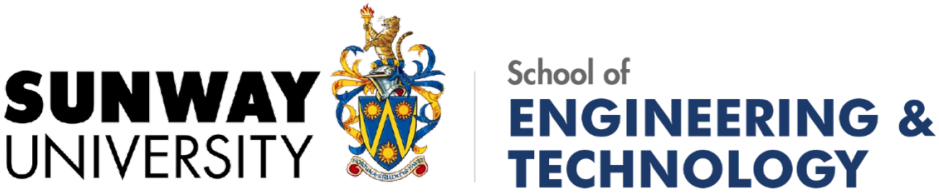


CAPSTONE PROJECT 2
Final Report

Playing Versus Tetris with a Genetic Algorithm

by

Yap Wei Xiang
21067939

Bachelor of Science (Honours) in Computer Science

Supervisor: Dr Richard Wong Teck Ken

Semester: September 2024
Date: 13 December 2024

Department of Smart Computing and Cyber Resilience (DSCCR)
School of Engineering and Technology
Sunway University

Abstract

This capstone project explores the use of genetic algorithms (GAs) to optimize gameplay strategies in Versus Tetris, a competitive multiplayer variant of the classic Tetris game. Due to the NP-complexity of Tetris, traditional algorithms struggle to find efficient solutions, making nature-inspired approaches like GAs a promising alternative. The methodology involved developing a simulation environment where a GA was used to evolve a set of weights for an evaluation function that guided the bot's decision-making process. The algorithm incorporated population initialization, fitness function calculation, and genetic operations (selection, crossover, mutation) to iteratively improve gameplay performance.

While the GA did not result in significant improvements in attack efficiency, with performance declining in this area, it showed notable gains in survivability. The bot was able to place more pieces before the game ended, indicating that the GA successfully optimized strategies for longer gameplay under certain conditions. The evaluation metrics focused on attack efficiency and survivability, with the results reflecting a clear trade-off: while attack efficiency worsened, the bot's ability to survive longer in the game improved.

These results suggest that the GA was able to enhance defensive strategies but struggled to optimize offensive tactics effectively. Future work could refine the fitness function to better balance attack and defense, incorporate additional gameplay features like the hold mechanic and lookahead, and explore hybrid models that combine genetic algorithms with reinforcement learning techniques. This research provides valuable insights into the challenges and potential of using nature-inspired algorithms to tackle complex, real-time decision-making in competitive gaming.

Contents

Abstract i

List of Tables iv

List of Figures v

1 Introduction 1

1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Aim . . . 3
1.4 Objectives . . . 3
1.5 Project Scope . . . 3

2 Literature Review 4

2.1 The Difficulty of Tetris . . . 4
2.1.1 Complexity Classes . . . 5
2.1.2 Justifying Non-traditional Algorithmic Approaches . . . 5
2.2 Approaches to Tetris . . . 6
2.2.1 Learning by Imitation . . . 7
2.2.2 Upper Confidence Bounds for Trees . . . 8
2.2.3 Deep Reinforcement Learning . . . 9
2.2.4 Meta-heuristic Algorithms . . . 11
2.3 Playing Tetris with Nature-inspired Algorithms . . . 14
2.3.1 Genetic Algorithm . . . 14
2.3.2 Ant Colony Optimisation . . . 15
2.3.3 Particle Swarm Optimisation . . . 16

3 Methodology 18

3.1 Defining the Rules . . . 18
3.1.1 Tetromino Sequence Generation . . . 18
3.1.2 Matrix Dimensions . . . 19
3.1.3 Spins . . . 19
3.1.4 Attacks . . . 20
3.1.5 Gravity . . . 25
3.1.6 Other Features . . . 26
3.2 Building the Simulation . . . 26
3.2.1 OpenTris . . . 26

3.2.2 Bot Behaviour 28
3.3 Genetic Algorithm Implementation 30
3.3.1 Representation 30
3.3.2 Population Initialisation 31
3.3.3 Fitness Function 31
3.3.4 Selection 31
3.3.5 Crossover 32
3.3.6 Mutation 32
3.3.7 Termination Criteria 32
3.4 Evaluating Gameplay 32
3.4.1 Evaluating Attack Efficiency 32
3.4.2 Evaluating Survivability 32

4 Results and Discussion **34**
4.1 Validation of Objectives 34
4.2 Genetic Algorithm Results 34
4.3 Evaluating Final Weight Vector 36

5 Conclusion **38**
5.1 Limitations 38
5.2 Future Work 38

6 References **39**

List of Tables

2.1 Probability of each piece in Chen's "hard" mode. 11
2.2 Score table for lines cleared in Lewis' Tetris game [11]. 15

3.1 I Tetromino Wall Kick Data. 20
3.2 T, Z, S, L and J Tetromino Wall Kick Data. 20
3.3 Attack Table for Line Clears 21
3.4 Attack Table for T-spins 24
3.5 Combo Table 24

4.1 Weights of best fit individuals in iterations 1, 20, 50, and 75. 35
4.2 Average attack efficiency of best fit individuals in iterations 1, 20, 50, and 75. 36
4.3 Average pieces placed by best fit individuals in iterations 1, 20, 50, and 75 when playing survival. 36

List of Figures

1.1 A typical modern Tetris game where four lines are about to be cleared. The Tetromino on the left of the matrix is the *Hold* piece and the pieces to the right of the matrix are collectively known as the *Queue*. 1
1.2 A typical game of Versus Tetris. Both players are trying to send lines to each other. The grey blocks are *Garbage Lines* sent from Player 2 (right) to Player 1 (left). 2

2.1 Visualisation of the sets P, NP, NP-hard and NP-complete. 6
2.2 Abstraction of Zhang, Cai, and Nebel's System Components 7
2.3 An example of a single node of the matrix state and piece in the planning tree. 9
2.4 Z and S pieces. 11

3.1 The Tetrominos and their names. 18
3.2 I piece is spawned at the 21st row of the matrix, but the player can survive by slotting the piece into the eighth column of the matrix. 19
3.3 The four rotation states of all seven Tetrominos. The black dots represent centres of rotation. 21
3.4 Example of garbage lines (gray) being cleared by an L piece. 22
3.5 T-piece in a 3×3 box. 22
3.6 Visualisation of the T-spin condition. At least three of the four grey squares need to be filled. 22
3.7 Examples of T-spins. 23
3.8 Example of Stored Attack and Cancelling. 25
3.9 OpenTris Class Diagram 27
3.10 Example of a Tuck. 28
3.11 Example of a Z-spin 29
3.12 A game of survival. 33

4.1 Plot for fitness over iterations 35

1 Introduction

Tetris is a popular video game created in 1984 by computer programmer Alexey Pajitnov [1]. It is a puzzle game that requires players to strategically place sequences of pieces known as "Tetrominos", into a rectangular Matrix (refer to Figure 1.1). In the classic game, players attempt to clear as many lines as possible by completely filling horizontal rows of blocks, but if the Tetrominos surpass the top of the Matrix, the game ends.

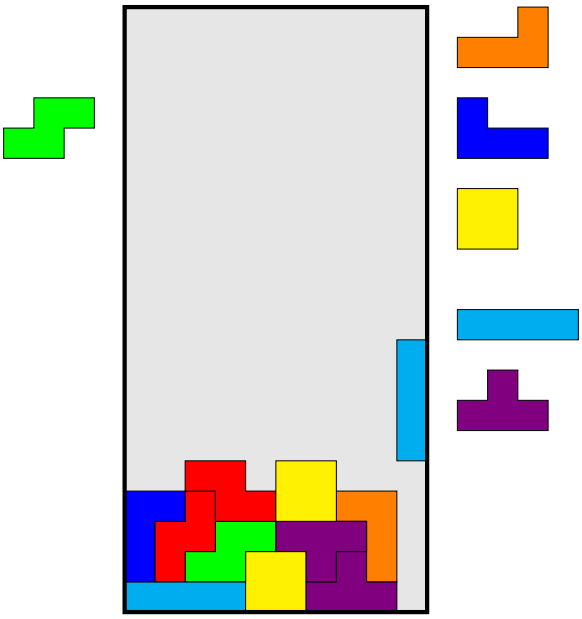


Figure 1.1: A typical modern Tetris game where four lines are about to be cleared. The Tetromino on the left of the matrix is the *Hold* piece and the pieces to the right of the matrix are collectively known as the *Queue*.

Since its release, mathematicians and computer scientists have been intrigued by the game of Tetris, leading to a diverse array of research endeavours exploring the various facets of the game, including its computational complexity [2], and its possibility of being won [3, 4].

1.1 Motivation

In their paper, Demaine, Hohenberger, and Liben-Nowell showed that it is NP-complete to optimise several natural objective functions of Tetris [2]. NP-completeness poses a significant challenge in computational problem-solving, as it denotes the absence of polynomial-time algorithms for efficient solutions [5]. Moreover, the discovery of a polynomial-time algorithm for any NP-complete problem implies that any problem in the set of NP, encompassing efficiently verifiable but potentially difficult problems, could

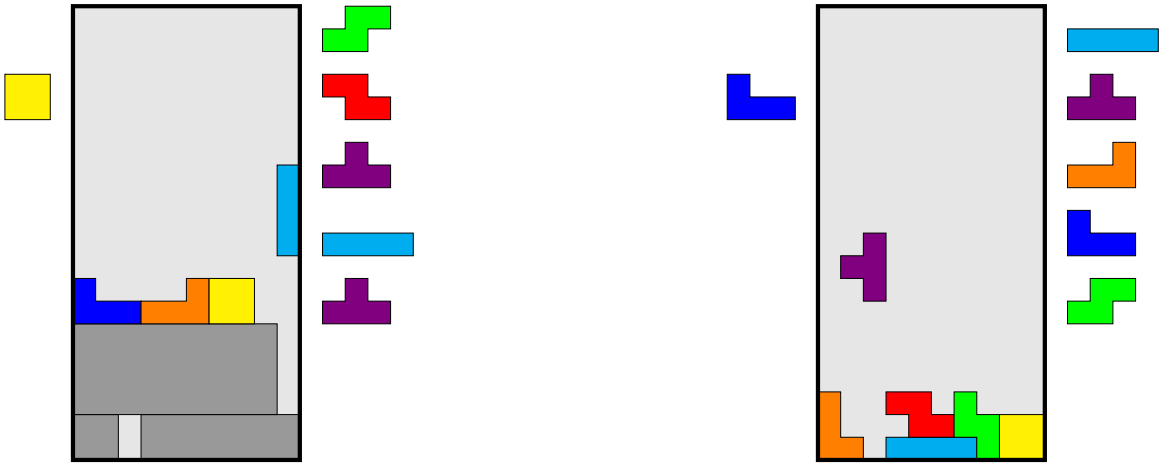


Figure 1.2: A typical game of Versus Tetris. Both players are trying to send lines to each other. The grey blocks are *Garbage Lines* sent from Player 2 (right) to Player 1 (left).

be solved in polynomial time [5]. NP-completeness extends beyond Tetris, with real-life instances of NP-complete arising in diverse fields such as route optimisation [6], job scheduling [7], and medicine [8].

