

VG: Overview, Internals, and Short Read Mapping

Jouni Sirén
UCSC Genomics Institute

Slides available at: <https://jltsiren.kapsi.fi/files/dalhousie2021.pdf>

Reference genomes

A reference genome is a model of the data. It describes what we expect the genome to look like.

A good model:

1. is easy to use and understand; and
2. describes the data with sufficient accuracy and in sufficient detail.

Traditional reference sequences often fail the second point.



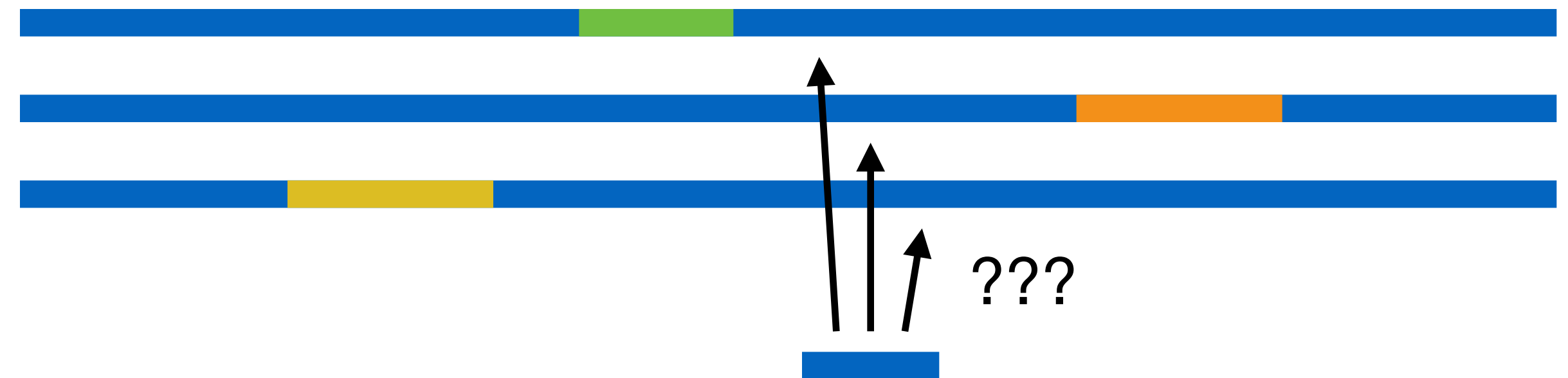
Multiple references

We could try to use a representative collection of haplotypes as the reference.

The haplotypes are highly similar, and the model fails to tell when the similar-looking positions are equivalent.

If a sequence aligns to multiple equivalent positions, the alignment is often useful.

If we find equally good alignments to non-equivalent positions, the mapping is not very informative.



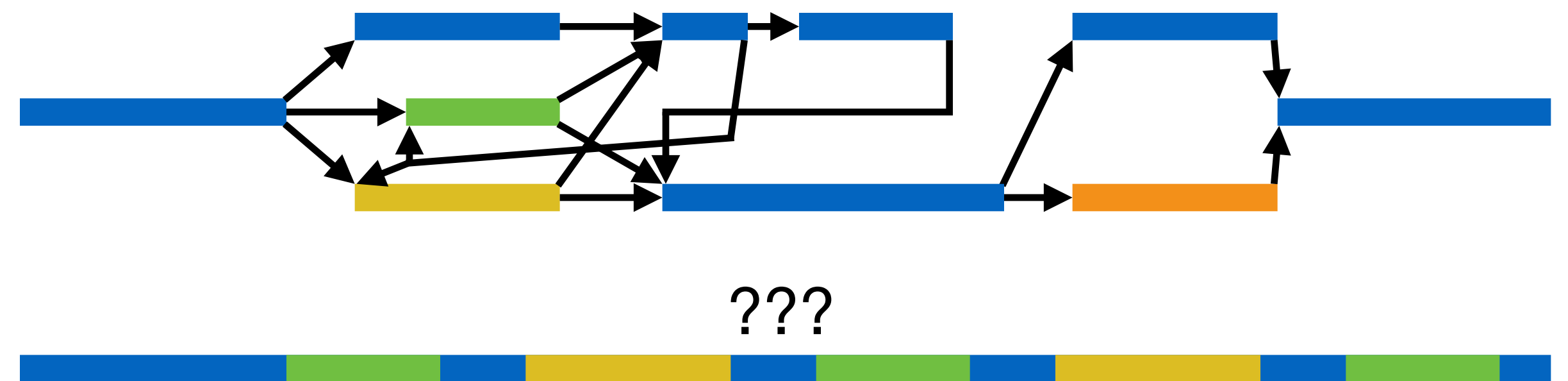
Genome graphs

If we align and collapse the equivalent positions, we get a graph representing the haplotypes.

We need cycles to represent certain kinds of structural variation accurately.

Cycles introduce paths that make no biological sense.

Complex regions in the graph may also be computationally expensive.

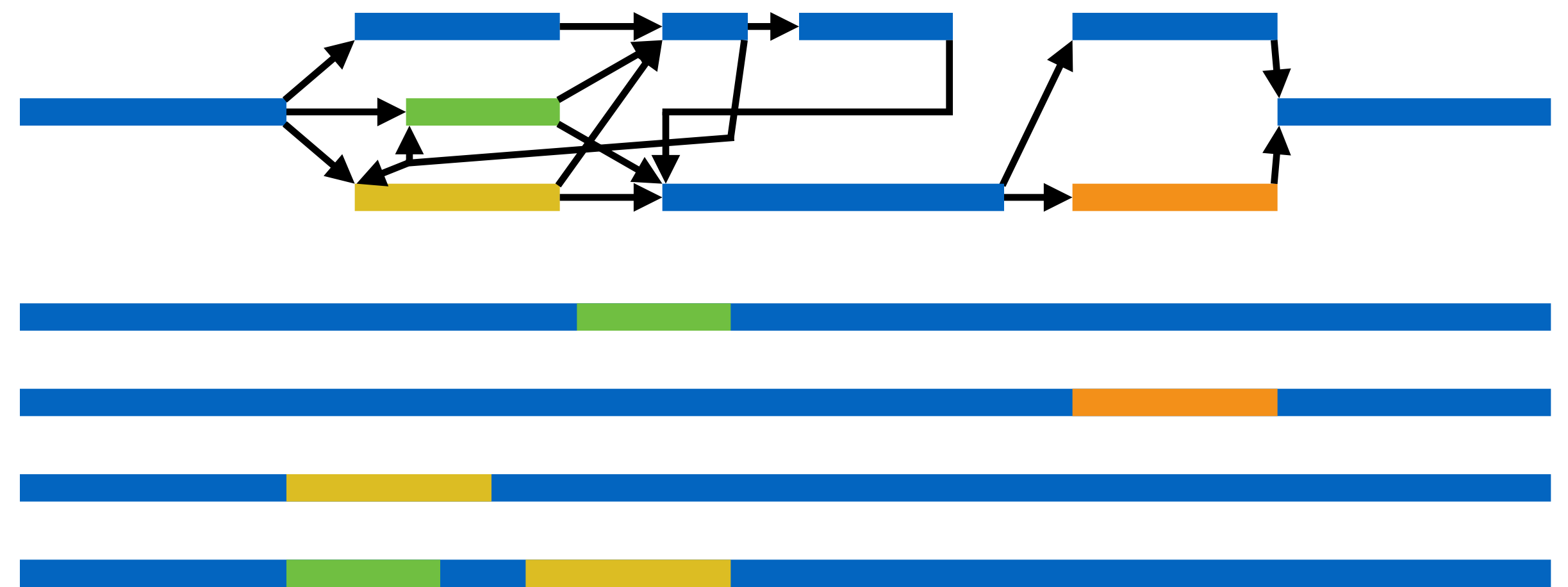


Graph and paths

If we embed the original haplotypes as paths in the graph, we get the best of both worlds.

The graph tells which positions in the haplotypes are equivalent, and the haplotypes tell what kind of paths we expect to see.

As the paths restrict the scope of the model, using it becomes computationally cheaper.



VG Toolkit

VG toolkit

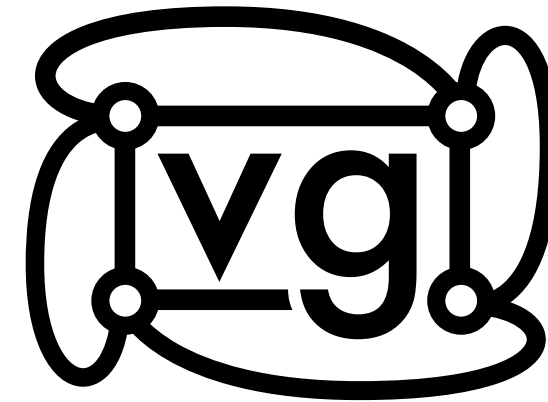
The VG toolkit is a large collection of algorithms for working with genome graphs.

It started as rapid prototyping with Protobuf.

There are now many file formats, interfaces, and libraries for wider interoperability.

We are removing unnecessary branches and relying more on stand-alone tools.

Some documentation can be found in the wiki: <https://github.com/vgteam/vg/wiki>.



<https://github.com/vgteam/vg>

Garrison et al.: **Variation graph toolkit improves read mapping by representing genetic variation in the reference.**

Nature Biotechnology, 2018.

[DOI: 10.1038/nbt.4227](https://doi.org/10.1038/nbt.4227)

Hickey et al.: **Genotyping structural variants in pangenome graphs using the vg toolkit.** Genome Biology, 2020.

