CS 4701: Survival of the Tetris

Rohit Biswas (rb625), Marvin Liu (tl474)

Introduction

Tetris

The game Tetris is a puzzle video game where the player of the game is rewarded points for staying alive. This project attempts to use a genetic algorithm to create an Artificial Intelligence (AI) to solve this game.

Tetris has a player juggle randomly generated tetrominoes (shapes generated by adjoining four squares in various manners). All of the possible tetrominoes are shown below in Figure 1.

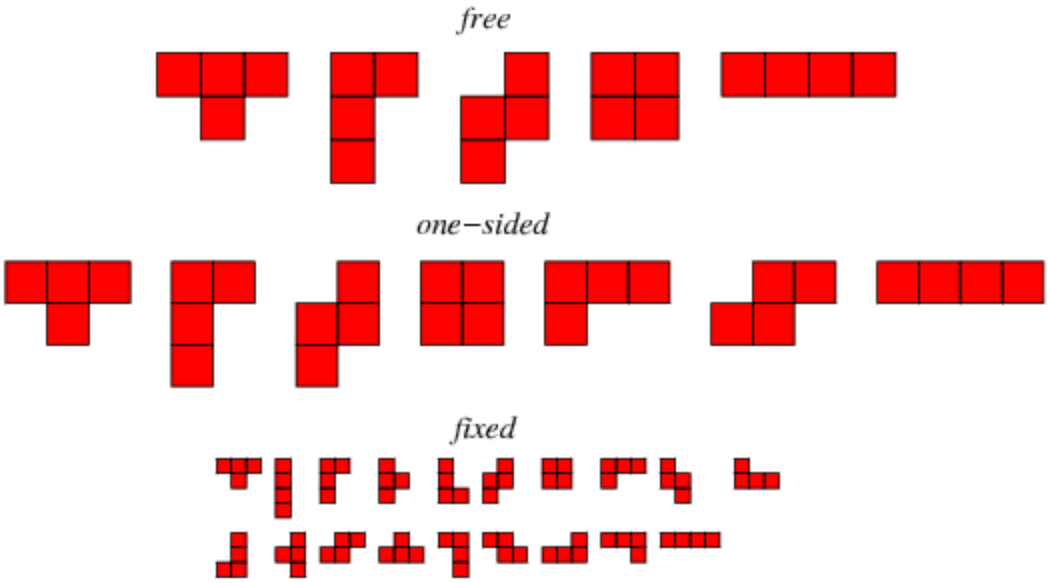


Figure 1: All Tetrominoes. In order from left to right in the one-sided row, they are the: "*T,*" "*J,*" "*Z,*" "*O,*" "*L,*" "*S,*" and "*I*" Tetrominoes

As these tetrominoes fall to the bottom of the screen, the player has to organize them in such a way so that the highest stack of the blocks does not reach the top of the screen and block the next incoming block. The player can clear blocks by filling a whole row with blocks. Further, the player gets a score based on how many lines the player has cleared. A player's final score is determined when the losing condition is reached, so the goal is to maximize the score while doing the best to not reach the losing condition.

Genetic Algorithms

A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state[1]. Genetic Algorithms were applied here to generate “players” with different weights described below. The process of training such a genetic algorithm involves using concepts from evolution and natural selection to create an optimal player from a larger player population. The evolution of the players comes from repeated rounds of running the player to see how it fares, selecting the best players from a pool, and then making slight mutations/crossovers between the “surviving” players. Thus, the best performing player will be outputted from this process, as described in further detail below.

Framework

Tetris Game

For the GUI of the game, we first started out with an open source Python clone of Tetris from Al Sweigart (http://inventwithpython.com/tetromino.py). Then, we stripped out all the keyboard input code as well as the fall down timer code, since these would no longer be necessary once we implement an AI to place the tetrominoes automatically.

Key Functions

First we implemented the function evaluateBoard and several associated helpers, which given a board configuration and a weight vector, returns a score of how good the move is using the heuristics that we will define in a later section.

Next, we implemented the function getAllMoves, which given a board and the current falling piece, generates possible board configurations which include the falling piece. We generated the possible board configurations by placing all possible rotations of the falling piece at each column going from left to right, and then dropping it down until it falls into place. This actually misses the cases of when there is an overhanging tile and it is possible to move the piece side to side after it has dropped for a while.

---

[1] Russell, Stuart J., and Peter Norvig. Artificial Intelligence: A Modern Approach. Englewood Cliffs, NJ: Prentice Hall, 2010. Print.

