

Assembling Genomes

BCH394P/364C Systems Biology / Bioinformatics

Edward Marcotte, Univ of Texas at Austin



Beijing Genomics Institute



“If it tastes good you should sequence it...
you should know what's in the genes of that species”
Wang Jun, Chief executive, BGI

(Wikipedia)

**The NovaSeq in the UT GSAF core generates
>1.4 terbases of sequence in a 1-day run**

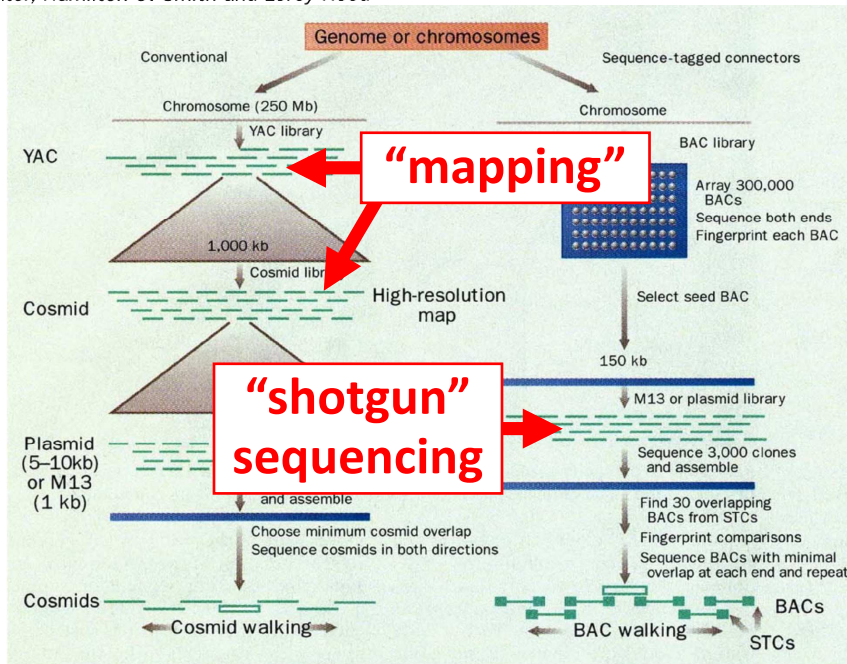


→ Many millions of 75-150 bp reads



A new strategy for genome sequencing

J. Craig Venter, Hamilton O. Smith and Leroy Hood



NATURE · VOL 381 · 30 MAY 1996

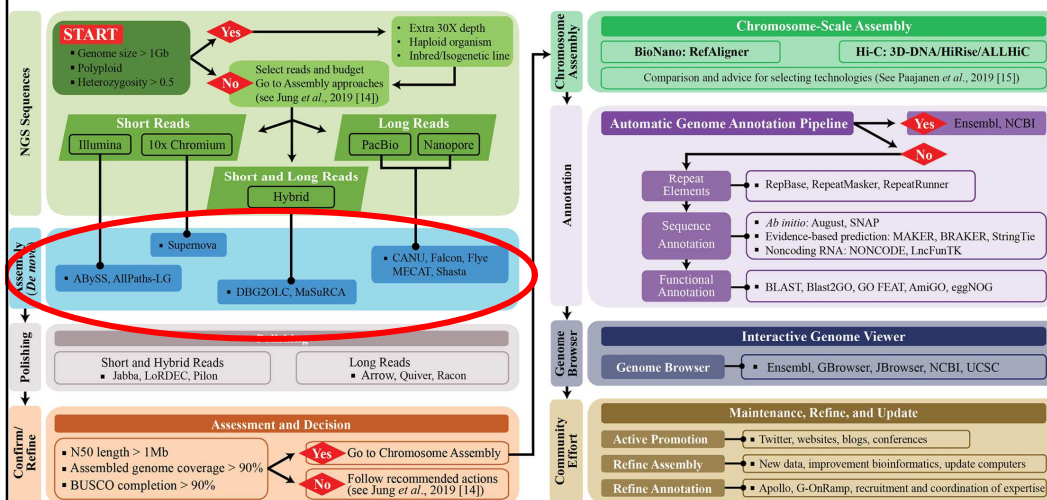
(Translating the cloning jargon)

