Utrecht University

# Multiple Sequence Alignment Using Model-Based Evolutionary Algorithms

June 8, 2018

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the
Department of Information and Computing Sciences

**Author**
Mattijs van de Kuijlen, BSc

**Student Number**
ICA-4151593

**Supervisor**
dr. ir. Dirk Thierens

**Abstract**

The construction of high quality multiple sequence alignments (MSAs) is an important problem in the field of bioinformatics. MSAs are used for a wide range of different purposes, such as phylogenetic analysis, conserved motif identification and structure prediction. In this thesis we present an approach for constructing multiple alignment profiles of high quality for solving the MSA problem. The multiple alignment profiles that are used in this thesis are the positional weight matrices (PWMs). A standard evolutionary algorithm and two variants of the Gene-Pool Optimal Mixing Evolutionary Algorithm (GOMEA) will be used for constructing high quality PWMs. The performance and the scalability of the evolutionary algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm will be compared. We will show that the Linkage Tree Genetic Algorithm performs significantly better than the algorithm of Botta and Negro and the univariate GOMEA algorithm. We will also show that there is no significant performance difference between the algorithm of Botta and Negro and between the univariate GOMEA algorithm. Finally, we will show that both variants of the GOMEA algorithm scale quite poorly in comparison to the algorithm of Botta and Negro.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

The alignment of biological sequences is one of the most important techniques in the field of bioinformatics [42]. The goal of aligning biological sequences, i.e. DNA sequences, RNA sequences and protein sequences, is to construct an alignment of at least two sequences in such a way that the similarities between these sequences are maximized [13, 34]. In other words, this means that the sequences are arranged such that a maximal number of identical or similar residues is found.

The problem of aligning a set of biological sequences is a difficult problem [11]. This is due to the fact that even though multiple sequences might be very similar, they might look very different. This is normal for sequence structures that have evolved from a common ancestor for example. During the evolutionary process elements in a sequence might be substituted with other elements from the alphabet. Furthermore, elements may be removed from a sequence or new elements might be inserted into a sequence during evolution. Thus, even though multiple sequences might have evolved from a common ancestor, they could have evolved in quite different ways. For this reason, they could seem very different. The sequence alignment problem thus has to deal with the difficulty that multiple sequences can look very different, even though they are very similar.

The sequence alignment problem can be divided into two variants [24]. The first variant is the local sequence alignment problem, in which the focus lies upon maximizing the similarities between only parts of the sequences. The second variant however is concerned with maximizing the similarities between entire sequences. This variant is called the global sequence alignment problem. Not only can the sequence alignment problem be divided into two different variants, it can also be divided into two different instances. These instances are called the pair-

wise sequence alignment problem (PSA) and the multiple sequence alignment problem (MSA). For the PSA problem the goal is to maximize the similarities between exactly two sequences, while the goal is to maximize the similarities between three or more sequences for the MSA problem. This thesis will focus on the global sequence alignment problem for the MSA instance of the sequence alignment problem.

Multiple sequence alignments provide an extremely powerful way of analyzing biological sequence structures [35, 38]. They are used for a wide range of different purposes, such as phylogenetic analysis, conserved motif identification and structure prediction. The goal of phylogenetic analysis is to learn the relationships that exist between various organisms. Conserved motif identification is concerned with identifying motifs that were not changed during evolution, i.e. which were conserved during the evolutionary process, and are thus important to the structure and the function of the set of related sequences. Finally, structure prediction aims at predicting the function that a residue fulfills in a sequence structure. The problem of finding good or optimal multiple sequence alignments is thus of vital importance in analyzing biological sequence structures. Analyzing such structures is not only relevant from a purely research oriented perspective, it can also be very relevant to our society. If we understand how biological structures evolve and function, we can possibly develop new and better medicines or possibly we could even neutralize entire diseases. This line of research could thus also be of use to our society in the sense that it could improve the quality of life of ill people and in the sense that it could decrease the annual health care expenses when less people have diseases.

To find the optimal MSA for a set of sequences, the Needlemann and Wunsch algorithm [32], which finds optimal solutions to the PSA problem, needs to be generalized [35]. The problem with this generalization is that the MSA problem is NP-Complete [39]. Therefore, it is only possible to find optimal solutions for very small sets of sequences. However, the volume of sequence data keeps growing at an exponential rate. Let us look at two well-known ribosomal RNA databases to illustrate that the volume of available sequence data is too large for exact algorithms. The SILVA SSU database already contained over 3.4 million ribosomal RNA sequences in July 2012 [40] and the RDP database already contained over 2.8 million ribosomal RNA sequences in October 2013 [14]. Thus, even years ago these databases were already too large for exact methods for solving the MSA problem.

Finding solutions of high quality for the MSA problem is thus very relevant to be able to analyze biological sequence structures. It is infeasible to find optimal solutions with absolute certainty, because exact algorithms cannot be used for all but the smallest sets of sequences. It is thus of very useful to develop efficient algorithms that can find good solutions for the MSA problem.

## 1.2 Research Goals

As already mentioned before, there exist exact algorithms for solving the MSA problem [35]. These algorithms are however infeasible for almost all sets of sequences in the sense that they require far too much time to find the optimal MSA. Exact algorithms are only feasible for a few very small sets of sequences. Because exact algorithms are infeasible, all algorithms for solving the MSA problem are heuristic algorithms. Notable heuristic algorithms include for example MUSCLE [19], T-Coffee [37], Clustal W [47] and SAGA [36].

Besides these algorithms, another interesting approach for solving the MSA problem has been proposed by Botta and Negro [9]. The method that they propose is to find a good positional weight matrix (PWM), which represents a multiple alignment profile of the given input sequences. By using the Needleman and Wunsch algorithm [32], the MSA can be computed by aligning each of the input sequences with the PWM. A good PWM is found by evolving a set of PWMs with the help of a basic genetic algorithm. This genetic algorithm does not use any clever optimizations, nor does it use any advanced techniques for improving the quality of the PWMs that are evolved. Even though they use a basic evolutionary algorithm in the sense that the PWM allows the use of very simple genetic operators, their results seem to indicate that this method performs well in comparison with many other heuristic algorithms for solving the MSA problem. This is thus a very interesting approach for the reason that it still performs quite well, even though it is much simpler than most MSA algorithms. This algorithm will be described in more detail in Chapter 5.

As was mentioned before, MSA is used amongst others for identifying conserved motifs [35]. The existence of motifs that are preserved during the evolutionary process, seems to indicate that some elements inside the sequences are related in some way. This would mean that the elements in the sequences in the MSA are not all independent of one another. The method for learning the PWMs as presented by Botta and Negro [9], does not assume any relation between any two positions in a PWM. The performance of this method could thus possibly be improved if these relations are taken into account during the process of evolving the PWMs. In other words, the performance of this method could possibly be improved if we could incorporate the learning of linkage information, i.e. learning of the relations that exist between elements in the sequences in the MSA, into the algorithm.

One algorithm that incorporates linkage learning is the Gene-Pool Optimal Mixing Evolutionary Algorithm (GOMEA) [6, 46]. This algorithm makes it possible to use linkage information to improve the quality of the solutions that are obtained during the evolution of the population. By using GOMEA instead of a basic genetic algorithm for learning the PWMs, the final quality of the PWMs and therefore also the final quality of the MSA could possibly be improved. The aim of this research therefore is to investigate the performance of replacing the basic genetic algorithm with GOMEA for the method as presented by Botta and Negro [9].

In the GOMEA algorithm it is possible to use multiple different structures [46]. In this thesis the focus will be on comparing the performance of the univariate GOMEA structure, of the linkage tree GOMEA structure and of the algorithm presented by Botta and Negro [9]. The goal of this comparison is to investigate the feasibility of using GOMEA for learning good PWM profiles for the MSA problem. The linkage tree structure is able to represent interactions between the variables, where the univariate structure assumes independence between variables. Because we assume that not all variables in the MSA problem are independent, we expect that the linkage tree structure constructs a better model for the MSA problem and therefore we expect to be able to find better solutions with the linkage tree structure than with the univariate structure.

## 1.3 Outline

The remainder of this thesis is outlined in the following manner. First of all, we will provide a discussion on related research for the MSA problem in Chapter 2. Thereafter, a description of genetic algorithms will be given in Chapter 3. Chapter 4 will then explain the GOMEA algorithm in detail. We will provide a description of the algorithm as presented by Botta and Negro [9] in Chapter 5. We will analyze the computational complexity of the algorithms in Chapter 6. We will then describe the experiments for determining the parameter settings for both variants of the GOMEA algorithm in Chapter 7. Thereafter, we will compare the performance and the scalability of the algorithms in Chapter 8. Finally, we will present our conclusions in Chapter 9.

# Chapter 2

# Related Work

Multiple sequence alignment algorithms can be divided into three categories, namely exact algorithms, progressive algorithms and iterative algorithms [13]. The MSA problem is NP-Complete, which means that exact algorithms are infeasible for almost all sets of sequences [39]. For this reason, we will not provide a discussion of exact algorithms for solving the MSA problem. Instead this chapter aims at giving an overview of previous research into both progressive algorithms and iterative algorithms in respectively Section 2.1 and Section 2.2.

## 2.1 Progressive Alignment Algorithms

Progressive alignment algorithms are heuristic algorithms for finding good multiple sequence alignments for a given set of sequences [13]. These algorithms try to find a good alignment by splitting the problem into subproblems, which are then solved step by step until a good multiple sequence alignment for all of the sequences has been found. The approach that these algorithms take, is to construct a guide tree. In such a guide tree the leaves correspond to the input sequences and internal nodes correspond to a multiple sequence alignment of the sequences in their subtree. This means that the root of such a tree corresponds to the multiple sequence alignment of the given input sequences. These trees are constructed by combining at each step two nodes, which can be both leaf nodes or internal nodes, that have the smallest distance. In other words, at each step the two nodes that result in the alignment with the highest similarity are combined. Progressive alignment algorithms thus try to construct a multiple sequence alignment in a greedy manner. The problem with such a greedy approach is that gaps, which are spaces in the sequences that have been introduced to align the sequences as good as possible, cannot be altered after they have been introduced. This can lead to the problem that the alignments get stuck in a local minimum.

One of the progressive alignment algorithms is Kalign [28]. In most progressive alignment algorithms, the distances that are used in constructing the guide tree, are calculated by creating a matrix that contains the pairwise distances between all of the sequences. In this way the distances are calculated in an exact way. The downside of this approach is that it is computationally expensive. Therefore, the Kalign algorithm approximates the pairwise distances with the help of the Wu-Manber algorithm for computing an approximate string-matching. In this way the Kalign algorithm tries to construct high quality MSAs much faster than other algorithms that use the exact distance calculations.

CLUSTAL W is another program for constructing progressive alignments [47]. The CLUSTAL W program starts by computing a matrix of pairwise distances between the sequences. Users have a choice between computing this distance matrix with a fast approximate algorithm or with a slower but more accurate dynamic programming algorithm. The next step that this program takes, is to construct a guide tree with the help of the Neighbor-Joining method. The third and final step of the CLUSTAL W program is to construct a progressive alignment by combining groups of sequences using pairwise alignments in the branching order that is specified by the guide tree. The CLUSTAL W program improves the progressive alignment strategy by using sequence weights calculated from the guide tree and by using gap penalties that are position specific.

ProbCons is a progressive alignment algorithm that can, on a fundamental level, be interpreted as a pair-hidden Markov model-based algorithm [17]. The ProbCons algorithm consists of five steps, with the addition of a possible post-processing step for refining the quality of the obtained solution. In the first step a posterior-probability matrix is computed for each pair of sequences. These matrices are then used in the second step to compute the expected accuracy of each possible alignment for each pair of sequences. For each pair of sequences, the alignment with the highest expected accuracy is then selected. In the third step a probabilistic consistency transformation is performed. This transformation ensures that, for each of the sequence pairs, the similarity of the given sequence pair to all the other sequences is incorporated into the comparison of the sequence pair. The fourth and fifth step are the normal steps for a progressive alignment algorithm, i.e. constructing a guide tree and computing a progressive alignment.

The T-Coffee algorithm tries to overcome the pitfalls of the greedy approach that is employed by the progressive alignment method [37]. For this purpose, the T-Coffee algorithm incorporates knowledge about the alignments between all sequence pairs into the progressive alignment step. In other words, the T-Coffee algorithm adds additional knowledge to the alignment that is created each time that two nodes in the guide tree are combined. To incorporate such knowledge, the T-Coffee algorithm computes a library of alignments before executing the progressive alignment step. The first step in creating this library is to construct two primary libraries of pairwise alignments. One library consists of global pairwise alignments and is created using the CLUSTAL W program. The other library consists of local pairwise alignments and is created using the Lalign program of the FASTA package. The second step in constructing the library, is to derive weights for each alignment in both of the primary libraries. The

next step is to combine both primary libraries into a single library. Then the final step in constructing the library, is to extend the library constructed in the third step in such a manner that the weight of each alignment will incorporate some global information about the entire library. This is done by a heuristic algorithm. The library that is obtained in this way can then be used to construct a progressive alignment of higher quality than the alignment obtained by the standard progressive alignment method.

## 2.2 Iterative Alignment Algorithms

Iterative alignment algorithms are algorithms that produce an alignment by iteratively refining this alignment [35]. These algorithms thus consist of two different components, namely a method for computing an alignment of the input sequences and a method for refining this alignment. The iterative alignment methods can be divided into deterministic methods and into stochastic methods. The advantage of using the stochastic methods over using the deterministic methods is that there is a clear separation between the refinement process and the objective function for the stochastic methods. On the other hand, the benefit of the deterministic methods is that their results are reproducible. The results of the stochastic methods are not reproducible, due to the randomness that is part of these methods.

The MAFFT algorithm is a deterministic iterative alignment algorithm that constructs an alignment of the given input sequences by making use of the fast Fourier transform [25]. The fast Fourier transform is used for identifying homologous segments. For producing an alignment of a pair of sequences, the MAFFT algorithm computes a homology matrix of the sequence pair. This homology matrix can be used to consistently align the homologous regions in the two segments by using the dynamic programming approach for finding the optimal pairwise alignment between the segments. Besides the use of the fast Fourier transform, the MAFFT algorithm also uses an improved scoring system that improves the accuracy of the alignments. The scoring system is improved in two ways. The first improvement is obtained by creating a normalized similarity matrix that contains both positive and negative values, whereas most other methods use similarity matrices that contain only positive values. The second improvement is obtained by altering the gap penalties. When a sequence is aligned with a group of sequences that has already been aligned, it is possible that a gap is introduced in this sequence. If there is already a sequence in the group of aligned sequences that has a gap at the same position as the gap that is being introduced in this sequence, then no gap penalty is assigned for introducing this gap. The alignments obtained by the MAFFT algorithm have a quality that is comparable with the most accurate methods that were known at the time that this algorithm was proposed. The benefit of using the MAFFT algorithm over using these other methods, is that the MAFFT algorithm finds the alignment much faster than these methods do.

The MUSCLE algorithm is an iterative alignment algorithm that consists of

three steps [19]. In the first step an initial progressive alignment of the given input sequences is constructed. The second step aims at improving the initial guide tree and the initial progressive alignment that are obtained by the first step of the algorithm. In this second step a guide tree is constructed by using a clustering algorithm on a Kimura distance matrix, which is also computed in this step. For each iteration, the previous tree and the new tree are compared. If more than one iteration has been executed and the number of internal nodes that has been changed has not decreased, then the second step terminates. The third and final step also aims at iteratively refining the alignment. In this step the sequences are divided into two disjoint subsets by removing an edge from the guide tree. For both subsets a multiple alignment profile is extracted. These profiles are then realigned with one another with the help of a profile-profile alignment method. If the quality of the alignment is improved by this method, then the new alignment is kept. Otherwise, the new alignment is discarded. The third step terminates either after a maximum number of iterations, as defined by the users, have been executed or if it has been tried to remove all the edges and none of these removals have resulted in an improvement of the final alignment.

The group of stochastic iterative alignment algorithms consists amongst others of simulated annealing algorithms [27] and of genetic algorithms [11, 24, 26, 29, 34, 36, 38]. These genetic algorithms employ different approaches for finding a high quality multiple sequence alignment. The SAGA algorithm is a popular algorithm for sequence alignment that uses a large set of twenty-two different mutation and crossover operators, two different crossover operators and twenty different mutation operators [13, 36]. SAGA employs a schedule that automatically decides which of the different operators are to be used. The SAGA algorithm is able to obtain good alignments, but this comes at the expense of a high computation time. A few of the reproduction operators that are employed by the SAGA algorithm, will be discussed in Chapter 3. On the other hand, the GA-ACO algorithm tries to improve the quality of the alignments by applying a local search method for each generation [29]. The algorithm uses an ant colony optimization method for the local search step. The ant colony optimization method uses a sliding window to detect mismatched block in the alignment, which it then tries to improve. Yet another approach is employed by the MSAGMOGA algorithm [26]. This algorithm is a multi-objective genetic algorithm that aims at optimizing three objective functions at the same time. The first objective is to maximize the similarity between the sequences in the alignment. The second objective is to minimize the affine gap penalty, which corresponds to alignments in which the gaps are grouped together as much as possible. The third an final objective is to maximize the support of the alignment, i.e. maximize the number of sequences that are present in a solution of good quality. The MSAGMOGA algorithm can find alignments with a better accuracy than SAGA does and it also does this faster. The algorithm proposed by Botta and Negro [9] is also a stochastic iterative alignment algorithm. We will however not discuss it here, because this algorithm will be described in detail in Chapter 5.

# Chapter 3

# Genetic Algorithms

In nature, individuals compete with one another to obtain access to certain resources such as water, food and shelter [43, 44]. Individuals require access to these resources in order to be able to survive. For each individual the capability of obtaining access to these resources and thus the capability to survive, is dependent of the genetic features of the individual. Individuals that have a better set of genetic features, are more likely to survive. Not only do such individuals have a better chance of obtaining access to necessary resources, they also have a better chance of finding a mate and thus a better chance of producing offspring. This means that good genetic features are passed on to successive generations, while bad genetic features will eventually be lost. The process of evolution will thus cause an increase in the fitness of the genetic features in a species, which means that species will become more and more adapted to their environment during each successive generation. This phenomenon, in which only the fittest individuals survive and in which only to best genetic features will be preserved, has been described by Charles Darwin as "the survival of the fittest".

The concepts of natural selection and "the survival of the fittest" form the basis of the class of genetic algorithms [3]. Genetic algorithms are algorithms that are used in solving both search problems and optimization problems. The idea of these algorithms is that they try to improve, i.e. "evolve", a set of solutions over a number of iterations, i.e. "generations". During each iteration the solutions will reproduce and only the best, i.e. "the fittest", solutions will survive. The method that genetic algorithms employ, is thus analogous to the method of natural evolution. The basic structure of a genetic algorithm is represented in Algorithm 1 [49]. The components of the algorithm will be explained further on in this chapter.

Genetic algorithms have several strong points [21, 50]. For one, they have a very generic nature. Due to this generic nature, genetic algorithms can be applied to many different problems. On the other hand, this also means that their performance will often be worse than domain-specific algorithms. Genetic algorithms

**Algorithm 1** A basic genetic algorithm

```
1: procedure GA
2:     Initialize population
3:     Evaluate fitness of individuals in the population
4:     while Termination criterion not satisfied do
5:         Select solutions for reproduction
6:         Perform reproduction
7:         Mutate solutions
8:         Evaluate fitness of the new solutions
9:         Select the fittest solutions for the next generation
10:    end while
11: end procedure
```

thus provide a simple framework for solving many different optimization and search problems, even though domain-specific algorithms often achieve better results. Furthermore, it is relatively easy to combine genetic algorithms with techniques such as local search. This means that it is possible to use many techniques for the purpose of improving the quality of the solutions that are obtained by the algorithm. It is also possible to incorporate domain knowledge relatively easy into genetic algorithms, which means that genetic algorithms can be tailored to the problem that is under investigation. Even though genetic algorithms are thus very generic in nature, their performance can be improved for the specific problem that is under investigation by adapting the algorithm to the problem at hand. Finally, genetic algorithms can explore multiple parts of the search space at once. This means that they are less likely to get stuck in a local optimum than that several other techniques are.

