Universiteit Leiden

Master Computer Science

Adapting and Applying Evolutionary Algorithms to variable length Representations of Optimization Problems

Name:	Matthijs de Zeeuw
Student ID:	s2705028
Date:	23-04-2025
Specialisation:	Artificial intelligence
1st supervisor:	Prof.dr. T.H.W. Bäck
2nd supervisor:	Dr. A.V. Kononova

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Traditional optimization problems often assume a fixed dimensionality for the solution space, which limits their applicability to real-world scenarios. There, the optimal structure and length of solution vectors are often not predetermined. variable-length optimization algorithms address this issue by allowing both the structure and length of potential solutions to be dynamically optimized. In this work, we adapt certain evolutionary algorithms to deal with variable-length problem representations. First, existing work on variable length optimization is reviewed. Then, adaptations of classical binary evolutionary algorithms are introduced that can deal with variable length search spaces. To evaluate these algorithms, a variable length version of the OneMax problem is introduced. Additionally, a popular continuous evolutionary algorithm, CMA-ES, is extended to a variable length form (VL-CMA-ES) and tested on a simple continuous optimization problem called FunctionMatch. The experiments that were done show that these adaptions are able to solve the test problems.

Contents

1 Introduction **1**

2 Related work **1**
2.1 Metameric representation . . . 2
2.2 Selection operators . . . 2
2.2.1 Constraints . . . 3
2.2.2 Parsimony pressure . . . 3
2.2.3 Tournament selection . . . 3
2.2.4 Niche selection . . . 3
2.3 Crossover operators . . . 4
2.3.1 Cut-and-splice crossover . . . 4
2.3.2 Spatial crossover . . . 5
2.3.3 Synapsing variable length Crossover . . . 6
2.3.4 Similarity crossover . . . 8
2.4 Mutation . . . 8

3 Test problems **8**
3.1 Variable-Length OneMax . . . 10
3.2 FunctionMatch . . . 11

4 Fixed-dimensional pseudo-boolean and continuous algorithms **11**
4.1 Greedy Hill Climber . . . 12
4.2 Randomized Local Search . . . 12
4.3 $(1 + \lambda)$ EA with static mutation rates . . . 13
4.4 fGA . . . 13
4.5 Two-rate $(1 + \lambda)$ EA with adaptive mutation rates . . . 14
4.6 $(1 + \lambda)$ EA$_{norm}$ . . . 15
4.7 $(1 + \lambda)$ EA$_{var}$ . . . 15
4.8 $(1 + \lambda)$ EA$_{log-n}$ . . . 16
4.9 Self-adjusting $(1 + (\lambda, \lambda))$ GA . . . 16
4.10 vGA . . . 17
4.11 CMA-ES . . . 17

5 Variable-dimensional algorithms **19**
5.1 Variable-Length greedy Hill Climber . . . 19
5.2 Variable-Length Random Local Search . . . 20
5.3 Various Variable-Length Evolutionary Algorithms . . . 20
5.4 Variable-Length self-adjusting Genetic Algorithm . . . 20
5.5 VL-vGA . . . 21
5.6 VL-CMA-ES . . . 21

6 Experiments **23**
6.1 Comparing VL-gHC and VL-LRS . . . 23
6.1.1 Results . . . 23
6.2 Comparing VL-EA algorithms . . . 24
6.2.1 Results . . . 24

6.3 Comparing VL-saGA and VL-vGA 26
6.3.1 Results 28
6.4 Comparing VL-vGA with VL-vGA Mutation Only 28
6.4.1 Results 31
6.5 VL-CMA-ES 31
6.5.1 Results 33

7 Conclusions **33**

1 Introduction

Most optimization problems assume that the dimensionality of the solution space is fixed. However, for some real-world problems, the optimal length of the solution vector cannot be known a priori. In such cases, variable length problem representations can be useful. In variable length optimization problems, the length of potential solutions can change. The variable length optimization problem can be defined as follows:

$$\begin{aligned} &\text{minimize } f(\mathbf{x}, n), \\ &\text{subject to } x_{i,L} \leqslant x_i \leqslant x_{i,U}, i = 1, ..., n, \\ &\qquad 1 \leqslant n \leqslant n_{max} \end{aligned} \tag{1}$$

Where U and B are the lower and upper bounds for the variable x_i, and n_{min} and n_{max} are the minimum and maximum dimensionality of the search space.
The aim of this project is to investigate how evolutionary algorithms can be applied to variable-length representations of optimization problems. We limit our investigation to single-objective optimization and focus on classic evolutionary algorithms. Section 2 provides an overview of previous work on variable-length optimization. In Section 3, the variable length version of the OneMax problem (VL-OneMax) is introduced and the variable length continuous Function-Match problem is described. Section 4 gives a description of various classical binary evolutionary algorithms, as described in [2], and of a popular continuous evolutionary optimization algorithm (CMA-ES). In Section 5 the variable length adaptations of all algorithms described in the previous section are presented. In the experimental section 6, the binary evolutionary algorithms are tested on the VL-OneMax problem. For testing VL-CMA-ES, a simple continuous optimization problem called FunctionMatch is used. Section 7 concludes the paper and outlines directions for future research.

2 Related work

For Evolutionary Algorithms (EAs), solutions are typically represented as vectors of variables of a fixed length. These vectors are often referred to as genomes or chromosomes. Research into variable-length problem optimization has been ongoing for about six decades. One of the first experiments on a variable-length problem was conducted by Schwefel [11]. The aim was to find an optimal two-phased nozzle design using an evolutionary strategy. The nozzle designs consisted of a sequence of cone-shaped segments with varying diameters. Mutation was effected by making small random adjustments to the nozzle length and diameters of the segments. This involved manually replacing segments with others of different diameters and adding new segments. By repeatedly mutating the initial nozzle design and evaluating its performance, the optimal shape and number of segments could be found. Since then, various methods for dealing with variable-length problem representations have been proposed. These include methods that allow some variables to remain unexpressed. Sprave and Rolf [12] used a problem representation that encodes each solution as two vectors: one with the values of each variable and one encoding which of the variables are active. The advantage of this approach is that the solution-vectors are fixed-length, allowing the use of existing length-preserving operators.
variable length methods have been used to tackle multi-objective problems. Yu [14] used the Non-dominated Sorting Genetic Algorithm (NSGA-II) with variable length chromosomes for

vehicle maneuver planning, employing a variation of the one-point crossover operator that chooses separate crossover points for each parent. This way, the crossover operator produces offspring of different lengths from the parents. The mutation step consists of three operations: gene mutation, gene deletion, and gene copying. Each variable in the solutions is duplicated, deleted, or mutated with a certain probability. The algorithm also uses a reduction operator that removes redundant variables from solutions (variables that do not have a significant effect on the solution's fitness), which saves computational resources.
Much of the more recent work on variable length optimization problems focuses on so-called **metameric representations**. This term is used in biology to refer to organisms whose anatomy consists of many similar but not identical segments [9].

2.1 Metameric representation

The metameric representation is inspired by metamerism in nature. That is to say: a metameric solution-vector consists of multiple metavariables. The wind farm problem, as described in [9], is an example of such a metameric problem representation. The aim is to place a number of wind turbines within a finite area, in such a way that the total amount of energy produced is optimized. The optimal number of wind turbines is not known in advance and can vary during the optimization process, making this a variable-length problem. A solution-vector for the wind farm problem consists of a number of **metavariables** representing the wind turbines. Each metavariable consists of a number of **design variables**. In the case of the wind farm problem, there are two design variables: the x- and y-coordinate denoting its location. For a given metameric problem the metavariables all have the same design variables. Only the values of the design variables differ between metavariables. A metameric solution vector can be defined as follows: a solution vector $\mathbf{x}$ consists of n metavariables, each of which contains v design variables such that $x_{i,j}$ represents the j^{th} design variable of the i^{th} metavariable. The number of metavariables in a solution can vary, but the number of design variables within a metavariable is fixed [8]. Ryerkerk et al. [9] provide an overview of previous research on metameric optimization problems.

2.2 Selection operators

Many variable length Evolutionary Algorithms (VL-EAs) use the same selection operators commonly found in fixed-length evolutionary algorithms. However, there are downsides to using these types of operator for variable-length optimization. One problem is uncontrolled bloat. This occurs when the solution-length within the population increases rapidly without a corresponding rise in population-fitness. Evaluating unnecessarily long solution vectors wastes computational resources [9]. Another problem is premature convergence. In variable-length optimization, the aim is not only to find the optimal solution but also to find the optimal solution length. A lack of length diversity within a population can lead to a premature convergence of the algorithm. Fixed-length selection operators select individuals based on their fitness. The length of an individual does not influence it's chances of being selected. However, taking solution-length into account during the selection step can be helpful in dealing with the problems of uncontrolled bloat and premature convergence in variable length problems. There are different ways of dealing with these problems [8].

2.2.1 Constraints

One simple way of dealing with the problem of uncontrolled bloat is to apply constraints to the solution-length. However, this only limits the effect of uncontrolled bloat. The solution-length may still increase uncontrollably, resulting in a population consisting almost entirely of solutions of maximum length [9].

2.2.2 Parsimony pressure

Another way of preventing uncontrolled bloat is to apply parsimony pressure. This works by adding a term to the evaluation function that penalizes longer solutions. When applying parsimony pressure, bloated solutions are less likely to be selected because shorter solutions with a similar fitness value will be penalized less [9].

2.2.3 Tournament selection

Length diversity within the population can be ensured in multiple ways. One way is to use tournament selection with small tournament sizes [9]. Small tournament sizes mean less selection pressure. This will increase the chances of weaker individuals being selected, but it will also result in more length diversity in the population, which is the desired effect.

2.2.4 Niche selection

Niche selection works by performing local selection on subsets of the population. A subset or niche consists of solutions of similar lengths. The local selection is done by applying a selection operator (e.g. tournament selection) to each niche. Within each niche, a subset of individuals is selected. These subsets are then combined to form the new generation. The number of individuals to be selected for each niche is determined by the population size and the number of niches (e.g. for 10 niches and a population size of 100, 10 individuals must be selected for each niche). If the number of individuals within a niche is smaller then the number that needs to be selected at each niche, all individuals in that niche are selected. This means that, after combining all the subsets, the size of the new population P_{G+1} might be smaller than the required population-size. In such cases, the local selection operator is applied to the entire population P_G until the size of P_{G+1} matches the required population-size. Niches are created for only a subset of each possible solution length within the population. A window function determines which lengths are considered. This results in a selection window $\mathbf{W}$, where $\mathbf{W} = \{L_{lb}, L_{lb} + 1, \ldots, L_{ub}\}$. There are different ways of determining the upper and lower bounds L_{lb} and L_{ub}.

Fixed-length window This assumes that the optimal solution length l^* is already known, i.e., $L_{lb} = L_{ub} = l^*$. Since only one niche is created, no niche selection occurs. Since, for variable-length problems the optimal solution-length is not known a priori, this option is not useful for variable-length problems.
Static window The static window requires some knowledge of the optimal solution length. Niches are created for a range of lengths, i.e., $L_{lb} < L_{ub}$. Niches ar created for each length l, with $L_{lb} < l < L_{ub}$. Using a static window ensures that the new population P_{G+1} contains individuals of all lengths within the specified window.
Moving window When using a moving window, niches are created for all lengths close to

the length of the best solution l_G^* in a generation G. Here $L_{lb} = l_G^* - \frac{w}{2}$ and $L_{ub} = l_G^* + \frac{w}{2}$. The width w of the window is defined by the user. No prior knowledge of the optimal solution length is required.

Biased window The biased window determines the lower and upper bounds based on l_G^* and a bias factor B_G.

$$\begin{aligned} L_{lb} &= \min\left(l_G^* - w/2 + B_G, l_G^*\right) \\ L_{ub} &= \max\left(l_G^* + w/2 + B_G, l_G^*\right) \end{aligned} \tag{2}$$

B_G is the bias term. When B_G increases, the window widens. $B_G = 0$ at initialization. It is updated for each new generation according to:

$$B_G = l_G^* - l_{G-1}^* + B_{G-1} \cdot \exp(-\lambda\sqrt{B_{G-1}}) \tag{3}$$

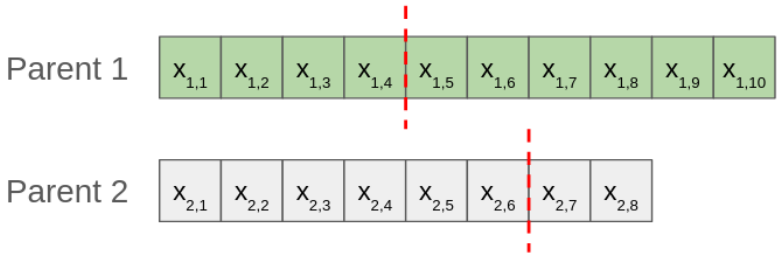
Here, B_{G-1} is the value of B for the previous generation. The exponential part of the equation ensures that B_G moves to $l_G^* - l_{G-1}^*$. The decay rate is determined by the decay factor λ and B_{G-1} [9].