CLONE LIBRARIES USED FOR GENOME MAPPING AND SEQUENCING

Vector	Human-DNA insert size range	Number of clones required to cover the human genome
Yeast artificial chromosome (YAC)	100–2,000 kb	3,000 (1,000 kb)
Bacterial artificial chromosome (BAC)	80–350 kb	20,000 (150 kb)
Cosmid	30–45 kb	75,000 (40 kb)
Plasmid	3–10 kb	600,000 (5 kb)
M13 phage	1 kb	3,000,000 (1 kb)

NATURE · VOL 381 · 30 MAY 1996

Contemporary genome assembly is fairly complex, but at its core are assembly algorithms that grew from the shotgun concept



Twelve quick steps for genome assembly and annotation in the classroom
 PLoS Comp Biology (2020), doi:10.1371/journal.pcbi.1008325

Beverly Micro "Pure White Hell" Jigsaw Puzzle (10,000,000,000 Piece)



Thinking about the basic shotgun concept

- Start with a very large set of random sequencing reads
- How might we match up the overlapping sequences?
- How can we assemble the overlapping reads together in order to derive the genome?

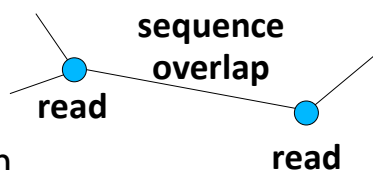
Thinking about the basic shotgun concept

- At a high level, the first genomes were sequenced by comparing pairs of reads to find overlapping reads
- Then, building a graph (*i.e.*, a network) to represent those relationships
- The genome sequence is a “walk” across that graph

The “Overlap-Layout-Consensus” method

Overlap: Compare all pairs of reads
(allow some low level of mismatches)

Layout: Construct a graph describing the overlaps

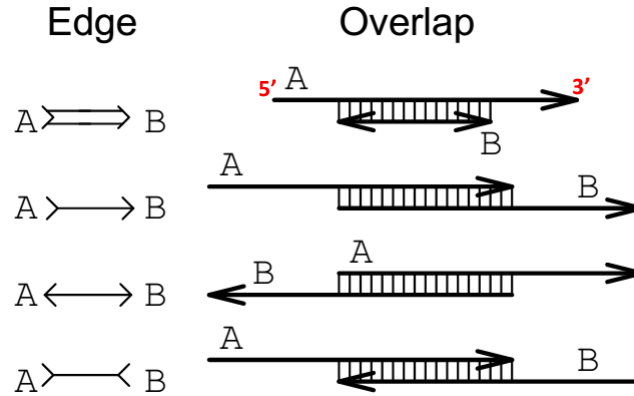


Simplify the graph

Find the simplest path through the graph

Consensus: Reconcile errors among reads along that path to find the consensus sequence

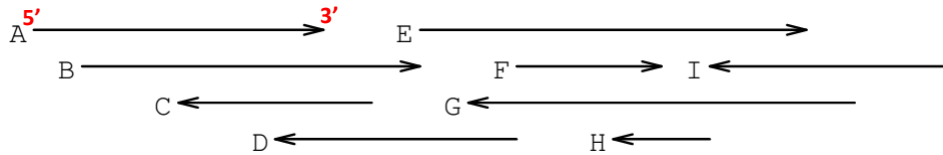
Building an overlap graph



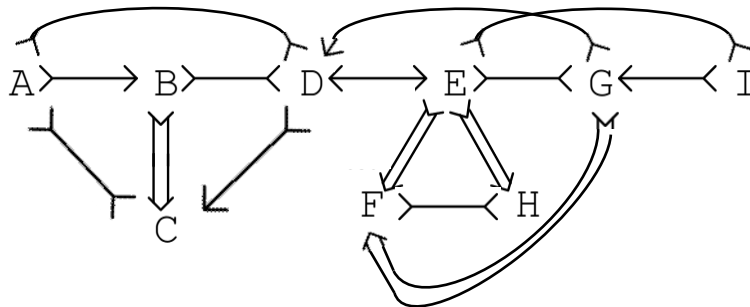
EUGENE W. MYERS. *Journal of Computational Biology*. Summer 1995, 2(2): 275-290

