Genetic Algorithms and Tetris

Eckel, TJHSST AI2, Spring 2024

Background and Explanation

Recall that **genetic algorithms** are a powerful tool for improving on the idea of hill climbing to optimize a solution in situations where all of the following are true:

- A strategy can be precisely quantified by a specific set of variables given certain numeric values.
- The outcome of the strategy can also be precisely quantified.
- There are interactions between the variables that make simple hill climbing inefficient or unlikely to succeed.

Remember that the components of a genetic algorithm are as follows:

1. A **population**.
2. A **fitness function**.
3. A **selection method**.
4. A **breeding process**.
5. A small chance of a **mutation**.

This assignment extends our understanding into a totally different context – we will watch an AI learn to play a simplified form of Tetris!

Tetris

I won't go over the general gameplay of Tetris; if you've never played it before, Google it! It's pretty fun… For this assignment, we'll be using a simplified version, not quite the same as normal Tetris. I'll specify:

- The game board is height 20 and width 10. This can be hardcoded. You will need to accept input in the form of a 200 character string in row major order from the top left to the bottom right with blank spaces represented by a space character and blocked squares represented with a # symbol. You can either store your state this way or code a translation function to whatever format you prefer.
- Any completed rows disappear and the rows above shift down by one. The player gains points as follows:
    - One row eliminated = 40 points
    - Two rows simultaneously eliminated = 100 points
    - Three rows simultaneously eliminated = 300 points
    - Four rows simultaneously eliminated = 1200 points
- There are no point bonuses of any kind in our version (for instance, no multiplier for clearing the entire board).
- Our version will not be concerned with time or speed in any way (for instance, we won't simulate the piece slowly falling). On each turn, a random Tetris piece will be chosen, and the AI will choose the orientation of the piece and the location on the board to drop it.
- Our version will not include any of those fancy strategies where you slide a piece sideways at the last moment to fill a gap; pieces may only be dropped from the top of the board directly downwards.
- The game ends when any portion of the most recently placed piece is out of bounds (off the top of the board).
- The seven pieces, and all possible orientations of each piece, are shown in a text file available on the course website.

The first part of this assignment is simply to make a working Tetris game; that is, it should be able to place a piece, eliminate any rows as necessary, and keep track of the points. Once that's working, we'll develop an algorithm to generate and improve strategies.

Part 1: Model Tetris

For the first part of this assignment, I want to simply test that your Tetris model works correctly, so I'll give you a board state and have you create every possible state that could come next (for any orientation of any piece in any location).

Specifically:

- Your input string will be a single command line argument (given in quotes, so that it may contain space characters). It will be 200-characters long and consist only of spaces and # symbols, representing empty and blocked squares in row major order from top left to bottom right.
- Your code will then need to loop. For every Tetris piece, for each possible orientation of that piece, for every possible place that orientation of that piece could be dropped, drop the piece onto the board and generate the resulting board state.
- If dropping that piece onto the board completes any lines, **those lines should be eliminated and all of the rows above should shift down by one**. (A common mistake last year was not implementing this logic in this part of the assignment.)
- Your output will be a file with the ***exact file name*** `"tetrisout.txt"` which will contain all boards that could result from one additional turn of the game. The fill will contain one board per line, each board being a 200-character string (just like the input, only space and # characters). The only exception is if the game would be over, explained below.
- If the most recently played piece does not fully fit onto the board, the game is over. In this case, instead of printing the board, you should print the precise string `"GAME OVER"`. (My grading script will count how many `"GAME OVER"` strings there are in your file.)
    - For example, if there is a column that is blocked from the bottom row, 19, all the way up to row 3, then dropping the 1x4 bar vertically onto that column will end the game. The 20 rows of the board are indexed from 0 to 19; dropping the bar in that location would place its four blocking squares in rows 2, 1, 0, and -1. There is no row -1, the piece is out of bounds, and the game is over.
- As a quick check: `"tetrisout.txt"` should contain exactly 162 lines. There are 162 possible piece/orientation/location possibilities, and every one of those 162 possibilities should be tried on any given board. As explained above, some of those possibilities may result in the game ending, so one or more lines will say `"GAME OVER"`, but either way there should be exactly 162 lines.
- A test case is available on the course website, along with a script to verify your output. **Please test your code before submitting**; this is an assignment where so many different minor mistakes can be made…

Specification for BLUE Credit: Model Tetris

Submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code accepts a single command line argument – one board state, a single 200-character string as specified in the previous section.
- Your code creates a file with the ***exact file name*** `"tetrisout.txt"` where it prints every possible board state that could occur after one more move. Order doesn't matter. As explained above, if putting a piece in a certain location would end the game, it should print the *precise string* `"GAME OVER"` instead of a board state. (I will count how many `"GAME OVER"` strings there are in your file.)
- Runtime should be very short; I'll accept anything under 2 minutes, but if your code takes longer than a second the rest of this assignment will be very slow indeed!

