UNIVERSITY OF HERTFORDSHIRE

Department of Computer Science

Modular BSc Honours in Computer Science

6COM1054 - Artificial Intelligence Project

Final Report

April 2023

Investigating Genetic Algorithms for Playing Tetris

A V Gavrilov

20010918

Supervised by: Paul Moggridge

Table of Contents

Abstract

The goal of this project is to develop an intelligent agent for playing the well-known board game Tetris, by utilising genetic algorithms and heuristic evaluation functions to optimise the AI player's performance. Within the report, the process of development is explained deeply in each chapter, including the initial implementation of a Tetris game in Python using the Pygame library, which was created with the aid of the online tutorial and stages of integration of Artificial Intelligence.

The report demonstrates that by teaching the agent to consider various strategies and factors, it will be able to make better decisions about where to position the game pieces. An approach inspired by natural evolution is used to fine-tune these choices, allowing the AI to learn and improve its performance over time.

Upon completion of the final product, the project effort will be subject to honest and constructive critique to evaluate its success. The assessment will be based on the degree to which the Tetris AI project has met its initial desired outcomes. In a reflective conclusion, a full discussion of the findings, as well as the information and insights obtained throughout the development process, will be included.

Acknowledgements

I express my heartfelt gratitude to Paul Moggridge, my supervisor during this project. His encouragement, professional insights, and enthusiasm greatly contributed to its success. Paul's constructive feedback assisted me in rectifying mistakes and improving my work. Our meetings, which were both entertaining and productive, aided in advancement and growth. I am truly fortunate to have had such a dedicated and knowledgeable mentor.

1.0 Introduction

The objective of this project is to develop Artificial Intelligence for the classic puzzle game of Tetris using genetic algorithms. The trained AI will control gameplay, and the emphasis will be on examining the efficacy of agent evolution strategies in such complicated tasks and their ability to outperform human performance benchmarks.

In the upcoming chapters of this report, they will delve into the rich history of Tetris, its rules, and the process of recreating the game by utilising an online tutorial as a foundation with the implementation of additional adjustments to the code to tailor it to the specific needs of this project.

Following the completion of the game's development, an in-depth evaluation of the code was conducted to identify opportunities for integrating initial AI agents. This process involved the creation of simple bots to explore interactions with the Tetris grid and various game elements. The results of these explorations proved instrumental in fostering a deeper understanding of game mechanics and set the groundwork for the subsequent development of genetic algorithms.

Ultimately, this report will present the implementation process of genetic algorithms, elucidating the rationale for their employment in the project. It will address conflicting objectives, such as simultaneously clearing blocks while minimising the formation of holes and pile-ups in the Tetris structure. Therefore, the AI agent will be granted access to information about the game via specific grid parameters, including the height of the structure, bumpiness, and the number of holes created, in order to improve the precision of its decision-making process. The primary focus will involve adjusting these parameters through the process of natural evolution to achieve targeted objectives, such as maximising the number of cleared lines and accelerating block clearance. Upon completion of the project, the AI agent's performance and evolutionary progress were rigorously evaluated.

The end product allows users to play Tetris independently, observe the training process of genetic algorithms through generations of agents, and experience the final, trained AI agent in action.

Python was chosen as the primary programming language because of its extensive range of libraries and frameworks, including the Pygame library, which was used to develop the interface of Tetris.

The appendices section of the report will include the full source code.

2.0 Literature Review and Research

Extensive efforts and thorough studies have been undertaken to establish a comprehensive understanding of the project's development. Conducted literature review summarised previous research on a related topic, identifying essential concepts, theories, and findings to establish the foundation for a new research endeavour. In this case, the research was concentrated on the development of Artificial Intelligence for the classic puzzle game of Tetris, utilising genetic algorithms.

To get started with adding AI to Tetris, it was required to first understand the game's rules and mechanics. The scholarly study "Tetris is hard, even to approximate" (Demaine, 2003) provided a detailed description of Tetris, as well as conducted research on the game's computational complexity. This document served as a significant resource for the project's initial Tetris implementation and aided in building initial ideas for the future development of AI.

Whereas game creation was not the major focus of this project, it was a necessary requirement for generating finished artefact. Therefore, the further study included research in the field of the development of Tetris in Python, which resulted in the discovery of a useful online tutorial (Techwithtim, 2022) that served as the guide for the game's implementation. This tutorial provided step-by-step instructions with extensive explanations, enabling comprehension of core programming methods and the Pygame library. It allowed me to efficiently build Tetris in Python and customise the code to meet the requirements of the AI agent development process by leveraging the tutorial's insights.

There was also research on the usage of Artificial Intelligence in games. (Togelius, 2010) presented several ways of AI implementation in various game elements in his work, which provided me with a great insight into the context of AI development in games, setting the scene for the application of AI in Tetris. Another great resource, "Artificial Intelligence: a Modern Approach" (Russel, 2015) covered a wide range of Artificial Intelligence principles, which considerably contributed to the comprehension of the deployment of intelligent agents in the project by using various approaches. This paper helped to make knowledgeable decisions throughout the stage of implementation of genetic algorithms.

The research progressed in the direction of the employment of Genetic Algorithms since it was the primary way for teaching AI to play Tetris in the project. The valuable source which also served as an initial inspiration for this project was a YouTube video in which a programmer demonstrated the development of the Tetris AI with the help of genetic algorithms (CodeBullet, 2020). This visual presentation not only highlighted the potential and effectiveness of the process of natural selection and evolution that resulted in a functional AI agent but also offered knowledge of the practical aspects of the implementation of such techniques. There was also a wide range of academic papers focused on the application of GAs (genetic algorithms) in developing intelligent agents for Tetris. The main purpose of genetic algorithms in the project was to configure the weights of specific parameters of the game environment that the agent had access to, and more in-depth information on this process will be provided in the subsequent chapters of this report, however, the set of these parameters must have been established first. By

identifying the most significant aspects of the Tetris grid to consider, (da Silva, 2017) and (Lewis, 2015) gave a valuable solution to this challenge. Understanding the importance of each of the parameters was crucial knowledge that helped me to develop an effective a robust Tetris AI.

The research in the area of usage of Genetic Algorithms in Tetris was deemed the most important of the external study since it supplied valuable insights into the complexity of combining Tetris gameplay with genetic algorithms, which ultimately informed the design of the AI in the current project.

In conclusion, this literature review examined studies on AI implementation in games, and how the genetic algorithms have been integrated into Tetris. It exhibited the methods of AI development in the game environment and proved the efficiency of optimising agent performance by applying the process of natural evolution. This whole knowledge aided to set the stage for the project of creating my own approach for implementing Artificial Intelligence in Tetris.

The following chapters of the report will discuss how the information gathered from this literature review was applied to the project, including a detailed explanation of the development process of Tetris and the integration of genetic algorithms.

3.0 Introduction to Tetris

3.1 History of the Game

Tetris is a famous video game created by Soviet mathematician Alexey Pazhitnov on June 6, 1984 (Prisco, 2019). It swiftly became a popular cultural symbol, sparking copyright fights during the closing years of the Cold War (Temple, 2004).

Figure 1 Gameboy Tetris

3.2 Rules

The game is played on a grid of 20 rows and 10 columns, that starts empty and consequently fills up with falling pieces. These pieces are called “Tetrominoes” and can be 7 distinct shapes comprised of 4 blocks.

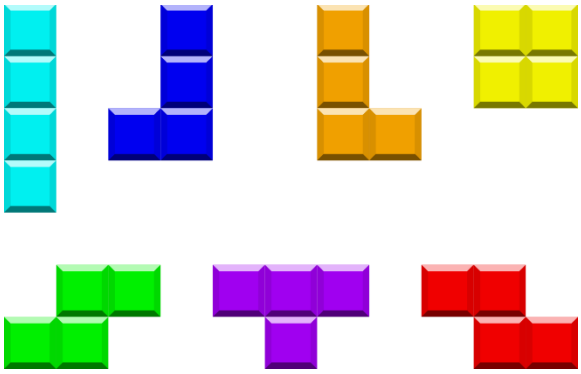


Figure 2 Seven Tetrominoes

The pieces fall from the top one at a time, until they collide with the other tetromino or the grid floor. The main objective is to manipulate these falling pieces by moving and rotating them, with the aim of fitting them into a structure of other Tetrominoes to form complete lines across a grid. When the line is created, it disappears, freeing up space on the grid and awarding points to the player. As the player progresses through the game, the pace of the falling increases gradually, demanding quicker reaction and decision-making. When there is no room on the top of a grid for the next piece, the game ends. Therefore, players must strategically position the pieces to reduce gaps and excessive block heaps in a grid in order to complete as many lines as possible and accumulate the maximum score before the inevitable conclusion.

3.3 Scoring System

Tetris' scoring system has changed throughout its history, with different implementations and versions of the game providing new techniques to calculate scores. Original Tetris featured a relatively simple scoring system in which players received points for each line cleared. As Tetris grew in popularity and was adapted for other platforms, the scoring system was changed to encourage more skilful gameplay and include a stronger element of strategy.

One of the most well-known scoring systems was implemented in the Nintendo Entertainment System (NES) version of Tetris (Tetris Wiki, 2020).

This version awarded points depending on the number of cleared lines:

- Single line clear: 40 points
- Double line clear: 100 points
- Triple line clear: 300 points
- Four line clear (Tetris): 1200 points

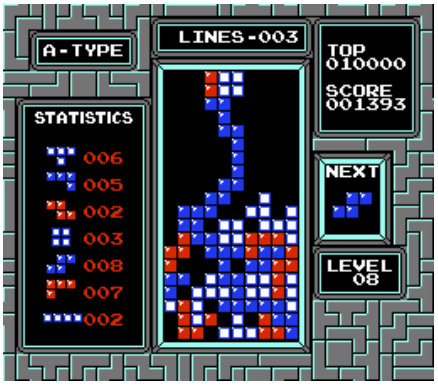


Figure 3 NES version of Tetris

3.4 Tetris from an AI Perspective

Notwithstanding the simple mechanics, Tetris is still considered a complex game. Therefore, it presents a great challenge for AI developers. The need for spatial reasoning makes it an ideal playground for exploring the capabilities of AI algorithms.

To summarise, Tetris is a monument to the power of simplicity in game design, capturing players for nearly four decades with its accessible yet complex gaming principles. Tetris is a great subject for the research of artificial intelligence techniques and their prospective applications in complex problem-solving tasks due to its persistent appeal and the cognitive abilities required.

The next chapter will demonstrate the step-by-step process of recreating this popular game with the help of the aforementioned tutorial.

4.0 Game Implementation

This section delves into the Tetris game's implementation using Python and the Pygame library. The development process, which was supervised by an online tutorial, includes the design of the interface, mechanics, and essential functionality. Custom modifications specific to the needs of this project will also be described. However, this section will only explain the key code from the tutorial and the full code will be provided in the Appendices section of this report.

4.1 Following a Tutorial

The Tetris tutorial from TechWithTim was composed of a series of videos that broke down the development process into simple sections. Each video focused on a different part of the game, making it simple to follow.

Figure 4 Online tutorial on Tetris

4.2 Setting up the Environment

The initial step of a tutorial required the preparation of the development environment. This step included:

- Installation of the Pygame library
- Creating the Python file for the game

The IDE was chosen to be the PyCharm. This code editor is robust and purpose-built for Python programming, making it a good choice for the project.

4.3 Layout of Tetris

The beginning of tutorial code started with defining several global variables that determined the layout of the game window and play area.

- **"s_width"** and **"s_height"** variables defined the width and the height of the game window, which were 800 and 700 pixels respectively.
- **"play_width"** and **"play_height"** variables were set to 300 and 600 pixels and represented the size of the play area, where the Tetris pieces and grid appear.
- **"block_size"** defined the size of a single block in the play area.
- **"top_left_x"** and **"top_left_y"** variables calculated the top-left coordinates of the play area within the game window. These variables were essential for the proper alignment of the game area.

```
# GLOBALS VARS
s_width = 800
s_height = 700
play_width = 300  # meaning 300 // 10 = 30 width per block
play_height = 600  # meaning 600 // 20 = 30 height per block
block_size = 30

top_left_x = (s_width - play_width) // 2
top_left_y = s_height - play_height
```

Figure 5 Global variables

4.4 Creating Tetromino shapes with rotations

The tutorial provided a starter file that included the initial setup for the tetromino pieces with their rotations.

```
S = [['.....',
      '......',
      '..00..',
      '.00...',
      '.....'],
     ['.....',
      '..0..',
      '..00.',
      '...0.',
      '.....']]
```

Figure 6 "S" shape

There were a total of seven unique shapes - S, Z, I, O, J, L, and T, which were represented as lists of matrices. Each shape was stored as a list of different rotations and consisted of dots and zeros, where dots represented an empty space and zeros denoted a block of the shape.

As shown in Figure 5, the shape "S" has two possible rotations. Similarly, other shapes were also represented in this format.

Then all the shapes were stored in a list called **"shapes".** After, a list called **"shape_colors"** was defined, which contained the RGB values for each corresponding shape. Each colour in a list was stored as a tuple of three integers ranging from 0 to 255.

```
shapes = [S, Z, I, O, J, L, T]
shape_colors = [(0, 255, 0), (255, 0, 0), (0, 255, 255), (255, 255, 0), (255, 165, 0), (0, 0, 255), (128, 0, 128)]
```

Figure 7 Shapes and Shape_Colors lists

4.5 Class Piece

The following step was on creating a piece class.

The class had the values of its position represented by x and y coordinates on a grid. The shape and colour values were also passed to the class as well as the rotation of a piece.

```
class Piece(object):
    def __init__(self, x, y, shape):
        self.x = x
        self.y = y
        self.shape = shape
        self.color = shape_colors[shapes.index(shape)]
        self.rotation = 0
```

Figure 8 Class Piece

4.6 Creating a Grid

The focus of the next step was on establishing the fundamental structure of the Tetris game, which was a grid. It presented a multidimensional list that contains 20 lists of 10 elements which were represented as rows and columns. Each element of a list was a tuple, that has an RGB value of that current position.

```
def create_grid(locked_positions={}):
    grid = [[(0, 0, 0) for x in range(10)] for x in range(20)]  # intialise black color for each grid position

    for i in range(len(grid)):
        for j in range(len(grid[i])):
            if (j, i) in locked_positions:
                c = locked_positions[(j, i)]
                grid[i][j] = c
    return grid
```

Figure 9 Function create_grid

The function also receives a parameter **"locked_positions"** which is a dictionary containing the coordinates of occupied positions on a grid. The function iterates through this dictionary to update the grid with the colours of tetrominoes.

4.7 First code adjustment (7-bag-randomiser)

The next phase of the tutorial involved the creation of a "**get_shape"** function, which originally returned a random tetromino piece with the help of the function **"random.choice"**. However, I decided to modify this function and implemented a more balanced approach to generate pieces. This approach is known as a "7-bag randomiser" or just a "bug" feature.

The bag feature works by generating a bag that contains all seven tetromino shapes. When a new piece is required, one is chosen at random from the bag. The bag is then refilled again, and the process is repeated. This guarantees that each tetromino piece is given more fairly to the player, reducing long streaks of the same piece or excessive waiting times for a certain piece (TetrisWiki, 2020).

I also decided to add this feature, because I thought it could be beneficial for the future AI agent. Since this approach reduces the element of luck and allows for more strategic gameplay.

```
def create_bag():
    global bag
    bag = shapes.copy()  # get copy of a shapes list
    random.shuffle(bag)  # rearrange pieces in the bag


def get_next_tetromino():
    global bag
    if not bag:
        create_bag()
    return bag.pop()  # pop the piece from a bag


def update_pieces():
    global current_piece, next_piece
    current_piece = next_piece
    next_piece = get_next_tetromino()


# GET TETRIS PIECE WITH RANDOM SHAPE
def get_shape(shape):
    return Piece(5, 0, shape)
```

Figure 10 7 bag feature code

Figure 10 explains the code of this functionality. This feature was created with the help of 4 helper functions:

The **"create_bag"** function creates the copy of the current list containing the shapes of all 7 pieces. After it randomly rearranges the order of pieces in the bag.

The **"get_next_tetromino"** function checks if the bag is empty, if it is not, it draws the piece from a bag one by one, until it is empty.

