Genetic Algorithms and Genetic Programming on Comparison Sorting

Final Degree Dissertation
Degree in Mathematics

Etor Arza Gonzalez

Supervisor:
Francisco de la Hoz Mendez (Patxi)

Leioa, 1 May 2018
Contents

Introduction v

1 Theory background 1
   1.1 First definitions .................................................. 1
   1.2 Classic Genetic Algorithms ..................................... 3

2 GA on function maximization 8
   2.1 Description of the algorithm .................................... 8
   2.2 Algorithm’s performance ......................................... 10
       2.2.1 Uniparametric test function ................................. 10
       2.2.2 Inverted Rastrigin’s function ............................... 11
       2.2.3 Inverted Himmelblau’s function ............................ 12

3 GA on interpolation 13
   3.1 Description of the algorithm ..................................... 13

4 GA on sorting 16
   4.1 First attempt: Hollands Broadcast Language .................. 16
       4.1.1 Theory introduction .......................................... 16
       4.1.2 Our approach .................................................. 18
   4.2 Our second attempt: Linear Genetic Programming .............. 24
       4.2.1 Our Linear Programs Explained ............................. 24
       4.2.2 Results .......................................................... 26

5 Methodology and Coding Challenges 28
   5.1 Max Search ............................................................ 28
   5.2 Polynomial Interpolation ......................................... 29
   5.3 List Sorting .......................................................... 29

6 Future Work 31

Appendix A Implemented Code 32
   A.1 Matlab and Python Code for Max Search ......................... 33
       A.1.1 Main Functions and Classes ................................ 33
       A.1.2 Utility Functions .............................................. 50
       A.1.3 Python Code .................................................... 53
   A.2 Matlab and Python Code for Polynomial Interpolation ............ 55
       A.2.1 Main Functions and Classes ................................ 55
       A.2.2 Utility Functions .............................................. 60
       A.2.3 Python Code .................................................... 61
   A.3 Code of List Sorting ................................................ 61
       A.3.1 1st original approach ....................................... 61
A.3.2  1st restricted approach, with r as new operator  
A.3.3  2nd approach, final version
Introduction

A Genetic Algorithm (GA) is guided random search algorithm, more precisely, an evolutionary algorithm [4]. The fundamental idea behind evolutionary algorithms is simple: a random population (sample of solution candidates, represented as a set of strings) is drawn uniformly at random, then, the performance (fitness) of these solutions is measured, and based on this information, we sample a new population, whose fitness is expected to improve the one of the previous population.

Each of this pairs \{measure and sample\} is called a generation. Specifically, on genetic algorithms, the sampling is performed in a special way: first we select the best individuals (each \{string , fitness\} tuple) of the population, then we combine them and finally, we mutate them (apply small changes to promote diversity). We repeat \{measure and sample\} until a desired solution is found, the number of maximum iterations is reached or other stopping criteria are met. Throughout the whole algorithm, the individual with the highest fitness value is saved. Each time an individual's fitness is calculated, we compare it with the found best fitness, and if it exceeds it, we save the corresponding individual. When the algorithm terminates, this best fitted individual's string is the found solution.

Unlike other search algorithms such as hill climbing, gradient descent and greedy algorithms, GAs are very good global search techniques, but they may miss local optima, and most of the times, they are only capable of finding suboptimal solutions. An advantage of GAs is that they do not need specific domain knowledge, thus making them very versatile. However, this is sometimes a disadvantage, since they can not exploit this domain specific knowledge, and therefore, when this information is available, other techniques that exploit it usually perform better than GAs (gradient descent, for example). To some extent, GAs can be combined with other search techniques, obtaining the so called hybrid algorithms, capable of exploiting local information but without loosing the global scoop.

The versatility of GAs, makes them a nice optimization technique when no other problem specific algorithms are available. This versatility, however, has its limits, and for very complex problems, there may be encoding problems. For the GA to be able to work, our problem’s solutions (search space) need to be represented as strings, which is not always possible. Moreover, being able to represent the solutions like a strings is not enough to ensure convergence on a reasonable time, since the quality of the encoding of these solutions also influences the success of the GA search, and the time it takes to converge. On a good encoding, for most of the individuals, a small change on the string has to make little change on the fitness, in other words, the fitness landscape across the solution space has to be as smooth as possible. In addition, if we intend to use classic genetic operators, the encoding needs to be as tight as possible, this means, related features should be placed one next to another. This is necessary because classic genetic operators are more likely to preserve parts of strings that are near each other, and so meaningful feature combinations should be close to one another if they are to make it to the next generation.
Regarding our project, our initial idea for this project was to create sorting algorithms using genetic algorithms, and on the way to our goal, we tried simpler examples to learn about the field, following Thomson’s Rule: 'It is faster to make a four-inch mirror than a six-inch mirror than to make a six-inch mirror.'

First we created a simple GA that tries to find the global maximum of an uniparametric or biparametric function on a given interval or rectangle. Then based on this example, we created a function that optimizes the interpolating points of a n degree polynomial. Finally, and after plenty trial and error, we created small sorting algorithms with the use of GA.

In this document, we briefly discuss the theory of Genetic Algorithms (GA) on chapter 1. We implemented a GA that tries to find the global maximum of an uniparametric or biparametric function, and tested its performance with different classical test functions. We show our work on the matter on chapter 2. Then, we show our implementation of a GA that tries to find the optimal interpolating points for a given function and interval on chapter 3. On chapter 4, we discuss our attempts at sorting lists using genetic algorithms and genetic programming. Finally, on chapter 5, we address some of the programming challenges we encountered while implementing the algorithms.
Chapter 1

Theory background

On this chapter we introduce the foundations of Genetic Algorithms, and some of the most important classical operators.

1.1 First definitions

These notions are based on [7] and [11].

Definition 1. (alphabet):

We say that a finite set of unique symbols $\Omega$ is an alphabet.

Definition 2. (string and character):

We say that $S = s_1s_2...s_n \in \Omega \times \Omega \times ...n \text{ times} \times \Omega$ is a string of length n over the alphabet $\Omega$. In this case, we say $\Omega^n$ is the set of strings of length n over the alphabet $\Omega$. The $s_i$ elements of a string are said to be characters.

Definition 3. (genetic operator):

Let $P$ and $Q$ be subsets of $\bigcup_{i=0}^{\infty} \Omega^i$. We say that a function $\alpha$ is a genetic operator, $\alpha : P \rightarrow P(Q)$, 

$S \rightarrow \alpha(S)$

where $P(Q)$ is the set of all probability distributions over $Q$.

Note: when $\alpha(P)$ is a degenerate probability distribution, we say the genetic operator to be deterministic. In this case, we may identify $\alpha(P)$ with a corresponding subset of $Q$.

Example 1. (identity operator):

Let $\Omega = \{0, 1\}$ be our alphabet, and $\Omega^n$ our set of strings. We define $\alpha$ as follows:

$\alpha : (\Omega^n) \rightarrow \Omega^n$

$S = s_1s_2...s_n \rightarrow \alpha(S) = S$

Example 2. (permutation):

Let $\Omega$ be an arbitrary alphabet, and $\Omega^n$ our set of strings. We define $\alpha$ as follows:
\[ \alpha : (\Omega^n) \overset{\alpha}{\longrightarrow} \Omega^n \]
\[ S = s_1s_2...s_n \overset{\alpha}{\longrightarrow} \alpha(S) = \sigma(S) \]

where \( \sigma \) is a permutation \( \in S_n \).

**Definition 4. (search space):**

Let \( \Omega \) be an alphabet.
A **search space** is a finite subset of \( \bigcup_{i=0}^{\infty} \Omega^i \)

When running a search algorithm, our goal is to find the optimal string from the search space given a fitness function.

Note: Sometimes, we may want our search space to be a subset of \( \Omega^n \).

**Definition 5. (deterministic fitness function):**

Let \( V \) be a search space. A fitness function \( \mu \) maps strings on \( V \) to positive values:
\[ \mu : V \overset{\mu}{\longrightarrow} \mathbb{R}^+ \bigcup \{0\} \]
\[ S \overset{\mu}{\longrightarrow} \mu(S) \]

Actually, this is a simplified definition of a more general fitness function. A more general definition would consider \( \mu \) to be a function that maps each individual to a random variable on \( \mathbb{R}^+ \bigcup \{0\} \).

**Definition 6. (stochastic fitness function):**

Let \( V \) be a search space. A fitness function \( \mu \) maps strings on \( V \) to a random variable over the positive numbers.
\[ \mu : V \overset{\mu}{\longrightarrow} \mathcal{P}(\mathbb{R}^+ \bigcup \{0\}) \]
\[ S \overset{\mu}{\longrightarrow} \mu(S) = X \]

This second point of view may be useful when, for example, the computation of the fitness function is costly, but it can be approximated by using Monte Carlo sampling. The fitness function, in this case, can also be viewed as a noisy function, as seen in [13] and [14].

**Example 3.** Let \( \Omega = \{0,1\} \) be our alphabet, and \( \Omega^n \) our search space.
We can define \( \mu \) as follows:
\[ \mu : V \overset{\mu}{\longrightarrow} \mathbb{R}^+ \bigcup \{0\} \]
\[ S = s_1s_2...s_n \overset{\mu}{\longrightarrow} \mu(S) = \sum_{i=0}^{n} s_i \]

In this case, the best individual would be \( S_{\text{best}} = 11...1 \) and it’s fitness value would be \( n \).
1.2 Classic Genetic Algorithms

In the early era of genetic algorithms, the presence of operators observed in nature was
dominant in the area. These first algorithms were based on three operators: selection,
crossover and mutation.

Selection

Selection is a generic term used to define a family of operators, whose purpose is to discard
the unfit and promote the fit, thus improving the overall fitness of the new population.
We now show the most natural way to define a selection operator.

Operator 1. (simple selection):

Given a search space $V$, a population $P$ of size $m$ and $\mu$ a fitness function, we define $\beta$ our
simple selection operator as follows:

$$\beta : V \rightarrow \mathcal{P}(V)$$

$$P = \{S_1, S_2, ..., S_m\} \quad \mapsto \quad \beta(S) = \{X_1, X_2, ..., X_q\}$$

Where $X_i \equiv \{P(X_i^{obs} = S_j) = \frac{\mu(S_j)}{\sum_{k=1}^{m} \mu(S_k)}\} \quad \forall i \in \{1, 2, ..., q\}$

Using this selection operator, often leads to premature convergence, if most individuals
are almost null fitted, or, if an early great individual is discovered, the population reaches
uniformness very quickly. To avoid this, we can use a similar operator:

Operator 2. (simple selection with sigma truncation):

Given a search space $V$, a non fitness-uniform population $P$ of size $m$ and $\mu$ a fitness function, we define $\beta_{st}$ our
selection operator as follows:

$$\beta_{st} : V \rightarrow \mathcal{P}(V)$$

$$P = \{S_1, S_2, ..., S_m\} \quad \mapsto \quad \beta_{st}(S) = \{X_1, X_2, ..., X_q\}$$

Where $X_i \equiv \{P(X_i^{obs} = S_j) = \frac{r \mu(S_j)}{\sum_{k=0}^{m} r \mu(S_k)}\} \quad \forall i \in \{1, 2, ..., q\}$ and $r$ is defined as fol-

$$r : \mathbb{R} \cup \{0\} \rightarrow \mathbb{R} \cup \{0\}$$

$$t \quad \mapsto \quad r(t)$$

$$\tilde{r}(t) = 1 + \frac{t - \text{mean}(\{\mu(S_1), \mu(S_2), ..., \mu(S_m)\})}{c \ \text{std}(\{\mu(S_1), \mu(S_2), ..., \mu(S_m)\})}$$

$$r(t) = \begin{cases} 
0 & \tilde{r}(t) \leq 0 \\
\tilde{r}(t) & \tilde{r}(t) > 0 
\end{cases}$$
This operator is correctly defined, as we have stated that the population is not fitness-uniform. In the case of fitness-uniformness, we can extend the definition of this operator by stating that in an fitness-uniform situation, the result after applying the operator is a uniform distribution or even an identity operator.

**Note1:** $c$ is a parameter that measures how separated is the sigma-truncated distribution of fitness, a bigger $c$ means a less sparse distribution. Usually, we want to set $c = 2$.

**Note2:** This operator also adds pressure on the late stage of convergence, enhancing the differences between individuals so that small improvements are taken into account.

**Note3:** This operator is known as the roulette wheel selection with sigma truncation. This is the selection operator we used on our implementations.

### Crossover

Crossover is an operator that mixes two individuals into two new individuals, potentially mixing two valuable string pieces and making an even better one. The classic crossover operator, crosses two individuals by choosing two cut points and exchanging the middle piece with one another.

![Figure 1.1: two point crossover illustration](image)

Let $\Omega^n$ be our search space. We define the two point crossover operator as follows:

\[
\beta : \Omega^n \times \Omega^n \rightarrow \Omega^n \times \Omega^n \\
\{S, \tilde{S}\} \mapsto \beta(\{S, \tilde{S}\}) = \{\beta_1(S, \tilde{S}), \beta_2(S, \tilde{S})\} = \{X_1, X_2\}
\]

We can represent $X_1$ as $(i, k)$ where $i$ represents the first crossover cut point and $k$ represents the second crossover cut point. $X_1$ and $X_2$ are probability distributions that completely depend on each other, thus, $P(X_2|X_1 \equiv (i, k))$ is a degenerate probability distribution:

\[
P(X_2|X_1 \equiv (i, k)) = \begin{cases} 
0 & X_2 \neq (i, k) \\
1 & X_2 = (i, k)
\end{cases}
\]

There is a bijection between the free $X_1$ (or $X_2$) distribution and $\mathcal{P}$, the following probability distribution over $\{1, 2, ..., n\} \times \{1, 2, ..., n\}$:
\[ P \equiv (\mathcal{U}_1(\{1, 2, \ldots, n-2\}), \mathcal{U}_2(\{k+1, k+2, \ldots, n-1\}|k = \mathcal{U}_1^{obs})) \]

Where \( \mathcal{U}_i \) represents a random uniform distribution.

The bijection actually represents each pair of cut-points of the crossover operator as \((i, k)\).

With this kind of crossover operator, we may destroy important information on the middle of the individuals string. To better illustrate this point, we introduce a new concept.

**Definition 7.** *(schema/schemata):*

Let \( \Omega \) be our alphabet. Let our search space be \( \Omega^n \).

We expand our alphabet to \( \Omega^* = \Omega \cup \{\#\} \), where \# represents a symbol \# \( \notin \Omega \). In this way, we obtain an expanded search space: \((\Omega^*)^n\). We may also call the expanded search space *schemata space*. We say that any string of our expanded search space is a *schema*.

Let \( S = s_1s_2\ldots s_n \in \Omega^n \) be a string, and let \( H = h_1h_2\ldots h_n \in (\Omega^*)^n \) be an schema.

We say *string S follows schema H* \( \iff \forall i \in \{1, 2, \ldots, n\}, s_i = h_i \lor h_i = \# \)

From an intuitive point of view, we may think that \# is a wild card symbol, a symbol that matches any other symbol.

Note: We may denote *string S follows schema H* as \( S \hookrightarrow H \).

**Proposition 1.** Let \( \Omega^n \) be our search space and let \((\Omega^*)^n \) be our schemata space. Let \( \mathcal{P}(\Omega^n) \) be the set of all subsets of \( \Omega^n \). There exists a subset \( H \subset \mathcal{P}(\Omega^n) \) and a bijective application:

\[
\varphi : (\Omega^*)^n \quad \varphi((\Omega^*)^n) = H
\]

\[ H \quad \varphi(H) = \{S \in \Omega^n | S \hookrightarrow H\} \]

The idea of schemata arises from a need to represent useful parts of a string, for a particular purpose or fitness niche. Let’s see some examples to better understand this point:

**Example 4.** Let \( \Omega^n = \{0, 1\}^n \) be our search space, with \( n > 2 \). Let \( \mu_1(S) = s_1 + s_2 \) be our fitness function. Consider the schema \( H_1 = 11\#\ldots\# \).

Observe that \( \forall S \hookrightarrow H_1, \mu_1(S) = 2 = \max_{S \in \Omega^n} \mu_1(S) \).

In fact, \( \mu_1(S) = 2 \iff S \hookrightarrow H_1 \)

Therefore, \( H_1 \) is the schema that contains the best strings with respect to fitness function \( \mu_1 \).
Example 5. Let $\Omega^n = \{0, 1\}^n$ be our search space, with $n > 4$. Let $\mu_2(S) = s_1 + s_n$ be our fitness function and let $H_2 = 1\#\ldots\#\ldots\#1$ be our schema.

Observe that $\forall S \rightarrow H_2$, $\mu_2(S) = 2 = \max_{S \in \Omega^n} \mu_2(S)$. In fact, $\mu_2(S) = 2 \iff S \rightarrow H_2$

Therefore, $H_2$ is the schema that contains the best strings with respect to fitness function $\mu_2$.

Example 6. Now consider the strings $S_0 = 0000\ldots00$, $S_1 = 1100\ldots0$ and $S_2 = 10\ldots0000\ldots1$ of length $n$. Let $\beta$ be our two point crossover operator.

\[
P(\beta_1(S_0, S_1) \rightarrow H_1) = \frac{n - 2}{n - 1}
\]

\[
P(\beta_2(S_0, S_1) \rightarrow H_1) = 0
\]

On the other hand,

\[
P(\beta_1(S_0, S_2) \rightarrow H_2) = 0
\]

\[
P(\beta_2(S_0, S_2) \rightarrow H_2) = 0
\]

Therefore, we see that it is more likely that $H_2$ schema is disrupted during a two point crossover operation. This simple example illustrates how the two point crossover operator disrupts sparse schemata more often than it disrupts tight schemata. Therefore, the representation we use with this operator, should depend on tight schemata, this is, related valuable information should be encoded on the string in the shortest way possible.

To solve this problem, we may draw a string distribution from the selected space, based on a distance (i.e. hamming distance), and sample from that distribution [18], [1]. With this new combination method, the population itself decides which string positions are important, and which are not.

Mutation

Mutation is the last one of the classic genetic operators. Mutation ensures new variability enters the population, because genetic material on an initial population tends to disappear with selection and crossover alone. We present the mutation operator as follows:
Operator 4. (simple mutation):

Let $\Omega$ be our alphabet, $\Omega^n$ our search space. We define $\beta_c$, our simple mutation operator as follows:

$$
\beta_c : \Omega^n \rightarrow \mathcal{P}(\Omega^n)
$$

$$
S \rightarrow \beta_c(S) = X
$$

where $X = x_1x_2...x_n$ and:

$$
x_i \equiv \begin{cases} 
  s_i & \text{with probability } c \\
  s \in \Omega - \{s_i\} & \text{with probability } \frac{1 - c}{\text{card}(\Omega) - 1}
\end{cases}
$$

Note 1: Usually, $c < 0.05$, since higher values $c$ values tend to slow down or even prevent convergence. For high $c$ values, the GA is essentially a random walk.

There are other mutation operators, based on distances between strings (such as Hamming distance for an arbitrary search space, or Cayley and Spearman distances if $\Omega^n = S_n$, where $S_n$ is the set of all permutations of length $n$) The simple mutation operator can be represented as a Hamming distance based distribution. In fact, the combination of crossover and mutation can be substituted by sampling from a distribution, with the so called EDAs, as seen in [18], [1].
Chapter 2

GA on function maximization

One of the advantages of Genetic Algorithms is its capability to optimize non differentiable or even non continuous functions. They are also a good at finding global minimum at functions with several local minima. However, they are not good at finding the actual minimum, as we will illustrate in this section, by using GA to try to find a global optima of different functions. Therefore, GA can be combined with other local search techniques to ensure local optimality, avoid premature convergence and sometimes, even speed up the search, as seen in [9], [6], [24].

2.1 Description of the algorithm

The Algorithm tries to find the maximum on the given closed n dimensional rectangle (K) using the classical genetic operators, and given the multivariate function f. K can be represented by two of its corners, $K \equiv (a_1, a_2, ..., a_n, b_1, b_2, ..., b_n)$, where $a = a_1...a_n$ is one of the corners, and $b = b_1...b_n$ the opposite corner.

If we denote length_of_representation by $l$, the search space is a discretization of $K$, discretized in $(2^l)^n = |K_{disc}|$ points. We encoded the search space in binary alphabet $\Omega = \{0,1\}$ using a bijective function $\phi_K$ defined as follows:

$$
\phi_K : K_{disc} \rightarrow (\Omega^l)^n
$$

$$
X = (x_1, ..., x_n) \rightarrow \phi(X) = ((bin \circ \theta_1)(x_1), ..., (bin \circ \theta_n)(x_n))
$$

Where $\theta_i(x_i, a_i, b_i) = \frac{(x_i - a_i)2^l}{b_i - a_i}$ and

$$
\text{bin} : \{1, 2, ..., 2^l\} \rightarrow \Omega^l
$$

$$
t_{dec} \rightarrow \text{bin}(t) = t_{bin}
$$

A sketch of the algorithm is shown below. For more on the implementation of the algorithm, refer to chapter 5.
Algorithm 1: search_max

Input:
$K = (a_1, a_2, ..., a_n, b_1, b_2, ..., b_n)$: a tuple containing the upper $a_i$ and lower $b_i$
bounds of the components of the search space, it represents two of the opposite
corners of $K$.  
$f(x_1, ..., x_n)$: function to be maximized. The function has to have $n$ input
arguments (corresponding to the $i$ indexes above).

