row, col):

move method to handle the moving of the selected piece to the position we choose, check if there is a selected piece and if the pos that we need to move to is = 0, methods call function move with selected piece and new position want to move to it and moves it to new position, then remove this valid position from list of valid moves not to use it again.

def draw\_valid\_moves(self, moves):

this method to interface circles in valid positions after we selected piece to know which positions, we can move to it.

def change\_turn(self):

this method to know who is in this turn to play, at first, we make list of valid moves =0, and check if this is turn of white make turn =blue else make turn= white to exchange turn.

def get\_board(self):

this method to return board.

def ai\_move(self, board):

this method to put in board the shape of new board that algorithm do and exchange turn.

**-** **Piece Class:**

The main function of this is to create game pieces and draw them on the board, calculate their position in the square, and save their current position.

Function:

1) \_\_init\_\_(self, row, col, color):

This function is the constructor that initiates the start values for each piece take its row, column and its color and initially set the value which tells us this piece is a king or not by FALSE

And call the function that calculates the piece in the square

2) calc\_pos(self):

This function basically makes some math logic to draw or put the piece in the middle of

square.

By determining the X-axis position and Y-axis position.

3)  def make\_king(self):

This function converts the piece status if it reaches the king’ condition (Make the king variable = TRUE).

4) def draw(self, win):

This function draws the piece shape (circle)

By calculating the radius and passing it to the function (Draw\_circle) in pygame

And check if this piece is king it converts its shape to the CLOWN image we use.

5)  def move(self, row, col):

When a piece makes a move this function takes its new coordinates and update its position and use calc\_pos to draw it in a new square.

6) def \_\_repr\_\_(self):

This function fixes any error that happened while drawing pieces by showing what we return from it.

**- Constants File:**

This class is used for the constant variables and features that we use during the program.

1. The width and the height of the board

WIDTH, HEIGHT = 800, 800

1. The number of rows and columns in the board

ROWS, COLS = 8, 8

1. the size of each square in the board

SQUARE\_SIZE = WIDTH//COLS

1. the squares at the board are with blue and black colors

BLACK = (0,0,0)

BLUE  = (0,0,255)

1. AI pieces is colored by blue while player with white

WHITE = (255,255,255)

1. when the player select a piece the red color used to define the possible moves

RED   = (255, 0, 0)

1. the gray color is used as a boarder for all pieces

GREY  = (128, 128, 128)

1. It’s the crown that be inside the piece that considered as king piece

CROWN = pygame.transform.scale(pygame.image.load('checkers/assets/crown.png'), (44, 25))

**- Main Function:**

FPS = 60

This is the number to render the game in a fixed frames number per second.

WIN = pygame.display.set\_mode((WIDTH, HEIGHT))

The WIN is a constant variable that we store in it the window display that we get from the built in function (pygame.display.set.mot((WIDTH, HEIGHT))) in Pygame library.

pygame.display.set\_caption('Checkers')

This line put the name of game (Checkers) in the GUI Header by using the built in function (pygame.display.set\_caption(‘Checkers’)) in Pygame library.

def get\_row\_col\_from\_mouse(pos):

This function to get the position of the piece that mouse clicked.

def main():

This function will start all game as following:

clock = pygame.time.Clock()

This to make the game run in fixed speed

game = Game(WIN)

This to create a board and start the game by creating a game object from the game class and pass to it the WIN variable.

while run:

This is the start of event loop

if game.turn == BLUE:

This to make the ai algorithm start by calling the algorithm function and pass to it the game board, depth, alpha, beta, color of ai pieces, and the game object as following:

value, new\_board = alpha\_beta\_algo(game.get\_board(), 4, float('-inf'), float('inf'), BLUE, game)

game.ai\_move(new\_board)

this function is an algorithm of code that controls how the computer will play and imagine the next steps to win player,

it takes the current shape of the board, depth of the tree that draws the game, alpha, beta, max player = (turning variable), then

we will check if depth ==0 or there is a winner in the game, then return board evaluate (which is the heuristic value) and the board then check if this is AI turn (which mean we want to maximize the winning score)

if it’s max => (AI turn)

the algorithm loops on all moves in this state(board) and for every move, the algorithm compares two values (evaluation value and the maxEval (-inf) and choose the max between them.

And then it checks the alpha-beta pruning:

Start to compare the alpha value with the evaluation value and choose the max between them.

Then check alpha and beta condition if beta <= alpha (which mean this road will not influence the decision don’t investigate it). It will break the iteration and move on next move

If not, it will check if the maxEval value == Evaluation value then it means this move is the best move and return the maxEval value and this move which lead him to win.

This is the player turn algorithm that will generate or simulate how the human player will play and make all these steps but in a way that will be minimizing the winning score for AI so it chooses the min value instead of the max and returns the best move to achieve that.

if game.winner() != None:

This to quit the game if ai win or the human player.

for event in pygame.event.get():

This is to check if any event happens in the current time and implements the required according to the event.

game.update()

This to update any changes that happens in the game board.

pygame.quit()

This to quite from the pygame library.

main()

This to call the main function and run the code inside it.

**( Applied Algorithms )**

**(5) Details of the algorithm(s)/approach(es) used (can be explained using block diagrams):**

- Alpha–beta pruning is a [search algorithm](https://en.wikipedia.org/wiki/Search_algorithm) that seeks to decrease the number of nodes that are evaluated by the [minimax algorithm](https://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves) in its [search tree](https://en.wikipedia.org/wiki/Game_tree). It is an adversarial search algorithm used commonly for machine playing of two-player game.

