Tetris-Playing Agents Generated by Genetic Algorithm with Wisdom of Artificial Crowds

Balsam M. Hindi, John R. Kinney, Kelly A. Morris, Kevin C. Reilly
[Computer Engineering and Computer Science]
Speed School of Engineering
University of Louisville, USA
bmhind01@louisville.edu
jrkinn02@louisville.edu
kamorr05@louisville.edu
kcreil01@louisville.edu

***Abstract* -- We used a Genetic Algorithm with a Wisdom of Artificial Crowds algorithm to create a Tetris-playing agent. This Tetris-playing agent takes advantage of a heuristic that considers how each piece will affect the current board state based on a number of criterion, and will select the best move accordingly with respect to the current heuristic constants. This approach was fairly successful in terms of the agent evolving its solutions to efficiently play a game. Additionally, the Wisdom of Artificial Crowds approach, in combination with the Genetic Algorithm, served to improve upon the overall average of the expert solutions presented to it by taking the weighted average of the fittest solutions offered to it over a span of several games and generating a unique solution based upon it.**

***Keywords* -- Genetic Algorithm, Wisdom of Artificial Crowds, Tetris, agent**

I. Introduction

A. *Objective*

Tetris is a very popular video game where the player attempts to efficiently pack blocks which fall one at a time from the center of the top of a board as if under the power of gravity. If the player successfully fills an entire row on the board with blocks, that row clears and the player gains points.

The objective of this project was to create a genetic algorithm, augmented by a wisdom of artificial crowds algorithm which would optimize Tetris game play by minimizing aggregate height of all columns, the aggregate differences in height between consecutive columns, and holes present on the board, while maximizing the total number of lines cleared and in so doing maximizing the points accumulated per game [1].

B. *NP- Completeness*

Solving this problem in an offline version of the game, where the piece sequence is known in advance, was proven to be NP-complete by Breukelaar, Demain, et al in their paper *Tetris is Hard, Even to Approximate,* which implies that the version of the game used here would be at least NP-complete to solve as the complexity is increased by the stochasticity of the piece sequence and the partial observability of the environment [2]. This will be discussed further in a later section of this paper.

A problem is classified as an NP-Problem if it is solvable in polynomial time by a nondeterministic Turing machine, and NP-hard if an algorithm used to solve a given NP-problem can be used to solve any other NP-problem. If a problem is both NP and NP-hard, it is considered to be NP-complete [3].

C. *Genetic Algorithms*

The concept of a genetic algorithm is based on the stochastic beam search algorithm; however, in genetic algorithms, a solution is created through many small mutations to a set of solutions, with the intention of creating a superior solution in the end [4]. This is accomplished by initially creating a population of solutions by some means, often through randomness. From this initial population, a new generation is created by combining two strong parent solutions, with the strength of a solution being determined through a fitness function [4]. The combining of parent solutions is accomplished through a crossover function and results in one or more child solutions, with the goal of creating a child solution that is stronger than either parent [4]. By repeating this process, a new generation of solutions is created, which goes on to create a descendant generation of their own; this process continues until some desired result or criterion is met [4]. To maintain genetic diversity within a genetic algorithm, a mutation function is used; this periodically disrupts a child solution by some random means, which keeps the “evolution” of the solutions from stagnating [4].

The wisdom of artificial crowds algorithm takes a number of proposed solutions which are determined to be good solutions for a given problem. It then analyzes these solutions for consensuses and creates a final solution from the agreement found amongst these experts.

D. Tetris

Tetris is a video game created in 1984 by a Russian computer scientist and mathematician named Alexey Pajitnov [5]. Just four years later in 1988, the game was discovered at a trade show in Las Vegas by Henk Rogers who then released it through his company, Bullet-Proof software, for the PC and NES; it went on to sell over two million copies and maintains its popularity today [5]. While there are now many versions and adaptations of Tetris, for this research, the classic NES rules first published in 1985 were used [5]. Tetris gameplay consists of a 10 by 22 block board, where the top two rows are hidden from the player. The next piece to fall will originate from the center of this hidden area. There are seven unique pieces called tetrominos, which are provided in random order to the player. A lookahead piece is provided in a space to the right of the game board so the player has knowledge of the piece in play, as well as the piece that will directly follow. As the game progresses, the pieces fall faster and faster, thereby increasing the difficulty level. The objective of the game is to keep the pieces from pilling up to the top of the board; once this condition is met, the game is lost. To prevent this condition from being met, the player attempts to pack pieces in as efficiently as possible, as anytime an entire row is filled, that row is cleared and the height of the pile decreases. As the piece enters the board, it falls as if it is under gravity at a speed specified by the level the user has achieved. The player is able to rotate each piece as it falls by repetitively striking a specific key; the piece will move through the rotational iterations in order then repeat. The NES rules use a clockwise rotation system, and do not support lock delay, wall kicks, or hard drops; however, soft drops are allowed with an associated reward for doing so [6]. Lock delay is number of frames waited before a tetromino locks into place. In this version of game play without lock delay, the tetromino locks into place immediately upon making contact with the pile or bottom of the board [6]. Wall kicks allow for rotations when the piece is up against a wall. When rotating would take the piece out of the board space, it is accounted for by automatically shifting the piece over to keep it in the board space [6]. In this version, which does not support wall kicks, a piece against a wall would not be allowed to rotate into the next state; rather, the player would need to shift the piece over until the rotation could be made with the piece remaining in the board space. Hard drop is the ability for a player to hit a button which sends the piece to the ground/ top of pile instantaneously; however, this ability is unsupported in this version of the game. Soft drop is the ability for a player to speed a piece’s descent to the ground, thereby getting their next piece faster. Additionally, the player is rewarded 1 point per line down while using soft drop. Points are accumulated by clearing row and are increased in proportion to the level the player has achieved. The value of the points increases exponentially based on the number of lines cleared with a single move [7].

