Advanced Algorithms for Bioinformatics (P4)

RNA-Sequencing Script

K. Reinert & S. Andreotti

Nicolas Balcazar Corinna Blasse An Duc Dang
Hannes Hauswedell Sebastian Thieme

SoSe 2010

Contents

1 Introduction

DNA is a molecule which living organisms use to encode themselves and to pass on this information to their offspring. To understand the fascination of life, we need on the one hand knowledge of the coding and regulatory regions within the DNA and on the other hand knowledge about processes inside a cell. To acquire knowledge about the DNA, we need the sequence of the nucleotides. Several sequencing methods have been proposed since the early 70s. The first such sequencing method devloped is calle dSanger sequencing after his inventor Frederick Sanger.

1.1 Sanger biochemistry

This sequencing method was first developed in the early 1970s. Since the early 1990s, it exists semi-automated implementations of this method. Today, there are two approaches to sequence DNA in high-throughput pipelines based on the approach. The first approach clones randomly fragmented DNA pieces into a plasmid . The second approach used PCR amplification with primer which flank the target (targeting resequencing). The output of both is an amplified template. An amplified template means the amplification of the original DNA sequence. The following description is for targeting resequencing with PCR. The sequencing step takes place in a 'cylce sequencing' reaction with several rounds. A complementary primer is annealing to the known sequence of interest. In each round of primer extension, there is a stochastic probability to terminate by annealing a fluorescent labelled dideoxynucleotides (ddNTPs). This kind of termination results in a mixture of end-labelled sequences. Each label, on the terminating ddNTP of any given fragment, corresponds to the nucleotide identity of its terminal position. The sequences are determined by gel electrophoresis followed by a laser excitation of the fluorescence labels coupled to four color detection of emission spectra. A software translates these detection into DNA sequence. This method provides a limited level of parallelization because of a limitation in simultaneous electrophoresis. Although sequencing with Sanger is also cost and time expensive, if it is used, a read-length of up to around 1000bp can be achieved. DNA is as the carrier of hereditary information inherently static. Only by the expression of genes within a cell proteins are produced and metabolism takes place. In order to understand the processes inside a cell, there is hence a high interest in messenger RNA (mRNA) which transfers the transcript gneomics information to the the ribosomes. Depending on processes not yet

completely understood and which are caused by different regulatory mechanisms genetic information can be transcribed in several isoforms resultung a a variety of different mRNAs.

1.1.1 Microarray

Microarray is also a general term for different methods, for example DNA-microarray or protein-microarray. This approach allows an analysis of thousands of biological samples simultaneously with only a small amount of biological sample. DNA-microarray is an example for sequencing by hybridisation. Usually oligos are designed as a representation of the reference sequence observing several criteria like sufficient genome coverage, similar melting temperature, no self complementary, and (near) uniqueness in the genome. This method is cost efficient for genomic resequencing but is also used to determine the quantity of mRNA expression of several genes. The protocol for measuring mRNA levels of a probe first extracts the mRNA from the cell. After several purification and amplifications steps, the mRNA is transformed in cDNA and fixed on glass or membrane slides. For a better comparison, a reference is printed on the slide too. To distinguish both, the transformed mRNA and the reference are fluorescent labelled with different colours, Carbocyanin 3 (Cy3;green emission) or Carbocyanin 5 (Cy5;red emission),respectively. After the hybridisation step the emission of each position on the microarray is scanned with a laser determining the fluorescence intensity. One limitation of this method is that sequences which are repetitive or subject to cross hybridisation cannot easily be identified. Furthermore, there is a limitation to the abiity to detect RNA splicing events or detect novel genes.

1.2 Next-generation sequencing

Today, another approach is available, called next-generation sequencing (NGS), which subsumes a couple of different technologies, liek foer exmaple thos by Illumina (Solexa), Applied Biosystems (SOLID), or Roche (454). Compared with the Sanger biochemistry the NGS is less expensive for comprehensive analysis of genomes, transcriptomes and interactomes. In this script we will concentrate on RNA-Sequencing.

1.2.1 RNA-Seq

RNA-Sequencing is also called "Whole Transcriptome Shotgun Sequencing". This is a technique to sequence cDNA in order to get information about the RNA sample content. As it was pointed out above DNA cannot give us all informations for understanding cell processes but the transcriptome can. Hence RNA-Seq aims at determining the level of mRNA expression of a cell at a given point in time by sequencing the corresponding cDNA.

1.2.2 Advantages of RNA-Sequencing

RNA-Sequencing is an high-throughput analysis method and hence we can get a lot of information in a short time at low costs. Furthermore, no bacterial cloning of cDNA needed and hence there are no bacterial cloning artefacts which affect the data or the experiment. In contrast to microarrays, this method reached a high coverage and new exons and genes can be discovered. Additionally, this method can be used to handle detect different RNA isoforms.

There are different implementations of RNA-Sequencing pipelines, in this script we will explain the ERANGE - pipeline (figure: 1.2.3) briefly [6] .

1.2.3 ERANGE - Pipeline

This pipeline is one example of RNA-Sequencing pipelines to explain the main procedures of this method. in a first step the RNA will be purified by poly(A)-selection. After two rounds of poly(A) selection, the RNA is fragmented to an average length of 200 nucleotides. RNA is unstable and quickly degrades because of ubiquitous RNase molecules. Hence we have to convert the RNA into cDNA before degradation. This is done by random priming. The cDNA will sequenced and we obtain reads that are subsequently mapped to a genome. The normalized transcript prevalence is calculated with an algorithm from the ERANGE package.

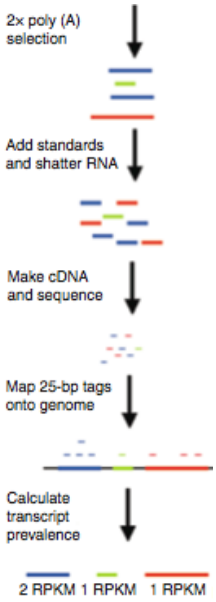


Figure 1: This figure shows the main steps of the ERANGE-pipeline described in the text.

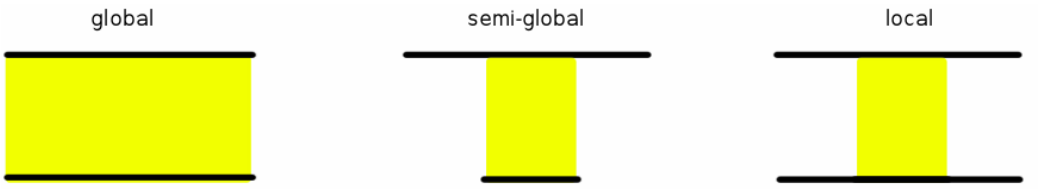


Figure 2: alignment variants

1.2.4 Read-mapping

There are three main alignment approaches, global alignment, local alignment and semi-global alignment (figure: 2). A global alignment is the alignment of two sequences with approximately the same length like in the Needleman-Wunsch algorithm. Here we calculate the optimal-similarity score of a pairwise alignment to make a point about the similarity between both sequences. If we want to study the similarity of two sub-strings like domains of proteins we use local alignment (Smith-Waterman algorithm). The alignment between a large sequence and a shorter one is called semi-global alignment, because we calculate a global alignment of the shorter sequence within a sub-sequence of the longer one. This approach is used to align reads onto the genome. The term read-mapping describes the task of computing semi-global alignments of many short sequences (reads) to a long sequence (the genome). There are different algorithms for semi-global alignments available. To choose the correct one there are different questions which could help.

Important Questions regarding choice of algorithm and implementation:

Input How long are the reads? How many are there?

How big is the genome?

What sequencing technology was used?

ist there a known error-profile or quality values?

Are gapped or ungapped alignments needed?

What kind of scoring / distance measure is adequate?

According to the answer of the questions above we have different algorithms to working with. Tools which are designed for short sequences allowing a limited number of mismatches or short gaps e.g. Bowtie or MAQ. If we want allow a higher number of mismatches and work with longer sequences we could use tools like RazerS or MrsFast.

In general we divide mapping tools into two categories. 1) tools using a filtering step with subsequent verification and 2) tools that presprocess the genome in an index and search directly for approximate matches using this index. In this script we discuss approaches out of both categories. In the first part we discuss the Bowtie algorithm especially the Burrows-Wheeler-Transformation (BWT) and L-to-F mapping. Furthermore, we show how we could find an exact-match and an inexact-match. In a second part we give an introduction in different filtering approaches like PEX and q-gram-counting.

2 Definitions

An *alphabet* Σ is a nonempty set of symbols (also called letters). In the following, Σ will always denote a finite and ordered alphabet.

A *pattern* P over an alphabet Σ is a sequence of elements of Σ.

The *length* of a pattern P is denoted by P.

P[i] denotes the symbol at the i-th position of P. The *substring P[i,j]* is a pattern of symbols from the i-th to the j-th position in P.

3 Suffix Arrays (Short recapitulation)

A suffix array is an array of integers containing the starting positions of suffixes of a string in lexicographical order of the corresponding suffixes.

Definition. Given a text T of length n, the suffix array for T, called suftab, is an array of integers of range 1 to $n+1$ specifying the lexicographic ordering of the $n+1$ suffixes of the string $T\$$.

3.0.5 Example 1

Consider the string `dog$`. The suffix array for `dog$` is **[4,1,3,2]**, because the suffixes of `dog$` are `$`, `dog$`, `g$` and `og$` (in this lexicographic order).

3.0.6 Example 2

Consider the string `abracadabra$`. In this case `$` represents a character appearing only once and which is lexicographically less than any other letter in the string which we have to introduce so that no suffix can be a prefix of another suffix.

1	2	3	4	5	6	7	8	9	10	11	12	index
a	b	r	a	c	a	d	a	b	r	a	$	string

It has twelve suffixes:		index	sorted suffix
abracadabra$		12	$
bracadabra$		11	a$
racadabra$		8	abra$
acadabra$		1	abracadabra$
cadabra$	that can be sorted	4	acadabra$
adabra$	into lexicographical	6	adabra$
dabra$	order to obtain:	9	bra$
abra$		2	bracadabra$
bra$		5	cadabra$
ra$		7	dabra$
a$		10	ra$
$		3	racadabra$

⇒ For the string `abracadabra$`, the suffix array is **[12, 11, 8, 1, 4, 6, 9, 2, 5, 7, 10, 3]**.

There is a number of linear time agorithms available to construct a suffix array.