To address these challenges, researchers have explored alternative approaches to tackle NP-complete problems, including the use of nature-inspired algorithms [9]. Although they might fail at finding optimal solutions, nature-inspired algorithms are able to return acceptable solutions in shorter running times [10]. In the context of optimising Tetris gameplay, studies have shown the effectiveness of using nature-inspired algorithms in playing the classic single-player game [11] [12]. However, there remains limited research on the effectiveness of nature-inspired optimisation algorithms in the multiplayer versus variant of the game.

1.2 Problem Statement

Versus Tetris (refer to Figure 1.2) presents a unique challenge in computational gaming due to its complex dynamics and real-time competitive nature. While previous research regarding the use of nature-inspired algorithms for Tetris optimisation have focused on single-player scenarios, the effectiveness of these algorithms in the multiplayer context remains largely unexplored. Despite the demonstrated success of these algorithms in improving single-player Tetris gameplay, their application to the multiplayer variant poses distinct challenges due to a different rule set and differing objectives that require further investigation.

1.3 Aim

The aim of this capstone project is to assess the effectiveness of genetic algorithms in playing the game of Versus Tetris. By integrating insights from genetic algorithms, the project seeks to create a robust and adaptable Tetris-playing software capable of competing against human players or other Tetris-playing programs. Through this endeavour, the project aims to contribute valuable insights into the application of nature-inspired algorithms in addressing computationally complex problems.

1.4 Objectives

The objectives of this project are as follows:

1. Formulate the problem of Versus Tetris for game AI.
2. Develop a playable game of Tetris that simulates gameplay for training and emulation.
3. Utilise a genetic algorithm to optimise gameplay strategies in Versus Tetris.

1.5 Project Scope

This project will focus specifically on the evaluation of genetic algorithms in the context of multiplayer versus Tetris. It will entail the development of a playable Tetris game capable of simulating gameplay and the training of algorithms. This simulation environment will facilitate in the analysis and evaluation of the algorithm's performance.

2 Literature Review

Tetris, a timeless puzzle game created by Alexey Pajitnov in 1984, has captivated players and researchers alike with its engaging gameplay and complex strategic elements. The game challenges players to manipulate sequences of geometric shapes known as "Tetrominos" to clear horizontal lines within a confined space. Despite its simple premise, Tetris has inspired extensive research in computational complexity and artificial intelligence due to its inherent difficulty.

This review will cover the fundamental challenges of Tetris, explore existing approaches, and assess the potential of nature-inspired algorithms to enhance gameplay strategies.

2.1 The Difficulty of Tetris

In their article, Demaine, Hohenberger, and Liben-Nowell [2] proved that optimising several natural objectives of Tetris is NP-complete, even with a deterministic finite piece sequence. A deterministic finite piece sequence refers to a game at which the player knows every single piece in the sequence, and where there are finite pieces in the sequence, i.e. the game can end without the player losing. The authors defined the natural objectives of the game as follows [2]:

1. maximising the number of rows cleared;
2. maximising the number of piece placed;
3. maximising the number of Tetrises - clearing four lines on the same move;
4. minimising the height of the highest filled grid square.

In 2020, Asif et al. [13] demonstrated that playing any game of Tetris with a matrix of eight or more columns, or four or more rows, is NP-complete, further showcasing the difficulty of the game . Both of these papers aim to highlight the difficulty of the game, but what does difficulty actually entail?

This section aims to elucidate some of the key concepts of computational complexity, which involves the study of intrinsic difficulties of computational problems [14], and attempt to justify the use of non-traditional algorithmic approaches to play the game.

2.1.1 Complexity Classes

The study of computational complexity asks questions about the intrinsic difficulty of computational problems [14]. Complexity classes are usually defined by referring to computation models and by putting suitable restrictions on them [15].

The class P encompasses all decision problems that are polynomial time solvable using a deterministic model of computation [16]. In this model, for any given input, the machine's computation follows a single predetermined path [5].

The complexity class *NP*, on the other hand is the class of all decision problems that can be solved in polynomial time by a nondeterministic algorithm [17]. The nondeterministic model of computation allows for guessing correct solutions out of polynomially many options in constant time [18], and solutions can be verified in deterministic polynomial time [5].

If all problems in *NP* can be reduced to some problem X, X is said to be *NP-hard* [5]. Reductions are useful in showing the relationship between computable problems, as a reduction from problem A to problem B tells us that problem B is at least as hard as problem A [18].

For a problem to be considered *NP-complete*, it must be a member of both the *NP* and *NP-hard* classes (refer to Figure 2.1) [18]. Therefore, *NP-complete* problems are some of the hardest problems in *NP* [16].

Since all problems in *NP* can be reduced to any *NP-complete* problem, if a deterministic polynomial time algorithm can be found for an *NP-complete* problem, it would also mean that the set *NP* is polynomial-time solvable, proving *NP* and P are equal sets. The question of whether $P = NP$ has famously been immortalised as one of the millennium prize problems from the Clay Institute of Mathematics [19]. Most researchers believe that $P \neq NP$ since years of effort have failed to yield efficient algorithms for *NP-complete* problems [16].

2.1.2 Justifying Non-traditional Algorithmic Approaches

It is important to note that even though Demaine, Hohenberger, and Liben-Nowell's [2] proof is based on a variant of Tetris that is quite different from Versus Tetris, the difficulty of Versus Tetris can be implied from the proof. Versus Tetris introduces additional gameplay factors, such as the competitive aspect where players can send "garbage" lines to their opponents, adding layers of complexity. As such, Versus Tetris is arguably more difficult than its classic variant, or Demaine, Hohenberger, and Liben-Nowell's variant [2].

Given the NP-completeness of Tetris [2, 13], it is evident that traditional determin-

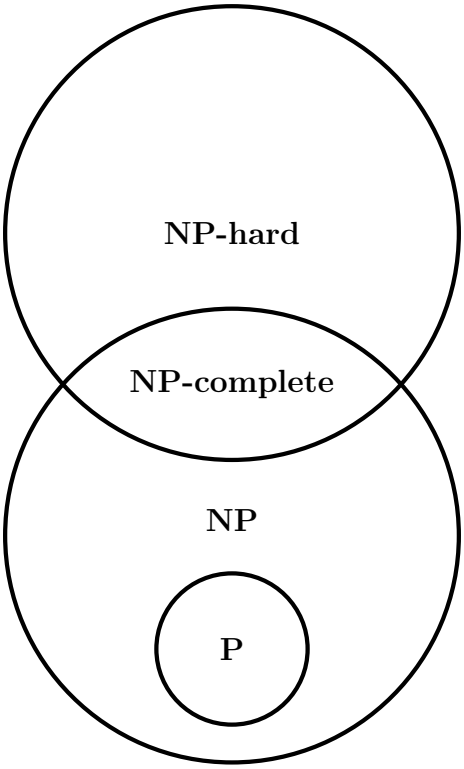


Figure 2.1: Visualisation of the sets P, NP, NP-hard and NP-complete.

istic algorithmic approaches would face significant challenges in efficiently playing the game [14]. The NP-completeness of the game implies that obtaining exact solutions are computationally intractable within a reasonable time frame.

To address this, we may relax the requirements for a solution. This can be done by either broadening the set of acceptable solutions or focusing on average-case scenarios instead of worst-case scenarios [14]. This shift in focus allows the use of more practical, non-traditional algorithms that provide acceptable solutions within a reasonable time, even if they are not optimal.

2.2 Approaches to Tetris

Now that the difficulty of Tetris has been established, we turn our attention to potential strategies that can be used to handle the game's complexity. In this section, we explore some of the innovative approaches that have been utilised to tackle the game of Tetris, such as imitation learning, upper confidence bound fort trees, deep reinforcement learning, and meta-heuristic algorithms.

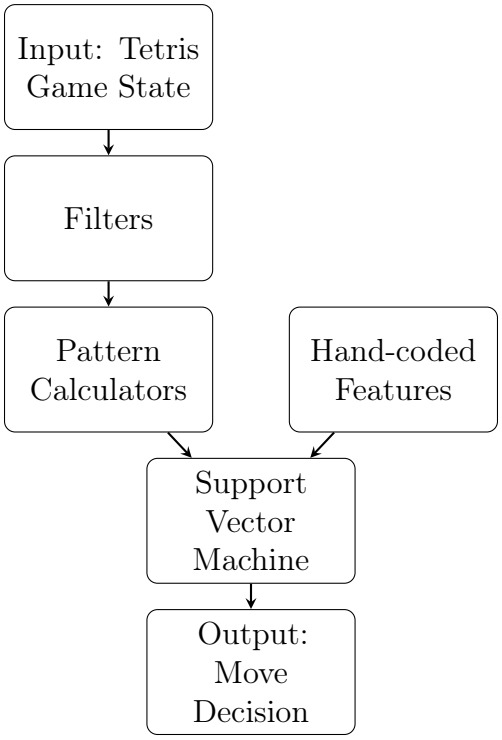


Figure 2.2: Abstraction of Zhang, Cai, and Nebel's System Components

2.2.1 Learning by Imitation

In 2010, Zhang, Cai, and Nebel [20] employed a learning by imitation approach to the game of Tetris. Imitation learning aims to mimic human behaviour in some given task by learning a mapping between observations of demonstrations [21].

In their approach, played Tetris games were fed into a machine learning model as inputs, where the model would receive positive feedback when a decision matched that of the imitated system, and negative feedback otherwise [20]. The learning is deemed successful if the trained model maintains similarity even when faced with data outside of the training set.

The learning system consists of several components, including filters, pattern calculators, and support vector machines (SVMs) (refer to Figure 2.2) [20]. Because of the time consuming nature of training SVMs, each Tetromino had its own dedicated filters, pattern calculators and SVMs, i.e. seven SVMs [1] are working together in the artificial player. The output of any particular SVM describes how similar a candidate move is to the choice of the imitated player.

Filters are made up of a multitude of patterns, which can be thought of as specific configurations in the playing field that the model uses to recognise and evaluate different game states [20]. Patterns do not take into account the whole matrix, but only a small

[1] Seven SVMs for the seven Tetrominos.

area in the playing field. Thus, the authors included a list of 19 hand-coded features that takes a more global look at the game, some notable features include the number of holes that will be created after the current placement, and the number of removed lines after the current placement. Both the activated patterns and the features are passed into the support vector machine, and the move that is calculated to be the most similar is played.

To evaluate their approach, Zhang, Cai, and Nebel used a human player as well as Fahey's artificial player [22] as systems for their model to imitate [20]. The outcome shows that their model can learn different styles of gameplay based on the imitated player, which is to be expected. When pitting the two learned models against each other in their variant of multiplayer Tetris [2], they found that the human imitated machine performed better than the one imitated from Fahey's AI. However, The system imitated from Fahey's AI outperformed that of the human imitated system in single player games.