2.3 Crossover operators

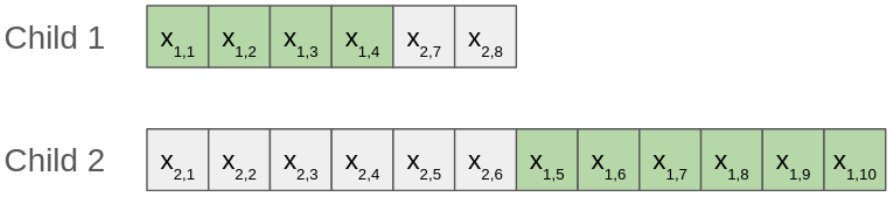
It is not absolutely necessary for variable-length algorithms to use length-varying crossover operators. Some algorithms use traditional fixed-length crossover operators and rely on the mutation operator to change solution lengths [13, 7]. There are also methods that do not use crossover at all [5, 10]. An issue that needs to be considered is the disruptive effect of crossover operators. For many evolutionary algorithms, the success of the crossover operator depends on its ability to preserve so-called building blocks present in the parent solutions. A building block is a subsequence that positively affects the fitness of a solution. Disruption occurs when the crossover operator fails to preserve these building blocks [9]. Various crossover operators have been proposed:

2.3.1 Cut-and-splice crossover

A simple variable length crossover method is cut-and-splice crossover. It is a n-point crossover method where the crossover points do not have to match between the parents. The number of metavariables exchanged can also differ for each parent, so the lengths of both children might differ from the lengths of either parent solution. Cut-and-splice crossover can be very disruptive when used in metameric problem representations, as it exchanges metavariables based on their location in the solution vector and the crossover points [9].



(a)

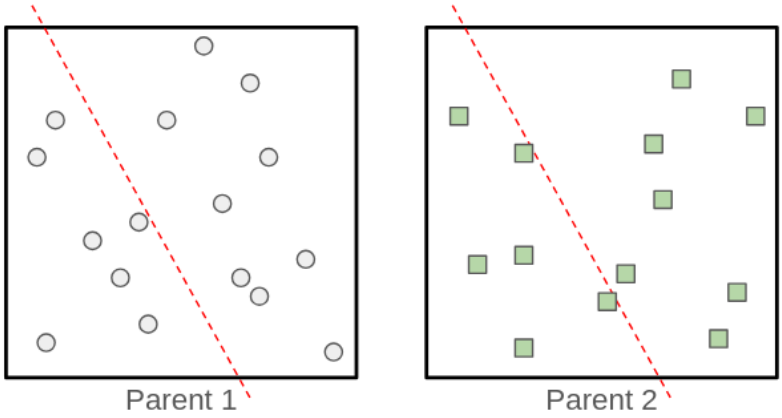


(b)

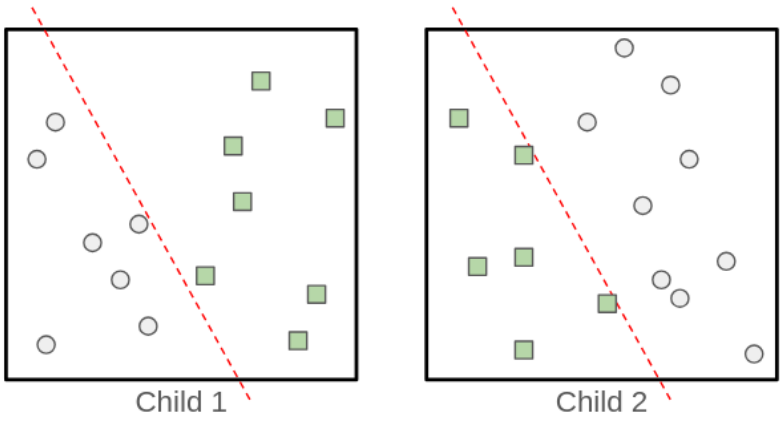
Figure 1: Example of a cut-and-splice crossover operation. A random point is selected on each parent. The parents are split up and recombined to form the offspring

2.3.2 Spatial crossover

Spatial crossover operates largely in the phenotype space. To decide which variables will be exchanged in the crossover operation, the spatial crossover method maps all solutions onto a lower-dimensional space. The split is made by dividing the space into two parts via a random line, in the 2-dimensional case, or a $n-1$ dimensional plane. Here n is the number of dimensions. In this way, the spatial relationships between the variables are preserved [1]. Spatial crossover is useful when the spatial location strongly influences the fitness function. For example, consider the metameric representation of the wind farm problem, where the aim is to create a layout of a wind farm with an unknown number of wind turbines. Each turbine is represented as a metavariable with an x- and y-coordinate [9]. The split can be made on the basis of the x- and y-coordinates, preserving the spatial relationship between the different wind turbines.



(a)



(b)

Figure 2: Example of a spatial crossover operation. The red, dotted line represents the separating line.

2.3.3 Synapsing variable length Crossover

Synapsing variable length Crossover (SVLC) tries to find similar sequences within the parent solutions. It was proposed by Hutt and Warwick [4]. First, the operator identifies similar sequences of values. An adapted version of the Needleman-Wunsch algorithm is used to do this. It works as follows. The algorithm locates the Longest Common Subsequence (LCS) in both parents. Splits are made at the beginning and end of the LCS on both parents. This results in two new pairs: the parts of the parent solutions that came before the LCS and the parts that came after the LCS. These can then be treated as two new problems. By repeating this process (until the remaining parts are smaller than a predetermined minimum length), a set of common subsequences is found. The variables within the identified regions are then linked or 'synapsed' together. Then, the parent solutions are realigned in such a way that the parts that are linked are opposite to each other. Crossover points are randomly chosen within the synapsed regions. This way, only the differences between the two parent solutions are exchanged [8]. Figure 3 gives an example of such an operation.

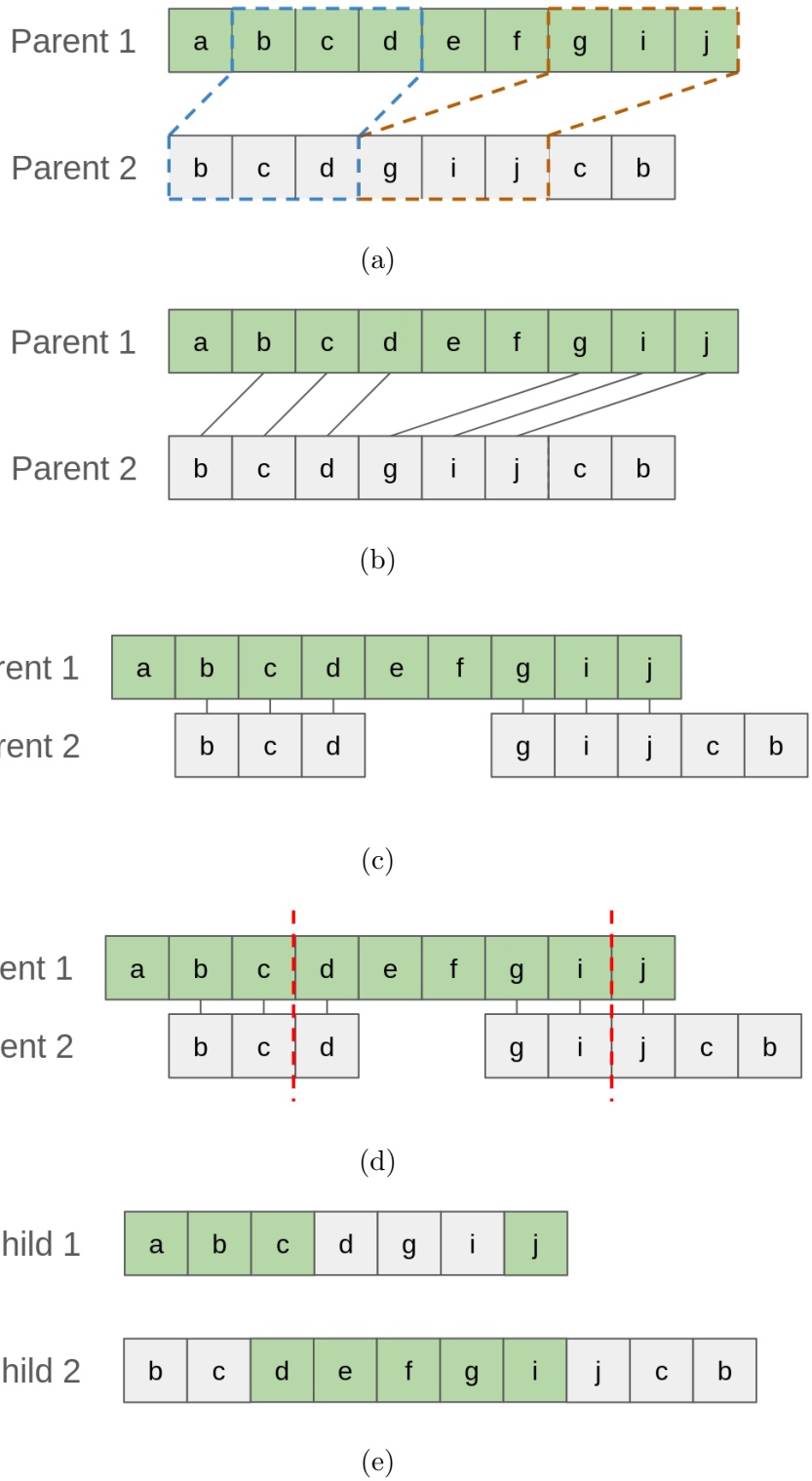


Figure 3: Example of a SVLC crossover operation: The Longest Common Subsequences between the two parents are identified (3a) The variables within the identified regions are linked or 'synapsed' together (3b) and realigned in such a way that the common subsequences are opposite each other (3c). Crossover points, denoted by the red, dotted lines, are then chosen within the 'synapsed' regions (3d). The variables between the crossover points are swapped out to create the children (3e). This way the common sequences are kept intact. The letters denote variables with the same values.

2.3.4 Similarity crossover

Like SVLC, similarity crossover tries to preserve common subsequences in the parent solutions. However, unlike SVLC similarity crossover looks at the metavariables. It uses a dissimilarity measure to identify pairs of similar metavariables in both parent solutions [8]. The dissimilarity between two metavariables M_1 and M_2 (one from each parent) is given by:

$$D = \frac{1}{n} \sum_{i=1}^{n} \frac{|M_{1,i} - M_{2,i}|}{L_i} \tag{4}$$

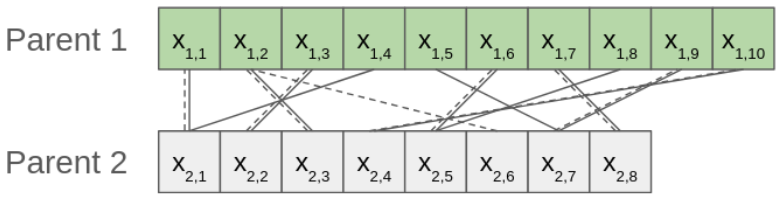
Where $M_{1,i}$ and $M_{2,i}$ are the values of the i^th design variable of the metavariables M_1 and M_2. L_i represents the length of the domain of the i^th design variable. $D = 0$ means that the metavariables are identical [8]. Similarity crossover begins by identifying similar metavariables in both parents. Then, each metavariable in one parent is linked to its most similar counterpart in the other parent and vice versa (see figure 4a). In most cases, this results in pairs of two metavariables being mutually linked. However, in some instances, two or more metavariables in one parent are linked to the same metavariable in the other parent. In the third step, the metavariables are separated into groups of one or more metavariables based on the links between them. This way, both solutions are segmented into subsets, where each subset of metavariables is linked to a subset of similar metavariables in the other solution (figure 4b). The number of subsets is the same for both parents, but the subsets do not always contain the same number of metavariables. A random number of these paired subsets will then be swapped between the parents to form the offspring (figure 4c). The advantage over spatial crossover is that similarity crossover can be applied to all metameric problem definitions and that the user does not have to choose which metavariables need to be considered. On the other hand, similarity crossover is more likely to be disruptive than spatial crossover [9].

2.4 Mutation

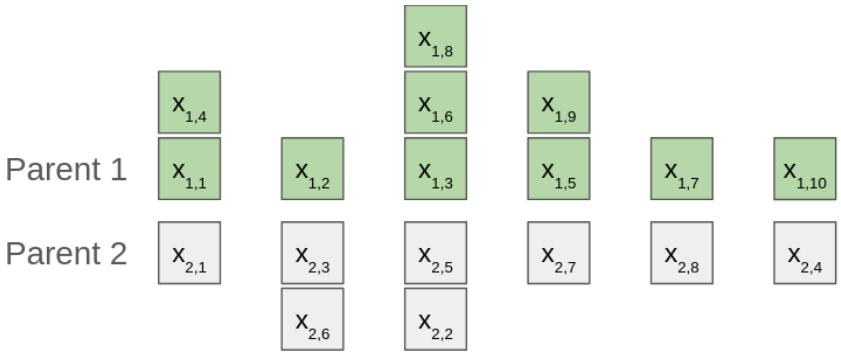
There are various mutation operators that can be applied. These include design variable mutation, metavariable-addition, metavariable deletion, and metavariable permutation. The operators can be used together where each operator is applied to the solution with a certain probability. [9]

3 Test problems

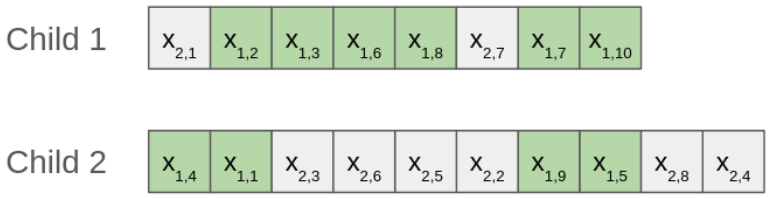
As described in Section 2, most work on variable-length optimization focuses on metameric problem representations. Consequently, it is difficult to find any binary optimization problem that is not metameric. We decided to adapt the OneMax problem since it is a simple test problem well suited for evaluating adapted classic binary evolutionary algorithms. For the continuous case, we chose to adapt the CMA-ES algorithm. To test whether the concept works, we chose a simple continuous optimization function: the FunctionMatch problem. In this section, we propose the adapted version of OneMax and describe the FunctionMatch problem.



