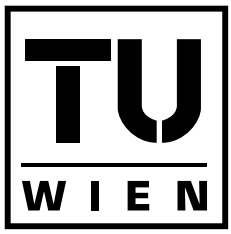


TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

D I P L O M A R B E I T

An Alignment Graph based Evolutionary Algorithm for the Multiple Sequence Alignment Problem

ausgeführt am Institut für
Computergraphik und Algorithmen (E186)
der
Technischen Universität Wien

unter Anleitung von
ao.Univ.Prof. Dr. Günther Raidl
und
Univ.Ass. Dr. Gabriele Koller

durch

Stefan Leopold
9725043
Ludwig-Kössler-Platz 3/6/1
A-1030 Wien

Datum

Unterschrift

Abstract

This diploma thesis presents a new approach for multiple sequence alignment (MSA), which tries to overcome some of the main problems of existing strategies.

At first it gives a formal definition of the MSA problem, elucidates occuring difficulties and describes existing MSA algorithms particularly with regard to evolutionary algorithms.

Then the maximum weight trace problem is introduced, which transforms the problem of computing a multiple alignment with the best score to the problem of finding a trace with maximum weight in a so–called alignment graph. In our approach, this alignment graph contains information of pairwise alignments computed by *CLUSTAL W* as well as global information gained by analysing the consistency between all sequences.

Finally an evolutionary algorithm (EA) is presented, which is used to compute a multiple alignment based on the information of this alignment graph. This EA uses a greedy heuristic to create a set of initial alignments. Then fast variation operators as well as global optimisation strategies like path relinking are applied to further improve these alignments iteratively.

Various tests and comparisons with other MSA programs indicate that this new approach is able to outperform even widely used MSA packages like *CLUSTAL W* on many reference alignments of the BAliBASE library.

Contents

Chapter 1

Multiple Sequence Alignment

1.1 Introduction

Sequence alignment plays an important role in molecular sequence analysis. There are three domains of biological sequences, namely DNA, RNA and proteins. The sequence of a DNA molecule can be modelled as a string over a 4-character alphabet, each character representing one of the four nucleotides that make up DNA. RNA can be modelled in a similar way. The third class of biological macromolecules, the proteins, are chains of amino acids and can be represented as strings over a 20-character alphabet.

The residue alphabet of each domain is shown in Figure 1.1, examples for DNA and protein sequences are given in Figure 1.2.

DNA:
A ... Adenine
C ... Cytosine
G ... Guanine
T ... Thymine

RNA:
A ... Adenine
C ... Cytosine
G ... Guanine
U ... Uracile

Proteins:

A ... Alanine	**C** ... Cysteine	**D** ... Aspartate
E ... Glutamate	**F** ... Phenylalanine	**G** ... Glycine
H ... Histidine	**I** ... Isoleucine	**K** ... Lysine
L ... Leucine	**M** ... Methionine	**N** ... Asparagine
P ... Proline	**Q** ... Glutamine	**R** ... Arginine
S ... Serine	**T** ... Threonine	**V** ... Valine
W ... Tryptophan	**Y** ... Tyrosine	

Figure 1.1: Residue alphabet of DNA, RNA and proteins

DNA sequences:

s_1	C	A	T	G	A	G	A	T	C
s_2	A	C	T	C	A	G	T	A	C

Protein sequences:

s_1	R	S	S	F	G	D	D	H			
s_2	R	S	A	T	C	C	Y	D	F	H	T
s_3	S	S	T	F	C	A	D	D	H		
s_4	R	S	A	T	F	G	A	D	D	T	

Figure 1.2: Examples for DNA and protein sequences

Sequence alignment is a way of placing one sequence above the other in order to make the correspondence between characters or substrings of different sequences clear. Such alignments can be used to

- determine evolutionary relationships that exist among various organisms (phylogenetic analysis),
- identify conserved motifs which are important for the structure and function of a group of related proteins,
- improve secondary and tertiary structure prediction for RNA and proteins.

Naturally it only makes sense, if all involved sequences are defined on the same domain.

Definition 1.1 (Sequence alignment): Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings over the finite alphabet A, each string S_i consisting of l_i ordered characters $s_{i,\cdot}$:

$$S_i = s_{i,1}s_{i,2}\ldots s_{i,l_i}, \quad \forall i = 1, 2, \ldots, n$$

We define a new alphabet $\hat{A} = A \cup \{-\}$ by adding the symbol dash '$-$' to represent spaces. Then a set $\hat{S} = \{\hat{S}_1, \hat{S}_2, \ldots, \hat{S}_n\}$ of strings over the alphabet $\hat{A}$ is called alignment of the set S, if the following properties are fulfilled:

1. All strings in $\hat{S}$ have the same length $\hat{l}$ with

$$\max_{i=1\ldots n}(l_i) \leq \hat{l} \leq \sum_{i=1}^{n} l_i.$$

2. Ignoring dashes, string $\hat{S}_i$ is identical with string S_i , $\forall i = 1, 2, \ldots, n$.

3. $\hat{S}$ has no column that only contains spaces.

Hence an alignment can be interpreted as an array with n rows, where the i-th row contains string $\hat{S}_i$. Depending on the number of sequences, we can differentiate between pairwise sequence alignment ($n = 2$) and multiple sequence alignment ($n \geq 3$).

For examples of a pairwise DNA and a multiple protein sequence alignment based on the sequences given in Figure 1.2 see Figure 1.3.

Pairwise alignment:

s_1	C	A	–	T	G	A	G	–	A	T	C
s_2	–	A	C	T	C	A	G	T	A	–	C

Multiple alignment:

s_1	R	S	S	–	–	F	G	–	D	D	H	–
s_2	R	–	S	A	T	C	C	Y	D	F	H	T
s_3	–	S	S	–	T	F	C	A	D	D	H	–
s_4	R	S	–	A	T	F	G	A	D	D	–	T

Figure 1.3: Examples for a DNA and a protein sequence alignment

In order to solve the sequence alignment problem, we need a measure to determine how good an alignment is. A simple method to distinguish amongst the many possibilites of this arrangement is the distance approach, which provides a measure of how much two strings differ [27].

Costs are assigned to elementary edit operations, like the substitution of a character by another one, or the insertion/deletion of an arbitrary character. The distance between two aligned sequences is then defined as the minimal total cost needed to transform one string into the other one.

For an example of this approach see Figure 1.4.

Substitution costs: 1
Costs for insertion/deletion: 2

Pairwise alignment:

s_1	C	A	–	T	G	A	G	–	A	T	C
s_2	–	A	C	T	C	A	G	T	A	–	C
	2		2		1			2		2	

Total transformation costs: 9

Figure 1.4: Distance between two sequences

Having such a distance function $d(\hat{S}_i, \hat{S}_j)$ for two aligned sequences $\hat{S}_i$ and $\hat{S}_j$, the pairwise alignment problem can be formulated as follows:

Definition 1.2 (Pairwise alignment problem): Let $S = \{S_1, S_2\}$ be a set of two strings over the same alphabet A. Compute the alignment $\hat{S} = \{\hat{S}_1, \hat{S}_2\}$ of S over the alphabet $\hat{A}$ that minimises the distance $d(\hat{S}_1, \hat{S}_2)$.

Scoring a multiple alignment is more complex than its pairwise counterpart, as it can be formalised in different ways. The most commonly used definition is the sum-of-pairs multiple alignment problem, which can be formulated as follows:

Definition 1.3 (Sum-of-pairs multiple alignment problem): Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings over the same alphabet A. Compute the alignment $\hat{S} = \{\hat{S}_1, \hat{S}_2, \ldots, \hat{S}_n\}$ of S over the alphabet $\hat{A}$ that minimises the sum of the distances of all pairs $\hat{S}_i$ and $\hat{S}_j$:

$$\min_{\hat{S}} \left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d(\hat{S}_i, \hat{S}_j) \right)$$

Another scoring method is the similarity approach [27]. Here we are looking for a score that reflects how much two strings are alike. Matching characters are rewarded and gaps are penalised by subtracting a gap penalty. The optimisation task is then defined as finding the alignment that maximises the similarity score.

For an example see Figure 1.5.

Character match: 2
Character mismatch: 0
Gap penalty: -1

Pairwise alignment:

s_1	C	A	–	T	G	A	G	–	A	T	C
s_2	–	A	C	T	C	A	G	T	A	–	C
	-1	2	-1	2	0	2	2	-1	2	-1	2

Similarity score: 8

Figure 1.5: Similarity between two sequences

As shown in [27], distance and similarity are related concepts, thus, under certain conditions distance computations can be reduced to similarity computations and vice versa.

Both, pairwise and sum-of-pairs multiple alignments, can be exactly solved by dynamic programming [18], which converts the original problem to the problem of searching for the shortest path in a weighted directed acyclic graph. However, due to the fact that the converted problem is NP–hard [32], algorithms that guarantee to find the true optimal solution have running times growing exponentially with the size of the problem. Therefore, most of the practical multiple alignment algorithms for larger instances are based on heuristics providing an approximate solution to the problem.

1.2 Multiple Protein Sequence Alignment

When comparing protein sequences, simple scoring schemes as presented in the last section, are not enough. Amino acids have biochemical properties, that influence their relative replaceability in an evolutionary scenario. Thus it is important to use a scoring scheme that reflects this probability as well as possible.

A standard procedure toward this goal is based on the use of similarity matrices [28] such as the point accepted mutation matrix *PAM* or the block substitution matrix *BLOSUM*. These similarity matrices contain scores for all possible matches and mismatches of amino acid symbols based on the frequency of occurrence of these changes in known protein sequence databases. With other words, a score is assigned according to its biological likeliness to every possible substitution or conservation.

Definition 1.4 (Scoring matrix) Let $\mathbf{M}$ be such a scoring matrix between any two characters x and y of the alphabet A, which has the following characteristics:

- $\mathbf{M}(x, y) = \mathbf{M}(y, x), \quad \forall x, y \in A$
- $\mathbf{M}(x, -) = G$, where G is a fixed gap penalty
- $\mathbf{M}(-, -) = 0$

Definition 1.5 (Symbol score) Let S_i and S_j be two strings over the alphabet A and $\hat{S}_i$ and $\hat{S}_j$ the corresponding aligned strings over the alphabet $\hat{A}$. Then the symbol score **SS** for the aligned sequences $\hat{S}_i$ and $\hat{S}_j$ is defined as:

$$\mathbf{SS}(\hat{S}_i, \hat{S}_j) = \sum_{k=1}^{\hat{l}} \mathbf{M}(\hat{s}_{i,k}, \hat{s}_{j,k})$$

Additionally each sequence usually receives a weight proportional to the amount of independent information it contains. Such sequence weights are an attempt to minimise redundant information, based on the relatedness of the sequences. They can be derived from a phylogenetic tree for the sequences.

Definition 1.6 (Weighted symbol score) Let W be such a weight matrix for every pair of aligned sequences. Then the weighted symbol score **WSS** for the aligned sequences $\hat{S}_i$ and $\hat{S}_j$ is defined as:

$$\mathbf{WSS}(\hat{S}_i, \hat{S}_j) = W_{i,j} \cdot \sum_{k=1}^{\hat{l}} \mathbf{M}(\hat{s}_{i,k}, \hat{s}_{j,k})$$

Insertions and deletetions are scored using gap penalties. If we define a gap as continuous block of spaces, then a gap of length l has a penalty score of $l \cdot G$, where $G < 0$ is the fixed gap (extension) penalty. This is called the linear gap penalty function. Adding up this penalty for each gap of an aligned sequence gives the linear gap penalty score (**LGPS**).

There is another type of gap penalty, in which the first space receives a gap opening penalty $O < G < 0$, which is stronger than the penalty for gap extending spaces. A gap of length l has a cost of $O + (l-1) \cdot G$ then. This is called the affine gap penalty function. Adding up this penalty for each gap of an aligned sequence gives the affine gap penalty score (**AGPS**).

As the affine gap penalty score is more appropriate than the linear gap penalty score from the biological point of view, it is used more often.

This finally leads to the most frequently examined sum-of-pairs multiple alignment problem, optimising a weighted sum-of-pairs function with affine gap penalties.

Definition 1.7 Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings over the same alphabet A. Compute the alignment $\hat{S} = \{\hat{S}_1, \hat{S}_2, \ldots, \hat{S}_n\}$ of S over the alphabet $\hat{A}$ that maximises the sum of the weighted symbol score $\mathbf{WSS}(\hat{S}_i, \hat{S}_j)$ and the affine gap penalty score $\mathbf{AGPS}(\hat{S}_i)$ for all aligned sequences $\hat{S}_i$:

$$\max_{\hat{S}} \left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{WSS}(\hat{S}_i, \hat{S}_j) + \sum_{i=1}^{n} \mathbf{AGPS}(\hat{S}_i) \right)$$

1.3 Problems associated with MSA

All alignment methods only make sense if they are assumed to be dealing with a set of homologous sequences, i.e. with sequences sharing a common ancestor. Having inappropriate sequences, MSA algorithms usually produce a meaningless alignment.

Unfortunately accurate multiple alignments are difficult to build. There are two main reasons for this:

- Firstly, it is difficult to evaluate the quality of a multiple alignment. This is a biological problem and lies in the definition of correctness – what should a biologically correct alignment look like?
- Secondly, even if a function is available for the evaluation, it is algorithmically very hard to produce the alignment with the optimal score.

1.3.1 Choice of the Objective Function

The problem is to find a mathematical function – the so called objective function – which is able to measure the biological quality of an alignment. In theory, an objective function should incorporate everything that is known about the sequences including their structure, their function and their evolutionary history. This information is rarely known and so it is usually replaced with sequence similarity. Thus for multiple protein sequence alignment, a

simpler function is often used: weighted sums-of-pairs with affine gap penalties. Under this model, each sequence receives a weight proportional to the amount of independent information it contains and the score of the multiple alignment is equal to the sum of all the weighted pairwise scores. Insertions or deletions are measured using affine gap penalties that penalize a gap once for opening and then proportionally to its length.

Although a sum-of-pairs function is clearly wrong from an evolutionary point of view because it assumes every sequence within the set to be an ancestor of every other sequence, the simplicity of its implementation and its validation as reasonable indicator [30] made it popular within the most widely used MSA packages.

The major problems of the affine gap penalty scheme are the two parameters, the gap opening and the gap extension penalty, whose adequate values can only be set empirically and may vary from one set of sequences to the next. There has been made an extensive review describing the different ways of scoring gaps in a multiple alignment [2], which proposes other, simpler types of gap penalties, e.g. quasi natural gap penalties. However, the affine gap penalty scheme is still most commonly used.

Another approach to evaluate multiple alignments, which we also follow in our method, is the use of a consistency based objective function reflecting the level of consistency between a multiple alignment and a library containing pairwise and/or local alignments of the same sequences. Such methods have the advantage of not being dependent on a specific similarity matrix but rather on any method able to align two sequences at a time.

The first consistency based MSA method has been introduced by Kececioglu [15], who transformed the MSA problem to the maximum weight trace problem as described in chapter 3.1. He presented a branch-and-bound algorithm, whose implementation could optimally align up to 6 tyrosine kinase protein sequences of length 273 to 285 in a few minutes. However, larger examples required excessive space and therefore could not be handled by this exact approach.

Reinert et al. [26] expressed the maximum weight trace problem as an integer linear program. They used a branch-and-cut algorithm and were able to solve several problem instances, for which common dynamic programming algorithms failed. Still, there are restrictions on the number and the length of the sequences.

Fauster [7] applied two fast greedy heuristics to the maximum weight trace problem, which were able to solve even larger instances. The solutions found this way were further improved by local improvement methods and a tabu search algorithm. Comparisons with other programs validated the efficiency of this approach. One of these heuristics is also used in our methods and presented in section 4.3.

Another consistency based method has been described by Notredame et al. [22]. He combined pairwise alignments (as computed by *CLUSTAL W* [28]) and local alignments (computed by Lalign [14]) in a so–called primary library. Then he applied a heuristic – the library extension – analysing the consistency between a pairwise alignment and all other sequences to add some global information. Finally this library was used to derive a phylogenetic tree for a progessive alignment algorithm (see next section), which computed the multiple alignment. A generalisation of the idea of this library extension is also used in our work and presented in chapter 3.4.

Aside, some other scoring models like profiles or hidden Markov models have been developed and tested, a brief overview can be found in [20].

Depending on the used objective function and the underlying scoring model (similarity matrix, gap penalties,...), the optimal multiple alignment is usually different. Therefore, there is also the task of finding a scoring model that produces alignments as close as possible to the true biological alignment besides the task of finding the algorithm to search for the optimal alignment.

1.3.2 Computation of the Alignment

As said before, most of the used practical alignment algorithms are based on heuristics producing quasi-optimal alignments. They are all based on different paradigms and each of them is well suited for certain situations. This wide range of available methods makes it hard to decide which approach is best suited for a given purpose. In general, these approaches can be classified in three main categories – progressive, exact and iterative algorithms – which are briefly described below. For a more detailed review and an extensive bibliography to existing methods see the survey of Notredame [20].

- The majority of multiple sequence alignment heuristics is carried out using a progressive approach. The most widely used algorithm of this category is *CLUSTAL W* [28]. This method gradually builds an align-

ment by first estimating the evolutionary distance between each pair of sequences in the set and afterwards aligning the sequences one by one to the multiple alignment according to decreasing similarity. As this sequence addition can be performed with a pairwise alignment algorithm – *CLUSTAL W* uses the dynamic programming algorithm of Myers and Miller [17] – the approach has the great advantage of speed and simplicity. However, because of the greedy nature of this heuristic, no quality guarantees can be given. This means that if any suboptimal choices are made during the computation, they cannot be corrected later on when more sequences are added to the alignment.

- Another approach is to use extensions of dynamic programming algorithms for simultaneously aligning multiple sequences. They always produce optimal alignments with regard to the used objective function, however, they have drawbacks of exponential time complexity, running time and memory requirement, so such algorithms can only handle a small number of sequences and they are usually limited to the sum-of-pairs multiple alignment problem.

- Iterative algorithms start with a random or (by other methods) precomputed alignment and try to refine it through a series of iterations until no more improvements can be achieved. Such modifications can be made either by deterministic strategies or by stochastic methods like simulated annealing and evolutionary algorithms. Although iterative algorithms cannot provide any guarantees for finding the optimal solution within reasonable time, they have already proved to be robust and much less sensitive to the number of sequences than their deterministic counterparts. The main advantage is the good conceptual separation between optimisation and the objective function, which allows the usage of various objective functions. However, if used alone, iterative algorithms have the drawback of converging relatively slowly and thus requiring more computing time. Especially in combination with other methods, such as an alignment improver, they are nevertheless very promising. As our approach is based on an evolutionary algorithm, a detailed review of existing EAs for the multiple sequence alignment problem is given in the next section.

Chapter 2

Evolutionary Algorithms

2.1 General

Evolutionary algorithms (EAs) are stochastic search methods based on the concept of biological evolution [4]. They are very useful to find (near) optimal solutions for combinatorial optimisation problems which traditional methods fail to solve efficiently. The existing approaches to evolutionary algorithms – including e.g. genetic algorithms [10], evolutionary programming [8] or genetic programming [16] – are all based on the assumption that simulating an evolutionary process in a population of potential solutions evolves better solutions. Biological terms are conveniently used to describe this process:

- The **chromosomes** represent the potential solutions. Every chromosome is typically composed of several **genes**, the solution parameters.
- A set of chromosomes forms a **population**. Successive populations are referred to as **generations**.
- To create new chromosomes (**offsprings**), two kinds of operators are typically used: **Crossovers** are used to exchange genes between two chromosomes, while **Mutations** change one or more genes in a single chromosome.
- Based on the principle of **survival-of-the-fittest**, chromosomes with a good performance (according to an applied fitness function) are more likely to be selected to produce offsprings for the next generation.

The evolutionary process starts with the first generation usually composed of random chromosomes. It is also possible to add good chromosomes from previous computations or solutions provided by other heuristics to this initial

population (**seeding**). This bears the risk of misguiding the optimisation process toward a local optimum. However, starting the evolution in a more useful region of the search space often allows to reach a more appropriate final solution very quickly. Furthermore, if the best chromosome of each generation is always selected for the next generation (**elitism**), seeding also ensures that the solution of the EA will always be at least as good as the solutions provided by previous runs or other heuristics.

During the evolutionary process the chromosomes are exposed to different variation operators. Such crossover and mutation operators can work in a completely random way or perform changes based on special improvement heuristics.

In every population the best chromosomes are selected to form the next generation. This selection is based on a **fitness function** which assigns a fitness value to every chromosome (**evaluation**). Chromosomes with a better fitness value have a higher probability to 'survive' and to produce offsprings for the next generation. This selection is repeated until enough offsprings have been chosen.

The whole evolutionary process is iterated until a termination condition – e.g. terminate after k generations, terminate if the best solution in the population has not changed within the last k generations, terminate when a given fitness value limit is reached – is fulfilled. Figure 2.1 shows the typical structure of an evolutionary algorithm:

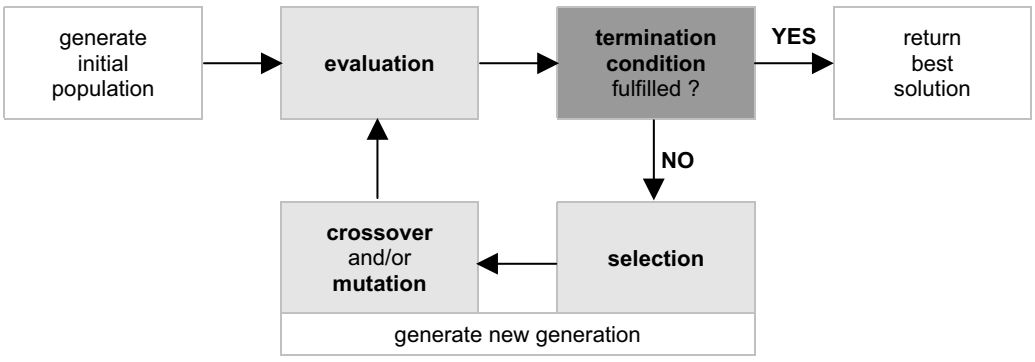


Figure 2.1: Structure of an evolutionary algorithm

The main advantage of evolutionary algorithms over other optimisation methods is that there is no need to provide a particular algorithm to solve a given problem. It only needs a fitness function to evaluate the quality of different solutions and a reasonable coding of the problem, which allows to generate new chromosomes by suitable crossover and mutation operators in an efficient way.

As EAs implicitly are a parallel technique, they can also be implemented effectively on parallel computers to speed up the evolutionary process.

2.2 Evolutionary Algorithms for MSA

There have already been made a lot of attempts to solve the MSA problem by evolutionary algorithms, as they are able to produce good alignments for larger test sets as well as for test sets with low sequence similarity. However, any sequence alignment algorithm must trade off computation speed for alignment accuracy. Applying rather straightforward EAs to the multiple alignment problem therefore often results in fairly slow convergence or – if the EAs is terminated too early – in poor results. Besides, general methods like seeding or parallelisation, problem specific crossover and mutation operators are essential to speed up the evolutionary process.

2.2.1 Existing Evolutionary Algorithms for MSA

In the following the main characteristics of nine existing evolutionary algorithms are presented. These algorithms differ in many features like the chromosome representation of the multiple alignment, the used fitness function and the applied operators.

The first evolutionary algorithm using an adequate set of problem specific crossover and mutation operators was *SAGA* [21]. *SAGA* is a straightforward implementation of a genetic algorithm, which tries to optimise a weighted sum-of-pairs function with natural or quasi-natural affine gap penalties. Like most of the existing algorithms, it uses a two–dimensional array to represent the multiple alignment. 22 different operators are applied to the chromosomes to perform crossovers (One Point Crossover, Uniform Crossover) and mutations (Gap Insertion, different variants of Block Shuffling, Block Searching, Local Optimal or Sub-Optimal Rearrangement). To control the usage of these operators a dynamic scheduling scheme is used, i.e. each operator has a probability of being chosen that is optimised during the run.

Based on *SAGA*, two other evolutionary algorithms have been developed: *PGA* [3], which is a parallel implemention of *SAGA*, and *SAGA-COFFEE* [23], which tries to optimise a consistency based objective function: Let $W_{i,j}$ being the percent identity between two aligned sequences, $C_{i,j}$ being the number of aligned characters, that are shared between the pairwise alignment and a library containing all pairwise alignments, and $L_{i,j}$ being the length of the pairwise alignment. Then the consistency based objective function of *SAGA-COFFEE* is computed as follows (with n being the number of sequences):

$$COFFEE = \left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} W_{i,j} \cdot C_{i,j} \right) / \left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} W_{i,j} \cdot L_{i,j} \right)$$

In contrast to these previously presented algorithms, *GA* [13] uses a number string of the gap positions to represent the multiple alignment. Therefore all operators are designed to work with these gap positions to optimise the alignment based on a sum-of-pairs fitness function. Altogether three crossover (CombineBest, GoodPosCombine, OnePointCombine) and four mutation operators (MergeSpace, MoveSpaceCol, BreakSpaceCol, MoveRowSpace) are used.

One more evolutionary algorithm using the same chromosome representation as *SAGA* is *EP* [6]. This EA tries to optimise the following fitness function:

$$Fitness = SymbolScore - GapScore,$$

with *SymbolScore* being the overall score for the number of matched symbols of all columns and with *GapScore* being the number of all gaps of all columns. For optimisation, *EP* uses one crossover (RecombineMatchedCol) and four mutation operators (RandomShuffle, LocalShuffleOne, GrowMatchedCol, LocallyAlignBlock). Additionally, it provides the possibility to use a heuristic for initialisation by precomputing all pairwise alignments and merging them in a sequential manner (following a random permutation of all sequences).

The genetic algorithm of Cai et al. [5] uses one crossover operator (One Point Crossover) and a heuristic called Local Gap 0-1 Alignment to optimise a sum-of-pairs fitness function. Additonally gaps are inserted at random positions for mutation. Gusfield's Center Star Algorithm [11] is used to compute an approximative alignment for the initial population. Then gaps are inserted at random positions to create diversity.

Another EA is *MSA EA* [31], which uses *CLUSTAL W* to create one chromosome of the initial population. The other chromosomes are randomly initialised. Altogether one crossover (RecombineMatchedColumns) and four mutation operators (LocalShuffle, BlockShuffle, GrowMatchedColumns, CleanUpGapColumns) are used for optimisation.

The genetic algorithm of Zhang et al. [33] focuses on the identification of fully matched columns. One crossover operator (One Point Crossover) and a randomised mutation operator are applied to the chromosomes during the evolutionary process.

PHGA [19] is a parallel genetic algorithm, applied to the converted problem of finding the shortest path in a weighted directed acyclic n–dimensional graph. It uses a number string of edge directions to represent a multiple alignment in the search space. For the correspondence between an aligment and this number string of edge directions see Figure 2.2. A sum-of-pairs function with affine gap penalties is used for optimisation. The designed operators work similarly as already described methods (One Point Crossover, Random Shuffle). The initialisation is performed by a progressive alignment strategy, adding the sequences in random order.