[DOI: 10.1186/s13059-020-1941-7](https://doi.org/10.1186/s13059-020-1941-7)

Compiling VG

On Ubuntu

```
$ git clone --recursive https://github.com/vgteam/vg.git
$ cd vg
$ make get-deps # Requires sudo privileges
$ make -j 4 # Number of parallel jobs
```

On macOS with Homebrew

```
$ git clone --recursive https://github.com/vgteam/vg.git
$ cd vg
$ brew bundle
$ export PATH="/usr/local/opt/coreutils/libexec/gnubin:/usr/local/opt/bison/bin:/usr/local/bin:$PATH"
$ export LD_LIBRARY_PATH=/usr/local/lib/
$ export LIBRARY_PATH=/usr/local/lib/
$ export CFLAGS="-isystem /usr/local/include/"
$ make -j 4 # Number of parallel jobs
```

VG has many dependencies, which makes the compilation a fragile process, especially on macOS. A fresh Ubuntu virtual machine provides the best results.

VG releases

Releases Tags

Edit release Delete

Latest release



v1.31.0
08faee0

Verified

Compare ▾

vg 1.31.0 - Caffaraccia

jeizenga released this 24 days ago · 98 commits to master since this release

 Click here to DOWNLOAD 

Don't forget to mark the static binary executable:

```
chmod +x vg
```

Docker Image: `quay.io/vgteam/vg:v1.31.0`

There is a VG release with a precompiled Linux binary and a Docker image every six weeks. The examples in this talk will be based on **VG version 1.31.0**.

VG data model

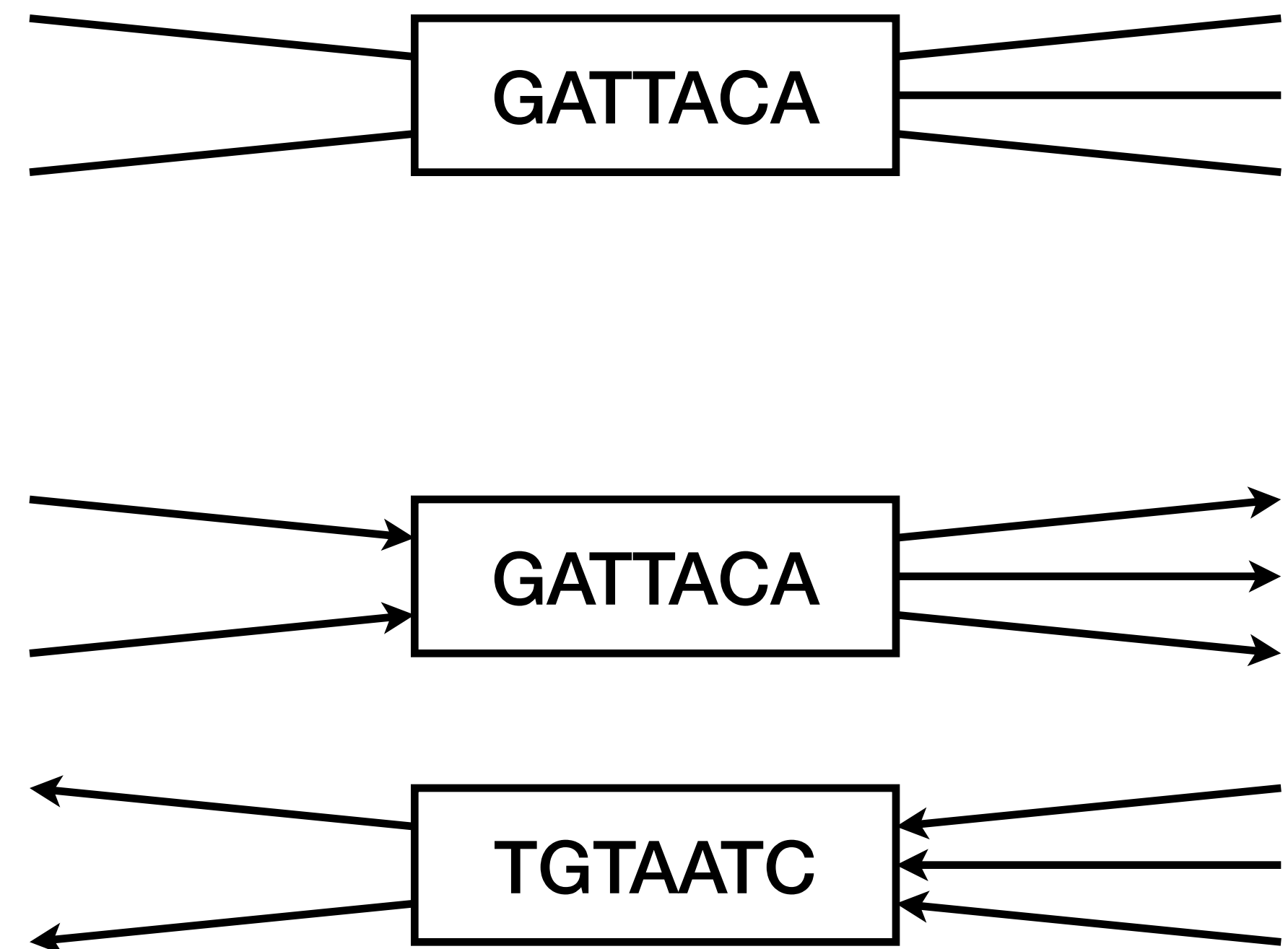
VG data model is based on **bidirected sequence graphs**.

Each node has undirected edges adjacent to its left and right sides.

Forward traversal enters from the left, reads the sequence, and exits from the right.

Reverse traversal enters from the right, reads the reverse complement, and exits from the left.

This can be simulated with a directed graph with separate nodes for each orientation.



Data model details

- Nodes should be short.
 - GCSA2 and the minimizer index do not work with nodes longer than 1024 bp.
 - Some operations create temporary copies of the sequence.
 - Visualization may display the sequence within the node.
- Cycles should be rare.
 - Ideal graph is a "**VCF graph**": a linear reference with edits.
 - There should be a good **snarl decomposition**.
- Paths may represent references, haplotypes, variants, alignments, and annotations.
 - **Paths** are stored in the graph itself.
 - **Threads** are lightweight paths with limited functionality stored in a GBWT index.

Snarl decomposition

Snarl is an induced subgraph separated by two node sides from the rest of the graph.

Chain is a sequence of snarls.

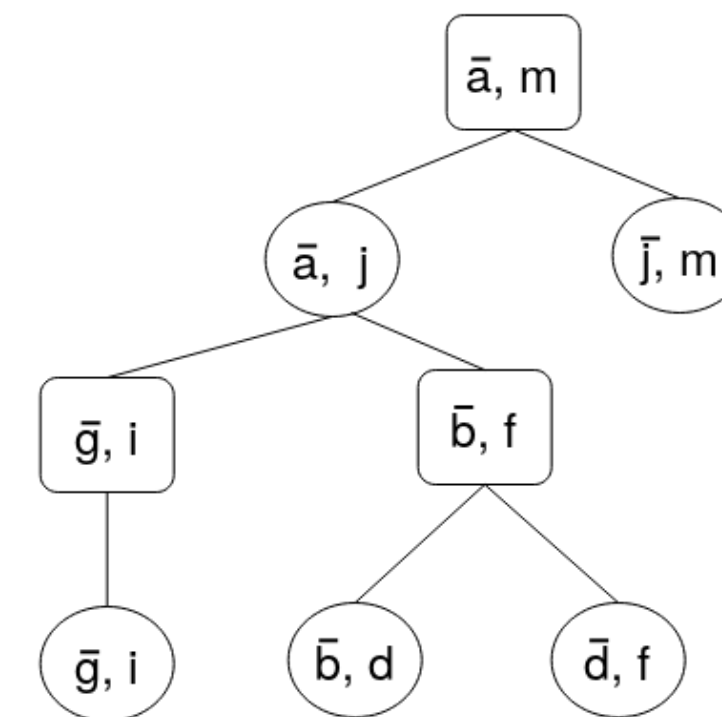
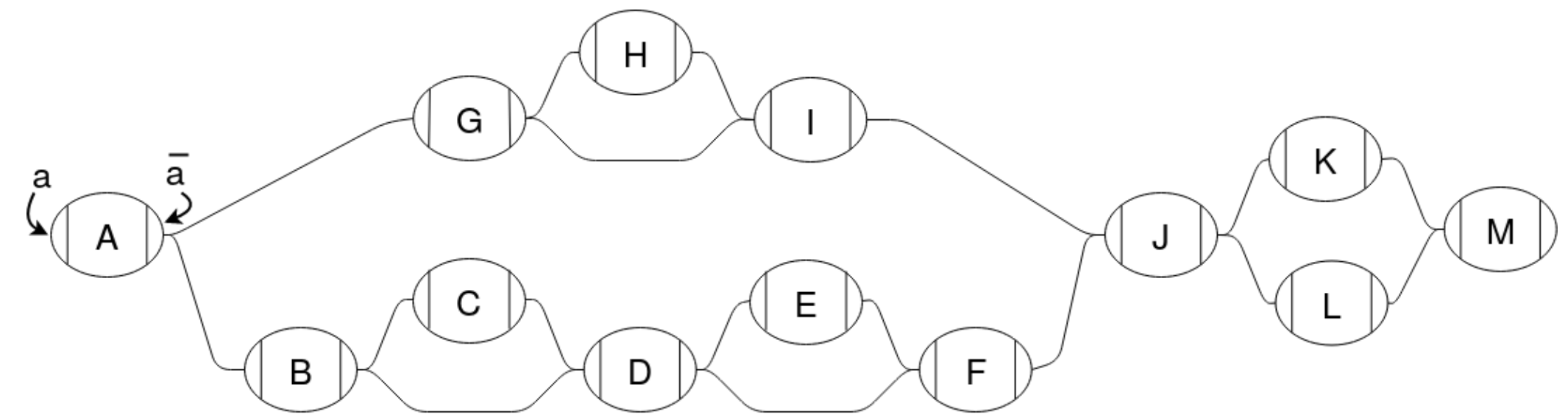
A snarl may contain multiple chains.

