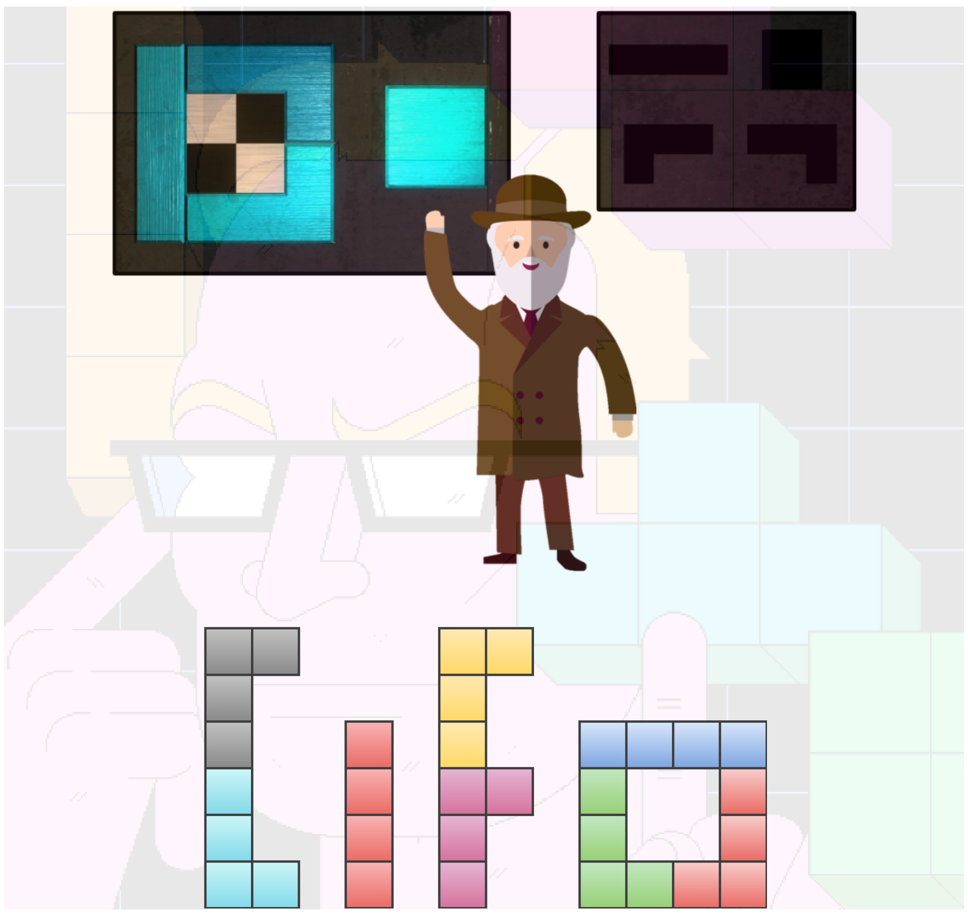
Developing a Tetris Block Puzzle Solver with Genetic Algorithms

Computational Intelligence for Optimization

20/21 Project

NOVA Information Management School (NOVA IMS)



Darwin's Chosen:

Beatriz Pereira (20200674@novaims.unl.pt)
Frederico Santos (20200604@novaims.unl.pt)
Ivan Kisialiou (20200998@novaims.unl.pt)
Tiago Ramos (20200613@novaims.unl.pt)

Introduction

We implement GAs as an approach to solving Tetris Block Puzzles, wherein given a grid of a predefined size and a set of rotatable tetrominoes, the solver is asked to find a valid configuration of block placements that fills the grid as best as possible.

We consider this type of combinatorial problem to be both approachable and potentially challenging for people of all ages, which motivated our choice. Furthermore, its rules should be immediately evident from a single image:

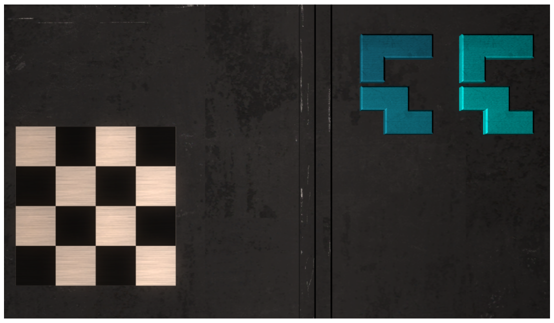


Figure 1: An example of a tetris block puzzle, from the free game "Sigils of Elohim"

We detail our approach, results and discussion in the following sections.

Methodology

Several steps and considerations specific to this challenge must be made before implementing GAs:

Formalization

An appropriate definition of the solution space and fitness function are crucial to the development of this project. Our goal is to ensure that the search space is composed of only valid solutions, and that fitness is a strong indicator of how close an individual is to a perfect solution (characterized by maximum possible fitness).

