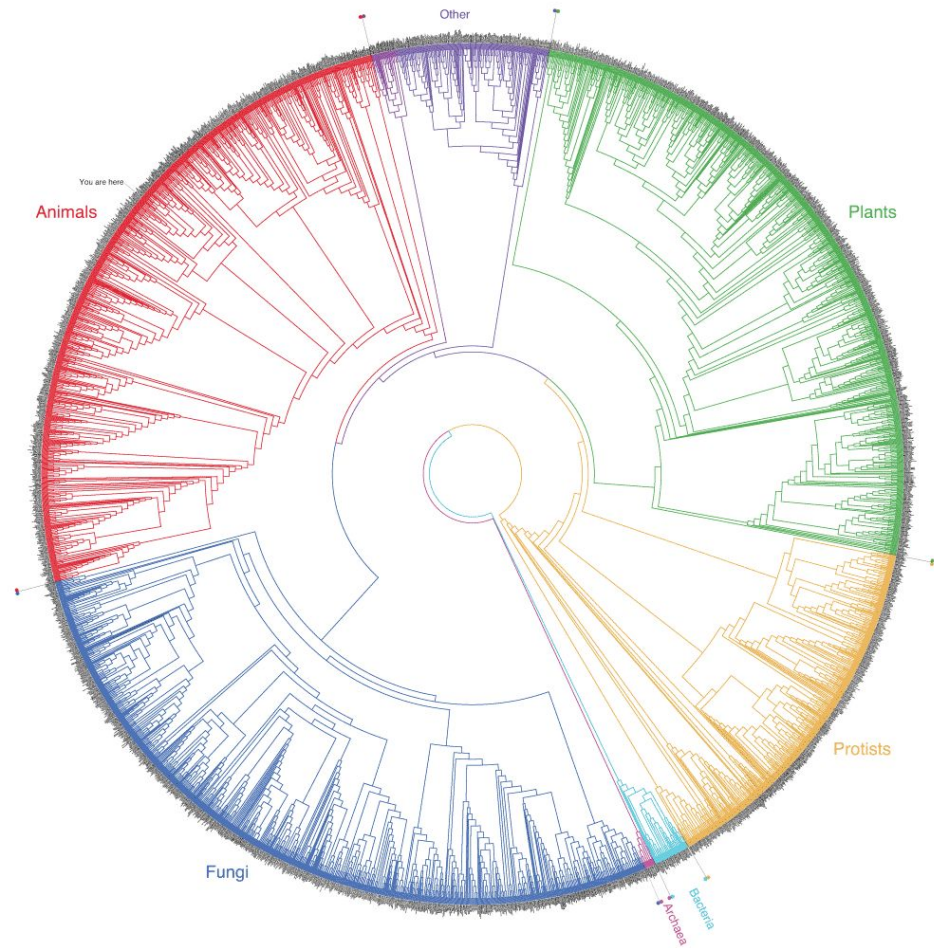


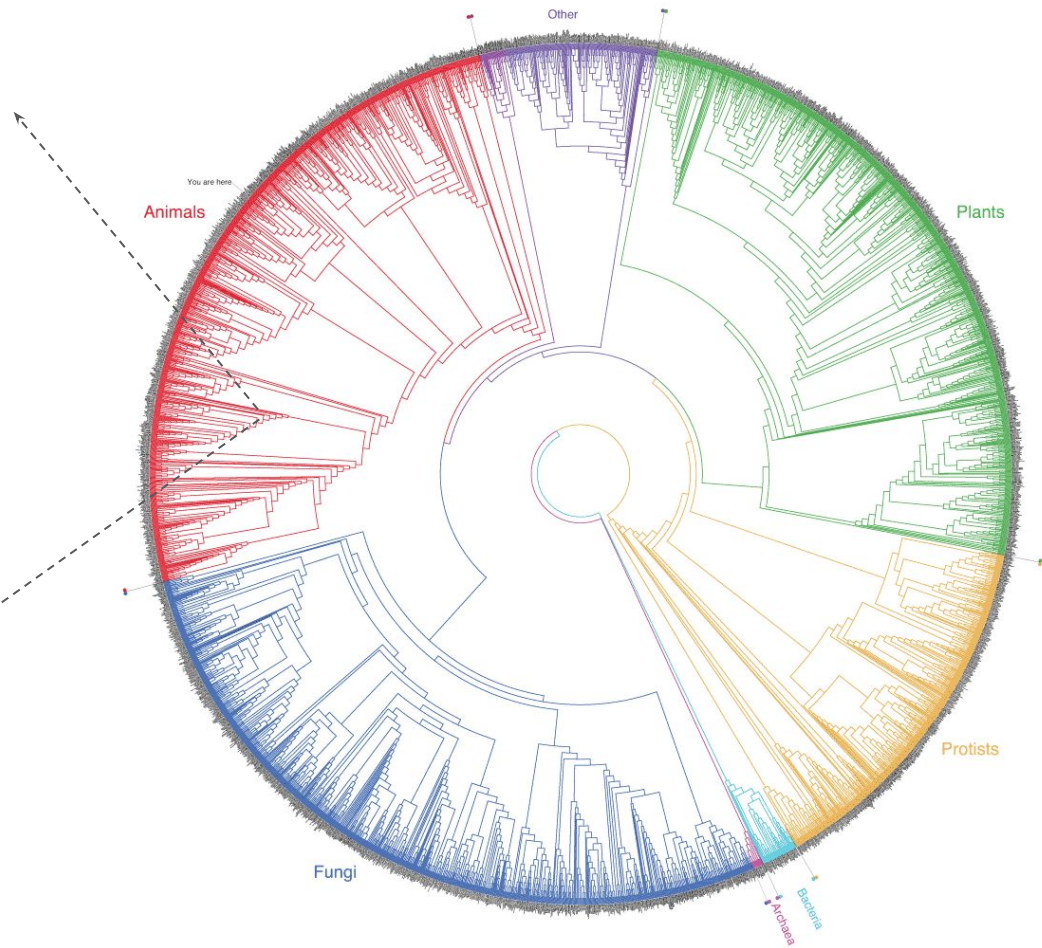
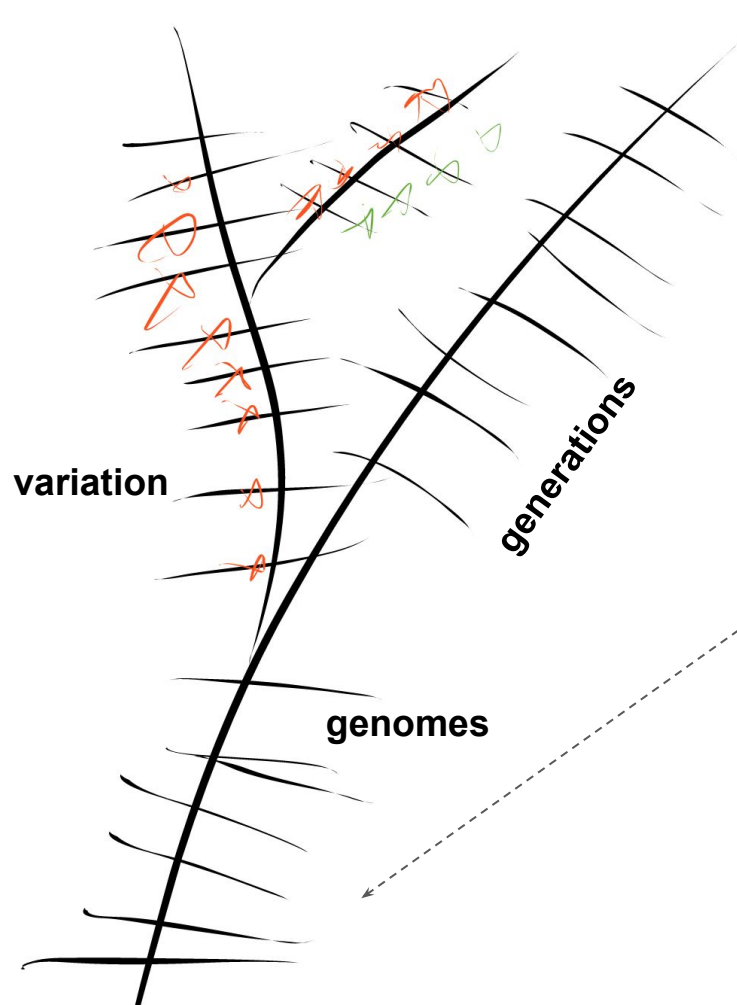
# Computational Pangenomics #CPANG18

