Lecture 11: de Bruijn Graphs and Metagenomics

Bioinformatics Algorithms CSC4181/6802

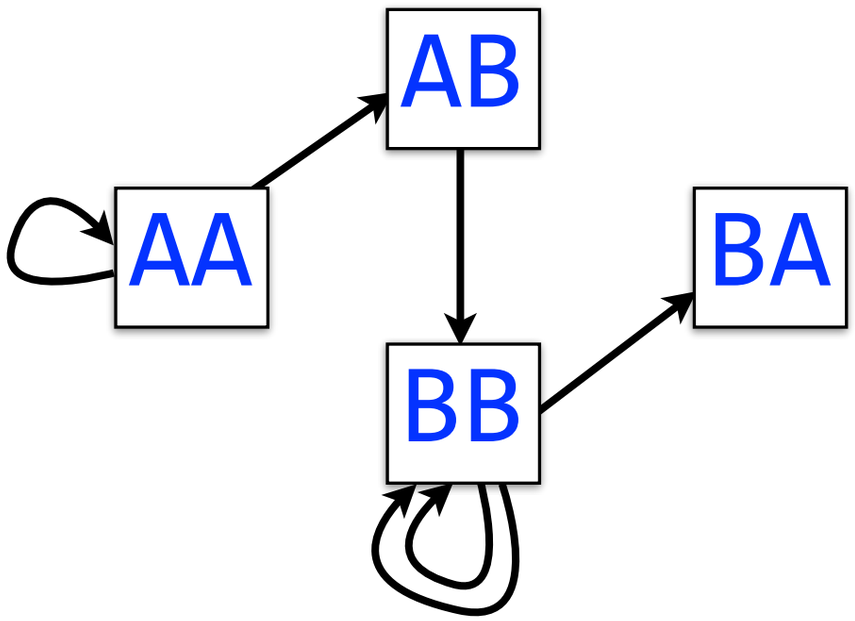
Most slides used are from Ben Langmead's Teaching Materials (www.langmead-lab.org/teaching-materials)

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

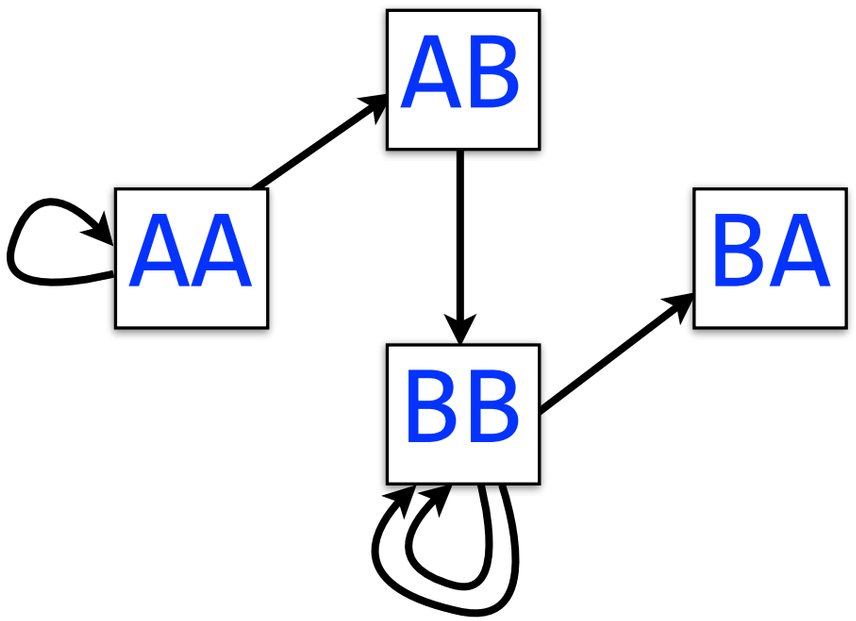
L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB BB, BA



One edge per *k*-mer

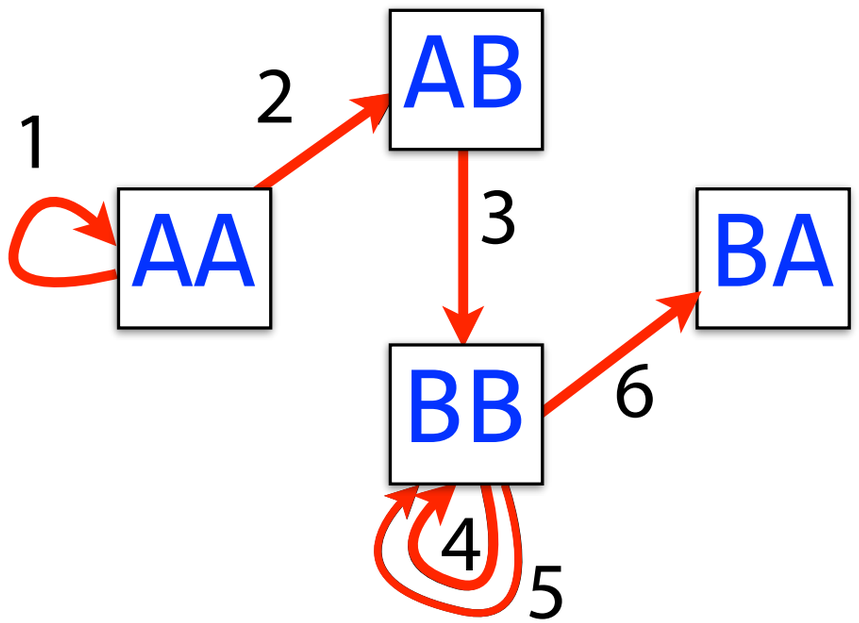
One node per distinct *k*-1-mer

De Bruijn graph



Walk crossing each edge exactly once gives a reconstruction of the genome

De Bruijn graph



AAABBBBA

Walk crossing each edge exactly once gives a reconstruction of the genome. This is an *Eulerian walk*.

Directed multigraph

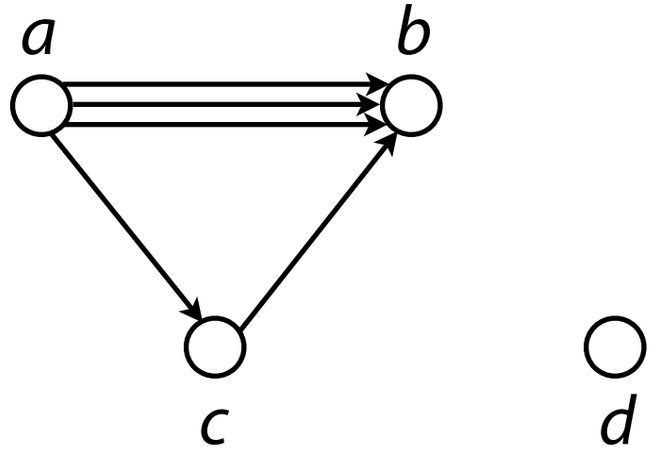
Directed **multigraph** $G(V, E)$ consists of set of *vertices, V* and **multiset** of *directed edges, E*

Otherwise, like a directed graph

Node's *indegree* = # incoming edges

Node's *outdegree* = # outgoing edges

De Bruijn graph is a directed multigraph



$V = \{ a, b, c, d \}$

$E = \{ (a, b), (a, b), (a, b), (a, c), (c, b) \}$

Repeated

Eulerian walk definitions and statements

Node is *balanced* if indegree equals outdegree

Node is *semi-balanced* if indegree differs from outdegree by 1

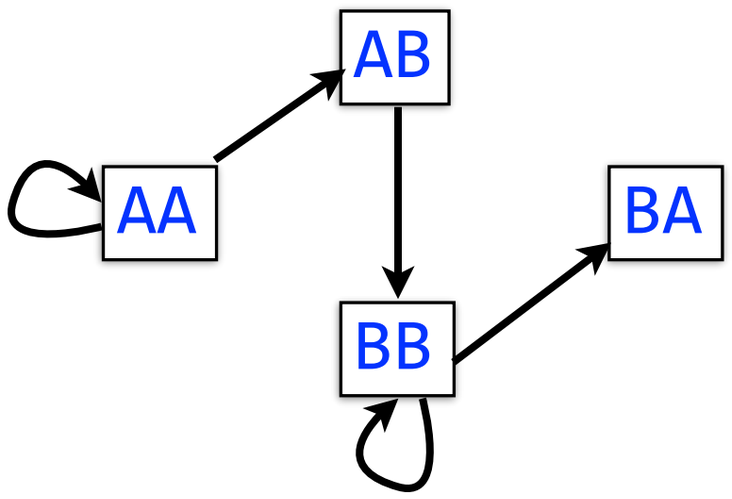
Graph is *connected* if each node can be reached by some other node

Eulerian walk visits each edge exactly once

Not all graphs have Eulerian walks. Graphs that do are *Eulerian*.
(For simplicity, we won't distinguish Eulerian from semi-Eulerian.)

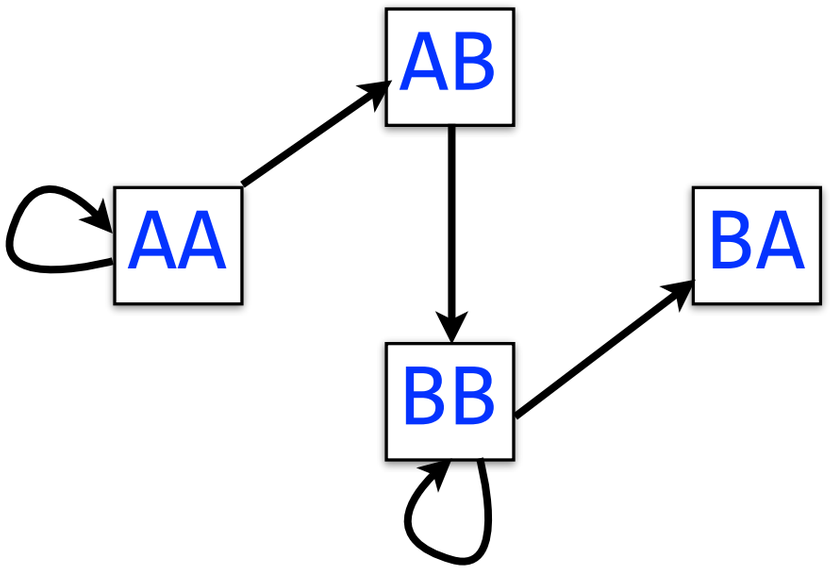
A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced

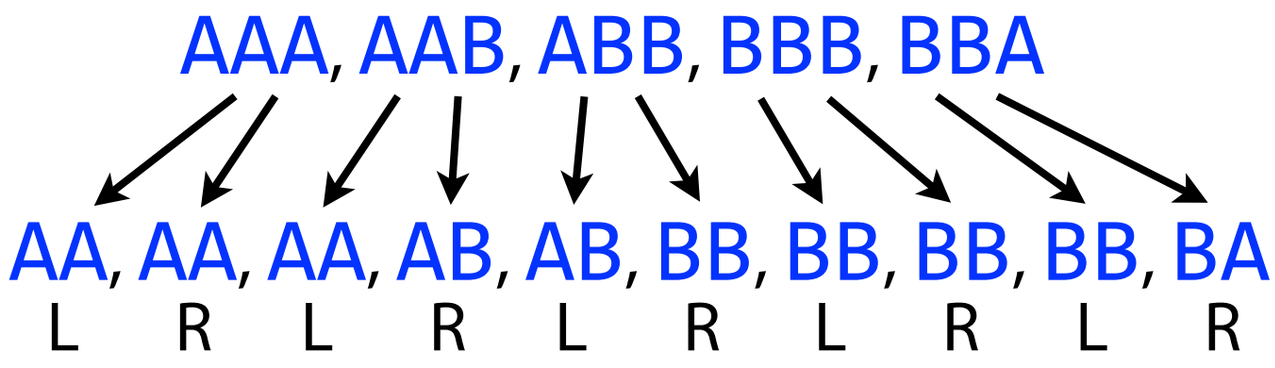
Jones and Pevzner section 8.8



De Bruijn graph

Back to de Bruijn graph





Is it Eulerian? Yes

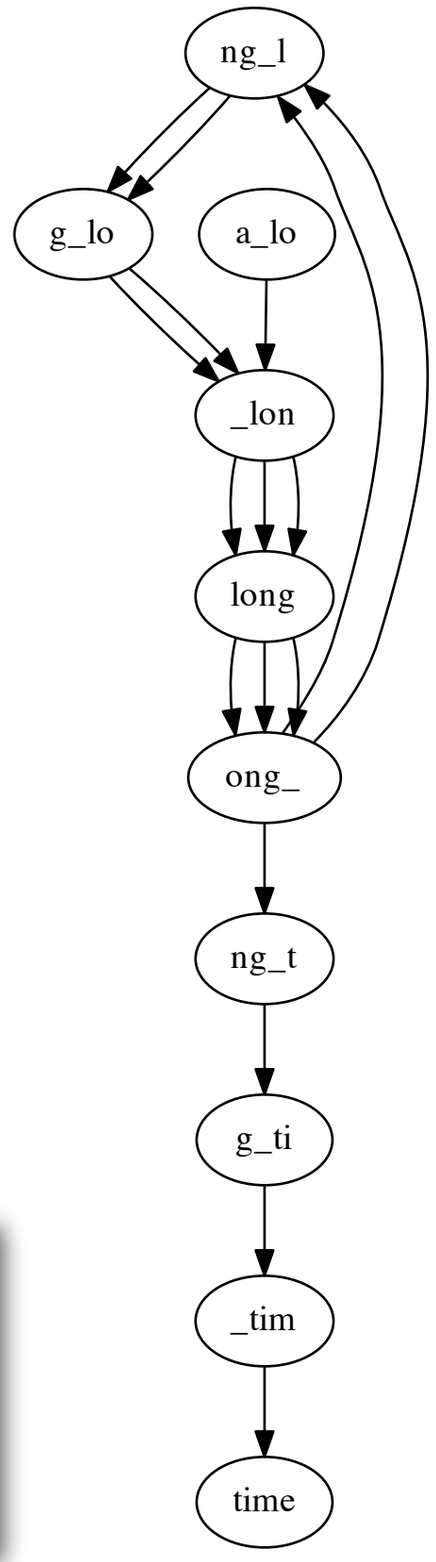
Argument 1: AA → AA → AB → BB → BB → BA

Argument 2: AA and BA are semi-balanced, AB and BB are balanced

De Bruijn graph

Full illustrative de Bruijn graph and Eulerian walk implementation:

http://bit.ly/CG_DeBruijn



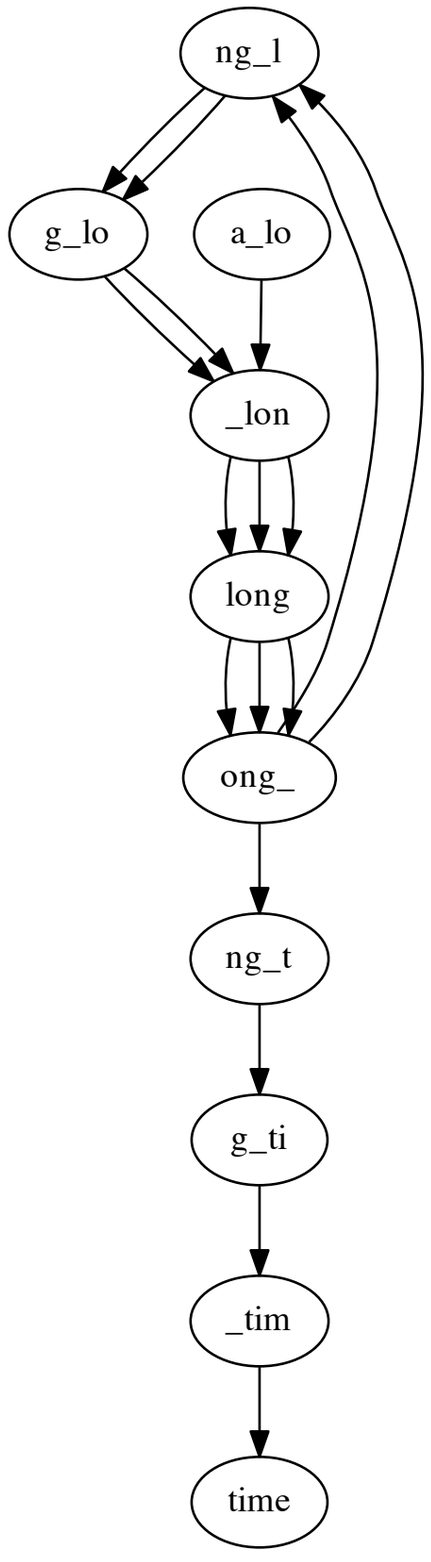
Example where Eulerian walk gives correct answer for small *k* whereas Greedy-SCS could spuriously collapse repeat:

```
>>> G = DeBruijnGraph(["a_long_long_long_time"], 5)
>>> print G.eulerianWalkOrCycle()
['a_lo', '_lon', 'long', 'ong_', 'ng_l', 'g_lo',
'_lon', 'long', 'ong_', 'ng_l', 'g_lo', '_lon',
'long', 'ong_', 'ng_t', 'g_ti', '_tim', 'time']
```

De Bruijn graph

Assuming perfect sequencing, procedure yields graph with Eulerian walk that can be found efficiently.

We saw cases where Eulerian walk corresponds to the original superstring. Is this always the case?

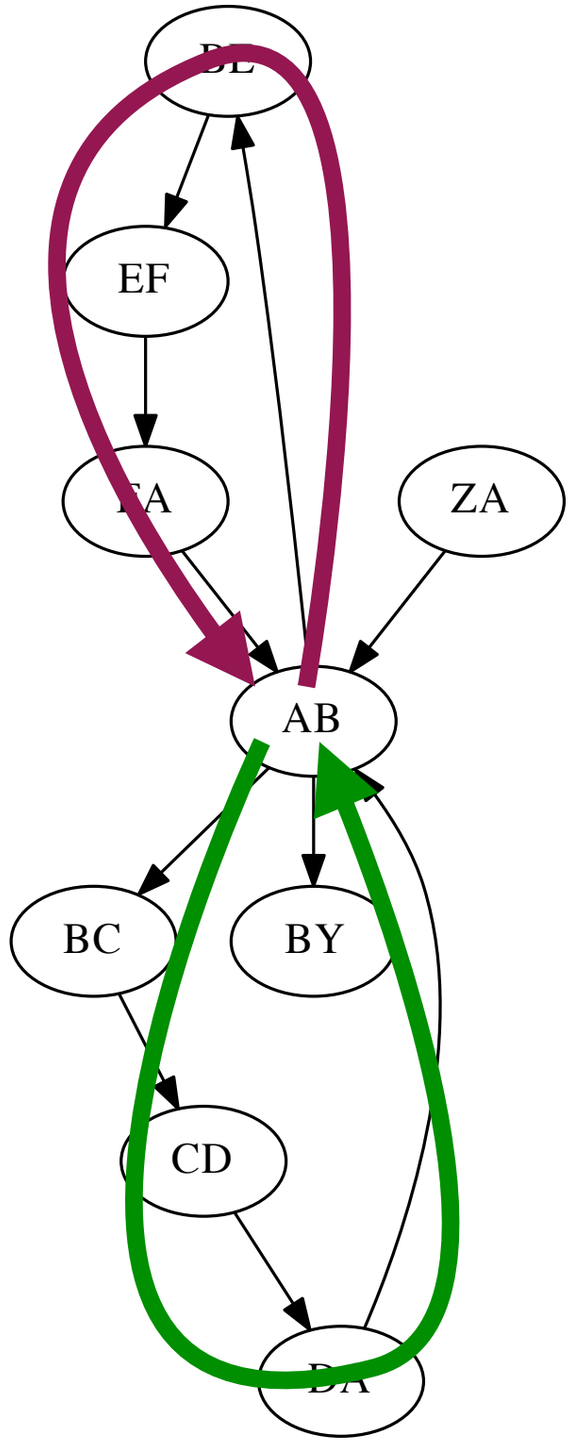


De Bruijn graph

Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY



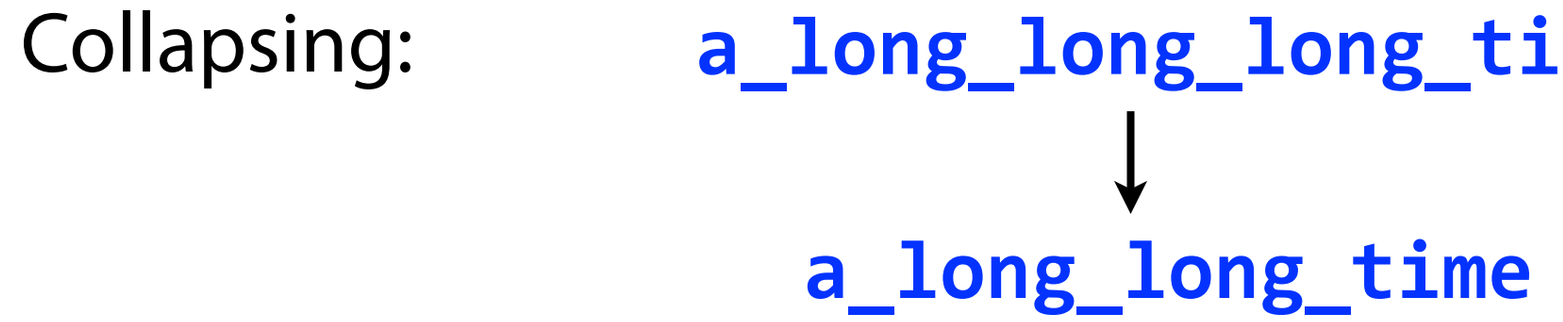
Problem 2:

We've been building DBGs assuming "perfect" sequencing: each *k*-mer reported exactly once, no mistakes. Real datasets aren't like that.