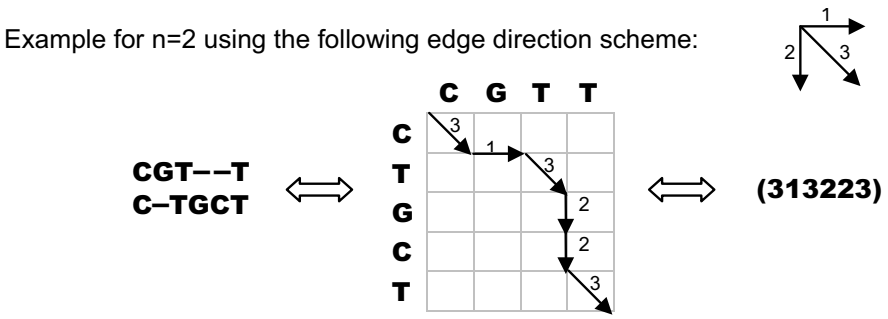


Figure 2.2: Correspondence between an alignment and a number string of gap positions

In the next two sections, an overview of the presented crossover and mutation operators is given. For comparison, these operators are all described with regard to a 2-dimensional array chromosome representation, which allows to refer to the multiple alignment as chromosome. Unless otherwise defined, a block is a number of consecutive characters or gaps in one or more sequences.

The term subalignment refers to a number of consecutive columns of the alignment. Columns containg only gaps are not displayed as they are usually deleted after any modification of the chromosome (e.g. CleanUpGapColumns in *EA*) or applied to other mutation operators (e.g. BreakSpaceCol in *GA*).

2.2.2 Crossover Operators

Crossover operators are responsible for creating one or more new chromosomes (offsprings) by combining the attributes of existing chromosomes (parents).

One Point Crossover used in *SAGA* [21]

One character of the first parent is randomly selected and the chromosome is cut straight at this position. The second parent is tailored so that the right and left parts of each chromosome can be joined together while keeping the original sequence of characters. Any void space that appears at the junction point is filled with gaps, so this method is a local rearrangement mutation as well. An example is given in Figure 2.3.

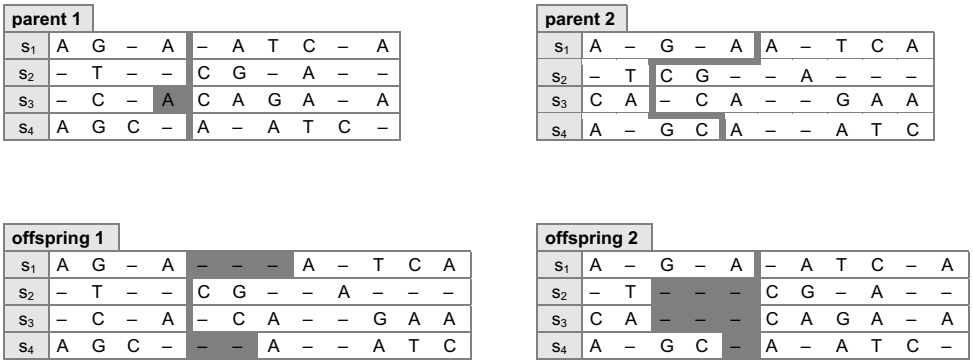


Figure 2.3: One Point Crossover

Similar versions of this operator, where also the first chromosome is not necessarily cut straight, are used in *PHGA* [19], *GA* [13] as well as by the genetic algorithms of Cai et al. [5] and Zhang et al. [33].

Uniform Crossover used in *SAGA* [21]

All consistent columns between two parents are selected. Two columns are consistent, when they contain the same characters by reference to the original sequence (or a gap between the same characters). Blocks between such columns can be directly swapped without further considerations. This swapping can be done in a completely stochastic or in a semi-hill climbing way, if only the combination of swapped blocks with the best fitness is chosen. An example is given in Figure 2.4.

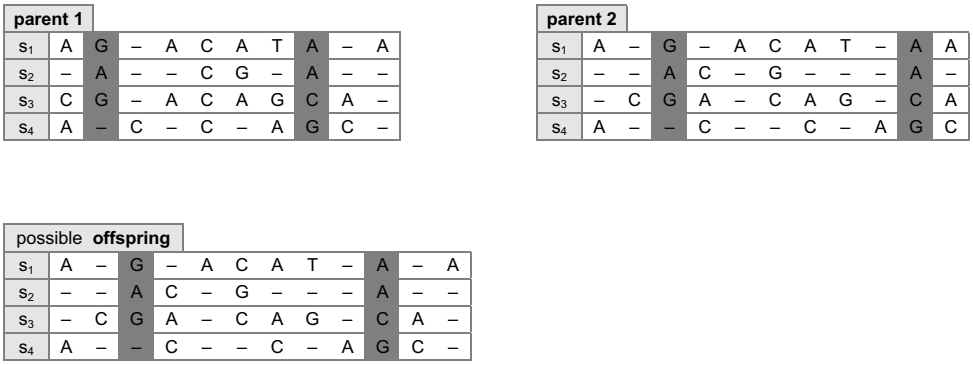


Figure 2.4: Uniform Crossover

RecombineMatchedCol used in *EP* [6], *MSA EA* [31]

All fully matched columns of the first parent that are not present in the second parent, but can be added there without disrupting any existing matched columns, are determined. One of these matched columns is selected with uniform probability and generated in the second chromosome by inserting extra gaps. An example is given in Figure 2.5.

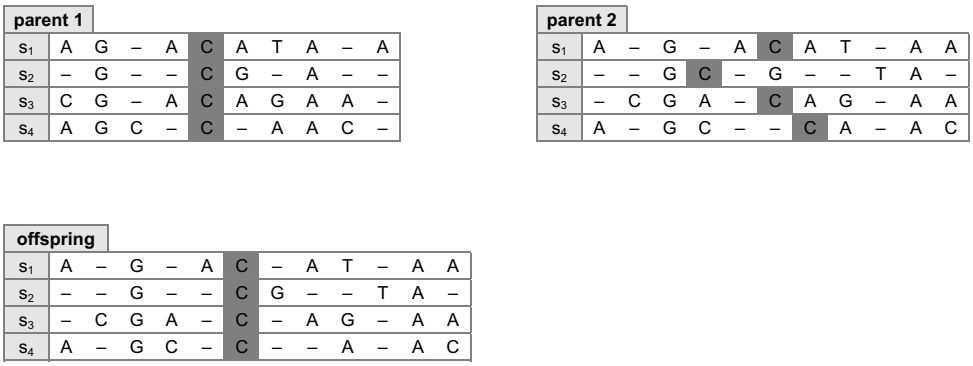


Figure 2.5: RecombineMatchedCol

CombineBest used in *GA* [13]

A random number of consistent columns, which additionally have the same absolute position in both alignments, are selected. Similarly to Uniform Crossover, blocks between such columns can be directly swapped and only the blocks giving the better fitness are used to produce the offspring. An example is given in Figure 2.6.

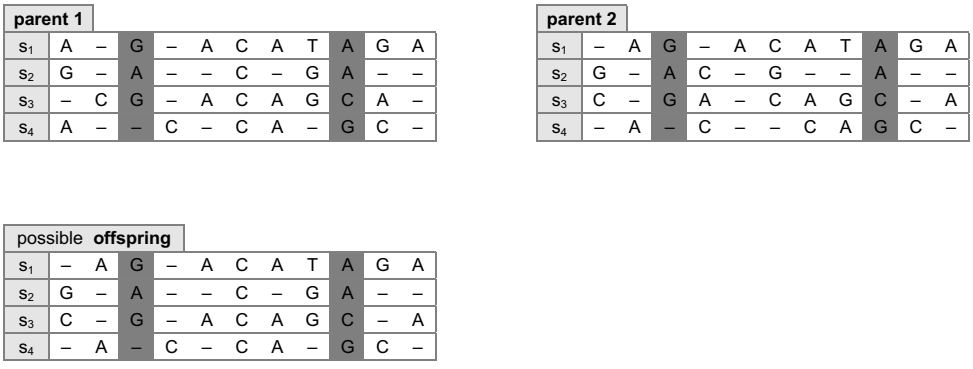


Figure 2.6: CombineBest

GoodPosCombine used in *GA* [13]

Two consistent columns, which additionally have the same absolute position in both alignments, are selected. Between those columns, all gaps appearing

in both chromosomes are chosen and placed at the same position in the offspring. Then gaps appearing in only one chromosome are randomly selected and inserted until enough gaps exist in each row. An example is given in Figure 2.7.

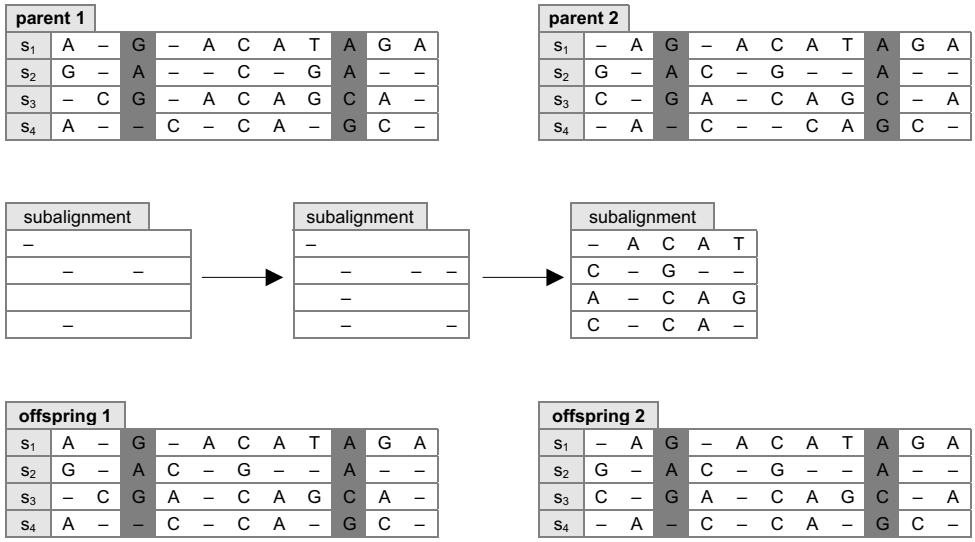


Figure 2.7: GoodPosCombine

2.2.3 Mutation Operators

While crossover operators combine different patterns, there is still a need to generate new ones. A common method is to swap characters and gaps (e.g. *PHGA* [19]) or just to insert a constant number of gaps into each sequence at a random position (e.g. Cai et al. [5]), but there are still some more mutation operators, which try to speed up the evolutionary process using local heuristics.

RandomShuffle, LocalShuffle, BlockShuffle used in *EP* [6], *MSA EA* [31]

One ore more consecutive characters of a sequence are randomly selected and tried to be moved to the left or the right side. If both neighbors are gaps, the direction of the movement is chosen randomly. An example is given in Figure 2.8.

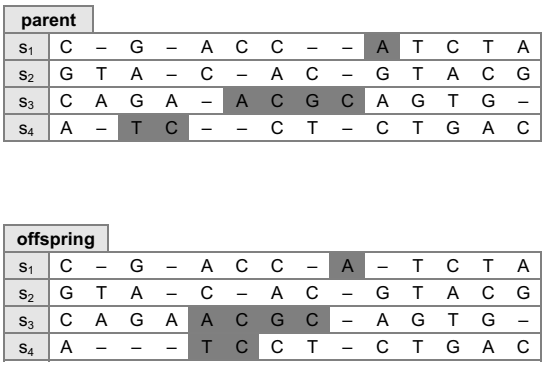


Figure 2.8: RandomShuffle, LocalShuffle, BlockShuffle

GrowMatchedCol used in *EP* [6], *MSA EA* [31]

A fully matched column with no more than one adjacent fully matched column is selected randomly in the alignment. Then – if possible – a new, fully matched column is generated next to it by swapping gaps with characters. An example is given in Figure 2.9.

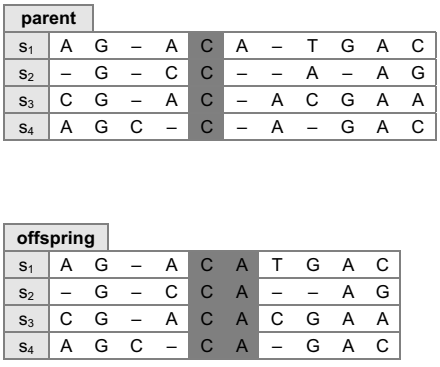


Figure 2.9: GrowMatchedCol

LocalShuffleOne used in *EP* [6]

One of the sequences is selected at random and scanned to find all characters having one or more gaps adjacent to them. One of these characters is selected with uniform probability and shifted to each of these possible posi-

tions. The position of the character giving the best fitness becomes the new destination for it. An example is given in Figure 2.10.

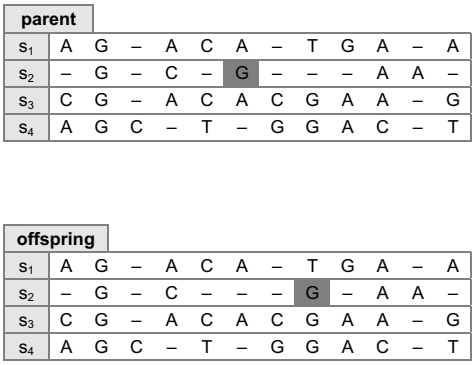


Figure 2.10: LocalShuffleOne

MergeSpace used in *GA* [13]

Two or three consecutive, but not necessarily adjacent gaps in a row are randomly selected, merged together and shifted to a randomly chosen position between two adjacent characters in the sequence. An example is given in Figure 2.11.

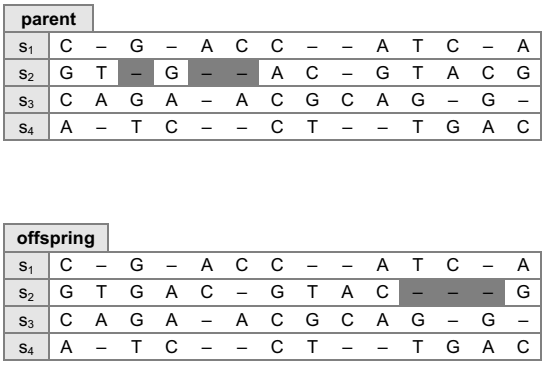


Figure 2.11: MergeSpace

MoveSpaceCol used in *GA* [13]

Two or three adjacent gaps in a row are randomly selected and all other gaps in these columns are determined. Then this block of gaps is shifted to a randomly chosen position not containg any gaps. An example is given in Figure 2.12.

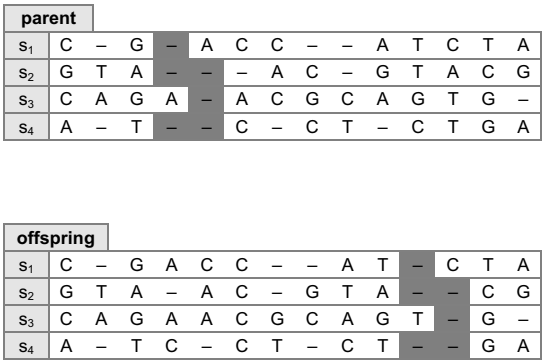


Figure 2.12: MoveSpaceCol

BreakSpaceCol used in *GA* [13]

A column containing only gaps is randomly selected. In each sequence, this gap is moved to a another randomly chosen position. An example is given in Figure 2.13.

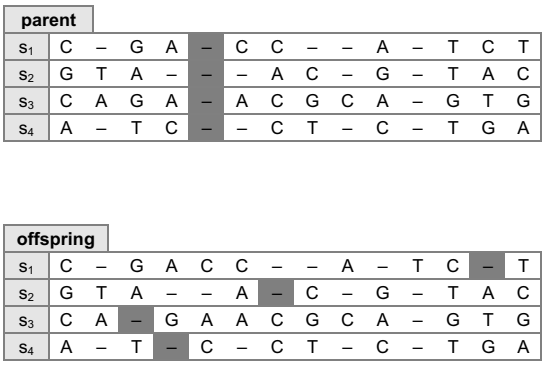


Figure 2.13: BreakSpaceCol

MoveRowSpace used in *GA* [13]

A subalignment with short length is extracted from the full alignment. Then a target sequence is produced by deleting all gaps from a randomly selected sequence in the subalignment. Additionally a template sequence containing the characters with the highest occurrence in each column of the subalignment is generated. These two sequences are aligned by dynamic programming under the condition that gaps are treated as 'base characters' in the template sequence and no new gaps are allowed to be inserted in the template sequence. After the aligning process, the new target sequence replaces the corresponding old one in the subalignment. An example is given in Figure 2.14.

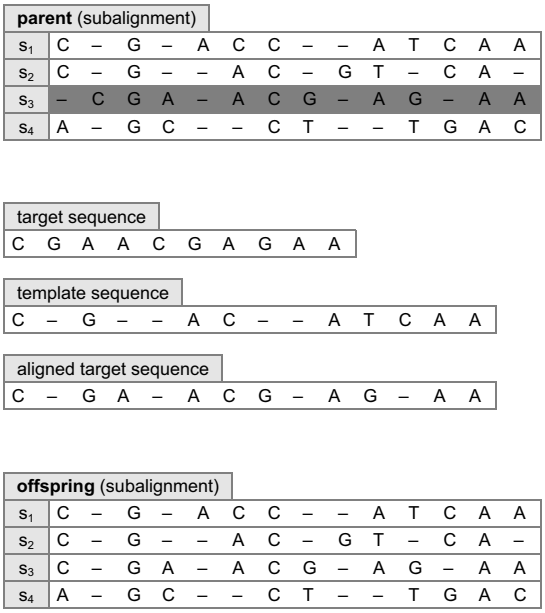


Figure 2.14: MoveRowSpace

Local Gap 0-1 Alignment used by Cai et al. [5]

A subalignment of length two or three is randomly selected and tried to be improved by local Gap 0-1 Alignment, which means that to the left of any subsequence either a gap is inserted or not. The combination giving the best fitness is chosen to be part of the new alignment. An example is given in Figure 2.15.

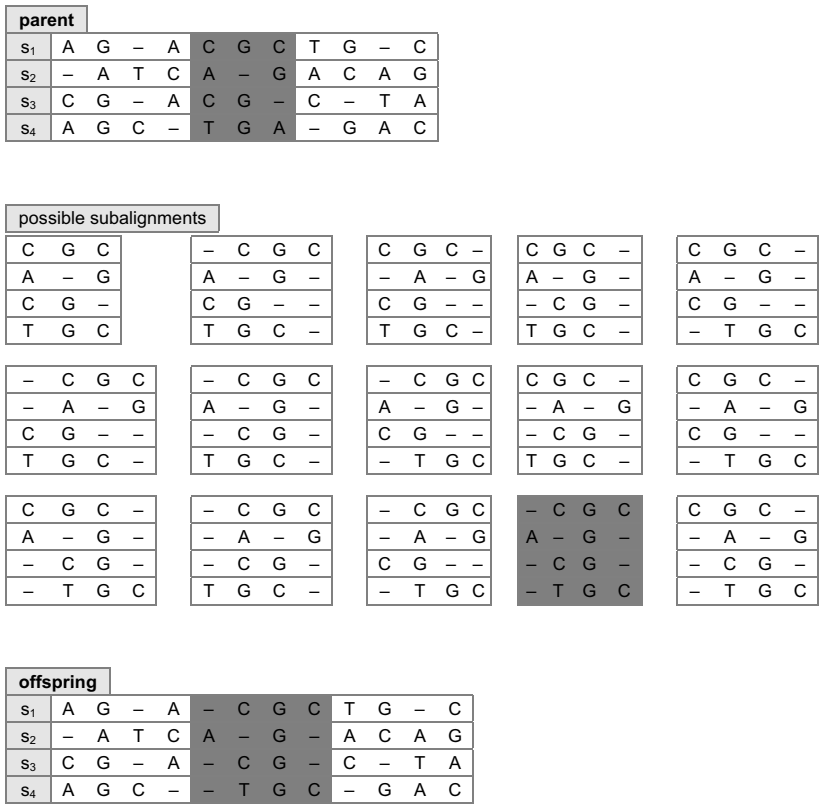


Figure 2.15: Local Gap 0-1 Alignment

LocallyAlignBlock used in *EP* [6]

All subalignments that are located between fully matched columns are determined and one of these subalignments is selected at random and optimised by the following algorithm:

1. Extract the subalignment and fill this region in the alignment with gaps.

2. Convert the extracted subalignment into a set of subsequences by removing all gaps.

3. Find the longest subsequence and insert it back into the alignment in the corresponding row, remove columns still having only gaps.

4. Place all other sequences in turn at the beginning of the corresponding row in the alignment and move the characters (starting with the last

one) into the free space to its right to the position giving the (locally) maximum fitness gain.

As this operator would always generate the same arrangement for the sub-alignment, the randomised version of it places the sequences in step 4 not in turn but in a randomly permuted order. An example is given in Figure 2.16.

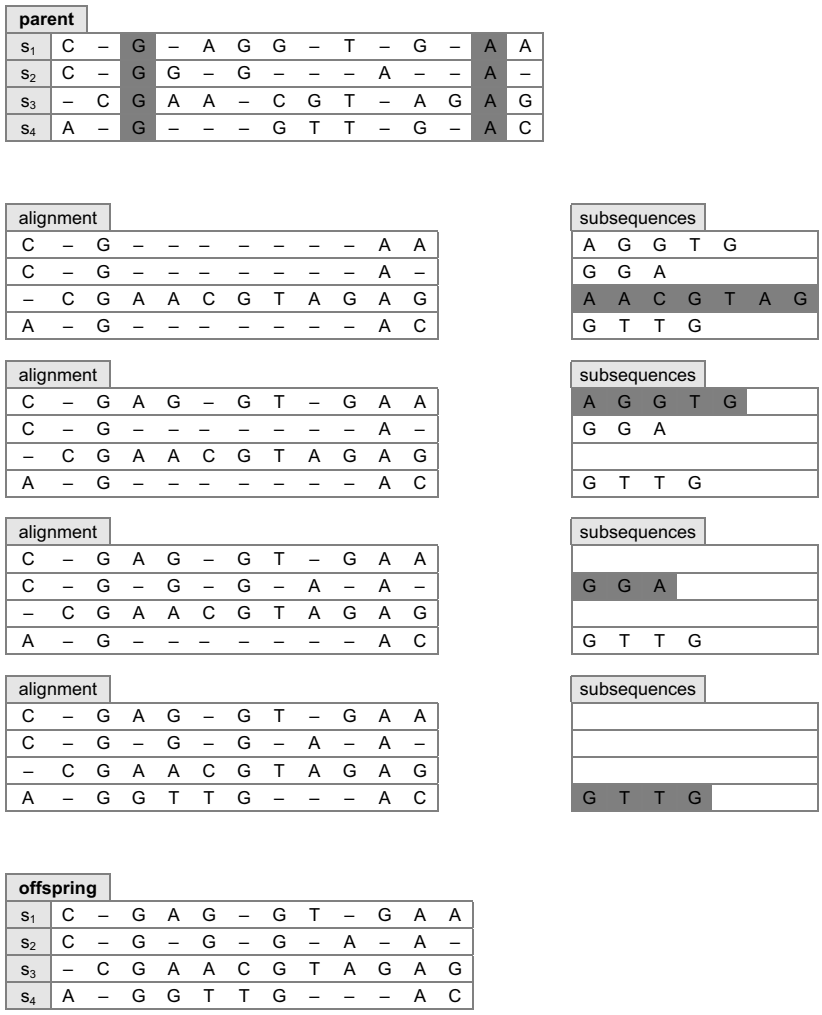


Figure 2.16: LocallyAlignBlock

BlockShuffling used in *SAGA* [21]

A block of consecutive characters of one or more sequences is randomly selected and splitted horizontally (according to the phylogenetic tree) or vertically. Then this (sub-)block is moved to the left or right position, either randomly or in a semi-hill climbing way, looking for the position with the best fitness. Similarly, this operator can be used on a block of consecutive gaps. An example is given in Figure 2.17.

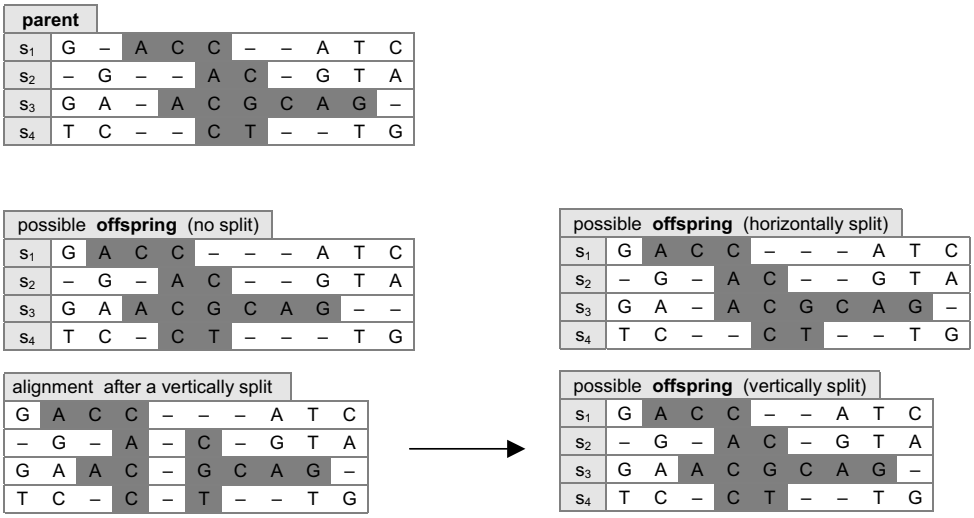


Figure 2.17: BlockShuffling

Local Optimal or Sub-Optimal Rearrangement used in *SAGA* [21]

A block is selected at random and all gaps inside are tried to be placed on the position giving the (locally) maximum fitness gain. If the optimisation requires less than a specific number of combinations (about 2000), this is done by exhaustive examination of all gap arrangements, else a local alignment genetic algorithm is used.

Block Searching used in *SAGA* [21]

A subsequence of random length (about 5-15 characters) without any gaps is selected in one of the sequences and used to form the initial profile. In a section tailored randomly (about 50-150 alignment positions) around the position of this profile, all subsequences of the same length are compared with the initial subsequence. The best matching one is selected and added to form the new profile. Then the best match in the remaining sequences is selected and added to the profile again. This goes on iteratively until a match with the profile has been identified in all of the sequences. Finally the sequences in the alignment are rearranged according to the formed profile. An example is given in Figure 2.18.

