Alexander Sullivan – 26776710

–**Note**:I am continuing to work on this for 10 days and the implementation described here may change even by the beginning of this weekend (Friday evening).

Introduction: I am solving **Block Battle**. The hardest problem was considering the combination bonus. The double t-spin should be properly weighted highest but it also takes planning and accurate placement to even prepare for a *double t-spin* (example on website). The top of the leader board also has a strategy of attacking the opponent when they stack one large side of the board in preparation for a large combination. Thus the goal of the robot is to properly weight the value of a strong combination (through large stacks) against the risk of the opponent getting a combo and *pushing a stack above the* '*limit*' *of the board* (by sending the 'garbage lines' to the opponent).

The bot I designed considers the entire board, but the features largely calculate the state of the board below the **max height** (by dropping each shape to just above max height). I began the implementation in **python** but transitioned to **java**. I transitioned because from now to May 13th I plan on continuing to optimize for the May 15th deadline. I explain my continued *road map* in this report. While python was simple to work with, as I continue to add calculations during my turn time, I am worried python would be too slow (especially when implementing 2+ lookahead after considering the next piece). This fear arose from experience as well as noticing only one top 20 bot uses python. I want to note that I am continuing to work on this bot for the next week as the **methods** section explains where I am going next. I focused heavily on extracting features and an optimal way to represent the state of the board thus I have excellent performance (up to 141 block clears without going for bonus) but have not finished looking into the future, taking advantage of the opponents board, or utilizing the combination system (which is in my roadmap).

Now that I have implemented features and know which ones to continue with, I believe I can build, in the next 10 days an extremely competitive bot (top 50 is the goal). I have a bot that understands the 'tactics' important for end-game play where combinations are hard to come by. The road map and last method explained talks about implementing the 'strategy'.

Methods

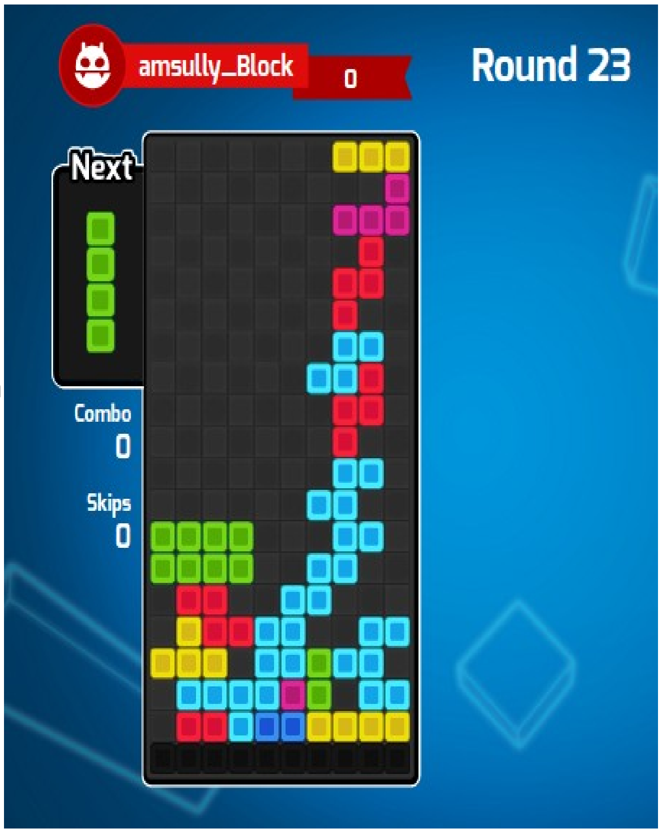
Python: The python implementation was discarded after getting a feel for the game, but should be noted. The bot iteratively ran through the board bottom up and looked to place the block in the first valid location (where it 'fits'). It would perform this for each orientation and choose the best fit by the minimum bottom-left x,y coordinate. This method performed at a low level because there was no check for if the block could actually get to this location (based on a primitive rotation/drop method). If a space of a Z block (or any block) would arise in the game, this would consistently be looked at as the best location and the piece would not be able to get there (because a previous block was placed blocking it…a photo is shown below). This phase was very important as it helped me understand board representation and small quirks to the game (the field is represented sideways, using the native methods was very valuable). Although this method was extremely fast, it was difficult to incorporate new features such as preparing double t-spins and such because there was no path being generated. I knew, when preparing to code the next phase, I needed a robust path search starting at the initial block. This was one of the reasons I switched to java as it would be faster to traverse these paths.

