ALGODAN Algorithmic Data Analysis

De novo sequencing and gene regulatory motifs

Esko Ukkonen

Aalto University

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

I: *De novo* sequencing of *Melitaea cinxia*

- CoE Algodan
  - Esko Ukkonen, Veli Mäkinen, Leena Salmela, Niko Välimäki
- CoE on Metapopulation Research, UH
  - **Ilkka Hanski**, Rainer Lehtonen, Virpi Ahola, Pasi Rastas
- Institute of Biotechnology, UH
  - Petri Auvinen, Lars Paulin, Panu Somervuo
  - Liisa Holm, Patrik Koskinen

Genome assembly problem

Photo: Niclas Fritzén

Sequencing machine

Reads:
TAAATTGACCATAAAT
AACGGATCGGGACAC
ATCATAATCCAGCGAAC
ACTCTCTAAATTGACC
AATTGACCATAAATCATA
TCGGGACACAAATA
AGCGAACGGATCGG

ACTCTCTAAATTGACC ATCATAATCCAGCGAAC TCGGGACACAAATA
TAAATTGACCATAAAT AGCGAACGGATCGG
AATTGACCATAAATCATA AACGGATCGGGACAC

ACTCTCTAAATTGACCATAAATCATAATCCAGCGAACGGATCGGGACACAAATA

Why to sequence this butterfly?

- Model species in metapopulation and eco-evolutionary research
- Long line of research on this butterfly by the group of Prof. Ilkka Hanski

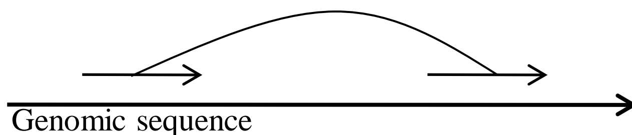
Photo: Christian Fischer

Overview of the data

	454	SOLiD	Illumina	PacBio
Reads	12 million	210 million	349 million	2.7 million
Read length	400-800 bp	50 bp	75-150 bp	Avg 2700 bp Max 23 kbp
Errors	Indels	Mismatches	Mismatches	Indels
Paired end	-	-	460 bp,710 bp	-
Mate pairs	7 kbp, 17 kbp	2-5 kbp	1-3 kbp	-
Other	Mostly single end	Color coded	-	High error rate

Total size of data: 50 Gbp
Length of the genome: 350 Mbp

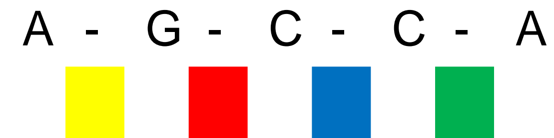
Mate pairs:



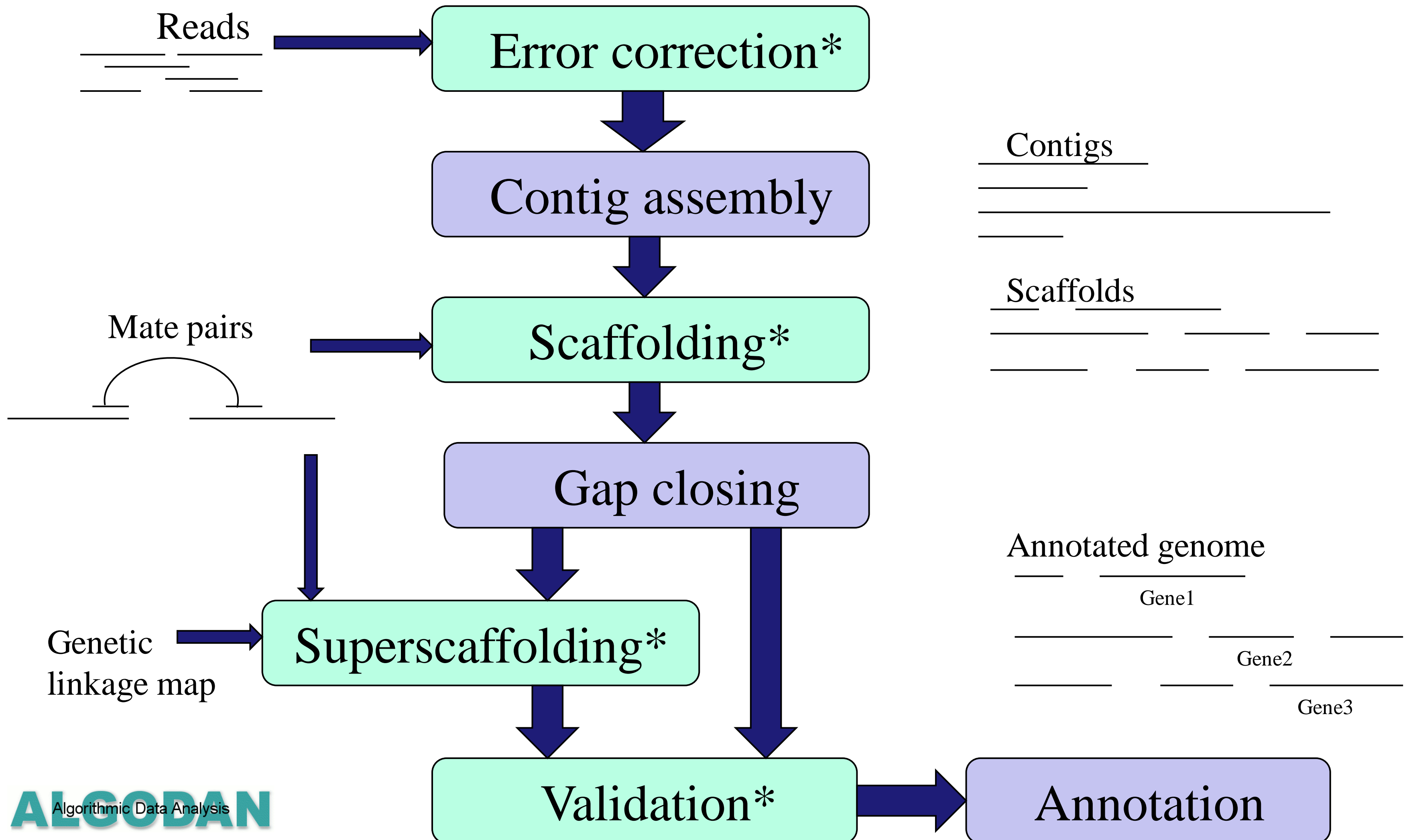
SOLiD color coding:

A C G T

A
C
G
T



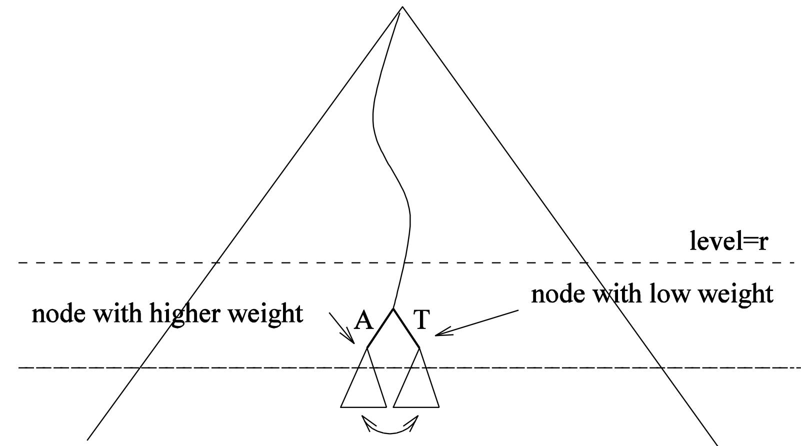
Genome assembly workflow



ALGODAN Algorithmic Data Analysis

New methods: correcting sequencing errors

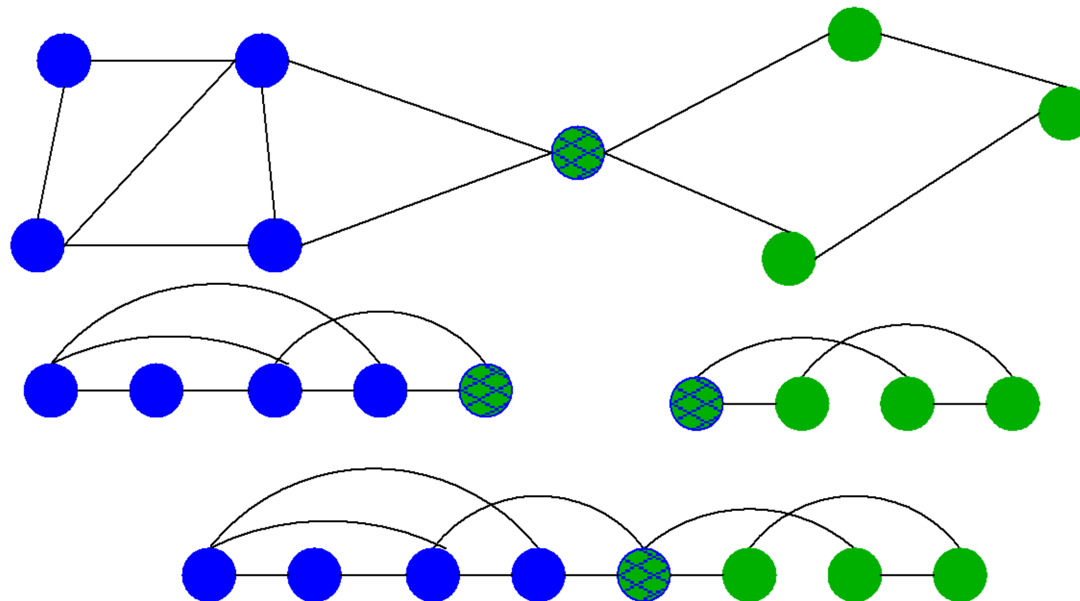
- HybridSHREC
  - Build generalized suffix trie of the reads
  - Detect and correct errors by examining the structure of the trie
- CORAL
  - Build k-mer index of the reads
  - Build multiple alignments of reads sharing k-mers



