Assembly in Practice: Part 2: DBG

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

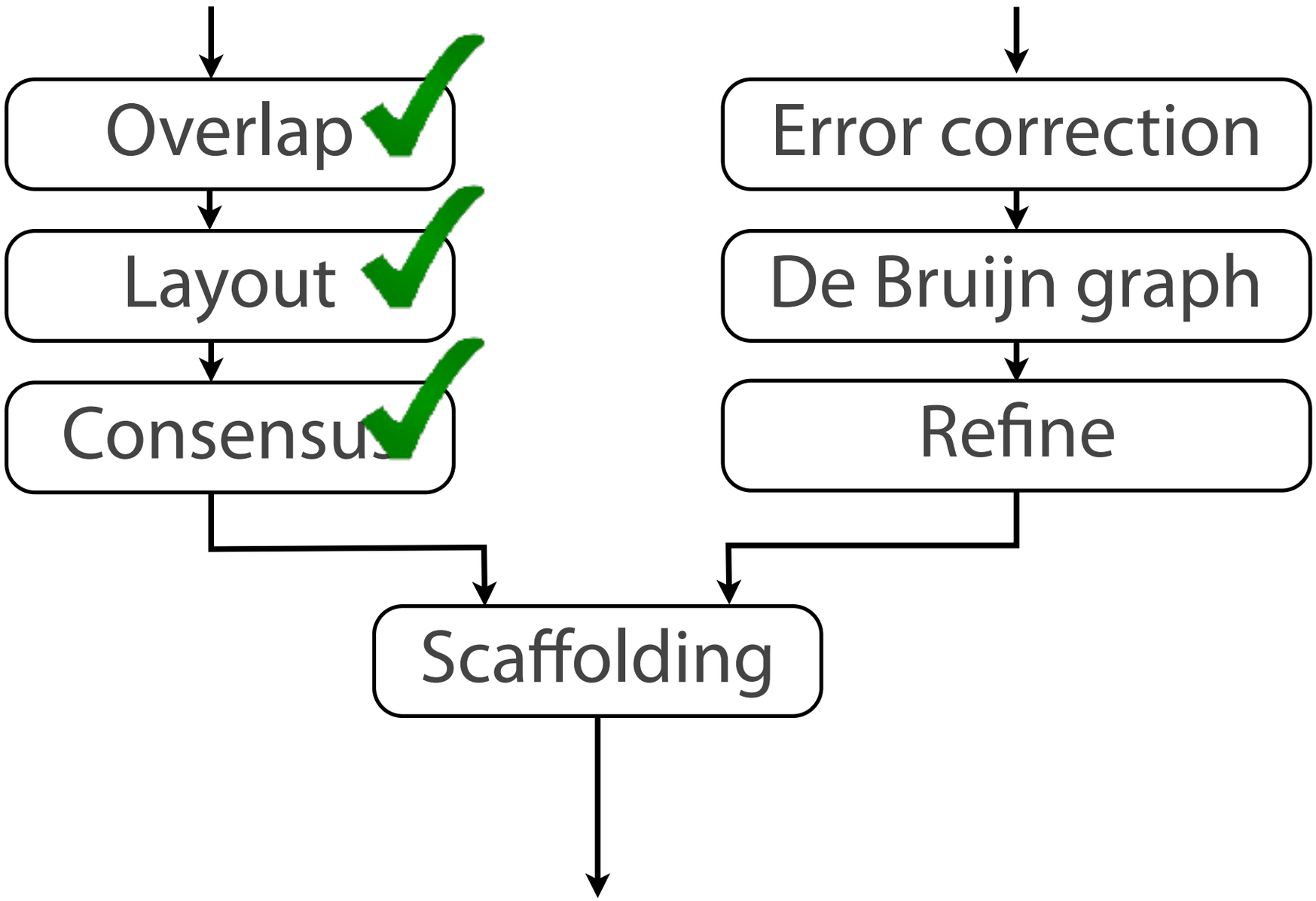
Department of Computer Science

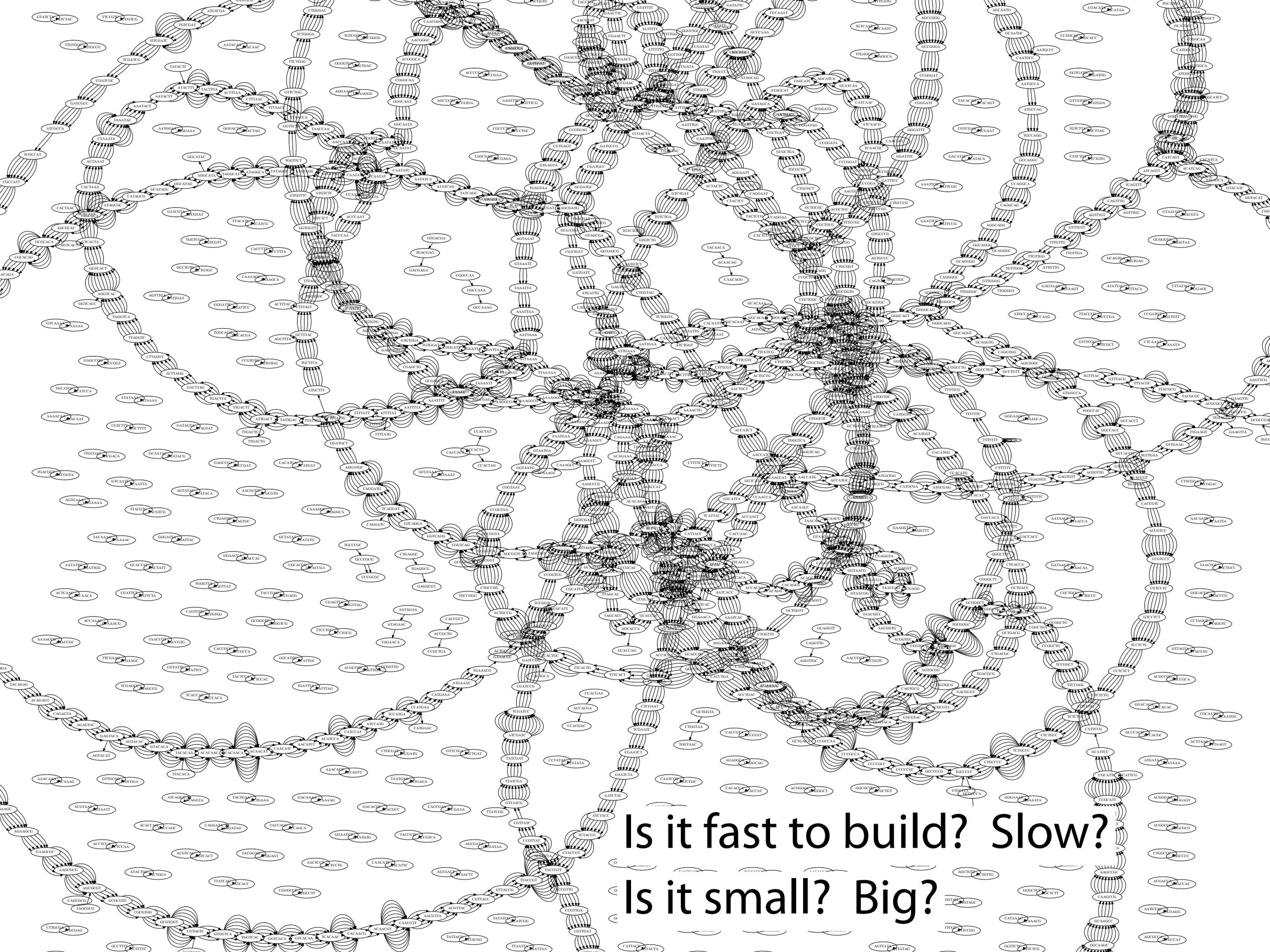
Please sign guestbook (www.langmead-lab.org/teaching-materials) to tell me briefly how you are using the slides. For original Keynote files, email me (ben.langmead@gmail.com).

Assembly alternatives

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

Alternative 2: De Bruijn graph (DBG) assembly




Is it fast to build? Slow?
Is it small? Big?

De Bruijn graph

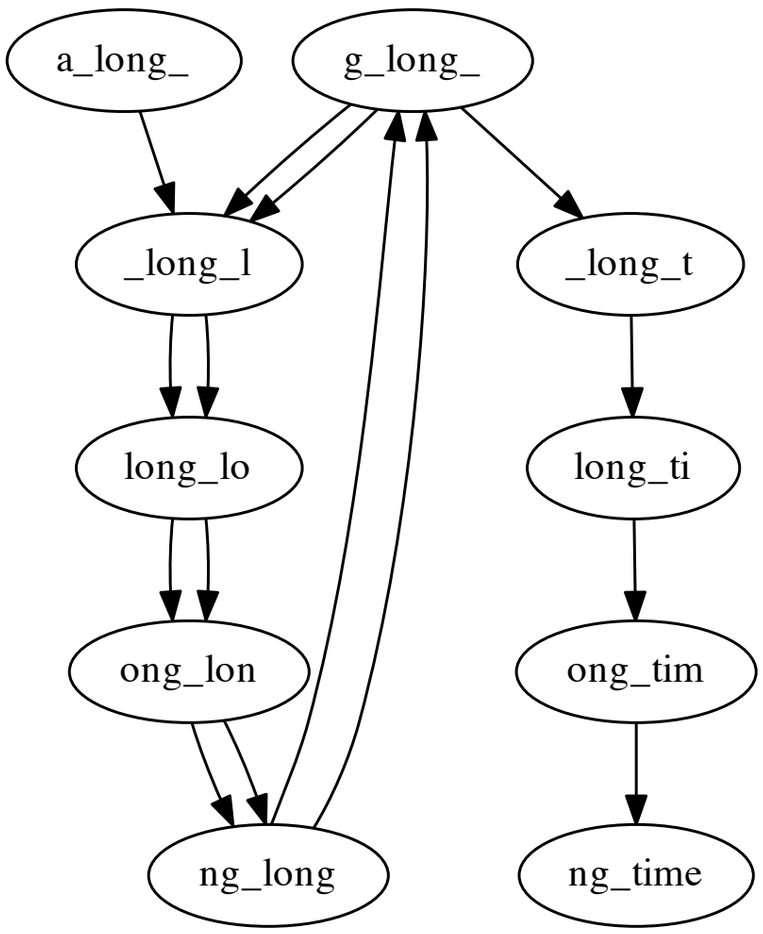
Pick $k = 8$

Genome: a_long_long_long_time

Reads: a_long_long_long, ng_long_l, g_long_time

k-mers:

```
a_long_l          ng_long_   g_long_t
 _long_lo          g_long_l   _long_ti
  long_lon                     long_tim
   ong_long                     ong_time
    ng_long_
     g_long_l
      _long_lo
       long_lon
        ong_long
```



For each read:

For each *k*-mer:

Add *k*-mer's left and right *k*-1-mers to graph if not there already. Draw an edge from left to right *k*-1-mer.

De Bruijn graph

d = 6 x 10^9 reads
n = 100 nt
} ≈ 1 week-long run of

Illumina HiSeq 2000

Sequencer outputs **d** reads of length **n**, total length N = **dn**.

To build graph: Pick k. Usually k is short relative to read length (k = 30 to 50 is common).

For each read:

For each k-mer:

Add k-mer's left and right k-1-mers to graph if not there already. Draw an edge from left to right k-1-mer.

De Bruijn graph

d = 6 x 10^9 reads
n = 100 nt
} ≈ 1 week-long run of

Illumina HiSeq 2000

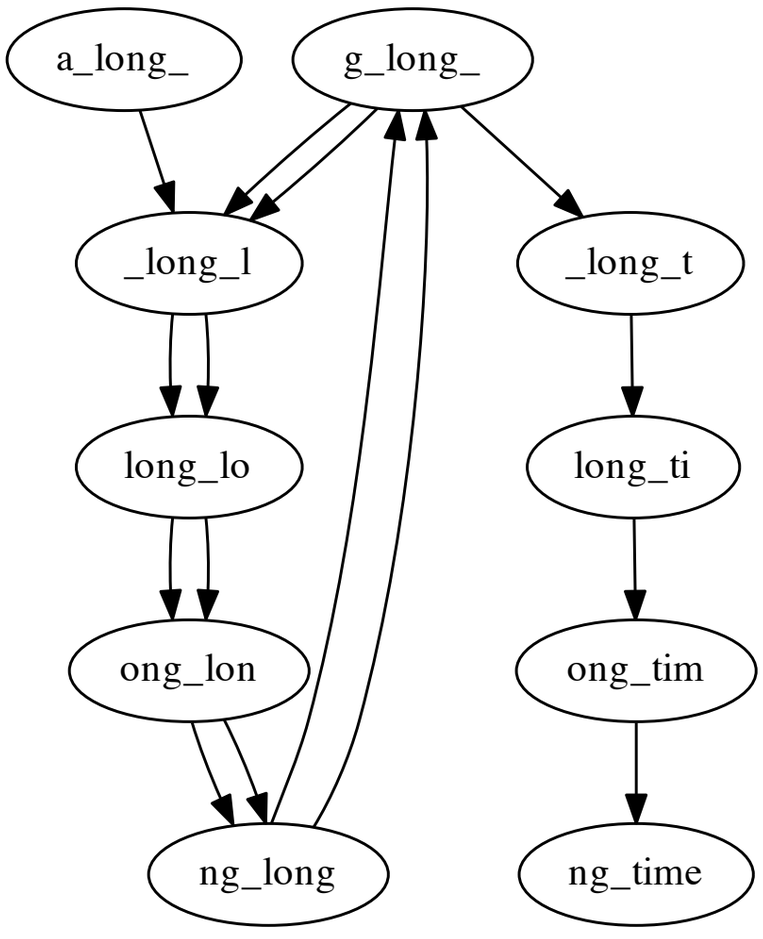
Sequencer outputs **d** reads of length **n**, total length N = **dn**.

