Fast Tetris

Project Members:

- Shaaf Salman (ID: 21L6083)
- Abdul Hadi (ID: 21L6077)
- Haider Khan (ID: 21L6067)

Project Evaluation and Analysis Report

1. Introduction

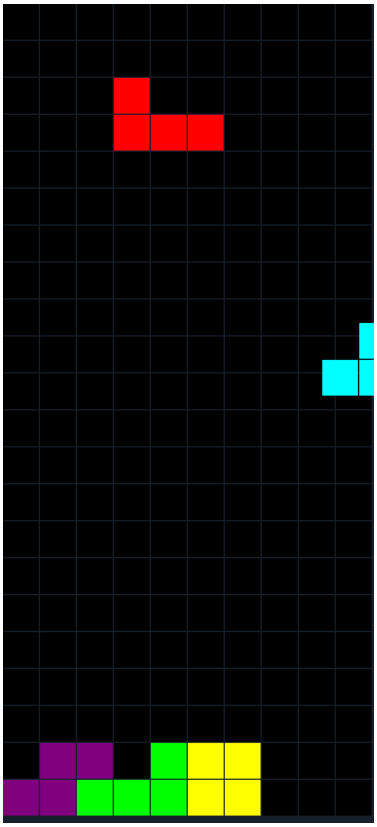
The project aims to develop a Genetic Algorithm (GA) for evolving players in a Tetris-like game. The GA iteratively refines a population of players by selecting the fittest individuals, applying genetic operators such as crossover and mutation, and persisting the state to enable the resumption of the evolution process. This report provides a detailed evaluation and analysis of the project's structure, logic, and implementation.

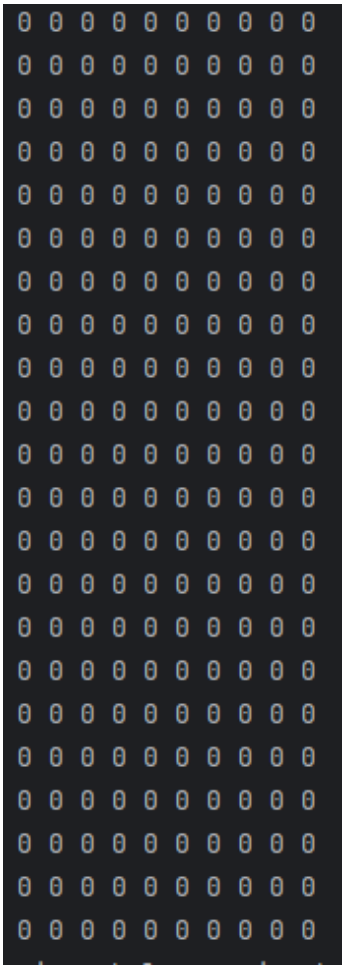
2. Structure and Logic

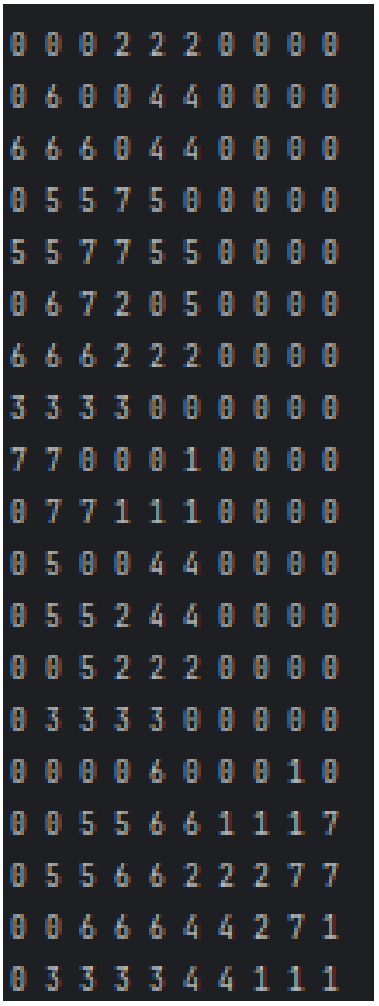
2.1 Class Architecture

- **Player:** Represents an individual player in the game, characterized by attributes such as height, lines cleared, holes, blockades, and score.
- **Game:** Models the Tetris-like game environment, providing methods for game logic, state management, and scoring.
- **Renderer:** Handles rendering of the game environment, including rendering player actions, game state, and graphical elements.

