Explain the algorithm in the code :

1. **fitness_function(env, population)**: This function calculates the fitness scores for each chromosome in the population. It iterates over the population and runs the **gen_scoring_function** for each chromosome using the provided environment (**env**). The total score for each chromosome is appended to a list and returned as the fitness scores.
2. **Genetic** class:

- **__init__(self, sample_space, pop_size, chrom_len, mutation_rate=0.3, crossover_rate=0.5)**: The constructor initializes the Genetic class with the sample space, population size, chromosome length, mutation rate, and crossover rate. It generates an initial population based on the provided parameters and divides it into two subpopulations **X** and **Y**.
- **_generate_chromosome(self, length, sample)**: This private method generates a single chromosome of a given length from the provided sample space. It randomly selects elements from the sample space without replacement.
- **_generate_population(self, pop_size, chrom_size, sample_space)**: This private method generates the initial population by creating multiple chromosomes using the **_generate_chromosome** method. It returns the generated population.
- **_divide(self, population)**: This private method divides the population into two subpopulations, **X** and **Y**, by shuffling the indices and splitting them into halves.
- **_generate_new_generation(self, x_chroms, y_chroms)**: This private method generates a new generation by performing crossover between chromosomes from the **X** and **Y** subpopulations. It randomly selects chromosomes from both subpopulations, swaps the first half of their genes, and returns the new chromosomes.
- **_crossover(self, x, y)**: This private method performs the crossover operation by swapping the first half of the genes between chromosomes **x** and **y**. It returns the modified chromosomes.
- **_fitness_rank(self, chroms, env)**: This private method ranks the chromosomes based on their fitness scores. It uses the **fitness_function** to calculate the fitness scores for each chromosome and sorts the chromosomes based on these scores. The sorted chromosomes are returned.
- **_mutate(self, chroms)**: This private method introduces mutations in the chromosomes. It randomly selects genes from the chromosomes and adds a random value between -2 and 2 to the selected genes. It returns the mutated chromosomes.

- **run(self, env, num_generations, num_mutation, mu_gen, epochs_per_generation)**: This method executes the genetic algorithm optimization. It iterates over the specified number of generations and performs epochs within each generation. It ranks the chromosomes based on their fitness scores, generates a new generation through crossover, and introduces mutations at specified intervals. The method returns the best chromosomes from subpopulations **X** and **Y** after the optimization process.

The code aims to optimize a population of chromosomes using a genetic algorithm by performing selection, crossover, and mutation operations. It uses fitness scores to rank the chromosomes and generates new generations to improve the overall fitness of the population.

explanation of the main parts:

1. Setting up the environment and parameters:
    - The **env** variable is initialized as an instance of the **TetrisEnv** class, representing the Tetris game environment.
    - Various parameters are set, such as **population_size**, **num_generations**, **mutation_rate**, **use_visuals_in_trace**, **sleep_time**, **chrom_len**, **crossover_rate**, **num_mutation**, **mu_gen**, **epochs_per_generation**, and **sample_space**.
2. Creating an instance of the Genetic class:
    - An instance of the **Genetic** class is created with the provided parameters, including the **sample_space** which represents the range of values for the chromosomes.
3. Running the genetic algorithm:
    - The **run** method of the **Genetic** class is called with the specified parameters to execute the genetic algorithm optimization.
    - The final population of chromosomes is stored in the **final_population** variable.
4. Testing the chromosomes:
    - The best two chromosomes from the final population are selected and used to run the Tetris game using the **genetic_scoring_function**.

- The results, including the ratings, rotations, total score, and any messages, are printed for each chromosome.

5. Evolutionary steps:
    - After testing all the chromosomes and obtaining the results, the code suggests evolving new chromosomes based on the best-performing ones. However, this part is currently commented out.

This code allows us to run a genetic algorithm to optimize the performance of Tetris-playing chromosomes. It iterates over a specified number of generations, performs crossover and mutation operations, and evaluates the fitness of the chromosomes using the Tetris game environment. The best-performing chromosomes are selected and their performance is examined.

explanation of the score function:

1. Function signature:
    - The function **gen_scoring_function** takes three parameters:
        - **tetris_env**: An instance of the **TetrisEnv** class, representing the Tetris game environment.
        - **gen_params**: A list or array containing the genetic parameters used to calculate the score.
        - **col**: The column index where the current piece will be placed.
2. Obtaining the game status:
    - The **board**, **piece**, and **next_piece** are obtained from the Tetris environment using the **get_status()** method.
3. Scoring calculations:
    - The function iterates over the possible rotations of the current piece (0 to 3) and evaluates the score for each rotation.
    - A copy of the current **board** is created to simulate the placement of the piece and calculate the score.

- The **tetris_env.test_play()** method is called with the current **board**, **piece**, **col**, and rotation index (**i**) to obtain the score and the updated board after placing the piece.
- If the score is negative, a penalty is applied (**abs(score * gen_params[4]) * -10000000000**) to discourage such moves, and the score is added to the **scores** list along with the rotation index (**i**).
- If the score is non-negative, various board information metrics (e.g., **agg_height**, **n_holes**, **bumpiness**, **cleared**, **num_pits**, **max_wells**, **n_cols_with_holes**, **row_transitions**, **col_transitions**) are calculated using the **get_board_info** function. These metrics are multiplied by their respective genetic parameters (**gen_params**) and added to the score.
- The final score for each rotation is stored in the **scores** list as **[score, i]**.

