Genome Assembly 2

BCB 5200 Introduction Bioinformatics I

Fall 2017

Tae-Hyuk (Ted) Ahn

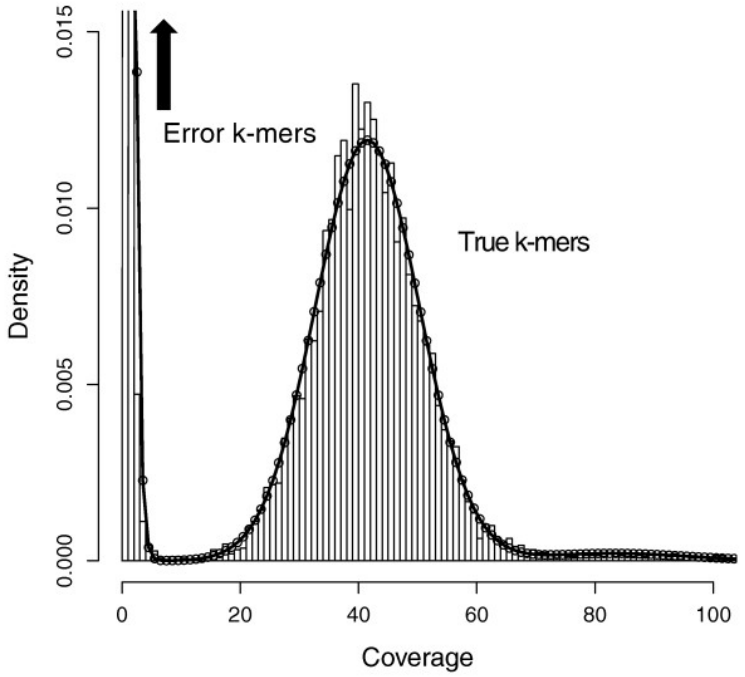
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS UNIVERSITY™

EST. 1818

Cleaning up the data

- Trim reads with low quality calls
- Remove short reads
- Correct errors:
  - Find all distinct k-mers (typically k=15) in input data
  - Plot coverage distribution
  - Correct low-coverage k-mers to match high-coverage
  - Part of several assemblers, also stand-alone Quake of khmer programs

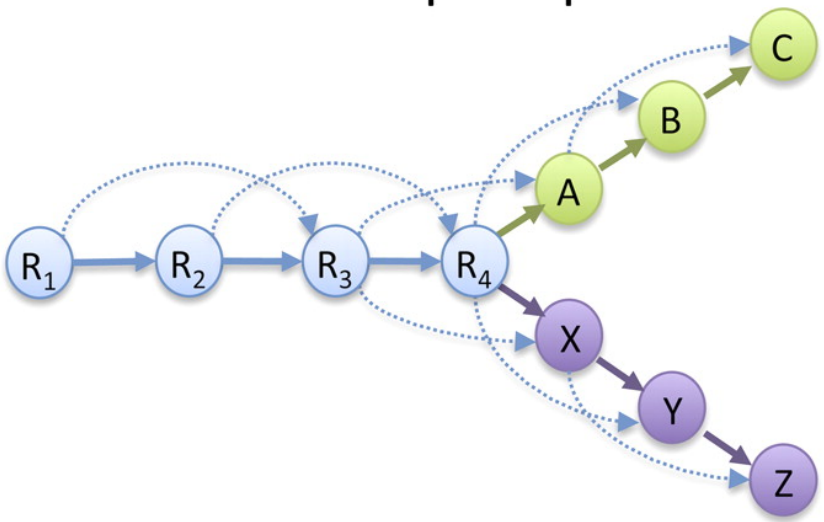


OLC vs De Bruijn

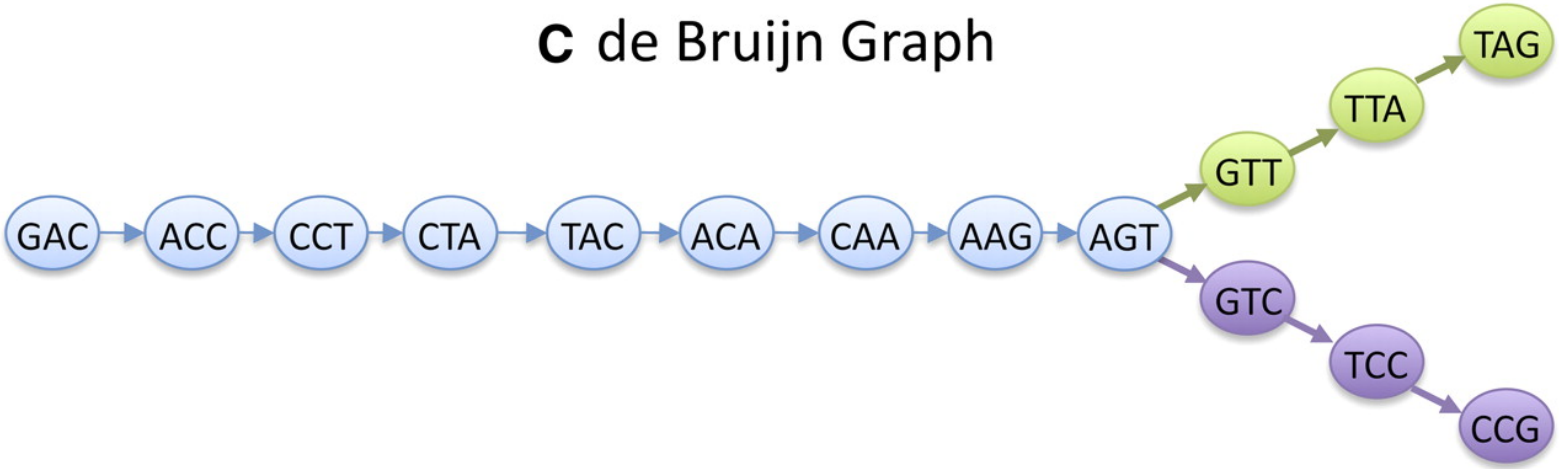
A Read Layout

```
R1: GACCTACA
R2:   ACCTACAA
R3:    CCTACAAG
R4:     CTACAAGT
A:       TACAAGTT
B:        ACAAGTTA
C:         CAAGTTAG
X:       TACAAGTC
Y:        ACAAGTCC
Z:         CAAGTCCG
```

B Overlap Graph



C de Bruijn Graph



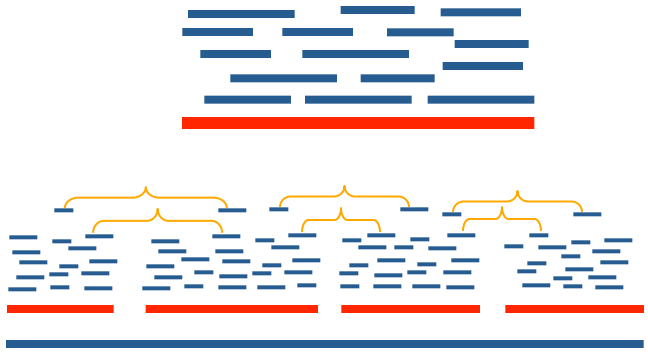
Overlap-Layout-Consensus

Assemblers (Old): ARACHNE, PHRAP, CAP, TIGR, CELERA

Assemblers (New): PBcR, Canu, Falcon

Overlap: find potentially overlapping reads

Layout: merge reads into contigs and contigs into supercontigs



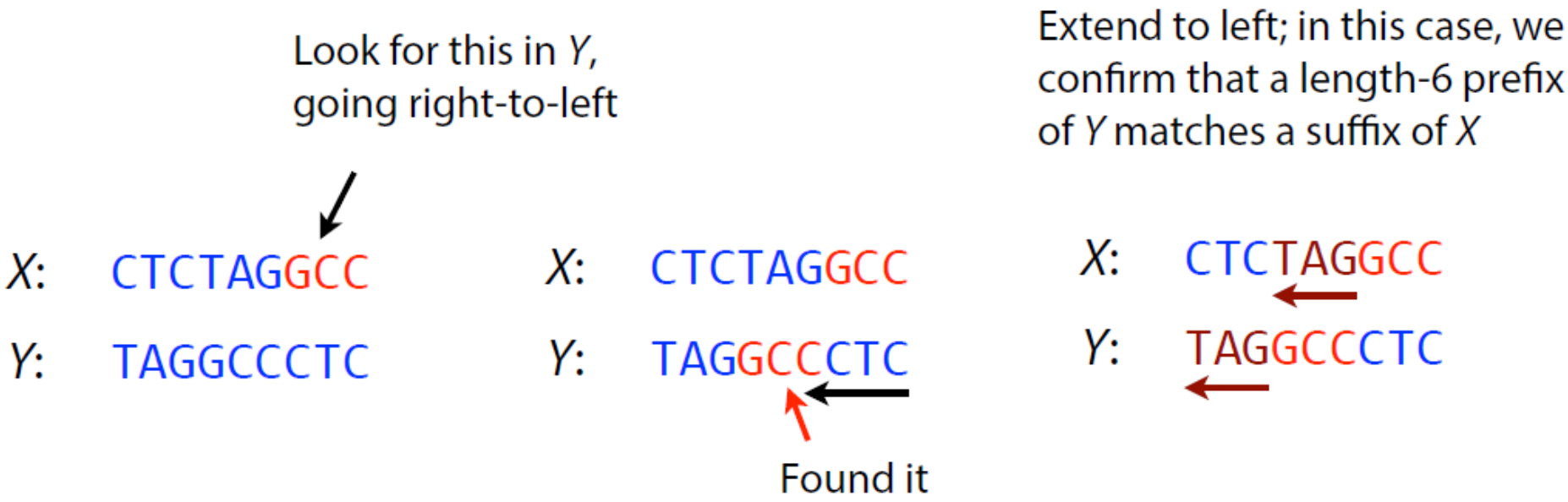
Consensus: derive the DNA sequence and correct read errors

..ACGATTACAATAGGTT..

Finding Overlaps

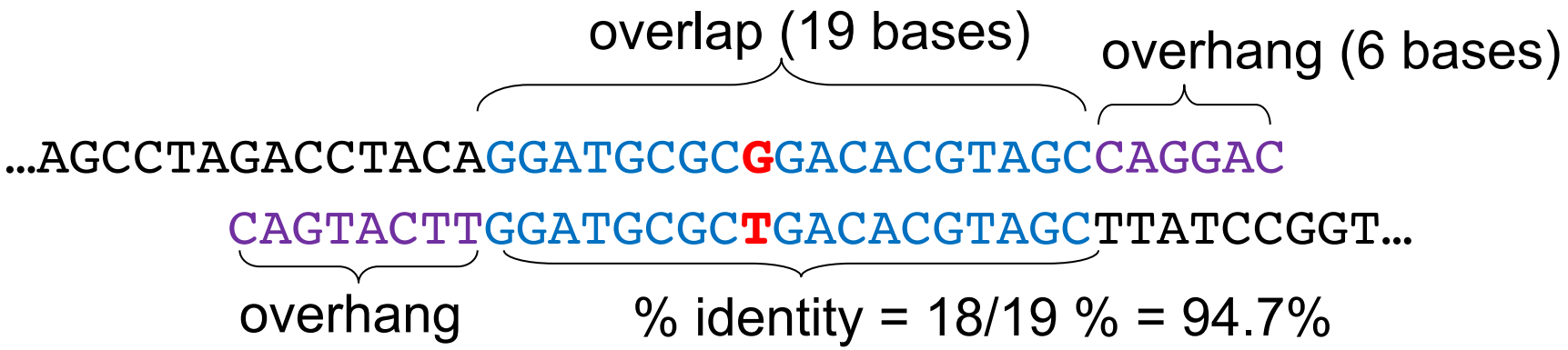
Can we be less naive than this?