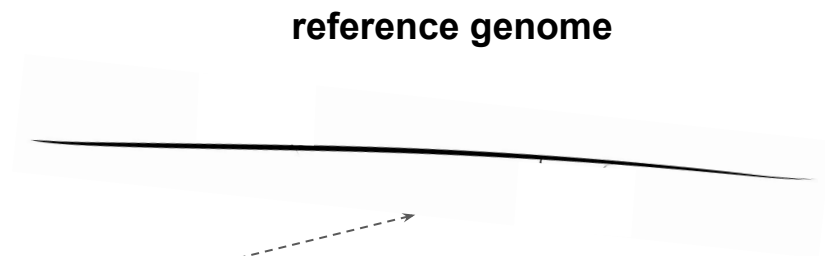
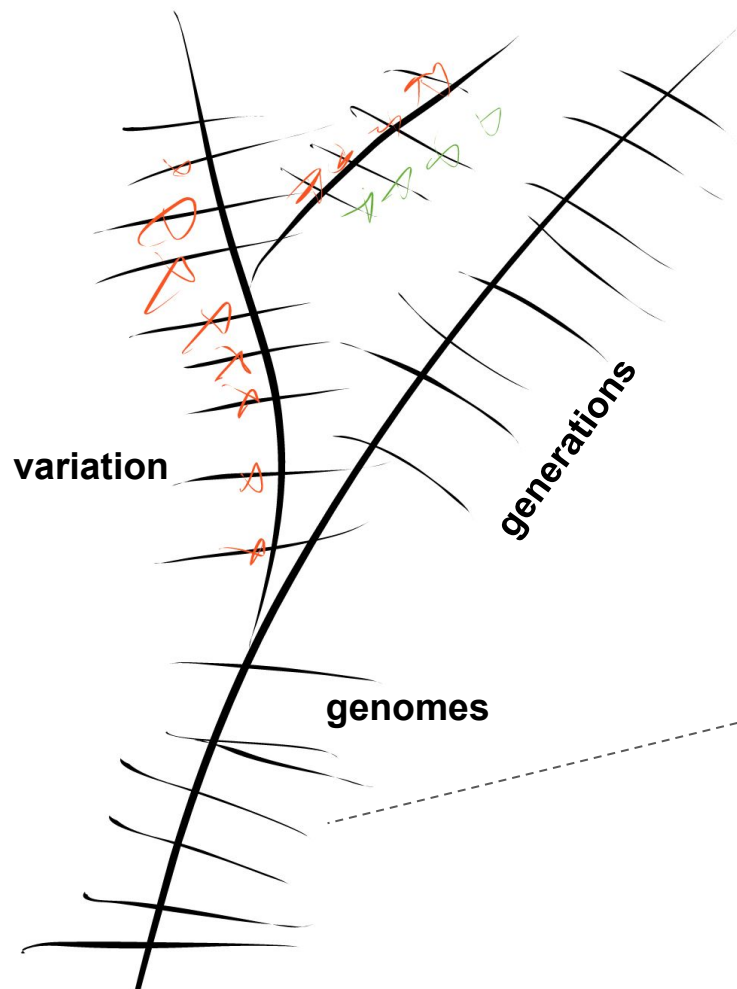
Day 1 (March 6, 2018)

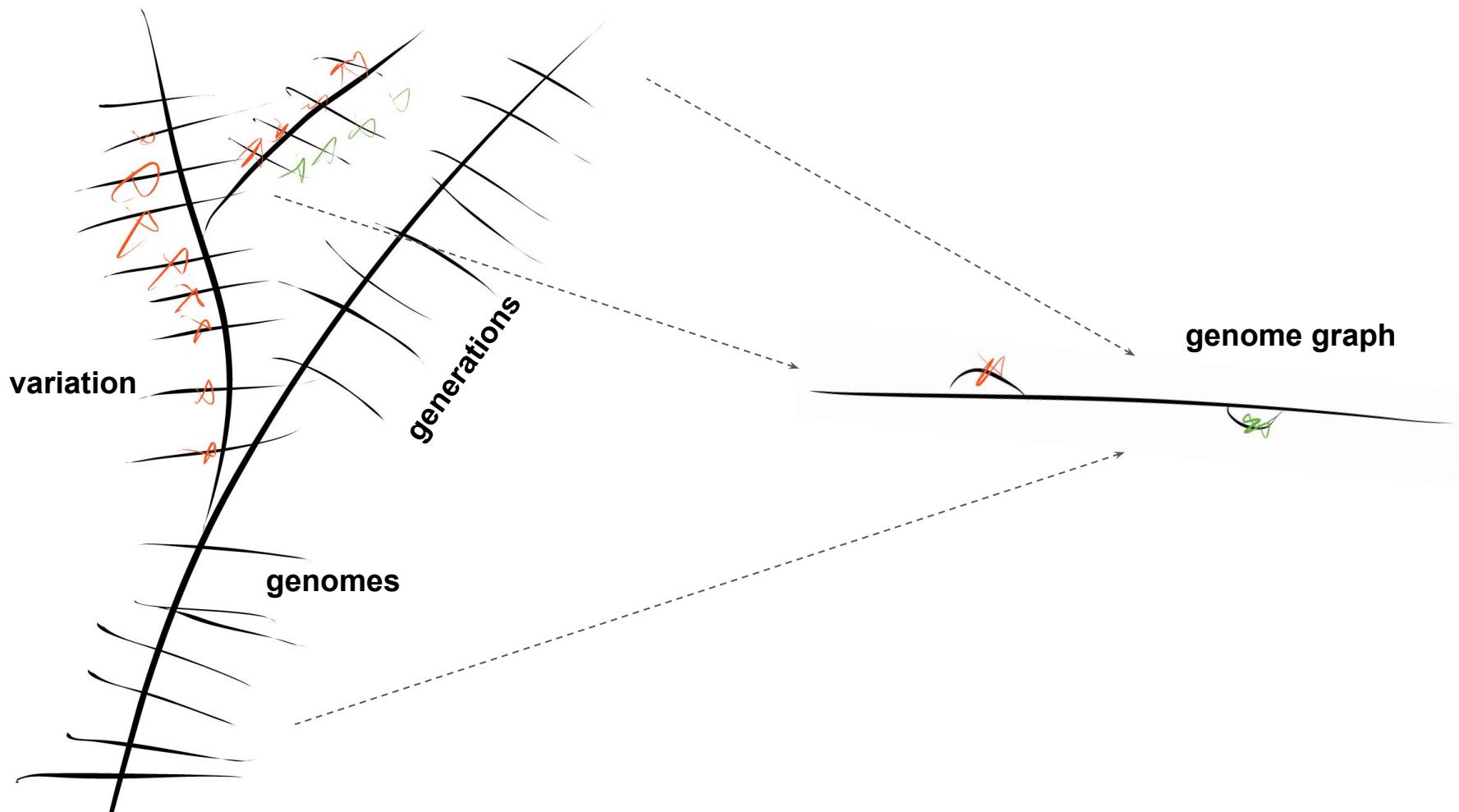
Tobias Marschall, Erik Garrison, and Jordan Eizenga

