2-assembly-part2

Monday, January 19, 2026 6:55 PM

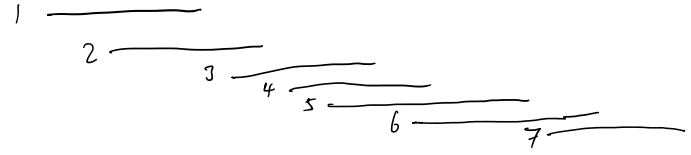
Last time we discussed the assembly problem & some problems with shortest common superstring. Today, we will discuss three practical approaches

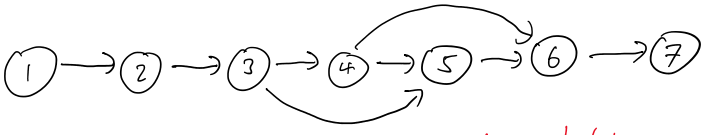
Assembly Strategy 1: Overlap - Layout - Consensus [Celera, 2000]

Overlap graph

Nodes = reads

Edges = overlaps (directional)

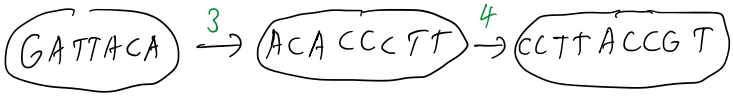




easy example

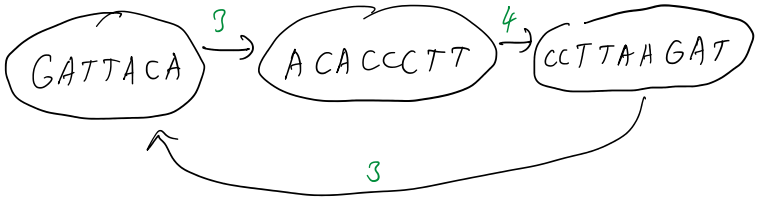
We can define overlap by percentage of read, or by absolute size of overlap.

Simple example:



Assembly: GATTACACCCTTACCTT

More complicated: (cycles)



Assembly:

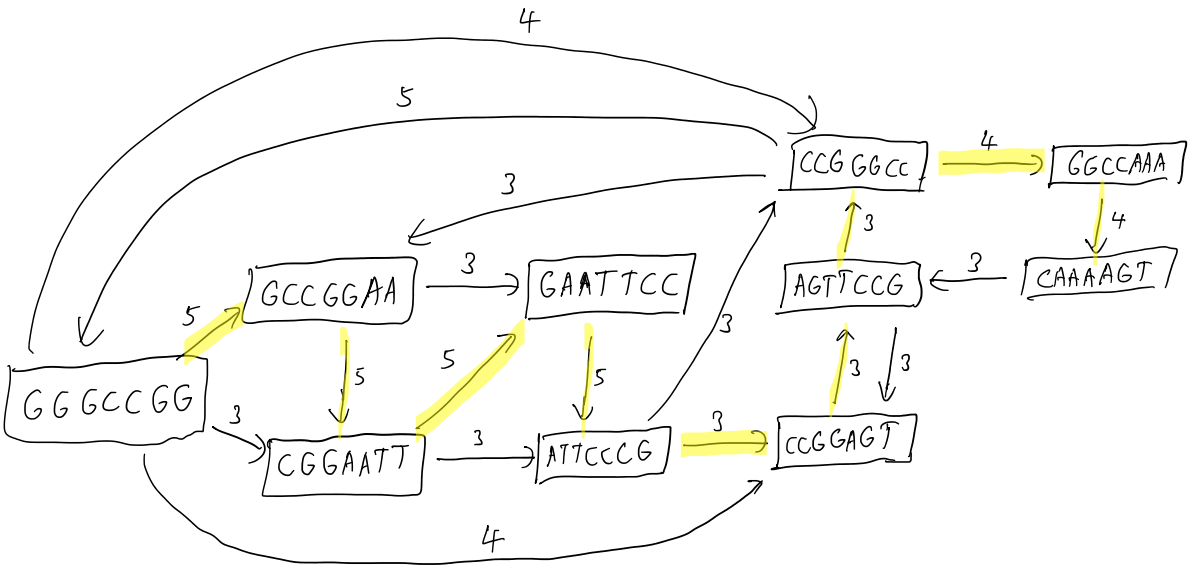
?

