Natural Computing Assignment
2023/2024

November 16, 2023

Abstract

This assignment investigates three problems related to natural computing algorithms. The first task requires analyzing the effect of cooperation, the second problem asks to solve a simple game and experiment with the algorithm parameters, while the third one involves more advanced concepts related to handling sub-routines. For each problem, numerical and graphical analysis is performed, as well as further interesting points are made. The appendix provides some further information and clarifications but can be skipped if everything is clear. All the code is available and is attached together with this report.

Contents

1 Problem 1

1.1 Question A

1.1.1 Setup

Metaheuristic choice. For this problem, the primary choice is *Particle Swarm Optimization* (*PSO*) [1] algorithm. It is inherently cooperative - each particle interacts with each other via the global best position allowing them to gradually shift towards it while exploring space local to them and update each other whenever better positions are found. The algorithm is also simple, efficient, applicable to many problems, and easy to parallelize, which is very useful for experiments.

Objective choice. Many possible objectives range from trivial problems, such as all-ones [2], to real-world problems, such as solar photovoltaic systems [3]. Since the task requires performing theoretical analysis, it is best to have a well-defined objective that can be visualized and expressed mathematically. For this reason, the suggested *Rastrigin* function [4] was chosen with specific bounds, but the negative of it was returned to convert the task to maximization:

$$f(\mathbf{x}) = -\left(10D + \sum_{d=1}^{D}(x_d^2 - 10\cos(2\pi x_d))\right) \quad (1)$$

$$\text{where } \mathbf{x} = \begin{pmatrix} x_1 & \cdots & x_D \end{pmatrix}^\top$$

$$\text{and } x_d \in [-5.12, 5.12], \text{ for } d = 1, ..., D$$

The function for $D = 2$ can be visualized in Figure 3. As D increases, the problem difficulty increases due to the exponential growth of the number of local minima in space, which is also known as the curse of dimensionality [5].

Configuration. The chosen constant number of total evaluations was 5000 - it is enough to determine the general behavior as the population size changes. The chosen initial dimension size was $D = 25$ since it emphasizes well enough what happens to the fitness score as the population size changes, even if the optimal solution is not found. The chosen parameters for the PSO algorithm were as follows: $\omega = 0.5$, $\alpha_1 = 2.5$, $\alpha_2 = 1.5$, $\mathbf{v}_{max} = 0.1 \times (x_{max} - x_{min})$ and the global best update strategy was set to `best_ever`. These values showed the best results when evaluated for various large D (> 100). All experiments were seeded to be reproducible and multiple runs (typically 100) were performed and averaged for any combination of parameters to get more stable results.

1.1.2 Optimal Population Size

For this experiment, the population size was varied from 1 to 100, i.e., $N \in [1, 100]$. For every population size, 500 experiments were performed, the results of which were averaged. Each experiment produced a fitness score of the best particle at the end of all generations.

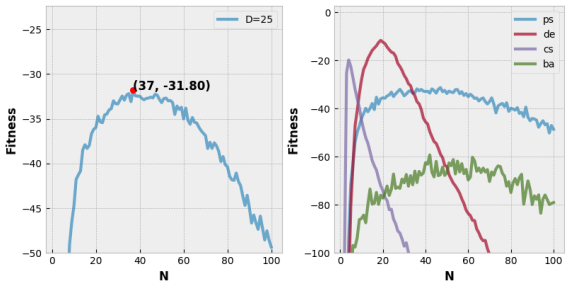


Figure 1: Left: optimal population size (marked with a dot) for negative *Rastrigin* objective with $D = 25$. Right: Fitness of different metaheuristic algorithms against population size for negative *Rastrigin* objective with $D = 25$.

Figure 1 (left) shows that the optimal population size for $D = 25$ and `num_evaluations` $= 5000$ lies within $N^* \in (30, 50)$ and with more experiments it would probably be maximized at around 40 (current averaged runs and noise give 37 as the maximum). We can see that the fitness dependency on N is non-linear - it steeply increases for $N < 20$, after which it slows down, and, eventually, after $N = 37$, starts decreasing further gaining some acceleration, but the change in slope is not as abrupt as when it was increasing.

The initial increase can be explained by the fact that, the more particles we add, the more cooperation happens in the search space - the global best is updated more optimally since more particles cover a larger space area. For $N < 5$, this effect is so prominent that we cannot even see the low fitness scores - this makes sense because a couple of particles can barely rely on the global best - there is not enough evidence for it to be a strong attraction point.

The decrease of the performance after $N = 37$, however, is inevitable because, the more particles there are, the fewer generations are performed (since $G = \frac{E}{N}$, where G - # generations, E - # evaluations, N - # particles). This means that each particle updates its position fewer times, effectively slowing down the space exploration, which causes the presence of many particles not to have the same benefits as just having fewer particles to adjust their position more often. That is because for larger N more time is needed (more genera-

tions) for the updates to catch up and outweigh the effect of initial randomness.

The non-linear dependency is not surprising - it means that, as N changes, fitness does not change by some constant factor, but rather based on the explorative and exploitative behavior of the algorithm (which is influenced by a specific parameter configuration) and by the objective surface. In other words, there is a range, rather than a fixed point, at which particle interaction for a fixed number of evaluations prevails and shows smooth results; in our case, such range would be between 20 and 60.

1.1.3 Other Metaheuristics

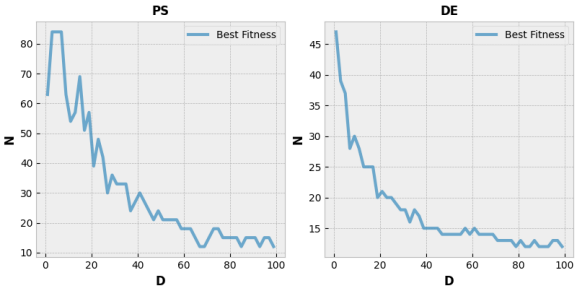
Another interesting experiment involves varying the algorithm type. Three further metaheuristic algorithms were implemented that also display cooperative behavior, each in its way: *Differential Evolution* (*DE*) [6], *Cuckoo Search Optimization (CSO)* [7], and *Bat Algorithm* (*BA*) [8]. See subsection A.1 for more details about each. Plotting these along the original *Particle Swarm Optimization* algorithm results allows us to get a more general idea about the optimal population size (Figure 1 right).

The optimal population size, as well as the fitness curvature, depends on the algorithm because some may be faster, have better-tuned parameters, or have a more unique behavior. For example, the steepness of *Differential Evolution* could be explained by the fact that when there are a lot of particles, too much variation and uncertainty may happen (because *DE* randomly samples 3 vectors from the population when updating the particle) causing the algorithm to diverge more quickly. However, the general trend suggests that there is always a steep increase at the start, after which the fitness gradually decreases as N grows, reinforcing our idea that having more particles is useless if they cannot perform any local search (or can perform only to a minimal extent).

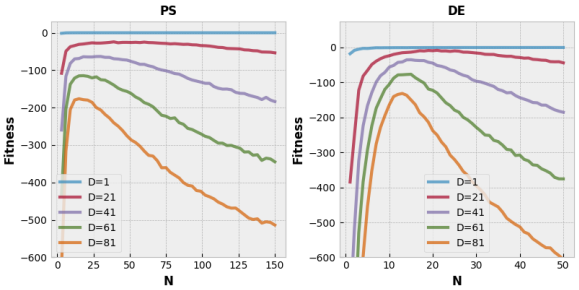
1.2 Question B

1.2.1 Varying Dimension Number

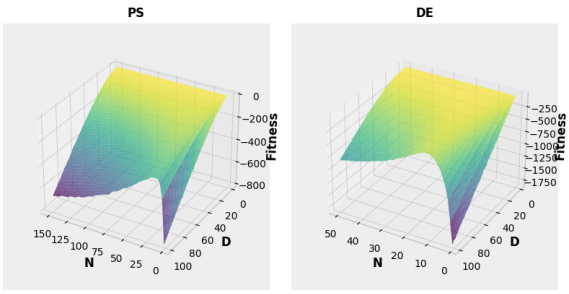
Considering *Rastrigin* function, the complexity of this objective can be changed by increasing or decreasing the number of dimensions. To check the relationship between the optimal population size and the number of dimensions, experiments were performed on a Cartesian product between a set of population sizes $N \in \{1, ..., 100\}$ and a set of dimension sizes $D \in \{1, ..., 100\}$. For each pair, 200 experiments were performed and averaged. Experiments were performed for *PSO* but *DE* algorithm was also investigated to have more certainty about the behavior.



(a) Population size against the number of dimensions



(b) Fitness against population size for different D



(c) Fitness against population size and dimensions

Figure 2: Optimal population size dependency on the number of dimensions for the negated *Rastrigin* function

Figure 2 shows three subfigures. The first one plots the optimal population size (the smallest number of particles required for the fitness to be as high as possible every time the algorithm terminates) against the number of dimensions. In general, the optimal population size has a clear dependency on the problem difficulty - as D increases, fewer particles achieve the best fitness. This time, the trend is non-linear due to the curse of dimensionality - the problem becomes exponentially harder as D increases.

For low dimensions ($D < 10$), we can afford to have many particles since even a small number of generations will be enough to determine a good solution (because the search space is small) - the more particles there are, the better the exploration is (of course, up to a certain extent, as discussed in the previous question).

For higher dimensions ($D > 50$), we prefer more generations, i.e., more updates than more particles, which makes sense because the space becomes so vast that there is not enough time for certainty about good regions of space to accumulate (for example, if $N = 50$, we only have $5000/50 = 100$ total generations!). This constitutes a fact that the initial randomness is scaled by a factor of D, which has to be compensated by the number of generations, and, therefore, having many particles is not enough to outweigh the benefits of just having fewer particles updating more frequently. Thus even if more particles gather better evidence for the update direction, which is especially useful for large D, this is not sufficient to get a better fitness score because this evidence is generated less often for larger N.

The second plot shows the curve profiles across different dimensions are similar, which aligns with our analysis for a fixed problem difficulty in subsection 1.1. We can see the peaks of the curves are slightly skewed (perhaps more obviously in *DE* plot), which corresponds to the optimal N for every D shown in the first plot. The third plot just presents a 3D illustration - fitness change along the N axis matches with our analysis in the previous sub-question, and fitness change along the D axis is rather obvious - the more difficult the problem is - the lower the fitness score. The combination of both of these effects forms the corresponding surface. We can also see that the curve of the optimal N for different D can be acquired by projecting the highest fitness values to the plane formed by N and D, which is exactly what is shown in the first subplot.

Having both *PSO* and *DE* plotted suggests that this trend between D and optimal N is rather general, although certain algorithms will have unique behavior. For example, in this case, *Differential Evolution* prefers lower population sizes as it seems to be more vulnerable to the increase of dimensionality (we can see, e.g., that at $D = 61$, the fitness is already at < -300, whereas for *PSO*, it is just below -150), since it does not keep the memory of the global best and tries to adjust the updates based on random particles, which is particularly bad for high D due to multiplied initial randomness.

What is also worth noting is that, if we look at a different scale, e.g., when D is very large (> 1000), the trends are a bit different from those in Figure 2. See subsubsection A.3.1 for more details.

1.2.2 Varying Objective Function

Another aspect in which the problem complexity can be varied is by making the objective itself more difficult. Along with *Rastrigin*, five other dimension-based objectives were investigated (in the order of complexity): *Sphere* [9], *Griewank* [10], *Ackley* [11], *Rosenbrock* [12], *Schwefel* [13], which are all visualized in Figure 3 for $D = 2$, but note that the result of each function was negated to turn the problem into a maximization task. The difficulty for all of them increases with the number of dimensions. For more details about these objectives, see subsection A.2.

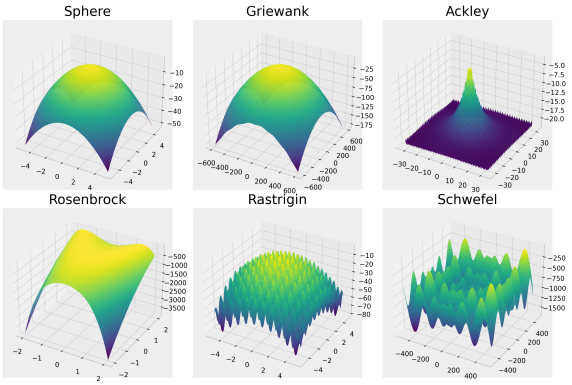


Figure 3: Common optimization objectives that depend on input dimensionality D (here $D = 2$), the simplest one is the *Sphere* function and the hardest one is *Schwel* function with many irregular local optima. A small note that the surface of *Griewank* may look smooth but it also has regularly distributed many small local optimum points.

