CS3243 Tetris Project

Varun Gupta Gao Bo Advay Pal Chang Chu-Ming
Herbert Ilhan Tanujaya

13-04-2017

1 Introduction

Tetris is likely one of the world's most famous and popular games. In this report, we describe how we devise an agent to play the game of Tetris. We use an agent that greedily picks the best possible next state from a given state, by using a heuristic function to approximate the value of a state. To train our heuristic function, we use a novel algorithm that is a combination of the well known Genetic Algorithm (GA) and Particle Swarm Algorithm (PSO). Our agent manages to clear AVERAGE million lines on average with a max of MAX million lines, demonstrating that our algorithm is effective.

2 Agent Strategy

Our agent uses a linear weighted sum of features as the heuristic function for a given state. Given a state and a piece, the agent computes the heuristic value for all possible next states, and then greedily picks the next state with the maximum heuristic value.

$$\sum w_i \times f_i(s)$$

3 Features

We used the following features for our heuristic function:

- Altitude Difference: The difference between the height of the highest column and the height of the lowest column
- Number of Columns With Holes: The number of columns with holes, where a hole is defined as an empty square directly beneath a filled square
- Height of the highest column
- Number of holes in the entire board
- Number of Wells: The number of columns that have a height less than that of the 2 adjacent columns
- Rows cleared: The number of rows cleared for that particular move
- Total Column Height: The sum of the heights of all the columns
- Total Column Height Difference: The sum of the difference of heights between adjacent columns
- Column Transition: Number of adjacent squares in a column with opposite parity (where parity is defined as either full or empty)
- Deepest Well: Height of the lowest column

- Row Transition: Number of adjacent squares in a row with opposite parity (where parity is defined as either full or empty)
- Weighted Block: Sum of value of every square, where a square's value is equal to the row it is in (numbered from the bottom)
- Well Sum: Using the same definition of wells as above, this sums up the depth of every well

While running our training algorithms, we noticed that some features were more important than others. In particular, the algorithm assigned highly negative weights to Column Transition and Well Sum, while assigning highly positive weights to Number of Wells. This was slightly unexpected as we thought that Rows Cleared would have the largest positive weights, to predispose the algorithm towards clearing more rows. However, the weight for this heuristic varied wildly between positive and negative, indicating that perhaps sometimes the algorithm preferred to not greedily pick moves that were clearing rows, but rather chose to maintain an even surface at the top.

4 Our Algorithm

Our algorithm consists of a combination of a genetic algorithm and particle swarm optimization (PSO) algorithm. We run both of these algorithms on different islands with population sizes of a 100 each. Each member of our population is a heuristic (set of weights). The key idea is that both of the algorithms run individually, but after each generation, we copy the 10 best heuristics on each island to the other one.

We will first describe the working principle of the Genetic and the PSO algorithms. Subsequently, we will discuss the rationale behind why we chose to juxtapose these two different algorithms in our search for the best heuristic, as well as how we did so.

Genetic algorithm
For this algorithm, a crossover is done by taking two parent heuristics, and for every feature selecting the weight from one of the parents randomly. A mutation is defined as multiplying the current weight of a feature with a value normally distributed with a mean of 1 and a standard deviation of 1.
Each generation of our genetic algorithm consists of the following sequence of steps

1. Evaluate fitness of each set of heuristics
2. Keep top half of population and cross-over the rest
3. Random mutation of each feature of crossed-over heuristics with probability of a tenth
4. Migration of a tenth of the population to PSO island

We keep the top half of the population to ensure that each set of heuristics that perform well will remain within the population. The mutation introduces some randomness to the genetic algorithm to avoid being trapped within a local maximum.

PSO algorithm
The PSO algorithm treats every heuristic as a point in 13-dimensional space. This is the position of the heuristic. Each heuristic also has a velocity, a vector in the 13-dimensional space, which is added to the position to get new population members. Each generation of our PSO algorithm consists of the following sequence of steps

1. Set velocity of each member to a linear weighted sum of the old velocity, the personal best of that member, and the global best
2. Get new position of each member by adding its velocity to the current position
3. Evaluate fitness of each set of heuristics
4. Migration of a tenth of the population to PSO island

For PSO, it is almost certain that every member of the population is mutated in every generation since weighted velocity is very unlikely to be 0.