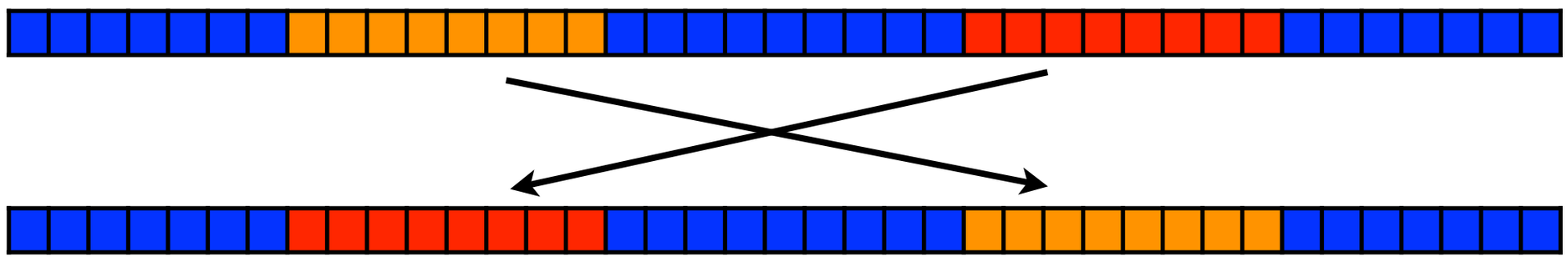
Third law of assembly

Repeats make assembly difficult; whether we can assemble without mistakes depends on length of reads and repetitive patterns in genome

Collapsing:



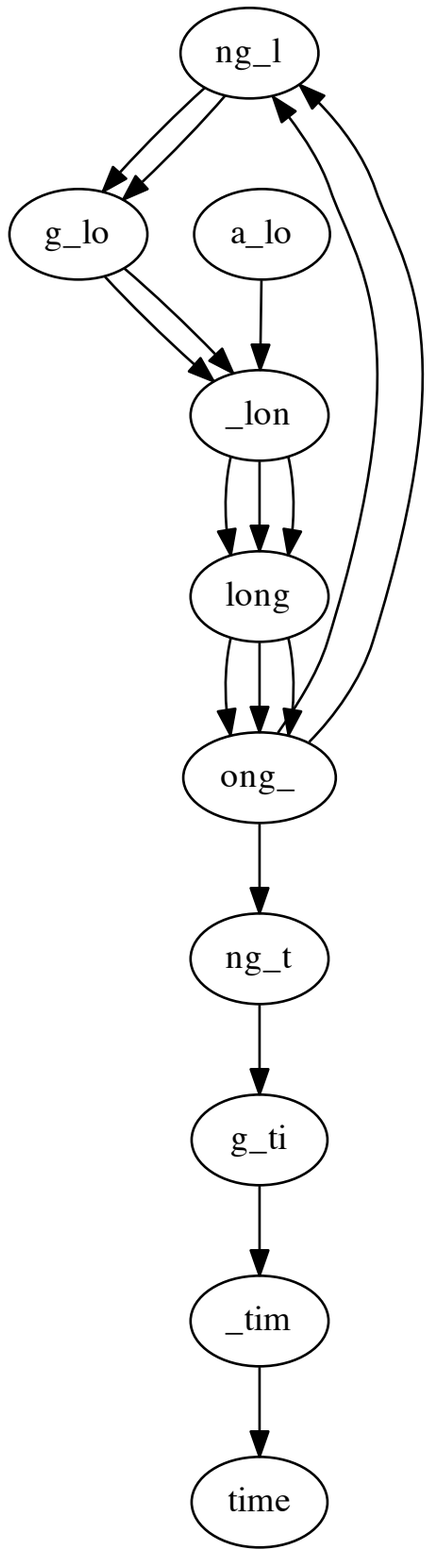
Shuffling:



De Bruijn graph

Gaps in coverage (missing *k*-mers) lead to *disconnected* or non-Eulerian graph

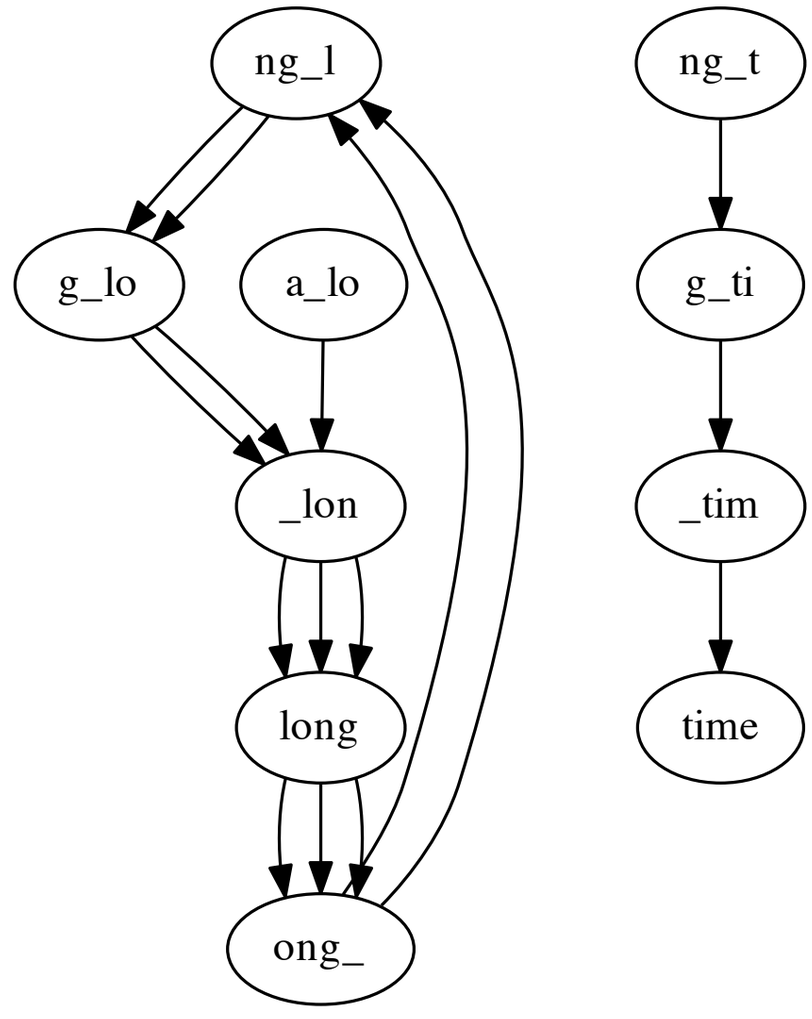
Graph for `a_long_long_long_time`, $k = 5$:



De Bruijn graph

Gaps in coverage (missing *k*-mers) lead
to *disconnected* or non-Eulerian graph

Graph for `a_long_long_long_time`, $k = 5$ but *omitting* `ong_t` :

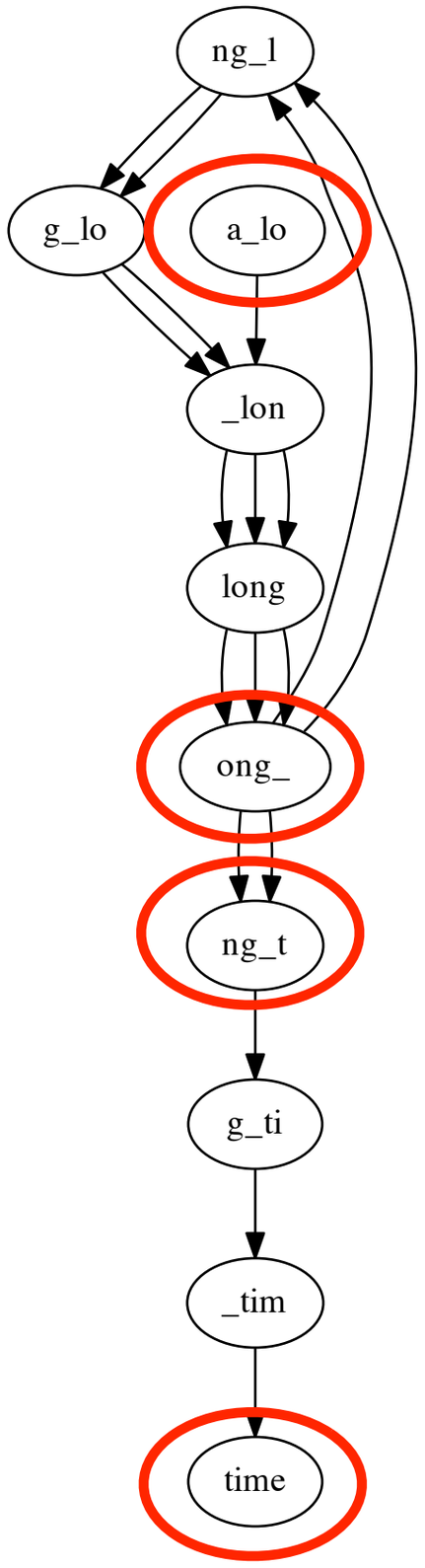


De Bruijn graph

Coverage *differences* make graph non-Eulerian

Graph for `a_long_long_long_time`,
$k = 5$, with *extra copy* of `ong_t` :

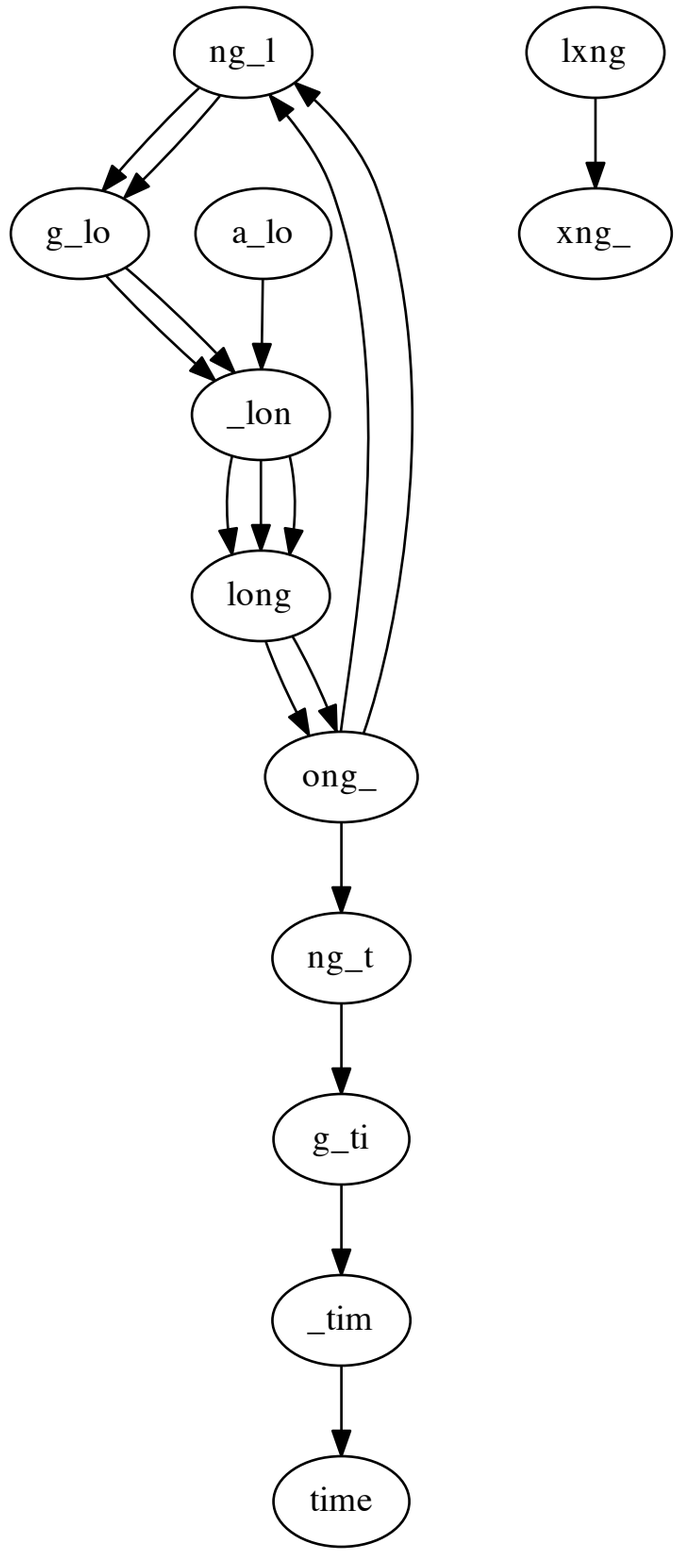
4 semi-balanced nodes



De Bruijn graph

Errors and differences between chromosomes also lead to non-Eulerian graphs

Graph for `a_long_long_long_time`, $k = 5$ but with error that turns one copy of `long_` into `lxng_`



De Bruijn graph

Casting assembly as Eulerian walk is appealing, but not practical

Uneven coverage, sequencing errors, etc make graph non-Eulerian

Even if graph were Eulerian, repeats yield many possible walks

Kingsford, Carl, Michael C. Schatz, and Mihai Pop. "Assembly complexity of prokaryotic genomes using short reads." *BMC bioinformatics* 11.1 (2010): 21.

De Bruijn Superwalk Problem (DBSP) seeks a walk over the De Bruijn graph, where walk contains each read as a *subwalk*

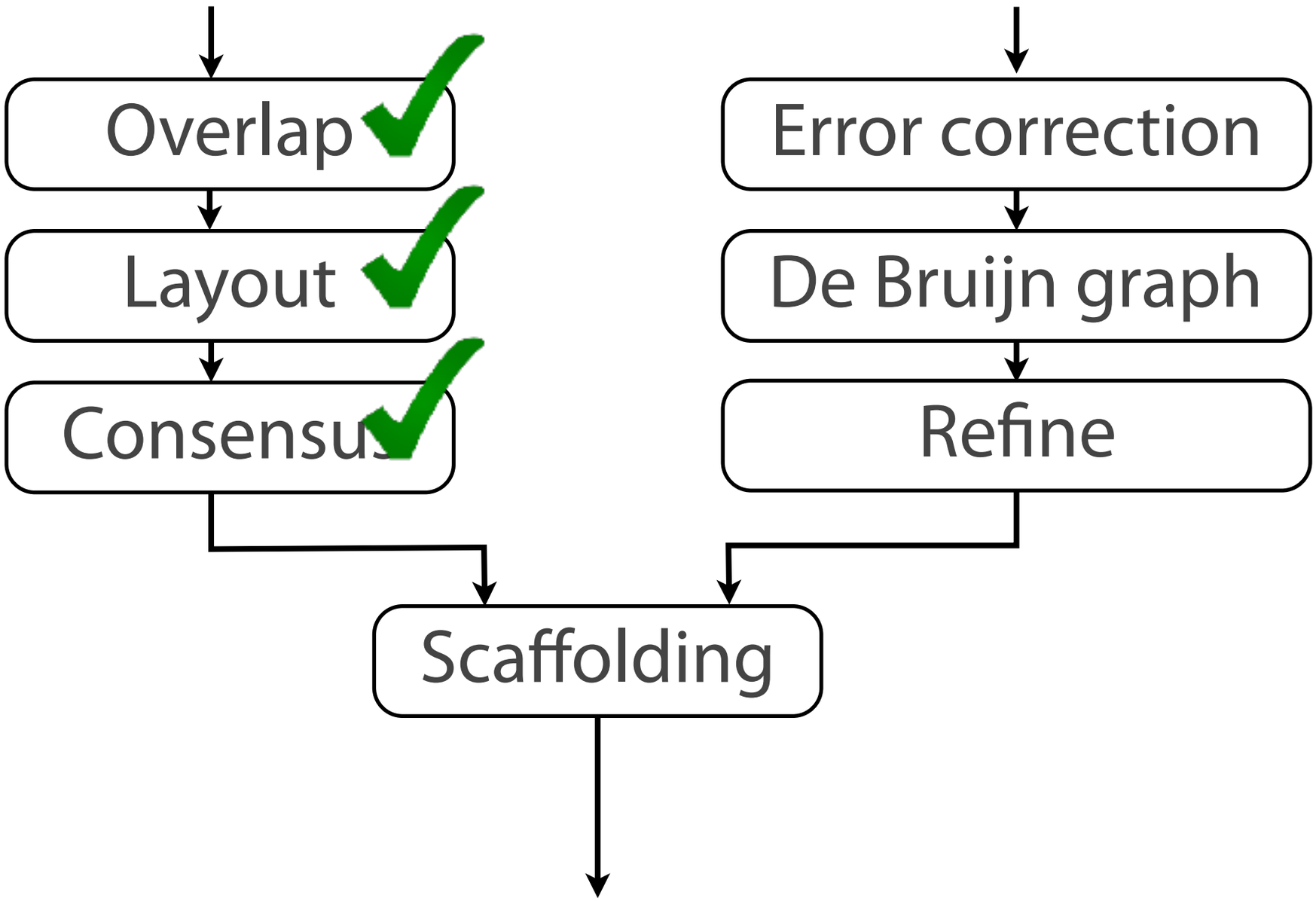
Proven NP-hard!

Medvedev, Paul, et al. "Computability of models for sequence assembly." *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, 2007. 289-301.

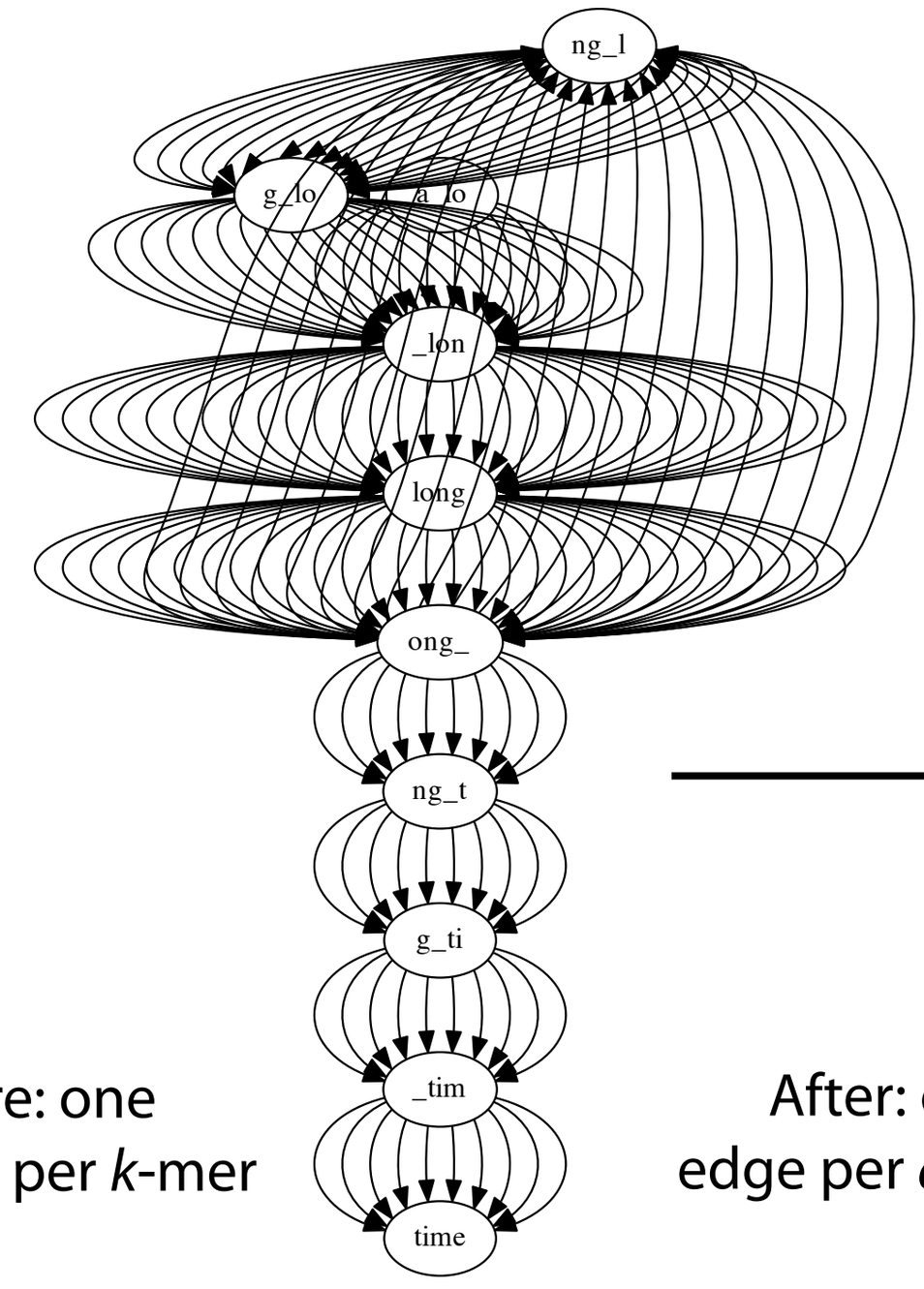
Assembly alternatives

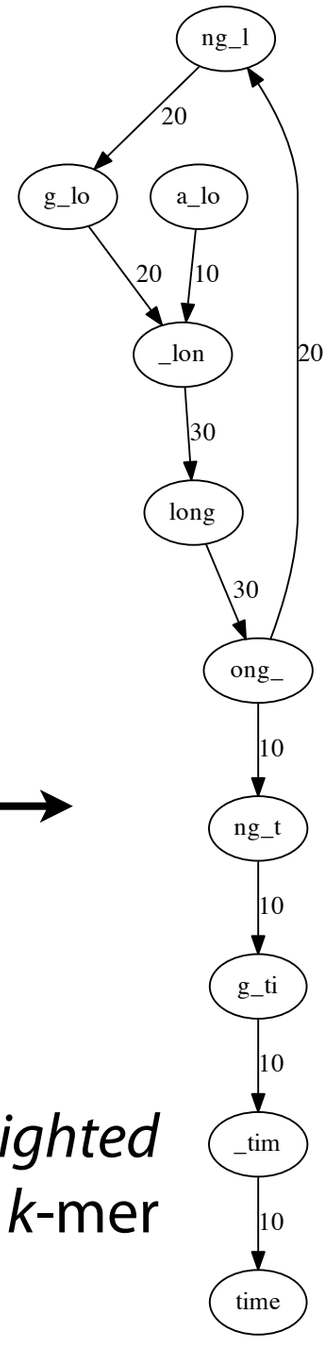
Alternative 1: Overlap-Layout-Consensus (OLC) assembly

Alternative 2: De Bruijn graph (DBG) assembly



De Bruijn graph



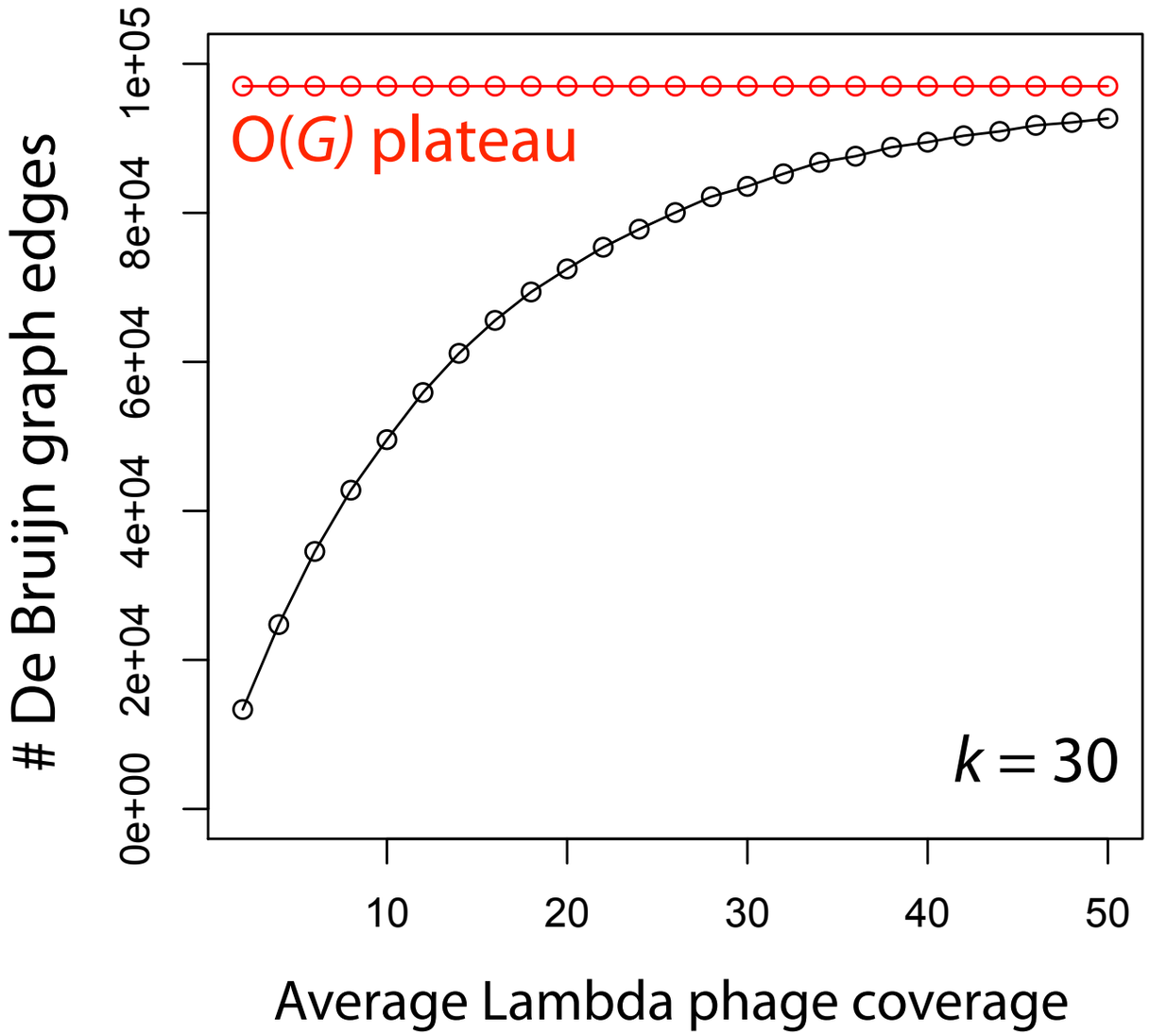


Before: one edge per *k*-mer

After: one *weighted* edge per *distinct k*-mer

Error correction

When data is error-free, # nodes, edges in De Bruijn graph is O(min(*G*, *N*))



What about data with sequencing errors?

Error correction

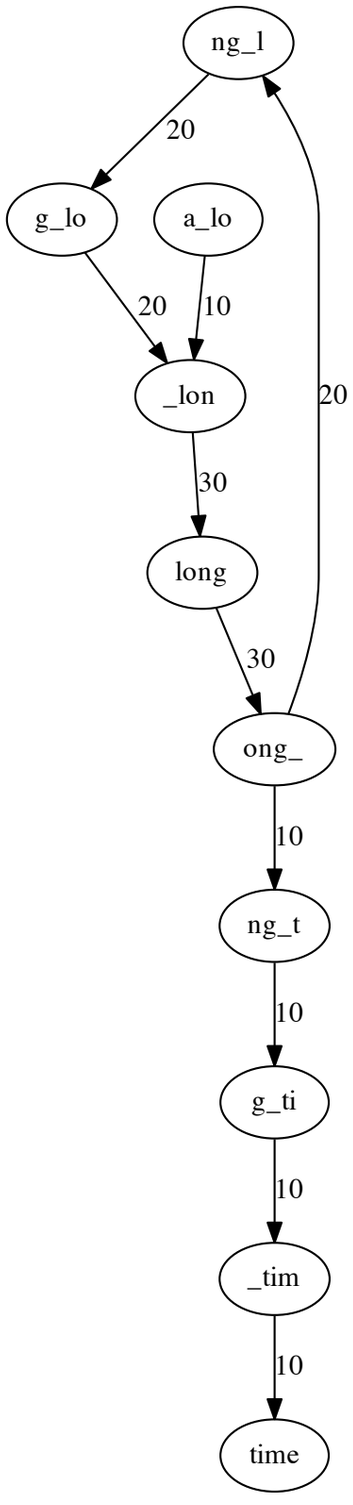
Correcting errors up-front prevents De Bruijn graph from growing far beyond O(*G)* plateau

How to correct?

Analogy: how to spell check a language you've never seen before?

