CS 68: BIOINFORMATICS

Prof. Sara Mathieson
Swarthmore College
Spring 2018

Outline: Jan 31

- DBG assembly in practice
- Velvet assembler
- Evaluation of assemblies
- (if time) Start: string alignment

Candidate job talk:
Today! 11:30-12:20

Notes:

- Lab 1 due tonight (fill out README)
- Assembly reading posted: spend 1.5 hours max
- Office hours today 1-3pm
- Fill out partner form for Lab 2 (if you know who you want to work with)

Recap: Creating Eulerian paths

Recap Fleury's Algorithm

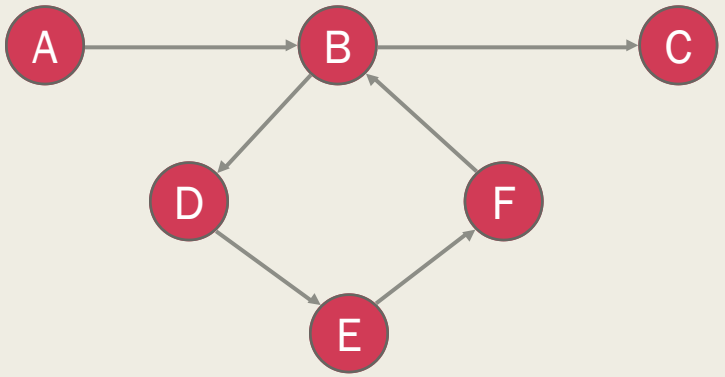
- Start with vertex u where (outdegree – indegree) = 1
- Until there are no more edges:

  Choose $e = (u,v)$ where removing e will not disconnect the graph (if possible)
  Traverse e, delete e, and proceed with $u = v$

Recap Fleury's Algorithm

- Start with vertex ***u*** where (outdegree – indegree) = 1
- Until there are no more edges:

  Choose ***e = (u,v)*** where removing ***e*** will not disconnect the graph (if possible)
  Traverse ***e***, delete ***e***, and proceed with ***u = v***



Avoids going from:
A -> B -> C
And then getting stuck

Recap recursive algorithm

```
find_cycle(u, cycle):
        for each edge e = (u,v):
                remove e
                find_cycle(v, cycle)
        push u onto cycle
```

<u>To use in practice</u>:

- Start with cycle as an empty stack
- Start at any node u (or start at a node with (outdegree – indegree) = 1 for a path)
- After completion, pop elements off cycle to find the correct order

De Bruijn graphs in practice

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

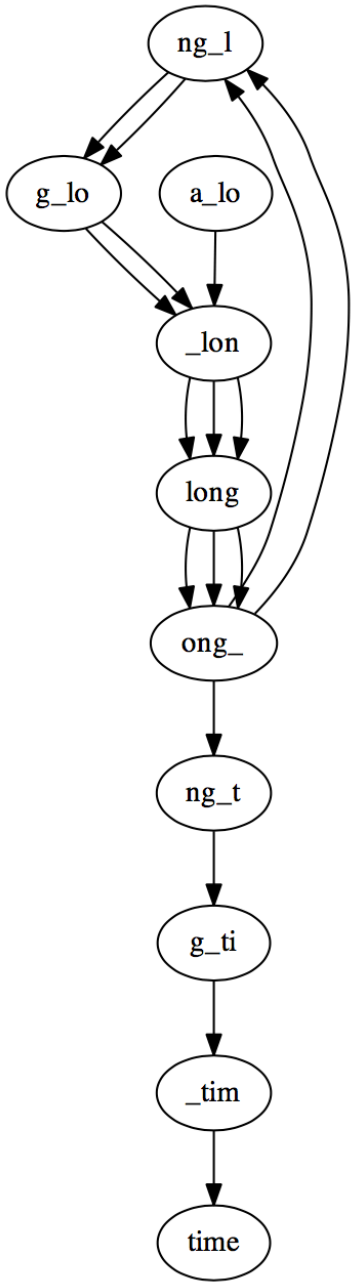
Start with an input string: `a_long_long_long_time`

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



Slide: adapted from Ben Langmead, John Hopkins

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

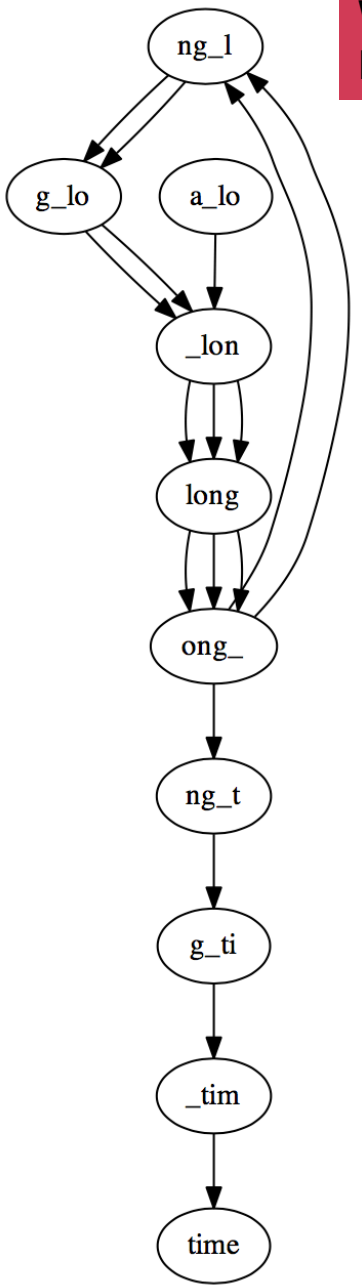
Start with an input string: `a_long_long_long_time`

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



Will always give an Eulerian graph. Why?