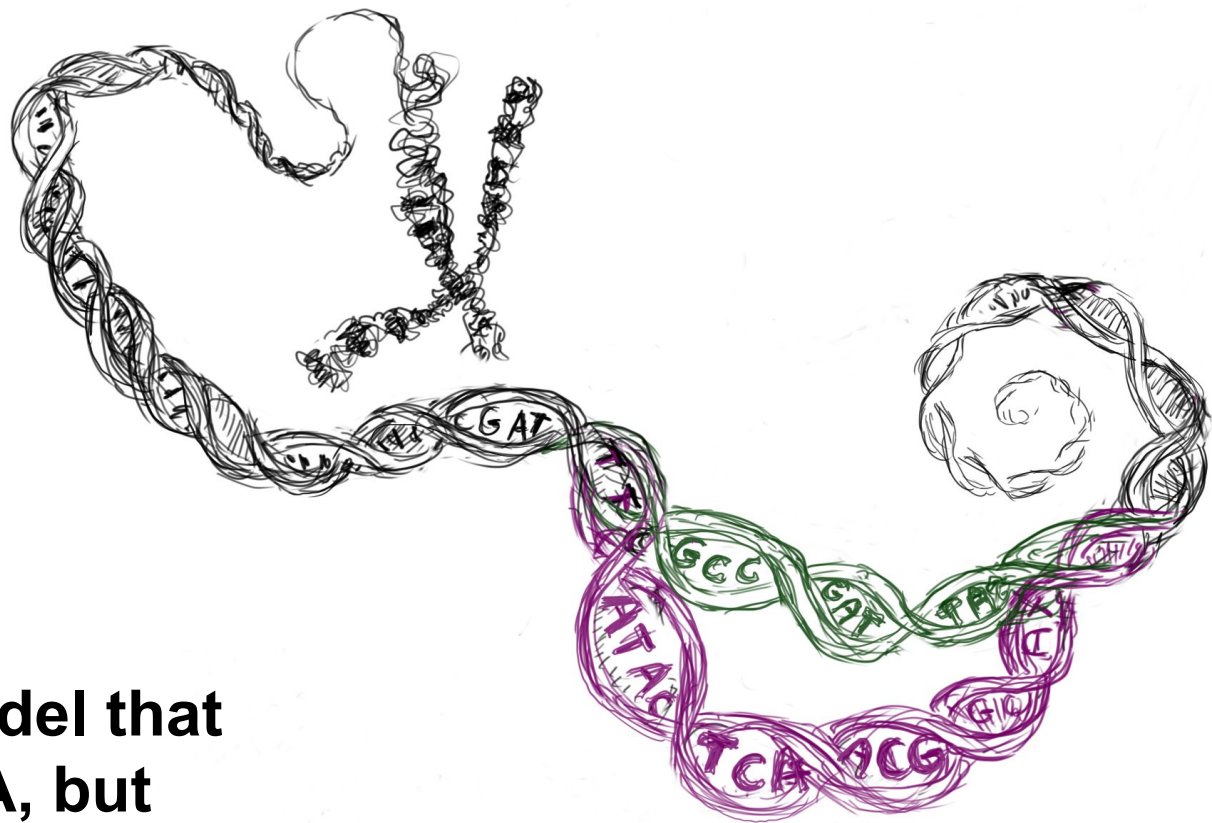
# Genome variation graphs







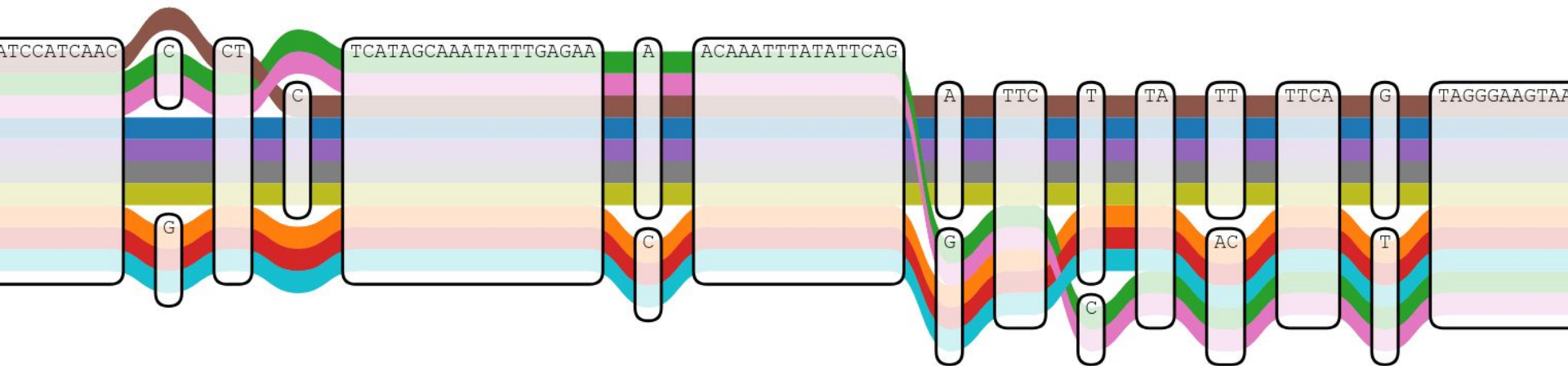




**We want a model that  
looks like DNA, but  
represents many genomes  
at the same time.**







**Variation graph**

<https://vgteam.github.io/sequenceTubeMap/>

# Multiple sequence alignments ~ variation graphs

*traditional MSA*

(a)

.	.	P	K	M	I	V	R	P	Q	K	N	E	T	V	.
T	H	.	K	M	L	V	R	.	.	.	N	E	T	I	M

*consensus sequence*

(b)

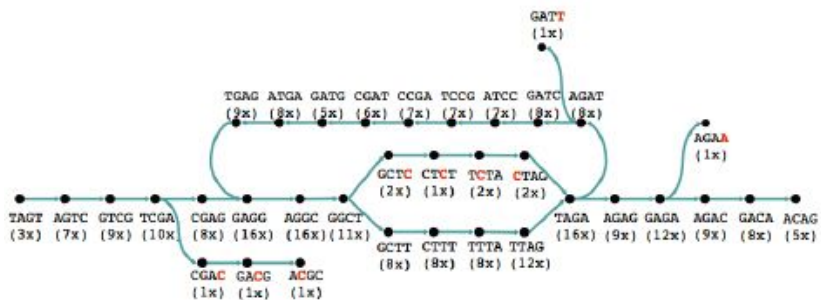
*positionally-matching  
regions aligned*

(c)

*multiple sequence  
alignment*

(d)

# Assembly graphs ~ variation graphs



<http://plus.maths.org/content/os/issue55/features/sequencing/index>, credit Daniel Zerbino

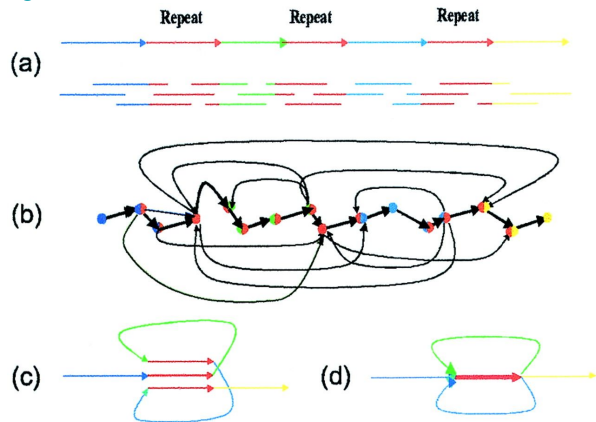
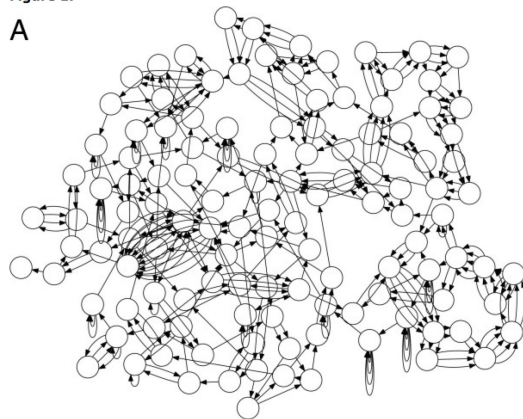


Figure 1.