II. Prior Work (Literary Review)

A. Complexity of Tetris

Multiple papers have been written on the complexity of optimizing Tetris game play. The earliest known work found, which considered the complexity of Tetris, was written in 1999 by Kostreva and Hartman called *Multiple Objective Solution for Tetri*s, a technical report published in the Journal of Recreational Mathematics. This paper was referenced in Demaine, Hohenberger, and Liben-Nowell’s work *Tetris is Hard, Even to Approximate,* where they stated that Kostreva and Hartman studied Tetris from the perspective of control theory, and in their work, they leveraged dynamic programming which used a heuristic measure to select the optimal move [2]. Further foundational study was undertaken by John Brzustowski in his work *Can You Win at Tetris* where he answers the question: “Is it possible to beat Tetris?” In this work, Brzustowski creates a theoretical version of TETRIS he calls tetris, where the drop speed of the piece is assumed to be slow enough that the player has all the time necessary to make whichever move they would like. Furthermore, in this theoretical tetris there are no points accumulated, rather the time of gameplay is maximized. If play continues, the player is winning until gameplay ceases, at which point the player has lost [8]. This is directly correlated to how TETRIS points are accumulated; to continue gameplay, the player must clear rows which accumulates points, or they will lose. Therefore, this simplification of the game space is still applicable to the real game [8]. Through analysis of single piece sequences, two-piece sequences, and board sizes of varying dimensions, Brzustowski concludes that there does not exist any winning strategy to beat Tetris. In fact, if Tetris were an adversarial game where the game could pick the next piece to thwart the player, then the adversary is proved to eventually force a loss for the player through the selection of sequential Z and S pieces [8]. It is notable that Brzustowski discussed the concept of making and smoothing what he called bumpy lanes in the proof for lemma 1 [8]. In this Lemma, Brzustowski proves a winning algorithm for a specific set of pieces: square piece and one of the following J, L or I, then a bumpy lane will be created. The algorithm directs the player to fill a bumpy lane if possible or place a piece to create a bumpy lane in the lowest row level possible [8]. This logic hints towards the undesirability of a bumpy board state, which is reflected in our heuristic that seeks to minimize the difference in height between consecutive columns, i.e. reduce bumpiness. In *How to Lose at Tetris*, Heidi Burgiel builds upon this final proof to show a more precise computer strategy for forcing a loss by the player. Burgiel proves that a Tetris game consisting of alternating Z and S pieces will always end within playing 70,000 tetrominos [9]. This creates a specific, defined game without randomness that the player is bound to win. While some games of Tetris are provably winnable, Burgiel then shows that almost all Tetris games will eventually end. An infinite sequence will somewhere contain any given infinite sequence, such as the alternating Z and S sequence described above [9]. Therefore, since Tetris cannot be understood as a winnable game under normal playing conditions, it is concluded that Tetris play can be optimized but not perfected.

The complexity for the offline version of Tetris was proven to be NP-complete as far as we could find, first by Domaine, Hohenberger and Liben-Nowell in their paper *Tetris is Hard, Even to Approximate.* The authors were able to prove that maximizing the number of rows cleared for a given piece sequence, the number of pieces placed on the board in the space of a given game, and the number of times a piece placement cleared four rows simultaneously (a situation the authors called a Tetris) as well as minimizing the maximum height of any filled square on the board was NP-complete. This work used the offline version of the game in which the player was made aware of the piece sequence that would be presented before the game started [2]. Furthermore, a reduced piece set of five rather than 7 pieces and a board with a complicated initial fill state [2]. The authors stated that their decision to study the offline game, which caused the task environment to be fully observable, implied that the difficulty of the online version, where the task environment was partially observable, would be at least as difficult which in turn implies that the version played in this project is also NP-complete. This logical step was directly derived from the very technique used to prove the objectives in Domaine, Hohenberger and Liben-Nowell's work. They first proved that maximizing lines cleared was NP-hard through a reduction of the 3-partition problem [2]. The remaining statements were proven through an extension of their reduction of the proof for the first statement. A reduction, according to Mahesh Viswanathan in his paper *Reductions*, is a technique of relating one problem to another, in such a way as to be able to use the solution for one to solve the other. [10]. He goes on to say that if a problem, call it A, reduces to another problem, call it B, then A is computationally at most as difficult as B; conversely, B is at the minimum as difficult as A. This concept can be used to relate the form of Tetris used in the proof for NP-completeness. We can say that "offline Tetris" is a reduction of "online Tetris", thus we conclude as an extension of the proof discussed here, that online Tetris is both at most and at least of the same hardness as offline Tetris, and we can conclude that it is at least NP-hard. This same logical leap was made also by Amine Boumaza in his paper *How to design good Tetris players*. Boumaza intuites that since offline Tetris is less complicated than online Tetris, it follows that online Tetris is at least as complex as offline Tetris [11].

A derivative work was created by Breukelaar, HoogeBoom, and Kosters which proved the same NP-completeness of Tetris in a more succinct way by using a smaller sequence of tetrominos, a smaller board size, and creating an easier reduction [12]. The basic technique used to prove the complexity was the same, using a reduction of the 3-partition problem, but they were able to greatly reduce the number of possible piece placements by considering buckets of two columns rather than each column individually [12]. It should also be noted that the authors of both papers collaborated on a version of *Tetris is Hard, Even to Approximate*, with no new conclusions [13].

B. Genetic Algorithms

Other researchers have seen the potential to solve games like Tetris with algorithms influenced by nature like genetic algorithms and neural networks. Flom and Robinson used a genetic algorithm approach to evaluate a few crossover functions under varying conditions: consistent piece set and limited pieces, random piece set and limited moves, and unlimited piece set [14]. The environment they created for their agents simplified the game space in that the agent did not have to simulate piece rotation; rather, it selected the orientation and column space for piece placement, and the score was strictly a count of lines made (thereby counting lines cleared) without scaling for simultaneous clearing and levels reached. Furthermore, the agent only took into account the current piece in play, disregarding the look-a-head piece. The function used by the Tetris agent to determine which move to make next was a weighted linear sum [14]. The weights associated with this function were what the genetic algorithm was striving to optimize. The heuristics associated with these weights were: pile-height, closed holes, number of lines completed, and bumpiness [14]. The intention was to minimize all heuristics, aside from lines complete, which was maximized [14]. Fitness was measured through two means: lines created before a loss (or if applicable, reaching the piece set limit), and lines per piece played [14]. The second of the two fitness measures was added with the intent to limit the potential effect the move restriction could have on an agent's ability to complete lines [14]. Each set of weights was evolved through one of the aforementioned crossover methods and subjected to a chance of mutation. The result was then analyzed through the act of an agent playing Tetris and evaluating moves based on the produced weights [14]. Out of the four crossover methods analyzed, Flom and Robinson found that uniform and HUX performed best of the four and one-point performed the worst [14]. They reasoned that one-point failed due to loss of genetic diversity, while uniform and HUX were less likely to create copies of members of the previous generation, thereby maintaining their diversity [14]. For each case evaluated, it was found that the agents successfully evolved the desired characteristics, but manually entering weights which corresponded to a strong agent previously evolved did not tend to result in a strong agent, which was surprising [14].

