Evolutionary Computation and Artificial life
Evolving Tetris Controllers Tom Choi, Will Richards, Adi Subramanian

Introduction

Tetris is a game that has commonly been a topic of experimentation in the field of evolutionary computing. Before the development of HyperNeat, many researchers considered a variety of heuristics to create the optimal Tetris controller. In lieu of HyperNeat working, we attempted to emulate these studies.

In our project, we considered four heuristics to create a controller that plays Tetris as well as possible. We first calculated all valid board configurations with two lookahead pieces. Then, each board state is evaluated with respect to four heuristics. We evaluate each board state from two moves ahead, and then make the move that corresponds to the optimal board state.

We chose to use four board heuristics that we believe contained meaningful information about the fitness of a board state. The four board-state heuristics chosen were: aggregate row height, number of holes (empty squares below a tetris block) , numbers of rows clear, and bumpiness (variation in column heights). All of these heuristics have been considered by other studies in the past, and have shown to be important in creating a good controller.

Hence, the general objectives of the project is optimizing coefficients associated with the four heuristics which ranges from -1 to 1. The more negative a coefficient gets, the more it punishes its corresponding heuristic. On the other hand, the more positive a coefficient gets, the more rewards its corresponding heuristic.

Infrastructure

We implemented Tetris and the evolutionary components of this project from the ground up. Board.java and Piece.java control the game logic. TetrisGraphics.java controls the visualizations of the board states. Player.java defines and constructs an individual in our population. We represent the genome of a player as 4 scalar values, each corresponding to a heuristic. TetrisEA.java controls the logic to run an experiment.

In an experiment, we perform 5 runs of tetris for each player to determine a fitness. After fitnesses are determined, elite individuals (if any) are set aside for the next generation. Individuals are then subject to undergo mutation and crossover.

The mutation probability of an individual determines the chance that a single scalar value will receive a mutation. The mutation range determines the maximum value that a scalar value can be incremented or decremented by. If an individual is subject to crossover, then it will randomly swap between 1 and 4 scalar values with another individual. Simulate.java simulates a new tetris run for a stored genome.

Parameters related to evolutionary algorithm such as population size, number of generations, mutation and crossover probability, mutation range, number of elites, etc. can be manipulated within params.txt.

Approach

As mentioned above, we used a heuristic based approach to this problem. In Tetris you can see the board, your current piece to place, and the next piece. We figured out the board state for each possible placement of the two pieces and then calculated the “goodness” of that board state. The “goodness” of a state is simply the value of each factor (bumpiness, height,

holes, and completed lines) for the given state times their respective coefficient for the candidate solution. We do this for every piece until the candidate loses the game or reaches our score cap of 5000. We then repeat this 5 times, as any given run has significant variance in solutions.

The overall fitness of a candidate solution is the average game score over their five trials. We had few methods of slowing convergence and instead decided the best way of finding the optimal solution was to start with a large population size - in this case 50. Due to the fact that each run of 10 generations took around 2 hours, we did not have the time to test a large number of parameters. Instead, we held elite clones constant at 2, max fitness constant at 5000, and crossover probability constant at .3. We tried values of 5 and 10 for tournament size, .05 and .15 for mutation probability, and .1, .2, and .3 for max mutation rate.

Our genotype is simply the four coefficients for each heuristic. Evaluating similar phenotypes would be difficult for this problem, and we don't worry about finding phenotype similarity as we don't use any type of fitness sharing or other similar methods to encourage diversity.

Evaluating Fitness

For all simulations, we put a hard limit on the fitness a solution could reach before terminating the simulation. We first used the number of rows cleared during a single random simulation as a solution's fitness. Because each simulation used a new random sequence of pieces, the fitness of a single solution could significantly vary from run to run. To account for this, we ran 5 simulations for each solution, and took the average number rows cleared to assign fitness. This ensured that individuals with a higher fitness didn't get a single lucky run, and could perform well on multiple sequences. Overall, having 5 runs per solution greatly improved the overall fitness of our genepool over the course of the evolution, but it made the whole simulation 5 times slower.

