

YOUR NAME: _____ DATE: _____

LAST FOUR DIGITS OF YOUR UF-ID: ____ ____ ____ ____ Please Print Clearly (Block Letters)

YOUR PARTNER'S NAME: _____ DATE: _____

LAST FOUR DIGITS OF PARTNER'S UF-ID: ____ ____ ____ ____ Please Print Clearly

Date Assigned: 22 March 2013 IN CLASS

Date Due: 05 April 2013 E-SUBMISSION of Parts II and III

In this homework assignment you may work in groups of two persons only. You may not copy from others, and you may not copy code from the Internet, textbook, or other sources.

However, you may study with others or read your textbook to determine general solutions. Then you must complete the problems as your own work, not copying others' work.

Questions about this homework should be addressed to your TA first. You can find your TA's email, office hours, etc. at the class website: <http://www.cise.ufl.edu/~wchapman/COP2800/officehours.html>

This homework has three parts: (I) Vocabulary Questions, (II) Regular Program, (III) Advanced Program. There is no penalty for guessing.

Part I. Vocabulary Questions

[10 points total]

Vocabulary: (terms you need to know to discuss the subject intelligently) – Define the following terms using 1-3 sentences (and a diagram, if needed): **[2 points each]**

- a. *Singly Linked List*
- b. *Doubly Linked List*
- c. *Array index(es)*
- d. *Abstract class* (in Java)
- e. *AbstractLinkedList* (in Java)

Use your text editor (Notepad++) to generate a file called "Part1.txt". Include this file in the ZIP file along with the code for Parts II and III.

You **must** have in the upper right-hand corner: (i) "COP2800-S13-HW4-PartI", (ii) your name, and (iii) last four digits of your UFID.

Part II. Regular Program

[30 points total]

TASK: Create a Java Program that plays TicTacToe using array operations (this assignment is the second half of the TicTacToe game begun in Assignment 3). Like Assignment 3, this one is designed to make you think, so most of the code is not provided. You have to do more work here... but we allow you to work in groups of two, so that should make it easier for you.

PROGRAMMING PROCEDURE:

- (1) We will be using your Assignment 3 code as a starting point for this assignment. If your code did not work properly, you may download a correct solution to Assignment 3 from this link:

<http://www.cise.ufl.edu/~wchapman/COP2800/misc/Assignment3-solution.zip>

- (2) Make a public Java *Class* called **IntelligentTicTacToe**, with methods **makeMove**, **assessOffensiveOpps** and **assessDefensiveOpps**.

Note: assessOffensiveOpps and assessDefensiveOpps are helper methods for makeMove. (They implement a block of functionality that is called several times.)

makeMove: This method automatically chooses a best move for the current player using the procedure described below, and makes this move on the game board by calling **updateTTT**. This method should return boolean true if a move was made, or false if it is no longer possible to win (a tie has occurred, or the opponent has won). **updateTTT** should not be called if this method returns false.

makeMove should create two 2D integer arrays whose sizes are equal to the size of the game board: *defensiveOppsArray* and *offensiveOppsArray*. The elements of these arrays store the defensive and offensive strategic values of playing each position.

We observe that every position on a TicTacToe game board is an element of 2, 3, or 4 **victory paths**. The center piece of odd-sized boards is an element of 4 paths: 2 diagonals, 1 row, and 1 column. Other diagonal positions are members of 3 paths: 1 diagonal, 1 row, and 1 column. All other positions belong to only 2 paths: 1 row, and 1 column. A good defensive TicTacToe strategy is to maximize the number of victory paths blocked by playing each position, with consideration to how much progress the opponent has made along each path. Likewise, a good offensive strategy is to maximize the number of unblocked potential victory paths advanced by playing each position, with consideration to how much progress you have made along each path.

The details of determining the value of blocking or advancing each path will be handled by the helper methods **assessDefensiveOpps** and **assessOffensiveOpps**, respectively. The responsibility of **makeMove** is to loop over every position (*i,j*) in the game board, determine all the paths (*i,j*) is a member of, call the helper methods on each path, sum the return values from the helper methods, and store the result in the correct OppsArray variable (*defensiveOppsArray* or *offensiveOppsArray*). Already played positions should have a value of 0 in the OppsArray variables. Refer to the example on page 4 to see this algorithm expressed visually.

