**User Manual/ Description of MKB Components**

Table of Contents

[1 TetrisPaul4 1](#_Toc386057129)

[1.1 Training tab: 1](#_Toc386057130)

[1.1.1 Train from Regular Game! 2](#_Toc386057131)

[1.1.2 Training from a Fail Case 3](#_Toc386057132)

[1.2 Watch Tetris Tab 5](#_Toc386057133)

[2 ProperTrainingCaseConverter 7](#_Toc386057134)

[3 ProperTrainingCaseConverter2 8](#_Toc386057135)

[4 Creating ANNs 8](#_Toc386057136)

[5 Genetic program / large scale validator 10](#_Toc386057137)

[5.1 Using Genetics as a Genetic Algorithm 10](#_Toc386057138)

[5.2 Using Genetics as a Large Scale Validator 13](#_Toc386057139)

# TetrisPaul4

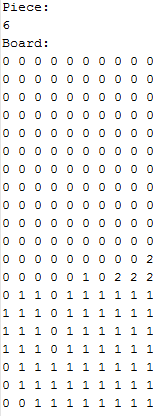
As the name suggests this program is the fourth and final iteration of the Tetris player known as MKB. Immediately, one should notice that there are two tabs—“Watch Tetris” and “Training.” Each tab’s purpose is quite different and should be thought of as entirely separate programs. For simplicity of distribution however, they were combined into one program and separated into tabs. The “Watch Tetris” tab is for configuring and watching the performance of a Tetris player. The “Training” tab is the program used to gather training data to train the artificial neural networks that can be found in the “Watch Tetris” tab.

## **Training tab:**

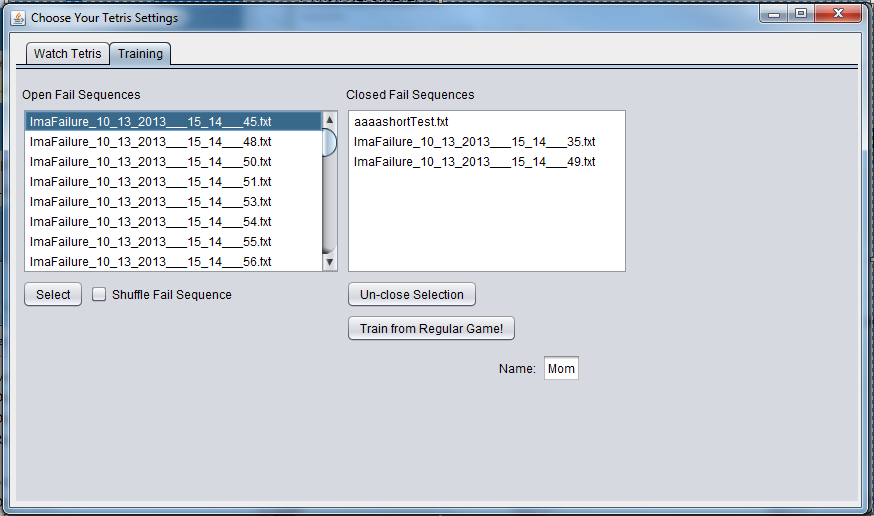
The training tab, as mentioned, is the interface to the program that collects training data from a user to train artificial neural networks (ANNs). A user has two options to generate training cases. They may disregard all other options on the screen and click “Train from Regular Game!” or they may choose a “fail sequence” to work through.

### Train from Regular Game!

This will start a game of Tetris from the beginning and the user is prompted to enter their name. The currently falling tetromino is then rotated and shifted left or right to where the GA “thinks” the piece should be placed. A “shadow” of the GA’s choice appears on the game board as red. If the user has no objections to the placement, a simple “hard down” (down arrow) will place the piece as the GA would like and no data is recorded. However, if the user disagrees with the GA’s choice placement, the user simply need move the falling tetromino to where they prefer it. The action of placing the tetromino anywhere other than the red “shadow” results in two training cases being recorded by the program—the *bad* case of what the GA wanted to do, and the *good* case of what the human chose to do.   
 Note: I chose this method of collecting training cases by correcting the GA instead of just letting a user play Tetris and recording ALL of their moves for the following reasons. There may be more than one good place to put a piece. For example given an empty game board what is the difference between putting a line piece all the way on the left or all the way on the right? Correcting the GA only on clearly bad placements ensures trivial differences in placement like this are ignored.