Genetic algorithms also have several weak points [21, 50]. One weak point of genetic algorithms is that they have multiple parameters. This means that it can be quite difficult and time consuming to find a parameter setting that allows the algorithm to produce good results. Another weak point of genetic algorithms is that they are stochastic in nature. Due to their stochasticity, there is absolutely no guarantee on the quality of the final solutions. The results can be very good for one run of the algorithm, while they are very bad for another run. Therefore, they are not suited for critical applications, in which good solutions are required all the time. Finally, genetic algorithms have a high computational workload, because of the fact that it is necessary to evaluate a lot of possible solutions. For this reason, they are not suitable for use in applications that need to return results in real-time.

The basic structure of a genetic algorithm has already been depicted in Algorithm 1 [49]. The specific components of which genetic algorithms consist, have not been investigated more thoroughly however. Any genetic algorithm consists of several basic components [20]:

1. A representation of the problem.
2. A function that evaluates the quality of any given solution.

3. A population of solutions to the problem.
4. A method to select the solutions that will be used for reproduction.
5. An operator for the reproduction of multiple solutions, i.e. an operator that creates new solutions from a set of parent solutions.
6. An operator for mutating solutions.
7. A method for selecting which solutions need to be carried over to the next generation.
8. A condition that specifies when the algorithm should terminate.

All of these components will be described in more detail in the remainder of this chapter. Section 3.1 will focus on the representation of the problem. A description about how the quality of a solution can be evaluated will be given in Section 3.2. The focus of Section 3.3 will be on the population mechanism that is used in genetic algorithms. The method for selecting which solutions will be used for reproduction will be described in Section 3.4. Section 3.5 and Section 3.6 will focus on how new solutions are created by respectively describing the reproduction process and the process of mutating the solutions. In Section 3.7 the focus will be on how genetic algorithms select the solutions that are carried over to the next generation. Finally, in Section 3.8 the focus will be on describing under what conditions a genetic algorithm terminates.

## 3.1 Problem Representation

To be able to use a genetic algorithm for the purpose of finding a solution to a given problem, it is necessary that a suitable representation of the problem is designed [20]. A solution to the problem under investigation can have a structure that can neither be stored efficiently in a computer nor can be easily manipulated by the algorithm. Let us look at solutions to the knapsack problem to illustrate this. A solution to the knapsack problem consists of a set of items that can be fitted into the knapsack. If we represent such a solution in the genetic algorithm as a three-dimensional drawing of the knapsack containing the items, then such a solution probably cannot be stored very efficiently nor can such a solution be manipulated very easily. Would we represent a solution as a binary string that specifies for each item whether or not it is contained in the solution, then the solution could be stored much more efficiently and the algorithm could manipulate such a solution much more easily. This example should illustrate that it important to develop a suitable representation of the solutions for use by the algorithm.

For genetic algorithms, there are thus two different representations of a solution to the problem [2]. The first representation is the representation of a solution in the problem space, i.e. the representation of a real solution. This type of representation of a solution is called a phenotype. The other representation is the representation of a solution in the genetic algorithm, which is

known as the genotype (sometimes it is also called an individual or a chromosome). Each genotype consists of multiple parameters/variables that are known as genes. A representation of the problem is thus a mapping from phenotype to genotype [20]. The mapping from a phenotype to a genotype is also known as encoding. The goal of using the genetic algorithms is to find real solutions to the problem, i.e. solutions in the phenotype space. The genetic algorithms only operate upon genotypes. To find a real solution to the problem, we thus require a mapping from genotype to phenotype. Such a mapping is known as a decoding. A decoding needs to map each genotype to at most one phenotype. If it would be possible to map a single genotype to multiple phenotypes, then we have no way of knowing what the correct solution is. For this reason, a decoding is not allowed to map a single genotype to multiple phenotypes.

## 3.2 Fitness Functions

In a genetic algorithm it is essential that we are able to evaluate the quality of solutions correctly [3, 21]. The goal of genetic algorithms is to find solutions of high quality. Of course, we cannot find high quality solutions if we do not know how to evaluate the quality of solutions. In a genetic algorithm a fitness function is used for the purpose of evaluating the quality of a solution. A fitness function takes as input a single genotype and it returns a value that represents the quality of the solution that is represented by the genotype. The fitness function can be as complex as is required for the problem. The fitness function can thus be a complex multi-objective optimization function for example. It is essential that a good fitness function is used for the problem at hand, i.e. a fitness function that is able to rank solutions in a correct way based on their quality. If we cannot accurately decide when a solution is better than another solution, then the genetic algorithm will most likely not be able to direct the search in the direction of better solutions.

## 3.3 Populations

The goal of using a genetic algorithm is to improve the overall quality of a set of solutions to the problem at hand, instead of just improving the quality of a single solution [20, 43]. In the genetic algorithm terminology such a set of solutions is known as the population. An important parameter of the population is the population size, which denotes the number of solutions that the population consists of. In general, the population size stays constant during the entire run of the genetic algorithm. There are however some cases in which the population size may change during the run of the genetic algorithm, but we will not discuss such situations here. If we have a larger population size, then there is a higher chance that the population contains more different solutions than in the case that the population size is very small. When a population contains a higher number of different solutions, the genetic algorithm is able to explore

a larger part of the search space. It would thus be beneficial to have large populations, because then we would expect that we are able to explore large parts of the search space and therefore we expect to find good solutions. The problem however is that a genetic algorithm will converge in $O(n \log n)$ function evaluations, in which $n$ denotes the population size. For very large populations, it thus takes a long time before the algorithm terminates. Therefore, the choice of the population size is always a trade-off between choosing to explore a larger part of the search space and choosing how long it should take for the algorithm to return a result.

At the start of the genetic algorithm, it is necessary to construct an initial population [50]. This initial population specifies where the search for a high quality solution should start. There are different ways in which the population can be initialized. The most popular way is to initialize the population randomly. Most of the times the random initialization will be done according to the uniform distribution. The reason for using a random initialization of the population is that we expect to create a diverse population in this way, which would mean that we would have a good coverage of the search space. A diverse population would thus allow the genetic algorithm to explore a large part of the search space, and therefore we expect to be able to find good solutions. Another way to initialize the population is to incorporate domain knowledge. This means that we either add solutions that are constructed by domain experts or that we incorporate some domain knowledge into the probability distribution that is used for the random initialization of the population. A problem with using solutions that are generated by domain experts is that the exploration may be limited to a small part of the search space. The benefit of random initialization over initialization with solutions generated by domain experts is therefore that we expect to be able to explore a larger part of the search space.

## 3.4 Selection

At the beginning of each iteration, the genetic algorithm needs to select the solutions in the population that are allowed to reproduce [43, 50]. The selection process aims at ensuring that more emphasis is placed on the fitter solutions during the reproduction process. If more emphasis is put on the fitter solutions during the reproduction process, then we are more likely to create new solutions of higher quality. The purpose of selection is thus to drive the search process in the direction of the fitter individuals. The degree to which the better solutions are emphasized in the selection process is known as the selection pressure. The higher the selection pressure, the more emphasis will be placed on the fitter solutions in the population. If the selection pressure is very high, then we are likely to lose the diversity in the population very fast. When we lose diversity in the population, we narrow the part of the search space to which the exploration is focused. If the selection pressure is very high, we will thus quickly lose the ability to explore large parts of the search space. On the other hand, it does take a long time for the population to converge if the selection pressure is very low. We thus do not want to have too high a selection pressure due to the fact

that it is more likely that the population will converge to a sub-optimal solution, and we do not want a very low selection pressure due to the fact that it will then take very long for the population to converge.

In this section a brief description of several selection techniques will be provided, starting with a description of tournament selection in Section 3.4.1. In Section 3.4.2 a description of ranking selection will be given. Finally, Section 3.4.3 will focus on truncation selection.

### 3.4.1 Tournament Selection

In tournament selection a set of $n$ solutions is selected from the population, where $n$ is equal to the size of the population [50]. Each of the $n$ slots in this set is filled by holding a tournament of size $k$. In a tournament of size $k$, $k$ solutions are selected from the population at random. From these solutions the fittest is selected to fill the slot. The tournament size $k$ is used to control the selection pressure of the algorithm. Because each solution is selected at random and because $k$ solutions are selected for each tournament, we expect that each solution is selected in $k$ different tournaments. This means that we expect to fill $k$ slots with the fittest solution in the population. Furthermore, the chance of filling a slot with a very bad solution decreases if the tournament size becomes bigger. This follows directly from the fact that there is a higher chance to also use a fitter solution in the tournament when we use more different solutions in each tournament. A larger tournament size thus corresponds with a higher selection pressure. Usually, a tournament size of two is used. A tournament size above five is rare, because then the selection pressure would be too high.

### 3.4.2 Proportional Selection

Proportional selection is a selection technique that bases the probability of selecting a solution on its fitness [43, 50]. The probability of selecting a fit solution is higher than the probability of selecting a less fit solution. The probability that a solution $I$ is selected is equal to the fitness of $I$ divided by the total fitness of the population, as described by Equation 3.1.

$$p_{selected}(I) = \frac{fitness(I)}{\sum_{j=1}^{populationsize} fitness(I_j)} \tag{3.1}$$

Based on the selection probabilities of the solutions, each solution can be assigned a part of the interval $[0, 1]$. For each of the solutions that need to be selected, a value in the interval $[0, 1]$ is randomly picked and the corresponding solution is selected. This is analogous to the idea of spinning a roulette wheel, in which the different slots correspond to the different solutions and in which the size of each slot is based on the probability of selecting that solution. Therefore, proportional selection is also known as roulette wheel selection. The downside of using this technique is that there may be a solution that has a fitness that

is much higher than the fitness of all the other solutions in the population. In this case, it is not unlikely that this solution takes over the entire population. The diversity of the population will then thus be lost.

### 3.4.3 Truncation Selection

In truncation selection a single parameter $T$ is used [4]. This parameter $T$ is known as the truncation threshold. It is used to determine which of the solutions are eligible to be selected for reproduction. Truncation selection starts by sorting the population on the basis of the fitness of the solutions. Thereafter, the solutions for reproduction are selected from the fraction $T$ of the best solutions. Each of these solutions has the same probability of being selected. Let us illustrate truncation selection by an example. Suppose that the population consists of a hundred solutions and that $T = 0.85$. This is the situation in which we want to select the top 85% of the solutions, where the ranking of the solutions is determined by their fitness value. In this situation the fifteen solutions with the lowest fitness are discarded and the set of solutions that is to be used in the reproduction process is selected from the remaining solutions. Each of these remaining solutions has a probability of $\frac{1}{85}$ of being chosen each time that a solution is selected.

## 3.5 Crossover

In genetic algorithms, solutions can be reproduced by using crossover operators [20, 22, 43]. Crossover operators combine information from the given parent solutions into new offspring solutions. Usually, crossover operators are used to combine two parent solutions into two offspring solutions. There are however also crossover operators that combine more than two parent solutions and there are also crossover operators that produce only a single offspring solution. The idea behind crossover is that it is desirable to combine the good features from the parent solutions in the offspring solutions, because then we would expect that the offspring solutions have a higher fitness than the parent solutions. This means that the search progresses in the direction of the fitter solutions. When using crossover operators, there is a choice to apply them all the time or to apply them with a non-zero probability. Any crossover operator is used to create new solutions, which are used for exploring the search space. Crossover operators are also used for combining the good features from the parent solutions, i.e. they are used for exploiting the structure of the parent solutions. The problem is that a better exploration strategy causes a worse exploitation strategy, and vice versa. Therefore, it is not possible to have both a perfect exploration strategy and a perfect exploitation strategy. This means that any good crossover operator needs to provide an adequate trade-off between exploration and exploitation

In this section a brief description of two crossover operators will be provided. Section 3.5.1 will provide a description of uniform crossover, and in Section 3.5.2

a description of N-point crossover will be given.

### 3.5.1 Uniform Crossover

Uniform crossover creates two offspring solutions from two parent solutions by copying each gene from one of the parent solutions [22]. For each gene, a random choice is made about whether we need to copy the gene from the first parent solution to the first offspring solution and from the second parent solution to the second offspring solution or whether we need to copy the gene from the first parent solution to the second offspring solution and from the second parent solution to the first offspring solution. To make these decisions, a binary crossover mask is constructed at random. For the first offspring solution we copy a gene from the first parent solution whenever the crossover mask contains a 1 at the corresponding position and we copy a gene from the second parent if the crossover mask has a 0 at the corresponding position. The opposite is done to create the second offspring solution. An example of uniform crossover can be found in Figure 3.1. Uniform crossover is employed by many different genetic algorithms, one of them being the SAGA algorithm [36].

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent 1: | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| Parent 2: | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Mask: | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| Child 1: | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| Child 2: | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

Figure 3.1: An example of uniform crossover

### 3.5.2 N-point Crossover

N-point crossover, also known as multi-point crossover, is a crossover technique that starts by selecting $N$ crossover points randomly [22, 43]. These $N$ crossover points define the points at which the parent solutions can be split, i.e. they define the points at which the parent solutions can be divided into separate blocks. In this way, a total of $N + 1$ blocks is created for each parent solution. Each block is copied to one of the two offspring solutions. The two offspring solutions are created by switching, at each crossover point, between copying blocks from the first parent solution and from the second parent solution. An example of N-point crossover for $N = 4$ can be found in Figure 3.2, where the lines represent the crossover points. Two instances of N-point crossover

that are often used, are one-point crossover and two-point crossover. One-point crossover is also one of the crossover operators that is employed by the SAGA algorithm [36].

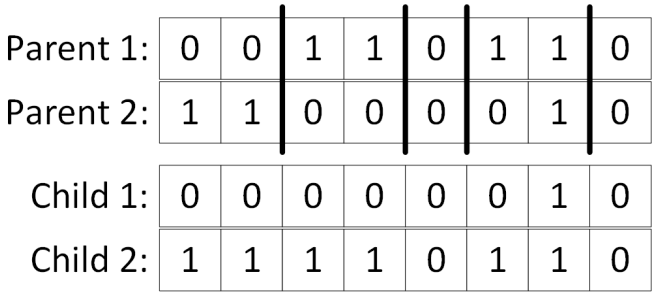



Figure 3.2: An example of 4-point crossover

## 3.6 Mutation

After new solutions have been generated by the use of a crossover operator, the solutions will undergo mutation [20, 43]. The way in which the solutions are mutated, is dictated by the specific mutation operator that is used. There are many different mutation operators that can modify the solutions in different ways. Some mutation operators can only modify a single solution at any given time, while other mutation operators change multiple solutions all at once. Mutation operators are used for the purpose of introducing new genetic material into the population. Mutation operators are thus used for maintaining diversity in the population. During crossover, genetic information from the parent solutions is exploited in order to create new offspring solutions. Crossover does not introduce any new information and thus causes a decrease in the diversity of the population. This also means that the ability to explore certain parts of the search space is lost. Therefore, the search may get stuck in a local optimum. Mutation operators overcome this problem by introducing new genetic material. The modifications made by mutation operators need to be random and unbiased to ensure that they can create any possible solution, which means that the entire search space can be investigated. If the mutations are too large, then no knowledge about fit solutions will be exploited and the algorithm will then basically perform a random search of the search space. Therefore, mutation may not occur to often. For this purpose a small value of the mutation probability, which determines how often a part of the solution will be modified, has to be chosen.

In this section a brief description of several mutation operators will be provided. Section 3.6.1 will give a brief description of the bit flip mutation operator. In Section 3.6.2 a brief description of the interchanging mutation operator will be provided. Finally, a brief description of an MSA domain specific mutation operator will be given in Section 3.6.3.

### 3.6.1 Bit Flipping

The bit flip mutation can only be applied to binary encoded genotypes [43]. The bit flip mutation operator flips the value of each gene from 0 to 1 and from 1 to 0 with a certain probability. A mutation mask is created to perform the bit flips. In this mutation mask a value of 1 indicates that the bit needs to be flipped. An example of the bit flip mutation can be found in Figure 3.3.

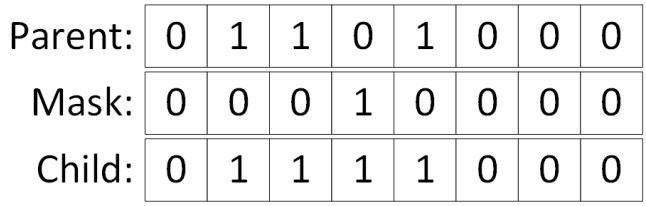



Figure 3.3: An example of the bit flip mutation

### 3.6.2 Interchanging

The interchanging mutation operator randomly selects two genes in the genotype and it swaps their values [43]. It can be applied to genotypes of any encoding. An example of the interchanging mutation operator can be found in Figure 3.4, where the shaded genes are the genes that need to be swapped.

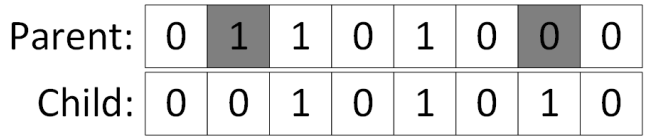
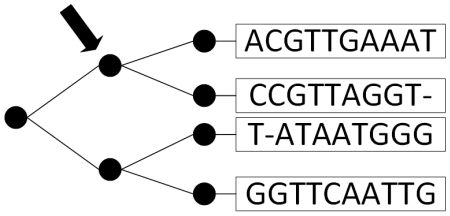



Figure 3.4: An example of the interchange mutation

### 3.6.3 Gap Insertion

The gap insertion mutation operator is a domain specific mutation operator for the MSA problem that is used by the SAGA algorithm [36]. This mutation operator adds a gap to each of the sequences in the multiple sequence alignment. The size of this gap is randomly chosen and it is the same for all the sequences. First of all, the set of sequences needs to be split into two subsets. These subsets are created by randomly selecting a point at which to split the underlying phylogenetic tree of the alignment and splitting the set of sequences based on the sequences in the subtrees that are created. Thereafter, a random position in the first subset of the sequences is chosen. For each sequence in this subset,

a gap is then inserted at this position. Finally, a position in the second subset of the sequences is selected. This position should be within some predefined maximum distance of the point selected for the first subset. For each sequence in this second subset, a gap is then inserted at this position. An example of gap insertion mutation with a gap size of two can be found in Figure 3.5. In this example the gap is inserted after the third gene in the first subset of sequences and the gap is inserted after the sixth gene in the second subset of sequences. The split point is indicated by the arrow in Figure 3.5a, it can be seen that the sequences are divided into two subsets of two sequences each. The resulting alignment can be found in Figure 3.5b.




| | |
|---|---|
| Seq 1 | ACG--TTGAAAT |
| Seq 2 | CCG--TTAGGT- |
| Seq 3 | T-ATAA--TGGG |
| Seq 4 | GGTTCA--ATTG |

(a) The underlying phylogenetic tree

(b) The resulting alignment

Figure 3.5: An example of the gap insertion mutation

## 3.7 Replacement

At the end of each generation of the genetic algorithm, a choice has to be made about which of the solutions are carried over to the next generation [20, 43]. During each generation, new solutions are created from the population through crossover and mutation. Usually, the size of the population remains constant during the entire run of the algorithm. It is thus necessary to decide which solutions of the population need to be replaced by the new solutions that have been generated, if any need to be replaced at all. The strategy to decide which solutions are replaced is known as the replacement strategy.

There are two different kinds of replacement strategies [45]. The first type of replacement strategy is known as generational replacement. In generational replacement, each generation $n$ new solutions are constructed from a population of size $n$. In other words, the number of newly generated solutions is the same as the number of solutions in the population. These new solutions will replace the entire population. There is one problem with this approach. The problem is that there is no guarantee that the new solutions will be fitter than the solutions in the population, and therefore it is possible that good solutions are thrown away in favor of bad solutions. To alleviate this problem slightly, generational replacement is most often combined with an elitist strategy. This elitist strategy ensures that one or a few of the fittest solution from the population are not replaced. Although such a strategy alleviates the problem a little, it is still possible that a lot of good solutions are discarded.

The second type of replacement strategy is known as steady-state reproduction [43, 45, 50]. In steady-state reproduction, each generation $\lambda$ new solutions are created from a population of size $\mu$. Usually, $\lambda$ is much smaller than $\mu$. Only a relatively small number of solutions will thus be replaced. There are different methods for deciding which solutions should be replaced in steady-state reproduction, but usually the fittest solutions from both the population and the new solutions are carried over to the next generation.