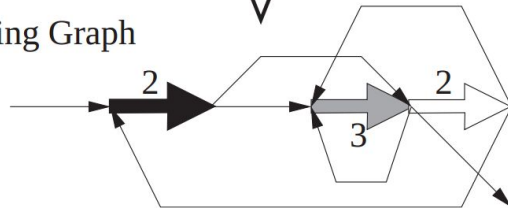
A



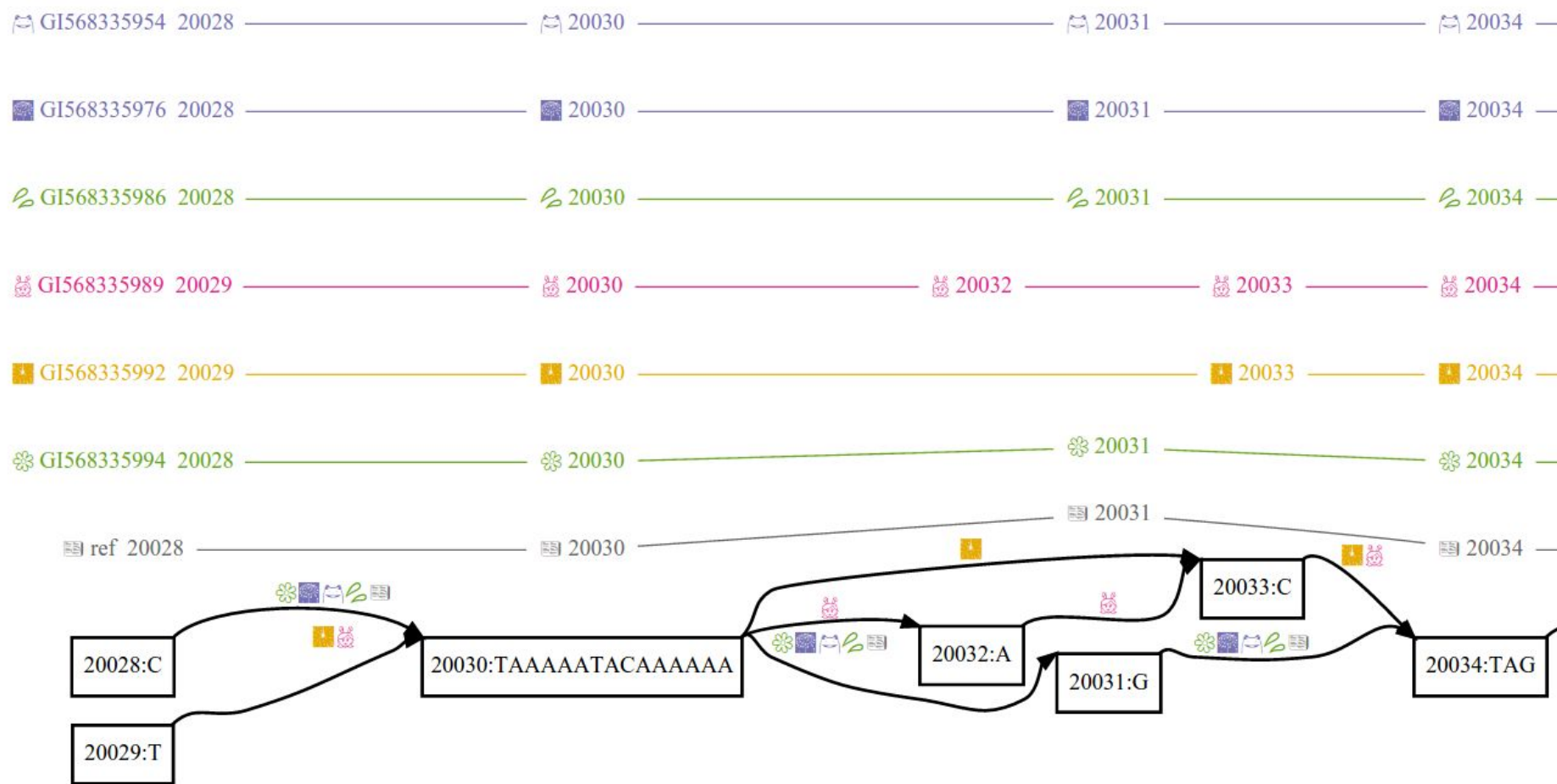
Genome



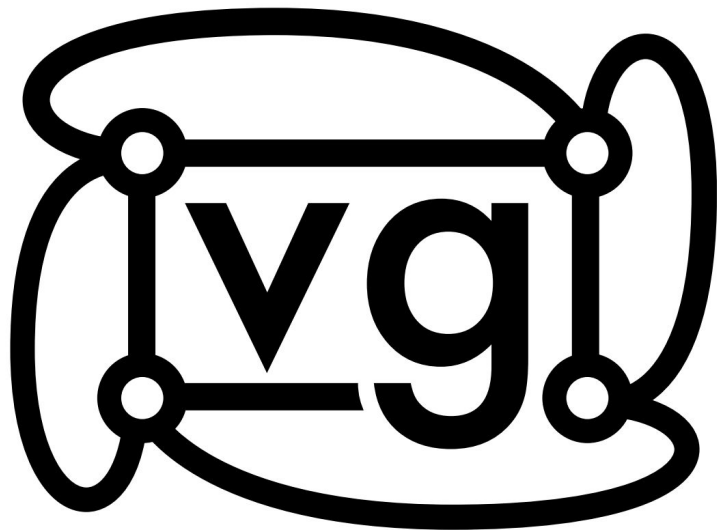
String Graph



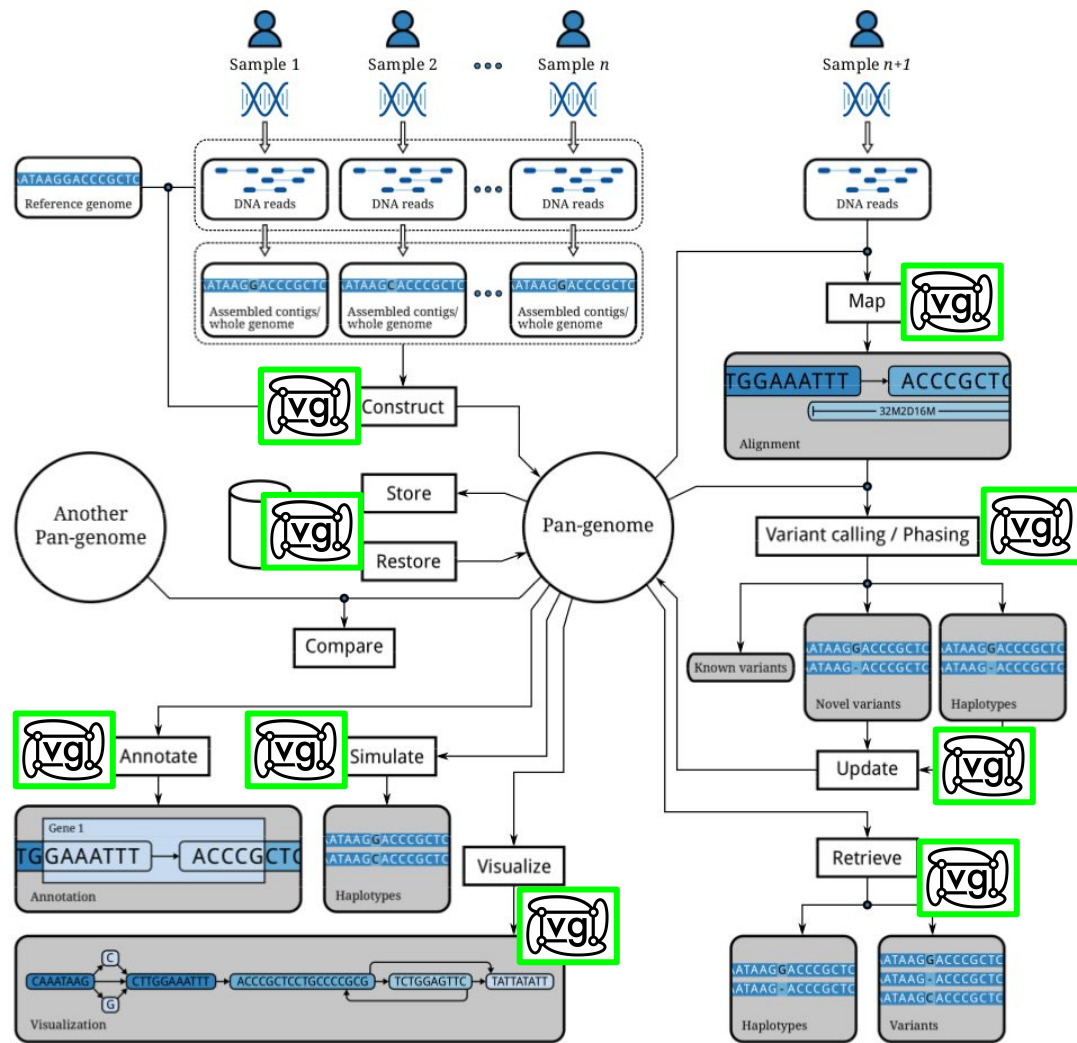




\*a fragment of the MHC



[github.com/vgteam/vg](https://github.com/vgteam/vg)



# Construction (from VCF)

POS	ID	REF	ALT
...			

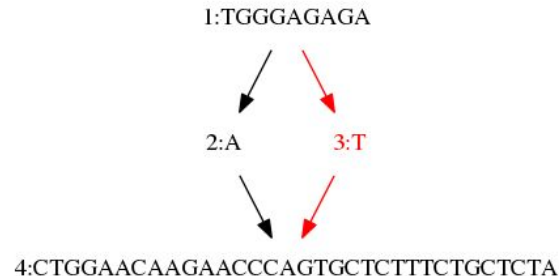
I:TGGGAGAGAACTGGAACAAGAACCCAGTGCTCTTTCTGCTCTA

For each variant

1. cut the reference path  
around the variant
2. add the novel (ALT)  
sequence to the graph

# Construction (from VCF)

POS	ID	REF	ALT
10	.	A	T
...			



For each variant

1. cut the reference path around the variant
2. add the novel (ALT) sequence to the graph

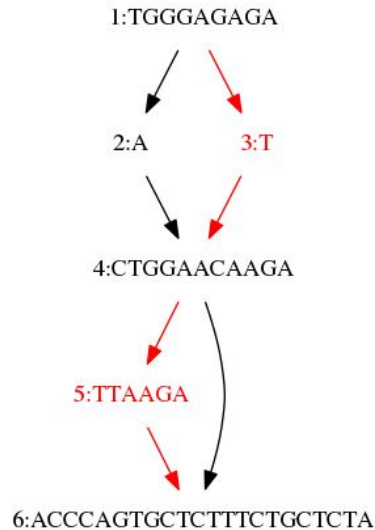


# Construction (from VCF)

POS	ID	REF	ALT
10	.	A	T
21	.	A	ATTAAGA
...			

