**Group Members**

Rick Haffey (login: *rihaffey*)

**Compiling the program**

* Navigate to the directory containing the C++ code files
* Execute the following from the command line:
  + **g++ \*.cpp –o ./output/program**

**Running the program**

* Copy the *program* executable file created above to */home/rihaffey*
* From the command line, excute the *myplayer.sh* script (located in the same directory)

**Static evaluation and heuristics**

* For each player, the code iterates through every cell of the game board, and for each cell
  + Examines every possible win it could be a part of:
    - Vertical
    - Horizontal
    - SE diagonal
    - SW diagonal
  + Heuristic: At each orientation, every cell could be involved in up to *n* (pieces-To-Win) possible wins (except where various positions are unusable due to cutoffs at or near board edges)
  + Heuristic: The value of every possible win = (2 ^ the number of owned pieces), so that the value increases exponentially as each possible win gets closer to an actual win
  + These values are summed to produce the cell's value
  + Static eval: If a state represents an *actual* win, the calculator returns a reference value (much higher than a normal calculated value) to represent this
  + Static eval: The process also takes into consideration blocks for a 'save', and returns a value that's less than a win, but much higher than most other calculated values (so as to *not* be chosen over a potential immediate win, but to be chosen in most other cases over just 'strategic' options.
  + Static eval: A draw evaluates to 0
* The cell values are summed to produce the overall game state value
* The minimax algorithm uses the *difference* between MAX and MIN’s state heuristic scores as the result of the UTILITY function.

**Minimax and Alpha-Beta pruning**

The logic used is based closely on the pseudocode for the algorithms described in AIMA. I’ve also included iterative deepening and timeout handling as follows:

The minimax algorithm is run so that the terminal test function exits if any of the following three criteria are met:

* The game is complete (WIN1, WIN2, or DRAW)
* The current search depth is greater than an iteratively increasing max\_depth value
* The current run-time is within a configured threshold of the overall timeout period

The logic runs through the process, storing the results of the previous iteration, increasing the depth, and trying again. If the process nears the timeout during an iteration, processing exits and the result from the previous iteration is used to determine the next move.

I’ve also added a ‘first-chance’ move evaluator that checks for obvious moves (immediate win, immediate block, etc.), and if those conditions are present, returns the ‘obvious’ move. This cuts down on the minimax processing effort and time in those cases, and also provides a failsafe in the event something in the minimax algorithm isn’t calculating correctly.

**Results**

I used various methods for testing the program.

* During early development, most of the tests were manual, mimicking the behavior of the referee from the command line.
* As the player logic became more ‘stable’, I implemented a simple referee and random player, to mimic locally the behavior of the competition
* I also ran tests running two of my players against each other

Although I have seen big improvements in the performance (time and success) with most of my coding iterations, I’m still seeing situations where my heuristic is leading to counter-intuitive moves. As an example, there are cases where the heuristic doesn’t lead to selection of a block move in cases where the opponent will win without the block. In order to ‘mitigate’ this, I’ve included the first-chance move handler that takes a win or block as the move choice (prior to starting the minimax search), if one of those moves is available.

**Fitness of evaluation and heuristic functions**

**Additional AI Component: Machine Learning**

**SourceCode**

I’ve been storing all my code in Github during the development process. In addition to the code submitted on D2L, my entire source code repository is available [here](https://github.com/rickhaffey/CSC480/tree/master/FinalProject) for reference. You’ll see that I’ve had multiple versions of the project (C#, C, C++). In the early stages, I used C# (the language I’m most comfortable with) to solidify my ideas, build an initial, base implementation, etc. I then converted over to C++ (with an aborted C conversion in between), in order to support compiling and building in a Linux environment. [I was working on the assumption that the Mono .NET framework wouldn’t necessarily be present on the competition server.]