Computational Pangenomics #CPANG19

Day 4 (September 12, 2018)

Erik Garrison and Mikko Rautiainen

Wrap up of day 3

New vg commands: index (sorting), explode, chunk, pack.

Bacterial pangenes and pangenomes.

vg index -a (start-node sorted alignment index)

vg construct -r small/x.fa -v small/x.vcf.gz >x.vg
vg index -x x.xg -g x.gcsa -k 16 x.vg
vg map -d x -G <(vg sim -n 100 -e 0.01 -i 0.005 -l 50 -a -x x.xg) >aln.gam
vg index -d aln.gam.idx -a aln.gam
vg index -d aln.gam.idx -D


```
{"key":"+a+26+0", "value":{"refpos": [{"is_reverse": true, "offset": 103, "name": "x"}], "identity
{"key":"+a+26+0", "value":{"refpos": [{"is_reverse": true, "offset": 103, "name": "x"}], "identity
{"key":"+a+32+0", "value":{"refpos": [{"is_reverse": true, "offset": 142, "name": "x"}], "identity
{"key":"+a+36+0", "value":{"refpos": [{"is_reverse": true, "offset": 172, "name": "x"}], "identity
{"key":"+a+42+0", "value":{"refpos": [{"offset": 186, "name": "x"}], "identity": 1.0, "sequence":
{"key":"+a+43+0", "value":{"refpos": [{"offset": 189, "name": "x"}], "identity": 1.0, "sequence":
{"key":"+a+46+0", "value":{"refpos": [{"offset": 201, "name": "x"}], "identity": 1.0, "sequence":
{"key":"+a+49+0", "value":{"refpos": [{"is_reverse": true, "offset": 204, "name": "x"}], "identity
{"key":"+a+52+0", "value":{"refpos": [{"offset": 219, "name": "x"}], "identity": 0.979999999999999
{"key":"+a+53+0", "value":{"refpos": [{"is_reverse": true, "offset": 222, "name": "x"}], "identity
{"key":"+a+55+0", "value":{"refpos": [{"is_reverse": true, "offset": 221, "name": "x"}], "identity
{"key":"+a+55+0", "value":{"refpos": [{"is_reverse": true, "offset": 221, "name": "x"}], "identity
{"key":"+a+55+0", "value":{"refpos": [{"offset": 255, "name": "x"}], "identity": 0.979999999999999
{"key":"+a+55+0", "value":{"refpos": [{"is_reverse": true, "offset": 221, "name": "x"}], "identity
{"key":"+a+55+0", "value":{"refpos": [{"is_reverse": true, "offset": 221, "name": "x"}], "identity
```

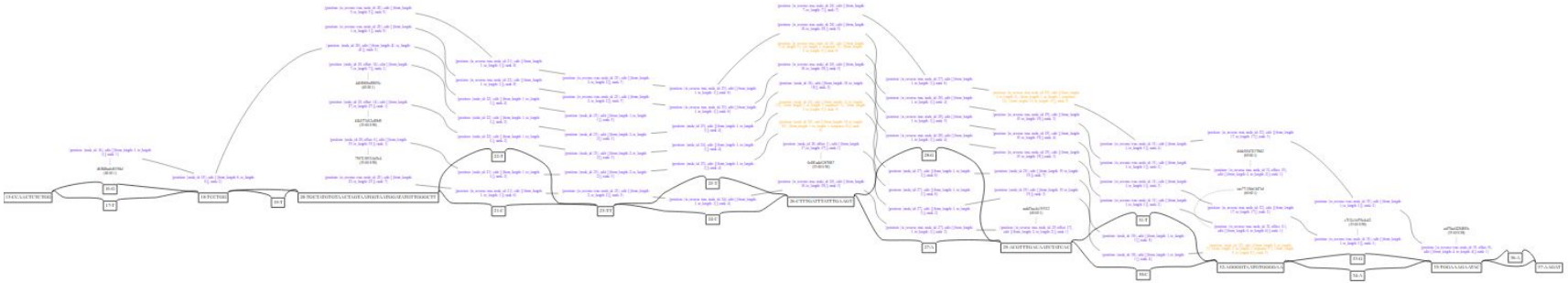
Sorting alignments (by start node id)

```
vg index -A -d aln.gam.idx | vg view -a -
vg index -A -d aln.gam.idx >aln.sort.gam
vg view -a aln.sort.gam | jq '.path.mapping[0].position.node_id' | head
```

```
16
18
20
20
20
22
16
16
32
32
….
```

vg index -N (node to alignment index)

```
vg index -N -d aln.sort.gam.idx aln.sort.gam
vg find -d aln.sort.gam.idx -o 24 | vg view -a - | wc -l
vg find -d aln.sort.gam.idx -o 23 | vg view -a - | wc -l
vg find -x x.xg -n 24 -c 1 >m.vg
vg view -dA  <(vg find -d aln.sort.gam.idx -A m.vg) <(vg find -x x.xg -G
<(vg find -d aln.sort.gam.idx -A m.vg))
```



vg explode (break graphs apart)

```
vg mod -pl 16 -e 3 x.vg | vg explode - parts
parts/component0.vg     x
parts/component1.vg     x
parts/component2.vg     x
parts/component3.vg     x
parts/component4.vg     x
parts/component5.vg
parts/component6.vg     x
parts/component7.vg     x
parts/component8.vg     x
parts/component9.vg     x
```

vg chunk (break graphs into pieces)

```
vg chunk -x x.xg -n 10
ls chunk*
```

```
chunk_0_ids_1_23.vg
chunk_1_ids_21_46.vg
chunk_3_ids_66_90.vg
chunk_5_ids_109_133.vg
chunk_7_ids_153_177.vg
chunk_9_ids_197_210.vg
chunk_0_ids_1_5_trace_annotate.txt
chunk_2_ids_43_68.vg
chunk_4_ids_88_112.vg
chunk_6_ids_131_155.vg
chunk_8_ids_175_200.vg
```

vg pack (graph coverage vectors)

vg pack -x x.xg -g aln.gam -d

seq.pos	node.id	node.offset	coverage
0	1	0	0
1	1	1	0
2	1	2	1
3	1	3	0
4	1	4	1
5	1	5	1
6	1	6	2
7	1	7	2
8	2	0	0
9	3	0	2
10	4	0	2
11	5	0	0
12	6	0	2
…			

Questions

How confident are you at building your own workflows within vg framework?

How confident are you creating graphs using vg msga?

How confident are you at modifying graphs using vg mod?

How confident are you working with the JSON output of vg?

How confident are you to use vg on assembly graphs?

vg results

New Results

Sequence variation aware genome references and read mapping with the variation graph toolkit

Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Michael F Lin, Benedict Paten, Richard Durbin

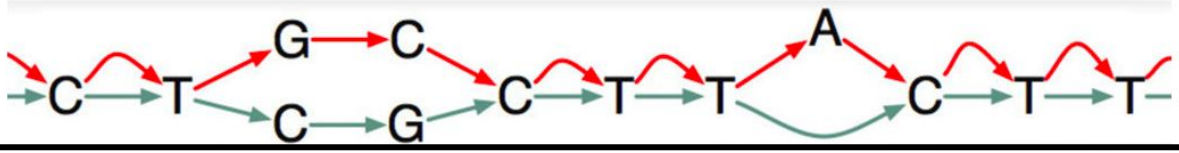
doi: https://doi.org/10.1101/234856

STAT Sections Topics Multimedia Popular STAT Plus Job Board About Newsletters Log In / Subscribe

IN THE LAB

As DNA reveals its secrets, scientists are assembling a new picture of humanity

CARL ZIMMER @carlzimmer / OCTOBER 7, 2016



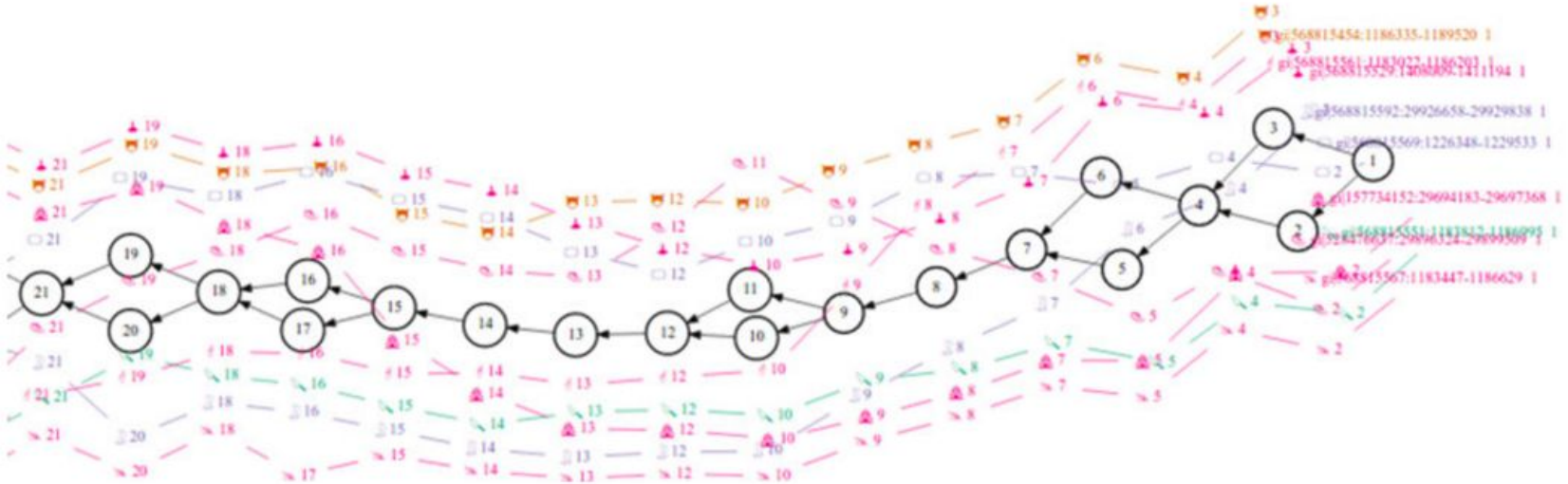
STAT Sections Topics Multimedia Popular STAT Plus Job Board About Newsletters Log In / Subscribe

IN THE LAB

As DN… assem…

CARL ZIMMER

Paten, a computational biologist at the University of California, Santa Cruz, belongs to a cadre of scientists who are building the tools to look at genomes in a new way: as a single network of DNA sequences, known as a genome graph.



ERIK GARRISON USING VG SOFTWARE (WELLCOME TRUST SANGER INSTITUTE)

STAT Sections Topics Multimedia Popular STAT Plus Job Board About Newsletters Log In / Subscribe

IN THE LAB

As DN… assem…

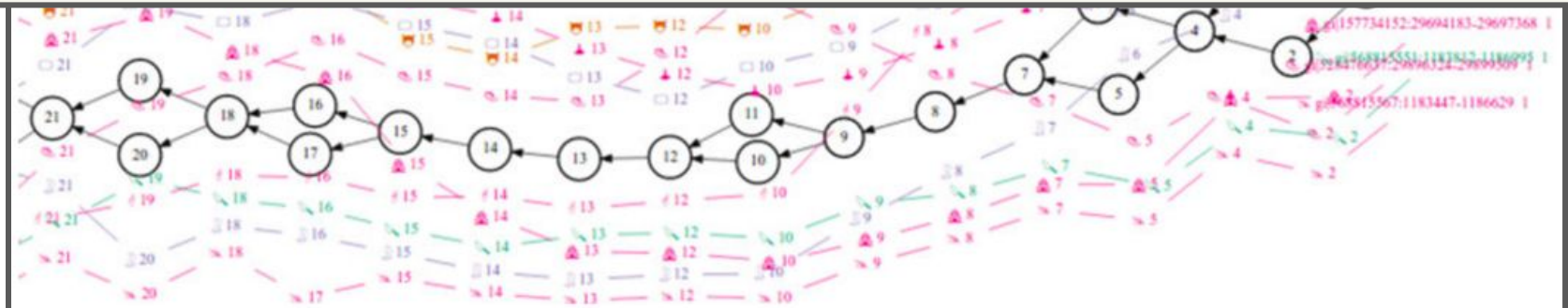
CARL ZIMMER

Paten, a computational biologist at the University of California, Santa Cruz, belongs to a cadre of scientists who are building the tools to look at genomes in a new way: as a single network of DNA sequences, known as a genome graph.