Here is bestFit for iterating from the bottom of the board up (continues on next page).

```
def bestFit(self):
    for y in range(19,-1,-1):
      for x in range(10):
        orientation_coordinates =self.fitWithOffset_anyOrientation(y, x, piece, 
field)

          if(orientation_coordinates != None):
            return orientation_coordinates
```

Here is an example of the python bot not properly positioning the blocks and believing it can fit in an impossible location:



I also developed a few **tools** during this phase. The first was a simple *bots.txt print-out* that would output the engines responses after processing each round. This document wasn't extremely valuable so I developed a **prettyprint** python script that would take the resulting bots.txt and create a nice, properly orientated, *game_print.txt* file which would have the two bots boards side by side (in a ascii printout manner.). This allowed for an intuition of what the bot was doing.

Java – Beginning

Switching to **java**, I began the path traversing algorithm. Initially I built a recursive solution that, at each state, performed every possible movement. This, obviously would result in a “Left” - “Right” infinite loop so constraints were added to not allow the bot to move left after a right (and vice versa). Another constraint on the problem I decided on was to only perform the rotation at the beginning. This was because the search space ballooned and it was difficult to control with the recursive approach (the next method addresses this issue). This lowered the number of states that were required to search but became unwieldy. The result was a simple method that would traverse down most possible paths, and our base case was created in the *Field.evaluateScore* function. The first task of *evaluateScore* was to generate a temporary array of where the shape was located, but if this location was where the end of the board was, or where another block was, it would return the -*(Double.MAX_VALUE)* which essentially means that this was an impossible state and was our **base case**. While this implementation allowed me to test many features, I found that I need to evolve to an iterative solution to optimally handle some of the edge cases (ie double t-spin) without searching thousands of extra areas.

Psuedo-code of the main searchSpace method. Note rotations checked in main function once. Note the number of parameters required to 'tame' the features of this recursive solution.

```
    private Result searchSpace(Shape currShape, Field field, ArrayList<MoveType>
moves, String prev, int turn) {

         double locationScore = field.evaluateScore(currShape);
         Result spaceResult = new Result(moves, locationScore);
          if (locationScore == -Double.MAX_VALUE) {
              return spaceResult;
         }

         // Copy field to new field, set curr piece. Search space for next piece.
         Field nextField = field.getNewField(currShape);

              spaceResult = checkRight(spaceResult, (currShape), field, new
ArrayList<MoveType>(moves));

              spaceResult = checkLeft(spaceResult, (currShape), field, new
ArrayList<MoveType>(moves));

              spaceResult = checkDown(spaceResult, (currShape), field, new
ArrayList<MoveType>(moves));

         return spaceResult;
    }
```

