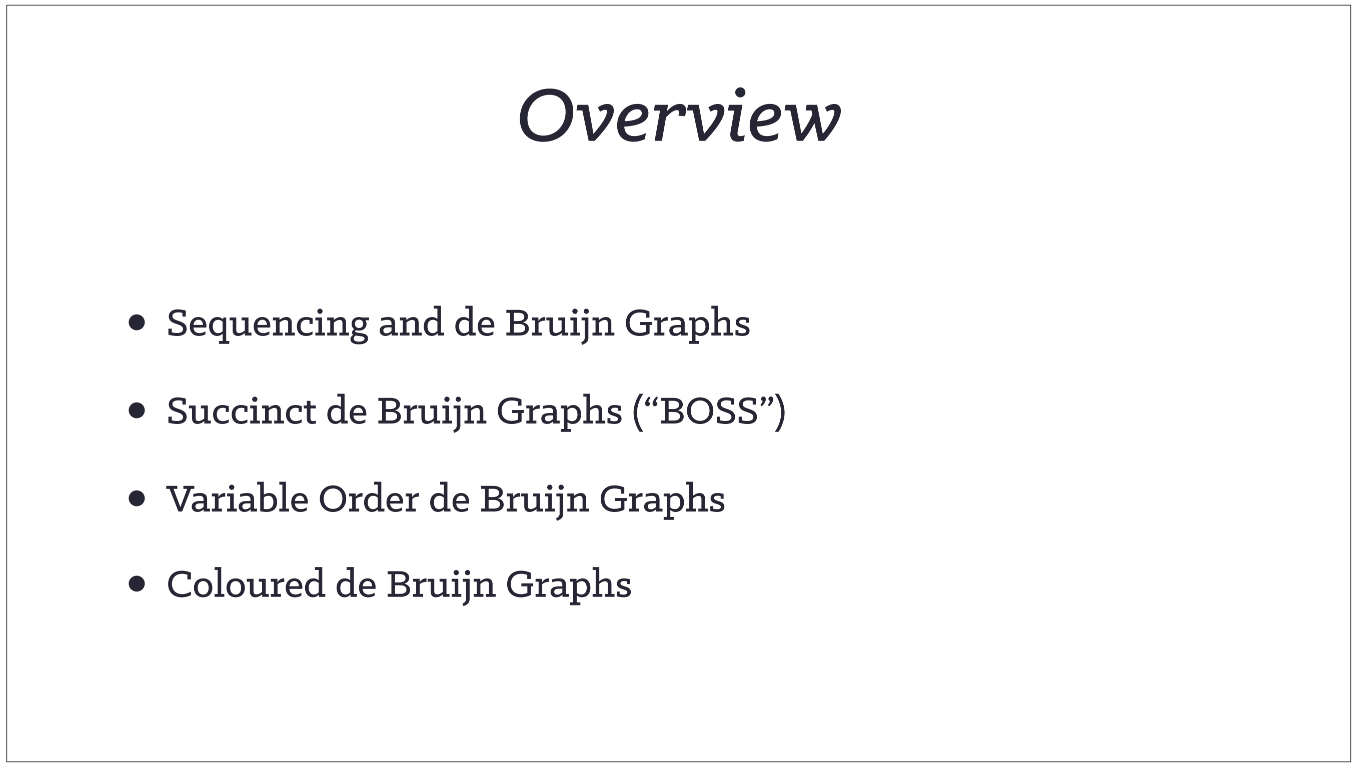
Succinct de Bruijn Graphs

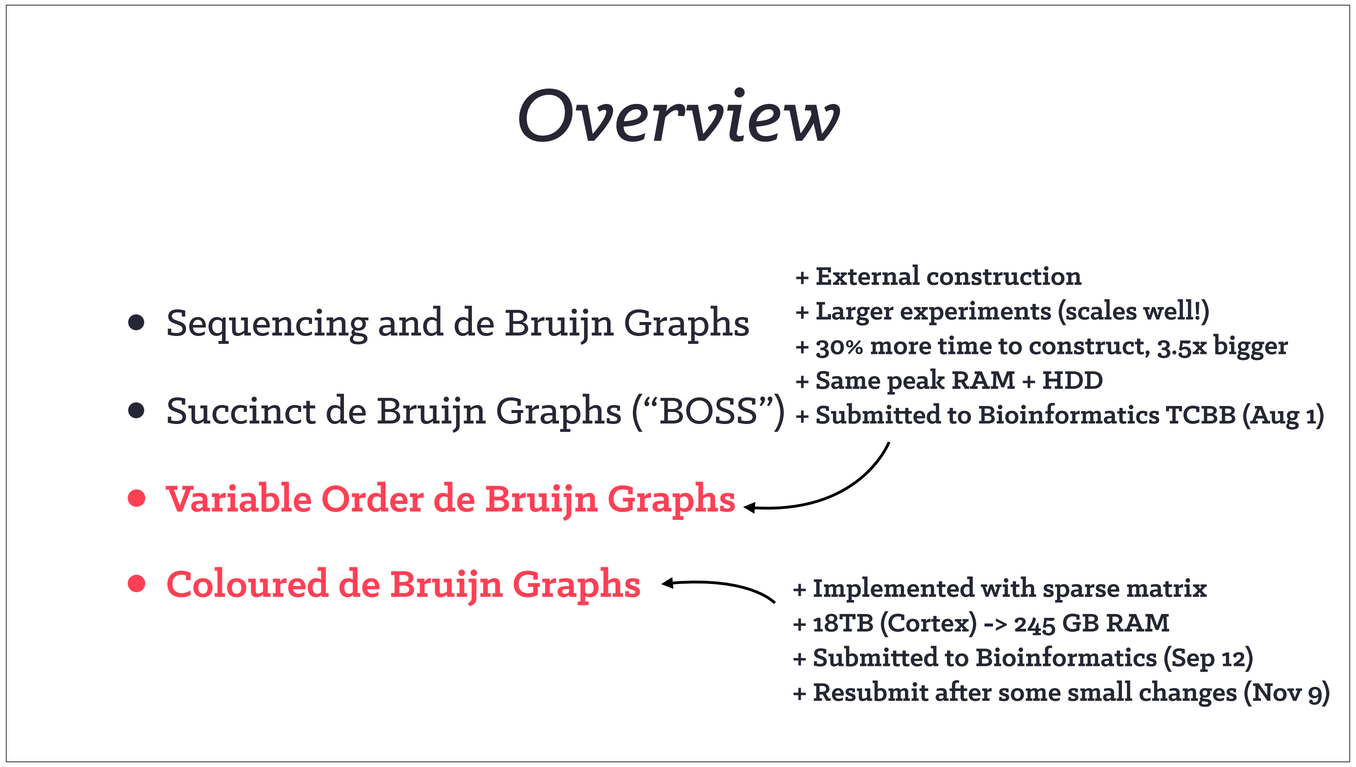
Alex Bowe



Overview

- Sequencing and de Bruijn Graphs
- Succinct de Bruijn Graphs (“BOSS”)
- Variable Order de Bruijn Graphs
- Coloured de Bruijn Graphs

- seq: read into comp
    - most common way to use the sequencing data
    - our way of reducing memory use



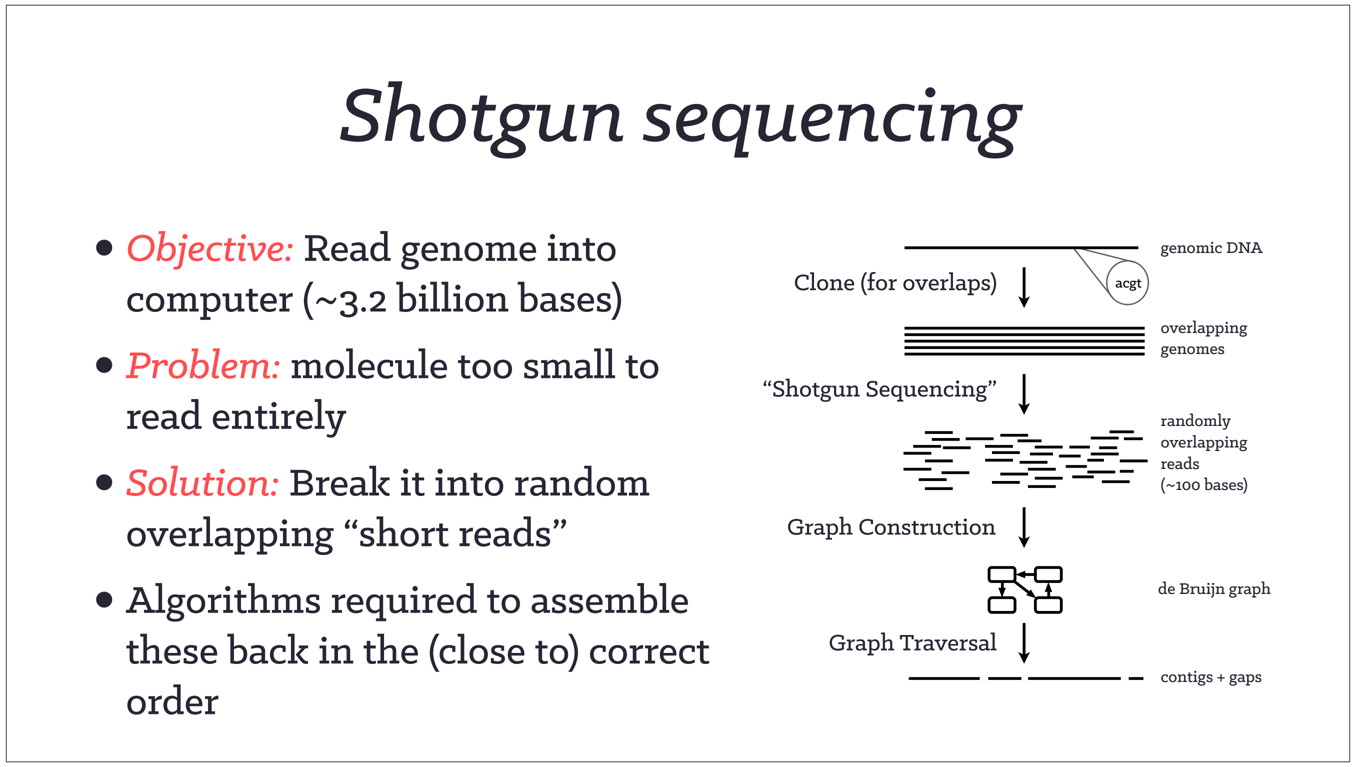
Overview

- Sequencing and de Bruijn Graphs
- Succinct de Bruijn Graphs ("BOSS")
- Variable Order de Bruijn Graphs
- Coloured de Bruijn Graphs

+ External construction
+ Larger experiments (scales well!)
+ 30% more time to construct, 3.5x bigger
+ Same peak RAM + HDD
+ Submitted to Bioinformatics TCBB (Aug 1)

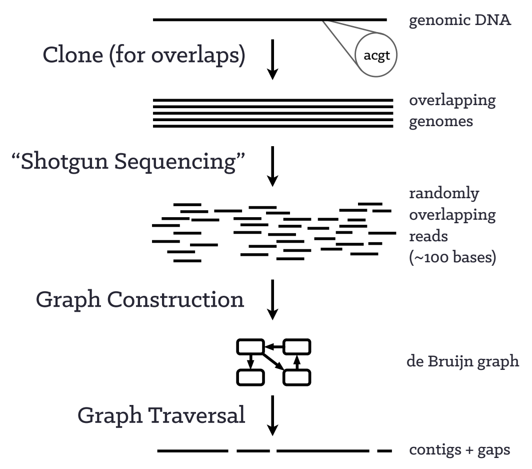
+ Implemented with sparse matrix
+ 18TB (Cortex) -> 245 GB RAM
+ Submitted to Bioinformatics (Sep 12)
+ Resubmit after some small changes (Nov 9)

- small changes: experiments on k, single core, etc
- Havent heard back from TCBB



Shotgun sequencing

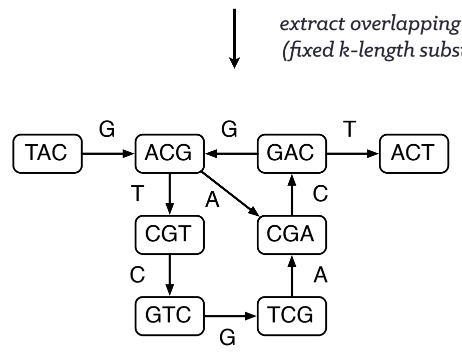
- *Objective:* Read genome into computer (~3.2 billion bases)
- *Problem:* molecule too small to read entirely
- *Solution:* Break it into random overlapping "short reads"
- Algorithms required to assemble these back in the (close to) correct order

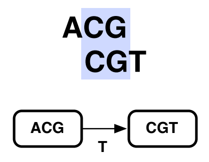


most common way at the moment of getting…
make use of chemical and photographic methods

de Bruijn graphs

T = TACGACGTCGACT *sequencing read*





edge if k-1 overlap

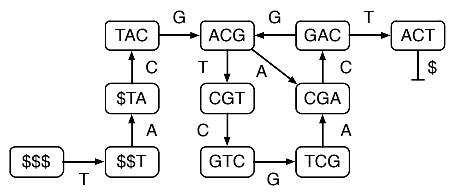
Traversal methods differ, but proposed as finding Eulerian path [Pevzner et al. 2001]

Succinct de Bruijn Graphs

Alex Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya

Construction

T = $$$TACGACGTCGACT$

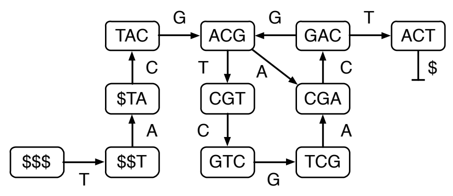


Node			*W*
$	$	$	T
C	G	A	C
$	T	A	C
G	A	C	G
G	A	C	T
T	A	C	G
G	T	C	G
A	C	G	A
A	C	G	T
T	C	G	A
$	$	T	A
A	C	T	$
C	G	T	C

Add dummy edges to ensure every node has an incoming and an outgoing edge – maintain occurrences and relative sorted order in each col

Construction

T = $$$TACGACGTCGACT$

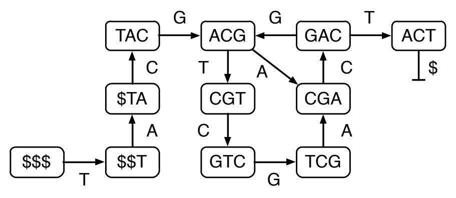


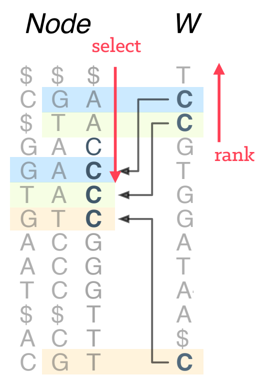
Node			*W*
$	$	$	T
C	G	A	**C**
$	T	A	**C**
G	A	C	G
G	A	**C**	T
T	A	**C**	G
G	T	**C**	G
A	C	G	A
A	C	G	T
T	C	G	A
$	$	T	A
A	C	T	$
C	G	T	**C**

relative sorted order means we can follow edges by counting

Construction

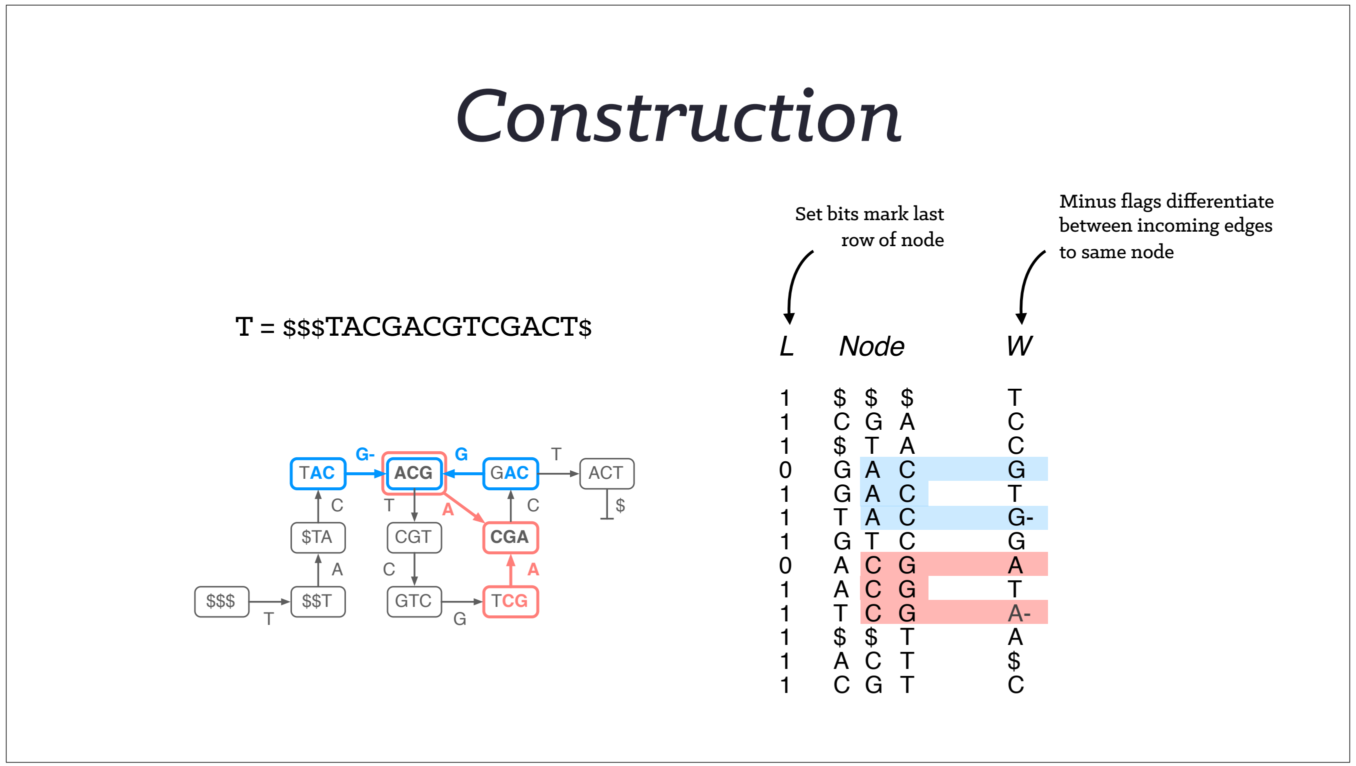
T = $$$TACGACGTCGACT$





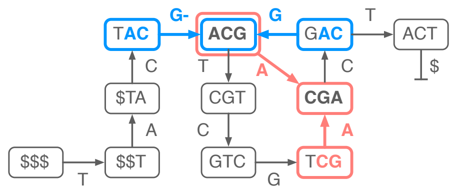
Node			*W*
$	$	$	T
C	G	A	**C**
$	T	A	**C**
G	A	C	G
G	A	**C**	T
T	A	**C**	G
G	T	**C**	G
A	C	G	A
A	C	G	T
T	C	G	A
$	$	T	A
A	C	T	$
C	G	T	**C**

O(1) using rank and select



Construction

T = $$$TACGACGTCGACT$



Set bits mark last row of node

Minus flags differentiate between incoming edges to same node

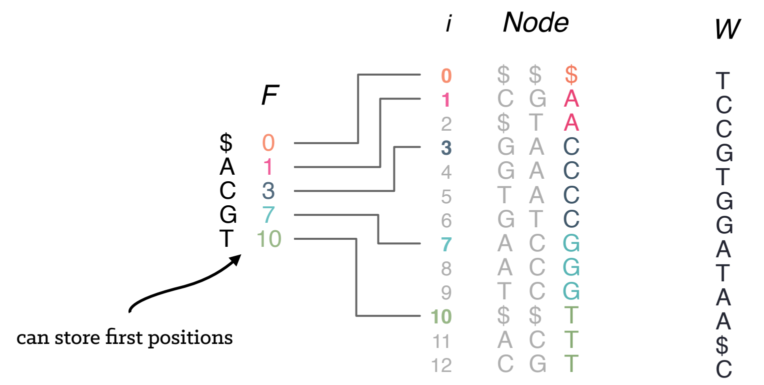
L	*Node*			*W*
1	$	$	$	T
1	C	G	A	C
1	$	T	A	C
0	G	A	C	G
1	G	A	C	T
1	T	A	C	G-
1	G	T	C	G
0	A	C	G	A
1	A	C	G	T
1	T	C	G	A-
1	$	$	T	A
1	A	C	T	$
1	C	G	T	C

