Bioinformatics: Fragment Assembly

Walter Kosters, Universiteit Leiden

IPA—Algorithms&Complexity, 29.6.2007

www.liacs.nl/home/kosters/

We study the following problem from bioinformatics:

> Given several strings of DNA, construct the “best” superstring that contains them all.

Here, “contains” means “as (consecutive) substring”. Note that concatenation does this job — but not so well. Sometimes the approximate total length is known.

Literature: J. Setubal and J. Meidanis, Introduction to Computational Molecular Biology, PWS Publishing Company, Boston, 1997; and D. Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge Universtiy Press, 1997.

Intermezzo   **Tools**

The problem originates from the so-called shotgun method, that breaks a given piece of DNA (or rather many copies of that) into several smaller pieces, that can be "sequenced". The goal is to reconstruct the original DNA-string.
There are many practical problems. We focus on some theoretical issues.

Another approach to obtain this goal is sequencing by hybridization (SBH), see later.

Fragment assembly **Example**

We start with the strings `ACCGT`, `CGTGC`, `TTAC` and `TACCGT` over the 4-base alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. One possible way to assemble them is by the following layout:

```
ACCGT        --ACCGT--
CGTGC        ----CGTGC
TTAC         TTAC-----
TACCGT       -TACCGT--
             _________
             TTACCGTGC
```

We use the overlaps as much as possible. The sequence below the ——— is called the consensus (sequence), obtained from a majority vote among the bases in each column.

Complications — Errors 1

There can be errors in the fragments. The simplest ones are base call errors: base substitutions, insertions and deletions (and so also transpositions). A first example:

```
ACCGT         --ACCGT--
CGTGC         ----CGTGC
TTAC          TTAC-----
TGCCGT        -TGCCGT--
              _________
              TTACCGTGC
```

The first `G` in the last fragment should have been an `A`. Note that majority voting still produces the right consensus.

Complications — Errors 2

Here we see an incorrect `A` present in the second fragment:

```
ACCGT        --ACC-GT--
CAGTGC       ----CAGTGC
TTAC         TTAC------
TACCGT       -TACC-GT--
             ----------
             TTACC-GTGC
```

Again, majority voting produces the right consensus, also using spaces (-'s) in the multiple alignment. The - in the consensus is discarded later on.

Complications — Errors 3

This time there was a deletion in the third (or fourth) base of the last fragment:

```
ACCGT          --ACCGT--
CGTGC          ----CGTGC
TTAC           TTAC-----
TACGT          -TAC-GT--
               _________
               TTACCGTGC
```

And again, majority voting produces the right consensus.

Other errors are chimeras (fragments sometimes stick together) and contamination. Unfortunately, there usually is not much you can do about these.

Some biology

DNA has two oriented strings (strands) consisting of bases `A/C/G/T`, paired through `A`–`T` and `C`–`G`.
So this looks like:

```
         -------->
5' ...AATACCCG... 3'
         ||||||||
3' ...TTATGGGC... 5'
         <--------
```

If we blow the thing to pieces, we (might) get `AATACCCG` and `CGGGTATT`. The second is called the reverse complement of the first.
We only need to reconstruct one of the two strands.

Fragment assembly **Compl's — Unknown orientation**

So fragments can also be used "backward". In that case we employ the reverse complement of the string, e.g., `CGTAGT` for `ACTACG` (use `A`–`T` and `C`–`G`). An example:

```
CACGT     --->    CACGT--------
ACGT      --->    -ACGT--------
ACTACG    <---    --CGTAGT-----
GTACT     <---    -----AGTAC---
ACTGA     --->    ---------ACTGA
CTGA      --->    ----------CTGA
                 ______________
                  CACGTAGTACTGA
```

Note that we suddenly have 2^n possible combinations for a set of n fragments — or a few less.