Given the nature of the challenge, the necessity for an algorithm that could fit the given pieces into the grid became evident. We call this algorithm the "(Tetris) Fitter". This shapes the way we can formalize both individuals and fitness. A thorough explanation of the Fitter can be found in Appendix A.

Individual

An individual is represented by all the pieces from the set of pieces, ordered in a sequence with each piece having an orientation (rotated in a certain way). The pieces are selected without replacement. Each piece in the sequence is represented as a string of `<letter><number>`, where the `<letter>` represents the shape of a piece, and the `<number>` its rotation.

For the letter, there are 7 types of pieces: `[I, L, J, T, S, Z, O]`.

For the number, there are 4 possible orientations (90º increments): `[0, 1, 2, 3]`.

Fitness Function

Logically, if all pieces are used without violating any rules, a maximum fitness would have been achieved. This is analogous to observing that a perfect solution is represented by a grid with no empty spaces. We first developed the fitness function in respect to how many pieces fit into the grid; however, only using this information was limiting, as an exponentially large amount of solutions with near perfect fitness existed.

Generally, our intuition is that the closer we are to the correct solution the closer the remaining empty spaces are. We emulate this rationale by penalizing our fitness function with the sum of the Manhattan (L_1) distances between empty squares in the grid. Our fitness function is thus characterized by the formula:

$$F = F_{occupation} \times 100 - F_{compactness}$$

where $F_{occupation} = |pieces\ in\ grid| \times 4$ and $F_{compactness} = \sum_{i \neq j} h_1(i, j)$

In summary, this construction implies that no individual will violate any constraint of the problem, that every possible configuration and solution for a given grid is representable by an individual, and that the corresponding fitness will be an accurate representation of how perfectly the grid is filled.

Moreover, because sequence of pieces is important and every individual is represented by a string of prefixed length, it is suitable for the implementation of genetic algorithms, necessitating no further transformations.

GA Implementation

Below we present a diagram summarizing the choices of implementation made to run experimental comparisons.

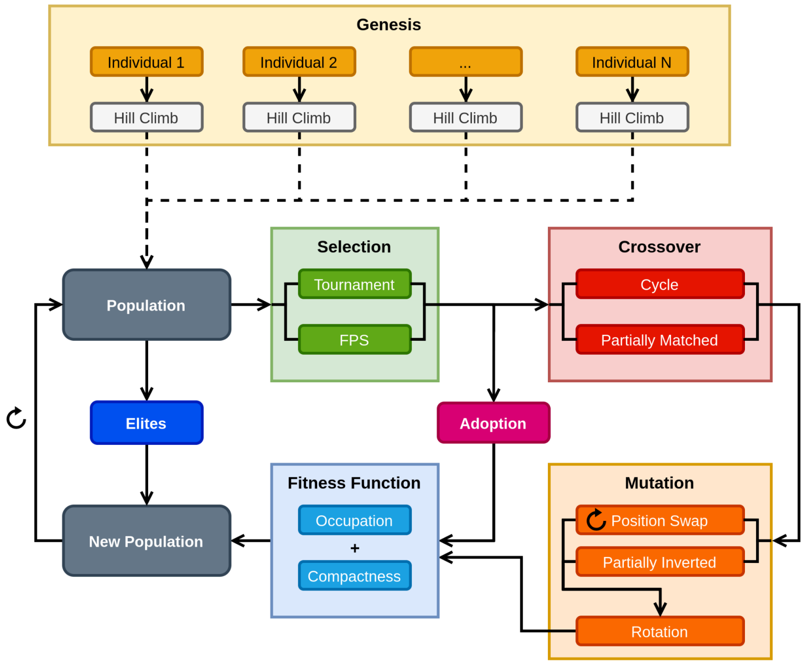


Figure 2: GAO Architecture

Genesis

For the generation of a problem and the subsequent search space, we took two approaches. The first was to employ an automated and semi-deterministic algorithm (A thorough explanation on the approach to this generator can be found in Appendix C). Due to the constraints it introduced, we opted for the second approach when moving forward with our testing, which is to hardcode both grid shape and piece set to be used by the algorithm.

We then initialize each individual's representation in random order, with random orientation for each piece. We then optimize each individual's fitness with Hill Climbing. We apply the same method when adopting individuals.

Selection

Two selection methods were implemented: Tournament Selection and Fitness Proportion Selection. Our intuition points towards having similar results from either operator, considering the major differentiating factor of better individuals is how compact the pieces are fit.

Regarding elitism: despite our definition of fitness function mitigating the relative flatness of its landscape, the fitness of an individual can still change radically upon even the slightest of mutations. To prevent such mutations, we chose to implement elitism as default. We predict that altering the number of elites will have varied results, depending on the number of global optima.

Adoption

In order to avoid premature convergence, we had an idea: give each couple a slight probability to adopt a completely new individual, without impacting the probabilities of crossover. We intuitively believe this method to allow for exploration of individuals not at all similar to the current population, as long as the probability of adoption is a very low value.