Hacker News new | comments | show | ask | jobs | submit

▲ As DNA reveals its secrets, scientists are assembling a new picture of humanity (statnews.com)

109 points by yawz 167 days ago | hide | past | web | 30 comments | favorite



ERIK GARRISON USING VG SOFTWARE (WELLCOME TRUST SANGER INSTITUTE)

STAT Sections Topics Multimedia Popular STAT Plus Job Board About Newsletters Log In / Subscribe

IN THE LAB

WhitneyLand 167 days ago [-]

There's got to be a lot more to the story that I don't understand.

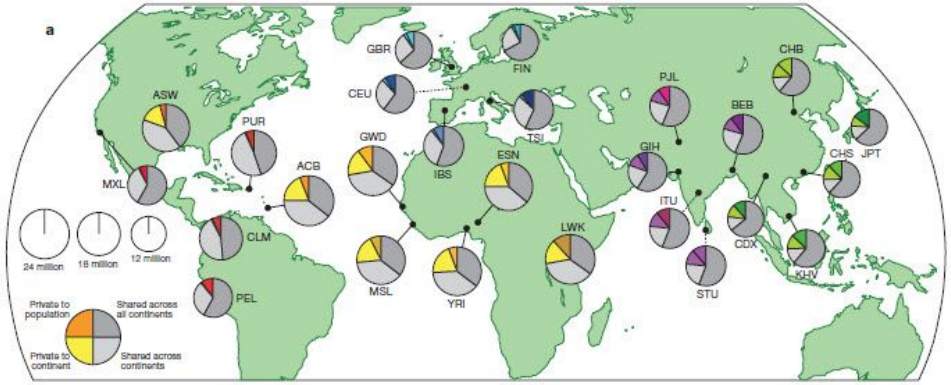
Why wouldn't it have been obvious 16 years ago that a thoughtfully designed data model was necessary, possibly using graphs, to account for variability and other attributes of the genome?

Surely it was foreseeable that tooling would be crucial and that a solid software foundation would be invaluable to enable efficient and flexible processing for years to come?

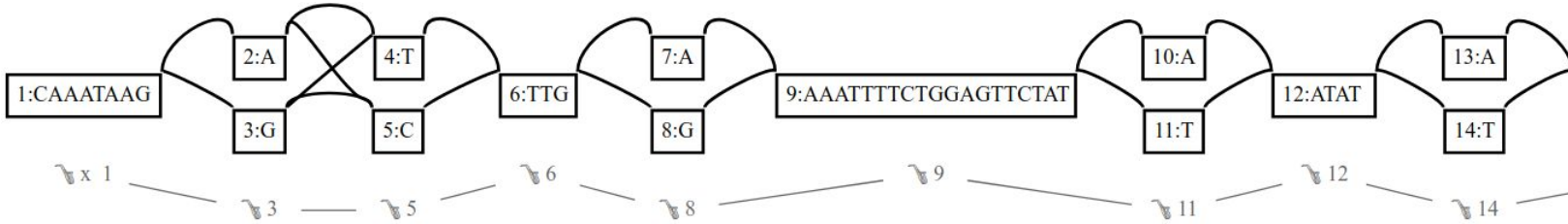
Who were the lead developers supporting the original public genome project and what were they thinking?

ERIK GARRISON USING VG SOFTWARE (WELLCOME TRUST SANGER INSTITUTE)

A human pangenome from 5000 haplotypes

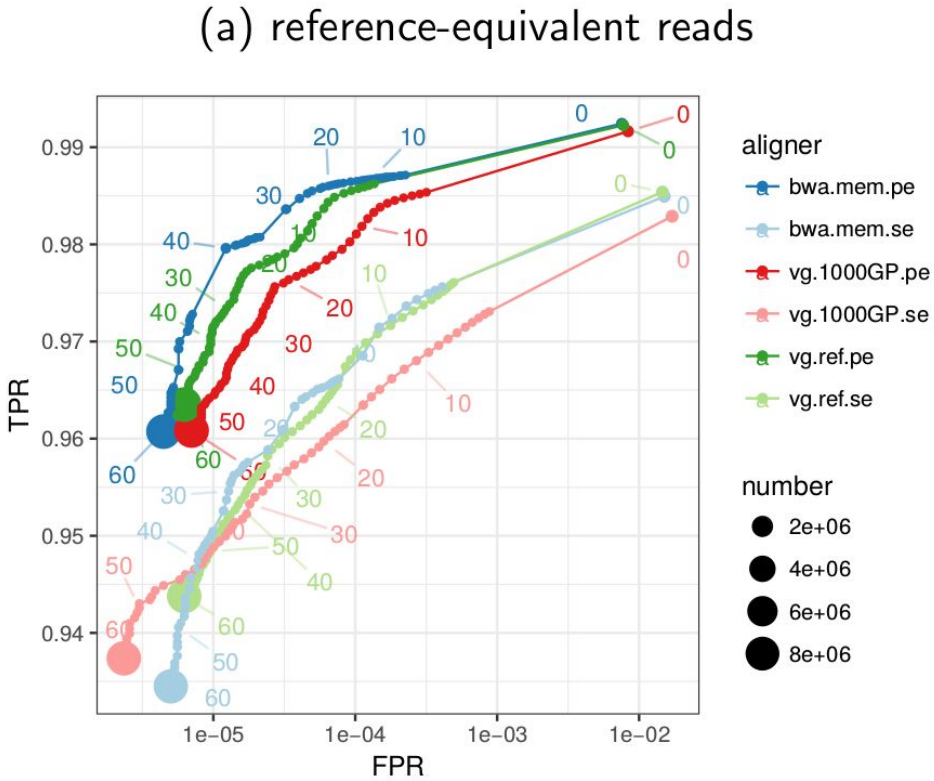


16	34903151	rs555344304	G	T
16	34903155	rs187546899	C	A
16	34903156	rs541113245	A	G
16	34903174	rs559070302	T	C
16	34903183	rs192120739	A	T
16	34903274	rs546024860	A	G
16	34903297	rs564406385	G	A
16	34903302	rs185743236	T	C
16	34903306	rs190571468	C	G
16	34903308	rs572200526	G	GCAAA
16	34903329	rs193033389	C	A
16	34903333	rs529317120	A	C
16	34903336	rs547889018	A	T



The 1000G phase 3 release encodes ~80M variants. We build a graph from it + the GRCh37 reference, and then index this for mapping (~70G total).

ROC of reads simulated from NA24385's haplotypes

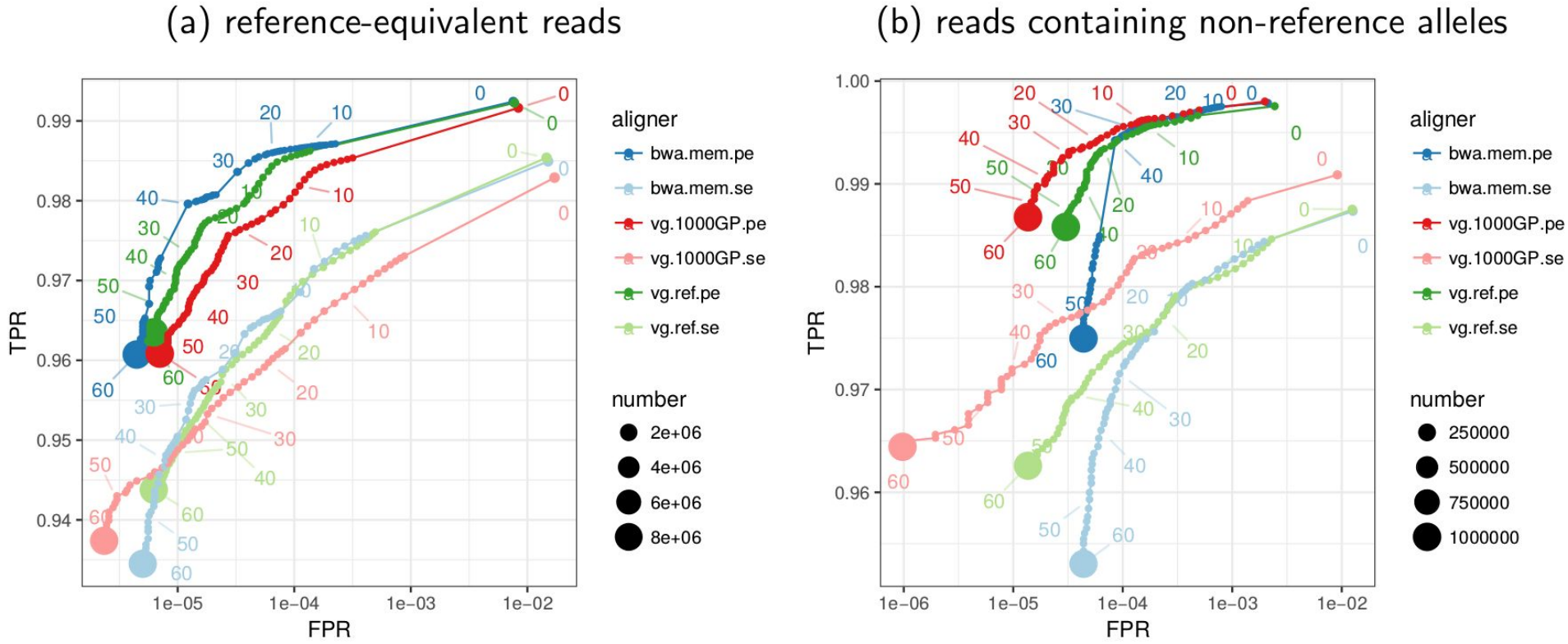


We simulate reads from the parentally-phased haplotypes of NA24385 (HG002), for which we have a "truth set" established by the NIST Genome in a Bottle project.

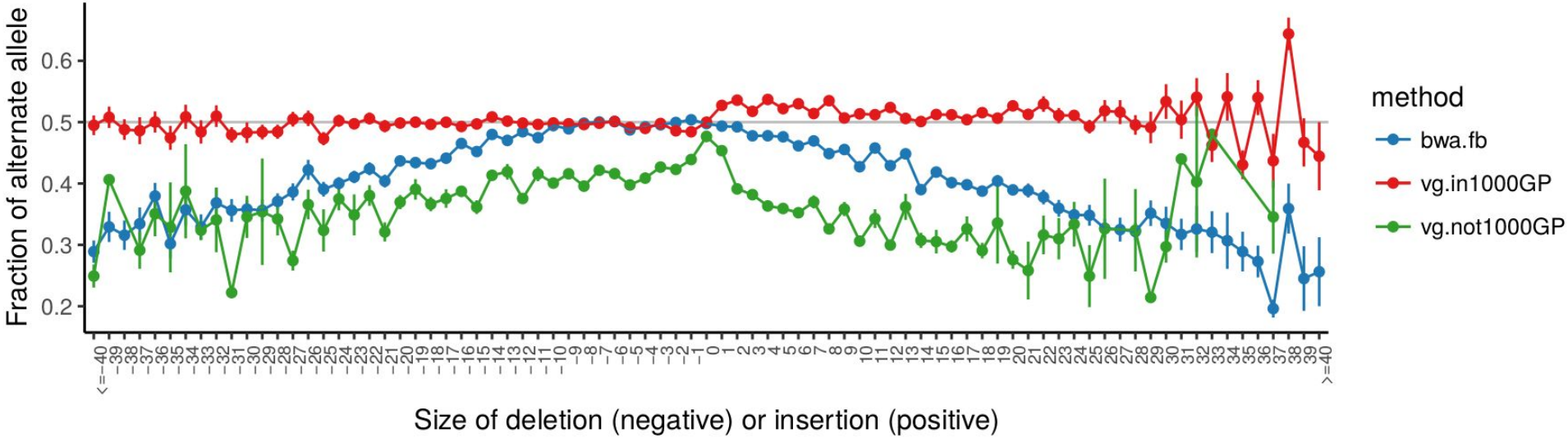
We then map the simulated reads using several different reference systems:

1) GRCh37 (using bwa mem)
2) GRCh37 (using vg)
3) 1000G pangenome (using vg)

ROC of reads simulated from NA24385's haplotypes

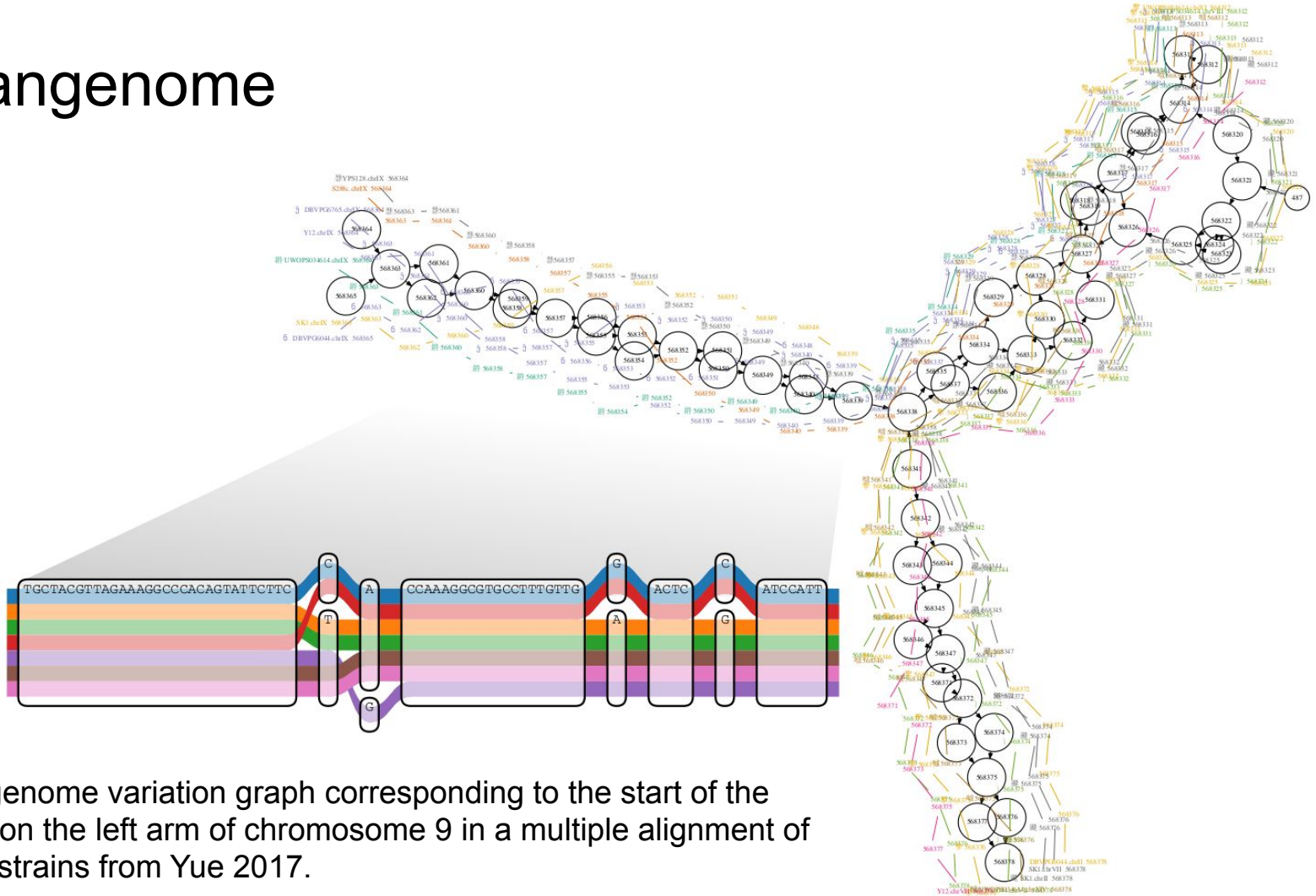


Allele observation bias in real NA24385 data



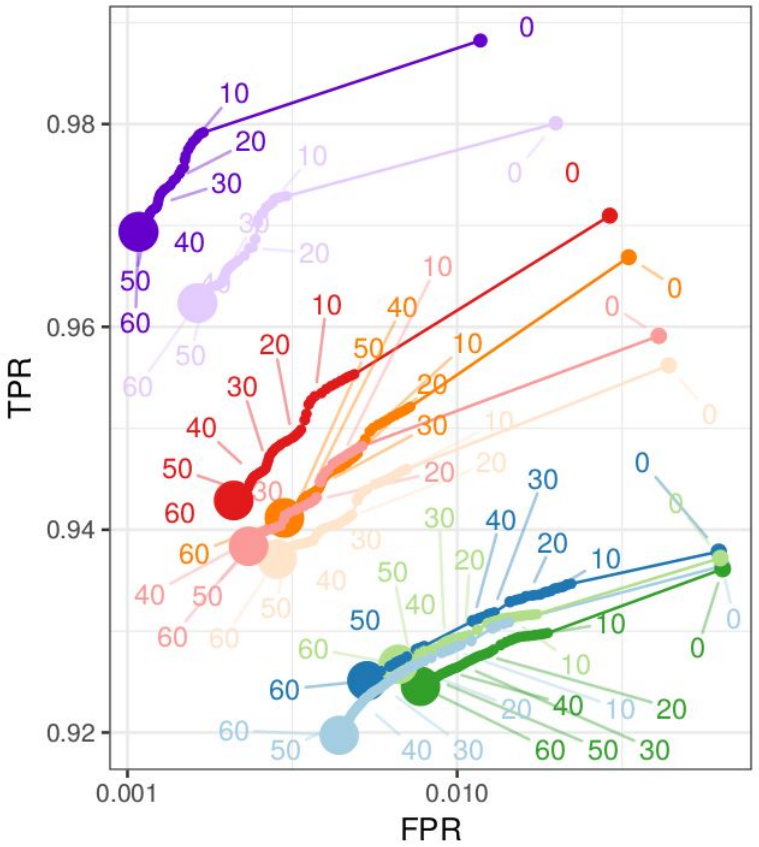
At “true” heterozygous variants in NA24385, we count how many reads map to the reference and to the alternate over various allele lengths. We see no bias when mapping to allels in the 1000G graph.

Yeast pangenome



A region of a yeast genome variation graph corresponding to the start of the subtelomeric region on the left arm of chromosome 9 in a multiple alignment of *de novo* assembled strains from Yue 2017.

Simulation from SK1 strain



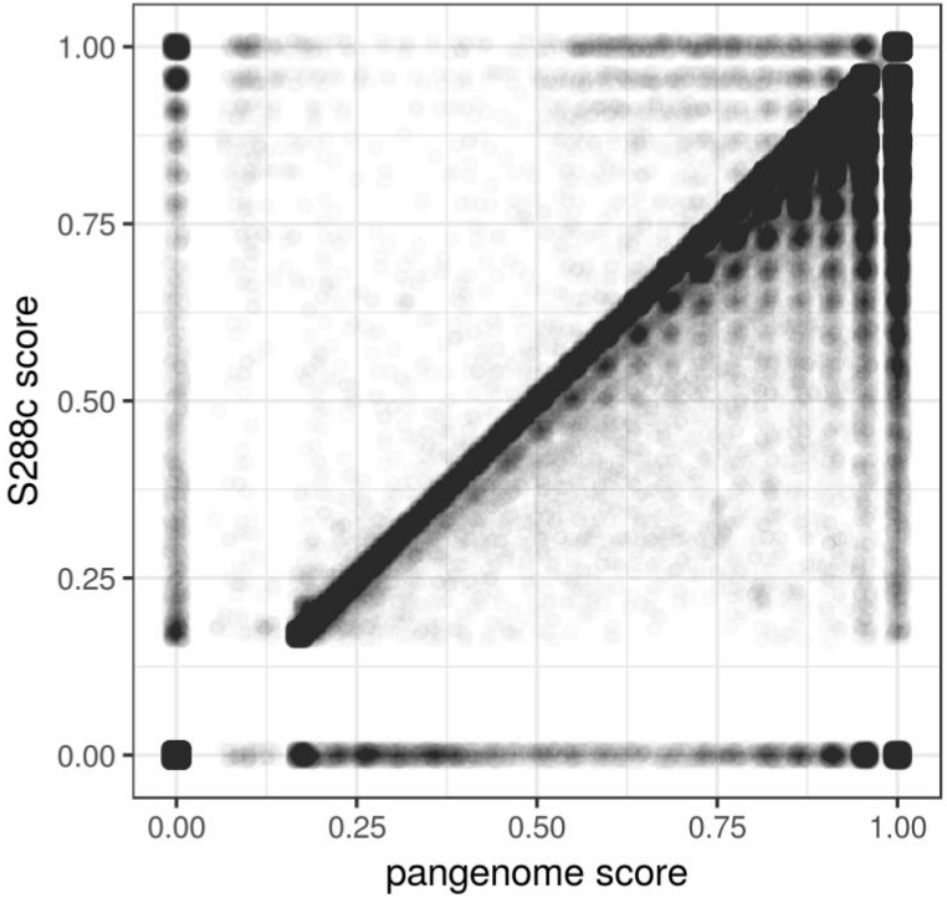
We simulate reads from SK1, and then map them back to different versions of the pangenome.

The pangenome outperforms the linear reference (S288c), but graph complexity limits our overall performance.

Mapping real data to the yeast pangenome

Here we map data from 12 Cerevisiae strains against our yeast pangenome and a linear reference for S288c.

74.6% of reads map with equal scores to both the pangenome and linear references, 24.9% map better to the pangenome, and only 0.5% map better to the linear reference.



vg as a metagenomics tool

We can avoid some common problems in metagenomics resequencing applications by using vg to align reads against an assembly graph.

The worst metagenome from the coolest place

Artic fresh water viromes (Svalbard viral metagenome)

https://www.ebi.ac.uk/ena/data/view/PRJEB5265 # study
https://www.ebi.ac.uk/ena/data/view/ERS396648 # specific data, from Svalbard
http://advances.sciencemag.org/content/1/5/e1400127 # paper

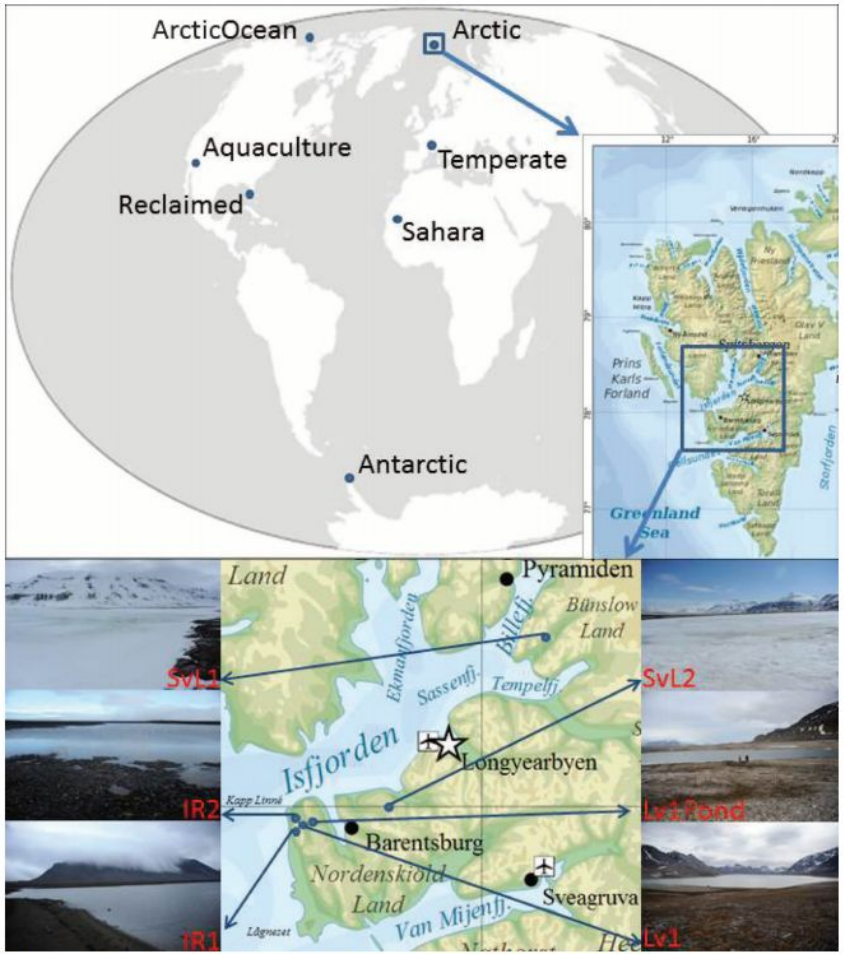


Fig. 1. Diagram depicting the global location of the freshwater environments studied. A detailed position of the Arctic lakes in Spitsbergen and photographs of the lakes at the time of sampling are shown. Coordinates (latitude/longitude) of the Arctic lakes: Lv1 (Lake Linnevatnet) (78°03.864′N; 13°46.308′E); Lv1Pond (Borgdammane) (78°04.254′N; 13°47.652′E); IR1 (Lake Tunsjøen) (78°03.375′N; 13°40.313′E); IR2 (78°02.935′N; 13°41973′E); SvL1 (Lake Nordammen) (78°38.279′N; 16°44.025′E); SvL2 (Lake Tenndammen) (78°06.118′N; 15°02.024′E). [Svalbard map was obtained from http://es.wikipedia.org/wiki/Svalbard#mediaviewer/Archivo:Topographic_map_of_Svalbard.svg and published under terms of the GNU Free Documentation License (http://commons.wikimedia.org/wiki/Commons:GNU_Free_Documentation_License_1.2).]

