6 Advanced Multiple Alignment (script by K. Reinert)

This exposition is based on the following sources, which are 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

In this chapter we discuss how to combine three different algorithmic techniques to address the problem of computing good multiple sequence alignments:

- divide-and-conquer,
- dynamic programming and
- branch-and-bound.

First we will describe the *DCA (divide-and-conquer)* method. then we will discuss the naïve dynamic programming approach and will show how to speed it up using branch-and-bound. Finally, we will discuss a hybrid method that uses both DCA and the branch-and-bound approach.

6.1 The DCA method

Given a set of k sequences $s_1 \ldots s_k$ for which we would like to compute a multiple alignment, DCA is based on the following idea:

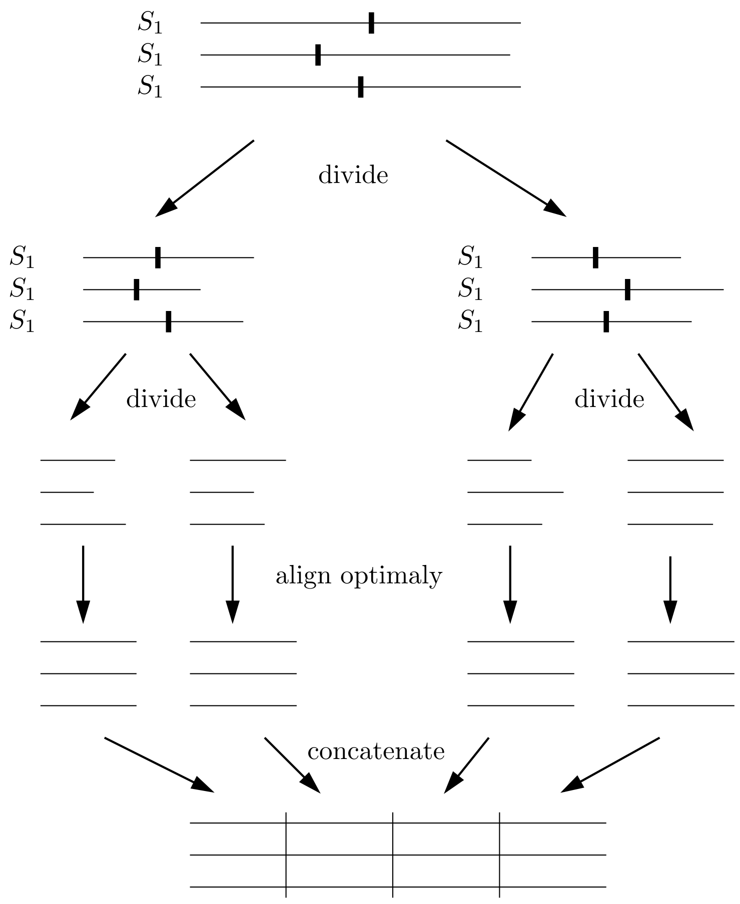
> Cut each of the k sequences and solve the resulting subproblems recursively. When the problem size is small enough, use an exact algorithm.

The solutions can be combined simply by concatenation.

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

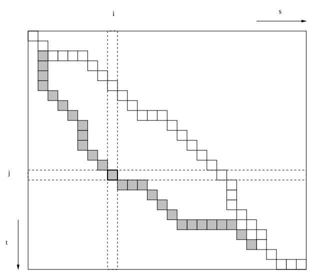
Trivial ideas for determining the cut positions are not very successful. Hence, the main question is how to choose the cut position.

This illustrates the main idea:



To help determine good cut sites, we will make use of so-called *additional cost* matrices for each pair of sequences and each possible cut position.

The additional cost is the score difference between the optimal global alignment and the best alignment going through position i and j.

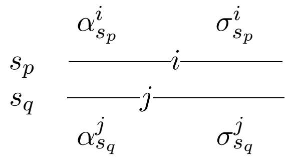


Based on this, we will determine *C-optimal* families of cut positions, for which the (weighted) sum of pairwise additional costs is minimized.

To define this more formally, assume we are given two sequences $s_p = s_{p1} \dots s_{pn}$ and $s_q = s_{q1} \dots s_{qm}$.

Let c denote a cost function and let A^* denote an optimal alignment of s_p and s_q.

Let (i, j) be a pair of *possible cut positions* in s_p and s_q with $0 \leq i \leq n$ and $0 \leq j \leq m$. We use α_s^i or σ_s^i to denote the i-th prefix, or suffix, of a string s, respectively:



If A is an alignment of a prefix of s_p to a prefix of s_q, and if B is an alignment of the two remaining suffixes, then let A++B denote the concatenation of the two alignments.