The **"update_pieces"** function gets the first two pieces from a bag a put them in the variable of the current and next pieces. More of this functionality will be discussed further in the report.

The **"get_shape"** function simply returns the Piece object, where the "shape" parameter will be passed as a value from the **"get_next_tetromino"** function.

4.8 Drawing a Grid

The function of this code simply draws the grey lines of a grid for the visibility of squares in the structure.

```
def draw_grid(surface, grid):
    # coordinates
    sx = top_left_x
    sy = top_left_y

    # DRAW LINES FOR A GRID
    for i in range(len(grid)):
        pygame.draw.line(surface, (128, 128, 128), (sx, sy + i * block_size), (sx + play_width, sy + i * block_size))
        for j in range(len(grid[i])):
            pygame.draw.line(surface, (128, 128, 128), (sx + j * block_size, sy),
                             (sx + j * block_size, sy + play_height))
```

Figure 11 draw_grid function

To begin drawing the lines, the function obtains the coordinates of the global variables. Then it iterates through the grid rows and columns, drawing a line each time.

4.9 Convert Shape Format

Since each piece is represented as a multidimensional list, the tutorial code provided a function that transfers this list in a form that the computer can understand.

```
def convert_shape_format(shape):
    positions = []# list of positions
    # get a sublist of the shape depending on its rotation
    format = shape.shape[shape.rotation % len(shape.shape)]

    # iterate through the shape and get positions for each block
    for i, line in enumerate(format):
        row = list(line)
        for j, column in enumerate(row):
            if column == '0':
                positions.append((shape.x + j, shape.y + i))
    for i, pos in enumerate(positions):
        positions[i] = (pos[0] - 2, pos[1] - 4)

    return positions
```

Mainly the purpose of this function is to get the positions of each block of a tetromino piece.

It returns the list containing four tuples of x and y coordinates of each block.

Figure 12 convert_shape_format function

4.9 Determine a Valid Space

The next stage of the tutorial was to ensure that the piece could only move in a valid space. Figure 13 shows a function that examines the grid to confirm that the current place is not occupied for further movement.

```
def valid_space(shape, grid):

    # check for black color on a grid and create list of accepted positions
    accepted_pos = [[(j, i) for j in range(10) if grid[i][j] == (0, 0, 0)] for i in range(20)]
    accepted_pos = [j for sub in accepted_pos for j in sub]

    # format the shape of a current piece
    formatted = convert_shape_format(shape)

    # check if piece is in an accepted_pos
    for pos in formatted:
        if pos not in accepted_pos:
            if pos[1] > -1:
                return False
    return True
```

Figure 13 valid_space function

The function receives two parameters: shape and grid. The function loops through the grid to see if the position the pieces are trying to be moved to has a colour other than black, which is represented as (0, 0, 0). If the position has colour, it means that the position is occupied, otherwise, it is free.

4.10 Check if the Game is Lost

In order to end the game, the function was needed to check if the player has lost the game.

```
def check_lost(positions):
    for pos in positions:
        x, y = pos
        if y < 1:  # if piece above the screen
            return True

    return False
```

The function takes a list of positions as a parameter and checks if any position in the given list is above the screen, or if the y-coordinate is less than 1.

Figure 14 check_lost function

4.11 Clearing the Rows

When all the positions in the row are filled the row should be cleared, and the rows above it must be pushed down. This was done with the help of the function provided in the tutorial **“clear_rows”**.

The function checks whether the row is full by iterating through the grid. The iteration begins at the bottom of the grid and goes all the way up.

The full row is removed from the grid and rows above it are shifted downward.

Since the row is deleted, the function compensates for this by adding one empty row to the top of the grid.

```
def clear_rows(grid, locked):
    inc = 0
    # iterate through grid from bottom
    for i in range(len(grid) - 1, -1, -1):
        row = grid[i]
        # if ther is block
        if (0, 0, 0) not in row:
            inc += 1
            ind = i
            for j in range(len(row)):
                try:
                    del locked[(j, i)]
                except:
                    continue
    # update the locked positions
    if inc > 0:
        clear_row.play()
        for key in sorted(list(locked), key=lambda x: x[1])[::-1]:
            x, y = key
            if y < ind:
                newKey = (x, y + inc)
                locked[newKey] = locked.pop(key)

    return inc
```

Figure 15 clear_rows function

4.12 Draw Next Shape

```
def draw_next_shape(shape, surface):
    font = pygame.font.SysFont('comicsans', 30)
    label = font.render('Next Shape', 1, (255, 255, 255))

    sx = top_left_x + play_width
    sy = top_left_y + play_height / 2 - 100
    format = shape.shape[shape.rotation % len(shape.shape)]

    # iterate through shape
    for i, line in enumerate(format):
        row = list(line)
        for j, column in enumerate(row):
            if column == '0': # if block
                # draw
                pygame.draw.rect(surface, shape.color,
                                 (sx + j * block_size, sy + i * block_size, block_size, block_size), 0)

    surface.blit(label, (sx + 10, sy - 50))
```

Figure 16 draw_next_shape function

This function just draws the next falling tetromino on the play screen's right side. When a zero appears in the list of formatted shapes indicating the presence of the block, the function draws a square according to the tetromino colour.

4.13 Draw Window

The next function in the tutorial draws the window of Tetris. It includes displaying the title, score, and pieces on the game board.

```
def draw_window(surface, grid, score=0):
    surface.fill((0, 0, 0))  # fill surface in black

    # DRAW TITLE
    pygame.font.init()
    font = pygame.font.SysFont('comicsans', 60)
    label = font.render('Tetris', 1, (255, 255, 255))

    # PUT TITLE ON MIDDLE OF A SCREEN
    surface.blit(label, (top_left_x + play_width / 2 - (label.get_width() / 2), 30))

    # draw the blocks on a grid (based on color in grid[i][j])
    for i in range(len(grid)):
        for j in range(len(grid[i])):
            pygame.draw.rect(surface, grid[i][j],
                             (top_left_x + j * block_size, top_left_y + i * block_size, block_size, block_size), 0)

    # DRAW RED BORDERS OF A GRID
    pygame.draw.rect(surface, (255, 0, 0), (top_left_x, top_left_y, play_width, play_height), 4)

    # DRAW SCORE
    font = pygame.font.SysFont('comicsans', 30)
    label = font.render('Score: ' + str(score), 1, (255, 255, 255))
    # CORDS FOR SCORE LABEL
    sx = top_left_x + play_width
    sy = top_left_y + play_height / 2 - 100
    surface.blit(label, (sx + 20, sy + 160))

    draw_grid(surface, grid)
    # pygame.display.update()
```

Figure 17 draw_window function

The function displays the name of the game “Tetris” on top of the screen. Then it iterates through the grid parameter that is passed to the function which contains the RGB values of the piece, then it is passed to the **“pygame.draw.rect”** function and aligns the block based on their coordinates.

The function also displays the score, which functionality will be discussed in the next step.

4.14 The Main Loop

Every game has a main loop, where the functions of the game are running constantly. This loop also checks for any events that occur, such as user input and updating the game variables.

The loop provided in this section highlights the key functionality and the code with all features will be provided at the end of this report in the Appendices.

The game loop is called the **"main"** and starts with defining some variables.

```
create_bag()
def main(win):
    locked_positions = {}
    grid = create_grid(locked_positions)
    change_piece = False
    run = True
    current_piece = get_shape(get_next_tetromino())
    next_piece = get_shape(get_next_tetromino())
    clock = pygame.time.Clock()
    fall_time = 0
    fall_speed = 0.37
    level_time = 0
    rows = 0
    score = 0
```

First, I initialised the bag of tetromino pieces.

After the function defined the dictionary with the locked positions which were initially empty and were passed to the created grid.

The **"change_piece"** is a Boolean variable that defines whether the next piece should appear on the screen.

The **"run"** variables control the state of the game.

Figure 18 Defining variables in the loop

Then the current piece and the next piece were obtained from the bag.

4.15 Game Difficulty

As time goes on the game should become more difficult. This was reached by increasing the falling speed of a piece.

```
while run:
    grid = create_grid(locked_positions)
    fall_time += clock.get_rawtime()
    level_time += clock.get_rawtime()
    clock.tick()


    if level_time / 1000 > 5:
        level_time = 0
        if fall_speed > 0.12:
            fall_speed -= 0.005
```

In the internal loop of the main function. We increase the falling speed every 5 seconds by 0.005s.

Figure 19 Increasing the Difficulty

4.16 The Falling of a Piece

The following section of the code implements the capability for making a tetromino fall.

```
if fall_time / 1000 >= fall_speed:
    fall_time = 0
    current_piece.y += 1
    if not (valid_space(current_piece, grid)) and current_piece.y > 0:
        current_piece.y -= 1
        change_piece = True
```

Figure 20 Falling piece functionality

The falling speed is determined by the falling speed variable, which moves the current location by one in the y-coordinate. In addition, if a piece location is not in the valid space on the grid, the y-coordinate is subtracted by one to compensate for the piece's disposition. This code also sets the variable **"change_piece"** to **True** when the piece collided with the ground or another piece.

4.17 User Input

The player can interact with the game by pressing the arrow keys on the keyboard. Using the **"pygame.even.get"** function, allows for processing the events that occur in the game which in this case are the key presses.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        run = False
        # pygame.display.quit()

    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_ESCAPE:
            run = False
            pygame.mixer.music.stop()
            pygame.display.quit()
        if event.key == pygame.K_LEFT:
            current_piece.x -= 1
            if not (valid_space(current_piece, grid)):
                current_piece.x += 1
        if event.key == pygame.K_RIGHT:
            current_piece.x += 1
            if not (valid_space(current_piece, grid)):
                current_piece.x -= 1
        if event.key == pygame.K_DOWN:
            current_piece.y += 1
            if not (valid_space(current_piece, grid)):
                current_piece.y -= 1
        if event.key == pygame.K_UP:
            current_piece.rotation += 1
```

Figure 21 User Input functionality code

- When the left arrow key (**pygame.K_LEFT**) is pressed, the current tetromino piece moves one unit to the left (**current_piece.x** = 1). The code then uses the **valid_space()** method to determine whether the piece is in a valid space and if it is not, it returns the piece to its original place (**current_piece.x** += 1).
- Same functionality is implemented for the right key press.
- The current piece is rotated when the up arrow key (**pygame.K_UP**) is pushed. If the piece is not in a valid space after rotation, the rotation is undone by decrementing the **current_piece.rotation** value.
- When the **Escape** is pressed the game stops.

4.18 Second code Adjustment (Hard Drop)

I also decided to add the hard drop function to the game in addition to the current controls. The hard drop function allows players to rapidly place the current tetromino piece in its final landing position, which speeds up gameplay and accelerates line clearing. This feature also improves the overall player experience by allowing them to play the game in a faster and more strategic manner.

```
if event.key == pygame.K_SPACE:
    while True:
        current_piece.y += 1
        if not (valid_space(current_piece, grid)):
            current_piece.y -= 1
            break
```

Figure 22 Hard Drop functionality code

This code continuously moves the piece by one position down until it reaches the ground.

4.19 Changing of a Piece

The following function is used to change the piece when the tetromino collides with the ground or other pieces.

The position of each tetromino block is first converted using the **"convert_shape_format"** function.

The colour and coordinates of a dropped component are then added to the locked position dictionary.

```
shape_pos = convert_shape_format(current_piece)
if change_piece:
    for pos in shape_pos:
        p = (pos[0], pos[1])
        locked_positions[p] = current_piece.color
    current_piece = next_piece
    next_piece = get_shape(get_next_tetromino())
    change_piece = False
```

Figure 23 Changing piece functionality

Finally, the piece is changed, and the next piece is formed, also at the end the **change_piece** variable is set to False.

4.20 Third code Adjustment (Scoring System)

The tutorial code included a basic scoring system that awards 10 points for each cleared line. However, I chose to use the NES score system, as described earlier in the previous article. I have also added a feature that shows the number of lines that have been cleared.

```
rows = clear_rows(grid, locked_positions)
clear_lines += rows
if rows == 1:
    score += 40
elif rows == 2:
    score += 100
elif rows == 3:
    score += 300
elif rows == 4:
    score += 1200
else:
    score == 0
```

Figure 24 NES scoring system functionality code

This code is used within the changing piece functionality. The function **clear_rows** is called when the piece is dropped to see if the move removed any lines. Because the **clear_rows** function returns the number of cleared rows, I simply set this value to the **cleared_lines** variable. Then, for the actual score, I used the NES scoring system.

Then I passed the cleared lines as a parameter to the **draw_window,** which displayed them in the game.

```
# DRAW SCORE
font = pygame.font.SysFont('comicsans', 30)
label = font.render('Score: ' + str(score), 1, (255, 255, 255))
label_score = font.render('Lines: ' + str(rows), 1, (255, 255, 255))

# CORDS FOR SCORE LABEL
sx = top_left_x + play_width
sy = top_left_y + play_height / 2 - 100
surface.blit(label, (sx + 20, sy + 160))
surface.blit(label_score, (sx + 20, sy + 190))
```

Figure 25 Drawing cleared lines code

4.21 Main Menu

The final step of the tutorial involved the design of a primary menu so that when the game is first loaded, the user can press a button to begin playing. This main menu will also appear if the user loses the game, allowing them to restart it.

```
def main_menu(win):
    run = True
    while run:
        win.fill((0, 0, 0))
        draw_text_middle(win, 'Press Any Key To Play', 60, (255, 255, 255))
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
            if event.type == pygame.KEYDOWN:
                main(win)

    pygame.display.quit()
```

Figure 26 Main menu code

The code draws a title and, depending on user input, either ends or loads the game.

4.22 Setting up the Window

Finally, the end code creates the Pygame window, assigns it a caption, and invokes the main menu.

```
win = pygame.display.set_mode((s_width, s_height))
pygame.display.set_caption("Tetris")
main_menu(win)  # start game
```

Figure 27 Setting up the windonw code

4.23 Fourth code Adjustment (Adding Music and Sound Effects)

Another part of the game that was improved was the general user experience. To do this, the original Tetris music and sound effects were integrated into the game. I have decided to add this feature not only to offer a more immersive and genuine Tetris experience to the players but also to provide a nostalgic link to the old game.

The music that was added was the Tetris theme, also known as "Korobeiniki". It is a Russian folk song that has been associated with the game since its creation. The catchy melody and fast tempo of the music compliment the dynamic character of Tetris, creating a captivating and entertaining environment for players (Plan-Blasko, 2015).

In addition to the theme song, a sound effect for the clearing lines event was added. This sound provides rapid auditory feedback to players, which improves the entire game experience, and also contributes to the player's sense of accomplishment and happiness (Huang, 2017).

