CSC442: Intro to AI
Project 1: Game Playing

September 3, 2018

This project is about designing, implementing, and evaluating an AI program that plays a game against human or computer opponents. You should be able to build a program that beats you, which is an interesting experience. The project has three components:

1. First, you need to develop a basic program that plays tic-tac-toe.
2. Second, you need to extend this program to play a more interesting (i.e., harder) variant of tic-tac-toe.
3. Third, you will further extend your program to support an alternate, more difficult win condition.
4. You must also write up your work, describing your approach and evaluating your results.

Upload a ZIP archive with everything to BlackBoard before the deadline. Include a README file explaining how to build and run your project. Full details are given below.

FYI: If you'd like a nice firsthand (and even somewhat technical) account of building a world-class game playing program, check out *One Jump Ahead: Computer Perfection at Checkers*, by Jonathan Schaffer (ISBN-13: 978-0387765754).

Grading

Graduate Students	
Basic TTT program	25%
Advanced TTT program	25%
Second advanced TTT program	25%
Writeup	25%

Part I: Basic Tic-Tac-Toe

I don't think I need to cover the basic rules of Tic-Tac-Toe (TTT). If you don't know them, check Wikipedia.

You *must* use the state-space search paradigm that we have been exploring in class. Think about what this means for Tic-Tac-Toe... What do you need to do to represent the problem? Think about it... Then read on for our thoughts.

You did think about and didn't just read on, right? Ok. For state-space search, we must represent the following elements:

- States: Obviously, the state of the TTT board is the arrangement of X's and O's (and blanks) on the 3x3 board. Don't forget that the state also includes whose turn it is to move.
- Actions: The applicable actions in a state are those in which the player whose turn it is to move puts their mark into an empty cell on the board.
- Transition model: Takes a state, updates the board with the appropriate mark at the appropriate position, and toggles which player is next to play.
- Initial state: Empty board, and we will assume X plays first.
- Terminal states: A board with three co-linear cells (horizontally, vertically, or diagonally) containing the same (non-empty) mark is a win for that player. A board with no empty spaces and no winner is a draw.

The classes (data structures) for these things should suggest themselves. Go write them now.

Once you have the representation, you *must* use the state-space paradigm to select moves for the computer player. I realize that for basic Tic-Tac-Toe, you could probably write a big `if` statement to decide what move to make in a given state. But what would you learn from doing that? And anyway, it won't work for a more complicated game (see Part II of the project).

Do something simple first. For example, even selecting randomly from among the applicable (possible) actions will allow your program to play a legal game. Asking the user for a move is another useful strategy for testing and for playing against your program. With the right design, the search strategy is "pluggable".

Regardless, for Part 1 you *must* implement the MINIMAX algorithm (AIMA Fig. 5.3) even though it is overkill for basic TTT. You can debug it in the simple game, and then you'll be able to use it harder games.

For basic TTT, a move is simply a number from 1-9 indicating which position the player is marking, according to the following chart:

1	2	3
4	5	6
7	8	9

Note that this encoding starts at 1, not 0.

With this in mind, design your program to operate as follows:

- Communicate via the terminal (a.k.a. standard input, output, and error, a.k.a., stdin, stdout, and stderr, a.k.a `System.in`, `System.out`, and `System.err` in Java).
- Your program must read opponent's moves from standard input (`System.in` in Java) and print its own moves to standard output (`System.out`).
- Do not print anything else to standard output. This will allow us to connect two programs and play them against each other.
- Any other messages must be printed to standard error (`System.err`). In particular, you should use this stream when you prompt the user for input.

- Start by asking the user if they want to play X or O. Your program should read a line that will be either "x" or "o" in response (you should handle upper and lower case). As stated above, you may assume that X moves first.

- For the opponent's moves, you should print the board to the standard error and prompt for a move. Read the move (a line with one number for Basic TTT; two numbers for 9-Board as described below). Check that the move is legal. Then apply it, and maybe print the board again.

- For the computer's moves, call your state-space search method to compute the move, print it to standard output, apply it, and print the board (to stderr).

- Print messages to stderr during the search or summary statistics after a search that illustrate the operation of your program.

- When the game is over, print out who won and start a new game (this will allow us to play programs among many teams automatically).

You are of course welcome to develop fancier interfaces, including graphical ones. It's not really the point of the project however, so don't expect extra credit for it.

In your writeup, document your efforts by explaining your design. Refer to concepts and algorithms discussed in class or in the textbook. Provide traces or screenshots to illustrate the operation of the program. Analyze your program's performance. Did it work? Did it play well? How long did it take to play? And so on.

Part II: Advanced Tic-Tac-Toe

Once you have the basic game working, you need to extend your program to play so-called "9-Board Tic-Tac-Toe," which works as follows:

- You have nine 3x3 TTT boards arranged in a 3x3 grid.

- The goal of the game is to win a single one of the boards like in regular TTT.

- However there is one crucial constraint: If a player has just played at some position on some board, then the next player must play on the board in *the corresponding position* in the grid. For example, if a player marks the bottom right *position*, position 9, of any board, then the next player must mark any open space on the bottom right *board* (the board at position 9). The first player to play can play anywhere. If the board required by the preceding rule is full, the player can play on any board.

This is a neat variant on TTT. You should probably spend some time in your writeup explaining why (or, perhaps, why not).