Building an overlap graph

Reads

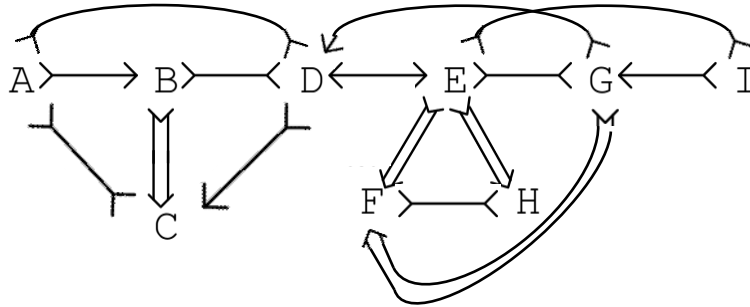


Overlap graph



EUGENE W. MYERS. *Journal of Computational Biology*. Summer 1995, 2(2): 275-290 (more or less)

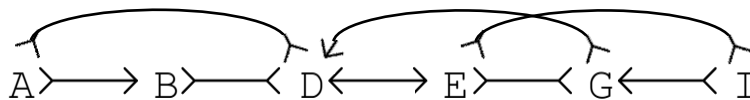
Simplifying an overlap graph



1. Remove all contained nodes & edges going to them

EUGENE W. MYERS. *Journal of Computational Biology*. Summer 1995, 2(2): 275-290 (more or less)

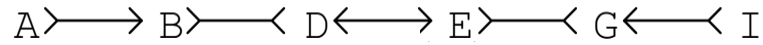
Simplifying an overlap graph



2. Transitive edge removal:
Given $A - B - D$ and $A - D$, remove $A - D$

EUGENE W. MYERS. *Journal of Computational Biology*. Summer 1995, 2(2): 275-290 (more or less)

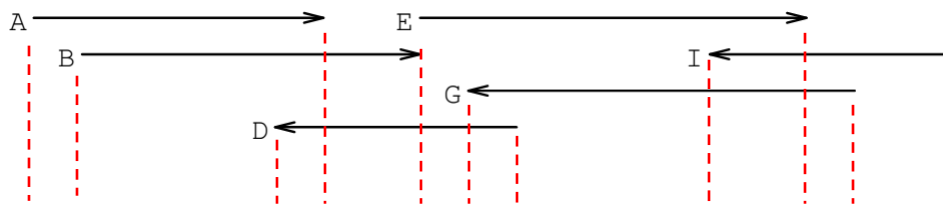
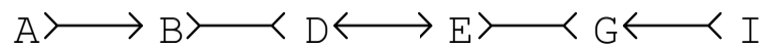
Simplifying an overlap graph



3. If un-branched, calculate consensus sequence
If branched, assemble un-branched bits and then decide how they fit together

EUGENE W. MYERS. *Journal of Computational Biology*. Summer 1995, 2(2): 275-290 (more or less)

Simplifying an overlap graph

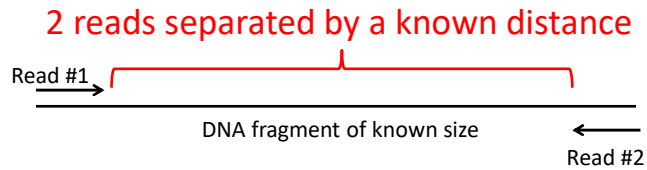


“contig” (assembled contiguous sequence)

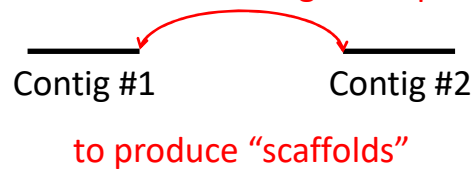
EUGENE W. MYERS. *Journal of Computational Biology*. Summer 1995, 2(2): 275-290 (more or less)

This basic strategy was used for most of the early genomes.

Also useful: “mate pairs”



Contigs can be ordered using these paired reads



GigAssembler (used to assemble the public human genome project sequence)

