CS3243 Team Project Report : Learning to Play Tetris with Big Data !

Team Number 05
Group :
GEORGE, AISHWARYA - A0174981R, HRITTANE, MERIEM - A0175247Y,
MASSART, RAPHAEL - A0175639N, NGUYEN, VIVIAN- A0179045Y,
WANG, CHIH-HAO - A0175111U

0.1 Introduction

In this project we aim to create an artificial intelligence program that can play Tetris, a popular puzzle video game, and employs techniques involving the use of Big Data. The end goal of our project was more about building a scalable system that used innovative learning techniques to build a game strategy, rather than maximizing the average number of rows cleared by the program, as we felt the former is where the more interesting aspect of the project was.

WANG, CHIH-HAO created a guide for the rest of the group, to explain clearly and step by step how to edit our code and run it on Sunfire, save it in our machines, and export data in a graph [1]

0.1.1 Our Contributions

During our research we found that many of the available Tetris AI projects were quite similar — therefore to ensure we contributed some new discoveries to the existing work we tried to use more innovative methods wherever possible. The final version of our agent implements a total of eight heuristics which it uses to decide what move to make — as well as this, we also implemented a "one-piece lookahead" function which the AI also uses to aid its evaluation. Most of the literature we came across used genetic algorithms to determine the optimal weights for their respective heuristic functions. Therefore, to challenge ourselves and see if we could get decent results from another algorithm, we decided to explore the lesser used method of Particle Swarm Optimization (PSO) to determine the weights for our heuristics. We then used multi-threading with PSO to further optimize the process of deciding the weights. The following sections of the report go into detail about our process and experience with these methods.

0.2 Heuristics

We considered each configuration of the Tetris grid to have some associated cost/gain, and we evaluated this cost/gain by considering a number of heuristics. To calculate the cost/gain of a grid, we wrote separate functions for each important heuristic of the grid, added the cost/gain of each of these heuristics into a weighted function which we then sought to optimize. As previously mentioned, we obtained the weights for the heuristic function by using the PSO algorithm in Java.

We used the following heuristics to help predict which move is the best one to make next. The AI aims to minimize the value of the resulting heuristic function, which considers the below 8 features at various weights ;

*1) **Rows Cleared** : The number of rows cleared in a move. This is the primary objective in a game strategy — one of the main ways we prevent death*

*2) **Total Height** : The aggregate height values for all the columns. This is the secondary objective in a game strategy, keeping the overall height low reduces probability of death when adding the next piece*

*3) **Maximum Height** : The height value of the highest column. This heuristic gives more precise information about how close the agent is to death*

*4) **Height Differences** : The sum of all the height differences between adjacent columns. The other height heuristics do not provide information about the distribution of heights, and this one covers this*

*5) **Number of Holes** : Represents the number of empty single cells on the grid where there is another filled cell above it. The presence of holes indicates spaces in rows that are not filled, and therefore cannot be cleared*

Our first step in designing our AI involved coding the individual heuristic functions. Through trial-and-error, we hard-coded the weights with intuition to see if we could improve our rows cleared, and we obtained roughly 150 lines. Later in the document we will talk about the use of Particle Swarm Optimization (PSO). We found out the use of PSO was optimal when the number of particles was below 2n, n the number of heuristics (Needs citation). In order to increase the number of particles and therefore improve our algorithm, we added these 3 heuristics.

*6) **Deepest Well** : A group of empty cells — one cell wide and at least one cell deep — with the highest depth value. Height difference doesn't capture the deepest well, therefore we need this function because the deeper a well is, the harder it is to fill*

*7) **Column Transition** : The aggregate number of transitions between empty and non-empty cells for all columns. For both the transition functions, the higher the value the higher the penalty — this is because empty single cells are more difficult to fill than groups of empty cells*

*8) **Row Transition** : The aggregate number of transitions between empty and non-empty cells for all rows — same reason as column transition.*

The additional feature we mentioned earlier, "one-piece lookahead", calculates the minimum cost of all legal moves for any of the 7 pieces that may be given one-piece ahead of our current piece. It then returns the next move we should play to minimize the sum of all minimum costs for every possible piece that we might receive in the next round.