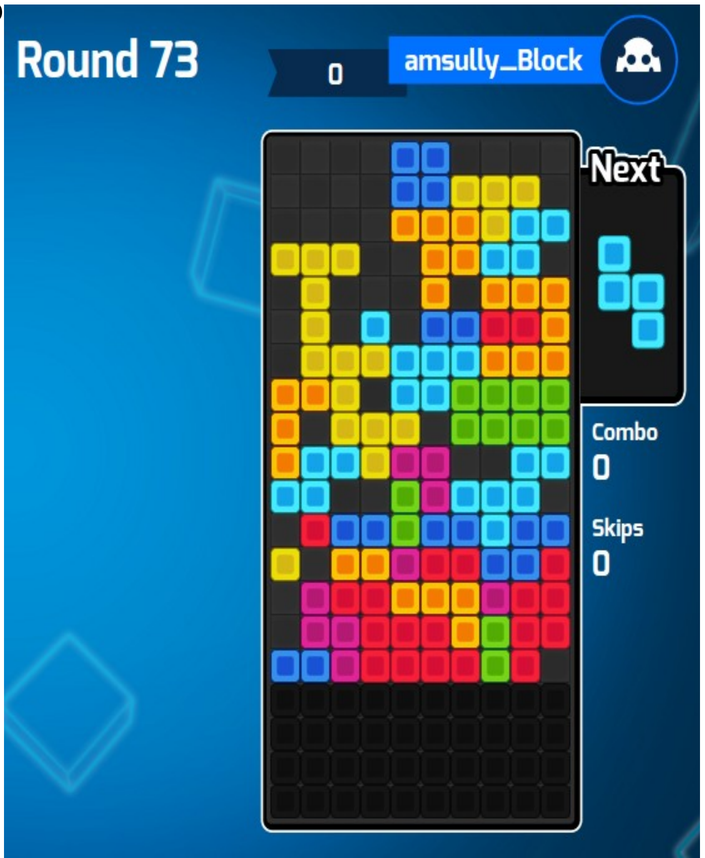
Now I began the most time-consuming task of deciding on features that I could utilize. The first I implemented was simply minimizing **max-height:**

```
score = (max_height * -1);
```

This first feature was an interesting test to running and competing against people but it would perform poorly as instead of clearing rows, it would minimize the height only. The next feature implemented was '**resulting rows**' which would take the max-height and subtract the number of completed rows. This was valuable as it now encouraged the bot to complete rows. At the same time we including a **hole count**. This counted the number of 'inaccessible' empty cells resulting from placing down a block.

```
    double resultingHeight = max_height -
         complete_rows;
    double score = (resultingHeight * -10) +
         (hole_count * hole_parameter);
```

The parameters here were calculated through a simple parameter sweep. I incorporated a *hole_parameter* which was:



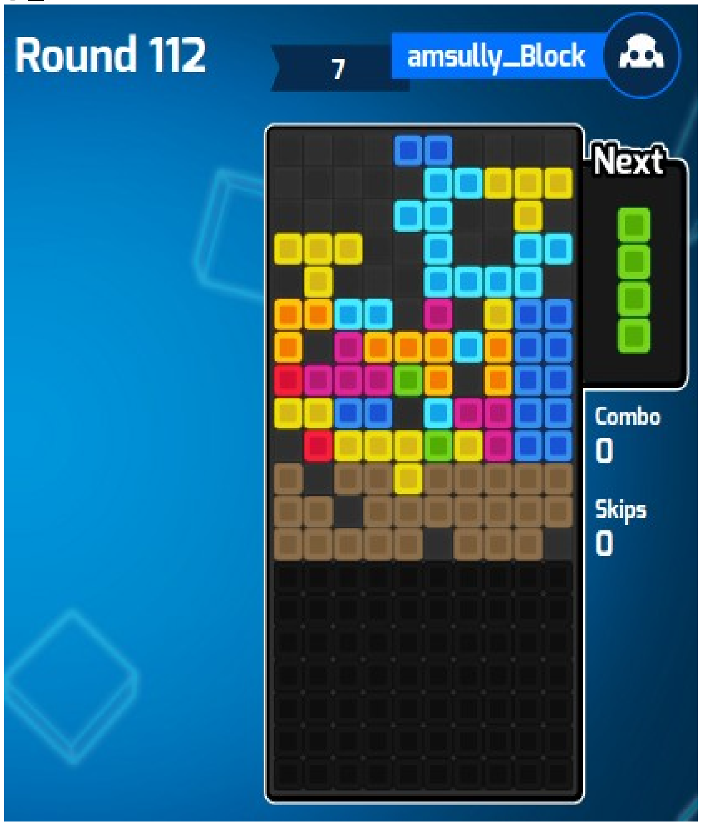
```
	double hole_parameter = -20/Math.pow(max_height,2);
```

This parameter was used to discount the result of the number of holes as the height of the blocks went up. These additional features ended up drastically improving the initial min-height metric because we could now clear the values. A photo above shows us getting to the 73rd round.

