4 NGS read mapping

This exposition has been developed by Knut Reinert. It is based on the following sources, which are all recommended reading:

1. Li, H, and Homer, N. (2010) *A survey of sequence alignment algorithms for next-generation sequencing*. Briefings in Bioinformatics 11 (5) (September 21): 473-483.
2. Holtgrewe, M., Emde A.-K., Weese D., Reinert K. (2011) *A Novel And Well-Defined Benchmarking Method For Second Generation Read Mapping*. BMC Bioinformatics 12 (1): 210.

The term *read mapping* has itself established since a couple of years for another, well studied problem, namely *approximate string matching* with certain application driven constraints.

The constraints are:

- usually DNA (or RNA) is considered (which means a *small alphabet* size).
- we have to map *short* strings (about 50 to 3000 bases) to a large string (billions of bases).
- there are relatively *few errors* allowed (usually around 3-4%, some application might go up to 10%).
- the problem sizes are very large (*billions* of small strings map to a string of size up to several billion characters).

4.1 Second-generation sequencing technologies

	454 FLX/Roche	Solexa/Illumina	SOLiD/ABI
Sequencing approach	pyrophosphate release	bridge amplification	ligation
Read lengths	400–500bp	36bp	35bp or 25bp (MP)
Mate pairs	yes	yes	yes
Output/Run	400–600Mbp in 10h	$>$ 1.5Gbp in 2.5d	3–4Gbp in 6d
Accuracy depends on	homopolymer length ($>$ 6 problematic)	nucleotide position in the read	nucleotide position in the read

GS FLX Titanium Series

Genome Analyzer 2

SOLiD System 2.0 Analyzer

The last slide was "old".

Illuminas HiSeq 2500 now produces at least 600 Gbp in about 12 days. That is about one billion reads, of length 100-150 bp in mate pairs. In addition, the end of higher throughput does not seem to be reached.

In addition, new technologies allow the sequencing of *single molecules*.

What are the applications for which this technology is currently used?

4.2 Second-generation sequencing applications

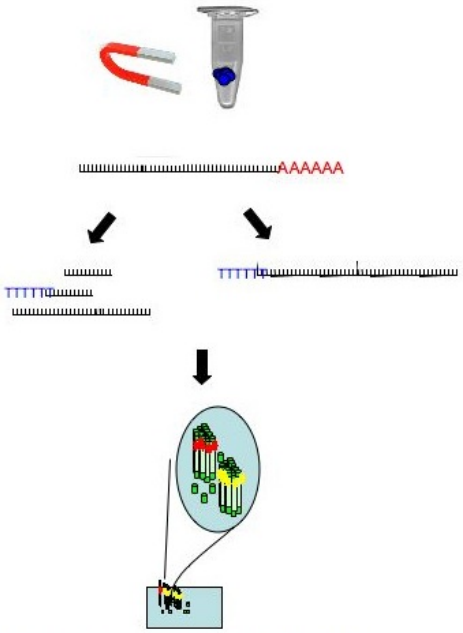
Genome Assembly	**RNA-Sequencing**	**Genome Comparison**
Resequencing	Expression Profiling	SNPs
De novo assembly	Alt. splicing microRNA	Rearrangements

Metagenomics	**Epigenetics**
	ChIP-Seq
	DNA Methylation

4.3 RNA-Sequencing

How RNA-Seq works:

- RNA isolatation
- Reverse transcription to cDNA
- Fragmentation
- (Size selection)
- Sequencing

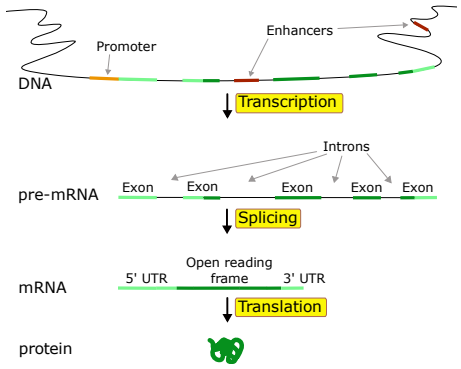


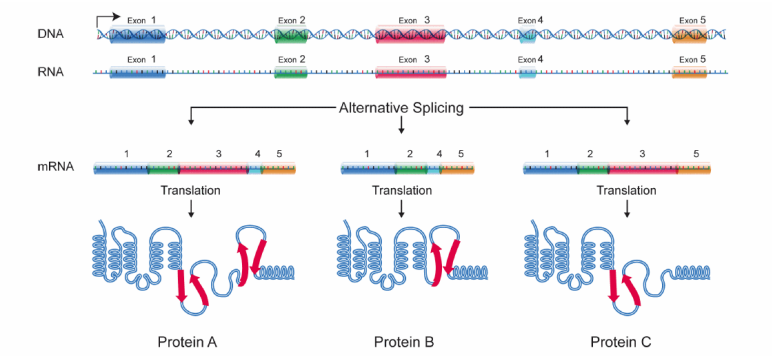
Illumina Solexa, Roche 454, or ABI SOLiD
Graphic shown here is Illumina

RNA-Seq applications:

- **Expression profiling:** Quantify gene expression levels
- **Alternative splicing:** Which mRNAs are generated from the same gene?
- **microRNA:** Where is the genomic source, which genes are regulated?

