MIDDLE EAST TECHNICAL UNIVERSITY

NORTHERN CYPRUS CAMPUS

CNG 462

Artificial Intelligence

Assignment 2

Prepared by

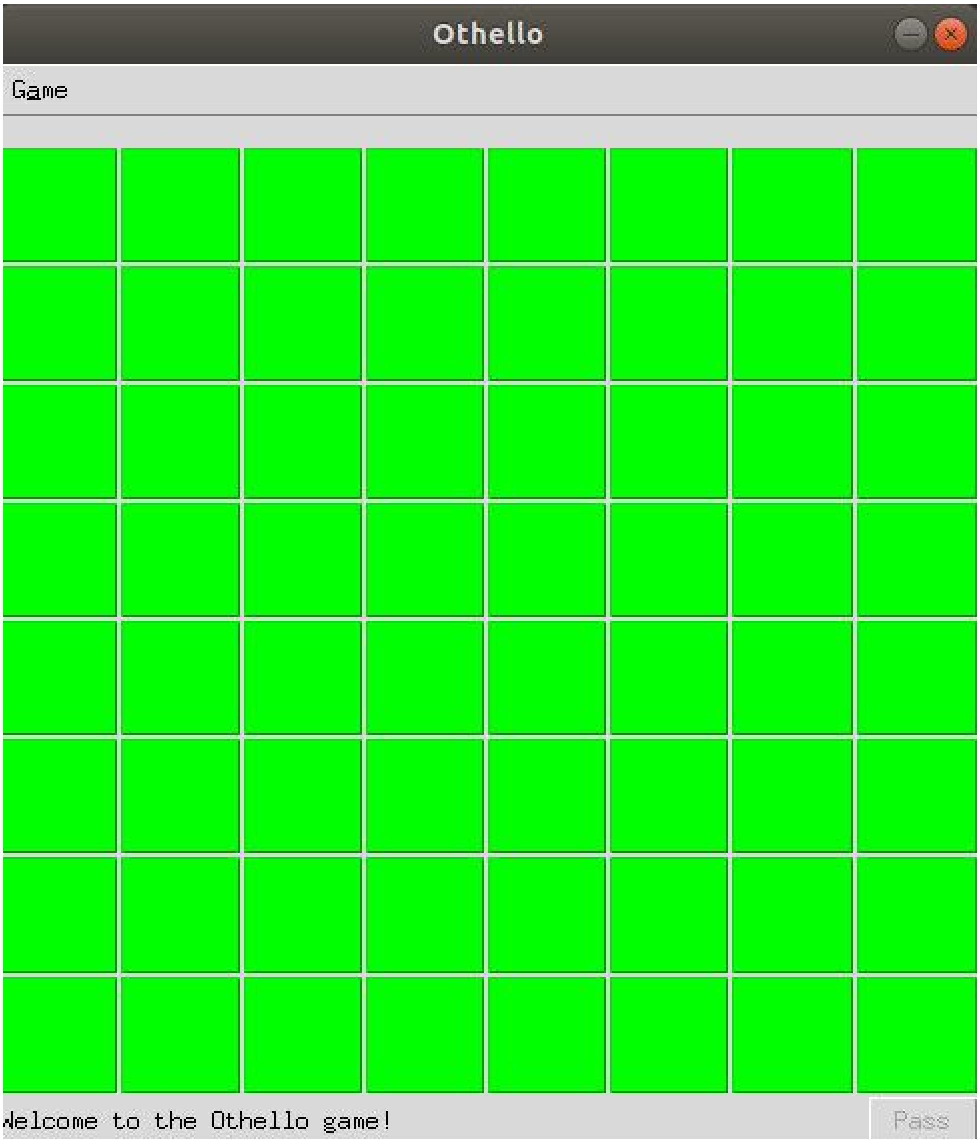
Alper Buğra ÇALIŞIR 2016715

Overview

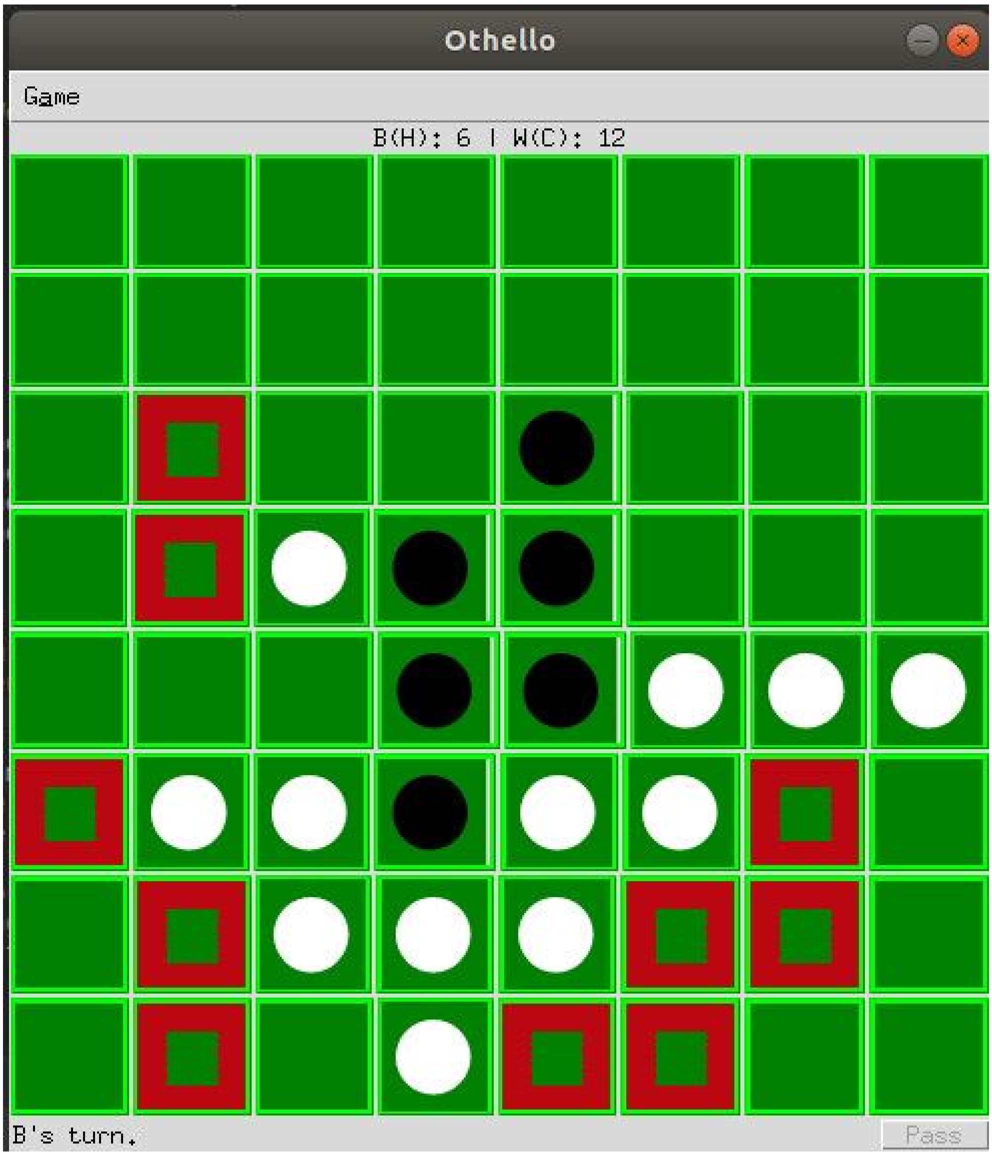
In this assignment I have implemented a game, Othello, in order to practice the skills I have learnt in the class. This game is implemented using Minimax algorithm with Alpha-Beta pruning. I have used 2 different heuristics when implementing the game and let the user will choose it. I used Python 3.6 for my implementation and used *tkinter* module for GUI implementation. My system is Ubuntu 18.04.

Illustrations

Outlook of the game:



After few plays:



Note that **RED** zones indicate possible valid positions to play. They can be opened/closed using menu.

Heuristics

I have chosen my heuristics according to the paper of "An Analysis of Heuristics in Othello " written by Sannidhanam and Annamalai from University of Washington [1].

1) Greedy

The first approach is to use greedy strategy. In greedy strategy players try to maximize the number of coins at any point.