- add bit vector to distinguish nodes (which are ranges)
- corresponds to these incoming edges
- distinguishing these is important for other operations

Construction

cyclic property: don't need to store full node labels

just last two columns (to count)



	F
$	0
A	1
C	3
G	7
T	10

i	*Node*			*W*
0	$	$	$	T
1	C	G	A	C
2	$	T	A	C
3	G	A	C	G
4	G	A	C	T
5	T	A	C	G
6	G	T	C	G
7	A	C	G	A
8	A	C	G	T
9	T	C	G	A
10	$	$	T	A
11	A	C	T	$
12	C	G	T	C

Only store edges, order defines the context

In Total

for m edges:

$\sigma \log(m)$ bits

F	*L*	*W*
	1	T
	1	C
	1	C
	0	G
0	1	T
1	1	G-
3	1	G
7	0	A
10	1	T
	1	A-
	1	A
	1	$
	1	C

m bits

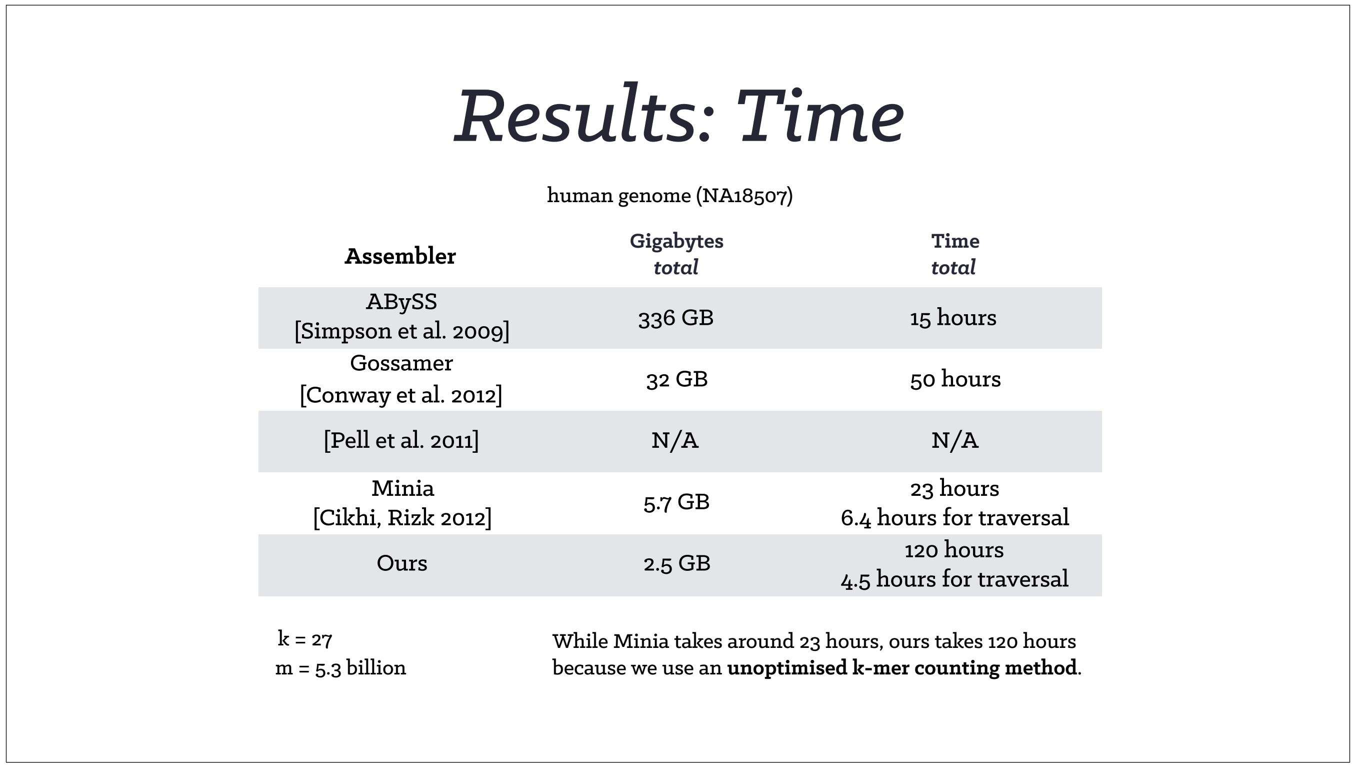
$m \log(2\sigma) = m + m \log(\sigma)$ bits

Total: $2 + \log(\sigma) + o(1)$ bits per edge = 4 bits for DNA (~2 compressed)

Results: Size

Assembler	**Approach**	**Bits** *per edge*
ABySS [Simpson et al. 2009]	Distributed hash table	~300
Gossamer [Conway et al. 2012]	Succinct bit vector	28.5
[Pell et al. 2011]	Bloom filter w/ false pos. edges	4~9
Minia [Cikhi, Rizk 2012]	Bloom filter w/ aux. struct	13.5
Ours	BWT-inspired w/ succinct rank/select indices	4 + o(1) (~2 after compression)

Depending on k (here **k = 27**)



Results: Time

human genome (NA18507)

Assembler	**Gigabytes** *total*	**Time** *total*
ABySS [Simpson et al. 2009]	336 GB	15 hours
Gossamer [Conway et al. 2012]	32 GB	50 hours
[Pell et al. 2011]	N/A	N/A
Minia [Cikhi, Rizk 2012]	5.7 GB	23 hours 6.4 hours for traversal
Ours	2.5 GB	120 hours 4.5 hours for traversal

k = 27
m = 5.3 billion

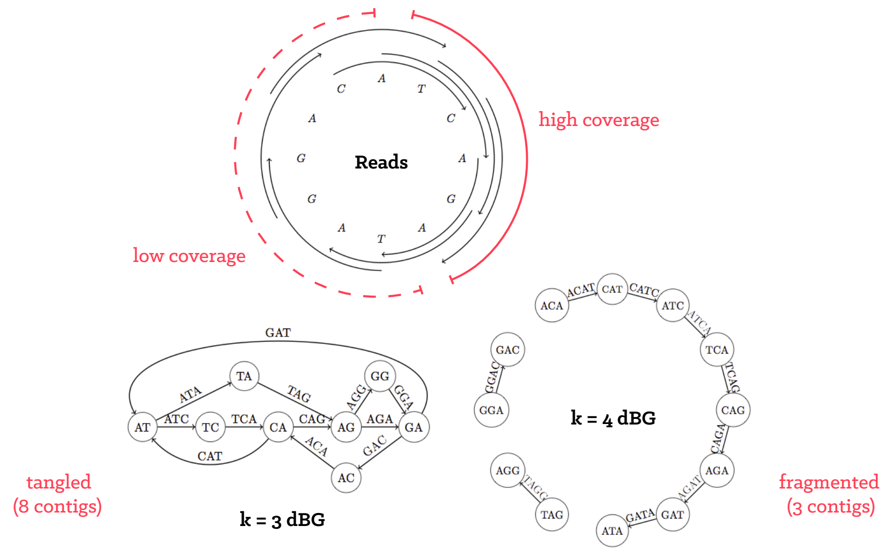
While Minia takes around 23 hours, ours takes 120 hours because we use an **unoptimised k-mer counting method**.

Outdated, still running tests on new version minia and new version of ours

Variable Order de Bruijn Graphs

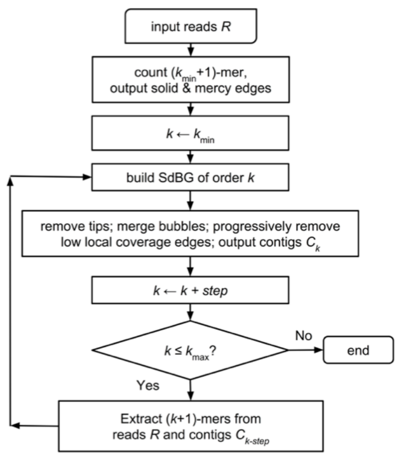
Alex Bowe, Christina Boucher, Travis Gagie, Simon J. Puglisi, Kunihiko Sadakane

Coverage is not uniform



Trade off in deciding k

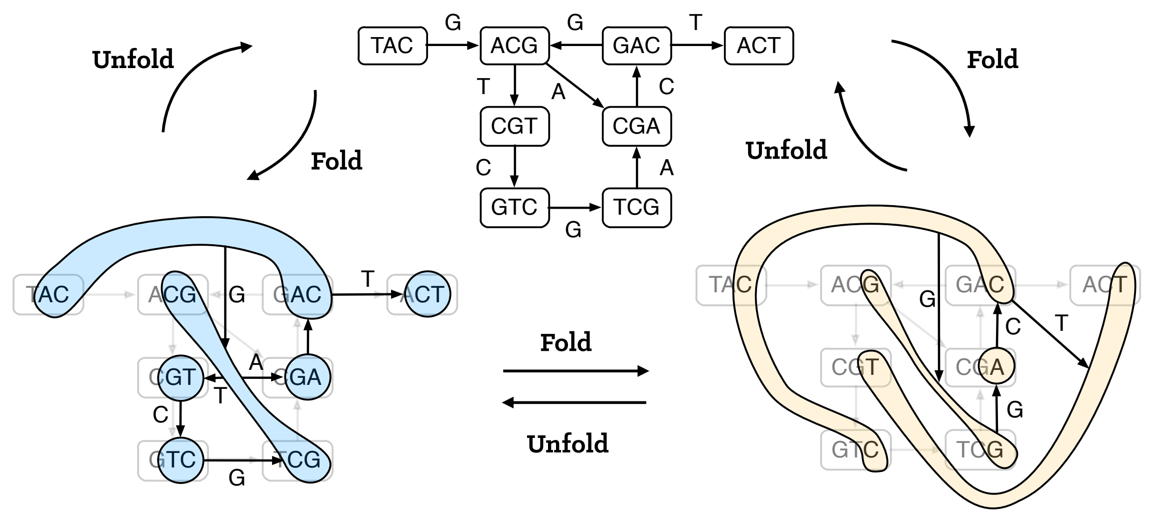
Iterative de Bruijn graphs


input reads R
count (k_{min}+1)-mer, output solid & mercy edges
$k \leftarrow k_{min}$
build SdBG of order k
remove tips; merge bubbles; progressively remove low local coverage edges; output contigs C_k
$k \leftarrow k + step$
$k \leq k_{max}$?
No
end
Yes
Extract (k+1)-mers from reads R and contigs $C_{k\text{-}step}$

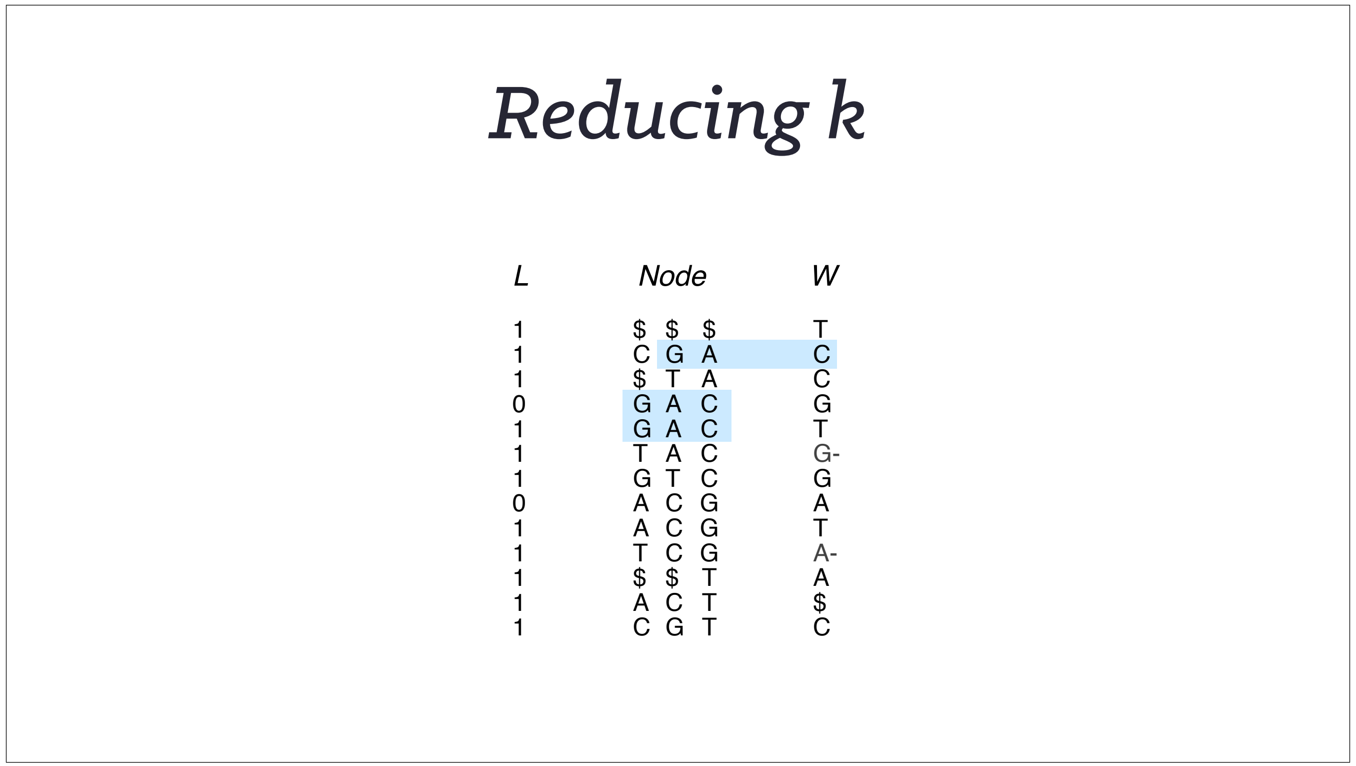
factor t slowdown
(for t steps of k)

- Process reads many times – one of slowest parts
- step != 1 (which would be better)

Variable-order de Bruijn graphs



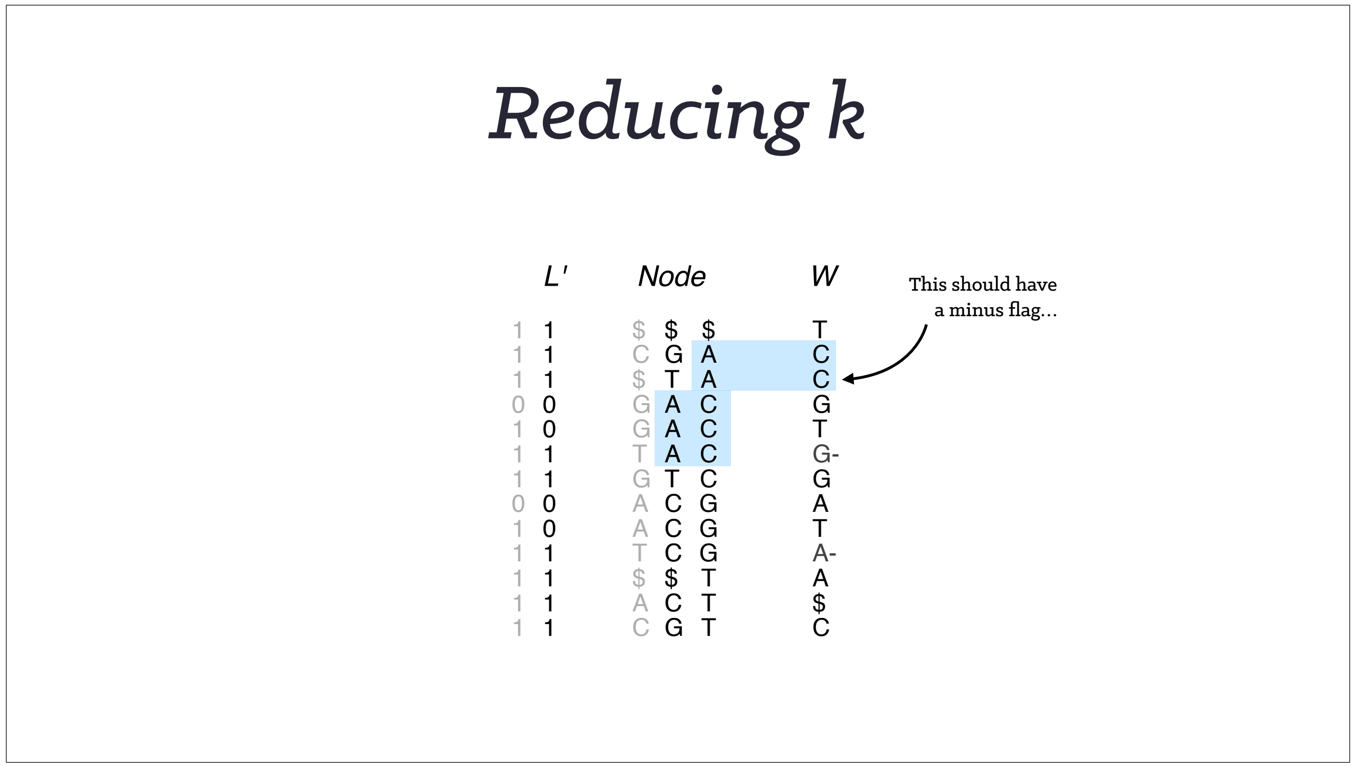
- succinct, on the fly, any k
- construct graph only once: avoid t slowdown



Reducing k

L	*Node*	*W*
1	$ $ $	T
1	C G A	C
1	$ T A	C
0	G A C	G
1	G A C	T
1	T A C	G-
1	G T C	G
0	A C G	A
1	A C G	T
1	T C G	A-
1	$ $ T	A
1	A C T	$
1	C G T	C

- Take our standard succinct dBG

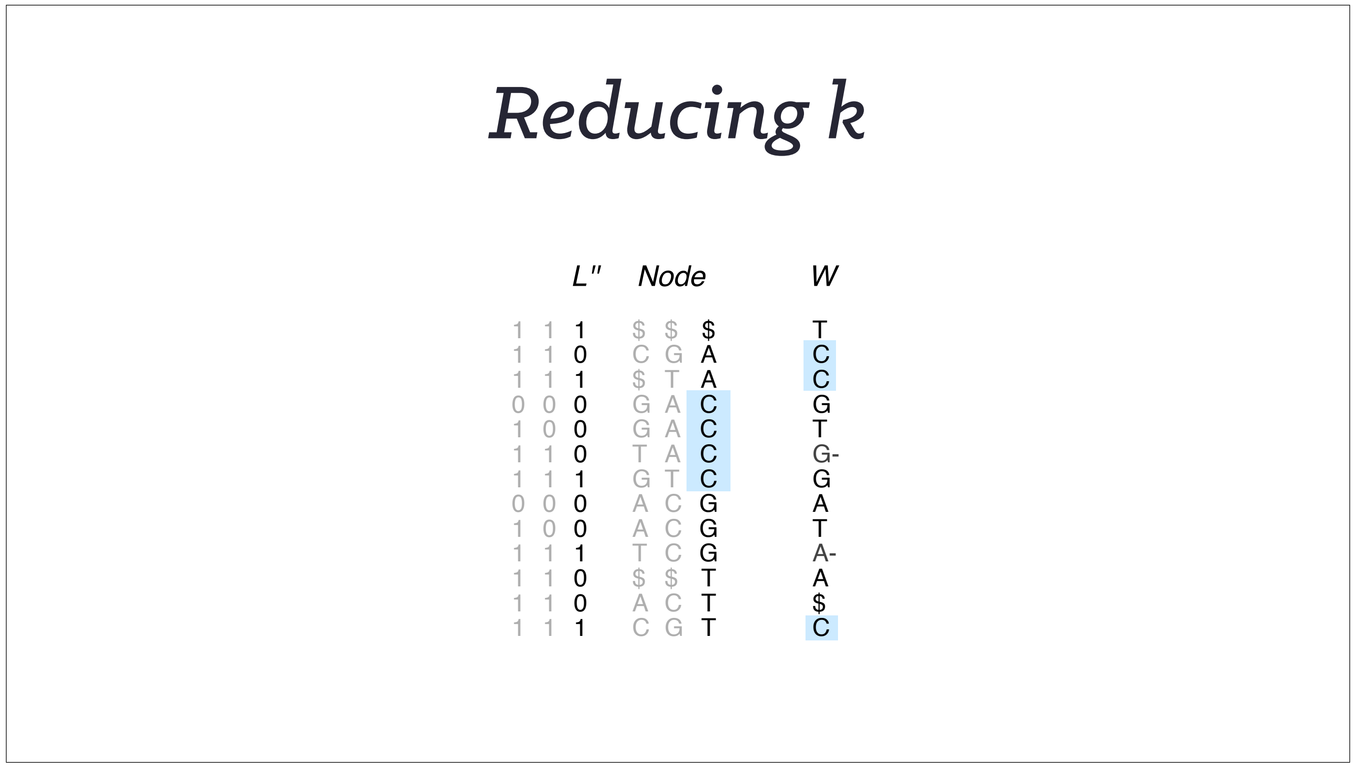


Reducing *k*

	L'		*Node*		*W*
1	1	$	$	$	T
1	1	C	G	A	C
1	1	$	T	A	C
0	0	G	A	C	G
1	0	G	A	C	T
1	1	T	A	C	G-
1	1	G	T	C	G
0	0	A	C	G	A
1	0	A	C	G	T
1	1	T	C	G	A-
1	1	$	$	T	A
1	1	A	C	T	$
1	1	C	G	T	C

This should have a minus flag...

