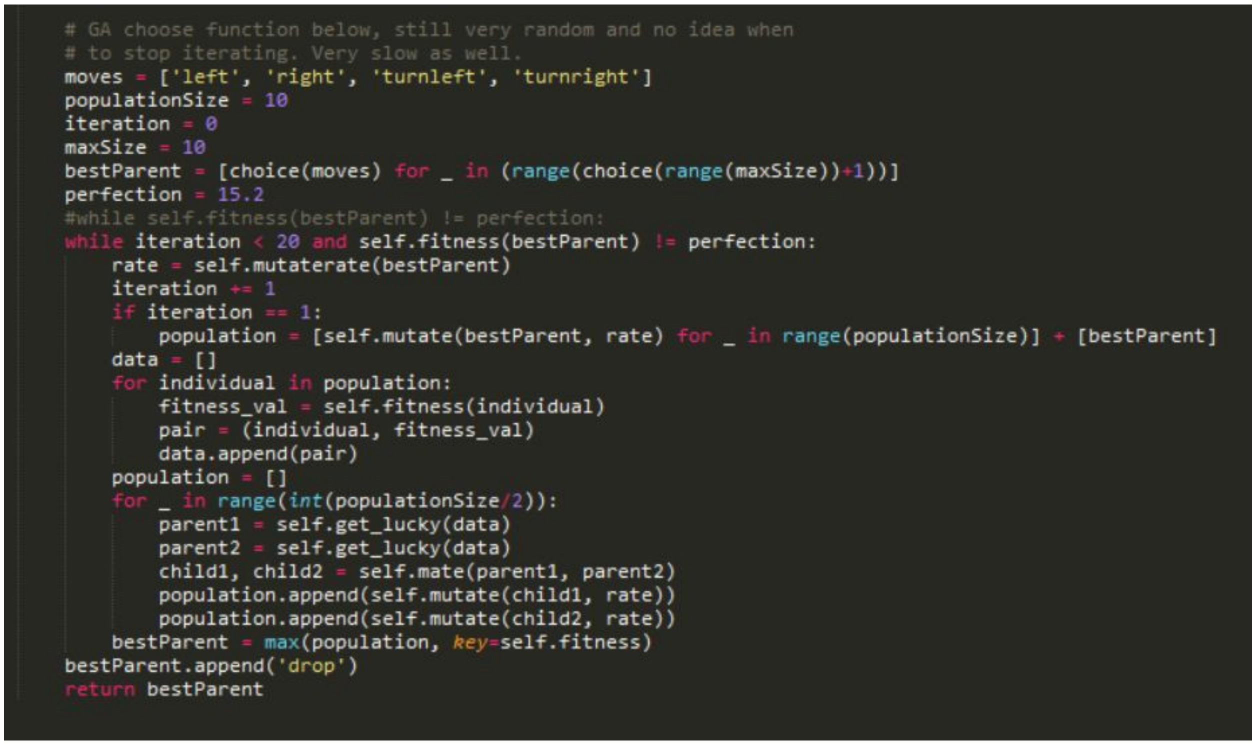
Name: Raymond Zhu
ID: 27841881

Introduction

This bot is programmed to play the game AI Block Battle. The problem being solved by this bot is which move should be picked in order to pick the best set of moves for the current situation. The set of moves to be selected from are: left, right, turnleft, turnright, down and drop. Now the features that are hard about this problem is how much should a bot care for a certain feature being calculated about the current field being played on. For example, if the bot places a piece all the way to the left it will reduce the amount of holes on the field, but also increase the maximum height of that column of blocks on the field. In this case, both features can be a negative feature. What move should the bot choose to avoid these negative features, and how should these negative and positive features be weighed versus other features? Another feature that's hard in this game is whether to be more aggressive in clearing large amount of blocks, or to be safer clearing fewer blocks. For example, a bot could be safer by clearing one line at a time when it gets the chance to, but this doesn't allow for the bot to utilize the blocks height to later clear multiple lines rather than one. A bot could also be more aggressive by clearing more rows by stacking blocks on one side and adding a small amount of blocks to the other side. Obviously, the more aggressive side has higher risk, but also high reward because it'll send garbage lines to the opponent's field, making it harder for the opponent's bot to win. In my implementation the bot considers the whole board instead of just the adjacent blocks. This will allow the bot to readily access all information of the board without having to always simulate moves to check which spot is the best to place the current piece being operated on. I have also implemented both type of bots, aggressive and passive. My newer versions try to adjust the weights of the features to make the bot more passive, but my older bots, such as version 98, was much more aggressive, and lost even with wide point leads.

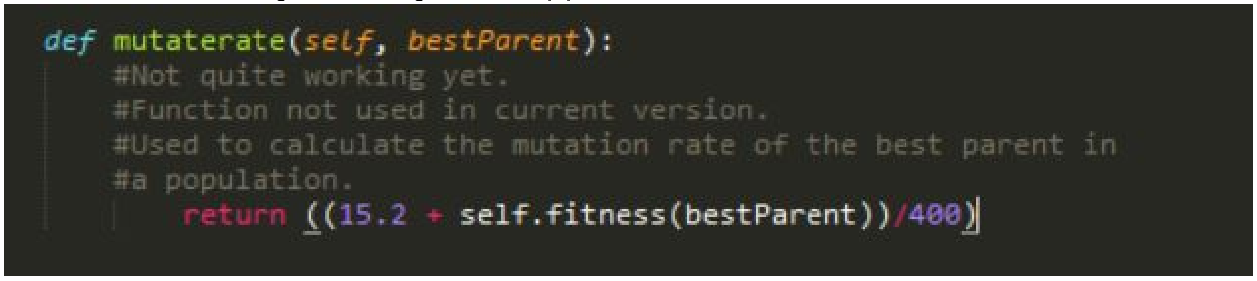
Methods

The first programming method I tried was a genetic algorithm approach. This did not end up being my approach in the final version of the bot, but was attempted in earlier versions of the bot. The reasons this wasn't in my final version was because the genetic algorithm bot was still very random in its selection of moves compared to the random strategy that was given to us in the starter bot. Not only that, but the problem with a genetic algorithm approach is that there's no clear criteria to stop running the algorithm besides a "good enough" fitness value. Then the problem would be what is a good enough fitness value? The answer could vary depending on the current situation and the field. Also, if a value is deemed "good enough" before a better move is found, the worse move is chosen because it appeared before the better solution. Another approach, would be to run the genetic algorithm for a set number of iterations and pick the best set of moves found during that iteration. Obviously this has its own flaws, such as missing a better move that could be done and the number of tests that need to be done to pick the iteration value. The approach for the genetic algorithms choose function, which is the function that actually chooses the moves to be done by the bot on the field, is shown here:

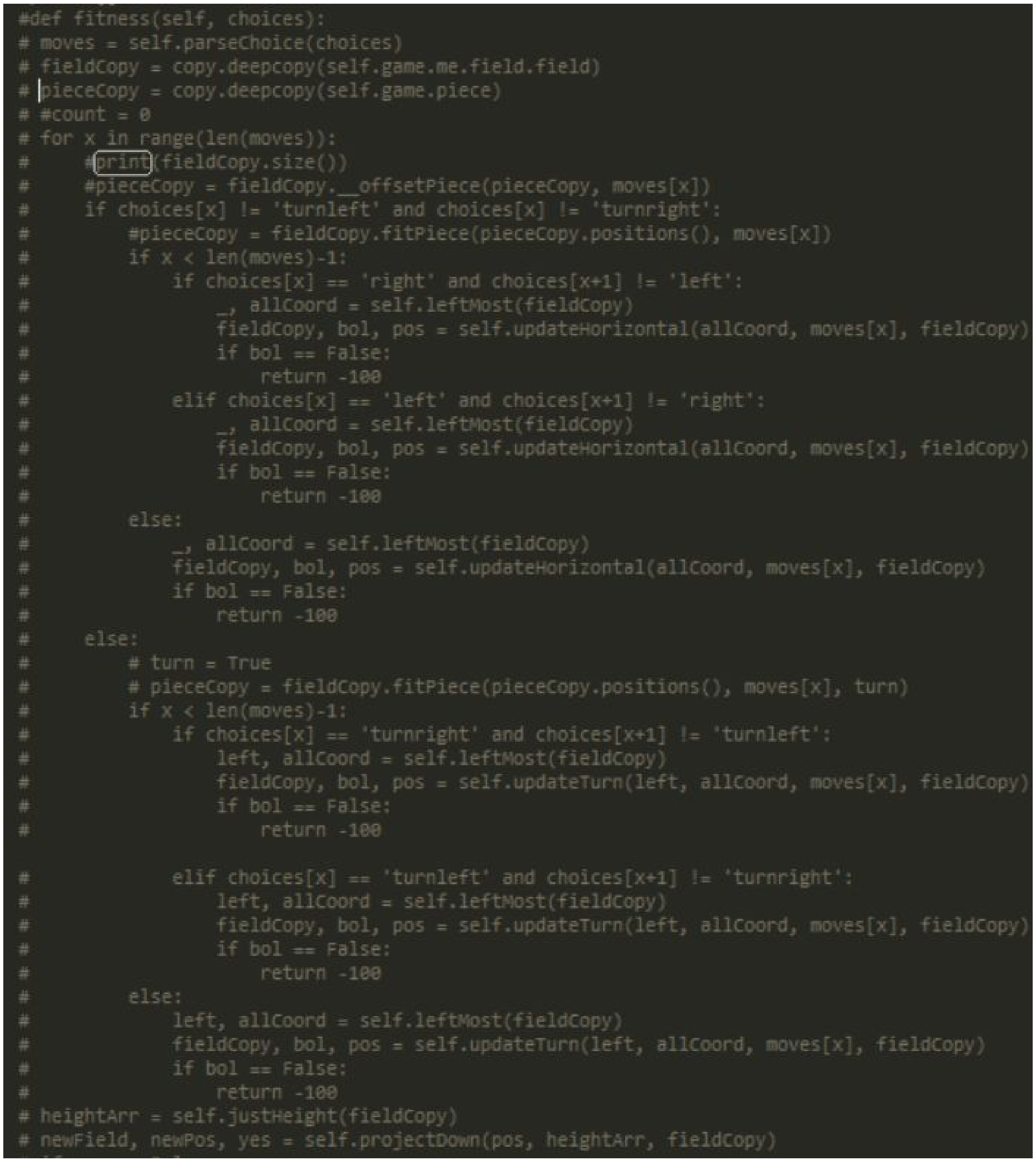


```
# GA choose function below, still very random and no idea when
# to stop iterating. Very slow as well.
moves = ['left', 'right', 'turnleft', 'turnright']
populationSize = 10
iteration = 0
maxSize = 10
bestParent = [choice(moves) for _ in (range(choice(range(maxSize))+1))]
perfection = 15.2
#while self.fitness(bestParent) != perfection:
while iteration < 20 and self.fitness(bestParent) != perfection:
    rate = self.mutaterate(bestParent)
    iteration += 1
    if iteration == 1:
        population = [self.mutate(bestParent, rate) for _ in range(populationSize)] + [bestParent]
    data = []
    for individual in population:
        fitness_val = self.fitness(individual)
        pair = (individual, fitness_val)
        data.append(pair)
    population = []
    for _ in range(int(populationSize/2)):
        parent1 = self.get_lucky(data)
        parent2 = self.get_lucky(data)
        child1, child2 = self.mate(parent1, parent2)
        population.append(self.mutate(child1, rate))
        population.append(self.mutate(child2, rate))
    bestParent = max(population, key=self.fitness)
bestParent.append('drop')
return bestParent
```

Another hard problem about using the genetic algorithms approach is deciding on a good mutation rate, and deciding what the perfect fitness would be if it was to be used as a criteria in the while loop shown above. The attempt at a fitness function and mutation rate for a genetic algorithm approach can be seen below:



```
def mutaterate(self, bestParent):
    #Not quite working yet.
    #Function not used in current version.
    #Used to calculate the mutation rate of the best parent in
    #a population.
        return ((15.2 + self.fitness(bestParent))/400)
```