Many VG algorithms use the hierarchical decomposition defined by snarls and chains.

Paten et al.: **Superbubbles, Ultrabubbles, and Cacti.**

Journal of Computational Biology, 2018.

[DOI: 10.1089/cmb.2017.0251](https://doi.org/10.1089/cmb.2017.0251)



Graph implementations

libhandlegraph defines a common sequence graph interface based on **handles** that encode node id and orientation.

libbdsg contains graph implementations.

HashGraph is a hash table of node records with adjacency lists. It has replaced the Protobuf-based .vg as the default graph format.

XG is a smaller immutable graph based on bit-packed arrays.

Others: ODGI, PackedGraph, GBWTGraph.

Eizenga et al.: **Efficient dynamic variation graphs**. Bioinformatics, 2020.

[DOI: 10.1093/bioinformatics/btaa640](https://doi.org/10.1093/bioinformatics/btaa640)

<https://github.com/vgteam/libhandlegraph>

<https://github.com/vgteam/libbdsg>

GFA format

GFA (Graphical Fragment Assembly) is emerging as the standard interchange format for genome graphs.

A text-based TSV format.

Some features (e.g. overlaps between nodes) are mostly used by genome assemblers.

Others (e.g. rGFA tags, W-lines with haplotype metadata) are more useful for pangenomics tools.

<https://github.com/GFA-spec/GFA-spec/blob/master/GFA1.md>

example.gfa

```
H VN:Z:1.0
S 11 G
S 12 A
S 13 T
S 14 T
S 15 A
S 16 C
S 17 A
S 21 G
S 22 A
S 23 T
S 24 T
S 25 A
L 11 + 12 + *
L 11 + 13 + *
L 12 + 14 + *
L 13 + 14 + *
L 14 + 15 + *
L 14 + 16 + *
L 15 + 17 + *
L 16 + 17 + *
L 21 + 22 + *
L 21 + 23 + *
L 22 + 24 + *
L 23 + 24 - *
L 24 + 25 + *
P A 11+,12+,14+,15+,17+ *,*,*,*
P B 21+,22+,24+,25+ *,*,*
W sample 1 A 0 5 >11>12>14>15>17
W sample 2 A 0 5 >11>13>14>16>17
W sample 1 B 0 5 >21>22>24<23<21
W sample 2 B 0 4 >21>22>24>25
```

S	name	sequence				
L	from	orientation	to	orientation	overlap	
P	name	path	overlaps			
W	sample	haplotype	contig	begin	end	path

Indexes: GCSA2

GCSA2 is an FM-index for deterministic Wheeler graphs.

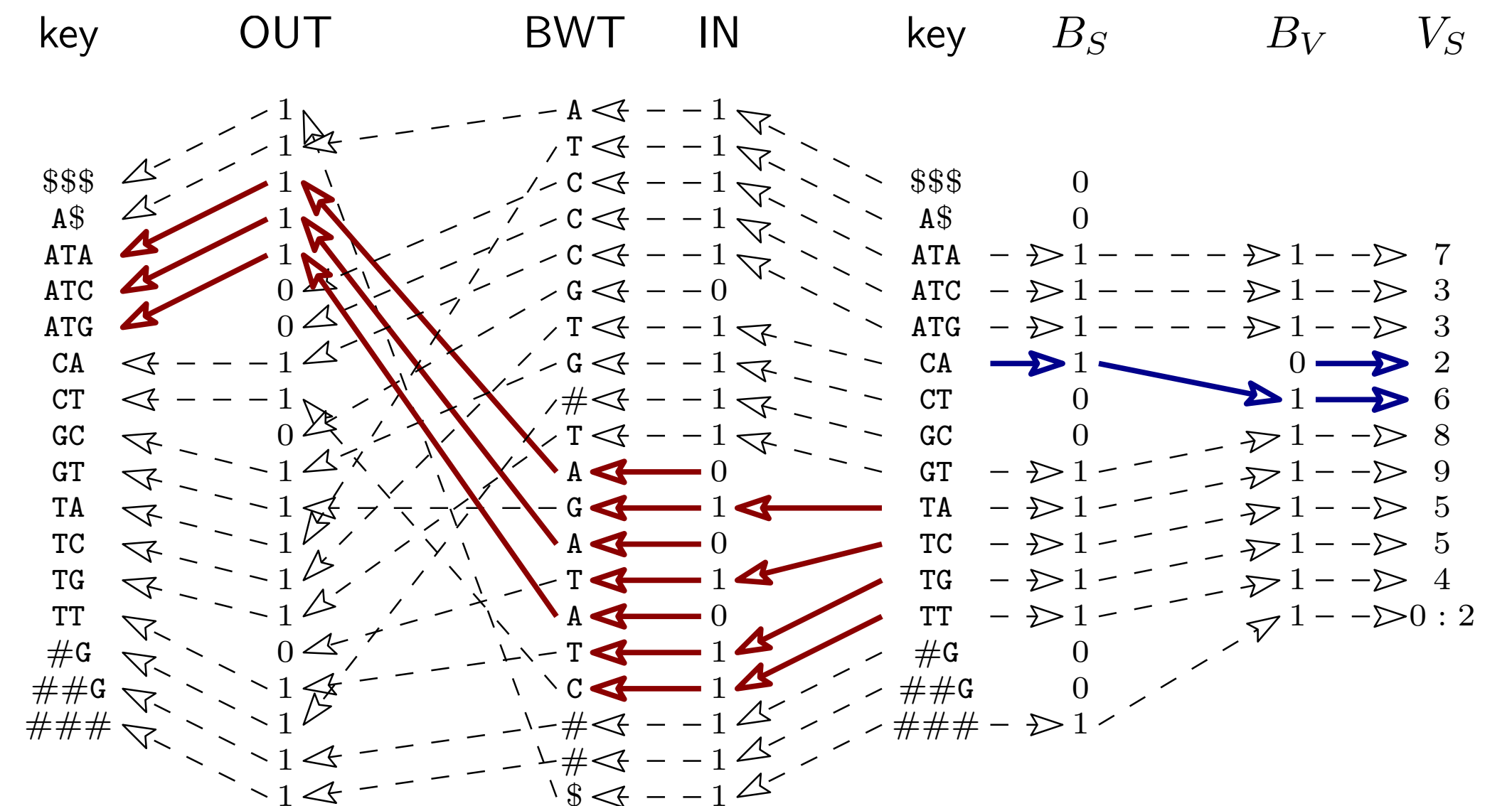
Older than Wheeler graphs: labeled nodes and LF-mapping follows incoming edges.

Because there is probably no equivalent Wheeler graph, we stop prefix-doubling at length 256. Longer matches may be false positives.

We usually index the graph in both orientations, allowing us to search for the pattern and its reverse complement at the same time.

There is also a CST based on PSV/NSV/RMQ queries on the LCP array. We mostly use it for `parent()` queries.

$$\text{BWT.rank}(\text{IN.select}(i, 1), c) = \text{B}_c.\text{rank}(i, 1)$$



Sirén: **Indexing variation graphs.**

ALENEX 2017.

[DOI: 10.1137/1.9781611974768.2](https://doi.org/10.1137/1.9781611974768.2)

