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 genetic algorithm and particle swarm optimization (PSO) algorithm. We run both of these algorithms on populations of 100 sets of weights. The algorithms are run on what we call islands. The 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 set of heuristics.

Genetic algorithm
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. The probability of mutation is set at ten percent.

PSO algorithm
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

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 sometimes it made a leap and jumped by almost an order of magnitude in terms of lines cleared. On the other hand, Genetic seems to keep getting better, but only quite slowly. From these experimental results, we gathered that Genetic was doing more of exploitation, finding the maximum in the local area, while PSO was carrying out exploration, looking for good solutions all over the search space. Hence we thought, that perhaps if we combined the two, then PSO could lead Genetic to better areas, which Genetic then drills down into.

5 Experiments and Analysis

Diagram 1: Learning Diagram 2: Performance of agent
Experimentation 1. Architecture specifications on which the algorithm was implemented 2. The time taken to train 3. The performance of the agent
Analysis

6 Scaling to Big Data

We parallelised our algorithm by running the PSO and Genetic islands on different cores, and futher by calculating the fitness of each population member on different cores. We then ran our algorithm on the NSCC cluster, which provides 12 cores per compute node.
To demonstrate the effect of parallelisation, we compared the speed performance of two versions of the algorithm, single-threaded and multi-threaded version, sharing the same set of parameters. We started both algorithms as two separate jobs on the NSCC cluster 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 the baseline for comparison 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 about 12 times speedup, on the setup we used, demonstrating that we can achieve linear speedup by parallelising our algorithm.
The major computational cost we were incurring was in the calculation of fitness values for each member of the population, as this involved running an entire tetris game for them. By parallelising this, we achieved significant speedup, reducing the computation time by an order of magnitude.
Talk about parallelising algorithm. Mention MPI Get speedup

7 Conclusion

8 References