Multiple Genome Alignment

This exposition is based on:

1. Michael Höhl, Stefan Kurtz, Enno Ohlebusch: Efficient multiple genome alignment, Bioinformatics, vol. 18, suppl. 1, 2002, S312–S320.

2. Documentation of the MGA tool:
   `http://bibiserv.techfak.uni-bielefeld.de/mga/doc/`

3. For the chaining problem, see e. g. Dan Gusfield: Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997. Section 13.3.

We will present the algorithms used in the **MGA** tool for multiple sequence alignment.

Introduction

Algorithms for multiple sequence alignment are generally based on the following principles:

1. Iterative pairwise alignment (`ClustalW`, `T-Coffee`, ...)

   Iteratively, in a tree-like fashion, two multiple alignments for two disjoint subsets of sequences are merged into a single multiple alignment for the union of these subsets. (+ many extra heuristics)

   However, this does not scale to genomic size.

2. Anchor-based multiple alignment (`MUMmer`, `MGA`)

   MGA uses stretches of *identical* bases occuring in *all* input sequences to "anchor" the multiple alignment.

   This is particularly efficient if the genomic sequences are fairly similar.

Outline of the MGA algorithm

1. Find a set of *multiple exact matches*. The MGA algorithm uses multiMEMs (to be defined below).

2. Select an optimal chain of matches among all matches found. The multiple alignment is *anchored* at these matches, that is, the chained matches are considered aligned.

3. Close the gaps between the chained matches using *recursive calls* of the same algorithm (but with weaker parameters for anchors).

4. The remaining gaps are:

   (a) *handed over* to another alignment tool (e. g., ClustalW), if they are "short", or

   (b) *left open*, if they are "long".

Maximal Multiple Exact Matches

We are given $k \geq 2$ genome-sized sequences $G_0, \dots, G_{k-1}$. Characters are indexed from 0:

$$G = G[0]G[1] \dots G[n-1] = G[0 \mathrel{..} n-1], \quad |G| = n\,.$$

A *multiple exact match* is a $(k+1)$-tuple $(l, p_0, \dots, p_{k-1})$ such that the same substring of length l starts at position p_q in genome G_q, for all $q \in [0 \mathrel{..} k-1]$:

- $l > 0$
- $\forall q:\ p_q \in [0 \mathrel{..} \left|G_{k-1}\right| - l]$
- $\forall q, q':\ G_q[p_q \mathrel{..} p_q{+}l{-}1] = G_{q'}[p_{q'} \mathrel{..} p_{q'}{+}l{-}1]$

Example – MultiMEMs (2)

Here is a small example:

G_0 : a c g a c a t a g g g a c g c c c t c a $\$_0$ (positions: 0 1 2 3 4 … 12)

G_1 : t g a t c g c t a t c c g a c g a a a t t t $\$_1$ (positions: 21 … 25 … 32 … 35)

G_2 : t c g t t g a g t a g a g a c c g c t c a $\$_2$ (positions: 44 45 … 59)

G_3 : a c c t c c a c a a t g c g t c $\$_3$ (positions: 66 … 78)

The string positions are shown with respect to the concatenated string $S := G_0\$_0G_1\$_1 \ldots \$_{k-2}G_{k-1}$, which will be used later. In this example, $(2, 12, (32-21), (45-44), (78-66)) = (2, 12, 11, 1, 12)$ is a multiple exact match. (Can it be extended to the left or to the right?)

MultiMEMs (3)

A multiple exact match $(l, p_0, \dots, p_{k-1})$ is *left maximal* if it cannot be extended to the left, that means one of these two conditions is true:

- $\exists q : \; p_q = 0$, or
- $\exists q, q' : G_q[p_q - 1] \neq G_{q'}[p_{q'} - 1]$.

