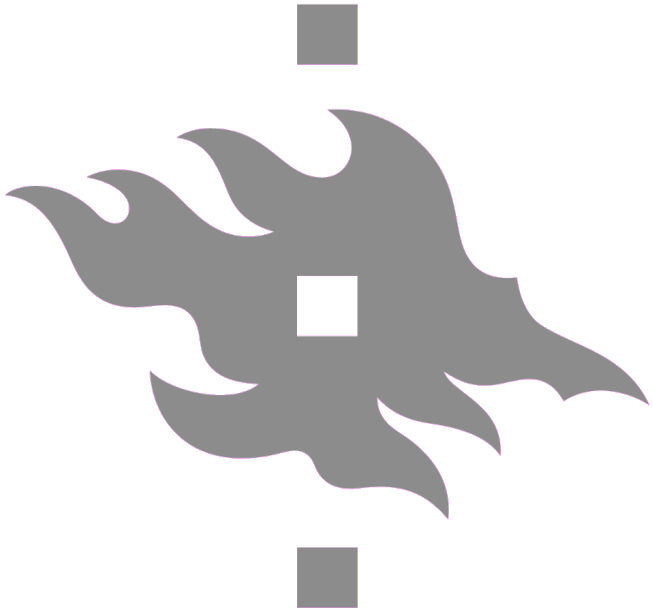
Scalable Indexing of Highly Repetitive Data



Hector Ferrada

Travis Gagie

Tommi Hirvola

Simon J. Puglisi : University of Helsinki

Outline

- Indexing/searching highly repetitive data
  - Problem, Motivation, What's been done
- "Solution": Hybrid Indexing
- Some (preliminary) Experimental Results
- Future Directions

Indexing Highly Repetitive Data

- Genomic Collections: 100's or 1000's of genomes of individuals of the same species
- Multi-author Collections: Wikipedia archives; Source code repositories
- Web crawls: copied/quoted/reused text and images; boilerplate
- Archives: Backup facilities; Personal online storage (like Google Drive)

Highly Repetitive Genomic Data

There are many indexes* for approximate pattern matching (read alignment) in 1 genome, but they don't scale well to 1000s of genomes

---

*BFAST, Bowtie, BWA, CUSHAW, GASSST, MAQ, Novoalign, SeqAlto, SeqMap, SHRiMP, Slider, Snap, SOAP, Stampy, Taipan, Velvet, etc.

Aim (of this work)

Find a way to scale current read aligners to multiple genomes that is **independent of the aligner itself**.

Choose an aligner (your favorite aligner); we provide an algorithmic tool to make it work for multiple genomes.

One restriction...

We will cap – at index construction time –

- Maximum pattern (read) length M, and
- Maximum number of alignment errors, K

For many biological applications patterns are "small": 10s to 100s of characters

Two Algorithmic Tools

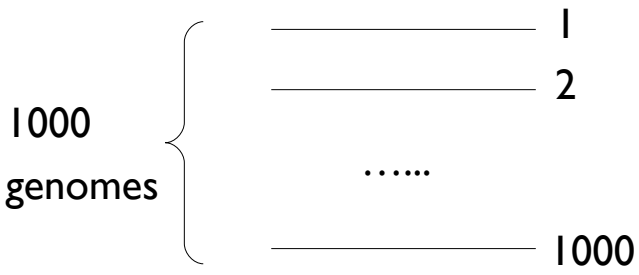
Our index is based on two main algorithmic tools...

- LZ77 parsing (or factorization)
  - Widely used in data compression (gzip and 7zip)
  - We use it for compression AND pattern matching

- 2-dimensional, 2-sided range reporting
  - A notion from computational geometry

The Hybrid Index...

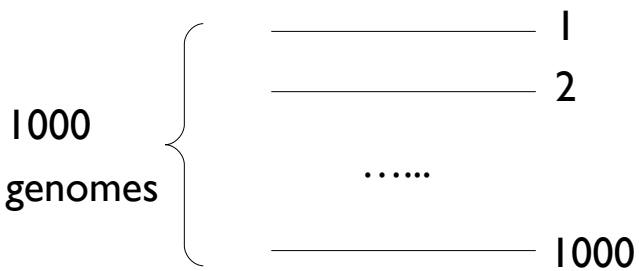
Input



M : upper bound on read length; e.g. M = 100

K : maximum # of alignment errors; e.g. K = 3

Input

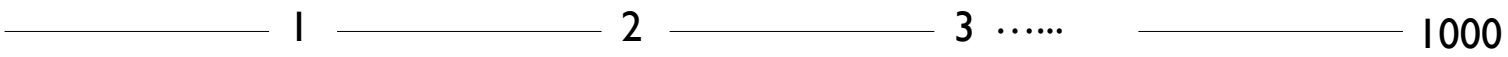


M : upper bound on read length; e.g. M = 100

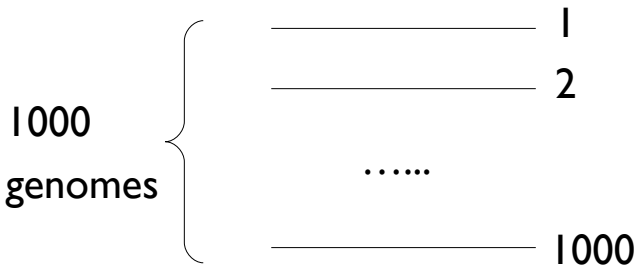
K : maximum # of alignment errors; e.g. K = 3

Indexing

1. Concatenate genomes into one long string



Input

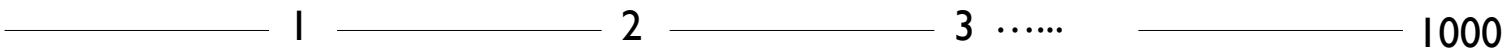


M : upper bound on read length; e.g. M = 100

K : maximum # of alignment errors; e.g. K = 3

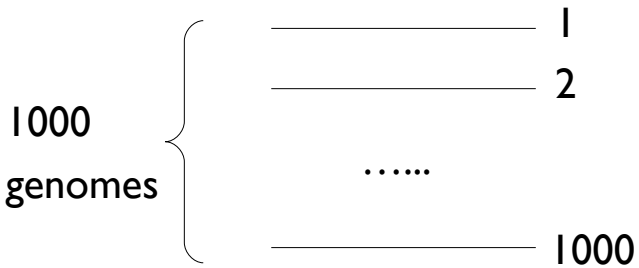
Indexing

1. Concatenate genomes into one long string



2. Compute LZ77 parsing

Input

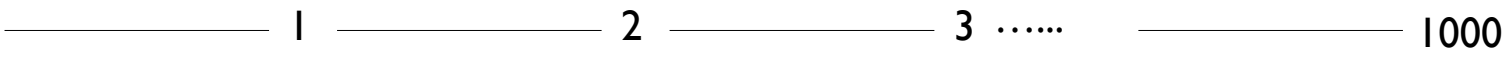


M : upper bound on read length; e.g. M = 100

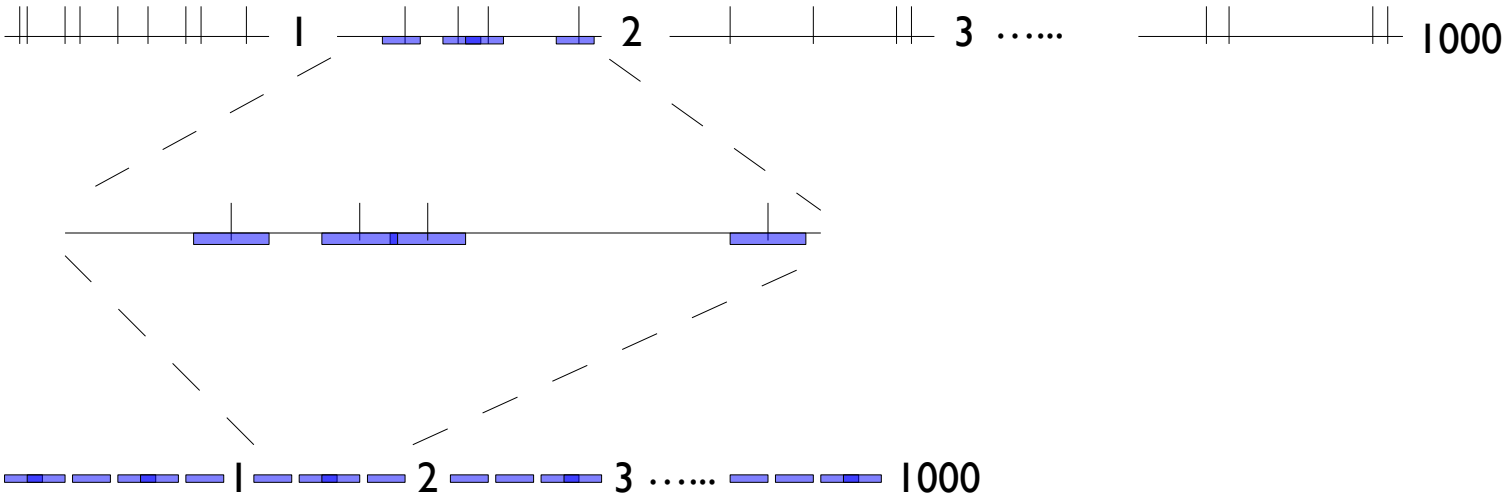
K : maximum # of alignment errors; e.g. K = 3

