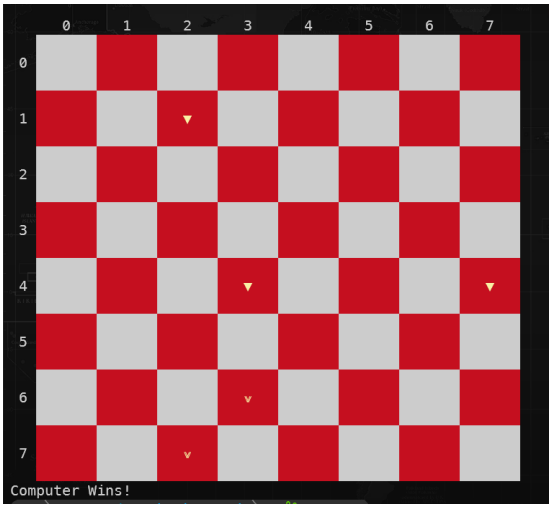
CSD 311

ASSIGNMENT 2

Documentation - CHECKERS AI



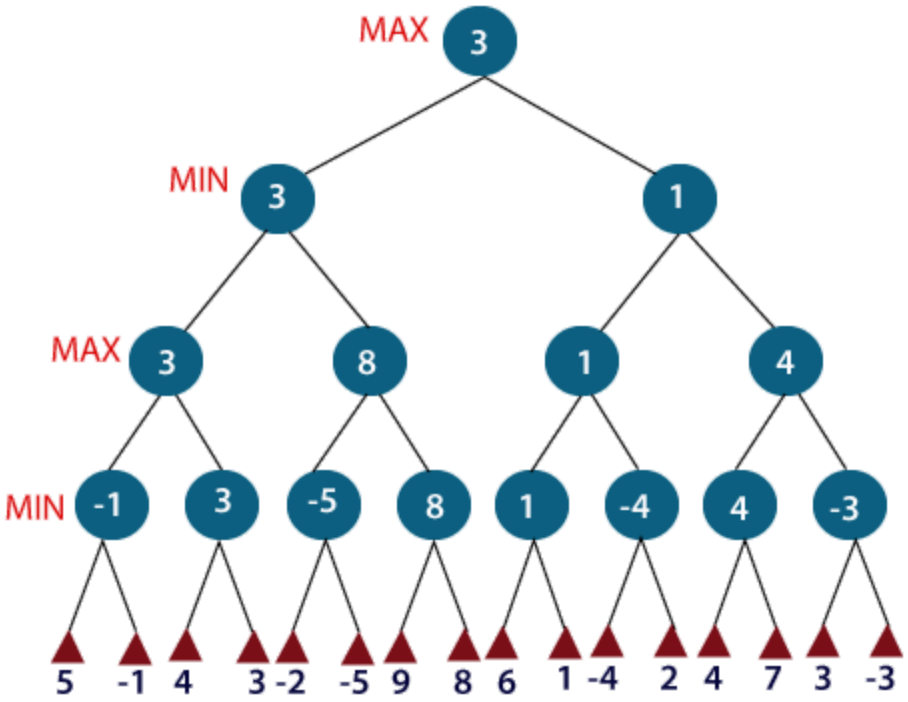
1. Algorithms -

- **Minimax Algorithm-**

The time complexity of minimax is O(b^m) and the space complexity is O(bm), where b is the number of legal moves at each point and m is the maximum depth of the tree.

It is basically a search technique for game trees. It is assumed that the participants will take turns making moves. It employs a utility function whose values are beneficial to player 1 when they are large and beneficial to player 2 when they are small. As a result, the purpose of the first player (MAX) is to choose a move that maximises the utility function. The purpose of the second player (MIN) is to choose a move that

minimises the utility function (hence the name of the algorithm). Under the premise that



the opponent would play flawlessly, it maximises the utility.

- **Random Algorithm-**

The Computer picks up the move from the available list of legal moves randomly and makes its turn. For this, we have used the random.choice() function of python imported from the random package, and passed the list of legal moves as an argument to the choice function. There is no logic behind making a random move and there won't be any guarantee of AI winning the game as the moves are random without any sort of intelligence technique.