MultiMEMs (4)

A multiple exact match $(l, p_0, \dots, p_{k-1})$ is *right maximal* if it cannot be extended to the right, that means one of the following two conditions is true:

- $\exists q: \; p_q + l = |G_q|$, or
- $\exists q, q': G_q[p_q + l] \neq G_{q'}[p_{q'} + l]$.

A multiple exact match is *maximal* if it is left maximal and right maximal. A maximal multiple exact match is called a *multiMEM*.

(For $k = 2$ multiMEMs have been called *MEM* = maximal exact match).

Understanding MultiMEMs (5)

The maximality of multiMEMs only means that we cannot extend it to the left or to the right by adding *whole columns* in a multiple alignment. Indeed, the following *is* a multiMEM:

t a t g a c g c g t t g

c c a c g c g t g

...although the matched positions are "contained" in a much larger multiMEM:

t a t g a c g c g t t g

c c a c g c g t g

Some notation

The algorithm uses the concatenation of all genomes, separated by special characters.

Let $\$_0, \dots, \$_{k-1}$ be new symbols, not contained in the alphabet Σ.

Let $S := G_0\$_0G_1\$_1 \dots \$_{k-2}G_{k-1}$ and $n := |S|$.

Let $S_i := S[i \,..\, n-1]\$_{k-1}$ denote the i-th suffix of $S\$_{k-1}$.

Some more notation... (2)

Next we define a kind of "coordinate system" on S.

Let t_q denote the starting position of G_q in S, and $t_k := n+1$.

Let $\sigma(i) := q$ if the i-th letter of S belongs to G_q, that is, $t_q \leq i < t_{q+1} - 1$, and undefined otherwise.

Let $\varrho(i) := i - t_{\sigma(i)}$ be the relative position of i in the genome $G_{\sigma(i)}$.

These definitions are made such that

$$G_q[j] = S[t_q + j]$$

and

$$S[i] = G_{\sigma(i)}[\varrho(i)] .$$

Even more notation... (3)

Let $P_u(q)$ denote the set of indices i such that u is a prefix of S_i and $\sigma(i) = q$, that means the set of all starting positions of u in the specific genome G_q.

Look at this part of the previous example:

	44	45														59						
G_2 :	t	c	g	t	t	g	a	g	t	a	g	a	g	a	c	c	g	c	t	c	a	$\$_2$

Here we have $t_2 = 44$, $\sigma(59) = 2$, $\varrho(59) = 59 - 44 = 15$, and $P_{\texttt{cg}}(2) = \{45, 59\}$.

Finding MultiMEMs

So how can we *find* multiMEMs?

MGA uses a *suffix tree* T for $S\$_{k-1}$, which can be computed in $O(n)$ time. (Actually, the real implementation uses a suffix array, but here we will explain the algorithm using suffix trees.)

The nodes of the suffix tree T will be denoted with a bar, e. g. $\bar{u}$, where u is the string which is the concatenation of the edge labels on the path from the root of T to $\bar{u}$. (Thus u is the path label of $\bar{u}$.)

The *leaves* of T are the nodes $\overline{S(i)}$, corresponding to the suffixes of $S\$_{k-1}$.

Finding MultiMEMs (2)

Now we show how to compute the position sets $P_u(q)$. The suffix tree T is traversed in DFS order starting from the root = $\bar{\varepsilon}$.

We write $\bar{u} \rightarrow \bar{w}$ if there is an edge from $\bar{u}$ to $\bar{w}$ in T directed away from the root. (This implies $|u| < |w|$.) An edge $\bar{u} \rightarrow \bar{w}$ is *processed* only after *all* edges in the subtree below $\bar{w}$ have been processed. (That is, we perform a *bottom-up traversal*.)

The definition of "$\rightarrow$" implies $P_u(q) = \bigcup\{P_w(q) : \bar{u} \rightarrow \bar{w}\}$.

Finding MultiMEMs (3)

Finding MultiMEMs (4)

The *leaf case* is $u = S(i)$. There we have

$$P_u(q) = \begin{cases} \{i\} & \text{if } \sigma(i) = q, \\ \emptyset & \text{otherwise.} \end{cases}$$

Thus we know how to initiate the bottom-up traversal.