To build graph: Pick k. Usually k is short relative to read length (k = 30 to 50 is common).

k-mers (edges): *O(N)*

nodes is at most 2 · (# edges); typically much smaller due to repeated k-1-mers *O(N)*

De Bruijn graph



How much work to build graph?

For each *k*-mer, add 1 edge and up to 2 nodes

Reasonable to say this is O(1) expected work

Say hash map holds nodes & edges

Say *k*-1-mers fit in O(1) machine words, and hashing O(1) words is O(1) work

Querying / adding a key is O(1) expected work

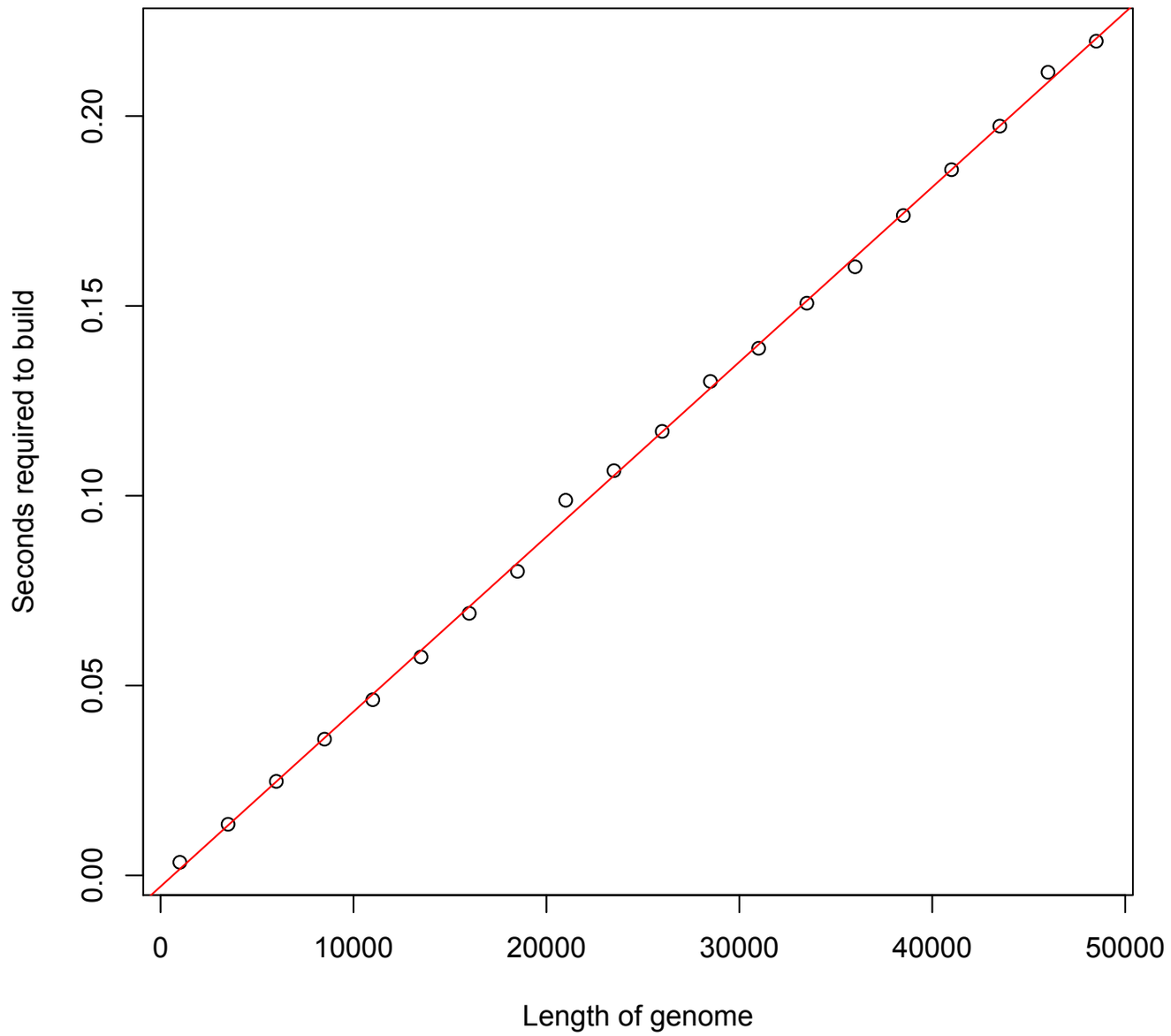
O(1) expected work for 1 *k*-mer, **O(*N*) overall**

De Bruijn graph

Timed De Bruijn graph construction applied to progressively longer prefixes of lambda phage genome, $k = 14$

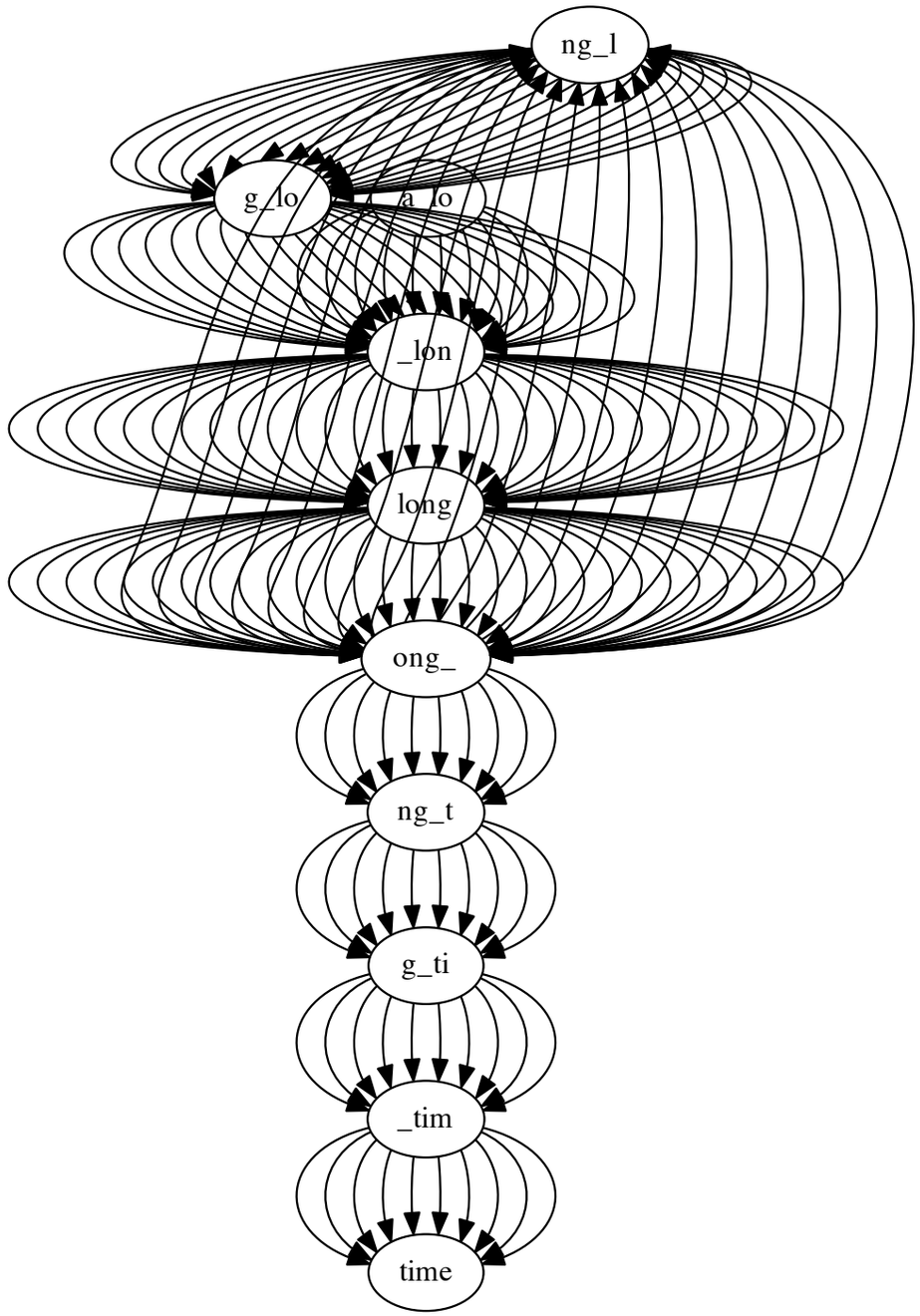
O(N) expectation works in practice

(in this case at least)

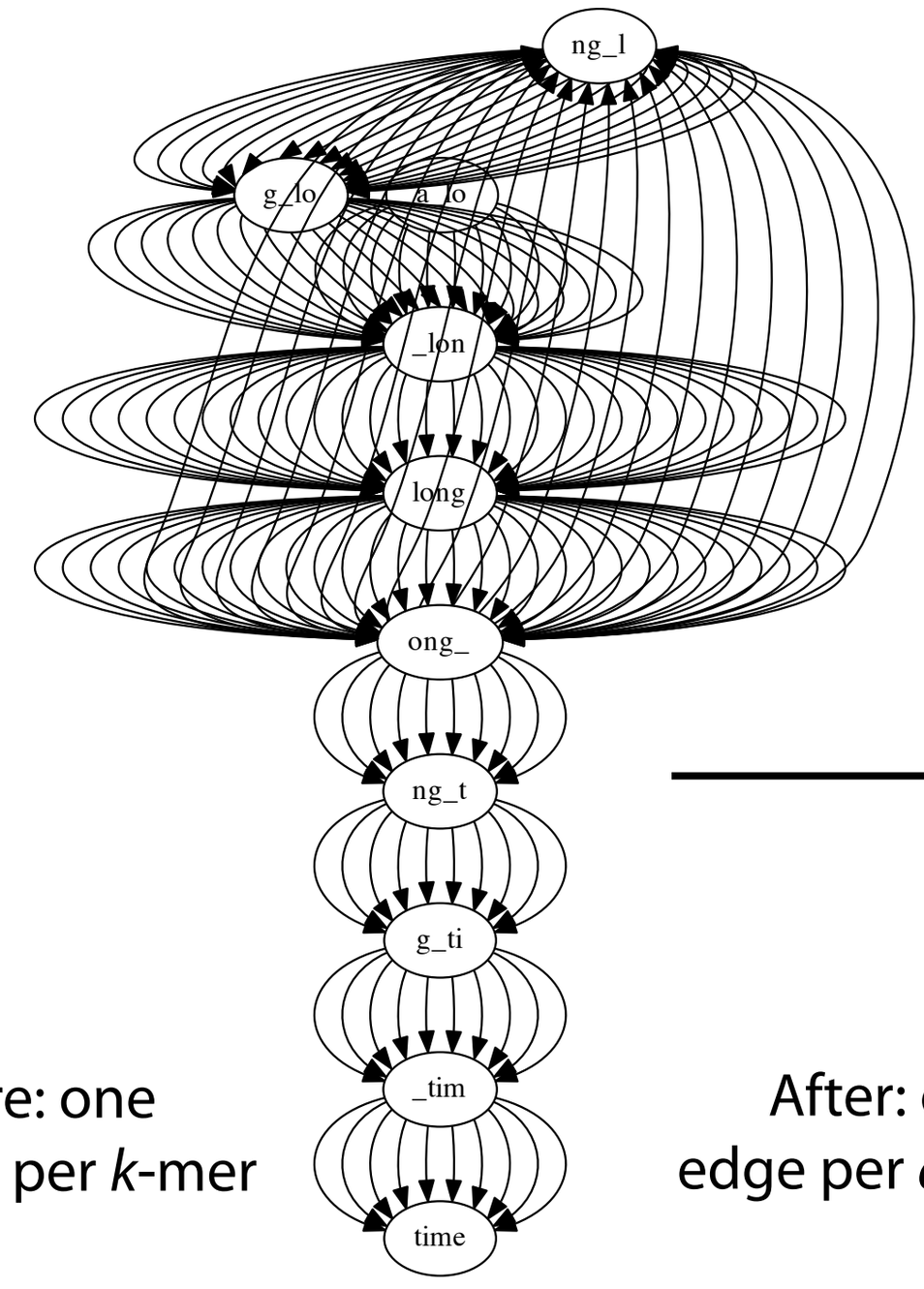


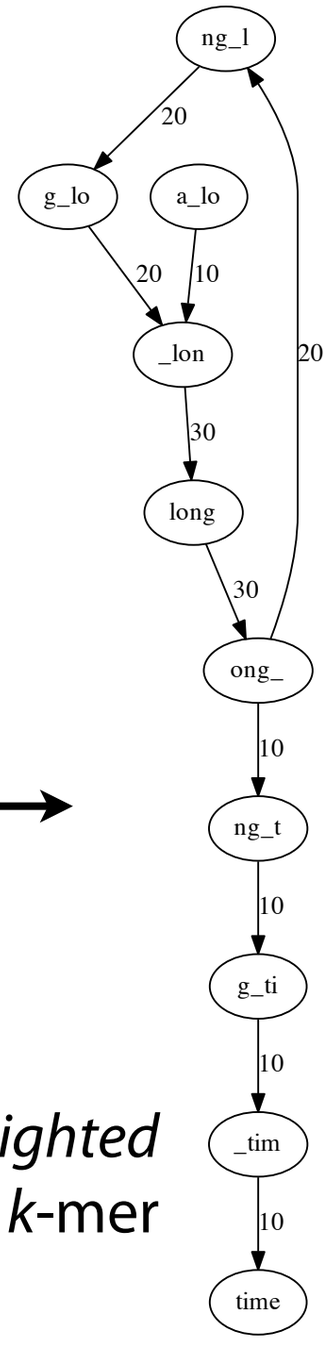
De Bruijn graph

In typical assembly projects,
average coverage is ~ 30 - 50



De Bruijn graph





Before: one edge per *k*-mer

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

De Bruijn graph

of nodes and edges both O(*N*)