(a)



(b)



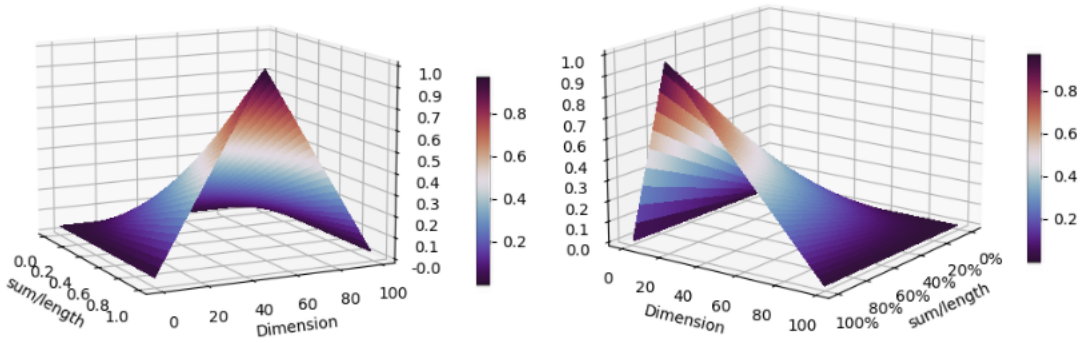
(c)

Figure 4: Example of a similarity crossover operation. 4a: each metavariable is linked to its most similar counterpart in the other parent-solution. The solid lines represent the links of each metavariable in *parent*1 to its most similar counterpart in *parent*2. The dotted lines represent the links from metavariables in *parent*2 to their most similar counterpart in *parent*1. These links do not have to mutual (e.g. both $x_{1,1}$ and $x_{1,4}$ are linked to $x_{2,1}$ but $x_{2,1}$ is linked only to $x_{1,1}$. 4b: metavariables form subgroups based on the links. 4c: a some of the subgroups are swapped out between the parents.

3.1 Variable-Length OneMax

The OneMax problem is a simple binary optimization problem that is defined as follows [2]:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i, \quad \mathbf{x} \in \{1,0\}^n \tag{5}$$



(a) $l^* = 50$ (b) $l^* = 20$

Figure 5: VL-OneMax with $l_{min} = 1$ and $l_{max} = 100$.

To test binary variable length optimization algorithms we need a version of the OneMax problem that takes two inputs: the solution $\mathbf{x}$ and the length of the solution $l(\mathbf{x})$. We will call this new problem VL-OneMax. The VL-OneMax function should return 1 if $l(\mathbf{x})$ equals the target length l^* and $x_i = 1$ for $i = 1, \ldots, l(\mathbf{x})$. First we define $d(\mathbf{x}, \mathbf{x}^*)$ to be the distance between $l(\mathbf{x})$ and l^*.

$$d(\mathbf{x}, \mathbf{x}^*) = |l^* - l(\mathbf{x})| \tag{6}$$

The VL-OneMax function as a product of two terms:

$$f(\mathbf{x}) = w_x \cdot s_x \tag{7}$$

The second term, s_x, is defined similar to the original OneMax problem (equation 5), but is multiplied by $1/n$. This way $0 \leqslant s_x \leqslant 1$. The equation for s_x is:

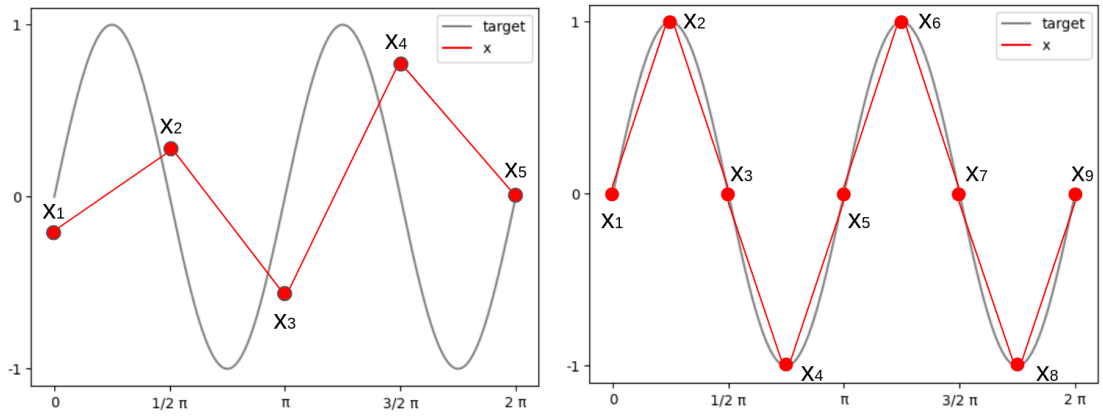
$$s_x = \frac{1}{n} \sum_{i=1}^{n} x_i, \quad \mathbf{x} \in \{1,0\}^n \tag{8}$$

The first term in equation 7 is defined as follows:

$$w_x = \begin{cases} 1 - \frac{d(\mathbf{x},\mathbf{x}^*)}{l^* - l_{min}} & \textbf{if} \quad l(\mathbf{x}) \leqslant l^* \\ \\ 1 - \frac{d(\mathbf{x},\mathbf{x}^*)}{l_{max} - l^*} & \textbf{if} \quad l(\mathbf{x}) > l^* \end{cases} \tag{9}$$

Here l_{min} and l_{max} are the minimum and maximum solution length allowed. If $d(\mathbf{x}, \mathbf{x}^*) = 0$ the weight $w_x = 1$. If $l(\mathbf{x}) = l_{min}$ or $l(\mathbf{x}) = l_{max}$ the weight will be $w_x = 0$.

3.2 FunctionMatch



(a) 5 variables (b) 9 variables

Figure 6: Example of a solution and a target function for Function-Match. The target-function is: $f(x) = \sin(2 \cdot x)$. The figure on the left shows a solution with a length of 5 variables, the figure on the right shows a solution of 9 variables long. A longer solution would allow for an even more accurate approximation of the target function.

The function-match problem consists of approximating the shape of a target function over a certain interval. An approximation of the target function consists of several points with various y-coordinates, equally distributed over the interval [4]. These points are connected by lines. The number of points can vary. The higher the number of points, the more finely grained the resulting approximation. The target function $f(x)$ is specified within the range $x_{min} \leqslant x \leqslant x_{max}$. The points in a solution are distributed equally along this range (e.g., for a solution of length 4, $x_{min} = 0$ and $x_{max} = 9$, the x-coordinates would be $0, 3, 6, 9$). The optimization involves finding the optimal y-coordinates for each point and the optimal number of points. Thus, a solution x consists of a list of variables, where each variable represents the y-coordinate of a point. In this work, the target-function used is $f(x) = a \cdot \sin(b \cdot x)$ where a is the amplitude of the sine wave, b is the frequency, and x is between 0 and 2π.

4 Fixed-dimensional pseudo-boolean and continuous algorithms

In this section, various binary evolutionary algorithms are described, taken from the IOHprofiler [2] benchmarking platform, which provides implementations of a number of classic optimization algorithms. In Section 5, we explain how these classic optimization algorithms can be adapted to deal with variable-length optimization problems.

4.1 Greedy Hill Climber

Algorithm 1 greedy Hill Climber [2]

Sample $\mathbf{x} \in \{0,1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$.
for t =1,2,3,... **do**
  $d \leftarrow 1 + (t \bmod n)$
  $\mathbf{x}^* \leftarrow \mathbf{x}$
  $\mathbf{x}^*_d \leftarrow 1 - \mathbf{x}^*_d$
  Evaluate $f(\mathbf{x}^*)$
  if $f(\mathbf{x}^*) \geqslant f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^*$
  end if
end for

The greedy Hill-Climber algorithm (gHC) works by flipping one bit each iteration. It starts by initializing $\mathbf{x} \leftarrow \{0,1\}^n$, where n is the solution-length. The algorithm goes through the solution one bit at a time from left to right. Each iteration, one bit is flipped to create a candidate-solution $\mathbf{x}^*$. The candidate solution $\mathbf{x}^*$ is compared to the old solution $\mathbf{x}$. If $\mathbf{x}^*$ has a higher fitness-value than $\mathbf{x}$, $\mathbf{x}$ is replaced by $\mathbf{x}^*$ [2].

4.2 Randomized Local Search

Algorithm 2 Randomized Local Search [2]

Sample $\mathbf{x} \in \{0,1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$.
for t =1,2,3,... **do**
  $\mathbf{x}^* \leftarrow \text{flip}_1(\mathbf{x}^*)$
  Evaluate $f(\mathbf{x}^*)$
  if $f(\mathbf{x}^*) \geqslant f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^*$
  end if
end for

Random Localized Search (RLS) is similar to the greedy Hill Climber algorithm, as it also flips one bit in each iteration. However, the bit-flip mutation is no longer applied to each variable in turn. Instead, in each iteration, the bit to be flipped is selected randomly [2].

4.3 $(1 + \lambda)$ EA with static mutation rates

Algorithm 3 $(1+\lambda)$ Evolutionary Algorithm with static mutation rates [2]

Sample $\mathbf{x} \in \{0,1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$.
for t =1,2,3,... **do**
  for i = 1,...,λ **do**
    Sample $l^{(i)} \sim \text{Bin}(n, \frac{1}{n})$
    Create $\mathbf{y}^{(i)} \leftarrow \text{flip}_{l^{(i)}}(\mathbf{x})$, and evaluate $f(\mathbf{y}^{(i)})$
  end for
  $\mathbf{x}^* \leftarrow \text{argmax}\{f(\mathbf{y}^{(1)}, ..., f(\mathbf{y}^{(\lambda)}\}$
  if $f^* \geqslant f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^*$
  end if
end for

The $(1 + \lambda)$ Evolutionary Algorithm with static mutation rates ($(1+\lambda)\text{EA}_{SMR}$) optimizes the solution $\mathbf{x}$ by mutating and evaluating λ new solutions $\mathbf{y}_0, \ldots, \mathbf{y}_\lambda$. The best new solution is selected as the candidate solution $\mathbf{y}^*$. If y^* has a higher fitness value than x, the old solution is discarded and $\mathbf{x} \leftarrow \mathbf{y}^*$ [2]. For the mutation step, the algorithm uses the $flip_l$-operator, as given in 4.

Algorithm 4 $\text{flip}_l(x)$ operation [2]

Input $\mathbf{x} \in \{0,1\}^n$
  $l \in \mathbb{N}$
Randomly select l bits $\{i_1, ..., i_l\} \in \mathbb{N}$ to be mutated
$\mathbf{y} \leftarrow \mathbf{x}$
for $i_0, ..., i_l$ **do** $\mathbf{y}_i \leftarrow 1 - \mathbf{x}_i$

The $flip_l$ operator randomly selects l bits and flips them. EA_{SMR} samples the mutation strength l for each iteration from a binomial distribution [2].

4.4 fGA

Algorithm 5 fast Genetic Algorithm [2]

Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$.
for t =1,2,3,... **do**
  for i = 1,...,λ **do**
    Sample $l^{(i)} \sim D^{\beta}_{n/2}$
    Create $\mathbf{y}^{(i)} \leftarrow \text{flip}_{l^{(i)}}(\mathbf{x})$, and evaluate $f(\mathbf{y}^{(i)})$
  end for
  $\mathbf{x}^* \leftarrow \text{argmax}\{f(\mathbf{y}^{(1)}, ..., f(\mathbf{y}^{(\lambda)}\}$
  if $f^* \geqslant f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^*$
  end if
end for

The fast Genetic Algorithm (fGA) also utilizes the $flip_l$ operator to generate λ new solutions. The mutation strength is sampled from the power law $D^{\beta}_{n/2}$, where n represents the length of the solution. Doer et al. [2] used $\beta = 1.5$. This means that large mutation strengths are used frequently, while small mutation strengths are still used reasonably often [2].

4.5 Two-rate $(1 + \lambda)$ EA with adaptive mutation rates

Algorithm 6 The two-rate $(1 + \lambda)$ EA with adaptive mutation rates [2]

Sample $\mathbf{x} \in \{0, 1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$.
for t =1,2,3,... **do**
  for i = 1,...,$\lambda/2$ **do**
    Sample $l^{(i)} \sim \text{Bin}_{>0}(n, r/(2n))$
    Create $y^{(i)} \leftarrow \text{flip}_{l^{(i)}}(\mathbf{x})$, and evaluate $f(\mathbf{y}^{(i)})$
  end for
  for $i = \lambda/2 + 1$,...,λ **do**
    Sample $l^{(i)} \sim \text{Bin}_{>0}(n, 2r/n)$
    Create $\mathbf{y}^{(i)} \leftarrow \text{flip}_{l^{(i)}}(\mathbf{x})$, and evaluate $f(\mathbf{y}^{(i)})$
  end for
  $\mathbf{x}^* \leftarrow \text{argmax}\{f(\mathbf{y}^{(1)}, ..., f(\mathbf{y}^{(\lambda)}\}$
  if $f^* \geqslant f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^*$
  end if
  if $\mathbf{x}^*$ has been created with mutation rate $r/2$ **then** $s \leftarrow 3/4$ **else** $s \leftarrow 1/4$
  Sample $q \in$ [0,1] uniformly at random.
  if $q \leqslant s$ **then** $r \leftarrow \max\{r/2, 2\}$ **else** $r \leftarrow \min\{2r, n/4\}$
end for

The two-rate $(1 + \lambda)$ EA with adaptive mutation rates uses two different mutation rates to create new solutions. Half of the offspring are created using a mutation rate of $r/2n$, while the other half are created using a mutation rate of $2r/n$. The variable r is updated after each iteration. There are two possible updates: $r \leftarrow \max\{r/2, 2\}$ and $r \leftarrow \min\{2r, n/4\}$. Which of the two updates occurs is decided by a random decision, which is biased toward the mutation rate that was responsible for creating the best-performing offspring [2].