For each variant

1. cut the reference path around the variant
2. add the novel (ALT) sequence to the graph

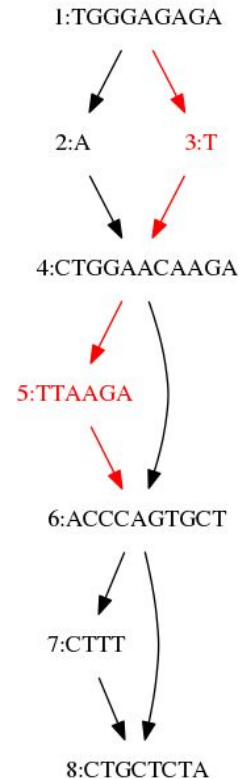


# Construction (from VCF)

For each variant

1. cut the reference path around the variant
2. add the novel (ALT) sequence to the graph

POS	ID	REF	ALT
10	.	A	T
21	.	A	ATTAAGA
31	.		TCTTT

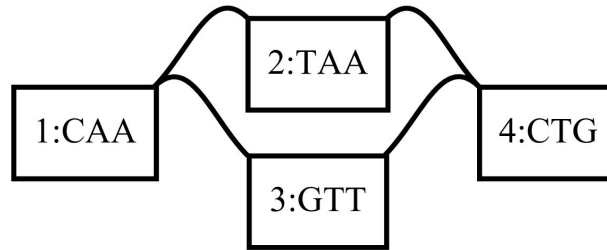


# Local alignment to the graph

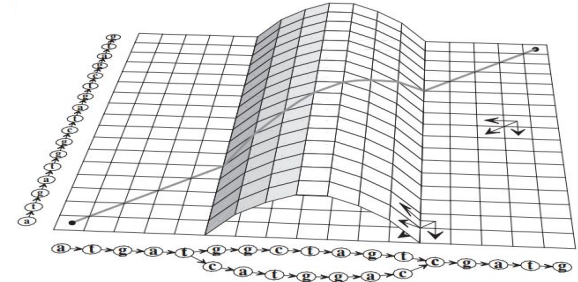
query:  
**CAAATTCT**

	C	A	A
C	<u>2</u>	0	0
A	0	<u>4</u>	2
A	0	2	<u>6</u>
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	<u>4</u>	1	0
T	2	<u>6</u>	3
T	0	4	<u>8</u>
C	0	2	5
T	0	2	4



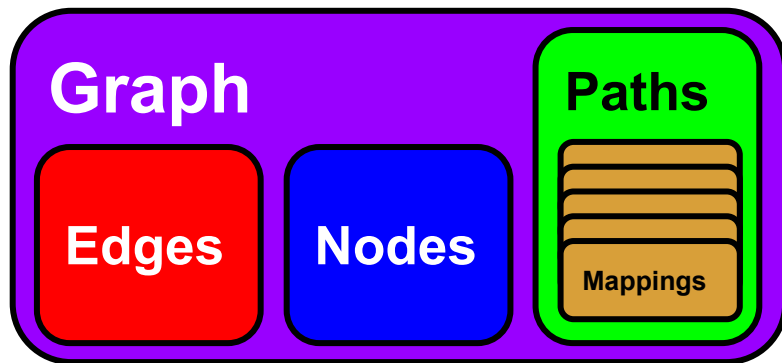
	C	T	G
C	2	0	0
A	0	0	0
A	1	0	0
A	2	0	0
T	2	4	1
T	5	4	3
C	<u>10</u>	7	6
T	7	<u>12</u>	9

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

# Data model

Basic entity is a *Graph*:



*Implemented in vg using protobuf, JSON, RDF, and GFA*

# Graph

```
// *Graphs* are collections of nodes and edges.  
// They can represent subgraphs of larger graphs  
// or be wholly-self-sufficient.  
// Protobuf memory limits of 67108864 bytes mean we typically keep the size  
// of them small generating graphs as collections of smaller subgraphs.  
//  
message Graph {  
    repeated Node node = 1; // The `Node`s that make up the graph.  
    repeated Edge edge = 2; // The `Edge`s that connect the `Node`s in the graph.  
    repeated Path path = 3; // A set of named `Path`s that visit sequences of oriented `Node`s.  
}
```

<https://github.com/ekg/vg/blob/master/src/vg.proto>

# Node

```
// *Nodes* store sequence data.  
message Node {  
    string sequence = 1; // Sequence of DNA bases represented by the Node.  
    string name = 2; // A name provides an identifier.  
    int64 id = 3; // Each Node has a unique positive nonzero ID within its Graph.  
}
```

# Edge

```
// *Edges* describe linkages between nodes. They are bidirected, connecting the
// end (default) or start of the "from" node to the start (default) or end of
// the "to" node.
//
message Edge {
  int64 from = 1; // ID of upstream node.
  int64 to = 2; // ID of downstream node.
  bool from_start = 3; // If the edge leaves from the 5' (start) of a node.
  bool to_end = 4; // If the edge goes to the 3' (end) of a node.
  int32 overlap = 5; // Length of overlap between the connected `Node`s.
}
```

# Path

```
// Paths are walks through nodes defined by a series of `Edit`s.  
// They can be used to represent:  
//   - haplotypes  
//   - mappings of reads, or alignments, by including edits  
//   - relationships between nodes  
//   - annotations from other data sources, such as:  
//       genes, exons, motifs, transcripts, peaks  
//  
message Path {  
    string name = 1; // The name of the path.  
    repeated Mapping mapping = 2; // describe the order and orientation in which the Path visits `Node`s.  
    bool is_circular = 3; // Set to true if the path is circular.  
    int64 length = 4; // Optional length annotation for the Path.  
}
```



# Mapping

```
// A Mapping defines the relationship between a node in system and another entity.  
// An empty edit list implies complete match, however it is preferred to specify the full edit structure.  
// as it is more complex to handle special cases.  
//  
message Mapping {  
    Position position = 1; // The position at which the first Edit, if any, in the Mapping starts. Inclusive.  
    repeated Edit edit = 2; // The series of `Edit`s to transform to region in read/alt.  
    int64 rank = 5; // The 1-based rank of the mapping in its containing path.  
}
```

# Position

```
// A position in the graph is a node, direction, and offset.  
// The node is stored by ID, and the offset is 0-based and  
// counts from the start of the node in the specified orientation.  
// The direction specifies which orientation of the node we are  
// considering, the forward (as stored) or reverse complement.
```

```
message Position {  
    int64 node_id = 1; // The Node on which the Position is.  
    int64 offset = 2; // The offset into that node's sequence at which the Position occurs.  
    bool is_reverse = 4; // True if we obtain the original sequence of the path by reverse complementing  
    string name = 5; // If the position is used to represent a position against a reference path  
}
```

# Position

```
// Example:
//
//      seq+      G A T T A C A
//      offset+   → 0 1 2 3 4 5 6 7
//
//      seq-      C T A A T G T
//      offset-   → 0 1 2 3 4 5 6 7
//
// Or both at once:
//
//      offset-    7 6 5 4 3 2 1 0 ←
//      seq+      G A T T A C A
//      offset+   → 0 1 2 3 4 5 6 7
```

# Edit

```
// Edits describe how to generate a new string from elements
// in the graph. To determine the new string, just walk the series of edits,
// stepping from_length distance in the basis node, and to_length in the
// novel element, replacing from_length in the basis node with the sequence.
//
//
// There are several types of Edit:
// - *matches*: from_length == to_length; sequence is empty
// - *snps*: from_length == to_length; sequence = alt
// - *deletions*: to_length == 0 && from_length > to_length; sequence is empty
// - *insertions*: from_length < to_length; sequence = alt
//
message Edit {
    int32 from_length = 1; // Length in the target/ref sequence that is removed.
    int32 to_length = 2; // Length in read/alt of the sequence it is replaced with.
    string sequence = 3; // The replacement sequence, if different from the original sequence.
}
```

# Alignment

```
// Alignments link query strings, such as other genomes or reads, to Paths.
```

```
//
```

```
message Alignment {
```

```
    string sequence = 1; // The sequence that has been aligned.
```

```
    Path path = 2; // The Path that the sequence follows in the graph it has been aligned to
```

```
    string name = 3; // The name of the sequence that has been aligned. Similar to read name in BAM.
```

```
    bytes quality = 4; // The quality scores for the sequence, as values on a 0-255 scale.
```