Output:
$(\text{max } x, \text{max } f(x))$: a tuple containing found maximum and the point at which
the maximum was found.

Parameters:
$popsize$: The size of the population used by de GA.  
$max\_iterations$: The maximum number of generations to be computed.
$length\_of\_representation$: The length of the binary vector used to encode each $x_i$
component.

1 $\text{pop} \leftarrow \text{initialize\_population}(popsize, length\_of\_representation)$
2 $i \leftarrow 1$
3 $\text{best\_fitness} \leftarrow -\infty$
4 $\text{pop}\_\text{calculate\_fitness}(f, K)$
5 if $\max_{x \in \text{pop}} (x.\text{fitness}) > \text{best\_fitness}$ then
6     $\text{best\_fitness} \leftarrow \max_{x \in \text{pop}} (x.\text{fitness})$
7     $\text{best\_individual} \leftarrow \text{argmax}_{x \in \text{pop}} (x.\text{fitness})$
8 end
9 while $i < \text{max\_iterations}$ do
10     $\text{pop} \leftarrow \text{select}(\text{pop})$
11     $\text{pop} \leftarrow \text{crossover}(\text{pop})$
12     $\text{pop} \leftarrow \text{mutation}(\text{pop})$
13     $\text{pop}\_\text{calculate\_fitness}(f, K)$
14     if $\max_{x \in \text{pop}} (x.\text{fitness}) > \text{best\_fitness}$ then
15         $\text{best\_fitness} \leftarrow \max_{x \in \text{pop}} (x.\text{fitness})$
16         $\text{best\_individual} \leftarrow \text{argmax}_{x \in \text{pop}} (x.\text{fitness})$
17     end
18     $i \leftarrow i + 1$
19 end
20 $\text{search\_max} \leftarrow (\text{best\_individual}\_\text{decode}(), \text{best\_fitness})$
2.2 Algorithm’s performance

We only implemented the algorithm for functions with one and two variables, but the code can easily be extended to more dimensions. However, visualization on higher dimensions may be difficult.

To tested the performance of the algorithm, we used three test functions:

1) An uniparametric, multiple local maxima test function of our choice.
\[ f_1(x) = \cos(3x)(\frac{1}{4}x^2 + x + 5 + 5\sin^3(\cos(45x)) + 4\cos(45x) - 2x) \]

2) Inverted Rastrigin’s function, A = 10.
\[ f_2(x, y) = 40 - (x^2 + y^2) - 10\cos(2\pi x) + 10\cos(2\pi y) \]

3) Inverted Himmelblau’s function.
\[ f_3(x, y) = 30 + 14x + 21x^2 - x^4 + 22y - 2x^2y + 13y^2 - 2xy^2 - y^4 \]

\[ f_2 \] and \[ f_3 \] are popular choices when it comes to benchmarking of search algorithms, for further reading, refer to [20], [10].

2.2.1 Uniparametric test function

This function has many local optima, but only one global optimum, as it can be seen in figure 2.1. Running the algorithm 300 times with parameters:

\[ \text{popsize} = 100 \]
\[ \text{length}_\text{of}_\text{representation} = 15 \]
\[ \text{max}_\text{iterations} = 30 \]

The algorithm converged to a point near the global maxima on an average of 92.7%. We considered that the algorithm converged when:
\[ |x_{\text{output}} - x_{\text{max}}| < 0.1 \], which is represented by the area between the green lines in figure 2.1.

The algorithm, therefore, does not get stuck in local optima very often with this particular function.
2.2.2 Inverted Rastrigin’s function

This function, as can be seen in figure 2.3, has also many local optima, but only one global maximum, at \((x,y) = (0,0)\).

The red dots in these figures represent the output from the algorithm, after 300 runs, using the same parameters we used on \(f_1\). In this case, we can see the algorithm was not able to find the global optima most of the times. However, there are some options to improve the effectiveness, such as random restarts [22], [21], hybrid approaches with simulated annealing [19] or with particle swarm optimization [16]. The increase of the population size, and among other benefits, ‘the accuracy of the genetic algorithm approaches, but does not reach, 100%’. The greater the population size the greater the chance that the initial state of the population will contain a chromosome representing the optimal solution’ [8], but with an increased computation time. We estimated (see figure 2.2) the time complexity of the algorithm as a function of popsize to be \(O(\text{popsize})\), with fixed parameters:

\[
\text{length\_of\_representation} = 15 \\
\text{max\_iterations} = 30
\]

However, the population increase is not always beneficial, as can be seen in [2].
2.2. Algorithm’s performance

2.2.3 Inverted Himmelblau’s function

Himmelblau’s function is intended to test an algorithm’s ability to find multiple optima, since this function has 4 global optima, represented by black dots in figure 2.4. The red dots represent the output obtained after 300 runs with parameters:

\begin{align*}
\text{popsize} &= 100 \\
\text{length\_of\_representation} &= 15 \\
\text{max\_iterations} &= 30
\end{align*}

We estimated the probability of the algorithm outputting near each of the nodes with 300 runs, and we found the probabilities to be:

\begin{align*}
p(x_{\text{out}} > 0 \text{ and } y_{\text{out}} > 0) &= 0.497 \\
p(x_{\text{out}} > 0 \text{ and } y_{\text{out}} < 0) &= 0.237 \\
p(x_{\text{out}} < 0 \text{ and } y_{\text{out}} > 0) &= 0.167 \\
p(x_{\text{out}} < 0 \text{ and } y_{\text{out}} < 0) &= 0.100
\end{align*}

This suggests the GA results depend on the starting individuals, and the way individuals are chosen to crossover and mutate. It also shows how GA are a powerful tool to find different good answers to a problem.
Chapter 3

GA on interpolation

3.1 Description of the algorithm

Given a closed interval \([a, b]\), the functions \(f(x), H(x)\), where \(\frac{dH}{dx} \equiv f\) and an integer \(n\), the algorithm tries to choose the optimal interpolating points for a \(n\) degree polynomial, to interpolate \(f\) by minimizing \(\int_a^b \text{abs}(f(x) - \text{pol}(x))dx\).

Let \(V\) be the set of all closed intervals over \(\mathbb{R}\), let \(\mathcal{F}(\mathbb{R})\) be the set of all integrable functions on \(\mathbb{R}\) and let \(\mathcal{P}\) be the set of all polynomials over \(\mathbb{R}\). Without further encoding details, the fitness function \(\mu\) the algorithm uses is defined as follows:

\[
\mu : \mathcal{F}(\mathbb{R}) \times V \times \mathcal{P} \rightarrow \mathbb{R}
\]

\[
(f, [a, b], \text{pol}) \mapsto \mu(f, [a, b], \text{pol}) = 100 - \log(\int_a^b \text{abs}(f(x) - \text{pol}(x))dx)
\]

However, for high degree polynomials, due to the Runge’s phenomenon [23] it is difficult to estimate the value of the integral using numerical values or even analytically, due to the oscillator behavior of the polynomial. That is why we decided to consider an alternative fitness function, for any \(n > 10\):

\[
\mu_{n>10} : \mathcal{F}(\mathbb{R}) \times V \times \mathcal{P} \rightarrow \mathbb{R}
\]

\[
(f, [a, b], \text{pol}) \mapsto \mu(f, [a, b], \text{pol}) = 100 - \log(\text{abs}(\int_a^b f(x)dx - \int_a^b \text{pol}(x)dx))
\]

With this fitness function, the algorithm is trying to minimize \(\text{abs}(\int_a^b f(x)dx - \int_a^b \text{pol}(x)dx)\) instead of \(\int_a^b \text{abs}(f(x) - \text{pol}(x))dx\). However, in small \(n\) instances, this second fitness function is not as effective, as can be seen in figure 3.1. It does work on higher number of interpolating points, as we can see in figure 3.2 (a). For very high degree polynomials, however, the oscillations on the edges of the interval get more intense, increasing the error, 3.2 (b). All the tests on this section were done considering the following parameters:

- \(\text{popsize} = 100\)
- \(\text{length_of_representation} = 50\)
- \(\text{max_iterations} = 30\)
3.1. Description of the algorithm

(a) \( \int_{a}^{b} |f(x) - \text{pol}(x)| \, dx \) as loss function

(b) \( |\int_{a}^{b} f(x) \, dx - \int_{a}^{b} \text{pol}(x) \, dx| \) as loss function

Figure 3.1: Comparison of the two fitness functions on a 6 point interpolation on the function. Error is measured as the loss of the first fitness function.

(a) \( n = 25 \)

(b) \( n = 70 \)

Figure 3.2: Interpolation is quite successful in medium valued n’s, but it gets unstable for higher degree polynomials. Once again, error is measured with the loss function \( \int_{a}^{b} |f(x) - \text{pol}(x)| \, dx \).
Algorithm 2: interpolate

Input:
(a, b): a tuple containing the upper a and lower b bounds of the interpolating interval.
\( \mu \): fitness function, depends on the function to be interpolated, it’s indefinite integral and the considered (a, b) interval.

Output:
pol = \((x_1, \ldots, x_n)\): a n sized tuple, containing the interpolating x values. The interpolating points, therefore are \((x_1, f(x_1)), \ldots, (x_2, f(x_2))\).

Parameters:
popsize: The size of the population used by de GA.
max_iterations: The maximum number of generations to be computed.
length_of_representation: The length of the binary vector used to encode each \( x_i \) component.

1. \( \text{pop} \leftarrow \text{initialize\_population}(\text{popsize}, \text{length\_of\_representation}) \)
2. \( i \leftarrow 0 \)
3. \( \text{best\_fitness} \leftarrow -\infty \)
4. \( \text{pop}\_\text{calculate\_fitness}((a, b), \mu) \)
5. if \( \max_{x \in \text{pop}} (x.\text{fitness}) > \text{best\_fitness} \) then
6. \( \text{best\_fitness} \leftarrow \max_{x \in \text{pop}} (x.\text{fitness}) \)
7. \( \text{best\_individual} \leftarrow \arg\max_{x \in \text{pop}} (x.\text{fitness}) \)
8. end
9. while \( i < \text{max\_iterations} \) do
10. \( \text{pop} \leftarrow \text{select}(\text{pop}) \)
11. \( \text{pop} \leftarrow \text{crossover}(\text{pop}) \)
12. \( \text{pop} \leftarrow \text{mutation}(\text{pop}) \)
13. \( \text{pop}\_\text{calculate\_fitness}((a, b), \mu) \)
14. if \( \max_{x \in \text{pop}} (x.\text{fitness}) > \text{best\_fitness} \) then
15. \( \text{best\_fitness} \leftarrow \max_{x \in \text{pop}} (x.\text{fitness}) \)
16. \( \text{best\_individual} \leftarrow \arg\max_{x \in \text{pop}} (x.\text{fitness}) \)
17. end
18. \( i \leftarrow i + 1 \)
19. end
20. \( \text{interpolate} \leftarrow (\text{best\_individual}.\text{decode}(), \text{best\_fitness}) \)
Chapter 4

GA and Genetic Programming on list sorting

Genetic Programming (GP) is a branch of evolutionary algorithms that considers the search space as a set of programs, and the goal is to find the optimal program in the search space for a given task. One of the difficulties lies on the encoding of the programs. Some early attempts include Learning Classifier Systems [11], [12], [17], and Holland’s Broadcast Language (HBL), [11]. Other approaches, consider (GP) as computational trees, obtaining a Turing complete adaptable (through special genetic operators) programming languages [3] and GP with linear computation programs have also been tried [25].

4.1 First attempt: Hollands Broadcast Language

It has not yet been proven HBL is a Turing complete language, although Holland [11] suggested it to be. We chose to use HBL as our first implementation to our problem because it is a very natural extension of GA.

4.1.1 Theory introduction

In this section we briefly discuss the theory of HBL, and we give some examples. Most of this section has been taken from [11], [5] and [7]. We simplified the original approach to make it easier to implement and understand.

HBL is a message/action programming language, based on strings. Each ‘instruction’ of the program is of the form condition $\rightarrow$ action, where an action is performed $\iff$ the message received matches the condition field on the BU.

From [5] (adapted):

'We first describe the different structures constituting the language: the symbols, the broadcast units and broadcast devices. The interpretation of the symbols, broadcast units and broadcast devices will then follow. The broadcast language alphabet $\Omega$ is finite and contains ten different characters. We consider the set of strings over $\bigcup_{n=1}^{\infty} \Omega^n = S$.

$$\Omega = \{0, 1, *, :, :\hat{\downarrow}, \downarrow, \uparrow, \triangle, p, '\}$$

Let $I$ be an arbitrary string from $\Omega$. In $I$, a character is said to be quoted if it is preceded by the character ’. A broadcast unit $I_n$ is an arbitrary string from $B$, which only contains a unquoted *, and it is on the first position of the string. A set of broadcast units may be concatenated to form a broadcast device. A broadcast device $I$ may contain $0 \leq n \leq \infty$
broadcast units $I_1...I_n$ if $n = 0$ then, $I$ does not contain any broadcast unit and $I$ is then called a null device. A null device does not broadcast a signal under any circumstances. A broadcast device which is not null is said to be active and it may broadcast an output upon the detection of appropriate signals.

First, we need to process BD into several BU. We begin by uncommenting all BD by removing each ‘ symbol and the symbol immediately next to it. Then, the resulting string is divided so that every resulting BU begins with the symbol * and only contains that symbol once. Then, each BU is classified into one of five categories depending on the position of the first 3 : symbols (any subsequent : symbol is ignored).

0. any BU that does not fit on the other types. These are considered null BU.
1. *$I_1 : I_2$
2. * : $I_1 : I_2$
3. *$I_1 :: I_2$
4. *$I_1 : I_2 : I_3$

We call any $I_i$ a piece of BU. Each of the bu types has a different effect. We here give a very brief summary of how the BU processed.

1. If $I_1$ message is on the environment, cast $I_2$ to the environment.
2. If $I_1$ message is not on the environment, cast $I_2$ to the environment.
3. If $I_1$ and $I_2$ messages are on the environment, delete $I_2$ from the environment.
4. If $I_1$ and $I_2$ messages are on the environment, cast $I_3$ to the environment.

We first process type 4 BU, and then the others. Various ways of execution order can be considered, (Randomized, by type or by the order in which they appear on the BD). In the original BU design, if several messages match the same BU, one of them is chosen at random uniformly. This is done in order for the BD to be able to simulate any distribution (given enough length and computing power). We decided to consider a ordered environment, so that the oldest messages are processed first. This way, we ensure the execution of the BD is consistent on each run.

Although in the original approach all characters were allowed on environmental messages, on our approach, we restricted them to binary sequences, in order to reduce the number of BD that do nothing. In addition to this, our interpretation of the special characters is not the same as on the original paper and we did not use the symbol p on our project to simplify the implementation.
4.1. First attempt: Hollands Broadcast Language

◊: If this symbol is found on an action piece, it is ignored. If \( I = s_1s_2...s_n \) is our condition piece and \( M = m_1...m_r \) our environmental message, with \( r \geq m \). Then ◊ can appear on three positions:

1. **position** \( s_1 = ◊ \)
   
   - If \( s_2s_3...s_n \) matches \( m_{r-s}...m_{r-1}m_r \) \( \rightarrow \) \( I \) matches \( M \).

2. **position** \( s_i = ◊, i \notin \{1,n\} \)
   
   - If \( s_1...s_{i-1}s_{i+1}...s_n \) matches \( m_1...m_{i-1}m_{i+1}...m_{r-1}m_r \) \( \rightarrow \) \( I \) matches \( M \).

3. **position** \( s_n = ◊ \)
   
   - If \( s_1 \in \{◊, ∇, ▼\} \) \( \rightarrow \) \( s_n \) is ignored.
   - ElseIf: \( s_1...s_{n-1} \) matches \( m_1...m_{n-1} \) \( \rightarrow \) \( I \) matches \( M \).

Notice that due to the restriction in 3. **position**, \( s_1 = ◊ \land s_n = ◊ \) is not possible.

∇, ▼: These symbols have a similar effect as ◊ symbol, but if found at 2. **position**, they are ignored. They also save the exceeding part of the environmental message, and 'inject' it on the action piece, if the symbol is repeated at the action string. I.e. Let \( BU_1 = *∇11 : 00∇0 \) be our BU and \( \{0111, 000\} \) be the set of our environmental messages. Notice the following:

- 000 does not match ∇11
- 0111 matches ∇11, therefore, 01 is saved in memory, and since ∇ appears on the action piece, message 00010 is broadcast to the environment.

△: This symbol acts as a single position copier. It will match any one character, and if the △ also appears in the action message, it will inject the copied character. Only one first △ instance is considered in the condition pieces and on the action piece, any subsequent occurrences are discarded. If there is no △ in the condition piece, any occurrences of this symbol in the action piece are discarded.

4.1.2 Our approach

Our first attempt only considers lists of size 8. We now introduce the major algorithms used in this first attempt.

The algorithms used

To define the interaction between the messages and the environment, we introduce a procedure called **process_messages**:
Algorithm 3: process_messages

Input:
- \(sl\): The list to be ordered.
- \(M\): A list containing all environmental messages. All messages are binary arrays.

Output:
- \(M_{\text{new}}\): New environmental messages.
- \(sl\): The list to be ordered, after it has been modified.

```plaintext
new_M ← ∅
forall message ∈ M do
    if length(message) != 8 then
        continue
    else
        identifier ← message[0] message[1]
        pointer0, pointer1 ← min(pointer0, pointer1), max(pointer0, pointer1)
        switch identifier do
            case 00 or 11 do
                if sl[pointer0] > sl[pointer1] then
                    M_new ← M_new ∪ {11 ∪ decimal_to_binary(pointer0)}
                else
                    M_new ← M_new ∪ {00 ∪ decimal_to_binary(pointer0)}
                end
            end
            case 10 do
                sl ← sl[pointer1 : end] ∪ sl[pointer0 : pointer1] ∪ sl[pointer1]
            end
            case 01 do
                sl[pointer0], sl[pointer1] ← sl[pointer1], sl[pointer0]
            end
        end
    end
end
return sl, M_new
```
We also define the procedure \textit{process\_BD}, which interacts with the environmental messages.

\begin{algorithm}[H]
\begin{algorithmic}[1]
\State \textbf{Input:}
\State \textit{M}: A list containing all environmental messages. All messages are binary arrays.
\State \textit{BD}: The broadcast device to be processed.
\State \textbf{Output:}
\State \textit{M}: Modified environmental messages.
\State \textbf{Parameters:}
\State \textit{max\_env\_messages}: The number of maximum environmental messages.
\State \textit{max\_env\_message\_length}: Any environmental message containing more characters than this parameter, will be cropped to this length.

\State $BU\_list \leftarrow \text{decompose}(BD)$
\State \textbf{forall} $BU \in BU\_list$ \textbf{do}
\State \hspace{1em} \textbf{forall} message \in M \textbf{ do}
\State \hspace{2em} \textbf{if} match($BU$, message) \textbf{ then}
\State \hspace{3em} reply \leftarrow \text{answer}(BU, message)$
\State \hspace{3em} M \leftarrow M - \{message\}$
\State \hspace{3em} M \leftarrow M \cup reply$
\State \hspace{3em} \textbf{break}$
\State \hspace{1em} \textbf{end}$
\State \hspace{1em} \textbf{end}$
\State \textbf{end}$
\State M \leftarrow \text{shorten\_and\_cut}(M, max\_env\_messages, max\_env\_message\_length)$
\State \textbf{return} M
\end{algorithmic}
\end{algorithm}
We define the algorithm that sorts a list given a list and a BD.

**Algorithm 5: sort_list_with_BD**

**Input:**
- \( BD \): input BD.
- \( shuffled_list \): the list to be shuffled.

**Output:**
- \( shuffled_list \): shuffled list once it has been ordered.

**Parameters:**
- \( start\_env\_message\_list \): The initial environmental message list. We used \( \{11000111\} \) to force the first comparison.
- \( max\_iterations \): The maximum number of comparison rounds.

```plaintext
1 \( M \leftarrow start\_env\_message\_list \)
2 \( i \leftarrow 0 \)
3 \( \text{while } i < max\_iterations \text{ do} \)
4 \( \quad i \leftarrow i + 1 \)
5 \( \quad shuffled\_list, M \leftarrow process\_messages(shuffled\_list, M) \)
6 \( \quad M \leftarrow process\_BD(M, BD) \)
7 \( \text{end} \)
8 \( \text{return } shuffled\_list \)
```
We define the Fitness function, \textit{calculate\_fitness}.

\begin{algorithm}
\caption{Fitness\_function} \label{alg:fitness}
\begin{algorithmic}
\State \textbf{Input:}
\State \texttt{BD}: BD whose fitness is evaluated.
\State \textbf{Output:}
\State \textit{calculate\_fitness}: the calculated fitness value.
\State \textbf{Parameters:}
\State \texttt{number\_of\_lists\_sampled}: the number of lists on which the fitness is calculated.
\State \State \texttt{i} $\leftarrow 0$
\State \texttt{fitness\_value} $\leftarrow \emptyset$
\While {\texttt{i} < \texttt{number\_of\_lists\_sampled}}
\State \texttt{i} $\leftarrow i + 1$
\State \texttt{sl} $\leftarrow$ \texttt{generate\_random\_list(shuffled\_list\_length)}
\State \texttt{sl\_copy} $\leftarrow$ \texttt{copy(sl)}
\State \texttt{sl} $\leftarrow$ \texttt{sort\_list\_with\_BD(BD, sl)}
\State \texttt{fitness\_hist} $\leftarrow$ \texttt{calculate\_entropy(sl\_copy) - calculate\_entropy(sl)}
\EndWhile
\State \texttt{return} \texttt{mean(fitness\_hist)}
\end{algorithmic}
\end{algorithm}
Finally, the algorithm we used to try to discover the best BD on the search space.