4.6 $(1+\lambda)$ $\mathbf{EA}_{norm}$

Algorithm 7 $(1+\lambda)$ EA_{norm} [2]

Sample $\mathbf{x} \in \{0,1\}^n$ uniformly at random $f(\mathbf{x})$
for t =1,2,3,... **do**
  for $i = 1, ..., \lambda$ **do**
    $l^{(i)} \sim \min\{N_{>0}(r, r(1-r/n)), n\}$
    Create $\mathbf{y}^{(i)} \leftarrow \mathrm{flip}_{l^{(i)}}(\mathbf{x})$ and evaluate $f(\mathbf{y}(i))$
  end for
  $i \leftarrow \min\{\ j \mid f(\mathbf{y}^{(j)}) = \max\{f(\mathbf{y}^{(k)}) \mid k \in [\lambda]\}\}$
  $r \leftarrow l^{(i)}$
  if $f(\mathbf{y}^{(i)}) \geqslant f(\mathbf{x})$ **then** $\mathbf{x} \leftarrow \mathbf{y}^{(i)}$
end for

The $(1+\lambda)$ EA with normalized mutation rate samples the mutation strength from a normal distribution with mean r and variance $r(1-r/n)$. The variable r is updated in each iteration according to the mutation strength used to create the best-performing offspring [2]. The pseudocode is provided in 7.

4.7 $(1+\lambda)$ $\mathbf{EA}_{var}$

Algorithm 8 $(1+\lambda)$ EA_{var} [2]

Sample $\mathbf{x} \in \{0,1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$
for $t = 1, 2, 3, ...$ **do**
  for $i = 1, ..., \lambda$ **do**
    $l^{(i)} \sim \min\{N_{>0}(r, F^c r(1-r/n)), n\}$
    Create $\mathbf{y}^{(i)} \leftarrow \mathrm{flip}_{l^{(i)}}(\mathbf{x})$ and evaluate $f(\mathbf{y}(i))$
  end for
  $i \leftarrow \min\{\ j \mid f(y^{(j)}) = \max\{f(\mathbf{y}^{(k)}) \mid k \in [\lambda]\}\}$
  if $r = l^{(i)}$ **then** $c \leftarrow c+1$ **else** $c \leftarrow 0$
  $r \leftarrow l^{(i)}$
  if $f(\mathbf{y}^{(i)}) \geqslant f(\mathbf{x})$ **then** $\mathbf{x} \leftarrow \mathbf{y}^{(i)}$
  $i \leftarrow \min\{\ j \mid f(\mathbf{y}^{(j)}) = \max\{f(\mathbf{y}^{(k)}) \mid k \in [\lambda]\}\}$
  if $r = l^{(i)}$ **then** $c \leftarrow c+1$ **else** $c \leftarrow 0$
  $r \leftarrow l^{(i)}$
  if $f(\mathbf{y}^{(i)}) \geqslant f(\mathbf{x})$ **then** $\mathbf{x} \leftarrow \mathbf{y}^{(i)}$
end for

The $(1+\lambda)$ EA with normalized standard bit mutation and controlled variance ($(1+\lambda)$ EA_{var}) is similar to EA_{norm}. However, it adapts not only the mean r but also the variance during the optimization process [2].

4.8 $(1+\lambda)$ $\mathbf{EA}_{log-n}$

Algorithm 9 $1{+}\lambda$ EA_{log-n} [2]

Sample $\mathbf{x} \in \{0,1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$
for $t = 1, 2, 3, ...$ **do**
  for $i = 1, ..., \lambda$ **do**
    $p^{(i)} = (1 + \frac{1-p}{p} \cdot \exp(0.22 \cdot \mathcal{N}(0,1)))^{-1}$
    $l^{(i)} \sim \mathrm{bin}_{>0}(n, p^{(i)})$
    Create $y^{(i)} \leftarrow \mathrm{flip}_{l^{(i)}}(\mathbf{x})$ and evaluate $f(\mathbf{y}(i))$
  end for
  $i \leftarrow \min\{j | f(\mathbf{y}^{(j)}) = \max\{f(\mathbf{y}^{(k)}) | k \in [\mathbf{y}]\}\}$
  $p \leftarrow p^{(i)}$
  $\mathbf{x}^* \leftarrow \mathrm{argmax}\ \{f(\mathbf{y}^{(1)}), ..., f(\mathbf{y}^{(\lambda)})\}$
  if $f(\mathbf{y}^{(i)}) \geqslant f(\mathbf{x})$ **then** $\mathbf{x} \leftarrow \mathbf{y}^{(i)}$
end for

The EA_{log-n} uses log-normal self-adaptation on the mutation strength [2].

4.9 Self-adjusting $(1 + (\lambda, \lambda))$ GA

Algorithm 10 self-adjusting $(1 + (\lambda, \lambda))$ GA [2]

Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$
for t =1,2,3,... **do**
  for $i = 1, ..., \lambda$ **do**
    $l^{(i)} \sim \mathrm{bin}_{>0}(n, \lambda/n)$
    Create $y^{(i)} \leftarrow \mathrm{flip}_{l^{(i)}}(\mathrm{x})$ and evaluate $f(y(i))$
    $x^* \leftarrow \mathrm{argmax}\{f(y^{(i)}), ..., f(y^{(i)})\}$
  end for
  for $i = 1, ..., \lambda$ **do**
    create $y^{(i)} \leftarrow cross_c(x, x*)$ and evaluate $f(y(i))$
    $y^* \leftarrow \mathrm{argmax}\{f(y^{(i)}), ..., f(y^{(i)})\}$
  end for
  for $i = 1, ..., \lambda$ **do**
    if $f(y^*) > f(x)$ **then** $x \leftarrow y^*; \lambda \leftarrow \max\{\lambda/F, 1\}$
    if $f(y^*) = f(x)$ **then** $x \leftarrow y^*; \lambda \leftarrow \min\{\lambda F^{1/4}, n\}$
    if $f(y^*) < f(x)$ **then** $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$
  end for
end for

The self-adjusting $(1 + (\lambda, \lambda))$ Genetic Algorithm (saGA) adapts the number of offspring λ after each iteration based on whether better-performing solutions are found. For the mutation step the flip_l operator is used with mutation strength $l^{(i)} \sim \mathrm{bin}_{>0}(n, \lambda/n)$ The saGA algorithm also uses crossover. In the crossover step, λ offspring are created by repeatedly recombining x and x^*. Here, x^* is the best solution that was generated during the mutation step. For the crossover phase, the algorithm uses the biased crossover operator cross_c 11.

Algorithm 11 $cross_c(\mathbf{x}, \mathbf{x}^*)$ [2]

$\mathbf{y} \leftarrow \mathbf{x}$
Sample $l \sim \text{Bin}_{>0}(n, c)$
select l different positions $\{i_1, ..., i_l\} \in [n]$
for $j \in l$ **do** $y_j \leftarrow x^*_j$

The $cross_c$ operator produces one offspring. First, the original solution x is copied ($y \leftarrow x$). Then l positions are randomly selected. For each of these positions, the value is replaced with the value of x^*. The bias value c determines the probability of each bit being selected for crossover. The offspring is evaluated, and the best one, y^*, is compared with x. If an improved offspring has been created, λ is decreased; if not, λ is increased [2].

4.10 vGA

Algorithm 12 "vanilla" Genetic Algorithm [2]

for $i = 1, ..., \mu$ **do**
  Sample $x^{(i)} \in \{0, 1\}^n$ uniformly at random
end for
for t =1,2,3,... **do**
  Apply roulette-wheel selection to $\{x^{(1)}, ..., x^{\mu}\}$ to select μ parent individuals $\{y^{(1)}, ..., y^{(\mu)}\}$
  for $i = 1, ..., \mu/2$ **do**
    with probability p_c apply one-point crossover to $y^{(i)}$ and $y^{(2i)}$ at random crossover point $j \in [n]$
  end for
  for $i = 1, ..., \mu$ **do**
    Sample $l^{(i)} \sim \text{Bin}(n, p_m)$
    $y^{(i)} \leftarrow \text{flip}_{l^{(i)}}(y^{(i)})$
    Evaluate $f(y^{(i)})$
  end for
  for $i = 1, ..., \mu$ **do**
    Replace $x^{(i)}$ by $y^{(i)}$
  end for
end for

The vanilla Genetic Algorithm (vGA) starts with a population of μ individuals. It has a selection, a crossover and a mutation step. The algorithm uses roulette-wheel selection, allowing individuals to be selected multiple times. One-point crossover is applied to $\mu/2$ pairs of individuals. The resulting individuals are then mutated using the flip$_l$-operator 4, where each bit has a probability of $p_m = 2/n$ of being flipped [2].

4.11 CMA-ES

Lastly, we will consider a popular algorithm for continuous optimization problems. The Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) samples candidate solutions from a multivariate normal distribution [6]. The multivariate probability distribution is specified by

the mean **m** and covariance matrix C. During the optimization process, the covariance matrix and mean of the distribution are adapted in such a way that the sampled solutions become optimal (see algorithm 13). The algorithm works as follows: First, λ individuals are sampled from the probability distribution. An individual k of generation $g+1$ is given by:

$$x_k^{g+1} \sim \mathbf{m}^{(g)} + \sigma^{(g)}\mathbf{y}_k \qquad \text{for } k = 1, ..., \lambda \tag{10}$$

Where $\mathbf{m}^{(g)}$ is the mean of the population at generation g, $\sigma^{(g)}$ is the overall standard deviation or step size, and $\mathbf{y}_k \sim \mathcal{N}(0, C^{(g)})$ is the mutation direction sampled from a normal distribution with zero mean and covariance matrix $C^{(g)}$. The variable λ represents the population size [3]. In the selection step, the candidate solutions are ranked based on their fitness scores. The best μ candidate solutions are selected. The next step is the recombination step. The mean is updated using the weighted average of the means of the μ best solutions. The mean of the next generation $m^{(g+1)}$ is given by:

$$m^{(g+1)} = m^g + c_m \sum_{i=1}^{\mu} w_i (x_{i:\lambda}^{(g+1)} - m^g) \tag{11}$$

Where c_m is the learning-rate. The index $(i : \lambda)$ denotes the i^{th} best point, such that $f(\mathbf{x}_{i:\lambda}) \leqslant f(\mathbf{x}_{i+1:\lambda}) \leqslant, ..., \leqslant f(\mathbf{x}_{\lambda:\lambda})$ [3]. In the last step the step-size and the covariance matrix are adapted. For both updates CMA-ES uses so-called evolution paths. The evolution paths accumulate search-directions from previous steps in the optimization process [6]. CMA-ES constructs two separate evolution paths. One, p_σ, is used for step-size adaptation, while the other, p_c is used for updating the covariance-matrix. Both are updated each iteration. The update for p_σ is given by:

$$p^\sigma = (1 - c_\sigma)p^\sigma + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}\, C_g^{-1/2} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{12}$$

Where (1-c_σ) and ($\sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}$) are normalization terms.

$$C_g^{-1/2} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{13}$$

Represents the direction of the current search-step. μ_{eff} is the variance selective sample mass given by $\mu_{eff} = (\sum_{i=1}^{\mu} w_i^2)^{-1}$ [3]. The evolution path p_σ exploits correlations between previous search directions. If most previous search steps were in a similar direction, the step-size is likely to increase. If there is little correlation among previous search-steps, the step-size tends to decrease. The adaptation of the covariance matrix is done with both a rank-one and a rank-μ update. The rank-μ update uses the μ best candidate solutions to estimate the parameters of $C^{(g+1)}$. The rank-μ update is given by:

$$C^{(g+1)} = (1 - c_\mu)C^{(g)} + c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \mathbf{y}_{i:\lambda}^{(g+1)^\top} \tag{14}$$

Where c_μ is the learning rate for the rank-μ update of the covariance matrix, and **y** is the mutation direction (see 10). The rank-1 update uses the evolution path p_c to update the covariance-matrix. One of the reasons for this is that, for a given mutation direction **y**, the outer product $\mathbf{y}\mathbf{y}^\top = -\mathbf{y}(-\mathbf{y})^\top$. This means that the sign information is lost when calculating

the covariance matrix $C^{(g+1)}$. To preserve this information, the evolution path p_c is used. The update for p_c is given by:

$$p_c^{g+1} = (1 - c_c)p_c^{(g)} + \sqrt{c_c(2 - c_c)\mu_{eff}}\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{15}$$

Where c_c is the decay-rate [3]. The rank-1 and rank-μ updates are then combined in the update rule for the covariance matrix C [3]:

$$C^{(g+1)} = (1 + c_1\delta(h_\sigma)) - c_1 - c_\mu \sum_{i=1}^{\lambda} w_i)\mathbf{C} + c_1\mathbf{p}_c\mathbf{p}_c^\top + c_\mu \sum_{i=1}^{\mu} w_i\mathbf{y}_{i:\lambda}^{(g+1)}\mathbf{y}_{i:\lambda}^{(g+1)^\top} \tag{16}$$

Algorithm 13 CMA-ES [3]

