Due Date: October 23, 2017 at 11:59pm

Submission Instructions: Assignments are to be submitted through LEARN, in the Dropbox labelled *Assignment 2 Submissions* in the Assignment 2 folder. Late assignments will be accepted up until October 26 at 11:59pm. Please read the course policy on assignments submitted after the official due date. *No assignment will be accepted, for any reason, after 11:59pm on October 26.*

Lead TA: Michael Tu (z3tu@uwaterloo.ca). Office hours on Friday (Oct 6) from 2:00-4:00pm and Thursday Oct 12 (2:00-4:00pm) in DC 2306C.

Announcements: The following exercises are to be done individually.

Question 1 (30 pts)

For this question you will write a program to solve TSP using simulated annealing. The setup for the problem is exactly the same as for the A*-search question in Assignment 1, and you are allowed to re-use any code you would like from your Assignment 1 submission. The data set for this problem is also identical to that used in Assignment 1, and I have posted a copy in the file TSPproblem.zip in the Assignment 2 folder.

a. Explain what local search operators you use (make sure that the operators preserve a tour).

b. Experiment with at least 3 different annealing schedules, and report your findings. What schedule would you chose to use?

c. Run the program on the 36-city TSP problem instance given in the file problem36. Plot how the cost of the solution changes during one run of the algorithm. What is the cost of the best solution that your algorithm found (do not let it run for more than 5 minutes)?

d. Is your version of simulated annealing complete? Explain why or why not.

e. Is it optimal? Explain why or why not.

Submit the following;

- A well documented copy of your code.
- A discussion of the local search operators you used.
- A discussion of the annealing schedules you used and how they influenced your search results.
- Your well-labeled plot from question *c*.
- Answers to questions posed in *d* and *e*.

Question 2 (10 points)

a. Prove the following assertion: For every game tree, the utility obtained by MAX using minimax search against a suboptimal MIN will never be lower than the utility obtained playing against an optimal MIN.

b. Can you come up with a game tree in which MAX can do better using a suboptimal strategy (that is, not using minimax search) against a suboptimal MIN? If yes, then draw the tree and explain how MAX can do better. If no, then explain why not.

Question 3 (15 points)

a. Describe how the minimax and alpha-beta algorithms change for two-player, non-zero sum games in which each player has a distinct utility function and both utility functions are known to each player.

b. If there are no constraints on the two utility functions, if it possible for any node to be pruned by alpha-beta? Why or why not?

c. What is the players' utility functions on any state differ by at most a constant k, making the game almost cooperative. Is it possible for any node to be pruned by alpha-beta? Why or why not?

Question 4 (45 points)

This question looks long but **Don't Panic!**. Most of this question is detailed documentation!

In this question, you will design an agent that plays the popular game of "Connect Four"[1]. We have provided an interactive framework for playing the game, including the rules of the game and a rudimentary basic player utilizing minimax search. You will implement a better agent that uses alpha-beta pruning and better evaluation functions.

If you are interested, as an *optional* component of this assignment, you may enter your agent in a tournament against other submitted agents.

Notes

- We are providing code for this question. Download a2-programming.zip. You should find the following `basicplayer.py, connectfour.py, implementation,py, LISCENCE, main.py, tests.py, tree_searcher.py, util.py`. The only file you need to modify is `implementation.py`.

[1]This assignment and the source code for this Connect Four implementation are based on material from MIT OpenCourseWare (OCW). They have been modified from the original distribution. This is distributed under a Creative Commons License.

- You should use Python 3 for this assignment.
- Your answers for the programming portion belong in the file `implementation.py`. You may change the other files for debugging purposes, but before you submit your assignment, you should test with the other files in their original state. When your code is tested on our end all files besides `implementation.py` will be replaced by the original copy.
- A number of basic tests have been provided for you in `tests.py`, to test basic competence of your agent. Please note that they are by no means exhaustive, and passing them is not a guarantee of good performance. Before running the full set of tests, run `python -m unittest tests.TestAlphaBetaSearch`. Only after you've passed these tests then should you go on and run the full tester, which may be accessed by running `python -m unittest`.
- **Remember**: Your `implementation.py` must contain your student ID and a name for your agent. To do this, uncomment and edit the appropriate lines at the start of the file. If you wish to participate in an optional tournament with other agents, please set `COMPETE` to True.
- In a separate file (`report.txt` or `report.pdf`, for example), you will include a short description of the "better" evaluation function you create and document your design choices for it (see Part (c)).
- **Submit**: Upload to the LEARN Dropbox your report and your `implementation.py` file. If you have additional `.py` files that are required for your code to run, you may submit those as well.

Introduction to Connect Four

This question is about adversarial search, and it will focus on the game "Connect Four".