After building the OppsArray variables, **makeMove** should sum the two arrays to produce a third 2D array, *sumOppsArray*. It should then select a best move by selecting the maximum from *sumOppsArray*, and call **updateTTT** to play that move. If there is a tie for the maximum, **makeMove** should select the first occurrence in a top-bottom, left-right traversal of the board. Victory is impossible if all moves have a value of 0.

```
public static boolean makeMove() {  
    ...code for the method goes here...  
}
```

assessDefensiveOpps: This helper method determines the value of blocking a path, assuming the current player is *sym*. The following rules should be used to assign a value to path.

Rule	Description	Return	4 x 4 Example (sym = 'O')
1	Path is empty.	1	* * * *, return 1
2	Path is already blocked.	0	* X * O, return 0
3	Path is not blocked, and contains N opponents.	N+1	X X * *, return 3
4	Critical Move: path is not blocked, contains N opponents, and opponent will win on next move	N+11	X X X *, return 14

```
public static int assessDefensiveOpps(char[] path, char sym){
    ...code for the method goes here...
}
```

assessOffensiveOpps: This helper method determines the value of advancing along a path, assuming the current player is *sym*. The following rules should be used to assign a value to path.

Rule	Description	Return	4 x 4 Example (sym = 'O')
1	Path is empty.	1	* * * *, return 1
2	Path is blocked by opponent.	0	* X * O, return 0
3	Path is not blocked by opponent, contains N of my symbols.	N+1	O O * *, return 3
4	Critical Move: path is not blocked, contains N of my symbols, and I will win by playing this path.	N+101	O O O *, return 104

```
public static int assessOffensiveOpps(char[] path, char sym){
    ...code for the method goes here...
}
```

- (3) Override the **promptUserTTT** method you inherited from **UserTicTacToe**, and make the necessary modifications so that the computer plays as 'O', and the user plays as 'X'. Your method should also return immediately if a victory or tie occurs. (Hint: Copy+paste your old **promptUserTTT** method into your new **IntelligentTicTacToe** class. You should only have to make minimal modifications to satisfy this requirement.)
- (4) Create a test class **TestITTT** that invokes your **IntelligentTicTacToe.promptUserTTT** method.

COLUMN

0	1	2	3
0	*	*	*
1	*	*	*
2	*	O	*
3	*	*	*

ROW

```
defensiveOppsArray[2][2]=
assessDefensiveOpps( {'*', '*', '*', '*'}, 'X') + //1
assessDefensiveOpps( {'*', '*', '*', '*'}, 'X') + //1
assessDefensiveOpps( {'*', 'O', '*', '*'}, 'X'); //2

offensiveOppsArray[2][2]=
assessOffensiveOpps( {'*', '*', '*', '*'}, 'X') + //1
assessOffensiveOpps( {'*', '*', '*', '*'}, 'X') + //1
assessOffensiveOpps( {'*', 'O', '*', '*'}, 'X'); //0
```

defensiveOppsArray

3	3	2	4
2	4	4	2
3	0	4	3
4	3	2	3

offensiveOppsArray

3	1	2	2
2	2	2	2
1	0	2	1
2	1	2	3

Example shown in
class on 3-22

- (5) You should verify that your program works correctly by printing out the *defensiveOppsArray* and *offensiveOppsArray* variables for a variety of board states. We have created a web utility that will allow you to see if your arrays are correct.

<http://www.cise.ufl.edu/~wchapman/COP2800/misc/IntelligentTicTacToe/simulator.php>

We have also printed the result of the computer playing against itself for board sizes 3x3 up to 9x9, with the corresponding values of the *defensiveOppsArray* and *offensiveOppsArray* variables. These games can be viewed from the following link:

<http://www.cise.ufl.edu/~wchapman/COP2800/misc/IntelligentTicTacToe>

Part III. List-Based Assessment

[30 points total]

Programming with Lists. The objective of this programming assignment portion is to augment the functionality of the program you wrote in Part II, using a list representation to determine and recall already-computed offensive and defensive moves.