2.2.2 Upper Confidence Bounds for Trees

Monte-Carlo Tree Search (MCTS) is a best-first search method that is based on a randomised exploration of the search space [23]. The algorithm evaluates states to determine the most rewarding actions, with rewards being calculated from the leaf to the root [24]. MCTS has the added benefit of learning from previous explorations, gradually building a game tree in memory, resulting in more accurate estimation of the most promising moves [23]. In 2006, Kocsis and Szepesvári [25] proposed a method, which was deemed Upper Confidence Bound for Trees (UCT), to improve the performance of the algorithm by applying a bandit algorithm known as UCB1[3] to Monte-Carlo Planning.

The UCB1 formula is employed to balance the exploration-exploitation trade-offs during the search [24]. It helps prioritise actions that are either promising or under-explored, which effectively prunes less promising branches of the search tree. This results in a more efficient search process and a smaller, more focused tree.

In 2011, Cai, Zhang, and Nebel [24] employed this approach to the game of Tetris. In their approach, the authors considered each state of the matrix, along with a given piece as a single node in the planning tree (refer to Figure 2.3). After an action is taken, if any number of rows are cleared, a predefined reward is given to the action and the most rewarding action is selected as the resulting move. To further improve efficiency, the authors pruned the search tree further, removing actions that resulted in disadvantageous holes in the field.

[2] In Zhang, Cai, and Nebel's variant of multiplayer Tetris, clearing n lines sends $n-1$ attack rows to the opponent, with each attack row containing $n-1$ empty cells [20]. Attack rows are added to the bottom of the opponent's matrix.

[3] UCB stands for Upper Confidence Bound

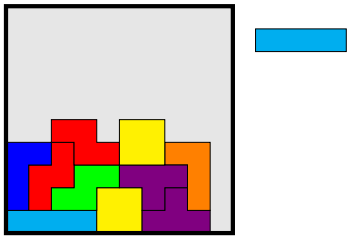


Figure 2.3: An example of a single node of the matrix state and piece in the planning tree.

Experimental results show that the artificial player shows promise, beating Fahey's AI [22] 91 times out of 100 games using the multiplayer Tetris rules mentioned in Section 2.2.1 [24]. The results also show that the artificial player outperformed the SVM based pattern player from Cai, Zhang, and Nebel's previous work, where they approached the game with imitation learning [20].

2.2.3 Deep Reinforcement Learning

In 2016, Stevens and Pradhan [26] adopted a deep reinforcement approach to the game of Tetris, using a convolutional neural network to approximate a Q function, which reflects the maximum predicted discounted sum of future rewards possible by following a given policy from states to actions (refer to Equation 2.1).

$$Q^*(s,a) = \max \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi] \tag{2.1}$$

where s_t is the state at time t, a_t is the action taken at time t, π is a policy function indicating what action to take in the current state, and is followed at every step from $t+1$ onwards, r_t is the reward obtained by taking action a_t at state s_t, and γ is a discount factor [26].

The authors defined a state as an image of the current game screen [26]. Actions were defined in two ways, single actions, where a single input used to move a piece is considered an action, and grouped actions, where an entire sequence of actions is considered that takes a piece from its initial point at the top of the matrix to the bottom. They also used two different types of rewards, the first being the scores given from the emulator used, which was defined as the number of lines cleared. The second type of reward was defined as a heuristic function proposed by Lee [27] (refer to Equation 2.2).

$$-0.51 \times Height + 0.76 \times Lines - 0.36 \times Holes - 0.18 \times Bumpiness \tag{2.2}$$

This heuristic takes into account the sum of the heights of every column, the number of cleared lines, the number of holes in the matrix, and the "bumpiness" - sum of the

absolute differences in height between adjacent columns [27]. Interestingly, the weights of the heuristic were calculated by using a genetic algorithm.

The training process was further improved by using an epsilon-greedy policy, where the agent takes the best known move most of the time, but has a small probability ϵ to take a random action [26]. They also tried a novel approach where an agent that was already trained to play well could suggest moves to the algorithm. In this approach, a random action is selected with probability ϵ_{rand}, an action suggested by the transferred agent is selected with probability ϵ_{agent}, and the best known action to the machine is selected otherwise.

Other than that, a method known as experience replay was used to create a memory for the neural network to train on. This memory consisted of every "experience" that the network had encountered, and was sampled when training the network. Prioritised sweeping was also utilised, where the priorities were placed on experiences which had the greatest error, and therefore the most room to improve.

From the experimental results, one of the agents showed the most potential. This agent utilised a standard greedy-epsilon policy - no trained agent was used to suggest moves, grouped actions over single ones, heuristic rewards and random sampling over prioritised sweeping. It outperformed the other agents proposed in this paper, with an average score of 18. However, its capabilities were far inferior to that of Lee's bot, which had an average score of 200.

Later in 2021, Chen [28] used a modified version of Stevens and Pradhan's algorithm as a baseline, and proposed several different approaches taken to further optimise the algorithm. The variant of the game used to test the author's algorithms has similar rules to that of modern Tetris games, where the system will generate bags containing a piece sequence with all 7 pieces, and bags are given to the player in sequence. This piece generation system ensures that the player will not need to experience a piece drought, since a piece is guaranteed to be given in the current or next bag. The game also has a hold feature, where the player has the option to hold on to a piece for future use, and swap it out for the piece in play.

Chen proposed that the quadratic rewards from the emulator scores in Stevens and Pradhan's paper might encourage the agent to take riskier moves, since clearing multiple lines would greatly increase the rewards given. However, this resulted in shorter survival times, as the agent waits for particular pieces. Knowing this, the author proposed utilising a linear reward function instead of a quadratic one. Comparing the results of both approaches, the authors found that the agent trained with a linear reward function outperformed that of a quadratic one, in both the survival time aspect, and the game

Figure 2.4: Z and S pieces.

Table 2.1: Probability of each piece in Chen's "hard" mode.

	Z	S	I	J	L	O	T
Probability	0.25	0.25	0.1	0.1	0.1	0.1	0.1

score aspect. However, it should be noted that the quadratic models have higher points per piece dropped, meaning that they were much more efficient than the linear ones.

Chen also proposed the use of a "harder" Tetris game, which had a higher probability for Z and S pieces (refer to Figure 2.4). The author set the probability of each piece as shown in Table 2.1. Doing this significantly decreased training time, reducing it from 3 days to a mere 6 hours.

Finally, Chen proposed an update rule that takes the probability of each piece into account (refer to Equation 2.3). This idea emerged from the observation that the only randomness found in a Tetris game is from piece generation. The training process involved enumerating all possible next pieces and actions to find the best action, storing the expected reward and Q-value for each state to a replay buffer and randomly sampling from the replay buffer to update the Q-network. Result of this change showed that the models trained under these update rules performed consistently better than the original models.

$$Q(s) \leftarrow Q(s) + \alpha[\mathbb{E}_{allPossiblePieces}[r + \gamma \max Q(s') - Q(s)]] \tag{2.3}$$

Experimental results show that Stevens and Pradhan's model [26] scored an average of 704.68 with modern rules in place, but Chen's model significantly outperformed it, with an average score of 40163.58, increasing almost 57-fold.

2.2.4 Meta-heuristic Algorithms

According to Yang [29], the term "meta-heuristic" does not have any agreed upon definitions, with some scholars even using the term interchangeably with the word "heuristic". Even so, most definitions agree on the same things. Almufti et al. [30] consolidated some of the properties that characterise most meta-heuristics as follows:

1. Meta-heuristics are strategies that guide the search process.
2. The goal is to efficiently explore the search space in order to find near-optimal solutions.

3. Meta-heuristic algorithm techniques range from simple local search procedures to complex learning processes.
4. Meta-heuristic algorithms are non-deterministic and approximate in nature.
5. Meta-heuristic algorithms are not designed to solve a specific problem.

Here, we will look into two works of literature that cover the use of two distinct meta-heuristic algorithms, namely, the harmony search algorithm, and the most valuable player algorithm, in optimising Tetris agents.

Harmony Search Algorithm

In 2011, Romero, Tomes, and Yusiong utilised the harmony search algorithm, a meta-heuristic algorithm that mimics the improvisation of music players [32], to approach Tetris [31]. In the context of the algorithm, harmonies are solution vectors containing potential weights of solutions. The algorithm can be described in the following steps [31]:

1. **Initialisation**: Program parameters are defined and the harmony memory is filled with random harmonies; each harmony is evaluated using a set objective function.
2. **Harmony Improvisation**: A new solution is created utilising these three methods:
   (a) **Creation of a new solution**: A new solution is created randomly with a probability of $1 - r_{accept}$, or an existing solution in the harmony memory is selected with a probability of $1 - r_{accept}$.
   (b) **Pitch adjustment**: The elements of the new harmony are modified with a probability of r_{pa}
   (c) **Evaluation**: The new harmony is evaluated using the objective function defined.
3. **Selection**: When a terminating condition is met, the best harmony in the harmony memory is selected.

In their approach, Romero, Tomes, and Yusiong [31] used a list of 19 feature functions, assigning numerical weights to each function. The best move for a current piece is chosen using a linear summation of these feature functions with their corresponding weights:

$$V(s) = \sum_{i=1}^{19} w_i f_i(s) \tag{2.4}$$

where s is the current state (board configuration), w_i represents each of the weights of the i^{th} feature function $f_i(s)$ and $f_i(s)$ is a function that maps a state to a real value. The

objective of the optimisation process is to find an optimal set of weights that result in the most number of cleared rows. Since 19 feature functions were defined, the solutions generated by the program came in the form of a vector of 19 weights.

In each iteration of the algorithm, when a set of harmonies (in the form of weights) are generated, Romero, Tomes, and Yusiong [31] then passed the weights to their own Tetris simulator, which plays a complete game of Tetris, utilising the weights provided to decide on piece placements. After playing the game, the simulator then returns the number of rows cleared by the agent. This serves as the objective function, where more cleared rows means better agent performance. When a termination condition has been met, the program then outputs the best solution created so far.

To evaluate their performance, Romero, Tomes, and Yusiong [31] used the spawned piece to cleared rows ratio as a metric. The lower this metric was, the better the performance of the algorithm. After letting the algorithm run for two weeks straight, the program was almost able to achieve the theoretical best case, which was derived by Fahey [22] to be 2.5.

Most Valuable Player Algorithm

The most valuable player algorithm (MVPA) is a meta-heuristic algorithm inspired from sport where players form teams and compete collectively to win championship games, and individually amongst each other to win the MVP trophy [33]. In 2022, Armanto, Putra, and Pickerling [34] compared the use of this algorithm with the genetic algorithm (GA) in a classic Tetris environment. This algorithm can be described in the following steps:

1. **Initialisation**: Program parameters are defined; candidate solutions are generated.
2. **Team Formation**: A number of teams are determined and players are divided evenly throughout the different teams.
3. **Individual Competition**: Each player tries to develop their skills based on the existing franchise players and MVP.
4. **Team Competition**: Two teams are selected at random to compete with one another, where the game cannot result in a loss.
5. **Application of Greediness**: After several competitions, individual player skills are evaluated to see if any improvements were made after the games. If a player is better after the games, it will be used for the next iteration.
6. **Application of Elitism**: 1/3 of all players with the worst skills will be replaced by 1/3 of the players with the best skills.
7. **Remove Duplicates**: If there are individuals who are identical to one another, one

of these individuals will be updated randomly.

