Learning to Play Tetris with Big Data (Team 25)

Goh Wei Wen	Kevin Leonardo Anantha	Li Hong Sheng Gabriel	Li Jiayao	Zhang Shuoyang
A0156085B	A0161302Y	A0104274N	A0160257J	A0158000X

April 21, 2018

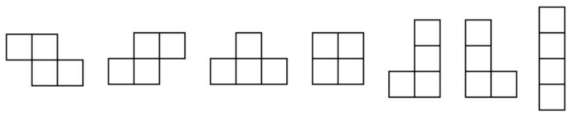
Abstract

Tetris is a popular computer game. The player is given a sequenece of randomized tetromino pieces one at a time and must pack them into a rectangular gameboard. Any completely filled row of the gameboard is cleared and all filled squares above it are shifted down by one row. We have designed an utility-based agent to play *Tetris* which is able to clear millions of rows by maximizing the number of rows cleared. We illustrate how the agent is able to achieve this with the help of the *genetic algorithm* designed by our team, and our game playing strategy, *expetimax*, which is used in optimally positioning a randomly given tetromino. Besides, we describe the method utilised to enable our agent to scale up to big data. Furthermore, we showcase the performance of the learning algorithm and its speedup on big data. Last but not least, we discuss the areas that we can further improve on to obtain a better result.

1 Introduction

Tetris is a popular computer game invented by mathematician Alexey Pazhitnov in the mid-1980s [1]. The goal of the game is to clear as many rows as possible. In this paper, we present our *genetic algorithm* that learns a set of weights for an utility function, combined with expectimax, which attempts to optimally position (by rotating 90° or 180° in any direction) a piece of tetromino on a *Tetris* grid. The learning process of this algorithm is detailed in section 2.2 and game playing strategy in 2.3. Empirical results, in section 3, show that our algorithm is able to learn a set of weights to play Tetris within a relatively short period of time, and it scales relatively well when run on big data.

The gameboard we used in the project is a 10 rows × 20 columns 2D grid. The 7 pieces of tetromino used in the game are shown below:



The game begins with a blank grid and ends when the last shape that is placed reaches the top of the grid. The **goal** of this game is to **maximize** the number of rows cleared.

2 Methodology

2.1 Features

The following 6 features that are used in our *genetic algorithm* were referenced from an article[2] with some modifications.

1. **Height difference**: the height difference between the highest column and the lowest column in the gamboard.

2. **Number of rows cleared**: number of rows whose cells have been completely filled (and thus removed from the grid).

3. **Row Transitions**: a row transition refers to the occurrence of a filled cell next to a hole in the same row.

4. **Column Transitions**: a column transition refers to the occurrence of a filled cell that is next to a hole in the same column.

5. **Number of Holes**: number of empty cells in the grid such that each cell has at least one filled cell on top of it in the same column.

6. **Number of Wells**: wells refer to groups of holes whose left and right cells have been completely filled.

Each of these features is assigned a **real-valued** weight, which is used by our utility function to estimate the optimality of a state. Let S be a state, w_n be the weight for the nth feature t_n, the utility of S can be represented by the following function:

$$f(S) = \sum_{n=1}^{6} w_n \times t_n \tag{1}$$

As mentioned in equation 1, we aim to maximize $f(S)$ for each random given tetromino piece.

2.2 Learning Method

To maximize the equation 1, we run our *genetic algorithm* to learn an optimal weight (w_n) for each feature (t_n). A high-level description of our genetic algorithm is outlined as below:

1. **Initialize population**: a population is a list of individuals. Each individual of the population is a $1 \times n$ weight matrix, where n is the number of features that we have chosen, as listed in 2.1. More formally, suppose that we have weights w_i, w_{i+1}, w_{i+2} ... w_n. Then, each individual can be represented as an n-tuple:

$$[w_1, w_2, ..., w_i, w_{i+1}, w_{i+2}, ..., w_n]$$

In our setting, the initial weights are generated randomly. Specifically, for features, height difference, row transitions, column transitions, number of holes and number of wells, the initial weights are a random *negative* number between 0 to 1. For feature, number of rows cleared, the initial weight is a random *positive* number between 0 to 5.

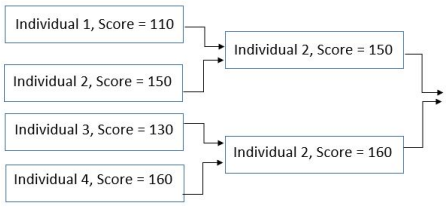
2. **Evaluate the initial population**: this is achieved by playing *Tetris* using each set of weights in the population. The evaluation score for each individual is the *total number of rows cleared* in the game. One point to note is that in order to fairly evaluate each individual, we evaluate each individual 5 times and take the average of these 5 scores as the final score for the individual.