<https://github.com/jltsiren/gcsa2>

Indexes: GBWT

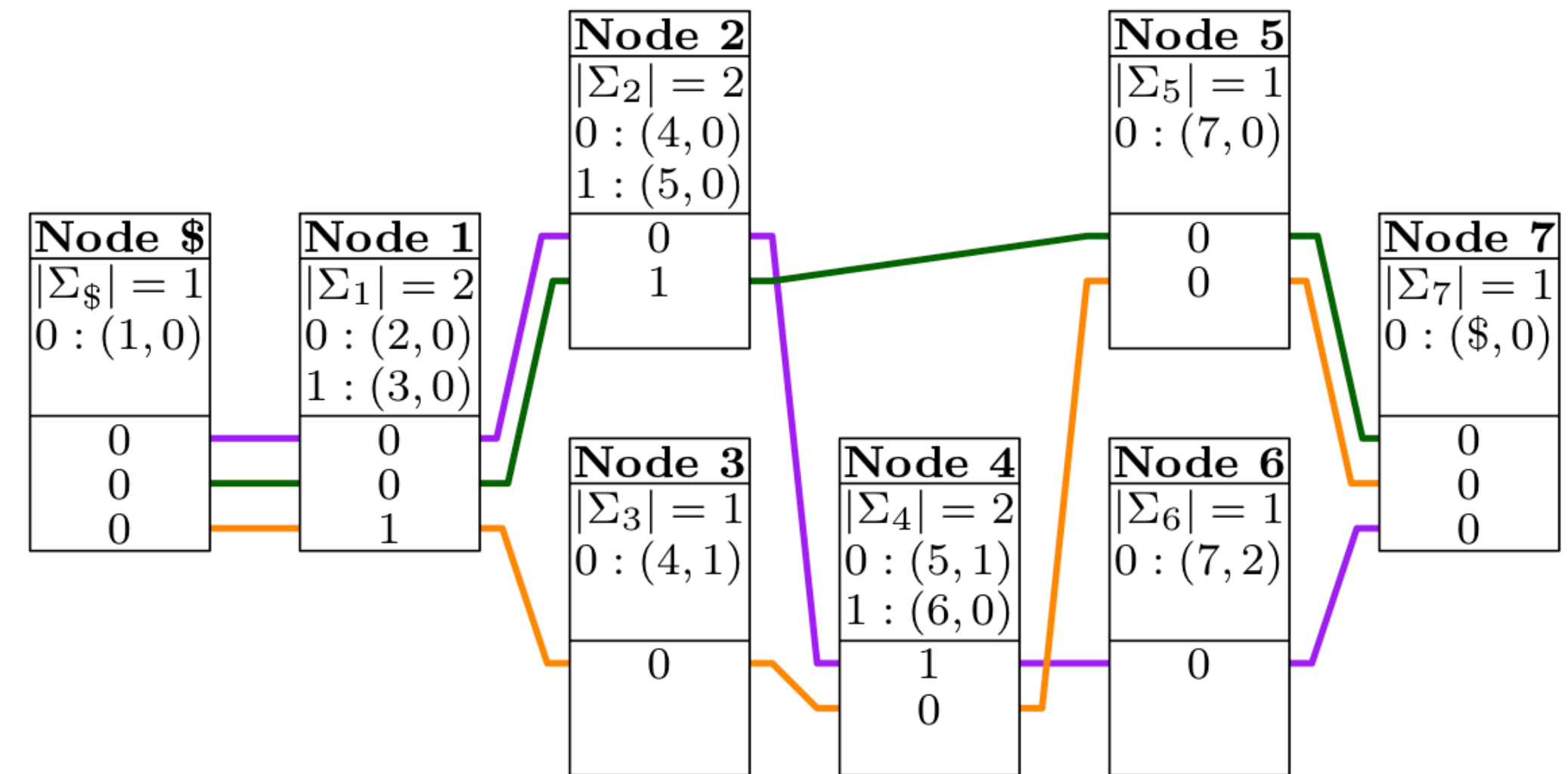
GBWT is a run-length encoded FM-index for integer sequences.

We choose to interpret the integers as nodes and the sequences as haplotype paths.

We partition the BWT into nodes and replace successor nodes with edge ranks.

Extensions: Bidirectional GBWT, r-index for fast document listing, cached GBWT for repeated traversals of small subgraphs...

GBWTGraph adds sequences to support the HandleGraph interface for the subgraph induced by the haplotype paths.



Sirén et al.: **Haplotype-aware graph indexes**. Bioinformatics, 2020.

[DOI: 10.1093/bioinformatics/btz575](https://doi.org/10.1093/bioinformatics/btz575)

<https://github.com/jltsiren/gbwt>

<https://github.com/jltsiren/gbwtgraph>

Indexes: Distance index

Determining the shortest distance between two graph positions can be slow.

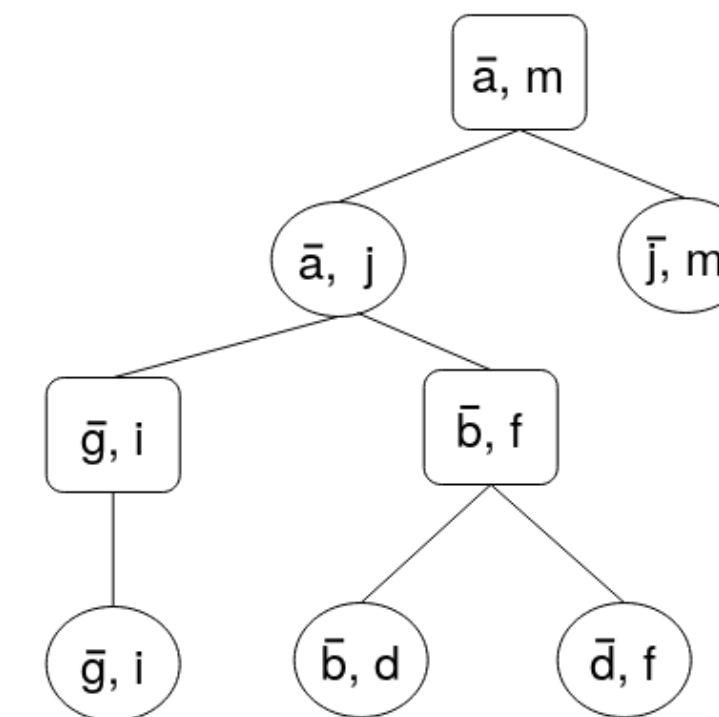
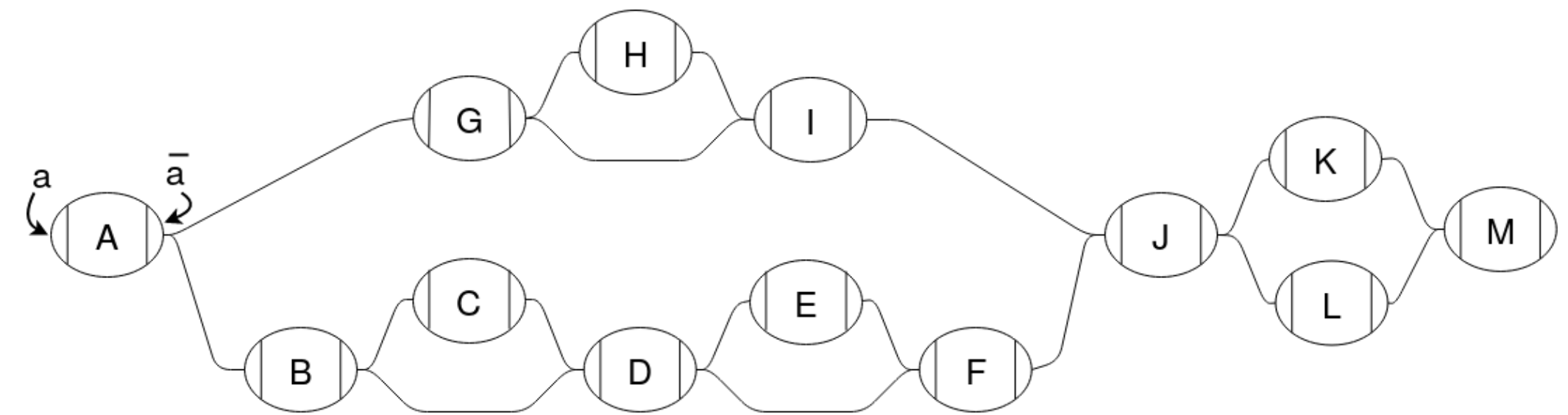
If we have the snarl decomposition of the graph, we can compute the distance faster in the snarl tree.

Useful for applications such as seed clustering that compute distances between a large number of pairs.

Chang et al.: **Distance indexing and seed clustering in sequence graphs.**

Bioinformatics, 2020.

[DOI: 10.1093/bioinformatics/btaa446](https://doi.org/10.1093/bioinformatics/btaa446)



Indexes: Minimizer index

A (w, k) -**minimizer** is the k -mer with the smallest hash value among the k -mers and their reverse complements in a $k + w - 1$ bp window.

Minimizers are a common (but slightly inelegant) way of selecting a subset of k -mers for indexing.

A minimizer index is a simple hash table mapping k -mers to sets of positions.

If the hits are sorted, we can easily restrict the query to a subgraph.

In the Giraffe mapper, the minimizer index also caches information from the distance index for positions close to the root of the snarl tree. This reduces cache misses when the hits are scattered around the graph.

Index construction is fast: typically 5–10 minutes for (the haplotypes in) a human graph.

Minimizer indexes tend to be faster and larger than FM-indexes and to work better with high error rates.

Graph Construction

VG basics

VG has a large number of **subcommands**.

General principles:

- Graphs can be in any HandleGraph format.
- Output graph is often written to stdout and filename - means stdin.
- Option **-p** often writes progress information to stderr.
- **vg <command>** prints usage information for that subcommand.
- The directory for temporary files can be set with environment variable **TMPDIR** (the examples use **`\${HOME}/scratch/tmp**).

```
$ vg
vg: variation graph tool, version v1.31.0 "Caffaraccia"

usage: vg <command> [options]

main mapping and calling pipeline:
  -- construct      graph construction
  -- index          index graphs or alignments for random access or mapping
  -- map            MEM-based read alignment
  -- giraffe        fast haplotype-aware short read alignment
  -- augment        augment a graph from an alignment
  -- pack           convert alignments to a compact coverage index
  -- call           call or genotype VCF variants
  -- help          show all subcommands

For more commands, type `vg help`.
For technical support, please visit: https://www.biostars.org/t/vg/

# Convert GFA to HashGraph, pipe it to vg stats, and print basic stats
$ vg convert -g -a example.gfa | vg stats -z -
nodes  12
edges  13

# Progress information from vg gbwt
$ vg gbwt -p -o example.gbwt -G example.gfa
Building input GBWTs
Input type: GFA
Opening GFA file example.gfa
Validating GFA file example.gfa
Found 12 segments, 2 paths, and 4 walks in 0.00111055 seconds
Storing reference paths as sample _gbwt_ref
GBWT insertion batch size: 464 nodes
Parsing segments
Parsed 12 nodes in 9.482e-06 seconds
Parsing metadata
Parsed metadata in 0.00121103 seconds
Metadata: 6 paths with names, 2 samples with names, 3 haplotypes, 2 contigs with names
Indexing paths/walks
Indexed 2 paths and 4 walks in 0.000557962 seconds
GBWTs built in 0.00379489 seconds, 0.0148582 GiB

Serializing the GBWT to example.gbwt
GBWT serialized in 0.00186231 seconds, 0.018631 GiB
```

Example data

We use 1000 Genomes Project chromosomes 19 and 20 with a total of 5008 haplotypes of length ~120 Mbp.

