CS681: Advanced Topics in Computational Biology

Week 7 Lectures 2-3

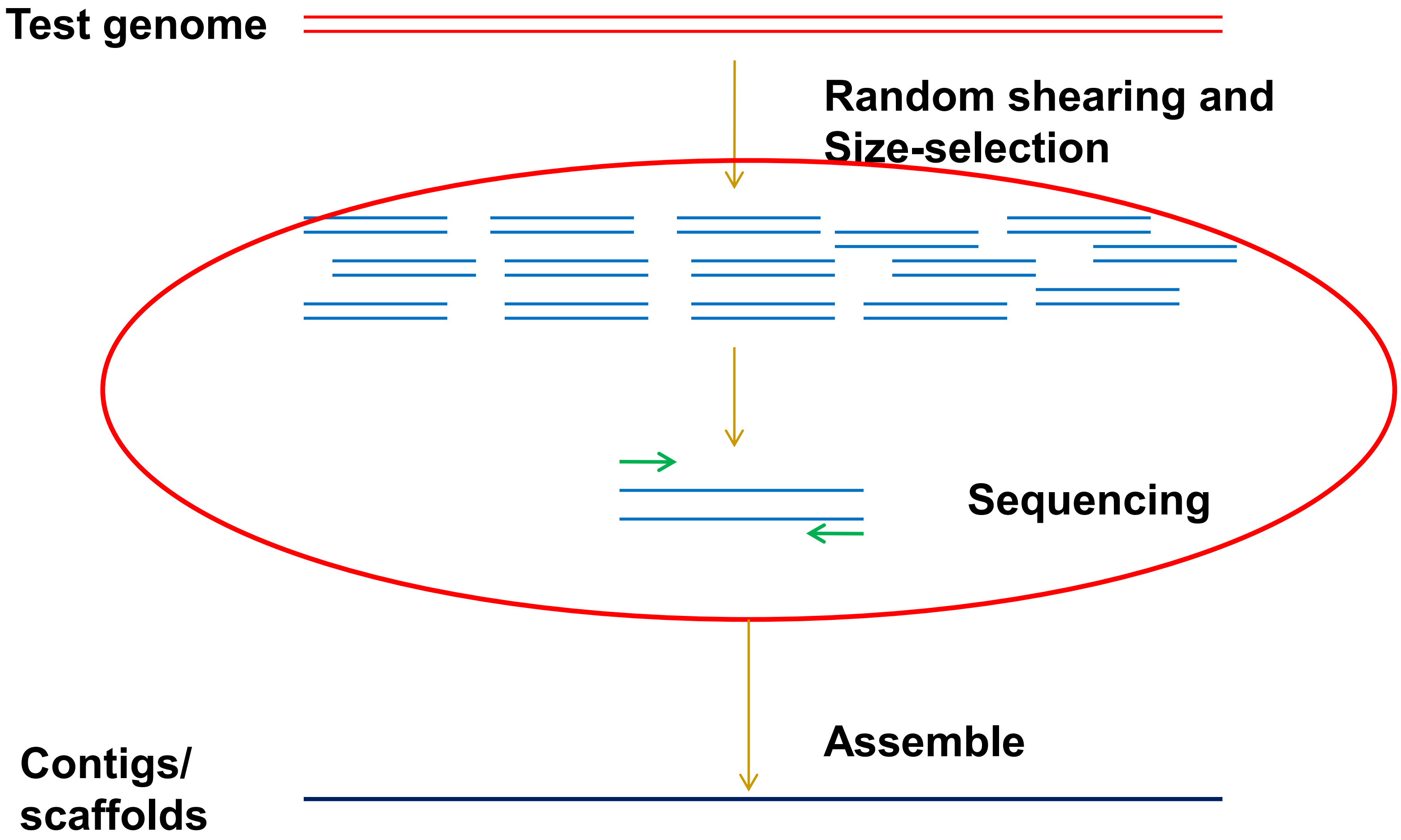
Can Alkan

EA224

calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs681/

Genome Assembly



Test genome

Random shearing and Size-selection

Sequencing

Assemble

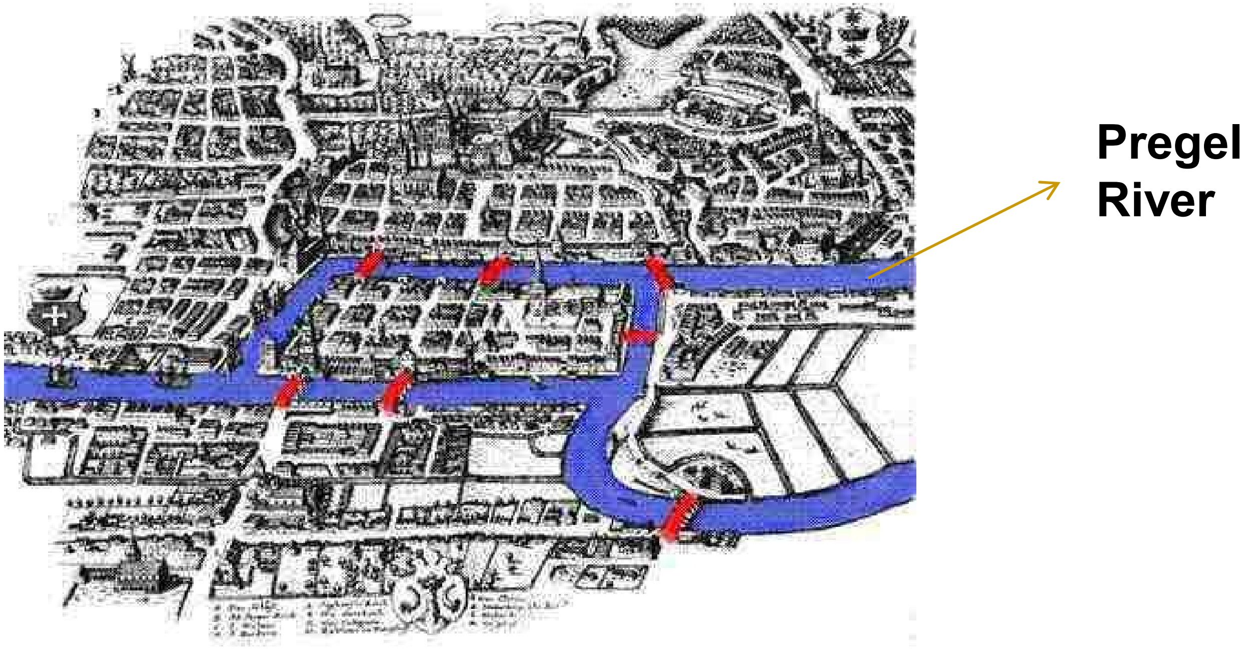
Contigs/ scaffolds

Graph problems in assembly

- Hamiltonian cycle/path
  - Typically used in overlap graphs
  - NP-hard
- Eulerian cycle/path
  - Typically used in de Bruijn graphs

The Bridge Obsession Problem

Find a tour crossing every bridge just once
Leonhard Euler, 1735

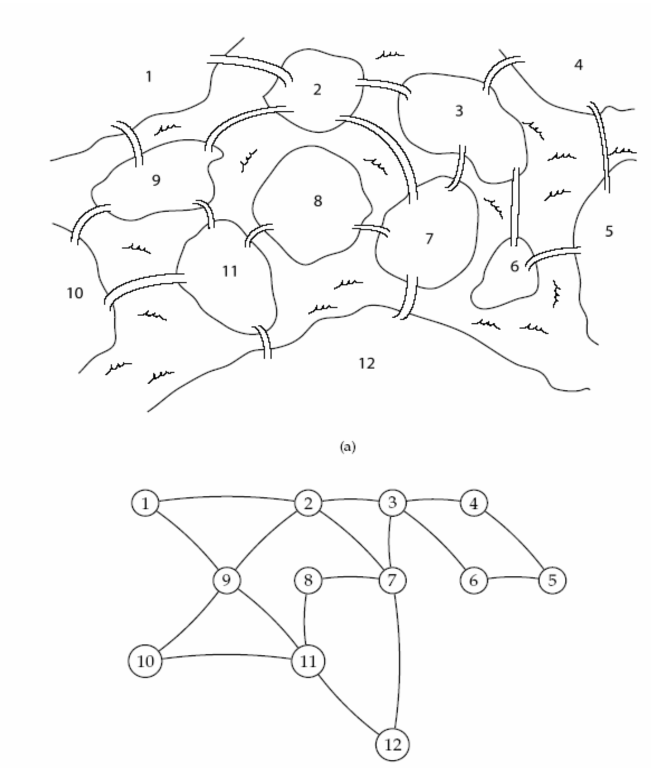


Bridges of Königsberg (Kaliningrad)

Eulerian Cycle Problem

- Find a cycle that visits every ***edge*** exactly once

- Linear time

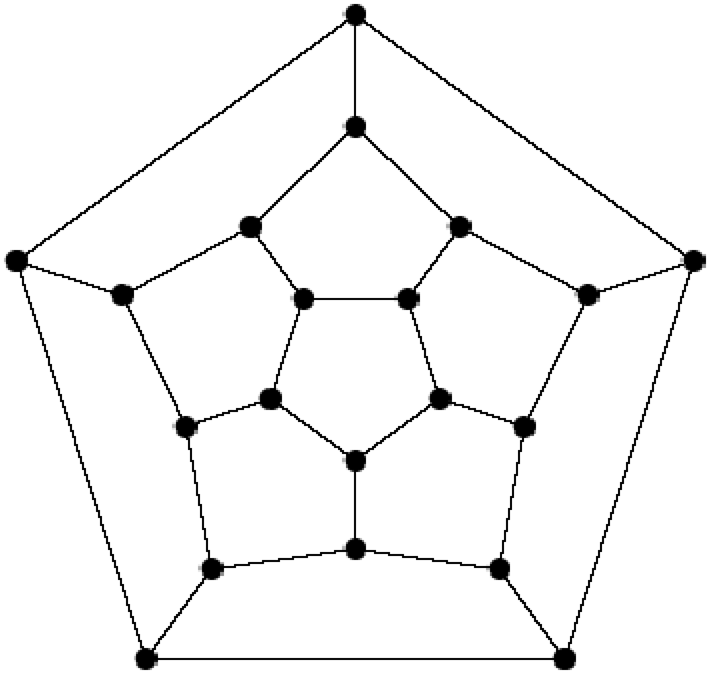


(a)

More complicated Königsberg

Hamiltonian Cycle Problem

- Find a cycle that visits every ***vertex*** exactly once
- NP – complete



Game invented by Sir William Hamilton in 1857

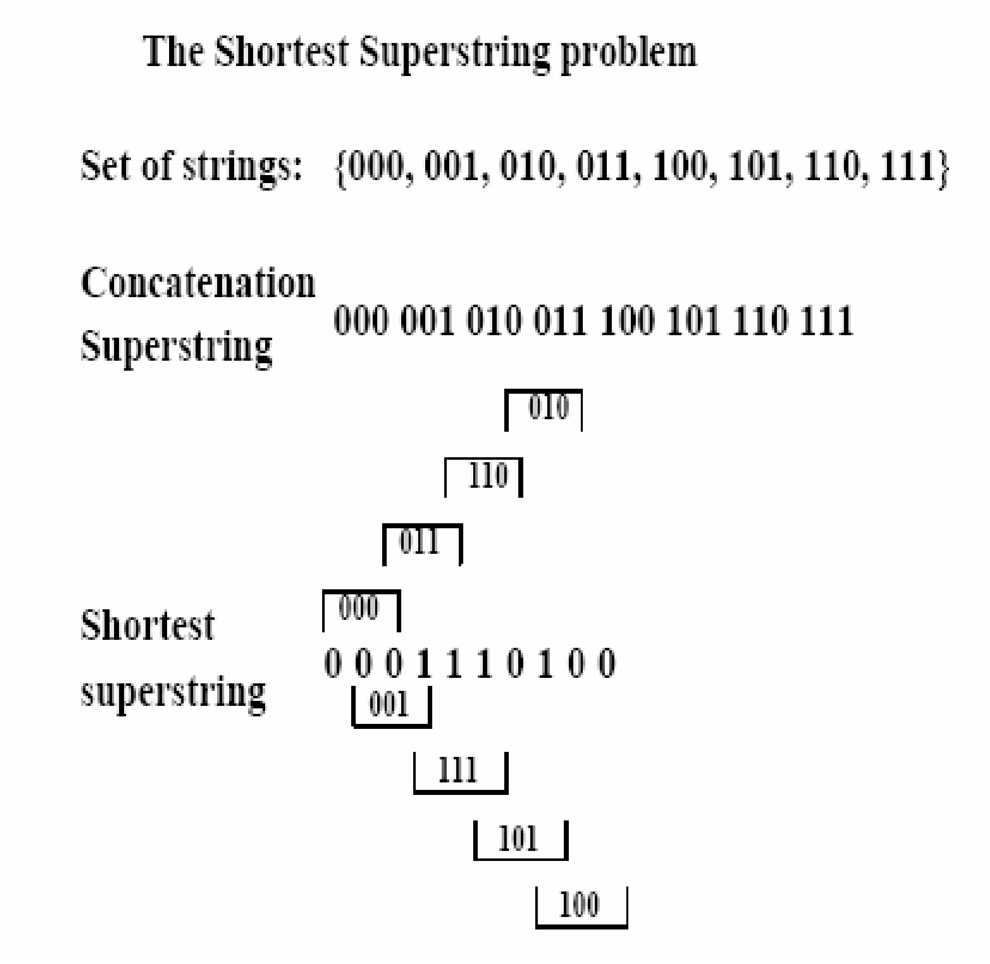
Traveling salesman problem

- TSP: find the shortest path that visits every vertex once
  - Directed / undirected
  - NP-complete
  - Exact solutions:
    - Held-Karp: $O(n^2 2^n)$
  - Heuristic
    - Lin-Kernighan