## 3.8 Termination

Genetic algorithms are stochastic search methods that aim at finding the optimal solution to the problem a hand [20, 41]. These algorithms could terminate once the optimal solution has been found. Unfortunately, the optimal solution is quite often not known. Therefore, a genetic algorithm has no way of knowing when the optimal solution is reached. In this case, a genetic algorithm does not have any indication of when it needs to terminate. Furthermore, there is no guarantee that a genetic algorithm will actually find the optimal solution. Even if the optimal solution is known, there is thus no way to guarantee that a genetic algorithm will ever terminate. For this reason, a termination criterion has to be specified. There can be a lot of different termination criteria, but there are a few termination criteria that are used most often:

1. A maximum amount of time has elapsed since the start of the run of the algorithm.
2. The total number of fitness evaluations has reached a predefined limit.
3. The overall improvement of the fitness of the population stays under a predefined threshold for a specified number of generations.
4. The diversity of the population becomes less than a specified threshold.

# Chapter 4

# The Gene-Pool Optimal Mixing Evolutionary Algorithm

For many problems it is essential that genetic algorithms are capable of learning the relations that exist between the variables, which is known as linkage learning [12]. In the case that no linkage information can be exploited, it is difficult for genetic algorithms to solve many problems. This is due to the fact that basic genetic algorithms, such as described in Chapter 3, employ recombination operators that can be very disruptive with respect to the building blocks of good solutions. These recombination operators can be very disruptive if they have not been designed carefully in order to incorporate required knowledge about the problem at hand. When using such disruptive recombination operators, there is a large chance that the offspring solutions will have a lower fitness than the parent solutions, which is described by Yu and Goldberg [52] in the following way:

> *If linkage exists between two genes, recombination might result in lowly fit offspring with high probability if those two genes are not transferred together from parents to offspring.*

In case that the recombination operators have not been designed very carefully, there is thus only a small probability that the offspring solutions will have a higher fitness than the parent solutions. This means that we expect to find at most a few fitter solutions during each generation. It can however also happen that no fitter solution is found during a single generation. For the reason that we expect to find only a few or no fitter solutions during any single generation, it will take a long time for the population to converge. In case that the variables are not all independent of one another, it can thus be very difficult for genetic algorithms to efficiently search the entire search space.

Recombination operators can thus be very disruptive for problems in which there exist certain relations between the problem variables. It might however not be directly intuitive why this is the case. Therefore, we will illustrate how disruptive recombination operators can be by an example. Suppose that we have two solutions. Each solution is a string of eight bits. The first solution is 01010101 and the second solution is 10101010. The fitness of each solution is obtained by taking the sum of two different fitness functions. The first fitness function calculates the fitness of the bits at the even numbered positions and the second fitness function calculates the fitness of the bits at the odd numbered positions. Both fitness functions have their optimum at 1111. These solutions thus consist of two building blocks. Now assume that we recombine these solutions with the help of one-point crossover, where the crossover point is after the fourth gene. We then get the solutions 01011010 and 10100101. In both these solutions the optimum 1111 is no longer present in any of the two building blocks. The optimal building blocks thus get disrupted by not taking into account the relation that exists between the genes. The building blocks would also have been disrupted if we would have used a different crossover point or if we would have used different crossover operators that have not been designed carefully for this specific problem. We hope that this example illustrates how disruptive recombination operators can be for problems in which there exist relations between the genes, and thus how important it is that we are able to exploit linkage information in genetic algorithms.

Standard genetic algorithms are thus not able to exploit linkage information [30]. One class of genetic algorithms that is able to learn and exploit linkage information, is the class of estimation of distribution algorithms (EDAs). These algorithms learn the linkage information by estimating a complete probability distribution during each generation. It is quite expensive to estimate a complete probability distribution. Thus, while EDAs can learn and exploit linkage information, this comes at the cost of a high computational workload. There is however another genetic algorithm than can learn linkage information without the need to estimate an entire probability distribution, namely the Gene-Pool Optimal Mixing Evolutionary Algorithm (GOMEA). The structure of the GOMEA algorithm is depicted in Algorithm 2 [6]. It can be seen that the GOMEA algorithm starts the same way as standard genetic algorithms do. It first creates a population of random solutions. These solutions are then iteratively improved during each iteration of the algorithm. Each iteration a linkage model is built, which represents the relations between the variables. This model is used in the Optimal Mixing procedure, which creates the set of offspring solutions. The set of offspring solutions replaces the entire population at the end of each generation, i.e. GOMEA uses a generational replacement strategy.

This chapter will focus on explaining the components of the GOMEA algorithm in more detail, starting with a description of the structure of the linkage model in Section 4.1. Thereafter, a description of the Optimal Mixing procedure will be given in Section 4.2. Finally, Section 4.3 will describe a technique for improving the rate at which the population converges.

**Algorithm 2** The GOMEA algorithm

```
1: procedure GOMEA(n, l)                    ▷ population size n and problem size l
2:     for i ∈ {0, 1, ..., n − 1} do
3:         P_i ← CreateRandomSolution()
4:         EvaluateFitness(P_i)
5:     end for
6:     while Termination criterion not satisfied do
7:         x^best ← argmax_{x∈P}{fitness[x]}
8:         S ← TournamentSelection(P, n, 2)
9:         LearnModel(S)
10:        for i ∈ {0, 1, ..., n − 1} do
11:            O_i ← OptimalMixing(P_i)
12:        end for
13:        P ← O
14:    end while
15: end procedure
```

## 4.1 FOS Model

The GOMEA algorithm uses a generic linkage model to represent the groups of variables that are related to one another in a certain way, i.e. the groups of variables that are important in determining the quality of a solution [6, 46]. The model that is used by the GOMEA algorithm is known as a family of subsets (FOS) model, denoted by $\mathcal{F}$. A FOS is a set of subsets of a certain set $S$, which is often the set of variable indices. Each subset can possibly appear more than once in the FOS model and each variable can occur in more than one FOS subset. We require that each variable is present in at least one of the FOS subsets, because it would not be possible to modify the corresponding variable during the recombination process otherwise. The FOS model can also be defined in a more mathematical way. In this case, we say that a FOS model is a subset of the power set of $S$, i.e. of $\mathcal{P}(S)$. The set $S$, which represents the set of variable indices, can be represented as $\{1, 2, \ldots, l\}$, where $l$ denotes the number of problem variables. In this case, the FOS model $\mathcal{F}$ can be represented as $\mathcal{F} = \{\mathbf{F}^1, \mathbf{F}^2, \ldots, \mathbf{F}^{|\mathcal{F}|}\}$, where $\mathbf{F}^i \subseteq \{1, 2, \ldots, l\}$ and $i \in \{1, 2, \ldots, |\mathcal{F}|\}$. Let us give an example of a FOS model. Suppose that we have a set of solutions, in which each solution is a string of eight bits. In this situation, a possible FOS model would be $\{\{1\}, \{2, 3, 4, 5\}, \{4, 5, 6\}, \{6, 7\}, \{8\}\}$.

There are different FOS structures that are used by the GOMEA algorithm. Each of these structures is used to represent different relations between the variables. In this section we will look at three of these FOS structures. In Section 4.1.1, we will look at the univariate FOS structure. Thereafter, we will describe the marginal product FOS structure in Section 4.1.2. Finally, we will focus on the linkage tree FOS structure in Section 4.1.3.

### 4.1.1 Univariate Structure

The univariate structure is the simplest of the different FOS structures [30]. The univariate structure represents a mutual independence model, in which there is absolutely no relation between the variables. The FOS model thus consists of only singleton sets, i.e. $\forall i \in \{1, 2, \ldots, l\} : \mathbf{F}^i = \{i\}$. The benefit of using the univariate structure is that it is not necessary to perform linkage learning. Therefore, the algorithm can complete each iteration faster. The downside of using the univariate structure is that we may oversimplify the problem. We assume that there are no relations between the variables, while there can exist relations between the variables in reality. Therefore, it can be more difficult for the algorithm to explore the search space in an efficient manner.

### 4.1.2 Marginal Product Structure

In the marginal product structure, the FOS model consists of mutually exclusive subsets of variables [46]. Each variable thus occurs in exactly one FOS subset, i.e. $\forall i, j \in \{1, 2, \ldots, |F|\}, i \neq j : \mathbf{F}^i \cap \mathbf{F}^j = \emptyset$. Variables in the same FOS subset are related in some way, while variables in different FOS subsets are independent of one another. The marginal product structure is thus only able to represent relations between the variables themselves. It is not possible to also represent higher-order relations between different groups of variables with the marginal product structure.

### 4.1.3 Linkage Tree Structure

The linkage tree structure can be used to represent more complex relations between the variables than that is possible with both the univariate structure and the marginal product structure [6, 8, 30, 46]. In the linkage tree structure it is not only possible to model relationships between individual variables, but it is also possible to model relationships between groups of variables. The leaf nodes in the linkage tree correspond to the singleton FOS subsets, i.e. are the same FOS subsets as used in the univariate structure. Each internal node of the linkage tree has exactly two children and its FOS subset contains the variables that are in the union of the FOS subsets of its children. For any internal node, the FOS subsets of its children are mutually exclusive. The root of the linkage tree is the FOS subset that contains all variables. When GOMEA uses a linkage tree. it is also know as the Linkage Tree Genetic Algorithm (LTGA) or as LT-GOMEA.

A linkage tree is constructed using a hierarchical clustering method [5, 6, 46]. The hierarchical clustering method can use different similarity or distance measures for determining which FOS subsets need to be merged. A similarity measure that is often used is that of mutual information (MI). The formula for computing the mutual information of two FOS subsets $\mathbf{F}^i$ and $\mathbf{F}^j$ is given in

Equation 4.2, and the corresponding formula for computing the entropy is given in Equation 4.1. In Equation 4.1, $\Omega_{\mathbf{F}^i}$ denotes the set of all possible configurations of the variables in the FOS subset $\mathbf{F}^i$. The hierarchical clustering method starts from the singleton FOS subsets and it merges the pair of singleton FOS subsets that have the highest MI value. Thereafter, the hierarchical clustering methods iteratively keeps merging the most similar nodes into a new node in the linkage tree. The hierarchical clustering procedure terminates once a FOS subset of all variables has been created. In case that one of the nodes is not a singleton FOS subset, it is too expensive to compute the MI value of the pair of FOS subsets from a computational point of view. Therefore, the unweighted pair group method with arithmetic mean (UPGMA) is used to determine the similarity of two FOS subsets in this case, see Equation 4.3. This hierarchical clustering method for constructing the linkage tree can be implemented to run in $O(nl^2)$ time, where $n$ denotes the population size and $l$ denotes the number of variables in each solution.

$$H(\mathbf{F}^i) = \sum_{x \in \Omega_{\mathbf{F}^i}} -P(\mathbf{F}^i = x) \cdot \log\left(P(\mathbf{F}^i = x)\right) \tag{4.1}$$

$$MI(\mathbf{F}^i, \mathbf{F}^j) = H(\mathbf{F}^i) + H(\mathbf{F}^j) - H(\mathbf{F}^i \cup \mathbf{F}^j) \tag{4.2}$$

$$MI^{UPGMA}(\mathbf{F}^i, \mathbf{F}^j) = \frac{1}{|\mathbf{F}^i| \cdot |\mathbf{F}^j|} \cdot \sum_{X \in \mathbf{F}^i} \sum_{Y \in \mathbf{F}^j} MI(X, Y) \tag{4.3}$$

## 4.2 Optimal Mixing

The GOMEA algorithms creates new solutions through an Optimal Mixing (OM) procedure [6, 16, 30, 46]. The specific OM procedure that is used by GOMEA is known as Gene-Pool Optimal Mixing (GOM). GOM is a specific instance of OM, in which the donor solutions are randomly chosen from the entire population. In the GOM procedure exactly one offspring solution is created for each solution in the population. GOM starts by creating a copy of the parent solution. Thereafter, GOM tries to iteratively improve this copied solution. This iterative procedure loops over all FOS subsets in a random order. For each FOS subset, a random donor solution is selected from the population. The variables specified by the FOS subset are copied from the donor solution to the new solution. If the fitness of this new solution is not worse than the fitness of the old solution, then this change is accepted. Otherwise, the change is reverted. The reason for accepting new solutions that have the same fitness as the original solution is that we are able to traverse plateaus in the fitness landscape in this way. Therefore, the algorithm is less likely to get stuck in a local optimum. It should be noted that in case that the linkage tree FOS structure is used, the FOS subset of the root node is excluded from the GOM procedure. The FOS subset of the root node of the linkage tree contains all variables. If this FOS subset would be used in the GOM procedure, it is possible that entire solutions get replaced. This could mean that the diversity of the population decreases too fast. The FOS subset of the root node is thus excluded to ensure that the population does not converge too fast. Pseudo-code for the GOM procedure can be found in Algorithm 3.

**Algorithm 3** Gene-Pool Optimal Mixing

```
1: procedure GOM(x)                                          ▷ solution x
2:     b ← o ← x
3:     fitness[b] ← fitness[o] ← fitness[x]
4:     for i ∈ {1, 2, ..., |F|} do
5:         p ← Random({P_0, P_1, ..., P_{n−1}})
6:         o_{F^i} ← p_{F^i}
7:         if o_{F^i} ≠ b_{F^i} then
8:             EvaluateFitness(o)
9:             if fitness[o] ≥ fitness[b] then
10:                b_{F^i} ← o_{F^i}
11:                fitness[b] ← fitness[o]
12:            else
13:                o_{F^i} ← b_{F^i}
14:                fitness[o] ← fitness[b]
15:            end if
16:        end if
17:    end for
18: end procedure
```

## 4.3 Forced Improvement

Forced Improvement (FI) is a technique that is used for increasing the convergence rate of the population [6, 7, 16, 30]. FI is executed after the GOM phase has executed. In essence, FI applies a second GOM iteration to the solution. There are however three differences between the standard GOM phase and the GOM phase that is employed by the FI procedure. The first difference is that the donor solutions are not randomly selected in the FI phase, instead the best solution from the population is used as the donor for each FOS subset. Secondly, only changes that do strictly improve the fitness of the solution are accepted during the FI phase. This is for the reason that the diversity of the population would decrease too fast in case that we accept copying parts of the best solution that do not result in a fitter solution. The third and final difference is that the FI phase stops as soon as the fitness of the solution has been increased. This is again for the reason that we want to ensure that the diversity of the population is not lost too fast. If the FI phase is also not capable of increasing the fitness of the solution, then the solution is replaced by the best solution in the population. The FI phase can be entered if one of two possible situations occurs. The first situation in which the FI phase is entered, is when the solution has not been changed during the initial GOM phase. The second situation in which the FI phase is entered, is when the best solution has not changed for more than $1 + \lfloor \log_{10}(n) \rfloor$ generations. The number of generations in which the best solution has not changed, is known as the no-improvements stretch (NIS). Because we accept changes that do not strictly improve the fitness of the solution during the first GOM phase, it is possible that the algorithm keeps moving over a fitness plateau without ever leaving it. The second situation in which the FI phase is entered, aims to deal with this situation.

# Chapter 5

# The Algorithm

In Section 1.2, we have already briefly introduced the approach for finding high quality MSAs that was proposed by Botta and Negro [9] (and by Negro [33] in his corresponding master thesis). Their approach is to evolve a set of positional weight matrices (PWMs), instead of directly evolving a set of MSAs. These PWMs act as multiple alignment profiles for the given set of input sequences. For the purpose of evolving a set of PWMs, they use a steady-state genetic algorithm. This steady-state genetic algorithm was implemented using the GALib library [51]. They tested the performance of their algorithm under the settings that 50% of the population was replaced during each generation, that the population consisted of 200 PWMs and that the algorithm was run until there either has not been a change in the fitness of the solutions for ten generations or until 500 generations have been evaluated. They do not clearly specify which selection scheme was used, but we suspect that they used the roulette wheel selection scheme, since this is the default selection scheme for steady-state genetic algorithms in the GALib library. They compared the performance of their algorithm against many well-known MSA algorithms, such as SAGA [36], Clustal W [47] and T-Coffee [37]. In most test cases, their algorithm found solutions of a higher quality. In only a single test case, their algorithm found solutions that have a significantly lower quality. Their results thus seem to indicate that their algorithm performs quite well in comparison with all these other algorithms. Because their algorithm seems to perform quite well and because their algorithm is relatively simple when compared to some of the other MSA algorithms, their approach seems very interesting.

In this chapter we will explain the components of the algorithm in more detail, starting with a description of the representation of the problem in Section 5.1. Thereafter, we will describe how the fitness of a PWM is calculated in Section 5.2. Section 5.3 will focus on explaining the crossover operator that is used, while Section 5.4 will focus on describing the mutation operator. Finally, we will describe how we can construct an MSA from a given PWM in Section 5.5.

## 5.1 Problem Representation

As we have mentioned before, the algorithm does not evolve a set of MSAs directly [9, 33]. It instead evolves a set of PWMs. This is contrary to many other algorithms for solving the MSA problem, of which we have described a few in Chapter 2. Before we define what PWMs are, let us briefly define MSAs more formally in order to be able to better understand the differences between the different representations. MSAs are alignments of sets of sequences [10, 26]. In this context, a sequence is defined as a set of characters. A more formal definition of a sequence is found in Definition 5.1.1. In the case of biological sequence alignment, a sequence can for example be a string of DNA. In an MSA, the sequences have been aligned by inserting gaps into the sequences in such a way that we aim to maximize the similarity between the sequences. A formal definition of a multiple sequence alignment can be found in Definition 5.1.2.

**Definition 5.1.1** (Sequence)**.** A sequence of length $k$, is a set of $k$ characters taken from an alphabet of characters $\Sigma$. A sequence $s$ thus is a set $s = \{c_1, \ldots, c_k\}$, such that $\forall i \in \{1, \ldots, k\} : c_i \in \Sigma$.

**Definition 5.1.2** (Multiple sequence alignment)**.** Given a set of $n \geq 3$ sequences $\{S_1, \ldots, S_n\}$ of varying lengths $l_1$ to $l_n$, a multiple sequence alignment is a set of sequences $\{\bar{S}_1, \ldots, \bar{S}_n\}$ that is obtained by inserting gaps (represented by the character '-') into the sequences $\{S_1, \ldots, S_n\}$. The sequences $\{\bar{S}_1, \ldots, \bar{S}_n\}$ satisfy the following properties:

1. $\forall i, j \in \{1, \ldots, n\} : \bar{l}_i = \bar{l}_j$, where $\bar{l}_i$ and $\bar{l}_j$ denote the length of sequences $\bar{S}_i$ and $\bar{S}_j$.
2. There is no position $p$ with $1 \leq p \leq \bar{l}$, where $\bar{l}$ denotes the length of the sequences in the multiple sequence alignment, in which all of the sequences have a gap character.

Now let us define what a PWM is. A PWM is a matrix that describes for each character and position in the MSA, the probability that the specified character occurs at the specified position in one of the sequences of the MSA [9, 33]. From Definition 5.1.3 it follows that each row in the PWM represents either a gap character or a character from the alphabet of characters that can occur in one of the sequences. Furthermore, it follows that each column of the PWM represents a single position in the sequences in the MSA. One reason for favoring the PWM representation of solutions over the MSA representation of solutions, is that PWMs can be more compact than MSAs. The length of the sequences in the MSA is the same as the length, i.e. the number of columns, of the PWM. This means that when the MSA consists of more sequences than there are characters in the alphabet of characters that can occur in the MSA, then a PWM has smaller dimensions than an MSA has. Another reason for modeling solutions as PWMs, is that it is possible to use standard genetic operators for modifying PWMs. If the solutions would directly be modeled as MSAs, then it would be necessary to use more complex genetic operators.

**Definition 5.1.3** (Positional weight matrix)**.** A positional weight matrix is an $m \times n$ matrix that provides a compact description of the information content of a multiple sequence alignment. The number of rows $m$ is equal to the number of characters that can occur in the multiple sequence alignment, i.e. it is equal to $card(\Sigma) + 1$ (the size of the alphabet of characters that can occur in a sequence plus the gap character). The number of columns $n$ is equal to the length of the sequences in the multiple sequence alignment. Each entry in the positional weight matrix represents the probability that a certain character occurs at a certain position in one of the sequences of the multiple sequence alignment. The probability that a character $c$ occurs in one of the sequences of the multiple sequence alignment at a position $p$ is denoted by $PWM[c][p]$.