A similar experiment was performed by Elad Shahar and Ross West. They also evolved a weighted sum through means of a genetic algorithm to create a Tetris-playing agent [15]. The following heuristics were associated with the weights: filled spot count, weighted filled spot count, maximum altitude, hole count, lines cleared, altitude delta, deepest hole, sum of all holes, horizontal roughness, vertical roughness, well count, weighted holes, highest holes, and game status [15]. The fitness of a given weighted sum was evaluated through the score produced by an agent playing tetris where best moves were chosen using the produced weighted sum [15]. The crossover method used switched certain values between parent solutions, and child solutions were occasionally mutated by altering one aspect of the produced solution [15]. Three evaluation environments were used to limit the number of lines cleared to 100, 1000, and unlimited [15]. They found that the 100 and 1000-line environments produced heuristics which caused the agents to tend towards plays that cleared more lines simultaneously, thus increasing their score in a shorter time frame [15]. The unlimited environment produced heuristics which allowed games to last for an incredible amount of time, Shahar and West report a population of 50 agents taking 175 hours to complete [15].

The most influential experiment performed in the field of creating Tetris playing agents was the work done by Colin Fahey, who used two computers one which played Tetris while the other displayed the game [16]. The Tetris-playing agent used a camera to view the screen and 8 relays controlled by RS-232 text command to electrically "press keys" on the Tetris computer's keyboard. Thus, the setup mimicked, almost precisely, a human's interaction with the game. Fahey was able to program a very efficient Tetris-playing Robot [16].

As far as we are aware, there has not been any research that utilizes wisdom of artificial crowds (WoAC) to aid in the creation of an efficient Tetris-playing agent. However, WoAC has been used successfully in developing efficient solutions to other NP-complete games such as Sudoku [17] and Light Up [18].

C. Other Approaches: Creating Tetris Agents

Other approaches to creating Tetris-playing agents have been created as well. Matt Stevens and Sabeek Pradhan used deep reinforcement learning to create a Tetris-playing agent. They measured the performance of an agent based on game length (number of placements made) and game score (number of lines cleared) [19]. This approach had to overcome the obstacle of evolving an agent that could score high enough to register a score of any significance [19]. To account for this, they used moves played as a measure of quality, as it gave an intermediate value that was more attainable than lines cleared, thus allowing the agent to have enough time to learn and improve [19]. They explored the effect time between action and reward had on the agent's performance, thus comparing an agent trained to react to agents trained to predict varying moves ahead [19]. Unsurprisingly they found that learning converged significantly more quickly for the instant reward agents than those whose reward was further removed [19].

Another approach to solving Tetris was examined by Donald Car in his paper *Applying Reinforcement Learning to Tetris*, where he reviewed work others have done in this field. Reinforcement learning has some interesting implications in the arena of training a Tetris-playing agent. Car notes that reinforcement learning has the potential to influence an individual agent at the decision-making level, rather than at the generational level as genetic algorithms do [20]. Unfortunately, the implementation of reinforcement learning on full Tetris suffered from poor performance, at one point unlearning its policy unexpectedly [20]. It was reasoned that the stochasticity of Tetris posed a challenge to the style of reinforcement used in this experiment, one which was not overcome [20].

Finally, the Harmony Search algorithm, a Meta-heuristic algorithm based on the improvisation process of musicians, was used by Romero, Tomes, and Yusiong to create a Tetris-playing agent [21]. The agent created in this experiment was evaluated based on rows cleared. Their research showed that the Harmony Search algorithm produced Tetris intelligent agents which were quite efficient, and the method was relatively straightforward to implement [21].

III. Proposed Approach

Our approach is based upon Tetromino, a basic Tetris clone in Python built with pygame. The original program was initially altered to use the actual scoring grid used by Tetris rather than merely counting lines cleared, as Tetromino does, as well as removing hard drops. Otherwise, the Tetromino program was able to be used for our simulations. This program utilizes a column-major array of character arrays to hold the current board state and dictionaries with the following fields: shape, rotation, x, y, and color for the pieces. The shape field is a character in the set {‘S’, ‘Z’, ‘J’, ‘L’, ‘I’, ‘O’, ‘T’} corresponding to the 7 piece types used in Tetris. The rotation and color values are integers referring to array indices, and the x and y fields indicate the location of the 5x5 template used to store pieces and their rotations.

A. Heuristic Function

Our simulation utilized a genetic algorithm with wisdom of artificial crowds supplement to generate a normalized set of heuristic function weights. Four board heuristics were analyzed by the simulation. The first three of these are negative influences, indicating a worse board state for higher values. As a result, these are expected to have negative weights. Furthermore, these all utilize a getColumnHeights function to determine the heights of the board columns. This function iterates through the columns, finding the highest filled square and appending the board height minus that value into the return array. The first was aggregate height, consisting of the sum of the heights of all columns on the board. This was determined by calling getColumnHeights and taking the sum of the returned values. The second was holes, indicating the number of empty squares on the board below the highest filled square in each column. This was determined by iterating through the subarray for each column of the board under the corresponding height value from getColumnHeights, returning the count of empty squares across the iteration. The third was bumpiness, which shows the aggregate change in height between adjacent columns. This was determined by calling getColumnHeights and taking the sum of the absolute values of the differences between adjacent members of the returned array. The final heuristic was lines complete, corresponding to the number of completely filled rows on the board. This heuristic is a positive influence, resulting in an ideally positive weight. Since the heuristic function was used after piece placement but before the piece was integrated into the board, so the full rows are not yet removed from the board. This was determined from a function defined by the base Tetrominos program called removeCompletedLines, which also returns the number of lines completed. This function iterates across the rows of the board, calling a function to determine if the line is complete and, if so, updates the user interface to remove the line. The called function iterates across the columns and determines if any member of the board in the passed row is blank, returning true if all are not. The set of these weights and the set of actual heuristic data for a given board state were regarded by the program as four-dimension vectors. To determine the heuristic function value for a given board state, the dot product of the weights and the data values was taken, resulting in a scalar fitness value.

