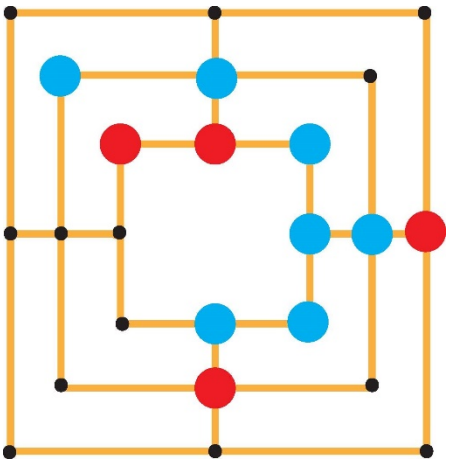
SALIM HABIB UNIVERSITY
(FORMERLY BARRETT HODGSON UNIVERSITY)

Nine Men's Morris

AI Project

Artificial intelligence Lab

25 Dec 2022



INSTRUCTOR

Sir. Waseem Abdul Rauf

REPRESENTATIVE

Ali Salman	CS-VI
Bushra Rubab	CS-VI
Dileep Kumar	CS-VII

Description

In recent years, a number of games have been solved using computers. All games were solved using knowledge-based method.

In this project we are going to code one of the oldest games still played "**Nine Men's Morris".** This game is a board game between two players: White and Black piece.

Project Specification

Nine Men's Morris Game will be **AI** based on a game theory decision rule of "**MinMax"**. Game plays more efficient move through all possible moves for an optimal solution. Player chooses the best move for all possible moves.

Game Rules

It is board game Consisting of three concentric squares, connected by the middles of each of the inner square's sides to the middle of the corresponding outer square's side. Pieces are played on the corner vertices. Each player has 9 piece and the game board. There are total of 24 locations for pieces.

Both players try to form 'mills' (three of their pieces in a straight line). A mill allows them to remove an opponent piece from the game. A player wins when their opponent has only 2 pieces left, or by leaving them without any possible moves.
There are 3 phases to the game:

1. Placing pieces on vacant points (9 turns each)
2. Moving placed pieces to adjacent points.
3. Moving pieces to any vacant point (when the player has been reduced to 3 men)

My approach

I have created 2 heuristics to have the AI learn how to play the game.

1. Number of Pieces: The higher the difference in the number of pieces of player 1 and player 2, the better.
2. Number of potential mills: The higher the number of mills potentially formed, the better.

Heuristic #2 performs much better than heuristic #1.

Important main function has been placed in main.py and are imported file used to play the game.

Game Feature

This is a human vs our AI. You can use two types of heuristic algorithms here. One uses number of remaining pieces as the basis for learning, and the other uses number of possible Mills formed as its factor. We can choose either one of the algorithms for the AI and the game will start. The AI is fully functional and can create mills, remove pieces etc. The AI performs decently.