Say $l = 3$



We're doing this for *every pair* of input strings

Overlap between two sequences



overlap - region of similarity between regions
overhang - un-aligned ends of the sequences

The assembler screens merges based on:

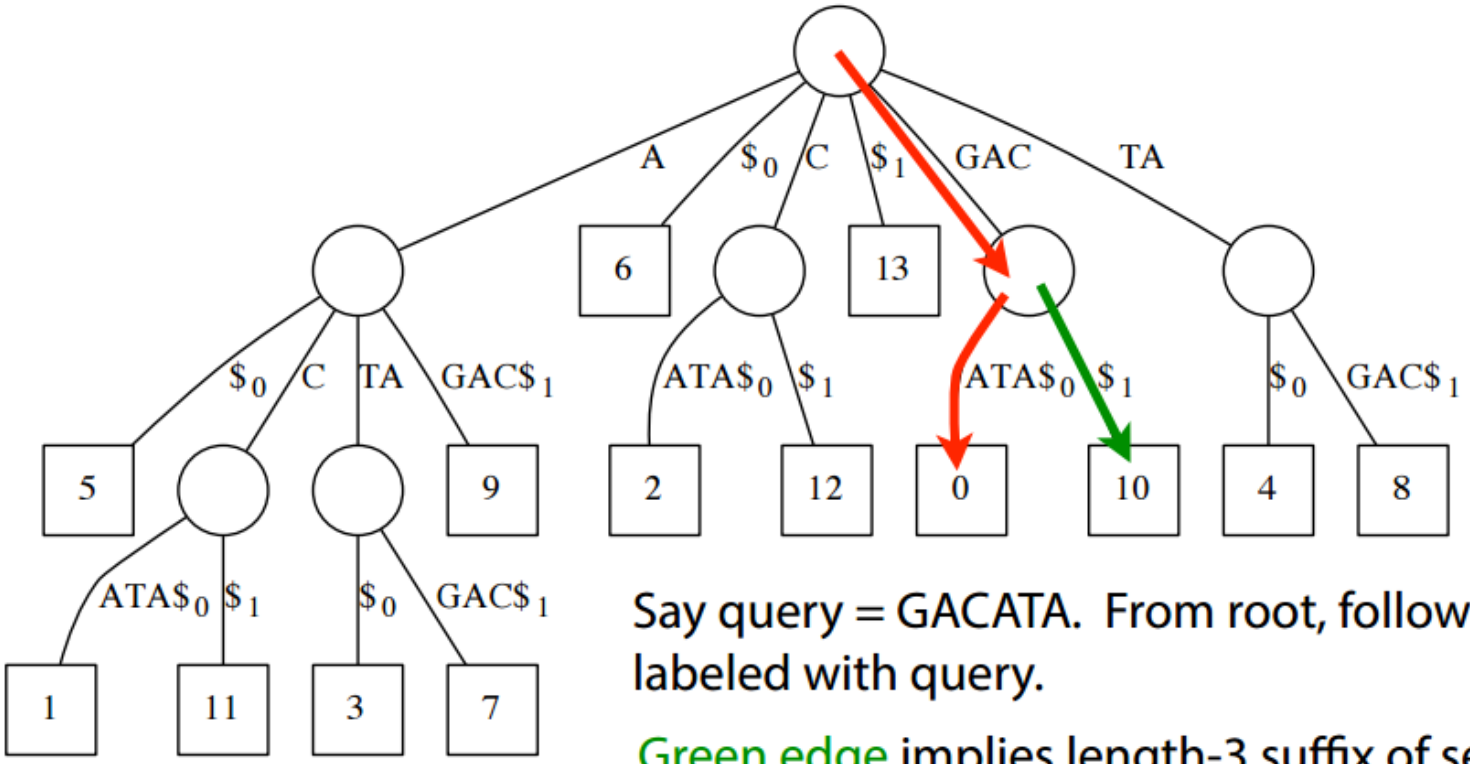
- length of overlap
- % identity in overlap region
- maximum overhang size.

Finding Overlaps

- Can we use suffix trees for overlapping?
- Problem: Given a collection of strings *S*, for each string *x* in *S* find all overlaps involving a prefix of *x* and a suffix of another string *y*
- Build a generalized suffix tree of the strings in *S*

Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"} GACATA$_0$ATAGAC$_1$



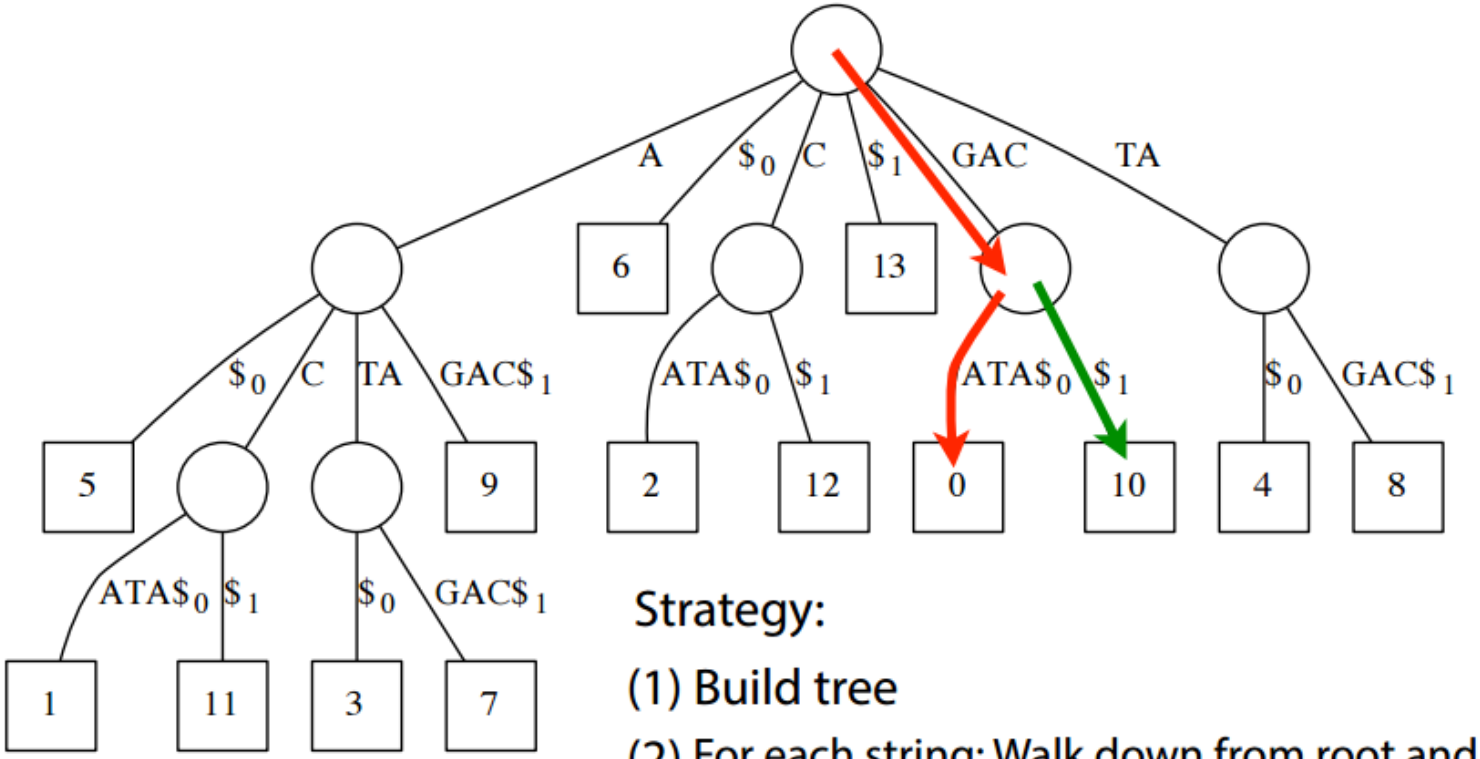
Say query = GACATA. From root, follow path labeled with query.

Green edge implies length-3 suffix of second string equals length-3 prefix of query

ATAGAC
|||
GACATA

Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"} GACATA$0ATAGAC$1

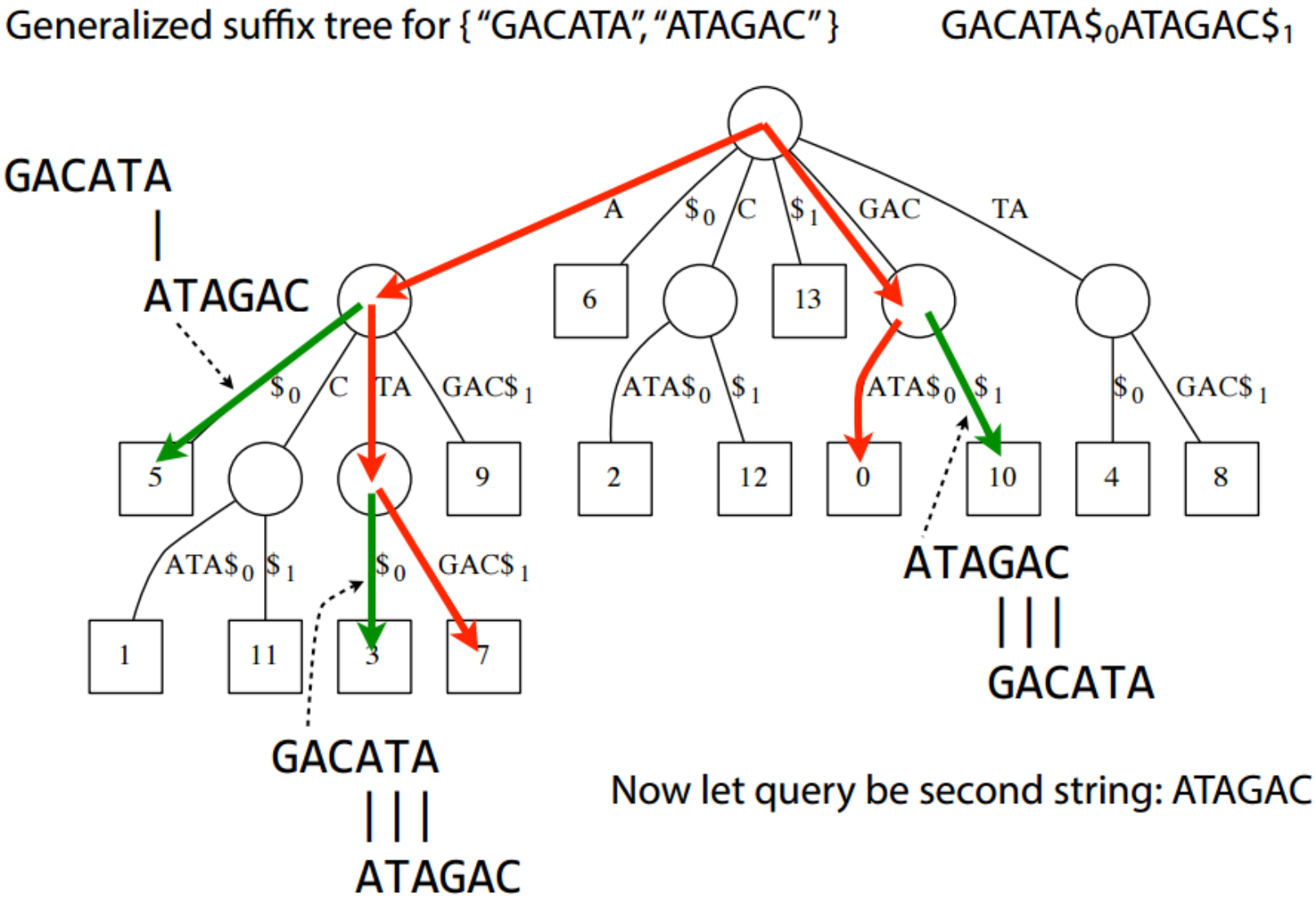


Strategy:

(1) Build tree

(2) For each string: Walk down from root and report any outgoing edge labeled with a separator. Each corresponds to a prefix/suffix match involving prefix of query string and suffix of string ending in the separator.

Finding overlaps with suffix tree



Allow Mismatches?

