César Ramírez & Pedro Urbina
Group 2351 Pair 9

LAB ASSIGNMENT 2: GAMES

1. DOCUMENTATION OF MINIMAX + ALPHA-BETA PRUNING.

a. Implementation details:

i. Which tests have been designed and applied to determine whether the implementation is correct?

In order to determine that the implementation is correct we have first tested our algorithm using the tictactoe game. When we achieved that our code worked, we made sure that we obtained the same results than using the simple MinMax strategy, but possibly faster. Then, we temporarily changed the verbose attribute in tournament.py to 1 or 2 so that we can see when pruning was carried out, as we added a message information about pruning when the verbose attribute in the MinimaxAlphaBetaStrategy class is bigger than 0. This, together with the achievements in time we will see later confirmed that our AlphaBeta strategy was working.

ii. Design: Data structures selected, functional decomposition, etc.

The implementation of the MinimaxAlphaBeta algorithm was carried out exactly in the same way as the simple Minimax algorithm. We created a class called MinimaxAlphaBetaStrategy with the same structure as the MinimaxStrategy. This class can be initialized in the same way as the MinimaxStrategy class, with an heuristic, the maximum depth of the search and a verbose attribute in order to activate information messages. The important method is next_move which selects the next movement to be carried out by the player using an instantiation of this class. This method uses the auxiliary methods _min_value and _max_value to achieve the said purpose.

iii. Implementation.

The implementation of the MinimaxAlphaBetaStrategy class was exactly the same as the MinimaxStrategy, changing some details in the next_move, _min_value and _max_value methods.

The main difference is that both _min_value and _max_value methods now receive alpha and beta values in order to prune the tree when necessary. Following the pseudocode given to us, we initialize alpha to -infinity and beta to infinity in the next_move function in the next_move method. Then we start applying _min_value to every successor state and choose the maximum, as the player that starts is always max. In the _min_value function now we have to update the beta value correctly and prune when the minimax value is smaller or equal than alpha. The same changes must be applied to the _max_function, but the rest of the implementation is the same as the MinimaxStrategy class.

iv. Other relevant information.

Code of the MinimaxAlphaBetaStrategy class can be found in the strategy.py document, as we decided not to include the code in order not to fill lots of pages, which would make the final report less readable.

b. Efficiency of alpha-beta pruning.

i. Complete description of the evaluation protocol.

In order to evaluate the efficiency of the alpha-beta pruning we have added some code in order to perform the corresponding measurements. In order to switch between simple minimax and alpha-beta pruning, tournament.py had to be changed constantly.

In order to perform the time measurements we used the time python library, printing the difference between the start and end of the tournament run (tour.run(...) instruction).

In order to measure the improvement in a computer-independent manner, we thought it was a good idea to count the total recursive calls that are done through the execution of the whole program to choose the next state. In order to do that, we created a class variable in both classes MinimaxAlphaBetaStrategy and MinimaxStrategy. Everytime there is a call to the min or max functions, this class variable is incremented by 1. This way we were able to see how many method calls were carried out for each strategy.

After executing many tournaments varying the different heuristics, strategies and maximum depth of the search we obtained the results that follow.

ii. Tables in which times with and without pruning are reported.

Game : Depth : Heuristics used	**Execution time of the tournament with only one repetition.**		**Performance enhancement due to pruning**
	Minimax Strategy (seconds)	**Minimax AlphaBeta Strategy (seconds)**	
Tictactoe : 3 : Dummy	2.008574009	0.6345052719	3.165574973
Tictactoe : 4 : Dummy	8.553754568	1.198735237	7.135649561
Tictactoe : 5 : Dummy	27.1386869	2.078977585	13.05386219
Reversi : 2 : Dummy	10.57495642	9.154441595	1.155172198
Reversi : 3 : Dummy	60.372962	24.06869864	2.508360044

Reversi : 4 : Dummy	419.4633372	71.62647128	5.856261375
Reversi : 2 : PonderationMax	42.93464303	29.80803609	1.440371412
Reversi : 3 : PonderationMax	9588.59343	3797.49496	2.52497858
Reversi : 2 : HeuristicParityMobilityCornersTop2	70.41214538	51.10343504	1.37783586
Reversi : 3 : HeuristicParityMobilityCornersTop2	983.055546	361.7843299	2.717241917

iii. Computer independent measures of improvement.