Figure 4 shows fitness against population size curves for different objectives with $D = 25$ using *PSO*. We can see that more complex problems prefer a higher population size, e.g., for *Rastrigin* the optimal range is 30 to 50, whereas for *Ackley* it is 20 to 40. This is the opposite of what was observed when the difficulty was changed with respect to the problem itself - lower population sizes yielded better fitness, but here the more difficult the problem is, i.e., the more irregular its space is, the higher the population size is needed, on average, to achieve the best results. This means that, for each objective, there is a specific range of the population size at which the particles interact the best for a fixed number of evaluations. For example, *Sphere* function has no local optimum points,

thus it is sufficient to just have a few particles interact to reach the global optimum, whereas *Rastrigin* has many local optima, requiring more particles to interact at the cost of fewer generations to achieve the best fitness score.

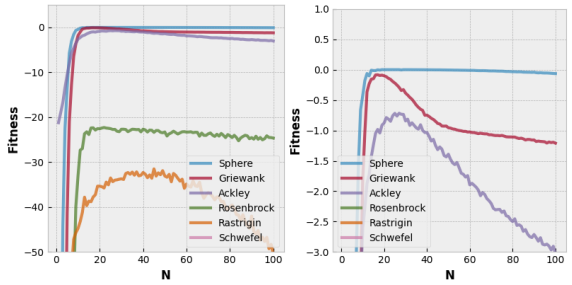


Figure 4: Fitness dependency on the population size N for different objectives. The left plot shows some of the zoomed-in objectives (except Schwefel) and the right one zooms in even more towards the best objectives. The algorithm is *PSO*, $D = 25$ and `num_evaluations` $= 5000$.

This behavior, however, changes when D is high - more difficult problems prefer lower population sizes than the simpler ones, which is again, due to the initial randomness that disappears quicker for problems with fewer local optimums but has a big impact on objectives where more exploration and updates are needed to shift towards better solutions. Figure 5, although noisy due to fewer runs, depicts this. For every objective, the optimal population size as D increases still follows the same trend described earlier for specifically *Rastrigin* objective, although it can vary a little, depending on the function and the algorithm type. For *Sphere*, particularly, the start of the curve is rather different, but that is only because this function is very simple and even a few particles reach the global best, i.e., a fitness of 0, there is no need to choose a larger population size.

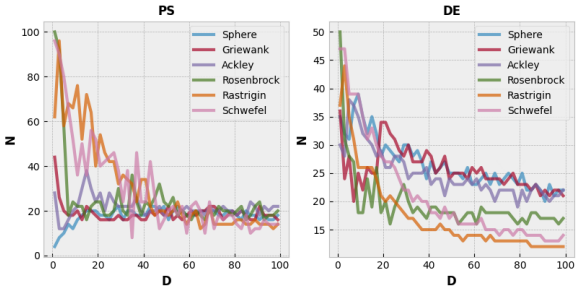


Figure 5: Optimal population size dependency on the number of dimensions for the negated common objective functions with $D = 25$ and algorithms *PSO* and *DE*.

To summarize, the optimal population size decreases as the dimension size increases and this trend is generally applicable to every objective analysed. However, if we want to specifically model our solution for a fixed dimension size, then this optimal number will be different for each objective since it will depend on its complexity for that particular D, otherwise, it may not be fair to compare the objectives at a fixed D.

2 Problem 2

2.1 Setup

2.1.1 Game Generation

For Sumplete [14] game generation, the suggested values were adopted - the deletion rate was chosen $\frac{1}{3}$, the integer value range was taken $G_{ij} \in [1, 9]$, and the board size $k \geq 3$ was varied.

It is important to note that, whenever complexity in this question is mentioned in terms of specific k values, it is only applicable to this specific game configuration.

For instance, if the allowed cell value range was only $G_{ij} \in \{1, 2\}$, then even for small k, it would be a very difficult problem since the algorithm would tend to stuck very easily as there would be many correct deletion criteria row-wise and column-wise. See subsection B.1 for more discussion.

2.1.2 Encoding

The chosen encoding is a $k \times k$ binary matrix that is flattened to a vector of dimension $D = k \times k$. The solution can be naturally represented as a binary genome. In other words, for every game, we have 2^{k^2} possible states, where each cell can be either marked as "deletable" (takes the value of 1) or not (takes the value of 0). Therefore, if we have, for example, the following game for $k = 3$:

$$G = \begin{array}{ccc|c} 7 & 9 & 1 & 8 \\ 5 & 7 & 8 & 12 \\ 5 & 7 & 8 & 15 \\ \hline 12 & 14 & 9 & \end{array}$$

a solution, or a genome, would be represented as a binary vector:

$$\mathbf{x} = \text{flatten}\left(\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}\right)$$
$$= \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}^\top$$

This way all the search space can be covered without any redundant solution representations (because any change in value would result in a valid different state of the game).

This representation completely covers the problem, which is discrete, is easy to understand, and is compatible with many genetic algorithm operators [15] which can be explored for this problem.

2.1.3 Algorithmic Details

Selection

Although primarily *roulette-wheel selection* [16] was considered, which samples individuals according to a probability distribution based on their fitness values, it was soon discovered that this strategy applies too much selection pressure, causing the solutions to easily get stuck in a local optimum for $k > 6$.

For this reason *tournament selection* [17] was applied, which samples T individuals N times and advances the best one to the next generation, where T - tournament size and N - population size. This technique maintains more diversity, because, even if relatively bad genomes survive, they carry information potentially very useful for crossover when near-optimum solutions require some uniqueness, as well as keep a possibility to branch out to different directions when a lot of exploration is needed. Additionally, tournament selection performs better than other selection methods in terms of convergence rate and time complexity [18].

It is also worth mentioning that elitism [19, 20] was considered, where a fraction of the best solutions are directly passed to the next generation, bypassing crossover and mutation, however, this technique was not applied since, for $k > 6$, exploration is far more desired than exploitation, especially if exact solutions are required.

Crossover

Although there are many crossover techniques [21], most of them are developed for vector-based solutions. Because our genomes (if they were reshaped back to $k \times k$) encode a structural representation of the game, it might be worth considering spacial crossover techniques such as *matrix crossover* [22], where a submatrix of genes are swapped. In our case, it might be particularly useful to consider swapping a certain fraction of rows and columns since they encode partial solutions (a row or a column must sum to a particular value). It was noticed that for low k, this method helps to converge significantly faster.

The problem, however, arises as we reach $k > 6$, where even if locally correct solutions are preserved, they more often than not correspond to some local optimum. For this reason, a multi-point crossover was also implemented [23], where K segments of random length are swapped between two $1D$ genomes. This effectively exchanges

bits of rows and bits of columns[1], introducing some stochasticity which is crucial when k is high. Additionally, this avoids the problem of exchanging fragments that contain no meaningful information during the first generations, if we were to stick with just spacial crossover.

Mutation

Although, like for crossover, there are many operators for mutation [24], the choice always depends on the problem restrictions [25, 26]. Our solutions do not encode complex structures, like sequences, or require unique values, thus simple methods, such as *bit-flip mutation* [27], are sufficient.

Bit-flip mutation indeed helps to introduce some diversity by flipping each gene with a certain probability p_m, however, our problem might require even more diversity, especially for large k, which could not be introduced by simply increasing p_m as it would randomize our solutions too much. Therefore, another mutation technique was adopted where a random fraction $p'_m \in [0, p_{\text{upper}}]$ of rows or columns are randomly regenerated[2]. Combined with *tournament selection*, this technique often helps to escape local optimum points.

Refit and Escape

Two further techniques were implemented after observing how new solutions are generated as the algorithm runs. The first one was called *refit* which reevaluates the fitness scores for each genome before selection. This is done by assigning a decay factor ρ for each all-time best solution representation. In other words, whenever a new best genome is found (based on its fitness score), it is added to the set of all-time local best solutions, and every time another genome reaches such representation, its fitness score is recomputed as follows:

$$f_n \leftarrow \rho^C \times f_n \tag{2}$$
$$C \leftarrow C + 1 \tag{3}$$

where C is the counter, i.e., the number of times this local best solution was reached by all the other genomes in the past and f_n is the fitness score of the n^{th} individual.

The second one is *escape* which tracks the mean solution representation and, if such mean genome is computed for `num_repetitions` times in a row, the genomes with the same representation as the mean genome or those with very similar structure are completely regenerated, i.e., the algorithm is partially restarted.

Refit, perhaps, was inspired by some reinforcement learning techniques, for example, in *Dyna-Q+* [28] the states that have not been tried for a long time are given some bonus reward and the agent is encouraged to try them, which is very useful when the environment changes. In our case, the game does not change but it is useful to decay the fitness ("reward") for local-optimum states since we know they are not exact solutions. *Escape* was implemented by simply observing how the solution distribution changes and noticing that they sometimes tend to converge to some mean value. Although these two techniques do not help as much for $k > 8$, they still keep the search going until the complete solution is found. Figure 6 shows obvious benefits of *refit* and *escape* strategy for $k = 7$.

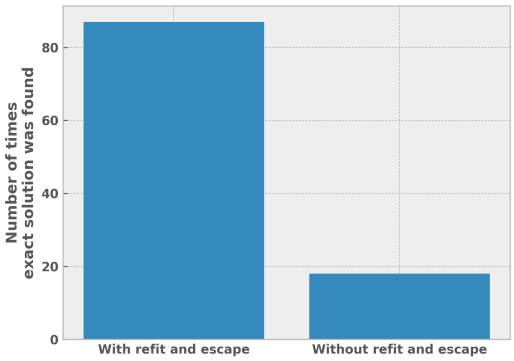


Figure 6: Then number of times exact solution was found using **row-col** fitness strategy evaluated for 100 runs, each for a maximum of 1000 generations. Tournament size was 4, crossover strategy was **n-point**, and mutation strategy was **row-col shuffle**.

Algorithm

A simplified function for running the genetic algorithm for the *Sumplete* game is presented in Algorithm 1. Notice how the algorithm is terminated once the total number of generations has been reached or a valid solution of the game has been encountered.

[1] In a typical $1D$ case, more bits of rows would be exchanged than of columns, since the solutions are flattened in a row-by-row manner. For this reason, 50% of the time, $1D$ vector pairs are reshaped to $k \times k$, transposed, flattened, crossovered, then again reshaped to $k \times k$, transposed, and flattened to $1D$ to keep the exchange between rows and columns fair.

[2] Actually, each selected row or column is shuffled rather than completely regenerated since the solution might also encode a total fraction of cells that should be deleted (also, implementation is more compact).

Algorithm 1 Simplified GA for *Sumplete*

Require: T ▷ Total timesteps
Require: $\mathbf{G}$ ▷ Game representation

function RunSumpleteAlgorithm(T, $\mathbf{G}$)
  $\mathbf{P} \leftarrow$ InitializePopulation()
  $\mathbf{f} \leftarrow$ ComputeFitnesses($\mathbf{P}$)
  $\mathbf{s} \leftarrow$ SelectBest($\mathbf{P}$, $\mathbf{f}$)

  while $T \neq 0$ **do**
    $\mathbf{P}, \mathbf{f} \leftarrow$ Refit($\mathbf{P}$, $\mathbf{f}$)
    $\mathbf{P} \leftarrow$ Sampling($\mathbf{P}$, $\mathbf{f}$)
    $\mathbf{P} \leftarrow$ Crossover($\mathbf{P}$)
    $\mathbf{P} \leftarrow$ Mutation($\mathbf{P}$)
    $\mathbf{P} \leftarrow$ Escape($\mathbf{P}$)
    $\mathbf{f} \leftarrow$ ComputeFitnesses($\mathbf{P}$, $\mathbf{G}$)
    $\mathbf{s} \leftarrow$ SelectBest($\mathbf{P}$, $\mathbf{f}$)

    if IsValid($\mathbf{s}$, $\mathbf{G}$) **then**
      break
    else
      $T \leftarrow T - 1$
    end if
  end while

  return s
end function

2.1.4 Parameters

Please see subsection 2.4 for the discussion of parameter choices. It also includes the default configuration.