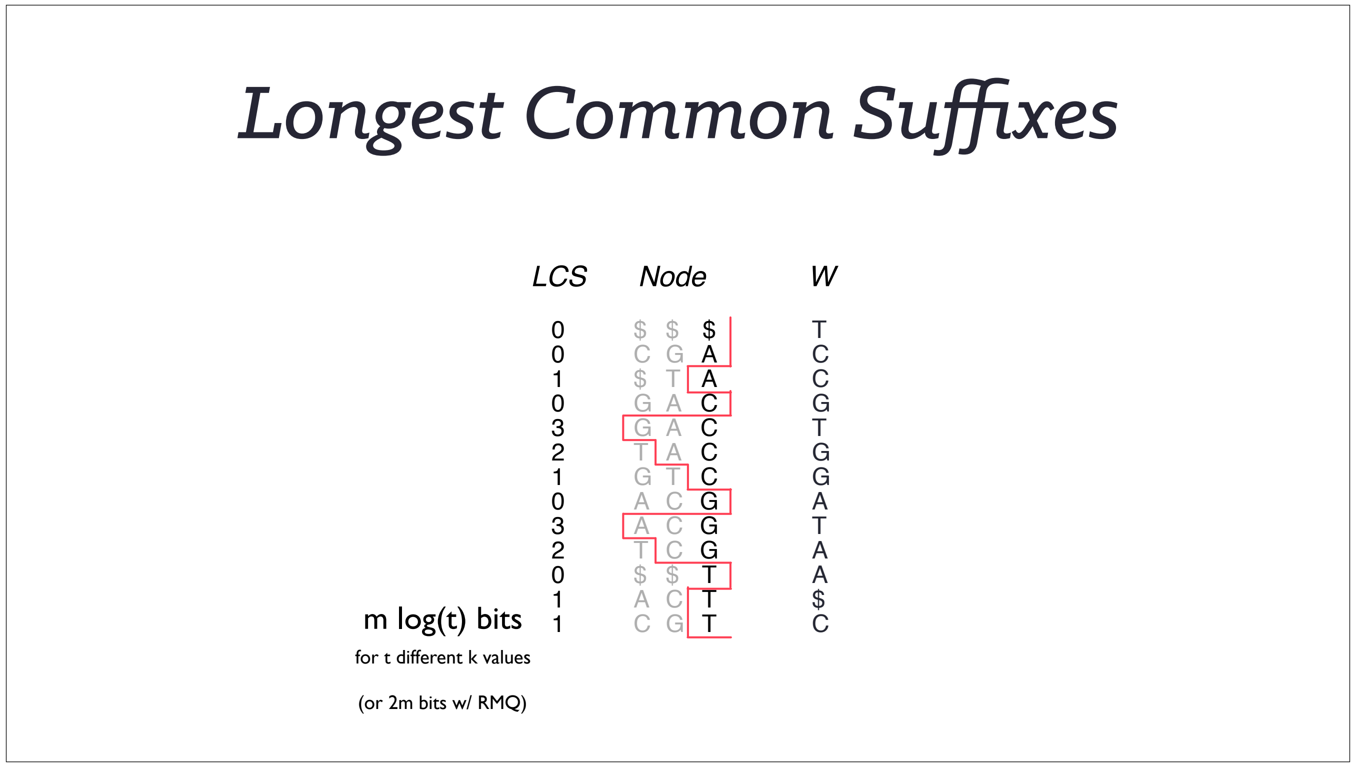
– 1 sometimes becomes 0, 0 never 1, never adding anything to make it a new node



Reducing k

		L″	*Node*			*W*
1	1	1	$	$	$	T
1	1	0	C	G	A	C
1	1	1	$	T	A	C
0	0	0	G	A	C	G
1	0	0	G	A	C	T
1	1	0	T	A	C	G-
1	1	1	G	T	C	G
0	0	0	A	C	G	A
1	0	0	A	C	G	T
1	1	1	T	C	G	A-
1	1	0	$	$	T	A
1	1	0	A	C	T	$
1	1	1	C	G	T	C

– Echelon form



Longest Common Suffixes

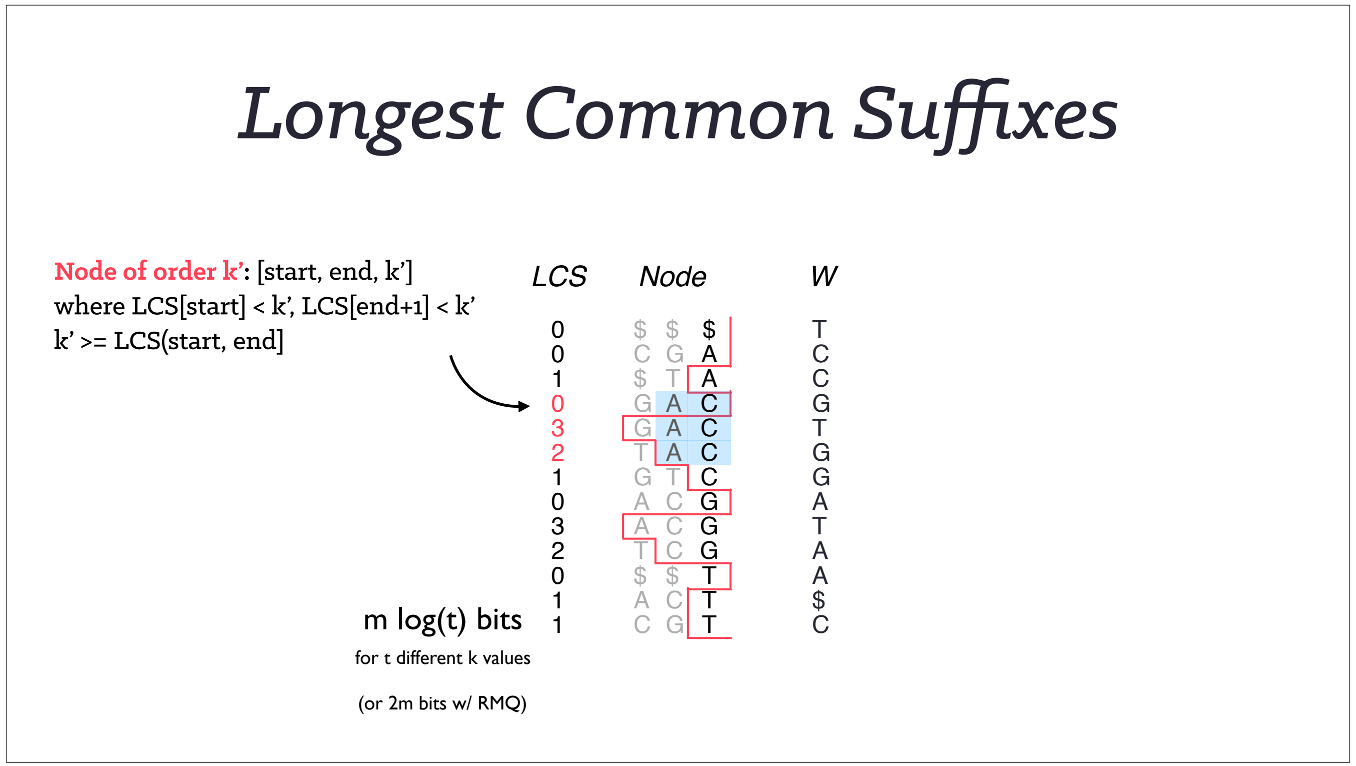
LCS	*Node*			*W*
0	$	$	$	T
0	C	G	A	C
1	$	T	A	C
0	G	A	C	G
3	G	A	C	T
2	T	A	C	G
1	G	T	C	G
0	A	C	G	A
3	A	C	G	T
2	T	C	G	A
0	$	$	T	A
1	A	C	T	$
1	C	G	T	C

m log(t) bits

for t different k values

(or 2m bits w/ RMQ)

– Measures the shared suffix length with previous node



Longest Common Suffixes

Node of order k': [start, end, k']
where LCS[start] < k', LCS[end+1] < k'
k' >= LCS(start, end]

LCS	*Node*			*W*
0	$	$	$	T
0	C	G	A	C
1	$	T	A	C
0	G	A	C	G
3	G	A	C	T
2	T	A	C	G
1	G	T	C	G
0	A	C	G	A
3	A	C	G	T
2	T	C	G	A
0	$	$	T	A
1	A	C	T	$
1	C	G	T	C

m log(t) bits

for t different k values

(or 2m bits w/ RMQ)

- Replaces L and minus flags
- This obs the ONE thing you should remember
- LCS can be discretised

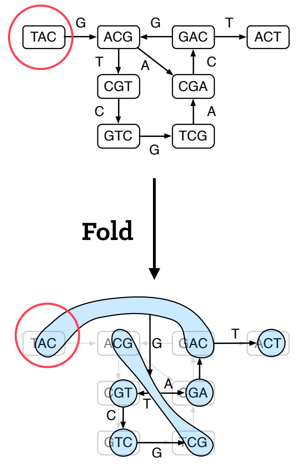
Fold

Easy to **increase/decrease range** if stored in **Wavelet Tree**

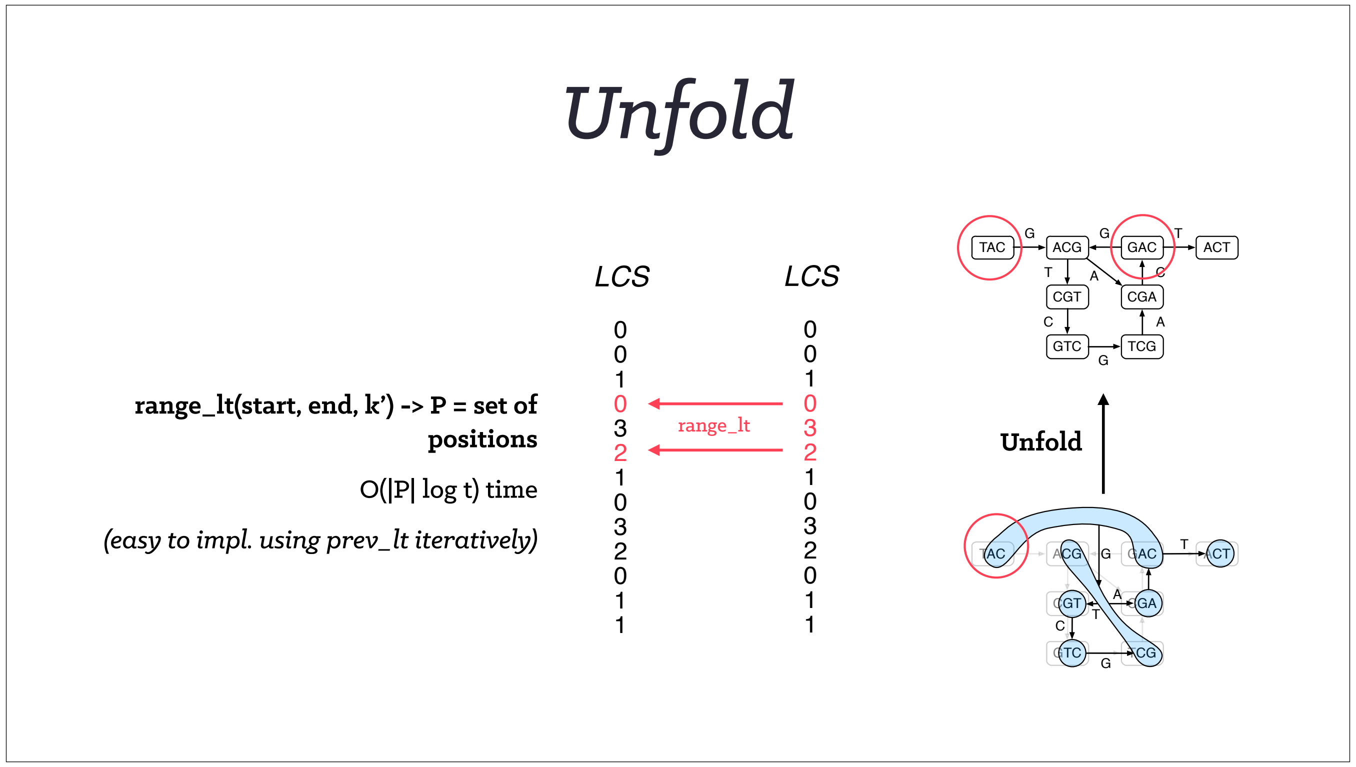
[**prev_lt(start, k'), next_lt(end, k')**)

O(log t) time

LCS		*LCS*
0		0
0		0
1		1
0	prev_lt	0
3		3
2		2
1	next_lt	1
0		0
3		3
2		2
0		0
1		1
1		1



- Replaces L and minus flags
- This obs the ONE thing you should remember
- LCS can be discretised



Unfold

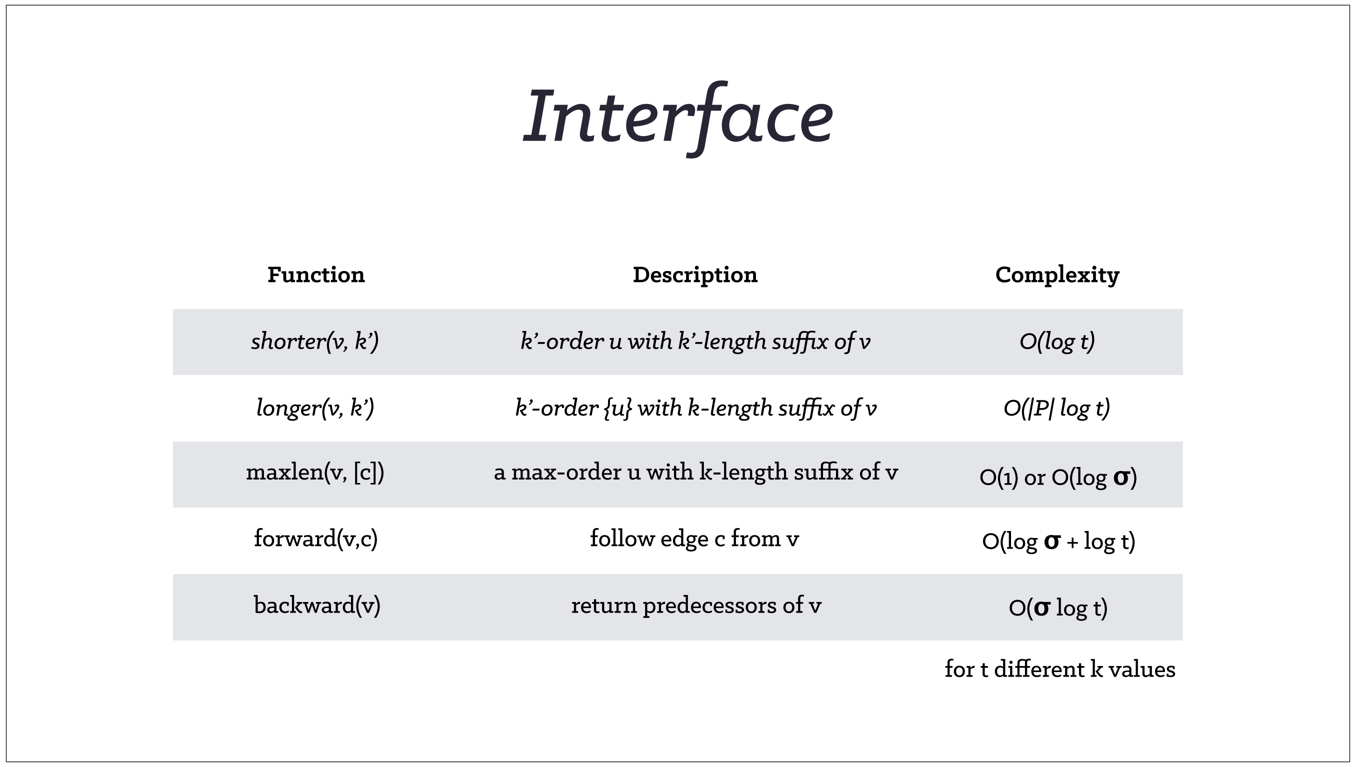
range_lt(start, end, k') -> P = set of positions

O(|P| log t) time

(easy to impl. using prev_lt iteratively)

LCS		*LCS*
0		0
0		0
1		1
0	←	0
3	range_lt	3
2	←	2
1		1
0		0
3		3
2		2
0		0
1		1
1		1

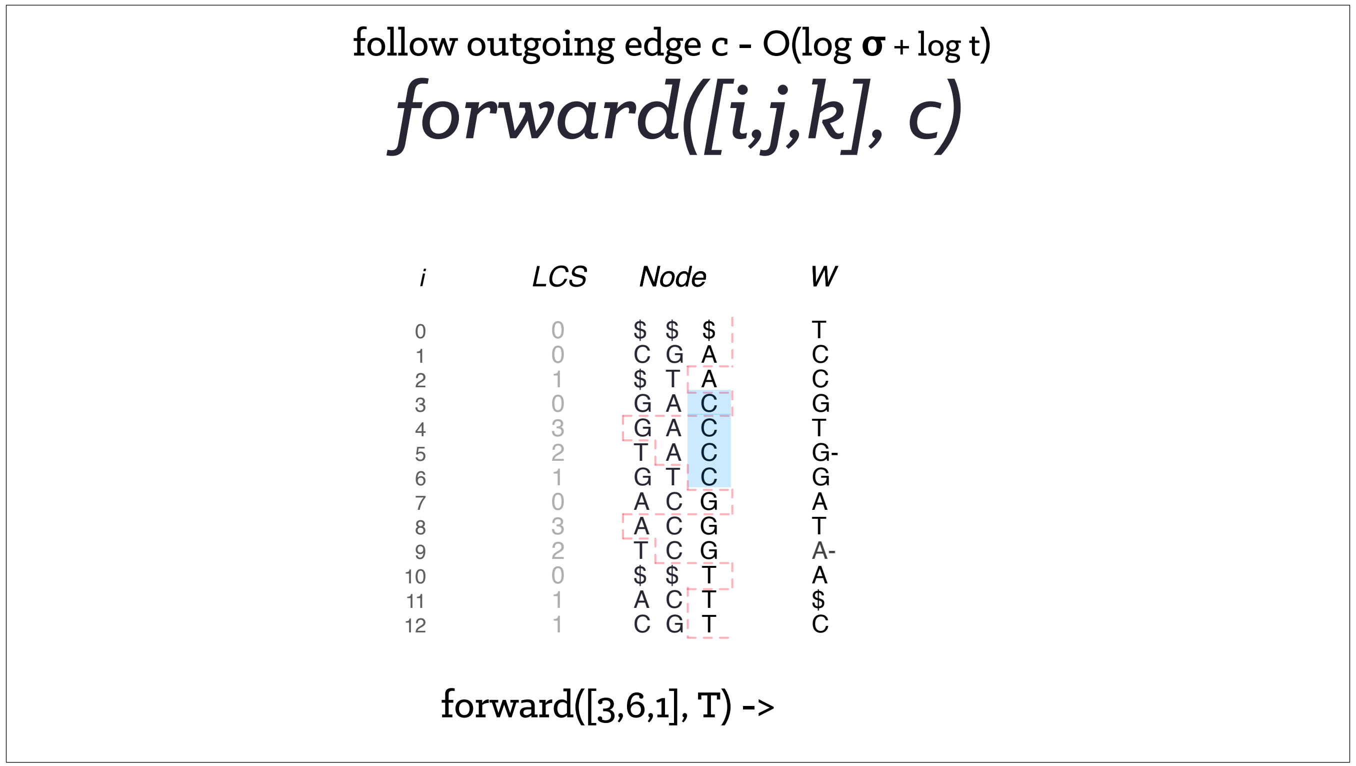
– return a set because increasing context means we are possibly in multiple sets at once.