Game : Depth : Heuristics used against itself	**Sum of the total number of recursive calls to compute the next_state in all the tournament. Tournament with only one repetition.**		
	Minimax Strategy	**Minimax AlphaBeta Strategy**	**Performance enhancement due to pruning**
Tictactoe : 3 : Dummy	14832	3594	4.12687813
Tictactoe : 4 : Dummy	62536	6436	9.716594158
Tictactoe : 5 : Dummy	211144	12006	17.58654006
Reversi : 2 : Dummy	19480	9132	2.133158125
Reversi : 3 : Dummy	111166	26806	4.147056629
Reversi : 4 : Dummy	785378	69076	11.36976663
Reversi : 2 : PonderationMax	14370	10222	1.40579143
Reversi : 3 : PonderationMax	401183	215892	1.858257833
Reversi : 2 : HeuristicParityMobilityCornersTop2	90452	42926	2.107161161
Reversi : 3 : HeuristicParityMobilityCornersTop2	1338998	344422	3.887666874

iv. Correct, clear, and complete analysis of the results.

Both tables show huge enhancements in the performance of the players when alpha-beta pruning is used. This difference in performance clearly increases exponentially on the depth. This is the result we expected, as the bigger the depths we explore, the bigger the parts of the tree are pruned, resulting in less calls to min and max methods which has an obvious correlation with execution time.

We also know that the efficiency of pruning depends on the order in the search, it is better if good movements are explored first. In the worst case there is no improvement and, in the best case, using perfect order we would reduce temporal complexity from $O(b^{d})$, in the simple minimax algorithm, to $O(b^{d/2})$, being ‘b’ the branching factor and ‘d’ the depth of the tree which we have varied in the tables. However in our case the temporal complexity should be $O(b^{3d/4})$ as our heuristics follow a random order of exploration.

So, in conclusion, alpha-beta pruning is always a good approach when trying to minimize the execution time of your player. It also gives the same complete (as the search tree is finite) and optimal results as the ordinary minimax algorithm, supposing, as it is the case that the opponent is also optimal.

v. Other relevant information.

All the code used for this section can be found in strategy.py, demo_tournament_updated.py (the code for measuring time was only implemented for the normal tournament (test = 0)) and demo_tournament_tictactoe.py.

2. DOCUMENTATION OF THE DESIGN OF THE HEURISTIC.

a. Review of previous work on Reversi strategies, including references in APA format.

In order to optimize our heuristics, we mainly used three sources of information:

- Parekh, P., Bhatia, U. S., Sukthankar, K., (n.d.). Othello/Reversi using Game Theory techniques. http://play-othello.appspot.com/files/Othello.pdf
- Cherry, K. A. (2011). An intelligent Othello player combining machine learning and game specific heuristics.
- Sannidhanam, V., & Annamalai, M. (2015). An Analysis of Heuristics in Othello.

The ideas in these documents clearly inspired and guided us through the creation of our final heuristics.

b. Description of the design process:

i. How was the design process planned and realized?

In order to implement the different heuristics, each one programmed their own heuristics and then we made them compete with each other so that we could select the best ones for the tournaments. However, some of the functions such as end_game and combined_based_function were used by the two of us. Most of our tests have been carried out in the demo_tounament_updated.py file, where our heuristic classes are defined, and different ways of testing them in different tournament modalities have been implemented.

Our first approach to design the heuristics was an intuitive approach, we designed functions which counted the difference in the number of pieces of each player, the number of movements that could be done in a given state and another to prioritize the corners. Before researching on the internet we knew that a good heuristic must be a combination of those functions and possibly some others, less intuitive.

