9 Multiple Sequence Alignment

This exposition is based on the following sources, which are recommended reading:

1. D. Mount: *Bioinformatics*. CSHL Press, 2004, chapter 5.
2. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997, chapter 14.
3. Chao, Zhang: *Sequence comparison*, Chapter 5

9.1 Introduction

Two facts of biological sequence comparison:

1. High sequence similarity $\stackrel{\text{usually}}{\longrightarrow}$ significant functional/structural similarity
2. Evolutionary/functionally related sequences can differ significantly at the primary sequence level.[1]

9.2 Introduction

What is a multiple sequence alignment? A multiple sequence alignment is simply an alignment of more than two sequences, like this:

```
SRC_RSVP   -FPIKWTAPEAALY---GRFTIKSDVWSFGILLTELTTKGRVPYPGMVNR-EVLDQVERG
YES_AVISY  -FPIKWTAPEAALY---GRFTIKSDVWSFGILLTELVTKGRVPYPGMVNR-EVLEQVERG
ABL_MLVAB  -FPIKWTAPESLAY---NKFSIKSDVWAFGVLLWEIATYGMSPYPGIDLS-QVYELLEKD
FES_FSVGA  QVPVKWTAPEALNY---GRYSSESDVWSFGILLWETFSLGASPYPNLSNQ-QTREFVEKG
FPS_FUJSV  QIPVKWTAPEALNY---GWYSSESDVWSFGILLWEAFSLGAVPYANLSNQ-QTREAIEQG
KRAF_MSV36 TGSVLWMAPEVIRMQDDNPFSFQSDVYSYGIVLYELMA-GELPYAHINNRDQIIFMVGRG
```

(A small section of six tyrosine kinase protein sequences.)

In this example multiple sequence alignment is applied to a set of sequences that are assumed to be homologous (have a common ancestor sequence) and the goal is to detect homologous residues and place them in the same column of the multiple alignment.

However, this is by far not the only use. While multiple sequence alignment (MSA) is a straightforward generalization of pairwise sequence alignment, there are lots of new questions about scoring, the significance of scores, gap penalties, and efficient implementations.

Definition.
Assume we are given k sequences $x_1, \ldots, x_k$ over an alphabet Σ.

Let $- \notin \Sigma$ be the *gap symbol*. Let $h\colon (\Sigma \cup \{-\})^* \to \Sigma^*$ be the mapping that removes all gap symbols from a sequence over the alphabet $\Sigma \cup \{-\}$. For example, h(`-FPIKWTAPEAALY---GRFT`) = `FPIKWTAPEAALYGRFT`.

[1] basically, this says that the reverse of 1. is *not* true in general.

Then a *global alignment* of $x_1, \ldots, x_k$ consists of k sequences $x'_1, \ldots, x'_k$ over the alphabet $\Sigma \cup \{-\}$ such that

- $h(x'_i) = x_i$ for all i,
- $|x'_i| = |x'_j|$ for all i, j, and
- $(x'_{1,p}, \ldots, x'_{k,p}) \neq (-, \ldots, -)$ for all p.

Example.

```
 x1  =  G C T G A T A T A G C T
 x2  =  G G G T G A T T A G C T
 x3  =  G C T A T C G C
 x4  =  A G C G G A A C A C C T

x'1  =  - G C T G A T A T A G C T
x'2  =  G G G T G A T - T A G C T
x'3  =  - G C T - A T - - C G C -
x'4  =  A G C G G A - A C A C C T
```

Below we give a list of the most common uses of multiple sequence alignment.

- detecting faint similarities in sequences that are not detected by pairwise sequence comparison.
- detecting structural homologies.
- grouping proteins into families.
- computing the consensus sequence in assembly projects.
- inferring evolutionary trees.
- and more ...

We now give some more details about the different uses.

9.3 MSA and evolutionary trees

One main application of multiple sequence alignment lies in phylogenetic analysis. Given an MSA:

```
a1 =  N  -  F  L  S
a2 =  N  -  F  -  S
a3 =  N  K  Y  L  S
a4 =  N  -  Y  L  S
```

We would like to reconstruct the evolutionary tree that gave rise to these sequences, e.g.:

N Y L S
N K Y L S
N F S
N F L S
+K
−L
Y to F
N Y L S

Before we can apply algorithms for phylogenetic tree reconstruction we need to find out how the positions of the sequences correspond to each other.

9.4 Protein families

Assume we have established a family $s_1, s_2, \ldots, s_r$ of homologous protein sequences. Does a new sequence s_0 belong to the family?

One method of answering this question would be to align s_0 to each of $s_1, \ldots, s_r$ in turn. If one of these alignments produces a high score, then we may decide that s_0 belongs to the family.

However, perhaps s_0 does not align particularly well to any one specific family member, but does well in a multiple alignment, due to common motifs etc.

9.5 Sequence Assembly

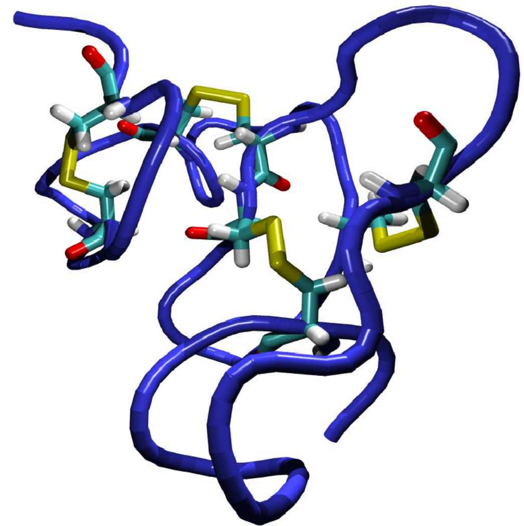
Assume we are given a layout of several genomic reads in a sequencing project that were produced using the shotgun sequencing method. These fragments will be highly similar, and hence easy to align. Nevertheless we want to do this with great speed and accuracy:

```
f1  ACCACAACCCTGCATGGGGCAT-ATTTGGCCTAGCT
f2                 AGGGCCTTATATG-GCTAGCT-CGTTCCCGGGCATGGC
f3             GCATGGGGCATTATCTGGCCTAGCT--GAT
f4                                    CCGTTCCCGG-CTTGGCAACG
=========================================================
cns ACCACAACCCTGCATGGGGCATTATCTGGCCTAGCT-CGTTCCCGGGCATGGCAACG
```

9.6 Conservation of structural elements

The below figure shows the alignment of N-acetylglucosamine-binding proteins and the tertiary structure of one of them, the hevein.

```
AATAHAQRCG EQGSNMECPN NLCCSQYGYC GMGGDYCGKG ..CQNGACYT
VAATNAQTCG KQNDGMICPH NLCCSQFGYC GLGRDYCGTG ..CQSGACCS
VGLVSAQRCG SQGGGGTCPA LWCCSIWGWC GDSEPYCGRT ..CENK.CWS
AATAQAQRCG EQGSNMECPN NLCCSQYGYC GMGGDYCGKG ..CQNGACWT
AATAQAQRCG EQGSNMECPN NLCCSQYGYC GMGGDYCGKG ..CQNGACWT
......QRCG EQGSGMECPN NLCCSQYGYC GMGGDYCGKG ..CQNGACWT
SETVKSQNCG .......CAP NLCCSQFGYC GSTDAYCGTG ..CRSGPCRS
RGSAE..QCG RQAGDALCPG GLCCSSYGWC GTTVDYCGIG ..CQSQ.CDG
AGPAAAQNCG .......CQP NFCCSKFGYC GTTDAYCGDG ..CQSGPCRS
AGPAAAQNCG .......CQP NVCCSKFGYC GTTDEYCGDG ..CQSGPCRS
RGSAE..QCG QQAGDALCPG GLCCSSYGWC GTTADYCGDG ..CQSQ.CDG
RGSAE..QCG RQAGDALCPG GLCCSFYGWC GTTVDYCGDG ..CQSQ.CDG
TGVAIAEQCG RQAGGKLCPN NLCCSQWGWC GSTDEYCSPD HNCQSN.CK.
......EQCG RQAGGKLCPN NLCCSQYGWC GSSDDYCSPS KNCQSN.CK.
```



The example exhibits 8 cysteins that form 4 disulphid bridges and are an essential structural part of those proteins.

