Computational Biology
Lecture 12: Physical mapping by restriction mapping
Saad Mneimneh

In the beginning of the course, we looked at genetic mapping, which is the problem of identify the *relative* order of genes on a chromosome. Now we will look at physical mapping. A physical map of the DNA tells the location of precisely defined sequences along the molecule. Physical mapping often deals with real distances and not just a relative ordering. We have two approaches for physical mapping: Restriction mapping and Hybridization mapping. Restriction mapping involves the mapping of restriction sites (precise short sequences) of a cutting enzyme based on the lengths of the restriction fragments. Hybridization mapping involves the mapping of clones (fragments) based on hybridization data with probes. The mapping of clones in this case is not exact; we try to find a likely overlap of the clones.

We will start with restriction mapping. We have two varations on the restriction mapping problem:

- Double Digest Problem
- Partial Digest Problem

The Double Digest Problem DDP

The basic problem of restriction mapping is the following. We cut the DNA with a restriction enzyme and we obtain a multiset of fragment lengths (digestion). The goal is to map (order) the fragment legnths on the DNA. By doing this, we figure out the exact location of precisely defined short sequences: the restriction sites.

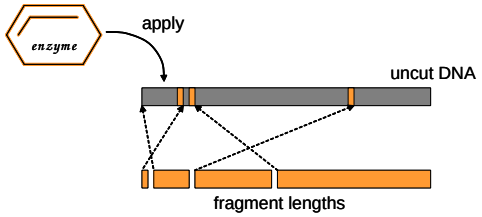


Figure 1: Digestion

But the lengths of fragments obtained from digestion as depicted in the figure above do not constitute enough information: any order is consistent with the experiment. For this reason, we use more than one enzyme. In double digestion we use two enzymes A and B. We bring three copies of the DNA and conduct three separate experiments. In the first experiment, we cut the first copy of the DNA with enzyme A to obtain a multiset of lengths A. In the second experiment, we cut the second copy of the DNA with enzyme B to obtain a multiset of lengths B. Finally, in the third experiment, we cut the third copy of the DNA with both enzymes A and B to obtain a multset of lengths C.

Now ordering the lengths of A and B cannot be done arbitrarily since every cut (restriction site) in the first two experiments occurs in the third one. The problem then is to determine the ordering of the lengths in A, B and C that is consistent with all three experiments.

This is called the Double Digest Problem depicted in Figure 2.

Formally, the Double Digest Problem is the following:

Double Digest Problem DDP

Given:

- multiset A of fragment lengths from enzyme A
- multiset B of fragment lengths from enzyme B
- multiset C of fragment length from enzymes A$\wedge$B

Determine an ordering of the fragments that is consistent with the three experiments.

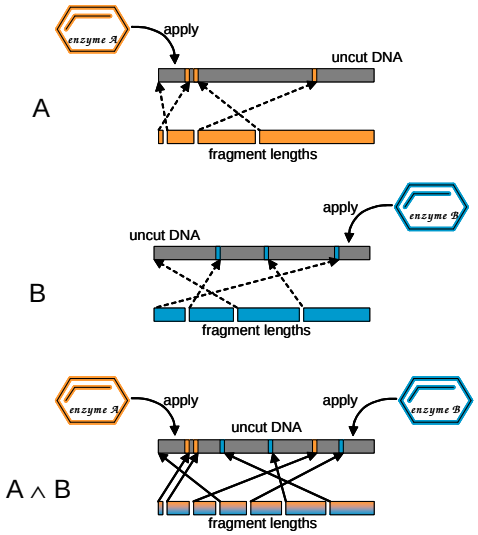
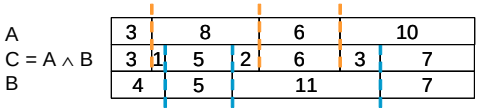


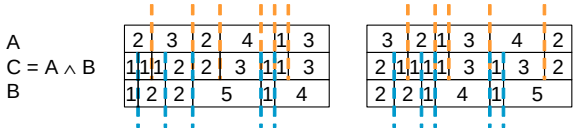
Figure 2: Double Digestion

Example:
$A = \{3, 6, 8, 9\}$
$B = \{4, 5, 7, 11\}$
$C = \{1, 2, 3, 3, 5, 6, 7\}$
Then a possible solution would be:



The solution of a DDP might not be unique. In fact the number of solutions grows exponentially in number of fragments. Here's an example illustrating two possible solutions for the same instance.