B. Running the Simulation

The genetic algorithm begins its simulation by generating the initial population of genomes. Many genomes are created, consisting of the 4 heuristic weights. It is these weights that will be compared and optimized by the genetic algorithm. For each genome, random values from a uniform distribution of values between -1 and 1 are added for each of the 4 heuristic weights. The genomes also hold a field for the score corresponding to the simulation for that set of values.

The simulation then runs a runSimulation function for each genome to simulate its set of heuristic weights. The runSimulation function simply calls the runGame function a set number of times, returning the aggregate sum of the returned values from these calls. The runGame function will run a 500 piece-limited simulation of a single game of Tetris. This function begins by generating a blank board using a second function from the base Tetrominos implementation, generating the first two pieces, and declaring many other variables. The function then enters the turn-wise loop, which was implemented in two ways: single piece and look-ahead piece. This loop begins by generating a new look-ahead piece and making deep copies of the current board state and current piece. The function then calls a function to determine the best placement of the given piece onto the board using the currently-tested heuristic genome. In the case of the single piece version, only the current piece is evaluated, while for the look-ahead piece logic, the current piece in combination with the next piece given were evaluated together. This function returns a dictionary containing the piece state and heuristic score of the best placement. This piece is put into position and, once settled, added to the original uncopied board. The completed lines are removed, and the score is incremented as determined by the Tetris rules. The level is incremented, if necessary, and the current piece is nulled out. However, if the piece is forced to be placed above the board proper, the game is over, and the loop will break. Otherwise, the game will continue through the next turn, until the piece limit is reached.

The function called to determine piece placement begins by making another deep copy of the passed board and piece. The function continues into the 1st loop, iterating over the rotations of the passed piece. This loop will run 1, 2, or 4 times. The piece is rotated, and then the 2nd loop is entered, iterating over the possible horizontal positions on the board for that piece rotation. For each valid position and rotation on the next piece, the piece is placed at the top of the board in the corresponding state and then dropped into place sitting on another piece, removing lines if necessary. The process of these two loops to place a piece is repeated, nesting further, to handle rotations and translations of the look-ahead piece. Once the 4th loop is entered, and the piece placed, all the heuristic values for the board state are calculated, multiplied by the genome's weights, and added into a score variable. If the game is forced to be over based on that piece placement, the score is made extremely negative. If the current score is better than the current best, its score is replaced with the current one and the corresponding best piece placement is replaced with the current as well. Once all possible pairs of placements are iterated though, the best score and piece placement are returned in a dictionary to the calling function.

C. *Combining Genomes*

Once all genomes run their simulations, consisting of several games, the population of genomes is modified by a function to create the next generation of heuristic weights. This function begins by selecting the best 10 percent of genomes by score and adding them to the next generation genome list as the elite members of the previous generation. Then the function iterates through a reproduction routine enough times to fill the population to the same size as before. This routine selects 2 parents by a tournament selection method. This method randomly selects 10 percent of the population and returns the genomes with the best 2 scores from that population. These two are then sent to a crossover function, which calculates the average, weighted by score, of each of the two parents' heuristic weights. These values are combined into a dictionary, normalized as a four-dimension vector, and returned. The next generation method then calls a mutation function, mutating on a small rate. Three mutation functions were experimented with. The first mutation selects one of the four heuristic weights at random and either increments or decrements by 0.2, selected randomly. The resulting heuristic vector is again normalized and returned. The new genome is then appended onto the next generation, and at the end of iterating the function returns this new generation.

A variation on this mutation function was also used in experimentation. In this variation, rather than selecting a value in a range which included both positive and negative values, only positive or only negative values were made available for the mutation based on the sign of the current heuristic constant selected. This was an attempt to capitalize on our observation that some heuristics should be negative values while others should be positive values. In this mutation, the randomly selected constant to undergo mutation would be made more negative if it were originally negative or more positive if it were originally positive, thus magnifying the behavior it was already displaying and thereby speeding the removal of weak agents and potentially pushing strong agents out of a local minimum.

The final mutation function was created to take the best heuristics found in the randomly produced first generation and use those values as the pool of values available for the mutation. It was observed that most randomly generated initial populations were able to produce a few decent Tetris playing agent. Since these heuristics were removed from the evolution process, it was hypothesized that these values would provide the desired genetic diversity with a greater chance of steering the agent towards stronger traits since these heuristics had produced strong results in an agent already.

At the end of processing generations, the program concludes by running a wisdom of artificial crowds function.

After each generation's simulation, the genome with the best score is saved in a separate list. At the end of the simulation iterations, this list will contain one genome per generation. The wisdom of artificial crowds algorithm is applied to this list. An aggregate sum for each heuristic weight across this list is taken. The average value of each weight is calculated, and the set is normalized as a four-dimension vector.

IV. Experimental Results

The quality of a Tetris-playing agent produced by our algorithms was based on that agent's fitness, which was measured by the agent's score in Tetris games played. The score, as mentioned in the introduction, was a measure of total lines cleared where the value of a cleared line was increased in relation to the game level the agent had achieved, and the number of lines cleared with a single piece. This measure was also indirectly related to game length, as higher levels and greater numbers of lines cleared are directly related to the length of game play. While these results are not directly comparable to the other methods used to create Tetris-playing agents, we can state that the agent produced by our genetic algorithm was comparable to the Harmony search algorithm, in that it was able to create an agent that played the game with some level of efficiency and the implementation of the algorithm was relatively straight forward. Thus, it outperformed the reinforcement learning approach in its efficiency and was a much simpler implementation than the deep learning approach.

A. *Data*

The data used for this project was randomly generated using a python module. Each individual in the population was assigned a random floating-point value between 1 and negative one inclusive. Each individual in the population gets four of these random values, one for each constant of the vector.

The Tetris simulator used in this project was based on the Tetromino project written by Al Sweigart.

B. *Result*

Experiments were performed using combinations of different population sizes, mutation percentages, and percent elitism. The population sizes were 20, 50, or 100, with mutation rates of either one or two percent and elitism of either 15 or 20 percent. Each experimental configuration was run for 25 generations to evolve, except for configurations using look-ahead logic, which were given 14 generations due to the significantly slower run time when using this algorithm.