It stops 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.

The core idea of the algorithm the algorithm maintains two values, α and β, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially, α is -∞ and β is ∞, both players start with their worst possible score. Whenever the maximum score that the minimizing player β is assured of becomes less than the minimum score that the maximizing player α is assured of (i.e., β < α), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

**Example**

**Step 1:** At the first step the Max player will start first move from node A where **α= -∞** and **β= +∞**, these value of alpha and beta passed down to node B where again **α= -∞** and **β= +∞**, and Node B passes the same value to its child D.

Diagram

Description automatically generated

**Step 2:** At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

**Step 3:** Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

Diagram

Description automatically generated

In the next step, algorithm traverse the next successor of Node B which is node E, and the values of **α= -∞**, and **β**= 3 will also be passed.

**Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (**-∞**, 5) = 5, hence at node E **α**= 5 and **β**= 3, where **α**>=**β**, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

Diagram

Description automatically generated

**Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (**-∞**, 3) = 3, and **β**= **+∞**, these two values now passes to right successor of A which is Node C.

At node C, **α**=3 and **β**= **+∞**, and the same values will be passed on to node F.

**Step 6:** At node F, again the value of **α** will be compared with left child which is 0, and max (3,0) = 3, and then compared with right child which is 1, and max (3,1) = 3 still **α** remains 3, but the node value of F will become 1.

Diagram

Description automatically generated

**Step 7:** Node F returns the node value 1 to node C, at C **α**= 3 and **β**= **+∞**, here the value of beta will be changed, it will compare with 1 so min (**∞**, 1) = 1. Now at C, **α**=3 and **β**= 1, and again it satisfies the condition **α**>=**β**, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.

Diagram

Description automatically generated

**Step 8:** C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.

Diagram

Description automatically generated

**Resources**

1. [**https://www.javatpoint.com/ai-alpha-beta-pruning**](https://www.javatpoint.com/ai-alpha-beta-pruning)
2. [**https://en.wikipedia.org/wiki/Alpha%E2%80%93beta\_pruning#:~:text=Alpha%E2%80%93beta%20pruning%20is%20a,%2C%20Go%2C%20etc**](https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#:~:text=Alpha%E2%80%93beta%20pruning%20is%20a,%2C%20Go%2C%20etc)**.).**

**( Experiments & Results )**

**(6) Development platform:**

1. **Visual Studio**
2. **Python 3.9.0 version**
3. **Pygame Library -> to develop checkers 2D game.**
4. **GitHub Repo ->** [**https://bit.ly/33TEyHw**](https://bit.ly/33TEyHw)
5. **Google Drive ->** [**https://bit.ly/3HjgInc**](https://bit.ly/3HjgInc)

**( Analysis, Discussion, and Future Work )**

**(7) Analysis of the results, what are the insights?**

The core idea of the algorithm the algorithm maintains two values, α and β, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially, α is -∞ and β is ∞, both players start with their worst possible score. Whenever the maximum score that the minimizing player β is assured of becomes less than the minimum score that the maximizing player α is assured of (i.e., β < α), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

Diagram

Description automatically generated

**(8) What are the advantages / disadvantages?**

- **Advantages**

* Great graphics
* Advanced checkers engine
* The ai algorithm makes a decision in a short time in small depth.
* Perfect for a quick break or a few hours of fun
* Aim to get a checker to the end of the board

- **Disadvantages**

* Do not play only defensively
* Be willing to sacrifice pieces

**(9) Why did the algorithm behave in such a way? What might be the future modifications you’d like to try when solving this problem?**

- The alpha-beta calculation is comparable to the minimax calculation in that the two of them observe a similar best move from position p and both will relegate a similar worth of anticipated that advantage should it. Alpha-beta is quicker than minimax because that it doesn't investigate some parts of the tree that won't influence on the esteem of value.

Alpha-beta algorithm is not altogether a different algorithm than mini-max, it’s basically an optimized version of it.

it optimizes mini-max by reducing the computing time by visiting fewer nodes in a decision tree than visited in mini-max.

- The future modifications we would like to try when solving this problem is to make the ai algorithm for the checkers game using genetic algorithms cause it search from a population of points, not a single point, supports multi-objective optimization, se probabilistic transition rules, not deterministic rules and so on.

**10) Diagrams**

**1)use case:**

Diagram

Description automatically generated

**2) flow chart:**

Diagram

Description automatically generated

**11) Member’s roles:**

**Documentation: (All)**

**Project Code:**

1. **Alpha\_beta Algorithm**
   * 1. **All**
2. **Board class**
   * 1. **Ahmed Samir Hishmat (**201900040**)**
     2. **Abdelrahman Emad Abdallah (**201900425**)**
3. **Game class**
   * 1. **Sohaila Mohsen Saad (**201900357**)**
     2. **Adham Sayed Mohamed (**201900117**)**
4. **Piece Class**
   * 1. **Tarek Mostafa Esmail (**201900387**)**
5. **Constants file**
   * 1. **Ahmed Samir Hishmat (**201900040**)**
     2. **Abdelrahman Emad Abdallah (**201900425**)**
6. **Main file**
   1. **All**