4.4 RNA-Seq - Alternative Splicing





Two approaches to determine splice variants:

1. Cut the genome at known splice sites and map mRNA reads onto combinations of merged genome fragments
2. Map as many mRNA reads as possible onto the genome and use coverage and known introns to detect new splice sites. Proceed as above.

Second Approach[a]:

- Map reads
- Assemble uniquely mapped reads
- Generate possible splices
- Try to map the non-uniquely mapped reads onto splices

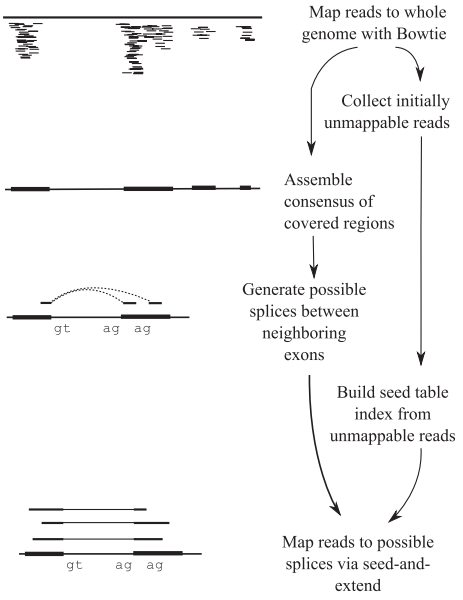


Fig. 1. The TopHat pipeline. RNA-Seq reads are mapped against the whole reference genome, and those reads that do not map are set aside. An initial consensus of mapped regions is computed by Maq. Sequences flanking potential donor/acceptor splice sites within neighboring regions are joined to form potential splice junctions. The IUM reads are indexed and aligned to these splice junction sequences.

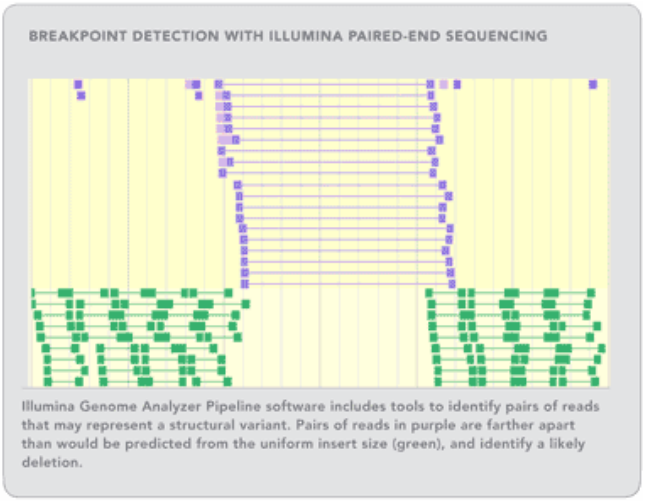
[a] Trapnell C, Pachter L, Salzberg SL. (2009) *TopHat: discovering splice junctions with RNA-Seq*, Bioinformatics

Genome Assembly	**RNA-Sequencing**	**Genome Comparison**
Resequencing	Expression Profiling	SNPs
De novo assembly	Alt. splicing microRNA	Rearrangements

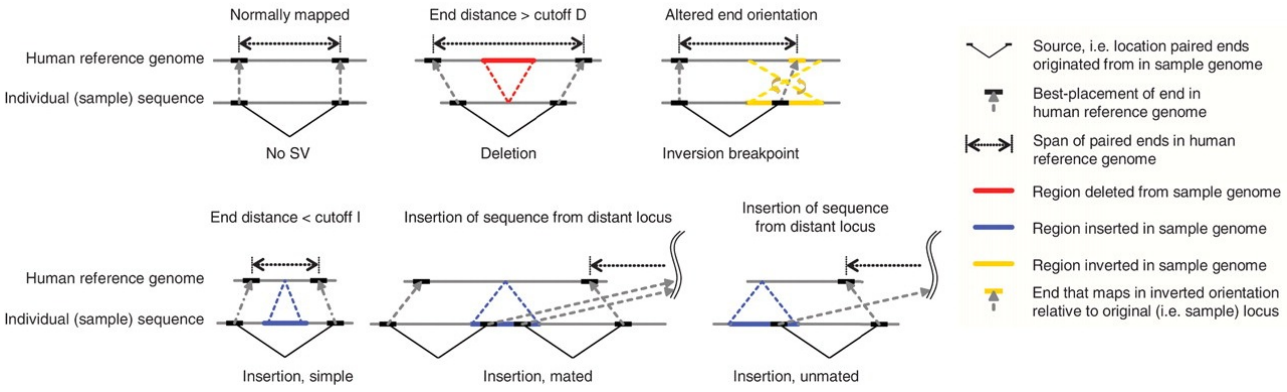
Metagenomics	**Epigenetics**
	ChIP-Seq
	DNA Methylation

4.5 Genome Comparison

- Sequence paired-end reads of an unknown genome (sample)
- Map them onto a known reference genome (target)
- Search for small mutations (SNPs) or large structural variations (rearrangements) between them