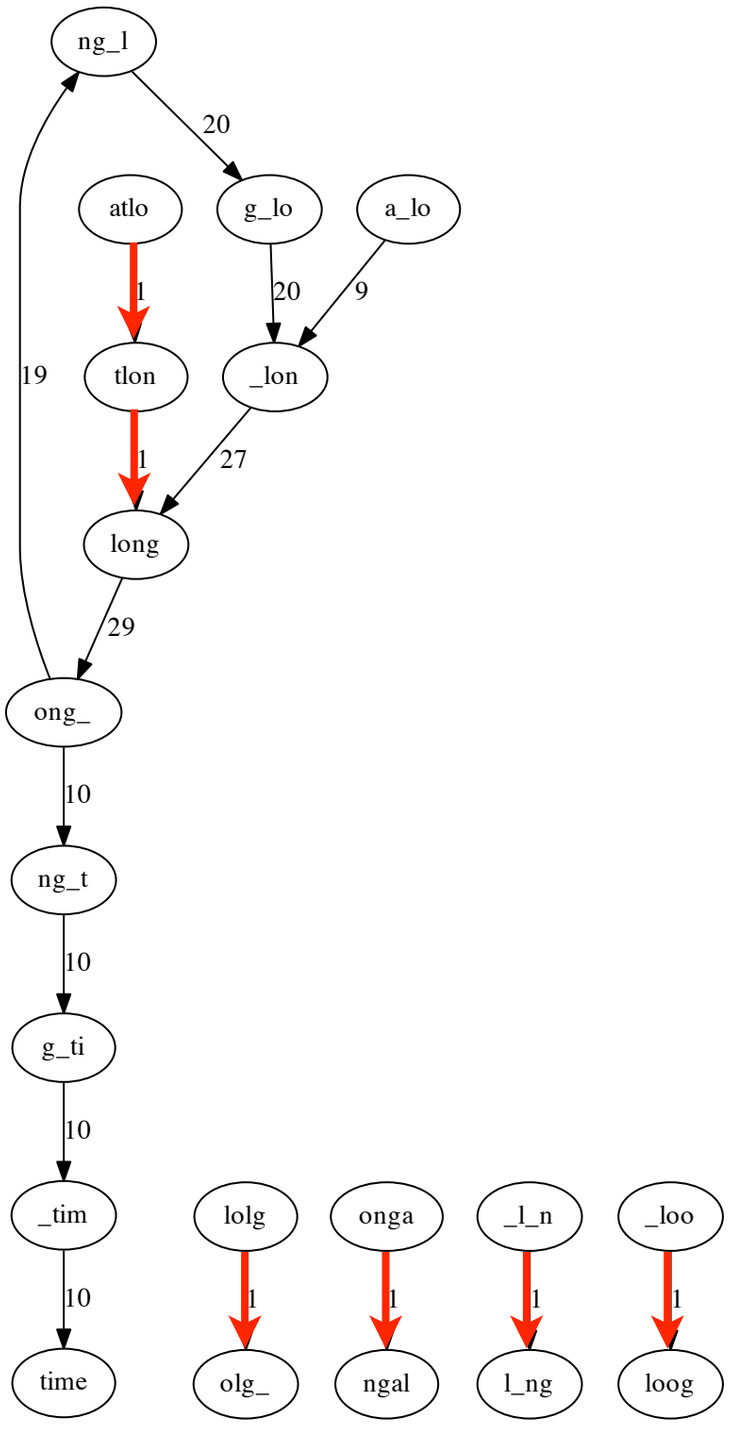
Errors tend to turn frequent words (*k*-mers) to infrequent ones. Corrections should do the reverse.

Error correction



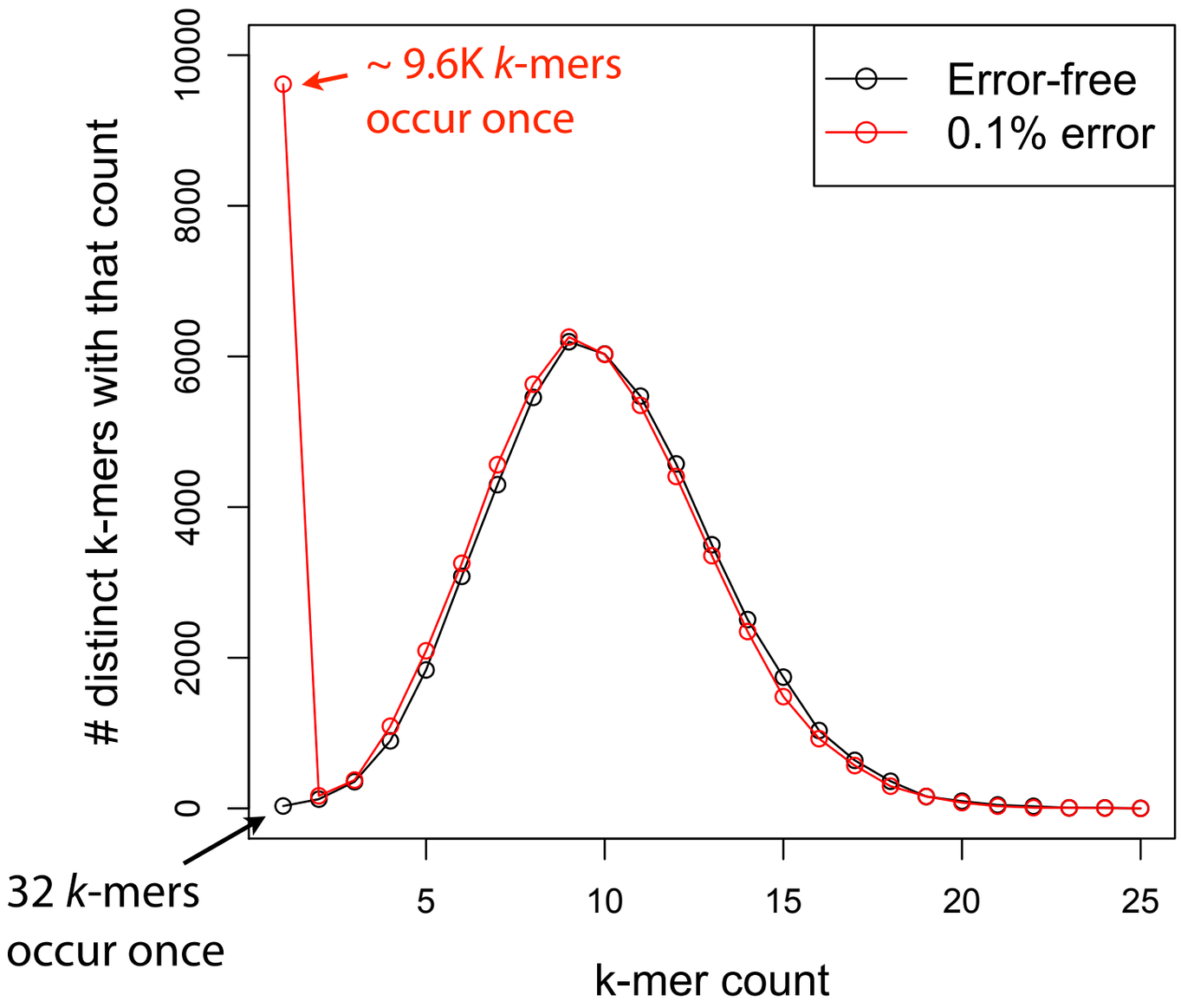
Left: Take example, mutate a *k*-mer character randomly with probability 1%

Right: 6 errors yield 10 new nodes, 6 new weighted edges, all with weight 1

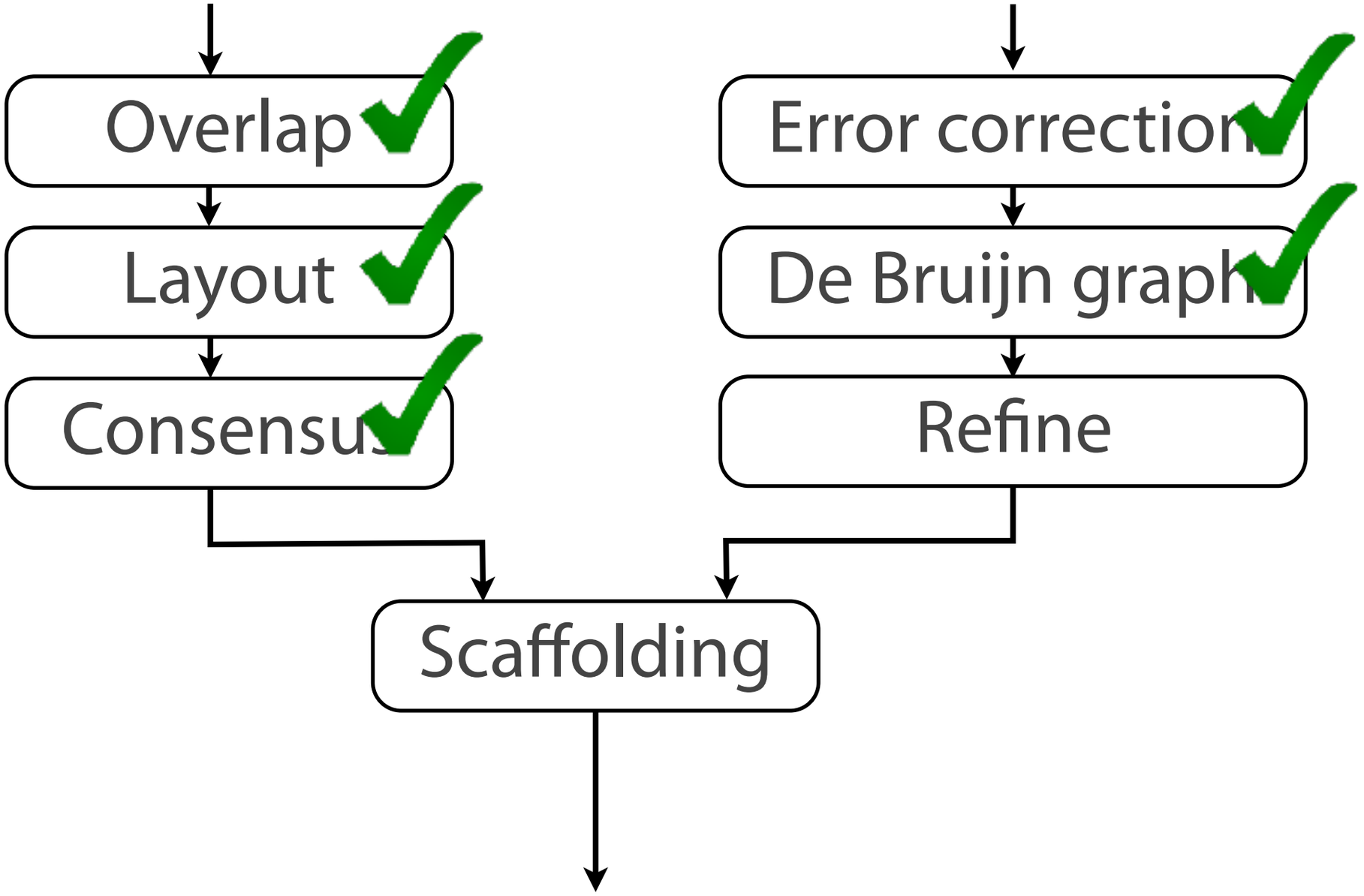


Error correction

k-mers with errors usually occur fewer times than error-free *k*-mers



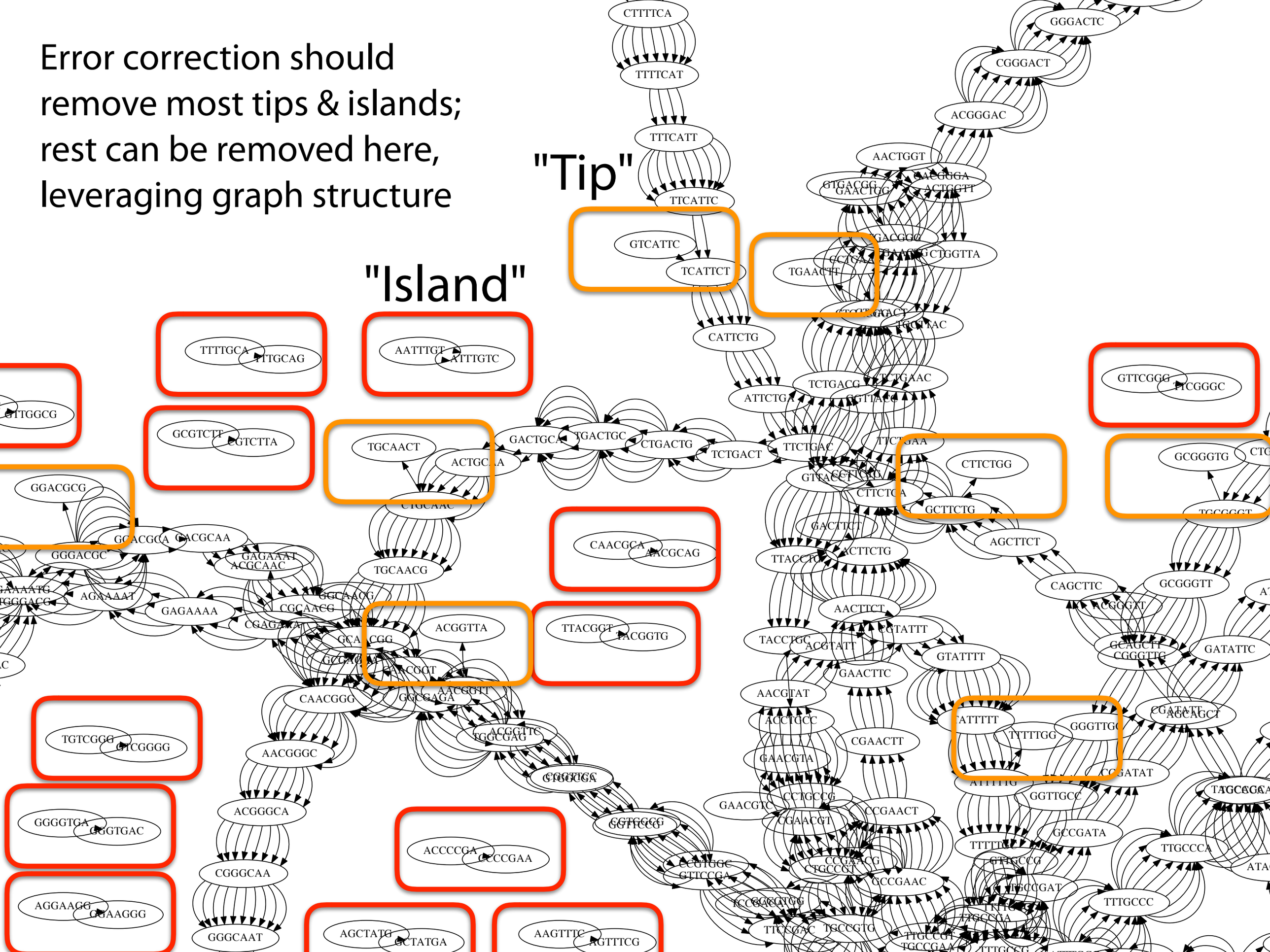
Assembly alternatives

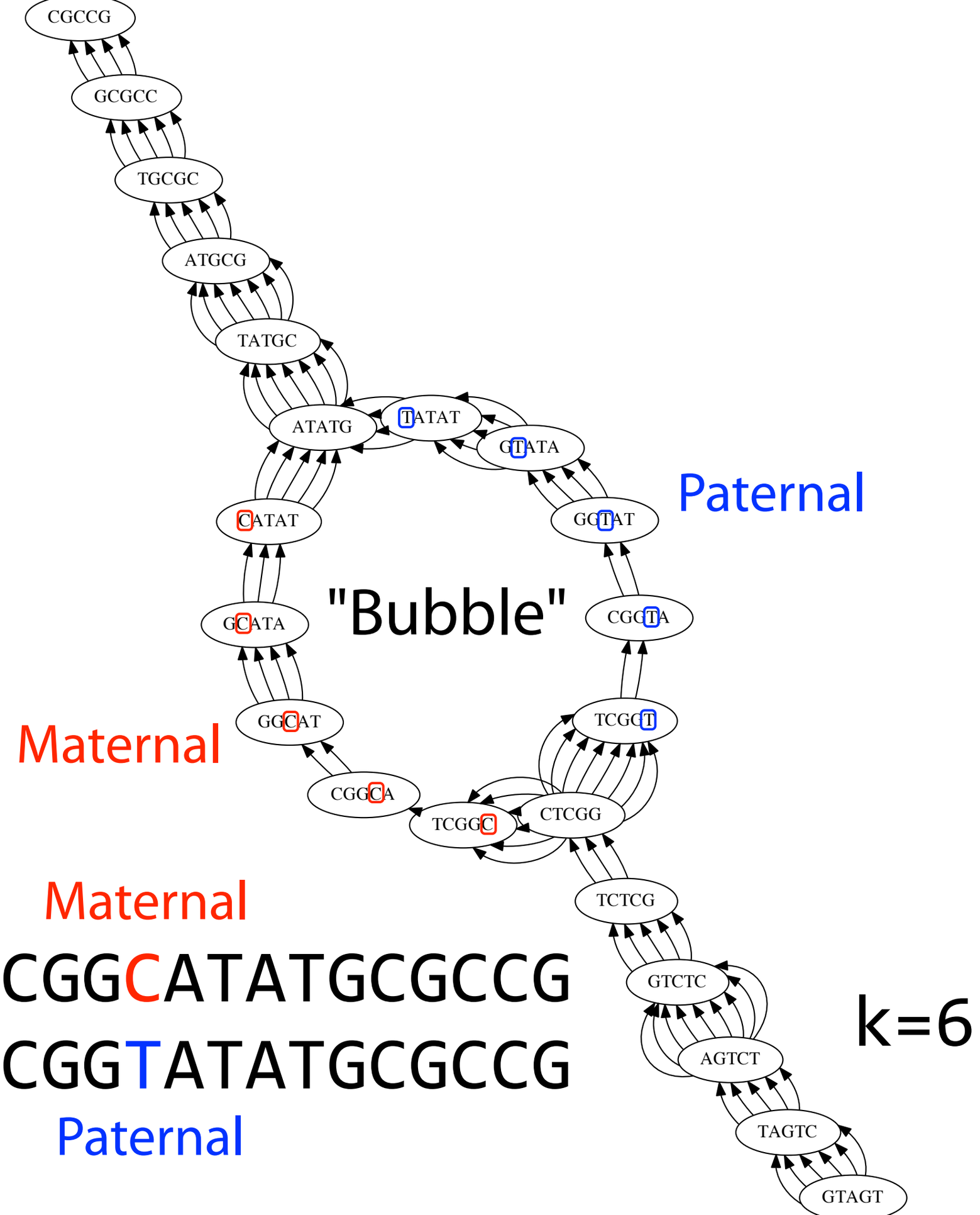


Error correction should remove most tips & islands; rest can be removed here, leveraging graph structure

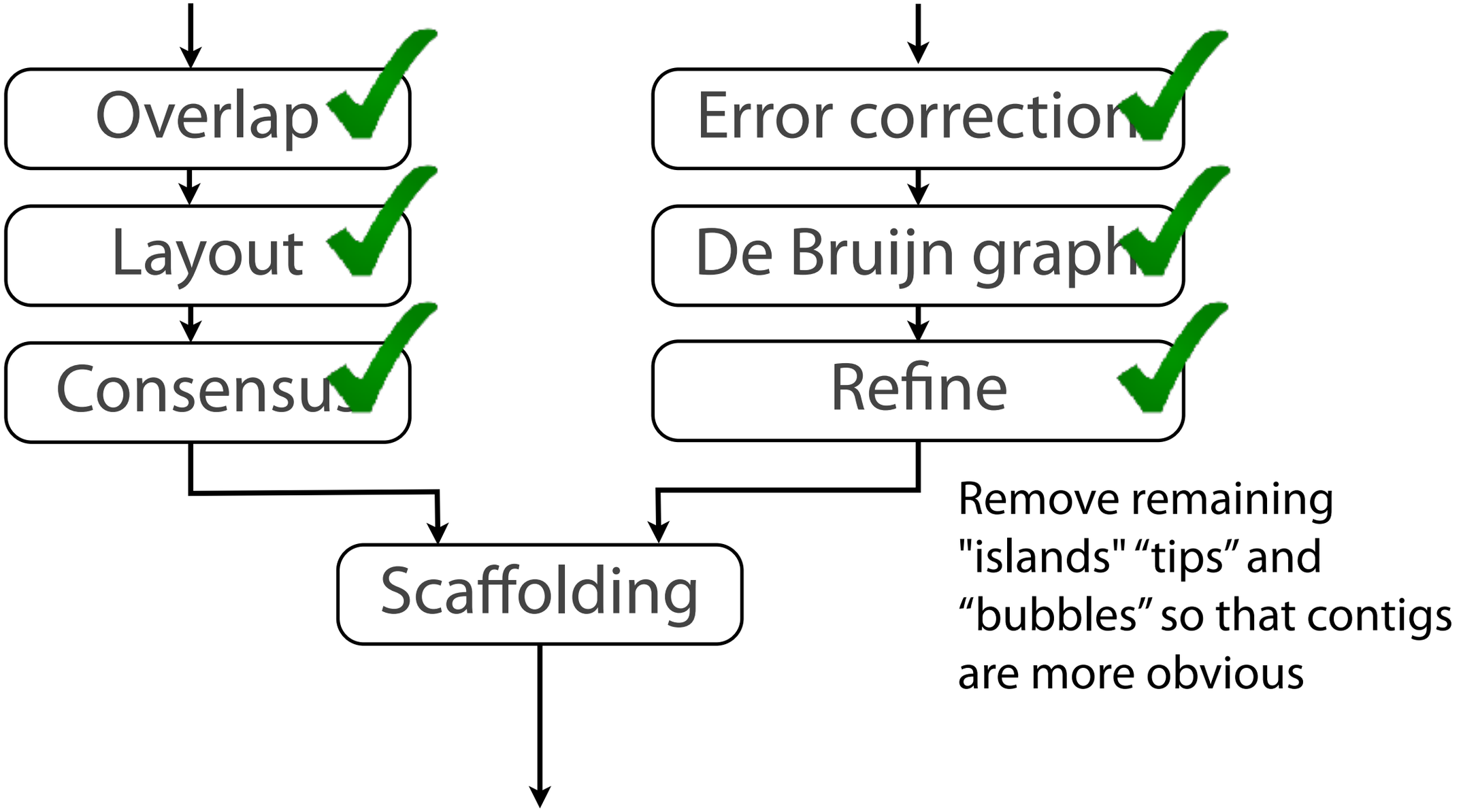
"Tip"

"Island"




CGCCG
GCGCC
TGCGC
ATGCG
TATGC
ATATG
TATAT
GTATA
GGTAT
CATAT
GCATA
CGGTA
GGCAT
TCGGT
CGGCA
TCGGC
CTCGG
TCTCG
GTCTC
AGTCT
TAGTC
GTAGT
Paternal
"Bubble"
Maternal
Maternal
GTAGTCTCGGCATATGCGCCG
GTAGTCTCGGTATATGCGCCG
Paternal
k=6