Similar to harmony search, a list of features are selected, and both the MVPA and GA are used to optimise the weights for each feature.

Experimental results show that MVPA proved to optimise the game better than GA in terms of score and speed. In an endless game, MVPA managed to reach a score of 249 million while the GA only reached 68 million. The MVPA was also shown to find an acceptable solution faster than the GA in trials where the authors limited the spawn pieces. They observed that the MVPA converged much faster.

2.3 Playing Tetris with Nature-inspired Algorithms

Nature-inspired optimisation algorithms are a class of meta-heuristic algorithms. These algorithms are based on natural phenomena and biological systems, borrowing ideas like natural selection and evolution to converge on acceptable solutions [35]. In this section, we will highlight three such algorithms that have been adopted to attempt to tackle the game of Tetris, specifically, the aforementioned genetic algorithm (GA), ant colony optimisation (ACO), particle swarm optimisation (PSO).

All these works are similar to the ones mentioned in Section 2.2.4, in the sense that they all utilise a list of features from the board state and that they all utilise these algorithms to optimise the weights for said features to be used in a state evaluation function.

2.3.1 Genetic Algorithm

In 2005, Flom and Robinson [36] utilised a Genetic Algorithm (GA) to weight an evaluation function for Tetris. In their work, a variation of the genetic algorithm known as GENITOR was used. In a standard GA, the parents are replaced by their offspring after recombination [37]. The GENITOR approach differs in the sense that the offspring do not replace parents, but rather replace low ranking individuals in the population.

An agent's fitness was determined by the rank of the utility that the agent returned [36]. Flom and Robinson experimented with two different ways to calculate this utility. The first was to calculate the number of rows cleared by the agent before a game ends (either by losing, or by using up a finite number of pieces). The second way was to use the ratio of cleared rows per piece played. They found that both determiners had no noticeable difference in learning rate, and ultimately chose to use the first metric for the rest of their experimentation.

They also experimented with different crossover operations to find out which would

have the biggest positive impact on learning rate. This resulted in them using a uniform crossover approach. In their work, binary representations of the weights were used as chromosomes. Every generation, two parents were selected completely at random and recombined to make one child. With uniform crossover, a crossover mask with equally distributed 1-bits is used in the recombination process [38]. For example,

Parent 1: `001111`
Parent 2: `111100`
Mask: `010101`
Child 1: `011110`
Child 2: `101101`

Child 1 is parent 1 with some bits replaced as specified by the mask and child 2 is parent 2 with some bits replaced as specified by the mask.

Results show that the agents were able to continue learning each generation, and that even after 1000 runs, the agents have yet to breed into convergence [36]. Flom and Robinson concluded that genetic algorithms work very well at searching for good weights for the evaluation function used to allow an agent to play Tetris.

Later in 2015, Lewis [11] also used GA to attempt playing Tetris. In his implementation of the game, the agent was awarded a score after a line clear occurred (refer to Table 2.2).

Table 2.2: Score table for lines cleared in Lewis' Tetris game [11].

Lines Cleared	Score
1	2
2	5
3	15
4	60

The metric used to calculate the fitness of a candidate is the score efficiency, which is defined as the average score per pieces played. that a player earns during a game of Tetris. A 4-line clear requires at least 10 Tetrominos. Thus, the theoretical maximum for this metric is 60 score $\div$ 10 pieces = 6. Experimental results show that under this metric, the algorithm mostly converged after 30 generations with a minimum scoring efficiency of 2.4, which is equivalent to 20% of the theoretical maximum.

2.3.2 Ant Colony Optimisation

In 2009, Chen et al. [39] applied Ant Colony Optimization (ACO) to optimize the weights of a linear value function in the game of Tetris. The value function evaluates possible

game states based on 16 feature functions, where each feature is scaled between 0 and 1.

The weight vector, represented as w, is encoded into a 16-bit vector, forming a weight graph with 256 edges, where each edge corresponds to a bit value of '0' or '1'. Ants traverse this graph to find the optimal path, which corresponded to the optimal weight vector for the Tetris agent. This approach allowed the ACO algorithm to systematically explore different weight configurations and identify the most effective one for gameplay.

The performance of the ACO algorithm was compared with various reinforcement learning methods. Initially, when no heuristic was applied, the ACO algorithm converged prematurely to a suboptimal solution, resulting in an average of about 7,000 lines removed. This premature convergence highlighted the limitations of ACO without additional guidance mechanisms.

To address the issue of premature convergence, the researchers introduced a dynamic heuristic. This modification significantly enhanced the algorithm's performance. With the dynamic heuristic in place, the ACO algorithm's average lines removed exceeded 17,000. Moreover, the best weight configuration discovered by the algorithm achieved an impressive performance of over 23,700 lines removed. This marked improvement demonstrated the effectiveness of incorporating dynamic heuristics to guide the ants towards better solutions and avoid local optima.

The results of this study were further compared to those of other traditional reinforcement learning techniques. The ACO with dynamic heuristic outperformed most of these methods, including hand-coded approaches and Genetic Algorithms (GA). The superior performance of the ACO approach, particularly with the dynamic heuristic, underscored its potential in optimizing Tetris gameplay and surpassing the effectiveness of many conventional methods.

2.3.3 Particle Swarm Optimisation

In Langenhoven, Heerden, and Engelbrecht's [12] paper, the authors investigate the application of Particle Swarm Optimization (PSO) in training neural networks to play Tetris. The algorithm was used to optimize the weight values of neural networks. Unlike traditional hand-coded heuristic approaches, PSO enables the automatic discovery of effective gameplay strategies. By treating each neural network as a particle within the swarm, the algorithm iteratively improves the network's performance based on a fitness function, which in this case is the game score achieved by the Tetris-playing agent.

The use of PSO for training neural networks in this context represents a significant shift from conventional methods. Traditional approaches rely heavily on hand-coded heuristics, which require extensive domain knowledge and manual tuning. PSO, on the

other hand, automates the discovery process, allowing the neural networks to evolve and adapt their strategies over time. This automatic optimization is achieved through the collective behavior of the swarm, where each particle adjusts its position based on both its own experience and the experience of neighboring particles.

In the experiments conducted, neural networks were trained using PSO over multiple iterations. The results indicated a continuous improvement in the performance of the agents as they learned to play Tetris more effectively. Initially, the PSO-trained agents displayed suboptimal performance compared to traditional hand-coded heuristics. However, over time and with more iterations, the agents began to exhibit increasingly sophisticated gameplay strategies. Despite these advancements, the PSO-trained agents have not yet surpassed the performance of the best hand-coded algorithms, suggesting room for further optimization and refinement in the training process.

Despite these advancements, the PSO-trained agents have not yet surpassed the performance of the best hand-coded algorithms. This outcome suggests that while PSO is effective in automating the training process and uncovering new strategies, it still requires further optimization. The iterative nature of PSO allows for gradual improvement, but achieving human-level performance in Tetris remains a challenge that necessitates ongoing refinement and experimentation.

The experimental results demonstrate the potential of PSO in the domain of game-playing AI, particularly in automating the training process and discovering novel strategies. The study underscores the feasibility of PSO for such applications while also pointing to the challenges that remain in achieving human-level performance in Tetris through automated training methods.

3 Methodology

This chapter outlines the systematic approach employed to investigate the application of a genetic algorithm to Versus Tetris. We define the rules of the game in Section 3.1, show our thought process in building the simulation in Section 3.2, describe our implementation of the algorithm in Section 3.3, and propose methods of evaluation in Section 3.4.

3.1 Defining the Rules

Tetris is a game that has many different variations, each with varying rules. In this section, we will formally define the game's rules that will be enforced throughout our implementation.

3.1.1 Tetromino Sequence Generation

Regardless of the variant, there are seven distinct Tetrominos present in every game of Tetris, each with their own names (refer to Figure 3.1). However, the way the sequences of Tetrominos are generated differ depending on the variant defined. In our variant of the game, the rules of piece generation will be similar to that defined by Chen [28].

The system used to generate piece sequences is known as *7-bag*. It works by generating a sequence of all seven Tetrominos permuted randomly, as if they were drawn from a bag [40]. The sequence is then given to the player before another bag is generated. This means that there are 7! or 5,040 different permutations of the seven pieces for any given bag. The system gives each one of these permutations an equal probability of occurring. This ensures that there will not be periods of piece drought – where a player waits for an extended period of time for a preferred piece,[1] or piece rain – where a player keeps receiving the same pieces consecutively.

(a) O (b) T (c) Z (d) S (e) L (f) J (g) I

Figure 3.1: The Tetrominos and their names.

[1]The maximum number of pieces between a piece X and another piece X is 12.

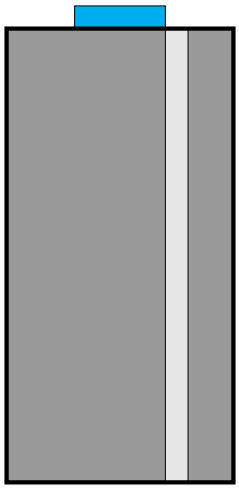


Figure 3.2: I piece is spawned at the 21st row of the matrix, but the player can survive by slotting the piece into the eighth column of the matrix.

3.1.2 Matrix Dimensions

Under official guideline Tetris rules, the matrix dimension is 10 columns wide and 22 rows tall, but the top two rows are hidden. These two rows will allow the player to survive even if the middle of the matrix is filled up by moving the piece to the side of the matrix (refer to Figure 3.2).

3.1.3 Spins

The rotation system used by guideline Tetris games is known as the *Super Rotation System* (SRS), and it defines how Tetrominos spawn, rotate, and what kicks can be performed [40]. It is important for us to define the rotation rules here so that we can create a simulation of Tetris that is as accurate as possible.

When unobstructed, the Tetrominos all rotate around a single point (refer to Figure 3.3). The O and I pieces rotate around an intersection of grid-lines whereas the other pieces rotate around the centre of a grid-box.

However, if the piece is obstructed by the wall, the floor, or another fixed piece, the game attempts to "kick" the Tetromino into an alternative position nearby [40]. When this occurs, the game tests 5 potential positions to kick the piece towards; if the position of all the tests are unavailable, the rotation itself fails, resulting in the piece not rotating at all. Other than the O piece, all the other pieces are able to kick into different positions. The I piece has its own set of kick values, but the remaining pieces all share identical ones.

Tables 3.1 and 3.2 show the kick values for these pieces [40]. The kick values represent translations relative to rotations. The following conventions will be used for naming rotation states for the tables:

Table 3.1: I Tetromino Wall Kick Data.