Part 2: Genetic Algorithm Application and Parameters

Once Tetris is working, we'll want to be able to define a strategy, and in order for the genetic algorithm to work, we must define that strategy as a set of numeric values.

So this is what we will do. Each AI strategy taken individually will play a Tetris game according to the following general plan:

1) Receive the current board state and a randomly chosen Tetris piece that will be placed next.
2) Generate each possible board (ie, loop over each possible orientation of the given piece and each possible location on the board to place it) and rate each resulting board according to a heuristic.
3) Make the move that produces the highest heuristic rating.
4) Repeat until the game is over.

Pseudocode for this looks something like this:

```
def play_game(strategy):
    board = make_new_board()
    points = 0
    while game is not over:
        piece = randomly chosen piece
        for each orientation of piece:
            for each column on the board where this orientation will fit:
                poss_board = place(piece, orientation, location, board)
                poss_score = heuristic(poss_board, strategy)
                # Keep track of the board with the highest heuristic score however you like!
        board = (the board with the highest heuristic score)
        if any lines were cleared:
            points += new_points    #reminder: 1 row cleared --> 40 points, 2 --> 100, 3 --> 300, 4 --> 1200
    return points
```

The question here, of course, is: what does "better" look like? What could a heuristic look at to rank some boards as better than others? The exciting thing about genetic algorithms is that we don't have to define a strategy, we just have to give it enough information that the algorithm can determine what information is actually useful on its own.

So: decide on a list of things about a Tetris board that can be counted or scored (ie, board height, deepest well, number of gaps, etc). Then, define a strategy by generating a corresponding list of particular coefficient values that could be either positive or negative. To rate any board, each quantity will be found and multiplied by its corresponding coefficient, with the resulting values summed. We don't have to have any idea what aspects of the Tetris board might be important – we'll just write the code to calculate as many aspects as we can think of, and then the algorithm itself will figure out which characteristics are appealing, unappealing, or irrelevant. The strategies that intelligently rank boards, prioritizing the quantities that matter most, will play the longest and get the highest scores.

IMPORTANT NOTE: Be sure to give any board that ends the game a **large negative value**, so your code never chooses to end the game on purpose!

Pseudocode for the heuristic scoring algorithm looks something like this:

```
def heuristic(board, strategy):
    a, b, c = strategy  # As many variables as you want!
    value = 0
    value += a * (perhaps highest column height?)
    value += b * (perhaps deepest well depth?)
    value += c * (perhaps number of holes in board, ie empty spaces with a filled space above?)
    value += d * (perhaps the number of lines that were just cleared, in the move that made this board?)
    # add as many variables as you want - whatever you think might be relevant!
    return value
```

I encourage you to be creative about which things you'd like to measure. I came up with seven; I read one research paper that used twelve!

Aside: A Cool Sort-of-Advanced Extension

Last year, after playing around with this a lot, we found that adding not only a coefficient but also an *exponent* to our variables increased the capability of the genetic algorithm to learn. (This is also represented in genetic algorithm research papers we found.) It allowed, for instance, a situation where *one* of something wasn't really that bad, but *four* of them was super bad, because the exponent learned to be very big and raising four to that exponent made a *much* larger number. So that would look like this:

Pseudocode for the heuristic scoring algorithm looks something like this (c for coefficient, e for exponent):

```
def heuristic(board, strategy):
    ac, ae, bc, be, cc, ce = strategy  # As many variables as you want!
    value = 0
    value += ac * (perhaps highest column height?) ** ae
    value += bc * (perhaps deepest well depth?) ** be
    # etc
    return value
```

This is not required to succeed at this assignment, and does make your code take longer to execute, but I think it's pretty cool and if your code isn't working it might be sufficient to unstick you.