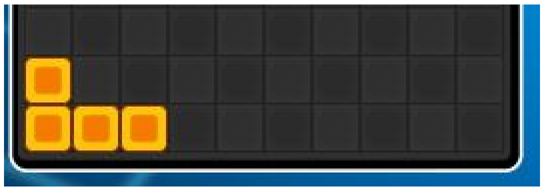
```
#def fitness(self, choices):
# moves = self.parseChoice(choices)
# fieldCopy = copy.deepcopy(self.game.me.field.field)
# pieceCopy = copy.deepcopy(self.game.piece)
# #count = 0
# for x in range(len(moves)):
#     #print(fieldCopy.size())
#     #pieceCopy = fieldCopy.__offsetPiece(pieceCopy, moves[x])
#     if choices[x] != 'turnleft' and choices[x] != 'turnright':
#         #pieceCopy = fieldCopy.fitPiece(pieceCopy.positions(), moves[x])
#         if x < len(moves)-1:
#             if choices[x] == 'right' and choices[x+1] != 'left':
#                 _, allCoord = self.leftMost(fieldCopy)
#                 fieldCopy, bol, pos = self.updateHorizontal(allCoord, moves[x], fieldCopy)
#                 if bol == False:
#                     return -100
#             elif choices[x] == 'left' and choices[x+1] != 'right':
#                 _, allCoord = self.leftMost(fieldCopy)
#                 fieldCopy, bol, pos = self.updateHorizontal(allCoord, moves[x], fieldCopy)
#                 if bol == False:
#                     return -100
#         else:
#             _, allCoord = self.leftMost(fieldCopy)
#             fieldCopy, bol, pos = self.updateHorizontal(allCoord, moves[x], fieldCopy)
#             if bol == False:
#                 return -100
#     else:
#         # turn = True
#         # pieceCopy = fieldCopy.fitPiece(pieceCopy.positions(), moves[x], turn)
#         if x < len(moves)-1:
#             if choices[x] == 'turnright' and choices[x+1] != 'turnleft':
#                 left, allCoord = self.leftMost(fieldCopy)
#                 fieldCopy, bol, pos = self.updateTurn(left, allCoord, moves[x], fieldCopy)
#                 if bol == False:
#                     return -100

#             elif choices[x] == 'turnleft' and choices[x+1] != 'turnright':
#                 left, allCoord = self.leftMost(fieldCopy)
#                 fieldCopy, bol, pos = self.updateTurn(left, allCoord, moves[x], fieldCopy)
#                 if bol == False:
#                     return -100
#         else:
#             left, allCoord = self.leftMost(fieldCopy)
#             fieldCopy, bol, pos = self.updateTurn(left, allCoord, moves[x], fieldCopy)
#             if bol == False:
#                 return -100
# heightArr = self.justHeight(fieldCopy)
# newField, newPos, yes = self.projectDown(pos, heightArr, fieldCopy)
```

The next problem with my genetic algorithm approach was that the above fitness function would've been very slow for sets of moves, which in this function is called choices, that are large. Thus, another problem with the genetic algorithm approach is what size should the max size be for any set of moves being generated? As seen in my choose function the max size I chose is declared in the variable called maxSize, which is given the value ten. Even with this limit there needs to be a limit to how big the population can be. This is another problem with the genetic algorithm approach, deciding the best population size. Given the time limit by the time bank for the whole game, a move needs to be calculated very fast. Balancing the iteration and population size to fit this need causes run times to either be very slow, meaning the bot will not last long in games that should go for a long time, or the moves to be very random, and thus this genetic algorithm approach being useless. This is now more of a problem of the accuracy of the solution versus the time complexity of the algorithm and iterations being done. Thus, the genetic algorithm approach was abandoned for this bot.

In terms of the actual approach for this genetic algorithm, there would a list of possible moves to select from initially, in this case it was left, right, turn left or turn right. This would be the initial best parent, which is generated randomly from the selection mentioned before for some max size, in this case ten. It would then iterate through twenty times, or until it finds a set of moves that achieved the perfect fitness. The function would then calculate the mutation rate of the best parent, the better solutions would have a fitness close to zero in this genetics approach. This is because the original implementation was originally going to use mostly negative weights, and then get multiplied by some positive number that would hold a value for a feature. Then if this was the first iteration it would generate a population given a max population size by mutating the original parent by its mutation rate. Next, we record the pairs of individual within the population with their respective fitness value. After emptying the current population to create a new population, we take the array of pairs and use our function to calculate the two parents that'll mate within this iteration. During this iteration we'll attempt to mutate the children given a mutation rate. Lastly, we determine the best parent for the next generation from the current children by using their respective fitness value.

