Growing with Big Data, A Tetris Player: Project Report by Group 22 *

Loh Han Tong, Victor
A0135808B

Ryan Tan Wen Jun
A0135747X

Tan Yu Wei
A0142255M

Li Jiaxin Cassandra
A0131652N

April 8, 2018

1 Introduction

The purpose of this project is to create a utility based agent to maximise the number of rows removed in a game of Tetris. This tetris playing agent uses a heuristic function to estimate the utility of each state.

In this report, we discuss how this agent was designed and the features used to evaluate the utility of the board. We will also look at how we have implemented and used genetic algorithm to train a tetris agent that could play Tetris decently well, averaging about 19,700,000 lines cleared.

2 Strategy

The agent's heuristic function sums the linear weights $w(k)$ of features $\varphi_k(s)$ (As stated in subsection 2.1) for a given state of the board, s, where n is the number of features as shown below:

$$\hat{V}(s) = \sum_{k=0}^{n} w(k)\varphi_k(s)$$

Where at every turn, the agent evaluates, among all possible moves using the heuristic function, the move that gives the best utility.

2.1 Features Selected

This is the list of 11 features that we have selected. They allow us to evaluate each state s based on certain characteristics of the board.

- **NUM_ROWS_REMOVED** – Number of rows removed
- **MAX_HEIGHT**
- **TOTAL_HEIGHT**
- **TOTAL_DIFF_HEIGHT** – Sum of all difference in height of all columns
- **LANDING_HEIGHT** – Height of where the next piece lands
- **NUM_HOLES** – Number of empty cells with at least one filled cell above
- **COL_TRANSITION** – Number of filled cells adjacent to empty cells, summed over all columns
- **ROW_TRANSITION** – Same as the above, but applied to rows
- **COVERED_GAPS** – Number of empty cells with a filled cell anywhere above them
- **TOTAL_WELL_DEPTH** – Sum of the depth of all wells
- **HAS_LOST** – Gives a penalty of -10000 if move result in loss, else give 100

2.2 Genetic Algorithm

For our implementation of the genetic algorithm, Each chromosome has a weight vector where each gene (weight value) corresponds to one of the 11 features stated in subsection 2.1, and a fitness score.

*Contributions are weighed by the opacity of the author's name

The fitness score of each chromosome is defined as the mean score of playing 50 games using that individual's chromosome weight.

This is our implementation of the genetic algorithm:

1. Start out with 1000 individuals with random weights. Initially calculate their fitness score.

2. Select 40% of population via Stochastic Universal Sampling to be potential parents

3. Generate 40% of population as offspring by the process below:

   (a) Randomly select 2 parents from the pool generated above

   (b) Crossover with 80% chance, by taking weighted average of genes

   (c) Mutate these 2 offsprings with 8% chance by adding 1/10 times the random gaussian value.

   (d) calculate fitness score for the 2 offsprings

   (e) Add to offspring pool

4. Cull bottom 40% and replace with offsprings in offspring pool

5. Repeat steps 2 to 4 for each generation, till convergence

Convergence is determined by the score of the best individual in the population. If this score has not improved for 50 generations, we terminate the algortihm.

2.3 Parallelisation and Speedup

Each generation of the algorithm required running games to evaluate fitness. This meant that as the weights progressively got better, each generation started taking a longer time to evaulate.

We decided to parallelise the games by running each game on its own thread. Playing 100 games each, with a set of decent set of weights [1] , the time taken for the parallelised version was 2059 seconds while the sequential version was 6897, giving a speedup of 3.34 times.

Another way that we have tried speeding up the learning algorithm was to reduce the size of the board by reducing its height. Our team ran 2 different learners, one learning on a 9x10 board, while the other learning on a 13x10 board. The learner with 9 rows, even at later generations, took an average of 30 minutes per generation, while the latter, took an average of 4 hours per generation.

