# Mini Project Report

Zhu Yufan (A0238993J)

First of all, the overall algorithm is based on the minimax search algorithm with alpha-beta pruning. We will use a heuristic function, which shall be discussed in the latter section, to evaluate the desirability of each state of the board from the point of view of the black player. (i.e, the larger the score of a particular board, the more desirable the board is for the black player.) When it is the White player's turn to move, we invert the board to find the most desirable move for the White player and return the most negative score of that board since the score of the board is from the perspective of the black player. The algorithm also incorporates alpha-beta pruning where alpha only updates for the black player and beta is only updated for the white player to reduce the number of boards expand, thus increasing the level searched.

Since the question states a time constraint of making each move within actual 3 seconds, regardless of the machine where the algorithm runs. It is therefore suitable to enhance the algorithm with iterative deepening, where the depth of the minimax search increases gradually within the given time. To achieve this, we first find the time before the algorithm's execution and calculate the end time to be incremented by 2.90. Second, when entering each iteration, namely when evaluating each possible next step, we compare the current time with the supposed end time. If it exceeds 2.90 seconds, we return the best move we have found so far. The 0.1 second left allows the algorithm to finish executing the remaining instructions. The 2.90 seconds limit works fine both on my laptop and desktop, but in case it does not work so well on another machine, please kindly adjust the value to a smaller one, say 2.80s.

In addition, to increase the pruning for the alpha-beta algorithm, moving ordering is also adopted. The possible moves of a certain board are stored in a priority queue sorted according to each board’s score in descending order. This means that the most promising move will be evaluated first, and pruning is more likely to happen.

Memoization is used in the evaluate_board function to further speed up the algorithm by storing the score of each board as a dictionary in the class PlayerAI(). This avoids calling the evaluate_board function repeatedly on the same board at the base cases. Such an approach is viable as the number of the possible state of the 6x6 breakthrough game is still relatively small and we are evaluating the board in various places (when generating possible moves, and when evaluating the base case). I also create a stringify_state function to produce a string representation of a board since the original list representation of the board is not hashable.

On the other hand, the heuristic of my evaluation_board function takes several factors into consideration. The most obvious case is that for the winning state, the score should be a large positive number if the black player wins the game and a large negative number otherwise. Also, if the white pawns are in the second last row, meaning that we are almost losing, such a state should be penalized significantly.

There four following basic factors are considered:

- The piece score in the evaluation function stores the amount by which the number of black pawns exceeds the number of white pawns.
- The position score in the evaluation function stores the amount by which the sum of distances of each black pawn from its baseline exceeds that of white pawns.
- The offensive score in the evaluation function stores the farthest distance from the one black pawn to the baseline.
- The defensive score in the evaluation function stores the closest distance from the one white pawn to the black pawn's baseline.

The larger the more desirable for all four factors. However, unfortunately, the weightage of each factor is determined empirically. I have created several test players which only consider one of the aforementioned factors to play against each other. After much testing, I decided to assign more weightage to the defensive score as the defensive player performs the best among all the test players while assigning a much low weight for the offensive score as it renders the player more prone to move forward recklessly. A comprehensive player is also designed to consider all four above basic stats for further testing. The source code for all five players is attached below in the document.

Moreover, after actually playing the game myself, I have discovered various non-obvious features of the board that we may what to ponder in our evaluation function:

- The danger penalty in the evaluation function stores the number of white pawns that can attack our black pawns. (i.e., which can be captured by white pawns in the next turn).
- In addition, the protected bonus calculates whether an endangered black pawn is being protected by other black pawns, meaning that if the white pawns actually attack our black pawn, the protecting black pawn can attack the white pawn in revenge.
    - The weight for the protected bonus is designed to the higher than the danger penalty. Therefore, if the pawn is endangered without being protected, we will penalize it heavily as we will lose the pawn without being able to revenge. On the other hand, if the pawn is actually endangered but protected at the same time, we will rather reward the state. With such a strategy, we can lure the opponent to eat our pawn such that we can immediately counterattack. This is especially effective when playing against players that prioritize attacking.
- The other feature that came to my attention is when there are empty columns. I found that if we have columns without black pawns, it becomes more difficult to defend in terms of blocking the opponent on the column and attacking the opponent on the neighbouring column. Therefore, I have created this empty column penalty, to penalize boards with empty columns. Furthermore, a more precarious situation is when there is a column with a white pawn but not a black pawn. We will double penalize that case.

