**Due Monday February 14th, 1 a.m. sharp!**

**Teaching computers to play games**

**Introduction**

Humans and other animals engage in many activities that aren't directly essential to life (those would be work), but are pursued for the sheer joy that comes from facing, and possibly overcoming, a challenge: moving a ball across a field, moving your body and a skipping rope around each other, finding the word that fits in a crossword, or tweaking a Python program. Some of these recreational activities have enough structure to be identified as games. Computers can be programmed to play some games, with varying degrees of success. Some reasons for teaching computers to play games are:

* It's useful, for example, for [training](http://en.wikipedia.org/wiki/Computer_chess) human players
* Programming a computer to play helps us understand the game better.
* Programming a computer to play helps us to understand things, other than games, that computers can potentially do.
* Programming a computer to play hard games can be challenging, interesting, and fun for its own sake. Some game-playing strategies for computers are completely [solved](http://webdocs.cs.ualberta.ca/%7Echinook/), others aren't.

In this assignment you will be programming computers to play a restricted set of games: [two-player](http://en.wikipedia.org/wiki/Game_theory#One-player_and_many-player_games), [sequential move](http://en.wikipedia.org/wiki/Game_theory#Simultaneous_and_sequential), [zero-sum](http://en.wikipedia.org/wiki/Game_theory#Zero-sum_and_non-zero-sum), [perfect-information](http://en.wikipedia.org/wiki/Game_theory#Perfect_information_and_imperfect_information) games. Lots of games have these features: tic-tac-toe, chess, go, checkers, mancala, and [nim](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/a2.html), for example. Important characteristics are:

Two-player

There are two (usually distinct) players, we'll call them Player A and Player B.

Sequential-move

Players take turns making sequential moves (the individual players' moves are called [plies](http://en.wikipedia.org/wiki/Ply_%28game_theory%29)). Plies change the game's **layout**, for example, the configuration of Xs and Os together with the player about to play, in tic-tac-toe. The only possible outcomes of the game are a win for Player A (and a loss for Player B), a win for Player B (and a loss for Player A), or a tie. In some such games, no tie is possible.

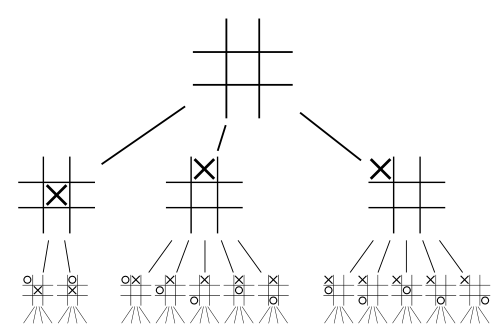
Zero-sum

The benefit of any ply for Player A is exactly inverse to that ply's benefit for Player B. For example, a ply that wins for Player A loses for Player B. A ply that takes Player A closer to a win takes Player B closer to a loss. If we could measure and add the benefit for Player A to the benefit for Player B of any ply, the result would be zero

Perfect information

Both players know all plies made by their opponent.

Position

The game layout, together with the sequence of plies that take you there. All the possible positions reachable from a given position form a [game tree](http://en.wikipedia.org/wiki/Game_tree). Notice that starting from an initial position, there is exactly one path (sequence of plies) for each reachable position (sequence of plies plus layout). An important subtle point: two different positions can have the same layout, for example you can reach the layout having XOX across the top row of a 3x3 tic-tac-toe in two different ways, starting from an empty game with X to play.

Your program will implement a [minimax](http://en.wikipedia.org/wiki/Minimax) strategy for choosing the next ply. The computer (using your program, of course) determines the best choice for, say, Player A's next ply by considering all legal next plies available and choosing one that takes Player A closest to winning. In zero-sum games, taking Player A closest to winning means taking Player B farthest from winning, so the computer chooses a next ply for Player A that forces the poorest choice of follow-up ply for Player B. Since we assume that Player B is also intelligent, Player A considers Player B's strongest (maximum) response to each of A's potential nex plies. From among B's strongest potential responses, A chooses the weakest (minimum). In other words, A chooses the next ply to minimize the maximum choice available to Player B, hence the name minimax.

