Tetris Game Playing Agent with Genetic Algorithm

Introduction

The goal of this project is to explore and compare different ways to go about creating a game playing agent of the game "Tetris." In the game, the goal of the player is to stack an endless series of differently shaped blocks without the stack reaching the top of the screen. When a row is filled with blocks, it is cleared and the player is rewarded some amount of points. If multiple rows are cleared at the same time, then more points are rewarded to the player. By performing this comparison, we will find which approach is better suited to use when one wishes to implement a game playing agent. The methods discussed could be applied to games similar to Tetris, and even beyond that with some adaptations. The approach that will be explored in this paper will be using a genetic algorithm in an attempt to find a state evaluation function that will optimize the game playing agent for maximum time played. One approach I have found uses a genetic algorithm to optimize for maximum score, and another approach uses an artificial neural network with reinforcement learning to optimize for score. I will attempt to compare the performance of my approach with both of these.

Related Work

https://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/

The first approach attempts to find an optimal state evaluation function using a genetic algorithm, similar to how I will be approaching the problem. For the evaluation function, they used a set of seven attributes of the space in order to evaluate the utility. These seven attributes are: total height, number of holes (blocks in a row that are not filled), number of blockades (holes with blocks further stacked on top of them), possible clears, edges of the current piece touching other pieces, edges of the current piece touching a wall, and edges of the current piece touching a floor. After completing their experiments, the approach did yield a result better than their initial evaluation function, which was just assigned arbitrary weights. This was to be expected though, since arbitrarily assigning values is not very accurate. In their genetic algorithm implementation, the fitness function they used was simply the score that the agent was a able to achieve before it lost the game (i.e., the stack of blocks reached the top of the playable area).

In the implementation of their genetic algorithm, they used a simple list of seven doubles to represent the seven attributes that were being used for state evaluation. The population of each generation was 16, and they simulated 10 generations of agents. In the simulation, they used a board size that was smaller than that typical Tetris board of 10x20. Instead they used an 8x20 board. The reason for this was that when simulating the agents playing the game, some would last for hours, which added up to an unreasonable amount of time. In the initial generation, each list consisted of randomly generated numbers. Each set of weights was used for an evaluation function, and a game was simulated, with the final score being recorded. The algorithm was implemented with a tournament style of selection for the best scores.

The lists would be paired up, and the best score of that pair would be accepted to "breed" the next generation. To generate a population for each new generation, each attribute in the list would be randomly selected from one of the parent lists with equal probability. There was also a "mutation" aspect, where an attribute had a chance of being set to a random value. In the experiments, this value was set to 10%.

While this method ultimately proved successful in finding an evaluation function that could play better than the the evaluation function with arbitrarily selected values, there are areas in which the method could be tweaked in order to further improve it. It's possible that a different set of attributes could be used to evaluate the states, or some of the ones used in this method could be tweaked to be better optimized. The fitness function of the genetic algorithm could also be changed to either account for the amount of time played, or the amount of score in a certain time frame.

"Reinforcement Learning and Neural Networks for Tetris" - Nicholas Lundgaard, Brian McKee

This approach was performed by a group of researchers from the University of Oklahoma. The researchers implemented a series of Tetris playing agents: A random agent (which randomly placed pieces), multiple types of heuristic agents, and learning agents that used multiple types of neural networks and reinforcement learning. To evaluate the state of the space, they used the following series of values: the current piece, the next piece, the board height, the board level (a boolean that determined if the top of the block stack was level within a certain range), "Has Single Width Valley,"(boolean which determines if there is a slot that an "I" piece might fit in to), "Has Multiple Valleys,"(boolean that determines if there are multiple single width valleys), and the number of buried holes on the board(which is similar to the blockades of the previous work. This value was restricted to just be 1 value that determined the range of the number of holes). The researchers also defined a set of actions that the agent could perform on a given state space. Their research consisted of testing and comparing multiple types of heuristic based agents with multiple types of reinforcement / neural net based agents. They also used a random agent that randomly placed pieces on the board to provide a baseline for comparison.