After the user has placed 100 pieces (corrected or not), a pop up informs the user that whatever corrections they have made have now been recorded (written to a file), and if they would like to exit, it is now safe to do so without losing data. Corrections are stored in .\\Resources\\Corrections\\<TRAINERNAME>\\<TIMESTAMP> relative to the folder TetrisPaul4.jar was started. Two txt documents are created *each* time a pop up appears to the trainer—the positive training cases (human placements) and the negative (what the GA wanted to do). Note that < TRAINERNAME > is the name the trainer enters, and <TIMESTAMP> is a timestamp of the current date. The names of the txt files are generated using a timestamp with the word “GOOD” or “EVIL” depending on if the file contains positive training cases or negative respectively. An example of exactly what is written in the txt document is shown at left.  
 The integer following the line “Piece:” is always in the range 1-7 indicating the unique tetromino code (note this is technically unnecessary, but to “read” what piece was placed is not worth the effort of simply writing the piece code to the file). 0’s represent empty spaces in the game board, 1’s represent filled blocks, and 2’s represent the placement of the tetromino in question. The example to the left is one *positive* correction made by the trainer. One of these txt documents will contain anywhere between 0 and 100 “PlacementRecords.” This information about how and where training cases are stored is important when training a neural network as we will get to later.

### Training from a Fail Case

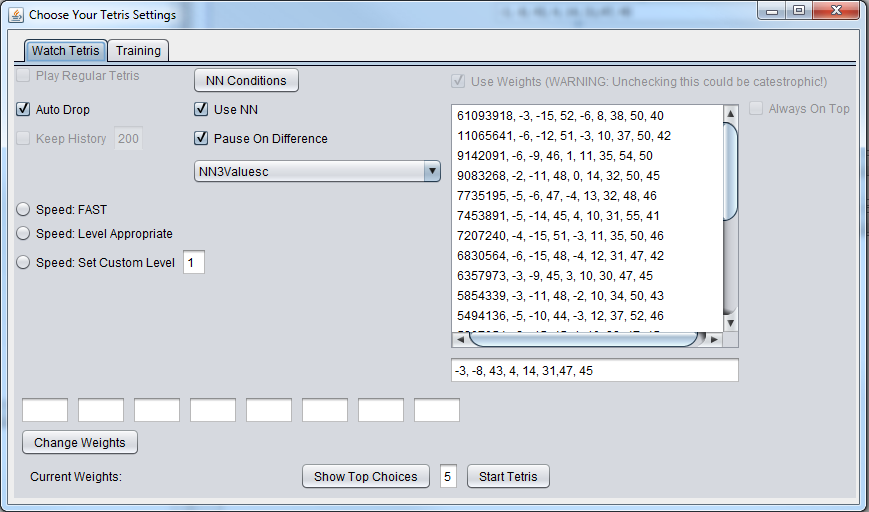


If the user trainer were to only provide training cases from the aforementioned “Train from Regular Game!” we would not have a fair representation of the Tetris domain. Unless the trainer received a particularly difficult sequence of Tetrominos (a rarity), the training cases received would only involve a minimally high stack height, maxing out at around a stack height of perhaps 5. Perhaps the definition of “how human” a placement is is relative to the stack height. Without training cases representing a higher stack height, an ANN will be unprepared for such situations.

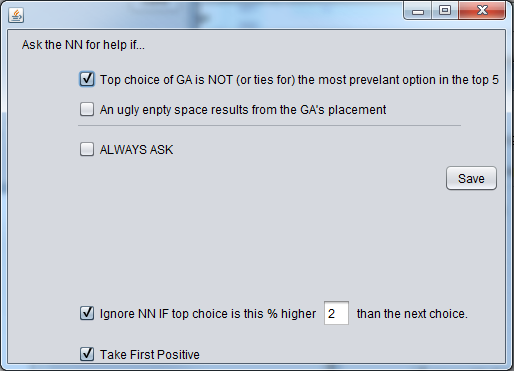
To ensure we include “higher” training cases, we require the user to also provide training cases from playing through a number of “fail sequences.” A “fail sequence” is simply the last 100 piece and board configurations the GA encountered before a game over. I created a temporary program to play simplified Tetris and save the final 100 board and piece configurations. Approximately 400 fail sequences were recorded using the initial weights of . To produce training cases from fail sequences, follow these instructions.