Many versions of tetris give a score bonus breaking multiple lines at once. We thought it would be most interesting to use a score bonus in our final fitness evaluation method, as it would incentivise the intelligent behavior of trying to break multiple lines at once. This method did equally well in the number of lines it cleared, and visualizations of the fittest strategies show intention to clear multiple lines at once. The fitness values for clearing 1,2,3, and 4 lines were 2,5,15, and 60 respectively. These are reduced values of the common tetris scoring guidelines.

Baseline Tests

Our first baseline test was placing blocks in random columns. While we haven't been able to calculate the exact probability of breaking a line with this behavior, in a hundred trials this strategy never broke a single line. This tells us that Tetris is not trivial and that any solution that manages to break multiples lines is very likely to be working intelligently.

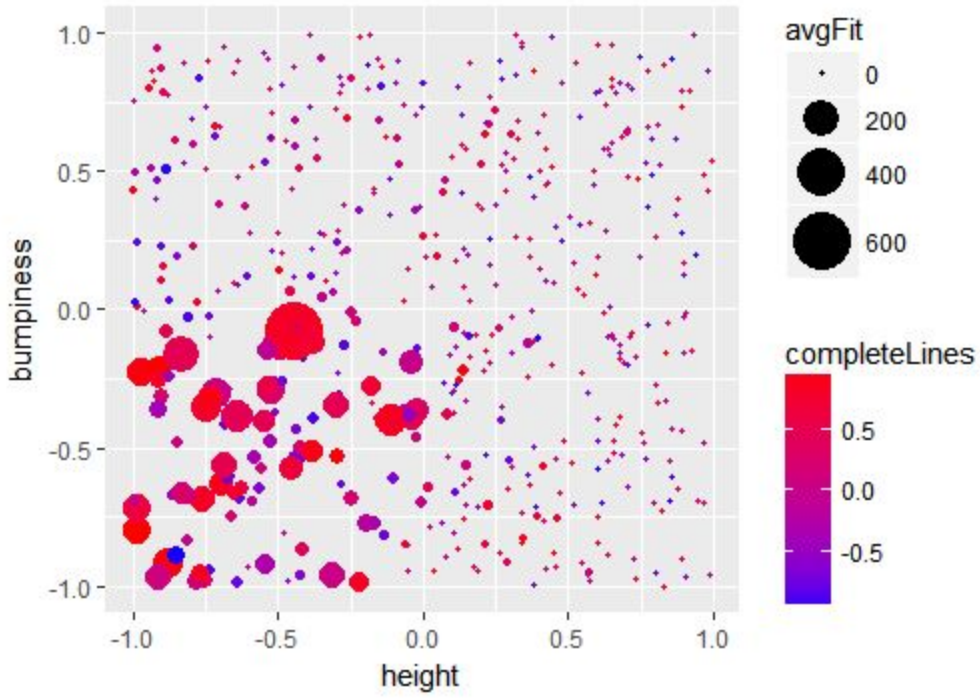
Our second baseline is generating random values for each coefficient for the four heuristics. This systematically searches the solution-space, using a brute-force sampling approach. In this case we see the average fitnesses of these controllers (over five trials) varied from 0 to 627.2. As you can see, this generated some results that are nearly as good as our evolved outputs. This emphasizes the importance of a good starting population. A majority of our randomly generated controllers were terrible, however, which isn't surprising.