**Algorithm 7: find_best_BD**

**Input:**

`void`

**Output:**

`Best_BD`: the best BD found.

**Parameters:**

- `popsize`: The size of the population used by de GA.
- `max_ga_iterations`: The maximum number of generations to be computed.
- `length_of_BD`: The length of the BD considered in the search space.

```plaintext
1 pop ← initialize_random_uniform_population(popsize, length_of_BD)
2 best_fitness ← −∞
3 pop.calculate_fitness()
4 if max(x.fitness) > best_fitness then
   5   best_fitness ← max(x.fitness)
   6   best_individual ← argmax(x.fitness)
7 end
8 i ← 0
9 while i < max_iterations do
10   pop ← select(pop)
11   pop ← crossover(pop)
12   pop ← mutation(pop)
13   pop.calculate_fitness()
14   if max(x.fitness) > best_fitness then
15      best_fitness ← max(x.fitness)
16      best_individual ← argmax(x.fitness)
17   end
18   i ← i + 1
19 end
20 return best_individual
```
The problems with this approach

On our first implementation, we found several problems:

- The fitness value was always 0 no matter what. This was caused by the fact that it is highly unlikely to randomly generate a BD that was able to make a change to the fitness list. To solve this, we classified all BD into two categories: null fitness BDs and BDs with non null fitness. We then initialized the population with only BD from the second category.

- We now had the initial population full of non null fitness BDs since we had initialized them discarding any null fitness BD. Bear in mind that for a population size of 20, it took about 30 minutes to initialize our population, so it was a highly inefficient solution. In addition, once initialized, the first two iterations destroyed any valuable schemata on the population and fitness was once again null every time.

To solve this, we tried to reduce the search space: the BU contained on each BD were set to be type1 and have the following structure:

\[ \ast XXXXXXXX : YYYYYYYY \]

Where \( X \in \{0, 1, \Diamond, \nabla, \Delta\} \) and \( Y \in \{0, 1\} \)

- With this we were able to get BDs that on average, improved any given lists entropy. Nonetheless, we still were not able to converge to a BD that was able to completely sort any given list.

We tried a new selection method, based on EDAs, truncating the population (selecting the best k individuals) and then sampling each position of the BD as an independent probability distribution. This did not improve the obtained solution.

We also tried introducing a new operator, \( r \) which would be randomly (fair coin toss) set to 0 or 1 each time it was read. This new operator improved our best found solution, but it depended too much on randomness, since the solutions we obtained were not deterministic programs. Each time they were executed, the outcome was different, and this is a undesirable property when it comes to list sorting.

Finally we decided to start from scratch with a smaller search space, and a new approach.

4.2 Our second attempt: Linear Genetic Programming

Based on the Turing Machine idea, we created a linear program with several commands and we evolved it using classic genetic operators.

4.2.1 Our Linear Programs Explained

We have two pointers, \( p_0 \) and \( p_1 \), and they represent a position (an index) on the list, \( p_1 \) is initialized to 0, and \( p_1 \) is initialized to \( \text{floor} \left( \frac{\text{length of list}}{2} \right) \).
We then defined the following operators: $s, i, 0+, 1+, 0-, 1-, 0++, 0--, 1++, 1-- , c$.

$s$: when this symbol is found, the following is executed:

If $p_0 \neq p_1$

$\text{compare\_and\_swap}(p_0, p_1, \text{shuffled\_list})$

Where $\text{compare\_and\_swap}$ compares the elements of the list corresponding to $p_1$ and $p_0$ and swaps the elements if it makes the list more ordered.

$i$: when this symbol is found, the following is executed:

If $p_0 \neq p_1$

$\text{compare\_and\_insert}(p_0, p_1, \text{shuffled\_list})$

Where $\text{compare\_and\_insert}$ compares the elements of the list corresponding to $p_1$ and $p_0$ and inserts the element corresponding to $p_0$ in the place it belongs: next or before the element corresponding to $p_1$.

$k++$ and $k-$: $k$ can be either $0$ or $1$. If $0+$ is found we execute:

$p_0 \leftarrow p_0 + 1 \mod (\text{list\_length})$

Likewise, if we find $1-$:

$p_1 \leftarrow p_1 - 1 \mod (\text{list\_length})$ is executed.

$k++$ and $k-$: $k++$ is equivalent to executing the command $k+$, $\lfloor \frac{\text{list\_size}}{8} \rfloor$ times consecutively. The homologous happens with $k--$.

$c$: When this command is found, it is ignored.
The optimization algorithm is the same as the one used on the first approach, so we won’t be repeating it here. It should be noted that for list sizes 4 and 5, a brute force search was used, since it was more efficient than our algorithm.

4.2.2 Results

This time we were able to find algorithms that sorted any given list. We compared the obtained algorithms with two well known algorithms: insertion sort and quicksort. Bear in mind that when the list is ordered, the algorithm is terminated and the number of comparisons is measured. This is not normally the case, since the algorithm normally knows when to terminate. However, we measured the maximum number of comparisons needed to sort any list, and thus when trying to sort any list, if we execute the GP with the maximum number of comparisons needed for all lists, we know the list will be sorted.

To make the comparison between different algorithms fairer, on the insertion sort, after each insertion, we checked if the list is ordered, and if so, we terminated the algorithm, just like on the GP.

We did not do this on quicksort, due to the recursive nature of the algorithm. Therefore, the value that must be taken into account is the worst case for the obtained genetic program, since this program cannot by itself terminate, the only way to ensure any list can be sorted by this algorithm, is to let it make all the comparisons it makes on the worst case scenario.

Figure 4.2: The number of comparisons required to sort a list given its size. Lighter colors represent the mean of the number of comparisons needed, and the darker colors represent the maximum required comparisons. (Smaller is better)
We only computed the GP for lists of size 4 to 10, since for bigger list the computation time required to obtain a reasonable solution on our machine would exceed 6 hours. In figure 4.2 we can see a comparative chart. The obtained best algorithm is worse than this two well known algorithms.
Chapter 5

Methodology and Coding Challenges

On this last chapter we will talk about some of the difficulties we faced when implementing the code. There were various different unsuccessful attempts which were not included on this document, because of the lack of space, and the obtained unexciting results.

5.1 Max Search

![Diagram of GA implementation]

Figure 5.1: General structure of our GA implementation. Blue quadrilaterals represent properties, and green quadrilaterals represent classes.

We decided to structure the code using Matlab’s OOP capabilities. Our main object is a Population type object, which contains the methods needed to run the GA. This structure was used both in Max Search and Polynomial Interpolation.

Some of the major challenges we faced while implementing this algorithm:

- Sometimes, selection function would crash. It was caused because very occasionally, specially with small population sizes, all individuals would have the same fitness, and therefore, there would be an 0 division error while calculating the probability of any individual on the roulette wheel selection. To solve this, we applied a small normal noise, before the selection function (line 17 of select_ga.m).
While encoding and decoding between binary and real numbers, we had numerical problems because of the limit of integer size on Matlab. That is why we introduced a small python script (*Bigint.py*), since python has unlimited precision integers. Matlab has also got a third party package that supports unlimited sized integers, called **Variable Precision Integer Arithmetic**, but it was not easy to use and it had some limitations, so we figured out it would be easier to leave the big integer part to python. We see the call the python script on line 31 of *code_ab.m* and line 19 of *decode_ab.m*. Python returns to Matlab a string that first needs to be converted to a Matlab string, and then it can be evaluated with the `eval()` command.

5.2 Polynomial Interpolation

- The major problem we faced is the numerical error when integrating the L1 distance of two functions on Matlab. To avoid numerical errors, we introduced another loss function, which worked surprisingly well for big degree polynomials \((n > 10)\). More details on section 3.1.

5.3 List Sorting

- The Broadcast Device approach turned to be a big waste of time. We only considered type 1, 2 and 4 BUs, since type 3 BUs would not change much anyway. This approach had many problems, such as infinite loops, out of memory errors (caused by unlimited message lengths). After correcting all these problems, we did obtain a BD that seemed to improve a list’s entropy on average. On figure 5.2 we see that the distribution of fitness values is slightly off to the right, and the mean fitness value is 0.0902. There is still much to improve, since it can barely be called a sorting algorithm.

![Histogram of best BD's fitness values](image)

Figure 5.2: Histogram of the found best solution’s fitness values over all lists of size 8 (there are 8! different lists of this size.) The red line represents the number of instances with null fitness. Positive fitness values represent the list’s entropy was improved, and negative values represent it was worsened.

To improve the obtained result, we restricted our search space as stated in 20. We also introduced ‘r’ the random operator. With these changes, we were able to find a better
solution, *lwlwllww:11rrr00r*wlwl0w11:10111010. With this BD, we obtained an average fitness value of 0.142. The distribution of the fitness values over all possible lists can be seen in figure 5.3. The obtained result depends on the used random seed, since the ‘r’ operator is not deterministic (more about the ‘r’ operator on 20).

![Histogram of best BD's fitness values](image)

Figure 5.3: Histogram of the found best solution’s fitness values over all lists of size 8 (there are 8! different lists of this size.) The red line represents the number of instances with null fitness. Positive fitness values represent the list’s entropy was improved, and negative values mean it was worsened.
Chapter 6

Future Work

Our final approach for list sorting can still be improved, since it is a very simple algorithm. Whenever a number is inserted, we know that the element corresponding to the inserted index is correctly arranged with the inserted element. If we again insert on the same location this information can be exploited, which is what insertion sort does. For reasonable sized GPs, however, there is no solution on the search space that can exploit this information. It would not take much to adapt our second approach to take advantage of this information.

The idea would be to obtain small groups of 'linked' elements, and these groups would be arranged subsets. These linked groups would be created or extended whenever an insertion happened. The insertion operator can be changed to binary insertion. In regard to swaps, whenever a swap between two elements of the same group happens, this could be ignored. If a swap occurs between two different groups, these could be combined like mergesort does, obtaining a single bigger group.

When it comes to optimization methods, EDAs [18], [1] could be implemented. Also, other crossover operators could be used [15].
Appendix A

Implemented Code
A.1 Matlab and Python Code for Max Search

A.1.1 Main Functions and Classes

main.m

```matlab
function [ population ] = main( txtfile , rep_len , pop_size , i_max )
% use nextline.m to read the txtfile
clear nextline
mode = nextline(txtfile);
fitness_opts = nextline(txtfile);
fitness_opts = eval(fitness_opts);
fitness_mode_opts = {mode,fitness_opts{:}};
len = eval(nextline(txtfile));
popsize = eval(nextline(txtfile));
imax = eval(nextline(txtfile));

% uncomment these lines to override data.txt values with input values
% len = rep_len;
% popsize = pop_size;
% imax = i_max;

switch mode
    case 'function_max'
        C = zeros(1,len);

        % initialize population with randpop, and write mode_fitness_opts on population
        population = randpop(C,popsize,fitness_mode_opts);

        close all % close all previous figures
        hold all

        % we create two points to name the legends, make room for legends, and preallocate x axis
        scatter(0,0,'.','red')
        scatter(0,imax,'.','blue')
        legend('Max. fitness','Mean fitness','Location','southeast')

        for i = 1:imax
            matingpool = select_ga(population);
            matedpool = crossover(matingpool);
            population.pop = mutate(matedpool);
            scatter(i,population.bestfitness,'.','MarkerEdgeColor','red')
            scatter(i,population.meanfitness,'.','MarkerEdgeColor','blue')
            drawnow
        end

        plot function and found maxima
        figure
        a = fitness_mode_opts{2};
        b = fitness_mode_opts{3};
        f = fitness_mode_opts{4};
        x = decode_ab(population.bestpopindividual.chrompack.genome,length(C),a,b);
        hold all

        mindist = min([x-a,b-x]);
        xmax = fminbnd(@(x) f(x),x - (mindist/1000),x + (mindist/1000));
        %imax = -8.3778;
        plot(a:(b-a)/1000000:b,f(a:(b-a)/1000000:b))
        %ga maximum
        plot(xmax,f(xmax), 'ro', 'markersize', 4)
        plot([xmax-0.1,xmax-0.1], ylim, 'color',[0 0.6 0.3])
        plot([xmax+0.1,xmax+0.1], ylim, 'color',[0 0.6 0.3])
        legend( 'F1', 'maxima', 'convergence area', 'Location','southeast')
        disp('error: ')
        disp( abs(f(xmax) - f(x)) )
```
% code below used to measure the probability of terminating the algorithm on the convex area

```matlab
if 0.1 > abs(x - xmax)
    population = 1;
else
    population = 0;
end
hold off
fclose all;
population = {x, f(x)}
```
```matlab
function [fitness] = calculatefitness(genome, fitness_opts)
    mode = fitness_opts{1};

    switch mode
    case 'function_max'
        fitness = function_max(genome, fitness_opts);
    case 'multiparam_max'
        fitness = function_max_multipar(genome, fitness_opts);
    end

function [fitness] = function_max(genome, fitness_opts)
    %mode = 'function_max'
    %fitness_opts = { 'function_max', a, b, @(x) f(x) }

    f = fitness_opts{4};
    x = decode_ab(genome, length(genome), fitness_opts{2}, fitness_opts{3});
    fitness = f(x);
end

function [fitness] = function_max_multipar(genome, fitness_opts)
    %mode = 'multiparam_max'
    %fitness_opts = { 'multiparam_max', a1,b1,a2,b2, f }

    f = fitness_opts{6};
    x = decode_ab(genome(1,:), length(genome(1,:)), fitness_opts{2}, fitness_opts{3});
    y = decode_ab(genome(2,:), length(genome(2,:)), fitness_opts{4}, fitness_opts{5});
    fitness = f(x,y);
end
```
A.1. Matlab and Python Code for Max Search

Chrompack.m

```matlab
classdef Chrompack
    properties
        genome
    end
    methods
        function r = individualize(chrompacks)
            individuals = (arrayfun(@(x)Individual(x), chrompacks, 'UniformOutput', false));
            r = [individuals{:}];
        end
        function obj = Chrompack(gen)
            if nargin
                obj.genome = gen;
            end
        end
        function r = fitness(obj, fitness_mode_opts)
            r = calculatefitness(obj.genome, fitness_mode_opts);
        end
    end
end
```
function [ C ] = code_ab ( x , n , a , b )

% CODE_AB returns the binary code of the entered number. Number of digits
% and ab interval is also taken into account
% x --> char / int --> value to be coded to binary, x[A,a,b] is a must
% n --> int OPTIONAL --> number of digits of the binary code
% a --> numeric OPTIONAL --> lower bound of interval
% b --> numeric OPTIONAL --> upper bound of interval
% IMPORTANT: function has to be called with 4 arguments at least once
% before calling it with only one argument, as persistent values need to
% be set.

persistent per_abd;
persistent persistent_n;

% NARGIN 1
if nargin == 1
    % if x is not an string
    if ~ ( ischar ( x ) )
        % if x is not a string nor a number --> error
        if ~ ( isnumeric ( x ) )
            error ( ' x is not numeric nor string ' )
        end
    end
    % check if x A [a,b], else --> error
    if per_abd ( 1 ) <= x && per_abd ( 2 ) >= x
        % code with python module, output is python string.
        % we then convert python string to matlab string
        C_string = char ( py . phyton . bigint . code ( x , persistent_n , per_abd ( 1 ) , per_abd ( 2 ) ) ) ;
        % finally, we evaluate function
        C = eval ( C_string ) ;
    else
        error ( ' x[A,a,b] is not true ' )
    end
else
    % try to call code_ab with a converted string
    C = code_ab ( num2str ( x , ' %100.30 f ' ) ) ;
end

% NARGIN 4
% set persistent values for ab and n
elseif nargin == 4
    % error check a > b
    if a > b
        error ( ' b > a on [a,b] interval ' )
    end
    per_abd = [ a , b , b-a ] ;
    persistent_n = n ;
    C = code_ab ( x ) ;
end
end
function [ matedpool ] = crossover ( matingpool )
% Input ---> array of Individuals
% Output ---> array of Individuals
chrompackarray = [ matingpool . chrompack ]; %we obtain the chrompacks from individuals
outputchroms = simplecrossover ( chrompackarray ); %perform crossover
matedpool = individualize ( outputchroms ); %transform obtained chromosomes to individuals
end

function [ crossedchrompacks ] = simplecrossover ( chrompacks )
% SIMPLECROSSOVER
% Input ---> array of Chrompack
% Output ---> array of Chrompack since array of genomes may be problematic
% crossover inside each chromosome.
% [ a a a b b c c c ] & [ t t t e e f f f ] -----> [ a a a e e c c c ] & [ t t t b b f f f ]
% control --> logical variable 0 when pop even, 1 when pop odd
control = 0;

%check if population is multiple of 2
len = length ( chrompacks );
if mod ( len , 2 )
    control = 1;
    warning ( 'population size is odd, an individual will not be crossed' )
end

%we shuffle input
chrompacks = shuffle ( chrompacks );

%preallocate output for speed
crossedchrompacks ( 1 : len ) = Chrompack ();

%crossing
for index = 1 : floor ( len / 2 )
    parents = chrompacks ( 1 , [ index * 2 - 1 , index * 2 ] );
    crossedchrompacks ( [ index * 2 - 1 , index * 2 ] ) = couplemate ( parents ( 1 ) , parents ( 2 ) );
end

if control %if population is odd, last individual is not crossed and kept
    crossedchrompacks ( end ) = chrompacks ( end );
end

function [ child1 , child2 ] = couplemate ( parent1 , parent2 ) %we mate a couple of chrompacks
% Input ---> Chrompack , Chrompack
% Output ---> Chrompack , Chrompack
% we first convert input into genome
parent1 = parent1 . genome ;
parent2 = parent2 . genome ;

%get nchrom and chromsize
[nchrom , chromsize ] = size ( parent1 );

%get random positions
randomindex = sort ( randi ( chromsize - 1 , 2 , nchrom ) );

child1gen = parent1 ;
child2gen = parent2 ;
for i = 1 : nchrom
    child1gen ( i , ( randomindex ( 1 , i ) + 1 ) : randomindex ( 2 , i ) ) = ... 
    child2gen ( i , ( randomindex ( 1 , i ) + 1 ) : randomindex ( 2 , i ) ) = ... 
    parent1 ( i , ( randomindex ( 1 , i ) + 1 ) : randomindex ( 2 , i ) ) = ... 
    parent2 ( i , ( randomindex ( 1 , i ) + 1 ) : randomindex ( 2 , i ) ) = ... 
end
child1 = Chrompack ( child1gen );
child2 = Chrompack ( child2gen );
%mode ---> tell the main file what mode will be operating on

function_max

%Fitness function options. For 'function_max' mode, a cell containing:
% { 'function_max', a, b, f }

{ -10 , 10 , @(x) cos(3*x) .* (0.25*x.^2+5+5*sin(cos(45*x)).^3+4*cos(x*45)-2*x) }

%The sample Individual's genome (C)

15

%Population size (popsize)

100

%Number of iterations (imax)

20

%93

END
datamultimodal.txt

%mode ----> tell the main file what mode will be operating on

% Fitness function options. For 'multiparam_max' mode, a cell containing:
% { 'multiparam_max', a1 , b1 , a2 , b2 , f }

% demo1
%{-10 , 10 , -10, 10 , @(x,y) cos(0.05.*x.*y) .* (0.25.*x.^2+y)}

% Rastrigin A = 10
%{-5 , 5 , -5, 5 , @(x,y) 40 -(x.^2 + y.^2) + 10.*cos(2.*pi.*x) + 10.*cos(2.*pi.*y)}

%Himmelblau’s function
%{-6 , 6 , -6, 6 , @(x,y) 200 -(x.^4 + 2.*x.^2.*y - 22.*x.^2 + y.^2 - 22.*y +121) -(x.^2 + 2.*x.*y .^2 - 14.*x + y.^4 - 14.*y.^2 + 49)}

%The sample Individual’s genome (C)
15

%Population size (popsize)
100

%Number of iterations (imax)
30

END
decode_ab.m

1 function [ x ] = decode_ab( C , n , a , b )
2 %DECODE_AB returns the binary code of the entered number. Number of digits
3 %and ab interval is also taken into account
4 % C—>char / mat of doubles ——> value to be decoded to decimal
5 % n—>int OPTIONAL ——> number of digits of the binary code
6 % a—>numeric OPTIONAL ——> lower bound of interval
7 % b—>numeric OPTIONAL ——> upper bound of interval
8 %IMPORTANT: function has to be called with 4 arguments at least once
9 %before calling it with only one argument, as persistent values need to
10 %be set.
11
12 persistent per_abd;
13 persistent persistent_n;
14
15 %Nargin 1
16 %decode C
17 if nargin == 1
18 if ischar(C)
19 x = eval(char(py.phyton.bigint.decode( C , persistent_n , per_abd(1) , per_abd(2) )));
20 elseif isnumeric(C)
21 a = mat2str(C); %vector ——> str [1 2 3] ——> str [1,2,3]
22 x = decode_ab(strrep(a, ’ ‘, ’ , ’));
23 end
24 end
25
26 %NARGIN 4
27 %set persistent values for ab and n
28 elseif nargin == 4
29
30 if a > b
31 error( ’b > a on [a,b] interval’)
32 end
33 per_abd = [a , b] ;
34 persistent_n = n ;
35 x = decode_ab(C);
36 end
37 end
38 end
39
getdata_mutimodal.m

