CS3243 Tetris Project

Team 40
Aadyaa Maddi (A0161468Y)
Ronak Lakhotia (A0161401Y)
Kushagra Goyal (A0161459Y)
Sashankh Kumar (A0162363J)
Mukesh Gadupudi (A0161426L)

April 21, 2018

1 Abstract

This report proposes a Tetris game-playing agent that uses a genetic algorithm to evolve a set of optimal weights for features in a state space search of the game. We found that using an optimized evolutionary approach allows optimal solutions to develop naturally.

2 Introduction

Tetris has been used as a benchmark for learning game-playing algorithms because of its large state space and sizeable branching factor. This report describes how we trained our game-playing agent (Section 3) to play Tetris in an efficient manner by selecting the best performing heuristics (Section 4), which we learned using a genetic algorithm (Section 5). We used a multi-threaded approach (Section 6) to reduce the learning time. This approach works well for large amounts of training data, and performs even better with some novel optimizations (Section 7).

3 Agent Strategy

Our agent greedily picks the next possible best move from a given state by using a heuristic function to evaluate the value of a particular state, which is given as:

For a given grid state B, we have

$$V(B) = \sum_{k=1}^{n} \omega(k).\phi_k(B) \tag{1}$$

where n is the number of features, $\omega = (\omega(1), ..., \omega(n))$ is the weight vector, and $\phi_k(B)$ for $k = 1, ..., n$ are the feature functions for board position B.

The weighted sum of the feature values gives us our heuristic function.

4 Features

In this section we describe the features [1][2] we implemented for our heuristic function.

1. *Rows Cleared*: Number of lines cleared after a move.
2. *Maximum Column Height*: The height of the highest column in the playing grid.

3. *Mean Column Height*: Average of heights of all columns.
4. *Negated Number of Holes*: The negated value of the number of holes in the playing field.
5. *Negated Squares Above Holes*: The negated number of blank or empty squares above each hole.
6. *Surface Smoothness*: The degree of smoothness of the surface of the playing field, where 'surface' refers to the topmost blocks in each column.
7. *Number of Wells*: The number of wells in the playing field where a well is defined as a succession of empty squares in a column with the columns on either side having a height greater than the current column.
8. *Deepest Well*: The depth of the deepest well on the playing field.
9. *Cumulative Well Sum*: Sum of the accumulated depths of the wells.
10. *Row Holes*: Number of rows with at least one hole in them.
11. *Column Holes*: Number of columns with at least one hole in them.
12. *Row Transitions*: Number of filled cells adjacent to empty cells summed over all rows. It is assumed that the outside of the grid is full (both left and right).
13. *Column Transitions*: Number of filled cells adjacent to empty cells summed over all rows.

5 Algorithm

This section explains how we implemented the genetic algorithm.

5.1 Implementation

We implemented a genetic algorithm [3] to find the optimal weights for features in our evaluation function.

A chromosome in our population consisted of 13 genes, each of which corresponded to a feature. The fitness of a chromosome was calculated as the average number of rows cleared in 5 games using its weights.

The initial population of 200 chromosomes was then ranked by fitness and the top 60% was selected to participate in the crossover for the next generation.

In the crossover, we chose each gene from either of two participating chromosomes with equal probability to generate the offspring.

With a mutation probability of 10%, we altered a random gene in each offspring by adding a random value from a Gaussian distribution. The mutation introduced some randomness to the genetic algorithm so that it would not be trapped in a local maximum.

We then discarded the lowest 30% of the population and replaced them with the offspring to form the next generation.

5.2 Discussion

Our genetic algorithm initially selected the two fittest parents out of a random 10% sample of the population for each crossover. If the chosen sample is entirely unfit, the bad genes will propagate through generations. To eliminate this possibility, we considered the top 60% of the population to crossover instead.

6 Scaling to Big Data

This section explains the rationale behind using a multi-threaded approach for our learning algorithm and how we implemented it.

6.1 Purpose

We trained our agent for 250 generations to achieve an optimal set of weights. This required approximately 1,000 iterations of the game per generation, totalling to 250,000 iterations.

On a single thread, computation of fitness values for the entire population could take several hours for a later generation. Parallelization allowed us to distribute these time-consuming computations across multiple cores, thereby reducing the computation time significantly.

6.2 Implementation

We used the Thread Pool pattern and the Java ExecutorService library to parallelize our learning algorithm. Each chromosome in a population was multi-threaded into 5 threads, where each thread ran one instance of the game.

The results from the 5 threads were then used to calculate the average fitness value for that chromosome.

6.3 Speedup

We observed the execution time of the single-threaded and multi-threaded versions of our learning algorithm, with all other parameters constant.

After 12 hours of execution, the single-threaded algorithm reached the 22nd generation while the multi-threaded algorithm reached the 38th generation.

Figure 2 compares the time taken by both the versions to evolve the first 23 generations. Over these generations, we observed that there is a speedup factor of 8, which is likely to increase with the number of generations.