When we started getting good results, against the given test heuristics, random, dummy, complex, etc, we started researching on the internet till we found the 3 pdfs cited on section a. We tried to implement the described functions and we tested them against our old heuristics.

After the first tournaments, we had a clear idea of which evaluation functions were good, so we just needed to find out the best weightings for these evaluation functions. In order to do that we tried to automate the process. For that, we created the one_heuristic_against_others function with which we can easily test how good is a given heuristic that we are testing, against a list of other reference heuristics. Then, using this function a lot of times, we could find using nested loops the best ponderations for our heuristics.

ii. Did you have a systematic procedure to evaluate the heuristics designed?

As explained in the previous section, one_heuristic_against_others helped us to automate the process of testing how good a given heuristic is, testing it against a list of reference heuristics. This list of reference heuristics varied throughout the development of the heuristics, usually using the best heuristics we had designed before programming this new heuristic as reference.

iii. Did you take advantage of strategies developed by others? If these are publicly available, provide references in APA format; otherwise, include the name of the person who provided the information and give proper credit for the contribution as "private communication".

The main source of information for creating our heuristics were the references in section a.

c. Description of the final heuristic submitted.

Our final submission consists of three heuristics, 2 of which are the same with different weights of the functions involved, so we have essentially 2 different heuristics. Both of them are weighted combinations of simpler evaluation functions. To find out the best weights for these functions, long executions of the demo_tournament_updated.py have been carried out using the option test = 2 which automatically tries different weights of a given set of evaluation functions and prints the results of confronting each of them with a set of reference evaluation functions.

Our first heuristic is called HeuristicPonderationMax. It is an efficient ponderation of 4 evaluation functions, as two of them go over the successors of the current state we made them into a single loop. The weight given to each evaluation function is the one determined by the previously explained process. The 4 used evaluation functions are:

- result_end_game: calculates the difference between the scores of the 2 players in the actual state.
- maximize_possibly_captured_pieces: calculates the number of pieces the player could eat if it had unlimited moves in the actual state.
- maximize_captured_piece: finds out the maximum number of eaten pieces in that state.
- corners_based_function: returns a percentage of the number of corners the player has.

The other two heuristics are HeuristicParityMobilityCorners1 and HeuristicParityMobilityCorners2. Their underlying philosophy is the same as the HeuristicPonderationMax but using different evaluation functions. The process to obtain the weights of these evaluation functions was the same as the one described before. The 3 used evaluation functions are:

- corners_based_function: returns a percentage of the number of corners the player has.
- parity_function: measures how well is the player doing in respect to the actual score, therefore it is similar to result_end_game.
- best_mobility_function: returns the percentage of the player valid moves over all the valid moves that both players could do in the current state.

These last three evaluation functions were developed following the principles analyzed in the third article referenced in section a. It reinforced even more some ideas that we had in mind at the beginning of our analysis. Ideas such as the importance of capturing the corners during the game or the relevance of each player's valid moves, and the advantage that these would provide the player.

d. Other relevant information.

In order to test the heuristics before uploading them to the different tournaments organized by the teachers, we created our final document 2351_p2_09_ramirez_urbina.py inside the tournament folder. We run it with the file demo_tounament.py in the same way (we suppose) as it is done by the teachers in charge of the tournament. 2351_p2_09_ramirez_urbina.py contains the only code necessary for our 3 best heuristics to work and demo_tounament.py has been edited to create a tournament between the heuristics in the tournament folder, containing 2351_p2_09_ramirez_urbina.py.

We will now include some of the modifications we did in order to program and find out our best heuristics, but a better visualization of the code can be achieved opening the modified files. Changes made to heuristic.py and demo_tournament_updated.py, the file we used to execute all our tests, are now included. However, other edited files such as 2351_p2_09_ramirez_urbina.py and demo_tournament.py, despite they have been changed or completely created, have not been included as they only contain redundant information in order to test our final submitted files to the different tournaments.

