CS3243 Project: Playing Tetris with Big Data

FAN Zi Ying (A0179993B), LUO Jiuheng(A0179963H), QIU Kangqing(A0178655M), SONG Jing (A0179516U), ZHONG Yuyi (A0179956A); Group 1
April 2018, National University of Singapore

I. Abstract

This project features a Tetris AI agent that evolves weights using the genetic algorithm. The agent follows basic Tetris rules with the goal of clearing as many lines as possible. This report describes our explicit design of an agent that can achieve high-performing weights and scalability to big data. *The accompanying code requires Java 8 to run*.

II. Agent Strategy

Game Play

To play the game, our agent uses a function pickMove() that chooses the optimal move among all legal moves, where "optimal" refers to the move with the highest benefit score. The benefit score is a linear combination of six features, with weights obtained from the training phase and feature values obtained by simulating the move on a copy of the current board.

Features.

Our agent uses the same six features as in El-Tetris (El-Ashi), an algorithm of superior performance compared to the Tetris agent invented by Dellacherie that had been regarded as the best one-piece Tetris player for many years (Gabillon et al. 1755). Review of existing literature shows that El-Tetris dominates most other agents due to its feature selection, so we chose to directly implement its six heuristics.

1. Landing height.

The height at which a piece is dropped. Since the game ends when the heights exceeds the limit, this feature is essential.

2. Rows cleared.

This heuristic describes the performance standard of the overall Tetris game.

3. Row transitions.

Row transitions count the number of boundaries between filled and empty cells in a row, thus reflecting the flatness of a row.

4. Column transitions.

Column transitions count the number of boundaries between filled and empty cells in a column, thus serving as an indicator of hollow areas present in the board.

5. Number of holes.

A hole is defined as "an empty cell that has at least one filled cell above it in the same column" (El-Ashi).

6. Well sums.

Wells are defined as successive empty cells within a column. Note that there may be multiple wells in a column.

Genetic Evolution

To evolve its weights, our agent uses the genetic algorithm and several enhancements from the standard implementation.

Population size. Table 1 and Figure 5 of Roeva et al.'s paper suggests that as population size increases, computation time increases linearly but performance levels

off. For problems with a small number of dimensions d (d=6 in our case,) a common rule of thumb is to use a population size of 10 * d. In light of this, we chose an initial population size of 100. Lobo and Lima summarize that "population size requirements differ depending on the stage of the search" (200); as the generations proceed, the overall population is already converging towards better performance, so there is less need to preserve the full spectrum of genetic diversity. Inspired by the population reduction scheme in Zamuda and Brest's work, our algorithm evolves the population size over n generations, where during the later (n/2) generations, the population size decreases exponentially, with a minimum size of 41 to ensure an adequate tournament size (see below). This has the additional benefit of saving computation time.

Parent selection. In each generation, we use tournament selection with tournament size equal to 10% of the population size. Among popular methods, it has been shown that tournament selection has desirable performance and in particular, "converges more quickly than roulette wheel selection" (Zhong et al. 1121). Moreover, this method is able to easily "facilitate parallelism" (Gordon and Whitney 177).

Crossover. Uniform crossover is performed, with a mixing ratio of 0.5, meaning each offspring inherits approximately half its weights from each parent. Uniform crossover is appropriate (as opposed to one-point or two-point,) since the features we use can be considered independent of each other.

Mutation. Each new child may have one of its weights mutated with a probability equal to the mutation rate as determined below.

Crossover and mutation rates. Lin et al. propose a scheme for adapting the crossover and mutation rates "in response to the evaluation results of the respective offspring in the next generation" (889). Based on their improved performance, we implemented the mechanism as described in their paper, with two modifications. Firstly, crossover rate was increased to 0.9 from 0.5, since the crossover operation is key to generating offspring that are distinct from their parents, and mutation rate was decreased to (1/6) from 0.5, following a popular rule of thumb in determining mutation rate according to the number of features. Secondly, our algorithm determines θ, the amount of adjustment of the rates, using the formula

$$\theta = 100 \times \frac{\mu}{\sigma^2}$$

where μ and σ^2 are the average and variance respectively, of the population's fitness scores in the current generation. This uses the index of dispersion as a measure of convergence to achieve the same effect of "[increasing] the step sizes to reduce the probability of the GA getting stuck in a local optimum" when the population converges (898).

Parallelization

For efficiency and scalability, parallelization is used to evaluate the game score of each individual in the population, by spawning a new thread for each individual. This means that our code can effectively scale to large population sizes with sublinear increase in computation time. Furthermore, the multiple

games averaged in determining the game score are played in parallel as well, meaning that the compromise between time and variance in game score is minimized. In addition, parallelization can also be added to the tournament selection process, as well as in evaluating each move in pickMove(). Although these two latter areas are currently unimplemented since our model already trains at a fast speed, this means our algorithm design enables further enhancements that can efficiently run on even larger problem sizes.

III. Experiments and Analysis

Average score vs generation. As expected, as the agent evolves longer, the average game score obtained by individuals of the population increases as well.

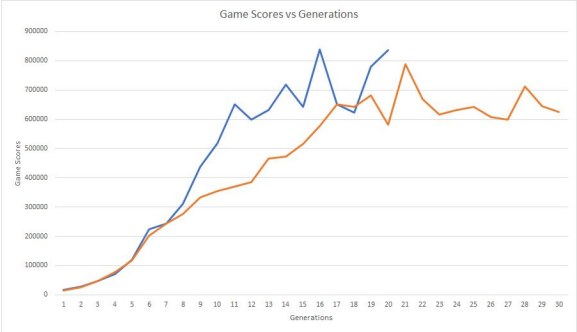

Figure 1. Average game score vs generation.