9.7 Scoring schemas

What is the quality, or the score of a multiple alignment? There are many possible alignments and the natural question is of course which one is the best under some scoring scheme that resembles the relevant biological question best.

Generally one can define similarity measures or distance measures as it is the case for pairwise alignment. In what follows we will use interchangeably *distance* measures, for which we try to minimize the *cost*, and *similarity* measures for which we try to maximize the *score*.

9.8 Projection of a multiple alignment

We will also refer to a multiple alignment $x_1', \ldots, x_k'$ of k sequences $x_1, \ldots, x_k$ by a $k \times l$-matrix A with

$$l = |x_1'| \qquad (= |x_2'| = |x_2'| = \ldots = |x_k'|) \ ,$$

where $A[i][j]$ contains the j-th symbol of x_i'.

In what follows we need the definition of multiple alignments of subsets of the k strings.

Definition. Let A be a multiple alignment for the k strings $x_1, \ldots, x_k$ and let $I \subseteq \{1, \ldots, k\}$ be a set of indices defining a subset of the k strings. Then we define A_I as the alignment resulting from first removing all rows $i \notin I$ from A and then deleting all columns consisting entirely of blanks. We call A_I the *projection* of A to I. If I is given explicitly, we simplify notation and write, e.g., $A_{i,j,k}$ instead of $A_{\{i,j,k\}}$.

Example.

```
a1 = - G C T G A T A T A G C T
a2 = G G G T G A T - T A G C T
a3 = - G C T - A T - - C G C -
```

```
a4 = A G C G G A - A C A C C T
```

The projection $A_{2,3}$ is given by first taking the second and third row of the alignment:

```
a2 = G G G T G A T - T A G C T
a3 = - G C T - A T - - C G C -
```

and then deleting the column consisting only of blanks.

```
a2 = G G G T G A T T A G C T
a3 = - G C T - A T - C G C -
```

9.9 Cost functions

Let $c : \mathcal{A} \to \mathbb{R}$ be a function that maps each possible alignment in the set of all alignments $\mathcal{A}$ to a real number. Our goal is to find an optimal multiple alignment A^*, that is,

$$A^* = \arg\min_{A \in \mathcal{A}} c(A) \qquad \text{or} \qquad A^* = \arg\max_{A \in \mathcal{A}} c(A) \ ,$$

depending on whether we maximize similarity or minimize a distance.

In the following we describe a few common cost functions, namely

- consensus score
- profile score
- weighted sum of pairs (WSOP) score (with linear gap costs)
- phylogenetic score

9.10 Consensus alignment

Let $A[\,][j]$ be any column of a multiple alignment A. Then the letter x_j is called the j-th *consensus*-letter, if the *consensus*-error

$$\sum_{i=1}^{k} d(x_j, A[i][j])$$

is minimal. The concatenation of the consensus letters yields the *consensus*-string. Hence the goal is to find an alignment A^* that minimizes the consensus error summed over all columns.

The cost $c(A)$ is then defined as follows:

$$c(A) = \sum_{j=1}^{l} \sum_{i=1}^{k} d(x_j, A[i][j]) \qquad x_j \text{ is the } j\text{-th consensus letter}$$

Example.

Let $d(x, y) = \begin{cases} 2 & \text{for } x \neq y, \quad (x, y) \in \Sigma \times \Sigma \\ 1.5 & \text{for } x = - \text{ or } y = -\text{, but } (x, y) \neq (-, -) \\ 0 & \text{otherwise } . \end{cases}$

```
              a1 =  -   G   C   T   G   A   T   A   T   A   A   C   T
              a2 =  G   G   G   T   G   A   T   -   T   A   G   C   T
              a3 =  A   G   C   G   G   A   -   A   C   A   C   C   T
                    -------------------------------------------------
consensus   :       -   G   C   T   G   A   T   A   T   A   X   C   T
column value:       3   0   2   2   0   0  1.5 1.5  2   0   4   0   0 = 16
```

An X in the consensus string symbolizes that the consensus can be any letter (occuring in the corresponding column).

9.11 Profile alignment

Profiles are another way to describe a motif common to a family of sequences. Given a multiple alignment of a set of strings, a *profile* is obtained by recording the frequency of each character (including the blank) for each column.

Aligning a string S to a profile A accounts to computing the weighted sum of the scores of the letters of S to the columns of the profile A. The alignment algorithm is very similar to the dynamic programming for two sequences.

For a character y and a column j, let $p(y, j)$ be the frequency of character y in column j. Then the *score* of aligning a character x with column j is

$$\sum_y p(y, j)s(x, y).$$

Note that here we need an entry for $s(-, -)$ in the scoring matrix.

Profiles are also often called *position specific scoring matrices* (PSSM) in the biological literature.

Example.
We align AABBC against a profile for 4 sequences using a similarity score.

```
a1 = A B C - A       profile:  c1   c2   c3   c4   c5      score:   A    B    C    -
a2 = A B A B A             A: .75       .25       .50              A  2   -1   -3   -1
a3 = A C C B -             B:      .75       .75                   B -1    2   -2   -1
a4 = C B - B C             C: .25  .25  .50       .25              C -3   -2    2   -1
                           -:           .25  .25  .25              - -1   -1   -1    1

---------------------------------------------------------------
   column   |    column value calculation            |   sum
  seq  prof |    A        B        C        -        |

   A    1   =   +0.75*2           -0.25*3            =   0.75
   A    -   =   -1.0 *1                              =  -1.0
   B    2   =            +0.75*2  -0.25*2            =   1.0
   -    3   =   -0.25*1           -0.50*1  +0.25*1   =  -0.5
   B    4   =            +0.75*2           -0.25*1   =   1.25
   C    5   =   -0.5*3            +0.25*2  -0.25*1   =  -1.25
                                                       -------
                                                         0.25
```

9.12 WSOP score

One of the most common scoring functions for MSA is the *(weighted) sum of pairs*, which is defined as the (weighted) sum of the score of all pairwise projections of the MSA, that is,

$$c(A) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} w_{i,j} \cdot c(A_{i,j})$$

Each pair (i, j) can be given a different *weight* $w_{i,j}$. Note that $c(A_{i,j})$ involves another summation over the columns of $A_{i,j}$. If we assume independence of the score of the alignment columns then we can rewrite this as follows:

$$c(A) = \sum_{h=1}^{l} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} w_{i,j} \cdot s(A[i,h], A[j,h])$$

The nice thing about WSOP is that we can move the "inner" summation (running over the column index h) "outside", to the front.

Example.

Let $s(x,y) = \begin{cases} 3 & \text{for } x = y \\ -2 & \text{for } x \neq y, (x,y) \in \Sigma \times \Sigma \text{ (mismatch)} \\ -1 & \text{otherwise (gaps)} \end{cases}$.

All the weight factors are 1.

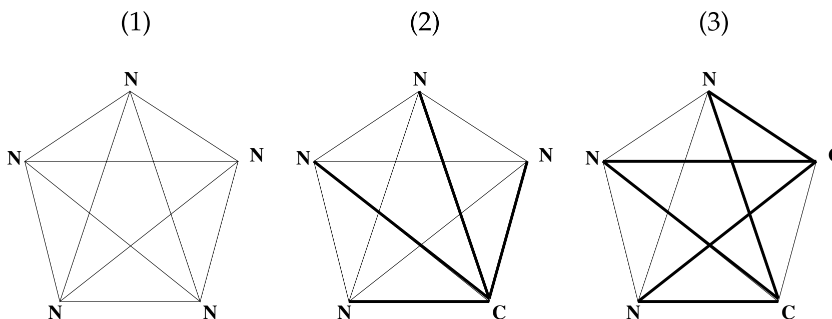
```
                a1 = -  G  C  T  G  A  T  A  T  A  A  C  T
                a2 = G  G  G  T  G  A  T  -  T  A  G  C  T
                a3 = A  G  C  G  G  A  -  A  C  A  C  C  T
                     ------------------------------------
score of column: -4  9 -1 -1  9  9  1  1 -1  9 -6  9  9 = 43
```

The sum of pair score has the advantage to take all pairwise information into account, however it is easily biased by over-representation of sequences from the same family. This disadvantage can be dealt with by choosing the weights accordingly.