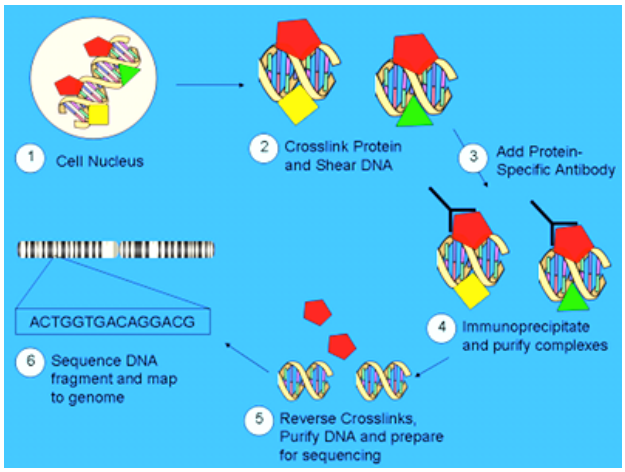


A deletion in the sample induces pairs of reads to be farther apart than predicted.

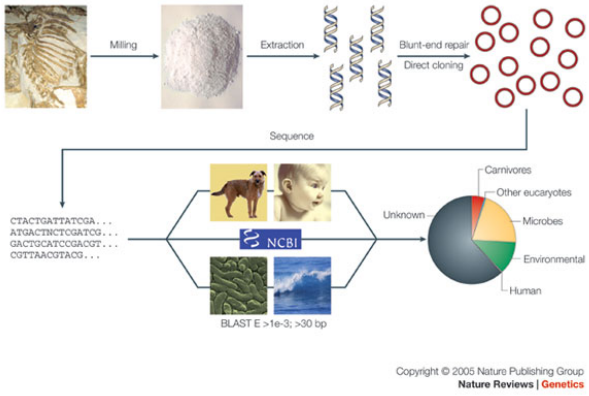


Inversions, deletions, translocations can also be detected.[1][2]

4.6 Other applications



ChIP-Sequencing[3]



Metagenomics[4]

Fundamental to almost all of these applications is the following problem:

Problem 1 (Read Mapping Problem). Given a set of read sequences $\mathcal{R}$, a reference sequence G, and a distance $k \in \mathbb{N}$. Find all pairs (r, g) with $r \in \mathcal{R}$, g is substring of G and $\text{dist}(r, g) \leq k$.

Common distance measures are Hamming distance or edit distance.
The pairs (r, g) are called **matches** of r.

However, depending on the application, we have to adapt the problem definition.

4.7 Objective functions

By now it should be clear to you, that the term read mapping subsumes a number of different objective functions. In the normal case we want to find the approximate occurrences of a complete read, that is, conduct a *semi-global* alignment.

If we have for example RNA reads, then the read may corresponds to several genomic loci that have been spliced together. In this case we speak of *split* alignment to distinguish it from local alignment. In split alignment we want to find the *complete* read, whereas this is not necessary in local alignment.

The problem of split alignment can be further subdivided depending on the decision whether we allow parts of the split read to be reverse complemented (e.g. assembly error), be missing, or out of order (e.g. genomics insertions or deletions).

Usually split read mapping is reduced to several subproblems of normal read mapping.

Finally, a distinction is made whether the approximate string matching supports (weighted) edit distance or only the Hamming distance.

While the edit distance is preferable, it makes the problem computationally harder. Often you will find in read mapping heuristics some "in-between" formulations (e.g. *supports mismatches and up to 2 insertions*).

Be aware of such limitations.

[1] Korbel JO, Urban AE, Affourtit JP, et al. (2007) *Paired-End Mapping Reveals Extensive Structural Variation in the Human Genome*, Science
[2] Bashir A, Volik S, Collins C, Bafna V, Raphael BJ. (2008) *Evaluation of paired-end sequencing strategies for detection of genome rearrangements in cancer*, PLoS computational biology
[4] Barski A, Cuddapah S, Cui K, Roh TY, Schones DE, Wang Z, Wei G, Chepelev I, Zhao K (2007) *High-resolution profiling of histone methylations in the human genome*, Cell
[4] Poinar HN, Schuster SC, et al. (2006) *Metagenomics to Paleogenomics: Large-Scale Sequencing of Mammoth DNA*, Science

If we have a fixed objective function for our special approximate string matching problem, we can still make distinctions about the set of matches we want to find. A reasonable distinction could be the tasks of finding:

1. *all* matches with up to k errors.
2. *all* best matches.
3. *any* best match.

Doing this of course implies to have a good definition, what we actually mean with a match?

Have a look at the following situation:

4.8 Benchmarking

Different kind of approximate matches.

