Computational Biology 2022 - Ex2

Sara Spagnoletto 345990808 - Computer Science Major
Lilach Herzog 203292792 - Biology Major
https://github.com/saraspagno/CB2/tree/master

1. Introduction

Futoshiki is a board-based puzzle game. The purpose of the game is to discover the digits hidden inside the board's cells. The rules dictate rows and columns cells must be filled with a digit between 1 and the board's size. On each row and column, each digit appears exactly once. At the beginning of the game, some digits might be revealed. The board might also contain some inequalities between the board cells; these inequalities must be respected and can be used as clues to discover the remaining hidden digits.

In this project, we attempt to build a **genetic algorithm** to solve a Futoshiki's board. The algorithm is executed from scratch, including the settings for function implementations and hyper-parameters.
In addition, the aim of this research is to compare three different kinds of genetics algorithms and their performance on the puzzle:
• Standard genetic algorithm with no board optimization.
• **Darwinian** genetic algorithm in which each board is optimized and its fitness is determined only after the optimization, but the next generation is created according to the before optimization.
• **Lamarck** genetic algorithm in which each solution is optimized, its fitness is determined only after the optimization, and the next generation is created according to the after optimization.

2. Building a genetic algorithm for the Futoshiki puzzle

The pencil process

The first step in finding a solution for the puzzle is the "pencil process." Before starting the genetic algorithm, we wanted to certify that the board was filled to its extent, avoiding useless iterations.
We followed fixed rules such as cells greater than other squares cannot be 1, squares greater than the board size minus 1 must be equal to the board size, etc. Those allowed us to have an initial screening of the board and, in most cases, discover multiple filled cells at the beginning of the algorithm.

Algorithm implementations decisions

Solutions' representations: the solution is represented as an np array of arrays (rows) and printed throughout the execution to display a board. The constraint of the "constant" numbers (pre-filled) is represented as a list of lists, where each list element has indexes and values. The constraint of the "greater" conditions is also represented as a list of lists, where each list element has indexes of great and small cells.

Generating random initial grid: the initial grids are generated randomly, but in a way that each row is strictly legal. Constant numbers are filled correctly, and each row is made of unique numbers from 1 to N. The legality is maintained throughout the algorithm execution. Therefore, the fitness score is calculated only on columns and greater constraints.

Fitness function: the fitness function is inverse; the higher the score the worse the solution. For each

"greater" number not respected we add 2 to the score. For each not unique value in a column, we add its frequency - 1 to the score.

Cross-over: a random number d is chosen between 0 and the board's size. In the new board, the first d rows are taken from the first grid, and the remaining from the other.

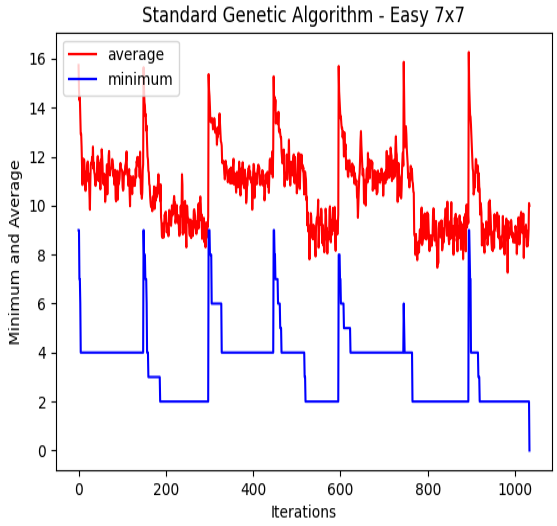
Mutations: a random number t is chosen between 1 and board's size + 1. In the board, t rows will be mutated and replaced randomly. This way keep the legality of each row.

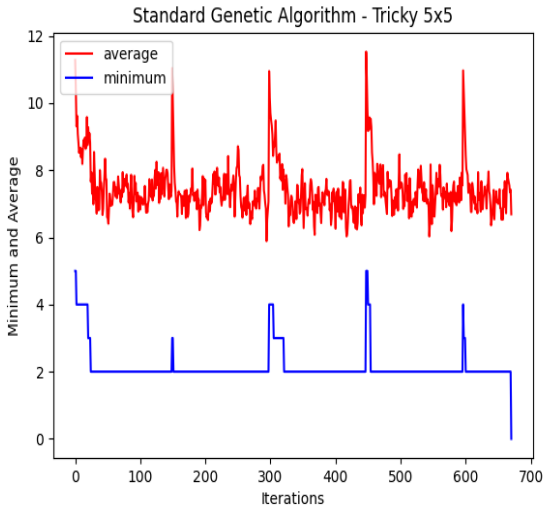
Selection: for each board, the probability of being selected is inversely normalized on its fitness score.

For each generation, a new population is created. The best board is replicated for 10% of the population. The remaining 90% is obtained by selection and cross-over. In addition, 20% of the population at random will undergo a mutation.