Multiple alignment:

		(1)		(2)		(3)	
s_1	...	N	...	N	...	N	...
s_2	...	N	...	N	...	N	...
s_3	...	N	...	N	...	N	...
s_4	...	N	...	N	...	C	...
s_5	...	N	...	C	...	C	...

Comparisons: (1) (2) (3)



	(1)	(2)	(3)
N-N pairs:	10	6	3
N-C pairs:	0	4	6
C-C pairs:	0	0	1
BLOSUM50:	70	34	22

(BLOSUM50 scores: N-N: 7, N-C: -2, C-C: 13)

An undesirable property of the WSOP cost function is the following: Consider a position i in an SP-optimal multi-alignment A^* that is *constant*, i.e., has the same residue in all sequences.

What happens when we add a new sequence? If the number of aligned sequences is small, then we would not be too surprised if the new sequence shows a different residue at the previously constant position i.

However, if the number of sequences is large, then we would expect the constant position i to remain constant, if possible.

Unfortunately, the SP score favors the opposite behavior, when scoring using a BLOSUM matrix: the more sequences there are in an MSA, the easier it is, relatively speaking, for a differing residue to be placed in an otherwise constant column.

Example:

$$\begin{array}{rccc} a_1 = & \dots & \mathtt{E} & \dots \\ \vdots & \dots & \vdots & \dots \\ a_{r-1} = & \dots & \mathtt{E} & \dots \\ a_r = & \dots & \mathtt{E} & \dots \end{array} \quad \leftrightarrow \quad \begin{array}{rccc} a_1 = & \dots & \mathtt{E} & \dots \\ \vdots & \dots & \vdots & \dots \\ a_{r-1} = & \dots & \mathtt{E} & \dots \\ a_r = & \dots & \mathtt{C} & \dots \end{array}$$

Using BLOSUM50, we obtain:

$$c(\mathtt{E}^r) = \frac{r(r-1)}{2} s(\mathtt{E},\mathtt{E}) = 6\frac{r(r-1)}{2}.$$

The value for $c(\mathtt{E}^{r-1}\mathtt{C})$ is obtained from this by subtracting $(r-1)s(\mathtt{E},\mathtt{E})$ and then adding $(r-1)s(\mathtt{E},\mathtt{C})$. So, the difference between $c(\mathtt{E}^r)$ and $c(\mathtt{E}^{r-1},\mathtt{C})$ is:

$$(r-1)s(\mathtt{E},\mathtt{E}) - (r-1)s(\mathtt{E},\mathtt{C}) = (r-1)6 - (r-1)(-3) = 9(r-1).$$

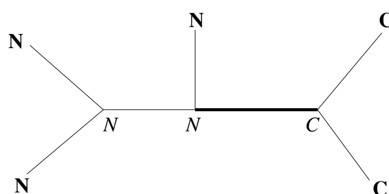
Therefore, the relative difference is

$$\frac{c(\mathtt{E}^r) - c(\mathtt{E}^{r-1}\mathtt{C})}{c(\mathtt{E}^r)} = \frac{9(r-1)}{6r(r-1)/2} = \frac{3}{r},$$

which *decreases* as the number of sequences r increases!

9.13 Scoring along a tree

Assume we have a *phylogenetic tree* T for the sequences that we want to align, i.e., a tree whose leaves are labeled by the sequences. Instead of comparing *all pairs* of residues in a column of an MSA, one may instead determine an optimal labeling of the internal nodes of the tree by symbols in a given column (in this case col. (3) from the example) and then sum over *all edges in the tree*:



Such an optimal *most parsimonious labeling* of internal nodes can be computed in polynomial time using the *Fitch* algorithm.

Based on this tree, the scores for columns (1), (2) and (3) are: $7\times7 = 49$, $6\times7-2 = 40$ and $4\times7-2+2\times13 = 52$.

9.14 Scoring along a star

In a third alternative, one sequence is treated as the ancestor of all others in a so-called *star phylogeny*:

```
N           C
      N
N           C
```

Based on this star phylogeny, assuming that sequence 1 is at the center of the star, the scores for columns (1), (2) and (3) are: $4 \times 7 = 28$, $3 \times 7 - 2 = 19$ and $2 \times 7 - 2 \times 2 = 10$.

9.15 Scoring Schemes

At present, there is no conclusive argument that gives any one scoring scheme more justification than the others. The sum-of-pairs score is widely used, but is problematic.

One advantage of the WSOP score (or cost) function is the ease with which gaps are modelled, since the score reduces to the summation of the pairwise scores for which the handling of gaps is well understood.

9.16 Assessing the quality of multiple alignments

Since there is no generally accepted cost function for multiple alignments it is difficult to assess their quality, especially in the case of aligning remote homologs.

There are several publically available databases which have *benchmark* alignments. The first one which is widely used is *BAliBASE* by Julie Thompson, Frédéric Plewniak and Olivier Poch (1999) Bioinformatics, 15, 87-88.
http://www-igbmc.u-strasbg.fr/BioInfo/BAliBASE2/

The sequences in the database are carefully aligned and expert curated. Then the *core* blocks of structural conserved regions are extracted. The score for an alignment is the percentage of those regions which it is able to find.

Below you can see one of the reference alignments. The crucial core blocks are printed in capital letters:

```
hmgl_trybr   1 kkdsnaPKRAMTSFMFFSS....dfrskhsdlsi.vemsKAAGAAWKEL
hmgt_mouse   1 .....kPKRPRSAYNIYVSesfqeakddsaqgkl.....KLVNEAWKNL
hmgb_chite   1 ...adkPKRPLSAYMLWLNsaresikrenpdfkv.tevaKKGGELWRGL
hmgl_wheat   1 ..dpnkPKRAPSAFFVFMGefreefkqknpknksvaavgKAAGERWKSL

hmgl_trybr 45 gpeeRKVYEEMAEKDKERYKREM.........
hmgt_mouse 40 speeKQAYIQLAKDDRIRYDNEMksweeqmae
hmgb_chite 46 ..kdKSEWEAKAATAKQNYIRALqeyerngg.
hmgl_wheat 48 seseKAPYVAKANKLKGEYNKAIaaynkgesa
```

The following table lists the results of some alignment programs evaluated with the BaliBase evaluation scheme. In parenthesis is the percentage of the highest score any program reached. For example, for the 1aboA alignment, COSA is the best (0.758) of all programs and has a score of *100%*. TCOFFEE reaches (0.579) which is *76%*.