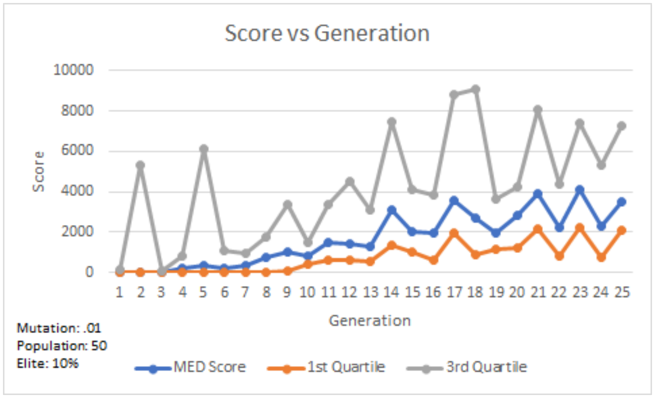


Fig. 1 Experimental results showing 1st quartile, 3rd quartile and average scores for trial run over 25 generations using mutation rate of 1%, population size of 50, and 10% elitism.

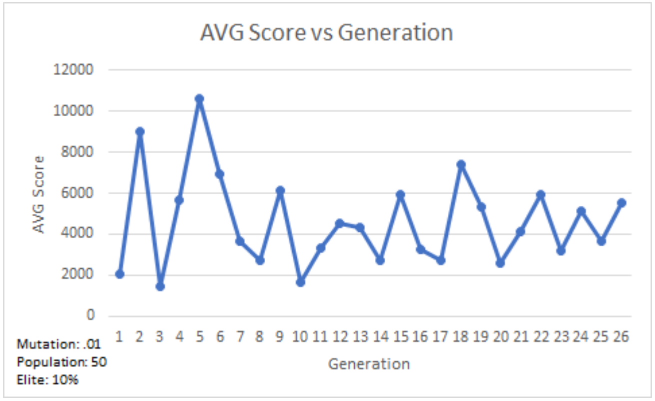


Fig. 2 Experimental results showing average scores for trial run over 25 generations using mutation rate of 1%, population size of 50, and 10% elitism.

As can clearly be seen in most of the graphs there was variability in agent performance under all experimental combinations. This could be a reflection of the fact that while Tetris can be played optimally, it cannot be played perfectly. It is possible for a strong Tetris playing agent to receive an impossible or difficult piece sequence. This variability in difficulty of play sequences could explain the spiking and dropping of high scores, which is present in most of the evolution graphs.

Overall increasing the size of the population did not seem to have a positive effect on the performance of the Tetris agent from a fitness standpoint, higher scores were recorded for smaller populations sizes. This is likely due to the decreased genetic diversity associated with smaller populations especially in reference to elitism, a smaller population size has fewer individuals in general and fewer individuals are selected based on an elitism percentage. This means that if the population generates a strong individual this same individual will continue to move forward without alteration and provide high scores.

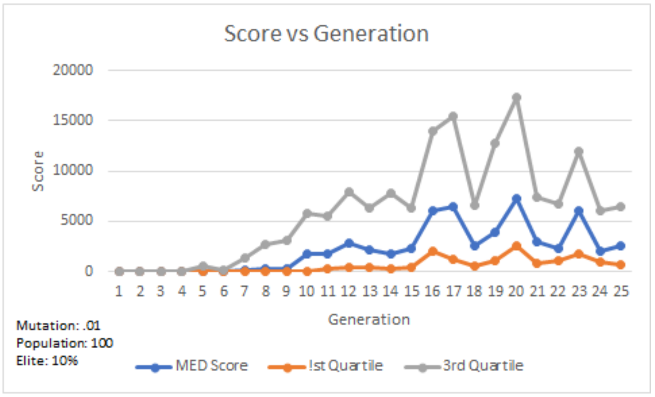


Fig. 3 Experimental results showing 1st quartile, 3rd quartile and average scores for trial run over 25 generations using mutation rate of 1%, population size of 100, and 10% elitism.

From the standpoint of steady evolution, the best results were obtained from a population of 20, with a mutation rate of 2 percent and elitism of 15 percent. The graph of average scores for this combination shows the best example of an exponential curve and steady improvement over time. Generally speaking, most of the other combinations experimented with tended to improve over time, but other combinations provided a less obvious curve and resulted in more of a general trend. This could again be the product of difficult versus easier piece sequences.

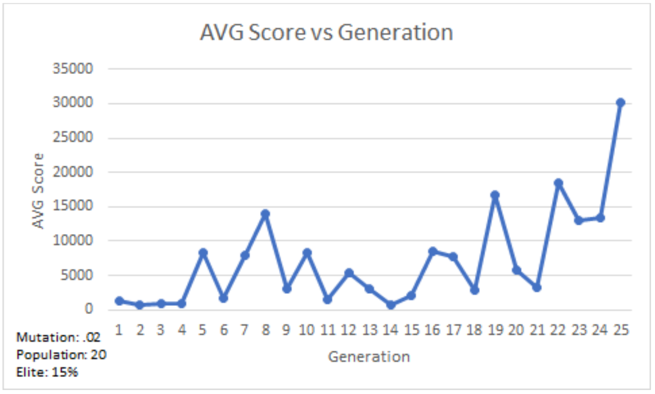


Fig. 4 Experimental results showing average scores for trial run over 25 generations using mutation rate of 2%, population size of 20, and 15% elitism.

The signed augmenting mutation function provided some promising results, trials were run on a population size of 20 with elitism percent of 20 or 15, and a mutation rate of 1 or 2 percent. Each trial using this modified mutation produced subsequent generations that trended stronger overall, but still produced random spiking. The trial using a 1 percent mutation rate and 20 percent elitism produced stronger agents (refer to fig. 6) but with a more stagnant evolution pattern, while the trial using a 2 percent mutation rate and 15 percent elitism produced weaker agents it showed a distinct trend of improvement (refer to fig. 5).

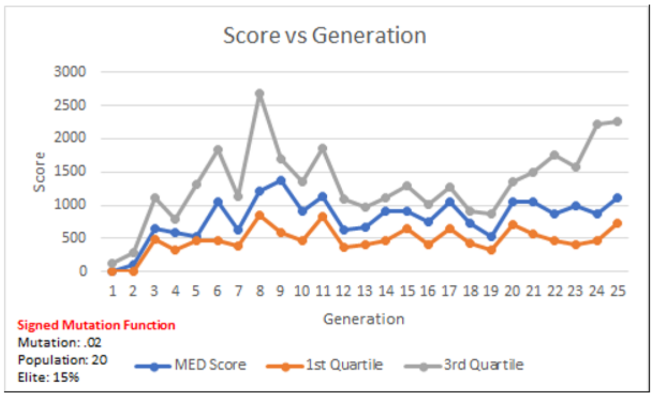


Fig. 5 Experimental results showing 1st quartile, 3rd quartile and average scores for trial run over 25 generations using mutation rate of 2%, population size of 20, and 15% elitism. The mutation function used was the augmented sign mutation.