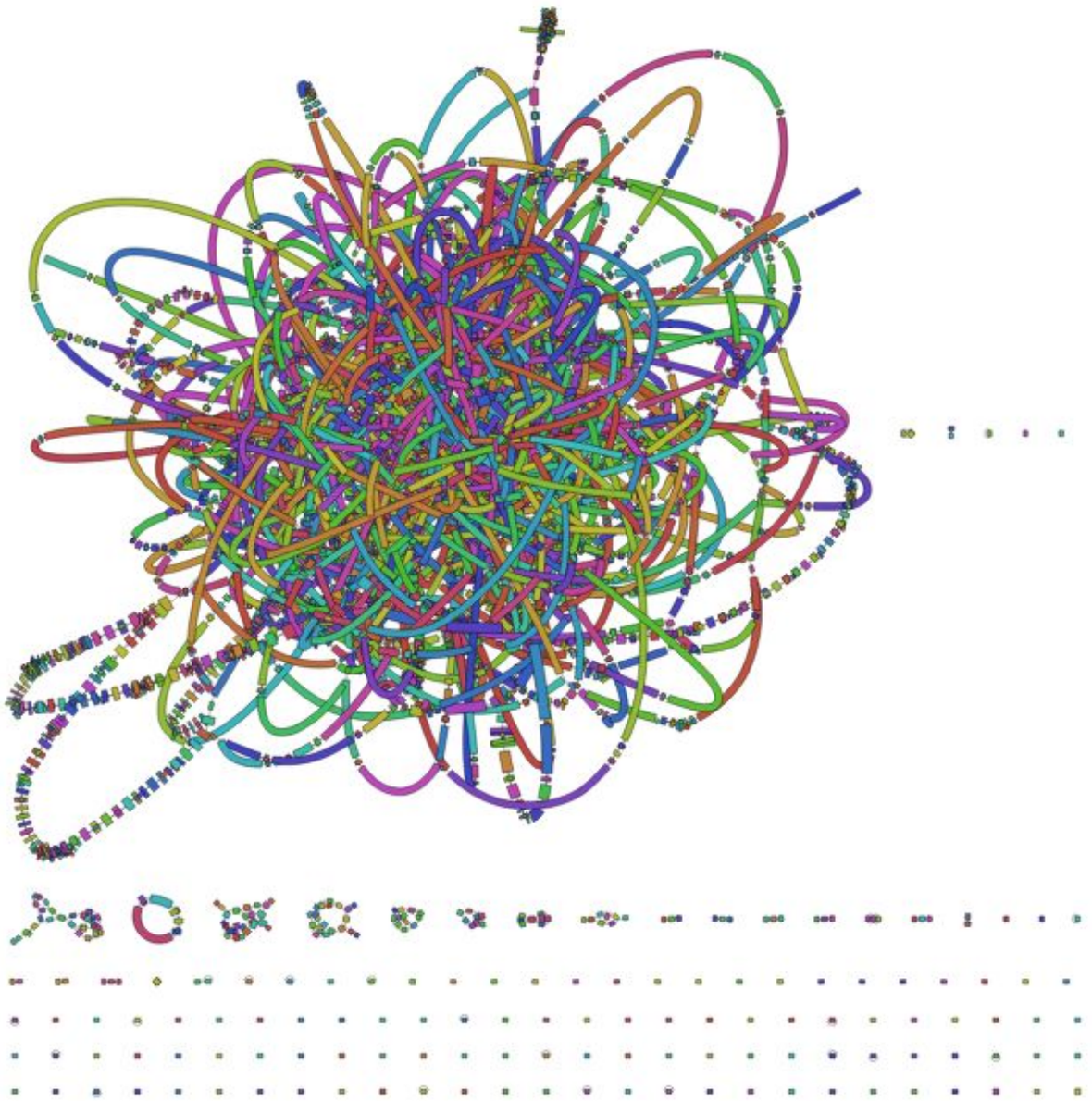
Assembly alternatives



Impact of k-mer size

- *Salmonella* genome assembly of 100bp Illumina reads

- **51-mer** = 4618 nodes and 6070 edges

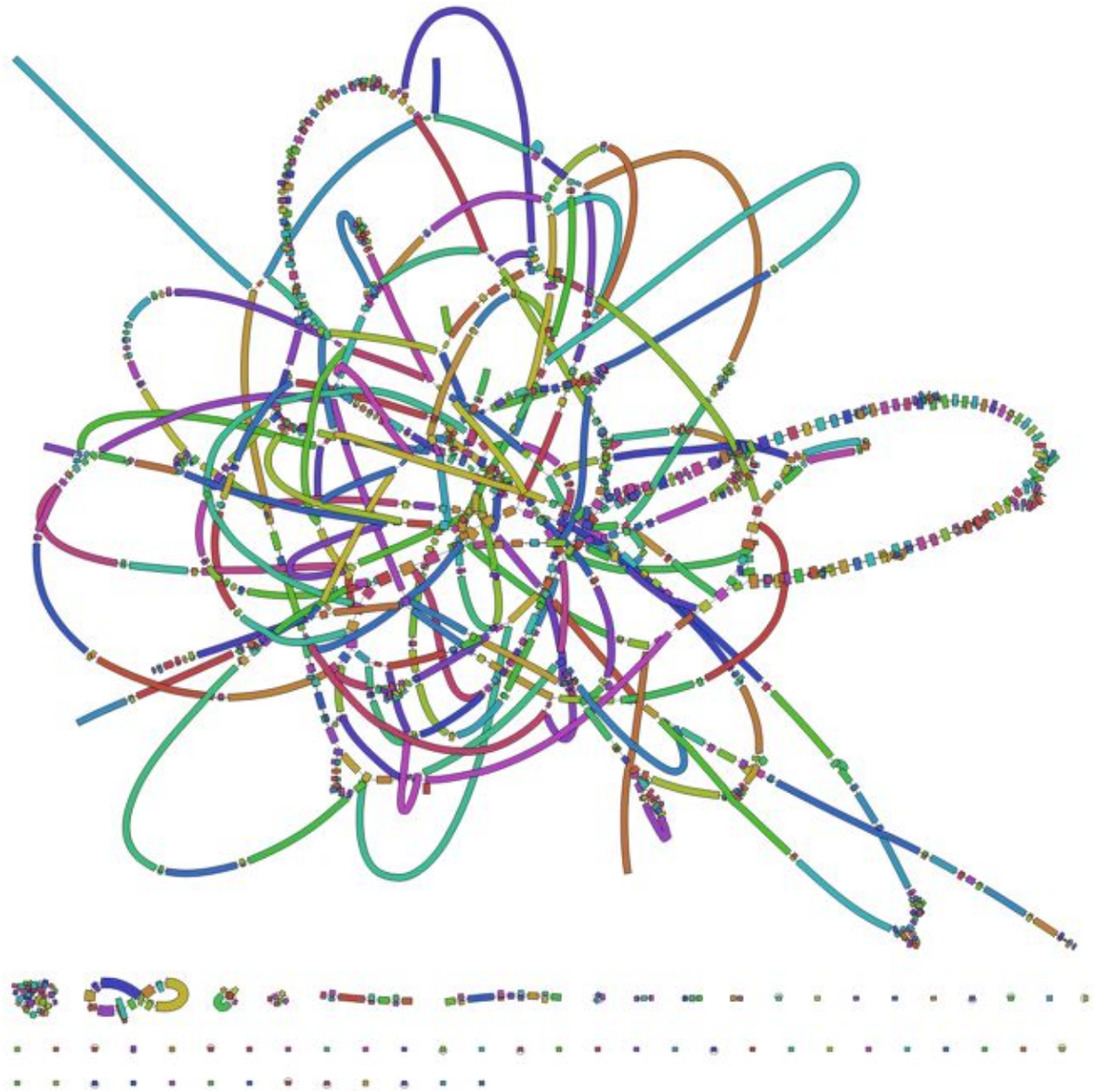


https://github.com/rrwick/Bandage/wiki/Effect-of-kmer-size

Impact of k-mer size

- *Salmonella* genome assembly of 100bp Illumina reads

- **61-mer** = 1357 nodes and 1768 edges

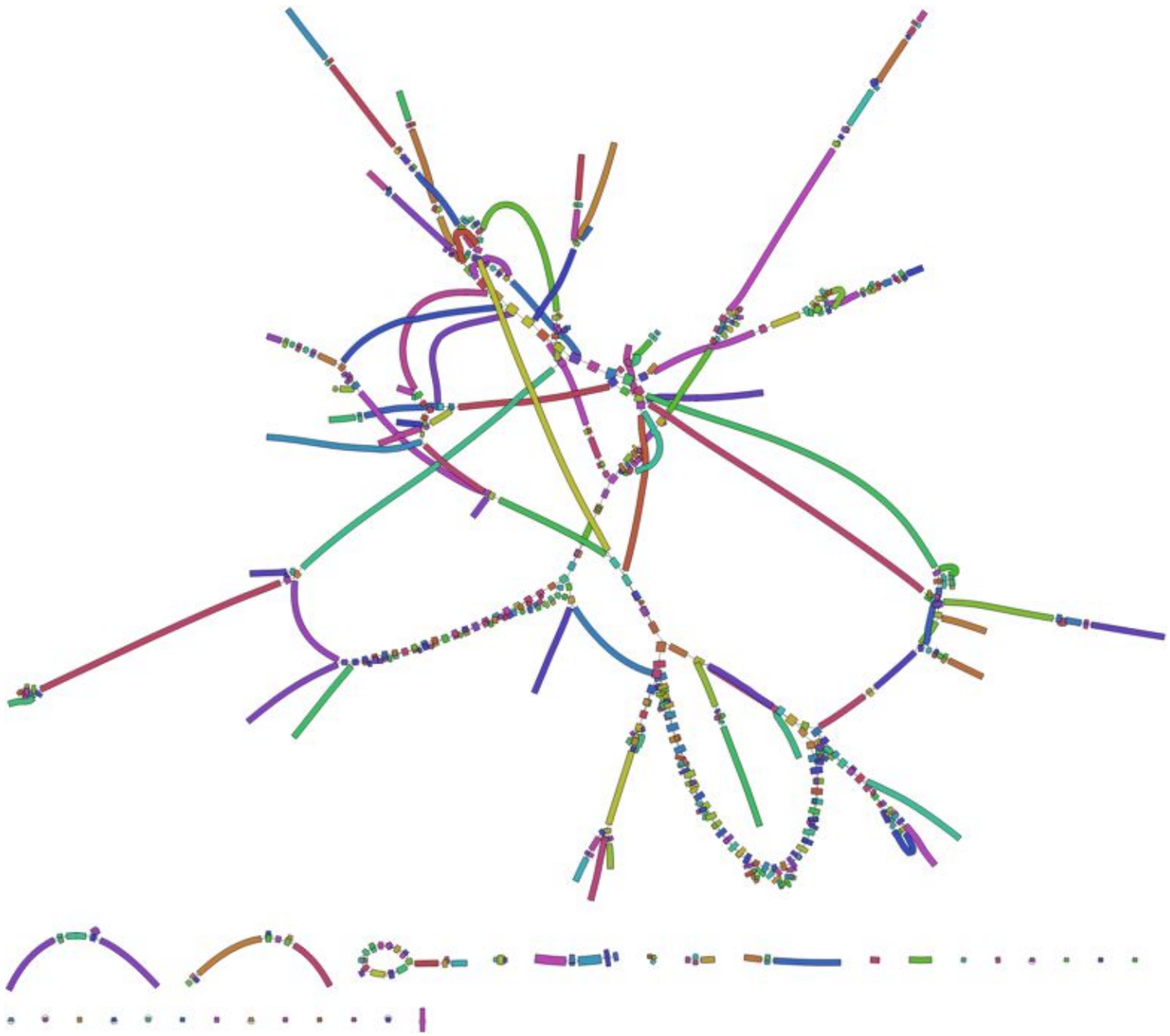


https://github.com/rrwick/Bandage/wiki/Effect-of-kmer-size

Impact of k-mer size

- *Salmonella* genome assembly of 100bp Illumina reads

- **71-mer** = 611 nodes and 765 edges

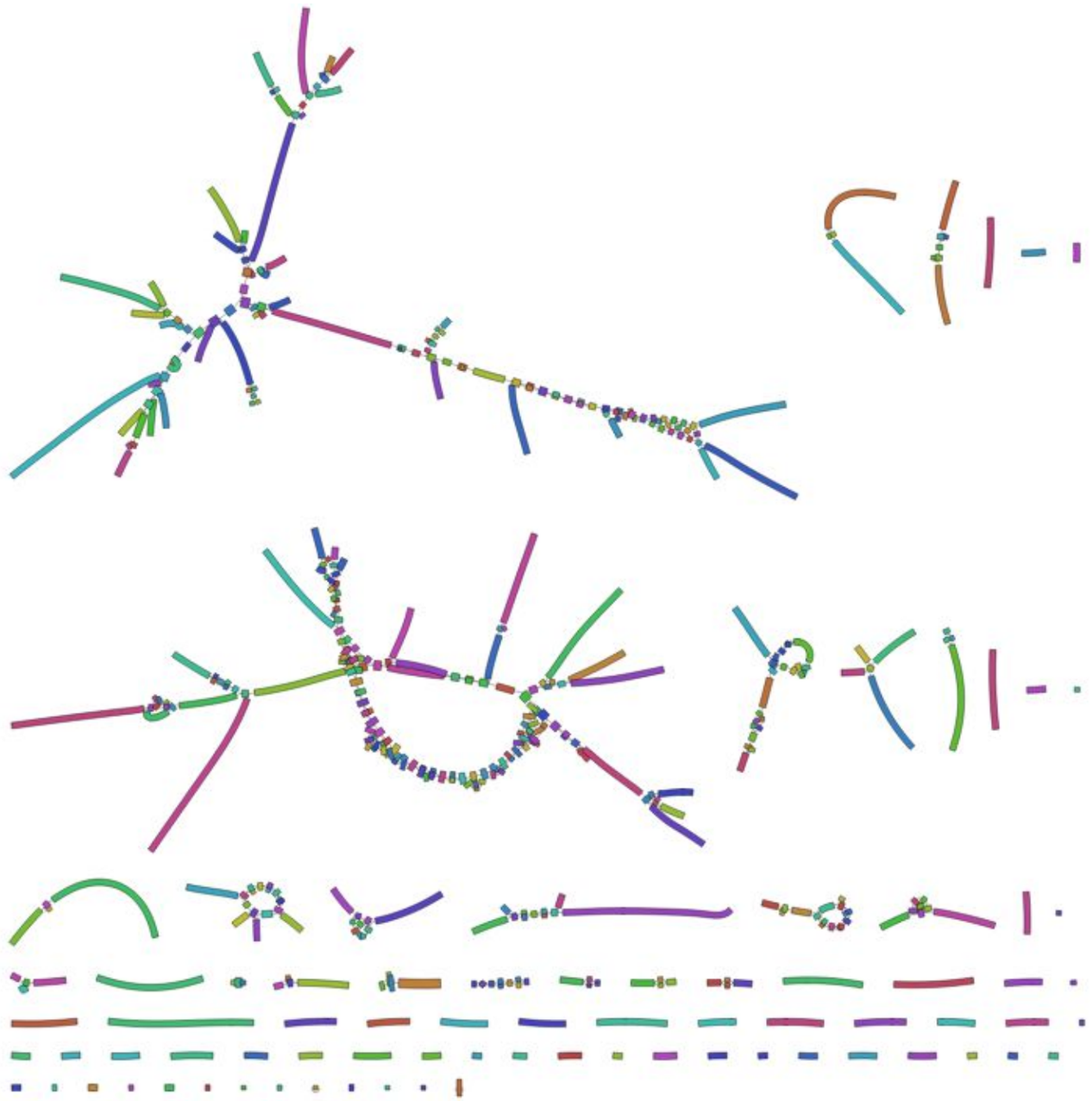


https://github.com/rrwick/Bandage/wiki/Effect-of-kmer-size

Impact of k-mer size

- *Salmonella* genome assembly of 100bp Illumina reads

- **81-mer** = 490 nodes and 512 edges



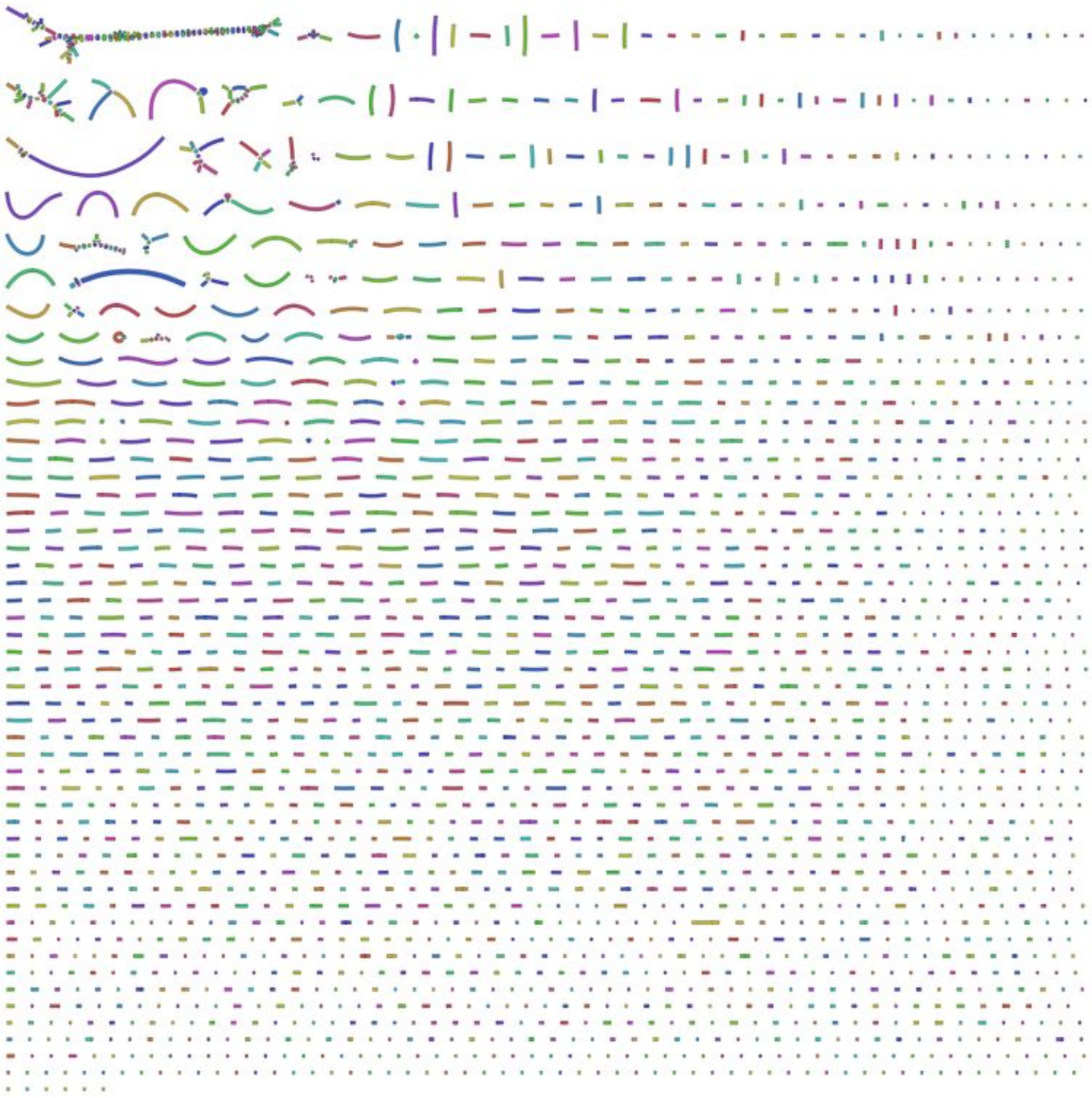
https://github.com/rrwick/Bandage/wiki/Effect-of-kmer-size

Impact of k-mer size

- *Salmonella* genome assembly of 100bp Illumina reads

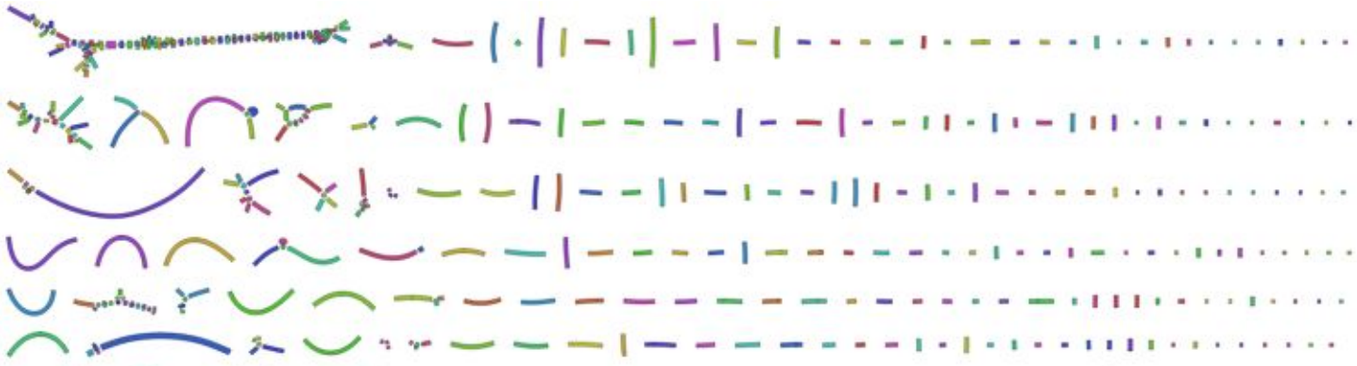
- **91-mer** = 2386 nodes and 304 edges

https://github.com/rrwick/Bandage/wiki/Effect-of-kmer-size



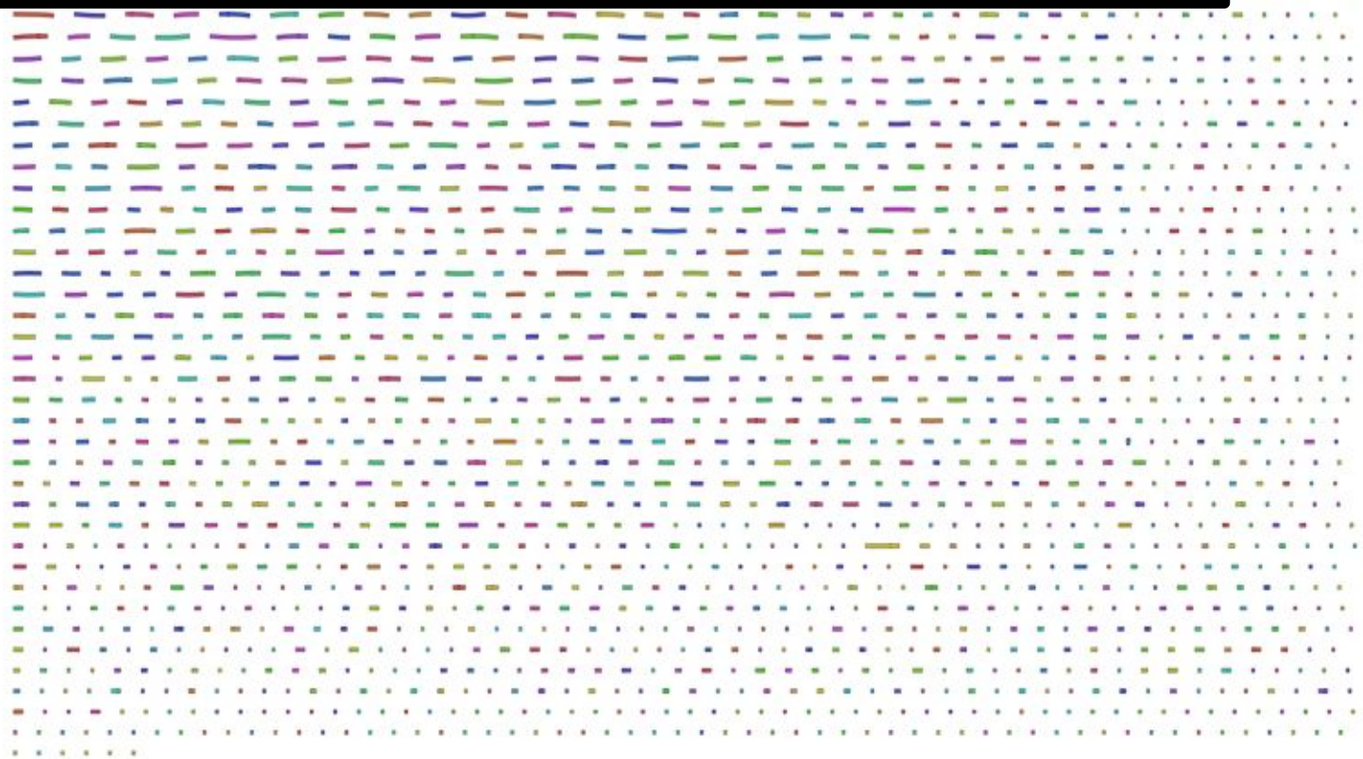
Impact of k-mer size

- *Salmonella* genome assembly of 100bp Illumina reads



- **91**
304 edges

Solution: Use a range of k-mer sizes and reconcile the results
- SPAdes Assembler