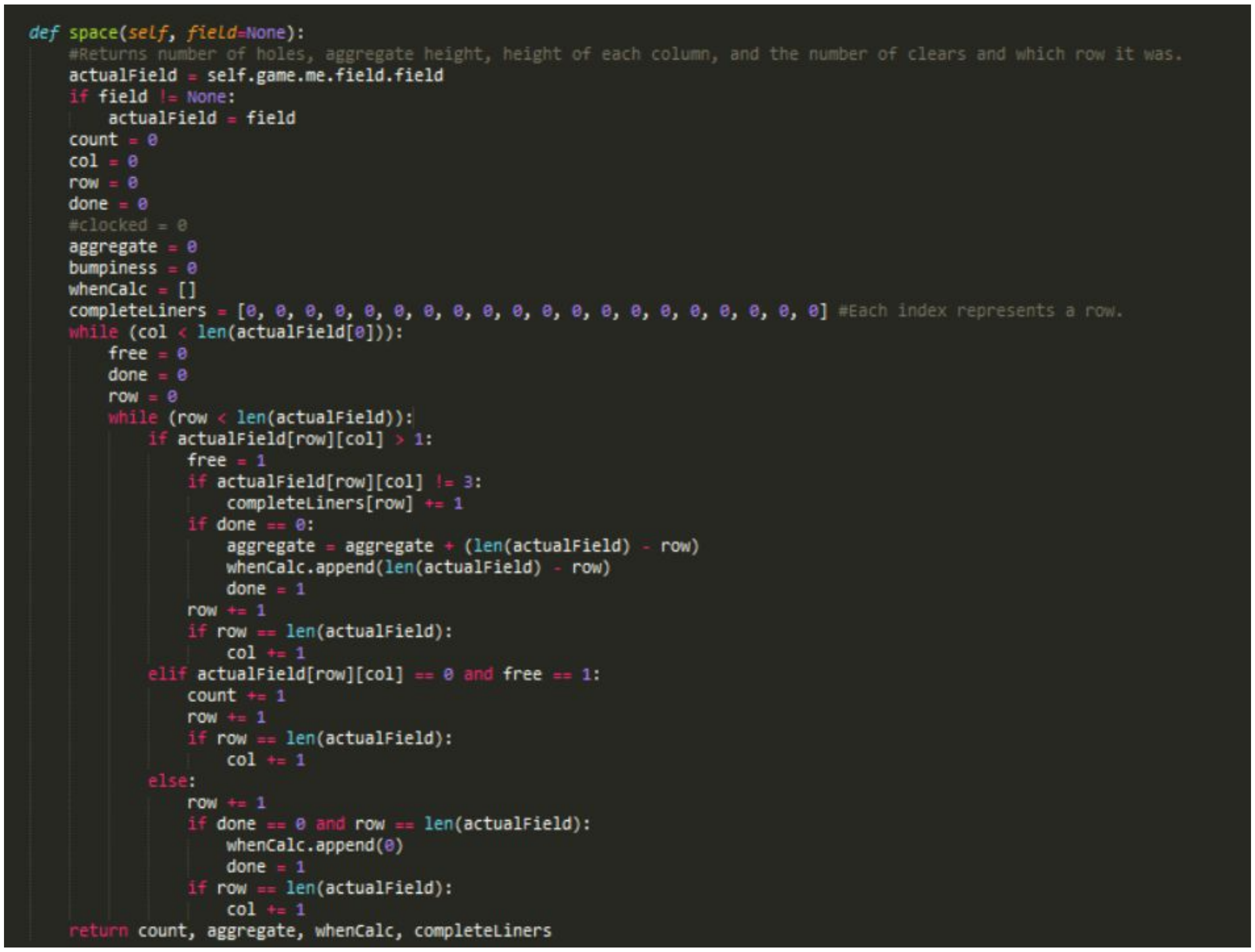
This implementation allows for the better parents to be more likely to mate with each other, while the individuals that had a bad fitness score has a smaller chance to mate with others. This would also allow for a variety of parents to be selected rather than just the best two parents, which would help spread traits within the population to create a better child. Then these better children can breed new children to become even better. Given a good enough iteration a child could hold a very good set of moves for the given field.

The next approach that was taken into consideration was reinforcement learning. No code was programmed for this approach. This is because before a code should be written there should be some thought put into it in order to avoid abandoning a code that took a lot of effort to write. Thus, my thought was what's the problems with the reinforcement learning approach? One problem that I thought of was, what would the reward be for the bot as the bot is performing the moves selected from this approach? If a reward was based off your current score the bot could think a perfectly good move in the beginning was bad because it didn't receive any points for it, but in reality there's just not enough blocks to clear a row that early in the game. Take this J piece, for example:



Your points would still be zero, but the choice of moves to get to that position was perfectly reasonable. If you used zero as a non-negative reward, but non-positive as well, you would be rewarding moves that could be potentially bad, such as just stacking the pieces together to form one big tower. If we used games won as a reward, we'd need a separate text file with a record on that game, but then the bot would be more programmed as the game being a state, and all the actions being done as one action to process a reward for the state to action relationship. This would also present the problem of what would the reward function reward for good games versus bad. Not only that, but generating a state list would be enormous for our game. It would have to encompass all the possible combinations that could be done on the board. In terms of using the characteristics of the game currently being played for the state, it would be better than having a game history, but it would still need to model all possible combinations that could happen in that one game with all the pieces available for the Block Battle game. You'd have to model all possible rotations per location, all positions possible at a certain height, and then even throw in the unclearable blocks as well as the garbage lines sent from the opponent. Accounting for all these states would be time consuming and unnecessary. Lastly, a single bad mistake can cost you in Block Battle. If you're busy waiting to be rewarded for your deeds a bot that already knows what's good and bad will have the upperhand in the beginning. Thus, this approach wasn't used in the final version of the bot.