Compl's — Repeated regions

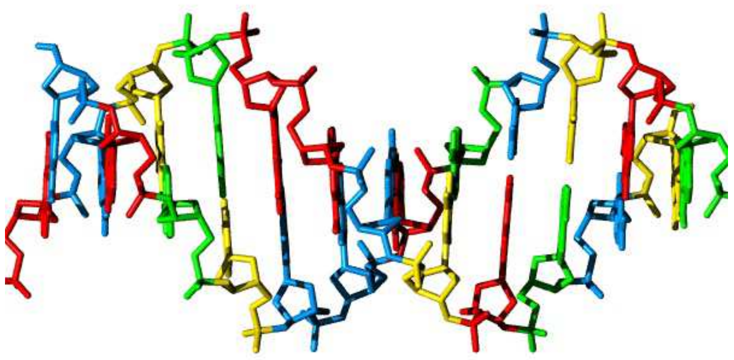
Repeated regions or repeats are sequences that appear two or more times in the target molecule. There are many complications, in particular if repeats are long (single bases are also repeats :-). We mention:

- where to put fragments totally contained in a repeat?
- the target $aXbXcXd$ can be assembled as $aXcXbXd$
- and $aXbYcXdYe$ can be assembled as $aXdYcXbYe$
- "inverted repeats", . . .

Compl's — Inverted repeats

An example of an inverted repeat is the following:

```
  TGCCTA-----              -----TAGGCA
  ----TAGCTCA              TGAGCTA----
 ___________              ___________
AACTGCCTAGCTCAGTT        AACTGAGCTAGGCAGTT
```



Compl's — Lack of coverage

Some parts of the target may be weakly covered, or even uncovered. In that case, the best you can hope for is a layout for every one of the contiguously covered regions, called contigs.

Perhaps the mean coverage gives an indication of the quality.

There are many formulas that connect the lengths of the actors, the number of contigs, . . . , using probability models, under (simplifying) assumptions.

Models — SCS 1

The simplest model for our problem is:

Shortest Common Superstring (SCS)
Input: A collection $\mathcal{F}$ of strings.
Output: A string S, as short as possible, such that for every $f \in \mathcal{F}$, S is a superstring of f.

Example: let $\mathcal{F} = \{\texttt{ACT}, \texttt{CTA}, \texttt{AGT}\}$. The sequence $S = \texttt{ACTAGT}$ is the (unique) shortest common superstring of $\mathcal{F}$.

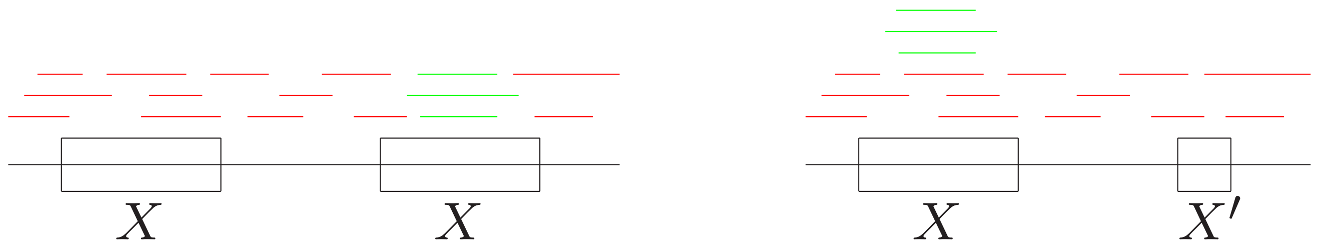
Shortest Common Superstring

The Shortest Common Superstring (SCS) problem is also useful for Pokemon.

According to Mark Stamp and Austin E Stamp, booster packs are created by selecting 5 common cards from a length 122 superstring.

Models — SCS 2

Practical shortcomings of this model are the following. Suppose the target looks like $aXbXc$ with some large repeat X. The SCS model could give a consensus $aXbX'c$ with X' shorter than X, where all fragments totally contained in the rightmost X are moved to the leftmost X, and left and right part of X' are unlinked:



The model allows no errors; orientation must be known. It is proven to be NP-hard, even in the binary case, when using a maximum on the length of the superstring.