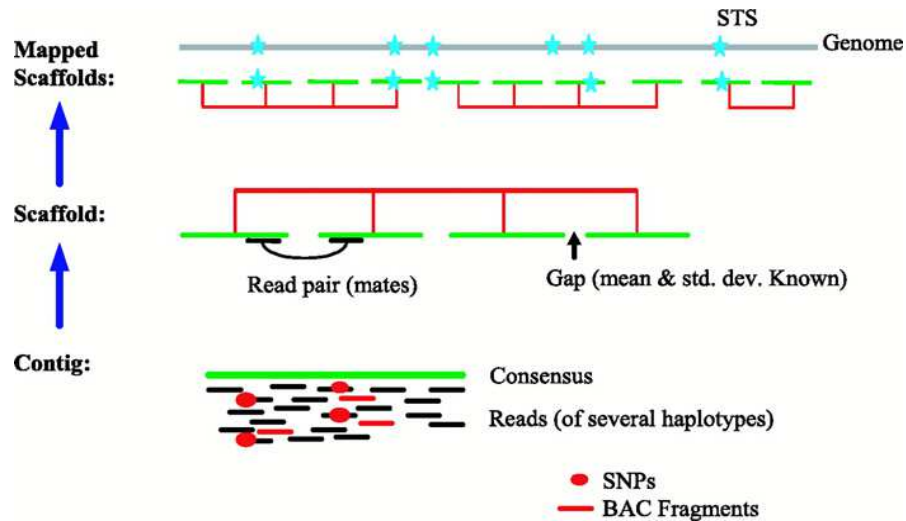


Jim Kent

David Haussler

Let's take a little walk through history to see what they did...

Whole genome Assembly: big picture



<http://www.nature.com/scitable/content/anatomy-of-whole-genome-assembly-20429>

GigAssembler – Preprocessing

1. Decontaminating & Repeat Masking.
2. Aligning of mRNAs, ESTs, BAC ends & paired reads against initial sequence contigs.
 - psLayout → BLAT
3. Creating an input directory (folder) structure.

```
chr1/  
chr1/contig1.e  
chr1/contig1.a  
chr1/contig1.c  
chr1/contig1.b  
chr1/contig1.d  
chr3/  
chr2/  
chr2/contig2.d  
chr2/contig2.b  
chr2/contig2.a  
chr2/contig2.c
```

RepBase + RepeatMasker

```
taejoon@fourierseq:~/RepBase/RepBase15.05.fasta$ ls -a
.
..
dcotrep.ref  mamsub.ref  rodsup.ref
angrep.ref   diarep.ref  mcotrep.ref  simple.ref
drorrep.ref  drorep.ref  mousub.ref  spurep.ref
appendix     fngrep.ref  nemrep.ref  synrep.ref
athrep.ref   fugrep.ref  oryrep.ref  tmpplanrep.ref
bctrep.ref   grasrep.ref plnrep.ref  tmpnemrep.ref
cbrrep.ref   humrep.ref  prirep.ref  tmpxenrep.ref
chlrep.ref   humsub.ref  prisub.ref  version
cinrep.ref   invsub.ref  pseudo.ref  vrtrep.ref
cinunc.ref   mamrep.ref  rodrep.ref  zebrep.ref
```

```
>MER51D ERV1 Homo sapiens
tgaggcaggagaaaaatagcagaggggaattggaagttggaataaaggagaaatgagtaaaagcangagagca
gaagcaaggtaaaagggcgggtgagcaagaagcaagataagaagcagaagttgagcagccaaaacaaaag
taagatnanaaagaagtggatgaaggaccacatggctggctagatccagaccaaacagtaagggcgag
ctctcagagatgggcatgtacattagagagaaaaagtatcttaaaatgaccccgatgataatcagct
cattaaagctcatgcatatggagctgcatatcatgcaigtacttaaaatgggagtgaggtagcgcga
agawgtcacaagcacacaggggcatagkattaagtaactaagcaaccacccatcaatcaaaagcgaga
tgcctggctagagattaggcagccttgggaagagaagaaaaaaacacataaaaagacccaaggtacac
caactgacgcctgattctcatttcgagaggtcagcccactctcccctctctgagagtgtaaacctgctg
taataaaactttgctgcttctgctatctggtggtgctcttccaattcttggggacaccaagagcct
ggaactgcacrgcaccakctgtaaca
>MIRb SINE2/ERNA Mammalia
cagaggggcagcgtggtgagtggaagagcacgggcttggagtcaggcagacctgggttcgaatcctg
gctctgcacttactagctgtgtgaccttggcaagtcacttaacctctctgagcctcagtttctctatc
tgtaaaaagggaataataatcctacctcagcaggttggtaggattaaatgagataatgcatgtaaa
ggccttagcacagtcctggcacacagtaagcgtcaataaatggtagctctattatt
>LTR45 ERV1 Homo sapiens
tgtaaccgcccggaccgcccacaaactgggctactctgttgatacaaaaatgcaagttacctgttaggta
taacagagcccaaacgcaagtcagtagccgggcatgtgcaatagaaaaagcttgaccttaacaa
caccagaaccaatgattctcccctcggaaaccaagaagacgggacatgaccggaacctgaatgcccga
actctttcagaagcaaaaggggtcgttggccgggaagatctggggctaaaatctgctcaacatacctta
ccgtaaatggtcaaatggagcctcactcagacctgccaagccaacattcctaaactctttccctt
gcctctgatcccttaaaactgcccagaccacaaatggggagacagattgagcccactcctgctct
ccttctggccggttttcaataaagcctttctttctcaaaagctgggcatagttatggctctgt
gtcatcaggcagcaagccatttctcgataaca
>MER80B HAT Homo sapiens
cagggctctttaaccagaggtccatggatgggcttcaggaggtctgtgaacctctgaaattatatacaa
aaatgttggtatattgcatatattgtttttctgggagaggttcacagcttctcatcagattctcaa
aggggtctatgatctmaaaaaggttaagaagccctg
```