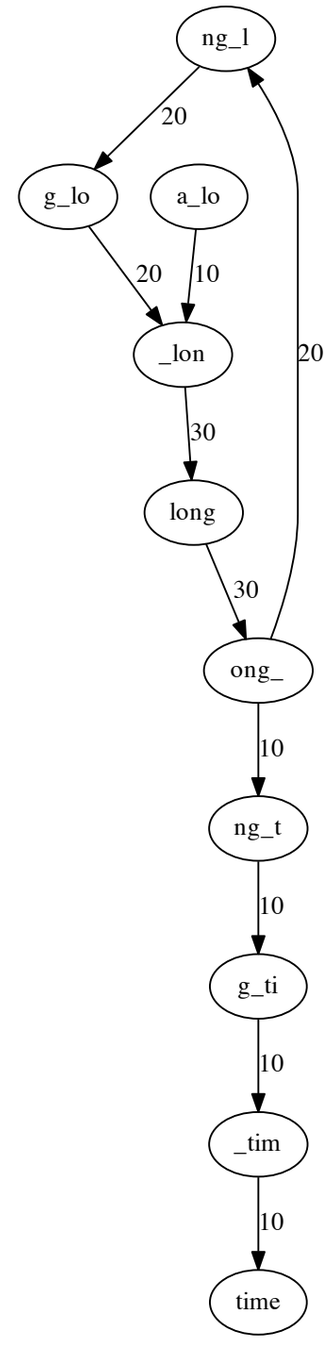
Say (a) reads are error-free, (b) we have one *weighted* edge for each *distinct k*-mer, and (c) length of genome is *G*

1 node per distinct *k*-1-mer, 1 edge per distinct *k*-mer

Can't have more distinct *k*-mers than *k*-mers in the genome; likewise for *k*-1-mers

So # of nodes and edges are both O(*G*)

Combine with the O(*N*) bound and the # of nodes and edges are both O(min(*N*, *G*))



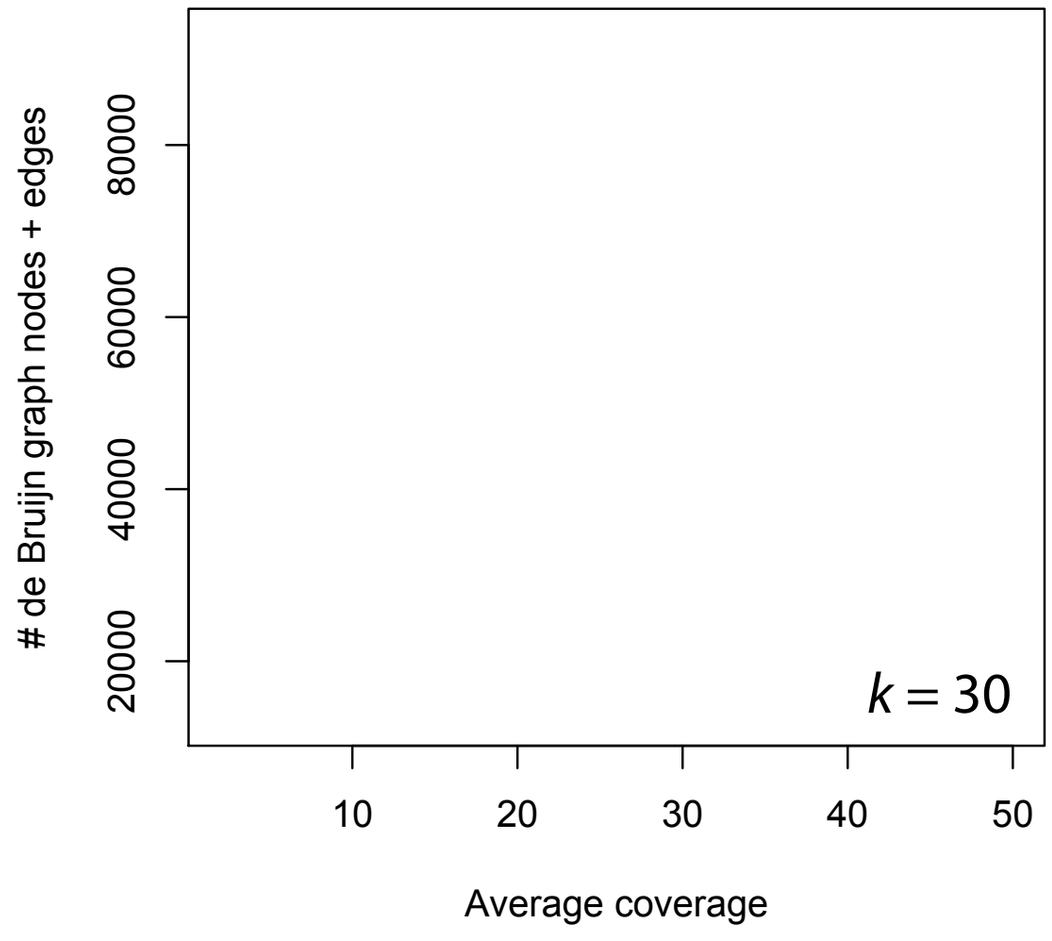
De Bruijn graph

At high coverage, O(min(*N*, *G*)) bound is advantageous

Genome: lambda phage (~48,500 bp)

Draw random *k*-mers until target average coverage (x axis) is reached

Build graph, sum # nodes and # edges (y axis)



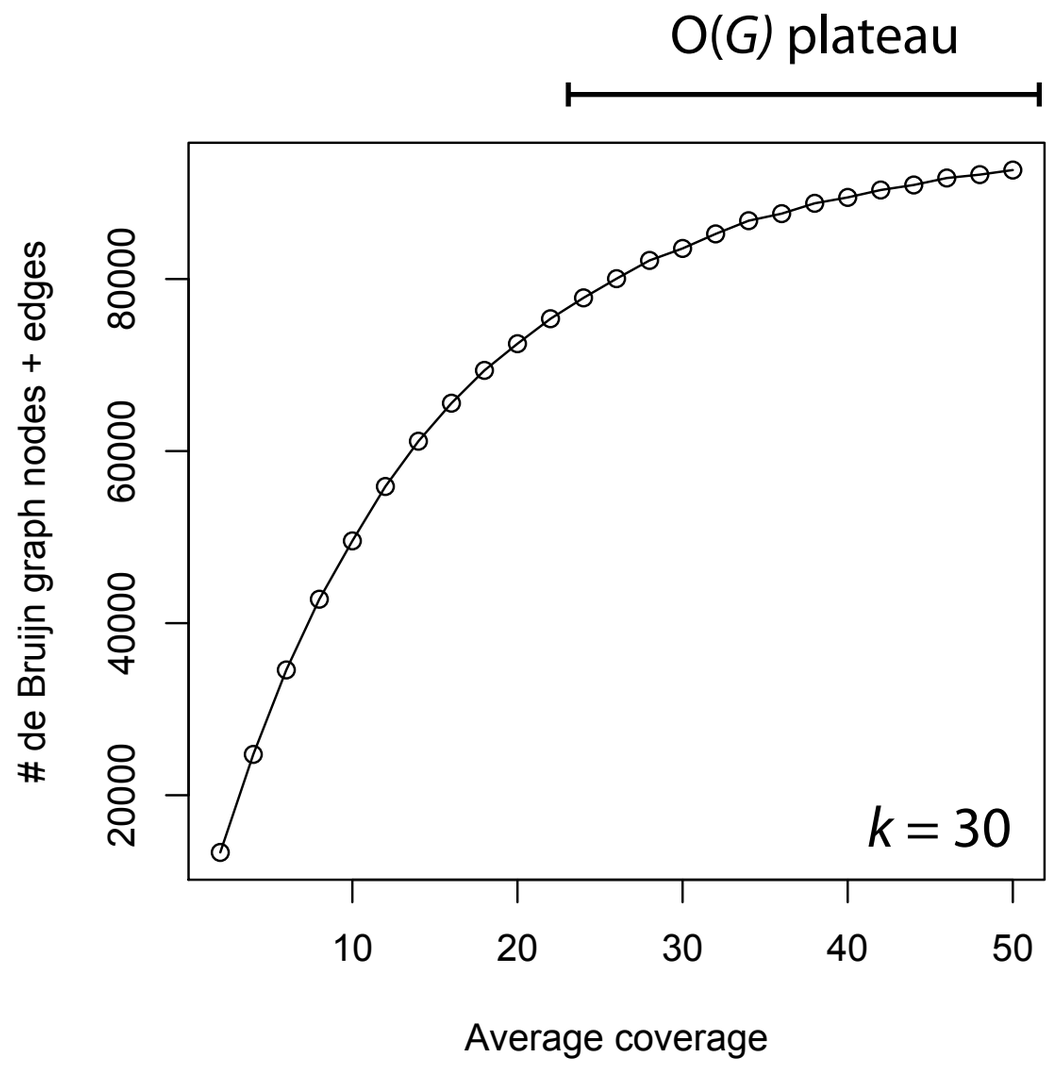
De Bruijn graph

At high coverage, O(min(*N*, *G*)) bound is advantageous

Genome: lambda phage (~48,500 bp)

Draw random *k*-mers until target average coverage (x axis) is reached

Build graph, sum # nodes and # edges (y axis)



De Bruijn graph

Advantages

Can build in O(N) expected time, N = total length of reads

With error-free data, space is O(min(N, G)); G = genome length

When average coverage is high, $G \ll N$

Compares favorably with overlap graph

Overlap graph has node for every read, edge for every overlap

Fast construction (suffix tree) is O($N + a$) time, where a is O(d^2)

De Bruijn graph

Disadvantages

Reads are immediately split into shorter *k*-mers, losing the ability to resolve some repeats resolvable by overlap graph

Only relatively short, exact overlaps are considered, which makes handling of sequencing errors more complicated

We lose *read coherence*. Some paths through De Bruijn graph are inconsistent with respect to input reads.

Assembly alternatives

	De Bruijn	Overlap
Time to build	$O(N)$	Suffix tree: $O(N + a)$ Dyn Prog: $O(N^2)$
Space	O(N) Error-free: $O(\min(N, G))$	$O(N + a)$

