CMPT317 Assignment 2 Report

Group description: Yuqing Tan (Beatrice) (yut630, 11119129)

1. Problem description

- The game Pawned is played on a 6*6 chess board, with 6 white pawns on one side and 6 black pawns on the opposite side. It requires two players to take turns to move one pawn at each time (like chess), and the movements of pawns are somewhat similar to chess as well. Each player can only control movement of one color of pawns.
- What pawns can do and cannot do:
  Can move forward one step (not two steps) if the square is not occupied;
  Can attack diagonally only if the diagonal square is occupied by one opposite pawn, the attacked pawn is removed from the board (until the end of game);
  Cannot move diagonally if the diagonal square is empty;
  Cannot move left, right or backwards;
- How to win:
  Be the first one to have one piece reaching the home row of opposite side;
  If both players cannot move, the player with more pieces on board wins, otherwise stalemate;

2. Solution description

- How the game is represented
  The game is represented through Python objects.

  Each game object has the following fields:
  gameState – a list of tuples that describes position of each pawn
  boardSize – the length of a side of the board (which is 6 for this assignment)
  whoseTurn – whose turn to move
  winner – initially no one, but it will be updated when the game reaches terminal state

  Each game object also has the methods for minimax algorithm to use:
  isTerminal() – checks if the game is at terminal state
  isMinNode() – returns true if it's Min's turn to move (Min=White in my implementation)
  isMaxNode() – returns true if it's Max's turn to move (Max=Black in my implementation)
  successors() – returns a list of game states that's reachable from the current state within one step
  utility() – returns the difference between black and white's heuristic evaluation

  Game state:

  List size: boardSize*2 (boardSize = 6 for this assignment)
  Indices [0 , (boardSize-1)]: tuples describing black pawns
  Indices [boardSize , boardSize*2-1]: tuples describing white pawns
  Positive and in-bound tuples represent pieces on board; negative value tuples (i.e. (-1,-1)) represent removed pieces.

  tuples and their positions

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

  Initial position (using 4*4 board as an example)

B1	B2	B3	B4	← Black's home row
W1	W2	W3	W4	← White's home row

The display() function would draw a boardSize*boardSize board and place the pawns according to the game state's description.

- Pure minimax algorithm (without alphabeta pruning)
  It takes a full game tree with limited depth and returns a MinimaxTree (self-defined tree structure).

```
Minimax(node):
If the current node is terminal
        Get the utility, put into a MinimaxTree node and return the node
Otherwise
        Get a list of minimax values MMValues from the current node's successors
        If the current node is max, get the maximum of MMValues as M
        If the current node is min, get the minimum of MMValues as M
        Put M into a MinimaxTree node and return the node
```

The purpose of returning a MinimaxTree rather than a single minimax value, is that it's friendly for debugging (printing out the tree or game boards), and the index of successors which provides the minimax value will be used to generate optimal moves.
Each minimax tree node is a 4-tuple (minimaxValue, indexValue, listOfChildValues, listOfChildNodes)
minimaxValue = minimax value of this node
indexValue = first index inside listOfChildValues that provides the minimaxValue
listOfChildValues = list of minimax values from children nodes
listOfChildNodes = list of children nodes (each is a 4-tuple like this one)

- Alpha beta pruning (added as another version of minimax algorithm)
  Rather than taking a full game tree like pure minimax, it generates successors dynamically as needed.
  Parameters:
  the current game object
  an alpha value (low bound, initially negative infinity)
  a beta value (high bound, initially positive infinity)
  the depth limit
  Return: a MinimaxTree

```
Alphabeta(node, alpha, beta, depth):
If the current node is terminal or depth is 0
        Get the utility, put into a MinimaxTree node and return the node
If the current node is max
        Set M-cur to negative infinity (smaller than any value)
        For each successor of current node
                M-suc = alphabeta(successor, alpha, beta, depth-1)
                M-cur = max(M-cur, M-suc.minimaxValue)
                alpha = max(M-cur, alpha)
                if beta <= alpha:
                        break // minimax value is already decided, ignore the rest children
If the current node is min
        Set M-cur to positive infinity (larger than any value)
        For each successor of current node
                M-suc = alphabeta(successor, alpha, beta, depth-1)
                M-cur = min(M-cur, M-suc.minimaxValue)
                beta = min(M-cur, alpha)
                if beta <= alpha:
                        break // minimax value is already decided, ignore the rest children
Put M-cur into a MinimaxTree node and return the node
```

