Tetris AI with Parallel Genetic Algorithms
Tetris Agent 23

Andres Galaviz A0145029J
Michael Noven A0146876M
James Spicer A0145973U
Ryan Craig A0149051L

January 19, 2017

National University of Singapore

Abstract

This report describes the use of genetic algorithms to train a Tetris playing AI. In particular this report discusses how parallelization methods drastically reduce the time taken by each playing generation. The AI followed standard Tetris rules with no lookahead allowed and was using randomly generated pieces, the goal is to clear as many lines as possible before reaching an end game state.

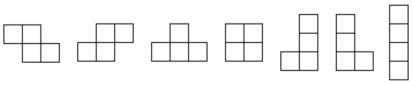

Figure 1: The seven possible shapes used in the game of Tetris

1 About The Heuristic

In the following section we are describing the heuristic we used to make this Tetris player.

1.1 Holes

We define a hole as a free square that has a filled square placed somewhere above it. As seen in Figure 2 having holes on the board makes it harder to clear a row. Thus we want to minimize the number of holes on our board.

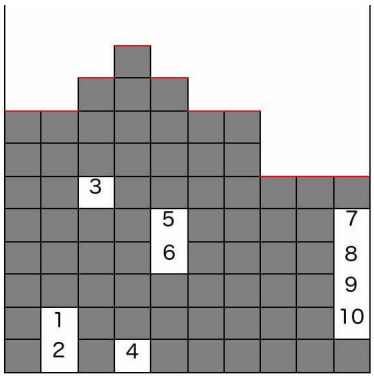

Figure 2: Number of holes = 10

1.2 Completed Lines

This is the most basic and most necessary heuristic out of them all. This is because we want to clear lines to get points. In other words, we want to maximize the number of rows completed. In Figure 3 you can see that if a straight piece is dropped four lines would be cleared.

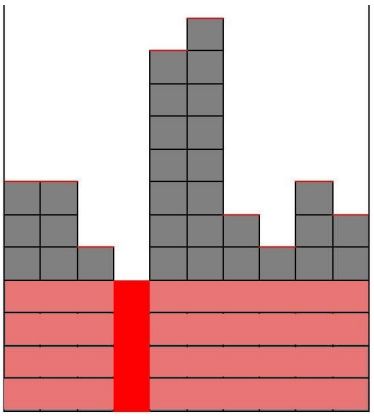

Figure 3: $LinesCompleted = 4$

1.3 Landing Height

Landing height is the new height of the column from where the pieces is dropped. Minimizing this is ideal as we want to keep our wall as low as possible.

1.4 Row And Column Transitions

For row transition it shows how many changes are happening on the row. To calculate this we go through each row and when the adjacent square is different from the current square you increase count by one.

Column Transition is very similar to row transition except instead of checking for changes in row the algorithm checks for column differences.

1.5 Well Sums

A Well is described as having a succession of empty squares in a column yet the squares on either side of them are filled.

$$WellSum = WellHeight + (WellHeight - 1) + .. + 1$$

Where $WellHeight$ is the deepness of the well.

We want this to be minimized as bigger wells can get covered making it harder to clear lines. Figure 4 shows 4 holes labeled as to show a general idea of how wells are counted.

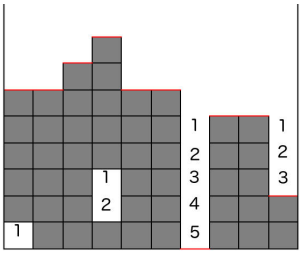

Figure 4: Max well shown = 5

1.6 Constructing The Final Heuristic

Now to get the score of the move we need to combine the six features into a feature vector:

$$\text{BoardFeatures} = \begin{bmatrix} landingHeight \\ rowsCleared \\ rowTransitions \\ columnTransitions \\ numHole \\ wellSums \end{bmatrix}$$

The equation is a linear combination of the features with the feature weights giving us a score output for the feature inputs.

Given we wanted to maximize completed lines we expect $rowsCleared$ to be a positive number. We wanted to minimize the other five features therefore we expect them to have a negative value.

Using our Genetic Algorithm (explained later) we were able to find very successful weight values for the specified features, thus allowing the AI to score a good amount of points. The best values we have found are:

$$\text{FeatureW} = \begin{bmatrix} -3.1472553592987946 \\ 2.46883837144299 \\ -2.2945510371937452 \\ -7.521200744605782 \\ -6.510648376902374 \\ -2.1908554239402918 \end{bmatrix}$$

Our fitness function is a linear combination of the feature weights and the board features. Namely:

$$\sum_{i=0}^{5} FeatureW[i] * BoardFeatures[i] = Score$$

2 Tetris AI

The AI algorithm described in this report uses a complete search approach. The AI simulates every playable move and then picks the one with the best score.

$$argmax(Score(move_i)) \; where \; i = \{0..NMoves\}$$

2.1 Score Function

Fitness of weights is defined as $Fitness(\vec{FeatureW})$ where the returned value is the number of lines cleared when using weight values $\vec{FeatureW}$. The worst possible fitness is 0, where the AI cleared zero lines. To obtain a better description of the fitness value we look at the aggregated fitness of 5 games to reduce the luck factor.

