CS W1001 Introduction to Computer Science

Homework 5 (12 points)

"Intelligent Competitions:"

The Rigor of Theory vs. the Enchantment of AI

Due Date:

Thursday, April 27 (the last class of the semester).

For this assignment, hardcopies (only) of the first Part are required, and both hardcopies and electronic submissions of Part II are required. See Part II below on details of what to submit, which includes a README file. Hardcopies are also due in class. The course web page provides directions for electronic submission.

Reading:

"An Invitation to Computer Science," GEB: "Little Harmonic Labyrinth" story, and the beginning of chapter five on recursion (just the first 4 pages). If you want to read more you'll have to get the book -- we included the table of contents in the photocopies in order to advertise to you the rest of the book. Note that the artificial intelligence sections of that book (not included in the photocopies) are pretty out-dated, since it was written in the 1970s.

The rest of the readings are from "Invitation to Computer Science". The following parts of Chapter 13 are on artificial intelligence, and are therefore abstractly related to Part II below, but are not needed before you start Part II below. Read Sections 13.1, 13.2, 13.3, 13.5 (only to the end of the paragraph at the top of page 614), 13.6. This will provide a conceptual framework of some artificial intelligence concepts, which includes the relative capabilities of humans versus machines, and how knowledge can be represented in a machine. This topic includes decision trees (a la "guess the animal" learned decision tree as shown in class) and search through possible moves of a game (a la Part II below).

The following parts of Chapter 10 are on theoretical models of computation -- they will prepare you for Part I below. Read sections 10.1, 10.2, 10.3.1, 10.3.2 (only skim this section to get a sense of Turing machines - you are not responsible for the details), 10.6, 10.7 (also just skim, since it is pretty technical -- we will go over the halting problem with less technical details in class), 10.8

Chapter 14 through Section 14.3. You will not be tested on these concepts, but now that you have learned so many fundamental concepts behind computer science, I highly recommend you read this introduction to the *responsibilities* that come with:

- The capabilities of algorithms.
- The increasing infrastructure (e.g., the internet) enabling massive digital storage and communication of information.

Part I: Theoretical Questions on Computational Models (4 points)

Do Part II first, since it will be covered in class first. Show your work or explain your answer to each of the following:

1. Chapter 10 of "Invitation", exercise 28 (but change "C++" to "Java" in the question).
2. Provide an argument that any and all "non-physical" (i.e., manipulating symbols or information) algorithms can be implemented using only the loops, conditions, variables and the input/output commands (e.g., println) of Java. Reference the details of the definition of algorithm, and provide a few intuitive sentences on why you think it is true. (Note that, since Turing Machines can do anything Java can do [this has been proven], by answering this question you are actually providing an argument for the Church-Turing Thesis.)

   Next, consider that these aspects of Java are indeed powerful enough to implement any Turing machine (this has been proven already -- you don't have to). Now, citing the Church-Turing thesis, provide a couple sentences giving another argument that any non-physical algorithm can be implemented with these aspects of Java. Note that you don't need to know any details of Turing machines to create this argument.

3. The "10-second halting problem" is to decide, given a computer program and its input, whether that program will halt within 10 seconds. Explain why this problem actually is computable (in contrast to the "full-fledged" halting problem). You can prove it is computable by providing an algorithm. Your algorithm does not have to be as detailed as pseudo-code.
4. You just realized you could make a lot of money if you created a compiler to detect infinite loops during compile time. This way, a programmer would find out that her/his program would go into an infinite loop during compilation, instead of finding out the "hard way" by running the program and dying before it ever stopped. Could you, or anyone else for that matter, create such a compiler? Why or why not? Remember that a compiler is just another computer program, and that each computer program implements some algorithm.

Part II: Othello (7 points)

For this assignment you will use Java to program the intelligent "brain" part of an on-line, web-based Othello player. In a word, this brain is a "move-chooser" to select what move the computer makes next. You or your friends can go to your web page and play against the Othello nemesis you create.