2.2 Question A

A fitness of 1 for the correct solution and a 0 for all the incorrect ones (in this question, this evaluation is referred to as `absolute`) provides no information about how the generations should evolve. In other words, the selection of individuals is always random because their fitness scores are all equal and the algorithm terminates whenever a fitness score of 1 is found.

Essentially this means many random solutions are tried out until the correct one is found. Therefore, if randomness is not dealt with effectively, the "search" will happen very slowly. For this reason, a rather aggressive mutation was chosen - *bit flip* with $p_m = 0.5$ (see Figure 7). Crossover was mixed between *n-point* and *rowcol-swap* to introduce even more variation.

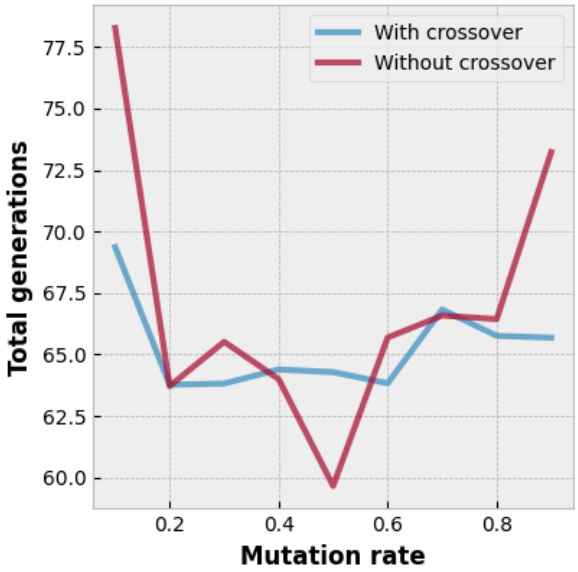


Figure 7: Total generations (averages across 1000 runs) that took to find a correct solution for $N = 1000$ using `absolute` fitness function. The blue line shows the dependency on the mutation rate without any crossover and the red one - with mixed crossover $p_c = 0.7$. Since the chosen crossover already introduces enough variation, the red line indicates that the mutation rate does not matter too much.

Mutation could be engineered more effectively if the problem allowed us to specify a satisfiable fitness score [29], however, in our case, it is a black-box problem. Still, perhaps a better algorithm would ensure the genomes do not repeat, at least when they are initialized, although this becomes less of an issue as k grows.

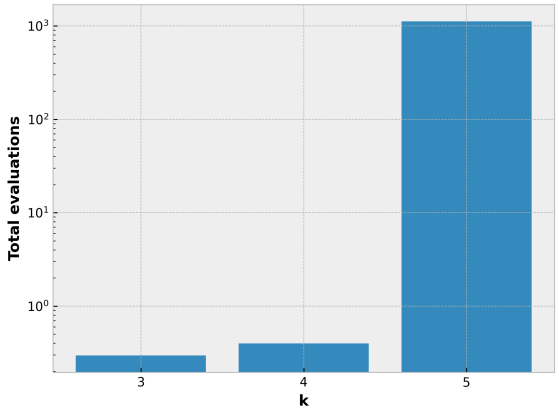


Figure 8: The number of total evaluations needed until the correct solution was found. The results are averaged across 10 runs.

Figure 8 shows the total number of evaluations needed to fully solve the problem for $k \in \{3, 4, 5\}$ (for $k > 6$, the solution would take a very long time). We could compare this with the number of possible board states for each k (as explained in subsubsection 2.1.2): for $k = 3$, we have $2^9 = 512$,

for $k = 4$, we have $2^{16} = 65,536$, and for $k = 5$ we already have over 33 million. This means the search space grows exponentially and, in the average case, we would need $\frac{2^{k \times k}}{2}$ unique tries until the correct solution is found. This suggests that the average number of generations for $k = 6$, even with a very large population, e.g., $N = 1,000,000$, would be huge: $\frac{2^{36}}{2 \times 1000000} \approx 35,000$, which would be pretty much infeasible to run with such N.

2.3 Question B

For this problem, the mutation rate was lowered since each of the fitness functions discussed further, provide some information about the selection criteria. In other words, we need less aggressive mutation to preserve the good genes that are identified as the algorithm runs. Three fitness functions were considered. All of them are normalized against the total number of rows and columns and scaled between 0 and 1.

Assume $\mathbf{s}_1$ is a vector or row sums, i.e, each i^{th} value tells to what number non-deleted i^{th} row values should sum up. $\mathbf{s}_2$, respectively, is a vector of column sums, as defined by the generated game. Further, assume $\mathbf{G}$ is the full $k \times k$ game board and $\mathbf{X}$ is a potential solution with ones at cells to be deleted and zeros at cells to be kept. Finally, $\hat{\mathbf{G}} = \mathbf{G} \odot (\mathbf{1} - \mathbf{X})$, where both operators are element-wise, is a $k \times k$ matrix with zeros corresponding to the deleted cells and the remaining values corresponding to the non-deleted cells.

Distance. The distance-based function computes the absolute difference between the sum of rows/columns and the corresponding values the sums should be equal to. The inverse of the normalized distance is taken ($\mathbf{s}_{i,j} \neq 0$):

$$d_1(\hat{\mathbf{G}}, \mathbf{s}, i) = \left| \mathbf{s}_{1,i} - \sum_{j'=1}^{k} \hat{\mathbf{G}}_{i,j'} \right| \tag{4}$$

$$d_2(\hat{\mathbf{G}}, \mathbf{s}, j) = \left| \mathbf{s}_{2,j} - \sum_{i'=1}^{k} \hat{\mathbf{G}}_{i',j} \right| \tag{5}$$

$$f(\hat{\mathbf{G}}, \mathbf{s}) = \left(1 + \sum_{i=1}^{k} \sum_{j=1}^{2} \frac{1}{2\mathbf{s}_{j,i}} d_j(\hat{\mathbf{G}}, \mathbf{s}, i) \right)^{-1} \tag{6}$$

Row-col. This function checks how many rows and columns are correctly solved:

$$e_1(\hat{\mathbf{G}}, \mathbf{s}, i) = \begin{cases} 1, & \text{if } d_1(\hat{\mathbf{G}}, \mathbf{s}, i) = 0, \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$e_2(\hat{\mathbf{G}}, \mathbf{s}, j) = \begin{cases} 1, & \text{if } d_2(\hat{\mathbf{G}}, \mathbf{s}, j) = 0, \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

$$f(\hat{\mathbf{G}}, \mathbf{s}) = \frac{1}{2k} \sum_{i=1}^{k} \sum_{j=1}^{2} e_j(\hat{\mathbf{G}}, \mathbf{s}, i) \tag{9}$$

Cellular. This function is similar to the previous one, except it counts the number of cells that appear in both correctly solved rows and columns:

$$c(\hat{\mathbf{G}}, \mathbf{s}, i, j) = \begin{cases} 1, & \text{if } d_1(\hat{\mathbf{G}}, \mathbf{s}, i) = 0 \\ & \text{and } d_2(\hat{\mathbf{G}}, \mathbf{s}, j) = 0, \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

$$f(\hat{\mathbf{G}}, \mathbf{s}) = \frac{1}{k^2} \sum_{i=1}^{k} \sum_{j=1}^{k} c(\hat{\mathbf{G}}, \mathbf{s}, i, j) \tag{11}$$

Mixed. The mixed metric is just an average of all three evaluation functions. This, however, makes the fitness computation much slower, especially for large N.

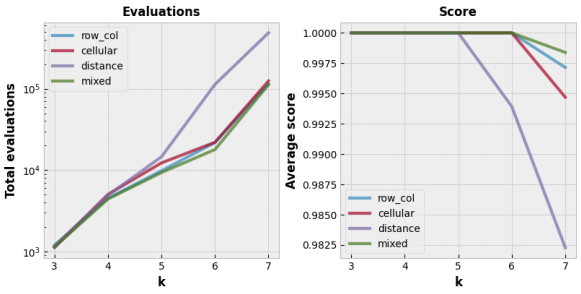


Figure 9: The number of total evaluations needed until the correct solution was found for each k. The results are averaged across 100 runs.

Figure 9 shows the performance for each of the discussed fitness functions and a combination of them. Experiments with up to $k = 7$ were performed, meaning each function already provides some information that the algorithm can utilize. Although **distance** metric provides more information about how far or close to the sub-solutions we are, low distance does not mean we are close to *globally correct* cases. Therefore **distance** potentially introduces even more local minima. **Row-col** and **cellular**, on the other hand, do not provide as much information and simply encourage the algorithm to try more until the correct sub-solutions are found. The key difference is that these sub-solutions are checked whether they are *globally correct* only after they are generated[3], not when they are partially complete which the **distance** metric takes into account. Ultimately, the best performance is acquired when we combine these metrics.

[3]This means there are fewer points in the fitness space which also means there are fewer local optimum points in the search space that the algorithm could get stuck at.

2.4 Question C

The discussion of the effect of the parameters is split into 4 parts. The experiments were performed on certain parameters while keeping the others fixed. See subsection B.2 about the default configuration and an explanation of each parameter. Please also note that experiments were only performed on some of the most important parameter groups. Additionally, please note that there are many ways to choose the parameters and even to make them adaptive [30, 31], however, in this question, simply different values of the parameters are checked, and the effect they cause.

For each parameter configuration, 50 experiments were averaged with a population size of $N = 1000$ and 1000 maximum generations. The experiments were done across different k and we consider the total number of fitness evaluations as our performance indicator. It was noticed, however, that the exponential dependency absorbs the information about the parameter dependency, even if we consider log scale. For this reason, all the plots contain normalized numbers of total evaluations. See subsection B.3 for more details about this normalization rule and the parameter plots at original log scales, i.e., after the 1st normalization step.

2.4.1 Selection

The chosen selection method was **tournament selection** which only has one parameter - `size` of `tournament` which controls how many individuals are selected that will be compared against each other in that tournament group. Figure 10 illustrates that, as k grows, the smaller size is preferred, which makes sense as it helps to keep the diversity which is crucial for this kind of problem.

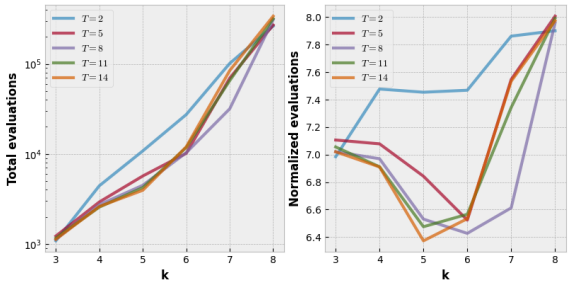


Figure 10: `Tournament size` effect on the algorithm. Left: we consider the log scale of total evaluations. This is just to show that even at log scale a lot of information may seem lost due to log-linear dependency. For more such plots, check subsection B.3. Right: we consider the normalized scale. In the rest of this section, only this kind of scale is considered.

We can see, for example, when $k = 5$, the best size is 14, which causes the algorithm to converge faster due to higher selection pressure. At $k = 7$, the optimal size drops to 8, and at $k = 8$ it drops as low as $k = 2$. It becomes even more obvious if we look at the blue curve which suggests that a tournament size of 2 is never a good choice, except when the problem becomes very hard, as it is when $k = 8$[4].

2.4.2 Crossover

Two crossover strategies were tested - **n-point** and **row-col swap**. The first one has a parameter `num_cross_point` which controls how many segments the genome is partitioned to before every other is swapped. Figure 11 (left) shows that there is no clear winner, however, we always prefer at least a few points. The results are quite noisy too since each segment length is random. We can still see that, for lower k, having more points, e.g., 4 (green line), is not preferred, however, for larger k, such as $k = 8$, it is not a problem. Perhaps what could be inferred is that, if k were to grow further, more points would be beneficial since it would more likely prevent premature convergence.

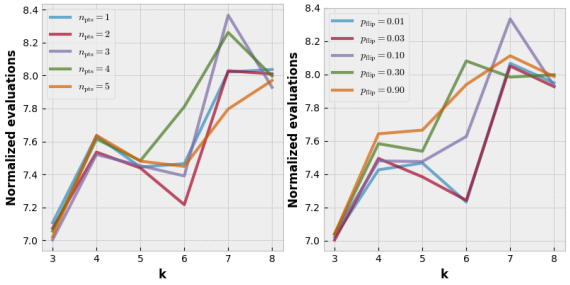


Figure 11: Left: `number of cross points` effect on the algorithm for $p_c = 0.7$ and crossover strategy **n-point**. Right: p_{flip} effect on the algorithm for $p_m = 0.2$ and mutation strategy **bit flip**.

