Overview

The objective of this project is to build a single-player Tetris game in Java that moves according to the user's input. We also develop an AI to play Tetris instead of the user.

Goals

Our personal goals for this project were to build our first game and a reinforcement learning (RL) agent to play the game. Additionally, a goal with our RL agent was to make it solely in Java without using any additional packages. It would also allow us to practice with automated testing, as it is difficult to determine if the resulting board is actually "correct".

Solution Design

TetrisPiece:

- Since there is are only 7 piece types and 4 rotations for each piece, in order to maintain O(1) calculations we manually precomputed both the body and skirts of each of these 28 configurations using https://tetris.wiki/File:SRS-pieces.png and store them in static arrays. These were O(1) since the methods would just return the loaded values
- For each new TetrisPiece, it has a pieceType and rotationIndex field specifying one of these configurations. During the constructor, these two fields are used to load in the rest of the piece data: width, height, body and skirt. Rotations were done by adding or subtracting 1 mod 4 and using a private constructor that takes the new rotationIndex

<u>TetrisBoard:</u>

- In order for Tetris to move quickly, all the getter methods should be O(1), which means we store many fields ahead of time so the getter methods can immediately return the value with no computation. There were 4 types of data calculated and stored in the constructors: general board, current board, current piece, and last move information
- In terms of performing the actions on the board, we created methods for each move that works by performing the move and if the move is valid, then leave it, otherwise revert it back to its original state. For instance, if move(Board.Action.RIGHT) is called, then it will call our associated method runMoveRight() that moves the piece right, and if it is out of bounds or any of the points in the body are already filled, then move it back to the original location. One exception is that performing a down or drop move required more code in order to update all the necessary variables. We use the same logic as our previous moves, but we also have to update the grid, clear any rows that are full, update the column heights, and reset the current piece
- For invalid input handling, widths and heights of negative values get moved up to 0 making an empty board where no next piece can be spawned. If no piece can be spawned, no moves can be run so there won't be any functionality possible but no errors or exceptions as well

<u>JBrainTetris:</u>

- The entire functionality is already set up in JTetris but we need to shift control from the user to the brain. We remove user input on the board by setting making tick empty and query the brain's next move using a brain timer every 1 millisecond

Our Brain - QLearningBrain (Karma):

- We implement a Q learning algorithm supported by a heuristic to learn how to play Tetris. Since our goal was to implement the AI with no packages, we strayed away from a Deep Q Learning (DQN) approach and instead solved the Q function with a lookup table.
- The issue with a lookup table is that each square is in a 10x20 grid with 8 possible states at each grid (7 pieces or empty), giving 8^{200} states to compute. Obviously this is too many states, so we need to reduce the possibilities using quantization. After some thought, we converged on the idea to only care about column heights for each state, giving 21^{10} possible states. We could do this since we made the restriction that the only moves we can make are a sequence of rotations and left/right moves at the top and then a drop (so no moving around and under a piece). This is still too much, so we can impose some pooling techniques that condenses every disjoint 2x3 box into a singular square with its maximum height. Combined with 7 piece type possibilities of the spawned piece and a maximum of 4 * 11 (number of rotations * (width + 1)) actions per state, there are $7^5 * 7 * 44 = 5176556$ total state action pairs. Encoding and decoding methods helped represent and expand states to make this quantization possible
- Each state action pair is a value in our lookup table, with the value representing the approximation of the Q function for taking that action at that state. The Q value can be thought of as the expected reward, so our goal is to choose the maximum. First, we need to find the values for each pair so we have the brain play many tetris games, store the reward from each action taken at each state, and use a temporal difference learning function to update the Q values. The reward function contains discrete rewards from the

score of tetris and surviving longer, as well as a smaller continuous reward to incentivize choosing boards in better states