4 The Burrows-Wheeler-Transformation

The Burrows-Wheeler transformation (BWT) was published by Michael Burrows and David Wheeler in 1994 [[3]] . It is a reversible lossless transformation algorithm, which permutes an input string into a new string. The string obtained from the BWT lends itself to an effective compression.

obacht: cites added (Dang)

4.1 Principle of BWT

The input for the BWT algorithm is a string S of length N with characters of the ordered alphabet Σ. In the following explanation the algorithm will be applied to this example input:

$S = mississippi,\ N = 11, \Sigma = \{i, m, p, s\}$

Then the BWT proceeds as follows: 1. Sort rotations

First we append at the end of S an special character, which does not belong to given ordered alphabet Σ, e.g. #. Moreover this special character # is lexicographically minimal meaning it is followed by all characters of Σ. As a result the new string $S' = S\#$ has $N + 1$ characters. In the next step a conceptual $(N + 1) \times (N + 1)$ matrix M whose rows are cyclic shifts of S is constructed. Afterwards all rows are sorted lexicographically (see Fig. 3). Remember that the matrix is just for visualization and does not exist in the implementation.

obacht: Good (Reinert)

2. Find last characters of rotations

Now, only the last and first column of matrix M are considered. Let they be string L respectively F, e.g.:

$L = ipssm\#pissii$ and $F = \#iiiimppssss$

Notice that the i-th character in L precedes in S the i-th character in F, since each row is a cyclic rotation. Furthermore there is another property of the Burrows-Wheeler transformation called rank preserving property:

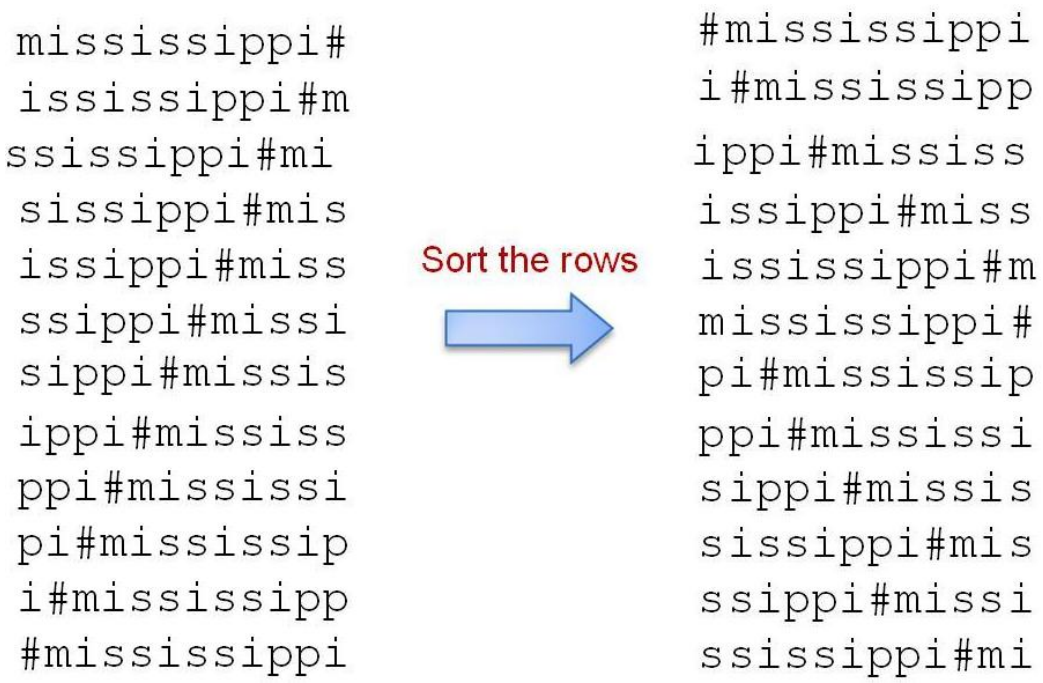


Figure 3: Conceptual matrix of the Burrows-Wheeler transformation for input example $S = mississippi$. All rows are cyclic rotations of $S\#$ and sorted lexicographically.

obacht: new formulation applied (Dang)

Lemma 4.1 (Rank preserving property)**.** *Let $c \in \Sigma$ be a fixed character. Furthermore let i and j with $i < j$ be two indices with $c = L[i] = L[j]$ and let M_i and M_j be the corresponding cyclic permutations. Then let k and l be the indices in F corresponding to the cyclic permutations M_i^r and M_j^r where M_x^r is obtained from M_x by shifting cyclically one letter to the right. Then it holds that $k < l$.*

Proof: M is lexicographically sorted and since $i < j$ it must hold that the $M_i^r[2, N+1]$ is lexicographically smaller than $M_j^r[2, N+1]$. Also observe that M_i^r and M_j^r both start with the same character c by assumption which does not change the order of M_i^r and M_j^r. Hence $k < l$.

The output of the BWT is the string L of length $N+1$ and the index I, which is the number of the row, where the input S' occurs (e.g. $I = 5$, counting from zero).

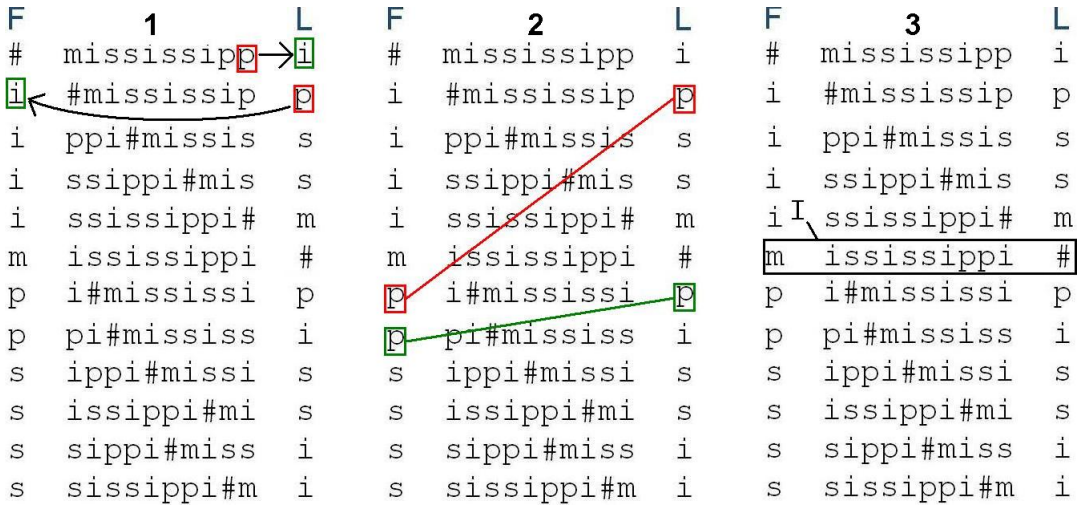


Figure 4: **1:** Due to the cyclic rotation all L[i] characters precede all characters F[i] in a cyclic rotation **2:** The BWT is rank preserving, i.e. the first p occurring in F corresponds to the first occurring p in L; the second p in F corresponds to the second p in L etc. for all characters. **3:** Index I is the row number where the unchanged string $S\#$ occurs.

4.2 BWT and suffix arrays

Since the BWT only stores one string characters, it is more memory efficient than suffix arrays which store integers. This string is over an alphabet Σ with only four letters A, C, G, T standing for the four nucleic acids. The memory space needed for a string of size n is n bytes (size of a character). On the contrary the suffix array will have to store for each base in the DNA a distinct integer number allocating 4 bytes (or 8 for 64 bit architectures) for each entry. Additionally using the suffix array the sequence information of the DNA needs to be saved, too, whereas this information can be read out of the BWT output, since it is a reversible algorithm. However, for reverting the BWT naively additional $4n$ bytes are needed, because of the mapping vector explained in chapter 4.3 on the following page. There are trade-off strategies to reduce the space consumption (see chapter 5.3, on page 20)

obacht: references added (Dang)

Due to many repetition patterns in a big input S the BWT matrix has rows ending with the same characters. Therefore L will have substrings of repeating characters, which is useful for compression algorithms, e.g. the Move-to-Front algorithm [[2]] combined with the Run-Length encoding [[1]]. Suffix arrays cannot be compressed this way, because they contain distinct integer values by definition. To determine the BWT one can use the suffix array, because the array contains the indices of the lexicographically sorted

suffixes in S, which would be F. Since all characters of F are successors of L, L is determined easily.

4.3 Retransformation

As mentioned before the BWT is reversible. In the following chapter the naive algortihm is explained, whereas chapter 4.4 on this page introduces another more efficient algorithm.
Using the input of the retransformation algorithm which is the string L of length $N+1$ with characters of the ordered alphabet $\Sigma' = \Sigma \cup \{\#\}$ and the index I we will get the original string S, e.g.:

$L =$ `ipssm#pissii`, $N = 11$, $\Sigma = \{i, m, p, s\}$ and $I = 5$

1. Find F using L

Since every column in matrix M is a permutation of the string $S\#$, the first column F and the last column L also are permutations of $S\#$ and therefore of one another. Since M has sorted rows and F is the first column, F is sorted. Thus, one can determine F by sorting L.

2. F to L mapping

In the next step we build the mapping vector T where $T[i]$ is the position of the character in L corresponding to character $F[i]$ according to the rank preserving property. In our example this would be:

$T = \{5, 0, 7, 10, 11, 4, 1, 6, 2, 3, 8, 9\}$

To recover S we start at the position of # in L, which is the last character in $S\#$. As already known the successor of # is the character of F in the same row, e.g.: `m` is the first character of S.

Afterwards we use our mapping vector T to get back from the `m` character in F to the corresponding `m` character in L. Then we jump from the last character of this row to the first character to get our next successor character in S and we get `mi` in this example.

The iteration repeats again by using the mapping vector T to get back from the `i` character in F to the corresponding `i` character in L and then by

jumping to the front to get `mis`. The algorithm terminates, when # in F is reached (see Fig 5).