Set Parameters
Set parameters $\lambda, w_{i=1\ldots\lambda}c_m, c_\sigma, d_\sigma, c_c, c_1$ and c_μ [3]
Initialization
$p_\sigma = 0, p_c = 0, C = I$ and $g = 0$
Choose $m \in \mathbb{R}^n$ and $\sigma \in \mathbb{R}_{>0}$
while termination criteria not met **do**
Sample new population of search points, for $k = 1, .., \lambda$
$z_k \sim \mathcal{N}(0, I)$
$\mathbf{y}_k = \mathbf{B}\mathbf{D}z_k \sim \mathcal{N}(0, C)$
$\mathbf{x}_k = \mathbf{m} + \sigma\mathbf{y}_k \sim \mathcal{N}(m, \sigma^2 C)$
selection and recombination
$\mathbf{m} \leftarrow \mathbf{m} + c_m\sigma \sum_{i=1}^{\mu} w_i\mathbf{y}_{i:\lambda}$
Step-size control
$\mathbf{p}_\sigma \leftarrow (1 - c_m)\mathbf{p}_\sigma + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}\mathbf{C}^{-\frac{1}{2}}$
$\sigma \leftarrow \sigma \cdot \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{p}_\sigma\|}{E\|\mathcal{N}(0,I)\|} - 1\right)\right)$
Covariance-matrix adaption
$\mathbf{p}_c \leftarrow (1 - c_c) + h_c\sqrt{c_c(2 - c_\sigma)\mu_{eff}}\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$
$w_i^\circ = w_i \cdot (1$ **if** $w_i \geqslant 0$ **else** $n/\|\mathbf{C}^{-\frac{1}{2}}\mathbf{y}_{i:\lambda}\|^2)$
$\mathbf{C} \leftarrow (1 + c_1\delta(h_\sigma) - c_1 - c_\mu \sum w_i)\mathbf{C} + c_1\mathbf{p}_c\mathbf{p}_c^\top + c_\mu \sum_{i=1}^{\lambda} w_i^\circ\mathbf{y}_{i:\lambda}\mathbf{y}_{i:\lambda}^\top$
end while

5 Variable-dimensional algorithms

Due to the similarity between some of the algorithms described in Section 4, most of them can be adapted in the same way. The first thing to consider is the initialization step. Because the dimensionality is no longer predetermined, it becomes necessary to specify an initial solution length n. There are different options. n can be initialized as $\frac{1}{2}(n_{max} - n_{min})$, or it could be selected randomly from $[n_{min}, n_{max}]$. In this work, the latter option is used.

5.1 Variable-Length greedy Hill Climber

To adapt the gHC algorithm, the mutation step is changed. Instead of only flipping one bit in each iteration, the selected bit can now be flipped, copied, or deleted. The variables p_{dup},

p_{del}, and p_m represent the probabilities of a duplication, a deletion, and a bit-flip mutation, respectively.

5.2 Variable-Length Random Local Search

Since the LRS algorithm uses the $flip_l$-operator (see algorithm 4) with solution strength $l = 1$, it is necessary to adapt this operator. In the case of LRS, where $l = 1$, this is quite straightforward. Two extra options are added (*i.e.* deleting and duplicating). Based on the specified probabilities p_m, p_{dup} and p_{del} the randomly selected bit is flipped, duplicated, or deleted. In the general case where $l \geqslant 1$, we must consider whether we choose to perform one type of mutation for all l bits, or decide for each bit separately which type of mutation will be used. In the latter case, there may be multiple bit-flips, bit-duplications, and bit-deletions in the same mutation step. The pseudocode for the length-varying version of flip$_l$ (VL-flip$_l$) is given below 14.

Algorithm 14 VL-flip$_l(\mathbf{x})$ operation

Input $\mathbf{x} \in \{0,1\}^n$
  $l \in \mathbb{N}$ ▷ number of bits to be flipped
Randomly select l bits $\{i_1, ..., i_l\} \in \mathbb{N}$,
$\mathbf{y} \leftarrow \mathbf{x}$
for all $j \in \{i_1, ..., i_l\}$ **do**
  $m \sim U(0,1)$
  if $m < p_m$ **then**
    $y_j \leftarrow 1 - x_j$
  else if $m < p_{del}$ **then**
    Delete bit y_l.
  else if $m < p_{dup}$ **then**
    Duplicate bit y_l.
  end if
end for

5.3 Various Variable-Length Evolutionary Algorithms

To adapt fGA, two-rate EA with adaptive mutation rates, EA$_{norm}$, EA$_{var}$, and EA$_{log-n}$, the only requirement is to replace the flip$_l$ operator with the VL-flip$_l$ operator 14.

5.4 Variable-Length self-adjusting Genetic Algorithm

saGA can be adapted by replacing the mutation operator with VL-flip$_l$ 14. The crossover operator must also be adapted. It can either be replaced with a length-varying operator (for this work, cut-and-splice crossover is used) or adapted to handle solutions of unequal lengths whilst remaining a fixed-length crossover operator. The cut-and-splice operator recombines x and x^* by randomly choosing a different crossover point for each solution. This results in two offspring instead of the one offspring created by the cross$_c$-operator. To deal with this problem, the cut-and-splice operator was slightly changed. The operator randomly chooses one of the offspring and returns it, discarding the other.

Algorithm 15 $cut - and - splice(\mathbf{x}, \mathbf{x}^*)$

Randomly select crossover point p_1 on $\mathbf{x}$ and p_2 on $\mathbf{x}*$
$\mathbf{y}1 \leftarrow \mathbf{x}[0 : p1] + \mathbf{x}^*[p2 :]$
$\mathbf{y}2 \leftarrow \mathbf{x}^*[0 : p2] + \mathbf{x}[p1 :]$

The other option would be to use the original cross$_c$-operator. In that case, the cross$_c$-operator has to be adapted to account for x and x^* not being the same length. The number of bits n from which to sample is set to the length of the smallest solution. This results in the following operator:

Algorithm 16 VL-cross$_c(\mathbf{x}, \mathbf{x}^*)$

$\mathbf{y} \leftarrow \mathbf{x}^*$
$n \leftarrow \min(|\mathbf{x}|, |\mathbf{x}^*|)$
Sample $l \sim \text{Bin}_{>0}(n, c)$
select l different positions $\{i_1, ..., i_l\} \in [n]$
for $j \in l$ **do** $y_j \leftarrow x^*_j$

In Section 6, we will experiment with both versions.

5.5 VL-vGA

The variable length version of this algorithm uses tournament-selection instead of roulette-wheel selection. As mentioned in 2.2, tournament-selection with small tournament sizes can already help to ensure a certain measure of length diversity in the population. The mutation operator flip$_l$ is again replaced by VL-flip$_l$. The crossover method could also be changed to cut-and-splice crossover. In Section 6.4, both the version using one-point crossover and the version using cut-and-splice crossover are evaluated.

5.6 VL-CMA-ES

Since the population is sampled from a multivariate distribution, which is determined by the Covariance Matrix $\mathbf{C}$ and the mean μ, duplicating a variable involves expending the Covariance Matrix and the mean vector. If we want to duplicate the i^{th} variable x_i to create x^*_i, we need to duplicate the corresponding mean and variance. The covariance values are not copied, but initialized at zero. Deleting a variable is more straightforward. It means deleting the corresponding row and column from the covariance matrix and deleting the corresponding mean from the mean vector. After a deletion or duplication, a few variables have to be adapted. Constants that depend on the solution-length are re-initialized. The matrices B and D are adapted by performing eigen-decomposition on the new covariance matrix. The mutation step can be either the normal mutation, a deletion, or a duplication. The duplication operation is applied with probability p_{dup} and the deletion with probability p_{del}. However, should the duplication and deletion probabilities remain constant? Unlike the VL-vGA algorithm, all solutions in a population will have the same length.
In VL-vGA the selection-operator can deal with detrimental length changes. Solutions who's fitness deteriorated after a change in length will likely not be selected into the next generation. In VL-CMA-ES, since the dimensionality changes for all solutions at the same time, the selection-operator cannot influence the dimensionality of the next population in this way.

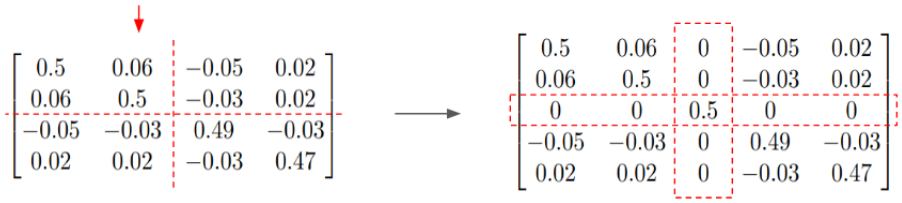


Figure 7: Example of a duplication of the second column and second row of the covariance matrix. As can be seen, only the variance is copied; all the covariance-values of the new variable are set to zero.

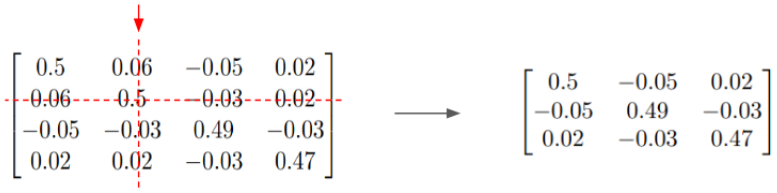


Figure 8: Example of the deletion of the second column and second row of the covariance matrix.

Regardless of which solutions are selected the dimensionality of the next population will not change. The only way of reversing a detrimental change in dimensionality is by another change in length, determined by the probabilities p_{dup} and p_{del}. This can prevent the algorithm from converging, since length variations can occur at any point in the optimization process. It is true for all VL-EAs, that length changes can occur at any point during optimization, but there the selection-operator can discard detrimental length changes. When an optimal solution-length has been found, most length-changes are detrimental and will result in solutions with lower fitness scores. These solutions will not be selected. This way, unlike VL-CMA-ES, the process can converge. There are different ways to deal with this problem. One option is, to let the duplication probability p_{dup} and the deletion probability p_{del} decay over time. This way, the search for the correct solution length is essentially a random search during the first part of the optimization-process. Another option is to adjust the probabilities p_{dup} and p_{del} based on the resulting fitness-values. If, after a duplication, the fitness of the highest-ranked individual is higher than that of the highest-ranked individual from the previous generation, p_{dup} is increased. If the fitness value is lower, the fitness value is decreased. Combining these two approaches would result in the following update:

$$p_{dup} = \begin{cases} (p_{dup} + a) \cdot p_{decay} & \textbf{if} \quad \mathcal{L}(\mathbf{x}') > \mathcal{L}(\mathbf{x}) \\ \\ (p_{dup} - a) \cdot p_{decay} & \textbf{if} \quad \mathcal{L}(\mathbf{x}') < \mathcal{L}(\mathbf{x}) \end{cases} \tag{17}$$

Where $\mathcal{L}$ is the loss that is to be minimized. For the FunctionMatch-problem this is the Mean Squared Error (MSE). A similar update is used for adapting the deletion probability p_{del}.

6 Experiments

6.1 Comparing VL-gHC and VL-LRS

In the first experiment, the variable-length version of the genetic Hill Climber algorithm is compared to Variable-Length Localized Random Search. The initial solution length l_{init} is randomly sampled between the minimum and maximum lengths $(l_{\text{min}}, l_{\text{max}})$. For all experiments, $l_{\text{min}} = 1$ and $l_{\text{max}} = 1000$.

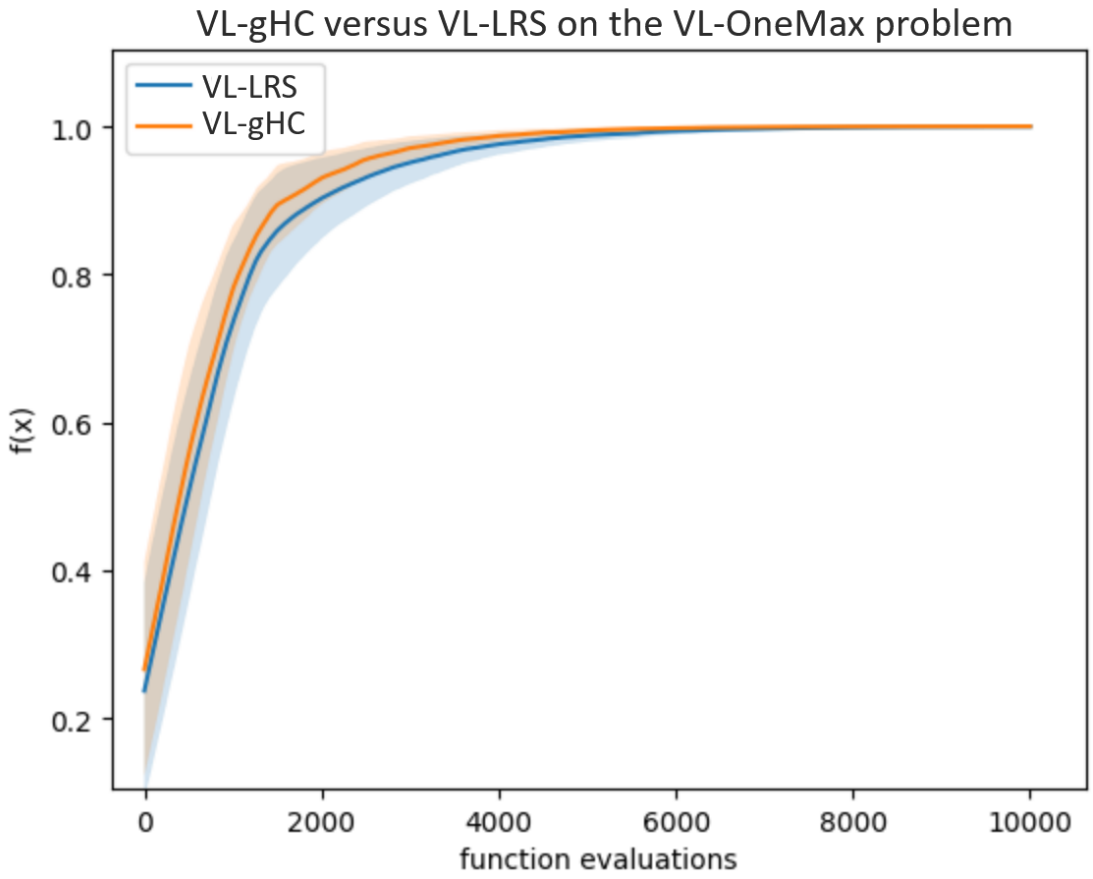


Figure 9: The VL-gHC algorithm and the VL-LRS algorithm run on the VL-OneMax problem with target length $l^* = 500$. All algorithms have been run with an evaluation budget of 10.000. For each algorithm, the results have been averaged over 100 random seeds. The standard-deviation is represented by the opaque areas. Both algorithms use the same mutation, duplication, and deletion probabilities: $p_{dup} = p_{del} = p_m = 1/3$.