```
pygame.mixer.init()
theme_soundtrack = pygame.mixer.music.load('theme.mp3')
clear_row = pygame.mixer.Sound('clear.mp3')
```

Figure 28 Loading the sound effects

I used Pygame's built-in mixer feature. The files were then imported into variables, and I added functionality to call them appropriately throughout the game.

4.24 Game Testing

The final result was visually appealing and functioned flawlessly when the game development process was completed. The game's improvements, including the bag feature, hard drop functionality, the original Tetris soundtrack, and sound effects for clearing lines, all contributed significantly to an immersive user experience.

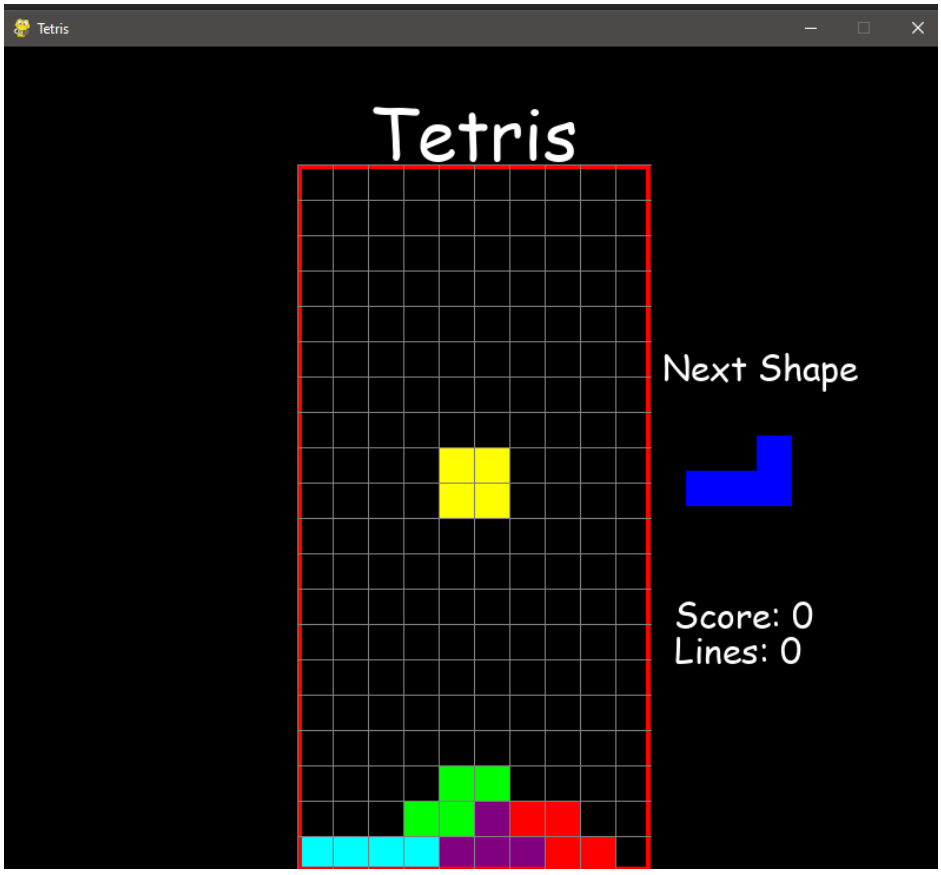


Figure 29 Tetris Game

After, extensive testing was performed by playing the game for a long period of time with no faults or bugs detected.

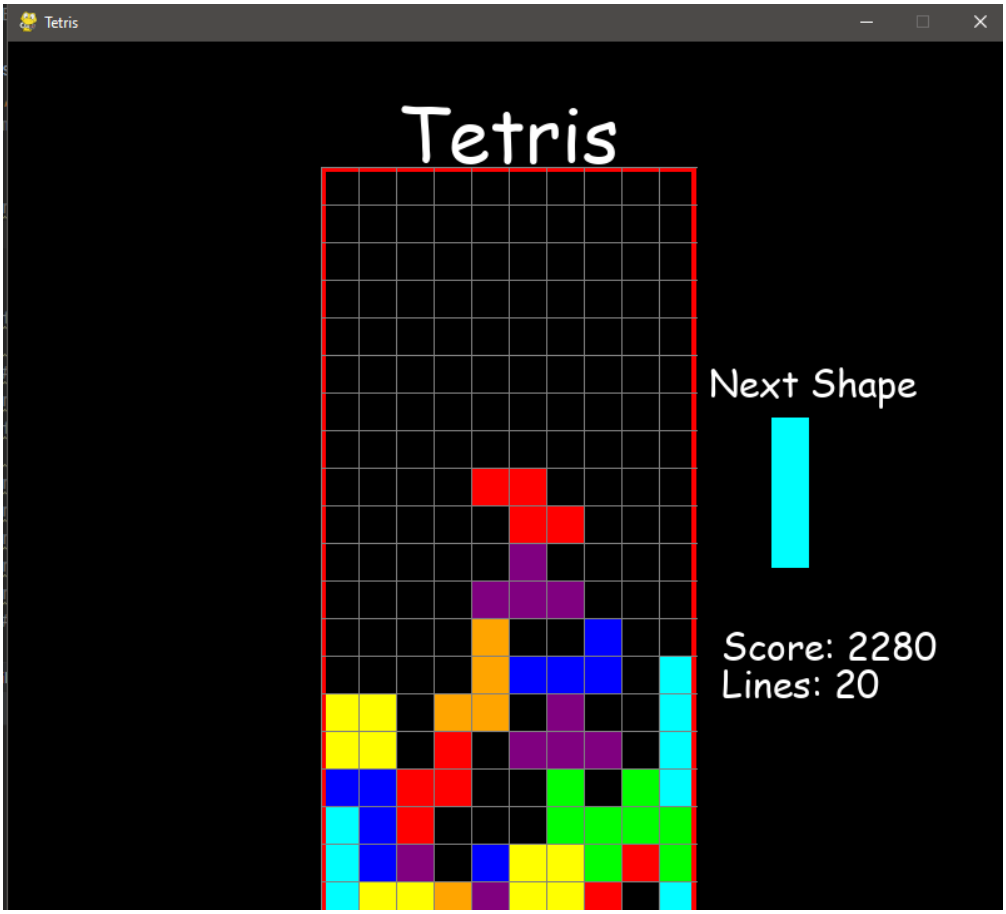


Figure 30 Playing Tetris

4.25 Conclusion

In conclusion, the game development process included several steps, such as setting up the environment, designing the Tetris grid, incorporating game mechanics, and improving the user experience. Additional changes to the code were made such as adding the "7-bag randomiser", hard drop feature, the original Tetris soundtrack, and sound effects. An organised approach was used throughout the development process, with Python and the Pygame library serving as the key tools. The full game's code, which can be obtained in the report's Appendices section, accomplishes the desired purpose of reproducing the classic Tetris experience with extra additions.

The following section of the report will focus on the initial AI analysis and the examination of potential ways for its integration into the game, establishing the framework for the succeeding stages of the project's implementation of genetic algorithms.

5.0 AI Analysis

In this chapter, the report will delve into the initial analysis of AI implementation in the game. Since Tetris presents a diverse set of challenges and complexities, it is crucial to have a comprehensive understanding of the game's various components and goals before AI integration. The primary purpose of this chapter is to investigate the various strategies for constructing Artificial Intelligence capable of achieving excellent performance while playing Tetris.

5.1 Initial Analysis

In Tetris, there are different challenges that players might face, and the AI should be able to handle a variety of aims. The AI can achieve the highest score in the game by clearing as many lines as it could whilst also optimally positioning the tetrominoes to prevent causing holes in the grid or stacking pieces too high. However, some players might prefer to focus on the goal of keeping a stable grid and only clear lines when it is strategically advantageous, therefore the AI should have a set of parameters to take into account to accommodate the scoring component as well as the ability of grid management.

The AI is expected to be able to deal with different tetromino sequences because this is an important aspect of Tetris since the players must adapt to the supplied shapes. Allowing the AI to handle multiple sequence styles will also provide additional flexibility.

5.2 Ways of AI Integration

The most common method for AI implementation in Tetris has been to create a linear evaluation function that assesses each possible tetromino placement to find the location with the highest value (Algorta, 2019). Therefore, this approach was chosen as the main strategy of the initial AI integration.

The critical challenge in developing an effective evaluation function lies in determining the appropriate set of features to consider while calculating the rating of each tetromino position. Several researchers have adopted similar strategies with varying degrees of success in order to achieve optimal results in Tetris.

Early research by Tsitsiklis and Van Roy (1996) focused on a set of two features: the number of holes and the highest column height. This approach resulted in a score of 30 cleared lines. Later, researchers expanded on this foundation and achieved higher scores by incorporating additional features. Specifically, Bertsekas (1996) employed lambda-policy iteration, which considered the height of each column and the difference in height between consecutive columns, also known as "bumpiness" or "roughness" of the grid. This approach led to a significant improvement, with a score of approximately 2,800 cleared lines.

5.3 Set of Features

The choice of features plays a crucial role in determining the performance of the AI agent, as demonstrated by the substantial difference in scores achieved by different feature sets. Therefore, it is essential to carefully select the features to ensure a high-performing and efficient Tetris AI agent.

It was decided to take into account the several set of features for the evaluation function: height of the placed piece, number of holes, bumpiness, and the clearing piece component.

5.3.1 Height of The Placed Piece

The first thing that was considered is the height of the placed piece. A high grid height increases the chances of reaching the top of the grid, resulting in loss. Minimising the height is critical for the AI agent since it allows for more manoeuvrability and, in the long term, the possibility of clearing more lines. By taking height into account as a characteristic, the AI agent can prioritise placements that keep the grid height low, extending the game and boosting the potential score.

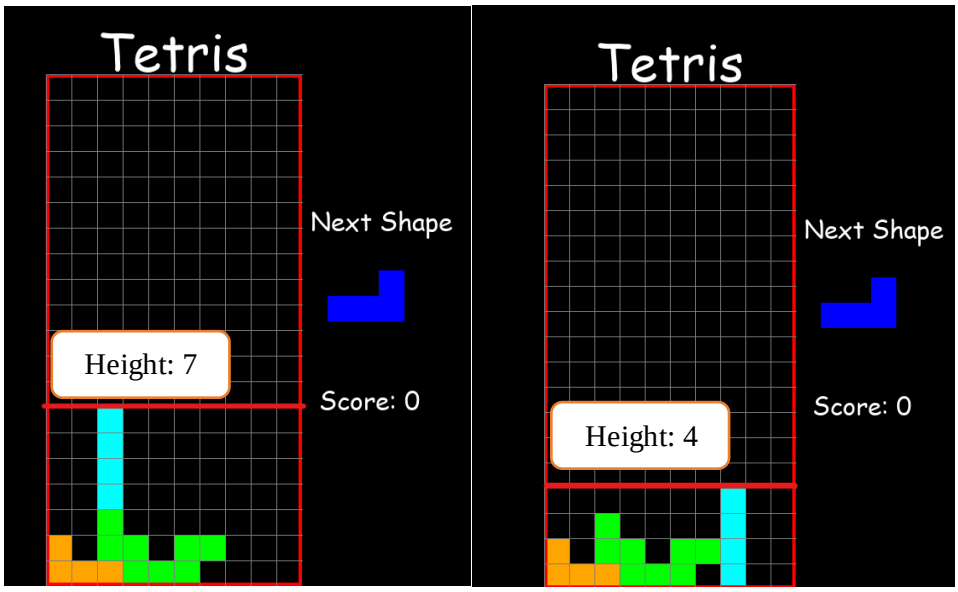


Figure 31 Height parameter

5.3.2 Number of Holes

One of the crucial features that might also considerably affect the performance of the AI agent is the "hole" parameter. A hole in the grid is defined as an empty cell that is covered by at least one full cell in the same column. The presence of holes can limit the AI's capacity to clear lines and, as a result, lower the overall score.

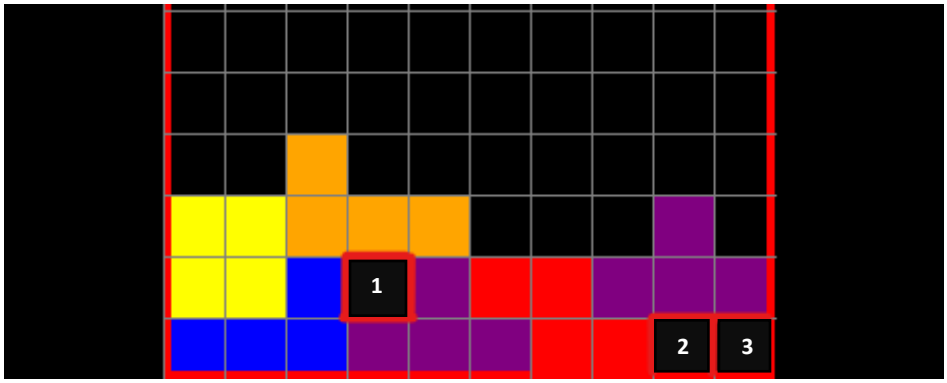


Figure 32 Number of holes parameter

Minimising the number of holes in Tetris is a critical goal for the AI agent. A higher number of holes usually implies poor tetromino placement, which makes clearing lines more difficult and increases the likelihood of reaching the top of the grid. The AI agent can prioritise placements that minimise the production of holes by including the number of holes as a feature in the evaluation process, resulting in more efficient gameplay and a higher score.

5.3.3 Bumpiness

The "bumpiness" parameter is another important aspect that might have a substantial impact on an agent's performance. Bumpiness gauges the grid's general irregularity by comparing the height differences between consecutive columns. A high bumpiness rating denotes an uneven grid, whereas a low one denotes a more uniform grid.

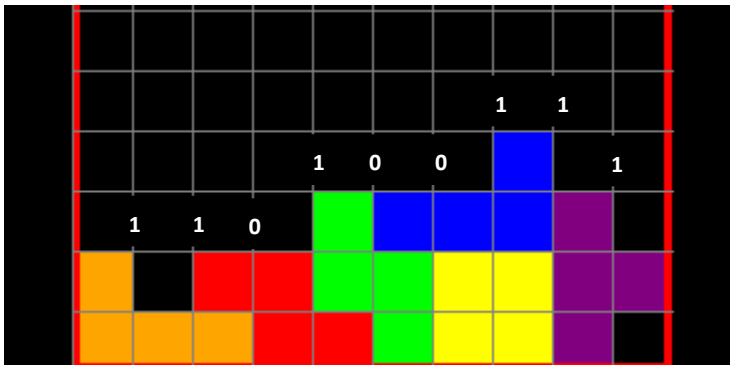


Figure 33 Bumpiness parameter

Maintaining minimal bumpiness is important because it allows for more efficient tetromino placement and easier line clearance. A grid with low bumpiness gives the AI agent more opportunities to position tetrominoes successfully, minimising the likelihood of holes appearing and enhancing the grid's irregularity.

5.3.4 Clearing Piece

The "clearing piece component" is another important characteristic to consider because it directly ties to the game's fundamental goal: clearing lines. This parameter assesses the likelihood of a tetromino placement clearing one or more lines upon landing on the grid. By including this element in the assessment mechanism, the AI agent can prioritise placements that result in immediate line clearances, which also contributes to higher points and longer gaming.

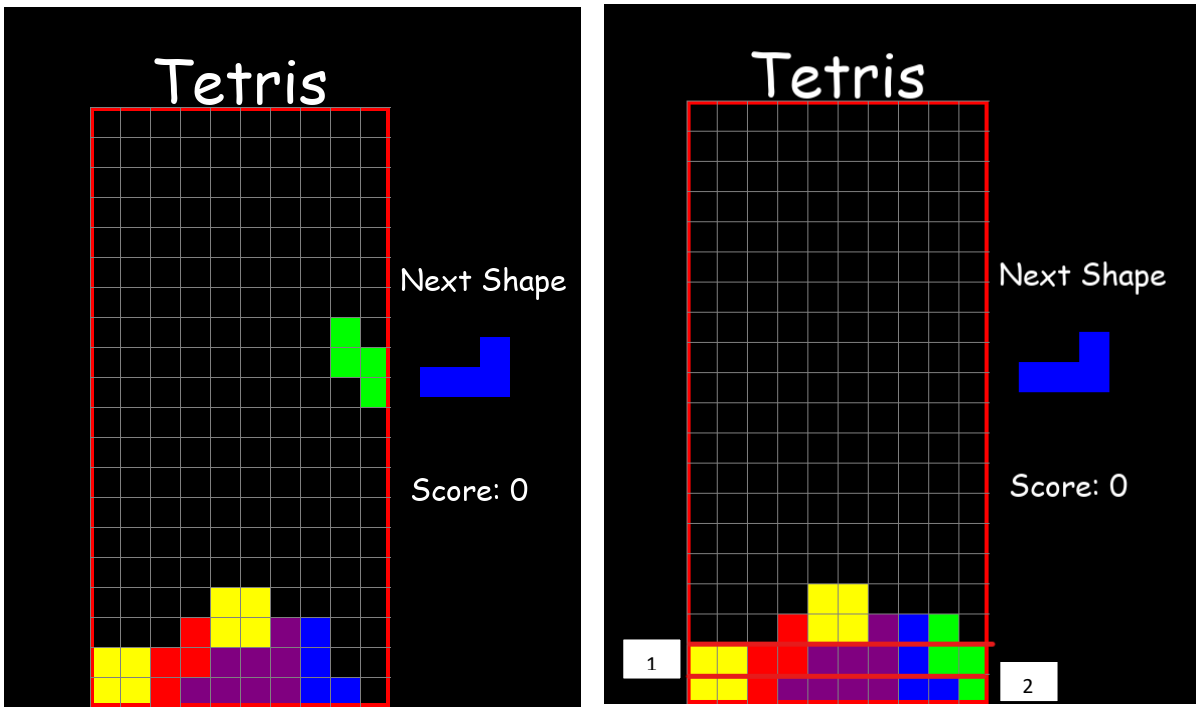


Figure 34 Clearing Piece parameter

5.4 Putting Heuristics Together

Now when the parameters are established, they can be used in a combination to be put in the evaluation function to determine the rating of the position of the dropped piece. It is given by the following formula:

$$w_1 * Height + w_2 * Number\ of\ Holes + w_3 * Bumpiness + w_4 * Clearing\ Piece$$

Where $w_1, w_2, w_3,$ *and* w_4 are the weights for these parameters. With the right combination of these weights, the AI agent can make well-informed decisions on where to place tetrominoes optimally on the grid.

5.5 Hand-Crafted Agents

Previously, human experts directly assigned the weights of the evaluation function (Fahey, 2003). This method used human logic to prioritise parameter values in a specified order. However, this approach has limits because the chosen weights might or might not produce optimal outcomes, and determining the best values for them can be a time-consuming effort. The challenge is that human intuition and skill may not always result in the best potential weights, necessitating the exploration of alternative methodologies for calculating these values.

5.6 Genetic Algorithms

The best possible solution, in this case, is to brute force the values of the weights with the help of Genetic Algorithms. Through a search process inspired by natural selection and evolution, this approach can automatically determine optimal weights for the evaluation function (Flom, 2005). The use of Genetic Algorithms reduces the requirement for human intuition and knowledge in assigning weights, allowing for a more effective and efficient search for parameter values.

However, before implementing Genetic Algorithms, several key components of this approach should be established first.

5.6.1 Individual

The individual offers a possible solution to the problem at hand. Each individual is defined by a set of genes, which are the exact parameter values (weights) that correlate to the selected features, such as height, number of holes, bumpiness, and clearing line piece component.

$$Individual = (w_1, w_2, w_3, w_4)$$

5.6.2 Generation

The generation is the collection of individuals. When initialising the generation, a diverse range of individuals with varying gene values are generated in order to explore different portions of the search space. This diversity can be achieved by assigning random values within a predefined range to the weights.

5.6.3 Fitness Function

The fitness function evaluates and scores each individual in the population based on their performance. Since Tetris already has a mechanism for calculating cleared lines, the fitness score

for a given individual is proportional to the number of lines it manages to clear while playing the game.

$$Fitness(Individual) = Number\ of\ Cleared\ Lines$$

5.6.4 Process of Natural Evolution

The process of applying genetic algorithms can be outlined through the series of steps that will lead to the effective optimisation of an AI agent. These phases cover the crucial aspects of genetic algorithms, such as initialising the population, assessing individual fitness, performing the selection, applying crossover and mutation operations, and iterating over generations.

5.6.5 Initialisation

The population is first initialised with 50 individuals. As stated earlier, each individual is filled with randomly generated floated numbers ranging from -10 to 10, derived from a normal distribution.

5.6.6 Assessing Performance

Each individual plays the game until it loses or reaches the number of 850 moves. The fitness function then evaluates each individual's performance and stores their fitnesses. Because the function is dependent on the number of cleared lines, the worst possible fitness is 0, indicating that the AI agent did not clear any lines during playtime.

5.6.7 Tournament Selection

Following the initial population of agents participating in the game and having their performance evaluated by the fitness function, the tournament selection procedure begins. This stage entails identifying two parents who will produce children for the future generation. To do this, a random sample of 5 individuals is drawn from the population, and the two fittest agents are chosen as parents. This method assures that the fittest agents have a higher chance of contributing to the next generation, enabling the improvement of the AI's performance.

5.6.8 Crossover

This procedure generates new agents with a mix of weights from their parents. The function randomly selects the respective weights from the two parents for each of the four attributes in the offspring. The process is conducted by selecting a random crossover point within the length

range of the first-parent and second-parent genomes. The genes of the parents on either side of the crossover point are then combined to produce two children. The first offspring is generated by combining first-parent genes before the crossover point with second-parent genes after the crossover point. Similarly, the second offspring is generated by combining second-parent genes before the crossover point with the first-parent genes after the crossover point, which results in producing two new children who inherit characteristics from both parent agents.

5.6.9 Producing a New Generation

The selection and crossover processes are repeated until a total of 50 offspring are produced. As a result, a new generation is formed with individuals that contain weights which contribute to better agent performance. The approach ensures that the next generation receives advantageous qualities from its parent agents, allowing for ongoing improvement of the AI's performance.

5.6.10 Mutation

Furthermore, each of the offspring weights has a slight probability of becoming mutated. A mutation is another important part of the genetic algorithms process because it introduces small random changes to the offspring's genetic material. This mechanism is necessary for several reasons:

- Diversification: Mutation promotes population diversity by limiting early convergence to poor solutions. Without mutation, the algorithm may converge on a local minimum or maximum too soon, missing out on potentially superior solutions (Lambora, 2019).
- Exploration: Mutation allows the algorithm to explore the search space by allowing it to investigate different weight values that would not have been accessed through crossover alone. This can lead to the development of potentially superior weights.

The mutation process is performed by modifying the offspring's weight by a value that lies within the range of -0.1 to 0.1. For each attribute, an offspring possesses, there is a 10% probability of it undergoing mutation.

5.7 Conclusion

In conclusion, the AI analysis section conducted a thorough examination of many areas of AI integration in Tetris, such as feature selection, the usage of genetic algorithms, and the application of mutation and crossover procedures. The research attempted to create a robust and efficient AI agent that could perform well in the game by carefully evaluating these characteristics. This study also emphasised the need of considering numerous criteria, such as height, holes, bumpiness, and line-clearing components, in order to provide a well-rounded evaluation of Tetris positions.

The following section of the report will discuss the requirements for a successful application of genetic algorithms, laying the groundwork for further development of AI.

6.0 Requirements Specification

The final step before starting the implementation of Genetic Algorithms is to define a set of prerequisites required for successful development.

6.1 Functional Requirements

Functional requirements are essential features and capabilities that the system should possess to achieve its intended goals.

- **Initial bots:** The initial bots will provide methods for interacting with the tutorial code. They will define how to access game variables, such as grid coordinates, piece shapes, and a score.
- **Genetic Algorithm Functions:** The code for implementing the evolution process, such as functions for population initialisation, mutation, crossover and tournament selection.
- **Customisation of the Tetris Variables:** To test the produced code, it is critical to be able to change game variables such as the falling piece's speed and the ability to halt the piece's descent and move it freely across the grid.

6.2 Non-Functional Requirements

These sets of requirements specify how the system should perform and what it can produce.

- **Testability** – The application should be able to provide feedback on the performance of AI agents, such as the score of each individual in every generation, the evolution of weights, and the genomes of the best-performing agents.
- **Guide** – A set of instructions should be produced to explain how to launch the game and which variables in the code to modify to observe different results.

6.3 Hardware

Genetic Algorithms may demand substantial computational power from the hardware they run on. Given the repetitive calculations involved and depending on the problem size, this method may require high-quality computer components, such as CPU, GPU, and RAM.

I have a gaming laptop, the Dell G7 17. This laptop contains all of the necessary hardware components, making it a suitable choice for training the AI without the need for acquiring extra hardware.

6.4 Time

The Genetic Algorithms approach is strong but time-consuming. AI training typically takes two to three days, depending on the complexity of the evaluation function, the size of the population,

and the number of generations. I planned to deploy 10-15 generations and expected training to take 1-2 days; however, there was a probability of some issues developing during training, so I allotted additional 2 days for this process.

After following a detailed discussion of the functional and non-functional criteria for integrating Artificial Intelligence, the following section will delve into the AI's actual implementation, focusing on the incorporation of Genetic Algorithms to optimise the Tetris agent's performance.

7.0 Agents Implementation

The initial stage of AI development commenced with the creation of various hand-crafted agents to examine the integration of the four features and investigate possible interactions with the tutorial code. Upon implementing each attribute for the evaluation function, the agents were tested to showcase the impact of these features on their performance.

7.1 Random Player

The primary method selected for interacting with the game through the code involved simulating key presses for the tetromino movements. This approach was chosen because it offered a simple and efficient way of integrating agents into the game. Furthermore, adopting an alternative strategy, such as modifying the coordinates of the pieces directly, could significantly increase the likelihood of introducing errors into the game code.

To simulate key presses, the Python library **pynput** was employed. It allowed the first bot to simply press arrow keys in a random sequence until the game is lost.