What if we want to allow mismatches and gaps in the overlap?

I.e. How do we find the best *alignment* of a suffix of *X* to a prefix of *Y*?

Dynamic programming

But we must frame the problem such that only backtraces involving a suffix of *X* and a prefix of *Y* are allowed

Finding overlaps

- What if we want to allow mismatches and gaps in the overlap?
- I.e. How do we "nd the best alignment Y: of a suffix of X to a pre"x of Y?: **Dynamic programming**

$$D[i,j] = \min \begin{cases} D[i-1,j] + s(x[i-1], -) \\ D[i,j-1] + s(-, y[j-1]) \\ D[i-1,j-1] + s(x[i-1], y[j-1]) \end{cases}$$

$s(a,b)$

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

```
X: CTCGGCCCTAGG
   ||| |||||
Y:    GGCTCTAGGCCC
```

How to initialize first row & column so suffix of *X* aligns to prefix of *Y*?

First column gets 0s
(any suffix of *X* is possible)

First row gets ∞s
(must be a prefix of *Y*)

Backtrace from last row

Y

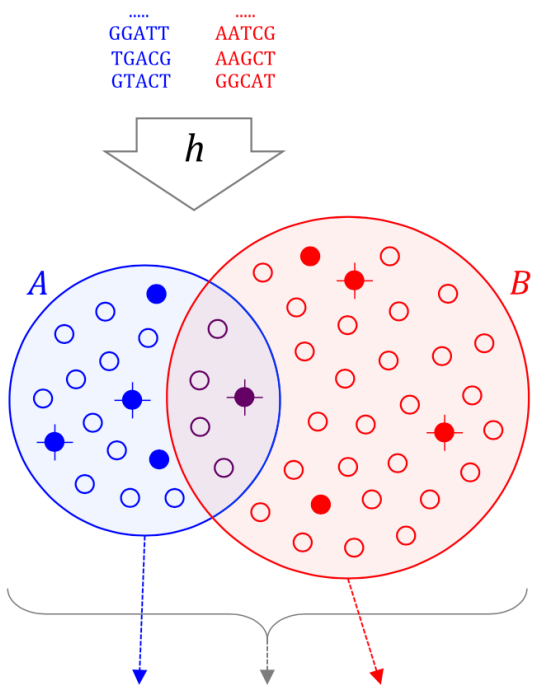
X		-	G	G	C	T	C	T	A	G	G	C	C	C
	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	C	0	4	12	20									
	T	0	4	8	14									
	C	0	4	8	8									
	G	0		4	12									
	G	0	0		8	16	16	24	26	30	36	44	52	60
	C	0	4	4		8	16	18	26	30	34	36	44	52
	C	0	4	8	4		8	16	22	30	34	34	36	44
	C	0	4	8	8	6		10	18	26	34	34	34	36
	T	0	4	8	10	8	8		10	18	26	34	36	36
	A	0	2	6	12	14	12	10		10	18	26	34	40
	G	0	0	2	10	16	18	16	10		10	18	26	34
	G	0	0	0	6	14	20	22	18	10		10	18	26

```
X: CTCGGCCCTAGG
   ||| |||||
Y:    GGCTCTAGGCCC
```

New Techniques

MinHash and mash:

Computing Jaccard distance (=> Average Nucleotide Identity) between any two genomic samples, very very quickly.



$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \approx \frac{|S(A \cup B) \cap S(A) \cap S(B)|}{|S(A \cup B)|}$$

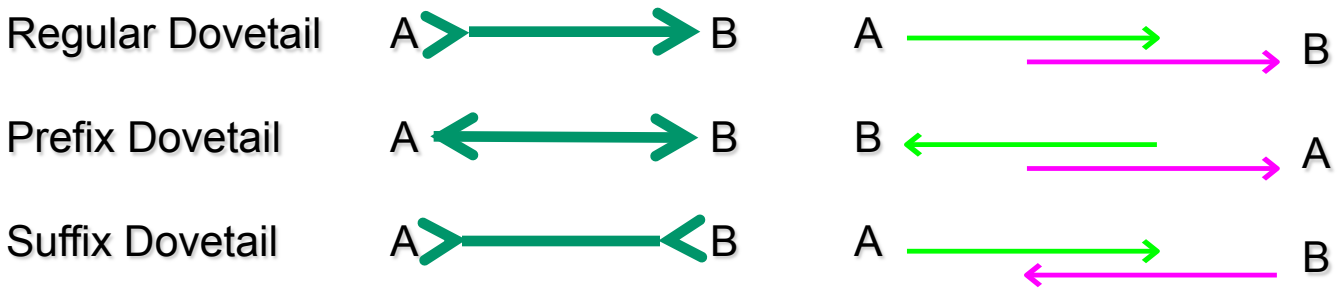
A ∪ B = Union = combined set of A and B

A ∩ B = intersection = set of shared objects

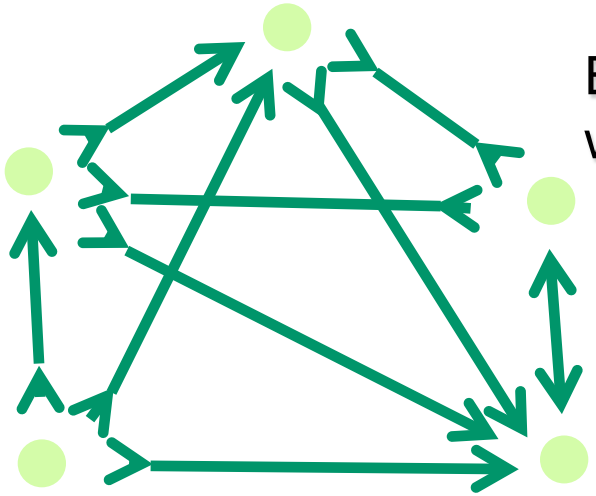
Overlap Graph

Bi-directed Graph Format

Edge Types:



E.G.:



Edges are annotated with deltas of overlaps

Layout

- Overlap graph is big and messy. Contigs don't "pop out" at us.
- Anything redundant about this part of the overlap graph?
- Bundle stretches of the overlap graph into unitigs

Layout

```
Read r1   :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2   :  CGTAATTTAGCTATCGGCTATCGTAAATGCTTAGC
Read r3   :  AACGTAATTTAGCTATCGGCTATCGTAAATGCTTA
Read r4   :  GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r5   :  GTATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r6   :  ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7   :  TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r8   :  ATATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r9   :  ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r10  :  CTATATAACGTAATTTAGCTATCGGCTATCGTAAA
```

Set of given reads

```
Read r1   :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r̄2   :   GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read r̄3   :     TAAGCATTTACGATAGCCGATAGCTAAATTACGTT
Read r4   :        GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r̄5   :         CATTTACGATAGCCGATAGCTAAATTACGTTATAC
Read r6   :          ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7   :           TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r̄8   :         CATTTACGATAGCCGATAGCTAAATTACGTTATAT
Read r9   :          ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r̄10  :           TTTACGATAGCCGATAGCTAAATTACGTTATATAG
```

Overlapping reads

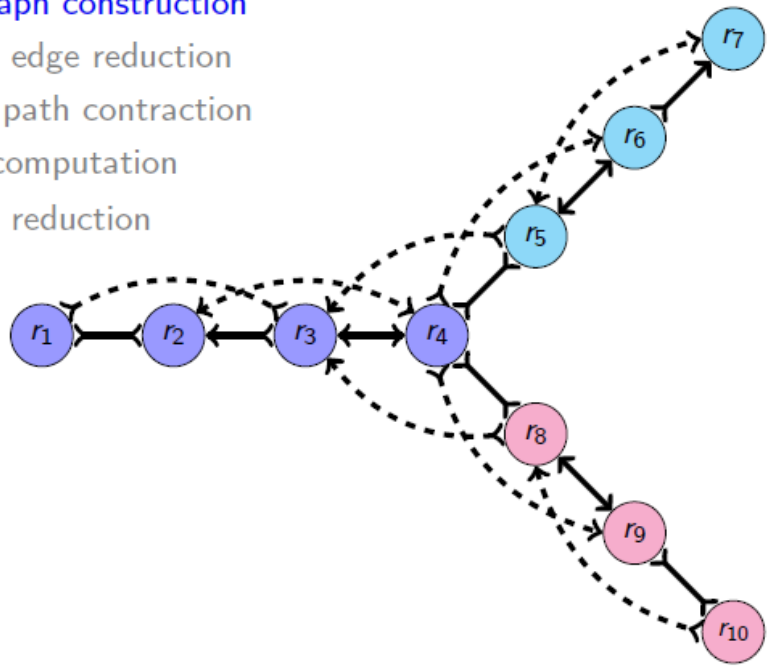
Overlap graph construction

Transitive edge reduction

Composite path contraction

Flow computation

Tree reduction



Layout

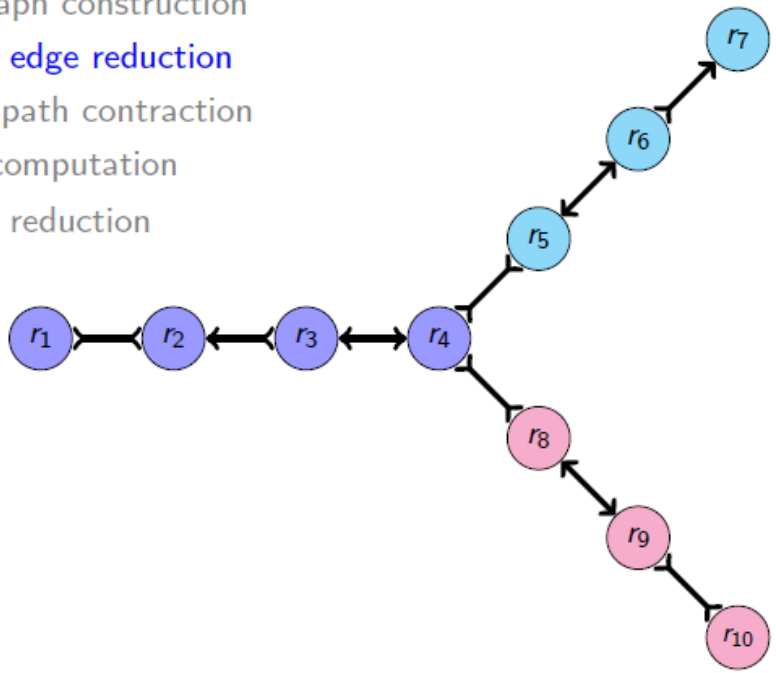
Read r_1 : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r_2 : CGTAATTTAGCTATCGGCTATCGTAAATGCTTAGC
Read r_3 : AACGTAATTTAGCTATCGGCTATCGTAAATGCTTA
Read r_4 : GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r_5 : GTATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r_6 : ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r_7 : TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r_8 : ATATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r_9 : ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r_{10} : CTATATAACGTAATTTAGCTATCGGCTATCGTAAA

Set of given reads

Read r_1 : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read $\bar{r}_2$: GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read $\bar{r}_3$: TAAGCATTTACGATAGCCGATAGCTAAATTACGTT
Read r_4 : GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read $\bar{r}_5$: CATTTACGATAGCCGATAGCTAAATTACGTTATAC
Read r_6 : ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r_7 : TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read $\bar{r}_8$: CATTTACGATAGCCGATAGCTAAATTACGTTATAT
Read r_9 : ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read $\bar{r}_{10}$: TTTACGATAGCCGATAGCTAAATTACGTTATATAG

Overlapping reads

Overlap graph construction
Transitive edge reduction
Composite path contraction
Flow computation
Tree reduction