	Test 1	Test 2	Test 3	Test 4	Test 5
0 → R	(0, 0)	(-2, 0)	(+1, 0)	(-2, -1)	(+1, +2)
R → 0	(0, 0)	(+2, 0)	(-1, 0)	(+2, +1)	(-1, -2)
R → 2	(0, 0)	(-1, 0)	(+2, 0)	(-1, +2)	(+2, -1)
2 → R	(0, 0)	(+1, 0)	(-2, 0)	(+1, -2)	(-2, +1)
2 → L	(0, 0)	(+2, 0)	(-1, 0)	(+2, +1)	(-1, -2)
L → 2	(0, 0)	(-2, 0)	(+1, 0)	(-2, -1)	(+1, +2)
L → 0	(0, 0)	(+1, 0)	(-2, 0)	(+1, -2)	(-2, +1)
0 → L	(0, 0)	(-1, 0)	(+2, 0)	(-1, +2)	(+2, -1)

Table 3.2: T, Z, S, L and J Tetromino Wall Kick Data.

	Test 1	Test 2	Test 3	Test 4	Test 5
0 → R	(0, 0)	(-1, 0)	(-1, +1)	(0, -2)	(-1, -2)
R → 0	(0, 0)	(+1, 0)	(+1, -1)	(0, +2)	(+1, +2)
R → 2	(0, 0)	(+1, 0)	(+1, -1)	(0, +2)	(+1, +2)
2 → R	(0, 0)	(-1, 0)	(-1, +1)	(0, -2)	(-1, -2)
2 → L	(0, 0)	(+1, 0)	(+1, +1)	(0, -2)	(+1, -2)
L → 2	(0, 0)	(-1, 0)	(-1, -1)	(0, +2)	(-1, +2)
L → 0	(0, 0)	(-1, 0)	(-1, -1)	(0, +2)	(-1, +2)
0 → L	(0, 0)	(+1, 0)	(+1, +1)	(0, -2)	(+1, -2)

- 0 = spawn state
- L = state resulting from a counter-clockwise rotation from the spawn state
- R = state resulting from a clockwise rotation from the spawn state.
- 2 = state resulting from 2 successive rotations in either direction from spawn.

3.1.4 Attacks

In Zhang, Cai, and Nebel's papers [20] [24], they defined attack rules for their variant of multiplayer Tetris. Here, we will define the attack rules of guideline Tetris, which will be used for our implementation.

Garbage

Garbage lines (or attack lines) are lines sent from one opponent to another. They all have a one column well that would allow a player to clear them (refer to Figure 3.4). Garbage is

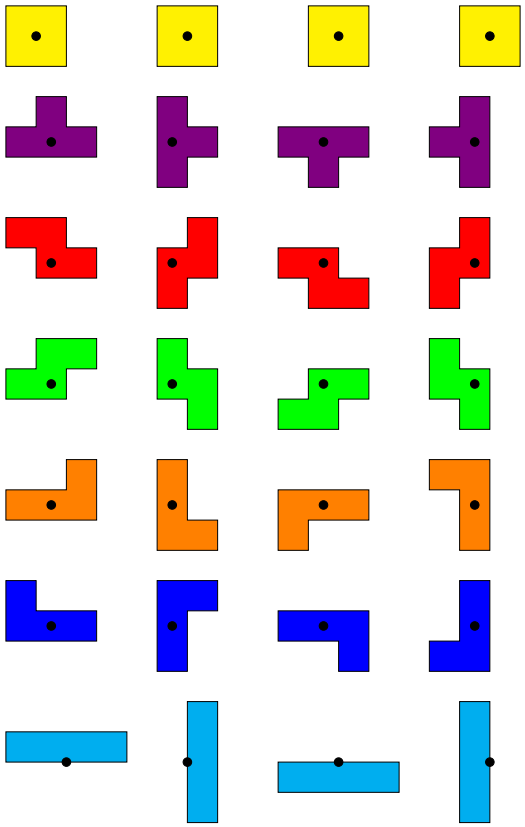


Figure 3.3: The four rotation states of all seven Tetrominos. The black dots represent centres of rotation.

Table 3.3: Attack Table for Line Clears

Lines Cleared	Garbage Sent
1	0
2	1
3	2
4	4

sent in batches, when a player performs an action that is able to send garbage lines, those garbage lines are grouped together, and the wells of grouped garbage lines are always on the same column. However, if the same player performs more attack actions, there may be multiple different wells present in garbage lines. Garbage lines are a key part of the game as players can also use these lines to send attacks back to the opponent.

Line Clears

Any action that clears two or more lines simultaneously will send garbage lines to the opponent (refer to Table 3.3). It may seem obvious that it is always better for the player to go for Tetrises (4-line clears), but we will shed light on another mechanic that will impact strategy largely in Section 3.1.4.

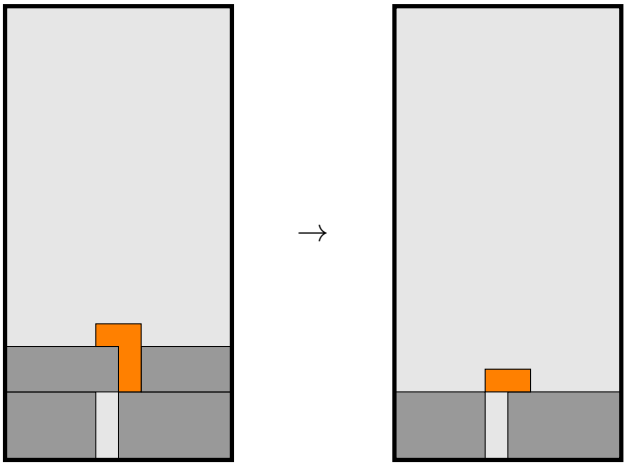


Figure 3.4: Example of garbage lines (gray) being cleared by an L piece.

T-spins

T-spins are a feature implemented in modern Tetris games that allow players to increase their scoring without simply performing Tetrises. To define a T-spin, we can first imagine the T piece in a 3 × 3 box (refer to Figure 3.5).

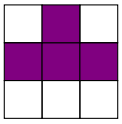


Figure 3.5: T-piece in a 3 × 3 box.

For a T-spin to occur, the last movement of a T piece must have been a rotation, and at least three of the four corners in the 3 × 3 box needs to be filled (refer to Figure 3.6).

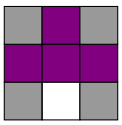
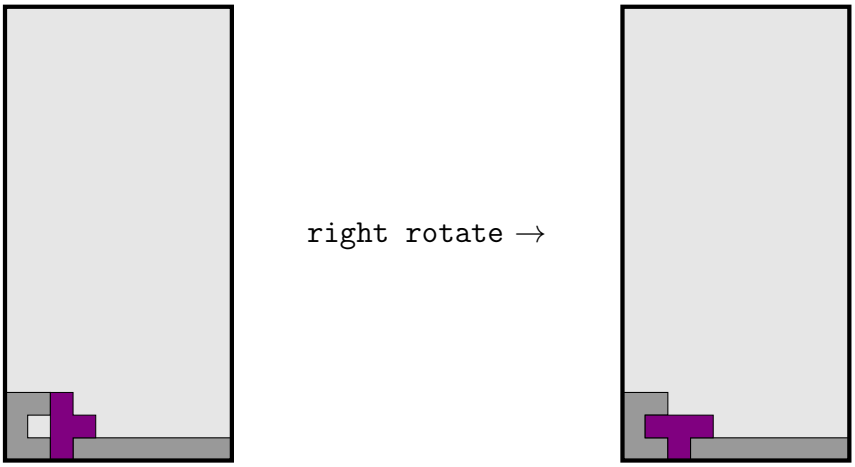


Figure 3.6: Visualisation of the T-spin condition. At least three of the four grey squares need to be filled.

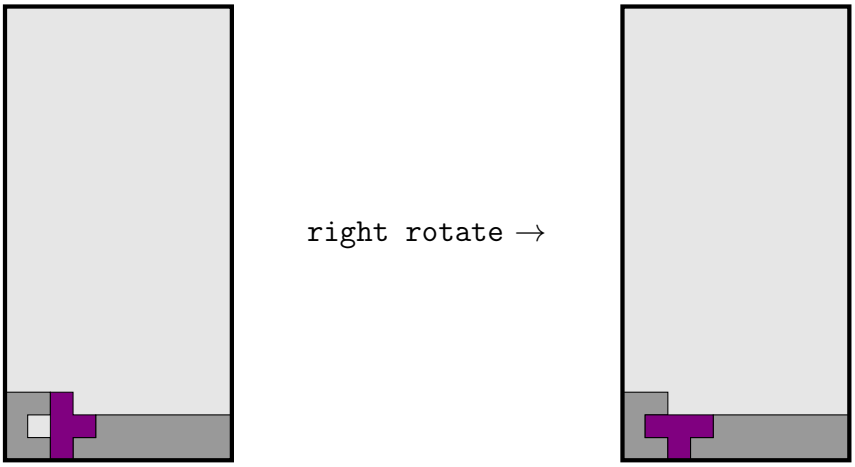
Regular T-spins send double the lines that they clear (refer to Table 3.4). Figure 3.6 shows a few examples of T-spins.

Back-to-back (B2B)

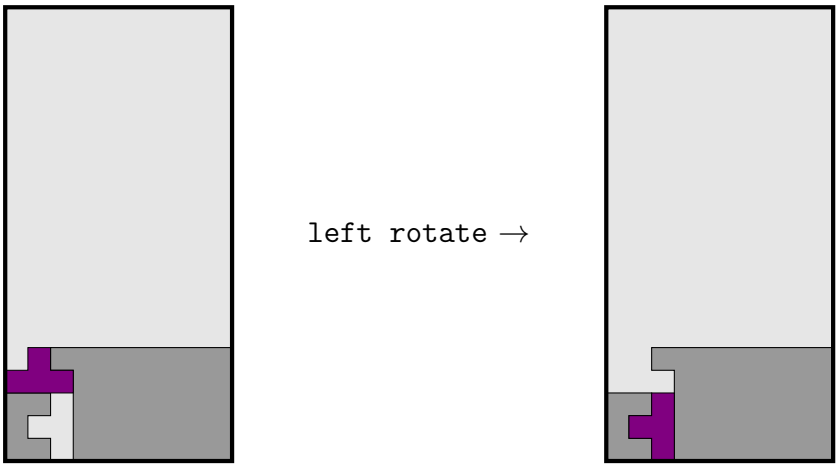
Back-to-back is a feature that encourages players to continuously play for Tetrises and T-spins. When these attacks are done back-to-back, it increases the garbage lines being sent to the opponent by one. For example, if a player performs a Tetris, and their following



(a) Example of a T-spin Single.



(b) Example of a T-spin Double.



(c) Example of a T-spin Triple.

Figure 3.7: Examples of T-spins.

Table 3.4: Attack Table for T-spins

T-Spin	Garbage Sent
Single	2
Double	4
Triple	6

attack is also a Tetris, they would send a total of 9 lines; 4 from the initial attack, and 4 + 1 back-to-back from the latter.

Combos

When a player clears multiple lines in quick succession, with every subsequent piece being used to clear at least one line, they initiate a combo. At certain points in a combo, additional lines are added to the garbage group sent to the opponent (refer to Table 3.5).