Assembly problem

- Genome assembly problem is finding **shortest common superstring** of a set of sequences (reads):
    - Given strings {s_1, s_2, …, s_n}; find the superstring T such that every s_i is a substring of T
    - NP-hard problem
    - Greedy approximation algorithm
        - Works for simple (low-repeat) genomes

Shortest Superstring Problem: Example



The Shortest Superstring problem

Set of strings: {000, 001, 010, 011, 100, 101, 110, 111}

Concatenation Superstring: 000 001 010 011 100 101 110 111

Shortest superstring: 0 0 0 1 1 1 0 1 0 0

Reducing SSP to TSP

- Define *overlap* (s_i, s_j) as the length of the longest prefix of s_j that matches a suffix of s_i.

aaaggcatcaaatctaaaggcatcaaa

aaaggcatcaaatctaaaggcatcaaa

overlap=12

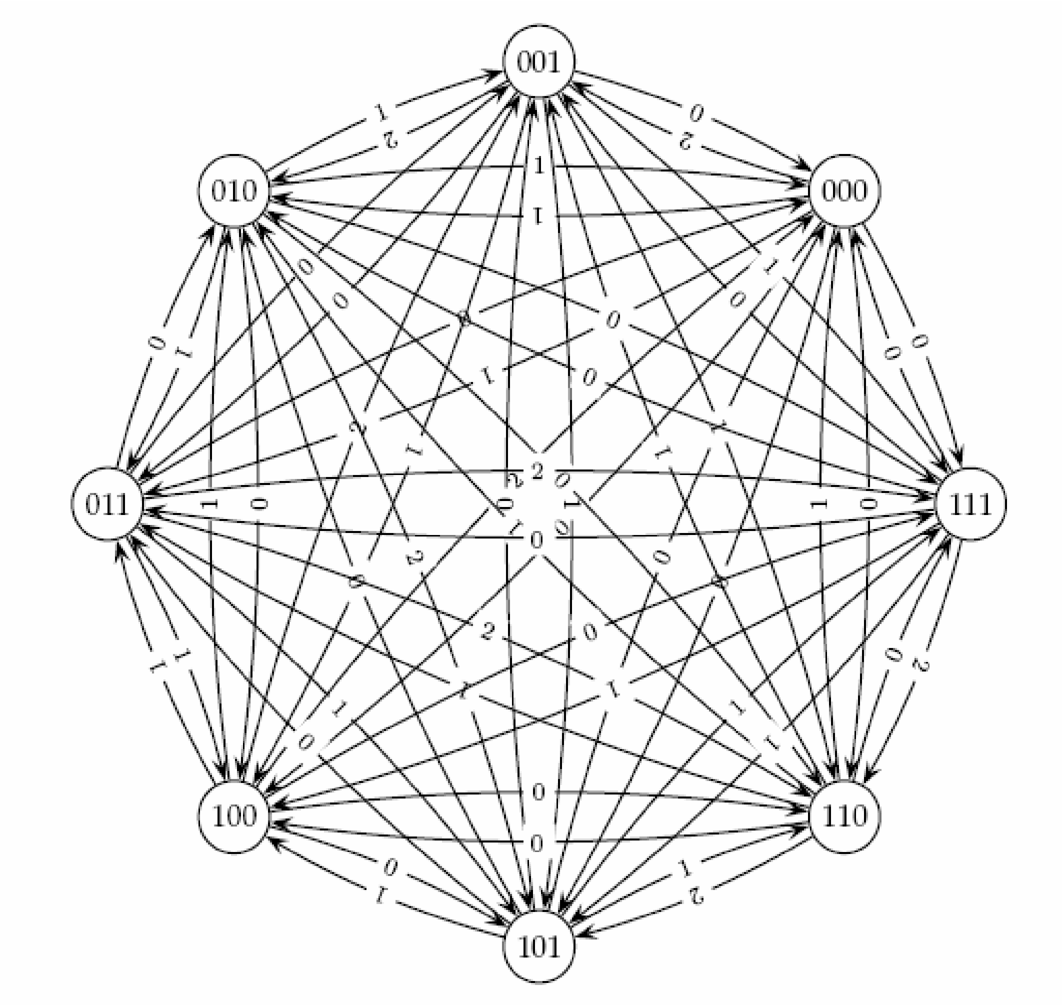
Reducing SSP to TSP

- Define *overlap* (s_i, s_j) as the length of the longest prefix of s_j that matches a suffix of s_i.

```
aaaggcatcaaatctaaaggcatcaaa
               aaaggcatcaaatctaaaggcatcaaa
```

- Construct a graph with *n* vertices representing the *n* strings $s_1, s_2, \ldots., s_n$.
- Insert edges of length *overlap* (s_i, s_j) between vertices s_i and s_j.
- Find the shortest path which visits every vertex exactly once. This is the **Traveling Salesman Problem** (TSP), which is also NP – complete.

Reducing SSP to TSP (cont'd)



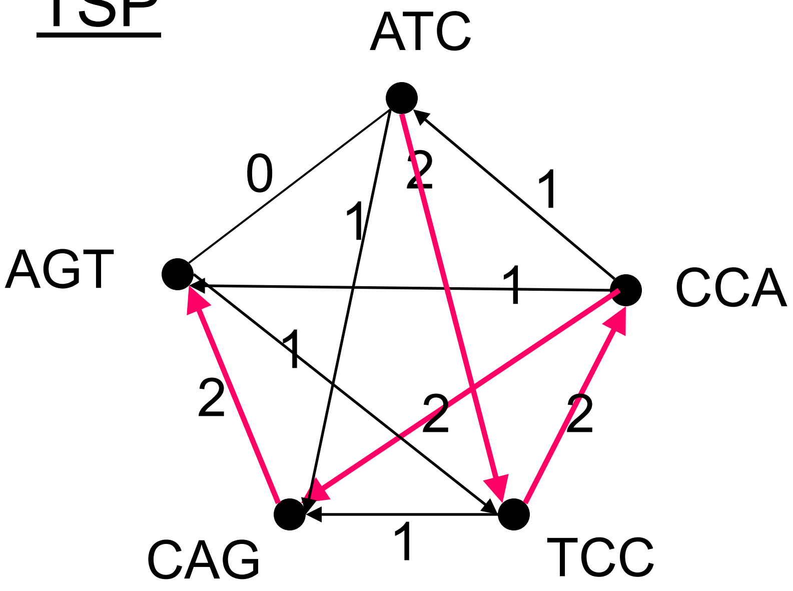
SSP to TSP: An Example

S = { ATC, CCA, CAG, TCC, AGT }

SSP

```
    AGT
  CCA
ATC
ATCCAGT
 TCC
   CAG
```

TSP



ATCCAGT

Assembly paradigms

- Overlap-layout-consensus
  - greedy (TIGR Assembler, phrap, CAP3...)
  - graph-based (Celera Assembler, Arachne)
    - SGA for NGS platforms
- Eulerian path on de Bruijn graphs(especially useful for short read sequencing)
  - EULER, Velvet, ABySS, ALLPATHS-LG, Cortex, etc.

Slide from Mihai Pop

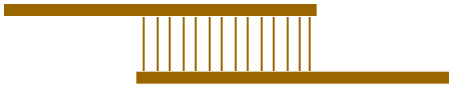
Overlap-Layout-Consensus

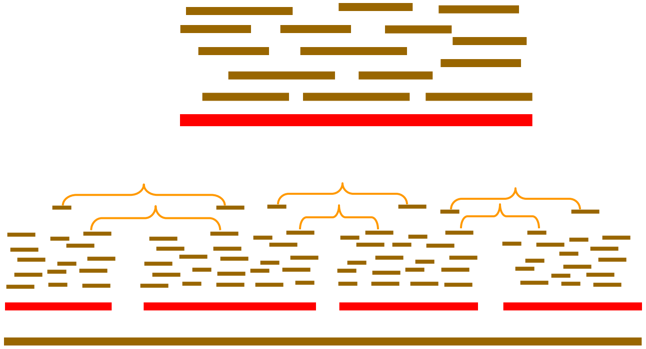
- Traditional assemblers: Phrap, Arachne, Celera etc.
- Short reads: Edena, SGA
- Generally more expensive computationally
  - Pairwise global alignments
- However, as reads get longer (>200bp ?) produce better results
  - They use the alignments of entire reads not isolated *k*-mer overlaps

Overlap-Layout-Consensus

Assemblers: ARACHNE, PHRAP, CAP, TIGR, CELERA

Overlap: find potentially overlapping reads





Layout: merge reads into contigs and contigs into scaffolds

Consensus: derive the DNA sequence and correct read errors

..ACGATTACAATAGGTT..

A quick example

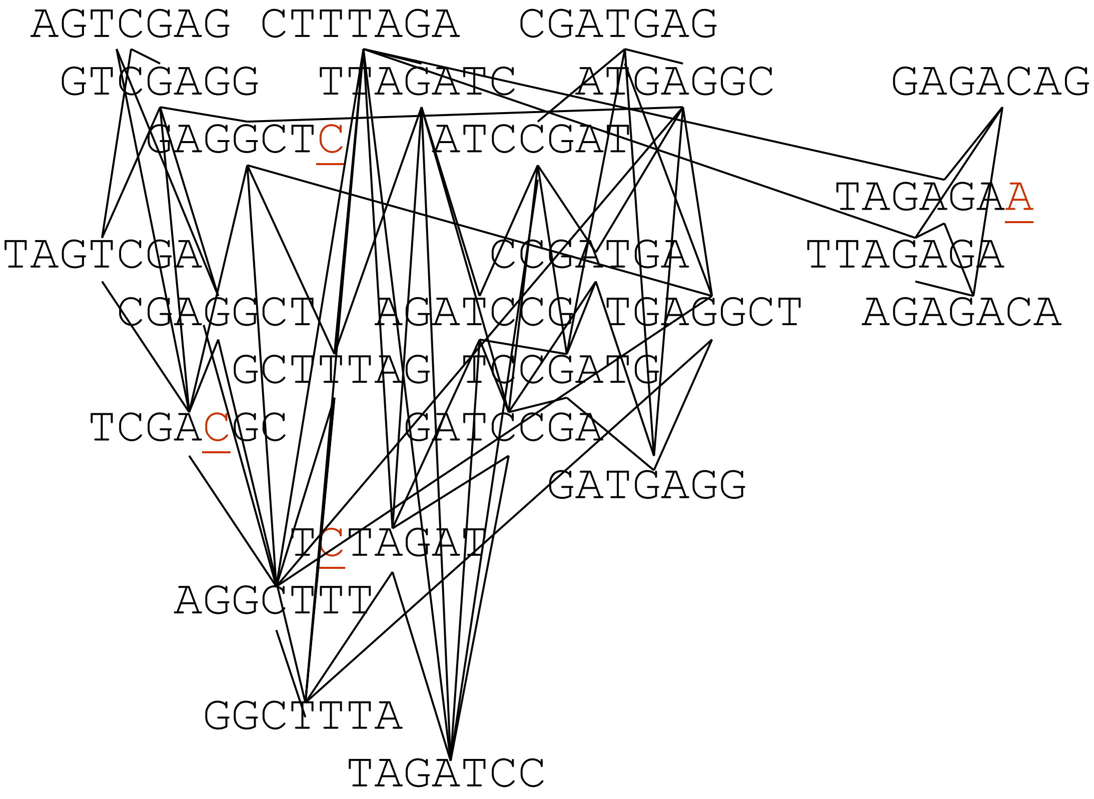
TAGTCGAGGCTTTAGATCCGATGAGGCTTTAGAGACAG

```
 AGTCGAG CTTTAGA  CGATGAG CTTTAGA
  GTCGAGG  TTAGATC  ATGAGGC    GAGACAG
     GAGGCTC   ATCCGAT AGGCTTT GAGACAG
 AGTCGAG    TAGATCC ATGAGGC  TAGAGAA
TAGTCGA  CTTTAGA CCGATGA    TTAGAGA
    CGAGGCT  AGATCCG TGAGGCT  AGAGACA
TAGTCGA GCTTTAG TCCGATG  GCTCTAG
   TCGACGC    GATCCGA GAGGCTT AGAGACA
TAGTCGA    TTAGATC GATGAGG TTTAGAG
  GTCGAGG TCTAGAT   ATGAGGC  TAGAGAC
      AGGCTTT  ATCCGAT AGGCTTT GAGACAG
 AGTCGAG   TTAGATT  ATGAGGC   AGAGACA
       GGCTTTA  TCCGATG    TTTAGAG
    CGAGGCT TAGATCC  TGAGGCT   GAGACAG
 AGTCGAG  TTTAGATC  ATGAGGC TTAGAGA
     GAGGCTT  GATCCGA GAGGCTT  GAGACAG
```

A quick example

```
 AGTCGAG CTTTAGA  CGATGAG CTTTAGA
  GTCGAGG  TTAGATC  ATGAGGC    GAGACAG
     GAGGCTC   ATCCGAT AGGCTTT GAGACAG
 AGTCGAG    TAGATCC ATGAGGC  TAGAGAA
TAGTCGA  CTTTAGA CCGATGA    TTAGAGA
    CGAGGCT  AGATCCG TGAGGCT  AGAGACA
TAGTCGA GCTTTAG TCCGATG  GCTCTAG
   TCGACGC    GATCCGA GAGGCTT AGAGACA
TAGTCGA    TTAGATC GATGAGG TTTAGAG
  GTCGAGG TCTAGAT   ATGAGGC  TAGAGAC
      AGGCTTT  ATCCGAT AGGCTTT GAGACAG
 AGTCGAG   TTAGATT  ATGAGGC   AGAGACA
       GGCTTTA  TCCGATG    TTTAGAG
    CGAGGCT TAGATCC  TGAGGCT   GAGACAG
 AGTCGAG  TTTAGATC  ATGAGGC TTAGAGA
     GAGGCTT  GATCCGA GAGGCTT  GAGACAG
```