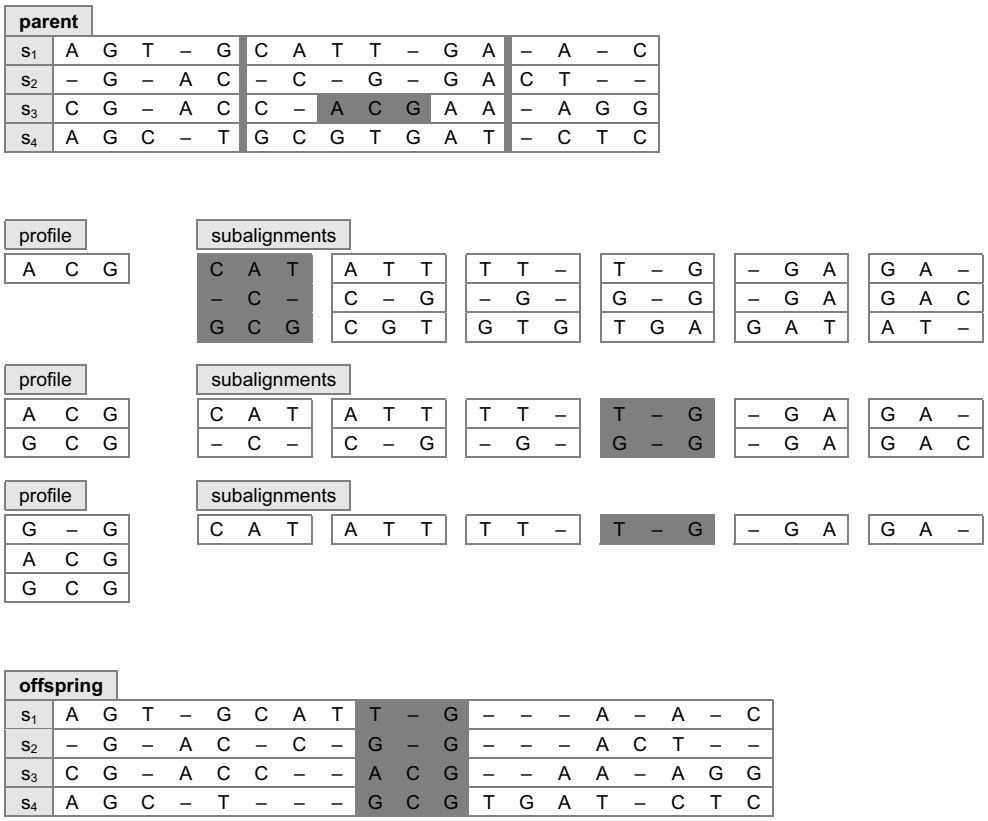


Figure 2.18: Block Searching

Gap Insertion used in *SAGA* [21]

The sequences are divided into two groups chosen by randomly splitting an estimated phylogenetic tree (as given by *CLUSTAL W*). Within each group, every sequence receives a gap insertion at the same position. In the stochastic version, the length of the inserted gaps and the insertion positions are randomly chosen. In the semi-hill climbing version, the insertion position in one group is chosen by exhaustively trying all the possible positions and comparing the fitness of the resulting alignments. An example is given in Figure 2.19.

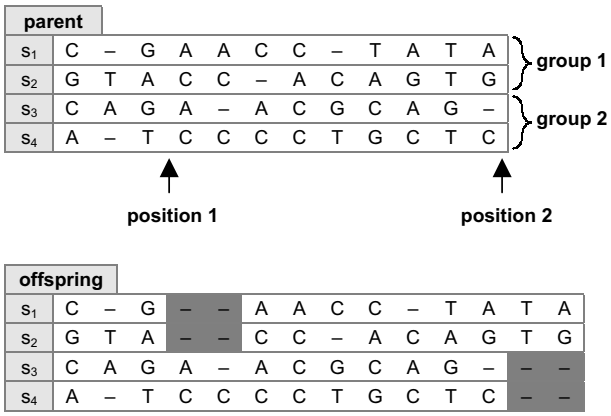


Figure 2.19: Gap Insertion

2.3 Review

A lot of different variation operators have already been designed for the multiple sequence alignment problem. Local heuristics can speed up the evolutionary process in many situations, which allows the EA to produce a (near) optimal solution in reasonable time. However, because of their greedy nature – all operations are performed only based on local considerations e.g. by creating a fully matched column – such heuristics can never guarantee any form of global improvement.

As a result of this shortcoming, all of these evolutionary algorithms only perform well on certain test instances (e.g. sequence sets of high similarity). On other test instances, however, the quality of the obtained alignments is rather

poor (also see the comparative results between our approach and *SAGA* in section 6.4). Unfortunately, tests with this latter set of test instances were only rarely performed by the authors of the EAs. Consequently, extensive analyses are still missing from the literature to date.

In chapter 4 a new evolutionary algorithm is presented, whose operators try to improve the multiple alignment based on additional global information. This additional information is extracted from the so called alignment graph, as introduced in the next chapter.

Chapter 3

The Alignment Graph

3.1 The Maximum Weight Trace Problem

The multiple alignment problem as introduced in chapter 1 can be expressed in various ways. In the complete maximum weight trace formulation, the characters of the n sequences S_i are viewed as vertices V in a complete n–partite graph G and the edges represent the corresponding pairwise character alignments. Every edge $e \in G$ has a non-negative weight w giving priority to the most reliable character pairs. We say that an alignment $\hat{S}$ realises an edge e of G if the two characters connected by e are placed in the same column of the alignment. The set of edges realised by an alignment $\hat{S}$ is called the trace of $\hat{S}$ and the weight of an alignment is the sum of the weights of the edges it realises. The problem is to find a trace with maximum weight, which is equivalent to the task of computing an alignment $\hat{S}$ of maximum weight.

Maximum weight trace as introduced by Kececioglu [15] is a restriction of the complete maximum weight trace formulation, as only a subgraph, the so–called alignment graph, and not the complete graph is used. Hence, it is also practicable for larger problem instances.

Definition 3.1 (**Alignment graph**) Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of n sequences over the same alphabet A, each sequence S_i consisting of l_i ordered characters $s_{i,\cdot}$:

$$S_i = s_{i,1}s_{i,2}\ldots s_{i,l_i}, \quad \forall i = 1, 2, \ldots, n$$

Then an alignment graph is a graph $G = (E, V)$, whose vertices correspond to the characters of the sequences and whose edges correspond to the pairwise

character alignments:

$$\begin{aligned} V &= \{s_{i,p}\}, \quad \forall i = 1, 2 \ldots, n, \quad \forall p = 1, 2, \ldots, l_i \\ E &= \{(s_{i,p}, s_{j,q}) | 1 \leq i < j \leq n, 1 \leq p \leq l_i, 1 \leq q \leq l_j\} \end{aligned}$$

Since vertices of the alignment graph and the characters of the sequences directly correspond to each other, the vertices of the graph are also referred to as characters.

Figures 3.1 and 3.2 show the correspondence between a multiple alignment and an alignment graph.

multiple alignment:

s_1	A	G	T	–	C	G
s_2	A	–	–	–	C	–
s_3	–	G	–	C	C	G
s_4	–	–	T	A	C	G

Figure 3.1: Multiple alignment

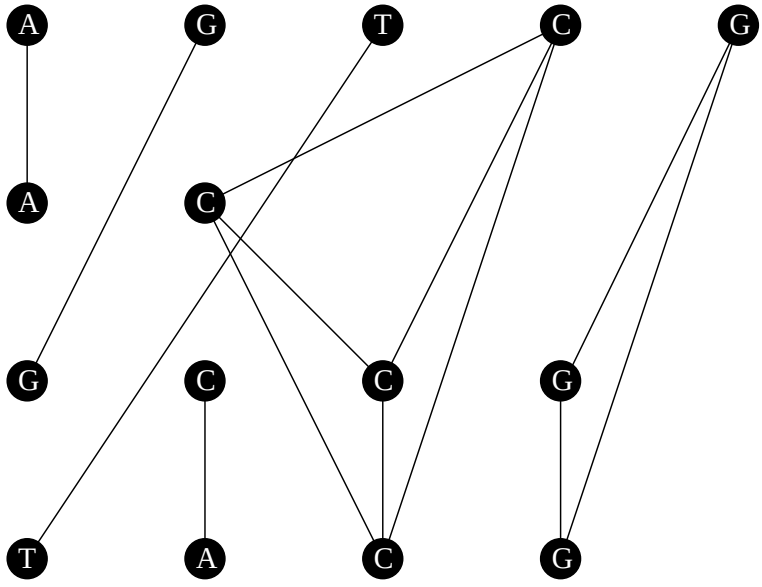


Figure 3.2: Alignment graph showing the edges realised by the multiple alignment in Figure 3.1

The edges of the alignment graph can be obtained from pairwise alignments as well as from other methods like local alignment algorithms.

Kececioglu proved that the maximum weight trace problem contains the sum-of-pairs MSA problem under certain conditions and that it is NP–hard.

Definition 3.2 (**Maximum weight trace problem**) Let S be a set of n sequences, $\hat{S}$ an alignment of S and $G = (V, E)$ the corresponding alignment graph. $\hat{S}$ realises an edge e of E if the two characters connected by e are placed in the same column of $\hat{S}$. The set of edges realised by an alignment $\hat{S}$ is called the trace of $\hat{S}$.
Then the maximum weight trace problem can be formulated as follows: Compute the alignment $\hat{S}$ that realises a trace with maximum weight:

$$\max_{\hat{S}} \sum_{e \in trace(\hat{S})} w(e)$$

Hence, the problem of computing a multiple alignment with the best sum-of-pairs score is transformed to the problem of finding a trace with maximum weight. As we also have to face this problem in our method, we will approach it more exactly in chapter 4.

3.2 Building the Alignment Graph

First of all the pairwise alignment graph is built. Each pair of sequences is aligned one by one using e.g. the dynamic programming algorithm of Myers and Miller [17]. So $\binom{n}{2}$ pairwise alignments have to be computed and stored in the pairwise alignment graph.

Without considering the complexity of building the pairwise alignments, the time to build the graph and the memory required is:

$$O(l \cdot n^2), \quad \text{where } l = \max_{i=1\ldots n} l_i.$$

The total time including the pairwise alignments is $O(l^3 \cdot n^2)$.

An example for pairwise alignments is given in Figure 3.3. The corresponding pairwise alignment graph is shown in Figure 3.4.

sequences

s_1	A	G	T	C	G
s_2	A	C			
s_3	G	C	C	G	
s_4	T	A	C	G	

pairwise alignments

s_1	A	G	T	C	G
s_2	A	–	–	C	–

s_1	A	G	T	–	C	G
s_4	–	–	T	A	C	G

s_2	–	A	C	–	
s_4	T	A	C	G	

s_1	A	G	T	C	G
s_3	–	G	C	C	G

s_2	A	C	–	–	
s_3	G	C	C	G	

s_3	G	C	C	G	
s_4	T	A	C	G	

Figure 3.3: Pairwise alignments as computed by *CLUSTAL W*

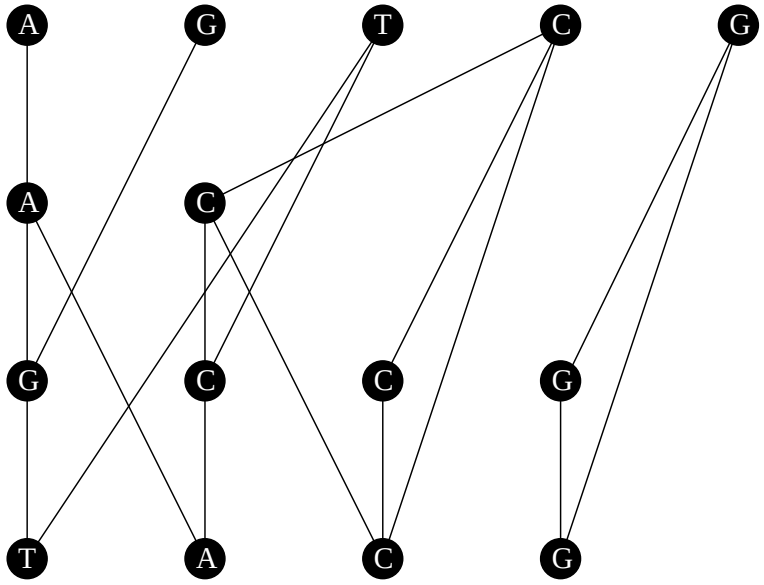


Figure 3.4: Pairwise alignment graph corresponding to the pairwise alignments of Figure 3.3

3.3 Assigning a Weight

Within this pairwise alignment graph, a weight is assigned to all edges representing the pairwise character alignments. These weights can be determined in various ways. The following scoring schemes have been implemented and compared:

Sequence identity score (SIS): For every pair of aligned sequences, the number of character matches m_1 and character mismatches m_0 is computed (comparisons with gaps remain unconsidered), then the percent sequence identity score is equal to:

$$SIS = \frac{m_1}{m_1 + m_0} \cdot 100,$$

which gives a result between 0% and 100%.

***CLUSTAL W*'s identity score (CIS)**: As above, a percent sequence identity score is calculated, but additionally, a weighted similarity matrix and two types of gap penalties (for opening and extending gaps) are used for this computation. For further details see Thompson et al. [28].

1-0 score (1-0S): Any edge, which connects the same characters, receives the value 1 (100%), otherwise its weight is set to 0 (0%).

All these previously described scoring schemes assign the same weight to each character of a pairwise alignment. Fauster [7] proposed a different method using *CLUSTAL W*'s similarity matrices such as *PAM* or *BLOSUM*. These similarity matrices contain scores for all possible matches and mismatches of amino acid symbols. However, applying such a score directly to an edge of our graph would only reflect the relation between this single aligned character pair and would not contain any information about the rest of the pairwise alignment. Therefore, always a block of consecutive aligned character pairs around one pair is considered during the computation. Fauster recommended the following scoring scheme (with $b = 10$).

***CLUSTAL W*'s similarity matrices with window filtering (CSM)**: Let $\hat{S}$ be a pairwise alignment of the sequences S_i and S_j, $\{e_1, e_2, \ldots, e_m\}$ a set of edges representing the m pairwise character alignments in $\hat{S}$, $M(e_k)$ a similarity matrix score reflecting the similarity between the corresponding characters of e_k and $(2 \cdot b + 1)$ the arbitrary length of the block of consecutive

aligned character pairs. Then the following weight is assigned to an edge e_k:

$$\begin{aligned} \text{CSM} &= M(e_k) + \sum_{i=1}^{b} \left[M(e_{k-i}) \cdot \left(1 - \frac{i}{b+1}\right) \cdot g^{-} \right] \\ &+ \sum_{i=1}^{b} \left[M(e_{k+i}) \cdot \left(1 - \frac{i}{b+1}\right) \cdot g^{+} \right], \end{aligned}$$

with

$$g^{-} = \begin{cases} 1 & \text{if there is no gap between position } k-1 \text{ and } k-i, \\ 0 & \text{else,} \end{cases}$$

and

$$g^{+} = \begin{cases} 1 & \text{if there is no gap between position } k+1 \text{ and } k+i, \\ 0 & \text{else.} \end{cases}$$

An extensive review and comparison of these four implemented scoring schemes can be found section 6.2.1. For exemplification the sequence identity score is used in the following sections of this chapter to assign a weight to each edge.

Figure 3.5 shows the weights for the pairwise alignments of Figure 3.3, the corresponding weighted pairwise alignment graph is given in Figure 3.6.

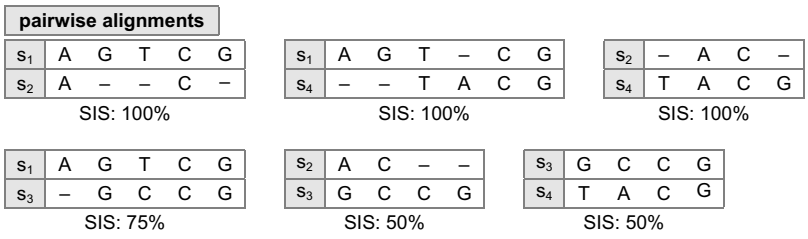


Figure 3.5: Pairwise alignments including weights (SIS)

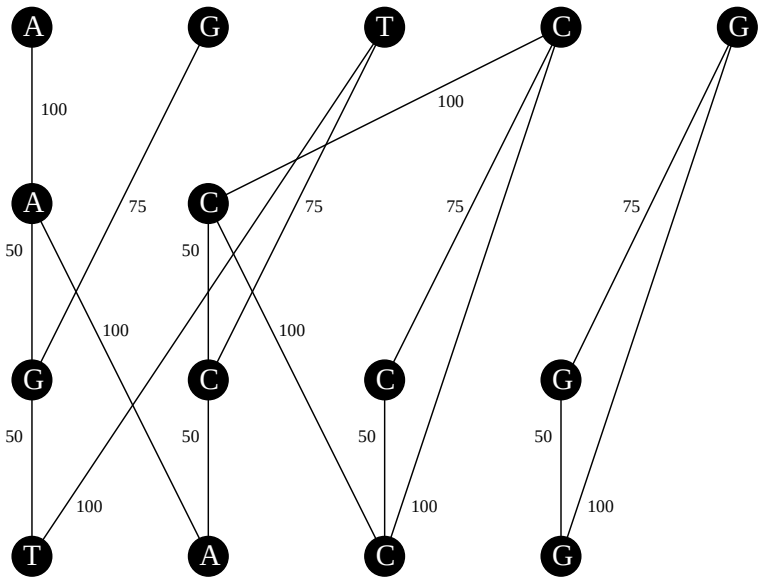


Figure 3.6: Corresponding pairwise alignment graph including weights

To put even more local and/or global information into the alignment graph, it is reasonable to add new edges or to modify the weight of existing edges based on more or less heuristic considerations. One method is presented in the next section: the alignment graph extension.

3.4 The Alignment Graph Extension

The overall idea of the alignment graph extension is to combine information in such a manner that the final weight of any pair of aligned characters reflects some of the global information contained in the whole alignment graph. This can be achieved by examining the consistency of each aligned character pair with the aligned character pairs from all other alignments. The process is called alignment graph extension and is a generalisation of the idea of the extended library as described by Notredame et al. [22].

In the simplest case this means that having a character X, which is incident to character Y as well as to character Z, the characters Y and Z are aligned through character X too. This relation can be represented by adding an edge between Y and Z or by increasing its weight accordingly, if this edge already exists.

An example is given in Figure 3.7.

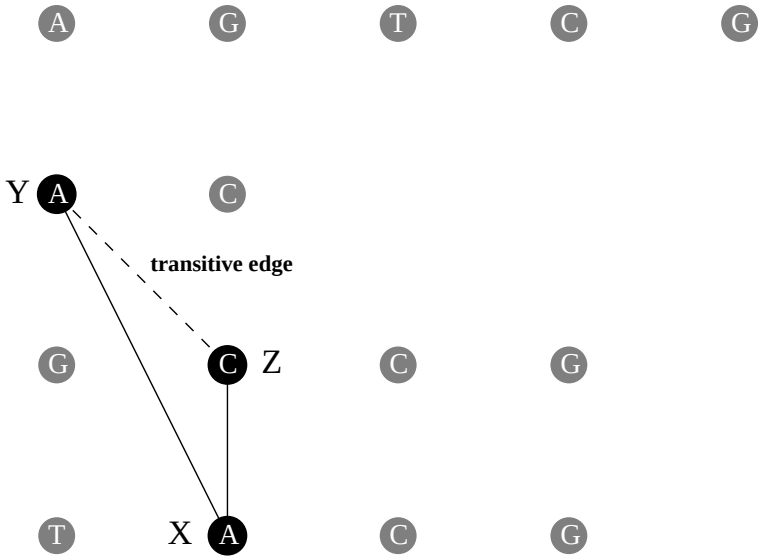


Figure 3.7: Transitive edge

In other words, if there exists an edge between character X and Y as well as an edge between character X and Z, the transitive relation between Y and Z seems to be a desirable alignment, too. This method can also be generalised to more than three characters.

Definition 3.3 (Edge path) Let $s_{i_\alpha,p_\alpha}, 1 \leq \alpha \leq k$ be characters of $k \geq 2$ different sequences $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$. Then a sequence of edges is called edge path from character s_{i_1,p_1} to character s_{i_k,p_k}, if there exists an edge between s_{i_α,p_α} and $s_{i_{\alpha+1},p_{\alpha+1}}, \forall \alpha = 1, 2, \ldots, k-1$.

Definition 3.4 (Transitive edge) If there exists an edge path between character s_{i_1,p_1} and character s_{i_k,p_k}, the edge between s_{i_1,p_1} and s_{i_k,p_k} is called transitive edge of level k.

As every edge connects characters of different sequences and each sequence is allowed only once on the edge path, the length of the edge path – and therefore the level k – is limited by the number of sequences.

Figure 3.8 shows some valid edge paths.

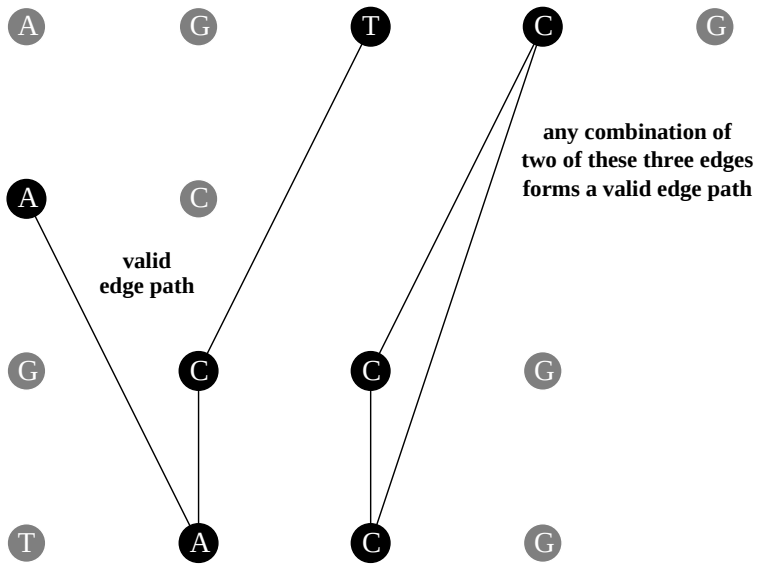


Figure 3.8: Valid edge paths

Building the alignment graph extension of level k therefore leads to the problem of finding all transitive edges of level $\leq k$ and adding them to the graph or updating their weights accordingly, if they already exist.

The extension of the alignment graph of Figure 3.6 is shown in Figure 3.9.

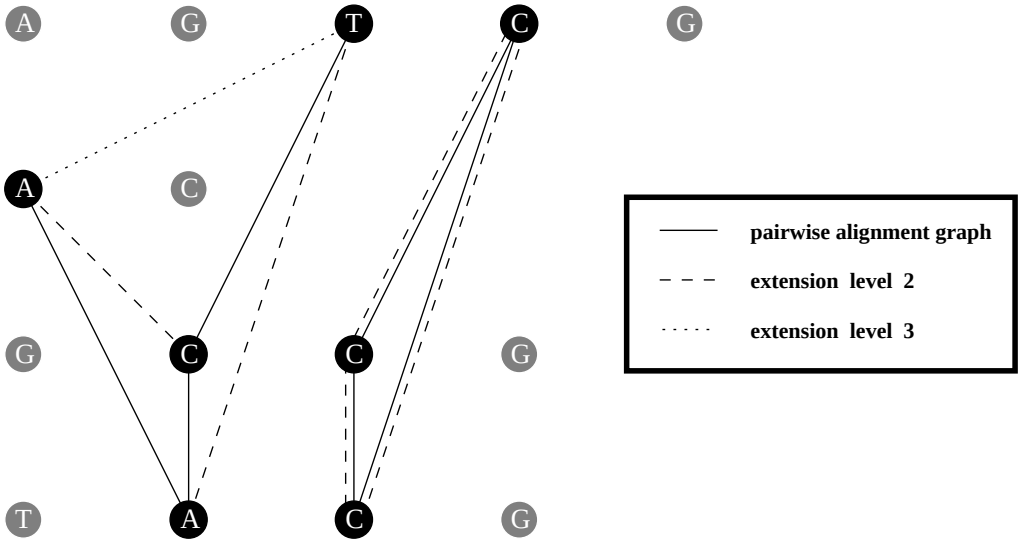


Figure 3.9: Alignment graph extension

The weights for the transitive edges e_t are calculated using

$$w(e_t) = \frac{w_{min}(e_i)}{(k-1)}, \quad k \geq 2,$$

where $w_{min}(e_i)$ is the minimum weight of all edges e_i on the edge path.

An example is given in Figure 3.10.

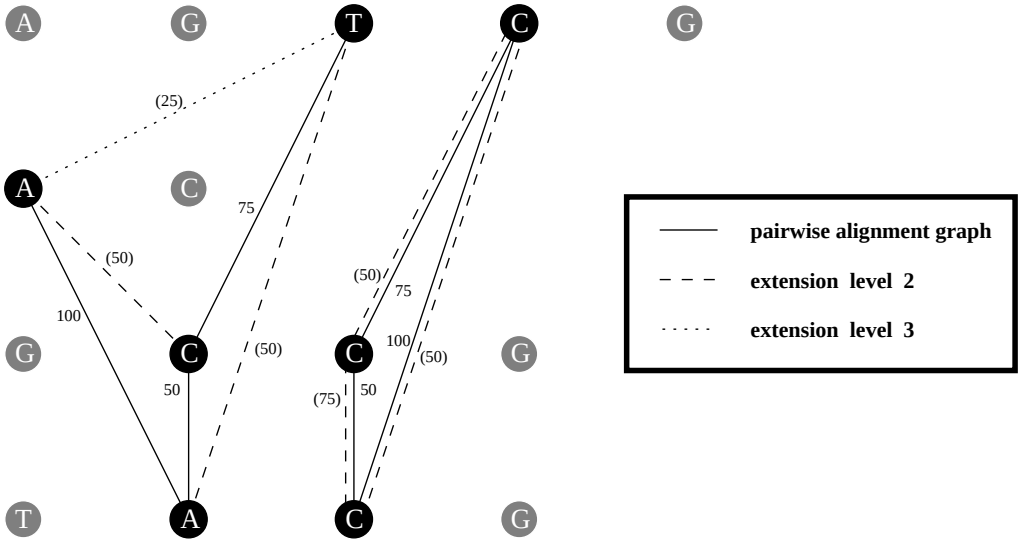


Figure 3.10: Alignment graph extension including weights

As characters connected by a transitive edge of a shorter edge path are more likely to be related than others, the length k of the edge path is also considered in this formula. For $k = 2$ this formula exactly provides the same weights as used for the extended library in [22]. Also note that an edge can be considered several times, as it can be the transitive edge of different edge paths (even of paths of different length). Therefore its weight can be updated more than once.

The alignment graph after the whole extension process is shown in Figure 3.11.

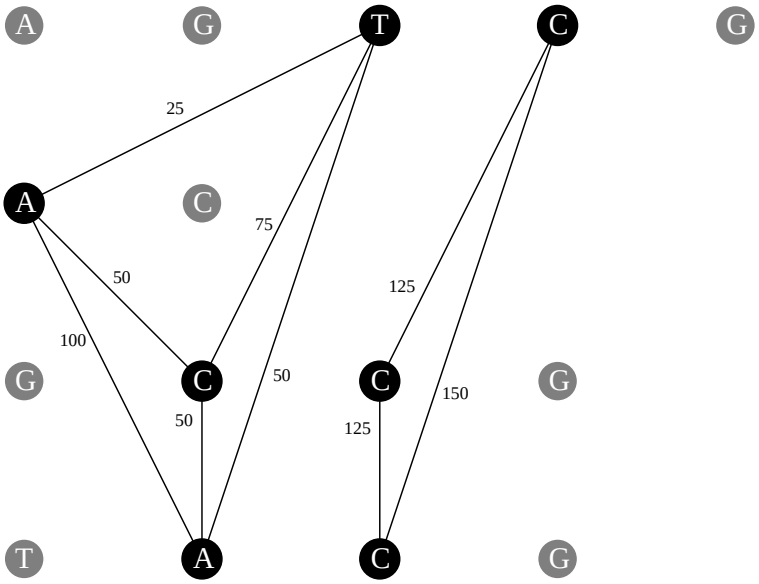


Figure 3.11: Alignment graph after extension

Note that for further computation, all weights are scaled to the interval $[0, 1]$.