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
```

You can get a feel for how the game works by playing it against the computer. For example, you can play O, while a computer player that does minimax search to depth 4 plays X: `python`

`main.py O`. See `python main.py -h` for a list of all options.

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 (X) puts a token in column 2

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

Pick a column #: -->
```

In this game, Player 1 just added a token to Column 2. 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).

The Code

ConnectFourBoard

`connectfour.py` contains a class entitled `ConnectFourBoard`. `ConnectFourBoard` objects are immutable. To make a move on a board, 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
```

You are welcome to call any methods associated with the object `ConnectFourBoard`. However, these are the key ones that you will use the most:

- `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.

Other Useful Functions

- `get_all_next_moves(board)` in `basicplayer.py` – Returns a generator of all moves that could be made on the current board
- `is_terminal(depth, board)` in `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)` in `util.py` – Runs the specified search function with iterative deepening, for the specified amount of time. Described in more detail below.
- `human_player` in `connectfour.py` – A special player, that prompts the user for where to place its token. See below for documentation on "players".

- `run_game(player1, player2, board = ConnectFourBoard())` in `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.

To play a game, you need to turn a 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. 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)
```

Writing a ConnectFour Agent

The goal of this question is to create a Connect Four playing agent using alpha-beta pruning and evaluation functions. We have provided you with code for an agent that uses minimax search with a rather silly evaluation function. Can you beat it?

The first thing you need to do is create a better evaluation function than the one we provide. An evaluation function takes one argument, an instance of `ConnectFourBoard`. They return an integer that indicates how favourable the board is to the current player. The problem with the current evaluation function (`basic_evaluate`) is that it treats all winning positions the same, no matter when they might occur (similarly with losing position) and this can sometimes lead to odd behaviour of the program, and certainly does not reflect how people may play who tend to like to reach winning positions sooner and losing positions later.

a. (Focused Evaluation Function) (5 points): In `implementation.py`, write a new evaluation function, `focused_evaluate`, which prefers winning positions when they happen sooner and losing positions when they happen later.

Here are some suggestions.

1. Look carefully at the `basic_evaluate` function.
2. 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.
3. 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.
4. Remember, `focused_evaluate` should be very simple and fast. The point of this exercise is mostly to familiarize you with the code and the basic evaluation function structure. Later in the assignment you are asked to develop something even more sophisticated.

You do **not** need to include your description of `focused_evalue` in your final report.

b. Alpha-Beta Pruning (25 points): In `implementation.py`, write a procedure `alpha_beta_search` which does alpha-beta pruning to reduce the search space used by minimax search.
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!

Other notes:

- 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.
- `util.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.
- This procedure is called by the player `alphabeta_player`. This is defined in `implementation.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.

c. A Better Evaluation Function (15 points):

Your goal is to implement a new procedure for static evaluation that outperforms the evaluation function in the `basic_player` provided to you; this procedure is called `better_evaluate` and a stub is provided for you in `implementation.py`. It is evaluated by the test case TestConnectFourPlay, which plays `my_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 with a goal of producing a player that wins at least twice as often as it loses.

Once you have designed the ultimate Connect Four evaluation function, include a short (max 500 word) description of your design choices. In particular, you should explain how your evaluation function works and why you believe it to be the best Connect Four agent.

A couple of additional notes:

- Your agent should win legitimately! It can not interfere with the other player.
- Remember to cite your references! There is a lot of information about the game of Connect Four available. You are welcome to read any of the work you like, but you must cite it. Citatations can be either in the written report or included in your code documentation. You may not use code that accesses online resources, not can you directly copy or paste someone else's implementation of alpha-beta search.

Advice

The rule of thumb with evaluation functions is that simpler is often better. Note that your time is limited. Given the choice between using that time to search deeper vs using it on a complex evaluation function, searching deeper is almost always better.

The full set of tests can take a long time to run. You can pick to only run a subset of tests, for example, `python -m unittest tests.TestAlphaBetaSearch`. You may then run the full tests via `python -m unittest`.

Important Notes

- After you've implemented `better_evaluate`, please comment out the following line in your `implementation.py`:

  `better_evaluate = memoize(basic_evaluate)`
  and then uncomment the following line:
  `better_evaluate = memoize(better_evaluate)`

  The original setting was set to allow you play the game initially, but it is unnecessary and incorrect once you've implemented your version of `better_evaluate`.

- Ensure you have entered your name and a name for your agent on the appropriate lines at the top of `implementation.py`.

- Set `COMPETE` to True (in `implementation.py`) if you would like to participate in a tournament. If there is enough interest then we will run a small tournament. This is just for fun and is entirely optional. No marks are awarded for participating. Similarly there are no penalties if you decide not to enter of if you enter and your agent does poorly.