Finding MultiMEMs (5)

For *branching nodes*, we compute a series of position sets $P_u^1(q), \dots, P_u^j(q), \dots, P_u(q)$ by *“processing”* the outgoing edges one after another.

Here j is the number of successor edges we have processed so far. The sets $P_u^j(q)$ are intermediate results. The last one, $P_u(q)$, is the result which is handed on to the parent node.

Let $\bar{u} \rightarrow \bar{w}$ be the next edge to be processed. We compute P_u^{j+1} by combining $P_u^j(q)$ and $P_w(q)$. At the same time, we detect multiMEMs and output them *on the fly*.

Finding MultiMEMs (6)

We enumerate all “candidate” $(k+1)$-tuples $(|u|, p_0, \dots, p_{k-1})$ such that:

1. $\forall q: \; p_q \in P_u^j(q) \cup P_w(q)$
2. $\exists q: \; p_q \in P_u^j(q)$
3. $\exists q: \; p_q \in P_w(q)$

Finding MultiMEMs (7)

We enumerate all "candidate" $(k + 1)$-tuples $(|u|, p_0, \ldots, p_{k-1})$ such that:

1. $\forall q : \; p_q \in P_u^j(q) \cup P_w(q)$
2. $\exists q : \; p_q \in P_u^j(q)$
3. $\exists q : \; p_q \in P_w(q)$

Why?

Finding MultiMEMs (8)

We enumerate all “candidate” $(k+1)$-tuples $(|u|, p_0, \dots, p_{k-1})$ such that:

1. $\forall q\colon\ p_q \in P_u^j(q) \cup P_w(q)$
2. $\exists q\colon\ p_q \in P_u^j(q)$
3. $\exists q\colon\ p_q \in P_w(q)$

$\Rightarrow$ Condition (1.) implies that $(|u|, \varrho(p_0), \dots, \varrho(p_{k-1}))$ is a *multiple exact match*.

Finding MultiMEMs (9)

We enumerate all “candidate” $(k+1)$-tuples $(|u|, p_0, \ldots, p_{k-1})$ such that:

1. $\forall q\colon\; p_q \in P_u^j(q) \cup P_w(q)$
2. $\exists q\colon\; p_q \in P_u^j(q)$
3. $\exists q\colon\; p_q \in P_w(q)$

$\Rightarrow$ Conditions (2.) and (3.) guarantee that not all positions p_q are taken exclusively from $P_u^j(q)$ or $P_w(q)$. At least two of them belong to different subtrees of $\bar{u}$.

Hence u is the consensus of a *right maximal* multiple exact match $(|u|, \varrho(p_0), \ldots, \varrho(p_{k-1}))$.

Finding MultiMEMs (10)

We enumerate all “candidate” $(k+1)$-tuples $(|u|, p_0, \ldots, p_{k-1})$ such that:

1. $\forall q\colon\ p_q \in P_u^j(q) \cup P_w(q)$
2. $\exists q\colon\ p_q \in P_u^j(q)$
3. $\exists q\colon\ p_q \in P_w(q)$

$\Rightarrow$ To enforce *left maximality*, we *reject* a tuple $(|u|, \varrho(p_0), \ldots, \varrho(p_{k-1}))$ if it turns out that $p_0 > 0$ and $S[p_0 - 1] = \ldots = S[p_{k-1} - 1]$.

Otherwise it is a (left *and* right) maximal multiple exact match, and we output it on the fly.

Finding MultiMEMs (11)

When all multiMEMs for $\bar{u} \rightarrow \bar{v}$ have been written out, we set $P_u^{j+1}(q) := P_u^j(q) \cup P_w(q)$ and proceed to the next child edge. After all children of $\bar{u}$ have been processed, we go on to the parent of $\bar{u}$.

So far we have seen that all outputs are multiMEMs.

