Examining How Performance Scales with Increased Complexity of Two Basic Tetris AI Agents

Alec Warren
Northeastern University
alecwarren19@gmail.com

Abstract

Tetris is the third most popular game of all time, and if this were written only a few years ago, it would actually be the number one most popular game of all time. To honor this timeless videogame, this project's purpose is to create a basic online single agent AI that can play a simulated game of Tetris, with the goal of being able to clear several levels of the game and score well before inevitably losing. Two different AIs are created, trained, and evaluated to examine if increased complexity is directly related to improved performance. Tetris provides a simple yet visually pleasing means of observing a single agent AI play a game. Furthermore, Tetris provides an extremely discrete state space, with a simple 20x10 grid as the game board and a discrete integer output for the result of a game: score and level reached. This project requires a fully custom simulation programmed from scratch so that the AI can exists just within this simulation and therefore be guaranteed to rely on the mechanics in this simulation, eliminating an error due to any various discrepancies between the countless versions of Tetris that exist. Overall, both AIs functioned relatively well, but were wildly inconsistent, failing to even score a single point every so often, and then just as often, clocking in scores in the hundreds of thousands, clearing anywhere from level 9-12. In the end, the less complex algorithm was faster with near identical performance.

1 Introduction

This project aims to create a Tetris AI that will be able to successfully make "optimal" decisions on how to move and place game pieces to progress through multiple levels of Tetris, where each level is exponentially harder than the prior. A simple yet highly functional, accurate, and extensible simulation will be created to "host" this AI. All of the programming will be done in python, with the python Tkinter module as the GUI for the simulation. This simulation will stay extremely true to the original rules for Tetris so that this is applicable in the real world. This is also to keep the rule set simpler than the more recent implementations of Tetris.

The AI is an online agent that will choose a set of moves that result in the piece being placed in an optimal place on the board. The optimal place will be determined by a state evaluation heuristic function, with several weighted parameters that will evaluate the utility of the given placement. These parameter weights can then be tweaked to attempt to "train" the AI. A massive component of the AI will be its ability to "prune" its decision making tree to prevent the AI from taking several seconds to calculate a decision for a single turn.

2 Meet the Agents

There are two agents that are created, trained, and tested in this project, one more complex than the other. In fact, the second AI leverages the first in its decision making process and then builds from its output. Thus, the second AI is inherently more complex.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Greedy is the first of the two AIs. Its name is given by its decision making process. It chooses the move with the most utility for this turn, ignoring what repercussions may follow in the next move, or any future moves.

Yield to Next is the second and more complex of the two AIs. This agent does exactly what its name implies: yields to the next tetrimino. This is done by calling the "Greedy" agent using the current playfield and the next tetrimino, and then recording the optimal location for the next tetrimino returned by this call. Then, a nearly identical maximizing agent is used to find the optimal path for the current tetrimino but removes any terminating states that result in interfering with the location the next tetrimino was determined to be placed at.

2 Background

Tetris:

A *tetrimino* is one of the 7 game pieces in Tetris, and a *mino* is an individual square that combines with other minos to make up the tetriminos. The playfield represents on large stack of minos and is created by repeatedly placing tetriminos on the playfield. Each turn begins with a tetrimino being spawned in at the very top of the playfield and then ends when the tetrimino is locked down onto the stack. The tetrimino will drop a row at a time at a certain rate depending on the level, with drop times getting exponentially shorter with each consecutive level. While the tetrimino is not locked down, it can move left and right, and rotate clockwise and counterclockwise if there is enough room for the move to be executed, as minos are solid and cannot overlap. The game ends when a tetrimino is placed above the skyline, or above the visible playfield.

Online Maximizing Agent:

The agent will decide on the best set of moves by maximizing its utility and return the set of moves that lead to the maximum possible utility, based on its parameter weights that are used in the AI's heuristic evaluation function. An important part of these maximizing decision tree agents is pruning so that the best path is returned within a reasonable amount of time by eliminating any unnecessary or repetitive actions from the set of actions a node can explore.

4 Alternative Considerations

For the scope and timeline of this project, a different type of AI would probably been unrealistic to fully implement, as even the simpler maximizing agent used in this project took about 50 hours to implement (that's just for the AI, not even the simulation). These obstacles aside, a potentially viable alternative would have been to create a neural network. The major inhibitor for the potential success of a neural network to solve Tetris is the sheer number of permutations that can present itself in a 10x20 board, and training could potentially take an unreasonable amount of time.

5 Methods

AI Outputs:

Both AIs calculate and output the same thing: a path to the optimal position to place the given tetrimino based on the current playfield and the utility this move would provide, based on its heuristic evaluation function. Formally, the outputted path is an array of strings, representing moves in chronological order, where each move is one of: "fall", "drop", "move_left", "move_right", "rotate_left", and "rotate_right". "Drop" is the final move in any path, as it indicates that the tetrimino has reached a terminating node in its decision tree and is done executing moves so it can be dropped down on top of the stack of minos on the playfield and be locked down, ending this turn. "Fall" on the other hand indicates that the piece is going to continue to descend in case a special opportunity presents itself as it is falling. Any consecutive sequence of "falls" that occur in the path immediately prior to the terminating "drop" are removed and thus just "drop" is left in their place as continually falling and then dropping once landing is the same as just dropping

AI Inputs:

The inputs are similar for both agents, with the only differencing being that the "yield to next" agent requires the next tetrimino to run its decision tree. These are the information given to the AI, in detail:

Playfield: 2D-Array of characters. A ' ' represents an empty space and a character ∈ ['c', 'y', 'p', 'g', 'r', 'b', 'o'] represents a mino with the color denoted by the starting character of the color.