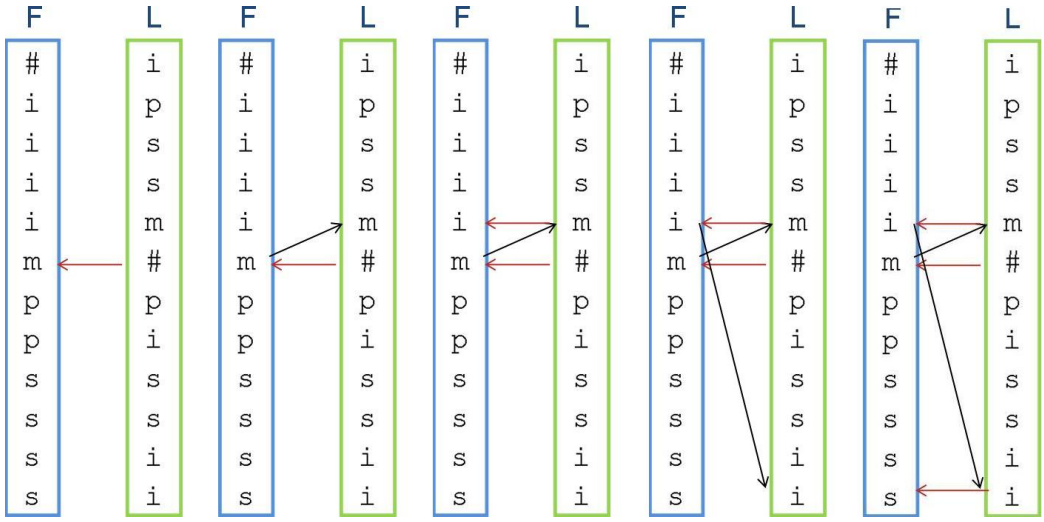


Figure 5: Recovering the first 3 characters of example string $S =$ `mississippi`. Starting at the special character # in L one jumps to the first character in the row to get the first character `m` of S in the first matrix column F. Afterwards one gets the corresponding `m` in L using the mapping vector T. Repeat this procedure until # in F is reached.

4.4 L-to-F mapping

In contrast to the previous section the following retransformation procedure uses mainly the L-to-F mapping. The L-to-F mapping maps i to j, where $L[i]$ and $F[j]$ correspond to the same character in the string S, i.e. $M_i^r = M_j$ (see section 4.1).
In addition to the retransformation, the L-to-F mapping can be used to search a pattern in the BWT of a text. The search algorithm iteratively searches the pattern from back to front and in each steps stores an interval of matrix rows starting with the current pattern suffix. The L-to-F mapping is used to find the cyclic right shifts of these rows starting with the previous pattern letter and thus representing matches of the pattern suffix which is one character longer. The repetition of this procedure results in the interval of rows starting with the whole pattern.
The L-to-F mapping needs the two structures C and Occ, which can be determined from the BWT.

C[c] C[c] contains for each c $\in \Sigma$ the total number of characters in T which are lexicographically smaller than c.

Example: C[] for `mississippi#`

i	m	p	s
1	5	6	8

Occ(c,q) Occ(c,q) is the number of occurrences of character c in prefix L[1,q].

Example: L = `ipssm#pissii` Occ(s,5) = 2,
Occ(s,12) = 4

The occurrence should be calculated for each character of the alphabet and each position in L. That means this mapping contains $\mid \Sigma \mid * \mid L \mid$ items. Without compression the implementation of this mapping would require O($\mid \Sigma \mid \cdot \mid L \mid$) space.

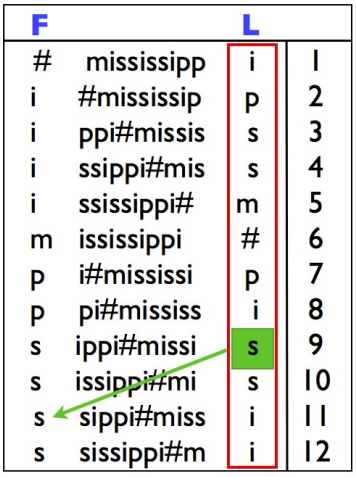
For mapping a character L[i] to F, you just need L, C, Occ and the formula

$$\underline{L - to - F}(i) = C[L[i]] + Occ(L[i], i)$$

What does this formula compute in detail?
C[L[i]] is the position before the first L[i] in F. Occ(L[i], i) is the number of occurrences of L[i] in string L[1,i]. That means the position before the first L[i] is calculated and j is added if L[i] is the j-th occurrence in L.

Example: i = 9

$$\begin{aligned} \underline{L - to - F}(9) &= C[L[9]] + Occ(L[9], 9) \\ &= C[s] + Occ(s, 9) \\ &= 8 + 3 = 11 \end{aligned}$$



5 EXACTMATCH

EXACTMATCH [4] is an algorithm that finds all occurrences of a pattern P in a text by using the Burrows-Wheeler-Transformation of the text and the help tables Occ and C introduced in chapter 4.4. One major advance compared to other exact string matching algorithms is the O(P) running time.
The algorithm uses the property that in the lexicographically sorted matrix M the rows beginning with a given sequence appear consecutively. In short the algorithm calculates the range of matrix rows beginning with successively longer suffixes of the read. This calculation is repeated until the suffix is the complete read. With each step the number of relevant matrix rows either shrinks or remains the same. In the end the algorithm determines either possible matching positions or an empty range of matrix rows (that means the read doesn't occur in the text).

5.1 Backward search algorithm

The *backward search algorithm* can be used to count the occurrences of a pattern P in a text. It requires the Burrows-Wheeler-Transformation L and the two structures C and Occ (see section 4.1, 4.4).

What is the main idea of this algorithm?
Remember that the BWT matrix rows are lexicographically sorted. Thus, all rows with prefix P occur in a contiguous interval and it is possible to calculate the first and last position of this interval. The first position can be denoted as *First* and the last as *Last*. The *number of occurrences* of P is $Last - First + 1$. The backward search running time is O(P) because the algorithm iterates over each character of P and determines the new interval of rows. For this only Occ and C are necessary which can be accessed in $O(1)$ time. The algorithm is shown in Figure 6.

5.2 Algorithm to locate P in T

After performing the backward search algorithm, First and Last were determined. In the matrix M each row is a cyclic permutation of the text T and each row M_i with $i = \underline{First}, \ldots, \underline{Last}$ is prefixed by P and thus represents an occurrence of P in T. The next step is to determine for each row M_i the position of the first row character in the text, which we denote as $pos_T(i)$.

The backward search algorithm works in P iterations for $i = \mathsf{P}, \ldots, 1$.
First and *Last* are initialized with the positions of the first and last occurrences of $P[\mathsf{P}]$ in F.

Before executing step 2, *First* and *Last* is the interval of rows beginning with $P[i, \mathsf{P}]$. Steps 4 and 5 determine the suffixes in this interval that are preceded by $P[i-1]$.

1) $i \leftarrow \mathsf{P}$;
   $First \leftarrow C[P[i]] + 1$;
   $Last \leftarrow C[P[i] + 1]$;

2) *While* $i > 1$ *and* $\underline{First} \leq \underline{Last}$ *do 3-5*

3) $i \leftarrow i - 1$;
   $c \leftarrow P[i]$;
   Proceed with the next character.

4) *Find and L-to-F map the first occurrence of c in L[First, Last]*
   First $\leftarrow C[c] + Occ(c, First - 1) + 1$;

5) *Find and L-to-F map the last occurrence of c in L[First, Last]*
   Last $\leftarrow C[c] + Occ(c, Last)$;

Finally the number of occurrences is $Last - First + 1$.

Text: `mississippi`
Pattern: `si`

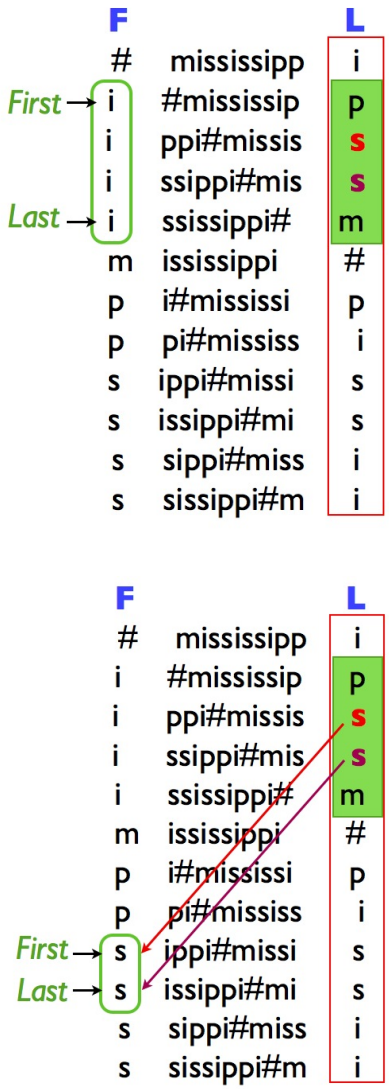


Figure 6: The backward search algorithm

For a row i with $M_i = T\#$ obviously holds $pos_T(i) = 1$. If for a row i with $M_i \neq T\#$ the cyclic rotation to the right by one character is $M_j = M_i^r$ then $pos_T(i) = pos_T(j) + 1$ holds.

Using these observations it is possible to reduce the space consumption by storing the positions $pos_T(i)$ only of certains rows i and determine the positions of the remaining rows indirectly. The following section describes the

algorithm in detail.

5.2.1 Compression of pos_T

The main idea of compressing pos_T is the following: If a row i is given then either $L[i] = \#$ and thus $pos_T(i) = 1$ or $pos_T(i) = pos_T(j) + 1$ for j the result of the L-to-F mapping of i. As described before $pos_T(i)$ of certain rows i needs to be known. Therefore in a preprocessing the position of every x-th character of T (x is a fixed distance) is stored in a data structure S.

Example:
$pos_T(1) = 12$
$pos_T(3) = 8$
$pos_T(6) = 1$
$pos_T(10) = 4$

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
First →	s	ippi#missi	s	9
Last →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Figure 7: Preprocessing. The position of every x-th letter in the text is marked and stored for the corresponding row in S.

After the preprocessing, the position in T is determined for each i = [First, Last]:

1) $j \leftarrow i$;
   $t \leftarrow 0$;

2) **while** row i is not marked **do**

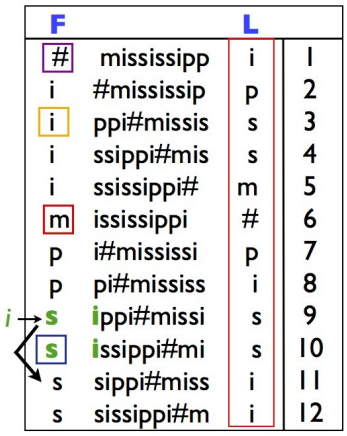
   $j \leftarrow C[L[j]] + Occ(L[j], j)$;
   $t \leftarrow t + 1$;

3) **return** $pos_T(i) = pos_T(j) + t$

Example. In the following the shown example (see above) will be continued. We have a BWT of the text `mississippi#`, the data structure S and First and Last of the pattern `si`. For each $i = [9, 10]$ we have to map the character $L[i]$ to its position in the text.