A quick example



AGTCGAG CTTTAGA CGATGAG

GTCGAGG TTAGATC ATGAGGC GAGACAG

GAGGCTC ATCCGAT

TAGAGAA

TAGTCGA CCGATGA TTAGAGA

CGAGGCT AGATCCG TGAGGCT AGAGACA

GCTTTAG TCCGATG

TCGACGC GATCCGA

GATGAGG

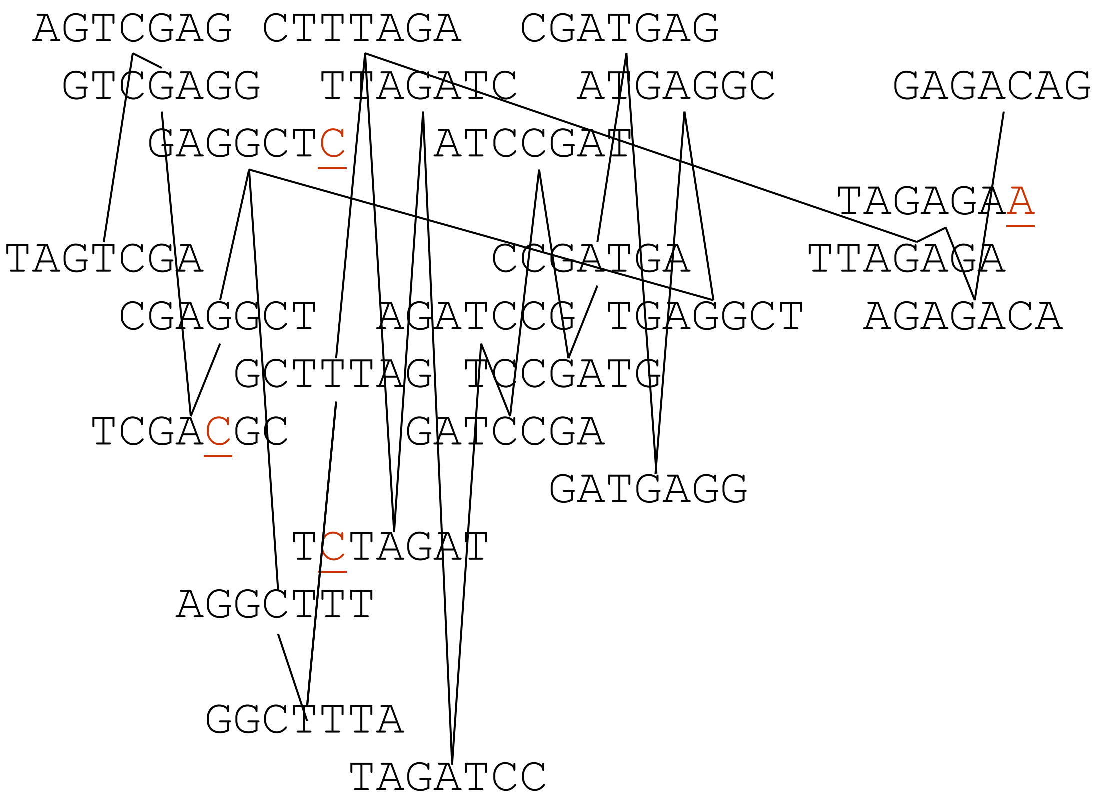
TCTAGAT

AGGCTTT

GGCTTTA

TAGATCC

A quick example



AGTCGAG CTTTAGA CGATGAG

GTCGAGG TTAGATC ATGAGGC GAGACAG

GAGGCTC ATCCGAT

TAGAGAA

TAGTCGA CCGATGA TTAGAGA

CGAGGCT AGATCCG TGAGGCT AGAGACA

GCTTTAG TCCGATG

TCGACGC GATCCGA

GATGAGG

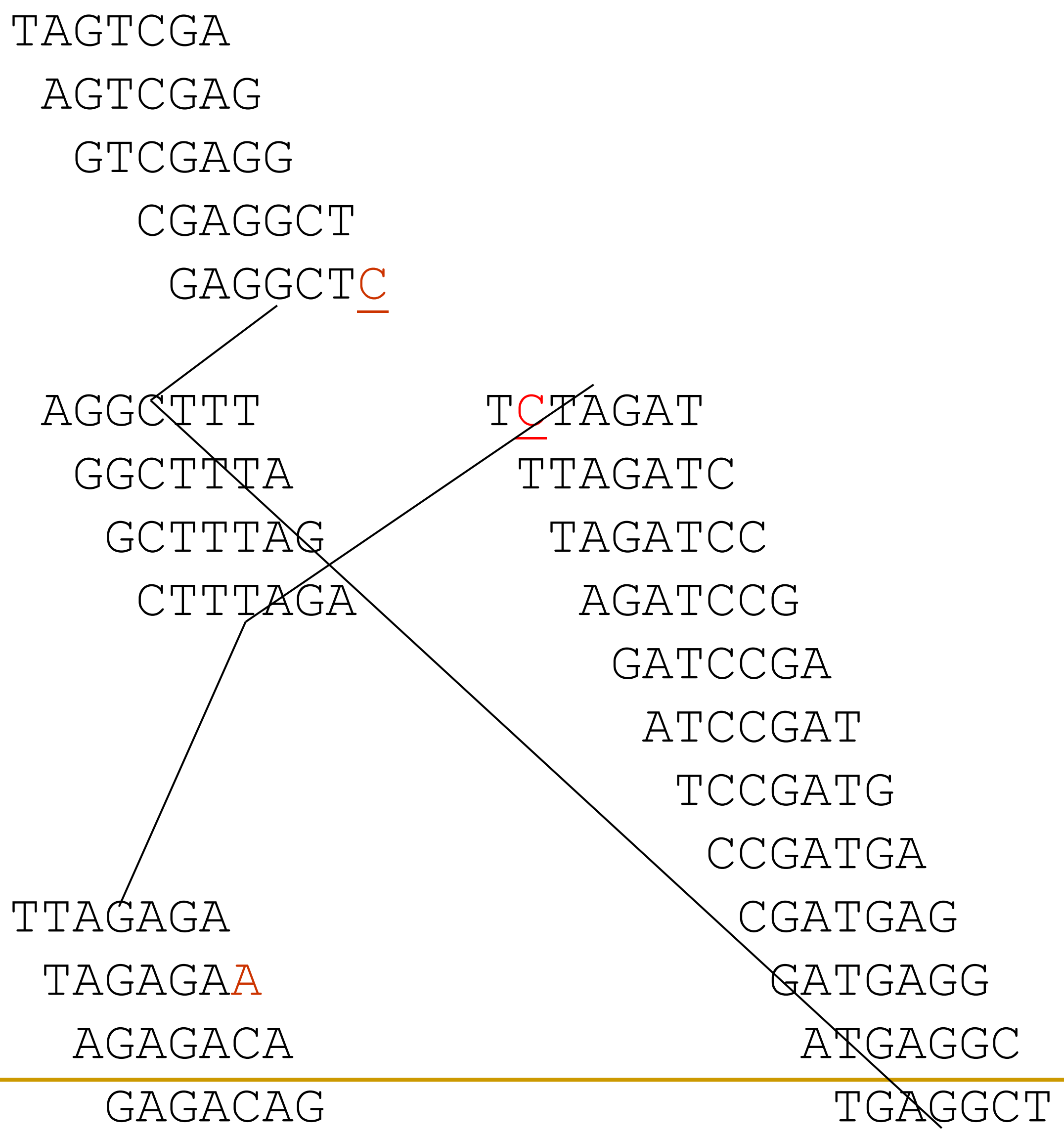
TCTAGAT

AGGCTTT

GGCTTTA

TAGATCC

A quick example



TAGTCGA
AGTCGAG
GTCGAGG
CGAGGCT
GAGGCTC

AGGCTTT
GGCTTTA
GCTTTAG
CTTTAGA

TCTAGAT
TTAGATC
TAGATCC
AGATCCG
GATCCGA
ATCCGAT
TCCGATG
CCGATGA
CGATGAG
GATGAGG
ATGAGGC
TGAGGCT

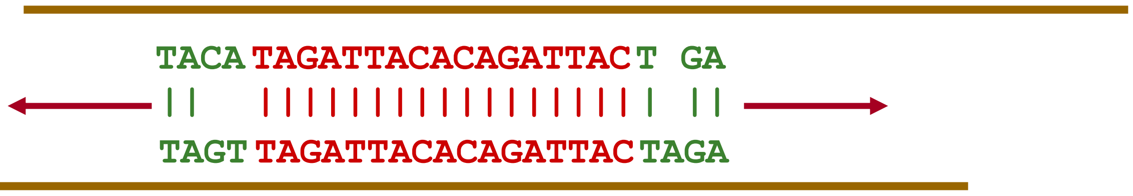
TTAGAGA
TAGAGAA
AGAGACA
GAGACAG

Overlap

- Find the best match between the suffix of one read and the prefix of another
- Due to sequencing errors, need to use dynamic programming to find the optimal *overlap alignment*
- Apply a filtration method to filter out pairs of fragments that do not share a significantly long common substring

Overlapping Reads

- Sort all k-mers in reads (k ~ 24)
- Find pairs of reads sharing a k-mer
- Extend to full alignment – throw away if not >95% similar



Overlapping Reads and Repeats

- A *k*-mer that appears N times, initiates N^2 comparisons

- For an *Alu* that appears 10^6 times → 10^{12} comparisons – too much

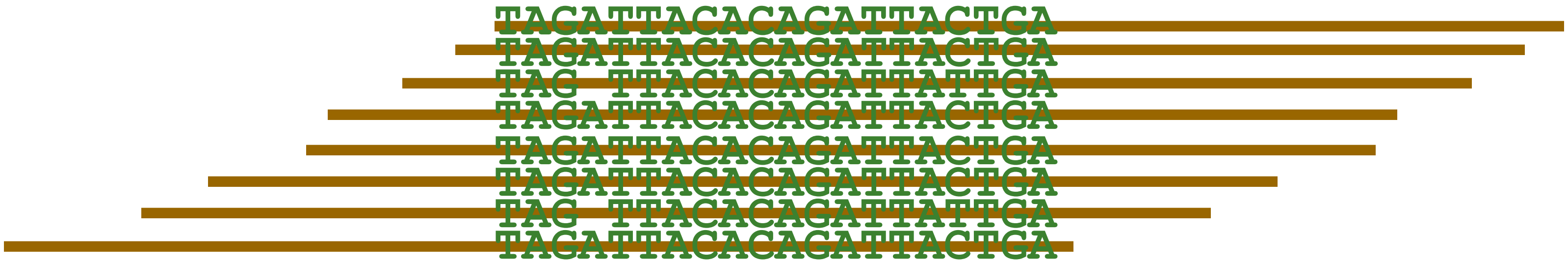
- **<u>Solution:</u>**

  Discard all *k*-mers that appear more than

  $t \times$ Coverage, ($t \sim 10$)

Finding Overlapping Reads

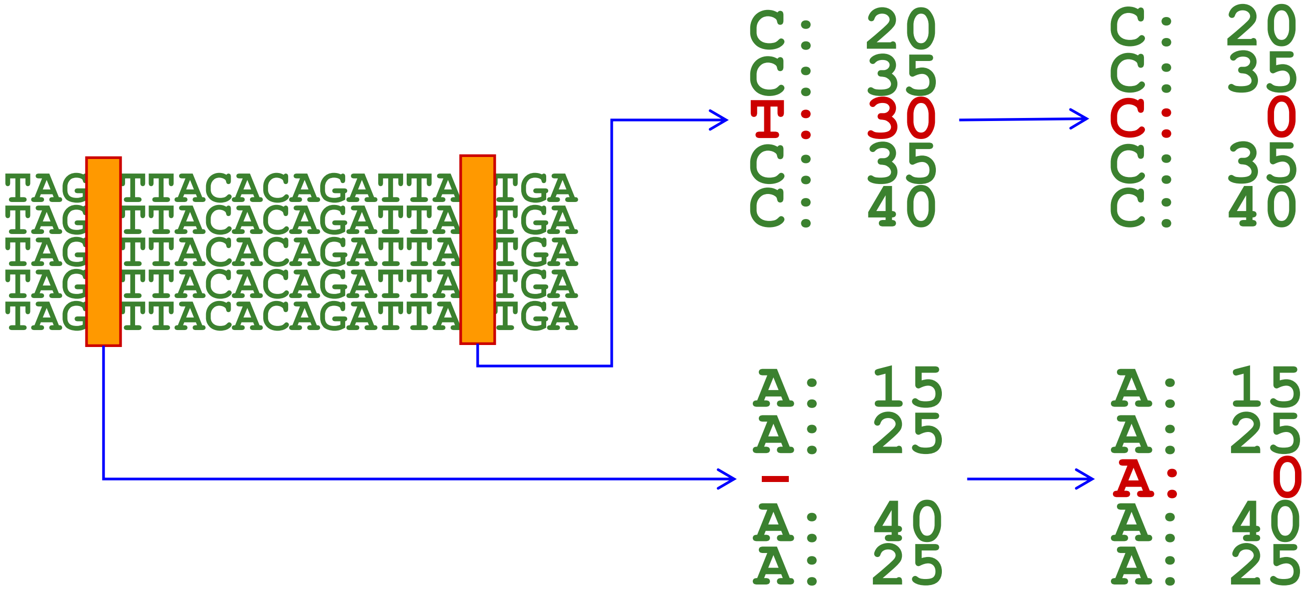
Create local multiple alignments from the overlapping reads



```
TAGATTACACAGATTACTGA
TAGATTACACAGATTACTGA
TAG TTACACAGATTATTGA
TAGATTACACAGATTACTGA
TAGATTACACAGATTACTGA
TAGATTACACAGATTACTGA
TAG TTACACAGATTATTGA
TAGATTACACAGATTACTGA
```