Indexing

1. Concatenate genomes into one long string

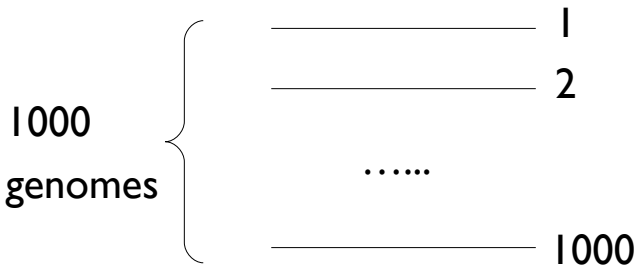


2. Compute LZ77 parsing



3. Patches of length M+K around each LZ77 phrase

Input

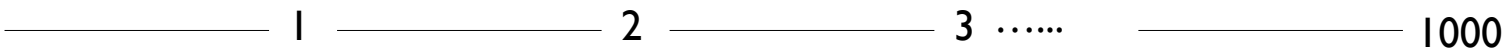


M : upper bound on read length; e.g. M = 100

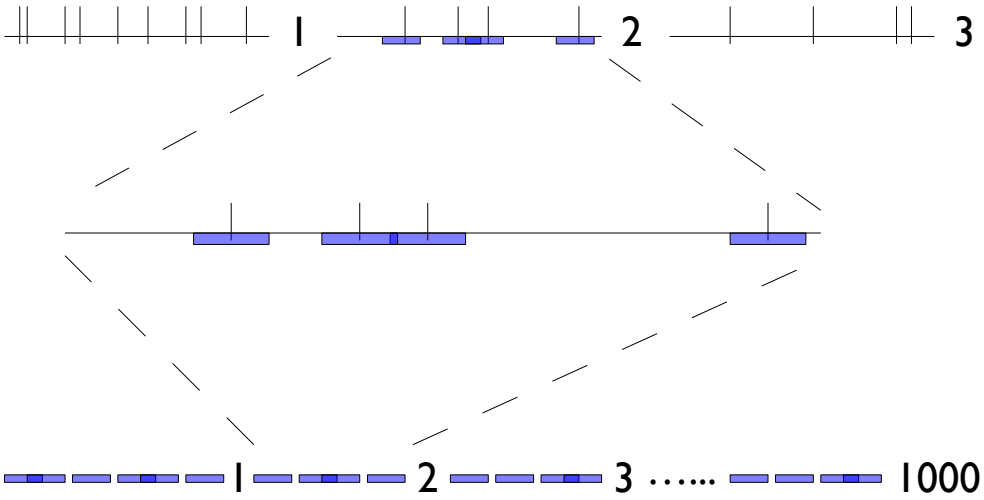
K : maximum # of alignment errors; e.g. K = 3

Indexing

1. Concatenate genomes into one long string



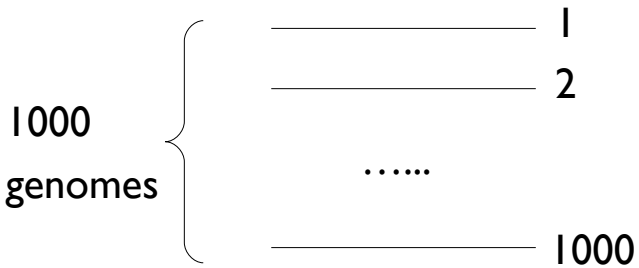
2. Compute LZ77 parsing



3. Patches of length M+K around each LZ77 phrase

4. Build a regular index on this filtered input

Input

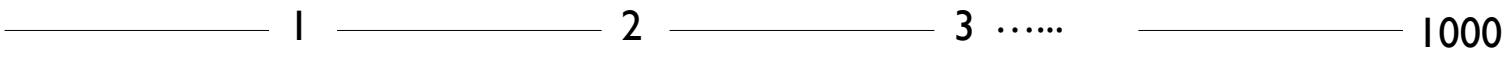


M : upper bound on read length; e.g. M = 100

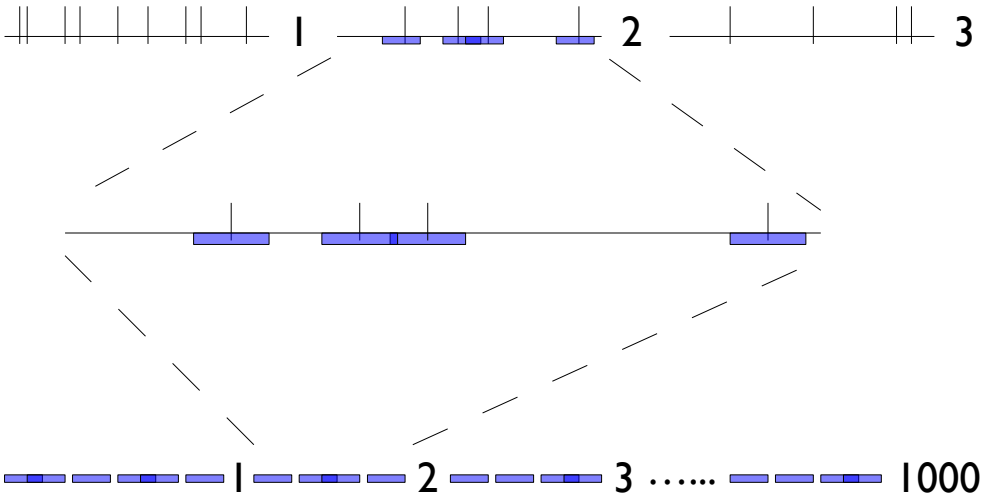
K : maximum # of alignment errors; e.g. K = 3

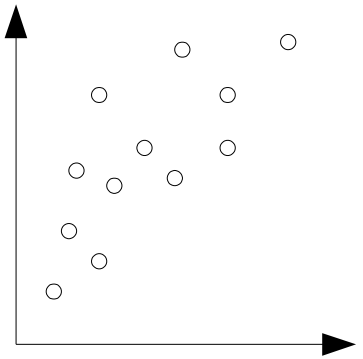
Indexing

1. Concatenate genomes into one long string



2. Compute LZ77 parsing





3. Patches of length M+K around each LZ77 phrase

4. Build a regular index on this filtered input

5. Phrase source boundaries in a 2D 2-sided range reporting data structure

Lempel-Ziv Parsing...

Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position $p_i < i$ in X

Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position $p_i < i$ in X

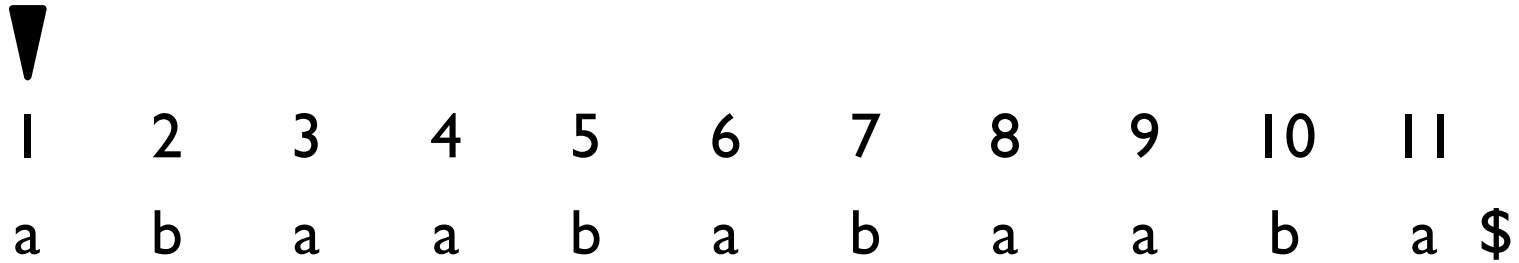
1	2	3	4	5	6	7	8	9	10	11	
a	b	a	a	b	a	b	a	a	b	a	$

Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position $p_i < i$ in X

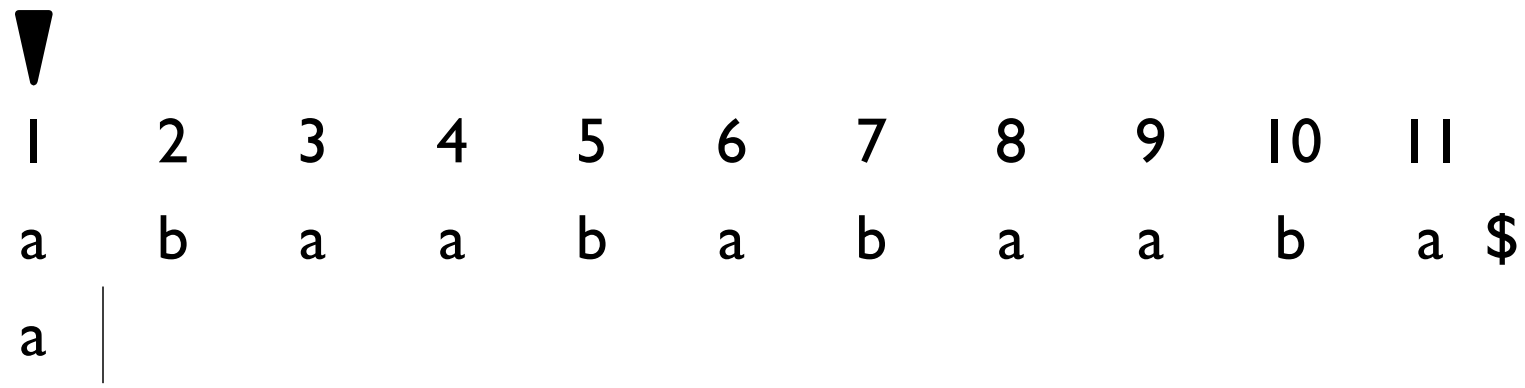


Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position $p_i < i$ in X

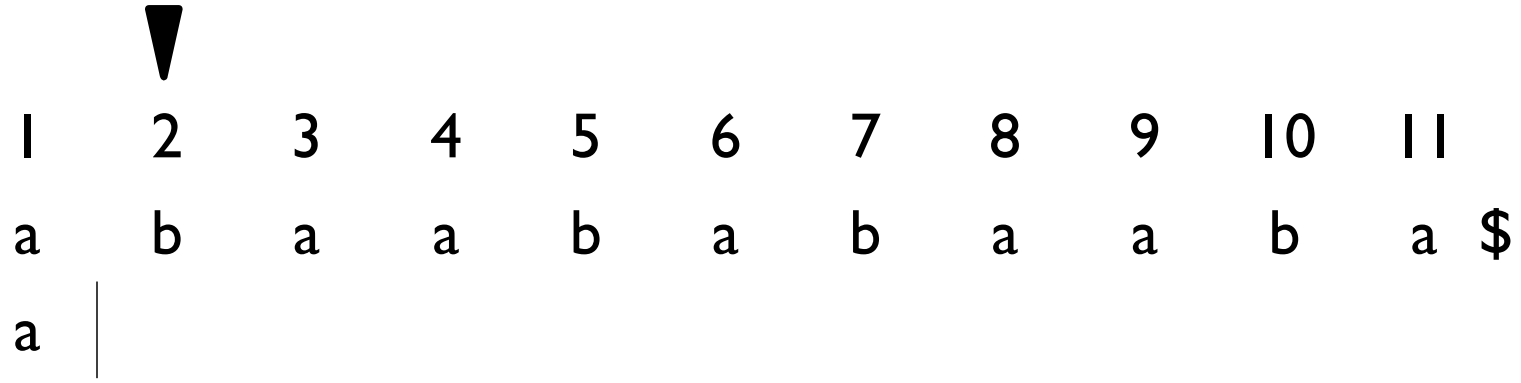


Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position p_i < i in X

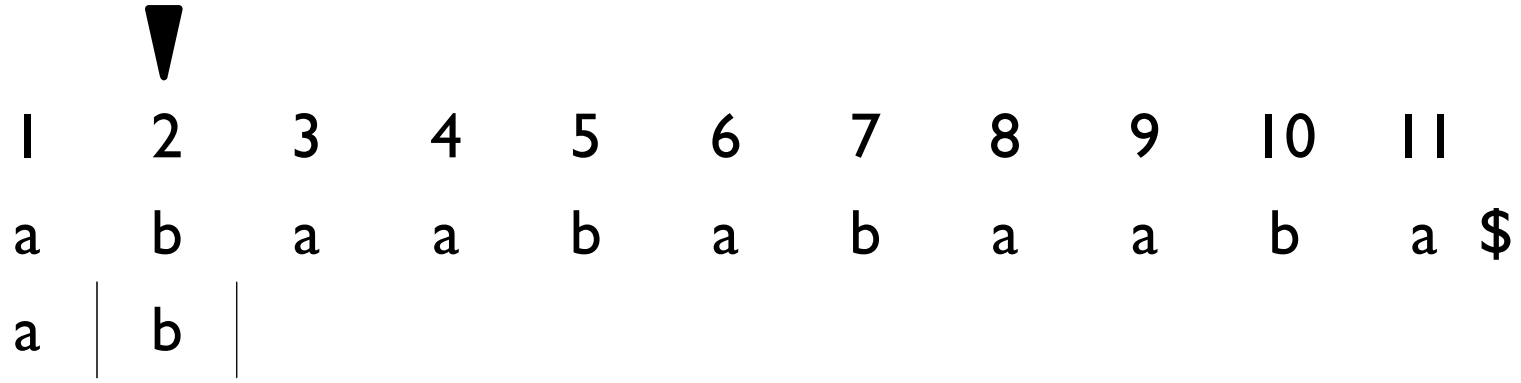


Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position $p_i < i$ in X

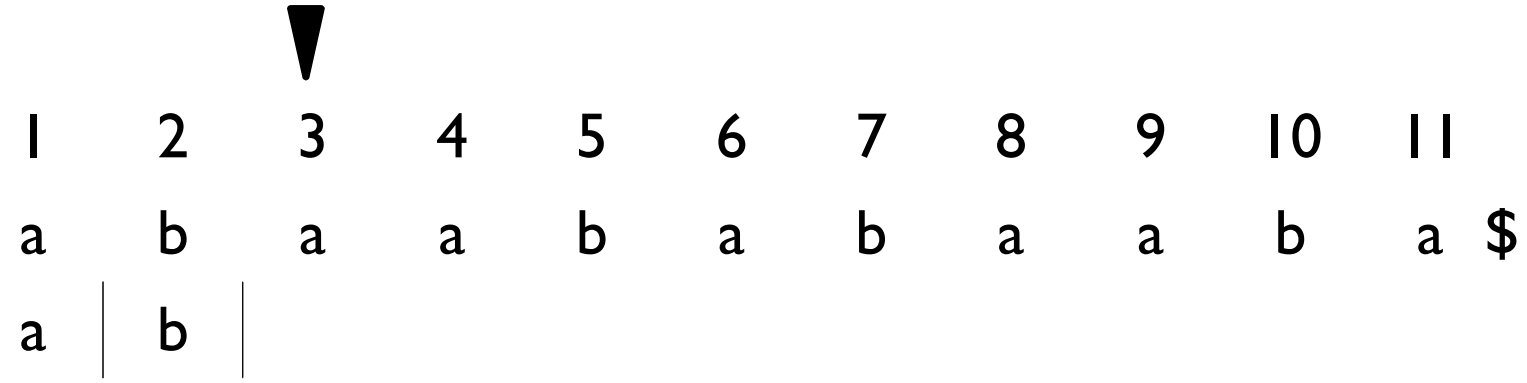


Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position $p_i < i$ in X

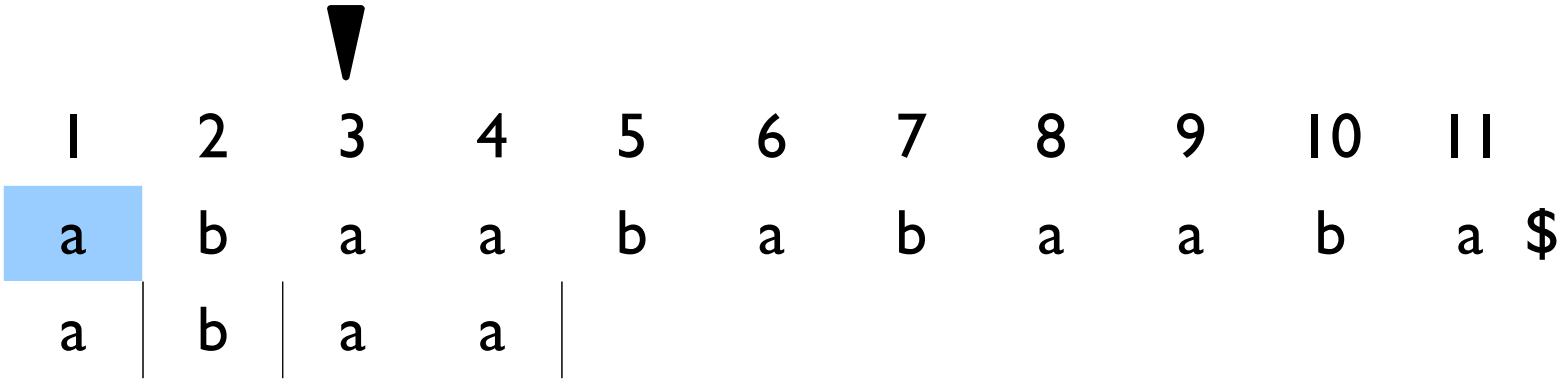


Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position p_i < i in X



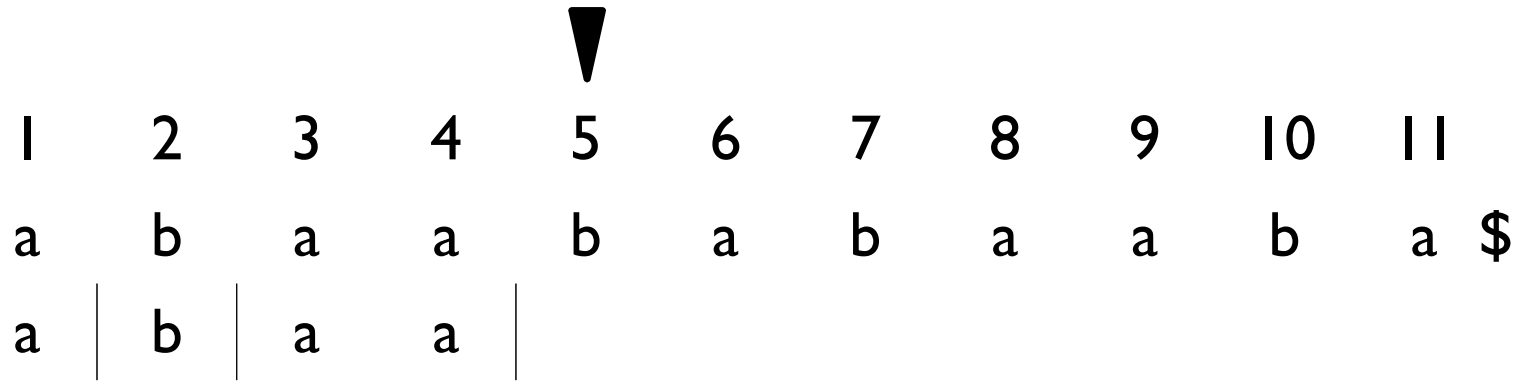
Source = (1,1)

Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position p_i < i in X

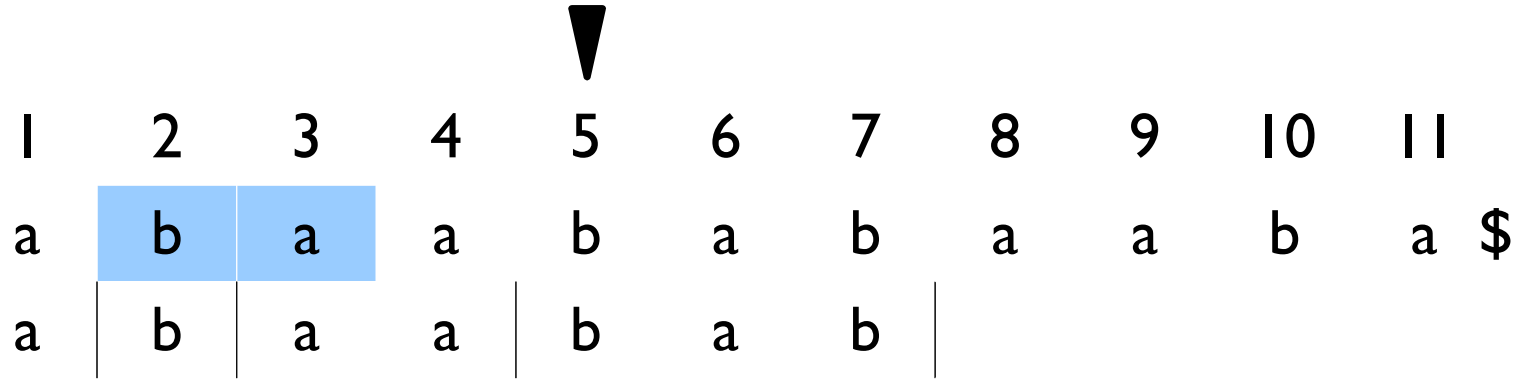


Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position p_i < i in X



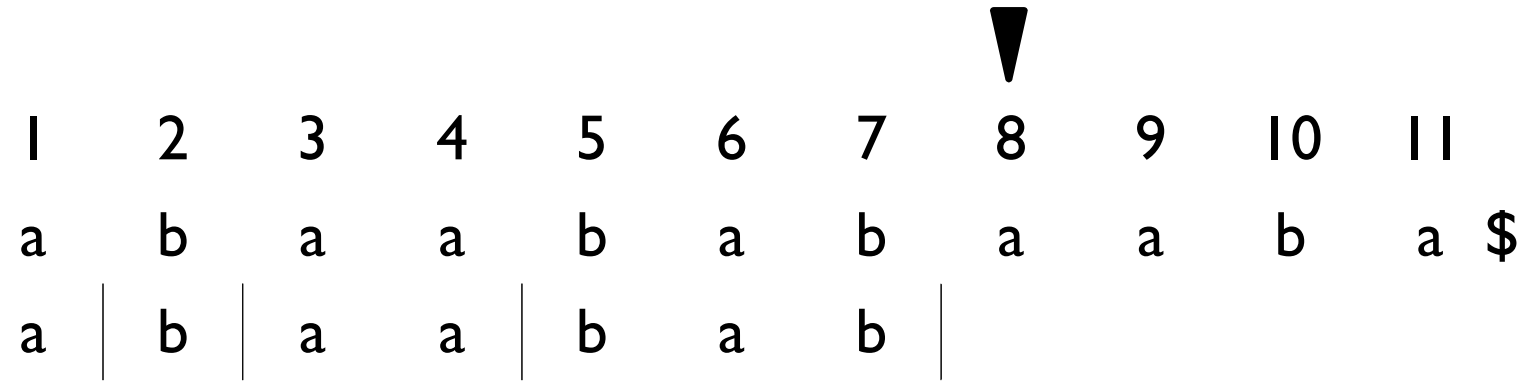
Source = (2,3)

Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position p_i < i in X

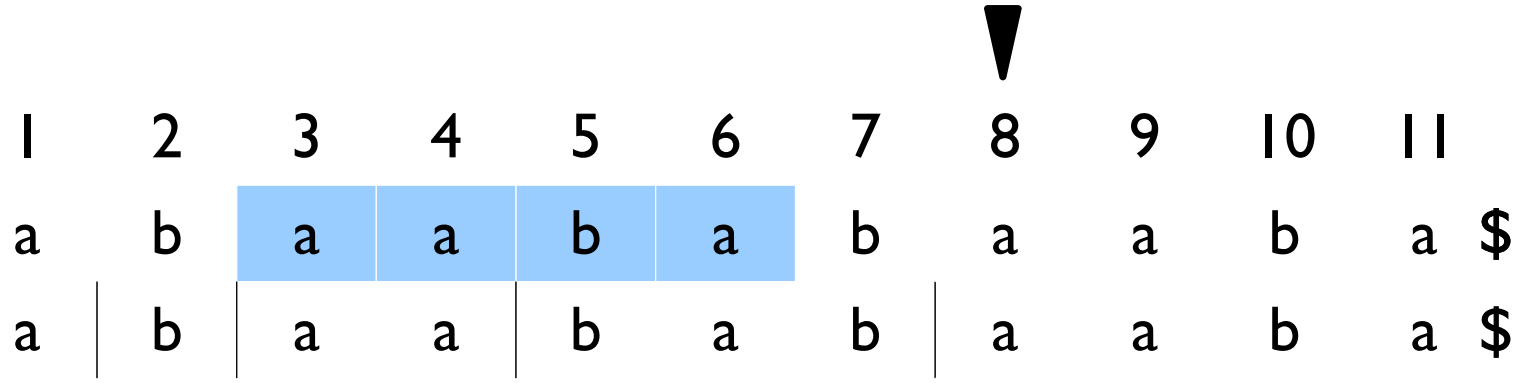


Lempel and Ziv (1977)

The Lempel-Ziv factorization (or parsing) breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then next phrase is:

- X[i..j], the shortest substring starting at i that has not occurred at any position p_i < i in X



Source = (3,6)

Compression

If our collection is highly repetitive

- LZ77 phrases will be long and so,
- z, the overall number of phrases, will be small

In every genome after the first, phrases will be very long, and broken only by differences between individuals (usually SNPs)

LZ77 is automatically (and fairly efficiently) learning the structure of the database

Pattern Matching...

Pattern Matching (read alignment)

We seek ALL the occurrences of a pattern R in a collection X

LZ77 allows us to talk about two different types of pattern occurrence

- Occurrences crossing a phrase boundary (PRIMARY)
- Occurrences wholly contained in a phrase (SECONDARY)

Strategy: find all the PRIMARY occurrences and use them and the structure of the LZ77 parse to find the SECONDARYs

Finding Primary Occurrences

Primary occurrences cross a phrase boundary...

Finding Primary Occurrences

Primary occurrences cross a phrase boundary...

Our restriction on pattern length $|R| < M$ affords us the following strategy:

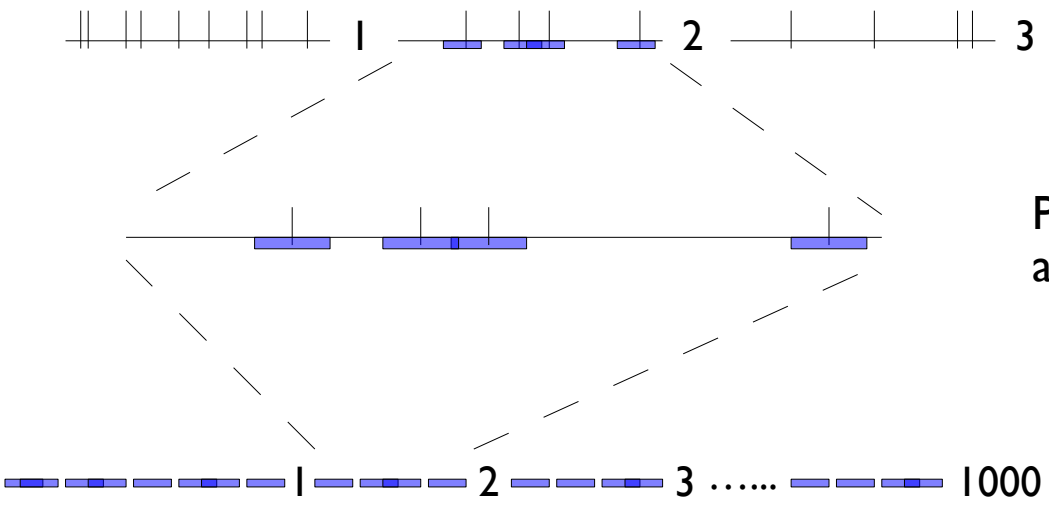
- For each phrase boundary i take the patch of $M+K$ symbols to the right and left of it in X, i.e. $X[i-M-K..i+M+K]$
- Concatenate these patches to form a *filtered string*
- Index the filtered string with a regular read aligner

Finding Primary Occurrences

Primary occurrences cross a phrase boundary...