% get probabilities of Himmelblau's function maxima

```matlab
memory = [0 0 0 0];
% hold all
x_vect = linspace(-5,5,1000);
% y_vect = linspace(-5,5,1000);
[x_mesh, y_mesh] = meshgrid(x_vect, y_vect);
% 
%f = @(x,y) 200 - (x.^4 + 2.*x.*y - 22.*x.^2 + y.^2 - 22.*x*y +121) - (x.^2 + 2.*x.*y.^2 - 14.*x + y.^4 - 14.*y.*2 + 49);
%f = @(x,y) 40 - (x.^2 + y.^2) + 10.*cos(2.*pi.*x) + 10.*cos(2.*pi.*y);
% z_mesh = f(x_mesh,y_mesh);
% mesh(x_mesh , y_mesh , z_mesh)
% %plot3 (3.5804, -1.8200, 200, 'k.', 'MarkerSize', 7)
% %plot3 (-2.8077, 3.1340,200,'k.', 'MarkerSize', 7)
% %plot3 (-3.7767,-3.2761,200,'k.', 'MarkerSize', 7)

for i = 1:300
  dat = main('datamultimodal.txt',15,100,30);
  if dat{1}>0 & dat{2}>0
    memory = memory + [1 0 0 0];
  elseif dat{1}>0 & dat{2}<0
    memory = memory + [0 1 0 0];
  elseif dat{1}<0 & dat{2}>0
    memory = memory + [0 0 1 0];
  elseif dat{1}<0 & dat{2}<0
    memory = memory + [0 0 0 1];
  end
end

disp(i)
% scatter3(dat{1},dat{2},dat{3},'r.')
% drawnow
% clear main
end

disp(memory)
% 149 71 50 30
```
getdata_unimodal.m

1  %get probabilities of Himmelblau's function maxima
2
3  close all
4  memory = [0 0];
5  % hold all
6  % x_vect = linspace(-5,5,1000);
7  % y_vect = linspace(-5,5,1000);
8  % [x_mesh, y_mesh] = meshgrid(x_vect, y_vect);
9  %
10  % f = @(x,y) 200 - (x.^4 + 2.*x.^2.*y - 22.*x.*y +121) - (x.^2 + 2.*x.*y.^2 - 14.*x + y.^2 - 14.*y + 49);
11  %
12  % z_mesh = f(x_mesh,y_mesh);
13  % mesh(x_mesh, y_mesh, z_mesh)
14  % plot3(3.5804, -1.8200, 200, 'k.', 'MarkerSize', 7)
15  % plot3(-2.8077, 3.1340, 200, 'k.', 'MarkerSize', 7)
16  % plot3(2.9905, 2.0118, 200, 'k.', 'MarkerSize', 7)
17  % plot3(-3.7767, -3.2761, 200, 'k.', 'MarkerSize', 7)
18  
19  for i = 1:100
20    dat = main('data.txt');
21  
22    % [not_found_maxima, found_maxima]
23    memory(dat + 1) = memory(dat + 1) + 1;
24    disp('----------')
25    disp(memory)
26  clear main
27  end
Individual.m

```matlab
classdef Individual < handle
    %INDIVIDUAL an individual of the population
    
    properties
        chrompack
        fitnessval = nan;
    end
    
    methods
        
    %fitness(obj) calculates the fitness value of the individual
    function r=fitness(obj,fitness_mode_opts)
        if isnan(obj.fitnessval)
            obj.fitnessval=fitness(obj.chrompack,fitness_mode_opts);
        end
        r=obj.fitnessval;
    end
    
    %constructor checks if genome provided in which case creates obj,
    %constructor checks if given argument is a genome (matrix) or a
    %chrompack (Chrompack object) and creates individual accordingly
    function obj = Individual(information)
        if nargin == 0
            elseif nargin == 1
                len = length(information,'Chrompack')
                if len == 1;
                    obj.chrompack = information;
                else
                    warning(['use individualize method from Chrompack class instead'])
                end
            elseif isa(information,'double')
                obj.chrompack = Chrompack(information);
            end
        end
    end
end
```
measure_time.m

1  list = zeros(1,4);
2  speedvect = [];
3  for i = 20:20:600
4    for j = 1:4
5      tic()
6        % repr length i = 15:30:615
7        % [1.0682  1.7389  2.4093  3.0609  3.8373  4.3579  5.0001  5.7808  6.3701
9          13.0258 13.7616 14.4588]
10        % main('datamultimodal.txt',i,20,30);
11    end
12    list(j) = toc();
13    speedvect = [speedvect, min(list)];
14    disp(i)
15  end
16  disp(speedvect)
% A.1. Matlab and Python Code for Max Search

mutate.m

function [mutated_Individual_array] = mutate(Individual_array, p)
% MUTATE mutate single character

% set default p mutation if not given by user
if nargin == 1
    p = 0.2;
end

chrompack_array = [Individual_array.chrompack];
preallocated_chroms(1, length(Individual_array)) = Chrompack();
for i = 1:length(chrompack_array)
    preallocated_chroms(i) = mutate_chrom(chrompack_array(i), p);
end
mutated_Individual_array = individualize(preallocated_chroms);

function [mutated_chrompack] = mutate_chrom(chrompack, p)
% MUTATE mutate single character
% input --> chrompack
% output --> chrompack

% mutate single bit only if favorable biased coin toss
if not(randi([0, floor(1/p)]))
    genome = chrompack.genome;
    randompos = randi(numel(genome));
    genome(randompos) = not(genome(randompos));
    mutated_chrompack = Chrompack(genome);
else
    mutated_chrompack = chrompack;
end
Population.m

```matlab
classdef Population < handle

  properties
  pop
  popsize = 0;
  bestpopindividual
  bestfitness = 0;
  meanfitness = 0;
  fitness_mode_opts

  end

  methods

  %after checking an Individual is given, adds it to the population
  function addindividual(obj, indiv)
    if not(isa(indiv, 'Individual'))
      error('addindividual method can only add Individual objects to Population.pop')
    end
    obj.pop=[obj.pop, indiv]; %add indiv to pop
    obj.popsize = obj.popsize + 1; %increment by 1 popsize
    %fitness(obj.pop);  %fitness can be calculated upon
    %addition, sacrificing speed
  end

  function obj=Population(Individual_array, fitness_mode_opts)
    if nargin > 0
      if isa(Individual_array(1), 'Individual')
        obj.pop=Individual_array;
      else
        error('Population constructor can only get Individual arrays as input')
      end
    end
    obj.popsize = obj.popsize + length(Individual_array);
    obj.fitness_mode_opts = fitness_mode_opts;
    %fitness(obj);
  end

  end

  %we calculate the the fitness of the entire population and save the
  %-best individual, and its fitness. We also calculate the mean
  %-fitness and the number of individuals

  function fitvector = fitness(obj)
    %calculate fitness only if not previously calculated
    [maxf, maxindex]=max(fitvector(obj.fitness_mode_opts), obj.pop);
    obj.meanfitness=sum(fitvector)/obj.popsize;
    if obj.bestfitness < maxf
      obj.bestfitness=maxf;
      obj.bestpopindividual=obj.pop(maxindex);
    end
    end
end
```
```matlab
% randpop.m

function pop = randpop(C, popsize, mode)
% RANDPOP generate random population from a given individual example
% Input:
% - C ------ binary code, as an example of the individuals to generate
% - popsize ---- the number of individuals to be created
% Output:
% - pop_of_individuals ----> randomly initialized population

% we extract examples size
[nchrom, chromlength] = size(C);

% we create a matrix of random genomes, concatenated one next to the other
% using mat2cell, we slice this concatenation of genomes to cells with one
% genome each

cell_of_genomes = mat2cell(randi([0,1], nchrom, chromlength * popsize), nchrom, ones(1, popsize) * chromlength);

% convert genomes to chromosomes, and chromosomes to individuals
cell_of_individuals = (cellfun(@(x) Individual(Chrompack(x)), cell_of_genomes, 'UniformOutput', false));

% convert cell array to normal array
array_of_individuals = {cell_of_individuals{:}};

pop = Population(array_of_individuals, mode);
end
```
%Selection with sigma trunc. and SRSWR
%This is how sigma truncation works:
% A) \( \text{fit} = \text{fit} - \{\text{mean(fit)} - c \times \text{std deviation(fit)}\} \)
% B) negative values set to 0
% stochastic remainder sampling without replacement
% (A) Expected individual count values calculated, and integer values
% calculated and assigned
% (B) Treat fractional parts as biased coin tosses until population is
% full

fitness = fit (population);  %we calculate fitness
fitness = fitness + normrnd(0,0.01,1,length(fitness));  %we add small noise to ensure non
uniformity
fitness = sigmatruncation(fitness,population.meanfitness);  % apply sigma truncation
%SRS WR selection method
choicector = fitness.*population.popsiz / sum(fitness);
selectedquantity = floor(choicector);  %calculate the integer parts
remainder = choicector - selectedquantity;  %calculate non-integer parts
sumselectedquantity = sum(selectedquantity);  %calculate sum of integer parts
if sumselectedquantity > population.popsiz  % check if new pop will be too big
e rror(’new population popsiz will exceed pop popsiz’)
end

P = remainder/sum(remainder);  %we make the sum of remainders be 1, thus calculate the
probabilities
C = cumsum(P);  %we calculate the cumulative probabilities
for ind=1:(population.popsiz - sumselectedquantity)  %iterate until population full
randomposition=(1+sum(C(end)*rand>C));  % + 1 on a random position to selectedquantity
selectedquantity(randomposition) = selectedquantity(randomposition) + 1;
end
if sum(selectedquantity) ~= population.popsiz
%disp(selectedquantity)
%disp(population.popsiz)
%check if poppopsiz == selectedquantity
e rror(’pop popsiz and selectedquantity mismatch’)
end

matingpool=repelem(population.pop,selectedquantity);

function [scalledfitvector] = sigmatruncation(fitvect,meanfitness)
c=2;
scaledfitvector = fitvect - (meanfitness - c*std(fitvect));
scaledfitvector(scaledfitvector<0) = 0;
end
A.1.2 Utility Functions

Gbundle.m

```matlab
classdef Gbundle < handle
    %MATEDPOOLINFO an auxiliary class to be able to do A(1,3) = Individual([chrompack1 chrompack2 chrompack3])
    
    %
    % % setgetnext(input)------> set chrompack array
    % % setgetnext()----------> get next chrompack
    % properties
    end

    methods (Static)

        function r = setgetnext(input)
            persistent sharedchrompacks;
            persistent index;
            if nargin
                index = index + 1;
            else
                index = 0;
            end
            sharedchrompacks = input;
            r = 0;
        end
    end
end
```
nextline.m

```matlab
function [text_line] = nextline(textfile)

% NEXTLINE read next line of a file, ignores comment lines, blank lines or
% lines starting with an space. terminate read wen END is found

persistent textID;
persistent control;
persistent per_textfile;

% If textID is empty —– open textfile to set textID
% —– set per_textfile as textfile
if isempty(textID)
    textID = fopen(textfile);
    per_textfile = textfile;
end

% detect changes in textfile
if per_textfile ~= textfile
    error('data file changed')
end

control = '%';
% read until non void line or non coment line found and only if no 'END' read
while (strcmp(control, '%') || strcmp(control, ' ') || strcmp(control, '
END')) && not(strcmp(control, 'END'))
    % read line
    text_line = fgetl(textID);
    % if line void, control = ''
    % else, control = text_line’s first character (to detect
    % comment lines)
    if isempty(text_line)
        control = '%';
    else
        control = text_line(1);
    end
end

% check if last line 'END', if so close file
if strcmp(text_line, 'END')
    disp('END line found. Text file will be closed. ')
    fclose(textID);
    clear textID
    clear control
    clear per_textfile
end
end
```
A.1. Matlab and Python Code for Max Search

shuffle.m

1 function | outarray | = shuffle ( inarray )
2 %SHUFFLE shuffles given array
3 index = randperm ( length ( inarray ) ) ;
4 outarray = inarray ( index ) ;
5
6 end
A.1.3 Python Code

bigint.py

```python
from decimal import *  # we need this module so that we can reduce the numerical error in code/decode functions
from math import floor
getcontext().prec = 100  # we set default precision to 100 (which is very high)

#return the binary code of an integer string (decimal integer)
def code(*args):
    # code(x)#
    # x --> integer
    # output --> list of 0 and 1 integers
    if len(args) == 1:
        # calculate binary string
        binary_string = bin(int(args[0]))
        # return an integer list
        return [int(x) for x in binary_string.split('b')[1]]

    # code(x, n)#
    # x --> integer
    # n --> integer [number of digits we want on binary code]
    # output --> list of 0 and 1 integers
    elif len(args) == 2:
        C = code(int(args[0]))
        # convert n to int to avoid problems
        n = int(args[1])
        # check if n too small, if not return C with zeroes at the begining
        if len(C) > n:
            print('len(C): ' + str(len(C)))
            print('n: ' + str(n))
            raise ValueError("len(C) > n, n is too small")
        else:
            dif = n - len(C)
            return [0] * dif + C

    # code(x, n, a, b)#
    # x --> float , x∈[a, b]
    # n --> integer ; number of digits we want on binary code
    # a --> float , lower bound of interval
    # b --> float , upper bound of interval
    # output --> list of 0 and 1 integers
```

elif len(args) == 4:
    getcontext().prec = int(floor(args[1]/4)) + 100
    x = Decimal(args[0])
    #convert n to int to avoid problems
    n = int(args[1])
    a = Decimal(args[2])
    b = Decimal(args[3])

    #set precision
    #10**n is a upper bound of (2**4)**n = 8**n
    getcontext().prec = int(floor(n/4)) + 100
    return str(code(int((Decimal(x) - Decimal(a)) / (Decimal(b) - Decimal(a)) * Decimal(2 **
        n - 1)) , n ))

# DECODE
# C ---> binary list
# output ---> integer
if len(args) == 1:
    C = args[0]
    n = len(C)
    # [1,1,0,1] = sum([1 2^0 , 2^1 , 2^2] = int(sum([2*(n - j - 1) for j in range(n) if C[j]]))

#### decode(C,n,a,b)#####
# C ---> binary list string , so that we can easily import an array from matlab
# n ---> integer , number of digits we have on binary code
# a ---> float , lower bound of interval
# b ---> float , upper bound of interval
# output ---> string(Decimal)
if len(args) == 4: #n = len(C) , however , we add it as an input argument for consistency with
code calls
    getcontext().prec = int(floor(args[1]/4)) + 100
    C = eval(str(args[0]))
    n = int(args[1])
    a = Decimal(args[2])
    b = Decimal(args[3])
    if not(len(C) == n):
        raise ValueError("len(C) and n mismatch ")
    x = Decimal(decode(C))
    # return value on string format , without scientific notation
    return format( Decimal(a + ( x / (2**n - 1) * (b - a)))) , 'f')
A.2 Matlab and Python Code for Polynomial Interpolation

Note: This code is the same on some modules.

A.2.1 Main Functions and Classes

main.m

```matlab
function [ population ] = main( txtfile )

% we want no warnings on badly conditioned polynomials
warning('off', 'MATLAB:polyfit:RepeatedPointsOrRescale')

% Read sample Individual’s genome
cl nextline
mode = nextline(txtfile);
fitness_opts = eval(nextline(txtfile));

% create fitness_mode_opts --> {‘mode’,%fitnessopts%}
fraction = {mode,fitness_opts{:}};

C_data = eval(nextline(txtfile));
popsize = eval(nextline(txtfile));
imax = eval(nextline(txtfile));
a = fitness_mode_opts{2};
b = fitness_mode_opts{3};
f = fitness_mode_opts{4};
n = fitness_mode_opts{5};

% Initialize population with randpop, and write mode_fitness_opts on popultion
population = randpop(C_data,popsize,fraction);

hold all

% we create two points to name the legends, make room for legends, and
% preallocate x axis
legend(’Max. fitness’, ’Mean fitness’, ’Location’, ’southeast’)
for i = 1:imax
  matingpool = select_ga(population);
  matedpool = crossover(matingpool);
  population.pop = mutate(matedpool);
  scatter(i,population.bestfitness,’.’, ’MarkerEdgeColor’, ’red’)
  scatter(i,population.meanfitness,’.’, ’MarkerEdgeColor’, ’blue’)
  drawnow
end

% plot function and found maxima
figure
% preallocate vector of positions of interpolating points
x = zeros(1,n);

% get best interpolating points
for i=1:n
  x(i) = decode_ab(population.bestpopindividual.chrompack.genome(1,:),size(C_data,2),a,b);
end
hold all

% plot f function
plot(a:(b-a)/1000:b,f(a:(b-a)/1000:b))
```
% plot interpolating polynomial
% get the polynomial coefficients
p = polyfit(x, f(x), n-1);
% define polynomial handle
pol = @(y) evalpol(p, y);

[~, error] = calculatefitness(population.bestpopindividual.chrompack.genome, fitness_mode_opts, true);

plot(x, f(x), 'go', 'markersize', 5);
text(0, 0, {'error: ', error});
plot(a:(b-a)/100000:b, pol(a:(b-a)/100000:b))

legend('f', 'interpolating points', 'ga interpolation', 'Location', 'southeast')
hold off
end
% Chapter A. Implemented Code

**calculatefitness.m**

```matlab
function [ fitness , error ] = calculatefitness( genome , fitness_mode_opts , calc_error )

if ~exist( 'calc_error' , 'var' )
  % third parameter does not exist, so default it to something
  calc_error = false;
end

mode = fitness_mode_opts{1};

switch mode
  case 'function_max'
    fitness = function_max( genome , fitness_mode_opts );
  case 'pol_interp'
    [fitness , error] = pol_interp( genome , fitness_mode_opts , calc_error );
end

%mode = 'function_max'
function [fitness] = function_max( genome , fitness_mode_opts )

%fitness_opts = { 'function_max' , a , b , @(x) f(x) }

f = fitness_mode_opts{4};
x = decode_ab( genome , length ( genome ) , fitness_mode_opts{2} , fitness_mode_opts{3}) ;
fitness = f(x);

%mode = 'pol_interp'
function [fitness, error] = pol_interp( genome , fitness_mode_opts , calc_error )

%fitness_opts = { 'pol_interp' , a,b,f,n };

% NEED TO ADD IF ROW OF GENOME REPEATED, FITNESS = 0

if rank( genome ) < size( genome , 1 )
    fitness = 0;
    return;
end

f = fitness_mode_opts{4};
n = fitness_mode_opts{5};
%preallocation
x = zeros(1,n);

%length of each chromosome
len = length( genome(1,:) );

for i=1:n
    x(i) = decode_ab( genome(i,:) , len , fitness_mode_opts{2} , fitness_mode_opts{3}) ;
end

%get the polynomial coefficients
p = polyfit( x,f(x) , n-1 );

%define polynomial handle
pol = @(y) evalpol( p , y );

f_integral = @(y) -exp(cos(y)).*(-1 + cos(y));

%define function to be integrated
```

h = @(y) abs(f(y) - pol(y));

% for fitness to be positive and positively oriented, we apply \(-\log(x + 0.00001)\)

pol_int = polyint(p);

if n > 2
    pol_int_value = abs(f_integral(fitness_mode_opts{3})-evalpol(pol_int,fitness_mode_opts{3})
    - f_integral(fitness_mode_opts{2}) + evalpol(pol_int,fitness_mode_opts{2}));
    fitness = pol_int_value;
    if calc_error == true;
        L1_norm = integral(h,fitness_mode_opts{2}, fitness_mode_opts{3}, 'RelTol',0,'AbsTol',1e-12);
        error = L1_norm;
    else
        error = 'unknown';
    end
else
    L1_norm = integral(h,fitness_mode_opts{2}, fitness_mode_opts{3}, 'RelTol',0,'AbsTol',1e-12);
    fitness = L1_norm + 0.00001;
    if calc_error == true;
        error = L1_norm;
    else
        error = 'unknown';
    end
end

fitness = 100 - log(fitness);
% display fitness

% we set fitness to 0 for negative fitness values
% if fitness < 0
%     fitness = 0;
% end
end
Chapter A. Implemented Code

Chrompack.m
The same file as the one defined in Max Search.

code_ab.m
The same file as the one defined in Max Search.

crossover.m
The same file as the one defined in Max Search.

data.txt

```matlab
%mode <--- tell the main file what mode will e be operating on
1 pol_interp

2 %Fitness function options. For 'pol_interp' mode, a cell containing:
3 3 {a, b, f, n} where n is the number of interpolating points
4 %
5 % 1 --> horizontal line
6 % 2 --> inclined line
7 % 3 --> parabola ...
8
9 {−4, 1, @(x) sin(x) .* cos(x) .* exp(cos(x)), 70} 
10
11 %The sample Individual’s genome (C) chrom length will be decided based on n 50
12 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
13
14 %Population size (popsize) 100
15 100
16
17 %Number of iterations (imax) 30
18 30
19
20 END
```
A.2. Matlab and Python Code for Polynomial Interpolation

**decode_ab.m**

The same file as the one defined in Max Search.