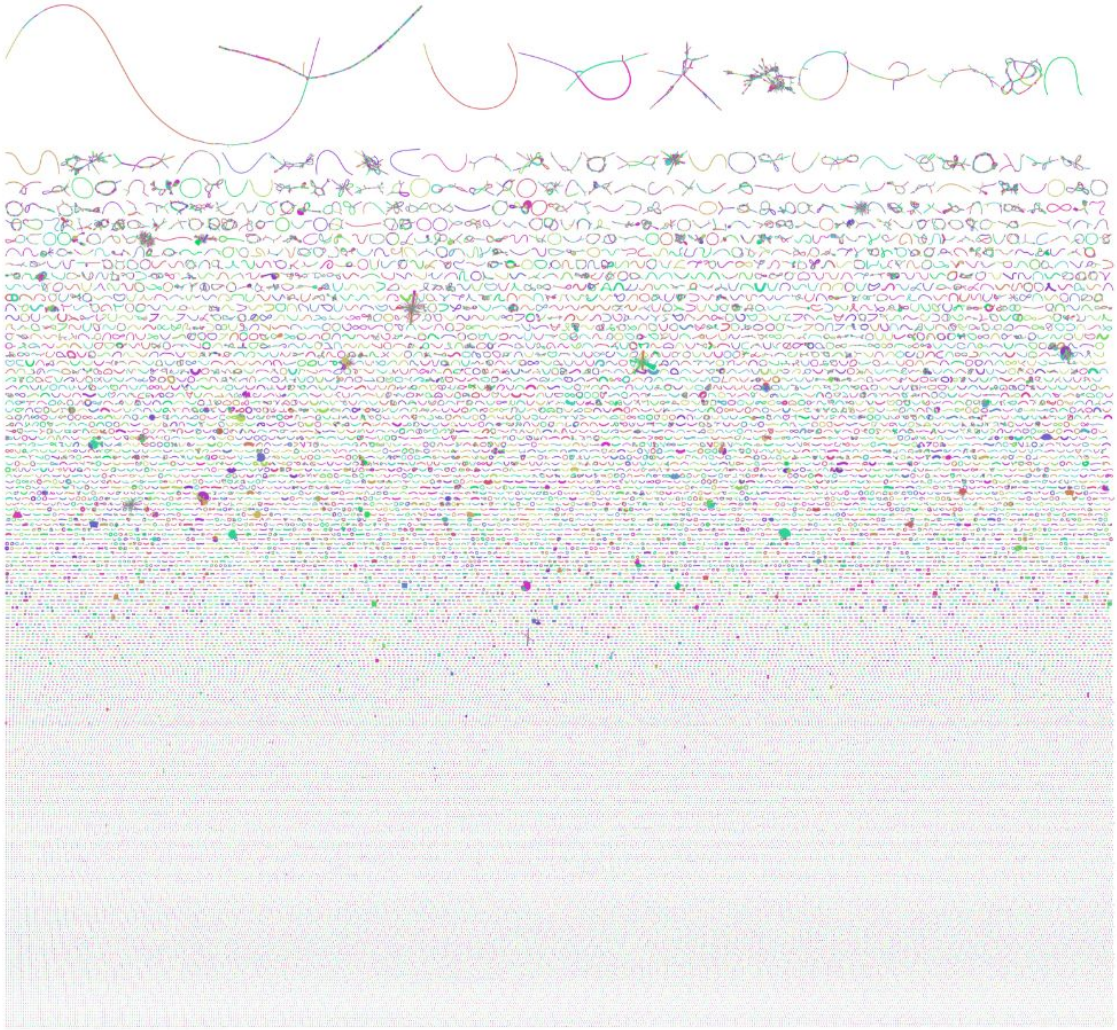
Pipeline

Reads -> minia3 -> contigs

The contig output from minia3 encodes a graph and may be converted to GFA using bcalm2 scripts!

Use bwa mem to align a subset of the reads to the contigs.

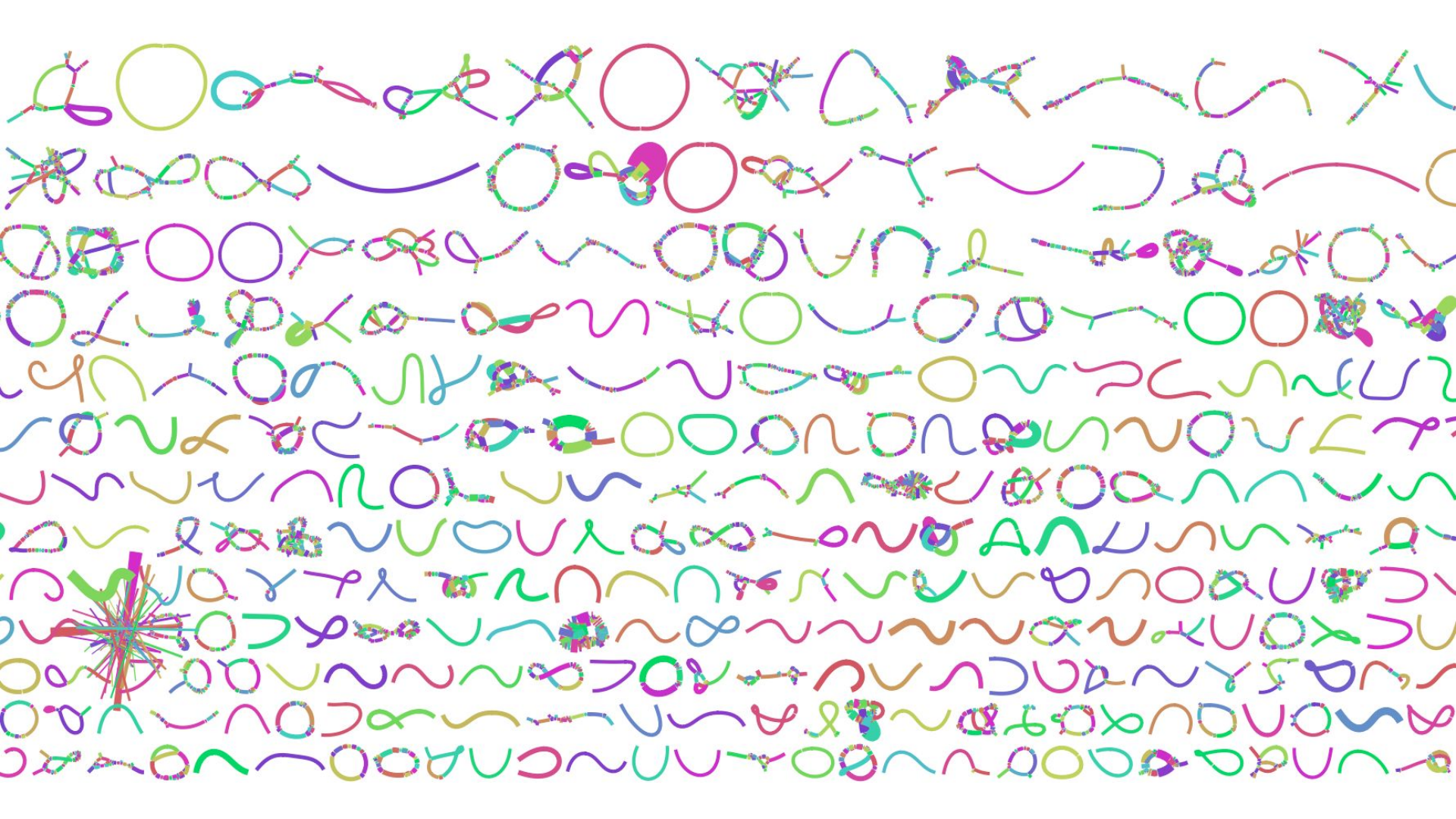
Use vg to align the same subset of the reads to the graph.

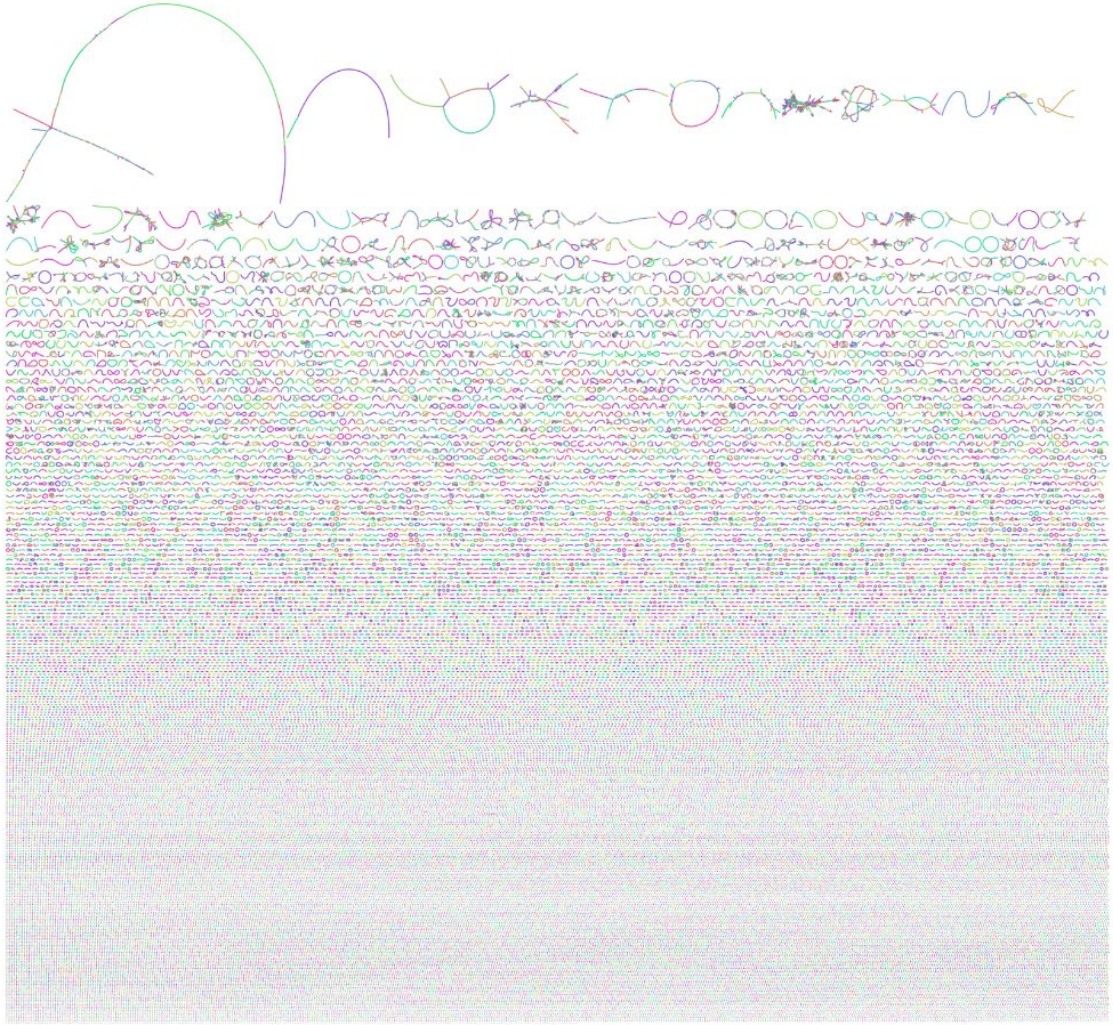


Raw assembly

There are very tangled portions of this graph that make alignment against it very hard.