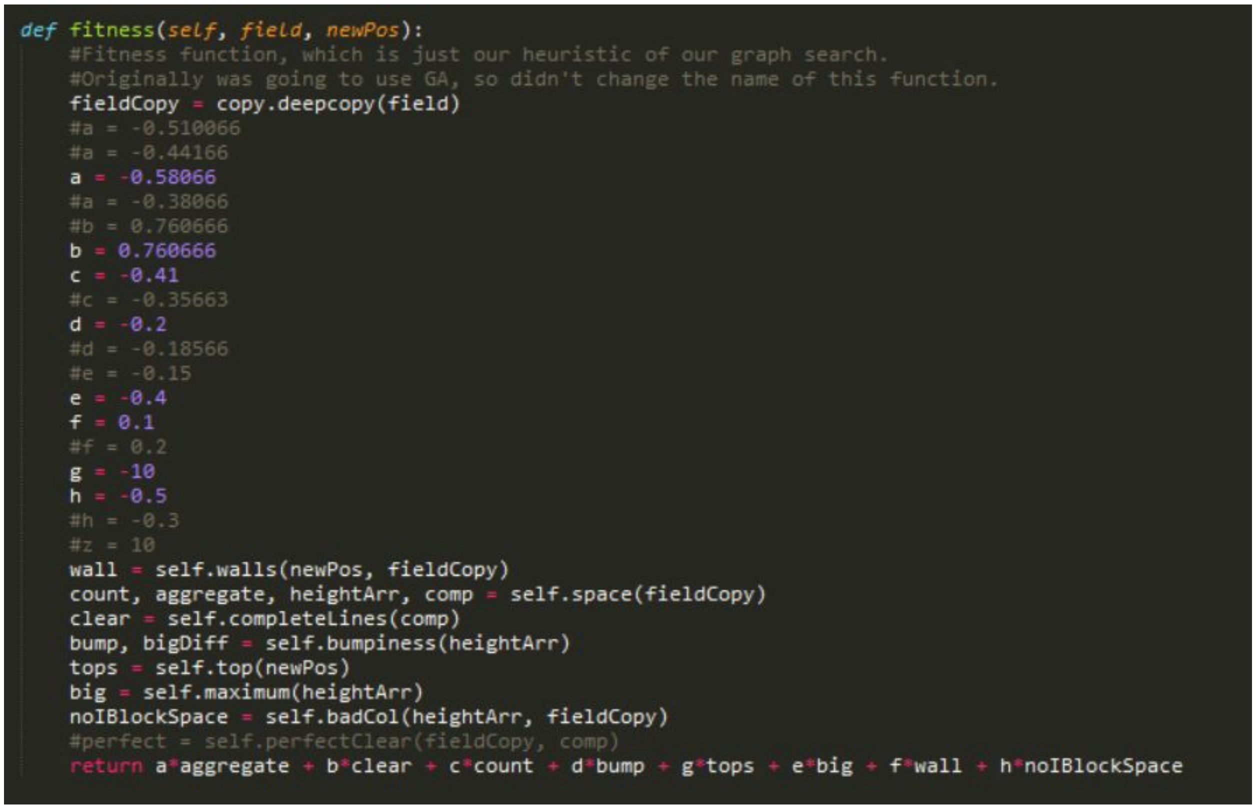
The next approach was a simple programming approach that searched the current field for the height of each column, as well as other features. This approach was very simple because it only accounted for heights, but not other notable features such as holes and bumpiness when selecting a move. This is obviously not a good solution to our problem, but it's simpler and faster than the approaches above. This is also less random than the genetic algorithm approach and doesn't rely on the reward after the action is done. This approach wasn't good because it doesn't have enough features calculated to accurately make a move. Thus, expanding this approach we calculate more features within the same function that calculates the heights per column. If the bot calculated these features separately the code would be more readable, but also increase time complexity. This code can be seen here:



```python
def space(self, field=None):
    #Returns number of holes, aggregate height, height of each column, and the number of clears and which row it was.
    actualField = self.game.me.field.field
    if field != None:
        actualField = field
    count = 0
    col = 0
    row = 0
    done = 0
    #clocked = 0
    aggregate = 0
    bumpiness = 0
    whenCalc = []
    completeLiners = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] #Each index represents a row.
    while (col < len(actualField[0])):
        free = 0
        done = 0
        row = 0
        while (row < len(actualField)):
            if actualField[row][col] > 1:
                free = 1
                if actualField[row][col] != 3:
                    completeLiners[row] += 1
                if done == 0:
                    aggregate = aggregate + (len(actualField) - row)
                    whenCalc.append(len(actualField) - row)
                    done = 1
                row += 1
                if row == len(actualField):
                    col += 1
            elif actualField[row][col] == 0 and free == 1:
                count += 1
                row += 1
                if row == len(actualField):
                    col += 1
            else:
                row += 1
                if done == 0 and row == len(actualField):
                    whenCalc.append(0)
                    done = 1
                if row == len(actualField):
                    col += 1
    return count, aggregate, whenCalc, completeLiners
```

This approach would simply traverse the field by rows and columns. The reason this function is being iterated through rows then column is because it's used to calculate holes. This bot's definition of hole is/are space(s), which isn't/aren't taken by a block, underneath one block or multiple blocks. Thus, this would be easier to calculate by analyzing by columns rather than rows. CompleteLiners, the variable used to calculate how many lines are cleared by a certain move, is also very simple. It calculates how many times there's a block occupying a certain space on the board, not including the unclearable blocks, and add that to its respective row within the completeLiners variable. This allows you to check how many of these indices contain a number equal to the width of the field, which indicates a line being cleared. This allows the bot to calculate features, such as cleared lines, in one iteration through the board. Features that require going through by column then row must be implemented separately and iterate through the field another time.

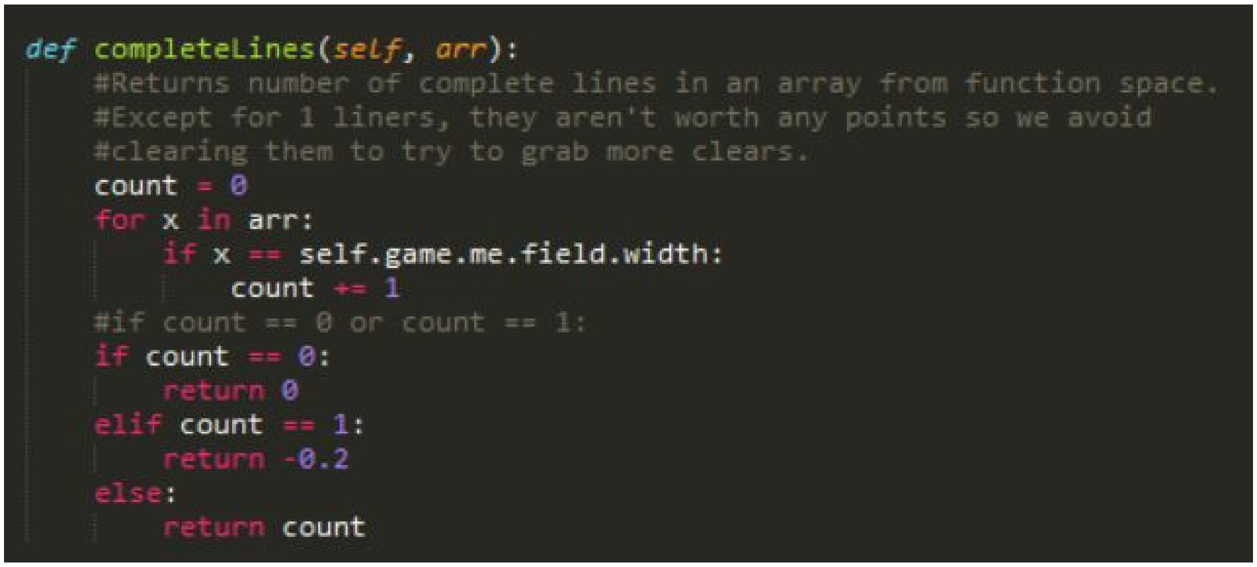
In our approach we also use a heuristic that tries to balance the weights of the features being calculated. This is one of the biggest problems with this approach, figuring out the best weights to judge what moves are the best given the current situation. This heuristic can be seen in this snippet of code:



```python
def fitness(self, field, newPos):
    #Fitness function, which is just our heuristic of our graph search.
    #Originally was going to use GA, so didn't change the name of this function.
    fieldCopy = copy.deepcopy(field)
    #a = -0.510066
    #a = -0.44166
    a = -0.58066
    #a = -0.38066
    #b = 0.760666
    b = 0.760666
    c = -0.41
    #c = -0.35663
    d = -0.2
    #d = -0.18566
    #e = -0.15
    e = -0.4
    f = 0.1
    #f = 0.2
    g = -10
    h = -0.5
    #h = -0.3
    #z = 10
    wall = self.walls(newPos, fieldCopy)
    count, aggregate, heightArr, comp = self.space(fieldCopy)
    clear = self.completeLines(comp)
    bump, bigDiff = self.bumpiness(heightArr)
    tops = self.top(newPos)
    big = self.maximum(heightArr)
    noIBlockSpace = self.badCol(heightArr, fieldCopy)
    #perfect = self.perfectClear(fieldCopy, comp)
    return a*aggregate + b*clear + c*count + d*bump + g*tops + e*big + f*wall + h*noIBlockSpace
```

