Lab 1: Box Game

Organization:

In a classic example of overthinking, I used Java to write classes for various game objects. I originally thought the BoxGame class would consist of some kind of game state and it would be self-playing. The remnants of thought-process remain but are largely ignored to match the API. The individual components are as follows:

- BoxGame - the principle component; holds game logic and rules to evaluate the board
- BoardPosition - a simple positional class holding the row and column location for a 2D array. This allowed for more simple traversal with the add method. This is hashable to keep track of visited positions when evaluating the state of a Board.
- InvalidBoardException - a lazy way to bypass writing bounds when attempting to traverse/manipulate the Board. Used to squash OutOfBounds exceptions and illegal piece placements.
- Board - a square, 2-dimensional array of characters. The static variables are the representations of available spaces and obstacles. The update method takes in a char (the new value) and any number of BoardPositions to attempt to update. If any BoardPosition is out of bounds or attempts to update an unavailable space, then the method throws an InvalidBoardException.
- Piece (should be called Box) - Represents different size boxes that can be placed on a Board. Created to transform locations into groups of BoardPosition coordinates. Would hopefully be used to scale.
- Strategy - Vestigial enum to keep track of a Player's approach (Minimax or AlphaBeta).
- Player - Class representing an agent. Instances hold a char id and Pieces that are used to update a Board by generating possible states based on placing Piece coordinates.

Evaluation functions:

score - General terminal function, typically evaluates a Board when there are no more available moves to play. A fairly simple calculation, it calculates the number of points scored by the current maximizing Player and subtracting points scored by the other(s). As the Board is traversed, a BFS is performed on Player spaces. Since all Pieces are multiples of 6, the points are divided proportionally. Sets are used to keep track of which positions have been visited or are enqueued to be visited in constant lookup time.

greedyPoints - Similar to score except does not account for opponent score(s). Evaluate a board per game rule points given to the maximizing player. In theory (ostensibly), since points are scored by connecting boxes, choosing a state with more points should create a larger perimeter to score additional points on.

blockMost - A scan through the board to determine if a player's piece blocks another player piece, in a hope to prevent it from being used to connect another playing piece (again, ostensibly).

Results

A printout of the results can be found in final_tests.txt. The longest running test was AB1vAB2, which could be explained by the larger search depth but I am a bit concerned that the problem lies with my evaluation method blockMost. I would have more expected the approach to succeed in creating a draw. I thought that this may have to do with the swap between the eval method blockMost and terminal method score, but the more I think the less that makes sense because terminal boards would still be able to prioritize net neutral scores over opponent points blocked. The highest win count and score were for AB0vAB0, with all 25 wins by 1 point. The lowest win count was 1 win, averaging 0.04 or 24 draws. This generally seems to align with the idea that prioritizing scoring (offensive) leads to better results than prioritizing blocking (defensive). Interestingly, MM1vMM2 and AB1vAB2 had averages lower than their win rate divided by runs (or winning by one point). This seems to suggest that there is at least one game in each where player 2 is able to win. This aligns with the players using different strategies being less predictable for the other if neither uses a fully exhaustive search.