Playing Tetris using Genetic Algorithms

Computational Intelligence for Optimization

Master's degree Program in Data Science and Advanced Analytics

NOVA Information Management School (NOVA IMS)

2020/2021

Beatriz Freitas[1], Catarina Pinheiro[2], Henrique Renda[3]

[1]20190252– m20190252@novaims.unl.pt

[2]20200654 – m20200654@novaims.unl.pt

[3]20200610 – m20200610@novaims.unl.pt

Abstract: As the third most purchased video game ever, the popularity of Tetris has been the target of several studies, namely studies related to artificial intelligence and computational intelligence, as it proves to be a good benchmark for these areas. This project objective's is to demonstrate the behavior of the implementation of genetic algorithms in the Tetris video game to optimize each move, this report will help the reader to understand the application of different combinations of selections, crossovers and mutations techniques taught during Computational Intelligence for Optimization classes.

The results obtained showed the viability of having an agent playing the classic version of this game where the central challenge of this maximization problem is to create the best educated plays that improve the result and create a favorable setup for future plays.

Keywords: Genetic Algorithms, Tetris, Artificial Intelligence, Computational Intelligence

Introduction

This project explores concepts of self-learning optimization using genetic algorithms inspired by Charles Darwin’s theory of natural evolution. We implemented this type of algorithms in the Tetris video game in order to create an agent that can play extremely well, evolving the pontuation thought the generations.

Throughout this report, the concepts behind the code that generates the initial population are explained, as well as how the problem's fitness function is composed and which selection, crossover and mutation algorithms are applied.

Background

1. Tetris Game

Tetris is a video game released in 1984 by the software engineer Alexey Pajitnov. The game give the players the permission to strategically allocate and/or rotate falling blocks (tetrominoes) in order to complete lines. When the player set one or more full rows they disappear, and the player earn points. The objective of the game is to avoid the accumulation of blocks in the top of the screen for as long as possible. The longer the player stay alive and playing, the higher their score will be.

Many versions of Tetris exist but in this project we will be using a standard version of Tetris with 10 columns, 24 rows and there are 7 different shapes of tetrominoes each of them composed by 4 blocks.

2. Utility Functions

To “teach” the computer how to learn to play this game and achieve a good score, our GA will decide the best move for a given Tetris piece by trying out all the possible moves (rotation and position). It computes a score for each possible move together with the future piece to spawn, and selects the one with the best score as its next move.

This project is a maximization problem that tends to focus on evolving nature of genetic algorithm in order to maximize the score of any individual and their chance of survival.

The file **tetrisUtils.py** contains the utility functions used for calculating factors that will further in turn will calculate the best action needed to obtain the best score.

a) get_effective_height:

This function calculates the height between the tetromino lowest point and its corresponding position’s highest point on board. In figure 1, the stone lowest point is at position 2, and at it’s corresponding position the highest point lies at 6, so the method will return “highest point” – 1 as the result.

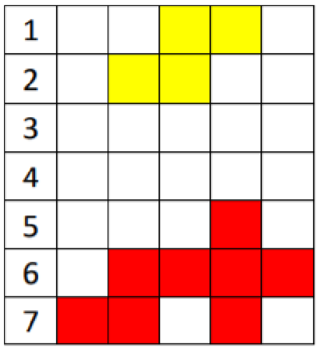


Figure 1 - Effective Height

b) get_board_with_tile:

This method creates a copy of the board and place the tile/stone/block on it, this is effective in understanding the fitness of each configuration of the tile when placed on board. An additional variable flattened is used for changing tile value to either 0 or 1. This method returns a copy of the board along with tile placed on it.

c) get_future_board_with_tile:

This method calls the *get_board_with_tile* function along with corresponding offsets. The offset in the horizontal direction is given by the program and the offset in the vertical direction is calculated using get_effective_height method. The returned value will be a copy of the board along with tile placed on it.

3. Main Heuristics

The score for each move is computed by assessing the grid the move would result in. This assessment is based on four heuristics: column height, complete lines, holes, and bumpiness, each of which methods are present in the **tetrisUtils.py** as well with the objective of either minimize or maximize.

a) get_col_heights:

We used this method to calculate the height of each column in the board. The height of any column refers to the maximum number of rows occupied by the block starting from bottom of the board and tells us how “high” a grid is. The returned output is a list of height of each column in the board. The following illustration is a given solution for the given grid of blocks.