For this reason, the number of crossing points was set to adopt dynamically: floor($\frac{k}{2}$). For **row-col swap**, we similarly have a parameter that controls the number of rows and columns to be swapped. Instead of performing another set of experiments, we take the inspiration of the previous experiment which suggests that there will be a dependency on k and set this number to also be floor($\frac{k}{2}$), i.e., 50% of rows or columns will be swapped regardless of k.

[4]Please note that this is specific to the population size as well. If N grew exponentially with k^2, most likely we would be fine even performing **roulette wheel** selection since we could afford high selection pressure. Of course, having such a large N would be infeasible.

Considering the crossover rate (fraction of individuals chosen for mating), Figure 12 shows that p_c has a rather uniform preference - generally, at least half of the population should be considered for mating. Lower values like $p_c = 0.2$ do not encourage new solutions to be formed because only 20% of the individuals exchange genes. It may be worth noting that **row-col swap** keeps the algorithm more stable (there is less variation in the curves too) since the swapped rows or columns are always of the same size. Additionally, for lower k values, **row-col swap** even performs better, i.e., requires fewer evaluations to reach the solution.

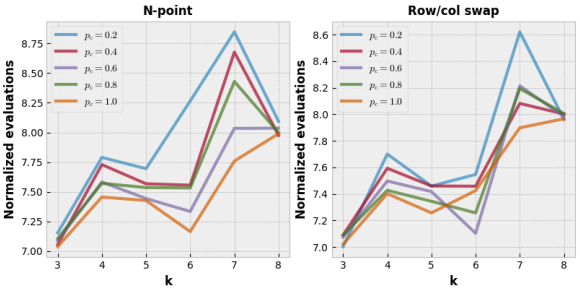


Figure 12: Crossover rate p_c effect on the algorithm for **n-point** and **row-col swap** crossover strategies.

2.4.3 Mutation

The mutation was also tested on two strategies - **bit flip** and **row-col shuffle**. The first one has a parameter p_{flip} which controls the fraction of genes being flipped in an individual. Figure 11 (right) shows this parameter's effect on the algorithm as k grows. Almost always a small value $p_{\text{flip}} < 0.05$ is preferred, except for $k = 8$, where $p_{\text{flip}} = 0.1$ takes the lead. This just shows that a more aggressive mutation causes good genes to be lost, however, again we would expect this number to grow (for a constant N) since introducing more randomness would speed up the search.

A similar argument could be applied for the fraction of rows and columns to be shuffled, which is a parameter for **row-col shuffle** strategy. In fact, the same argument could be applied to the mutation rate p_m parameter which, in our case, controls the fraction of individuals (see Figure 13).

Note how the algorithm consistently prefers $p_m \leq 0.1$ for both strategies and, when $k = 8$, a high mutation rate $p_m = 0.9$ becomes even better than some of the lower mutation rates, which is in alignment with what we stated earlier. Also note that similarly to **row-col swap** crossover, **row-col shuffle** mutation performs better than the non-row/column-based **bit flip**, at least for the lower values of k.

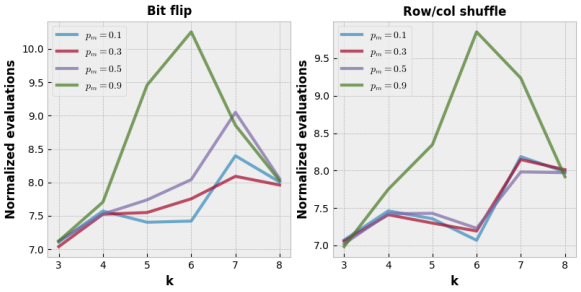


Figure 13: Mutation rate p_m effect on the algorithm for both **bit flip** and **row-col shuffle** mutation strategies.

2.4.4 Algorithmic

Finally, for completeness, it is worth mentioning the effect the population size and the number of generations have. Figure 14 shows that a larger population size is always preferred, regardless of k, which is natural, since the search space is covered by more individuals. A similar situation is for the number of generations - the larger the number, the more certain we are the solution will be found since it is unlikely for the algorithm to get stuck indefinitely at a single local optimum due to **refit** and **escape** strategies, however, the search could very well happen indefinitely for very large k.

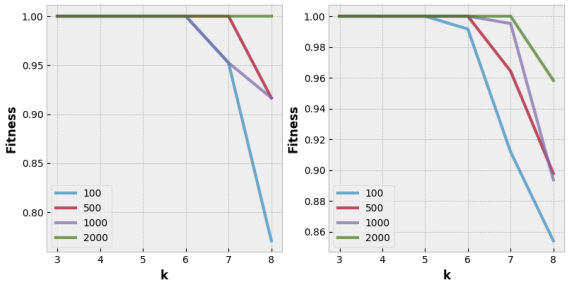


Figure 14: Left: population size N effect on the algorithm. Right: maximum number of generations effect on the algorithm.

Of course, we may not want to set N to be too high when we have small values of k, for instance, if we have $k = 3$, there are only $2^9 = 512$ possible game states, thus a population of $N = 500$ would almost guarantee the solution is found just by purely initializing the algorithm and very rarely it would require few or more generations.

3 Problem 3

3.1 Question A

3.1.1 Choice of Sequence

The primary chosen sequence is the **Fibonacci** series [32] which starts with $y_0 = 0, y_1 = 1$[5] and every subsequent number is the sum of the previous two:

$$y_x = y_{x-1} + y_{x-2} \tag{12}$$

For example, for $x = 7$, we would have the following sequence that can be expressed in vector form:

$$\mathbf{y}^{(x=5)} = \begin{bmatrix} 0 & 1 & 1 & 2 & 3 & 5 & 8 \end{bmatrix}^\top \tag{13}$$

Although it is a rather simple choice, it is a good starting point to test whether the designed *genetic programming* (GP) algorithm is capable of dealing with sequences in general. In other words, we model the sequence generation problem not just based on the mathematical rule (e.g., Equation 12) but also on the algorithm execution rule, i.e., the programming steps that would give us the full series (e.g., Equation 13).

Depending on the GP design, it could be much harder to unravel the full algorithm flow since it by itself subsumes any kinds of arithmetic/ logic/ relational etc rules. Thus, **Fibonacci** sequence is a simple case that allows us to test, at least in theory, whether our GP algorithm is capable of handling the dependencies on the previously executed parts of the built program (a necessary component in generating the full series).

3.1.2 GP Design: Representation

Program. The chosen representation for an individual is a tree, which is a standard approach [33] where branch nodes are *non-terminals*, e.g., +, max, and leaves are *terminals*, e.g., 1, π. There are many other kinds of representations [34, 35, 36, 37], many of which could be even more suitable for series problems, for example, in Self-Modifying CGPs [38], phenotype graphs encode dynamic execution flows controlled by genotypes, allowing better exploration of sub-routines and better generalization [39]. For the sake of simplicity and easier interpretation, we stick with the tree structures.

[5]The chosen representation is such that each element in a sequence is denoted by y and the position is denoted by subscript x.

Terminals. Two types of *terminals* were considered: numeric values, e.g., -3, 0, and numeric vectors, i.e., lists. Both types can take the form of a constant or a variable - the leaf is labeled accordingly. For the sake of simplicity, we only consider a few integer constants (we have background information that real values are not needed, additionally, they might only approach the solution without solving it exactly) and two variables - an integer and a list - as *terminals*.

Non-terminals. Only binary *non-terminals* were considered - this causes deeper and harder to understand trees and restricts the search space when we introduce depth limits, however, it makes it easier to handle the constraints since most of the operators are binary. Due to our problem requirements, three types of *non terminals* were constructed: algebraic, e.g., addition (+), subtraction (−), indexable, e.g., `push` (takes a list and a value) and `get` (takes a list and an index), and one flow operator - `for` (takes the number of loops and an expression). See subsection C.1 for a more comprehensive explanation of all the chosen and considered *non-terminals*.

3.1.3 GP Design: Algorithmic Details

Evolution. The evolution of each generation consists of four phases: *selection*, *crossover*, *mutation* and *validation*. Similar to arguments presented in subsubsection 2.1.3, *tournament selection* was chosen for a low T (2 to 5). It is helpful to keep some less fit programs throughout generations as they may contain locally good components, useful for more diverse recombination. In GPs, a common technique for *crossover* is sub-tree swapping [40, 41], which is implemented in this work for randomly selected sub-trees. Mutation typically has more variants [42, 43, 44] and here the best results are acquired by mixing different kinds of them, including *grow*, *shrink*, *renew* etc (see subsection C.2 for a more detailed explanation). *Validation* simply trims the trees that exceed the maximum allowed depth and regenerates new trees for those below the minimum depth.

Fitness. The fitness is constructed to test not only the value correctness in the produced sequence $\hat{\mathbf{y}}$ but also the correctness of the output type. More specifically, 25% of it is dedicated to test the predicted sequence length, another 25% to check the correct amount of the unique values, and the remaining 50% to check the correctness of those val-

ues[6]. Each check uses an inverse loss function, primarily with *mean squared error* (MSE)[7] [45]:

$$\text{fitness}_i = \frac{1}{1 + \text{MSE}(\mathbf{y}, \hat{\mathbf{y}})} \tag{14}$$

where $\mathbf{y}$ refers to the ground-truth sequence up to some step x and $\hat{\mathbf{y}}$ refers to its prediction. For array length and unique value checks, these vectors are of fixed length 1. Each fitness_i is summed with the weights specified previously. The value of the total fitness ranges between 0 and 1 and a fitness of 0 is given for any output that is not a list.

3.2 Question B

3.2.1 Setup

The results of the GP algorithm are not only presented in terms of the constructed rule for **Fibonacci** series but also in terms of its performance, i.e., the number of evaluations needed for different GP configurations. This shows how sensitive our GP design is.

As a baseline, a tournament size of 20 is used, $p_c = 0.7$ (the fraction of mating individuals), $p_m = 0.5$ (the fraction of mutated individuals), and $N = 10^3$. These parameters showed generally satisfiable results for different algorithm configurations, and are further not explored since the task does not require that. A larger mutation probability is for more aggressive exploration which helps find an exact solution, not an approximation. The algorithm stops once the exact solution is found.

The *terminal* set is always the same: we have integer constants $c \in [-2, 2]$ and two input variables: x (the remaining sequence length) and s (the list of starting sequence values). The default *non-terminals* include `add`, `mul`, `push`, `get` and `for`. The default minimum program tree depth is 2 and the maximum is 5.

Please see subsection C.3 that explains how to interpret a graphical representation of a genetic program tree for an ideal solution for **Fibonacci** series.

[6] If the lengths of the actual and the predicted sequence do not match, we compare the shorter sequence to the left slice and to the right slice of the longer sequence, with 25% weights.

[7] Since we use *tournament selection*, we only care about the ranking of individuals, not their absolute fitness values. Thus a specific loss function does not matter too much as long as it would rank individuals from best to worst correctly. We therefore just stick with the common MSE.

3.2.2 Time

Here we briefly investigate the fitness changes through time for the default setup of our GP algorithm. Figure 15 illustrates that during the first 20 generations, on average, the fitness increases faster, after which it slows down and slowly over time reaches the best score.

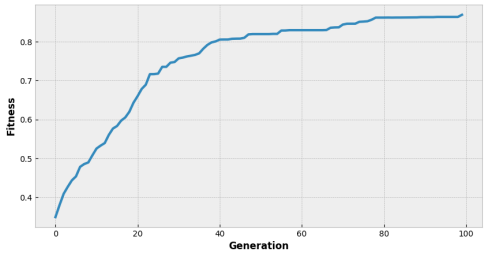


Figure 15: The average fitness at each evolution step for the default GP configuration. The total number of generations was set to 100 and a total of 50 runs were averaged.