Combo	Additional Garbage
0	0
1	0
2	1
3	1
4	2
5	2
6	3
7	3
8	4
9	4
10	4
11+	5

Table 3.5: Combo Table

Perfect Clears

A perfect clear occurs when a player clears the matrix completely, leaving no remnants of existing Tetrominos behind. It can be difficult to perform perfect clears in Versus Tetris, because if there is an odd number of garbage sent from the opponent, a perfect clear becomes completely impossible. Because of this, a perfect clear has the highest attack in a single move, sending 6 lines of garbage to the opponent.

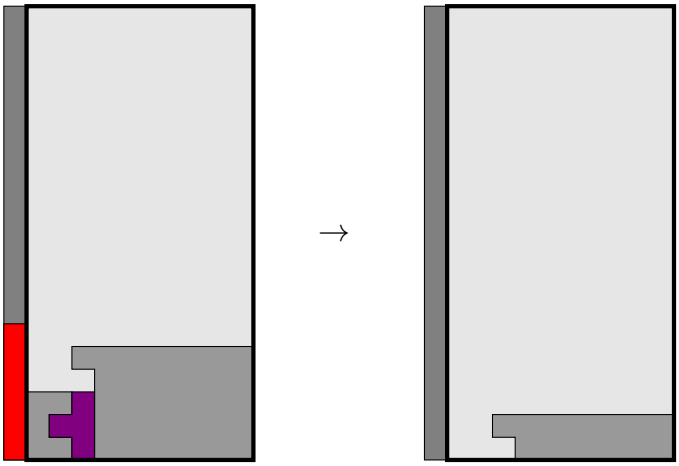


Figure 3.8: Example of Stored Attack and Cancelling.

Stored attack

When garbage is being sent to an opponent, the garbage lines do not suddenly appear in the opponent's matrix. It is stored in an attack potential, usually represented as a vertical meter on the side of the matrix (refer to Figure 3.8). This is to let the opponent see the potential attacks that might be coming, and allows them to clear lines in order to perform cancelling.

Cancelling is done when the garbage sent in a move is equivalent or higher to the number of garbage lines stored in the attack potential. This allows players to defend their matrix from unfavourable garbage. Figure 3.8 shows an example of cancelling a 6-line attack potential with a T-spin Triple (refer to Figure 3.7c).

3.1.5 Gravity

In modern Versus Tetris games, gravity starts slow but slowly increases in order to force the players to make decisions quicker. However, in our implementation, we are more interested in the quality of the decisions being made, not the speed at which it is being made. Therefore, in this implementation, gravity is not a factor.

This was implemented by disabling gravity entirely, forcing players to have to manually input "soft drop" until they reach their desired height. When the game is played, the speed of the bots will follow the pace of the opposing player, similar to a turn-based game like chess. The player will play a move, and the bot will play a move. With no gravity, the player will be able to take as long as they want to plan ahead.

3.1.6 Other Features

Here two additional features that may help a player improve decision making will be listed.

- **Hold**: Much like in Chen's [28] work, the hold function is used to hold on to a piece for future use and may be swapped any time with a piece in play.
- **Five-piece Lookahead**: In our implementation of the game, five pieces of lookahead will be given to the player at all times, i.e. the player will be able to see five-pieces in their queue. This does two things:
    1. It allows our player to potentially plan for future pieces.
    2. It allows our player to potentially calculate the probability of upcoming pieces.

In our implementation, although the five-piece lookahead and hold features are added, the bot is unaware of the current held piece or the five pieces ahead of it. The decision to do so was made because of the increasing space and time complexity required to generate the legal placements for these pieces.

3.2 Building the Simulation

In this section, we discuss the development of OpenTris, an open source Versus Tetris simulation built to aid in research work. Additionally, we delve into the logic behind our bot behaviour.

3.2.1 OpenTris

Since there were little to no tools found for Versus Tetris simulation, we built our own. OpenTris is an open source Versus Tetris simulation that is aimed to aid current and future research work. In this section, we outline the decisions made throughout the development process and explain each class involved in the simulation.

Development Decisions

To build the simulation, the programming language or framework used does not matter; we just needed to build a working simulation. We chose Python as our language, due to its ease of use, extensive community support, and the fact that it is one of the most commonly used programming languages in academic research. Python's readability and flexibility made it ideal for rapid prototyping and the iterative development required in this project.

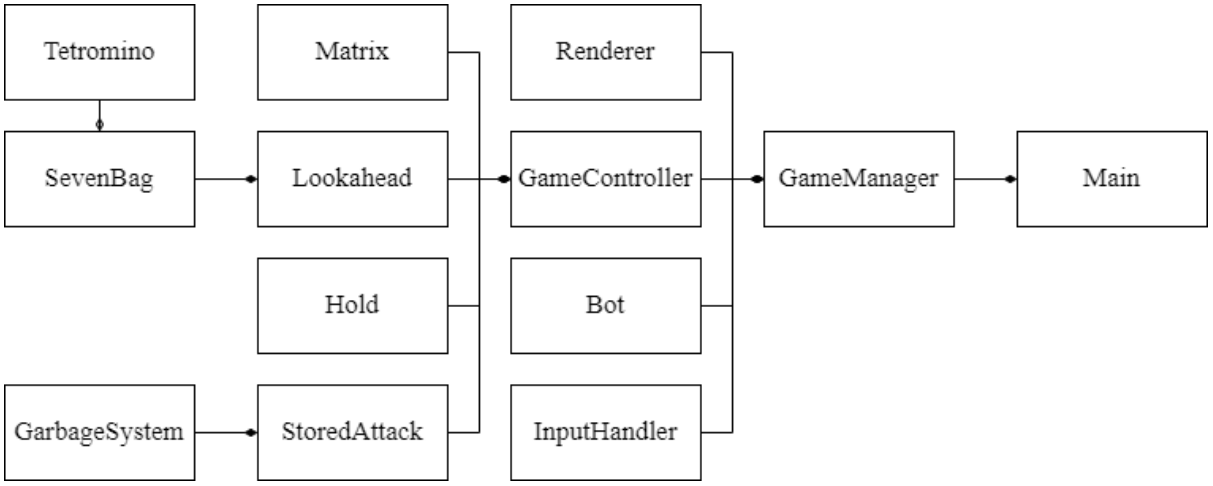


Figure 3.9: OpenTris Class Diagram

For the game itself, we utilised Pygame, a framework designed specifically with game development in mind. Pygame allowed us to focus on gameplay logic rather than low-level implementation details, such as managing game loops, rendering graphics, and handling user inputs. By leveraging Pygame's built-in features, we streamlined the development process and ensured that our efforts were directed toward creating a robust simulation.

Classes

Figure 3.9 shows a class diagram outlining the main classes in the program as well as the relationship between these classes. .

1. **Tetromino:** used to describe a Tetromino, including its name, position, colour and rotation.
2. **SevenBag:** used to describe the Seven Bag system defined in section 3.1.1
3. **Lookahead:** contains information on the next five pieces.
4. **GarbageSystem:** generates the holes for garbage being sent.
5. **StoredAttack:** used to store information on current stored attack, as defined in section 3.1.4
6. **GameController:** handles logic for movement, holding, attack calculation, and piece spawning.
7. **Renderer:** renders everything that the user sees, including the lookahead, hold, matrices and pieces.
8. **Bot:** handles bot behaviour, further expanded upon in section 3.2.2
9. **InputHandler:** allows human player to interact with the simulation.

Figure 3.10: Example of a Tuck.

10. **GameManager:** manages game state, ensures that game is turn based.
11. **Main:** houses the `main` function, the main entry into our programme.

Additional Details

We later added a piece limit to our simulation, due to the fact that when convergence occurs and the bots start agreeing more and more, they start playing too similarly for too long, resulting in a game that plays forever.

3.2.2 Bot Behaviour

In this section, we take a deeper look at how the Tetris AI actually works. We discuss how the bot finds legal placements, what the limitations are for our placement finding method, how our bot decides on a placement, as well as how the bot performs the actions required to achieve a chosen placement.

Generating Legal Placements

To generate the legal placements for a piece, the bot rotates copies of the current Tetromino and attempts to place them in each accessible column, iterating through all possible rotations. To optimize memory usage, the bot checks whether a new matrix state resulting from an action sequence has already been considered. If so, the sequence is ignored. Doing this allowed us to cut down on memory use, specifically with pieces that had symmetry with their other rotations, such as the 'O', 'I', 'Z', and 'S' pieces.

In order for the bot to consider a wider range of moves, it had to be able to perform two additional sequences:

- **Tucks:** Soft-dropping a piece and tucking it under a filled cell (refer to Figure 3.10).
- **Spins:** Soft-dropping a piece into a location and performing rotations that lead to a new matrix state, e.g. T-spins.

In order for the bot to find these placements, we look through the existing list of resulting landing states that have the same initial rotation and compare each state with its previous state. When a difference exists, we know that we should consider attempting tucks and spins in this position.

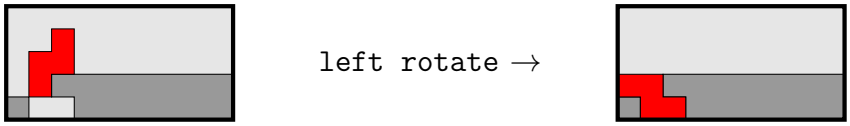


Figure 3.11: Example of a Z-spin

To find tucks, we ask the bot to drop a piece where we predict a piece is "tuckable" and ask it to move the piece left and right until a new matrix state is found. If a new matrix state is unable to be found, we simply drop the piece one cell lower and attempt the same action. These tucked placements are also considered when finding spins.

Finding spins was just about getting every considered landing state and trying to spin left and right at most four times to see if any new resulting matrix positions are reached. If a rotation fails to produce a new state, the bot abandons that direction.

Placement Limitations

The decision to ignore repeated states prevent certain spins such as Z-spins to occur (refer to Figure 3.11). This is due to the fact that rotating a Z left and right produce an identical shape. Our bot starts by considering the right rotate, but because of the characteristics of the kick table (refer to Table 3.2), a Z-spin can only occur after the bot performs a left rotate.

While this trade-off limits some moves, it significantly reduces memory consumption, ensuring efficient gameplay while preserving the majority of critical placements.

Also, due to the exponential space complexity of considering all five-lookahead pieces, our bot ended up not considering any of them. It also does not consider the hold piece's potential states. The bot sees holding as a way to retain the current matrix state. Once hold is used, it cannot be used again, so there is a possibility that the bot performs a hold and ends up with a worse piece for the scenario it is facing.

Placement Decision

To decide on a placement, the bot uses a weighted feature evaluation function (refer to Equation 3.1), much like the works included in Sections 2.2.4 and 2.3. This weights on this function are tweaked using a genetic algorithm; this will be expanded further in Section 3.3.

$$\sum_{i=1}^{10} w_i f(S)_i \tag{3.1}$$

where S is a state, f_i is the feature function of said state, and w_i is the corresponding

weight of feature f_i. The weights for each feature will be optimised using nature-inspired algorithms.