Slide: adapted from Ben Langmead, John Hopkins

Building k-mer graph with reads

$n = 6x10^9$ reads
$L = 100$ bp
$G = 3x10^9$ bp (humans)

* Pick k (for L = 100 bp, k = 20-40 is common)
* For each read:
    For each k-mer in read:
        - add L&R (k-1)-mers to graph as nodes (if not already there)
        - add edge from L -> R

From last time: # k-mers is **O(G)** => # nodes is **O(G)**, and # edges is **O(G)**

But: how do we know if (k-1)-mer is already in our graph or not? Do we have to compare with all nodes?

Implementation considerations

1) In practice, *k*-mers must be hashed so we can easily compare them

2) When graphs become larger, recursive solutions are no longer practical

What “messes” up our DBG?

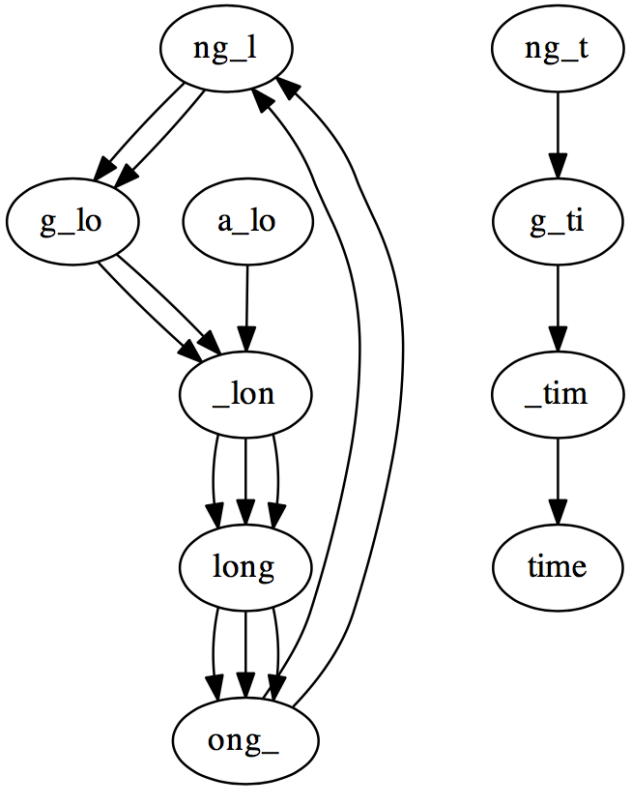
1) Repeats of length (*k*-1) or longer <- correction!
2) Gaps in coverage
3) Differences in coverage
4) Sequencing errors

Handout 2: work with someone new

Issues with DBGs

Gaps in coverage can lead to *disconnected* graph

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



Connected components are individually Eulerian, overall graph is not

Slide: adapted from Ben Langmead, John Hopkins

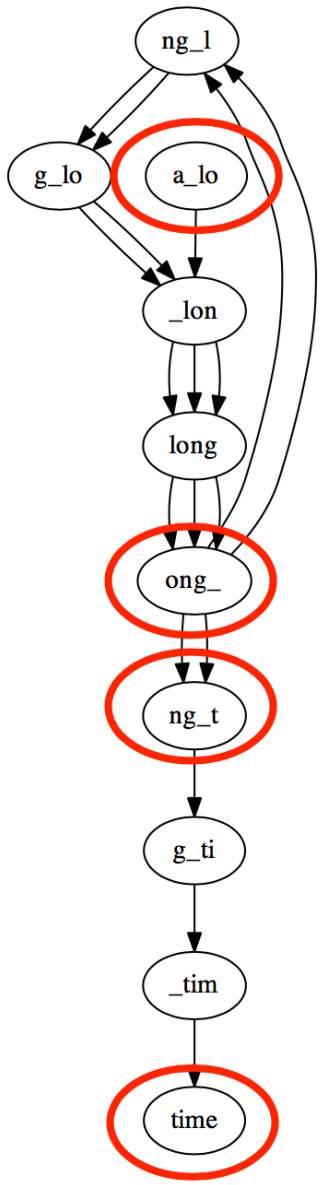
Issues with DBGs

De Bruijn graph

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



Slide: adapted from Ben Langmead, John Hopkins

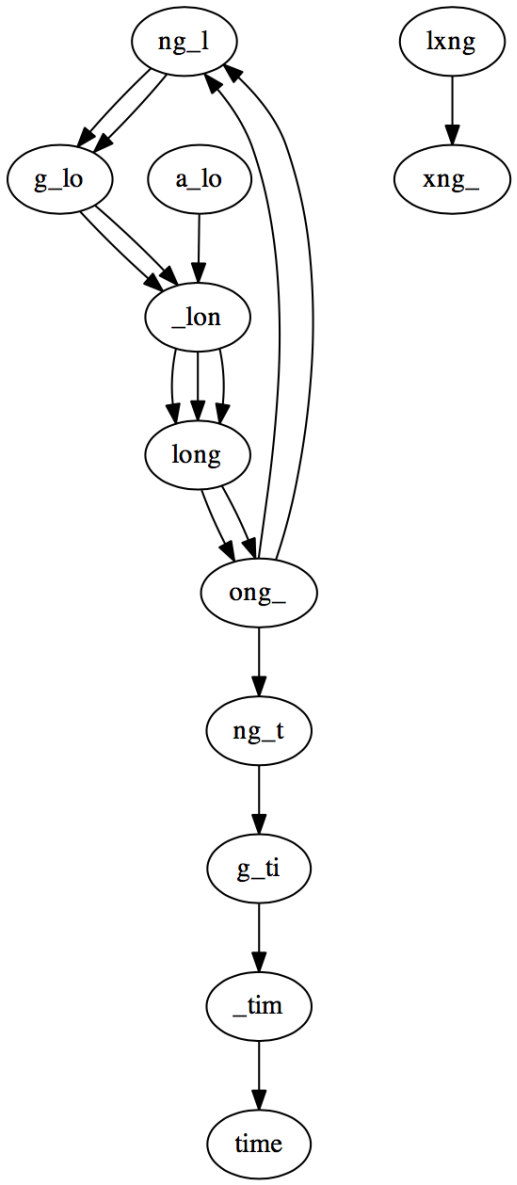
Issues with DBGs

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 a copy of `long_` into `lxng_`