Definition For each pair (i, j) of possible cut positions of s_p and s_q we define the *pairwise additional* *cost* with respect to c by:

$$\begin{aligned} C_{s_p,s_q}[i,j] \;&:=\; \min\{c(A{+}{+}B) \mid A \in \mathcal{A}(\alpha^i_{s_p}, \alpha^j_{s_q}), B \in \mathcal{A}(\sigma^i_{s_p}, \sigma^j_{s_q})\} \\ &\quad -c(A^*), \end{aligned}$$

where $\mathcal{A}(a,b)$ denote the set of all possible alignments of two sequences a and b.

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

The additional cost matrices can be easily computed using the "forward" and "reverse" pairwise alignment matrices, that is,

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

where

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

by definition.

Lets have a look at an example:

Let $s = \texttt{CT}$ and $t = \texttt{AGT}$. We use the unit cost function $d(x,y) = 1 - \delta_{xy}$ and a gap penalty of 1:

		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}

As stated above, the main problem is how to determine a good family of k cut 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 through a particular vertex $(c_1, \ldots, c_k)$ in the k-dimensional hypercube.

Definition Suppose that we are given a family of k sequences $s_1, \ldots, s_k$ and pairwise weights $w_{i,j}$. The *weighted SP multiple additional cost* of a family of cut 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 now can define *C-optimal* families of cut positions:

Definition Suppose that we are giving a cut position of one of the sequences, say position $\hat{c_1}$ of sequence s_1. A family of cut positions $(c_2, \ldots, c_k)$ of $s_2, \ldots, s_k$ 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:

$$C_{opt} = C_{opt}(\hat{c_1}) := \min\{C(\hat{c_1}, c_2, \ldots, c_k) \mid 0 \le c_i \le n_i, \forall i \in \{2, \ldots, k\}\}$$

The DCA approach is justified as follows:

Main assumption By cutting the sequences at C-optimal cut 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:

$$\begin{array}{c|ccc} & {}_0\,\mathrm{C}\,{}_1 & \mathrm{T}\,{}_2 & \\ \hline {}_0 & 0 & 1 & 3 \\ \mathrm{A}\ {}_1 & 0 & 0 & 2 \\ \mathrm{G}\ {}_2 & 1 & 0 & 1 \\ \mathrm{T}\ {}_3 & 3 & 2 & 0 \end{array}$$

C_{s_1,s_2}

$$\begin{array}{c|ccc} & {}_0\,\mathrm{C}\,{}_1 & \mathrm{T}\,{}_2 & \\ \hline {}_0 & 0 & 0 & 1 \\ \mathrm{G}\ {}_1 & 1 & 0 & 0 \end{array}$$

C_{s_1,s_3}

$$\begin{array}{c|cccc} & {}_0\,\mathrm{A}\,{}_1 & \mathrm{G}\,{}_2 & \mathrm{T}\,{}_3 & \\ \hline {}_0 & 0 & 0 & 1 & 2 \\ \mathrm{G}\ {}_1 & 2 & 1 & 0 & 0 \end{array}$$

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 has cost 6 and 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 that we can use to solve small instances of the problem of aligning k sequences. The DCA algorithm can be summarized as follows:

```
**Algorithm** DCA($s_1, s_2, \ldots, s_k, L$)
Input: sequences $s_1, \ldots, s_k$ and cut-off $L$
Output: alignment of $s_1, \ldots, s_k$
begin
    if $\max\{n_1, \ldots n_k\} \leq L$ then
      return MSA($s_1, s_2, \ldots, s_k$)
    else
      Set $\hat{c_1} := \lceil \frac{n_1}{2} \rceil$
      Set $(c_2, \ldots, c_k) :=$ C-opt($(s_1, \ldots, s_k), \hat{c_1}$)
      return DCA($\alpha_{s_1}^{\hat{c_1}}, \alpha_{s_2}^{c_2}, \ldots, \alpha_{s_k}^{c_k}$) ++DCA($\sigma_{\hat{c_1}}^1, \sigma_{c_2}^2, \ldots, \sigma_{c_k}^k$)
end
```

The function C-opt can naively be implemented as nested loops that run over all possible values $0, \ldots, n_i, \forall i$ and return the cut position with the minimal multiple additional costs.

The computation of C-opt requires at most $O(Nk^2n^2)$ steps, with $n = \max_i\{|s_i|\}$ and $N = \prod_i |s_i|$.

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 $c_2, \ldots, c_k$, if already a partial sum of the additional cost term exceeds the minimal cost found so far.