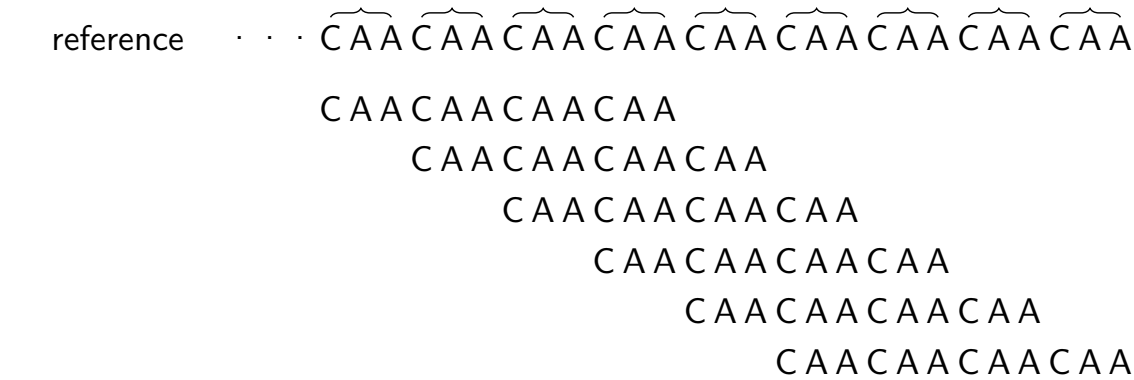
Say, we want to find the best two matches of the read in the reference sequence, with an edit distance of up to 3. Both locations in the reference sequence are shown. The row alignments shows two alignments of the read to the reference sequence that appear to be optimal. However, the alignments in the rows below have a lower edit distance than the right one.

Common sense would tell us that the alignments in the left column are not significantly different, though. Each alignment with distance k induces alignments with distance at most $k + 2$ by aligning the leftmost/rightmost base one more position to the left/right and introducing a gap.

Repeats are another issue. Consider the tandem repeats in the below figure.

Large period repeat.

Intuitively, we can identify the two distinct alignments in this situation. Now look at a tandem repeat with a shorter period:



Small period repeat.

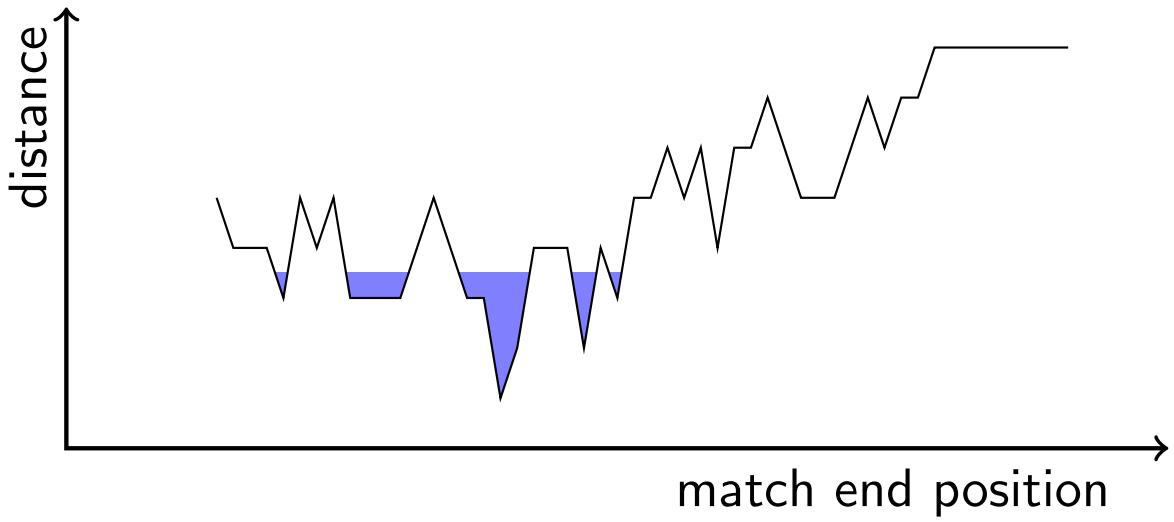
Do we really want to find all those alignments?

Counting alignments in this way would require a read mapper to find lots of positions in repeat regions. This is not desirable since reads from long tandem repeat regions would get a higher weight with this counting scheme than reads from short tandem repeat regions or reads from non-repeat regions.

Only weighting each found match with $1/n$ (where n is the number of positions the read aligns at) is also deficient (why?).

Hence it is more desirable to define when two matches are considered the same and when they should be counted separately.

Without giving the details, one can define an equivalence relation on the set of matches, which can be depicted as follows as an *error landscape*.



Error landscape.

Flooding this landscape to the respective error level gives a number of intervals, which in turn can be used to define the specificity and sensitivity of read mappers. In the benchmark it is sufficient to return one endposition within the interval.