To perform the alignment graph extension, a recursive depth-first-search algorithm can be used (see Listing 3.1). Limiting the depth of the recursion allows to set the level of the alignment graph extension.

Listing 3.1

```
.   global <bool> visited[1 ... number of sequences]


.   EXTENSION ()
.   begin
.     forall s = (1 ... number of sequences)
.       forall p = (1 ... length of sequence s)
.         visited[s] = false
.         DFS (s, p, s, p, 0, 0)
.   end
```

```
.   recursive DFS (s_start, p_start, s, p, k, w)
.   begin
.     visited[s] = true
.     forall s_end = (1 ... number of sequences)
.       if visited[s_end] == false
.         forall p_end = ((s_end, p_end) incident to (s, p))
.           edgew = weight of edge (s_end, p_end) (s, p)
.           if (w == 0) or (edgew < w)
.             minw = edgew
.           else
.             minw = w
.           if (k ≥ 2)
.             w = minw / (k − 1)
.             w = w/2 as any edge is considered twice
.             add edge (s_start, p_start) (s_end, p_end) with weight w to the
              graph (or update w accordingly if the edge exists)
.           if (k < desired level of extension)
.             DFS (s_start, p_start, s_end, p_end, k + 1, minw)
.           visited[s_end] = false
.   end
```

Having a pairwise alignment graph with n sequences and the maximum length l, there are at most $n-1$ edges incident to each character (one to each other sequence). So for the recursion with a maximum depth k, at most

$$(n-1) \cdot (n-2) \cdot \ldots \cdot (n-k) = O(n^k)$$

edges have to be considered for each of the $n \cdot l$ characters. Thus, in the worst case, the algorithm performs in time

$$O(l \cdot n^{k+1}).$$

However, as there are fewer edges and n is rather small compared to l in general, the running time is typically shorter.

3.5 Validation of the Extension

To rate how good the alignments which are derived from an alignment graph can be, reference alignments of the *BAliBASE* library [29] are compared

with their corresponding alignment graphs. In general, there are four possible relations between two characters of different sequences in the multiple alignment and their corresponding edge in the alignment graph:

1. The two characters are aligned in the multiple alignment and the edge exists in the alignment graph.

2. The two characters are not aligned in the multiple alignment and the edge does not exist in the alignment graph.

3. The two characters are aligned in the multiple alignment and the edge does not exist in the alignment graph.

4. The two characters are not aligned in the multiple alignment and the edge exists in the alignment graph.

While relations 1 and 2 characterise optimal situations, the occurrence of relations 3 and 4 should be as low as possible.

Definition 3.5 Let P be the set of all pairwise character alignments appearing in the reference alignment and $R_E \subseteq P$ be the subset of all pairwise character alignments for which a corresponding edge exists in the alignment graph $G = (V, E)$. Then R_E is also the subset of all edges of the alignment graph that are realised by the multiple alignment.

Thus, considering weights for all edges, an alignment graph should achieve the following criteria:

Goal 1: As many pairwise character alignments in the reference alignment as possible should have corresponding edges in the alignment graph, i.e. $|R_E|$ should be maximal.

Goal 2: At the same time, as few edges as possible should exist which do not correspond to pairwise character alignments, i.e. $|E \backslash R_E|$ should be minimal. At least, such non-realised edges should have a lower weight.

Based on these goals, various comparisons between the *BAliBASE* reference alignments and their corresponding alignment graph with and without extension have been performed. The results of these tests can be found in section 3.5.2.

3.5.1 BAliBASE v1.0

The performance of alignment programs usually depends on the number of sequences and the level of similarity between them. Additionally other factors like the existence of large insertions may affect the qualitiy of the produced aligment. *BAliBASE* is a database of more than 140 manually-refined multiple sequence alignments specifically designed for the evaluation and the unbiased comparison of multiple sequence alignment programs. These protein alignments can be divided into four categories (=reference sets), which encompass most of the situations arising when computing multiple alignments.

Reference 1 contains alignments of maximal 6 equi-distant sequences with pairwise percent identities within a specified range. Based on these percent identities, the reference set is further divided into three subcategories: Reference 1a ($< 25\%$), Reference 1b ($20\% - 40\%$), Reference 1c ($> 35\%$). All the sequences are of similar length, with no large insertions or extensions.

Reference 2 aligns up to three 'orphan' sequences (less than 25% identical) from reference set 1 with a family of at least 15 closely related sequences.

Reference 3 consists of up to 4 sub-groups, with less than 25% identity between sequences from different groups.

Reference 4 is divided into two sub-categories containing alignments of up to 20 sequences including N/C-terminal extensions (Reference 4a) and insertions (Reference 4b).

Note that some of the reference alignments generated some problems during computation (e.g. out of memory) and therefore have not been considered for further analyses. A complete list of all used test instances can be found in the appendix.

A frequently encountered problem when using reference alignments is the effect of ambiguous regions, which can only be aligned arbitrarily and lead to biases in the comparisons of programs. This is why the authors additionally annotated these alignments by marking blocks considered to be correctly aligned. Altogether these trusted regions represent about 58% of the aligned characters.

3.5.2 Results

As pointed out before, an alignment graph should meet special criteria as far as possible. Two characters which are not connected by an edge in the alignment graph only have a small probability to be aligned, whereas two connected characters are much more likely to be aligned. Therefore, considering all pairs of aligned characters in the reference alignment, as many corresponding edges as possible should exist in the alignment graph, i.e. $|R_E|$ should be maximal ($\rightarrow$ **Goal 1**).

$|R_E|/|P|$ is calculated for all *BAliBASE* alignments with and without the alignment graph extension and can be seen in Figure 3.12. The respective improvements made through the alignment graph extension for each reference set are shown in Table 3.1.

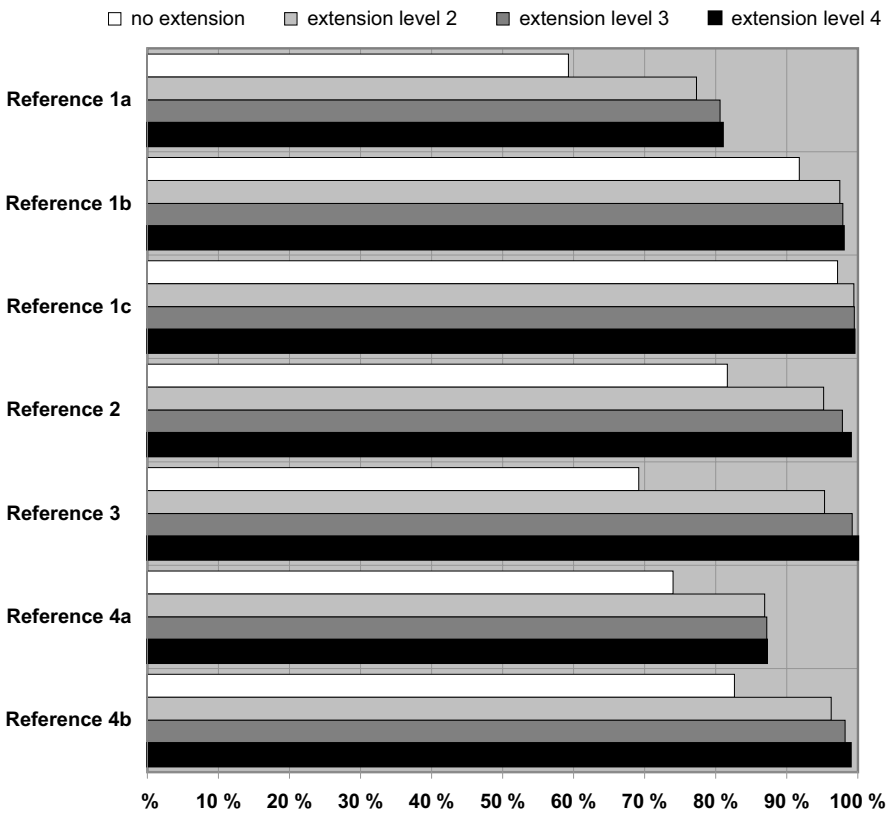


Figure 3.12: $|R_E|/|P|$ in percent

	no ext	no ext → level 2	level 2 → level 3	level 3 → level 4
Reference 1a	59,25 %	+18,02 %	+3,31 %	+0,36 %
Reference 1b	91,75 %	+5,71 %	+0,42 %	+0,10 %
Reference 1c	97,14 %	+2,29 %	+0,05 %	+0,00 %
Reference 2	81,61 %	+13,57 %	+2,62 %	+1,18 %
Reference 3	69,15 %	+26,15 %	+3,91 %	+0,80 %
Reference 4a	73,97 %	+12,91 %	+0,28 %	+0,00 %
Reference 4b	82,61 %	+13,62 %	+1,97 %	+0,77 %

Table 3.1: Increase of $|R_E|/|P|$ in percent

As can be seen, extending the pairwise alignment graph generally increases the quoted percentage. However, the improvements are quite different: while sequences with high similarity like reference set 1c already start with a high percentage and therefore will not be improved much by a further extension process, for sequences with lower similarity even the extension to level 3 provides appreciable results.

As the extension to level 4 only shows marginal improvements for all reference sets and also causes memory problems for larger instances, it has not been considered in the following.

On the downside, during the extension process a lot of edges are added to the alignment graph, which are not realised in the reference alignments. Figures 3.13 and 3.14 show the average size $|E|$ of the alignment graph of each reference set with and without extension. Additionally, the number of realised edges $|R_E|$ is charted. Note the logarithmic scaling used in the diagrams.

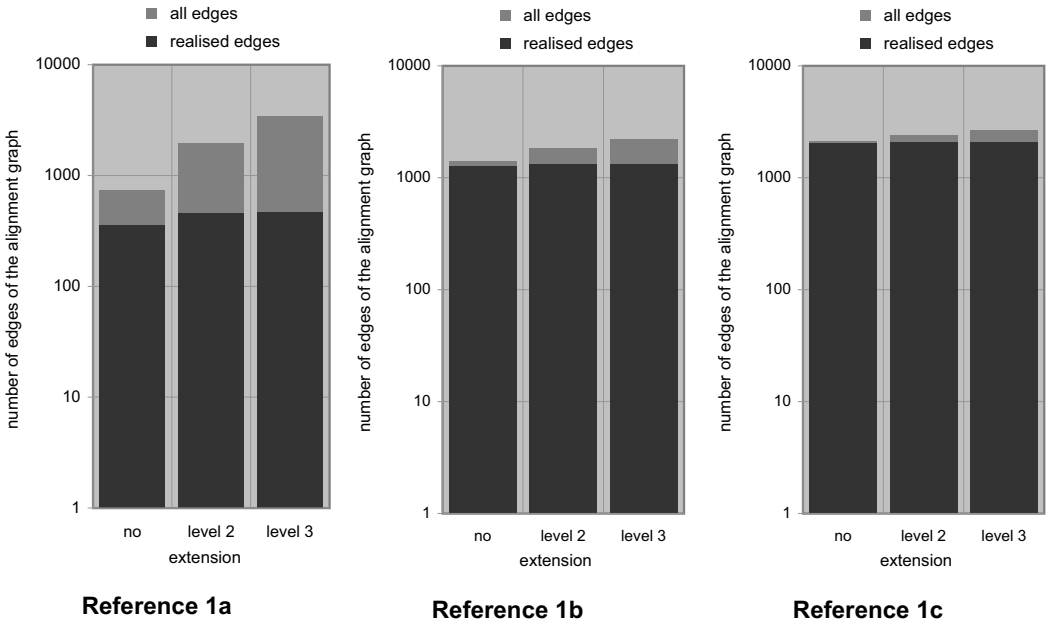


Figure 3.13: Size of the alignment graph (part A)

As documented in Figure 3.13 and 3.14, the extension process generally adds much more non-realised than realised edges to the alignment graph. For some reference alignments with lower similarity, the size of the alignment graph even grows exponentially with the level of extension.

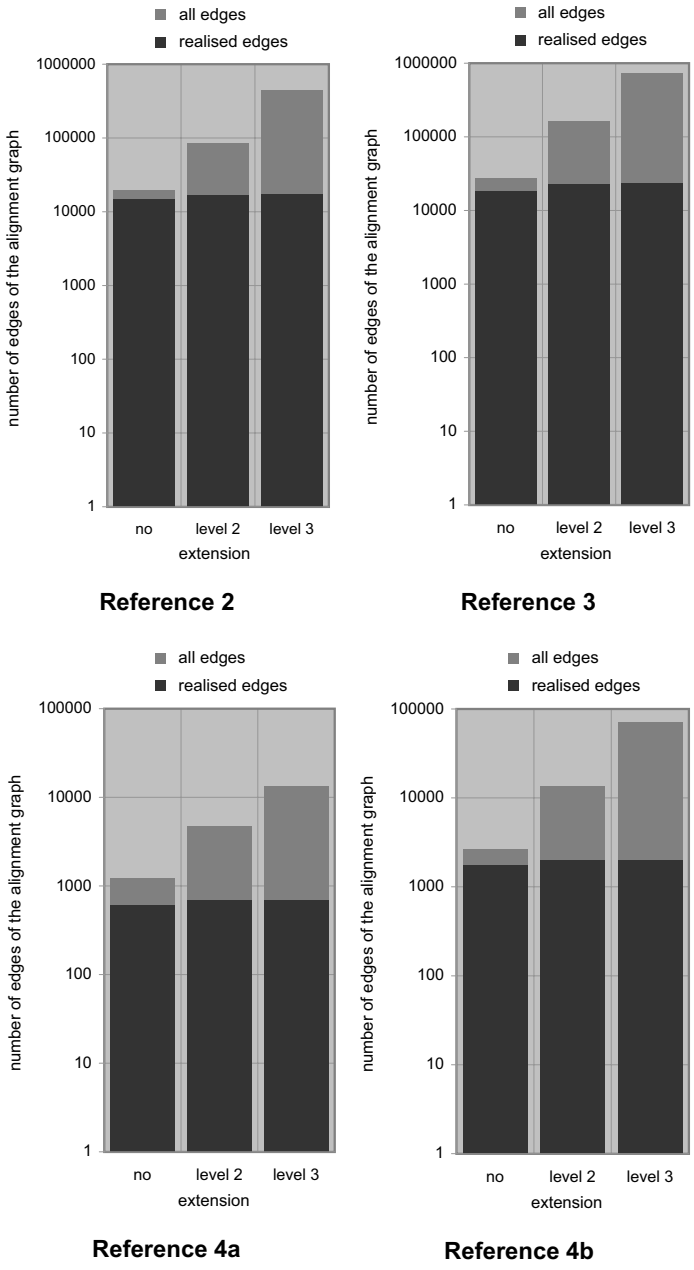


Figure 3.14: Size of the alignment graph (part B)

Just considering the previous results, it is difficult to decide whether it is worth to extend the alignment graph to a higher level or to work just with the pairwise alignment graph. On the one hand, there is the improved num-

ber of realised edges with respect to the reference alignments, but on the other hand, there is the additional computational effort and more importantly, the super–proportionally growing size of the alignment graph with its extra memory requirements.

For a satisfying answer to this question, it is also necessary to consider the weights of the edges: edges realised in the reference alignments should have a higher weight than non-realised ones ($\rightarrow$ **Goal 2**).

As shown in the next diagrams, an extended alignment graph generally fulfills this condition. The validation proceeds in two steps:

Step 1: The edges are partioned into intervals I according to their weights and for each interval, $|R_E(I)|/|E(I)|$ is computed. Histograms showing these frequencies are generated for each reference set and charted in Figures 3.15 and 3.16.

As desired, the extension process increases the quoted probabilities for edges of higher weight and decreases these probabilities for edges of lower weight. So, an edge of higher weight in the extended alignment graph also provides a higher probability, that the corresponding characters are aligned in the reference alignment too (compared with an edge of same weight in the pairwise alignment graph). These improvements become especially apparent for test instances of lower sequence similarity (like reference set 1a in Figure 3.15), but are also observable for all other reference alignments.

What is still unknown is the number of edges in each of these intervals of Figures 3.15 and 3.16. Edges with high weight should generally predominate in the solution. The weight-based distribution of the edges is analysed in the following.

Step 2: All edges of the alignment graph are sorted by weight in descending order. Then, one by one, an edge is picked and checked whether it is realised by the alignment or not. This proportion is plotted for all reference sets in Figures 3.17 and 3.18. Note that D denotes the subset of all already examined edges of E.

As shown in Figures 3.17 and 3.18, much more edges of high weight of the extended alignment graph than of the pairwise alignment graph are incorporated in the reference solution. So, generally fewer edges of the the extended alignment graph compared with the pairwise alignment graph need to be

examined to obtain the same percentage of edges for which a corresponding pairwise character alignment exists in the reference alignment.

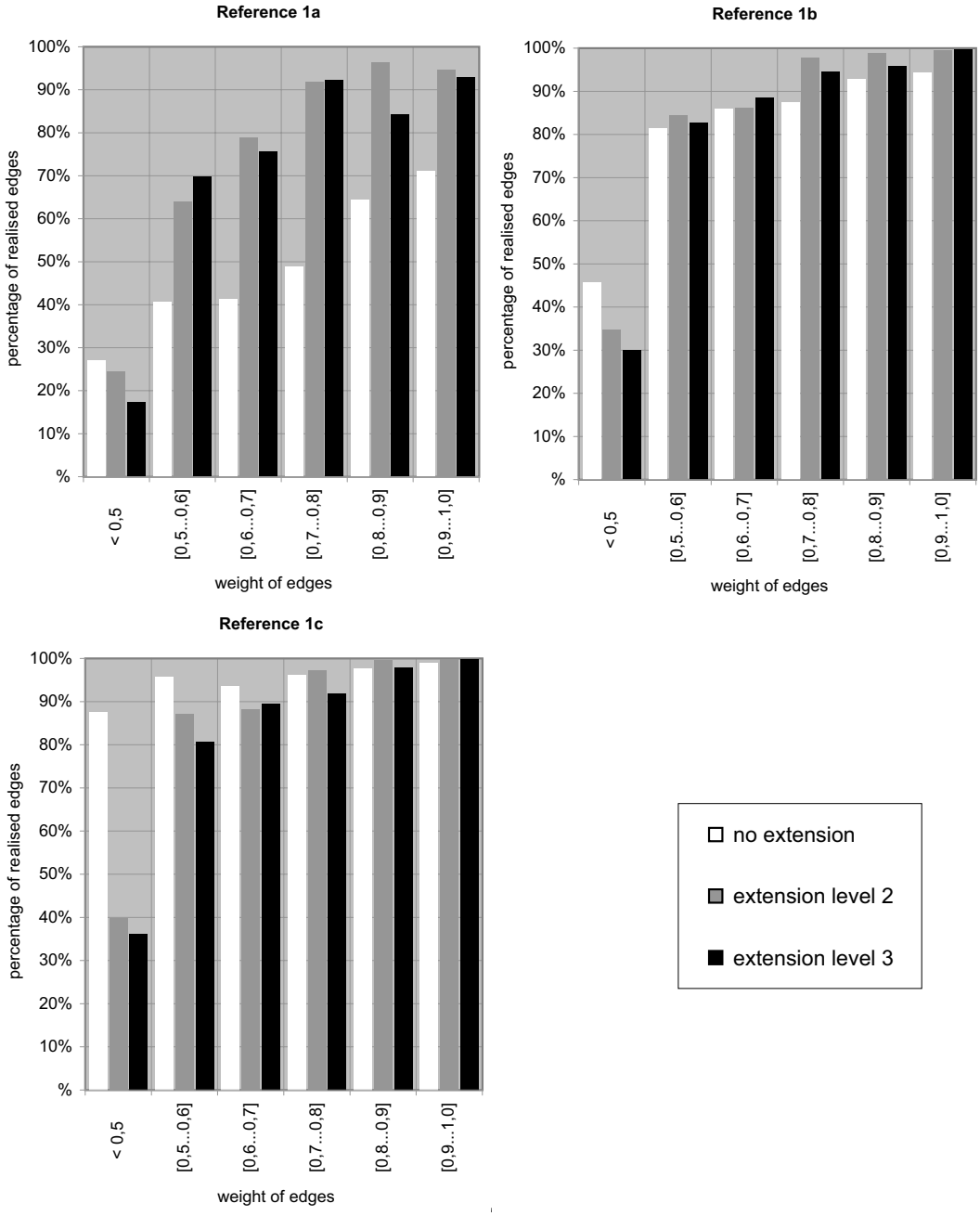


Figure 3.15: Histograms showing the percentage of realised edges $|R_E(I)|/|E(I)|$ (part A)

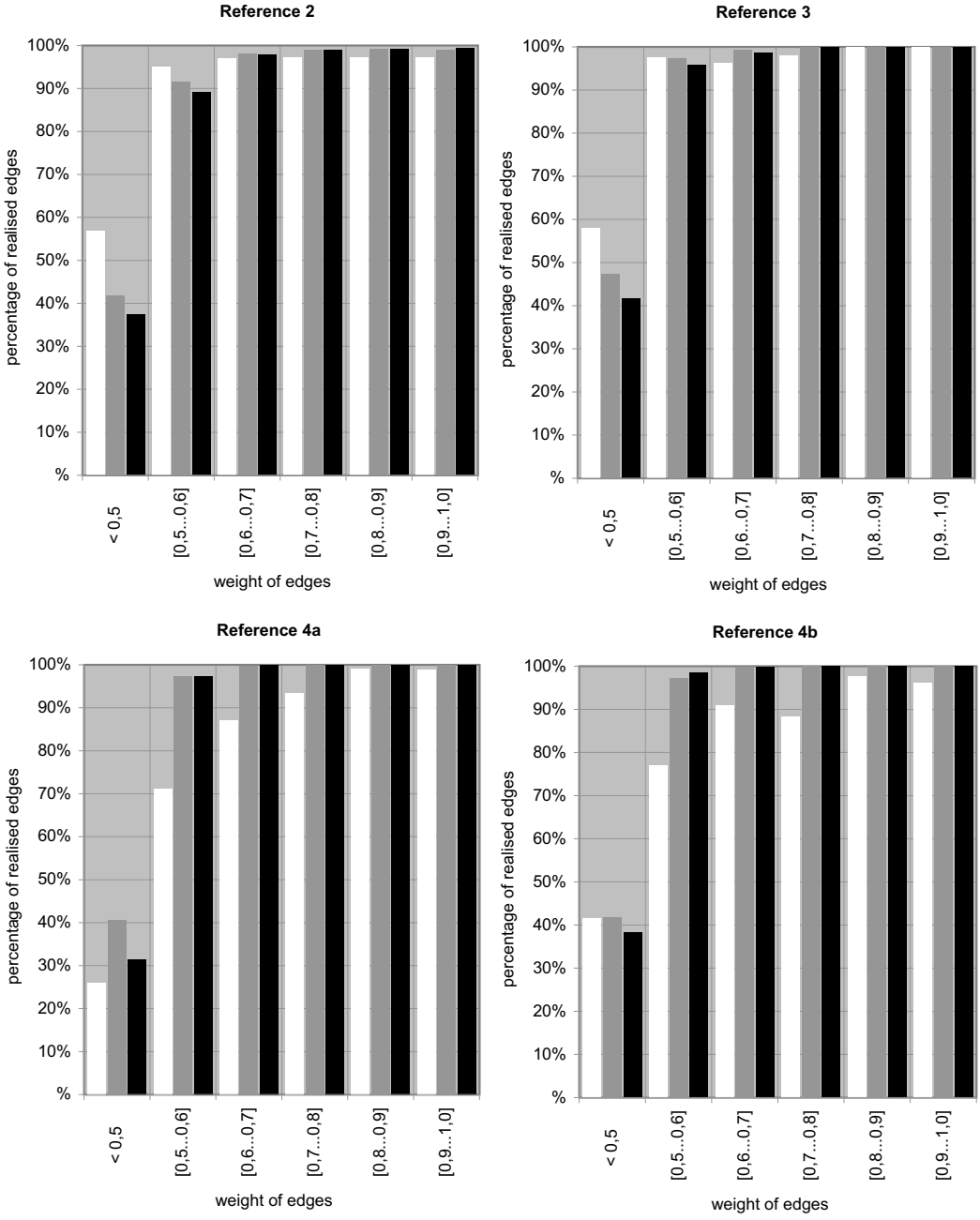


Figure 3.16: Histograms showing the percentage of realised edges $|R_E(I)|/|E(I)|$ (part B)

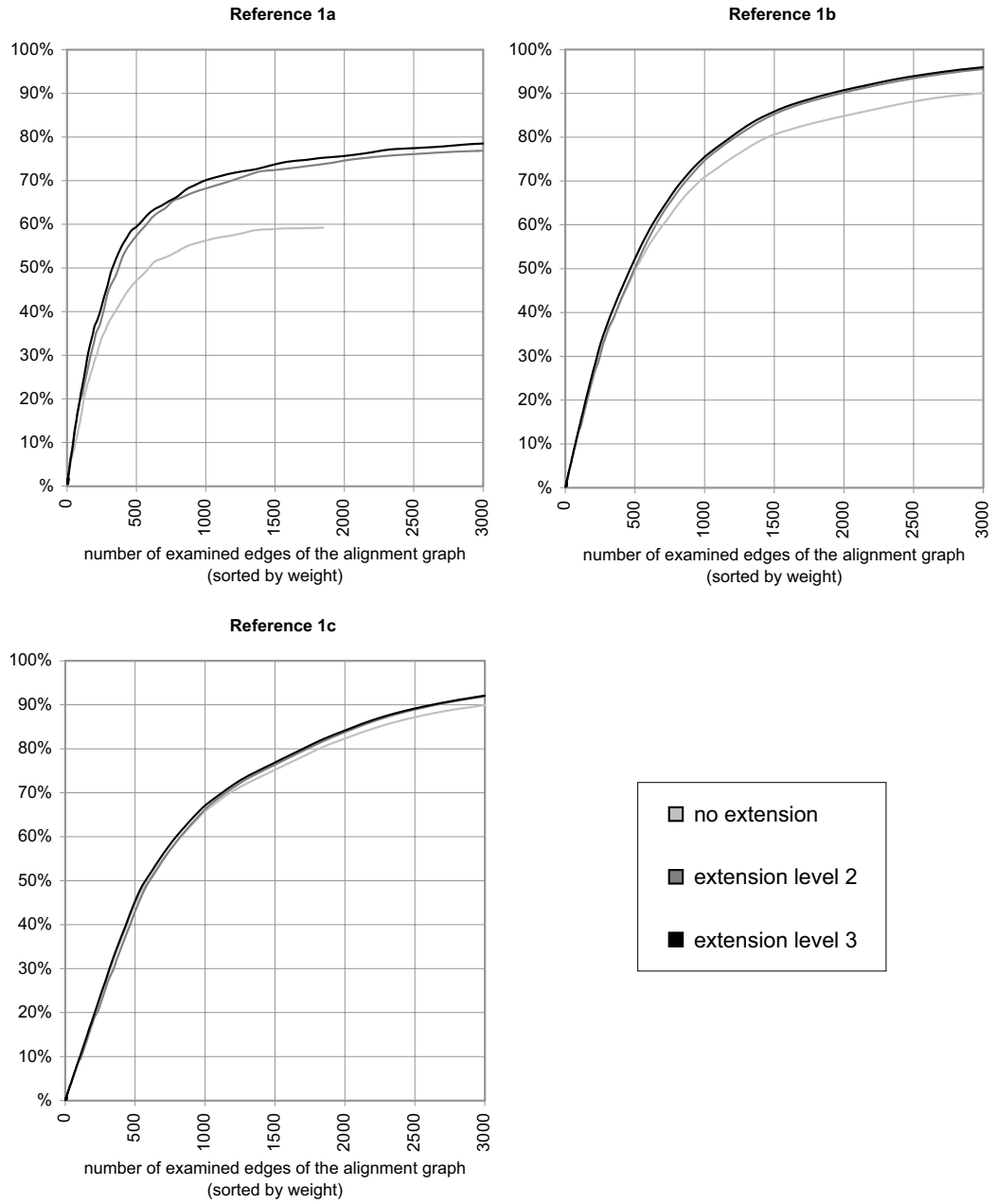


Figure 3.17: Relation between $|D|$ and $|R_D|/|P|$ in percent (part A)