Given that we only had four heuristics with coefficients that vary from -1 to 1, it is possible for us to visualize the fitness landscape. We collected data on random values for the coefficients of each heuristic and did some data analysis on it in order to get a better idea of the fitness landscape. The first thing we did was run a simple OLS regression on the coefficients with the dependent variable being the fitness. We found that the holes heuristic did not help to evaluate a board. When including the square of the holes coefficient we found that it was heavily negative and significant, indicating that the best solutions weren't impacted by holes at all, and that the only impact of the holes heuristic was to weaken the impact of the other, important factors. Beyond that the regression yielded the expected coefficients for the other variables - you want a short, even stack with as many completed lines as possible. It's also worth noting that none of the other variables squared were significant, indicating that none of them have an independent parabolic distribution with fitness - which would indicate a peak or optimal value. Instead, the fitness either depends on the ratio of the three variables or is higher the closer to the extreme each of these variables are.

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)     9.859      1.581   6.237 9.57e-10 ***
height        -16.847      2.793  -6.032 3.17e-09 ***
completeLines  12.498      2.761   4.526 7.53e-06 ***
holes          -4.195      2.752  -1.524 0.128087
bumpiness     -10.603      2.712  -3.909 0.000105 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    14.331      2.362   6.067 2.59e-09 ***
height        -17.459      2.780  -6.280 7.41e-10 ***
completeLines  12.509      2.744   4.558 6.52e-06 ***
bumpiness     -10.299      2.704  -3.809 0.000157 ***
holesSquared  -13.777      5.309  -2.595 0.009739 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Given these results, we discount holes from our fitness landscape visualizations. However, we included it in our actual algorithm with the expectation that the optimal solution has a value of zero for holes. Below is a fitness landscape for the other three variables.