2.2 Grid







2.3 Block

```
4 usages  Shaaf Salman
class LBlock(Block):
    """
    Class representing the L-shaped Tetris block.
    """

    Shaaf Salman
    def __init__(self):
        super().__init__(id=1)
        self.cells = {
            0: [Position( row: 0,  col: 2), Position( row: 1,  col: 0), Position( row: 1,  col: 1), Position( row: 1,  col: 2)],
            1: [Position( row: 0,  col: 1), Position( row: 1,  col: 1), Position( row: 2,  col: 1), Position( row: 2,  col: 2)],
            2: [Position( row: 1,  col: 0), Position( row: 1,  col: 1), Position( row: 1,  col: 2), Position( row: 2,  col: 0)],
            3: [Position( row: 0,  col: 0), Position( row: 0,  col: 1), Position( row: 1,  col: 1), Position( row: 2,  col: 1)]
        }
        self.name = "L"
        self.num_rotations = 4
        self.move( rows: 0,  columns: 3)
```

3 Genetic Algorithm (GA)

3.1 Introduction

The Genetic Algorithm (GA) implemented in this project aims to evolve players in a Tetris-like game environment. Through iterative refinement of a population of players, the GA selects the fittest individuals based on predefined criteria, applies genetic operators such as crossover and mutation to generate offspring, and continues the evolution process indefinitely until the GA reaches a mature state capable of playing endlessly.

3.2 Summary of GA

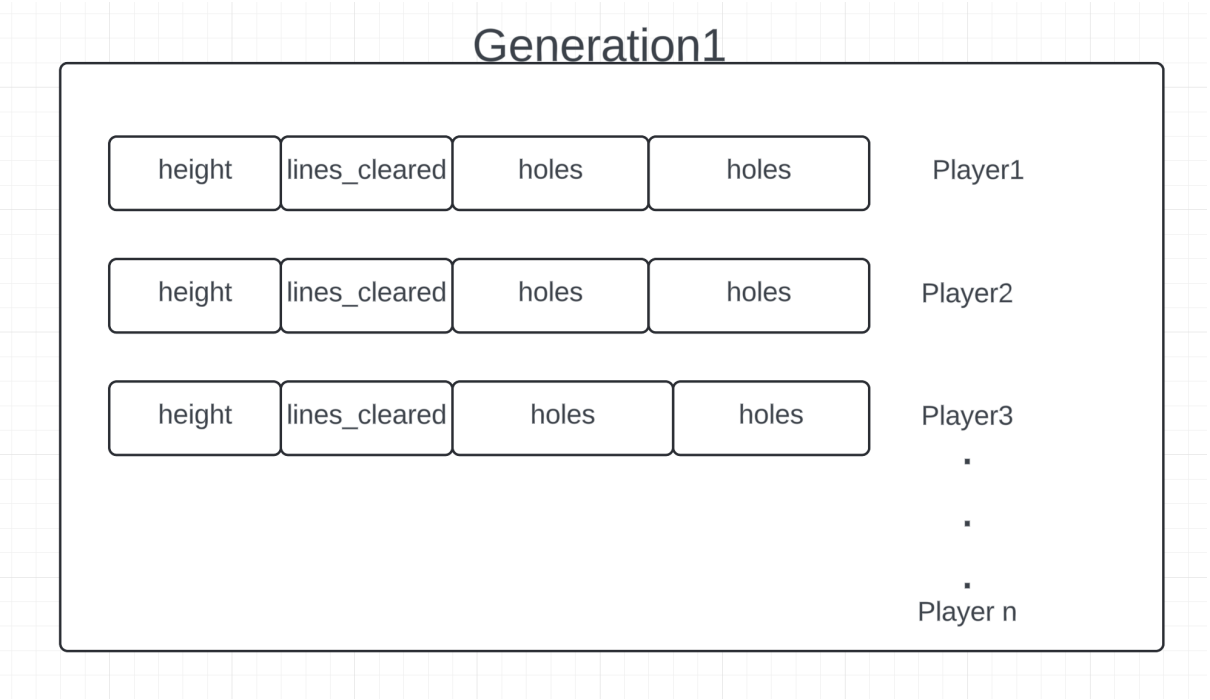
Initialisation: The GA initiates players with random weights for attributes such as height, lines cleared, holes, and blockades. These weights are crucial in determining player behaviour and strategy during gameplay.