Self Playing

Once we implemented the above functions, we modified the run loop of the game to play by itself. For each falling piece, we first use *getAllMoves* to get all the possible boards that include the falling piece with the current board. Then, we run the function *evaluateBoard* on the list of possible boards and find the one with the highest score. This gives us the board resulting from the move chosen by our AI. Then, we clear any complete lines and update the score accordingly and replace the current board with this new board. Thus, in each run loop, the AI is placing a single piece onto the board.

Genetic Algorithm Trainer

To implement the trainer of our Tetris game, we made a copy of our modified game described in the previous section and further extended it. First, we removed all of the GUI to speed up the performance. Then, we used the DEAP evolutionary algorithm framework[2] to train our genetic algorithm. To do this, we first generated a list of randomly generated unit weight vectors. Then, we used a for loop to iterate through the desired number of generations. Within each generation, we evaluated the fitness of each member of the current population, implemented tournament selection to select the parents of the next generation, implemented crossover and mutation to create offspring, and created the next generation through selection of the current generation plus the offspring. These steps will be expanded upon in the Genetic Algorithm section of this report.

Algorithm

Heuristics

We used several heuristics to evaluate the strength of each potential move. These heuristics include the total height of all the columns on the board, the number of complete lines on the board, the number of holes (situations where there's an empty space with a tile above it in the same column) on the board, and the variance in the heights of the columns.

---

[2]Fortin, Félix-Antoine, Francois-Michel De Rainville, and Marc-André Gardner. "DEAP: Evolutionary Algorithms Made Easy." *Journal of Machine Learning Research* 13 (2012): n. pag. July 2012. Web. Nov.-Dec. 2016.

Total Height

The total height is the sum of the height of each of the columns, including any holes (blank spaces which have pieces above it in the same column). The reason that we chose this heuristic is that we wanted to minimize the total height of all the columns, since intuitively, the higher our tetris board, the closer we are to losing.

Lines Completed

The number of lines completed is the number of completely full rows that would be created if we were to put the current piece at the location that we are evaluating. Intuitively, we want to maximize the number of complete lines, since clearing lines increases our score and gives us more space to continue playing.

Holes

A hole is defined as a blank space on the board where there is a tile in the same column above it. We want to minimize the number of holes on the board, since having a hole requires the player to clear all the lines above the hole first before being able to fill the hole, which can be very difficult.

Height Variance

We calculate the variance in the heights of each column by just summing the difference in heights of each adjacent column. This is expressed through the following formula:

$$Height\ Variance = |Height(Col\ 1) - Height(Col\ 2)| + ... + |Height(Col\ 9) - Height(Col\ 10)|$$

We want to minimize the variance in heights of the columns, because intuitively, having columns with very uneven heights make it harder to clear a line.

Genetic Algorithm Training

Genetic Representation of Solution

For each tetrominoes, we evaluate the move by computing a linear combination of our 4 heuristics and choosing the move that results in the highest score. The formula for calculating the score of a move is as follows:

$$Score = \alpha \times Total\ Height + \beta \times Lines\ Completed + \gamma \times Holes + \delta \times Height\ Variance$$

We are looking for a vector $w = <\alpha, \beta, \gamma, \delta>$ which performs the best. Thus, this weight vector is the genetic representation of the solution, and we will use genetic programming to look for the optimal weight vector which results in the highest scores.

Fitness Function

To calculate the fitness of an individual weight vector, we play 10 games using that weight vector and take the total number of lines cleared over the 10 games as the fitness score. If we define $runGame(w)$ as playing a single game of Tetris using the weight vector w in our board evaluation, we have the following definition of fitness:

$$fitness(w) = \sum_{n=1}^{10} runGame(w)$$

We choose to aggregate the number of lines cleared over 10 games in order to reduce the variance in scores, since the actual number of lines cleared can vary from game to game due to the randomness in the order that the pieces appear in. Initially, we also limited each game to only use a maximum of 400 tetrominoes to prevent the games from running for an extremely long time. Eventually we increased this number to 5000 in later generations as it became difficult to differentiate the fitness of the population based on only 400 tetrominoes.

Initialization of Population

We seeded the initial population with randomly generated unit weight vectors. Initially, we chose a population size of 100 such that there would be sufficient coverage of the solution space, but also not so many such that the training process would take an inordinate amount of time. This is a value that we would like to increase, as discussed in the Future Improvements section.