Crossover

A few challenges arose when applying genetic operators to these individuals: while the set of pieces (i.e. the letter component of the string) are constrained to a specific set, rotations (the number component) are not, posing an added layer of complexity to the implementation.

One choice is to separate both components, akin to a double chromosome. Then crossover or mutations could be applied independently, and results zipped together at the end. However, this would introduce a level of undesirable randomness. An illustrative example can be found in Appendix B.

As such, we decided to transform each representation into an index of pieces of one parent, then apply the same transformation given the link $letter: index$ of the first parent and apply partially matched crossover and cycle crossover to this transformed sequence. With this implementation, the offspring will inherit the piece index and its corresponding rotation from one of the parents, while still retaining a valid set of pieces.

Example: `Parent 1 = [Z3, I0, S2] -> [Z, I, S] -> [0, 1, 2]`

`Parent 2 = [I2, S3, Z0] -> [I, S, Z] -> [1, 2, 0]`

Mutation

We cannot change the piece of the individual itself except for changing the positions of the strings or changing the second element of each string, i.e., the rotation of each piece. Therefore, we implemented a two-step mutation. For the first step, we used either Partially Inverted Mutation or Position Swap Mutation. For the second step, we used Rotation Mutation.

Rotation Mutation

Given an individual, with a certain probability for each string of its representation, we randomly select a different orientation for the piece. Programmatically, we randomly select a value from 0 to 3 for the second element of the string as long as the value is different from the original string.

Results and Discussion

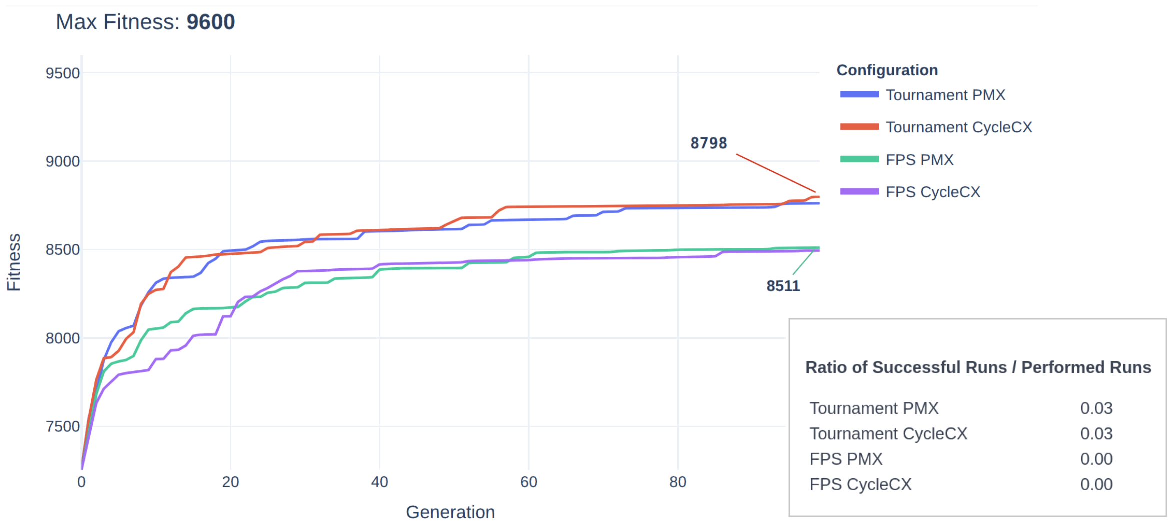
Given the purpose of the solver, we value more highly a configuration that finds a perfect solution. In the case of no global optima (which can happen with initialized randomized sets of pieces), we would prefer a configuration that consistently fits the grid as neatly as possible. As such, we have applied both Ratio of Successful Runs and "Average Best Fitness" (ABF) to measure different configurations of the algorithm.

We tested several configurations, which can be found in Appendix D, presenting the most relevant ones. A further exploration of this testing can be found in the appendix. In our early testing, we found overwhelming success of the solver regarding smaller sized problems (<=7 pieces). Any configuration would be successful with an average of 50 generations. Therefore, we expanded our testing to medium (7 to 12 pieces) and even larger sized puzzles. At >100 generations per run, the impact of a given configuration became much clearer. This is coherent with the fact that the search space becomes exponentially larger with the size of a puzzle, which favors operators able to generate individuals with strong building blocks from the parents. However, it is notable how our construction already allows for extremely fast convergence on smaller puzzles, outperforming humans, and random search, at the same task.

We decided against initializing our individuals with hill climbing when running for multiple iterations. It took an individual a very long time to find the local optima using hill climbing due to the time complexity of searching for new neighbours in large grids. It was also empirically observed that initializing with or without hill climbing did not make an impact in the algorithm convergence time.