Layout

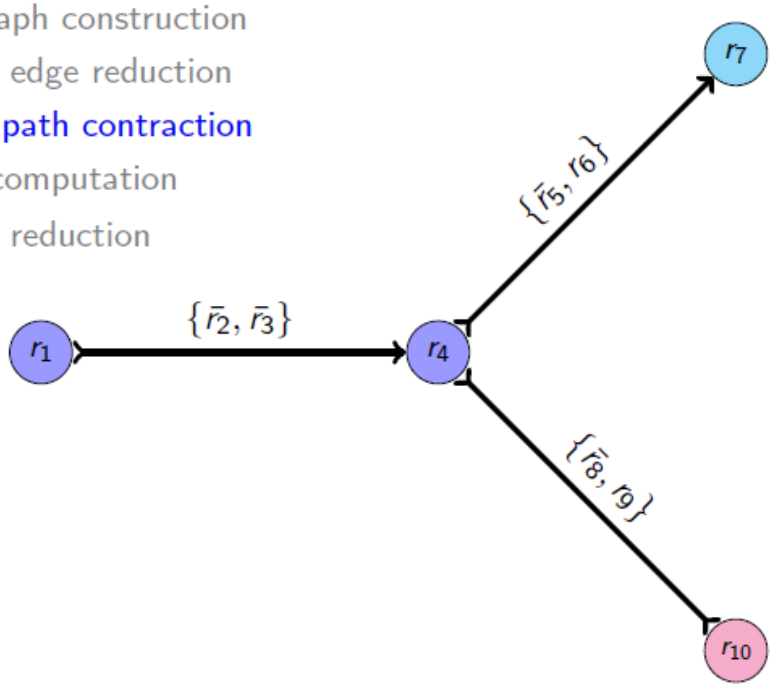
Read r_1 : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r_2 : CGTAATTTAGCTATCGGCTATCGTAAATGCTTAGC
Read r_3 : AACGTAATTTAGCTATCGGCTATCGTAAATGCTTA
Read r_4 : GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r_5 : GTATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r_6 : ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r_7 : TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r_8 : ATATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r_9 : ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r_{10} : CTATATAACGTAATTTAGCTATCGGCTATCGTAAA

Set of given reads

Read r_1 : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read $\bar{r}_2$: GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read $\bar{r}_3$: TAAGCATTTACGATAGCCGATAGCTAAATTACGTT
Read r_4 : GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read $\bar{r}_5$: CATTTACGATAGCCGATAGCTAAATTACGTTATAC
Read r_6 : ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r_7 : TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read $\bar{r}_8$: CATTTACGATAGCCGATAGCTAAATTACGTTATAT
Read r_9 : ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read $\bar{r}_{10}$: TTTACGATAGCCGATAGCTAAATTACGTTATATAG

Overlapping reads

Overlap graph construction
Transitive edge reduction
Composite path contraction
Flow computation
Tree reduction



Layout

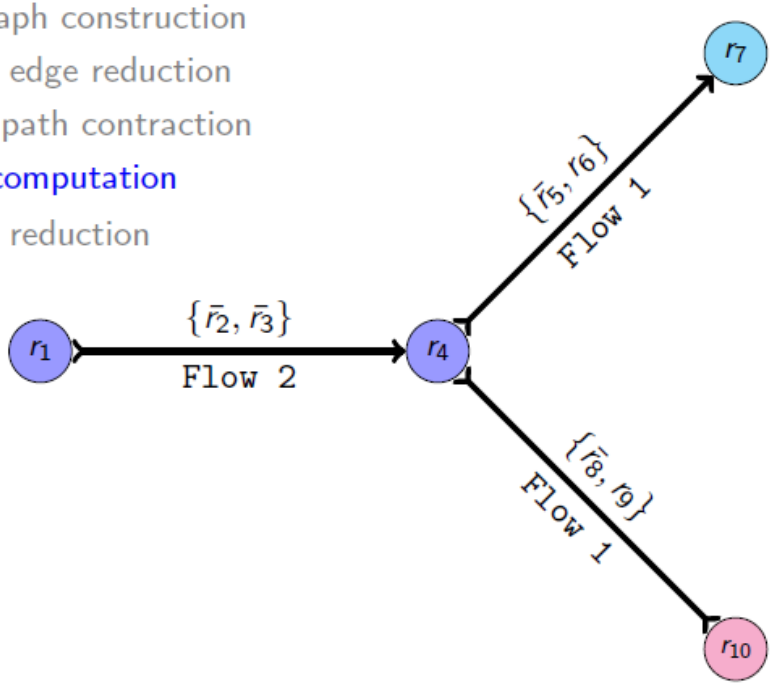
Read r_1 : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r_2 : CGTAATTTAGCTATCGGCTATCGTAAATGCTTAGC
Read r_3 : AACGTAATTTAGCTATCGGCTATCGTAAATGCTTA
Read r_4 : GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r_5 : GTATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r_6 : ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r_7 : TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r_8 : ATATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r_9 : ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r_{10} : CTATATAACGTAATTTAGCTATCGGCTATCGTAAA

Set of given reads

Read r_1 : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read $\bar{r}_2$: GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read $\bar{r}_3$: TAAGCATTTACGATAGCCGATAGCTAAATTACGTT
Read r_4 : GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read $\bar{r}_5$: CATTTACGATAGCCGATAGCTAAATTACGTTATAC
Read r_6 : ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r_7 : TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read $\bar{r}_8$: CATTTACGATAGCCGATAGCTAAATTACGTTATAT
Read r_9 : ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read $\bar{r}_{10}$: TTTACGATAGCCGATAGCTAAATTACGTTATATAG

Overlapping reads

Overlap graph construction
Transitive edge reduction
Composite path contraction
Flow computation
Tree reduction



Layout

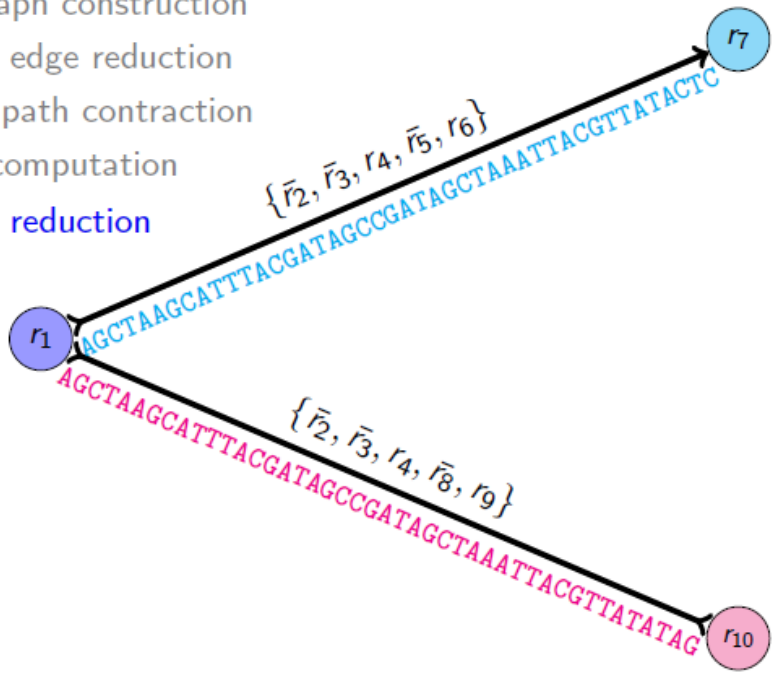
```
Read r1   :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2   :  CGTAATTTAGCTATCGGCTATCGTAAATGCTTAGC
Read r3   :  AACGTAATTTAGCTATCGGCTATCGTAAATGCTTA
Read r4   :  GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r5   :  GTATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r6   :  ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7   :  TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r8   :  ATATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r9   :  ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r10  :  CTATATAACGTAATTTAGCTATCGGCTATCGTAAA
```

Set of given reads

```
Read r1   :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r̄2   :   GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read r̄3   :    TAAGCATTTACGATAGCCGATAGCTAAATTACGTT
Read r4   :      GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r̄5   :       CATTTACGATAGCCGATAGCTAAATTACGTTATAC
Read r6   :        ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7   :         TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r̄8   :       CATTTACGATAGCCGATAGCTAAATTACGTTATAT
Read r9   :        ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r̄10  :         TTTACGATAGCCGATAGCTAAATTACGTTATATAG
```

Overlapping reads

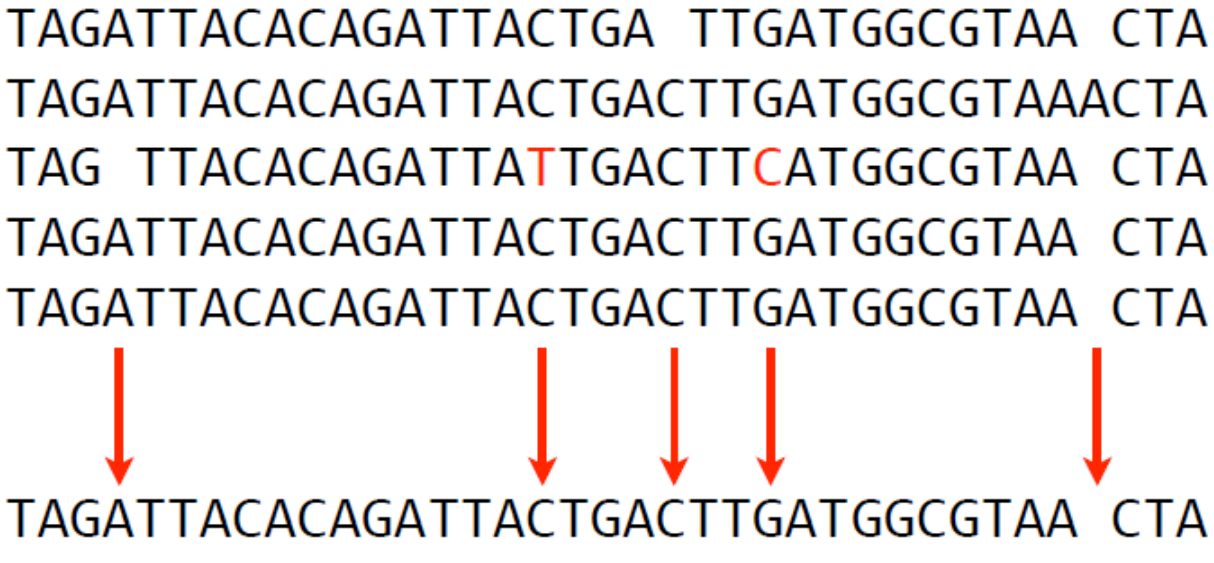
Overlap graph construction
Transitive edge reduction
Composite path contraction
Flow computation
Tree reduction




```
Contig 1:
AGCTAAGCATTTACGATAGCCGATAGCTAAATTACGTTATACTC
Contig 2:
AGCTAAGCATTTACGATAGCCGATAGCTAAATTACGTTATATAG
```

Contig is a set of overlapping DNA segments.

Consensus



Take reads that make up a contig and line them up

Take *consensus*, i.e. majority vote

At each position, ask: what nucleotide (and/or gap) is here?

Complications: (a) sequencing error, (b) ploidy

Say the true genotype is AG, but we have a high sequencing error rate and only about 6 reads covering the position.

