CS3243
Tetris Genetic Algorithm

Group 11
Yan Hong Yao Alvin (A0159960R) and Herman Wong (A0139964M) and
Lim Jun Quan (A0158575R) and Cherry Goh (A0147928N)

Abstract

This report describes the use of Genetic Algorithm to train the weights of a Tetris AI given a set of heuristics. In addition, this report discusses the effect of multi-threading on performance, the heuristics chosen for the program and the weights generated.

1. Introduction

The objective of the project is to create a utility-based agent to maximize the amount of rows cleared in a game of Tetris. To accomplish this, the agent utilizes a heuristic function whose weights have been trained using a Genetic Algorithm. The rules of Tetris for this project involves no known lookahead piece and randomly generated pieces.

2. Components

In this section, we will discuss the different components that enable us to both simulate a tetris player as well as learn from each simulation.

The following are the components that allow us to simulate a tetris game:

2.1 Player Skeleton

The player skeleton is the component that allows us to play the tetris game

2.2 State

This component contains all the information and methods to the current tetris grid

2.3 Next State

The Next State component allows us to get the heuristic values for the state of the tetris board after a move has been made. On top of that, it calculates the number of rows cleared after the move has been made; our fitness function. Hence, the main role of this component is to calculate the different heuristic values based on the best move made from the current state.

The next few components allow us to run and learn from the simulation, then apply genetic algorithm for all the simulations:

2.4 Learner

The Learner component is the object that we use to simulate the tetris game using a set of weights. We use Learner to run PlayerSkeleton; and the Learner will track the number of rows cleared when the game is lost along with the set of weights used for each feature.

2.5 LearningAlgorithm

This component is where all the learning takes place. It collates all the learners and mate then together to reproduce different set of weights, where it'll be store in an external text file.

3. Heuristics

In this section, we will describe the different features that we included into our heuristic and also elaborate on why we chose these few features. The heuristic is a linear combination of the above features' weights as well as the features' values. Each feature is calculated based on the attributes of the Tetris Board after a piece has been placed.

$$P(function) = \sum_{i=0}^{6} FeatureWeight[i] * FeatureValue[i] \quad (1)$$

The purpose of this heuristic value is to allow the **PlayerSkeleton** to pick the move that maximises this value. We will discuss why this value is being maximised later in this report.

A number of features were tested. However, some additional features reduced the computation speed and showed no significant improvement in increasing the agent's efficacy, and thus were discarded. The final set of features chosen are as follows:

3.1 Features

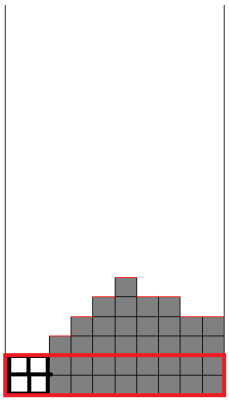

Figure 1: Rows Cleared

item **Number of Rows Cleared** — This heuristic tracks the number of line clear for each piece that is added into the next state.

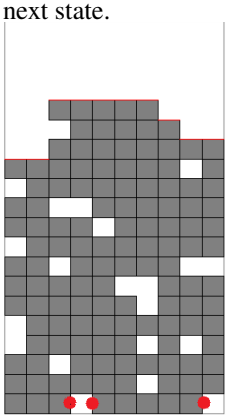

Figure 2: Rows Transition

- **Row Transition** — It is when a filled cells is adjacent to empty cells and vice versa on the same row summed over all rows. This heuristic was chosen because we wanted to minimise the number of empty spaces between filled cells within a row. This feature can hopefully select a move such that the row is packed together instead of leaving empty cells within filled cells in a row.

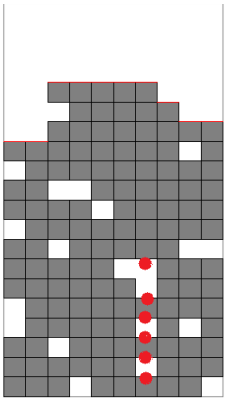

Figure 3: Column Transition

- **Column Transition** — It is when a filled cells is adjacent to empty cells and vice versa on the same column summed over all column. The purpose of this feature is similar to row transitions, except that it's counting for columns.

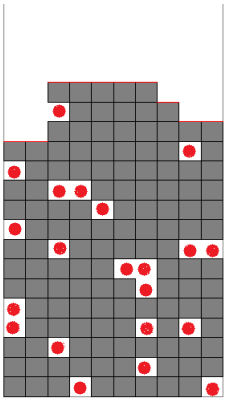

Figure 4: Number of holes

- **Number of Holes** — A hole is defined to be an empty cell that has at least 1 filled cell above it in a column. We chose this feature as we feel that having holes will significantly increase the difficulty of clearing rows for the tetris grid.