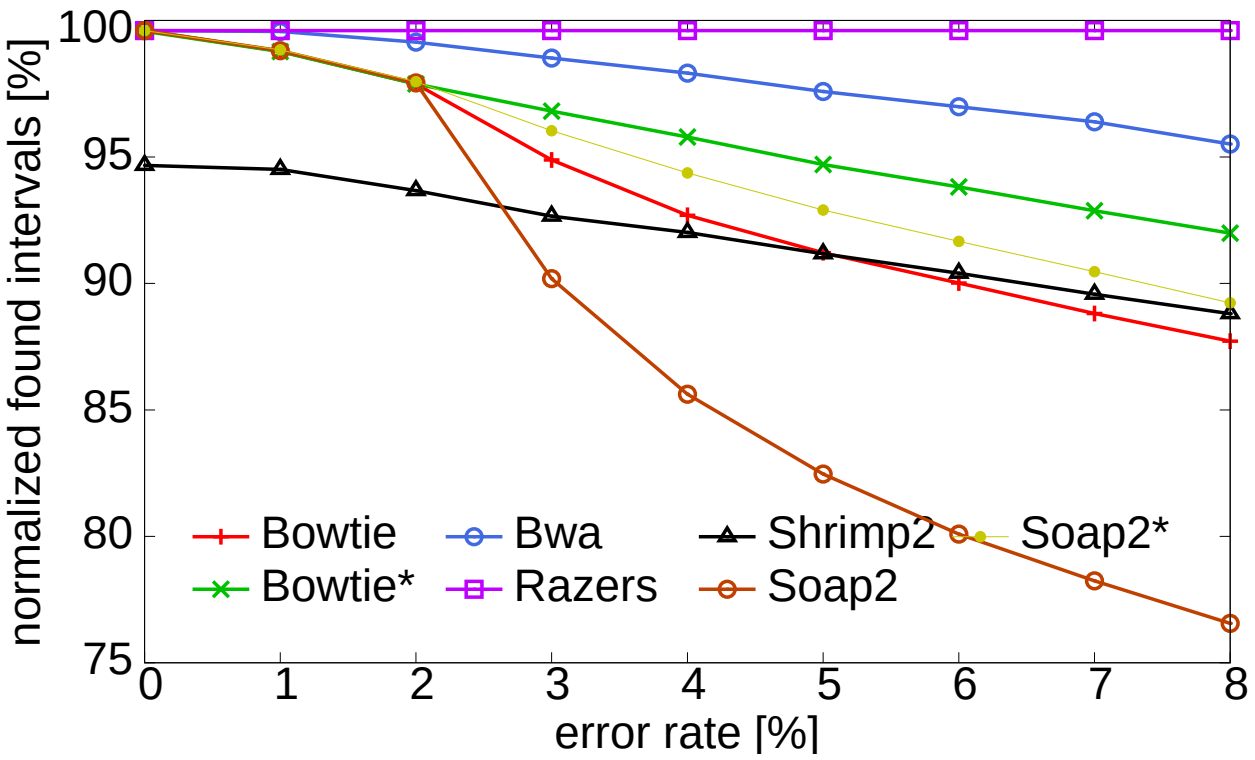
Some of the intervals will be labelled *optimal*, if they contain a matching position with the minimal distance (e.g. edit or hamming). If we benchmark read mapping application with the goal *find any best*, then it should return an alignment ending in one of those intervals. If we have the goal *find all best*, the read mapper should return all optimal intervals, etc.

This can now be used to make comprehensive comparisons between different methods to compare their performance.

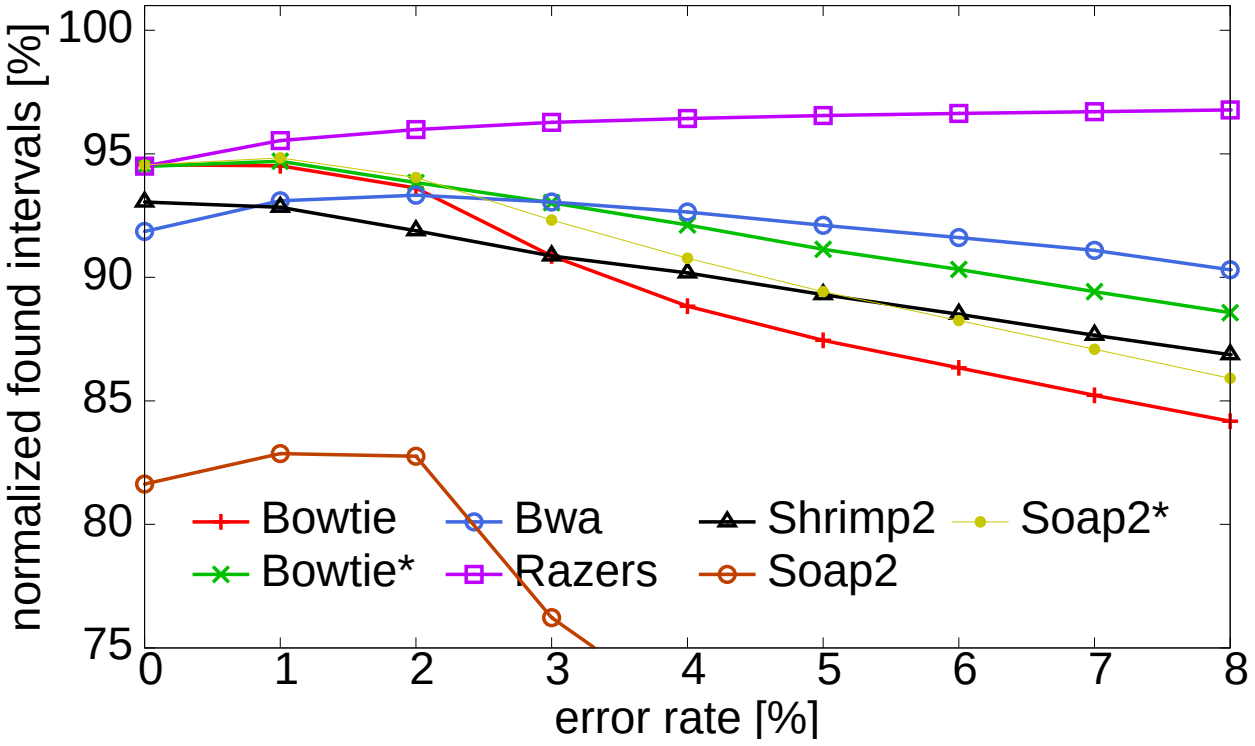
As an evaluation metric we use the number of *normalized found intervals*.