Some OLC-based assemblers

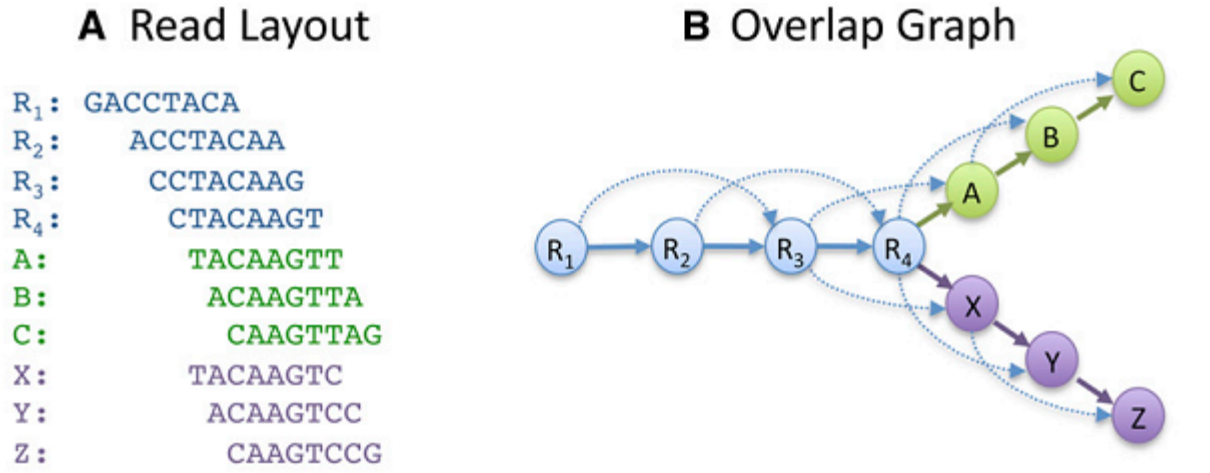
- Celera Assembler with the Best Overlap Graph (CABOG)
  - Designed for Sanger sequences, but works with 454 and error-corrected PacBio reads
- Newbler, a.k.a. GS *de novo* Assembler
  - Designed for 454 sequences, but works with Sanger reads
- Omega, overlap-graph *de novo* assembler for metagenomics
- Canu, scalable and accurate long-read assembly using MinHash

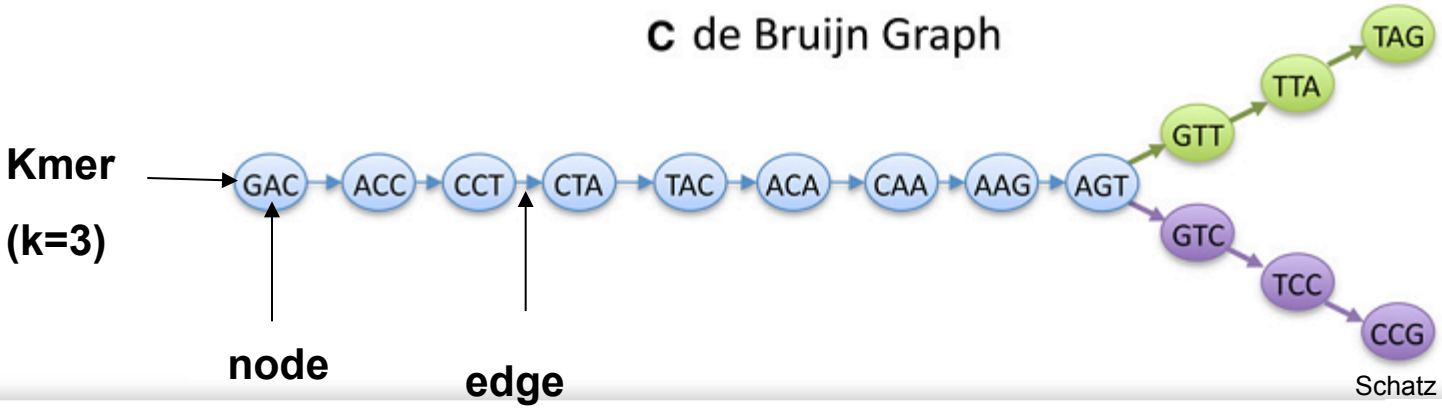
OLC drawbacks

- Building overlap graph is slow.
- Overlap graph is big; one node per read, and in practice # edges grows superlinearly with # reads
- 2nd-generation sequencing datasets are ~ 100s of millions or billions of reads, hundreds of billions of nucleotides total

Assembly outline

- **Kmer: a substring of defined length. For the purposes of this talk a substring of the sequence read**
- **de Bruijn graph: a graph representing overlaps between kmers**





Schatz et al., Genome Res. (2010)

k-mer

"k-mer" is a substring of length k

S: GGCGATTCATCG

A 4-mer of S: ATTC

All 3-mers of S:

GGC
GCG
CGA
GAT
ATT
TTC
TCA
CAT
ATC
TCG

mer: from Greek meaning "part"

I'll use "*k*-1-mer" to refer to a substring of length *k* - 1

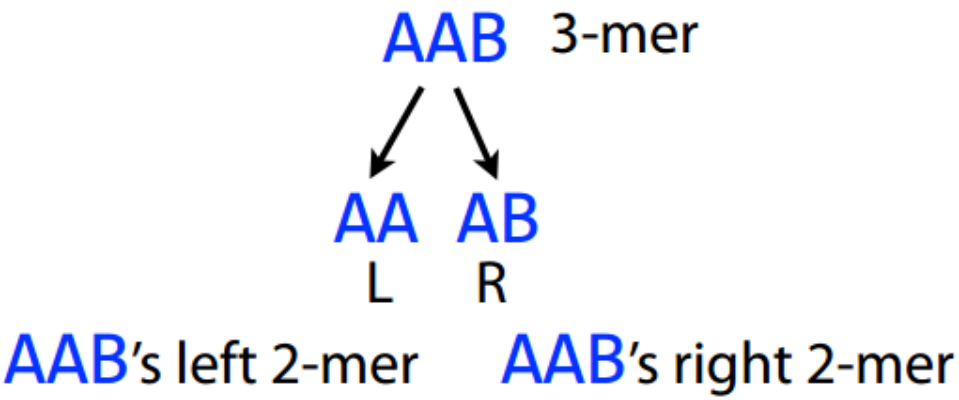
Courtesy of Ben Langmead. Used with permission.

De Bruijn graph

As usual, we start with a collection of reads, which are substrings of the reference genome.

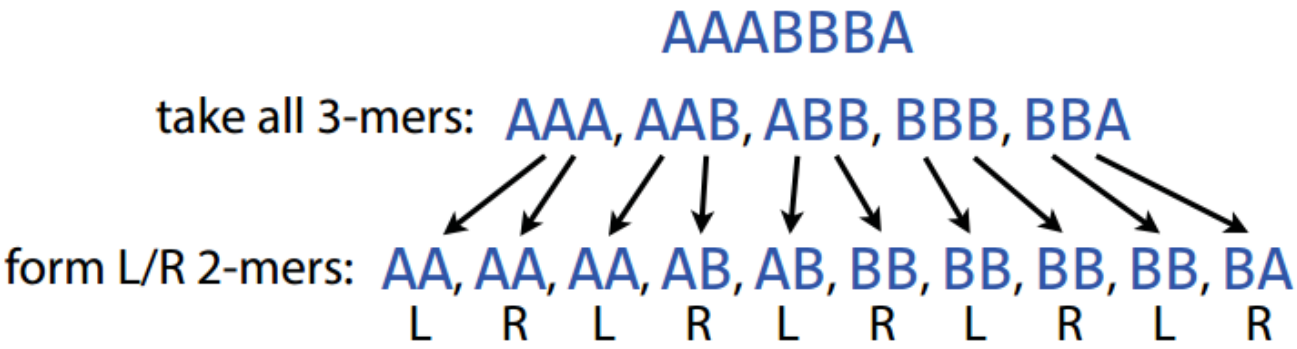
AAA, AAB, ABB, BBB, BBA

AAB is a *k*-mer (*k* = 3). AA is its *left* *k*-1-mer, and AB is its right *k*-1-mer.

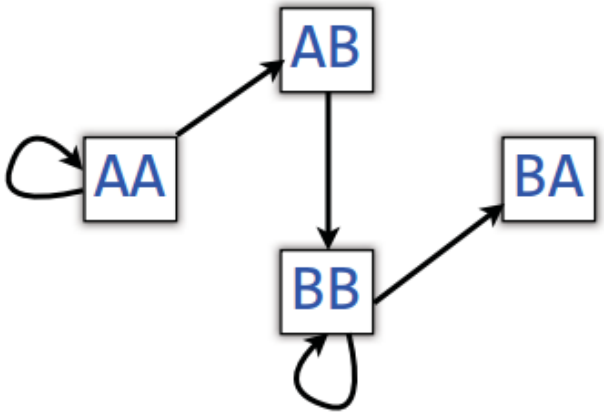


De Bruijn graph

Take each length-3 input string and split it into two overlapping substrings of length 2. Call these the *left* and *right 2-mers*.

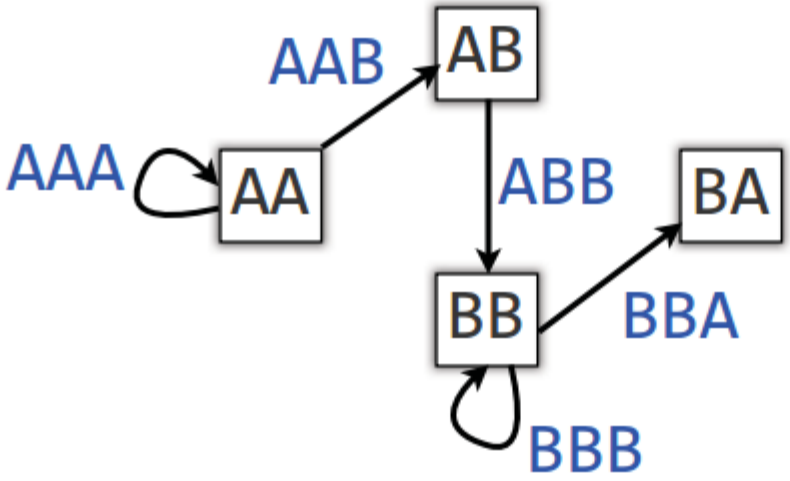


Let 2-mers be nodes in a new graph. Draw a directed edge from each left 2-mer to corresponding right 2-mer:



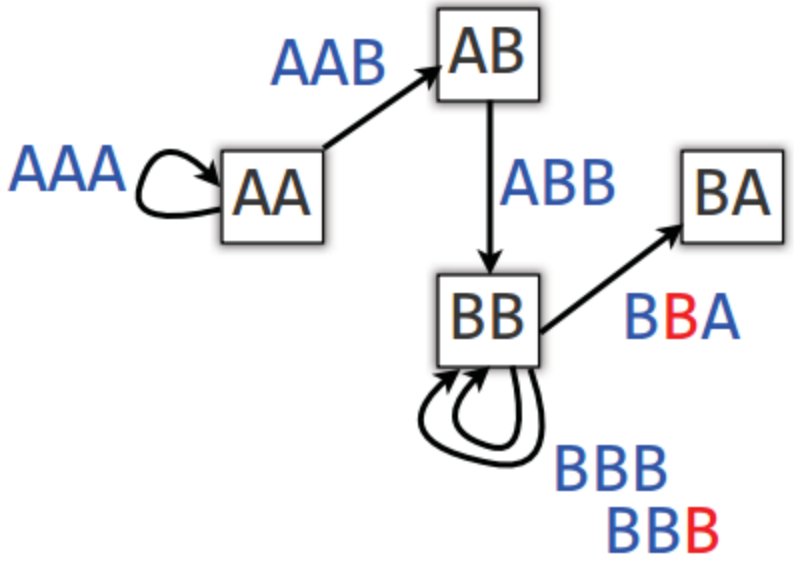
Each *edge* in this graph corresponds to a length-3 input string

De Bruijn graph



An edge corresponds to an overlap (of length *k*-2) between two *k*-1 mers.
More precisely, it corresponds to a *k*-mer from the input.

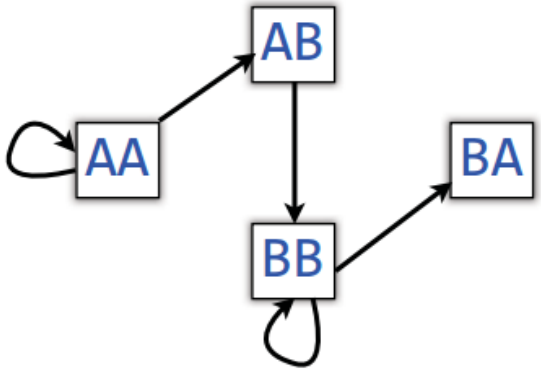
De Bruijn graph



If we add one more B to our input string: AAABBBBA, and rebuild the De Bruijn graph accordingly, we get a *multiedge*.