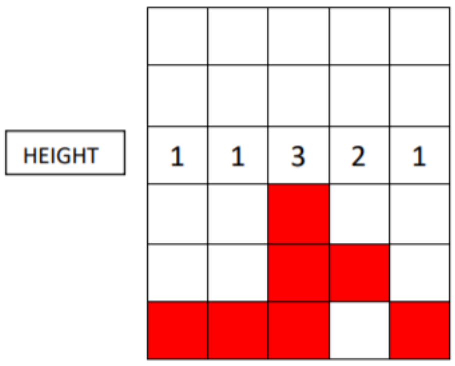


Figure 2 - Height of each column

b) get_hole_count:

A hole is defined as a part of the board which is empty but contains block/tiles on top of it. The output of this method is the number of holes in the playing board.

In the following figure is represented an example where the number of holes is equal to 3.

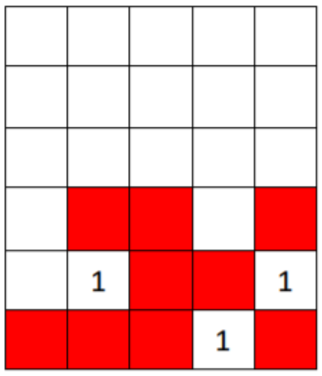


Figure 3 – Representation of holes in the board

c) get_bumpiness:

This method returns an integer corresponding to the bumpiness, that is defined as the absolute difference between the adjacent heights of the board.

Using the figure 2 as reference we calculate the bumpiness like:

$$\begin{aligned} Bumpiness &= |1-1| + |3-1| + |2-3| \\ &\quad + |1-2| \\ &\quad = 0 + 2 + 1 + 1 = 4 \end{aligned}$$

d) get_board_and_lines_cleared:

This function calculates the number of lines that are fully filled with tetrominoes. For example, if we use as input the following image the method will return the board and an integer with value 2.

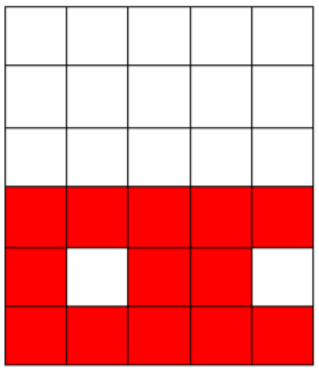


Figure 4 - Lines cleared in the board

Genetic Algorithm

I. Initial Population

To start implementing our GA in the Tetris game, we choose to have a random generation of initialize 40 population pools using the function *generateInitialPopulation* present in the file **geneticTetris.py** evolving for 50 generations with a mutation rate of 10%. Each individual in this set is a solution to the problem you want to solve.

II. Fitness Function

The fitness function or the cost function of a genetic algorithm is the operator that quantifies the quality of the solutions. This function that takes as input each candidate solution to the problem, usually known as a chromosome, and indicates how close it came to meeting the overall specification of the desired solution.

The calculation of the fitness value will occur many times in a GA and it should be fast so the performance of the computation will not be affected.

To keep track of each step of the game, we created an empty action_queue using the method *_init_* in the file **tetrisAgents.py** that initializes the weight for column heights (weight_height), the weight for holes (weight_holes), the weight for bumpiness (weight_bumpiness) and the weight for cleared lines (weight_line_clear). Our fitness function will be composed by linear combination of these four heuristics.

To understand how we manage the weights we need to better understand how each action of our game will be registered. To do that, we first implement the method *calculate_actions* that calculates next set of actions to be taken for the current tile/stone. The calculated actions then are added to the action_queue property.

To return the next action from the action_queue we end up creating the function *get_action*. Important to mention that if the action_queue property is empty then it calculates the next set of actions using *calculate_actions*. Important to mention that this last method will use the function *get_fitness* to choose the best movement.

III. Selection

There are multiples ways to select individuals, in the selection algorithms the choose of one individual is made according to the current population. In the present project we started doing two of the existing algorithms: Fitness Proportionate Selection (or “Roulette Wheel”) and Tournament Selection.