A	C	G	G	A	A	-	C	C	G
A	C	G	G	A	A				
	C	-	G	A	A	A	C	C	
		G	G	A	A	-	C	G	
				A	A	-	C	C	G

New methods: MIP Scaffolder

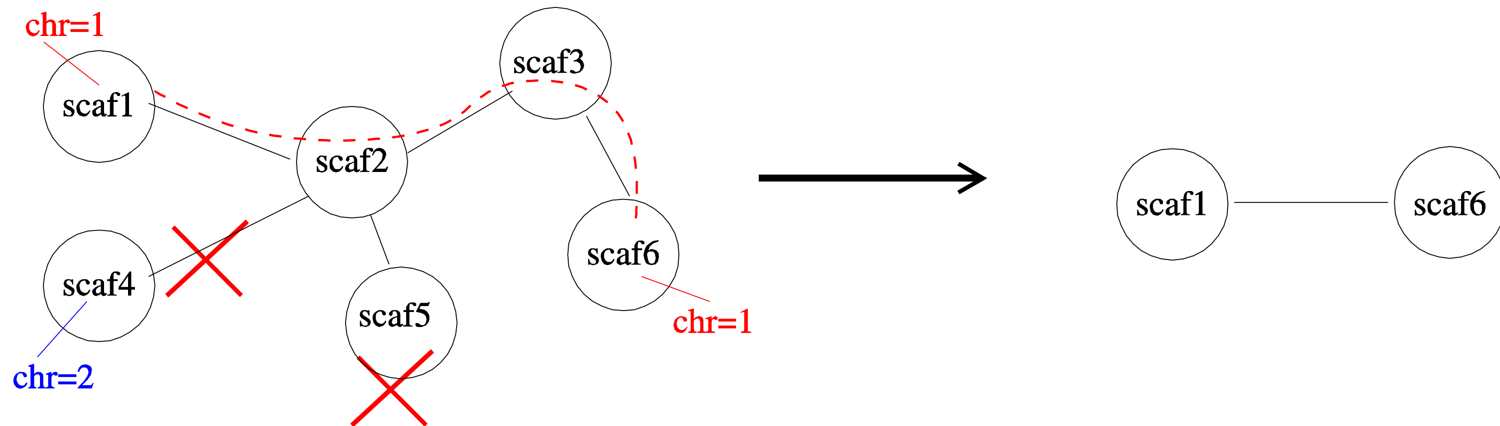
- Goal: Use mate pairs to organize contigs into linear scaffolds
- Partition the problem into small subproblems of restricted size
- Solve each subproblem exactly with mixed integer programming
- Combine the solutions of the subproblems



ALGODAN Algorithmic Data Analysis

New methods: superscaffolding

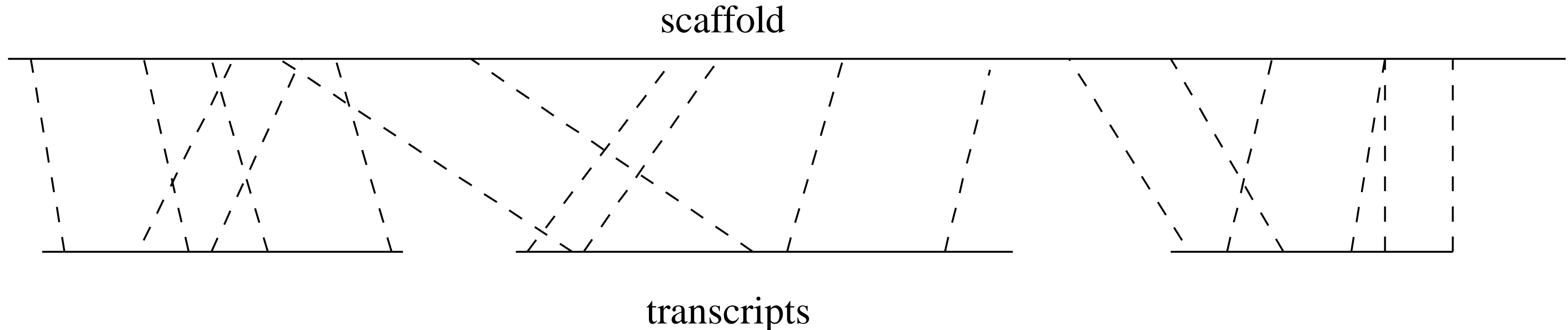
- Genetic linkage map assigns some scaffolds to chromosomes
- We use the genetic linkage map and mate pairs to further connect scaffolds into superscaffolds
- Find paths of mate pair links between scaffolds in the same chromosome



ALGODAN Algorithmic Data Analysis

New methods: validation

- Find local maximal approximate matches of transcripts and scaffolds



- For each transcript find maximal colinear chains of the above matches
- The more transcripts can be aligned, the more complete the genome is.

ALGODAN Algorithmic Data Analysis

Statistics of the draft genome

	Number of contigs/scaffolds	N50	Total length
Contigs	49,851	13,489	360,975,554
Scaffolds	8,262	119,328	389,896,394
Superscaffolds	1,453	330,752	282,503,348
Final	**6,299**	**258,308**	**393,309,151**

N50:
The total length of contigs longer than the N50 statistic is at least half the length of the whole contig collection.

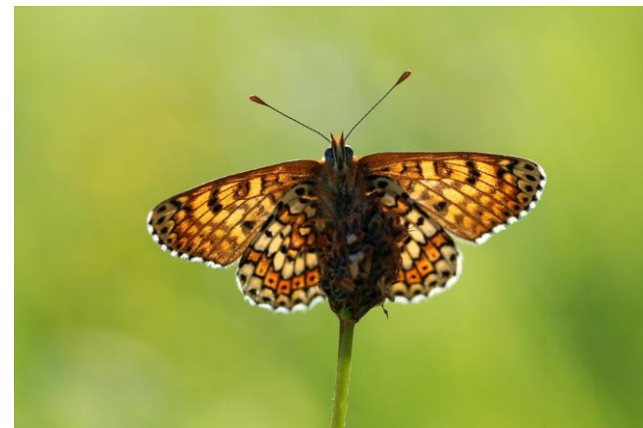
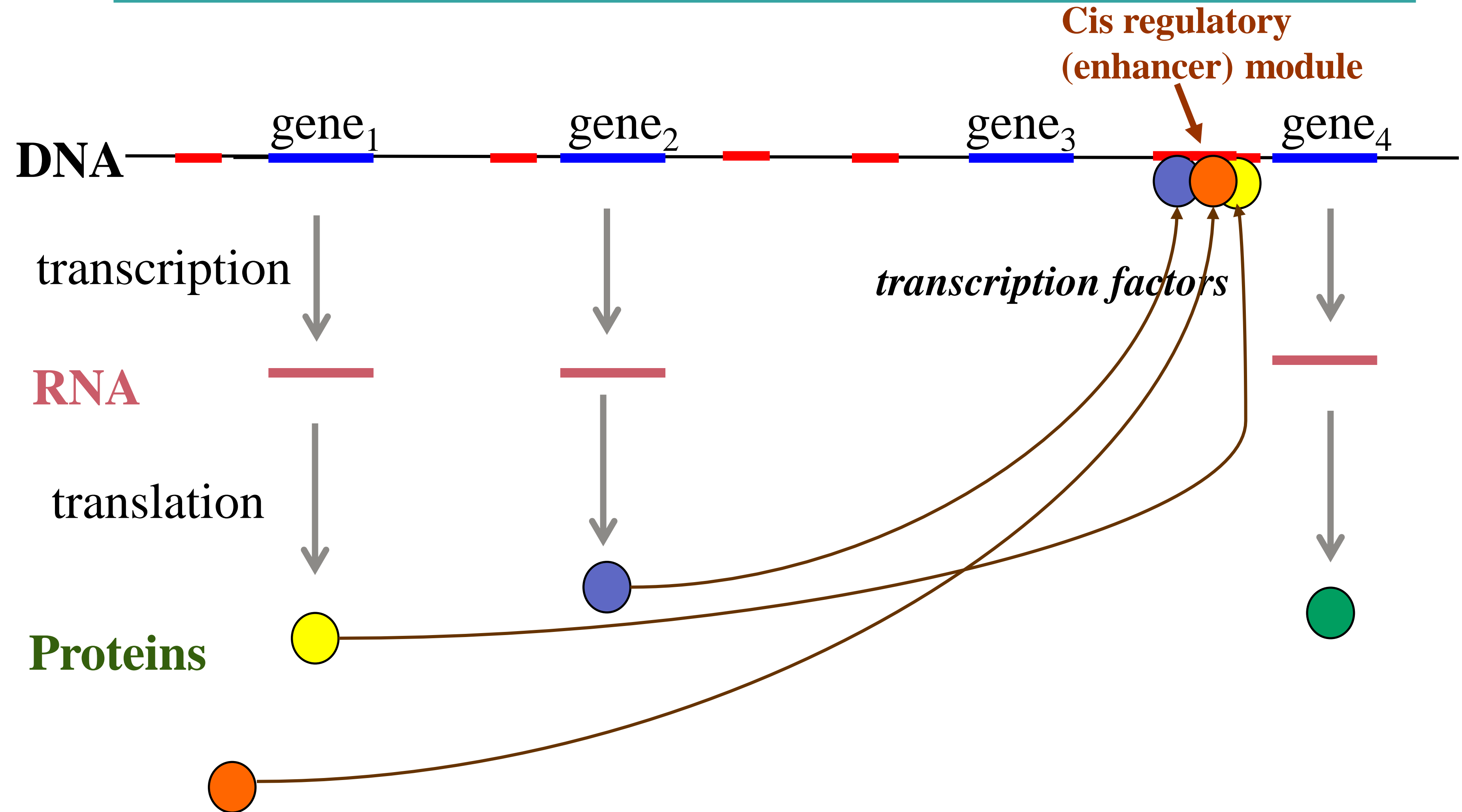


