# CSC242 Intro to AI
# Project 1: Game Playing

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. 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.

We are hoping to be able to play a tournament between student programs. Please watch BlackBoard for additional announcements regarding this.

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

| | |
|---|---|
| Basic TTT program | 25% |
| Advanced TTT program | 50% |
| Writeup | 25% |

## Collaboration Policy

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

You may discuss the project and program design with others, but you should try to write the code yourself.

If you do your own work, you *must* include the following statement near the start of your README file: “I did not collaborate on this project and all work is my own.”

If you feel it would be helpful, you may collaborate on the code with other students in groups of no more than three (3). You *must* report the names of your collaborators near the start of your README file.

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.

## 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. Some of you may have already seen TTT 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, 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.

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.

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, I would print the board to the standard error and prompt for a move. Read the move (a line with one number for Basic TTT and two numbers for 9-Board). 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 many teams automatically).

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 also works.

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 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 one of the boards like in regular TTT.
- However there is one other *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 player one 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 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.

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 the simpler search strategies are inadequate for more complex search spaces. So you will need to implement some of the techniques for adversarial (game tree) search. For example, depth-limited search becomes necessary, and 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. Evaluate its performance.

If you get 9-Board working, you might also be interested in these other variants of TTT:

- So-called 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). An interesting extension if you've got 9-board working.
- 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.

You can find more information about these online.

## Programming Practice

You may use Java, Python, or C/C++. Other languages (Haskell, Clojure, Lisp, ...) by arrangement with the TAs only. Do not use any non-standard libraries without consulting the TAs.

Use good object-oriented design. Comment your code liberally.

Include a `README.txt` describing how to build and run your program. It would probably be a good idea to learn how to use `ant` or `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.

## 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 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).