Graph is not connected; largest component is not Eulerian



Slide: adapted from Ben Langmead, John Hopkins

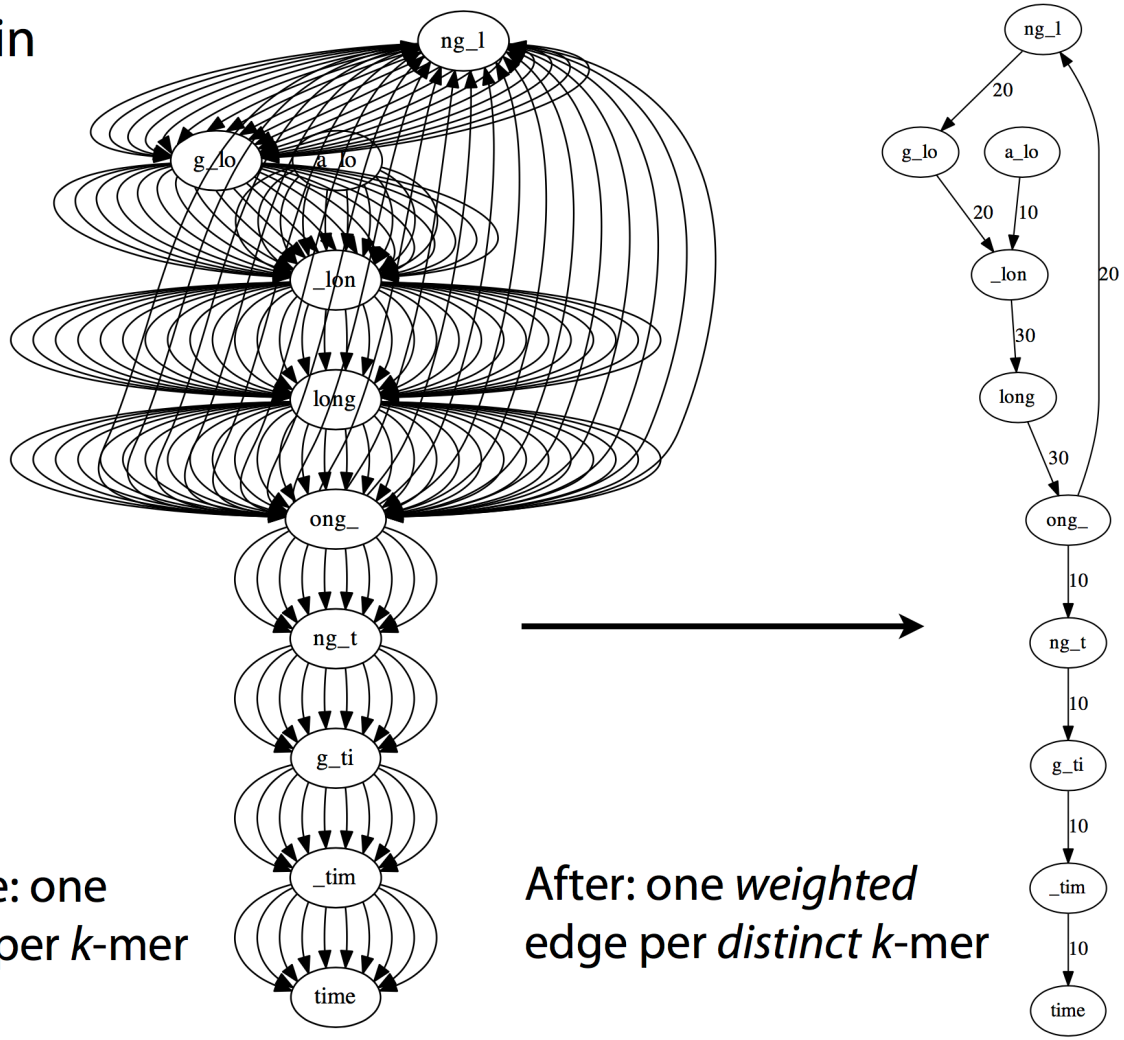
One workaround for coverage issues:

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

Slide: adapted from Ben Langmead, John Hopkins

Issues with DBGs

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) is an improved formulation where we seek 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.

Slide: adapted from Ben Langmead, John Hopkins

What did we give up by going from OLC to DBG?

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.