3. **Tournament selection**: tournament selection is a robust selection mechanism that is commonly used in genetic algorithms [4]. In our setting, we utilize binary tournament selection, a simple mechanism to select a better individual among 2 randomly chosen individuals to produce the next generation. Therefore, a parent with a higher score is more likely to be chosen in *tournament selection*. Suppose the score for individual i is s_i. The population can now be seen as list of individual scores:

$$[s_1, s_2, s_3, ..., s_{n-2}, s_{n-1}, s_n]$$

In our setting, we first randomly choose 2 individuals from the top 50% of the population. Then, we select the better individual among these two to be a *parent*. In the same way, we select another individual

to make up a pair of *parents* to produce a child, which is a set of weights. This can be visualized in the partial tournament tree shown below:



The child's set of weights is obtained from its 2 parents, with weights randomly selected from each of the parent. By using *tournament selection*, the next generation is produced.

4. **Swap mutation**: in our setting, a classical mutation method, *swap mutation*, is utilized in the algorithm. *Swap mutation* is a simple O(1) operation that is easy to perform with small computation cost compared to other mutation methods, such as insertion, inversion, and displacement which changes all weights in the worst case [5]. To preserve the optimality of the *parents*, we will not explore the search space of weights too far away from the current maximum to avoid getting worse weights. Therefore in our setting, the mutation occurs when a probability following *Gaussian distribution* surpasses a certain threshold set by us.

5. **Result**: in our setting, we keep track of the highest evaluation score obtained and the corresponding set of weights over 25 generations, with a population size of 25 individuals.

2.3 Optimized game-play using expectimax

We used the expectimax algorithm with depth 1 using our utility function as a heuristic for improving how each individual in the population selects the next tetromino piece to play in the *Tetris* game (i.e. the number of rows cleared would be maximized by that move).

At each turn, all possible moves (orientations and positions) of the current piece are evaluated using a utility function. The utility function is a weighted sum of different features of the board after a move is played (refer to equation 1). The actual weight for each feature is determined using our *genetic algorithm*.

After a utility function with the most optimal value for each weight is generated, we use the expectimax algorithm to select the next optimal move for each individual of the population. For each legal move, the algorithm looks ahead one piece and calculates the weighted average (using the piece history as weights) of the utility of the board after the best move is selected. Assuming the piece history accurately represents the piece frequency, the algorithm will choose the move with the highest expected utility.

In contrast to minimax, this strategy does not assume that the adversary will pick the most optimal move. Thus, we cannot prune less-optimal branches, resulting in expectimax being significantly slower than minimax. However, similar to Rodgers and Levin (2014) [3], we achieved better results when using expectimax over minimax for *Tetris*. This can be attributed to the fact that we use our additional knowledge of the adversary (the fact that it selects i moves randomly) to our advantage instead of assuming the worst case.

2.4 Parallelism

As we perform *genetic algorithm* to select an optimal set of weights, each individual set of weights has to be sequentially evaluated by running *Tetris* to get its evaluation score. This would potentially incur a huge amount of time, if a large size of population and generations are set. Therefore sequential evaluation of individuals will not being able to scale to big data. To

speed up the entire evaluation process, we utilize Java 8 *parallel computing*, which divids a problem into several subproblems, and solve these subproblems in parallel [6]. With parallelism, we run our evaluation process for each individual in the population concurrently and efficiently. The comparison of parallel and sequential programming is analysed in section 3.2, Figure 2.

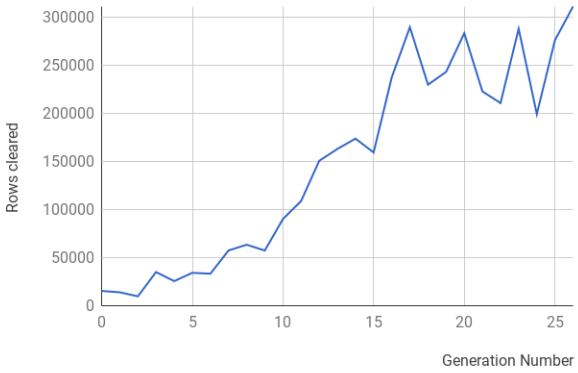


Figure 1: Number of rows cleared for each generation

3 Performance and Statistics

3.1 Score and Weights

The maximum score that our genetic algorithm obtained (without using expectimax for selecting moves) was 311143. The following are the weights used to generate this score:

Height Difference	-0.08382226928735714
Number of Rows Cleared	2.299698319650534
Row Transitions	-0.5409023035972038
Column Transitions	-0.5779452238352727
Number of Holes	-0.7237267127345023
Number of Wells	-0.11110083421382877

Using these exact set of weights and the expectimax algorithm to select moves during gameplay, our algorithm managed to attain a score of approximately **5 million** within only 9 hours, on a desktop computer with a quad-core Intel Core i7 3.4GHz processor.

