CS3243: Introduction to Artificial Intelligence
Learning to Play Tetris with Big Data!

Group 09

Chia Le Jing (A0159971M), Conan Kian Jia Ren (A0139080J),
Lee Yan Hwa (A0140887W), Ngin Yun Chuan (A0139697H), Ong Qi Yong (A0135763B)

20 April 2018

1 Introduction

Our project aims to create an utility-based Tetris agent whose goal is to maximize the number of lines cleared in a game of Tetris. Our agent uses a heuristic function that considers multiple features of the Tetris game state, and the best heuristic function was learned through Tetris game simulations with a Genetic Algorithm (GA). Our agent achieved a 100-game average of 26.7 million rows cleared with a peak of 124.6 million rows cleared in a single game. In the process, we have developed a Tetris agent learning approach that has been carefully optimised to produce better agent performance within a shorter amount of time.

2 Agent Design

Our agent uses a simple one-layer local state search. At each turn of a game, the resultant game state of all legal moves of the current falling Tetris block will be evaluated with a heuristic function. The move resulting in the highest evaluated heuristic value will be taken.

This heuristic function is a linear weighted sum of the following 8 Tetris game state features:

1. ϕ_1: *TotalHeight* - Sum of the heights of all columns. Height is the position is the highest filled cell in a column.
2. ϕ_2: *Bumpiness* - Sum of the differences between consecutive column heights.
3. ϕ_3: *Wells* - Sum of the squares of the depths of valleys, which are columns with depth lower than its flanking columns.
4. ϕ_4: *NumFilledCells* - Number of filled cells in the field.
5. ϕ_5: *EdgesTouchingWall* - Number of filled cells touching the left and right game boundaries.
6. ϕ_6: *MeanHeightDiff* - Average of the differences between each column's height and the average column height.
7. ϕ_7: *HolesVolume* - Number of empty cells with least one filled cell above them.
8. ϕ_8: *ColTransitions* - Number of empty cells that are adjacent to a filled cell on the same column.

It would take in game state S to have the form:

$$h(S) = \begin{cases} \sum_{i=1}^{8} w_i \phi_i(S) & \text{if } \neg hasLost(S) \\ -\infty & \text{otherwise} \end{cases}$$
$$\text{where } w_i \in [-1, 1],\ \phi_i \in [0, 1]$$

A value of $-\infty$ is assigned to a game loss as it must be a worse state than any non-losing state in which potentially more rows could be cleared. Features are normalised to strictly range from 0 to 1.

3 Learning Approach

The use of GA as a global search technique to learn heuristic functions of utility-based Tetris agents has been successful and is well-documented [1] [2]. Therefore, we looked to utilize GA to search for the optimal set of weights $\vec{w}^*$. For the comparison of the performances of multiple sets of weights, we defined the performance measure of a set of weights $\vec{w}$ to be the average rows cleared over 100 Tetris game simulations, by an agent using that set of weights in its heuristic function.

We hand-coded our own GA Java class *GeneticAlgorithmLearner* that gives us full control over the inner workings of the GA. After experimentation and research, we settled on the following GA properties that worked well for us:

- *Population size* of 100 individuals (i.e. sets of weights) that are randomly initialised

- *Fitness function* defined as the average rows cleared over 10 Tetris game simulations

- Elitist selection of 80% of the population as the next generation, using 2-way *tournament selection* with a single winner, while killing off the remaining 20%

- Uniform random selection of surviving elites as parents

- *Weighted average crossover* to produce offsprings:

$$\vec{w'} = fitness(\vec{w_{p1}}) \cdot \vec{w_{p1}} + fitness(\vec{w_{p2}}) \cdot \vec{w_{p2}}$$

- Mutation of offsprings with 10% chance of having one of its weights altered by a gaussian mutation operator:

$$w_i \leftarrow N(w_i, \frac{2}{10})$$

- Restarting runs when there is no new best individual found after 50 generations

These properties appear to sufficiently balance between training time and potential to converge to the global maxima.

Our definition of the fitness of an individual in our GA is the 10-game average of rows cleared. Considering the high variance of rows cleared for a Tetris agent arising from the randomness of generated blocks sequences, the 10-game average serves as a decent indicator of performance while not being as computationally intensive as our standard performance measure of 100 games.

Rather than have each generation be completely formed from breeding, we purposefully allowed for elitism [3], which ensures that best-performing sets of weights are preserved after GA's crossover and mutation phase. This follows from our observation that minor tweaks of the weights can lead to huge changes in performance.