3. Implementation Description

- Game utility and heuristic evaluation

  If no one wins or checkmates, the game utility will be calculated from:
  (Black's heuristic – White's heuristic)
  black is max, and white is min.
  Aspects of heuristic evaluation:
  (a) Number of pieces on the board
  (b) Sum of steps of all pieces away from home row
  (c) Number of successors (representing mobility)

  heuristic = (a) * 2 + (b) + (c)

  If someone checkmates (checkmate is discussed later in this report), the game utility will immediately become:
  36 - (number of steps to that state), if black checkmates
  - [36 - (number of steps to that state)], if white checkmates
  (36=6^2)

  If someone wins, the game utility will immediately become:
  216 - (number of steps to that state)*6, if black wins
  - [216 - (number of steps to that state)*6], if white wins
  (216 = 6^3)

- Transitions (legal moves) and successors (list of game states)

  The transition function move() can be called by successor() in game object as well as by user.
  Parameters:
  player – black or white
  pawnIndex – corresponding to index in gameState (list of tuples describing pawn positions)
  whereToMove – 3 posibilities
  0: move forward
  1: attack left diagonal
  2: attack right diagonal
  Return: a new game state if the move is legal; otherwise none

```
move(player, pawnIndex, whereToMove):
    If pawnIndex or whereToMove is out of bound
        Return none
    currentPosition = gameState[pawnIndex]
    If currentPosition == (-1, -1)
        Return none // this pawn is already removed from the board
    nextPosition = calculated from currentPosition and whereToMove
    If nextPosition is out of bound (i.e. < 1 or > boardSize)
        Return none
    If whereToMove == 0 and nextPosition is occupied
        Return none
    newGS = duplicate(gameState)
    If whereToMove == 1 or 2
        If nextPosition is occupied by an opposite pawn P
            newGS[P] = (-1, -1) // remove from board
        Else
            Return none
    newGS[pawnIndex] = nextPosition
    return newGS
```

```
successor():
    succ = new list
    for each pawnIndex controlled by whoseTurn
        for each whereToMove in [0,1,2]
```

```
                        newGS = move(whoseTurn, pawnIndex, whereToMove)
                        If newGS is not none
                                Append newGS to succ[]
        Return succ
```

- Decision on the next optimal move

  It takes the depth limit and returns an optimal game state

```
optimal(depth):
        // run alphabeta pruning on the current game object
        alpha = negative infinity
        beta = positive infinity
        minimaxTree = Alphabeta(self, alpha, beta, depth):
        // get successors of current game object
        succ = successors()
        If succ is empty
                Return none
        Else
                // indexValue is the branch that provides the minimax value
                Return succ[minimaxTree.indexValue]
```

- How moves are randomized

  The optimal move will be randomized if there are several choices with the same minimax value. In the MinimaxTree generated from alphabeta pruning, none of the minimax values is accurate except the top one. Therefore instead of running alphabeta pruning on the current game object, I ran it on all of the successors and do the top level of minimax by hand to get a randomized result.

```
randomOptimal(depth):
        succ = successors()
        If succ is empty
                Return none
        For each s in succ // remember s is a game state
                obj = createGameObject(s)
                ab = alphabeta(obj, alpha, beta, depth)
                Append ab to abList // a list of MinimaxTree, same size as succ
        If whoseTurn is max or black:
                mm = max(ab.minimaxValue for ab in abList)
        If whoseTurn is min or white:
                mm = min(ab.minimaxValue for ab in abList)
        For ab in abList
                If ab.minimaxValue == mm
                        Append the index of ab in abList to randList
        rand = a random item in randList // which is an index of successor
        Return succ[rand]
```

- Checkmate detection

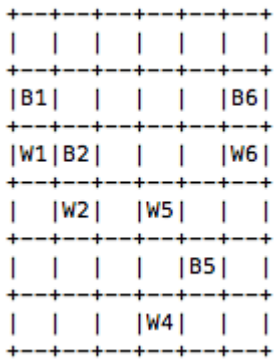
  In my Pawned game, a checkmate is defined as a state where one player is not blocked or threatened by anything to reach the opposite home row, or only one step away from winning. I brought in this idea to make the game search more efficient even after alphabeta pruning – the program will be able detect advantages and disadvantages early.
  A new version of alphabeta pruning stops at terminal state as well as checkmate state. After checkmate, the program follows traditional minimax until terminal state.