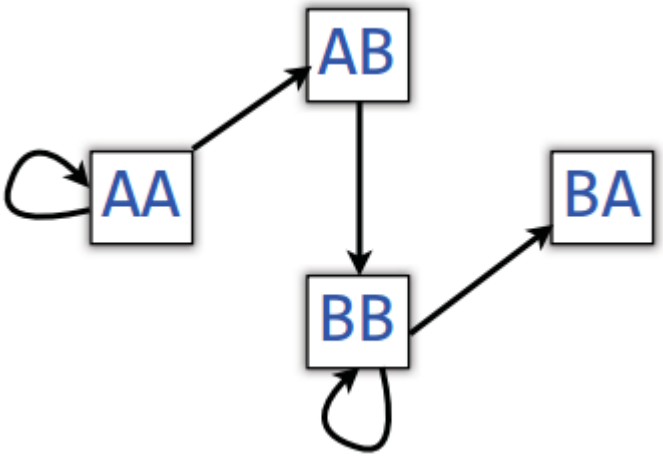
Eulerian walk definitions and statements

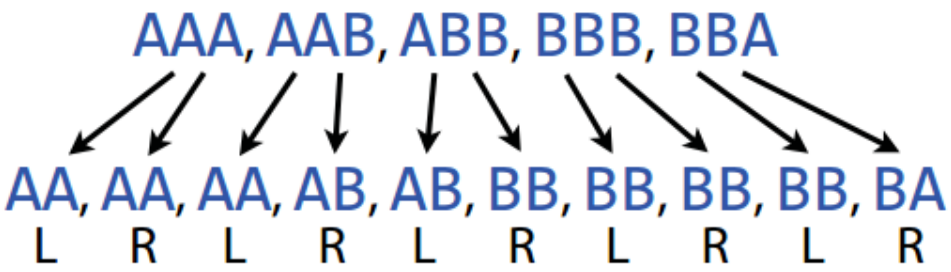
- Node is balanced if indegree equals outdegree
- Node is semi-balanced if indegree differs from outdegree by 1
- Graph is connected if each node can be reached by some other node
- Eulerian walk visits each edge exactly once
- Not all graphs have Eulerian walks. Graphs that do are Eulerian. (For simplicity, 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



De Bruijn graph

Back to our 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

A procedure for making a De Bruijn graph for a genome

Assume *perfect sequencing* where each length-*k* substring is sequenced exactly once with no errors

Pick a substring length *k*: 5

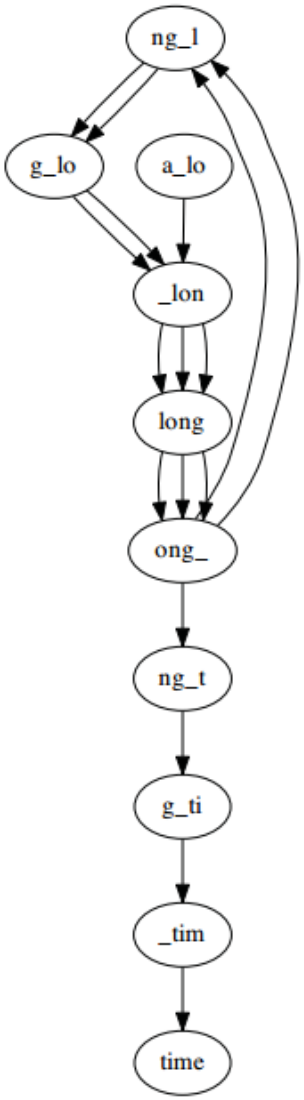
Start with each read: `a_long_long_long_time`

Take each *k* mer and split into left and right *k*-1 mers

`long_`

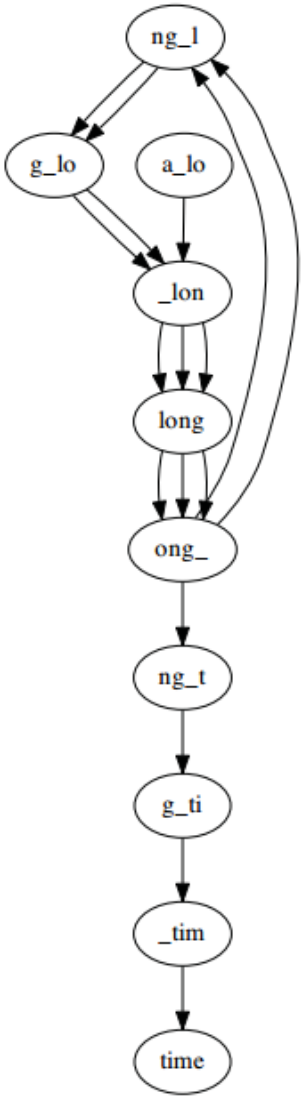
`long ong_`

Add k-1 mers as nodes to De Bruijn graph (if not already there), add edge from left *k*-1 mer to right *k*-1 mer

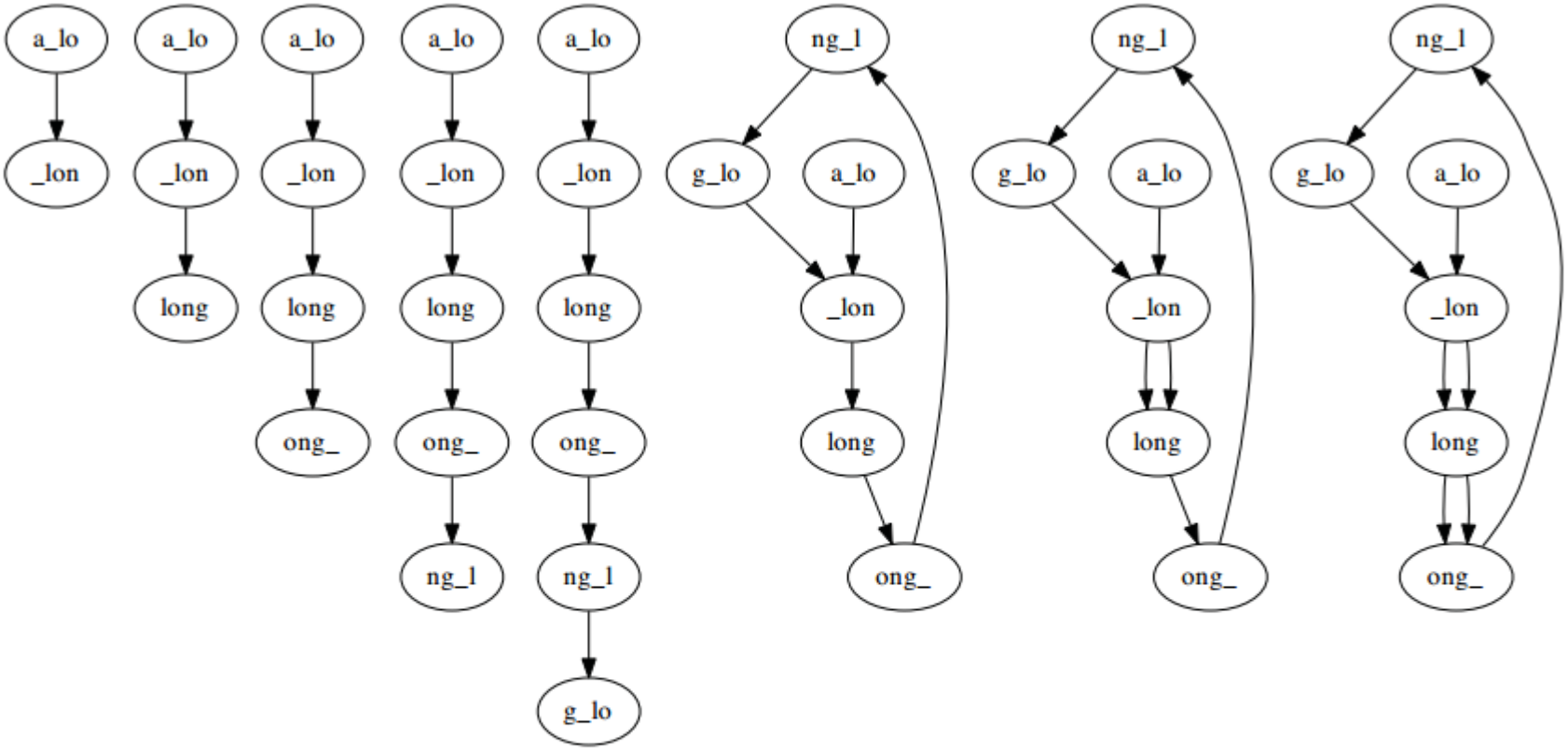


De Bruijn graph

- For genome assembly each k-mer is recorded in “twin” nodes – one node in the forward direction and one node in reve complement
- k is odd so no node can be its own reverse complement
- We will not show reverse complement twin nodes to cut down on clutter