Programming Language: Python
Algorithm: MinMax
IDE: PyCharm

Code

Main.py

```
# Function to print the board.
# Takes the board positions list as a parameter.
from copy import deepcopy


def printBoard(board):
    print(board[0] + "(00)----------------------" + board[1] +
          "(01)----------------------" + board[2] + "(02)")
    print("|                            |                            |")
    print("|                            |                            |")
    print("|                            |                            |")
    print("|       " + board[8] + "(08)--------------" +
          board[9] + "(09)--------------" + board[10] + "(10)      |")
    print("|       |                    |                    |       |")
    print("|       |                    |                    |       |")
    print("|       |                    |                    |       |")
    print("|       |         " + board[16] + "(16)-----" +
          board[17] + "(17)-----" + board[18] + "(18)        |       |")
    print("|       |         |                    |          |       |")
    print("|       |         |                    |          |       |")
    print("|       |         |                    |          |       |")
    print(board[3] + "(03)---" + board[11] + "(11)----" + board[19] + "(19)
" +
          board[20] + "(20)----" + board[12] + "(12)---" + board[4] + "(04)")
    print("|       |         |                    |          |       |")
    print("|       |         |                    |          |       |")
    print("|       |         |                    |          |       |")
    print("|       |         " + board[21] + "(21)-----" +
          board[22] + "(22)-----" + board[23] + "(23)        |       |")
    print("|       |                    |                    |       |")
    print("|       |                    |                    |       |")
    print("|       |                    |                    |       |")
    print("|       " + board[13] + "(13)--------------" +
          board[14] + "(14)--------------" + board[15] + "(15)      |")
    print("|                            |                            |")
    print("|                            |                            |")
    print("|                            |                            |")
    print(board[5] + "(05)----------------------" + board[6] +
          "(06)----------------------" + board[7] + "(07)")
    print("\n")


# Function to find adjacent locations for a given location.
# Returns a list of adjacent location
def adjacentLocations(position):
    adjacent = [
        [1, 3],
        [0, 2, 9],
```

```
        [1, 4],
        [0, 5, 11],
        [2, 7, 12],
        [3, 6],
        [5, 7, 14],
        [4, 6],
        [9, 11],
        [1, 8, 10, 17],
        [9, 12],
        [3, 8, 13, 19],
        [4, 10, 15, 20],
        [11, 14],
        [6, 13, 15, 22],
        [12, 14],
        [17, 19],
        [9, 16, 18],
        [17, 20],
        [11, 16, 21],
        [12, 18, 23],
        [19, 22],
        [21, 23, 14],
        [20, 22]
    ]
    return adjacent[position]


# Function to check if 2 positions have the player on them
# Takes player symbol as input (1 or 2)
# Board list as input
# p1 and p2, the two positions
# Returns boolean values
def isPlayer(player, board, p1, p2):
    if (board[p1] == player and board[p2] == player):
        return True
    else:
        return False


# Function to check if a player can make a mill in the next move.
# Return True if the player can create a mill
def checkNextMill(position, board, player):
    mill = [
        (isPlayer(player, board, 1, 2) or isPlayer(player, board, 3, 5)),
        (isPlayer(player, board, 0, 2) or isPlayer(player, board, 9, 17)),
        (isPlayer(player, board, 0, 1) or isPlayer(player, board, 4, 7)),
        (isPlayer(player, board, 0, 5) or isPlayer(player, board, 11, 19)),
        (isPlayer(player, board, 2, 7) or isPlayer(player, board, 12, 20)),
        (isPlayer(player, board, 0, 3) or isPlayer(player, board, 6, 7)),
        (isPlayer(player, board, 5, 7) or isPlayer(player, board, 14, 22)),
        (isPlayer(player, board, 2, 4) or isPlayer(player, board, 5, 6)),
        (isPlayer(player, board, 9, 10) or isPlayer(player, board, 11, 13)),
        (isPlayer(player, board, 8, 10) or isPlayer(player, board, 1, 17)),
        (isPlayer(player, board, 8, 9) or isPlayer(player, board, 12, 15)),
        (isPlayer(player, board, 3, 19) or isPlayer(player, board, 8, 13)),
        (isPlayer(player, board, 20, 4) or isPlayer(player, board, 10, 15)),
        (isPlayer(player, board, 8, 11) or isPlayer(player, board, 14, 15)),
        (isPlayer(player, board, 13, 15) or isPlayer(player, board, 6, 22)),
```

```
        (isPlayer(player, board, 13, 14) or isPlayer(player, board, 10, 12)),
        (isPlayer(player, board, 17, 18) or isPlayer(player, board, 19, 21)),
        (isPlayer(player, board, 1, 9) or isPlayer(player, board, 16, 18)),
        (isPlayer(player, board, 16, 17) or isPlayer(player, board, 20, 23)),
        (isPlayer(player, board, 16, 21) or isPlayer(player, board, 3, 11)),