As a compromise I pruned nodes with greater than in-degree 8 to make a smoother graph.



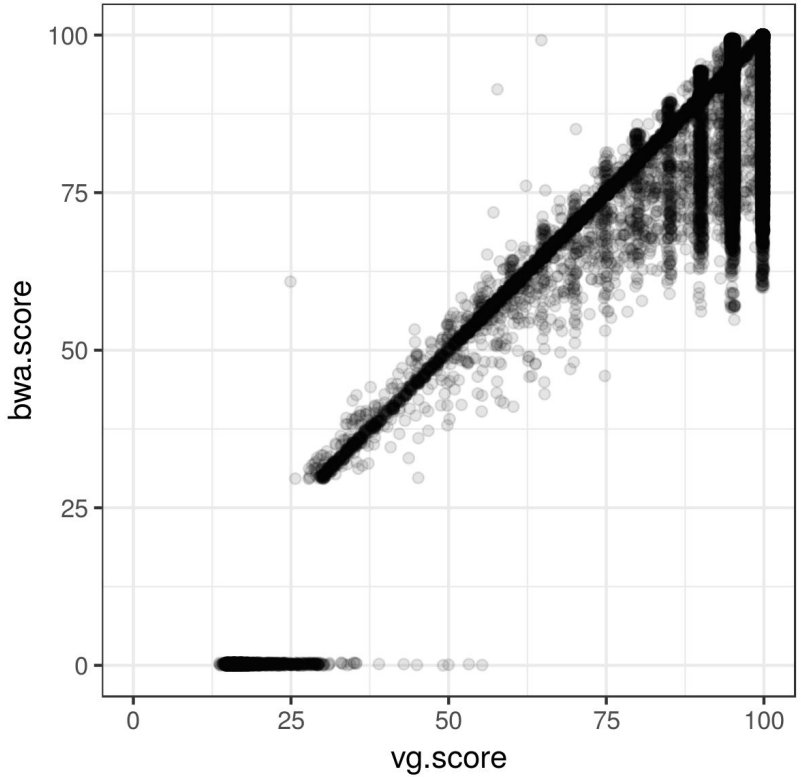


Smoothed graph

This is dramatically easier to index and align against.

Note greater number of short contigs (bottom portion).

Results: minia3 assembly k=51 abundance=3



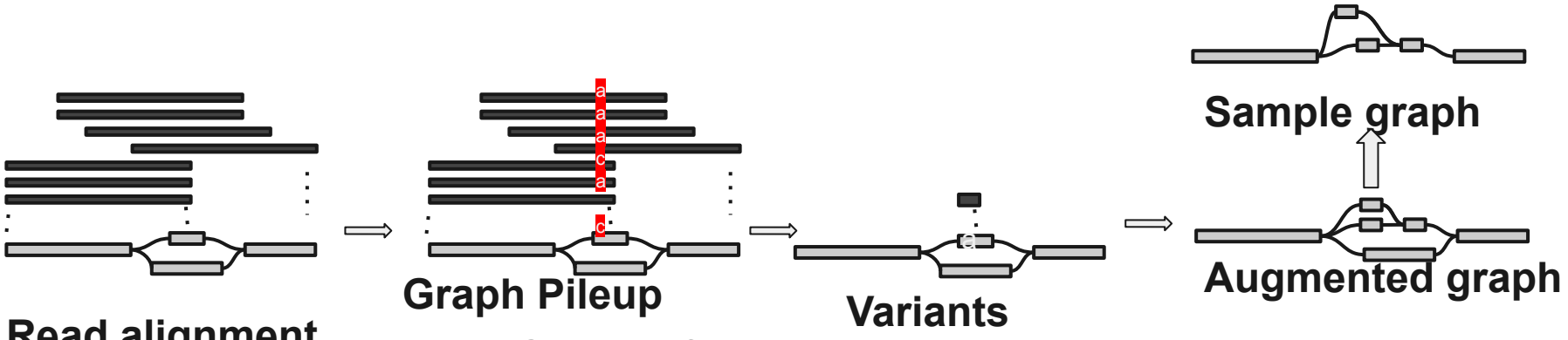
This on 100k of 200k reads which were held out of the assembly.

Mean score with vg:	94.83228
Mean score with bwa:	86.94674
Mapping rate vg:	0.96128
Mapping rate bwa:	0.96096

The mapping rate is similar, the difference is that vg can map almost all the reads matching the assembly graph full length. This is more extreme with shorter k in the assembly.

vg call

Variant calling on the graph



Read alignment

- GAM format
- from `vg map`

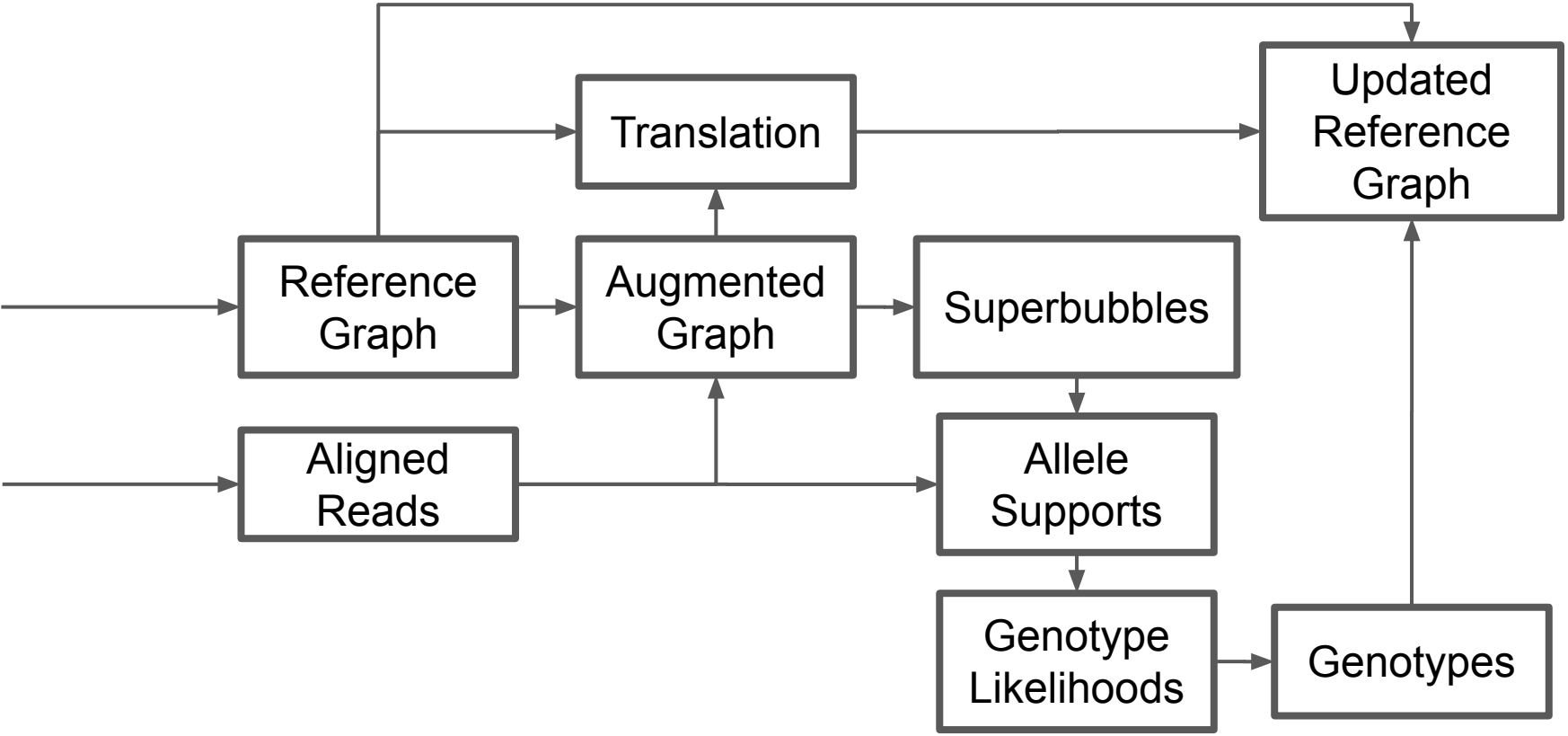
- VG pileup format
- from `vg pileup`

- VG format
- from `vg call`

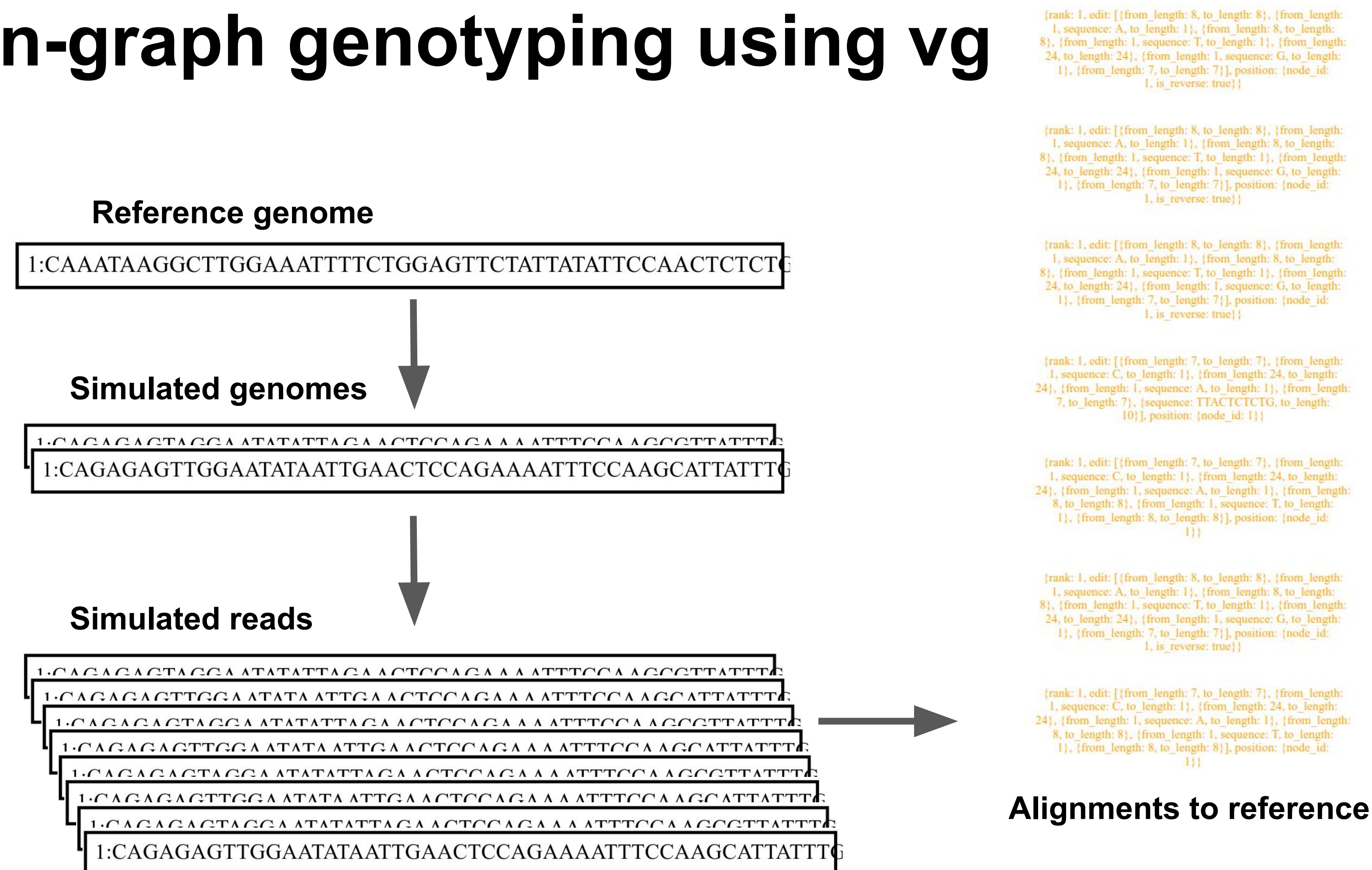
Glenn Hickey, Adam Novak, Benedict Paten, Mike Lin

