Guidelines – Read Carefully! Please check each problem for problem-specific instructions. You are provided a zip file containing a single folder named xyz007 with the following folders and files:

xyz007
xyz007/mp440.py
xyz007/main.py

You should first rename the folder name xyz007 to be your NETID. If you are working in a team, the folder should be named with both group members' NETIDs, in the format of NETID1_NETID2. The folder name letters can be either upper or lower case. When you are ready to submit, remove any extra files (e.g., some python interpreter will create .pyc files) that are not required and zip the entire folder. The zip file should also be named with your NETID as NETID.zip (or NETID1_NETID2.zip for groups with two members). For this MP, main.py may be removed at the time of submission. On the other hand, this MP requires the submission of a report in PDF format. Please name your PDF file as report.pdf and put it under the main submission folder, i.e.,

NETID/report.pdf or NETID1_NETID2/report.pdf

You are required to write your program adhering to Python 2.7 standards. In particular, DO NOT use Python 3.x. Beside the default libraries supplied in the standard Python distribution, you may use ONLY numpy and matplotlib libraries.

You should only work on mp440.py and do not create additional python files. The file main.py is for you to test your implementations. Note that during grading, your implementation will be called in different ways than what are in main.py, which serve only as minimal sanity checks. You should not use any variable or functions from main.py in your code. All the functions that you should implement are specified in the skeleton mp440.py file.

As mentioned in class, you may form groups of up to two students. Only a single student should submit the solution.

Problem 1 [100 points]. Minimax and alpha-beta pruning for the Reversi game. This MP asks you to implement a rudimentary game engine for playing the Reversi game, an introduction of which can be found at https://en.wikipedia.org/wiki/Reversi.

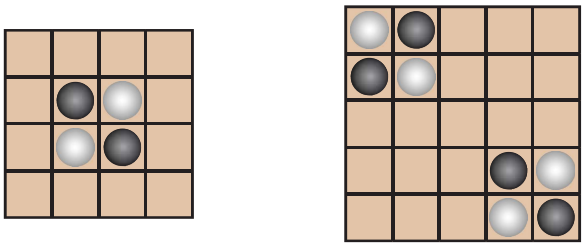

Figure 1: The initial states for reversi games on 4×4 and 5×5 boards used in this MP.

Your main task here is to implement the minimax algorithm and then the alpha-beta pruning heuristic for playing smaller versions (4×4 and 5×5) of the game. The game play used for the MP is as follows. In general, the game board is like a go board with a size of $n \times n$; black and white tokens will be placed on the cells of the board. There are two players, player one (always uses black tokens) and player two (always uses white tokens). Player one always starts first. The players then take turns to make a move by potentially placing a single token of the color used by the current player on the board. For a given board, there are some initial game pieces on the boards. The

standard initial states for the 4×4 and 5×5 boards used in this MP are given in Fig. 1.

Note that in our evaluation, the initial state may be arbitrary. There are several rules used in our game, which may be different from the versions that you have played before. These rules are:

1. When it is a player's turn, the player may place a new token (that the player uses) on the board if the new token to be placed and another token of the same color *tightly sandwich* one or more tokens of the opposite color. In this case, actually placing the token will cause all sandwiched opposite colored token to change color. In the example in Fig. 2, for the left most board, player one can place a black token anywhere on the first row (row 0), i.e. $(0, 0)$, $(0, 1)$, $(0, 2)$, and $(0, 3)$. If the player places a token on $(0, 1)$, then the move ends with the board shown in the middle figure of Fig. 2. If the player places a token on $(0, 2)$, the board will look like the right figure of Fig. 2. On the other hand, all other empty cells on the rows other than row 0 are invalid for player one to place a token.

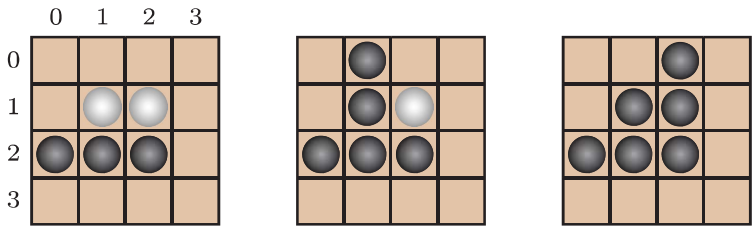

Figure 2: [left] A current state where we assume that player one moves next. [middle] The result if the player places a token on $(0, 1)$. [right] The result if the player places a token on $(0, 2)$.

2. When it is a player's turn, the player can place at most one token on the board. The player then ends their turn. If a player has valid moves (i.e., moves that causes tokens of the opposite color to flip), then the player MUST place a token on the board in a valid manner.

3. After a player either makes a move by placing a token on the board (and performing the resulting color flipping) or if the current player cannot make any valid move, the other player then takes over the game to make a next move.

4. The game ends when neither player can make any further valid moves. At this point, the player with more tokens on the board is the winner. The game may end up in a draw.

5. The *value of the game* in this MP for a given state is the number of black tokens on the board minus the number of white tokens on the board.

a) Basic game rule implementation (20 points). To enable your computation, some basic implementation over the game rules needs to be done. For encoding the state of the game, we use a list of lists. The top level list is a list of the rows and each of the sub-list is a row of the board. An element in the lower level list is a letter that can be either ' ', 'B', or 'W'. For example, the initial state of the 4×4 game shown in Fig. 1 is

```
[[' ', ' ', ' ', ' '], [' ', 'B', 'W', ' '], [' ', 'W', 'B', ' '], [' ', ' ', ' ',
' ']]
```

A first function you need to implement is

`get_move_value(`*state*`,` *player*`,` *row*`,` *column*`)`,

which computes the number of pieces of opponent tokens that get flipped by the intended move at $(row, column)$. This function should not actually modify the state. It computes the flipped tokens *if* a token for the current player is placed at $(row, column)$. This is worth 10 points.