Next, you will slightly alter your strategy to make it play differently. And then, we will tell you a Unix command that will automatically run a tournament between these two Othello brains that you have created to see which one is better (that is, which wins more frequently).

Finally, you will pit your creation against the ultimate Othello-playing arch-nemesis, Edgar. Edgar is bad, badder than Darth Vader; meaner than a junkyard dog. This will be another simple Unix command. Edgar will beat your Othello brain, I am afraid, since the Othello brain you create will have some intrinsic limitations. But, how badly will your brain be slaughtered? Stayed tuned for more!

Othello is played on an 8-by-8 board (i.e., a chess board). The pieces are discs that are black on one side and white on the other. When you play, the goal is to dominate the board with your color. Each move, you put one new piece on the board. If the new piece you just added "surrounds" a row, column or diagonal sequence of your opponent's pieces, you get to flip those pieces to your color.

The rules become clear when you actually play the game. Try playing a game of Othello now, at http://www.cs.cmu.edu/~astro/ (follow the links to "Exegesus" and then "Beat Edgar"). At this web page, you are actually playing against Edgar, so plan to be creamed.

If you are interested, you can learn about how Edgar was programmed at www.cs.columbia.edu/~evs/ml/hw4EDGAR.html. Edgar, an intelligent computer program, is one of two leading fictional characters (the other is a human) in the novel, "Exegesis", by Astro Teller. Find out more about this novel from Astro's home page. Edgar is also a non-fictional, intelligent Othello player developed by the same Astro Teller. Edgar's programming used advanced techniques in artificial intelligence which are beyond the scope of this course. Edgar sits patiently as part of an advertisement for "Exegesis" at www.randomhouse.com/vintage/exegesis.

Instructions to Make Your Othello Brain

These instructions will tell you how to set up Othello on the web to play human-against-computer, and also how to pit tournaments that play the computer against itself.

1. Create a directory inside your public_html directory called othello by typing the following two commands at the dollar-sign prompt:

   ```
   cd public_html
   mkdir othello
   ```

2. Go into that directory

   ```
   cd othello
   ```

   and copy all the Java source code and a couple other files from ~es66/public_html/othello/ by typing (actually, you can just cut and paste the following into the window with the $ prompt):

   ```
   cp -r ~es66/public_html/othello/audio .
   cp -r ~es66/public_html/othello/images .
   cp -r ~es66/public_html/othello/*.java .
   cp -r ~es66/public_html/othello/othello.html .
   cp -r ~es66/public_html/othello/FinalWeights .
   cp -r ~es66/public_html/othello/traindata .
   ```

   (the -r makes it copy all subdirectories, "recursively".) Don't forget those periods at the end, each after a space.
3. After you copy the contents you should have the following files. (You do not need to work with most of these files -- these instructions will tell you which to look at and change.)
   - MyPlayer.java - the class you will modify
   - MyPlayer2.java - the second class you will modify
   - othello.html - the html page to be displayed
   - OthelloBoard.java - board information
   - OthelloCache.java - allows console play against players
   - OthelloEdgar.java - Edgar
   - Move.java - a class that contains info about each move
   - OthelloDisplay.java - information to be displayed as an applet
   - OthelloPlayer.java - The player class
   - audio/ - a directory with all audio information
   - images/ - a directory with all visual images
   - FinalWeights - used by edgar
   - traindata - used by edgar
4. You will then compile the entire project. To do this, all you have to do is type "javac OthelloDisplay.java" and "javac OthelloCache.java". It will compile everything else for you and will create respective .class files for each .java file above.

It will give you one warning -- ignore it. Didya know that a compiler warning is not nearly as bad as a compiler error, but they should not be ignored unless the instructor or a law enforcement officer tell you to do so.