Added code in heuristic.py (evaluation functions):

```python
def result_end_game(state: TwoPlayerGameState) -> float:
    """Return game result as if game ended in this state."""
    state_value = 0
    scores = state.scores

    assert isinstance(scores, (Sequence, np.ndarray))
    score_difference = scores[0] - scores[1]

    if state.is_player_max(state.player1):
        state_value = score_difference

    elif state.is_player_max(state.player2):
```

```python
        state_value = - score_difference

    else:
        raise ValueError('Player MAX not defined')

    return state_value
```

```python
def maximize_possibly_captured_pieces(state: TwoPlayerGameState) -> float:
    """Returns how many pieces could the player eat in his turn if he had unlimited moves."""
    state_value = 0
    actual_score = result_end_game(state)

    if state.end_of_game:
        state_value = actual_score

    else:
        # scores[0] => scores player1, scores[1] => scores player2
        successors = state.game.generate_successors(state)

        if state.is_player_max(state.player1):
            for successor in successors:
                state_value += ((successor.scores[0] - successor.scores[1]) - actual_score)
        elif state.is_player_max(state.player2):
            for successor in successors:
                state_value += ((successor.scores[1] - successor.scores[0]) - actual_score)
        else:
            raise ValueError('Player MAX not defined')

    return state_value
```

```python
def maximize_captured_piece(state: TwoPlayerGameState) -> float:
    """Returns the maximum number of pieces that can be captured in the current state."""
    state_value = 0
    actual_score = result_end_game(state)

    if state.end_of_game:
        state_value = actual_score

```

```
    else:
        # scores[0] => scores player1, scores[1] => scores player2
        successors = state.game.generate_successors(state)

        if state.is_player_max(state.player1):
            for successor in successors:
                    state_value = max(state_value, (successor.scores[0] -
successor.scores[1]) - actual_score)
        elif state.is_player_max(state.player2):
            for successor in successors:
                    state_value = max(state_value, (successor.scores[1] -
successor.scores[0])- actual_score)
        else:
            raise ValueError('Player MAX not defined')

    return state_value
```

```
def     ponderation_maximize(state:     TwoPlayerGameState,     p_actual,
p_max_captured, p_sum_captured, p_corners) -> float:
         """This   function   returns   an   efficient   ponderation   of
result_end_game, maximize_captured_piece,
    maximize_possibly_captured_pieces and corners_based_function."""
    state_value = 0
    actual_score = result_end_game(state)

    if state.end_of_game:
        state_value = actual_score

    else:
        # scores[0] => scores player1, scores[1] => scores player2
        successors = state.game.generate_successors(state)
        corners_score = corners_based_function(state)
        score_maximum_captured = 0
        score_possibly_captured = 0

        if state.is_player_max(state.player1):
            for successor in successors:
                     score_maximum_captured = max(score_maximum_captured,
(successor.scores[0] - successor.scores[1]) - actual_score)
                      score_possibly_captured += ((successor.scores[0] -
successor.scores[1]) - actual_score)

        elif state.is_player_max(state.player2):
```

```
            for successor in successors:
                    score_maximum_captured = max(score_maximum_captured,
(successor.scores[1] - successor.scores[0])- actual_score)
                     score_possibly_captured += ((successor.scores[1] -
successor.scores[0]) - actual_score)

        else:
            raise ValueError('Player MAX not defined')

        # final ponderation of the calculated scores
         state_value = actual_score * p_actual + score_maximum_captured *
p_max_captured    +    score_possibly_captured    *    p_sum_captured    +
corners_score * p_corners

    return state_value
```

```
def parity_function(state: TwoPlayerGameState) -> float:
    """Measures how well the player is doing in respect to the actual
score."""
    state_value = 0

    if state.end_of_game:
        state_value = result_end_game(state)

    else:
        player1_score = state.scores[0]
        player2_score = state.scores[1]

         score = 100 * (player1_score - player2_score)/(player1_score +
player2_score)

        if state.is_player_max(state.player1):
            state_value = score

        elif state.is_player_max(state.player2):
            state_value = -score

    return state_value
```

```
def corners_based_function(state: TwoPlayerGameState) -> float:
    """Measures the difference in the number of corners captured."""
    state_value = 0
```

```
    if state.end_of_game:
        state_value = result_end_game(state)

    else:
        height = state.game.height
        width = state.game.width
         corners = [state.board.get((1, 1)), state.board.get((1, width)),
state.board.get((height, 1)), state.board.get((height, width))]

        label_player1 = state.game.player1.label
        label_player2 = state.game.player2.label
        score = 0

        corners_count_player1 = corners.count(label_player1)
        corners_count_player2 = corners.count(label_player2)

        if (corners_count_player1 + corners_count_player2) != 0:
                          score    =   100     *   (corners_count_player1   -
corners_count_player2)/(corners_count_player1 + corners_count_player2)

        if state.is_player_max(state.player1):
            state_value = score

        elif state.is_player_max(state.player2):
            state_value = -score

    return state_value
```

```
def best_mobility_function(state: TwoPlayerGameState) -> float:
    """Returns the percentage of the player valid moves over all the
valid moves that both
    players could do."""

    if state.end_of_game:
        state_value = result_end_game(state)

    else:
        label_player1 = state.game.player1.label
        label_player2 = state.game.player2.label

          player1_valid_moves = state.game._get_valid_moves(state.board,
label_player1)
```

```
        player2_valid_moves = state.game._get_valid_moves(state.board,
label_player2)

        score = 0

        number_player1_valid_moves = len(player1_valid_moves)
        number_player2_valid_moves = len(player2_valid_moves)

        if ((number_player1_valid_moves + number_player2_valid_moves) !=
0):
            score = 100 * (number_player1_valid_moves -
number_player2_valid_moves)/(number_player1_valid_moves +
number_player2_valid_moves)

        if state.is_player_max(state.player1):
            state_value = score
        elif state.is_player_max(state.player2):
            state_value = -score

    return state_value
```

```
def combined_based_function(state: TwoPlayerGameState, functions,
weights) -> float:
    """Auxiliary function used to give a ponderation of the input
evaluation functions."""
    state_value = 0

    if len(functions) != len(weights):
        return state_value

    elif state.end_of_game:
        state_value = result_end_game(state)

    else:
        state_values = []
        for function in functions:
            state_values.append(function(state))

        for (weight, state_value_aux) in zip(weights, state_values):
            state_value = state_value + weight * state_value_aux

    return state_value
```