Slide: adapted from Ben Langmead, John Hopkins

But we still have advantages…

Building the de Bruijn graph:

- O(nL) since we go through each read and k-mers along the length of the read
- O(G) space to store since both # edges and # nodes are O(G)

Finding paths through the graph:

- assuming Eulerian, O(G) to traverse since # edges is O(G)

Velvet Assembler

Velvet Assembler (Zerbino & Birney, 2008)

- The first truly practical de Bruijn graph assembler
- Combines several algorithms to simplify the raw k-mer graph

Resource

Velvet: Algorithms for de novo short read assembly using de Bruijn graphs

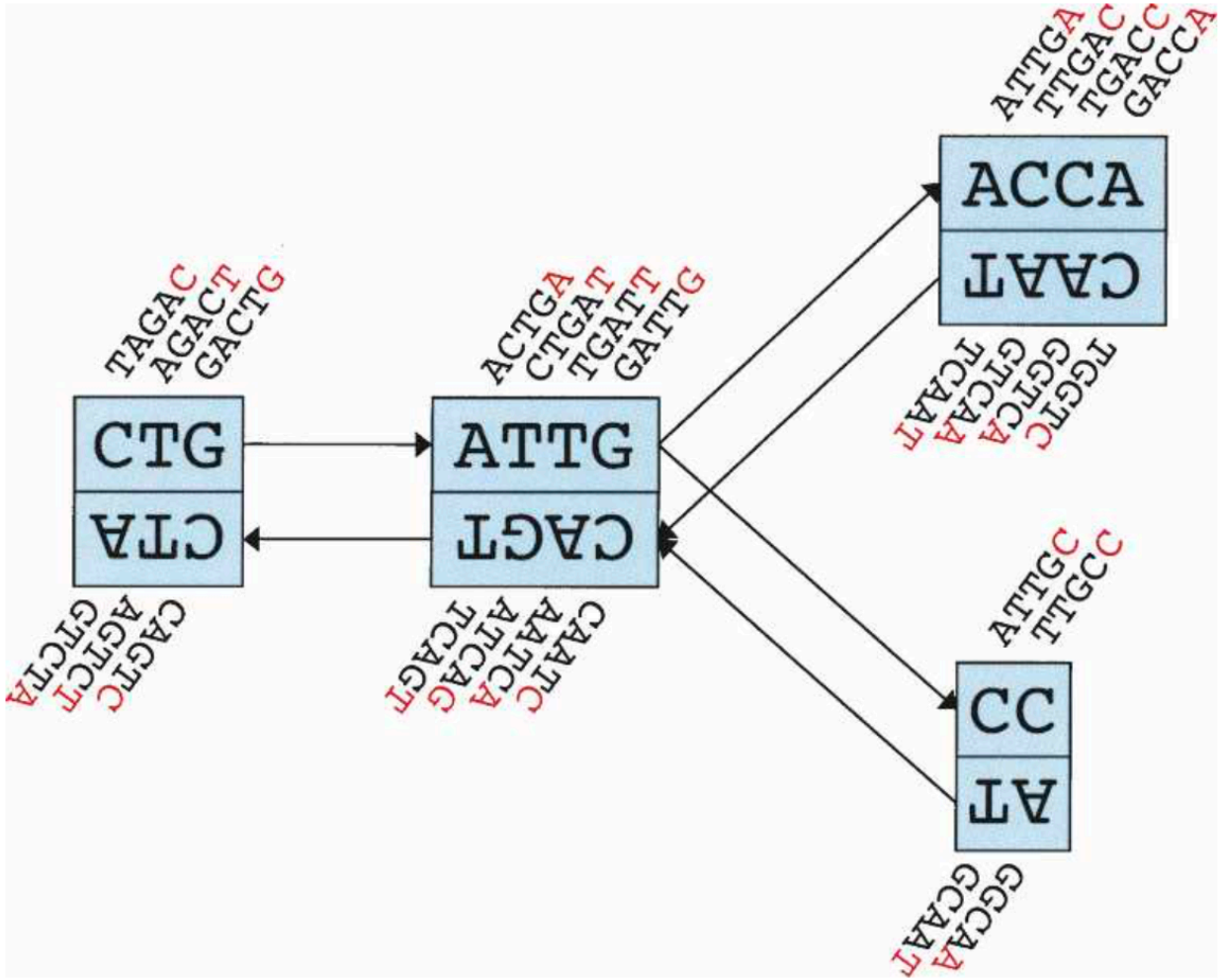
Daniel R. Zerbino and Ewan Birney[1]

EMBL-European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge CB10 1SD, United Kingdom