Once a move is decided and needs to be executed, we need to actually execute it. Implement

`execute_move(`*state*`,` *player*`,` *row*`,` *column*`)`

to achieve this. Note that the new state should be returned by this function that does not share memories with the old state. This is worth 5 points.

Then, you are to implement the function

`is_terminal_state(`*state*`)`

that computes whether a given state is a terminal state. Think about when a state is a terminal state for the given game rules. The function should return `True` if the state is a terminal state. Otherwise, it should return `False`. This is worth 5 points.

Lastly, you should implement a function

`count_pieces(`*state*`)`

that computes the current state of the game as the number of black tokens on the board and the number of white tokens on the board. Note that the value of the game is defined as the number of black tokens minus the number of white tokens. So we can compute easily the value of the game using this function.

b) Basic minimax algorithm implementation (30 points). You are now ready to start implementing the minimax (MinMax, MM) algorithm. This should be a direct implementation of the pseudo-code from the book and drill down to the bottom of the game tree. The main function to be implemented is

`minimax(`*state*`,` *player*`)`,

which should return the optimal value of the game, as can be computed by `count_pieces`, for the given state and the player (i.e., the player that is going to make the next move). It should also return the next move for the place, that is the row and column to place a token for the player. The return value should be encoded as a 3-tuple, i.e., $(value, row, column)$. If the player cannot make a move, the $(row, column) = (-1, -1)$.

Then, you should implement the function

`full_minimax(`*state*`,` *player*`)`

that returns the optimal game value and a sequence of player moves that lead to the optimal game, for the given *state* and *player* in the function call. This should be returned as the value of the game and a list of 3-tuples, e.g.,

`(value, [(`p_1`,` r_1`,` c_1`),` `...,` `(`p_k`,` r_k`,` c_k`)])`.

The p_i in a 3-tuple is the player that is making the move. This is not always alternating between 'B' and 'W' (why?). Your implementation should be able to compute solutions for the 4×4 game as shown in the left figure of Fig. 1 using no more than 3 minutes on a modern PC. My basic implementation without any optimization runs in about 25 seconds on a 3 year old laptop PC. Note that your implementation will be tested on other initial states on the 4×4 boards.

c) Evaluating your implementation on the 4×4 game (15 points). With your implemen-

tation of `minimax`, test it on the 4×4 game as shown in the left figure of Fig. 1. In the PDF file to be submitted, provide an optimal game play with all the moves until the game reaches a terminal state. You may simply copy the output from the call to `full_minimax`. Provide the terminal state for the optimal game play. What does the answer mean? Is there a player that is guaranteed to win the game? If so, then which player?

Furthermore, you should report the number of terminal states that are encountered in the process of running `minimax` at the top level.

d) minimax algorithm with alpha-beta pruning (20 points). Augment your minimax implementation to support alpha-beta pruning. Specifically, you should implement the counter parts of `minimax` and `full_minimax` with the updated names `minimax_ab` and `full_minimax_ab`. The input parameters are the same for all these functions. You implementation should be able to solve the 4×4 game in no more than 10 seconds and it should also solve the 5×5 game in 5 minutes (mine runs in 50 seconds). Again, your implementation will be tested on a variety of initial states on 4×4 and 5×5 boards (that are simpler than the default 5×5 game).

e) Evaluating your implementation with alpha-beta pruning on the 4×4 game and the 5×5 game (15 points). Run your algorithm on the two initial games given in Fig. 1. Report the time that is used for the 4×4 game with and without alpha-beta pruning. Also, for both 4×4 and 5×5 games, report the number of terminal nodes that are seen by the procedure, as well as the number of truncation operations (i.e., early return) made by alpha-beta pruning.

Provide the terminal state for the optimal game play, for the 5×5 game. What does the answer mean for the 5×5 game? Is there a player that is guaranteed to win the game?

The output of the provided main.py, with a reference implementation, may look like (a move of $(-1, -1)$ indicates the game reached a terminal state):

```
Printing the initial game state for a 4 x 4 game:
[' ', ' ', ' ', ' ']
['B', 'B', 'B', 'B']
['W', 'W', 'W', 'W']
[' ', ' ', ' ', ' ']

The sate of the game is: (4, 4)

Game value if black places on (3, 2): 2
Game value if white places on (0, 2): 2

State after executing the move of (3, 3) for player one
[' ', ' ', ' ', ' ']
['B', 'B', 'B', 'B']
['W', 'W', 'B', 'B']
[' ', ' ', ' ', 'B']

Running full minimax:
(10, [('B', 3, 0), ('W', 0, 3), ('B', 3, 3), ('W', 0, 1), ('B', 0, 2), ('W', 3, 1), ('B', 0, 0), ('B', 3, 2), ('W', -1, -1)])
Elapsed time: 0.371000051498

Running full minimax w/ alpha-beta pruning:
(10, [('B', 3, 0), ('W', 0, 3), ('B', 3, 3), ('W', 0, 1), ('B', 0, 2), ('W', 3, 1), ('B', 0, 0), ('B', 3, 2), ('W', -1, -1)])
Elapsed time: 0.12700009346
```

f) Performance bonus (7–10 bonus points) We will add your full minimax (w/o alpha-beta pruning) running time for a 4×4 game and your full alpha-beta pruning running time for a 5×5 game and based on the total time, assign 7–10 bonus points to the top 10 winning groups. The assignment will be 10–9–9–8–8–8–7–7–7–7 in increasing order of running time.

A very important note is that your implementation should NOT use any randomization, i.e., when we run your algorithms on the same input, they should always produce the same output. These output should in particular be the same as what you put in your report.

