Tetris AI
Learning using Parallel Genetic Algorithm

Team 18
Lai Cheng Yu
Tan Zheng Wei
Ten Chee Onn David Jonathan
Janik Geerken

Abstract

This report describes the implementation of our Tetris AI. We will be discussing the heuristics used to evaluate each state, followed by the search algorithm used to determine the best move to take next. A Genetic algorithm is used to train and find the optimal weight vector for our weighted heuristics, augmented with parallelization to reduce the time taken and improve the reliability of our results. Additionally, the tetris game is played based on a purely randomized generation of pieces. The following is a link to code on GitHub: https://github.com/zhengwei143/cs3243-project.

1. About the Heuristic

The following section describes the heuristics the Tetris AI uses to make decisions

1.1 Holes

A hole is a free square that has at least one filled square above it. As Figure 1 shows, having a hole makes a row more difficult to clear because we will have to clear all the rows above it before we can fill the hole. As such, we want to **minimize** the number of holes on the board.

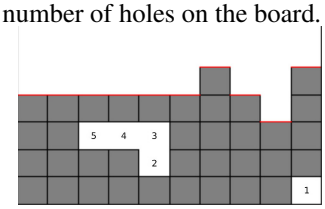

Figure 1

1.2 Complete Lines

Complete Lines is the number of completed lines in the game board. Completing a line removes that line from the game board, and increases the player score. Clearing lines also gives us more space for more game pieces. As such, we want to **maximize** the number of completed lines. Figure 2 shows 2 lines (in green) that will be cleared after the block (in red) is placed.

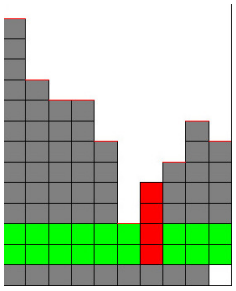

Figure 2

1.3 Well sums

A well is defined as having consecutive empty spaces in a column with squares on either side of them being filled.

$$h_n = min(height_{n-1}, height_{n+1})$$
$$WellHeight(col_n) = h_n * (h_n + 1)/2$$
$$WellSum = \sum_{n=1}^{numCols} WellHeight(col_n)$$

WellHeight is the deepness of the well, that increases in a quadratic manner, so as to penalize deeper wells in a greater fashion. This is because having larger wells increases the potential that it will get covered, creating a large hole. As such, we want to **minimize** the number and magnitude of such wells. Figure 3 shows a large well at the right most side of the board, and that is something we try to avoid.

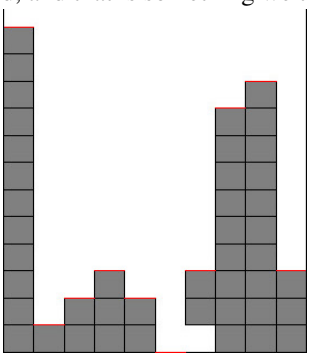

Figure 3

1.4 Bumpiness

A bump is present when there is a variation in column height between 2 columns.

$$bumpiness = \sum_{n=1}^{numCols-1} |height(col_n) - height(col_{n+1})|$$

Significant presence of bumps could be bad because a large bump would likely signal that there are columns that are too high and others that are too low. It would be unlikely that we would be able to clear the rows that are high anytime soon. Furthermore, a low bumpiness score signals that the grid is about evenly filled with objects making it easier to clear lines and get points. Therefore, we want to minimize the bumpiness score.

1.5 Aggregate Height

$$AggregateHeight = \sum_{n=1}^{numCols} height(col_n)$$

The aggregate height tells us how high the objects are. The higher this score, the less space we have between our pieces and the top of the grid. We want to minimize this score so we have more space to drop objects, and also because hitting the top of the grid ends the game.

1.6 Blockades (Unused)

A blockade is any block that is above a hole. A blockade is bad because putting more blocks on top of a hole would make it harder to remove the hole. While not having any holes would be an optimal solution, sometimes one or two holes are inevitable. When that happens, we want to clear away the hole as soon as possible.

As such, we want to minimize the blockade score. This heuristic ultimately went unused as the presence of it did not help improve the overall score. We found that severely punishing holes was overall a more dominant and better heuristic than including the blockade score. Figure 5 shows a blockade on the left-most column with a score of 6.