Interface

Function	Description	Complexity
shorter(v, k')	*k'-order u with k'-length suffix of v*	*O(log t)*
longer(v, k')	*k'-order {u} with k-length suffix of v*	*O(|P| log t)*
maxlen(v, [c])	a max-order u with k-length suffix of v	O(1) or O(log **σ**)
forward(v,c)	follow edge c from v	O(log **σ** + log t)
backward(v)	return predecessors of v	O(**σ** log t)
		for t different k values

- Wont go into detail, suffice it to say that forward and backward are implemented by zooming in to find… zoom back out after following the edge



follow outgoing edge c - O(log **σ** + log t)

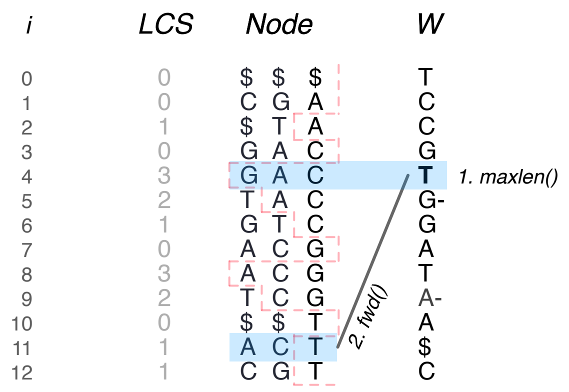
forward([i,j,k], c)

i	*LCS*	*Node*	*W*
0	0	$ $ $	T
1	0	C G A	C
2	1	$ T A	C
3	0	G A C	G
4	3	G A C	T
5	2	T A C	G-
6	1	G T C	G
7	0	A C G	A
8	3	A C G	T
9	2	T C G	A-
10	0	$ $ T	A
11	1	A C T	$
12	1	C G T	C

forward([3,6,1], T) ->

- so if we want to actually follow that edge but keep the same k value follow outgoing edge c - O(log **σ** + log t)

forward([i,j,k], c)

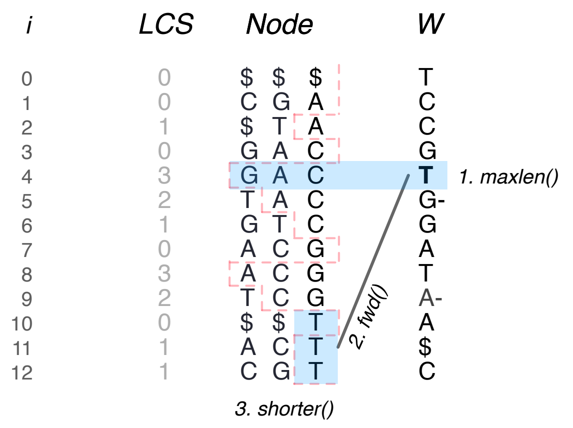


i	*LCS*	*Node*			*W*
0	0	$	$	$	T
1	0	C	G	A	C
2	1	$	T	A	C
3	0	G	A	C	G
4	3	G	A	C	**T**
5	2	T	A	C	G-
6	1	G	T	C	G
7	0	A	C	G	A
8	3	A	C	G	T
9	2	T	C	G	A-
10	0	$	$	T	A
11	1	A	C	T	$
12	1	C	G	T	C

forward([3,6,1], T) ->

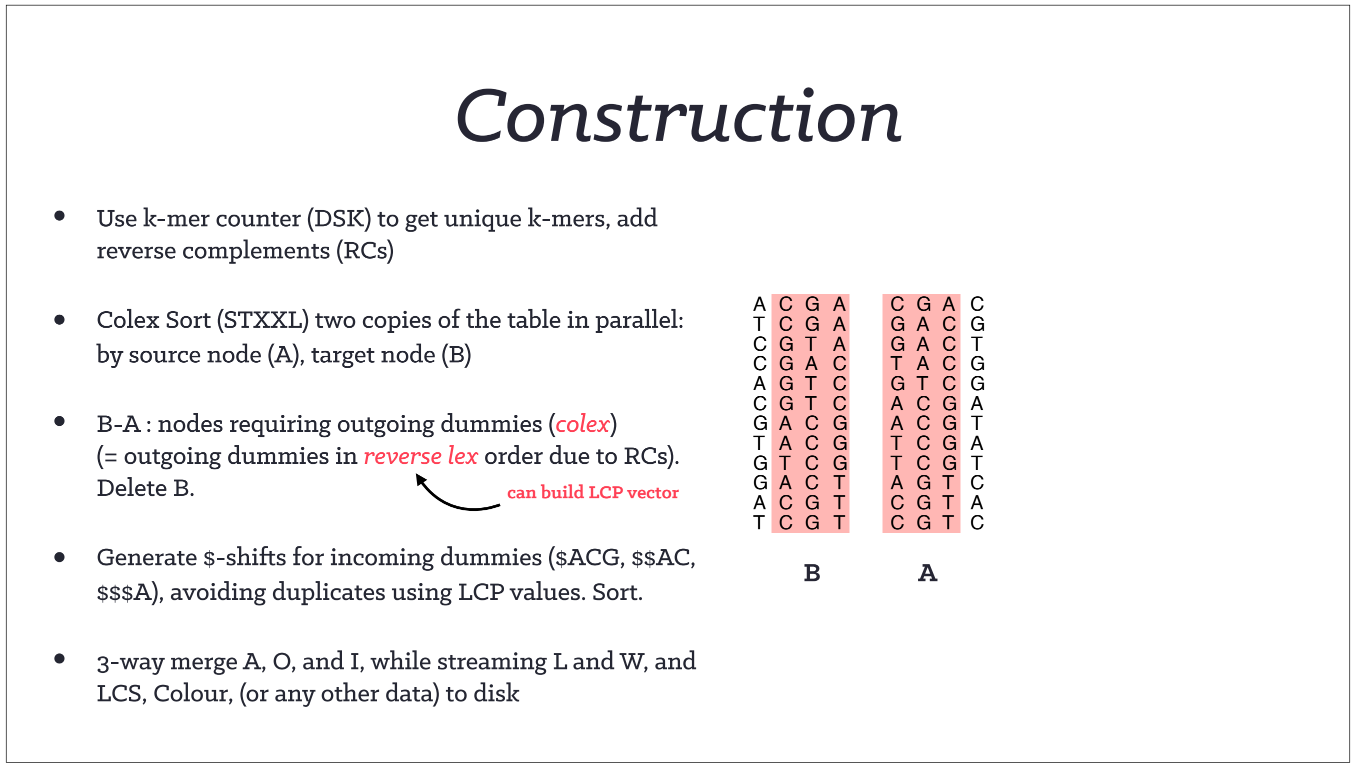
follow outgoing edge c - O(log **σ** + log t)

forward([i,j,k], c)



i	*LCS*	*Node*	*W*
0	0	$ $ $	T
1	0	C G A	C
2	1	$ T A	C
3	0	G A C	G
4	3	G A C	**T**
5	2	T A C	G-
6	1	G T C	G
7	0	A C G	A
8	3	A C G	T
9	2	T C G	A-
10	0	$ $ T	A
11	1	A C T	$
12	1	C G T	C

forward([3,6,1], T) -> [10,12,1]



Construction

- Use k-mer counter (DSK) to get unique k-mers, add reverse complements (RCs)
- Colex Sort (STXXL) two copies of the table in parallel: by source node (A), target node (B)
- B-A : nodes requiring outgoing dummies (*colex*) (= outgoing dummies in *reverse lex* order due to RCs). Delete B.

  can build LCP vector
- Generate $-shifts for incoming dummies ($ACG, $$AC, $$$A), avoiding duplicates using LCP values. Sort.
- 3-way merge A, O, and I, while streaming L and W, and LCS, Colour, (or any other data) to disk

A	C	G	A
T	C	G	A
C	G	T	A
C	G	A	C
A	G	T	C
C	G	T	C
G	A	C	G
T	A	C	G
G	T	C	G
G	A	C	T
A	C	G	T
T	C	G	T

B

C	G	A	C
G	A	C	G
G	A	C	T
T	A	C	G
G	T	C	G
A	C	G	A
A	C	G	T
T	C	G	A
T	C	G	T
A	G	T	C
C	G	T	A
C	G	T	C

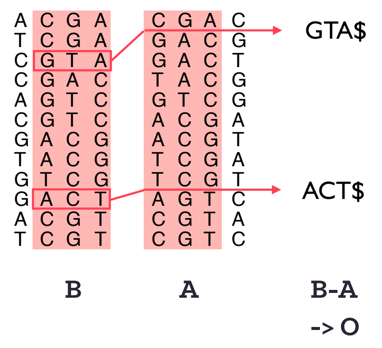
A

- cant freq. filter because doing so might remove lower order nodes that would be of an acceptable freq.

Construction

- Use k-mer counter (DSK) to get unique k-mers, add reverse complements (RCs)
- Colex Sort (STXXL) two copies of the table in parallel: by source node (A), target node (B)
- B-A : nodes requiring outgoing dummies (*colex*) (= outgoing dummies in *reverse lex* order due to RCs). Delete B.

  can build LCP vector
- Generate $-shifts for incoming dummies ($ACG, $$AC, $$$A), avoiding duplicates using LCP values. Sort.
- 3-way merge A, O, and I, while streaming L and W, and LCS, Colour, (or any other data) to disk



Construction

- Use k-mer counter (DSK) to get unique k-mers, add reverse complements (RCs)

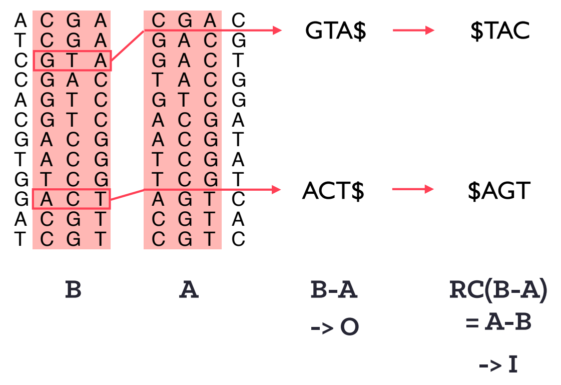
- Colex Sort (STXXL) two copies of the table in parallel: by source node (A), target node (B)

- B-A : nodes requiring outgoing dummies (*colex*) (= outgoing dummies in *reverse lex* order due to RCs). Delete B.

  can build LCP vector

- Generate \$-shifts for incoming dummies (\$ACG, \$\$AC, \$\$\$A), avoiding duplicates using LCP values. Sort.

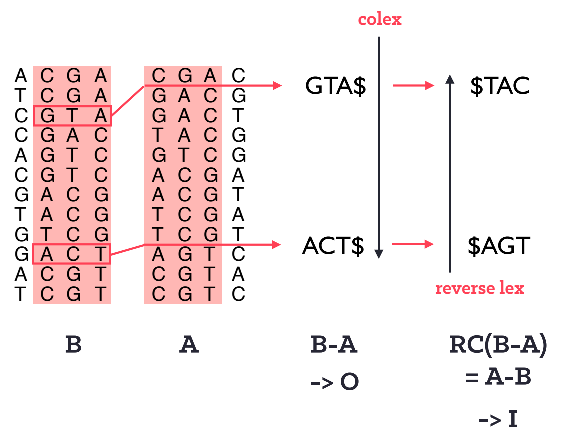
- 3-way merge A, O, and I, while streaming L and W, and LCS, Colour, (or any other data) to disk



Construction

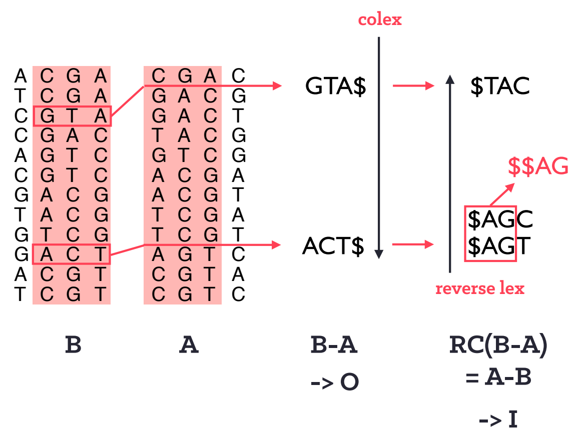
- Use k-mer counter (DSK) to get unique k-mers, add reverse complements (RCs)
- Colex Sort (STXXL) two copies of the table in parallel: by source node (A), target node (B)
- B-A : nodes requiring outgoing dummies (*colex*) (= outgoing dummies in *reverse lex* order due to RCs). Delete B.

  can build LCP vector
- Generate $-shifts for incoming dummies ($ACG, $$AC, $$$A), avoiding duplicates using LCP values. Sort.
- 3-way merge A, O, and I, while streaming L and W, and LCS, Colour, (or any other data) to disk



Construction

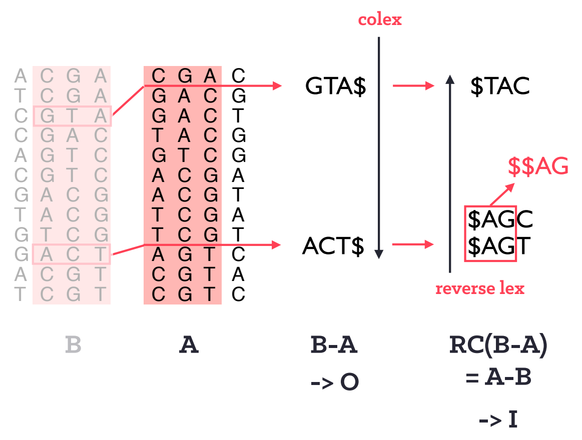
- Use k-mer counter (DSK) to get unique k-mers, add reverse complements (RCs)
- Colex Sort (STXXL) two copies of the table in parallel: by source node (A), target node (B)
- B-A : nodes requiring outgoing dummies (*colex*) (= outgoing dummies in *reverse lex* order due to RCs). Delete B.
  can build LCP values
- Generate $-shifts for incoming dummies ($ACG, $$AC, $$$A), avoiding duplicates using LCP values. Sort.
- 3-way merge A, O, and I, while streaming L and W, and LCS, Colour, (or any other data) to disk



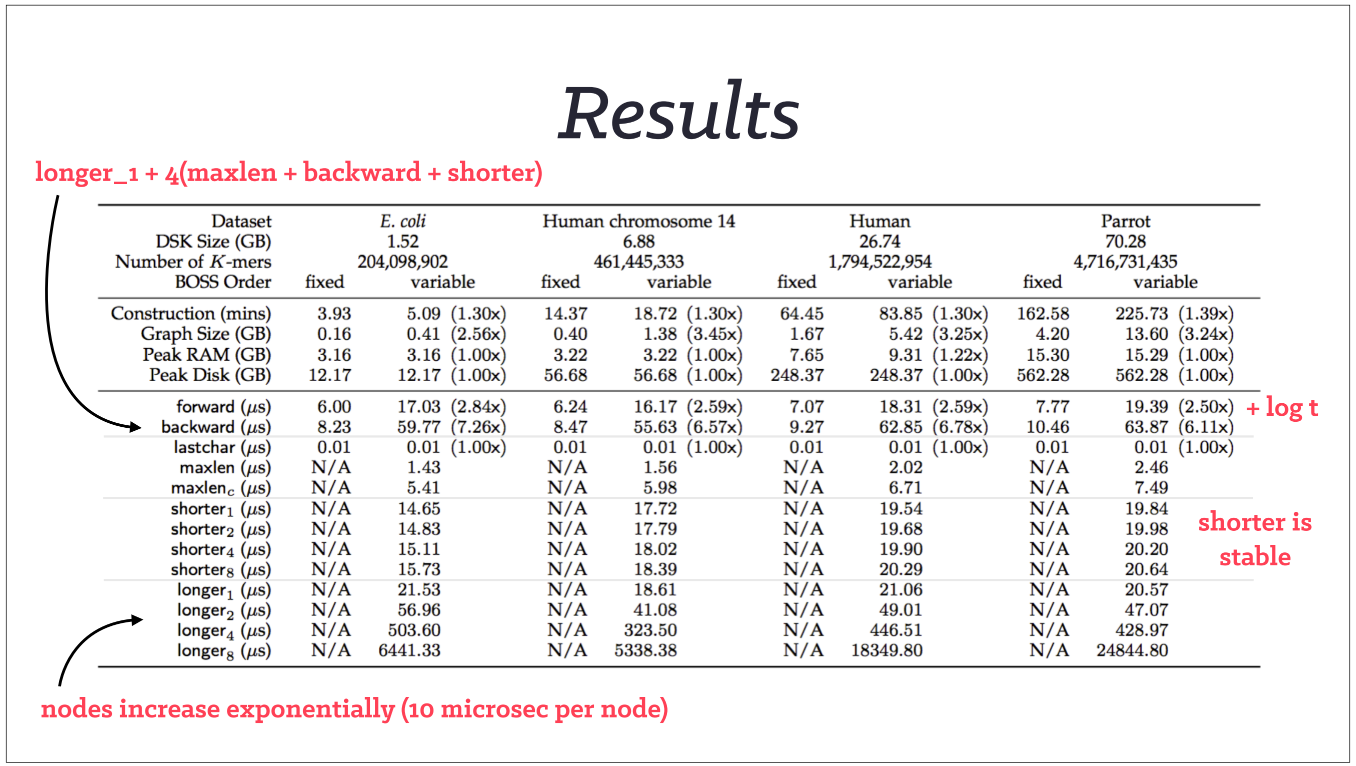
– measure shared prefix length to avoid duplication
– Only need one copy of satellite data (A)

Construction

- Use k-mer counter (DSK) to get unique k-mers, add reverse complements (RCs)
- Colex Sort (STXXL) two copies of the table in parallel: by source node (A), target node (B)
- B-A : nodes requiring outgoing dummies (*colex*) (= outgoing dummies in *reverse lex* order due to RCs). Delete B. can build LCP vector
- Generate $-shifts for incoming dummies ($ACG, $$AC, $$$A), avoiding duplicates using LCP values. Sort.
- 3-way merge A, O, and I, while streaming L and W, and LCS, Colour, (or any other data) to disk



– measure shared prefix length to avoid duplication
– Only need one copy of satellite data (A)



Results

longer_1 + 4(maxlen + backward + shorter)