  A checkmate detection takes a player and returns a triple (pawnIndex, distanceToWin, type)

```
checkmateP(player):
        If player has a pawn such that no other pawns are in its front or diagonal till the opposite's home row
                return (pawnIndex, distanceToWin, 1) // type 1 checkmate
        If player has a pawn such that it's one step away from the opposite's home row
        and there's one opposite's pawn in its diagonal
                return (pawnIndex, 1, 2) // type 2 checkmate
        return (-1, -1, -1)
```

The optimal() method is enhanced to get the checkmating player(s) and calculate the difference between their distanceToWin's to decide who poses more threat.

Example

White has the next turn W5 and B5 are both checkmating But since B5 has shorter distance to win White needs to capture B5 before moving W5	

- Game automation

The game needs to be automated in order to generate a flow of moves (AI vs. AI) or to be played by human players (human vs. AI or human vs. human).
In my automation, depth limit and the role of human player are passed in as command line arguments.

```
humanPlayer = none
depth = 1
/* parsing command line arguments and updating humanPlayer if needed */
// initialize game object
player = 'W' // white goes first
size = 6
p = Pawned(none, size, player) // no game state passing in
p.display()
while not p.isTerminal()
        nextState = p.randomOptimal(depth) // calculating optimal move
        if nextState is not none and p.whoseTurn == humanPlayer
        // make sure humanPlayer has something to move, otherwise automatically pass the turn
                nextState = none
                while nextState == none
                        input = getUserInput()
                        /* convert input to a list of integers */
                        nextState = move(p.whoseTurn, input[0], input[1])
        nextPlayer = p.switchPlayer()
        if nextState is not none
                p = Pawned(nextState, size, nextPlayer)
        else
                p = Pawned(p.gameState, size, nextPlayer)
        p.display()
```

4. Results and Conclusions

- Example of MinimaxTree and successive boards being printed out

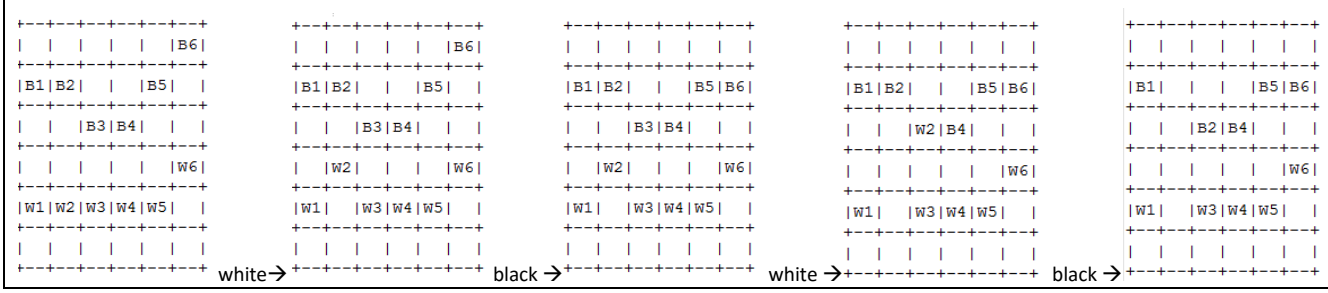
MinimaxTree from pure minimax algorithm:

```
depth 0: 4 0 [4, 0, -1, -1, -4, 4]
        depth 1: 4 0 [4, 4, 4, 4, 5]
                depth 2: 4 0 [4, 0, 0, 4, 0, 0, 0]
                depth 2: 4 2 [-36, 0, 4, 0, 0]
                depth 2: 4 2 [-36, 0, 4, 0, 0]
                depth 2: 4 3 [-36, 0, 0, 4, -36, 0]
                depth 2: 5 4 [-36, 0, 0, 0, 5, -36]
        depth 1: 0 0 [0, 1, 36, 4, 36, 4, 5]
                depth 2: 0 0 [0, 0, 0, 0, 0]
                depth 2: 1 1 [-36, 1, -4, -4, -36, -36]
                depth 2: 36 1 [0, 36, 0, 4, 0, 0, 0]
                depth 2: 4 2 [-36, 0, 4, 4, 0, 4, 0, 0]
                depth 2: 36 1 [-36, 36, 36, 0, 4, -36, -36]
                depth 2: 4 2 [-36, 0, 4, 0, 0, 4, -36, 0]
                depth 2: 5 6 [-36, 0, 4, 0, 0, 0, 5, -36]
        depth 1: -1 2 [36, 36, -1, 36, -1, 36, 36]
                depth 2: 36 2 [0, 5, 36, 36, 0, 0, 0]
                depth 2: 36 2 [0, 0, 36, 0, 0, 0]
                depth 2: -1 3 [-5, -5, -5, -1, -5, -5]
                depth 2: 36 2 [0, 0, 36, 0, 0]
                depth 2: -1 3 [-5, -5, -5, -1, -5, -5]
                depth 2: 36 2 [0, 0, 36, 36, 0, 4, 36, 0]
                depth 2: 36 2 [0, 0, 36, 36, 0, 0, 5, 0]
……
```