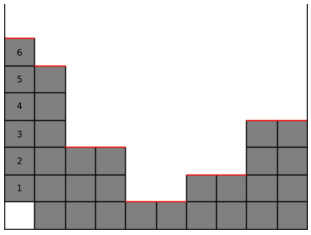

Figure 4

1.7 Combining the Heuristics

These heuristics are used to determine how good a certain move is by evaluating the resulting state of the blocks following the respective move. We approximate how good a state is through a linear weighted sum of the heuristics as such:

$$score = \sum_{i}^{heuristics} h_i * w_i$$

The higher the score, the better the state. This gives us the basis for us to compare different moves and decide on which move to choose, which will be further discussed in the subsequent section.

2. Search Algorithm

The search algorithm that the Tetris AI employs to search through the solution space is depth-two expectimax search.

Expectimax search is utilised when we do not know the result of an action in advance, which is perfect for Tetris as it is too computationally expensive to determine the true utility of every move by simulating the game to its end supposing the move is made.

As such, to better manage the real-time playing aspect of this AI, it uses depth-limited expectimax search of depth two instead, setting each chance node after the root node to be of equal probability and the expected utility of each chance node to be the weighted average of values of its children (the state score in this case).

The underlying problems are formalized as Markov Decision Processes. The implementation details of our depth-two expectimax search are as follows:

1. Suppose that the next given piece is an I-piece. Then, for every possible orient (o_1 and o_2 for vertical and horizontal respectively) and position ($p_1...p_{10}$ for o_1, $p_1...p_7$ for o_2) that the I-piece can be placed onto the field, we will have states $s_{1\times1}...s_{1\times10}$,$s_{2\times1}...s_{2\times7}$. The AI then estimates the true expectimax value for each state.
2. To obtain the estimated expectimax value of each state $s_{o\times p}$, the AI first exhaustively enumerates all states generated from all possible pieces ($piece_1...piece_7$) in all possible orients ($o'_1...o'_i$) and valid positions ($p'_1...p'_j$) that can be reached from $s_{o\times p}$. We will call these states $s'_{1\times1\times1}...s'_{7\times i\times j}$.
3. For each $piece_k$ from $piece_1$ to $piece_7$, the AI searches for the optimal o'_m and p'_n such that the evaluated score of state $s'_{k\times m\times n}$ is the maximum amongst all possible values of m and n. The AI then finds the average of these 7 optimal scores and assigns it as the estimated expectimax value of $s_{o\times p}$.
4. Once the estimated expectimax value for every state $s_{1\times1}...s_{2\times7}$ has been computed, the AI then makes the corresponding move of o_{max} and p_{max} on the I-piece to reach the state $s_{o_{\text{max}}\times p_{\text{max}}}$ with the maximum expectimax value.

This allows the AI to make a one-step prediction into the future, making the search algorithm less greedy.

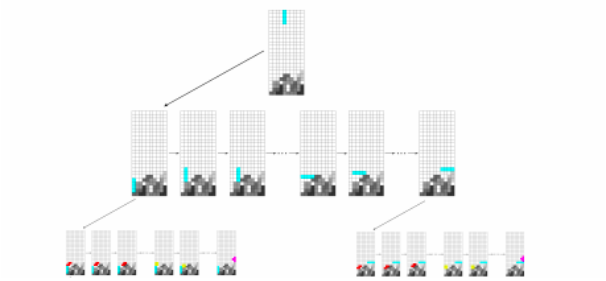

Figure 5: To find the best placement for the current tetrimino, the AI evaluates every possible board state for the current tetrimino and every possible next piece. [1]

Calculating the board state score for every possible placement for two pieces (2^{nd} piece unknown) is much more computationally-intensive as compared to only calculating for the given next piece, slowing the algorithm down by a factor of about 162 [1]. However, the AI is still able to make its decision quickly, in less than 1ms [1].

3. Genetic Algorithm

The following section describes the use a Genetic Algorithm in order to train and find the optimal weight vector to get the best score.

3.1 Initial Population

We start off with a population size of 1000 individuals. Each individual is representative of a weight vector:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix}$$

such that each w_i is a randomly generated value between 1 and -1. We calculate the fitness of the individual based on the number of lines cleared by playing the game with each respective weight vectors.

3.2 Sampling and Crossover

During each generation, we randomly select a sample of 10% of the population size and from there pick the 2 weight vectors with the highest fitness for crossover. Suppose we have two weight vectors:

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} \text{ and } v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}$$

the crossover between u and v creates a new (offspring) weight vector w based on the following formula:

$$w = u * fitness(u) + v * fitness(v)$$

Following which, the vector is normalized. The intuition here is to combine both vectors weighted by their respective fitness values so that it would favor the stronger vector.