Large enough to demonstrate techniques used with whole-genome human graphs but small enough to work in 24 GiB memory.

We should now have the following files:

- **reference.fa**: GRCh37 reference genome.
- **chr19.vcf.gz** and **chr20.vcf.gz**: bgzip-compressed VCF files.
- **chr19.vcf.gz.tbi** and **chr20.vcf.gz.tbi**: Tabix indexes for the VCF files.

get-data.sh

```
#!/bin/bash

SOURCE="ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp"

# Get the reference
rm -f reference.fa
curl ${SOURCE}/technical/reference/human_g1k_v37.fasta.gz > reference.fa.gz
gunzip reference.fa.gz

# Get the VCFs
SUFFIX=.phase3_shapeit2_mvncall_integrated_v5b.20130502.genotypes.vcf.gz
for i in 19 20; do
  rm -f chr${i}.vcf.gz chr${i}.vcf.gz.tbi
  curl ${SOURCE}/release/20130502/ALL.chr${i}${SUFFIX} > chr${i}.vcf.gz
  curl ${SOURCE}/release/20130502/ALL.chr${i}${SUFFIX}.tbi > chr${i}.vcf.gz.tbi
done
```

Graph construction

`vg construct` builds a graph from a FASTA reference genome and a VCF file.

Because VCF semantics are ambiguous, VG has to make some arbitrary choices.

Relevant options:

- `-r FILE`: Reference genome.
- `-v FILE`: VCF file.
- `-R REGION`: Restrict to this region or contig.
- `-C`: The region is a chromosome (disables some heuristics).
- `-a`: Store variants as paths.

build-graphs.sh

```
#!/bin/bash

# Memory usage is low enough that we could build the graphs in parallel.
# There will be some warnings on unsupported variant types.
for i in 19 20; do
    vg construct -r reference.fa -v chr${i}.vcf.gz -R $i -C -a > chr${i}.vg
done
```

Some graph statistics

```
# Number of nodes, number of edges, total length of sequences
$ vg stats -z -l chr19.vg
nodes 6373442
edges 8341806
length 60981512
$ vg stats -z -l chr20.vg
nodes 6444645
edges 8376323
length 64854544

# Number of paths
$ vg paths -v chr19.vg -L | wc -l
3672810
$ vg paths -v chr20.vg -L | wc -l
3632493
```

Joint node id space

Because we built the graphs independently, they have overlapping node ids.

`vg ids` can join the node id spaces.

Relevant options:

- `-j`: Join node id spaces.
- `-m`: Create an empty node mapping (for some GCSA2 construction approaches).

Now we have the following files:

- `chr19.vg` and `chr20.vg`: Graphs in HashGraph format.
- `node_mapping`: Empty node mapping.

join-ids.sh

```
#!/bin/bash
vg ids -j -m node_mapping chr19.vg chr20.vg
# Create a copy just in case
cp node_mapping node_mapping.backup
```

More statistics

```
# Original node id ranges
$ vg stats -r chr19.vg
node-id-range 1:6373442
$ vg stats -r chr20.vg
node-id-range 1:6444645

# Join node id spaces
$ ./join-ids.sh

# Joint id ranges
$ vg stats -r chr19.vg
node-id-range 1:6373442
$ vg stats -r chr20.vg
node-id-range 6373443:12818087
```

XG construction

XG is an immutable graph format that is often used for combining single-chromosome graphs into a whole-genome graph.

For historical reasons, it is called the XG index and built with `vg index -x`.

Option `-L` includes the alt paths (variants) created by `vg construct -a` in the XG. This is necessary for some tasks but makes the graph unnecessarily large for other tasks.

We create the following files:

- `all.xg`: Combined chr19 / chr20 graph.
- `variants.xg`: Combined graph with variants.

build-xg.sh

```
#!/bin/bash

# XG construction uses large memory-mapped files.
export TMPDIR=${HOME}/scratch/tmp

# XG with reference paths
vg index -x all.xg chr19.vg chr20.vg

# XG with reference paths and variants
vg index -x variants.xg -L chr19.vg chr20.vg
```

Graph statistics

```
# Number of nodes, number of edges, total length of sequences
$ vg stats -z -l all.xg
nodes 12818087
edges 16718129
length 125836056
$ vg stats -z -l variants.xg
nodes 12818087
edges 16718129
length 125836056

# Number of paths
$ vg paths -v all.xg -L | wc -l
2
$ vg paths -v variants.xg -L | wc -l
7305303
```


GFA import / export

VG relies on other tools (**minigraph**, **Cactus**, **pgggb**) for building graphs from assembled genomes.

`vg convert` can convert between various graph formats, including GFA.

Some `vg convert` options:

- `-g`: The input is in GFA format.
- `-a`: Convert to HashGraph.
- `-f`: Convert to GFA.

```
# Convert GFA to HashGraph
$ vg convert -g -a example.gfa > example.vg

# Convert HashGraph to GFA
$ vg convert -f example.vg > converted.gfa

# Number of segments
$ grep -c "^S" example.gfa
12
$ grep -c "^S" converted.gfa
12

# Number of links
$ grep -c "^L" example.gfa
13
$ grep -c "^L" converted.gfa
13

# Number of paths
$ grep -c "^P" example.gfa
2
$ grep -c "^P" converted.gfa
2

# Number of walks
$ grep -c "^W" example.gfa
4
$ grep -c "^W" converted.gfa
0
```

<https://github.com/lh3/minigraph>

<https://github.com/ComparativeGenomicsToolkit/cactus>

<https://github.com/pangenome/pggb>

Index Construction

The easy way

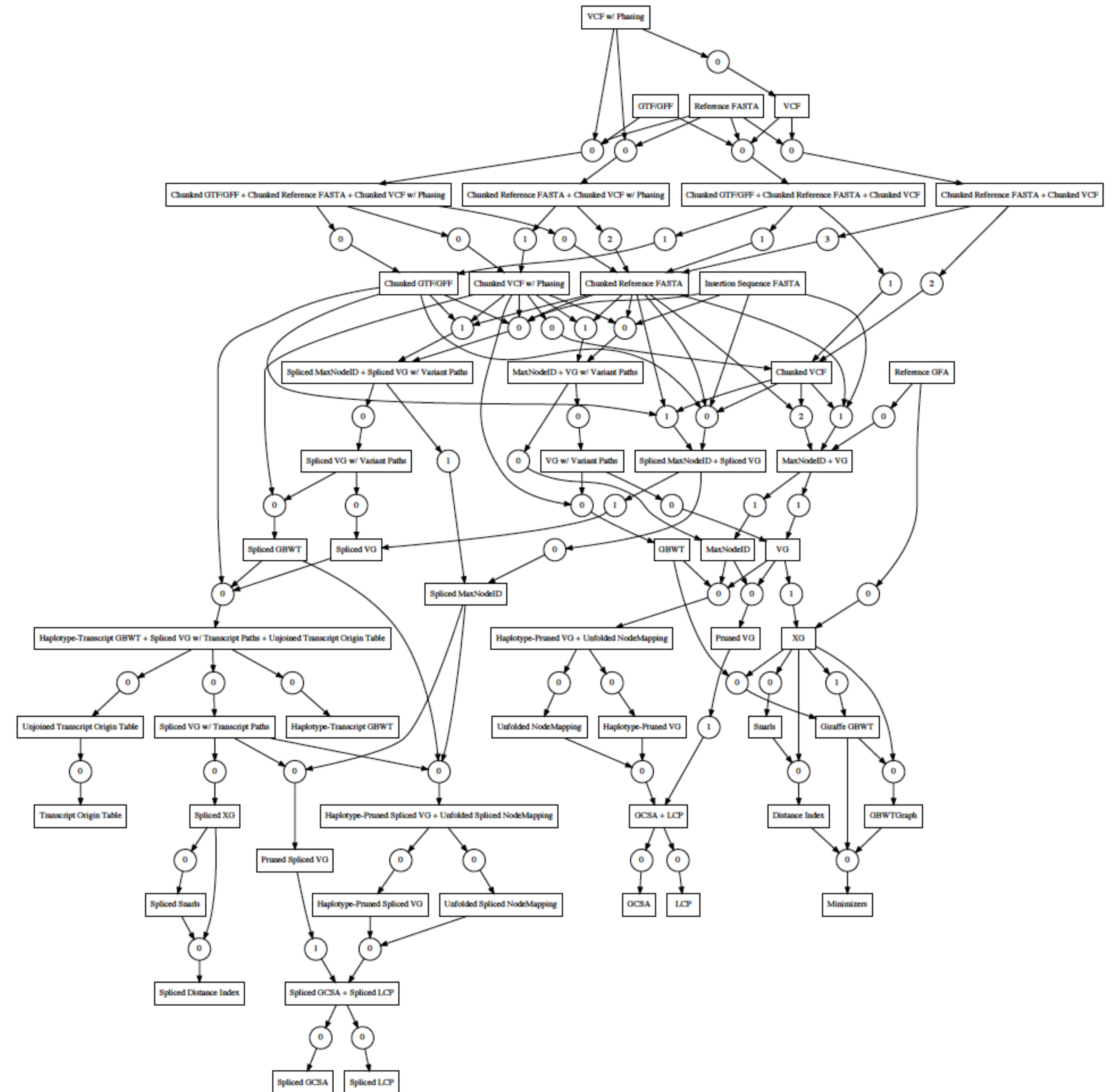
The graph on the right describes the dependencies between common inputs and index types.