```
from pynput.keyboard import Key, Controller
```

Initially, the necessary library was imported.

Figure 35 Importing key press simulation library

```
keyboard = Controller()


def pressKey(button):
    keyboard.press(button)
    keyboard.release(button)
```

After that, a keyboard object was created, and a function that accepts a key value as a parameter was created. This function was designed to press the received button as a parameter.

Figure 36 Function to simulate key presses

Subsequently, the first bot, the random player, was developed with the following function.

```
def random_player():
    movement_list = [Key.up, Key.left, Key.right, Key.down, Key.space]
    button = random.choice(movement_list)
    pressKey(button)
```

Figure 37 Random-Player Bot function

The function creates an array of possible keys to press during the game. The value of a button is then supplied to the **pressKey** method at random from this list.

7.1.1 Testing the Random Player

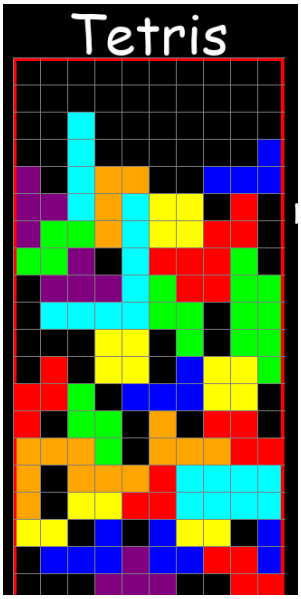


Figure 38 Random-Player gameplay

The tetromino pieces were effectively moved within the grid by the agent. Although the bot cleared only 1-2 lines as a result of the random movements, this agent demonstrated the successful implementation of key press simulations in the developed Tetris game. Furthermore, the bot highlighted that the **pynput** library was not the best solution for emulating key presses because it presses the keys too rapidly without the option to optimise this attribute. As a result, the library was changed to **pyautogui**, which enables key press speed to be adjusted.

7.2 Lowest-Point Player

The next bot was designed to place tetromino pieces in positions that would result in the lowest height, while also evaluating the height for each piece rotation. Given that the method for engaging with tetromino pieces had been established, the next step involved determining an approach for interacting with the Tetris grid.

Firstly, the function was created that accepts the piece object and grid as parameter values.

```
has_executed = False
def lowest_point_player(piece, grid):
    global has_executed
    if not has_executed:
        main_dict = {}
        for j in range(len(piece.shape)):
            piece.rotation = j
```

Figure 39 Lowes-Point Player function

The **has_executed** variable serves as a control parameter to initiate the evaluation process once for every time a piece is at the top of the grid. Subsequently, a dictionary to store the rating of each position at every rotation was created. The function then would start to calculate the position rating for each rotation by iterating through the possible piece rotations.

```
piece_pos = convert_shape_format(piece)
x0, y0 = piece_pos[0]
x1, y1 = piece_pos[1]
x2, y2 = piece_pos[2]
x3, y3 = piece_pos[3]

xb0, xb1, xb2, xb3, yb0, yb1, yb2, yb3 = x0, x1, x2, x3, y0, y1, y2, y3
```

Figure 40 Getting tetromino coordinates

Since the tutorial code included a function that returns the coordinates of each of the four tetromino blocks, this function was utilised to obtain the position of the current piece. This process was implemented within the aforementioned "for" loop and began by initialising the variable containing the coordinates. Next, each coordinate tuple was assigned to the x and y coordinates of each tetromino block. These variables were then copied to restore the initial coordinates for the next iteration.

```
x_min = min(x0, x1, x2, x3)
x_max = max(x0, x1, x2, x3)
y_min = max(y0, y1, y2, y3)
xb_min, xb_max, yb_min = x_min, x_max, y_min
height_moves = {}



while x_min != 0:
    x_min -= 1
    x0 -= 1
    x1 -= 1
    x2 -= 1
    x3 -= 1
```

Figure 41 Moving piece to the left

Subsequently, the minimum and maximum values from all four x and y coordinates were acquired, as the primary orientation of the piece on the grid was dependent on these values. Following that, a dictionary containing all positions and their heights was constructed.

Then, the loop decreased all the x values by one till the x with the smallest coordinate reached zero. This was done in order to set all four coordinates correctly to their positions.

```
x0_left, x1_left, x2_left, x3_left = x0, x1, x2, x3
x_max = max(x0, x1, x2, x3)

for i in range(10):
    x0, x1, x2, x3 = x0_left, x1_left, x2_left, x3_left
    x_max += i
    x0 += i
    x1 += i
    x2 += i
    x3 += i

    x_max = max(x0, x1, x2, x3)
    if x_max > 9:
        continue

    while y_min != 19:
        y_min += 1
        y0 += 1
        y1 += 1
        y2 += 1
        y3 += 1
        if grid[y0][x0] != (0, 0, 0) or grid[y1][x1] != (0, 0, 0) or grid[y2][x2] != (0, 0, 0) or grid[y3][x3] != (
        0, 0, 0):
            y_min -= 1
            y0 -= 1
            y1 -= 1
            y2 -= 1
            y3 -= 1
            break

    check_coords = [(x0, y0), (x1, y1), (x2, y2), (x3, y3)]
    height = min(check_coords, key=lambda x: x[1])[1]
```

Figure 42 Dropping the Piece

The code snippet above places each piece at all ten positions in a row and determines the height depending on the item's minimal y coordinate of tetromino blocks. Because the grid in Tetris is represented as a nested list, the lower the piece, the higher its y-coordinate.

```
        height_moves[i] = height

        x_min, x_max, y_min = xb_min, xb_max, yb_min
        x0, x1, x2, x3, y0, y1, y2, y3 = xb0, xb1, xb2, xb3, yb0, yb1, yb2, yb3
        main_dict[j] = height_moves


global max_height, go_coord, main_rotation
max_height = 0
go_coord = None
main_rotation = None

for rotation, positions in main_dict.items():
    for coord, height in positions.items():
        if height > max_height:
            max_height = height
            go_coord = coord
            main_rotation = rotation
```

Figure 43 Saving the position heights

The position and height were then stored and sent to the main dictionary, which also held the piece's current rotation. The values were restored, and the procedure was repeated until all positions had been examined.

Finally, global variables were constructed that held the critical information required for the agent to know exactly where to move the piece such as the rotation, height and coordinate of a position. This data was obtained by simply looping over the main dictionary and looking for the position with the highest y-coordinate value (height).

```
def movement(tetromino, rotation, x, coord):
    tetromino.rotation = rotation
    if x < coord:
        pressK("right")
    elif x > coord:
        pressK("left")
    else:
        pressK("down")
```

Figure 44 Movement Function

Then, a movement function was defined which would rotate and move the piece based on the passed coordinates and rotation in the parameters.

Finally, inside the main function the current piece's four block positions were acquired, as well as the minimal x-coordinate value and the agent was called.

