Help Session: Thursday, November 17th, 2:30pm (in class)

Design Discussion Mini-Assignment Due: Wednesday, November 30th, 2pm
Design Discussions: Wednesday, November 30th - Friday, December 2nd

On-Time Handin: Sunday, December 18, 11:59pm
Late Handin: Tuesday, December 20, 5pm
You cannot use a late pass on this assignment

Regular Demo: cs015_runDemo Othello
Snazzy Demo¹: cs015_runSnazzyDemo Othello

Demos may not work well over ssh!

To install: cs015_install Othello
To hand in: cs015_handin Othello

Introduction

Congratulations on choosing the best CS15 final project! Completing Othello will introduce you to the exciting field of Artificial Intelligence (or AI). Your assignment is to write a program that will allow a person and/or computer to play a game of Othello. Never fear - you do not need any prior knowledge of Othello (i.e., what it is, what it looks like, how to play it, etc.), or of AI, to do this final project.

Table of Contents

Introduction
Table of Contents
Assignment Specifications
The Rules of Othello
Program Design
Bells & Whistles
Coding Othello Incrementally
Othello Minimum Functionality Requirements

¹ Note that these demos were done in an older Java graphics package (Swing), so elements may show up differently if you try to replicate specific parts.
Assignment Specifications

Your program should support three modes of play: human vs. human, human vs. computer, and computer vs. computer. A computer player will determine its moves with a standard artificial intelligence algorithm called "MiniMax."

Your program should display the board and the game pieces graphically using JavaFX. At the beginning of the game, the program should display the initial board and the four initial game pieces (We strongly suggest that you see the demo for a more concrete idea of what the GUI might look like, and how the game works). Also, the user should be able to select either player as a human player or a computer player. Computer players should be able to play at one of the three levels of intelligence. If both players are computers, they may play at different levels. Moves made by a human player should come in the form of mouse clicks. Play alternates between the two players according to the standard rules of Othello - more details on the game itself appear later in this handout. A human player should only be able to make a move when it is their turn. When it is a human player’s turn, the board should indicate valid moves by showing playable squares in a different color. A computer player should be able to make an intelligent move using the MiniMax algorithm and should make this move on a time delay so that moves are visible. The user should be able to use any of the buttons in the game between moves, even when computer vs. computer is running. Pressing Apply Settings again should start a new game with updated players. There should be a reset button that resets the game to the initial 4 piece state. The deterministic checkboxes are not required for full functionality, but are present in the demo to give you a better idea of how computer vs. computer games run.

Before your knees start buckling under the sheer weight of everything on the next few pages, remember we will be having a help session very soon to explain all of the following concepts (e.g., MiniMax) in gory detail. Othello is more conceptually difficult than programming intensive - think very hard about your design before you begin to code.

Also, make sure you read both handouts carefully at least 5 times and play the demo at least 10 times. This will help you grasp the concepts of the project. We know it’s hard to read every word of an assignment handout. We’ve been there. But it’s super important, particularly for this assignment.

The Rules of Othello

The game of Othello is played on an 8 by 8 grid of board spaces. The game pieces (chips) are black on one side and white on the other (though you are free to use any colors you like, as long as they are distinct from one another). Each player is assigned a color. They take turns placing tiles on the board with the player's color face up. The object of the game is to have the most tiles of your color on the board when the game is over. You can flip your opponent's tiles from
their color to your color by "sandwiching" them. A valid move must sandwich at least one of your opponent's tiles. The diagrams on the next pages should make this clear:

**Initial Board and Tiles**
The four initial board pieces are as shown. Either Black or White can go first. Suppose White goes first.

1. **White About to Move**
Valid moves are 3d, 4c, 6e, and 5f. White sandwiches Black's tile at 4d by placing a tile 3d and in doing so flips over Black's tile at 4d.

2. **Black About to Move**
Valid moves are 3c, 3e, and 5c. Black must now try to sandwich one or more of White's tiles. Black places a tile at 3e and sandwiches White's tile at 4e.

