Lab 3

To work on this problem set, you will need to get the code, much like you did for earlier problem sets.

Your answers for the problem set belong in the main file `lab3.py`.

Game search

This problem set is about game search, and it will focus on the game "Connect Four". This game has been around for a very long time, though it has been known by different names; it was most recently released commercially by Milton Bradley.

A board is a 7x6 grid of possible positions. The board is arranged vertically: 7 columns, 6 cells per column, as follows:

```
  0 1 2 3 4 5 6
0 * * * * * * *
1 * * * * * * *
2 * * * * * * *
3 * * * * * * *
4 * * * * * * *
5 * * * * * * *
```

Two players take turns alternately adding tokens to the board. Tokens can be added to any column that is not full (ie., does not already contain 6 tokens). When a token is added, it immediately falls to the lowest unoccupied cell in the column.

The game is won by the first player to have four tokens lined up in a row, either vertically:

```
  0 1 2 3 4 5 6
0
1
2       O
3       O     X
4       O     X
5       O     X
```

horizontally:

```
  0 1 2 3 4 5 6
0
1
2
3
4
5 X X X O O O O
```

or along a diagonal:

```
  0 1 2 3 4 5 6
0
1
2             O
3 O         O X
4 O       O X X
5 O     O X X X

  0 1 2 3 4 5 6
0
1
2       X
3       O X O O
4       O O X X
5 X     O O X X
```

Playing the game

You can get a feel for how the game works by playing it against the computer. For example, by uncommenting this line in lab3.py, you can play white, while a computer player that does minimax search to depth 4 plays black.

```
run_game(basic_player, human_player)
```

For each move, the program will prompt you to make a choice, by choosing what column to add a token to.

The prompt may look like this:

```
Player 1 (☺) puts a token in column 0

  0 1 2 3 4 5 6
0
1
2
3
4
5 ☺

Pick a column #: -->
```

In this game, Player 1 just added a token to Column 0. The game is prompting you, as Player 2, for the number of the column that you want to add a token to. Say that you wanted to add a token to Column 1. You would then type '1' and press Enter.

The computer, meanwhile, is making the best move it can while looking ahead to depth 4 (two moves for itself and two for you). If you read down a bit farther in the lab3.py file (or farther into this lab writeup), we'll explain how to create new players that search to arbitrary depths.

The code

Here's an overview of the code in the various lab files that you may need to work with. The code contains inline documentation as well; feel free to read it.

ConnectFourBoard

`connectfour.py` contains a class entitled `ConnectFourBoard`. As you might imagine, the class encapsulates the notion of a Connect Four board.

`ConnectFourBoard` objects are *immutable*. If you haven't studied mutability, don't worry: This just means that any given ConnectFourBoard instance, including the locations of the pieces on it, will never change after it's created. To make a move on a board, you (or, rather, the support code that we provide for you) create a new `ConnectFourBoard` object with your new token in its correct position. This makes it much easier to make a search tree of boards: You can take your initial board and try several different moves from it without modifying it, before deciding what you actually want to do. The provided `minimax` search takes advantage of this; see the `get_all_next_moves`, `minimax`, and `minimax_find_board_value` functions in `basicplayer.py`.

So, to make a move on a board, you could do the following:

```
>>> myBoard = ConnectFourBoard()
>>> myBoard

  0 1 2 3 4 5 6
0
1
2
3
4
5

>>> myNextBoard = myBoard.do_move(1)
>>> myNextBoard

  0 1 2 3 4 5 6
0
1
2
3
4
5   X

>>> myBoard    # Just to show that the original board hasn't changed
```

```
  0 1 2 3 4 5 6
0
1
2
3
4
5

>>>
```

There are quite a few methods on the ConnectFourBoard object. You are welcome to call any of them. However, many of them are helpers or are used by our tester or support code; we only expect you to need some of the following methods:

- `ConnectFourBoard()` (the constructor) -- Creates a new ConnectFourBoard instance. You can call it without any arguments, and it will create a new blank board for you.
- `get_current_player_id()` -- Returns the player ID number of the player whose turn it currently is.
- `get_other_player_id()` -- Returns the player ID number of the player whose turn it currently isn't.
- `get_cell(row, col)` -- Returns the player ID number of the player who has a token in the specified cell, or `0` if the cell is currently empty.
- `get_top_elt_in_column(column)` -- Gets the player ID of the player whose token is the topmost token in the specified column. Returns `0` if the column is empty.
- `get_height_of_column(column)` -- Returns the `row` number for the highest-numbered unoccupied row in the specified column. Returns `-1` if the column is full, returns 6 if the column is empty. NOTE: this is the row index number not the actual "height" of the column, and that row indices count from 0 at the top-most row down to 5 at the bottom-most row.
- `do_move(column)` -- Returns a new board with the current player's token added to `column`. The new board will indicate that it's now the other player's turn.
- `longest_chain(playerid)` -- Returns the length of the longest contiguous chain of tokens held by the player with the specified player ID. A 'chain' is as defined by the Connect Four rules, meaning that the first player to build a chain of length 4 wins the game.
- `chain_cells(playerid)` -- Returns a Python set containing tuples for each distinct chain (of length 1 or greater) of tokens controlled by the current player on the board.
- `num_tokens_on_board()` -- Returns the total number of tokens on the board (for either player). This can be used as a game progress meter of sorts, since the number increases by exactly one each turn.
- `is_win()` -- Returns the *player ID number* of the player who has won, or `0`.
- `is_game_over()` -- Returns true if the game has come to a conclusion. Use `is_win` to determine the winner.

Note also that, because ConnectFourBoards are immutable, they can be used as dictionary keys and they can be inserted into Python set() objects.

Other Useful Functions

There are a number of other useful functions in this lab, that are not members of a class. They include the following:

- `get_all_next_moves(board)` (`basicplayer.py`) -- Returns a generator of all moves that could be made on the current board
- `is_terminal(depth, board)` (`basicplayer.py`) -- Returns true if either a depth of 0 is reached or the board is in the game over state.
- `run_search_function(board, search_fn, eval_fn, timeout)` -- (`util.py`) -- Runs the specified search function with iterative deepening, for the specified amount of time. Described in more detail below.
- `human_player` (`connectfour.py`) -- A special player, that prompts the user for where to place its token. See below for documentation on "players".
- `count_runs()` (`util.py`) -- This is a Python Decorator callable that counts how many times the function that it decorates, has been called. See the decorator's definition for usage instructions. Can be useful for confirming that you have implemented alpha-beta pruning properly: you can decorate your evaluator function and verify that it's being called the correct number of times.
- `run_game(player1, player2, board = ConnectFourBoard())` (`connectfour.py`) -- Runs a game of Connect Four using the two specified players. 'board' can be specified if you want to start off on a board with some initial state.

Writing your Search Algorithm

"evaluate" Functions

In this lab, you will implement two evaluation functions for the minimax and alpha-beta searches: `focused_evaluate`, and `better_evaluate`. Evaluate functions take one argument, an instance of ConnectFourBoard. They return an integer that indicates how favorable the board is to the current player.

The intent of `focused_evaluate` is to simply get your feet wet in the wild-and-crazy world of evaluation functions. So the function is really meant to be very simple. You just want to make your player win more quickly, or lose more slowly.

The intent of `better_evaluate` is to go beyond simple static evaluation functions, and getting your function to beat the default evaluation function: `basic_evaluate`. There are multiple ways to do this but the most-common solutions involve knowing how far you are into the game at any given time. Also note that, each turn, one token is added to the board. There are a few functions on ConnectFourBoard objects that tell you about the tokens on the board; you may be able to use one of these. You can look at the source for the evaluation function you are trying to beat by looking at `def basic_evaluate(board)` in `basicplayer.py`.

Search Functions

As part of this lab, you must implement an alpha-beta search algorithm. Feel free to follow the model set by `minimax`, in `basicplayer.py`.