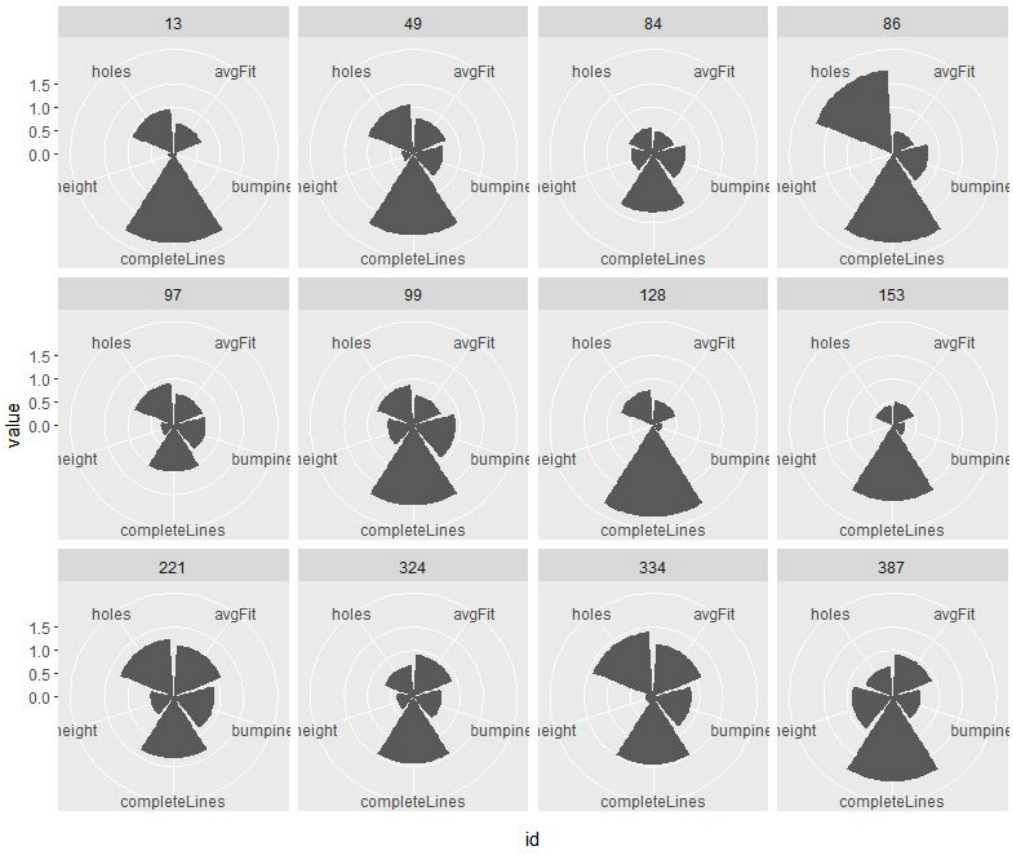


This plot provides some interesting results. The first thing worth noting is that complete lines seems to be a simple case of maximizing the coefficient. We see the point with the highest fitness has several points around it with similar coefficients for bumpiness and height, but opposite values for complete lines. Instead of being a really good solution, it's a really bad one. This isn't very surprising - our fitness function is a function of how many lines you clear. On the other hand, we see that the optimal fitness function doesn't simply minimize height and bumpiness values. Instead it seems like the ratios might be important. However, several regression models including the ratios weren't significant. One interesting thing to note is that if you consider whether not both bumpiness and height had negative coefficients as a variable, then height and bumpiness are no longer significant in the regression. This indicates that within the search space where both values are negative, there is no linear pattern to find the ideal values. This indicates that many standard prediction methods won't work for this problem. Instead, it makes it an ideal problem for an evolutionary algorithm to handle.