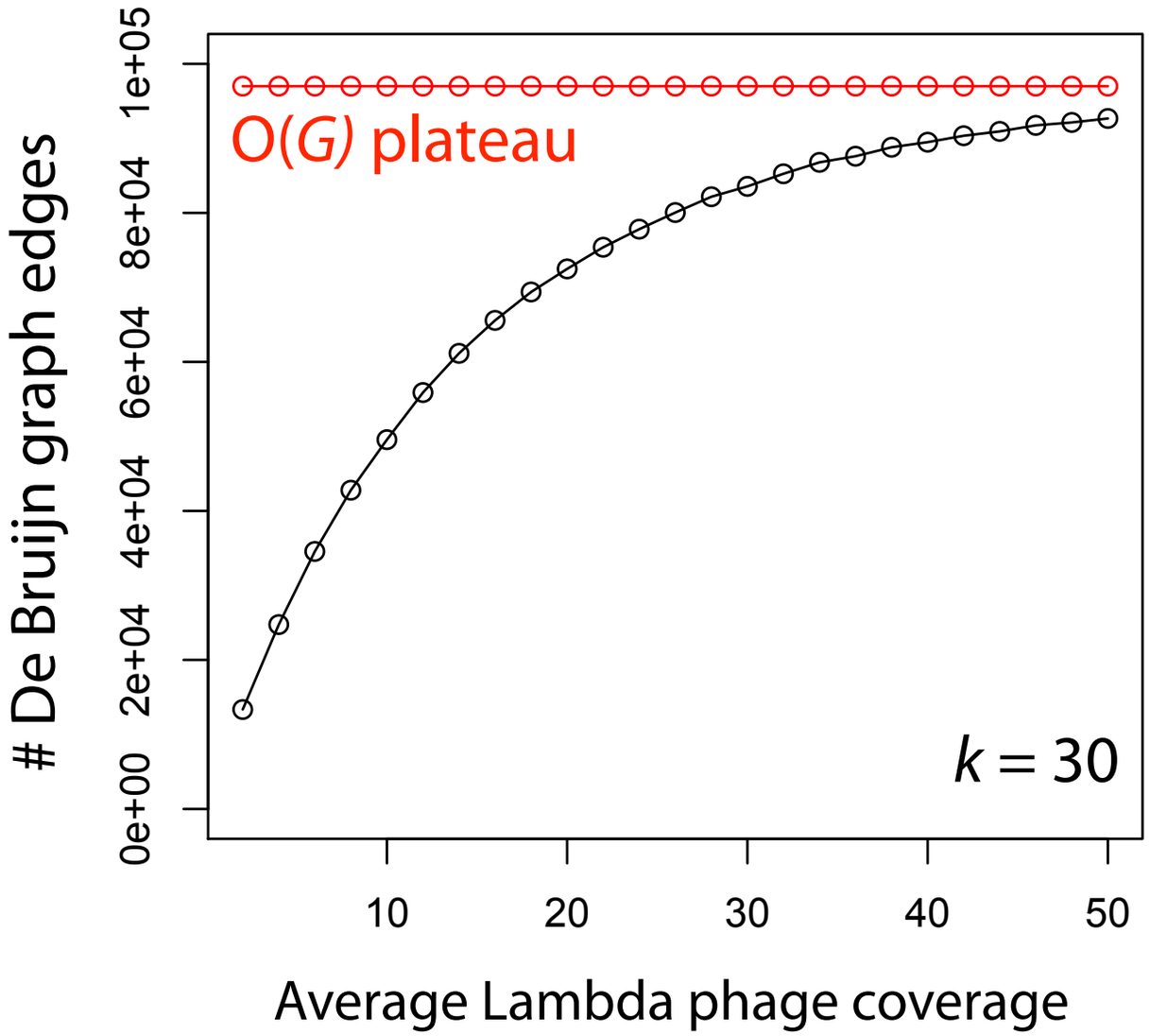
n = # reads
d = read length
$N = dn$ = # bases

a = # overlaps; $a \in O(n^2)$
G = source genome length

When average coverage is high, $G \ll N$ and the G is the more relevant bound for De Bruijn graph size

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

How many possible DNA strings of length *k*? 4^k

How many possible DNA strings of length 20? $4^{20} = 2^{40} \approx 1$ trillion

How many strings of length 20 in human genome? ~3 billion

For large *k*, set of *k*-mers in genome is tiny subset of all 4^k *k*-mers

Errors tend to yield *new k-mers* that don't appear elsewhere

> Given *k*-mer from genome, we expect most of its *neighbors*
> (e.g. by Hamming distance) are not in the genome

Analogy: correctly / incorrectly spelled words in collection of documents

Error correction

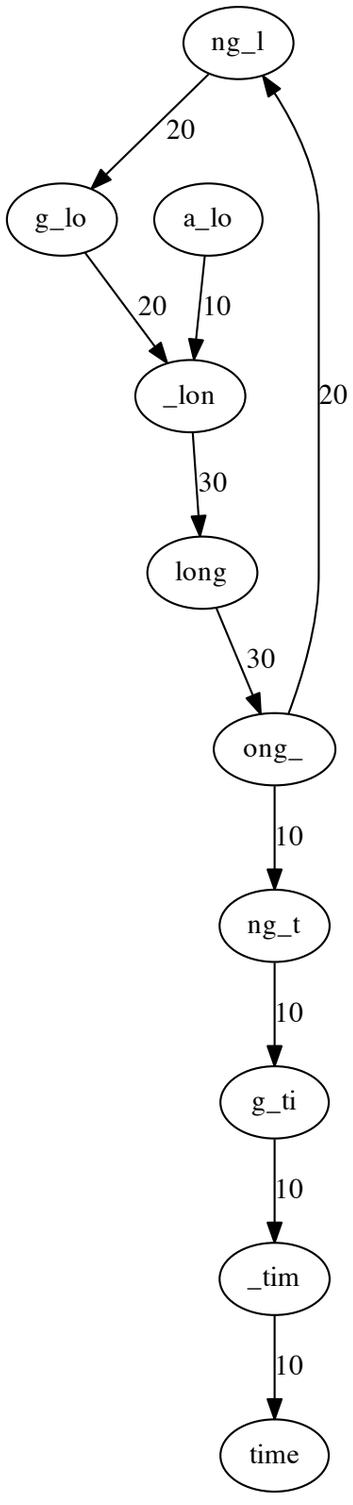
Correcting errors up-front prevents De Bruijn graph from growing far beyond O(*G)* plateau

How to correct?

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

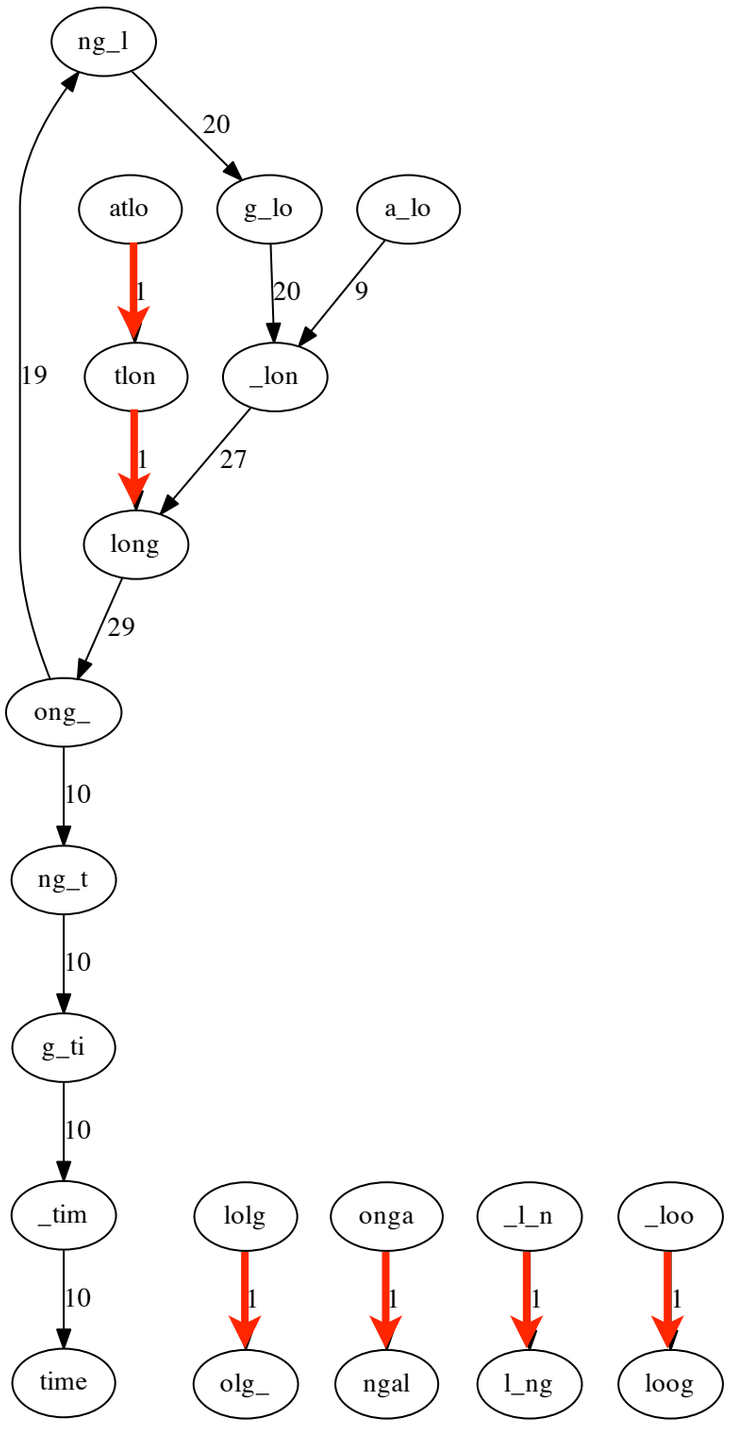
Errors tend to turn frequent words (*k*-mers) to infrequent ones. Corrections should do the reverse.

Error correction



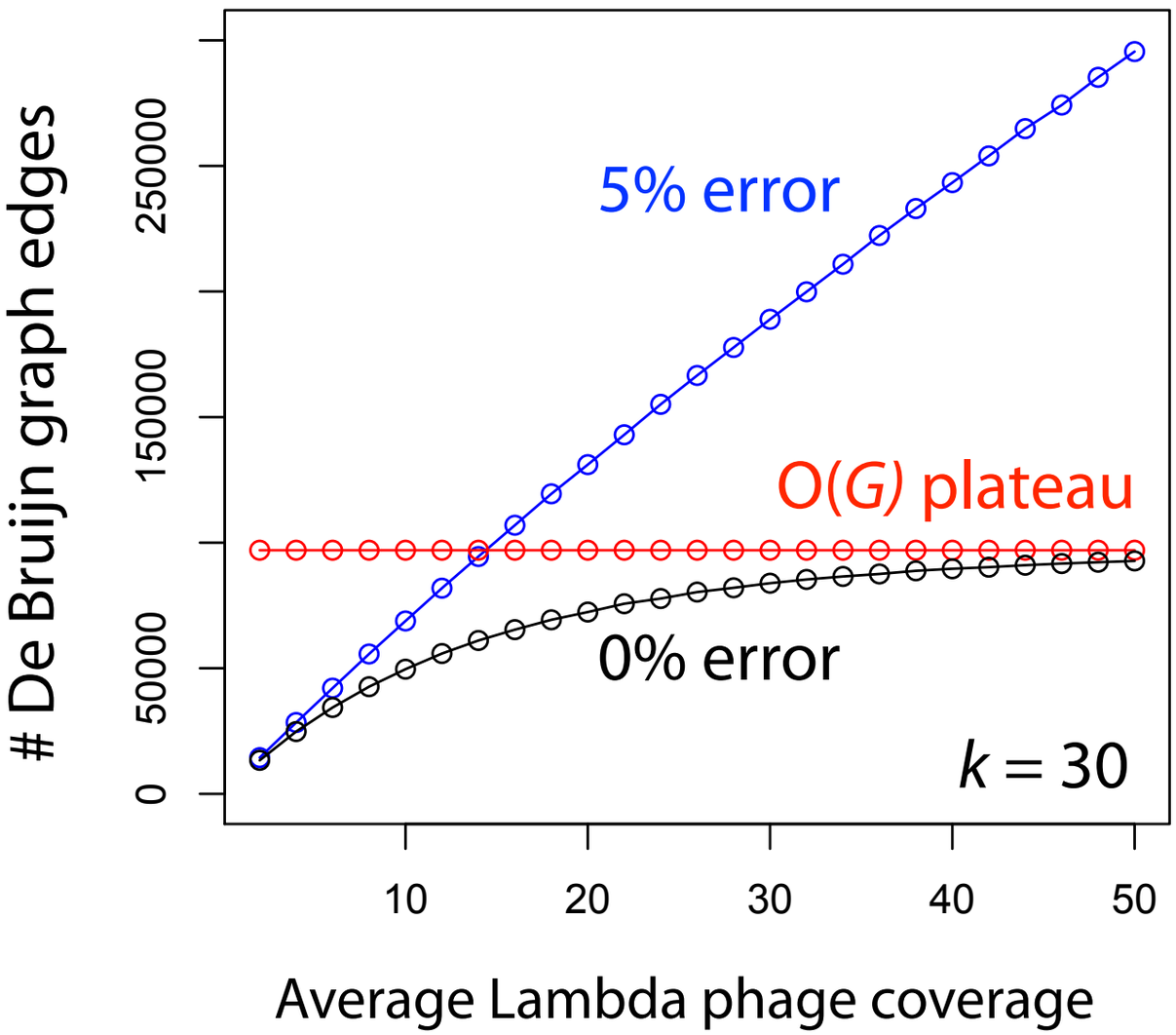
Left: Take example, mutate a *k*-mer character randomly with probability 1%