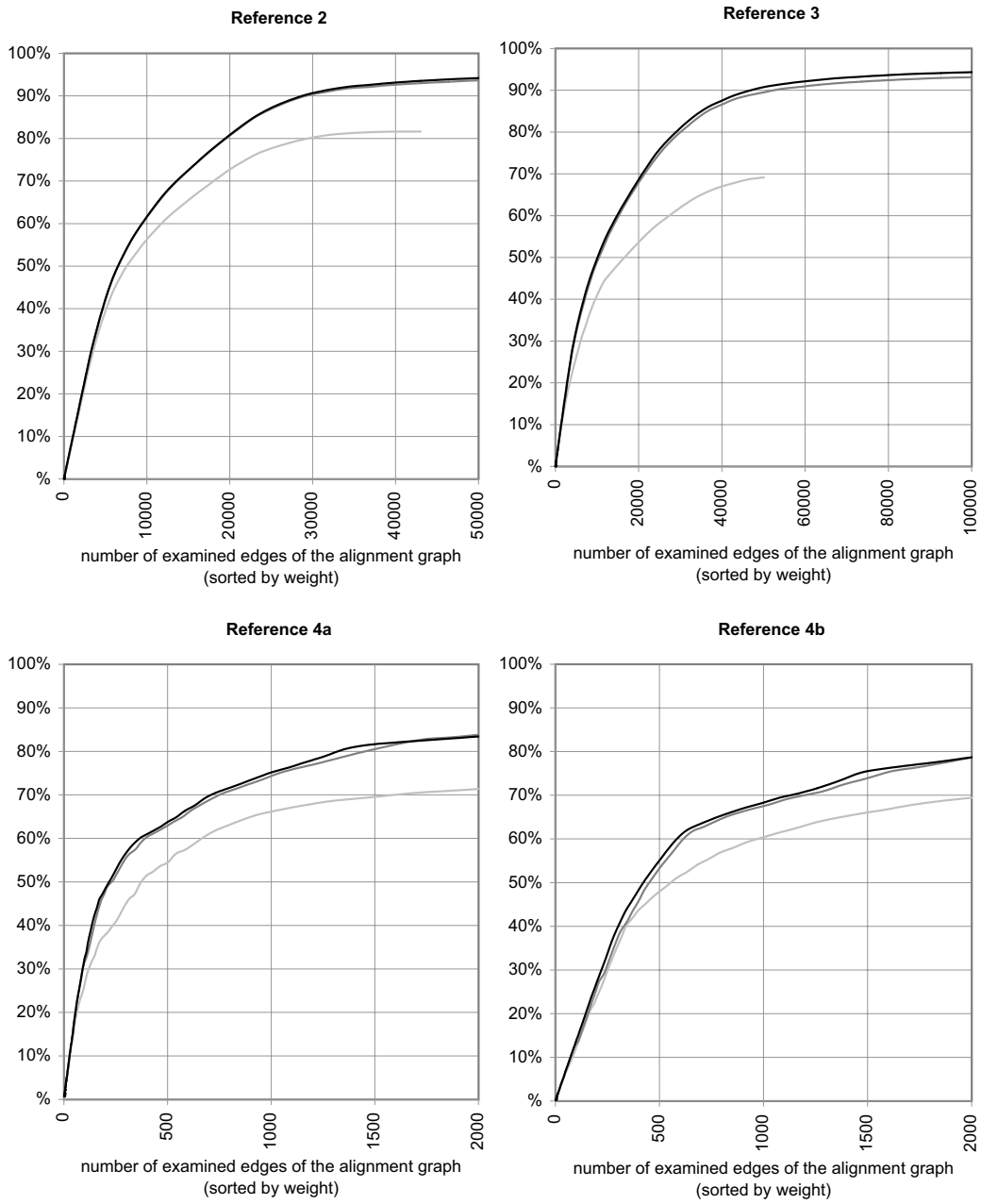


Figure 3.18: Relation between $|D|$ and $|R_D|/|P|$ in percent (part B)

Combining the results of this section, the following conclusions can be drawn:

- For sequences with lower similarity, the computational effort and the required memory for storing the alignment graph grows exponentially with the level of extension. However, beside of the non-realised edges, also a lot of edges realised in the reference alignments are added during the extension process. The extension also ensures, that realised edges generally have a much higher weight than non-realised edges.
- For sequences with high similarity, the pairwise alignment graph already includes most of the edges realised in the reference alignments. Therefore, only a few realised edges can be added during the extension process. However, the extension updates the weight of existing edges in such a way, that realised edges normally have a higher weight than non-realised edges. Besides, the additional computational effort is by far less dramatic than for sequences with lower similarity.

Following these conclusions, the alignment graph generally has been extended for further computation. Such standardised treatment seems to be a reasonable compromise regarding additional effort, profit and simplification. The level of the alignment graph extension is further discussed in section 6.2.1.

3.6 Distribution of Edges

As shown in the previous results, edges with high weights – respectively low rank when considering all edges sorted by weight in decreasing order – predominate in the *BAliBASE* reference alignments. When computing or improving a multiple alignment based on the alignment graph e.g. by applying an evolutionary algorithm, it is therefore reasonable to favour these edges by biasing the used operators.

Raidl et al. [25] analysed the weight-based distribution of edges in optimal solutions for different graph problems like the degree-constrained minimum spanning tree or the traveling salesman problem. They derived probabilites for selecting edges based on their rank to be incorporated into candidate solutions (such that the average number of edge-selections until finding an edge of an optimal solution is minimised) and proved, that an edge-selection strategy using these approximately optimal probabilites performs best. The following edge-selection strategy is proposed:

$$r = \left\lfloor \frac{2\log(1-u)}{\log|R_E| - \log(|R_E|+1)} \right\rfloor \mod |E| + 1$$

with r being the random edge-rank and $u \in [0, 1)$ being a uniformly distributed random number. Applying this approach to the MSA problem leads to the problem of finding an appropriate value $|R_E^*|$ for $|R_E|$, as the number of edges of the solution is unknown during the computation.

Therefore, we analyse the percentage of realised edges $|R_E|$ with respect to all edges $|E|$ of the alignment graph for all $BAliBASE$ alignments using the alignment graph extension. As this percentage depends much on the used reference set (see Figures 3.13 and 3.14), structural information about the current set of sequences also needs to be considered. Such information can be obtained by analysing the size of the alignment graph with and without the extension. For sequences with high similarity, only few edges are added during the extension, while for sequences with lower similarity, the alignment graph grows exponentially with the level of extension. $\Delta = (|E| - |P|)/|P|$ indicates the increase of the edges of the extended alignment graph E compared with the edges of the pairwise alignment graph P.

Figure 3.19 shows the relation between $|R_E|/|E|$ and Δ for all $BAliBASE$ reference alignments.

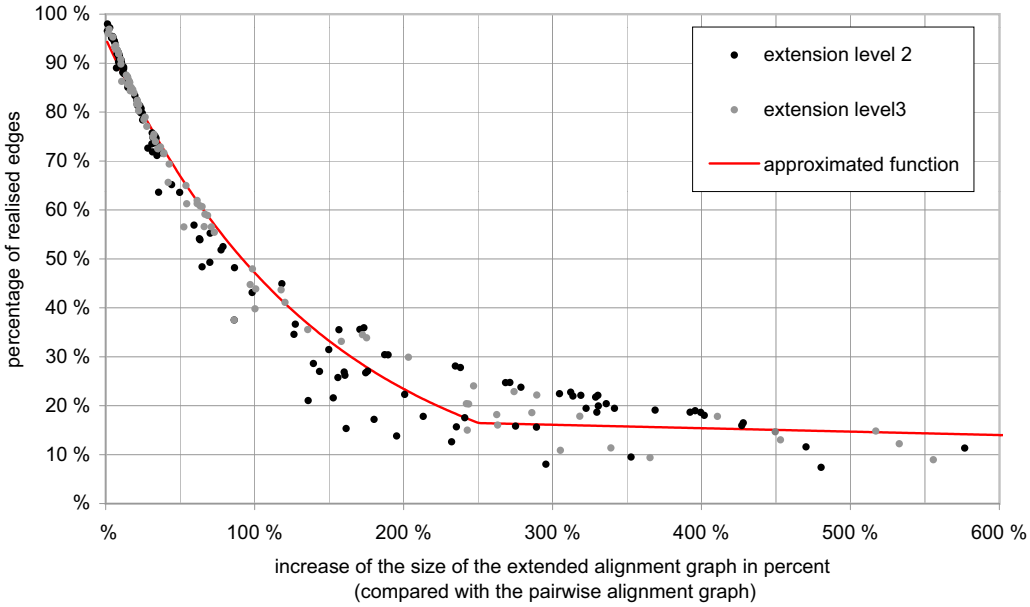


Figure 3.19: Relation between the percentage of realised edges $|R_E|/|E|$ and the increase of size of the alignment graph Δ in percent

The plotted values of Figure 3.19 are approximated by the following function:

$$f(x) = \begin{cases} 95 \cdot \mathbf{e}^{-0.007x} & \text{for } x \leq 250, \\ 18.2 - 0.007 \cdot x & \text{for } x > 250. \end{cases}$$

Applying these functions to our problem, we obtain the following estimation R_E^* for the approximative number of realised edges in an alignment graph with extension of level 3:

$$|R_E^*| = \begin{cases} 0.95 \cdot |E] \cdot \mathbf{e}^{-0.7 \cdot \Delta} & \text{for } \Delta \leq 2.5 \\ 0.01 \cdot |E| \cdot (18.2 - 0.7 \cdot \Delta) & \text{for } \Delta > 2.5 \end{cases}$$

Chapter 4

Our Evolutionary Algorithm

4.1 Preface

In this chapter we present a new evolutionary algorithm for the MSA problem which is able to compute multiple alignments based on the information of an alignment graph.

At first we give a characterisation of the traces of an alignment graph, as we extensively make use of the correspondence between multiple alignments and traces in our method. Then we introduce a heuristic which produces alignments of good quality in short time. Finally we present our evolutionary algorithm which further improves these alignments by applying a set of problem specific mutation and crossover operators.

Until otherwise defined, n refers to the number of sequences, l to the length of the longest sequence, and $\hat{l}$ to the length of the alignment throughout this chapter.

4.2 The Trace of an Alignment Graph

Based on a multiple alignment it is no problem to compute the corresponding trace and vice versa. This correspondence has already been described in the last chapter (see Figures 3.1 and 3.2).

So deriving a multiple alignment of an alignment graph leads to the problem of finding a feasible trace in this graph. Therefore we need a graph-theoretical characterisation of traces without considering the corresponding

multiple alignment. To this end we extend the alignment graph to a mixed graph [15] which includes edges and arcs and give a definition for a cycle in this mixed graph.

Definition 4.1 (Extended alignment graph) Let $G = (V, E)$ be a simple alignment graph and $H = \{(s_{i,p}, s_{i,p+1}) | 1 \leq i \leq n, 1 \leq p \leq l_i - 1\}$ be a set of directed edges with weight 0. Then these directed edges are called arcs and the mixed graph $\bar{G} = (V, E, H)$ is called extended alignment graph.

Definition 4.2 (Cycle) A path in a mixed graph $\bar{G}$ is a sequence of vertices $v_1, v_2, \ldots, v_n$, $n \geq 2$ such that either $(v_i, v_{i+1}) \in E$ or $(v_i, v_{i+1}) \in H$ for all i, $1 \leq i < n$. A path is called cycle, if the first and the last vertex on the path are the same.

Based on these definitions, we can give a graph-theoretical formulation for a trace:

Definition 4.3 (Trace) Let $\bar{G} = (V, E, H)$ be an extended alignment graph, $T \subseteq E$ and $\bar{G}^T = (V, T, H)$ the extended alignment graph induced by T. Then T is called a (feasible) trace if all existing cycles in $\bar{G}^T$ contain no arcs.

For examples of a feasible and an infeasible trace see Figures 4.1 and 4.2. Note that for explanation the arcs of the extended alignment graph are displayed in these examples too (dashed arrows).

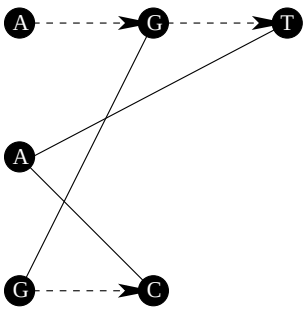


Figure 4.1: Feasible trace

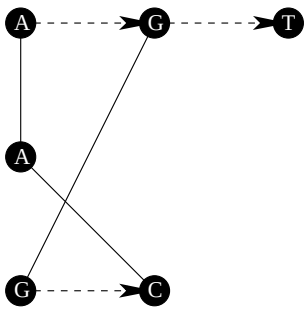


Figure 4.2: Infeasible Trace

We usually require that traces are locally optimal:

Definition 4.4 (Locally optimal) Let $\bar{G} = (V, E, H)$ be an extended alignment graph, a trace $T \subseteq E$ and $\bar{G}^T = (V, T, H)$ the extended alignment graph induced by T. Then the trace T is locally optimal if for all $e \in E \setminus T$ the following condition holds: $\bar{G}^{T'} = (V, T', H)$ with $T' = T \cup \{e\}$ contains at least one cycle with an arc.

Having a suboptimal trace, the corresponding multiple alignment can be ambiguous. This is also possible if not enough edges exist the alignment graph. For the sake of uniqueness we always create the shortest possible alignment then. An example is given in Figures 4.3 and 4.4.

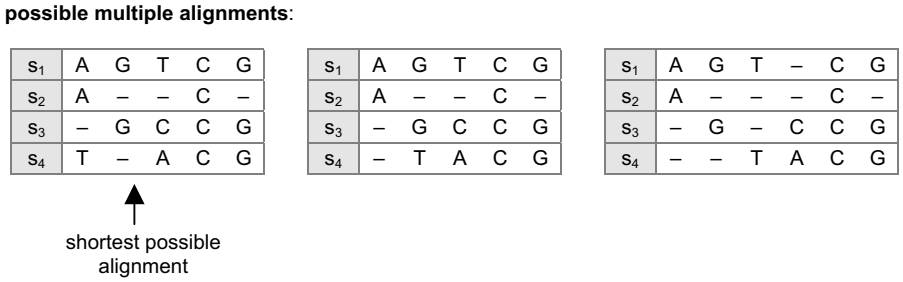


Figure 4.3: Possible multiple alignments to the trace of Figure 4.4

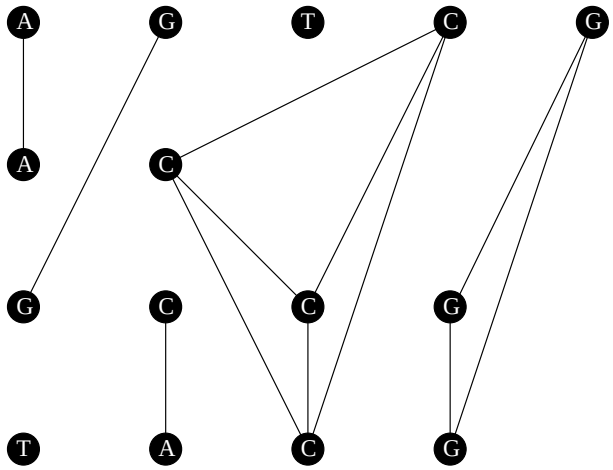


Figure 4.4: (Locally) suboptimal trace as edge T–T may be added.

Note that maximal $n \cdot l \cdot \frac{n-1}{2}$ edges can exist in a trace, i.e. from each character there is an edge to a character in each other sequence.

Finding the trace with maximum weight is known to be NP–hard (see section 3.1). Fauster [7] dealed with this problem by using two relatively fast greedy heuristics, local improvement strategies and tabu search. Although such methods can never guarantee optimality, they often provide reasonable solutions in short time. Besides, these results can be used as starting point for other optimisation strategies like evolutionary algorithms.

As one of these heuristics is also used in our methods, we will further describe it in the next section.

4.3 The Greedy2 Heuristic

The greedy2 heuristic is based on the following idea. All edges of the alignment graph are sorted by weight in descending order. One by one, these edges are tried to be realised by inserting them into the (originally empty) trace. If it is not possible to add the edge, the inconsistent edge is skipped and the process continues with the next edge. The set of all realised edges forms the trace.

Therefore, the main problem is to decide whether the current edge can be realised in the trace or not. Having a trace T and the induced extended alignment graph $\bar{G}^T = (V, T, H)$, an edge $e = (s_{i,p}, s_{j,q})$ can be inserted into the trace if one of the following conditions is fulfilled:

1. Any path in $\bar{G}^T$ between character $s_{i,p}$ and $s_{j,q}$ only consists of undirected edges and not of arcs, i.e. $s_{i,p}$ and $s_{j,q}$ are already aligned.

2. There exists no path from character $s_{i,p}$ to chararcter $s_{j,q}$, respectively from character $s_{j,q}$ to character $s_{i,p}$ with at least one arc.

Hence, it is necessary to search efficiently for all paths between two characters $s_{i,p}$ and $s_{j,q}$. In the following, only the principles for the search from character $s_{i,p}$ to character $s_{j,q}$ are described, the same principles hold for the search in the other direction:

1. Any path from character $s_{i,p}$ to character $s_{k,r}$ also implies a path from character $s_{i,p}$ to the characters $s_{k,r+1}, s_{k,r+2}, \ldots, s_{k,s_{|k|}}$, as these characters are connected by arcs.

2. No sequence needs to be considered more than once, i.e. once a sequence is left, it is not necessary to search on this sequence later again, as such a path can be reproduced by using an abbreviation over arcs. Hence only paths containing maximal $n-1$ edges of T need to be considered. The number of arcs of H in the path is not limited by n.

3. If $(s_{i,p^*}, s_{k,r})$ is the edge with minimal $p^* \geq p$ between sequence S_i and sequence S_k, there cannot exist an edge $e = (s_{i,p^+}, s_{k,r^-})$ with $p^+ > p^*$ and $r^- \leq r$.

Based on these principles, a breadth–first search is performed to determine the minimal positions in all sequences, which can be reached from character $s_{i,p}$. If the minimal position q^* in Sequence S_j is greater than q, there exists no path from character $s_{i,p}$ to character $s_{j,q}$ and the edge can be added to the trace. For further details see the description of Fauster [7].

In the worst case this breadth–first search takes time $O(n^3)$ for one character, so the overall complexity of the greedy2 heuristic is $O(|E| \cdot \log |E| + |E| \cdot n^3 \cdot \log l)$, with $|E| \cdot \log |E|$ being the time to sort all edges by weight.

In practice, the greedy2 heuristic usually performs much better. Tests indicate, that the overall complexity is $O(|E| \cdot \log |E| + |E| \cdot n^2 \cdot \log l)$ or even $O(|E| \cdot \log |E| + |E| \cdot n \cdot \log n \cdot \log l)$.

Fauster also presented a second version of this greedy2 heuristic with an advanced trace structure, which supersedes the time consuming breadth first search but requires to update the trace after any insertion. The overall complexity of this advanced greedy2 heuristic is bounded by $O(|E| \cdot \log |E| + |T| \cdot n^2 \cdot \log l + |T| \cdot \log l)$, with $|T| \leq n \cdot l \cdot \frac{n-1}{2}$.

4.4 Our Evolutionary Algorithm

As pointed out before, multiple sequence alignment and maximum weight trace are related. Therefore, it is also possible to use the trace data structure in a chromosome of an evolutionary algorithm to represent the multiple alignment. The task of the EA is to maximise the weight of the trace then. Such optimisation can be performed by realising as many edges of high weight as possible or by replacing edges of low weight with edges of high weight.

Compared with the used heuristics of existing EAs for the MSA problem

(see chapter 2), these methods have the advantage of using global information as reflected by the edges of the alignment graph for the optimisation instead of just local considerations like e.g. a fully matched column.

On the downside all these strategies have a certain amount of complexity, as it is always necessary to keep the trace feasible, i.e. no cycles containing arcs are allowed in the trace. So, methods which combine existing patterns or (randomly) create new patterns in the chromosomes have a rather slow performance compared to existing crossover and mutation operators.

To overcome this problem, we generally use a two-dimensional array to represent the multiple alignment of our chromosomes, but for some kind of mutation and crossover operators we additionally make use of the corresponding trace data structure. These operators proceed as follows:

1. First they use the multiple alignment of the chromosome to compute the corresponding trace.

2. Then they perform the modifications on the trace data structure, while the multiple alignment of the chromosome remains unchanged, it is only used for reference (e.g. to get the position of a character in the alignment).

3. Finally they compute the multiple alignment of the chromosome based on the modified trace.

Hence it is not necessary to store the trace in a chromosome, it is only computed if required. Thus, we always have to face the additional complexity of the transformation operations – $O(n^2 \cdot l)$ respectively $O(|T|) \leq O(n^2 \cdot l)$ – when applying these mutation and crossover operators.

Parallelising an evolutionary algorithm helps to improve the performance of the EA, as it usually preserves diversity in the population and allows to emphasise different characteristics in the chromosomes. So some of the existing EAs for MSA (e.g. *PGA* [3]) also made use of these advantages of a parallel implemention.

We follow this approach by using the so–called island model for our evolutionary algorithm. The population is divided in several subpopulations (islands), each consisting of a certain number of chromosomes. In each of these subpopulations the chromosomes evolve separately during the evolutionary process. Only from time to time (based on a provided probability)

the best chromosomes between these subpopulations are exchanged (migration).

The chromosomes of these subpopulations are modified by fast mutation and crossover operators, which directly work on the multiple alignment. The main tasks of these operators are

- to combine patterns of different chromosomes to form new chromosomes,
- to perform small (random) modifications to introduce new patterns in the population.

Additionally these operators are biased to favour patterns with good fitness by using the information of the alignment graph. These operators are One Point Crossover (see section 4.3.2), Best Consistent Cut Crossover (see section 4.3.3) and Block Shuffling Mutation (see section 4.3.4).

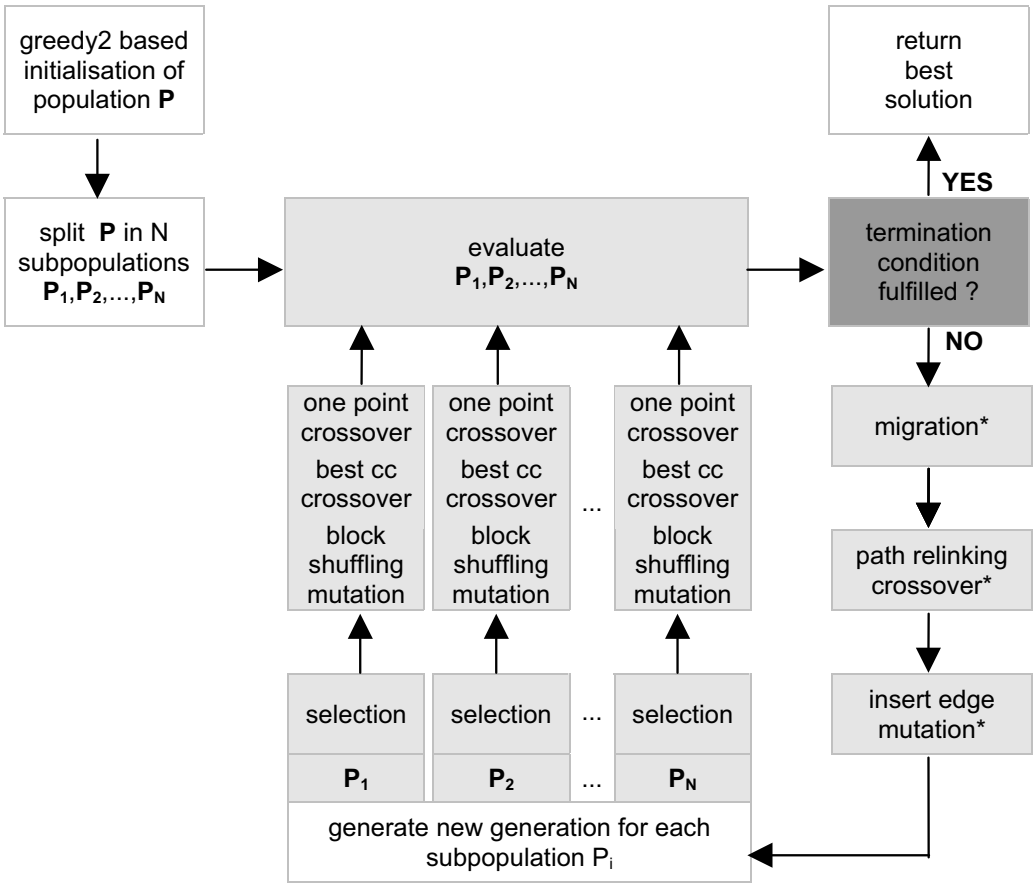
To include global optimisation, we developed two heuristic operators, which perform modifications on the trace data structure. These heuristics are Path Relinking Crossover (see section 4.3.5) and Edge Insertion Mutation (see section 4.3.6). Based on their complexity and the required transformation operations, these heuristics are not applied every generation. Like the migration, they have a probability of being selected and applied to the best chromosomes of each subpopulation.

For initialisation, a modified version of the greedy2 heuristic ist used, which allows to create diversity in the initial population. This method is presented in section 4.4.1

The fitness score is simply the sum–of–weights of all the edges, for which a corresponding pairwise character alignment exists in the multiple alignment.

Various parameters allow to make the EA adjustable (e.g. population size, termination condition, migration probability), an overview can be found in section 5.4.

The whole structure of our evolutionary algorithm is given in Figure 4.5.



* only applied to the best chromosomes of the subpopulations after a certain number of generations (based on a provided probability)

Figure 4.5: Structure of our evolutionary algorithm

4.4.1 Greedy2 Based Initialisation

As pointed out before, we use a modified version of the (advanced) greedy2 heuristic to create the initial population. Only the first chromosome is created as described in section 4.2. For all other chromosomes, the following modifications are used:

1. Only the best $|D|$ edges of E are tried to be inserted into the trace, with $D \subseteq E$ and $|D|$ being a random number between $n \cdot l \cdot \frac{1}{2}$ and $n \cdot l \cdot \frac{n-1}{2}$. Following the results of various tests, these boundaries seem to be a reasonable compromise between alignment quality and time complexity.