Finding Overlapping Reads (cont'd)

- Correct errors using multiple alignment



TAG TTACACAGATTA TGA

TAG TTACACAGATTA TGA

TAG TTACACAGATTA TGA

TAG TTACACAGATTA TGA

TAG TTACACAGATTA TGA

C: 20
C: 35
T: 30
C: 35
C: 40

C: 20
C: 35
C: 0
C: 35
C: 40

A: 15
A: 25
-
A: 40
A: 25

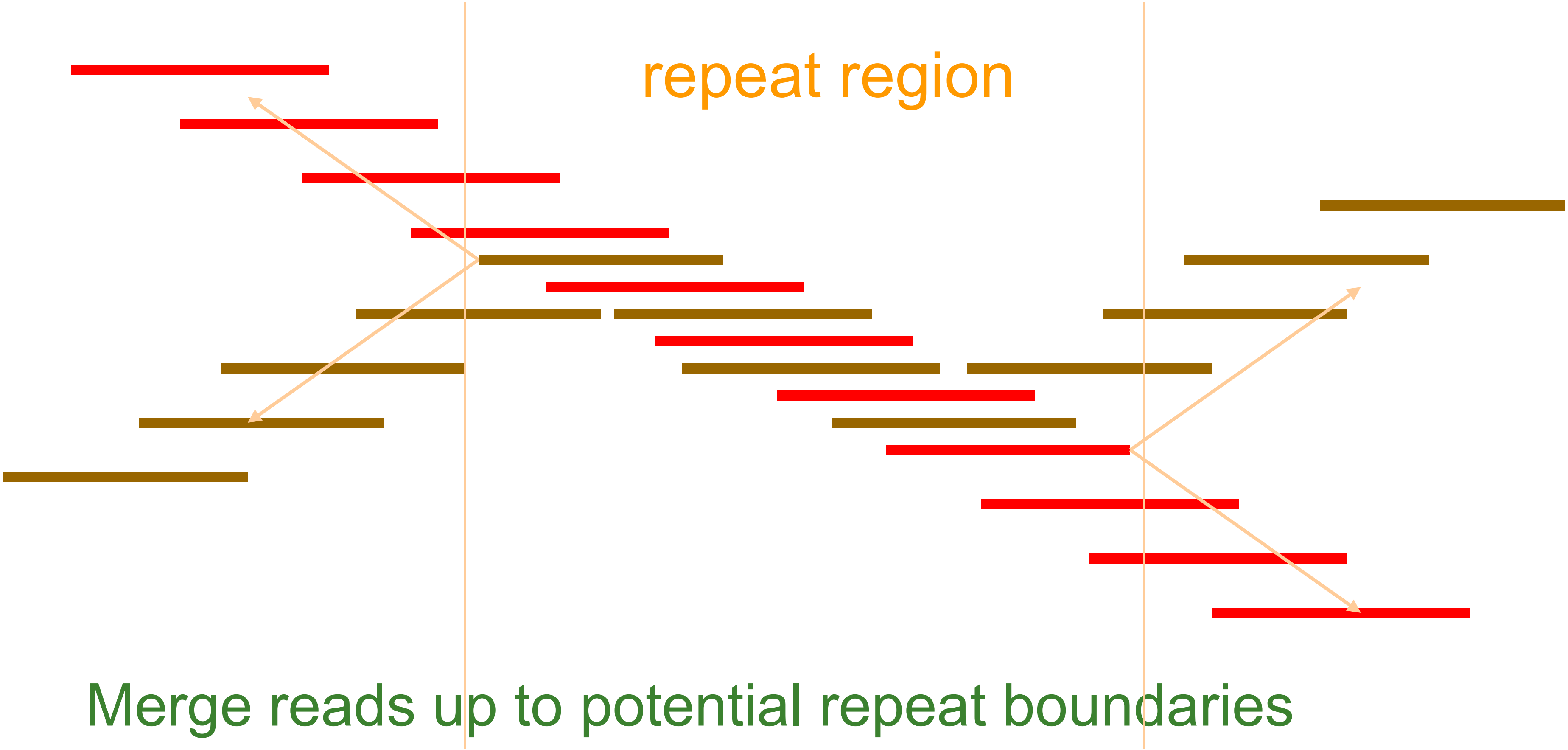
A: 15
A: 25
A: 0
A: 40
A: 25

- Score alignments
- Accept alignments with good scores

Layout

- Repeats are a major challenge
- Do two aligned fragments really overlap, or are they from two copies of a repeat?
- Solution: repeat masking – hide the repeats!!!
- Masking results in high rate of misassembly (up to 20%)
- Misassembly means alot more work at the finishing step

Merge Reads into Contigs



Merge reads up to potential repeat boundaries

Repeats, Errors, and Contig Lengths

- Repeats shorter than read length are OK
- Repeats with more base pair differencess than sequencing error rate are OK
- To make a smaller portion of the genome **appear** repetitive, try to:
  - Increase read length
  - Decrease sequencing error rate

Error Correction

Role of error correction:

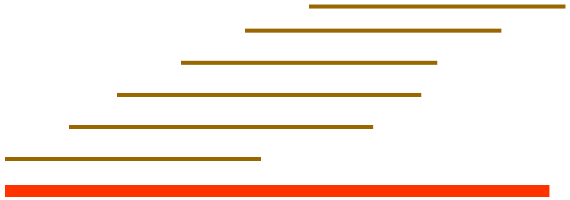
Discards ~90% of single-letter sequencing errors

decreases error rate

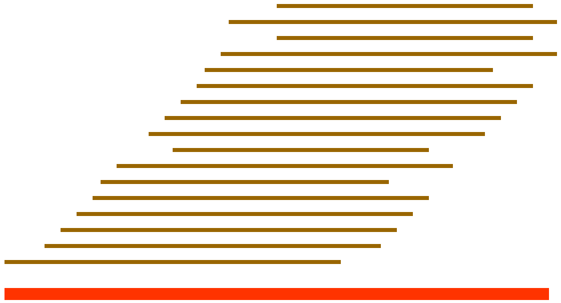
⇒ decreases effective repeat content

⇒ increases contig length

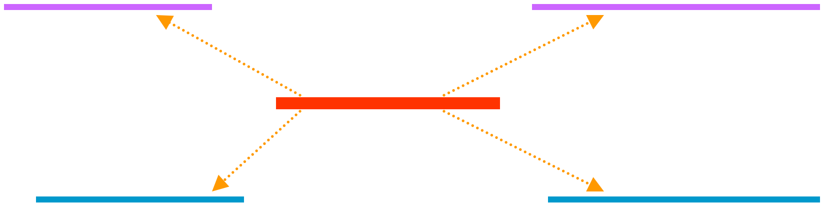
Link Contigs into Scaffolds



Normal density



Too dense:
Overcollapsed?

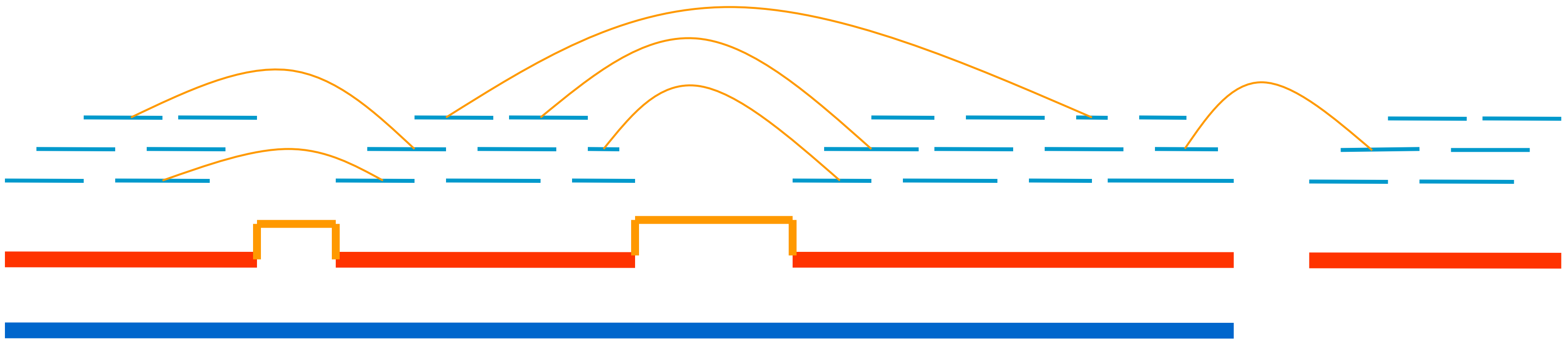


Inconsistent links:
Overcollapsed?

Link Contigs into Scaffolds(cont'd)

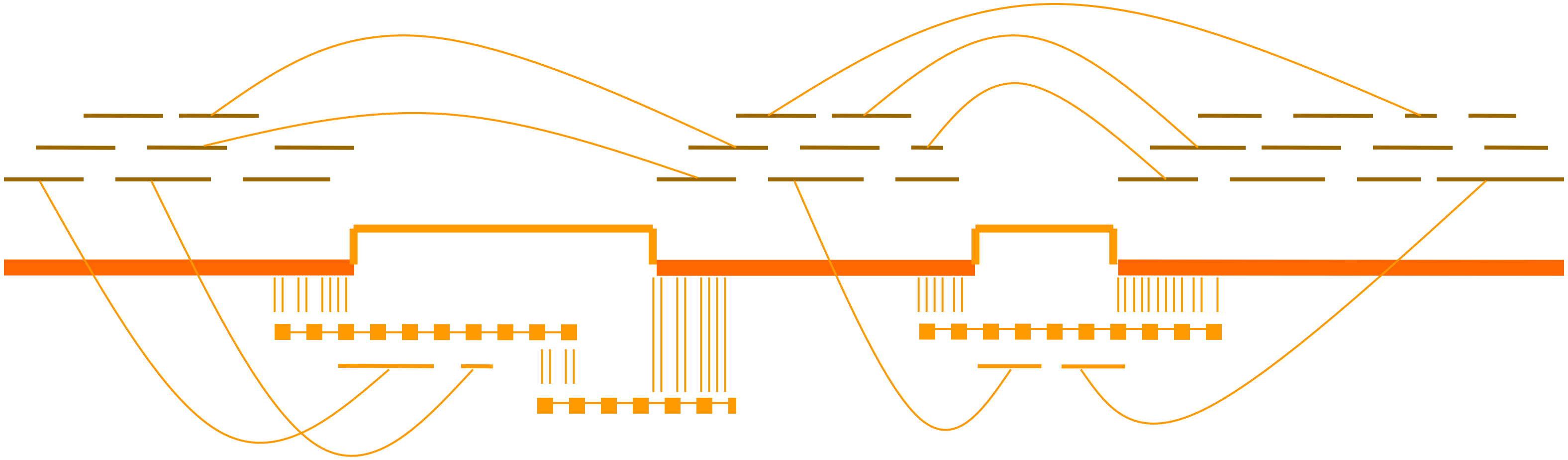
Find all links between unique contigs

Connect contigs incrementally, if ≥ 2 links

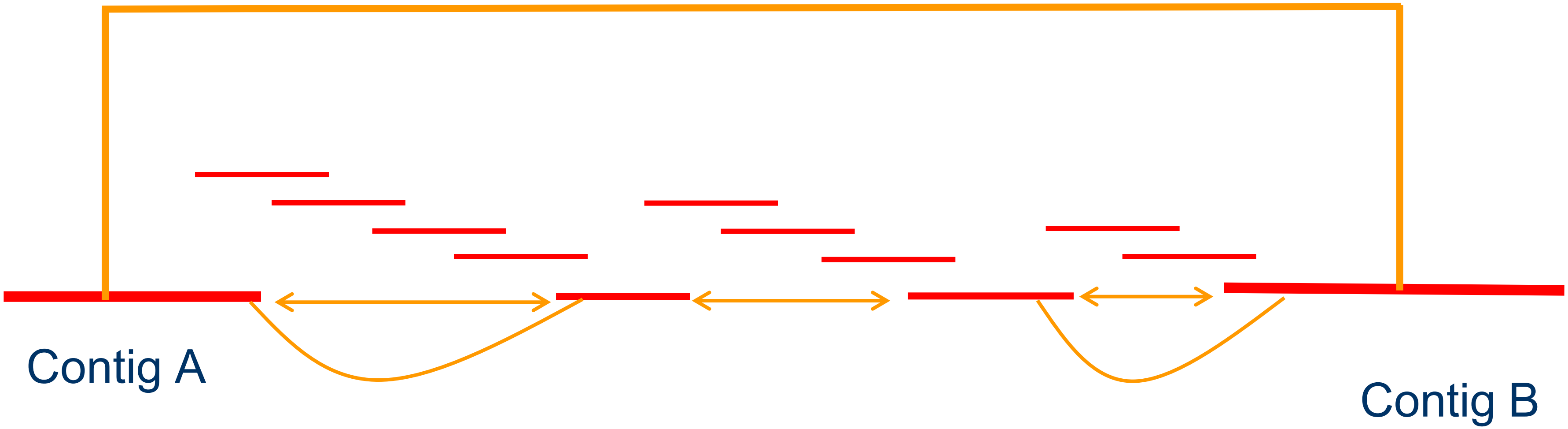


Link Contigs into Scaffolds (cont'd)