0.3 Finding Optimal Weights

0.3.1 Genetic Algorithm vs. Particle Swarm Optimization

There exist many algorithms to optimize the weights, The Genetic Algorithm and Particle Swarm Optimization were the most common and efficient ones. Concerning the performance of the two algorithms, PSO allows obtaining a better result in term of fitness values and computational costs. Moreover, the standard deviation of best fitness values in GA are more elevated compared to the ones obtained with PSO algorithm [2]. We chose to implement PSO and our reasoning for this is when we came across Tetris AI implementations [3], it required a genetic algorithm population of 1000 randomized weight vectors and at every evolution cycle of the algorithm, each vector would have to crossover with another weight vector, mutate individual components, and cull a percentage of the population until convergence is reached. Each new weight vector would be evaluated for its fitness after playing the game with a fitness function we define. This is computationally demanding compared to the PSO algorithm. For example, the particle swarm algorithm would need not more than 150 particles at most for our problem.

0.3.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population based algorithm that is modelled on the swarming behavior in birds flocks. Every swarm is composed of particles that try to optimize their positions, with information from their past experience as well as all the other particles' experiences. We can visualize each particle as a 1 x n matrix where n is the number of inputs (weights in our case) that searches through a search space of dimension n. The vector represents the particle's position initialized randomly before trying to move to the next position. The way we update the position is as follow :

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{1}$$

where t is the current training iteration, $x_i(t)$ and $x_i(t+1)$ are the i-th particle's current position and next position respectively, and $v_i(t+1)$ is the velocity calculated for the next iteration. Velocity plays an important role in PSO since it determines how the positions are updated. In addition, there are two positive acceleration constraints : cognitive component and social component, which helps a particle balance the effect of its previous experience, and helps a particle gravitate towards the global best position found by the swarm, respectively. The velocity is updated as follow :

$$v_i(t+1) = \phi v_i(t) + c_1 r_1(t)(y_i(t) x_i(t)) + c_2 r_2(t)(Y_i(t) x_i(t))$$

where ϕ is inertia weight to balance the swarm's exploration and exploitation ability, c_1 and c_2 are positive acceleration constants, and $r_1(t)$ and $r_2(t)$ are random vectors generated from Uniform distribution [3]. v(t) and x(t) range from a min and a max values and are initialized randomly from a uniform distribution. At first we assign the particle a random position (random weights for each heuristic) then complete one iteration during which, in our case, they play a game of Tetris. The result of each game is evaluated by a fitness function to measure the performance of the particle. Its value is then compared with the best fitness this particle achieved locally so far, as well as the global best fitness among all the particles. If the current iteration has a better fitness, the local and global best fitness are updated accordingly. The position of each particle are then adjusted based on its velocity, which is calculated using its current position, local best position, and the global best position. After the specified number of iterations, the 'swarm' of particles converge at an approximate global optimal position. The process continues with the remaining number of iterations, so that by the end we obtain a set of global optimal values for our heuristic weights which result in the best performance.

We used the Open Source JSwarm-PSO library to assist us in training the AI [4].

0.3.3 Experiments

Using our first fitness function, $f = rowsCleared$, we executed 100 iterations of the PSO algorithm and we realized it was not learning as efficiently as we liked. We reasoned that this could be because either our fitness function only gives partial information on the particle performance or we are not running enough iterations. We then decided to implement PSO with both 100 and 500 iterations to see if we could achieve a convergence after a longer period of time. We used three different fitness functions to see how much an influence on the learning rate of PSO the fitness function had. These functions were :

$$f_1 = rowsCleared - avgNumHoles - avgTotalHeight$$

where $avgNumHoles$ is the average number of holes in a game, and $avgTotalHeight$ is the sum of the total height during each state for the entire game divided by the number of states. None of these results performed very well, clearing about 500 lines on average. Concerning the parameters in PSO such as inertia weight, cognitive terms etc. - we decided to use the ones provided in the original paper about PSO [3] as these were tried and tested to work. The result of one of these runs using the f_1 fitness function, 30 particles, 100 iterations, is shown below.