6.1.1 Results

Both algorithms consistently find the optimal solution. VL-gHC slightly outperforms VL-LRS. The only difference between the two algorithms is that VL-LRS chooses the bit to be mutated randomly, whereas VL-gHC goes through the solution bit by bit. For this optimization problem, that might be an advantage. However, there appears to be no significant performance difference between the two algorithms.

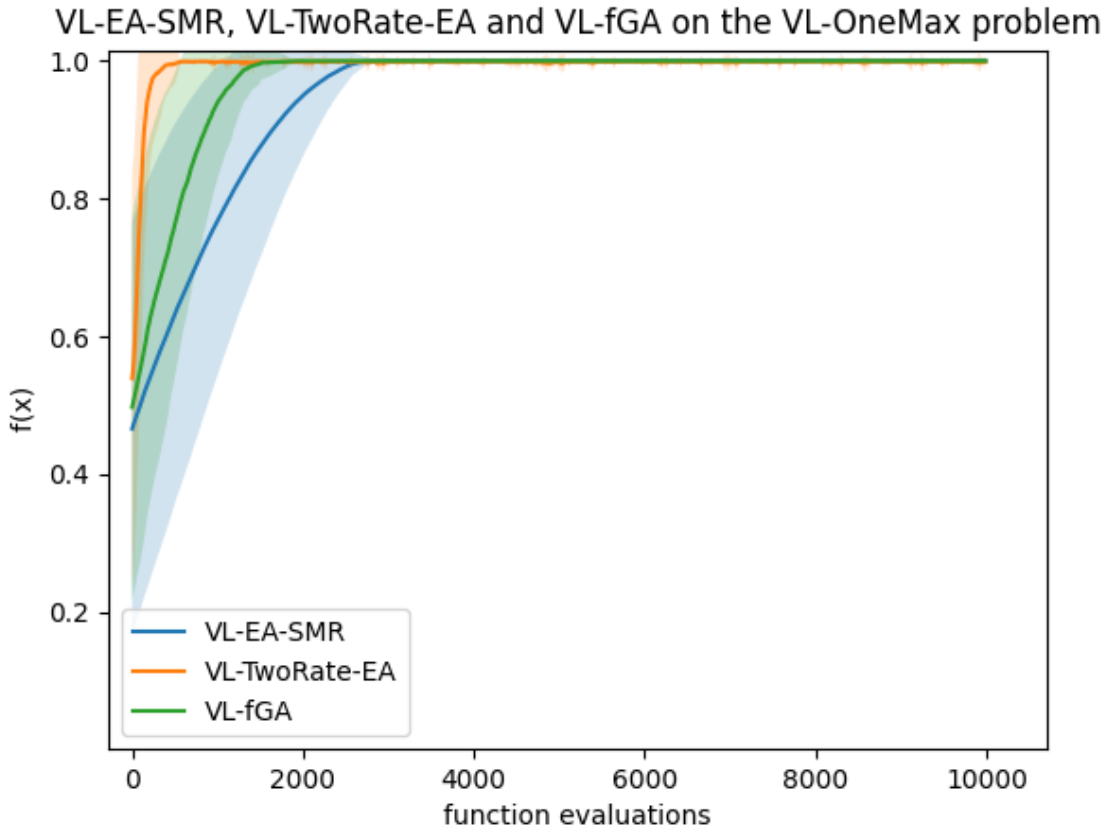


Figure 10: VL-TwoRate-EA and VL-EA$_{SMR}$ and VL-fGA on the VL-OneMax problem using target length $l^* = 500$. All algorithms have been run with an evaluation budget of 10,000. For each algorithm, the results have been averaged over 100 random seeds. The standard-deviation is represented by the opaque areas. All algorithms use $\lambda = 10$ and the same mutation, duplication, and deletion probabilities: $p_{dup} = p_{del} = p_m = 1/3$.

6.2 Comparing VL-EA algorithms

Next, we consider VL-(1+λ) EA$_{\text{norm}}$, VL-(1+λ) EA$_{\text{log-n}}$, and VL-(1+λ) EA$_{\text{var}}$. All algorithms run with $\lambda = 10$. The initial length l_{init} is again sampled at random.

6.2.1 Results

Figure 10 shows the performance of VL-EA$_{\text{SMR}}$, VL-TwoRate-EA, and VL-fGA. The Algorithm that performed best is VL-TwoRate-EA. Unlike VL-EA-SMR, it does not have one standard mutation rate but can switch between two different mutation rates. This probably enables the algorithm to learn faster than the other two on the VL-OneMax problem. VL-EA-SMR requires more function evaluations to find the optimal solution. This could be because it uses a standard mutation rate. Initially, a larger mutation rate can be beneficial as it allows for greater exploration. As mentioned in section 4.4, using the power law with $\beta = 1.5$ results in higher mutation strengths occurring more often than smaller mutation strengths. VL-fGA outperforms VL-EA-SMR but does not perform as well as VL-TwoRate-EA. As observed in Figure 11, VL-(1+λ) EA$_{\text{log-n}}$ and VL-(1+λ) EA$_{\text{var}}$ outperform the VL-EA$_{\text{norm}}$ algorithm, demonstrating that

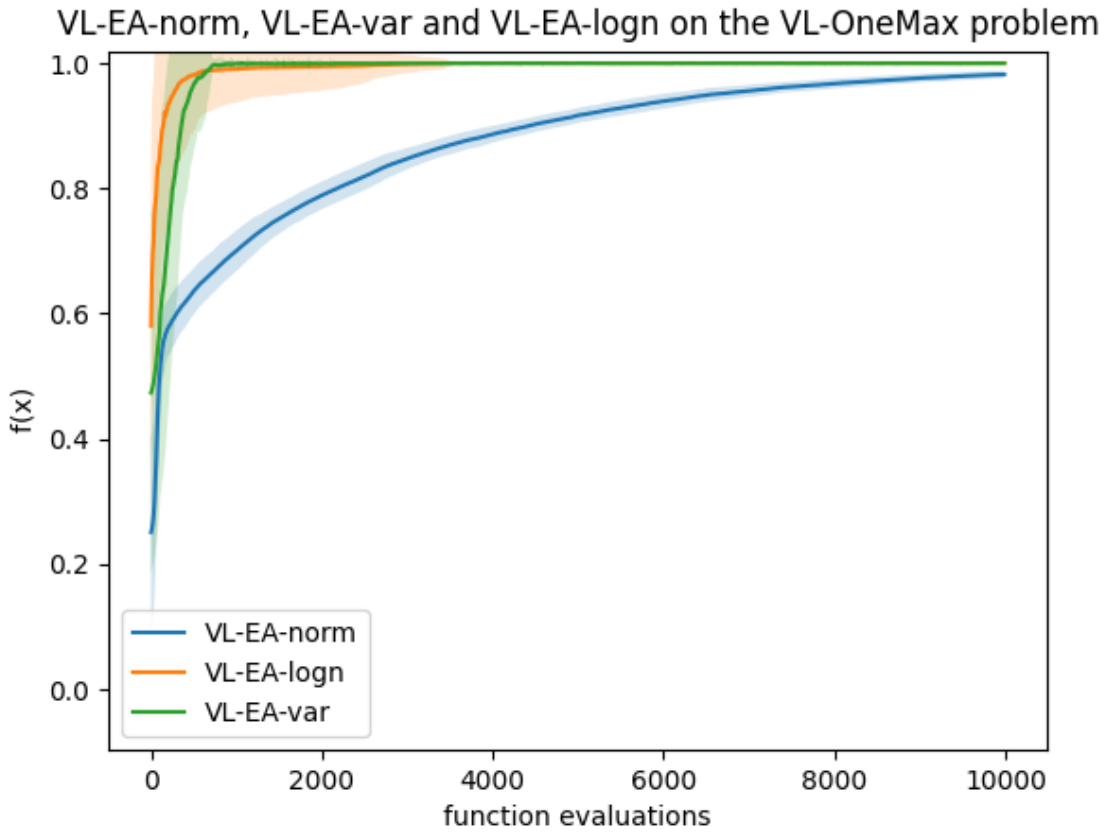


Figure 11: VL-EA$_{norm}$, VL-EA$_{log-n}$, and VL-EA$_{var}$ on the VL-OneMax problem with $l^* = 500$. Results have been averaged over 100 random seeds. The standard-deviation is represented by the opaque areas. For all three algorithms $p_{del} = p_{dup} = p_m = 1/3$. For V:-EA-norm: $r_0 = 1.5$.

these algorithms better balance exploitation and exploration.

6.3 Comparing VL-saGA and VL-vGA

Next, the Variable-Length self-Adjusting $(1+(\lambda,\lambda))$ GA is compared with the Variable Length (μ,λ) vGA. For both algorithms $\lambda = 10$ and $\mu = 10$. VL-saGA uses the cross$_c$ operator and VL-vGA uses one-point crossover. All algorithms have been run on an evaluation budget of 10,000.

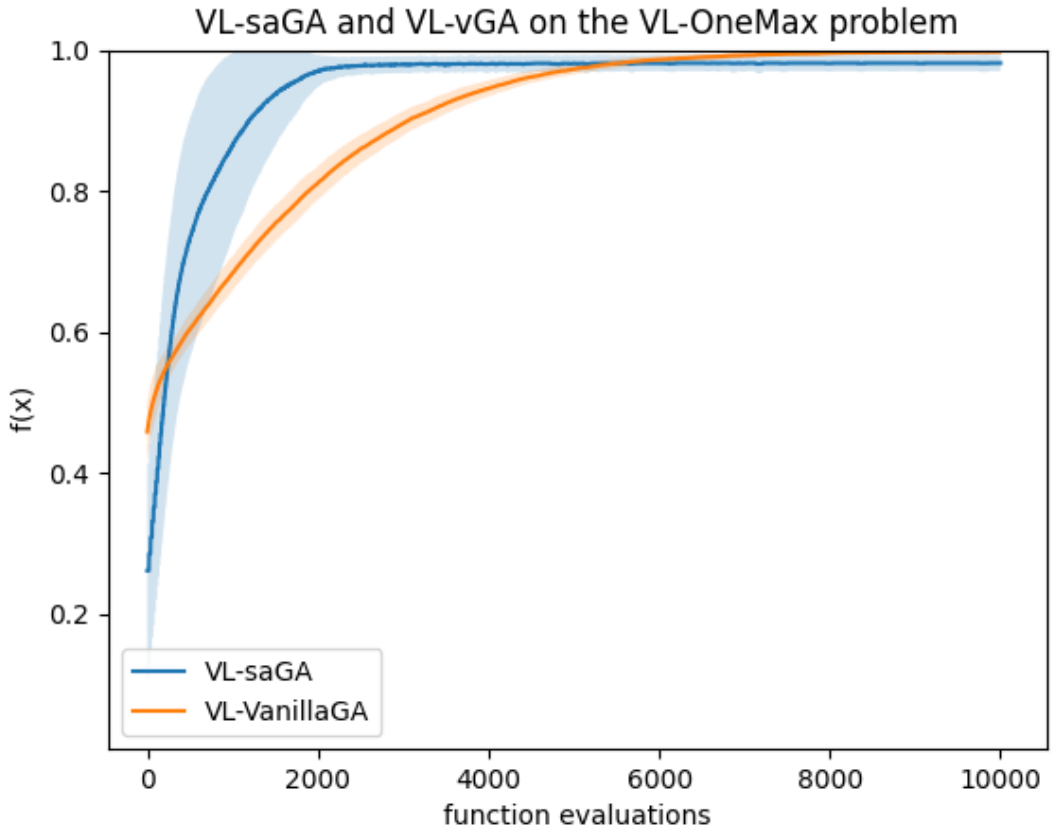


Figure 12: The VL-fGA and VL-saGA run on the VL-OneMax problem with target length $l^* = 500$. The results are averaged over 100 random seeds. The standard-deviation is represented by the opaque areas. For both algorithms the crossover and mutation probabilities have been optimized. For VL-saGA: $p_{del} = p_{dup} = 1/3$, $p_m = 1/500$, the crossover-strength $c = 0.1$ and $F_{init} = 1.1$. For VL-vGA: $p_{del} = p_{dup} = 1/3$ and $p_m = 1/100$ and the crossover probability $p_c = 0.7$.