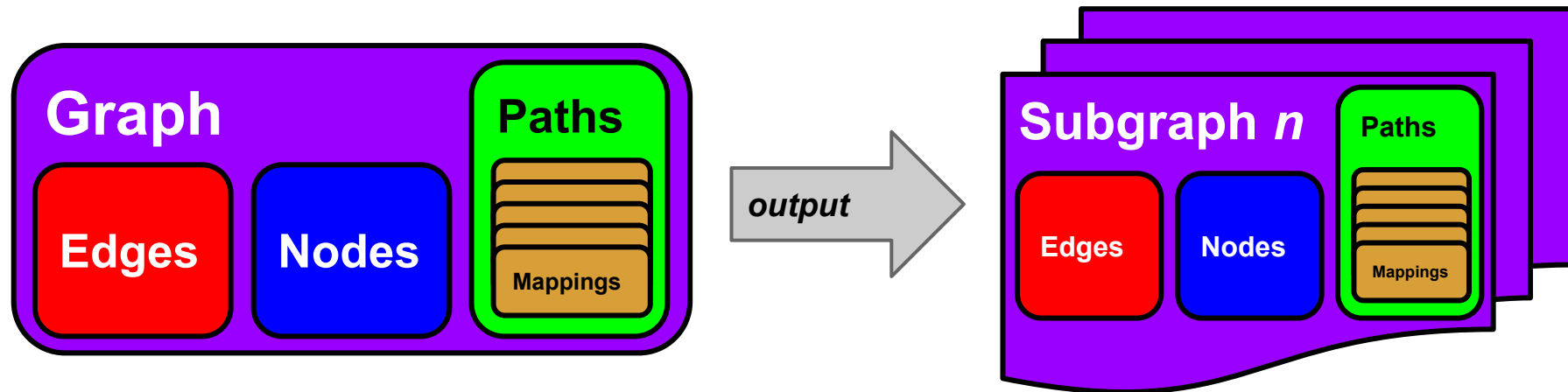
```
    int32 mapping_quality = 5; // The mapping quality score for the alignment, in Phreds.
```

```
    int32 score = 6; // The score for the alignment, in points.
```

```
....
```

# Serialization

To serialize the graph, we generate a stream of sub-graphs that can be reassembled into the whole.



# How to resequence using vg

Import a graph: ***vg construct / vg view***

Index it: ***vg index***

Query it: ***vg find***

Sample it: ***vg sim***

Map to it: ***vg map***

Call variants: ***vg call***

Build a graph: ***vg msga***

# Components of the vg toolchain

- Data model
  - <https://github.com/vgteam/vg/blob/master/src/vg.proto>
- VG C++ API
  - <https://github.com/vgteam/vg/blob/master/src/vg.hpp>
- XG (graph index)
  - <https://github.com/vgteam/vg/blob/master/src/xg.hpp>
- GCSA2 (sequence path index)
  - <https://github.com/jltsiren/gcsa2>



# vg construct

## tiny.fa

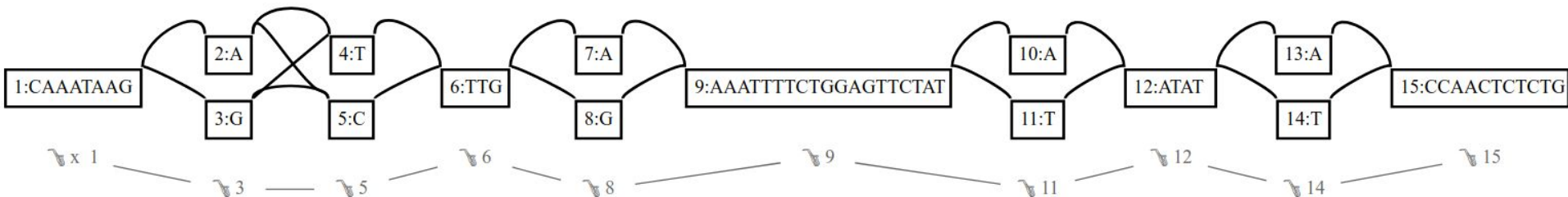
```
1:CAAATAAGGCTTGGAAATTTCTGGAGTTCTATTATATTCCAACCTCTCTG
```

## tiny.vcf.gz

#CHROM	POS	REF	ALT
x	9	G	A
x	10	C	T
x	14	G	A
x	34	T	A
x	39	T	A

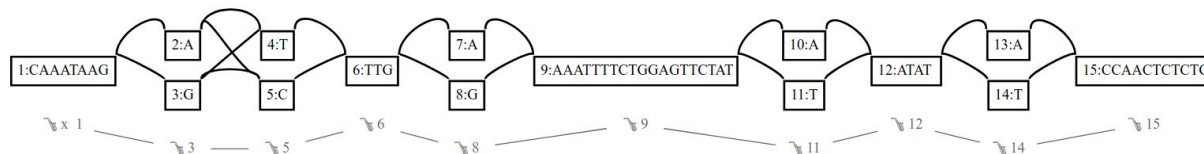
```
vg construct \  
-v tiny/tiny.vcf.gz \  
-r tiny/tiny.fa >tiny.vg
```

## tiny.vg



# vg index

tiny.vg



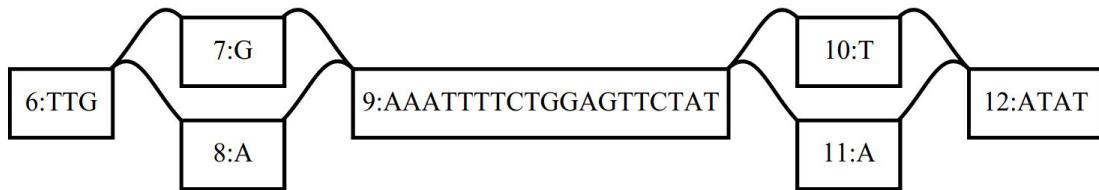
```
vg index tiny.vg \  
-x tiny.xg \  
-g tiny.gcsa -k 16
```

```
vg kmers -gk 16 tiny.vg | head -50  
ATTTGGAAATTTTCTG      2:0      G      G      9:10  
GTTTGGAAATTTTCTG      3:0      G      G      9:10  
CAAATAAGATTTGAAA      1:0      #      A      9:2  
GTTTGGAAATTTTCTG      3:0      G      G      9:10  
ATTTGGAAATTTTCTG      2:0      G      G      9:10  
GCTTGGAAATTTTCTG      3:0      G      G      9:10  
TAAGATTTGAAAATTT      1:4      A      T      9:6  
GCTTGGAAATTTTCTG      3:0      G      G      9:10  
AATAAGATTTGAAAAT      1:2      A      T      9:4  
CCTTATTTG$$$$$      3:-0     A,G     $      17:7  
ACTTGGAAATTTTCTG      2:0      G      G      9:10  
CTTGGAAATTTTCTGG      5:0      A,G     A      9:11  
CAAATAAGATTTGGAA      1:0      #      A      9:2  
CTTGGAAATTTTCTGG      5:0      A,G     A      9:11  
AATAAGATTTGAAA      1:1      C      T      9:3
```

# vg find

```
vg find -x tiny.xg \  
  -p x:20-25 -c 1 \  
  | vg view -d -
```

Query the nodes around x:20-25  
in the reference path “x”.



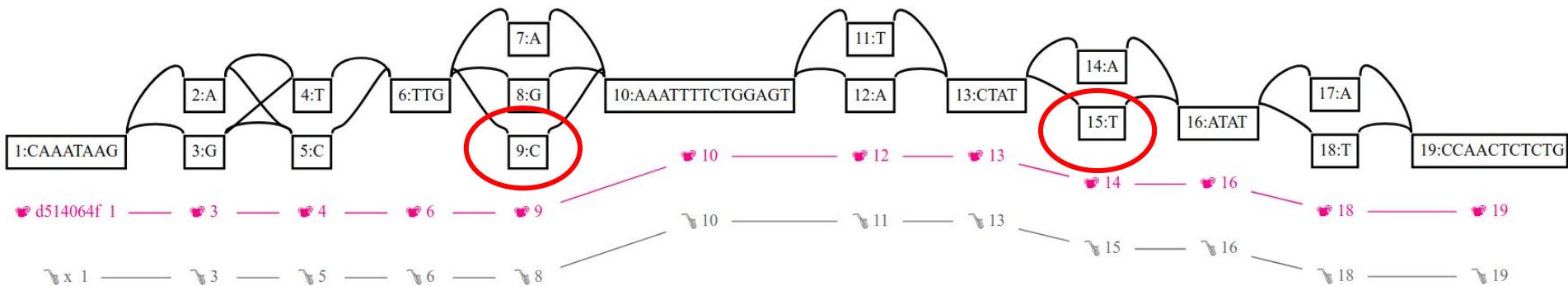
```
vg find -g tiny.gcsa \  
  -S TCCAGAAAATTTCAA  
→ 9:-7
```

Query the position of a particular  
sequence in the GCSA2 index.