Models — Reconstruction 1

For a refinement we define the substring edit distance of two strings a and b:

$$d_s(a,b) = \min_{s \in Sub(b)} d(a,s),$$

where $Sub(b)$ is the set of all substrings of b, and d is the "classical" edit distance: the minimum number of substitutions, insertions and deletions needed to change the first argument into the second. Note that $d_s(a,b) \neq d_s(b,a)$ in general.

If $a =$ `GCGATAG` and $b =$ `CAGTCGCTGATCGTACG`, the best alignment is

```
-----GC-GATAG----
CAGTCGCTGATCGTACG
```

with distance $d_s(a,b) = 2$.

Models — Reconstruction 2

The second model for our problem is:

Reconstruction
Input: A collection $\mathcal{F}$ of strings and an error tolerance ϵ with $0 \leq \epsilon \leq 1$.
Output: A string S, as short as possible, such that for every $f \in \mathcal{F}$ we have $\min(d_s(f, S), d_s(\bar{f}, S)) \leq \epsilon |f|$, where $\bar{f}$ is the reverse complement of f and $|f|$ is the length of f.

This means that we allow $100 \times \epsilon$ errors per 100 bases.
The problem is also NP-hard. No repeats, . . .

Models — Multicontig 1

We now also reward good linkage. Let us start with some examples:

Let $\mathcal{F} = \{\texttt{GTAC}, \texttt{TAATG}, \texttt{TGTAA}\}$.
We can produce two solutions with 2 contigs:

```
--TAATG   GTAC                 TAATG---   GTAC
TGTAA--                        ---TGTAA
```

The left one has an overlap of width 3, the right one of 2.
There is also a 1-contig solution, with smallest overlap 1:

```
TGTAA-----
--TAATG---
------GTAC
```

Models — Multicontig 2

The third model for our problem is:

Multicontig
Input: A collection $\mathcal{F}$ of strings, an integer $t \geq 0$ and an error tolerance ϵ with $0 \leq \epsilon \leq 1$.
Output: A partition of $\mathcal{F}$ in the minimum number of subcollections $\mathcal{C}_i$ $(1 \leq i \leq k)$, such that every $\mathcal{C}_i$ admits a t-contig with an ϵ-consensus.
(See Setubal-Meidanis for precise definitions.)

A t-contig is a (“connected”) layout where the smallest “linking” overlap has width at least t. Again: NP-hard.

Algorithms — Greedy 1

A greedy algorithm for the Shortest Common Superstring problem is easily conceived:

> Repeatedly find the two strings with the largest overlap, and replace them with their shortest superstring.

This is an approximating algorithm. Its solution can be proved to be of length at most 4/. . . /2.75/. . . times the optimal length. (Technical proofs. Several variations of the algorithm. Conjectured: 2.)

Algorithms — Greedy 2

An example of the greedy algorithm: we start with the four strings `TCAGT`, `CATCAG`, `GTG` and `GCA`.

The two most overlapping ones are `CATCAG` and `TCAGT`; they are replaced with `CATCAGT`, leaving us with `CATCAGT`, `GTG` and `GCA`.

Both `GTG` and `GCA` have a 2 base overlap with `CATCAGT`. Choose `GTG` (say), giving `CATCAGTG` and `GCA`.

The final solution is `GCATCAGTG`, which happens to be optimal.

Algorithms — Greedy 3

Another example of the greedy algorithm: we start with the three strings `GCC`, `ATGC` and `TGCAT`.

The two most overlapping ones are `ATGC` and `TGCAT`; they are replaced with `ATGCAT`, leaving us with `ATGCAT` and `GCC`.

These two strings have no overlap, so the final solution is their concatenation: either `ATGCATGCC` or `GCCATGCAT`, both of length 9. The optimal solution, `TGCATGCC`, has length 8 however!

