AI Project Report

Hy Nguyen and Emery Morua

May 2021

1 Abstract

This study attempts to use artificial intelligence in order to play Tetris. The success of the AI, in a game that theoretically could go on forever, was measured in terms of number of rows destroyed. A 7-bag modern approach to Tetris was used to generate the pieces.

2 Introduction

2.1 Tetris as a Non-trivial Game

Tetris is a game created in 1984 that involves tile matching "blocks" consisting of different orientations of consecutive cells. These blocks are referred to as tetrominoes, or shapes composed of four squares (cells) connected orthogonally. The tetrominoes are dropped, one at a time, on a 10 x 20 grid space and players are expected to tile match pieces together. Once a row of ten cells are filled with the squares composing the tetrominoes, those squares are deleted and squares above said row move down. Players lose by creating filling so many rows that the new blocks have no place to generate and the game is over; this is referred to as a block out. The objective of the game, since it can theoretically go on infinitely, is to get the highest score, which is typically calculated by how many rows were destroyed. The game poses a complex problem for artificial intelligence (even for a single-player game) as there are many possible combinations of pieces and many places and orientations that a new piece can be placed. Demaine, Susan Hohenberger and David Liben-Nowell of the Massachusetts Institute of Technology determined that Tetris is an NP-hard problem, which means it is as least as difficult to solve as any other NP problem. An algorithm needs to be fast enough to perceive the best move for a given piece in a given state space.

2.2 7-Bag - A Modern Approach

Older versions of Tetris generated the next piece randomly. In this study, we use a modern approach to tetromino piece generation called the 7-bag approach. The 7-bag approach essentially generates pieces by creating a set of each of the pieces and then randomly takes a piece to generate until the set is empty, the process starts over. This ensures that the same piece will generate again in, at maximum, 13 moves.

2.3 Background

This paper was strongly influenced by the work of Max Bergmark on his Bachelors Thesis "Tetris: A Heuristic Study". Bergmark's AI calculates the number of holes on a given grid using three rules that will be described in the Algorithms section.

We applied the same three rules to our grid of a different size (described in detail in section 3). “‘

3 Approach

3.1 Representation

3.1.1 Grid

Tetris can be represented using a grid. This grid is implemented using a two-dimensional array. Non-occupied cells of the grid are denoted as an empty string as follows: grid[rowIndex][columnIndex] = ”. Whereas, occupied cells are given a unique id; for example, grid[rowIndex][columnIndex] = ’a1’. A unique id is assigned to a cell so that when the repr() function is called to print a representation of the Tetris game state, one can easily identify the cells that a block occupies. In the actual Tetris game, usually the J-block is given a unique color and the S-block is given a different unique color, and so on. However, the downside to this is that we cannot easily identify two J-blocks contiguous to one another because they are of the same color which makes testing the correctness of our Tetris game more difficult. Hence, this is why I chose to use unique ids. I guarantee that there will be no two blocks in the grid that have the same id by having a dictionary of 234 ids which is more than the necessary 200 ids for a 20 row by 10 column grid (note: we need at least 200 ids because due to rows being destroyed, every block may be made up of only one cell). However, if we wanted to increase the dimensions of our grid, we may need to add additional unique ids to the dictionary of ids.

3.1.2 Block

The seven different type of blocks (O, I, J, L, T, S, Z) all have different orientations depending on how each block is rotated. For example, the I-block has two orientations: laying down and standing up. Each of these orientations is represented by an array of (row index, column index) coordinates. The part of a block that is lowest would have a row index of zero. Similarly, the part of a block that is left-most would have a column index of zero. Thus, the O-block is represented as follows: [(0,0), (0,1), (1,0), (1,1)]. The I-block laying down is [(0,0), (0,1), (0,2), (0,3)]; the I-block standing up is [(0,0), (1,0), (2,0), (3,0)].

It is important to note that not all orientations of all shapes have (0,0) as a coordinate. An example where (0,0) is not a coordinate is one orientation of the Z-block: [(0,1), (0,2), (1,0), (1,1)].