3.3.1 Generation Process:

Each generation consists of 10 players, and the GA evolves them using mutation and crossover operators. The process involves selecting the top-performing players, performing crossover between parents, and introducing randomness through mutation.

3.3.2 Gene Information

Example of what a generation looks like:



3.4 Configurable Environment:

The GA's environment is configured using parameters specified in the `config.py` file, allowing for flexibility and customisation of the evolutionary process.

```
class GAConfig:
    population_size = 10
    mutation_rate = 0.1
    crossover_rate = 0.5
    num_generations = 30
```

3.5.1 Very First Population

The GA initialises the first generation with 10 players, assigning random weights for each player's attributes within specified ranges.

3.5.2 Weights Sign Formation

```
     Lines Cleared ------------------------------------> pos
     Height        ------------------------------------> neg
     Blocakdes     ------------------------------------> neg
     Holes/Gaps    ------------------------------------> neg
```

3.5.3 Score Calculation

```
def initialize_population(self):
    self.population = []
    for _ in range(self.population_size):
        height_weight = uniform(-15.0, 0.0)
        lines_cleared_weight = uniform(0.0, 15.0)
        holes_weight = uniform(-15.0, 0.0)
        blockades_weight = uniform(-15.0, 0.0)


        player =Player(height_weight,lines_cleared_weight,h
oles_weight,blockades_weight)
        self.population.append(player)
```

3.6 Generation Process

3.6.1 Player Moves:

Each player in the generation is simulated in the game environment, making moves based on their assigned weights for attributes the moves calculation which will be discussed in detail afterwards.

3.6.2 Selection:

The GA selects the top-scoring players from the current generation to proceed to the next generation.

3.6.3 Crossover:

Crossover is performed between selected parent players to generate offspring with combined traits. selects a random crossover point and then performs crossover between two best players in a population.

```
1 usage
def generate_crossover_players(num_players, top_players):
    crossover_players = []
    for _ in range(num_players):
        parent1, parent2 = random.sample(top_players, k: 2)
        crossover_point = random.randint( a: 1, len(parent1) - 1)
        height_weight = parent1[:crossover_point] + parent2[crossover_point:]
        lines_cleared_weight = parent2[:crossover_point] + parent1[crossover_point:]
        holes_weight = parent1[crossover_point:] + parent2[:crossover_point]
        blockades_weight = parent2[crossover_point:] + parent1[:crossover_point]
        player = Player(height_weight, lines_cleared_weight, holes_weight, blockades_weight)
        crossover_players.append(player)
    return crossover_players
```

3.6.4 Mutation:

Random mutation is applied to introduce variability and prevent premature convergence. This is done by randomly changing weights of a player's attributes.

```
    1 usage
    def mutate_population(self):
        mutation_count = int(len(self.population) * self.mutation_rate)
        for _ in range(mutation_count):
            index = random.randint( a: 0, len(self.population) - 1)
            player = self.population[index]

            player.height_weight += random.uniform(-15.0,  b: 0.0)
            player.lines_cleared_weight += random.uniform( a: 0,  b: 15.0)
            player.holes_weight += random.uniform(-15.0,  b: 0.0)
            player.blockades_weight += random.uniform(-15.0,  b: 0.0)
```

3.7 After Process

After all players in the current generation have played, the GA initializes the next generation by selecting top players, performing crossover, and introducing random players.