More general: starting from

$$\{\mathtt{C(AT)}^k, \mathtt{(TA)}^k, \mathtt{(AT)}^k\mathtt{G}\}$$

for fixed $k \geq 1$, the algorithm outputs $\mathtt{C(AT)}^k\mathtt{G(TA)}^k$ of length $4k+2$, whereas the optimal string $\mathtt{C(AT)}^{k+1}\mathtt{G}$ has length $2k+4$.

This shows that the output string of the greedy algorithm can be twice as long as the correct one — and that the algorithm can be easily improved in a heuristic way.

Algorithms — Others

There are also quite different approaches:

- evolutionary algorithms
  individuals, i.e., candidates, have fitness given by the maximal number of overlapping characters

- DNA-computing
  encode all strings as suitable DNA(!) strands, generate all strands of fixed length, and step by step discard all wrong ones

Algorithms — Overlap graph 1

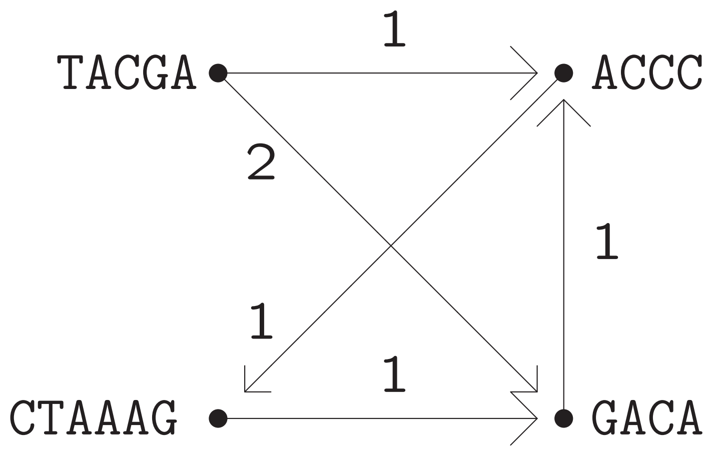
Finding common superstrings of the strings in a set $\mathcal{F}$ is the same as finding certain paths in a corresponding graph: the overlap (multi)graph.

The set of nodes V is just $\mathcal{F}$. A directed edge from $a \in \mathcal{F}$ to a different $b \in \mathcal{F}$ with weight $t \geq 0$ exists if the last t characters of a (its length t suffix) coincide with the length t prefix from b. Usually you only consider the largest t per pair (a, b).

We normally assume that the set $\mathcal{F}$ is substring-free: we do not allow different elements of $\mathcal{F}$ to be a substring of one another. (We just remove such substrings.)

Algorithms — Overlap graph 2

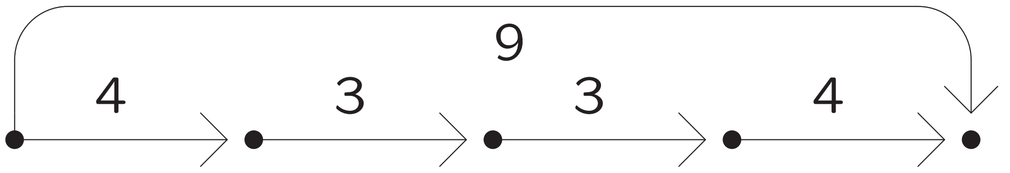
Let $\mathcal{F} = \{\texttt{TACGA}, \texttt{ACCC}, \texttt{CTAAAG}, \texttt{GACA}\}$. The overlap multigraph is (without edges with weight 0):


$$\begin{tikzcd}
\texttt{TACGA} \arrow[r, "1"] \arrow[dr, "2"] & \texttt{ACCC} \arrow[dl, "1"] \\
\texttt{CTAAAG} \arrow[r, "1"] & \texttt{GACA} \arrow[u, "1"]
\end{tikzcd}$$

Any path in the graph gives an alignment. A solution to the Shortest Common Superstring problem translates into a Hamiltonian path of maximum weight (visit all nodes). Or a Traveling Salesman Problem — if you want.