Active Tetrimino and Next Tetrimino – each is an object with an orientation, 'N', 'E', 'S', or 'W', X and Y coordinates, where (1,1) is the bottom left corner of the playfield, and the 2D-Array of characters representing the tetriminos minos. A ' ' represents an empty space. A ' ' represents an empty space and a character ∈ ['c', 'y', 'p', 'g', 'r', 'b', 'o'] represents a mino with the color denoted by the starting character of the color.

Press speed - An integer representing how many actions can be completed per second by the AI. This makes it so the AI can't instantly execute its entire set of moves, and must rather execute them after a given interval of time to simulate how a human would have to actually press buttons to move a tetrimino.

Level - An integer representing what level the game is currently on. Increases the difficulty of the gameplay by increasing how fast tetriminos fall. Acts a multiplier for calculating score.
Parameter Weights – Dictionary mapping weights to the different parameters used in the heuristic evaluation function.

AI's Game State Breakdown:

Tetrimino: Tetrimino Object that gets modified and put into child states to pass to child nodes.
Parent Move: Move executed by parent to reach this game state. Is used in pruning the tree and creating child states to pass to child nodes.
Inherited Moves: Set of potential moves for this node to explore given by parent.
Timer and Gravity Timer: Two separate values used together to determine when the tetrimino should fall down row.

The Decision Tree:

The decision tree for each agent is comprised of a tree of MAX nodes connected by edges representing a taken action. Each MAX nodes passes along a modified copy of the Each MAX node is responsible for pruning its resulting tree by eliminating any action for it's following MAX nodes that are either invalid (not room for the move to be executed) or would cause unnecessary repetition. For instance, a MAX node that was the result of moving the tetrimino to the right would have had the action "left" removed by its parent node from its given set of potential actions. This is because if a tetrimino moves to the right and then left in succession has not accomplished anything, and now there are going to be two duplicate decision trees. This repetition would exponentially slow down the decision tree's evaluation. A terminating node is reached when there are no more valid moves left, i.e., the tetrimino has locked down, and this terminating node places the tetrimino on the playfield and calls the heuristic evaluation function on this new temporary playfield and passes this value back up the decision tree.

6 Experiments

The experiments for this project were rather simple due to time constraints. At first, manually chosen parameter weights were used to test the AIs until they were deemed to be functioning properly. Initial trials were run with random parameter weights to gauge how the two agents performed with different weights. Then, the best set of parameters from the random trials was use as a starting point in the training for both the AIs.

A very strange implementation of hill climbing was implemented in the training of the AIs. Each parameter weight is given an upper and a lower bound in the AI's python file in which the random trials chose values from randomly, and from which the training chooses its next random weight modification to try. The hill climb takes in two important values: max delta and learning rate. The max delta is what percentage of each parameter's weight bounds the given parameter can be modified by. Then after running a batch using the new temporary set of modified parameter weights, a batch of trials is ran and the average score is compared against the score of the current unmodified weights. If the new modified weights are better, the weights are added to the current set of weights, by a factor of learning rate times (score delta / old score), such that a higher learning rate or larger increase in score would result in a more drastic change to parameter weights.

7 Analysis of Results

Both AIs functioned quite well, and their average scores were nearly identical. "Greedy" clocked in at an average score of 40,757 vs "Yield To Next" with 40,468, with both ending with an average level clear of 5.1. Despite similar performance, their efficiencies were unsurprisingly far apart, with "Greedy" having half the average runtime of its big brother. And this makes complete logical sense, as "Yield to Next" is essentially just two "Greedy's" combined. The highest score observed was by the "Yield to Next" agent, at over 510,000, reaching level 14, one shy of the max level 15. Unfortunately, due to how unrefined the AIs are, these amazing results are rather infrequent.

8 Conclusion

Overall, the results of the experiment were successful. Two agents were successfully created and were able to play through several levels of Tetris on average, with decent parameter weights. The biggest roadblock to making the AIs more refined was the lack of time. Not just for tweaking the code, but more importantly, running learning trials on the AIs. Since playing a single game of Tetris can take anywhere from a couple minutes to upwards of 15 minutes, very few trials could be run. Thus, small batch sizes of 5 had to be used, along with a learning rate of 25% so that the AI could complete more than a pathetic handful of learning batches overnight. With more time, batches of size 50-100 could have been used and this would have minimized the randomness and errors caused by the extremely incontinent performances of both AIs. And thus, with more

accurate batch averages, the AI would actually be able to converge.