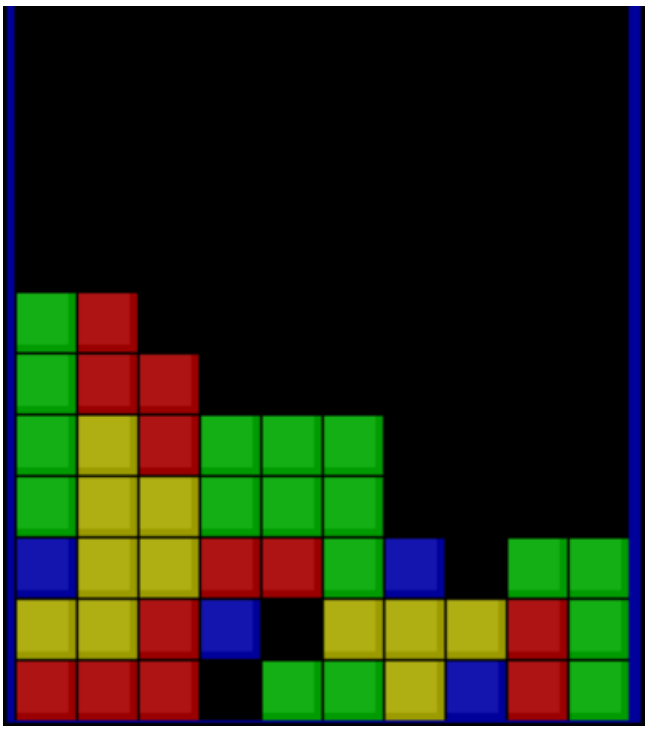
Cairo University

Faculty of Computers & Artificial Intelligence

Cognitive Science Project Report

Task 1 - Tetris solving genetic algorithm



08 May 2024

Proudly Done By

Omar Kouta	20220528
Ahmed Wessam	20210046
Mohammed Mohsen	20210353
Abdelrahman Fawzy	20210527
Khaled Waleed Alshaer	20210127

Reports and papers don’t have to be dull.

Index

1. Purpose

This Report was written as requested in the project statement, our code was written to solve the problem of finding a good algorithm to play Tetris without using machine learning, we explored genetic algorithms and wrote code to solve the problem.

2. Our Work

2.1 Introduction

The classical game of Tetris where you put blocks on top of one another to create a rectangular structure within a particular area or you lose.

We used a genetic algorithm as requested. We built a virtual chromosome where a chromosome here represents a player, each chromosome has some attributes of which the most important are the genes, genes are just numbers that act like weights that calculate the score for a move based on the board heuristics.

2.2 Solution

We have created 2 classes:

- Chromosome class which houses the genes and contains other methods that help in deciding the best move based on the current board, current piece, and the next piece.
- GeneticAlgorithm class is responsible for evolving the population to fit the current problem best, making a Tetris AI.

Chosen Values:

- **Percentage of Mutation: 20%**
- **Percentage of Selection: 35%**
- **Population Size: 15**
- **Chosen Seed: 42**

Genetic Algorithm Implementation:

1. **Fitness Calculation**: the fitness score for each chromosome in the current population is the score that the chromosome gets after ending a full Tetris game whether that would be winning or losing.

```
def cal_fitness(self):
    # Run the game for each chromosome and calculate the fitness score
    for chrom in self.chromosomes:
        chrom_score = tb.run_game_ai(is_training=True,chromosome=chrom,max_score=self.MAX_SCORE)
        chrom.update_fitness_score(game_score=chrom_score)
```

2. **Selection:** in the selection process we choose the top *30%* out of the population after sorting.

```
def selection(self):
    # Update the chromosome list (delete the rest) ceil(Top 30%)
    self.chromosomes = sorted(self.chromosomes, key=lambda x: x.fitness_score, reverse = True)
    self.chromosomes = self.chromosomes[0:max(2, int(math.ceil(len(self.chromosomes) * self.PERECENTAGE_OF_SELECTION)))]
```