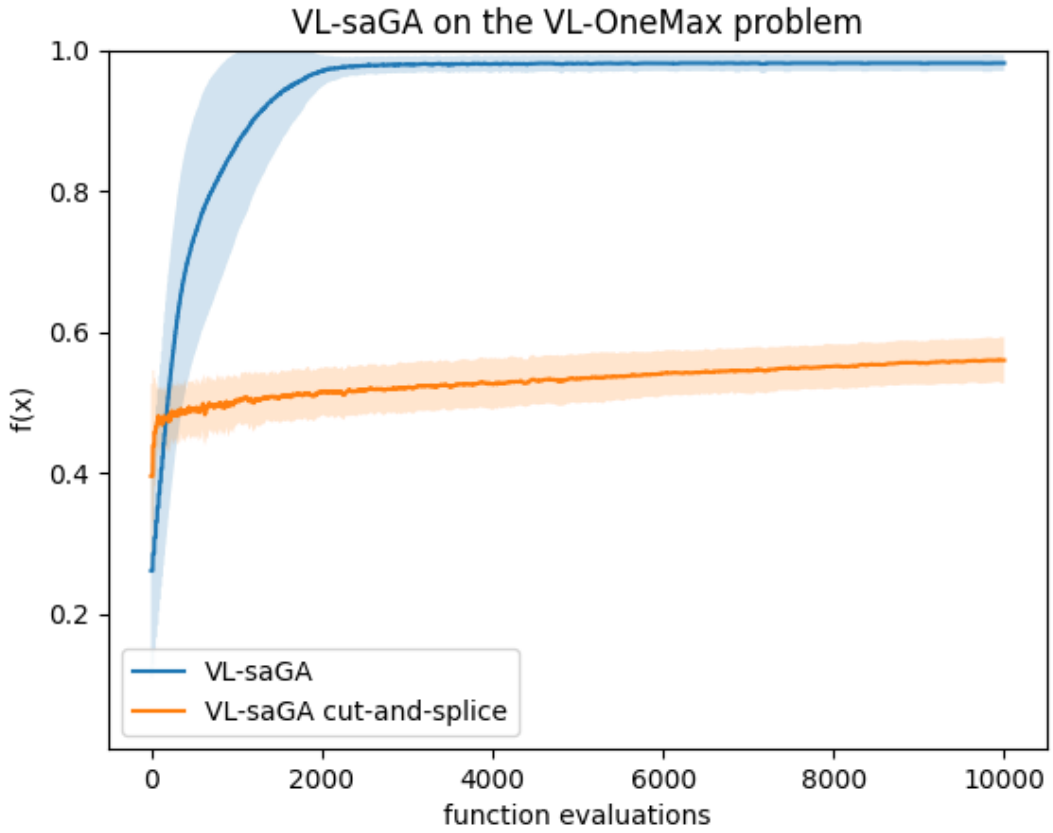


Figure 13: Two versions of the VL-saGA algorithm, one using fixed-length crossover and one using length-varying crossover (cut-and-splice crossover 15), run on the VL-OneMax problem with target length $l^* = 500$. The standard-deviation is represented by the opaque areas. The parameter-settings are the same as the previous experiment 12.

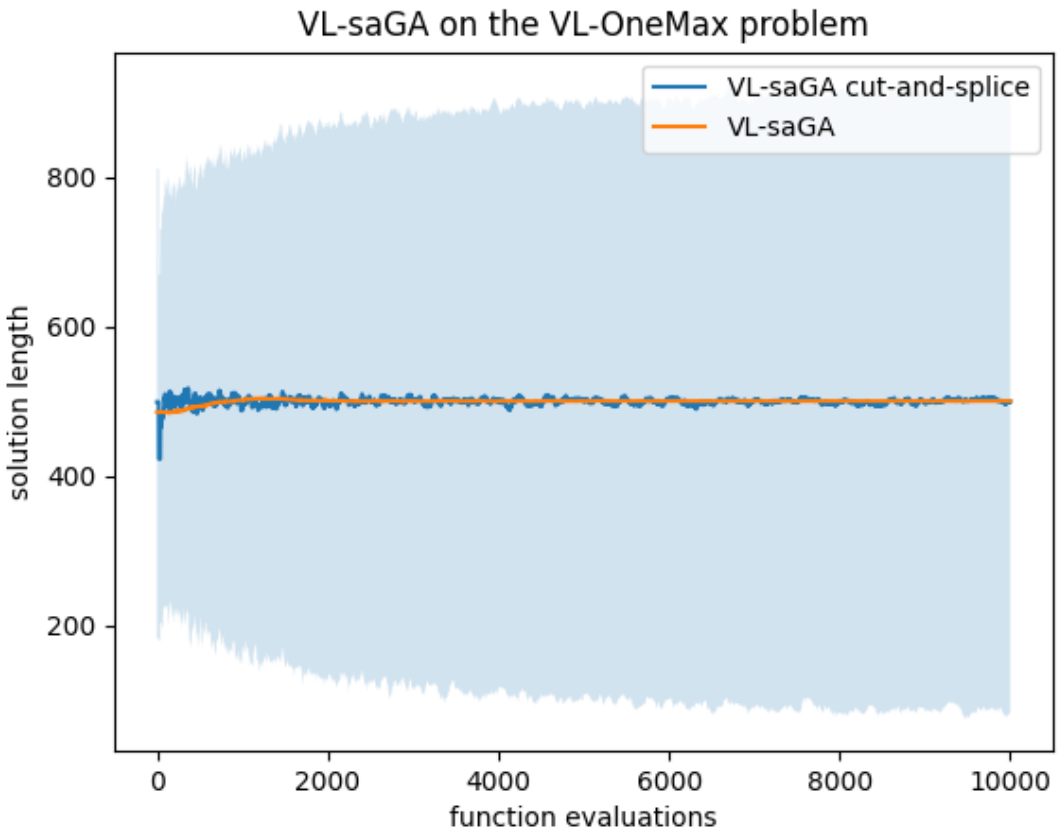


Figure 14: Change in solution-lengths during optimization as shown in figure 13. The opaque areas represent the length-distribution of the λ new candidate-solutions.

6.3.1 Results

Figure 12 shows VL-saGA and VL-vGA run on the VL-OneMax problem. VL-saGA and VL-vGA need more function evaluations to reach the optimal solution than most of the other algorithms. Replacing the cross_c operator with cut-and-splice crossover negatively affects the performance (see Figure 13). The cut-and-splice operator seems to be too disruptive. Figure 14 shows that VL-saGA with cut-and-splice crossover does not converge to a particular solution length. This might be because cut-and-splice crossover creates so much length-variation within the population that it counteracts the effect of the selection-operator on the average solution length. Consequently, the optimization process does not converge to any particular solution-length.

6.4 Comparing VL-vGA with VL-vGA Mutation Only

There are various parameter settings that can be used for the Variable Length vanilla Genetic Algorithm. The first question is whether to use a length-varying crossover operator or to rely on the mutation operator to vary the length of the solutions. The first version, VL-vGA, uses cut-and-splice crossover in combination with length-varying mutation. Since cut-and-splice crossover can result in offspring of vastly different lengths, one would expect that the variation in length within each generation is greater than when using length-varying mutation with one-point crossover. One could also choose to use no crossover whatsoever. The second question is

whether the performance of VL-vGA could be improved by using other methods such as niche-selection or by adding parsimony pressure. These questions are investigated in the following two experiments.

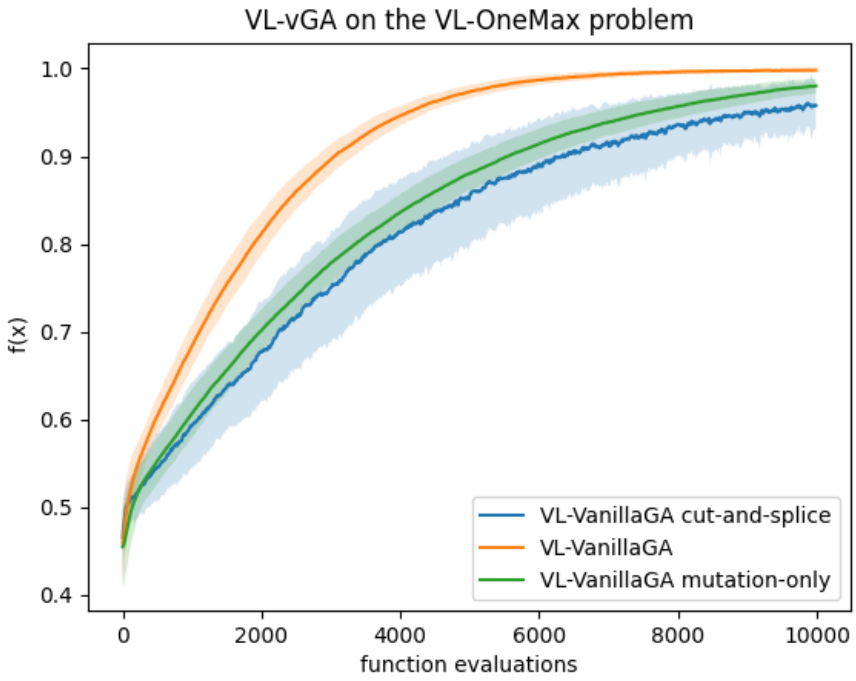


Figure 15: Different versions of the VL-vGA on the VL-OneMax problem with population size $\mu = 10$. For each algorithm, the results have been averaged over 100 random seeds. The standard-deviation is represented by the opaque areas. (For cut-and-splice crossover see 15)

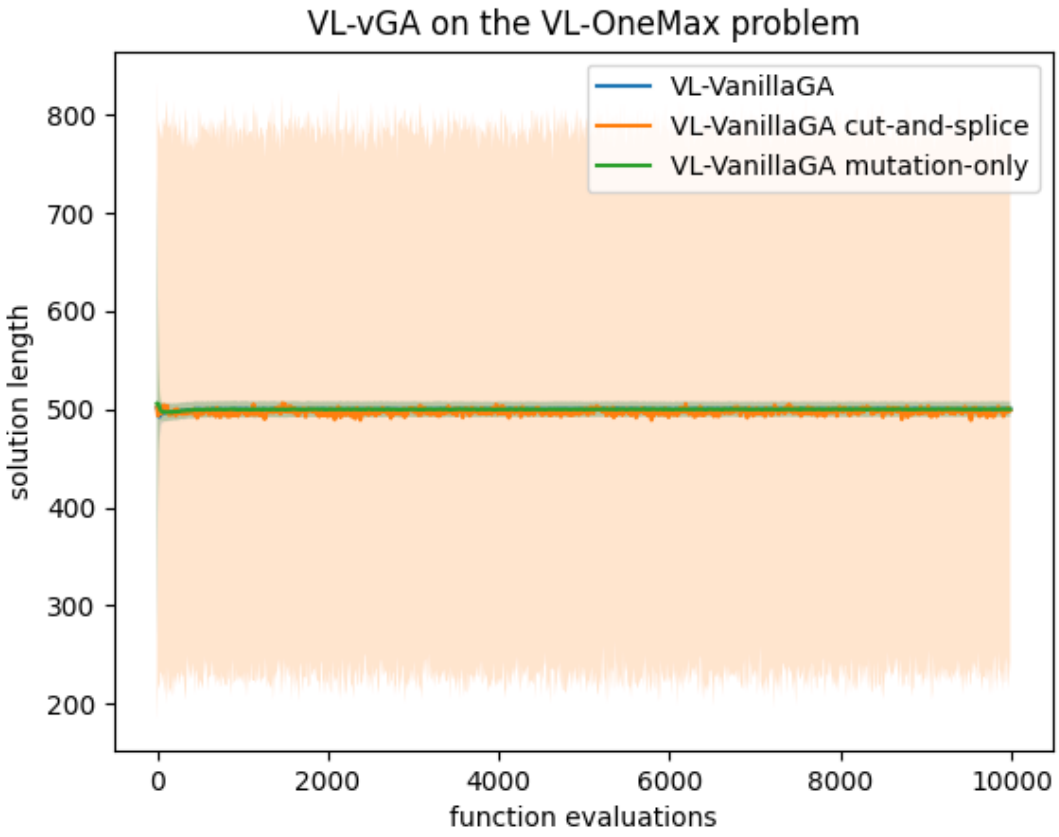


Figure 16: Change in solution-lengths during optimization as shown in figure 15. The opaque areas represent the length-distribution of the λ new candidate-solutions.

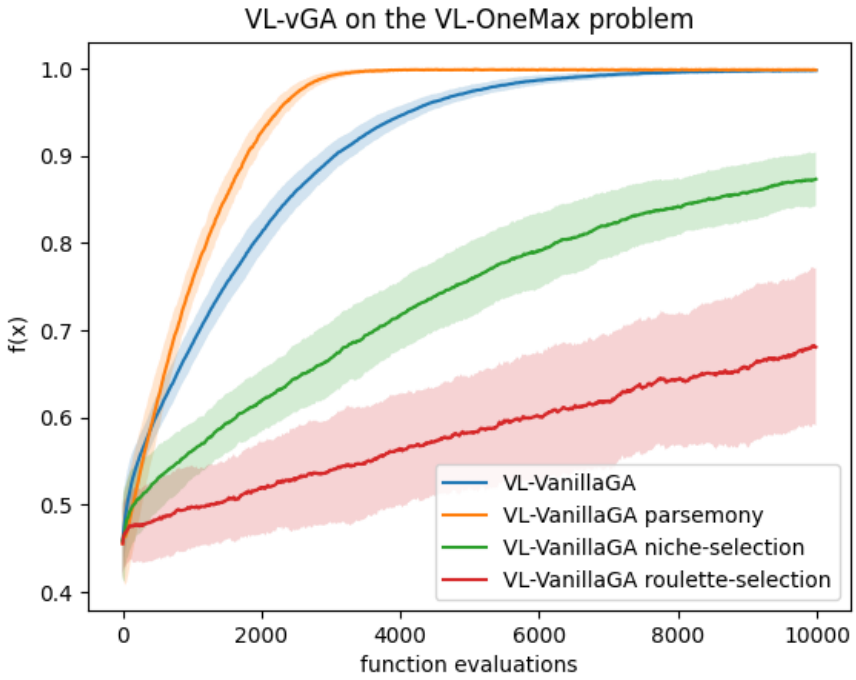


Figure 17: VL-vGA with different selection operators. The standard-deviation is represented by the opaque areas. The other parameters have not been changed. The standard VL-vGA uses tournament-selection with tournament-size $k = 3$.