I was not yet settled with these base features as for a bot to compete at the top level, it must understand a set of core features when there is no opportunity for a combination. This led me to think about a final feature that satisfied my desire to clear blocks extremely efficiently. My decision was **connected blocks**. Connected blocks weights the number of non-empty squares that are adjacent to the potential position of the shape. This provides preferences during states of the board that appear 'boring' where there are many potential locations that would not change the height or change the number of holes. I also broke out the number of **complete rows** from *resulting height.* This added more emphasis on actually clearing lines. This intuition led to another significant improvement and we can now compete above average (I approximate the rating is 1420 after being down-ranked from the other primitive bots) with zero lookahead!

```
          double score =   (resultingHeight    *    -10)
                    + (hole_count       *    hole_parameter)
                    + (complete_rows     *     2)
                    + (number_connected   *     5)
                     + (average_height    *    -.5);
```

This is a game where we achieved 111 cleared rounds! (note this match was against the classes number one rated **adakasky_Block**) This bot, against itself, can clear up to 141 rounds. We should note that we are solely basing our bot on features without lookahead. The weights were tuned with parameter sweeping (note that during this phase of building the bot, I was looking for the best features... I describe in the road map my plan to *polish* my features with a genetic algorithm).



While this method appears great and now we can start implementing combination features, we have to look back at how we initially found every state. We used recursion and the problem with recursion is that if we want to develop a feature path such as spinning a t-block into a row to get a large combination, we need to add exceptions to our recursive function through parameters to our method, and this results in sloppy code, complicated cases, and an inability to understand what is actually happening. If I want to implement the best bot I need to tell a bot to search for valid t-spins while not searching thousands of unnecessary spaces.

Which leads me to where I am today.

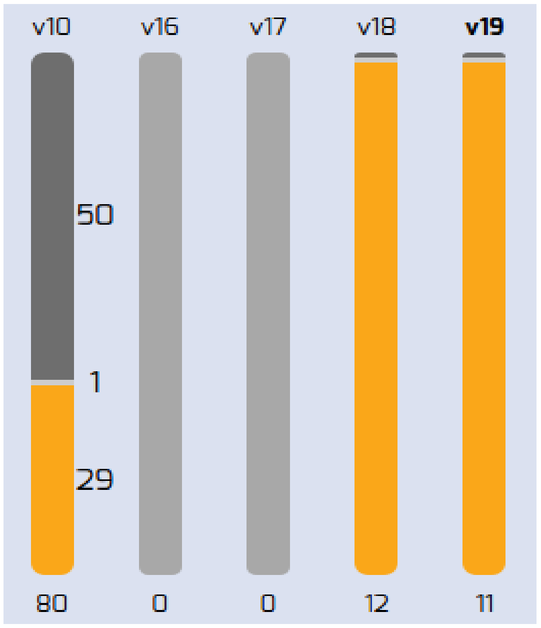
Java – Road Map - Where I am and next 10 days.

I am now implementing an iterative solution for these next 10 days that will give me full reign over what my bot decides to do/search. My iterative approach starts in the top left and iterates through the potential right moves and drops the block. When the block hits the bottom, rotations are applied, checks for if it is a t-spin are added, and checks for lateral movement are added (for 'scooting a piece under another). This process will give me the same search space as the recursive formulation as well as give me the ability to search for those final rotations and final movements. The complexity was the limiting factor for the recursive function and this iterative function allows for simplicity and easier forward planning (because in a recursive function we will be continuously calling 'searchSpace()' already, so we would need to add extra parameters to ensure the recurision realizes its a new forward plan). This will allow me to create every feature in a simple fashion and finally optimize the final parameters with a genetic algorithm.