3.1.3 Additional Data Structures

Additionally, I chose to create a class called Shape that has two attributes. Each block on the grid has a corresponding Shape object. The first attribute stores the block coordinates described in the section directly above. The second attribute is named the zero-zero-location and it stores the corresponding grid coordinates (row index, column index) where the block coordinate is (0,0). This allows one to easily find all the grid coordinates of a block by summing the zero-zero-location's row index with the block coordinates row index (and doing the same for column). This approach avoids scanning the grid which would increase the runtime. The other alternative is to use breadth-first search or depth-first search to find all the grid coordinates of a block but this is error prone and complicates the logic of the code.

3.2 Algorithms

3.2.1 Destroying a row

In Tetris, after a block is placed rows that are fully occupied are destroyed. Once a row is destroyed, all the grid cells above the deleted row shift down one row. Note, anywhere between zero and four rows may be destroyed from placing one block. Our code checks if a row needs to be destroyed by scanning each row to see if it is fully occupied. If the row is fully occupied, all the cells in the grid above the deleted row get shifted down by one row. We delete a single occupied row repeatedly until there are no more rows to delete. The Shape data structure needs to also be updated. We keep the zero-zero-location the same and update subtract one from the row index for every block coordinate to a corresponding cell that got shifted down one row.

3.2.2 Block placement combinations

The 7-bag approach will give us the block we need to place. It is up to the AI to decide what orientation (rotation) of the block to use and at what column index to place that orientation of the block. For example, an I-block has two orientations. In a grid with ten columns, the I-block can be placed in seven different columns laying down and ten different columns standing up. This means that the I-block has seventeen different combinations of where it can be placed. Our code first iterates over every orientation and for every orientation it

iterates over every column it may be placed in so long as it is within the bounds of the Tetris grid.

3.2.3 Heuristic: Three rules

There are three rules that each detect a different class of holes in a Tetris grid. The rules are as follows:

1. Empty cell to the left of a column where the top most cell in that column is the same height or above the empty cell.
2. Empty cell under a topmost filled cell.
3. Empty cell to the right of a column where the top most cell in that column is the same height or above the empty cell

Rule one states an unoccupied cell is a hole in column j if it is at or below the highest occupied cell in column j+1. Rule three is very similar but instead of looking to the right, we look left. It states an unoccupied cell is a hole in column j if it is at or below the highest occupied cell in column j-1. Finally, rule two states an unoccupied cell in column j is a hole if there is a higher cell in the same column j that is occupied.

3.2.4 Weighting the three rules

Without any weights the three rules would just give unit weight of 1 for each hole found. However, we may want to weight rule two higher if we thought a rule two holes were less desirable than rule one and rule two holes. For example, we may weight a rule one and rule three hole as 1 and a rule two hole as y, where y is the row index (assuming 1-based indexing) of the hole. In this case, rule one and rule three holes are weighted as g(y) = 1 and rule two holes are weighted as f(y) = y.

3.2.5 Putting everything together

Having discussed the block placement combinations, three rules, and weighting of the three rules, we can now discuss the mechanics of this AI. We first choose our weights for each of the three rules. Our goal is to figure out the best block placement combination (orientation and column index). For every block placement combination we calculate the sum of the three rules, where each rule weights its holes based off the prepicked weights. The block placement combination that has the minimum sum of the three rules is the one we choose. We then repeat the same process by placing the next generated block using the 7-bag randomizer.

3.2.6 Better algorithm

The weakness of the algorithm described in the "Putting everything together" section is that it is a greedy algorithm that does not consider future game states. An analogy to this is the a* algorithm which has a g(n) component of the current game state and a h(n) component of all the combinations of future game states. Our current algorithm only has a g(n) component. To add a h(n) component we need to consider the best block placement combination by considering all the blocks that can be generated next (since we use the 7-bag randomizer this is not necessarily all the seven blocks) and all the corresponding block placement combinations. Essentially, instead of looking at just the placement of the current block, we consider a placement of the current block affects the placement of the next block. This is called the one step prediction approach. There is also a two step prediction approach that considers the placement of the next two blocks. Furthermore, there is also an n step prediction approach that considers the placement of the next n blocks. However, when considering steps ahead the complexity and runtime increase dramatically.

