Лекция 12. Секвенирование ДНК

Burrows–Wheeler Transform

The Burrows–Wheeler transform (BWT) is a transformation of the text that makes it easier to compress. It can be defined as follows:

- Let $T[0..n)$ be a text. For any $i \in [0..n)$, $T[i..n)T[0..i)$ is a rotation of T. Let $\mathcal{M}$ be the $n \times n$ matrix, where the rows are all the rotations of T in lexicographical order.
- All columns of $\mathcal{M}$ are permutations of T. In particular:
  - The first column F contains the text symbols in order.
  - The last column L is the BWT of T.

Example 14: The BWT of $T =$ `banana$` is $L =$ `annb$aa`.

F						L
$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

Лекция 12. Секвенирование ДНК

Surprisingly, BWT is invertible, i.e., T can be reconstructed from the BWT L. We will later see how.

The compressibility of the BWT is based on sorting by context:

- Consider a symbol s that is followed by a string w in the text. (The text is considered to be cyclic.) The string w is called a right context of s. In the matrix $\mathcal{M}$, there is a row beginning with w and ending with s.
- Because the rows are sorted, all symbols with the right context w appear consecutively in the BWT. This part of the BWT sharing the context w is called a w-context block and is denoted by L_w.
- Context blocks are often highly compressible as they consist of symbols occurring in the same context.

Example 15: In Example 14, $L_{\texttt{n}} = \texttt{aa}$.

Here we have right contexts while earlier with higher order compression we considered left contexts. This makes no essential difference. Furthermore, the text is often reversed before computing the BWT, which turns left contexts into right contexts and vice versa.

Лекция 12. Секвенирование ДНК

Example 16: The context block $L_{\mathtt{ht}}$ for a reversed English text, which corresponds to left context `th` in unreversed text.

```
oreeereoeeieeeeaooeeeeeaereeeeeeeeeeeeereeeeeeeeeeaaeeaeeeeeeee
eaeeeeeeeeaeieeeeeeeeereeeeeeeeeeeeeeeeeeeeeeeeaeeieeeeeeaaieee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeaeieeeeeeeeeeeeeeeeeeeeeeeeeeeaee
eeeeeeeeeeeeeeeeeeereeeeeeeeeeieaeeeeieeeeaeeeeeeeeeieeeeeeeee
eeeieeeeeeeeioaaeeaoereeeeeeeeeeeaaeaaeeeeieeeeeeeieeeeeeeeaeeee
eeaeeeeeereeeaeeeeeieeeeeeeeeiieee. e  eeeeiiiiii e          ,
i   o       oo e  eiiiiee,er  ,  ,     ,  . iii
```

Some of those symbols in (unreversed) context:

```
t raise themselves, and the hunter, thankful and r
ery night it flew round the glass mountain keeping
agon, but as soon as he threw an apple at it the b
f animals, were resting themselves.  "Halloa, comr
ple below to life.  All those who have perished on
 that the czar gave him the beautiful Princess Mil
ng of guns was heard in the distance.  The czar an
cked magician put me in this jar, sealed it with t
o acted as messenger in the golden castle flew pas
u have only to say, 'Go there, I know not where; b
```

Лекция 12. Секвенирование ДНК

The context blocks are closely related to empirical entropies.

Theorem 17: For any text T and any k,

$$\sum_{w\in\Sigma^k} |L_w| H_0(L_w) = |T| H_k(T).$$

Proof. Let us first note that the equation for the kth order entropy is symmetric. Thus it does not matter whether we talk about the text or its reversal, or about the left or the right context.

Recall that n_w is the number of occurrences of w in the text. Then $n_w = |L_w|$ and

$$H_0(L_w) = -\sum_{s\in\sigma} \frac{n_{sw}}{n_w} \log \frac{n_{sw}}{n_w} = -\sum_{s\in\sigma} \frac{n_{ws}}{n_w} \log \frac{n_{ws}}{n_w}.$$

The strings L_w, $w \in \Sigma^k$, are distinct, i.e, they form a partitioning of the BWT into σ^k blocks (some of which may be empty). Furthermore,

$$\sum_{w\in\Sigma^k} |L_w| H_0(L_w) = -\sum_{w\in\Sigma^k} n_w \sum_{s\in\sigma} \frac{n_{ws}}{n_w} \log \frac{n_{ws}}{n_w} = nH_k(T).$$

Лекция 12. Секвенирование ДНК

According to the theorem, zeroth order compression of the context blocks achieves kth order compression of the text. This is known as compression boosting.