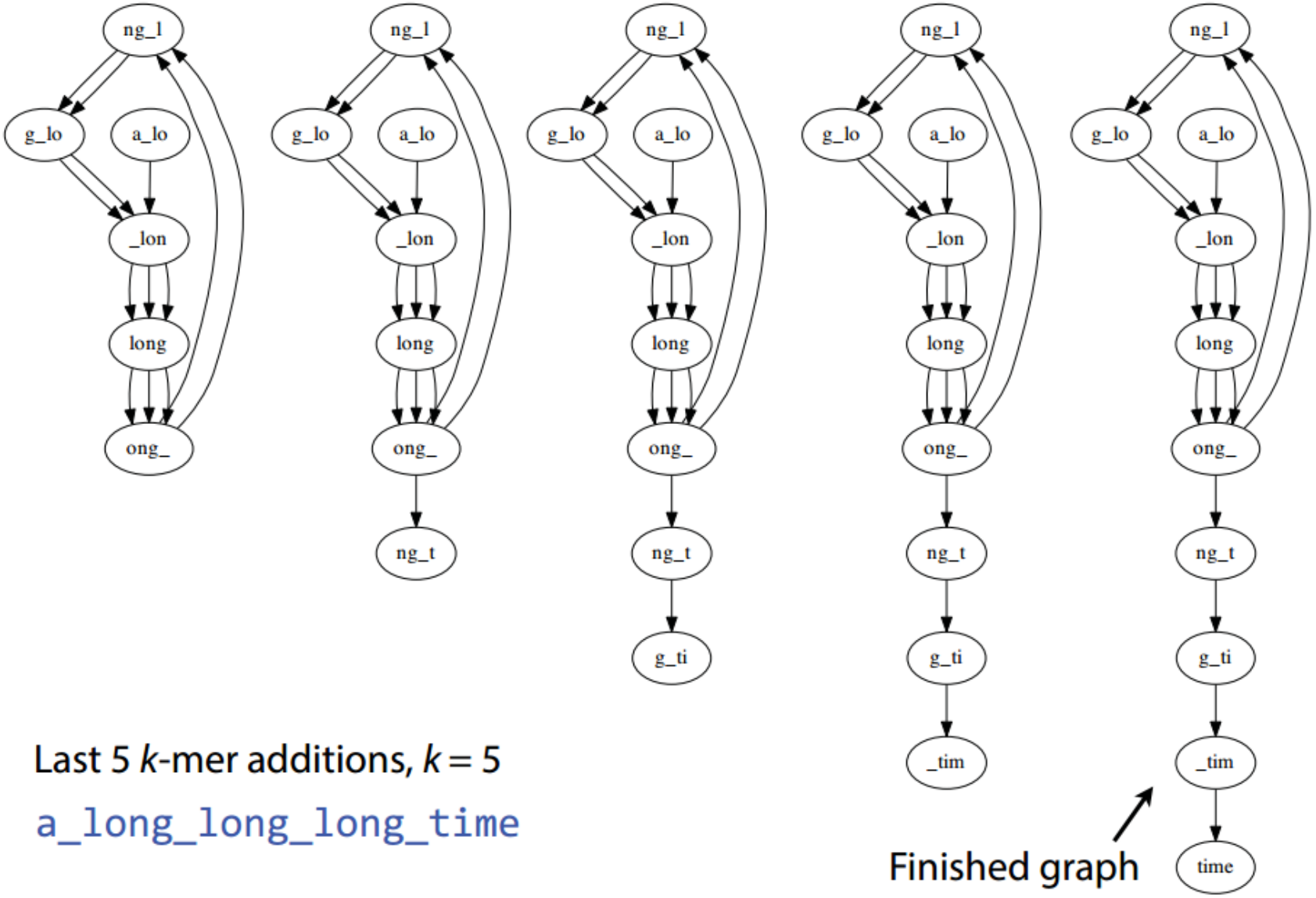
De Bruijn graph



First 8 *k*-mer additions, $k = 5$

```
a_long_long_long_time
```

De Bruijn graph



Last 5 *k*-mer additions, $k = 5$

a_long_long_long_time

Finished graph

De Bruijn graph

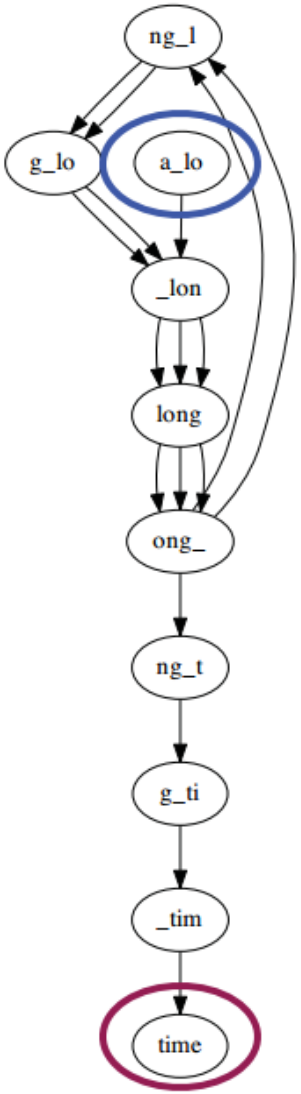
With perfect sequencing, this procedure always yields an Eulerian graph. Why?

Node for *k*-1-mer from left end is semi-balanced with one more outgoing edge than incoming *

Node for *k*-1-mer at right end is semi-balanced with one more incoming than outgoing *

Other nodes are balanced since # times *k*-1-mer occurs as a left *k*-1-mer = # times it occurs as a right *k*-1-mer

* Unless genome is circular



Repeats

No: graph can have multiple Eulerian walks, only one of which corresponds to original superstring

Right: graph for ZABCDABEFABY, $k = 3$

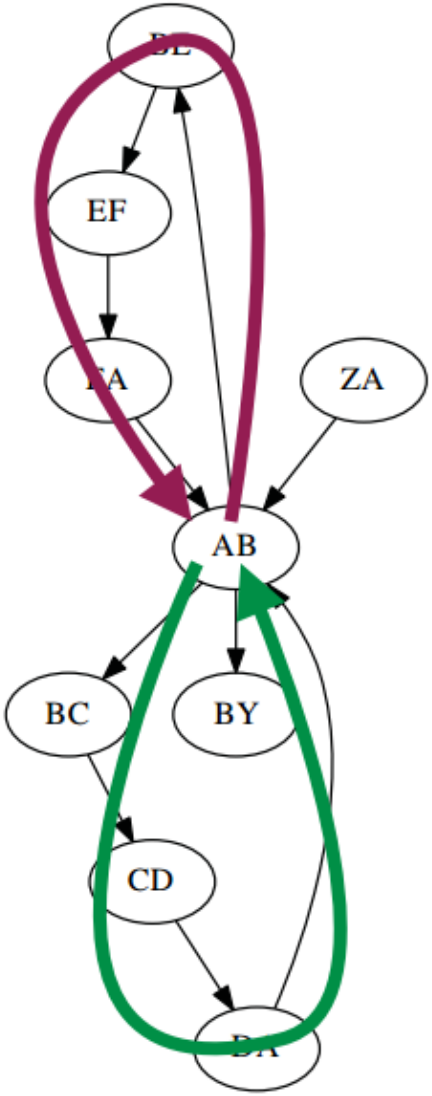
Alternative Eulerian walks:

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

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

These correspond to two edge-disjoint directed cycles joined by node AB

AB is a repeat: ZABCDABEFABY

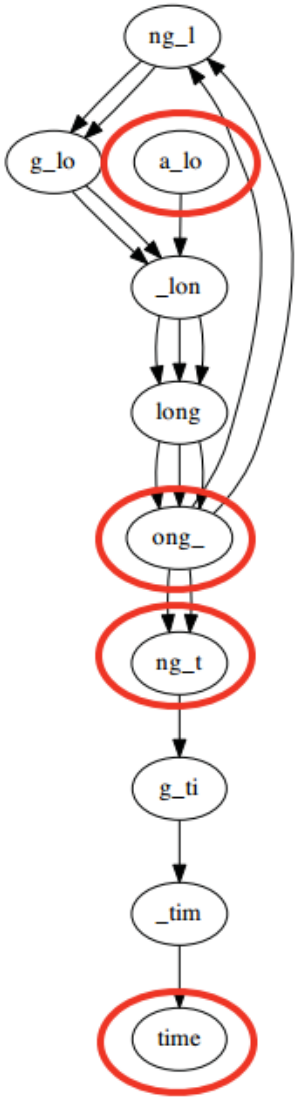


Differences in coverage

Differences in coverage also lead to non-Eulerian graph

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

Graph has 4 semi-balanced nodes, isn't Eulerian

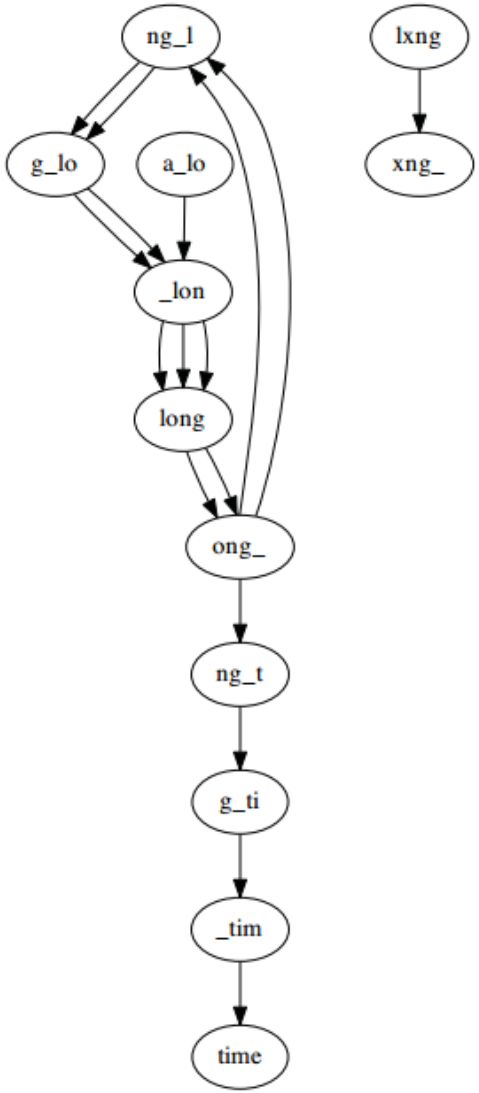


Errors and differences

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 a copy of `long_` into `lxng_`

Graph is not connected; largest component is not Eulerian



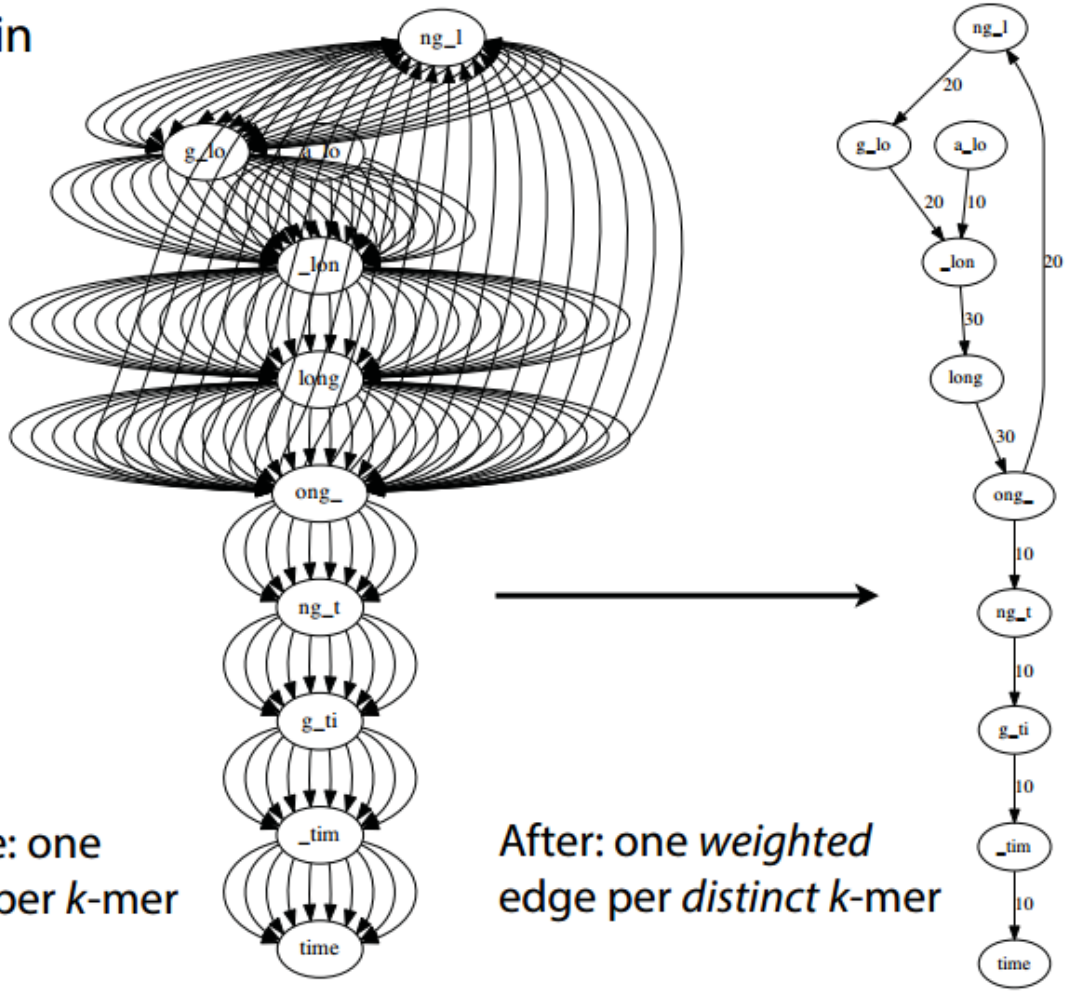
De Bruijn graph

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

Same edge might appear in dozens of copies; let's use edge *weights* instead

Weight = # times *k*-mer occurs

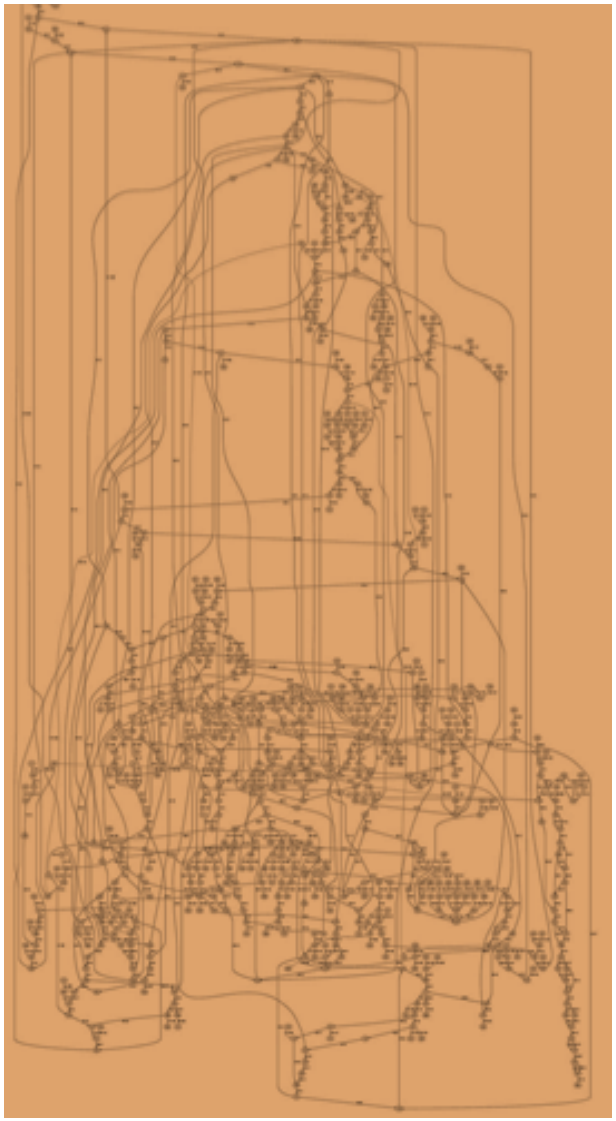
Using weights, there's one *weighted* edge for each *distinct k*-mer