After performing their experiments, the researchers found that the "Clear Board" heuristic agent, which prioritized minimizing holes and clearing rows preformed the best out of the heuristic agents. It was able to earn an average of 56,449 points over all the runs, compared to the next best heuristic agent with 1,231.56 average score over all the runs. They also tested a reinforcement learning agent, but the performance of the agent was not comparable to the "clear board" agent. They tested it in a single player environment, where it was found that the agent did not "learn" to improve its score over the runs. They also tested it in a competitive environment against a known "expert" agent, where the results were similar to the single player environment. Next, they used a low-level neural network to train the agent, which operated on only the low-level state representation that were mentioned earlier. After running 6,500 games, the performance was comparable to the random agent. It did however show an increase in

performance after a certain amount of runs, so the agent was learning and changing its strategy. Lastly, the researchers implemented a neural network that operated on the high level state representation and actions described above. This agent also demonstrated a learning capability, and the performance after 500 runs was comparable to the "clear board" agent. The average score over all the runs was 51,520 compared to the "clear board" agent's 56,449.

Approach

In both of the previously discussed approaches, the reward function that was used was the score of the agent after it was unable to continue the game. I propose that instead of using the score as the reward function, use the number of turns the agent lasts. The inspiration for this approach came from looking at the two previous approaches and wanting to test if the number of turns lasted would work as well or better than the approaches. I could not find any other approaches that used the time lasted as a reward function. In theory, if a time based agent can last for a significantly longer time than a score-based agent, then it will be able to eventually surpass the score of the score-based agent.

For my implementation, I wrote mostly everything from scratch in python 3.6. Although implement the project from scratch incurred some serious difficulties (mostly in the Tetris game simulation), it was almost necessary so I did not have unneeded features bogging down my simulation. The Tetris grid is represented as a two dimensional list (list of lists), and the pieces are also represented like this. On the board, an empty space is represented as a zero and a non-empty space is represented as a number in the range 1 - 7, depending on the piece occupying the space. In order to make the implementation more efficient, the board is not animated in a typical fashion with the pieces falling from the top of the board. The simulations are also run on a half size board in order to be able to run the simulations faster.

When running the simulation for a single game, the agent generates each possible position of the current piece on the current state of the board. It uses an evaluation function to determine the value of that state. It then simply picks the state with the highest utility and plays that move. The features that are evaluated are the total height of the board, the height difference of the highest piece and the lowest, number of holes, number of buried holes, and row clears of the state. A hole is defined as any empty spot that has the place directly above it filled. A buried hole is an empty piece that has a filled spot at any point above it on the board. When the agent is first created, it accepts a list as an argument. This list contains 5 integers which are the weights of the 5 attributes of the state. As I will discuss later, it is simple to add new feature evaluations to the agent, so future work should not be complicated. The state is evaluated using the following formula:

State value = (weight[0] * max_height) + (weight[1] * height_difference) + (weight[2] * num_holes) + (weight[3] * buried_holes) + (weight[4] * row_clears)

The weight values are what are varied and developed using the genetic algorithm.

The implementation of the genetic algorithm is relatively simple compared to actually creating the Tetris simulations. It simply has a population size, which is the number of different weight lists that are tested in a generation, and a variable amount of generations. During the first generation, each members' values are randomly generated. For each successive generation, the top contenders of previous generation are selected to "breed" the population of the next generation. In my implementation, the top half of the generation is chosen, and they create a new generation of the same population size. The contenders are chosen based upon the number of blocks they were able to place before the game ended, i.e. the amount of time they lasted playing the game. To create one "child" for the next generation, each of its weight values is randomly selected from one of the parents' weight values in that category. There is also a chance for a "mutation" in which an attribute may be randomly varied. In my simulations, this percentage was set at 0.1%. As the generations are running, the maximum of each generation and the average value of the number of turns is recorded.

Although some satisfactory results were found, as will be showed later, differences in my implementation prevented me from effectively being able to compare my results with the previously discussed approaches.

