Evolving Heuristic State-Value Function for Tetris

I. Introduction

This project describes the application of evolutionary algorithms in the game Tetris. Tetris is a game where the player must order falling blocks such that they create full horizontal lines which are then removed from the board. The goal of the player is to survive for as long as possible and to earn as many points as they can (there is no winning in Tetris). In this project we designed a computer program which plays the game. It chooses the next move by considering all possible subsequent board states via a rating function. This function is a linear combination of handcrafted features extracted from the game board.

The rest of the report is organized as follows. In Section II we describe our problem's environment, which is the game of Tetris, in further detail. In Section III we present our solution and explain the algorithms composing it. In Section IV we give an overview of the implementation and design choices for the major components of this work. In Section V we conduct various experiments using our described algorithm and software. In Section VI we analyze those results and explain them. Section VII concludes our work.

II. Description

Tetris was originally created by Alexey Pajitnov, a Russian software engineer, in 1984. It is played on a game board which is a 2-dimnesional rectangular grid, usually 10 cells wide and 20 cells tall. Throughout the game, pieces called *tetrominoes* (see Fig. 1) slowly fall from the top of the screen towards the bottom. Each tetromino is a positional combination of 4 squares in a different structure. The goal of the player is to rotate and move these pieces from side-to-side and rotate them in order to create seamless horizontal lines which are then cleared from the board and grant points to the player. For each line cleared, the pieces above it drop by one line. The more lines the player clears with a single tetromino, the more points they earn. The maximum number of lines that they can clear is with an "I" piece. This move is also known as "Tetris".

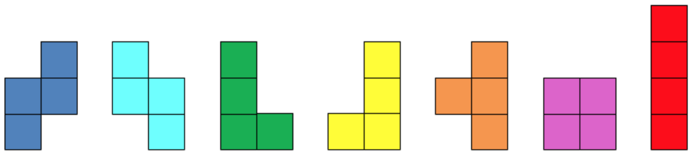

Fig. 1. The seven game pieces which are called tetrominoes. Each is referred to using the letter it most closely resembles, from left to right: Z, S, L, J, T, O, I.

A falling tetromino is stopped either by touching an already placed tetromino or by touching the bottom of the board. Once the current tetromino stops, a new one enters the board. The game ends if a new tetromino cannot enter the board because it immediately touches another one which then blocks it.

Tetris has been studied from a theoretical standpoint. It has been proven to be an NP-hard problem even if the tetromino sequence is finite and known in advance [1]. In addition, due to the existence of the S and Z pieces, there is always a sequence of tetrominoes which would cause the player to lose. The player will always lose, regardless of how "good" they place them [2].

In this project we seek a bot which plays the game optimally. This bot will rely on a state evaluation function which will decide the optimal spot to place the current tetromino in.

III. Methodology

The solution for this problem is composed of two components, the game bot and the evolutionary algorithm.

A. The Game Bot

The bot plays the game by evaluating each state it can reach from its current state, and picks the best one to proceed from. In order to do that, it has to be able to rank the states. Therefore, we utilize a scoring function which evaluates how good a board state is. Given this function, the bot can value each state it can reach by simulating the placement of the current tetromino in all possible places. It places the tetromino in the real board in the place which will maximize the board's score.

There are $7 \cdot 2^{W \cdot H}$ tetromino and board state combinations where W is the board's width and H is the board's height. For example, in the standard 10×20, we will have $\sim 1.12 \cdot 10^{61}$ possible states.

The ability to rotate and move the tetromino while it is falling makes our task significantly more difficult. It adds an element of time which makes the problem into a type of temporal planning problem, and increases the amount of states one can get from each state. Therefore, we decided to only support the ability to position and rotate the piece above the board before dropping it (and removed the ability to move it while it's falling).

Since the number of possible states is gargantuan, we must approximate our value function. To do that, we extract 7 features identified by the project's members, but which also appear in research done on the game [3].