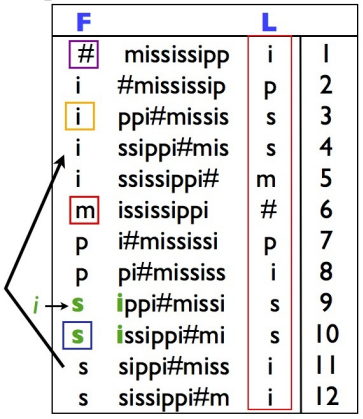
i = 9

Step 1



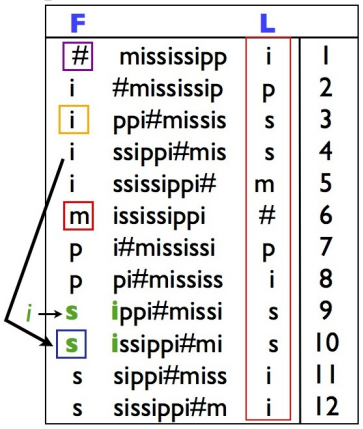
row 9 is not marked
→ L-to-F(9)=11
→ Look at row 11
t=1

Step 2



row 11 is not marked
→ L-to-F(11)=4
→ Look at row 4
t=2

Step 3



row 4 is not marked
→ L-to-F(4)=10
→ Look at row 10
t=3

row 10 is marked
Calculation of the $pos_T(9)$:
$pos_T(9) = pos_T(10) + 3 = 4 + 3 = \mathbf{7}$

i=10
The row 10 is marked and $pos_T(10)$ can be found in S: $pos_T(10) = \mathbf{4}$.
Thus, the pattern occurs at positions 7 and 4 in the text.

5.3 Space consumption of EXACTMATCH

As mentioned before EXACTMATCH needs L, C, Occ and pos_T for the algorithm. For a text of length n over the alphabet Σ, the length of L, C, Occ and pos_T is n, $\mid \Sigma \mid$, $n \cdot \mid \Sigma \mid$ and $\frac{n}{x}$.
The Burrows-Wheeler-Transformation L of the given text is just a string of characters. If the text is for example the human DNA, the memory footprint will be 3 billion $\times$ 1 byte (size of a character) without compression. However, there are also algorithms to compress L.
The C-table contains for each letter of Σ the total number of characters in T which are lexicographically smaller. So C contains $\mid \Sigma \mid$ items. Each element is a number less than n. For the example the alphabet contains the four letters *A, C, G, T* and the maximal size can be 3 billion. That means the memory footprint will be less than 4 $\times$ 4 bytes (size of the number).
The Occ mapping contains the number of occurrences for each character and each possible prefix of L. As described before (see 4.4) without compression Occ has $\mid \Sigma \mid \times n$ entries storing the number of occurrences. For the human DNA a space of 4 $\times$ 3 billion $\times$ 4bytes will be needed. One way to reduce the size of Occ is to store only every x-th index. The entries that are omitted can be reconstructed from stored entries at the cost of an increased running time.

6 Backtracking

6.1 How are mismatches handled?

Simply searching for exact matches is not sufficient. We have to be able to look for reads with indels or that match with at least one or two mismatches. This mismatches can occur due to sequencing errors or differences between the genomes of reference and query organisms.

The Bowtie algorithm now combines the ability of the EXACTMATCH algorithm to compute the complete set of all exact matches by moving along the pattern character by character with a simple error enumeration and backtracking.

This backtracking algorithm performs a depth-first search through the space of possible alignments. It does so by first exploring variants with no or few errors and stops as soon as a valid alignemnt is found. If a valid alignment exists, the algorithm will find it, but because the backtracking enumerates all possible alignments, the first valid alignment encountered by the algorithm will not necessarily be the 'best' in terms of number of mismatches or in terms of quality. It is also important to notice that the more mismatches we allow, the higher the runtime of the backtracking will be if the match contains many of the allowed errors.

The Backtracking algorithm works as follows: Go through the pattern P, from back to front and use this character to make a step in the EXACTMATCH algorithm. If the set of solutions (the interval in EXACTMATCH) becomes empty we change the pattern at the current position (to an arbitrary other character or gap) and increment the counter variable for the number of mismatches. If the counter exceeds the maximum number of mismatches, we have to backtrack (go back one position in the pattern and in EXACTMATCH) and try another character. If all posibilites are exhausted without finding a valid match, we also have to backtrack. While backtracking we have of course adapt the error counte accordingly.

6.1.1 Pseudocode

- **Input:**

  Text T

  Pattern P

  maximal number of mismatches (n)

- **Output:**

  Position of one match with at most n mismatches

For the pseudocode we define a function `int em(char c, int last, int first)` which return the number of matches for the current suffix of the pattern as maintained by EXACTMATCH and adapts first and last accordingly. For clarity reasons, this pseudocode works only for hamming distance.

Initialization:

first $\leftarrow$ 1, last $\leftarrow \mathsf{T} + 1$, $i \leftarrow \mathsf{P}$, $k \leftarrow 0$, k is the number of mismatches accumulated during the search

If backtrack(P,k,i,first,last) **Then** output: found last-first+1 occurrences of P;

bool backtrack(P,k,i,first,last)

If k $>$ n return false;

If last-first+1 $>$ 0 and i=0 **Then** return true;

If last-first+1 $<=$ 0 and i=0 **Then** return false;

tried characters $=\{P[i]\}$;

If em(P[i], first, last) $>$ 0 **Then**

If backtrack(P,k,i-1, first,last) **Then** return true;

While $\exists$ a character $c \in \Sigma \backslash$ tried characters

P[i]=c;

tried characters= tried characters $\cup\{c\}$

If em(P[i], first, last) > 0 **Then**

If backtrack(P,k+1,i-1, first,last) **Then** return true;

return false;

DIE ABBILDUNG IST FALSCH

6.1.2 Example: GGTA

In this small example we try to match $GGTA$ to the genome which does contain neither $GGTA$ nor GTA, but $GATA$, allowing one mismatch. We start with the first call of backtrack. EXACTMATCH search starts at the end and goes to the front of the read. So the first nucleotide we trie to match is A. The EXACTMATCH would give us in the em function of the pseudocode a positive count, so we call backtrack with a smaller pattern index and the same error count. Then T is matched and again backtrack is recursively called. Now we try to match G. The em function reports an empty range and hence we try other characters at this position. For each possible choice we change the pattern and check, whether we can now match by calling backtrack with an error count incremented by one and a smaller index. If this call finds a match we return true. If not, all choices of characters could not find a match and we return false. In our case we might try A instead of G, and call backtrack with $k = 1$. Now em returns a positive range (because the genome contains $GATA$) and we recurse again matching G. Then we call backtrack for a last time with $i = 0$ and return true, because the range is positive.

6.2 Excessive backtracking

Excessive backtracking occurs when there is already a mismatch in the begining of a read and it's likely to backtrack much more often then necessary.

Bowtie solves the problem of excessive backtracking with a technique of 'double indexing'. Two indices of the genome are created: one containing the BWT of the genome, called the 'forward' index, and a second containing the BWT of the genome with its character sequence reversed (not reverse complemented) called the 'mirror' index.

7 Bowtie

Bowtie [5] is a fast and memory-efficient read mapping tool. The goal of mapping tools is to map reads back to a reference genome to calculate read densities and their positions.

obacht: Read mapping should be introduced in **??**. Here you can refer to that. (Reinert)

Bowtie generates an index of the reference genome which is based on the Burrows-Wheeler transformation (BWT) (see section 4). The constructed index uses the Ferragina and Manzini exact matching algorithm [4] (see section 5) to search through the index.

Furthermore, Bowtie provides a backtracking search method to find inexact matches through the reference genome. In other words, Bowtie combines the EXACTMATCH algorithm with a backtracking algorithm 6 that permits mismatches and indels. If a valid alignment exists, then Bowtie will find it, but because the search is greedy, the first valid alignment encountered by Bowtie will not necessarily be the 'best' in terms of number of mismatches or in terms of quality.

8 Summary

In this section I will emphasize what techniques and arguments are central to this part of the lecture in form of questions to which you should be able to answer.

- RNA-Seq
  - What is the main goal of RNA-Seq? What biological questions can be answered.
  - How does a typical RNA-Seq pipeline proceed. Can you describe each step and the involved algorithmic problems?
- BWT
  - How ist the BWT defined and how can you compute it efficiently for a given string?
  - How can you compute the L to F mapping? (what auxiliary tables are used?)
  - How can you compute those tables?

  - How can you use the BWT and the L-to-F mapping for searching a pattern in a string exactly?
  - Why is the L-to-F mapping correct? (rank preserving property)
  - How can you save space at the expense of run time for the L-to-F mapping?
  - How do you obtain the positions of the matches? What space do you need?
  - How can you save space at the expense of run time during the computation of the positions?

- Back tracking
  - How does backtracking work if we can employ an exact matching algorithm of suffixes (or prefixes) of the patterns that works character by character.
  - How can you conceptually do backtracking with indels?

9 Filtering

Filtering algorithms are based on the idea that it is faster to find the positions of a non-matching string within a text, than to find the position of a match. These algorithms discard all parts of a text, which do not contain possible matching positions of a pattern string and at the same time find positions which might be match. These possible matching positions need to be verified by an additional string matching algorithm.

9.1 Disadvantages and advantages

Filtering algorithms are very sensitive to the error level $\alpha := \frac{k}{m}$ (k = error, m = pattern length). This affects the amount of a text that can be discarded from further consideration. With higher error levels the costs for verifications start to dominate and reduce the filter efficiency abruptly. In other words, if we have to verify most of the text, filtering algorithms are not worth applying.

If we can discard large segments of the text, filtering method will result in a fast search.

9.2 PEX

The PEX algorithm is one of the basic examples for filtering. This method is based on the pigeonhole principle.

9.2.1 Pigenhole

This principle says, we have m objects (pigeons) and n sets (holes) where $m < n$. If we place each object in one set there will be one set left empty. The goal within read-mapping algorithms is to find all approximated occurrences of a pattern (P) with length $\mathsf{P} = m$ in a text (T) with length $\mathsf{T} = n$. The PEX algorithm reduce the approximated search with most k errors to an exact searching with $k + 1$ sub-patterns. In this case the pigeons are the $k + 1$ pattern pieces and the holes are the k errors. The goal is to distribute the errors over the pattern pieces that one sub-pattern has to match without error. A more general formalisation was first formalised by Myers in 1994 which is the basis of the PEX algorithm.

Lemma 9.1. *Let Occ match P with k errors, $P = p^1, \ldots, p^j$ be a concatenation of subpatterns, and $a_1, \ldots, a_j$ be* ***non-negative*** *integers such that $A = \sum_{i=1}^{j} a_i$. Then, for some $i \in \{1, \ldots, j\}$, Occ includes a substring that matches p^i with at most $\lfloor \frac{a_i k}{A} \rfloor$ errors.*

Figure 8: We have 8 pigeons and try to place them in 9 holes. The result is, that one hole will be empty.