Please note that, if we were to consider only a single run, we would have irregular jumps whenever good components are found. If we inspect the curve, we could hypothesize that the following events take place:

- Just by initializing the algorithm, we already have components that push to a list, i.e., $\texttt{for}(x, \texttt{push}(s, \cdot))$ is relatively easy to find as it is the very first aspect that contributes to the fitness function and takes a maximum value of 0.25 if the produced sequence length is correct.
- Over the next 20 generations the algorithm tries to identify what makes the sequence unique (since uniqueness also contributes up to a maximum of 0.25) and we could suspect that, by generation 20, subtrees with components like $\texttt{get}(s, -2)$ or $\texttt{get}(s, -1)$ have been identified as useful. Having an expression, e.g., $1 + \texttt{get}(s, -1)$ pushed to the sequence at every iteration makes it increasing which not only gives a maximum fitness in terms of unique values but also in terms of value correctness (determined through MSE). This is the reason we have a sharper initial growth.
- From the 20th generation onward, it takes some time for the crossover and mutation to find the final suitable `get` component and the correct `add` *non-terminal* that correctly connects them, but the slowing down of the curve suggests that by 80th generation, most of the runs would have finished.

3.2.3 Depth

This section shows the results of our GP algorithm in terms of its capability of finding a solution over different maximum allowed tree depths. Figure 16, although noisy, generally suggests that a larger maximum depth means fewer times a correct rule will be found for a fixed number of generations.

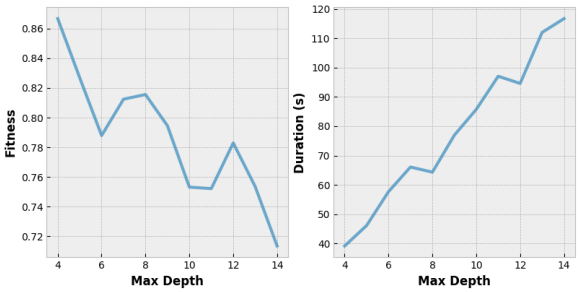


Figure 16: The average fitness score and the duration of finding the solution for different maximum depths. Experiments were averaged over 50 runs with a maximum of 50 generations per run.

This makes sense because the search space expands and the algorithm needs more effort. Sometimes the algorithm can perform quite well even at large depths (we have spikes at max depth 8 and 12) which suggests that it is possible for useful components to be maintained through recombination, instead of being pruned, and, in combination with invalid subtrees (e.g., adding a list to a scalar is the same as adding 0) or expressions that negate each other, they become dominant. Still, the time needed to execute taller trees grows at least linearly (we prevent nested execution of `for` loops) with depth. Therefore, better results can be considered if the trees are small.

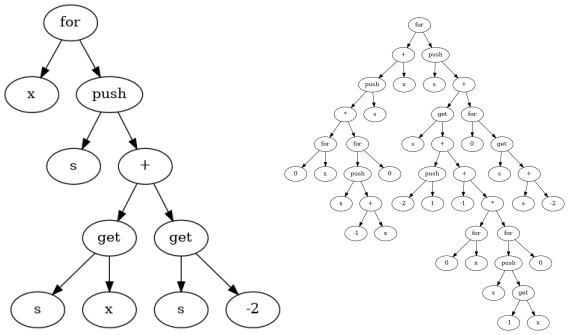


Figure 17: Identified solutions for **Fibonacci** series. Left: a solution with the maximum allowed depth of 4. Notice how `get`(s, x) is the same as `get`$(s, -1)$, i.e., x is clipped to be the maximum allowed index of s. Right: a solution with a maximum allowed depth of 10. Notice, for example, how the subtree of the first element of `for`, i.e., $+$, evaluates `push` to 0, since we cannot push to anything that is not a list, thus effectively we add 0 to x and get the same expression as `for`$(x, \cdot)$.

Figure 17 shows that, if the maximum allowed depth is large, then individuals' sizes grow excessively, a problem often characterized as **bloat** [46], which causes hard interpretability and could even lead to overfitting. For this additional reason, we always prefer smaller trees but the maximum size we allow will depend on the background knowledge we have.

There are, however, ways to make the solutions easier to understand, for instance, we could introduce methods or rules that post-process the individuals by removing the subtrees that have no impact, for example, by statistical pruning [47] or Bayesian selection [48]. We could also extend our fitness function with a penalty term for tree size.

3.2.4 Symbol Set

This section shows the results of our GP algorithm as we shrink and expand the symbol set. This time, the tournament size is reduced to 3, which also requires softening the mutation rate to $p_m = 0.2$. This allows us to account for larger symbol sets where it is easy to stuck at the local optimum. Low tournament size preserves diversity and lower mutation probability does not break the components as aggressively.

Figure 18 indicates that more generations are required as the symbol set expands. This is reasonable because the search space expands significantly even with a single added *non-terminal* since many different combinations could be acquired by incorporating it into the tree.

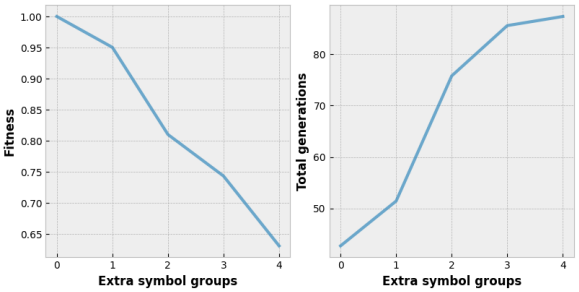


Figure 18: The average fitness score and the total number of generations required to the solution for different numbers of extra symbol groups. One extra symbol group includes one *non-terminal* and two *terminals*. Experiments were averaged over 30 runs with a maximum of 150 generations per run.

Figure 19 just shows that the trees are also harder to understand when more symbols are included. In fact, more meaningless branches are introduced with more symbols, since the chance to select the smallest optimal combinations of *terminals* and *non-terminals* decreases. For example, by introducing `sub`, we make it possible for our solutions

to contain, for instance, an expression $1-0$, which would have no effect.

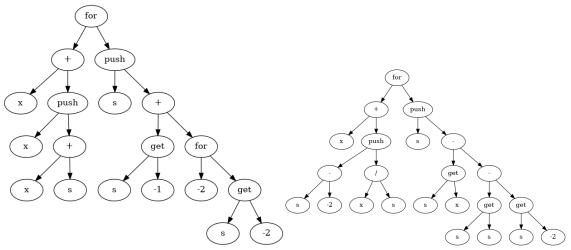


Figure 19: Identified solutions for **Fibonacci** series with a maximum allowed depth of 5. Left: a solution with only the necessary symbols. Nested `for` loops are not executed and pushing to a scalar returns a result of 0. Right: a solution with 4 extra *non-terminals* and 8 extra *terminals*. Notice, for example, how two subtraction operations are introduced instead of a single addition.

3.3 Question C

3.3.1 Expressiveness

Based on the results of the previous question, we could already determine one aspect of generality: expressive power. Our GP can be easily equipped with the same semantic components found in programming languages: conditional statements, for loops, variable initialization, sub-functions, etc. Therefore, at the very least our representations of solutions as trees with these programming components can express any kind of sequence rule that could be defined via coding statements. Being able to express any sequence rule[8] is crucial, but whether or not our algorithm can find it or is at least equipped with appropriate symbols and valid restrictions, is a further discussion of generality below.

3.3.2 Rule Generality

Considering **Fibonacci** series, our genetic program algorithm was shown only sequences of up to 20 numbers. If we want to check whether the discovered sequence generation rule is general, we need to test it on further numbers. Table 1 shows the sequences $\hat{\mathbf{y}}$ our discovered rule was able to generate for unseen x.

Indeed, the sequences are correct, which is easy to determine because the rule is generally correct.

[8] Of course, if the series comes, for example, from real-world data, such as time series, then there would be no way to express a real-world component in our tree that would exactly mimic the non-hand-crafted behavior.

$x-2$	$\hat{\mathbf{y}}$	$\text{Fit}(\mathbf{y}, \hat{\mathbf{y}})$
18	$\begin{bmatrix}0 & \cdots & 6765\end{bmatrix}^\top$	1
28	$\begin{bmatrix}0 & \cdots & 832040\end{bmatrix}^\top$	1
38	$\begin{bmatrix}0 & \cdots & 102334155\end{bmatrix}^\top$	1
48	$\begin{bmatrix}0 & \cdots & 12586269025\end{bmatrix}^\top$	1

Table 1: Results of applying the generated **Fibonacci** GP rule shown in Figure 17 (left) to unseen data, i.e., sequences of length ≥ 20.

For $x > 100$, it may be difficult to test numerically (due to numeric overflow) but we can still confirm that the derived rule follows the semantics of Equation 12 and the general programming behavior of list generation.

Perhaps an issue would arise if we allowed excessively large tree depth and added further *non-terminals*, such as `if-else`, in which case, our GP algorithm might just check the index and add a corresponding *terminal* instead of trying to find an underlying arithmetic rule; in fact, recombination would even encourage that. For this reason, it is necessary to evaluate the constructed rules over a wide range of samples, especially for sequences with complex patterns. As an example, Figure 20 illustrates what rule our algorithm generated with only a single training sample. It clearly is only specific for that particular input.

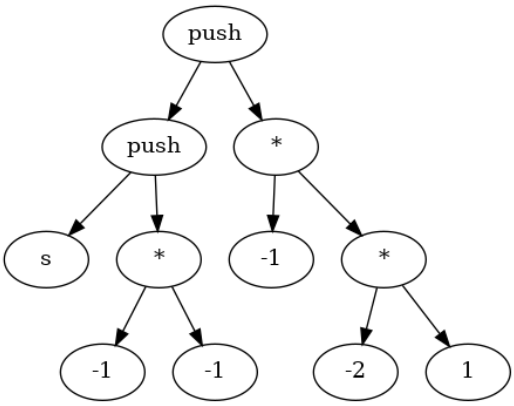


Figure 20: A generated rule for **Fibonacci** that yields a fitness of 1 for a single training sample with an input $x = 4$ and a starting sequence $\mathbf{s} = \begin{bmatrix}0 & 1\end{bmatrix}^\top$, i.e., we need to generate 2 more numbers. The rule simply pushes an additional 1 and then 2, which results in a correct sequence.

To summarize, we want smaller allowed maximum depth and more training samples to make our algorithm discover more general rules. For more complex sequences or other problems, such as non-discrete ones like regression, these issues can lead to significant concerns, including overfitting and code growth [49, 50].

3.3.3 Algorithm Generality

Another aspect of generality is the capability of the GP algorithm to adapt to different tasks. For this reason three further sequences were tested: **Tribonacci**, **Pell**, and custom **Arithmetic Geometric** (see subsection C.4 for more details), all of which are similar in terms of complexity to **Fibonacci**. Figure 21 shows 2 example rules generated for **Tribonacci** and **Pell** sequences.

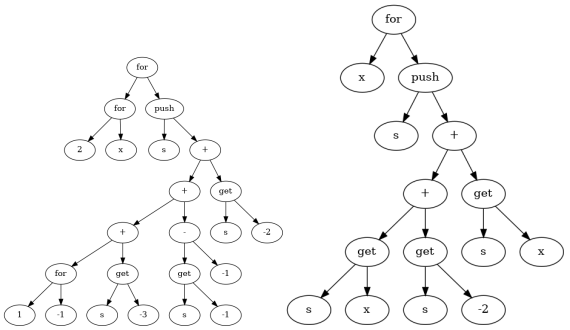


Figure 21: Identified solutions for **Tribonacci** and **Pell** series with a maximum allowed depth of 6. Nested `for` loops are not executed. Left: a solution for **Tribonacci** sequence. Notice some arithmetic redundancy: *terminal* 1 is added and subtracted. Right: a solution for **Pell** sequence. Notice how instead of multiplying by 2, the same value is added twice: this is actually easier to find due to recombination.

These graphs show that, indeed, our algorithm was able to find the correct rules, and is, therefore, general in this search aspect. Still, additional *terminals* and *non-terminals* were added before running the algorithm, for example, -3 and $\div$. This at least suggests that, even if our algorithm is capable of finding rules for other kinds of sequences, it always depends on a particular symbol set. In this aspect, we cannot say that our GP algorithm is general as without a sufficient set of symbols it could only approximate the solution.

Figure 22 shows the number of generations required, on average, to arrive at correct solutions for different sequence types. The more complex the rule is, the deeper the trees are and the longer the search is. For example, **Tribonacci** rule potentially takes longer find because the algorithm may stuck more easily at a local optimum, e.g., $\texttt{get}(s, -1) + \texttt{get}(s, -3) \times 2$, where even a semantically more correct rule (e.g., without $\times 2$) would yield worse fitness. This suggests that for even more complex problems, our algorithm may not even be able to escape from a sub-optimum in a reasonable amount of time.