Selection

During each generation, we calculated the fitness score for each weight vector in our population using the fitness function defined above. Once we had the fitness for each member of the population, we used the tournament selection method to select members for later breeding. Specifically, we repeatedly took a random sample of 10% of the members of the population and chose the 2 fittest members (“running a tournament”) from this random sample to breed.

Crossover, Mutation, and Reproduction

Once we had enough parents to breed members of the next generation, we performed the crossover, mutation, and reproduction step. For each vector, we gave it a 50% chance to crossover with another vector, 20% chance to mutate, and 30% chance to reproduce to the next generation.

For crossing over, we used one point crossover. Specifically, we crossed two vectors over at a single, randomly selected, point to create two new children vectors. For example, if our weight vectors were $w_1 = \ < \alpha_1, \beta_1, \gamma_1, \delta_1 >$ and $w_2 = \ < \alpha_2, \beta_2, \gamma_2, \delta_2 >$ and our randomly selected crossover point was between the second and third elements in the vectors, then we would get children vectors $w_1' = \ < \alpha_1, \beta_1, \gamma_2, \delta_2 >$ and $w_2' = \ < \alpha_2, \beta_2, \gamma_1, \delta_1 >$.

If a vector was chosen to be mutated, then we gave each value in the vector independently a 20% chance to mutate. Each mutation is selected from a Gaussian distribution with a mean of 0 and a standard deviation of 0.5.

If a vector was chosen to reproduce, it was simply added into the offspring population. As stated in the previous section, we repeated this process of crossover and mutation until we created a new generation that was the same size of our current generation.

