AlphaTetris: Competitive Tetris with Tree Search

Michael Wong, Quan Do, Kyung Taek Lee

May 18, 2018

Abstract

We created an AI that plays a popular version of competitive Tetris, the efficiency of which is evaluated by a score scheme that takes into account perfect clears, combinations, and t-spins. We represent the game as a dynamic tree in which each node consists of a current Tetris board state, a current piece to be placed, and a held piece. By performing lookahead search and ranking each leaf node according to different heuristics, we determine the best next move for any given state. We successfully implemented a Tetris-playing AI that optimizes for most of number simultaneous 4-line clearance.

Tetris is a 20 x 10 tiling game in which a player must configure different geometric shapes (called tetriminos) - one by one as they appear on the board - to create complete horizontal lines across the screen, which are then cleared - disappearing and shifting the rest of the board down. The tetrominos are of 7 types (I, J, L, Z, S, T, O); we will refer to a tetromino interchangeably as a piece in this report. The player can rotate tetriminos clockwise and counter-clockwise, as well as move them in either direction on the x-axis and only down on the y-axis. The game ends when the height of the structure formed by tetriminos on the board exceeds the height of the board (20 units on the y-axis). In the original version, Tetriminos are spawned in a random order and the identity of the next piece in the sequence that will follow the current piece is always known by the player.

It has been shown that many subproblems of Tetris – such as maximizing the number of cleared rows, minimizing the maximum height of the built structure, or maximizing the number of pieces placed before the game ends – are actually NP-Complete [2]. Part of the reason is that the state space of Tetris is very large and quite complex. First let us assume that each state consists of a given Tetris board and a current piece to be placed. If each next piece is random, we have a choice of 7 pieces, multiplied by a factor of 20 (an upper bound on the number of ways to place a piece on a board). Though the number of states is finite, it is difficult to estimate an effective lookahead depth. If we adopt the time constraint of a typical Tetris competitive round, and assume that we drop pieces with an average frequency of about 200 ms, we would then drop around 600 pieces throughout the span of a game. This gives us an upper bound of 140^{600} for the state space.

We can reduce this number figure if we factor in additional factors such as piece predictability (according to the random bag algorithm), line clearing, and others. However, we still end up with a relatively large space, and thus playing Tetris well is a non-trivial problem worth exploring.

Related Work

Quite some work has been done in developing a Tetris-playing AI, but few focused on the competitive version of the game, which is defined by a different scoring scheme regarding garbage lines, t-spins, combinations, and perfect clear. Three existing implementations seem to stand out.

The first by Stevens and Pradhan employs a deep reinforcement learning algorithm: a convolutional neural network is used to approximate a Q function indicating the next best move from a state [4]. Each state is represented as a 20 x 10 image, and the action to transition from one state to the next involves moving/rotating the current piece. During training, the Tetris agent adopts a standard epsilon-greedy policy: It will choose the optimal action most of the time (calculated according to the Bellman equation) but will also pick a random action with probability epsilon. Tuning epsilon hence naturally provides a trade-off between exploration and exploitation. This implementation runs on a non-competitive version of Tetris built in Python called MaTris, utilizing number of lines cleared as rewards. We do not have time to implement a neural network-based reinforcement learner, but building one in a competitive Tetris context would be one possible future work.

Cai, Zhang and Nebel proposed a different strategy that involves Monte Carlo tree search [5]. Noting the high

branching factor of the state space, they explore the tree using Upper Confidence Bound applied to Trees (UCT). Under this scheme, they conducted playouts for a selected number of children of the current states, then scored them according to their rewards. They then picked the child with the highest upper bound as the next move. In this way, we do not need to consider every single child specified by the branching factor. We do not implement a Monte Carlo tree search strategy, but this is one possible improvement for our algorithm. Additionally, Cai et al. also used some good heuristic criteria, such as number of holes in the built structure and the number of rows cleared simultaneously, which inspires our heuristic ranking.