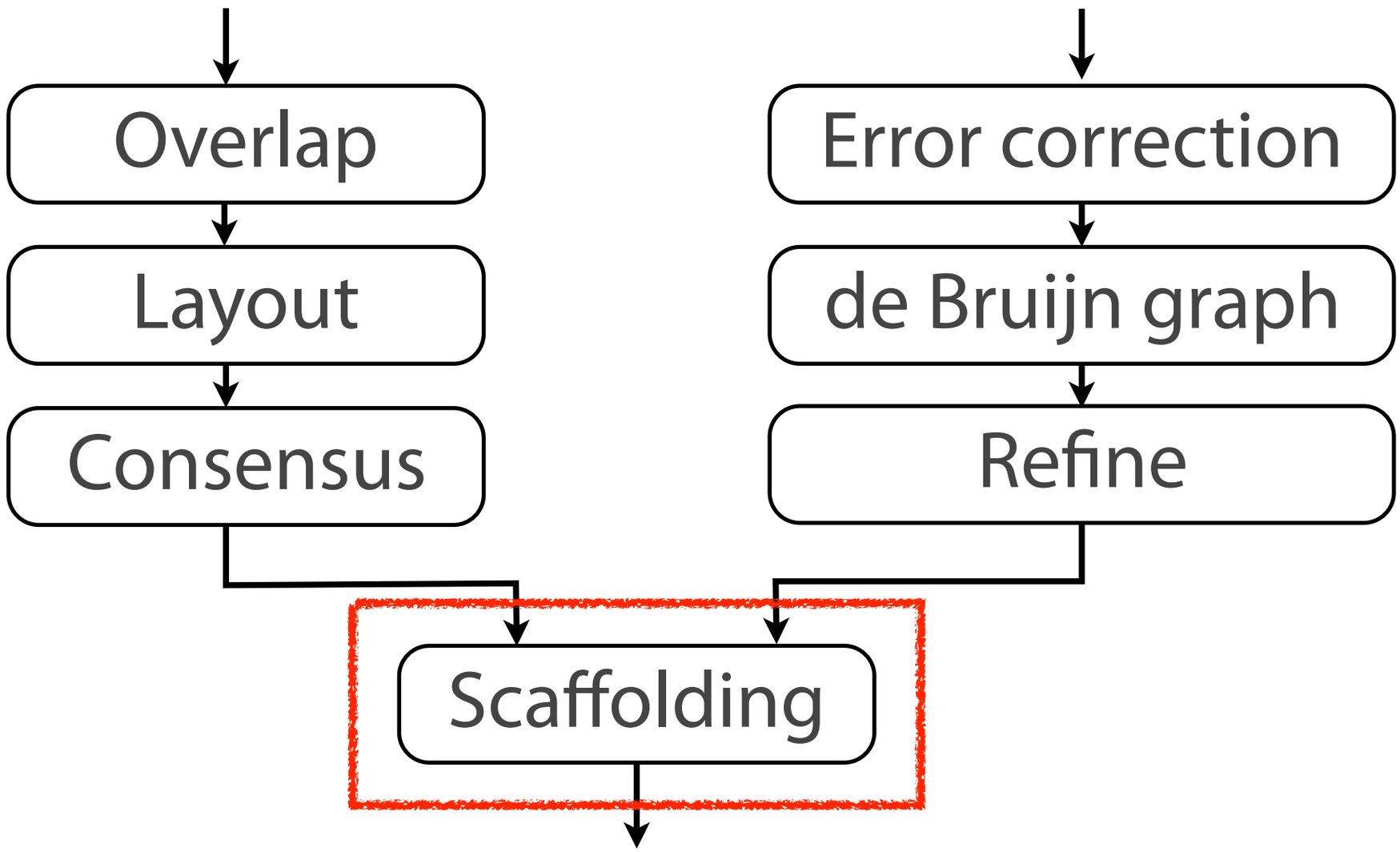


https://github.com/rrwick/Bandage/wiki/Effect-of-kmer-size

Assembly paradigms

1: Overlap-Layout-Consensus (OLC) assembly

2: de Bruijn graph (DBG) assembly



Scaffolding

Both OLC and DBG are concerned with constructing the longest, most accurate *contigs* possible

Contig is a stretch of unambiguously assembled sequence

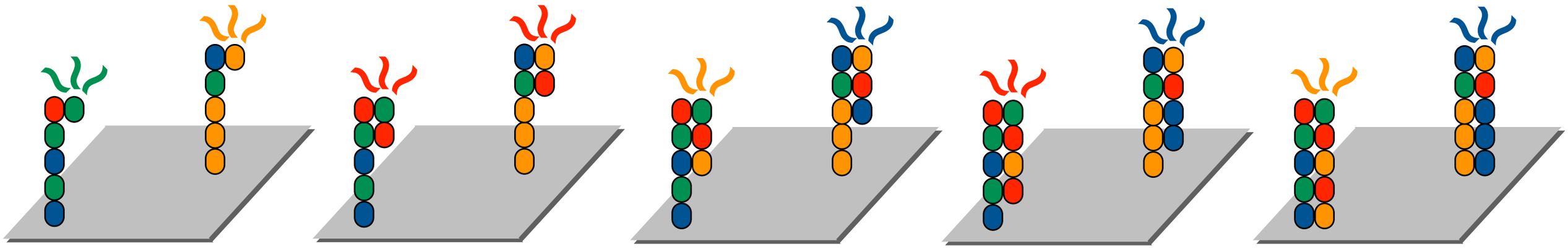
Scaffolding *orders* and *orients* contigs with respect to each other

For this we can use data from various sources, especially *paired ends*

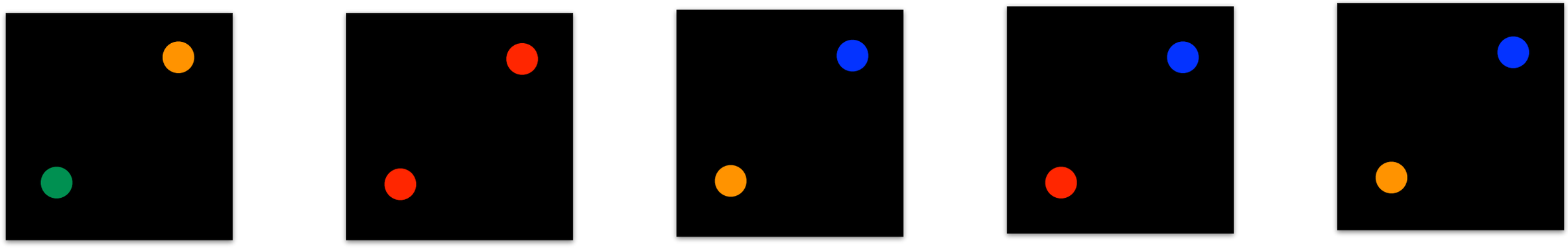
Scaffolding: paired-end sequencing

We discussed sequencing by synthesis

Process we discussed produces one contiguous read sequence



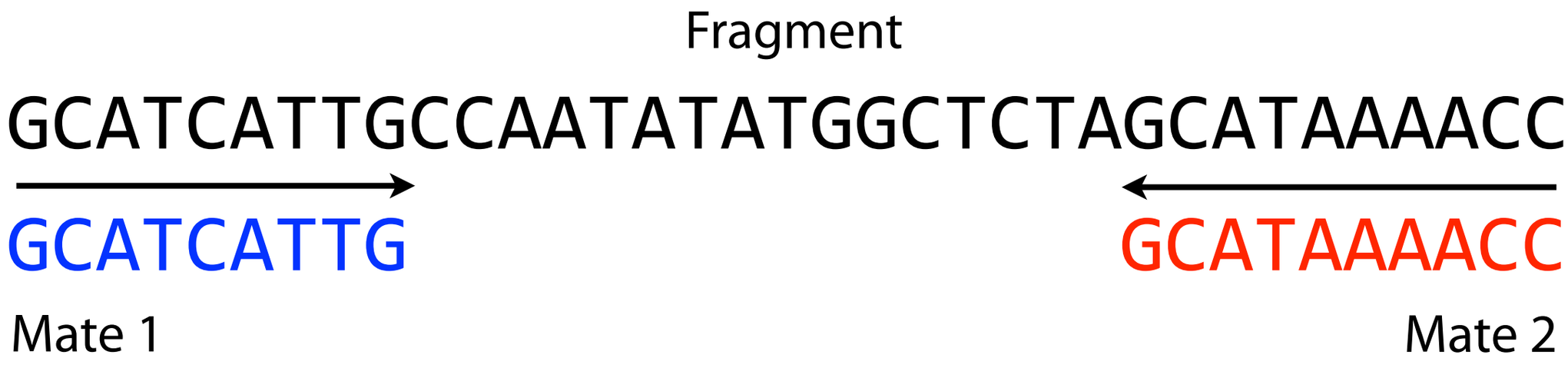
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5



Scaffolding: paired-end sequencing

Alternative protocol produces a *pair* of reads taken from either end of a longer *fragment*

Paired reads are also called *mates* to distinguish them from the *unpaired* reads we've been discussing

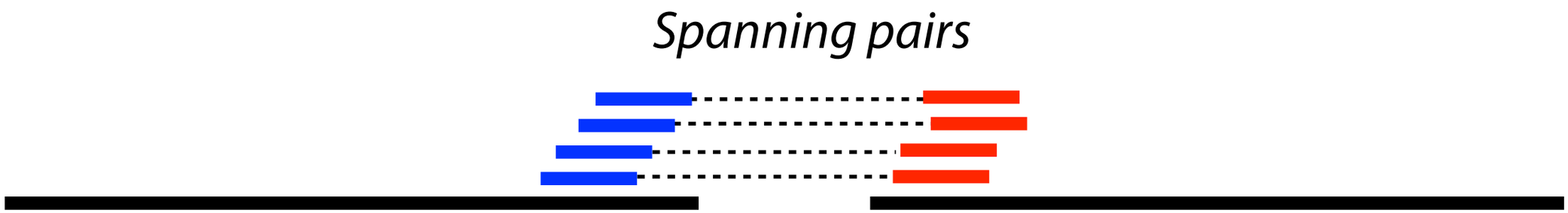


Depending on lengths, mates might overlap in the middle of the fragment

Scaffolding: paired-end sequencing

Say we have a collection of pairs and we assemble them as usual

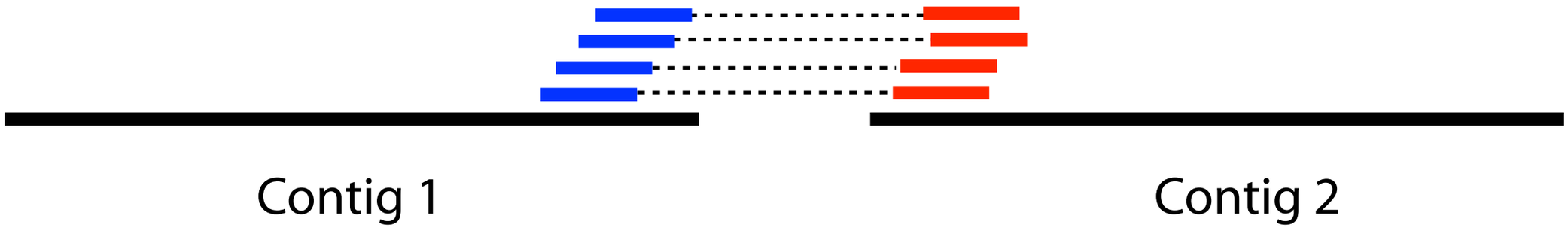
Assembly yields two contigs:



...and we discover that some of the mates at one edge of contig 1 are paired with mates in contig 2

Call these *spanning pairs*

Scaffolding: paired-end sequencing

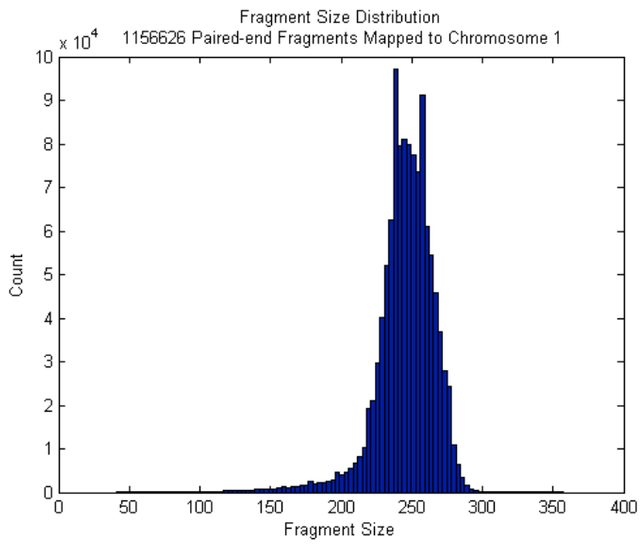


What does this tell us?

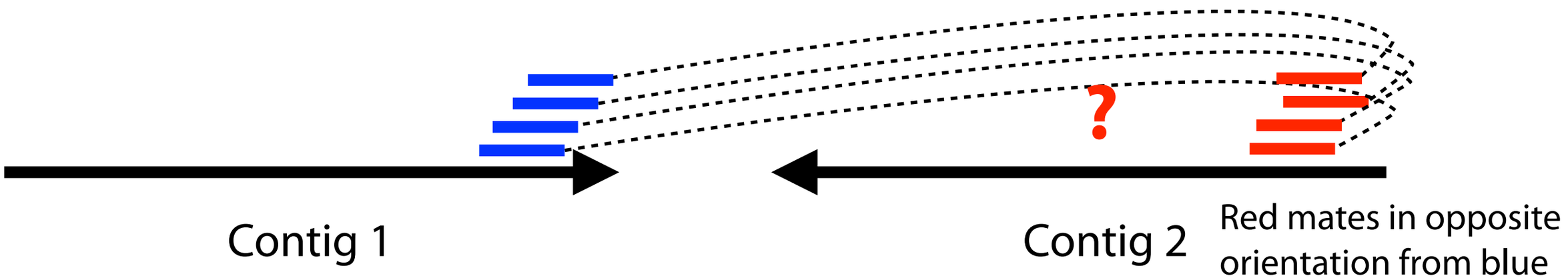
Contig 1 is close to contig 2 in the genome

In fact, we can *estimate distance between contigs* using what we know about fragment length distribution

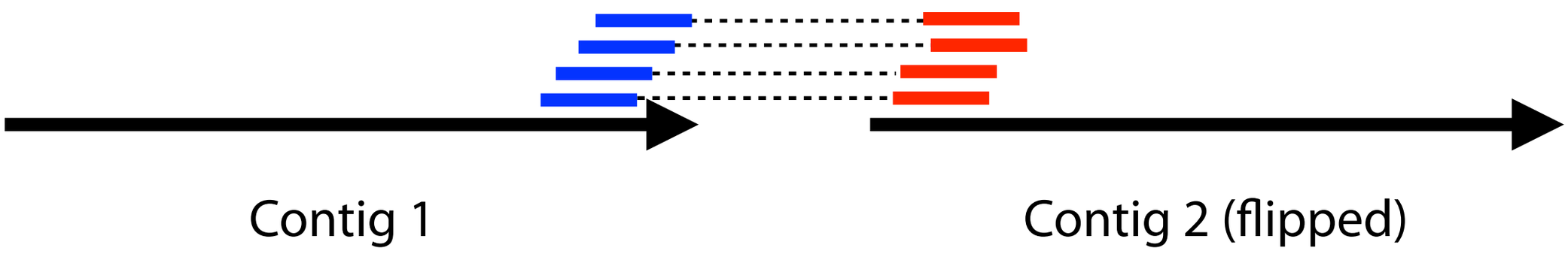
The more spanning pairs we have, the better our estimate



Scaffolding: paired-end sequencing



What does the picture look like if contigs 1 and 2 are close, but we assembled contig 2 "backwards" (i.e. reverse complemented)



Pairs also tell us about contigs' relative *orientation*

Scaffolding

Scaffolding output: collection of *scaffolds*, where a scaffold is a collection of contigs related to each other with high confidence using pairs

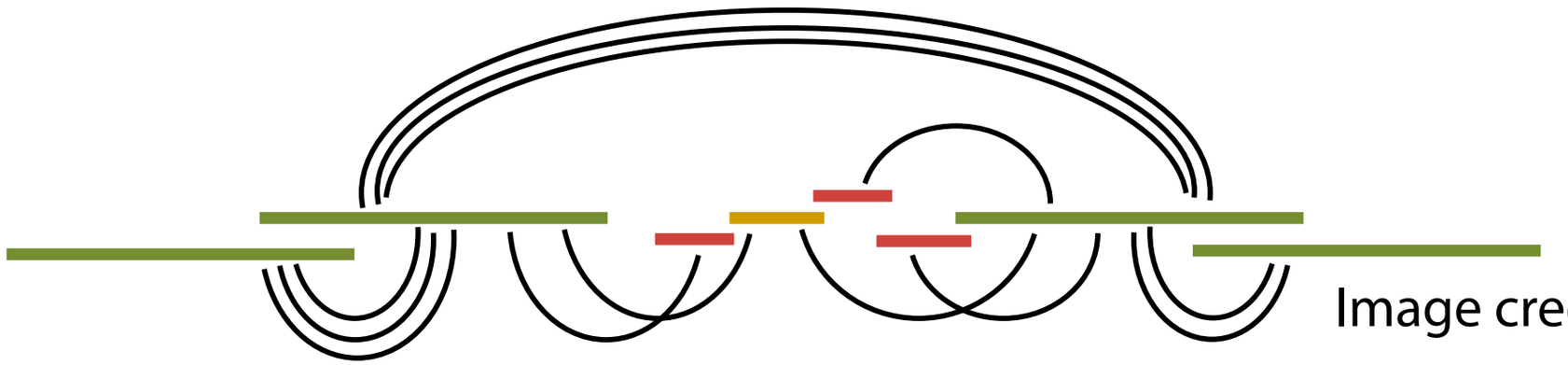


Image credit: Mike Schatz

SPAdes

Key tricks used by SPAdes assembler:

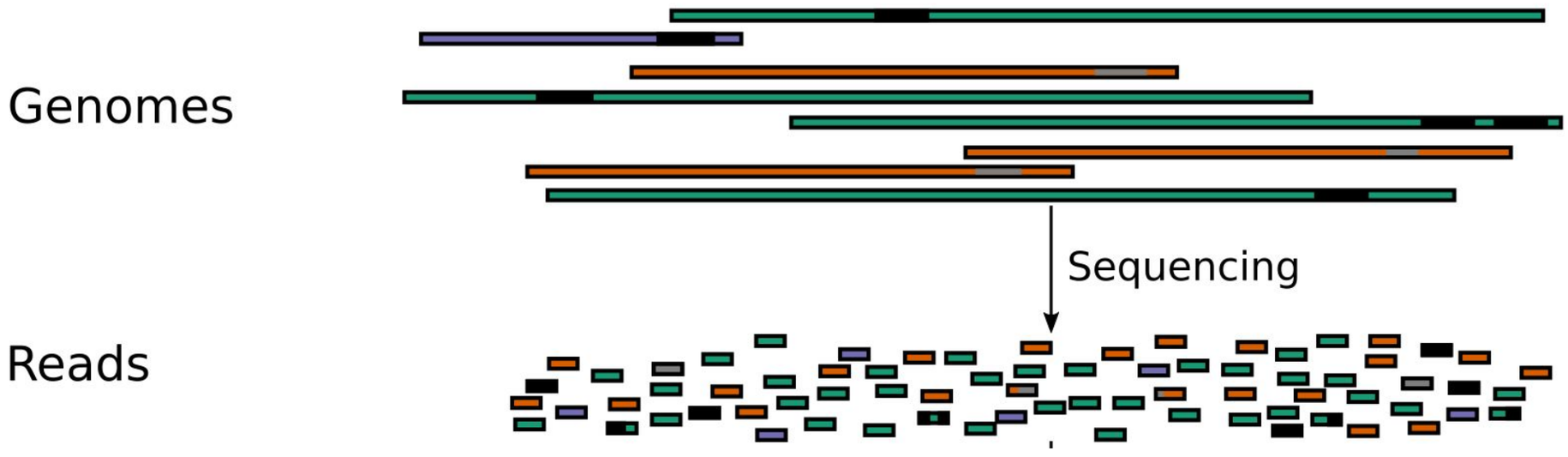
- Built-in error correction
- Generate dBGs across a range of k-mer sizes
- Use pair-end information to construct pre “scaffolded” graphs (instead of just post-processing)