Either way, with your coefficients and possibly exponents decided, this is how we will build our genetic algorithm here:

1 – Population

This is straightforward – figure out how many quantities you wish to measure on each board, and generate random lists of coefficients of corresponding length. An easy way to do this is to generate random floats from -1 to 1, but I leave the choice to you if you'd like to try something different.

2 – Fitness Function

This is also straightforward – let the strategy play Tetris until it loses and see what final point total it earned! There is a complication here in that there's a lot of random variation, but taking the average of 5 final point totals is approximately good enough to deal with that. So: to give a fitness score to a Tetris strategy, let it play 5 complete games and average the final point totals.

Pseudocode looks something like this:

```
def fitness_function(strategy):
    game_scores = []
    for count in range(num_trials):
        game_scores.append(play_game(strategy))
    return average(game_scores)
```

3 – Selection Method

Up to you! Do you like clones + tournaments? Something else? (I would strongly suggest more clones than 1 this time!)

4 – Breeding Process

This is actually much more straightforward than the ciphers. Each strategy is defined by a set of numbers. Just pick a random number from 1 to the strategy list's size - 1, randomly choose that many indices, copy those indices from parent 1, copy the rest from parent 2.

5 – Mutation

Mutation, in this case, would either be regenerating or modifying one of the values. Choose a new value, or perhaps add a random amount to one of the values.

Reminder: Use Global Parameters!

As before, there should be a list of capitalized global constants at the top that you can modify. I'm not giving you any guidance here; you'll have to find your own values. As before, **MAKE SURE YOU DON'T HARDCODE ANY OF THEM BY ACCIDENT!** Being able to change these easily is key to success in this assignment.

An Important Note on "Scores" and How This Is All Very Confusing

Before I move on, I just want to note that a lot of different things are getting scored in a lot of different ways in this algorithm. Hopefully this can avoid some confusion.

- During a game of Tetris, a random piece will be chosen, then your code needs to decide where to put it. To decide, it will make every possible next board and rank them *according to a heuristic score* that judges how good the board looks. This heuristic score will be determined by the strategy we're testing from the current population.
- During a game of Tetris, the AI player will earn points – 40 for clearing one row, 100 for clearing two, etc. When the game is over, the AI player will have earned *a final game score* for that game.
- To evaluate each heuristic strategy, we will run five games and average the resulting *game* scores, to produce a *fitness score*.

These are all different scores used in different places. If those three bullets are confusing, you should probably ask me about them!

Required Task

The goal is to run enough generations to create a single strategy that scores 50,000 points over a 5-game average.

In order to get there, since generations will take a long time to score, we will want this script to be able to save a generation and exit, so that we can load and resume growth if we wish.

As such, you should write a script with the following flow. When the script is run, the user should be given two choices:

- Start a NEW genetic process
- Load a SAVED genetic process

If the user chooses a NEW process, begin by randomly creating a population to begin the process and scoring each member according to the fitness function (by averaging the final point totals of five games). If the user chooses to load a SAVED process, your code should load a population of strategies **along with their fitness scores**.

Then the user should be presented with three choices:

- Watch a game played by the BEST strategy of the current generation (then return to this set of choices again)
- SAVE the current generation (each strategy **along with its fitness score**) to a file and end the script
- CONTINUE to evolve another generation

If the user opts to continue, after the next generation is created they should be presented with a summary of the results of the new generation:

- The coefficients in the best strategy printed to the screen, along with that strategy's score
- The average score of every strategy in the generation

…and then the loop should repeat, giving them the same three options: watch a game, save, or continue.

Some tips:

- As far as saving/loading goes, you can use `pickle` (see course website) or write custom code to write to & read from txt files. It might be a good idea to let the user specify a filename so the results of multiple different processes can be saved separately.
- Finally, if the user wants to watch a game, find some way to nicely print each board along the way.

This entire process is demonstrated in a video on the course website; be sure to watch it before submitting!

What If You Never Get to 50,000?

It's possible that you code this up and it just works. Yay! What if it doesn't?

Well – one possibility is that you've done something wrong. Some common problems include:

- Bad values for global variables.
- Bad choices for things about the board to measure.
- Not ensuring children are unique.
- A mutation process that causes children to become wacky, like ending up with a few values between -1 and 1 and one value that's 40,000,000,000 or something like that. Print some children and make sure they seem reasonable.

These problems can be subtle and hard to track down, but those are some starting points.