As we noted earlier, using a single value of k everywhere is not optimal in general. There exists a linear time algorithm for finding a sequence of variable length contexts $w_1, w_2, \ldots, w_h$ such that $L_{w_1} L_{w_2} \ldots L_{w_h} = L$ and the total compressed size is minimized. This is called optimal compression boosting.

For the best compression, we may need to take multiple context lengths into account. With BWT this is fairly easy using adaptive methods:

- For most symbols s in the BWT, the nearest preceding symbols share a long context with s, symbols that are a little further away share a short context with s, and symbols far away share no context at all.
- Thus adaptive methods that forget, i.e., give higher weight to the nearest preceding symbols are often good compressors of the BWT.
- Such methods can be context oblivious: They achieve good compression for all contexts without any knowledge about context blocks or context lengths.

Лекция 12. Секвенирование ДНК

Run-length encoding

An extreme form of forgetting is run-length encoding (RLE). RLE encodes a run s^k of k consecutive occurrences of a symbol s by the pair $\langle s, k \rangle$. Nothing beyond the run has an effect on the encoding.

Example 18: The run-length encoding of $L_{\texttt{ht}}$ from Example 16 begins: $\langle \texttt{o}, 1 \rangle$ $\langle \texttt{r}, 1 \rangle$ $\langle \texttt{e}, 3 \rangle$ $\langle \texttt{r}, 1 \rangle$ $\langle \texttt{e}, 1 \rangle$ $\langle \texttt{o}, 1 \rangle$ $\langle \texttt{e}, 2 \rangle$ $\langle \texttt{i}, 1 \rangle$ $\langle \texttt{e}, 4 \rangle$ $\langle \texttt{a}, 1 \rangle$ $\langle \texttt{o}, 2 \rangle$ $\langle \texttt{e}, 5 \rangle$.

RLE can be wastful when there are many runs of length one. A simple optimization is to encode the (remaining) run-length only after two same symbols in a row. This could be called lazy RLE.

Example 19: The lazy RLE of $L_{\texttt{ht}}$ from Example 16 begins: oree1reoee0iee2aoo0ee3.

RLE alone does not compress the BWT very well in general but can be useful when combined with other methods.

Лекция 12. Секвенирование ДНК

Move-to-front

Move-to-front (MTF) encoding works like this:

- Maintain a list of all symbols of the alphabet. A symbol is encoded by its position on the list. Note that the size of the alphabet stays the same.
- When a symbol occurs, it is moved to the front of the list. Frequent symbols tend to stay near the front of the list and are therefore encoded with small values.

After an MTF encoding of the BWT, the smallest values tend to be the most frequent ones throughout the sequence. Thus a single global model can achieve a good compression. This is what makes MTF encoding context oblivious.

Example 20: The MTF encoding of $L =$ annb$aa is 0202330.

list	next symbol	code
abn$	a	0
abn$	n	2
nab$	n	0
nab$	b	2
bna$	$	3
$bna	a	3
a$bn	a	0

Лекция 12. Секвенирование ДНК

There are also other techniques that transform the BWT into a sequence of numbers.

Whatever the final sequence is — the plain BWT, an RLE encoded BWT, an MTF encoded BWT, or something else — it needs to be compressed using an entropy encoder to achieve the best compression. However, the model needed is much simpler than what would be needed for direct encoding of the text. For example:

- The plain BWT can be compressed well using a single zeroth order adaptive model. For a similar compression rate, the text needs to be encoded using a complex higher order model.
- The MTF encoded BWT can be compressed well using a single zeroth order semiadaptive model. An equivalent direct text compression would need a higher order semiadaptive model.

Лекция 12. Секвенирование ДНК

Computing and inverting BWT

Let us assume that the last symbol of the text $T[n-1] = \$$ does not appear anywhere else in the text and is smaller than any other symbol. This simplifies the algorithms.