Genotyper

In-graph genotyping using vg

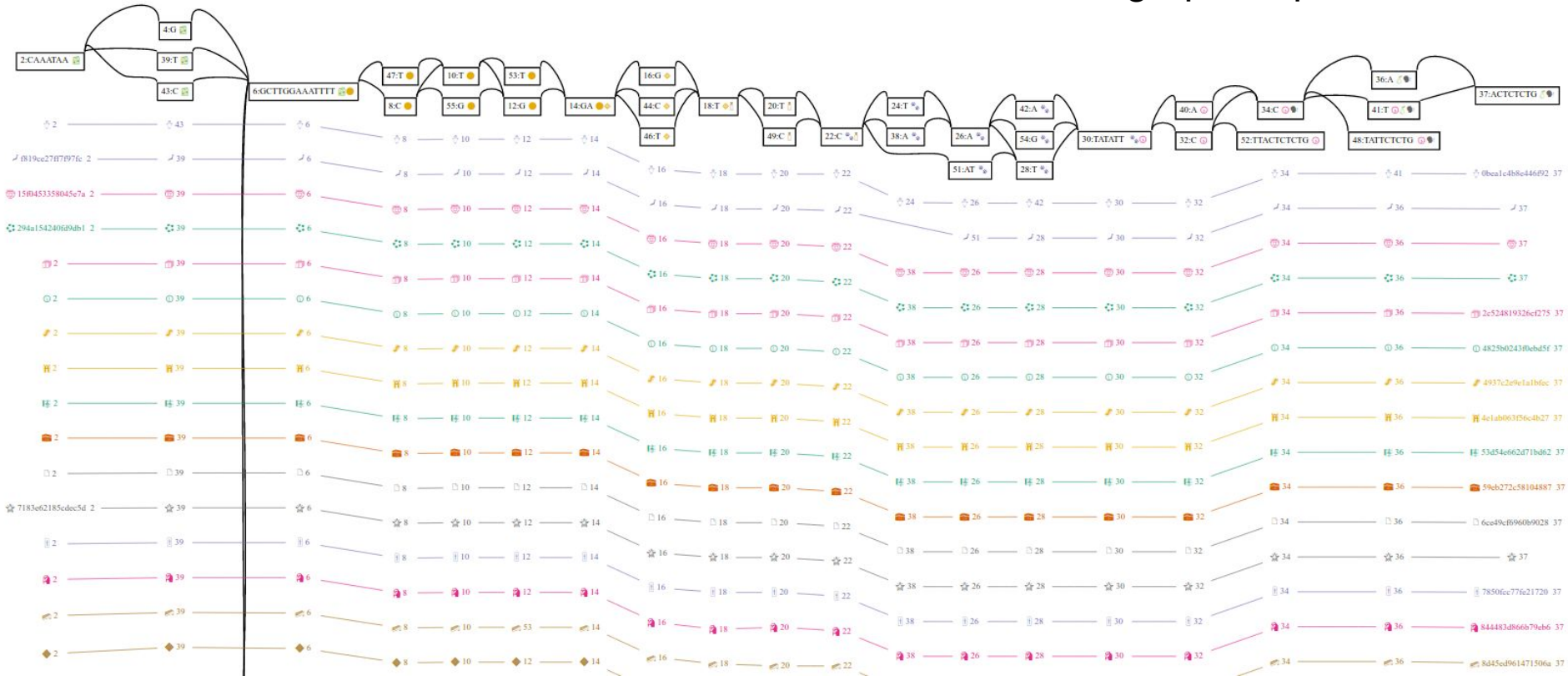


In-graph genotyping using vg

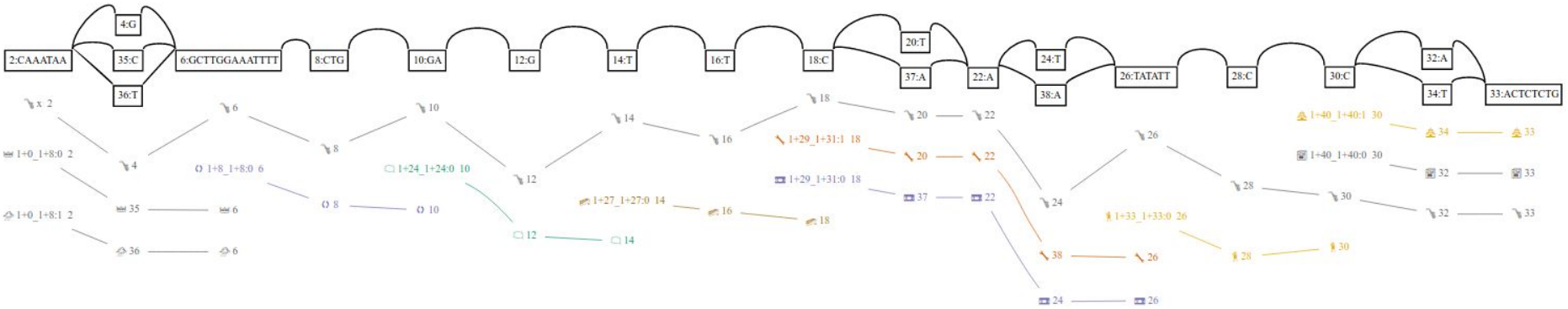


Augmented graph

The alignments have been fully embedded in the graph as paths.



Genotyper output ~ graph gVCF



The genotyper considers support for every bubble based on embedded paths and emits genotypes as “Locus” records that are each a set of alleles represented as paths relative to the base graph.

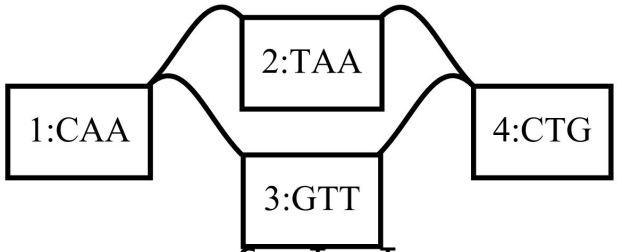
Implementation details:
Aligning against generic sequence graphs

No cycles allowed in string to DAG alignment

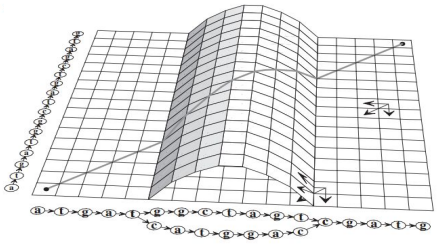
query:
CAAATTCT

	C	A	A
C	2	0	0
A	0	4	2
A	0	2	6
A	0	2	4
T	0	0	2
T	0	0	1
C	2	0	0
T	0	0	0

	T	A	A
C	0	0	0
A	0	2	2
A	3	2	4
A	4	5	4
T	6	3	3
T	4	4	1
C	2	2	2
T	2	0	0



	G	T	T
C	0	0	0
A	0	0	0
A	3	2	1
A	4	1	0
T	2	6	3
T	0	4	8
C	0	2	5
T	0	2	4



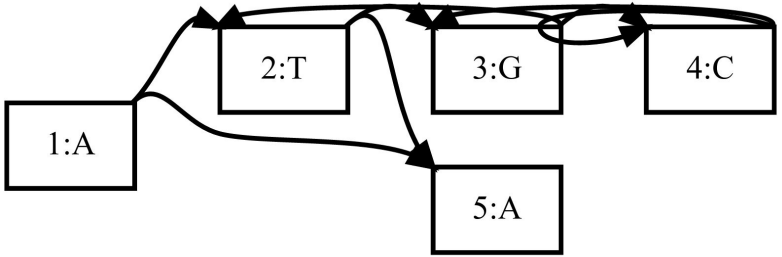
	C	T	G
C	2	0	0
A	0	0	0
A	1	0	0
A	2	0	0
T	2	4	1
T	5	4	3
C	10	7	6
T	7	12	9

1. fill the score matrixes
2. find the maximum score
3. trace back for alignment

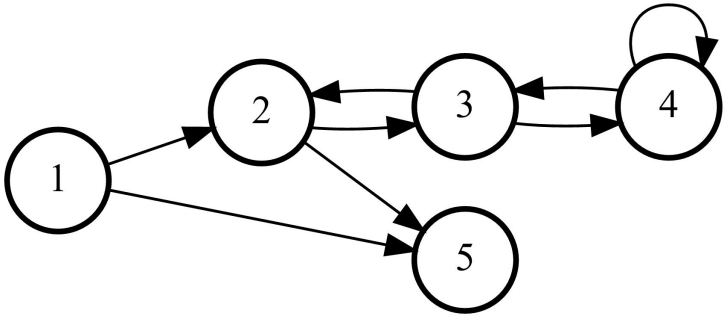
scores:
match = 2
mismatch = 2
gap_open = 3
gap_extension = 1

k-DAGification

This graph has multiple nested loops.



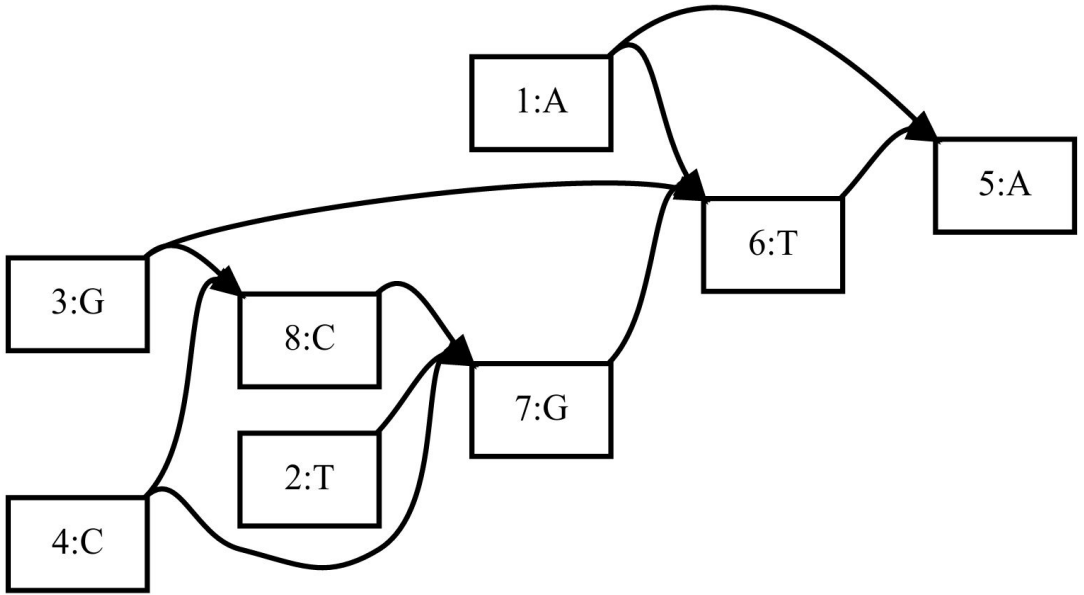
We will unroll this graph by copying the strongly connected component until any sequence of up to length k can be found in the DAG.



*Same graph visualized with and without sequences.