To further improve our GA's ability to discover the best set of weights, we made 3 unique modifications to the typical GA search methodology.

3.1 Forward Feature Selection

From our research, we had initially identified a total of 15 features that have been used in heuristic functions of high-performance Tetris agents [1] [2] [4]. When we tried learning with 15 features, we discovered that training results were very poor, although there was one particular run of GA in which we were able to get a good set of weights that reached an 100-game average of 2.6 million rows cleared.

We theorised that the search space was so big such that it would take an extremely long training period for our GA to find an optimal set of weights. Thus, we decided to reduce the size of our feature set to only 8 features using a greedy forward selection algorithm.

To do so, we first did learning of heuristic functions with just a single feature $\{\phi_1\}, \cdots, \{\phi_{15}\}$ and chose the feature that enabled the learning of a heuristic function that cleared the most rows (deemed as the most important). We then greedily added more features to the feature set, one by one, such that they improved the number of rows cleared by the biggest margin at each step. To heavily cut the computation time required for the evaluation of whether a set of features was more important than another, we used the number of rows cleared for a 10-row Tetris game as a proxy to that for a 20-row Tetris game.

As a result, we arrived at the abovementioned 8 features used in our subsequent learning, listed in descending order of importance: ϕ_1 *TotalHeight*, ϕ_2 *Bumpiness*, ϕ_3 *Wells*, ϕ_4 *NumFilledCells*, ϕ_5 *EdgesTouchingWall*, ϕ_6 *MeanHeightDiff*, ϕ_7 *HolesVolume* and ϕ_8 *ColTransitions*.

3.2 Hill-Climbing of Best Individual

In response to our lackluster GA performance, we made the observation that our GA should be made faster in converging towards a local optima upon discovery of a new fittest individual, following crossover and mutation. After some research, we found a proposal of a variant of GA that additionally does hill-climbing of only the best individual in each generation [5], and we adopted this idea into our GA implementation. This modification attempts to marry both the strengths of hill-climbing (in exploiting a local optima) and GA (in exploring the search space for a global optima) [5], and proved to be extremely instrumental in the discovery of our final best set of weights.

3.3 Seeding of GA Population

The last search optimisation was the enabling of the seeding of the initial GA population by passing of additional parameters to *GeneticAlgorithmLearner*. It greatly increases versatility of our learner by allowing us to resume from a known set of fit individuals, experiment with manual adjustments to weights, and mix various fit individuals across previous GA runs in hope of produce even fitter individuals during GA's crossovers. In particular, we used the weights from previous runs of GA to optimise our results and search time, rather than searching from scratch. This provides us a tool to leverage historical data and data from various sources to improve our learning performance.

3.4 Alternative Learning Approaches

We have also experimented using Particle Swarm Optimisation (PSO) [4] and Simulated Annealing (SA) as alternative global search techniques and compared it with our customised GA. Both are simpler algorithms with less parameters to tweak as compared to GA. However, SA returns extremely poor results for a problem like Tetris with a large search space. Standard PSO faces similar problems as the standard typical GA, with premature convergence due to a decrease of particle velocity, especially in high dimensional search spaces like Tetris. It then leads to fitness stagnation of the particles.

4 Speed Optimisations

We realized that as our agent started to perform better, training times begun increasing exponentially on our personal machines. Rather than procuring powerful computing clusters, we thought it was more meaningful to optimise our code to speed up the search for, and evaluation of, sets of weights for our agent's heuristic function, making sure to leverage parallelisation on multi-core machines as much as possible. After the following changes, our code was noticeably able to crunch through more, and longer, Tetris game simulations given a fixed amount of computing resources.

4.1 Faster Feature Computations

Using *VisualVM*'s Java application profiling capabilities, we observed that significant computational time and memory were spent on feature computation in *GameState* (our custom abstraction of the Tetris game state), as well as class instantiations.

As such, we reduced the time complexity of computing each feature to either $O(ROWS)$, $O(COLS)$ or even $O(1)$ by maintaining various derived state variables on every move. We also avoided creating new instances of *GameState* for each move; instead, we reused a single instance over all searches by implementing a method that copies all state and derived state variables between *GameState* instances.