3 Genetic Algorithm

3.1 Initial parameter set

The initial population of 200 subjects is created by randomly generating linear combinations of the feature vector. Each of these initial subjects is added to the population with a value between -10 and 10.

3.2 Offspring creation

The offspring population is constructed by selecting 10% of subjects from the current generation, applying a crossover operation to the pair of fittest parents in that selection and randomly carrying out a mutation. This operation is carried out until the offspring population size is 30% of the whole population.

3.3 Selection

To obtain the parents a random selection of 10% of the population is obtained, out of this 10% we select the fittest two vectors and carry out a crossover operation.

3.4 Crossover

The crossover operation is carried out by selecting vector features from one of the two parents. This feature is then copied to the offspring vector.

3.5 Mutation

The mutation is carried out to prevent the algorithm from getting stuck at the local maximum. When producing a new offspring there is a 10% chance of mutation. If a mutation occurs one of the components of the offspring, which is selected randomly, will be changed by $\pm 10\%$.

4 Parallelization

4.1 Purpose

In order to complete the vast quantities of calculations included in each generation of a genetic algorithm in a time that is meaningful, we deemed it necessary to adapt our program to utilise multithreading.

4.2 Load Balancing

To achieve this goal we implemented load balancing frameworks via the Java Executors library. Calling to certain services, we create a threadpool which consists of all threads available on the specific machine we are running on. We then applied this threadpool dynamically over the area we wished to run concurrently, the 200 member population that runs 5 games on each, per generation. If we have 8 threads available on our machine, we would allocate each thread 1 member of the populus and thus have 8 members playing concurrently. The executor framework would then manage the threads on their return such that when a thread completes it's members calculations, it will automatically give that thread the

next member to compute, until we have completed all 200 members.

4.3 Callables and Futures

A standard thread implemented in java does not have the capability to return variables to it's master node, unless it writes output to a seperate file which is then iterated over by the master. However, this naive approach runs into the heavy overheads associated with Java's I/O functions, taking back from the purpose of adding concurrency features.
To avoid these inefficiencies, we created a class named gameThread, which implements callable instead of runnable (standard thread). A callable thread has the ability to return variables, which are allocated a fixed amount of empty space in an arrayList ahead of time (as they are submitted) such that when they return a value, a pointer simply places the value in it's allocated place. Instructing the master to do this is achieved via creating an arraylist variable of type futures, which will fill the datastructure in the future upon thread completion.
The variable passed between these functions was the total rows cleared achieved by the AI game before it either lost the game or reached the predetermined piece limit.

4.4 Speedup

Running our program concurrently will achieve a speedup relative to the specifications of the machine running the program. However, running the genetic algorithm serially (no concurrency) in practice completes around 4 generations over the course of an hour. When running concurrent with a factor of 8 threads available, in the same time we could complete 12 generations. The amount of speedup achieved here is exponential however, as the higher the generation, the better the population as a whole will become. Thus each member of the population will play each game much longer than earlier generations, requiring more work per generation. We also ran the genetic algorithm on the NUS Tembusu Server, over a period of 14 hours achieving 207 generations and producing us with our current optimal weights.

5 Results

When running the AI with the best found weights to date, we average 600,000 lines cleared. With our current record being 3,189,562 lines cleared in a game. We still get scores in the low thousands.

```
201-211:Git Ryan$  java PlayerSkeleton
You have completed 3189562 rows.
```

Figure 5: Screenshot of our terminal window showing our successful run

```
[AndresGalaviz:TetrisAI AndresGalaviz$ java PlayerSkeleton
 You have completed 1665773 rows.
```

Figure 6: Screenshot of our terminal window showing our successful run clearing 3,189,562 lines.

```
C:\Users\James_000\Documents\Git\TetrisAI>java PlayerSkeleton
You have completed 690759 rows.
```

Figure 7: Screenshot of our average game.

```
➜ TetrisAI git:(lookahead) ✗ java PlayerSkeleton
You have completed 428188 rows.
```

Figure 8: Screenshot of one of our lowest game.

In order to achieve the weights which attained such goal, we ran the genetic algorithm on the NUS Tembusu Server for 14 hours, attaining these phenomenal scores after 207 generations. At this point the changes in weights where very low between the best performing weights of each generation.
The weights used are as follows:

6 Conclusion

We found that defining good heuristics is a crucial for increasing the quality of the AI. When we increased the number of heuristics from originally four to six the results increased dramatically.

Also when evaluating the how good a Tetris AI is, it should be taken into consideration how the AI is defined and what kind of rules are given. Since all pieces are completely random, the luck plays a big factor. To minimize the luck factor, we tried to predict the worst possible next move, after the best theoretical moves. This was done by looping over all pieces and the score of the worst move was then selected. This score was then added to the weight of the selected theoretical moves. If done right - this should significally decrease the luck factor. However, we encountered some problems when doing this and the AI was not performing well.

Appendix

A. How to run the program

```
java PlayerSkeleton -e // Evolution
java PlayerSkeleton -n // Run with no window
java PlayerSkeleton // Play with current best weights
```