1. Enter the name of the trainer in the textbox labeled “Name:” Since training from a fail sequence requires frequent game starts (a game start for each fail sequence), this text box saves the trainer from having to re-enter their name for each fail sequence and also helps with consistency (i.e. Paul vs paul vs PAUL).
2. Select a fail sequence from the list in the listbox labeled “Open Fail Sequences.”
3. Click the “Select” button to start the Tetris.
4. Similar to training from a regular game of Tetris, the user is then presented with a board configuration (in yellow) and in red is the proposed placement by the GA. The figure is also automatically rotated and shifted to fall into the “red” placement. If the trainer disagrees with the proposed placement, They simply need to move the tetromino to the desired location and rotation and place the piece. If they agree with the placement, they simply need to hard drop (hit down arrow) to place the piece in the proposed “red” placement.
   1. Option: Since the fail sequence is, in fact, a fail sequence, the corrections are “ignored” by the program. What we mean by that is that the next board and piece presented to the trainer are *as if* the previous tetromino were placed in the “red” placement originally proposed by the GA—whether the user corrected the placement or not. Making a large number of these corrections only to be “ignored” is psychologically fatiguing. To combat this fatigue, at any time a trainer can select the “Shuffle Fail Sequence” checkbox on the Training tab to go through the fail sequence in random order. In this manner, the next board and piece configuration should be different enough from the prior board and piece configuration to not feel “ignored,” but rather just presented with a completely different scenario to possible correct.
   2. Option: if the trainer makes a mistake (as humans tend to do), we do not want their mistake to be considered a positive placement. For this reason, there is “Oops” button the user should click should they make a mistake right after a mistake is made.
5. At the end of the fail sequence (after 100 tetrominos), The user is again presented with a popup visually confirming that their training cases are being recorded and the program can be exited without loss of training cases. Once “game over” appears on the game board, the game should be closed and a new fail sequence can be selected for play.
   1. Option: After a few moments, the Fail sequence selected to be played at the start will no longer appear in the “Open Fail Sequences” listbox and appear in the “Closed Fail Sequences” listbox. This is to keep track for the trainer which fail sequences have been played through and which ones haven’t. These lists are perpetuated betwixt runs of the program. If for some reason, the fail sequence needs to be put back in the “Open Fail Sequences” list (perhaps a trainer forgot to click “Oops” after mistakes) the trainer can simply select the fail sequence from the “Closed Fail Sequence” listbox and click the “Un-close Selection” button to move the fail sequence back into the playable listbox “Open Fail Sequence.”

## Watch Tetris Tab



This tab is a GUI designed for viewing the performance of different configurations of a Tetris player. Note that the grayed out options are extensions yet to be fully implemented. A user (viewer) can configure the player using the following options. A viewer may change any option “on the fly.” To view the configured player, click the “Start Tetris” button.