- *First stage:* re-implement the features that were extremely strong in the previous recursive phases. This will be the 'core' of the bot.
- *Second Stage:* Lookahead that includes the current block, next block, and a future lookahead that consider a t-block and line piece (for combos).
- *Third stage:* Focus on **double t-spin** features. Currently, to get into the top 20, the t-spin strategy is required. The 24th ranked player (Sunblock) does not currently go for t-spins. Implementing this will allow my bot to rise significantly. The extra features that will be required for t-spin include a 'build up feature' that, a 't' feature that determines if a t-block can be rotated in, and a 'combo'
- *Fourth stage:* Optimize parameters. This will be progressively done throughout the coding but at the very end, a genetic algorithm will be used to modify the parameter values in a way that estimates the optimal value of all the features.

Results:

This image shows the performance of the 3 iterations of the bot (note that it does not include my iterative solution as it is under development).



V10 is the Python bot and it ranked me down because it went **29-1-50**. I should have deactivated it!

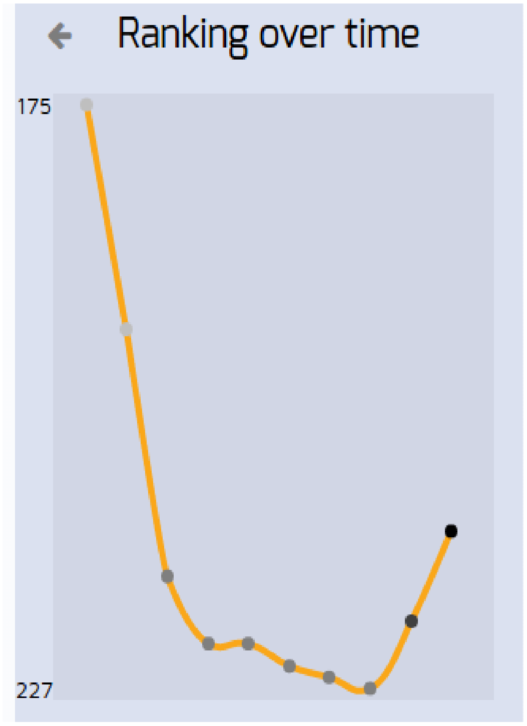
V18 is the Java bot that uses minimum height and holes. Before I changed this version it was **12-0** at the lower rating that the python bot dropped it to.

V19 is the Java bot that includes all features discussed in the 2nd method above. As of Wednesday night it is **11-0**.

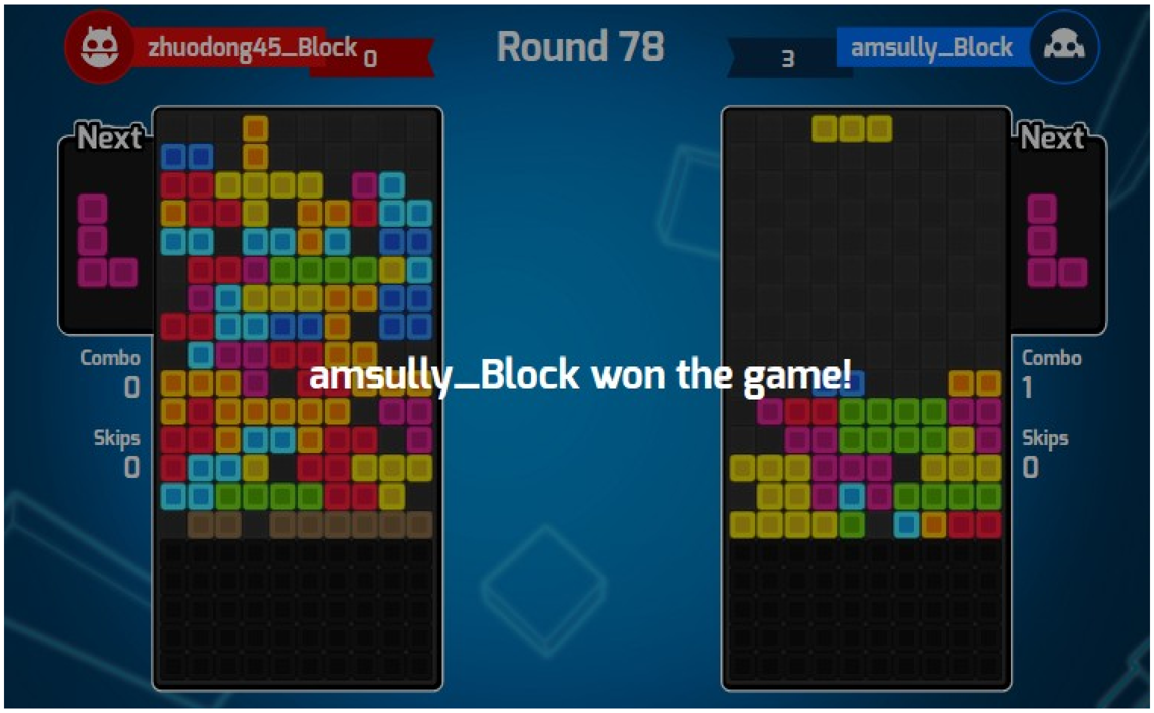
Overall my record is 52-1-50 and my level should rebound by Saturday.