Our restriction on pattern length |R| < M affords us the following strategy:

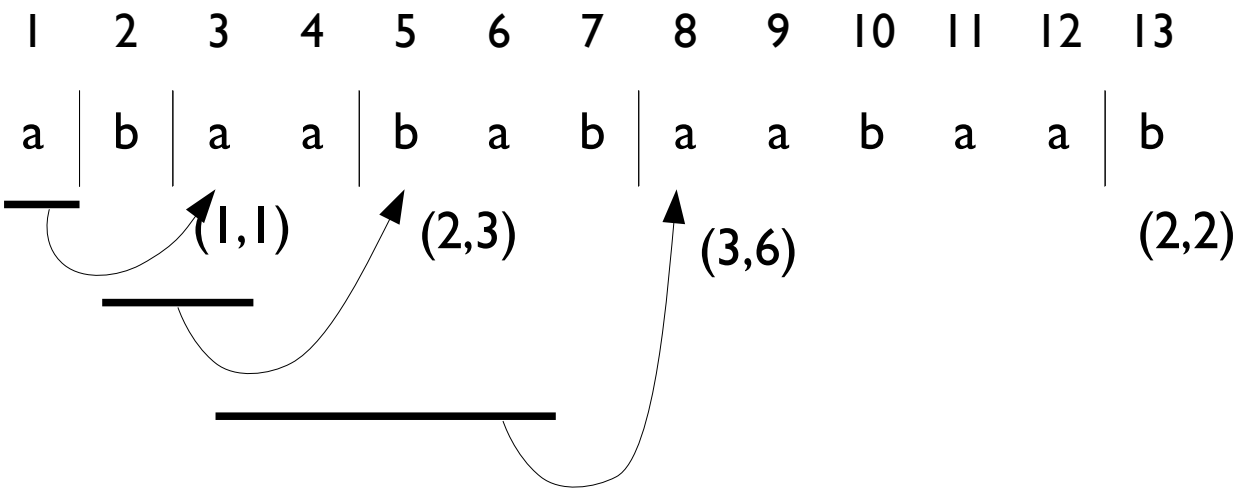
- For each phrase boundary i take the patch of M+K symbols to the right and left of it in X, i.e. X[i-M-K..i+M+K]
- Concatenate these patches to form a *filtered string*
- Index the filtered string with a regular read aligner



Secondary Occurrences...

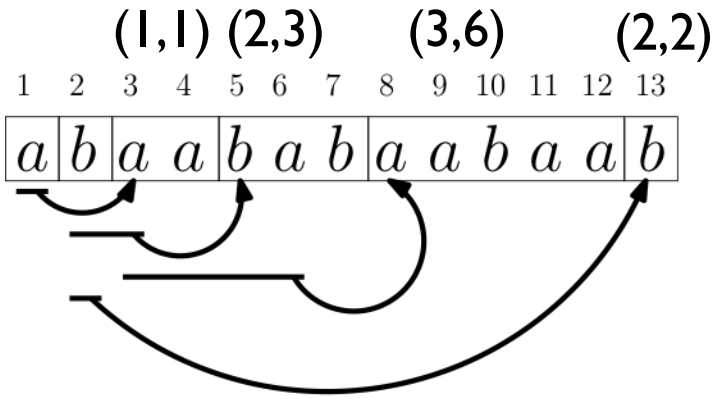
Phrase Sources

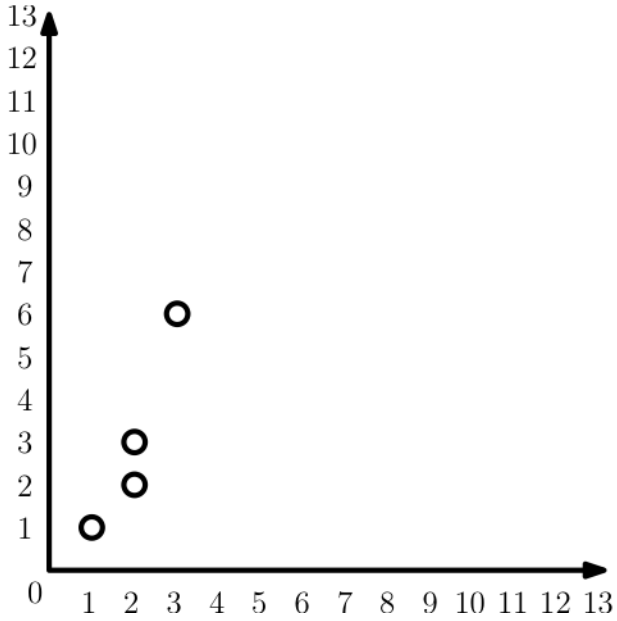
The source for an LZ phrase is a previous occurrence of it's longest repeating prefix



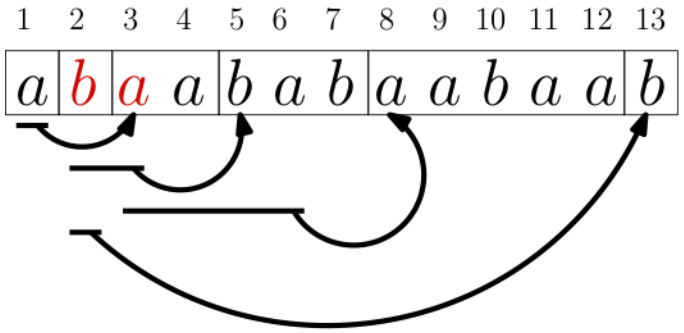
Intuition: we will use the phrase source structure to map primary occurrences forward, and so locate secondary occurrences

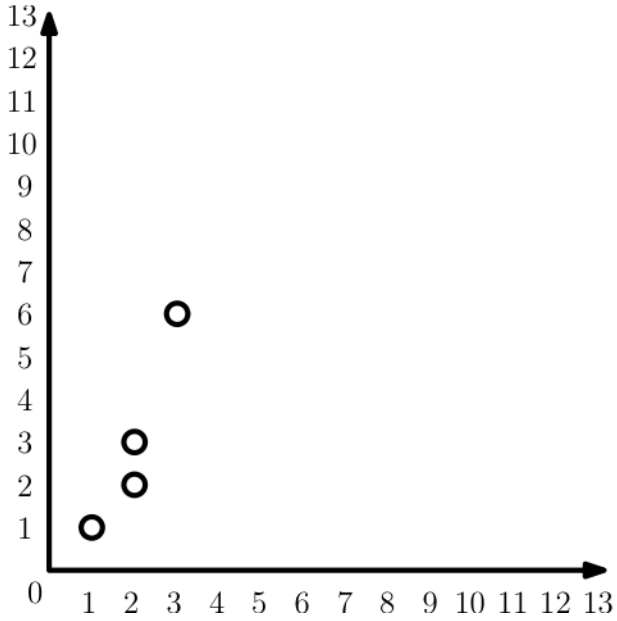
Phrase Sources on a Grid





Secondary Occurrences

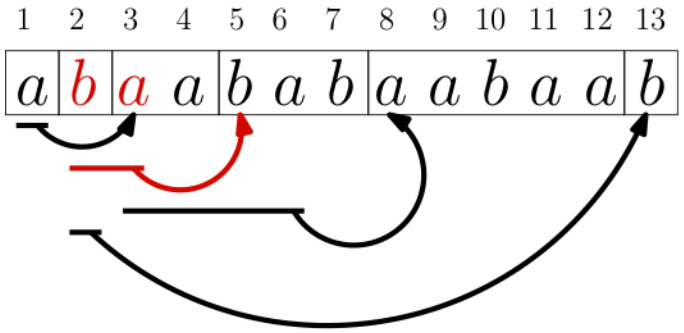


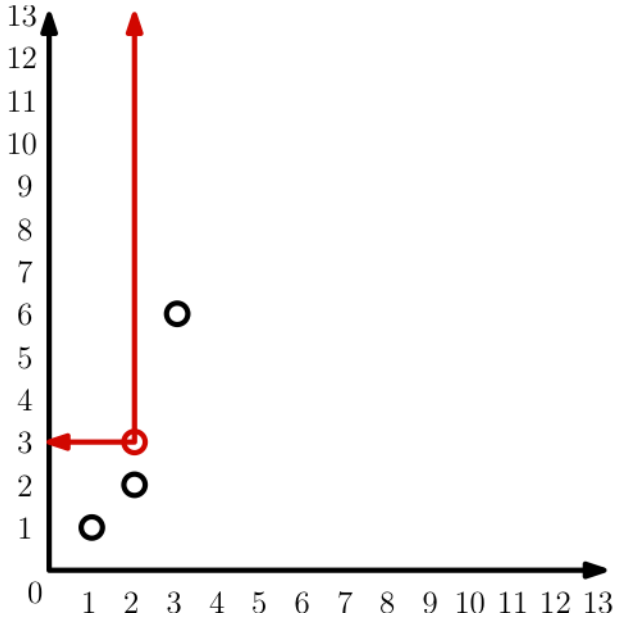


Start with a primary occurrence of *ba*

(primary because it crosses a phrase boundary)

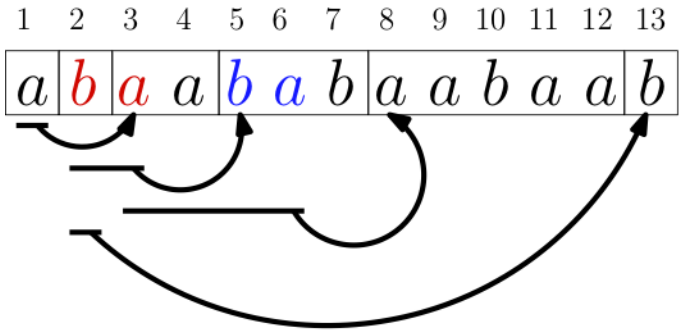
Secondary Occurrences

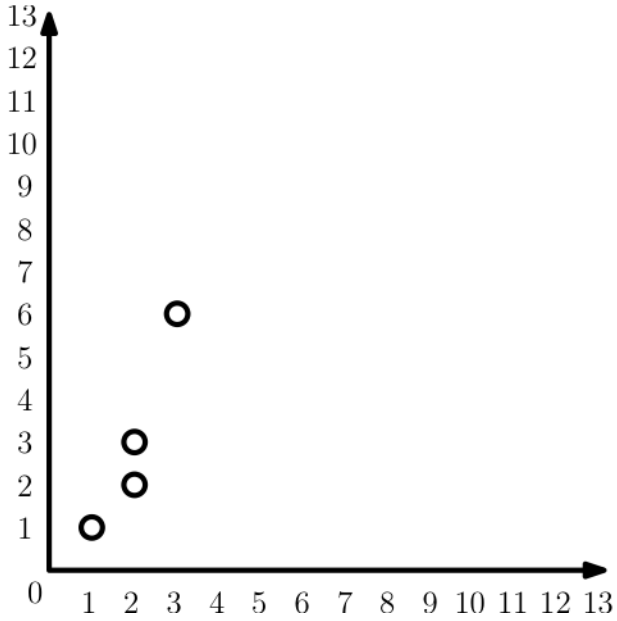




Are there any phrase sources covering this primary occurrence?