Photo: Gilles San Martin

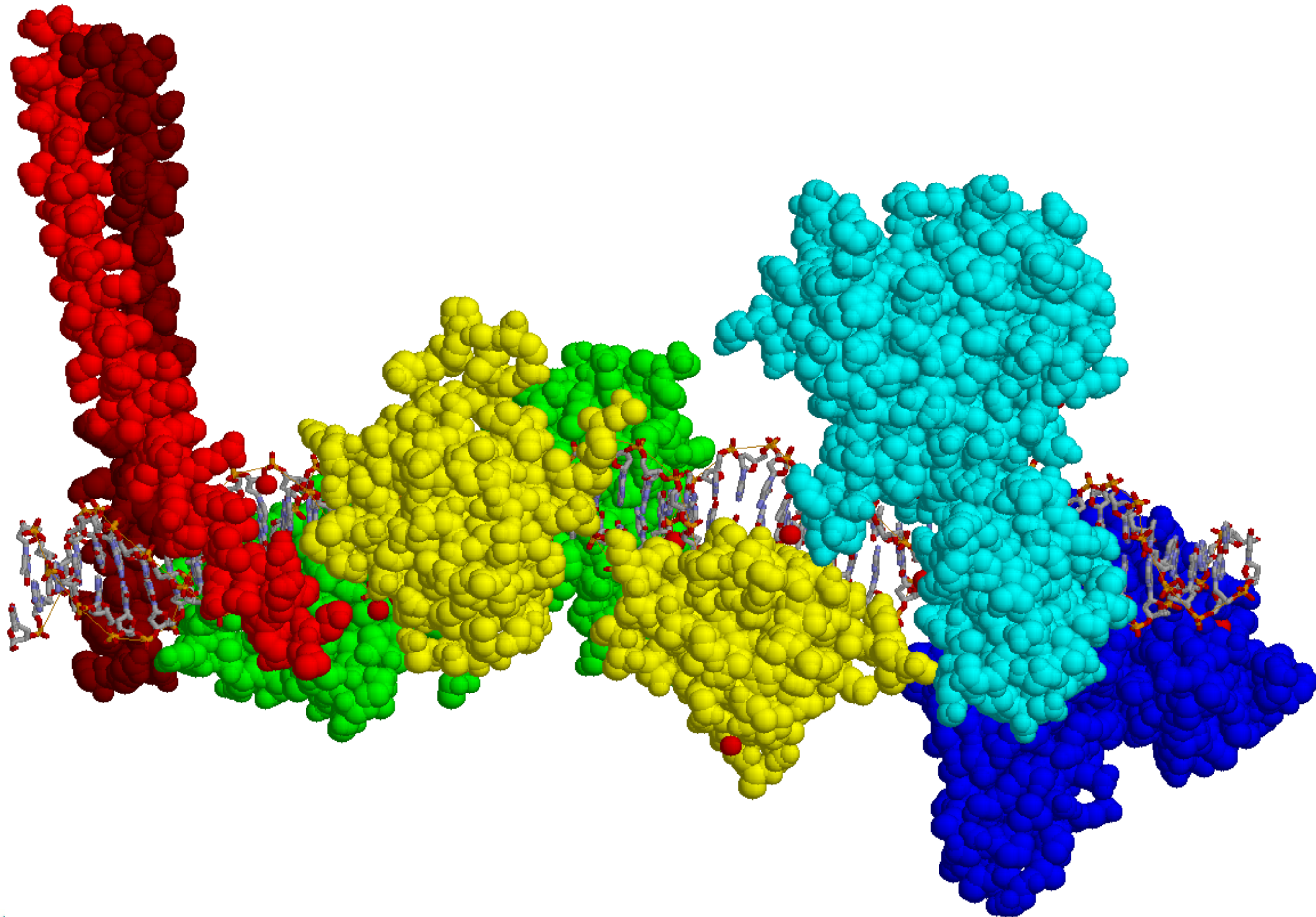
II: Genome-scale prediction of gene regulatory motifs

- CoE Algodan
  - Esko Ukkonen, Jarkko Toivonen, Teemu Kivioja
- CoE on Translational Genome-Scale Biology, UH
  - **Jussi Taipale** (Karolinska Institutet), Lauri Aaltonen, Arttu Jolma, Kimmo Palin, ...
- EU consortium SYSCOL

Gene regulatory modules (cis-regulation)



Regulatory module

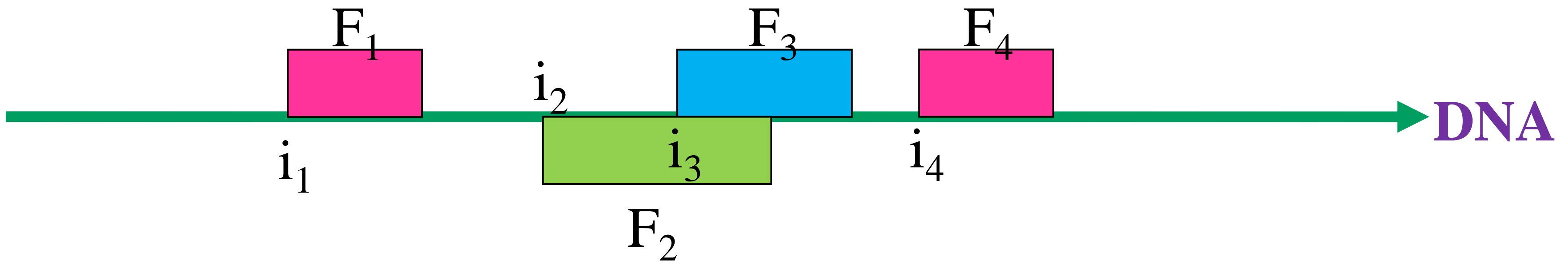


Characterization of a regulatory module?

- A regulatory module (*cis-regulatory module*) is a collection of TF binding sites on DNA; no precise definition available

- properties of a module:
  - consists of several <u>good</u> binding sites of TFs
  - the sites are spatially <u>clustered</u> together
  - the pattern of sites is <u>conserved</u>

Module structure

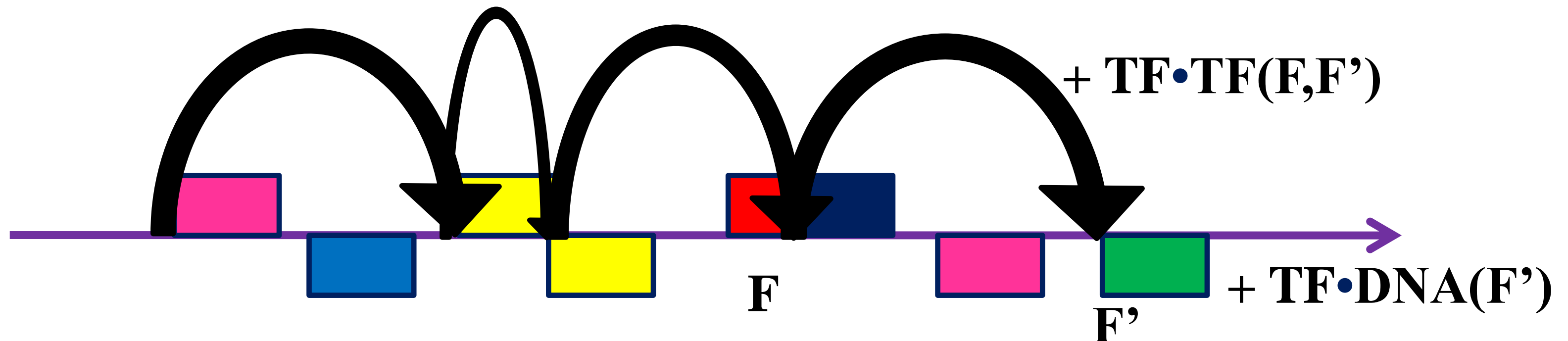


$$(i_1, F_1, \mathrm{Nor}), (i_2, F_2, \mathrm{Rev}), (i_3, F_3, \mathrm{Nor}), (i_4, F_4 = F_1, \mathrm{Nor})$$