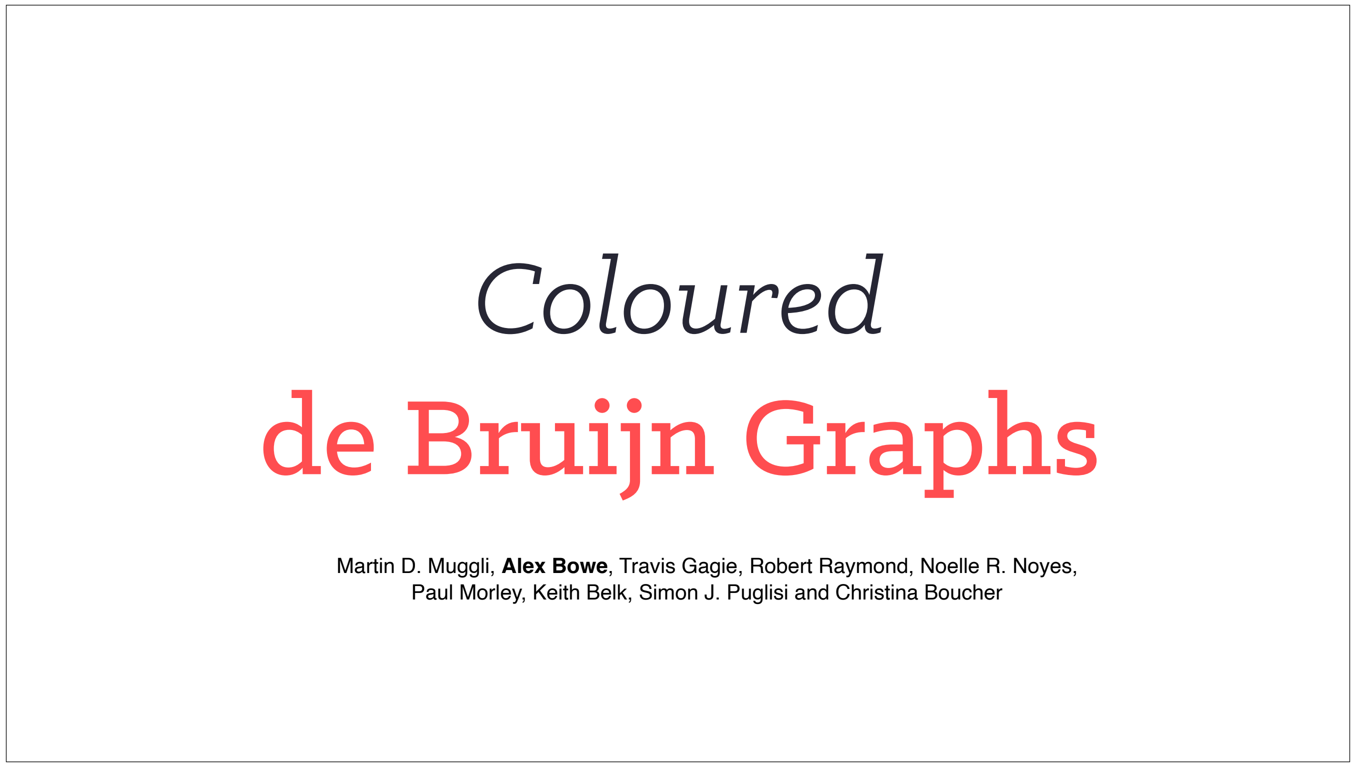
Dataset	*E. coli*		Human chromosome 14		Human		Parrot	
DSK Size (GB)	1.52		6.88		26.74		70.28	
Number of K-mers	204,098,902		461,445,333		1,794,522,954		4,716,731,435	
BOSS Order	fixed	variable	fixed	variable	fixed	variable	fixed	variable
Construction (mins)	3.93	5.09 (1.30x)	14.37	18.72 (1.30x)	64.45	83.85 (1.30x)	162.58	225.73 (1.39x)
Graph Size (GB)	0.16	0.41 (2.56x)	0.40	1.38 (3.45x)	1.67	5.42 (3.25x)	4.20	13.60 (3.24x)
Peak RAM (GB)	3.16	3.16 (1.00x)	3.22	3.22 (1.00x)	7.65	9.31 (1.22x)	15.30	15.29 (1.00x)
Peak Disk (GB)	12.17	12.17 (1.00x)	56.68	56.68 (1.00x)	248.37	248.37 (1.00x)	562.28	562.28 (1.00x)
forward (μs)	6.00	17.03 (2.84x)	6.24	16.17 (2.59x)	7.07	18.31 (2.59x)	7.77	19.39 (2.50x)
backward (μs)	8.23	59.77 (7.26x)	8.47	55.63 (6.57x)	9.27	62.85 (6.78x)	10.46	63.87 (6.11x)
lastchar (μs)	0.01	0.01 (1.00x)	0.01	0.01 (1.00x)	0.01	0.01 (1.00x)	0.01	0.01 (1.00x)
maxlen (μs)	N/A	1.43	N/A	1.56	N/A	2.02	N/A	2.46
maxlen_c (μs)	N/A	5.41	N/A	5.98	N/A	6.71	N/A	7.49
shorter_1 (μs)	N/A	14.65	N/A	17.72	N/A	19.54	N/A	19.84
shorter_2 (μs)	N/A	14.83	N/A	17.79	N/A	19.68	N/A	19.98
shorter_4 (μs)	N/A	15.11	N/A	18.02	N/A	19.90	N/A	20.20
shorter_8 (μs)	N/A	15.73	N/A	18.39	N/A	20.29	N/A	20.64
longer_1 (μs)	N/A	21.53	N/A	18.61	N/A	21.06	N/A	20.57
longer_2 (μs)	N/A	56.96	N/A	41.08	N/A	49.01	N/A	47.07
longer_4 (μs)	N/A	503.60	N/A	323.50	N/A	446.51	N/A	428.97
longer_8 (μs)	N/A	6441.33	N/A	5338.38	N/A	18349.80	N/A	24844.80

+ log t

shorter is stable

nodes increase exponentially (10 microsec per node)

- 30% from WT
- Most algorithms dont use backward anyway
- rest is pretty good (better than t)



Coloured

de Bruijn Graphs

Martin D. Muggli, **Alex Bowe**, Travis Gagie, Robert Raymond, Noelle R. Noyes,
Paul Morley, Keith Belk, Simon J. Puglisi and Christina Boucher

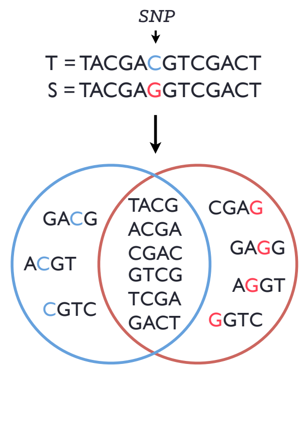
dropped decodable debruijn sequences, but still interested in doing this

Coloured De Bruijn Graphs

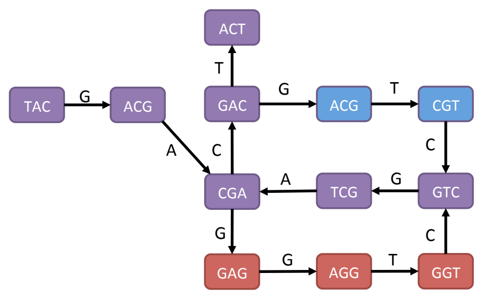
[Iqbal et al. 2012]

- *Goal:* Improve variant detection with (unassembled) genomes in a massive population.

- *Motivation:* pangenomic data increasing rapidly. Individual assemblies remove structure, takes time/memory.



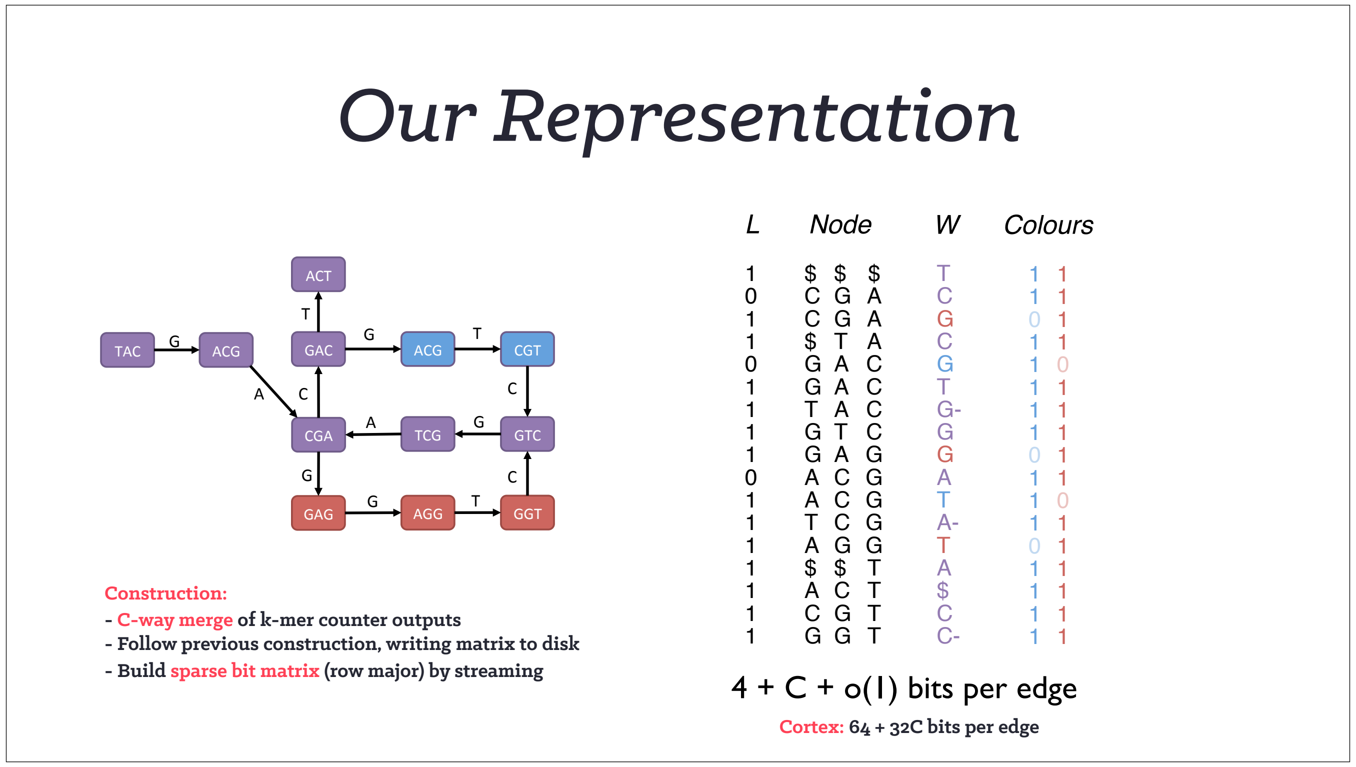
4-mers



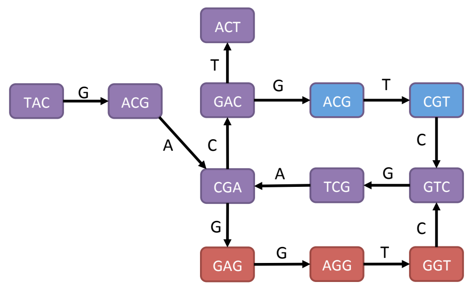
graph

Recent survey: "Computational pan-genomics: status, promises and challenges", Briefings in Bioinf., Oct 21

- here I've introduced another sample which has a single SNP
- Cortex: 2 algorithms… find branch, split by specified colours…



Our Representation



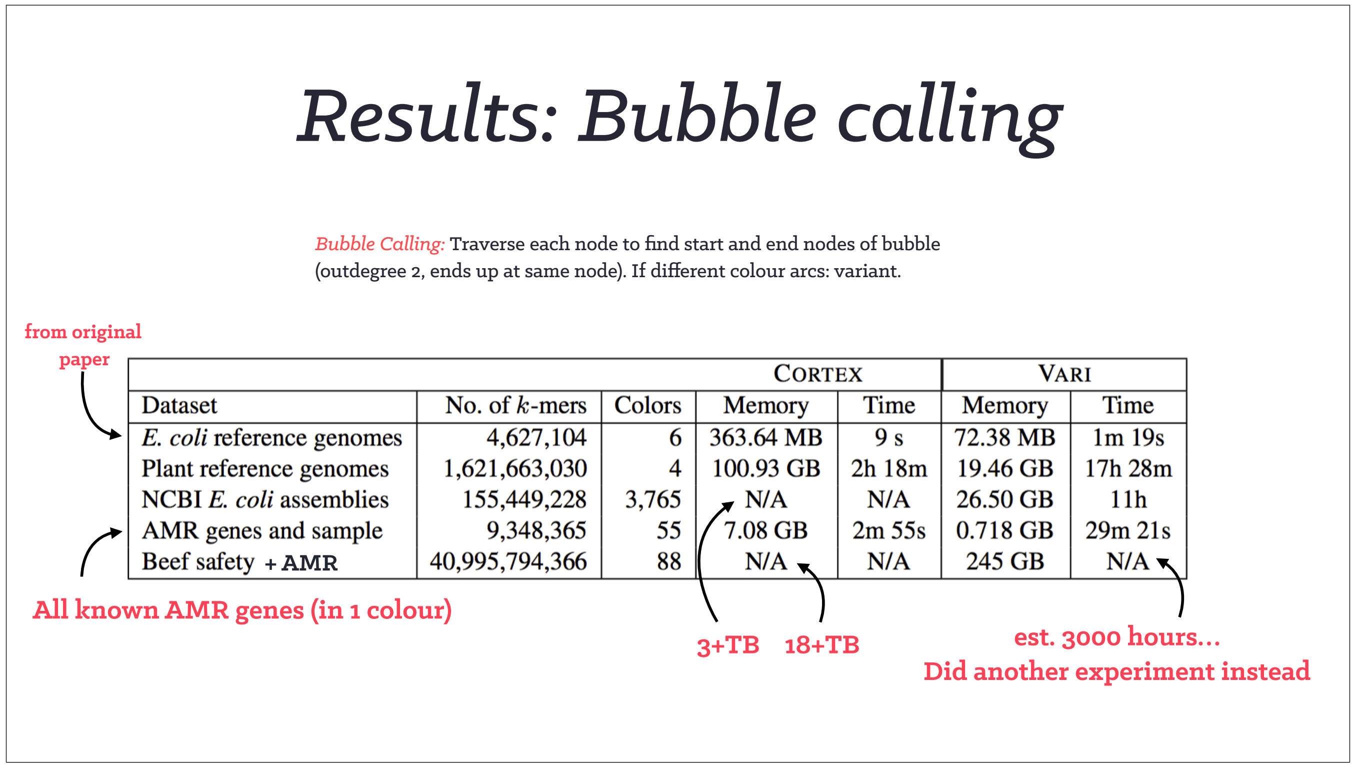
Construction:
- **C-way merge of k-mer counter outputs**
- **Follow previous construction, writing matrix to disk**
- **Build sparse bit matrix (row major) by streaming**

L	*Node*	*W*	*Colours*	
1	$ $ $	T	1	1
0	C G A	C	1	1
1	C G A	G	0	1
1	$ T A	C	1	1
0	G A C	G	1	0
1	G A C	T	1	1
1	T A C	G-	1	1
1	G T C	G	1	1
1	G A G	G	0	1
0	A C G	A	1	1
1	A C G	T	1	0
1	T C G	A-	1	1
1	A G G	T	0	1
1	$ $ T	A	1	1
1	A C T	$	1	1
1	C G T	C	1	1
1	G G T	C-	1	1

4 + C + o(1) bits per edge

Cortex: 64 + 32C bits per edge

- Really simple, but the varied construction algorithm helped make it
- access any colours in O(1), all colours in row in O(C)
- mostly 1s - sparse
- (4+C)m bits vs C4m bits



Results: Bubble calling

Bubble Calling: Traverse each node to find start and end nodes of bubble (outdegree 2, ends up at same node). If different colour arcs: variant.

from original paper

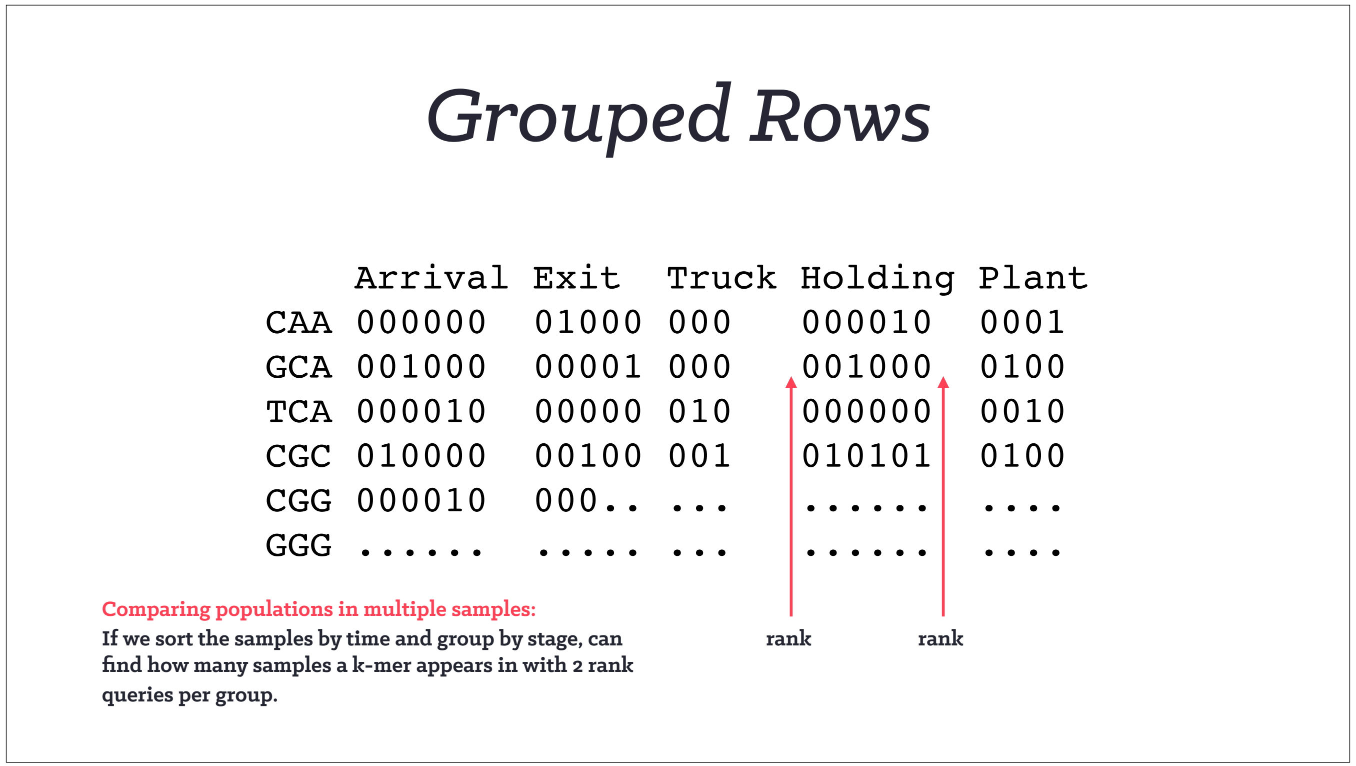
			Cortex		Vari	
Dataset	No. of k-mers	Colors	Memory	Time	Memory	Time
E. coli reference genomes	4,627,104	6	363.64 MB	9 s	72.38 MB	1m 19s
Plant reference genomes	1,621,663,030	4	100.93 GB	2h 18m	19.46 GB	17h 28m
NCBI *E. coli* assemblies	155,449,228	3,765	N/A	N/A	26.50 GB	11h
AMR genes and sample	9,348,365	55	7.08 GB	2m 55s	0.718 GB	29m 21s
Beef safety + AMR	40,995,794,366	88	N/A	N/A	245 GB	N/A

All known AMR genes (in 1 colour)

3+TB 18+TB

est. 3000 hours...
Did another experiment instead

- Beef supply chain stages



Grouped Rows

	Arrival	Exit	Truck	Holding	Plant
CAA	000000	01000	000	000010	0001
GCA	001000	00001	000	001000	0100
TCA	000010	00000	010	000000	0010
CGC	010000	00100	001	010101	0100
CGG	000010	000..	...	......	....
GGG	......	.....	...	......	....

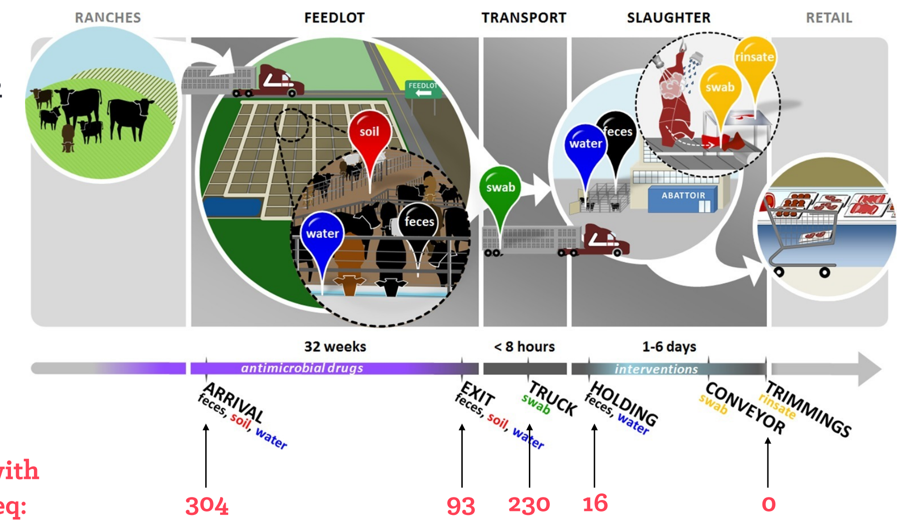
rank rank

Comparing populations in multiple samples:
If we sort the samples by time and group by stage, can find how many samples a k-mer appears in with 2 rank queries per group.