Secondary Occurrences

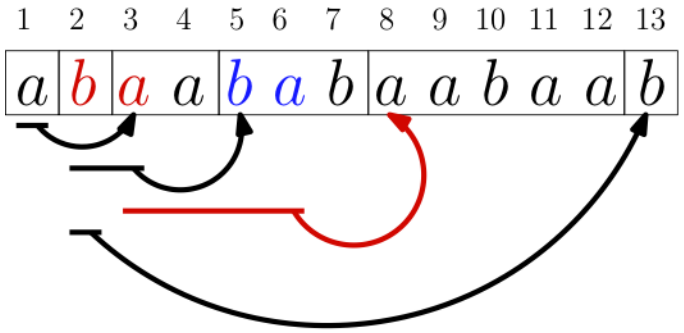


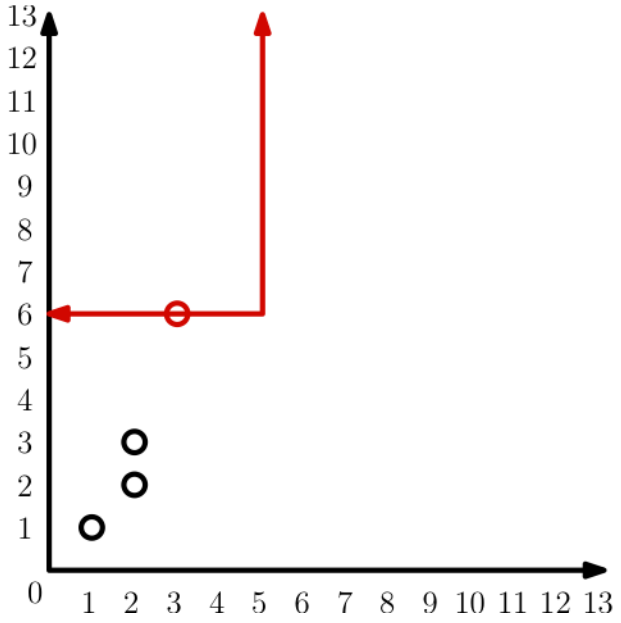


We have a secondary occurrence of *ba*

(with each point on the grid we stored the starting position of the corresponding phrase – 5 in this case)

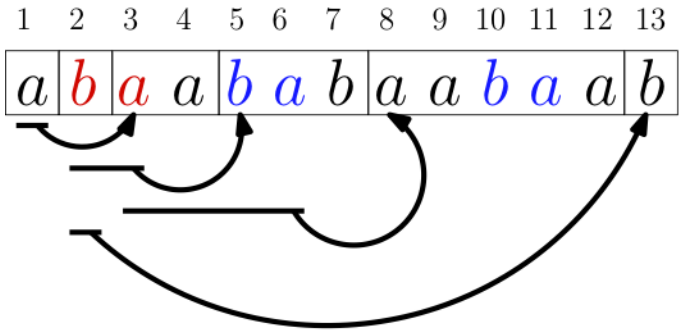
Secondary Occurrences

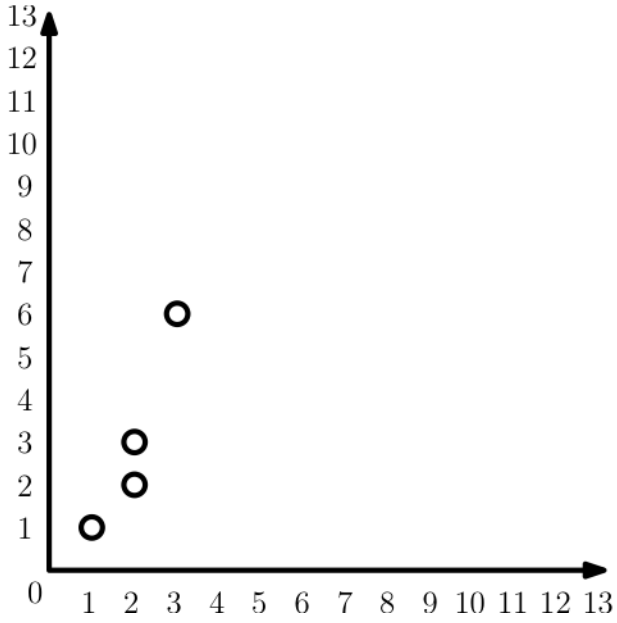




Are there phrase sources covering this secondary occurrence?

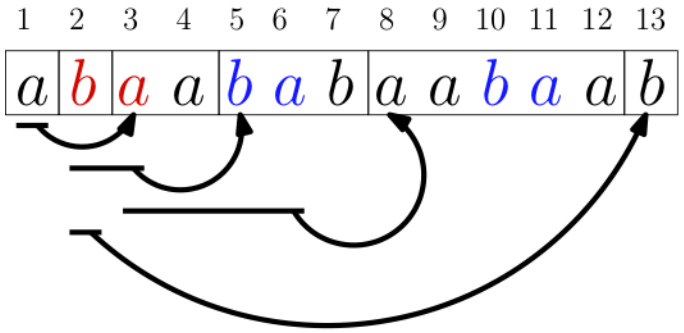
Secondary Occurrences

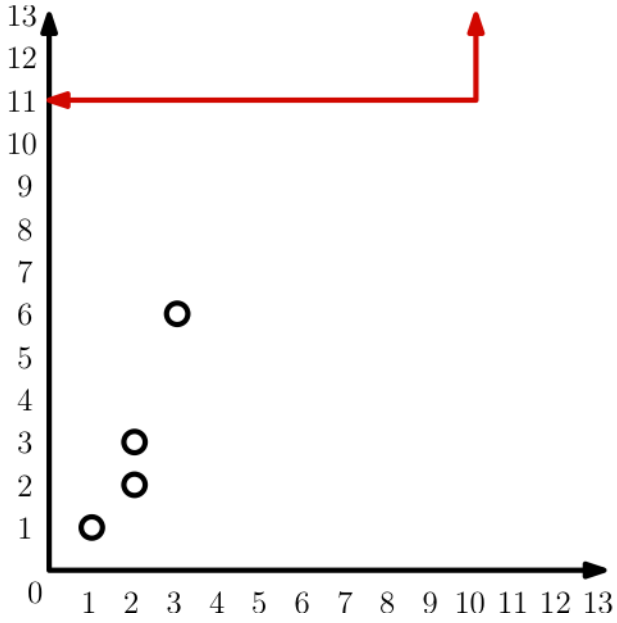




We have another secondary occurrence of *ba*

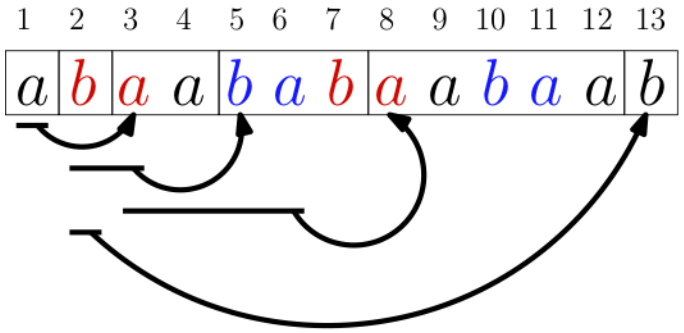
Secondary Occurrences

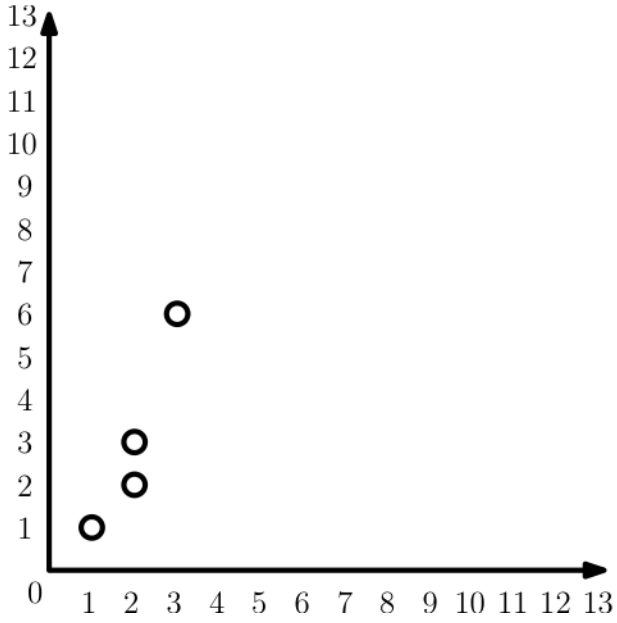




Are there phrase sources covering this secondary occurrence?