Finally, "The (Near) Perfect Bot" created by YiYuan Lee was able to play the single player Tetris version for over 2 weeks (in other words, it did not get into a losing position after 2 weeks) [6]. The backbone of this AI program was structured by 4 heuristics that determine the best next move in a tree search algorithm. Those 4 heuristics include: aggregate height, complete lines, holes, and bumpiness. The aggregate height heuristic tries to minimize the height of the built structure. Complete lines tries to maximize the number of cleared lines. Holes heuristic tries to minimize the number of holes in the built structure. Finally, bumpiness tries to minimize the sum of absolute differences between two adjacent columns. The optimal weights of four heuristics were determined by a genetic algorithm. Though we are not using a genetic algorithm, we borrowed the four heuristics as parts of our ranking system.

Formulation

Our implementation of the Tetris game, on which our AI will play, adheres to the official Tetris guideline [1]. New Tetris pieces are spawned via a random bag: The pieces are generated in aggregate groups of seven (the bag) that is then shuffled and dealt in random order. Additionally, players can hold pieces - saving them for later - if the current piece is not desirable. Players can only hold one piece at a time, and if there is already a piece being held, that piece and the current piece the player is controlling will swap. We also add a popular feature in newer versions of Tetris called "the t-spin" – a complicated move which involves rotating the T piece just as it is about to land, allowing it to fill up spaces not normally reachable [2]. In most, if not all, modern versions of the game, t-spins are rewarded even more handsomely than regular line clearing. The notion of combinations is also heavily rewarded: immediately after clearing a line (or a few lines), players may gain extra points by clearing another set of lines using the immediate next piece. Successive cleared lines are known as a chain, or more informally, a combo.

We will consider a popular competitive version of Tetris. This version has a 2-minute time limit and incorporates the concept of garbage lines: each time a line is cleared, we send garbage lines to the other player, and the higher the chain of clears the more garbage lines sent. These appear as gray horizontal lines that shift the opponents structure up, pushing them closer to losing. Most versions of the game will leave gaps amongst the garbage lines, making them clearable.

Methods

We implemented a tree search algorithm such that given a board, a held piece, a current piece, and a queue of next 5 pieces, we generate and maintain a dynamic search tree. The 3 primary functions of the tree search are: to create the tree, to select the next best move, and to update the tree.

In creating the tree, it is quite intuitive to format tiling tetriminos as a search problem. Each state is a board configuration with a certain number of pieces already in place. A transition function takes a new piece, an existing board, and a sequence of moves, and output a new board state with the new piece at a position specified by the moves. Since we can move and rotate a given new piece in different ways, we produce a collection of different move sequences; we thus end up with different branches corresponding to the new piece being placed at different locations. Since we also incorporate the ability for the player to hold a single piece at a time for future use, the current new piece can be swapped with this held piece. Thus, we need to consider child board configurations for the held piece too.

In selecting the best next move, give an board state, we look ahead two layers of nodes and apply a ranking on each individual leaf using multiple weighted heuristics. The heuristics include the number of holes, the depth of each hole, the number of clumped holes, the average height, and the bumpiness of the given board. With the rankings of each leaf node on the tree, the optimal next move will be in the direction of the highest ranked leaf node.

Once the tree has selected the next best move, the tree must be updated to contain all of the next moves and prune all nodes that are not applicable to the new board state. To do so, we set the chosen child as the new root node, prune all of its siblings (as well as all the nodes that were successors of its siblings), and create the children of all remaining leaf nodes.

We used Python 3 to create and run the AI. To render our custom implementation of the game Tetris according to the official rules, we used the Pygame (`https://www.pygame.org/news`) library. While using a

Heuristic	Average Pieces Dropped	Average Lines Sent
Random	22	0
Average Height	13	0
Bumpiness	38	2
Hole Clump	85	19
Hole Depth	121	28
Tetris	600	254
Normal	30	3
Combo	95	23
Combined	83	22

Figure 1: Heuristic Comparisons

premade implementation of Tetris might have been easier, we chose to implement our own in order to have more control and flexibility in making design choices for the AI. We were quite successful in implementing the api and are confident it can be used as a standalone package for other Tetris projects. The api comes with an object representation for Tetriminos, the Tetris board, a game controller, and the point system. In addition, we provide an implementation of the Tetris game (with graphics).