The list of numerical features and their description are as follows:

1. **Pile height:** The row number of the highest occupied cell in the grid.
2. **Hole count:** The number of holes[2] on the matrix.
3. **Column transitions:** The sum of all vertical transitions between occupied and unoccupied cells in each column.
4. **Row transitions:** The sum of all horizontal transitions between occupied and unoccupied cells in each row.
5. **Hole depth:**The number of filled cells above holes summed over all columns.
6. **Cumulative well depth:** The sum of accumulated depths of the wells.
7. **Row holes:** The number of rows that contain at least one hole.
8. **Line clears:** The number of lines cleared from the current placement.
9. **Attack:** The number of lines sent to the opponent or cancelled from the current placement.
10. **Back-to-back:** The back-to-back number from the current placement.

After evaluating each considered placement, the bot simply chooses the one that returns the highest evaluation value.

Taking Actions

After choosing a placement, the bot sees a list of strings that contain the instructions to reach the placement. We simply go through the list and perform the actions based on the strings.

3.3 Genetic Algorithm Implementation

The goal of the genetic algorithm is to optimise the Tetris bot's decision-making for Tetris gameplay by finding a set of suitable weights that allow the bot to play well.

3.3.1 Representation

Each individual was represented as a weight vector, where each weight corresponds to a specific feature. These weights collectively determine the bot's decision making process.

[2]A hole is an unoccupied cell with occupied cells above it

Each weight served as a chromosome in the genetic algorithm.

3.3.2 Population Initialisation

The initial population was generated randomly to ensure diversity. Each individual was assigned a unique set of weights, sampled uniformly across a range between -1 and 1. This diversity provided a broad search space for the algorithm to explore. The population was also fixed at 6, to ensure that fitness for the population can be calculated swiftly.

3.3.3 Fitness Function

To calculate the fitness function, we first had each of the individuals in the population play a game against one another. The fitness is then calculated using the average data from each match.

Initially, we used an individual's win-rate as their fitness, but we found this to be an unsuitable metric for Versus Tetris. If an individual was able to outlive its peers, it's win-rate would be incredibly high. The issue is that in the event that all players in a population are playing terribly, the algorithm will settle quickly into a highly suboptimal solution.

We then attempted to use average attack efficiency. The metric used to define attack efficiency is attack per piece, which is calculated using Equation 3.2. This came with its own problems, most notably, the bots became obsessed with high attack, but ended up with terrible survivability.

$$\text{Attack per Piece } (APP) = \frac{\text{Total Attack}}{\text{Total Pieces Placed}} \tag{3.2}$$

Finally, the fitness function that we decided on utilised both of these values. Our fitness was calculated using equation 3.3. We believe that this function best describes the goal of the bot. It should be trying to win, have high attack efficiency, all whilst maximising its pieces placed, and therefore its survivability.

$$\text{Fitness} = \text{Win-rate} + \text{Average Attack Efficiency} + \frac{1}{\text{MAX PIECES}} * \text{Pieces Placed} \tag{3.3}$$

3.3.4 Selection

For our implementation, we did ranked based parent selection. Each fitness was sorted into ranks, with higher ranked individuals having higher probabilities of being selected. However, we wanted to make sure that the algorithm explores as much as possible in earlier iterations and exploits as much as possible in later ones. Therefore, the probability

value of each ranked individual was scaled depending on the iteration. Later iterations made higher ranked individuals more likely to be selected.

3.3.5 Crossover

Crossover combined the genetic material of two parent individuals to produce offspring. A uniform crossover method was employed, where each weight in the offspring was randomly selected from one of the parents.

3.3.6 Mutation

Mutation introduced random changes to the offspring's weights to maintain diversity and prevent premature convergence. Each weight had a small probability of mutation, and the changes were applied by adding a small random value between -0.5 and 0.5. The mutation rate was set at 0.3 to balance exploration and exploitation.

3.3.7 Termination Criteria

The algorithm terminated after a 100 generations or when no significant improvement in the population's average fitness was observed over a predefined number of generations. These criteria ensured that the GA did not run indefinitely and stopped when further optimization became negligible.

3.4 Evaluating Gameplay

In this section, we will highlight two methods we will be evaluating the best-performing agent created from our algorithm. The two methods will be used to evaluate attack efficiency and survivability.

3.4.1 Evaluating Attack Efficiency

To evaluate attack efficiency of our bot, we reuse Equation 3.2 to calculate the Attack per Minute (APP). In order to test this, we would let the bot play normal single-player games of Tetris in isolation with limited pieces, and calculating the potential attacks that would be sent. The higher an algorithm's APP is, the more efficient it is at attacking.

3.4.2 Evaluating Survivability

To evaluate the survivability of our optimised bot, we let it play a mini-game known as "Survival". The goal of the bot in this mini-game is to survive as long as possible with

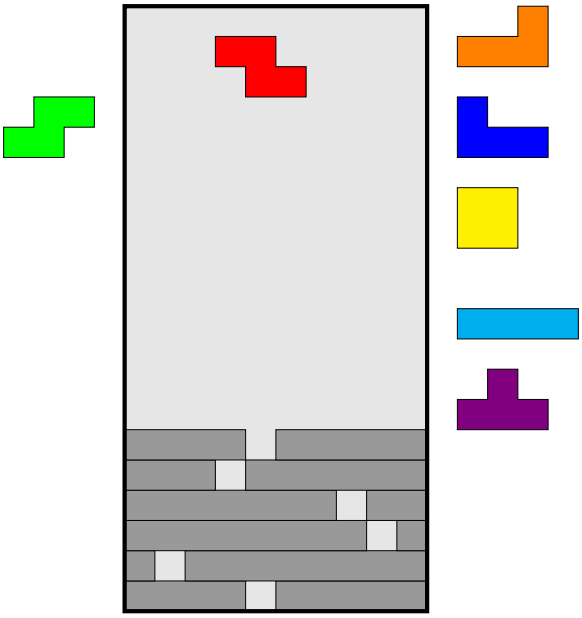


Figure 3.12: A game of survival.

single line garbage being stored in each turn (refer to Figure 3.12). Every time the player plays a piece that does not clear a line, a new garbage line appears at the bottom of the matrix

The metric that would be used for evaluating the bot would be the tetrominos placed. In this test, the higher the metric, the higher the survivability of the bot.

4 Results and Discussion

In this chapter, we take a look at our objectives and ensure that they have been delivered, the results obtained from the study, including the weight vector optimised by the genetic algorithm according to our fitness function, the outcomes of the two gameplay evaluation methods defined in section 3.4, and do a thorough analysis of the results.

4.1 Validation of Objectives

In this project, we have successfully achieved the three objectives we set out to deliver.

1. We successfully formulated the problem of Versus Tetris for game AI in our methodology.
2. We successfully developed a playable game of Tetris that simulates gameplay for training and emulation.
3. We successfully utilised a genetic algorithm to optimise gameplay strategies in Versus Tetris.

4.2 Genetic Algorithm Results

Figure 4.1 shows a plot for the fitness values of the best individuals in each iteration from the first to the 75th. Our run resulted in convergence occurring on the 75th run.

The fitness value can be seen fluctuating aggressively, even towards the later iterations. This is because of the fact that we use win-rate as part of our fitness function. As the individuals in the population learn to play better and better, the game becomes more difficult, and even the best performing agents end up with a lower win-rate. Although our fitness is fluctuating, the individuals are still learning to play better.

The weights for the best players from iteration 1, 20, 50, and 75 are listed in Table 4.1. Several features experience substantial changes in weight, reflecting their changing importance during the optimisation process. For example, the weight for Pile Height shifts from slightly positive in iteration 1 to strongly negative by the final iteration.

Some features can also be seen converging or stabilising at extreme values. For instance, Column Transitions consistently decreases, reaching -1 by iteration 50 and remaining there. However, there also exists some weights that exhibit non-linear trends. Row Transitions, for example, can be seen shifting from positive, to negative, and then again to positive and to negative.

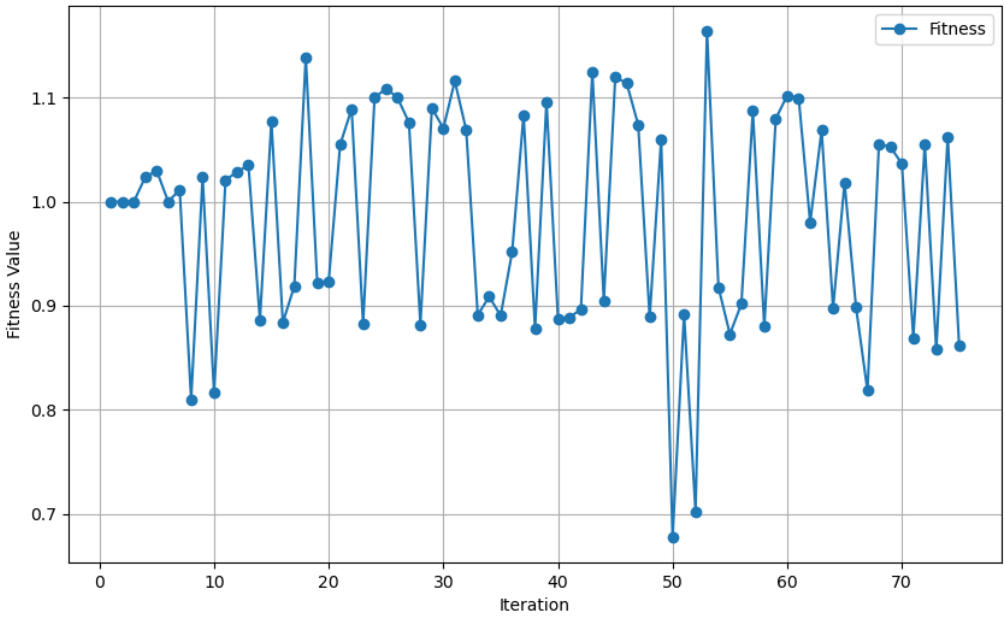


Figure 4.1: Plot for fitness over iterations

Table 4.1: Weights of best fit individuals in iterations 1, 20, 50, and 75.

Feature	Iteration Weights (5 d.p.)			
	1	20	50	75
Pile Height	0.21960	0.03682	-0.88214	-0.92416
Hole Count	0.42001	-0.35693	0.18129	-0.63448
Column Transitions	-0.67151	-0.54675	-1.00000	-1.00000
Row Transitions	0.44741	-0.94860	-0.06702	-0.75189
Hole Depth	-0.95575	-0.45945	-0.51573	0.02639
Cumulative Well Depth	-0.50302	-0.16005	-0.56181	-0.70165
Row Hole	-0.50031	0.00446	-0.17533	0.07719
Line Clears	0.37663	-1.00000	-0.85188	0.48772
Attack	-0.51747	-0.83629	-1.00000	-0.36715
Back-to-back	-0.30431	0.28046	0.98861	1.00000

Table 4.2: Average attack efficiency of best fit individuals in iterations 1, 20, 50, and 75.

Iteration Individuals	1	20	50	75
Attack Efficiency (5 d.p.)	0.03984	0.09307	0.07723	0.03564356435

Table 4.3: Average pieces placed by best fit individuals in iterations 1, 20, 50, and 75 when playing survival.