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



Error correction

As more *k*-mers overlap errors, # nodes & edges approach *N*

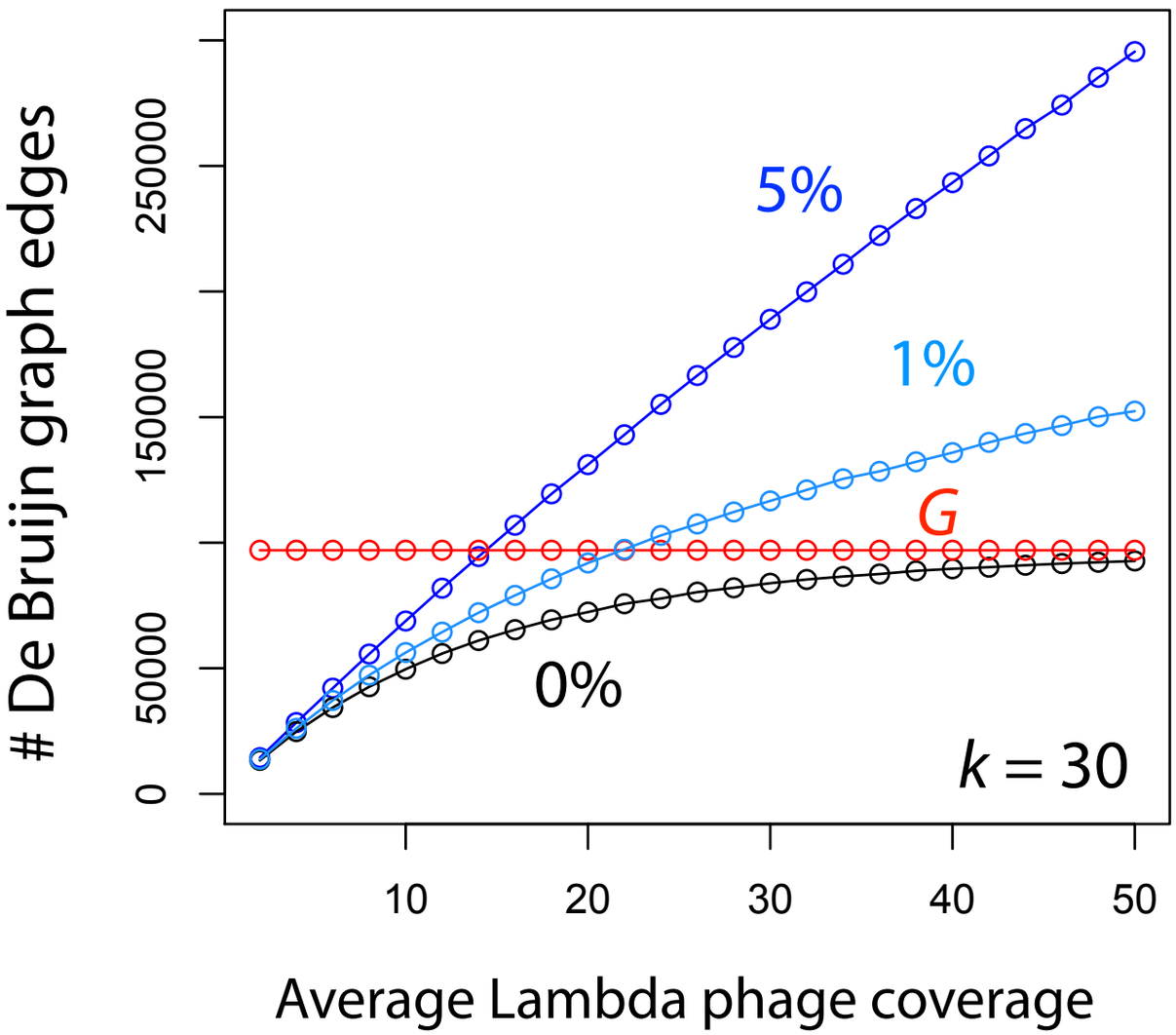


Same experiment as before, with 5% error added

Errors "push through" *G* bound

Error correction

As more *k*-mers overlap errors, # nodes & edges approach *N*



Same experiment as before, with 5% error added

Errors "push through" *G* bound

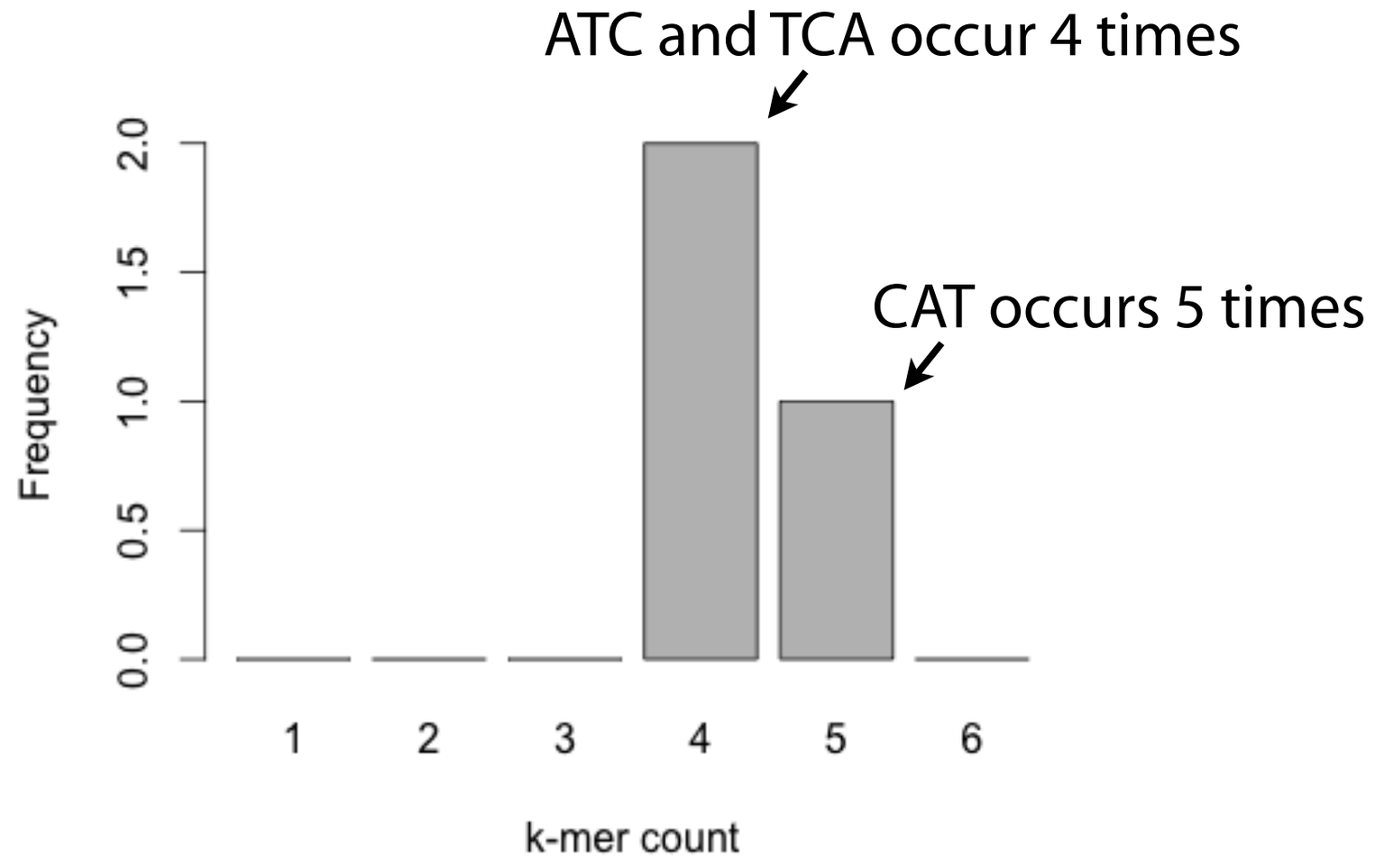
(Now with 1% error added)

Error correction

k-mer count histogram:

x axis is an integer *k*-mer count, y axis is # distinct *k*-mers with that count

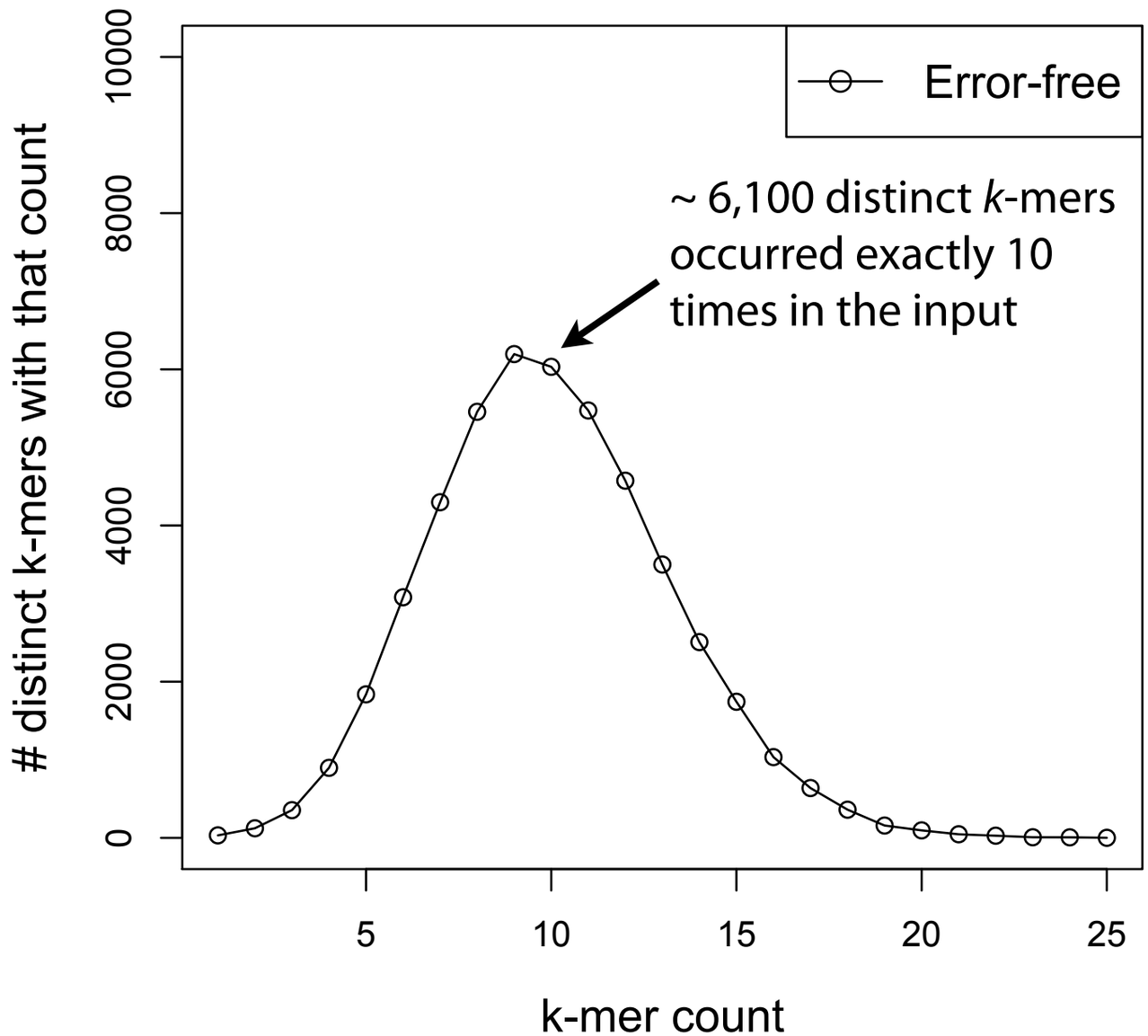
Right: such a histogram for 3-mers of CATCATCATCATCAT:



Error correction

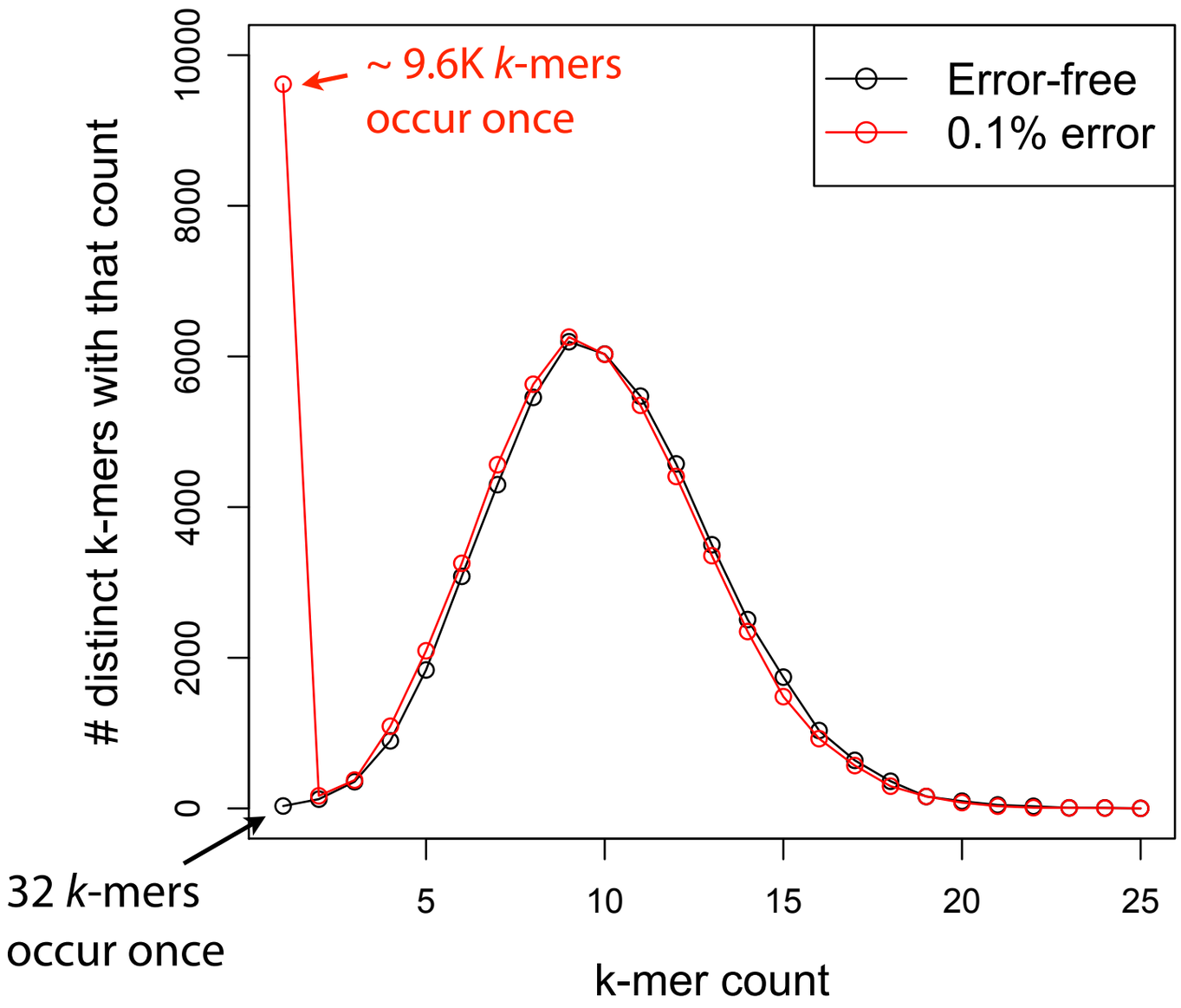
Draw 20-mers from genome randomly until each 20-mer has been drawn 10 times on average

How would the picture change for data with 1% error rate?