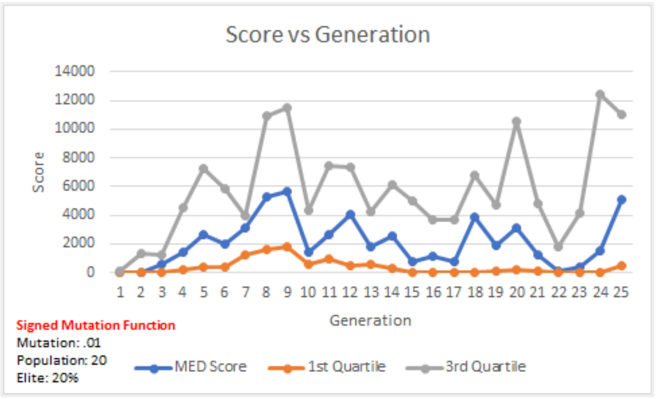


Fig. 6 Experimental results showing 1st quartile, 3rd quartile and average scores for trial run over 25 generations using mutation rate of 1%, population size of 20, and 20% elitism. The mutation function used was the augmented sign mutation.

The swap mutation was experimented on with a 1 percent mutation rate, population of 20 and elitism of 20 percent. The results for this set of criteria were interesting in that the first generations were abysmal; however, between generations 9 and 11, the heuristic scores began to climb steadily, one generation after the next and in the end a decent agent was produced (refer to fig. 7). This phenomenon of producing a stronger final individual when early generations are quite poor has been observed through other experimentation with genetic algorithms and would be an interesting topic for future research.

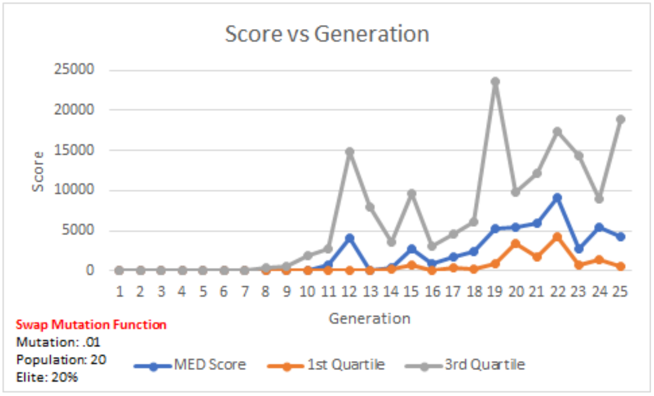


Fig. 7 Experimental results showing 1st quartile, 3rd quartile and average scores for trial run over 25 generations using mutation rate of 1%, population size of 20, and 20% elitism. The mutation function used was the early heuristic swap mutation.

Figure 8 shows a breakdown of the fitness score and heuristic constants for the strongest agents produced in the last generation of the genetic algorithm for all experimental combinations without look-ahead logic. While it was the case that agents produced at different times during the evolution process did achieve higher fitness values than the ones showcased here, these values were chosen to show the result of the evolutionary algorithm rather than chance. As can be seen, the strongest agent was produced by trials run with population of 20, 2 percent mutation rate, and elitism at 15 percent. It is interesting to note that this is the only positive lines cleared heuristic, which we hypothesized needed to be positive to produce a strong agent. Across these strong agents it is notable that the aggregate height heuristic was strongly negative, and bumpiness and holes were moderately negative. It was unexpected to see strong agents produced with negative lines cleared values as minimizing lines cleared would actively impede the score of the Tetris game. These results would imply that minimizing aggregate height of a Tetris game has a far greater impact on a Tetris players performance than working to maximize lines cleared.

TABLE I
BEST PERFORMING AGENT HEURISTICS USING SINGLE-PIECE ALGORITHM
.

Parameters	Bumpiness	Holes	Aggregate Height	Lines Cleared	Score
Swap Mutate M: 1 % E: 15 % Pop: 20	-0.341878	-0.32742803	-0.81250625	-0.34021157	73800
M: 1 % E: 20 % Pop: 20	-0.40654381	-0.40281927	-0.71291201	-0.40523478	26260
M: 2 % E: 15 % Pop: 20	-0.3211779	-0.26246908	-0.86267829	0.289380202	103220
M: 1 % E: 10 % Pop: 100	-0.36705203	-0.34971617	-0.77999803	-0.36684395	69240

Figure 9 shows the strongest results produced in the last generation of the genetic algorithm with look-ahead logic implemented. The results clearly indicated that the additional insight provided by the look-ahead piece positively impacted the overall average fitness of the Tetris agents produced. This can also be seen in Figures 10 and 11.

TABLE II
BEST PERFORMING AGENT HEURISTICS USING LOOK-AHEAD ALGORITHM.

Parameters	Bumpiness	Holes	Aggregate Height	Lines Cleared	Score
M: 2 % E: 15 % Pop: 20	-0.27897336	-0.50718315	-0.77487336	-0.25398897	103400
M: 2 % E: 20 % Pop: 20	-0.22159162	-0.31094434	-0.63659727	-0.67004081	109540
M: 5 % E: 15 % Pop:20	-0.18963445	-0.31094434	-0.85399079	-0.40065865	107660

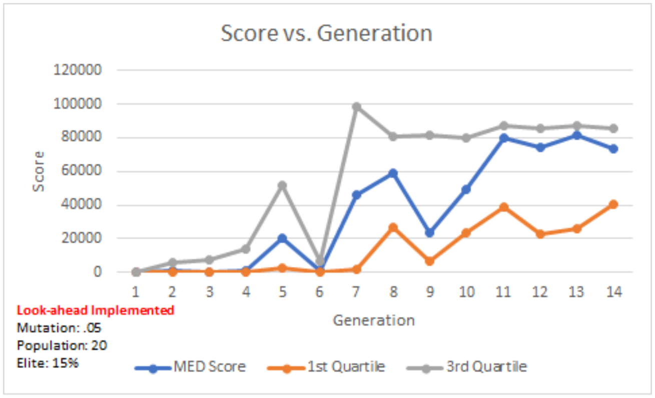


Fig. 10 Experimental results showing 1st quartile, 3rd quartile and average scores for trial run over 14 generations using mutation rate of 5%, population size of 20, and 15% elitism.

