Physical Mapping

A physical map of a DNA tells the location of precisely defined sequences along the molecule.

- Restriction mapping: mapping of restriction sites of a cutting enzyme based on lengths of fragments
  - Double Digest Problem DDP
  - Partial Digest Problem PDP

- Hybridization mapping: mapping clones based on hybridization data with probes
  - Non-unique probes
  - Unique probes

Unique probes

- Unique probes \( \rightarrow \) each probe occurs only once along the DNA.

- Unique probes are not easy to generate, they are usually long probes.
  - example: STS (Sequence Tag Site) probe is extracted from the DNA itself, often from endpoint of clone, and is sufficiently long that is unlikely to occur a second time on the DNA

- Finding the shortest covering string is this case can be done in polynomial time.
Shortest covering string

Assuming no hybridization errors

– (1) The length of the shortest covering string is equal to the number of probes.

– (2) The shortest covering string is now given by a special permutation of the probes.

– (1) + (2) => consecutive ones property $C1P$:

there exists a permutation of the columns of $D$, such that 1’s in each row occur in consecutive positions.

Example of error

• Assume
  – unique probes
  – Obtained the following $D$

• We cannot permute the columns to make $D$ satisfy $C1P$.

• We must have a hybridization error!
  – possible shortest covering string in this case: $ijk$

(3 possible permutations up to reversal $3!/2 = 3$)

Finding $C1P$ permutation

• We will assume no hybridization errors

• The shortest covering string problem reduces to finding a permutation of the columns of $D$ to put $D$ in $C1P$ form.

• We will not try to explicitly construct the $C1P$ permutation, but we will find it by repeatedly identifying neighboring probes.
Assumption 1

No hybridization errors
(i.e. $C_1P$ permutation exists)

Assumption 2

Non-inclusion: No clone $X$ contains another clone $Y$.

Assumption 3

Connectedness: for every partition of the set of probes into two non-empty sets $A$ and $B$, there exist probes $i \in A$ and $j \in B$ such that $C_i \cap C_j \neq \emptyset$
Assumption 4

Distinguishability: \( C_i \neq C_j \) for \( i \neq j \)

C1P reformulation (Lemma)

Let 1\( \ldots \)m be the correct ordering of probes and 1 \( \leq i < j < k \leq m \). Then

\[ C_i \cap C_j \subseteq C_i \cap C_k \]

Assumption 1: if \( c \in C_i \cap C_j \), then \( c \in C_i \cap C_k \)

Proof:

\[ \square \]

Closest probe

- Given a probe \( i \) and a set of probes \( P \) such that \( C_i \cap C_j \neq \emptyset \) for some \( j \in P \), a closest probe \( k \) in \( P \) to \( i \) is such that there is no probe in \( P \) between \( i \) and \( k \).

- If there is only one probe \( k \in P \) such that \( |C_i \cap C_j| \) is maximized, then there can be no probe between \( i \) and \( k \), and \( k \) must closest (either from left or from right).

- What if we have a number of probes in \( P \) such that \( |C_i \cap C_j| \) is maximized? In this case, the probe \( k \) with a minimum \( |C_j| \) is closest (either from left or from right).
Closest probe (cont.)

Consider one side of \( k \) (say right) and assume \( |C_i \cap C_j| = |C_i \cap C_k| = 0 \)
(this also means that \( C_i \cap C_j = C_i \cap C_k = \text{no errors} \))

\[
\begin{array}{ccc}
& i & j & k \\
\end{array}
\]

\( C_i \) and \( C_j \) must be different (distinctability)

There must be a clone that hybridizes with \( k \) but not with \( j \), because we cannot
have a clone that hybridizes with \( j \) but not with \( k \), otherwise it will be
contained in another clone, a contradiction (non-inclusion)

Therefore, \( |C_j| < |C_i| \)

---

Strict partial order relation
(closer)