- Modify your TicTacToe program from Assignment 4, Part II, to accept another input character “C”, which will signify the end of a game, and will start a new game (with the same board size) without terminating the program or clearing any of the ArrayLists that we discuss below.
- Write a new method ***convertTTTArrayToString*** that inputs the gameboard array and scans it in row-major order (row 1 first, then row 2, and so forth). While the scanning is going on, your method should concatenate the array rows to form a list. So, for a 3x3 TTT board, if row 1 reads (X, *, X), row 2 = (O, O, *), and row 3 = (O, X, O), with quotes around characters omitted for clarity, then the result of ***convertTTTArrayToString*** will be the string representation *TTTstate* = “X * X O O * O X O”. Now, append the current player’s symbol (X or O) to the string, as

```
TTTstate = TTTstate + " @ " + current_player_symbol;
```

```
public static String convertTTTArrayToString() {
    ...code for the method goes here...
}
```

- (3) Create a helper class **BoardState**, which has 3 public nonstatic variables: **TTTState**, **defensiveOppsArray**, and **offensiveOppsArray**. This class corresponds to 1 unit of your game board's memory. It holds the string representation of a game board, and the corresponding pre-computed OppsArray variables. The class specification is as follows:

```
public class BoardState{
    public String TTTState;
    public int[][] defensiveOppsArray;
    public int[][] offensiveOppsArray;
}
```

- (4) Add a public static variable, **memory**, to your **IntelligentTicTacToeArray** class. This variable will be an `ArrayList<BoardState>`, and will keep a record of all the previously encountered board configurations, along with their computed OppsArrays.

```
public static ArrayList<BoardState> memory = new ArrayLi...;
```

- (5) Make the following two changes to your **makeMove** method from Part II:

Before building the **defensiveOppsArray** and **offensiveOppsArray** for a game board, create a TTTState of the current game board, iterate through the **memory** `ArrayList` and check to see if any of it's **BoardState** elements already have a **TTTState** that is equal to this value. If you find a match, you can use the OppsArrays from the match. If you don't, you will need to compute the OppsArrays.

After computing the **defensiveOppsArray** and **offensiveOppsArray** for a game board, create a new instance of **BoardState**. Store your computed OppsArrays into the corresponding variables in this instance of **BoardState**. Also use **convertTTTArrayToString** to place the TTTState representation of the game board in the **TTTState** variable. Add this instance of **BoardState** to your memory `ArrayList`, so in the future, you will not need to recompute these OppsArrays.

Part IV. Extra Credit (text file like Part I, submit online)

[10 points each]

EC-1. What is the complexity of the array-based program in Part II in terms of number of operations (array accesses, additions, multiplications) as a function of board size N ? Justify your answer with a graph and/or equations.

Please do this part electronically.

Use your text editor (Notepad++) to generate a file called "Part4.txt". Include this file in the ZIP file along with the code for Parts II and III.

You **must** have in the upper right-hand corner: (i) "COP2800-S13-HW4-PartIV", (ii) your name, and (iii) last four digits of your UFID.

EC-2. What is the complexity of the list-based program in Part III in terms of number of operations (array accesses, additions, multiplications) as a function of board size N ? Justify your answer with a graph and/or equations.

EC-3. Why might you *not* want to use the approach in Part III (with the lists) for your main technique to determine the *OffensiveOppsArray* and *DefensiveOppsArray*? (Hint: The complexity grows as 2^M , where M is the number of possible moves...) Explain your answer in detail with equations and some numerical examples.

Electronic Submission of Parts II and III. Put all files you created in Parts I through III in a single ZIP file. Your ZIP file should contain all the files specified in Parts I through III. Submit this ZIP file electronically per the instructions at:

<http://www.cise.ufl.edu/~wchapman/COP2800/submit>

Part V. Evaluation of Submitted Code

Grading: Code does not compile or run	= 0 points.
Code compiles but does not run	= < 20 percent of points.
Code runs but wrong results	= 21 to 50 percent of points.
Code runs with correct results but no documentation (e.g., green comments in Part II)	= 51 to 70 percent of points.
Code compiles and runs, correct results, documentation present	= 71 to 100 percent of points.