High-scoring modules

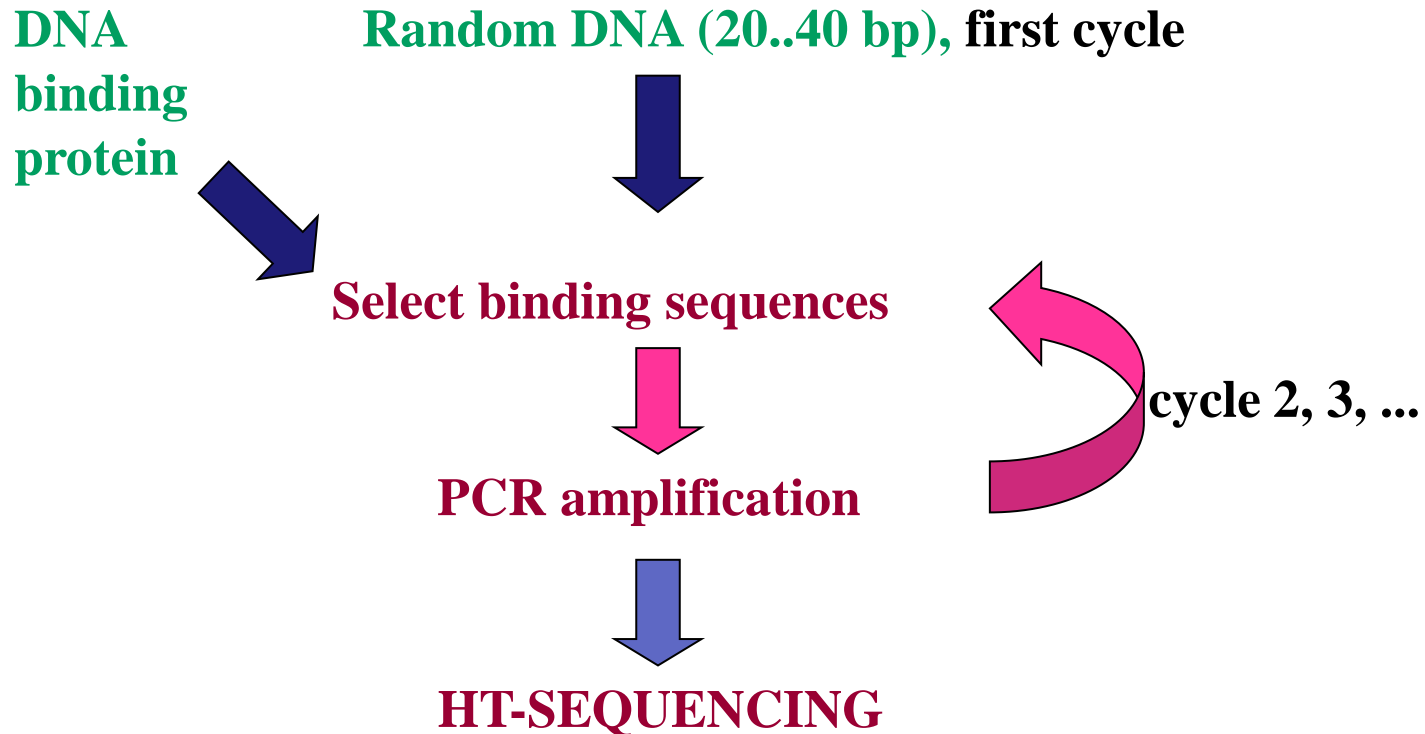
- Find the highest scoring module of *s*
  - Highest scoring subsequence -> Smith-Waterman style dynamic programming (Karlin-Altschul conditions)

$$W(i,F,o) = \max_{i'<i,F',o'} \begin{cases} W(i',F',o') + TF{\bullet}DNA(i,F,o) + TF{\bullet}TF(F,o,F',o',i-i') \\ TF{\bullet}DNA(i,F,o) \end{cases}$$



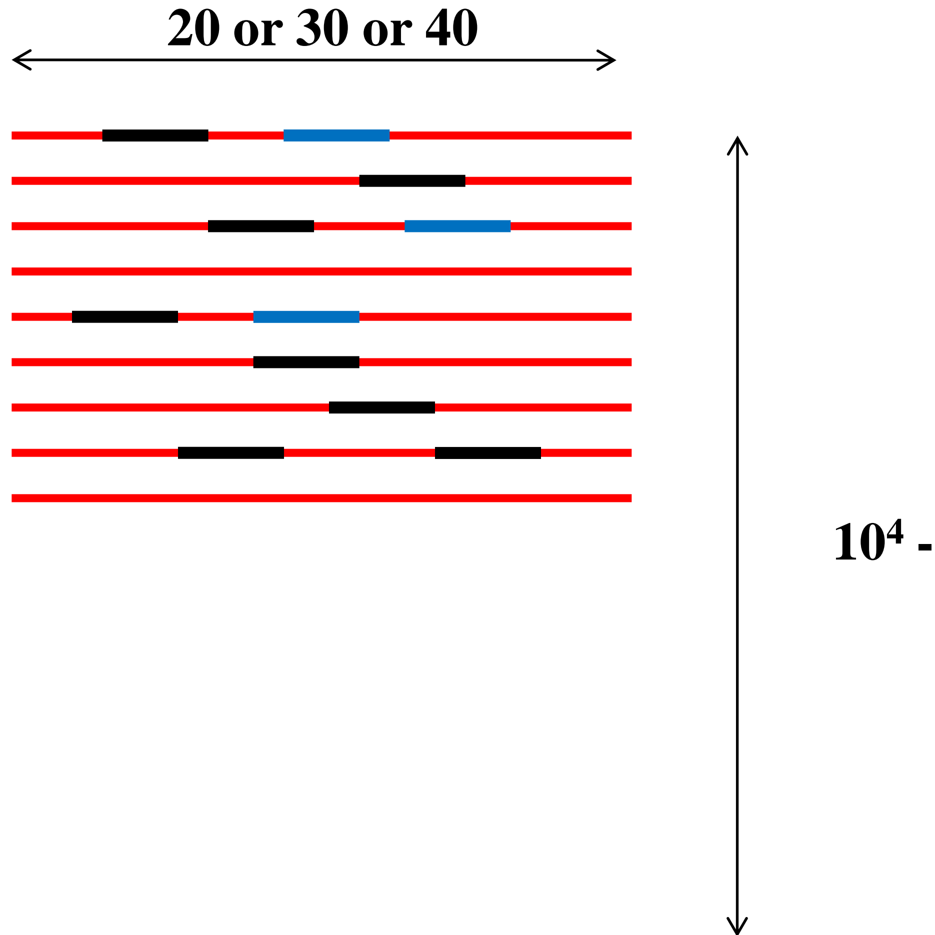
'Big' Data from SELEX protocol

SELEX = Systematic Evolution of Ligands by EXponential enrichment (Turk & Gold 1990)



A. Jolma, T. Kivioja et al.: Multiplexed massively parallel SELEX …, *Genome Res.* 2010

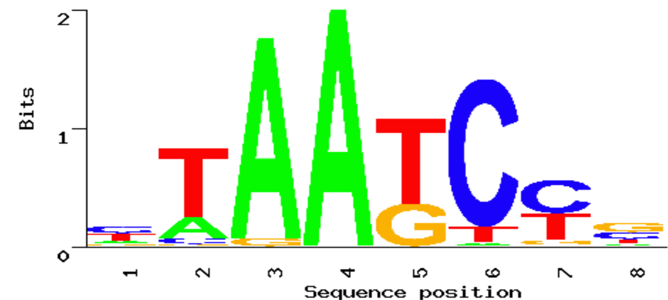
Find distribution of TF sites and their distances



10^4 -

TF-DNA interactions

- (classic) Position Weight Matrix (PWM):
  - Position-specific multinomial distribution of k-mers
  - k-mer positions independent of each other
  - alignment-based construction
  - New multi-PWM models

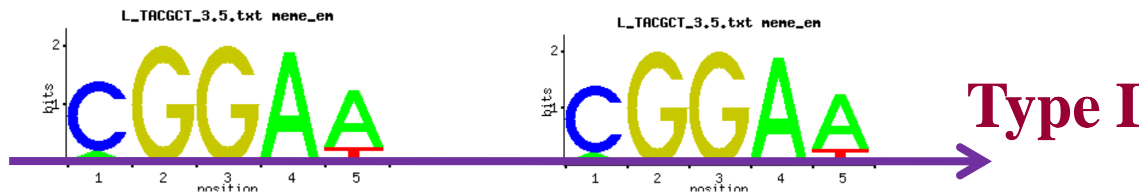


❑ A Jolma, J. Yan, Th. Whithington, J. Toivonen et al: DNA-Binding Specificities of Human Transcription Factors. *Cell* (2013)