For the final comparison between operators, we ran the algorithm on 2 different puzzles, 6x8 and 8x12. As is clear from the graph below, the configurations, where *tournament* was used as the selection operator, clearly dominated over the *fitness proportionate selection* ones with a wide margin. The likelihood of higher fitness individuals being selected constantly in a configuration with FPS makes the algorithm converge without diversification.

Ceteris paribus, using *partially mutated crossover* showed no significant difference over *cycle crossover*.



Ratio of Successful Runs / Performed Runs

Tournament PMX	0.03
Tournament CycleCX	0.03
FPS PMX	0.00
FPS CycleCX	0.00

Figure 3: Performance Analysis

Indeed, the ratio of successful runs over performed runs was low. No configuration achieved perfect fitness. Only 3% of the population, of configurations with tournament selection, were able to achieve the record fitness.

Many more comparisons can be found in Appendix E.

Conclusion & Future Work

We have successfully implemented many different operators with a genetic algorithm to an interesting problem. It resembles the Travelling Salesperson Problem in one way, since we must “visit” every piece and, in another way, it resembles the Knapsack problem, because we must rearrange the combination in such a way to be able to “visit” every piece!

Several possibilities exist in regards to future work and direction that we considered and would have pursued given a broader project scope and time constraints:

- Increased Complexity: the problem is formalized in such a way that several components can be tuned in order to generalize or expand its scope. For example:
  - Grid shape. While grid size is an obvious parameter choice, it is trivial to test different grid combinations, with different shapes and formats, by initializing a grid with some squares already filled in.
  - Piece set: The tetrominoes used for the purpose of the project are but a subset of a larger group of geometric shapes called polyominoes. The problem can be expanded to these n-squared shapes, or allow for a combination of differently sized pieces, or even non-connected polyominoes.
  - Dimensionality: Albeit difficult, it is possible to generalize the implementation to a 3 or even n-dimensional space. This requires careful reconfiguration of the pieces and the definition of rules for rotations, which we expect would necessitate algorithms for the generation of n-dimensional polyominoes from 2d counterparts.
- Initialization Algorithms: Instead of initializing from premade solved blocks and grouping them together, it would be very interesting to develop a GA to generate the puzzle, certainly with a fitness function to incorporate piece diversity (so that we don’t end with all square tetrominoes!)
- Meta-GA: Use of a genetic algorithm to fine-tune the parameters for the genetic algorithm itself would decrease manual search of optimal parameters.

Of course, we were not negligent towards the practical applications of the solver - in Appendix F, we present some solutions the solver found for the puzzles from which this project took inspiration.

References

- *Steam Community : Guide : Sigils of Elohim - Puzzle Solutions Part 1-3 (On-Going)*. https://steamcommunity.com/sharedfiles/filedetails/?id=327117676.
- Vanneschi, Leonardo. *Computational Intelligence for Optimization*.