The machine used for the training and learning the weights, and running all the above tests was ¡PLEASE WRITE THE SPEC OF NUS CLUSTER SERVER USED, ITS NUS COMPUTING CLUSTER xgp0¿

3 Results

The following results are from the weights shown in Table 1. These weights were derived from the genetic algorithm learner on a board with 13 rows at generation 132.

Features	**Weights**
NUM_ROWS_REMOVED	-0.10994115458466136
MAX_HEIGHT	-0.1154697834187254
TOTAL_HEIGHT	-0.04390525258236673
TOTAL_DIFF_HEIGHT	0.017912908135268947
LANDING_HEIGHT	-0.3044476707923254
NUM_HOLES	-0.38617473506172584
COL_TRANSITION	-0.12518629866820255
ROW_TRANSITION	-0.22806177833393343
COVERED_GAPS	-0.7696058904564755
TOTAL_WELL_DEPTH	-0.19377750577164388
HAS_LOST	0.13672271498097804

Table 1: Respective Weights for Features

The result of running 600 games can be seen in Figure 1, while some common metrics of the 600 games can be seen on Table 2.

Metrics	**Score**
Q1 (25th Percentile)	6,307,657.5
Median	13,655,622.0
Q3 (75th Percentile)	25,716,898.5
Mean	19,793,958.2
Max	216,319,742.0
Min	5125.0

Table 2: Common Metrics for the Scores

[1] weight vector used w = [0.00134246, −0.01414993, −0.00659672, 0.00140868, −0.02396361, −0.03055654, −0.06026152, −0.02105507, −0.0340038, −0.0117935, 1] played over 200 games in total, 100 games sequentially and 100 games in parallel, with a total average score of 841279

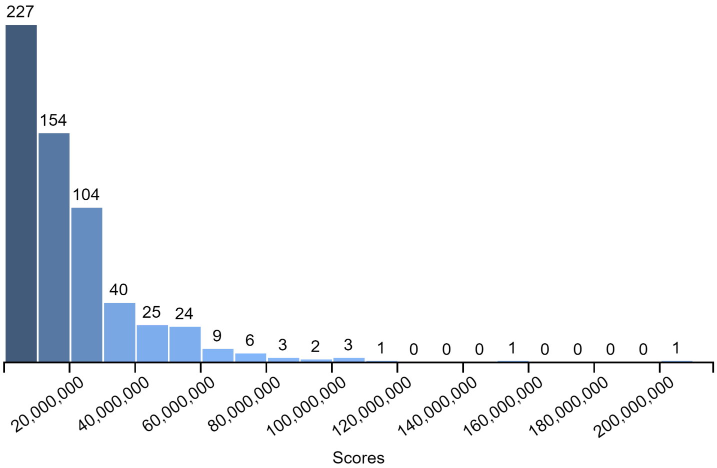

Figure 1: Results from 600 games

As we can see from the result, most of the games played lie below 30 million lines cleared. However we do see a few outliers that broke 50 million lines cleared, including our best run at 216,319,742 lines cleared.

4 Discussion and Findings

Our group initially decided to use Least Square Policy Iteration (LSPI) by Lagoudakis [4], seeing as earlier CS3243 groups such as Tan et al [1] and Nguyen et al [2] had been able to get satisfactory results from it. However, the results that we have produced from our implementation of LSPI was not satisfactory, giving an average of at most 8,000 rows cleared.

We speculated that there may have been an error in our implementation, but despite peer review, we were not able to rectify the issue. In a bid to try to improve our learnt weights, we tried implementing other features to better represent the state of the board. This was when our team realised the importance of the choice of features.

We initially implemented our original set of features without HAS_LOST. Most of which was taken from previous other works of tetris playing agents, such as the Tetris applications by Colin Fahey and Pierre Dellacherie [3], because of how well these features had worked.