* The “Auto Drop” checkbox (located in the upper left), will do as the name suggest when selected—the tetromino will hard drop immediately instead of falling one row per timestep.
* There are three options for the speed of play (located on the left side).
  + Speed: Fast
    - This option plays as quickly as the processor allows with an explicit time delay of 0
  + Speed: Level Appropriate
    - This option plays Tetris at a speed determined by the level. Note that this option makes no speed difference after a level of around 30.
  + Speed: Set Custom Level
    - This option, when selected, will play the Tetris game at a custom speed by “pretending” to be at the specified level in the adjacent textbox. This option is useful for slowing down play when examining the performance of a particular configuration.
* The “Use NN” checkbox does as the name implies. When selected, the selected ANN in the drop down list below is used in the Tetris player according to the conditions specified in the “NN Conditions” menu (accesses by clicking the “NN Conditions” ).
* The “NN Conditions” button opens a menu to configure specific conditions surrounding the use of an ANN. The setting used for the best performance of MKB are shown above.
  + If the first checkbox is selected, the “opinion” of the GA is put under review by the ANN is the top choice of the GA is not the most frequently occurring placement in the the top 5 rated placements of the current tetromino (See main paper for more information)
  + If the second checkbox is selected, the GA’s “opinion” is put under review if a new hole is created by the GA’s top choice for placement. Note that is option conts new holes *after* lines are cleared so that if a “temporary” hole is created in the process of clearing a line(s), this is not a trigger.
  + If the “ALWAYS ASK” check box is selected, the GA’s top choice is always put under review by the ANN.
  + If the “Ignore NN IF…” checkbox is selected (this is the configuration for the “back out” feature mentioned in the main paper) the player uses the “opinion” of the GA *even if* the ANN has been triggered to give its “how human” rating if the GA’s top choice is more than the specified percentage higher than the GA’s second choice. The idea behind this option is that if a particular placement is rated much higher than the next best option, it is higher for good reason and should be selected as the tetromino placement regardless of what the ANN’s opinion on the placement is.
  + The “Take First Positive” checkbox, when selected, does the following: When using the specified triggers of when to ask for the ANN’s opinion on the top 5 placments, instead of choosing the highest rated ANN rating, take the first positive rating when examining the top 5 choices of the GA from first choice to last. The idea behind this option is that the GA does do a pretty good job of selecting the best placement. However, from time to time, it requires a correction. The ANN’s purpose can be thought to be to just to answer yes or no to the question, “Is this a placement a human would do?” Regardless of “how human” a move is, take the first one that satisfies the condition of being “human like.”  
    Note: The how human rating given to each of the top 5 GA’s placements is a double value in the interval [-1,1] where 1 is “human-like” and -1 is NOT “human-like.” Therefore, anything >0 can be thought of as a move a human would do, and anything <0 can be thought of as a move a human would NOT do.
* The “Pause on Difference” check box, when selected, will pause the currently running Tetris game if the ANN has caused a different placement to be selected than the GA’s top choice for a particular tetromino and game board. This option is useful for evaluating the corrections of a particular ANN.
* The drop down box (located right below the “Pause on Difference” check box is where the viewer can select which ANN they would like to use for Tetris play. (NN3Valuesc has experimentally shown the best performance and is the ANN used in MKB)
* The list box on the left side is where the viewer can select a particular set of weights to use to generate the top 5 choices. The first number in each option in this list is the score reported when initially exploring different weights and their performance. After selecting a set of weights, the weights will appear in the text boxes along the bottom (hover over each text box with your mouse pointer for a description of what each weight is for), and are NOT yet being used by the Tetris player. Once the weights appear in the text boxes, the user can then alter them manually if they wish, and to implement the new weights in the Tetris player, click the “Change Weights” button to make the change.
* The “Show Top Choices” button will open a new window showing the top x (number specified in adjacent textbox) choices by the GA for each tetromino during game play. They are listed from left to right with the leftmost being the top choice and the rightmost being the xth choice. Underneath each proposed placement is displayed the GA’s rating and the ANN’s rating if the ANN’s opinion was sought.

# ProperTrainingCaseConverter

This program is simply used as a mass converter of the above mentioned training cases to a format usable by ANN applications such as Neuroph or Encog (never Neuroph. Never ever Neuroph. See Advice paper). Here are the instructions for its use

1. Move all the training case txt files generated as described above into one folder.
2. Run ProperTrainingCasesConverter.jar and navigate to the folder where put all the files you wish to convert from the previous step.
3. The converted training cases will all be written to a single file in the same folder you ran ProperTrainingCaseConverter.jar titled “RESULTYAYYAYYAY.txt”

Each training case becomes one row in the RESULTYAYYAYYAY.txt file. ProperTrainingCaseConverter converts training cases in the following way: the highest point of the placed tetromino is found and that row and the 9 rows below it are used to create the training row. If there are not enough rows in the game board, this is a problem since ANNs require the same input size across training cases. Therefore I needed pseudo-rows are created. These pseudo-rows consist of nine 1’s and one randomly placed 0 with the condition that the 0 cannot be generated under the placement of the tetromino for that particular training case. This is because we do not want to “confuse” the ANN by perhaps taking a perfectly good placement and generating “holes” underneath the placement. At the same time, we do not want to generate a wholly completed row (all ten 1’s) since this is a *good* thing to have (clears a line) when perhaps the placement should not be associated with a good outcome. After all this is completed, the 10 by 10 matrix is then converted to a one-dimensional array to be written to the output file. After these 100 numbers representing the placement are written, the 101st number in each training row is either a 1 or -1 depending on whether the placement is a positive (human move) or GA mistake respectively. Each training case has its mirror image included in the RESULTYAYYAYYAY.txt file as well.