The first algorithm approaches a good solution faster but sacrifices diversity in the population. The individual selection is made by a “roulette wheel”, where we start from position 0 and the next position is calculated by an uniform function, that takes 0 and the total fitness value as arguments. The selected individual is the one that is in the position calculated.

IV. Genetic Operators

Genetic operators are of major importance since when the selection is terminated, the parent’s population is inhabited by individuals that have been chosen as apt for surviving and mating. Genetic Operators allow the best (and worse) characteristics of the parents to be inherited by the offsprings, making the evolution possible.

Similarly to the practical classes, we’ve created a folder called ‘Charles’ where we built both the ‘crossover.py’ and ‘mutation.py’ files. It contains all the functions we have developed throughout our assignment for our genetic operators.

V. Crossover

Crossover takes two parents (chromosomes), to create a new offspring by switching segments of the parent genes. These operators have a role in the balance between exploitation and exploration, which will allow the extraction of characteristics from both parents, and hopefully that the resulting offspring possess good characteristics from the parents (Gallard & Esquivel, 2001).

In our approach, we took two simple crossover methodologies and used a plot of Average Score versus Generations to help us decide which technic would be more appropriate.

Single point crossover

Inside ‘crossover.py’ there’s the function **single_point_co** which takes as arguments both parents (*p1* and *p2*). Single point crossover is the most approved crossover which is widely in use. A crossover site (*co_point*) is randomly selected along the length of the mated strings and bits which are very next to the cross-sites are interchanged. So we have, for example, *offspring1* which has a first part coming from *p1* untill the *co_point* and from then on it’s all part of p2.

Arithmetic crossover

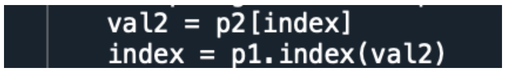
We also have inside ‘crossover.py’ the function **arithmetic_co** which also takes as arguments *p1*, *p2*. Arithmetic crossover is usually used in case of real-value encoding, and linearly combines the two parent chromosomes.

This linear combination is as per the following:

$$offspring1 = p1 * \alpha + (1 - \alpha)$$

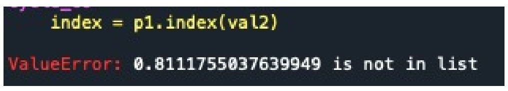
Cycle Crossover

Regarding Cycle Crossover, it wasn’t possible to implement. For the algorithm to operate accordingly we needed both representations of the parents to have the same elements, or else, when doing:



```
        val2 = p2[index]
        index = p1.index(val2)
```

It raised the following error:



```
    index = p1.index(val2)

ValueError: 0.8111755037639949 is not in list
```

Due to the fact that both parents have different elements inside their representations.

PMX Crossover

In PMX crossover we came across the same problem as Cycle Crossover. The parent’s representation need to have the same elements, or else we will have a *ValueError*.

Mutation:

In the mutation we implemented algorithms which altered in some way the selected candidate’s input. The mutation operator brings into the population new possible characteristics or gene values. The mutation rate decides the magnitude of changes to be made in an individual to produce the mutated individual which constitutes the individual of the next generation.

Bit Flip Mutation:

Bit Flip Mutation uses a certain individual’s index and flips the gene to a 0 if it’s a 1 and to 1 if it is a 0. Evidently, we can’t use this type of genetic operator since our individual’s representation it’s not binary.

Swap Mutation:

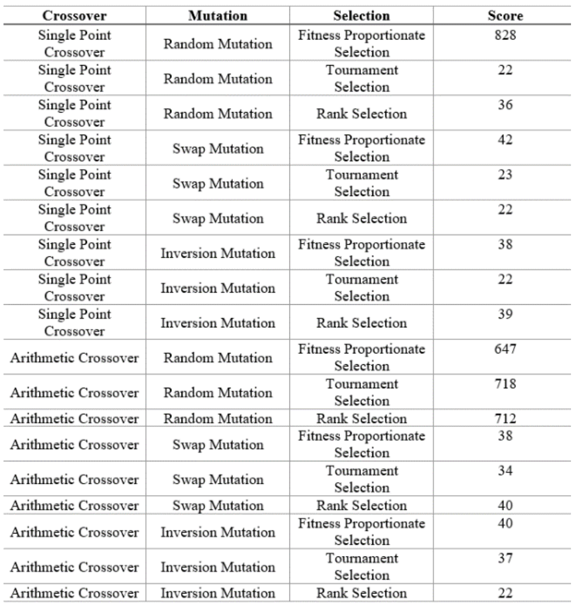
In swap mutation, we select two positions on the chromosome randomly and change the values between them. ‘mutation.py’ showcases the function ***swap_mutation*** which gets two mutation points (mut_points) and returns the interchange of the values.

