Group 9 :

Kenan Mukendi Kayembe

Furkan Salman

Lab 2 Gomoku, Minimax with Alpha-Beta pruning

Creating the Game Board:

```
# Initializing the board
board_size = 15
board = [[' ' for _ in range(board_size)] for _ in range(board_size)]


# Function to print the board
 Furkan Salman
def print_board():
    print("   ", end="")
    for i in range(board_size):
        print(chr(ord('a') + i), end=" ")
    print()
    for i in range(board_size):
        print("{:2d}".format(i + 1), end=" ")
        for j in range(board_size):
            print(board[i][j], end=" ")
        print()
```

Here first board size and board is initialized as global variables because we will need them throughout the process. The function to print the board is defined as printing numbers to identify the rows and letters to identify the columns and the character 'X' or 'O' is put on the screen based on the players' move information contained in the board matrix.

Checking Win Condition:

```
# Function to check if a given player wins the game.
 Furkan Salman
def check_win(player):
    # Check horizontal, vertical, and diagonal wins
```

```
    # Check horizontal, vertical, and diagonal wins
    for i in range(board_size):
        for j in range(board_size):
            if j + 4 < board_size and all(board[i][j + k] == player for k in range(5)):  # Check horizontal
                return True
            if i + 4 < board_size and all(board[i + k][j] == player for k in range(5)):  # Check vertical
                return True
            if i + 4 < board_size and j + 4 < board_size and all(
                    board[i + k][j + k] == player for k in range(5)):  # Check diagonal (top-left to bottom-right)
                return True
            if i >= 4 and j + 4 < board_size and all(
                    board[i - k][j + k] == player for k in range(5)):  # Check diagonal (bottom-left to top-right)
                return True
    # If no win condition is met, return False
    return False
```

Furkan Salman

Here is the defined function to check if a player wins the game with the current situation of the board. It traverses the board to check if there are any 5 pieces belonging to the player in a row horizontally, vertically, or diagonally. If there are, the function returns True to indicate that the player has won the game. Otherwise, it returns False to indicate that the player has not yet won.

Heuristic, Evaluating the Board:

```
    rows = board_size
    cols = board_size
    score = 0

    # Keep track of evaluated pieces to avoid double counting
    evaluated_pieces = np.zeros((board_size, board_size))

    # Count the number of consecutive pieces in all directions
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == player and evaluated_pieces[r, c] == 0:
                # Horizontal consecutive pieces
                if c <= cols - 5:
                    row_score = 1
                    for i in range(1, 5):
                        if evaluated_pieces[r, c + i]:
                            break
                        if board[r][c + i] == player:
                            row_score += 1
                            evaluated_pieces[r, c + i] += 1
                        else:
                            break
                    if row_score == 5:
                        return 1000
```

```
                score += row_score
```

To calculate the scores for the Minimax algorithm we need a heuristic. Here we used a function to evaluate the board for a player. For every piece on the board, if that place on the board is not visited before, the number of consecutive pieces is checked and summed up to evaluate row, column, and diagonal scores. Because the code to check it is long, above you can only see the horizontal counting of consecutive pieces to visited pieces. After counting visited pieces are marked as evaluated to not visit them again. This way we have a score for the player's pieces. But also we need to take into account of opponent's pieces because the current player doesn't want the opponent to have consecutive pieces. In the same way, a score is calculated for the opponent's consecutive pieces and subtracted from the sum of the player's score. Finally we have a score to evaluate the board and using it we have an idea about who is more likely to win given the situation of the board. We will use this score in the minimax algorithm.

Minimax with Alpha-Beta Pruning:

Base Cases

```
# Definition of the Minimax algorithm with Alpha-Beta pruning
Furkan Salman
def minimax_alpha_beta_pruning(player, depth, alpha, beta, maximizing_player):
    # Check for a terminal state (win or tie)
    if check_win(player):
        return (10000 - depth) if maximizing_player else (-10000 + depth), None
    if check_win('X' if player == 'O' else 'O'):
        return (-10000 + depth) if maximizing_player else (10000 - depth), None
    if depth == 0:
        score = evaluate_board(player)
        return score if maximizing_player else -1 * score, None
```

This function has parameters: player to specify AI or human player, depth to specify a limit of deepening the exploration of moves, alpha, and beta for pruning step, boolean variable maximizing_player to indicate if the current player is maximizing or minimizing.

Here first we have three stopping conditions for recursive calls of the function. First we check if the player wins with the current moves explored if so return with a really high score if maximizing else a really low score. The same check for the opponent. Lastly we check if we reached to depth limit if so we evaluate the board with the function specified before and return the score as positive if maximizing or else negative.