Fill gaps in scaffolds with paths of overcollapsed contigs



Link Contigs into Scaffolds (cont'd)



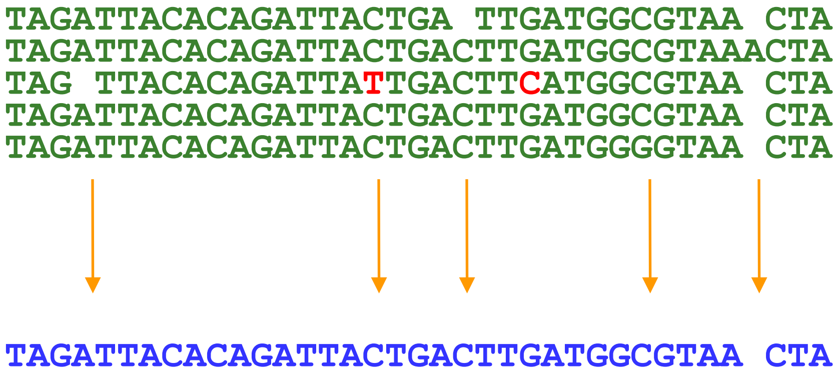
Define T: contigs linked to either A or B

Fill gap between A and B if there is a path in G passing only from contigs in T

Consensus

- A consensus sequence is derived from a profile of the assembled fragments
- A sufficient number of reads is required to ensure a statistically significant consensus
- Reading errors are corrected

Derive Consensus Sequence



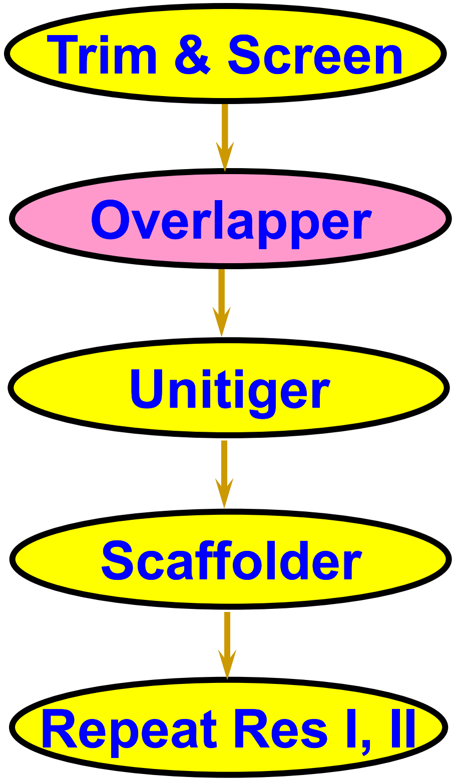
```
TAGATTACACAGATTACTGA TTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAAACTA
TAG TTACACAGATTATTGACTTCATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGGGTAA CTA

TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
```

Derive multiple alignment from pairwise read alignments

Derive each consensus base by weighted voting

Celera Assembler



Trim & Screen

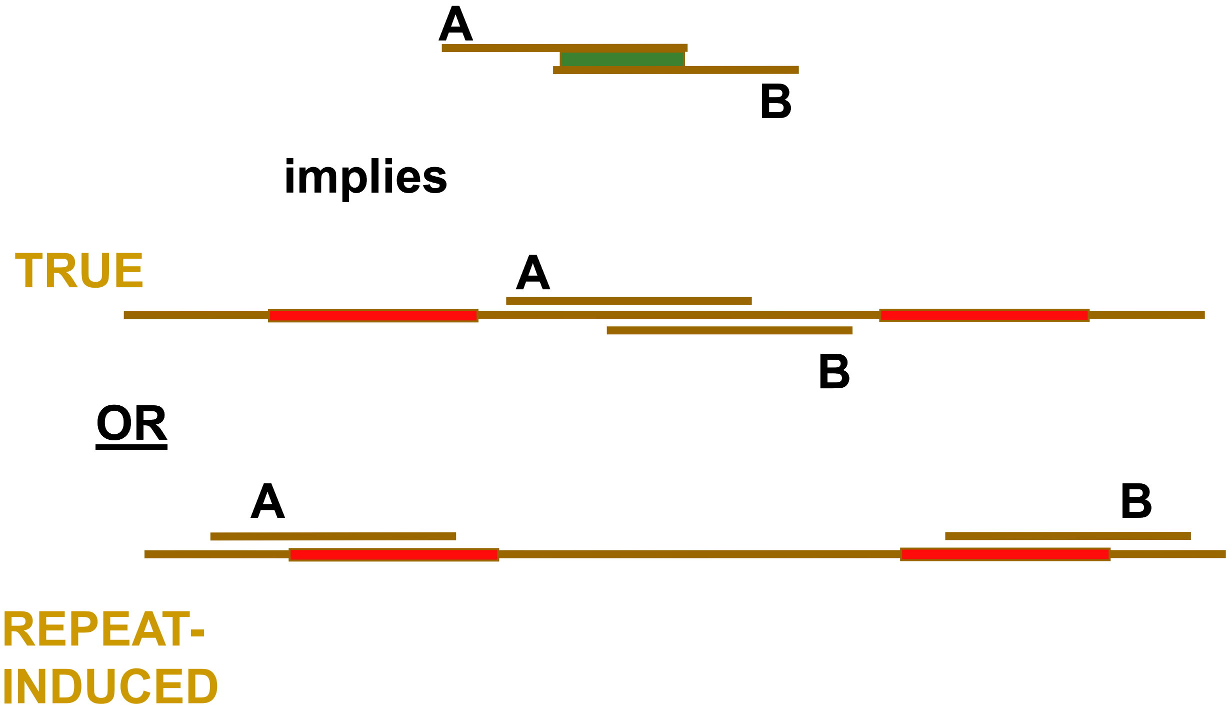
Overlapper

Unitiger

Scaffolder

Repeat Res I, II

Find all overlaps ≥ 40bp allowing 6% mismatch.



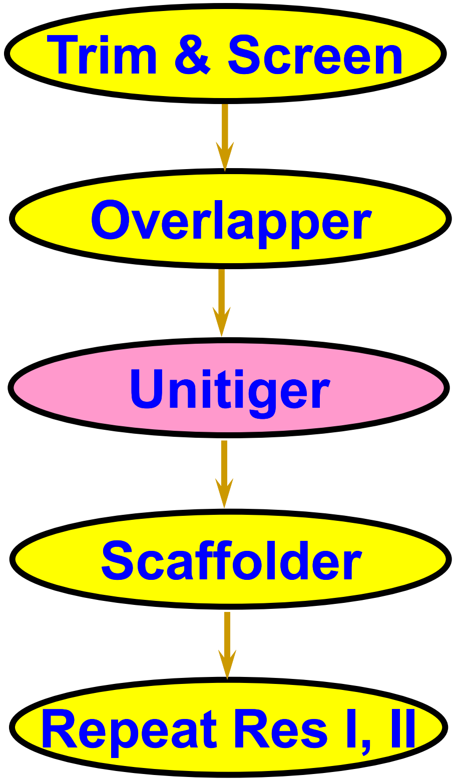
implies

TRUE

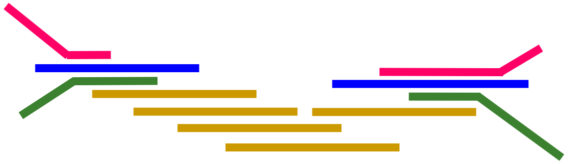
OR

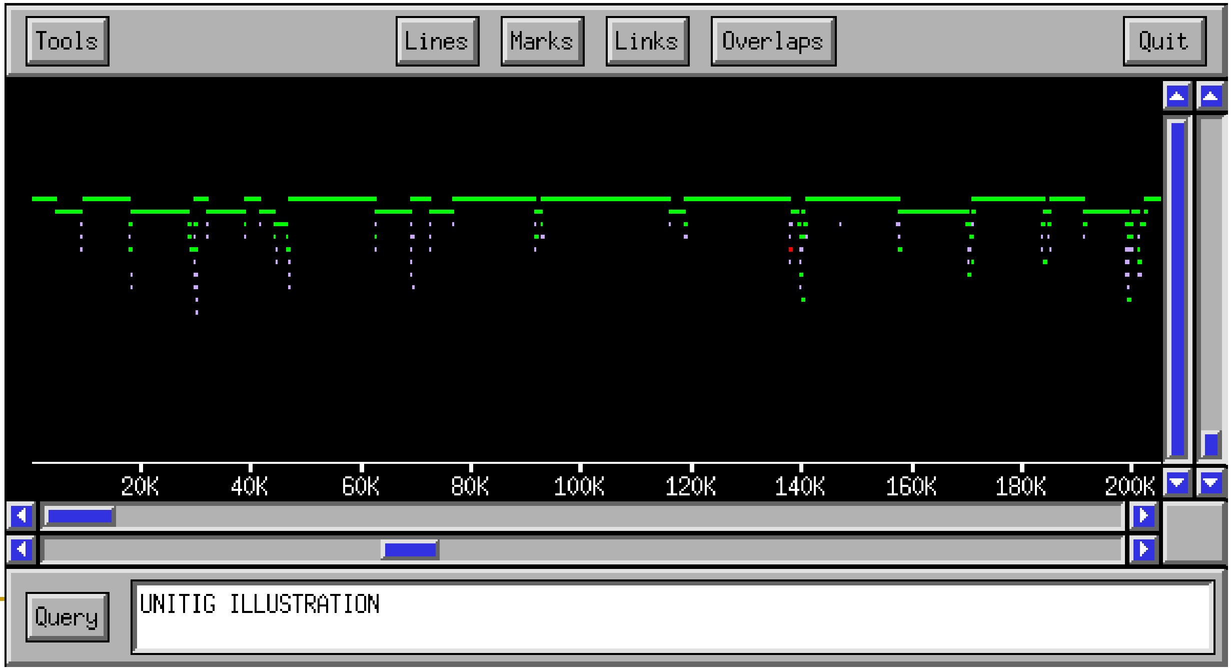
REPEAT-INDUCED

Celera Assembler



Compute all overlap consistent sub-assemblies:
Unitigs (Uniquely Assembled Contig)





UNITIG ILLUSTRATION

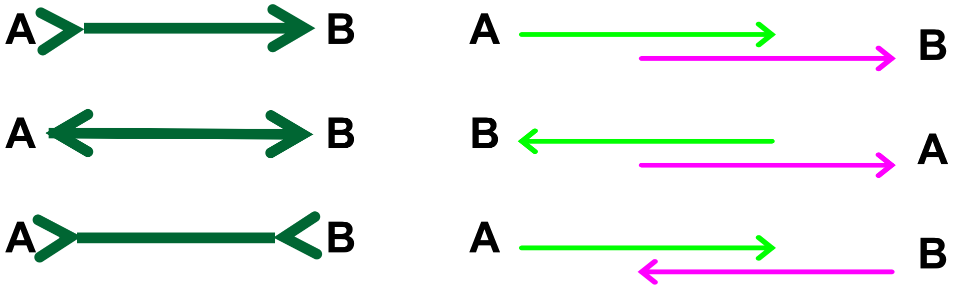
Celera Assembler

Edge Types:

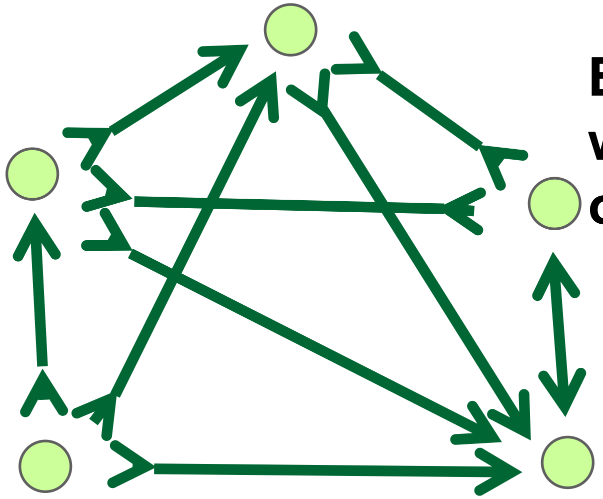
Regular Dovetail

Prefix Dovetail

Suffix Dovetail

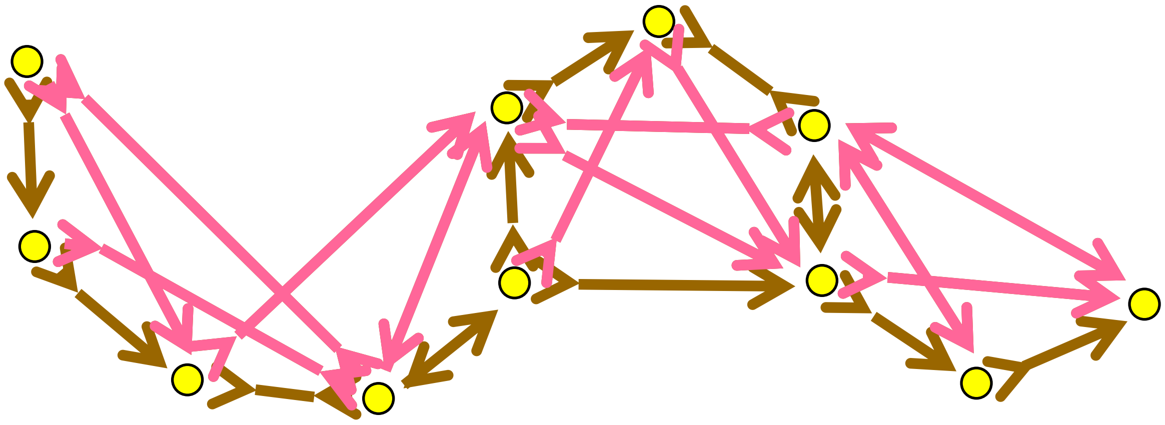


E.G.:

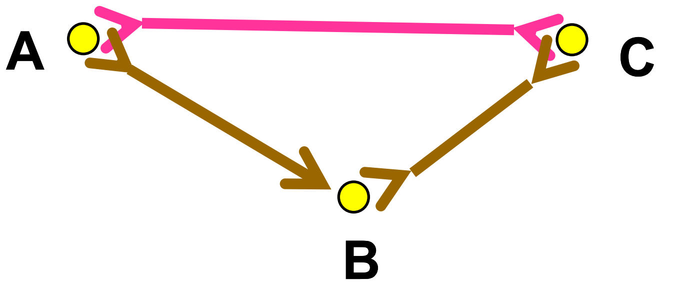


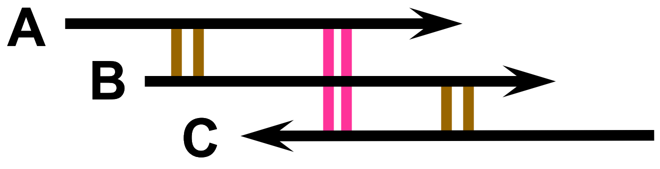
Edges are annotated with deltas of overlaps

The Unitig Reduction

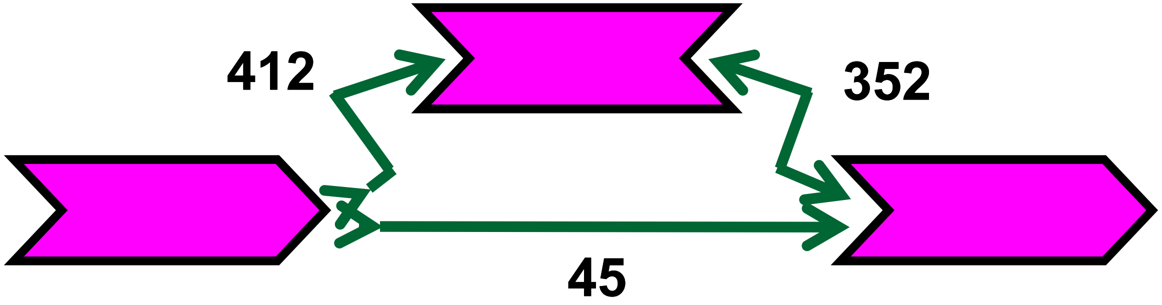


1. Remove “Transitively Inferrable” Overlaps:

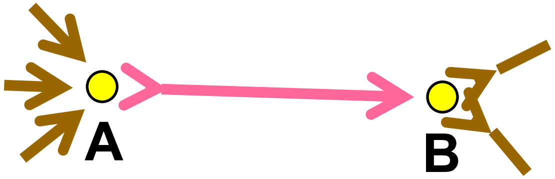


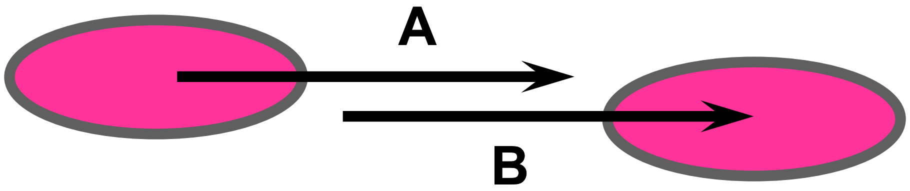


The Unitig Reduction



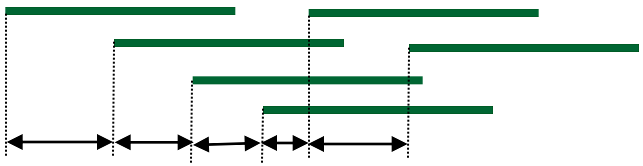
2. Collapse "Unique Connector" Overlaps:





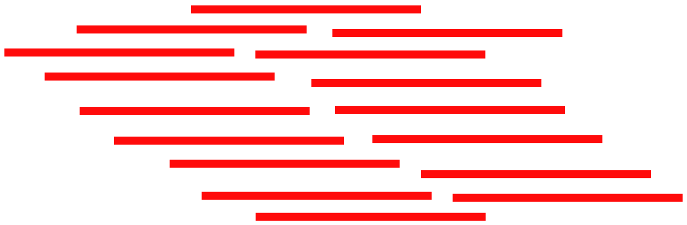
Identifying Unique DNA Stretches

Unique DNA unitig

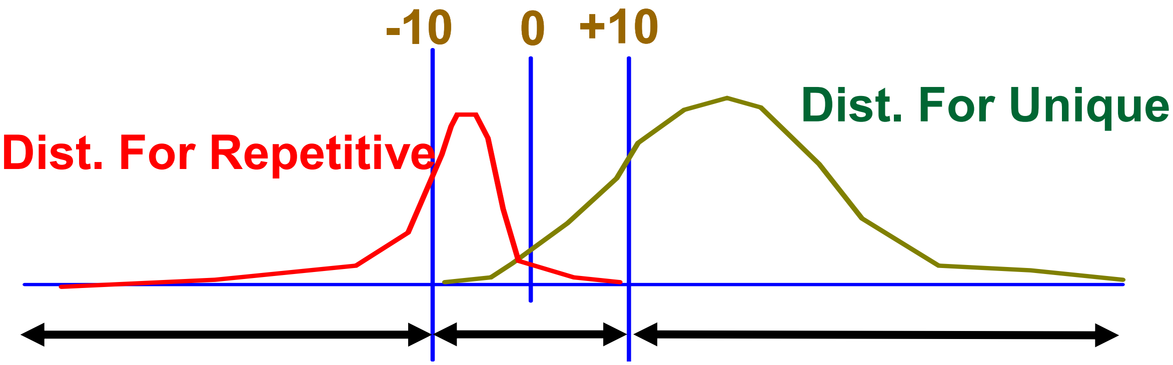


Arrival Intervals

Repetitive DNA unitig



Discriminator Statistic is log-odds ratio of probability unitig is unique DNA versus 2-copy DNA.