4. Path Searching Algorithm and Heuristic Calculation

4.1 Continual

After players of each generation play the game, the Genetic Algorithm (GA) proceeds to the player class, where it utilizes the path searcher class to generate all possible moves.

4.2 Path Searcher Class

The path searcher class takes the game instance and player weights as input. It then calculates all possible moves based on the current game state and logic.

what we have input form GA

“All weighs of the player and Game Instance”

4.2.3 All Possible Moves


User Weights and game Instance
1
Get All paths from fucntion
2
apply 1 path on grid and then calculate Constraints and store them
repeat for all paths
Path searcher
3
Pass all paths with their rank to Player
4
User chooses path with maximum rank
5
Path is returned to GA and then updated game instance again comes to Path Searcher

Repeat 5 step for every rotation of current block

Step 1 :

Move to very left top corner

Step 2 :

Go Down till collapses

Step 3 :

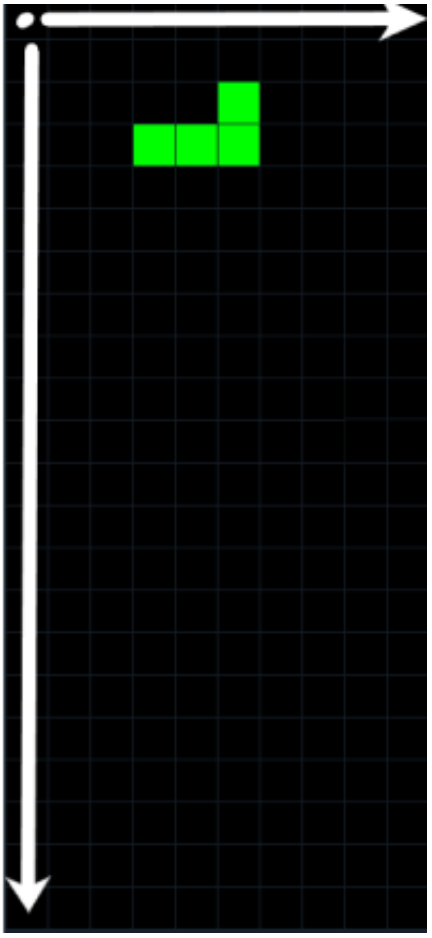
move one cell right

Step 4 :

Go Down again till collapses

Step 5 :

repeat till you reach very right



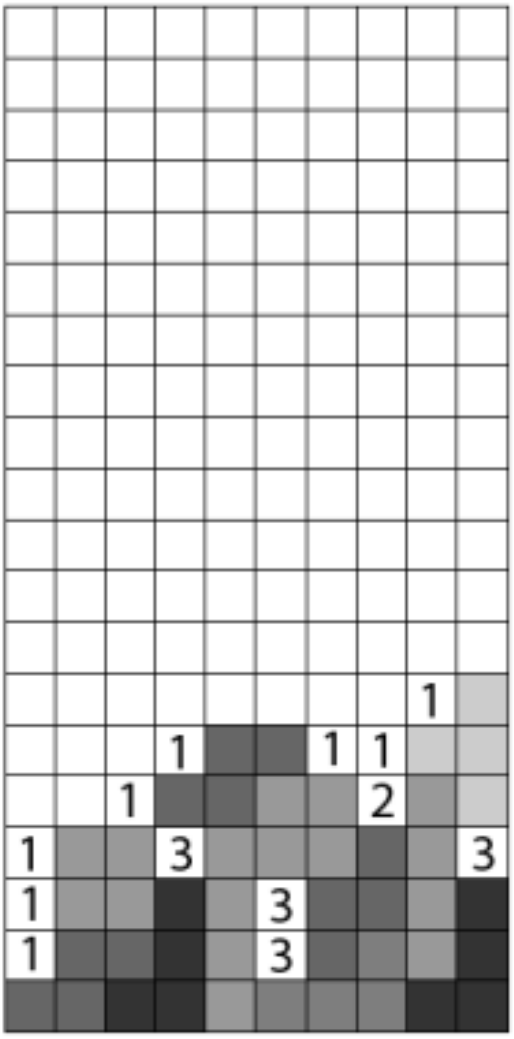
```
for rotation in range(game.current_block.number_of_rotations):
    for col in range(game.grid.num_cols):
        print("The column is ---------------------")
        print(col)
        current_grid = grid.copy()
        game_copy = game.copy()
        # game.current_block.print_details()
        moves = calculate_block_moves(current_grid, col, rotation)

        holes, blockades, full_rows, max_height = game_copy.apply_moves_to_grid(moves)

        path = self.create_path(moves, holes, blockades, full_rows, max_height)
        self.paths.append(path)
```

