**CS3243 Project**
**The Tetrominator - Designing a Tetris Agent**
13 April 2017
Team 05
.

**Adam Alsegard A0153097B**
**Erik Andersson A0153524L**
**Qufei Chen A0163986L**
**Wojciech Dziwulski A0153992U**
**Vemund Woeien A0153463H**

## Introduction

The purpose of this project is to design an AI agent that maximizes the number of rows cleared in a game of Tetris. This is done by designing a learning algorithm to train the agent to best utilize a set of features - i.e. optimizing the agent's utility function. In this report we will be outlining our experiments with using variations of the genetic algorithm and particle swarm optimization to determine the best implementation of the agent's heuristic function. We will also be discussing the application of parallel computing techniques to reduce computing time and allow scalability up to big data.

## Designing Features

The quality of the agent's heuristic depends on the characteristic features with which we evaluate the game board, making the selection of said features extremely important. We started off by establishing 15 different features (shown in Table 1). With these we can evaluate the state of the board, with each of these features possessing its own weight value. Features 14 and 15 are unique in that they are calculated using multiple weight values. Since the game board has 10 columns, feature 14 will be calculated with 10 unique weights, one applied to each column. Similarly, feature 15 is calculated with 9 weights, one for each pair of adjacent columns.

We began our experiments using all 15 of these features, but quickly found that certain features had a larger impact on the utility than others. This will be discussed further in the Optimal Set of Features section of our report.

| Feature Number | Feature Description |
|---|---|
| 1 | Number of filled spots |
| 2 | Weighted number of filled spots |
| 3 | Maximum column height |
| 4 | Minimum column height |
| 5 | Number of rows cleared |
| 6 | Maximum column height difference |
| 7 | Number of holes |
| 8 | Depth of the deepest hole |
| 9 | Number of hole clusters |
| 10 | Number of wells (uncovered holes 3 or more blocks deep) |
| 11 | Sum of all well depths |
| 12 | Game status (1 if game has been lost, 0 otherwise) |
| 13 | Sum of all column heights |
| 14 | Height of each column |
| 15 | Height difference between each pair of adjacent columns |

Table 1: Initial set of features

## Genetic Algorithm

In evolution, it's all about the fitness - only the strong pass their genome down. The building block of the algorithm is hence an individual whose fitness we can measure. The factor we will be learning (the genes of the individuals in our population) is a weight vector being applied to our feature set, which will decide on our Tetris strategy and hence the final score. Given a randomly generated population of individuals, we try to mix and reproduce them in order to produce the most promising offspring.

The optimization thus bases itself on 3 operations - selection, crossover and mutation. We choose the pairs of ”parents” using a technique called ”tournament selection” - randomly picking sub-populations of a prescribed size and find the best individuals among them. We then **cross** their genes **over** in a random fashion, picking the fitter ones with a prescribed probability. This **exploits** the currently held state without **exploring** other possibly successful combinations in the optimization hyperspace. The exploration occurs in the last step of the algorithm - it changes the genome of each individual randomly to see if a better fitness can be obtained.

We further experimented with the exploration vs. exploitation trade-off by varying some of the parameters that affect how the evaluation was executed. The parameters tested were the mutation rate, the population size, the tournament size, and the crossover process. These parameters will effectively control the extent to which one explores the search space; the higher the values the greater the exploration. We found the following values to yield the best results: population size of 1000, mutation rate of 0.05, and tournament size of 20. We also decided to implement an adaptive crossover rate that prioritizes stronger individuals by taking a larger fractions of the stronger gene for the new offspring.

After our initial implementation, we quickly identified a few improvements that could be made on our approach. The first adjustment was to use a normal instead of a uniform distribution for the initial generation and mutation of the genes. This change diversified the gene pool and allowed for values outside of a strict range, giving us a broader starting point for our learning algorithm. Another modification we attempted was to normalize the features by their respective means. Due to the diversity of the feature values the weights were more likely to be generated within 1-sigma range, which might be preferential to certain features. This approach didn’t bring any improvement, however, most probably because it capped the influence of the most representative features. Lastly, we determined that the search space produced by having 15 features was too large. We were able to dramatically improved the fitness score by picking a more restricted set of features, which resulted in a much smaller search space for the algorithm to traverse.

## Optimal Set of Features - Another Genetic Algorithm

One way to find the best feature set was brute force - just trying every possible combination of features. This approach quickly proved to be much too inefficient - 1365 instances of the genetic algorithm would have to be run for 4 features and 3003 for 5. We decided to find the optimal set with another optimization technique.