Cycles are sometimes real, such as circular bacterial or mitochondrial DNA, but might also result from repetitiveness or random chance.

We may also want to label the edges with the overlap amount.

Layout

We can think of assemblies as paths through the overlap graph that use every read once.

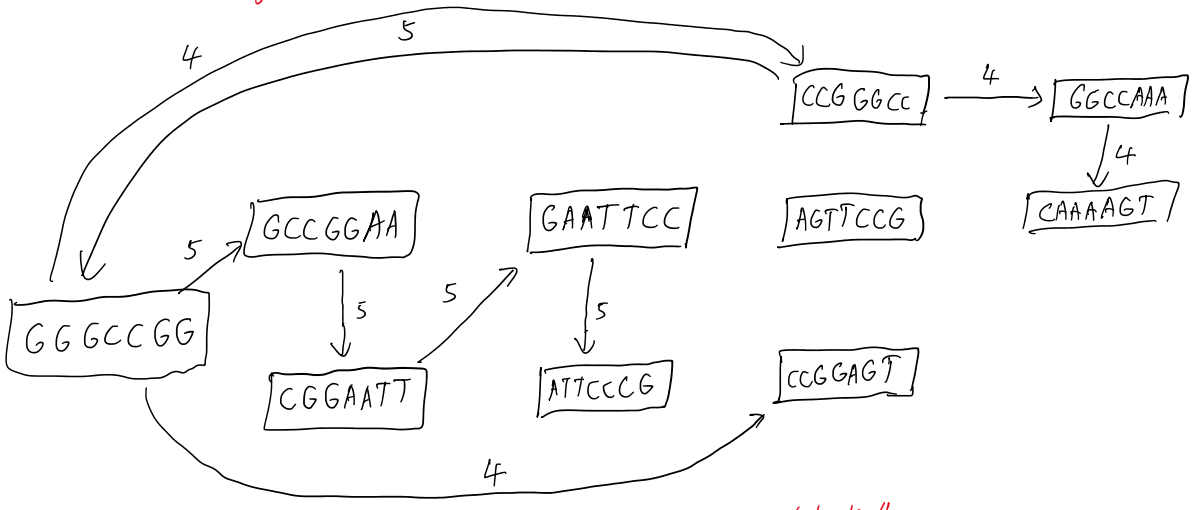


Original String: GGGCCGGAATTCCCGGAGTTCCGGGCCAGT

What creates all the unused edges? Repeats.

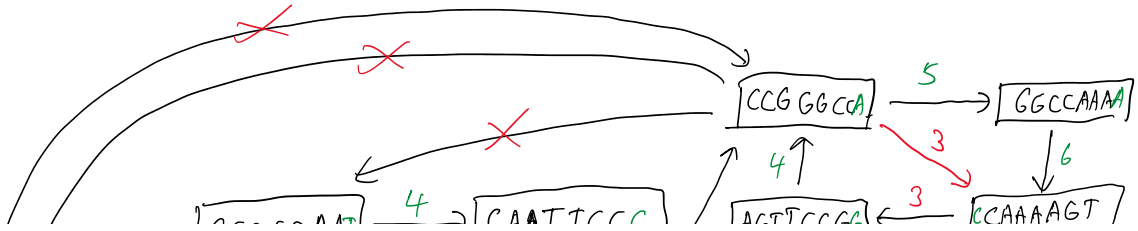
Two potential helpers: (1) increase read length
(2) increase overlap requirement