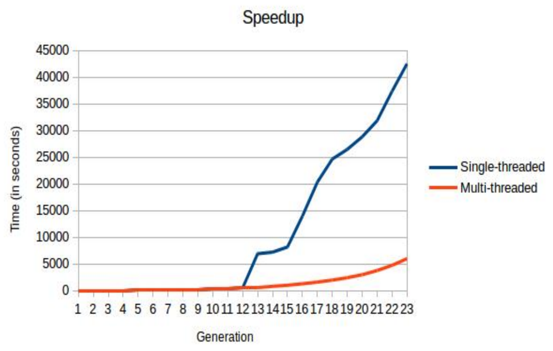

Figure 1: *Speedup achieved due to multi-threading as compared to single-threaded approach.*

7 Further Optimizations

This section explains the optimizations we added to the genetic algorithm to learn the optimal weights more accurately and efficiently.

7.1 Normalization

We normalized feature values to ensure their contributions to the evaluation function were solely determined by their weights, instead of their natural distributions. The trends in the weights were then representative of feature importance.

To obtain the means and variances of each feature, their distributions were observed over 4 million states. The normalized feature values were used in the evaluation function.

Without normalization, Negated Number of Holes ($\mu = 205.91$, $\sigma = 3.05$) will shift the evaluation function more than Surface Smoothness ($\mu = 1.08$, $\sigma = 4.23$), regardless of their relative importance.

7.2 Reduced Board Sizes

To further optimize training time from Section 6.3, we studied the effect of training our agent on smaller board sizes, with all other parameters constant. Best weights from training were chosen to be tested on larger boards.

For example, weights averaging 500,000 lines on the full board only completed 29,000 lines with 15 rows. Figure 2 shows a strong association between performance on smaller[1] and larger boards.

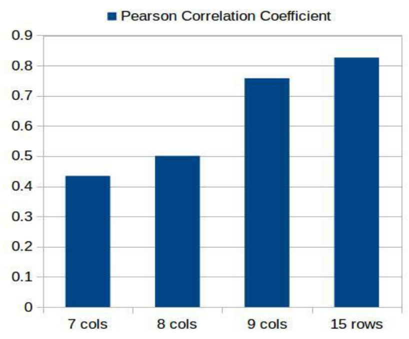

Figure 2: *Pearson correlation between performance on different smaller boards and the larger board.*

The 15-row board showed highest correlation ($\rho = 0.82$), while reducing training time significantly. Training on the reduced rows occasionally ranked good weights poorly owing to a horizon effect, where an unexpected improvement occurs beyond the 15th row. To mitigate this effect, the average fitness of each generation was also considered during weight selection since it is a more accurate measure of a generation's reliability.

8 Results

With the optimizations described in Section 7, our agent cleared 3,000,570 lines in the best case and 642,143 lines on average.

The average fitness showed a positive trend over generations, as shown in Figure 3.

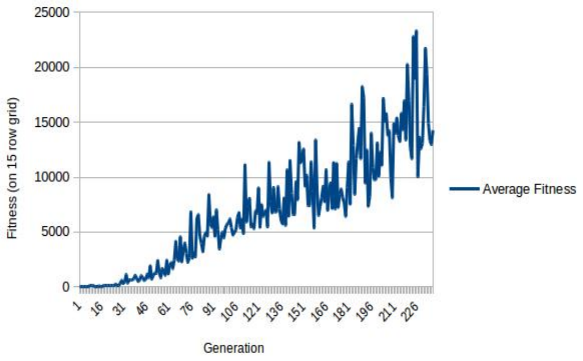

Figure 3: *Improvement in average fitness value over generations.*

9 Conclusion

We found that genetic algorithms are suitable for game-playing agents that use linear evaluation functions like Tetris.

Analyzing large amounts of game-play data provided us with domain-specific insights that helped us make statistical adjustments to improve the agent.

Since Tetris has a huge state space, training optimizations are crucial for obtaining optimal results in a sufficiently short time.

Further possible optimizations to the genetic algorithm include a feature selection implementation using the MapReduce Algorithm [4]. Moreover, a Quiescence Search [3] can be performed while training on reduced boards.

[1]Boards with reduced columns might prioritize different features, since the pieces fit differently. For example, wells might be considered more important in a 7-column board.

References

[1] Christophe Thiery, Bruno Scherrer. *Building Controllers for Tetris.* International Computer Games Association Journal, ICGA, 2009, 32, pp.3-11.

[2] Amine Boumaza. *How to design good Tetris players.* 2013.

[3] Niko Bohm, Gabriella Kokai, Stefan Mandl. *An Evolutionary Approach to Tetris.* MIC2005.

[4] Rania Saidi, Waad Bouaguel Ncir, and Nadia Essoussi. *Feature Selection Using Genetic Algorithm for Big Data.*

Appendix A Weights

Rows cleared	1.324002433
Maximum Height	0.922456272
Negated number of Holes	11.36137756
Surface Smoothness	-2.240630919
Negated Squares Above Holes	-3.899009256
Cumulative Wells	-6.215190494
Mean Height	-5.913481965
Row Holes	-11.42705303
Column Holes	-5.144538002
Deepest Wells	1.735036365
Number of Wells	0.212625253
Column Transitions	-8.553223037
Row Transitions	-6.35599271

Appendix B Usage

1. To play with the current best weights:
   ```
   java PlayerSkeleton
   ```
2. To run learning:
   ```
   java PlayerSkeleton args
   ```