Listing Possible Moves

```
    # Initialize variables
    best_score = float('-inf') if maximizing_player else float('inf')
    best_move = None
    # Generate all possible moves (select all empty cells)
```

```
    moves = []
    for i in range(board_size):
        for j in range(board_size):
            if board[i][j] == ' ':
                moves.append((i, j))
    # Shuffle the moves to avoid always choosing the same moves
    random.shuffle(moves)
```

Here we initialize the best score to minus infinity if maximizing player or else to infinity and best move to none because we don't have any yet. Then we gather all free places on the board to moves list to select possible moves out of this list.

Exploring Moves and Upgrading Alpha-Beta

```
    # Iterate over all possible moves
    for move in moves:
        # Apply the move
        row, col = move
        board[row][col] = player
        # Calculate the score after the move (deepening the search)
        score, _ = minimax_alpha_beta_pruning('O' if player == 'X' else 'X', depth - 1, alpha, beta,
                                              not maximizing_player)
        # Undo the move
        board[row][col] = ' '
        # Update the best score and best move
        if maximizing_player:
            if score > best_score:
                best_score = score
                best_move = move
            alpha = max(alpha, best_score)
            if alpha >= beta:  # minimizing_player won't choose current move because there is a move with lower score
                break
        else:
            if score < best_score:
                best_score = score
                best_move = move
            beta = min(beta, best_score)
            if beta <= alpha:  # maximizing_player won't choose current move because there is a move with higher score
                break
    # Return the best score and best move
    return best_score, best_move
```

For every possible move after doing that move we call the function and by recursive calls we traverse the tree of possible moves and get the score for the current move. Then we undo the move and update the Alpha and Beta parameters at the current depth level. For maximizing player if the score we get by doing this move is bigger than the old best score we update the best move, best score, and Alpha if the current best score is bigger than Alpha, and if the new Alpha is bigger than Beta we don't need to explore other possible moves at this level so we break the loop. For minimizing players if the score we get by doing the move is lower than the old best score we update the best move, best score, and Beta if current best score is lower than Beta and if new Beta is lower than Alpha we don't need to explore other possible moves at this level so it breaks the loop. Then best_score and best_move is returned to where function is called.

This way by assuming human player will act rationally and choose the best moves according to our heuristic function it discovers the best possible move for AI.

Main Game Loop

Player Selects 'X' or 'O' Turn

```
# Define the main game loop
 Furkan Salman *
def main():
    # Initialize variables
    human_player = input("Do you want to play as X or O? ").upper()
    while human_player not in ['X', 'O']:
        human_player = input("Invalid input. Do you want to play as X or O? ").upper()
    current_player = 'X'
    turn = 1
```

Human Player's Turn

```
    # Loop until a player wins or the board is full
    while turn <= board_size * board_size:
        # Print the current board
        print("Turn {}: Player {}".format(turn, current_player))
        print_board()
        # Get the player's move
        if current_player == human_player:
            while True:
                move = input("Enter your move (e.g. 'a1'): ")
                if (len(move) == 2 or len(move) == 3) and move[0] in 'abcdefghijklmno' and \
                        move[1] in '123456789101112131415':
                    row = int(move[1:]) - 1
                    col = ord(move[0]) - ord('a')
                    if board[row][col] == ' ':
                        break
            # Update the board
            board[row][col] = current_player
```

We give first turn to the 'X' player and start looping over turns until a player wins or there is no more free space on board. When it's human player's turn we print current situation of game board and ask for player's move and update the board according to move and change current player to AI for the next turn.

AI's Turn

```
        else:
            # Get the AI's move using Minimax with Alpha-Beta pruning
            # depth limit is 2, initial alpha = inf, initial beta = -inf, current player selected as maximizing
            _, (row, col) = minimax_alpha_beta_pruning(current_player, 2, float('-inf'), float('inf'), True)
            # Update the board
            board[row][col] = current_player
            print("AI chooses", chr(ord('a') + col), row + 1)
```

```
        # Check if the player has won
        if check_win(current_player):
            print_board()
            if current_player == human_player:
                print("You win!")
            else:
                print("AI wins!")
            return
        # Switch to the other player
        current_player = 'O' if current_player == 'X' else 'X'
        turn += 1
    # If no player has won and the board is full, it's a tie
    print_board()
    print("It's a tie!")
```

We get AI's move by calling the minimax function which returns the best move, here the depth parameter is set to 2 for faster decisions, but it can be increased for better decisions at the cost of longer computation time. After getting the best move board is updated and printed, and checked if AI wins, if it doesn't win player changes to human player again and the loop goes on like this until the board is full or one of them wins.