This is defined as follows: Each read gives at most one point. If a read matches at n locations (i.e. intervals), each found location gives $1/n$ point. To get percentages, the number of achieved points is divided by the number of reads and multiplied by 100.

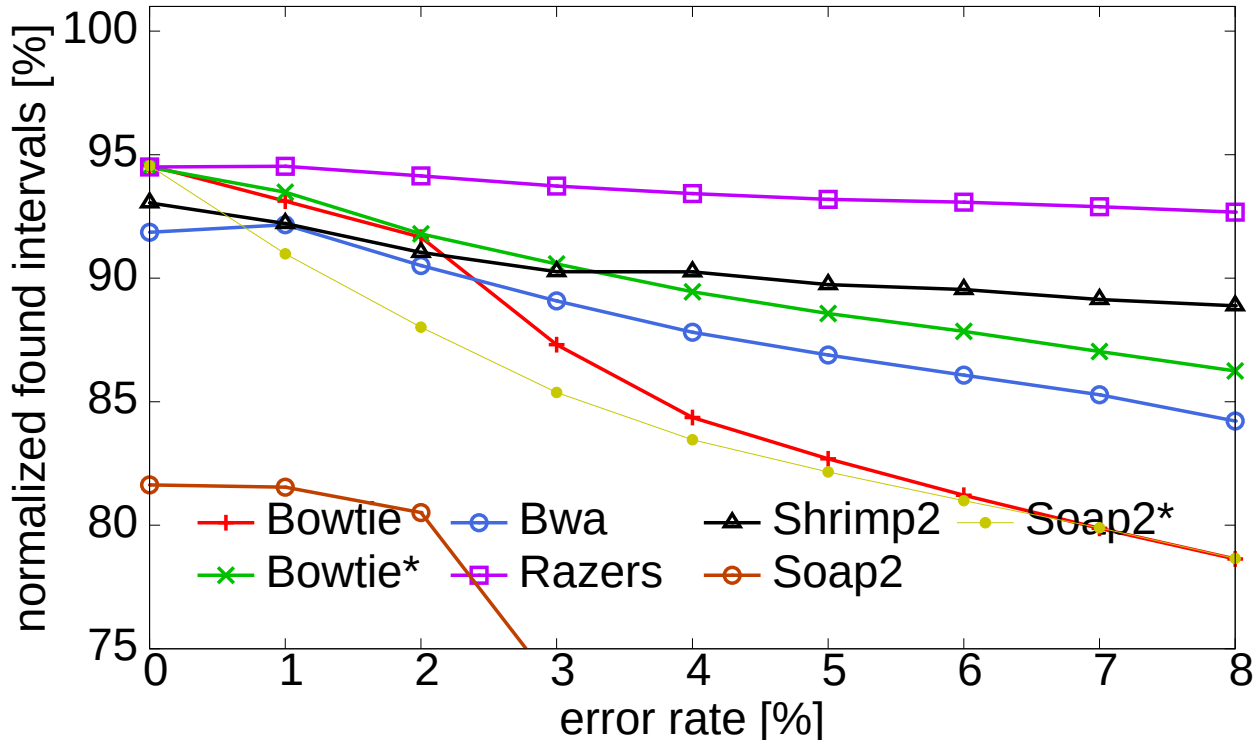
Have a look at results of recent read mappers (2011) for the three different categories (Illumina reads of *Drosophila Melanogaster*, 100 bp length), but mind that those plots do not give the run times.



Benchmark *any best*.



Benchmark *all best*.



Benchmark *all*.

4.9 Computational paradigms

Lets go back to algorithmic paradigms used in read mapping algorithms.

Given the large data, obviously all algorithms use some *string indices* to preprocess the reads, the genome, or both. The indices can be used directly for searching as in the case of the enhanced suffix array or Burrows Wheeler transform (BWT), or they are used to filter out regions that do not contain matches (as in the case of (gapped) q-gram indices).

You have already encountered a simple filter that is based on a q-gram index and uses a simple version of the q-gram lemma. This paradigm is called *q-gram counting*. We will sketch a more advanced algorithm. First, we will talk about q-gram pidgeonhole filter and a hieracharchical verification scheme introduced by Navarro.

5 Fast filtering algorithms

This exposition is based on

1. Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, chapter 6.5, pages 162ff.

We present the hierarchical filtering approach called *PEX* of Navarro and Baeza-Yates.

5.1 Filtering algorithms

The idea behind filtering algorithms is that it might be easier to check that a text position does *not* match a pattern string than to verify that it does.

Filtering algorithms *filter out* portions of the text that cannot possibly contain a match, leaving positions that could match.

The potential match positions then need to be *verified* with another algorithm like for example the bit-parallel algorithm of Myers (BPM).

Filtering algorithms are very sensitive to the *error level* $\alpha := k/m$ since this normally affects the amount of text that can be discarded from further consideration. (m = pattern length, k = errors.)

If most of the text has to be verified, the additional filtering steps are an overhead compared to the strategy of just verifying the pattern in the first place.