To summarize, our GP algorithm can be general if we have background knowledge about the problem, i.e., if we can utilize the full expressive power required for a specific sequence rule or a set of rules through algorithm configuration which includes defining a sufficient set of symbols and sufficient maximum allowed depth. Generally, the more non-terminals we include, the more general our algorithm will be but background knowledge is always preferred to avoid redundancies. Finally, the algorithm configuration should adapt to problem complexity: we may want to increase the population size, the number of maximum generations, adjust the tournament size, etc to keep the algorithm general in terms of its ability to perform meaningful evolutions until the solution is found.

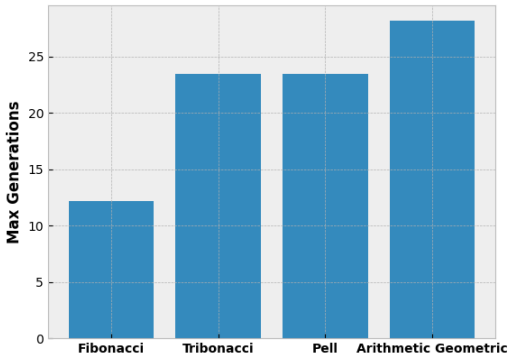


Figure 22: The number of generations, on average, required to reach the correct GP tree for 4 different sequence rules. In this case, the population size was increased to $N = 5 \times 10^4$ and tournament size to $T = 50$ due to more added *terminals* and *non-terminals*. The results were averaged only across 5 runs due to high computation costs.

References

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4. IEEE, 1995, pp. 1942–1948.

[2] K. Sutner, "Problem 88-8," *Math. Intelligencer*, vol. 10, no. 3, 1988.

[3] A. H. Elsheikh and M. Abd Elaziz, "Review on applications of particle swarm optimization in solar energy systems," *International Journal of Environmental Science and Technology*, vol. 16, no. 2, pp. 1159–1170, 2019. [Online]. Available: https://doi.org/10.1007/s13762-018-1970-x

[4] L. A. Rastrigin, "Systems of extremal control," *Nauka*, 1974.

[5] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[6] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, pp. 341–359, 1997.

[7] X.-S. Yang and S. Deb, "Cuckoo search via lévy flights," in *2009 World congress on nature & biologically inspired computing (NaBIC)*. Ieee, 2009, pp. 210–214.

[8] X.-S. Yang, "A new metaheuristic bat-inspired algorithm," in *Nature inspired cooperative strategies for optimization (NICSO 2010)*. Springer, 2010, pp. 65–74.

[9] "Sphere function," common benchmark function in optimization literature.

[10] A. O. Griewank, "Generalized descent for global optimization," *Journal of optimization theory and applications*, vol. 34, pp. 11–39, 1981.

[11] D. Ackley, *A connectionist machine for genetic hillclimbing*. Springer science & business media, 2012, vol. 28.

[12] H. Rosenbrock, "An automatic method for finding the greatest or least value of a function," *The computer journal*, vol. 3, no. 3, pp. 175–184, 1960.

[13] H.-P. Schwefel, *Numerical optimization of computer models*. John Wiley & Sons, Inc., 1981.

[14] H. Hector, "Sumplete is a wordle rival made entirely by chatgpt - here's how to play," *TechRadar*, 2023. [Online]. Available: https://www.techradar.com/news/sumplete-is-a-wordle-rival-made-entirely-by-chatgpt-heres-how-to-play

[15] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia tools and applications*, vol. 80, pp. 8091–8126, 2021.

[16] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[17] Y. Fang and J. Li, "A review of tournament selection in genetic programming," in *International symposium on intelligence computation and applications*. Springer, 2010, pp. 181–192.

[18] A. Shukla, H. M. Pandey, and D. Mehrotra, "Comparative review of selection techniques in genetic algorithm," in *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*. IEEE, 2015, pp. 515–519.

[19] R. R. Brooks, S. S. Iyengar, and J. Chen, "Automatic correlation and calibration of noisy sensor readings using elite genetic algorithms," *Artificial intelligence*, vol. 84, no. 1-2, pp. 339–354, 1996.

[20] C. W. Ahn and R. S. Ramakrishna, "Elitism-based compact genetic algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 4, pp. 367–385, 2003.

[21] G. Singh and N. Gupta, "A study of crossover operators in genetic algorithms," *Frontiers in Nature-Inspired Industrial Optimization*, pp. 17–32, 2022.

[22] B. C. Wallet, D. J. Marchette, and J. L. Solka, "Matrix representation for genetic algorithms," in *Automatic Object Recognition VI*, vol. 2756. SPIE, 1996, pp. 206–214.

[23] K. A. De Jong and W. M. Spears, "A formal analysis of the role of multi-point crossover in genetic algorithms," *Annals of mathematics and Artificial intelligence*, vol. 5, pp. 1–26, 1992.

[24] S. M. Lim, A. B. M. Sultan, M. N. Sulaiman, A. Mustapha, and K. Y. Leong,

"Crossover and mutation operators of genetic algorithms," *International journal of machine learning and computing*, vol. 7, no. 1, pp. 9–12, 2017.

[25] O. Abdoun, J. Abouchabaka, and C. Tajani, "Analyzing the performance of mutation operators to solve the travelling salesman problem," *arXiv preprint arXiv:1203.3099*, 2012.

[26] U. Khair, Y. D. Lestari, A. Perdana, D. Hidayat, and A. Budiman, "Genetic algorithm modification analysis of mutation operators in max one problem," in *2018 Third International Conference on Informatics and Computing (ICIC)*. IEEE, 2018, pp. 1–6.

[27] F. Chicano, A. M. Sutton, L. D. Whitley, and E. Alba, "Fitness probability distribution of bit-flip mutation," *Evolutionary computation*, vol. 23, no. 2, pp. 217–248, 2015.

[28] J. Peng and R. J. Williams, "Efficient learning and planning within the dyna framework," *Adaptive behavior*, vol. 1, no. 4, pp. 437–454, 1993.

[29] I. Rejer and J. Jankowski, "fgaam: A fast and resizable genetic algorithm with aggressive mutation for feature selection," *Pattern Analysis and Applications*, pp. 1–17, 2022.

[30] C. Bielza, J. A. Fernández del Pozo, and P. Larranaga, "Parameter control of genetic algorithms by learning and simulation of bayesian networks—a case study for the optimal ordering of tables," *Journal of Computer Science and Technology*, vol. 28, no. 4, pp. 720–731, 2013.

[31] A. Hassanat, K. Almohammadi, E. Alkafaween, E. Abunawas, A. Hammouri, and V. S. Prasath, "Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach," *Information*, vol. 10, no. 12, p. 390, 2019.

[32] L. Sigler, *Fibonacci's Liber Abaci: a translation into modern English of Leonardo Pisano's book of calculation*. Springer Science & Business Media, 2003.

[33] L. Vanneschi and R. Poli, "Genetic programming-introduction, applications, theory and open issues." 2012.

[34] J. F. Miller and S. L. Harding, "Cartesian genetic programming," in *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*, 2008, pp. 2701–2726.

[35] M. Brameier, W. Banzhaf, and W. Banzhaf, *Linear genetic programming*. Springer, 2007, vol. 1.

[36] T. Perkis, "Stack-based genetic programming," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. IEEE, 1994, pp. 148–153.

[37] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, 2001.

[38] S. L. Harding, J. F. Miller, and W. Banzhaf, "Self-modifying cartesian genetic programming," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1021–1028.

[39] S. Harding, J. F. Miller, and W. Banzhaf, "Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing," in *Genetic Programming: 12th European Conference, EuroGP 2009 Tübingen, Germany, April 15-17, 2009 Proceedings 12*. Springer, 2009, pp. 133–144.

[40] S. Dignum and R. Poli, "Sub-tree swapping crossover, allele diffusion and gp convergence," in *Parallel Problem Solving from Nature–PPSN X: 10th International Conference, Dortmund, Germany, September 13-17, 2008. Proceedings 10*. Springer, 2008, pp. 368–377.

[41] S. Dignum, "An analysis of genetic programming sub-tree swapping crossover with applications." Ph.D. dissertation, University of Essex, Colchester, UK, 2008.

[42] J. Page, R. Poli, and W. B. Langdon, "Mutation in genetic programming: a preliminary study," in *Genetic Programming: Second European Workshop, EuroGP'99 Göteborg, Sweden, May 26–27, 1999 Proceedings 2*. Springer, 1999, pp. 39–48.

[43] L. Beadle and C. G. Johnson, "Semantically driven mutation in genetic programming," in *2009 IEEE Congress on Evolutionary Computation*. IEEE, 2009, pp. 1336–1342.

[44] H. Zhang, Q. Chen, B. Xue, W. Banzhaf, and M. Zhang, "A semantic-based hoist mutation operator for evolutionary feature construction in regression," *IEEE Transactions on Evolutionary Computation*, pp. 1–1, 2023.

[45] T. O. Hodson, T. M. Over, and S. S. Foks, "Mean squared error, deconstructed," *Journal of Advances in Modeling Earth Systems*, vol. 13, no. 12, p. e2021MS002681, 2021.

[46] N. Javed, F. Gobet, and P. Lane, "Simplification of genetic programs: a literature survey," *Data Mining and Knowledge Discovery*, vol. 36, no. 4, pp. 1279–1300, 2022.

[47] P. Rockett, "Pruning of genetic programming trees using permutation tests," *Evolutionary Intelligence*, vol. 13, no. 4, pp. 649–661, 2020.

[48] G. F. Bomarito, P. E. Leser, N. Strauss, K. M. Garbrecht, and J. D. Hochhalter, "Bayesian model selection for reducing bloat and overfitting in genetic programming for symbolic regression," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2022, pp. 526–529.

[49] G. Paris, D. Robilliard, and C. Fonlupt, "Exploring overfitting in genetic programming," in *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 2003, pp. 267–277.

[50] M. Bhattacharya and B. Nath, "Genetic programming: A review of some concerns," in *International Conference on Computational Science*. Springer, 2001, pp. 1031–1040.

[51] S. Ruangwises, "Sumplete is hard, even with two different numbers," *arXiv preprint arXiv:2309.07161*, 2023.

A Appendix A

A.1 Algorithm Types

In subsection 1.1 few other swarm metaheuristic algorithms were checked for optimal population size. All of them have similar characteristics with *Differential Evolution* being a bit unique in having a specific bounded range for optimal N, which could be explained by the fact that it is the only algorithm that doesn't use `global_best` as part of cooperation among particles. The sections below explain the dynamics of each algorithm that was experimented with and also emphasize how individuals communicate with each other.

A.1.1 Particle Swarm Optimization

Algorithm. *PSO* starts with a group of random solutions and then searches for optima by updating generations. Each particle's movement is influenced by its local best-known position and is also guided toward the global best-known position in the search space, which is updated as better positions are found by other particles.

Cooperation. Particles communicate by sharing the global best position. This allows them to explore more promising regions of the search space, leading to faster convergence towards the optimal solution.

A.1.2 Differential Evolution

Algorithm. *DE* uses differential mutation, where the difference between two randomly selected population vectors is used to create the donor vector for recombination. The recombination and selection steps create a new population vector which replaces an old one if it provides a better solution.

Cooperation. Individuals cooperate by sharing information about their current positions and fitness values. This information is used to guide the mutation and recombination steps, allowing the population as a whole to move towards better solutions.

A.1.3 Cuckoo Search

Algorithm. *CS* is a nature-inspired algorithm based on the behavior of cuckoo birds. Each egg in a nest represents a solution, and a cuckoo egg represents a new solution. The aim is to use the new and potentially better solutions (cuckoos) to replace not-so-good solutions in the nests.

Cooperation. Cuckoos cooperate by sharing information about their nests (solutions). If a cuckoo finds a better nest than its current one, it will abandon its current nest and move to the better one. This way, all cuckoos gradually converge towards the best solutions.

A.1.4 Bat Algorithm

Algorithm. *BA* is also a biology-inspired algorithm based on the echolocation behavior of microbats. Each virtual bat flies randomly with a velocity toward prey (global best) with varying frequency and loudness. As it searches and finds its prey, it changes frequency, loudness, and pulse emission rate.

