This project is about designing, implementing, and evaluating an AI program that plays games. You should be able to build a program that beats you, which is an interesting experience. The project has two parts:

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.

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 [1].

You must turn in the following materials by the deadline:

1. The source code of your project

2. A write-up describing your efforts and results

Submissions will be through Blackboard. Upload a ZIP archive or something similar with everything in it. Do this before BB marks your submission as late.

Your grade for the project will depend on both aspects of your submission. The best code with a poor writeup will not get a great grade. Conversely, if your program doesn’t work very well, take the time to understand why and discuss it in your writeup. More info about writeups is given below.
Part I: Basic Tic-Tac-Toe

I don’t think I need to cover the basic rules of Tic-Tac-Toe (TTT). Some of you may even have already seen it in another class. If that’s you, then get this part over with and move on.

I recommend that you adopt the state-space search paradigm that we have been using in class. This means designing and implementing the following things:

- **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:** Obviously, update the board with the mark.

- **Initial state:** Empty board, either X’s or O’s turn to play.

- **Goal states:** Any board with three cells co-linear cells (horizontally, vertically, or diagonally) containing the same (non-empty) mark.

The classes (data structures) for these things should suggest themselves.

Then I would recommend starting by using the core state-space search algorithm (“Coolest Program Ever”). You will have to choose and implement a search strategy. Do something simple first. For example, even selecting randomly from among the applicable (possible) actions will allow your program to play a legal game. With the right design, the search strategy is “pluggable,” just like we saw in class. So you can improve it later. And you should be able to do better than random, since TTT is just not that hard.

You interface can be very simple. Here are some suggestions:

- **Communicate via the terminal (a.k.a. standard input and output).**

- **Start by asking the user if they want to play X or O (and which goes first).**

- **For their moves, maybe just print the board to the standard output and prompt for a move. Have a way for the user to specify which cell they want to mark. Check that their 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 out, apply it, and print the board.**

- **Print messages 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 offer to play again or quit.**
You are of course welcome to develop fancier interfaces, including graphical (point-and-click) ones. This is good for extra credit if the core functionality works.

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 should try extending your program to a more interesting game. I recommend that you try one of the following variants of Tic-Tac-Toe:

- 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.
- 9-board TTT: This uses a 3x3 grid of 3x3 boards, with some rules about where moves can be made.
- 10x10 TTT: The goal is to get 5 in a row, column, or diagonal.

You can find more information about these online. You are also welcome to try a harder game, like checkers or chess. Whatever you choose, describe it in your write-up (with references if needed).

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 may find that the simpler search strategies are inadequate for more complex search spaces. So you will probably need to implement some of the techniques for adversarial (game tree) search. For example, depth-limited search becomes pretty important, and of course alpha-beta pruning is the key to real game-playing programs.

Again, document your design, with reference to the Part I program. Illustrate the program in action. Analyze its performance.

**Programming Practice**

You can use any programming language you like. If you don’t have a preference, I recommend Java since the compiler will help you avoid many simple mistakes. Python is a reasonable op-
Lisp (or perhaps Clojure) is a powerful language for AI problems, although probably overkill for this project.

Include a README.txt describing how to build your program. It would probably be a good idea to learn how to use make if you don’t already, but various IDEs like Eclipse are possible also. The easier you make it for us to build and run your program, the better it is for everybody. If you aren’t familiar with these concepts, seek help early.

You should probably not need any non-standard libraries for this project. If you do, be very clear about the requirements in your writeup and in the README.txt. When in doubt, check with the TAs early.

**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 are not idiots.

Provide references to relevant material that you used in your project. Of course you can’t use other people’s code, but if you find good ideas somewhere, you can code them yourself and then document the usage in your writeup.

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.
The length of the report is up to you. For a CSC242 assignment, I would say that 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).

References