Speed Before (Rows/s):	27,113
Speed After (Rows/s):	68,794
Speedup Factor:	2.54x
Conditions: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2801 Mhz, 4 Core(s), 8 Logical Processor(s)	

4.2 Parallelisation of Game Simulations

During learning, we spawned multiple threads to match the number of CPU cores, for concurrent Tetris game simulations.

Speed Before (Rows/s):	45,226
Speed After (Rows/s):	170,446
Speedup Factor:	3.77x
Conditions: Intel(R) Core(TM) i7-4790K CPU @ 4.40GHz, 4 Core(s), 8 Logical Processor(s)	

The speed-up factor observed is rather close to an ideal speed-up, suggesting that our learning method could scale almost linearly with an increase in the number of CPU cores.

5 Results

Our best performing set of weights, and its statistics over 100 games, are as follows:

ϕ_1	-0.3907454899138355
ϕ_2	-0.7076471959178626
ϕ_3	-0.9999999978876627
ϕ_4	0.34814812107690407
ϕ_5	0.005918216014713197
ϕ_6	-0.506446236591784
ϕ_7	0.010013842808179697
ϕ_8	0.00000000304563703249

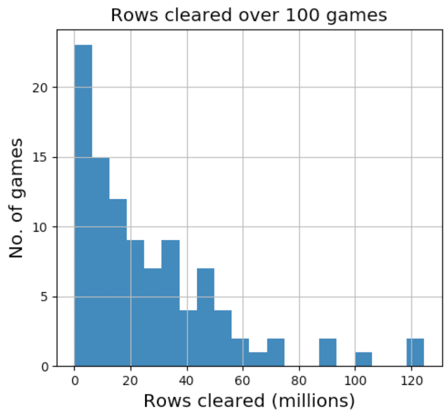

Figure 1: Rows cleared over 100 games

Metric	Rows Cleared / Game
Mean	26,682,778.9
Median	18,832,659
Min	32,008
Max	124,595,264

This set of weights corresponds to a Tetris game strategy that discourages formation of wells, an uneven surface and sum of total height of columns, but encourages filling of cells, the formation of holes (slightly) and edges touching the left and right game boundaries (slightly).

6 Conclusion and Future Work

In this paper, we presented the design of our Tetris agent and our learning methodology that utilises GA. In our methodology, we have heavily customised our GA with the addition of forward feature selection, hill-climbing of the best individual and seeding of the initial population. We also optimised feature computation and injected parallelisation in our code to significantly speed up Tetris game simulations. Our learning methodology has produced a Tetris agent that has remarkable performance, and the methodology could potentially learn and create even smarter Tetris agents on powerful computing clusters.

Our learning methodology can be extended in various ways. With more computational resources, forward feature selection could be performed on a larger pool of identified Tetris game state features (i.e. more than 15) and a larger set of important features (i.e. more than 8) could be selected for learning, raising the expressiveness of the heuristic function and maybe unlocking a better global optima. On the other hand, to reduce the training times, more experimentation can been done on reducing the size of the Tetris board (e.g. 18 rows x 10 columns) used for training so that training could be made faster and yet retain its effectiveness. Additionally, like how Tetris game simulations could be executed across threads due to their independence, parallelisation across multiple machines in a distributed system can also be explored.

Besides that, we could further explore the use of other search algorithms and their variants for the training of Tetris agents and possibly develop a hybridised learning approach that leverages the different strengths of these algorithms. One of such algorithms is reinforcement learning with cross-entropy [6].

References

[1] Yiyuan L., *Tetris AI - The (Near) Perfect Bot*, 2013 https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

[2] Lucky's Notes, *Coding a Tetris AI using a Genetic Algorithm*, 2011 https://luckytoilet.wordpress.com/tag/tetris/

[3] Baluja S. & Caruana R., *Removing the Genetics from the Standard Genetic Algorithm*, 1995 https://www.ri.cmu.edu/pub_files/pub2/baluja_shumeet_1995_1/baluja_shumeet_1995_1.pdf

[4] El-Ashi A. K., *El-Tetris - An Improvement on Pierre Dellacherie's Algorithm*, 2011 http://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/

[5] Wan, W. & Birch J. B., *An Improved Hybrid Genetic Algorithm with a New Local Search Procedure*, 2013 https://www.hindawi.com/journals/jam/2013/103591/

[6] Thiery, C. & Scherrer, B., *Improvements on Learning Tetris with Cross Entropy*, 2009 https://hal.inria.fr/inria-00418930/document