5. We now have to change the permissions on all the files so that the applet may be executable. The following commands will change permission on the othello directory, as well as all the contents within it including the contents of the audio and images directories (copy and paste them into the shell window for your convenience).

```
chmod a+r *
chmod a+r .
chmod a+x .
chmod a+x audio/ images/
chmod a+r audio/* images/*
```

6. You have not programmed the brain yet, but you can try it out anyway -- right now it will be playing randomly. Open up netscape and try playing Othello against the computer. The web page will be www.columbia.edu/~youremail/othello/othello.html. Be sure to have your sound on :) (Or, if you export your display as in the last homework, you can view the applet without a web browser by typing "appletviewer othello.html".) This will be the location of your "smart" Othello program, so you might want to put a link to it from your home page -- it's cool!
7. If there is a permissions problem, e.g., netscape gives a message saying it was not world readable and it has a protection 700 which prevents it from opening up the web page, try the following things: 1. Make sure you have copied the chmod commands very carefully from the homework assignment, including the periods ("."). The best way is to cut and paste into a Unix window from Netscape. 2. Make sure the .class files that result from compiling are readable. Either type "chmod a+r *" in the othello subdirectory, and/or type "umask 000" and compilations after that command will set the .class file permissions correctly.

   In general, with regard to this type of problem, it is always best to do the homework during a TA's office hours so you can ask when you encounter unanticipated technical "gotchas" and obstacles.

8. So as you may or may not notice, it is very easy to beat the computer. So the goal of this homework, is to create a "smart" Java program that will be tough to beat. There are many different strategies that can accomplish this, some of which are the subject of Ph.D. theses today. For example: You could create some kind of system where the computer "learns" what your strategies are in playing and tries to combat them affectively. Other strategies, like those probably used in the IBM Deep Blue Chess Playing Machine pick current moves based on what they predict the future to be. For example; they will choose one move, see what your possible move will be, then will see what its possible move will be, and so on till it reaches a particular level and can judge what current move will be the most beneficial in the long run.

   However, we can not use such complicated strategies this semester. Instead, the strategy to be used in your version of Othello is as follows. When it's the computers turn to move, it will generate a list of all its possible, legal moves. The rules of the game Othello stipulate that you can only go somewhere that will cause some of your opponent's pieces to be flipped (if there is no such option, your move is skipped and your opponent takes another move). Creating the list of legal moves has already been programmed for you. It will go through that list and give a rating to each possible move. This rating will be determined by some code THAT YOU WILL WRITE which will look at the result of the move and give a numerical rating. After all the boards are evaluated, the computer will pick the best board which has the highest rank and make the move that will produce that board. If there is a tie, it will pick one randomly.

   In a word, your code is a "move-chooser" to select what move the computer makes next.

   To do this, you only need to modify the file MyPlayer.java, in between the "**********"s -- take a look at that file now (with pico or emacs). You will see a portion of the code marked with *****. This is the only place you will have to modify anything. You will insert code that will go through a board, e.g., with nested loops, and come up with a score for that board.

The board is a two dimensional array called "board.board". (We thought just calling it "board" was too "boring" -- get it?) Basically, it is a grid. Usually, arrays start from 0, yet for this project, the gameBoard will be starting from 1. So the top left corner is board.board[1][1] and the bottom right is board.board[8][8]. Note that you put TWO SETS OF SQUARE BRACKETS when you give the coordinate of a location -- don't use a comma.

You really don't have to do very much to get this to happen. The board "situation" in the two-dimensional array is not the current state of the game -- it is a candidate ("hypothetical") situation corresponding to one of the moves the computer is considering. It the situation that would result if the computer chose that move. The computer has to decide if it wants to be in that situation. Your code will "rate" how good that situation is.

Again, for emphasis, the contents of the "board.board" 2-dimensional array represents a hypothetical board state that your code will give a rating to. It does not represent the current actual board state in the game being played. Of these possible next-move hypothetical board states, the one that YOUR CODE gives the highest rating to will be selected as the next move the computer makes.