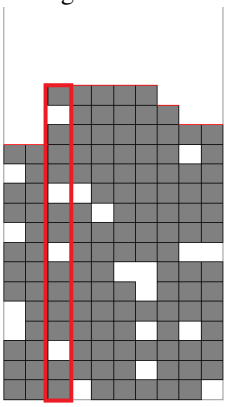

Figure 5: Max column height

- **Max Column Height** — The maximum column height of the current board state. By decreasing the max height, we will be further away from losing the tetris game.

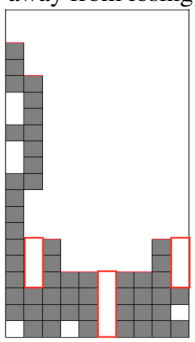

Figure 6: Wells Sum Feature

- **Wells Sum Feature** — For the well's feature, we exponentially penalised the height of the well using triangle number when calculating this feature. This is so that the higher the well, there will be more need for us to fill it up so as to prevent it from forming holes.

The first feature, rows cleared is initialised with a positive weights as we want to reward clearing more rows. The rest of the features are initialised with negative weights as we want to penalise them. Whenever we pick a move, we will choose the state that returns the biggest heuristic value.

4. Learning Method

Two different learning algorithms were considered for this project: Genetic Algorithm and Simulated Annealing. However, the Simulated Annealing algorithm was eventually discarded when it failed to offer promising results.

4.1 Genetic Algorithm

Genetic algorithm is an optimization strategy developed to partially mimic biology's natural selection. The algorithm functions by repetitively modifying a 'population'. At each generation, based on some criteria, a set of individuals are selected from the current population to serve as "parents" for the next generation. Subsequently, this allows the following generation to inherit ideal traits from the predecessors, until eventually a generation with optimal traits is achieved.

4.2 Methodology

4.2.1 Initial parameters — The initial population size is chosen to contain 100 learners. The learning algorithm is initialized by chosing a random value from the uniform distribution in the range [-8, 2] for each weight w_2, w_3, ..., w_6. We chose this range as the weight 2 to 6 are feature value that we want to minimise; hence, a higher chance of it being initialised is implemented. The weight for "Number of Rows Cleared", w_1, is initialised to a positive value with a range of [0, 20] as it is the only feature that should be maximised. The absolute range of value selected for each weight is largely arbitrary as we believe that it is the relative weight between each feature that matters, as they are floating point numbers.

4.2.2 Selection using Tournament Matching — Randomly select 8 individuals out of the population and select the 2 fittest individuals from the 8 for reproduction.

4.2.3 Reproduction/Crossover — After selecting 2 individuals from the population, a crossover is performed by randomly selecting a weight from either of the individual to be passed on to the next generation. Each of the weights between the two learners has equal probability of being chosen. This is performed for every weight in the set.

4.2.4 Mutation — This is carried out to prevent the algorithm from being trapped within a local maximum. After each reproduction, each individual's weight will have a 10 percent chance of mutating by a uniformly chosen value of up to +- 25 of the original range./

5. Distinguishing Factors

1. **Population Retention** — Since we are unable to predict the next piece available in a tetris game, the fitness levels of each species with the same weight is non-deterministic. To counter this, we will retain the best 20% of the species from the previous generation into the next generation without simulating the game with the species' set of weights again. This is to ensure that species with "potential" will be retained as we are afraid they may get unlucky the next simulation and produce a bad fitness level; resulting it its elimination.
2. **Averaging Fitness for Learners** — As mentioned earlier, Tetris is non-deterministic to a certain extent. In order to best minimise the effect this non-determinism would have on our program, fitness levels for each learner were evaluated by taking the average of its performance over three different Tetris games.
3. **Species Selection** — The way we select species from the population differs from the general genetic algorithm. In the most common version of this algorithm, the species are selected randomly with a probability that is related to their fitness level. In our initial attempts at training the AI, we used the general version whereby the probability of selecting the species is based on its fitness functions. However, we realised that this led to the species at a very fast rate. This is because in the first few runs of learning, all weights are randomly generated. With a population of 100, the variance of the fitness level amongst all species is huge as the bad ones are generally below 100 while there will be a few dominating species in the range of above 100,000. As such, the dominating species has a much larger probability of being selected, causing the next few population to converage very quickly at those dominating species. Hence, we used a selection process that increases the range of selection. We will randomly select 8 species from the population, with all species having equal chance, and choose the fittest 2 species out of the 8. By implementing this, we introduce more randomness when selecting species, allowing us to explore several different set of weights as the population will not converge as rapidly as the previous implementation.
4. **Reproduction Method** — The common genetic algorithm reproduces 2 species by having a crossover point and selecting the weights of species for the offspring based on the the crossover point. However, for our implementation, we have an equal chance of selecting each weight from the parents to produce the offspring. This is because we believe that there is no positional relation between the weights for a tetris game. Unlike the queen's problem in AIMA textbook, the number at each position is related to its adjacent one as the number at that particular position will affect the value at the next position. Example the string, "32752411", the number 3 at the first position actually affects the number at the second position which is 2. However, for tetris game, we feel that the position of our weights in the weight array does not affect it's value. Meaning, having the Rows Feature as the first element in the array or the third element in the array will not