2. D is not sorted by weight anymore, it is randomly permutated for each chromosome.

3. Edges, which could not be realised in the previous chromosome, are moved to the top of D.

Using the advanced trace structure of the greedy2 heuristic, the worst case time complexity for the first chromosome is

$$O(|E| \cdot \log |E| + n^4 \cdot l \cdot \log l + n^2 \cdot l \cdot \log l).$$

For all other chromosomes, the initialisation takes time

$$O(n^2 \cdot l + n^4 \cdot l \cdot \log l + n^2 \cdot l \cdot \log l)$$

in the worst case (with the first term being the time to permute D randomly and to shift the appropriate edges to the top of D).

Note that the obtained traces are often not locally optimal (if not enough edges exist in D). Therefore some additional randomness is introduced to each chromosome when the corresponding alignment to the given trace is computed.

4.4.2 One Point Crossover

The One Point Crossover operator has already been presented in section 2.2.2. As it can be implemented very efficiently and was used in most of the existing EAs for the MSA problem too, this operator has also been chosen for our evolutionary algorithm.

One character of the first chromosome is randomly selected and the chromosome is cut straight at this position. The second chromosome is tailored so that the right and the left parts of each chromosome can be joined together while keeping the original sequence. Any void space that appears at the junction point is filled with gaps.

For simplicity, we make the following assumption unless otherwise defined:

- The character of each sequence, which preceeds the cutting position, is called boundary character.
- Right part refers to all characters, which are placed to the right of the boundary characters. If there is no boundary character in the sequence, all the characters of this sequence belong to the right part.
- Left part refers to all characters, which do not belong to the right part. Therefore, the left part contains the boundary characters.

Having two multiple alignments, the single operations of One Point Crossover proceed as follows:

1. Randomly select a sequence of the first parent and one of its characters.
2. Determine the boundary characters in the first parent and copy the left part to the offspring.
3. Find the boundary characters in the second parent and copy the right part to the offspring.

So the entire algorithm takes time $O(n \cdot \hat{l})$ with usually $n \ll \hat{l}$.

As this operator starts with an empty offspring, any void space at the junction point is implicitly filled with gaps. For an example see Figure 2.3 in section 2.2.2.

One Point Crossover can be very disruptive around the junction point. To avoid this drawback, we added a second crossover operator, Best Consistent Cut Crossover.

4.4.3 Best Consistent Cut Crossover

The definition of two consistent columns as given in section 2.2.2 is very restrictive, as it requires that both columns contain exactly the same characters

by reference to the original sequence (or a gap between the same characters). Therefore it is often difficult to find such columns for crossover operators like Uniform Crossover.

We ease the requirements by defining a so–called consistent cut: Having two chromosomes and cutting the first chromosome straight at some position, the cut is called consistent, if no boundary character is aligned with a character of the right part of the second chromosome.

An example for a consistent cut is given in Figure 4.6.

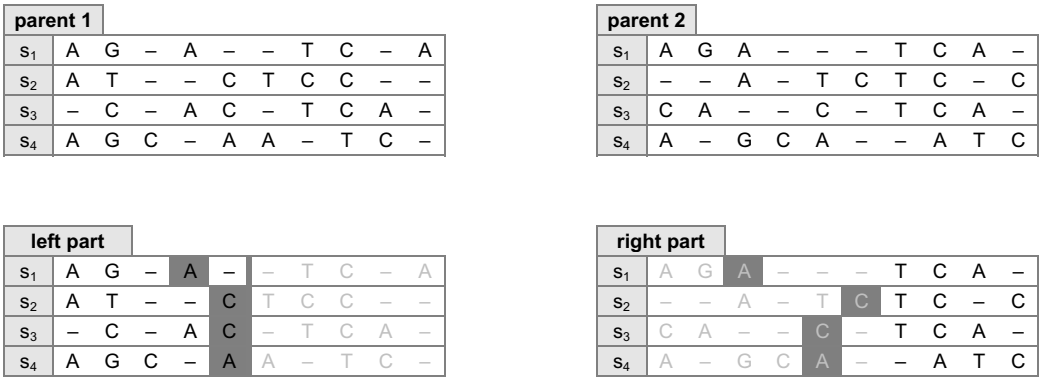


Figure 4.6: Example for a consistent cut

The Best Consistent Cut Crossover is a crossover heuristic, which determines the best consistent cut between two chromosomes A and B by

- computing all possible cuts,
- selecting the cuts which are consistent,
- computing the fitness of all offsprings created by a consistent cut.

This heuristic can be implemented very efficiently, as it is possible to compute the cuts incrementally, i.e. based on chromosome C_{i-1} created by the cut after column $(i-1)$, it is possible to compute C_i created by the consecutive cut after column i.

Compared with the implementation of One Point Crossover, it is therefore not necessary to consider the whole left part of the first chromosome and the whole right part of the second chromosome during the computation, it

is enough to consider the boundary characters of the current cut and shift them accordingly.

Figure 4.7 demontrates the modifications between two consecutive cuts. Note that the cut in offspring C_6 is not consistent, as two columns in the right part are affected by the movement of the boundary characters.

For a consistent cut, the following simple criterion must hold: Any column containing boundary characters, which need to be shifted during the computation of the cut, has to be empty after the shift.

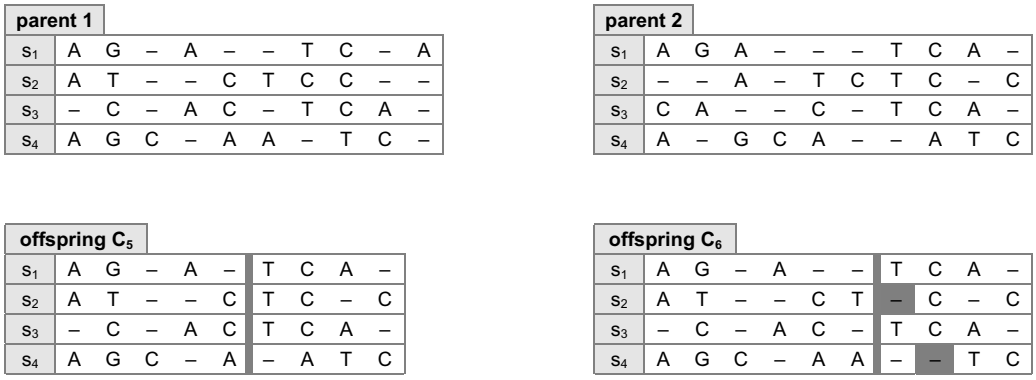


Figure 4.7: Example for the modifications between two consecutive cuts

Let $w_A(j)$ denote the sum–of–weights of all realised edges in column j of chromosome A and $w_B(k)$ denote the same for chromosome B. The fitness f_{C_i} of a chromosome C_i, which is created based on a consistent cut after column i, can be calculated as follows:

$$f_{C_i} = \sum_{j=1}^{i} w_A(j) + \sum_{k=i+1}^{\hat{l}_B} w_B(k)$$

Hence, precomputing $w_A(j)$ for all columns j, $1 \leq j \leq \hat{l}_A$ respectively $w_B(k)$ for all columns k, $1 \leq k \leq \hat{l}_B$, allows to evaluate C_i in time $O(\hat{l})$.

Having two parents A and B and the offsprings $C_i, 0 \leq i \leq \hat{l}_A$, the entire algorithm proceeds as follows:

1. Precompute $w_A(j)$ and $w_B(k)$ for all columns.
2. $C_0 = B$.

3. $i = 1$.

4. Compute C_i.

   - $C_i = C_{i-1}$.
   - $f_{C_i} = 0$.
   - Determine the boundary characters in A based on a cut after column i.
   - Find these characters in C_i and shift them to the position given by A.
   - Test if the cut after column i is consistent.
   - If the cut is consistent, calculate f_{C_i}.

5. If $f_{C_i} > f_{C_{max}}$ update C_{max}.

6. $i = i + 1$.

7. Iterate step 4 to 6 until $i = \hat{l}_A - 1$.

8. Return C_{max}.

As it takes time $O(n^2 \cdot \hat{l})$ to precompute $w_A(j)$ respectively $w_B(j)$ and $O(\hat{l} \cdot n)$ to compute C_i based on C_{i-1}, the entire algorithm takes time $O(n^2 \cdot \hat{l} + \hat{l}^2 \cdot n)$ with usually $n \ll \hat{l}$. If no consistent cut can be found between the two chromosomes, One Point Crossover is performed.

4.4.4 Block Shuffling Mutation

Random shuffle operators as described in section 2.2.3 provide the possibility to perform small modifications on the chromosome in short time. As such operators can also be adapted to consider the information of the alignment graph for movement, they have been applied to our evolutionary algorithm, too.

A block of consecutive characters (bounded by a gap or the begin/end of the sequence on the left and the right side) in one sequence is randomly selected. Then this block is moved to a left or right position either randomly or in a heuristic way, looking for the position with the best fitness.

The single operations of this algorithm perform as follows:

1. Randomly select a character.

2. Determine the block of characters of length b and the number of gaps g to the left and the right of the block.

3. Randomly select whether the whole block or just a part (the left or the right part) is moved.

4. Select the new position of the block

   - randomly.
   - in a heuristic way.

5. Move the block.

If the new position of the block is randomly selected, the entire algorithm takes time $O(2 \cdot b + g)$ with $b + g < \hat{l}$ and usually $n \ll \hat{l}$. If this selection is performed in a heuristic way, the entire algorithm takes time $O(2 \cdot b + g + b \cdot g \cdot n)$, as any character of the block of length b can be moved to g positions and needs to be compared with maximal $(n-1)$ characters of the other sequences.

To avoid huge complexity, the heuristic movement is only performed for blocks of small size ($b < 10$) and a limited number of gaps around the block ($g < 10$). An example is given in Figure 4.8.

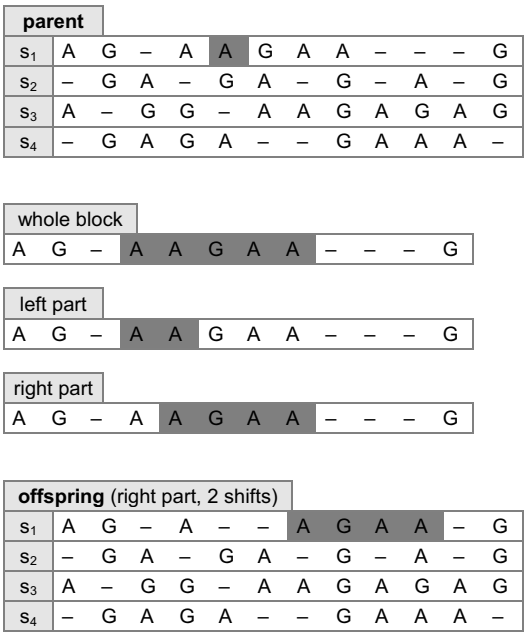


Figure 4.8: Block Shuffling Mutation

4.4.5 Path Relinking Crossover

The main idea of path relinking [9] is to trace a path between two possible solutions in the search space. On this path, all the interim solutions are incrementally created by small modifications. Consequently, these interim solutions can be evaluated incrementally too. Finally the best interim solution is kept.

Applying this approach to our evolutionary algorithm, a plausible path relinking operator for the traces of our chromosomes works as follows (note that we have to compute the corresponding traces to our chromosomes before):

Definition 4.5 Let A and B be the locally optimal traces of two possible solutions and I_i be the trace of one of the interim solutions $I_0, I_1, \ldots, I_N$ on the search path from A to B. Then for any trace $I_i, (0 \leq i \leq N)$ the following conditions are fulfilled:

- $I_0 = A$, $I_N = B$.
- $A \cap B \subseteq I_i$.
- $B \setminus I_i \subseteq B \setminus I_{i-1}$, i.e. once an edge of B is realised, it cannot be removed anymore.
- $A \setminus I_{i-1} \subseteq A \setminus I_i$, i.e. once an edge is removed from A, it cannot be added anymore.
- Any trace of an interim solution I_i is locally optimal, i.e. no other edge of the alignment graph can be inserted into I_i without removing an existing one. For our path relinking crossover operator we use the greedy2 heuristic to optimise the interim solutions during the path relinking process.

Having a trace I_{i-1}, the main question is how to compute the consecutive trace I_i. In our approach, we start with trace A and try to realise trace B column by column from left to right, i.e. we use the corresponding multiple alignment of trace B to select the appropriate edges (=the edges, which are realised by the multiple alignment in column i) and add them to I_i. At the same time, we have to remove edges from I_i to keep the trace feasible.

For all these operations, f_{I_i} (the fitness score of I_i) is simultaneously updated.

An example for Path Relinking Crossover is given in Figures 4.9 to 4.11. Note that edges, which are inserted by the greedy2 heuristic, are displayed with dashed lines in the corresponding step. For explanation, the corresponding multiple alignments to each interim solution I_i are shown too.

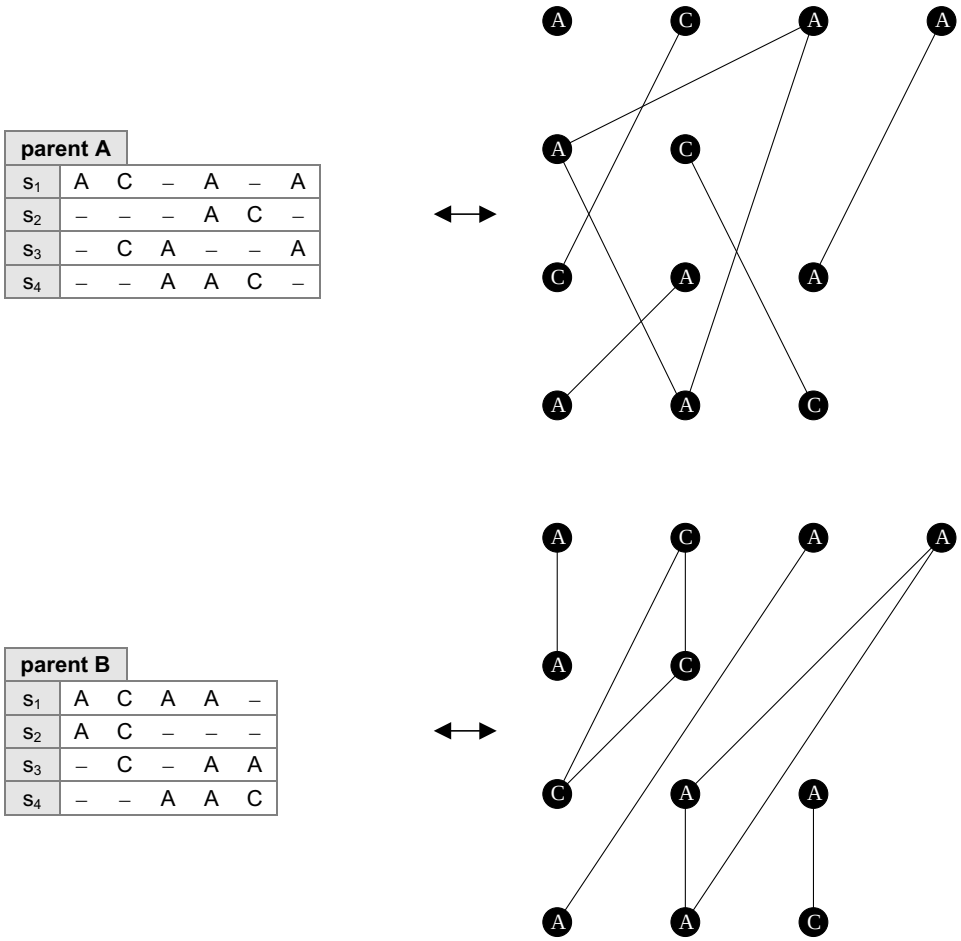


Figure 4.9: Path Relinking Crossover (part A): parent chromosomes

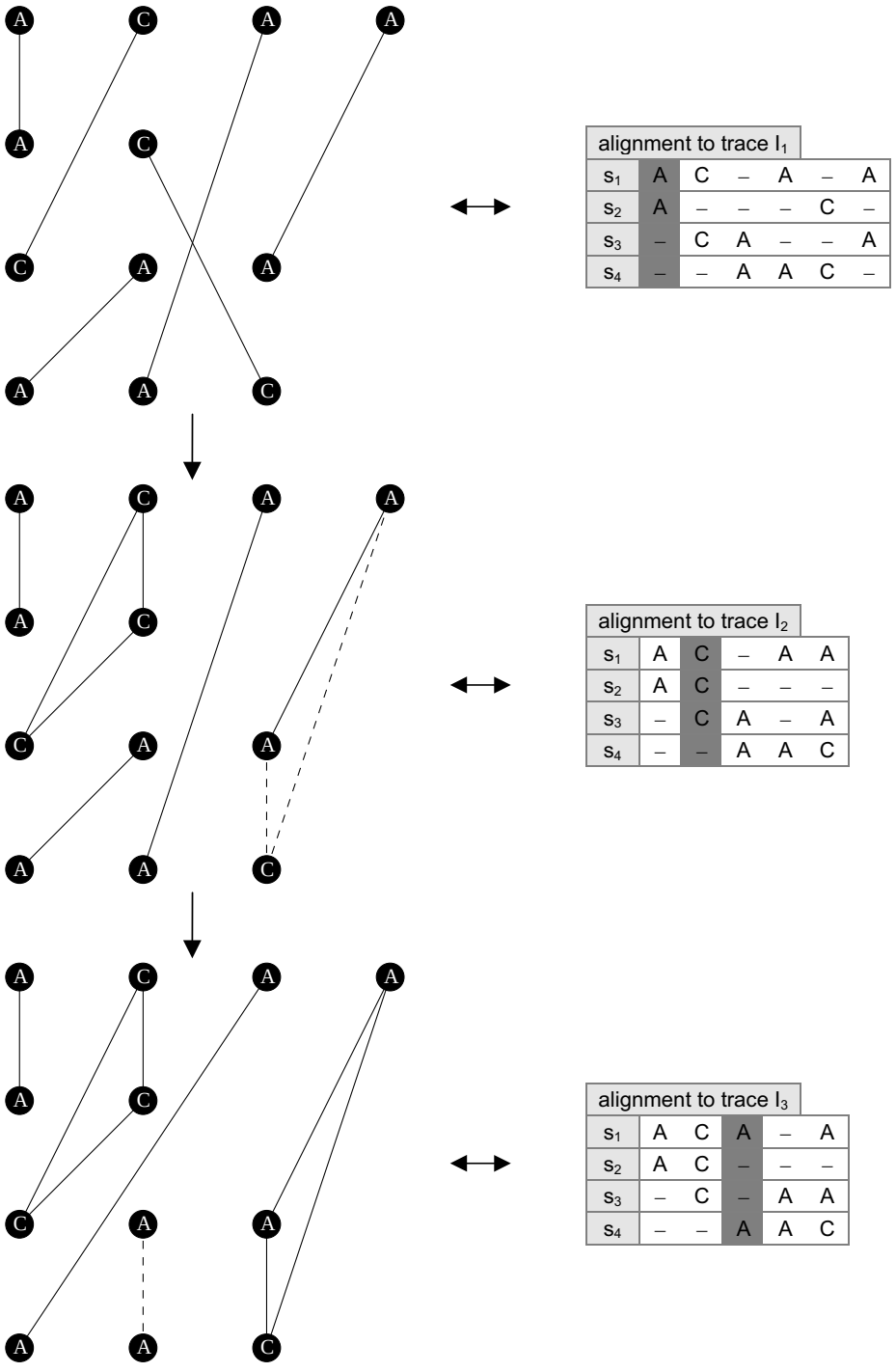


Figure 4.10: Path Relinking Crossover (part B): interim solutions

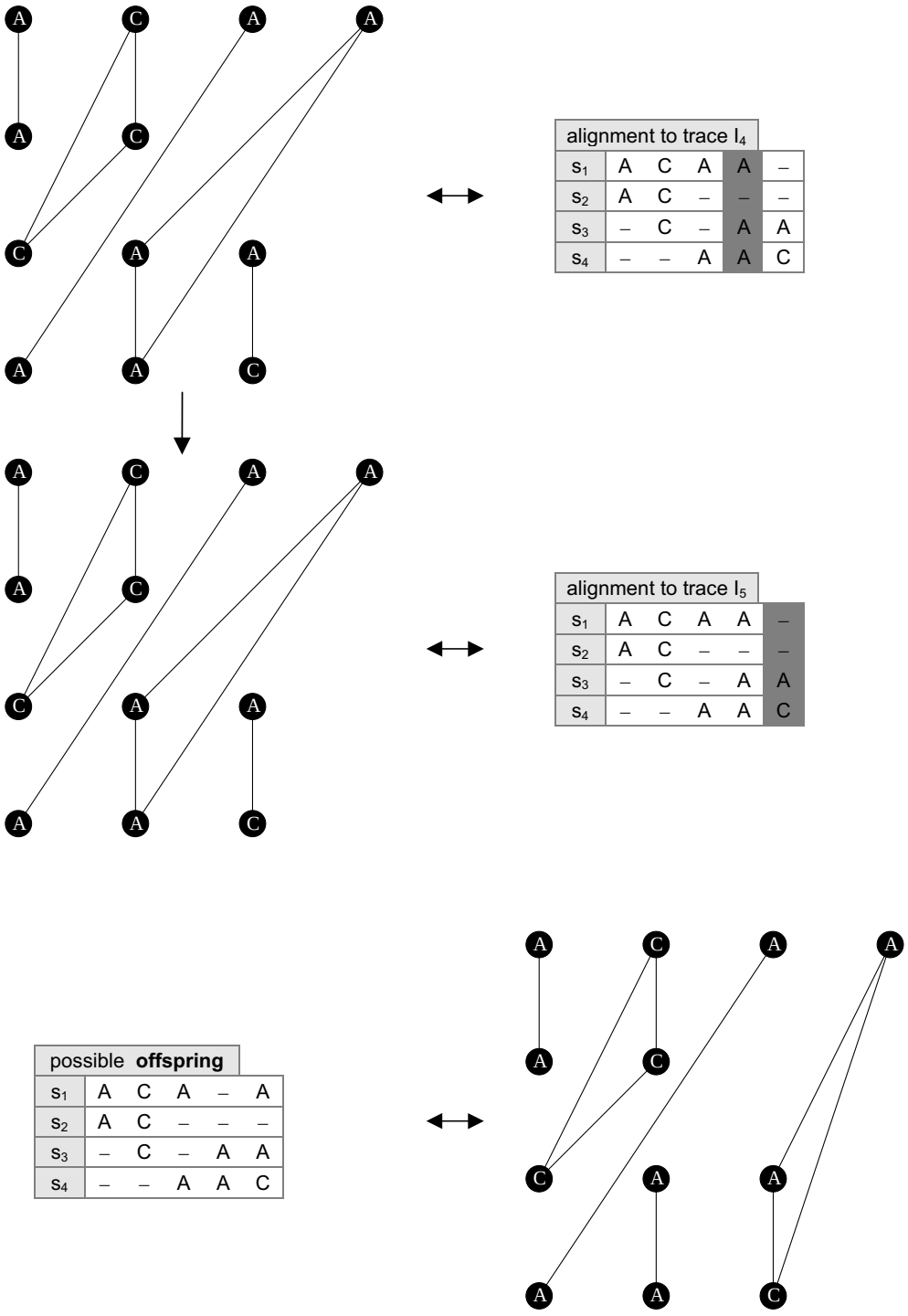


Figure 4.11: Path Relinking Crossover (part C): interim solutions and possible offspring

Let $G = (V, E)$ be the alignment graph, X_i the set of characters, which are placed in column i in the corresponding multiple alignment of trace B, and $Y_i = V \setminus \sum_{j=1}^{i} X_j$ the set of all characters, which are placed to the right of column i in the corresponding multiple alignment of trace B. Furthermore, let $Y_i^+ \subseteq Y_i$ denote the subset of all characters of Y_i, which are connected with a character of X_i by an edge of trace I_{i-1}.

Then the entire path relinking algorithm proceeds as follows:

1. $I_0 = A$.
2. $i = 1$.
3. Compute I_i.
   - $I_i = I_{i-1}$.
   - $f_{I_i} = f_{I_{i-1}}$, simultaneously update f_{I_i} in the following.
   - Step 1: Remove all edges of I_i, which connect a character of X_i with a character of Y_i^+.
   - Step 2: Insert all edges in I_i, which connect two characters of X_i (if these edges do not already exist). Note that this is always possible now, the trace remains feasible.
   - Step 3: Try to add all (not already existing) edges of E to I_i which connect a character of Y_i^+ with a character of Y_i to keep the trace locally optimal. We use the same method as described in the greedy2 heuristic to insert these edges.
4. If $f_{I_i} > f_{I_{max}}$ update $I_{\max}$.
5. $i = i + 1$.
6. Iterate step 3 to 5 until $i = \hat{l}_B - 1$.
7. Return I_{max}.

Additionally we have to perform three transformation operations – $O(n^2 \cdot l)$ – to compute trace A and B based on the corresponding parent alignments respectively the offspring alignment based on trace I_{max}.

As $|X_i| \leq n$ and $|Y_i^+| \leq n \cdot \frac{n-1}{2}$ hold, step 1 and step 2 can both be performed in time $O(n^2)$.

They are performed $\hat{l}$ times during one (entire) Path Relinking Crossover, so the overall complexity of these both steps is $O(\hat{l} \cdot n^2)$ in worst case.

To estimate the complexity of step 3, we have to face that any character of Y_i^+ can theoretically be connected with $(n-1) \cdot l$ other characters of Y_i. So, in the worst case, $n \cdot \frac{n-1}{2} \cdot (n-1) \cdot l$ edges need to be considered for the greedy2 heuristic.

However, there are usually much fewer edges in practice:

- Firstly, hardly ever all of the theoretically possible edges between characters of Y_i^+ and Y_i exist in E. Especially for test instances with higher sequence similarity, $|E|$ is rather small.
- Secondly, the size of Y_i decreases during Path Relinking Crossover (according to the increasing value of i).
- Finally, the more similar the parent chromosomes are, the fewer modifications are performed during the crossover. Hence, during the early steps of our evolutionary algorithm Y_i^+ is usually greater as more differences between the two chromosomes exist.

We have run our evolutionary algorithm on six different test alignments of the *BAliBASE* library with standard parameters (see section 5.4) and have analysed the average number of edges, which are inserted during one (entire) Path Relinking Crossover. These test instances differ in many features like the number of sequences, the length of the sequences and the sequence similarity.

The results are shown in Table 5.1.

	n	l	$n \cdot l$	$n^2 \cdot l$	avg
Reference 1a: 1tvxA	4	69	276	1104	298
Reference 1a: 1cpt	4	434	1736	6944	2014
Reference 1c: 1doxc	4	96	384	1536	65
Reference 1c: 1ad3	4	447	1788	7152	131
Reference 2: 1aboA	16	80	1280	20480	4108
Reference 2: 1ped	19	388	7220	137180	72812

Table 5.1: Average number of edges inserted by the greedy2 heuristic

Although the test instances are only a small sample and can not be seen as representative for all reference alignments, they still confirm our considerations from above and indicate that generally fewer than $n^2 \cdot l$ edges are inserted on average by the greedy2 heuristic.

So we obtain the following complexity for step 3 during one (entire) Path Relinking Crossover: $O(n^2 \cdot l \cdot \log(n^2 \cdot l) + n^5 \cdot l \cdot \log l)$ (also note the comments in section 4.3 to the complexity of the greedy2 heuristic in practice).