Error correction

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



Error correction

Idea: errors tend to turn frequent *k*-mers to infrequent *k*-mers, so corrections should do the reverse

Say each 8-mer occurs an average of ~10 times:

Read: `GCGTATTACGCGTCTGGCCT` (20 nt)

8-mers:

```
GCGTATTA: 8
CGTATTAC: 8
GTATTACG: 9
TATTACGC: 9
ATTACGCG: 10
TTACGCGT: 10
TACGCGTC: 11
ACGCGTCT: 11
CGCGTCTG: 10
GCGTCTGG: 10
CGTCTGGC: 11
GTCTGGCC: 9
TCTGGCCT: 8
```

times each 8-mer occurs in the reads.
"*k*-mer count profile"

All 8-mer counts are near average, suggesting read is error-free

Error correction

Suppose there's an error

Read: GCGTACTACGCGTCTGGCCT

```
GCGTACTA: 1
 CGTACTAC: 2
  GTACTACG: 1
   TACTACGC: 1
    ACTACGCG: 2
     CTACGCGT: 1
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

Below average

k-mer count profile has corresponding stretch of below-average counts

Around average

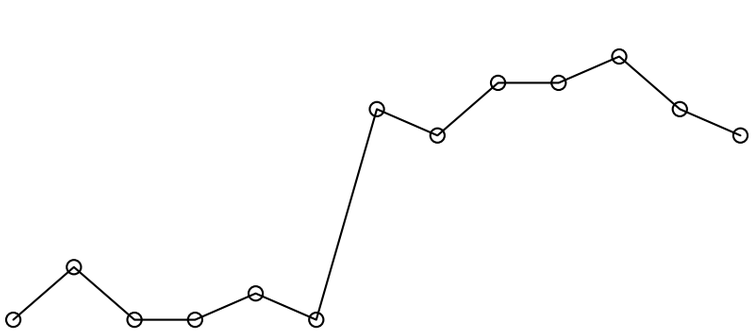
Error correction

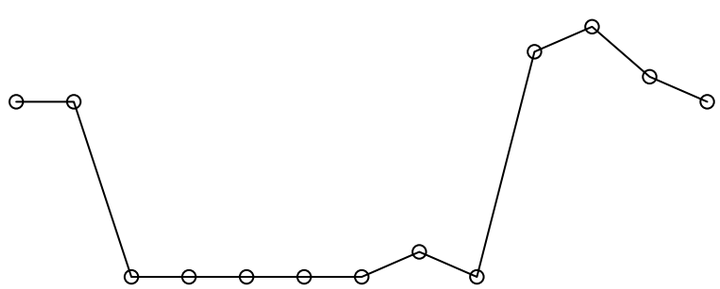
k-mer counts when errors are in different parts of the read:

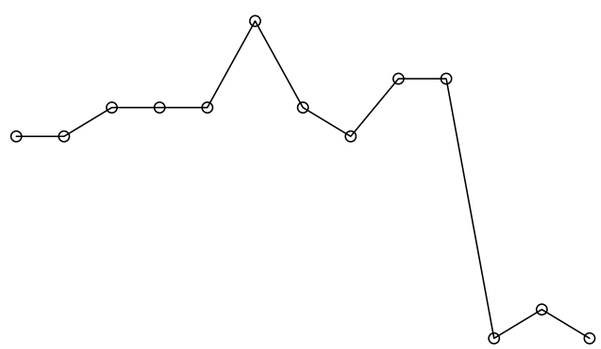
```
GCGTACTACGCGTCTGGCCT
GCGTACTA: 1
 CGTACTAC: 3
  GTACTACG: 1
   TACTACGC: 1
    ACTACGCG: 2
     CTACGCGT: 1
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

```
GCGTATTACACGTCTGGCCT
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACA: 1
   TATTACAC: 1
    ATTACACG: 1
     TTACACGT: 1
      TACACGTC: 1
       ACACGTCT: 2
        CACGTCTG: 1
         ACGTCTGG: 1
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