3. **Crossover**: we simply get all possible combinations between two parents and then get the weighted average of their genes based on their fitness score.

```
def crossover(self):
    # Return new chromosomes
    off_chroms = []
    num_of_offspring = self.NUM_OF_CHROMOSOMES - len(self.chromosomes)
    while (len(off_chroms) < num_of_offspring):
        for i in range(len(self.chromosomes)):
            if(len(off_chroms) >= num_of_offspring):
                break
            for j in range(i+1,len(self.chromosomes)):
                if(len(off_chroms) >= num_of_offspring):
                    break
                off_chroms.append(self.mating(self.chromosomes[i],self.chromosomes[j]))
    return off_chroms

def mating(self,chrom1,chrom2):
    temp_gene_p1 = chrom1.genes
    temp_gene_p2 = chrom2.genes
    total_fitness = chrom1.fitness_score + chrom2.fitness_score
    off_gene = np.array(temp_gene_p1)*(chrom1.fitness_score/total_fitness) \
             + np.array(temp_gene_p2)*(chrom2.fitness_score/total_fitness)
    off_gene = off_gene.tolist()
    return Chromosome(heuristic_names=self.HEURISTIC_NAMES,genes=off_gene,calculate_next_move=self.NEXT_MOVE)
```

4. **Mutation**: randomly choose the genes we are going to mutate based on the mutation percentage then mutate the gene by generating a random number between 0.1 and -0.1.

```
def mutate(self,off_chroms):
    # mutated chromosomes
    # Set of off chromosomes
        # eliminate randomly
    num_of_mutated = int(len(off_chroms) * self.PERECENTAGE_OF_MUTATED)
    for i in range(num_of_mutated):
        randi = random.randint(0,(self.NUM_OF_GENES * len(off_chroms))-1)
        off_chroms[randi//self.NUM_OF_GENES].genes[randi % self.NUM_OF_GENES] += random.uniform(-0.1,0.1)
    return off_chroms
```

Calculated Heuristics:

1. **Aggregate Height**: the total of the height of each column
2. **Completed Line**: the number of completed lines
3. **Number of Holes**: the number of holes
4. **Bumpiness**: the measure of how bumpy the surface is by calculating the absolute difference between each column and its adjacent one.
5. **Hole Segments**: the number of independent hole segments in each column
6. **Max Height:** the max height from the entire board
7. **Empty Columns:** the number of empty columns
8. **One Rows:** the number of rows that are full except in one position
9. **Two Rows:** the number of consecutive two rows that are full except in one position that they both share
10. **Three Rows:** the number of consecutive three rows that are full except in one position that they both share

11. **Four Rows:** the number of consecutive four rows that are full except in one position that they both share

Calculation of the Best Move:

We calculate the best move by iterating through all possible combinations of column position and rotation for the given piece. While doing so, we also calculate the best move of the next piece after the current move by calling the function again but on a new board.

```
def best_play(self,board,piece,next_piece):
    '''
    Returns the best play for the current piece and the score of the play
    '''
    best_rotation = 0
    best_column = -2 #
    play_score = -100000

    for column in range(-2, tb.BOARDWIDTH - 2):
        for rot in range(len(tb.PIECES[piece['shape']])):
            # use calc_move_data
            move_data = tb.calc_move_data(board, piece, column, rot, heuristics_names=self.heuristic_names)

            #if the move is valid
            if move_data['is_valid']:
                # calculate the score for each rotation and return the best rotation and its score
                temp_score = np.array(self.genes)
                temp_score = np.dot(temp_score, np.array(list(move_data['cal_data'].values())))

                if(next_piece != None and self.NEXT_MOVE):
                    next_move_data = self.best_play(move_data['new_board'],next_piece,None)
                    temp_score += next_move_data['score']

                if temp_score > play_score:
                    play_score = temp_score
                    best_rotation = rot
                    best_column = column

    piece['rotation'] = best_rotation
    piece['x'] = best_column
    piece['y'] = 0

    return {'best_column':best_column,'best_rotation': best_rotation, 'score': play_score}
```