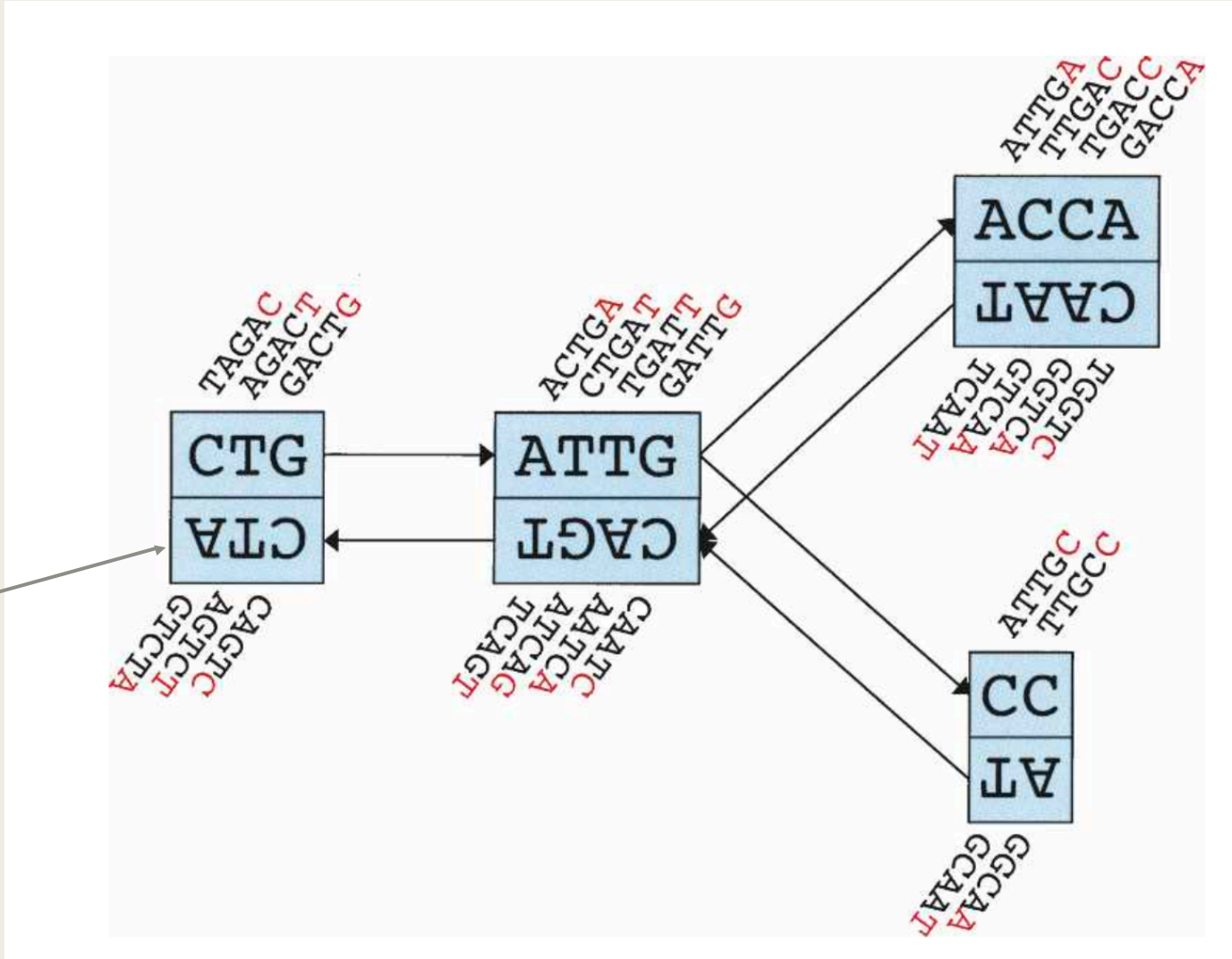
We have developed a new set of algorithms, collectively called "Velvet," to manipulate de Bruijn graphs for genomic sequence assembly. A de Bruijn graph is a compact representation based on short words (*k*-mers) that is ideal for high coverage, very short read (25–50 bp) data sets. Applying Velvet to very short reads and paired-ends information only, one can produce contigs of significant length, up to 50-kb N50 length in simulations of prokaryotic data and 3-kb N50 on simulated mammalian BACs. When applied to real Solexa data sets without read pairs, Velvet generated contigs of ~8 kb in a prokaryote and 2 kb in a mammalian BAC, in close agreement with our simulated results without read-pair information. Velvet represents a new approach to assembly that can leverage very short reads in combination with read pairs to produce useful assemblies.

[Supplemental material is available online at www.genome.org. The code for Velvet is freely available, under the GNU Public License, at http://www.ebi.ac.uk/~zerbino/velvet.]