Alexander Sullivan – 26776710

The performance picture shows my ranking over time. We can notice immediately where I implemented the strong features as my rating is currently re-leveling. I estimate that the rating of my current bot will level out around 1420 or 180th place (as of this writing is has risen to **1317**). The best bot was the last one I properly implemented and I can gauge this based on the quality of bots that I have been beating (upwards to a 1430 rating).



Here is a full screenshot of my most recent bot winning against a member in our class **zhuodong45_block (v6)** who is rated 1490 (ranked 154th) currently (he added version 7 recently). He is 3rd in the class. This match went 78 rounds. Note that these 'challenges' are unrated thus I cannot rank up beating these stronger opponents until I play enough 'ranked' matches. Thus my rating may be deflated when graded.



Alexander Sullivan – 26776710

3. Screenshot of your post in the discussion forum posting your gitlab repository (or wait until after really competing in blockbattle).
I wanted to leave a note here that I have a lot of time these next 10 days and will be waiting to link to my gitlab account on the discussion forum.
WHEN the competition begins, I will email the professor/TA account with a small update of my progress and how I faired as well as posting on the discussion forum.

Link to my Gitlab repository:
First note: You can view my previous bots by looking at the .tar.gz files. The current code is representative of my iterative approach that I plan on competing with. Thanks!
https://gitlab.com/amsully/block-battle-amsully

Discussion / Conclusion

Well I feel like I have just scratched the surface of what my bot is going to be able to do but what I have experienced is the effectiveness of even sall parameter sweeps to determine parameter values that will represent features well. I have thoroughly discussed my next 10 days and I do plan on being competitive.

Implementing the initial bots (python and java) were extremely important in inspiring the final iterative bot. I feel these methods allowed me to visualize what the top-rated opponents are doing so I can mimic the supposed features they are optimizing.

Although not specified, the objects of how a board is also very important. While I am using the native representation of the boards, I scoured the discussion page of strategies section and it seems very common to implement the board representation with an array of bits. I am considering interfacing this alternative bit-board with my resulting bot towards the latter part of next week. This would potentially speed up my bot and this can be important to handle 'gravity' which is applied as it makes moves.

Another note is the time handling. I currently do not handle time and might not come to it, but it would be interesting to optimize time usage by searching more of the search space in the middle of the game, or to consider the opponents board if he plays optimally. Hopefully the code for these high level competitors post there code publicly, if I am not able to crack top 24, after the competition has finished. This would be extremely valuable for future competitions on this website as all of these games have time limits.

For me, the part of the project I am reporting on is the feature definition phase which, as discussed in lecture, can be the most difficult part of the solution. I did not have the heart to optimize (with genetic algorithms or a iterative method) features that I know were not going to win the entire competition. By spending this extra time on the features, I am able to beat fellow classmates who perhaps were not so keen on the feature phase as well as define some goals that I believe will be successful against higher rated bots.

Overall, I am happy to have created an extremely competent bot for the due date but I am looking forward to creating my final bot that will have implemented all important features.

As mentioned above I will send an update to the class email address with my forum post and the results of my new bot.