Below is the code I have used:
First if statement check whether the current turn is equal to **W**hite or **B**lack then returns the current number of pieces on the board.

```
def greedy(board, turn):
  if turn == "W":
      return count_pieces(board, "W")
  else:
      return count_pieces(board, "B")
```

2) Coin-Parity

My second approach is to use coin-parity as a heuristic. This heuristic captures the difference in coins between the max player and the min player. We determine the return value as follows:

Coin Parity Heuristic Value =
100* (Max Player Coins –Min Player Coins)/ (Max Player Coins + Min Player Coins)

Below is the code I have written to implement. First it determines who is the maximizing player and vice versa. Then it directly uses this information to find the heuristic value.

```
def coin_parity(board, turn):
  if turn == PLAYER:
    max_player = "W"
    min_player = "M"
  else:
    max_player = "B"
    min_player = "W"

  return 100 * (count_pieces(board, max_player) - count_pieces(board, min_player)) / (count_pieces(board, max_player) + count_pieces(board, min_player))
```

Minimax

I have followed the below pseudocode[2] to make my minimax algorithm.

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):
    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

Below is the code for greedy heuristic with comments.

```
def greedy_alpha_beta_minimax(board, depth, turn, alpha, beta):
  #print("Depth: "+str(depth))
  global node_count  # To change variable as global
  movement = None
  # If the depth is 0 or game has reached an end.
  if depth == 0 or driver.end_game(board):
    return greedy(board, PLAYER), movement
  else:
    valid = driver.valid_positions(board, turn)  # Set of valid positions to play
    ties = []  # List of ties that may arise
    if not driver.has_valid_position(board, turn):
      child_node = copy(board)
      return greedy_alpha_beta_minimax(child_node, depth - 1, change_turn(turn), alpha, beta)[0], movement
    else:
      # if maximizing player
      if turn == PLAYER:
        best_value = -INF
        for position in valid:  # For every valid position tuple
          child_node = copy(board)  # Not to use the original board we create a dummy one
          move(child_node, position, turn)  # Hypothetically move it
          print("MAX-Depth in Situation 2: " + str(depth))
          child_value = greedy_alpha_beta_minimax(child_node, depth - 1, change_turn(turn), alpha, beta)[0]
          print("MAX-Child value: "+str(child_value))
          del child_node  # Then delete it
          # best
          best_value = max(best_value, alpha)
          #print("x: "+str(x))
          #alpha
          alpha = max(alpha, best_value)  # Best value that maximizer can currently guarantee
          #print("alpha: "+str(alpha))
          node_count += 1  # Increase node count by 1 to know that we have visited this node before pruning
          if beta <= alpha:  # Alpha-Beta pruning
            break
          if best_value == child_value:
            print("1-Best Value: " + str(best_value))
            print("1-Position Value: " + str(position))
            ties.append((best_value, position))  # Append the value to the ties list
          elif child_value > best_value:
            best_value = child_value
            print("2-Best Value: " + str(best_value))
            print("2-Position Value: " + str(position))
            ties = [(best_value, position)]  # Get that best value and its position
        best_value, movement = random.choice(ties)
        return best_value, movement
      # if minizing player
      else:
        best_value = INF
        for position in valid:  # For every valid position tuple
          child_node = copy(board)  # Not to use the original board we create a dummy one
          move(child_node, position, turn)  # Hypothetically move it
          print("MIN-Depth in Situation 2: " + str(depth))
          child_value = greedy_alpha_beta_minimax(child_node, depth - 1, change_turn(turn), alpha, beta)[0]
          print("MIN-Child value: " + str(child_value))
          del child_node
          # best
          best_value = min(best_value, child_value)
          #print("y: "+str(y))
          node_count += 1
          # beta
          beta = min(beta, best_value)  # Best value that minimizer can currently guarantee
          if beta <= alpha:  # Alpha-Beta pruning
            break
          if best_value == child_value:
            print("3-Best Value: " + str(best_value))
            print("3-Position Value: " + str(position))
```

```
        ties.append((best_value, position))  # Append the value to the ties list
    elif child_value < best_value:
        best_value = child_value
        print("4-Best Value: " + str(best_value))
        print("4-Position Value: " + str(position))
        ties = [(best_value, position)]  # Get that best value and its position

best_value, movement = random.choice(ties)
return best_value, movement
```

Node Count

To find the number of nodes visited by the algorithm what I have done is to put a global variable and increase that variable by 1 every time that my algorithm is called. This variable is put before the alpha-beta pruning since every time when the pruning happens there will be a break.

References

[1]: Vaishnavi Sannidhanam,Muthukaruppan Annamalai, “An Analysis of Heuristics in Othello ”
Retrieved from
https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf

[2]: Retrieved from
https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/