1) *Pile Height*: The maximum of all columns' heights. A column height is defined as the row number of its highest non-empty cell.
2) *Cumulative Height*: Sum of all columns' heights.

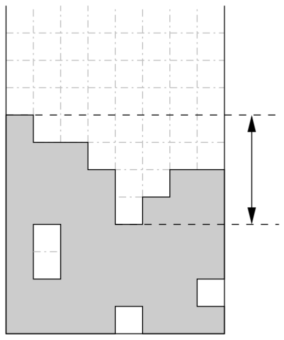

Fig. 2. An illustration of relative height (left) and wells (right).

3) *Relative Height*: The difference between the highest column and the lowest column. This is 4 in Fig. 2 (left).
4) *Holes*: Number of holes. A hole is an empty cell with nonempty cell somewhere above it.
5) *Roughness*: Difference between each column's height and the height of its neighbor to the right.
6) *Maximum Depth Well*: The length of the deepest well (with width of one). A well is a vertical line of empty cells which have nonempty cells of both their sides. The board edges are considered nonempty cells. This is 4 in Fig. 2 (right).
7) *Sum of all Wells:* Sum of all wells in the board. This is 7 in Fig. 2 (right).

Now that we can describe a board B by its feature vector $f(B)$, we will use a linear state-value function to evaluate it:

$$Board\text{-}Score(B) = \sum_{n=1}^{7} w_n \cdot f(B)_n \tag{1}$$

Let us now look at how the bot decides on where to drop the current piece and in which rotation. The bot simulates dropping the current piece in each rotation and placement, and evaluates these states. It will drop the piece in the place and rotation which lead to the board valued the highest.

In principle, we can calculate a lookahead for k moves forward by considering all pieces at each stage and averaging over them. But that would require enormous computational power and make for a slow and unusable bot. We must strike balance between looking ahead and the amount of time it takes to perform each action as a result of that lookahead.

Since in Tetris, we can see both the current piece and the next piece, we will consider them and not perform lookahead of future unknown pieces (see Fig. 3).

B. The Evolutionary Algorithm

In this algorithm, the genotypes are the weight vectors, and the phenotypes are the bots which play the game. For the initial population, we created 50 vectors each with 7 random integer values in $[-100, 100]$.

For the crossover operation, we used standard two point crossover with $p_c = 1$ which means the crossover always takes place. It returns two children for each pair of parents.

The mutation operator considers each gene, and mutates it with probability $p_m = 0.3$. The mutation works by multiplying the gene by a Gaussian distributed random number with $\mu = 1$ and $\sigma = 0.5$.

The fitness of an individual is measured by how well it performs in the game using the genotype as the weights for its value function. In Tetris, one can use 3 main ways to evaluate how good a performance is. We can use either the game's score, number of lines cleared, or number of tetrominoes dropped. We chose to evaluate the performance based on number of tetrominoes dropped, which is largely proportional (in large scales) to number of lines cleared. This method provides no advantage to clearing multiple rows simultaneously, so greedy line clearing is enforced.

We calculate the fitness of an individual as an arithmetic mean of the number of tetrominoes dropped across 5 games. Due to the low number of games, this average is not a perfect fitness measure. We observed multiple individuals which had 5 games with certain amount of tetrominoes dropped, and then 5 different games with a vastly different results. Moreover, playing whole games for fitness becomes slower as the generations improve. The better the individuals get, the longer each game lasts. A good game of Tetris can drop millions of pieces and take several days [4]. As a result, we reduced the board size to 6×12, which made the game harder and much shorter.