GigAssembler: Build merged sequence contigs (“rafts”)

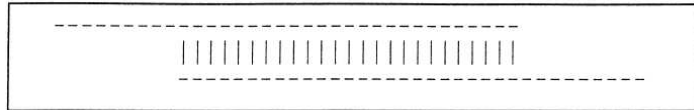
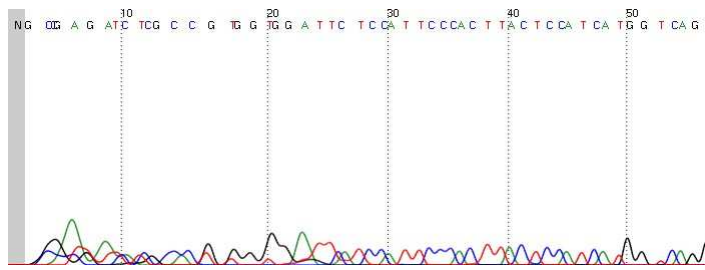


Figure 1 Two sequences overlapping end to end. The sequences are represented as dashes. The aligning regions are joined by vertical bars. End-to-end overlap is an extremely strong indication that two sequences should be joined into a contig.

Sequencing quality (Phred Score)



>gnljtj2299297598 name:fw01a01.x1 NCBI Accession: [AC243936](#) Mate pair: [2299297599](#)

Quality score: not available >=0 - <20 >=20 - <40 >=40 - <60 >=60 - <80 >=80 - <100

0	3	6	6	6	6	6	8	8	8	8	8	8	8	8	8	6	6	6	6	7	13	19	10	10	10	22	13	22	22	35	36	30																			
35	32	35	36	35	36	35	28	28	28	30	29	28	28	28	25	23	19	21	3	0	3	13	14	10	10	15	15	15	13	9	6	6	7																		
7	6	6	6	6	6	6	11	11	12	16	11	10	12	22	23	30	24	23	24	31	37	44	53	53	57	38	38	35	35	35	31	31	37																		
37	37	27	21	15	8	6	6	6	6	6	14	12	16	18	28	25	31	34	43	37	36	36	50	50	50	53	53	57	57	57	57	57	57																		
57	57	57	57	57	59	59	57	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59																	
68	68	68	68	68	68	50	51	51	51	51	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68																
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68															
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68														
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68													
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68												
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68											
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68										
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68									
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68								
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68							
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68						
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68					
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68				
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68			
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68			
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68		
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68		
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68

Sequencing quality (Phred Score)

$$Q = -10 \log_{10} P \leftarrow \begin{array}{l} \text{Base-calling} \\ \text{Error} \\ \text{Probability} \end{array}$$

or

$$P = 10^{-\frac{Q}{10}}$$

Phred quality scores are logarithmically linked to error probabilities

Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90 %
20	1 in 100	99 %
30	1 in 1000	99.9 %
40	1 in 10000	99.99 %
50	1 in 100000	99.999 %

http://en.wikipedia.org/wiki/Phred_quality_score

We're going to skip the remaining details of GigAssembler (mainly of historical interest now) to get to the key strategy for assembling all of the various contigs and paired end reads into a genome

GigAssembler: Build a "raft-ordering" graph

```

AAAAAAAAAAAAAAAAAAAA
a1a1a1a1  a2a2a2a2a2
BBBBBBBBBBBBBBBBBB
b1b1b1b1b1  b2b2b2
ccccccccccccccccccc
c1c1c1      c2c2c2c2
  
```

Figure 4 Three overlapping draft clones: A, B, and C. Each clone has two initial sequence contigs. Note that initial sequence contigs a1, b1, and a2 overlap as do b2 and c1.