- **Alpha-Beta Pruning with Minimax-**

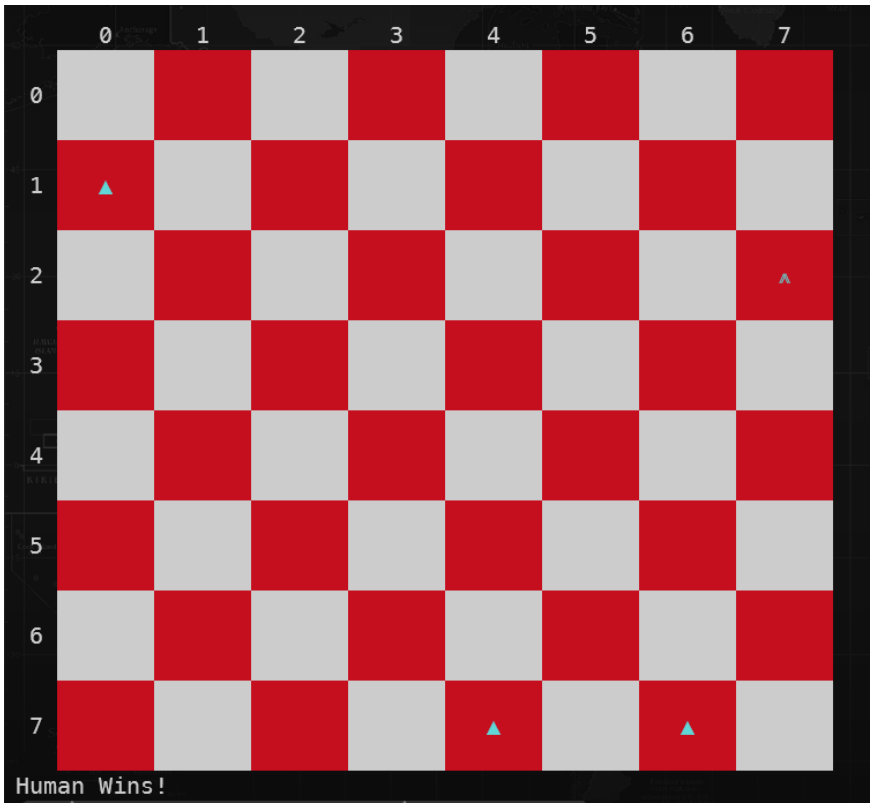
Attempts to determine the correct minimax option without inspecting each node in the game search tree. Many branches appear to be able to be overlooked (pruned). During the search, two values are created: alpha-value (related with MAX nodes) and beta-value (associated with MIN nodes).

Talking about the time complexity of Alpha-Beta pruning along with the Minimax algorithm, in worst case when there will be no node to be pruned, it is O(b^(d/2)), where b is the number of legal moves from minimax and d is the depth of minimax tree.

2. Features of our Checkers Game-

- Implemented both Random and Minimax Algorithm in our game. Players would be asked to make their choice as to whether to play with Random Algorithm or with more advanced Minimax Algorithm.
- Implemented Heuristics to calculate the score depending on the state of the game.
- Double or more jumps feature has also been implemented so that if a player or AI can conquer two or more pieces of opponent, they easily can.
- Implemented easy to understand GUI, showing different symbols for normal pieces and King pieces.

- Throughout the game, a list of the best possible moves for the user will be visible to help him and which move the computer played last can also be seen.



Human Wins

3. Files, methods and their definition -

We have created our program by distributing the functions into 4 files. Each file serves its own unique purpose which leads to the successful execution of Checkers game.

1. **Computer.py** - This file contains all functions related to the **running of the AI** It contains a **Class Computer** which contains the **Functions** -	
randomMove(self , game : Game) -> Move	For implementing Random Algorithm
minMaxSearch(self , game : Game) -> Game	For implementing Minimax Algorithm
numDefendingNeighbors(self , row : int, col : int , state) -> int	For getting best possible move for heuristic
heuristic(self , game : Game) -> int	For evaluating score depending upon the

	state
minVal(self, game: Game, alpha: int, beta: int, depth: int) -> int	Return the minimum value for Minmax function
maxVal(self, game: Game, alpha: int, beta: int, depth: int) -> int:	Return the maximum value for Minmax function
2. Game.py - This file contains all functions related to the **game rules** and **board movements** It contains the **Class Game** which contains the **Functions** -	
def newGame(self):	Setting the Board for New Game
def showBoard(self):	Displays GUI of the checkerboard.
def printPieces(self, val: int):	Helper function to print correct pieces.
def getLegalMoves(self, state: list) -> list:	Returns a list of all legal moves possible.
def getNormalMoves(self, moves: list, pieceType: int, startRow: int, startCol: int, state: list):	Function which governs a diagonal movement to be made by the checker.
def getCaptureMoves(self, moves: list, move: Move, pieceType: int, startRow: int, startCol: int, state: list):	Function which governs a capture move to be made by the checker.
def applyMove(self, move: Move, state: list):	Function which applies the given move and updates the board values.
def printListMoves(self, moves: Move):	Prints the list of possible moves for the user.
def printMove(self, moves: Move):	Prints moves made by the user/AI in coordinate form.
3. Move.py - This file contains all functions related to how all the moves are made on the given command of human and AI. It contains the **Class Move** which contain the **Functions** -	
def __init(self , *args)	Constructor to initialize default moves
4. Test.py - This file contains all the functions related to the main working of the game. It contains the main window from where the execution starts.	

4. Conclusion (Comment from our Side)-

Talking about various strategies, the Random algorithm is the most absurd one as it has no intelligence in it, picking only the random moves without considering capturing and winning moves.

In Minimax strategy, it takes account of various possible future moves by forming a tree and it also prefers capturing and winning moves over random moves, having a time complexity of O(b^m) where b= number of legal moves, m= depth of tree**.**

In Alpha-Beta Pruning, it works on the same principle as MiniMax, but it keeps cutting off the nodes to lessen the search. The time complexity is O(b^(m/2)) where b = number of legal moves, m = depth of tree**.**

Efficiency order	
Alpha - Beta Pruning	1
MiniMax	2
Random	3

5. Contributions -

Aman Yadav	Akansh Mittal	Aryaman Singh Rana