Each PWM is initialized in a random manner [33]. First of all, a random length is selected for the PWM. It should be noted that the length of the PWM is never allowed to be smaller than the length of the longest input sequence, because otherwise it would not be possible to construct a valid MSA from the PWM for the given set of input sequences. A matrix of this randomly selected length is initialized with only zeros. Thereafter, a probability of one is iteratively distributed over all rows in a column. This is done for each column separately. A PWM that is constructed using this procedure, is guaranteed to be consistent. This means that, for each column, the probabilities sum up to one. The procedure for initializing a PWM is depicted in Algorithm 4.

**Algorithm 4** Procedure for initializing PWMs

```
1:  procedure PWMInitialize(seq)   ▷ The given set of input sequences seq
2:      Get the length l of the longest sequence in seq
3:      Randomly choose a value r, with r ≥ l
4:      Initialize a PWM of length r with only zeros
5:      for j = 1 to PWM.length do
6:          probLeft = 1
7:          while probLeft > 0 do
8:              Let per be a random percentage
9:              toAdd = probLeft · per
10:             Randomly select a row i
11:             PWM[i][j] = PWM[i][j] + toAdd
12:             probLeft = probLeft − toAdd
13:         end while
14:     end for
15: end procedure
```

After we have randomly initialized the PWMs, we are not done yet [33]. This is for the reason that the PWMs could correspond to alignments that have one or more positions for which all of the sequences contain a gap. As can be concluded from Definition 5.1.3, such alignments are not valid multiple sequence alignments. To ensure that the PWMs correspond to valid multiple sequence alignments, we thus need to update the PWMs. For this purpose, we start by removing the positions that contain a gap in all of the sequences from the corresponding alignments. Thereafter, we construct new PWMs from these

valid multiple sequence alignments and replace the old PWMs with these newly generated PWMs. It should be noted that this procedure is also applied each time the algorithm generates new PWMs through reproduction and mutation, because in that situation it is also possible that the generated PWMs do not correspond to valid multiple sequence alignments. The procedure for updating a PWM is depicted in Algorithm 5.

**Algorithm 5** Procedure for updating PWMs

```
1: procedure PWMUpdate(pwm)                ▷ The PWM to update pwm
2:     Construct the alignment tmpAlignment corresponding to pwm
3:     Remove positions that only contain a gap from tmpAlignment
4:     Construct a new PWM tmpPWM from tmpAlignment
5:     Replace pwm by tmpPWM
6: end procedure
```

## 5.2 Fitness Evaluation

One way of determining the quality of a PWM with respect to the corresponding MSA, is to take the sum of similarities between the PWM and each sequence in the MSA [9, 33]. Each sequence in the MSA is scored by taking the sum of all probabilities of the characters in the sequence. More formally, this is defined as:

$$score(seq, PWM) = \sum_{i=0}^{L-1} PWM[seq[i]][i] \tag{5.1}$$

where $L$ denotes the length of both the sequence and the PWM and $seq[i]$ denotes the character at position $i$ in the sequence. The fitness function is then defined as:

$$fitness(MSA, PWM) = \sum_{seq \in MSA} score(seq, PWM) \tag{5.2}$$

The benefit of this fitness function is that it is very simple and can be applied to all types of sequences, i.e. not only to biological sequences. However, it is too simple for biological sequences in the sense that it does not take into account that we want to minimize the number of gaps and that we would like a distribution of the gaps that makes the alignment as compact as possible. Furthermore, this fitness function does also not take into account that some alignments may be very unlikely from a biological perspective. For this reason, a more complex fitness function for determining the quality of a PWM with respect to the corresponding MSA was used in the algorithm. This fitness function makes use of a BLOSUM [18, 31] substitution matrix. To ensure that is it clear how this fitness function works, we will first provide a brief description of substitution matrices.

During the evolution of a biological sequence family, some substitutions of characters are more likely to happen than other substitutions are [18, 23, 31]. In this

setting, a substitution is defined as replacing a character at a certain position in the sequence by another character from the alphabet. When determining the quality of a biological sequence alignment, it is necessary to take into account how likely it is that certain substitutions occur. This is necessary for the reason that we do not want to assign a high score to an alignment that is constructed in such a way that it contains substitutions that never occur in reality. In other words, we do not want to assign a high score to an alignment that is very unlikely from a biological perspective. For biological sequence alignment algorithms, we thus require a way to represent how likely it is that certain substitutions occur. This information is often represented by substitution matrices. A well-known type of substitution matrices are the BLOSUM matrices, see Definition 5.2.1 for a definition of BLOSUM matrices. There are multiple BLOSUM matrices that are different in the sense that they were constructed from a set of sequences that had a different percentage of similarity between the sequences. The most widely used BLOSUM matrix is however the BLOSUM62 matrix, which was built from a set of sequences with at least 62% similarity.

**Definition 5.2.1** (Blocks amino acid substitution matrices)**.** Blocks amino acid substitution matrices (BLOSUM) are square matrices that are used to indicate how likely it is that one amino acid gets substituted in favor of another amino acid during evolution. Each amino acid is represented in one of the rows and also in one of the columns. Each entry in the matrix is a logarithm of odds score of the ratio of the frequency of the corresponding amino acid substitution divided by the frequency of the amino acid substitution that we would expect based on chance alone.

The fitness function that was used in the algorithm for finding high quality alignments of biological sequences, consists of two sub-functions [33]. The first sub-function is used for assigning a score to a character $c$ at a position $p$ in a single sequence:

$$charScore(c,p) = \begin{cases} \max\limits_{x \in alphabet} BLOSUM[c][x] \cdot PWM[x][p] & \text{if } c \text{ is a character} \\ -GOP \cdot (1 - PWM['-'][p]) & \text{if } c \text{ is a dash that opens a gap} \\ -GEP \cdot (1 - PWM['-'][p]) & \text{if } c \text{ is a dash that extends a gap} \end{cases} \tag{5.3}$$

where $BLOSUM[c][x]$ denotes the score of substituting character $c$ with character $x$ according to the BLOSUM matrix, $GOP$, $GOP \in \mathbb{N}$, denotes the penalty for opening a gap (a value of ten was used) and $GEP$, $GEP \in \mathbb{N}$, denotes the penalty for extending a gap (a value of one was used). Only internal gaps were penalized in the algorithm, i.e. only gaps that break up the sequence. By using this function, the algorithm takes into account the number of gaps and how the gaps are distributed. Furthermore, this function ensures that the algorithm takes into account that some alignments may be very unlikely from a biological perspective. This function was used in the second sub-function that was used to compute a score for a single sequence in the MSA:

$$seqScore(seq, PWM) = \sum_{i=0}^{L-1} charScore(seq[i], i) \tag{5.4}$$

where $L$ and $seq[i]$ have the same meaning as in Equation 5.1. Based on these two sub-functions, the fitness function that is used in the algorithm is defined as:

$$score(MSA, PWM) = \sum_{seq \in MSA} seqScore(seq, PWM) \tag{5.5}$$

It should be noted that the fitness function also contains a corrective factor for the gap score, which is used to take into account some of the spatial constraints of biological sequences. However, we do not use this corrective factor for the reason that it is not very clearly described and therefore we do not know how this corrective factor functions and how it needs to be implemented.

## 5.3 Crossover

The variable one-point crossover operator is used by the algorithm, with a probability of 0.4 [33]. The variable one-point crossover operator is very similar to the standard one-point crossover operator, as described in Section 3.5.2. The difference between them is that a random crossover point is selected for each of the parents separately by the variable one-point crossover operator. In other words, the crossover point does not need to be the same for both parents. The algorithm uses a slightly modified version of the variable one-point crossover operator. It is required that each PWM has a minimum length that corresponds to the length of the longest input sequence. By using the standard variable one-point crossover operator, it is possible to construct PWMs that are shorter than the minimum required length. If this is the case, then a part of the parent from after the crossover point is copied. An example of this crossover operator can be found in Figure 5.1, where the black lines represent the crossover points and the minimum length of a solution is five. To ensure that the first child solution is long enough, it is necessary to copy a single value from the first parent from after the crossover point. This crossover procedure is simpler than the crossover procedure of some other MSA algorithms. It is for example simpler than the crossover procedure that is used by the SAGA [36] algorithm in the sense that it is not necessary to schedule two different crossover operators.

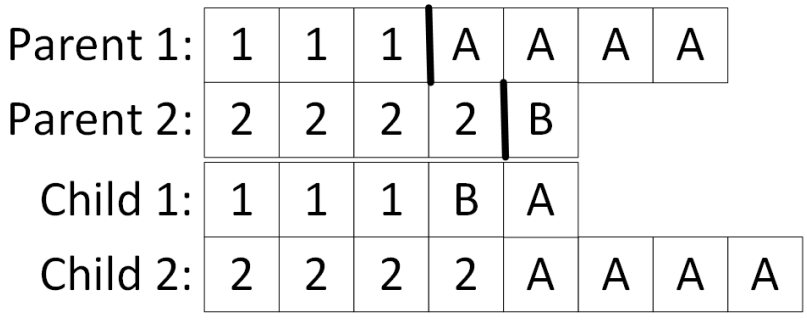



Figure 5.1: Example of the variable one-point crossover operator

## 5.4 Mutation

Two different mutation operators are employed by the algorithm [33]. The first mutation operator is used to be able to swap two columns from the PWM, while the second mutation operator is used for the purpose of changing the probabilities of two rows in a single column. The probability of applying either one of these mutation operators is 0.05. If mutation is applied, both mutation operators have the same probability of being selected. Pseudo-code for the mutation procedure that is employed by the algorithm can be found in Algorithm 6. This mutation procedure is simpler than the mutation procedure of several other algorithms for solving the MSA problem. This mutation procedure is simpler that the mutation procedure that is employed by the SAGA [36] algorithm for example. It is simpler in the sense that only two different mutation operators are used, instead of twenty. Furthermore, no scheduling scheme is necessary to determine which mutation operator is to be applied, instead both mutation operators will be applied with the same probability. Finally, the mutation operators are much simpler in the sense that it is not necessary to develop complex problem specific mutation operators.

**Algorithm 6** The mutation operator for evolving PWMs

1: **procedure** PWMMutation($PWM$) ▷ The matrix $PWM$ to be mutated
2: Let $f$ be the outcome of a random coin flip
3: **if** $f = HEADS$ **then**
4: Randomly choose a column $j_1$ from $PWM$
5: Randomly choose a column $j_2$ from $PWM$
6: Swap column $j_1$ and column $j_2$
7: **else**
8: Randomly choose a column $j$ from $PWM$
9: Randomly choose a row $i_1$ from $PWM$
10: Randomly choose a row $i_2$ from $PWM$
11: Let $per$ be a random percentage
12: $toAdd = PWM[i_1][j] \cdot per$
13: $PWM[i_1][j] = PWM[i_1][j] - toAdd$
14: $PWM[i_2][j] = PWM[i_2][j] + toAdd$
15: **end if**
16: **end procedure**

## 5.5 Constructing Multiple Sequence Alignments

The algorithm constructs an MSA by aligning each of the input sequences against the PWM [33]. To align a sequence against the PWM, a dynamic programming approach is used. This dynamic programming approach is very similar to the Needleman and Wunsch algorithm [32], except for the fact that we try to align a sequence against a PWM instead of aligning two sequences. In this approach, a score matrix and a choices matrix are constructed. The

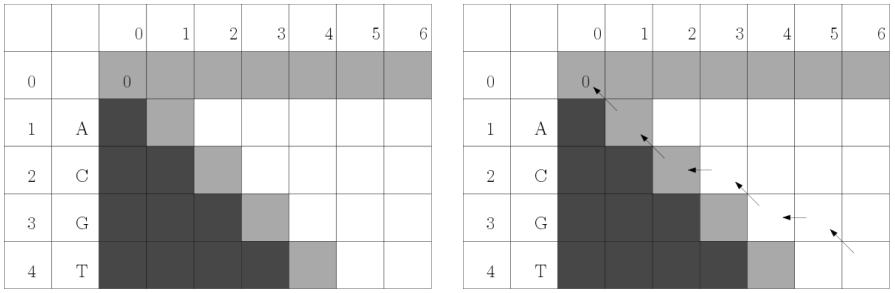
definition of the score matrix can be found in Definition 5.5.1. As described in this definition, each state in the score matrix represents the score of the best way to align the corresponding part of the sequence in the specified positions. For example, in Figure 5.2a the entry $ScoreMatrix[2][3]$ represents the score of the best alignment of the sequence "AC" in the first three positions. The dark gray boxes in this score matrix do not need to be computed for the reason that they do not correspond to valid alignments. For example, the entry $ScoreMatrix[2][1]$ corresponds to the score of the best alignment of the sequence "AC" in exactly one position, which obviously represents an impossible alignment. The light gray boxes are filled in an initialization step. The light gray boxes on the first row correspond to the alignment that contains only gaps, and the light gray boxes on the diagonal correspond to the alignment with no internal gaps. The remaining boxes can be filled by determining whether it is better to add the next character to the alignment or whether it is better to add a gap to the alignment. The entire procedure for filling the score matrix can be found in Algorithm 7.

**Definition 5.5.1** (Score matrix)**.** A score matrix is an $m \times n$ matrix that is used in aligning a sequence against a PWM. The number of rows $m$ is one larger than the number of characters in the sequence. The first row corresponds to a gap and the remaining rows correspond to the characters of the sequence. The order of the characters on the rows is the same as the order of the characters in the sequence. The number of columns $n$ is one larger than the number of columns of the PWM. The columns 1 to $n$ of the score matrix correspond to the columns 1 to $n$ of the PWM. Each entry $ScoreMatrix[i][j]$ represents the score of the best way to align the first $i$ characters from the sequence in exactly $j$ positions.

Along with the construction of the score matrix, a choices matrix is constructed, see Definition 5.5.2. The alignment of the given sequence can be constructed from the choices matrix by a traceback procedure that starts at the bottom right box of the matrix. In this traceback, it is only possible to move vertically or diagonally during each step. An example of a part of a choices matrix has been given in Figure 5.2b. If we start at the bottom right box in this example, we can see that we perform a diagonal move, which corresponds to adding the next character to the alignment. This means that we obtain the alignment "T". In the next step a vertical move is performed, which corresponds to adding a gap to the alignment. We thus obtain the alignment "-T". By completing this traceback procedure, we obtain the final alignment "AC-G-T".

**Definition 5.5.2** (Choices matrix)**.** A choices matrix is an $m \times n$ matrix that is used in aligning a sequence against a PWM. The number of rows $m$ is one larger than the number of characters in the sequence. The first row corresponds to a gap and the remaining rows correspond to the characters of the sequence. The order of the characters on the rows is the same as the order of the characters in the sequence. The number of columns $n$ is one larger than the number of columns of the PWM. The columns 1 to $n$ of the score matrix correspond to the columns 1 to $n$ of the PWM. Each entry $ChoicesMatrix[i][j]$ represents the state from which the corresponding entry of the score matrix was reached. For each entry, a diagonal pointer indicates that the corresponding state of the

score matrix was reached by adding the next character to the alignment and a vertical pointer indicates that the corresponding state of the score matrix was reached by to adding a gap to the alignment.




(a) An example of a score matrix for the sequence "ACGT" and a PWM of length six

(b) An example of a path through the score matrix for finding the alignment of the given sequence

Figure 5.2: Examples of the matrices that are used for finding the optimal alignment of a sequence against the PWM

**Algorithm 7** The procedure for filling the ScoreMatrix

1: **procedure** PWMComputeScoreMatrix($PWM$, $seq$)
2: $ScoreMatrix[0][0] = 0$
3: **for** $j = 1$ to $ScoreMatrix.length - 1$ **do** ▷ Initialize the first row
4: $ScoreMatrix[0][j] = ScoreMatrix[0][j-1] + charScore('-', j)$
5: **end for**
6: **for** $i = 1$ to $ScoreMatrix.height - 1$ **do** ▷ Initialize the diagonal
7: $ScoreMatrix[i][i] = ScoreMatrix[i-1][i-1] + charScore(seq[i-1], i)$
8: **end for**
9: **for** $i = 1$ to $ScoreMatrix.height - 1$ **do**
10: **for** $j = i + 1$ to $ScoreMatrix.length - 1$ **do**
11: $case1 = ScoreMatrix[i][j-1] + charScore('-', j)$
12: $case2 = ScoreMatrix[i-1][j-1] + charScore(seq[i-1], j)$
13: $ScoreMatrix[i][j] = \max\{case1, case2\}$
14: **end for**
15: **end for**
16: **end procedure**

# Chapter 6

# Computational Complexity Analysis

In this chapter we will analyze the computational complexity of the algorithm as proposed by Botta and Negro [9], of the univariate GOMEA algorithm, and of the Linkage Tree Genetic Algorithm. In this chapter we will also describe how our implementation of these algorithms differs from the implementation of these algorithms as described in Chapter 4 and in Chapter 5. We will start by analyzing the computational complexity of the initialization of the population in Section 6.1. Next, we will describe the computational complexity of updating a PWM in Section 6.2. Then, we will analyze the complexity of computing the fitness of a PWM in Section 6.3. Thereafter, we will analyze the complexity of the selection operator, the reproduction operator, the mutation operator and the replacement operator in respectively Section 6.4, in Section 6.5, in Section 6.6 and in Section 6.7. Finally, we will analyze the overall complexity of the algorithms in Section 6.8.

## 6.1 Initialization

The initialization of the population of PWMs is done according to the procedure that has been described in Section 5.1 [33]. There is however one small difference between our approach for initializing the PWMs and the approach for initializing the PWMs that has been described in Section 5.1. The difference between these approaches is that in our approach we also bound the number of columns from above, i.e. we limit the maximum number of columns that a PWM can have. We bound the maximum number of columns from above for the reason that we want to reduce the complexity of the initial PWMs. If we reduce the complexity of the initial PWMs, then less work is required to update these PWMs. We know that a valid multiple sequence alignment does not contain columns with only gaps (Definition 5.1.2) [10, 26]. If the PWMs would be

much longer than the sequences themselves, the probability of having columns with only gaps would become much larger. Therefore, we would expect that if we have very large PWMs, we would have to remove many columns from the corresponding alignments in order to create valid multiple sequence alignments. In our approach, the maximum number of columns that a PWM can have is twice as large as the length of the longest input sequence. We have chosen this bound, because we think that it provides a good trade-off between the size of the PWMs and the different multiple sequence alignments that they correspond to. This is for the reason that, even for the longest sequence, we have the possibility that we place a gap in the alignment for each character that we put in it and we think we can construct a large enough set of alignments in this manner. Our approach for initializing the population is depicted in Algorithm 8.

**Algorithm 8** Procedure for initializing the population of PWMs

```
1:  procedure PopulationInitialize(seq, size)   ▷ The given set of input
    sequences seq and the size of the population size
2:      Initialize an empty population Pop
3:      Get the length l of the longest sequence in seq
4:      for n = 1 to size do
5:          Randomly choose a value r, with 2 · l ≥ r ≥ l
6:          Initialize an empty PWM of length r
7:          for j = 1 to PWM.length do
8:              probLeft = 1
9:              while probLeft > 0 do
10:                 Let per be a random percentage
11:                 toAdd = probLeft · per
12:                 Randomly select a row i
13:                 PWM[i][j] = PWM[i][j] + toAdd
14:                 probLeft = probLeft − toAdd
15:             end while
16:         end for
17:         Add PWM to Pop
18:     end for
19: end procedure
```

The first step in the initialization of the population is to construct an empty population. This step requires $O(1)$ time. Thereafter, we need to determine the length $l$ of the longest sequence, in order to be able to determine the minimum length of the PWMs. For this we loop over all $s$ sequences, and for each sequence, we determine its length. Depending on whether the length of the sequence is stored as a property alongside it or not, this operation takes either $O(s)$ time or $O(s \cdot l)$ time (in this case we have to loop over each character in a sequence to determine its length). Finally, we fill the population with the required amount of PWMs. To construct a PWM, we have to determine the number of columns that the PWM should have and then we need to instantiate all of the columns. Instantiating a single column takes $O(1)$ time. We create between $l$ and $2 \cdot l$ columns, thus we create a PWM in $O(l)$ time. For a population of size $n$, we thus fill the population with PWMs in $O(n \cdot l)$ time. We thus initialize the

population in $O(s \cdot l + n \cdot l)$ time. Note that the initialization procedure that has been discussed in Section 5.1, also describes that we still have to update the PWMs. Because this updating of the PWMs is also used in other places than the initialization of the population, we will not analyze the complexity of this procedure in this section. Instead, we will analyze the complexity of this procedure in Section 6.2.