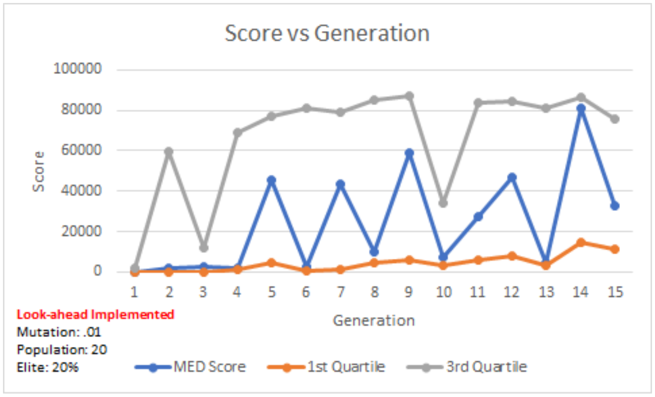
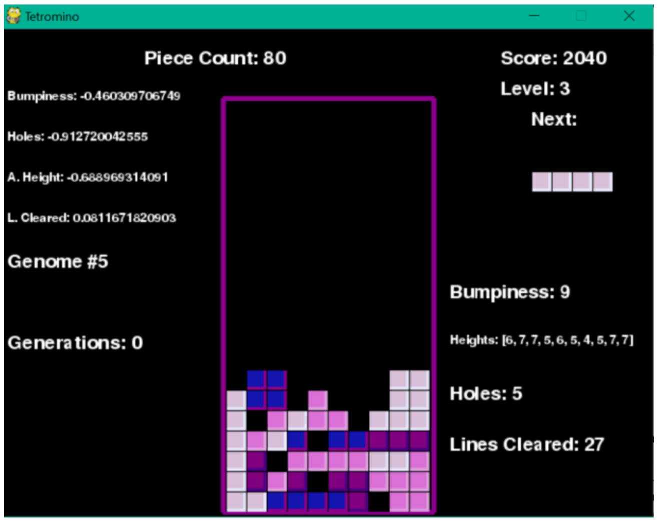


Fig. 11 Experimental results showing 1st quartile, 3rd quartile and average scores for trial run over 15 generations using mutation rate of 1%, population size of 20, and 20% elitism.

The addition of wisdom of artificial crowds was used to augment the single piece version of the genetic algorithm. It can be seen from the data shown in Fig. 12 that the additional application of the wisdom of artificial crowds algorithm resulted in agents that performed better than the average agent that utilized the genetic algorithm approach alone. While there are cases in which the genetic algorithm alone produced higher-scoring agents, the implementation of crowds allowed for a more stable growth within the generations. This produced final heuristics that consistently led to a strong simulated game, with final game scores frequently surpassing 100,000. Further experimentation can be applied to the wisdom of artificial crowds approach, in which selection of highest fitness heuristics per generation can be refined in order to generate an even stronger performing agent.

TABLE III
RESULTS FOR ALGORITHM RUN WITH WISDOM OF ARTIFICIAL CROWDS AUGMENTATION.

GEN #	MED Score	1st Quartile	3rd Quartile	MIN Score	MAX Score	AVG Score
1	0	0	120	0	20320	851.2
2	0	0	5335	0	122920	10362.4
3	0	0	70	0	6300	395.2
4	180	0	835	0	93480	3940.8
5	370	40	6110	0	138660	10764.8
6	180	0	1065	0	132260	7030.4
7	320	0	930	0	126520	6006
8	750	0	1755	0	63320	2826.8
9	1050	55	3370	0	138740	5224.4
10	790	405	1505	0	48440	2582.4
11	1510	615	3380	0	26660	3327.2
12	1450	605	4510	0	20560	3381.2
13	1260	545	3130	40	14260	2708.4
14	3120	1370	7455	80	33800	5848.4
15	2000	990	4080	0	16160	3422.8
16	1990	590	3810	0	17520	2690
17	3550	1970	8785	120	54180	7226.4
18	2710	890	9090	80	23700	5520.4
19	1990	1120	3605	0	10000	2631.6
20	2830	1240	4265	40	32320	3960
21	3880	2155	8050	0	26860	5856.8
22	2210	810	4380	0	18580	3382.8
23	4550	2290	9405	180	37160	6777.6
24	4110	2205	7430	180	21060	5146.8
25	2280	760	5310	0	19960	3643.2
26	3510	2120	7280	160	22940	5404.4



```
Piece Count: 80
Bumpiness: -0.460309706749
Holes: -0.912720042555
A. Height: -0.688969314091
L. Cleared: 0.0811671820903
Genome #5
Generations: 0
Score: 2040
Level: 3
Bumpiness: 9
Heights: [6, 7, 7, 5, 6, 5, 4, 5, 7, 7]
Holes: 5
Lines Cleared: 27
```

Fig. 13 Sample of the user interface simulation.

V. Conclusion

Overall the genetic algorithm was successful in evolving a Tetris-playing agent. The combination of the crossover and mutation routines were found to evolve heuristics used for an efficient Tetris-playing agent, which in general improved overtime. This was due in large part to the crossover method which took the weighted average for each constant of two of the fittest individuals from the current generation. The weight for each parent agent's contribution was based on that individual's score, thus the fitter, or more efficient, of the two parents had a greater influence on the produced individual.

The addition of the wisdom of artificial crowds was found to improve on the average solution produced by any of the genetic algorithms solutions used to produce it. This was a somewhat surprising result and we consider it a success.

Considering the look-ahead piece in the variation on the heuristic analysis produced markedly better scores when compared to a single piece analysis. This makes sense from a logical standpoint, as having greater insight into future board states should result in a stronger performance overall.

Furthermore, experimentation was conducted on some of the variables, mutation rate, population size, and elite selection rate, controlling the running of the genetic algorithm to find more effective solutions. This was a success, showing these factors play a major role in the generation of heuristics values.

The genetic algorithm we produced for creating a Tetris-playing agent relied on four heuristics to produce an agent, an improvement on this approach could potentially be to simply include more heuristics, this concept will be discussed further in the future research plans below.