6.2 Naïve Dynamic programming

The straightforward extension of the Needleman-Wunsch algorithm to compute an optimal (W)SOP-cost alignment A^* for k sequences $s_1, \ldots, s_k$ using dynamic programming takes time $O(n^k)$, with $n = \min_i |s_i|$. Obviously this is only practical for very few, short sequences ($k = 3, 4$)

Our goal now is to develop a branch-and-bound approach.

In the following we will use 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.

Given sequences $s_1, \dots, s_k$. The corresponding *edit graph* has nodes

$$V = \left\{ v = (v[1], v[2], \dots, v[k]) \mid v \in \mathbb{N}^k \text{and } 0 \leq v[i] \leq n_i,\ 1 \leq i \leq k \right\}$$

and edges

$$E = \left\{ (p, q) \mid p, q \in V,\ p \neq q \text{ und } q - p \in \{0,1\}^k \setminus \{0\}^k \right\}.$$

We use:

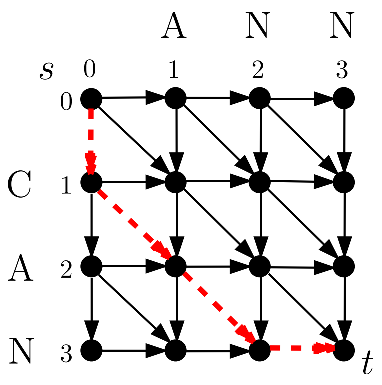
$$p \to q := \left\{ (p = v_0,\ v_1,\ \dots\ ,\ q = v_n) \mid (v_i, v_{i+1}) \in E,\ 0 \leq i < n \right\}$$

to denote the set of all paths from node p to q.

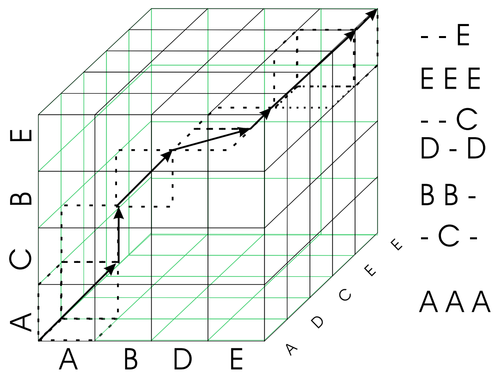
Similarly, we write $p \to q \to r$ for the set of all paths from p to r, which run through node q.

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

The following example shows a two-dimensional edit graph with source s, sink t and an optimal source-to-sink path $s \to t$ in dashed lines.



Another example:



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

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

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

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

We overload the notation of c, since an edge in the edit 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 $c(p \to^* q)$.

Any node v in the edit graph cuts each sequence s_i into a prefix $\alpha_i^v := \alpha_{s_i}^{v[i]}$ and a suffix $\sigma_i^v := \sigma_{s_i}^{v[i]}$, for $i = 1, \ldots, k$.

6.3 Dynamic programming and branch-and-bound

The edit graph is a *DAG (directed acyclic graph)* and a shortest path can be found by traversing the DAG in any topological order. This corresponds exactly to the natural extension of dynamic programming.

However, we do not want to construct the whole graph, as it has exponential size. To avoid this problem, we will take Dijkstra's algorithm, which efficiently finds a shortest path in a graph with non-negative edge costs, and modify it so that (hopefully only) necessary parts of the graph are created on the fly.

Algorithm [Dijkstra]

```
Input: Graph $G = (V, E)$, source node $s$, edge cost $c$
Output: predecessor map $\pi$ defining shortest path from $s$
for each node $v$ do
    Set $d[v] := \infty$ // initial distance from source
    Set $\pi[v] := 0$
Set $d[s] := 0$
Set $Q = V$
while $Q \neq \emptyset$ do
    $u :=$ ExtractMin($Q$) // extract $\arg\min_{v \in Q}\{d[v]\}$
    for each child $v$ of $u$ do
        if $d[v] > d[u] + c(u, v)$ then // relax
            $d[v] := d[u] + c(u, v)$
            $\pi[v] := u$
```

6.4 Branch-and-bound

As noted above, we want to modify the algorithm so that only necessary parts of the graph are explicitly constructed, in the context of branch-and-bound.

Suppose we are given an upper bound U for the length of the shortest path. We modify the algorithm as follows.

- Initially, we set $Q := \{s\}$, not $Q := V$.
- After extracting a node u of minimal distance from Q, we must *expand* u, that is, create all children of u and for any such child v, insert it into Q, if $d[u] + c(u, v) \leq U$ holds.