## 6.2 PWM Updating

The procedure for updating a PWM that we use, is the same as the procedure depicted in Algorithm 5 [33]. The first step in updating a PWM is to construct the corresponding alignment, for which we use a slightly altered version of the procedure that has been described in Section 5.5. The difference between our version and the version that has been described in Section 5.5, is that we fill a smaller part of both the score matrix and of the choices matrix. In the traceback procedure we start at the bottom right corner of the choices matrix and during each step we can move either diagonally upwards or vertically to the left. This means that entries right of the diagonal that starts at the bottom right entry of both the score matrix and of the choices matrix are not used in constructing the alignment, and for that reason we do not fill these entries. An example of this situation is depicted in Figure 6.1. In this figure the dark red entries do no need to be filled because they lie to the right of the diagonal that starts at the bottom right corner, which is depicted by the arrows.

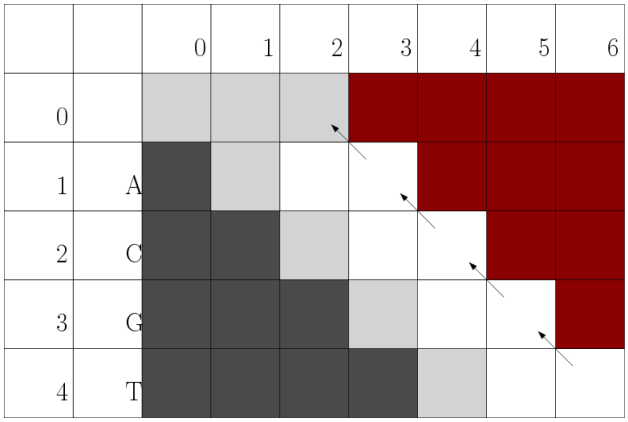



Figure 6.1: An example of a score/choices matrix with the diagonal of the bottom right entry

To construct the alignment that corresponds to the PWM that we want to update, we need to align each of the sequences against the PWM. To align a sequence against the PWM, we first determine the entries that belong to the diagonal that starts at the bottom right corner. For this we need to determine the column index of the last column, which can be done in $O(1)$ time. Then

we need to iterate backwards over all of the rows and determine the entry of that row that lies on the diagonal that starts at the bottom right corner of the matrix. The rows of the matrices correspond to the sequence that is being aligned, thus we can determine the diagonal entries in $O(l)$ time. Thereafter, we need to fill the necessary entries of the score matrix and of the choices matrix. Filling an entry of the score matrix takes $O(l)$ time, because in case of a gap we need to determine whether or not the gap is internal by iterating over the characters in the sequence. Filling an entry of the choices matrix takes $O(1)$ time, because we only need to indicate whether we reached the corresponding state by a vertical or by a diagonal move. The score matrix and the choices matrix have $O(l)$ rows and $O(l)$ columns (for the reason that the number of columns of the PWM is a constant times the length of the longest sequence). Even though we do not need to fill all of the entries of both the score matrix and of the choices matrix, this is not reflected in the complexity analysis for the reason that it only reduces the number of entries that we have to fill by a constant factor. Filling the entries of the score matrix and of the choices matrix thus requires $O(l^3)$ time. Finally, we need to perform the traceback procedure to construct the alignment of the sequence. This is done by following the entries in the choices matrix from the bottom right corner to the top left corner. For each entry, we determine its state and then we either add the next character to the alignment or we add a gap to the alignment. Depending on whether we can get the character at a specified index in the sequence in $O(1)$ or in $O(l)$ time, the traceback procedure requires either $O(l)$ time or $O(l^2)$ time. Aligning a single sequence against a PWM thus requires $O(l^3)$ time. To construct the alignment of all of the sequences for the PWM, we thus require $O(s \cdot l^3)$ time.

The second step in updating a PWM is to remove the columns that contain only gaps from the corresponding alignment. For this purpose we first need to determine which of the columns only contains gaps. We can do this by looping over all columns and then check for each column if all of the sequences contain a gap in that column, which takes $O(s \cdot l)$ time. Thereafter, we need to remove the columns that only contain a gap. We need to remove $O(l)$ columns, thus depending on whether we can remove a character at a specific index from a sequence in $O(1)$ time or in $O(l)$ time (in case a copy of the sequence is made that contains one character less, namely it does not contain the character at the specified index), we can remove the required columns in either $O(s \cdot l)$ time or in $O(s \cdot l^2)$ time. The second step in updating the PWMs thus requires either $O(s \cdot l)$ time or $O(s \cdot l^2)$ time.

The third step in updating a PWM is to construct a new PWM from the multiple sequence alignment that has been constructed in the previous two steps. To do this, we need to construct a column for the PWM for each column of the multiple sequence alignment. To construct a single column, we need to determine, for each character from the alphabet, the probability with which that character occurs in the specified column of the multiple sequence alignment. We do this by first counting the number of times that each character from the alphabet occurs in the column, which is done by looping over all sequences, and then we divide the number of occurrences of each character by the total number of sequences. Depending on whether we can get the character at a specified index in a sequence in $O(1)$ time or in $O(l)$ time, we can construct a column in

either $O(s + a)$ time or in $O((s \cdot l) + a)$ time, where $a$ denotes the number of characters in the alphabet. To create the new PWM, we need to construct $O(l)$ new columns. We can thus construct a new PWM in either $O((s + a) \cdot l)$ time or in $O(((s \cdot l) + a) \cdot l)$ time.

The fourth and final step in updating a PWM is to replace the old PWM by the new PWM. This only requires the updating of a single reference, and thus only takes $O(1)$ time. Now let us determine the computational complexity of the entire procedure for updating a PWM. Let us first look at the worst case scenario, i.e. at the scenario in which we cannot get the character at a specified index in a sequence in $O(1)$ time and in which we cannot remove a character at a specific index from a sequence in $O(1)$ time. In this situation, the procedure for updating a PWM has the following computational complexity:

$$\begin{aligned} O(s \cdot l^3 + s \cdot l^2 + ((s \cdot l) + a) \cdot l + 1) &= O(s \cdot l^3 + s \cdot l^2 + s \cdot l^2 + a \cdot l + 1) \\ &= O(s \cdot l^3 + a \cdot l) \end{aligned}$$

Let us now look at the best case scenario, i.e. at the scenario in which we can get the character at a specified index in a sequence in $O(1)$ time and in which we can remove a character at a specific index from a sequence in $O(1)$ time. In this situation, the procedure for updating a PWM has the following computational complexity:

$$\begin{aligned} O(s \cdot l^3 + s \cdot l + (s + a) \cdot l + 1) &= O(s \cdot l^3 + s \cdot l + s \cdot l + a \cdot l + 1) \\ &= O(s \cdot l^3 + a \cdot l) \end{aligned}$$

Thus, no matter the computational complexity of getting the character at a specified index in a sequence and no matter the computational complexity of removing a character at a specified position in a sequence, we require $O(s \cdot l^3 + a \cdot l)$ time to update a PWM. We can even simplify this formula a bit by observing that the size of the alphabet can never be larger than the total number of characters in all of the sequences, i.e. $a \leq s \cdot l$. This follows from the fact that the alphabet is extracted from the input sequences themselves. It thus follows that we require $O(s \cdot l^3)$ time to update a PWM.

## 6.3 Fitness Evaluation

The approach that we use for calculating the fitness of a PWM, is the same as the approach that has been described in Section 5.2 [33]. Because we do not store the corresponding alignment with the PWM, we first need to construct the alignment. As described in Section 6.2, the alignment corresponding to a PWM can be constructed in $O(s \cdot l^3)$ time. After having constructed the alignment, we need to determine the score of this alignment. To determine the score of the alignment, we take the sum of the sequence scores of all the sequences in the alignment. The score of a single sequence can be computed by determining the character score of all of the characters in the sequence. Determining the character score of a single character requires $O(l)$ time, for the reason that we have to iterate over all characters in a sequence when we need to determine

whether or not a gap is internal. This means that we can compute the score of a single sequence in $O(l^2)$ time, and that we can compute the score of the entire alignment in $O(s \cdot l^2)$ time. To evaluate the fitness of a PWM, we thus require $O(s \cdot l^3)$ time.

## 6.4 Selection

We use two different selection operators, one for the algorithm of Botta and Negro [9] and one for both the univariate GOMEA algorithm and for the Linkage Tree Genetic Algorithm. As mentioned in Chapter 5, we suspect that Botta and Negro use roulette wheel selection. For both variants of the GOMEA algorithm, we use tournament selection. We will analyze the computational complexity of the roulette wheel selection operator in Section 6.4.1, and we will analyze the computational complexity of tournament selection in Section 6.4.2.

### 6.4.1 Roulette Wheel Selection

In roulette wheel selection we assign each solution in the population a piece of the interval $[0, 1]$, based on the fitness of that solution [43, 50]. Because we store the fitness of a PWM as a property alongside it, we can assign the intervals in $O(n)$ time. Thereafter, we select the same number of solutions as there are solutions in the population. Selecting a single solution can be done by randomly selecting a value in the interval $[0, 1]$ and then adding the solution that was assigned the corresponding part of the interval $[0, 1]$, in which the chosen value lies, to the set of selected solutions. To find the interval in which the corresponding solution lies, we need to loop over all intervals. This means that selecting a single solution requires $O(n)$ time. Roulette wheel selection thus takes $O(n^2)$ time. We can however reduce the complexity of roulette wheel selection by storing the intervals, and their corresponding PWMs, in a data structure such as a red-black tree [15]. In which case, we can reduce the computational complexity of roulette wheel selection from $O(n^2)$ time to $O(n \cdot \log(n))$ time.

### 6.4.2 Tournament Selection

In tournament selection we hold multiple tournaments, in which we randomly pick a certain number of solutions and select the fittest one [50]. In our implementation, we also made sure that we do not select a single solution multiple times in the same tournament. In each tournament we pick $k$ solutions and compare their fitness to determine which solution needs to be selected. For the reason that we store the fitness of a PWM as a property alongside it, this operation takes $O(k)$ time. We select as many solutions as that there are solutions in the population, thus tournament selection requires $O(n \cdot k)$ time. In our implementation we have that $k = 2$, i.e. $k$ is a constant, which means that

the computational complexity of tournament selection is in fact $O(n)$ time.

## 6.5 Reproduction

For the algorithm of Botta and Negro [9] we perform reproduction with the help of the variable one-point crossover operator. We will analyze the complexity of the variable one-point crossover operator in Section 6.5.1. For both variants of the GOMEA algorithm we perform reproduction with the help of the GOM operator [6, 16, 30, 46]. To use the GOM operator, we first need to learn a linkage model from the selected solutions. We will therefore start by analyzing the complexity of learning the linkage model in Section 6.5.2, and thereafter we will analyze the computational complexity of the GOM operator in Section 6.5.3.

### 6.5.1 Variable One-Point Crossover

The variable one-point crossover operator is used to reproduce two PWMs by randomly selecting two crossover points and swapping the columns that appear after the crossover points between the PWMs [33]. Furthermore, it is possible that an extra step has to be taken to ensure that both PWMs at least have the minimum required number of columns. To determine the crossover point for a PWM, we randomly need to pick a value between zero and the length of the PWM. Determining the length of a PWM can be done in $O(1)$ time, for the reason that we can store the length of the PWM as a property that gets updated each time the PWM is updated. After the crossover points have been chosen, the columns need to be swapped. This operation requires $O(l)$ time per PWM. Finally, we have to determine if the PWMs have enough columns. If they do not have enough columns, then we need to add the required number of extra columns to the PWMs. This operation can again be executed in $O(l)$ time. In total, the variable one-point crossover operator needs to be used $O(n)$ times (we perform crossover on a fraction of the population, as specified by the reproduction probability, and each call reproduces two PWMs). The variable one-point crossover operator thus requires $O(n \cdot l)$ time per generation.

### 6.5.2 Model Learning

For the univariate model we do not need to perform a linkage learning step, instead we need to create singleton FOS subsets for each column in the longest PWM in the set of selected solutions. [30, 46] The first step is thus to determine the length of the longest PWM. This is done by looping over all selected PWMs and determining the length of each PWM. This step requires $O(n)$ time. Thereafter, we need to instantiate $O(l)$ singleton FOS subsets, which requires $O(l)$ time. The univariate model is thus constructed in $O(n + l)$ time.

To construct the linkage tree model, we use the nearest neighbor chain clustering method [16]. In our implementation, the mutual information matrix is a similarity matrix instead of a distance matrix. This means that we do not need to negate the mutual information values. We do not perform the nearest neighbor chain clustering method directly on the selected PWMs, but instead we perform the nearest neighbor chain clustering method on the alignments corresponding to the selected PWMs. This is done for the reason that the real solutions that we are interested in are the multiple sequence alignments. The PWMs are just a means to describe the contents of the multiple sequence alignments in a compact and more easily manipulatable manner. For this reason, we think that it is more logical to determine the relations between the columns in the alignments, than to determine the relations between the columns in the PWMs. We can construct all alignments in $O(n \cdot s \cdot l^3)$ time, which results in $n \cdot s$ sequences. The nearest neighbor chain clustering method will then take $O(n \cdot s \cdot l^2)$ time. The entire linkage tree is thus constructed in $O(n \cdot s \cdot l^3)$ time.

### 6.5.3 Gene-Pool Optimal Mixing

Both variants of the GOMEA algorithm use the Gene-Pool Optimal Mixing procedure as the reproduction operator [6, 16, 30, 46]. The first step in the GOM procedure is to make two copies of the solution that needs to be reproduced. To copy a PWM, we need to copy all columns. To copy a single column, we need to copy the contents of each row. Because of the fact that the number of rows of a PWM is equal to the number of characters in the alphabet, we can copy a column in $O(a)$ time. This means that a column can be copied in $O(s \cdot l)$ time, because of the simplification described in Section 6.2. Copying a PWM thus takes $O(s \cdot l^2)$ time.

The second step in the GOM procedure is to shuffle the order of the FOS subsets in the linkage tree model. The FOS subsets are shuffled using the Fisher-Yates Shuffle [1]. The linkage tree contains $O(l)$ FOS subsets. Therefore, we can shuffle the FOS subsets in the linkage tree model in $O(l)$ time.

Thereafter, we need to iterate over all FOS subsets. Each iteration we first choose a random donor from the population and we copy the required columns from the donor solution to the offspring solution, which requires $O(l)$ time. Then we need to determine if the offspring solution has changed by checking if it is equal to the backup solution. To check if two PWMs are equal, we need to check if all of their columns are equal. To check if two columns are equal, we need to check if all of their rows are equal. This equality check thus takes $O(s \cdot l^2)$ time. If the offspring solution has changed, we need to compute its fitness. To ensure that the offspring PWM corresponds to a valid multiple sequence alignment, we first update it before we compute its fitness. We have already established in Section 6.2 that a PWM can be updated in $O(s \cdot l^3)$ time and we have already established in Section 6.3 that we can compute the fitness of a PWM in $O(s \cdot l^3)$ time. If the fitness of the offspring solution has not decreased, we need to copy the offspring solution to the backup solution. Otherwise, we need to copy the backup solution to the offspring solution. Each iteration of a single FOS subset

thus takes $O(s \cdot l^3)$ time. Iterating over all FOS subsets therefore takes $O(s \cdot l^4)$ time.

After the normal GOM phase has ended, the Forced Improvement phase is entered [6, 7, 16, 30]. The Forced Improvement phase differs slightly from the normal GOM phase, but not in such a way that the complexity of the Forced Improvement phase is different than the complexity of the normal GOM phase. The Forced Improvement phase thus also requires $O(s \cdot l^4)$ time. This means that a single PWM can be reproduced in $O(s \cdot l^4)$ time, using the GOM procedure. Each generation we thus require $O(n \cdot s \cdot l^4)$ time in order to reproduce all of the PWMs in the population.

## 6.6 Mutation

For the mutation procedure, the mutation operators that have been described in Section 5.4 are used [33]. Besides these two mutation operators, we have also added two additional mutation operators. These additional mutation operators are only used in both variants of the GOMEA algorithm. The first mutation operator that we have added, is to randomly remove a column from the PWM, but only in case that the PWM does not become shorter than the minimum required length. This mutation operator is depicted in Algorithm 9. The second mutation operator that we have added is to add a new column at a random position in the PWM, which is depicted in Algorithm 10. We have added these mutation operators for the reason that in the algorithm proposed by Botta and Negro [9], the size of the PWMs changes during the crossover procedure. During the GOM procedure, the length of the PWMs does not change however. We have thus added these mutation operators to investigate whether or not the performance of both variants of the GOMEA algorithm is affected by being able to change the size of the PWMs.

For the mutation operator in which we randomly swap two columns, we first need to select two columns randomly. This takes $O(1)$ time. Thereafter, we need to swap both of the columns in $O(1)$ time. We can thus randomly swap two columns in $O(1)$ time. To update the probabilities of two rows in a single column, we need to select two rows, determine a random percentage and add or subtract the required probability from the selected rows. This can also be done in $O(1)$ time. For randomly deleting or adding a column, we require $O(l)$ time (for the reason that we need to move columns to the left or right). Thus, when we do not use the additional mutation operators, we can perform a single mutation in $O(1)$ time. Otherwise, we perform a single mutation in $O(l)$ time. Because of the fact that we need to mutate $O(n)$ solutions during each generation, the mutation phase requires either $O(n)$ time or $O(n \cdot l)$ time.

**Algorithm 9** The mutation operator for deleting a column from the PWM

1: **procedure** DeleteMutation($PWM$) ▷ The matrix $PWM$ to be mutated
2: If $PWM.length \leq$ minimum number of columns, then return
3: Randomly choose a column $j$ from $PWM$
4: Delete column $j$ from $PWM$
5: **end procedure**

**Algorithm 10** The mutation operator for adding a column to the PWM

1: **procedure** AdditionMutation($PWM$) ▷ The matrix $PWM$ to be mutated
2: Randomly choose a position $j$ in $PWM$
3: Add a new column to $PWM$ at position $j$
4: **end procedure**

## 6.7 Replacement

The algorithm proposed by Botta and Negro [9] uses a steady-state replacement strategy, of which we will analyze the complexity in Section 6.7.1. The GOMEA algorithm uses a generational replacement scheme [6], which we will analyze in Section 6.7.2.

### 6.7.1 Steady-State Replacement

In steady-state replacement, each generation, a part of the population is replaced by the offspring solutions [45]. In most cases, this is done by replacing the worst solutions in the population by the best solutions in the set of offspring solutions. To determine the worst and the best solutions, we need to sort both the population and the set of offspring solutions, based on the fitness of the solutions. The sorting procedure requires $O(n \cdot \log(n))$ time. Thereafter, we need to iterate over the population and over the set of offspring solutions to select the solutions that will survive. This can be done in $O(n)$ time. The steady-state replacement procedure thus takes $O(n \cdot \log(n))$ time.

### 6.7.2 Generational Replacement

In the generational replacement strategy, each generation, the entire population is replaced [45]. It is possible to apply an elitist strategy, in order to ensure that some of the solutions from the population will survive. We however do not use the elitist strategy. We can replace the population by simply updating its reference to point to the set of offspring solutions. This means that the generational replacement scheme requires $O(1)$ time.