Appendix

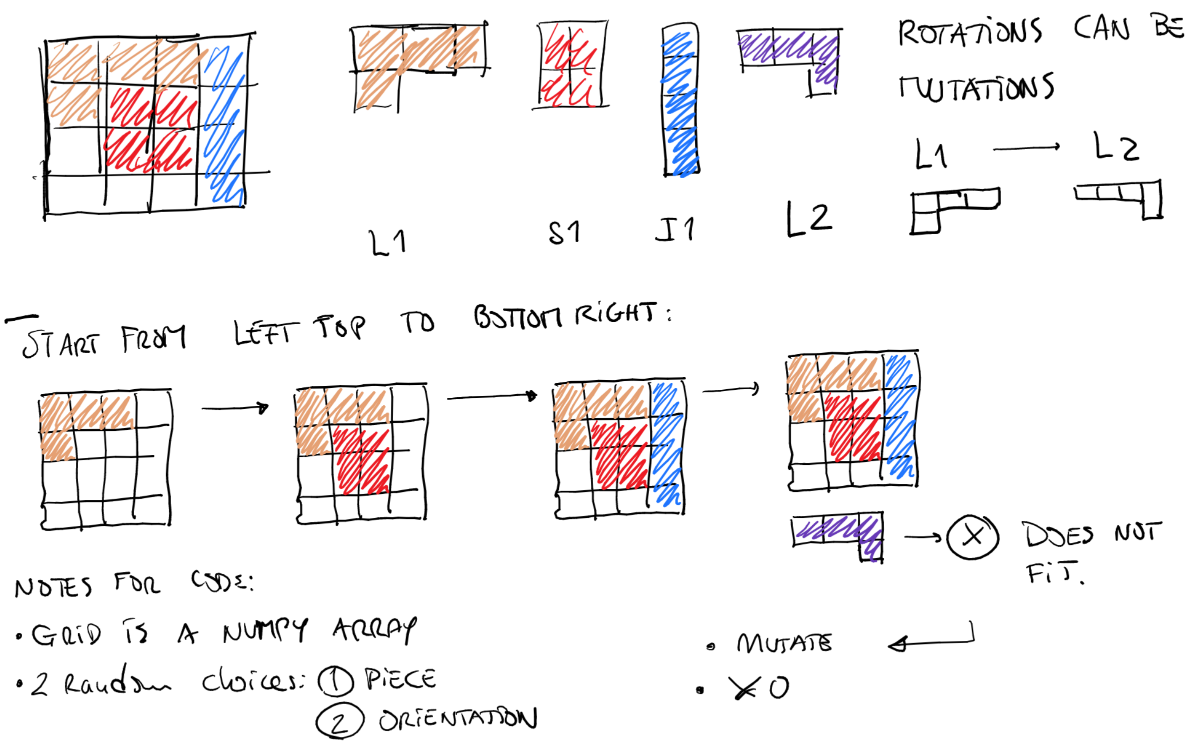
Appendix A - The Tetris Fitter

A grid is defined as an array where 0s denote an empty space.

The Fitter algorithm takes each piece (or string) of the individual sequentially, and places it according to its defined configuration in the first available (top-left to bottom-right) matching space on the grid. When placing the piece in a certain coordinate, if it overlaps with any other piece or goes out of bounds, it is considered a non-matching coordinate and the algorithm moves on to the next possible coordinate. If it does not find any matching space, it does not place the piece and attempts to place the next piece in the sequence.

Example: Given $O1$ (a square block piece rotated by 90 degrees clockwise), its 2d array representation would be given by any four coordinates (i, j), $(i + 1, j)$, $(i, j + 1)$, $(i + 1, j + 1)$. By looking at the available spaces in the grid, the algorithm searches for the first (i, j) for which all 4 coordinates of the array are empty. When it finds such an (i, j), those spaces are filled and the algorithm attempts to place the next piece in the sequence, in the same manner.

Below it is shown a draft of this algorithm.

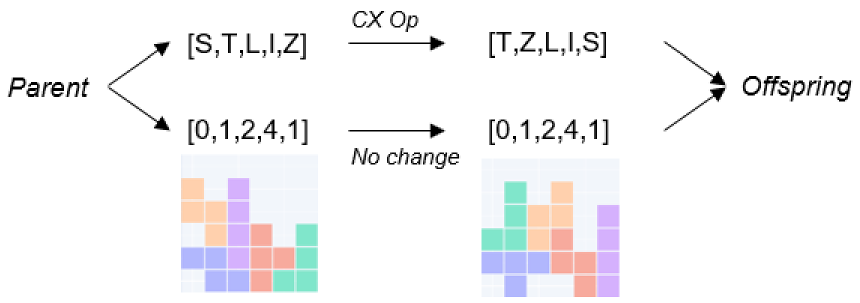


L1
S1
J1
L2
ROTATIONS CAN BE MUTATIONS
L1
L2
START FROM LEFT TOP TO BOTTOM RIGHT:
DOES NOT FIT.
• MUTATE
• XO

NOTES FOR CODE:
- GRID IS A NUMPY ARRAY
- 2 Random choices: ① PIECE
  ② ORIENTATION

Appendix B - Example of detached crossover

Consider a case where some crossover was applied to only the sequence of pieces.



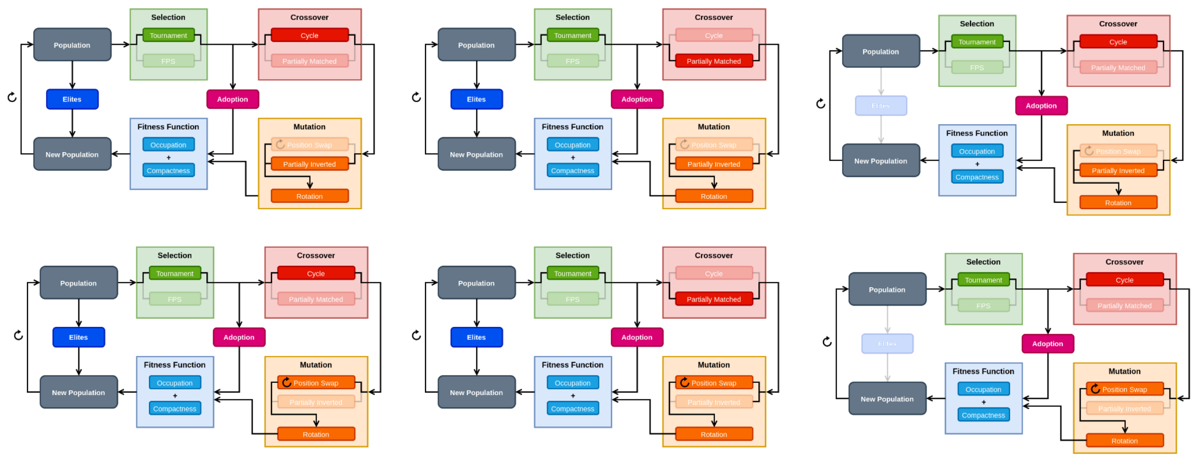
Although no operator was applied to the sequence of digits, inherited pieces have new rotations from their parent (except I and L piece), which is analogous to the pieces' rotation having mutated. We would therefore lose precious information from the parents, as well as losing control over mutation.