-10 0 +10

Dist. For Repetitive

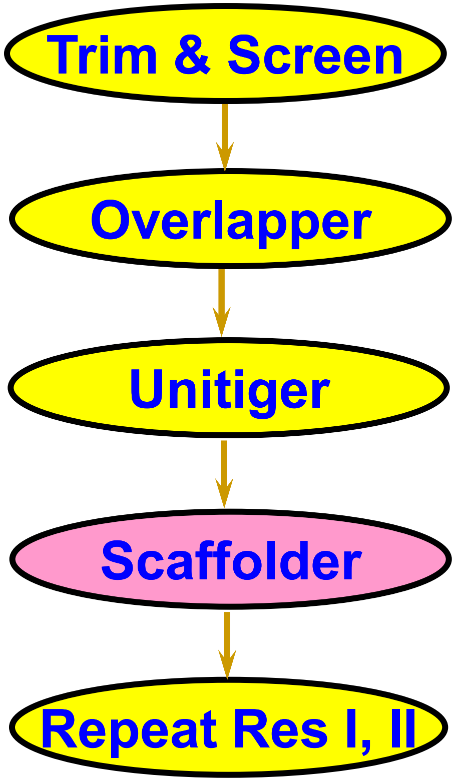
Dist. For Unique

Definitely Repetitive

Don’t Know

Definitely Unique

Celera Assembler



Trim & Screen

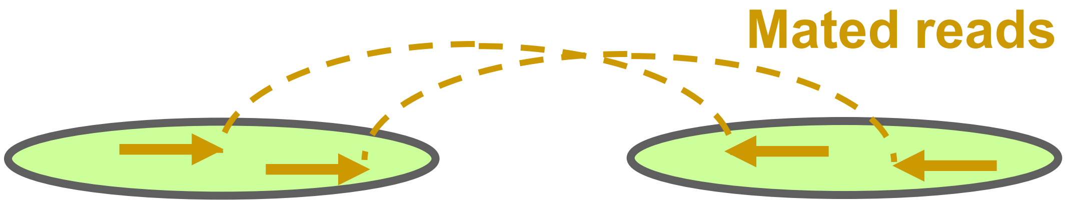
Overlapper

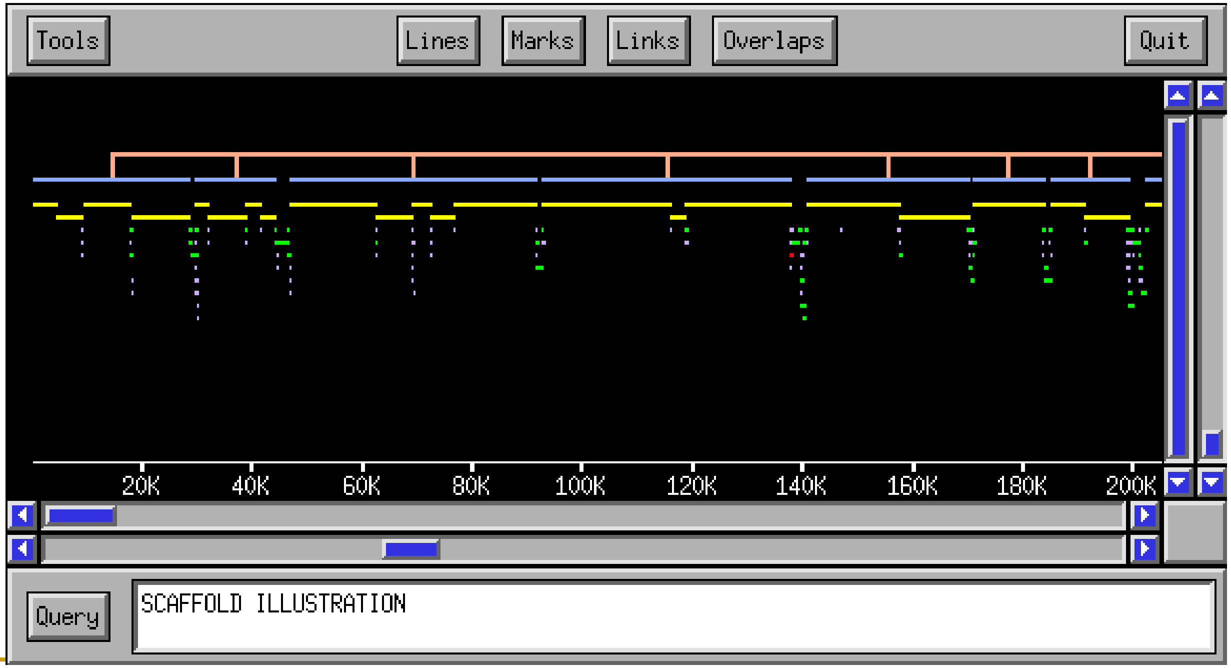
Unitiger

Scaffolder

Repeat Res I, II

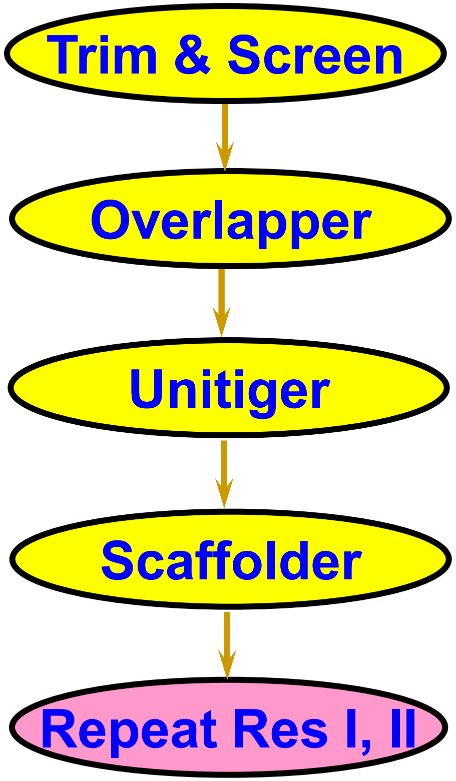
Scaffold U-unitigs with confirmed pairs



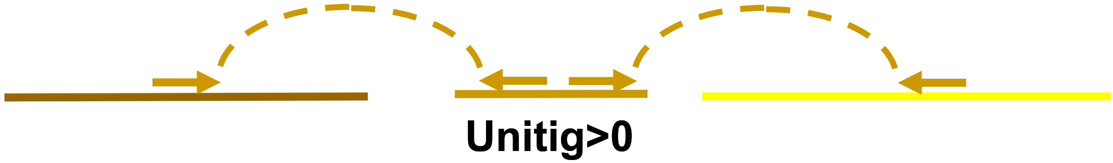


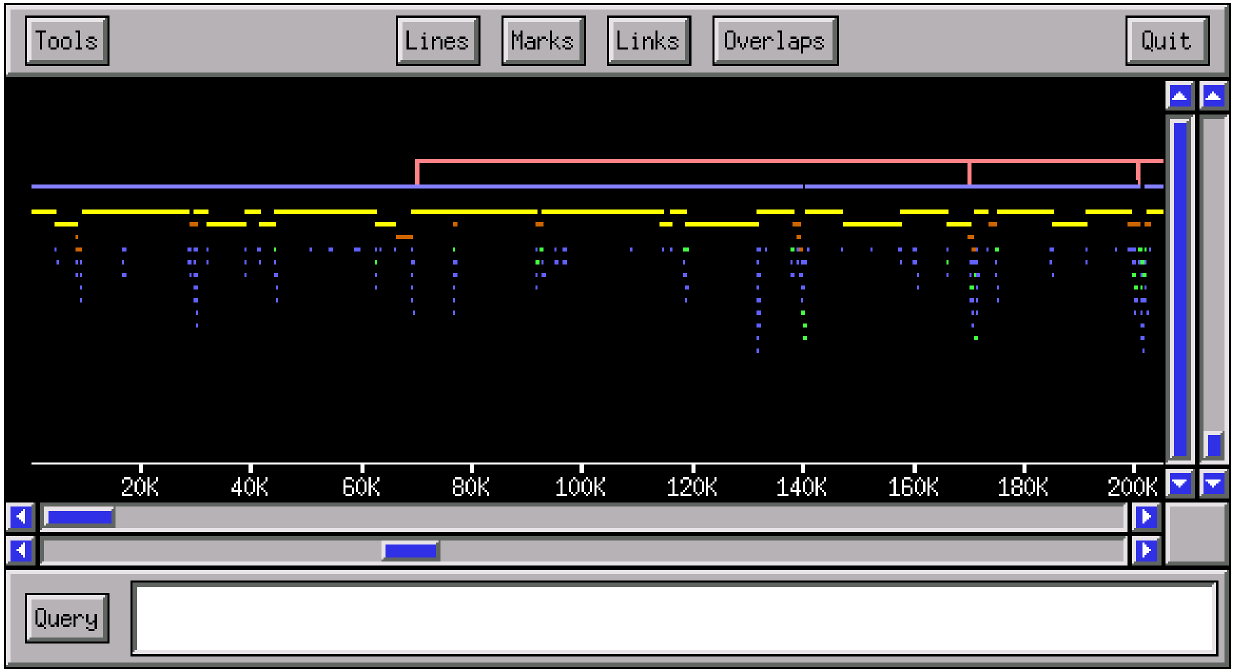
SCAFFOLD ILLUSTRATION

Celera Assembler



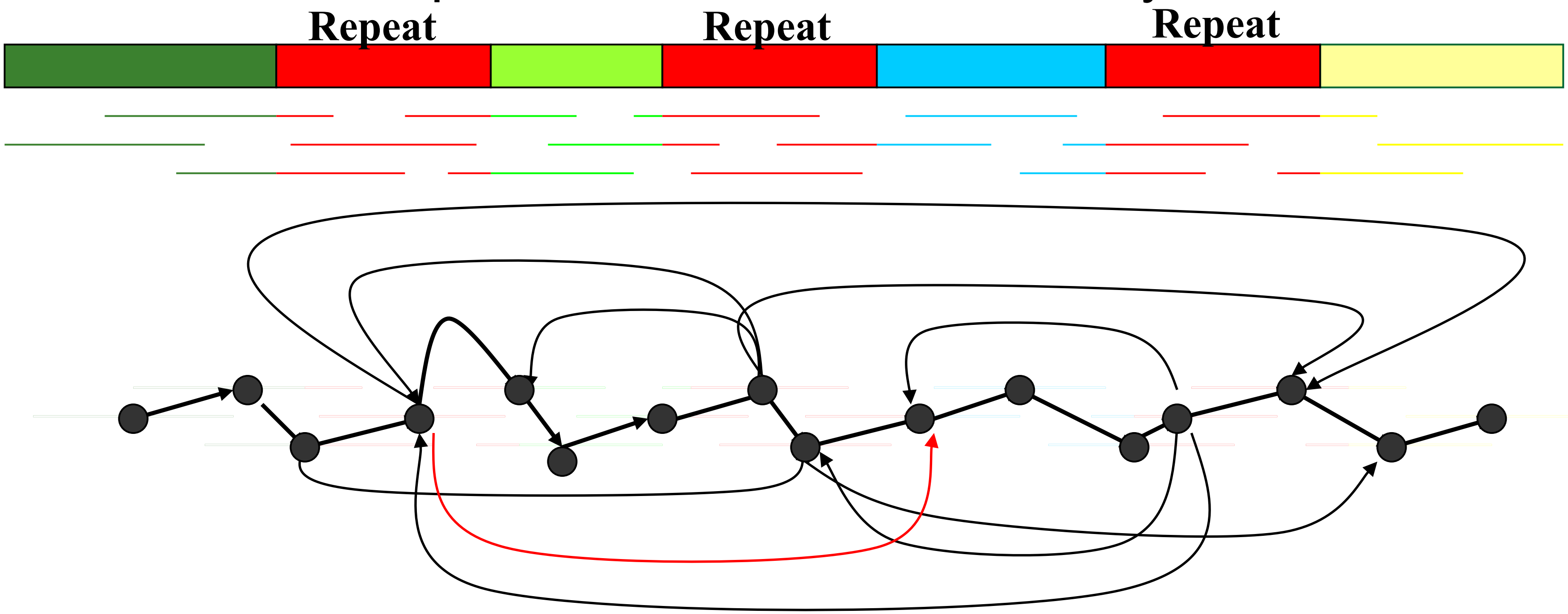
Fill repeat gaps with doubly anchored positive unitigs





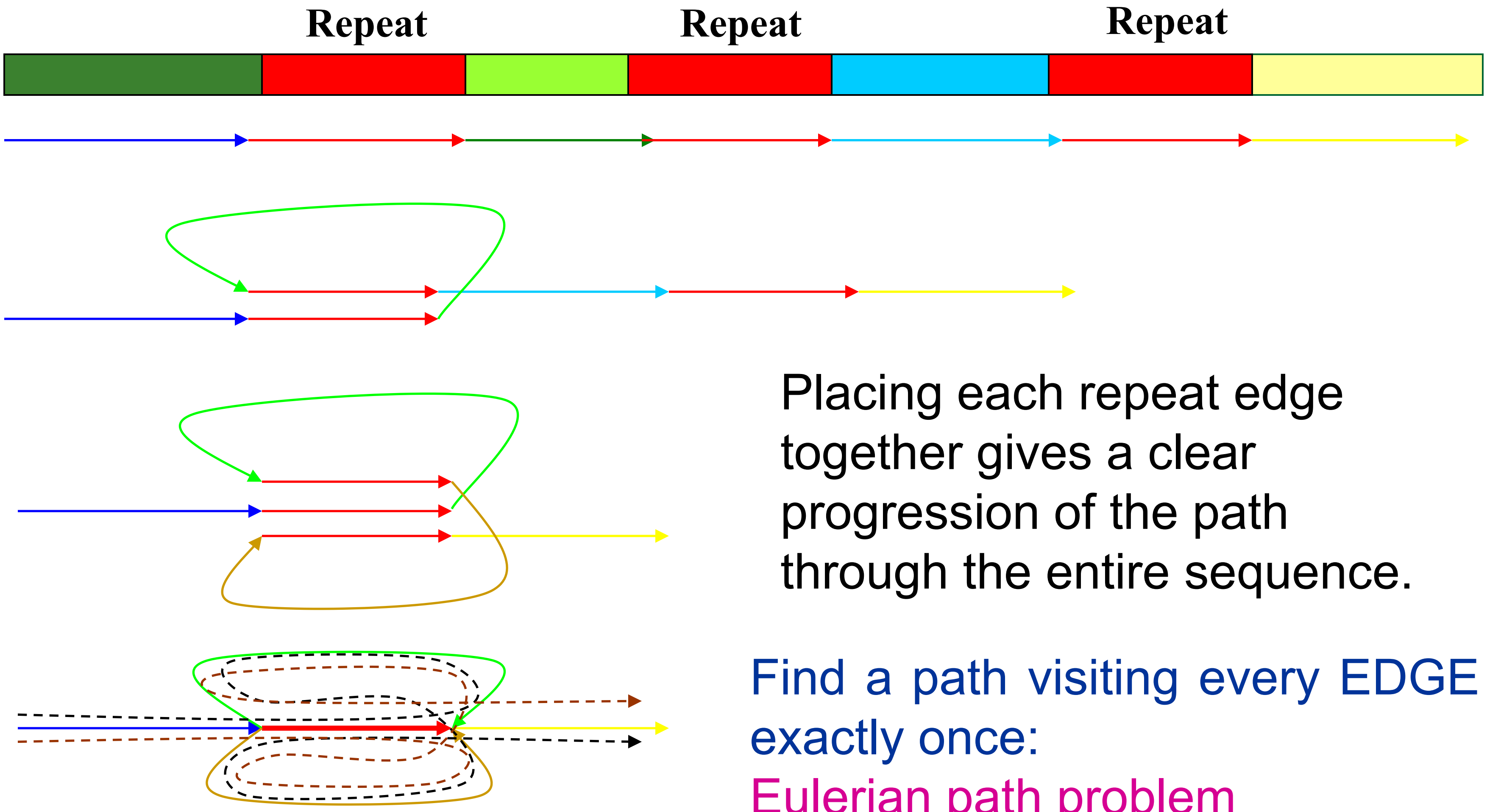
Overlap Graph: Hamiltonian Approach

Each vertex represents a read from the original sequence. Vertices from repeats are connected to many others.



Find a path visiting every VERTEX exactly once: Hamiltonian path problem

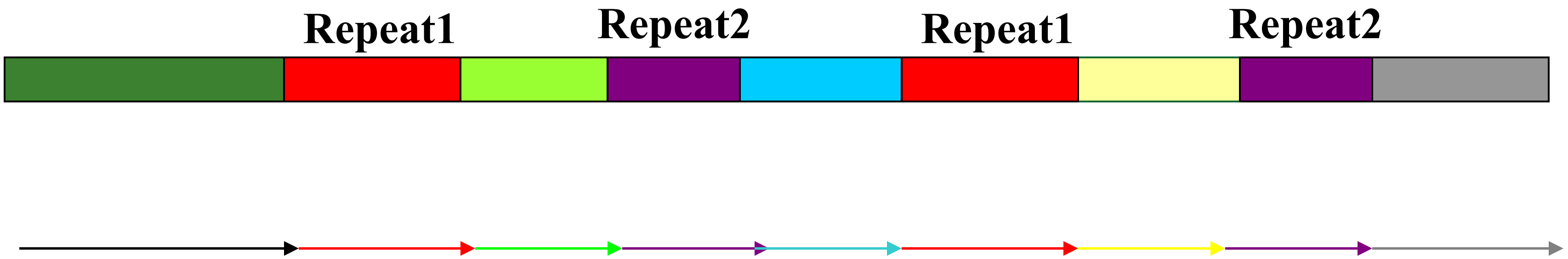
Overlap Graph: Eulerian Approach

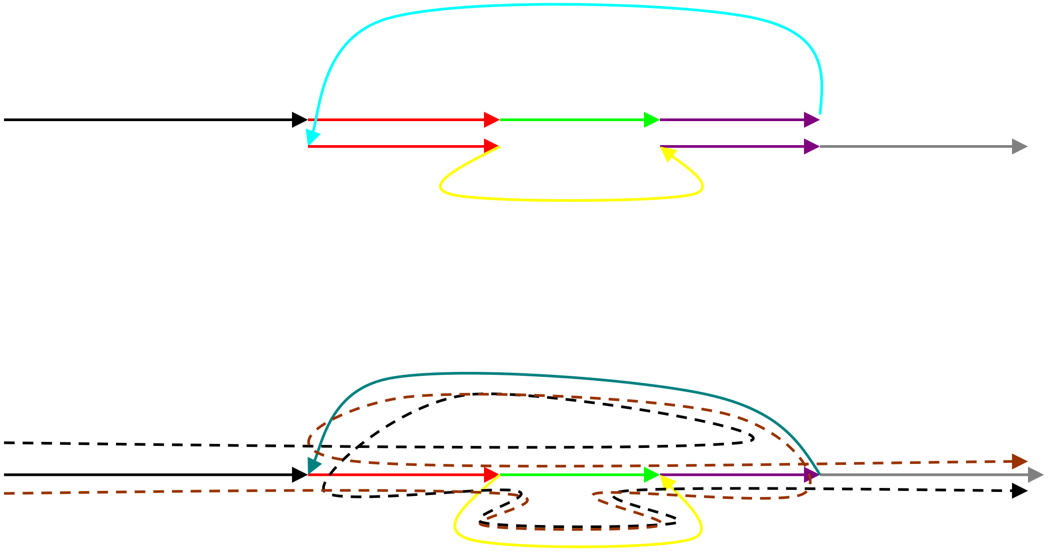


Placing each repeat edge together gives a clear progression of the path through the entire sequence.