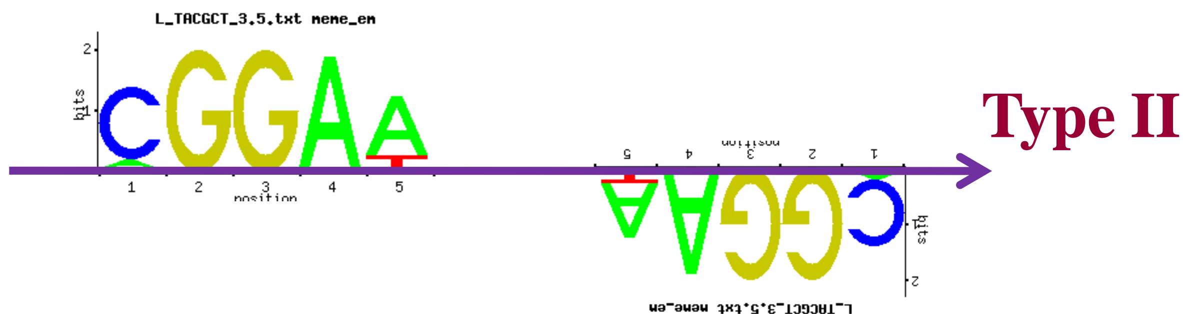
❑ C Pizzi, P Rastas & E Ukkonen: Finding Significant Matches of Position Weight Matrices in Linear Time. *IEEE/ACM Trans. Comput. Biology Bioinform* (2011)

❑ J Korhonen et al: MOODS: … , *Bioinformatics* (2010)

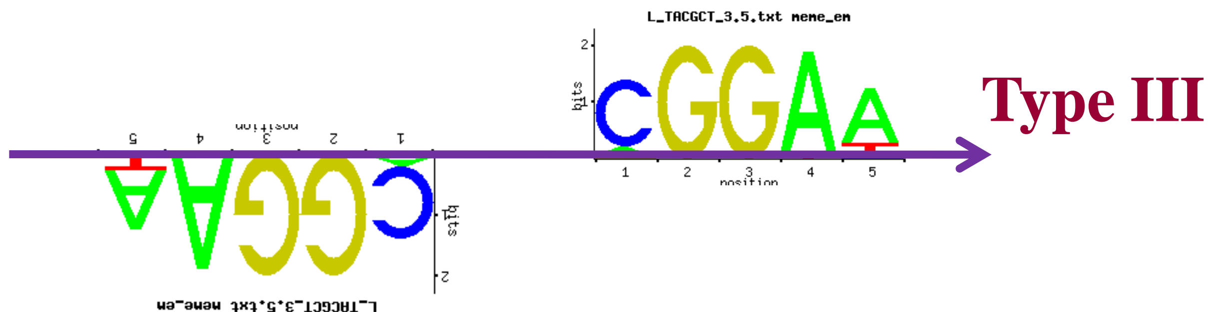
TF-TF interactions: Example dimer FLI/ERG1 - FLI/ERG1



Type I



Type II

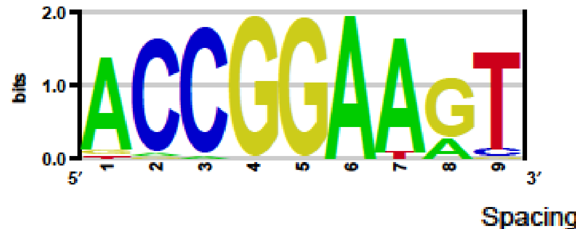


Type III

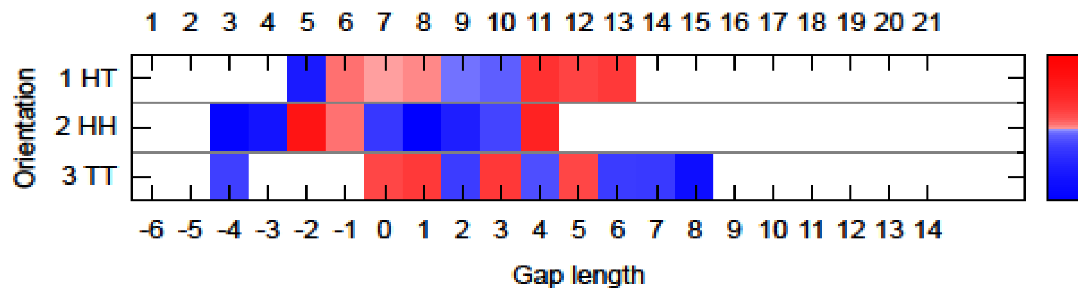
ALGODAN Algorithmic Data Analysis

New co-operative binding model & motif scanner

PWM
(EWS/FLI)

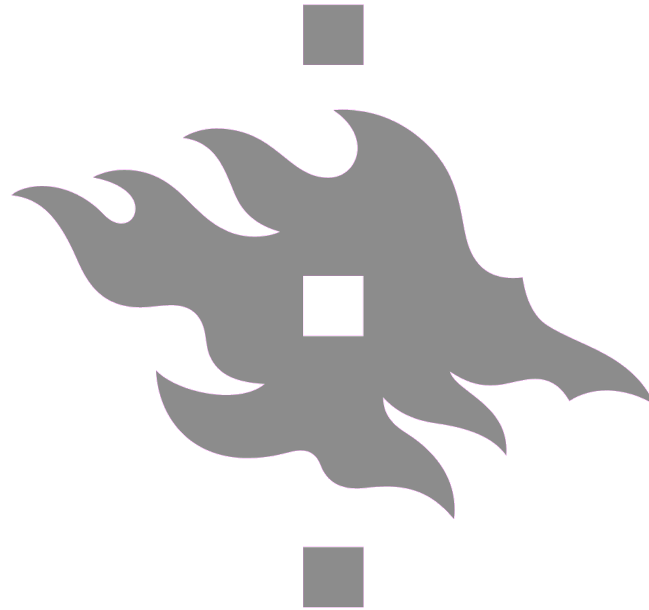


COB



-> New motif 'scanner': PWM-models + co-operative binding models for pairs of TFs + nucleosome binding model

Scalable Indexing of Highly Repetitive Data



Travis Gagie, Juha Kärkkäinen, Dominik Kempa

Simon J. Puglisi

ALGODAN

Algorithmic Data Analysis

Indexed Pattern Matching

Given a collection of strings T[1,n], build a data structure on T so that later, given some pattern P[1,m], we can quickly find all the occurrences of P in T.

We would also like:

- The index to be small ~ the size of T when it is compressed
- To be able to find approximate matches of P (up to k errors)

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)

Indexing Highly Repetitive Data

There are many indexes for approximate pattern matching in regular collections, but they don't scale well to massive, highly repetitive collections

---

*suffix tree, suffix array, FM-index, inverted file.

Aim (of this work)

Find a way to scale current indexes to highly repetitive collections that is **independent of the index itself**.

Choose an index (your favorite index); we provide an algorithmic tool to make it work for repetitive collections.

One restriction...

We will cap – at index construction time –

- Maximum pattern length m, and
- Maximum number of errors, k

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

Two Algorithmic Tools

Our index is based on two main algorithmic tools…

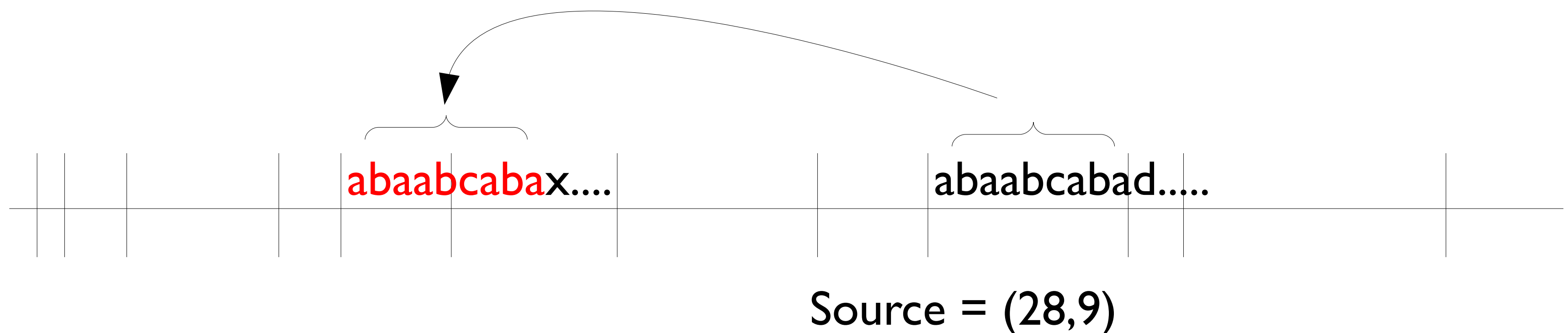
- LZ77 parsing (or factorization)
  - Widely used in data compression (gzip and 7zip)
  - We use it for compression AND pattern matching
- 2-dimensional, 2-sided range reporting
  - A notion from computational geometry

Lempel-Ziv Parsing

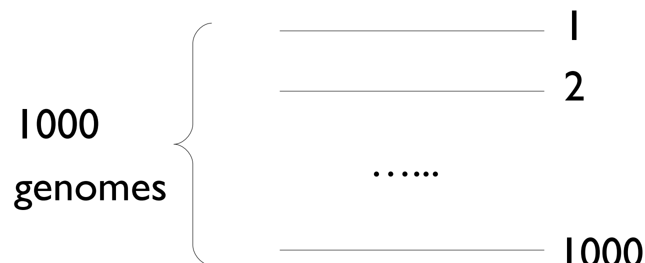
The Lempel-Ziv parsing greedily breaks a string X of n symbols into z phrases.