\( j \) closer than \( k \) to \( i \) iff

\[ |C_i \cap C_j| > |C_i \cap C_k| \]

or

\[ |C_i \cap C_j| = |C_i \cap C_k| \neq 0 \text{ and } |C_j| < |C_k| \]

---

Algorithm

The algorithm maintains an ordered set \( s = p_{m-1} \ldots p_0 \) denoting a correct
sub-sequence of consecutive probes.

\[
\begin{align*}
\text{compute } & |C_i| \text{ and } |C_j| \text{ for all probes } i \text{ and } j \\
& x = p_{m-1} = p_m = i \text{ for any } j \\
& P = \{ \text{all probes except } j \} \\
\text{repeat} \\
& \text{if } O_{m-1} \cap C_j = \emptyset \text{ for all } j \in P \\
& \quad \text{then reverse } x \\
& \quad \text{choose } j = P \text{ with } \min |C_j| \text{ that maximizes } |O_{m-1} \cap C_j| \\
& \quad \text{if } x = x_m \\
& \quad \quad \text{then } x = x_j \\
& \quad \quad \text{else } x = x_j \\
& \quad \quad P = P \setminus \{ j \} \\
& \text{until } P = \emptyset
\end{align*}
\]
DNA Sequencing

DNA sequencing

- To sequence a DNA is to obtain the string of bases that it contains.
- It is impossible to sequence the whole DNA molecule directly.
- We may however obtain a piece of a certain length cut at random and sequence it. This is called a fragment.
- By using cloning and cutting techniques we can obtain a large number of sequenced fragments.
- The goal is to reconstruct the DNA molecule based on the fragments overlap (now the overlap is determined by the explicit sequences).

Ideal case

- We know the length of the DNA (e.g. = 10 bases)
- There are no errors in sequencing the fragments

ACCGT
CGTGC
TTAC
TACCGT

- Align sequences ignoring and gaps
- Find consensus by majority voting

ACCGT
CGTGC
TTAC
TACCGT

TTCCGTGC
Insertion errors

ACCGT  --ACCGT--
CAGTGC  -----CAGTGC
TTAC    TTAC------
TACCGT  --TACCGT--
          TTACCGTGC

Insertion of A in the second fragment
Gap in consensus will be discarded
In this example, it still works because of majority voting

Deletion error

ACCGT  --ACCGT--
CAGTGC  -----CAGTGC
TTAC    TTAC------
TACCGT  --TACCGT--
          TTACCGTGC

The first C was deleted from 4th fragment
Consensus still works

Chimeric fragment

Two disjoint fragments join to form one fragment
that is not originally part of the DNA

ACCGT  --ACCGT--
CAGTGC  -----CAGTGC
TTAC    TTAC------
TACCGT  --TACCGT--
          TTACCGTGC
          TTA---TGC
Unknown orientation

which strand a particular fragment belongs to?

CACGT → CACGT
ACGT → -ACGT
ACTACG → --CGTAGT
GTACT → -----AGTAC
ACTGA → --------ACTGA
CTGA → --------CTGA

We have 2^n possibilities

Repeats

Repeats of the form X X X

Repeats of the form X Y X Y
Inverted repeats

Inverted repeat

Lack of coverage

We have more than one contig

DNA sequencing

- Shortest common superstring SCS
  - “An elegant theoretical abstraction, but fundamentally flawed” – R. Karp

Given a set of fragments \( F \),

Find the shortest string \( s \) that contains every \( f \in F \) as a substring

- This is NP-hard
- The SCS might not be what we really want
Bad example (repeats)

Shortest common superstring will give:

Solving SCS

We are going to consider a Hamiltonian path approach to solving the SCS problem

Overlap graph

- Consider the complete directed weighted graph $G = (V,E)$, called the overlap graph
  - $V = F$ (each fragment is a vertex)
  - $(u,v) \in E$ with weight $t$ iff $t$ is the length of the maximal suffix of $u$ that is a prefix of $v$
- We allow self loops and zero weight edges
A path defines a superstring

- Every simple path $P$ in the overlap graph involving a set of vertices (fragments) $A$ defines a superstring $s(P)$ for the set $A$.