Given the name of the workflow and a set of inputs, `vg autoindex` tries to figure out how to build the necessary indexes.

This does not scale efficiently to whole-genome human graphs yet.

Instead of using `vg autoindex`, we build the indexes manually.

<https://github.com/vgteam/vg/wiki/Index-Construction>



GBWT construction

- The basic algorithm inserts a batch of sequences into a dynamic FM-index.
 - Based on the **BCR** algorithm (Bauer et al, TCS, 2013) and **RopeBWT2** (Li, Bioinformatics, 2014).
- We can partition the construction by chromosome, run multiple jobs in parallel, and merge the results quickly.
 - In each job, the main thread generates paths, while a background thread inserts them into the GBWT index.
 - Because the alphabets are disjoint, merging is almost trivial.
- When the input consists of VCF files, we match paths with VCF contigs by name and assume that they are disjoint.
 - As VCF parsing is slow, we first parse the input into temporary files.
 - VCF is a variant-based format, while we generate full paths before inserting them (and their reverse complements).
 - To save memory, we make multiple passes over the parse and generate paths in batches of e.g. 200 samples.

VCF to GBWT

Some `vg gbwt` options:

- `-x FILE`: Use this graph.
- `-v`: The inputs are VCF files.
- `-o FILE`: Write the GBWT to this file.
- `--preset X`: Use this preset. `1000gp` is good for generating full-length haplotypes for human graphs.
- `--num-jobs N`: Limit the number of parallel jobs to save memory.

We use `variants.xg` and the VCF files and build `all.gbwt` with 5008 haplotypes over chr19 and chr20.

build-gbwt.sh

```
#!/bin/bash

# For VCF parses and temporary GBWTs
export TMPDIR=${HOME}/scratch/tmp

# This will take a few hours.
vg gbwt -x variants.xg -o all.gbwt --preset 1000gp -v chr19.vcf.gz chr20.vcf.gz
```

GBWT statistics

```
# Basic GBWT metadata
$ vg gbwt -M all.gbwt
10016 paths with names, 2504 samples with names, 5008 haplotypes, 2 contigs with names

# Extract the GBWT from the VG wrapper
$ vg view -x GBWT all.gbwt > extracted.gbwt

# Print more information using standalone GBWT tools
$ gbwt/benchmark extracted
GBWT benchmark v1.2.0

Index name:          extracted

Compressed GBWT:    extracted (bidirectional)
Total length:      92564668356
Sequences:         20032
Alphabet size:     25636176
Effective:         25636175
Runs:              131479952 concrete / 131499980 logical
DA samples:        90407892
BWT:               363.601 MB
DA samples:        310.42 MB
Total:             674.199 MB
Metadata:          10016 paths with names, 2504 samples with names, 5008
haplotypes, 2 contigs with names
```

GFA to GBWT

When building GBWT from GFA with both P-lines and W-lines, VG interprets P-lines as reference paths and W-lines as threads.

If there are only P-lines, VG interprets them as threads and parses GBWT metadata from path names.

More `vg gbwt` options:

- `-G`: The input is a GFA file.
- `-g FILE`: Build GBWTGraph and write it to this file.

GBWTGraphs can be converted into other graph formats with `vg convert` option `-b`.

```
# Build GBWT and GBWTGraph
$ vg gbwt -o example.gbwt -g example.gg -G example.gfa

# GBWT metadata
$ vg gbwt -M example.gbwt
6 paths with names, 2 samples with names, 3 haplotypes, 2 contigs with names

# Thread names from GBWT
$ vg gbwt -T example.gbwt
_thread_gbwt_ref_A_0_0
_thread_gbwt_ref_B_0_0
_thread_sample_A_1_0
_thread_sample_A_2_0
_thread_sample_B_1_0
_thread_sample_B_2_0

# Convert GBWTGraph to HashGraph
$ vg convert -b example.gbwt -a example.gg > example.vg

# Path names from HashGraph
$ vg paths -v example.vg -L
A
B
```

Sampled GBWT

When we add variants to a graph, both true and false positive mappings become more likely (Pritt et al., Genome Biology, 2018).

VG usually gets best results with variants that occur in at least ~1% of the haplotypes.

`vg gbwt -l` generates `n` (default `n = 64`) artificial paths per graph component with a greedy algorithm that tries to sample `k`-node (default `k = 4`) subpaths in the input GBWT proportionally.

We now take `all.xg` and `all.gbwt` and produce `sampled.gbwt` and `sampled.gg`.

sample-gbwt.sh

```
#!/bin/bash
# Sample the GBWT and build the corresponding GBWTGraph
vg gbwt -x all.xg -o sampled.gbwt -g sampled.gg -l all.gbwt
```

GBWT statistics

```
# Full GBWT metadata
$ vg gbwt -M all.gbwt
10016 paths with names, 2504 samples with names, 5008 haplotypes, 2 contigs with names

# Sampled GBWT metadata
$ vg gbwt -M sampled.gbwt
128 paths with names, 64 samples, 64 haplotypes, 2 contigs

# File sizes
$ ll -h all.gbwt sampled.gbwt
-rw-rw-r-- 1 parallels parallels 675M Mar 28 16:20 all.gbwt
-rw-rw-r-- 1 parallels parallels 173M Mar 28 19:11 sampled.gbwt
```

Graph pruning

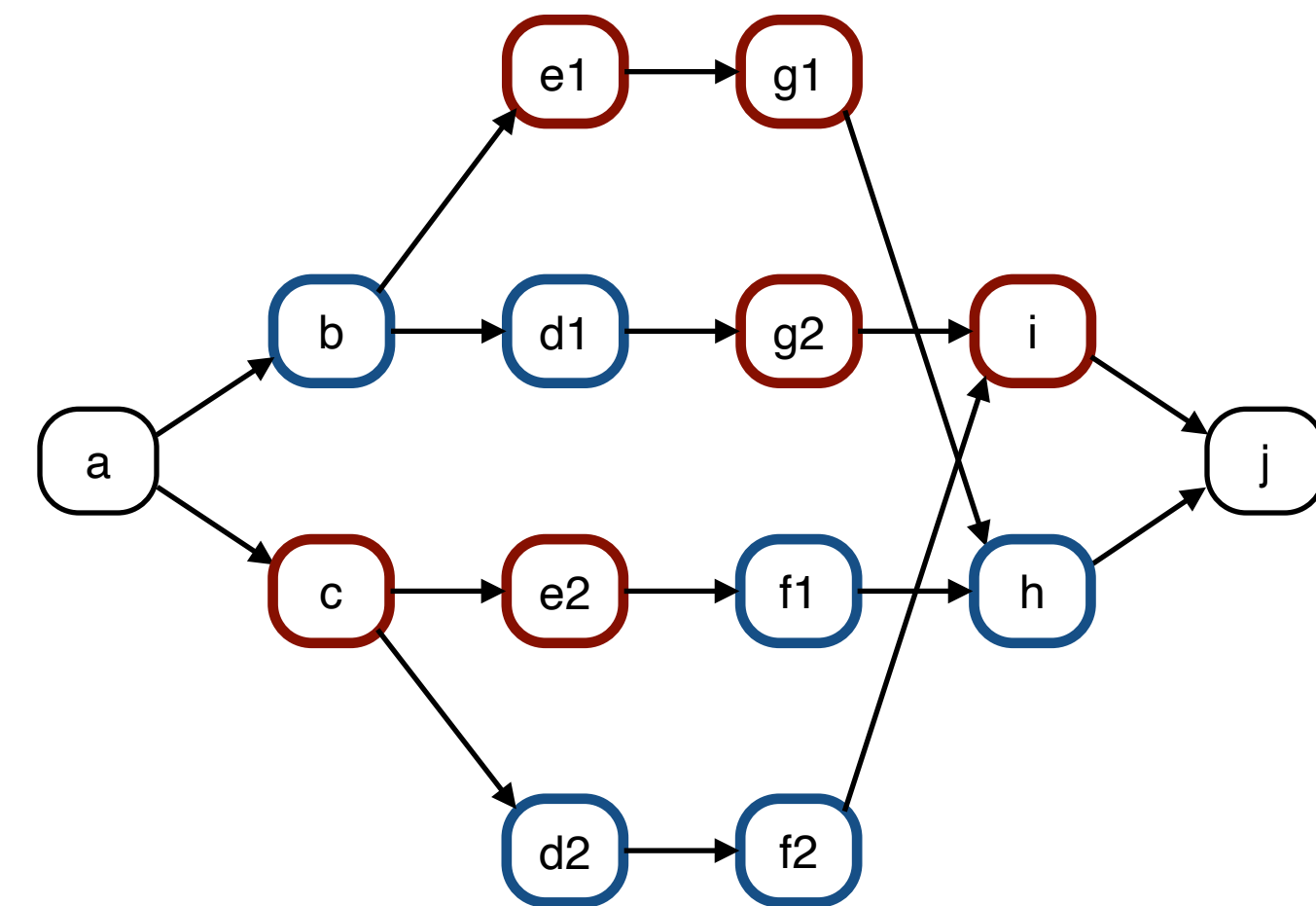
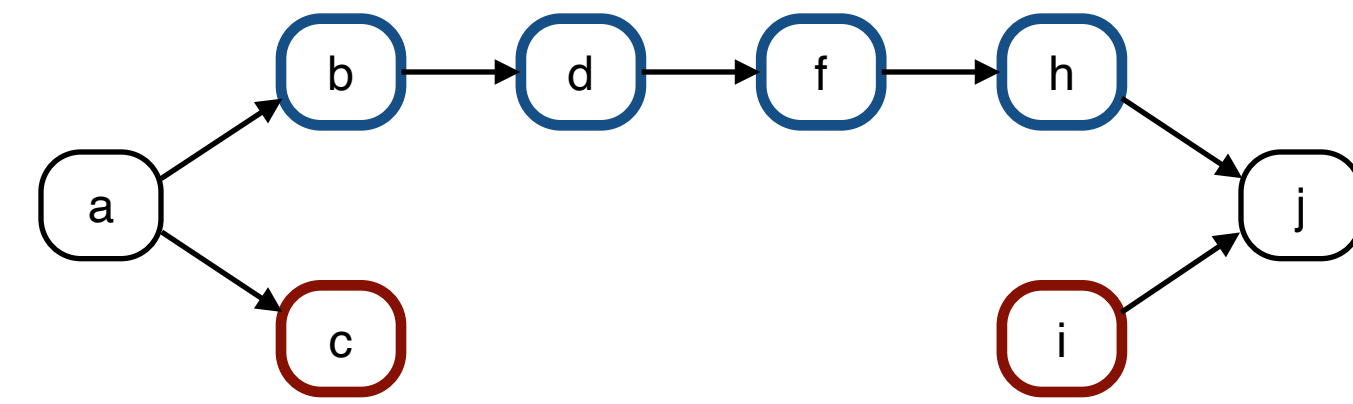
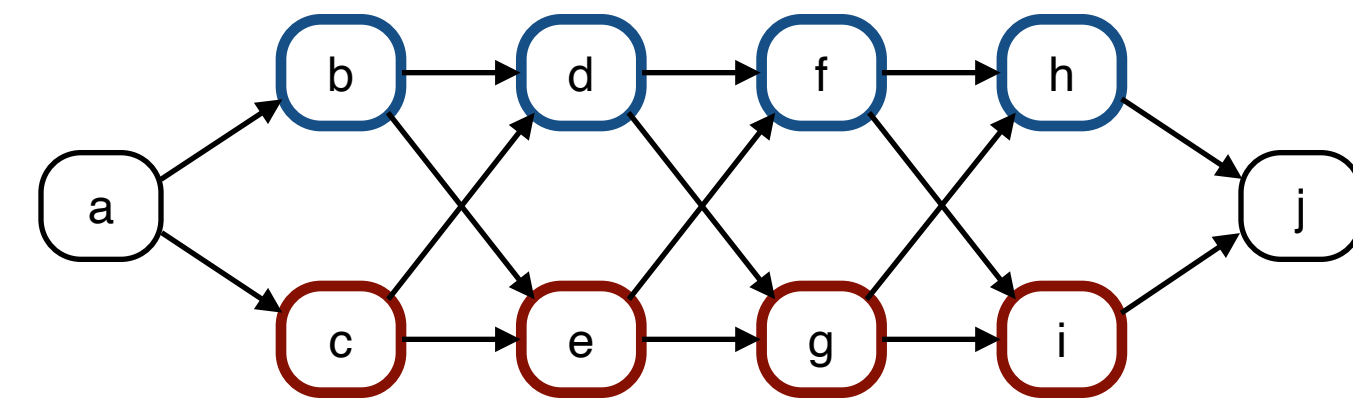
For GCSA2, we need a deterministic Wheeler graph that is 256-equivalent to the original.