affect the the outcome. Hence, we implemented such that the weights selected from parents does not have to come in sets that are predetermined in the order of the feature's weight in the array.

5. **Multi-threaded Learning** — The nature of the Tetris AI allows us to utilize multi-threading technology as any two board states and weight sets between learners is independent. By utlisiing Java's ExecutorService, we created a thread for each Learner to execute them in parallel. This implementation allows us to maximise the system's cores if there are idle ones. By maximising a system's cores, we are able to significantly reduce the time needed for one whole population to finish learning; allowing us to proficiently learn from massive amount of learners in a sufficiently short time.

6. Experimental Results

6.1 Weights generated by Genetic Algorithm

These weights are generated by the Genetic Algorithm over 20 iterations for approximately 12 hours.

Features	Weights
Rows Cleared	8.580230809919557
Row Transition	-7.323298208568506
Column Transition	-5.79715846115661
Number of Holes	-4.194861543999046
Well Sum	-3.908041863765
Max Column Height	-7.193861810393222

Table 1. Weights for each feature
The signs of each weights were as expected. Weights which we wanted to minimise came out as negative, while weights we wanted to maximise were positive.

6.2 Performance

To gauge the performance, we used the above mentioned weights and ran PlayerSkeleton 100 times and recorded the number of lines cleared. The results are shown in table 2.

Metric	values
Average	700,242
Max	3,035,634
Min	14,261
Median	668,002
Standard Deviation	579,925

Table 2. Metrics for 100 runs

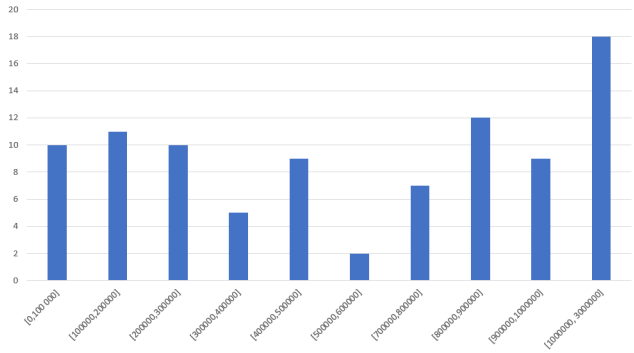

Figure 7: Rows cleared (100,000 interval)

The distribution of the result is shown in Figure 7. While we managed to clear such a significant amount of lines, we still obtain figures in the low thousands as seen from table 2. The large variance may due to the relatively small data set that we have computed. We averaged approximately 700,000 lines cleared with the current set of weights with the best record of 3,035,634 lines cleared.

7. Novel Ideas

In this section, we will discuss some of the novel ideas that our team came up with but did not utilise due to the lack of accessibility to high-end technology.

7.1 Looking Ahead

We have tested implementing a two-ply look-ahead strategy. Instead of only considering how a particular legal move will affect the next state of the Tetris game, the program will consider the effect of the move for the next two states. To accomplish this, for each possible legal move, the tetris AI agent will additionally consider each possible legal move after that move and calculate heuristic values based on the state that is achieved after those two moves. As the tetris game does not have a greedy property, this approach was chosen to make sure that the immediate best move facilitates the choice of the second best move.

However, due to the increase in computation time needed to decide on each move, we eventually decided use a single-ply search in our submission. There was significant increase of lines cleared using a 2-ply search of at least 5x, therefore this avenue may be more feasible with a stronger computation cluster and with a larger time frame to train.

8. Conclusion

In conclusion, after experimenting with a variety of different heuristics, 'number of rows cleared', 'row transition', 'column transition', 'number of holes', 'max column height' and 'wells sum feature' were determined to be the most effective features in maximizing the efficacy of our Tetris AI (under the set of rules provided in this assignment). By using a novel two-ply lookahead approach, our agent is able to perform its task with greater accuracy and through the use of multi threading, our program will also be easily scalable to big data.