Inversion Mutation:

In Inversion Mutation, we select a subset of genes and invert the entire string in the subset. If we take look into ‘mutation.py’ we can find the ***inversion_mutation*** which takes as argument the individual. First, we sample two indexes from the individual through *mut_points* and then we sort it and invert for the mutation.

Discussion

Since the beginning of the semester we’ve learned several algorithms and, to demonstrate our knowledge, we thought that would be an interesting idea to understand which Genetic Operators worked best between them. For that we have created a selection of 18 combinations present in the following table:



Crossover	Mutation	Selection	Score
Single Point Crossover	Random Mutation	Fitness Proportionate Selection	828
Single Point Crossover	Random Mutation	Tournament Selection	22
Single Point Crossover	Random Mutation	Rank Selection	36
Single Point Crossover	Swap Mutation	Fitness Proportionate Selection	42
Single Point Crossover	Swap Mutation	Tournament Selection	23
Single Point Crossover	Swap Mutation	Rank Selection	22
Single Point Crossover	Inversion Mutation	Fitness Proportionate Selection	38
Single Point Crossover	Inversion Mutation	Tournament Selection	22
Single Point Crossover	Inversion Mutation	Rank Selection	39
Arithmetic Crossover	Random Mutation	Fitness Proportionate Selection	647
Arithmetic Crossover	Random Mutation	Tournament Selection	718
Arithmetic Crossover	Random Mutation	Rank Selection	712
Arithmetic Crossover	Swap Mutation	Fitness Proportionate Selection	38
Arithmetic Crossover	Swap Mutation	Tournament Selection	34
Arithmetic Crossover	Swap Mutation	Rank Selection	40
Arithmetic Crossover	Inversion Mutation	Fitness Proportionate Selection	40
Arithmetic Crossover	Inversion Mutation	Tournament Selection	37
Arithmetic Crossover	Inversion Mutation	Rank Selection	22

Figure 5 - Tested Combinations

The results showcase that combinations that use Arithmetic Crossover and Random Mutation alongside with no specific selections methods present a good result. Furthermore, the winning group was the one that features Single Point Mutation, Random Mutation and Fitness Proportionate Selection. As we can see (figure 6 in appendix), the latter not only reaches a global optimum sooner, but also, reaches a higher peak. On the other hand, Random Selection seems to be a common factor, hence we have decided to do some parameter tuning on the mutation rate of the combo Single Point Mutation, Random Mutation and Fitness Proportionate Selection.

As shown in figure 7 by varying the mutation rate between 0.1 and 0.3 we've reached the best score yet of 1008. Throughout the generations we can observe that the 0.1 mutation seems to be the highest contrasting with 0.3 mutation. However, 0.2 gives us the global optimum solution.

Conclusions

In this report, the ideas taught during the semester were always present. Despite some difficulties, we managed to complete a work that clearly exposes a good implementation of a genetic algorithm as well as an attempt to improve its functioning. We can thus admit that it was from this optimization of the Tetris game that we demonstrated the potential of this type of algorithms to solve logical problems and that require human effort to guarantee an optimal solution.

However, we are aware that our solution is not the best and we believe that it can still be improved by implementing different selection, mutation, and crossover techniques, as well as testing different number of population size and generations to create a more solid conclusion regarding the algorithm's results.

References

Gallard, R. H. & Esquivel, S. C., 2001. Enhancing evolutionary algorithms through recombination and parallelism. Journal of Computer Science & Technology, Volume 1.

Lee, Yiyuan.,2013. Tetris AI – The (Near) Perfect Bot. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

Yadlapalli, Priyanka & Kora, Padmavathi - Crossover Operators in Genetic Algorithms: A Review - International Journal of Computer Applications.

Vanneschi, L - Computational Intelligence for Optimization.