To compute the BWT, we need to sort the rotations. With the extra symbol at the end, sorting rotations is equivalent to sorting suffixes. The sorted array of all suffixes is called the suffix array (SA).

F						L
$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

SA						
6	$					
5	a	$				
3	a	n	a	$		
1	a	n	a	n	a	$
0	b	a	n	a	n	a	$
4	n	a	$			
2	n	a	n	a	$	

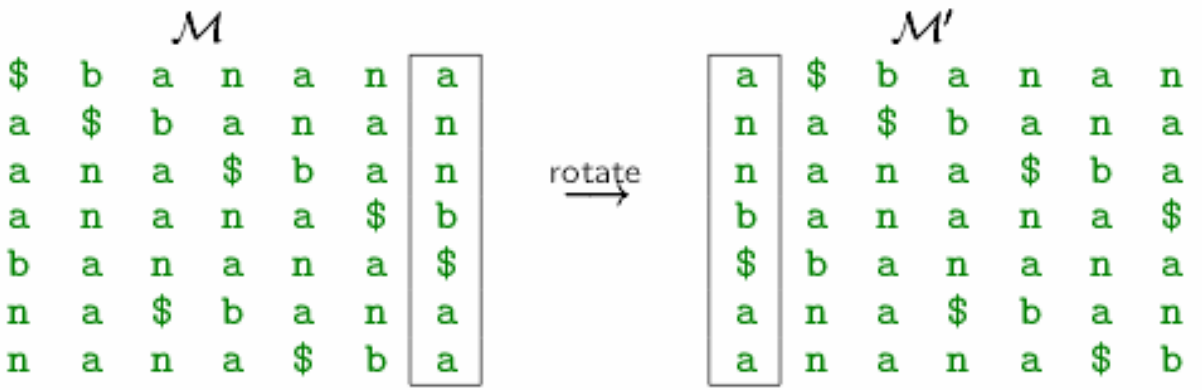
There are linear time algorithms for suffix sorting. The best ones are complicated but fairly fast in practice. We will not described them here, but they are covered on the course *String Processing Algorithms*.

Лекция 12. Секвенирование ДНК

We will take a closer look at inverting the BWT, i.e., recovering the text T given its BWT.

Let $\mathcal{M}'$ be the matrix obtained by rotating $\mathcal{M}$ one step to the right.

Example 21:



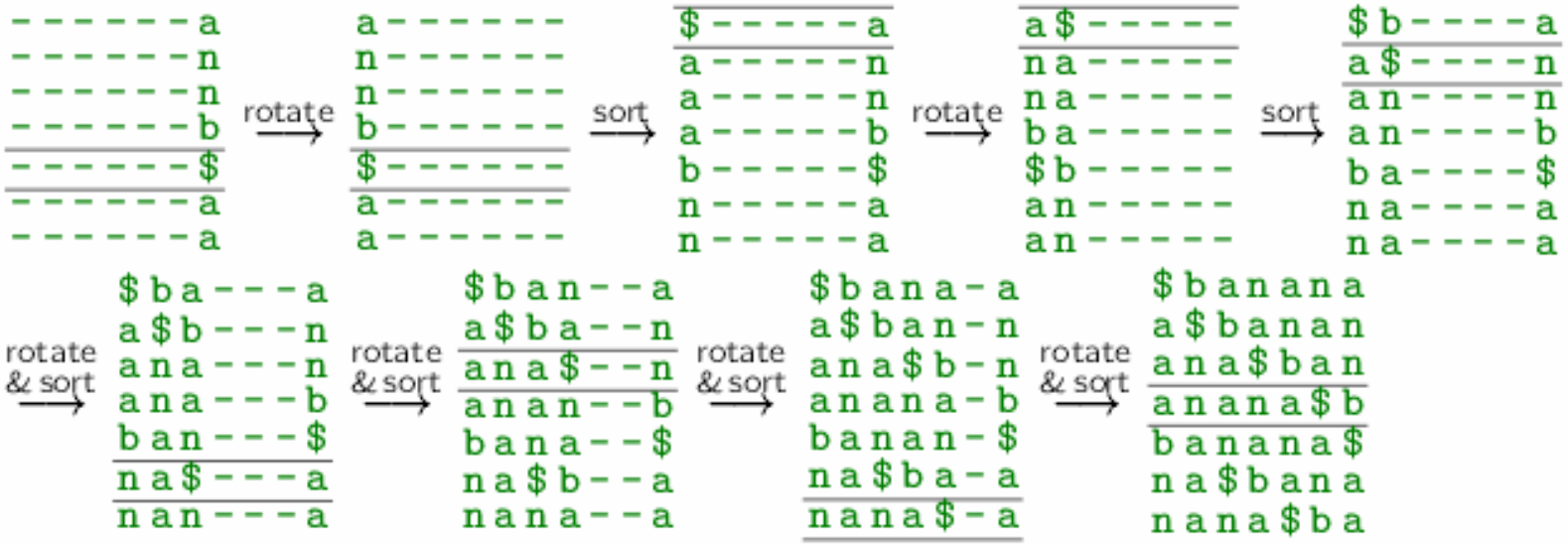
- The rows of $\mathcal{M}'$ are the rotations of T in a different order.
- In $\mathcal{M}'$ without the first column, the rows are sorted lexicographically. If we sort the rows of $\mathcal{M}'$ stably by the first column, we obtain $\mathcal{M}$.