The self.tops function in this snippet calculates whether or not the moves selected would make the piece lose the game. Obviously, we'd like to avoid this possibility if possible, which is why it has a very negative weight. The self.walls function calculates how many blocks are touching the current piece being operated on. The self.badCol function calculates the number of columns that have zero blocks occupying it. Without this function it'd be more likely, probability wise, that a space that only an I piece rotated to the right or left once could fit in. This piece isn't even guaranteed to spawn during a match, meaning it's a negative feature because without that space a piece could've used it more effectively through the match to clear rows. Lastly, for the more obvious features, self.maximum gets the max of the array of heights, and the bigger this is the less likely a bot would want to put a piece in that column. This is to avoid stacking closer to the end of the game.

Continuing on to the more obvious features, aggregate height, represented by the aggregate variable, should be a fairly negative feature because you don't want a piece to turn itself to create higher heights, for example the I block. Having this as a negative feature also helps clear rows because the pieces will be more likely to turn in a way that would reduce the aggregate height. Clearing lines is one of the biggest features that should be rewarded because it's how you win the game and hurt the enemy by sending garbage lines to them. This should obviously have a very high weight compared to all the other features. The variable count holds the amount of holes in the field given to the self.space function. This is also a negative feature because it reduces the amount of space that can readily be used to move the piece and place it. This, on the other hand, is not as bad as having pieces turning itself unnecessarily to have a higher height. Thus, it is given a lower negative weight than the height features. Bumpiness will calculate the difference in height length for each column close to it, the bigger the difference the harder it is to fit pieces in it. This feature can be good to have if you're going for a risky approach and want to do large clears at once to throw opponents off, but generally it's not advised to perform such a move. Lastly, here's a weight that may catch you off-guard that isn't shown above in the fitness function:



```python
def completeLines(self, arr):
    #Returns number of complete lines in an array from function space.
    #Except for 1 liners, they aren't worth any points so we avoid
    #clearing them to try to grab more clears.
    count = 0
    for x in arr:
        if x == self.game.me.field.width:
            count += 1
    #if count == 0 or count == 1:
    if count == 0:
        return 0
    elif count == 1:
        return -0.2
    else:
        return count
```

This function calculates the number of complete lines from the array calculated in the function self.space. The weight that might catch you off-guard is that one liners are penalized. We don't want the bot to waste lines that can be easily cleared by a one liner, it'd be better to save that line for a multiple line clear. This concept is similar to when you're investing in stock, except this function invests that the future will be bright, and hopefully giving pieces that will clear multiple lines for the space created from the investment. Otherwise, the bot will go bankrupt, if we're sticking with the analogy using stocks, and lose this game. Given the weight of such a feature, the bot must make a decision of whether this investment is worth it.

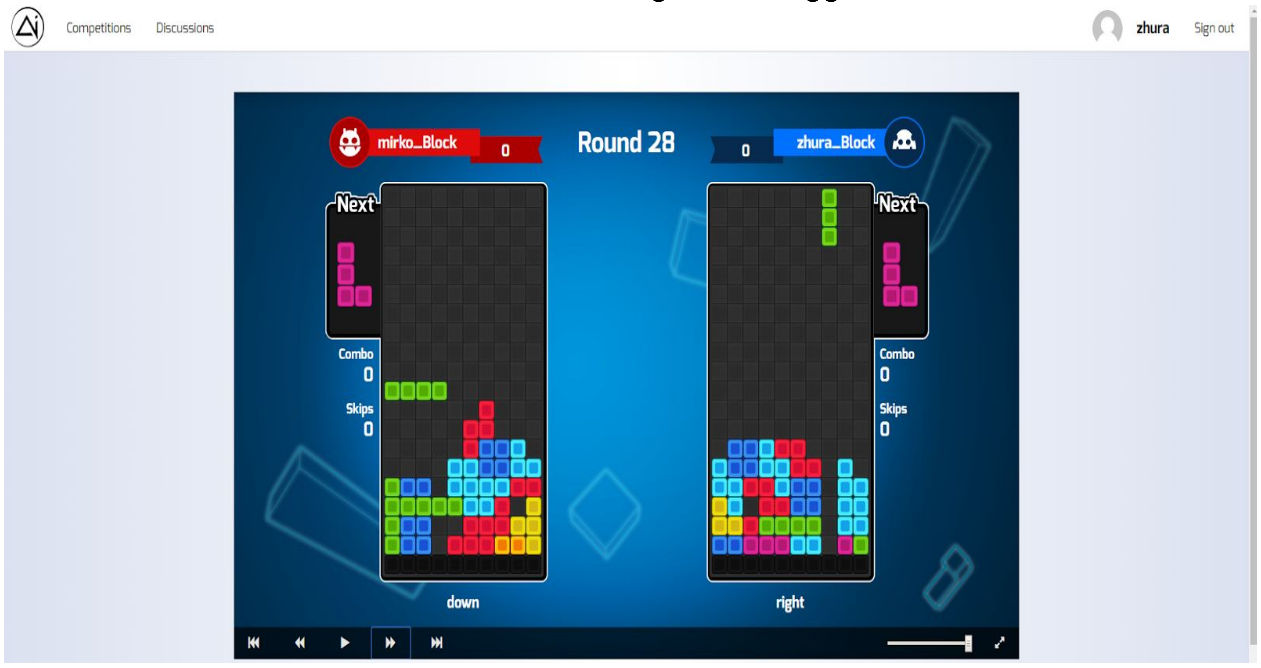
Now with a basic background of the program we can discuss about the actual approach. The approach was to simulate almost all possible moves with the current piece for the given board. The reason it's almost all possible moves, is because it doesn't approach the problem like the top bots on the website that would move the piece down and then select left or right per row for all columns. Thus, it won't be able to analyze small holes under some blocks. This approach would instead analyze the field from the top of the field and move left and right to figure out the best place to put it. Before it moves left or right the piece would also turn to the right once. This would allow for all the turns the piece can do to be analyzed for all the spaces on the field. This would only require turning the piece to the right, and does not need to turn the piece to the left because all the possible turns can be done in 4 turns. Using this approach it wouldn't be random like the genetic algorithm approach, and it would have a clear stopping criteria. It would also make good moves from the beginning compared to reinforcement learning, without the need for all the memory and time that a state space would use. It would also provide a clear reward compared to reinforcement learning, as seen in the fitness function above. There would also be no need to parse moves like the original genetic algorithms implementation would from a string of moves to offsets that would be used to simulate moving a piece from its current position to another position on the field.