Last year, though, a number of totally stuck students were able to eliminate any of the errors above and still made no progress… until an idea started to spread about performing a kind of high-level strategic whittling down process. They would run their code and breed a few generations and then look at the best few strategies. Then, they would find the things that their code seemed to find no value in – measures of the board that ended up with low coefficients – and they would delete those measures. Then they would restart the genetic algorithm over again but with fewer variables.

This is a completely valid, intelligent thing to do! BUT: the problem with this is that students often found that only a couple of things were actually necessary for a good strategy, and two coefficients doesn't make for much of a genetic algorithm, rendering some of the process rather pointless. This is fine; if you learn enough from the results of prior genetic algorithms to break the assignment in the end, then you still got the benefits of the process somehow.

You can also make a small number of measures into a more authentic genetic algorithm by following the advice to add exponents in the section on the "Advanced Extension" a couple pages back.

But, one important thing: **if you do improve the performance of your algorithm by removing measures yourself, rather than simply trusting the genetic algorithm to do so for you, I want you to report what you cut and why in your analysis below.**

Analysis

Finally, before submitting, I want you to write me a little bit about your results. (Not much, don't worry.) Make a document (doc, txt, or pdf) that answers these questions:

1) First, show proof (output from your code) that shows your genetic algorithm did improve over time. I should see at least 3 generations of improvement. (That is, the average of generation 1 should be greater than generation 0, average of 2 greater than average of 1, and average of 3 greater than average of 2.)
2) How many generations did it take before you had one strategy that scored over 50,000 points? How much time did it take to run through all those generations (approximately – minutes, hours, days?) If you ran multiple processes that didn't work, but slowly edited until you had a process that did work (see previous section), describe that here too.
3) Print out the coefficients in the best strategy you've been able to generate, and copy them into the document. Note what characteristics of the board you decided to measure, and which coefficient corresponds to each characteristic. Then briefly analyze the results. Are the things that your strategy assigns positive/negative worth to the ones you would've expected? (I'm looking for thoughts like "I assumed that any winning strategy would assign a large negative value to the height of the highest peak on the board, but strangely my winning strategy's coefficient for this measure was almost zero!")
4) Did you like this assignment? Should I keep it?

Submission

Your submission for this assignment will need to contain multiple things.

1) Your Python script, as specified above.
2) A saved generation of your algorithm in which the best strategy scored over 50,000 points.
3) A document (txt, doc, or pdf) where you answer the three questions specified on the previous page.

I'm going to ask you to **submit a single zip file** to the submission form; when it is extracted, it should, contain only those three files. **Do not submit the files separately – make a single zip file and submit that.**

How I Will Grade You

I will grade your assignment in the following way:

1) I will read your analysis.
2) I will run your code, load your best strategy, and watch it play a game.
3) I will run your code again, start a new genetic process, and breed one generation. I should see that generation 1 gets better results than generation 0.

Specification for RED Credit

Submit **a single zip file to Google Drive** (NOT three separate files).

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- The folder contains the three files specified on the previous page.
- Your code performs as expected in both scenarios described above.
- Your analysis is complete.

Specification for BLACK Credit

I still haven't settled on exactly what I want for BLACK credit here. I can think of a few ideas and am happy to entertain others. Either way, propose something to me and we'll decide on how to turn it in / grade it:

- Use a genetic algorithm to break a different / more complex code than the Substitution Cipher. Ask me about ADFGVX or Playfair as two possible options.
- Use a genetic algorithm to improve your Othello strategy somehow. Or just experiment; maybe come up with the best weighting for every square on the board? You'd need to scale back the Othello turn time in order to make this reasonable; maybe restrict to only looking 3 moves in the future or something like that. Tell me about your plan and what you hope to learn from it before starting so we agree it's a good line of inquiry.
- Find an interesting way to twist this assignment. For example, could we train code to get the highest score given a finite number of pieces, so perhaps we could train it to prefer using 1x4s to clear 4 rows at once? Can you think of another interesting way to push a genetic algorithm to do something different in Tetris? A note: I'm not interested in ideas that are in the category of "make a more realistic game of Tetris that's like the one humans play"; we're looking for insight into the algorithm, not a chance to play more Tetris.
- Use genetic algorithms to do something else entirely. I'd really like this unit to have a Semester 1-style Modeling Challenge attached to it, and I haven't found the right thing. Do some research, try something?