Pedro de Vasconcellos Oporto
HW 3
CS 4341 - C term 2019
2 Connect-N project

The main modifications to the project implementation when compared to what was proposed in HW2 are regarding the code structure. The program was heavily simplified with now no classes needing to be created beyond the alpha_beta_agent. The caching of the board states skippable cells turned out to not improve the time it took for the agent to score the heuristic of successor states, as it also took significant space in memory.

The heuristic function itself did not change from previously. The strategy for evaluating a board is counting the number of sequences of N cells that could in future turns become a winning Connect-N sequence. This means these sequences have a combination of empty cells and cells from only one player. The value of this sequence is then evaluated based on how many cells from that player are in that sequence. For example (for N = 4):
“0010”, “1000”, and “0200” are sequences of value 1; “0101”, “0110” and “2200” are sequences of value 2; “1110”, “0222” and “1101” are sequences of value 3; “1111” and “2222” are sequences of value 4; “0000”, “2100”, “1112” and “2121” are sequences of value 0. These sequences will be counted by looping through every cell and the 4 possible directions (Vertical, Horizontal, +45 degrees, -45 degrees). A score will be kept for both players for each state (“1011” gives points to player 1 and “0022” gives points to player 2). It also stores the score weights for sequences of 0, 1, 2, 3 and 4 pieces owned by the same player

- Example: score_weights = [0, 10, 25, 100, 10000] means
- sequences of 0 get 0 points
- sequences of 1 get 10 points
- sequences of 2 get 25 points
- sequences of 3 get 100 points
- sequences of 4 get 10000 points

With both players scores calculated, the heuristic function combines them using an offensiveness/defensiveness weight balancing. The offensiveness is a value between 0 and 1 and the defensiveness is then calculated as (1 - offensiveness). The final score for a board = (offensiveness * its own score) - (defensiveness * opponent’s score). The heuristic is called utility_cn_1 and is within the alpha_beta_agent class. Several heuristics functions with this algorithm were compared. The variation between them was with the offensiveness and score_weights arguments.

The utility function was wrapped inside the max_value and min_value functions of the min-max alpha-beta algorithm provided by the lecture. In addition, the max_value function also returns the action *a* which corresponds to the column number that a piece would be placed next for the corresponding utility that is returned by the function. This is only needed for when it returns to the go function, which needs to know which action to take as the AI player. Furthermore, the terminal test was broken down into three major steps. The max_value and min_value functions first check if the given board state has a winner, by calling state.get_outcome. If there is a winner, it returns very high or low values as the utility, depending on who is winning. It then checks for a tie, or if the state has not more possible plays to be made. This is check by calling state.freecols() and making sure it returns a value higher than 0. Otherwise, it returns 0 as the utility. Lastly, it checks if the pruning reached the maximum given depth, which then triggers it to call the utility function on the given state. From that point on, if these terminal state options were not reached, the functions continue with the pruning down the successor states.

At first, the maximum depth for the agent was defined in its instantiation. By testing multiple agents against each other and against random players, benefits to both higher and lower depths were clear. The tests were made by inserting multiple agents in a tournament, with a mix of random agents and alpha-beta agents that took in different parameters on instantiation (offensiveness, score weights, and maximum depth). At first, very distinct values were alternated, and smaller variations were then made on the winning configurations for a new tournament set and so on. This primitive iteration of the agent's configuration could be in the future improved by using a genetic algorithm running tournaments within generations. The genetic "code" would be the combination of the offensiveness and score weights attributes, as the max depth would be more in function of the CPU capacity.

- Offensiveness took values between 0.2 and 0.8 at every 0.05 increment
- Score weights varied between [0,10,25,100,1000], [0,10,50,250,10000], [0,10,100,10000], [0,10,50,500,10000], [0,10,50,500,10000], [0,10,50,5000,100000]
- Max depth values varied from 2 to 8

Example tournament set up with random agents:

```
agents = [
   agent.RandomAgent("random1"),
   agent.RandomAgent("random2"),
   agent.RandomAgent("random3"),
   aba.AlphaBetaAgent("alphabeta6DEF", 6, [0,10,50,5000,1000000], 0.30),
   aba.AlphaBetaAgent("alphabeta6ATK", 6, [0,10,50,5000,1000000], 0.40),
   aba.AlphaBetaAgent("alphabeta6",6,  [0,10,50,5000,1000000], 0.35)
]
```

Example result:

```
SCORES:
14 alphabeta6ATK
12 alphabeta6DEF
9 alphabeta6
-2 random2
-4 random3
-4 random1
```

Example tournament set up with only alpha beta agents for a longer runtime:

```
agents = [
    aba.AlphaBetaAgent("alphabetaDEF" + str(x), 6, [0,10,50,5000,1000000], x*0.01) for x in
range(30,41)
]
```

The max_depth argument was initially implemented for the alpha-beta agent but was then abandoned, as different depths showed to be better options in different stages of the game. Higher depth calculations such as 6 and 8 turned out to inconsistently go over the time limit for a turn (15 seconds). Although this might have been due to the code being run in a Chromebook, the agent was later changed to slowly increase the depth it goes with calculations as turns went by. The reasoning for this is that for later stages in the game, a tree of depth 8, for example, would hit several terminal states more often before reaching the bottom of the search tree, making it narrower. This reduces the number of calculations and thus the turn time. Moreover, it was noted for late enough stages in the game, the AI could actually navigate all the way to the end of the game as the maximum depth. For example, with 18 plays left for the board to be full, and the AI calculated all possible future states (18 depth) quick enough and return the complete and optimal play. The staging parameters for the provided code are simply defined by the performance of my machine and could be changed for better AI capabilities. The depth for a turn is defined at the beginning of the go function and is based on the board size, N in Connect-N, and the current number of pieces already placed on the board. This AI could be severely improved in the future with a more systematic staging development. Depth starts at a value > 0 and is decreased every max_value or min_value call until it reaches the value of 0 as a terminal state. The following is the code that defined the depth value at the beginning of the go method:

```
if turn < brd.h:
        depth = 2
elif turn < 2 * brd.h + brd.w:
        if brd.w < 10 or turn > 0.5*brd.h + brd.w:
                depth = 4
        else:
                depth = 3
elif (turn < (brd.h-2) * (brd.w-1) and brd.n == 4) or turn < (brd.h-1) * (brd.w-1):
        depth = 6
        else:
                depth = (brd.h * brd.w) - turn
```