Proof:

Let k_i be the number of errors in p^i. Then the following holds:

$$k = \sum_{i_1}^{j} k_i \tag{1}$$

Following the Lemma there exists at least one i with $\lfloor \frac{a_i k}{A} \rfloor \geq k_i$.

We proof the Lemma by contradiction. Therefore assume that there is no such i and it holds $\lfloor \frac{a_i k}{A} \rfloor < k_i$; $\forall k_i$.

Then the following chain of inequalities ist true for each k_i:

$$k_i \geq \lfloor \frac{a_i k}{A} \rfloor + 1 > \frac{a_i k}{A}.$$

Now it is easy to derive a cotradiction to our assumption, beacuse the following holds

$$k = \sum_{i=1}^{j} k_i \geq \sum_{i=1}^{j} (\lfloor \frac{a_i k}{A} \rfloor + 1) > \sum_{i=1}^{j} \frac{a_i k}{A} = k$$

$$k > k \lightning$$

Hence or assumption must have been wrong and as consequence the Lemma is correct.

9.2.2 Basic procedure

The goal is to find an approximate occurrence of a pattern P of length m in a text T of length n with at most k errors. The first step within the

basic procedure is to divide the pattern string into k+1 pieces such that each pattern piece has the same probability to occur in the text. If not enough information is available a uniform distribution is assumed and the pattern will be divided into pieces with approximately the same length. The goal of this is that one piece of the pattern has to match without error. The second step to search all pieces simultaneously with a multi-pattern string matching algorithm, like Aho-Corasick or Wu-Manber. Since we have $k+1$ pieces and allow at most k errors, each possible occurrence of the pattern will match at least one of the pattern pieces exactly. The last step is the verification step.

The first two steps are intuitively clear. In the section below, the verification step will be described more detailed.

9.2.3 Verification

For the verification we compare the entire pattern with the appropriate text region. Assume the pattern piece $p^i = P[start_i, end_i]$ ($start_i$ and end_i are the start and end position of the i-th pattern piece), matches the text $T[j : j+(end_i-start_i)]$. The complete pattern match has a length of $m+k$ and can begin at most $start_i-1+k$ positions before position j in T. Also it finishes at most $m-end_i+k$ positions after $j+(end_i-start_i)$ in T. The reason is that the k errors can occur on the left or right of the current text match. Hence, we have to check the text area $T[j-(start_i-1)-k : j+(m-start_i)+k]$ of length $m+2k$.

figure wird noch erstellt!!

Example

pattern: annual

text (T): any_annealing

error (k): 2

Dividing: annual $\Rightarrow p^1 = an$, $p^2 = nu$, $p^3 = al$

Searching: **an** in $t \Rightarrow$ pos $1, 5$ ($T =$ **an**y_**an**nealing)

nu in $t \Rightarrow$ pos $None$

al in $t \Rightarrow$ pos 9 ($T =$ any_anne**al**ing)

Verification: three occurrences in $t \Rightarrow 9, 10, 11$

$t[9]$	$t[10]$	$t[11]$
annea-	anneal	anneali
annual	annual	annual-

Note that in the above example the number of verifications can be reduced by one since the matches of *an* at position 5 and of *al* at position 9 are triggering the same verification.

9.2.4 Problems

If the pattern does not match in a text, this will result in many verifications which are unsuccessful. For example, searching for the pattern *annual* in

$$an\_unusual\_example\_with\_numerous\_verification$$

where the pattern does not match. Furthermore, if the text is a repeated region or contains one, repeated verifications will result. Problems like this will increase the costs of verification and reduce the filter efficiency. To solve this problem, we can use an alternative approach. **Instead to verify the entire pattern piece by piece, we can verify a short sub-pattern which is less expensive.** This is the idea of hierarchical verification.

9.2.5 Hierarchical verification

Instead to divide the pattern directly in $k+1$ pieces, we do it hierarchically. First, the pattern is divided in two pieces and searched for each piece with $k = \lfloor \frac{k}{2} \rfloor$, following the Lemma 9.1. Furthermore, the halves are recursively split until the error rate reaches zero. This has to be the case if we have $k + 1$ leafs (Figure: 9.2.4). If we built the tree, we can verify the pattern in a less expensive way, compared with the verification introduced in 9.2.3.

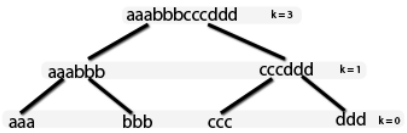


Figure 9: Balanced tree with at most k = 3 errors. The childs of each node have to have at most $k = \lfloor \frac{k}{2} \rfloor$ errors.

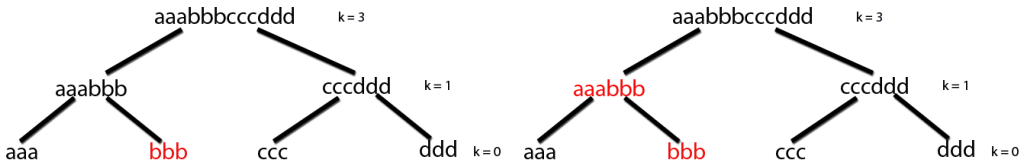


Figure 10: verification with balanced tree
1. find exact match of leafs within the text. 2. go from leafs to the rood so long as the occurrence full fill the most accepted error in the node, else reject the position in the text.

Table 1: Internal variables if $k+1$ not power of two

variable	description	calculation
$left$	number of pattern pieces in the left subtree	$\lceil \frac{(k+1)}{(2)} \rceil$
lk	error of current node in the left subtree we look at	$\lfloor \frac{(left \cdot k)}{(k+1)} \rfloor$
$right$	number of pattern pieces in the right subtree	$k+1-left$
rk	error of current node in the right subtree we look at	$\lfloor \frac{(right \cdot k)}{(k+1)} \rfloor$

We have verify the pattern *aaabbbcccddd* in the text *xxxbbbxxxxxx*, with the associated balanced verification tree (9.2.4). In a first step we search for one leaf that returns an exact match within the text. In this case this is the pattern piece *bbb*. Instead to of verifying the tree root (entire pattern), we verify a smaller sub-pattern. This sub-pattern is the parent of the pattern piece we have found *aaabbb*. In this level of the tree only one error is allowed, but the best match contains three errors. Hence there is no possibility to extent the sub-pattern with at most one error and we can reject the entire pattern.

What do we do, if the number of the $k+1$ pattern pieces are not a power of two? In this case we try to build the tree as balanced as possible. Therefore we introduce new internal variables shown in Table: 1.

example for hierarchical search with a not balanced tree If we want to search the pattern of section 9.2.3 with this approach, we have to build the tree as balanced as possible like in figure 9.2.5.

CreateTree($p = p_i p_{i+1} \dots p_j,\ k,\ myParent,\ idx,\ plen$)
// Note: the initial call is: CreateTree ($p,\ k,\ nil,\ 0,\ \lfloor m/(k+1) \rfloor$)
Create new node $node$
$from(node) = i$
$to(node) = j$
$left = \lceil (k+1)/2 \rceil$
$parent(node) = myParent$
$err(node) = k$
if $k = 0$
  then $leaf_{idx} = node$
  else
    $lk = \lfloor (left \cdot k)/(k+1) \rfloor$
    CreateTree($p_i \dots p_{i+left \cdot plen-1},\ lk,\ node,\ idx,\ plen$)
    $rk = \lfloor ((k+1-left) \cdot k)/(k+1) \rfloor$
    CreateTree($p_{i+left \cdot plen} \dots p_j,\ rk,\ node,\ idx + left,\ plen$)
fi

The code depicted constructs the tree top down and and guarantees, that the during the execution of the algorithm Lemma 9.1 holds if we choose the factor of the lemma as computed in the pseudocode (lk and rk).

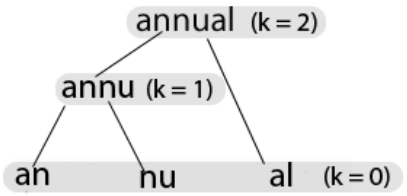


Figure 11: Tree as balanced as possible
with at most k = 2 errors. The children of each node have to have at most k errors as described in Table 1.

The search step is now an very intuitive step. First we search for exact-matches in the leafs and look in the parent of the leaf if we can extend the match to a sub-pattern with at most k errors. We do this until we reach the end of the text and reject each possible occurrence if the actual condition of the most accepted error in the node is not full fill. An example is given in Table 2.

$$pattern = annual$$

$$text = any\_annealing$$

$$k = 2$$

Table 2: searching with an tree 9.2.5 balanced as possible

1. Found	**an**_yannealing
Search for	"annu" with $k = 1$
inside	**any_a**nnealing
failed	abort the verification and reject the position of the text
2. Found	an_y **an**nealing
Search for	"annu" with $k = 1$
inside	any_**annea**ling
found	condition full fill go upper in the tree
Search for	"annual" with $k = 2$
inside	**any_annealin**g
found	report position 9,10,11 like in 9.2.3
3. Found	an_yanne**al**ing
Search for	"annual" with $k = 2$
inside	**any_annealin**g
found	end of text abort search an report results (position 9,10,11)

9.3 Q-Gram-Counting based approaches

9.3.1 Basics

Given a sequence S, a q-gram is any sub-sequence of length q (also referred to as k-mer, where k is the length).

For two sequences S_1, S_2 of equal length the set of common q-grams is the set of overlapping sub-sequences of length q.

Lemma 9.2 (Q-Gram-Lemma)**.** *Two sequences S_1, S_2 of length l and edit distance $\leq k$ share at least $t = l - kq - q + 1$ common q-grams.*

The q-gram-lemma is based on these two observations:

1) one sequence of length l contains $l - q + 1$ overlapping q-grams

2) every mismatch can destroy at most q q-grams
   $\rightarrow k$ mismatches destroy at most kq q-grams

9.3.2 QUASAR

QUASAR stands for "**Q[u]**-gram **A**lignment based on **S**uffix **AR**rays" (TODO cite). It is a tool that computes approximate local matches based on q-gram counting and the q-gram-lemma. An approximate local match is defined as a region of a certain minimal length on two sequences with edit distance $\leq k$.

Important: Since we deal with Read-Mapping we only consider semi-global alignments, not local alignments. The theory behind QUASAR explained in the following is therefore reduced to the special case of semi-global alignments.