This cycle $\mathcal{M} \xrightarrow{\text{rotate}} \mathcal{M}' \xrightarrow{\text{sort}} \mathcal{M}$ is the key to inverse BWT.

Лекция 12. Секвенирование ДНК

- In the cycle, each column moves one step to the right and is permuted. The permutation is fully determined by the last column of $\mathcal{M}$, i.e., the BWT.
- By repeating the cycle, we can reconstruct $\mathcal{M}$ from the BWT.
- To reconstruct T, we do not need to compute the whole matrix just one row.

Example 22:

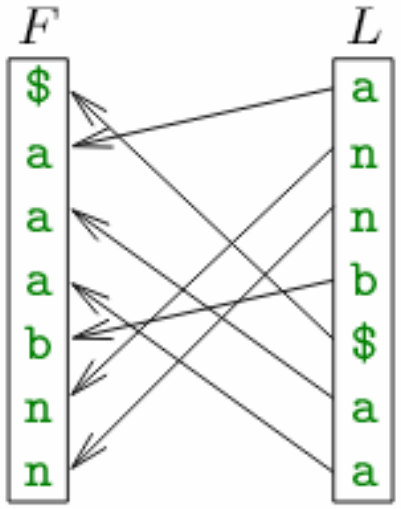


Лекция 12. Секвенирование ДНК

The permutation that transforms $\mathcal{M}'$ into $\mathcal{M}$ is called the LF-mapping.

- LF-mapping is the permutation that stably sorts the BWT L, i.e., $F[LF[i]] = L[i]$. Thus it is easy to compute from L.
- Given the LF-mapping, we can easily follow a row through the permutations.

Example 23:



Лекция 12. Секвенирование ДНК

Here is the algorithm.

Algorithm 24: Inverse BWT
Input: BWT $L[0..n]$
Output: text $T[0..n]$
Compute LF-mapping:
(1) for $i \leftarrow 0$ to n do $R[i] = (L[i], i)$
(2) sort R (stably by first element)
(3) for $i \leftarrow 0$ to n do
(4) $(\cdot, j) \leftarrow R[i]$; $LF[j] \leftarrow i$
Reconstruct text:
(5) $j \leftarrow$ position of \$ in L
(6) for $i \leftarrow n$ downto 0 do
(7) $T[i] \leftarrow L[j]$
(8) $j \leftarrow LF[j]$
(9) return T

The time complexity is linear if we sort R using counting sort.