– A nice feature of row major
– Such as stages in a beef supply chain

Beef Safety/AMR

Approach: Traverse AMR graph using AMR genes as guide, count # of AMR genes (27 min)



# AMR genes with high k-mer freq:

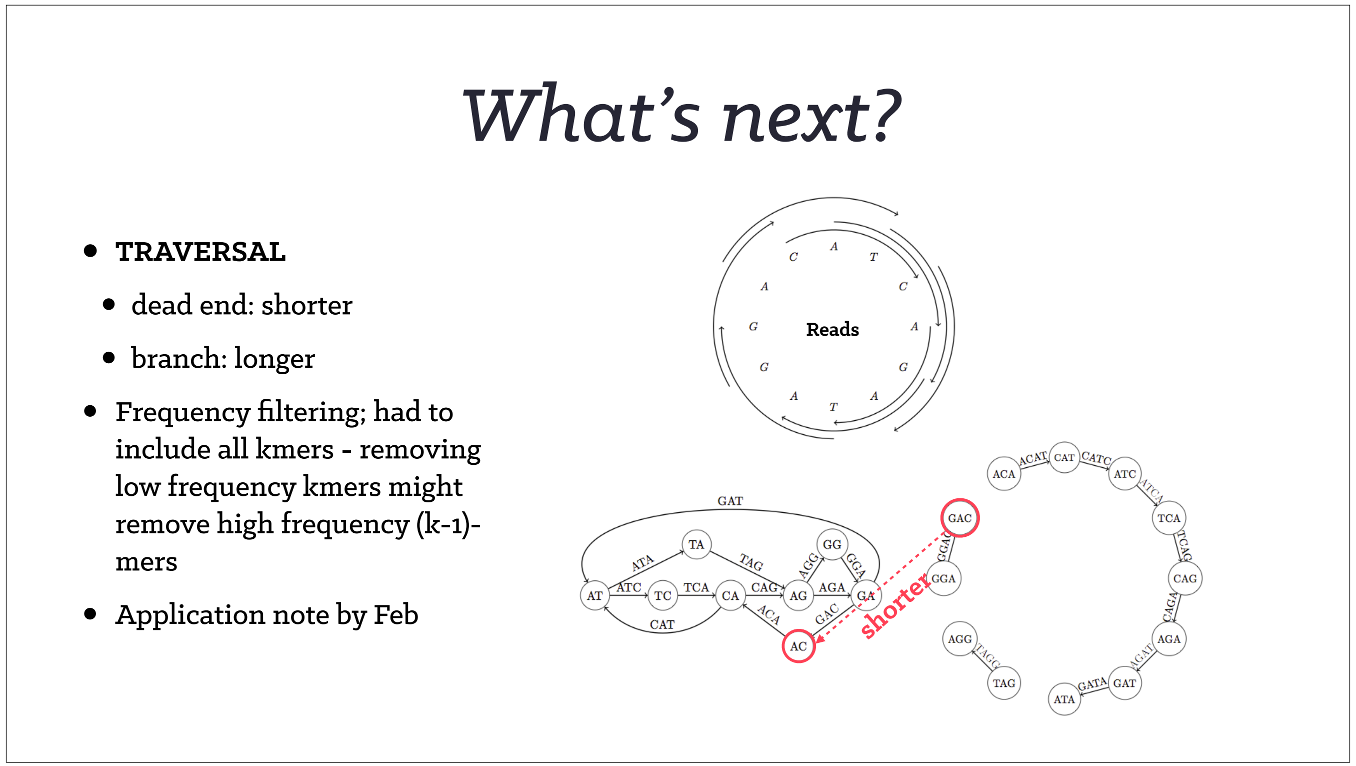
Hypothesis: Antimicrobial interventions at slaughter were effective.

- avoid allowing it to evolve
- Using beef reads as a guide, but canceling whenever we deviated from AMR

Summary

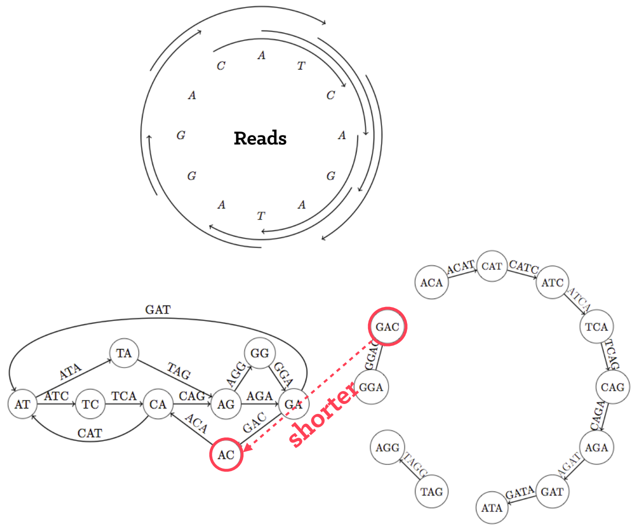
- Compact dBG (2 bits per edge), first BWT approach
- *V*ariable Order dBG
  - 3.5x size, 30% slower to build, 3x slower to traverse
  - New traversal methods. (Faster than IDBA?)
- New: coloured de Bruijn graph
  - slower, but 3TB -> 26.5 GB (e. coli), 18TB -> 245GB (beef safety)
  - Looking at colour per read, which is useful in some applications
- New: fast external construction
- Code: github.com/cosmo-team/cosmo

Thank you



What's next?

- **TRAVERSAL**
  - dead end: shorter
  - branch: longer
- Frequency filtering; had to include all kmers - removing low frequency kmers might remove high frequency (k-1)-mers
- Application note by Feb

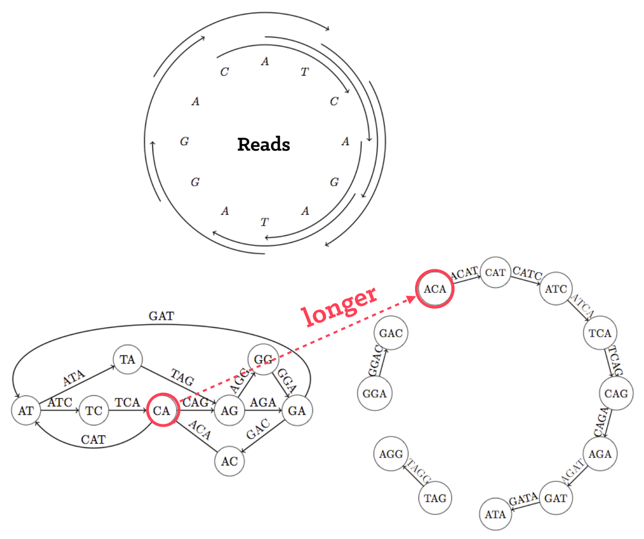


– Can probably find least branchy in log d time?

Can we do this by removing edges (preprocessing)? find all dummy edges, shorten them until one row, then…

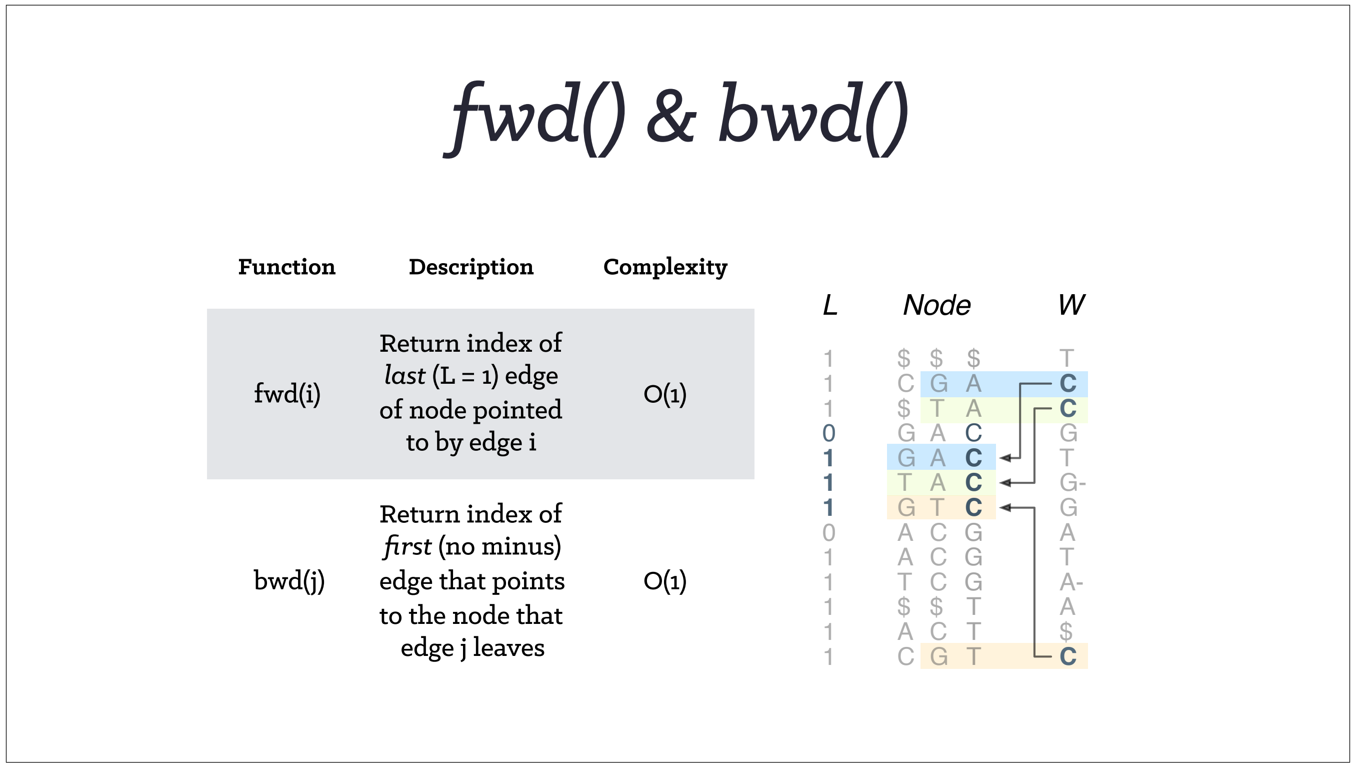
What's next?

- **TRAVERSAL**
  - dead end: shorter
  - branch: longer
- Frequency filtering; had to include all kmers - removing low frequency kmers might remove high frequency (k-1)-mers
- Application note by Feb



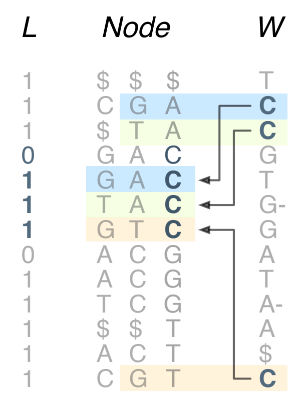
Use traversal history to determine prefix
(longer can be sped up?)

(preprocessing?)



fwd() & bwd()

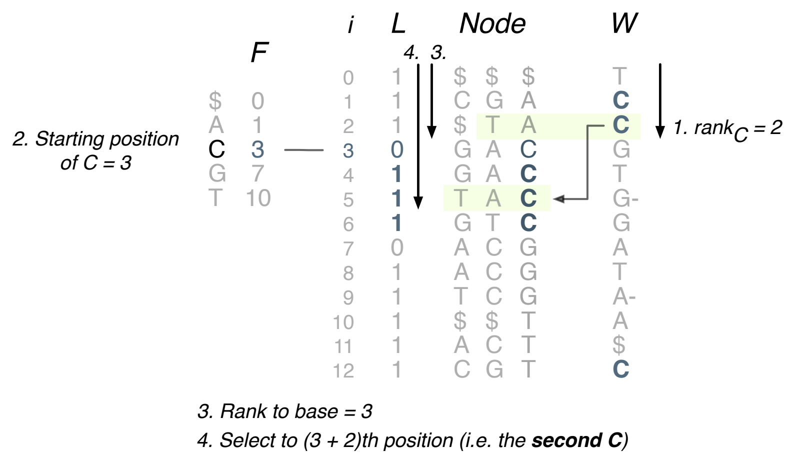
Function	**Description**	**Complexity**
fwd(i)	Return index of *last* (L = 1) edge of node pointed to by edge i	O(1)
bwd(j)	Return index of *first* (no minus) edge that points to the node that edge j leaves	O(1)



L	*Node*			*W*
1	$	$	$	T
1	C	G	A	**C**
1	$	T	A	**C**
0	G	A	C	G
1	G	A	**C**	T
1	T	A	**C**	G-
1	G	T	**C**	G
0	A	C	G	A
1	A	C	G	T
1	T	C	G	A-
1	$	$	T	A
1	A	C	T	$
1	C	G	T	**C**

defined over edges
- fwd -> last edge of the destination node (1)
- bwd -> from 1, first predecessor edge (no minus)

Example: fwd(2)



F	
$	0
A	1
C	3
G	7
T	10

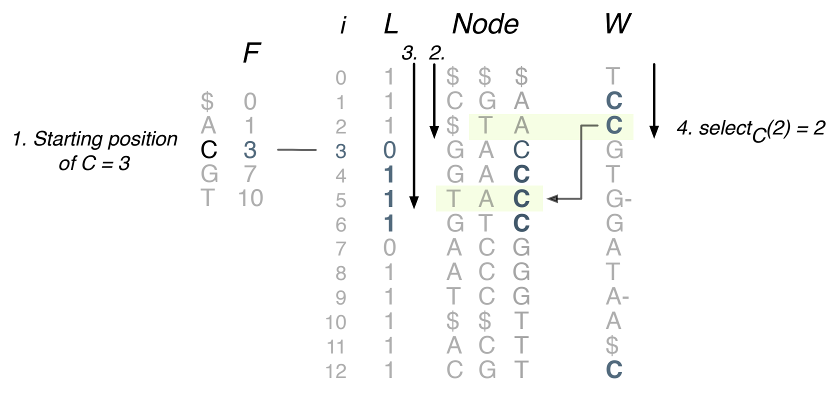
i	L	Node			W
0	1	$	$	$	T
1	1	C	G	A	**C**
2	1	$	T	A	**C**
3	0	G	A	C	G
4	**1**	G	A	**C**	T
5	**1**	T	A	**C**	G-
6	**1**	G	T	**C**	G
7	0	A	C	G	A
8	1	A	C	G	T
9	1	T	C	G	A-
10	1	$	$	T	A
11	1	A	C	T	$
12	1	C	G	T	**C**

3. Rank to base = 3

*4. Select to (3 + 2)th position (i.e. the **second C**)*

bwd is done in a similar fashion, but going from F to W instead.

Example: bwd(5)



F	
$	0
A	1
C	3
G	7
T	10

i	*L*	*Node*			*W*
0	1	$	$	$	T
1	1	C	G	A	**C**
2	1	$	T	A	**C**
3	0	G	A	**C**	G
4	**1**	G	A	**C**	T
5	**1**	T	A	**C**	G-
6	**1**	G	T	**C**	G
7	0	A	C	G	A
8	1	A	C	G	T
9	1	T	C	G	A-
10	1	$	$	T	A
11	1	A	C	T	$
12	1	C	G	T	**C**

2. Rank to base = 3
3.1. Rank to current edge = 5,
3.2. 5 - 3 = 2, so we are at the ***second C***

bwd is done in a similar fashion, but going from F to W instead.

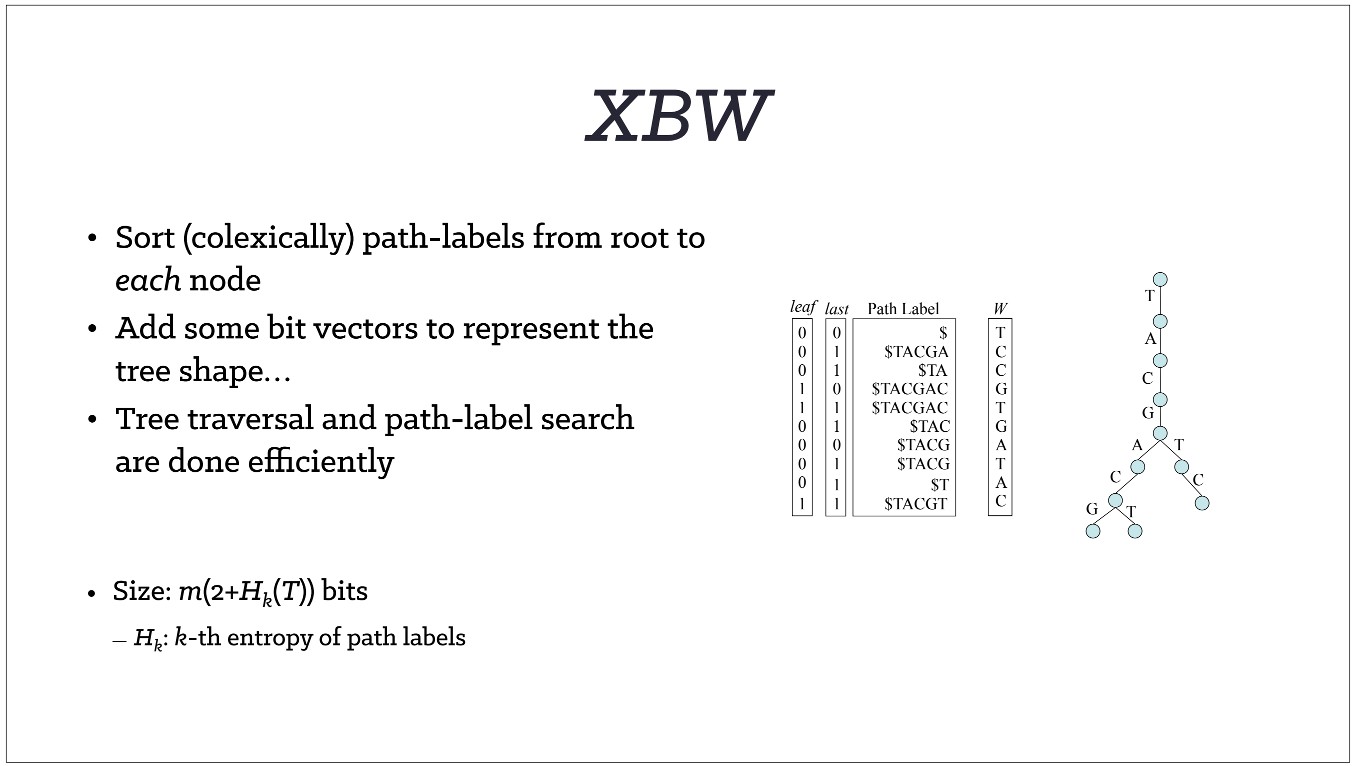
BWT

- Burrows Wheeler Transform
- Permute letters in a string in lexicographic
  order of their reversed prefixes
  (‘colex’ order)

- Size: $nH_k(T) \leq n \log \boldsymbol{\sigma}$
  H_k: k-th entropy of T
- Using only W, search and decode are possible

Prefix	W
$	T
$TACGA	C
$TACGACGTCGA	C
$TA	C
$TACGAC	G
$TACGACGTCGAC	T
$TAC	G
$TACGACGTC	G
$TACGACG	T
$TACG	A
$TACGACGTCG	A
$T	A
$TACGACGTCGACT	$
$TACGACGT	C

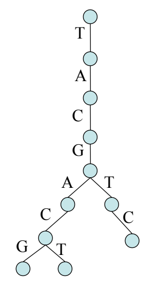
T = $TACGACGTCGACT



XBW

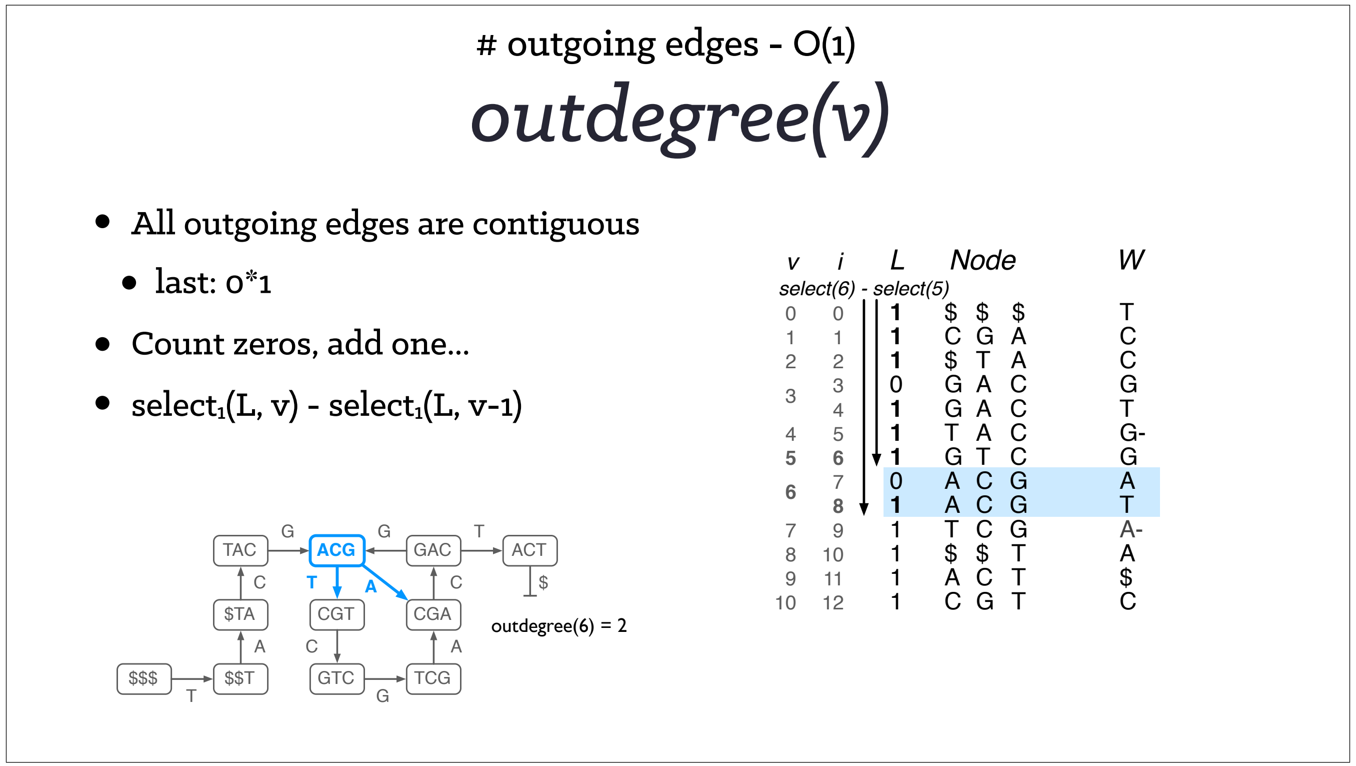
- Sort (colexically) path-labels from root to *each* node
- Add some bit vectors to represent the tree shape…
- Tree traversal and path-label search are done efficiently

leaf	*last*	Path Label	*W*
0	0	$	T
0	1	$TACGA	C
0	1	$TA	C
1	0	$TACGAC	G
1	1	$TACGAC	T
0	1	$TAC	G
0	0	$TACG	A
0	1	$TACG	T
0	1	$T	A
1	1	$TACGT	C



- Size: $m(2+H_k(T))$ bits
  - H_k: k-th entropy of path labels

– inspired our impl.