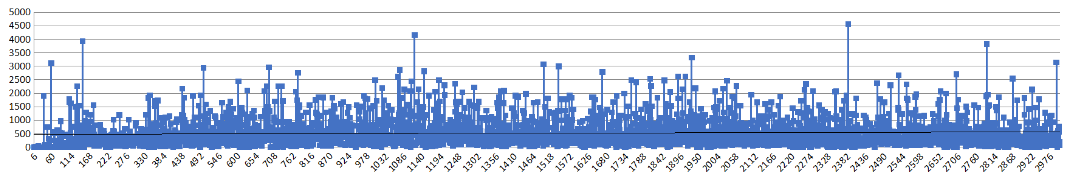


FIGURE 1 – PSO 30 particles 100 iterations (coefficients from the paper) f = rows - avgNumHoles - avgTotalHeight

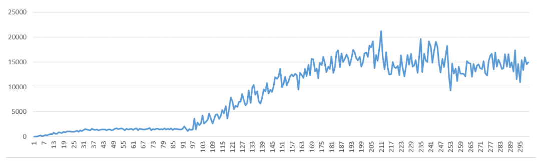


FIGURE 2 – *fit* ; 5 games per particle ; 30 particles ; 300 iterations ; Range [-1, 1]

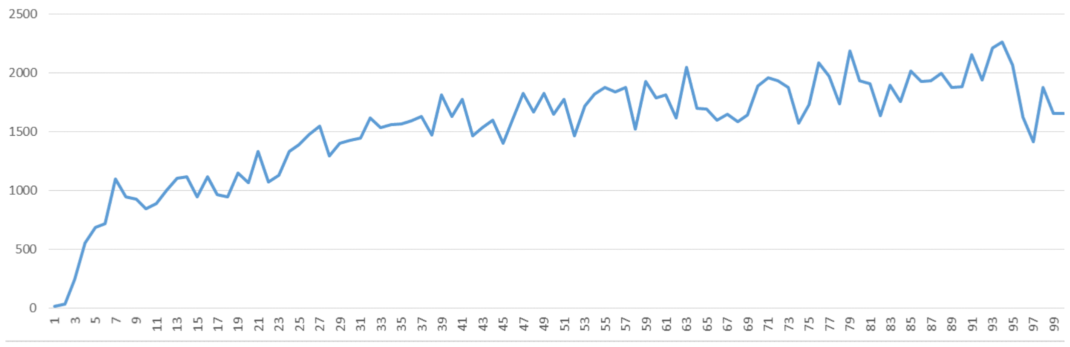


FIGURE 3 – *fit* ; 5 games per particle ; 30 particles, 100 iterations ; Range : [-10, 10]

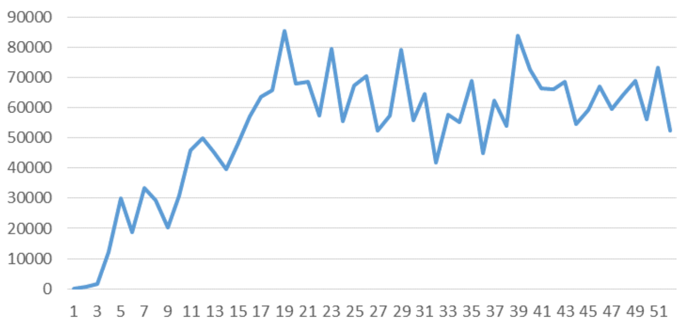


FIGURE 4 – *fit* ; 5 games per particle ; 30 particles ; 50 iterations ; Range : [-3, 3]

We realized that for f_1, the $rowsCleared$ would outweigh the other heuristics, $avgNumHoles$ and $avgTotalHeight$ would average to a single digit while $rowsCleared$ would average to around 500 (as shown in the graph above). From this point onward, we multiplied all heuristics except $rowsCleared$ by 500. By doing this, the other heuristics would be evaluated as more important than $rowsCleared$, hence the choice of ratios was imposed to have an appropriately scaled function, so $avgNumHolesRatio$ and $avgTotalHeightRatio$ values will range between 0 and 1 before being multiplied by 500. By doing this, we can make each heuristic be weighted on the same scale. At this point, we also added one additional game heuristics called deepest well for a total of 5 heuristics. Finally we ended up with the fitness function below :

$$fit(g) = RC + \frac{P_{max} - P_{avg}}{P_{max}}.500 + \frac{H_{max} - H_{avg}}{H_{max}}.500$$

$$+ \frac{C_{max} - C_{avg}}{C_{max}}.500 + \frac{R_{max} - R_{avg}}{R_{max}}.500$$

Where RC is the number of rows cleared, P is the number of holes, H is the maximum height, C is the numbers of Column transitions and R is the number of Row transitions

The performance of this updated fitness function is shown in the figure below.