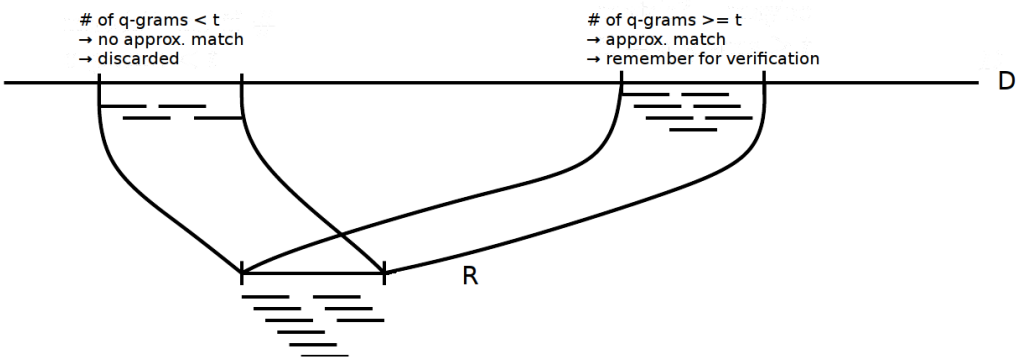


Figure 12: Q-gram counting (D = Database, R = Read)

Overview of the QUASAR pipeline:

1) Index: create a q-gram-index pointing into a suffix array for constant time lookup of all q-frams of the D.

2) Blocks: divide genome into non-overlapping blocks (buckets) and define a counter per block

3) Counting: lookup all q-grams of R in the index, retrieve the positions from the suffix array and increment the corresponding counters

4) Threshold: all blocks with counter $\geq t$ are remembered for verification

The last two steps are repeated for every read. It is remembered for which read(s) a block's threshold is exceeded (a block is only verified for those).

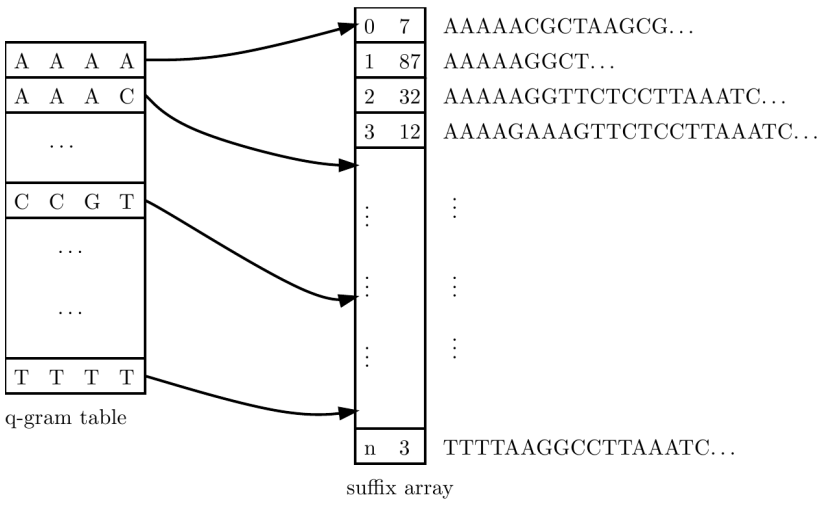
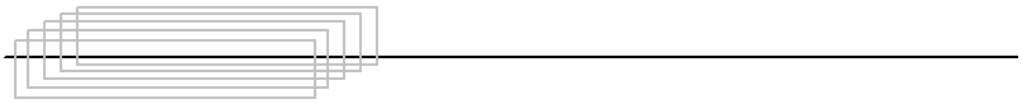


Figure 13: Q-gram index

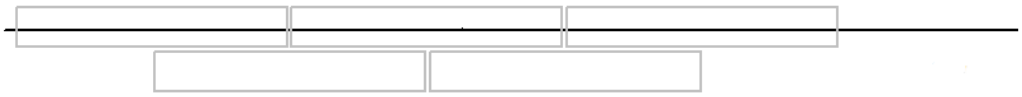
1) Index For constant-time lookups a q-gram-index is constructed. The table of all q-grams in all reads is created which needs space $O(\mid \Sigma \mid^q)$ and the position of all exact matches for all q-grams is recorded. This can be achieved by pointing to the corresponding position in a suffix array. as shown in fig.13.



2) Blocking A trivial approach to counting would be to look at all (overlapping) sub-strings of length of the read ($+k$ in case we allow gaps) in D and count the amount of matching q-grams. This would however require

$$\mathsf{D} - \mathsf{R} + 1 \; \textit{counters}$$

$\rightarrow$ very memory expensive.



Alternatively you can devide the genome into non-overlapping buckets, called blocks and keep one counter for each. To not miss out on the approx. matches spanning a block border we introduce a second row of shifted blocks and increase the length to $b \geq 2 * (\mathsf{R} + k)$. This results in $\frac{\mid D \mid}{b}$ counters.

You should always keep in mind, though, that a larger block results in worse specificity!

3) Counting lookup all q-grams in R, retrieve the positions from the suffix array, and increment corresponding counters.

4) Threshold all blocks with counter $\geq t$ are remembered for verification

9.3.3 Gapped Q-Grams

Motivation Based on the previously discussed theory it should be evident that a larger q results in less random hits and therefore a higher filtration rate (which is desirable). However it also results in a lower threshold which counteracts this effect.

A possible improvement is the use of gapped q-grams, which provide a higher filtration rate while maintaining high thresholds.

Definition While the formal definition of gapped q-grams / shapes is more extensive, the following terms should be understood:

		Example: `##.#..#`
shape Q	set of $\mathbb{N}_0$, positions of #	$\{0, 1, 3, 6\}$
size of Q: Q	"number" of `#`	4
span of Q: $s(Q)$	"number" of `#,.`	7

Shapes consist of positions that are verified ("#") and positions that are not verified, the gaps("."). Note that this does not imply a mismatch at the ".s", it just allows for a mismatch at this position (without rejecting the q-gram-match).

Thresholds The q-gram lemma can be generalised to gapped q-grams:

$$t = w - s(Q) - \mathsf{Q}k + 1$$

However it is not tight anymore, i.e. you can still expect to see at least t q-grams, but in many situations you can provably expect more.

Consider these two shapes and compute their thresholds:

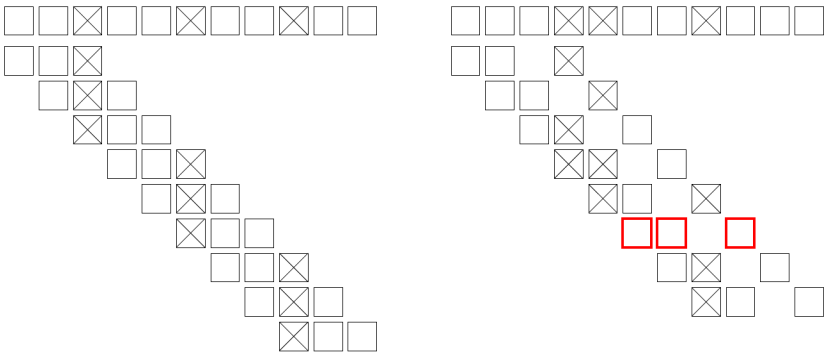
shape	`###`	`##.#`
$t =$	$11 - 3 - 3 * 3 + 1 = 0$	$11 - 4 - 3 * 3 + 1 = -1$

$w = 11$
$k = 3$

Obviously both shapes seem useless ($t \leq 0$).

Now let us experimentally determine the worst possible placement of errors:

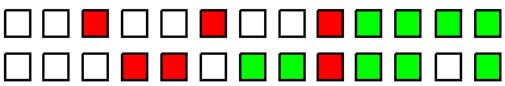
shape	`###`	`##.#`	$w = 11$ $k = 3$



As we can see $t = 1$ for `##.#`! This shows that the q-gram-lemma is not tight anymore, as well as showing that gapped shapes are often superior to ungapped shapes of equal size.

Minumum Coverage There is another intricacy when using gapped q-grams. Consider the shapes `###` and `##.#` for $w = 13$ and $k = 3$.

In both cases $t = 2$, but for `###` four consecutive matches suffice to reach this treshold, while `##.#` requires five consecutive matches. The number of characters that have to match for a shape q to reach a threshold of t is called the minimum coverage of q at threshold t.



Obviously a higher minimum coverage increases the filter specificity.

Summary

- Gapped Q-grams improve the filter efficiency by magnitudes

- placement of gaps in the Q-gram influences threshold and minimum coverage
- threshold and minimum coverage both influence filter efficiency
- there is no closed formula known for computing the threshold of gapped Q-grams (the Q-Gram Lemma is only a lower bound)

9.3.4 Verification

Verification, in approaches that use q-gram-counting, is completely independent of the filtering step. It can be performed after all candidate regions have been identified (QUASAR), or "on-the-fly" (RazerS).

The algorithms employed range from traditional heuristic approaches, like BLAST (QUASAR) to specialised DP-based algorithms like Myers Bitvector algorithm (RazerS). When using hamming distance, scoring is trivial (count mismatches along the diagonal).

Classical DP-Approaches

- Needleman-Wunsch for global alignments, Smith-Waterman for local alignments.
- semi-global by setting only first row (not first column) to 0
- both use $O(nm)$ space and run-time.
- by only remembering the last column and doing a backtrace later, we can reduce space-requirement to $O(m)$ (Hirschberg)

Ukkonen's algorithm

- observation that each cell's value differs $\{-1, 0, +1\}$ from its neighbours'
- based on that you can quickly realise when a value will never become "good" again in a column (once it has reached $k + 1$) and stop there
- this results in something similar to a banded alignment
- run-time O(km)

Myers Bitvector algorithm

- do not save absolute values in the DP, but the differences to above cell ($\in \{-1, 0, +1\}$)
- columns then encoded as bit-vectors
- dependencies/relation of cells are encoded as bit-operations (AND, OR, OR NOT)
- columns are computed by bit-shifting similar to Shift-Or algorithm
- depending on read length a complete column maybe calculated simultaneously

$\Delta v_{i,j}$:

		A	N	N	E	A	L	I	N	G
	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	1	1	1	1
N	1	1	-1	0	1	1	0	1	1	1
N	1	1	1	-1	-1	1	1	0	0	0
U	1	1	1	1	0	0	1	1	1	1
A	1	1	1	1	1	-1	-1	0	1	1
L	1	1	1	1	1	1	-1	-1	-1	0

Figure 14: V-Table of Myers Bitvector DP

9.4 Summary

In this section I will emphasize what techniques and arguments are central to this part of the lecture in form of questions to which you should be able to answer.

- Pidgeonhole principle
  - Can you explain the general version of the pidgeonhole principle?
  - Can you argue why choice of the k in each level of the PEX algorithm is correct?
  - Can you explain why a balanced tree is better than an unbalanced tree in the hierarchical verification?

  - In the hierarchical verification an exact match triggers a verification in the level above. Can different exact matches trigger the same verification? How can this be avoided?