3. **White About to Move**
Valid moves are 2f, 3f, 4f, 5f, and 6f. White decides to place a tile at 3f and hence sandwiches two of Black's tiles, one at 3e and one at 4e diagonally. They both flip over to white.

4. **Black About to Move**
Valid moves are 2e, 3c, and 5c.
At this point in the game, Black is behind because he/she has only one piece on the board while White has 6. One
well-placed tile, however, can flip many other tiles in Othello. With each move, a player must sandwich at least one of their opponent's tiles - this constitutes a valid move. You must also consider the scenario where we've sandwiched multiple pieces of the opposing color, and must now flip each of these sandwiched pieces to become our color. For the rest of this handout, when we use the term “move”, we are referring to valid moves only. If no move is possible, then a player must forfeit their turn. The game ends when neither player can move (note that this includes when all the board spaces are filled). At that point, the player with the most tiles of their color on the board is the winner.

The Computer Player

The following is a brief description of how you will teach the computer player to play an intelligent game of Othello. There is a more detailed, supplementary handout that will describe the MiniMax algorithm in greater depth.

The computer generates all the moves it could possibly make given the current board state (locations/numbers of each player’s pieces). For each of these possible moves, the computer player tries out that move and sees what moves the opponent would be able to make from there. This recursive process continues for a finite number of iterations (determined by the intelligence level of the computer player). The computer player can look at the final board states resulting from each of its hypothetical moves and determine which final board state is most desirable. Then, it makes the move that corresponds to that board state. See the AI handout for more detail.

The key part is to determine which board state, resulting from one of the possible moves, is most desirable for a computer player. It is not difficult to develop a mathematical function that will take a given board state and spit out a number representing the board state’s relative score and positional advantage. Such a function could be called evaluateBoard(...). The higher the value returned by evaluateBoard(...) for a given board state, the better a computer player's chances of winning from that board state. However, no one has been able to find a perfect function - one that will always distinguish correctly between board states from which a computer player can win, tie, or be forced to lose. (More details on evaluateBoard(...) will come in the Othello AI handout.)

What is particularly nice about the MiniMax algorithm is that the "intelligence" of the computer player can be controlled, simply by restricting the number of moves it tries to predict. If it were to just predict all of its possible moves from the original board, and stop there, it would be a fairly stupid opponent. If it were to incorporate predictions of subsequent opponent moves, it would be more difficult to defeat. For each level of prediction, the computer will get more intelligent. (side note: The tradeoff here is memory and computing time. Each level you go deeper requires exponentially more time and RAM to compute. Othello is a relatively simple game, and the Sunlab machines are pretty fast, so the AI could probably look ahead further and not get too
slow. But some computers running more complex AIs such as Chess might take minutes—even used to take *hours*—to come up with moves.)

You should be aware that in order to implement the MiniMax algorithm, you will need to write a single recursive method.

More Details on evaluateBoard(...)

Usually, several criteria are used to evaluate the board from both the computer player’s and the opponent’s point of view. Then the difference between the two values (the computer’s value minus the opponent’s value) for a given move is used as a measure of the computer’s relative advantage or disadvantage. The actual numeric value will be based on the number of pieces each player has and where they are located on the board. For instance, a corner square has more weight than a center square. We will supply you with relative weights for each space on the board. However, these are not necessarily the optimal values. You may determine your own weight values if you wish!

Program Design

The Board

The centerpiece of Othello’s design is the actual game board where the players place their pieces. To logically represent an Othello board in memory, we need to specify each spot and the piece that occupies it. However, to see an Othello board on the screen, grid lines need to appear to indicate where the board spaces are, and each piece needs to display itself as a correctly colored circle.

Does a player make moves haphazardly on the game board? No! That player is not likely to do very well. Instead, each player thinks about possible moves and their consequences before actually choosing and making a final move. A person can do this in their head. A computer player, lacking a head but having far more computing power than a simple mammal, can compute the results of many possible scenarios and choose the one which is the most advantageous. Each scenario computed by the computer player should not be seen visually. Only the final move that the computer player chooses should appear graphically on the screen.