Your `alpha_beta_search` function must take the following arguments:

- `board` -- The ConnectFourBoard instance representing the current state of the game
- `depth` -- The maximum depth of the search tree to scan.
- `eval_fn` -- The "evaluate" function to use to evaluate board positions

And optionally it takes two more function arguments:

- `get_next_moves_fn` -- a function that given a board/state, returns the successor board/states. By default get_next_moves_fn takes on the value of `basicplayer.get_all_next_moves`
- `is_terminal_fn` -- a function that given a depth and board/state, returns True or False. True if the board/state is a terminal node, and that static evaluation should take place. By default `is_terminal_fn` takes on the value of `basicplayer.is_terminal`

You should use these functions in your implementation to find next board/states and check termination conditions.

The search should return the *column number* that you want to add a token to. If you are experiencing massive tester errors, make sure that you're returning the column number and not the entire board!

TIP: We've added a filed called `tree_searcher.py` to help you debug problems with your alpha_beta_search implementation. It contains code that will test your alpha-beta-search on static game trees of the kind that you can work out by hand. To debug your alpha-beta-search, you should run: `python tree_searcher.py`; and visually check the output to see if your code return the correct expected next moves on simple game trees. Only after you've passed `tree_searcher` then should you go on and run the full tester.

Creating a Player

In order to play a game, you have to turn your search algorithm into a *player*. A player is a function that takes a board as its sole argument, and returns a number, the column that you want to add a piece to.

Note that these requirements are quite similar to the requirements for a search function. So, you can define a basic player as follows:

```
def my_player(board):
    return minimax(board, depth=3, eval_fn=focused_evaluate,
timeout=5)
```

or, more succinctly (but equivalently):

```
my_player = lambda board: minimax(board, depth=3,
eval_fn=focused_evaluate)
```

However, this assumes you want to evaluate only to a specific depth. In class, we discussed the concept of *iterative deepening*. We have provided the `run_search_function` helper function to create a player that does iterative deepening, based on a generic search function. You can create an iterative-deepening player as follows:

```
my_player = lambda board: run_search_function(board,
search_fn=minimax, eval_fn=focused_evaluate)
```

Just win already!

You may notice, when playing against the computer, that it seems to make plainly "stupid" moves. If you gain an advantage against the computer so that you are certain to win if you make the right moves, the computer may just roll over and let you win. Or, if the computer is certain to win, it may seem to "toy" with you by making irrelevant moves that don't change the outcome.

This isn't "stupid" from the point of view of a minimax search. If all moves lead to the same outcome, why does it matter which move the computer makes?

This isn't how people generally play games, though. People want to win as quickly as possible when they can win, and lose slowly so that their opponent has several opportunities to mess up. A small change to the basic player's static evaluation function will make the computer play this way too.

- In `lab3.py`, write a new evaluation function, `focused_evaluate`, which prefers winning positions when they happen sooner and losing positions when they happen later.

It will help to follow these guidelines:

- Leave the "normal" values (the ones that are like 1 or -2, not 1000 or -1000) alone. You don't need to change how the procedure evaluates positions that aren't guaranteed wins or losses.
- Indicate a certain win with a value that is greater than or equal to 1000, and a certain loss with a value that is less than or equal to -1000.
- Remember, `focus_evaluate` should be very simple. So don't introduce any fancy heuristics in here, save your ideas for when you implement `better_evaluate` later on.

Alpha-beta search

The computerized players you've used so far would fit in well in the British Museum - they're evaluating all the positions down to a certain depth, even the useless ones. You can make them search much more efficiently by using alpha-beta search.

It may help to read the appropriate chapter of the text if you're unclear on the details of alpha-beta search.

- Write a procedure `alpha_beta_search`, which works like `minimax`, except it does alpha-beta pruning to reduce the search space.
- You should always return a result for a particular level as soon as alpha is greater than or equal to beta.
- `lab3.py` defines two values `INFINITY` and `NEG_INFINITY`, equal to positive and negative infinity; you should use these for the initial values of alpha and beta, as discussed in class.

This procedure is called by the player `alphabeta_player`, defined in `lab3.py`.

Your procedure will be tested with games and trees, so don't be surprised if the input doesn't always look like a Connect 4 board.

Hints

In class, we describe alpha-beta in terms of one player maximizing a value and the other person minimizing it. However, it will probably be easiest to write your code so that each player is trying to *maximize* the value from their own point of view. Whenever they look forward a step to the other player's move, they *negate* the resulting value to get a value for their own point of view. This is how the minimax function we provide works.