Profiling many microbes at once

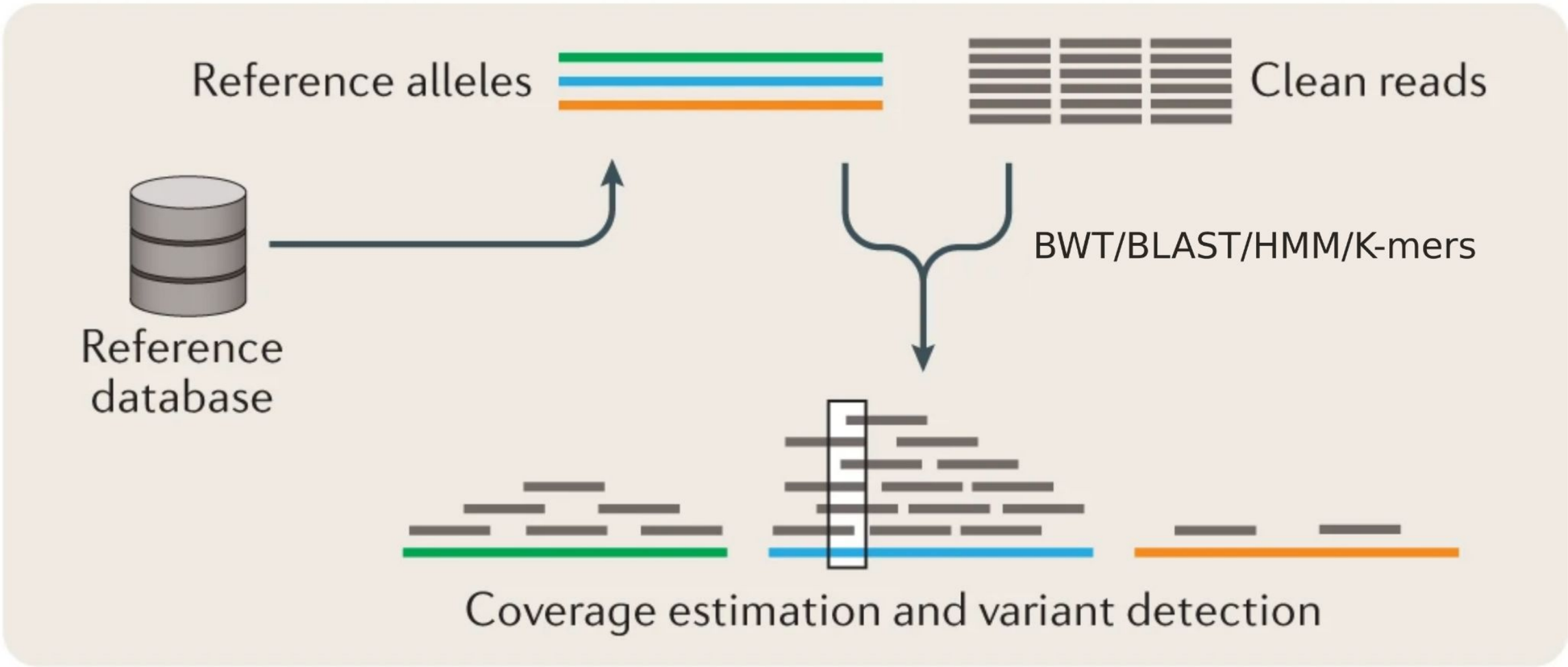
Metagenomics

Genomes

Metagenomics

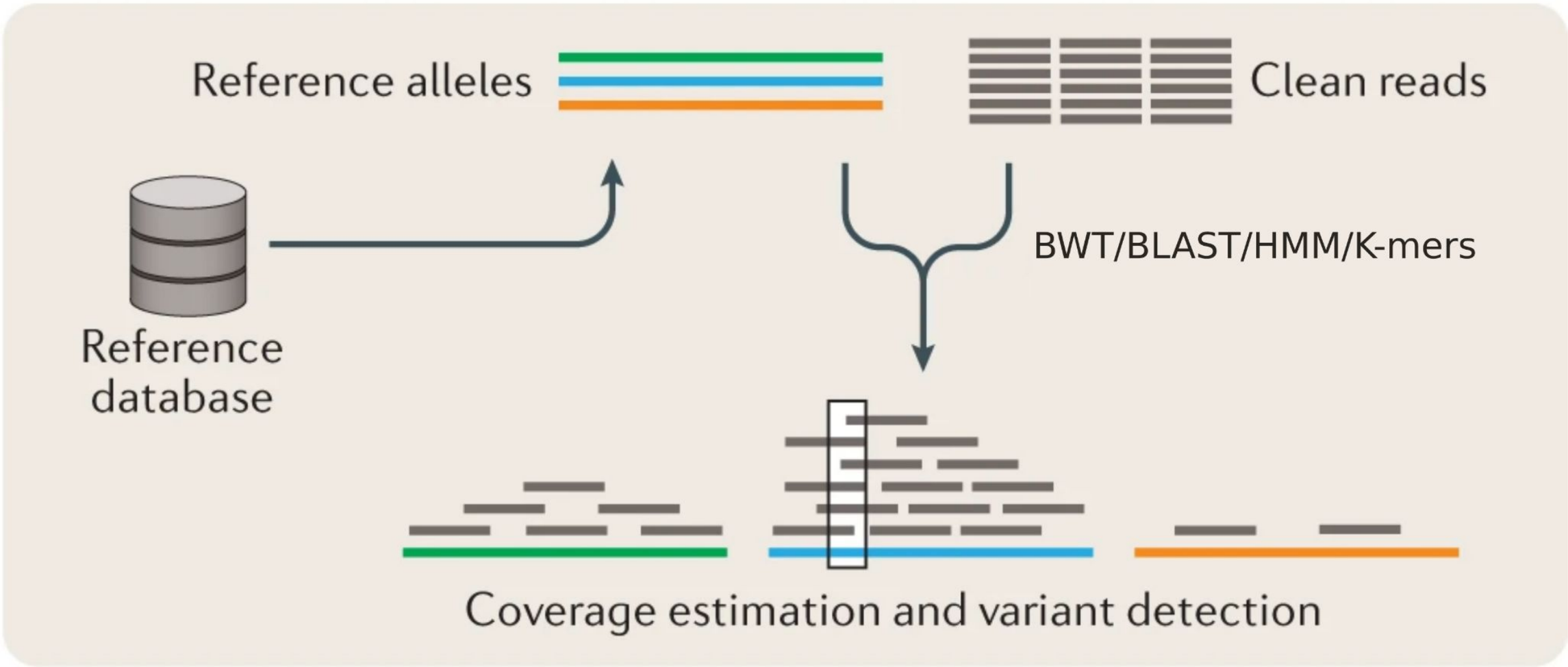


Read-based analyses



10.1038/s41576-019-0108-4

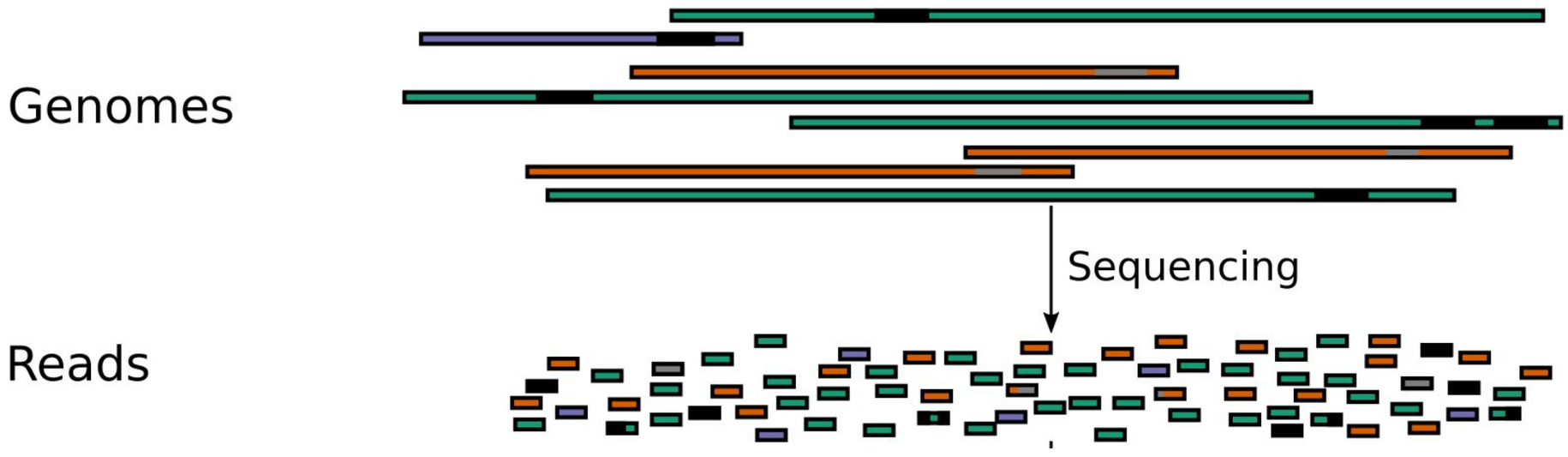
Read-based analyses



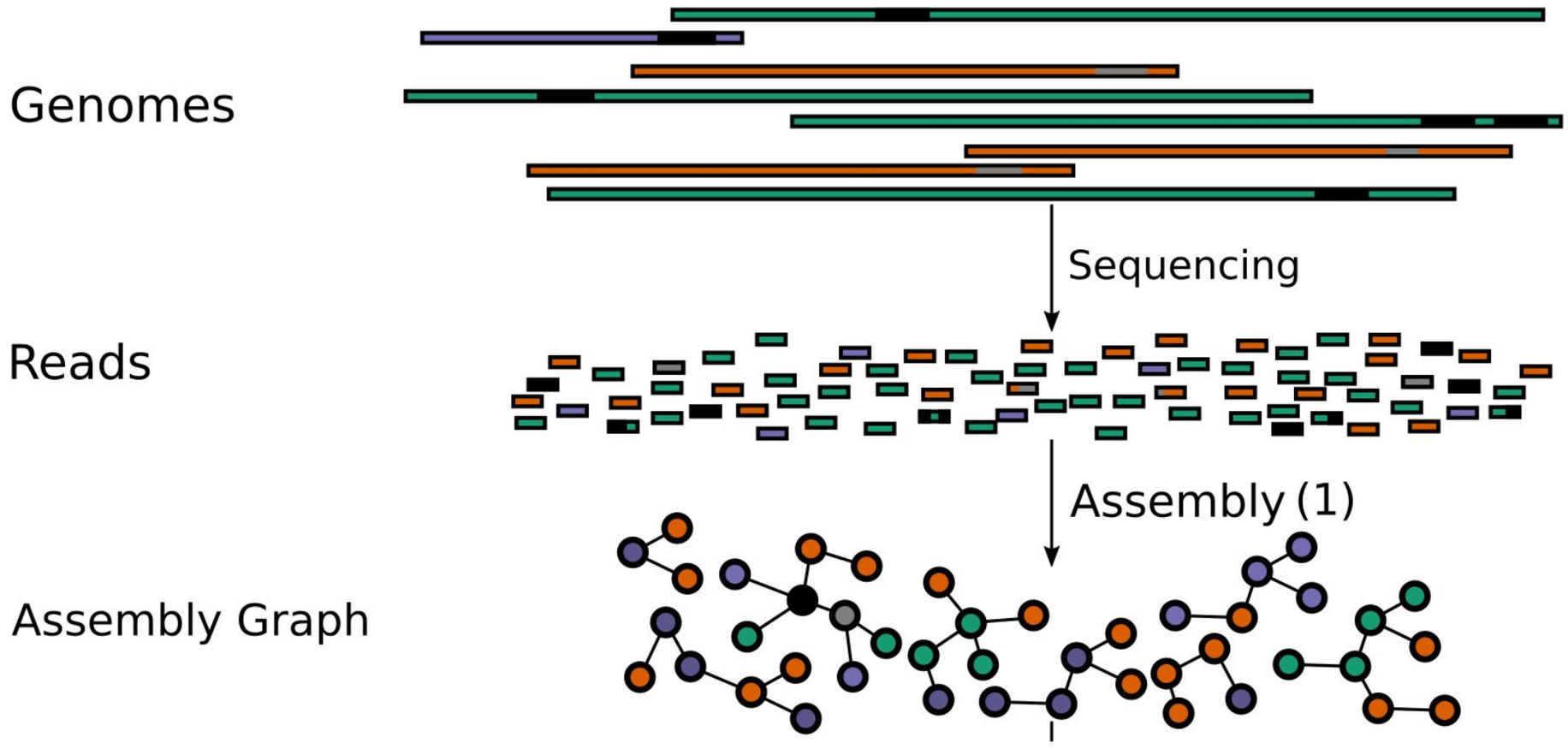
10.1038/s41576-019-0108-4

But - lose wider context, can't resolve alleles, can't find new things!

Metagenomics



Metagenomics



Metagenomics

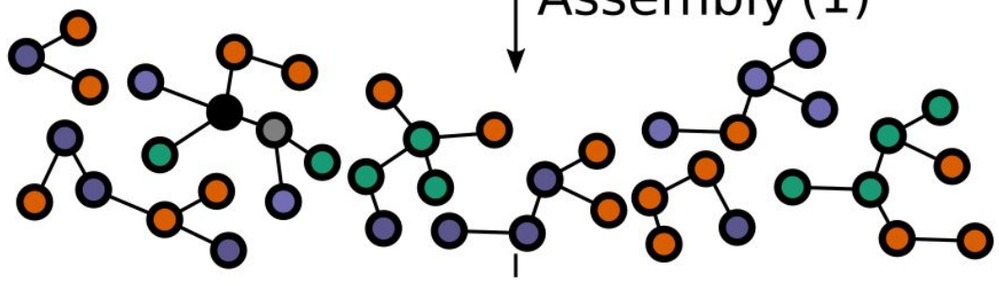
Genomes

Sequencing

Reads

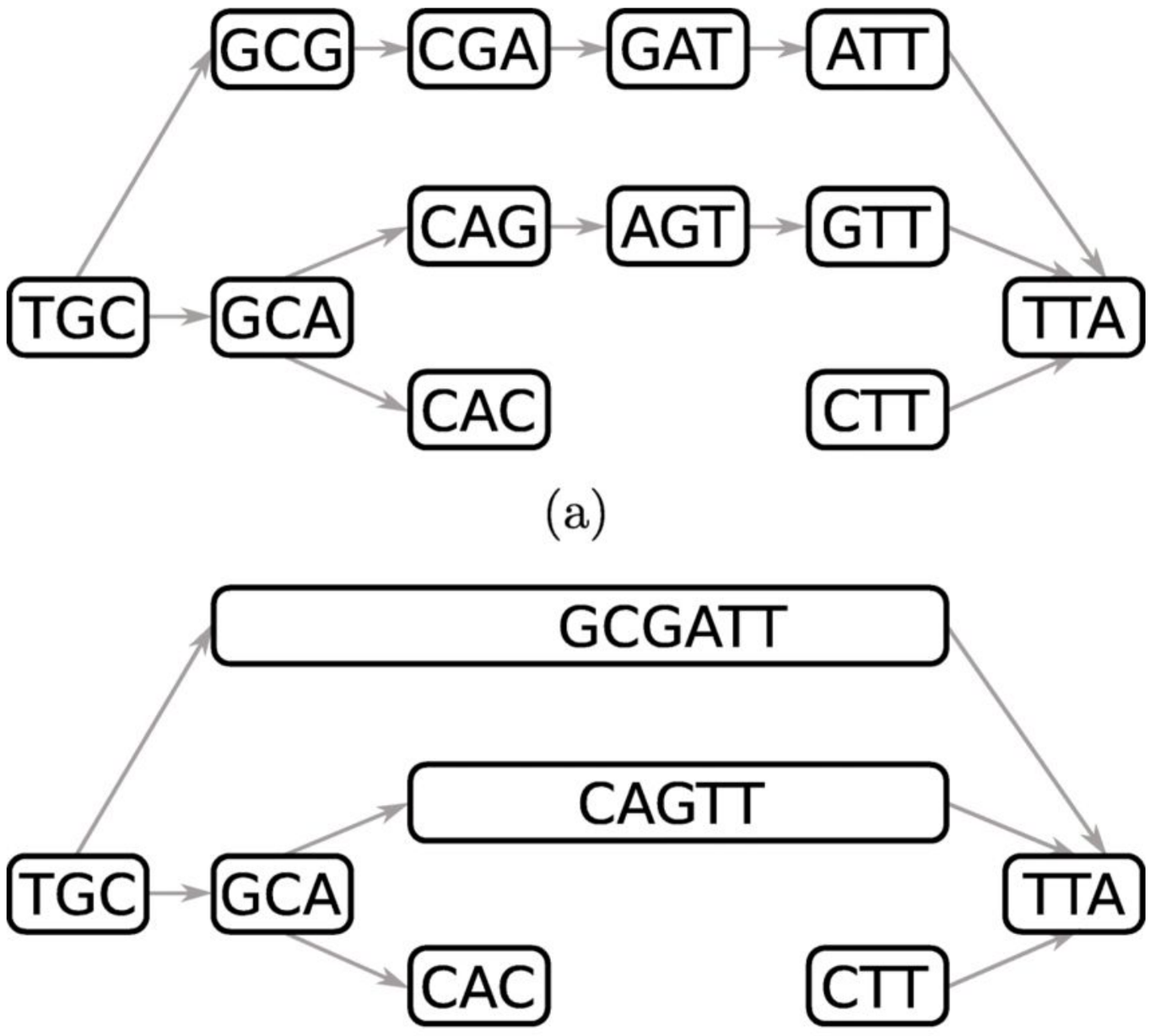
Assembly (1)

Assembly Graph



Wildly varied coverage!
How do we resolve repeats, closely related etc?

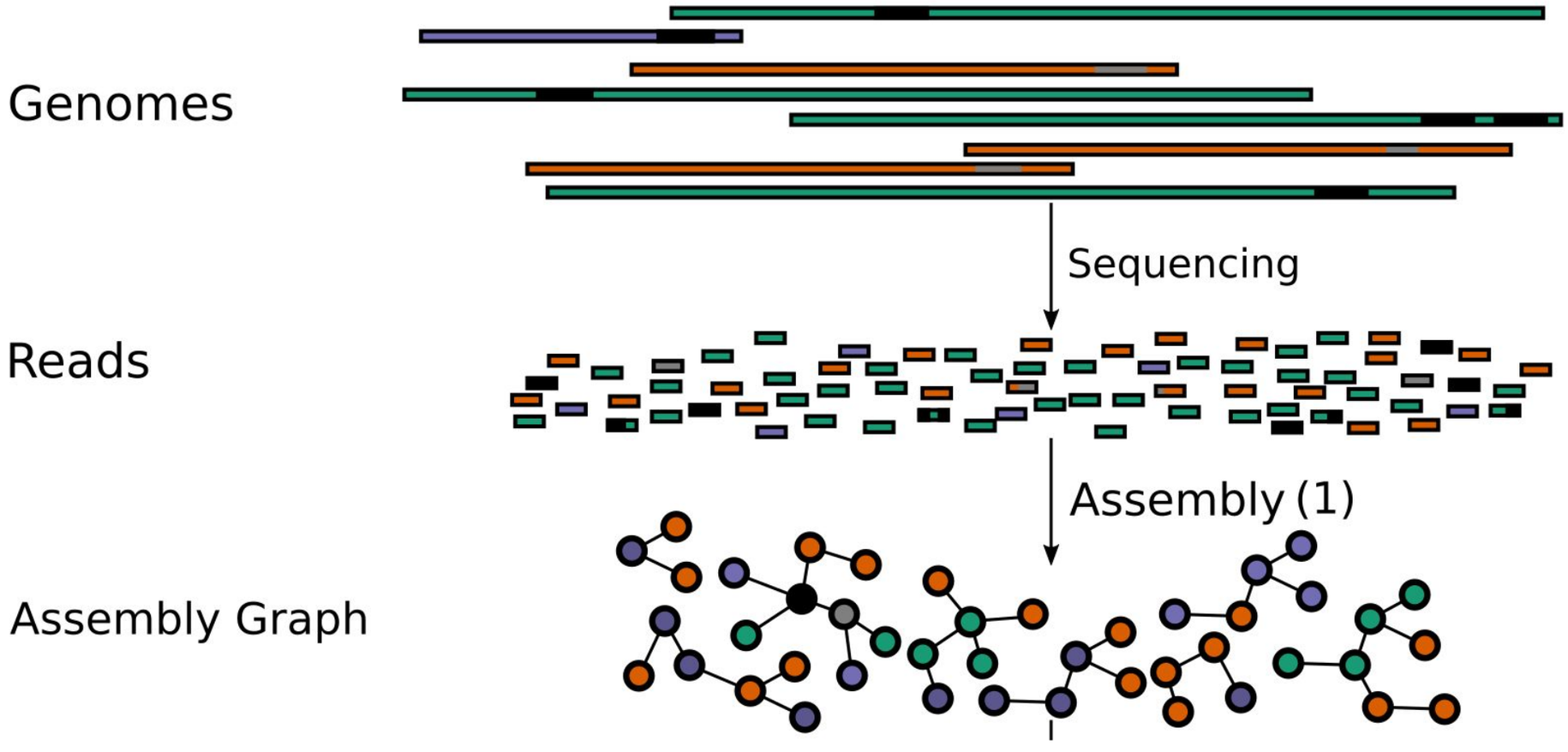
Aside: compacted de Bruijn graphs



(a)

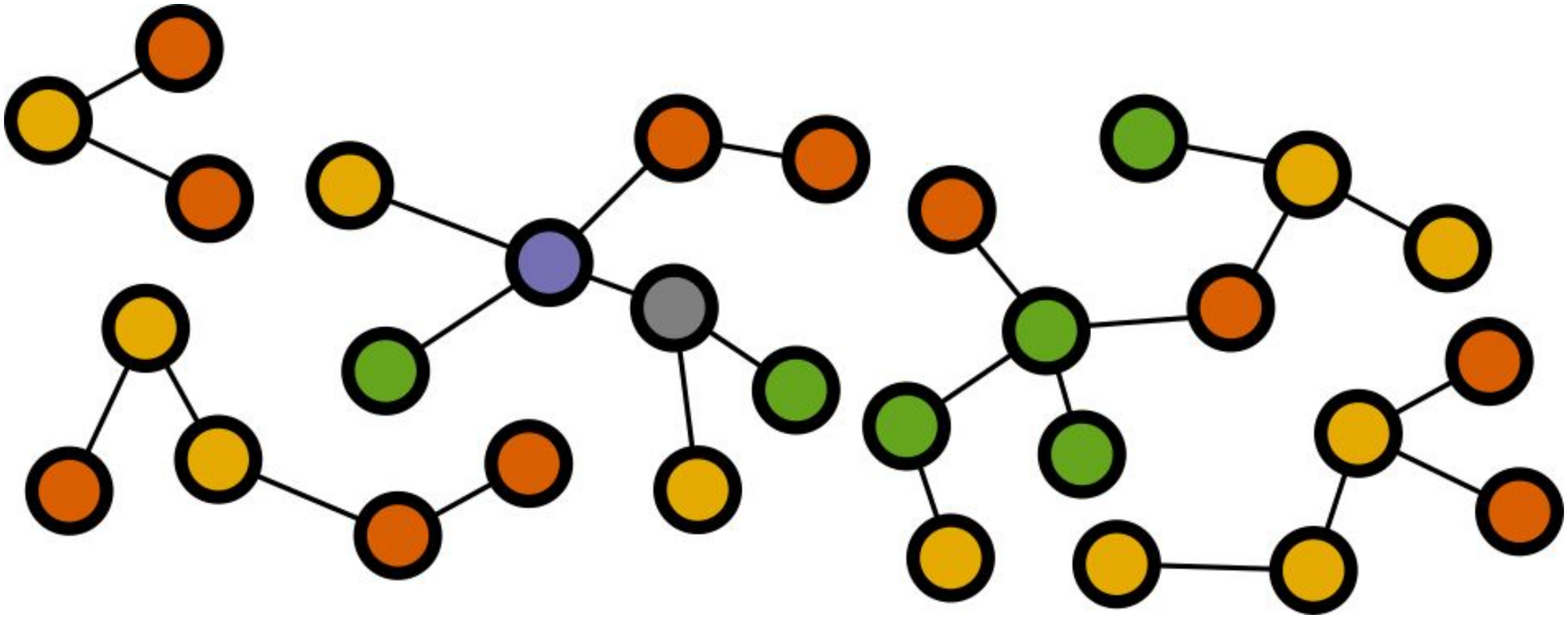
(b)

Metagenomic Assembly

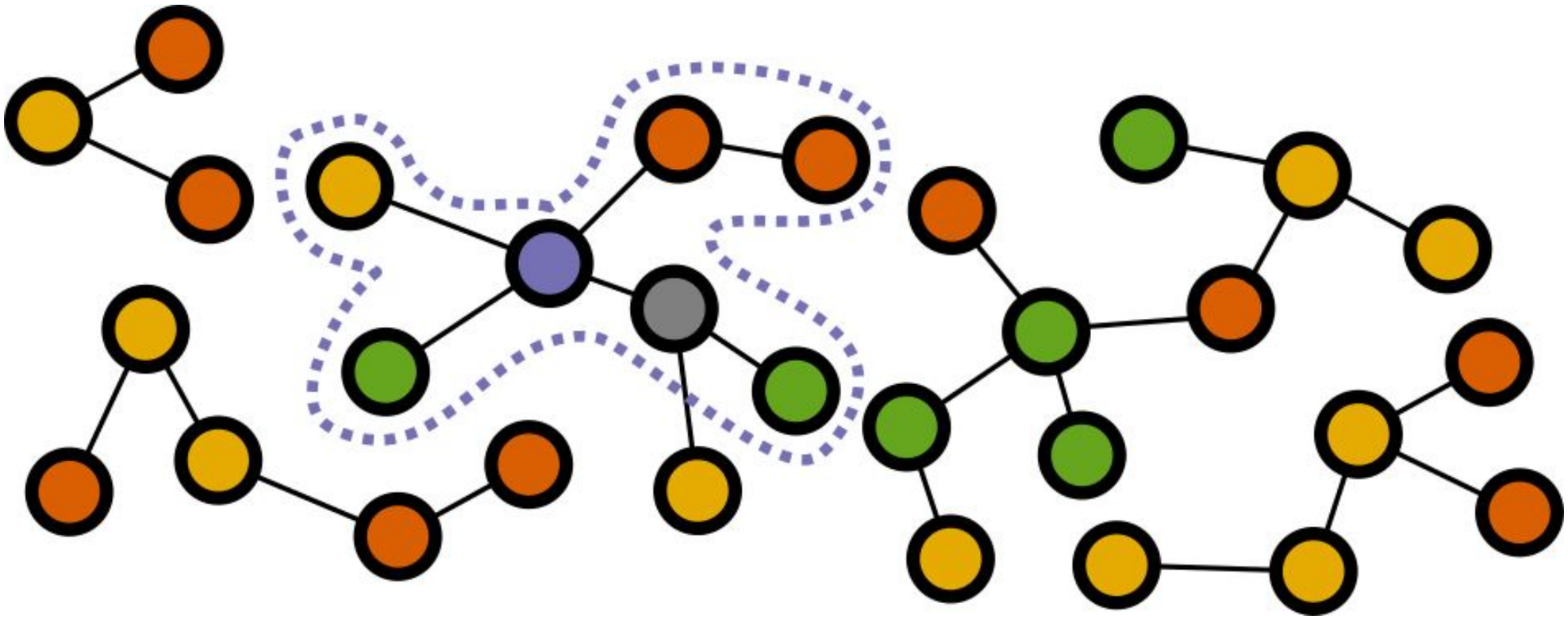


Wildly varied coverage!
How do we resolve repeats, closely related etc?