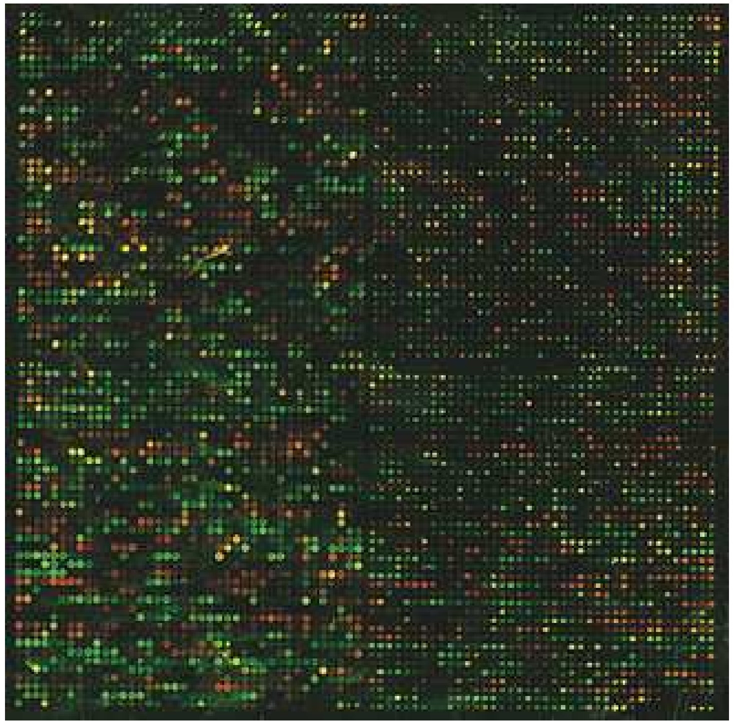
```
AGTATTGGCAATC---AATCGATG------------   solution of
---------------------ATGCAAACCT-----   length 36,
----TTGGCAATCACT------------CCTTTTGG   "weakest link" 0
------------------------------------
AGTATTGGCAATCACTAATCGATGCAAACCTTTTGG   greedy algorithm

AGTATTGGCAATC--------CCTTTTGG--------   solution of
---------AATCGATG--------TTGGCAATCACT   length 37,
--------------ATGCAAACCT-------------   "weakest link" 3
-------------------------------------
AGTATTGGCAATCGATGCAAACCTTTTGGCAATCACT   topological sorting
```



Sequencing by hybridization

Yet another technique, as mentioned before, is sequencing by hybridization (SBH). This technique uses a DNA-array, consisting of many small pieces of DNA (say, all $4^6 = 2^{12} = 4096$ possible strings of length 6, or some subtle selection), and determines whether or not each string occurs as a substring. This information is used for reconstruction.



SBH and Eulerian paths

Instead of looking for Hamiltonian paths in the overlap graph, we now focus on Eulerian paths (that traverse all edges) in another graph — which is a simple task.

Suppose we start with strings of length ℓ. Vertices correspond to $(\ell-1)$-tuples, edges (!) to fragments. For example, with fragments `ATG`, `TGG`, `TGC`, `GTG`, `GGC`, `GCA`, `GCG` and `CGT` ($\ell = 3$), we find two solutions:

`ATGGCGTGCA`

$$\begin{tikzcd}
 & GT \arrow[d, "6"'] & & CG \arrow[ll, "5"'] & \\
AT \arrow[r, "1"] & TG \arrow[rr, "7"] \arrow[dr, "2"'] & & GC \arrow[u, "4"'] \arrow[r, "8"] & CA \\
 & & GG \arrow[ur, "3"'] & &
\end{tikzcd}$$

`ATGCGTGGCA`

$$\begin{tikzcd}
 & GT \arrow[d, "5"'] & & CG \arrow[ll, "4"'] & \\
AT \arrow[r, "1"] & TG \arrow[rr, "2"] \arrow[dr, "6"'] & & GC \arrow[u, "3"'] \arrow[r, "8"] & CA \\
 & & GG \arrow[ur, "7"'] & &
\end{tikzcd}$$

Exercises—1

Try the following exercises from pages 139–140 of the Setubal-Meidanis book (see handouts):

3, 6, 7, 12, 13, 18, 1

Answers: `www.liacs.nl/home/kosters/bio/`

Suffix Arrays

And now for some suffix arrays.