**Algorithm 11** The genetic algorithm for evolving PWMs

```
1: procedure EvolvePWMs
2:     Initialize population of PWMs
3:     Update the PWMs
4:     Evaluate the fitness of the PWMs in the population
5:     Determine the fittest solution in the population
6:     while Termination criterion not satisfied do
7:         Select PWMs for reproduction
8:         Perform reproduction
9:         Mutate solutions
10:        Update the offspring PWMs
11:        Evaluate the fitness of the offspring PWMs
12:        Perform replacement
13:        Update the fittest solution
14:    end while
15: end procedure
```

## 6.8 Algorithm Complexity

Before we will analyze the overall computational complexity of all three algorithms, let us start by providing a brief recap of the computational complexity of the different parts of the algorithms. For initializing the population of PWMs, we require $O(s \cdot l + n \cdot l)$ time, where $s$ denotes the number of input sequences, $l$ denotes the length of the longest input sequence and $n$ denotes the size of the population. A single PWM can be updated in $O(s \cdot l^3)$ time and we can also compute the fitness of a single PWM in $O(s \cdot l^3)$ time. Roulette wheel selection takes $O(n \cdot \log(n))$ time, while tournament selection requires $O(n)$ time. The variable-one-point crossover operator requires $O(n \cdot l)$ time per generation. The univariate FOS model can be learned in $O(n + l)$ time, while the linkage tree FOS model can be learned in $O(n \cdot s \cdot l^3)$ time. The Gene-Pool Optimal Mixing procedure takes $O(n \cdot s \cdot l^4)$ time per generation. The mutation phase takes either $O(n)$ time or $O(n \cdot l)$ time, depending on whether or not we also include the operators for randomly deleting and adding columns. The steady-state replacement scheme requires $O(n \cdot \log(n))$ time and the generational replacement scheme requires $O(1)$ time.

In Algorithm 1 [49], we already provided a basic overview of the structure of genetic algorithms. We now present an updated overview of the structure of genetic algorithms in Algorithm 11, which is specifically tailored to the genetic algorithms for evolving PWMs. One step in Algorithm 11, which we have not depicted in Algorithm 1, is that we keep track of the fittest solutions during the run of the algorithm. To update the fittest solution, we need to iterate over all solutions in the population and determine which solution has the highest fitness. Because the fitness of a PWM is stored as a property alongside it, we can update the fittest solution in $O(n)$ time.

The setup, i.e. creating and updating the initial population and computing

the fitness of the solutions and determining the fittest solution, is the same for the algorithm proposed by Botta and Negro [9] and for both variants of the GOMEA algorithm. This setup has the following computational complexity:

$$\begin{aligned} &O(s \cdot l + n \cdot l + n \cdot s \cdot l^3 + n \cdot s \cdot l^3 + n) \\ &\qquad = O(s \cdot l + n \cdot l + 2 \cdot (n \cdot s \cdot l^3) + n) \\ &\qquad = O(s \cdot l + n \cdot l + n \cdot s \cdot l^3 + n) \\ &\qquad = O(n \cdot s \cdot l^3) \end{aligned}$$

To evolve the population of PWMs, all three algorithms evolve the PWMs until a certain stopping criterion has been met. Our stopping criterion is that there either has not been a change in the fitness of the solutions for ten generations or that 500 generations have been evaluated [9, 33]. We can evaluate whether or not the stopping criterion has been met in $O(1)$ time. This stopping criterion is arbitrary in the sense that it could take far more or far less than 500 generations for the population to converge. In our analysis, we will therefore look at the number of generations that are required for the population to converge and not at a maximum number of generations for which the algorithm is allowed to evolve the solutions in the population. The number of generations that is necessary for the population to converge depends on the length of the sequences, i.e. the number of generations that is necessary for the population to converge is a function of the length of the sequences. This is for the reason that smaller sequences result in smaller PWMs, which in turn indicates that we expect that there will be less variation between the PWMs. Because the GOMEA algorithm uses more information about the structure of the solutions in order to direct the evolution in the direction of the fitter solutions, we expect that the GOMEA algorithm reaches convergence faster than the algorithm that has been proposed by Botta and Negro [9] does. We will therefore denote the number of generations that is required for the population to converge in the algorithm that has been proposed by Botta and Negro [9] by $g(l)$ and we will denote the number of generations that is required for the population to converge in the GOMEA algorithm by $g'(l)$.

Let us now start by analyzing the computational complexity of evolving the population of PWMs for the algorithm proposed by Botta and Negro [9]. For this algorithm, the computational complexity of evolving the population is:

$$\begin{aligned} &O(g(l) \cdot (2 \cdot (n \cdot \log(n)) + n \cdot l + 2 \cdot n + 2 \cdot (n \cdot s \cdot l^3))) \\ &\qquad = O(g(l) \cdot (n \cdot \log(n) + n \cdot l + n + n \cdot s \cdot l^3)) \\ &\qquad = O(g(l) \cdot (n \cdot \log(n) + n \cdot s \cdot l^3)) \\ &\qquad = O(g(l) \cdot n \cdot \log(n) + g(l) \cdot n \cdot s \cdot l^3) \end{aligned}$$

Let us then analyze the computational complexity of evolving the PWMs for the univariate GOMEA algorithm, which has the following computational complexity:

$$\begin{aligned} &O(g'(l) \cdot (3 \cdot n + n \cdot s \cdot l^4 + n \cdot l + 2 \cdot (n \cdot s \cdot l^3) + l + 1)) \\ &\qquad = O(g'(l) \cdot (n + n \cdot s \cdot l^4 + n \cdot l + n \cdot s \cdot l^3 + l + 1)) \\ &\qquad = O(g'(l) \cdot n \cdot s \cdot l^4) \end{aligned}$$

Finally, let us analyze the computational complexity of the evolution phase of the Linkage Tree Genetic Algorithm. The computational complexity of evolving the population of PWMs for the Linkage Tree Genetic Algorithm is:

$$\begin{aligned} O(g'(l) \cdot (2 \cdot n + 3 \cdot (n \cdot s \cdot l^3) &+ n \cdot s \cdot l^4 + n \cdot l + 1)) \\ &= O(g'(l) \cdot (n + n \cdot s \cdot l^3 + n \cdot s \cdot l^4 + n \cdot l + 1)) \\ &= O(g'(l) \cdot n \cdot s \cdot l^4) \end{aligned}$$

It thus follows that the algorithm proposed by Botta and Negro [9] runs in $O(g(l) \cdot n \cdot \log(n) + g(l) \cdot n \cdot s \cdot l^3)$ time and that both variants of the GOMEA algorithm run in $O(g'(l) \cdot n \cdot s \cdot l^4)$ time.

# Chapter 7

# Parameter Settings: Experimental Study

For the algorithm of Botta and Negro [9], we use the same parameter settings as they do, which we have already described in Chapter 5. They have however not very clearly described which substitution matrix they use, but we suspect that they have used the BLOSUM62 matrix [31] and therefore we will also use this substitution matrix in all of our experiments. We do not know if the same parameter settings will also work well for both the univariate GOMEA algorithm and for the Linkage Tree Genetic Algorithm. Therefore, we will test several parameter settings in this chapter. In this way, we hope to find a parameter setting that performs well for both algorithms for the experiments in which we compare the performance of all three of the algorithms.

We will start with a description of our experimental setup in Section 7.1. Thereafter, we will present the results of the experiments in Section 7.2. Finally, we will discuss our results in Section 7.3.

## 7.1 Experimental Setup

All of our experiments will be run on the BAliBASE multiple alignment benchmark [48], of which we will use version 4. The BAliBASE benchmark is divided into multiple reference sets that are all designed to represent a set of real world multiple alignment problems, each having different properties. Ideally, we would like to run our tests on all reference sets, because then we could find a parameter setting that works well in general. Unfortunately, the computational cost of performing any single experiment is quite high and both the time frame that is available to us for performing all of the experiments is limited and the computational resources that are available to us for performing all of the experiments are limited. While Botta and Negro [9] were able to perform many experiments

due to the fact that they had access to a distributed computing network, we can only run a limited number of experiments due to the fact that we only have access to a single laptop. Therefore, we will only run our experiments on alignment BB11001 from reference set RV11. We are well aware that this is a very limited set of alignments to perform our experiments on and we are also well aware that it is likely that we will find a parameter setting that does not work well in general, but unfortunately this is the best that we can do under the given time constraints and under the given limited availability of computational resources.

To determine how well the different parameter settings perform, we will use the program for scoring a test multiple sequence alignment with respect to a given reference multiple sequence alignment, which is provided as a part of the BAliBASE benchmark [48]. This program computes two different scores, namely the sum-of-pairs score and a column score. Both of these scores are measured on a scale from zero to one. The sum-of-pairs score is used to measure the percentage of the character pairs that has been aligned correctly, which in turn helps us determine to what extent the algorithm is successful in aligning a part/all of the sequences correctly. The column score on the other hand measures the percentage of correctly aligned columns, which means that the column score determines the ability of the algorithm to align all of the sequences correctly. The column score is thus more strict in the sense that for any column the alignment can either be correct or not, while in the sum-of-pairs score columns can be partially correct due to the fact that at least some sequences have been aligned correctly. Let us illustrate this by a brief example. Suppose that we have $s$ sequences and that we have aligned $s-1$ sequences correctly and that for the remaining sequence we have aligned all columns incorrectly. In this scenario, the alignment would still be quite good according to the sum-of-pairs score due to the fact that most sequences have been aligned correctly. The column score however would determine that this alignment is completely incorrect. This small example should illustrate that the sum-of-pairs score is more informative in the sense that we can at least give a non-zero score to partially correct alignments. For the reason that the sum-of-pairs score is better able to measure partial correctness of alignments and for the reason that Botta and Negro [9] used the sum-of-pairs score to determine the quality of the alignments that they generated, we will only look at the sum-of-pairs score when determining the quality of our alignments We will thus use the sum-of-pairs score to compare the three algorithms and we will also use the sum-of-pairs score to test the quality of our parameter settings.

We will test different settings of the population size and different settings of the probabilities of applying the different mutation operators. Because of the limited available time and the limited available computational resources, we will only test a few settings for each parameter. Furthermore, we will only test the parameter settings on the univariate GOMEA algorithm. The univariate GOMEA algorithm is simpler than the Linkage Tree Genetic Algorithm, i.e. the Linkage Tree Genetic Algorithm is able to learn more complex linkage models, and therefore a parameter setting that gives good results for the univariate GOMEA algorithm should also provide equally good or better results for the Linkage Tree Genetic Algorithm. We are however aware that, in this way,

we might reject parameter settings that do not perform well for the univariate GOMEA algorithm, while they would perform well for the Linkage Tree Genetic Algorithm.

For the population size we will test four different settings, namely a population of 50 PWMs, a population of 100 PWMs, a population of 150 PWMs and a population of 200 PWMs. For the probabilities of applying the different mutation operators, we will test two different settings:

1. $M1$, which is the setting in which we only use the mutation operators for swapping two columns and for updating the probabilities of two rows in a single column. Both mutation operators have the same probability, i.e. both mutation operators have a probability of being applied of 0.5. This is the same mutation scheme as used by Botta and Negro [9]. This setting can be used to help determine whether or not it is beneficial to be able to change the size of the PWMs by deleting or adding a column, because it provides a baseline to compare the other mutation setting to.
2. $M2$, which is the setting in which all four mutation operators are applied. All of the mutation operators have a probability of being applied of 0.25.

We will repeat each experiment thirty times to be able to decide whether or not there is a statistically significant difference between the settings. To determine whether or not there is a statistically significant difference, we will compare each of the settings using an independent samples t-test with a significance level of $\alpha = 0.05$, in which we do not assume that the variance of both groups is equal. In each of these t-tests, our null hypothesis is that there is no performance difference between both parameter settings and our alternative hypothesis is that there is a significant performance difference between both parameter settings.

All experiments will be run on a laptop that has an Intel Core i5-6200U 2.30 GHz dual core processor and 8 GB of RAM memory, which runs on Windows 10 Home version 1709. The application itself is a multi-threaded $C\#$ application, of which we run a 64-bit release build.

## 7.2 Results

The results of the experiments for the different parameter settings on alignment BB11001 from reference set RV11 for the univariate GOMEA algorithm can be found in Table 7.1. In this table we have recorded both the mean of the sum-of-pairs scores and the standard deviation of the sum-of-pairs scores, which is given in between brackets. From this table it follows that the mean of the sum-of-pairs scores increases as the size of the population increases. Furthermore, we observe that the mean of the sum-of-pairs scores is higher when we use all four mutation operators than when we use only the two mutation operators that are used by Botta and Negro [9].

| | *50* | *100* | *150* | *200* |
|---|---|---|---|---|
| *M1* | 0.295 (0.114) | 0.399 (0.182) | 0.406 (0.170) | 0.460 (0.175) |
| *M2* | 0.560 (0.263) | 0.574 (0.219) | 0.655 (0.223) | 0.720 (0.161) |

Table 7.1: The sum-of-pairs scores of the experiments of the different parameter settings on alignment BB11001 from reference set RV11 for the univariate GOMEA algorithm

The results of the independent samples t-tests of the experiments for the different parameter settings on alignment BB11001 from reference set RV11 for the univariate GOMEA algorithm can be found in Table 7.2. In this table the bold entries denote that there is a statistically significant difference in the sum-of-pairs scores of the corresponding parameter settings. We observe that, for all population sizes, there is a statistically significant difference in the sum-of-pairs scores between the experiments in which we used only two mutation operators and the experiments in which we used all four mutation operators. Furthermore, we observe that within a group of experiments that uses the same mutation operators, there is in most cases no statistically significant difference between the sum-of-pairs scores of the different population sizes.

| | *50, M1* | *100, M1* | *150, M1* | *200, M1* | *50, M2* | *100, M2* | *150, M2* | *200, M2* |
|---|---|---|---|---|---|---|---|---|
| *50, M1* | - | **0.010** | **0.004** | **0.000** | **0.000** | **0.000** | **0.000** | **0.000** |
| *100, M1* | **0.010** | - | 0.888 | 0.195 | **0.008** | **0.001** | **0.000** | **0.000** |
| *150, M1* | **0.004** | 0.888 | - | 0.229 | **0.010** | **0.002** | **0.000** | **0.000** |
| *200, M1* | **0.000** | 0.195 | 0.229 | - | 0.089 | **0.030** | **0.000** | **0.000** |
| *50, M2* | **0.000** | **0.008** | **0.010** | 0.089 | - | 0.825 | 0.137 | **0.006** |
| *100, M2* | **0.000** | **0.001** | **0.002** | **0.030** | 0.825 | - | 0.161 | **0.005** |
| *150, M2* | **0.000** | **0.000** | **0.000** | **0.000** | 0.137 | 0.161 | - | 0.198 |
| *200, M2* | **0.000** | **0.000** | **0.000** | **0.000** | **0.006** | **0.005** | 0.198 | - |

Table 7.2: Results of the independent samples t-tests of the experiments for the different parameter settings on alignment BB11001 from reference set RV11 for the univariate GOMEA algorithm

For each parameter setting, we have also recorded the mean and the standard deviation of the number of generations that were necessary to perform a single run of the univariate GOMEA algorithm on alignment BB11001 from reference set RV11 in Table 7.3. From this table it follows that a lot more generations are required to perform a single run of the algorithm when all four mutation operators are used than when we use only the mutation operators that are used by Botta and Negro [9].

## 7.3 Discussion

We conclude that we obtain better multiple sequence alignments when we do not only use the column swap mutation operator and the mutation operator in

| | *50* | *100* | *150* | *200* |
|---|---|---|---|---|
| *M1* | 24.50 (7.394) | 30.80 (14.480) | 37.50 (23.832) | 54.93 (59.830) |
| *M2* | 455.63 (135.471) | 488.83 (61.162) | 467.43 (92.206) | 479.53 (77.703) |

Table 7.3: Generations required to perform a single run for each of the different parameter settings for the univariate GOMEA algorithm on alignment BB11001 from reference set RV11

which we update two rows in a single column, but when we also use the column deletion and the column addition mutation operators. When all four mutation operators are used, the PWMs are evolved during many more generations. We suspect that this is for the reason that the diversity of the population is not lost as quickly in this case.

Each time we create a new PWM through reproduction, be it with the help of the one-point crossover operator used by Botta and Negro [9] or be it with the help of the Gene-Pool Optimal Mixing operator used by the GOMEA algorithm, we also update the PWM. As we have already described in Section 5.1, we update a PWM by constructing its corresponding alignment, removing columns that only contain gaps from this alignment and finally by replacing the PWM by a new PWM that is constructed from this alignment. When we update a PWM, we reduce its diversity. This is for the reason that, for each column of the PWM, each row can only have a limited number of different states. If we have $s$ sequences, then each row within a column is in one of $s + 1$ states. For example if we have four sequences, then each row within a column is in one of the following states: 0, 0.25, 0.5, 0.75 or 1. For each column, the number of different configurations of that column is thus limited.

The order of the non-gap characters in the sequences in the multiple sequence alignment is fixed. This follows from the fact that a multiple sequence alignment is created by inserting gaps at certain positions in the sequences. This means that the only way in which we can alter a multiple sequence alignment, is by either adding or deleting new gaps or by changing the positions in which the gaps occur. To change the positions in which the gaps occur, we need to change the contents of the columns of the PWM. To either add gaps to the multiple sequence alignment or to delete gaps from the multiple sequence alignment, we either have to add new columns to the PWM or we need to remove columns from the PWM. During the Gene-Pool Optimal Mixing procedure we only change the contents of the columns of a PWM, which means that we can only change the positions in which the gaps in the multiple sequence alignment corresponding to the PWM occur. Because the number of possible states of a column is limited, because not each change of the contents of the columns of the PWMs actually changes the corresponding multiple sequence alignment and because not each change in the corresponding multiple sequence alignment increases the fitness of the PWM, we expect that the population diverges fairly quickly. This can either be due to the fact that we only have a few PWMs that are good donors or because we regularly enter the Forced Improvement phase. Using only the Gene-Pool Optimal Mixing procedure to create new PWMs, we thus expect to lose the diversity in the population fairly quick.

The variable one-point crossover operator used by Botta and Negro [9] does change the size of the PWMs. This change can be quite drastic in the sense that we can cross over large parts of the parent solutions. The mutation operators are thereafter used to change the contents of some of the columns of the PWMs. In this way, the corresponding multiple sequence alignments can be altered much more than when we use the GOMEA algorithm with the same mutation operators. We thus suspect that using the mutation operators for adding columns to the PWMs or for deleting columns from the PWMs are beneficial for the GOMEA algorithm for the reason that they ensure that we can change the corresponding multiple sequence alignments in more ways, which should ensure that the population will converge less quickly.

Looking at the results we see that when all four mutation operators are used, a population of 200 PWMs perform significantly better than a population of either 50 PWMs or a population of 100 PWMs. There is however no significant performance difference between a population of 150 PWMs and a population of 200 PWMs. Because it is better to use all four mutation operators and because less work is required to evolve a smaller population, we will use all four mutation operators for our experiments and each experiment will be run on a population of 150 PWMs.

# Chapter 8

# Algorithm Comparison: Experimental Study

In this chapter we aim to compare both the performance of the algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm and we also aim to compare how well these algorithms scale when the problem becomes larger. We will start by comparing the performance of the algorithms in Section 8.1. We will thereafter end by comparing the scalability of the algorithms in Section 8.2.

## 8.1 Performance Comparison

In this Section we will compare how well the algorithm of Botta and Negro [9], the univariate GOMEA algorithm and the Linkage Tree Genetic Algorithm perform. We will start by describing our experimental setup in Section 8.1.1. Thereafter, we will present the results of the experiments in Section 8.1.2. Finally, we will present our conclusions about the performance of the algorithms in Section 8.1.3.

### 8.1.1 Experimental Setup

We will compare the performance of all three algorithms based on how well they perform on version 4 of the BAliBASE multiple alignment benchmark [48]. We will compare the performance of the three algorithms on the basis of the sum-of-pairs score of the multiple sequence alignments that they generate. As already mentioned in Section 7.1, the computational cost of performing any single experiment is quite high and both the time frame that is available to us for performing all of the experiments is limited and the computational resources

that are available to us for performing all of the experiments are limited. Therefore, we will only be able to compare the performance of all three algorithms on a single test alignment. Ideally, we would like to test the performance of the algorithms on at least one alignment from each of the different reference sets. This is for the reason that we are then able to determine if the algorithms perform well in general or that they are only suited to align sequence sets that have specific properties. We have however only tested our parameter settings on reference set RV11. We do not know whether or not these parameter settings generalize well, i.e. we do not know whether or not these parameter settings will actually generate good multiple sequence alignments on reference sets other than reference set RV11. If we would thus test the performance of both variants of the GOMEA algorithm on reference sets other than reference set RV11, we are not able to draw any meaningful conclusions in case that both variants of the GOMEA algorithm perform poorly on these reference sets. For this reason, we will only test the three algorithms on a single alignment from reference set RV11, namely on alignment BB11001.