You will need to keep track of your range for alpha-beta search carefully, because you need to negate this range as well when you propagate it down the tree. If you have determined that valid scores for a move must be in the range [3, 5] -- in other words, alpha=3 and beta=5 -- then valid scores for the other player will be in the range [-5, -3]. So if you use this negation trick, you'll need to propagate alpha and beta like this in your recursive step:

```
newalpha = -beta
newbeta = -alpha
```

This is more compact than the representation we were using on the board in lecture, and it captures the insight that the player evaluates its opponent's choices just as if the player was making those choices itself.

Ask a TA if you are confused by this.

A common pitfall is to allow the search to go beyond the end of the game. So be sure to use is_terminal_fn to determine the end.

A better evaluation function

This problem is going to be a bit different. There's no single right way to do it - it will take a bit of creativity and thought about the game.

Your goal is to write a new procedure for static evaluation that outperforms the `basic_player` one we gave you. It is evaluated by the test case `run_test_game_1`, which plays `your_player` against `basic_player` in a tournament of 4 games. Clearly, if you just play basic_player against itself, each player will win about as often as it loses. We want you to do better, and design a player that wins at least 2 times more than it loses in the four games.

We should hardly need to state that your solutions should win legitimately, not by somehow interfering with the other player. This isn't Spassky vs. Fischer.

An additional warning, regarding this particular subject matter: There is a fair bit of published research and online information about "Connect Four", and about search algorithms in general in Python. You're welcome to read any documents that you'd like, as long as you cite them by including a comment in the relevant part of your code. However, you may not re-use complete third-party implementations of search algorithms, and you may not write code that access online resources (other than the 6.034 test server) while it is executing. The tester can't easily test for these, so it doesn't. The TA's will be looking over

your code, though, and they will not be amused if they find a student using someone else's work in their lab.

Relatedly, while it would be possible to pass the lab using `human_player`, we can't give you credit for doing so. The TA's will be amused if they find out that you did this, but they'll have to revoke credit for the relevant portions of the lab.

Advice

In an evaluation function, simpler can be better! It is much more useful to have time to search deeper in the tree than to perfectly express the value of a position. This is why in many games (not Connect Four) it takes some effort to beat the simple heuristic of "number of available moves"; you can't get much simpler than that and still have something to do with winning the game.

If you write a very complicated evaluation function, you won't have time to search as deep in the tree as `basic_evaluate`. Keep this in mind.

IMPORTANT NOTES

After you've implemented `better_evaluate`, please change the line in your `lab3.py` From

```
better_evaluate = memoize(basic_evaluate)
```

to

```
better_evaluate = memoize(better_evaluate)
```

The original setting was set to allow you play the game earlier on in the lab, but become unnecessary and incorrect once you've implemented your version of `better_evaluate`.

TIP: To save time, change the `if True` to `if False` on line 308 of tests.py. Disabling this test will skip the game test that usually takes a few minutes to finish. But be sure to remember to turn it back on when you have passed all your other offline tests!

```
# Set this if-guard to False temporarily disable this test.
if True:
    make_test(type = 'MULTIFUNCTION',
              getargs = run_test_game_1_getargs,
              testanswer = run_test_game_1_testanswer,
              expected_val = "You must win at least 2 more games than
you lose to pass this test",
              name = 'run_test_game'
              )
```

Tournament

Please set "COMPETE" to True if you would like to participate in a tournament. If there is enough interest/capable AIs then we will face you off against your fellow classmates and perhaps some

undisclosed prize. The first round will cull entries which can't beat the basic_player enough times so please set "COMPETE" to False if you haven't managed to pass that test.

Survey

Please answer these questions at the bottom of your `lab3.py` file:

- How many hours did this problem set take?
- Which parts of this problem set, if any, did you find interesting?
- Which parts of this problem set, if any, did you find boring or tedious?

FAQ

When you uncomment the line which memoizes your evaluator that improves the efficiency of your player over a single game where it sees the same board states many times. However, it doesn't take into account player_id so if you want to test an evaluation function using the test boards provided you should remove the memoization by commenting that line.

6.034 Artificial Intelligence

Fall 2010