        (isPlayer(player, board, 12, 4) or isPlayer(player, board, 18, 23)),
        (isPlayer(player, board, 16, 19) or isPlayer(player, board, 22, 23)),
        (isPlayer(player, board, 6, 14) or isPlayer(player, board, 21, 23)),
        (isPlayer(player, board, 18, 20) or isPlayer(player, board, 21, 22))
    ]

    return mill[position]


# Return True if a player has a mill on the given position
# Each position has an index
def isMill(position, board):
    p = board[position]
    # The player on that position
    if p != 'x':
        # If there is some player on that position
        return checkNextMill(position, board, p)
    else:
        return False


# Function to return number of pieces owned by a player on the board.
# value is '1' or '2' (player number)
def numOfPieces(board, value):
    return board.count(value)


# Function to remove a piece from the board.
# Takes a copy of the board, current positions,
# and player number as input.
# If the player is 1, then a piece of player 2 is removed, and vice versa
def removePiece(board_copy, board_list, player):
    for i in range(len(board_copy)):
        if player == '1':
            opp = '2'
        else:
            opp = '1'
        if(board_copy[i] == opp):
            if not isMill(i, board_copy):
                new_board = deepcopy(board_copy)
                new_board[i] = 'x'
                # Making a new board and emptying the position where piece is
removed
                board_list.append(new_board)
    return board_list


# Generating all possible moves for stage 1 of the game.
# That is, when the players are still placing their pieces.
def possibleMoves_stage1(board):
    board_list = []
    for i in range(len(board)):
```

```
        # Fill empty positions with player 1
        if(board[i] == 'x'):
            # Creating a clone of the current board
            # and removing pieces if a Mill can be formed
            board_copy = deepcopy(board)
            board_copy[i] = '1'

            if (isMill(i, board_copy)):
                # Remove a piece if a mill is formed on that position
                board_list = removePiece(board_copy, board_list, '1')
            else:
                # No mill, so just append the position
                board_list.append(board_copy)

    return board_list


# Generating all possible moves for stage 2 of the game
# That is, when both players have placed all their pieces
def possibleMoves_stage2(board, player):

    board_list = []
    for i in range(len(board)):
        if(board[i] == player):
            adjacent_list = adjacentLocations(i)

            for pos in adjacent_list:
                if (board[pos] == 'x'):
                    # If the location is empty, then the piece can move there
                    # Hence, generating all possible combinations
                    board_copy = deepcopy(board)
                    board_copy[i] = 'x'
                    # Emptying the current location, moving the piece to new
position
                    board_copy[pos] = player

                    if isMill(pos, board_copy):
                        # in case of mill, remove Piece
                        board_list = removePiece(
                            board_copy, board_list, player)
                    else:
                        board_list.append(board_copy)
    return board_list


# Generating all possible moves for stage 3 of the game
# That is, when one player has only 3 pieces
def possibleMoves_stage3(board, player):

    board_list = []

    for i in range(len(board)):
        if(board[i] == player):
            for j in range(len(board)):
                if (board[j] == 'x'):
                    board_copy = deepcopy(board)
                    # The piece can fly to any empty position, not only
```

```
adjacent ones
                    # So, generating all possible positions for the pieces
                    board_copy[i] = 'x'
                    board_copy[j] = player

                    if isMill(j, board_copy):
                        # If a Mill is formed, remove piece
                        board_list = removePiece(
                            board_copy, board_list, player)
                    else:
                        board_list.append(board_copy)
    return board_list


# Checks if game is in stage 2 or 3
# Returns possible moves accordingly
def possibleMoves_stage2or3(board, player='1'):
    if numOfPieces(board, player) == 3:
        return possibleMoves_stage3(board, player)
    else:
        return possibleMoves_stage2(board, player)


# ALL FUNCTIONS NECESSARY FOR AI:


# Class to check if the game is completed, and who won
class evaluate():
    def __init__(self):
        self.evaluate = 0
        self.board = []


pruned = 0
states_reached = 0
alpha = float('-inf')
beta = float('inf')
depth = 2
ai_depth = 3


# Function to invert the board, to train the artificial intelligence
def InvertedBoard(board):
    invertedboard = []
    for i in board:
        if i == "1":
            invertedboard.append("2")
        elif i == "2":
            invertedboard.append("1")
        else:
            invertedboard.append("x")
    return invertedboard