Let it be noted that all neural networks trained using training data in this format performed quite poorly along with poor convergence. For this reason ProperTrainingCaseConverter2 was created to function similarly, but to create training cases in a different format.

# ProperTrainingCaseConverter2

As mentioned, ProperTrainingCaseConverter2 works very similarly to ProperTrainingCaseConverter in the sense that a user first places the training cases to be converted for ANN training in one folder, starts ProperTrainingCaseConverter2.jar, and navigates to that folder. The difference is, is that instead of the 100 inputs followed by the desired output, we now have 8 inputs followed by the desired output. These 8 inputs are the heuristics listed in the main paper applied to the given tetromino placement. Neural Networks trained on data in this format performed marginally better both in Tetris play as well at convergence during training.

# Creating ANNs

This was done using the great and wonderful tools provided by Jeff Heaton in the Encog Workbench ([www.heatonresearch.com](http://www.heatonresearch.com)). For a more complete guide to ANNs and how to use his workbench, see the aforementioned site and wiki. To import the training case file we created from using the ProperTrainingCaseConverter or ProperTrainingCaseConverter2 into Encog for ANN creation, follow these steps

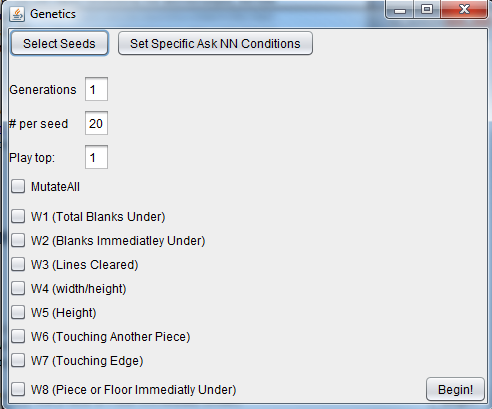
1. Change the file extension of the training cases file from .txt to .csv (the format welcomed by Encog)
2. Once a new project has been created in the Encog Workbench, click Tools , Generate Training Data, and then select the first option, “Copy Training Set from File.
3. Enter what you would like the Encog training file name to be in the “File Name:” textbox.
4. Click OK and navigate to where your newly converted csv file is and select it.
5. Whatever you chose to name your Encog file will now appear in your list to the left. Right click it and select “Analysit Wizard.”
6. Since we are dealing with numbers, change “Goal” to “Regression” and your “Normalization Range” to “-1 to 1” if you plan to use TANH as your activation functions or select “0 to 1” for sigmoid activation functions. I found for my data the best results were from using TANH activation, but NOT normalizing the inputs. Select the “Tasks” tab if the current window if you wish to not normalize your data.
7. At this point, one may wish to choose now to generate code from the resulting ANN, but this option can be done later as well. If you wish to set your code generation now, select the “Code Generation” tab in the current window and select a language (Java for me). You also likely want to check the box labeled “Embed Data.” This generates your ANN in your target language and hard codes the weights for the neural network.
8. You will then be presented with the contents of a “.ega” file. This is a script file that Encog Workbench understands ( see here for an example of how to edit this file <http://www.heatonresearch.com/wiki/Workbench_Classification_Example> ). You will want to change all the “isclass” fields to 0 (indicating false) since the data represents numerical data and not classification data. If there are any auto-detected classes, you will also have some instructions inside the classes tag. Delete these since, again, we are not dealing with classes, but numerical data.
9. When satisfactory, click “Execute” to generate the training cases as specified in the “.ega” file and begin Network training!  
   Note that in the ega file, a neural network structure is suggested. This of course can be changed, or if the neural network is not satisfactory, another can be made using the workbench and selecting thr ANN options in a GUI format.  
   Note: In training, do not be afraid to use the Perturb option. This can have a drastically good effect on training. You may need to Perturb 20-30 times before the ANN figures out the problem with decent error.
10. To use an ANN in code, examine how the Neural Networks library is set up. But for starters, right click the file to the left that contains “(BasicNetwork)” following the name specified in a previous step, and select “Generate Code.”
11. Select your target language and “Embed Data.”
12. At least in the Java generated code from Workbench version 3.2.0, you will need to change two parameters in the “createNetwork()” method in the line that calls “methodFactor.create(…).” The last two parameters passed to this method are both 0’s but the first 0 needs to be replaced with your number of input neurons and the second 0 replaced with the number of output neurons. You will get a runtime error if you do not make this change.
13. In whatever project you wish to use this generated code, you will need to include “commons-math3-3.0.jar,” and “encog-core-3.2.0.jar” –both available from [www.heatonresearch.com](http://www.heatonresearch.com). You probably downloaded them together when you downloaded the Workbench.  
    Note: The Encog Workbench generated code is nice, but you are still in charge of converting to the proper inputs for the ANN and normalizing the data if the ANN was trained on normalized data. Examine the NeuralNetworks library component source code of MKB works for an example implementation.