```
pos = convert_shape_format(current_piece)
x1, x2, x3, x4 = pos[0][0], pos[1][0], pos[2][0], pos[3][0]
x_min = min(x1,x2,x3,x4)

if current_piece.y > 2:
    lowest_point_player(current_piece, grid)
    movement(current_piece, main_rotation, x_min, go_coord)
```

Figure 45 Calling Lowes-Point Player inside the main function

7.2.1 Testing the Lowest-Point Player

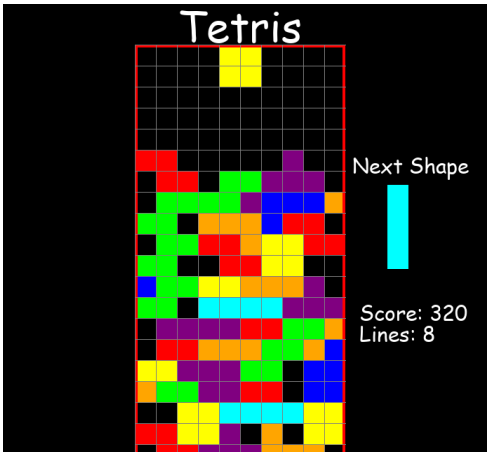


Figure 46 Lowest-Point Player gameplay

This agent outperformed the random player agent noticeably. It could keep the grid more stable and clear 7-8 lines on average.

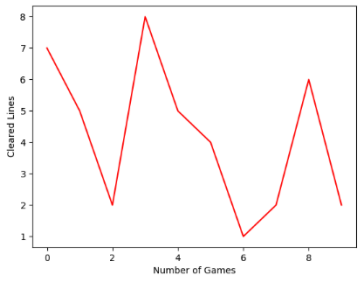


Figure 47 Plot of Lowest-Point Agent Performance

The agent demonstrated the efficiency of the grid evaluation method and provided the working framework for the implementation of the future parameters.

7.3 Adding Number of Holes Feature

Since the lowest-point player agent included a function for evaluating height at each position, it could be modified by adding different values to the height parameter which would serve as the position ranking.

The next feature that was implemented was the ability to count the number of holes in the grid.

```
def get_holes(grid):
    holes = 0
    overhead = 0
    for i in range(10):
        found_cell = False
        over_in_row = 0
        for j in range(20):
            if grid[j][i] != (0,0,0):
                found_cell = True
                over_in_row+=1
            if found_cell and grid[j][i]==(0,0,0):
                holes += 1
                overhead += over_in_row
    return holes
```

Figure 48 Function to Count Holes

In the game, a hole is defined as a black colour cell with an RGB value of (0,0,0). To find these holes, a function was written that iterated through the grid, looking for black areas. If this location was discovered and it had a block above, this spot was considered a hole.

The grid in the tutorial code is updated every time the tetromino is changed. As a result, the agent capability for calculating holes could not be implemented on the present grid because it could not obtain the grid's live state. The solution was to make a grid copy and update it with the coordinates while calculating the rating for each point. Because the grid is represented as a dictionary that is constantly updated, making a simple copy of it would be ineffective and could alter the values of the original grid; therefore, using the "**deepcopy**" library, I created a deep copy of the grid that could be changed without affecting the original grid.

```
check_coords = [(x0, y0), (x1, y1), (x2, y2), (x3, y3)]
grid_test = copy.deepcopy(grid)
grid_test[y0][x0], grid_test[y1][x1], grid_test[y2][x2], grid_test[y3][x3] = (1,1,1), (1,1,1), (1,1,1), (1,1,1)

rating = min(check_coords, key=lambda x: x[1])[1] - get_holes(grid_test)
```

Figure 49 Adding hole count parameter

The portion of a code calculating the position's height was changed to calculate the rating with additional features. After each cycle of dropping a piece at each position, a copy of the grid was

produced and filled with values based on the coordinates. This copy was then utilised to call the hole-calculation method. Because the presence of holes in a grid is detrimental to performance, the location that causes any holes should be rated lower. As a result, this input was sent to the rating variable with a negative sign.

7.3.1 Testing the Agent with the Hole-Count Parameter

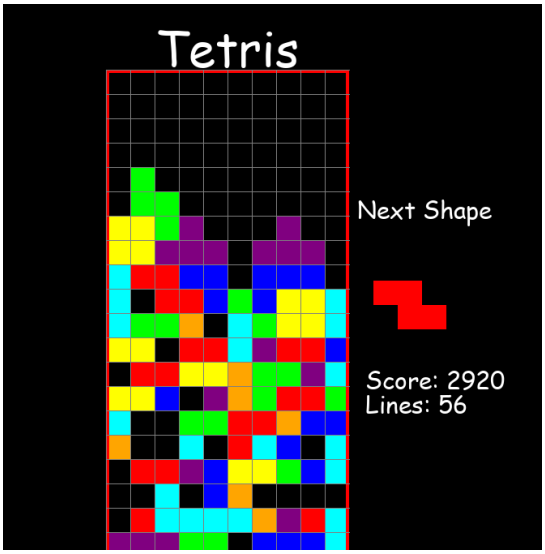


Figure 51 Agent with "Hole-Count" gameplay

The hole parameter produced a significant impact on the agent's performance. The agent was able to clean on average around 60-70 lines per game.

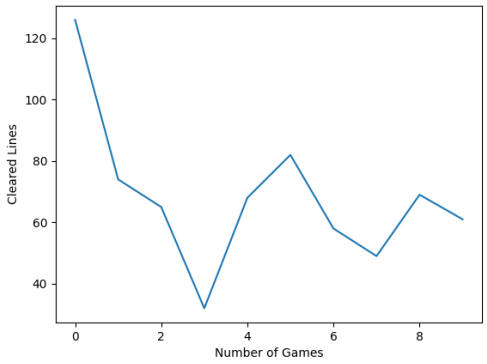


Figure 50 Plot of agent with "Hole-Count" performance

7.3.2 The Clearing Line Bug

During testing this agent also discovered the bug in the tutorial code responsible for clearing line functionality.

When there is a partial line between two full rows, the clearing function clears the row but does not entirely move the structure down, resulting in an empty row in the tetromino structure.

The bug was decided to be ignored due to project submission deadlines and the fact that the bug occurrence was rare, and it did not cause any serious problems with gameplay.

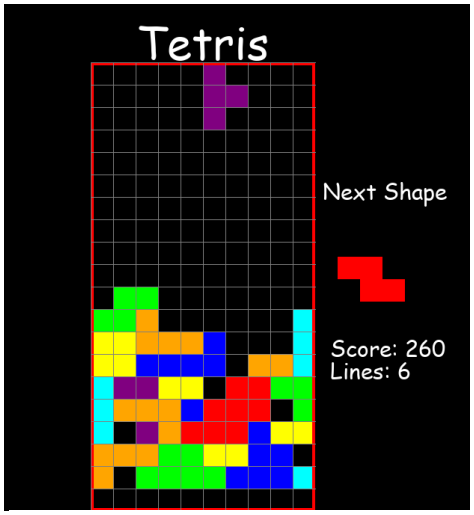


Figure 52 Clearing Line Bug

7.4 Adding Bumpiness

Another attribute that was added to the evaluation function was the bumpiness of a grid.

```
def calculateBumpiness(grid):

    bumpiness = 0
    def col_height(col):
        for row in range(len(grid)):
            if grid[row][col] != (0,0,0):
                return len(grid) - row
        return 0

    for col in range(len(grid[0]) - 1):
        left_height = col_height(col)
        right_height = col_height(col + 1)
        bumpiness += abs(left_height - right_height)

    return bumpiness
```

Figure 53 Bumpiness Function

The function has an internal method for determining the column heights. Then the function iterates through the grid and calls this method to get the heights of each column. Finally, the absolute difference between adjacent columns is calculated and added to the bumpiness variable.

The function was then called similarly to the "Number of Holes" parameter and then added to the rating of each position. Since the high bumpiness worsens the performance of an agent, this attribute reduces the rating of a move that causes high bumpiness and was passed with a negative sign.

7.4.1 Testing Agent with Bumpiness Parameter

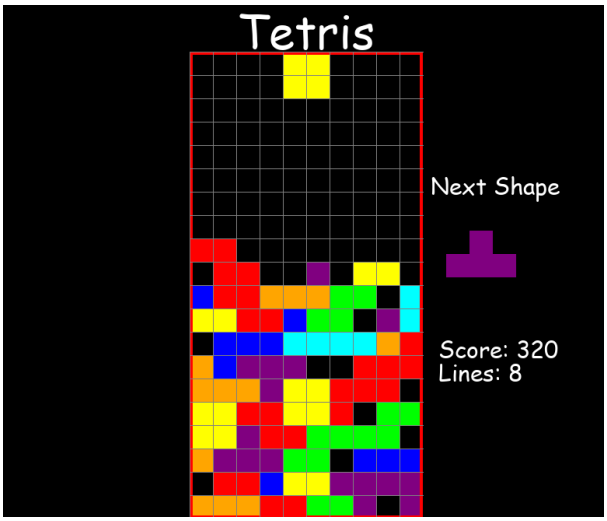


Figure 54 Agent with added Bumpiness parameter

Surprisingly, adding bumpiness to the evaluation function decreased an agent's performance by nearly 6 times. During the ten-run episode, the agent barely cleared more than 15 lines.

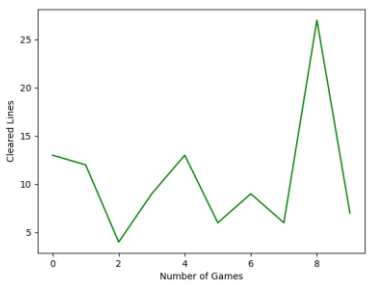


Figure 55 Plot of Agent with Bumpiness Performance

This performance could be rationalised by the fact that the hole and bumpiness characteristics both diminish a position's rating equally, implying that the agent would examine the best rating that satisfies both the hole number and the grid roughness values.

7.5 Adding the Clearing Piece Parameter

Finally, the last parameter responsible for increasing the rating of the position if a piece dropped there would clear lines was added to the evaluation function.

```
def clearing_piece(grid):
    inc = 0
    for i in range(len(grid) - 1, -1, -1):
        row = grid[i]
        if (0, 0, 0) not in row:
            inc += 1

    return inc
```

The function is similar to the clearing line function from the tutorial code. It iterates through the grid starting at the bottom and returns the number of cleared lines.

Figure 56 Clearing Piece Function

Then, this attribute was supplied to the evaluation function with a positive sign since the more lines the agent clears, the higher its performance.

7.5.1 Testing Agent with Clearing Piece Parameter

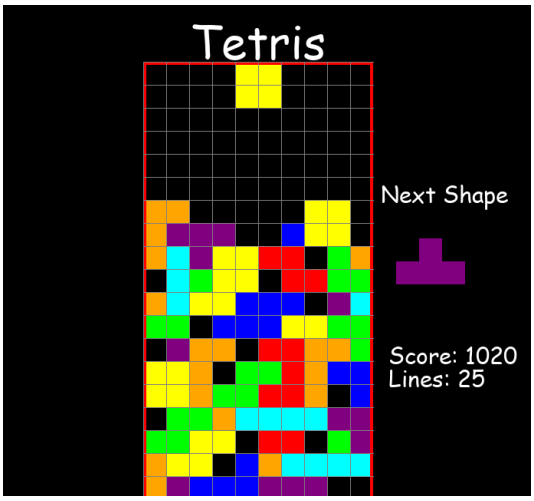


Figure 57 Agent with Clearing Piece Parameter Gameplay

The agent with all four attributes included in the evaluation function performed on average with 20 cleared lines per game.

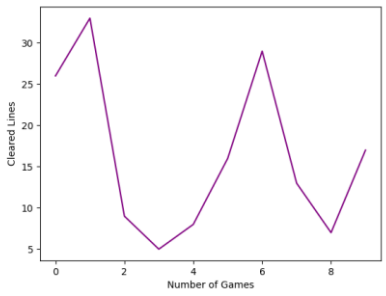


Figure 58 Plot of Agent with Clearing Piece Parameter Performance

7.6 Overall Testing

It was critical to examine the trends that emerged from the application of each of the four parameters in order to see how each parameter affected the agent's performance.

The testing was carried out on ten games for agents with one, two, three, and four features added to the evaluation function. The following outcomes were obtained from this test:

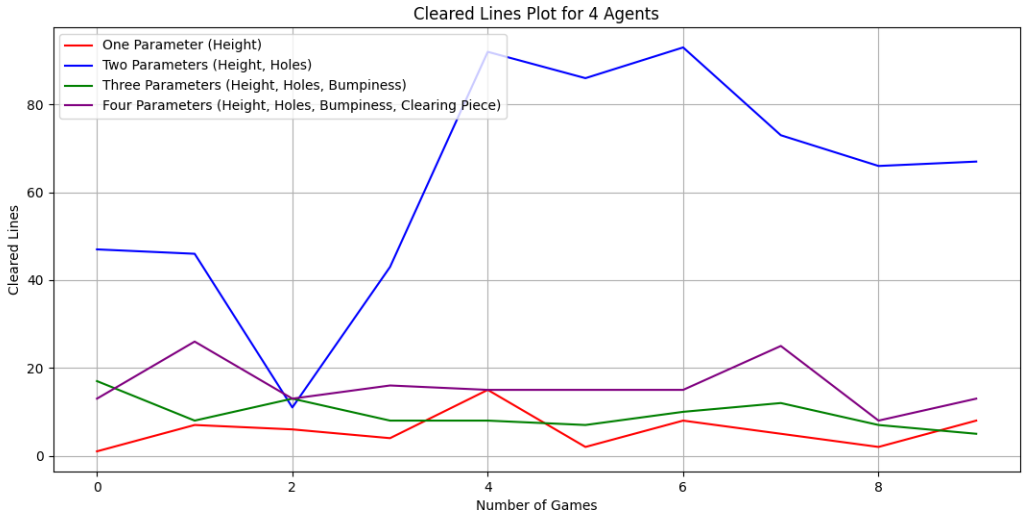


Figure 59 Plot of 4 Agent Performances

It is apparent that the agent with only two parameters - holes and height, outperformed the others. The agent incorporating all four parameters achieved the second-best performance; however, the number of cleared lines compared to the agents with three and one parameters was not significantly higher. This testing of attributes revealed a modest improvement as the number of features increased in the evaluation function. Nevertheless, the suboptimal performance may be attributed to the evaluation function treating all four features equally, while some parameters should have a greater impact on the position ranking than others. Furthermore, this analysis highlighted the importance of the hole parameter, as it significantly influences the agent's gameplay.

7.7 Conclusion

The project moved forward as intended after the successful testing of numerous agents with varying numbers of parameters within the evaluation function. The Agent Testing revealed the trends of the impact of different parameters and provided valuable insight into the methods of interacting with the tutorial code.

However, one important challenge remained: calculating the appropriate weights for these parameters. To overcome this issue, the project's next phase will concentrate on constructing genetic algorithms, to find the optimum potential weight combination.

8.0 Genetic Algorithms Implementation

This section will be focused on the technical side of implementing Genetic Algorithms. It will describe the incorporation of functions for mutation, crossover and tournament selection operators, along with the initial code for the population initialisation.

8.1 Initialisation

```
population_size = 50
num_generations = 30
mutation_rate = 0.1


def create_individual():
    return [random.uniform(-10, 10) for _ in range(4)]


def create_initial_population():
    return [create_individual() for _ in range(population_size)]
```

Figure 60 Code for Initialisation

Initially, the population size and the number of generations were determined. A decision was made to set the number of individuals in one generation to 50. This quantity of agents would improve the distribution of weights, potentially speeding up the process of identifying the optimal solution. The simulation was designed to run for 30 generations.

Consequently, two functions were developed: one for generating an individual with random weights and another for creating an entire generation comprised of random agents

8.2 Fitness Function Implementation

```
def fitness(individual, generation, individual_idx):
    print("Generation:", generation, 'Individual:', individual_idx, "Chromosome:", individual)
    return main(win, individual, generation, individual_idx)
```

Figure 61 Fitness Function Code

Given that the fitness function was determined to be the number of lines each individual cleared during the game, it essentially executes the game and returns the score upon game over. As demonstrated in the code, the main function was slightly modified to return the number of cleared lines and also display the generation and individual on the game screen, enhancing the visualisation of this process.

8.3 Selection Function Implementation

```
def selection(population, fitnesses):
    # Tournament selection
    tournament_size = 5
    parents = []
    for _ in range(2):  # Select 2 parents
        competitors = random.sample(list(zip(population, fitnesses)), tournament_size)
        winner = max(competitors, key=lambda x: x[1])
        parents.append(winner[0])
    return parents
```

Figure 62 Selection Function Code

The selection function takes the population and their respective fitness scores achieved during the game as input. This function then selects two parents from a combined list consisting of five individuals and their corresponding scores of cleared lines. Subsequently, it identifies the winner in this sample by seeking the fittest individual.

8.4 Crossover Function Implementation

```
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2
```

Figure 63 Crossover Function Code

The crossover function, as defined in the code, accepts two parents as input. It then generates a random crossover point within the range of 1 and the length of the parent's weights list minus 1. Subsequently, the function creates two new offspring by combining the weights of the parents at the determined crossover point.

8.5 Mutation Function Implementation

```
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] += random.uniform(-0.1, 0.1)
```

Figure 64 Mutation Function Code

The mutation function accepts the individual and a mutation rate as the parameters. It then iterates through the weights and alters the corresponding weight by adding a random value within the range of -0.1 and 0.1 with the probability defined in the mutation rate.

8.6 Gathering the Data

For the testing it was essential to gather the data during the process of agents evolution, therefore the functions for saving the population, fitness scores and best individual were implemented.