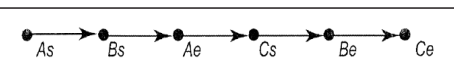


Figure 5 Ordering graph of clone starts and ends. This represents the same clones as in Fig. 4. (As) The start of clone A; (Ae) the end of clone A. Similarly Bs, Be, Cs, and Ce represent the starts and ends of clones B and C.

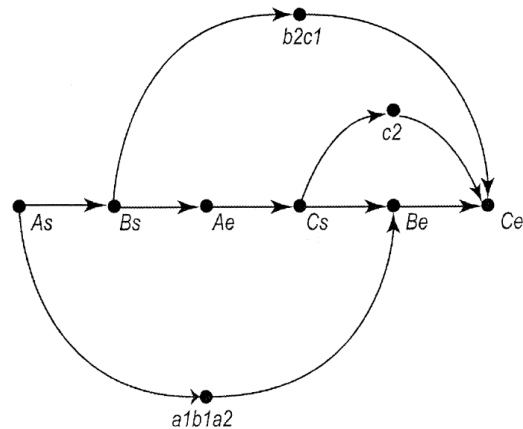


Figure 6 Ordering graph after adding in rafts. The initial sequence contigs shown in Fig. 4 are merged into rafts where they overlap. This forms three rafts: a1b1a2, b2c1, and c2. These rafts are constrained to lie between the relevant clone ends by the addition of additional ordering edges to the graph shown in Fig. 5.

GigAssembler: Build a “raft-ordering” graph

- Add information from mRNAs, ESTs, paired plasmid reads, BAC end pairs: building a “bridge”
 - Different weight to different data type: (mRNA ~ highest)
 - Conflicts with the graph as constructed so far are rejected.
- Build a sequence path through each raft.
- Fill the gap with N's.
 - 100: between rafts
 - 50,000: between bridged barges

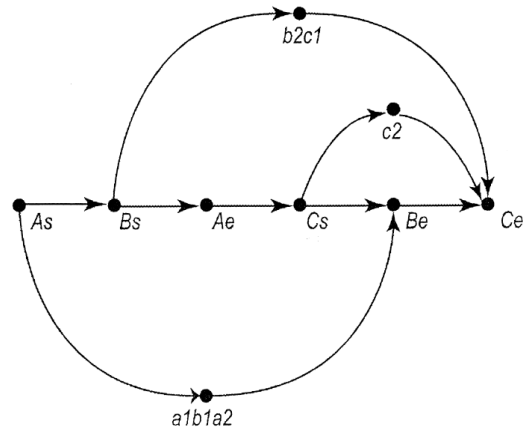
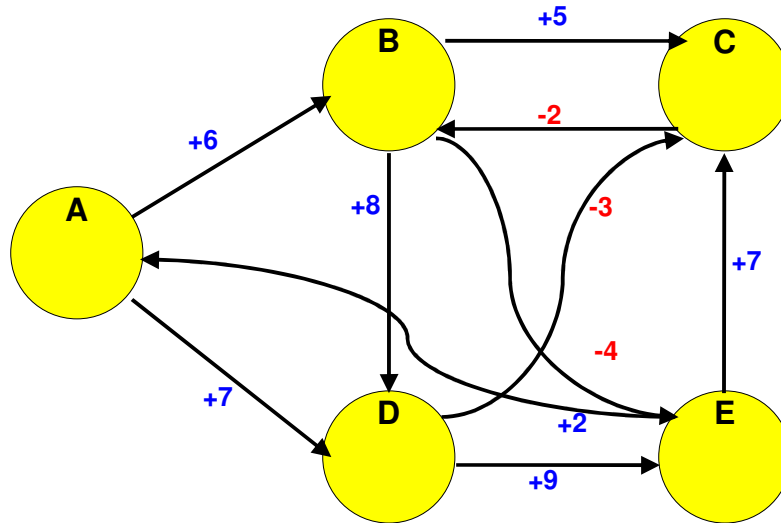


Figure 6 Ordering graph after adding in rafts. The initial sequence contigs shown in Fig. 4 are merged into rafts where they overlap. This forms three rafts: *a1b1a2*, *b2c1*, and *c2*. These rafts are constrained to lie between the relevant clone ends by the addition of additional ordering edges to the graph shown in Fig. 5.