The biggest problem we faced developing the algorithm was avoiding early convergence. From the start of the running, the fitness seemed to decrease steadily to one or two errors only. Finding a way of breaking from that local minimum was a great challenge. We were able to partially solve this problem by frequently restarting the algorithm. Ultimately, we decided to implement a maximum of 30 restarts, each of which with 150 iterations, for a total of 4500 maximum iterations for attempting to solve the problem. These parameters were decided following a great amount of testing.

Here are two graphs showing the minimum and average fitness in each iteration while solving a board, one for an easy 7x7 board, and one for a trick 5x5 board.





The spikes in the graphs are a consequence of the algorithm restarts. It can be noted that the Easy 7x7 took more time than the Tricky 5x5.

Comparison between Standard, Darwinian, and Lamarck Genetic Algorithm

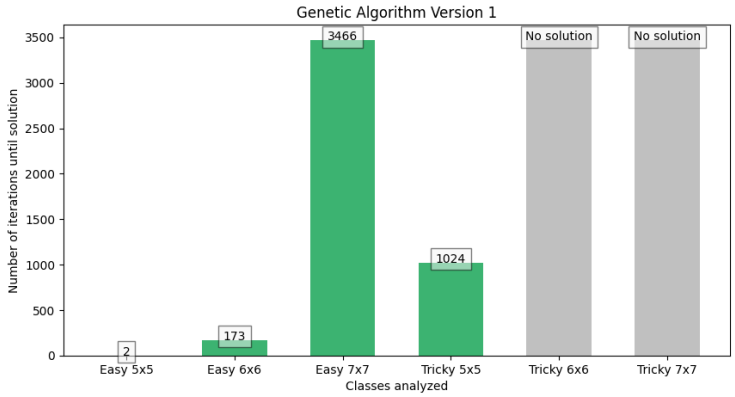
Optimization: Implementing the optimization was a great challenge. As mentioned above, the rows of each board were already strictly legal under every constraint. The only errors that could have possibly been optimized were: having not-unique elements in columns or greatness cells in columns. The question was how

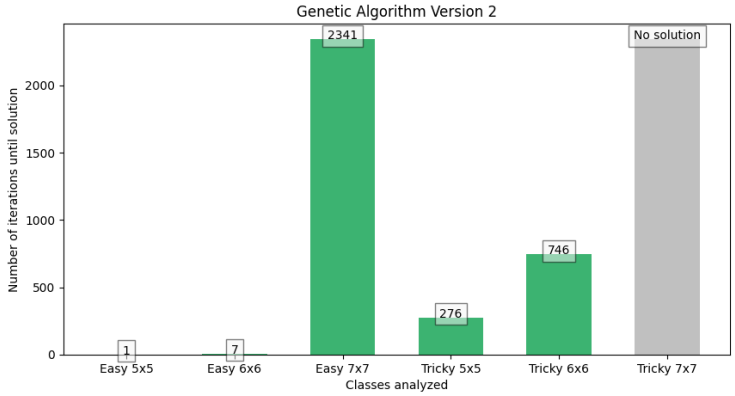
to optimize these errors keeping the board's rows legal at every iteration. Finally, we decided to optimize not-unique columns by replacing the first-row that caused multiple occurrences. We tackled the problem and hopefully solved it while still keeping the rows legal.

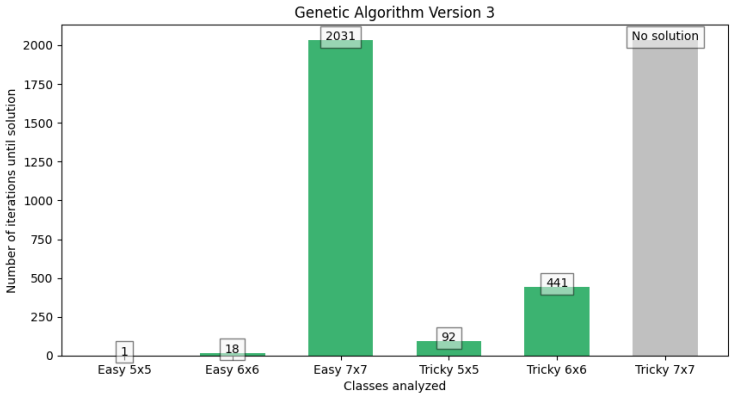
But this doesn't guarantee to improve the board's score; especially if the solution is a local minimum, this will most likely worsen it. Because of this, in the Lamarck version, we decided to keep the optimized board only if the score improved. The process acted as a sort of targeted mutation, which contributed vastly to improving the result of the cross-over.

3. Conclusion and results

Following there are three graphs showing the average number of iterations until solution for multiple difficulties and board's sizes. The average is taken among 10 runs of the algorithm. Each graph is using a different type of genetic algorithm, respectively: Standard, Darwinian and Lamarck.







From the results we conclude Darwinian and Lamarck have a better performance in terms of finding a solution for the Tricky 6x6. In addition, on average, the Darwinian took less iterations than standard, and Lamarck less iterations than Darwinian.