This approach is simple and effective compared to the approaches explored above. It would also work well against other bots because it can judge from the start where to put pieces and the optimal position for that piece. It would also solve the problem of time complexity because it shortens the need simulate as many moves as a genetic algorithm would, and it wouldn't have to build a state list that a reinforcement learning algorithm would. Not only that, but it can be designed to be aggressive or passive in its approach when finding the best moves to send to the server. Lastly, it's very easy to debug compared to other approaches because a bug can be found again and again for a problem versus an approach that was more random like the genetic algorithm approach.

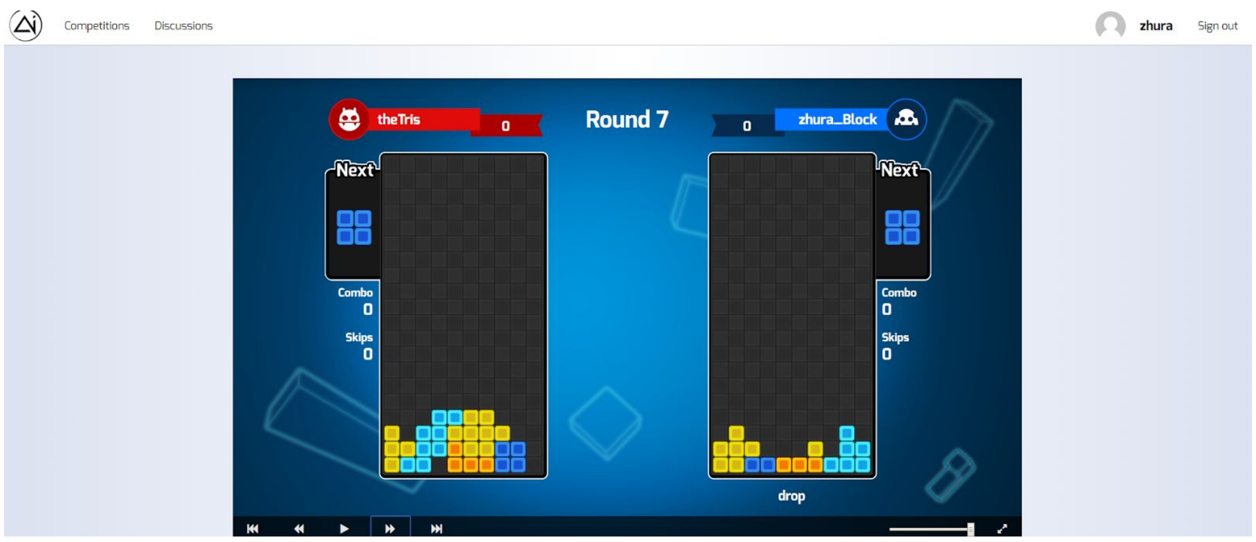
Results

In the end, the simple graph search with a heuristic worked the best. One reason it was the better approach was that it was less random than a genetic algorithm approach. It also had a clear stopping criteria that couldn't be found in a genetic algorithm because a perfect fitness is variable, and even if calculated may not always be accurate. This raises validity issues in terms of the algorithm, and thus the simple graph search algorithm would be better to avoid such issues. The reinforcement learning approach would need a state list that was enormous, causing a huge memory usage as well as time complexity when selecting the best move based off the state and when calculating all the possible states. The simple graph search algorithm can also be tuned to be more aggressive or passive easily, without creating or changing a whole function within the code. This would reduce the time needed to code the solution. In both the approaches it would be hard to test some cases. Reinforcement learning would require to test all possible states and testing that the actions are done correctly. The genetic algorithm approach would be too random to figure out what the bug is and how it occurred. The genetic algorithm approach can also be unsteady in terms of time complexity because of the variable length of the set of moves, which all have to be simulated and checked that it fit that new position. Each set would also require an evaluation to check how good the set of moves was for the current field. This would require reducing the time complexity of many functions, and could result in complex code that would be harder to debug than a simple function with a larger time complexity. The simple graph search, that can either search by row then column or column then row, reduces this time complexity by focussing on the non-duplicate and valid moves before passing them into more time consuming functions. The strategy that only based its moves on height was extended upon, and was obviously worse than the current bot's strategy. The current bot now has more features than just height; it has holes, bumpiness, lines cleared, etc.. Lastly, in terms of passive versus aggressive both have their appeals, but I believe a more aggressive bot would be better. An ideal passive bot would try to keep the number of lines on the board to a minimum, but will clear any line possible, obviously it will still know clearing more lines is better, and wouldn't get much of a chance to send garbage lines to the opponent.