On the other hand, if large portions of the text can be discarded quickly, then the filtering results in a faster search.

Filtering algorithms can improve the average-case performance (sometimes dramatically), but not the worst-case performance.

5.2 The pigeonhole principle

Assume that we want to find all occurrences of a pattern $P = p_1, \ldots, p_m$ in a text $T = t_1, \ldots, t_n$ that have an edit distance of at most k.

If we *divide* the pattern into $k + 1$ pieces $P = p^1, \ldots, p^{k+1}$, then at least *one* of the pattern pieces match *without error*.

There is a more general version of this principle first formalized by Myers in 1994:

Lemma 2. *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 nonnegative 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 $\lfloor a_i k/A \rfloor$ errors.*

Proof: *Exercise.*

So the basic procedure is:

1. *Divide:* Divide the pattern into $k + 1$ pieces of approximately the same length.
2. *Search:* Search all the pieces simultaneously with a multi-pattern string matching algorithm. According to the above lemma, each possible occurrence will match at least one of the pattern pieces.
3. *Verify:* For each found pattern piece, check the neighborhood with a verification algorithm that is able to detect an occurrence of the whole pattern with edit distance at most k. Since we allow indels, if $p_{i_1} \ldots p_{i_2}$ matches the text $t_j \ldots t_{j+i_2-i_1}$, then the verification has to consider the text area $t_{j-(i_1-1)-k} \ldots t_{j+(m-i_1)+k}$, which is of length $m + 2k$.

5.3 An example

Say we want to find the pattern *annual* in the texts

t_1 = *any_annealing* and

t_2 = *an_unusual_example_with_numerous_verifications*

with at most 2 errors.

1. *Divide:* We divide the pattern *annual* into p^1 = *an*, p^2 = *nu*, and p^3 = *al*. One of these subpattern has to match with 0 errors.
2. *Search:* We search for all subpatterns:

```
1: searching for an:  in t_1:  find positions 1, 5
                      in t_2:  find position  1
2: searching for nu:  in t_1:  find no positions
                      in t_2:  find positions 5, 25
3: searching for al:  in t_1:  find position 9
                      in t_2:  find position 9
```

3. *Verification:* We have to verify 3 positions in t_1, and 4 positions in t_2, to find 3 occurrences in t_1 and *none* in t_2.

5.4 Hierarchical verification

The toy example makes clear that *many* verifications can be triggered that are unsuccesssful and that many subpatterns can trigger the *same* verification. Repeated verfications can be avoided by carefully sorting the occurrences of the pattern.

It was shown by Baeza-Yates and Navarro that the running time is dominated by the multipattern search for error levels $\alpha = k/m$ below $1/(3\log_{|\Sigma|} m)$. In this region, the search cost is about $O(kn\frac{\log_{|\Sigma|} m}{m})$. For higher error levels, the cost for verifications starts to dominate, and the filter efficiency deteriorates abruptly.

Baeza-Yates and Navarro introduced the idea of hierarchical verification to reduce the verification costs, which we will explain next. Then we will work out more details of the three steps.

Navarro and Baeza-Yates use Lemma 2 for a *hierarchical verification*. The idea is that, since the verification cost is high, we pay too much for verifying the *whole* pattern *each* time a small piece matches. We could possibly reject the occurrence with a cheaper test for a shorter pattern.

So, instead of directly dividing the pattern into $k + 1$ pieces, we do it hierarchically. We split the pattern first in two pieces and search for each piece with $\lfloor k/2 \rfloor$ errors, following Lemma 2. The halves are then recursively split and searched until the error rate reaches zero, i. e. we can search for exact matches.

With hierarchical verification the area of applicability of the filtering algorithm grows to $\alpha < 1/\log_{|\Sigma|} m$, an error level three times as high as for the naive paritioning and verification. In practice, the filtering algorithm pays off for $\alpha < 1/3$ for medium long patterns.

Example. Say we want to find the pattern P = `aaabbbcccddd` in the text T = `xxxbbbxxxxxx` with at most $k = 3$ differences. The pattern is split into four pieces p^1 = `aaa`, p^2 = `bbb`, p^3 = `ccc`, p^4 = `ddd`. We search with $k = 0$ errors in level 2 and find *bbb*.

```
level 0              aaabbbcccddd              with k=3 errors
                     /          \
level 1         aaabbb          cccddd        with k=1 errors
                /    \          /    \
level 2       aaa    bbb      ccc    ddd      with k=0 errors
```

Now instead of verifying the complete pattern in the complete text (at level 0) with $k = 3$ errors, we only have to check a slightly bigger pattern (`aaabbb`) at level 1 with one error. This is much cheaper. In this example we can decide that the occurrence `bbb` cannot be extended to a match.

```
level 0              aaabbbcccddd            with k=3 errors
                     /          \
level 1          AAABBB          cccddd      with k=1 errors
                 /    \          /    \
level 2        aaa    BBB      ccc    ddd    with k=0 errors
```

5.5 The PEX algorithm

Divide: Split pattern into $k+1$ pieces, such that each piece has equal probability of occurring in the text. If no other information is available, the uniform distribution is assumed and hence the pattern is divided in pieces of equal length.

Build Tree: Build a tree of the pattern for the hierarchical verification. If $k+1$ is not a power of 2, we try to keep the binary tree as balanced as possible.