6.4.1 Results

As can be seen in Figure 15, the version of VL-vGA using cut-and-splice crossover performs worse than the other versions. It seems that using a length-varying crossover operator is too disruptive. Figure 16 shows that the cut-and-splice crossover operator causes each generation to have a high level of length diversity. The mutation-only algorithm performs better, but not as well as the version with one-point crossover. Figure 17 shows the VL-vGA with one-point crossover for different selection operators. The version with roulette-wheel selection learns rather slowly. Parsimony pressure performs better than tournament selection. VL-vGA with niche-selection performance better than roulette-wheel selection but worse than tournament-selection. This might be because premature convergence was not an issue for VL-vGA on the VL-OneMax problem. Ensuring more length-diversity within the population would just drive the algorithm to do less exploitation and more exploration.

6.5 VL-CMA-ES

In section 5.6, a description was given different options for regulating the duplication and deletion probabilities. This section contains experiments with the various options. For the FunctionMatch problem, increasing dimensionality should be beneficial since solutions with more variables will be able to better approximate the target function. On the other hand, duplicating or deleting variables can result in an increase in the mean squared error, especially when the function approximation is already close to the target function. Probably, leaving out the decay factor for the duplication and deletion probabilities will prevent the algorithm from converging toward the end of the optimization process. In these experiments, the decay factor will be $p_{decay} = 0.95$ or $p_{decay} = 1$, with the adaptation factor being $a = 0.05$ or $a = 0$.

Setting p to 1 and a to 0 means that there is no decay or adaptation of the duplication or deletion probabilities occurs. The target function j will be:

$$j(x) = \sin(10 \cdot x) \tag{18}$$

The initial dimensionality is set to $d_{init} = 10$ for all the following experiments.

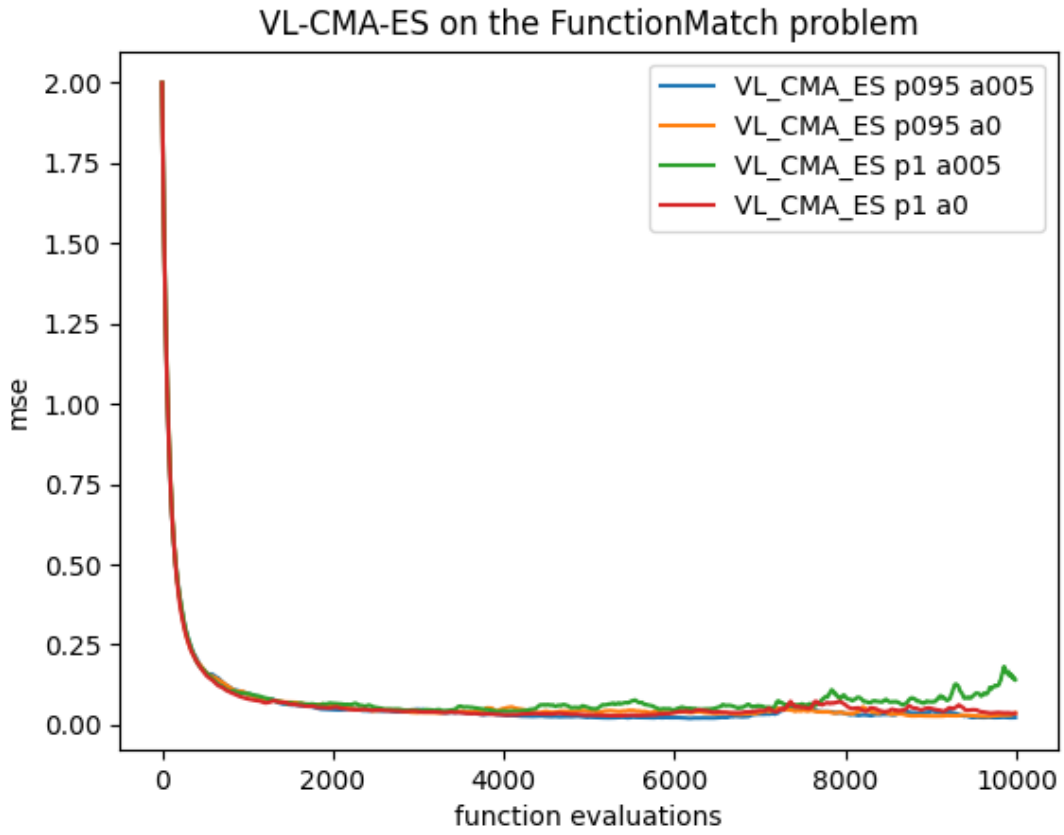


Figure 18: VL-CMA-ES on the function-match problem. $p1$ and $p095$ represent $p_{decay} = 1$ (no decay takes place) and $p_{decay} = 0.95$. $a0$ and $a005$ stand for the adaption factor $a = 0$ and $a = 0.05$ The target function is $j(x) = \sin(10 \cdot x)$.

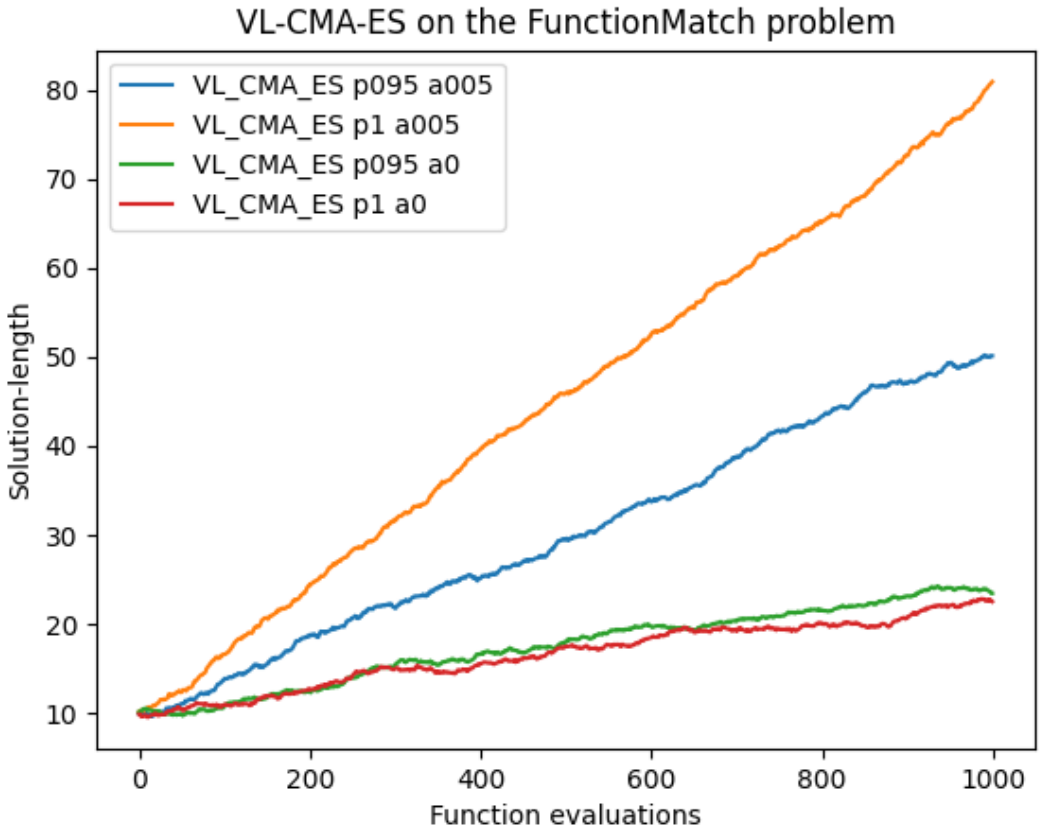


Figure 19: The change in dimensionality for VL-CMA-ES on the function-match problem.

6.5.1 Results

As can be seen in figure 18, the version of VL-CMA-ES with $p_{decay} = 0.95$ and $a = 0.05$ performs best. However, apart from the version with $p_{decay} = 1$ and $a = 0.05$, the loss is only slightly lower. Setting p_{decay} to $p_{decay} = 1$ mainly affects the later stages of the optimization process, causing the loss to fluctuate significantly. This is especially true if $a = 0.05$. Looking at figure 19, which shows how the solution length changes during optimization, it is clear that choosing $a = 0.05$ does increase the solution length significantly. The version with $p_{decay} = 1$ and $a = 0.05$ found a larger dimensionality; however, the performance was still worse than the other three versions due to the lack of convergence towards the end of the optimization process. Surprisingly, the performance of the other three versions is very similar, though the version with $p_{decay} = 0.95$ and $a = 0.05$ did reach a significantly higher dimensionality. This might be explained by the fact that, for the function-match problem, the benefit of adding more variables gets smaller and smaller for each variable that is added.

7 Conclusions

In this work, we have presented adaptations of various classic evolutionary algorithms for binary optimization and a variable-length version of the CMA-ES algorithm. The adapted binary algorithms were able to find the optimal solutions for the variable length OneMax problem. Adding length-varying crossover negatively affected the performance of VL-saGA and VL-vGA.

VL-vGA with niche-selection and Vl-vGA with parsimony-pressure did not perform as well as the VL-vGA with tournament-selection. This might be because the VL-OneMax problem is quite a simple problem and lack of population diversity and uncontrolled bloat were not a problem. For more complicated problems, it might prove to be beneficial after all. The crossover operators mentioned in the related works section, except cut-and-splice crossover, were not taken into consideration. These operators are specific to metameric representations, which were not considered in this work. As mentioned above, an overview of metameric algorithms can be found in [9]. On the whole, we can conclude that generally length-varying mutation is sufficient to deal with the length optimization part of variable-length optimization problems. Cut-and-splice crossover seems to be too disruptive. Using more sophisticated crossover operators, like SVLC might result in a better performance of VL-saGA and VL-vGA. However, at least for the VL-OneMax problem, both algorithms can find the correct solution-length with only mutation as a length-varying operator. The experiments show that VL-CMA-ES works on the simple continuous optimization problem. The adaptation method for the deletion and duplication probabilities does result in an increase in the number of dimensions for the Function Match problem, and the decay factor helps to stabilize the optimization process towards the end of the optimization process. Because of the nature of the FunctionMatch problem, there is no target dimensionality. That is to say, a higher dimensionality is always better. Further testing on other optimization problems is needed to see how well VL-CMA-ES can find the optimal dimensionality of a given variable-length optimization problem. The experiment proves that the concept works on simple problems. Whether and how well it works on more complicated problems is a question for further research.

References

[1] David M Cherba and William Punch. "Crossover gene selection by spatial location". In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. 2006, pp. 1111–1116.

[2] Carola Doerr et al. "Benchmarking discrete optimization heuristics with IOHprofiler". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2019, pp. 1798–1806.

[3] Nikolaus Hansen. "The CMA evolution strategy: A tutorial". In: *arXiv preprint arXiv:1604.00772* (2016).

[4] Benjamin Hutt and Kevin Warwick. "Synapsing variable-length crossover: Meaningful crossover for variable-length genomes". In: *IEEE transactions on evolutionary computation* 11.1 (2007), pp. 118–131.

[5] Kyung-Joong Kim and Sung-Bae Cho. "Automated synthesis of multiple analog circuits using evolutionary computation for redundancy-based fault-tolerance". In: *Applied Soft Computing* 12.4 (2012), pp. 1309–1321.

[6] Zhenhua Li and Qingfu Zhang. "What does the evolution path learn in CMA-ES?" In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2016, pp. 751–760.

[7] Guillermo Molina, Enrique Alba, and El-Ghazali Talbi. "Optimal sensor network layout using multi-objective metaheuristics." In: *J. Univers. Comput. Sci.* 14.15 (2008), pp. 2549–2565.

[8] Matthew L Ryerkerk et al. "Solving metameric variable-length optimization problems using genetic algorithms". In: *Genetic Programming and Evolvable Machines* 18 (2017), pp. 247–277.

[9] Matthew Lee Ryerkerk. "Metameric representations in evolutionary algorithms". PhD thesis. Michigan State University, 2018.

[10] Yerbol A Sapargaliyev and Tatiana G Kalganova. "Open-ended evolution to discover analogue circuits for beyond conventional applications". In: *Genetic Programming and Evolvable Machines* 13 (2012), pp. 411–443.

[11] Hans-Paul Schwefel. "Projekt MHD-Staustrahlrohr: Experimentelle optimierung einer zweiphasendüse". In: *Teil I. Technischer Bericht 11.034/68*. 35. AEG Forschungs institute, 1968.

[12] Joachim Sprave and Susanne Rolf. "Variable-dimensional optimization with evolutionary algorithms using fixed-length representations". In: *International Conference on Evolutionary Programming*. Springer. 1998, pp. 261–269.

[13] Tan Yew Teck and Mandar Chitre. "Direct policy search with variable-length genetic algorithm for single beacon cooperative path planning". In: *Distributed Autonomous Robotic Systems: The 11th International Symposium*. Springer. 2014, pp. 321–336.

[14] Benjamin James Yu. "A Genetic Algorithm Framework using Variable Length Chromosomes for Vehicle Maneuver Planning". PhD thesis. Massachusetts Institute of Technology, 2022.