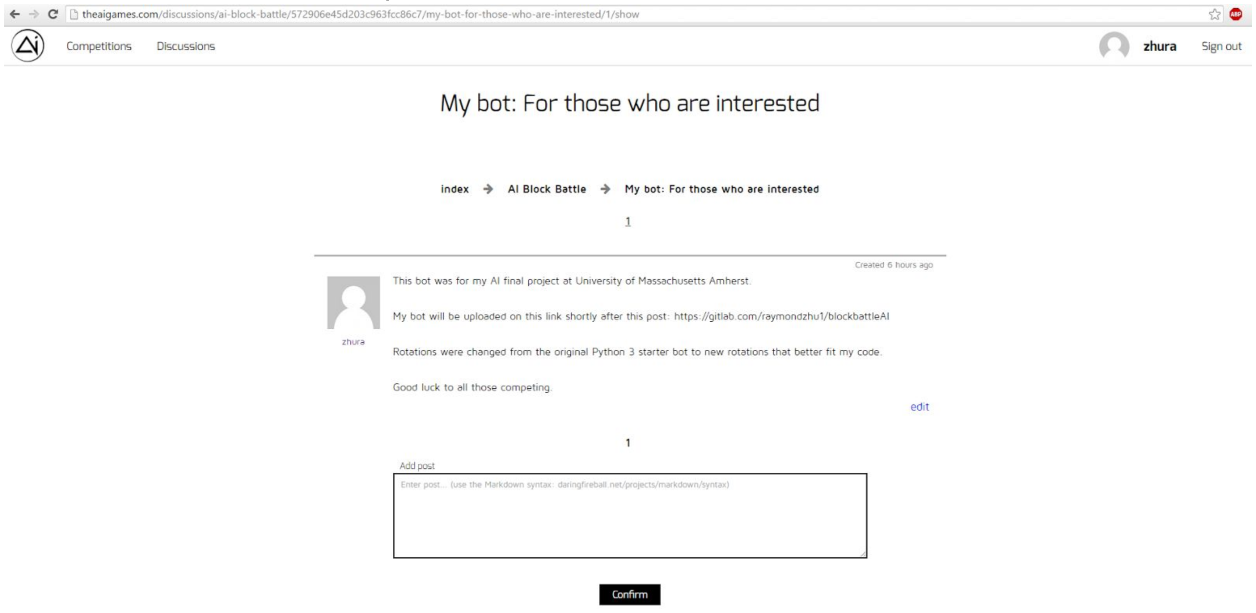
The aggressive both can be more risky and attempt to stack pieces, while leaving a space that would allow for one piece to clear multiple lines. This would then send garbage lines to the opponent, and the more rows cleared the more garbage lines. As it's easier and wouldn't rely on chance, an aggressive both would be better than a ideal passive bot. An ideal passive bot would require pieces to always allow for a clear to happen in a small amount of rounds, while not increasing their aggregate height by too much for a clear. This ideal passive bot would basically win by allowing the opponent to kill themselves from trying to setup a multiple line clear. An aggressive bot is more reliable and easier to find good weights for their respective features when calculating the board's fitness. Thus, an aggressive bot is used, but in the past mine was too aggressive and the level of aggressiveness had to be lowered. A bot that has a balance of these characteristics would help lower the risk of the aggressiveness, but yet allow for more garbage lines to be sent to the opponent.
This screenshot shows the bot in action during a small aggressive move:



You can see my bot, zhura_Block, has left enough space for an I block, turned left or right, to fit into in order to clear three lines. While, a more passive approach can be seen here:



Instead of stacking the last piece that was placed, the bot decided it was better to clear the row of all its blocks.
This is a screenshot of the post on the discussion forum:


theaigames.com/discussions/ai-block-battle/572906e45d203c963fcc86c7/my-bot-for-those-who-are-interested/1/show
Competitions
Discussions
zhura
Sign out

My bot: For those who are interested

index → AI Block Battle → My bot: For those who are interested

1

Created 6 hours ago

zhura

This bot was for my AI final project at University of Massachusetts Amherst.

My bot will be uploaded on this link shortly after this post: https://gitlab.com/raymondzhu1/blockbattleAI

Rotations were changed from the original Python 3 starter bot to new rotations that better fit my code.

Good luck to all those competing.

edit
1
Add post
Enter post... (use the Markdown syntax: daringfireball.net/projects/markdown/syntax)
Confirm

This is the link to the gitlab code repository:
https://gitlab.com/raymondzhu1/blockbattleAI

Discussion and Conclusions

In terms of time complexity and performance, as explained in the results section, the simple graph search with a hybrid between aggressive and passive seems to be performing the best with a smaller time complexity than the other approaches. This result in time complexity and performance was expected against the genetic algorithm approach because the genetic algorithm approach didn't give a clear stopping criteria and became too random without enough iterations. The perfect fitness displayed above was more of a placeholder value, but tried to allude to the weights of the features to be mostly negative and a zero fitness score would be good. When playing against bots, it was also easy to tell more passive bots versus slightly aggressive ones would lose, and that overly aggressive bots could be defeated by a slightly passive bots. This is explained in the results section as well, and was clear from the beginning. In terms of future possibilities for the bot there's multiple things that can be done, as stated before. The bot could drop down as low as possible and check for possible moves per row to be made at that position, such as move left or right in order to fit a previously inaccessible hole. This would help clear multiple lines, and be both more aggressive and passive. This feature is seen in most of the top ranked bots, while lower ranked bots do not do this. Another improvement to add in is more look ahead, currently the bot will only decide for the current situation what the best move is, rather than the best move to make given a good move it can make the next round given the next piece. This would help shorten block towers, and create more chances for clearing multiple lines rather than single line clears. An idea for a new mechanic in the game could be sending the row with the most holes on your field to your opponent after you clear more than one row. This would make garbage lines much harder to deal with, currently garbage lines are easy to deal with because it's usually only missing one block to complete it.

Some side notes to the actual development of the bot, is that the original rotations for turning a piece is incorrect in the Python 3 starter bot, and may be in the other Python starter bot as well. These rotations sometimes represented a piece of the opposite type, such as a Z piece being represented as an S piece. According to the discussion forums the offset one of the user gave was based off of the left-most position of a block within a piece, rather than representing the offset for each block within a piece. I think having an offset for each respective block for a piece would make more sense rather than basing it off the left-most block within a piece. Even if the calculations were for each block respectively of a piece the rotations given by the starter bot would be even worse. This was hard to debug, given that the rotations were given to us and there isn't much information on it. As said before, this may just be the way my bot does rotations.