- In order to quantify how good a board is, we used inspiration from a few heuristics we found online and customized the implementation. Similarly, we used an epsilon-greedy policy with decay to facilitate exploration vs exploitation (especially at the beginning of training). The exploration side of the policy also had its own epsilon-greedy policy with decay to choose between picking a random action and the action that gives the best board value in order to have more diverse experiences. Hyperparameters for training and heuristics were chosen arbitrarily from trial and error. During training, Q values are saved that can be loaded later as weights

Assumptions

We make numerous small assumptions in our helper methods. For example, updateGrid places the current piece, which assumes it is valid ahead of time. We also assume that the input is not a 0x0 grid, though if it is 0x0 then it will still run but just do nothing. We also assume that the y-value returned from dropHeight corresponds with the y-value of the lower-left section of the piece's bound box when it's dropped from infinitely above. Additionally, when comparing two pieces, we compare both their types and rotation even if the rotation wouldn't affect the body of the board (ie. square). Finally, we assume that the computer has enough memory to handle working with the brain.

Scope of Solution

A huge limitation of our brain is that quantized Q learning reduces the problem into a much simpler version that loses a lot of information. Subsequently, the performance of the brain is much worse than what it would be if we applied a convolutional neural network to the direct board for approximating the Q function. If we were to regain information, then Java would run out of memory and it wouldn't work. Speaking of memory, we are inefficient with allocating memory, especially in TetrisPiece. Each piece stores a precomputed matrix of body and skirt values for each rotation, meaning there are 28 arrays for only 1 configuration. Obviously that is not ideal, but it is a tradeoff we made to make the getter methods O(1). On the positive side, QLearningBrain can be trained on any dimensions of Tetris as long as its pooling factors are adjusted accordingly.

Quality of Solution

Our code uses many helper methods to segment the computation into clear and distinct processes. There was not much abstraction used for this project, rather we used decomposition to make it clean and concise. Prioritizing encapsulation, methods were set to private whenever possible. For the brain, calculations were decomposed to separate objects with implicit hierarchies (QLearningBrain -> QTable -> QState + Memory -> QAction) with each object only having access to the information it requires and nothing more. We added helpful comments and variable/method names for guiding through the complex brain code.

Karma

Our karma was the QLearningBrain (see solution design above for implementation details). We probably could have made a better AI if we used libraries or deep RL approaches, but out of curiosity we wanted to make a standard Q learning approach for Tetris. It doesn't work great on its own, but when it has the heuristic for assistance it can play games in the low thousands of placed pieces. While it was a fun proof of concept, after seeing other Tetris AI projects, it seems as if the genetic algorithm approach is superior.

Also it is key when you are loading the Q value pretrained weights in qTable.txt that the path to it in JBrainTetris.FILEPATH is actually referencing qTable.txt given your working directory. Current file path is src/main/java/assignment/qTable.txt. If it can't be found, a System.err message will be printed and the heuristic will be run. Making JBrainTetris.FILEPATH to null will initiate training the brain.

Interesting Results

A fun board to play on is one with a width of 4. Since a horizontal stick has a width of 4, when the piece is placed it looks like it automatically disappears. Additionally, to test if pooling was a significant loss of information when quantizing the state space, we did a lot of testing on a 5x10 grid. It was there where we found that the brain was learning how to clear rows and play well, but correspondingly we found how difficult it is to play on such a small board with pieces that large. Finally, a fun test we had was to hold a rotate key (clockwise or counterclockwise) and see what happens. Due to wall kicks, there are instances where the piece appears to be hovering for a long time. The hovering would go on forever, but it eventually terminates when a DOWN timer is called that places the piece.