The rating will go in the variable called "rank". Therefore, your goal is to scan the board's contents and rate how good a situation that would be in. The rating must end up as a value in variable "rank".

One simple strategy would be to count up the number of pieces of your color are on the board and assigning "rank" to that value. This makes sense since the goal is to have as many pieces of your color as possible.

Before you change MyPlayer.java at all, it doesn't look at the board at all, and the "rank" variable always ends up with the value zero (0). This means it is giving the result of each possible move the same value, so it has no basis for selecting. In this case, since it is a tie, it selects one of the moves randomly. This is why you were playing against a random player, above, when you tried out the web page.

COLOR: What color are you? Well the way Himani (Fall 1998 TA) set set up for you is that the computer is always white and the user is always black. But, when it plays computer-versus-computer, the same strategy will be used to play black. Therefore, you must program it to be able to play black or white.

This is easy to do because Himani set up a variable called "color" corresponding to the color the computer is controlling (think of the variable as "my color"). You use this variable to compare against individual board locations. So when you write your function, and you want to see if the piece located at (5,3) is your color you would do something like "if (board.board[5][3] == color)" (of course, that if-statement is missing the "then"-part). Also, you can test if a spot on the board is empty by comparing it to "OthelloBoard.EMPTY", as in, "if (board.board[5][3] == OthelloBoard.EMPTY)" (capitalization matters). You can also test if the piece on the board is the opponent's color, if you want. Yet I will leave you to figure that one out. (Hint: each position on the board has either your color, the opponent's color, or is empty.)

COMMENTS: Be sure to comment your code. Comments are done using //. This will help YOU debug your code and it will also help the TA in figuring out what your code is meant to do.

Play a few games against Edgar and against the random player to get a feel for the game and a sense of tactics and strategy. In addition to counting up the number of positions with your color, there are several other possible strategies that can be added on top of this. In fact, simply counting these pieces is a pretty "greedy" algorithm, and does not result with a very effective Othello player -- a random player will often beat that strategy!

Possible strategies include:

- If I am white and I have white pieces on the corners, than this board is worth more than a board that doesn't have the pieces on the corners. This is because it is impossible to surround corner pieces. Therefore, these positions should be worth more. How much more? That is up to you -- take an "educated guess" and try it out!
- Likewise, edge positions are also worth more, since they are difficult to surround -- but not as much as corners.
- Having a position right next to a corner is bad since it might make it possible for the opponent to get the corner, so these positions could be worth negative points.
- Make your strategy symmetric, e.g., treat all corners and edges the same.

Please feel free to add in any bells or whistles you choose to make your player "smarter". Use your imagination -- but recognize that some of the strategies you can think of might be too difficult to program right now. For example, in principle you could program it to look ahead a few moves, but that would be really rough! If you have a brilliant idea that you'd like to implement but don't know where to start, please feel free to contact a TA.

Your minimal requirements are to include a nested loop to count the number of pieces of your color on the board, and the addition of another strategy on top of that such as giving an extra bonus for having a corner location.

9. **Compiling and debugging.** So after you're done writing your this evaluating function, you will certainly want to test it. To do this, you will simply go to the command prompt ($) and will type "javac MyPlayer.java" to compile it. Once you compile with no compiler errors, you must restart your web browser to activate the change (Or, hold down the shift key while you click the reload button, but this only works on newer versions of Netscape -- the ones in MUDD don't support that unless you run netscape from a xterm.) **You will have to do this each time you make a change and recompile!**

   You will probably have lots of errors your first couple of time around. Don't get discouraged! And I would suggest spending some time to figure out what's wrong before asking for help (but also recognize when you are "stuck").

   Use compiler errors as guidelines. Each error will have a line number. Go to the first line number that you have an error on and see whats wrong. Maybe you forgot to close a parenthesis or forgot a semicolon or attempting to use a variable that is not defined. Also remember, that fixing up one error may clear up a lot of other errors. Also be sure you to use the features of emacs or pico. After you write each line, press the TAB key. If the line doesn't align up the way you think it should, something is probably wrong in your parenthesis or bracket count. See also the list of debugging hints in the previous homework assignment.

