CMPT310: Summer 2025:
Assignment 2: Adversarial searches
TicTacToa game

In this assignment various algorithms for adversarial search are to be implemented for generalized n x n TicTacToa game:

1. MinMax
2. MinMax with AlphaBeta Pruning
3. MonteCarlo Tree search (MCTS)

i) Currently game is playable with AI playing random.
There is a timer setting for the game that is set by default to -1. This means no time limit. Once you set the time limit to > 0, that time in second is used to limit the depth search. This means a cutoff variation of MinMax and AlphaBeta needs to be implemented as part of this assignment. Depth-cutoff version of the above algorithm is to be implemented as a separate function. There are drivers for minmax and alphabeta in games.py to select the right execution path depending on the value of the timer.

ii) There are 2 scripts for you to edit: games.py and Montecarlo.py. The area you must provide your implementation is specified by comments like saying "*your/student code goes here*" plus the points given for that part. In this assignment, you can add supporting function and code for the existing shell when needed, but not to circumvent them. In another word, you need to stay as close as possible to current shell project and provide your solution in the area specified.

iii) MonteCarlo Tree Search (MCTS):
MonteCarlo Tree Search algorithm's shell is given in monteCarlo.py. There are various parts in there for you to complete.

iv) You need to come up with a smart Evaluation function for the game. It must give highest value to the board configuration which has the most potential for a win by the AI player. Find out what characteristics in the current layout are more promising and then try to quantify them. A shell version of the evaluation function,**eval1()**, is given in game.py. There are different ways to achieve this, use your game skill, creativity, and time. There are some suggestions given there to start.

v) In various area in the code, there are comments explaining what is going on. Read the comment carefully since they explain what functionalities are expected and provides information how to achieve them.

vi) There is an important optimization regarding state caching which you must implement. Without this, you won't be able to achieve the expected performance below.

Once done, you should be able to play 3 x 3 TicTacToa for any of the algorithms with no time limit (meaning timer set to -1). Human player should not be able to beat AI player at any algorithm. For 4x4 and 5x5 you will need to set the timer to a positive value (3, 4, 5, 6), so it can be playable timewise. To debug, you can print out the depth value in the iterative deepening loop to see how much time each iteration take. Usually, timer=5 or 6 is the limit of playability. Higher value can cause long delays for AI to play. There are 4 run configurations 3 x 3, 4 x 4, 5 x 5, and 6 x 6 available when you load the project in PyCharm.

Observations:

1- Time limit set in the gui is **nominal**. This means the actual time the algorithm may take could be higher, and sometimes much higher!. This is the case for minmax and alphabeta cutoff version. It is because the time limit is checked at the beginning of each iteration, and since these algorithms are dfs based, they may take a long time to complete.
2- For minmax and alphabeta based algorithms, with proper implementation (using state caching), AI should be able to always tie the game with **nominal** time limit set to <=5. This is what we mean by saying this game is solved! Again, remember that for 6 x6 at the start, it may easily take up to 60 to 80 seconds for the AI to make a move. This time quickly comes done in subsequent moves.
3- MonteCarlo (MCTS) AI player is an *anytime* algorithm, so based on the time limit, it plays reasonably well, but its performance depends on the number of playout it manages to perform in that time limit. This number can be increased noticeably by using a state caching solution. MCTS can play a perfect game with proper implementation for 3x3 and 4x4 in real time (t<4 sec). It usually needs at least 15000 playout to be able to figure out best move properly. That is why it is hard to play decently for 5 x 5 or 6 x 6 under 5 seconds timelimt. Again, you can print out the playout count at the end of each move to see how many playout has been tried. Also print out the number of cached states to see how much they are important to save time. Pay attention do not print stuff while it is doing processing between the moves since it slows it down, also for testing do not run in debug mode.

Rubric:

This assignment rubric is based on correct implementation of each algorithm and the performance. The resulting performance numbers and quality of the agent would follow if the implementation are correct. In general, based on your computer specs, all algorithms should be playable with time limit of 4 or at most 5 seconds. Sometimes choosing 5 seconds for timer may result in longer time for AI to play (for minmax based algorithms, not for MonteCarlo MCTS) due to the last depth iteration may take longer to return. Grades are split among various parts of the assignment and specified in the related areas in the 2 py scripts.
Performance: As pointed out in various location above, you will need to implement a state caching strategy so your performance numbers are in the range. The exact design and implementation of such caching is left to you and no direct points are given to it. Missing performance range (meaning

algorithms worth with timer set at most to 5 or 6) results in 10 pts deduction overall, even if all individual sections are done properly.

For upload zip the 2 files (game.py, monteCarlo.py)and upload it. The TAs will insert them in their game shell and run it in various setting to check performance, then check your code for each section to make sure they are implemented as requested. It is **important** to limit the code modification in the 2 scripts to the specified area, except for caching strategy which may need its own separate functions or code sections.

Good luck.