Algorithm [Modified Dijkstra with upper bound U]

```
Input: Graph $G = (V, E)$, source node $s$, edge cost $c$, upper bound $U$
Output: predecessor map $\pi$ defining shortest path from $s$
begin
Create node $s$.
Set $d[s] := 0$
Set $Q := \{s\}$
while $Q \neq \emptyset$ do
        $u :=$ ExtractMin($Q$) // extract $\arg\min_{v \in Q}\{d[v]\}$
        for each potential child $v$ of $u$ do
        if $d[u] + c(u, v) \leq U$ then // bound
          if $v$ has not yet been created then // create
              Create $v$
              Set $d[v] := \infty$
          if $d[v] > d[u] + c(u, v)$ then // relax
              Set $d[v] := d[u] + c(u, v)$
              Set $\pi[v] := u$
end
```

6.5 Projections

We will discuss how to obtain a useful bound U.

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

Definition 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

```
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 -
```

6.6 Standard-Bounding

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

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

defines a lower bound, since the projection of an optimal alignment A^* to i and j certainly has a cost larger than the optimal alignment of σ_i and σ_j. In other words,

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

Hence, in the expansion of a node u, we only have to consider a child v, if the following holds:

$$c(s \to^* u) + c(u, v) + L(v \to t) \leq U,$$

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

6.7 Carillo-Lipman Bounding

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

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

By this theorem, a shortest path cannot go through a node v, if for any pair i, j holds:

$$c(A^*(\alpha^v_i, \alpha^v_j)) + c(A^*(\sigma^v_i, \sigma^v_j)) - c(A^*(s_i, s_j)) + L(s \rightarrow t) > U.$$

Define $CL_{i,j}(v)$ as the left side of above inequality and call a node v *CL-valid*, if $CL_{i,j}(v) \leq U$ for all pairs i, j. In the expansion of a node u, only consider a child v if it is CL-valid.

6.8 Carillo-Lipman Bounding

Proof

1. We have $L \leq \sum_{1 \leq i < j \leq k} c(A^*(s_i, s_j))$.
2. We have $\sum_{1 \leq i < j \leq k} c(A^*_{i,j}) \leq U$.
3. Thus,

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

because all terms are ≥ 0. □

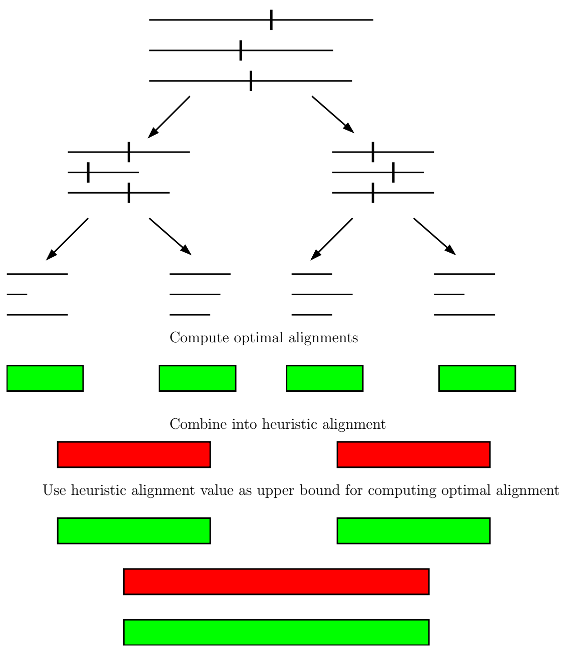
6.9 Combining DCA and the branch-and-bound Approach

The divide-and-conquer algorithm (DCA) is called with a stop length L at which the recursion stops. Each DCA alignment is a heuristic alignment giving an upper bound U. The better the upper bound U is, the better the branch-and-bound algorithm will work.

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 algorithm that can "correct" the heuristic alignment into an optimal one.
- Continue this strategy until the last division is reached, or 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 one waits, the better is the alignment – up to optimal.


Compute optimal alignments
Combine into heuristic alignment
Use heuristic alignment value as upper bound for computing optimal alignment

6.10 Summary

- Divide-and-conquer alignment needs the computation of good cut positions and a module to compute an optimal sum-of-pair alignment.
- The naïve generalization of the Needleman-Wunsch algorithm to more than two sequences 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 can build the graph on the fly and explore it using a modification of Dijkstra's algorithm.
- The standard branch-and-bound can be augmented by Carillo-Lipman bounding.
- The DCA and branch-and-bound techniques can be combined to provide a hybrid algorithm.