Code added to demo_tournament_updated.py:
Heuristic classes:

```
class HeuristicDummy(StudentHeuristic):

    def get_name(self) -> str:
        return "dummy"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        # Use an auxiliary function.
        return self.dummy(123)

    def dummy(self, n: int) -> int:
        return n + 4

class HeuristicRandom(StudentHeuristic):

    def get_name(self) -> str:
        return "random"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        return float(np.random.rand())

class HeuristicSimpleEval(StudentHeuristic):

    def get_name(self) -> str:
        return "heuristic_1"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        return simple_evaluation_function(state)

class HeuristicEndGame(StudentHeuristic):
    """Heuristic using result_end_game evaluation function"""

    def get_name(self) -> str:
        return "HeuristicEndGame"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        return result_end_game(state)

class HeuristicMaxCaptureblePieces(StudentHeuristic):
      """Heuristic  using  maximize_possibly_captured_pieces  evaluation
function"""
```

```
    def get_name(self) -> str:
        return "HeuristicMaxCaptureblePieces"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        return maximize_possibly_captured_pieces(state)

class HeuristicBestCapture(StudentHeuristic):
    """Heuristic using maximize_captured_piece evaluation function"""

    def get_name(self) -> str:
        return "HeuristicBestCapture"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        return maximize_captured_piece(state)

class HeuristicCorners(StudentHeuristic):
    """HeuristicCorners"""

    def get_name(self) -> str:
        return "HeuristicCorners"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        return corners_based_function(state)

class HeuristicPonderationMax(StudentHeuristic):
    """Heuristic using ponderation_maximize evaluation function.
           Combines    HeuristicEndGame,    HeuristicMaxCapturablePieces,
HeuristicBestCapture
    and corners_based_function in an efficient way"""

    def get_name(self) -> str:
        return "HeuristicPonderationMax"

    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        p_actual = 0.2
        p_max_captured = 0.1
        p_sum_captured = 0.3
        p_corners = 0.4
            return  ponderation_maximize(state,  p_actual,  p_max_captured,
p_sum_captured, p_corners)

class HeuristicParityMobilityCorners1(StudentHeuristic):
```

```
        """   Combines   corners_based_function,   parity_function   and
best_mobility_function
   with some optimized poderations"""

   def get_name(self) -> str:
       return "HeuristicParityMobilityCorners1"

   def evaluation_function(self, state: TwoPlayerGameState) -> float:
               functions   =   [corners_based_function,   parity_function,
best_mobility_function]
       weights = [0.3, 0.3, 0.4]

       return combined_based_function(state, functions, weights)

class HeuristicParityMobilityCorners2(StudentHeuristic):
        """   Combines   corners_based_function,   parity_function   and
best_mobility_function
   with some optimized poderations"""

   def get_name(self) -> str:
       return "HeuristicParityMobilityCorners2"

   def evaluation_function(self, state: TwoPlayerGameState) -> float:
               functions   =   [corners_based_function,   parity_function,
best_mobility_function]
       weights = [0.7, 0.1, 0.2]

       return combined_based_function(state, functions, weights)
```