A new genetic algorithm was thus implemented whose fitness measure was the ultimate score of the genetic algorithm and whose individuals’ genes were the feature numbers used for calculating the utility. The most vital difference was the gene generation subroutine which made sure that the constituents of the genome were unique integers instead of not-necessarily-unique doubles. The investigation revealed that the most optimal set of 4 features was [7,9,11,15] achieving 71,146 rows cleared and the most optimal set of 5 features was [7,9,10,11,15] achieving a fitness score of 102,837 rows cleared.

## Resolving Local Maximums

As we experimented with the genetic algorithm we discovered that the fitness score would occasionally plateau at certain values. This is indicative of reaching a local maximum. We were able to determine the presence of local maximums by monitoring the growth rate of the fitness score over subsequent generations. Once we have detected that the growth rate of the fitness score has decreased under a certain threshold, we tried two different methods in an attempt to escape the local maximum: increasing the mutation rate of the genetic algorithm, and applying particle swarm optimization.

In the first method, we increased the mutation rate of the genetic algorithm to introduce more variation into the gene pool in an attempt to produce a more diverse population. This, in theory, should provide a wider range of weight values and thus more opportunities for the algorithm to overcome the local maximum. But our experimental results showed that while increasing the mutation rate did lead to the escape from the local maximum, the increase in diversity also slowed down the growth rate of future generations. This resulted in an overall lower fitness score after 30 generations (see Figure 2).

In the second method, we attempted to use Particle Swarm Optimization (PSO) to break out of the local maximum. This optimization method places each instance, called a particle, in an n-dimensional search space (with n = the number of features). Each particle stores it's personal best location in the search space, as well as unique velocity. A global best for the entire population is also stored. After each iteration, all of the particles will move through the search space towards the direction of the best solution. Each time we detected a local maximum we mapped each individual in the genetic algorithm to a particle, ran the PSO algorithm for 3 iterations, and then mapped each resulting particle back into an individual for the genetic algorithm. From our experiments, we found that PSO was indeed very effective at escaping local maximums, and some PSO iterations resulted in a significant increase in the fitness score. However, this result is not consistent. In some iterations of the PSO, the fitness score would actual decrease, producing a lower fitness than the previous generation. However, the increases generated by the algorithm were greater than the occasional decreases, and we resulted in an overall higher fitness score (see Figure 2).

## Experimental Results

Table 2 summarizes our experimental results. When experimenting with the adaptive mutation rate and PSO, we used the formula $\frac{currentFitness - pastFitness}{pastFitness}$ to calculate the growth rate, and used a threshold of 0.2 as our baseline.

| Algorithm | Lines cleared | Time (s) |
|---|---|---|
| Genetic 30 generations | 97487 | 2309 |
| Genetic 50 generations | 102837 | 3404 |
| Genetic 30 generations + adaptive mutation rate | 25877 | 524 |
| Genetic 30 generations + particle swarm optimization | 106160 | 1385 |
| Genetic 50 generations + particle swarm optimization | 152594 | 1957 |

Table 2: Results with features [7,9,10,11,15], pop-size 50, crossover rate 0.7, mutation rate 0.05.

From our experiments, the highest fitness score we obtained was 152594 lines cleared acheived through combination of the genetic algorithm with PSO. This score was achieved with the following set of weights: [15.76, -11.43, -13.29, -0.92, -3.69, -4.41, -3.06, -4.29, -3.85, -3.70, -2.87, -5.21, -2.05, -3.44]. One area of interest is that all of our weights are negative values. This is related to our features being undesirable for a Tetris board. For example, as you don't want holes in the board, the weight corresponding to the feature "number of holes" will be negative (within reasonable limits) to minimize the number of holes in the resulting state after completing a move.
Another interesting result yielded from a stability test executed on the resulting weight vector. When re-running 10 consecutive games with the same weights, results ranged from a low of 16,703 lines cleared to 256,803 at best. The standard deviation was 72,227 for a mean of 147,780 rows cleared.

## Training with Multiple Cores

An intuitive way of achieving speedup during the agent's learning process is to divide the independent workloads to different processing units. This was done by splitting the workload at the mid-level of the code execution. For each generation the genetic algorithm explored different weights originating from the weight array; one such set of weights is called an Individual. For every Individual the fitness score is calculated, as an average over 10 games. Therefore, for every generation we need to run 10 games for every Individual in the population. This is the part in the genetic algorithm that is the most computational heavy and it is at a high enough level to parallelize without extensive overhead.