Conversely, every multiMEM is output exactly once: Let u be its consensus. The point of time when a multiMEM is output comes, when the last successor edge $\bar{u} \rightarrow \bar{w}$ is processed such that w is an extension of the multiMEM consensus u to the right in one of the genomes.

Thus when the algorithms leaves $\bar{u}$, all multiMEMs for the string u have been output exactly once.

This proves the correctness of the algorithm.

Running Time and Space Requirement

Now we estimate the running time and space requirements.

The position sets $P_u(q)$ can be implemented as linked lists. This way they can be merged in constant time. There are $O(n)$ nodes, and each node carries k position sets. Therefore the total space and time requirement for the position sets is $O(kn)$.

Running Time and Space Requirement (2)

The enumeration of the set of $(k+1)$-tuples $(|u|, p_0, \ldots, p_{k-1})$ takes time proportional to its size (exercise). Recall that $(|u|, \varrho(p_0), \ldots, \varrho(p_{k-1}))$ is always a right maximal multiple exact match due to conditions (1.)–(3.). Hence the running time of the enumeration is $O(r)$, where r denotes the number of right maximal multiple exact matches. The total running time is $O(kn + r)$.

Running Time and Space Requirement (3)

However we can avoid first generating the right maximal multiple exact matches which are not left maximal and then rejecting them afterwards. Instead we can use a more clever enumeration which produces multiMEMs directly.

The idea is to partition the position sets $P_u(q)$ further into subsets $P_u(q, x)$ according to the “left” character x immediately preceeding the occurences of u in G_q.

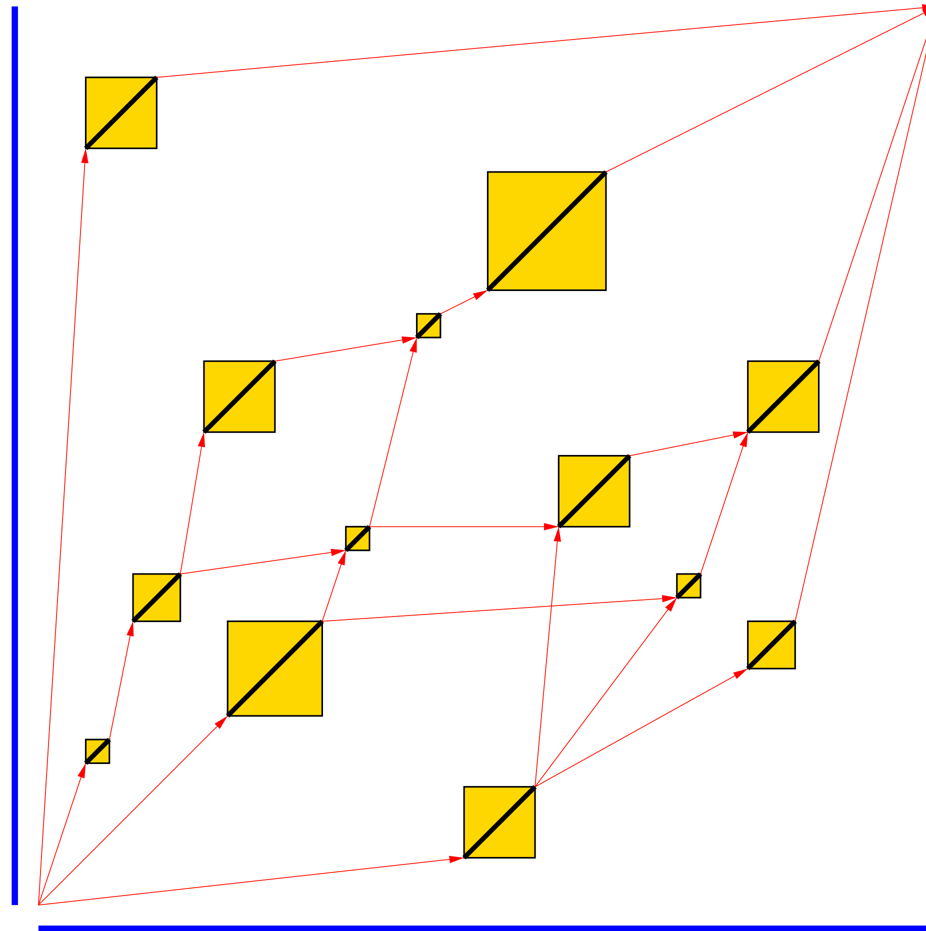
This leads to a total running time of $O(\#\Sigma kn + m)$, where m is the number of multi-MEMs. The details shall be worked out in the exercises.