2.3 Experimenting

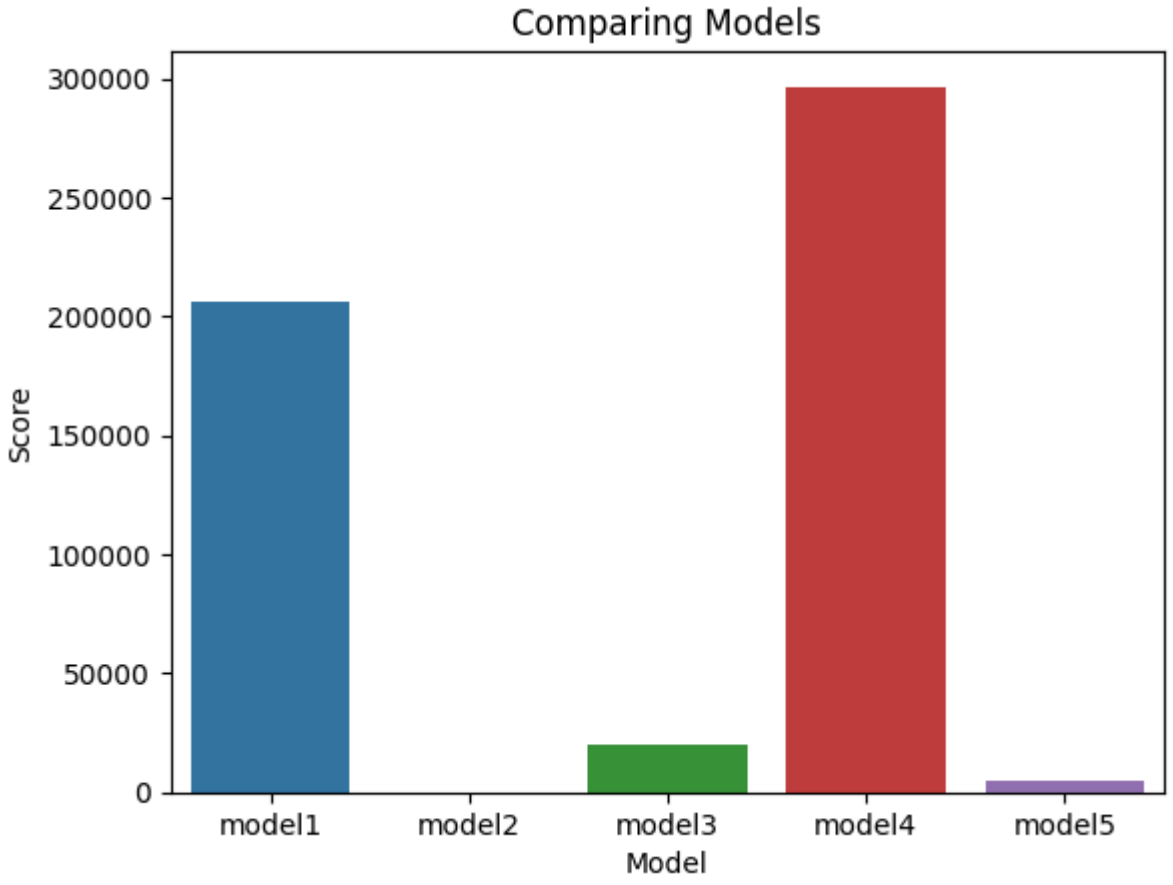
Our winning condition is for the chromosome to get a score of 500,000, which we have surprisingly beat by a huge margin.

We started by creating 5 different models.