We ran the parallelized algorithm on 1, 4 and 12 cores to calculate the speedup. To remove the element of chance each parameter-setup ran the genetic algorithm for five iterations, and all iterations with fitness score below 1500 were classified as outliers and removed since they failed to hit the bottleneck of the computations.

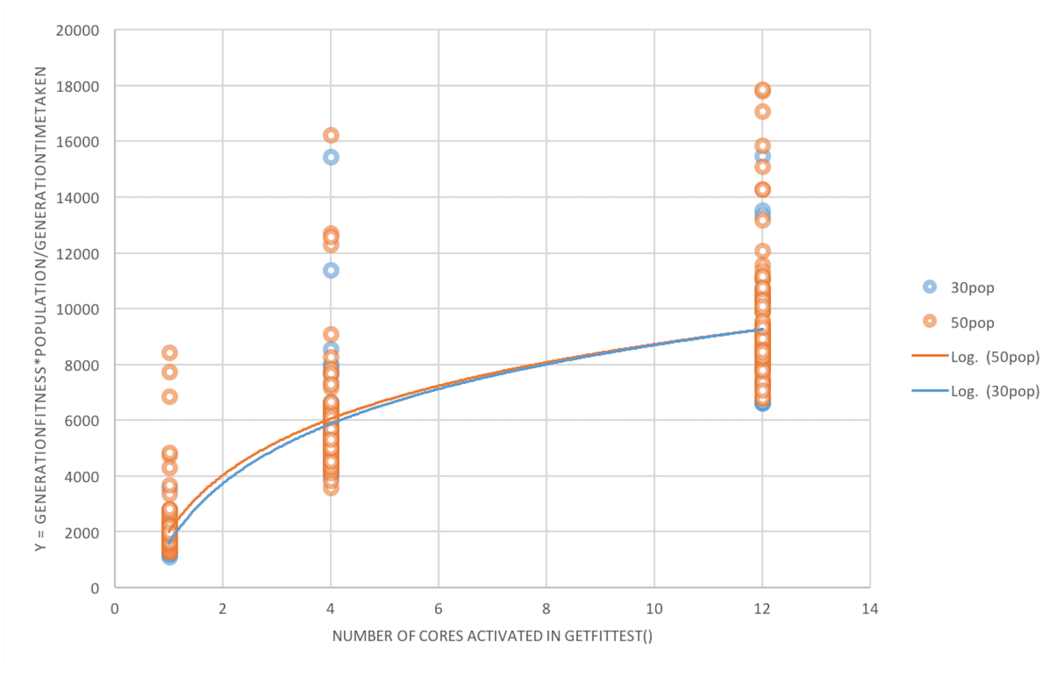

Figure 1: Fitness weighted with population over active cores

The ratio $\frac{GenerationFitness \cdot population}{GenerationTimetaken}$ measures the relative throughput at each generation. We multiply the ratio with its associated population size in order to balance it; this is to ensure that runs with a population size of 30 do not show higher throughput than runs with a population size of 50 without regard to that the 50 population run explores 20 more weight-setups each generation.

With one active core as the baseline, we see that a jump to 4 active cores results in throughput speedup of 300% and a jump to 12 active cores results in a speedup of 460% (see Figure 1). In conclusion we can say that while more active cores are advantageous, but we can only at best achieve a logarithmic speedup.

Another way to scale up the algorithm for big data (when board size and/or number of pieces increases significantly) would be to parallelize the pickMove function. A version of this was implemented and tested with different column sizes (10, 40, 120). However, the tests took too long and wasn't able to run to completion before report submission. It could be that the tested data is still too small to overcome the additional overhead that the parallelization implies, and we need to increase it further to see the actual speedup.

## Conclusion

Using variations of the genetic algorithm combined with particle swarm optimization, we were able to determine the best possible set of weights for our feature set. Our learning algorithm obtained the following weight set [15.76, -11.43, -13.29, -0.92, -3.69, -4.41, -3.06, -4.29, -3.85, -3.70, -2.87, -5.21, -2.05, -3.44] which resulted in a score of 152594 rows cleared. Although our algorithm resulted in a fairly high fitness score, there was great variability in our results. As the game of Tetris is based upon some randomness, variability was expected rather than surprising. Nonetheless, the amount thereof was higher than our prior beliefs. Overall, the agent designed has successfully fulfilled the project objectives and achieved state-of-the-art score in the simple, yet mesmerising, game of Tetris.