```
def save_population(population, generation):
    filename = f"population_generation_{generation}.txt"

    with open(filename, "w") as f:
        for individual in population:
            weights_str = " ".join(map(str, individual))
            f.write(f"{weights_str}\n")


def save_fitnesses(fitnesses, generation):
    filename = f"fitnesses_generation_{generation}.txt"

    with open(filename, "w") as f:
        for fitness in fitnesses:
            f.write(f"{fitness}\n")


def save_best_individual(individual, generation):
    filename = f"best_individual_generation_{generation}.txt"

    with open(filename, "w") as f:
        weights = individual[0]  # Extract weights from the tuple (weights, fitness)
        weights_str = " ".join(map(str, weights))
        f.write(f"{weights_str}\n")
```

Figure 65 Code for Gathering Data

The **save_population** function takes as input the population and the current generation number. It generates a filename based on the generation number and writes the population weights to it.

The weights of each individual are separated by spaces, and each individual is recorded on a new line.

A similar pattern is followed by the **save_fitnesses** function. It accepts the fitnesses and generation number as input, generates a filename, and writes the fitnesses to the file, one per line

Finally, the **save_best_individual** function takes as input the best individual and generation number. It generates a filename based on the generation number and saves the best individual's weights to it.

8.7 Integration of Genetic Algorithms into the Game

```
def main_menu(win):
    run = True
    while run:
        win.fill((0, 0, 0))
        draw_text_middle(win, 'Press Any Key To Play', 60, (255, 255, 255))
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
                pygame.mixer.music.stop()
            if event.type == pygame.KEYDOWN:
                # main(win)
                population = create_initial_population()
                save_population(population, 0)
                for generation in range(num_generations):
                    fitnesses = [fitness(individual, generation, i) for i, individual in enumerate(population)]
                    save_fitnesses(fitnesses, generation)

                    best_individual = max(zip(population, fitnesses), key=lambda x: x[1])
                    save_best_individual(best_individual, generation + 9)
                    print(f"Generation {generation}: Best fitness = {best_individual[1]}, Weights = {best_individual[0]}")

                    new_population = []
                    while len(new_population) < population_size:
                        # Selection
                        parent1, parent2 = selection(population, fitnesses)

                        # Crossover
                        offspring1, offspring2 = crossover(parent1, parent2)

                        # Mutation
                        mutate(offspring1, mutation_rate)
                        mutate(offspring2, mutation_rate)

                        new_population.extend([offspring1, offspring2])
                    population = new_population
                    save_population(population, generation)
```

Figure 66 Code for Integration of Genetic Algorithms

The integration of the genetic algorithms was implemented in the main menu function of the tutorial code.

The procedure begins when a key is pressed in the Tetris main menu and the initial population is formed first and saved to a file. The algorithm then repeats the preceding stages for the specified number of generations:

- Run the fitness function to assess the fitness of each individual in the population.
- Save the current generation's fitnesses to a file.
- Save the best individual (with the highest fitness) to a file.
- Print to the console the best fitness and weights of the current generation.
- Create a new population by performing selection, crossover, and mutation on the existing population.
- Save the newly created population to a file.

The cycle is repeated until all generations are processed. For each generation, the final population, fitnesses, and best individuals are kept in separate files for subsequent analysis.

Then, the process of the training began, and the results obtained are discussed in the following chapter of the report.

9.0 AI Testing

The Genetic Algorithm (GA) testing phase was critical for observing the evolution and performance of the AI agents across generations.

9.1 Evolution Testing

The first generation did not produce satisfying results since the agents worked inadequately and rewarded the moves that creates holes and high structure.

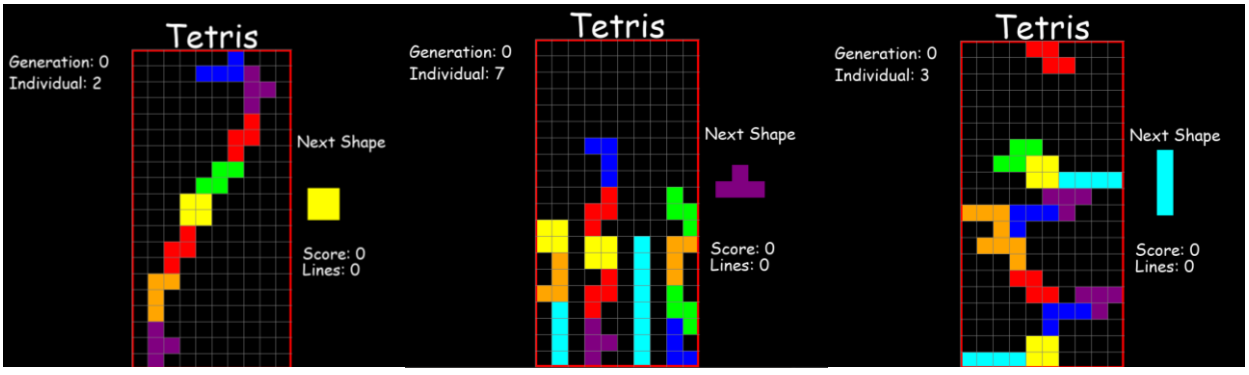


Figure 67 Initial AI Performance

However, as the generations passed, the agents began to grow and adapt, resulting in considerable gains in performance**.** At generation 4, the agents achieved the maximum possible score in 850 moves, prompting the move limit to be increased to 1150. This allowed for a deeper investigation of the agents' capabilities and gave them a chance to evolve. Training proceeded,

and by generation 14, I thought that the agents had achieved the optimal weights for their performance and there was no need for continuing the evolution process. As a result, the training process was halted at this moment.

The plot below demonstrates the performance of each individual of each generation.

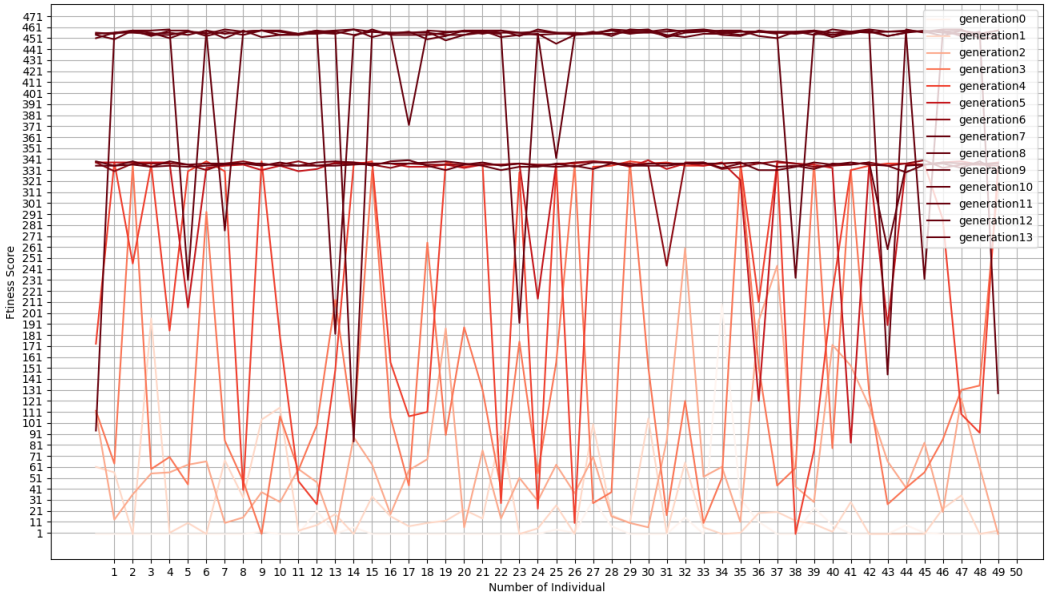


Figure 68 Performance of Each Individual Across Generations

The fitness scores fluctuated significantly in the early phases, showing that the AI had not yet found optimal weights. The plot got more stable as the generations progressed, implying that proper weights were eventually determined.

The following plot reveals the average performance of each generation as well.

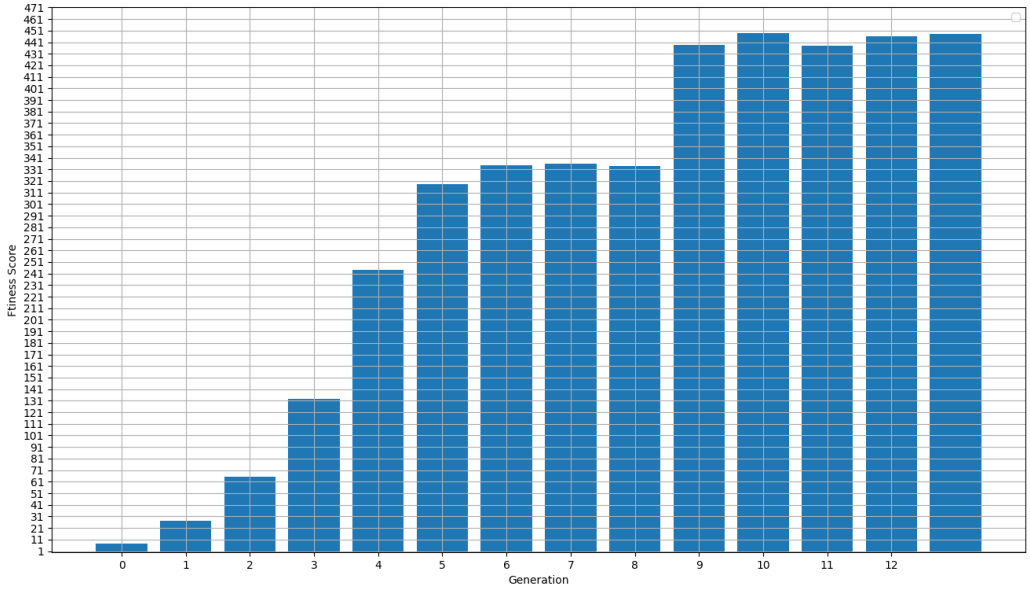
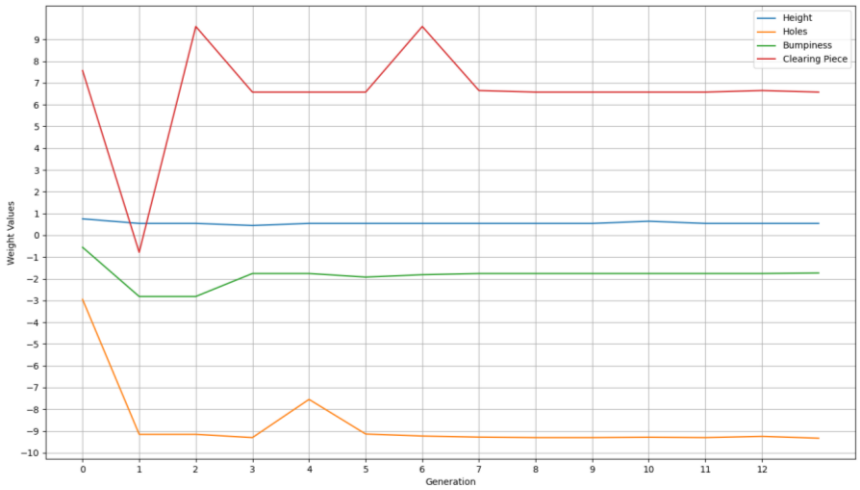


Figure 69 Average Fitness per Generation

This plot also highlights the effectiveness of the Genetic Algorithm, as the agent was capable of clearing an average of 450 lines every game in just 8 generations.

The next plot will demonstrate the evolution of the weights by displaying the best-performed agent of each generation.



The final agent exhibited the following results:

- **Height Weight**: Throughout the training process, the weight for the height parameter remained relatively stable. The final value was approximately 0.5, indicating that the AI deemed this parameter as less critical, but still retained a positive value.
- **Holes Weight**: The value for this weight experienced significant changes during the training process. Initially, the value was -3, but by the end of the training, the weight reached a value of -9.32. This suggests that the AI considered the holes parameter as the most crucial of all features.
- **Bumpiness Weight**: The bumpiness parameter underwent substantial changes only until generation 3. After that, the weight value stabilised and reached -1.73 by the final generation. The AI also regarded this feature as influential, though not as much as the holes parameter. This implies that, for optimal performance, the agent can tolerate some grid irregularity to minimise the number of holes.
- **Clearing Piece Weight**: This parameter reached a value of 6.5 at the end of the training process. The AI considered this parameter important for the agent's performance as well.

9.2 Testing the Trained AI

When the taught AI was tested, its performance proved to be exceptional. The agent cleared over 1,500 lines every game, demonstrating the efficiency of the training process and the AI's capacity to adapt and optimise its gameplay strategy.

The weights used for the final AI agents were the following:

- **Height**: 0.5436822764440379
- **Holes**: -9.328911430903139
- **Bumpiness**: -1.735608284805026
- **Clearing** Piece: 6.577302970502618

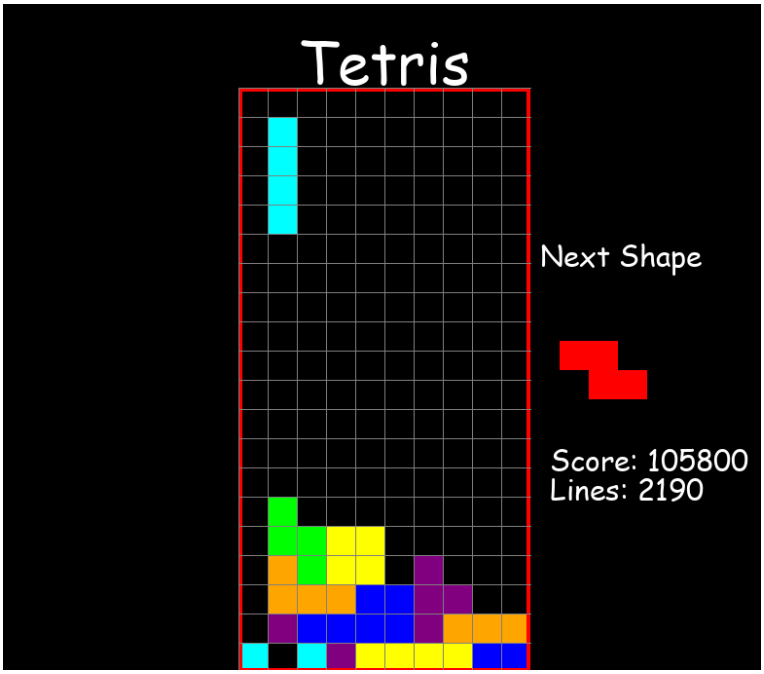


Figure 70 Trained AI Performance

9.3 Conclusion

In conclusion, the testing of the agent and AI throughout the project proved the efficacy of the algorithms and approaches used to construct intelligent Tetris agents. The procedure entailed a detailed investigation of Genetic Algorithms, which were crucial in optimising the weights for each parameter. The agent's performance gradually increased, and the remarkable performance of the trained AI, clearing about 1,500 lines every game, underpins the project's success.

10.0 Final Product

In the final product, options to play Tetris, observe the Genetic Algorithm process, and load the trained AI were integrated and presented as selectable choices in the game's main menu. To enhance the user experience, hints for in-game movements were incorporated into the game screen. The complete source code will be provided in the Appendices section of the report.

Ultimately, the entire game was compiled into an executable file to facilitate convenient launching on any computer. Thus, the final deliverable represents the culmination of the project and can be easily accessed through the Tetris.exe file.

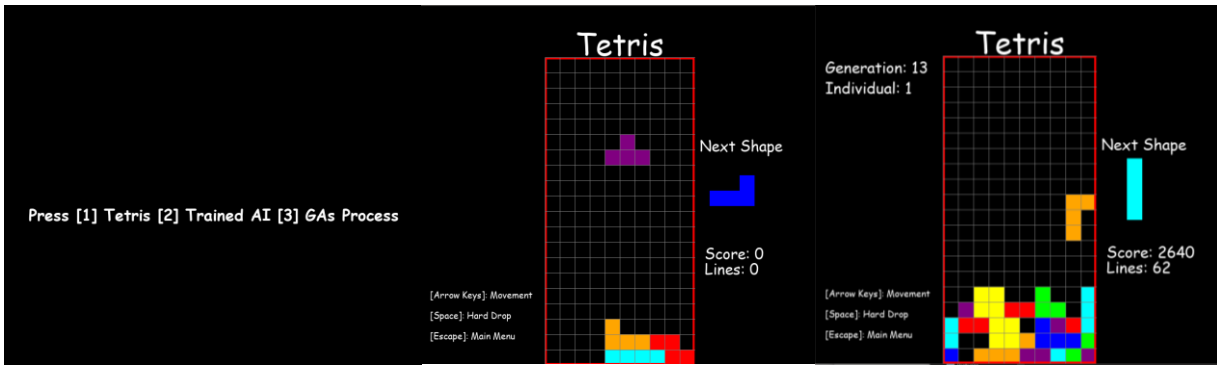


Figure 71 Final Product

11.0 Evaluation

11.1 Game Implementation Evaluation

It is evident that the project successfully accomplished its primary objective at this stage, which was to develop a well-functioning Tetris game. The PyGame library proved to be an appropriate choice for the game development library, as it facilitated a faster understanding of the tutorial code and enabled the discovery of improved ways to integrate future AI agents.

Throughout the game implementation, the project encountered various challenges, including learning a new library, making adjustments to the tutorial code, and addressing bugs. The majority of these challenges were effectively overcome, resulting in a robust and stable game. However, there was one primary issue that remained unresolved by the end of this project – the line-clearing bug. This bug emerged several times during the project development, and although it did not significantly hinder the development process, resolving this issue would have improved the final product's quality. Nonetheless, due to the project's submission deadlines, this issue was unfortunately left unaddressed.

Despite these challenges, this stage of the project yielded notable accomplishments. Valuable insights were gained into the field of game development, and familiarity with new libraries was acquired. Although the tutorial did not heavily employ object-oriented programming principles in its code, the project still served to reinforce knowledge in this domain. Additionally, the creation of initial agents to interact with the tutorial code further enhanced my analytical and critical thinking skills.

11.2 AI Implementation Evaluation

The primary goal of constructing an intelligent agent capable of delivering extraordinary performance was successfully achieved during this stage of the project, as proven in the AI Testing part of this report. The decision to employ Genetic Algorithms as the major method for producing artificial intelligence proved useful since they efficiently solved the issues of determining optimal values through natural selection and evolution processes.

Several complex challenges were encountered during this stage, including understanding the fundamentals of Genetic Algorithms, mastering simple operators, and devising strategies to integrate them into the tutorial code. These challenges were gradually overcome as the project progressed, ultimately leading to the development of an impressive AI agent.

While the final performance of the trained agent was commendable, there is always room for further improvement. Potential enhancements include refining the evaluation function's logic to explore every possible end position for each piece, allowing for advanced manoeuvres such as "T-spins" to boost performance. Additionally, incorporating more parameters into the evaluation function, such as well count, aggregate height, and cumulative height, could further improve the agent's performance. However, as previously indicated, due to time constraints, it was not possible to install every potential feature.

Overall, this stage of development greatly contributed to the extension of my skills in AI and Data Science. It enabled me to reinforce my knowledge of Python data structures and acquire expertise in the natural selection process. I gained competence in Genetic Algorithms and refined my talents in visualising enormous amounts of data and gathering extensive knowledge, both of which are considered the most valuable accomplishments achieved.

11.3 Time Management

The project was allocated a total of 14 weeks for development, necessitating the use of a well-structured plan to ensure effective time management. This plan was presented in a previous assignment in the form of a Gantt Chart, which divided the project development stages into 2–3 week sprints. However, some stages demanded more time than anticipated for successful implementation. These deviations were primarily due to the time required for learning new information alongside development, which proved to be more extensive than initially expected.

Additionally, the testing of AI agents was a time-consuming process, as it took approximately 2-3 days to train only half of the intended generations. Even minor adjustments to the code necessitated restarting the training process. Despite these challenges, the project also facilitated the development of essential skills such as stress resilience and improved time management. Consequently, the project's primary workflow remained largely consistent with the initial plan, leading to the successful delivery of both the completed artefact and a comprehensive project report.

12.0 Project Conclusion

In conclusion, this project met its key goals of developing a functional Tetris game and creating an intelligent agent capable of giving exceptional performance.

Numerous problems were met and overcome throughout the project, such as learning new libraries, comprehending the principles of Genetic Algorithms, and integrating the AI agent into the game's code. These obstacles were handled methodically, culminating in a well-functioned Tetris game and a high-performing AI agent.

This project was a great demonstration of the power and potential of AI in game development. It demonstrated the effectiveness of Genetic Algorithms and laid the groundwork for future projects in this field.

13.0 Bibliography

Algorta, S. and Şimşek, Ö., 2019. The game of Tetris in machine learning. arXiv preprint arXiv:1905.01652.

Bertsekas, Dimitri P and Tsitsiklis, John N. NeuroDynamic Programming. Athena Scientific, 1996.

Code Bullet 2020, I created an A.I. to DESTROY Tetris. Available at: https://www.youtube.com/watch?v=QOJfyp0KMmM&list=LL&index=31&ab_channel=CodeBullet

da Silva, R.S. and Parpinelli, R.S., 2017, October. Playing the original game boy tetris using a real coded genetic algorithm. In 2017 Brazilian Conference on Intelligent Systems (BRACIS) (pp. 282-287). IEEE.

Demaine, E.D., Hohenberger, S. and Liben-Nowell, D., 2003, June. Tetris is hard, even to approximate. In COCOON (pp. 351-363).

Fahey, C., 2003, Tetris ai, Available at: http://www. colinfahey.com/tetris/tetris.html

Flom, L. and Robinson, C., 2005. Using a genetic algorithm to weight an evaluation function for Tetris.

Huang, M., Ali, R. and Liao, J., 2017. The effect of user experience in online games on word of mouth: A pleasure-arousal-dominance (PAD) model perspective. Computers in Human Behavior, 75, pp.329-338.

Lambora, A., Gupta, K. and Chopra, K., 2019, February. Genetic algorithm-A literature review. In 2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon) (pp. 380-384). IEEE.

Lewis, J., 2015, Playing Tetris with Genetic Algorithms. Available at: http://cs229.stanford.edu/proj2015/238_report.pdf

Plank-Blasko, D., 2015. 'From russia with Fun!': Tetris, Korobeiniki and the ludic soviet. The Soundtrack, 8(1-2), pp.7-24.

Prisco, J., 2019, "Tetris: The Soviet 'mind game' that took over the world", CNN, 1 November. Available at: https://edition.cnn.com/style/article/tetris-video-gamehistory/index.html#:~:text=It%20was%20June%206%2C%201984,them%20to%20form%20gapless%20lines

Techwithtim 2022, Learn how to create Tetris step-by-step. Available at: https://www.techwithtim.net/tutorials/game-development-with-python/tetris-pygame/

Temple, M. (Director). From russia with love. BBC, 2004

Tetris Wiki, 2020, Scoring, Available at: https://tetris.fandom.com/wiki/Scoring

Tetris Wiki, 2020, Random Generator, Available at: https://tetris.fandom.com/wiki/Random_Generator

Tsitsiklis, John N and Van Roy, Benjamin. Feature-based methods for large scale dynamic programming. Machine Learning, 22(1-3):59–94, 1996.

Yannakakis, G.N. and Togelius, J., 2018. Artificial intelligence and games (Vol. 2, pp. 2475-1502). New York: Springer.

14.0 Appendix (Full Source Code)