Data	COSA	TCOFFEE	PRRP	CLUSTALW	DIALIGN
BaliBase Reference 1 short V1					
1aboA	**0.758 *(100)***	0.579 *(76)*	0.327 *(43)*	0.755 *(100)*	0.645 *(85)*
1idy	0.723 *(94)*	0.196 *(25)*	**0.769 *(100)***	0.608 *(79)*	0.050 *(7)*
1r69	**0.673 *(100)***	0.567 *(84)*	0.520 *(77)*	0.640 *(95)*	0.333 *(49)*
1tvxA	0.219 *(81)*	**0.272 *(100)***	0.219 *(81)*	0.123 *(45)*	0.088 *(32)*
1ubi	0.500 *(66)*	0.447 *(59)*	0.500 *(66)*	**0.760 *(100)***	0.180 *(24)*
1wit	**0.992 *(100)***	0.925 *(93)*	**0.992 *(100)***	0.892 *(90)*	0.800 *(81)*
2trx	0.788 *(97)*	**0.814 *(100)***	0.447 *(55)*	0.795 *(98)*	0.792 *(97)*
avg.	**0.665 *(100)***	0.543 *(82)*	0.539 *(81)*	0.653 *(98)*	0.413 *(62)*

9.17 Methods for MSA

Most optimization problems using the above cost functions are *NP*-hard. Hence exact solutions cannot be expected for more than 7-15 sequences and one has to resort to heuristic approaches. Most multiple alignment methods can be classified as follows:

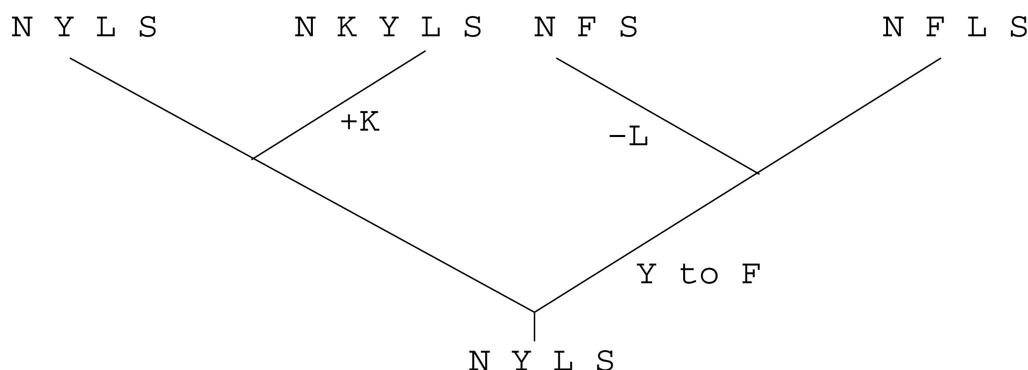
Iterative algorithms (Realigner,PRRP, SAGA)
Progressive algorithms (ClustalW, Muscle)
Consistency based algorithms (TCoffee, ProbCons)
Motif searching algorithms (Dialign, Blocks, eMotif)
Probabilistic methods (HMMs, Gibbs-Sampling)
Divide-and-Conquer algorithms (DCA,OMA)
Exact algorithms (MSA,COSA,GSA)

We give now a short overview of the central ideas.

9.18 Progressive methods

Progressive alignments start by aligning the most similar sequences first in the hope that the fewest errors are made. Then, progressively, more and more sequences are aligned to the already existing alignment.

This is typically done with the help of a (heuristic) phylogenetic tree.



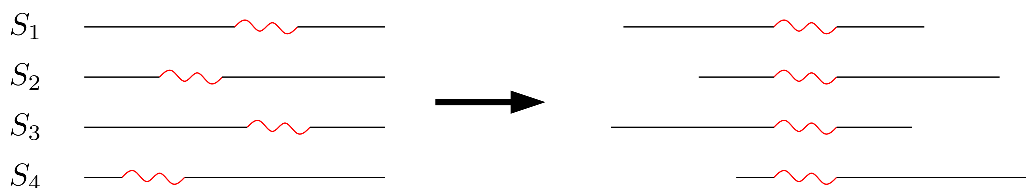
9.19 Iterative methods

The major problem with progressive methods is their sensitivity to a bad initial alignment. Iterative methods attempt to avoid this by repeatedly aligning subgroups of the sequences and then by aligning these subgroups into a larger alignment.

Since they run certain heuristics several times they are normally somewhat slower than progressive alignments. the most prominent programs here are PRRP (Gotoh) and SAGA (Notredame and Higgins).

9.20 Motif based methods

These methods try to find local motifs and use those as anchors.



The most prominent algorithms here are programs from the Dialign family and the BlockAligner.

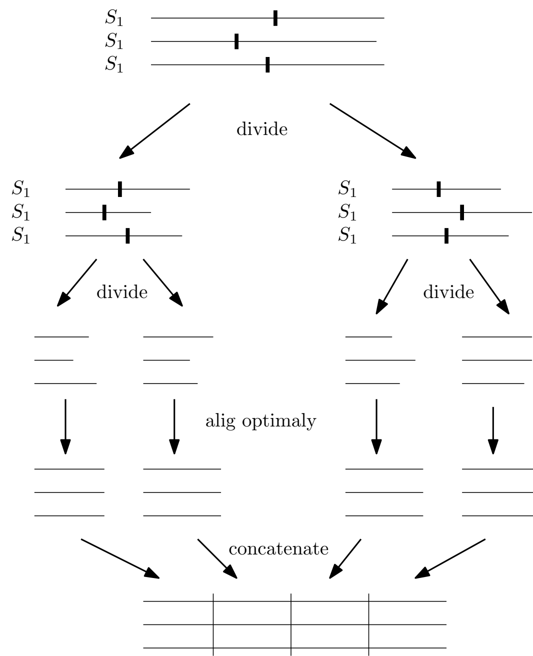
9.21 Probabilistic methods

The most prominent of these methods are Profile HMMs and the Gibbs sampler. These methods try to maximize the likelihood in a probabilistic model of multiple alignment.

9.22 Divide-and-conquer methods

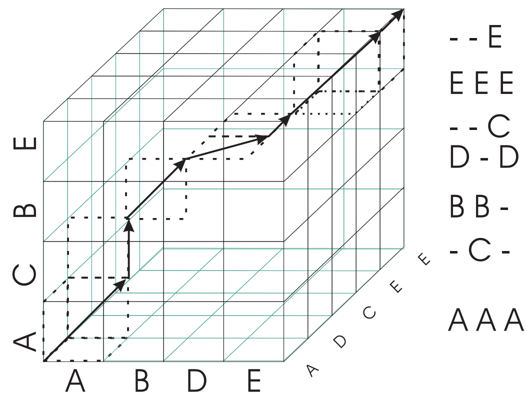
The idea is straightforward: cut the k sequences and solve the resulting subproblems recursively. The solutions can be combined trivially. If the problem size is small enough an exact algorithm can be employed.

On the one hand it is clear that optimal cut positions exist, on the other hand it is clear that it is NP-hard to find them. Trivial ideas for determining the cut positions are not very succesful. Stoye showed how to compute relatively good cut positions.

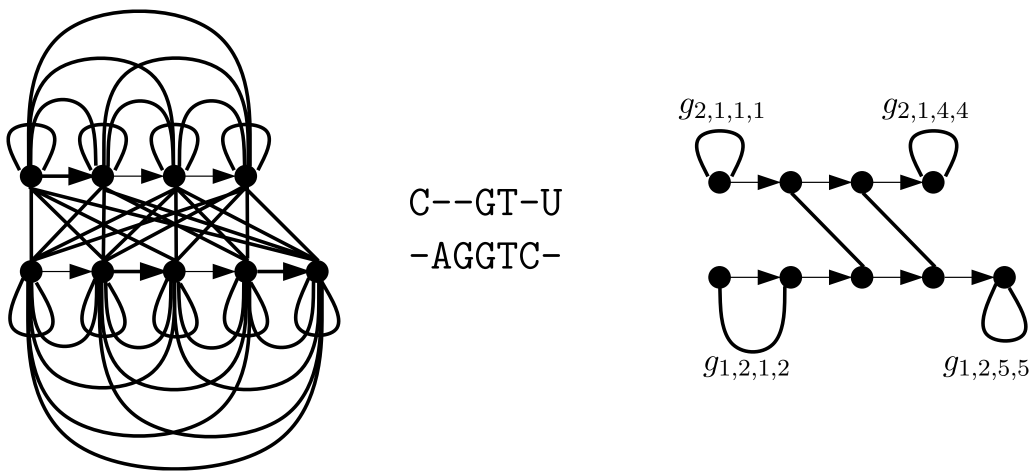


9.23 Exact methods

Exact methods are either based on the natural extension of the dynamic programming algorithm for two sequences (e. g. (Lipman, Gupta, Altschul, Kececioglu),(Reinert,Lermen)).



Alternatively, exact algorithms are based on a graph-theoretic model that even allows arbitrary gap costs (Kececioglu, Reinert et al, Reinert, Althaus et al.).



und einen guten Rutsch!