Velvet paper: Figure 1



Velvet paper: Figure 1

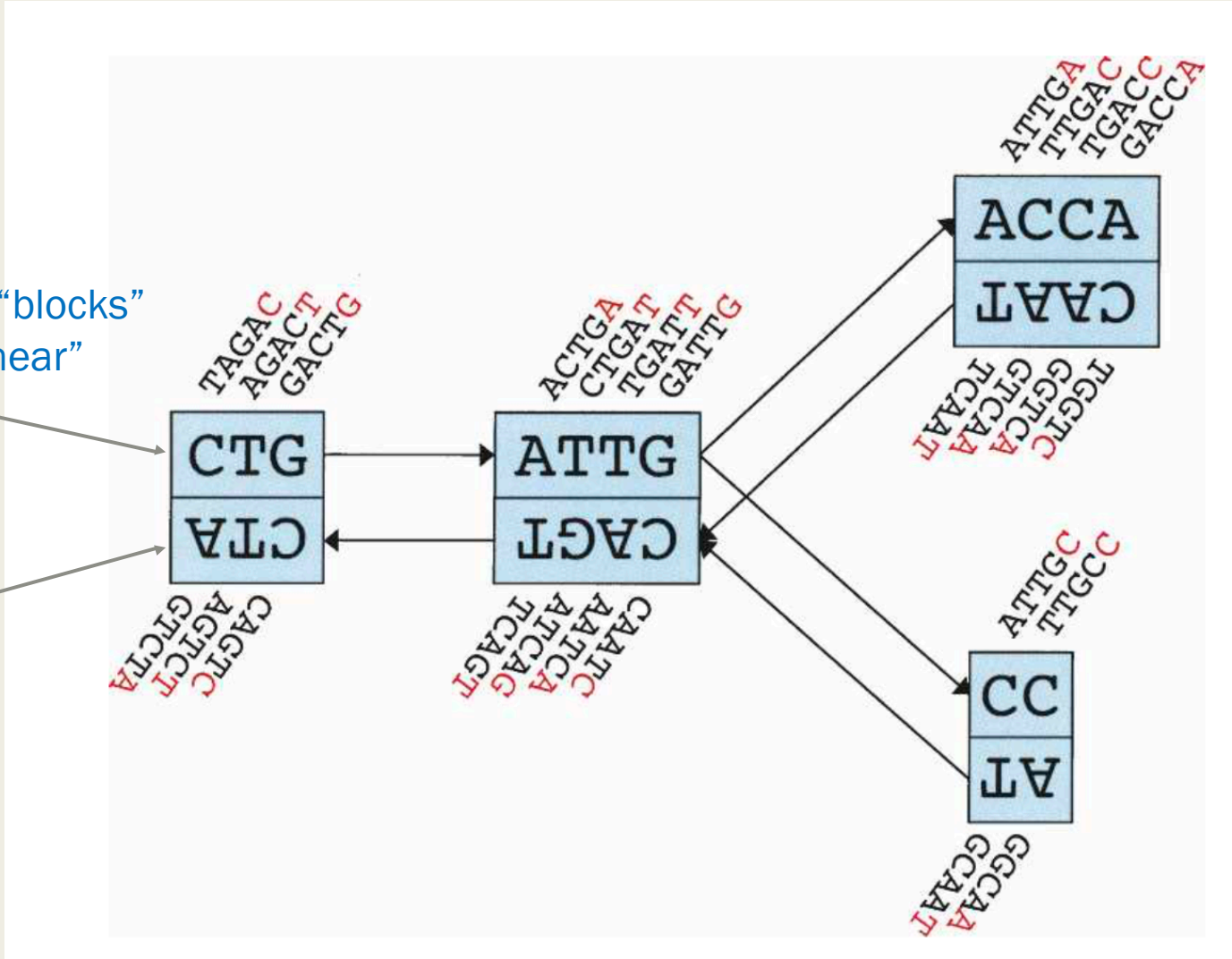


Reverse complement

Velvet paper: Figure 1

Combine nodes into “blocks” when graph is “linear”