Our primary design philosophy was modularity. At the center of the design is the Tree node, a class which represents a state of the Tetris board. Each node uses a transition algorithm to generate successor states and make new nodes from those states. We use recursive methods to control the growth of the tree and rank and retrieve the best terminal states. The search tree is also programmed with adjustable depth to mitigate memory issues.

The Ranker class provides different heuristics to rank each state. Different heuristics prioritize different properties of each state and use different strategies to evaluate a given board position. Combinations of these heuristics also provide interesting results as we discuss in the next section.

Results

We ran our tree search algorithm using different heuristic ranking strategies, then compared their effectiveness based on the number of lines sent. We conducted 10 runs of each scenarios, then took the average. Each trial was given a piece limit (600). Trials terminated when the piece limit was reached or the game ended because the built structure reached the top of the board. We limit each trial with a fixed number of pieces because the speed of the AIs performance varies on different systems.

We first consider the individual heuristic, then we test out some pre-determined strategies that combine a selected number of heuristics by taking their weighted sum. We provide a brief description for the evaluation strategies behind each heuristic. Note that we weight each state rank with 40% and the number of lines that state cleared with 60%.

Individual Heuristics:

Average Height: we eliminate the lowest column height and take the average of the remaining heights

Bumpiness: we take the difference in height between adjacent columns

Hole Clump: we ran BFS on each hole and merged adjacent holes into one

Hole Depth: we calculate the total difference in height of each hole to the actual height of the column

Aggregate Heuristics:

Tetris Rank: a strategy to maximize the number of times 4 lines are cleared at the same time. In competitive Tetris, this awards more points, and is called a Tetris (hence the name). To achieve this, we combine the average height, hole clump, and bumpiness heuristics, giving more weight to the hole clump heuristic.

Combo Rank: a strategy to maximize the longest combo. A combo occurs when a player clear lines after lines using successively spawned pieces. For this, we combine the number of holes, hole depth, bumps, and average height heuristics.

Normal Rank: a strategy where we combine all four heuristics with equal weights.

Combined Rank: for each leaf node, we ran the previous three ranking strategies, and assign the node with the highest rank output.

As we see in Fig. 1, Hole Depth works surprisingly well. Tetris Rank does not utilize this heuristic though. We observe that most heuristics consistently outperform random piece placement. However, no heuristic reached the piece limit except for the Tetris Rank strategy.

Future Work

An extension of our project could be to use reinforcement learning. The "near" perfect AI Tetris bot[6] played was implemented using reinforcement learning, and similar techniques may enhance the performance of our Tetris AI. Another extension would be to use genetic algorithms to find optimal weights of each game strategy heuristic. The AI right now weights the heuristics based on hand written rules constructed with our human knowledge based on hundreds of hours of combined play time. By learning the optimal heuristic weights on its own, we might see more interesting and consistent results.

A future direction to test the skill of this AI is to play against other humans on platforms like Tetris Friends or Tetris Online Poland. We decided to work on It would be interesting to observe exactly how our AI program compares to against other players in terms of performance.

We would also like to implement the AI with a much prettier graphic representation.

References

1. "Tetris Guideline." Tetris Wiki, 29 June 2009, https://tetris.wiki/Tetris_Guideline.

2. Shaver, Morgan. "How to Perform a T-Spin in Tetris." Tetris - Tetris Holding, 4 August 2017, https://tetris.com/article/70/how-to-perform-a-t-spin-in-tetris.

3. Demaine, Erik; Hohenberger, Susan; Liben-Nowell, David. Tetris is Hard, Even to Approximate. arXiv, 21 October 2002. https://arxiv.org/abs/cs/0210020.

4. Stevens, Matt; Pradhan, Sabeek. Playing Tetris with Deep Reinforcement Learning. Stanford University Press, 2016.

5. Cai, Zhongjie; Zhang, Dapeng; Nebel; Bernhard. Playing Tetris Using Bandit-Based Monte-Carlo Planning. University of Freiburg, 2010.

6. Lee, Yiyuan. Tetris AI - The (Near) Perfect Bot. Code My Road, 14 April 2013. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/.