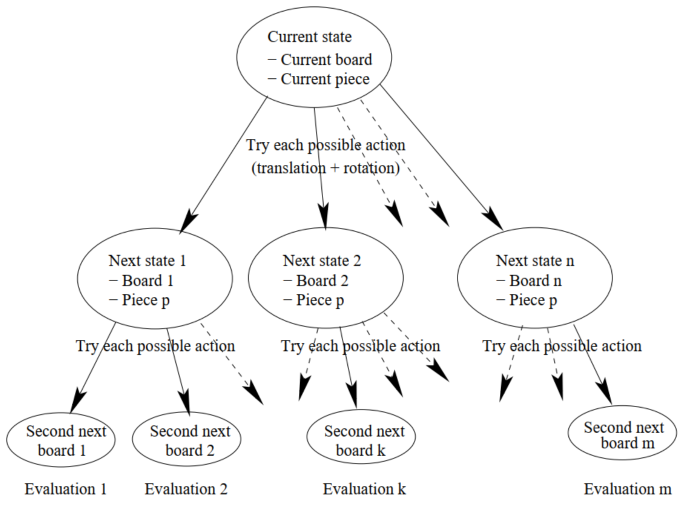

Fig. 3. A two-ply evaluation process. In this context, translation is the location of the piece, i.e., offset from the right edge of the board.

IV. Software Overview

The code of this project was written in Python and can be found at https://github.com/Lior8/EvolutionaryTetris.

A. Tetris Environment

The game environment uses a very simple game loop to run (See Fig. 4). It initiates the board, which is an empty matrix. It then randomly chooses the next tetromino. It informs the player of the current board's state, the current tetromino and the next one. Then it gets the location and rotation to drop the current tetromino in back from the player.

The drop is a loop which continuously checks for a collision between the current tetromino and other ones already placed in the board or the board's bottom. If there is a collision, we freeze the tetromino in place and check if part of it (or all of it) is above the game board. If it is, the game is over. If not, we clear any full horizontal lines created as a result and continue to the next piece.

The Tetris environment is written as a collection of methods (for the important ones see Fig. 4), which always get the game board alongside the other parameters needed. It was done this way to ease the ability to duplicate the board in the bot decision process as it requires copying and modifying it many times which would be harder if it was designed as a standalone class.

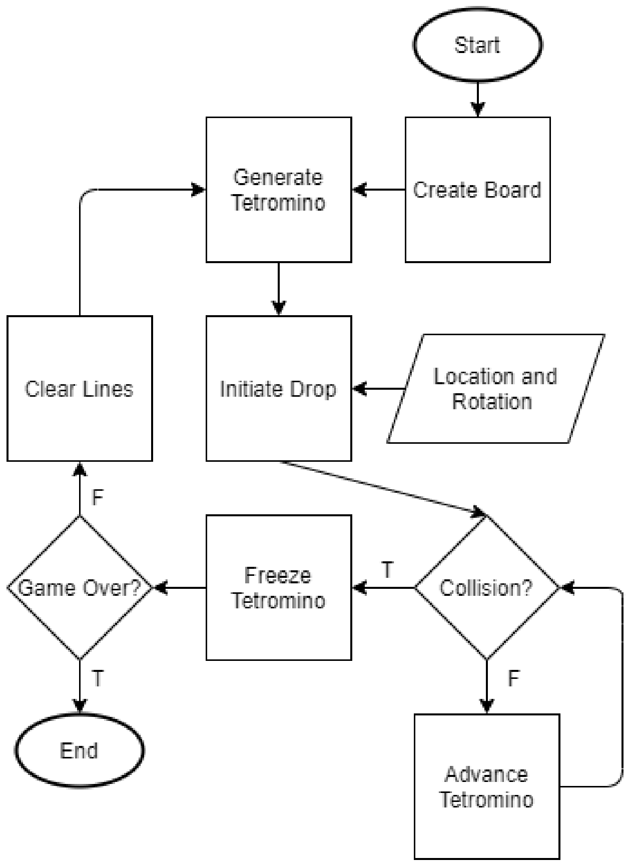

Fig. 4. A flowchart of the game's logic.

B. Game Bot

Given the the current board's state, the current tetromino, and the next tetromino, the bot performs a two-ply search for all possible states it can reach with these two pieces.