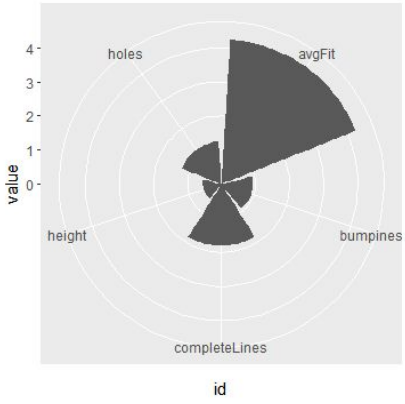
```
Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)            4.680      2.490   1.879  0.06079 .
completeLines         12.377      2.569   4.818 1.93e-06 ***
holesSquared         -13.284      4.970  -2.673  0.00777 **
coefSignsCorrectTRUE  40.941      4.860   8.424 3.99e-16 ***
height                -2.356      3.160  -0.746  0.45627
bumpiness              2.996      2.983   1.004  0.31570
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

To visualize the relationship, we plotted the scaled values of all randomly generated solutions with fitnesses above 100. It's worth noting that here low values meant more negative

for many of the variables. Overall this supports the guesses we made above. Below is a plot of all but one - which had such a large value of avgFit that it skewed the other values.



And here is the same plot for that solution:



Selection Method: Tournament Selection with Elitism

There existed some cases where some of the best solutions from the previous generation were not selected. This led to dramatic poor performances and slow evolution in the following generations. To account for this problem, we preserved two elites for every generation

and let the rest of the population randomly undergo mutation and crossover. We tested a tournament size of 5 and 10 (10% and 20% of the population respectively). We did not implement fitness sharing or any similar method to disincentive convergence. This is likely why our algorithms converged within 15 generations. However, we compensated for this by beginning with a larger initial population.

Mutation and crossover

We assigned each individual a certain probability to mutate. If an individual is subject to mutation, a random allele is incremented/decremented by a random value within the set mutation range. If one were subject to mutation, we randomly selected one allele in its chromosome and randomly incremented or decremented its weight or value by 20%. Once the mutation was over, we then normalized its chromosome by dividing each component by the length of its vector.

We used vector weighted average method of crossover. If two individuals were chosen to crossover, we first randomly decided how many alleles to crossover and then randomly crossovered corresponding alleles between two genomes. Chromosomes were normalized after crossover as well.

Results and analysis

Overall, our evolutionary algorithm of evolving heuristic weights was relatively well suited for the problem of tetris. While we were able to achieve impressive individual runs, many of our solutions had inconsistent performance. Certain heuristics may have been trained to only recognise optimal board-states only for certain board configurations. Once al board was roughly level, the controller could regularly clear lines prevent a game over. Very early on, however, many solutions occasionally generate irregular board configurations early on, and are unable to intelligently compare board states with the heuristics that overspecialized in comparing board states for regular, flat boards. Because each solution is tested on only 5 board states, solutions with high outlier fitnesses are heavily rewarded. Evaluating each solution with more runs would help to train our board-state evaluation on a wider variety of board-state configurations. We were unable to use more than 5 runs per individual to to computational resources.

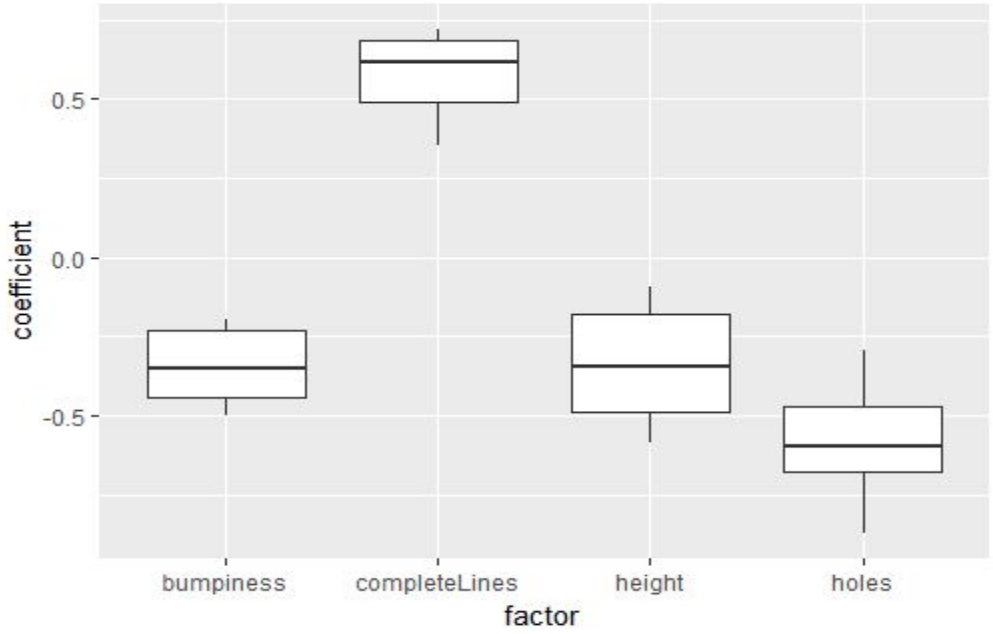
Optimal parameters

In order to find a set of optimal parameters, we used bash scripts to test various parameter combinations. A smaller mutation rate tended to generate fitter solutions. Of the seven solutions that had a fitness above 1000, 5 were generated with a mutation rate of 0.05. Only two fit solutions were found with a mutation rate of 0.15, and no solutions were found with a mutation rate of 0.25 (although we cut these trials short due to time constraints). Mutation range did not seem to have a large effect on fitness. Two fit solutions were found with a mutation range of .1, two fit solutions had a mutation range of 0.2, and three fit solutions had a mutation range of 0.3. Tournament size also did not seem to have a large effect on the fitness of generated solutions. Four fit solutions were found with a tournament size of 10, and three were found with a