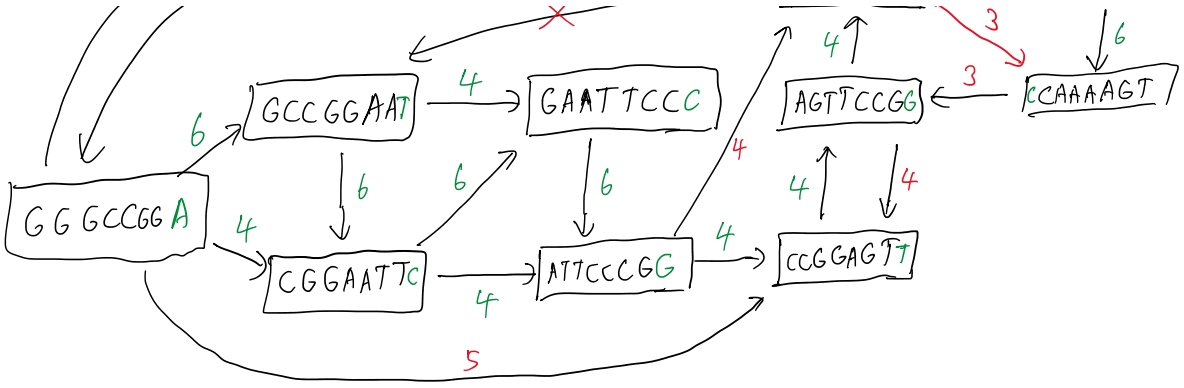
Filter out edges with < 4 overlap



Might leave disconnected graph, but simplifies graph substantially.

Increased read lengths from 7 to 8.





This can increase the lengths of correct overlaps.

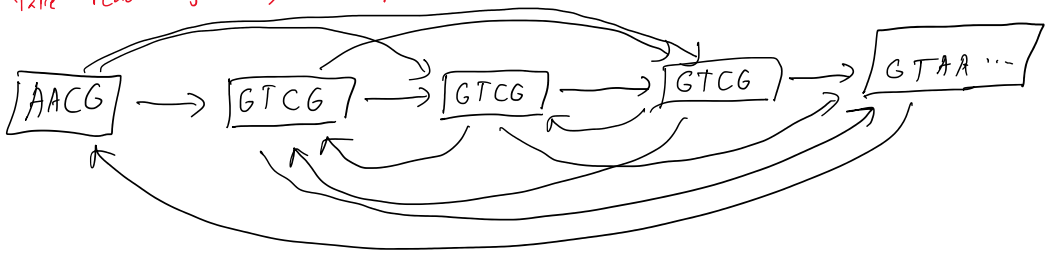
Careful: This time, there are fewere spurious edges, but that is only sometimes the case.

Exercise: Prove that given a random genome, if we sample n reads of length 100 vs. length 150, we will in expectation have "almost" the same number of ≥5-overlaps that are spurious.

More importantly, substantially increasing read length can resolve tandem repeats.

Ex. S = AACGTCGTCGTCGTAA ···

Let's take read length 4, overlap 1, reads (evenly spaced)



What about read length 8, overlap 2, reads (evenly spaced)

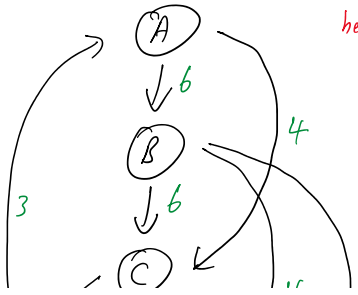
Longer reads allow capturing context outside of repeat region.

What about non-exact overlap?

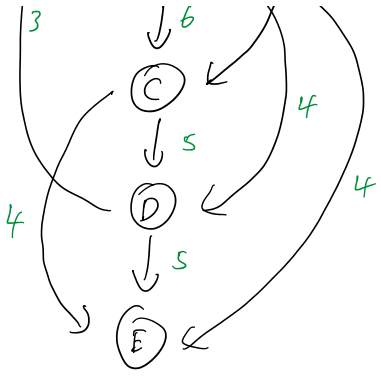
aligned bases

we use simple aligned bases count for overlap here, but can use more complex formula

(A) CCGTTCCACT
(B) TTCAACTTAG
(C) TCAATTTAAC
(D) ATTCAGCCG



(C) TCAATTTAAC
(D) ATTCAGCCG
(E) TTAGCGAA



Still find path as usual. But now can't collapse into single string because of mismatches.

