**CS3243 Term Project – Let’s Play Tetris**

|  |  |  |
| --- | --- | --- |
| **Group 10** | Cheong Ee Ju Martin (A0087013M) | Law Wen Yong  (A0081205U) |
| Nguyen Hoang Thanh Duc (A0090551L) | Worapol Ratanapan  (A0074997E) | Wong Shun Heur, John (A0088015H) |

**Approach**

Since Tetris is a game with a single goal (clear the most number of lines without filling the board) and the move to make depends on the board’s current state, a utility-based agent was implemented by our group.

There are numerous variations to how utility can be calculated, since we felt there was not much innovation that could be created in this aspect, our group decided to reference existing features[[1]](#footnote-1) and innovate on other aspects.

|  |  |
| --- | --- |
| **Features** | **Description** |
| Rows Cleared | Number of rows cleared by a move |
| Landing Height | The height which a piece lands + (height of piece) / 2 |
| Row Transitions | Number of transitions between a filled and empty cell in a row |
| Column Transitions | Number of transitions between a filled and empty cell in a column |
| Number of Holes | A hole is defined as an empty cell with at least one filled cell above it |
| Well Sum | A well is defined as an empty cell with filled cells on its left and right |

**Feature Weights**

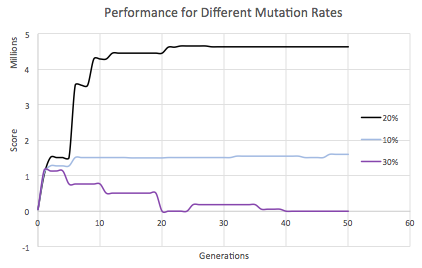
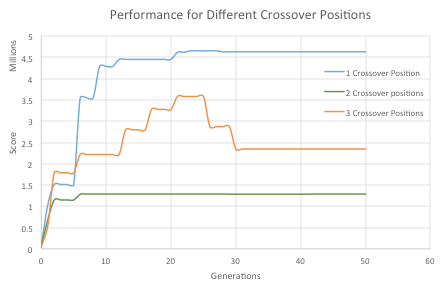
The performance of the utility agent would be very limited if its features were not appropriately weighted. To find the appropriate weights, learning agents were created. The learning agent would generate a set of weights, based on an internal algorithm, and an evaluator would test the performance of these weights; the results were then returned to the learning agent so that it could repeat and improve. Since the pieces are randomized, the average across 3 games was taken when evaluating performance.

**Learning Algorithms**

For the learning agent’s internal weight-generation algorithm, there are again many available algorithms to choose from. Since there are an infinite number of weight combinations, a local search algorithm is necessary. Our group decided on using the genetic algorithm and particle swarm algorithm, as they were appropriate for this multi-dimensional search space.

**Genetic Algorithm**

|  |  |
| --- | --- |
| Population Size | Initial trials with a population of 200 quickly revealed that the algorithm would take too long to run. As a result, we reduced the size to 50 chromosomes |
| Crossover | We implemented a position-based crossover, where chromosomes were merged at either index 1, 2 or 3 |
| Mutation | Mutation rates were varied at 10%, 20% or 30%. At each round, we randomly selected only one weight to mutate. This was to reduce the risk of killing off a strong candidate, while allowing growth for weaker candidates |
| Reproduction | For each generation, the worse 60% of the previous generation was replaced by newly generated chromosomes |



A crossover position of 1 is best possibly because interchanging 2 or 3 features alters the combination too much and reduces the effectiveness of the produced chromosome. While a mutation rate of 20% is best probably because 10% introduces too little diversity while 30% produces too much fluctuation and kills off strong chromosomes.