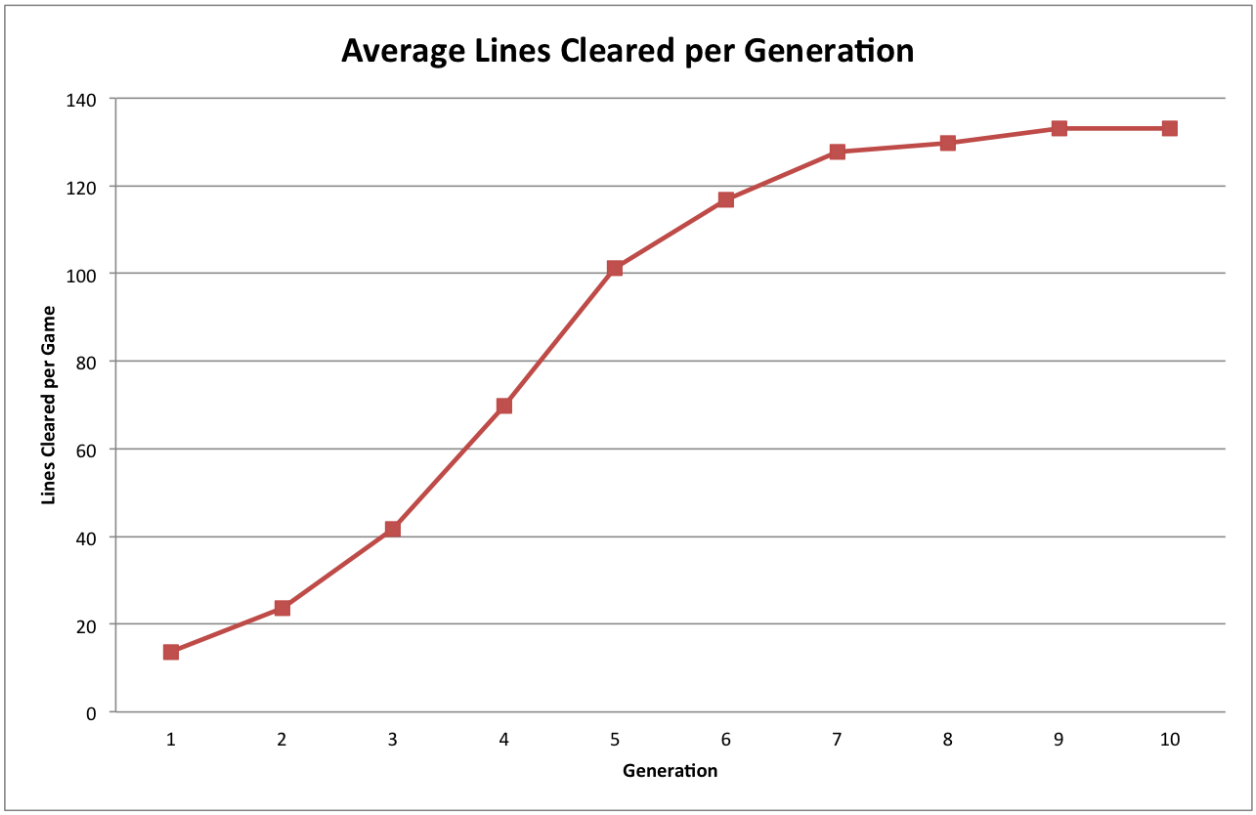
Selection of the New Population

In each generation, we aggregated together the current generation and the produced offspring (so 200 members in total). Then, we used tournament selection again to select 100 of these 200 members to survive onto the next generation. In doing so, we ensured that the least fit members of our population typically did not survive to the next generation to breed, thus improving the overall fitness of our population.

Results

Initial Attempt

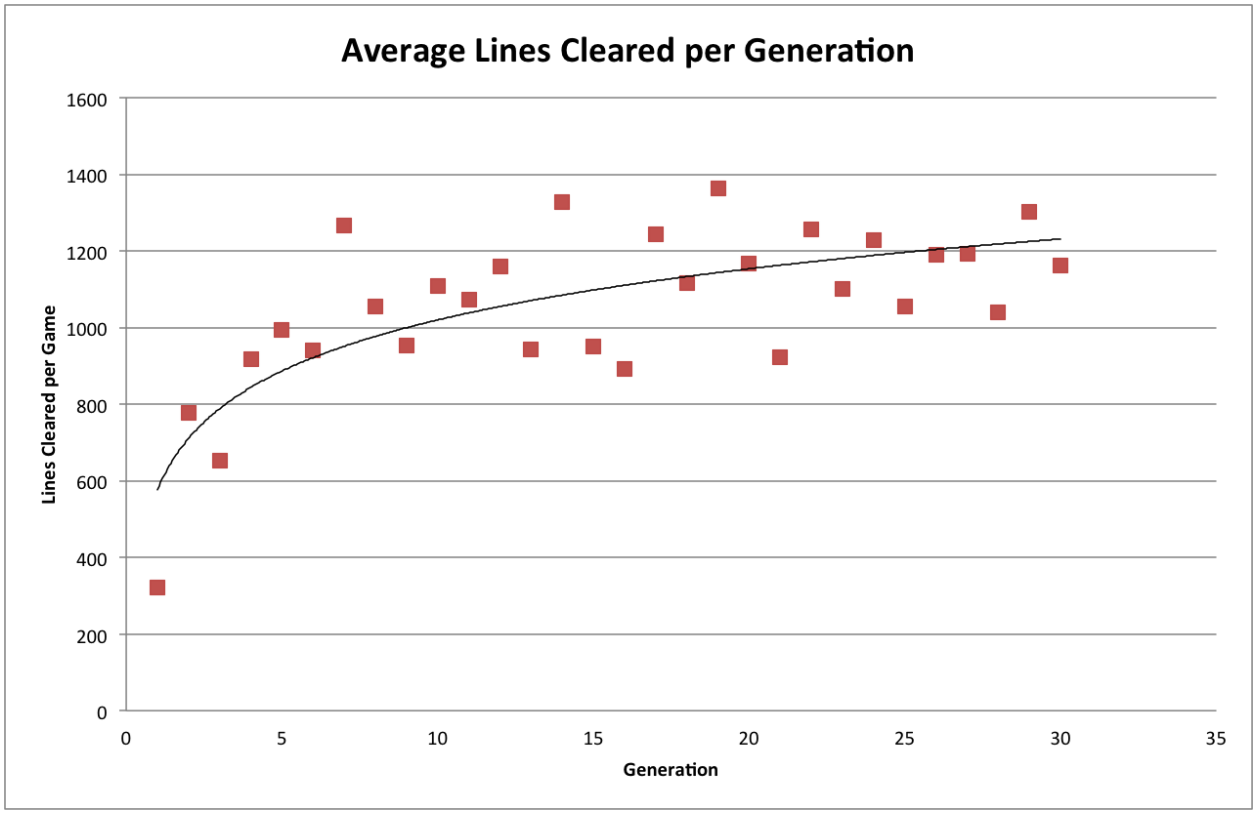
Initially we trained the genetic algorithm with a randomly generated population of 100 for 10 generations. Each member's fitness was evaluated by summing the total score over 10 games limited to 400 tetrominoes per game. One-point crossover was used to create offspring from parents and mutation was applied to the offspring with 5% probability. 30% or 30 of the weakest members of the population was replaced per generation. The results are shown in the following graph:



From this graph we can clearly see the genetic algorithm at work, with the average fitness of the population increasing in each generation. Eventually, between generation 9 and 10, the average fitness did not change. This was not because we had converged on an optimal answer, but rather because we had set a limit of 400 tetrominoes per game. At this point, most of the vectors in our population were capable of playing up to 400 tetrominoes without losing, and we were unable to determine which vectors were actually better.