# Function to generate inverted board lists from a list of positions.
def generateInvertedBoardList(pos_list):
    result = []
```

```
    for i in pos_list:
        result.append(InvertedBoard(i))
    return result


# Function to find possible mill counts for a certain player.
def getPossibleMillCount(board, player):
    count = 0

    for i in range(len(board)):
        if (board[i] == "X"):
            if checkNextMill(i, board, player):
                count += 1
    return count


# Function to find if a potential mill is in correct formation
# Return boolean values
def potentialMillInFormation(position, board, player):
    adjacent_list = adjacentLocations(position)

    for i in adjacent_list:
        if (board[i] == player) and (not checkNextMill(position, board,
player)):
            return True
    return False


# Function to get how many pieces can potentially form a mill
def getPiecesInPotentialMillFormation(board, player):
    count = 0

    for i in range(len(board)):
        if (board[i] == player):
            adjacent_list = adjacentLocations(i)
            for pos in adjacent_list:
                if (player == "1"):
                    if (board[pos] == "2"):
                        board[i] = "2"
                        if isMill(i, board):
                            count += 1
                        board[i] = player
                else:
                    if (board[pos] == "1" and potentialMillInFormation(pos,
board, "1")):
                        count += 1
    return count


# Our main function to find solutions for the Game. Uses MiniMax algorithm.
def minimax(board, depth, player1, alpha, beta, isStage1, heuristic):
    finalEvaluation = evaluate()

    global states_reached
    states_reached += 1

    if depth != 0:
```

```
        currentEvaluation = evaluate()

        if player1:

            if isStage1:
                possible_configs = possibleMoves_stage1(board)
            else:
                possible_configs = possibleMoves_stage2or3(board)

        else:

            if isStage1:
                possible_configs = generateInvertedBoardList(
                    possibleMoves_stage1(InvertedBoard(board)))
            else:
                possible_configs = generateInvertedBoardList(
                    possibleMoves_stage2or3(InvertedBoard(board)))

        for move in possible_configs:

            if player1:

                currentEvaluation = minimax(
                    move, depth - 1, False, alpha, beta, isStage1, heuristic)

                if currentEvaluation.evaluate > alpha:
                    alpha = currentEvaluation.evaluate
                    finalEvaluation.board = move
            else:

                currentEvaluation = minimax(
                    move, depth - 1, True, alpha, beta, isStage1, heuristic)

                if currentEvaluation.evaluate < beta:
                    beta = currentEvaluation.evaluate
                    finalEvaluation.board = move

        if player1:
            finalEvaluation.evaluate = alpha
        else:
            finalEvaluation.evaluate = beta

    else:

        if player1:
            finalEvaluation.evaluate = heuristic(board, isStage1)
        else:
            finalEvaluation.evaluate = heuristic(
                InvertedBoard(board), isStage1)

    return finalEvaluation

# HEURISTICS:

# Heuristic that finds number of pieces on the board.
# Lose if less than 3 pieces
```

```
def numPiecesHeuristic(board, isStage1):
    if not isStage1:
        movablePieces = len(possibleMoves_stage2or3(board))
        if numOfPieces(board, '1') < 3 or movablePieces == 0:
            evaluation = float('inf')
        elif numOfPieces(board, '2') < 3:
            evaluation = float('-inf')
        else:
            evaluation = 2 * (numOfPieces(board, '1') -
                              numOfPieces(board, '2'))
    else:
        evaluation = 1 * (numOfPieces(board, '1') - numOfPieces(board, '2'))

    return evaluation


# Heuristic that calculates potential mills as the factor.
def potentialMillsHeuristic(board, isStage1):
    global movablePieces
    evaluation = 0

    numPossibleMillsPlayer1 = getPossibleMillCount(board, "1")

    if not isStage1:
        movablePieces = len(possibleMoves_stage2or3(board))

    potentialMillsPlayer2 = getPiecesInPotentialMillFormation(board, "2")

    if not isStage1:
        if numOfPieces(board, '2') <= 2 or movablePieces == 0:
            evaluation = float('inf')
        elif numOfPieces(board, '1') <= 2:
            evaluation = float('-inf')
        else:
            if (numOfPieces(board, '1') < 4):
                evaluation += 1 * numPossibleMillsPlayer1
                evaluation += 2 * potentialMillsPlayer2
            else:
                evaluation += 2 * numPossibleMillsPlayer1
                evaluation += 1 * potentialMillsPlayer2
    else:
        if numOfPieces(board, '1') < 4:
            evaluation += 1 * numPossibleMillsPlayer1
            evaluation += 2 * potentialMillsPlayer2
        else:
            evaluation += 2 * numPossibleMillsPlayer1
            evaluation += 1 * potentialMillsPlayer2

    return evaluation
```