Example:
$A = \{1, 2, 2, 3, 3, 4\}$
$B = \{1, 1, 2, 2, 4, 5\}$
$C = \{1, 1, 1, 1, 1, 2, 2, 3, 3\}$
Then here are two solutions:



Equivalence of Solutions

Finding all the solutions to the Double Digest problem is a difficult task. Moreover, some solutions might be equivalent. For instance, given a solution, it is possible to obtain another one by just reflection it (see Figure 3).

Although the two solutions of Figure 3 are different, no fragment length data A, B, and C could possibly distinguish between the two.

This motivates us to study equivalence between solutions. This is particularly useful because, on one hand, when we find a solution, we need to be confident that it is really distinct from the solutions we have already seen. On the other hand, when we find a solution, it would be useful to count and enumerate the equivalent solutions since any of them could be the real answer.

First, we need to carefully define what "equivalent" means since we can have many notions of equivalence. We will discuss in detail two different notions of equivalence: *Overlap Equivalence* and *Cassette Transformation Equivalence*.

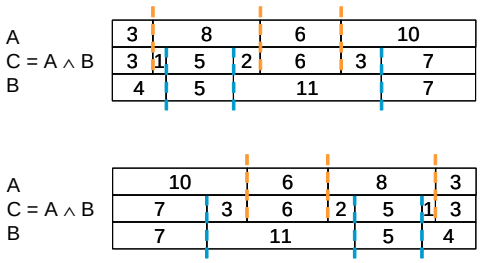


Figure 3: Reflection

Overlap Equivalence

Overlap equivalence is more general then reflections, i.e. a reflection is a special case of an overlap equivalence. We first start with a definition:

Definition [overlap matrix] Let $\{A_i\}$ be the set of fragments of A. Let $\{B_j\}$ be the set of fragments of B. A solution to the DDP defines an overlap matrix O where $O_{ij} = 1$ iff A_i overlaps with B_j.

Now two solutions are overlap equivalent if they define the same overlap matrix. It is obvious that two solutions that are the reflections of each other are overlap equivalent since reflection only changes the orientation but not the overlap of the fragments.

Given a solution, we would like to generate all its overlap equivalent solutions. What transformations can we make on a solution that lead to another equivalent solution? And how many equivalent solutions do we have?

There are two operations that transform one solution to another overlap equivalent one: *reflection* and *permutation*. Before describing these, we need to define the notion of a *component*. A component is defined as follows: if a solution has $t-1$ coincident cuts in A and B (and hence C), it will have t components which can be permutated in $t!$ ways without changing the overlap matrix, since none of the overlaps cross over a coincident cut. Each component can also be reflected without changing the overlap matrix, leading to 2^t equivalent solutions for a given permutation of the components.

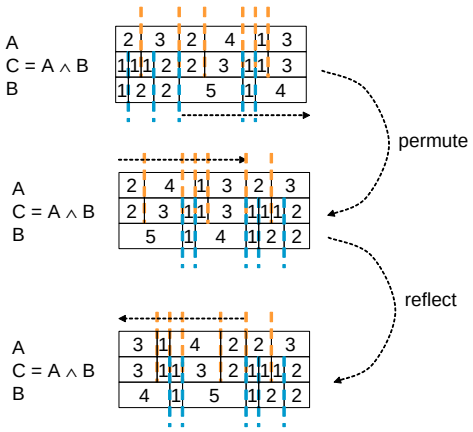


Figure 4: Permuting and reflecting components

So far we can have $2^t t!$ overlap equivalent solutions. Do we have more? Yes. If we have a subset of $\{A_i\}$ that are completely contained in some fragment B_j, these fragments can be permuted without chaging the overlap matrix since they all overlap with B_j only. Formally, we let:

$$\alpha_j = \{A_k : A_k \subset B_j\}$$
$$\beta_i = \{B_k : B_k \subset A_i\}$$

We can permute the fragments of each α_j in $|\alpha_j|!$ ways and the fragments of each β_i in $|\beta_i|!$ ways without changing the overlap matrix. So now the total number of overlap equivalent solutions is:

$$2^t t! \prod_j |\alpha_j|! \prod_i |\beta_i|!$$

Actually the number of equivalent solutions derived above is not correct but it is almost correct. One more consideration must be made. If a component has only one fragment in either A or B, that fragment completely contains the fragments in the opposite set. This corresponds to a α_j or β_i. Hence a reflection of one of these components is already counted in our permutation of α_j or β_i. We ignore reflections for such components. If there are s such components, the correct number of equivalent solutions is:

$$2^{(t-s)}t! \prod_j |\alpha_j|! \prod_i |\beta_i|!$$

A similar but more general notion of equivalence is *Overlap Size Equivalence*. Two solutions are overlap size equivalent if they define the same overlap sizes. Overlap equivalence is a special case of overlap size equivalence. In other words, if two solutions are overlap equivalent they are are also overlap size equivalent. The converse is not true. Consider for example an instance where a length is repeated (either in A or in B), the corresponding fragments can be swapped to create an overlap size equivalent solutions that have a different overlap matrix.

Cassette Transformation Equivalence

This is a more sophisticated notion of equivalence. For a given solution, we define a set of consecutive blocks in C, $I_C = \{C_k : i \leq k \leq j\}$ where $1 \leq i \leq j \leq |C|$ and C_k is the k^{th} fragment of C in the solution. The cassette defined by I_C is (I_A, I_B), where I_A is the set of fragments in A that contain some fragment in I_C and I_B is defined similarly. An illustration of a cassette is given below.

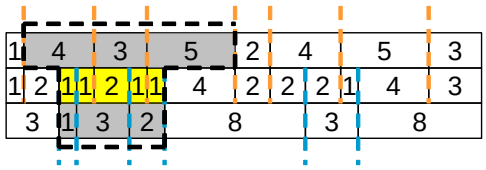


Figure 5: Cassette for $I_C = \{C_3, C_4, C_5, C_6, C_7\}$

One observation of cassettes is that on each side, the fragment from either A or B (but not both) may extend beyond the border of I_C. The lengths of these extensions on both sides are called the *Left and Right Overlap* respectively. If the fragment from A forms the left overlap, this length is given a negative sign, while for a left overlap in B, it is positive. The opposite is the case for the right overlaps of A and B. These signs are important when determining cassette equivalence.

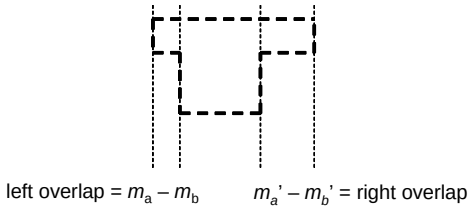


Figure 6: Left and right overlaps of a cassette

There are two transformations on cassettes that can lead to equivalent solutions: *exchange and reflection*. Two disjoint cassettes can be exchanged if they have the same left and right overlap. A cassette can be reflected if the magnitude of the left and right overlaps for a cassette are the same but the signs are different. Figure 7 and Figure 8 illustrate these two transformations.