For the algorithm of Botta and Negro [9], we will use the same parameter settings that they have used. See Chapter 5 for a description of these parameter settings. For both variants of the GOMEA algorithm we will use the parameter settings that have been selected in Section 7.3, i.e. we will use a population of 150 PWMs and we will also use the mutation operators for randomly adding columns to a PWM or for randomly deleting columns from a PWM. In case of both variants of the GOMEA algorithm, we will apply mutation with a probability of 0.05. If mutation needs to be applied, then one of the four different mutation operators is chosen randomly. The BLOSUM62 substitution matrix [31] will be used in all three algorithms.

For the reason that we want to determine whether or not there is a statistically significant difference between the algorithms, we will repeat each of the experiments twenty times. We will compare the algorithms using independent samples t-tests with a significance level of $\alpha = 0.05$. For all of these t-tests, we do not assume that the variance of the algorithms is equal. For these t-tests, our null hypothesis is that there is no difference in performance between the algorithms and our alternative hypothesis is that there is a difference in performance between the algorithms.

Both the laptop that we use and the application that we use, are the same as described in Section 7.1. This means that we run the experiments on a laptop that has an Intel Core i5-6200U 2.30 GHz dual core processor and 8 GB of RAM memory, which runs on Windows 10 Home version 1709, and that the we run a 64-bit release build of the application, which is a multi-threaded *C#* application.

### 8.1.2 Results

The results of the experiments for comparing the performance of the evolutionary algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm on alignment BB11001 from reference

set RV11 can be found in Table 8.1. In this table we have recorded both the mean of the sum-of-pairs scores (the higher the mean of the sum-of-pairs scores, the better the algorithm performed) and the standard deviation of the sum-of-pairs scores, which is given in between brackets. From this table it follows that the mean of the sum-of-pairs scores is higher for the algorithm of Botta and Negro [9], than that the mean of the sum-of-pairs scores for the univariate GOMEA algorithm is. It also follows that the mean of the sum-of-pairs scores of the Linkage Tree Genetic Algorithm is the highest of all three algorithms. Furthermore, we observe that the mean of the sum-of-pairs score of the univariate GOMEA algorithm fluctuates much more than the mean of the sum-of-pairs scores of the other two algorithms does, i.e. the standard deviation of the sum-of-pairs scores is much higher for the univariate GOMEA algorithm.

| | *BB11001* |
|---|---|
| *EABotta* | 0.695 (0.069) |
| *Univariate GOMEA* | 0.652 (0.217) |
| *LTGA* | 0.776 (0.052) |

Table 8.1: The sum-of-pairs scores of the different algorithms on alignment BB11001 from reference set RV11

The results of the independent samples t-tests of the experiments for comparing the performance of the evolutionary algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm on alignment BB11001 from reference set RV11 can be found in Table 8.2. In this table the bold entries denote that there is a statistically significant difference between the sum-of-pairs scores of the different algorithms. We observe that there is a statistically significant different between the sum-of-pairs scores of the Linkage Tree Genetic algorithm and between the sum-of-pairs scores of the other algorithms. Furthermore, we observe that there is no statistically significant difference between the sum-of-pairs scores of the algorithm of Botta and Negro [9] and between the sum-of-pairs scores of the univariate GOMEA algorithm

| | *EABotta* | *Univariate GOMEA* | *LTGA* |
|---|---|---|---|
| *EABotta* | - | 0.398 | **0.000** |
| *Univariate GOMEA* | 0.398 | - | **0.021** |
| *LTGA* | **0.000** | **0.021** | - |

Table 8.2: Results of the independent samples t-tests of the experiments for the different algorithms on alignment BB11001 from reference set RV11

### 8.1.3 Conclusion

The univariate GOMEA algorithm has a much higher standard deviation for the sum-of-pairs scores than the other two algorithms have. This means that the performance of the univariate GOMEA algorithm can fluctuate a lot between two or more runs. By inspecting the results more closely, we have observed that there is one run of the univariate GOMEA algorithm that produced a multiple sequence alignment that is of a higher quality than any of the multiple sequence alignments that have been generated in any of the runs of the other algorithms. There is no statistically significant performance difference between the algorithm of Botta and Negro [9] and between the univariate GOMEA algorithm. We should however note that we can better use the algorithm of Botta and Negro [9] in case that we perform a limited number of runs of the algorithm. This is for the reason that the performance of this algorithm is much more constant, while we have more chance to find bad alignments when we execute only a few runs of the univariate GOMEA algorithm.

Furthermore, we have found that there is a statistically significant performance difference between the Linkage Tree Genetic Algorithm and the other algorithms. The mean of the sum-of-pairs scores is higher for the Linkage Tree Genetic Algorithm than that it is for the other two algorithms. We thus conclude that the Linkage Tree Genetic Algorithm is the best of the three algorithms when we only take the performance into account.

## 8.2 Scalability Comparison

In this section we aim to analyze how well the univariate GOMEA algorithm and the Linkage Tree Genetic Algorithm scale in comparison to the algorithm of Botta and Negro [9]. We have already analyzed the scalability of the algorithms from a theoretical point of view in Chapter 6 by analyzing the computational complexity of the algorithms. The current comparison on the other hand is much more practically oriented. To analyze the scalability of the algorithms, we have performed several experiments. We will describe the setup of these experiments in Section 8.2.1 and we will present the results of these experiments in Section 8.2.2. We will finish by analyzing the scalability of all three algorithms in Section 8.2.3.

### 8.2.1 Experimental Setup

If we look at the algorithm of Botta and Negro [9] at one hand and at both variants of the GOMEA algorithm on the other hand, we can see that there are quite some differences between these algorithms. They have different selection operators, different reproduction operators and slightly different mutation operators. The biggest difference between these algorithms is however the way in which new solutions are generated, i.e. the biggest difference is between the

different crossover operators that are used by the algorithms. The big difference is that we do use linkage information in the crossover operator of the Linkage Tree Genetic Algorithm, while we do not use any linkage information in the crossover operator of the algorithm of Botta and Negro [9] and in the crossover operator of the univariate GOMEA algorithm. Therefore, we have chosen to limit our experiments for the scalability of the algorithms to the scalability of the crossover operators.

Furthermore, we have that the variable one-point crossover operator and the Gene-Pool Optimal Mixing operator are very different. This means that there are hardly any metrics that we could measure on both operators in order to determine their scalability. We also observe that the variable one-point crossover operator is far less complex than that the Gene-Pool Optimal Mixing operator is, which means that we can measure far less metrics for the variable one-point crossover operator than that we can measure for the Gene-Pool Optimal Mixing operator. For these reasons we will limit our experiments to the Gene-Pool Optimal Mixing operator. This means that we will only perform experiments for both variants of the GOMEA algorithm.

To determine the scalability of the Gene-Pool Optimal Mixing Operator we will perform a single run of only a single generation for both variants of the GOMEA algorithm on alignment BB11002 from reference alignment set RV11 of version 4 of the BAliBASE multiple alignment benchmark [48]. We have chosen to run our experiments on alignment BB11002 instead of on alignment BB11001, as we did for all of our earlier experiments, for the reason that this alignment is several times larger. Therefore, we expect that we are better able to observe the differences between these algorithms. We will use the same parameters settings as have been described in Section 8.1.1. We have however made one small change to the initialization of the PWMs. We do not longer initialize PWMs with a random length, but each PWM is initialized with a length that is 1.5 times the length of the longest input sequence. This is done to be able to compare the algorithms better, i.e. to rule out differences due to different sized PWMs. For each experiment we will measure several metrics:

1. We will measure the number of milliseconds that are necessary to construct the multiple sequence alignments from which the FOS model is learned. Because we do not need to construct the multiple sequence alignments to learn the FOS model for the univariate GOMEA algorithm, this is the only metric that we will measure for the Linkage Tree Genetic Algorithm only.

2. We will measure the number of milliseconds that are necessary to construct the FOS model.

3. We will measure the number of times that we replace the columns of one PWM with columns from another PWM during the initial phase of the Gene-Pool Optimal Mixing operator.

4. We will measure the number of times that we need to update a PWM during the initial phase of the Gene-Pool Optimal Mixing operator. Due

to the fact that we only need to update a PWM if we actually changed it, this number can be smaller than the number of times that we replace columns of the PWM with columns from another PWM.

5. We will measure the number of times that the Forced Improvement phase is entered.
6. We will measure the number of times that we replace the columns of one PWM with columns from another PWM during the Forced Improvement phase of the Gene-Pool Optimal Mixing operator.
7. We will measure the number of times that we need to update a PWM during the Forced Improvement phase of the Gene-Pool Optimal Mixing operator.

As described in both Section 7.1 and in Section 8.1.1, we will run our experiments on a laptop with an Intel Core i5-6200U 2.30 GHz dual core processor that has 8 GB of RAM memory. This laptop runs on Windows 10 Home version 1709. Our application is a multi-threaded *C#* application. Of this application, we run a 64-bit release build.

### 8.2.2 Results

The results of the experiments for testing the scalability of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm on alignment BB11002 from reference set RV11 can be found in Table 8.3. From this table it follows that a lot more time is required to learn the FOS model in the Linkage Tree Genetic Algorithm than that is required to learn the FOS model in the univariate GOMEA algorithm. Furthermore, it follows that the number of PWM updates during the initial phase of the Gene-Pool Optimal Mixing operator is a little more that two times as large for the Linkage Tree Genetic Algorithm than that it is for the univariate GOMEA algorithm. Finally, we observe that in the univariate GOMEA algorithm the Forced Improvement phase has been entered several times, while the Forced Improvement phase was not entered at all in the Linkage Tree Genetic Algorithm.

| | *Univariate GOMEA* | *LTGA* |
|---|---|---|
| *MSA construction (milliseconds)* | 0 | 6923 |
| *Linkage model learning (milliseconds)* | 1 | 522534 |
| *Initial column replacements* | 28950 | 57600 |
| *Initial PWM updates* | 26898 | 54258 |
| *Forced Improvement phases executed* | 7 | 0 |
| *Forced Improvement column replacements* | 1351 | 0 |
| *Forced Improvement PWM updates* | 1266 | 0 |

Table 8.3: Metrics for the scalability of the univariate GOMEA algorithm and the Linkage Tree Genetic Algorithm on alignment BB11002 from reference set RV11

### 8.2.3 Analysis

In this section we will analyze and compare the scalability of the algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm. As mentioned in Section 8.2.1, we only look at the scalability of the used crossover operators. Because the crossover operator of the GOMEA algorithm consists of two separate steps, i.e. the model learning phase and the actual reproduction phase, our analysis will be split into two parts. In the first part, see Section 8.2.3.1, we will focus on analyzing the scalability of the model learning phase In the second part, see Section 8.2.3.2, we will analyze the scalability of the actual reproduction phase.

#### 8.2.3.1 Model Learning

From the results obtained, we can observe that hardly any time is required to construct the FOS model in the univariate GOMEA algorithm. To construct the FOS model in the univariate GOMEA algorithm we only need to determine the number of columns of the largest PWM and thereafter we need to instantiate the required number of singleton FOS subsets. To determine the number of columns of the longest PWM, we need to iterate over all PWMs and check if their length is larger than the length of the longest PWM found so far. This requires only one or two operations per PWM, depending on whether or not we need to update the length of the longest PWM found so far. To instantiate a singleton FOS subset, we only need to perform a single operation. This means that the construction of the FOS model in the univariate GOMEA algorithm scales quite well with respect to both the number of PWMs and the length of the PWMs (the length of the input sequences).

On the other hand we have the Linkage Tree Genetic Algorithm. According to the results obtained, we observe that in the Linkage Tree Genetic Algorithm quite a lot of time is required for learning the FOS model. This is due to the fact that in the Linkage Tree Genetic Algorithm we actually need to learn linkage information, which is not necessary in the univariate GOMEA algorithm. The first step in learning the linkage tree FOS model is to construct the multiple sequence alignments that correspond to the PWMs in the population. To construct the corresponding multiple sequence alignment for a single PWM, we need to construct both the score matrix and the choices matrix and thereafter we need to perform a traceback procedure. These steps need to be performed for each of the input sequences separately. Because both the score matrix and the choices matrix are quite large, even for relatively short sequences, this requires a lot of computations to be performed. Thus, quite a lot of computations need to be performed to align even a single sequence against a PWM. This means that this step does not scale very well with respect to the number of input sequences, with respect to the length of the input sequences and with respect to the number of PWMs in the population.

The second and also final step in learning the linkage tree FOS model is to

perform a hierarchical clustering method. In the hierarchical clustering method we need to construct a similarity matrix to determine which singleton FOS subsets need to be clustered. Thereafter, we need to keep updating the similarity matrix to determine which FOS subsets need to be clustered next. The size of the initial similarity matrix is determined by the length of the input sequences. The number of computations that is required to construct a single column of the similarity matrix depends on the number of sequences in the combined set of multiple sequence alignments of all the PWMs in the population. This step also requires a lot of computations to be performed and this step also does not scale well with respect to the number of input sequences, with respect to the length of the input sequences and with respect to the number of PWMs in the population.

We thus conclude that, no matter the number of PWMs in the population and no matter the length of the input sequences, constructing the FOS model in the univariate GOMEA algorithm does hardly add any overhead to the crossover operator. On the other hand we have that the learning of the FOS model in the Linkage Tree Genetic Algorithm adds a lot of overhead to the crossover operator and we have that the learning of the FOS model in the Linkage Tree Genetic Algorithm does not scale well with respect to the number of input sequences, with respect to the length of the input sequences and with respect to the number of PWMs in the population. If we look at the results obtained, we observe that for a relatively small test (relatively few sequences and relatively short sequences) we only require a single millisecond to construct the FOS model in the univariate GOMEA algorithm and we require nearly nine minutes to learn the FOS model in the Linkage Tree Genetic Algorithm. In the algorithm of Botta and Negro [9] we do not need to learn a FOS model at all. If we thus compare all three algorithms on the overhead of constructing the FOS model, we conclude that the univariate GOMEA scales quite well in comparison with the algorithm of Botta and Negro [9] and we conclude that the Linkage Tree Genetic Algorithm scales quite poorly in comparison to the algorithm of Botta and Negro [9].

#### 8.2.3.2 Reproduction

From the results obtained, we can observe that we need to update PWMs more often during the initial phase of the Gene-Pool Optimal Mixing operator for the Linkage Tree Genetic Algorithm than that we need to do for the univariate GOMEA algorithm. Of course this is not surprising due to the fact that the linkage tree contains more FOS subsets than the univariate model does. To be more specific, if the univariate model consists of $s$ FOS subsets, then the linkage tree consists of $(2 \cdot s) - 2$ FOS subsets. We would thus expect that the number of times a PWM is updated in the Linkage Tree Genetic Algorithm is slightly less than two times the number of times that a PWM is updated in the univariate GOMEA algorithm. Instead, we have that the number of times a PWM is updated in the Linkage Tree Genetic Algorithm is slightly more than two times the number of times that a PWM is updated in the univariate GOMEA algorithm. This must be due to the fact that the column swaps in the

Gene-Pool Optimal Mixing operator do more often create a new PWM in the Linkage Tree Genetic Algorithm than that they do in the univariate GOMEA algorithm. In other words, this means that the singleton FOS subsets do more often replace a column by an equal column from another PWM than that the larger FOS subsets do. Furthermore, we have observed that the univariate GOMEA algorithm does more often enter the Forced Improvement phase than that the Linkage Tree Genetic Algorithm does. The number of PWM updates during the Forced Improvement phase is however still many times smaller than the number of PWM updates during the initial phase of the Gene-Pool Optimal Mixing operator, thus this difference is not as significant as the difference in the number of PWM updates during the initial phase of the Gene-Pool Optimal Mixing operator.

Based on the results obtained, we conclude that the number of times a PWM is updated in the Gene-Pool Optimal Mixing operator is roughly twice as large for the Linkage Tree Genetic Algorithm than that it is for the univariate GOMEA algorithm. This means that the Linkage Tree Genetic Algorithm scales quite well in comparison with the univariate GOMEA algorithm, when we compare them on the number of PWM updates performed during the reproduction phase. Unfortunately, updating a PWM is quite an expensive operation. As we can observe from the results, twice as many PWM updates can easily mean tens of thousands of additional PWM updates. It should also again be noted that our test was relatively small, i.e. consisted of relatively few sequences and the sequences were relatively short. This means that twice as many PWM updates could also easily mean hundreds of thousands or millions of additional PWM updates. This means that even updating a PWM twice as many times already results in a significant amount of extra work. All in all, we can thus say that the Linkage Tree Genetic Algorithm does not scale that well when compared to the univariate GOMEA algorithm, even though the amount of work is roughly only twice as large.

If we look at the variable one-point crossover operator that is used by the algorithm of Botta and Negro [9], we can see that it is much simpler than the Gene-Pool Optimal Mixing operator. Because we do not need to update a PWM in the variable one-point crossover operator itself, the variable one-point crossover operator is far less expensive from a computational point of view than that the Gene-Pool Optimal Mixing operator is. If we look at the number of times that we need to update a PWM in the algorithm of Botta and Negro [9], we can see that each PWM needs to be updated at most once during each generation. If we look at the results obtained, we thus observe that we need to perform a PWM update hundreds of times more during a single generation of both the univariate GOMEA algorithm and the Linkage Tree Genetic Algorithm than that we need to do for the algorithm of Botta and Negro [9]. This means that both the univariate GOMEA algorithm and the Linkage Tree Genetic Algorithm scale very poorly when compared to the algorithm of Botta and Negro [9].

# Chapter 9

# Conclusions

The aim of this research has been to investigate whether the approach that was taken by Botta and Negro [9] for finding high quality multiple sequence alignments could be improved by incorporating linkage learning into the evolution of the solutions. For this purpose, we have compared the algorithm of Botta and Negro [9] against both the univariate GOMEA algorithm and against the Linkage Tree Genetic Algorithm. We will start this chapter by summarizing our results in Section 9.1. Thereafter, we will discuss the quality of our research in Section 9.2. Finally, we will provide several directions of future research in Section 9.3.

## 9.1 Summary

To determine whether or not the univariate GOMEA algorithm and the Linkage Tree Genetic Algorithm are able to improve upon the performance of the algorithm of Botta and Negro [9], we started by testing multiple parameter settings. These tests showed us that a better performance is obtained when we incorporate mutation operators for randomly adding a column to a PWM and for randomly deleting a column from a PWM to the mutation scheme of the GOMEA algorithm. We argued that this must be for the reason that these mutation operators ensure that the population diverges less quickly. Based on these tests we chose a population size of 150 PWMs and a mutation scheme that incorporated the mutation operators for randomly adding a column to a PWM and for randomly deleting a column from a PWM.

Thereafter, we compared the performance of the algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm. The results showed that there is no statistically significant difference in performance between the algorithm of Botta and Negro [9] and between the univariate GOMEA algorithm. The results did however show that there is

a statistically significant performance difference between the Linkage Tree Genetic Algorithm and between both the algorithm of Botta and Negro [9] and the univariate GOMEA algorithm. On average, we obtained the highest quality multiple sequence alignments using the Linkage Tree Genetic Algorithm. These tests also showed us that the performance of both the algorithm of Botta and Negro [9] and of the Linkage Tree Genetic Algorithm is much more constant than the performance of the univariate GOMEA algorithm, i.e. the standard deviation of the quality of the multiple sequence alignments is much higher for the univariate GOMEA algorithm than that it is for both the algorithm of Botta and Negro [9] and for the Linkage Tree Genetic Algorithm. Based on these results we thus conclude that, by incorporating linkage learning into the evolution of the solutions, we can improve upon the quality of the multiple sequence alignments that are obtained with the approach of Botta and Negro [9].

Finally, we compared the scalability of the algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm. For this purpose, we started by analyzing the worst-case computational complexity of these algorithms. We showed that the algorithm of Botta and Negro [9] has a worst-case computational complexity of $O(g(l) \cdot n \cdot \log(n) + g(l) \cdot n \cdot s \cdot l^3)$ time and that both variants of the GOMEA algorithm have a worst-case computational complexity of $O(g'(l) \cdot n \cdot s \cdot l^4)$ time, where $l$ denotes the length of the longest input sequence, $s$ denotes the number of input sequences, $n$ denotes the population size and both $g(l)$ and $g'(l)$ are functions that determine the number of generations that are necessary for the algorithms to converge. Thereafter, we provided a more practically oriented comparison of the scalability of the algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm. This comparison showed us that the overhead of learning a linkage tree is quite high and that the number of PWM updates during a single generation is hundreds of times larger for both variants of the GOMEA algorithm than that it is for the algorithm of Botta and Negro [9]. We thus conclude that both variants of the GOMEA algorithm scale quite poorly in comparison with the algorithm of Botta and Negro [9].