Лекция 12. Секвенирование ДНК

Summary of BWT algorithm

Suffix array of string X:

S(i) = j, where $X_j \ldots X_n$ is the j-th suffix lexicographically

- BWT follows immediately from suffix array
  - Suffix array construction possible in O(n), many good O(n log n) algorithms
- Reconstruct X from BWT(X) in time O(n)
- Search for all exact occurrences of W in time O(|W|)
- BWT(X) is easier to compress than X

Лекция 12. Секвенирование ДНК

The de Bruijn graph

Structure

In the de Bruijn graph, each node N represents a series of overlapping k-mers (cf. Fig. 1 for a small example). Adjacent k-mers overlap by $k - 1$ nucleotides. The marginal information contained by a k-mer is its last nucleotide. The sequence of those final nucleotides is called the sequence of the node, or $s(N)$.

Each node N is attached to a twin node $\tilde{N}$, which represents the reverse series of reverse complement k-mers. This ensures that overlaps between reads from opposite strands are taken into account. Note that the sequences attached to a node and its twin do not need to be reverse complements of each other.

The union of a node N and its twin $\tilde{N}$ is called a "block." From now on, any change to a node is implicitly applied symmetrically to its twin. A block therefore has two distinguishable sides, in analogy to the "k-mer edges" described in Pevzner et al.'s 2001 paper.

Nodes can be connected by a directed "arc." In that case, the last k-mer of an arc's origin node overlaps with the first of its destination node. Because of the symmetry of the blocks, if an arc

Лекция 12. Секвенирование ДНК

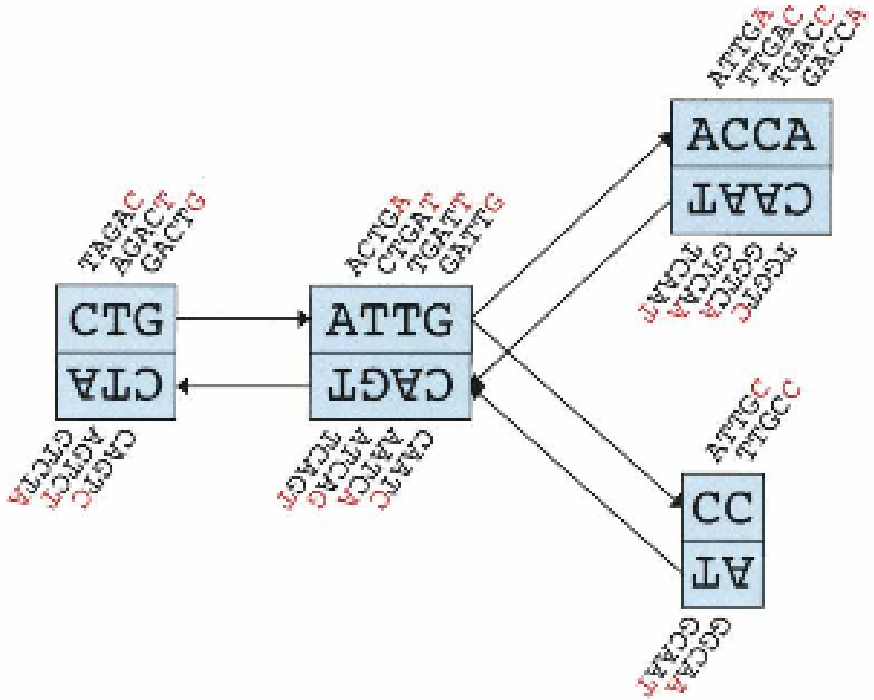


Figure 1. Schematic representation of our implementation of the de Bruijn graph. Each node, represented by a single rectangle, represents a series of overlapping *k*-mers (in this case, $k = 5$), listed directly *above* or *below*. (Red) The last nucleotide of each *k*-mer. The sequence of those final nucleotides, copied in large letters in the rectangle, is the sequence of the node. The twin node, directly attached to the node, either *below* or *above*, represents the reverse series of reverse complement *k*-mers. Arcs are represented as arrows *between* nodes. The last *k*-mer of an arc's origin overlaps with the first of its destination. Each arc has a symmetric arc. Note that the two nodes on the *left* could be merged into one without loss of information, because they form a chain.

Лекция 12. Секвенирование ДНК

goes from node A to B, a symmetric arc goes from $\bar{B}$ to $\bar{A}$. Any modification of one arc is implicitly applied symmetrically to its paired arc.