**Reference:**

https://www.codeproject.com/Articles/37024/Simple-AI-for-the-Game-of-Breakthrough

I have gained much high-level inspiration from this article, especially for my evaluation function but I have implemented all the algorithms on my own.

```python
import copy
import utils
import math

def piece_evaluation(board):
    number_of_pieces = 0
    for i in range(6):
        for j in range(6):
            if(board[i][j] == 'B'):
                number_of_pieces += 1
            if(board[i][j] == 'W'):
                number_of_pieces -= 1
    return number_of_pieces

# compare the number of possible moves
def mobility_evaluation(board):
    invert_state = copy.deepcopy(board)
    utils.invert_board(invert_state)
    return len(possible_moves(board)) - len(possible_moves(invert_state))

# compare the total number of distance from the base line
def position_evaluation(board):
    total_position_score = 0
    for i in range(6):
        for j in range(6):
            if(board[i][j] == 'B'):
                total_position_score += i
            elif(board[i][j] == 'W' and i ):
                total_position_score -= (5 - i)

    return total_position_score

# find the position of the farthest pawn
def reckless_evaluation(board):
    score = 0
    for i in range(6):
        for j in range(6):
            if board[i][j] == "B":
                score = max(score, i)

    return score

# find the position of the white pawn that is cloest to our baseline
def defensive_evaluation(board):
    score = 6
    for i in range(6):
        for j in range(6):
            if board[i][j] == "W":
                score = min(score, i)

    return score

# find the number of endangered pawns
def cautious_evaluation(board):
    score = 0
    for i in range(6):
        for j in range(6):
            if board[i][j] == "B":
                for pos in ((i + 1, j + 1), (i + 1, j - 1)):
                    if (pos[0] >= 0 and pos[0] < 6) and (pos[1] >= 0 and pos[1] < 6):      # if move takes pawn outside the board
                        if board[pos[0]][pos[1]] == "W":
                            score -= 1
                            break

    return score

# consider all the 5 above factor
def comprehensive_evaluation(board):
    # the wining state and losing state is alway positive and negative infinity
    for i in range(6):
        if board[0][i] == 'W':
            return -math.inf
        if board[5][i] == 'B':
            return math.inf

    piece_score = 0
    position_score = 0
    offensive_score = 0
    defensive_score = 6
    danger_penalty = 0

    for i in range(6):
        for j in range(6):
            if board[i][j] == "B":
                offensive_score = max(offensive_score, i)
                piece_score += 1
                position_score += i
                for pos in ((i + 1, j + 1), (i + 1, j - 1)):
                    if (pos[0] >= 0 and pos[0] < 6) and (pos[1] >= 0 and pos[1] < 6):
                        if board[pos[0]][pos[1]] == "W":
                            danger_penalty -= 1
                            break
            elif board[i][j] == "W":
                defensive_score = min(defensive_score, i)
                piece_score -= 1
                position_score -= (5 - i)

    final_score = piece_score + position_score + 2 * defensive_score + 0.5 * offensive_score + danger_penalty
    return final_score

class PlayerNaive:
    ''' A naive agent that will always return the first available valid move '''
    def make_move(self, board):
        return utils.generate_rand_move(board)

class PlayerDefensive:
    def evaluate_state(self, board):
        for i in range(6):
            if board[0][i] == 'W':
                return -math.inf
            if board[5][i] == 'B':
                return math.inf

        return defensive_evaluation(board)

    def possible_moves(self, board):
        moves = []
        starting_positions = []
        for i in range(6):
            for j in range(6):
                if (board[i][j] == 'B'):
                    starting_positions.append([i, j])

        for x_cord, y_cord in starting_positions:
            for x_move, y_move in [[1, 0], [1, -1], [1, 1]]:
                new_move = [[x_cord, y_cord], [x_cord + x_move, y_cord + y_move]]
                if(utils.is_valid_move(board, new_move[0], new_move[1])):
                    moves.append(new_move)

        return moves

    def make_move(self, board):
        best_score = -math.inf
        best_move = None

        for move in self.possible_moves(board):
            new_state = copy.deepcopy(board)
            utils.state_change(new_state, move[0], move[1])
            new_score = self.evaluate_state(new_state)

            if new_score > best_score:
                best_score = new_score
                best_move = move

        return best_move

class PlayerReckless:
    def evaluate_state(self, board):
        for i in range(6):
            if board[0][i] == 'W':
                return -math.inf
            if board[5][i] == 'B':
                return math.inf
        return reckless_evaluation(board)

    def possible_moves(self, board):
        moves = []
        starting_positions = []
        for i in range(6):
            for j in range(6):
                if (board[i][j] == 'B'):
                    starting_positions.append([i, j])

        for x_cord, y_cord in starting_positions:
            for x_move, y_move in [[1, 0], [1, -1], [1, 1]]:
                new_move = [[x_cord, y_cord], [x_cord + x_move, y_cord + y_move]]
                if(utils.is_valid_move(board, new_move[0], new_move[1])):