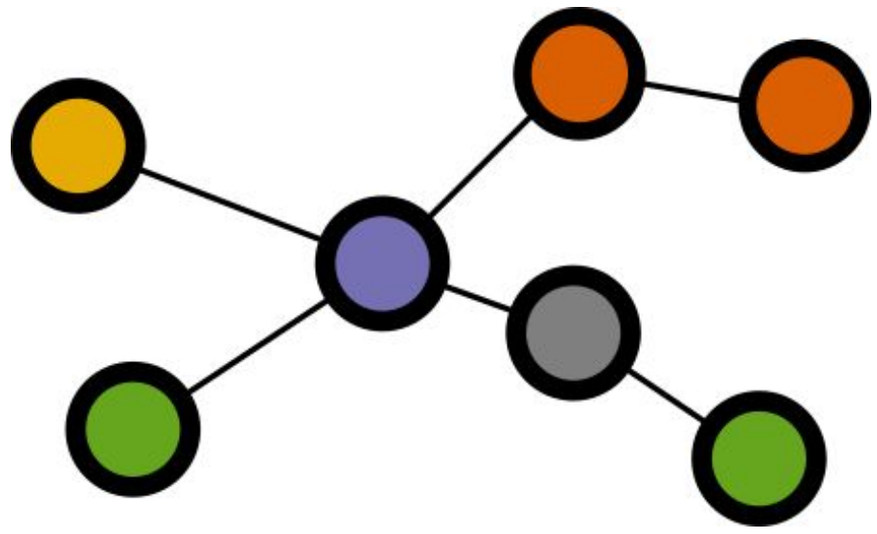
Metagenomic Assembly



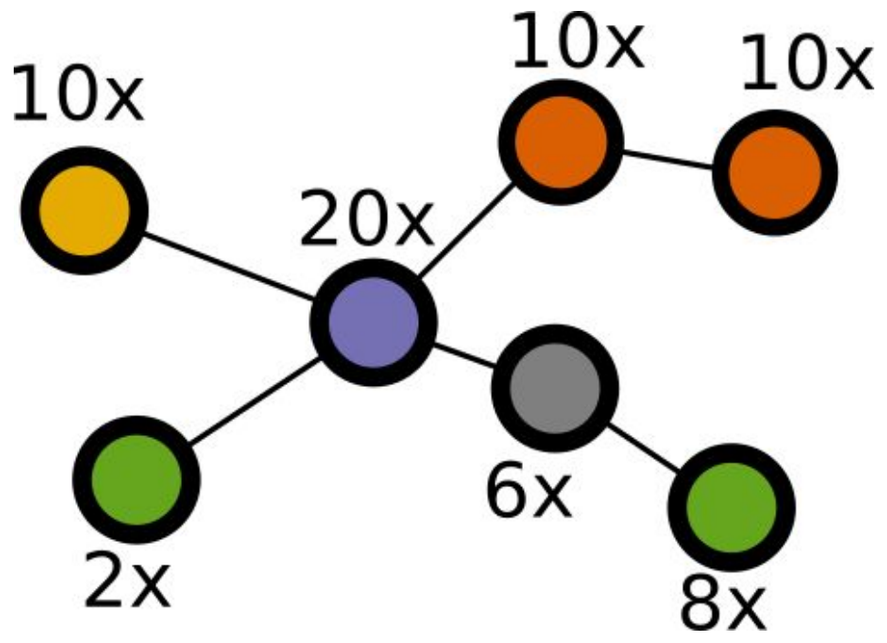
Metagenomic Assembly



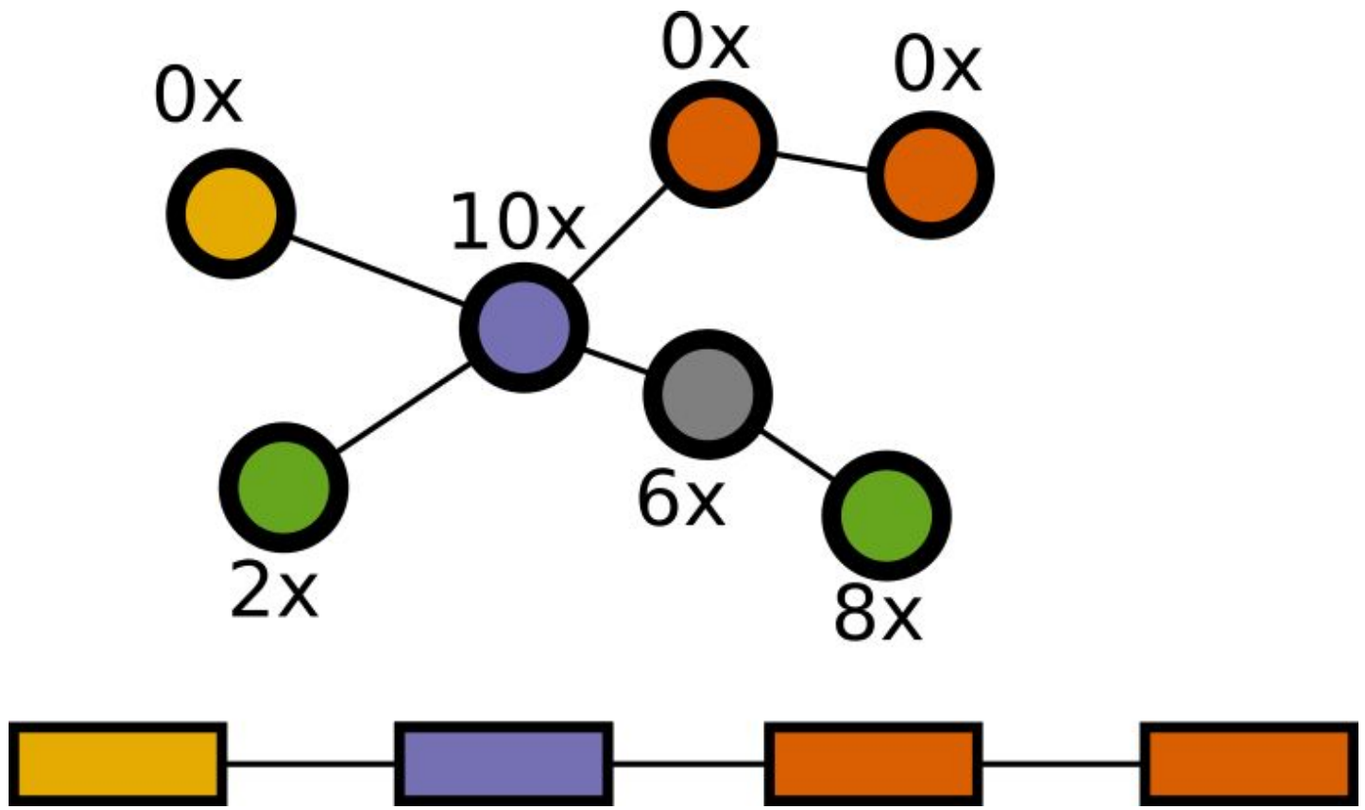
Metagenomic Assembly



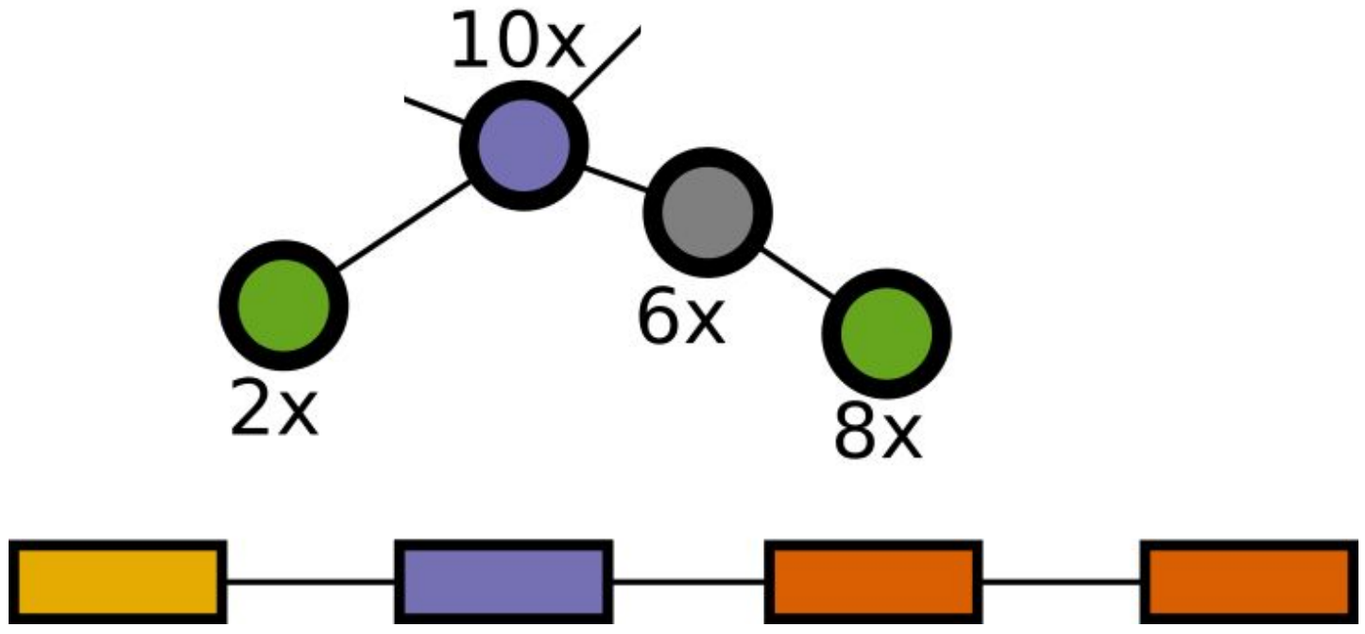
Metagenomic Assembly



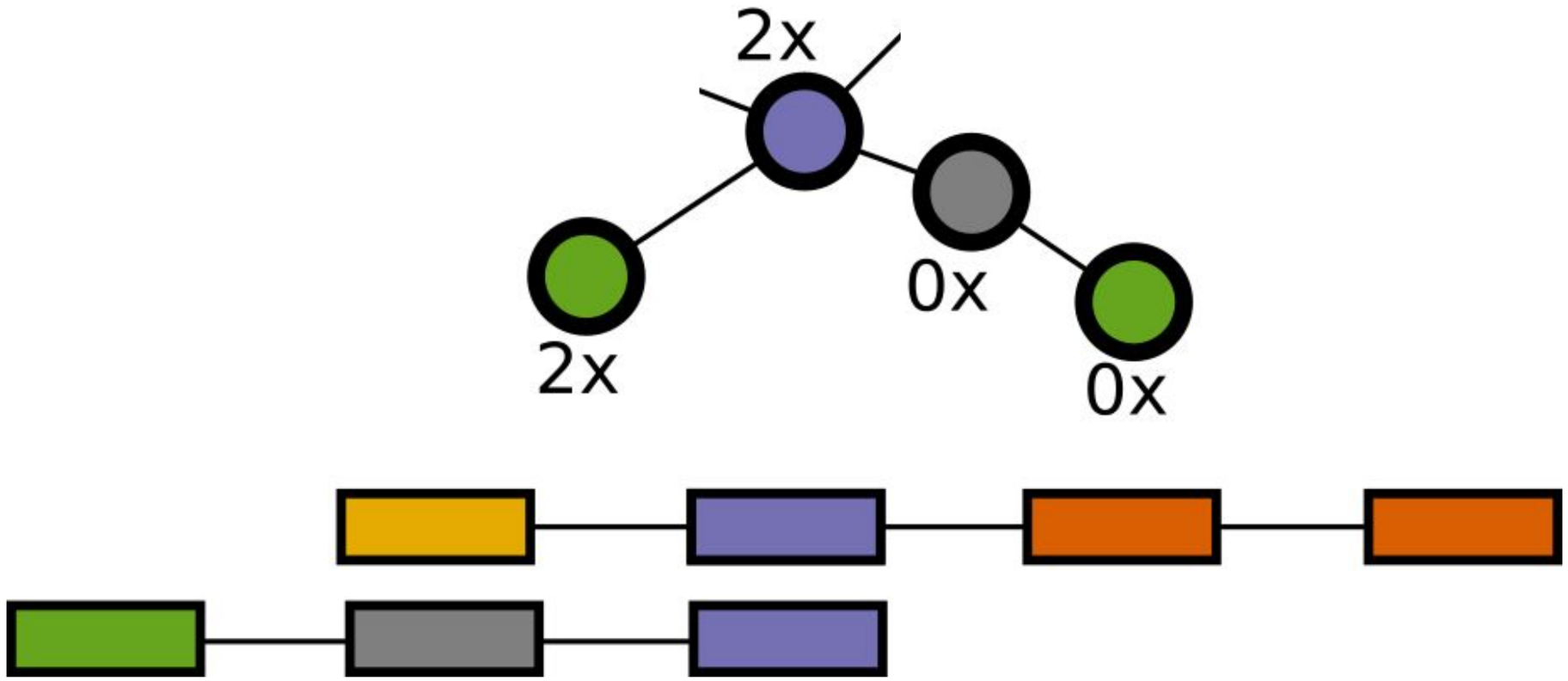
Metagenomic Assembly



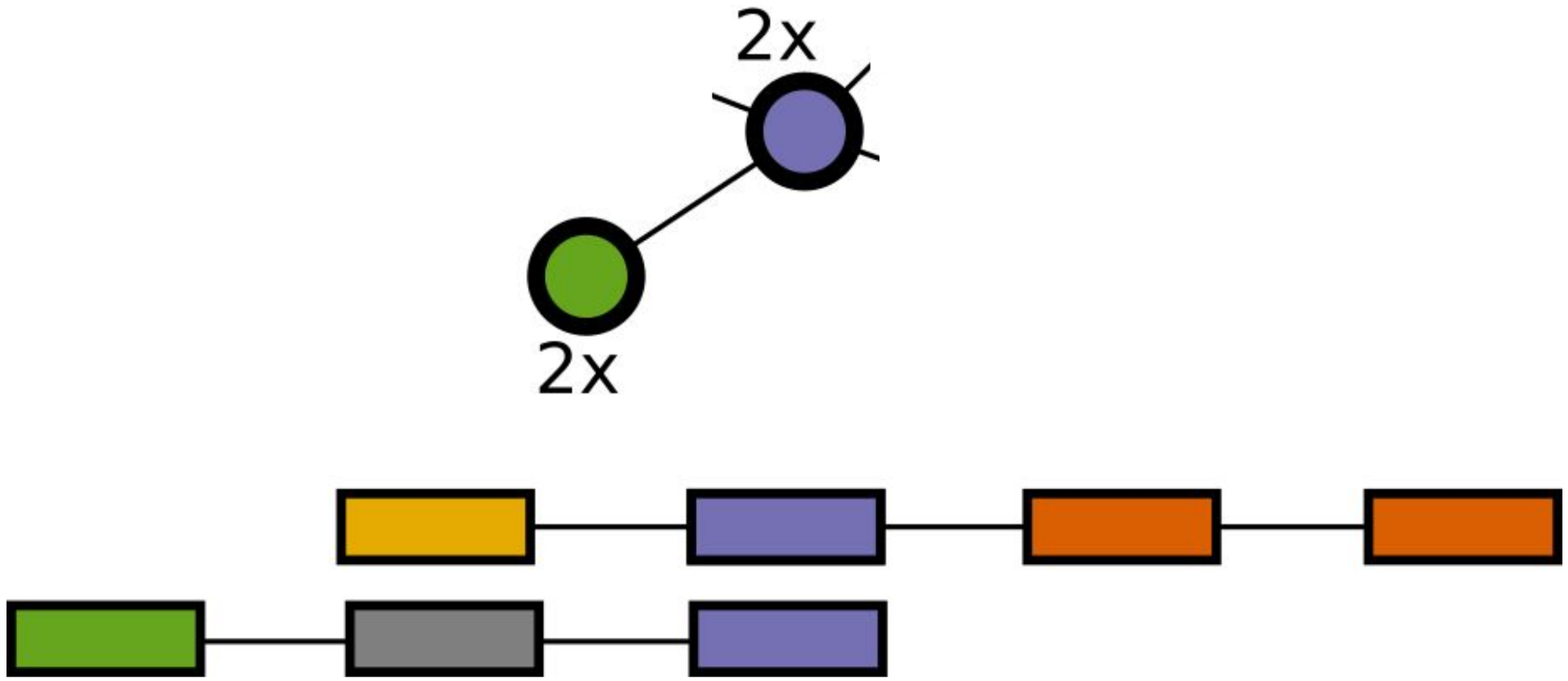
Metagenomic Assembly



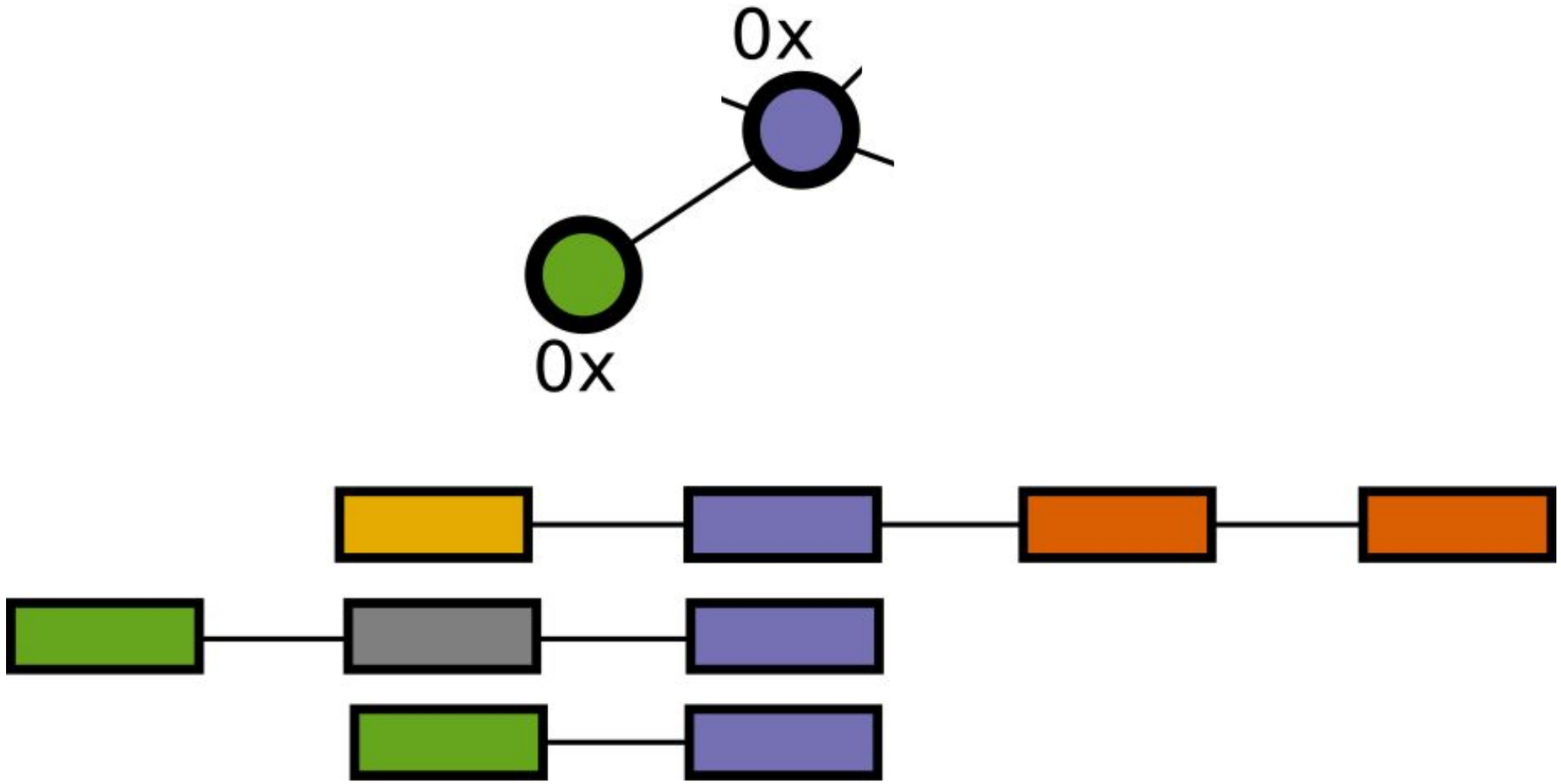
Metagenomic Assembly



Metagenomic Assembly

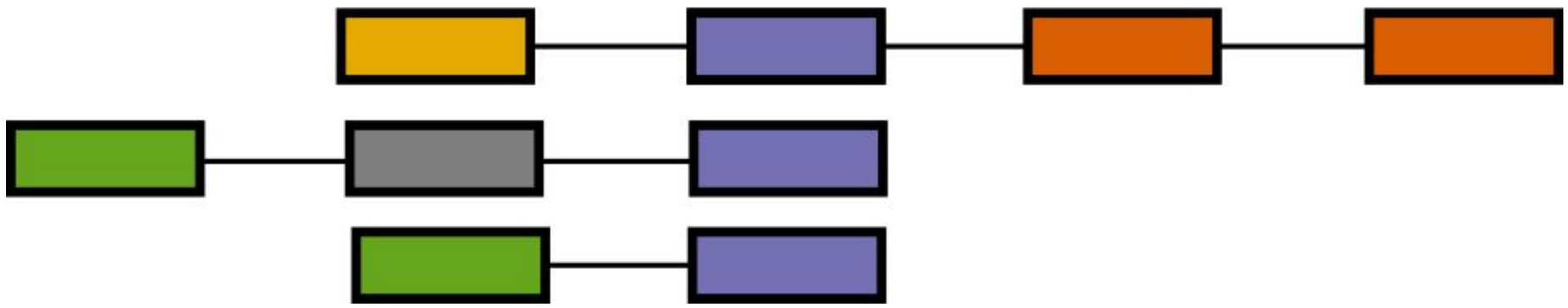


Metagenomic Assembly

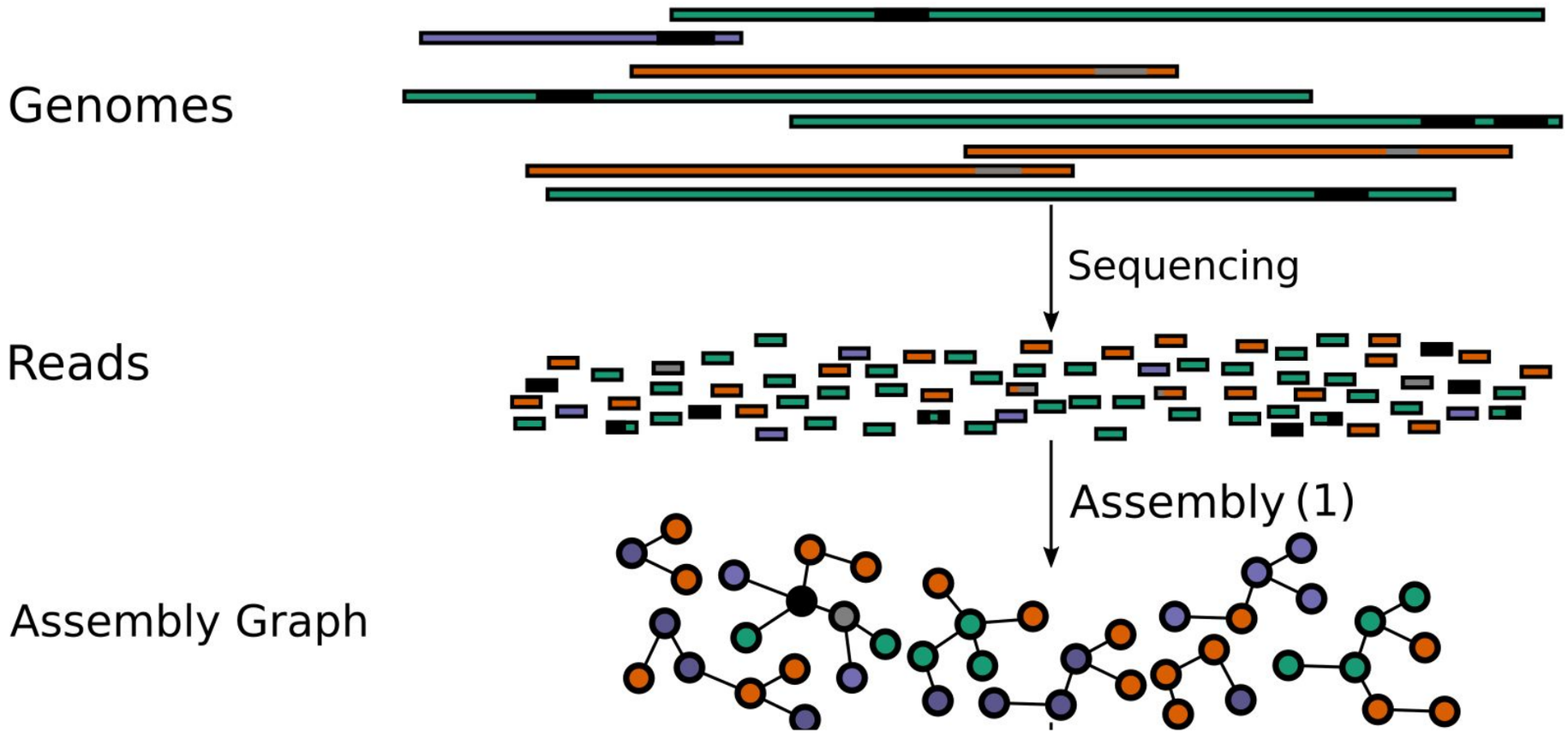


Metagenomic Assembly

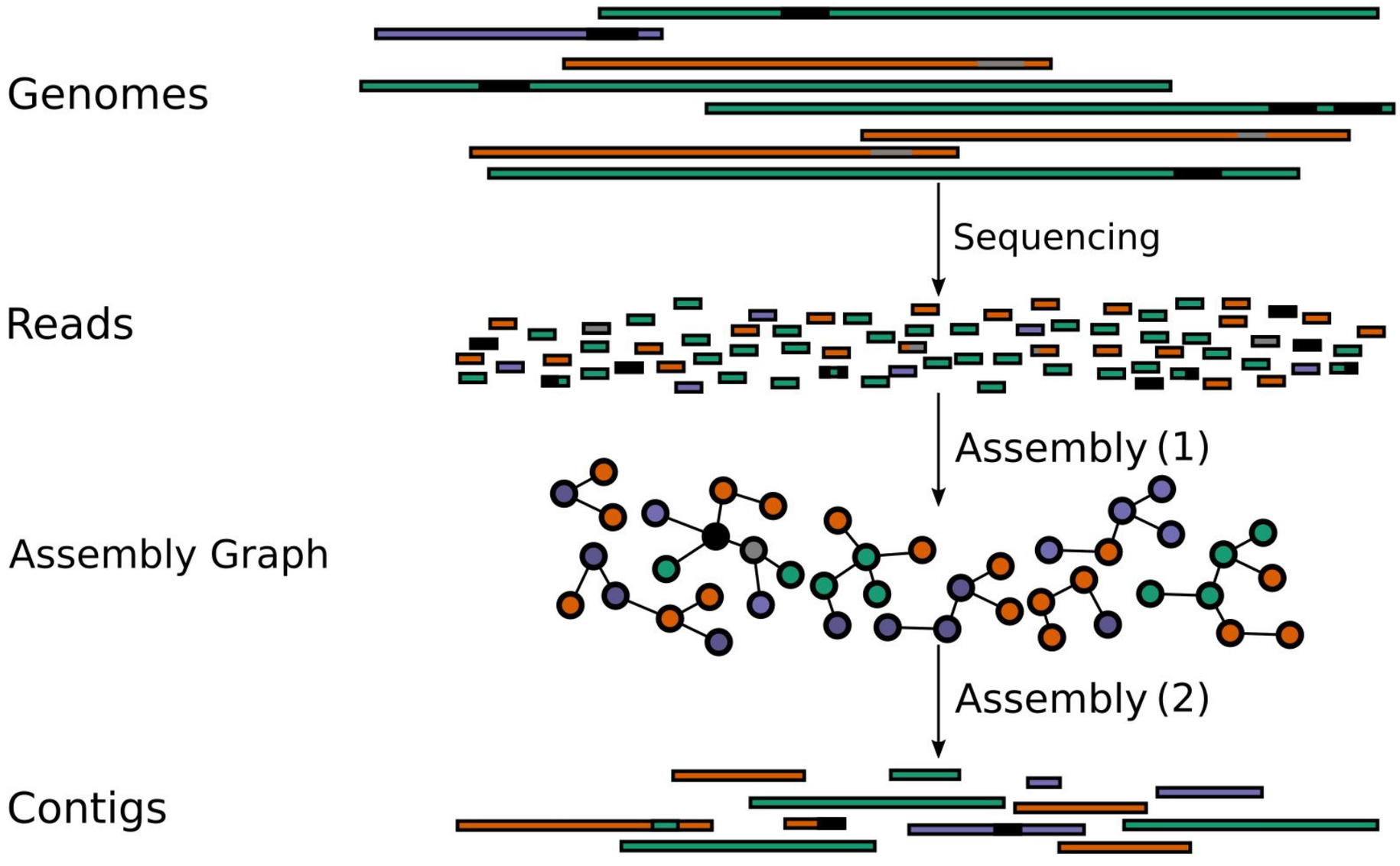
Coverage and lots of other tricks and heuristics!