Layout

```
CCGTTCCACT
    TTCAACTTAG
     TCAATTTAAC
        ATTCAGCCG
          TTAGC-GAA
```

Consensus: CCGTTCAA[C/T]TTAGC[C/-]GAA

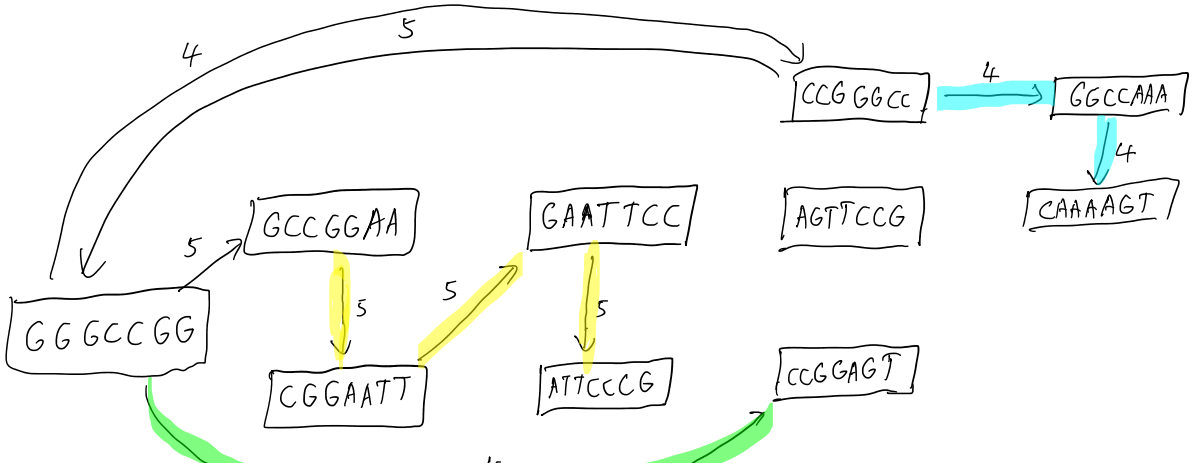
?

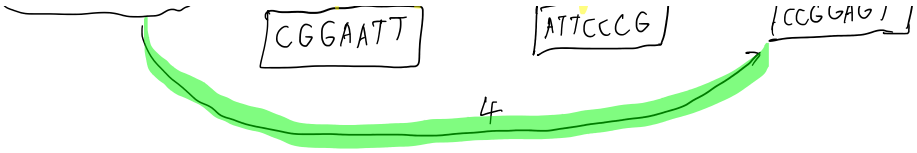
After finding the path, we take the consensus sequence of all the reads to form a "contig".

Define: A contig is a bunch of reads merged together into a longer consensus sequence.

If we are very lucky, the contig is the entire genome / chromosome.

But, sometimes the graph is broken or there's a cluster of repeats that we cannot resolve, so we need multiple paths to use up all the reads.





GGGCCGGAGT
GCCGGAATTCCCG
CCGGGCCAAAGT
AGTTCCG

} Contigs

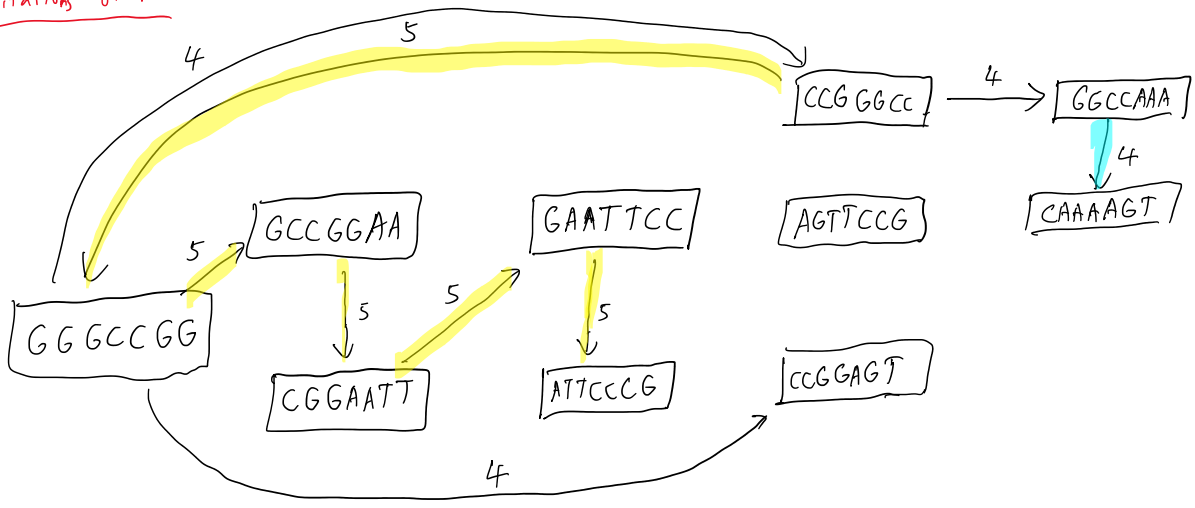
NSO = 12

analog in shortest common superstring problem is when you just concatenate strings without any overlap.