```
GCGTATTACGCGTCTGGTCT
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACG: 9
   TATTACGC: 9
    ATTACGCG: 9
     TTACGCGT: 12
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGT: 1
           GTCTGGTC: 2
            TCTGGTCT: 1
```

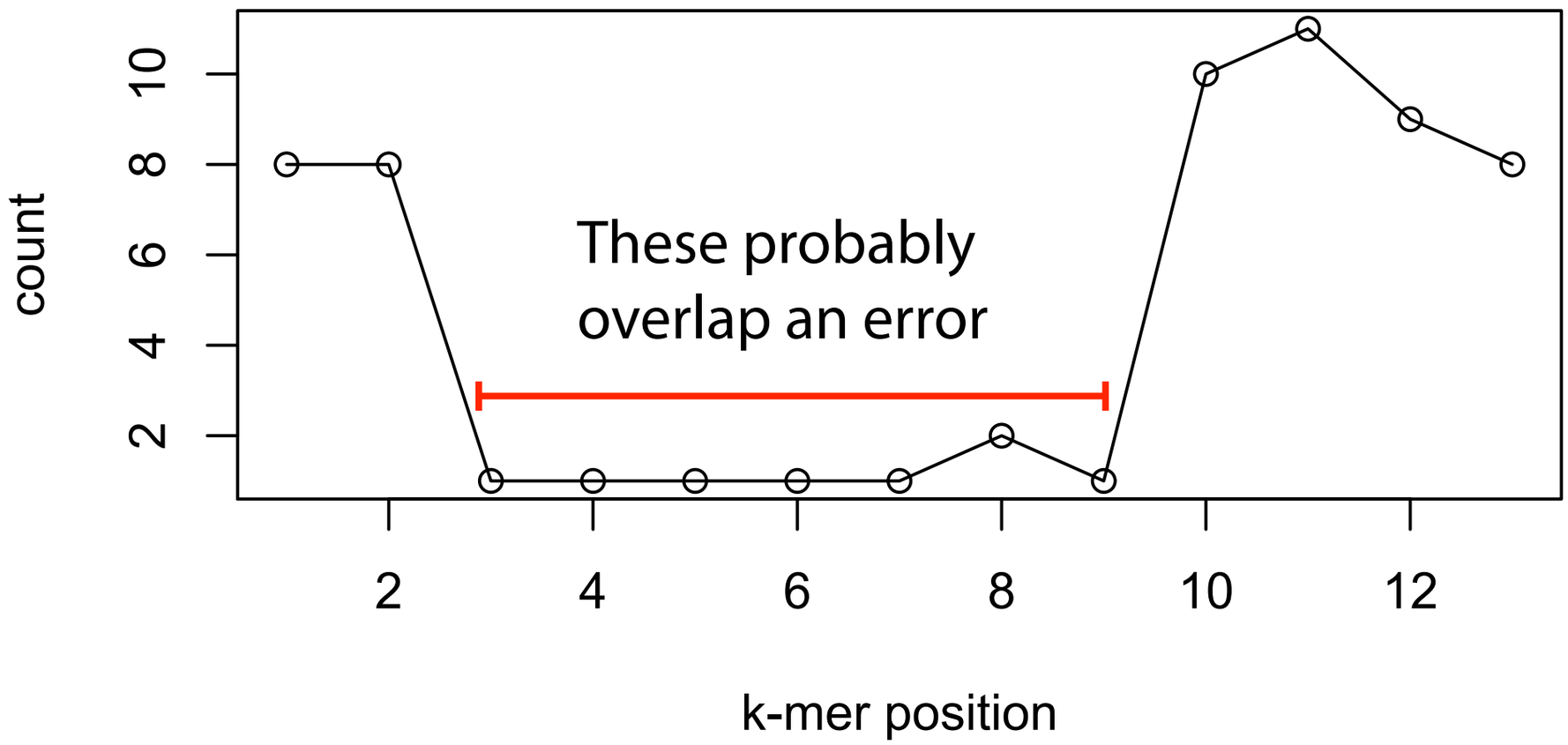






Error correction

Count profile indicates where errors are



Error correction

Simple algorithm, given a count threshold *t*:

For each read:

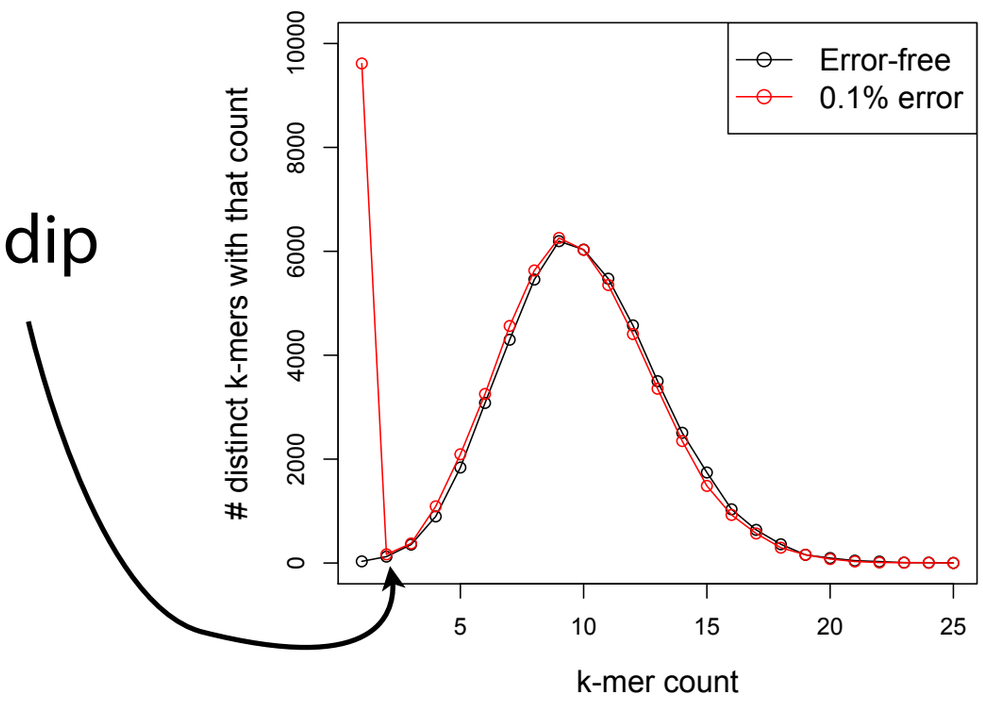
For each k-mer:

If *k*-mer count < *t*:

Examine *k*-mer's neighbors within some Hamming/edit distance.

If neighbor has count ≥ *t*, replace old *k*-mer with neighbor.

Pick *t* corresponding to dip between the peaks



Error correction: implementation excerpt

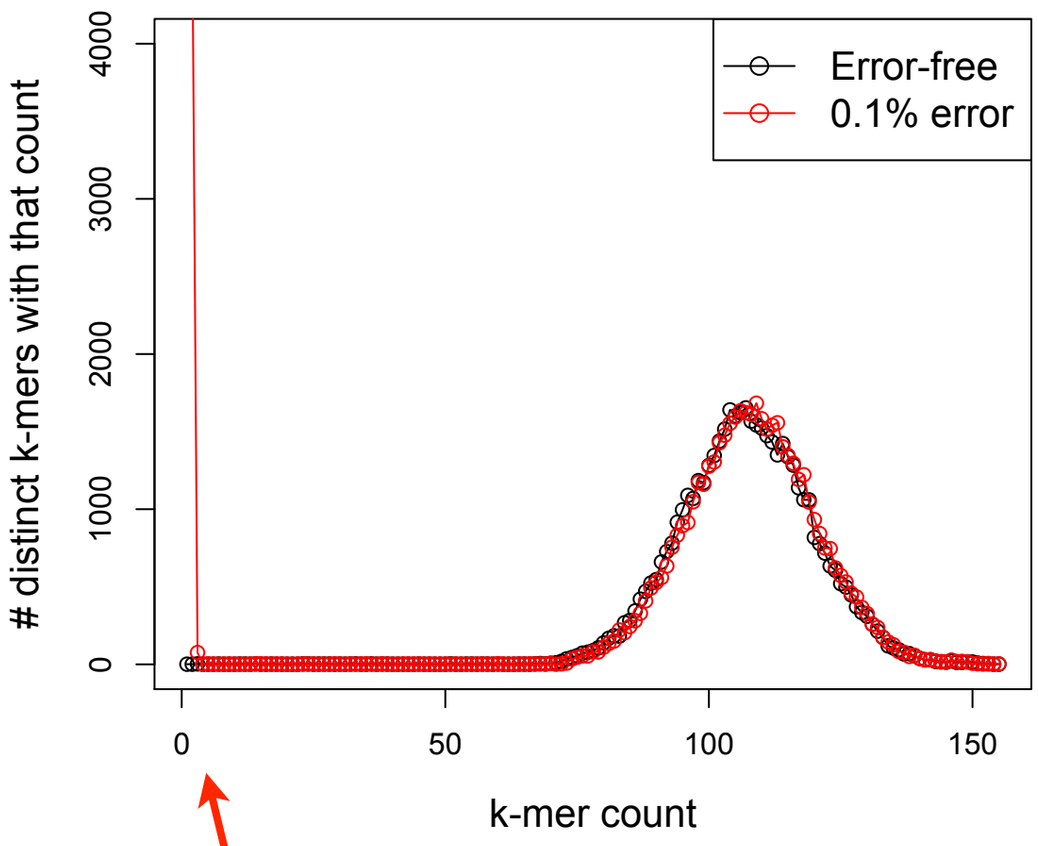
```
def correct1mm(read, k, kmerhist, alpha, thresh):
    ''' Return an error-corrected version of read.  k = k-mer length.
        kmerhist is kmer count map.  alpha is alphabet.  thresh is
        count threshold above which k-mer is considered correct. '''
    # Iterate over k-mers in read
    for i in range(0, len(read)-(k-1)):
        kmer = read[i:i+k]
        # If k-mer is infrequent...
        if kmerhist.get(kmer, 0) <= thresh:
            # Look for a frequent neighbor
            for newkmer in neighbors1mm(kmer, alpha):
                if kmerhist.get(newkmer, 0) > thresh:
                    # replace with neighbor
                    read = read[:i] + newkmer + read[i+k:]
                    break
return read
```

Full Python example: http://bit.ly/CG_ErrorCorrect

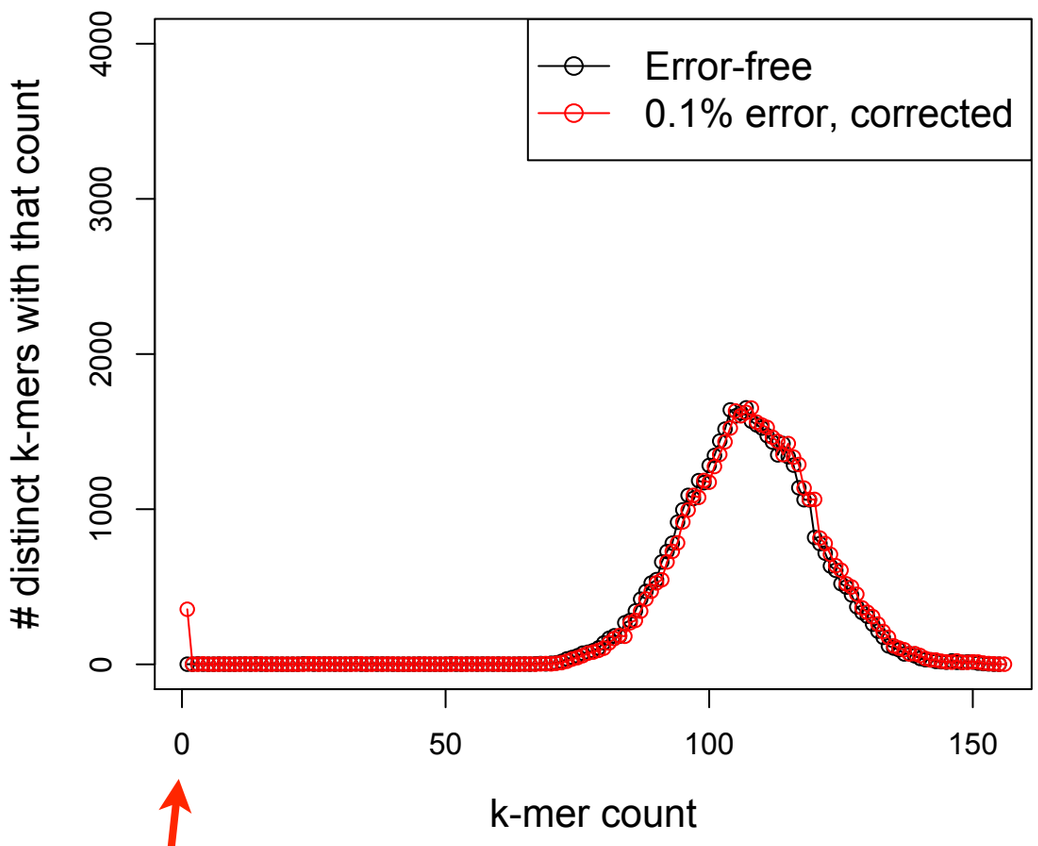
Error correction: results

Corrects 99.2% of errors in an example with 0.1% error added

Before



After