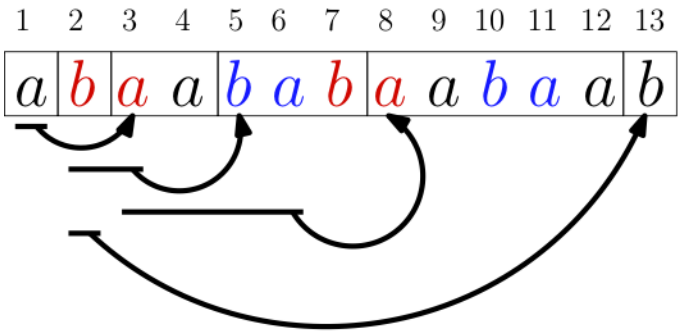
Secondary Occurrences

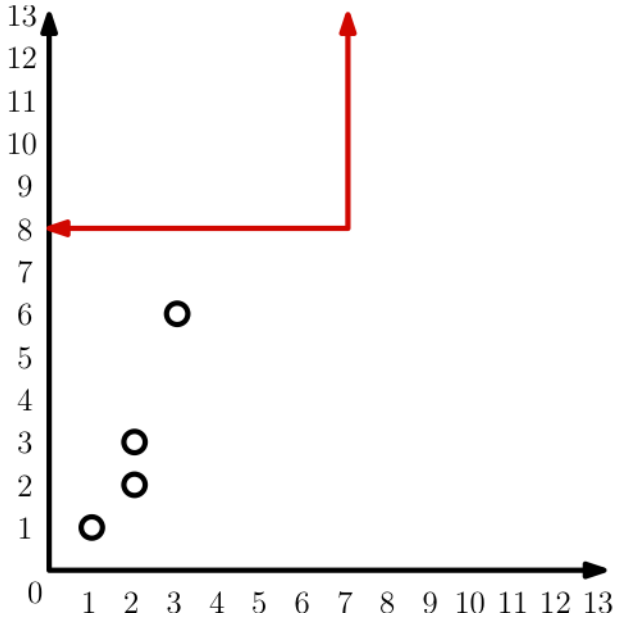




Repeat for each primary occurrence of *ba*

Secondary Occurrences





2D, 2-sided Range Reporting

Reporting secondary occurrences this way is *fast*

- O(loglogz) time per point in theory (predecessor + RMQ)
- Very fast in practice

Also space-efficient

- The grid stores z points, so we need only O(z) space
- 3z integers in practice: source start, source end, phrase start

2D, 2-sided Range Reporting

Reporting secondary occurrences this way is *fast*

- O(loglogz) time per point in theory (predecessor + RMQ)
- Very fast in practice

Also space-efficient

- The grid stores z points, so we need only O(z) space
- 3z integers in practice: source start, source end, phrase start

The structure assumes NOTHING about how we found the primaries

- We are free to use any method

Performance...

Experimental Setup

Disclaimer: these results are proof-of-concept only

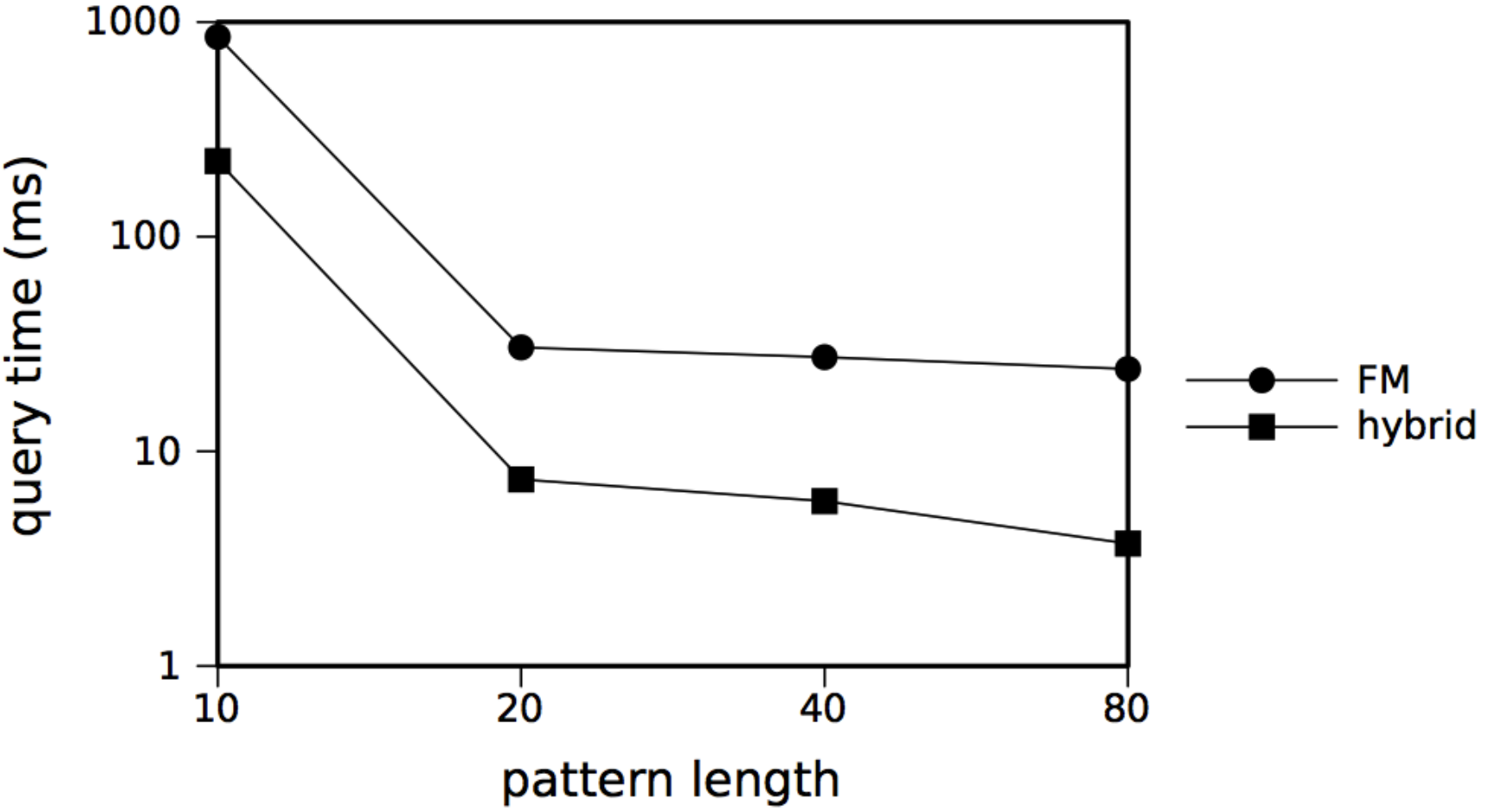
Collection: 37 individual genomes of *Saccharomyces cerevisiae*, totalling 440MB, from the Saccharomyces Genome Resequencing Project

Indexes:

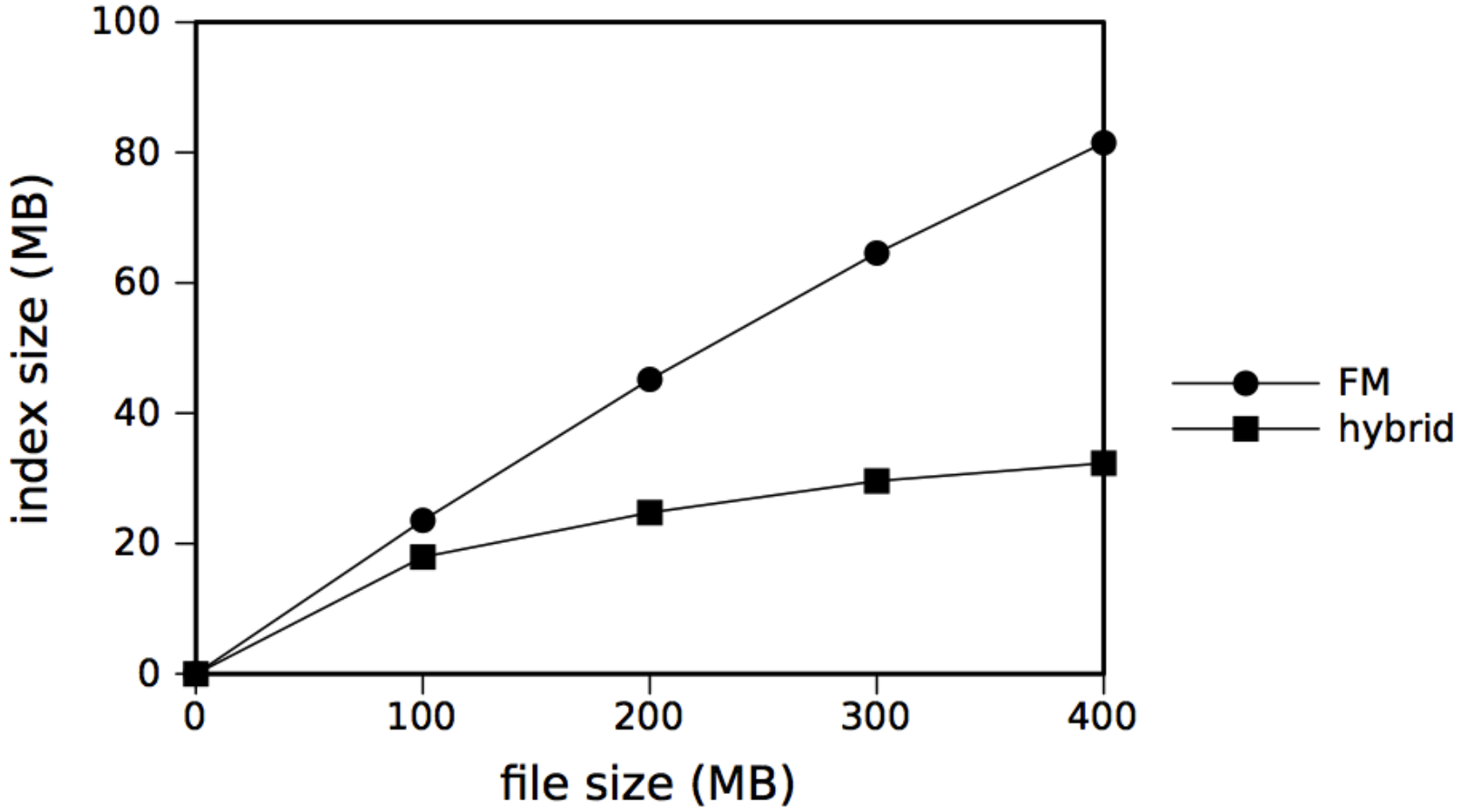
- FM: a very fast FM-index by Gog and Petri (2013)
- Hybrid: FM used on filtered text, M+K = 100