Define: The NSO of a genome assembly is the minimum length L s.t. contigs of size $\geq L$ make up 50% of the total assembly.

a perfect assembly (of a chromosome) has NSO = chromosome size since everything is in a contig

Limitations of NSO



CCGGGCCGGAATTCCCG
AGTTCCG
CCGGAGT
GGCCAAAGT

} Contigs

NSO = 9

So be careful as NSO is not perfect.

Exercise: What should be optimized when comparing assemblies?

Problem: How hard is it to find an optimal path that visits every node? NP-complete Hamiltonian path

↳ even if we had a perfect overlap graph with no errors, it might be hard to compute a single path reaching everything.

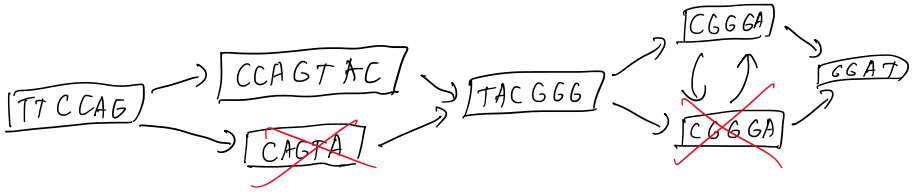
Have to use heuristics!

Assembly Strategy 2: String graphs (successor of overlap graphs) [Myers, 2005]

A string graph is a simplified overlap graph

Rules:

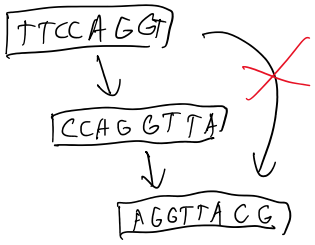
(1) Reads contained entirely within another read are removed.



This implies that you might be allowed to visit the same node twice, so no longer the Hamiltonian path problem.

Genome is still some tour of the string path ideally, but not Hamiltonian

(2) Transitive reduction: if $A \to B \to C$, then the $A \to C$ edge can be removed without reducing the strings that can be spelled out by tours

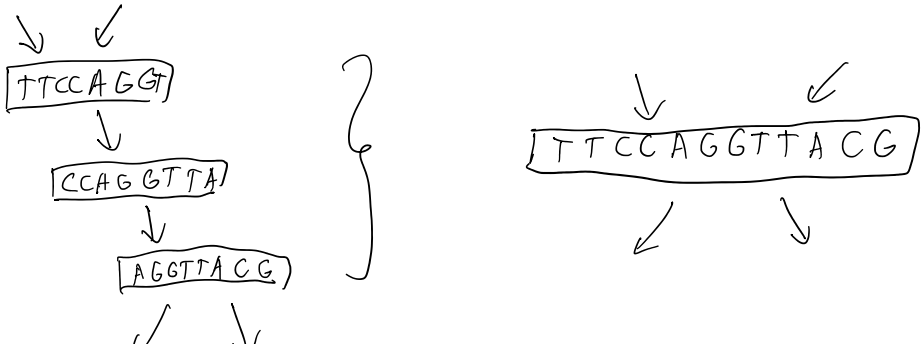


Reduces branching factor of edges, making paths easier to choose.

(3) Collapse paths with internal vertices

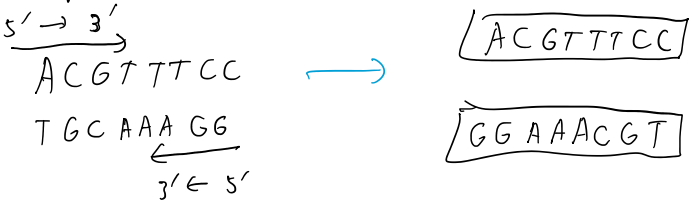
Define junction vertex as any vertex with in- or out-degree > 1

Define internal vertex as any vertex with in- and out-degree $= 1$



(4) Reminder: DNA is double stranded, and we don't actually know which strand we are reading.

