Detailed Making Process

1. In 11th grade, I made a project for CS: a program to play a perfect game of tic-tac-toe (although a project at this level was *not* required; the requirements were relatively basic).
2. Towards the beginning of 12th grade, I started working on an AI program to *learn* to play tic-tac-toe. I used the 11th-grade project as the basic framework for the game itself. My objective was to have it never lose (since that's the best you can expect consistently in tic-tac-toe) while playing as O (since it's easier to win as X).
3. I started by writing several functions which would help the program make a decision when it was its turn in a game. The current ones are next_win(), next_lose(), possible_win(), possible_lose(), centre(), flank(), anti_flank() and just_opposite().
4. Then, I created a procedure to record the details of each game in a text file, and to process those details in the next game. I also added a simple encryption to the file (mostly just for fun).
5. I then wrote two functions, analyze() and final_choice(), to go through the stored data, assign scores to them, and rank them based on those scores. The ranked functions would then be used as the decision-making process for a single game.
    1. It took a lot of trial-and-error to find a scoring scheme that worked; I tried the natural numbers, the Fibonacci series, and some arbitrary sequences, until I instinctively realized that a binary sequence (the powers of two) might do the job; the top function would always receive a higher score than all the others put together, so the binary sequence is less sensitive to minor variations. The score gained by the top function in one game could never be neutralized by just one more game; it would require more repetitions for that to happen. I tried the binary sequence—and it worked!
    2. I ran into a problem in creating a unique function sequence in final_choice(). In the case where two functions had equal scores, I wanted a fair random choice. But I also wanted that previously used decision-making processes that had lost a game would not be reused. For this, I needed the program to create all satisfactory permutations, eliminate the previously used ones, and choose among the remaining ones. But creating all the permutations was taking a *lot* of time, essentially freezing the program. To get around this, I made the program create 11 random satisfactory permutations one by one until an unused one appeared. If none appeared, I let it just give up and choose a random decision-making process. So by sacrificing a tiny bit of accuracy, I saved heaps of time and energy.
6. I started training the AI program by manually playing each game with it. Thousands of times. To help with this, I added a "dev tools" feature that allowed me a backdoor into

the program: by putting in the password at the very beginning, I could gain extra insight into the progress of the training.

1. “function scores” made it display the current scores for each function, whether or not they would be used.
2. “refresh” made it wipe the file it was using to store data.
3. “try this” let me verify that there was at least one perfect decision-making process by feeding it in and playing one game with it.
4. “show thinking” made it display the process it had decided on.
5. “always continue” allowed me to skip the step where it asked whether I’d like to play another game or not.
6. “this letter” allowed me to skip the step of choosing X or O before every game.
7. “back” simply exited the dev tools menu and started the game.

7. Meanwhile, I shortened and refined the decision-making functions using other new functions (like filter(), which returns the overlap between available options and a function’s choices). I also made a lot of bug fixes with various aspects of the code.
8. I realized that there was an error in the anti_flank() function that rendered it useless. I corrected this error and also deleted a couple of functions which were incorrect.
9. Finally, the program worked perfectly, repeatedly gaining enough data in fewer games (27, 143 and 52 each) to be capable of never losing. I ran it thrice in a row to be sure that it wasn’t a fluke—and it wasn’t :)