Each phrase occurs somewhere prior in the string.

Store two integers for each phrase, indicating position & length of prior occurrence (called the source).



Input



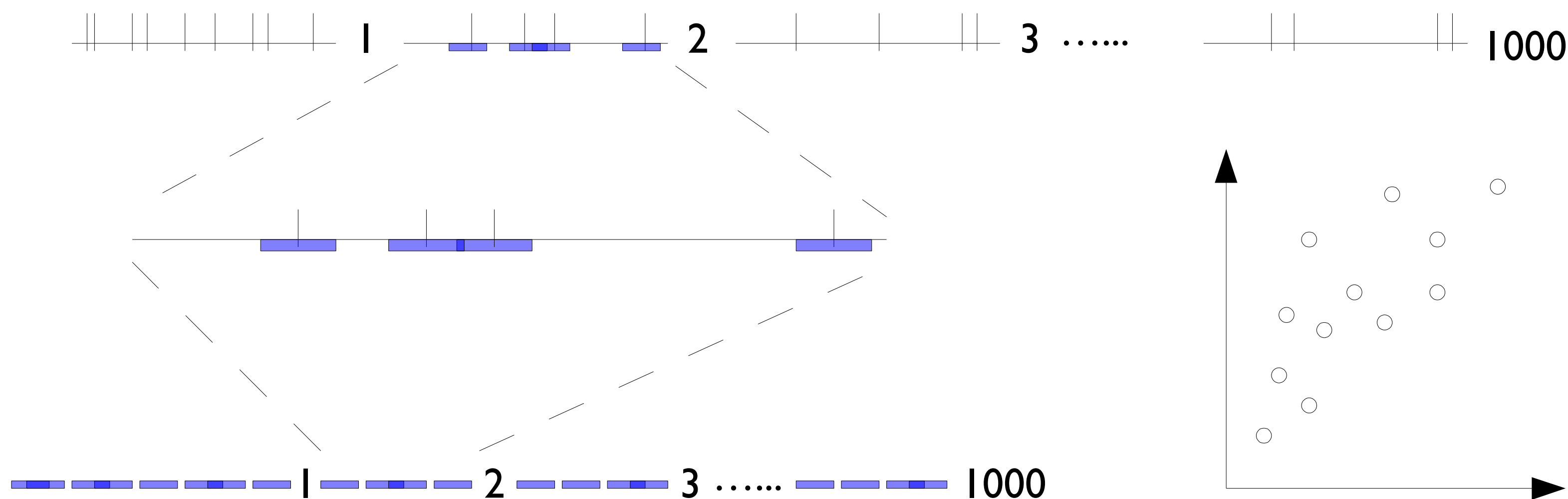
M : upper bound on read length; e.g. M = 100

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

Asymptotics

Space: $O(z\log(n/z))$

- z is the size of the compressed text
- (modest increase over the compressed size)

Pattern Matching: $O(m\log m + r\log\log n)$

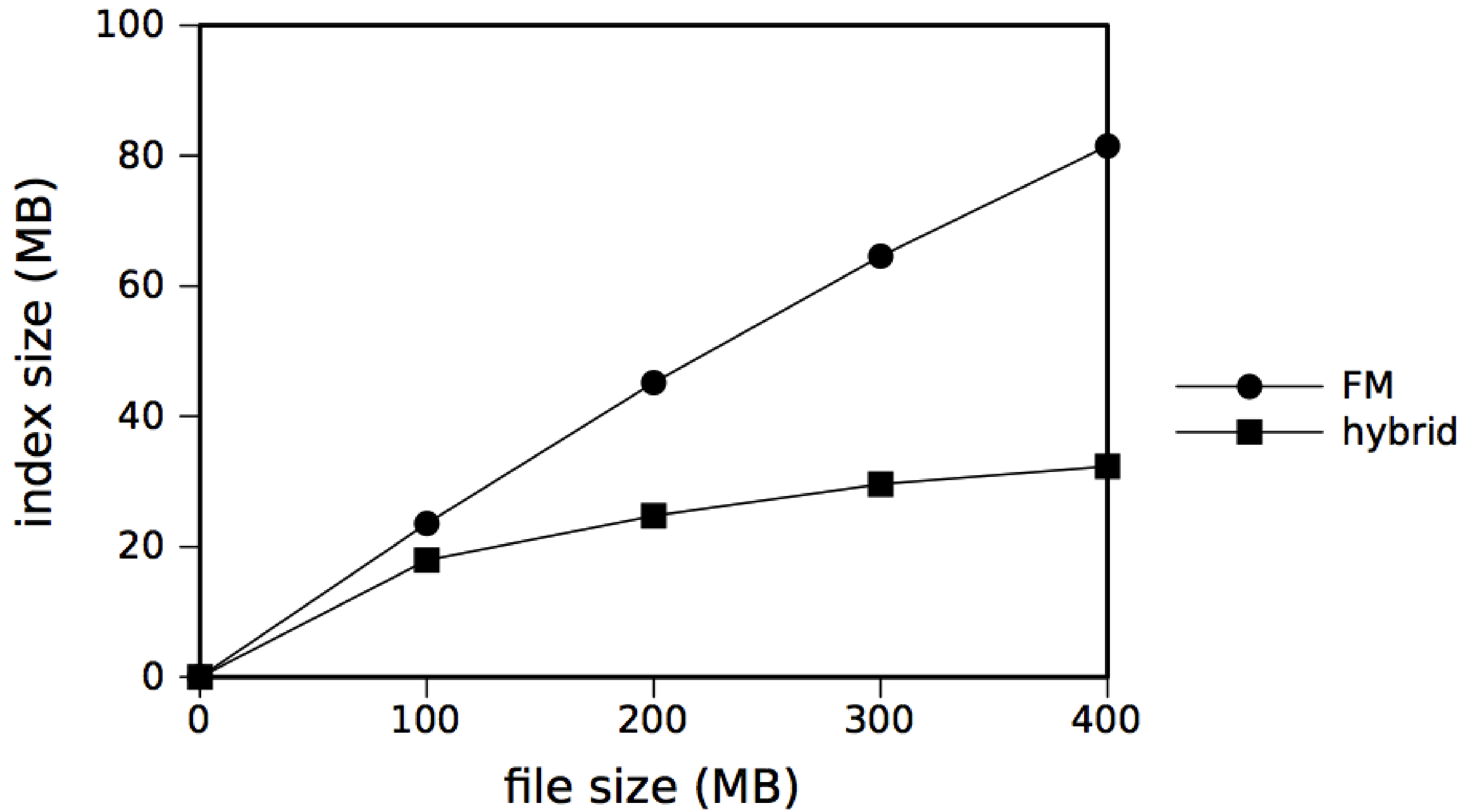
- Time to find all r occurrences of a pattern of length m

Random Access: $O(m + \log n)$

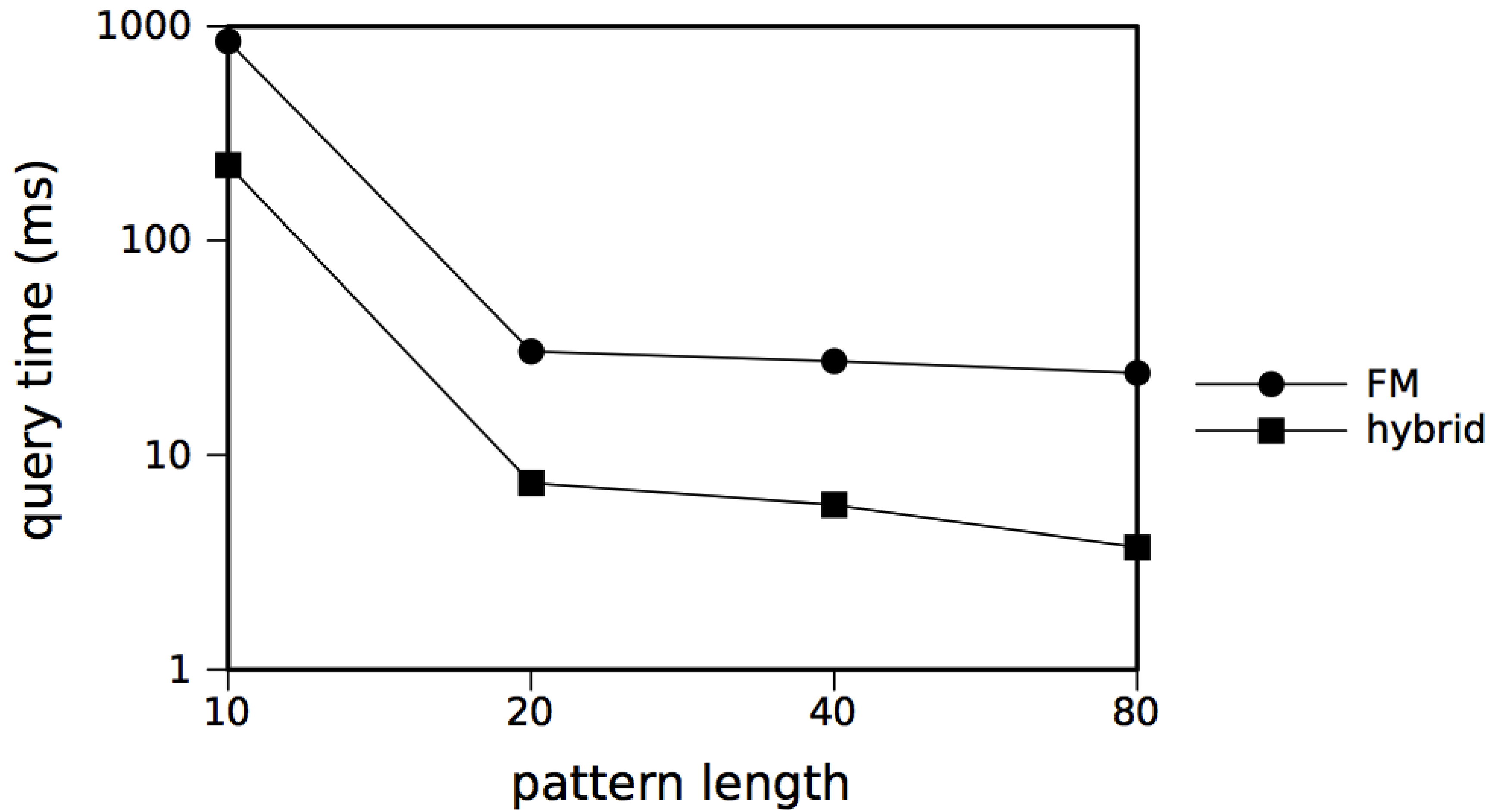
- Time to extract a m symbols from the compressed collection