Rationale behind combination of algorithms
We ran PSO and GA individually. In doing so, we realised that PSO was in general not doing very well, but that it sometimes made a leap and jumped by almost an order of magnitude in terms of lines cleared. On the other hand, Genetic seemed to keep getting better, but only quite slowly. From these experimental results, we hypothesised that Genetic may have been doing more of exploitation, finding the maximum in the local area, while PSO may have been doing more of exploration, looking for good solutions all over the search space. Hence we thought that perhaps if we combined the two, PSO could lead Genetic to better areas, which Genetic could then drill down into. This would allow us to achieve a more balanced trade-off between exploitation and exploration, which we hoped would make our algorithm more efficient.

5 Further Optimisation

The hyperparameters for both the genetic algorithm and the PSO algorithm need to be optimized for our integrated solution to learn at an optimal rate. Given limited time, we decided to focus our attention on the hyperparameters of the PSO algorithm. The rationale behind this is that while the genetic algorithm learns at a steady rate, the PSO algorithm tends to make large jumps in progress and can explore larger swaths of the search-space in a shorter time. We felt that if we successfully could optimize the PSO algorithm, it would be able to find a solution close to the global maximum and share this information with the genetic algorithm. The genetic algorithm would then refine the solution further.
One of the issues that we observed is that even with the same set of hyperparameters, different initializations of random variables such as mutation can lead to different rates of learning. Thus, we ran multiple jobs in parallel and averaged their results to determine which set of hyperparameters work best.
The hyperparameters of the PSO algorithm consist of TODO and TODO, the constants which determine the effect of the global and personal bests on each iteration of velocity change; and TODO, the constant of inertia. Typically[CITE], TODO and TODO are both set to 2.0, which leaves TODO to be optimized. A high TODO will favour exploration, while a low TODO will favour exploitation. Based on the observations of Shi and Russell [1], we decided to first test values of TODO between 1.2 to 0.6. While initializations with low TODO generally performed better, the results were not conclusive. Thus, we decided that instead of a fixed TODO, we could vary it based on how close the population is to the global maximum. This is estimated by taking TODO of the average fitness scores of the population. The formula is as follows:
TODO
We tested different values of k by running multiple jobs for an hour and comparing the average of the fitness scores. Through experimentation, we found that the algorithm performs best when k is 0.2.[Some more detail?]
[1] Y. Shi and R. C. Eberhart. Parameter selection in particle swarm optimization. 1998,InEvolutionary programming VII(pp. 591-600). Springer Berlin/Heidelberg.

6 Experiments and Analysis

Diagram 1: Learning **TODO**
Diagram 2: Performance of agent **TODO** Wondering whether performance should be here and in experimentation too??

Experimentation
1. Architecture specifications on which the algorithm was implemented - E5-2690 v3 @ 2.60GHz
2. The time taken to train - Using a version of our algorithm that used 2 threads, we were able to get to a maximum of around 4 million lines cleared in around a day. Running a parallelised version (see below) we got to a maximum of around 18m in (so and so) hours.

3. The performance of the agent **TODO**
Analysis **TODO**

7 Scaling to Big Data

In order to scale to Big Data, we considered how our algorithm performed in the presence of multiple cores. We found that parallelising our algorithms through the use of multithreading lead to a linear speedup, demonstrating that our algorithm scaled very well in the presence of multiple cores. We attribute this to the fact that both Genetic Algorithms and PSO algorithms are embarassingly parallel, which is why we were able to achieve an optimal speedup quite easily.

We parallelised our algorithm by running the PSO and Genetic islands on different cores, and further parallelised it by calculating the fitness of each population member on different cores. We did this as the major computational cost the algorithm was incurring came from the calculation of the fitness values, since that involved running entire tetris games.
To demonstrate the effect of parallelisation, we recorded the speedup of our algorithm when run on multiple cores. We started both algorithms as two separate jobs on single machines in the NSCC cluster (which provide 12 cores each) at the same time, and allowed them to run for around 20 hours before comparing the results. We used the total number of lines cleared in the entire time span as a rough estimate of the total amount of CPU time used. Our results showed that the single-threaded algorithm cleared a total of 158182993 lines, while the multi-threaded version cleared a total of 1899941699 lines. This shows that parallelisation provides a speedup of 12 times, which is optimal since each machine has 12 cores.

8 Conclusion

9 References