A move in 9-Board is two numbers from 1-9. The first number is the position of the board (using the same encoding as for single boards in TTT). The second number is the position on that board. Your program for 9-Board must read and write these moves from standard in and out. It should check that the moves it read are legal before applying them.

If you did the first part of the assignment properly, you should be able to reuse much of the code for this part. However you will find that pure MINIMAX is infeasible for more complex search spaces. So you will need to implement some of the advanced techniques for adversarial (game tree) search. For full points, you *must* implement depth-limited search (H-MINIMAX with a heuristic function) and alpha-beta pruning. Both of these are standard in real-world game-playing programs.

Again, document your design, with reference to the Part I program. In your writeup, describe your heuristic function (or functions if you try different things). Illustrate the program in action. Evaluate its performance, including showing the effects of alpha-beta pruning.

Part III: More Advanced TTT

You must additionally implement one of the following variants of tic-tac-toe:

- Ultimate or Super Tic-Tac-Toe: Like 9-board, but instead of winning a single child board, you try to win 3 child boards in row horizontally, vertically, or diagonally (like TTT).

- 3D TTT, also known as Qubic: This is played on a 4x4x4 arrangement of boards (3x3x3 is provably easy).

- 4D TTT: This is played on a 3x3x3x3 arrangement of boards, where the goal is to maximize the number of rows, columns, and diagonals of 3 with your mark.
- 10x10 TTT (or any $n \times n$ for $n > 3$) is unfortunately not very interesting, so this is not an option.

You can find more information about these online, but **do not download code from the Internet.** Your program may not be able to play these games very well, so be sure that your writeup makes clear what you are doing to try to have it play well.

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) Any collaborators (see below)
   (b) How to build your project
   (c) How to run your project's program(s) to demonstrate that it/they meet the requirements
2. All source code and build files for your project (see below)
3. A writeup describing your work in PDF format (see below)

We must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better grade your will be.** It is your job to make both the building and the running of programs easy and informative for your users.

Programming Practice

Use good object-oriented design. No giant `main` methods or other unstructured chunks of code. Comment your code liberally and clearly.

You may use Java, Python, or C/C++ for this project. Other languages (Haskell, Clojure, Lisp, etc. by arrangement with the TAs only.

You may **not** use any non-standard libraries. Python users: that includes things like NumPy. Write your own code—you'll learn more that way. Note: if you can provide a compelling argument that **b**asic NumPy functions (e.g., array primitives) will make your program more efficient, you may use them, but **a**bsolutely no high level algorithms may be used from external libraries.

If you use Eclipse, you *must* make it so that we can build and run your project without Eclipse. Document exactly what needs to be done in your README (Makefile, `javac` or `gcc` incantations, whatever). Eclipse projects with no build instructions will receive 0.

Your code must build on Fedora Linux using recent versions of Java, Python, or `gcc`.

- This is not generally a problem for Java projects, but don't just submit an Eclipse project without the build and run instructions detailed above.

- Python projects must use Python 3 (recent version, like 3.6.x). Mac users should note that Apple ships version 2.7 with their machines so you will need to do something different.
- If you are using C or C++, you must use "`-std=c99 -Wall -Werror`" and have a clean report from `valgrind`. And you'd better test on Fedora Linux or expect problems.

Writing Up Your Work

As noted above, it is crucial that you present your work clearly, honestly, and in its best light. We will give lots of credit for good writeups. We will not like submissions with sloppy, unstructured, hard to read writeups that were clearly written at the last minute.

Your goal is to produce a technical report of your efforts for an expert reader. You need to convince the reader that you knew what you were doing and that you did it as best you could in the (entire) time available.

Write up what you did (and why). If you didn't get to something in your code, write about what you might have done. (There's always *something* you might have done.) If something didn't work, write about why and what you would do differently. But don't write "I would have started sooner." Your readers already know that.

Your report *must be your own words*. Material taken from other sources must be properly attributed if it is not digested and reformulated in your own words. **Plagiarism is cheating.**

Start your writeup early, at least in outline form. Document your design decisions as you make them. That way half the write-up is done by the time you're running the program. Don't forget to include illustrations of the program in action (traces, screenshots) and some kind of evaluation.

Your report must be typeset (not handwritten). Using LaTeX and BibTeX makes things look nice and is worth learning anyway if you don't know it, but it's not a requirement. Your report must be submitted as a PDF file. If you want to use a word processor, be sure to generate and submit a PDF from it, not the document file.

The length of the report is up to you. For a CSC442 assignment, 3–5 pages is the *minimum* that would allow you to convince the reader. Be sure to include screenshots and/or traces (which wouldn't count towards the 3-5

pages minimum of actual text).

Late Policy

Don't be late. But if you are: 5% penalty for the first hour or part thereof, 10% penalty per hour or part thereof after the first.

Collaboration Policy

You will get the most out of this project if you write the code yourself.

That said, collaboration on the coding portion of projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC442.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the group should submit code on the group's behalf in addition to their writeup. Other group members should submit only their writeup.
- All members of a collaborative group will get the same grade on the coding component of the project.

You may NOT collaborate on your writeup. Your writeup must be your own work. Attribute any material that is not yours. Cite any references used in your writeup. Plagiarism is cheating.