Cooperation. Bats share information about their best-known positions with each other not just by simply communicating the positions, but through a more complex interaction (loudness and pulse rate, both of which are algorithm parameters), which makes it unique compared to other swarm algorithms.

A.2 Objective Types

In subsubsection 1.1.2 multiple objectives were introduced and their surfaces were plotted in Figure 3 for $(d_1 = 100, d_2 = 100)$. They are all similar in a way that their complexity depends on the number of dimensions D, however, each is different by the surface irregularity. Further subsections describe the mathematical properties of these functions (except for *Rastrigin* which was described in subsection 1.1).

Please note that all the objectives in this section are described in their raw form, meaning they require to be minimized, however, all the experiments that were done using them negated the result to turn the task into maximization. Also, the chosen bounds are most commonly found in the literature.

A.2.1 Sphere Function

This is typically considered the easiest to optimize. It's a convex function and doesn't have any local minima other than the global minimum. Metaheuristic algorithms can solve it efficiently because, on each iteration, all solutions are guaranteed to be better, as there is only one optimum to adjust to.

$$f(\mathbf{x}) = \sum_{d=1}^{D-1} \left(100(x_{d+1} - x_d^2)^2 + (1 - x_d)^2\right) \quad (15)$$

where $x_d \in [-5.12, 5.12]$, for $d = 1, ..., D$

A.2.2 Griewank Function

The *Griewank* function, although topologically looks similar to the sphere function, has many local minima at small scales. The locations of these minima are sinusoidally modulated, which can make it difficult for optimization algorithms to navigate toward the global minimum or maximum if negated.

$$f(\mathbf{x}) = \frac{1}{4000} \sum_{d=1}^{D} x_d^2 - \prod_{d=1}^{D} \cos\left(\frac{x_d}{\sqrt{d}}\right) + 1 \quad (16)$$

where $x_d \in [-600, 600]$, for $d = 1, ..., D$

A.2.3 Ackley Function

The $2D$ form of this function can be characterized by a flat outer region and a hole in the center. Although many of the local optimum points in this function make it more challenging, many of them are better than others as the spike/hole gets deeper, meaning improvements over local optimum points can guide the metaheuristic toward the global optimum.

$$f(\mathbf{x}) = -20 \exp\left(-0.2\sqrt{\frac{1}{D}\sum_{d=1}^{D} x_d^2}\right) - \exp\left(\frac{1}{D}\sum_{d=1}^{D} \cos(2\pi x_d)\right) + 20 + e \quad (17)$$

where $x_d \in [-32.768, 32.768]$, for $d = 1, ..., D$

A.2.4 Rosenbrock Function

This function is more difficult than *Sphere* and *Griewank* functions because it's not separable, meaning the dimensions are interdependent. It has a narrow, curved valley that contains the global optimum, and finding this valley can be challenging for an optimization algorithm, especially for high-dimensional problems.

$$f(\mathbf{x}) = \sum_{d=1}^{D-1} \left(100(x_{d+1} - x_d^2)^2 + (1 - x_d)^2\right) \quad (18)$$

where $x_d \in [-2.048, 2.048]$, for $d = 1, ..., D$

A.2.5 Schwefel Function

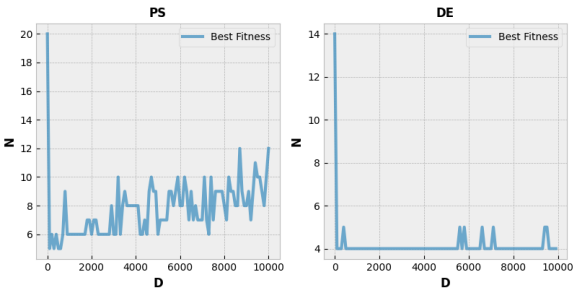
The Schwefel function is one of the most difficult benchmark functions to optimize. It has a large number of local optima that are irregularly distributed throughout the search space and far from the global minimum/maximum.

$$f(\mathbf{x}) = 418.9829D - \sum_{d=1}^{D} x_d \sin\left(\sqrt{|x_d|}\right) \quad (19)$$

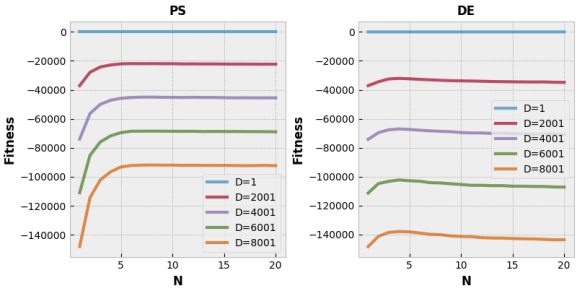
where $x_d \in [-500, 500]$, for $d = 1, ..., D$

A.3 Additional Plots

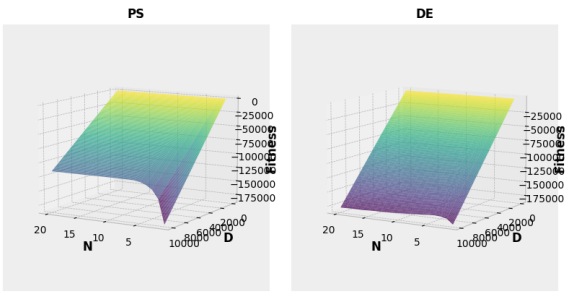
A.3.1 Higher Dimensions



(a) Population size against the number of dimensions



(b) Fitness against population size for different D



(c) Fitness against population size and dimensions

Figure 23: Optimal population size dependency on the number of dimensions for the negated *Rastrigin* function

Figure 23 shows how the optimal population size depends on the dimension size for the negated *Rastrigin* function, in the same way as Figure 2, just

with much higher dimensions. Note that the number of evaluations this time was chosen 500 instead of 5000 and the total number of runs for each (N, D) pair was 10 to acquire the results within a reasonable time.

For such a large D the space becomes unexplorable and thus relying on randomness yields better results than having to explore more. Still, very large population size $N > 20$ is not preferred, which is in alignment with our analysis in subsubsection 1.1.2, just that relative to the scale of D, we may want to put more trust into random initialization since there is a higher chance for more particles to be initialized near good solutions and this probability, although is always the same, is better and better to rely on for very large D. The same cannot be said, however for DE which does not keep the memory of the global best, and thus the attraction point cannot be determined that well, i.e., the algorithm cannot utilize the advantages of more space being covered since the updates are not based on any global attraction.

A.3.2 Metaheuristic Dynamics

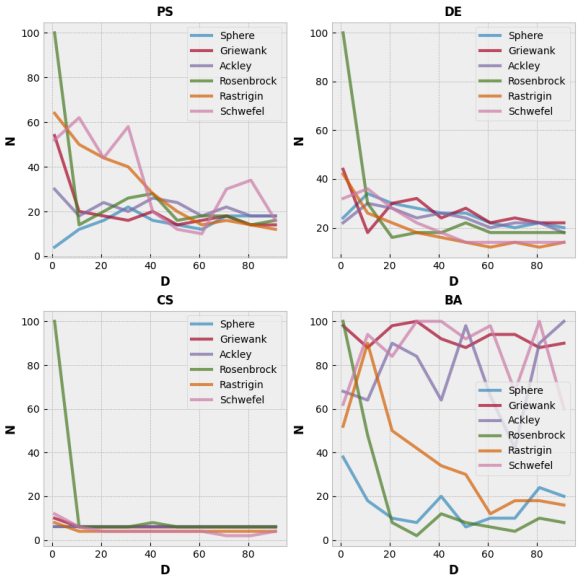


Figure 24: Optimal population size for different D across common negated objective functions for different algorithms.

Figure 24 just shows the optimal population size as the dimensionality changes for every objective function and every swarm algorithm implemented. A total of 15 runs were averaged for each (N, D) pair. It is simply here to reinforce the analysis of subsubsection 1.1.2, where it was stated that regardless of the objective or the algorithm, as the dimensionality increases, the optimal population size decreases. It is only to illustrate the general trend and the specific differences would have to be analyzed based on the algorithm behavior and the function type. Note that *Bat Algorithm* may look unique, however, if we cover a larger scale of N, the pattern is still the same (Figure 25).

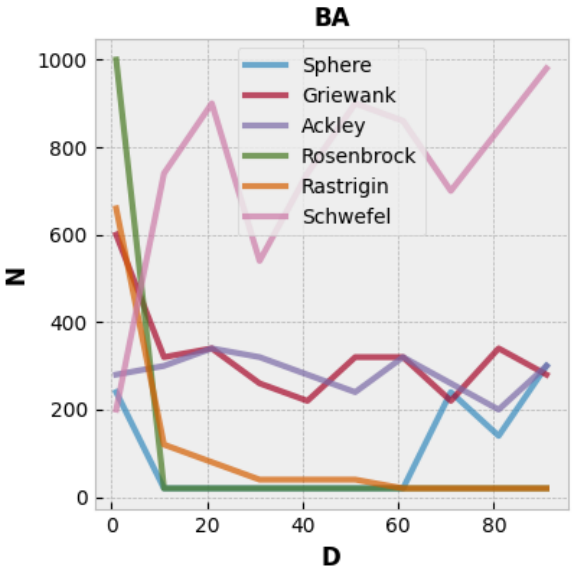


Figure 25: Optimal population size for different D across common negated objective functions for BA with specifically large population size range.

B Appendix B

B.1 Game Difficulty

Sumplete problem is NP-hard [51], however, as it scales, it only becomes unsolvable when the set of possible values for each game cell is fixed. For example, it would be quite easy to solve the game even when k is large if the upper value bound for each cell was k^2. In other words, almost always the sum for each row or column would require adding up unique values, and finding a correct deletion criteria for at least one row or column would almost guarantee that it is the true one, allowing one to easily solve for the next rows and columns, without backtracking.

Put another way, a game with $G_{ij} \in [1, k^2]$ encodes a lot of entropy and a clever algorithm would easily utilize it. Figure 26 shows how a simple GA with a fitness function that counts the number of correct rows and columns achieves a high fitness score, potentially solving the game completely in some runs, for large k, given the upper value bound is k^2.

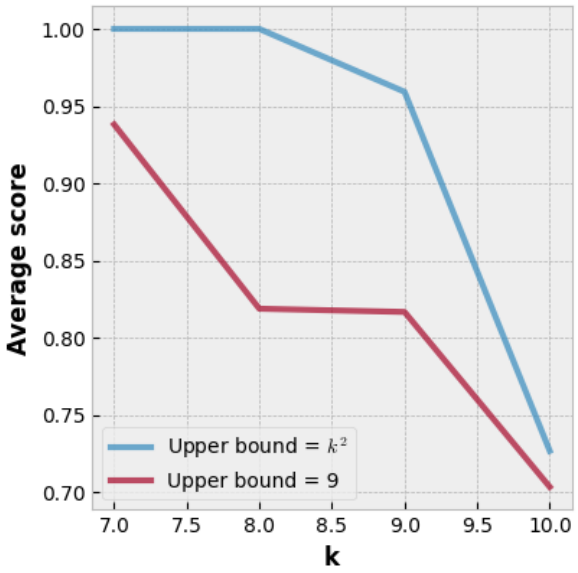


Figure 26: Average best fitness score after running a maximum of 200 generations with a population size of 1000 for $k \in \{7, 8, 9, 10\}$ with different upper value bounds for each game cell.

B.2 Default Parameters

Major parameters, like mutation rate or crossover probability, were experimented with before the optimal choices were made. Others like `decay_factor`, `max_mean_counts` etc., were not experimented with but rather chosen throughout the development of the full algorithm and running experiments with different configurations on the spot.

B.2.1 Selection

- `elite_frac`: The fraction of the top individuals to select from the population as elites (they skip crossover and mutation). Defaults to 0. Never used in our experiments because for large k we always prefer novel solutions.
- `tournament_size`: The number of individuals sampled from the population that compete against each other for transiting to the next generation. Defaults to 2, which may not be the best for lower values of k but is crucial for large k where many local optimum points exist.