The minimax algorithm is naturally recursive: the computer rates the possible next plies for Player A based on its rating of the best follow-up plies for Player B, and it uses the same algorithm — minimax — to rate the follow-up plies for Player B based on its rating of the possible follow-up plies for player A. And so on. The recursion terminates when the game is in a terminal state: win, loss, or tie --- at that point no follow-up plies are possible.

Minimax must also contend with limited computer resources. If in our example above Player A has only enough processing power to consider the list of available plies, then the algorithm must estimate A's chances at a depth of one ply, without considering B's responses. With more resources, A would be able to consider A's chances at a depth of 2 plies: both the list of plies available to A, and for each of them, the list of plies B could make in response (in order to consider B's strongest response). With more resources minimax can consider the A's chances at a depth of 3, 4, 5, or more plies beyond the current position. In many games there aren't resources to consider all possible sequences of plies leading from the current position to a terminated game, so the rating of the current best move is based on an estimate, often called a [heuristic](http://en.wikipedia.org/wiki/Minimax#Minimax_in_the_face_of_uncertainty), evaluated after plies have been considered to a depth that is computationally feasible.

As a side-effect of this assignment, you will learn lots about recursion, unordered tree structures and inheritance.

**Required work**

You'll need to download [minimax.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/minimax.py), [tic\_tac\_toe.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/tic_tac_toe.py), and [game\_tree.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/game_tree.py), where you'll find some of your work has already been completed by good-natured boffins. The remaining work is specified in [docstring](http://www.python.org/dev/peps/pep-0257/) comments, and you must write implementations that satisfy those specifications. We **strongly** recommend that your implementations use recursion, since the problems you are solving are recursive in nature. If you decide to avoid recursion, possible consequences include your implementation not working (and thus losing marks), or your implementation being long, complicated, and hard to understand (and thus losing marks). You'll notice that the recursive implementations completed for you tend to be brief and clear.

Since minimax is such a powerful general idea, it's natural to implement the algorithm in a superclass, and then customize it in a subclass for each particular game you want the computer to play. You will complete the implementation of [minimax.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/minimax.py) by implementing the body of method **minimax** to find the best ply (as well as a number rating just how good that ply is) for the current player. Numbers rating how good or bad plies can be fall between minimax.GameState.LOSE and minimax.GameState.WIN, with minimax.GameState.DRAW being of equal advantage to either player. Here's how minimax chooses the best play, and rates how good it is:

* Explore the list of plies available to, say, Player A to find one yielding the highest predicted score for the current player (equivalent to the lowest predicted score for Player B. The plies available fall into several categories:

**The ply ends the game:**

The predicted score is evaluated according to whether Player A wins, loses, or ties, using the terminal evaluation function. Return one of minimax.GameState.WIN, minimax.GameState.LOSE, or minimax.GameState.DRAW, as appropriate.

**The ply doesn't end the game, but you don't have resources to look more plies ahead:**

This corresponds to parameter **foresight** having a non-positive value after being reduced by 1 for this ply. The score is estimated using the heuristic evaluation, a game-specific technique you devise for estimating the score based on the current board layout only.

**The ply doesn't end the game, but you are allowed to look ahead further:**

This corresponds to parameter **foresight** having a positive value after being reduced by 1 for this ply. The predicted score for A is calculated by considering the best score possible available to Player B. This is the recursive part: you should evaluate the best score available to Player B using method **minimax** itself.

In order for the **minimax** algorithm to make exact predictions of scores, **foresight** must be at least as large as the maximum number of plies possible. But that opens up the problem of performance: it takes considerable time for a computer to look ahead even a few moves. Consider a tic-tac-toe game, starting from an empty board. Two intelligent players will produce a cat's game: a tie with every square filled. That means that **minimax** must look ahead 9 plies, considering 9 plies at the start, then (up to) 8 plies for each of them, then (up to) 7 plies, and so on. A coarse overestimate suggests 9! positions must be checked, although the true number is somewhat less since many illegal positions aren't explored: for example, no more Os are placed after a full row of Xs has been placed. Even so, this takes a few dozen seconds on modest hardware. To earn full marks on this assignment, you'll need to implement both of the enhancements described in the next section to improve performance.

