Final Project Specification

Michelle Cone | Theresa Gebert | Yuan Jiang

mcone@college.harvard.edu | tgebert@college.harvard.edu | yuanjiang@college.harvard.edu

1 Signatures/Interfaces

1.1 Full Interface/Contract

- First of all, we will have a "game" class that includes `check_win`, which checks if the game has been won yet and which player has won. It will take in an interface which is the board of a particular size that the players will interact with. It will also keep track of how many rounds have been played. We will have a `begin_game` method that initializes the game between two players.
- Next, we will have a "board" class, in which its sole function is to display the current state of the board. One method we will write for the board class is a `print_board` function that outputs a printed version of the current board (open spaces, occupied spaces, and if a space is occupied, which player is occupying it). Another method we will have is `update` which takes in a piece and a placement coordinate, and places that piece on the board, and updates the printed board to reflect these changes. The board class will also keep track of the current state of the board using a 2D array.
- We will have a "player" class. Every time we instantiate a player, we also provide them the strategy that they will play. In other words, we will have a "random" player, a "greedy" player, and a "minimax" player. For the player class, we will have a score function (keeps track of the current score for the player), and a `place` function that places the player's move

onto the board (piece and position determined by the player's specified algorithm). This function will reject a move if it is an invalid placement.

- Finally, we will have a "shapes" class and 21 subclasses for each piece. The reason we will need 21 subclasses is due to the fact that each piece is unique and will have a different list of points as a result. The shapes class will take care of the rotate and flip functions which are the same for each piece. We will only need to override the "ID," "size," and "points" for each new shape. Every time we wish to place a piece, we will pass in a reference point, and thus each piece will be represented as an array of tuples (coordinates), with the reference point at the head of the array. Every time a player chooses to place down a piece, the player will call the board and display the new state of the game.

1.2 Things to Consider

- **What types of abstractions will you provide? How will these abstractions reduce the conceptual complexity of your project?**
  We will be implementing all of the components of our game solver as classes and objects. This reduces the conceptual complexity because we are hiding the functions that we will use in each of our classes. For example, the players will be able to rotate the pieces by some degree but they won't be able to see how our program does this.

- **Which values and functions will you expose? Which will you hide?**
  None of our functions will be exposed but some of the values will be exposed. Exposed values include things like the current state of the board, which the player only knows based on what is printed out; the reference point, type, and orientation of a shape, which the player chooses when placing things on the board; the score for each player; and the shapes each player has left.

- **What properties should clients of the component respect (e.g., all input values must be of a certain format)?**
  We don't require that the clients respect any properties because we have asserts at the begin-

ning of each of our functions. If the client does not enter a correct value then there will be an assert failure. Obviously, we expect that they should enter in correct values if they want the game to work, but there shouldn't be any bugs in our code created by players entering invalid values.

- **What properties will the components ensure?** As stated above, for each of our components, at the beginning of our functions, we check that the input is valid. For example, we only allow the player to rotate the shapes clockwise by 90, 180, or 270 degrees.

2 Timeline

We currently have skeleton code for our board, players, and pieces. For the next week, we will be aiming to finish writing one function per night.

1. April 21 – Finish writing classes for board, players, and pieces (including flip and rotate functions).
2. April 22 – Write `valid_moves` function that takes in a single piece and the current board and returns an list of coordinate arrays, where each coordinate array is a possible conformation to place the piece on the board.
3. April 23 – Write `random_move` function for the random player. The function will take in a list of pieces, and return a random piece placement on the current board.
4. April 24 – Write `greedy_move` function for the greedy - Monte Carlo player. The function will take in a list of pieces, and return the current most optimal piece placement on the board.
5. April 25 – Begin writing functions for minimax player. This player will be similar to the greedy - Monte Carlo player but will consider more than just one move ahead in time.
6. April 26 – Write minimax. Finish implementing minimax algorithm. Consider writing code for several different versions (more or less steps ahead in the game, smaller or larger board, etc.)

7. April 27 – CHECKPOINT - Finish polishing up code for all pieces and algorithms. Test to see if everything is working. If not, work on debugging. Send to TF when done.
8. April 28 – Write interface so that we can observe the two players playing against each other.
9. April 29 – Start implementing user-player functionality. Write `user_move` that take in the current board, a piece, and a coordinate, and returns the piece placed on the board. Test to see if this works.
10. April 30 – Make demo video. Start writeup.
11. May 1 – Finish writeup.
12. May 2 – Make sure everything works. Polish up everything. Submit!

3 Progress Report

Please see attached .ipynb file in email to see all the code we have written for our project so far.

4 Version Control

We have all set up github accounts and are working with all our files (code, latex, etc.) in one shared folder.