Dead Man's Tetris

Joshua Cheung (A0134910N)
Ajay Karpur (A0132198A)
Frederic Lafrance (A0134784X)
Iain Meeke (A0132729A)

April 18, 2015

1 Introduction

We considered three different approaches for this assignment: neural networks, particle swarm, and genetic algorithms. We chose to implement a genetic algorithm, as we felt that it would be easier to implement and to parallelize (i.e. with multiple cores). Neural networks seemed promising, but it is a supervised learning method, and we did not find enough training data to use. We considered using video data of world-record Tetris games, annotated using Amazons Mechanical Turk service. However, we decided that this would be impractical; we would have to slice the videos and review the annotations ourselves. This report describes our chosen heuristics, our learning method, our results, and our strategies to scale our method.

2 Heuristics and Evaluation Function

We decided that it would be best to have a low number of heuristics, for two reasons. First, it is easy to reason about a few heuristics and to compute them by hand for testing purposes. Second, having fewer heuristics lowers the memory footprint and the computation time of fitness values, which is important in the context of scaling the algorithm to large numbers of Tetris games.

Heuristics are evaluated by qualities of a board state after a move to be considered is performed. The most important indicator of a high performing agent is one that produces the highest score before losing, or highest number of cleared rows (without multipliers), which is the heuristic we want maximized. Other negative qualities of a board state were holes (empty squares in the board that have filled squares in the same column above them) , sum of the heights of the columns, and bumpiness (the total absolute differences in height between adjacent columns). [2] These are anticipatory qualities that increase the chance of a lost game (and are generally regarded as poor play), so we decided to include them as heuristics. To evaluate the desirability of a given board, we use a weighted sum of these heuristics. At every step, we simply choose a move by testing all legal possibilities and picking the one that results in the most desirable board. For our AI player, the problem thus becomes: what set of weights will clearly differentiate between good boards and bad boards? This is where the learning method comes in.

3 Learning Method

We decided to use a genetic algorithm to mutate and improve an initial set of weights, converging at the set which provides the highest fitness function. We maintain a population of individuals who play a given number of games using the heuristics above and a personal set of weights to rank the boards. The fitness of an individual is defined as the aggregate score over all its games.

Once all of the individuals have played their games, we order them by fitness and select the elite (the top elitism%, where elitism is a parameter to the algorithm), to carry over to the next generation. The process of evolution is done by crossover and then a random mutation. The crossover operates on two parent individuals and creates two child individuals; for each feature, we flip a coin to determine whether the first parent will give its feature to the first or the second child (the second parent gives its feature to the other child). We thus repeatedly choose two random and distinct individuals from the elite, and apply the crossover to generate two new individuals. This process is repeated until enough new individuals have been generated.

The new generation is then put through a mutation process. Every individual has a small chance (a few percent at most) of ”mutating”. When an individual mutates, a random feature is changed to a new random value between -1 and 1. The mutation operation is used to prevent the individuals from getting stuck in a local maximum, wherein most top individuals have very similar weights. By randomly mutating individuals, we allow ”sideways” moves through the search space.

This sequence of events (game playing, crossover and mutation) is known as a generation. Our learning method is simply to evolve individuals through a large number of generations, and stop once the weights seem to converge.

4 Experimental Results

We experimented with the chosen algorithm by varying all the parameters that affect the evolution of generations. These were the heuristics chosen, the mutation rate and process, the crossover process, population size and number of games played per individual. Below are the parameters which gave us the best performance:

- population size = 1000
- mutation rate = 5%
- 20 games each individual plays per generation
- elitism of 25 individuals

To speed up the learning process, we reduced the board height from 21 rows to 11 with the same width, anticipating a drop in fitness function while maintaining a large enough board to mimic performance on an actual board size. We also implemented strategies to ensure the learning progress was efficient over a large amount of Tetris games, discussed under scaling considerations.

5 Observations and Analysis

Many observations were made about how the genetic algorithm parameters affect the learning outcome and which parameters are ideal for our application, especially the elitism percentage on the population size. First, the population size had to be sufficiently large as a search space but not so large to slow the learning progress with no effective progress. Within a large population size, the elitism percentage has to be small enough to be selective. The initial elitism percentage was 50% and we were unable to see learning progress on different sets of the other parameters, as the performance fluctuated with no upward trend due to a high elitism percentage including individuals with low performance to breed. Additionally, too low of an elitism rate would cause inbreeding among the few individuals survived, preventing variations of the succeeding generation.

6 Scaling considerations

The following are ideas that we considered to ensure that our system would be able to scale to large amounts of Tetris games, including novel ways of monitoring progress to change parameters that speed up learning.

6.1 Learning cessation

One important part when learning a problem is knowing when no more progress is made and deciding to quit learning. This is a problem that has been studied in particular with neural networks. In [1], Shultz et al. introduce two parameters to control learning cessation: threshold and patience. Threshold specifies what is considered progress from one generation / learning cycle to the next, in terms of absolute difference between scores. Patience indicates how many consecutive generations without progress we are willing to tolerate before stopping the algorithm. In our case, we would measure threshold against the best individual of each generation. As long as the best fitness would vary by a value of at least threshold from one generation to the next, we would consider this generation to have progressed. After patience generations without progress, we declare that the algorithm has stalled and stop it. This is useful when scaling as it can prevent us from doing unproductive work (i.e. running generations even though no improvement occurs). Hence, we can potentially save a lot of time when learning without compromising the quality of our best individuals.

6.2 Parallelism

Genetics algorithms have one part that can easily be parallelized, which also happens to be the most expensive in our case: calculating the fitness values for each individual. As detailed before, each individual of a generation plays 15 games, and its fitness is the sum of the scores for those games. Clearly, every fitness computation may run in parallel. We can thus start a number of worker threads and have each of them take one individual, compute its fitness and repeat the process until all individuals have had their fitness computed. Since every generation typically contains at least tens of individuals, we expected large speedups in this part of the algorithm.

We implemented threading using a Java ThreadPoolExecutor. We then uploaded our code to a 36-core compute-optimized Amazon EC2 instance to train for some time.

6.3 Variable mutation rate

We came up with the novel idea of modulating the mutation rate based on how quickly the algorithm is learning. If the fitness has not changed significantly for several continuous generations i.e. flatlining, we increase the mutation rate to break out of local maxima as fast as possible. If we notice that the fitness is oscillating or otherwise behaving in a volatile manner, we decrease the mutation rate to enable steady increase without overshooting.

We implemented this using a moving average (ie. smoothing), but did not see particularly useful results. We therefore chose to discard this method for our actual training phase, despite its potential as an interesting idea.

Appendices

1 Code organization and documentation

The PlayerSkeleton file is organized as follows:

- Internal class StateEx: Inherits State, and contains some additional methods to test moves against the current state without modifying it.
- Internal class Individual: Used throughout the genetic algorithm implementation. Contains a set of weights for the heuristics and methods to play a game using the strategy described above.
- main method: Starts the program. If given the switch -g, the program attempts learning a new set of weights. Otherwise, it runs a game with a hardcoded set of weights that we found by experimentation.
- Genetic algorithm methods: These methods implement the core functions of the genetic algorithm, namely applying the mutation and crossover operators, computing the fitness of each individual and running each generation.

References

[1] Thomas R Shultz, Eric Doty, and Frédéric Dandurand. Knowing when to abandon unproductive learning. In *Proceedings of the 34th Annual Conference of the Cognitive Science Society. Austin, TX: Cognitive Science Society*, pages 2327–32, 2012.

[2] Lee Yiyuan. Tetris ai: The (near) perfect bot. Code My Road: Programming Projects and Articles, 2013.