Experiments

The baseline expectation for the implementation is that the performance of the agents would improve as the program ran through the generations. In order to test this, I simply just need to keep track of the performance during each generation and then compare the later generations to the earlier ones. One assumption about the experiments is that each run will have a similar ordering if pieces so the results can be compared. While obviously not set to a certain order, it is important to use the same selection method for the next piece for every run. To do this, when selecting the next piece I would remove the selected piece from the selection pool. After the selection pool became empty, I would refill it with a random ordering of pieces. This ensured that there was only one opportunity per cycle for the same piece to be received twice in a row. This also prevents sequences of pieces that would cause the agent to automatically lose.

The tests were conducted with various population and generation sizes. For each generation, I recored the average number of turns over the population and the maximum number of turns for that generation. I ran multiple sets of tests with varying number for population and generation size. The data presented below shows the average turn count for each generation for runs of 100 population with 50 generations, and 30 population with 30 generations. The value for each member of the population is itself an average of 3 runs, in an attempt to prevent outliers. Also below is a graph with the max turn count for each generation of the 50 gens and 30 gens run.

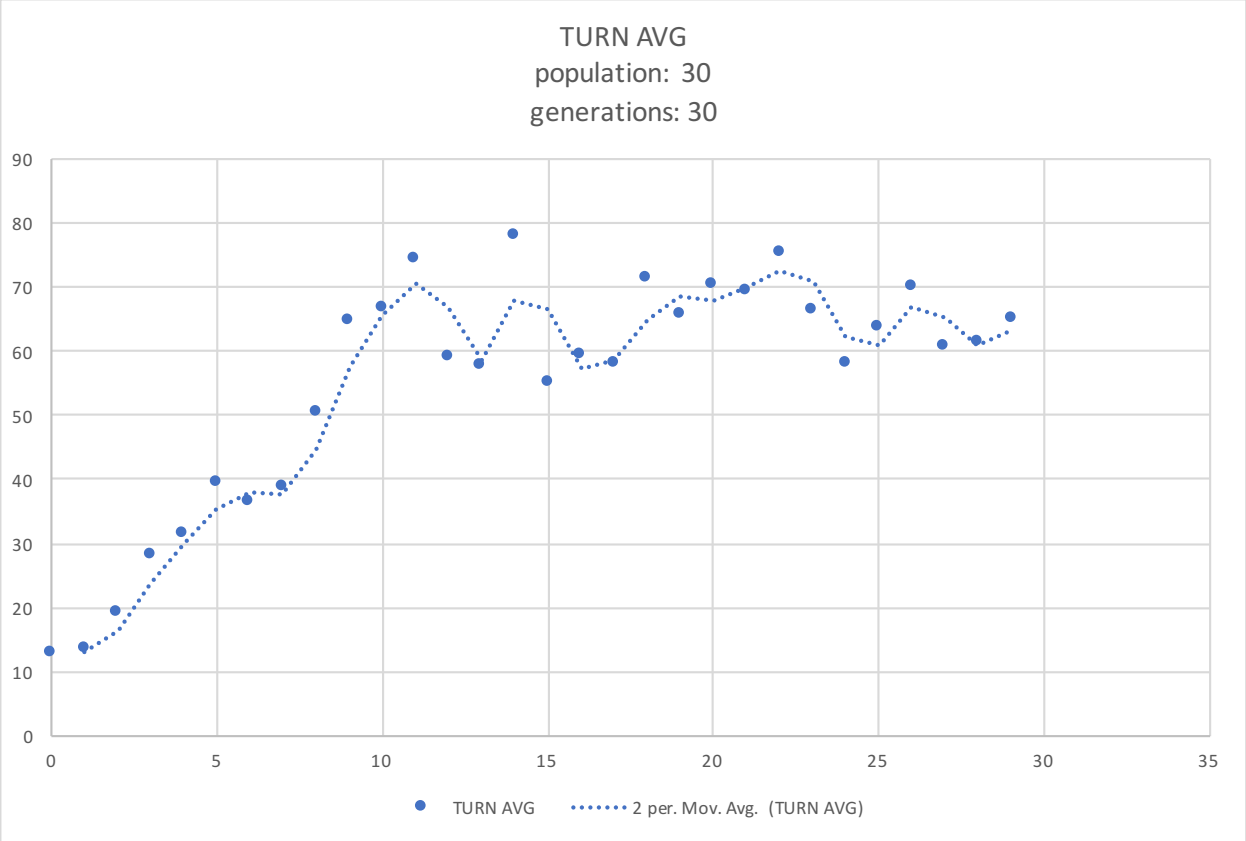