7 Multiple Sequence alignment

This exposition is based on the following sources, which are all recommended reading:

1. Reinert, Stoye, Will, An iterative method for faster sum-of-pairs multiple sequence alignment, Bioinformatics, 2000, Vol 16, no 9, pages 808-814.
2. Stoye, Divide-and-conquer Multiple Sequence Alignment, TR 97-02 Universität Bielefeld, 1997

7.1 An exact method

In the following description we will combine three basic algorithmic techniques:

1. divide-and-conquer
2. dynamic programming
3. branch-and-bound

7.2 Divide-and-conquer

The idea is straightforward:

- Cut the k sequences.
- Solve the resulting subproblems recursively.
- The solutions can be combined trivially.
- If the problem size is small enough, an exact algorithm can be employed.

How can we divide the problem into subproblems? Assume this is an optimal global alignment (and $k = 2$, for simplicity):

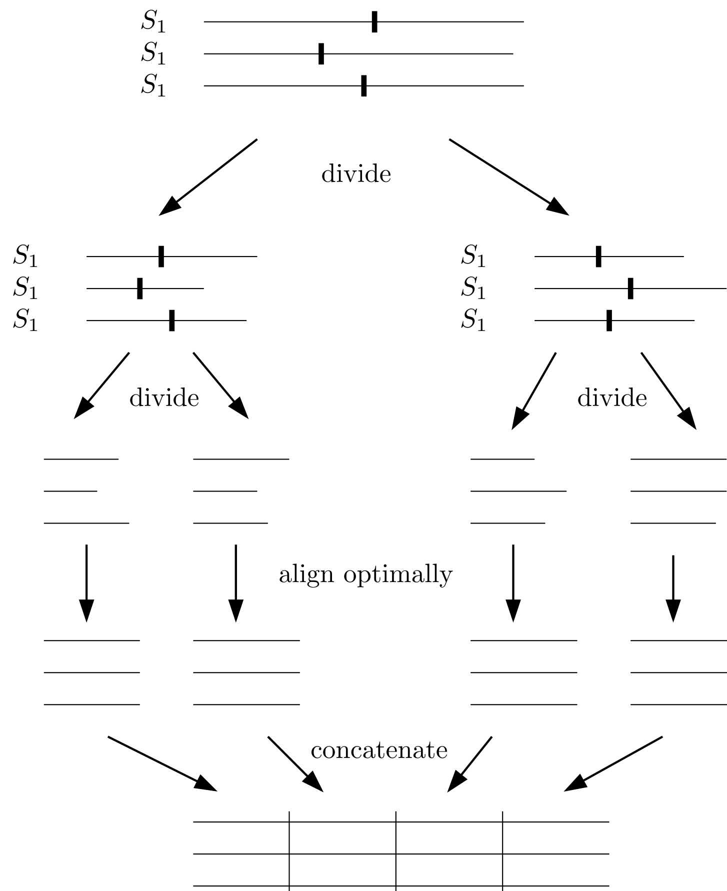
```
GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
 ++ ++++H+ KV   + +A  ++          +L+ L+++H+ K
NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

Then we can *split* this alignment at *any* position and will obtain two optimal global alignments for both sides.

```
GSAQVKGHGKKVADAL        TNAVAHV---D--DMPNALSALSDLHAHKL
 ++ ++++H+ KV           + +A  ++          +L+ L+++H+ K
NNPELQAHAGKVFKLV        YEAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

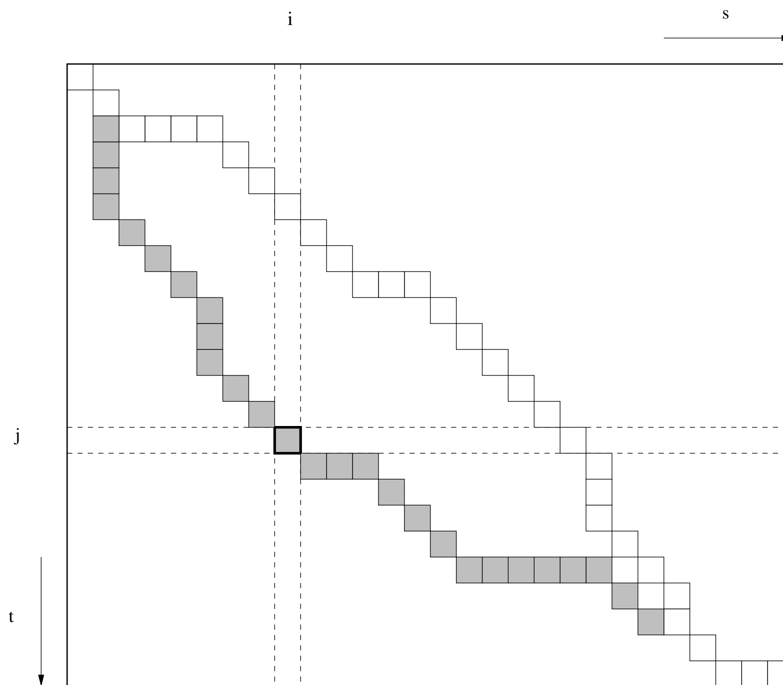
(Really? Exercise!) However, since we do not have a global alignment yet, we also do not know *where* to slit the sequences. We have reduced the problem of finding an optimal alignment to another problem: Finding an optimal cut position.

On the one hand it is clear that optimal cut positions exist, on the other hand it is clear that it is NP-hard to find them. Trivial ideas for determining the cut positions are not very successful. Stoye showed how to compute relatively good cut positions.



7.3 Additional cost

The idea is to compute so called additional cost matrices for each pair of sequences and each possible cut position. The *additional cost* for a position (i, j) is the difference between the optimal global alignment and the best alignment going through position (i, j).



We will define *C-optimal* families of slicing positions such that the (weighted) sum of pairwise additional costs is minimized.

In the following, $\alpha_s^i := s[1..i]$ denotes the i-th prefix of a string s, $\sigma_s^j := s[j{+}1..|s|]$ the j-th suffix of a string s, $\mathcal{A}((s,t))$ denotes the space of all possible alignments between strings s and t, and A ++ B denotes the concatenation of two alignments A and B.

Definition 1. Let s and t be two sequences, and c be an alignment score function. Then for each pair (i, j) of possible *slicing positions* of s and t, $0 \leq i \leq| s |, 0 \leq j \leq| t |$, the *pairwise additional cost* with respect to c is defined by:

$$C_{s,t}[i,j] := \min\left\{c(A \mathbin{++} B) \;\middle|\; A \in \mathcal{A}((\alpha_s^i, \alpha_t^j)), B \in \mathcal{A}((\sigma_s^i, \sigma_t^j))\right\} - c(A^*)$$

The matrix $C_{s,t} := (C_{i,j})_{0 \leq i \leq |s|, 0 \leq j \leq |t|}$ is called the *additional cost matrix* of s and t with respect to c.

The additional cost matrices can easily be computed using the pairwise forward and reverse distance matrices. The forward distance matrix is obtained by the Needleman-Wunsch algorithm for pairwise global alignment. The reverse distance matrix is obtained similarly, but the recursion is carried out in the opposite direction. Then we have

$$C_{s,t}[i,j] = D^f_{s,t}[i,j] + D^r_{s,t}[i,j] - c_{opt}(s,t)\,,$$

where

$$c_{opt}(s,t) = D^f_{s,t}[| s |, | t |] = D^r_{s,t}[0,0]\,.$$

Let us have a look at an example:

Using the unit cost function $d(x, y) = 1 - \delta_{xy}$:

		C	T
	0	1	2
A	1	1	2
G	2	2	2
T	3	3	2

D^f_{s,t}

	C	T	
A	2	2	3
G	1	1	2
T	1	0	1
	2	1	0

D^r_{s,t}

	C	T	
	0	1	3
A	0	0	2
G	1	0	1
T	3	2	0

C_{s,t}

(Exercise: How would this look like for affine gaps costs?).