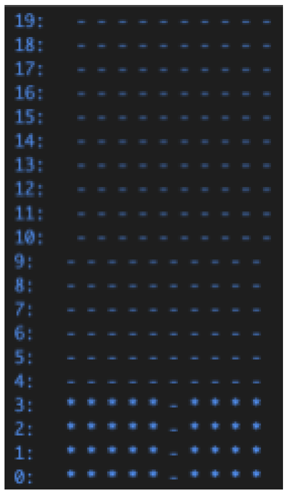
Problems Encountered

One issue with developing Tetris was that we first started with TetrisPiece, the most basic part, and worked our way up. This allowed for simplicity but made it difficult to test if our code was properly working until we finished most of TetrisBoard. We also originally used dropHeight to drop the piece, but found that in cases where there is an overhanging piece (see testing), drop height will teleport the value back up. Subsequently, we switched to a while loop that runs down many times until it can't. While this was accurate, we found that when the AI was dropping from the top that it was extremely slow. Thus, to speed up our code by a factor of almost 20x, we first checked if there were any pieces above the current piece and if there weren't then we used dropHeight, otherwise we ran down until we couldn't. We also initially only had shallow copies in our copy constructor but found issues arised, so some data structures had to be deep copied.

Testing

Since there wasn't an easy way to verify if the overall board state was correct for randomized inputs, it wasn't reliable to perform full blackbox tests, so we relied more heavily on unit testing, specifically white box unit testing. These tests primarily evaluated if the moves themselves matched up with our expectations or if the board would function properly under certain conditions like if there is a smaller dimension board.

Row clearing grid environment



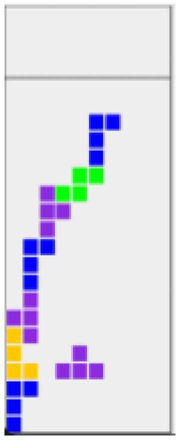
```
19:  - - - - - - - - - -
18:  - - - - - - - - - -
17:  - - - - - - - - - -
16:  - - - - - - - - - -
15:  - - - - - - - - - -
14:  - - - - - - - - - -
13:  - - - - - - - - - -
12:  - - - - - - - - - -
11:  - - - - - - - - - -
10:  - - - - - - - - - -
9:  - - - - - - - - - -
8:  - - - - - - - - - -
7:  - - - - - - - - - -
6:  - - - - - - - - - -
5:  - - - - - - - - - -
4:  - - - - - - - - - -
3:  * * * * * _ * * * *
2:  * * * * * _ * * * *
1:  * * * * * _ * * * *
0:  * * * * * _ * * * *
```

Automated Black Box Unit Testing

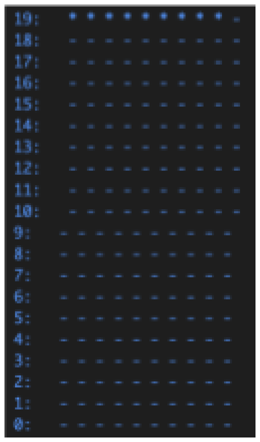
- To verify that the rows were clearing as expected, we created our own grid environment with an empty column down the middle. As our current piece was a vertical stick, the expected behavior would be an empty grid, as all the rows would clear. Thus, we compared the board after applying the move to an empty board

- One of the edge cases that we encountered that could have some unintended consequences was if there was an overhang. For instance, if there was a large stack with a lot of empty space underneath and we tried to place a piece under it, then the piece should be expected to be placed where it landed, not above the piece in the stack. For this test, we generated a nearly full row at the highest y-value and dropped in each of the 7 pieces just below it, checking to see if it would hit the floor of the grid or try to teleport above the overhang row