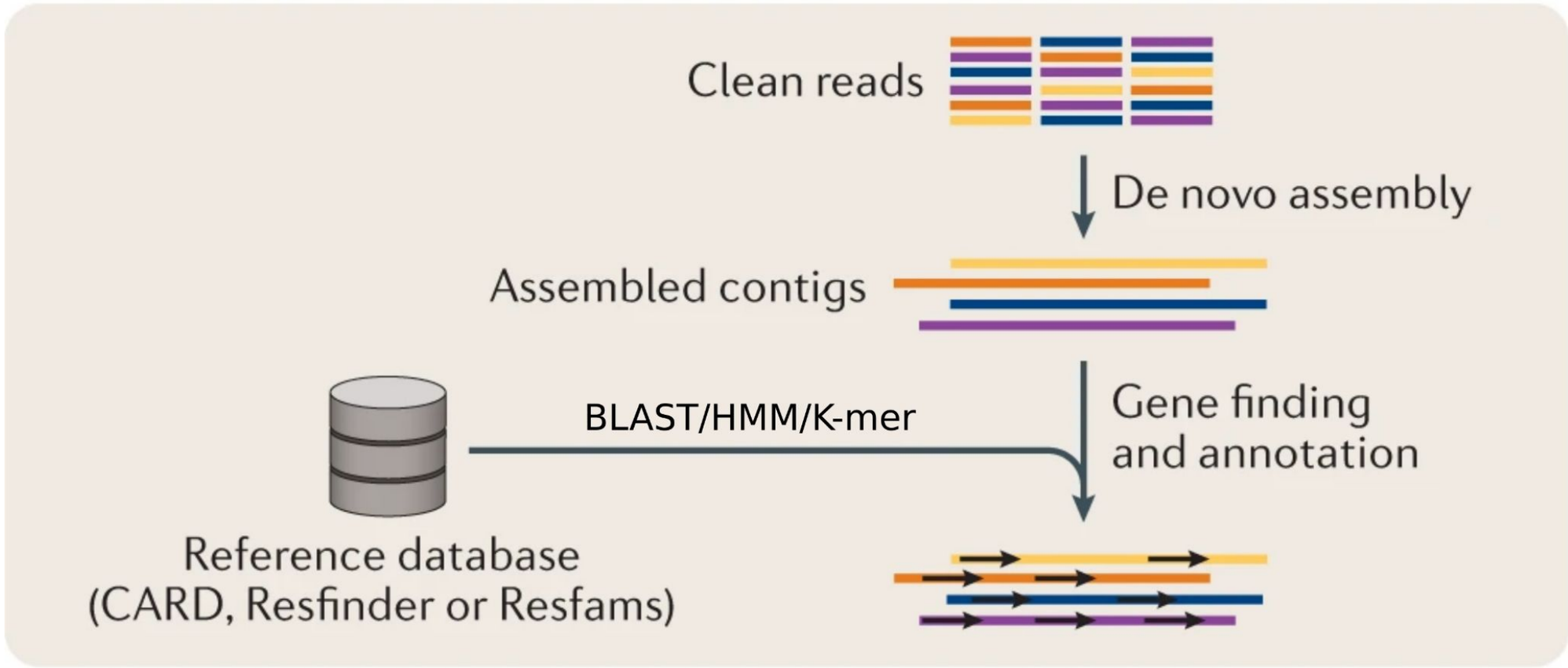
Metagenomics



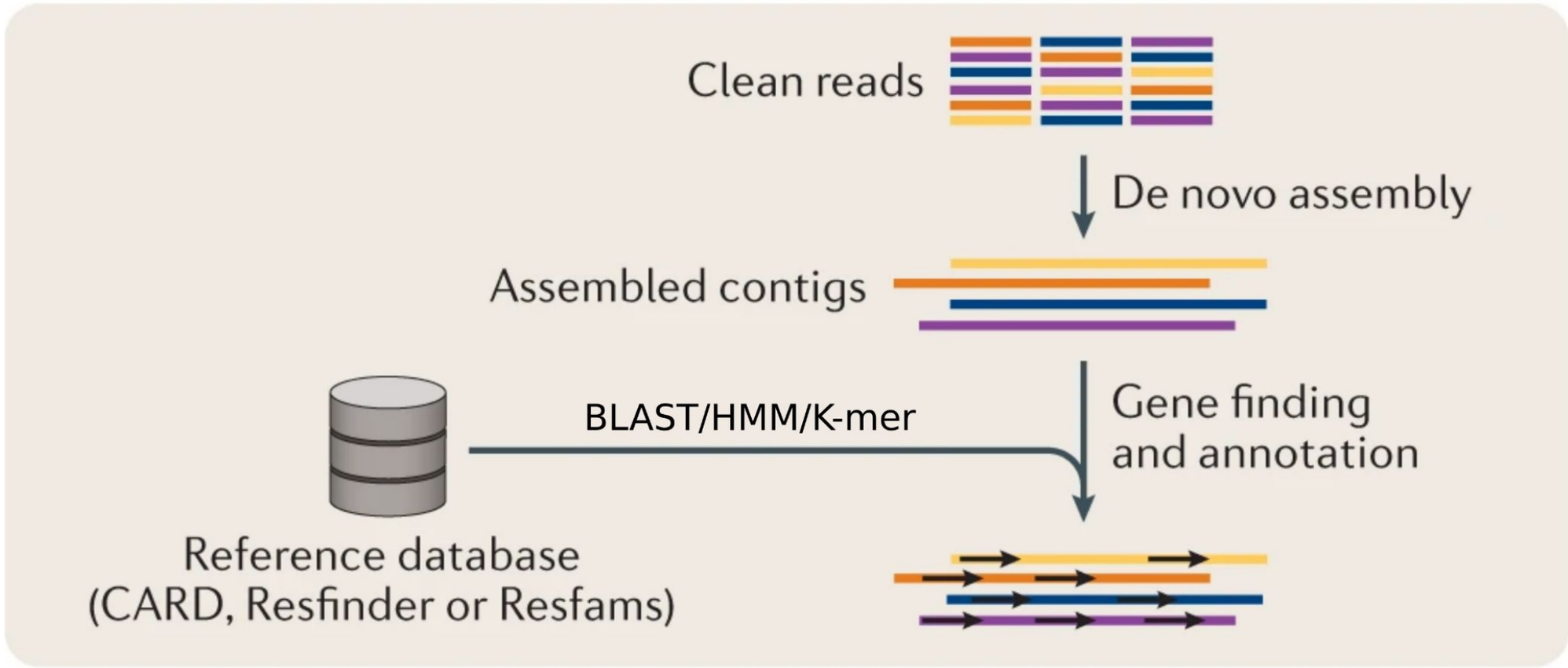
Metagenomics



Contig-based Analyses

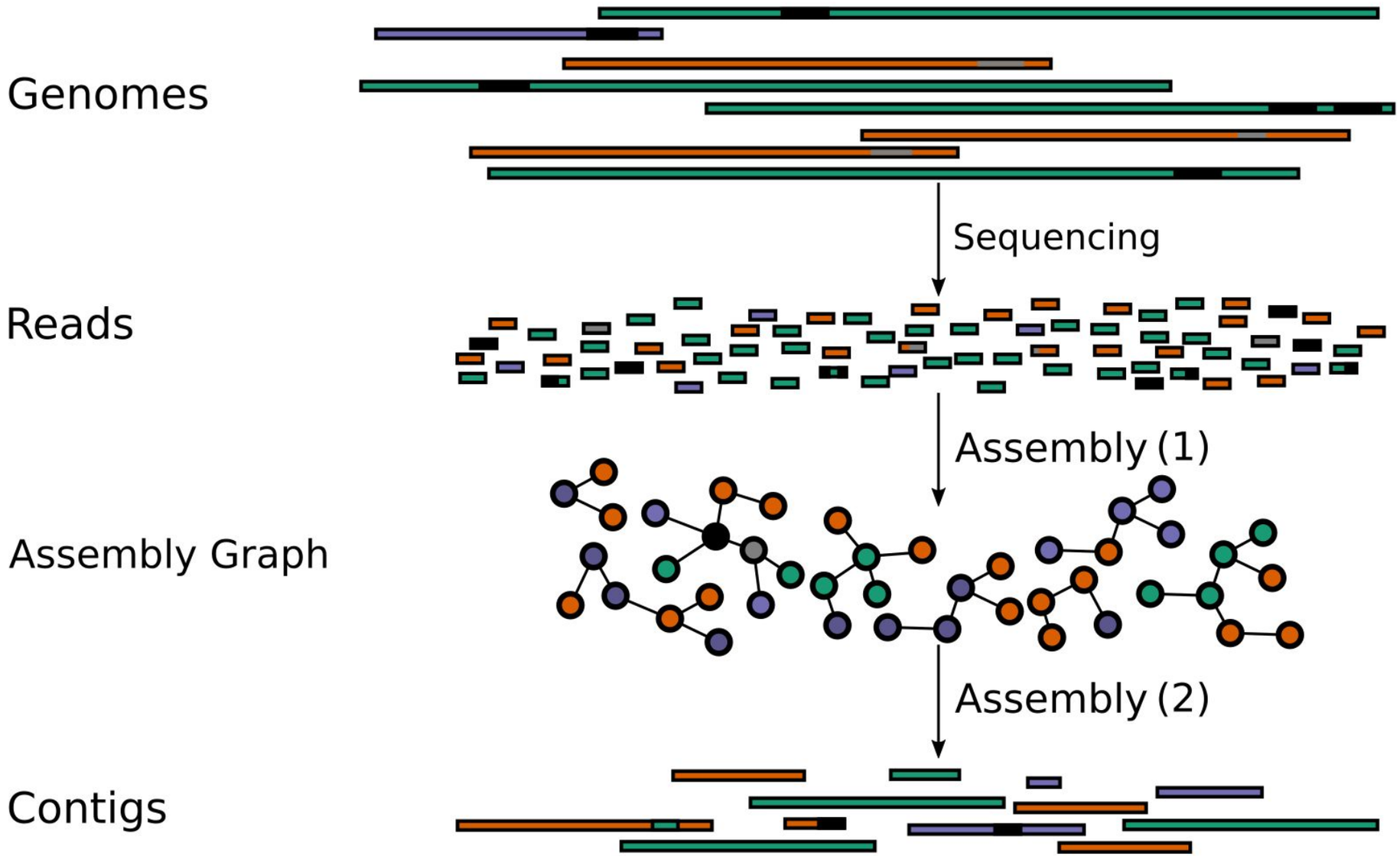


Contig-based Analyses

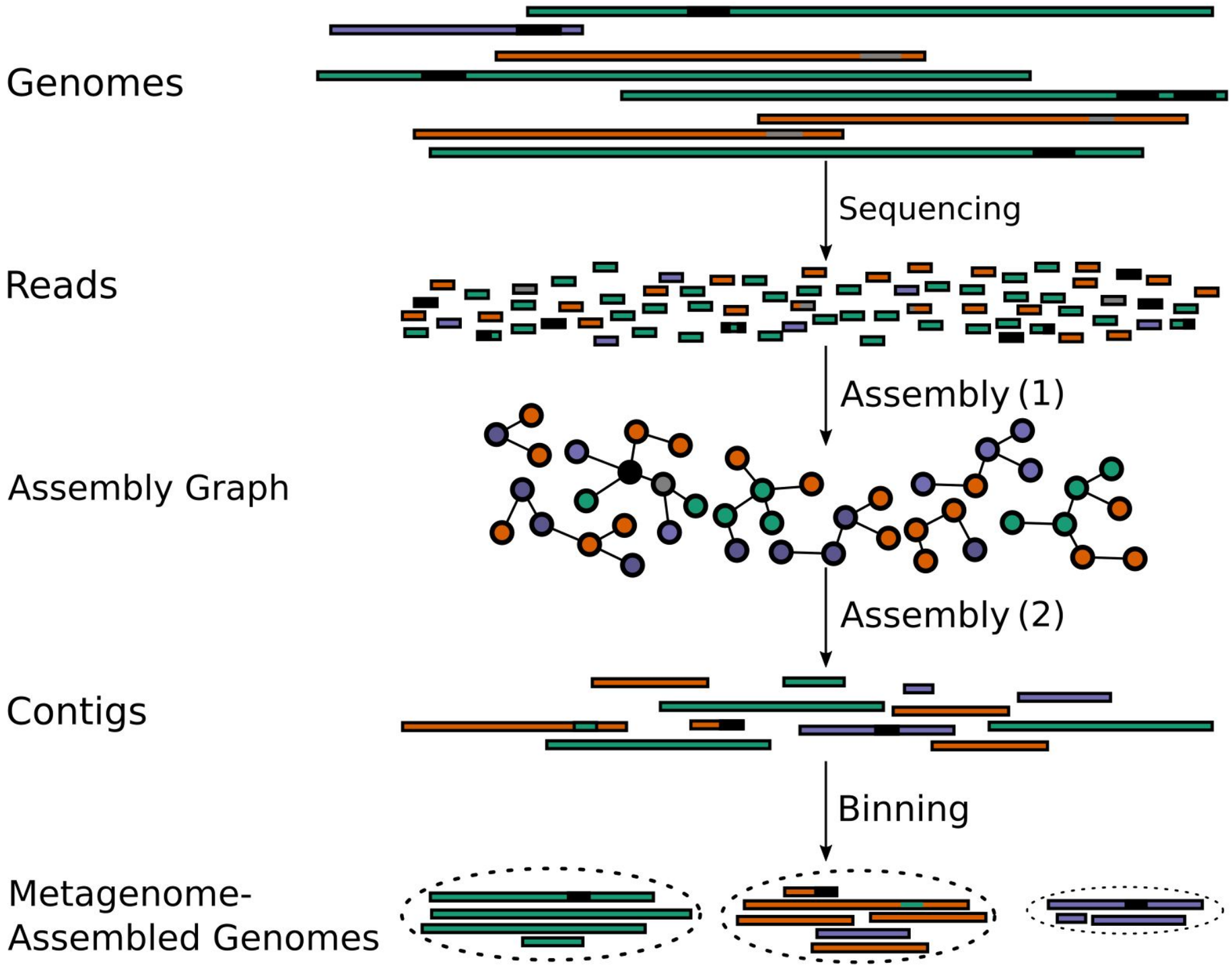


But - which genes came from which genome?

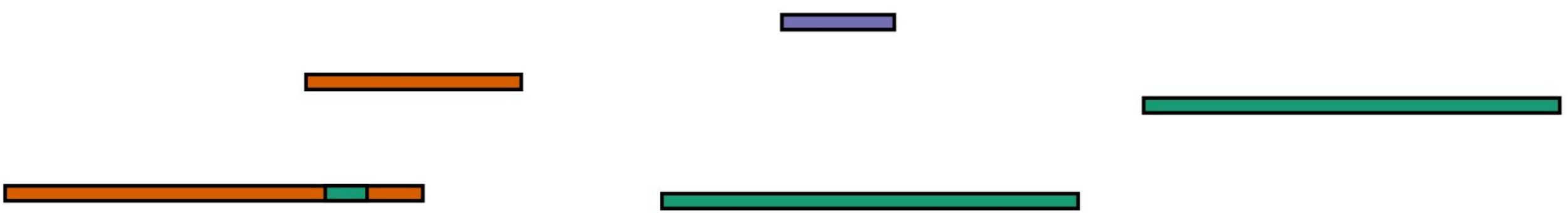
Metagenomics



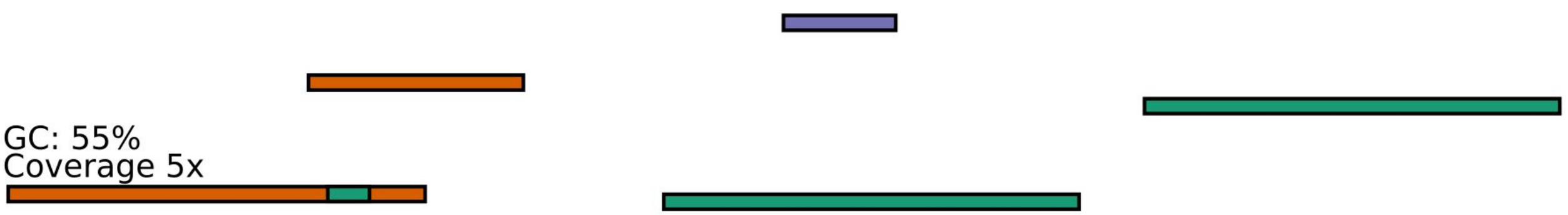
Metagenomics



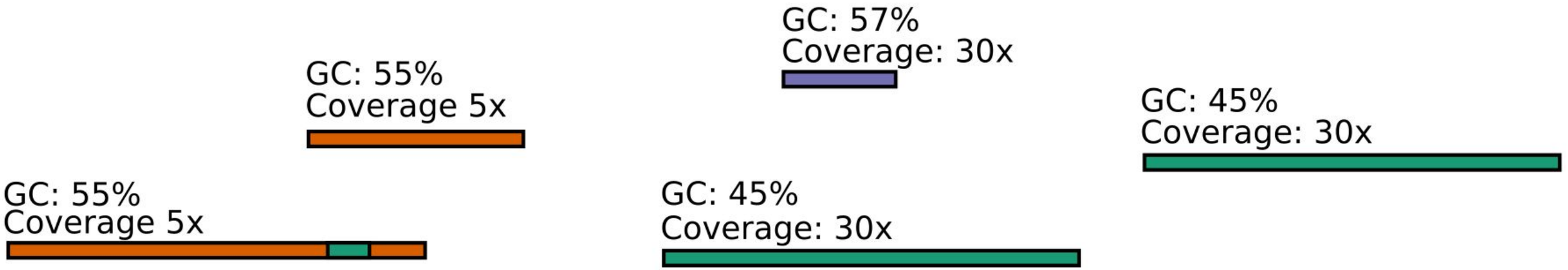
Metagenome Assembled Genome (MAG) Binning



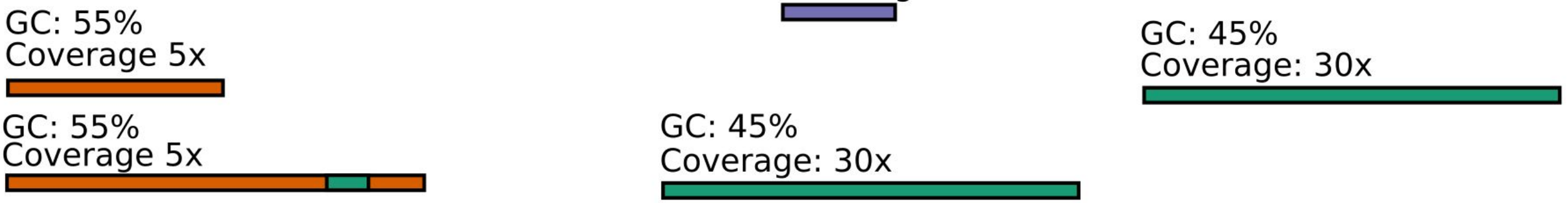
Metagenome Assembled Genome (MAG) Binning



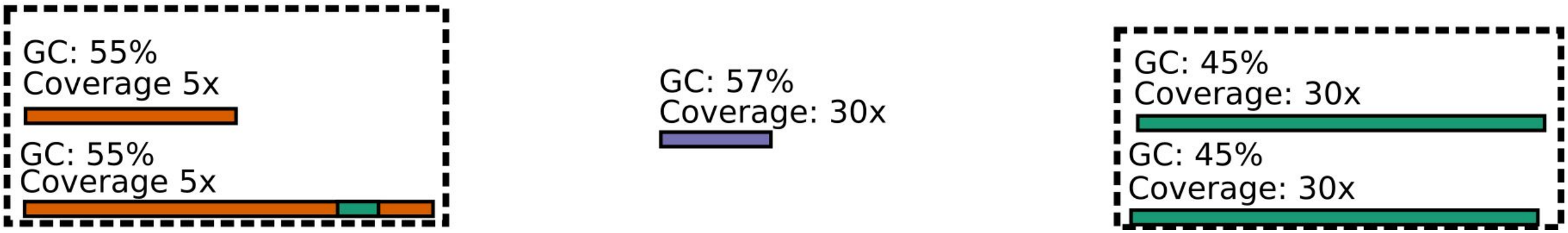
Metagenome Assembled Genome (MAG) Binning



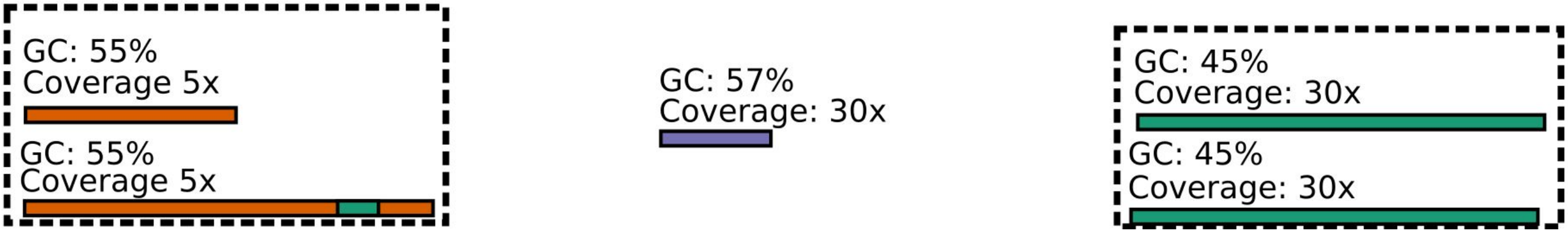
Metagenome Assembled Genome (MAG) Binning



Metagenome Assembled Genome (MAG) Binning

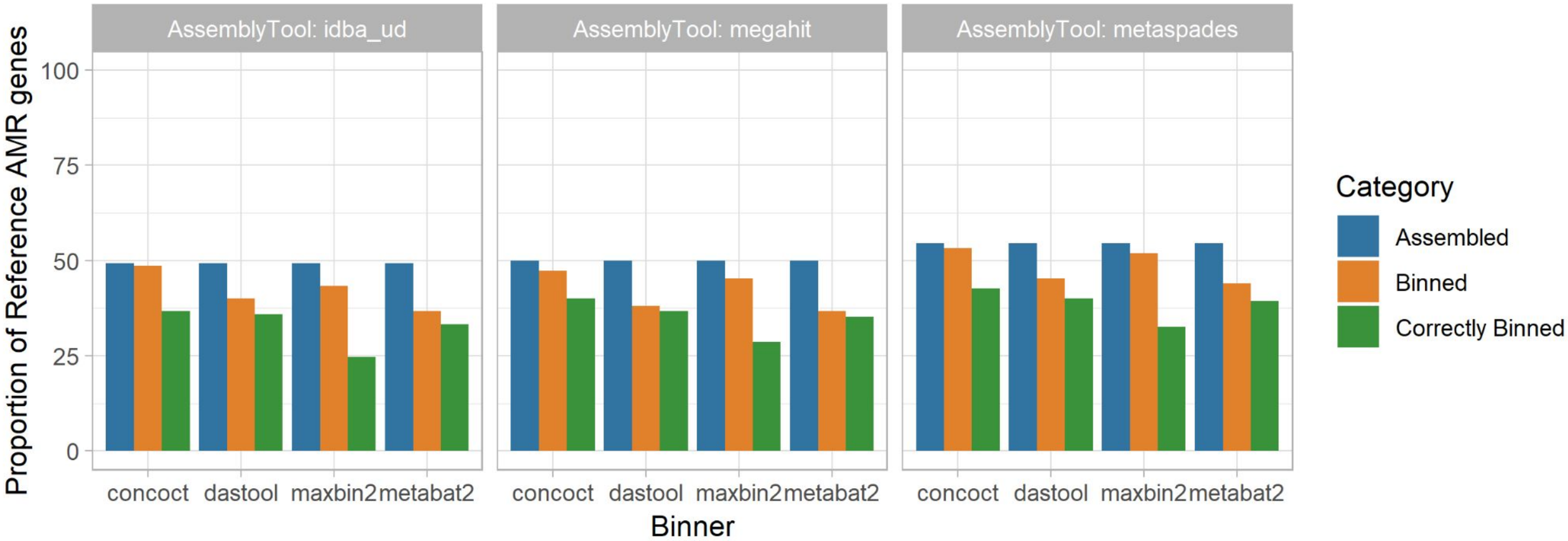


Metagenome Assembled Genome (MAG) Binning



Metabat and many other MAG binning tools cluster contigs using **composition** and **coverage** data

Lots of context but terrible sensitivity



Overview

- de Bruijn Graphs are key to modern assembly methods (scales in minimum of depth or genome size)
- Error correction is vital to effective dBG methods (remove low abundance k-mers)
- Scaffolding/paired-end data important for usable short-read assemblies.

- Read-based metagenomics maximise sensitivity but lack precision and context
- Coverage data is key to assembly but metagenomic assembly is hard and fragmented
- Coverage and composition can be used to group contigs into MAGs
- Lots of data but low sensitivity