3.2 Rate of Learning

The display of results seen in **Figure 1** shows that our algorithm results in well above 100,000 rows cleared, after 10^{th} generation. Moreover, from 15^{th} generation onwards, the algorithm is able to reach above 200,000 in rows cleared, with a high rate of increase.

These results indicate that after a few runs of the genetic algorithm, it can play the game quite well. In other words, our algorithm is able to learn a good set of weights within a relatively small number of generations.

However, after 20 or more generations, our algorithm is unable to improve its learning on the set of weights, as there is no significant increase in the number of rows cleared.

Furthermore, since the AI agent and especially the genetic algorithm have a lot of operations to process, we decided to use parallel processing to speed it up to scale big data (as mentioned in section 2.4). **Figure 2** shows the comparison of time taken between parallel programming and sequential programming for the same generation number. Note that from the 10^{th} generation onwards, the time taken for sequential programming to spawn new generations is too long so we chose not to display it on the graph.

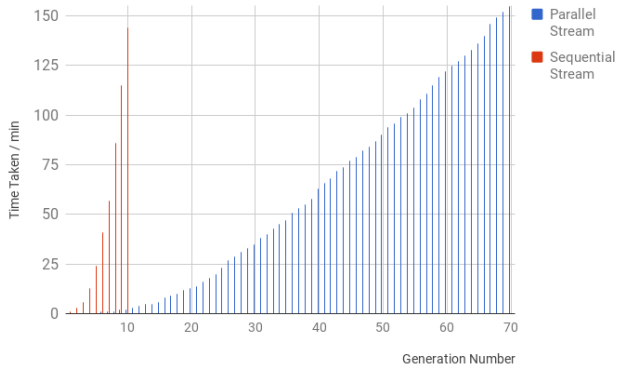


Figure 2: Time Taken for Sequential & Parallel

From the graph, it is clear that parallel programming takes much less time to complete each generation as compared to sequential programming. Hence, parallel programming is able to scale.

4 Future Improvements

As the analysis in section 3.2 shown, our genetic algorithm can demostrate a good learning rate from 0^{th} to around 16^{th} generation. Afterwards, the learning rate fluctuates. We may assume that at this point, the exploration of weights conducted by our algorithm in the seach space reaches a local optimal. Therefore more exploitation of the search space of weights is needed. Considering *genetic algorithm* might not be the best method of learning the optimal value for the weights. We tested on *particle swarm optimization* (PSO), a population based stochastic optimization technique [7]. However, in our experiment, the number of rows cleared with PSO is not ideal, with an average of 5,000. Therefore, a further research on how to improve the learning rate after reaching a local optimal should be conducted.

Furthermore, our research on the Internet turned up a lot of articles on AI *Tetris* agents that used different sets of features with varying degrees of success. Perhaps, we could have experimented with a larger variation of those features. This might help our genetic algorithm to learn a reasonably optimal set of weight values even faster.

5 Conclusion

The *genetic algorithm* presented in this report has been used to learn a set of metrics/weights that is used to maximise the utility output by our utility function. This utility is an estimation of how good a state would be. By comparing the utility scores, *expectimax* is able to determine the optimal move (position and orientation) for a random given tetromino piece. Furthermore, using these set of weights, the expectimax algorithm, used for game playing, has been shown to be able to handle at least 5 million tetris shapes (possibly more given more computing power and time). Additionally, parallel programming scales in the learning of weights.

References

[1] About Tetris. (n.d.). Retrieved from https://tetris.com/about-us

[2] Kamel, M. A. (2011, June 1). El-Tetris - An Improvement on Pierre Dellacheries Algorithm. *imake*. Retrieved from http://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/

[3] Rodgers, P., & Levin, J. (2014). An investigation into 2048 AI strategies. *IEEE Digital Library*. Retrieved from https://ieeexplore.ieee.org/abstract/document/6932920/

[4] Miller, B. L., & Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex systems, 9(3), 193-212*. Retrieved from https://pdfs.semanticscholar.org/df6e/e94e2cf14c38e9cff4d2446a50db0aedd4ca.pdf

[5] Liu, C., & Kroll, A. (2016). On the performance of different mutation operators of a subpopulation-based genetic algorithm for multi-robot task allocation problems. *arXiv preprint arXiv:1606.00601*. Retrieved from https://arxiv.org/abs/1606.00601

[6] The Java Tutorials. Retrieved from https://docs.oracle.com/javase/tutorial/collections/streams/parallelis

[7] Shi, Y., & Eberhart, R. C. (1999). Empirical study of particle swarm optimization. *IEEE Digital Library*. Retrieved from https://ieeexplore.ieee.org/abstract/document/785511/