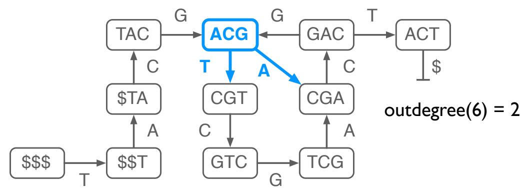
# outgoing edges - O(1)

outdegree(v)

- All outgoing edges are contiguous
    - last: 0*1
- Count zeros, add one...
- select$_1$(L, v) - select$_1$(L, v-1)

select(6) - select(5)

v	*i*	*L*	*Node*			*W*
0	0	**1**	$	$	$	T
1	1	**1**	C	G	A	C
2	2	**1**	$	T	A	C
3	3	0	G	A	C	G
	4	**1**	G	A	C	T
4	5	**1**	T	A	C	G-
5	**6**	**1**	G	T	C	G
6	7	0	A	C	G	A
	8	**1**	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

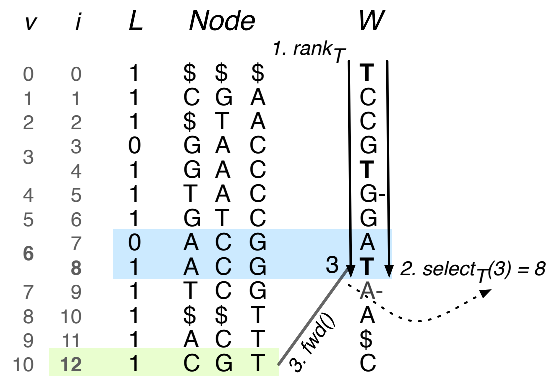


– find the difference between positions of 1s follow edge c from v - O(1)

outgoing(v, c)

- Search outgoing edges for c or c$^-$
- select(rank(i)): position of last occ. in the range [0,i]
- x = select$_c$(W, rank$_c$(W, v))
- if not in outgoing-edge range (known from outdegree) try c$^-$
- return fwd(x)

outgoing(6, T)



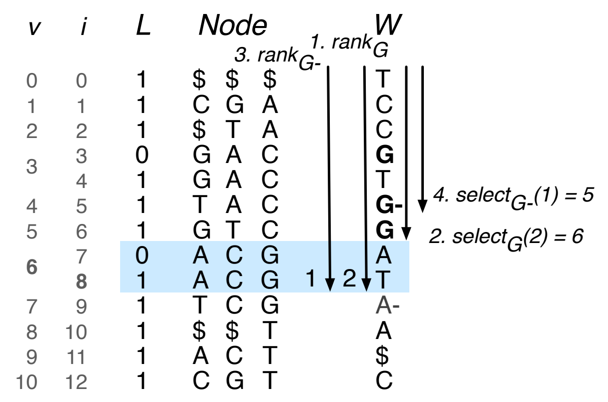
v	*i*	*L*	*Node*			*W*
0	0	1	$	$	$	**T**
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	G
	4	1	G	A	C	**T**
4	5	1	T	A	C	G-
5	6	1	G	T	C	G
6	7	0	A	C	G	A
	8	1	A	C	G	**T**
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	**12**	1	C	G	T	C

idea: find node 6 row (select) to last row
select a rank: find position of last in that range

follow edge c from v - O(1)

outgoing(v, c)

outgoing(6, G) - no outgoing G or G- edge



v	*i*	*L*	*Node*			*W*
0	0	1	$	$	$	T
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	**G**
	4	1	G	A	C	T
4	5	1	T	A	C	**G-**
5	6	1	G	T	C	**G**
6	7	0	A	C	G	A
	8	1	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

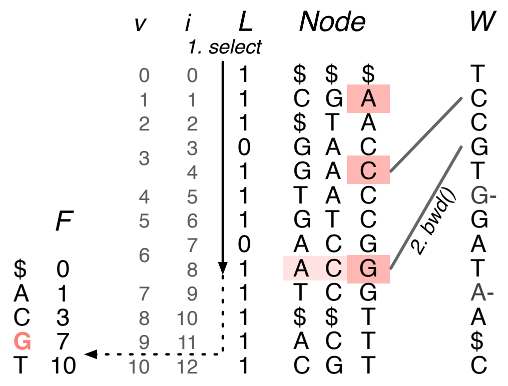
1. $rank_G$
3. $rank_{G-}$
4. $select_{G-}(1) = 5$
2. $select_G(2) = 6$
1
2

first rank and select to find G, no G...

return label of v - O(k)

node(v)

- Reminder: Node[] not stored, but last (kth) letter is stored in F.
- Calculate bwd() k times, using resulting indexes to reverse lookup in F.



F	
$	0
A	1
C	3
G	7
T	10

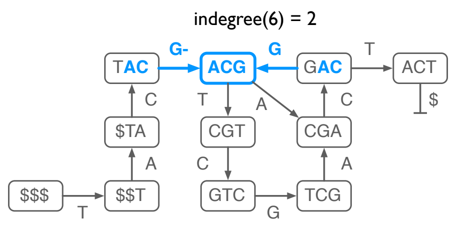
v	i	L	Node			W
0	0	1	$	$	$	T
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	G
	4	1	G	A	C	T
4	5	1	T	A	C	G-
5	6	1	G	T	C	G
6	7	0	A	C	G	A
	8	1	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

- select over last to find index, then find F_inv(i)
- assume F_inv O(1) lookup?

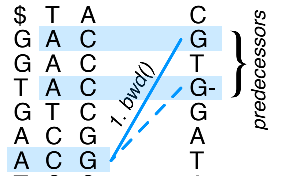
# incoming edges - O(1)

indegree(v)

- use bwd() to give us *first* predecessor node
- if there are more, they sort after as c-
- rank/select to next c, subtract ranks of c- to find degree



v	i	L	Node			W
0	0	1	$	$	$	T
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	G
	4	1	G	A	C	T
4	5	1	T	A	C	G-
5	6	1	G	T	C	G
6	7	0	A	C	G	A
	8	1	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

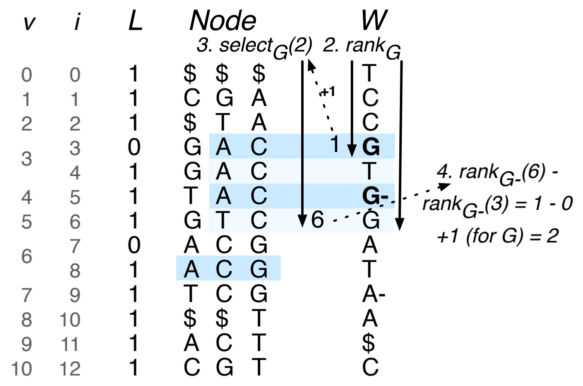


- rank how many Gs before,
- select the next G
rank the number of G-s in between

# incoming edges - O(1)

indegree(v)

- x = rank to count previous c edges
- select(x+1) finds *next* c edge
- rank c- to count occurrences between



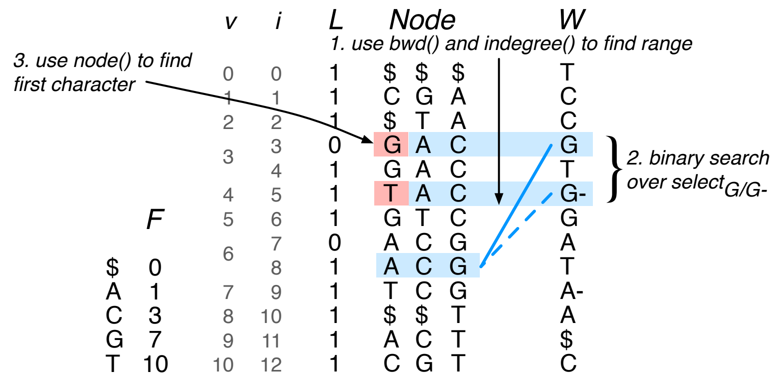
v	i	L	Node			W
0	0	1	$	$	$	T
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	**G**
	4	1	G	A	C	T
4	5	1	T	A	C	**G-**
5	6	1	G	T	C	G
6	7	0	A	C	G	A
	8	1	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

predecessor of v starting with c - O(k log **σ**)

incoming(v, c)

- Similar to indegree() to locate predecessors
- These nodes differ only in their first character, and are sorted

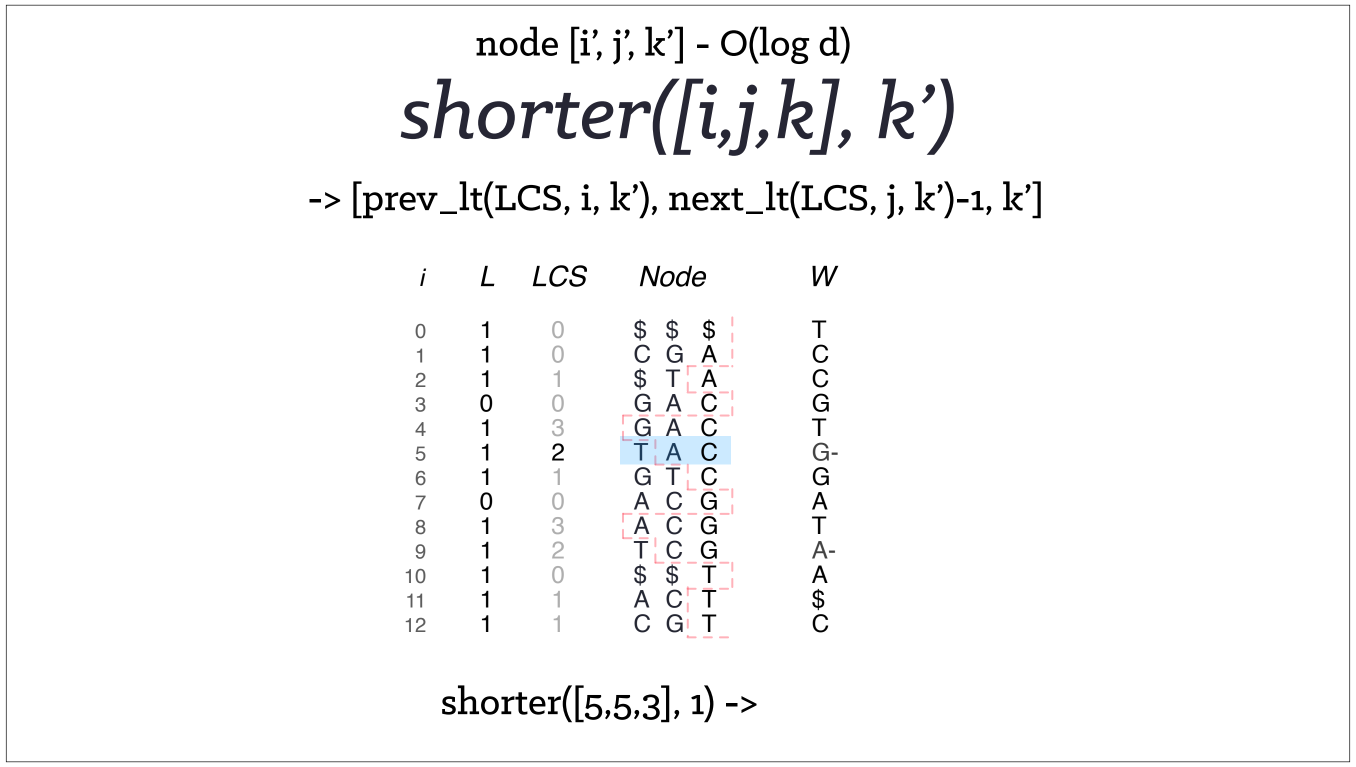
- Can use node() to find first character - O(k)
- binary search: O(log **σ**) time



3. use node() to find first character
1. use bwd() and indegree() to find range
2. binary search over select$_{G/G-}$

F	
$	0
A	1
C	3
G	7
T	10

v	*i*	*L*	*Node*			*W*
0	0	1	$	$	$	T
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	G
	4	1	G	A	C	T
4	5	1	T	A	C	G-
5	6	1	G	T	C	G
6	7	0	A	C	G	A
	8	1	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C



node [i’, j’, k’] - O(log d)

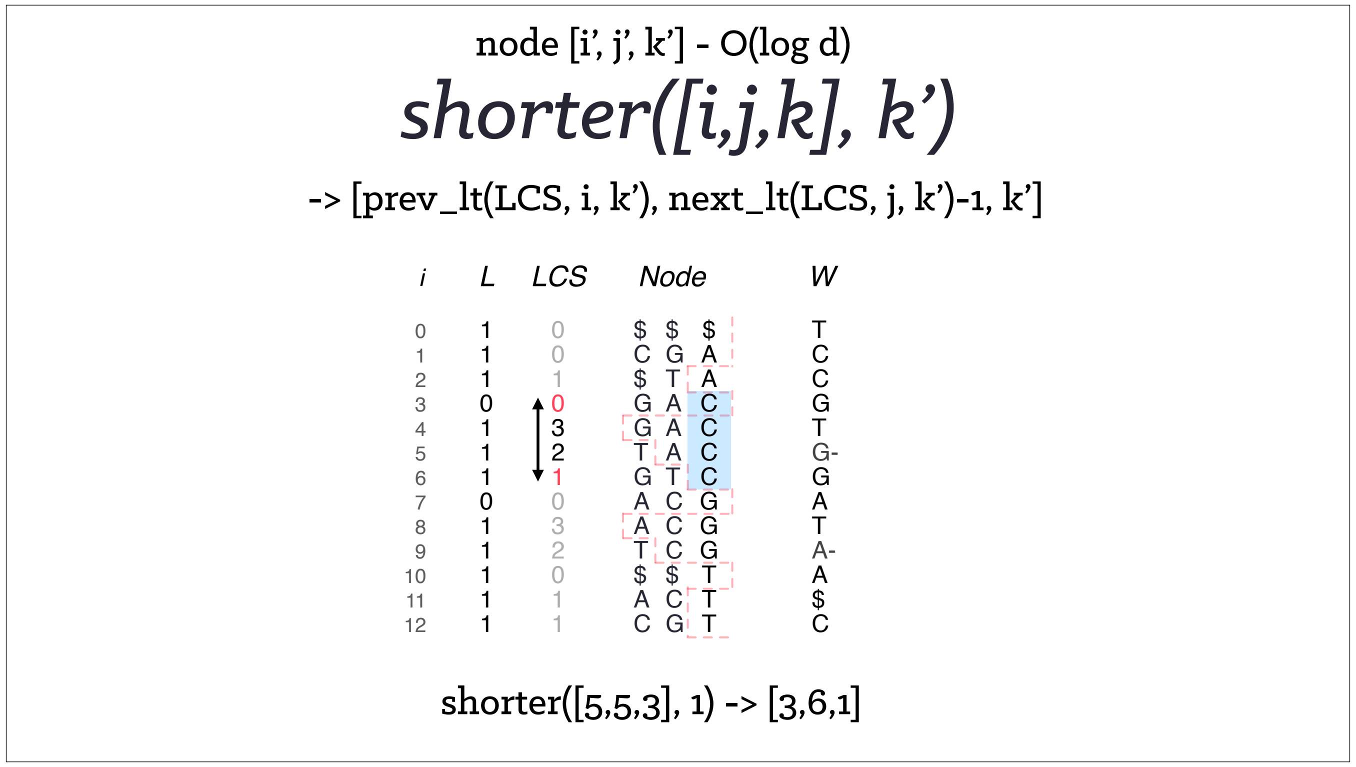
shorter([i,j,k], k’)

-> [prev_lt(LCS, i, k’), next_lt(LCS, j, k’)-1, k’]

i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	T
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

shorter([5,5,3], 1) ->

- Wavelet tree -> O(log d)



node [i', j', k'] - O(log d)

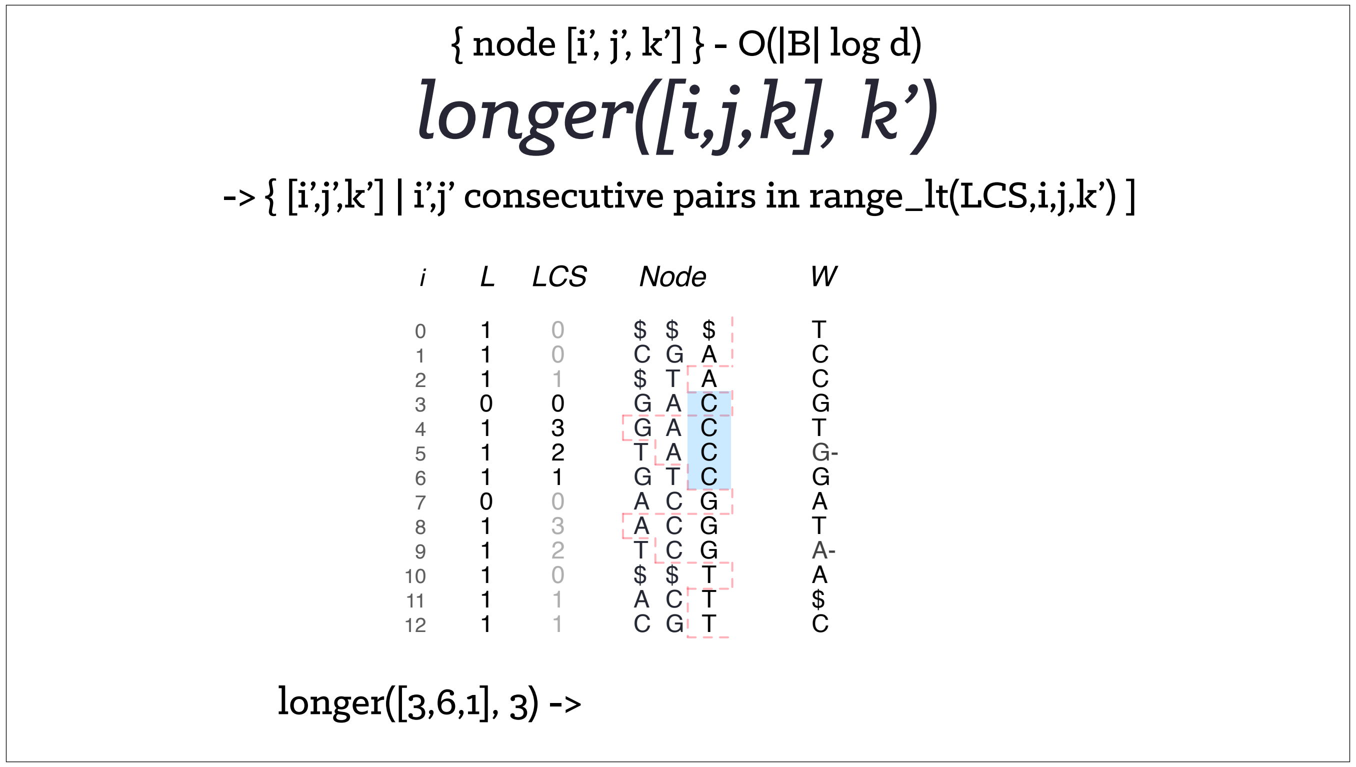
shorter([i,j,k], k')