If there is too much local complexity in the graph, the Wheeler graph may be too large.

We **prune** graph regions where **k**-mers (default **k** = 24) make more than **e** (default **e** = 3) nontrivial edge choices.

Then we may remove high-degree nodes, restore reference paths, or unfold local haplotypes in the pruned regions.

We build GCSA for the pruned graph and use the index with the original graph.



Pruning in practice

Haplotype unfolding creates duplicate nodes. We use **node mapping** to keep track of their original ids and to create a GCSA index that maps to the original graph.

Some `vg prune` options:

- `-u`: Unfold haplotypes.
- `-g FILE`: Use this GBWT.
- `-m FILE`: Use this node mapping.
- `-a`: Append to an existing node mapping.

We use `chr19.vg`, `chr20.vg`, and `all.gbwt` to produce pruned graphs `chr19.unfolded.vg` and `chr20.unfolded.vg`.

prune-graphs.sh

```
#!/bin/bash

# For building temporary XG indexes
export TMPDIR=${HOME}/scratch/tmp

# Prune and unfold haplotypes
for i in 19 20; do
    vg prune -u -g all.gbwt -a -m node_mapping chr${i}.vg > chr${i}.unfolded.vg
done
```

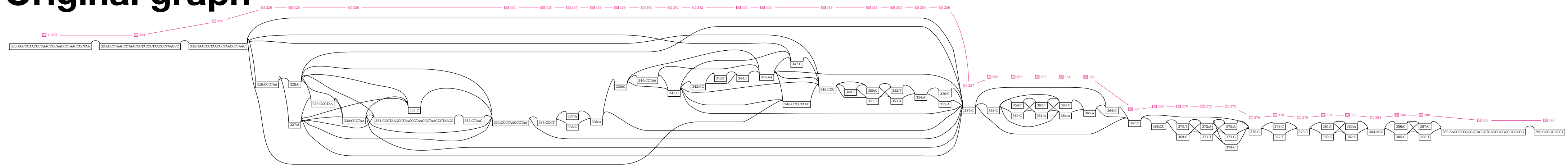
Graph statistics

```
# Original graphs
$ vg stats -z -l chr19.vg
nodes 6373442
edges 8341806
length 60981512
$ vg stats -z -l chr20.vg
nodes 6444645
edges 8376323
length 64854544

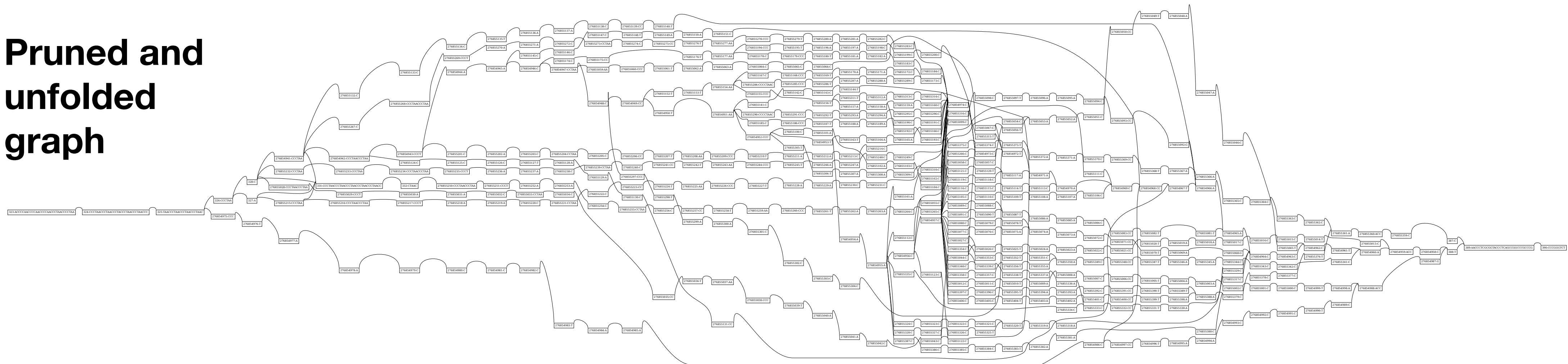
# Pruned graphs
$ vg stats -z -l chr19.unfolded.vg
nodes 7306423
edges 9238425
length 64341208
$ vg stats -z -l chr20.unfolded.vg
nodes 6996528
edges 8891383
length 66934937
```

Degenerate cases

Original graph



Pruned and unfolded graph



The part with many edges connects the trie of haplotype prefixes to the trie of reverse suffixes. There is a high degree of nondeterminism: almost every leaf has label C.

GCSA2 construction

GCSA2 construction uses a semi-external **prefix-doubling** algorithm.

We first generate all 16 bp paths in the graph and store them in one file per chromosome.

With four prefix-doubling steps, we increase path length to (up to) 256 bp.

Each step starts with a **pruning** step that processes the paths in lexicographic order, marks the ones that no longer have to be extended, and merges ranges of redundant paths.

The **extend** step reads the paths from one chromosome at a time and generates new paths to a new file.

Prefix-doubling is followed by the **merging** step, which is similar to the pruning step.

Final construction is done in a single pass over the merged path files, with σ additional read pointers completing a second pass by keeping track of the results of LF-mapping.

GCSA in practice

GCSA construction typically needs hundreds of bytes of disk space per bp, and the total I/O volume is kilobytes per bp.

The progress information from option `-p` can help when something goes wrong.

Relevant `vg index` options:

- `-g FILE`: Build GCSA and store it in `FILE` and `FILE.lcp`.
- `-f FILE`: Use this node mapping.

We use the unfolded graphs and the node mapping as inputs and produce `all.gcsa` and `all.gcsa.lcp`.

build-gcsa.sh

```
#!/bin/bash

# We need almost 60 GiB of disk space
export TMPDIR=${HOME}/scratch/tmp

vg index -g all.gcsa -f node_mapping -p chr19.unfolded.vg chr20.unfolded.vg
```

GCSA statistics

```
# Extract the GCSA and the LCP from VG wrappers
$ vg view -x GCSA all.gcsa > extracted.gcsa
$ vg view -x LCP all.gcsa.lcp > extracted.lcp

# Print some information using standalone GCSA2 tools
$ gcsa2/benchmark/query_gcsa extracted
GCSA2 query benchmark v1.3.0

Base name:          extracted

Paths:              486992624
Edges:              501419052
Samples:            67249859 (at 52403160 positions, 35 bits each)
Max query:          256

Core index:         328.545 MB
Samples:            355.358 MB
Counter:            108.958 MB
LCP array:          471.804 MB
Total size:         1264.67 MB
```

Distance index

Distance index construction is still a bit inconvenient.