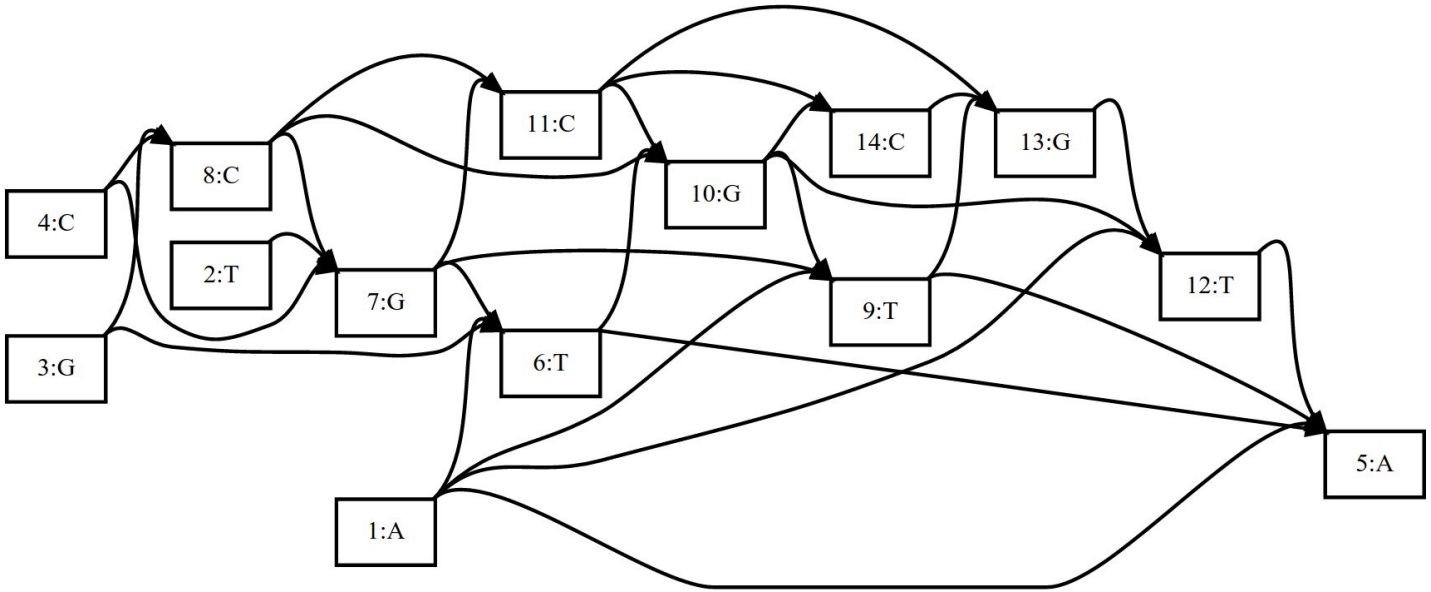
k-DAGification example

k=1 / k=2



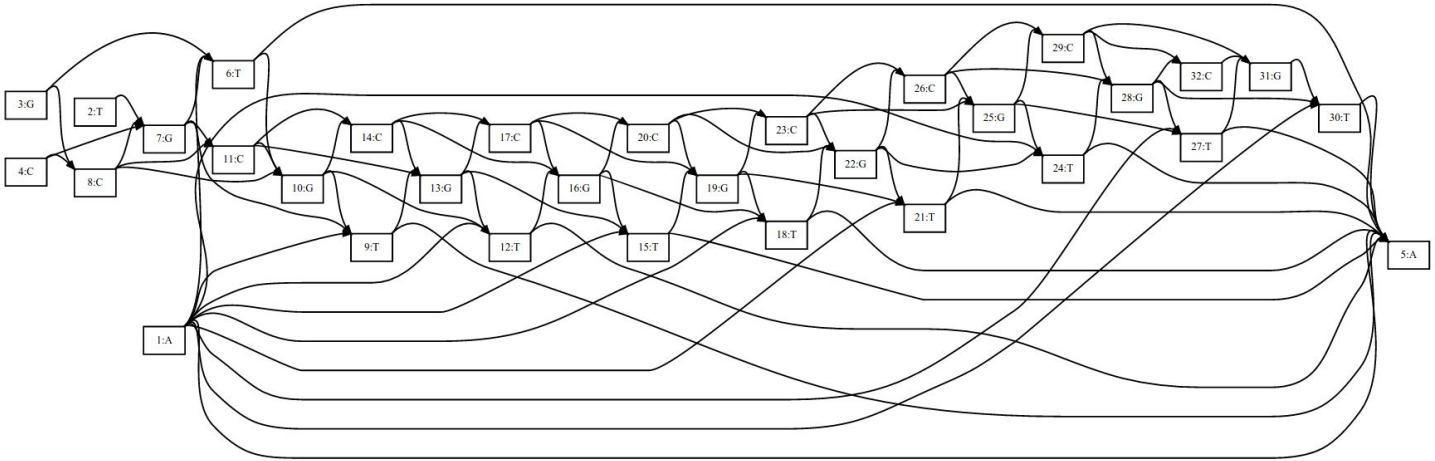
k-DAGification example

k=4



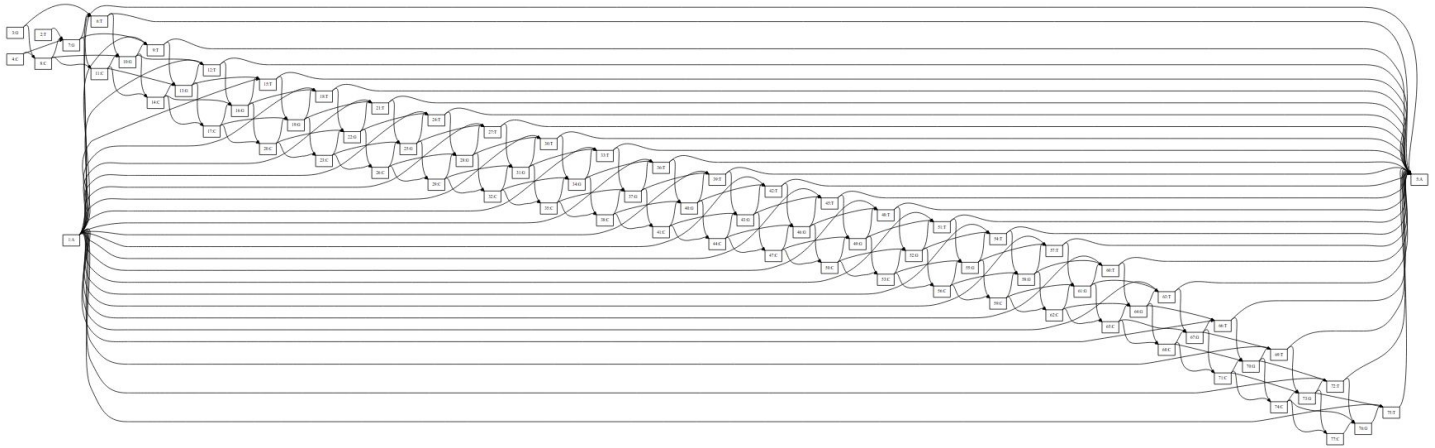
k-DAGification example

k=10



k-DAGification example

k=25

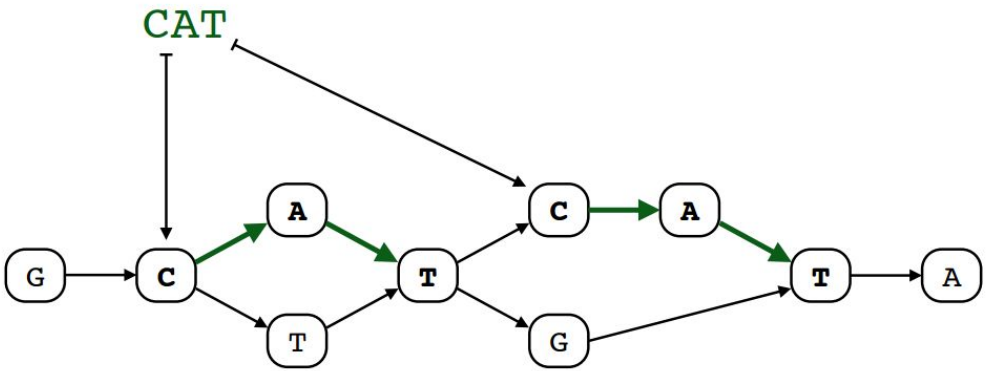


Implementation details: GCSA2

Indexing path sequences: GCSA2

Given a graph where paths are labeled by strings, a path index is a text index for the strings.

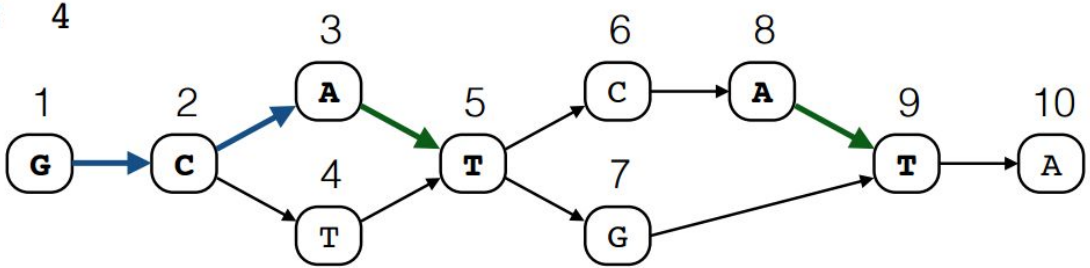
A path query finds the (start nodes of) the paths labeled by a kmer.



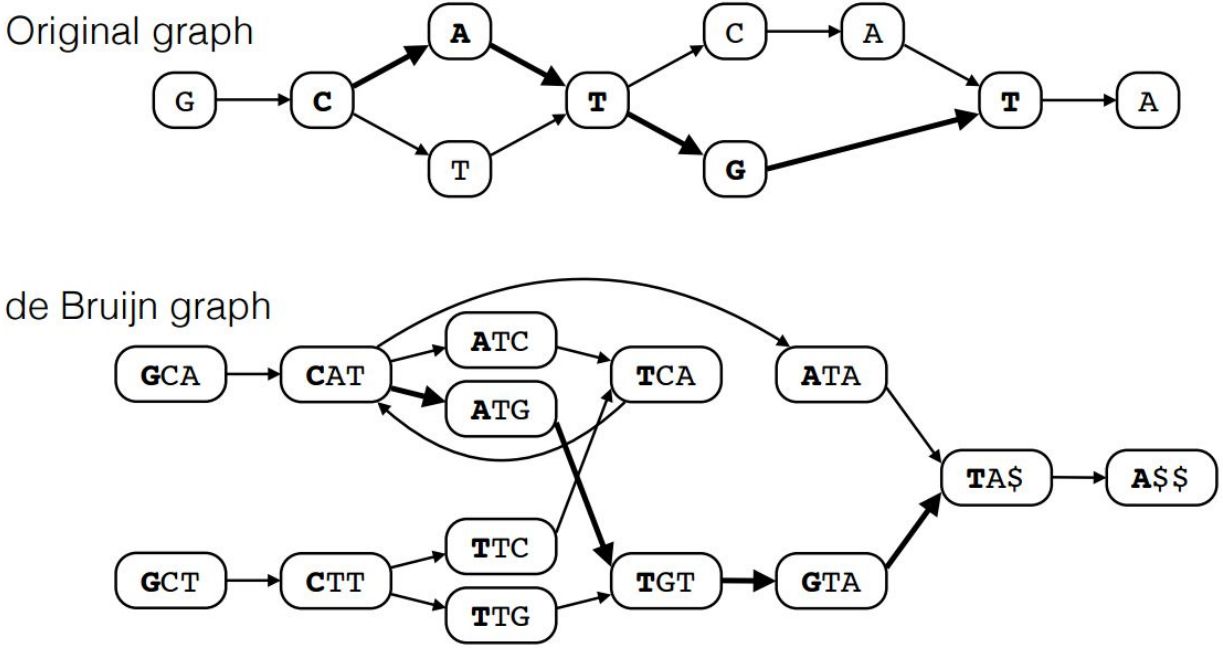
Jouni Sirén http://jltsiren.kapsi.fi/files/sendai2016.pdf

```
A$$   10
ATA   8
ATC   3
ATG   3
CAT   2, 6
CTT   2
GCA   1
GCT   1
GTA   7
TA$   9
TCA   5
TGT   5
TTC   4
TTG   4
```

- A kmer index based on a hash table supports queries of length k efficiently.
- If we sort the kmers, we can use them as a suffix array-like index for shorter queries.
- The kmer index can also simulate a de Bruijn graph.