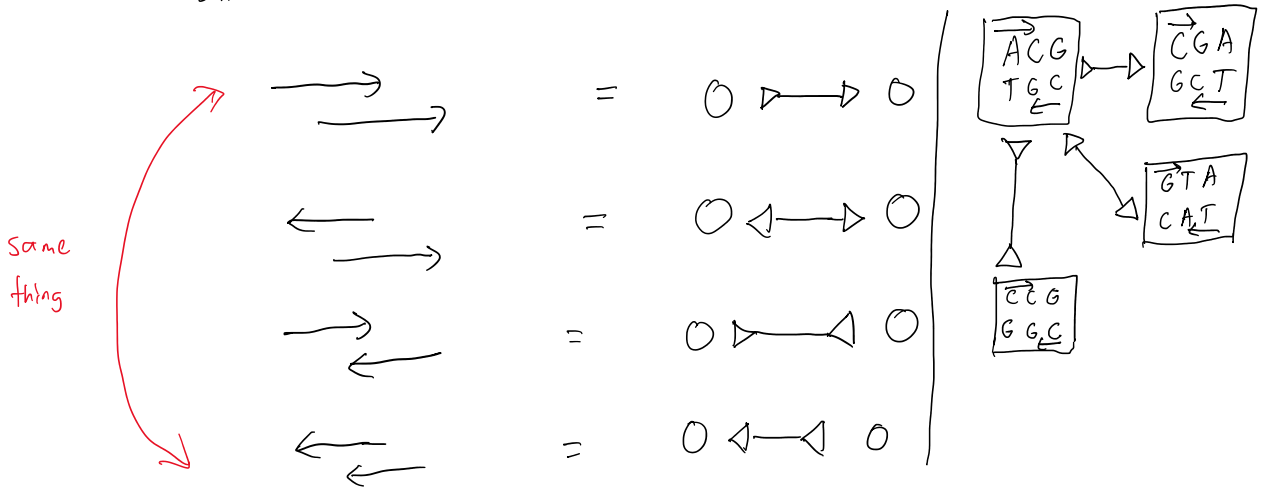
One solution, put every read as two separate nodes, going both ways



This doubles the number of nodes.

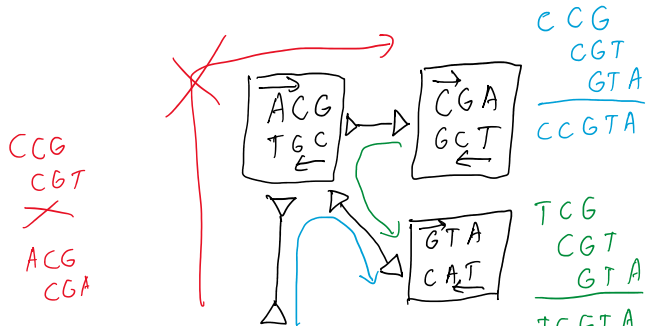
Alternate solution Bidirectional String Graphs (keep lexicographically smaller)

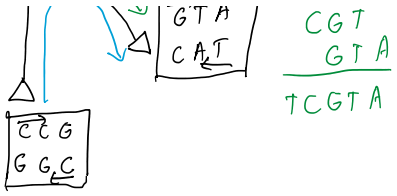
Collapse node w/ reverse complement, but add information about traversal direction to edges.



We can traverse every edge in either direction, but only if we continue using the same strand we entered on.

- If we enter a node via an in-arrow, we must leave via an out arrow
- If we enter a node via an out-arrow, we must leave via an in arrow





We want a short spanning walk in this bidirectional string graph, a tour that visits every node at least once.

But, the shortest spanning walk in a bidirectional string graph problem is also NP-hard. We have made things a lot easier, but not asymptotically easier.

Exercise: Prove that determining whether or not there exists a spanning walk in a bidirectional string graph of length at most k is NP-hard.

Assembly strategy 3: de Bruijn graph (Σ)

Define: The k-dimensional de Bruijn graph on m symbols is a directed graph (graph theory) with m^k vertices corresponding to each length-k string. A directed edge goes from every $a\alpha \to \alpha b$, where $a, b \in \Sigma$ and $\alpha \in \Sigma^{k-1}$.