Finding an Optimal Chain of MultiMEMs

We have found a list of multiMEMs. Naturally, very short MultiMEMs are put aside from consideration. The next step in MGA is to find an optimal set of multiMEMs to be used as anchors.

Finding an Optimal Chain of MultiMEMs (2)

This is, however, instance of the “chaining problem”:



Finding an Optimal Chain of MultiMEMs (3)

The multiMEMs (in the figure, we have only two sequences) become vertices of a graph, and an edge is drawn between two multiMEMs whenever they can both be present in a multiple alignment. The edge is directed from left to right with respect to the aligned sequences. For multiMEMs

$$M = (l, p_0, \dots, p_{k-1}), \quad M' = (l', p'_0, \dots, p'_{k-1})$$

we have

$$(M \to M') \in E \iff \forall q : p_q + l \leq p'_q \, .$$

Moreover we add artificial source and sink vertices at the left and the right end of the sequences. (One could think of them as multiMEMs of length 0.)

Every vertex has a weight corresponding to the length of the multiMEM. The edges are unweighted.

Since the graph $G = (V, E)$ is directed and acyclic, a heaviest source-sink path can be found by dynamic programming (using a topological ordering of the graph) in $O(V + E)$ time.

Finding an Optimal Chain of MultiMEMs (4)

Deciding whether two multiMEMs are consistent with an alignment takes $O(k)$ time for k sequences. Thus the running time is dominated by the construction of the graph, which is $O(km^2)$ for m multiMEMs.

Note, however that an edge is never used in the heaviest directed source-sink path if it is possible to put another vertex in between.

Moreover one can exploit the geometric nature of the problem. An algorithm by Myers and Miller (1995) for the chaining problem runs in $O(m \log^k m)$ time and $O(km \log^{k-1} m)$ space, and recently this was further improved by Abouelhoda and Ohlebusch (2003, CPM) to $O(n \log^{k-2} n \log \log n)$ time and $O(n \log^{k-2} n)$ space.

Closing the Gaps

In the third phase, MGA closes the gaps between the anchors.

This is done by recursively applying the same method a certain number of times, but each time with a lower threshold for the multiMEM size. The gaps that are still left over are handled as follows:

- "Short" gaps are closed by an external multiple sequence alignment program (e. g., ClustalW – but other choices are possible).
- "Long" gaps remain unaligned. This is a "feature", not a "bug":

  "[Leaving long gaps open] makes sense biologically: since there are no matches exceeding the length threshold, there is no detectable similarity and the gaps are not forced into an alignment. This way, mga can cope with long insertions, deletions, etc. [Note (cg): but not with reversals!], *retaining its overall efficiency."*

(Cited from `http://bibiserv.techfak.uni-bielefeld.de/mga/doc/`)

ClustalW outline

We close with an outline of the ClustalW algorithm:

1. All pairs of sequences are aligned separately in order to calculate a distance matrix giving the divergence of each pair of sequences.

2. A guide tree is calculated from the distance matrix.

3. The sequences are progressively aligned according to the branching order in the guide tree.

Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson, CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice, Nucleic Acids Research, Vol. 22, No. 22, p. 4673–4680, 1994.

(See also the ClustalX program, a convenient graphical user interface for ClustalW)