- Therefore, a Hamiltonian path in the overlap graph defines a superstring for the set of fragments $F$.

- A Hamiltonian path must exist because the graph is complete (how many do we have?).

Example

\[
P = a b c\]

\[
\begin{align*}
a &= \text{TACGA} \\
b &= \text{ACCG} \\
c &= \text{CTAAAG} \\
d &= \text{GAGC}
\end{align*}
\]

\[
s(P) = \text{TACGACCGCTAAAG}
\]
Does a superstring define a path?

- We have seen that every Hamiltonian path corresponds to a superstring.
- Is the converse true?
  - No: A superstring can contain arbitrary characters that are not present in any fragments
- Does a shortest superstring correspond to a Hamiltonian path?
  - Yes: if $F$ is substring-free, i.e. no fragment in $F$ is contained in another

Example

The shortest superstring is

\[ \text{AGCT} \]

There is no Hamiltonian path $P$, such that $s(P) = \text{AGCT}$

Substring-free collection $F$

Let $F$ be a substring free set, then for every shortest superstring $s$, there is a Hamiltonian path $P$, such that $s(P) = s$.

Proof: assume the fragments appear in $s$ as follows (no gaps and no one can be contained in another)
Non substring-free $F$

- If $F$ is not substring-free, then we can remove all fragments from $F$ that are substrings of other fragments
- We end up with a set $F$
- But any superstring of $F$ is a superstring of $F$
- Therefore, we can use $F$

Length of string v.s. weight of path

- Let $P$ be a Hamiltonian path.
- Let $w(P)$ be the weight of $P$.
- Let $|F| = \sum_{a \in F} |a|$
- Then $|s(P)| = |F| + w(P)$ [proof is simple]
- Therefore, the shortest common superstring corresponds to the Hamiltonian path with minimum weight

Proof

Let $P$ be a Hamiltonian path with minimum weight

we need to show that $s(P)$ is a shortest superstring

- Let $s$ be a shortest superstring with $|s| < |s(P)|$
- Then there is a Hamiltonian path $P$ such that $s = s(P)$
- $|s(P)| = |F| + w(P) < |s(P)| = |F| + w(P)$
- Therefore, $w(P) < w(P)$, contradiction
Hamiltonian path approach

- Finding a minimum weight Hamiltonian path is NP-hard (you can reduce HAMPATH to it)
- Unfortunately, there is no “better” approach to solve SCS, because SCS itself is NP-hard
- Let’s consider a greedy algorithm for finding a Hamiltonian path

Greedy algorithm

- Greedy:
  - start with an empty path
  - repeatedly add the least weighted available edge until you get a Hamiltonian path
- Every time we add an edge \((u,v)\), we need to check:
  - \((u,v)\) does not create a cycle with the previously added edges
  - \(u\) has no previously added outgoing edge
  - \(v\) has no previously added incoming edge

Greedy algorithm

```plaintext
sort edges by their weight: e_1, e_2, ..., e_n
for all v \in V
  m(v) \leftarrow 0
  out(v) \leftarrow 0
H \leftarrow \emptyset
i \leftarrow 1
while \(|H| < |V| - 1
  (u,v) \leftarrow \text{next edge}
  if out(u) = 0 and m(v) = 0
    H \leftarrow H \cup e
    out(u) \leftarrow 1
    m(v) \leftarrow 1
    i \leftarrow i + 1

To build the graph: trivially \(O(n^2)\)
(could be done optimally in \(O(n + |R|\)) using suffix trees)
To run the algorithm: \(O(|R| \log |R|)\)
```
Example

Greedy algorithm will choose:
- ATGC
- TGCAT
- GCC
- ATGCATGCC

Optimal is:
- TGCAT
- ATGC
- GCC
- TGCATGCC