Tournament configuration (in demo_tournament_updated.py):

```

# Possible Initial States
intermediate_state_small = (
   [
       '..B.B..',
       '.WBBW..',
       'WBWBB..',
       '.W.WWW.',
       '.BBWBWB',
   ]
)

initial_state = (
```

```
    [
        '........',
        '........',
        '........',
        '...WB...',
        '...BW...',
        '........',
        '........',
        '........'
    ]
)

intermediate_state_large = (
    [
        '........',
        '...B.B..',
        '..WBBW..',
        '.WBWBB..',
        '..W.WWW.',
        '..BBWBWB',
        '........',
        '........'
    ]
)

initial_board_global = intermediate_state_small

repetitions = 1 # tournament repetitions
depth = 2 # search depth used by the search algorithms
max_sec_per_move = 5

# different tournament moddalities can be selected
test = 0 # normal tournament
#test = 1 # only one heuristic tested against others
(tested_against_heuristics)
#test = 2 # optimize one heuristic's ponderations

# here we choose the players (herusitic classes) which will play
against each other in case of normal tournament
strats = {'End': [HeuristicPonderationMax], 'EndMaxBest':
[HeuristicParityMobilityCorners1]}
```

```
# this varibles are used in one_heuristic_against_others, when not running a normal tournament
tested_heuristic = {'0': [HeuristicPonderationMax]}
tested_against_heuristics = {'1': [HeuristicParityMobilityCorners1]}#, '2': [HeuristicParityMobilityCorners2]}
```