**Individual.m**

The same file as the one defined in Max Search.

**mutate.m**

The same file as the one defined in Max Search.

**Population.m**

The same file as the one defined in Max Search, except line 7 is set to:

$$\text{bestfitness} = -10^{30};$$

**randpop.m**

```matlab
function [ pop ] = randpop ( C , popsize , mode )
% RANDPOP generate random population from a given individual example
%Input
% C ---- binary code , as an example of the individuals to generate
% popsize ---- the number of individuals to be created
%Output
% pop_of_individuals ----> randomly initialized population

% we extract examples size
nchrom = mode{5};
chromlength = size(C,2);

% we create a matrix of random genomes, concatenated one next to the other
% using mat2cell, we slice this concatenation of genomes to cells with one
% genome each
cell_of_genomes = mat2cell ( randi ([0 , 1] , nchrom , chromlength * popsize) , nchrom , ones(1 , popsize)*chromlength );

% convert genomes to chromosomes, and chromosomes to individuals
cell_of_individuals = ( cellfun (@( x ) Individual ( Chrompack ( x ) ) , cell_of_genomes , 'UniformOutput' , false ) );

% convert cell array to normal array
array_of_individuals = { cell_of_individuals {} };

pop = Population ( array_of_individuals , mode );

end
```

**select.m**

The same file as the one defined in Max Search.

**Gbundle.m**

The same file as the one defined in Max Search.

**nextline.m**

The same file as the one defined in Max Search.

---

A.2.2 Utility Functions
Chapter A. Implemented Code

shuffle.m
The same file as the one defined in Max Search.

evalpol.m

```matlab
function [ k ] = evalpol( P, x )
k= zeros(1, length(x) );
for s=1:length(P)
    k=k+P(s) .* x .^ (length(P)−s);
end
```

A.2.3 Python Code

bigint.py
The same file as the one defined in Max Search.

A.3 Code of List Sorting

A.3.1 1st original approach

main.py

```python
import modules_ga as mo
import random
import copy
from matplotlib import pyplot as plt
from tqdm import tqdm
from statistics import stdev, mean

def consecutive_list_generator()
    for i1 in range(8):
        for i2 in range(7):
            option_list = list(range(8))
            option_list.remove(i1)
            i2 = option_list[i2]
            for i3 in range(6):
                option_list = list(range(8))
                option_list.remove(i1)
                option_list.remove(i2)
                i3 = option_list[i3]
                for i4 in range(5):
                    option_list = list(range(8))
                    option_list.remove(i1)
                    option_list.remove(i2)
                    option_list.remove(i3)
                    i4 = option_list[i4]
                    for i5 in range(4):
                        option_list = list(range(8))
                        option_list.remove(i1)
                        option_list.remove(i2)
                        option_list.remove(i3)
                        option_list.remove(i4)
                        i5 = option_list[i5]
                        for i6 in range(3):
                            option_list = list(range(8))
                            option_list.remove(i1)
                            option_list.remove(i2)
                            option_list.remove(i3)
                            option_list.remove(i4)
                            option_list.remove(i5)
                            i6 = option_list[i6]
```

for i7 in range(2):
    option_list = list(range(8))
    option_list.remove(i1)
    option_list.remove(i2)
    option_list.remove(i3)
    option_list.remove(i4)
    option_list.remove(i5)
    option_list.remove(i6)
    i7, i8 = option_list[i7], option_list[(i7 + 1) % 2]
    yield [i1, i2, i3, i4, i5, i6, i7, i8]

random.seed(4)
a = mo.broadcast_device(sr = '1w:ww:00wc:::i1c::01w01cc010wc10w1100c1w0*01*00w00:011
ccw00100B0007c1::1101101ww1wc0'::1:1:1:101101*01cc1:::c0111c::10c1w01:1011ccb1w01cww0:0011
w10c0w111ccc11::w0B:11w00w+wcc*'::10w00:100www:1w11c1:::c::110w:c10:1cww::0:10:wc1*::1w:*w1:1w:00
cw0cww0101*:c00c*00*001'::w0w0100011:wc0*c11000wwc01ww:1:1:1:Bc01c0B1clww1w101:wc0100cww0*wwc
w1w10:::0w11:1:11'::cww1w:0w::c1w1c01111w010:wc1cc0100c1c01ccw1:1:1::wc11c0www:wb*10::
c:c0111c1w00w0w1c1'::1:11101100:11:0c0c0c1:::110:c011101:01110:w:0cw100+:0c:1w:010:1')

fitness_value = mo.fitness_function(a,
number_of_shuffled_lists_measured=100,
comparision_rounds=20,
max_nullfit_on_10=10)

ed = []
i = 0
for sl in consecutive_list_generator():
    i += 1
    if i % 100 == 0:
        print(i)
        random.seed(5+i)
        sl_copy = copy.copy(sl)
        mo.sort_list_with_bd(a, sl_copy, 20)
ed.append((mo.measure_order(sl) - mo.measure_order(sl_copy))

    print(f'fitness value: {fitness_value}')
    print('-----')
    print(sl)
    print(sl_copy)
    print('-----')
    print(f'entropy difference: {mo.measure_order(sl) - mo.measure_order(sl_copy)}')

    print(ed)
    print(mean(ed))
plt.hist(ed, bins = [-2, -1.75, -1.5, -1.25, -1, -0.75, -0.5, -0.25, -0.01, 0.01, 0.25, 0.5, 0.75, 1.25, 1.5, 1.75, 2])
plt.show()
# Chapter A. Implemented Code

```python
import math
import random
from statistics import stdev, mean
import numpy as np
from tqdm import tqdm

# region finess and list modules

def argmax(iterable):
    return max(enumerate(iterable), key=lambda x: x[1])[0]

# measure list's entropy on logarithmic scale
def _measure_order(input_list):
    """calculate the level of disorder of the input list. It is measured on a logarithmic scale
    """
    sum_of_difference = sum([abs(input_list[i] - i) for i in range(len(input_list))])
    return math.log1p(sum_of_difference)

# binaryze and debinaryze
def _decimal(list_of_binary_strings):
    return sum((2**i for i in range(len(list_of_binary_strings)))
               if list_of_binary_strings[len(list_of_binary_strings) - i - 1] == '1')

# get binary list of strings '1' and '0' of size bin_len.
def _binary(decimal_number, bin_len):
    if decimal_number < 0:
        raise ValueError('Negative numbers have no binary')
    c = str(bin(decimal_number))
    c = c[2:]
    c = [char for char in c]
    return (bin_len - len(c)) * ['0'] + c

# return n sized shuffled list
def _generate_random_list(n):
    return_list = list(range(n))
    random.shuffle(return_list)
    return return_list

# interprets messages by updating the list to be ordered and updating the message list.
# Only one comparison per iteration
def _process_messages(shuffled_list, env_messages):
    max_messages = 8
    max_message_length = 8
    list_size = len(shuffled_list)
    available_comparation = True  # This variable makes sure only 1 comparation per iteration is done
    return_messages = []
    message_size = int(2*(math.log2(list_size)) + 2)
    piece_length = int((message_size - 2) / 2)
    for i in range(len(env_messages)):
        current_message = env_messages[i]
        if len(current_message) == message_size:
            # ignore message
            # compare elements (only once per iteration)
```
A.3. Code of List Sorting

```python
if env_messages[i][0:2] == ['1', '1'] or env_messages[i][0:2] == ['0', '0']:
    # advaliable_comparation = False
    first_piece = _decimal(env_messages[i][2:2 + piece_length])
    second_piece = _decimal(env_messages[i][2 + piece_length:2 + piece_length * 2])

    # if change is beneficial
    if (shuffled_list[first_piece] < shuffled_list[second_piece]
        and first_piece > second_piece)
        or (shuffled_list[first_piece] > shuffled_list[second_piece]
            and first_piece < second_piece):
        # if first piece < second piece, just change the prefix
        if first_piece < second_piece:
            env_messages.insert(0, ['1', '1'] + env_messages[i][2:])
            env_messages.pop(i + 1)  # delete compared message
        # else, swap places and change prefix
        else:
            env_messages.insert(0, ['1', '1'] +
                env_messages[i][2 + piece_length:2 + piece_length * 2] +
                env_messages[i][2:2 + piece_length])
            env_messages.pop(i + 1)

        # if change is NOT beneficial
        else:
            if first_piece < second_piece:
                env_messages.insert(0, ['0', '0'] + env_messages[i][2:])
                env_messages.pop(i + 1)
            else:
                env_messages.insert(0, ['0', '0'] +
                    env_messages[i][2 + piece_length:2 + piece_length * 2] +
                    env_messages[i][2:2 + piece_length])
                env_messages.pop(i + 1)

    # swap pieces
    elif env_messages[i][0:2] == ['1', '0']:
        index0 = _decimal(env_messages[i][2:2 + piece_length])
        index1 = _decimal(env_messages[i][2 + piece_length:2 + 2 * (piece_length)])
        index0, index1 = sorted([index0, index1])[0], sorted([index0, index1])[1]
        shuffled_list[:] = (shuffled_list[index1:] +
            shuffled_list[index0:index1] +
            shuffled_list[:index0])

    # swap elements
    elif env_messages[i][0:2] == ['0', '1']:
        index0 = _decimal(env_messages[i][2:2 + piece_length])
        index1 = _decimal(env_messages[i][2 + piece_length:2 + 2 * (piece_length)])
        if index0 != index1:
            shuffled_list[index0], shuffled_list[index1] = shuffled_list[index1],
            shuffled_list[index0]

        else:
            return_messages.append(env_messages[i])
    env_messages = env_messages[0:max_messages]
for i in range(len(env_messages)):
    env_messages[i] = env_messages[i][0:max_message_length]
return return_messages
```

# sorts list with broadcast_device

def sort_list_with_bd(b_device, shuffled_list, max_iterations):
    i = 0
    env_messages = [['1', '1', '0', '0', '0', '1', '1', '1']]  # process_messages(shuffled_list, env_messages)
    while i < max_iterations and env_messages:
        i += 1
        process_broadcast_device(b_device, env_messages)
        _process_messages(shuffled_list, env_messages)
```

# return broadcast device's fitness on a single list
def fitness_function(b_device,
                      number_of_shuffled_lists_measured,
                      comparison_rounds=150,
                      max_nullfit_on_10=10,
                      fast_0_fitness=False,
                      list_size=8)
    
    if number_of_shuffled_lists_measured != 1:
        number_of_nullfit = 0
        fit_list = []
        for i in range(number_of_shuffled_lists_measured):
            #print each individual once
            #if i == 0:
            #print(b_device)
            fit = fitness_function(b_device,
                        number_of_shuffled_lists_measured = 1,
                        max_nullfit_on_10 = max_nullfit_on_10,
                        comparison_rounds = comparison_rounds,
                        fast_0_fitness = fast_0_fitness,
                        list_size = list_size)
            fit_list.append(fit)
    
    if fit == 0:
        # we get rid of bd if no response in first iteration
        if i == 1 and fast_0_fitness:
            return 0.0
        number_of_nullfit += 1
        # we get rid of bd if too many null fitness on first 10 iterations
        if number_of_nullfit == max_nullfit_on_10 and i < 10:
            return 0.0
    
    if i == number_of_shuffled_lists_measured - 1:
        # percentage of improved lists - 0.5
        #return (sum((1 for i in fit_list if i > 0.00001)) / number_of_shuffled_lists_measured) - 0.5
        # the mean of improvement
        return mean(fit_list)
    
    shuffled_list = _generate_random_list(list_size)
    shuffled_entropy = _measure_order(shuffled_list)
    sort_list_with_bd(b_device, shuffled_list, comparison_rounds)
    return (shuffled_entropy - _measure_order(shuffled_list))

# endregion

# region genetic algorithm operators

# initializes bd of length bd_length and initializes random characters based on given
#prob_dist = [0.1, 1, *, :, w, b, B, 1, c]
def initialize_random_bd(bd_length, prob_dist):
    random.seed()
    return broadcast_device(sr = [ _get_char(prob_dist) for i in range(bd_length) ])}

# auxiliary function for initialize_random_bd
def _get_char(prob_dist, possible_chars = ('0', '1', '*', ':', 'w', 'b', 'B', 'l', 'c')):
    return random.choices(possible_chars, weights = prob_dist, k=1)[0]

# initializes random bd with no null fitness
def initialize_random_bd_no_0_fitness(bd_length,
    comparison_rounds,
    number_of_shuffled_lists_measured,
    fast_0_fitness,
    max_nullfit_on_10
):
prob_dist = (0.25, 0.25, 0.05, 0.15, 0.15, 0.01, 0.01, 0.01, 0.12)

bd = initialize_random_bd(bd_length=bd_length, prob_dist=prob_dist)
fitness = fitness_function(bd, number_of_shuffled_lists_measured=
number_of_shuffled_lists_measured,
    fast_0_fitness=fast_0_fitness, comparision_rounds=comparision_rounds,
    max_nullfit_on_10=max_nullfit_on_10)

it = 0
while fitness == 0:
    it += 1
    if it % 50 == 0:
        print(str(it) + ' iterations without bd initialization')
    bd = initialize_random_bd(bd_length=bd_length, prob_dist=prob_dist)
    fitness = fitness_function(bd, number_of_shuffled_lists_measured=
number_of_shuffled_lists_measured,
    fast_0_fitness=fast_0_fitness, comparision_rounds=comparision_rounds,
    max_nullfit_on_10=max_nullfit_on_10)

return bd

# get bd list of fitness. If fitness on first list is 0, set bd's fitness to 0
# without further operations
def _bd_list_fitness(bd_list, number_of_lists_sampled, comparision_rounds, max_nullfit_on_10):
    fitness_list = []
    for i in range(len(bd_list)):
        random.seed(4)
        fitness_list.append(fitness_function(b_device=bd_list[i],
            number_of_shuffled_lists_measured=number_of_lists_sampled,
            comparision_rounds=comparision_rounds,
            max_nullfit_on_10=max_nullfit_on_10))

    return fitness_list

# scale fitness with sigma truncation
def _sigma_truncation(list_of_fitness):
    c = 2
    sigma = stdev(list_of_fitness)
    mu = mean(list_of_fitness)
    return [(list_of_fitness[i] - (mu - c*sigma)) for i in range(len(list_of_fitness))]

# returns list with number of offspring using roulette wheel selection
def _roulette_selection(weights):
    """performs weighted selection or roulette wheel selection on a list
    and returns the index selected from the list""
    sorted_indexed_weights = sorted(enumerate(weights));
    indices, sorted_weights = zip(*sorted_indexed_weights);
    # calculate the cumulative probability
    tot_sum = sum(sorted_weights)
    if tot_sum == 0:
        prob = [1/len(sorted_indexed_weights) for i in range(len(sorted_indexed_weights))]
    else:
        prob = [x / tot_sum for x in sorted_weights]
        cum_prob = np.cumsum(prob)
        # select a random a number in the range [0,1]
        random_num = random.random()
        for index_value, cum_prob_value in zip(indices, cum_prob):
            if random_num < cum_prob_value:
                return index_value

# mates using two point crossover
def crossover(mate0, mate1):
sr0 = mate0.sr
sr1 = mate1.sr

mating_indexes = [random.randint(0, len(sr1) - 1), random.randint(0, len(sr1) - 1)]
mating_indexes.sort()

return |
  broadcast_device(
    sr = sr0[:mating_indexes[0]] +
    sr1[mating_indexes[0]:mating_indexes[1]] +
    sr0[mating_indexes[1]:],
  )
  broadcast_device(
    sr = sr1[:mating_indexes[0]] +
    sr0[mating_indexes[0]:mating_indexes[1]] +
    sr1[mating_indexes[1]:])
|

# mutation anywhere, taking into account prob mutation and prob_dist

def simple_mutation(bd, prob_mutation = 0.02, **kwargs):
  n_mutations = math.floor(len(bd.sr) * prob_mutation)
  if 'prob_dist' in kwargs:
    prob_dist = kwargs['prob_dist']
  else:
    prob_dist = (0.20, 0.20, 0.10, 0.10, 0.03, 0.02, 0.03, 0.03, 0.11)
    pass
  partial_sum = [sum(prob_dist[0:i]) for i in range(1, len(prob_dist) + 1)]

  while n_mutations > 0:
    n_mutations += -1
    bd.sr[random.randint(0, len(bd.sr) - 1)] = _get_char(partial_sum)

  # endregion

#region broadcast device modules

class broadcast_device:
  sr = []
  _active_sr = []
  type = -2

  # kwargs
  def __init__(self, **kwargs):
    ...
      
    #returns the list sr as a string
    def __str__(self):
      return str(''.join(self.sr))
#returns the length of sr

```python
def __len__(self):
    return len(self.sr)
```

#removes all quotes and quoted elements from sr and returns the list (it does not change sr)

```python
def _unquote(self):  # return unquoted list of chars
    if self.sr:
        if self.sr[0] == "c":
            return [self.sr[i] for i in range(1, len(self)) if self.sr[i] != "c" and self.sr[i] != "c"]
        else:
            return [self.sr[i] for i in range(0, len(self)) if i == 0 or (self.sr[i] != "c" and self.sr[i] != "c")]
    else:
        return []
```

#returns list of broadcast units, splitting them by *. It automatically uncomments sr

```python
def piecewise(self):
    self_unquoted = self._unquote()
    if not self_unquoted:
        return []
    unquoted_string = " " . join(self_unquoted)
    piecewise_strings = unquoted_string . split ("*")
    return [broadcast_device(sr=i) for i in piecewise_strings if i != ""]
```

#sets _active_sr and returns the length of the set list. If return the number of ‘:’ found

```python
def _decompose_and_set_active_sr(self):
    b = [] # in this list we save de indexes at which instances of ‘:’ were found
    count = 0 #number of ‘:’ instances found
    for i in range(len(self)):
        if self.sr[i] == "":" # when ‘:’ instance found, append it to b
            b . append (i)
            count += 1
    if count == 3: # when third instance of ‘:’ found, ignore rest
        self._active_sr = [self.sr[0:b[0]], self.sr[b[0]:b[1]], self.sr[b[1]:b[2]]]
    return count
    if count == 2: # if only two instances found, return appropriate list of sr pieces
        self._active_sr = [self.sr[0:b[0]], self.sr[b[0]:b[1]], self.sr[b[1]:]]
    return count
    elif count == 1: # if only one instance of ‘:’ found, return appropriate list
        self._active_sr = [self.sr[0:b[0]], self.sr[b[0]:]]
    return count
    return 0 #if no ‘:’ found, return 0
```

#set type, and while doing so modify self.sr to try to convert it to a unit
#it also sets the active string representation

```python
def set_type(self):
    if self.type == -2:
        if not self.sr:
            self.type = 0
        return
    ### we need to make sure c is not on self.sr before setting type ###
    #self.sr = self.piecewise()[0].sr  # Make sure that the bc device is a bc unit, or just take the first bu
    count = self._decompose_and_set_active_sr()
    if count == 0:
        self.type = 0
    elif count == 1:  # it could be type 1 bu
        if self._active_sr[0] and self._active_sr[1]:  #[1 , 1] --> type 1
            self.type = 1
        else:  # in this case, we have found our two valid ‘:’, it could be type 2,3,4
            if not self._active_sr[0]:  #if first component is empty [ - , 1 , 1 ]
                if not self._active_sr[1] and self._active_sr[2]:  # # 2nd and 3rd components are non empty --> type 2 [ - , 1 , 1 ]
                    self.type = 2
                else:
                    if first component is non empty [ 1 , - , 1 ]
                        if not self._active_sr[1]:  #if second component empty [ 1 , - , ? ]
                            # # 3rd component is empty --> type 3 [ 1 , - , ? ]
```
if self._active_sr[2]: # if third component non-empty [1, -1, 1]
    self.type = 3
elif self._active_sr[2]: #[1,1,1]
    self.type = 4
if self.type == -2:
    self.type = 0 #when no other type mach, set type to 0

# splits the bd into b units, activates their ASR and calls _process_broadcast_units
def process_broadcast_device(bd, env_mes_list, max_mes = 5, max_message_len = 8):
    bu_list = bd.piecewise()
    for bu in bu_list:
        bu.set_type()
        _process_broadcast_units(bu_list, env_mes_list, max_mes, max_message_len)

def _process_broadcast_units(array_of_active_units, env_mes_list, max_mes, max_message_len):
    # maxMes maximum number of messages to be outputted
    new_env_mes_list = []
    # we first process type 4 units
    t4_ind = [i for i in range(len(array_of_active_units)) if array_of_active_units[i].type == 4] # select type 4 units
    for current_bu_ind in t4_ind:
        # process type 4 units in order
        ASR = array_of_active_units[current_bu_ind].active_sr # get the bu's active string representation
        if not ASR[0] or not ASR[1] or not ASR[2]:
            continue
        i, replay_from_bu = 0, False
        while i < len(env_mes_list) and not replay_from_bu:
            b_B_l = [(), (), ()]
            if _match4_I1(ASR[0], env_mes_list[i], b_B_l):
                j = 0
                while j < len(env_mes_list) and not replay_from_bu:
                    if _match4_I2(ASR[1], env_mes_list[j], b_B_l):
                        replay_from_bu = True
                    j += 1
            if replay_from_bu:
                _write_reply(ASR[2], env_mes_list, b_B_l)
            i += 1
        i = 1
        if replay_from_bu:
            _write_reply(ASR[2], env_mes_list, b_B_l)
        if len(env_mes_list) > max_mes:
            del env_mes_list[-max_mes:]
    # we then process type 1 .
    t12_ind = [i for i in range(len(array_of_active_units))]
    for current_bu_ind in t12_ind:
        if array_of_active_units[current_bu_ind].type in {1,2}:
            ASR = array_of_active_units[current_bu_ind].active_sr # get the bu's active string representation
            i, replay_from_bu = 0, False
            b_B_l = [(), (), ()]
            while i < len(env_mes_list) and not replay_from_bu:
                if _match1_I1(ASR[0], env_mes_list[i], b_B_l):
                    replay_from_bu = True
                _write_reply(ASR[1], env_mes_list, b_B_l)
                i += 1
            if not replay_from_bu:
A.3. Code of List Sorting