} not the usual bioinformatics definition

We will use subgraphs of this de Bruijn graph, and in bioinformatics, we often refer to those subgraphs as "de Bruijn graphs"

Idea: break everything up into k-mers, and then use Eulerian cycles on edges

Read: CCAATGATTACAGG

CCAATGAT
CAATGATT
AATGATTA
ATGATTAC
TGATTACA
GATTACAG
ATTACAGG

} k-mers ($k=8$)

A de Bruijn graph has nodes representing k-mers, and edges connecting k-mers that are adjacent in a read.

(implying that they overlap in all but one position)

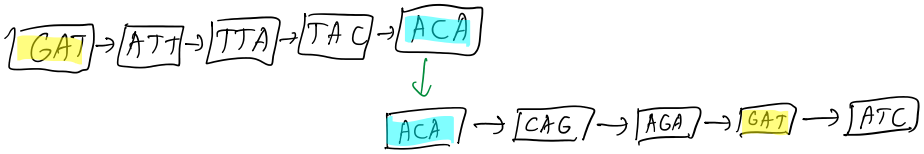
CCAATGAT
CAATGATT

k and $k-1$

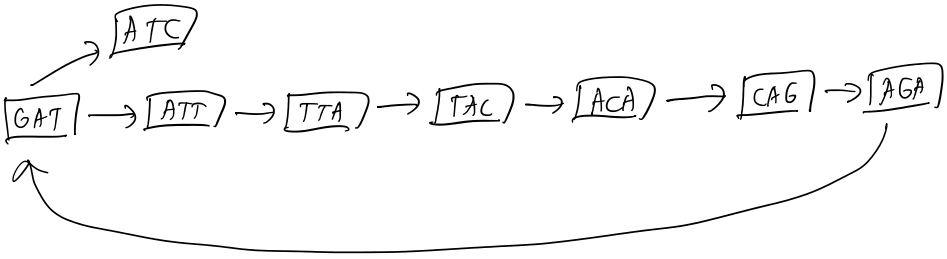
Can label edges with $(k-1)$-mer overlap

We can create a de Bruijn graph from an overlap graph by breaking up each read into many nodes for each constituent k-mer, and then merging nodes corresponding to the same k-mer.

Overlap graph



de Bruijn graph



Recall: An Eulerian path is a walk that uses every edge exactly once.

Notice: If $dG(s)$ is the de Bruijn graph of string s, then s corresponds to some Eulerian path in $dG(s)$

(An Eulerian path uses every edge, so it uses every k-mer overlap within reads)

Eulerian can be solved efficiently!

Alg: Notice that in order for a directed graph to have an Eulerian path, at most one node s can have greater out-degree than in-degree (by at most 1) and at most one node t can have greater in-degree than out-degree (by at most 1). All other nodes must have the same in-degree as out-degree.

Why?

So we start our algorithm by connecting $t \to s$, so that all nodes have the same in-degree as out-degree. (not always necessary)

Now, we will find an Eulerian cycle (path which returns to starting node)

Strategy, just keep walking in circles.

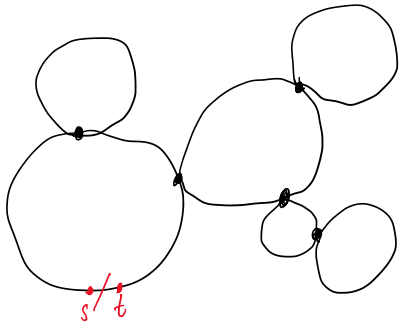
Repeat until all edges are used:

[illegible] [illegible] tour with unused edges ← how can we

Repeat until all edges are used:

- Start on any node w of current tour with unused edges ← how can we find these quickly?
- Walk until you are stuck. Notice that if you are stuck, you must be back at w, because otherwise there's always an exit edge since in- and out-degree matched.

Picture:



Then we can just unroll into a single big cycle.

And we can cut the edge we originally added b/t $s \leftarrow t$ to get a walk.

Exercise: What's the runtime? Can you implement this algorithm in pseudocode?

Problem: It's easy to find an Eulerian walk, but there are often exponentially many. Selecting between them is often hard. (depending on optimization, NP-hard)

Also, with errors in reads, often NO Eulerian walk

Exercise: Can you modify de Bruijn graphs to simplify them, similar to how we converted overlap graphs to string graphs?