Tournament run (in demo_tournament_updated.py):

```
def one_heuristic_against_others(ponderations: bool):
    """This function runs tournaments confronting the heuristic in the tested_heuristic
        dictionary against all the heuristics in the tested_against_heuristics dictionaty"""

   scores_backup = []

   # for each heuristic in tested_against_heuristics a tournament
   for heuristic_key in tested_against_heuristics.items():
       strats = tested_heuristic.copy()
       strats.update([heuristic_key])

       scores, totals, names = tour.run(
           student_strategies=strats,
           increasing_depth=False,
           n_pairs=repetitions,
           allow_selfmatch=False,
       )
       # we save the relevant results of the tournament in scores_backup
       tested_heuristic_wins = list(list(scores.values())[0].values())[0]
       tested_against_name = list(names.values())[1]
       scores_backup.append([tested_against_name, tested_heuristic_wins])

   # print results
   tested_heuristic_name = list(names.values())[0]
   print()
   print()
   print('FINAL RESULTS')
   print('Tested heuristic: ' + tested_heuristic_name)
   print('[won_games : against_heuristic]')
   final_won = 0
```

```
    for result in scores_backup:
        final_won += result[1]
          print('[%d / %d : ' %(result[1], repetitions*2) + result[0] +
']')

    if ponderations:
          print('You heurisitc with ponderations: %.2f, %.2f, %.2f has
won: %d' %(i,j,k,final_won))
    else:
        print('You heurisitc has won: %d' %final_won)


def create_match(player1: Player, player2: Player) -> TwoPlayerMatch:
    """Function to create a match between 2 players."""

    initial_board = initial_board_global

    if initial_board is None:
        height, width = 8, 8
    else:
        height = len(initial_board)
        width = len(initial_board[0])
        try:
                                                            initial_board     =
from_array_to_dictionary_board(initial_board)
        except ValueError:
            raise ValueError('Wrong configuration of the board')

    game = Reversi(
        player1=player1,
        player2=player2,
        height=height,
        width=width,
    )

    initial_player = player1
    game_state = TwoPlayerGameState(
        game=game,
        board=initial_board,
        initial_player=initial_player,
    )
```

```
    return TwoPlayerMatch(game_state, max_sec_per_move=max_sec_per_move,
gui=False)

tour = Tournament(max_depth=depth, init_match=create_match)




# print information about the tournaments that will be run
print()
print('Playing with depth %d. Initial Board:' %(depth))
print(*initial_board_global, sep = "\n")
print()
print(
   'Results for tournament where each game is repeated '
    + '%d (%d x 2) times, alternating colors for each player' % (2 *
repetitions, repetitions),
)
print()



# depending on the value of the test variable different ways of
confronting heuristics will be used
if test == 0 :
   ##### NORMAL TOURNAMENT #####
   print('NORMAL TOURNAMENT')

   start = time.time()
   scores, totals, names = tour.run(
       student_strategies=strats,
       increasing_depth=False,
       n_pairs=repetitions,
       allow_selfmatch=False,
   )
   print('Execution time: %s' %(time.time() - start))
   print()
   print('\ttotal:', end='')
   for name1 in names:
       print('\t%s' % (name1), end='')
   print()
   for name1 in names:
       print('%s\t%d:' % (name1, totals[name1]), end='')
```

```python
        for name2 in names:
            if name1 == name2:
                print('\t---', end='')
            else:
                print('\t%d' % (scores[name1][name2]), end='')
        print()

elif test == 1:
    ##### TESTING A SINGLE HEURISTIC AGAINST OTHERS #####
    print('TESTING A SINGLE HEURISTIC AGAINST OTHERS')

    one_heuristic_against_others(ponderations = False)

elif test == 2:
    ##### TRYING DIFFERENT PONDERATIONS FOR A GIVEN HEURISTIC #####
    print('TRYING DIFFERENT PONDERATIONS FOR A GIVEN HEURISTIC')

    #ponderation_list = [num * 0.10 for num in range(10)]
    ponderation_list = [0.1, 0.2, 0.3, 0.4]

    for i in ponderation_list:
        for j in ponderation_list:
            #for k in ponderation_list:
            if (i + j) <= 1:
                k = 1 - j - i
                #if (i + j + k) <= 1:
                    #z = 1 - (i + j + k)

                """ For each of the ponderations in the ponderation list
we define a class
                 in order to test how this class would work, and then we
test it """
                class HeuristicToMaximize(StudentHeuristic):
                    def get_name(self) -> str:
                        return "HeuristicToMaximize"

                                            def  evaluation_function(self,  state:
TwoPlayerGameState) -> float:
                                            functions  =  [corners_based_function,
parity_function, best_mobility_function]
                        weights = [i, j, k]
```

```
                        return combined_based_function(state, functions,
weights)
                        #return ponderation_maximize(state, i, j, k, z)

                tested_heuristic = {'0': [HeuristicToMaximize]}

                one_heuristic_against_others(ponderations = True)
```