Each node has two members *from* and *to* indicating the first and the last position of the pattern piece represented by it. The member *err* holds the number of allowed errors. A pointer *myParent* leads to its parent in the tree. (There are no child pointers, since we traverse the tree only from the leafs to the root.) An internal variable $left$ holds the number of pattern pieces in the left subtree. *idx* is the next leaf index to assign. *plen* is the length of a pattern piece.

Algorithm CreateTree generates a hierarchical verification tree for a single pattern. (Lines 12 and 14 are justified by Lemma 2.)

```
(1)  CreateTree( p = p_i p_{i+1} ... p_j, k, myParent, idx, plen )
(2)  // Note: the initial call is: CreateTree ( p, k, nil, 0, ⌊m/(k+1)⌋ )
(3)  Create new node node
(4)  from(node) = i
(5)  to(node) = j
(6)  left = ⌈(k+1)/2⌉
(7)  parent(node) = myParent
(8)  err(node) = k
(9)  if k = 0
(10)   then leaf_idx = node
(11)   else
(12)       lk = ⌊(left · k)/(k+1)⌋
(13)       CreateTree( p_i ... p_{i+left·plen−1}, lk, node, idx, plen )
(14)       rk = ⌊((k+1−left) · k)/(k+1)⌋
(15)       CreateTree( p_{i+left·plen} ... p_j, rk, node, idx+left, plen )
(16) fi
```

Example: Find the pattern P = `annual` in the text T = `annual_CPM_anniversary` with at most $k = 2$ errors. First we build the tree with $k+1 = 3$ leaves. Below we write at each node n_i the variables $(from, to, error)$.

```
                    "annual" n4=(1,6,2)
                    /                  \
           "annu" n3=(1,4,1)            \
            /         \                  \
"an" n0=(1,2,0)   "nu" n1=(3,4,0)   "al" n2=(5,6,0)
    |                 |                  |
leaf 0            leaf 1             leaf 2
```

Search: After constructing the tree, we have $k+1$ leafs $leaf_i$. The $k+1$ subpatterns

$$\{\, p_{from(n)}, \ldots, p_{to(n)},\ n = leaf_i,\ i \in \{0, \ldots, k\} \,\}$$

are sent as input to a multi-pattern search algorithm (e. g. Aho-Corasick, Wu-Manbers, or SBOM). This algorithm gives as output a list of pairs (pos, i) where *pos* is the text position that matched and i is the number of the piece that matched.

The PEX algorithm performs verifications on its way upward in the tree, checking the presence of longer and longer pieces of the pattern, as specified by the nodes.

```
(1)  Search phase of algorithm PEX
(2)  for (pos, i) ∈ output of multi-pattern search do
(3)      n = leaf_i; in = from(n); n = parent(n);
(4)      cand = true;
(5)      while cand = true and n ≠ nil do
(6)              p_1 = pos − (in − from(n)) − err(n);
(7)              p_2 = pos + (to(n) − in) + err(n);
(8)              verify text t_{p_1} ... t_{p_2} for pattern piece p_{from(n)} ... p_{to(n)}
(9)                 allowing err(n) errors;
(10)             if pattern piece was not found
(11)               then cand = false;
(12)               else n = parent(n);
(13)             fi
(14)     od
(15)     if cand = true
(16)       then report the positions where the whole p was found;
(17)     fi
(18) od
```

We search for `annual` in `annual_CPM_anniversary`. We constructed the tree for `annual`. A multi-pattern search algorithm finds: $(1,1)$, $(12,1)$, $(3,2)$, $(5,3)$. (Note that leaf i corresponds to pattern p^{i+1}). For each of these positions we do the hierarchical verification:

```
Initialization for (1,1);
n=n0; in=1; n=n3; cand=true;
While loop;
  a) p1=1-(1-1)-1=0;  p2=1+(4-1)+1=5;
  verify pattern annu in text annua with 1 error => found !
  b) p1=1-(1-1)-2=-1; p2=1+(6-1)+2=8;
  verify pattern annual in text annual_C => found !
  c) report end positions (6,7,8)

Initialization for (3,2);
n=n1; in=3; n=n3; cand=true;
While loop;
  a) p1=3-(3-1)-1=0;  p2=3+(4-3)+1=5;
  verify pattern annu in text annua with 1 error => found !
  b) p1=3-(3-1)-2=-1; p2=3+(6-3)+2=8;
  verify pattern annual in text annual_C => found !
  c) report end positions (6,7,8)

Initialization for (12,1);
n=n0; in=1; n=n3; cand=true;
While loop;
  a)  p1=12-(1-1)-1=11; p2=12+(4-1)+1=16;
  verify pattern annu in text _anniv with 1 error => found !
  b)  p1=12-(1-1)-2=10; p2=12+(6-1)+2=19;
  verify pattern annual in text M_annivers => NOT found !
```

5.6 Summary

- Filtering algorithms prevent a large portion of the text from being looked at.
- The larger $\alpha = k/m$, the less efficient filtering algorithms become.
- Filtering algorithms based on the pigeonhole principle need an exact, multi-pattern search algorithm and a verification capable approximate string matching algorithm.
- The PEX algorithm starts verification from short exact matches and considers longer and longer substrings of the pattern as the verification proceeds upward in the tree.