It runs over each position and rotation of the current tetromino and drops it at that location. For each of these new board states, it considers the next piece as if it was the current piece in a one-ply search and drops it in all positions and rotations on that board. Then, it extracts the features from each board it reached using these two steps, and calculates its value.

The value of each board state in the outer loop (using the current tetromino) is the maximum value of the states reached from it in its inner loop (using the next tetromino). The place and rotation we will drop the current tetromino in the real board is the one we calculated the highest value for.

It then sends the position and rotation to the environment, and waits for an update (next move / game over). We count the amount of tetromino dropped, and when the game is over we return this number.

The bot is a class which can be initialized with the weight vector and has the function *play_game*() which makes it play a game.

The feature extraction is an interface which receives the board and returns a vector of its features in the order they appear in Section III-A. It does so by iterating over the game board two times in a column-after-column manner. One for searching all the wells and their sum, and the other iteration is for all other features.

C. Evolutionary Algorithm

The evolutionary algorithm follows the traditional structure. Initialize a population, measure its individuals' fitness, perform crossover and mutation on them, and repeat with the new population.

The evolutionary algorithm is done via a class which can be initialized with the different parameters (or using the default ones). It has the function *evolve*() which can be called to run the evolutionary algorithm. It also logs its results to the screen and to a file for later usage.

We utilize multiprocessing for the fitness function by separating each individual's fitness calculation into an independent task. The task is to run 5 games with that individual and return the number of tetrominoes dropped in each of these games. Once all tasks are done, we have the fitness for each individual in the population and can proceed to perform all other methods in the same process as they are not computationally heavy as the fitness calculation.

V. Results

Using our value function with a 6×12 game board resulted in the top game of 61,025 pieces dropped, and top performers across different runs dropping more than 10,000 pieces per game on average.

An example of such top performer it a bot with the weight vector $w = (-0.185, -1.89, -30.895, -312.942, -47.799, -17.346, -161.521)$ which had a best game of 19,812 pieces dropped and an average of 12,824 pieces dropped per game.

Fig. 5-7 describe 3 different runs using the parameters described in Section III-B across 30 generations.

VI. Discussion

Our expectation was that all weights in the top performers in each run will be negative as human players desire to minimize

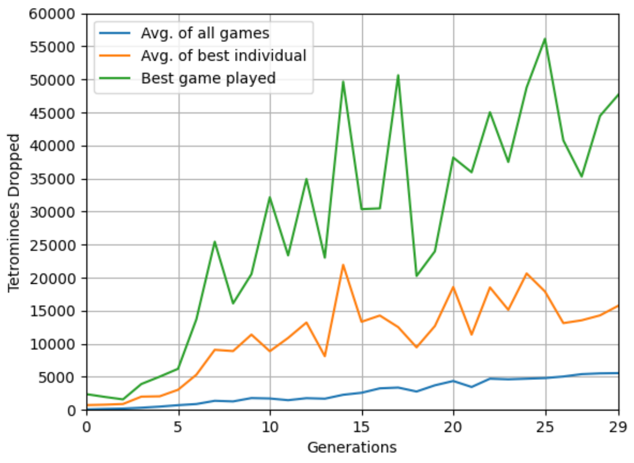

Fig. 5. Run 1.

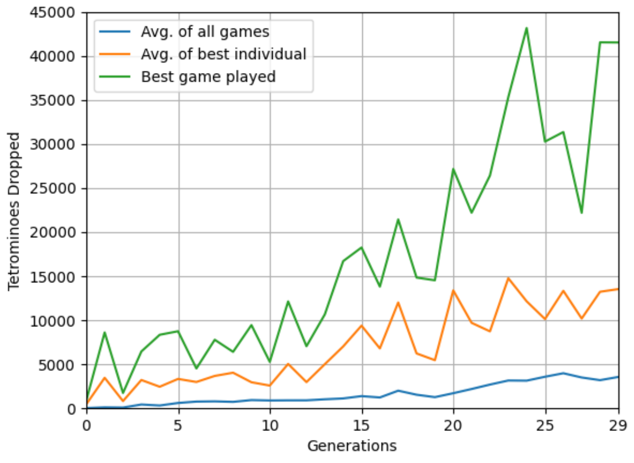

Fig. 6. Run 2.