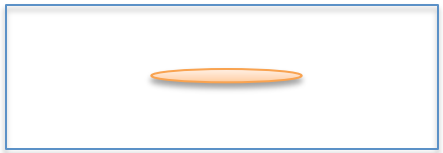
Before: one edge per *k*-mer

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

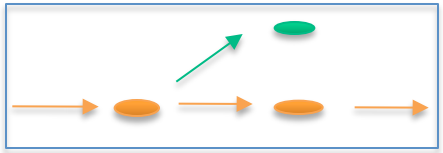
Node Types



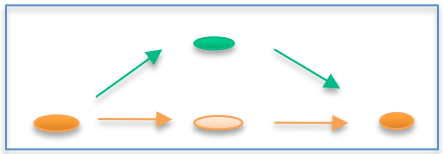
(Chaisson, 2009)



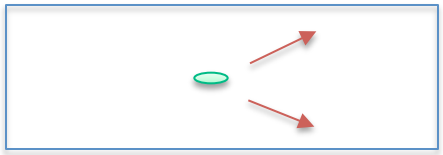
Isolated nodes (10%)



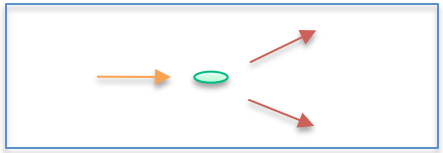
Tips (46%)



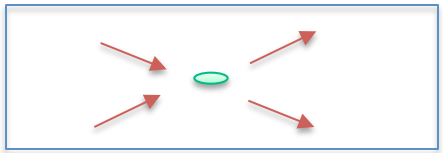
Bubbles/Non-branch (9%)



Dead Ends (.2%)



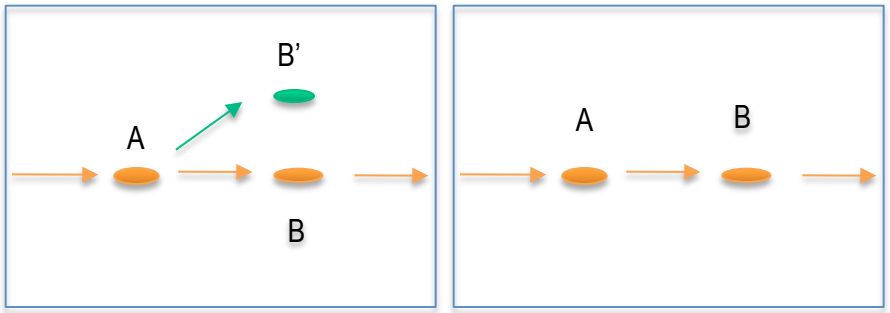
Half Branch (25%)



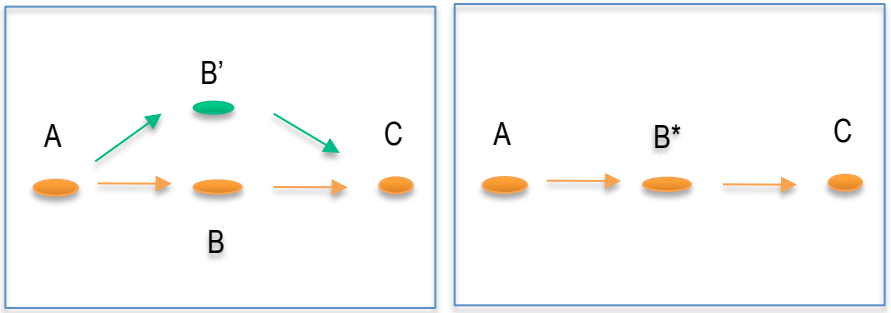
Full Branch (10%)

Error Correction

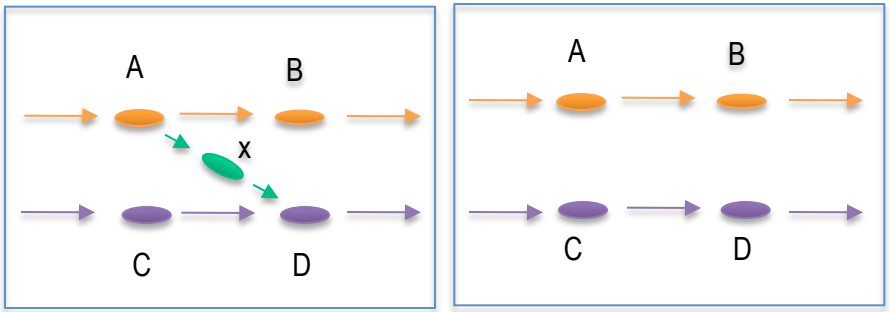
- Errors at end of read
  - Trim off ‘dead-end’ tips



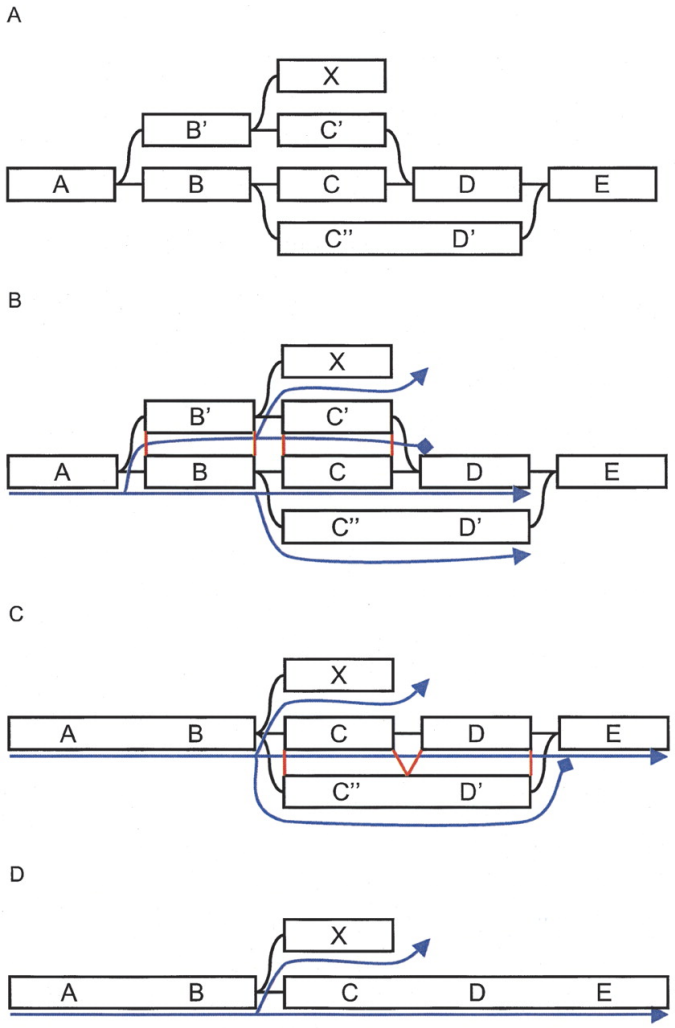
- Errors in middle of read
  - Pop Bubbles



- Chimeric Edges
  - Clip short, low coverage nodes



Example of Tour Bus error correction in Velvet.



Zerbino D R , Birney E Genome Res. 2008;18:821-829

Example of Tour Bus error correction. (A) Original graph. (B) The search starts from A and spreads toward the right. The progression of the top path (through B' and C') is stopped because D was previously visited. The nucleotide sequences corresponding to the alternate paths B'C' and BC are extracted from the graph, aligned, and compared. (C) The two paths are judged similar, so the longer one, B'C', is merged into the shorter one, BC. The merging is directed by the alignment of the consensus sequences, indicated in red lines in B. Note that node X, which was connected to node B', is now connected to node B. The search progresses, and the bottom path (through C' and D') arrives second in E. Once again, the corresponding paths, C'D' and CD are compared. (D) CD and C'D' are judged similar enough. The longer path is merged into the shorter one.

Handling Repeats

1. Repeat detection
   - **pre-assembly:** find fragments that belong to repeats
     - statistically (most existing assemblers)
     - repeat database (*RepeatMasker*)
   - **during assembly:** detect "tangles" indicative of repeats (Pevzner, Tang, Waterman 2001)
   - **post-assembly:** find repetitive regions and potential mis-assemblies.
     - *Reputer, RepeatMasker*
     - "unhappy" mate-pairs (too close, too far, mis-oriented)
2. Repeat resolution
   - find DNA fragments belonging to the repeat
   - determine correct tiling across the repeat
   - Obtain long reads spanning repeats

De Bruijn graph

What are the limitations of De Bruijn graphs?

Reads are immediately split into shorter *k*-mers; can't resolve repeats as well as overlap graph

Only a very specific type of "overlap" is considered, which makes dealing with errors more complicated.

Read coherence is lost. Some paths through De Bruijn graph are inconsistent with respect to input reads. Need to thread reads though De Bruijn graph to recover information lost when reads are fragmented into k-mers.

This is the OLC ↔ DBG tradeoff

Single most important benefit of De Bruijn graph is speed and simplicity.

Example Result

	OLC t >= 75	De Bruijn k = 61	De Bruijn k = 67	De Bruijn k = 59

Table 1. Assembly statistics for *C. elegans* data set

	SGA	Velvet	ABySS	SOAPdenovo
Scaffold N50 size	26.3 kbp	31.3 kbp	23.8 kbp	31.1 kbp
Aligned contig N50 size	16.8 kbp	13.6 kbp	18.4 kbp	16.0 kbp
Mean aligned contig size	4.9 kbp	5.3 kbp	6.0 kbp	5.6 kbp
Sum aligned contig size	96.8 Mbp	95.2 Mbp	98.3 Mbp	95.4 Mbp
Reference bases covered	96.2 Mbp	94.8 Mbp	95.9 Mbp	95.1 Mbp
Reference bases covered by contigs ≥1 kb	93.0 Mbp	92.1 Mbp	93.9 Mbp	92.3 Mbp
Mismatch rate at all assembled bases	1 per 21,545 bp	1 per 8786 bp	1 per 5577 bp	1 per 26,585 bp
Mismatch rate at bases covered by all assemblies	1 per 82,573 bp	1 per 18,012 bp	1 per 8209 bp	1 per 81,025 bp
Contigs with split/bad alignment (sum size)	458 (4.4 Mbp)	787 (7.2 Mbp)	638 (9.1 Mbp)	483 (4.4 Mbp)
Total CPU time	41 h	2 h	5 h	13 h
Max memory usage	4.5 GB	23.0 GB	14.1 GB	38.8 GB

100 MBase genome, 33.8M read pairs, 100bp reads each end, 250bp insert size

Genome Evaluation/Analysis Papers

- GAGE: A critical evaluation of genome assemblies and assembly algorithms

- GAGE-B: an evaluation of genome assemblers for bacterial organisms