Once you have a function that will come up with the best next ply, you'll want to see how well the computer uses it in an actual game. You'll find a complete implementation (those boffins again) of **GameState**'s subclass **NimState** in [nim.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/nim.py), used to determine the next ply in a game of [nim](http://en.wikipedia.org/wiki/Nim) (normal-play nim). You'll also find that [play\_nim.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/play_nim.py) allows you to play nim. Feel free to look over the boffins code, and play with it, to guide your own work.

Emulate the boffins' work on **NimState** by implementing a subclass of **GameState** called **TicTacToe**, in [tic\_tac\_toe.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/tic_tac_toe.py). You'll find docstrings, and you will write implementations that satisfy the docstrings' specifications. You'll need to devise a heuristic in order to make an educated guess of a tic-tac-toe game's outcome when there aren't the resources to examine each sequence of moves until the sequences reaches an end of the game. Although in 3x3 tic-tac-toe it may be feasible to compute all possible sequences of moves, this becomes impractical in a 4x4 or 5x5 game.

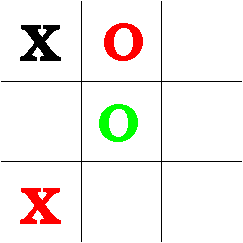
You will complete the implementation of [game\_tree.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/game_tree.py). The specification for some methods that the boffins omitted are in the docstring comments. The idea is to provide tools to explore the possible positions in a game tree.

**Enhancements**

If you complete the previous work in this assignment perfectly, you will earn 90%. To qualify for the remaining 10%, you must improve minimax using either memoization or alpha-beta pruning (or both). We will be able to measure whether you have successfully improved performance.

Once you complete, test, and play with [game\_tree.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/game_tree.py) a little, you will verify the observation that in many game trees there is huge redundancy in layouts. This means that **minimax** could be optimized by abandoning searches when you reach a layout that has been reached previously, since the game layout is what matters for predicting the outcome. You could keep track of layouts that have already been reached with a dictionary, using the layout's string representation as the key. The technique of improving performance by recording already-visited states is called [memoization](http://en.wikipedia.org/wiki/Memoization). We have provided minimax with parameter **layout** with default value **None** in case you'd like to implement this. You may use **layout** to store a reference to a dictionary of layouts that is consulted and updated during the execution of minimax. You are also free to ignore **layout**.

Another way to improve the performance of **minimax** is using [alpha-beta pruning](http://en.wikipedia.org/wiki/Alpha-beta_pruning). This approach allows you to avoid portions of the search space that won't change the outcome. Here's the idea.

In the tic-tac-toe game to the right, X began the game in the upper-left corner, and minimax is trying to find the strongest reply by O. Suppose that minimax determines that the strongest follow-up that X can offer to the green ply will force a draw, if played intelligently by both players. Then minimax determines that there is at least one follow-up to O's red ply that will guarantee a loss for O (the red X in the bottom left). At this point there is no need to consider any more follow-ups to O's red ply, since there is enough information to conclude that minimax won't choose red over green. We prune the search by ignoring all other follow-ups to O's red ply, and continue to consider other possible plies for O. This pruning saves a great deal of computation about a branch that wouldn't change the outcome.

We have included a parameter called **pred\_max** for minimax, which you can use to keep track of the maximum predicted score achieved by the predecessor of the current position in minimax's search of a game tree. **pred\_max** is meant to be useful in helping you prune searches analogous to those in the previous paragraph. However, you are also welcome to ignore the parameter.

**Submitting your work**

Submit the following files:

* Completed **minimax.py**
* Completed **tic\_tac\_toe.py**
* Completed **game\_tree.py**

You earn 30% if your code passes the following unit tests:

* [test\_minimax.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/test_minimax.py)
* [test\_tic\_tac\_toe.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/test_tic_tac_toe.py)
* [test\_game\_tree.py](http://www.cdf.utoronto.ca/%7Ecsc148h/winter/assignments/a2/test_game_tree.py)

If, and only if, you pass the unit tests above, we will consider your work for a grade greater than 30% based on:

* Another battery of unit tests (unpublished) to increase our confidence that your implementations satisfy the specifications in the docstring comments, worth 30%
* Visual inspection of your code to determine whether your solutions conform to sound programming practice and style, including modularity, good choice of variable names, clarity in code and documentation, worth 30%
* Up to 10% for implementing improved versions of **minimax**, as discussed above