7.4 Finding slicing positions

We now return to the problem of finding a family of k slicing positions for the sequences $s_1, s_2, \ldots, s_k$. Analogously to the WSOP score we define the *weighted multiple additional cost* imposed by forcing the multiple alignment path of the sequences to visit a particular vertex $(c_1, \ldots, c_k)$ of the k-dimensional hypercube.

Definition 2. Assume we are given a family of k sequences $s_1, \ldots, s_k$ and pairwise factors $w_{i,j}$. Then the *weighted SP multiple additional cost* of a family of slicing positions $(c_1, \ldots, c_k)$ is defined as:

$$C(c_1, \ldots, c_k) = \sum_{1 \le i < j \le k} w_{i,j} C_{s_i,s_j}[c_i, c_j]$$

We use this definition to define *C-optimal* families of slicing positions:

Definition 3. Suppose that we are prescribing a slicing position in one of the sequences, say position $\hat{c}_1$ of sequence s_1. Then a family of slicing positions $(c_2, \ldots, c_k)$ for $s_2, \ldots, s_k$ is called *C-optimal* with respect to $\hat{c}_1$ if the multiple additional cost $C(\hat{c}_1, c_2, \ldots, c_k)$ is minimal. The corresponding minimal additional cost is denoted by:

$$C_{opt} = C_{opt}(\hat{c}_1) := \min \left\{ C(\hat{c}_1, c_2, \ldots, c_k) \;\middle|\; 0 \le c_i \le n_i, \forall i \in \{2, \ldots, k\} \right\}$$

Using this definition one can apply the central heuristic of the divide-and-conquer approach:

By cutting the sequences at C-optimal slicing positions, the concatenation of optimal multiple alignments of the prefix and the suffix sequences yields very good multiple alignments of the original sequences.

We extend the previous example. Consider $s_1 = CT$, $s_2 = AGT$, $s_3 = G$. The corresponding pairwise additional-cost matrices are:

	C	T
0	1	3
A: 0	0	2
G: 1	0	1
T: 3	2	0

C_{s_1,s_2}

	C	T
0	0	1
G: 1	0	0

C_{s_1,s_3}

	A	G	T
0	0	1	2
G: 2	1	0	0

C_{s_2,s_3}

Set all pairwise weights to 1 and assume $\hat{c_1} = 1$. It is not difficult to verify that $(c_2, c_3) = (1, 0)$ is C-optimal with respect to $\hat{c_1}$, since

$$C(1, 1, 0) = C_{s_1,s_2}[1, 1] + C_{s_1,s_3}[1, 0] + C_{s_2,s_3}[1, 0] = 0.$$

Given this cut the best possible alignment has cost 7. The unique optimal alignment, however, has cost 6. It can be achieved with the cut $(2, 1)$ which is also C-optimal with respect to $\hat{c_1}$.

```
C        -T       C-T            -C        T        -CT
A   ++   GT   =   AGT            AG   ++   T    =   AGT
-        G-       -G-            -G        -        -G-

    cut (1,1,0)                      cut (1,2,1)
      cost 7                           cost 6
```

Assume we have a multiple sequence alignment program *MSA* which computes the optimal alignment of k sequences. Then the basic DCA (divide-and-conquer algorithm) looks as follows:

1. DCA($s_1, s_2, \dots, s_k, L$);
2. **if** $\max\{n_1, \dots n_k\} \leq L$
3. **then** return MSA($s_1, s_2, \dots, s_k$);
4. **else** return DCA($\alpha_1^{\hat{c_1}}, \alpha_2^{c_2}, \dots, \alpha_k^{c_k}$) ++ DCA($\sigma_1^{\hat{c_1}}, \sigma_2^{c_2}, \dots, \sigma_k^{c_k}$);
5. **fi**
6. where $\hat{c_1} := \lceil \frac{n_1}{2} \rceil$;
7. $(c_2, \dots, c_k) :=$ C-opt$((s_1, \dots, s_k), \hat{c_1})$;

The function C-opt can naively be implemented as nested loops that run over all possible values $c_i = 0, \dots, n_i$, for all i, and return the slicing position with the minimal multiple additional costs. A direct improvement of this procedure is the combination of the loops with a simple branch-and-bound procedure that cuts off combinations of the $c_2, \dots, c_k$, if already a partial sum of the additional cost term exceeds the minimal cost found so far (exercise).

7.5 Dynamic programming

The straightforward extension to compute an optimal (W)SOP-cost alignment A^* using dynamic programming takes time $O(k^2 2^k N)$ with $N = \prod_i n_i$. Obviously this is only practical for very few, short sequences ($k = 3, 4$). For 10 sequences of length 100 the time and space consumption would be in the order of 10^{25}.

In the following we switch to the *edit graph* representation of a (multiple) alignment. In this representation finding an optimal alignment corresponds to finding a shortest path in a directed acyclic graph (DAG).

All considerations can also be done with affine gap costs, but for the sake of exposition we use linear gap costs.

The graph has the node set

$$V = \left\{ v = (v[1], v[2], \ldots, v[k]) \;\middle|\; v \in \mathbb{N}^k \text{ and } 0 \le v[i] \le n_i, \quad 1 \le i \le k \right\}$$

and the edge set

$$E = \left\{ (p, q) \;\middle|\; p, q \in V,\ p \ne q \text{ and } q - p \in \{0,1\}^k \setminus \{0\}^k \right\}$$

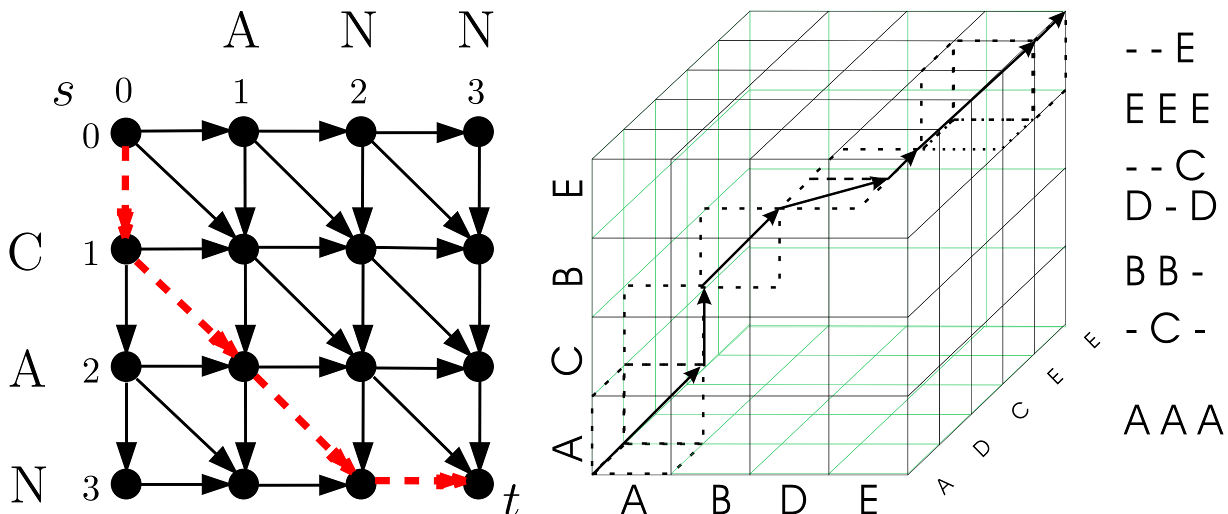
We denote with $p \to q$ the set of all paths from a node p to a node q

$$p \to q := \left\{ (p{=}v_0,\ v_1,\ \ldots\ ,\ v_n{=}q) \;\middle|\; (v_i, v_{i+1}) \in E,\ 0 \le i < n \right\}$$

Analogously we write $p \to q \to r$ for the set of all paths from p to r, which run through node q.

The graph has two special nodes, the *source* $s = (0, 0, \ldots, 0)$, and the *sink* $t = (n_1, n_2, \ldots, n_k)$.

In the below picture you see a two- and a three-dimensional edit graph with source s, sink t and an optimal source-to-sink path $s \to t$.