3.3 Mutation

After each crossover, we introduce the element of mutation that could happen to each newly created (offspring) weight vector. The chance for mutation is set at 5%, which then randomly selects a weight attribute and adds a value between $\pm$ 0.2. The mutated vector is then normalized again.

3.4 Completion of a Generation

Sampling, crossing of vectors and mutations are carried out repeatedly until we have generated 30% of the initial population. From here, we will remove the vectors with the lowest fitness until the population returns to its original size. This cycle of creating and subsequently removing vectors from the population constitutes a single generation. By repeatedly iterating through the generations, the implementation ensures that the overall fitness of the vectors are non-decreasing.

4. Parallelization

4.1 Multi-Threading Game Runs

The initial implementation to calculate fitness of a weight vector was to run a single game and return the number of lines cleared. However, we found that the fitness of these supposedly "good" weight vectors were unreliable and inconsistent in producing good results over multiple runs. Thus, in order to ensure the consistency and accuracy of our results, whilst not forsaking computational efficiency, the calculation of each fitness value was implemented by running 10 games concurrently in multiple threads and subsequently returning the average score.

4.2 Constraints

However, despite implementing multi-threaded games, we still found that the genetic algorithm took a long time to iterate over its generations. Thus, we added another constraint - to limit the number of rows on the field to 11 instead of the original 21. We found that this restriction made it such that games ended much faster, because it is much easier to hit the top. On the other hand, a surprising outcome of this was that, whilst the average fitness of these weight vectors were much lower, playing the game with these weight vectors without the constraint showed an increase in score by 4 orders of magnitude. For example:

With constraint: $fitness(w) = 5460.1$

Without constraint: $fitness(w) = 20M+$

We believe that the constraint imposes a bias towards weight vectors that keep the tetris board as low as possible, which significantly decreased the likelihood of losing.

4.3 Speedup

Running our genetic algorithm sequentially reports an initial population creation time of 1000 vectors at around 45.0 minutes. When running concurrently with 10 threads (to obtain the average fitness of a weight vector over 10 runs), we attained a speedup of around 2.5 times to 18.3 minutes. However, the amount of speedup increases exponentially with more generations being created. This is because the population of each generation gets stronger which results in the runtime increasing for every single run of each weight vector.

5. Results

Over the course of the various runs of our Genetic Algorithm, the best weight vector we found is the following:

$$w = \begin{bmatrix} 0.1636736030816534 \\ -0.11117594223369093 \\ -0.20390418721234355 \\ -0.9501423421384158 \\ -0.12846584618379997 \end{bmatrix}$$

which has completed $20M+$ lines as shown below at the time of writing this report.

```
e0015828@xcnc26:~$ tail /temp/zhengwei/5h-gen1/nohup.out
You have completed 20362539 rows.
You have completed 20362539 rows.
You have completed 20362539 rows.
You have completed 20362539 rows.
You have completed 20362539 rows.
You have completed 20362541 rows.
You have completed 20362541 rows.
You have completed 20362542 rows.
You have completed 20362543 rows.
You have completed 20362543 rows.
```

6. Conclusion

Through the course of this project, we have experienced first-hand the different components in building an actual AI, from heuristics and searching, to the training of the AI. Firstly, we found that choosing the right heuristics plays an important role, and it is not just about the number of heuristics we implement. We started off with 4 heuristics and added 2 heuristics (well sums and blockades), however we after the removal of the blockades heuristic, we found significantly better scores.

Secondly, we underestimated the time it actually takes to train and find the optimal weights for to evaluate our heuristics with. While the genetic algorithm may be optimized for efficiency, it still takes significantly long to churn out new generations, especially so when the overall fitness is increasing (or is already high) - then games would take much longer.

Thirdly, we acknowledged the influence of luck in not just the playing of the game (in terms of the sequences of pieces generated), but also in the training of the weights in the genetic algorithm. It is not uncommon for a single weight vector to see a big difference in scores between different games, and we took this into account by approximating the fitness with an average. Furthermore, as there are 5 heuristics, the corresponding vector space is 5-dimensional and thus, even randomly generating 1000 weight vectors would hardly be sufficient to take the whole vector space into account.

Lastly, we found that constraining the number of rows on the grid to 11 (down from 21) not just significantly reduced the time it takes to train the weights, but also put a bias on weight vectors that perform better on 11 rows, which by extension, performed significantly better (at least 3 orders of magnitude) when playing on 21 rows.

References

[1] Max Bergmark. “Tetris: A Heuristic Study”. KTH Royal Institute of Technology, May 2015.

[2] Yiyuan Lee. “Tetris AI - The (Near) Perfect Bot”. In: (Apr. 2013). URL: https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/.