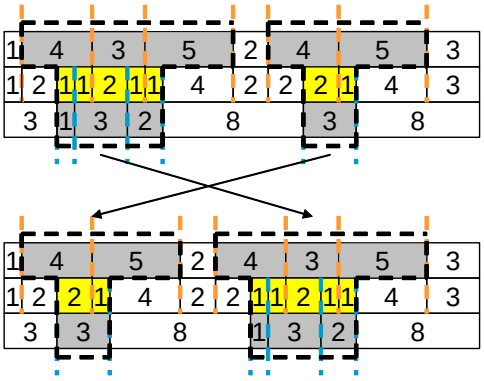


Figure 7: Cassette exchange, left overlap = -2, right overlap = 4

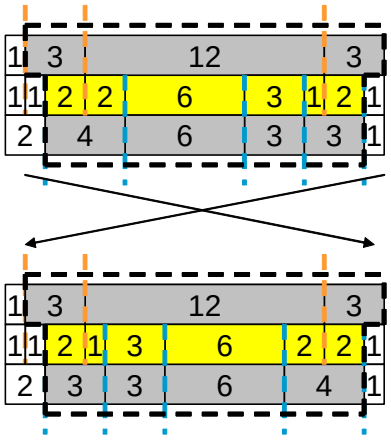


Figure 8: Cassette reflection, left overlap = -1, right overlap = 1

As with overlap equivalence we want to know if we can generate all cassette equivalent solutions. To do this we need to draw a relationship between the cassettes in a solution and an Euler Cycle. A *Euler Cycle* is a cycle that goes through every edge in a graph once (it may visit a vertex many times). In an edge colored graph, an alternating Euler cycle is a cycle in which consecutive edges have different colors. For every solution, we can construct a special edge bi-colored graph called the *Border Block Graph* that admits an alternating Euler cycle (Pevzner, 1995). Every cassette equivalent solution corresponds to an alternating Euler cycle in that graph and vice-versa. Therefore, by enumerating all Euler cycles in the border block graph of a given solution, we find all cassette equivalent solutions.

Pevzner proved the following:

In an edge colored graph, two alternating Euler cycles can be transformed into one another by a series of path excahnges and path reflections, where a path exchange and a path reflection are as shown in the following two figures:

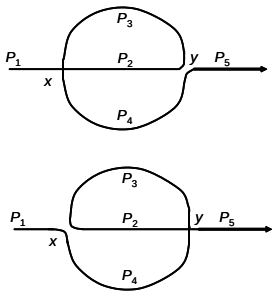


Figure 9: Path exchange: $P = P_1P_2P_3P_4P_5 \rightarrow P' = P_1P_4P_3P_2P_5$

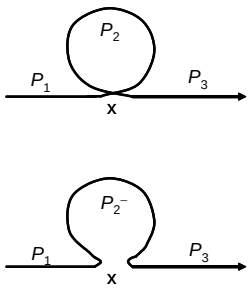


Figure 10: Path reflection: $P = P_1P_2P_3 \rightarrow P' = P_1P_2^-P_3$

To construct a border block graph we need to define the notion of a border block. The border blocks of a given fragment in A or B are the leftmost and rightmost fragments in C that are completely contained in the given fragment. Formally, we define:

$$I(A_i) = \{C_k : C_k \subset A_i\}$$
$$I(B_j) = \{C_k : C_k \subset B_j\}$$

The border blocks of a given fragment X are the leftmost and rightmost members of $I(X)$ when $|I(X)| > 1$. If $|I(X)| = 1$ then the border blocks of X are not defined. A block is a border block if it is the left or right border block of some fragment.

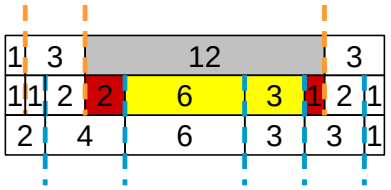


Figure 11: $I(A_3) = \{C_4, C_5, C_6, C_7\}$, border blocks of A_3 are C_4 and C_7. C_1, C_2, C_3, C_4, C_7, C_8, and C_9 are all border blocks

We have the following simple Lemma concerning border blocks:

Lemma:

- Each fragment X with $|I(X)| > 1$ contains exactly two border blocks.
- $I(A_i) \cap I(B_j) \leq 1$.
- Assume no cuts in A and B coincide, then each border block is a border block for some A_i and some B_j, except C_1 and $C_{|C|}$.

The border block graph is constructed as follows:

- We let $\beta = \{C_k : C_k \; is \; a \; border \; block\}$.
- The set of vertices is the set of lengths of border blocks in our solution. i.e. $V = \{|C_k| : C_k \in \beta\}$.
- For each fragment $X \in A \cup B$ with $I(X) > 1$, an edge connects the corresponding vertices (lengths of its two border blocls), i.e. $E = \{(|C_i|, |C_j|) \mid C_i \; and \; C_j \in \beta \cap I(X) \; for \; some \; X \in A \cup B\}$.
- Each edge is labelled by its fragment X and is colored A or B depending on which set X is in.