A path π of length l from $s = (0, \ldots, 0)$ to $t = (n_1, \ldots, n_k)$ corresponds to an alignment defined by the following matrix:

$$A_{ij} = \begin{cases} - & \text{if } v_j[i] - v_{j-1}[i] = 0 \\ s_i[v_j[i]] & \text{if } v_j[i] - v_{j-1}[i] = 1 \end{cases} \qquad \text{for } 1 \le i \le k,\ 1 \le j \le l$$

The cost of an alignment is the sum of the cost of all edges:

$$c(\pi) := \sum_{i=0}^{l-1} c(v_i, v_{i+1})$$

with

$$c(v, w) := \sum_{1 \le i < j \le k} d(a_{i*}, a_{j*})$$

We overload the notation of c, since an edge in the grid graph corresponds exactly to a column in a multiple alignment. We denote the shortest path in $p \to q$ with $p \to^* q$ and its length with $c(p \to^* q)$.

A node r in the grid cuts each sequence s_i into a prefix α_i^r and a suffix σ_i^r.

7.6 Dijkstra's algorithm

Computing the shortest path in a DAG (directed acyclic graph) can be done by traversing the DAG in any topolocial order, which in turn corresponds exactly to the natural extension of dynamic programming.

However, this would imply *constructing* the whole graph which is a major obstacle. Hence we resort to an in theory more costly algorithm, namely Dijkstra's algorithm for graphs with non-negative edge costs.

```
1  Dijkstra(G, s, cost, dist, pred);
2  node_pq<int> Q(G); int c; node u, v; edge e;
3  for ( all nodes v ∈ G) do
4      pred[v] = 0; dist[v] = ∞; Q.insert(v, dist[v]);
5  od
6  dist[s] = 0;
7  Q.decrease_inf(s, 0);
8  while (!Q.empty()) do
9          u = Q.del_min();
10         for all adjacent edges e, u do
11             v = G.target(e); c = dist[u] + cost[e];
12             if c < dist[v]
13                then dist[v] = c; pred[v] = e; Q.decrease_inf(v, c);
14             fi
15         od
16 od
```

As noted before, constructing the complete graph is much too expensive. Hence we have to adapt Dijkstra's algorithm and construct it as needed starting from the source s.

In the priority queue Q we store the values of the shortest paths for the respective prefixes of the sequences. In each step we delete the node q with the minimal value k from Q. This node is then *expanded* which means that all neighboring nodes r of q, that are not already in Q, are inserted into Q with the value $k + c(q, r)$. In case r is already in Q, we relax the triangle inequality.

Dijkstra's algorithm guarantees, that the value of the node u that is removed from the priority queue Q equals the value of the shortest path from s to u.

We discuss three strategies to reduce the number of nodes that are inserted into Q during the expansion of a node.

7.7 Projections

First we need the definition of multiple alignments of subsets of the k strings.

Definition 4. Let A be a multiple alignment for the k strings $s_1, \ldots, s_k$ and $I \subseteq \{1, \ldots, k\}$ be a set of indices defining a subset of the k strings. Then we define A_I as the alignment resulting from first removing all rows $i \notin I$ from A and then deleting all columns consisting entirely of blanks. We call A_I the *projection* of A to I. If I is given explicitely, we simplify notation and write $A_{i,j,k}$ instead of $A_{\{i,j,k\}}$.

Example 5.

```
a1 = - G C T G A T A T A G C T
a2 = G G G T G A T - T A G C T
a3 = - G C T - A T - - C G C -
```

```
a4 = A G C G G A - A C A C C T
```

The projection $A_{2,3}$ is given by first taking the second and third row of the alignment:

```
a2 = G G G T G A T - T A G C T
a3 = - G C T - A T - - C G C -
```

and then deleting the column consisting only of blanks.

```
a2 = G G G T G A T T A G C T
a3 = - G C T - A T - C G C -
```

7.8 Standard-Bounding

Assume we have for each node r of the grid graph a lower bound l for the cost of the shortest path in $r \to t$. For example

$$L(r \to t) := \sum_{1 \le i < j \le k} c(A^*(\sigma_i^r, \sigma_j^r))$$

defines a lower bound, since the projection of an optimal alignment A^* to i and j certainly has a cost no better than the optimal alignment of σ_i and σ_j. Hence

$$c(r \to^* t) \ge \sum_{1 \le i < j \le k} c(A^*(\sigma_i^r, \sigma_j^r)) .$$

Hence, in the expansion of a node q, we do not need to insert a neighbor r into Q if the following holds:

$$c(s \to^* q) + c(q, r) + L(r \to t) > U$$

where U is any upper bound for $c(A^*)$.

That means if the sum of the shortest path to q *plus* the cost of the edge from q to r *plus* a lower bound for a shortest path from r to t is already greater than an upper bound for an optimal alignment, then there cannot exist any optimal alignment through q and r. Hence there is no need to insert r into Q.

7.9 Carillo-Lipman Bounding

The idea of Carillo and Lipman is subsumed in the following theorem which is valid for any optimal alignment A^*.

Theorem 6 (Carillo,Lipman)**.**
Let A^* be an optimal alignment of the k sequences $s_1, \ldots, s_k$. Let $L = L(s \to t)$ be the standard lower bound for $c(A^*)$ and let $U = c(A^{heur})$ be an upper bound for $c(A^*)$. Then the following inequality holds for each projection of a pair i, j of sequences:

$$c(A^*_{i,j}) \le c(A^*(s_i, s_j)) + U - L$$

Proof:

$$\begin{aligned} U - L &\ge \sum_{1 \le i < j \le k} (c(A^*_{i,j}) - c(A^*(s_i, s_j))) \\ &\ge c(A^*_{i,j}) - c(A^*(s_i, s_j)),\ \forall\ 1 \le i < j \le k \\ \Rightarrow c(A^*_{i,j}) &\le c(A^*(s_i, s_j)) + U - L \end{aligned}$$

Applying the above theorem a shortest path cannot go through r if for any pair i, j holds:

$$c(A^*(\alpha_i^r, \alpha_j^r)) + c(A^*(\sigma_i^r, \sigma_j^r)) - c(A^*(s_i, s_j)) + L(s \to t) > U$$

Define $CL_{i,j}(q)$ as the left side of above inequality and call a node r *CL-valid*, if $CL_{i,j}(q) \leq U$ for all pairs i, j. Otherwise r is called *CL-invalid*.

The Carillo-Lipman bound was introduced 1988, but we will show that a combination of Standard-Bounding together with a different edge weighting performs provably better.

7.10 GDUS-Bounding

The so-called Goal-directed-unidirectional-search, also known as the $\mathcal{A}^*$-algorithm tries to direct the computation of the shortest path more into the direction of the sink t.

The algorithm uses a lower bound $l(q \to t)$. First the cost of an edge (q, r) is redefined as follows:

$$c'(q, r) := c(q, r) - (l(q \to t) - l(r \to t))$$

where $l(a \to b)$ is a lower bound for the cost $c(a \to^* b)$ of a shortest path from a nach b ist. If $l()$ fullfills the *consistency*-condition:

$$c(q, r) \geq l(q \to t) - l(r \to t), \; \forall (q, r) \in E$$

then it is easy to show (exercise) that the redefinition of the edge costs does not change the optimal path and that the new edge weights are still positive.

We will use $\ell := L(q \to t)$ as a lower bound. Indeed L fullfills the consistency condition:

$$\begin{aligned} c(q, r) + L(r \to t) &= \sum_{1 \leq i < j \leq k} (c(A^*(\sigma_i^r, \sigma_j^r)) + c(q, r)) \\ &\geq \sum_{1 \leq i < j \leq k} c(A^*(\sigma_i^q, \sigma_j^q)) \\ &= L(q \to t) \end{aligned}$$

The better the lower bound, the more directed the search. In the extrem case, if L is tight, then we extract only the nodes on the optimal path from the priority queue Q. In the other extreme if $L = 0$, we have standard bounding.