Appendix C - Search Space Generation

Given the constraints of the problem, we designed a puzzle generator that creates a set of pieces guaranteeing that an individual can achieve at least one perfect fitness (i.e., have no empty spaces in the grid). Because the problem of such a task is defined as NP-hard, there is no easy path to a solution. However, by breaking the problem into smaller problems, we could create a set of possible solutions for the smaller problem then scale the solution up.

We created multiple smaller blocks of $(4, 4)$ size, each an unique combination of tetrominoes that solved the block. To create our puzzle, we sampled, with replacement, from this set of $(4, 4)$ blocks $n \times m$ times. By using this method, we were able to generate a puzzle that was always completely solvable.

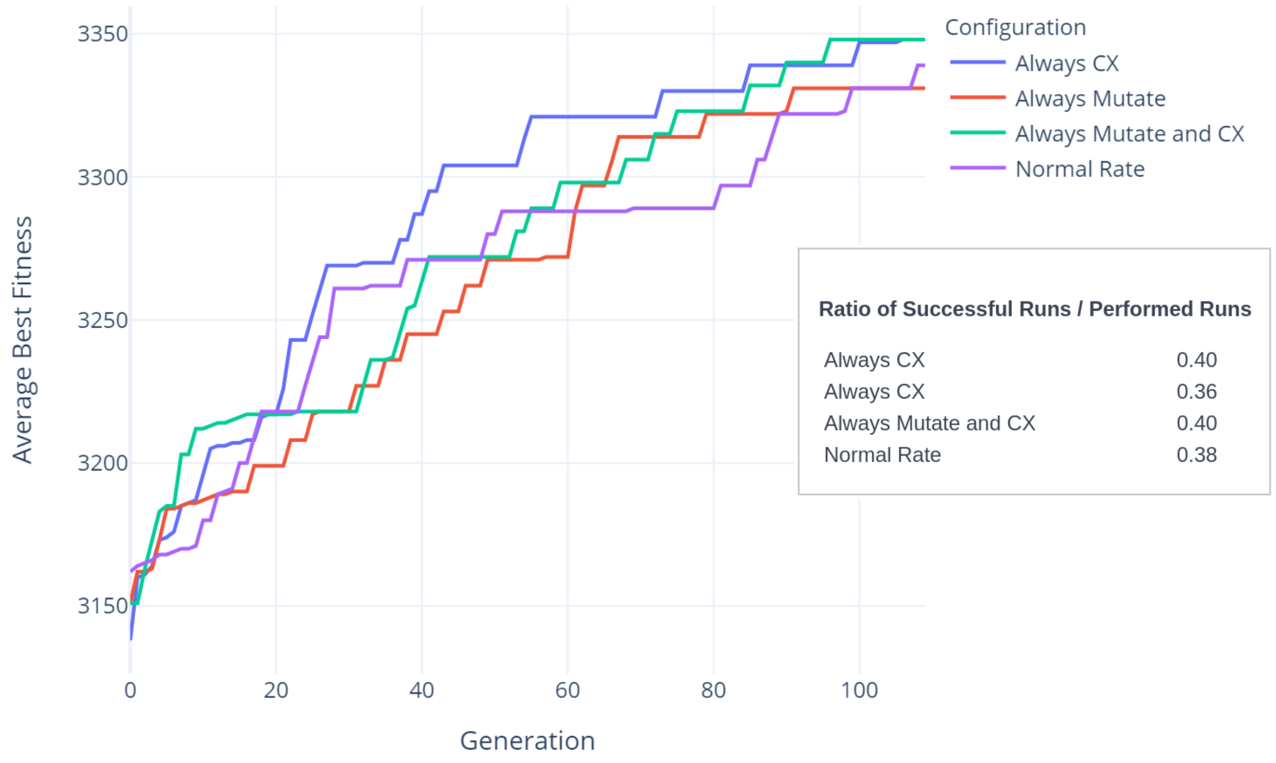
Appendix D - Tested GAO Configurations



Appendix E - Exploration of Experimental Comparison Results

This appendix presents results and a brief discussion that illustrates how our choices of parameters were informed.