**Particle Swarm Optimizer**

Particle swarm is another algorithm that originates from nature. For this algorithm, a set of particles is first created, each with random positions (-100, 100) and random velocities (-200, 200) in 6 dimensions. A particle’s position in one dimension is mapped to the weight of one feature of our utility function to evaluate performance. For each iteration, every dimension of every particle was updated by the formula: . Each particle’s velocity was also updated by the formula[[2]](#footnote-2):

3 main parameters were varied to optimize the algorithm for learning the feature weights.

|  |  |
| --- | --- |
| Inertia weight (*w*) | The smaller the value, the faster the particle reaches a local maximum. It was found that a value of 0.7 was best. A value of 0.1 led to the particles reaching a low local maximum too quickly with a smaller chance of better values being discovered. A value of 0.95 performed badly as well, because the particles progressed too slowly. |
| Personal weight () | The higher the value, the less affected a particle is by the global best position. |
| Global weight () | The higher the value, the more a particle would tend to the global best position. It was found that having equal values of = 1.4 and = 1.4 was best. If was too large, particles stagnated at their local best position. Conversely, if were too large, the particles would get stuck around the current best particle’s local maximum. |

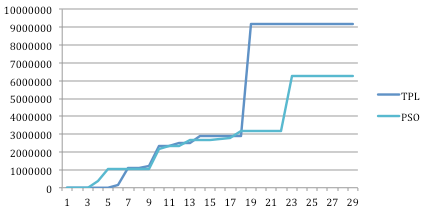
**Tetris Premier League**

A method we thought of to improve performance was to combine the particle swarm optimizer and the genetic algorithm, to mutate particles in a bid to further help the particles escape local maximums.

In our Tetris league, each team consists of a set of players; and each player’s value is mapped to a weight of a feature of our utility function. Teams play in a season consisting of n iterations; at the end of the season, the worst performing teams are eliminated. In our league, the next season progresses with new teams being created by merging existing teams; these new teams would replace the eliminated teams. Players are not stagnant, so some mutation of their values takes place as well with the use of PSO velocity.

Since the mutation in PSO is directional with PSO velocity unlike the random mutation in GA, it is safer to eliminate only 10% of the particles because there is a large chance of getting trapped in the local maximum if more particles are crossover with almost the same parameters as the global best parameter.

It was found that a season of length n = 3 iterations was ideal, and merging the parameters we had learnt in the previous two algorithms, it was found that the league algorithm improved performance.



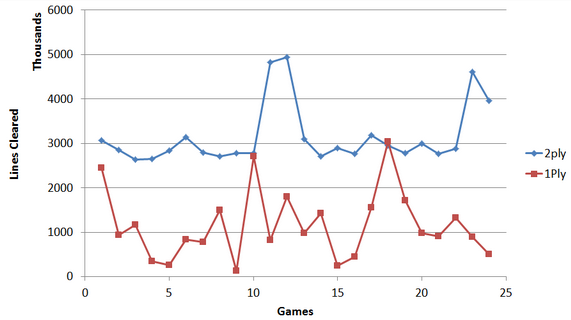
**Looking-ahead**

Trying new algorithms and testing different parameters is difficult, but we still wanted to extract more performance. As such, we explored including a single move look-ahead function into our algorithm.

A full exploration of all possible moves for the second stage would be computationally intensive, thus we utilized two techniques to speed up the decision-making.

Firstly, we found that we could explore the second state of only the 3 best moves, and performance was still better than without looking-ahead. Secondly, when exploring the second state of the 3 best moves, we cached the best move for the subsequent state, so that this would not have to be evaluated again for the next move.

Looking-ahead produced better performance, as the agent was able to consider a more comprehensive set of options. The recorded performance was more stable as well, because when the agent did not look-ahead, it sometimes performed quite poorly.



**Conclusion**

The final weights learned from the league algorithm are as in the Java program. Given more time we would want to run the league algorithm together with the look-ahead function to see whether it can create even better weights. We would also want to take the average of more games to increase accuracy.

Through this project we gained a deeper understanding of the implementation of learning algorithms. We also learned about the difficulties in finding optimal parameters for the algorithm. Lastly, we also came up with methods to reduce the computationally expensive task of looking-ahead.

1. Feature functions were referenced from El-Ashi (Dec 2012). El-Tetris – An Improvement on Pierre Dellacherie’s Algorithm. Retrieved from: http://ielashi.com/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/ [↑](#footnote-ref-1)
2. In this report we only detail the main parameters of the algorithm, the full explanation and formula is referenced from Angeline, Peter J., Carlise, and Anthony J., et al. (Jan 2006). PSO Tutorial. Retrieved from http://www.swarmintelligence.org/tutorials.php [↑](#footnote-ref-2)