Overhang example



Overhang grid environment



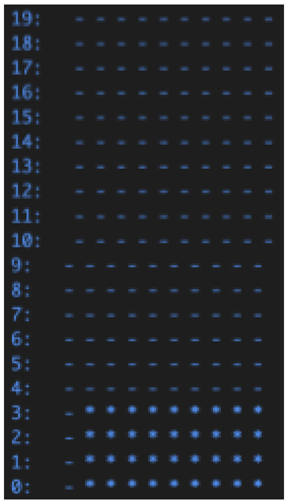
```
19:  ■ ■ ■ ■ ■ ■ ■ ■ ■ -
18:  - - - - - - - - - -
17:  - - - - - - - - - -
16:  - - - - - - - - - -
15:  - - - - - - - - - -
14:  - - - - - - - - - -
13:  - - - - - - - - - -
12:  - - - - - - - - - -
11:  - - - - - - - - - -
10:  - - - - - - - - - -
9:  - - - - - - - - - -
8:  - - - - - - - - - -
7:  - - - - - - - - - -
6:  - - - - - - - - - -
5:  - - - - - - - - - -
4:  - - - - - - - - - -
3:  - - - - - - - - - -
2:  - - - - - - - - - -
1:  - - - - - - - - - -
0:  - - - - - - - - - -
```

Automated White Box Unit Testing

- For each of the moves besides drop, we tested if the move actually moved each piece in the proper direction. For instance, for moving right we checked if calling the move method would indeed increase the x-value by 1 for every piece. We also made sure that this test would throw an IllegalArgumentsException when given a spawn point out of bounds, which we verified in a try catch block. For rotations, we confirmed that the rotation index would increase (clockwise) or decrease (counterclockwise) by 1 and correctly applied mod 4 when it would need to start around at 0 for the indexes. By testing rotations, we also tested the wall kicks as edge cases to see if the movement against the wall was indeed valid

- Ensuring that the currentPiece and currentPiecePosition would get updated appropriately, we checked that they were both initialized to null and spawned correctly
- To test if each block would be placed correctly, we recognized the variables that were changed while updating the grid, clearing any rows that are full, and resetting the current piece. With this in mind, we created a test environment with four nearly full rows and tested if placing a horizontal stick in the center would update the variables appropriately. For instance, in this grid environment, the max height is expected to be 5 and the current piece is expected to change to null
- The last whitebox test we had was verifying if the dropHeight method works as intended. Using the same grid environment as placing the piece, we verified if the dropHeight would be 2 if we were placing a horizontal stick in the middle of the grid

Placing piece and drop height grid environment



```
19:  - - - - - - - - - -
18:  - - - - - - - - - -
17:  - - - - - - - - - -
16:  - - - - - - - - - -
15:  - - - - - - - - - -
14:  - - - - - - - - - -
13:  - - - - - - - - - -
12:  - - - - - - - - - -
11:  - - - - - - - - - -
10:  - - - - - - - - - -
9:  - - - - - - - - - -
8:  - - - - - - - - - -
7:  - - - - - - - - - -
6:  - - - - - - - - - -
5:  - - - - - - - - - -
4:  - - - - - - - - - -
3:  _ * * * * * * * * *
2:  _ * * * * * * * * *
1:  _ * * * * * * * * *
0:  _ * * * * * * * * *
```

<u>Manual Testing Edge Cases</u>

- We determined that some of the tests were far too complicated to realistically create test code for, thus we decided to test them visually
- Since most of our other tests relied on the size of the grid to perform their test, we weren't able to reuse those tests at different grid sizes, so we decided to manually test whether Tetris was implemented properly at any given dimension. Starting from a negative dimensions and a 0x0 grid, we increment our size until it decently surpassed the default 10x20 grid and check these edge cases of smaller and larger grid inputs

- We also tested the infinite spin edge case (see interesting results) to see if our code would allow for wall kicks to occur infinitely

<u>AI Testing</u>

- *Test making valid move (automated white box):* In order to double check if each move made is valid, we check that over the course of playing 100 games if any moves made makes the piece invalid or if the result is out of bounds, signifying an impossible move was requested. This is done by extracting and checking if current piece and move are valid using JUnit
- *Check memory in bounds (automated white box):* Since we set a cap for maximum memory size, we want to make sure that it doesn't go over, which would lead to too large of a memory database slowing training. This is done by running 100 games and checking if the memory every exceeds that limit using JUnit
- *Loading weights (manual black box):* Sometimes an invalid file is asked to be loaded for the QTable weights so we have a test for what would occur with invalid weight files. A System.err print statement will be called when this occurs, and we checked a few file configurations to check if file isn't found or if the weight matrix is too large or smaller than state action space its loading into
- *Check last move placed (manual black box):* Our AI will always end its QAction with a drop, meaning that the last action should be a drop. To check this, we set up a print statement that will notify the user if a drop is missing, which we found never occurred
- *Check encodings (automated black box):* We want to make sure that when we encode and decode the boards and states respectively, that they are correct. Instead of making a

mapping to check what it should be, we test if they are each other's inverse (which should occur). For each possible state, we decode it then encode it and compare if they are the same states using JUnit