Reverse complement

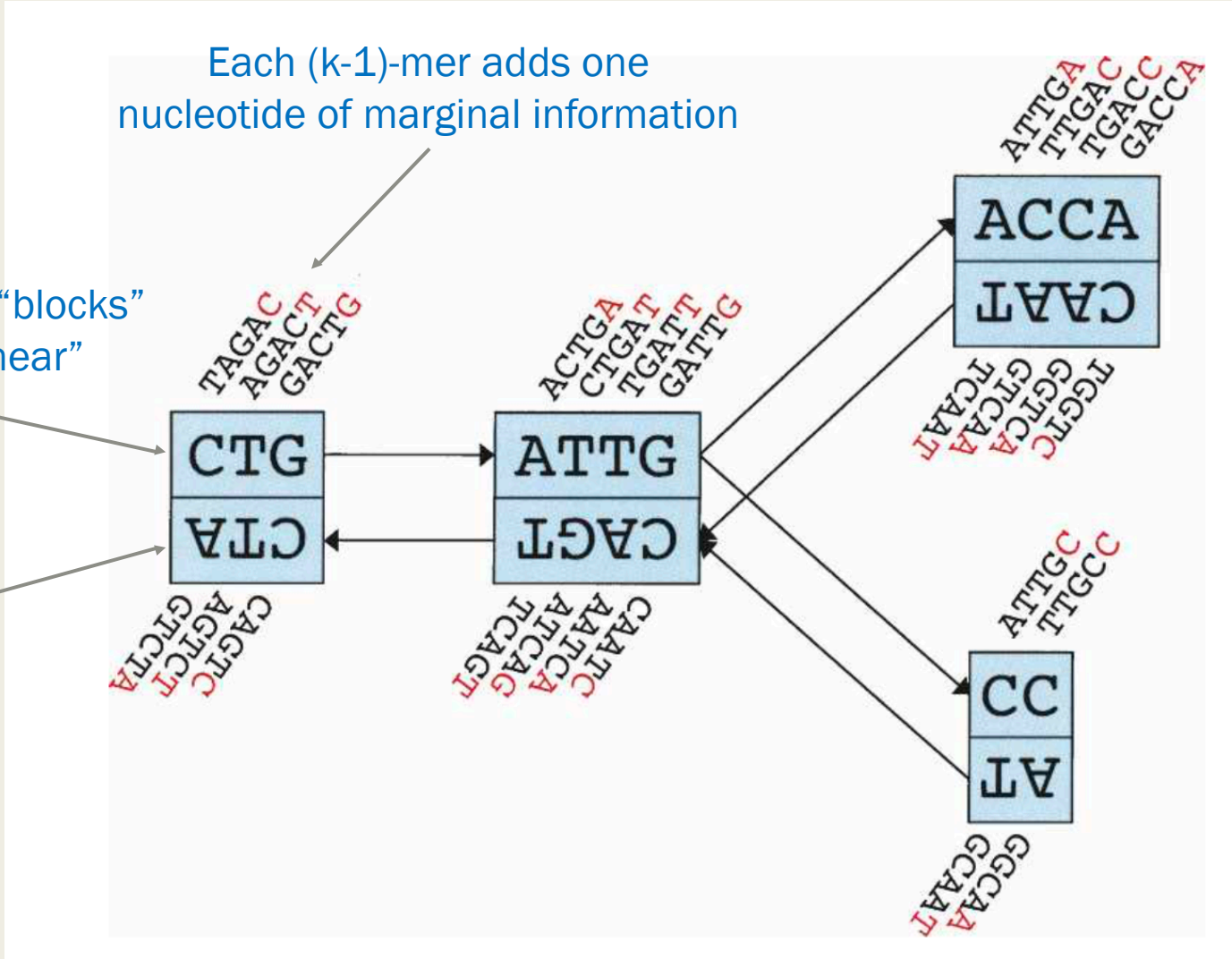


Velvet paper: Figure 1

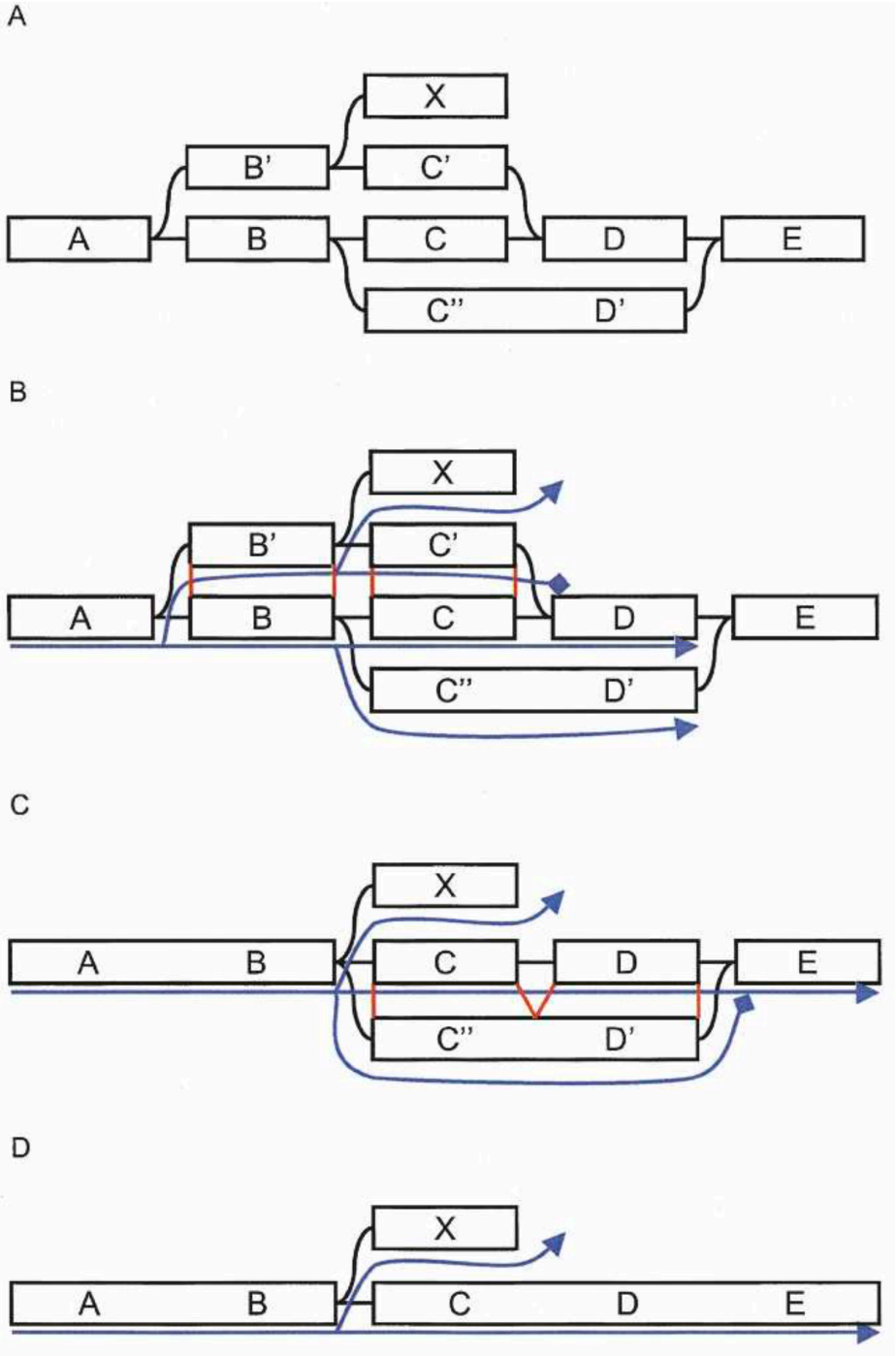
Each (k-1)-mer adds one nucleotide of marginal information

Combine nodes into “blocks” when graph is “linear”