# vg sim

Use a haplotype representing some variants relative to the tiny.vg to build a new graph:

```
vg msga -g tiny.vg -Nz \  
-s CAAATAAGGTTTGCAAATTTCTGGAGTACTATAATATTCCAACCTCTCTG \  
>truth.vg
```



We can then use it as a generative model and sample reads from it:

```
vg sim -l 50 -n 10 -s 1337 -x truth.xg >truth.reads
```

# vg map

```
vg map -x tiny.xg -g tiny.gcsa -T truth.reads >aln.gam
```

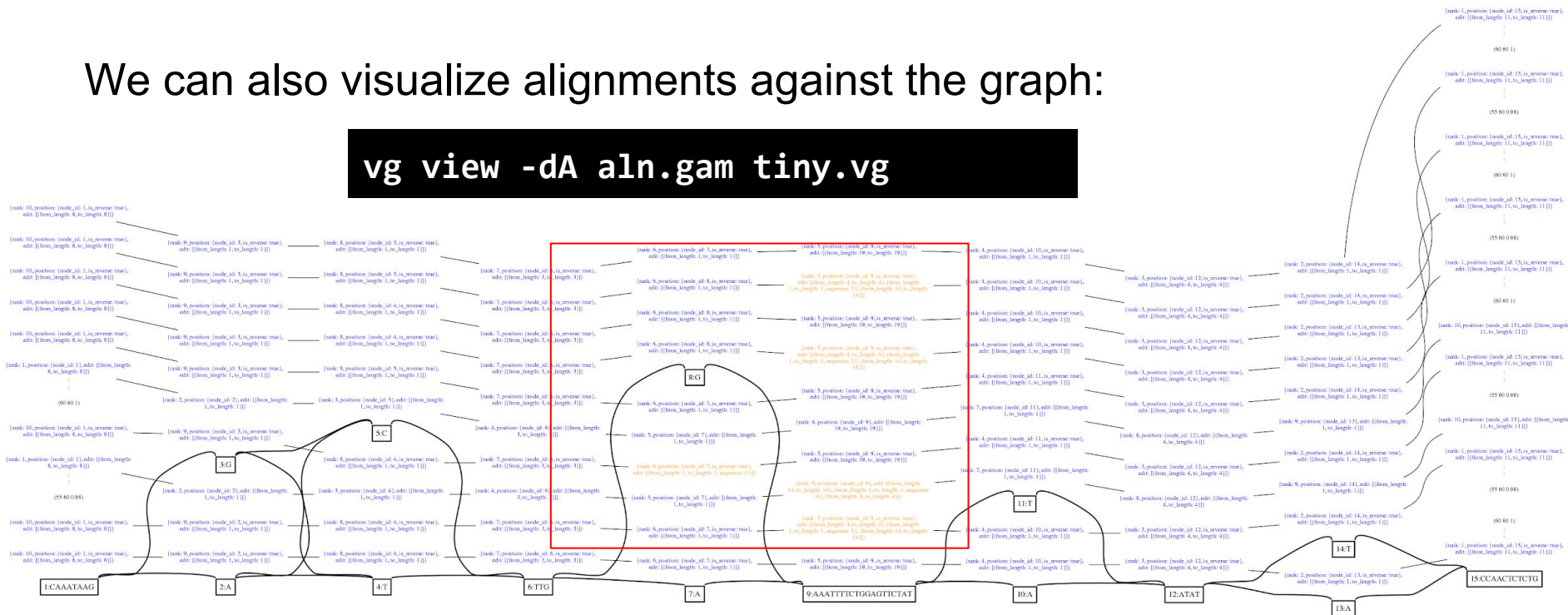
```
{
  "sequence": "CAGAGAGTTGGTATATTATAGAACTCCAGAAAATTTCCAAACCTTATTTG",
  "identity": 1,
  "path": {
    "mapping": [
      {
        "position": {
          "node_id": 15,
          "is_reverse": true
        },
        "edit": [
          {
            "from_length": 11,
            "to_length": 11
          }
        ],
        "rank": 1
      },
      {
        "position": {
          "node_id": 13,
          "is_reverse": true
        },
        "edit": [
          {
            "from_length": 1,
            "to_length": 1
          }
        ]
      }
    ]
  }
}
```

```
vg view -a aln.gam
```

# alignment viz

We can also visualize alignments against the graph:

```
vg view -dA aln.gam tiny.vg
```



Blue represents perfect match.  
Yellow represents a mismatch.



# Day 2



# Introduction to viral quasispecies

# Subcommands

Introduce subcommands: surject, vectorize, msga, prune

Interleave: explain subcommands and have participants try them on toy examples from previous day

Practical: five virus mix

**Day 3**

# Introduction to drug resistance in bacteria

# Practical 1: Build a single-gene graph

- build a gene-model for *gyrA* and map reads to it for a handful of samples.
- Infer the sequence of this gene in each sample (vg call/mod)

## Practical 2: Gene presence via assembly

- Assembly (minia3) of each sample, thread mcr-1/2/3 gene sequences through it to determine presence/absence of this colistin-resistance gene

**Day 4**





# vg overview and practical

One slide per interesting subcommand, describing general idea:

**Construct, view, index, find, sim, map, mpmmap, surject, msga, mod, prune, call, augment, vectorize, pack, chunk, deconstruct, snarls, explode, concat, simplify, translate**

One slide showing an example use case.

Day 1: construct, view, index, find, sim, map

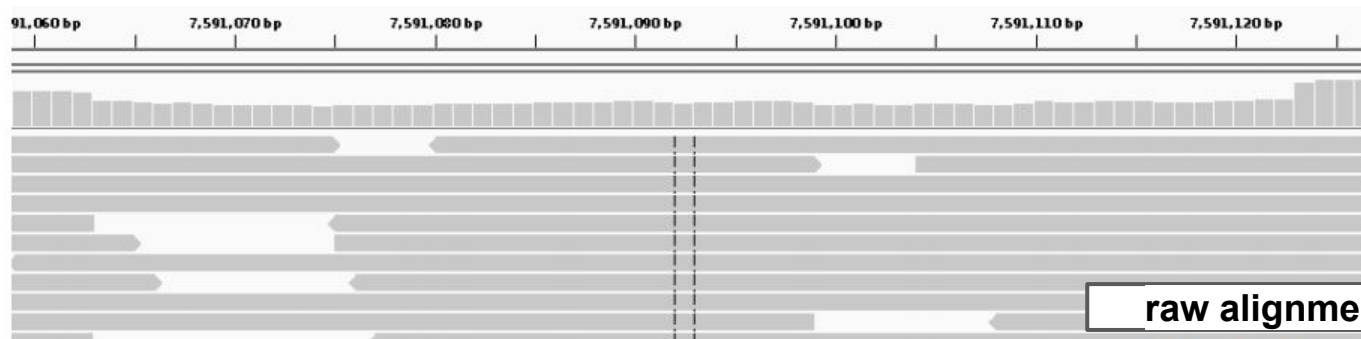
Day 2: surject, vectorize, msga, prune

Day 3: call, mod, pack, augment, chunk, explode

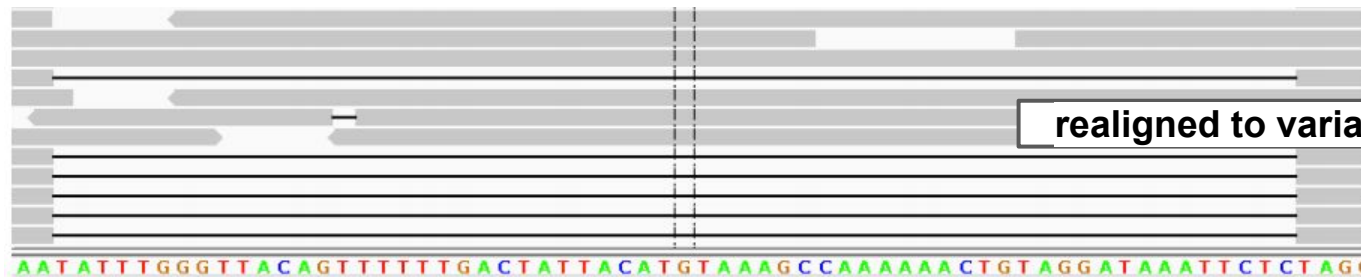
Day 4: snarls, mpmmap, simplify, translate