The following figure illustrates this construction.

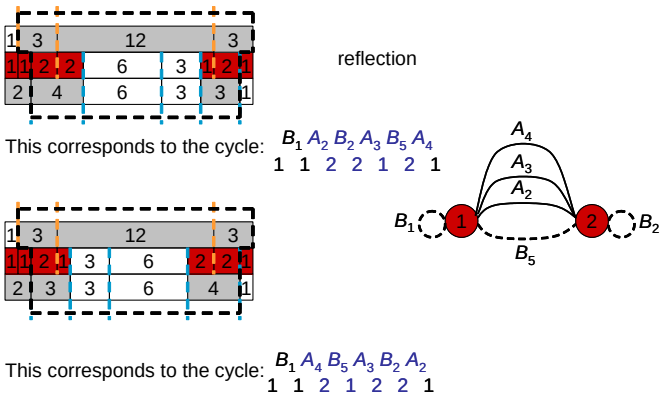


Figure 12: Border Block Graph

Note that according to the Lemma above, the border block graph is balanced for every vertex (i.e. the vertex has equal number of A color edges and B color edges), except possibly for $|C_1|$ and $|C_{|C|}|$. This means by adding one or two additional edges (depending on the edge colors for $|C_1|$ and $|C_{|C|}|$) we can make the block border graph balanced and hence it will have an alternating Euler cycle (Pevnzer 1995).

Note that cassette transformations (exchanges and reflections) do not change the border block graph. Pevnzer result is the following (see Figure 12):

- A solution $[a, b]$ corresponds to an alternating Euler cycle in its border block graph
- Let P be the alternating Euler cycle corresponding a solution $[a, b]$. If a solution $[a', b']$ is obtained from $[a, b]$ by cassette exchange (reflection), it will have an alternating Euler cycle P that can be obtained from P by a path exchange (reflection).
- Let P be an alternating Euler cycle obtained from P by path exchange (reflection). Then there is a solution $[a', b']$ that can be obtained from $[a, b]$ by cassette exchange (reflection), where P corresponds to $[a', b']$.

DDP is NP-complete

The DDP problem is NP-complete. To show that a problem is NP-complete we need to show two things:

- it is in NP, i.e. a solution can be verified in polynomial time
- any problem in NP can be reduced to it in polynomial time. It is sufficient therefore to show that an NP-complete problem can be reduced to it in polynomial time

First let us prove that a solution to the DDP problem can be verified in polynomial time. Let $[a, b]$ specify the locations of all restriction sites be a solution that we need to verify. Construct the multiset $g = a \cup b \cup \{0, L\}$ where L is the length of the DNA, i.e. the sum of lengths of A or B or C. Sort g such that g_1, g_2, ... are in sorted order. Let $c = \{c_i : c_i = g_{i+1} - g_i, 0 < i < |g| \; and \; g_{i+1} \neq g_i\}$. If $[a, b]$ is a correct solution, then c and C must be identical. We can check if they are identical by sorting them both and checking one element at a time is sorted order. The running time of this verification is $O(|g| \log |g|)$ which is definitely polynomial in the size of the solution $[a, b]$.

Second, we need to show that any problem in NP reduced to DDP in polynomial time. We will show this by showing that the set partition problem, which is NP-complete, reduces to DDP in polynomial time. The set partition problem is the following: given a set I of integers of total sum J, determine whether we can divide the set into two sets of equal sum $J/2$. Given an instance of the set partition problem (I, J), we construct the following instance of the DDP problem: $A = I$, $B = \{J/2, J/2\}$, $C = I$. Clearly this DDP problem has a solution if and only if the set partition problem has a solution.