Appendix

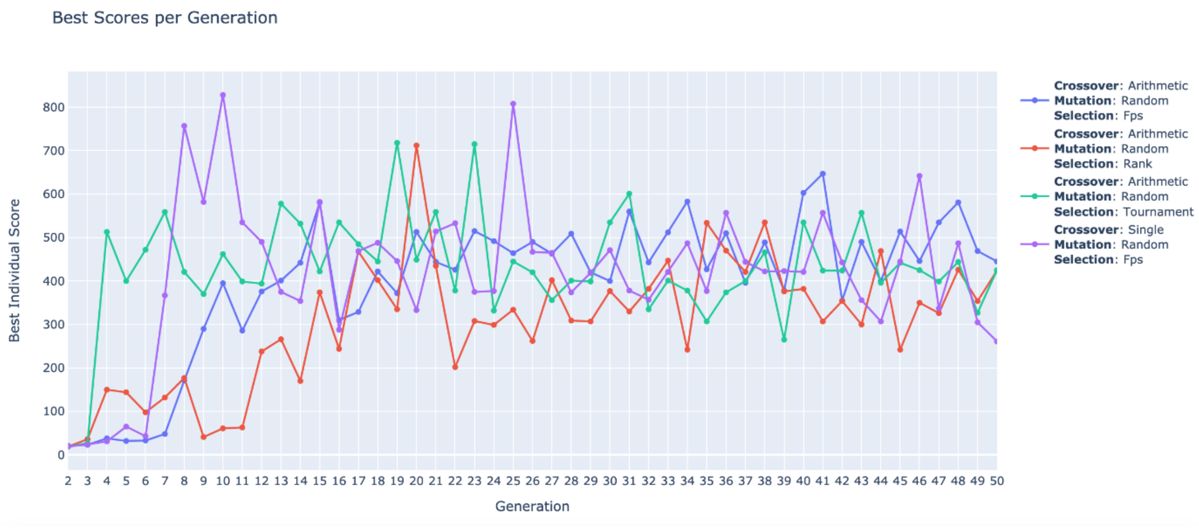


Figure 6 – Comparison between the best combinations

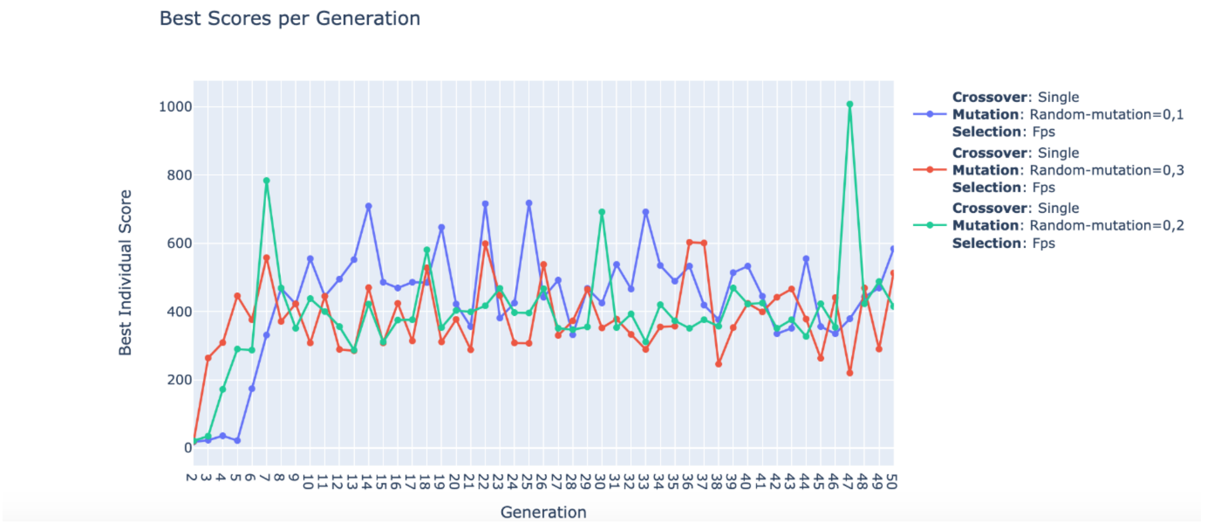


Figure 7 - Comparison between different mutation rates