- *Check weights file to see Q values (manual white box):* We saved the Q values into files and directly looked at the values to see if they were diverging. We don't know which value should be the maximum, but it should converge on a few best moves with max values which we found. We also switched weight initialization from random to 0 to get an idea of which states were being trained on to see how much we should train for
- *Display training and testing results at key epochs (visual black box test):* To get an idea of how it was training and testing, we printed the score of training and testing games at specific epochs to get an idea of how/if it was learning over time. The fact that the score improves over time showed us that it was both learning and picking optimal actions with the trained Q values. This helped for hyperparameter tuning and hypothesis testing
- *Test on different board sizes (visual black box test):* We experimented with both training and testing on a 5x10 board and it worked smoothly. It does need to be retrained though
- *Brute force board testing using AI (automated black box):* We trained and tested the AI on an extremely large amount of games which by default, with the above tests, checked how the board performs over many states

Social Impact

Accessibility needs to be considered when designing technology because it is important that it doesn't limit the population using it. If there exists only a certain portion of people who can use the technology, then it won't be able to reach its full potential and will significantly reduce its scalability, not to mention that it would be a poor idea from a business standpoint as someone trying to make money. If Tetris were to be made more accessible to say someone visually impaired, it would make sense that each "tick" would represent a sound and colors would be distinguished as much as possible for someone who might suffer from color blindness. This way, any limitations that a user who suffers from those conditions might have would be minimized, if not completely eliminated. For instance, one of the features Tetris could include would be a color blindness setting that would adjust based on the user's color blindness diagnosis like Protanopia and Tritanopia (Srivastava).

Outside of Tetris, some of the most common examples of accessibility in technology include using accessible keyboards and speech-to-text recognition to allow those physically and visually impaired to easily type (Ciccarelli). This is especially helpful, as it allows people with these disabilities to still get use out of these technologies, when it would otherwise be nearly impossible. On top of these, to help better the field as a whole, technology creators need to perform tests on accessibility, just like any other tests they would do on their software. Speaking from personal experience, we have both worked with Chrome Lighthouse and used their built-in accessibility tests, which was particularly helpful for finding areas that might be difficult for users with disabilities. If companies or software developers routinely perform and adjust based on these tests, then they would acquire a greater population of users, which would be beneficial for both the business and the users.

Pair Programming

- 9/23: 2 hours together to understand and plan project
- 9/24: 1 hour of A driving, 1 hour of B driving

- 9/27: 1 hour of A driving, 2 hours of B driving
- 9/30: 1 hour of A driving, 2 hours of B driving
- 10/1: 4 hours of A working separately
- 10/4: 1 hour of B driving
- 10/5: 1 hour of B driving
- 10/6: 10 hours of working together on separate topics

We worked better with maintaining consistency with GitHub this project which was nice. We also went down a rabbit hole of building the Q learning brain which was fun to get both of us involved in. We definitely both learned a lot and general work went pretty smoothly as always.

Works Cited

Ciccarelli, David. "7 Innovations That Are Transforming Accessible Technology." *Voices*, 10 Sept. 2022, https://www.voices.com/blog/accessible-technology/.

Srivastava, Sudeep. "How to Design Accessibility App for Visually Impaired?" *Appinventiv*, 22 Aug. 2022, https://appinventiv.com/blog/design-accessibility-app-for-visually-impaired/.