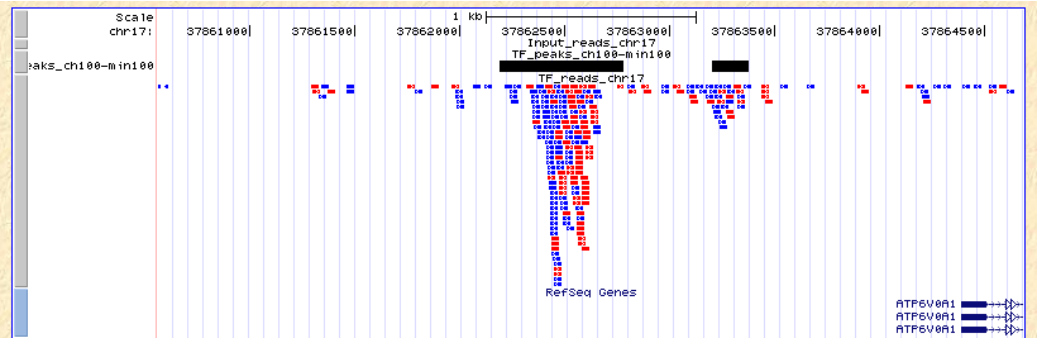
- q-gram counting
  - Describe the data structures used in the Quasar q-gram counting approach.
  - Is the use of a suffix array really necessary? What else could be done?
  - Be able to prove the q-gram lemma.

- gapped q-grams
  - Why is the q-gram lemma for not tight any more for gapped shapes?
  - Given two shapes with the same threshold for some given parameters. What other criteria can you use to prefer one to the other?

10 Locating

In RNA-Seq as well as in DNA-Seq reads can potentially map to several locations, these reads are called multi-reads. The question if obviously, where in the genome they really stem from. A possible strategy to resolve this question is to map multi-reads to all possible locations. If similar reads from different locations differ slightly, then the idea is to cluster reads from the same regions. The clustering can be done by grouping reads that share common differences to other groups of reads.

Assume now that we have mapped every read to all possible locations. Obviously this gives us also the information where multiple reads map (a region to which at least two reads map that map also to another location).



11 Bounding

Now we want to compute a multiple alignment of the reads in repeat regions to analyze and separate them using correlated differences. To compute a multiple alignment we need well defined boundaries of the fragments. All these fragments also have to have the same length, since the separation procedure described in section 13 uses hamming distance between fragments that are computed over the rows of the alignment, and these distances will be biased if fragments with different lengths are considered.

Hence, choosing a wide window will reduce the number of fragments it can contain. Choosing a deep window containing many fragments will be narrow and is unlikely to have enough distinguishing base sites, while a wide window containing many such sites will be shallow and not allow us to separate many fragments.

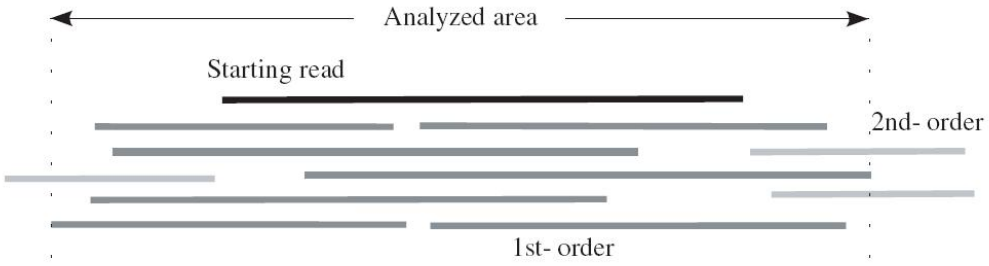


Figure 15: Schematic view of analyzed overlaps in a multi-alignment.

11.1 Finding an optimal window

We consider as an optimal window a representative window that is feasible and has maximum depth. A representative window is a widest window of a given depth, where the width of a window is the length of the intersection of the layout intervals of its fragment.

Let d be the depth of the layout at the location where we want to compute an optimal window. To find an optimal window, we will compute a representative window of depth d, $d-1$, $d-2, \ldots$ until we find one that is feasible.

One way of doing this (ref tammi) (see fig. **??**) is to chose one read out of a repeat region and use it as a starting read to elongate from. All reads aligning to this starting read (1st-order) will form a contig and will be the area analyzed in section 12 and 13. The reads then aligning again with the 1st-order reads (2nd-order) will be used as new starting reads. And the process is repeated until all reads in the data set have been analyzed.

12 Identifying DNPs

After computing the multi-alignment of read sequences the next step is to find all the positions of alignment columns, which contain mismatches and therefore are useful to distinguish between repeat copies in the following separating step. In this chapter the identification of the so called 'defined nucleotide positions' (DNPs) with the algorithm by Tammi et. al. **QUOTE!** will be described. Although Tammi et. al. developed their algorithm for the assembly problem of whole genome shotgun sequencing data, this algorithm can be also applied to RNA sequencing.

12.1 Idea of Tammi's algorithm

First the algorithm looks for all alignment columns, which contain $\geq D_{min}$ non-consensus bases (deviations) and therefore are potential DNPs. In whole genome shotgun sequencing the most frequent base in an alignment column is assumed as the consensus base (in this chapter columns denote alignment columns). Since the results have shown that only one DNP is not enough to separate the repeat copies, a second, support column is needed (see Fig. 16).

```
...AGCCGTCAGA...
...AGCCGTCAGA...
...AGCCCTCTGA...
...TGTCGTCTGA...
...AGTCGTCTCA...
...AGTCGTCTGA...
```

Figure 16: Example of two DNPs (blue columns) with $D_{min} = 2$ in an multi-alignment. The first blue column has 3 deviating bases of type C. The second blue column is the supporting column with 2 deviating bases of type A.

In the basic method the algorithm searches for all DNP pairs with $\geq D_{min}$ deviations which coincide in the same rows of the multi-alignment. The

extended method computes the probability of those observed coincidences and checks, whether the deviations occurred by chance or not.

12.1.1 Mathematical definitions

Let u, v be two fixed positions in the multi-alignment M, then all the definitions for u also hold for v. The base at column u in jth sequence is $a_{u,j}$. If this base $a_{u,j}$ deviates from consensus, then the indicator variable $I_{u,j} = 1$ and $I_{u,j} = 0$ otherwise. The probability for a deviation (error rate) at u in j is $p_{u,j} = P(I_{u,j} = 1)$. The total number of deviations at u is determined by $N_u = \sum_{j=1}^{k} I_{u,j}$, where k is the number of rows in M. I_j indicates the coincidences in j: $I_j = I_{u,j} I_{v,j}$. $I_j = 1$ if there are deviations at both positions in j and $I_j = 0$ otherwise. Finally the total number of coincidences is defined as $C = \sum_{j=1}^{k} I_j = \sum_{j=1}^{k} I_{u,j} I_{v,j}$.
As we can see no formula contains $p_{u,j}$ yet, that is to say the quality values of the base calling is neglected. Therefore there are two approaches how to handle the error rates. The first assumes an independent and identically distributed error rate, whereas the second includes non-iid error rates.

12.2 Approach with iid error rates

Assume the deviations are iid, which is not always true, then it holds $p_{u,1} = ... = p_{u,k}, p_{v,1} = ... = p_{v,k}$.
Because of this, we can use the hypergeometric distribution that determines the probability for a given number of successes in n draws from a finite population without replacement.

In a toy example this means we have in a bucket in total N balls consisting of M white balls and the rest $N - M$ balls are black. Now we draw n balls without returning any balls out of the bucket, then the probability for drawing x white balls out of the n draws is:

$$P_x(N, M, n) = \frac{\binom{M}{x}\binom{N-M}{n-x}}{\binom{N}{n}},$$

where $\binom{M}{x}$ is the number of combinations to draw x white balls out of maximal M white balls and $\binom{N-M}{n-x}$ the number of combinations to draw $n - x$ black balls out of maximal $N - M$ black balls. Moreover $\binom{N}{n}$ is in the denominator, because all the combinations of drawing with returning are left out.

Analogously we derive the distribution C given $N_u = n_u, N_v = n_v$ as:

$$P(C = x) = \frac{\binom{n_v}{x}\binom{k-n_v}{n_u-x}}{\binom{k}{n_u}},$$

$$0 \leq x \leq n_v, 0 \leq n_u - x \leq k - n_v$$

obacht: grafik ergänzen (Dang)

The k balls represent bases in v where the n_v deviating bases represent white and the $k - n_v$ non-deviating bases represent black balls. For every of the n_u deviating bases in u we draw a ball and want to to determine the probability that there are x coincidences (white balls).

In reality the deviations do not occur iid therefore the non-iid approach is needed.

12.3 Approach with non-iid error rates

With different error rates computing the expectation value of C using the hypergeometric distribution will be complicated, since the formula is split into many different terms, because each deviation has a separate probability ($p_{u,i}$). Yet, assuming the probabilities of the deviations are different, but very small, then we can use the Poisson distribution and our goal is to compute the mean $E(C \mid N_u = n_u, N_v = n_v)$ of it. For calculating the conditional probability the Poisson distribution still approximates well, since the conditioning on N_u and N_v introduces weak dependencies.

From the definition of the number of coincidences $C = \sum_{j=1}^{k} I_{u,j} I_{v,j}$ follows:

$$E(C \mid N_u = n_u, N_v = n_v) = \sum_{j=1}^{k} E(I_{u,j} = 1 \mid N_u = n_u) E(I_{v,j} = 1 \mid N_v = n_v)$$

and from the definition of conditional probability (Bayes' theorem) follows:

$$P(I_{u,j} = 1 \mid N_u = n_u) = \frac{P(I_{u,j}=1, N_u=n_u)}{P(N_u=n_u)} = \frac{P(I_{u,j}=1, N_u^{(j)}=n_u-1)}{P(I_{u,j}=1, N_u^{(j)}=n_u-1)+P(I_{u,j}=0, N_u^{(j)}=n_u)},$$

where $N_u^{(j)} = N_u - I_{u,j}$ is the number of deviations at column u without the sequence j.