```python
b_B_l = [[] , [] , []]
_write_reply(ASR[2], env_mes_list, b_B_l)

# shorten too lengthy messages and select only first max_mes messages
while len(env_mes_list) > max_mes:
    env_mes_list.pop()
for i in range(len(env_mes_list)):
    while len(env_mes_list[i]) > max_message_len:
        env_mes_list[i].pop()

# return True if match. Update b_B_l in that case. Else, return False and flush b_B_l
# kwargs only for internal recursive calls
def _match4_I1(ASR0, current_mes, b_B_l, **kwargs):
    # The case where 'b' or 'B' is at the first position, do_not_validate = False
    # In any other call, (recursive), do_not_validate = True
    # We only validate if it's the non recursive call of _match4_I1
    if 'do_not_validate' in kwargs:
        do_not_validate = kwargs['do_not_validate']
    else:
        do_not_validate = False
    if not do_not_validate:
        if len(ASR0) == 0:
            return False
        validated_ASR0 = _validate_ASRi(ASR0)
    else:
        validated_ASR0 = ASR0

    # len validated_ASR0 > 1 cases
    # comparing message and bu length, where to start reading env_mes,
    # if bu[0] = 'b' or 'B' i + i_0 is the first one to read
    i_0 = len(current_mes) - len(validated_ASR0)
    i = 1
    if validated_ASR0[0] == 'b':
        b_B_l[0] = current_mes[0:i_0+1]
    else: # validated_ASR0[0] = 'B'
        b_B_l[1] = current_mes[0:i_0+1]
    while i + i_0 < len(current_mes) and _match_char(validated_ASR0[i], current_mes[i+i_0], b_B_l):
        i = i+1
    if i + i_0 == len(current_mes):
        return True
    else:
        b_B_l[0], b_B_l[1], b_B_l[2] = [], [], [] #we do it this way to change b_B_l and not
        # the object it references on this particular function
        return False
elif validated_ASR0[-1] in {'b', 'B'}:
```
# Move 'b' or 'B' to first position, change current_message in similar fashion, and solve recursively

```python
return _match4_I1(
    validated_ASR0[1][len(validated_ASR0) - 1] + validated_ASR0[1:].len(validated_ASR0) - 1],
    current_mes[1][len(validated_ASR0) - 1:] + current_mes[0:].len(validated_ASR0) - 1],
    b_B_l,
    do_not_validate = True
)
```

# 'w' at the first or last position only processed as 'multiple' wild cards if no 'b' or 'B' found at first and last position. Else, 'w'

```python
elif validated_ASR0[0] == 'w':
    # Recursive call for validated_ASR0[0] = 'b': 'b' saved in temporal storage
    # and 'w' replaced with 'b'. After recursive execution:
    # → if match True, restore original 'b' on b_B_l
    # → if match False, b_B_l is reset by recursive call
    temp_b_stored = b_B_l[0]
    b_B_l[0] = []
    validated_ASR0[0] = 'b'
    if _match4_I1(validated_ASR0, current_message, b_B_l, do_not_validate = True):
        b_B_l[0] = temp_b_stored
        return True
    else:
        return False

elif validated_ASR0[-1] == 'w':
    # the same as last elif, but considering the last position instead
    #temp_b_stored = b_B_l[0]
    b_B_l[0] = []
    validated_ASR0[-1] = 'b'
    if _match4_I1(validated_ASR0, current_message, b_B_l, do_not_validate = True):
        b_B_l[0] = temp_b_stored
        return True
    else:
        return False
else:
    if len(validated_ASR0) != len(current_message):
        b_B_l[0], b_B_l[1], b_B_l[2] = [], [], []
    return False
else:
    for i in range(len(current_message)):
        if not _match_char(validated_ASR0[i], current_message[i], b_B_l):
            b_B_l[0], b_B_l[1], b_B_l[2] = [], [], []
        return False
    return True
```

# to be called after _match4_I1. It works the same, but in case of no match, b_B_l
# remains unchanged.

```python
def _match4_I2(ASR1, current_mes, b_B_l, **kwargs):
    backup_b_B_l = [b_B_l[i] for i in range(len(b_B_l))]
    if not ASR1:
        #if not element found in ASR1
        return False
    if ASR1[0] == 'b':
        # check for duplicate 'b', 'B' and 'l' in b_B_l and ASR1
        if b_B_l[0] == 'b':
            while ASR1 and ASR1[0] == 'b':  # while ASR1 not empty and it's first element is 'b'
                ASR1.pop(0)
            if b_B_l[1] == 'B':
                while ASR1 and ASR1[0] == 'B':  # while ASR1 not empty and it's first element is 'B'
                    ASR1.pop(0)
        if not ASR1:  # if not element found in ASR1
            return False
    elif ASR1[0] == 'B':
        # while ASR1 not empty and it's first element is 'B'
        ASR1.pop(0)
        if b_B_l[1] == 'B':
            while ASR1 and ASR1[0] == 'B':  # while ASR1 not empty and it's first element is 'B'
                ASR1.pop(0)
        if not ASR1:  # if not element found in ASR1
            return False
```
A.3. Code of List Sorting

```python
return False

# we do the same with the last element
if ASR1[-1] == 'b':
    if b_B_l[0]:
        while ASR1 and ASR1[-1] == 'b':
            ASR1.pop()
elif ASR1[-1] == 'B':
    if b_B_l[1]:
        while ASR1 and ASR1[-1] == 'B':
            ASR1.pop()

if not ASR1:  # if not element found in ASR1
    return False

# we handle the case when l is already used and finally, we validate.
if b_B_l[2]:  # if b_B_l['l'] not empty,
    ASR1.insert(1, 'l')
validated_ASR1 = _validate_ASRi(ASR1)
if 'l' in validated_ASR1:
    validated_ASR1.remove('l')

if not 'validated_ASR1' in locals():
    validated_ASR1 = _validate_ASRi(ASR1)
if _match4_I1(validated_ASR1, current_mes, backup_b_B_l):
    b_B_l[0], b_B_l[1], b_B_l[2] = backup_b_B_l[0], backup_b_B_l[1], backup_b_B_l[2]
    return True
else:
    return False

# calls _match4_I1
def _match1_I1(ASR0, current_mes, b_B_l, **kwargs):
    return _match4_I1(ASR0, current_mes, b_B_l, **kwargs)

# return ASRi with valid b, B and l instances
# return error if ASRi is empty
def _validate_ASRi(ASRi):
    if len(ASRi) > 1:
        return_ASRi = []
        b_0 = ASRi[0] == 'b'
        B_0 = ASRi[0] == 'B'
        first_bB_found = b_0 or B_0
        first_l_found = False
        if first_bB_found:  # if b or B found 2 at ASRi[0], append it to return_ASRi
            return_ASRi.append(ASRi[0])
        for i in range(len(ASRi)):
            if not first_bB_found and ASRi[i] in {'b', 'B'}:  # if it's the first instance of 'b' or 'B', but not i=0
                if i == len(ASRi)-1:  # if the first bB instance is the last element, append it to the list
                    return_ASRi.append(ASRi[i])
            if b_0 or B_0:
                first_bB_found = True
            elif not first_l_found and ASRi[i] == 'l':
                return_ASRi.append(ASRi[i])
            if i in range(len(ASRi)):
                if ASRi[i] not in {'b', 'B', 'l'}:
                    raise ValueError('Processed ASRi cannot be empty')
        return return_ASRi
    else:
        raise ValueError('Processed ASRi cannot be empty')

    return return_ASRi
```

# returns False if no match occurs,
# returns True if character or wild card match, or one time 'l' match
# updates b_B_l to include found l (if found)
def _match_char(char_bu, char_current_mes, b_B_l):
    if char_bu == char_current_mes:
        return True
    elif char_bu in {'0', '1'}:
        return False
    elif char_bu == 'w':
```
return True
elif char_bu == '1':
    # if b_B_l[2] is not empty when trying to write '1'
    if b_B_l[2] == [char_current_mes]:
        b_B_l[2] = [char_current_mes]
    return True
else:
    raise ValueError('b_B_l[2] is not empty when trying to write '1'

# write reply considering broadcast unit's last component
def _write_reply(answer_ASRi, env_mes_list, b_B_l, max_mes_len=9):
    reply_mes = []
    for j in range(len(answer_ASRi)):
        if len(reply_mes) > max_mes_len:
            del reply_mes[:max_mes_len]
        env_mes_list.insert(0, reply_mes)
    return
    # first three cases, for when b, B, or l is written (only once)
    # last case, write 0 or 1
    if answer_ASRi[j] == 'b':
        if b_B_l[0] != []:
            reply_mes.extend(b_B_l[0])
        b_B_l[0] = []
    elif answer_ASRi[j] == 'B':
        if b_B_l[1] != []:
            reply_mes.extend(b_B_l[1])
        b_B_l[1] = []
    elif answer_ASRi[j] == 'l':
        if b_B_l[2] != []:
            reply_mes.extend(b_B_l[2])
        b_B_l[2] = []
    elif answer_ASRi[j] in {'1', '0'}:
        reply_mes.append(answer_ASRi[j])
    elif answer_ASRi[j] == 'w':
        pass
    else:
        raise ValueError('non valid character encountered, \{\}'.format(answer_ASRi[j]))
    env_mes_list.insert(0, reply_mes)

#endregion

def fit(max_iterations = 20, popsize = 20, bd_len = 500):
    population = [initialize_random_bd_no_0_fitness(bd_len,20,15,True,4) for i in tqdm(range(popsize))]
    print('start')
    max_fitness = -1e10
    current_max_fitness = -1e10
    for i in range(max_iterations):
        fitness_list = _bd_list_fitness(bd_list=population,
                                         number_of_lists_sampled=100,
                                         comparison_rounds=20,
                                         max_nullfit_on_10=10)
        current_max_fitness = max(fitness_list)
        current_max_index = argmax(fitness_list)

        # population reinitialization
        if current_max_fitness == 0: #if null fitness on all devices, reinitialize pop
            print('reinitializing -')
            population = [initialize_random_bd_no_0_fitness(bd_len) for i in tqdm(range(popsize),
                                                        desc='initalizing')] #
            print('iteration:{}'.format(i))
            current_max_fitness = 0
            # continue
        if current_max_fitness > max_fitness:
            best_bd = population[current_max_index]
            max_fitness = max(fitness_list)

        fitness_list = _sigma_truncation(fitness_list)
        newpop = list(}

Chapter A. Implemented Code
73
for j in range(math.floor(popsize/2)):
    bd_ind0 = _roulette_selection(fitness_list)
    bd_ind1 = _roulette_selection(fitness_list)
    children_bd = crossover(population[bd_ind0], population[bd_ind1])
    simple_mutation(children_bd[0], 0.02)
    simple_mutation(children_bd[1], 0.02)
    newpop.extend(children_bd)
population = newpop
print(f'\n−−−−−
{i/max_iterations}% done\nmax_fit:{max_fitness}\ncurrent_max_fit:{current_max_fitness}\n
−−−−−
best bd: {str(best_bd)}\n−−−−−

import cProfile
import re
cProfile.run('fit', 'restats')
import pstats
p = pstats.Stats('restats')
p.strip_dirs()
p.sort_stats('cumulative').print_stats(10)

...
A.3.2 1st restricted approach, with r as new operator

main.py

```python
import modules_ga as mo
import random
import copy
from matplotlib import pyplot as plt
from statistics import mean

def consecutive_list_generator():
    for i1 in range(8):
        for i2 in range(7):
            option_list = list(range(8))
            option_list.remove(i1)
            i2 = option_list[i2]
            for i3 in range(6):
                option_list = list(range(8))
                option_list.remove(i1)
                option_list.remove(i2)
                i3 = option_list[i3]
                for i4 in range(5):
                    option_list = list(range(8))
                    option_list.remove(i1)
                    option_list.remove(i2)
                    option_list.remove(i3)
                    i4 = option_list[i4]
                    for i5 in range(4):
                        option_list = list(range(8))
                        option_list.remove(i1)
                        option_list.remove(i2)
                        option_list.remove(i3)
                        option_list.remove(i4)
                        i5 = option_list[i5]
                        for i6 in range(3):
                            option_list = list(range(8))
                            option_list.remove(i1)
                            option_list.remove(i2)
                            option_list.remove(i3)
                            option_list.remove(i4)
                            option_list.remove(i5)
                            i6 = option_list[i6]
                            for i7 in range(2):
                                option_list = list(range(8))
                                option_list.remove(i1)
                                option_list.remove(i2)
                                option_list.remove(i3)
                                option_list.remove(i4)
                                option_list.remove(i5)
                                option_list.remove(i6)
                                i6, i8 = option_list[i7], option_list[(i7 + 1) % 8]
                                yield [i1, i2, i3, i4, i5, i6, i7, i8]

good_sr = '*00111111:01000000+wwwwwww:11rrrrrr'
sr = '*+lwllllll:11rrrr00rs+lwll0w11:10111010'
a = mo.broadcast_device(sr=sr)
shuffled_list = [6, 2, 3, 4, 1, 5, 7, 0]
mo.sort_list_with_bd(a, shuffled_list, 20)

print('--------------------')
ed = []
i = 0
for sl in consecutive_list_generator():
    i += 1
    if i % 100 == 0:
        print(i)
random.seed(5+i)
```

Chapter A. Implemented Code

75
sl_copy = copy.copy(sl)
mo.sort_list_with_bd(a, sl_copy, 20)
ed.append(mo._measure_order(sl) - mo._measure_order(sl_copy))

' ' ' 
print(f'fitness value: {fitness_value}')
print('------')
print(sl)
print(sl_copy)
print('------')
print(f'entropy difference: {mo._measure_order(sl) - mo._measure_order(sl_copy)}')
' ' '

print(ed)
# 0.14154995352778119
print(mean(ed))

plt.hist(ed, bins=[-2, -1.75, -1.5, -1.25, -1, -0.75, -0.5, -0.25, -0.01, 0.01, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2])
plt.title("Histogram of best BD's fitness values")
plt.show()

#mo.fit()
import math
import random
from statistics import stdev, mean
import numpy as np
from tqdm import tqdm
import multiprocessing.pool
import functools
from matplotlib import pyplot as plt
import time

#region tools

def argmax(iterable):
    return max(enumerate(iterable), key=lambda x: x[1])[0]

# https://stackoverflow.com/questions/492519/timeout-on-a-function-call

def timeout(max_timeout):
    """Timeout decorator, parameter in seconds."""

    def timeout_decorator(item):
        """Wrap the original function."""

        @functools.wraps(item)
        def func_wrapper(*args, **kwargs):
            """Closure for function."""

            pool = multiprocessing.pool.ThreadPool(processes=1)
            async_result = pool.apply_async(item, args, kwargs)
            # raise a TimeoutError if execution exceeds max_timeout
            return async_result.get(max_timeout)

        return func_wrapper

    return timeout_decorator

# dynamic plotting

def update_line(hl, new_data):
    hl.set_xdata(np.append(hl.get_xdata(), new_data))
    hl.set_ydata(np.append(hl.get_ydata(), new_data))
    plt.draw()

#endregion

# region fineness and list modules

# measure list’s entropy on logarithmic scale

def _measure_order(input_list):
    ""
    calculate the level of disorder of the input list. It is measured on a logarithmic scale
    :param input_list: the list to be measured
    :return: positive float
    ""

    sum_of_difference = sum((abs(input_list[i] - i) for i in range(len(input_list))))
    return math.log1p(sum_of_difference)

# binaryize and debinaryize

def _decimal(list_of_binary_strings):
    return sum((2 ** i for i in range(len(list_of_binary_strings))
                if list_of_binary_strings[len(list_of_binary_strings) - 1 - i] == '1'))

# get binary list of strings '1' and '0' of size bin_len.

def binary(decimal_number, bin_len):
    if decimal_number < 0:
A.3. Code of List Sorting

```python
raise ValueError('Negative numbers have no binary')
c = str(bin(decimal_number))
c = c[2:] # convert from '0101' to ['0', '1', '0', '1']
return (bin_len - len(c)) * ['0'] + c

# return n sized shuffled list
def _generate_random_list(n):
    return _list = list(range(n))
    random.shuffle(_return_list)
    return _return_list

# interprets messages by updating the list to be ordered and updating the message list.
# Only one comparation per iteration
# clean
def _process_messages(shuffled_list, env_messages):
    list_size = len(shuffled_list)
    available_comparation = True # This variable makes sure only 1 comparation per iteration is done
    piece_length = int(math.log2(list_size))
    message_size = int(piece_length * 2 + 2)
i = 0
while i < len(env_messages):
    if len(env_messages[i]) == message_size:
        # just 'delete' message
        if env_messages[i][0:2] == ['0', '0']:
            env_messages.pop(i)
i = i - 1
        # compare elements (only once per iteration)
else:
    elif env_messages[i][0:2] == ['1', '1'] and available_comparation:
        available_comparation = False
        # if first piece < second piece, just change the prefix
        first_piece = _decimal(env_messages[i][2:2 + piece_length])
        second_piece = _decimal(env_messages[i][2 + piece_length:2 + piece_length * 2])
        if:
            shuffled_list[first_piece] < shuffled_list[second_piece]
            and first_piece < second_piece
        else:
            shuffled_list[first_piece] > shuffled_list[second_piece]
            and first_piece < second_piece:

            env_messages.insert(0, ['0', '0', '0'] + env_messages[i][2:])
            env_messages.pop(i + 1)
else:
    # else, swap places and change prefix
    if env_messages[i][0:2] == ['0', '0'] +
        env_messages[i][2 + piece_length:2 + piece_length * 2] +
        env_messages[i][2:2 + piece_length])
        env_messages.pop(i + 1)

    # swap pieces
    else:
        index0 = _decimal(env_messages[i][2:2 + piece_length])
        index1 = _decimal(env_messages[i][2 + piece_length:2 + 2 * (piece_length)])
        index0, index1 = sorted([index0, index1])[0], sorted([index0, index1])[1]
        shuffled_list[:i] = (shuffled_list[index1:] +
            shuffled_list[index0:)index1]
        shuffled_list[i] = (shuffled_list[i] +
            shuffled_list[index0:)index0])
        env_messages.pop(i)
i = i - 1

    # swap elements
    else:
        index0 = _decimal(env_messages[i][2:2 + piece_length])
        index1 = _decimal(env_messages[i][2 + piece_length:2 + 2 * (piece_length)])
        if index0 < index1:
            shuffled_list[index0], shuffled_list[index1] = shuffled_list[index1],
                shuffled_list[index0]
```

Chapter A. Implemented Code

# sorts list with broadcast_device

```python
def sort_list_with_bd(b_device, shuffled_list, max_iterations):
    env_messages = [['1', '1', '0', '0', '1', '1', '1']]
    _process_messages(shuffled_list, env_messages)
    i = 0
    while i < max_iterations and env_messages:
        i = i + 1
        process_broadcast_device(b_device, env_messages)
    _process_messages(shuffled_list, env_messages)
```

# return broadcast device's fitness on a single list

```python
def fitness_function(b_device, number_of_shuffled_lists_measured = 100, max_nullfit_on_10 = 3, fast_0_fitness = False):
    if number_of_shuffled_lists_measured != 1:
        number_of_nullfit = 0
        fit_list = []
        for i in range(number_of_shuffled_lists_measured):
            fit = fitness_function(b_device, 1)
            fit_list.append(fit)
            if fit == 0:
                if i == 1 and fast_0_fitness: #we get rid of bd if no response in first iteration
                    return 0.0
                number_of_nullfit += 1
                if number_of_nullfit == max_nullfit_on_10 and i <= 10:
                    return -0.5
                if i == number_of_shuffled_lists_measured -1:
                    # percentage of improved lists - 0.5
                    return (sum((1 for i in fit_list if i > 0.00001)) / number_of_shuffled_lists_measured) - 0.5
                    # the mean of improvement
                    return mean(fit_list)
    list_size = 8
    max_iterations = 150
    shuffled_list = _generate_random_list(list_size)
    shuffled_entropy = _measure_order(shuffled_list)
    sort_list_with_bd(b_device, shuffled_list, max_iterations)
    return (shuffled_entropy - _measure_order(shuffled_list))
```