The suffix array of a string is the *lexicographically sorted array of all its suffixes*. Usually we give the indexes where the suffixes begin.

Example: the string `example` has 7 (non-empty) suffixes:

`ample`, `e`, `example`, `le`, `mple`, `ple`, `xample`.

So the array is $[2, 6, 0, 5, 3, 4, 1]$.

Some history

The story begins in the 1990s, when finally Ukkonen came up with a linear time construction for suffix trees (see previous lecture). Full details: Dan Gusfield's book, or Pekka Kilpeläinen's lecture notes:

`www.cs.uku.fi/~kilpelai/BSA07/index.shtml`

A depth first "lexical" suffix tree traversal easily gives the suffix array.

In 2003 three independent algorithms to directly construct suffix arrays (introduced by Myers and Manber) in linear time (sometimes together with the so-called lcp-array = lenghts of the longest common prefixes; together they are equivalent with suffix trees) were found: Kärkkäinen-Sanders, Ko-Aluru and Kim-Sim-Park-Park.

Why suffix arrays?

Suffix trees and suffix arrays are great when one wants to find, e.g., all overlaps in a large set of strings.

Often a special final character `$` is attached to the string at hand, to avoid a suffix that matches a prefix of another suffix: `xabxa`.

How to find an occurrence of a substring P of a string T? Perform a binary search on the suffix array SA: compare P to the middle element of SA, and so on. With help of the lcp-array, this can be done in $O(n + \log(m))$ time, where $n = |P|$ and $m = |T|$. (Don't forget the "preprocessing"; it works if you have many Ps.)

Kärkkäinen-Sanders

The Kärkkäinen-Sanders algorithm is the easiest (but perhaps not the best). It goes like this:

- recursively construct the suffix array of the suffixes starting at positions i that are not a multiple of 3: $1, 2, 4, 5, 7, 8, 10, 11, \ldots$
- construct the suffix array of the others using the result of the first step
- merge the two suffix arrays into one

Fragment assembly **Example**

```
          1
01234567890
mississippi
```

- start with `ississippi` ($i = 1$), `issippi` ($i = 4$), `ippi` ($i = 7$), `i00` ($i = 10$, with extra 00), `ssissippi` ($i = 2$), `ssippi` ($i = 5$), `ppi` ($i = 8$)
  we find $[3, 2, 1, 0, 6, 5, 4] \Rightarrow [10, 7, 4, 1, 8, 5, 2]$

- do `mississippi`, `pi0`, `sippi`, `sissippi`: $[0, 9, 6, 3]$

- merge the two suffix arrays: $[10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]$

The lcp value for `issippi` and `ississippi` is $4 = lcp(2, 3)$.

The lcp-array

How can the lcp-array help when searching for a substring?

Suppose we are looking for $P = \texttt{abcdemn}$. Suppose that we do a binary search in $L = \texttt{abcdefg...}$, . . . , $M = \texttt{abcdefg...}$, . . . ,$R = \texttt{abcdxyz...}$ (within the suffix array). P matches the first $\ell = 5$ characters of L, and the first $r = 4$ of R. Here $lcp(L, M) > \ell$.

What can we conclude now? And how does this work in general?

Exercises—2

- Read the 2 page copy of part of the Kärkkäinen-Sanders paper (see handout).
- Try to understand the algorithm.
- Explain why it is linear.