Find a path visiting every EDGE exactly once:
Eulerian path problem

Multiple Repeats





Can be easily constructed with any number of repeats

Pre-assembly

NGS ERROR CORRECTION

Ideally

…ATGTTTT…

…ACGTATT…

…ATGTTTT…

…ACGTTTT…

…ATGTTTT…

…ATGTTCT…

reference

… ACGTTAATGTTTTAGTATCGGAAATTACG…

…ATGTTTT…

…ATGTTTT…

…ATGTTTT…

…ATGTTTT…

…ACGTATT…

…ACGTTTT…

Challenges

- Unknown reference genome
- Billions of reads
- Non-uniform error distribution
- Non-uniform genome sampling
- Polymorphisms
- Repeats

Approaches

- Spectrum alignment problem:
  - Chaisson et al., 2004, 2008; Chin et al., 2009; Quake (Kelley et al., 2010); Reptile (Yang et al., 2010)
- Suffix tree:
  - SHREC (Schroder et al., 2009)
  - SHREC (Salmela and Schroder, 2010)
- Alignment based:
  - CORAL (Salmela, 2011)
- Most incorporate the base quality values

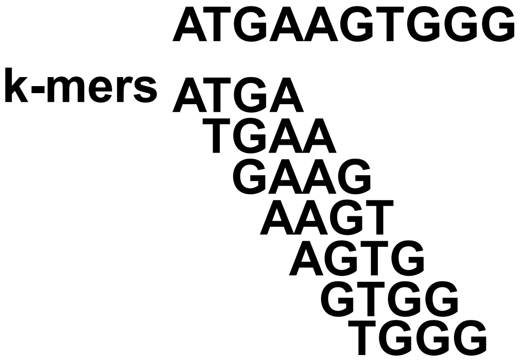
COUNTING KMERS

Counting k-mers for assembly

- Error correction
    - Erroneous reads will have low-frequency k-mers
- Contamination detection
    - Sequence from DNA contamination will be represented at a very low coverage
- Repeat detection
    - Very high frequency k-mers: repeat/duplication
    - Handle accordingly
- k-mers in NGS data sets can easily overwhelm memory capacity

Counting k-mers

- Given sequencing reads count how many times each k-mer occurs
- De Bruijn graph assemblers
  - Euler (Pevzner et al. 2001)
  - Velvet (Zerbino et al. 2008)
  - Allpaths (Butler et al. 2008)
  - ABySS (Simpson et al. 2009)
  - SOAPDenovo (Li et al. 2010)
- Error Correction: Quake (Kelley et al. 2010)
- k-mer counters: Jellyfish (Marçais et al. 2011), BFCounter (Melsted et al., 2011)



```
      ATGAAGTGGG
k-mers ATGA
        TGAA
         GAAG
          AAGT
           AGTG
            GTGG
             TGGG
```

Memory usage

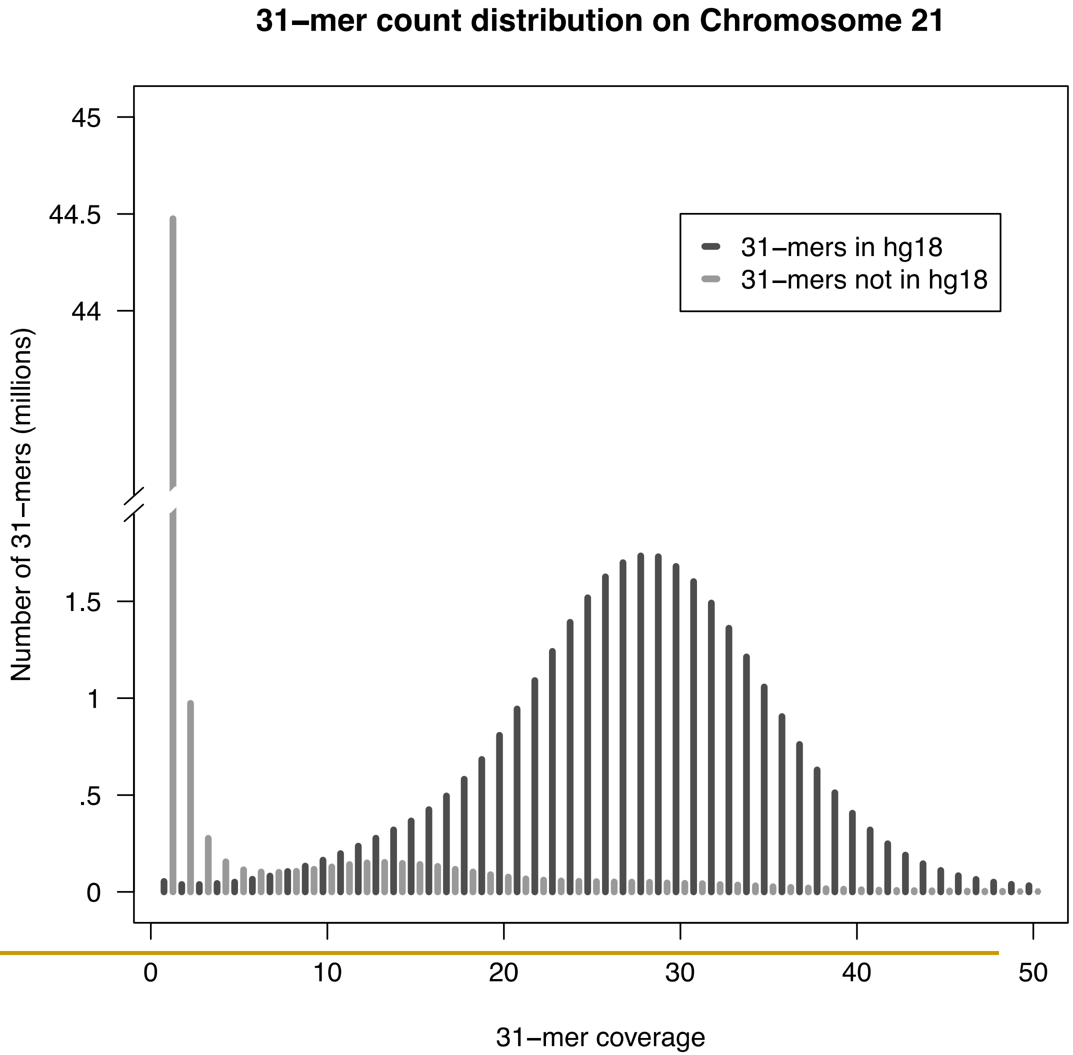
- Simple method

Store each k-mer in a
hash table with a counter

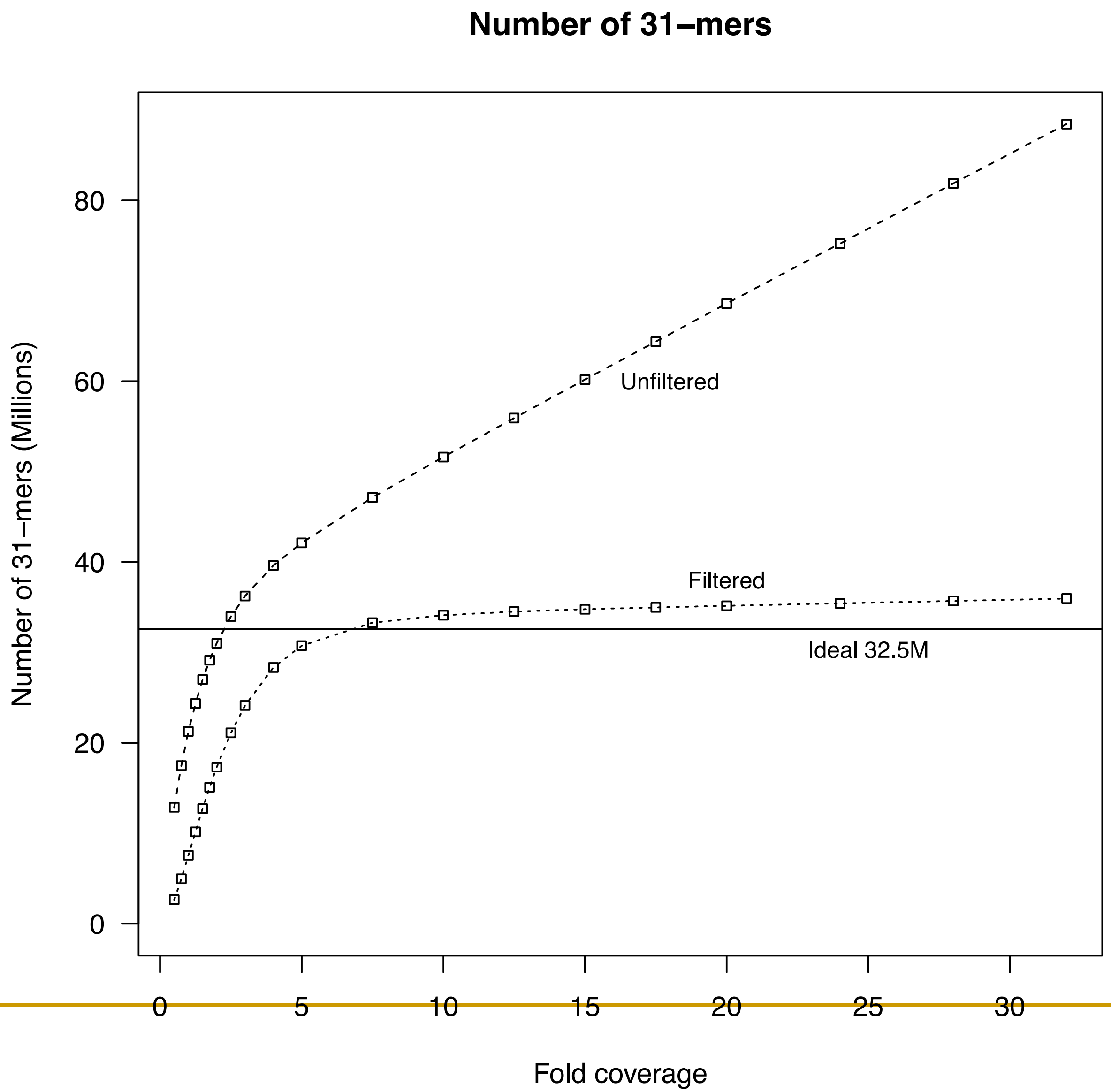
- Memory needed
    - store canonical k-mers
    - 2 bits for each of A,C,G,T
    - k/4 bytes per k-mer (k=31, 8 bytes)
    - 1-2 bytes per counter
    - +10% hash table overhead
- For a genome of size G, expect to see up to G distinct k-mers (2.5-3 billion for Human)
- ~ 36 Gb of memory

Number of k-mers

- This ignores the effect of sequencing errors
- 31-mers in reads aligned to Chr21
- Illumina 100x100 32-fold coverage
- Mapped 31-mers to reference
- 99.9% of unique k-mers are errors



Removing unique k-mers

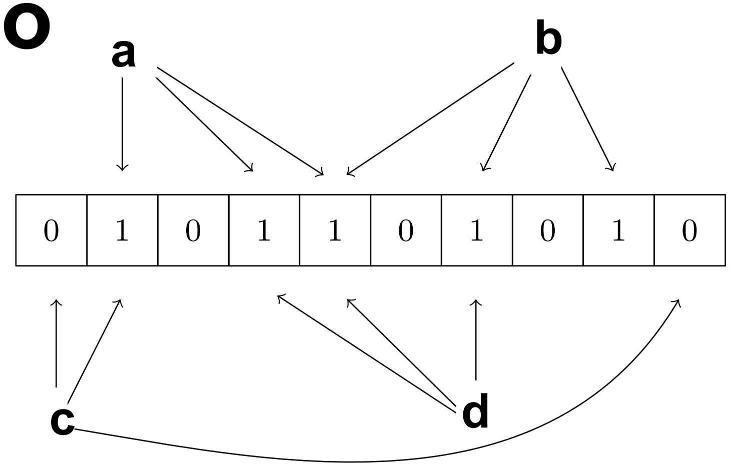


Bloom filter

- Bloom filter encodes a set of k-mers
- Uses a bit array B of length m and d hash functions
  - to insert x, we set $B[h_i(x)] = 1$, for $i=1,\ldots,d$
  - to query y, we check if $B[h_i(y)]$ all equal 1, for $i=1,\ldots,d$
- Need an estimate for n, the number of k-mers to insert

Bloom filter example

- a and b **are inserted in to a Bloom filter with m = 10, n=2, d=3**
- c is not in the set, since **some bits are 0**
- d has not **been inserted, but is still reported in the set, a false positive**
- Bloom filters have no false negatives



0	1	0	1	1	0	1	0	1	0

Bloom filter

- Storing n k-mers in m bit array with d hash functions has a false positive rate of

$$\approx(1-e^{-d\,n/m})^{d}$$

- Given n and m, the optimal d is ≈m/n ln(2)
- Examplem = 8n, d=5 gives 2.16% fpr
  m = 6n, d=4 gives 5.6% fpr
  m = 4n, d=3 gives 14.6% fpr
- m=8n, corresponds to storing 1 byte per k-mer

Algorithm

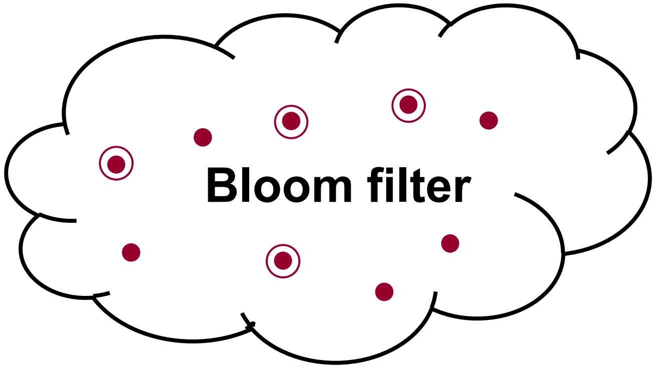
- Use a Bloom filter and a hash table

ATGAAGTGGG

k-mers ATGA ←
TGAA ←
GAAG ←
AAGT ←
AGTG ←
GTGG ←
TGGG ←

AGTGGGTGAA

k-mers AGTG ←
GTGG ←
TGGG ←
GGGT ←
GGTG ←
GTGA ←
TGAA ←



First Pass

Second Pass

Hash table

TGGG	0 1 2
AGTG	0 1 2
~~GTGA~~	~~0 1~~
GTGG	0 1 2

Algorithm

- This scheme guarantees
  - k-mers seen twice will be in the hash table
  - some unique k-mers will slip through
  - second pass gives accurate counts and allows to discard false positives
- Memory usage
  - full for k-mers in hash table (~ 9 bytes)
  - minimal for k-mers in bloom filter (~ .5-1 bytes)

Results whole genome

- 25-mers in 36 bp reads
- 2.37 billion distinct 25-mers in hg18
- 12.18 billion 25-mers in the sequencing data
  - 9.35 billion unique
  - 2.83 billion with coverage 2 or greater

Program	Time (hrs)	Memory (G)
BFCounter	23.82	42
Naïve	> 26.83	>128

NEXT: DE BRUIJN GRAPHS