4 Analysis

In order to gauge the effectiveness of our AI, we let it play 200 games for three different times with the following weight functions (SECTION) for each trial. We played on a 16 row by 10 column board instead of a 20 row by 10 column board which decreases the number of rows destroyed thus reducing the amount of time required to play 200 games.

1. $f(y) = 1$, $g(y) = 1$ (unweighted)
2. $f(y) = y$, $g(y) = y^1$
3. $f(y) = y^3$, $g(y) = y^2$

4.1 AI Performance With $f(y) = 1 \, and \, g(y) = 1$

The mean amount of destroyed rows for these trials was 87.72 with a standard deviation of 54.05. Not surprisingly, the unweighted algorithm yielded the least amount of rows destroyed on average. It's important to note, also, that the standard deviation, is relatively high to its mean, which is observed in all three trials; our algorithm is consistently inconsistent.

4.2 AI Performance Using $f(y) = y$, $and \, g(y) = 1$

The mean amount of destroyed rows for these trials was 2710.88 with a standard deviation of 2592.10. Clearly there was a lot of variation

between matches. the AI was able, in one game, to destroy 15289 (the maximum of our data set) rows before reaching a block out, but, in another game, the AI only destroyed 25 (the minimum of our data set). However, outliers this low were very rare; in fact, it was not very often that the algorithm would destroy less than 100 rows before reaching a block out. Across the board these functions for f(y) and g(y) led to the best performance of out algorithm.

4.3 AI Performance With $f(y) = y^3 \, and \, g(y) = y^2$

The mean amount of destroyed rows for these trials was 172.28 with a standard deviation of 151.53. As with the trial using$f(y) = y, and g(y) = 1$**, the we made f(y) have a polynomial degree of n and g(y) a degree of n - 1. We expected that this would allow the AI to better decide what was a good move and what was a bad move for the future state space at the cost of an increase in runtime; however, this was clearly not the case. In fact, this trial was the only trial to score below 20 destroyed rows before reaching a block out, and it did this more than once. In fact, the minimum number of rows destroyed before blockout for this trial was 6.**

4.4 One step prediction

We implemented the one step prediction approach, however, the runtime increases by a factor of roughly 162. This made it impractical to run the one step approach for 200 games. Moving forward, we could possibly write a more efficient algorithm to make a one step prediction approach more practical.

5 Conclusion

This paper discusses a heuristic approach to the classic game of Tetris. With the objective of destroying the most amount rows possible before reaching a block out, we devised three rules to find different holes for our algorithm to decide the best block placement

1. Empty cell to the left of a column where the top most cell in that column is the same height or above the empty cell.
2. Empty cell under a topmost filled cell.
3. Empty cell to the right of a column where the top most cell in that column is the same height or above the empty cell

We also weighted to favor rule two by using two functions f(y) and g(y). For three trials, defined said functions the following ways

1. $f(y) = 1$, $g(y) = 1$ (unweighted)
2. $f(y) = y$, $g(y) = y^1$
3. $f(y) = y^3$, $g(y) = y^2$

We then ran 200 games for each trial and tabulated the number of rows destroyed before a block out was reached. It was found that trial 2 performed much better than the other trials, even though trial three shared the difference of polynomial degree between f(y) and g(y). It was expected that trial 3 would yield the best results at the cost of runtime of thee trial, but this was not the case.

This problem has potential for much more complexity. For instance, instead of doing BFS with one step predictions, we could instead go forward two steps in order to get better predictions about the optimal path of a future game board. We could also alter functions f(y) and g(y) and change the difference between their polynomial degree.

References

[Bergmark, 2015] Bergmark, M. (2015). Tetris: A heuristic study : Using height-based weighing functions and breadth-first search heuristics for playing tetris.

[Bergmark, 2015]