Therefore, in the worst case, the overall complexity of Path Relinking Crossover is $O(n^2 \cdot l + \hat{l} \cdot n^2 + n^2 \cdot l \cdot \log(n^2 \cdot l) + n^5 \cdot l \cdot \log l)$ with usually $n \ll l \leq \hat{l}$.

4.4.6 Insert Edge Mutation

The Insert Edge Mutation operator is based on the idea to select an edge with high weight and to realise it in the chromosome.

The selection of the edge is performed by the edge–selection strategy as presented in section 3.6. All edges of the alignment graph are sorted by weight in descending order and based on the provided formula a random rank for the edge selection is computed. If the edge with this obtained rank already exists in the chromosome, the edge with the next lower rank is examined. This process is iterated until an edge not yet in the chromosome is found.

When adding an edge to a trace, it is usually necessary to remove one or more edges before – preferably a set of edges with small overall weight – to keep the trace feasible. By removing these edges, however, it may be possible to add some other (new) edges to the trace. Therefore the selection of an adequate set of edges is difficult.

We try to overcome this problem by applying the greedy2 heuristic to a subalignment of the chromosome. An example for Insert Edge Mutation is given in Figures 4.12 and 4.13.

Let $e = (s_{i,p}, s_{j,q})$ be the selected edge with p_l being the position of $s_{i,p}$ and p_r being the position of $s_{j,q}$ in the corresponding alignment of the chromosome. Let us assume $p_l < p_r$, then $b = p_r - p_l + 1$ denotes the length of the subalignment. Let Y_l be the set of all characters which are placed to the left of column p_l in the alignment, Y_r the set of characters which are placed to the right of column p_r, and X the set of all characters, which are placed within the subalignment (including characters of columns p_l and p_r).

Then Insert Edge Mutation proceeds as follows:

1. Select edge e (as described above) and determine the position of the corresponding characters in the parent alignment.
2. Add all edges of the parent alignment to the trace, which connect two characters of Y_l or two characters of Y_r. Thus, the transformation operation (multiple alignment $\rightarrow$ trace) only needs to be performed for a part of the alignment.
3. Insert edge e into the trace. Note that this is always possible, the trace keeps feasible.
4. Put all edges of the parent alignment which connect two characters of X into a set D^+.
5. Put all edges of the alignment graph which are incident to a character of X and not already exist in the trace into a set D^-.
6. Sort D^+ and D^- by weight in descending order.
7. Try to add the edges of D^+ to the offspring. Then try the same for the edges of D^-. These operations are performed by the same methods as described in the greedy2 heuristic.
8. Compute the offspring alignment based on the trace.

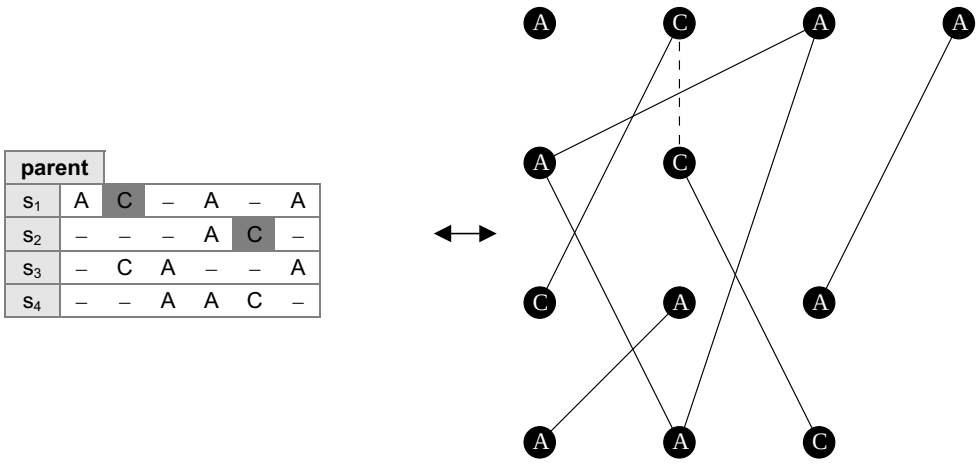


Figure 4.12: Insert Edge Mutation (part A): parent chromosome and edge to be included (dashed line)

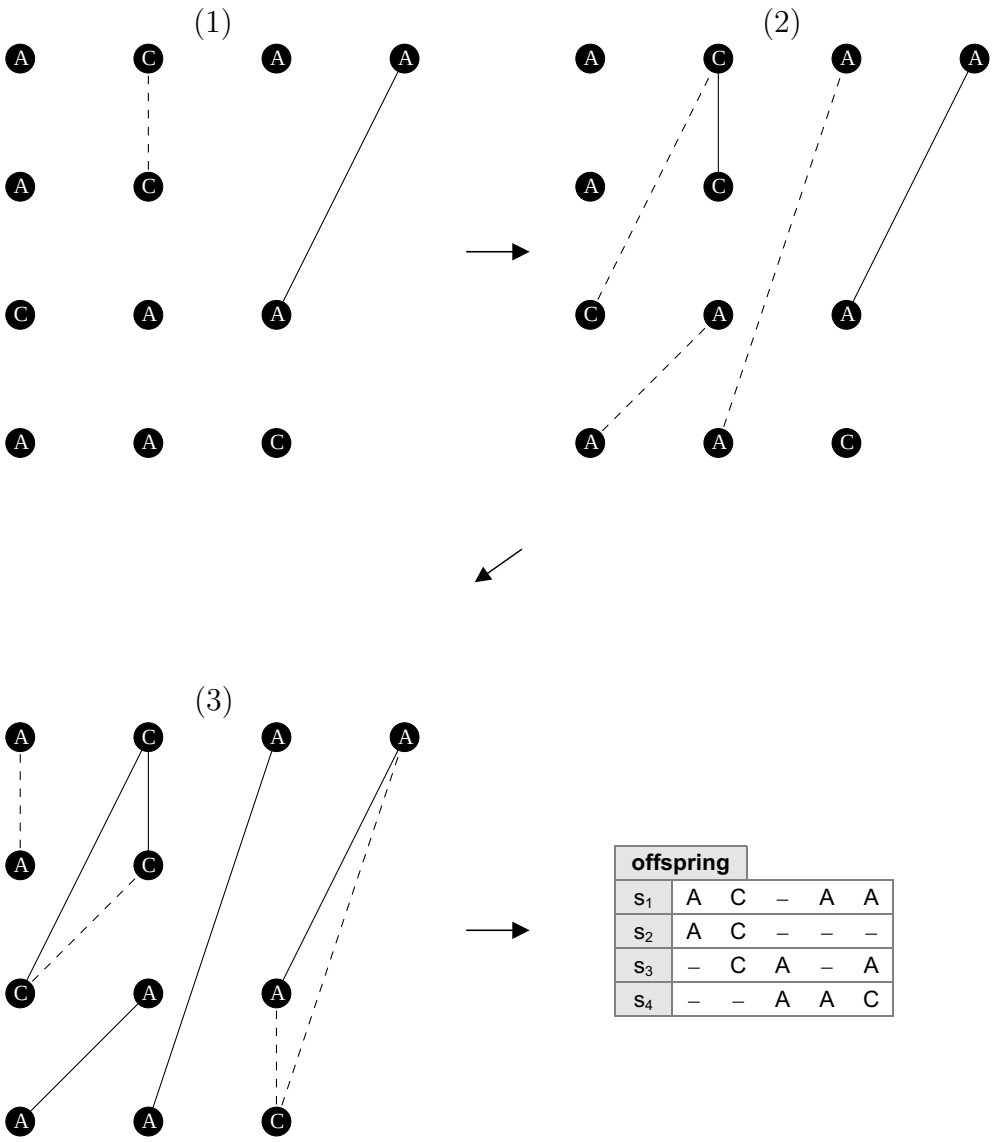


Figure 4.13: Insert Edge Mutation (part B): (1) empty trace, adopt all edges (outside block) of parent, insert new edge → (2) add all possible edges which exist in parent → (3) add all other possible edges

Note that the trace in Figure 4.12 is only shown for explanation, it is not actually computed by our algorithm.

To avoid huge complexity, we require $(|D^+| + |D^-|) \leq n \cdot b \cdot \frac{n-1}{2}$ by selecting only a limited number of edges in step 5.

Thus, in the worst case (also note the comments in section 4.3 to the complexity of the greedy2 heuristic in practice), steps 6 and 7 can be performed in time $O(n^2 \cdot b \cdot \log(n^2 \cdot b) + n^2 \cdot b \cdot n^3 \cdot \log l)$.

Considering the time for the transformation operations – $O(n^2 \cdot l)$ – the overall complexity of Insert Edge Mutation is $O(n^2 \cdot b \cdot \log(n^2 \cdot b) + n^5 \cdot b \cdot \log l + n^2 \cdot l)$ in the worst case with $b < l$ and usually $n \ll l$.

Chapter 5

Implementation Details

5.1 Used Tools

All algorithms were written in *C++* and were run on a Pentium 4 with 1.9 GHz and 1 GB RAM under a SuSE Linux 8.1 platform using the g++ 3.2 compiler from the GNU Compiler Collection.

For the implemenation of the evolutionary algorithm, the *C++* library *EAlib 1.1* [24] was used. This library consists of an evolutionary computation framework, which provides a toolkit for the fast implementation of evolutionary algorithms. The object orientated architecture makes it highly extensible and easily adaptable to different optimisation problems.

Additionally the *LEDA 4.4* [1] program package, which provides a sizable collection of data types and algorithms was included.

For the documention of the source code, *DoxyGen 1.2.17* [12] was used.

5.2 Classes and Modules

Following the paradigms of object–orientated programming, all implemented methods were encapsulated in classes and modules, each providing various input, output and access operators. A short description of the integral classes of our evolutionary algorithm is given in the following.

ClustalW: This class represents a low–level interface to *CLUSTAL W*, which allows to load and save alignments in different file formats, to have access to *CLUSTAL W*'s scoring matrices and to compute pairwise alignments.

Sequence: This class encapsulates a single sequence without gaps, which is implemented as a string.

MultiSequences: This class contains a set of sequences, which is implemented as an array of type *Sequence*.

MultiAlignment: This class represents an alignment of an associated set of sequences. It contains a reference to a *MultiSequences* object and is implemented as a two–dimensional array. The implemented methods allow to find and move blocks of consecutive characters, to randomly create an alignment and to check the feasibility of the alignment.

AGEdge: This class represents an edge of the alignment graph. It stores the two sequences, the positions of both incident characters and an assigned weight.

AGEdgeList: This class is a space–efficient container for all edges of the alignment graph, implemented as doubly linked list of type *AGEdge*.

AlignmentGraph: This class represents the corresponding alignment graph to a given set of sequences. It provides methods for adding new or deleting existing edges, for assigning a weight to each edge or for performing the aligment graph extension. It also provides the method to compare the alignment graph with a given alignment, so that characteristics like the average number of realised edges can be analysed. The class contains a reference to the associated *MultiSequences* and *AGEdgeList* objects and is implemented as a two–dimensional array of type *AGNode*.

Each element of type *AGNode* represents one character and contains a list of all incident edges of type *AGEdge*. These lists are implemented as sorted sequences, which are realised by skip lists.

MATrace: This class represents a (feasible) trace for a given alignment graph. It provides the methods to determine, whether an edge can be added to the trace or not, and allows to insert and remove edges. It contains a reference to the associated *AlignmentGraph* object and is implemented as follows: For any two sequences, a list of all edges connecting these sequences is stored. As these lists are realised by sorted sequences, the trace is implemented as a two–dimensional array of sorted sequences.

MultiAlignmentChrom: This object is derived from the class *chromosome* of the *EAlib*. It contains a *MultiAlignment* object representing the multiple alignment of the chromosome and a reference to the associated *AlignmentGraph* object. The methods allow to initialise and evaluate the chromosome. All the described mutation and crossover operators of our evolutionary algorithm are also implemented in this class.

MultiAlignmentIsland: This object is derived from the class *islandModelEA* of the *EAlib*. It implements the migration strategy for our EA and calls Insert Edge Mutation and Path Relinking Crossover based on a given probability.

5.3 Program Flow

The program flow of our evolutionary algorithm for the MSA problem is given in Figure 5.1.

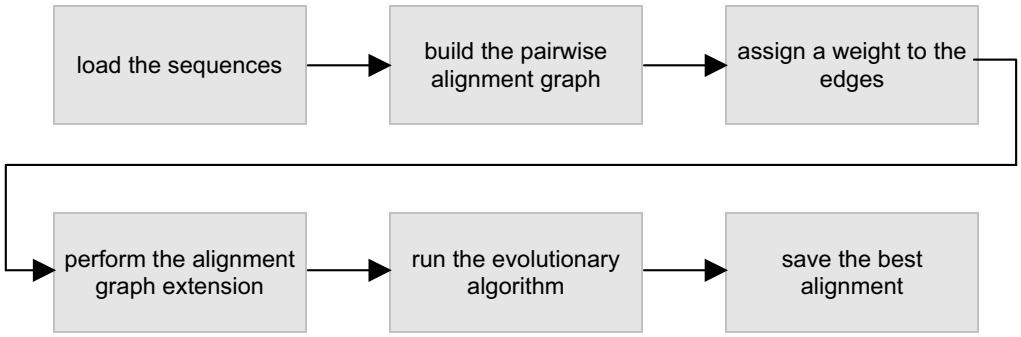


Figure 5.1: Program flow

5.4 Parameters

Various parameters allow to make our evolutionary algorithm adjustable. A short overview of the integral parameters with their default values is given in Table 5.1.

parameter	default	description
edgew	7*	Weighting strategy used to assign a weight to each edge (0=SIS, 1=CIS, 3=1-0S, 7=CSM).
dfsk	3*	Level of the alignment graph extension.
ifile	–	Name of the input alignment file.
rfile	–	Name of the output alignment file.
anfile	–	Name of the (optional) annotation file.
popsize	100	Size of the population.
islk	4	Number of subpopulations.
tgen	10000*	Maximal number of generations until termination.
tcgen	1000*	Number of generations for termination according to convergence.
elit	1	Use elitism (0=No, 1=Yes).
dupelim	1	Eliminate duplicates (0=No, 1=Yes).
pcross	1	Probability for creating a new chromosome by crossover.
pmut	-1	Probability for mutating a new chromosome (Poisson distributed random number).
pmig	0.001	Probability with which migration takes place after each generation.
pprc	0.02*	Probability with which Path Relinking Crossover is performed each generation.
piem	0.1*	Probability with which Insert Edge Mutation is performed each generation.

Table 5.1: Overview of the parameters of our EA

For some of these parameters the default values were obtained by rigid optimisation (marked with *), while for other parameters the standard values as provided by the *EAlib* are used. For further information see section 6.2.

Chapter 6

Experimental Results

6.1 Evaluating the Alignment Quality

To rate the (biological) quality of a computed alignment, we use the evaluation program *baliscore.c* from the *BAliBASE* library. This program compares a computed alignment with the corresponding *BAliBASE* reference alignment and rates it based on a calculated sum-of-pairs score.

Definition 6.1 (Sum-of-pairs score) Let S be a set of n strings and $\hat{S}$ the corresponding alignment of length $\hat{l}$. Let $p_{i,j}(k) = 1$, if the two characters on positions (i, k) and (j, k) in $\hat{S}$ are also aligned in the *BAliBASE* reference alignment $\hat{S}_r$ of length $\hat{l}_r$, or $p_{i,j}(k) = 0$ otherwise. Then the sum-of-pairs score $SPS(k)$ for column k is defined as:

$$SPS(k) = \sum_{i=1}^{n} \sum_{j=1(i \neq j)}^{n} p_{i,j}(k).$$

The sum-of-pairs score SPS for the whole alignment is defined as:

$$SPS = \left(\sum_{k=1}^{\hat{l}} SPS(k) \right) / \left(\sum_{k_r=1}^{\hat{l}_r} SPS(k_r) \right),$$

which always produces a score between 0 (no pair of characters is correctly aligned) and 1 (optimal alignment).

Additionally, *baliscore.c* calculates a second sum-of-pairs score *(SPS*)*, which only considers the trusted regions of the reference alignments for comparison by using the *BAliBASE* annotation files (also see section 3.5.1).

6.2 Optimising the Program Parameters

The parameter setting is mainly responsible for the overall performance of our MSA program. However, not all of the parameters are equally important. While some of them hardly affect the performance of our approach (or only affects it in certain situations) and standard values provide a reasonable compromise for general use, others are mainly responsible for the running time of the EA and the quality of the obtained solution. These parameters (marked with * in section 5.4) are further analysed in the following and tried to be optimised using the test instances of reference set 1 of the *BAliBASE* library.

Naturally, these optimised values are not perfect for any test instance. However, as shown in the comparative results of section 6.4, they provide a reasonable compromise between algorithm speed and solution quality.

6.2.1 Weighting Strategy and Level of the Alignment Graph Extension

As described in section 3.3, four different weighting schemes have been implemented to assign a weight to the edges of the pairwise alignment graph:

- *S*equence identity score (SIS),
- *CLUSTAL W*'s identity score (CIS),
- 1-0 score (1-0S),
- *CLUSTAL W*'s similarity matrices with window filtering (CSM).

For comparison of these weighting schemes, each of them is used to build the corresponding pairwise alignment graph. Then this graph is extended to levels 2 and 3. Finally the greedy2 heuristic is applied to derive a multiple alignment.

To evaluate the quality of each of these computed alignments, *baliscore.c* is used to calculate the corresponding SPS-score and SPS*-score. The results for the test alignments are shown in Tables 6.1 and 6.2.

	SIS		CIS		1-0S		CSM	
	SPS	*SPS**	*SPS*	*SPS**	*SPS*	*SPS**	*SPS*	*SPS**
Reference 1a	0,445	0,588	0,443	0,566	0,343	0,464	**0,476**	**0,619**
Reference 1b	0,839	0,949	0,838	0,948	0,779	0,905	**0,855**	**0,959**
Reference 1c	0,941	0,979	0,941	**0,980**	0,922	0,969	**0,942**	**0,980**

Table 6.1: Comparing different weighting schemes (extension level 2)

	SIS		CIS		1-0S		CSM	
	SPS	*SPS**	*SPS*	*SPS**	*SPS*	*SPS**	*SPS*	*SPS**
Reference 1a	0,459	0,596	0,451	0,573	0,372	0,508	**0,487**	**0,628**
Reference 1b	0,839	0,946	0,842	0,945	0,790	0,915	**0,856**	**0,958**
Reference 1c	0,941	0,979	0,941	0,979	0,923	0,969	**0,942**	**0,980**

Table 6.2: Comparing different weighting schemes (extension level 3)

As shown in Tables 6.1 and 6.2, the greedy2 heuristic always produces the best results using *CLUSTAL W*'s scoring matrices with window filtering (CSM) to build the alignment graph. Also the further extension of the alignment graph to level 3 provides appreciable improvements, especially for test instances with lower sequence similarity.

Thus, in the following, we will use *CLUSTAL W*'s scoring matrices with window filtering to build the pairwise alignment graph and we will extend it to level 3 in general.

6.2.2 Usage of Path Relinking Crossover and Insert Edge Mutation

Analysing the evolutionary process, we can see that two parameters mainly influence the performance of our evolutionary algorithm: the probability with which Path Relinking Crossover takes place (*pprc*) and the probability with which Insert Edge Mutation is performed each generation (*piem*).

Two main effects can be observed, when these parameters are varied:

- Using the operators only rarely, the evolutionary algorithm converges rather slowly.

- Applying them rather frequently, the running time of our EA increases dramatically.

These results are not very surprising as the operators are designed to speed up the evolutionary process on the one hand but also have a huge complexity compared to the other operators of our EA on the other hand.

Therefore, we tried to optimise these parameters by analysing the performance of our EA using different probabilites for applying Path Relinking Crossover ($pprc = 0.1\%, 0.02\%, 0.01\%, 0.002\%$) and Insert Edge Mutation ($piem = 0.1\%, 0.01\%$). For meaningful results our evolutionary algorithm was run 10 times with each parameter configuration on reference set 1 of the *BAliBASE* library. To ensure that the EA has converged, the EA was always run for 50000 generations ($tgen = 50000$).

For comparison, the following two values were computed: t (in seconds), the time to perform one generation in the EA on average, and g, the generation when the best solution is obtained. Additionally the approximative computational time $t \cdot (g + tcgen)$ with $tcgen = 1000$, 3000, and 5000 was calculated, as for further computation a more restrictive termination condition is applied (also see next section – $tcgen$, termination according to convergence). The corresponding results are shown in Table 6.3.

Note that the obtained solutions were always of similar alignment quality, hence the corresponding fitness scores respectively SPS scores are not displayed in Table 6.3.

$piem$	$pprc$	t [sec]	g	$t \cdot (g + 1000)$	$t \cdot (g + 3000)$	$t \cdot (g + 5000)$
0.1%	0.1%	0,0362	3806	173.97	246.37	318.77
0.1%	0.02%	0,0323	3935	**159.40**	**224.00**	288.60
0.1%	0.01%	0,0316	4185	163.84	227.04	290.24
0.1%	0.002%	0,0312	4392	168.23	230.63	293.03
0.01%	0.1%	0,0336	4326	178.95	246.15	313.35
0.01%	0.02%	0,0294	4677	166.90	225.70	**284.50**
0.01%	0.01%	0,0290	4849	169.62	227.62	285.62
0.01%	0.002%	0,0285	5271	177.18	234.18	291.18

Table 6.3: Approximative computational time ($piem$ and $pprc$ are varied)

Based on the results of Table 6.3 (note that we will use $tcgen = 1000$ for further computation – see next section), a probability of 0.1% for Insert Edge Mutation and 0.02% for Path Relinking Crossover is applied to our EA by default.

6.2.3 Termination Condition

In the last section, our evolutionary algorithm was always run for 50000 generations. However, in all test instances of reference set 1 of the *BAliBASE* library, the best solution was already found in former generations (see Table 6.3). Thus, an optimised (more restrictive) termination condition allows to speed up the evolutionary process while keeping the quality of the computed alignments.

Therefore, our EA was run on these test instances with different termination conditions again. The respective running times and the quality of the obtained alignments were compared to the results of the former runs. Various tests indicate that the following combination of termination conditions performs best on reference set 1 of the *BAliBASE* library: $tcgen = 1000$ with $tgen = 10000$.

Hence, these values are used in the following sections to terminate our evolutionary algorithm.

6.3 Overall Results

For meaningful results, our evolutionary algorithm was run 10 times on each test instance of the *BAliBASE* library using the default parameters as provided in section 5.4.

The respective values compared to the greedy2 heuristic are shown in Table 6.4. Note that Sum(w) denotes the sum–of–weights of all the edges, for which a corresponding pairwise character alignment exists in the solution, and Var(w) denotes the variance of these sum–of–weight scores based on the 10 runs performed with each test instance.

	greedy2			EA (avg of 10)			
	Sum(w)	*SPS*	*SPS**	Sum(w)	Var(w)	*SPS*	*SPS**
Reference 1a	516.080	0.487	0.628	535.561	7.896	0.497	**0.634**
Reference 1b	1074.468	0.856	**0.958**	1076.131	0.553	0.858	**0.958**
Reference 1c	1594.748	0.942	**0.980**	1595.251	0.086	0.941	**0.980**
Reference 2	17278.645	0.830	**0.892**	17284.870	11.714	0.831	**0.892**
Reference 3	21628.881	0.731	0.839	21675.844	39.657	0.740	**0.846**
Reference 4a	4772.292	0.636	0.775	4784.547	10.938	0.643	**0.777**
Reference 4b	5206.853	0.794	**0.877**	5210.772	2.934	0.794	0.875
Overall running time of our EA:				$\approx$ 55 hours (1 run)			

Table 6.4: Overall results

The full result table containing the scores of each test instance can be found in the appendix.

As shown in Table 6.4, our evolutionary algorithm is generally able to improve (or even optimise) the sum–of–weights score of the alignment obtained by the greedy2 heuristic. However, these improvement are quite different. While for test instances with higher sequence similarity the greedy2 heuristic already provides near optimal solutions, test instances with lower sequence similarity can be significantly improved by our EA. This phenomenon has already been observed before (see section 3.5) and is a result of the quality of the respective alignment graph (particularly reflected by the weight-based distribution of the edges in the alignment graph).

Considering the biological quality of the solutions (the SPS score), the following observation can be made. Although our EA usually increases the sum–of–weights score of the given multiple alignment, the biological quality often does not improve accordingly (e.g. reference set 4b). This shortcoming stems from the fact that we use different functions for optimisation and comparison. So it is possible that we have already found a biologically optimal alignment but our evolutionary algorithm still does not converge – e.g. a biologically correct edge (with high weight) is replaced by a set of biologically incorrect edges (each with smaller weight but with higher overall weight). Thus, future work should aim to further improve the biological quality of the alignment graph to make it biologically more correct (e.g. by considering information of local alignments).

Nevertheless, our solutions are still fairly good compared to alignments ob-

tained by other MSA programs. This fact is further documented in the next section.

6.4 Comparisons with other MSA Programs

To evaluate the overall performance of our evolutionary algorithm and the quality of the obtained solutions, our EA is compared with three existing MSA programs using all reference alignments of the *BAliBASE* library. These other MSA packages are:

- *CLUSTAL W* 1.82 [28], the most popular progressive alignment strategy,
- Fauster's Meta Heuristics *(FMH)* [7], a combination of greedy heuristics, local improvement strategies and tabu search,
- *SAGA* 0.95 [21], a widely–used evolutionary algorithm for MSA.

The comparative results between our EA and the three other methods are shown in Table 6.5.

	EA (avg of 10)		*CLUSTAL W*		*FMH*		*SAGA*	
	SPS	*SPS**	*SPS*	*SPS**	*SPS*	*SPS**	*SPS*	*SPS**
Reference 1a	0.497	0.634	0.512	0.650	0.509	**0.655**	0.395	0.471
Reference 1b	0.858	**0.958**	0.828	0.914	0.866	0.943	0.788	0.875
Reference 1c	0.941	**0.980**	0.943	0.965	0.949	0.970	0.915	0.933
Reference 2	0.831	**0.892**	0.795	0.875	0.814	0.880	0.738	0.804
Reference 3	0.740	0.846	0.749	0.863	0.754	**0.867**	0.617	0.716
Reference 4a	0.643	0.777	0.621	0.760	0.639	**0.782**	0.459	0.522
Reference 4b	0.794	0.875	0.757	0.850	0.801	**0.900**	0.615	0.705

Table 6.5: Comparison with other MSA programs

The full result table containing the respective scores of each test instance can be found in the appendix.

Note that these average scores for entire reference sets have to be treated with care – especially for reference alignments with lower sequence similarity (e.g. reference set 1a). While our EA performs rather similar (or sometimes even better) for most of these test instances, for some of them the results

are much worse – based on the nonsatisfying biological quality of the corresponding alignment graph there. Hence, average scores for entire reference set can be some kind of misleading.

Considering all used test instances of the BAliBASE library our EA is able to compute the best solution in 54% of all instances (compared to 28% by *CLUSTAL W*, 37% by Fauster's Meta Heuristics and 15% by *SAGA*).

Naturally, the running time cannot be ignored totally. So, it can take some time until the EA converges, especially for test instances with many sequences. However, it should always be taken into account that we use the greedy2 heuristic for initialisation, so a termination of the EA ahead of time still provides solutions of good quality.