Because $I_{u,j}, N_u^{(j)}$ are independent, $N_u, N_u^{(j)}$ are Poisson distributed and $\lambda_u = \sum_{i=1} p_{u,i}$ and $\lambda_u^{(j)} = \lambda_u - p_{u,j}$, respectively this formula holds:

$$\frac{P(I_{u,j}=1, N_u^{(j)}=n_u-1)}{P(I_{u,j}=1, N_u^{(j)}=n_u-1)+P(I_{u,j}=0, N_u^{(j)}=n_u)} \approx \frac{p_{u,j} e^{-\lambda_u^{(j)}} {\lambda_u^{(j)}}^{n_u-1}/(n_u-1)!}{p_{u,j} e^{-\lambda_u^{(j)}} {\lambda_u^{(j)}}^{n_u-1}/(n_u-1)! + (1-p_{u,j}) e^{-\lambda_u^{(j)}} {\lambda_u^{(j)}}^{n_u-1}/n_u!}$$

$$= \frac{n_u p_{u,j} \cancel{e^{-\lambda_u^{(j)}}} \cancel{\lambda_u^{(j)^{n_u-1}}} / n_u!}{n_u p_{u,j} \cancel{e^{-\lambda_u^{(j)}}} \cancel{\lambda_u^{(j)^{n_u-1}}} / n_u! + (1-p_{u,j}) \cancel{e^{-\lambda_u^{(j)}}} \cancel{\lambda_u^{(j)^{n_u}}} / n_u!} = \frac{n_u p_{u,j}}{n_u p_{u,j} + \lambda_u^{(j)}(1-p_{u,j})}$$

The formula for $P(I_{v,j} = 1 \mid N_v = n_v)$ for column v is applied analogously and we get

$$E(C \mid N_u = n_u, N_v = n_v) \approx \sum_{j=1}^{k} \left(\frac{n_u p_{u,j}}{n_u p_{u,j} + \lambda_u^{(j)}(1-p_{u,j})} \cdot \frac{n_v p_{v,j}}{n_v p_{v,j} + \lambda_v^{(j)}(1-p_{v,j})}\right) .$$

12.3.1 Testing of the Poisson distribution

With the Poisson distribution, which approximates the distribution of C given $N_u = n_u, N_v = n_v$ we can test the supposed hypothesis that coincidences occur by chance. Therefore we calculate the probability p^{corr} to observe c_{obs} or more coincidences by chance:

$$p^{corr} = 1 - \sum_{i=0}^{c_{obs}-1} Po(i).$$

$Po(i)$ is the probability for the Poisson variable with the mean $E(C \mid N_u = n_u, N_v = n_v)$. The hypothesis is accepted, if $p^{corr} > p_{max}^{corr}$ a given threshold. The special case for columns with lots of expected sequencing error Tammi et. al. introduce correction strategy not explained here.

13 Separating repeat copies

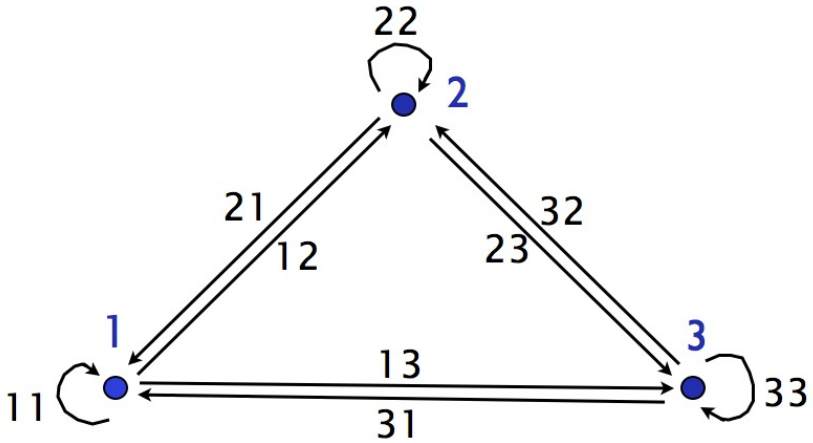
After DNP identifying, the algorithm of Kececioglu[**?**] can be used to separate the reads into groups. Afterwards each group should contain the reads of one repeat copy.
The algorithm assumes that the multi alignment can be split into k copies of the repeat. The idea is the following:
There exists a partition into k classes $P_1, P_2, \ldots, P_k$. For each class a consensus string $S_1, S_2, ..., S_k$ can be defined such that the sum of errors

$$\sum_{1 \leq i \leq k} \sum_{F \in P_i} D(F, S_i)$$

between a S_i and all the other reads of P_i is minimal.

13.1 Constructing a graph theoretical problem

The described problem can be formulated as a graph theoretical problem.
We consider the K_n (the complete directed graph with n nodes) and weight the edges. Nodes represent reads and edges represent overlaps between two reads. Each edge is weighted with the Hamming distance of the two reads and can be labeled with the number of the first and the second read. The following figure shows K_n for 3 reads (the weights are not visualized).
Now we want to find k star centers and an edge set that spans all nodes such that the overall weight of all chosen edges is minimized. This problem is also called the *k-star problem.*



13.2 Formulation as an ILP

The k-star problem can be formulated as an ILP. For a given K_n the following **variables** are needed.

- x_{ij}

  x_{ij} is the variable for the edge between node i and j (the pairs are ordered). This variable encodes if the edge is part of a k-star or not. For n nodes there exists n^2 x-variables.

- y_i

  y_i is the variable for each node i. It encodes if the node is a center of a star. For n nodes there exists n y-variables.

The ILP tries to find a partition into k groups that minimize

$$\sum_{1 \leq i \leq k} \min_{F* \in P_i} \left\{ \sum_{F \in P_i} H(F, F^*) \right\}$$

F^* is the consensus sequence for one class P_i. For all the other reads F of P_i the Hamming distance is computed. In K_n the weight of the edges correspond to the Hamming distances.
The **objective function** of the ILP is formulated as

$$\min \sum_{i \neq j} w_{ij} x_{ij}$$

The ILP has $3n^2 + 3n + 1$ **constraints**:

- $\forall i \; \forall j \quad x_{ij} \geq 0$

  Decides if an edge is part of a k-star or not.
  $$x_{ij} = \begin{cases} 1, & \text{edge is part of a k-star} \\ 0, & \text{otherwise} \end{cases}$$

- $\forall i \qquad y_i \geq 0$

  Decides if a node is a center of a k-star or not.
  $$y_i = \begin{cases} 1, & \text{node is a center of a k-star} \\ 0, & \text{otherwise} \end{cases}$$

- $\forall j, \qquad \sum_{1 \leq i \leq n} x_{ij} \geq 1$

  Each node must have one incoming edge.

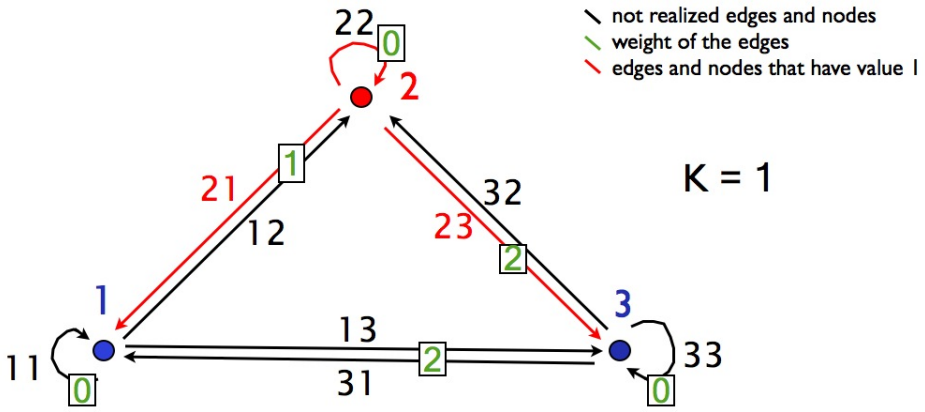
- $\forall i \; \forall j \quad y_i \geq x_{ij}$

  Only the star centers have outgoing edges.

- $\sum_{1 \leq i \leq n} y_i \leq k$

  There are at most k star centers.

13.2.1 Example



The figure shows an example for a K_n. The red edges show one possible solution. The formulated ILP is as follows:
Objective function:

$$\min \; x_{12} + 2x_{13} + x_{21} + 2x_{23} + 2x_{31} + 2x_{32}$$

Constraints:

$\forall i\, \forall j,$	$x_{ij} \geq 0$	$x_{11} = 0, x_{12} = 0, x_{13} = 0, x_{21} = 1, x_{22} = 1, x_{23} = 1, x_{31} = 0, x_{32} = 0, x_{33} = 0$
$\forall i,$	$y_i \geq 0$	$y_1 = 0, y_2 = 1, y_3 = 0$
$\forall j,$	$\sum_{1 \leq i \leq n} x_{ij} \geq 1$	$x_{11} + x_{21} + x_{31} = 0 + 1 + 0 = 1,$ $x_{12} + x_{22} + x_{32} = 0 + 1 + 0 = 1,$ $x_{13} + x_{23} + x_{33} = 0 + 1 + 0 = 1$
$\forall i\, \forall j,$	$y_i \geq x_{ij}$	$y_1 = 0 = x_{11}, y_1 = 0 = x_{12}, y_1 = 0 = x_{13},$ $y_2 = 1 = x_{21}, y_2 = 1 = x_{22}, y_2 = 1 = x_{23},$ $y_3 = 0 = x_{31}, y_3 = 0 = x_{32}, y_3 = 0 = x_{33}$
	$\sum_{1 \leq i \leq n} y_i \leq k$	$y_1 + y_2 + y_3 = 0 + 1 + 0 = 1 = k$

13.2.2 Solving the ILP

The ILP is solved by using LP relaxation and branch-and-bound. If the solution is still not integral a rounding method described by Kececioglu should be used.
A K_n and a fractional solution is given. For all fractional y_i's the average of the weights of the adjacent edges can be computed:

$$a_i = \sum_{j:x_{ij}>0} \frac{w_{ij}}{|\{j:x_{ij}>0\}|}$$

The y_i with the highest solution is assigned as a center of a star graph.
Finally the solution defines the k star graphs that could be found in K_n. The hope is, that each star graph contains the reads from one repeat copy.

13.3 Summary

In this section I will emphasize what techniques and arguments are central to this part of the lecture in form of questions to which you should be able to answer.

- Finding DNPs

  - What are DNPs and what are they used for?
  - How can you find DNPs?
  - What is the probability to observe x coincidences between 2 columns with n_u and n_v deviations and k rows in total? Assume i.i.d. deviation probabilities.

- Separating repeat copies (aka grouping reads)
  - What is the k-star problem?
  - How can it be applied to separate multi-reads?
  - Formulate an ILP to solve it. How can the ILP be solved?

- Putting it all together
  - Describe the iterative process to separate multi-reads of an RNA-Seq dataset.

References

[1] J. Abel. Improvements to the burrows-wheeler compression algorithm: After bwt stages, 2003.

[2] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. Communications of the ACM-Vol. 29, No. 4, 1986.

[3] M. Burrows and D. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Systems Research, 1994.

[4] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. Computer, page 390, 2000.

[5] T. Langmead and B. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol., 10, R25, 2009.

[6] A. Mortazavi. Mapping and quantifying mammalian transcriptomes by RNA-Seq. Nat. Meth, 5:621–628, 2008.