Reverse complement

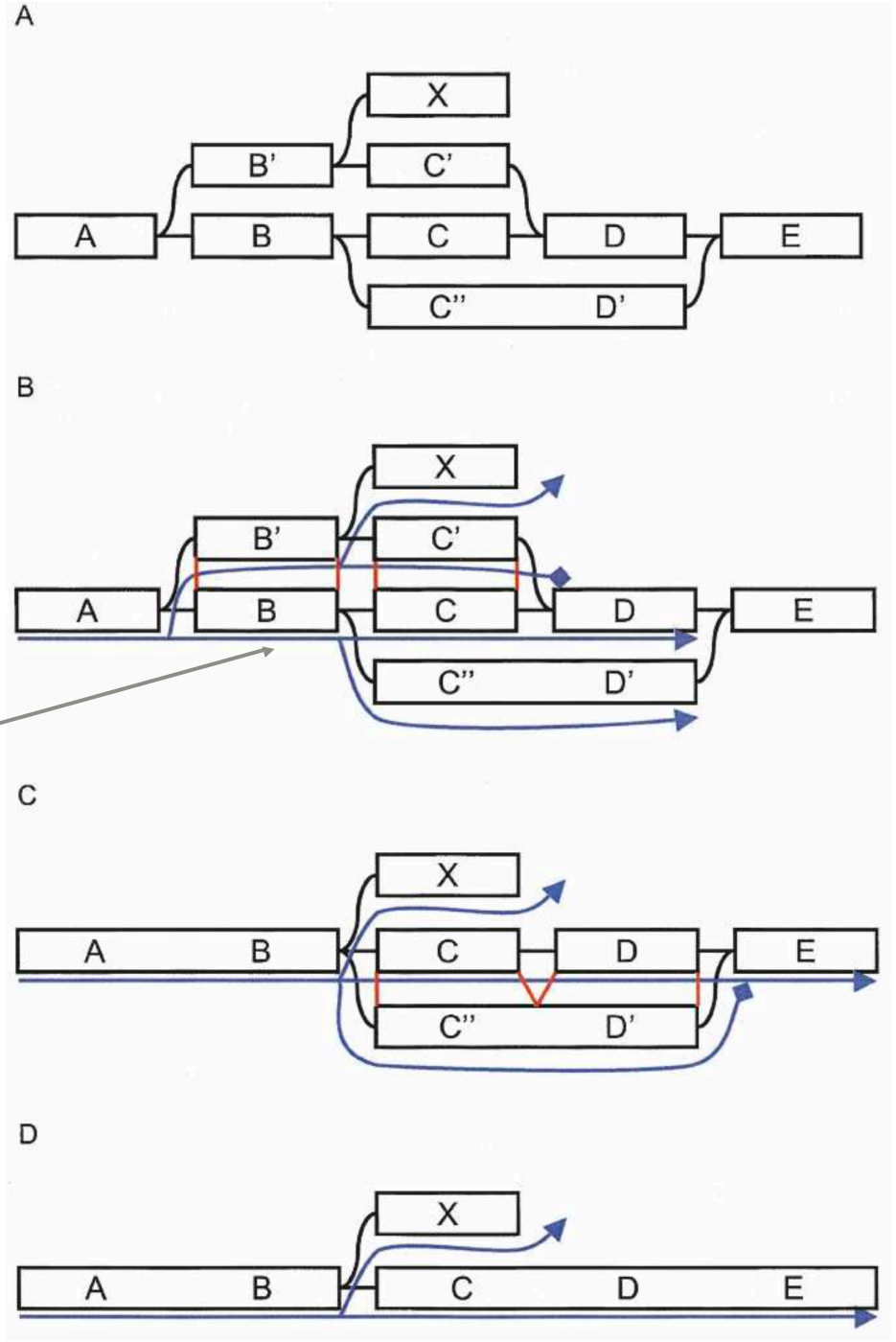


Velvet paper: Figure 2 (Tour Bus algorithm)



Velvet paper: Figure 2 (Tour Bus algorithm)

When we hit D for the second time, compare the sequences of the two ways we got there: BC and B'C'. If they are judged similar, error correct and merge



Evaluating Assemblies

Assembly evaluation

- **N50**: for a set of contigs, N50 is the greatest length such that at least half the bases of the assembly are in a contig with length N50 or longer

1a. Contigs, sorted according to their lengths.

Image: The Molecular Ecologist

Assembly evaluation

- **N50**: for a set of contigs, N50 is the greatest length such that at least half the bases of the assembly are in a contig with length N50 or longer

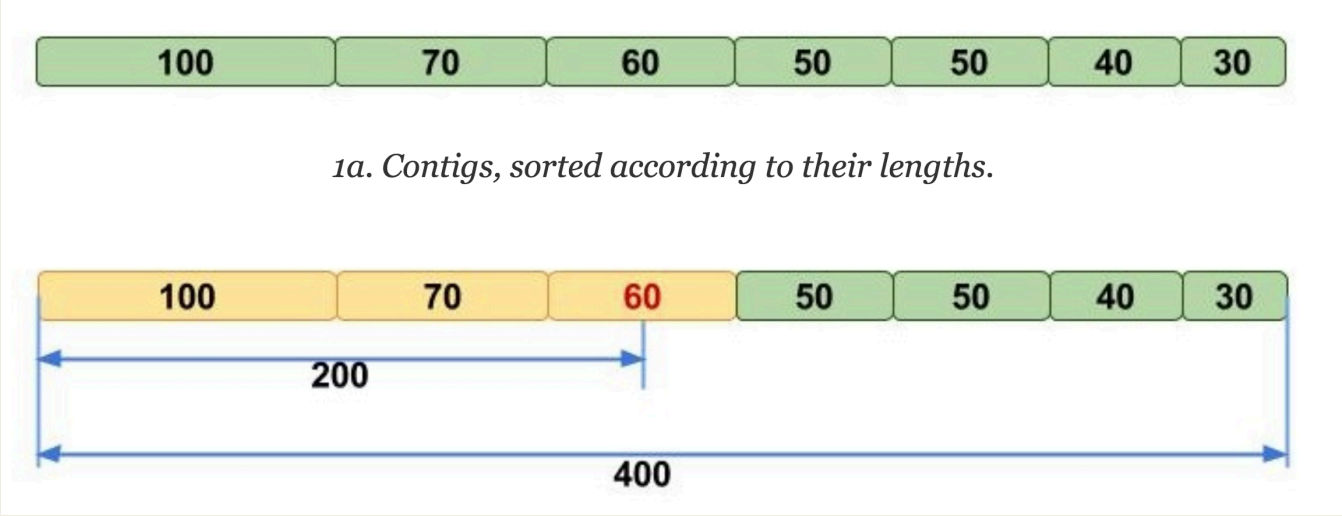


Image: The Molecular Ecologist

Why is N50 a bad evaluation metric?

Why is N50 a bad evaluation metric?

- We could just loop through cycles in our a graph over and over, generating large (incorrect) contigs
- We need a better way to evaluate the quality of assemblies
- Take away: simulated data is every valuable. Take an existing genome, simulate random reads, then try to reconstruct.
- How do we tell how similar our reconstruction is to the original genome we simulated from?

Next topic: sequence alignment

- Goal: given two sequences, what is the best match or “alignment” between them?