   NOTE: The TA's will often ask you if you tried to solve the problem before coming to them first and they will ask you to try explain what you think the problem is.

   JAVA CONSOLE: While the game is playing, you may not be fast enough to catch whats going on. Thus, there is a feature in every browser called the console. You should be able to bring it up using the Netscape (or Internet Explorer) options. As you play the game, the moves and a grid of the gameboard will be displayed to help you figure out whats going on -- it will show you the rating given to each move it is considering, and which move it actually chose. Also, this is where you will see output if you put in System.out.println()'s to debug your code.

10. **Tournament comparisons**. Now that you have programmed a good brain, there is one more step. You need to compare its performance against another brain. To do this, edit MyPlayer2.java to have the same strategy as MyPlayer.java, except slightly different. That is, you can copy your code from MyPlayer.java into MyPlayer2.java using cut and paste or just retyping it. Then, in MyPlayer2.java make a small change that you think could make a difference. For example, you could make the corners worth even more, or even less. This could make it play much worse, or maybe you think it would make it play much better. Compile MyPlayer2.java.

Now, you can try a head-to-head match against MyPlayer.java and MyPlayer2.java by typing

```
java OthelloCache MyPlayer
```

(capitalization matters here at the dollar-sign prompt) which will play one game of MyPlayer (black) against MyPlayer2 (white). This will show you what happens during one game between the two -- take a look at the output to make sure it is working.

However, one game is not enough for a fair comparison, since there is a random element. Therefore, play 50 games of MyPlayer (black) against MyPlayer2 (white) by typing:

```
java OthelloCache scoreonly MyPlayer
```

You will notice that it takes a while to play all 50 of these games. Take this time to think about why this takes so long: for each game there are about 64 moves; for each move there are between 1 and 10 possible moves; for each possible move the computer must run your function to evaluate the move; this function may involve nested loops.

Also you can play one game of MyPlayer (black) against Edgar (white) -- this will play one game but will list out all the steps and a lot of other junk edgar uses.

```
java OthelloCache
```

(Note that Edgar was trained on white so he can only be white.) And you can play 50 games of MyPlayer (black) against Edgar (white) -- This will play 50 games and show only the scores of each game, and then the average performance of the players over those 50 games.

```
java OthelloCache scoreonly
```

If one player doesn't appear to be absolutely slaughtering the other consistently, you may need even more than 50 games to get an accurate sense of their relative performance. Therefore, you can do a few tournaments to add up to 100 or 200 games.

11. **Submission.** You are required to submit three things and only three things for this assignment: MyPlayer.java, MyPlayer2.java and a README file. This README file must describe the strategies in MyPlayer.java and MyPlayer2.java. It must also describe which you expected to be better, and then it must give the numerical results of the head-to-head tournaments (given on the last couple lines when you run a tournament). Write a few sentences drawing conclusions about these results: what do they mean to you about Othello-playing strategies? Keep the whole README down to a couple paragraphs. You will get points taken off if there is no README file.

As always, if you have any questions or concerns, please contact your TA's or instructor.

This will be a simple yet challenging means to apply much of the theory you have learned this semester. And if you're possibly interested in pursuing a major in computer science, it will help you with that decision. If you enjoyed coming up with the algorithm and writing the code, then you may just be suited for the long and prosperous :) road ahead.

Part III: (1 point)

Do nothing. On free point!

Acknowledgements for Part II: Thanks to Himani Naresh for helping write the first draft of this homework assignment, and to Himani Naresh again, as well as Eleazar Eskin and Astro Teller for creating the Java code to do Othello strategies on the web. Thanks to Astro Teller for use of the Edgar code.

email: *evs at cs dot columbia dot edu*