We thus conclude that, by incorporating linkage learning into the evolution of the solutions, we can improve upon the quality of the multiple sequence alignments that are obtained with the approach of Botta and Negro [9]. Furthermore, we conclude that both variants of the GOMEA algorithm scale quite poorly in comparison with the algorithm of Botta and Negro [9]. This means that it is beneficial to incorporate linkage information to be able to find multiple sequence alignments of a higher quality, but that this comes with a cost in the sense that it does require much more work to find such a solution.

## 9.2 Discussion

As we have mentioned before, there is quite a high computational cost associated with running any single experiment and both the time frame that was available to us for performing all of the experiments was limited and the computational

resources that were available to us for performing the tests were quite limited. This means that our results do not provide an indefinite proof of whether or not we can improve upon the approach of Botta and Negro [9] by incorporating the use of linkage information for evolving the PWMs. We have tested a very limited set of parameter settings, which means that the univariate GOMEA algorithm could still perform better than the algorithm of Botta and Negro [9] does in case that a different parameter setting is used. Furthermore, we do not know whether or not the Linkage Tree Genetic Algorithm is better than the algorithm of Botta and Negro [9] in general or that it is only better on the single test that we have performed. This research should thus most of all be seen as an initial feasibility study in which we have shown that it seems promising to incorporate linkage information into the approach of Botta and Negro [9].

While we were not able to provide an indefinite proof of whether or not we can improve upon the approach of Botta and Negro [9] by incorporating the use of linkage information for evolving the PWMs, we were able to provide a thorough analysis of the scalability of the algorithms. We provided a detailed description of the computational complexity of all parts of the algorithms. We believe that this description is quite useful in gaining an understanding of how the workload of the algorithms scales with respect to both the parameters of the multiple sequence alignment problem and with respect to the parameters of the algorithms themselves. Furthermore, we believe that the practical scalability analysis provides an insightful comparison of how much PWM updates are necessary in a single generation of any of the three algorithms. We thus believe that the strong point of this research is in the fact that it provides a thorough analysis of the scalability of the algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm for the multiple sequence alignment problem.

## 9.3 Future Work

We have shown that both variants of the GOMEA algorithm scale quite poorly when compared to the algorithm of Botta and Negro [9]. This means that a run of both the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm on even a relatively small test case already takes quite a lot of time. This in turn means that it is often not feasible to use these algorithms in any practical situation, unless we can massively parallelize these algorithms by running them in the cloud for example. One direction of future research could thus be to investigate if and how we could reduce the workload of these algorithms, without the need to parallelize them.

For the Linkage Tree Genetic Algorithm, we have shown that one of the reasons that this algorithm scales so badly is that it takes quite some time to construct the linkage tree. In our implementation, this linkage tree needs to be constructed again for each generation. It would be faster if we would only need to learn the linkage tree once per run of the algorithm. We do however not know if the performance of the algorithm is drastically affected if we only

learn the linkage tree once, instead of learning the linkage tree again each time that we reach the next generation. It could thus be an interesting direction of future research to determine whether or not the performance of the Linkage Tree Genetic Algorithm is drastically affected by learning the linkage tree in advance once.

For determining the parameter setting that we have used for both variants of the GOMEA algorithm, we have only performed a limited amount of tests. Because we have only performed a limited amount of tests, we do not know whether or not our parameter settings are any good. There could possibly be other parameter settings that can drastically improve the performance of both the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm. Furthermore, we have only tested our parameter settings on a single test case. This means that we do not know whether or not these parameter settings generalize well or not. One direction of future research could thus be to test a larger set of parameter settings on a larger number of test cases for both variants of the GOMEA algorithm.

Another direction of future research could be to compare the performance of the algorithm of Botta and Negro [9], of the univariate GOMEA algorithm and of the Linkage Tree Genetic Algorithm on a larger number of tests cases. As we have mentioned before, we do not really know whether or not the incorporation of linkage information is beneficial due to the fact that we have not tested the algorithms on enough test cases. It is thus necessary to perform more tests in order to determine whether or not it is really beneficial to incorporate linkage information into the evolution of the PWMs.

# Bibliography


[1] Abejide Olu Ade-Ibijola. "A Simulated Enhancement of Fisher-Yates Algorithm for Shuffling in Virtual Card Games using Domain-Specific Data Structures". In: *International Journal of Computer Applications* 54.11 (2012).

[2] Enrique Alba and José M Troya. "A survey of parallel distributed genetic algorithms". In: *Complexity* 4.4 (1999), pp. 31–52.

[3] David Beasley, David R Bull, and Ralph Robert Martin. "An overview of genetic algorithms: Part 1, fundamentals". In: *University computing* 15.2 (1993), pp. 56–69.

[4] Tobias Blickle and Lothar Thiele. *A comparison of selection schemes used in genetic algorithms*. 1995.

[5] Peter A. N. Bosman and Dirk Thierens. "On Measures to Build Linkage Trees in LTGA". In: *Parallel Problem Solving from Nature - PPSN XII*. PPSN 2012. Ed. by Carlos A. Coello Coello et al. Lecture Notes in Computer Science 7491. Springer Berlin Heidelberg, 2012, pp. 276–285. ISBN: 978-3-642-32937-1.

[6] Peter A.N. Bosman and Dirk Thierens. "Linkage Neighbors, Optimal Mixing and Forced Improvements in Genetic Algorithms". In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. GECCO '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 585–592. ISBN: 978-1-4503-1177-9. DOI: 10.1145/2330163.2330247. URL: http://doi.acm.org/10.1145/2330163.2330247 (visited on 02/13/2018).

[7] Peter A.N. Bosman and Dirk Thierens. "More Concise and Robust Linkage Learning by Filtering and Combining Linkage Hierarchies". In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. GECCO '13. Amsterdam, The Netherlands: ACM, 2013, pp. 359–366. ISBN: 978-1-4503-1963-8. DOI: 10.1145/2463372.2463420. URL: http://doi.acm.org/10.1145/2463372.2463420 (visited on 02/13/2018).

[8] Peter A.N. Bosman and Dirk Thierens. "The Roles of Local Search, Model Building and Optimal Mixing in Evolutionary Algorithms from a Bbo Perspective". In: *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '11. Dublin, Ireland: ACM, 2011, pp. 663–670. ISBN: 978-1-4503-0690-4. DOI: 10.1145/

2001858.2002065. URL: http://doi.acm.org/10.1145/2001858.2002065 (visited on 02/13/2018).

[9] Marco Botta and Guido Negro. "Multiple sequence alignment with genetic algorithms". In: *Computational Intelligence Methods for Bioinformatics and Biostatistics*. CIBB 2009. Ed. by F. Massuli, L. E. Peterson, and R. Tagliaferri. Lecture Notes in Computer Science, vol 6160. Springer. Berlin and Heidelberg, 2010, pp. 206–214.

[10] Humberto Carrillo and David Lipman. "The Multiple Sequence Alignment Problem in Biology". In: *SIAM Journal on Applied Mathematics* 48.5 (1988), pp. 1073–1082. ISSN: 00361399. URL: http://www.jstor.org/stable/2101469 (visited on 02/13/2018).

[11] K. Chellapilla and G. B. Fogel. "Multiple sequence alignment using evolutionary programming". In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. Vol. 1. IEEE. 1999, pp. 445–452. DOI: 10.1109/CEC.1999.781958.

[12] YP Chen et al. *A survey of linkage learning techniques in genetic and evolutionary algorithms*. Tech. rep. 2007014. Illinois Genetic Algorithms Laboratory, Apr. 2007.

[13] Biswanath Chowdhury and Gautam Garai. "A review on multiple sequence alignment from the perspective of genetic algorithm". In: *Genomics* 109.5 (2017), pp. 419–431. ISSN: 0888-7543. DOI: https://doi.org/10.1016/j.ygeno.2017.06.007. URL: http://www.sciencedirect.com/science/article/pii/S0888754317300551 (visited on 02/13/2018).

[14] James R. Cole et al. "Ribosomal Database Project: data and tools for high throughput rRNA analysis". In: *Nucleic Acids Research* 42.D1 (2014), pp. D633–D642. DOI: 10.1093/nar/gkt1244. eprint: /oup/backfile/content_public/journal/nar/42/d1/10.1093_nar_gkt1244/3/gkt1244.pdf. URL: +%20http://dx.doi.org/10.1093/nar/gkt1244 (visited on 02/13/2018).

[15] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009. Chap. 13, pp. 308–338.

[16] R De Bokx. "Parallelizing the Linkage Tree Genetic Algorithm and Searching for the Optimal Replacement for the Linkage Tree". MA thesis. TU Delft, 2015. URL: http://resolver.tudelft.nl/uuid:6cf6c908-0f5d-4096-b3ad-aa96fd1ff382 (visited on 12/05/2017).

[17] Chuong B Do et al. "ProbCons: Probabilistic consistency-based multiple sequence alignment". In: *Genome Research* 15.2 (2005), pp. 330–340.

[18] Sean R Eddy. "Where did the BLOSUM62 alignment score matrix come from?" In: *Nature biotechnology* 22.8 (2004). Nature Publishing Group, pp. 1035–1036. DOI: 10.1038/nbt0804-1035.

[19] Robert C Edgar. "MUSCLE: a multiple sequence alignment method with reduced time and space complexity". In: *BMC bioinformatics* 5.1 (Aug. 2004), p. 113. ISSN: 1471-2105. DOI: 10.1186/1471-2105-5-113. URL: https://doi.org/10.1186/1471-2105-5-113 (visited on 02/13/2018).

[20] Agoston E Eiben, James E Smith, et al. *Introduction to Evolutionary Computing*. 2nd. Vol. 53. Springer, 2003.

[21] Peter J Fleming and Robin C Purshouse. "Evolutionary algorithms in control systems engineering: a survey". In: *Control engineering practice* 10.11 (2002), pp. 1223–1241. ISSN: 0967-0661. DOI: https://doi.org/10.1016/S0967-0661(02)00081-3. URL: http://www.sciencedirect.com/science/article/pii/S0967066102000813 (visited on 02/13/2018).

[22] Oğuzhan Hasançebi and Fuat Erbatur. "Evaluation of crossover techniques in genetic algorithm based optimum structural design". In: *Computers & Structures* 78.1 (2000), pp. 435–448. ISSN: 0045-7949. DOI: https://doi.org/10.1016/S0045-7949(00)00089-4. URL: http://www.sciencedirect.com/science/article/pii/S0045794900000894%22 (visited on 02/13/2018).

[23] Steven Henikoff and Jorja G Henikoff. "Amino acid substitution matrices from protein blocks". In: *Proceedings of the National Academy of Sciences* 89.22 (1992). National Academy of Sciences, pp. 10915–10919. ISSN: 0027-8424. eprint: http://www.pnas.org/content/89/22/10915.full.pdf. URL: http://www.pnas.org/content/89/22/10915 (visited on 02/13/2018).

[24] H Nabeel Kaghed, S Eman Al–Shamery, and Fanar Emad Khazaal Al-Khuzaie. "Multiple Sequence Alignment based on Developed Genetic Algorithm". In: *Indian Journal of Science and Technology* 9.2 (2016). ISSN: 0974 -5645. URL: http://www.indjst.org/index.php/indjst/article/view/84236 (visited on 02/13/2018).

[25] Kazutaka Katoh et al. "MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform". In: *Nucleic Acids Research* 30.14 (2002), pp. 3059–3066. DOI: 10.1093/nar/gkf436. eprint: /oup/backfile/content_public/journal/nar/30/14/10.1093_nar_gkf436/2/gkf436.pdf. URL: +%20http://dx.doi.org/10.1093/nar/gkf436 (visited on 02/13/2018).

[26] Mehmet Kaya, Abdullah Sarhan, and Reda Alhajj. "Multiple sequence alignment with affine gap by using multi-objective genetic algorithm". In: *Computer methods and programs in biomedicine* 114.1 (2014), pp. 38–49. ISSN: 0169-2607. DOI: https://doi.org/10.1016/j.cmpb.2014.01.013. URL: http://www.sciencedirect.com/science/article/pii/S016926071400025X (visited on 02/13/2018).

[27] Jin Kim, Sakti Pramanik, and Moon Jung Chung. "Multiple sequence alignment using simulated annealing". In: *Bioinformatics* 10.4 (1994), pp. 419–426. DOI: 10.1093/bioinformatics/10.4.419. eprint: /oup/backfile/content_public/journal/bioinformatics/10/4/10.1093/bioinformatics/10.4.419/2/10-4-419.pdf. URL: +%20http://dx.doi.org/10.1093/bioinformatics/10.4.419 (visited on 02/13/2018).

[28] Timo Lassmann and Erik LL Sonnhammer. "Kalign–an accurate and fast multiple sequence alignment algorithm". In: *BMC Bioinformatics* 6.1 (Dec. 2005), p. 298. ISSN: 1471-2105. DOI: 10.1186/1471-2105-6-298. URL: https://doi.org/10.1186/1471-2105-6-298 (visited on 02/13/2018).

[29] Zne-Jung Lee et al. “Genetic algorithm with ant colony optimization (GA-ACO) for multiple sequence alignment”. In: *Applied Soft Computing* 8.1 (2008), pp. 55–78. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2006.10.012. URL: http://www.sciencedirect.com/science/article/pii/S1568494606000822 (visited on 02/13/2018).

[30] Hoang N Luong et al. “Medium-Voltage Distribution Network Expansion Planning with Gene-pool Optimal Mixing Evolutionary Algorithms”. In: *Artificial Evolution*. Ed. by Pierrick Legrand et al. Cham: Springer International Publishing, 2014, pp. 93–105. ISBN: 978-3-319-11683-9.

[31] David W. Mount. “Using BLOSUM in Sequence Alignments”. In: *Cold Spring Harbor Protocols* 6 (2008). DOI: 10.1101/pdb.top39. eprint: http://cshprotocols.cshlp.org/content/2008/6/pdb.top39.full.pdf+html. URL: http://cshprotocols.cshlp.org/content/2008/6/pdb.top39.abstract (visited on 02/13/2018).

[32] Saul B Needleman and Christian D Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of molecular biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4. URL: http://www.sciencedirect.com/science/article/pii/0022283670900574 (visited on 02/13/2018).

[33] Guido Negro. “A Stochastic Evolutionary Algorithm for Multiple Sequence Alignment”. MA thesis. University of Turin, 2009. 82 pp.

[34] Hung Dinh Nguyen et al. “Aligning Multiple Protein Sequences by Parallel Hybrid Genetic Algorithm”. In: *Genome Informatics* 13 (2002), pp. 123–132. DOI: 10.11234/gi1990.13.123.

[35] Cédric Notredame. “Recent progress in multiple sequence alignment: A survey”. In: *Pharmacogenomics* 3 (Feb. 2002), pp. 131–144.

[36] Cédric Notredame and Desmond G Higgins. “SAGA: Sequence Alignment by Genetic Algorithm”. In: *Nucleic Acids Research* 24.8 (1996), pp. 1515–1524. DOI: 10.1093/nar/24.8.1515. eprint: /oup/backfile/content_public/journal/nar/24/8/10.1093_nar_24.8.1515/2/24-8-1515.pdf. URL: +%20http://dx.doi.org/10.1093/nar/24.8.1515 (visited on 02/13/2018).

[37] Cédric Notredame, Desmond G Higgins, and Jaap Heringa. “T-Coffee: A novel method for fast and accurate multiple sequence alignment”. In: *Journal of Molecular Biology* 302.1 (2000), pp. 205–217. ISSN: 0022-2836. DOI: https://doi.org/10.1006/jmbi.2000.4042. URL: http://www.sciencedirect.com/science/article/pii/S0022283600940427 (visited on 02/13/2018).

[38] Francisco M Ortuño et al. “Optimizing multiple sequence alignments using a genetic algorithm based on three objectives: structural information, non-gaps percentage and totally conserved columns”. In: *Bioinformatics* 29.17 (2013), pp. 2112–2121. DOI: 10.1093/bioinformatics/btt360. eprint: /oup/backfile/content_public/journal/bioinformatics/29/17/10.1093_bioinformatics_btt360/2/btt360.pdf. URL: +%20http://dx.doi.org/10.1093/bioinformatics/btt360 (visited on 02/14/2018).

[39] Elmar Pruesse, Jörg Peplies, and Frank Oliver Glöckner. "SINA: Accurate high-throughput multiple sequence alignment of ribosomal RNA genes". In: *Bioinformatics* 28.14 (2012), pp. 1823–1829. DOI: 10.1093/bioinformatics/bts252. eprint: /oup/backfile/content_public/journal/bioinformatics/28/14/10.1093_bioinformatics_bts252/2/bts252.pdf. URL: +%20http://dx.doi.org/10.1093/bioinformatics/bts252 (visited on 02/14/2018).

[40] Christian Quast et al. "The SILVA ribosomal RNA gene database project: improved data processing and web-based tools". In: *Nucleic Acids Research* 41.D1 (2013), pp. D590–D596. DOI: 10.1093/nar/gks1219. eprint: /oup/backfile/content_public/journal/nar/41/d1/10.1093/nar/gks1219/2/gks1219.pdf. URL: +%20http://dx.doi.org/10.1093/nar/gks1219 (visited on 02/14/2018).

[41] C. Reeves. "Genetic algorithms". In: *Handbook of metaheuristics*. International Series in Operations Research & Management Science 57 (2003). Ed. by F. Glover and G. A. Kochenberger. Springer, Boston, MA.

[42] David James Russell. *Multiple Sequence Alignment Methods*. Springer, 2014.

[43] S. N. Sivanandam and S. N. Deepa. *Introduction to genetic algorithms*. Springer, 2007.

[44] Mandavilli Srinivas and Lalit M Patnaik. "Genetic algorithms: A Survey". In: *Computer* 27.6 (June 1994), pp. 17–26. ISSN: 0018-9162. DOI: 10.1109/2.294849. URL: doi.ieeecomputersociety.org/10.1109/2.294849 (visited on 02/14/2018).

[45] Wallace K.s Tang et al. "Genetic algorithms and their applications". In: *Signal Processing Magazine, IEEE* 13 (Dec. 1996), pp. 22–37.

[46] Dirk Thierens and Peter A. N. Bosman. "Optimal Mixing Evolutionary Algorithms". In: *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. GECCO '11. Dublin, Ireland: ACM, 2011, pp. 617–624. ISBN: 978-1-4503-0557-0. DOI: 10.1145/2001576.2001661. URL: http://doi.acm.org/10.1145/2001576.2001661 (visited on 02/14/2018).

[47] Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson. "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice". In: *Nucleic Acids Research* 22.22 (1994), pp. 4673–4680. DOI: 10.1093/nar/22.22.4673. eprint: /oup/backfile/content_public/journal/nar/22/22/10.1093_nar_22.22.4673/2/22-22-4673.pdf. URL: +%20http://dx.doi.org/10.1093/nar/22.22.4673 (visited on 02/14/2018).

[48] Julie D. Thompson et al. "BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark". In: *Proteins: Structure, Function, and Bioinformatics* 61.1 (July 25, 2005), pp. 127–136. DOI: 10.1002/prot.20527. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/prot.20527. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/prot.20527 (visited on 03/29/2018).

[49] M. Tomassini. “Evolutionary algorithms”. In: *Towards Evolvable Hardware*. Lecture Notes in Computer Science 1062 (1996). Ed. by E. Sanchez and M. Tomassini. Springer, Berlin, Heidelberg, pp. 19–47.

[50] Rasmus K. Ursem. “Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization”. PhD thesis. University of Aarhus, Apr. 1, 2003.

[51] Matthew Wall. *GAlib: A C++ library of genetic algorithm components*. 1996. URL: `http://lancet.mit.edu/ga/` (visited on 02/14/2018).

[52] TL Yu and D. E Goldberg. “Toward an Understanding of the Quality and Efficiency of Model Building for Genetic Algorithms”. In: *Genetic and Evolutionary Computation–GECCO 2004*. GECCO 2004. Ed. by K. Deb. Lecture Notes in Computer Science 3103. Springer, Berlin, Heidelberg, 2004.