On these nodes and arcs, reads are mapped as "paths" traversing the graph. Extracting the nucleotide sequence from a path is straightforward given the initial k-mer of the first node and the sequences of all the nodes in the path.

Construction

The reads are first hashed according to a predefined k-mer length. This variable k is limited on the upper side by the length of the reads being hashed, to allow for a small amount of overlap, usually $k = 21$ for 25-bp reads. Smaller k-mers increase the connectivity of the graph by simultaneously increasing the chance of observing an overlap between two reads and the number of ambiguous repeats in the graph. There is therefore a balance between sensitivity and specificity determined by k .

Лекция 12. Секвенирование ДНК

For each *k*-mer observed in the set of reads, the hash table records the ID of the first read encountered containing that *k*-mer and the position of its occurrence within that read. Each *k*-mer is recorded simultaneously to its reverse complement. To ensure that each *k*-mer cannot be its own reverse complement, *k* must be odd. This first scan allows us to rewrite each read as a set of original *k*-mers combined with overlaps with previously hashed reads. We call this new representation of the read's sequence the "roadmap."

A second database is created with the opposite information. It records, for each read, which of its original *k*-mers are overlapped by subsequent reads. The ordered set of original *k*-mers of that read is cut each time an overlap with another read begins or ends. For each uninterrupted sequence of original *k*-mers, a node is created.

Finally, reads are traced through the graph using the roadmaps. Knowing the correspondence between original *k*-mers and the newly created nodes, Velvet proceeds from one node to the next, creating a new directed arc or incrementing an existing one as appropriate at each step.

Лекция 12. Секвенирование ДНК

Simplification

After constructing the graph, it is generally possible to simplify it without any loss of information. Blocks are interrupted each time a read starts or ends. This leads to the formation of "chains" of blocks, or linear connected subgraphs. This fragmentation of the graph costs memory space and lengthens calculation times.

These chains can be easily simplified. Whenever a node A has only one outgoing arc that points to another node B that has only one ingoing arc, the two nodes (and their twins) are merged. Iteratively, chains of blocks are collapsed into single blocks.

The simplification of two nodes into one is analogous to the conventional concatenation of two character strings, and also to some string graph based methods . This straightforward transformation involves transferring arc, read, and sequence information as appropriate.

Лекция 12. Секвенирование ДНК

k-mer

"*k*-mer" is a substring of length *k*

S: GGCGATTCATCG

A 4-mer of *S*: ATTC

All 3-mers of *S*:
GGC
GCG
CGA
GAT
ATT
TTC
TCA
CAT
ATC
TCG

mer: from Greek meaning "part"

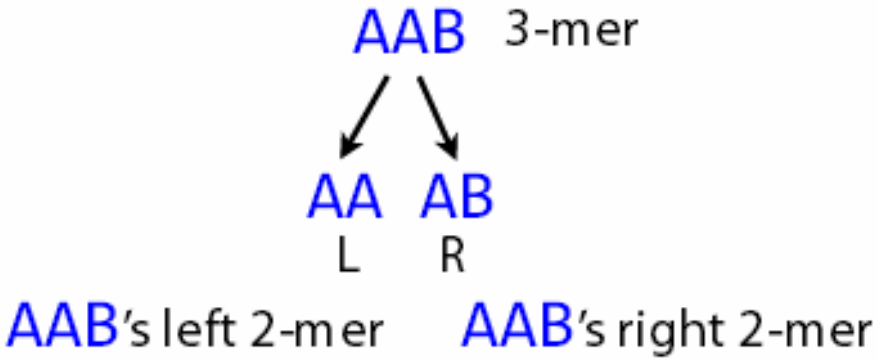
I'll use "*k*-1-mer" to refer to a substring of length *k* - 1

Лекция 12. Секвенирование ДНК

As usual, we start with a collection of reads, which are substrings of the reference genome.