Game Sequence Examples

Depth = 2, human player = 'X', AI = 'O'

Do you want to play as X or O? X

Turn 1: Player X

a b c d e f g h i j k l m n o

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

Enter your move (e.g. 'a1'): g7

Turn 2: Player O

```
  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
 1
 2
 3
 4
 5
 6
 7                    X
 8
 9
10
11
12
13
14
15
```

AI chooses j 8

Turn 3: Player X

```
  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
 1
 2
 3
 4
 5
```

6

7 X

8 O

9

10

11

12

13

14

15

Enter your move (e.g. 'a1'): h7

Turn 4: Player O

a b c d e f g h i j k l m n o

1

2

3

4

5

6

7 X X

8 O

9

10

11

12

13

14

15

AI chooses i 9

Turn 5: Player X

```
  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
1
2
3
4
5
6
7                 X  X
8                       O
9                    O
10
11
12
13
14
15
```

Enter your move (e.g. 'a1'): i7

Turn 6: Player O

```
  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
1
2
3
4
5
6
7                 X  X  X
```

8 O

9 O

10

11

12

13

14

15

AI chooses h 10

Turn 7: Player X

a b c d e f g h i j k l m n o

1

2

3

4

5

6

7 X X X

8 O

9 O

10 O

11

12

13

14

15

Enter your move (e.g. 'a1'): j7

Turn 8: Player O

```
  a b c d e f g h i j k l m n o
1
2
3
4
5
6
7               X X X X
8                   O
9                 O
10                O
11
12
13
14
15
```

AI chooses k 9

Turn 9: Player X

```
  a b c d e f g h i j k l m n o
1
2
3
4
5
6
7               X X X X
8                   O
9                 O   O
```

```
10                  O
11
12
13
14
15
```

Enter your move (e.g. 'a1'): f7

```
  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
 1
 2
 3
 4
 5
 6
 7              X  X  X  X  X
 8                    O
 9                  O     O
10                  O
11
12
13
14
15
```

You win!

Depth = 2, human player = 'O', AI = 'X'

Do you want to play as X or O? O

Turn 1: Player X

```
  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
```

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

AI chooses d 5

Turn 2: Player O

a b c d e f g h i j k l m n o

1

2

3

4

5 X

6

7

8

9

10

11

12

13

14

15

Enter your move (e.g. 'a1'): h8

Turn 3: Player X

  a b c d e f g h i j k l m n o

 1

 2

 3

 4

 5   X

 6

 7

 8   O

 9

10

11

12

13

14

15

AI chooses c 6

Turn 4: Player O

  a b c d e f g h i j k l m n o

 1

 2

3

4

5 X

6 X

7

8 O

9

10

11

12

13

14

15

Enter your move (e.g. 'a1'): h9

Turn 5: Player X

a b c d e f g h i j k l m n o

1

2

3

4

5 X

6 X

7

8 O

9 O

10

11

12

13

14

15

AI chooses b 7

Turn 6: Player O

  a b c d e f g h i j k l m n o

 1

 2

 3

 4

 5   X

 6   X

 7   X

 8   O

 9   O

10

11

12

13

14

15

Enter your move (e.g. 'a1'): h10

Turn 7: Player X

  a b c d e f g h i j k l m n o

 1

 2

 3

 4

5 X

6 X

7 X

8 O

9 O

10 O

11

12

13

14

15

AI chooses a 8

Turn 8: Player O

a b c d e f g h i j k l m n o

1

2

3

4

5 X

6 X

7 X

8 X O

9 O

10 O

11

12

13

14

15

Enter your move (e.g. 'a1'): h11

Turn 9: Player X

```
   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
 1
 2
 3
 4
 5           X
 6        X
 7     X
 8 X                    O
 9                     O
10                     O
11                     O
12
13
14
15
```

AI chooses e 4

```
   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
 1
 2
 3
 4              X
 5           X
 6        X
 7     X
```

8 X O

9 O

10 O

11 O

12

13

14

15

AI wins!

Optimization

Minimax with Alpha Beta pruning is a deterministic algorithm that searches through the entire game tree to find the best move, taking into account the opponent's possible responses. Alpha-beta pruning helps to reduce the number of nodes that need to be evaluated by cutting off parts of the search tree that are known to be irrelevant. While testing the code we realized that depending on the depth size the AI will have a more or less long response time, the accuracy of the AI to predict future moves of the player, and affect the game's pace. For instance, increasing the depth parameter will result in a longer response time whereas a small depth parameter will result in a shorter response time. This is explained by the fact that the method we used (Minimax) is based on Depth-First Search.

A possible fix to the optimization problem is reducing the board size, and reducing the number of possible moves. It is also possible to use another algorithm like Monte Carlo Tree Search in order to get a faster game pace. The Monte Carlo Tree Search method is a stochastic algorithm that relies on random simulations to estimate the value of a decision or strategy. The Monte Carlo Tree Search method can be faster than Minimax with Alpha Beta pruning for some problems because it doesn't need to search through the entire game tree. However, it can also be less accurate and may require more iterations to converge to a good solution.