                    moves.append(new_move)

        return moves
    def make_move(self, board):
        best_score = -math.inf
        best_move = None

        for move in self.possible_moves(board):
            new_state = copy.deepcopy(board)
            utils.state_change(new_state, move[0], move[1])
            new_score = self.evaluate_state(new_state)

            if new_score > best_score:
                best_score = new_score
                best_move = move

        return best_move

class PlayerPosition:
    def evaluate_state(self, board):
        for i in range(6):
            if board[0][i] == 'W':
                return -math.inf
            if board[5][i] == 'B':
                return math.inf
        return position_evaluation(board)

    def possible_moves(self, board):
        moves = []
        starting_positions = []
        for i in range(6):
            for j in range(6):
                if (board[i][j] == 'B'):
                    starting_positions.append([i, j])

        for x_cord, y_cord in starting_positions:
            for x_move, y_move in [[1, 0], [1, -1], [1, 1]]:
                new_move = [[x_cord, y_cord], [x_cord + x_move, y_cord + y_move]]
                if(utils.is_valid_move(board, new_move[0], new_move[1])):
                    moves.append(new_move)

        return moves

    def make_move(self, board):
        best_score = -math.inf
        best_move = None

        for move in self.possible_moves(board):
            new_state = copy.deepcopy(board)
            utils.state_change(new_state, move[0], move[1])
            new_score = self.evaluate_state(new_state)

            if new_score > best_score:
                best_score = new_score
                best_move = move

        return best_move

class PlayerKiller:
    def evaluate_state(self, board):
        for i in range(6):
            if board[0][i] == 'W':
                return -math.inf
            if board[5][i] == 'B':
                return math.inf

        return piece_evaluation(board)

    def possible_moves(self, board):
        moves = []
        starting_positions = []
        for i in range(6):
            for j in range(6):
                if (board[i][j] == 'B'):
                    starting_positions.append([i, j])

        for x_cord, y_cord in starting_positions:
            for x_move, y_move in [[1, 0], [1, -1], [1, 1]]:
                new_move = [[x_cord, y_cord], [x_cord + x_move, y_cord + y_move]]
                if(utils.is_valid_move(board, new_move[0], new_move[1])):
                    moves.append(new_move)

        return moves

    def make_move(self, board):
        best_score = -math.inf
        best_move = None
```

```python
        for move in self.possible_moves(board):
            new_state = copy.deepcopy(board)
            utils.state_change(new_state, move[0], move[1])
            new_score = self.evaluate_state(new_state)

            if new_score > best_score:
                best_score = new_score
                best_move = move

        return best_move

class PlayerCautious:
    def evaluate_state(self, board):
        for i in range(6):
            if board[0][i] == 'W':
                return -math.inf
            if board[5][i] == 'B':
                return math.inf

        return cautious_evaluation(board)

    def possible_moves(self, board):
        moves = []
        starting_positions = []
        for i in range(6):
            for j in range(6):
                if (board[i][j] == 'B'):
                    starting_positions.append([i, j])

        for x_cord, y_cord in starting_positions:
            for x_move, y_move in [[1, 0], [1, -1], [1, 1]]:
                new_move = [[x_cord, y_cord], [x_cord + x_move, y_cord + y_move]]
                if(utils.is_valid_move(board, new_move[0], new_move[1])):
                    moves.append(new_move)

        return moves

    def make_move(self, board):
        best_score = -math.inf
        best_move = None

        for move in self.possible_moves(board):
            new_state = copy.deepcopy(board)
            utils.state_change(new_state, move[0], move[1])
            new_score = self.evaluate_state(new_state)

            if new_score > best_score:
                best_score = new_score
                best_move = move

        return best_move

class PlayerComprehensive:
    def evaluate_state(self, board):
        for i in range(6):
            if board[0][i] == 'W':
                return -math.inf
            if board[5][i] == 'B':
                return math.inf

        return comprehensive_evaluation(board)

    def possible_moves(self, board):
        moves = []
        starting_positions = []
        for i in range(6):
            for j in range(6):
                if (board[i][j] == 'B'):
                    starting_positions.append([i, j])

        for x_cord, y_cord in starting_positions:
            for x_move, y_move in [[1, 0], [1, -1], [1, 1]]:
                new_move = [[x_cord, y_cord], [x_cord + x_move, y_cord + y_move]]
                if(utils.is_valid_move(board, new_move[0], new_move[1])):
                    moves.append(new_move)

        return moves

    def make_move(self, board):
        best_score = -math.inf
        best_move = None

        for move in self.possible_moves(board):
            new_state = copy.deepcopy(board)
            utils.state_change(new_state, move[0], move[1])
            new_score = self.evaluate_state(new_state)

            if new_score > best_score:
                best_score = new_score
                best_move = move

        return best_move
```