```
# IMPORTS
import pygame
import random
from pynput.keyboard import Key, Controller
import pyautogui
import copy

pyautogui.PAUSE = 0
# creating the data structure for pieces
# setting up global vars
# functions
# - create_grid
# - draw_grid
# - draw_window
# - rotating shape in main
# - setting up the main

"""
10 x 20 square grid
shapes: S, Z, I, O, J, L, T
represented in order by 0 - 6
"""

# Initiate pygame objects
pygame.font.init()
pygame.mixer.init()

# Load sound effects
theme_soundtrack = pygame.mixer.music.load('theme.mp3')
clear_row = pygame.mixer.Sound('clear.mp3')

# GLOBALS VARS
s_width = 800
s_height = 700
play_width = 300  # meaning 300 // 10 = 30 width per block
play_height = 600  # meaning 600 // 20 = 30 height per block
block_size = 30

# Coords
top_left_x = (s_width - play_width) // 2
top_left_y = s_height - play_height

# SHAPE FORMATS

S = [['.....',
      '......',
      '..00..',
      '.00...',
      '.....'],
     ['.....',
      '..0..',
      '..00.',
      '...0.',
      '.....']]
```

```
Z = [['.....',
      '.....',
      '.OO..',
      '..OO.',
      '.....'],
     ['.....',
      '..O..',
      '.OO..',
      '.O...',
      '.....']]

I = [['..O..',
      '..O..',
      '..O..',
      '..O..',
      '.....'],
     ['.....',
      'OOOO.',
      '.....',
      '.....',
      '.....']]

O = [['.....',
      '.....',
      '.OO..',
      '.OO..',
      '.....']]

J = [['.....',
      '.O...',
      '.OOO.',
      '.....',
      '.....'],
     ['.....',
      '..OO.',
      '..O..',
      '..O..',
      '.....'],
     ['.....',
      '.....',
      '.OOO.',
      '...O.',
      '.....'],
     ['.....',
      '..O..',
      '..O..',
      '.OO..',
      '.....']]

L = [['.....',
      '...O.',
      '.OOO.',
      '.....',
      '.....'],
     ['.....',
      '..O..',
```

```
      '..0..',
      '..00.',
      '.....'],
     ['.....',
      '.....',
      '.000.',
      '.0...',
      '.....'],
     ['.....',
      '.00..',
      '..0..',
      '..0..',
      '.....']]

T = [['.....',
      '..0..',
      '.000.',
      '.....',
      '.....'],
     ['.....',
      '..0..',
      '..00.',
      '..0..',
      '.....'],
     ['.....',
      '.....',
      '.000.',
      '..0..',
      '.....'],
     ['.....',
      '..0..',
      '.00..',
      '..0..',
      '.....']]

# Lists of shapes and their colors
shapes = [S, Z, I, O, J, L, T]
shape_colors = [(0, 255, 0), (255, 0, 0), (0, 255, 255), (255, 255, 0), (255,
165, 0), (0, 0, 255), (128, 0, 128)]
# index 0 - 6 represent shape

# Create empty bag that will contain tetromino shapes
bag = []


# Piece object
class Piece(object):
    def __init__(self, x, y, shape):
        self.x = x
        self.y = y
        self.shape = shape
        self.color = shape_colors[shapes.index(shape)]
        self.rotation = 0


# Create grid
def create_grid(locked_positions={}):
```

```
    grid = [[(0, 0, 0) for x in range(10)] for x in range(20)]  # intialise
black color for each grid position

    # Check for colors
    for i in range(len(grid)):
        for j in range(len(grid[i])):
            if (j, i) in locked_positions:
                c = locked_positions[(j, i)]
                grid[i][j] = c
    return grid


# Convert the shape of the piece
def convert_shape_format(shape):
    positions = []  # list of positions
    # get a sublist of the shape depending on its rotation
    format = shape.shape[shape.rotation % len(shape.shape)]

    # iterate through the shape and get positions for each block
    for i, line in enumerate(format):
        row = list(line)
        for j, column in enumerate(row):
            if column == '0':
                positions.append((shape.x + j, shape.y + i))
    for i, pos in enumerate(positions):
        positions[i] = (pos[0] - 2, pos[1] - 4)

    return positions


# Check for valid spaces in the grid
def valid_space(shape, grid):
    # check for black color on a grid and create list of accepted positions
    accepted_pos = [[(j, i) for j in range(10) if grid[i][j] == (0, 0, 0)]
for i in range(20)]
    accepted_pos = [j for sub in accepted_pos for j in sub]

    # format the shape of a current piece
    formatted = convert_shape_format(shape)

    # check if piece is in an accepted_pos
    for pos in formatted:
        if pos not in accepted_pos:
            if pos[1] > -1:
                return False
    return True


# Check if the game is over
def check_lost(positions):
    for pos in positions:
        x, y = pos
        if y < 1:  # if piece above the screen
            return True

    return False
```

```
###### 7-bag-randomizer #########
def create_bag():
    global bag
    bag = shapes.copy()  # get copy of a shapes list
    random.shuffle(bag)  # rearrange pieces in the bag


def get_next_tetromino():
    global bag
    if not bag:
        create_bag()
    return bag.pop()  # pop the piece from a bag


def update_pieces():
    global current_piece, next_piece
    current_piece = next_piece
    next_piece = get_next_tetromino()


##############################################

# GET TETRIS PIECE WITH RANDOM SHAPE
def get_shape(shape):
    return Piece(5, 0, shape)


# Draw text in the main menu
def draw_text_middle(surface, text, size, color):
    font = pygame.font.SysFont('comicsans', size, bold=True)
    label = font.render(text, 1, color)

    surface.blit(label, (
        top_left_x + play_width / 2 - (label.get_width() / 2), top_left_y +
play_height / 2 - (label.get_height() / 2)))


# Draw the tetris grid (lines)
def draw_grid(surface, grid):
    # coordinates
    sx = top_left_x
    sy = top_left_y

    # DRAW LINES FOR A GRID
    for i in range(len(grid)):
        pygame.draw.line(surface, (128, 128, 128), (sx, sy + i * block_size),
(sx + play_width, sy + i * block_size))
        for j in range(len(grid[i])):
            pygame.draw.line(surface, (128, 128, 128), (sx + j * block_size,
sy),
                             (sx + j * block_size, sy + play_height))


# Clear full rows
def clear_rows(grid, locked):
    inc = 0
```

```
    # iterate through grid from bottom
    for i in range(len(grid) - 1, -1, -1):
        row = grid[i]
        # if ther is block
        if (0, 0, 0) not in row:
            inc += 1
            ind = i
            for j in range(len(row)):
                try:
                    del locked[(j, i)]
                except:
                    continue
    # update the locked positions
    if inc > 0:
        clear_row.play()
        for key in sorted(list(locked), key=lambda x: x[1])[::-1]:
            x, y = key
            if y < ind:
                newKey = (x, y + inc)
                locked[newKey] = locked.pop(key)

    return inc


# Draw the next shape
def draw_next_shape(shape, surface):
    font = pygame.font.SysFont('comicsans', 30)
    label = font.render('Next Shape', 1, (255, 255, 255))

    sx = top_left_x + play_width
    sy = top_left_y + play_height / 2 - 100
    format = shape.shape[shape.rotation % len(shape.shape)]

    # iterate through shape
    for i, line in enumerate(format):
        row = list(line)
        for j, column in enumerate(row):
            if column == '0':  # if block
                # draw
                pygame.draw.rect(surface, shape.color,
                                 (sx + j * block_size, sy + i * block_size,
block_size, block_size), 0)

    surface.blit(label, (sx + 10, sy - 50))


# Display everything in the game
def draw_window(surface, grid, score=0, rows=0, generation=0,
individual_idx=0, condition=True):
    surface.fill((0, 0, 0))  # fill surface in black

    # DRAW TITLE
    pygame.font.init()
    font = pygame.font.SysFont('comicsans', 60)
    label = font.render('Tetris', 1, (255, 255, 255))

    # PUT TITLE ON MIDDLE OF A SCREEN
```

```
    surface.blit(label, (top_left_x + play_width / 2 - (label.get_width() /
2), 30))

    # draw the blocks on a grid (based on color in grid[i][j])
    for i in range(len(grid)):
        for j in range(len(grid[i])):
            pygame.draw.rect(surface, grid[i][j],
                             (top_left_x + j * block_size, top_left_y + i *
block_size, block_size, block_size), 0)