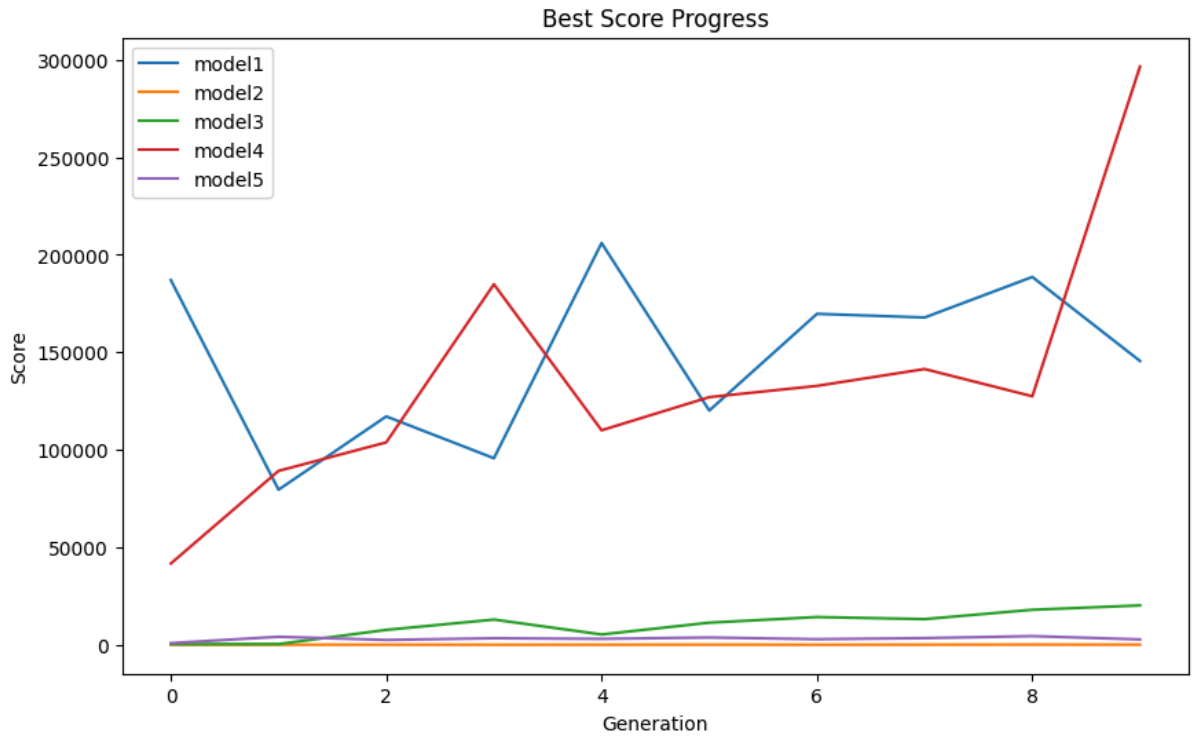
1. **Model 1**:
    a. Chosen Heuristics: `['aggregate_height', 'complete_lines', 'holes', 'bumpiness']`
    b. Calculate Next Move: `False`

2. **Model 2**:
    a. Chosen Heuristics: `['one_rows', 'two_rows', 'three_rows', 'four_rows']`
    b. Calculate Next Move: `False`
3. **Model 3**:
    a. Chosen Heuristics: `['aggregate_height', 'complete_lines', 'holes','bumpiness',’hole_segment’,’max_height’,’empty_columns’,'one_rows', 'two_rows', 'three_rows', 'four_rows']`
    b. Calculate Next Move: `False`
4. **Model 4**:
    a. Chosen Heuristics: `['aggregate_height', 'complete_lines', 'holes', 'bumpiness',’four_rows’]`
    b. Calculate Next Move: `False`
5. **Model 5**:
    a. Chosen Heuristics: `['aggregate_height', 'complete_lines', 'holes','bumpiness',’hole_segment’,’max_height’,’empty_columns’,'one_rows', 'two_rows', 'three_rows', 'four_rows']`
    b. Calculate Next Move: `True`

With the following best scores:



The following best score progress:



That got us to conclude that the following heuristics: `['aggregate_height', 'complete_lines', 'holes', 'bumpiness','four_rows']` had the highest potential. However, we still didn't break our intended goal. Therefore, we thought of powering up both the best values in Mode 1 and Model 4 and allowing it to use the next piece while deciding which move to play next.

So here is what we did:

- Got the best 2 Chromosome from Model 4 and rerun them but with calculated next move TRUE:

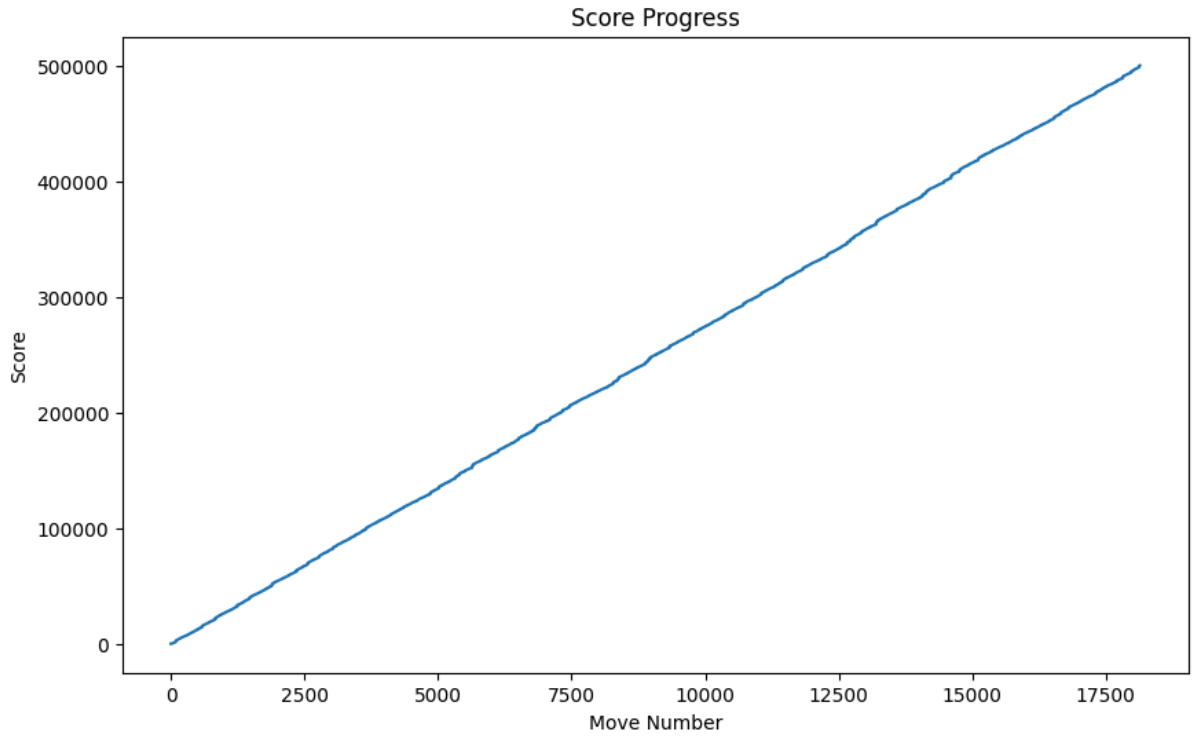
```
[5.522078466021565,3.865153526785135,-3.5865905986090034,-6.
719837436623738,8.408346626937025] with Score: 296465

[5.387269303109375,3.8794766494803907,-3.6804923867413235,-6
.709630184000435,8.339967392082738] with Score: 141408
```

Results:

First Chromosome:

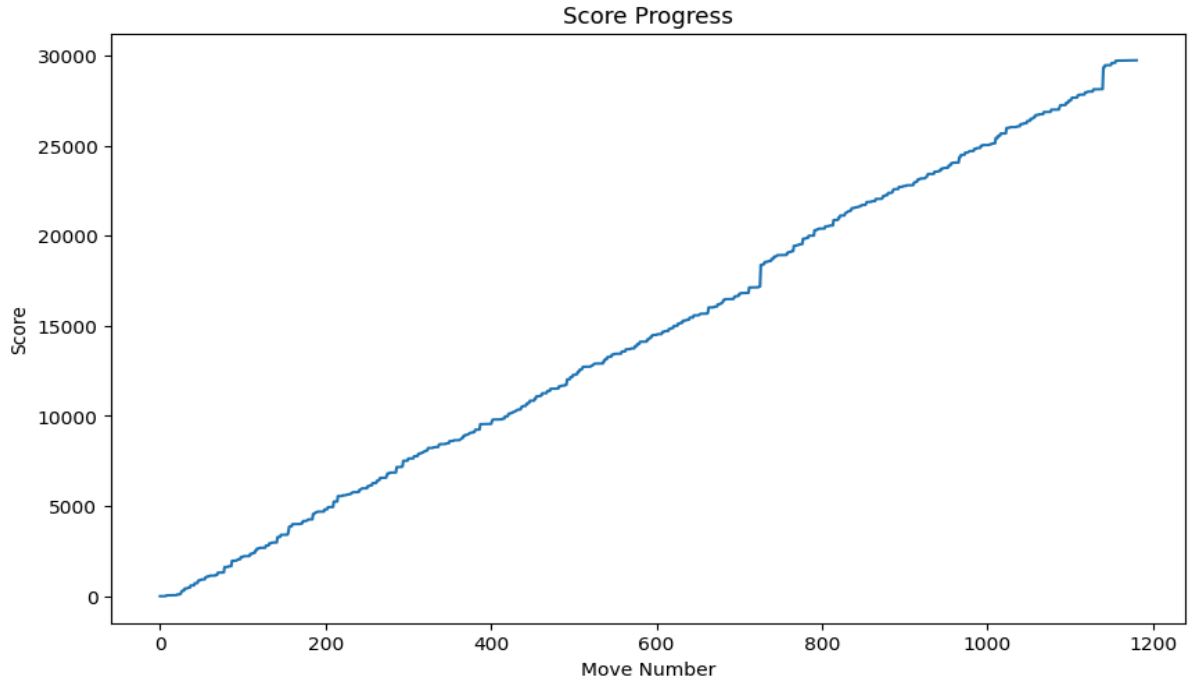
Second Chromosome (**ACHIEVING THE INTENDED GOAL**):