There are multiple design options regarding the implementation of the Othello board. One option is to make two different board classes, one that is visual and represented on the screen and one that is only for the computer player to evaluate moves. This relationship between these two types of boards should suggest the use of certain object-oriented concepts that we have studied.
The Othello board should logically be represented as a two-dimensional array of references to pieces - kind of reminiscent of Tetris, huh? As in Tetris, there should be board spaces. You can also choose to make array checking easier by "padding" spaces surrounding the board.

The Players

It may seem as if the three modes of play (human vs. human, human vs. computer, and computer vs. computer) will have to be implemented separately. However, with Java, we can design a simple implementation that takes care of all three modes. How? By using polymorphism.

Imagine a Player interface. A Player should be able to communicate with the game board and then return the player's move. Doing this would obviously involve calling some other methods - i.e. checking if there are any available moves left to make, checking if a given move is valid, checking if the player has won, etc. This Player would also have some properties - for example, color. Now let's define two classes that implement Player: HumanPlayer and ComputerPlayer. These two classes would both define the makeMove() method of the generic Player. For the HumanPlayer, getting the next move involves getting mouse clicks from the user until one corresponds to a valid move. For the ComputerPlayer, getting the next move involves using the AI algorithm, which will be covered in detail in the complementary Othello AI handout.

Game Play

Having all of these lovely classes is wonderful, but how do the players play the game? An impartial Referee which tells players to go in order might be a good idea. The Referee tells a player to move. When that player is done moving, it notifies the Referee, and the Referee then tells the other player to move. This continues, alternating players, until the game is over.

However, the Referee cannot do this in a simple loop. Why? If there is not a pause between each player’s move you won’t see the move take effect on the screen. (Think about two computer players. They are both capable of calculating and making moves so quickly that two computers could play an entire game in a fraction of a second.) To solve this problem, you need to somehow put a pause between each move. How? Use a javafx.animation.Timeline and a private class, TimeHandler, that implements EventHandler. Put the code that tells one player to make a move in the handle(...) method of the TimeHandler. But we don’t want to tell the players to make a move at regular intervals; the time it takes for a player to make a move varies greatly throughout the game. So the first line of your handle(...) method should stop the Timeline. When you want to tell the next player to make a move, call play(). This will wait for a short amount of time before calling the player’s move method. Because the program waits before calling the next player’s move method you will have time to see the result of the last move show up on the screen. If you are confused, don’t worry. This is not a typical or immediately logical use for a javafx.animation.Timeline, but it is
necessary for you to see the results of a move before the next player makes their move. If you'd like help understanding this please come talk to a TA on hours!

**Getting User Input**

When a human player is taking a turn, the user should be able to click on a particular space on the board and place a piece there, if the move is valid. When a computer player is taking a turn, the user should not be able to affect the board.

You'll also want to create a private class that implements `EventHandler` to handle mouse clicks. You can use generics to have an `EventHandler` that handles events of type `MouseEvent`. You have experience with `KeyEvents` from DoodleJump and Tetris and `MouseEvents` from Lab 4. Also make sure to take advantage of JavaFX documentation!

If it is a human player's turn, we would like the squares that are valid moves to turn a different into a color. We would like squares to react to clicks by telling the player which square was clicked. When a computer player starts its turn, however, we would like the squares to not react at all. So, we want our game to know that when it is a computer's turn it shouldn't accept mouse clicks from a human player (the user). What's something you could use to model this binary state of the current player?

**Final Design Note**

The above design tips are for one version of an implementation that we found intuitive to code - i.e., there are many different ways to set up your board spaces and then check if a move is valid. You are free to use or not use any of the above design tips as long as you take us through your decision process during your Othello Design Discussion and in your code comments. Make sure that your use of classes is not excessive but sufficient - a class and the concept it models should go hand-in-hand. The existence of every class in your program should be justified by its contribution to the way you implement the program. Also, remember to have an intuitive yet strict division of labor among your classes - i.e., there should not be three special cases for the three modes of play, and neither player (human or computer) should need to know whether its opponent is human or computer.