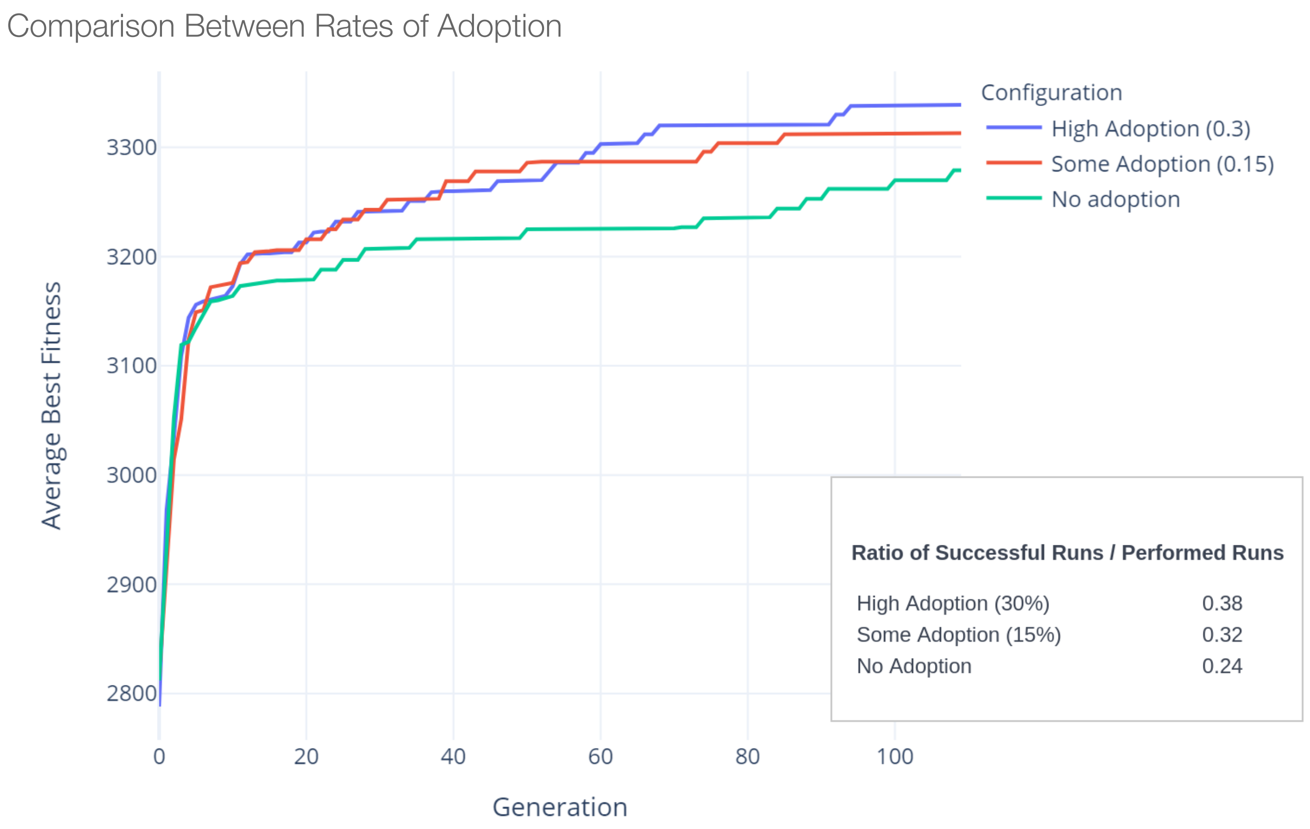
Comparison between different rates of operators



Ratio of Successful Runs / Performed Runs	
Always CX	0.40
Always CX	0.36
Always Mutate and CX	0.40
Normal Rate	0.38

Table: Comparison Between Op Rate	Population Size	# Elites	Runs	Selection type	Crossover type	Prob(CX)	Mutation Type	Prob(mut)	Prob(rotation mutation)	Prob(adoption)
Always CX	20	2	50	Tournament	Cycle	1	Swap	0.4	0.25	0.1
Always Mutate	20	2	50	Tournament	Cycle	0.75	Swap	1	0.25	0.1
Always Mutate and CX	20	2	50	Tournament	Cycle	1	Swap	1	0.25	0.1
Normal Rate	20	2	50	Tournament	Cycle	0.75	Swap	0.4	0.25	0.1

Discussion: The above is an example of a test due to which we eventually decided to advance forward with a permanent rate of crossover, but a lower rate of mutation. The SR ratio pointed us to high rates for both; given the ABF plot, we prefer the “Always CX” option, as it demonstrated the ability to generate good fitness with fewer generations, with the same Success Ratio.