-> [prev_lt(LCS, i, k'), next_lt(LCS, j, k')-1, k']

i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	T
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

shorter([5,5,3], 1) -> [3,6,1]

- Some corner cases, but thats the basic idea



{ node [i’, j’, k’] } - O(|B| log d)

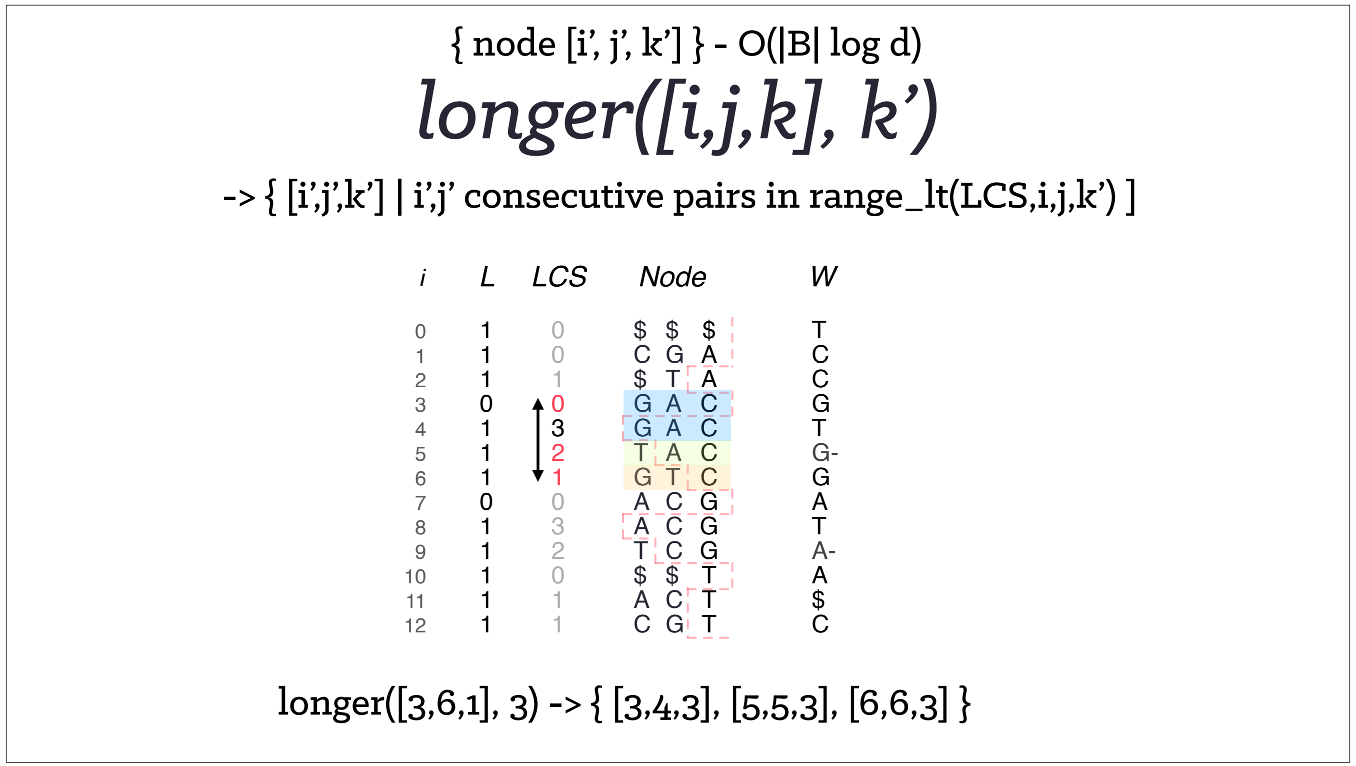
longer([i,j,k], k’)

-> { [i’,j’,k’] | i’,j’ consecutive pairs in range_lt(LCS,i,j,k’)]

i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	T
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

longer([3,6,1], 3) ->

- B is the set of nodes that have the same suffix
- this will make sense



{ node [i', j', k'] } - O(|B| log d)

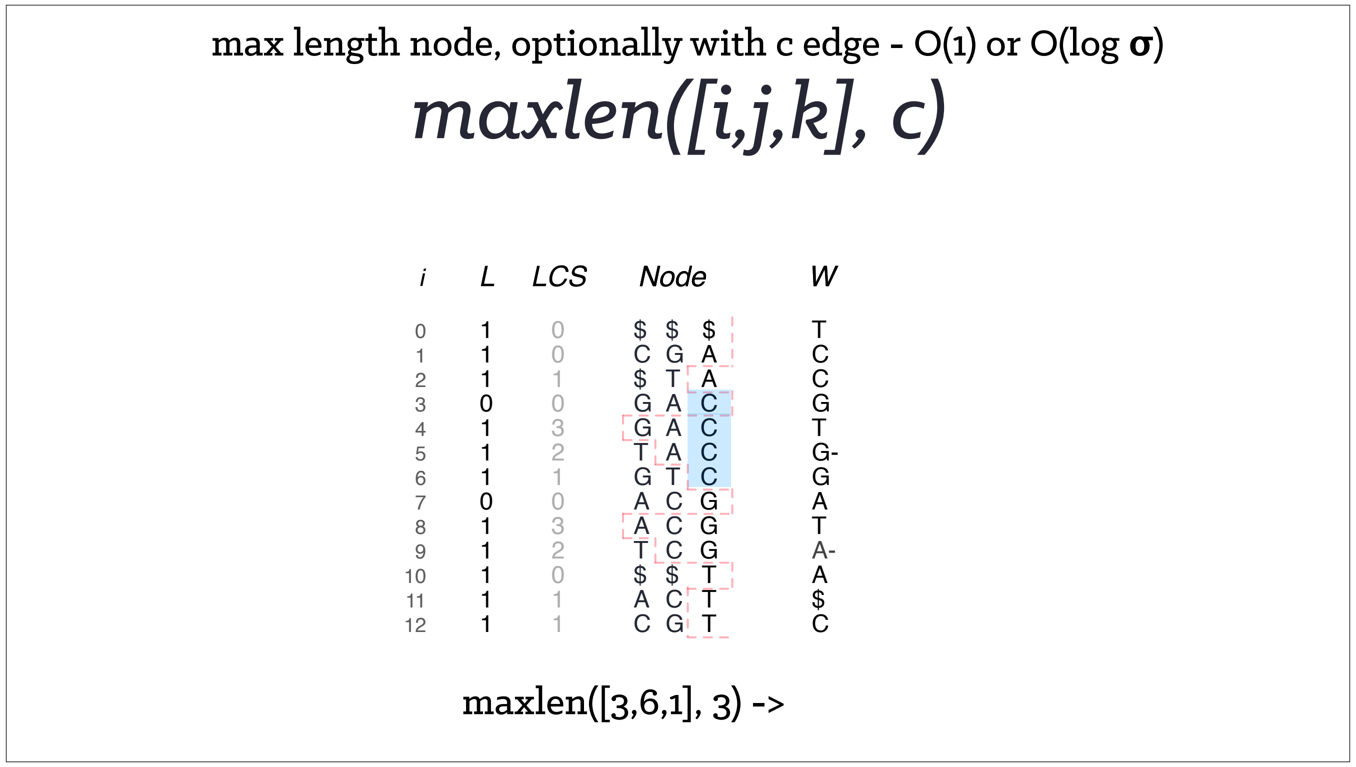
longer([i,j,k], k')

-> { [i',j',k'] | i',j' consecutive pairs in range_lt(LCS,i,j,k')]

i	*L*	*LCS*	*Node*	*W*
0	1	0	$ $ $	T
1	1	0	C G A	C
2	1	1	$ T A	C
3	0	0	G A C	G
4	1	3	G A C	T
5	1	2	T A C	G-
6	1	1	G T C	G
7	0	0	A C G	A
8	1	3	A C G	T
9	1	2	T C G	A-
10	1	0	$ $ T	A
11	1	1	A C T	$
12	1	1	C G T	C

longer([3,6,1], 3) -> { [3,4,3], [5,5,3], [6,6,3] }

- |B| = 3
- but can increase exponentially (prefix of node label)



max length node, optionally with c edge - O(1) or O(log **σ**)

maxlen([i,j,k], c)

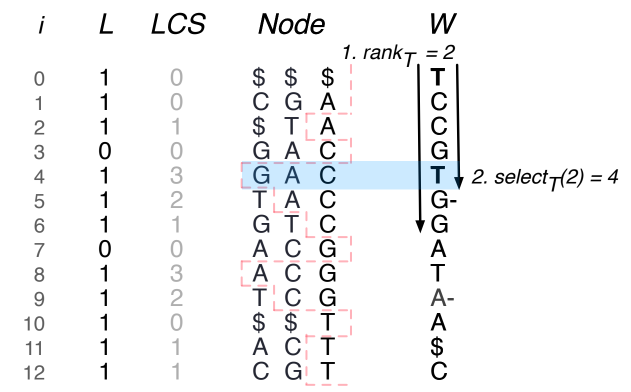
i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	T
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

maxlen([3,6,1], 3) ->

- we support fwd and bwd using a function called maxlen

max length node, optionally with c edge - O(log **σ**)

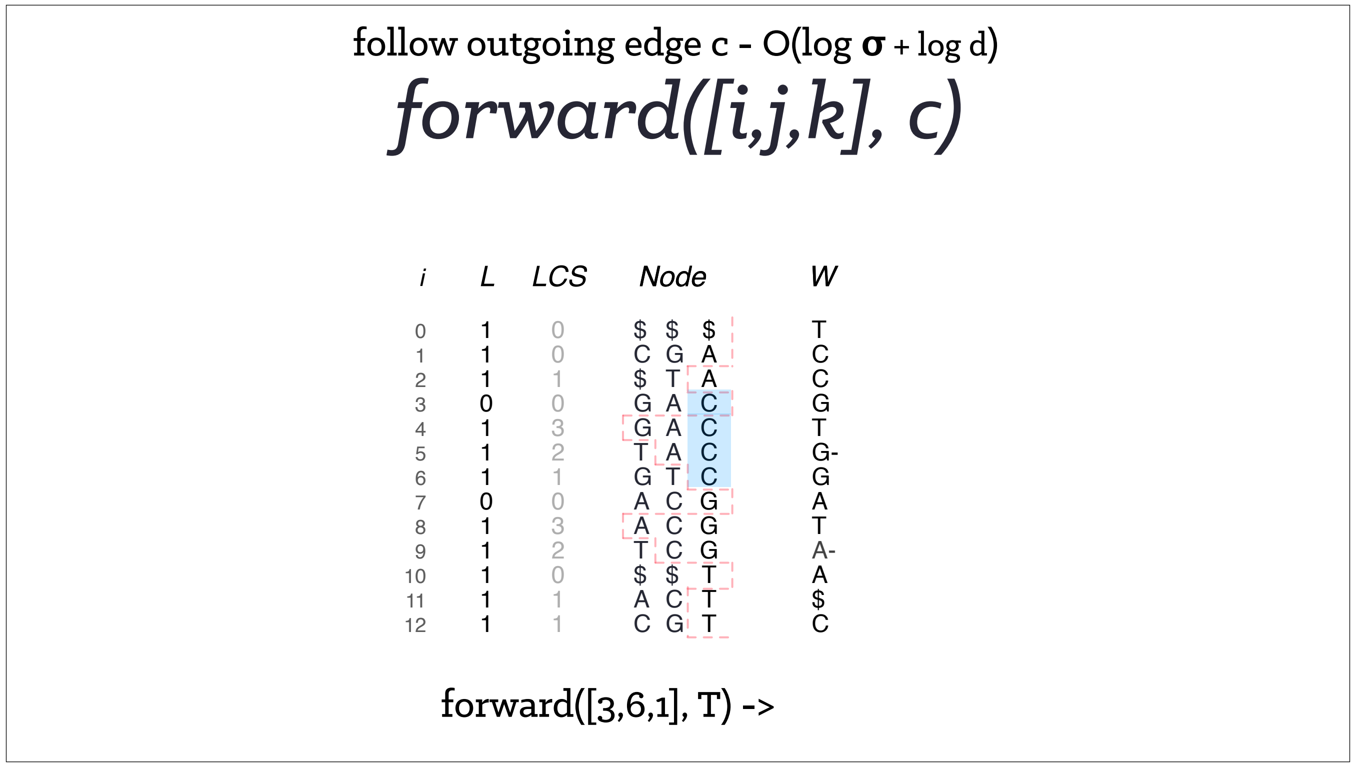
maxlen([i,j,k], c)



i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	**T**
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	**T**
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

maxlen([3,6,1], 3) -> [4,4,3]

- If we dont care about c, dont rank/sel - O(1)
- Can easily return node if needed (3,4), but we dont need it.



follow outgoing edge c - O(log **σ** + log d)

forward([i,j,k], c)

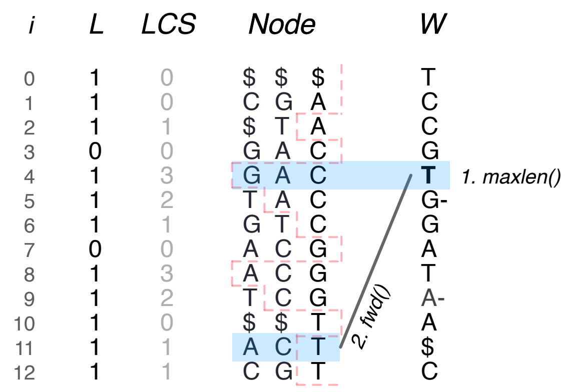
i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	T
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

forward([3,6,1], T) ->

- we support fwd and bwd using a function called maxlen

follow outgoing edge c - O(log **σ** + log d)

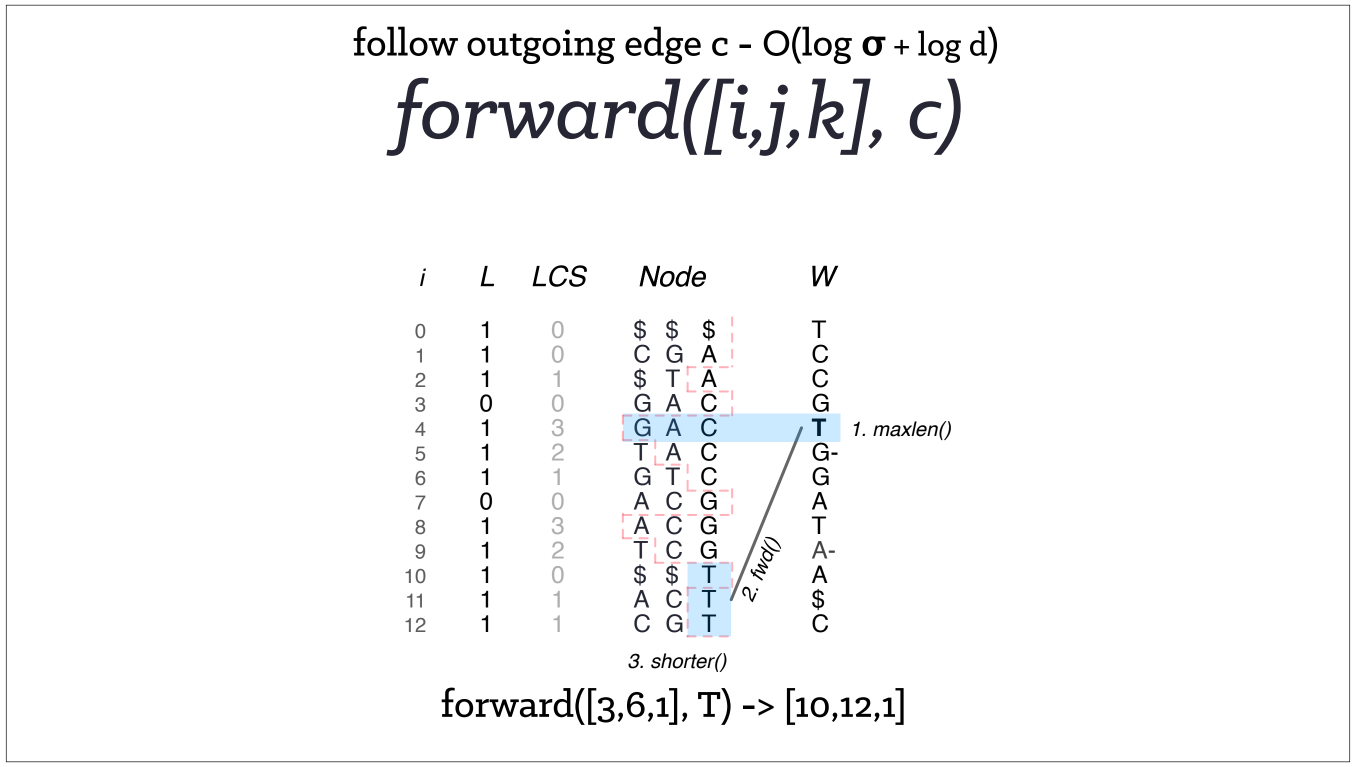
forward([i,j,k], c)



i	*L*	*LCS*	*Node*	*W*
0	1	0	$ $ $	T
1	1	0	C G A	C
2	1	1	$ T A	C
3	0	0	G A C	G
4	1	3	G A C	**T**
5	1	2	T A C	G-
6	1	1	G T C	G
7	0	0	A C G	A
8	1	3	A C G	T
9	1	2	T C G	A-
10	1	0	$ $ T	A
11	1	1	A C T	$
12	1	1	C G T	C

forward([3,6,1], T) ->

- we support fwd and bwd using a function called maxlen



follow outgoing edge c - O(log **σ** + log d)

forward([i,j,k], c)

i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	**T**
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

forward([3,6,1], T) -> [10,12,1]

- we support fwd and bwd using a function called maxlen

predecessors - O(**σ** log d)

backward([i,j,k])

i	*L*	*LCS*	*Node*	*W*
0	1	0	$ $ $	T
1	1	0	C G A	C
2	1	1	$ T A	C
3	0	0	G A C	G
4	1	3	G A C	T
5	1	2	T A C	G-
6	1	1	G T C	G
7	0	0	A C G	A
8	1	3	A C G	T
9	1	2	T C G	A-
10	1	0	$ $ T	A
11	1	1	A C T	$
12	1	1	C G T	C

backward([3,6,1]) ->

predecessors - O(**σ** log d)

backward([i,j,k])

i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	T
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

1. longer(k+1)

backward([3,6,1]) ->

predecessors - O(**σ** log d)

backward([i,j,k])

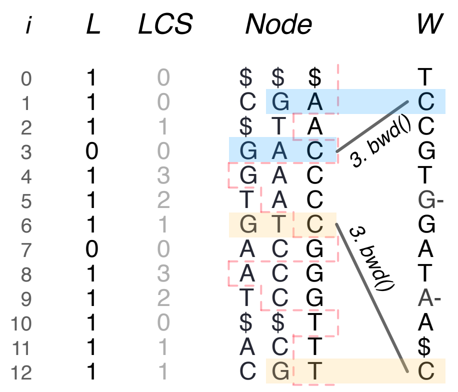
i	*L*	*LCS*	*Node*	*W*
0	1	0	$ $ $	T
1	1	0	C G A	C
2	1	1	$ T A	C
3	0	0	G A C	G
4	1	3	G A C	T
5	1	2	T A C	G-
6	1	1	G T C	G
7	0	0	A C G	A
8	1	3	A C G	T
9	1	2	T C G	A-
10	1	0	$ $ T	A
11	1	1	A C T	$
12	1	1	C G T	C

2. maxlen()

backward([3,6,1]) ->

predecessors - O(**σ** log d)

backward([i,j,k])



i	L	LCS	Node	W
0	1	0	$ $ $	T
1	1	0	C G A	C
2	1	1	$ T A	C
3	0	0	G A C	G
4	1	3	G A C	T
5	1	2	T A C	G-
6	1	1	G T C	G
7	0	0	A C G	A
8	1	3	A C G	T
9	1	2	T C G	A-
10	1	0	$ $ T	A
11	1	1	A C T	$
12	1	1	C G T	C

backward([3,6,1]) ->

predecessors - O(**σ** log d)

backward([i,j,k])

i	*L*	*LCS*	*Node*			*W*
0	1	0	$	$	$	T
1	1	0	C	G	A	C
2	1	1	$	T	A	C
3	0	0	G	A	C	G
4	1	3	G	A	C	T
5	1	2	T	A	C	G-
6	1	1	G	T	C	G
7	0	0	A	C	G	A
8	1	3	A	C	G	T
9	1	2	T	C	G	A-
10	1	0	$	$	T	A
11	1	1	A	C	T	$
12	1	1	C	G	T	C

4. shorter()

backward([3,6,1]) -> {[1,2,1], [10,12,1]}