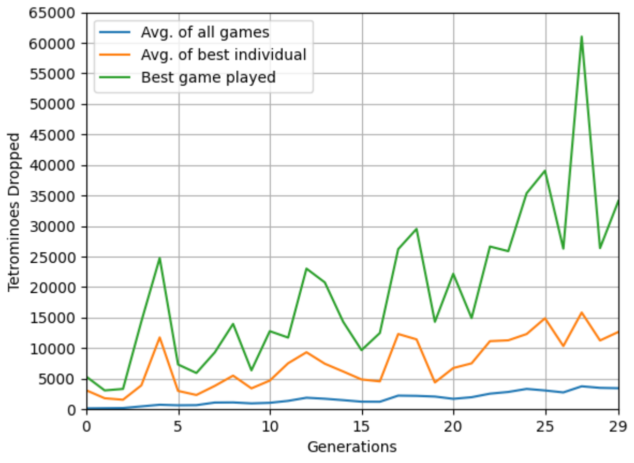

Fig. 7. Run 3.

each of the features we defined when playing with a greedy line clearing strategy.

The two features that seem to be the most important when determining the board's value are *Holes (4)* and *Sum of All Wells (7)* as their weights consistently had the highest absolute value by a large margin from the rest of the features, with *Holes (4)* being the more important one.

Across the different runs, only the two main features remained with large negative weights. *Relative Height (3)* and *Roughness (5)* usually got weights significant enough to affect the value, but not close to those of the main features. Furthermore, even they got turned off from time to time in competitive individuals or even across entire generations. This shows that they are not as important as *Holes (4)* and *Sum of All Wells (7)*.

Pile Height (1), *Cumulative Height (2)*, and *Maximum Depth Well (6)* got consistently turned off in competitive individuals across most runs. It can be hypothesized that the value function assigns lower weight to height and higher weight to more dangerous phenomena such as holes or wells, as if they are left untreated, would quickly lead to a game over. By assigning high weight to height, the bot might prefer to increase the number wells or holes to keep the height down, making the board more unstable and easier to lose from.

Maximum Depth Well (6) gets turned off probably because it conveys known information. Since it is already present in *Sum of All Wells (7)*, it is deemed redundant.

While all the non-main features have appeared turned off in competitive individuals, it has never happened simultaneously. It can be hypothesized that they share some information about the board's state and value. This causes only some of them to be turned off and not all of them together. We would need to further test the underlying relations between these features to substantiate this hypothesis.

VII. Conclusion

In this project we tested the usage of evolutionary algorithms in order to evolve heuristic state-value functions for the game of Tetris. The performance of the state-value function was tested via running simulations of the game using a bot which utilizes two-ply search to determine its next move. It determined the value of each move using the evolving weights in a linear function which approximates the board's value.

The results showed that the evolutionary algorithm did work and found good weights for the state-value function. The last generations performed significantly better than the initial ones, and reached games over 10,000 tetrominoes dropped regularly.

References

[1] E. D. Demaine and S. Hohenberger and D. Liben-Nowell, “Tetris is Hard, Even to Approximate,” in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 351–363.
[2] H. Burgiel, “How to lose at Tetris,” The Mathematical Gazette, vol. 81, no. 491, pp. 194–200, 1997.
[3] C. Thiery and B. Scherrer, “Building controllers for Tetris,” International Computer Games Association Journal, vol. 32, pp. 3–11, 2009.
[4] N. Böhm and G. Kókai and S. Mandl, ”An Evolutionary Approach to Tetris,” in The Sixth Metaheuristics International Conference, 2005.