Improved Attempt

In our subsequent attempts, we made several big changes in order to get better results. We still trained the genetic algorithm on a randomly generated population of 100. Each member's fitness was evaluated by summing the total score over 10 games, but this time 5000 tetrominoes were used per game (hopefully allowing us to differentiate members of our population better). This time, we used the process described in the Algorithm section of the report. We used tournament selection to select an offspring population, and for each member of the offspring population, randomly chose to perform one-point crossover, mutation, or reproduction. For each generation, once we created an offspring population of size 100, we combined the current generation and offspring generation and used tournament selection to choose 100 members out of the total of 200 to create population for the next generation. The results are shown in the following graph:



This steady increase shows the improvement of the AI as the program progresses through generations. It is necessary to note that the gain in information seems to level off at around the 7th generation onwards. Further, there is a lot of noise present in the latter portion of the graph. This apparent increase in bias of the model can come from the untuned nature of the

hyperparameters of the genetic algorithm. The optimization of the weights yields little progress for the AI, as they seem to be optimized pretty early on in the process. The bias does decrease a bit after the 20th generation, possibly due to a homogeneity of the resulting “selected” population.

Future Improvements

Increasing Search Space to Deal with Overhang

When we created the set of boards for an AI to consider, we did not make an exhaustive set of boards, unfortunately. The set of boards to be considered only included the next boards where the current piece would be applied in its various rotations across the entire “floor” of the current board, where the floor is the top most blocks that have already been set. However, this does not account for overhang which can occur. The simplest case to consider is the one where an “s” block lands as the first block. There are other blocks that can go under the inevitable “overhang” produced, but those boards aren’t considered. These boards are also most likely a better option to the other boards, as they reduce holes among other points where the score for this board might be better.

Looking Ahead

In our algorithm, we only evaluated the best possible move given the current piece. This, however, excludes a crucial piece of information that is available to us, which is the next piece that will appear. If the algorithm was to evaluate moves based on what the current and next piece are, it may be able to find a different move for the current piece, which when combined with the upcoming piece, produces a better score than the best move knowing only the current piece. We did not do this looking ahead initially because the search space was would have been squared, and we were already facing performance issues, which will be detailed in the next section.

Tuning Genetic Algorithm Hyper-parameters

The algorithm has many hyper-parameters, which include: population size, selection method, sample size for tournament selection, crossover method, mutation method, the relative

probabilities to crossover, mutate, or reproduce, number of games used to evaluate fitness of a single vector, maximum tetrominoes per game, and number of generations.

In setting up our genetic algorithm, we somewhat arbitrarily decided on what these hyper-parameter values would be based off of intuition, but given more time, we could experiment a lot more with these values. The biggest reason that we did not experiment much with these values was time, due to the large amount of time it took to train the algorithm under just a single set of parameters. With our current implementation, it took almost 10 hours to train our algorithm on a population size of 100 for enough generations to produce decent results.

If we had more time or computing power and access to distributed computing, we could experiment with using much larger population sizes and much higher limits on tetrominoes per game. The limit on tetrominoes per game is especially important because as the algorithms got good enough, they would all be able to easily clear all 400 tetrominoes without losing, so it was hard to distinguish which algorithm was actually better in the long run. We had to implement this limit, however, to prevent individual games from lasting too long and making our performance issues even worse.

Performance

Performance was one of the largest factors preventing us from creating a more powerful algorithm. We were training our genetic algorithm on a dual-core laptop, so we were limited in how big of a population we could use, how many tetrominoes to use per game, and how many generations we could create, in the interest of time. The biggest bottleneck in terms of speed in our training is the evaluation of the fitness of each weight vector. Even though we were using DEAP, which is a distributed framework for genetic algorithms, we were still training our algorithm on a laptop and not a cluster of computers. If we had access to much more powerful distributed computing, we would be able to increase our population size from 100 to several thousand and run many more generations, which could produce better results for our algorithm. We believe that these performance changes would have had a drastic impact on the overall strength of our algorithm, but we did not have the computing power to implement such changes.