- Got the best 2 Chromosome from Model 1 and reran them but with calculated next move TRUE:
  - `[5.934778276089621,3.9432161697030246,-3.2543266900829693,-6.826623928690194] with Score:206012`
  - `[6.116385036656158,3.962787899764537,-3.1949896696401625,-6.890410003764369] with Score: 186992`

Results:

First Chromosome:



Second Chromosome:

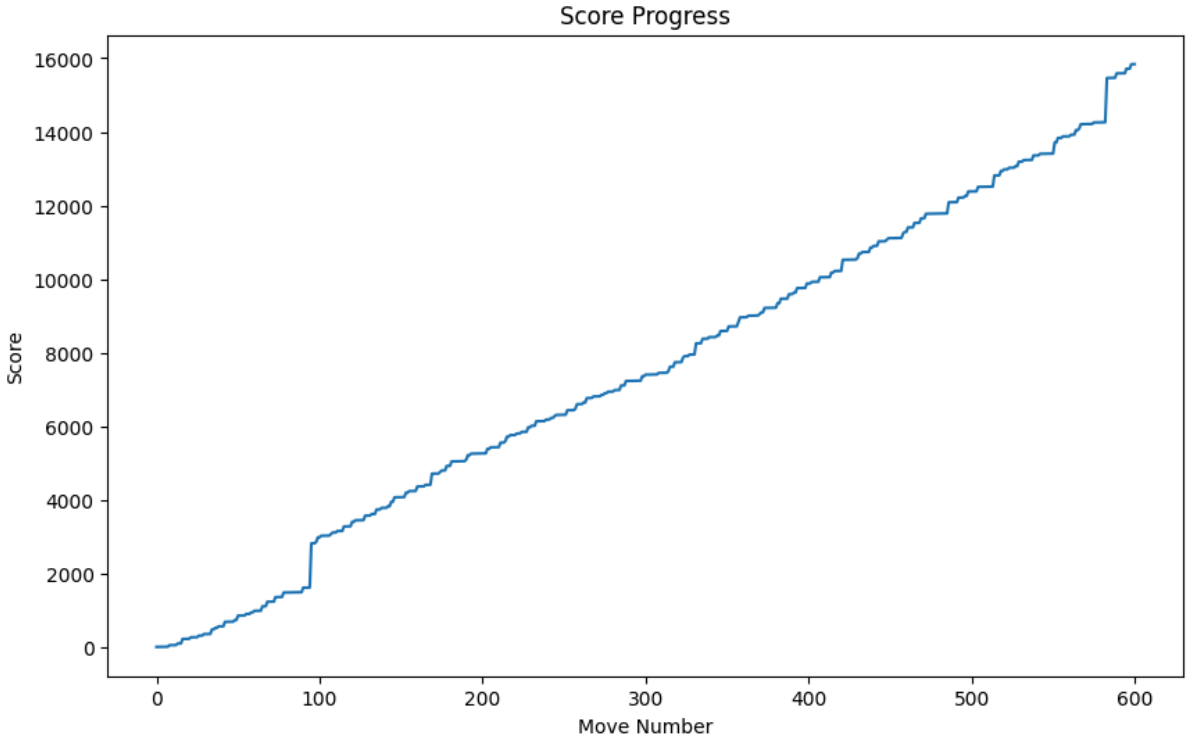
3. Requested Statistics

Chosen Seed: 42

Optimal Chromosome:

- **Model 4: Optimal Genes**
  `[5.387269303109375,3.8794766494803907,-3.6804923867413235,-6.709630184000435,8.339967392082738] with Score: [500039]`

- **Final test run with 600 iterations/moves with a Score of `15840`**



4. References

- https://github.com/camilanovaes/tetris-ai/tree/master?tab=readme-ov-file
- https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/