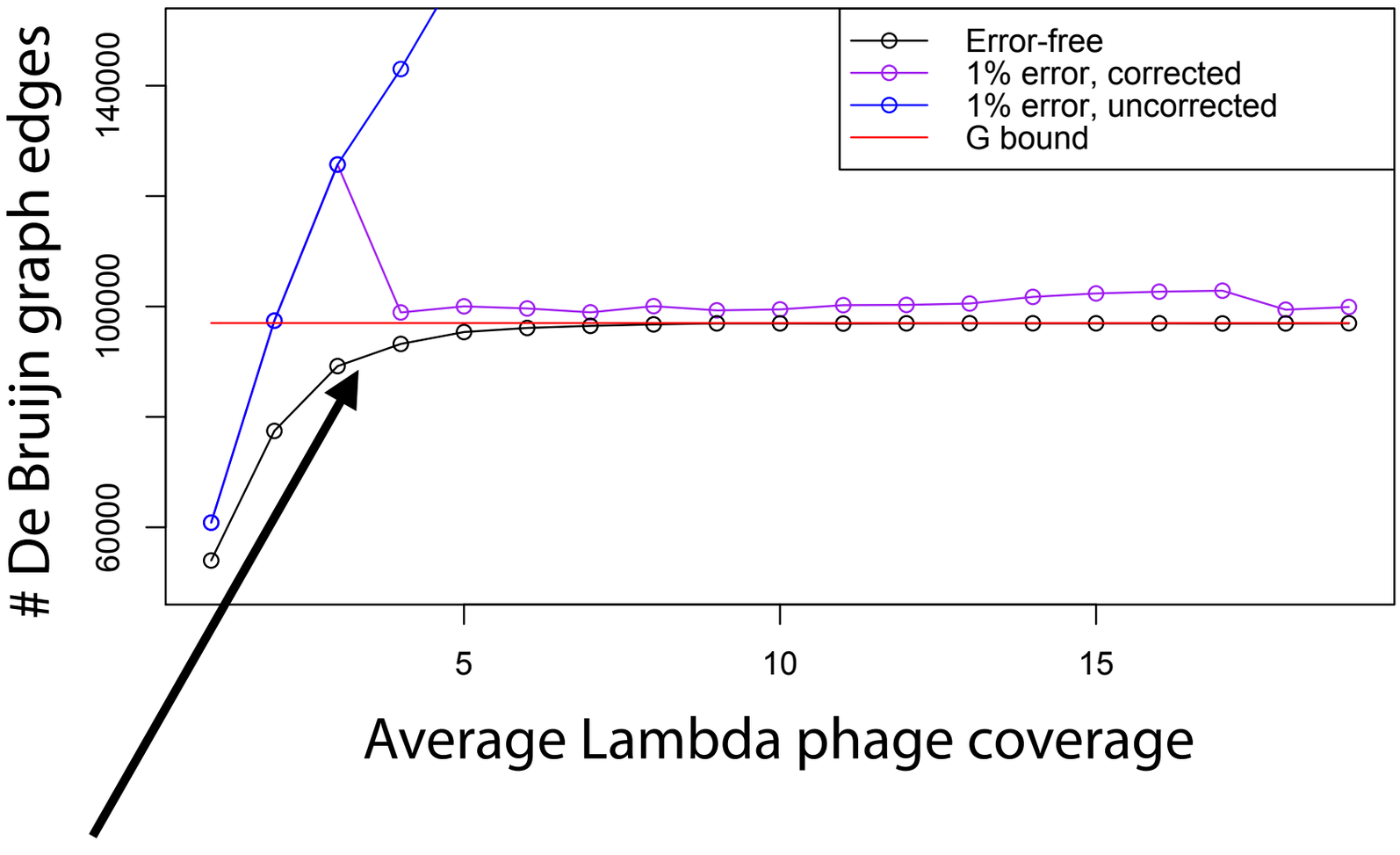
From 194K k-mers occurring exactly once to just 355

Error correction: results

Also works for 1% error...

Uncorrected, graph size is off the chart

Corrected, graph size is near G bound



...provided enough coverage to distinguish frequent/infrequent

Error correction

To work well:

Average coverage & *k* must be such that we can distinguish frequent from infrequent *k*-mers

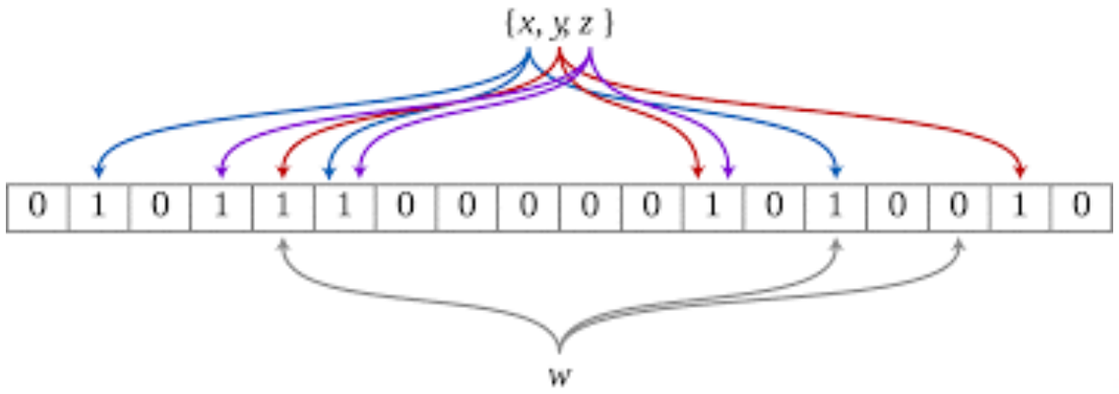
k-mer neighborhood explored must be broad enough to find frequent neighbors. Depends on error rate and *k*.

Alternately, we might give up on correcting and simply remove bad k-mers

Data structure for storing *k*-mer counts should be smaller than the De Bruijn graph

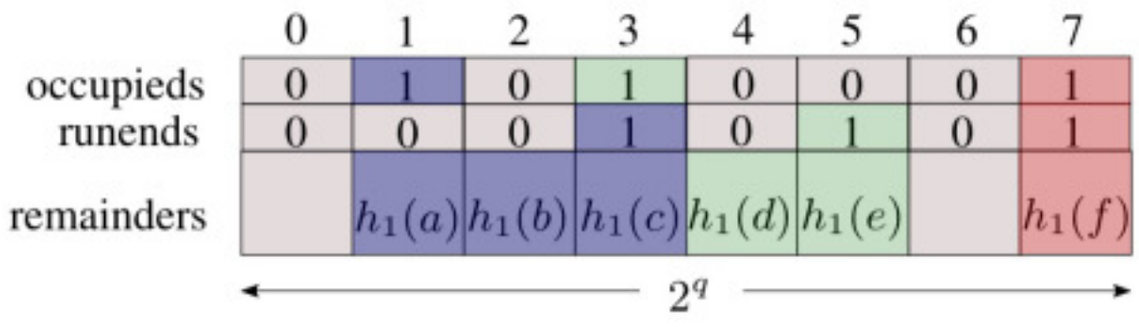
Otherwise, what's the point? 🙂

Data structures for error correction



Bloom filters

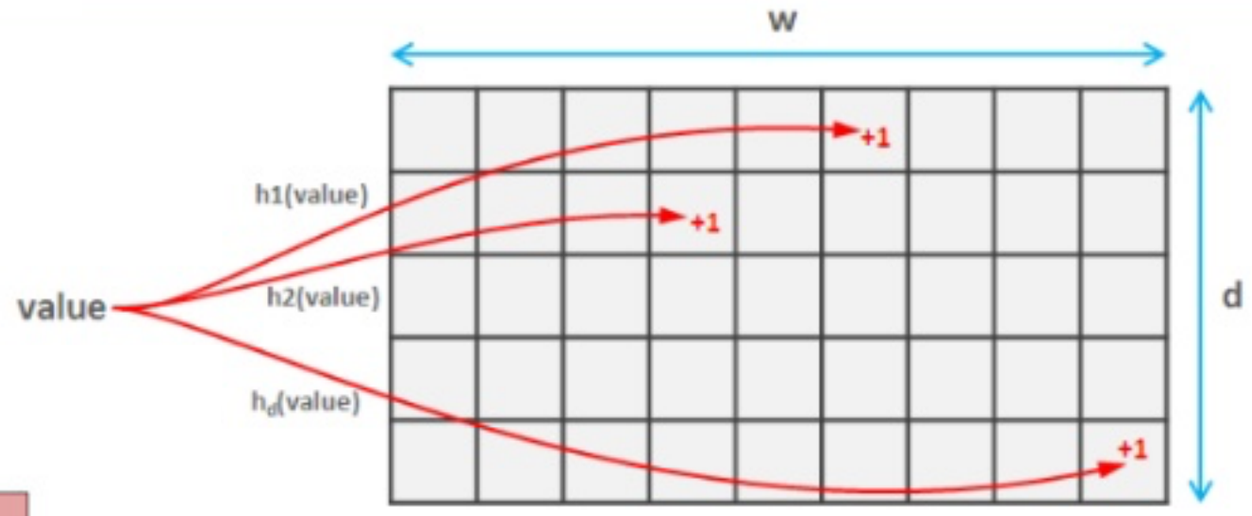
Song L, Florea L, Langmead B. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*. 2014;15(11):509.



Counting quotient filters

Pandey P, Bender MA, Johnson R, Patro R. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*. 2017; btx636.

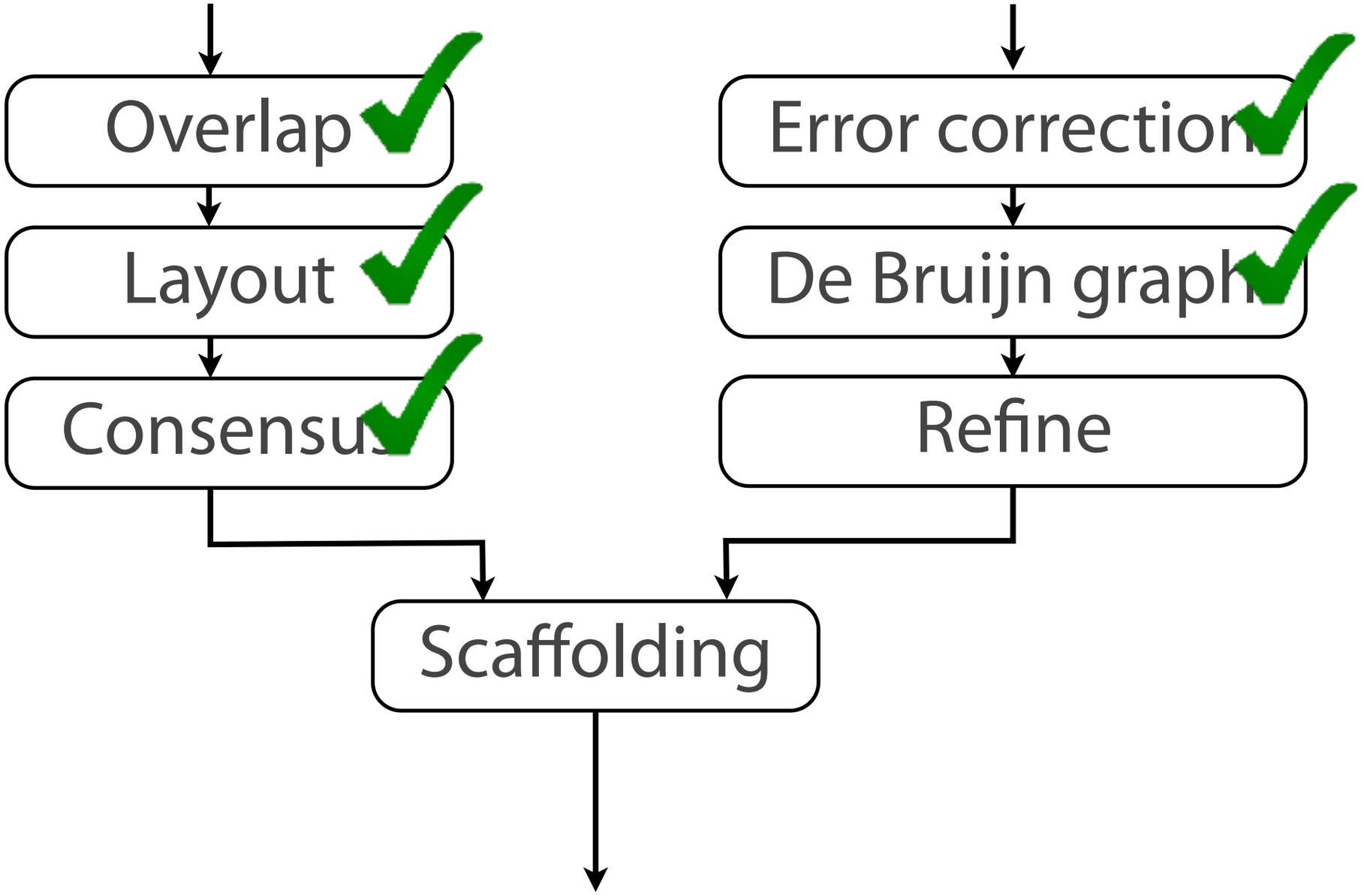
Don't need 100% accurate *k*-mer counts; just have to distinguish frequent and infrequent



CountMin sketches

Crusoe MR, Alameldin HF, Awad S, Boucher E, ..., Brown CT. The khmer software package: enabling efficient nucleotide sequence analysis. *F1000 Research*. 2015 Sep 25;4:900.

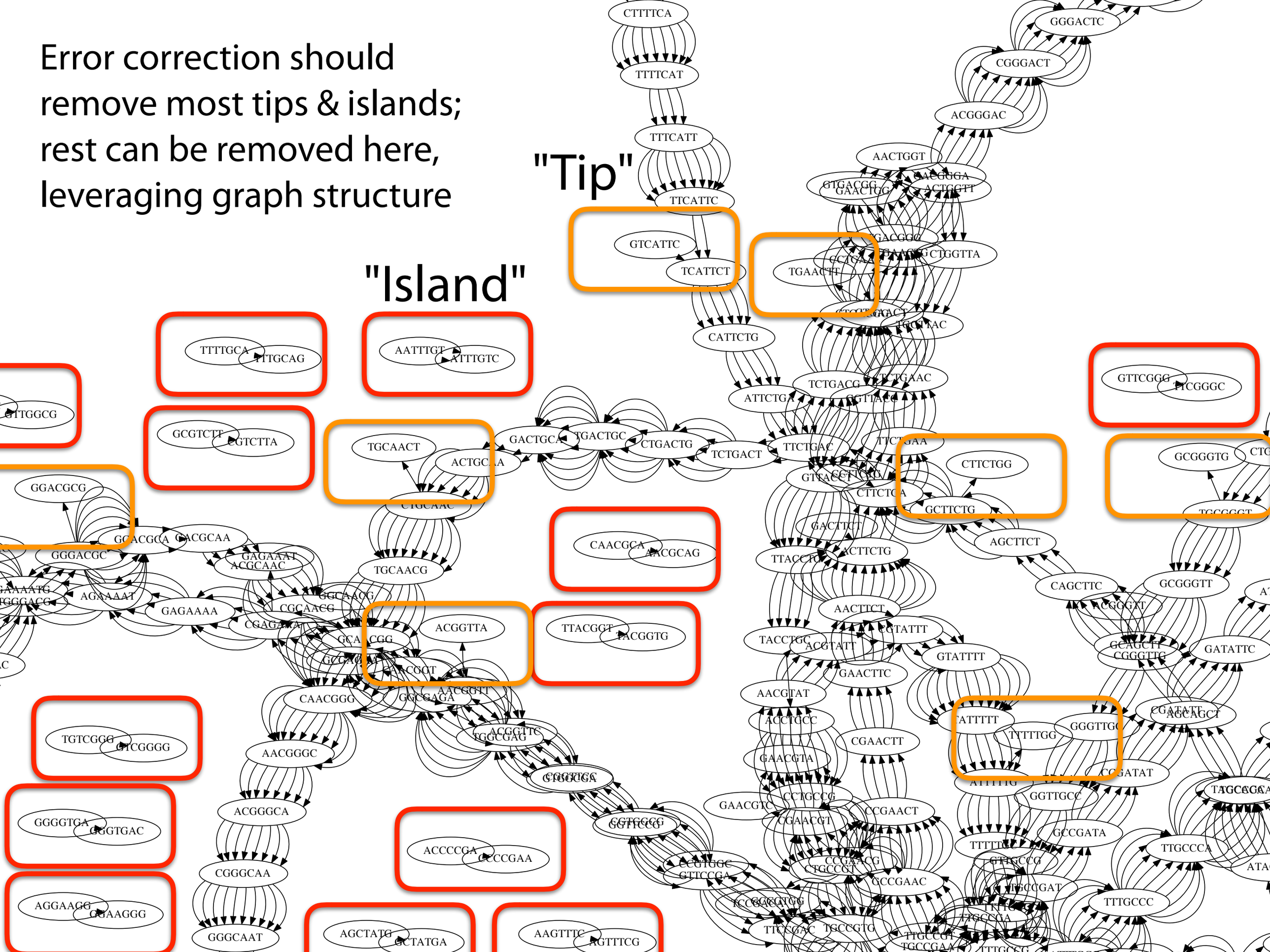
Assembly alternatives

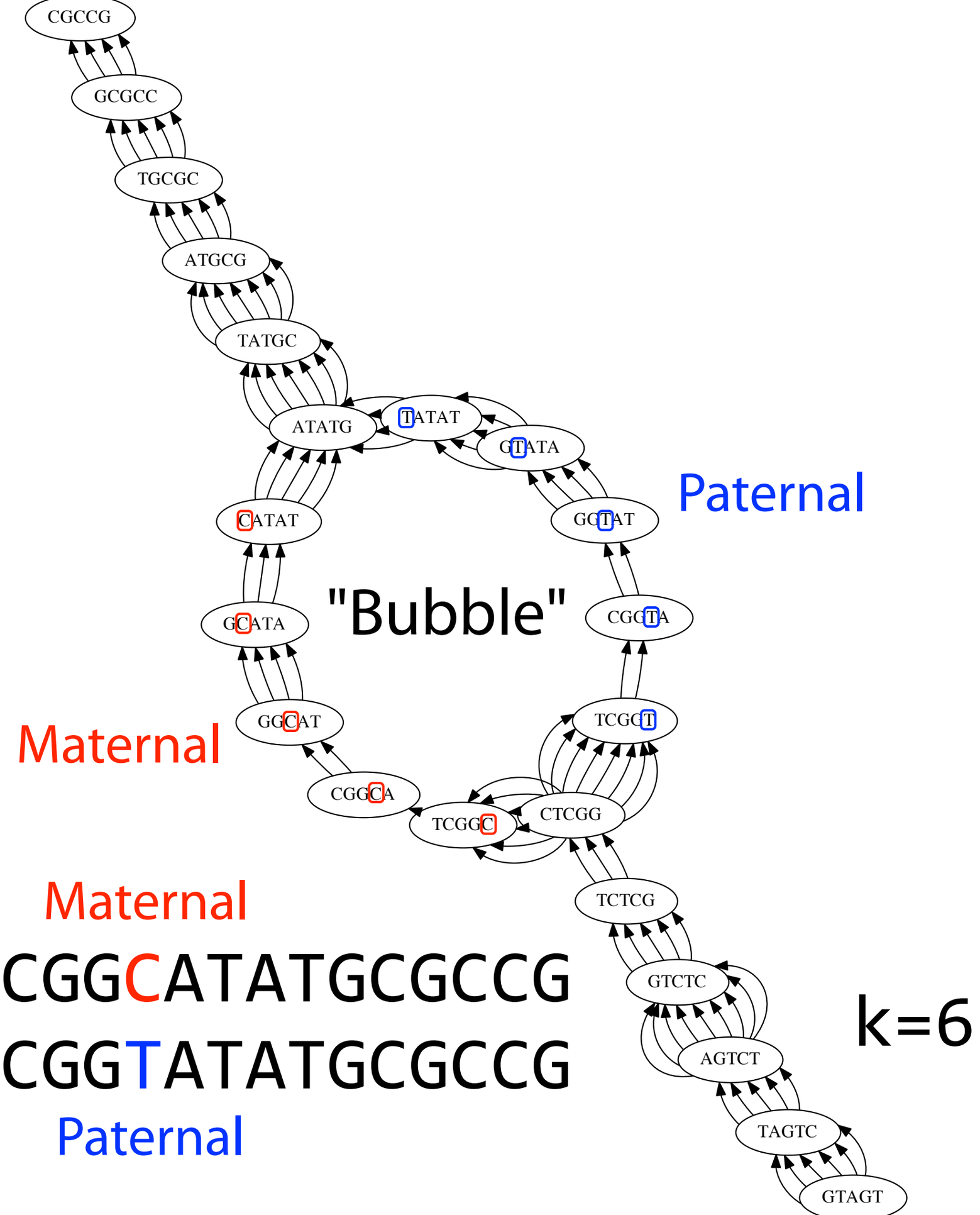


Error correction should remove most tips & islands; rest can be removed here, leveraging graph structure

"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