(To appear at LATIN 2014, next week)

Index Size vs. Collection Size (Yeast Genomes)



Query times



Index Construction

> "We have seen many papers in which the index simply *is*, without discussion of how it was created. But for a indexing scheme to be useful it must be possible for the index to be constructed in a reasonable amount of time."†

We have also figured out how to do LZ parsing in a scalable way.

- To appear at DCC 2014, next week. (Also see Poster Session).
- 100 human genomes ~ 24 hours (on one machine)

---

†Zobel, Moffat, Ramamohanarao, SIGMOD Record, 1996.

Future directions

How efficiently can we estimate compressibility?

- To decide quickly decide when this approach is best

Parallel and distributed indexing algorithms

Combinatorial properties of LZ parsing

- Relationship to grammar and BWT-based compression

Tuning the index for specific applications:

- Genomes: collaboration with Illumina Inc., UK
- Web: relative LZ parsing, VLDB 2012

Bidirectional Burrows-Wheeler Transform

Veli Mäkinen

Helsinki Institute for Information Technology HIIT,
Department of Computer Science, University of Helsinki, Finland

Based on an article by
Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen at
Proc. European Symposium on Algorithms (ESA 2013).

MOTIVATION

Suffix tree [Wei73,...] for text of length n from alphabet of size σ:

- $O(n \log n)$ bits
- Myriads of sequence analysis problems in $O(n)$ time

MOTIVATION

Suffix tree [Wei73,...] for text of length n from alphabet of size σ:

- $O(n \log n)$ bits
- Myriads of sequence analysis problems in $O(n)$ time

Compressed suffix tree [Sad07,...]:

- $O(n \log \sigma)$ bits
- Myriads of sequence analysis problems in $O(n \log^{\epsilon} n)$ time

MOTIVATION

Suffix tree [Wei73,. . .] for text of length n from alphabet of size σ:

- $O(n \log n)$ bits
- Myriads of sequence analysis problems in $O(n)$ time

Compressed suffix tree [Sad07,. . .]:

- $O(n \log \sigma)$ bits
- Myriads of sequence analysis problems in $O(n \log^{\epsilon} n)$ time

Compressed representations for BWT [GV00,FM00,Sad00,. . .]

- Kernel of compressed suffix trees
- A few sequence analysis problems in $O(n \log \sigma)$ time

MOTIVATION

Suffix tree [Wei73,...] for text of length n from alphabet of size σ:

- $O(n \log n)$ bits
- Myriads of sequence analysis problems in $O(n)$ time

Compressed suffix tree [Sad07,...]:

- $O(n \log \sigma)$ bits
- Myriads of sequence analysis problems in $O(n \log^{\epsilon} n)$ time

Compressed representations for BWT [GV00,FM00,Sad00,...]

- Kernel of compressed suffix trees
- A few sequence analysis problems in $O(n \log \sigma)$ time

Succinct space and linear time for myriads of problems?

Our Results at a Glance

Compressed representations for *bidirectional* BWT:

- $O(n \log \sigma)$ bits
- Many sequence analysis problems in $O(n \log \sigma)$ time
- Indexing solutions for a set of sequences:
  - Assume indexes are already built
  - Pair-wise analyses in $O(n)$ time

Our Results at a Glance

Compressed representations for *bidirectional* BWT:

- $O(n \log \sigma)$ bits
- Many sequence analysis problems in $O(n \log \sigma)$ time
- Indexing solutions for a set of sequences:
  - Assume indexes are already built
  - Pair-wise analyses in $O(n)$ time

- Main insights:
  - Conceptual: Visiting suffix tree nodes through suffix link tree $\rightarrow$ No need for LCP array
  - Conceptual: Synchronized traversal of two suffix link trees $\rightarrow$ Indexing solutions for all-against-all analyses
  - Technical: Avoiding LessThan query on wavelet trees $\rightarrow$ Constant time bidirectional backward step

Our Results at a Glance

Compressed representations for *bidirectional* BWT:

- $O(n \log \sigma)$ bits
- Many sequence analysis problems in $O(n \log \sigma)$ time
- Indexing solutions for a set of sequences:
  - Assume indexes are already built
  - Pair-wise analyses in $O(n)$ time

- Main insights:
  - Conceptual: Visiting suffix tree nodes through suffix link tree $\rightarrow$ No need for LCP array
  - Conceptual: Synchronized traversal of two suffix link trees $\rightarrow$ Indexing solutions for all-against-all analyses
  - Technical: Avoiding LessThan query on wavelet trees $\rightarrow$ Constant time bidirectional backward step

Theoretical / practical replacement of compressed suffix trees?

Our Results in Detail

Representation	1		2		3
Implementation	1a	1b	2a [CPM 2010]	2b	3
Space (bits)	$n \log \sigma +$ $+n + o(n)$	$n \log \sigma +$ $+o(n \log \sigma)$	$2n \log \sigma +$ $+o(n)$	$2n \log \sigma +$ $+o(n \log \sigma)$	$O(n \log \sigma)$
`isLeftMaximal`	$O(\log \sigma)$	$O(1)$	$O(\log \sigma)$	$O(1)$	$O(1)$
`isRightMaximal`	$O(1)$	$O(1)$	$O(\log \sigma)$	$O(1)$	$O(1)$
`enumerateLeft`	$O(\log \sigma)$	$O(1)$	$O(\log \sigma)$	$O(1)$	$O(1)$
`enumerateRight`			$O(\log \sigma)$	$O(1)$	$O(1)$
`extendLeft`	$O(\log \sigma)$	$O(\sigma)$	$O(\log \sigma)$	$O(\sigma)$	$O(1)$
`extendRight`			$O(\log \sigma)$	$O(\sigma)$	$O(1)$
Applications	MUM, SUS, MR, LB, QP, IPS, IPK		MUM, SUS, MEM, SR, NSR, MAW, IPS, IPK		BBB

SUS: shortest unique substrings; MR: maximal repeats; LB: longest border; QP: quasiperiod; IPS: inner product of substrings; IPK: inner product of k-mers; (N)SR: (near) supermaximal repeats; MAW: minimal absent words; BBB: bidirectional b&b (supported also by Implementation 2a).

RELATED WORK

- Bidirectional BWT [Lametal09,SOG10]:
  - Bidirectional backward step in $O(\sigma)$ time [Lametal09] and in $O(\log \sigma)$ time [SOG10].
  - We now improve this to $O(1)$ time (on ranges corresponding to suffix tree nodes).

RELATED WORK

- Bidirectional BWT [Lametal09,SOG10]:
  - Bidirectional backward step in $O(\sigma)$ time [Lametal09] and in $O(\log \sigma)$ time [SOG10].
  - We now improve this to $O(1)$ time (on ranges corresponding to suffix tree nodes).
- Avoiding LCP array construction to solve *maximal repeats* [BBO12]:
  - Visiting suffix tree nodes in level-wise order.
  - Analysis uses Weiner links.
  - We improve the space and time and show how to solve many related problems.
  - Our technique extends to *synhronized* search and enables indexing for all-against-all problems.

Related work

- Bidirectional BWT [Lametal09,SOG10]:
  - Bidirectional backward step in $O(\sigma)$ time [Lametal09] and in $O(\log \sigma)$ time [SOG10].
  - We now improve this to $O(1)$ time (on ranges corresponding to suffix tree nodes).
- Avoiding LCP array construction to solve *maximal repeats* [BBO12]:
  - Visiting suffix tree nodes in level-wise order.
  - Analysis uses Weiner links.
  - We improve the space and time and show how to solve many related problems.
  - Our technique extends to *synhronized* search and enables indexing for all-against-all problems.
- Alphabet-independent backward search [BN11,BN13]:
  - We extend the technique for bidirectional backward search.