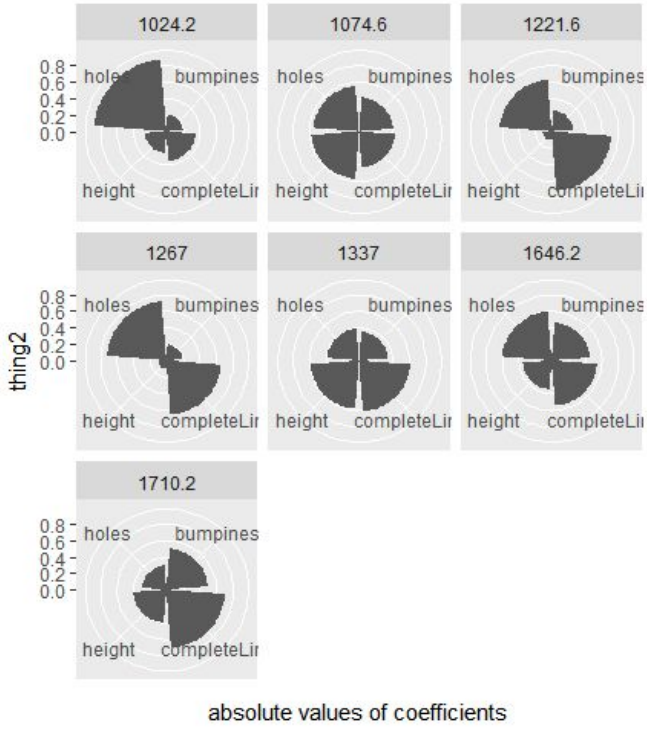
tournament size of 5. We found the most fit solution using a mutation rate of .15, a mutation range of 0.1, and a tournament size of 10.

Optimal Coefficients

We had 7 candidate solutions with a fitness over 1000. Among those, the coefficients for height varied from -.09 to -.58. The coefficients for completeLines varied from .35 to .72. The coefficients for holes varied between -.29 and -.87. Finally, the coefficients for bumpiness varied from -.19 to -.50. While the signs are as expected, the values are a little surprising. Holes seems to make much more of an impact than the random sample indicated, while complete lines was maximized much less than expected.



We looked for a pattern in the relationships between the variables in these successful candidates. Unfortunately, beyond getting the correct coefficients we did not see any. Instead we clearly see that there are many local maxima within this fitness landscape, and that we likely did not find the global maxima in our experiments.



Recursive Reflection

We originally tried to use HyperNEAT to evolve our controllers, but we ran into serious complications. We tried various input and output

representations for the HyperNEAT substrate, but we were unable to get any seed of intelligent behavior. Overall, we decided that instead of relying on an external library which was frustrating to work with, we should implement the genetic algorithms we have learned about ourselves to ensure our program runs as intended. We decided to implement the heuristic based approach that earlier studies had tried and simply evolve our coefficients. While we ran into the occasional problem with our new implementation, these were easy enough to catch and fix before we began to run our final experimentation.

On a similar note, we originally had a stretch goal of using co-evolution to develop an "evil" AI that tried to force you to lose as quickly as possible. Given that there is a sequence of tetrominoes that is guaranteed to end your run, we figured this would find the optimal solution relatively quickly. However, given the difficulties we had implementing the primary part of our project, we did not have time to implement and run experiments to change the piece distribution using a genetic algorithm.

Future Work

There are many ways we could expand this project in the future. The first possibility is to change or add heuristics. Thiery and Scherrer (2009) identify 32 different heuristics that have been used to create Tetris controllers in the past. We could have tried a variety of others to see which actually lead to the best possible solution.

Another change that we could have made was making our fitness more reliable. Five trials isn't enough to ensure that we're truly getting a good solution and that it didn't simply get lucky. However the evaluation time was the most expensive part of our algorithm and we didn't have time to run more simulations.

We also could have implemented a more robust genetic selection algorithm. At the moment we only had tournament selection and we didn't have anything to discourage convergence. Adding functionality for fitness proportional selection or fitness sharing might have improved our findings because we noticed that once it finds somewhat decent solution, it quickly converges. We also could have tried to vary our parameters more or simply run more trials, as in this study we were limited by time.

Finally, in the future we could improve this project by using a neural network. In particular, this is exactly the type of problem that can be solved by HyperNEAT. This controller would be greatly improved by the use of this algorithm to make decisions rather than the heuristic based approach that we ended up using.