Chapter 7

Conclusions

In this paper we presented a new approach for fast and accurate multiple sequence alignment, able to outperform even widely used MSA packages like *CLUSTAL W* on many test instances of the BAliBASE library. Three strategies were mainly responsible for the good performance of our algorithm: The usage of an alignment graph for the evaluation of multiple alignments, a fast heuristic to produce qualitatively good solutions in short time and the further optimisation of these alignments with a parallel evolutionary algorithm.

An alignment graph is able to reflect much more information than other evaluation methods like e.g. sum-of-pairs functions. So our alignment graph contained

- information of pairwise alignments as computed by *CLUSTAL W*,
- local information obtained by using an appropriate weighting scheme (*CLUSTAL W*'s similarity matrices with window filtering),
- global information gained by analysing the consistency between all sequences (alignment graph extension).

We validated the effectivity of this evaluation method by comparisons on reference alignments of the BAliBASE library and used this approach for our evolutionary algorithm.

A common problem of evolutionary algorithms is the long computational time required for useful results. To overcome this problem we applied a fast heuristic to create the initial population of our EA (greedy2 based initialisation). For many test alignments already these initial solutions were qualitatively better than the final solutions of other tested MSA packages.

The initial alignments were further improved by a parallel evolutionary algorithm, which uses fast variation operators as well as global optimisation heuristics. Using such a set of problem specific improvement operators, our method was able to compute the best solution in 54% of all used test instances of the BAliBASE library (compared to 28% by *CLUSTAL W*, 37% by Fauster's Meta Heuristics and 15% by *SAGA*) .

Future work should aim to further improve the biological quality of the alignment graph to make it biologically more correct (e.g. by considering information of local alignments).

Appendix

	greedy2			EA (avg of 10)				Clustal W		FMH		SAGA	
	Sum(w)	SPS	SPS*	Sum(w)	Var(w)	SPS	SPS*	SPS	SPS*	SPS	SPS*	SPS	SPS*
ref1a/1aboA	206,854	0,584	0,674	222,345	0,027	0,654	0,768	0,693	0,772	0,680	0,792	0,513	0,581
ref1a/1idy	210,257	0,604	0,673	211,499	0,331	0,656	0,676	0,546	0,569	0,621	0,662	0,354	0,446
ref1a/1r69	147,015	0,450	0,508	147,791	0,295	0,427	0,483	0,538	0,717	0,439	0,525	0,576	0,417
ref1a/1tvxA	162,554	0,530	0,658	165,992	0,309	0,478	0,583	0,223	0,200	0,492	0,600	0,633	0,700
ref1a/1wit	464,695	0,702	1,000	464,708	0,001	0,705	1,000	0,630	0,873	0,708	1,000	0,721	0,990
ref1a/2trx	288,362	0,692	0,737	290,259	2,175	0,692	0,737	0,660	0,707	0,694	0,707	0,568	0,430
ref1a/1bbt3	638,618	0,351	0,507	660,003	91,771	0,328	0,445	0,512	0,805	0,318	0,463	0,392	0,523
ref1a/1havA	336,912	0,227	0,415	361,186	5,031	0,224	0,417	0,222	0,338	0,281	0,556	0,210	0,336
ref1a/1sbp	624,307	0,481	0,560	643,256	8,523	0,489	0,578	0,467	0,594	0,492	0,594	0,336	0,444
ref1a/1uky	353,249	0,444	0,720	363,449	0,336	0,443	0,724	0,531	0,741	0,423	0,660	0,214	0,364
ref1a/2hsdA	497,880	0,543	0,644	516,933	0,076	0,572	0,696	0,482	0,635	0,563	0,676	0,393	0,516
ref1a/2pia	502,947	0,604	0,787	514,931	0,078	0,600	0,785	0,624	0,848	0,673	0,847	0,555	0,678
ref1a/3grs	369,383	0,301	0,479	382,430	0,014	0,298	0,469	0,377	0,410	0,304	0,490	0,213	0,281
ref1a/kinase	886,568	0,611	0,747	893,929	1,180	0,618	0,751	0,655	0,790	0,636	0,749	0,426	0,507
ref1a/1ajsA	900,481	0,344	0,469	921,804	4,988	0,361	0,473	0,386	0,574	0,413	0,587	0,226	0,283
ref1a/1cpt	973,199	0,700	0,822	978,786	15,219	0,705	0,829	0,697	0,836	0,697	0,820	0,632	0,786
ref1a/1lvl	533,604	0,391	0,580	592,861	3,727	0,403	0,557	0,368	0,467	0,411	0,578	0,194	0,201
ref1a/1pamA	1057,040	0,395	0,600	1086,742	3,753	0,401	0,600	0,408	0,581	0,397	0,619	0,079	0,123
ref1a/1ped	383,518	0,594	0,676	413,410	6,331	0,594	0,692	0,678	0,777	0,600	0,692	0,669	0,773
ref1a/2myr	535,886	0,330	0,442	612,458	25,577	0,370	0,475	0,394	0,538	0,384	0,533	0,085	0,123
ref1a/4enl	525,914	0,444	0,585	537,271	2,451	0,481	0,626	0,664	0,820	0,511	0,663	0,384	0,438
ref1a/gal4	754,520	0,392	0,522	800,302	1,522	0,431	0,577	0,518	0,698	0,451	0,588	0,306	0,420
ref1b/1aab	257,062	0,884	1,000	257,089	0,000	0,869	1,000	0,818	0,940	0,911	1,000	0,854	1,000
ref1b/1fjlA	533,982	1,000	1,000	533,982	0,000	1,000	1,000	0,994	1,000	1,000	1,000	0,989	1,000
ref1b/1hfh	612,773	0,868	0,966	612,773	0,000	0,868	0,966	0,820	0,886	0,880	0,946	0,861	0,927
ref1b/1hpi	223,555	0,923	0,989	223,555	0,000	0,923	0,989	0,864	0,935	0,923	0,946	0,885	0,924
ref1b/1csy	601,115	0,851	0,979	601,188	0,006	0,849	0,977	0,861	0,933	0,878	0,975	0,807	0,979
ref1b/1pfc	665,214	0,859	0,986	665,407	0,027	0,859	0,986	0,774	0,897	0,862	0,979	0,832	0,986
ref1b/1tgxA	159,461	0,817	0,947	160,178	0,000	0,837	0,935	0,833	0,914	0,837	0,866	0,647	0,704
ref1b/1ycc	311,392	0,816	0,918	311,427	0,001	0,819	0,918	0,810	0,885	0,825	0,832	0,720	0,737
ref1b/3cyr	320,732	0,802	0,903	320,734	0,000	0,799	0,898	0,722	0,784	0,830	0,875	0,789	0,867
ref1b/451c	299,182	0,618	0,814	303,374	0,079	0,629	0,820	0,555	0,649	0,640	0,739	0,611	0,674
ref1b/1ad2	650,520	0,853	0,963	652,551	0,393	0,862	0,963	0,821	0,949	0,868	0,971	0,845	0,931
ref1b/1aym3	871,294	0,882	0,950	873,932	0,053	0,892	0,950	0,927	0,957	0,893	0,947	0,874	0,971
ref1b/1gdoA	718,386	0,799	0,927	719,099	0,099	0,802	0,927	0,850	0,908	0,812	0,902	0,844	0,908
ref1b/1ldg	1216,160	0,935	0,992	1216,170	0,000	0,935	0,993	0,920	0,967	0,940	0,996	0,886	0,946
ref1b/1mrj	856,982	0,917	0,997	857,700	0,289	0,923	0,997	0,853	0,993	0,927	1,000	0,896	0,985
ref1b/1pgtA	804,819	0,936	1,000	804,945	0,012	0,936	1,000	0,941	1,000	0,936	1,000	0,866	1,000
ref1b/1pii	843,610	0,810	0,873	843,610	0,000	0,810	0,873	0,787	0,841	0,813	0,871	0,799	0,862

	greedy2			EA (avg of 10)				Clustal W		FMH		SAGA	
	Sum(w)	SPS	SPS*	Sum(w)	Var(w)	SPS	SPS*	SPS	SPS*	SPS	SPS*	SPS	SPS*
ref1b/1ton	1013,540	0,864	0,955	1014,343	0,313	0,866	0,955	0,718	0,801	0,880	0,953	0,752	0,824
ref1b/2cba	932,422	0,803	0,968	932,860	0,137	0,806	0,968	0,702	0,875	0,796	0,968	0,715	0,898
ref1b/1ac5	1105,360	0,787	0,990	1116,528	0,700	0,805	0,997	0,788	0,974	0,802	0,986	0,629	0,789
ref1b/1dlc	1764,820	0,856	0,959	1772,218	0,809	0,857	0,960	0,848	0,960	0,864	0,961	0,742	0,824
ref1b/1eft	1263,950	0,854	0,935	1266,905	4,751	0,856	0,935	0,829	0,910	0,865	0,931	0,834	0,905
ref1b/1fieA	2579,370	0,946	0,984	2579,473	0,064	0,946	0,984	0,938	0,965	0,949	0,977	0,877	0,904
ref1b/1gowA	1256,710	0,768	0,922	1257,870	0,887	0,770	0,922	0,805	0,924	0,782	0,915	0,445	0,555
ref1b/1pkm	1502,370	0,898	0,987	1503,012	0,988	0,896	0,987	0,905	0,936	0,902	0,948	0,829	0,899
ref1b/1sesA	2734,580	0,962	0,992	2734,580	0,000	0,962	0,992	0,942	0,979	0,967	0,990	0,883	0,909
ref1b/2ack	1914,040	0,764	0,916	1919,985	0,363	0,766	0,917	0,705	0,865	0,775	0,914	0,498	0,613
ref1b/arp	2173,550	0,882	0,944	2174,388	0,614	0,883	0,945	0,818	0,901	0,891	0,952	0,753	0,858
ref1b/glg	2475,000	0,779	0,996	2481,983	6,004	0,783	0,999	0,750	0,956	0,787	0,992	0,739	0,933
ref1b/1adj	1572,080	0,943	1,000	1572,080	0,000	0,943	1,000	0,935	0,937	0,946	0,945	0,940	0,943
ref1c/1aho	347,498	0,914	1,000	347,547	0,005	0,909	1,000	0,816	0,920	0,933	1,000	0,920	0,983
ref1c/1csp	461,344	0,965	0,982	461,344	0,000	0,965	0,982	0,981	0,993	0,975	0,993	0,917	0,935
ref1c/1dox	316,715	0,918	0,937	316,715	0,000	0,918	0,937	0,914	0,860	0,918	0,860	0,848	0,797
ref1c/1fkj	641,620	0,923	1,000	641,999	0,007	0,924	1,000	0,901	0,971	0,927	0,987	0,926	0,982
ref1c/1fmb	423,683	0,943	0,983	423,683	0,000	0,943	0,983	0,964	0,984	0,964	0,984	0,964	0,984
ref1c/1krn	448,763	0,935	1,000	448,789	0,002	0,931	1,000	0,975	0,992	0,960	1,000	0,969	1,000
ref1c/1plc	529,073	0,938	0,976	529,089	0,001	0,938	0,976	0,914	0,953	0,938	0,976	0,943	0,976
ref1c/2mhr	760,186	0,970	0,985	760,186	0,000	0,970	0,985	0,978	0,990	0,981	0,990	0,970	0,990
ref1c/9rnt	646,260	0,958	0,995	646,260	0,000	0,958	0,995	0,938	0,963	0,965	0,978	0,991	0,973
ref1c/1amk	1666,620	0,988	1,000	1666,620	0,000	0,988	1,000	0,978	0,996	0,989	0,996	0,959	0,972
ref1c/1ar5A	742,413	0,900	0,985	744,506	0,014	0,909	0,985	0,953	0,986	0,910	0,982	0,982	0,992
ref1c/1ezm	2060,830	0,954	0,955	2061,920	0,000	0,953	0,955	0,981	0,950	0,956	0,927	0,920	0,892
ref1c/1led	845,753	0,941	0,977	846,252	0,386	0,936	0,970	0,900	0,932	0,953	0,969	0,744	0,757
ref1c/1ppn	1330,820	0,971	0,987	1330,960	0,016	0,972	0,987	0,987	0,984	0,982	0,982	0,986	0,987
ref1c/1pysA	972,133	0,936	0,984	972,290	0,013	0,935	0,984	0,950	0,919	0,948	0,920	0,938	0,900
ref1c/1thm	1054,210	0,880	0,972	1054,210	0,000	0,880	0,972	0,898	0,953	0,883	0,951	0,830	0,885
ref1c/1tis	1801,970	0,919	0,939	1805,555	0,008	0,918	0,940	0,965	0,973	0,946	0,958	0,940	0,949
ref1c/1zin	855,702	0,920	0,936	856,646	0,006	0,918	0,934	0,955	0,965	0,919	0,932	0,933	0,937
ref1c/5ptp	1387,300	0,938	0,970	1387,770	0,146	0,929	0,961	0,948	0,963	0,923	0,945	0,896	0,911
ref1c/1ad3	1829,690	0,943	0,988	1829,690	0,000	0,943	0,988	0,951	0,980	0,948	0,981	0,946	0,979
ref1c/1gpb	5481,220	0,979	0,994	5481,220	0,000	0,979	0,994	0,981	0,989	0,980	0,988	0,929	0,933
ref1c/1gtr	2604,280	0,949	0,991	2604,372	0,011	0,949	0,991	0,948	0,988	0,954	0,993	0,905	0,964
ref1c/1lcf	6192,960	0,958	0,994	6195,273	0,735	0,962	0,997	0,947	0,979	0,961	0,980	0,827	0,847
ref1c/1rthA	3319,710	0,915	0,967	3320,392	0,571	0,913	0,966	0,902	0,957	0,916	0,964	0,861	0,914
ref1c/3pmg	2105,470	0,972	0,998	2106,007	0,324	0,973	0,998	0,950	0,992	0,977	0,995	0,884	0,929
ref1c/actin	2637,230	0,961	0,991	2637,230	0,000	0,961	0,991	0,931	0,964	0,964	0,985	0,868	0,894
ref2/1aboA	3479,210	0,841	0,939	3479,828	0,221	0,842	0,940	0,797	0,880	0,838	0,904	0,857	0,910
ref2/1idy	6277,370	0,772	0,755	6283,406	14,780	0,772	0,757	0,774	0,810	0,723	0,735	0,828	0,873
ref2/1csy	7213,150	0,856	0,883	7234,726	3,812	0,859	0,882	0,805	0,862	0,841	0,891	0,754	0,810
ref2/1r69	8033,800	0,824	0,947	8036,496	1,960	0,824	0,947	0,792	0,941	0,806	0,944	0,765	0,857
ref2/1tvxA	4797,180	0,760	0,762	4797,180	0,000	0,760	0,762	0,737	0,753	0,743	0,764	0,723	0,746
ref2/1tgxA	4437,310	0,843	0,922	4442,422	8,421	0,842	0,921	0,735	0,851	0,802	0,876	0,795	0,853
ref2/1ubi	5241,300	0,834	0,879	5246,180	0,500	0,836	0,881	0,795	0,869	0,800	0,869	0,772	0,839
ref2/1wit	9977,300	0,811	0,953	9982,796	5,561	0,811	0,953	0,823	0,929	0,826	0,915	0,784	0,873
ref2/2trx	9391,440	0,877	0,902	9391,440	0,000	0,877	0,902	0,885	0,960	0,852	0,904	0,831	0,894
ref2/1uky	26579,500	0,829	0,895	26591,580	34,108	0,834	0,898	0,845	0,909	0,847	0,904	0,805	0,846
ref2/2hsdA	30432,400	0,849	0,901	30449,320	4,162	0,850	0,900	0,763	0,874	0,835	0,899	0,735	0,775
ref2/2pia	18095,600	0,889	0,992	18095,720	0,011	0,890	0,992	0,793	0,918	0,869	0,984	0,749	0,894
ref2/3grs	12599,800	0,799	0,836	12601,980	1,922	0,799	0,836	0,754	0,815	0,792	0,830	0,742	0,809
ref2/kinase	22942,000	0,847	0,903	22946,680	4,002	0,844	0,902	0,821	0,880	0,845	0,901	0,700	0,776
ref2/1ajsA	35902,000	0,795	0,818	35904,820	5,891	0,795	0,818	0,773	0,756	0,774	0,757	0,671	0,680

	greedy2			EA (avg of 10)				Clustal W		FMH		SAGA	
	Sum(w)	SPS	SPS*	Sum(w)	Var(w)	SPS	SPS*	SPS	SPS*	SPS	SPS*	SPS	SPS*
ref2/1cpt	19920,200	0,899	0,982	19920,200	0,000	0,899	0,982	0,846	0,950	0,897	0,974	0,721	0,807
ref2/1pamA	34682,100	0,769	0,917	34697,500	79,378	0,771	0,917	0,779	0,951	0,759	0,924	0,600	0,775
ref2/2myr	32099,600	0,812	0,887	32115,760	56,167	0,815	0,888	0,775	0,862	0,791	0,875	0,499	0,548
ref2/4enl	36193,000	0,861	0,872	36194,500	1,667	0,861	0,871	0,808	0,853	0,822	0,870	0,690	0,709
ref3/1r69	4697,140	0,484	0,773	4771,062	9,419	0,501	0,773	0,549	0,812	0,532	0,799	0,540	0,812
ref3/1wit	6014,630	0,795	0,849	6037,046	10,671	0,802	0,858	0,813	0,864	0,822	0,884	0,739	0,786
ref3/kinase	29306,100	0,760	0,844	29367,060	77,074	0,770	0,859	0,744	0,867	0,764	0,881	0,699	0,800
ref3/1pamA	34329,200	0,753	0,898	34370,960	91,914	0,753	0,894	0,786	0,955	0,740	0,897	0,588	0,741
ref3/1ped	29456,800	0,802	0,852	29492,720	30,468	0,810	0,865	0,808	0,879	0,824	0,887	0,586	0,651
ref3/4enl	33621,200	0,803	0,852	33667,960	39,047	0,809	0,860	0,786	0,826	0,818	0,863	0,666	0,671
ref3/2pia	13977,100	0,720	0,807	14024,100	19,004	0,736	0,811	0,754	0,840	0,780	0,861	0,502	0,551
ref4a/1aisB	660,506	0,161	0,243	785,585	47,120	0,222	0,260	0,315	0,371	0,209	0,314	0,154	0,157
ref4a/1ar1	3331,990	0,803	1,000	3332,132	0,120	0,805	1,000	0,760	0,977	0,811	1,000	0,635	1,000
ref4a/1au7A	379,745	0,333	0,357	381,898	1,482	0,337	0,357	0,184	0,150	0,299	0,357	0,131	0,107
ref4a/1ckaA	1386,400	0,924	1,000	1386,378	0,000	0,924	1,000	0,823	1,000	0,926	1,000	0,531	0,397
ref4a/1dynA	510,116	0,439	0,833	512,537	5,500	0,444	0,833	0,396	0,667	0,436	0,833	0,296	0,620
ref4a/1lkl	1364,280	0,878	1,000	1366,940	3,682	0,884	1,000	0,756	1,000	0,884	1,000	0,622	0,750
ref4a/1mfa	639,854	0,388	0,560	645,045	10,791	0,404	0,560	0,412	0,678	0,455	0,678	0,242	0,216
ref4a/1pfc	1797,980	0,492	1,000	1803,493	24,417	0,494	1,000	0,460	0,994	0,485	1,000	0,394	0,542
ref4a/1pysA	614,393	0,533	0,625	616,062	2,526	0,540	0,625	0,558	0,500	0,534	0,625	0,460	0,500
ref4a/1vln	6110,710	0,945	0,948	6112,552	4,314	0,943	0,948	0,925	0,978	0,939	0,948	0,688	0,608
ref4a/1ycc	1930,090	0,780	0,848	1930,897	0,664	0,785	0,861	0,781	0,875	0,789	0,872	0,604	0,510
ref4a/2abk	1497,240	0,615	0,667	1516,435	46,628	0,614	0,667	0,610	0,667	0,614	0,667	0,595	0,667
ref4a/kinase1	2650,680	0,698	0,792	2655,598	5,887	0,696	0,792	0,873	0,875	0,676	0,714	0,503	0,598
ref4a/1ag8	43938,100	0,918	0,978	43938,100	0,000	0,918	0,978	0,844	0,913	0,883	0,938	0,569	0,638
ref4b/1pysA	2606,380	0,633	0,747	2614,993	4,389	0,635	0,747	0,580	0,683	0,636	0,786	0,380	0,481
ref4b/1eft	1300,840	0,411	0,526	1327,843	6,692	0,409	0,507	0,363	0,395	0,429	0,573	0,079	0,096
ref4b/1ivy	4141,120	0,786	0,735	4141,373	0,099	0,786	0,735	0,818	1,000	0,822	0,870	0,680	0,611
ref4b/1qpg	2253,300	0,859	0,925	2254,610	2,031	0,865	0,928	0,903	1,000	0,882	0,950	0,703	0,858
ref4b/1thm1	3843,060	0,829	0,919	3845,673	14,753	0,832	0,919	0,706	0,769	0,851	0,957	0,719	0,833
ref4b/1thm2	2023,020	0,878	0,935	2023,085	0,009	0,878	0,935	0,852	0,862	0,887	0,935	0,826	0,770
ref4b/2cba	2947,070	0,832	0,929	2947,367	0,068	0,832	0,929	0,769	0,913	0,829	0,929	0,633	0,950
ref4b/S51	11674,000	0,853	1,000	11677,480	1,549	0,847	1,000	0,796	0,931	0,839	0,981	0,684	0,851
ref4b/S52	1550,990	0,890	1,000	1551,040	0,000	0,890	1,000	0,896	1,000	0,896	1,000	0,907	1,000
ref4b/kinase1	1409,220	0,828	0,923	1409,485	0,140	0,830	0,923	0,817	0,923	0,828	0,923	0,723	0,845
ref4b/kinase2	7973,040	0,853	0,941	7976,115	5,479	0,852	0,939	0,812	0,894	0,840	0,941	0,623	0,667
ref4b/kinase3	20760,200	0,874	0,943	20760,200	0,000	0,874	0,943	0,771	0,834	0,877	0,949	0,419	0,501

	greedy2			EA (avg of 10)				Clustal W		FMH		SAGA	
	Sum(w)	SPS	SPS*	Sum(w)	Var(w)	SPS	SPS*	SPS	SPS*	SPS	SPS*	SPS	SPS*
Reference 1a	516,080	0,487	0,628	535,561	7,896	0,497	0,634	0,512	0,650	0,509	0,655	0,395	0,471
Reference 1b	1074,468	0,856	0,958	1076,131	0,553	0,858	0,958	0,828	0,914	0,866	0,943	0,788	0,875
Reference 1c	1594,748	0,942	0,980	1595,251	0,086	0,941	0,980	0,943	0,965	0,949	0,970	0,915	0,933
Reference 2	17278,645	0,830	0,892	17284,870	11,714	0,831	0,892	0,795	0,875	0,814	0,880	0,738	0,804
Reference 3	21628,881	0,731	0,839	21675,844	39,657	0,740	0,846	0,749	0,863	0,754	0,867	0,617	0,716
Reference 4a	4772,292	0,636	0,775	4784,547	10,938	0,643	0,777	0,621	0,760	0,639	0,782	0,459	0,522
Reference 4b	5206,853	0,794	0,877	5210,772	2,934	0,794	0,875	0,757	0,850	0,801	0,900	0,615	0,705

Bibliography

[1] Algorithmic-Solutions. *LEDA 4.4*. Saarbrücken, 2003.

[2] S. Altschul, R. Carroll, and D. Lipman. Weights for data related by a tree. *Journal of Molecular Biology*, 207:647–653, 1989.

[3] L. Anbarasu, V. Sundararajan, and P. Narayanasamy. Multiple sequence alignment using parallel genetic algorithms. In *Proceedings of the 2nd Asia-Pacific Conference on Simulated Evolution and Learning*, pages 130–137, 1999.

[4] T. Bäck, D. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York, 1997.

[5] L. Cai, D. Juedes, and E. Liakhovitch. Evolutionary computation techniques for multiple sequence alignments. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 829–835, 2000.

[6] K. Chellapilla and G. Fogel. Multiple sequence alignment using evolutionary programming. In *Proceedings of the 1999 Congress on Evolutionary Computation*, pages 445–452, 1999.

[7] J. Fauster. *Neue heuristische Lösungsansätze für das Multiple Sequence Alignment Problem*. Vienna University of Technology, Vienna, 2003.

[8] D. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, New York, 1995.

[9] F. Glover, D. Corne, and M. Dorigo. Scatter search and path relinking. In *New Ideas in Optimization*, pages 295–355. McGraw Hill, 1999.

[10] D. Goldberg. *Genetic Algorithms in Search, Optimization and Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, 1997.

[12] D. Heesch. *DoxyGen 1.2.17*, 2002.

[13] J.-T. Horng, C.-M. Lin, B.-J. Liu, and C.-Y. Kao. Using genetic algorithms to solve multiple sequence alignments. In *Proceedings of the Second Genetic and Evolutionary Computation Conference*, pages 883–890, 2000.

[14] X. Huang and W. Miller. A time-efficient, linear space local similarity algorithm. *Advances in Applied Mathematics*, 12:337–357, 1991.

[15] J. Kececioglu. The maximum weight trace problem in multiple sequence alignment. In *Proceedings of the 4th Symposium on Combinatorial Pattern Matching*, pages 106–119, 1993.

[16] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, MA, 1992.

[17] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[18] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[19] H. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga. A parallel hybrid genetic algorithm for multiple protein sequence alignment. In *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 309–314, 2002.

[20] C. Notredame. Recent progresses in multiple sequence alignment: a survey. *Pharmacogenomics*, 3(1):131–144, 2002.

[21] C. Notredame and D. Higgins. SAGA: Sequence alignment by genetic algorithm. *Nucleic Acids Research*, 24(8):1515–1524, 1996.

[22] C. Notredame, D. Higgins, and J. Heringa. T-COFFEE: A novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 392:205–217, 2000.

[23] C. Notredame, L. Holm, and D. Higgins. COFFEE: An objective function for multiple sequence alignment. *Bioinformatics*, 14(5):407–422, 1998.

[24] G. Raidl. *EAlib 1.1.* Vienna University of Technology, 2002.

[25] G. Raidl, G. Kodydek, and B. Julstrom. On weight-biased mutation for graph problems. In *Proceedings of the Seventh International Conference on Parallel Problem Solving from Nature (PPSN VII)*, pages 204–213, 2002.

[26] K. Reinert, H.-P. Lenhof, P. Mutzel, K. Mehlhorn, and J. Kececioglu. A branch-and-cut algorithm for multiple sequence alignment. In *Proceedings of the First Annual International Conference on Computational Molecular Biology*, pages 241–249, 1997.

[27] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* International Thomson Publishing, 1996.

[28] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680, 1994.

[29] J. Thompson, F. Plewniak, and O. Poch. BAliBASE: A benchmark alignments database for the evaluation of multiple sequence alignment programs. *Bioinformatics*, 15:87–88, 1999.

[30] J. Thompson, F. Plewniak, and O. Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, 27(13):2682–2690, 1999.

[31] R. Thomsen, G. Fogel, and T. Krink. A Clustal alignment improver using evolutionary algorithms. In *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 121–126, 2002.

[32] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[33] C. Zhang and A. Wong. A genetic algorithm for multiple molecular sequence alignment. *CABIOS*, 13(6):565–581, 1997.