# endregion

# region genetic algorithm operators and initialization

```python
# initializes bd of length bd_length and initializes random characters based on given
# prob_dist = [0, 1, *, :, w, b, l, c,r]
def initialize_random_bd(bd_length, **kwargs):
    random.seed()
    if 'prob_dist' in kwargs:
        prob_dist = kwargs['prob_dist']
    else:
        prob_dist = ['0', '1', '*', ':', 'w', 'b', 'l', 'c', 'r']
        raise ValueError('No prob_dist introduced')
    pass
    if not 0.999 < sum(prob_dist) < 1.0001:
        raise ValueError('sum(prob_dist) = 1 is false')
    #partial_sum = [sum(prob_dist[0:i]) for i in range(1, len(prob_dist) + 1)]
    return broadcast_device(sr=
    ['*' if i % 18 == 0 else
     ':' if i % 18 == 9 else
     _get_char(prob_dist, ['w', '1']) if i % 18 > 9 else
     _get_char(prob_dist, ['r']) for i in range(bd_length)])
```
# auxiliary function for initialize_random_bd
# do_not_generate = list[str] reroll get_char if generated char in do

@do_not_generate = list[str], reroll get_char if generated char in do

def _get_char(prob_dist, do_not_generate=[]):
    random_number = random.random()
    index = 0
    index_selected = False
    while index < len(prob_dist) and not index_selected:
        if random_number < sum(prob_dist[0:index+1]):  # if undesirable char selected, call function recursively
            if ('0', '1', '*', ':', 'w', 'b', 'B', 'l', 'c', 'r')[index] in do_not_generate:
                return _get_char(prob_dist, do_not_generate=do_not_generate)
            else:
                return ('0', '1', '*', ':', 'w', 'b', 'B', 'l', 'c', 'r')[index]
        index = index + 1

# initializes random bd with no nule fitness
def initialize_random_bd_no_0_fitness(bd_length, prob_dist):
    bd = initialize_random_bd(bd_length, prob_dist=prob_dist)
    fitness = fitness_function(bd)
    it = 0
    while fitness < 0:
        it += 1
        if it % 500 == 0:
            print(str(it) + ' iterations without valid bd creation. ')
        del bd
    bd = initialize_random_bd(bd_length, prob_dist=prob_dist)
    fitness = fitness_function(bd, fast_0_fitness=True)
    return bd

# get bd list of fitness. If fitness on first list is 0, set bd's fitness to 0
# without further operations
def bd_list_fitness(bd_list):
    fitness_list = []
    state = random.getstate()
    for i in range(len(bd_list)):
        random.setstate(state)
        fitness_list.append(fitness_function(bd_list[i]))
    return fitness_list

# scale fitness with sigma truncation
def _sigma_truncation(list_of_fitness, **kwargs):
    if 'c' in kwargs:
        c = kwargs['c']
    else:
        c = 2
    sigma = stdev(list_of_fitness)
    mu = mean(list_of_fitness)
    return [(list_of_fitness[i] - (mu - c * sigma)) for i in range(len(list_of_fitness))]

# returns list with number of offspring using roulette wheel selection
def roulette_selection(weights):
    sorted_indexed_weights = sorted(enumerate(weights))
    indices, sorted_weights = zip(*sorted_indexed_weights)
    # calculate the cumulative probability
    tot_sum = sum(sorted_weights)
    if tot_sum == 0:
        prob = [1 / len(sorted_indexed_weights) for i in range(len(sorted_indexed_weights))]
    else:
        prob = [x / tot_sum for x in sorted_weights]
cum_prob = np.cumsum(prob)
# select a random a number in the range [0,1]
random_num = random.random()

for index_value, cum_prob_value in zip(indices, cum_prob):
    if random_num < cum_prob_value:
        return index_value

# mates using two point crossover
def crossover(mate0, mate1):
    sr0 = mate0.sr
    sr1 = mate1.sr

    mating_indexes = [random.randint(0, len(sr1) - 1), random.randint(0, len(sr1) - 1)]
    mating_indexes.sort()
    return 
    broadcast_device(
        sr=sr0[:mating_indexes[0]] + sr1[mating_indexes[0]:mating_indexes[1]] + sr0[mating_indexes[1]:],
        broadcast_device(
            sr=sr1[:mating_indexes[0]] + sr0[mating_indexes[0]:mating_indexes[1]] + sr1[mating_indexes[1]:])
    )

# simple mutation, without destroying ' ' and '*' instances and/or commenting them
# need to include non disruptant comments (not to comment ; and * instances)
def simple_respectful_mutation(bd, prob_mutation=0.02, **kwargs):
    n_mutations = len(bd.sr) * prob_mutation
    n_mutations = int(n_mutations) + int(
        n_mutations - int(n_mutations) > random.random()
    )

    if 'prob_dist' in kwargs:
        prob_dist = kwargs['prob_dist']
    else:
        raise ValueError('no prob_dist introduced')

    while n_mutations > 0:
        ran_num = random.randint(0, len(bd.sr) - 1)
        if bd.sr[ran_num] not in {'*', ':'}:
            if ran_num % 18 > 9:
                bd.sr[ran_num] = _get_char(prob_dist, ['w', 'l'])
            else:
                bd.sr[ran_num] = _get_char(prob_dist, ['r'])
        n_mutations -= 1

# endregion IN

# region broadcast device modules

class broadcast_device:
    sr = []
    _active_sr = []
    type = -2

    # kwags --> sr or string_representation: initializes with sr (it can be a string or a list)
def __init__(self, **kwargs):
    ...
A.3. Code of List Sorting

```python
:param kwarg: sr / string_representation: list or string to initialize broadcast

# set sr if input sr given
if 'string_representation' or 'sr' in kwarg: # check if sr is given
    if 'string_representation' in kwarg:
        input_sr = kwarg['string_representation']
    else:
        input_sr = kwarg['sr']
else:
    if isinstance(input_sr, str):
        self.sr = list(input_sr)
    elif isinstance(input_sr, list):
        self.sr = input_sr
    else:
        raise TypeError('unsupported type: {} for sr initialization.'.format(type(input_sr)))

# returns the list sr as a string
def __str__(self):
    return ' '.join(self.sr)

def __repr__(self):
    return str(self)

# returns the length of sr
def __len__(self):
    return len(self.sr)

# removes all quotes and quoted elements from sr and returns the list (it does not change sr)
def _unquote(self):
    if self.sr:
        return [self.sr[i] for i in range(1, len(self)) if self.sr[i] != 'c']

# returns list of broadcast units, splitting them by *. It automatically uncomments sr
def piecewise(self):
    self_unquoted = self._unquote()
    if not self_unquoted:
        return []
    unquoted_string = ' '.join(self_unquoted)
    piecewise_strings = unquoted_string.split('*')
    return [broadcast_device(sr=i) for i in piecewise_strings if i != '']

# sets _active_sr and returns the length of the set list. If return the number of ':' found
def _decompose_and_set_active_sr(self):
    b = []  # in this list we save de indexes at wich instances of ':' were found
    count = 0  # number of ':' instances found
    for i in range(len(self)):
        if self.sr[i] == ':':  # when ':' instance found, append it to b
            b.append(i)
        count += 1
        if count == 3:  # when third instance of ':' found, ignore rest
            self._active_sr = [self.sr[0:b[0]], self.sr[b[0] + 1:b[1]], self.sr[b[1] + 1:b[2]]]
            return count
        if count == 2:  # if only two instances found, return appropriate list of sr pieces
            self._active_sr = [self.sr[0:b[0]], self.sr[b[0] + 1:b[1]], self.sr[b[1] + 1:]]
            return count
        if count == 1:  # if only one instance of ':' found, return appropriate list
            self._active_sr = [self.sr[0:b[0]], self.sr[b[0] + 1:]]
            return count
    return 0  # if no ':' found, return 0

# set type, and while doing so modify self.sr to try to convert it to a unit
# it also sets the active string representation
def set_type(self):
    if self.type == -2:
        if not self.sr:
            ...
```
self.type = 0
return

# # we need to make sure c is not on self.sr before setting type #
# self.sr = self.piecewise()[0].sr # Make sure that the bc device is a bc unit, or just
take the first bu

count = self._decompose_and_set_active_sr()
if count == 0:
    self.type = 0:
elif count == 1:  # it could be type 1 bu
    if self._active_sr[0] and self._active_sr[1]:  # [1 , 1] --> type 1
        self.type = 1
else:  # in this case, we have found our two valid '':', it could be type 2,3,4
    if not self._active_sr[0]:  # if first component is empty [ - , ? , ? ]
        if self._active_sr[1] and self._active_sr[2]:  # 2nd and 3rd components are non empty --> type 2 [ - , 1 , 1]
            self.type = 2
    else:  # if first component is non empty [ 1 , ? , ? ]
        if not self._active_sr[1]:  # if second component empty [ 1 , - , ? ]
            if self._active_sr[2]:  # if third component non-empty [ 1 , - , 1]
                self.type = 3
        else:  # in this case, we have found our two valid '':', it could be type 2,3,4
            if self._active_sr[2]:  # [ 1 , 1 , 1]
                self.type = 4
        if self.type == -2:
            self.type = 0  # when no other type mach, set type to 0

def process_broadcast_device(bd, env_mes_list, max_mes=5):
    _process_broadcast_device_with_timeout(bd, env_mes_list, max_mes)

# splits the bd into b units, activates their ASR and calls _process_broadcast_units
# timeout creates memory leak bug
# @profile()
def _process_broadcast_device_with_timeout(bd, env_mes_list, max_mes):
    bu_list = bd.piecewise()
    for bu in bu_list:
        bu.set_type()
    _process_broadcast_units(bu_list, env_mes_list, max_mes)

# we first process type 4 units
..."""
t4_ind = [i for i in range(len(array_of_active_units))
    if array_of_active_units[i].type == 4]  # select type 4 units
for current_bu in t4_ind:
    # process type 4 units in order
    ASR = array_of_active_units[current_bu].active_sr  # get the bu’s active string
    representation
    if not ASR[0] or not ASR[1] or not ASR[2]:
        continue
    i, replay_from_bu = 0, False
    while i < len(env_mes_list) and not replay_from_bu:
        b_B_l = [[], [], []]
        if _match4_I1(ASR[0], env_mes_list[i], b_B_l):
            j = 0
            while j < len(env_mes_list) and not replay_from_bu:
                if _match4_I2(ASR[1], env_mes_list[j], b_B_l):
                    replay_from_bu = True
                j += 1
        i += 1
        if replay_from_bu:
            _write_reply(ASR[2], env_mes_list, b_B_l)
    while len(env_mes_list) > max_mes:
        env_mes_list.pop()
..."""

# we then process type 1 , 2 and 3 units
A.3. Code of List Sorting

```python
# A.3. Code of List Sorting

# Code snippet

t12_ind = [i for i in range(len(array_of_active_units))]
if array_of_active_units[i].type in {1, 2}:
    for current_bu_ind in t12_ind:
        if array_of_active_units[current_bu_ind].type == 1:  # for type 1 b units
            ASR = array_of_active_units[current_bu_ind].active_sr  # get the bu's active string representation
            i, replay_from_bu = 0, False
            while i < len(env_mes_list) and not replay_from_bu:
                b_B_l = [[]]  # Initialize
                if _match1_I1(ASR[0], env_mes_list[i], b_B_l):
                    replay_from_bu = True
                    _write_reply(ASR[1], new_env_mes_list, b_B_l)
                    i += 1
            else:
                if array_of_active_units[current_bu_ind].type == 2:  # for type 1 b units
                    ASR = array_of_active_units[current_bu_ind].active_sr  # get the bu's active string representation
                    i, replay_from_bu = 0, False
                    b_B_l = [[]]  # Initialize
                    while i < len(env_mes_list) and not replay_from_bu:
                        if _match1_I1(ASR[1], env_mes_list[i], b_B_l):
                            replay_from_bu = True
                            _write_reply(ASR[2], new_env_mes_list, b_B_l)
                            i += 1
            # shorten too lengthy messages and select only first max_mes messages
            env_mes_list[:] = new_env_mes_list[:]
            while len(env_mes_list) > max_mes:
                env_mes_list.pop()
        for i in range(len(env_mes_list)):
            while len(env_mes_list[i]) > max_message_len:
                env_mes_list[i].pop()
    # return True if match. Update b_B_l in that case. Else, return False and flush b_B_l
    # kwargs only for internal recursive calls
    def _match4_I1(ASR0, current_mes, b_B_l, **kwargs):
        # The case where 'b' or 'B' is at the first position, do_not_validate = False
        # In any other call, (recursive), do_not_validate = True
        # We only validate if it's the non recursive call of _match4_I1
        if 'do_not_validate' in kwargs:
            do_not_validate = kwargs['do_not_validate']
        else:
            do_not_validate = False
        if not do_not_validate:
            if len(ASR0) == 0:
                return False
            validated_ASR0, found_l_positions = _validate_ASRi(ASR0)
            else:
                validated_ASR0 = ASR0

        if validated_ASR0[0] in {'b', 'B'}:
            if len(validated_ASR0) > len(current_mes):  # if bu length too large, return False
                return False
            else:
                if len(validated_ASR0) == 0:
                    return False
                elif len(validated_ASR0) == 1:
                    if validated_ASR0[0] == 'b':
                        # return True if match. Update b_B_l in that case. Else, return False and flush b_B_l
```

b_B_l[0] = current_mes[:]
return True
else:
b_B_l[1] = current_mes[:]
return True

# len(validated_ASR0) > 1 cases
# comparing message and bu length, where to start reading env_mes,
# if bu[0] = 'b' or 'B', i + i_0 is the first one to read
i_0 = len(current_mes) - len(validated_ASR0)
i = 1
# update b_B_l
if validated_ASR0[0] == 'b':
b_B_l[0] = current_mes[0:i_0 + 1]
else:
b_B_l[1] = current_mes[0:i_0 + 1]

while i + i_0 < len(current_mes) and _match_char(validated_ASR0[i], current_mes[i + i_0], b_B_l):
i = i + 1
if i + i_0 == len(current_mes):
return True
else:
b_B_l[0], b_B_l[1], b_B_l[2] = [], [], [] # we do it this way to change b_B_l and not
# the object it references on this particular function
return False
e elif validated_ASR0[-1] in {'b', 'B'}:
# Move 'b' or 'B' to first position, change current_message in similar fashion, and solve recursively
return _match4_I1(
(validated_ASR0[len(validated_ASR0) - 1]
+ validated_ASR0[:len(validated_ASR0) - 1],
current_mes[len(validated_ASR0) - 1:] + current_mes[0:len(validated_ASR0) - 1],
b_B_l,
do_not_validate=True)

# 'w' at the first or last position only processed as 'multiple' wild cards if no 'b' or 'B' found at first and last position. Else, 'w'
e elif validated_ASR0[0] == 'w':
# Recursive call for validated_ASR0[0] = 'b'. 'b' saved in temporal storage
# and 'w' replaced with 'b'. After recursive execution,
# --> if match True, restore original 'b' on b_B_l
# --> if match False, b_B_l is reset by recursive call

temp_b_stored = b_B_l[0]
b_B_l[0] = []
validated_ASR0[0] = 'b'
if _match4_I1(validated_ASR0, current_mes, b_B_l, do_not_validate=True):
b_B_l[0] = temp_b_stored
return True
e else:
return False
e elif validated_ASR0[-1] == 'w':
# the same as last elif, but considering the last position instead

temp_b_stored = b_B_l[0]
b_B_l[0] = []
validated_ASR0[-1] = 'b'
if _match4_I1(validated_ASR0, current_mes, b_B_l, do_not_validate=True):
b_B_l[0] = temp_b_stored
return True
e else:
return False
e elif:
if len(validated_ASR0) != len(current_mes):

b_B_l[0], b_B_l[1], b_B_l[2] = [], [], []
return False
e else:
b_B_l[2] = [[]]
for i in range(len(current_mes)):
if not _match_char(validated_ASR0[i], current_mes[i], b_B_l[i]):
    b_B_l[0], b_B_l[1], b_B_l[2] = [], [], []
return False
return True

# to be called after _match4_I1. It works the same, but in case of no match, b_B_l
# remains unchanged.
def _match4_I2(ASR1, current_mes, b_B_l, **kwargs):
    backup_b_B_l = [b_B_l[0][:], b_B_l[1][:], b_B_l[2][:]]
    # this approach does not work for ["b","B",0,1,...] case if 'b' and 'B' present on b_B_l,
    # but such thing cannot happen if on I1 we only allow either 'B' or 'b'
    if not ASR1: # if not element found in ASR1
        return False
    if ASR1[0] == 'b': # check for duplicate 'b', 'B' and 'l' in b_B_l and ASR1
        while ASR1 and ASR1[0] == 'b':
            ASR1.pop(0)
        if ASR1[0] == 'B':
            while ASR1 and ASR1[0] == 'B':
                ASR1.pop(0)
    if not ASR1: # if not element found in ASR1
        return False
    # we do the same with the last element
    if ASR1[-1] == 'b':
        if b_B_l[0]:
            while ASR1 and ASR1[-1] == 'b':
                ASR1.pop()
    if ASR1[-1] == 'B':
        if b_B_l[1]:
            while ASR1 and ASR1[-1] == 'B':
                ASR1.pop()
    if not ASR1: # if not element found in ASR1
        return False
    # we handle the case when l is already used and finally, we validate.
    if b_B_l[2]: # if b_B_l["l"] not empty,
        ASR1.insert(1, "l")
validate_ASR1 = _validate_ASRi(ASR1)
    if "l" in validate_ASR1:
        validated_ASR1 = validate_ASRi(ASR1)
        validate_ASR1.remove("l")
if not 'validated_ASR1' in locals():
    validated_ASR1 = validate_ASRi(ASR1)
if _match4_I2(validated_ASR1, current_mes, backup_b_B_l):
    b_B_l[0], b_B_l[1], b_B_l[2] = backup_b_B_l[0], backup_b_B_l[1], backup_b_B_l[2]
return True
else:
    return False

# calls _match4_II
def _match1_I1(ASR0, current_mes, b_B_l, **kwargs):
    return _match4_II(ASR0, current_mes, b_B_l, **kwargs)

# return ASRi with valid b, B and l instances
# return error if ASRi is empty
# fills found_l_positions if necessary
def validate_ASRi(ASRi):
    found_l_positions = []
    if len(ASRi) > 1:
        return_ASRi = []
        b_0 = ASRi[0] == 'b'
        B_0 = ASRi[0] == 'B'
        first_bb_found = b_0 or B_0
        if first_bb_found: # if b or B found 2 at ASRi[0], append it to return_ASRi
return ASRi.append(ASRi[0])

for i in range(len(ASRi)):
    if not first_bB_found and ASRi[i] in {'b', 'B'}:
        # if it's the first instance of 'b' or 'B', but not i=0
        if i == len(ASRi) - 1:
            # if the first bB instance is the last element, append it to the list
            return ASRi.append(ASRi[i])
        first_bB_found = True
    elif ASRi[i] == 'l':
        return ASRi.append(ASRi[i])
    else:
        if ASRi[i] not in {'b', 'B'}:
            return ASRi = ASRi
else:
    raise ValueError("Processed ASRi cannot be empty")

return return_ASRi, found_l_positions

# returns False if no match occurs,
# returns True if character or wild card match, or one time 'l' match
# updates b_B_l to include found 'l' (if found)
def _match_char(char_bu, char_current_mes, b_B_l, pos=None):
    if char_bu == char_current_mes:
        return True
    elif char_bu in {'0', '1'}:
        return False
    elif char_bu == 'w':
        return True
    elif char_bu == 'l':
        b_B_l[2][0].append(pos)
        b_B_l[2].append(char_current_mes)
        return True
    elif char_bu == 'r':
        return bool(random.randint(0, 1))
    else:
        raise ValueError("{} not a valid character for _match_char".format(char_current_mes))