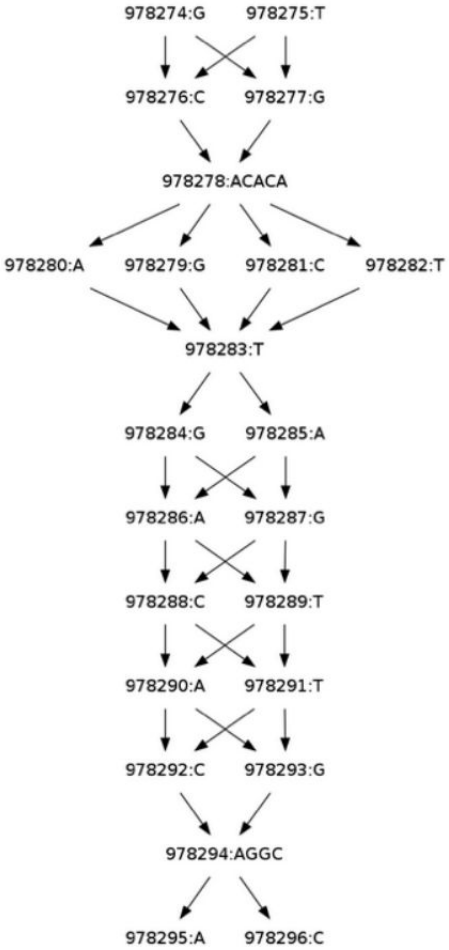


Jouni Sirén http://jltsiren.kapsi.fi/files/sendai2016.pdf



We can search for longer patterns by representing the kmer index as a de Bruijn graph.

Jouni Sirén http://jltsiren.kapsi.fi/files/sendai2016.pdf

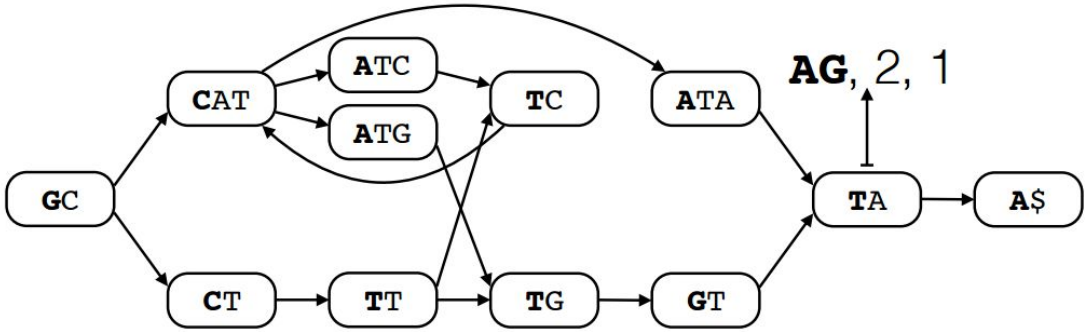


Some parts of the original graph may have **too many paths** through them. Those parts must be **pruned** before indexing.

The de Bruijn graph can also be pruned by **merging** the nodes with a **common prefix** of the label, if:

1. the shorter label **uniquely defines** the start node in the original graph; or
2. the start nodes **cannot be distinguished** by length-k extensions of the label.

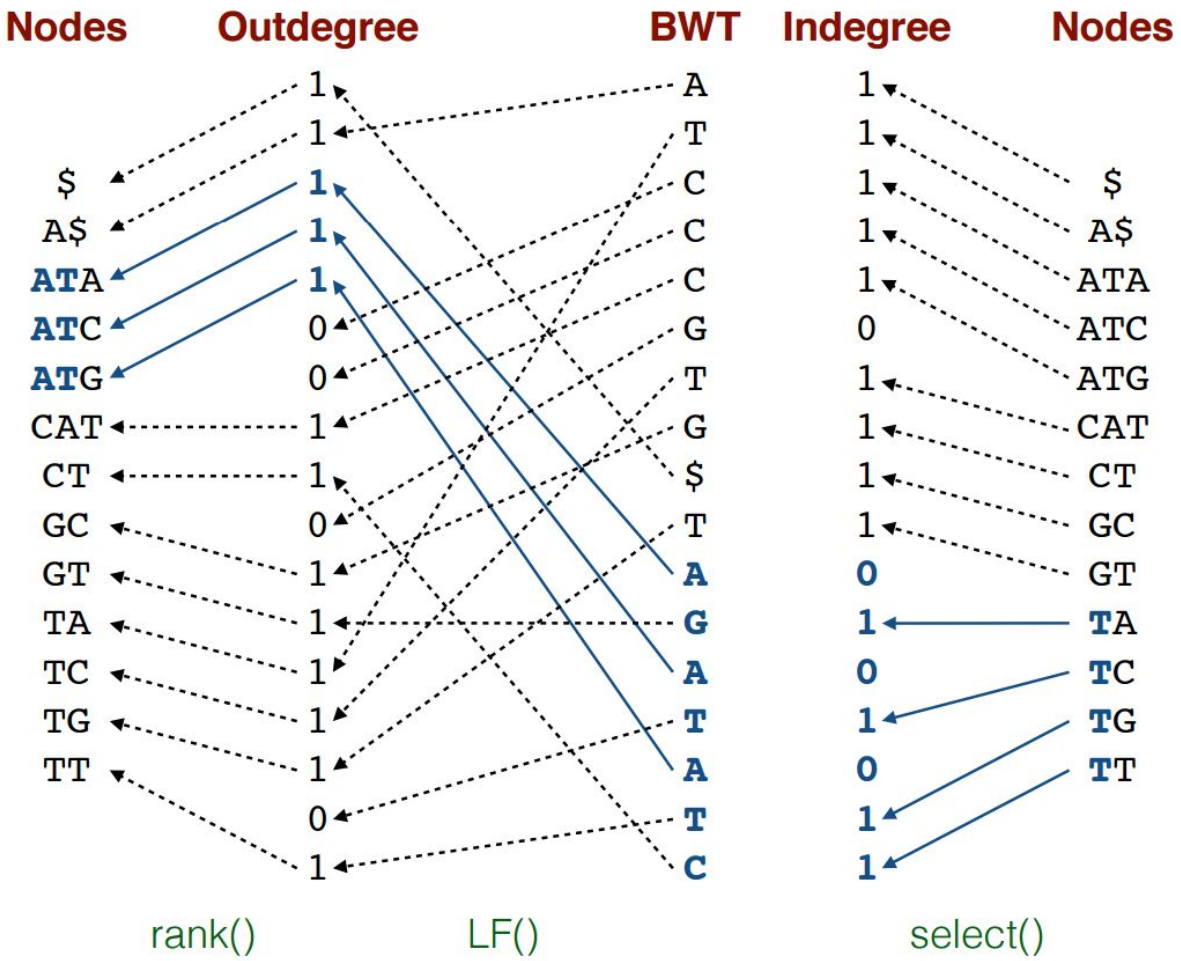
Jouni Sirén http://jltsiren.kapsi.fi/files/sendai2016.pdf



We store predecessor labels, indegree, and outdegree for each node. For the nodes at the beginning of unary paths, we also store pointers to the original graph. Edges can be determined if the nodes are stored in sorted order.

The encoding is similar to the Burrows-Wheeler transform and the FM-index. Typical space usage is 1–2 bytes/node.

Jouni Sirén http://jltsiren.kapsi.fi/files/sendai2016.pdf



Jouni Sirén http://jltsiren.kapsi.fi/files/sendai2016.pdf

Path length	16→32	16→64	16→128
Nodes:			
de Bruijn graph	6.23G	16.9G	118G
Pruned	4.39G	5.27G	5.76G
Index size:			
Full index	9.99 GB	9.22 GB	9.23 GB
Without pointers	4.10 GB	4.84 GB	5.27 GB
Construction:			
Time	7.20 h	11.4 h	15.5 h
Memory	43.8 GB	43.8 GB	43.8 GB
Disk	401 GB	424 GB	489 GB
I/O:			
Read	1.43 TB	2.11 TB	2.89 TB
Write	1.05 TB	1.71 TB	2.47 TB

1000GP human variation, vg mod -p -l 16 -e 4 | vg mod -S -l 100
32 cores, 256 GB memory, distributed Lustre file system

Jouni Sirén http://jltsiren.kapsi.fi/files/sendai2016.pdf

MEMs in vg

By including an LCP array, we can find SMEMs in a linear scan through the sequence in the same way as we do with linear references.

```
vg find -M ACCGTTAGAGTCAG -g h.gcsa
[["ACC",["1:-32"]],["CCGTTAG",["1:5"]],["GTTAGAGT",["1:19"]],["TAGAGTCAG",["1:40"]]]
```

ACGTGCCGTTAGCCAGTGGGTTAGAGTATCGATACAACTATAGAGTCAGAGCA

ACCGTTAGAGTCAG
ACC
CCGTTAG
GTTAGAGT
TAGAGTCAG

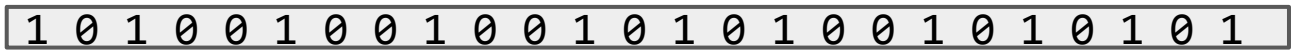
Sketch: backward search until we no longer match. Then execute *parent()* on the suffix tree node corresponding last non-empty BWT range to get a new range. Then continue backward searching to find the next SMEM.

Implementation details: XG

Succinct variation graphs (xg)

A *succinct data structure*

- uses an amount of space close to the minimum information-theoretic lower bound,
- but does so while allowing efficient queries!



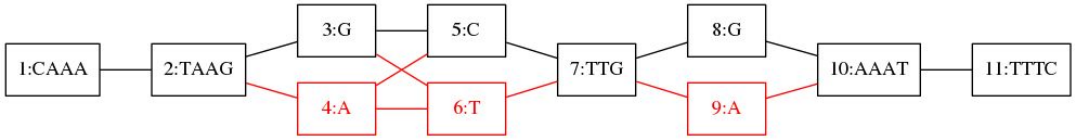
Core concept is the rank/select dictionary, a bit vector, e.g.

That (given q ∈ {0,1}) supports functions:

$$\mathbf{rank}_q(x) = |\{k \in [0 \ldots x] : B[k] = q\}|$$

$$\mathbf{select}_q(x) = \min\{k \in [0 \ldots n) : \mathbf{rank}_q(k) = x\}$$

xg: nodes and edges



nodes

label	CAAA TAAG G A C T TTG G A AAAT TTTC
nodes	1000 1000 1 1 1 1 100 1 1 1000 1000
id	1 2 3 4 5 6 7 8 9 10 11

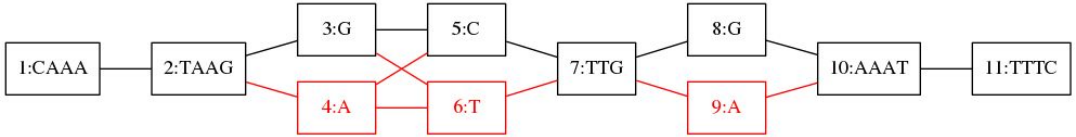
Graph storage is straightforward, and complicated only be the fact that nodes have variable length.

edges

The from and to integer vectors store [id][edges from or to id...][id+1][edges]

from	1 2 2 3 4 3 5 6 4 5 6 5 7 6 7 7 8 9 8 10 9 10 10 11 11
...	1 0 1 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1
to	1 2 1 3 2 4 2 5 3 4 6 3 4 7 5 6 8 7 9 7 10 8 9 11 10
...	1 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0

xg: positional paths



paths* (*for each path)

members `11110110000110011011001111`

ordered node ids `1 2 3 5 7 8 10 11`

node path offsets `0 4 8 9 10 13 14 18`

node starts `10001000111001100010000`

a collection of nodes and edges--- we can use this structure alone for annotating subgraphs.

In conjunction these structures allow navigating the graph using path-relative coordinates.

We can find the node at a particular path position by rank_1(i) on the node starts bit vector, and we can find the position of a node in a path using the ordered node ids, which is indexed using a wavelet tree.