MinimaxTree from alphabeta pruning (same tree as the above):

```
depth 0: 4 -1 [4, 0, -1, -1, 4, 4]
        depth 1: 4 -1 [4, 4, 4, 4, 5]
                depth 2: 4 -1 [4, 0, 0, 4, 0, 0, 0]
                depth 2: 4 -1 [-36, 0, 4]
                depth 2: 4 -1 [-36, 0, 4]
                depth 2: 4 -1 [-36, 0, 0, 4]
                depth 2: 5 -1 [-36, 0, 0, 0, 5]
        depth 1: 0 -1 [0]
                depth 2: 0 -1 [0, 0, 0, 0, 0]
        depth 1: -1 -1 [-1]
        depth 1: -1 -1 [-1]
        depth 1: 4 -1 [5, 4]
                depth 2: 5 -1 [0, 5, 0, 0, 0, 4, -36]
                depth 2: 4 -1 [0, 0, 0, 4, 0, 0, 4, 0]
        depth 1: 4 -1 [5, 4]
                depth 2: 5 -1 [0, 5, 0, 0, -36, -36]
                depth 2: 4 -1 [0, 0, 0, 4, 0, 0, 0]
……
```

Game boards

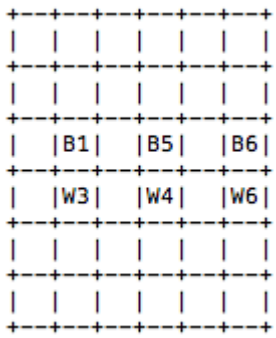


- Depth reached and respond time for each move (on average)

Depth	Pure minimax	Alphabeta pruning
4	0.5 sec	---
5	2 sec	0.5 sec
6	10 sec	2 sec
7	---	5 sec
8	---	20 sec

- Game performance

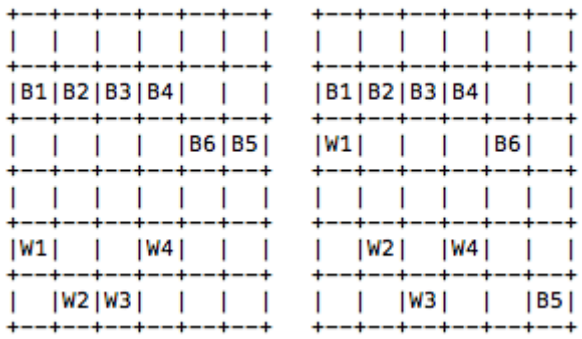
The game has guaranteed possibility for white or black to win as well as draw (stalemate).



From test2_draw.txt

One problem of the game is that AI sometimes doesn't focus on winning the game. After entering checkmate state, it fools around on the board moving other pieces, without winning the game like normal humans would do. I guess it's caused by improper heuristic function together with minimax algorithm – the utility for winning immediately and winning after several steps are the same. I've added some enhancement to make the AI focus on winning the game if it already checkmates (e.g. utility of winning = max utility – number of steps to win; winning early has better utility than winning later).

Of course the most significant problem is that this AI doesn't know or learn any strategies like humans do, it only works on the game tree. Although I can't calculate minimax by head, I know some strategies to beat the AI. For example, burning out 2 columns without the opposite pawns in the right hand side can guarantee a win.



From test3_blackwin.txt

Further improvements on the AI could be:

Store some preset strategies (advantage positions, successive moves, etc.) into the program

5. Implementation

Inside 317a2.tar:

README.txt tells how to run the code
Source code: Pawn.py, runPawn.py
Some test outputs