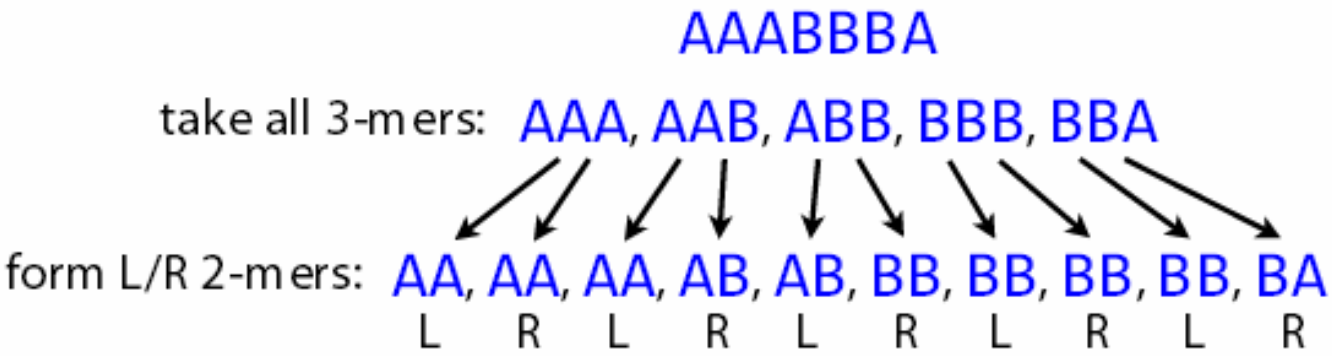
AAA, AAB, ABB, BBB, BBA

AAB is a *k*-mer ($k = 3$). AA is its *left* *k*-1-mer, and AB is its right *k*-1-mer.

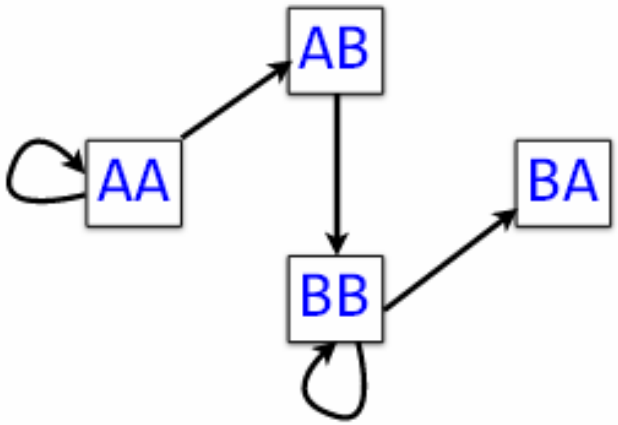


Лекция 12. Секвенирование ДНК

Take each length-3 input string and split it into two overlapping substrings of length 2. Call these the *left* and *right 2-mers*.

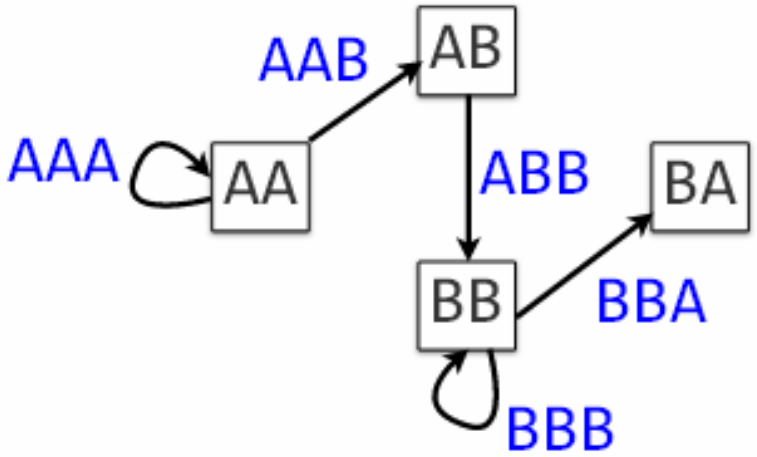


Let 2-mers be nodes in a new graph. Draw a directed edge from each left 2-mer to corresponding right 2-mer:



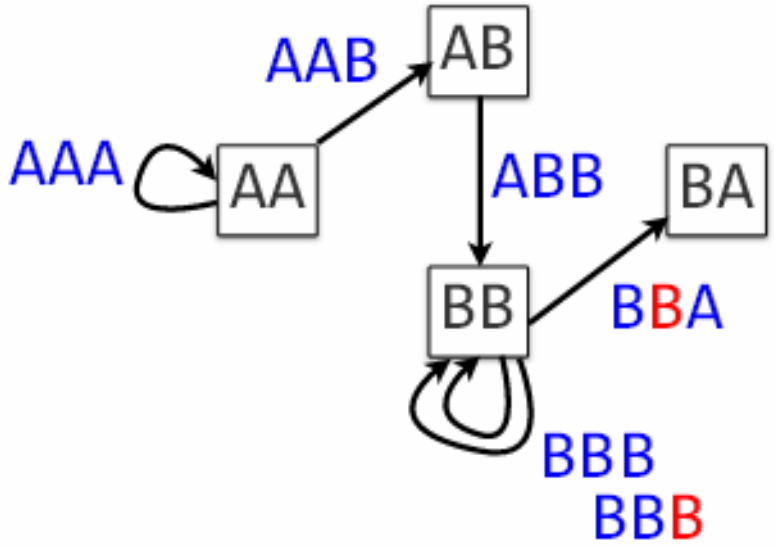
Each *edge* in this graph corresponds to a length-3 input string

Лекция 12. Секвенирование ДНК



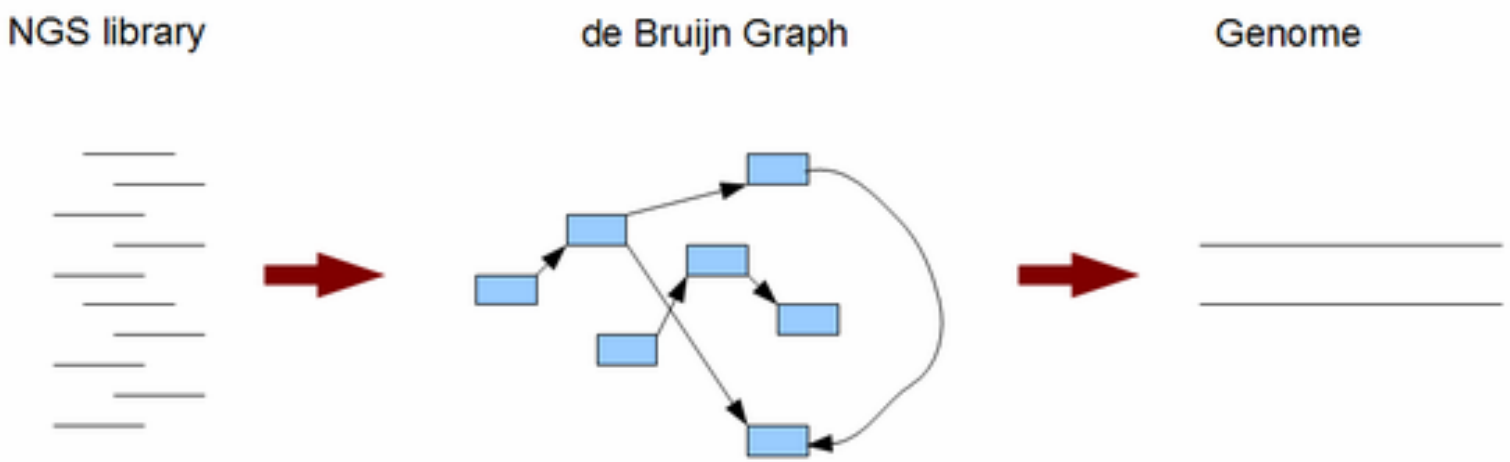
An edge corresponds to an overlap (of length k-2) between two k-1 mers.
More precisely, it corresponds to a k-mer from the input.

Лекция 12. Секвенирование ДНК



If we add one more B to our input string: AAABBBBA, and rebuild the De Bruijn graph accordingly, we get a *multiedge*.

Лекция 12. Секвенирование ДНК



Typically a de Bruijn graph-based genome assembly algorithm works in two steps. In the first step, short reads are broken into small pieces (k-mers) and a de Bruijn graph is constructed from those short pieces. In the next step, the genome is derived from the de Brujin graph.