Our approach to wisdom of artificial crowds could be improved upon to better account for consensus between experts. The nature of how individuals are created presents a challenge in that perfect consensus between individual's constant values is unlikely. Our approach of taking a weighted average of experts across the fittest solutions produced by multiple generations was an attempt to overcome this difficulty. However, it may be more effective to create a range in which individuals are deemed to "agree" on a value for a given constant. In this way, it would be possible to "poll" experts from the genetic algorithm and create an agent with the attributes agreed upon by these experts. Of course, a weighted average would still need to be taken for the range in which individuals agreed, but we would be assured that the values given were similar and would not run the risk of a strong outlier skewing the results, while unlikely this is a possibility considering our current approach.

A. *Future Research Plans*

Expanding on the work done in this experiment it would be of interest to explore variations on the current algorithm. Specifically, how would the agent be affected by the removal of time constraints on a given game? Brzustowski defined a scenario where Tetris could be beat if the game went on ad infinitum, expanding on this, how long could a well-formed agent play [8]? In this scenario we propose the fitness measure would also include length of game play, or potentially only reflect length of game play. Another expansion of interest would be to add the heuristics described by Domaine, Hohenberger and Liben-Nowell, namely to maximize the number of pieces placed on the board and to minimize the height of the tallest piece on the board. Would additional heuristics create a stronger agent? Is there a point where additional heuristics are no longer helpful?

Finally, it would be beneficial to continue to evolve an approach to the creation of a wisdom of artificial crowds algorithm for creating a Tetris-playing agent.

VI. Acknowledgements

This project was influenced by work done by L. Yiyuan in his post "Tetris AI - the (near) perfect bot" [1], and the work done by Al Swigart in his Invent with Python series [22].

VII. References

[1] Y. Lee. "Tetris AI - The (Near) Perfect Bot" Internet:https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/, April 14, 2013 [Nov. 12, 2018]

[2] E.D. Demaine, S. Hohenberger and D. Liben-Nowell. "Tetris is Hard, Even to Approximate." Lecture Notes in Computer Science, 2003, pp. 351-363

[3] E. Weisstein. "NP-Problem." http://mathworld.wolfram.com, [Online]. Available: http://mathworld.wolfram.com/NP-Problem.html. [Nov. 12, 2018]

[4] S. Russell, P. Norvig. Artificial Intelligence: A Modern Approach NJ: Upper Saddle River, 1995

[5] "History of Tetris." Internet: https://tetris.com/history-of-tetris, [Nov. 12, 2018]

[6] "Tetris (NES, Nintendo)." Internet: http://tetris.wikia.com/wiki/Tetris_(NES,_Nintendo). [Nov. 12, 2018]

[7] "Tetris (NES, Nintendo)." Internet: http://tetris.wikia.com/wiki/Scoring. [Nov. 12, 2018]

[8] J. Brzustowski. Can you win at Tetris? Master's thesis, U. British Columbia, 1992.

[9] H. Burgiel (1997). "How to Lose at Tetris." The Mathematical Gazette [Online] 81(491): 194-200. Available: https://www.jstor.org/stable/pdf/3619195.pdf?refreqid=excelsior%3A0a1b079adbb3a1fd554dc8a7a8be54be. [Nov. 12, 2018]

[10] M. Viswanathan. "Reductions." https://courses.engr.illinois.edu, pata. April 14, 2016 [Online]. Available: https://courses.engr.illinois.edu/cs374/sp2018/B/notes/reductions.pdf. [Nov. 12, 2018]

[11]A. Boumaza, (2013). How to design good Tetris players. Available: https://hal.inria.fr/hal-00926213/document [Nov. 26, 2018]

[12] R. Breukelaar, H.J. HoogeBoom, and W.A. Kosters. (2003) "Tetris is hard, made easy." Technical Report, Leiden Institute of Advanced Computer Science, Universiteit Leiden.

[13] R. Breukelaar, E.D. Demaine, S. Hohenberger, H.J. Hoogeboom, W.A. Kosters, and D. Liben-Nowell. (2004, April) "Tetris is Hard, Even to Approximate." International Journal of Computer Science. [Online]. 14 pp. 41-68 Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.5860&rep=rep1&type=pdf. [Nov. 11, 2018]

[14] L. Flom & C. Robinson. (2004). Using a genetic algorithm to weight an evaluation function for Tetris. Colorado State University, Tech. Rep. Luke. Available: https://pdfs.semanticscholar.org/2f55/98f4a2e211681835a500c484e262bdbaf50d.pdf [Nov. 13, 2018]

[15] E. Shahar, & R. West. Evolutionary AI for Tetris. Internet: http://www.cs.uml.edu/ecg/pub/uploads/AIfall10/eshahar_rwest_GATetris.pdf (2010) [Nov. 13, 2018]

[16] C. P. Fahey. (2003). Tetris. Colin Fahey.Internet: https://www.colinfahey.com/tetris/, 2007 [Nov. 13, 2018]

[17] J. Redding, J. Schreiver, C. Shrum, A. Lauf, & R. Yampolskiy. (2015, July). Solving NP-hard number matrix games with Wisdom of Artificial Crowds. In Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES), 2015 (pp. 38-43). IEEE. Available: https://ieeexplore.ieee.org/document/7272959 [Nov. 13, 2018]

[18] L. H. Ashby & R. V. Yampolskiy. (2011, July). Genetic algorithm and Wisdom of Artificial Crowds algorithm applied to Light up. In Computer Games (CGAMES), 2011 16th International Conference on (pp. 27-32). IEEE. Available: https://ieeexplore.ieee.org/document/6000341 [Nov. 13, 2018]

[19] M. Stevens, S. Pradhan. "Playing Tetris with Deep Reinforcement Learning." http://cs231n.stanford.edu, para. 2016 [Online]. Available: http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf. [Nov. 12, 2018]

[20] D. Carr. "Applying reinforcement learning to Tetris." https://www.colinfahey.com, para. May 30, 2005 [Online]. Available: https://www.colinfahey.com/tetris/ApplyingReinforcementLearningToTetris_DonaldCarr_RU_AC_ZA.pdf. [Nov. 12, 2018]

[21] V. Romero, L. Tomes, and J.P. Yusiong. (2011, Jan.) "Tetris Agent Optimization Using Harmony Search Algorithm." International Journal of Computer Science. [Online]. 8(1), pp. 22-31. Available: https://pdfs.semanticscholar.org/e6b0/a3513e8ad6e08e9000ca2327537ac44c1c5c.pdf [Nov. 11, 2018]

[22] A. Sweigart. "Tetromino (a Tetris clone)" [Nov. 12, 2018]