B.2.2 Crossover

- `p_c`: The crossover rate, i.e., the fraction of individuals taken for crossover. Defaults to 0.7.
- `num_cross_points`: The number of points to use for crossover when crossover strategy is **n-point** ($n + 1$ segments will be created and every second one will be swapped within each parent pair to create 2 children). Defaults to floor($\frac{k}{2}$).
- `frac_cross_rows_cols`: The fraction of rows/columns to be swapped between 2 parents when crossover strategy is **row-col swap**. Defaults to 0.5.

B.2.3 Mutation

- `p_m`: The mutation rate, i.e., the fraction of individuals taken for mutation. Defaults to 0.2.
- `p_flip`: The probability of each gene being flipped when mutation strategy is **bit flip**. Defaults to 0.1.
- `frac_mutate_rows_cols`: The fraction of rows/columns to be shuffled when mutation strategy is **row-col shuffle**. Defaults to 0.2.

B.2.4 Refit and escape

- `decay_factor`: The factor used in the decay formula (Equation 2) where the local optimum fitness is decayed. Defaults to 0.99.
- `max_mean_hits`: The maximum number of consecutive generations with the same mean genome. If further generations have the same mean, the algorithm is partially restarted. Defaults to 50.
- `sim_factor`: The similarity factor, i.e., if an individual has this fraction of genes matching another individual, it would be considered similar. Similar genomes are also refreshed if the algorithm gets stuck. It is also possible to penalize them if they are similar to local best solutions. Defaults to 0.8.

B.2.5 Algorithmic

- `N`: The population size. Defaults to 1000.
- `D`: The genome dimensionality. Defaults to k^2.
- `fitness_fn`: The evaluation function to use. Defaults to **row-col** which is fast and yields great results.
- `max_generations`: The maximum number of generations. Defaults to 1000.

For more parameters, please check the attached code.

B.3 Normalization

B.3.1 Normalization Rule

As discussed, there is an exponential dependency of total fitness evaluations on k but this trend does not really interest us because it is a rather obvious outcome. For this reason, we normalize the total number of evaluations across all values of k: we take the log and subtract the slope:

$$\bar{y} = \log y - \texttt{log\_slope} \times x \tag{20}$$

where y represents the total number of fitness evaluations at some particular x that represents the index of k. Log scale converts the exponential dependency to linear but we still want to subtract the slope, i.e., we want to detrend the dependency of evaluations in general on k and only keep the trends of evaluations specific to some parameter configurations. `log_slope` is just computed by finding the tangent of a curve of interest - we can do this due to log-linear dependency.

Ultimately, it is worth noting that game complexity is all that matters - even at large ranges of parameters, the performance is similar at the scale of k - this indicates that *Sumplete* is a very hard problem, suggesting that at larger values of k, pretty much any algorithm would perform the same as random search.

B.3.2 Additional Plots

Below are just the same plots shown in subsection 2.4, just at the original log scales. This is only included for completeness and for optional comparisons.

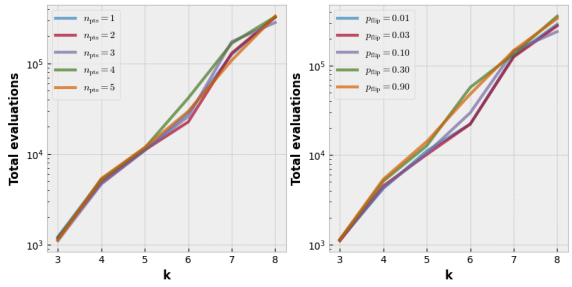


Figure 27: The unnormalized log scale version of Figure 11.

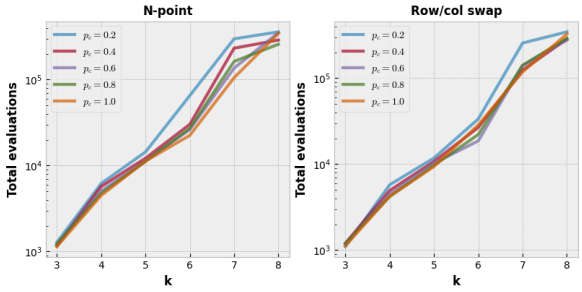


Figure 28: The unnormalized log scale version of Figure 12.

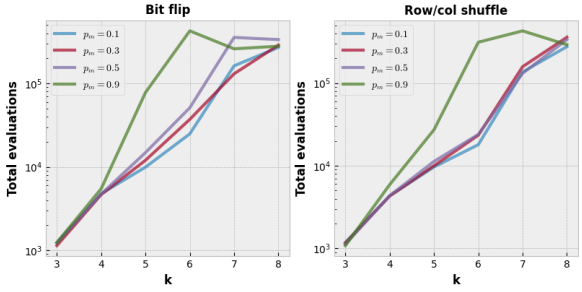


Figure 29: The unnormalized log scale version of Figure 13.

C Appendix C

C.1 Non-terminals

Many different kinds of mostly binary operators were considered, because some sequences may require conditional checks, access to arbitrary elements of the generated sequence so far, sub-sequence generations, and any other complex patterns. Additionally, it's rather easy to extend this list further *non-terminals*, for instance, we could add cos and sin in arithmetic expressions, or we could create a separate class, like converters that convert values from one measurement system to another, or one that provides custom functions, such as checking if the number is prime.

Please note that in the experiments that were performed, only a limited set of *non-terminals* were used to save some time in computation while still being able to prove the concept of a working algorithm. Those that were used are indicated specifically. Additionally, note that certain operators like arithmetic and comparison are defined only for scalars, and, although it is possible to define element-wise operations for lists, for simplicity this is avoided.

C.1.1 Arithmetic

- `add`: adds two numbers (selected)
- `sub`: subtracts one number from the other (selected)
- `div`: divides one number by the other (division by zero gives the maximum number allowed by the system) (selected)
- `mul`: multiplies one number by the other (selected)
- `mod`: checks the modulo of one value to the other (modulo of zero is redefined to give zero)
- `pow`: raises one number to the power of the other (power is restricted to the maximum number of 100)

C.1.2 Comparison

- `eq`: checks if the two numbers match (actually, this and `lt` or `gt` are the only comparison operators we need, others could be avoided if we swap the inputs, or negate the condition, or use disjunction)
- `lt`: checks if one number is lower than the other
- `le`: checks if one number is lower or equal to the other
- `gt`: checks if one number is greater than the other
- `ge`: checks if one number is greater or equal to the other

C.1.3 Logic

- `not`: negates the condition (note: unary)
- `and`: logical conjunction
- `or`: logical disjunction
- `all`: same as $\forall x$ where x represents an element in a list (note: unary)
- `any`: same as $\exists x$ where x represents an element in a list (note: unary)

C.1.4 Indexable

- `get`: gets an element from the specified list at the specified index (selected)
- `set`: sets a specified value at the specified list at the specified index and returns the list (note: ternary)
- `push`: adds a specified value as the last element to the specified list and returns the full list (selected)
- `pop`: removes the last element from the list and returns that removed element (note: unary)
- `insert`: inserts a specified value at the specified position in the specified list and returns it (more flexible than `push`) (note: ternary)
- `insert`: removes a value from the specified list at the specified position and returns that value (more flexible than `pop`)

C.1.5 Flow

- `before-after`: takes two expressions, e.g., two subtrees, executes both in that order but returns the result of only the second subtree. This is useful, for instance, if we wanted to model full sequence generation without specifying the start of the sequence: the *before* subtree could encode list initialization and the initial value additions to it, depending on the sequence the GP was trained for. This is avoided for simplicity.
- `if`: takes a conditional value and two expressions - one to execute if the value evaluates to true and the other to execute if the value evaluates to false (note: ternary).
- `for`: the simple variant takes the number of times to repeat the execution of the given expression and returns the final result of that expression (selected). A more complex case could include an expression that updates the step value and modifies the state of the algorithm by introducing temporary step variable in the expression subtree. The maximum number of loops is restricted. (note: can be ternary)
- `while`: takes a conditional expression and any other arbitrary expression and returns the final result of the arbitrary expression executed multiple times until the conditional expression evaluates to false. The number of maximum loops is restricted.

C.2 GP Mutation

The best results were acquired by mixing different types of mutation - five of which were considered. Note that when a random node is selected, it is never the root node.

- **Grow**. Selects a random node in the tree and generates a new subtree either up or down from it with random height, such that the resulting tree does not exceed the maximum allowed depth. Technically, this could result in smaller trees than the original ones but this is fine because the trees are quite often trimmed anyway since their branches tend to exceed the maximum allowed depth.
- **Shrink**. Selects a random node and replaces it with a random *terminal*.
- **Skip**. Selects a random *non-terminal* node and replaces its parent with that node.
- **Hoist**. Selects a random *non-terminal* and makes it the new parent node.
- **Renew**. Selects a random node and, depending on its type, replaces it with a new *terminal* or *non-terminal*. If it is *non-terminal*, the children are kept the same, although some children may be dropped or new ones could be generated if there is a change in arity.
- **Regen**. Generates a completely new tree to replace the original one. In combination with tournament selection for a low tournament size, this gives more chances for not-so-highly fit individuals to survive, e.g., those that have the potential to exceed even the best programs through more evolution.

C.3 GP Tree Graph

An example of a graphical representation of an ideal solution to the **Fibonacci** series is shown in Figure 30.

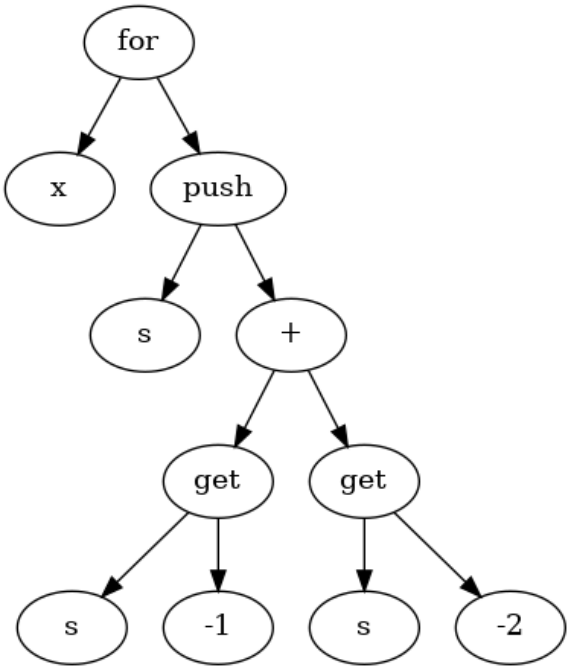


Figure 30: The ideal rule for generating **Fibonacci** full series given the remaining x elements in a sequence and the starting sequence $s = [0, 1]$.

There are two variables as *terminal*: x which is an integer representing how many entries in a sequence need to be generated (excluding the starting values) and s which is a list of the starting

sequence, in **Fibonacci** case, it is always $[0, 1]$. There are also two integer constants as *terminals*: -1 and -2, both used to access the last element and second to last elements of the list variable s.

Non-terminals include `for` which takes x as the first child and executes the second child for that many times as encoded by x. It returns the final result of the second child. `get` takes a list s and an index, which, like in *Python*, can be negative to indicate to start indexing from the other side. It returns the value at that bounded index or 0 if the list is empty. `push` takes as well a list s and a value to append to the end of that list and returns the full list. *Non-terminal* + simply adds the two numbers.

C.4 Additional Sequences

In addition to **Fibonacci**, three additional sequences were implemented, with a complexity similar to that of **Fibonacci**. Their arithmetic rules and the starting sequence values are defined as follows:

Tribonacci. This function sums the last three numbers of the sequence, instead of two, to determine the next one.

$$y_x = y_{x-1} + y_{x-2} + y_{x-3} \tag{21}$$

$$\mathbf{s} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}^\top \tag{22}$$

Pell. This function is the same as **Fibonacci**, except it multiplies the last number in the current sequence by 2 before adding it.

$$y_x = 2 \times y_{x-1} + y_{x-2} \tag{23}$$

$$\mathbf{s} = \begin{bmatrix} 0 & 1 \end{bmatrix}^\top \tag{24}$$

Arithmetic-Geometric. This rule is similar to **Pell**, except it also modifies the second to last sequence number before adding it - it is multiplied by 0.5.

$$y_x = 2 \times y_{x-1} + 0.5 \times y_{x-2} \tag{25}$$

$$\mathbf{s} = \begin{bmatrix} 0 & 1 \end{bmatrix}^\top \tag{26}$$

More complex rules were not considered as our algorithm would need a lot more time to explore more complex patterns.