First we have to find snarls in the graph with `vg snarls` and include the trivial ones with option `-T`.

Then we use `vg index`:

- `-s FILE`: Use these snarls.
- `-j FILE`: Build a distance index and store it in this file.

We need the combined graph `all.xg` and build distance index `all.dist`.

build-dist.sh

```
#!/bin/bash

# Find snarls, including trivial ones
vg snarls -T all.xg > all.snarls

# Build the distance index
vg index -s all.snarls -j all.dist all.xg
```

Minimizer index

Minimizer index construction is fast, and the tools using it can also build it on their own.

If we do not have a GBWTGraph, the construction will create a temporary one.

`vg minimizer` options:

- `-g FILE`: Use this GBWT.
- `-i FILE`: Store the index in this file.
- `-d FILE`: Annotate the hits with positions from this distance index.
- `-G`: The input graph is GBWTGraph.

We build `sampled.min` from `sampled.gbwt`, `sampled.gg`, and `all.dist`.

```
# Build the minimizer index and display progress information
$ vg minimizer -g sampled.gbwt -i sampled.min -d all.dist -G -p sampled.gg
Loading GBWT index sampled.gbwt
Loading GBWTGraph sampled.gg
Loading MinimumDistanceIndex all.dist
Building MinimizerIndex with k = 29, w = 11
19415804 keys (18831937 unique)
Minimizer occurrences: 21954507
Load factor: 0.578636
Construction so far: 11.1741 seconds
Writing the index to sampled.min
Time usage: 13.7561 seconds
Memory usage: 2.49887 GiB

# List the indexes and other outputs
$ ll -h all.* sampled.*
-rw-rw-r-- 1 parallels parallels 577M Mar 29 22:05 all.dist
-rw-rw-r-- 1 parallels parallels 675M Mar 28 16:20 all.gbwt
-rw-rw-r-- 1 parallels parallels 793M Mar 28 21:00 all.gcsa
-rw-rw-r-- 1 parallels parallels 472M Mar 28 21:00 all.gcsa.lcp
-rw-rw-r-- 1 parallels parallels 27M Mar 29 22:03 all.snarls
-rw-rw-r-- 1 parallels parallels 528M Mar 27 22:54 all.xg
-rw-rw-r-- 1 parallels parallels 173M Mar 28 19:11 sampled.gbwt
-rw-rw-r-- 1 parallels parallels 322M Mar 28 19:12 sampled.gg
-rw-rw-r-- 1 parallels parallels 821M Mar 29 22:12 sampled.min
```

Short Read Mapping

VG read aligners

vg map

The original short read aligner based on finding MEMs using GCSA2. The algorithm is similar to BWA-MEM.

Garrison et al.: **Variation graph toolkit improves read mapping by representing genetic variation in the reference.**

Nature Biotechnology, 2018.

[DOI: 10.1038/nbt.4227](https://doi.org/10.1038/nbt.4227)

Giraffe

Minimizer-based short read aligner that maps the reads to haplotype paths. Much faster than vg map.

Sirén et al.: **Genotyping common, large structural variations in 5,202 genomes using pangenomes, the Giraffe mapper, and the vg toolkit.**

bioRxiv, 2021.

[DOI: 10.1101/2020.12.04.412486](https://doi.org/10.1101/2020.12.04.412486)

vg mpmap

Similar to vg map, but the alignments are directed subgraphs instead of paths. Primarily for RNA-seq reads, so we skip it now.

Sibbesen et al.: **Haplotype-aware pantranscriptome analyses using spliced pangenome graphs.**

bioRxiv, 2021.

[DOI: 10.1101/2021.03.26.437240](https://doi.org/10.1101/2021.03.26.437240)

Output formats

GAM

The original Protobuf-based alignment format used by VG. Poorly documented and being phased out but still the default.

GAMP

The version of GAM used by `vg mppmap`.

GAF

Portable text-based TSV format. Downstream tools are more likely to support this.

<https://github.com/lh3/gfatools/blob/master/doc/rGFA.md>

VG can also output alignments in the standard SAM / BAM / CRAM formats.

Because these formats represent alignments relative to linear reference sequences, VG must project the alignments relative to a set of reference paths. This is a lossy process.

For historical reasons, this is called **subjecting** the alignments to reference paths.

Simulating reads

Because our example graph only contains two chromosomes, we have to use simulated reads.

We simulate 1 million 150 bp read pairs from 1000GP sample NA12878 with `vg sim`.

In a proper experiment, we would annotate the reads with reference positions and remove NA12878 and its close relatives from the graph we are mapping to.

We might also want to take the error profile from real reads.

We need `all.xg` and `all.gbwt` and output `reads.fq`.

simulate-reads.sh

```
#!/bin/bash

# 1 million pairs of 150 bp reads
PAIRS=1000000
LENGTH=150

# Substitution rate 0.003, indel rate 0.0003
SUBST=0.003
INDEL=0.0003

# Fragment length 500 bp, stdev 50 bp
FRAGMENT=500
STDEV=50

# Simulate the reads
vg sim -r -x all.xg -g all.gbwt -m NA12878 \
  -n $PAIRS -l $LENGTH \
  -e $SUBST -i $INDEL \
  -p $FRAGMENT -v $STDEV \
  -a > reads.gam

# Convert to FASTQ
vg view -X reads.gam > reads.fq
```

vg map algorithm

vg map uses a **seed-and-extend** algorithm similar to the one used in BWA-MEM (Li, arXiv, 2013).

1. Find (super-) **maximal exact matches** (MEMs) using GCSA2.
 - **LF()** extends the match to the left, **parent()** removes characters from the right (Ohlebusch et al., SPIRE 2010).
2. **Cluster** the seed MEMs and **chain** them in each promising cluster.
 - Distances are based on projections of the MEMs to the embedded paths.
3. **Unfold** the subgraphs around promising chains and transform them into DAGs.
4. Align the read to the DAG using an extension of the **Smith–Waterman algorithm**.

Using vg map

We need `all.xg`, `all.gcsa`, and `all.gcsa.lcp`, and we also use `all.gbwt` for better mapq estimation. The output will be `map.gam`.

Some `vg map` options:

- `-d STR`: Use this base name for indexes (or specify with `-x`, `-g`, and `-1`).
- `-t N`: Number of parallel mapping threads (one extra thread for output).
- `-f FILE`: Map the reads from this FASTQ file (can be gzip-compressed).
- `-i`: We have interleaved paired-end reads (or specify another FASTQ file).
- `-%`: Output GAF format.

```
# Map the reads using 7 threads
$ vg map -d all -t 7 -f reads.fq -i > map.gam

# Print some alignment statistics
$ vg stats -a map.gam variants.xg
Total alignments: 2000000
Total primary: 2000000
Total secondary: 0
Total aligned: 2000000
Total perfect: 1249207
Total gapless (softclips allowed): 1952126
Insertions: 46018 bp in 46002 read events
Deletions: 3219 bp in 3201 read events
Substitutions: 890472 bp in 890472 read events
Softclips: 2676 bp in 824 read events
Unvisited nodes: 4270702/12818087 (16047334 bp)
Single-visited nodes: 1622640/12818087 (17006534 bp)
Significantly biased heterozygous sites: 164630/3484484 (4.72466%)
```

Giraffe algorithm

The Giraffe algorithm is a custom seed-and-extend algorithm. With a few exceptions, it aligns the reads to haplotype paths while avoiding other paths.

1. Get seeds from the minimizer index.
 - Use all minimizers with at most 10 hits and some with up to 500 hits.
2. Cluster the seeds using the distance index.
3. Extend the seeds in promising clusters without gaps.
 - Most Illumina sequencing errors are substitutions, so most alignments should be gapless.
4. Align the best extensions from promising clusters using the **dozeu library**.
 - <https://github.com/ocxtal/dozeu>

Using Giraffe

We use `sampled.gg`, `sampled.gbwt`, `all.dist`, and `sampled.min` and output `giraffe.gam`.

Some `vg giraffe` options:

- `-g FILE`: Use this GBWTGraph (or `-x` for other graph types).
- `-H FILE`: Use this GBWT index.
- `-d FILE`: Use this distance index.
- `-m FILE`: Use this minimizer index (may be omitted).
- `-t N, -f FILE, -i`: As in `vg map`.
- `-o gaf`: Output GAF format.

```
# Map the reads using 7 threads
$ vg giraffe -g sampled.gg -H sampled.gbwt -d all.dist -m sampled.min -t 7 -f
reads.fq -i > giraffe.gam

# Print some alignment statistics
$ vg stats -a giraffe.gam variants.xg
Total alignments: 2000000
Total primary: 2000000
Total secondary: 0
Total aligned: 2000000
Total perfect: 1238868
Total gapless (softclips allowed): 1953675
Insertions: 44603 bp in 44417 read events
Deletions: 3762 bp in 3277 read events
Substitutions: 914710 bp in 914710 read events
Softclips: 560 bp in 82 read events
Unvisited nodes: 4295656/12818087 (16185980 bp)
Single-visited nodes: 1598937/12818087 (16965505 bp)
Significantly biased heterozygous sites: 166803/3484484 (4.78702%)
```

After mapping

- Genotyping and variant calling:
 - <https://github.com/vgteam/vg/wiki/Whole-genome-calling-and-genotyping>
 - <https://github.com/vgteam/vg/blob/master/README.md>
- GAM file filtering and manipulation with `vg filter`.