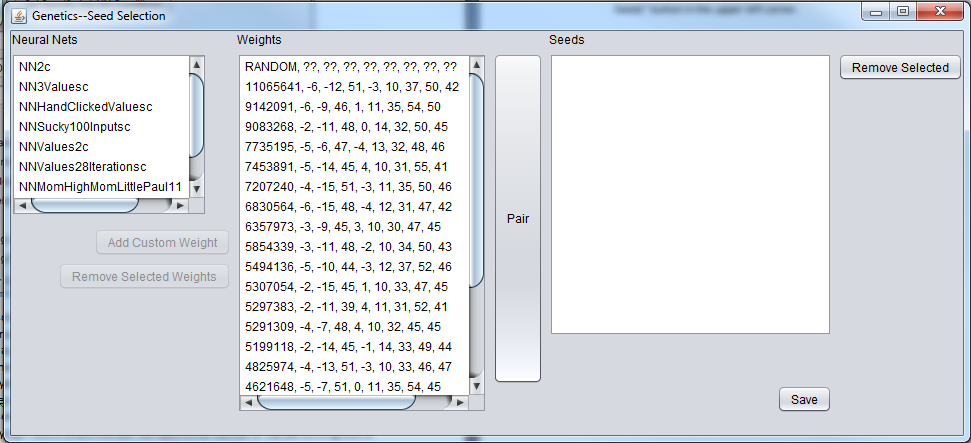
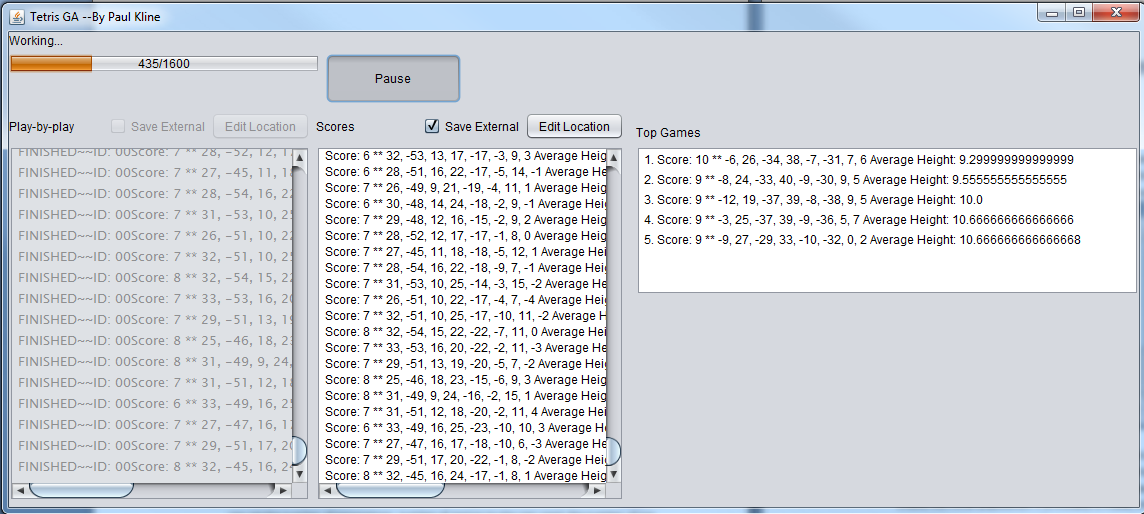
# Genetic program / large scale validator

Since both of these tasks involve playing a large number of (simplified) Tetris games, this is a single program that can do either by changing some options.

## Using Genetics as a Genetic Algorithm

Follow these steps to use “Genetics.jar” as a genetic algorithm. This is how MKB’s amazing set of weights was found when exploring random weights while using an ANN (NN3Values is the ANN usedfor yielding the best results. It did not disappoint).