Ratio of Successful Runs / Performed Runs	
High Adoption (30%)	0.38
Some Adoption (15%)	0.32
No Adoption	0.24

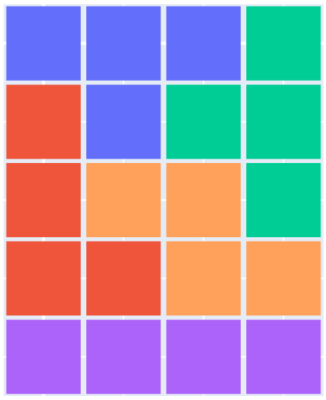
Table: Rates of Adoption	Population Size	# Elites	Runs	Selection type	Crossover type	Prob(CX)	Mutation Type	Prob(mut)	Prob(rotation mutation)	Prob(adoption)
High Adoption	20	2	50	Tournament	Cycle	1	Swap	0.4	0.25	0.3
Some adoption	20	2	50	Tournament	Cycle	1	Swap	0.4	0.25	0.15
No adoption	20	2	50	Tournament	Cycle	1	Swap	0.4	0.25	0

Discussion: The only difference in this test is the rate of adoption. It is relevant to highlight the positive impact of adoption on our solver, allowing fitness to scale better with more generations, a crucial factor for larger puzzles that need more generations to explore more of the search space. Subsequent tests showed even better (but only slightly) results for an adoption rate of 0.5, which we decided to keep moving forward.

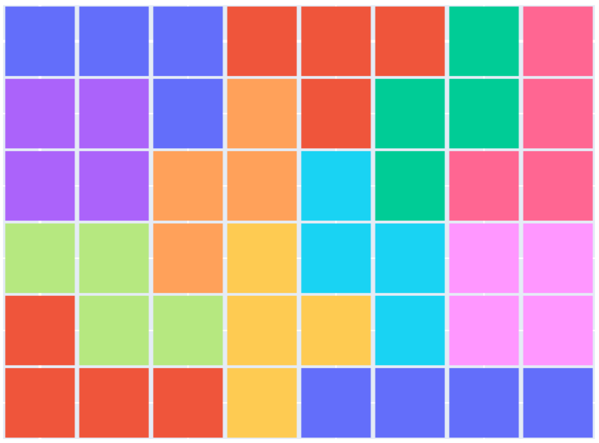
Appendix F - The **T**etris **BL**ock p**U**zzle **S**olver (T-BLUS) in Action

Below we present visualizations of the solver's proposed solutions to handcrafted puzzles, mostly from the game Sigils of Elohim (the hardest difficulty, of course). Some of them can also be found in the overwhelmingly applauded Talos Principle game, which we cannot recommend enough.

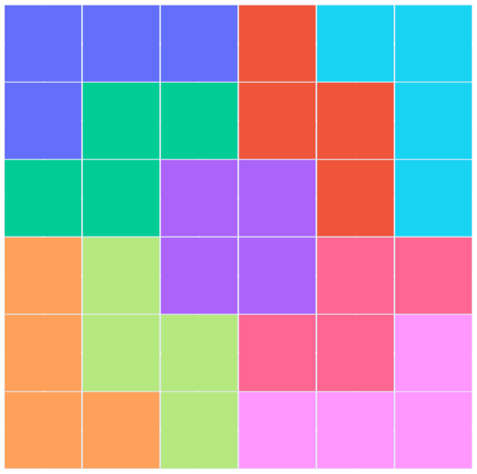
Fitness: 2000/2000
Pieces used: 5/5



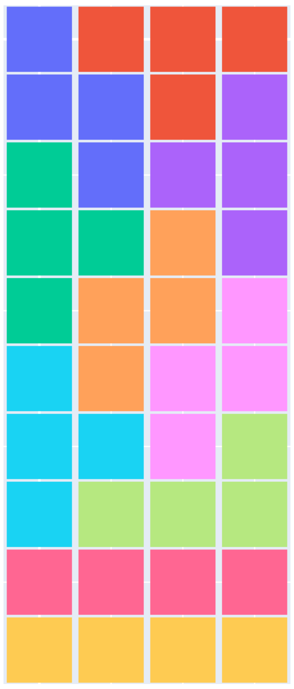
Fitness: 4800/4800
Pieces used: 12/12



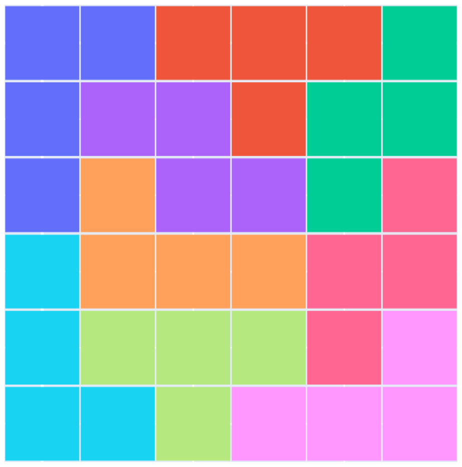
Fitness: 3600/3600
Pieces used: 9/9



Fitness: 4000/4000
Pieces used: 10/10



Fitness: 3600/3600
Pieces used: 9/9



Below is the best solution to our biggest puzzle of size 8x12.

Fitness: 9166/9600
Pieces used: 23/24