**Bells & Whistles**

There is plenty of room for creativity in this assignment. **Remember:** First get your program to meet the specs; then, if time permits, go ahead and add Bells & Whistles!

Play with the snazzy demos for good ideas! Here are some additional suggestions for Bells & Whistles:
- Animated piece flipping
- Keyboard shortcuts
- Display the last move made by a player
- Custom board weights
- Non-deterministic MiniMax algorithm (as seen in the regular demo)
- Drop-down menus
- Snazzy graphics
- Custom player colors

Anything you can think of! (except music, since department machines often don’t support sound)

**Coding Othello Incrementally**

This is a complex program with many different parts to it. The following is a suggested list of things to do - note: you may wish to do more than what is mentioned here.

1. Play the demo 10 times and make sure you have a complete understanding of the rules of Othello - particularly what constitutes a valid move and when the game is over. It is best to run the game human vs. human first to learn about what moves are valid and which aren’t.
2. Make sure you fully understand the computer player’s intelligence algorithm and the algorithms you will develop and use to check if any moves are available, to check if a move is valid, to check if a player has won, etc. Start writing pseudocode snippets that you can then translate into more formal pseudocode for the design checks.
3. Determine what classes and data structures you will use to implement these algorithms and make sure you fully understand how to manipulate them.
4. Determine the set-up of your GUI and the JavaFX classes it will use. Check the Javadocs for any classes you might need that you are not familiar with.
5. Design your entire program using the algorithms, data structures, and JavaFX classes you have decided on. Keep in mind efficiency considerations, the power of polymorphism, and that every class should have a specific purpose. When designing your program, make sure that no class seems superfluous, but also that you aren’t trying to cram everything into as few classes as possible.
6. Get your GUI up and running. In this step, the user should be able to choose one of the three modes of play (human vs. human, computer vs. human, or computer vs. computer) and a difficulty level for each computer player. The user should also be able to place tiles of the appropriate color on the board (though the playing of the game - surrounding your opponent’s tiles and flipping them - may not yet be supported).
7. Get human vs. human Othello working. This involves implementing the algorithms to check if any moves are available, to check if a move is valid, to check if a player has won, etc.
8. Get the computer player up and running. This involves implementing the MiniMax algorithm that takes into consideration board space weights. The design of your other classes should allow you to add this without making huge changes to other parts of your program.

We would highly recommend that you implement human vs. human play including move and game over checking and turn taking before trying to incorporate computer play, as computer play requires functional game play methods in addition to a complicated intelligent strategy. Think carefully about how you plan on tackling steps 7 and 8--these are big steps that should be broken down into smaller pieces that you test one at a time.

Othello Minimum Functionality Requirements

Your Othello project will have to do the following things in order for you to meet minimum functionality (REMINDER: you have to achieve minimum functionality on all projects to pass the course, and since this is the final project, you cannot hand in again!)

MF requirements are not the same as the requirements for full credit on the project. You should attempt the full requirements on every project to keep pace with the course material. An ‘A’ project would meet all of the requirements enumerated in the assignment specification section of the handout and have good design and code style.

To meet minimum functionality for Othello:

- Human vs. Human games run according to the rules.
  - Only the current player can make a move, and moves must be valid.
  - Pieces flip colors when a sandwich is made.
- Computer vs. Computer and Computer vs. Human games can both be run.
  - Only valid moves are made and pieces flip colors properly.
  - For games with a Computer player, there is a timeline delay so moves can be seen on screen.
- The computer player uses a recursive algorithm to choose its best move.
  - Move choices are made based on future possible configurations of the game.
  - Level 1 intelligence works correctly, and chooses optimal moves that take the number of pieces flipped into account--moves are not made randomly.

Good luck on your journey toward AI mastery, and remember:

**Start Early... Start Today... Start Yesterday!**

- William Shakespeare