4.3 Gene Calculation

After applying a move, the path searcher calculates various attributes, or genes, of the player. These include:

4.3.2 Holes

according to game analytics and it plays a major role so by some research and break down we came up to 3 type of holes



4.3.3 Blockades

any cell of the block if acts to assist in creation of a block it is termed as a blockade

4.3.4 Height

the maximum height of any tower in a grid after placing the block

4.3.5 Lines Cleared

lines cleared in the grid if that move is placed

4.4 Rank Calculation

The path searcher then computes the rank of each move based on the player weights, assigning a score to each move. It returns a list of paths ranked by score, allowing the player to choose the optimal move.

4.5 Limitations

Despite its effectiveness, the current implementation may have limitations, such as reliance on simplified heuristics and computational complexity.

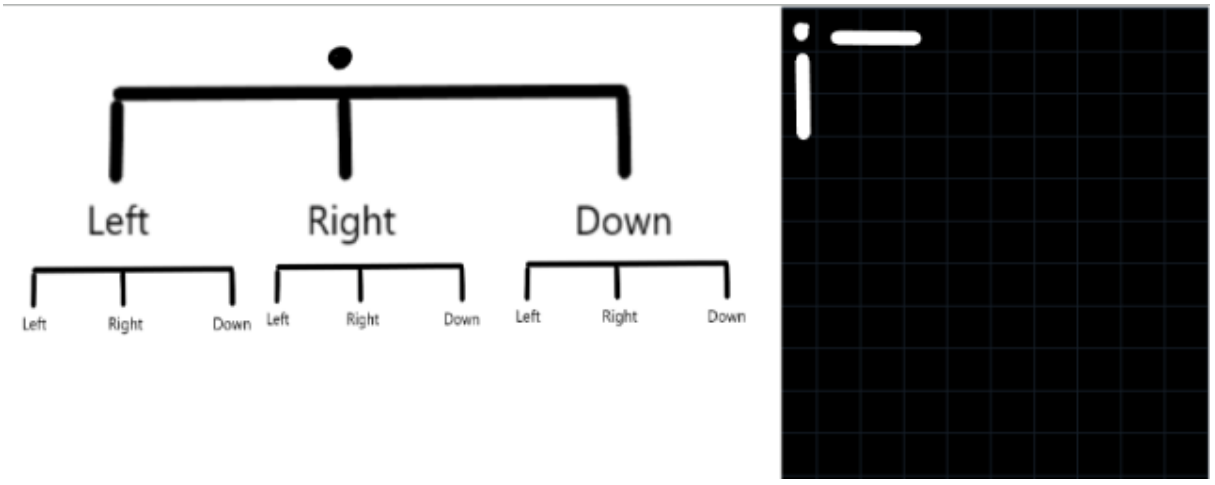
Code Prompt and Images

[Insert relevant code snippets and images here]

5. Ideal Scenario

5.1 A* Implementation

In an ideal scenario, the GA would implement the A* algorithm to calculate all possible moves of the grid, treating it as a tree structure. This approach would provide more accurate and efficient path searching.



5.2 Next Block Storage Technology

Future enhancements could involve extending the path searcher to calculate moves for both the current block and the next block. This improvement would enhance strategic planning and overall gameplay.

6. Analysis of Current Situation

The current implementation performs well, achieving an average score of approximately 250 to 300. After around 10 generations, scores begin to exceed 800 to 1000, indicating the effectiveness of the GA in improving player performance over time.

7. Conclusion