Partial Digest Problem PDP

The experimental procedure behind DDP is easy and straight forward; however, DDP is hard to solve efficiently since it is NP-complete. We present now another approach that is easier to solve for in practice; however, involves a much harder biological experiment. The biological experiment behind this approach is called Partial Digestion, leading to the Partial Digest Problem PDP. In partial digestion we cut the DNA with one enzyme only. However, we repeat the cutting procedure many times, in each time we vary the temperature of the experiment, making the enzyme "miss" some restriction sites. Eventually we hope to produce restriction fragments for every two possible cuts and every possible cut with a boundary. This is illustrated in the figure below.

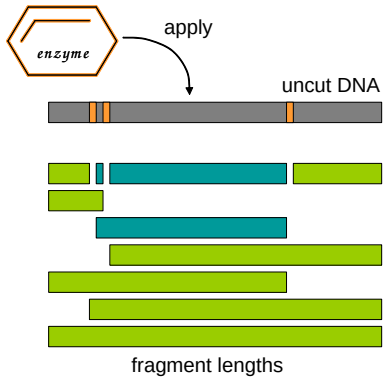


Figure 13: Partial Digestion

Now given all the obtained freagment lengths, we would like to map the restriction sites on the DNA. This is equivalent to the following problem known in computer science as the turnpike problem: given information about the distance between any pair of exits on the highway, reconstruct the exits on the highway. PDP can be stated formally as follows:

Partial Digest Problem PDP

Given a multiset ΔX of distances between every pair of points on the line, $|\Delta X| = \binom{n}{2}$,

Reconstruct the points on the line.

There is no polynomial time algorithm known for this problem. It is also not known whether it is NP-complete or not. There is a practical backtracking algorithm due to Skiena et al. 1990. Here's the algorithm:

- Find longest distance in ΔX, this decides the two outermost points, delete that distance from ΔX.
- Repeatedly position the longest remaining distance of ΔX.
- Since the longer distance must be realized from one of the two outermost points, we have two possible positions (left and right) for the point.
- For each of these two positions, check whether all the distances from the position to the points already positioned are in ΔX.
- If they are, delete all those distances from ΔX and proceed.
- Backtrack if they are not for both of the two positions.

Here's an exmaple of applying this algorithm. Let $\Delta X = \{2, 2, 3, 3, 4, 5, 6, 7, 8, 10\}$. Therefore, we have 5 points to position. The largest distance is 10. This determines the two outermost points, one in position 0 and another in position 10. We delete 10 from ΔX. Now we have $\Delta X = \{2, 2, 3, 3, 4, 5, 6, 7, 8\}$. The next largest distance is 8. We can put the corresponding point either in position 2 or position 8. At this point, it does not matter since the solution is symmetric. So let's say we put it at position 2. Now we need to delete all the distances from that new point to all previously positioned points, namely 2 and 8. Now $\Delta X = \{2, 3, 3, 4, 5, 6, 7\}$. The next largest distance is 7. We can put the point either in position 3 or position 7. Putting the point in position 3 would require us to have a distance of 1 in ΔX since we have a point at position 2. Therefore, we have to put it in position 7. Let's see if this is feasible, i.e. can we find all the distances from this new point to all the previously positioned points? We must find a 3, a 7, and a 5. All these distances are in ΔX so now we can delete them and proceed. We have $\Delta X = \{2, 3, 4, 6\}$ now. The next largest distance is 6. We can put the new point either in position 4 or position 6. Putting it in position 6 would require us to have a distance of 1 in ΔX since we have a point in position 7. Therefore it has to be in position 4. Checking the distances with all the previously positioned points we find a 2, a 3, a 4, and a 6 which are all in ΔX. We delete these distances and now $\Delta X = \emptyset$ and we stop. We have the solution $\{0, 2, 4, 7, 10\}$.

In the example above we did not backtrack. It is possible however to backtrack many times. In fact, it is possible that the number of backtracks is exponential. Constructing such an example is not trivial. Zhang 1994 provided an example. In practice, if we have real points in general positions, then one of the two choices will be pruned with probability 1. The algorithm runs in $O(n^2 \log n)$ expected time since we have $O(n^2)$ positions and all operations can be done on ΔX using binary search which takes $O(\log n)$ for each operation.

References

Waterman M., Introduction to Computational Biology, Chapter 3.
Pevzner P., Computational Molecular Biology, Chapter 2.