Now apply standard bounding on the grid graph with redefined edge costs. With the new edges costs a shortest path from s to q has costs

$$c'(s \to^* q) = c(s \to^* q) + L(q \to t) - L(s \to t).$$

Since $L(s \to t)$ is constant we can omit it and insert a neighboring node r with the value $\textsc{Prio}_q(r)$ into the priority queue Q, where

$$\textsc{Prio}_q(r) := c(s \to^* q) + c(q, r) + L(r \to t)$$

If we apply the standard-bounding technique, we will not insert a node r into Q whenever

$$\textsc{Prio}_q(r) > U$$

where U is an upper bound for $c(s \to^* t)$.

This does not exclude that r might be inserted later into Q, if it is visisted during the expansion of another neighboring node q'. However, if r is never inserted into Q, i. e. if

$$\textsc{Prio}(r) := \min_{(q,r) \in E} \textsc{Prio}_q(r) > U$$

then we call r *U-invalid*, otherwise *U-valid*.

The following theorem shows, that the standard bounding together with the redefined edge costs always exclude at least as many nodes as the Carrillo-Lipmann bound.

Theorem 7. *CL-invalidity implies U-invalidity.*

Proof:

$$\begin{aligned}
\mathrm{CL}_{i,j}(q) &= c(A^*(\alpha_i^q, \alpha_j^q)) + c(A^*(\sigma_i^q, \sigma_j^q)) \\
&\quad - c(A^*(s_i, s_j)) + L(s \to t) \\
&= c(A^*(\alpha_i^q, \alpha_j^q)) + c(A^*(\sigma_i^q, \sigma_j^q)) \\
&\quad - c(A^*(s_i, s_j)) + \sum_{1 \le m < n \le k} c(A^*(s_m, s_n)) \\
&= c(A^*(\alpha_i^q, \alpha_j^q)) + c(A^*(\sigma_i^q, \sigma_j^q)) + \sum_{\substack{1 \le m < n \le k \\ (m,n) \neq (i,j)}} c(A^*(s_m, s_n)) \\
&\le c(A^*(\alpha_i^q, \alpha_j^q)) + c(A^*(\sigma_i^q, \sigma_j^q)) \\
&\quad + \sum_{\substack{1 \le m < n \le k \\ (m,n) \neq (i,j)}} \Big(c(A^*(\alpha_m^q, \alpha_n^q)) + c(A^*(\sigma_m^q, \sigma_n^q)) \Big) \\
&= \sum_{1 \le m < n \le k} c(A^*(\alpha_m^q, \alpha_n^q)) + \sum_{1 \le m < n \le k} c(A^*(\sigma_m^q, \sigma_n^q)) \\
&= \sum_{1 \le m < n \le k} c(A^*(\alpha_m^q, \alpha_n^q)) + L(q \to t) \\
&\le c(s \to^* q) + L(q \to t) \\
&=: \textsc{Prio}(q)
\end{aligned}$$

This means whenever $\mathrm{CL}_{i,j}(q) > U$ for all i, j then also $\textsc{Prio}(q) > U$, which means that q is always U-invalid, if it is CL-invalid.

The proposed version of the algorithm was implemented and the new bounding achieved a speedup of up to twenty times over the Carrillo Lipman bounding. An affine implementation combined with the divide-and-conquer strategy would be a masters project.

7.11 Combining it all

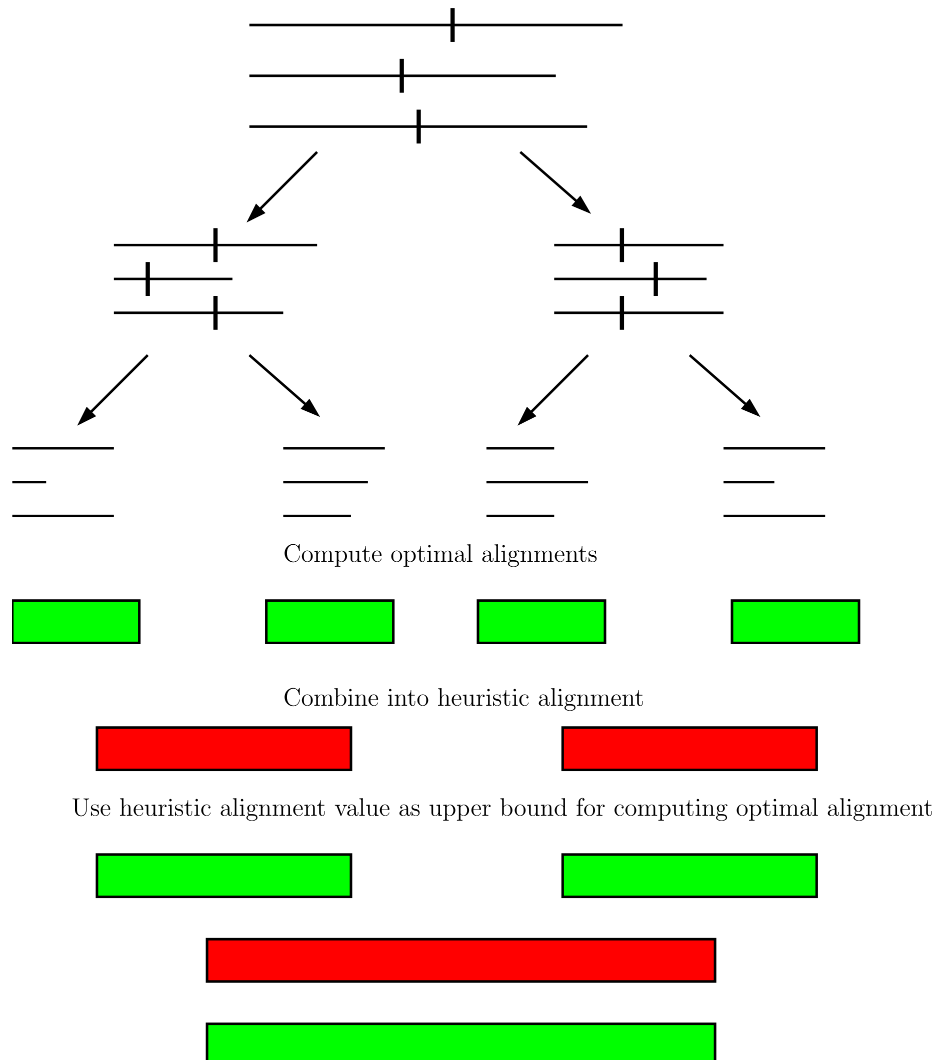
The divide-and-conquer Algorithm (DCA) is called with a stop length L at which the recursion stops. Obviously each DCA alignment is a heuristic alignment giving an upper bound U. The better the upper bound U the better our branch-and-bound algorithm works.

On the other hand, we would like to stop the DCA recursion early, since then we can expect a near to optimal alignment while the computation time increases due to the larger optimal alignments to be computed.

This motivates an iterative combination of both DCA and the optimal alignment procedure: Call DCA with a small value of L. Then, for the small alignments compute the optimal branch-and-bound alignment.

Combine these alignments. The resulting alignment is now directly a rather good heuristic alignment for the combined block. Hence its value can be used as an upper bound for the branch-and-bound algorithms who can "correct" the heuristic alignment into an optimal one.

We can continue this strategy until we reach the last division, or we can stop at any point of this procedure and have a heuristic alignment of quality that is at least as good as the original DCA. The longer we wait, the better is the alignment – up to optimal.



7.12 Summary

- Divide-and-conquer alignment needs the computation of good cut positions and a modul to compute an optimal sum-of-pair alignment.
- The obvious generalization of the DP based alignment algorithm is not practical.
- If we view the DP matrix as a *k*-dimensional edit graph we can apply standard branch-and-bound techniques to the graph.
- We build the graph on the fly and explore it using Dijkstra's algorithm.
- The standard branch-and-bound can be augmented by Carillo-Lipman bounding.
- A redefinition of the edge weights results in a goal directed search which is provably superior to the Carillo-Lipmann bound.