BIDIRECTIONAL BWT

T=xaby$yabx

T^r=xbay$ybax

sorted suffixes of T# and T^r#

	T#		T^r#
x	#	x	#
y	$yabx#	y	$ybax#
y	abx#	b	ax#
x	aby$yabx#	b	ay$ybax#
a	bx#	y	bax#
a	by$yabx#	x	bay$ybax#
b	x#	a	x#
#	xaby$yabx#	#	xbay$ybax#
b	y$yabx#	a	y$ybax#
$	yabx#	$	ybax#

character preceding each suffix

#<$<a<b<x<y

BIDIRECTIONAL BWT

T=xaby$yabx

L	
x	#
y	$yabx#
y	abx#
x	aby$yabx#
a	bx#
a	by$yabx#
b	x#
#	xaby$yabx#
b	y$yabx#
$	yabx#

[i,j] (rows: y abx#, x aby$yabx#)

T^r=xbay$ybax

L'	
x	#
y	$ybax#
b	ax#
b	ay$ybax#
y	bax#
x	bay$ybax#
a	x#
#	xbay$ybax#
a	y$ybax#
$	ybax#

[i',j'] (rows: b ax#, b ay$ybax#)

- $i' = i = C[\mathrm{a}]$
- $j' = j = C[\mathrm{a}+1] = C[\mathrm{b}] - 1$
- $L_{i\ldots j} = \mathrm{yx}$
- $L'_{i'\ldots j'} = \mathrm{bb}$

Bidirectional BWT

T=xaby$yabx

T^r=xbay$ybax

x	#
y	$yabx#
y	abx#
x	aby$yabx#
a	bx#
a	by$yabx#
b	x#
#	xaby$yabx#
b	y$yabx#
$	yabx#

x	#
y	$ybax#
b	ax#
b	ay$ybax#
y	bax#
x	bay$ybax#
a	x#
#	xbay$ybax#
a	y$ybax#
$	ybax#

- $i' = i + LessThan_{\mathrm{y}}(L_{i...j})$
- $j' = i + LessThan_{\mathrm{y}+1}(L_{i...j}) - 1$
- $i = C[\mathrm{y}] + rank_{\mathrm{y}}(L_{1...i-1}) + 1$
- $j = C[\mathrm{y}] + rank_{\mathrm{y}}(L_{1...j})$

Maximal Unique Matches (MUMs)

Theorem

Substring w is a maximal unique match (MUM) between $s \in \Sigma^*$ and $t \in \Sigma^*$ iff its only occurrences are $s[i, i + |w| - 1]$ and $t[j, j + |w| - 1]$ and extending w left or right looses one of the occurrences. We can discover all the τ maximal unique matches between s and t in $O((|s| + |t|) \log |\Sigma|)$ time and $O((|s| + |t|) \log |\Sigma| + \tau \log(|s| + |t|))$ bits of space.

- For example, on $s =$ `xaby` and $t =$ `yabx` mums are `x`, `y`, `ab`.

Algorithm

```
Algorithm mums(M, bidirectionalBWTindex, i, j, i', j', I)
(1)  left = rank_0(I, j) − rank_0(I, i − 1);
(2)  right = rank_1(I, j) − rank_1(I, i − 1);
(3)  if (left == 0 or right == 0)
(4)        return ;
(5)  if (!bidirectionalBWTindex.rightMaximal(i', j'))
(6)        return ;
(7)  if (bidirectionalBWTindex.leftMaximal(i, j) and left == 1 and right == 1)
(8)        M is a MUM;
(9)  for each c ∈ bidirectionalBWTindex.EnumerateLeft(i, j) do
(10)       (ii, jj, ii', jj') ← bidirectionalBWTindex.extendLeft(c, i, j, i', j');
(11)       mums(cM, bidirectionalBWTindex, ii, jj, ii', jj', I);
...
bidirectionalBWTindex, I ← constructIndex(s$t);
mums("",0, |s| + |t|, 0, |s| + |t|, I);
```

ALGORITHM

Algorithm `mums`(M, `bidirectionalBWTindex`, i, j, i', j', I)
(1) `left` $= rank_0(I, j) - rank_0(I, i-1)$;
(2) `right` $= rank_1(I, j) - rank_1(I, i-1)$;
(3) **if** (`left` $== 0$ **or** `right` $== 0$)
(4) **return** ;
(5) **if** (!`bidirectionalBWTindex.rightMaximal`(i', j'))
(6) **return** ;
(7) **if** (`bidirectionalBWTindex.leftMaximal`(i, j) **and** `left` $== 1$ **and** `right` $== 1$)
(8) M is a MUM;
(9) Recursion with each possible cM...

```
 1234567890
 xaby$yabx
0987654321
         [a]
SA   10 5 7 2 8 3 9 1 4 6
I     1 0 1 0 1 0 1 0 0 1
SA'  10 5 8 3 7 2 9 1 4 6
         [a]
```

ALGORITHM

Algorithm `mums`(M, `bidirectionalBWTindex`, i, j, i', j', I)
(1) `left` $= rank_0(I, j) - rank_0(I, i-1)$;
(2) `right` $= rank_1(I, j) - rank_1(I, i-1)$;
(3) **if** (`left` $== 0$ **or** `right` $== 0$)
(4) **return** ;
(5) **if** (!`bidirectionalBWTindex.rightMaximal`(i', j'))
(6) **return** ;
(7) **if** (`bidirectionalBWTindex.leftMaximal`(i, j) **and** `left` $== 1$ **and** `right` $== 1$)
(8) M is a MUM;
(9) Recursion with each possible cM...

```
 1234567890
 xaby$yabx
0987654321
                [b]
SA   10 5 7 2 8 3 9 1 4 6
I     1 0 1 0 1 0 1 0 0 1
SA'  10 5 8 3 7 2 9 1 4 6
                [b]
```

ALGORITHM

Algorithm `mums`$(M,$ `bidirectionalBWTindex`$, i, j, i', j', I)$
(1) `left` $= rank_0(I,j) - rank_0(I,i-1)$;
(2) `right` $= rank_1(I,j) - rank_1(I,i-1)$;
(3) **if** (`left` $== 0$ **or** `right` $== 0$)
(4) **return** ;
(5) **if** (!`bidirectionalBWTindex.rightMaximal`(i',j'))
(6) **return** ;
(7) **if** (`bidirectionalBWTindex.leftMaximal`(i,j) **and** `left` $== 1$ **and** `right` $== 1$)
(8) M is a MUM;
(9) Recursion with each possible cM...

```
 1234567890
 xaby$yabx
0987654321
         [ab]
SA   10 5 7 2 8 3 9 1 4 6
I     1 0 1 0 1 0 1 0 0 1
SA'  10 5 8 3 7 2 9 1 4 6
             [ba]
```

ANALYSIS

- Number of recursion steps can be bounded by the amount of explicit and implicit Weiner links in suffix tree, which is linear.
- Claimed space bound follows, except for the use of stack:
  - Must use explicit stack, and push the largest interval first; this guarantees $O(\log n)$ depth.
- Bitvector I can be dropped using *synchronized* bidirectional search on two indexes built on s and t separately.

Analysis

- Number of recursion steps can be bounded by the amount of explicit and implicit Weiner links in suffix tree, which is linear.
- Claimed space bound follows, except for the use of stack:
  - Must use explicit stack, and push the largest interval first; this guarantees $O(\log n)$ depth.
- Bitvector I can be dropped using *synchronized* bidirectional search on two indexes built on s and t separately.

- See the paper for more involved applications.

BIDIRECTIONAL STEP IN $O(1)$?

- Bidirectional step requires to count how many symbols smaller than a given symbol there are in a given BWT range (LessThan query).
  - This can be supported by *wavelet tree* in $O(\log \sigma)$ time.
- We show that LessThan query cannot be supported faster than $O(\log \sigma / \log \log n)$ unless using superlinear space.
- However, our algorithms need LessThan query only on ranges corresponding to suffix tree nodes.

BIDIRECTIONAL STEP IN $O(1)$?

- Bidirectional step requires to count how many symbols smaller than a given symbol there are in a given BWT range (LessThan query).
  - This can be supported by *wavelet tree* in $O(\log \sigma)$ time.
- We show that LessThan query cannot be supported faster than $O(\log \sigma / \log \log n)$ unless using superlinear space.
- However, our algorithms need LessThan query only on ranges corresponding to suffix tree nodes.

- It turns out that $O(1)$ time is possible in this restricted setting.

EPILOG

- Construction of bidirectional BWT index and compressed suffix tree is now possible in randomized $O(n)$ time in succinct space.
  - Djamal Belazzougui. Linear time construction of compressed text indexes in compact space. Accepted to *STOC 2014*.

EPILOG

- Construction of bidirectional BWT index and compressed suffix tree is now possible in randomized $O(n)$ time in succinct space.
  - Djamal Belazzougui. Linear time construction of compressed text indexes in compact space. Accepted to *STOC 2014*.
- Hence, ESA 2013 + STOC 2014 papers yield randomized $O(n)$ solutions to myriads of sequence analysis problems using asymptotically optimal space.