However, our original set of features do not take in account of whether the next move will result in a loss, therefore an important characteristic of the board has not been captured. Implementing HAS_LOST discourages our agent from making losing moves if there are other available non-losing moves.

After implementing this feature, the number of rows cleared on the average increased from 8,000 to 80,000. Nonetheless, the results were still unsatisfactory and the team decided to change the implementation of the learner to use our own version of genetic algorithm instead.

Metrics	Mean Score
13 Rows Learner (Generation 132)	19,793,958.2
9 Rows Learner (Generation 464)	12,872,842.8

Table 3: Comparison of the final top weights of each learner after 9 rows learner reached convergence

With regards to the 2 instances of learners running on 9 and 13 rows respectively. We found that the learner on the 9x10 board has converged much faster than the learner on the 13x10 board. However, testing on a full sized 20x10 board with 600 games, the player trained on the larger board gave a higher mean score than the player trained on a smaller board, as can be seen in Table 3.

The apparent tradeoff for this speed up in convergence, is that the set of weights obtained are not ideal for a full sized board, though the player still plays well. We think that learning on a smaller game board may be specialising the player for playing on a smaller game board size but does not enable them to generalise to larger game board sizes.

This illustrates the case that there exists a tradeoff between tweaking the size of the board (to reduce the time taken for convergence), as well as the quality of the weights learnt at the end, which would be another interesting area to look into if we had more time.

5 Further Considerations

In addition to the discussion on Section 2.3, another possible way that we could look into speeding up the fitness evaluation of each chromosome is by reducing the length of each game. If the player has reached the maximum number of moves made or has lost, report the score. However, this may lead to a set of weights that are specialised in playing only up to the maximum number of moves and not anymore. The combination of states that the player sees may also be biased to the states that are closer to the start of a game, thus not giving an evaluation that can be generalised to a standard game. Hence how we tweak the maximum number of moves made would be crucial in balancing the time taken versus quality of weights learnt.

Lastly, we can further parallelise the running of the genetic algorithm by distributing the fitness evaluation of each chromosome to separate machines in different clusters. This may decrease the computational time needed for each generation as the fitness evaluation for all chromosomes would now be running in parallel as well.

6 Conclusion

Our aim in this paper was to show we can use Genetic Algorithm to apply to strategies in terms of weights to our feature-based utility function to evaluate the best moves. We have showed that the algorithm could settle at a good set of weights, despite not having a guarantee that each consecutive generation would improve. The set of weights that we have derived based on these features could then be used in as a starting point in another algorithm in order to learn the optimal weights. One example may be Least Square Policy Iteration by Lagoudakis and Parr [4], of which our Tetris problem could be tailored to fit a control problem as stated by Lagoudakis.

We have shown that by reducing the size of the game board, we could still arrive at a set of weights that were decent. We have also discussed the tradeoff of reducing the size of the board, and the final weight's ability to generalise to larger board sizes. We also examined other ways to reduce the length of games and its possible implications on how generalisable the weights learnt at the end will be.

We also think that in order to design a utility based agent that could play Tetris well, we should not only rely optimising the weights for a set of features, but we also need to good features as well. As discussed in Section 4, having a set of features that could better represent the state of the game results in an agent that could potential play better. Perhaps another interesting area to research would be looking into creating algorithms that could optimise and learn new features of the board state, and thus the utility function, instead of only its weights.

References

[1] Shawn Tan et al, *Learning about reinforcement learning, with Tetris*, 2014 https://blog.wtf.sg/2014/01/12/learning-about-reinforcement-learning-with-tetris/

[2] Nhan Nguyen et al, *Learning to Play Tetris with Big Data*, 2016 https://github.com/ngthnhan/Tetris/blob/final/repor

[3] Colin P. Fahey, *Tetris AI*, 2003 https://www.colinfahey.com/tetris/tetris.html

[4] Michail G. Lagoudakis, Ronald Parr. *Journal of Machine Learning Research 4 (Dec), 1107-1149*, 2003