    # DRAW RED BORDERS OF A GRID
    pygame.draw.rect(surface, (255, 0, 0), (top_left_x, top_left_y,
play_width, play_height), 4)

    # DRAW SCORE
    font = pygame.font.SysFont('comicsans', 30)
    font1 = pygame.font.SysFont('comicsans', 18)
    label = font.render('Score: ' + str(score), 1, (255, 255, 255))
    label_score = font.render('Lines: ' + str(rows), 1, (255, 255, 255))

    # DRAW MOVEMENT INSTRUCTIONS
    label_movement = font1.render("[Arrow Keys]: Movement", 1, (255, 255,
255))
    label_drop = font1.render("[Space]: Hard Drop", 1, (255, 255, 255))
    label_menu = font1.render("[Escape]: Main Menu", 1, (255, 255, 255))

    # CORDS FOR LABELs
    sx = top_left_x + play_width
    sy = top_left_y + play_height / 2 - 100
    surface.blit(label, (sx + 20, sy + 160))
    surface.blit(label_score, (sx + 20, sy + 190))
    surface.blit(label_movement, (sx - 530, sy + 250))
    surface.blit(label_drop, (sx - 530, sy + 290))
    surface.blit(label_menu, (sx - 530, sy + 330))
    if condition:
        label_generation = font.render('Generation: ' + str(generation), 1,
(255, 255, 255))
        label_individual = font.render('Individual: ' + str(individual_idx),
1, (255, 255, 255))
        surface.blit(label_generation, (sx - 530, sy - 200))
        surface.blit(label_individual, (sx - 530, sy - 160))

    draw_grid(surface, grid)
    # pygame.display.update()


# Random Player bot
def random_player():
    movement_list = [Key.up, Key.left, Key.right, Key.down, Key.space]
    button = random.choice(movement_list)
    pressKey(button)


# Get number of holes
def get_holes(grid):
    holes = 0
    overhead = 0
```

```
    # Iterate through grid
    for i in range(10):
        found_cell = False
        over_in_row = 0
        for j in range(20):
            if grid[j][i] != (0, 0, 0):
                found_cell = True  # If found cell which is not empty
                over_in_row += 1
            if found_cell and grid[j][i] == (0, 0, 0):  # If found cell and
holes increase num of holes
                holes += 1
                overhead += over_in_row
    return holes


# Get bumpiness
def calculateBumpiness(grid):
    bumpiness = 0

    # Get column height
    def col_height(col):
        for row in range(len(grid)):
            if grid[row][col] != (0, 0, 0):
                return len(grid) - row
        return 0

    # Calculate the absolute difference
    for col in range(len(grid[0]) - 1):
        left_height = col_height(col)
        right_height = col_height(col + 1)
        bumpiness += abs(left_height - right_height)

    return bumpiness


# Get clearing piece
def clearing_piece(grid):
    inc = 0
    # Search for full row in the grid
    for i in range(len(grid) - 1, -1, -1):
        row = grid[i]
        if (0, 0, 0) not in row:
            inc += 1
            ind = i
    return inc


################################################
############### AI AGENT #######################
################################################

has_executed = False  # control parameter for agent initialisation


def agent(piece, grid, w1, w2, w3, w4):
    global has_executed
    if not has_executed:
```

```
        main_dict = {}  # dict with ratings
        for j in range(len(piece.shape)):  # iterate through rotations
            grid_copy = copy.deepcopy(grid)  # create copy of grid
            piece.rotation = j  # set the rotation
            piece_pos = convert_shape_format(piece)  # get piece blocks
coords
            x0, y0 = piece_pos[0]
            x1, y1 = piece_pos[1]
            x2, y2 = piece_pos[2]
            x3, y3 = piece_pos[3]

            xb0, xb1, xb2, xb3, yb0, yb1, yb2, yb3 = x0, x1, x2, x3, y0, y1,
y2, y3  # save the values

            # max y = 19 max x = 9
            # todo:
            #  BEFORE STARTING SAVE THE INITIAL ROTATION (piece.rotation)
            #  1) find min x
            #  2) see by how much change all x's if changing "min x" value to
0
            #  3) start to change y till Y MIN = 19 or piece present (not
(0,0,0) )
            #  4) calculate the height
            #  5) when checked and calculated, do the same with x = 1,2,3...
and continue till X MAX = 9
            #  6) when the optimal solution is found, calculate the moves
            #  (e.g if X initial = 5, X optimal = 3 => go left 2 times =>
drop)

            x_min = min(x0, x1, x2, x3)
            x_max = max(x0, x1, x2, x3)
            y_min = max(y0, y1, y2, y3)
            xb_min, xb_max, yb_min = x_min, x_max, y_min
            height_moves = {}

            # move piece to the left
            while x_min != 0:
                x_min -= 1
                x0 -= 1
                x1 -= 1
                x2 -= 1
                x3 -= 1

            x0_left, x1_left, x2_left, x3_left = x0, x1, x2, x3
            x_max = max(x0, x1, x2, x3)

            # start moving the piece to the right
            for i in range(10):
                x0, x1, x2, x3 = x0_left, x1_left, x2_left, x3_left
                # y0, y1, y2, y3 = yb0, yb1, yb2, yb3
                x_max += i
                x0 += i
                x1 += i
                x2 += i
                x3 += i

                x_max = max(x0, x1, x2, x3)
```

```
                if x_max > 9:
                    continue
                # start dropping the piece
                while y_min != 19:
                    y_min += 1
                    y0 += 1
                    y1 += 1
                    y2 += 1
                    y3 += 1
                    if grid[y0][x0] != (0, 0, 0) or grid[y1][x1] != (0, 0, 0)
or grid[y2][x2] != (0, 0, 0) or grid[y3][
                            x3] != (
                                0, 0, 0):
                        y_min -= 1
                        y0 -= 1
                        y1 -= 1
                        y2 -= 1
                        y3 -= 1
                        break
                # get coords
                check_coords = [(x0, y0), (x1, y1), (x2, y2), (x3, y3)]
                grid_test = copy.deepcopy(grid)  # create copy of a grid
                grid_test[y0][x0], grid_test[y1][x1], grid_test[y2][x2],
grid_test[y3][x3] = (1, 1, 1), (1, 1, 1), (
                    1, 1, 1), (1, 1, 1)  # fill the copied grid
                # todo: create a copy of a grid, and put these cords in a
copy

                rating = w1 * min(check_coords, key=lambda x: x[1])[1] + w2 *
get_holes(
                    grid_test) + w3 * calculateBumpiness(grid_test) + w4 *
clearing_piece(grid_test)  # calculate rating

                height_moves[i] = rating  # save position rating

                # reset values
                x_min, x_max, y_min = xb_min, xb_max, yb_min
                x0, x1, x2, x3, y0, y1, y2, y3 = xb0, xb1, xb2, xb3, yb0,
yb1, yb2, yb3

                main_dict[j] = height_moves  # save position rating with
rotation
                grid_test = grid_copy

        # Get rotation and coordinates of the best rating positon
        global max_height, go_coord, main_rotation
        max_height = -1000
        go_coord = None
        main_rotation = None

        for rotation, positions in main_dict.items():
            for coord, height in positions.items():
                if height > max_height:
                    max_height = height
                    go_coord = coord
                    main_rotation = rotation
```

```
        has_executed = True # set that the agent was executed

#################################################################

# Movement of an Agent
def movement(tetromino, rotation, x, coord):
    tetromino.rotation = rotation
    if x < coord:
        pressK("right")
    elif x > coord:
        pressK("left")
    else:
        pressK("down")

# KEY SIMULATIONS #
keyboard = Controller()

def pressKey(button):
    # time.sleep(1)
    keyboard.press(button)
    keyboard.release(button)


def pressK(button):
    pyautogui.press(button)

################################


################ GENETIC ALGORITHMS ###############################

# Global variables
population_size = 50
num_gnerations = 30
mutation_rate = 0.1

# Create individual with random weights
def create_individual():
    return [random.uniform(-10, 10) for _ in range(4)]

# Create first generation
def create_initial_population():
    return [create_individual() for _ in range(population_size)]

# Calculate the fitness
def fitness(individual, generation, individual_idx):
    print("Generation:", generation, 'Individual:', individual_idx,
"Chromosome:", individual)
    return main(win, individual, generation, individual_idx, True)

# Selection operator
def selection(population, fitnesses):
    # Tournament selection
    tournament_size = 5
    parents = []
```

```
    for _ in range(2):  # Select 2 parents
        competitors = random.sample(list(zip(population, fitnesses)),
tournament_size)
        winner = max(competitors, key=lambda x: x[1])
        parents.append(winner[0])
    return parents

# Mutation operator
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] += random.uniform(-0.1, 0.1)

# Crossover operator
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

############################## DATA GATHERING
####################################
def save_population(population, generation):
    filename = f"population_generation_{generation}.txt"

    with open(filename, "w") as f:
        for individual in population:
            weights_str = " ".join(map(str, individual))
            f.write(f"{weights_str}\n")


def save_fitnesses(fitnesses, generation):
    filename = f"fitnesses_generation_{generation}.txt"

    with open(filename, "w") as f:
        for fitness in fitnesses:
            f.write(f"{fitness}\n")


def save_best_individual(individual, generation):
    filename = f"best_individual_generation_{generation}.txt"

    with open(filename, "w") as f:
        weights = individual[0]  # Extract weights from the tuple (weights,
fitness)
        weights_str = " ".join(map(str, weights))
        f.write(f"{weights_str}\n")


def load_population(filename):
    population = []

    with open(filename, "r") as f:
        for line in f:
            weights_str = line.strip().split()
            individual = [float(w) for w in weights_str]
            population.append(individual)
```

```
    return population
#######################################################################



create_bag() # initalise the bag of tetromino pieces
def main(win, individual, generation, ind_idx, choice=True): # Choice for the
game option (if TRUE: GAs and Trained AI, else just normal Tetris)

    # Initial variables
    locked_positions = {}
    grid = create_grid(locked_positions)
    global has_executed
    change_piece = False
    has_executed = False
    run = True
    # Get pieces from the bag
    current_piece = get_shape(get_next_tetromino())
    next_piece = get_shape(get_next_tetromino())

    # Set time, music and initial scores
    clock = pygame.time.Clock()
    fall_time = 0
    fall_speed = 0.37
    level_time = 0
    rows = 0
    score = 0
    moves = 0
    clear_lines = 0
    pygame.mixer.music.play(-1)

    while run:
        grid = create_grid(locked_positions)
        fall_time += clock.get_rawtime()
        level_time += clock.get_rawtime()
        clock.tick()

        # Increase game speed every 5 seconds
        if level_time / 1000 > 5:
            level_time = 0
            if fall_speed > 0.12:
                fall_speed -= 0.005

        if choice:
            value = 0.01
        else:
            value = fall_speed

        # Drop piece from top
        if fall_time / 1000 >= value:
            fall_time = 0
            current_piece.y += 1
            if not (valid_space(current_piece, grid)) and current_piece.y >
0:
                current_piece.y -= 1
                change_piece = True
```

```
        # Check for user input
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
                # pygame.display.quit()

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE: # go to main menu
                    run = False
                    pygame.mixer.music.stop()
                    main_menu(win)
                # MOVEMENT
                if event.key == pygame.K_LEFT:
                    current_piece.x -= 1
                    if not (valid_space(current_piece, grid)):
                        current_piece.x += 1
                if event.key == pygame.K_RIGHT:
                    current_piece.x += 1
                    if not (valid_space(current_piece, grid)):
                        current_piece.x -= 1
                if event.key == pygame.K_DOWN:
                    current_piece.y += 1
                    if not (valid_space(current_piece, grid)):
                        current_piece.y -= 1
                if event.key == pygame.K_UP:
                    current_piece.rotation += 1
                    if not (valid_space(current_piece, grid)):
                        current_piece.rotation -= 1
                if event.key == pygame.K_SPACE:
                    while True:
                        current_piece.y += 1
                        if not (valid_space(current_piece, grid)):
                            current_piece.y -= 1
                            break

        ############ INITIATE AGENT
################################################

        # Get piece coordinates
        pos = convert_shape_format(current_piece)
        x1, x2, x3, x4 = pos[0][0], pos[1][0], pos[2][0], pos[3][0]
        x_min = min(x1, x2, x3, x4)

        # Run agent
        if choice:
            if current_piece.y > 2:
                agent(current_piece, grid, individual[0], individual[1],
individual[2], individual[3])
                movement(current_piece, main_rotation, x_min, go_coord)


################################################################################
##

        shape_pos = convert_shape_format(current_piece)
```

```
        # Fill grid with colors according to the piece coordinate
        for i in range(len(shape_pos)):
            x, y = shape_pos[i]
            if y > -1:
                grid[y][x] = current_piece.color

        # Give next piece
        if change_piece:
            for pos in shape_pos:
                p = (pos[0], pos[1])
                locked_positions[p] = current_piece.color
            current_piece = next_piece
            next_piece = get_shape(get_next_tetromino())
            change_piece = False
            has_executed = False

            # Set score
            moves += 1
            rows = clear_rows(grid, locked_positions)
            clear_lines += rows
            if rows == 1:
                score += 40
            elif rows == 2:
                score += 100
            elif rows == 3:
                score += 300
            elif rows == 4:
                score += 1200
            else:
                score == 0

        # Draw everything
        draw_window(win, grid, score, clear_lines, generation, ind_idx,
choice)
        draw_next_shape(next_piece, win)
        pygame.display.update()

        # Check for the game over
        if choice:
            if check_lost(locked_positions) or moves > 1150:
                draw_text_middle(win, "You Lost!", 80, (255, 255, 255))
                pygame.mixer.music.stop()
                pygame.display.update()
                pygame.time.delay(1500)
                if clear_lines == 0:
                    return 0
                else:
                    return clear_lines
                run = False
                # main(win)
        else:
            if check_lost(locked_positions):
                draw_text_middle(win, "You Lost!", 80, (255, 255, 255))
                pygame.mixer.music.stop()
                pygame.display.update()
                pygame.time.delay(1500)
                run = False
```

```
# MAIN MENU

def main_menu(win):
    run = True
    while run:
        win.fill((0, 0, 0))
        draw_text_middle(win, 'Press [1] Tetris [2] Trained AI [3] GAs
Process', 30, (255, 255, 255))
        pygame.display.update()
        # THREE OPTIONS
        for event in pygame.event.get():
            # IF QUIT
            if event.type == pygame.QUIT:
                run = False
                pygame.mixer.music.stop()
            if event.type == pygame.KEYDOWN:
                # Normal Tetris
                if event.key == pygame.K_1:
                    main(win, None, None, None, False)
                # Trained Agent
                if event.key == pygame.K_2:
                    player = [0.5436822764440379, -9.328911430903139, -
1.735608284805026, 6.577302970502618]
                    main(win, player, 13, 1, True)
                # Genetic Algorithms
                if event.key == pygame.K_3:
                    population = create_initial_population() # create first
generation
                    # save_population(population, 0)
                    # iterate through number of generations
                    for generation in range(num_gnerations):
                        # get fitness scores
                        fitnesses = [fitness(individual, generation, i) for
i, individual in enumerate(population)]
                        # save_fitnesses(fitnesses, generation)

                        # get best individual in a generation
                        best_individual = max(zip(population, fitnesses),
key=lambda x: x[1])
                        # save_best_individual(best_individual, generation)

                        # Print best individual
                        print(
                            f"Generation {generation}: Best fitness =
{best_individual[1]}, Weights = {best_individual[0]}")

                        # Create new population
                        new_population = []

                        # Perform evolution
                        while len(new_population) < population_size:
                            # Selection
                            parent1, parent2 = selection(population,
fitnesses)

                            # Crossover
```

```
                            offspring1, offspring2 = crossover(parent1,
parent2)

                            # Mutation
                            mutate(offspring1, mutation_rate)
                            mutate(offspring2, mutation_rate)

                            new_population.extend([offspring1, offspring2])
                        population = new_population
                        # save_population(population, generation)

    pygame.display.quit()

# Surfaces
win = pygame.display.set_mode((s_width, s_height))
pygame.display.set_caption("Tetris")
main_menu(win)  # start game
```