# When does a linear reference fail?

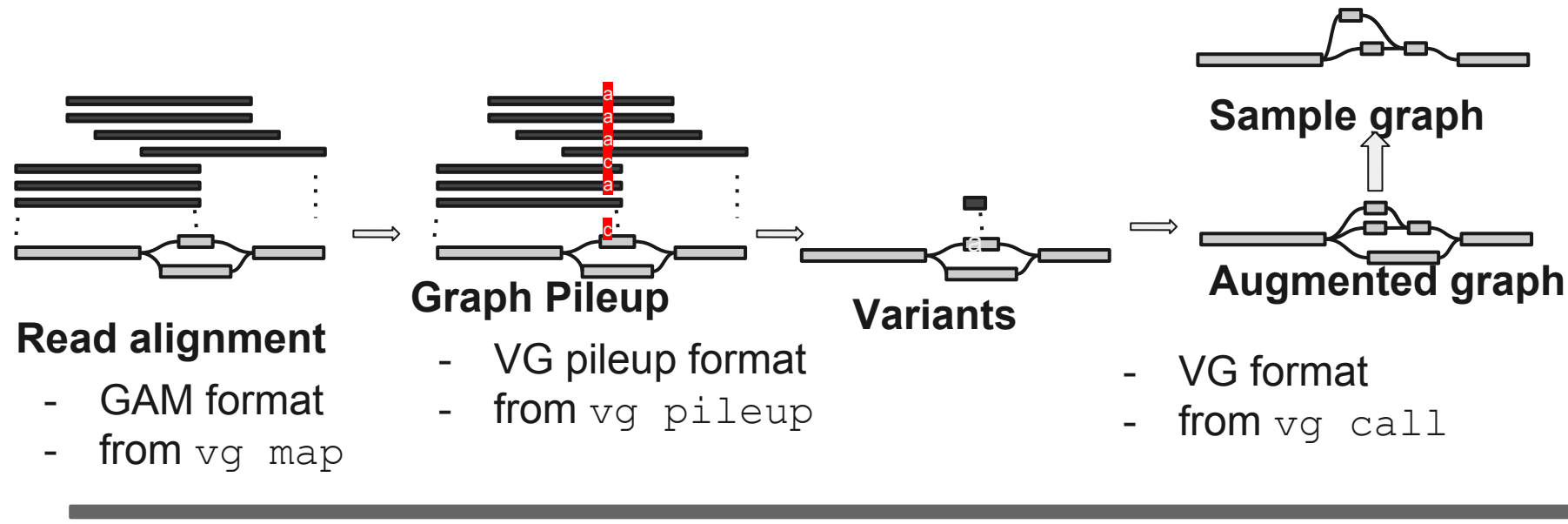


***realign to:***



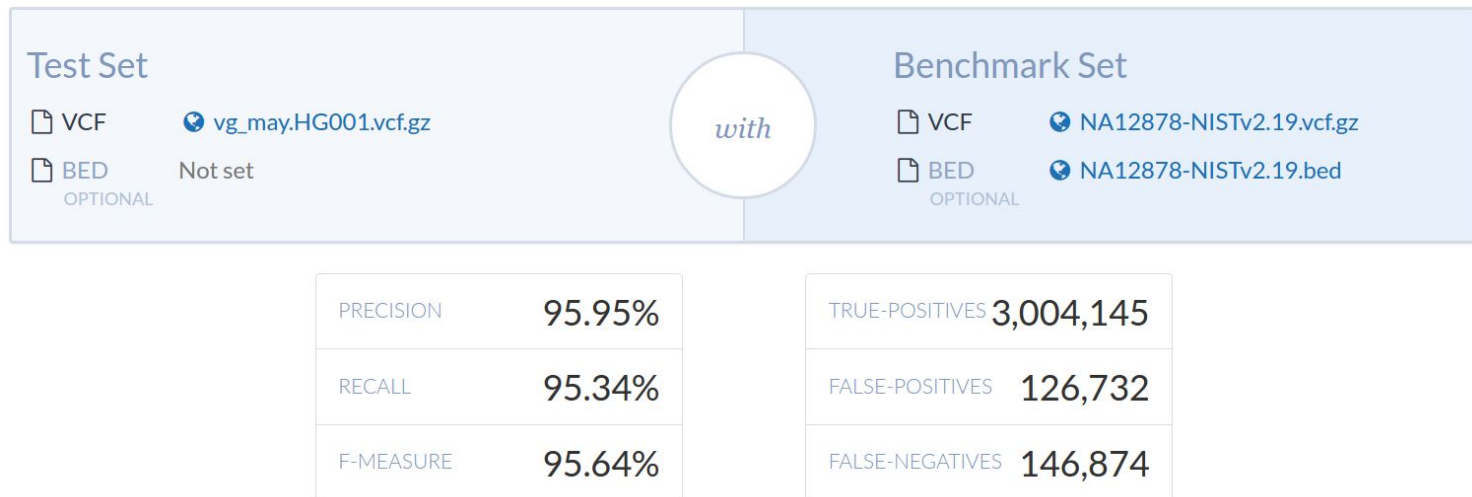


# Variant calling on the graph



Glenn Hickey, Adam Novak, Benedict Paten, Mike Lin

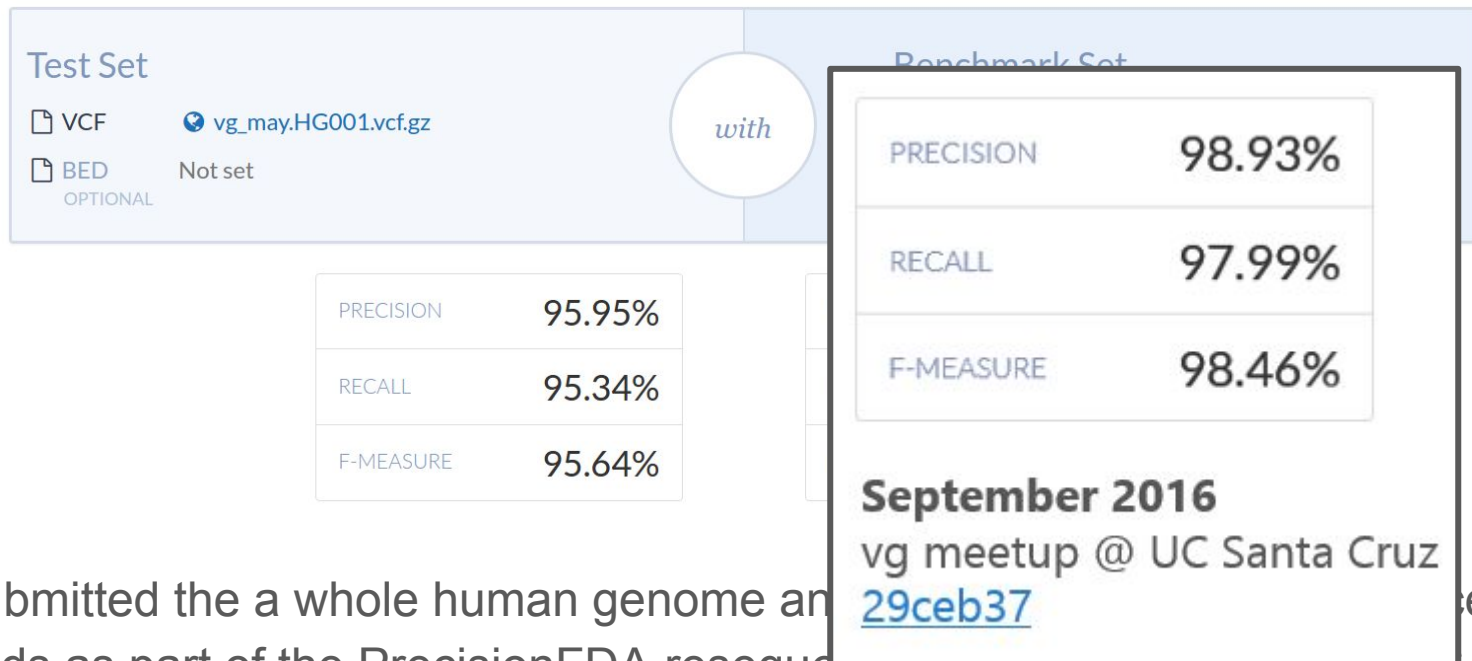
# Whole human genome analysis pipeline



We submitted the a whole human genome analysis using variation reference methods as part of the PrecisionFDA resequencing competition. (May 2016.)

We did not win... but we did get a star for: ★ **HEROIC-EFFORT**

# Whole human genome analysis pipeline



We submitted the a whole human genome and used these methods as part of the PrecisionFDA resequencing competition. (may 2016.)

We did not win... but we did get a star for: ★ **HEROIC-EFFORT**



# A community evaluation of reference graphs (in MHC)