Patterns: 3000 non-unary random patterns extracted from the collections, of lengths 10, 20, 40, 80

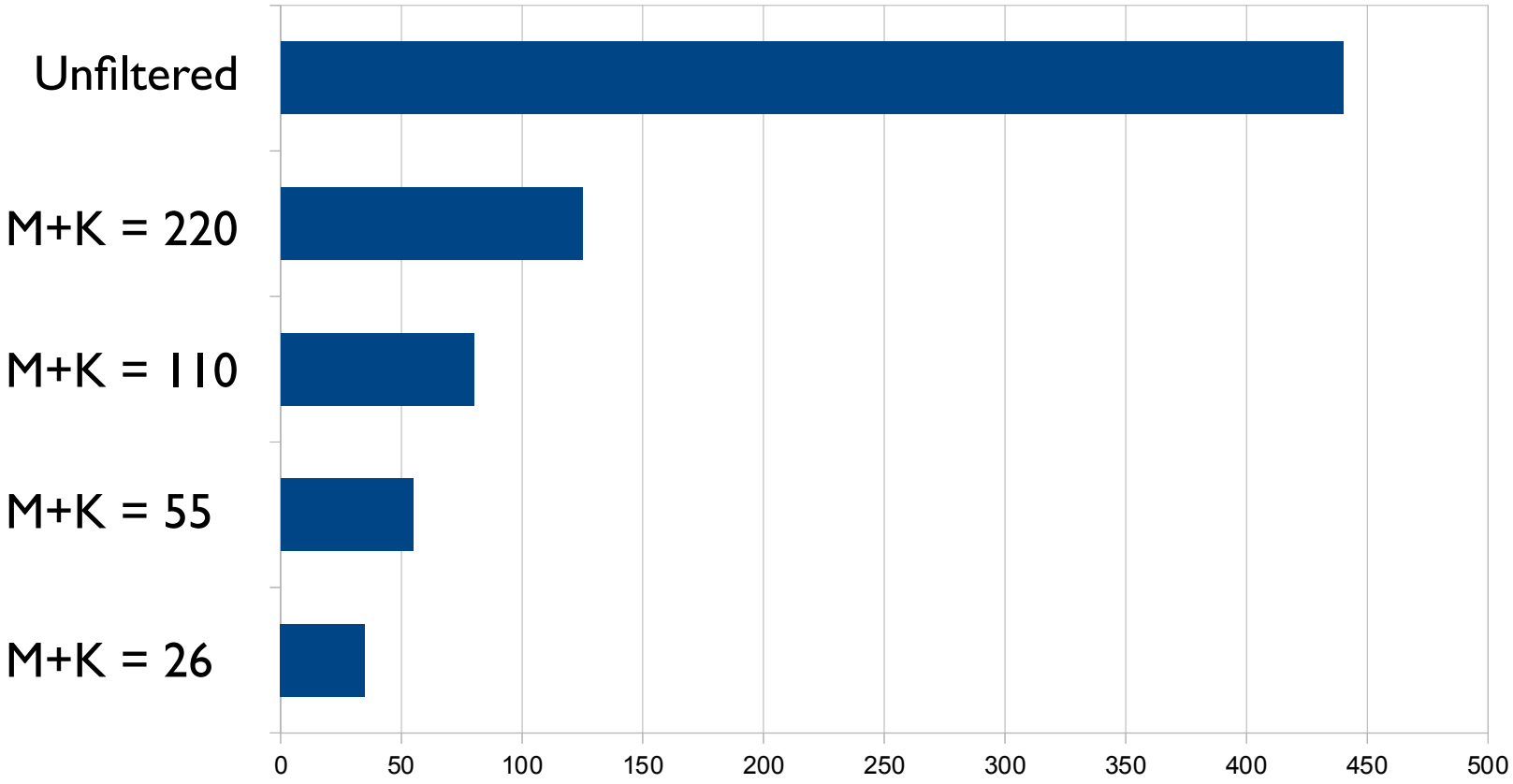
Query times


1000
100
10
1
query time (ms)
10
20
40
80
pattern length
FM
hybrid

Index Size vs. Collection Size



Filtered Text Size vs. (M+K)



Future directions...

1) Removing the restriction on M+K

Restricting M+K is right at the heart of our approach

To support longer patterns: break the pattern into multiple pieces of length M then fuse the results of each small pattern

2) Alternatives to LZ77 parsing

LZ77 is very general – assumes nothing about collection structure. This has advantages.

If we remove the blindfold, we can exploit collection structure in (at least) two ways...

RLZ: only allow sources to be in the first genome

- Construction (parsing) is easier, index probably bigger

Alignment-based parsing: multiple alignment informs parsing

- Smaller index, much slower to construct

3) Parsing/Construction Bottleneck

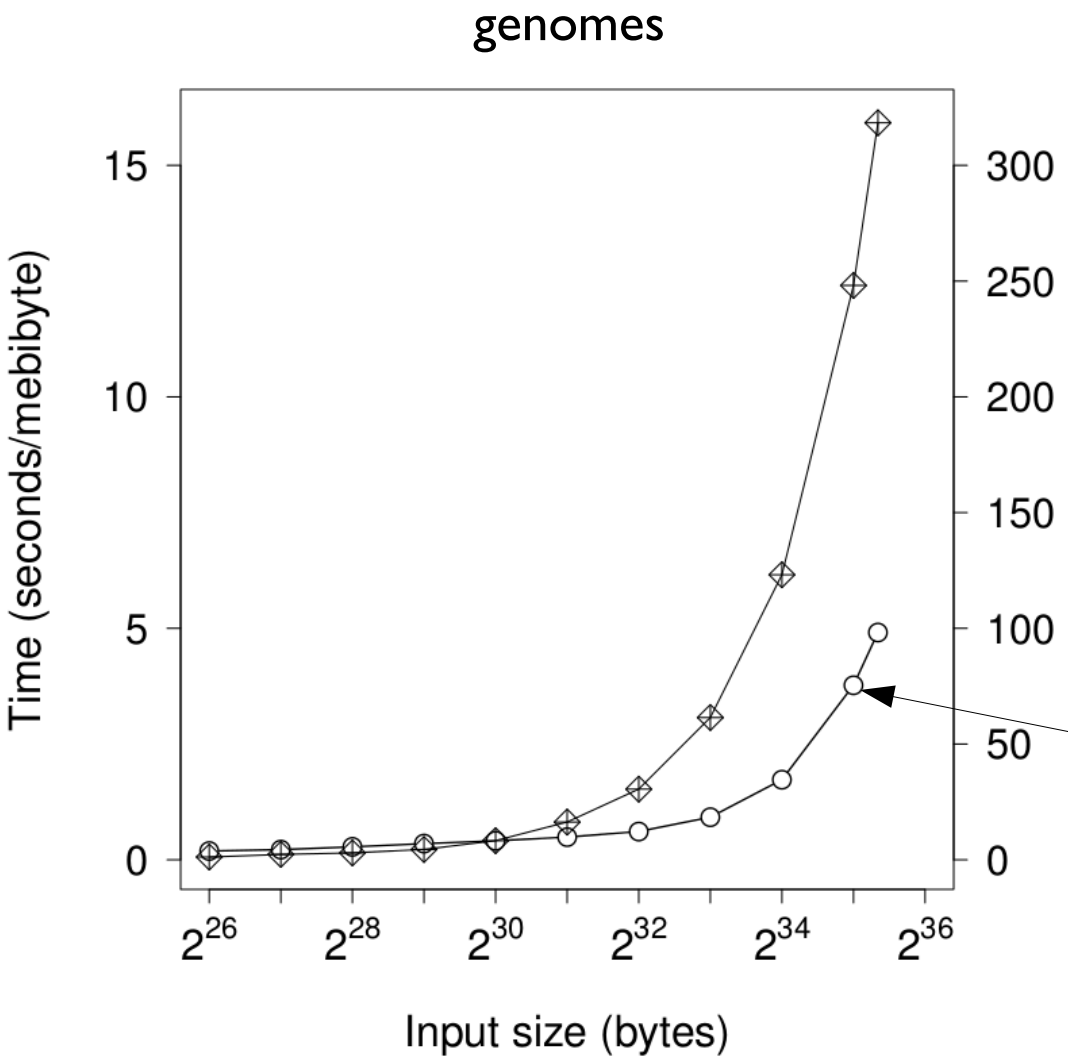
Computing LZ77 for really large inputs has been a long-standing open problem...

- ...and is the main reason experiments above were with only 440MB

Some breakthroughs here recently

- Joint work with Juha Karkkainen and Dominik Kempa
- (to be submitted to ALENEX next week)

4) Construction – external memory LZ parsing



32Gb input
(40 human genomes)
4Gb memory
<5 hours

Conclusion

- Hybrid indexing is a generic way to scale read aligners (or any other pattern matching index)
- Only restriction is an upperbound on the pattern length, M and the number of errors/edits allowed, K
- Code + preprint available:
  - puglisi@cs.helsinki.fi

Fin