However, from comparing the above two runs with 20 and 30 generations respectively, we discovered that performance did not improve visibly over the last ten generations. The instability near the end may be due to the decrease in tournament size, meaning that children were less likely enhance the overall population significantly. Therefore, we used the end result from the run with 20 generations.

Number of games. In order to determine the game score of an individual, the scores of NUM_GAMES games were averaged. Setting NUM_GAMES to be 2 proved to be unreliable due to the randomness between games. Running the program with 3 games and 4 games yielded highly similar results. Therefore, considering the tradeoff with time complexity, we decided to set the number of games to be 3.

Selection of children. Based on the OSGA (Offspring Selection GA) of Allenzeller and Wagner, we had implemented a check that only retained children who had better game score than the worse of its parents. However, after running many iterations, no child was ever rejected, so we removed this check to avoid unnecessary computation costs.

Mutation method. Initially, we believed that towards the second half of the evolution process, the partially optimized children should be mutated by a lesser degree in order to step incrementally closer to the optimum and avoid the risk that a drastic mutation disqualifies an advantageous set of weights. Thus, we implemented the function mutateDynamically(), which adjusts the weight by a percentage of its original value in the latter (n/2) generations. However, unlike what we had expected, this implementation led to slight decreases in average score and dramatically faster loss of diversity. Therefore, we abandoned the function and instead used our original mutation method.

Crossover and mutation rates. Our trials showed that Lin's rate-adapting scheme did not significantly improve the performance of our algorithm. This may be due to the fact that in each iteration, the value of θ tends to be quite low; thus, the algorithm behaves similar to a static-rate algorithm. However, since the scheme did appear to offer marginally better performance at a negligible cost, we retained its use.

IV. Results

Below are our final heuristics and weights.

Heuristic	Weight
Landing height	-2.5953031561074615
Rows cleared	6.135171396733583
Row transitions	-2.184882625105884
Column transitions	-5.9874618089311396
Number of holes	-7.098554416480493
Well sums	-2.4152223172808496

Table 1. Final heuristics and weights.

Using this set of weights, we ran the agent over 100 games to obtain the following performance metrics.

Min	Mean	Median	Max
2722	592512.4	488286.5	2181981

Table 2. Performance metrics over 100 runs.

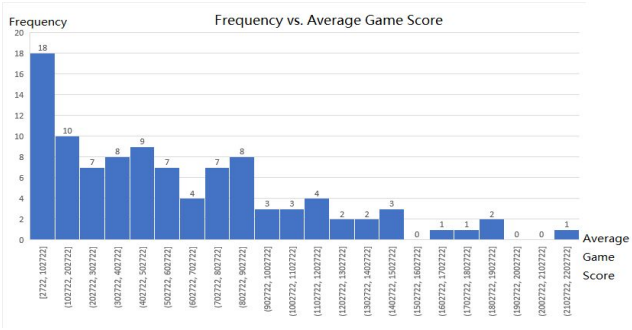

Figure 2. Frequency vs average game score.a

Over 100 runs, our AI agent demonstrated impressive performance, clearing almost 0.6 million lines on average and over 2 million lines at its best. Our current high score out of all trials is 4925897 lines cleared. These metrics are comparable to those of above-average implementations in literature, but by visually comparing the frequency distribution above, we believe our agent has a larger proportion of high-scoring runs than some previous agents.

V. Conclusion

The AI agent presented in this paper successfully combines features that have been shown to yield high performance from prior implementations with several novel modifications to the genetic algorithm. The weights derived from its evolution process have demonstrated high performance during game play. Moreover, this agent is scalable to big data, since its use of parallelization lends it efficient computation. Finally, we have identified areas of further optimization should future work be carried out on top of this project.

VI. References

Affenzeller, Michael and Stefan Wagner. "Offspring selection: A new self-adaptive selection scheme for genetic algorithms." *Adaptive and Natural Computing Algorithms*. Springer, Vienna, 2005, pp. 218-221.

El-Ashi, Islam Mohammad Anis Kamel. "El-Tetris - An Improvement on Pierre Dellacherie's Algorithm." *imake*, http://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/. Accessed 10 Apr 2018.

Gabillon, Victor et al. "Approximate dynamic programming finally performs well in the game of Tetris." *Advances in neural information processing systems*, 2013, pp. 1754-1762.

Gordon, V. Scott and Darrell Whitney. "Serial and Parallel Genetic Algorithms as Function Optimizers." *IGCA*, 1993, pp.177-183.

Lin, Wen-Yang et al. "Adapting Crossover and Mutation Rates in Genetic Algorithms." *J. Inf. Sci. Eng.*, vol. 19, no.5, 2003, pp. 889-903.

Lobo, Fernando G. and Claudio F. Lima. "Adaptive Population Sizing Schemes in Genetic Algorithms." *Parameter Setting in Evolutionary Algorithms.* Springer, Berlin, Heidelberg, 2007, pp. 185-204.

Roeva, Olympia et al. "Influence of the Population Size on the Genetic Algorithm Performance in Case of Cultivation Process Modelling." *Computer Science and Information Systems (FedCSIS)*, 2013, pp. 371-376.

Zamuda, Ales and Janez Brest. "Population Reduction Differential Evolution with Multiple Mutation Strategies in Real World Industry Challenges." *Swarm and evolutionary computation*, Springer, Berlin, Heidelberg, 2012, pp. 154-161.

Zhong, Jinghui et al. "Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms." *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, Vienna, 2005, pp. 1115-1121.
doi: 10.1109/CIMCA.2005.1631619