One of the most crucial developments we made was realizing we must be underestimating the effect of randomness on each game. For example, a particle with good weights may evaluate poorly simply due to the order of pieces the agent is given. To account for this, we evaluated each particle by having it play five games with the same weight configuration, and then averaging the fitness score across those five games. This way, the fitness score would be a more accurate evaluation of a particular weight configuration. We ended up with the fitness function shown below :

$$fit = \frac{\sum_{i=1}^{5} fit(i)}{5}$$

This drastically improved our results, and we were able to see our algorithm learning. We get up to 30,000 lines at this point.

Modifications of PSO parameters :

We also tested increasing the number of particles from 30 to 90, and we found that with 90 particles, some particles settled into better local maxima than others. Due to the large number of particles, more particles would get stuck in a local maxima with a small fitness value. Therefore, we continued using 30 particles since it provides more consistent results. We noticed that some weights reached a value of 1 and stayed there for many of our runs. We began to increase the possible range of values that the weight can take on while maintaining the same particle velocity. Intuitively, doing this allows the algorithm to find more precise weight values because the particles are searching a larger space. We found that a range of [-3,3] gave us the best results. Alternatively, we could have decreased particle velocity and kept the range the same to obtain similar results. We found that increasing the range to [-10,10] made the space too large to search. Figure 2 is a graph of rows cleared as a function of iterations with range [-1,1], Figure 4 is a graph of a run with range [-3,3], and Figure 3 is a graph of a run with range [-10,10]. Figure 4 using the range [-3,3] gave us our best result — 622902.8 lines cleared on average across 5 games. The weights for our best result are shown in the table below (Figure 3).

Heuristic	Weight Values
Rows Cleared	-1.5027068
Total Height	-0.11106747
Maximum Height	0.43475921
Height Differences	0.80292475
Number of Holes	3.0
Deepest Well	1.7848895
Column Transition	3.0
Row Transition	0.87486663

0.4 Scaling up to Big Data

The choice of PSO is primarily due to its ability to scale up to big data. A swarm refers to the number of particles we specify to search the weight space — this architecture means the algorithm can run with subswarms (smaller groups of the original swarm) on multiple threads/cores or even machines and then share the optimal global positions found between the subswarms upon every iteration. Each subswarm would update their global position to be the best position found amongst all of subswarms, and the algorithm will repeat for the remaining iterations. Essentially, multiple subswarms act as one large swarm. Intuitively, this method should allow us to find the optimal weight strategy quicker than running PSO with one swarm since subswarms can be processed in parallel, while inside a single swarm, each particle has to be processed sequentially. Because a particle swarm can be split into multiple subswarms based on the amount of available computing resources, PSO is highly parallelizable. Large data, for example, require a vector space of possibly hundreds or thousands of dimensions, may take hundreds or thousands of particles to search. With more threads/cores and machines, this large swarm can be partitioned into smaller subswarms, and computing time can be theoretically reduced depending on the parallel processing overhead.

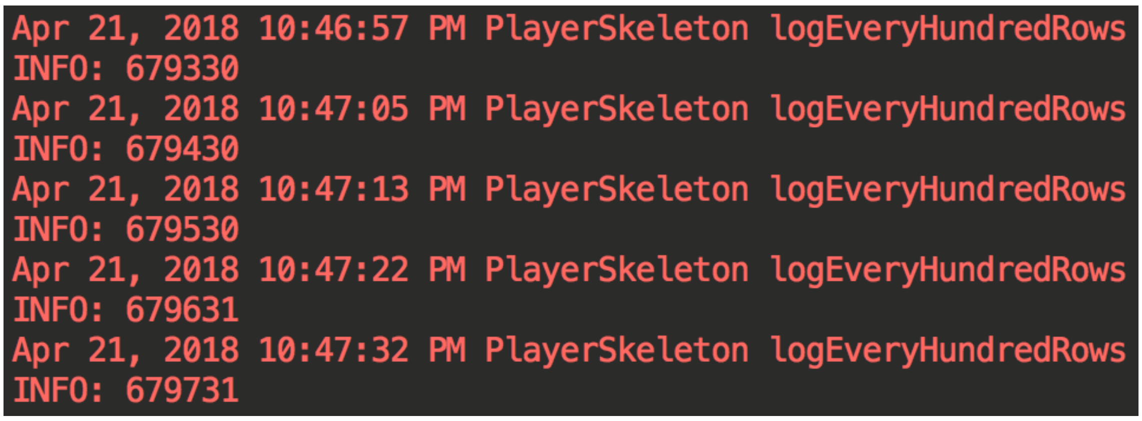
We implemented a multi-threaded PSO in Java. To test the multi-threading speedup factor, we ran a single-threaded and multi-threaded (10 threads) PSO with 6 iterations, 30 particles, 5 games per particle, for a total of 900 total games. The single-threaded run cleared 1774536.8 rows, completing 900 games in 121 minutes and 27 seconds. The multi-threaded run cleared 1779234 rows in 197 minutes and 27 seconds. Multithreading gave us a speedup of 62.58%.