# write reply considering broadcast unit's last component
def _write_reply(answer_ASRi, env_mes_list, b_B_l, max_mes_len=8):
    reply_mes = []
    for j in range(len(answer_ASRi)):
        if len(reply_mes) > max_mes_len:
            del reply_mes[-max_mes_len:]
        env_mes_list.insert(0, reply_mes)
        return
    # first three cases, for when b, B, or l is written (only once)
    # last case, write 0 or 1
    if answer_ASRi[j] == 'b':
        if b_B_l[0] != []:
            reply_mes.extend(b_B_l[0])
        b_B_l[0] = []
    elif answer_ASRi[j] == 'B':
        if b_B_l[1] != []:
            reply_mes.extend(b_B_l[1])
        b_B_l[1] = []
    elif answer_ASRi[j] == 'l':
        if b_B_l[2] != []:
            reply_mes.extend(b_B_l[2])
        b_B_l[2] = []
    elif answer_ASRi[j] in {'1', '0'}:
        if j in b_B_l[2][0]:
            b_B_l[2][0].pop(0)
        reply_mes.append(b_B_l[2].pop(1))
    else:
        reply_mes.append(answer_ASRi[j])
    elif answer_ASRi[j] == 'w':
        pass
    elif answer_ASRi[j] == 'r':
        reply_mes.append(str(random.randint(0, 1)))
A.3. Code of List Sorting

```python
else:
    raise ValueError('non valid character encountered, {}' . format (answer_ASRi[j]))
env_mes_list.insert(0, reply_mes)

# endregion

def fit (max_iterations=500,
popsize=1200,
bd_len=36,
    prob_dist=(0.20, 0.20, 0.00, 0.20, 0.00, 0.20, 0.00, 0.00, 0.00, 0.20):
    population = [initialize_random_bd_no_0_fitness(bd_len, prob_dist=prob_dist)
for i in tqdm(range(popsize))]
print('
−start−
')
max_fitness = 0
current_max_fitness = 0
max_fitness_history = []
for i in range(max_iterations):
    random.seed()
    fitness_list = _bd_list_fitness(population)
    current_max_fitness = max(fitness_list)
    max_fitness_history.append(current_max_fitness)
    if current_max_fitness > max_fitness:
        max_fitness = max(fitness_list)
        best_bd = population[argmax(fitness_list)]
        print(best_bd)
    average_fitness = mean([i for i in fitness_list if i != -0.5])
    fitness_list = _sigma_truncation(fitness_list, c=6)
    newpop = list()
    for j in range(int(math.floor(popsize / 2))):
        bd_ind0 = _roulette_selection(fitness_list)
        bd_ind1 = _roulette_selection(fitness_list)
        children_bd = crossover(population[bd_ind0], population[bd_ind1])
        simple_respectful_mutation(children_bd[0], 0.02, prob_dist=prob_dist)
        simple_respectful_mutation(children_bd[1], 0.02, prob_dist=prob_dist)
        newpop.extend(children_bd)
    population = newpop
print('iteration:{:.6f}/ curr. max:{:.4f} overall max:{:.4f} current mean:{:.4f}'.format
    i, max_iterations, current_max_fitness, max_fitness, average_fitness)
'

#region dinamic plotting
if i ==0:
    xdata = []
ydata = []
plt.show()
axes = plt.gca()
axes.set_xlim(0, max_iterations)
axes.set_ylim(0, 4)
line, = axes.plot(xdata, ydata, 'r-')
xdata.append(i)
ydata.append(average_fitness)
line.set_xdata(xdata)
line.set_ydata(ydata)
plt.draw()
plt.pause(1e-17)
time.sleep(0.1)

# endregion

plt.show()
print(best_bd)
```

...
import cProfile
import re
bat = 1
cProfile.run('fit', 'restats')
import pstats
p = pstats.Stats('restats')
p.strip_dirs()
p.sort_stats('cumulative').print_stats(10)
**A.3.3 2nd approach, final version**

**main_v10.py**

```python
import modules_ga_v10 as mo
import time
from matplotlib import pyplot as plt

print(""n \n------------------ \n"n")
time.sleep(0.5)

random.seed(4)

# print(sum((mo.measure_order(mo.generate_random_list(mo.list_size)) for i in range(200)))/200)

computed_sizes = [4,5,6,7,8,9,10]

mean_istep_count_hist = [2.9961, 5.0081, 7.4823, 10.5419, 13.8817, 18.0272, 22.5761]

max_istep_count_hist = [6, 10, 15, 21, 27, 35, 41]

max_qstep_count_hist = [5, 10, 19, 30, 41, 69, 96]


max_qstep_count_hist = [5, 10, 19, 30, 41, 69, 96]

computed_sizes = [4,5,6,7,8,9,10]

plt.plot(computed_sizes, max_pstep_count_hist, color = '#0040ff', label='GP maximum')
plt.plot(computed_sizes, max_qstep_count_hist, color = '#ff0505', label='quicksort maximum')
plt.plot(computed_sizes, mean_istep_count_hist, color = '#1bb628', label='insertion sort maximum')
plt.plot(computed_sizes, mean_qstep_count_hist, color = '#9ea0ff', label='GP mean')
plt.plot(computed_sizes, mean_qstep_count_hist, color = '#7f7f7f', label='quicksort mean')
plt.plot(computed_sizes, mean_istep_count_hist, color = '#83ed8c', label='insertion sort mean')
plt.xticks(computed_sizes)
plt.legend()
plt.xlabel('List size')
plt.ylabel('Number of comparisons')
plt.show()
```
# uncomment to perform search
mo.get_comp_number_data()
import math
import random
from statistics import stdev, mean
import numpy as np
from tqdm import tqdm
import functools
import os as sys
import time
import multiprocessing as mp
import copy

# declare global variables

# region parameters
seed = 4
list_size = 9
test_prog = 
    ['0++', '1+', '1-', '1-', '0-', '0--', '1--', '1--', 'c', '0+', '0--', '1-', '1-', '0++', 'i', '1--', 's', '0++', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-', '0-']

# program execution
max_comparisons = list_size * list_size * 10
max_program_iterations = 10 * list_size
program_size = 10

# fitness function parameters
number_of_lists_sampled = 500

# ga parameters
max_ga_iterations = 20
prob_mut = 0.02
popsize = 40
c = 2

# region tools

def argmax(iterable):
    return max(enumerate(iterable), key=lambda x: x[1])[0]

# endregion

# region other sorting algos

def qsort(l, first_call = True):
    if not l:
        return l, 0  # empty sequence case
    pivot = l[random.choice(range(0, len(l)))]
    head, count0 = qsort([elem for elem in l if elem < pivot], first_call=False)
    tail, count1 = qsort([elem for elem in l if elem > pivot], first_call=False)

    return head + [pivot] + tail

# endregion

def qsort(l, first_call = True):
    if not l:
        return l, 0  # empty sequence case
    pivot = 1[random.choice(range(0, len(l)))]
    head, count0 = qsort([elem for elem in l if elem < pivot], first_call=False)
    tail, count1 = qsort([elem for elem in l if elem > pivot], first_call=False)

    return head + [pivot] + tail

# taken from https://stackoverflow.com/questions/18262306/ quicksort-with-python

def qsort(l, first_call = True):
    if not l:
        return l, 0  # empty sequence case
    pivot = l[random.choice(range(0, len(l)))]
    head, count0 = qsort([elem for elem in l if elem < pivot], first_call=False)
    tail, count1 = qsort([elem for elem in l if elem > pivot], first_call=False)

    return head + [pivot] + tail

# endregion

A.3. Code of List Sorting
def isort(l):
    count = 0
    if first_call:
        l[:] = head + [pivot] + tail
        return count
    else:
        return head + [pivot] + tail, count

# http://interactivepython.org/courselib/static/pythonds/SortSearch/TheInsertionSort.html
def isort(l):
    comp_count = 0
    for index in range(1, len(l)):
        currentvalue = l[index]
        position = index
        while position > 0 and l[position - 1] > currentvalue:
            comp_count += 1
            position = position - 1
            l[position] = currentvalue
        if l == list(range(len(l))):
            return comp_count
        return comp_count

#endregion

#region shuffled list

# return n sized shuffled list
def generate_random_list(n):
    return _list = list(range(n))
    random.shuffle(return_list)
    return return_list

def measure_order(input_list):
    ""
    calculate the level of disorder of the input list. It is measured on a logarithmic scale
    :param input_list: the list to be measured
    :return: positive float
    ""
    sum_of_difference = sum((abs(input_list[i] - i) for i in range(len(input_list))))
    return math.log1p(sum_of_difference)

#endregion

#region program - commands

class ordering_program:

    prob_distr = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
    possible_chars = ('s', 'i', '0+', '1+', '0-', '1-', '0++', '0--', '1++', '1--', 'c')

    def __init__(self, n=None, **kwargs):
        if 'command_list' in kwargs or 'command_string' in kwargs:
            initialize = False
        else:
            initialize = True
        if not initialize:
            if 'command_list' in kwargs:
                self.command_list = kwargs['command_list']
            elif 'command_string' in kwargs:
                self.command_list = kwargs['command_list']

    def
A.3. Code of List Sorting

```python
self.command_list = [command for command in kwargs['command_string']]
```

```python
self._initialize_random_command_list(n):
    state = random.getstate()
    random.seed()
    self.command_list = list(random.choices(self.possible_chars, weights=self.prob_distr, k=n))
    random.setstate(state)
```

```python
def order_list_with_program(self, shuffled_list, initial_pos = math.floor(list_size / 2),
                         plusplus_difference = math.floor(list_size / 8):
    p0 = initial_pos
    p1 = 0
    len_of_moves = (1, plusplus_difference)
    comparison_counter_container = [0] # we make it a list to make it mutable

    for i in range(max_program_iterations):
        for command_pos in range(len(self)):
            current_char = self.command_list[command_pos]

            if comparison_counter_container[0] >= max_comparisons:
                return comparison_counter_container[0]

            if shuffled_list == list(range(list_size)):
                return comparison_counter_container[0]

            elif current_char in {'s', 'i'}:
                execute_command(current_char, shuffled_list, comparison_counter_container, p0, p1)
                #print('{} : {} - {}'.format(comparison_counter_container, p0, p1))

            elif current_char == 'c':
                pass

            else: # movement command found
                if current_char[0] == '0':
                    p0 += len_of_moves[len(current_char) - 2]
                elif current_char[0] == '1':
                    p1 += len_of_moves[len(current_char) - 2]
                p0 = p0 % list_size
                p1 = p1 % list_size

            comparison_counter_container[0] += 1

    def mutate_k_position(self, k):
        self.command_list[k] = list(random.choices(self.possible_chars, weights=self.prob_distr, k=1))[0]
```

# order_list_with_program commands

# command swap 's'
```python
def swap(shuffled_list, comparison_counter_container, p0, p1):
    if p0 == p1:
        return shuffled_list[min(p0,p1)], shuffled_list[max(p0,p1)] = sorted([shuffled_list[p0], shuffled_list[p1]
```
# Chapter A. Implemented Code

```python
# command insert 'i'
def insert(shuffled_list, comparison_counter_container, p0, p1):
    if p0 == p1:
        pass
    else:
        if shuffled_list[p0] > shuffled_list[p1]:
            element_to_be_inserted = shuffled_list[p0]
            shuffled_list.insert(p1 + 1, element_to_be_inserted)
        else:
            element_to_be_inserted = shuffled_list[p0]
            shuffled_list.insert(p1, element_to_be_inserted)

    if shuffled_list[p0] == element_to_be_inserted:
        shuffled_list.pop(p0)
    elif shuffled_list[p0 + 1] == element_to_be_inserted:
        shuffled_list.pop(p0 + 1)
    else:
        raise ValueError('Second element not erased')

    comparison_counter_container[0] += 1

# endregion

# region ga operators
class individual:
    def __init__(self, program):
        if not isinstance(program, ordering_program):
            raise TypeError('The initialization requires an ordering_program type object. {} type object was given'.format(type(program)))
        self.program = program

    def __str__(self):
        return str(self.program)

    def calc_fitness(self):
        random.seed(seed)
        all_fitness_values = []
        shuffled_lists = [generate_random_list(list_size) for i in range(number_of_lists_sampled)]
        for shuffled_list in shuffled_lists:
            all_fitness_values.append(fitness_function(self.program, shuffled_list))

        self.fitness = (mean(all_fitness_values) + min(all_fitness_values)) / 2

        return self.fitness

    def crossover(self, other):
        # mates using two point crossover
        ind_0_command_list = self.program.command_list
        ind_1_command_list = other.program.command_list

        mating_indexes = [random.randint(0, len(ind_0_command_list) - 1), random.randint(0, len(ind_0_command_list) - 1)]
        mating_indexes.sort()
```

A.3. Code of List Sorting

```python
new_individuals = [
    individual(
        command_list=ind_0_command_list[:mating_indexes[0]] +
        ind_1_command_list[mating_indexes[0]:mating_indexes[1]] +
        ind_0_command_list[mating_indexes[1]:]),
    individual(
        command_list = ind_1_command_list[:mating_indexes[0]] +
        ind_0_command_list[mating_indexes[0]:mating_indexes[1]] +
        ind_1_command_list[mating_indexes[1]:]))
]

# new_bds[0].message_spawner_set = spawn_mes0
# new_bds[1].message_spawner_set = spawn_mes1

return new_individuals

def mutate(self):
    mut_index = np.random.binomial(2, prob_mut, len(self.program))
    for i in range(len(self.program)):
        if mut_index[i]:
            self.program.mutate_k_position(i)

class population:
    best_individual = None
    best_fitness = -1e10

    def __init__(self):
        self.pop = [individual(get_working_program(program_size, list_size)) for _ in range(popsize)]
        self.iterations_done = 0

    def ga_iteration(self, verbose = 1):
        if verbose:
            time.sleep(0.5)
        fitness_list = [self.pop[i].calc_fitness() for i in tqdm(range(popsize))]
        time.sleep(0.5)

        # keeping track of best individuals and iteration
        best_index = argmax(fitness_list)
        best_iteration_fitness = fitness_list[best_index]
        best_iteration_individual = self.pop[best_index]
        mean_iteration_fitness = mean(fitness_list)
        self.iterations_done += 1

        if self.best_fitness < best_iteration_fitness:
            self.best_fitness = best_iteration_fitness
            self.best_individual = best_iteration_individual
        if verbose:
            print("{:4.4f} --> {}").format(self.best_fitness, str(self.best_individual))

        fitness_list = sigma_truncation(fitness_list)

        selected_pop = [self.pop[roulette_selection(fitness_list)] for _ in range(popsize)]
        if verbose:
            for indiv in self.pop:
                print("{:4.4f} --> {}").format(indiv.fitness, str(indiv))

        newpop = []
```
for i in range(0, popsize, 2):
    newpop.extend(selected_pop[i].crossover(self.pop[i + 1]))
self.pop = newpop
for indiv in self.pop:
    indiv.mutate()
if verbose:
    print('it: {:5} overall best: {:4f} -- current best: {:4f} -- current mean: {:4f}'.format(self.iterations_done, self.best_fitness, best_iteration_fitness, mean_iteration_fitness))

# fitness function
def fitness_function(program, shuffled_list):
    initial_entropy = measure_order(shuffled_list)
    n_comparisons = program.order_list_with_program(shuffled_list)
    bonus_well_ordered = 0
    if shuffled_list == list(range(list_size)):
        bonus_well_ordered = max_comparisons - n_comparisons
    return initial_entropy - measure_order(shuffled_list) + bonus_well_ordered

# scale fitness with sigma truncation
def sigma_truncation(list_of_fitness):
    sigma = stdev(list_of_fitness)
    mu = mean(list_of_fitness)
    return [(list_of_fitness[i] - (mu - c * sigma)) for i in range(len(list_of_fitness))]

# returns list with number of offspring using roulette wheel selection
def roulette_selection(weights):
    '''performs weighted selection or roulette wheel selection on a list
    and returns the index selected from the list'''
    # sort the weights in ascending order
    sorted_indexed_weights = sorted(enumerate(weights))
    indices, sorted_weights = zip(*sorted_indexed_weights)
    # calculate the cumulative probability
    tot_sum = sum(sorted_weights)
    if tot_sum == 0:
        prob = [1 / len(sorted_indexed_weights) for i in range(len(sorted_indexed_weights))]
    else:
        prob = [x / tot_sum for x in sorted_weights]
    cum_prob = np.cumsum(prob)
    # select a random a number in the range [0,1]
    random_num = random.random()
    for index_value, cum_prob_value in zip(indices, cum_prob):
        if random_num < cum_prob_value:
            return index_value

#endregion

def fit(verb=1, size=list_size):
    global list_size
    old_size, list_size = list_size, size
    global number_of_lists_sampled
    fit_population = population()
    best_hist = []
    for i in tqdm(range(max_ga_iterations), 'GA iteration '):
        fit_population.ga_iteration(verb=verb)
        if verbose == 1:
            print('-----')
        print('maxComparisions = {} '.format(max_comparisions))
A.3. Code of List Sorting

```python
print('--------')
best_hist.append(fit_population.best_individual.calc_fitness())
time.sleep(0.5)
print('Best individual: ')
print('--------')
print(fit_population.best_individual)
if verbose == 1:
    print(fit_population.best_individual.calc_fitness())
plt.plot(list(range(1, 1 + len(best_hist))), best_hist)
plt.show()
list_size = old_size
return fit_population.best_individual.program

def test_ord_prog(test_prog):
    random.seed(5)
global list_size
ls = list_size
pstep_count_hist, qstep_count_hist, istep_count_hist = [], [], []
all_well_ordered = True
for i in range(10000):
    shuffled_list0 = generate_random_list(ls)
    shuffled_list1 = copy.deepcopy(shuffled_list0)
    shuffled_list2 = copy.deepcopy(shuffled_list0)
    p = test_prog
    # sorting with different algorithms, and saving number of comparisons
    pstep_count_hist.append(p.order_list_with_program(shuffled_list0))
    qstep_count_hist.append(qsort(shuffled_list1))
    istep_count_hist.append(isort(shuffled_list2))
    if shuffled_list0 != list(range(len(shuffled_list0))):
        all_well_ordered = False
print('#LS = {}'.format(list_size))
print('#test_prog = {}'.format(test_prog))
return all_well_ordered,
mean(pstep_count_hist), mean(qstep_count_hist), mean(istep_count_hist),
max(pstep_count_hist), max(qstep_count_hist), max(istep_count_hist)

def get_comp_number_data():
    mean_pstep_count_hist, mean_qstep_count_hist, mean_istep_count_hist,
    max_pstep_count_hist, max_qstep_count_hist, max_istep_count_hist = [], [], [], [], [], []
hists = [mean_pstep_count_hist, mean_qstep_count_hist, mean_istep_count_hist,
    max_pstep_count_hist, max_qstep_count_hist, max_istep_count_hist]
for i in range(9, 15):
    # variable parameters
global list_size
global max_comparisons
global popsize
global program_size
global max_program_iterations
list_size = i
max_comparisons = list_size * list_size * 10
program_size = 20 + 10 * (i - 6)
max_program_iterations = list_size * 10

    test_prog = fit( verbose=0, size=i )
all_well_ordered, \
mean_pstep_count, mean_qstep_count, mean_istep_count, \
max_pstep_count, max_qstep_count, max_istep_count = test_ord_prog(test_prog)
    obtained_results = (mean_pstep_count, mean_qstep_count, mean_istep_count,
    max_pstep_count, max_qstep_count, max_istep_count)```
for hist_index in range(6):
    hists[hist_index].append(obtained_results[hist_index])

if not all_well_ordered:
    print(f'Not all well ordered, failed iteration: {i} \n\n——— \n
    −−−−−

')
    raise ValueError
else:
    print(f'iteration {i}')
    print(f'\n  mean_pstep_count_hist = {mean_pstep_count_hist} \n  mean_qstep_count_hist = {mean_qstep_count_hist} \n  mean_istep_count_hist = {mean_istep_count_hist} \n  max_pstep_count_hist = {max_pstep_count_hist} \n  max_qstep_count_hist = {max_qstep_count_hist} \n  max_istep_count_hist = {max_istep_count_hist} ')


def get_working_program(ps, ls):
    # correct global parameters
    global list_size
    global max_comparisons
    global program_size

    #backup original global variables
    list_size_copy = copy.copy(list_size)
    max_comparisons_copy = copy.copy(max_comparisons)
    program_size_copy = copy.copy(program_size)

    #set new global variables
    list_size = ls
    max_comparisons = ls * ls * 10
    program_size = ps

    working_program = False
    n_of_programs_checked = 0
    while not working_program:
        n_of_programs_checked += 1
        op_to_be_checked = ordering_program(n = ps)
        i = 0
        list_has_been_ordered = True
        while list_has_been_ordered and i < 100:
            i += 1
            shuffled_list = generate_random_list(ls)
            op_to_be_checked.order_list_with_program(shuffled_list)
            if shuffled_list != list(range(len(shuffled_list))):
                list_has_been_ordered = False

        if list_has_been_ordered:
            working_program = True
        if n_of_programs_checked % 15000 == 0:
            print(f'n of iterations without valid op created: {n_of_programs_checked}. List size {list_size} and op size {program_size}')
        if n_of_programs_checked > 1500000:
            raise ValueError

    # restore original global variables
    list_size = list_size_copy
    max_comparisons = max_comparisons_copy
    program_size = program_size_copy

    return op_to_be_checked

def local_search(n_iterations, start_string = '∗w0llwlwwwwwwww:1001 rr11 rrr 1 r 1 r r r '): # to be adapted to second approach
A.3. Code of List Sorting

```python
random.seed()
state = random.getstate
string = start_string
string_change = string
new_fitness = -1e10
print(start_string + ' --> ' + 'start string')
print('---start---')
for it in range(n_iterations):
    random.setstate = state
    fitness = fitness_function(broadcast_device(sr=string),10000,10,20,False,64)
    for new_string in dist_1_generator(string):
        random.setstate = state
        new_fitness = fitness_function(broadcast_device(sr=new_string),200,10,20,False,64)
        if new_fitness > fitness:
            random.setstate = state
            new_fitness = fitness_function(broadcast_device(sr=new_string),1000,10,20,False,64)
            if not new_fitness > fitness:
                continue
            fitness = new_fitness
            string = string_change
            random.setstate = state
            if fitness_function(broadcast_device(sr=string),10000,10,20,False,64) > fitness:
                print(new_string + ' --> ' + str(new_fitness))
        string = string_change
        print('Itertation: ' + str(it))

def dist_1_generator(string):
    possible_chars = ('0', '1', 'w', 'l', 'r')
    for i in range(len(string)) if i%30 != 0 and i%30 != 15:
        for char in possible_chars:
            if not (15 < i%30 < 30 and char == 'l'):
                yield string[:i] + char + string[i+1:]

#list_size = 6
#p = get_working_program(20, list_size)
#oldsteps = steps
#steps = min(max((p.order_list_with_program(generate_random_list(list_size)) for i in range(10000)), steps)
#if steps < oldsteps:
#    print('----')
#    print(steps)
```
Bibliography