1. The above screenshot is what the user of Genetics.jar will first encounter. Here is were you can set the number of generations, number of games to play per seed, the number of top games to play, and which of the weights you would like to mutate.
   1. Generations—this is simply the number of generations you would like to have played.
   2. # per seed—this is the number of Tetris games that are played *per seed*. We will get to seed selection soon.
   3. Play top—this is where the user can specify the number of top performing sets of weights to keep from a particular seed from a particular generation to be a seed in the next generation. For example, if generations is set to 3, # per seed is set to 300, and Play top is set to 2, and we start with 3 seeds to generate our initial population, we would have the following: The population of the first generation will be 300\*3 since the user has specified that each seed should be played 300 times and we have 3 seeds. Since we are keeping the top 2 results from each seed’s children the population size of generation two will be 300\*6 since we specified to play 300 games per seed, and we are keeping the top 2 games from each prior seed to generate our next population (generation 3 will have a population of 300\*12). One must be careful here since population size can easily get large very quickly.
   4. The checkboxes allow the user to choose which weights they would like to undergo mutation. If the checkbox corresponding to a particular weight is not selected, it will remain whatever the seed had for that weight throughout all play.
2. Before clicking the “Begin!” button to start the manic Tetris playing, the initial seeds of the initial population must be specified. You can specify the initial seeds by clicking the “Select Seeds” button in the upper left corner. This will result in displaying the following GUI. 
3. To create a seed, simply select an ANN from the list on the left (none if no ANN is desired), and a set of weights from the adjacent list and click pair. The pair you have just created will now appear in the “Seeds” list on the right. You may continue to pair seeds as desired to create the initial population. If a mistake is made, simply hold Ctrl and select all the seeds you wish to remove and click “Remove Selection” to remove them.
4. Click “Save” to close the Seed Selection window.
5. If *any* seed uses an ANN, click the “Set Specific Ask NN Conditions” to set these conditions as specified in the TetrisPaul4 section. Otherwise if no ANN is part of seed, it doesn’t matter what these conditions are.
6. Click “Begin!” to begin the manic Tetris playing. The following GUI will then be presented to the user.
7. The total number of Tetris games that have been requested to be played is calculated and how far along the Genetic program is in playing them all in indicated in the top left corner. However typically as the program progresses to later and later generations, the Tetris games will take longer and longer to play, so this status should not be interpreted as a “percent completed” in respect to time anyway.
8. The output of what the program is doing (i.e. the result of Tetris games using particular weights) can be seen in the text area labeled “Scores.” As a default, the contents of this text area are written to a file created in the same folder from where “Genetics.jar” was started. However, this location can be changed by clicking the “Edit Location” button should the user desire the output be placed somewhere else (Note the previous file will NOT be deleted).
9. For a quick look at how “things are going,” look under the “Top Games” text area to view the top 5 highest scoring Tetris games played by the program thus far. Revel in the neatness of this feature for step 9.
10. Depending on the seeds selected and other features, this program could take days or weeks to run to completion. This is why the “Save External” option is so nice. If the program is interrupted at any point (say, from an automatic update restart), every Tetris game played up to that point is saved safely in a file. Revel in the necessity of this feature for step 10.

## Using Genetics as a Large Scale Validator

To use “Genetics.jar” as a large validator is fairly straightforward if you follow the following steps.

1. Select the pair(s) you would like to large scale validate as seeds in the way described previously.
2. Set the specific ask NN conditions if necessary.
3. Make sure NONE of the weights are selected to mutate.
4. Now comes a choice. Say you want to see how a particular NN-weight pair performs over 1000 simplified Tetris games. One way to do this would be to set “Generations” to 1 and “# per seed” to 1000 (“Play Top” doesn’t matter since we are only playing 1 generation). The way Genetics is structured however, is that it will start 1000 threads at once to play 1000 games of Tetris. The user will likely notice performance lag which becomes a problem if the program needs to run for days/weeks. Since the population of generation determines the number of simultaneous Tetris games, a much more user friendly way (no performance lag) is to limit the number of Tetris games being played at any one time to 20. This can be accomplished by setting “Generations” to 50, “# per seed” to 20, and “play top” to 1. This will play all 1,000 games, but in chunks of 20 games at a time. ☺

This concludes documentation for all relevant programs created or used in the creation of MKB.