FIGURE 1. AVERAGE TURN COUNTS FOR 30 POPULATION, 30 GENERATION W/ MOVING AVERAGE

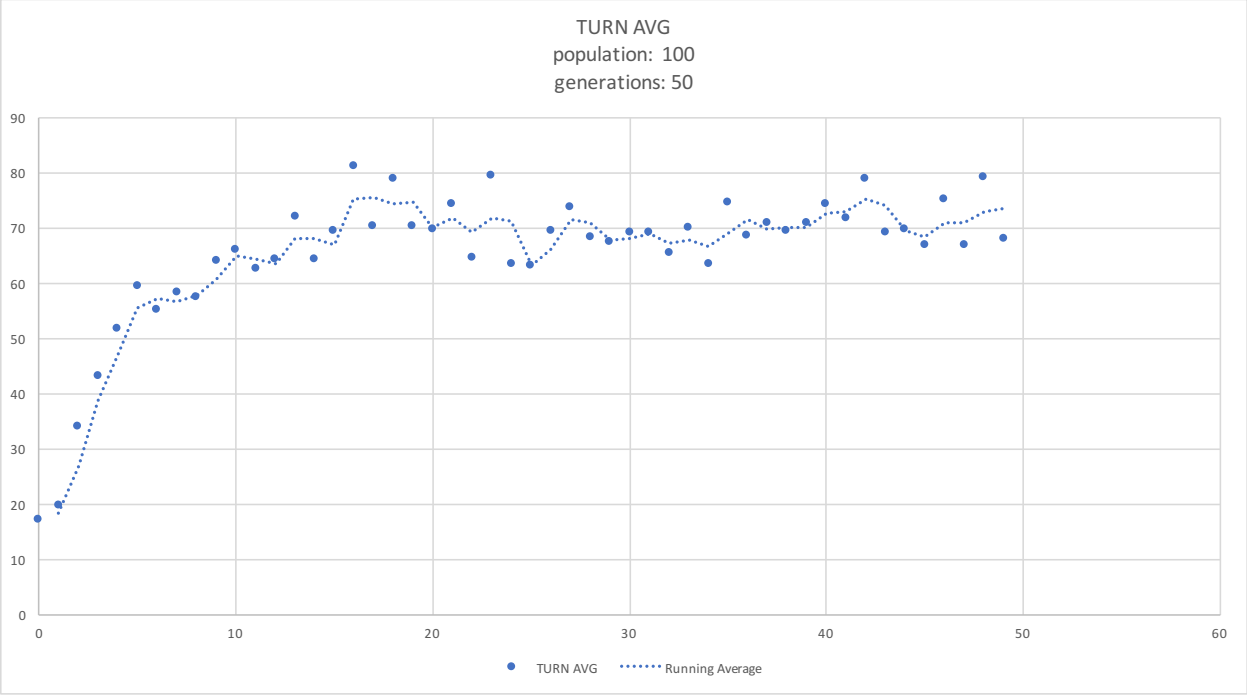

FIGURE 2. AVERAGE TURN COUNTS FOR 100 POPULATION, 50 GENERATIONS W/ RUNNING AVERAGE

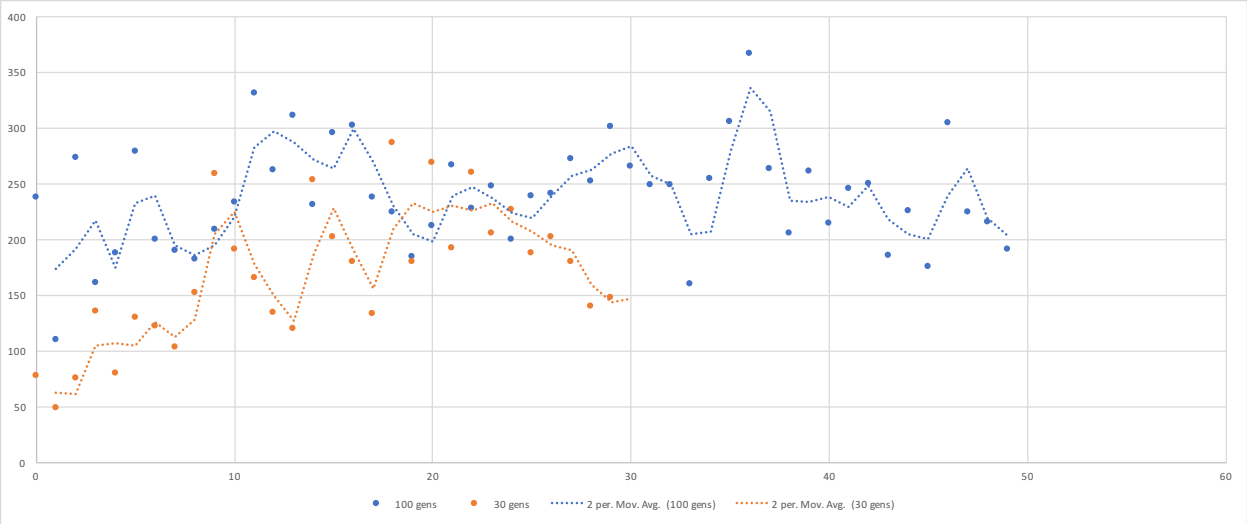

FIGURE 3. MAX TURN COUNTS FOR EACH RUN, W/ RUNNING AVERAGE

The two runs start out with great improvement in turn counts and then both even out in the range 60-80 turn average per generation. This evening out happens within the range from 15-20 for each run. According the figure 3, The max turn counts for each generation is greater in the 50 generation run than the 30 generation run.

Analysis

In regards to proving or disproving my hypothesis, the data does support the hypothesis that the genetic algorithm was able to create a better Tetris playing agent than the initial agents in the first generation. Although the maximum turn count evens out in the range 60-70 turns, it is important to note that the simulations are being run on a reduced size board of only 10 height. It is not surprising that the average turn counts even out after a certain number of generations. As the better candidates are selected from the previous generation, all of the children from the next generation will be similar in performance. After a certain amount of generations, the children perhaps become too similar, and produce children that are all alike. There was a mutation function that randomly changed values with a certain percentage, but it could not vastly improve overall results. After I was done running the genetic algorithm, I ran the best agent on a larger sized board. I ran into a problem when doing this though, and the maximum board height that could be run was 12. Because of this, I could not effectively compare my results to the previously mentioned work. The previous works did mention that their agents were able to have very high scores and over thousands of blocks placed on a 10x20 board, so it is a safe assumption that if my agent was run on normal sized board, then it would not perform as well as the other agents.

Conclusion

After analyzing the previous methods of trying to develop a Tetris playing agent, my goal was to develop my own using a combination of features of the two methods described above, along with some new features introduced by me. One of my goals was to create a genetic algorithm in order generationally improve upon a game playing agent. While the overall performance of my agent could not be effective

compared to the previous methods, my goal of creating an agent that improves upon itself was reached. As seen by the data presented above, the genetic algorithm improved the upon the initial population of agents. These results show that a genetic algorithm can be used effectively to improve upon an initial population. This method can be applied to any game where the state evaluation can be simplified to a series of values with corresponding weights.

In the future, I'd like to test out a large range of state evaluation functions for the Tetris game. I'd like to also further generalize my genetic algorithm framework so that it could be applied to any other game with similar state evaluation functions. I also plan on cleaning up the code and resolving bugs, so that better experiments can be run.