4. Selecting the best score:
   - The maximum score and its associated rotation index are determined using the **max** function and the **key** parameter to compare the first element of each score sublist (**item[0]**).
   - The maximum score value is stored in the **val** variable.
5. Returning the score and rotation index:
   - The function returns the maximum score value (**val[0]**) and its associated rotation index (**val[1]**).

This scoring function evaluates the fitness of a particular Tetris board configuration by considering various metrics and applying genetic parameters to calculate a score. The function helps in the optimization process of the genetic algorithm by guiding the selection of rotations for placing Tetris pieces.

explanation of the params file:

1. **get_board_info(area, next_board)**:
   - This function takes two parameters: **area** and **next_board**, which represent the game board.

- It calculates and returns various metrics related to the board, including:
    - **agg_height**: The aggregated height of all columns (sum of column heights).
    - **n_holes**: The number of empty spaces (holes) in the board.
    - **bumpiness**: The sum of the absolute height differences between consecutive columns.
    - **cleared**: The number of lines cleared in the current move.
    - **num_pits**: The number of columns with no blocks (empty columns).
    - **max_wells**: The depth of the deepest well (an empty space surrounded by blocks).
    - **n_cols_with_holes**: The number of columns with at least one hole.
    - **row_transitions**: The number of transitions between filled and empty cells in each row.
    - **col_transitions**: The number of transitions between filled and empty cells in each column.

2. **get_peaks(area)**:
    - This function takes the game board **area** as input.
    - It calculates and returns the heights of the highest blocks in each column.
    - The function iterates over each column and finds the first occurrence of a block (**1**) from the bottom of the column. The height of the block is determined by subtracting its position from the total number of rows.
    - The heights of the highest blocks are stored in the **peaks** array and returned.
3. **get_row_transition(area, highest_peak)**:
    - This function calculates the number of transitions between filled and empty cells in each row.
    - It takes the game board **area** and the **highest_peak** (the height of the highest block in the board) as inputs.
    - The function iterates over each row from the row below the highest peak to the bottom of the board and checks for transitions between adjacent cells in the same row.

- Each transition increments a counter, which is returned as the result.

4. **get_col_transition(area, peaks)**:
    - This function calculates the number of transitions between filled and empty cells in each column.
    - It takes the game board **area** and the **peaks** array (heights of the highest blocks in each column) as inputs.
    - The function iterates over each column and checks for transitions between adjacent cells in the same column.
    - Transitions are counted for rows between the highest peak and the second-to-last row.
    - The total number of column transitions is returned.
5. **get_bumpiness(peaks)**:
    - This function calculates the bumpiness metric, which represents the sum of the absolute height differences between consecutive columns.
    - It takes the **peaks** array (heights of the highest blocks in each column) as input.
    - The function iterates over each column (excluding the last column) and calculates the absolute difference between the height of the current column and the height of the next column.
    - The absolute differences are summed and returned as the bumpiness value.
6. **get_holes(peaks, area)**:
    - This function calculates the number of empty spaces (holes) in each column of the game board.
    - It takes the **peaks** array (heights of the highest blocks in each column) and the game board **area** as inputs.
    - The function iterates over each column and counts the number of empty cells (**0**) below the highest block (**peak**) in that column.
    - The number of holes in each column is stored in a list and returned.
7. **get_wells(peaks)**:
    - This function calculates the depth of the deepest well in each column.

- It takes the **peaks** array (heights of the highest blocks in each column) as input.
- The function iterates over each column and checks the heights of the adjacent columns.
- If the current column is at either end, the difference between the adjacent column height and the current column height is calculated. Otherwise, the maximum difference between the heights of the two adjacent columns is considered.
- The depths of the deepest wells in each column are stored in a list and returned.

8. **get_cleared_lines(current_board, next_board)**:
   - This function calculates the number of lines cleared in the current move.
   - It takes two game boards as inputs: **current_board** represents the board before the move, and **next_board** represents the board after the move.
   - The function counts the number of rows with all cells cleared (filled with **0**) in the **current_board** and the **next_board**.
   - The difference between the counts of cleared rows in the **next_board** and the **current_board** is returned.

These functions provide various metrics and information about the Tetris game board, which are used in the scoring function to evaluate the fitness of different board configurations and guide the optimization process.

The two best chromo :

[-65.74792661, -34.75471346, -27.49264635, -34.75471346, -78.82680045, -65.77022113, -35.54476263, -9.71014655]

[-66.65968282, -34.75471346, -27.49264635, -34.75471346, -76.13964912, -78.40617174, -78.58290884, -64.79227041]

Env Features :

agg_height / n_holes / bumpiness / row_transitions / score / num_pits / n_cols_with_holes / cleared

seed : 39

score for first chromo (Train): 173800

score for second chromo (Train): 173800

score for first chromo (Test): 192400

score for second chromo (Test): 192400