game.py

```
from main import *
import sys

def AI_vs_human(heuristic):
    board = []
    for i in range(24):
        board.append("x")

    evaluation = evaluate()

    print("STAGE 1")

    for i in range(9):
        printBoard(board)
        finished = False

        # TAKING USER INPUT FOR STAGE 1
        while not finished:
            try:
                pos = int(input("\nPlace a piece: "))
                if board[pos] == 'x':
                    board[pos] = '1'

                    if isMill(pos, board):
                        itemPlaced = False
                        while not itemPlaced:
                            try:
                                pos = int(input("\nRemove a '2' piece: "))
                                if board[pos] == '2' and not isMill(pos,
board) or (isMill(pos, board) and numOfPieces(board, '1') == 3):
                                    board[pos] = 'x'
                                    itemPlaced = True
                                else:
                                    print("Invalid position. Try again.")

                            except Exception as e:
                                print(str(e))
                                print("Invalid input, Try again.")
                    finished = True
                else:
                    print("There is a piece there already. Try again.")
            except Exception as e:
                print(str(e))
                print("Try again. Invalid input.")

        printBoard(board)
        evalBoard = minimax(board, depth, False, alpha, beta, True,
heuristic)

        if evalBoard.evaluate == float('-inf'):
            print("YOU LOST!")
            sys.exit()
        else:
            board = evalBoard.board
```

```
    print("STAGE 2")
    while True:
        printBoard(board)
        # TAKING USER INPUT FOR STAGE 2
        userMoved = False
        while not userMoved:
            try:
                pos = int(input("\nMove a '1' piece: "))

                while board[pos] != '1':
                    print("Invalid. Try again.")
                    pos = int(input("\nMove a '1' piece: "))

                userPlaced = False

                while not userPlaced:
                    newpos = int(input("'1' New Location: "))

                    if board[newpos] == 'x':
                        board[pos] = 'x'
                        board[newpos] = '1'

                        if isMill(newpos, board):
                            userRemoved = False
                            while not userRemoved:
                                try:
                                    pos = int(
                                        input("\nMill formed. Remove a '2'
piece: "))
                                    if board[pos] == "2" and not isMill(pos,
board) or (isMill(pos, board) and numOfPieces(board, "1") == 3):
                                        board[pos] = "x"
                                        userRemoved = True
                                    else:
                                        print("Invalid position")
                                except Exception:
                                    print("Error while accepting input")
                        userPlaced = True
                        userMoved = True
                    else:
                        print("Invalid Position. Try Again.")

            except Exception as e:
                print(str(e))
                print("Invalid entry. Try Again please.")

        if evaluateStage23(board) == float('inf'):
            print("YOU WIN!")
            exit(0)

        printBoard(board)

        evaluation = minimax(board, depth, False, alpha,
                             beta, False, heuristic)

        if evaluation.evaluate == float('-inf'):
            print("YOU LOST!")
```

```
            exit(0)
        else:
            board = evaluation.board


if __name__ == "__main__":
    print("WELCOME TO AI VS HUMAN 9 MENS MORRIS")
    print()
    AI_vs_human(potentialMillsHeuristic)
    # You can also use the number of pieces heuristic (worse results)
    # AI_vs_human(numPiecesHeuristic)
```

Output