0.5 Conclusion

We ran a game using a weight configuration on our heuristics that cleared an average of 227,297.4 lines across 5 games with 'one-piece lookahead' and achieved over 600,000 lines cleared resulting in a factor of 2.64. Based on this factor, although we did not have time to run our best weight configuration with 'one-piece lookahead', we believe we can clear at least 1,000,000 lines using the weight configuration that cleared 622,902.8 lines across 5 games. Our agent implementing our specified heuristics with their final optimized weight values and with 'one-piece lookahead' is shown below and has cleared around 650,000 lines [Figure 5, reference section]. This run has not yet reached termination but is likely to do so around 1,000,000 lines.

0.6 Resources

1. The Sunfire guide
   `hackmd.io/s/SJPWTzqs`
2. Comparing Particle Swarm Optimization method and Genetic Algorithm applied to the tidal farm layout optimization problem by O. A. L. Brutto, G. Sylvain, J. Thiebot, H. Gualous
   `https://www.researchgate.net/publication/308967731_Comparing_Particle_Swarm_Optimization_method_and_Genetic_Algorithm_applied_to_the_tidal_farm_layout_optimization_problem`
3. Swarm Tetris : Applying Particle Swarm Optimization to Tetris by L. Langenhoven, W. S. van Heerden, and A. P. Engelbrecht
   `https://ieeexplore.ieee.org/document/5586033/?reload=true&arnumber=5586033`
4. JSwarm-PSO Open Source Library
   `http://jswarm-pso.sourceforge.net/`



```
Apr 21, 2018 10:46:57 PM PlayerSkeleton logEveryHundredRows
INFO: 679330
Apr 21, 2018 10:47:05 PM PlayerSkeleton logEveryHundredRows
INFO: 679430
Apr 21, 2018 10:47:13 PM PlayerSkeleton logEveryHundredRows
INFO: 679530
Apr 21, 2018 10:47:22 PM PlayerSkeleton logEveryHundredRows
INFO: 679631
Apr 21, 2018 10:47:32 PM PlayerSkeleton logEveryHundredRows
INFO: 679731
```

FIGURE 5 – Agent including 'one-piece lookahead', the number after INFO : refers to the number of lines it has cleared

0.7 General Resources

We used these mostly for heuristics, but also for our general research into existing Tetris AI project methods.

1. Tetris AI — The (Near) Perfect Bot by Yiyuan Lee, April 2013
   `https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/`
2. On the Playing of Tetris
   `https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/`
3. Coding a Tetris AI using a Genetic Algorithm by Bai Li, May 2011
   `https://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/`
4. Applying Artificial Intelligence to Nintendo Tetris by MeatFighter, January 2014
   `http://meatfighter.com/nintendotetrisai/`
5. Tetris Artificial Intelligence by Tsai Wei-Tze, Yen Chi-Hsien, Ma Wei-Chiu, Yu Tian-Li, December 2013
   `cyen4.web.engr.illinois.edu/pdf/Tetris_AI.pdf`
6. Artificial Intelligence
   `https://web.engr.illinois.edu/~cyen4/pdf/Tetris_AI.pdfTetris`
7. What is particle swarm optimization ?
   `https://www.quora.com/What-is-particle-swarm-optimization-1`
8. El-Tetris, An Improvement on Pieree Dellarcherie s Algorithm
   `http://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/`
9. Improvement on a Tetris Agent
   `https://pdfs.semanticscholar.org/2d0d/eb544439e96f9f84fe1afc653bbf2f3bcc96.pdf`
10. Temporal Differences-Based Policy Iteration and Applications in Neuro-Dynamic Programming by Dimitri P. Bertsekas and Sergey Ioffe, August 1997
   `http://web.mit.edu/dimitrib/www/Tempdif.pdf`