Iteration Individuals	1	20	50	75
Tetrominos Placed	24.7	35.6	36.3	36.8

Back-to-back weights can be seen increasing over iterations, starting from a low negative value in iteration 1 to the maximum weight by the final iteration. This indicates growing emphasis on performing back-to-back or higher efficiency moves.

The evolution reflects the learning process where the model prioritises features most aligned with its objectives, adjusting dynamically based on intermediate feedback. The trends suggest a strategic shift from general stability (e.g., low pile height and hole minimization) to advanced tactics (e.g., maintaining back-to-back sequences and minimizing column transitions). The oscillation in weights during earlier iterations highlights an exploratory phase before settling on optimised priorities.

The weights fluctuating between negative and positive values are also due to the fact that win-rate is considered for fitness. Preliminary observations show that there are two classes of players:

1. **Aggressive:** They try to top out an opponent as quickly as possible.
2. **Defensive:** They try to survive as long as possible.

Sometimes in a generation, aggressive players have more wins, other times, defensive players do. This is why we can see the best fit individuals can have vastly different weights across generations.

Although we found that these two classes of players exist, we are still unable to predict the class of player from the weight vectors alone.

4.3 Evaluating Final Weight Vector

In this section, we run the best individuals in iterations 1, 20, 50, and 75 through the two evaluation methods mentioned in Section 3.4. Each individual was given the same seeds to ensure fairness. All individuals were evaluated ten times with different seeds, and their averages calculated and summarised in Tables 4.2 and 4.3.

The attack efficiency increases significantly from iteration 1 to iteration 20, suggesting that early in the evolutionary process, individuals evolve to improve offensive capabilities, prioritizing efficient generation of attacks. At iteration 50, we see the attack efficiency begin to drop dramatically. This trend indicates a strategic shift in the optimisation process: from a focus on attack efficiency to a more balanced approach emphasising survival and defensive strategies.

The steady increase in Tetromino placements indicates consistent improvement in survival skills over iterations. As the algorithm evolves, individuals are better able to clear lines and avoid topping out.

We think that what's happening is that best-performing players are simply outliving their peers. They may not be more efficient at attacking but are able to survive near-loss experiences better.

To play with any of the bots mentioned, users can input their respective weight vector into OpenTris and play with them directly.

5 Conclusion

This capstone project demonstrated the application of genetic algorithms to the challenging domain of Versus Tetris, a competitive multiplayer variant of the classic game. By developing a simulation environment and leveraging a genetic algorithm to optimize decision-making, the project effectively tackled the complexities of balancing attack efficiency and survivability in a dynamic, real-time setting. The evaluation metrics showed that the algorithm successfully achieved a nuanced balance between aggressive and defensive strategies, while adapting to the unpredictable nature of gameplay.

5.1 Limitations

Despite these successes, some limitations remain. The bot's inability to fully leverage the hold mechanic and its exclusion of lookahead beyond the immediate piece restricted its strategic depth. Additionally, while effective in most scenarios, the algorithm struggled with prolonged decision-making in edge cases, highlighting potential areas for improvement.

5.2 Future Work

Future work could address these limitations by incorporating advanced features such as multi-piece lookahead, dynamic probability-based decision trees, or hybrid algorithms that blend genetic optimization with reinforcement learning techniques. Expanding the scope of research to include alternative nature-inspired methods like ant colony optimization or particle swarm optimization could also yield valuable insights. Finally, transitioning the simulation to a high-speed, real-time implementation would further test the robustness and scalability of the approach in competitive environments.

By bridging the gap between computational complexity and practical gaming AI, this project paves the way for future explorations in optimizing not just Tetris but other NP-complete problems that demand adaptability and efficient solutions.

6 References

[1] Tetris Inc. *About Tetris*. https://tetris.com/about-us. [accessed Apr. 22, 2024].

[2] Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. "Tetris is Hard, Even to Approximate". In: *Computing and Combinatorics*. Ed. by Tandy Warnow and Binhai Zhu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 351–363. ISBN: 978-3-540-45071-9.

[3] John Brzustowski. "Can you win at Tetris?" Master's Thesis. 200 University Ave W, Waterloo, ON N2L 3G1, Canada: University of Waterloo, 1988.

[4] Heidi Burgiel. "How to lose at Tetris". In: *The Mathematical Gazette* 81.491 (1997), pp. 194–200. DOI: 10.2307/3619195.

[5] Michael Sipser. In: *Introduction to the Theory of Computation*. Cengage Learning, 2013.

[6] Veronika Lesch et al. "A Case Study of Vehicle Route Optimization". In: *CoRR* abs/2111.09087 (2021).

[7] Jeffrey D. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System sciences* 10.3 (1975), pp. 384–393.

[8] Jeff Arle and Kristen Carlson. "Medical diagnosis and treatment is NP-complete". In: *Journal of Experimental & Theoretical Artificial Intelligence* 33 (Mar. 2020), pp. 1–16. DOI: 10.1080/0952813X.2020.1737581.

[9] Lawrence Davis. "Job Shop Scheduling with Genetic Algorithms". In: *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. 1985, pp. 136–140.

[10] Wael Korani and Malek Mouhoub. "Review on Nature-Inspired Algorithms". In: *Operations Research Forum* 2 (July 2021). DOI: 10.1007/s43069-021-00068-x.

[11] Jason Lewis. "Playing Tetris with Genetic Algorithms". In: 2015. URL: https://api.semanticscholar.org/CorpusID:17416568.

[12] Leo Langenhoven, Willem S. van Heerden, and Andries P. Engelbrecht. "Swarm Tetris: Applying particle swarm optimization to tetris". In: *IEEE Congress on Evolutionary Computation*. 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586033.

[13] Sualeh Asif et al. "Tetris is NP-hard even with O(1) Rows or Columns". In: *Journal of Information Processing* 28 (Jan. 2020), pp. 942–958. DOI: 10.2197/ipsjjip.28.942.

[14] Oded Goldreich. "Computational complexity: a conceptual perspective". In: *SIGACT News* 39.3 (Sept. 2008). ISSN: 0163-5700. DOI: 10.1145/1412700.1412710. URL: https://doi.org/10.1145/1412700.1412710.

[15] Daniel P Bovet, Pierluigi Crescenzi, and Riccardo Silvestri. "A uniform approach to define complexity classes". In: *Theoretical Computer Science* 104.2 (1992), pp. 263–283.

[16] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. ISBN: 9780521424264.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. First Edition. W. H. Freeman, 1979. ISBN: 0716710455.

[18] MIT OpenCourseWare. *NP-Completeness*. 6.046J: Design and Analysis on Algorithms, Massachusetts Institute of Technology. 2015. URL: https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/resources/mit6_046js15_lec16/.

[19] *The Millennium Prize Problems - Clay Mathematics Institute — claymath.org*. https://www.claymath.org/millennium-problems/. [Accessed 15-05-2024].

[20] Dapeng Zhang, Zhongjie Cai, and Bernhard Nebel. "PLAYING TETRIS USING LEARNING BY IMITATION". In: 2010. URL: https://api.semanticscholar.org/CorpusID:10668262.

[21] Ahmed Hussein et al. "Imitation Learning: A Survey of Learning Methods". English. In: *ACM Computing Surveys* 50.2 (Apr. 2017). ISSN: 0360-0300. DOI: 10.1145/3071073.

[22] Colin Fahey. *Tetris AI*. [accessed Jun. 23, 2024]. 2003. URL: http://www.colinfahey.com/tetris/.

[23] Guillaume Maurice Jean-Bernard Chaslot Chaslot. "Monte-Carlo Tree Search". English. PhD thesis. Maastricht University, 2010. ISBN: 9789085590996. DOI: 10.26481/dis.20100930gc.

[24] Zhongjie Cai, Dapeng Zhang, and Bernhard Nebel. "Playing Tetris Using Bandit-Based Monte-Carlo Planning". In: Jan. 2011.

[25] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-46056-5.

[26] Matt Stevens and Sabeek Pradhan. "Playing tetris with deep reinforcement learning". In: *Convolutional Neural Networks for Visual Recognition CS23, Stanford Univ., Stanford, CA, USA, Tech. Rep* (2016).

[27] Yiyuan Lee. *Tetris AI - The (Near) Perfect Bot*. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/. [accessed Jul. 17, 2024].

[28] Ziao Chen. "Playing Tetris with deep reinforcement learning". Master's Thesis. University of Illinois at Urbana-Champaign, 2021.

[29] Xin-She Yang. *Nature-inspired optimization algorithms*. Academic Press, 2020.

[30] Saman Almufti et al. "Overview of Metaheuristic Algorithms". In: *Polaris Global Journal of Scholarly Research and Trends* 2 (Apr. 2023), pp. 10–32. DOI: 10.58429/pgjsrt.v2n2a144.

[31] Victor II M Romero, Leonel L Tomes, and John Paul T Yusiong. "Tetris agent optimization using harmony search algorithm". In: *International Journal of Computer Science Issues (IJCSI)* 8.1 (2011), p. 22.

[32] Zong Woo Geem, Joong Hoon Kim, and Gobichettipalayam Vasudevan Loganathan. "A new heuristic optimization algorithm: harmony search". In: *simulation* 76.2 (2001), pp. 60–68.

[33] Houssem Bouchekara. "Most Valuable Player Algorithm: a novel optimization algorithm inspired from sport". In: *Operational Research* 20 (Mar. 2020), pp. 1–57. DOI: 10.1007/s12351-017-0320-y.

[34] Hendrawan Armanto, Ronal Dwi Putra, and C Pickerling. "MVPA and GA Comparison for State Space Optimization at Classic Tetris Game Agent Problem". In: *Inform: Jurnal Ilmiah Bidang Teknologi Informasi dan Komunikasi* 7.1 (2022), pp. 73–80.

[35] R. Arya et al. *Nature-inspired Optimization Algorithms and Soft Computing: Methods, Technology and Applications for IoTs, Smart Cities, Healthcare and Industrial Automation*. Computing and Networks. Institution of Engineering and Technology, 2023. ISBN: 9781839535161. URL: https://books.google.com.my/books?id=lTjhEAAAQBAJ.

[36] Landon Flom and Cliff Robinson. "Using a Genetic Algorithm to Weight an Evaluation Function for Tetris". In: 2005. URL: https://api.semanticscholar.org/CorpusID:18453509.

[37] Darrell Whitley and Timothy Starkweather. "Genitor II: A distributed genetic algorithm". In: *Journal of Experimental & Theoretical Artificial Intelligence* 2.3 (1990), pp. 189–214.

[38] Gilbert Syswerda. "Uniform Crossover in Genetic Algorithms". In: Jan. 1989.

[39] Xingguo Chen et al. "Apply Ant Colony Optimization to Tetris". In: July 2009, pp. 1741–1742. DOI: 10.1145/1569901.1570136.

[40] *Hard Drop - Tetris Community*. https://harddrop.com/. [accessed Jul. 27, 2024].