Finding the shortest path across the
ordering graph using the
Bellman-Ford algorithm

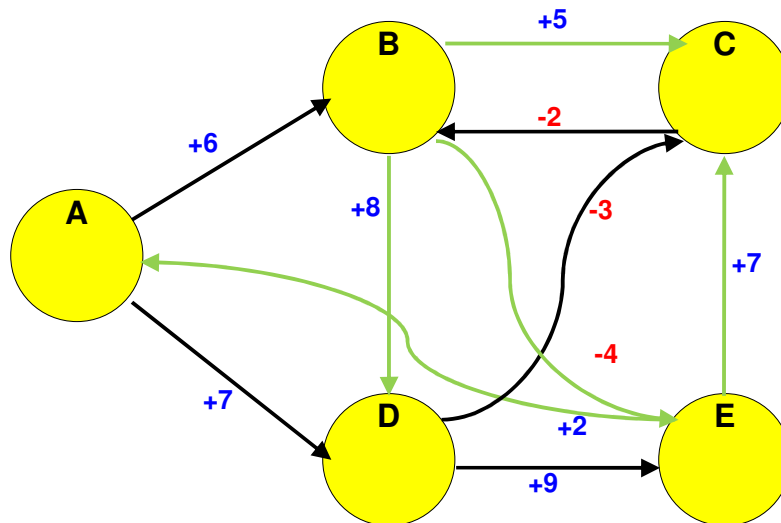
Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where N is the count of nodes)



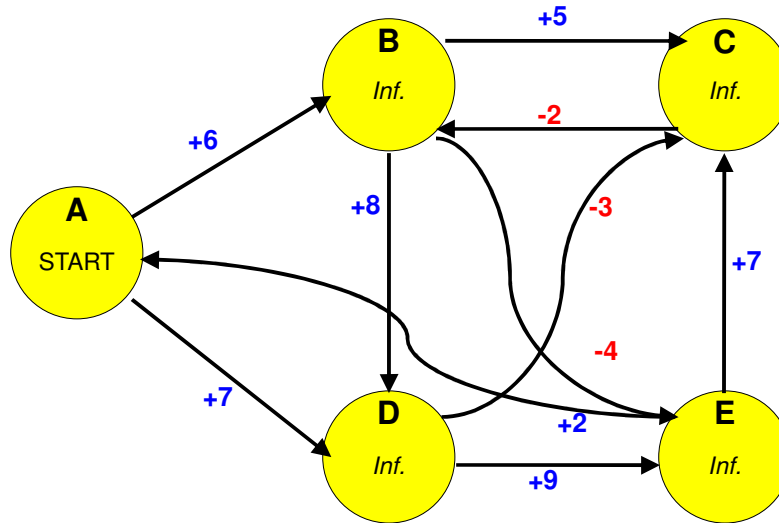
Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where N is the count of nodes)



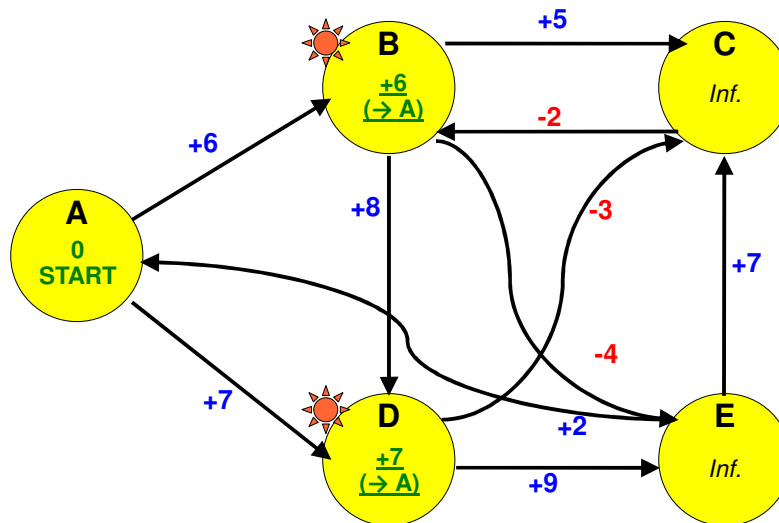
Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where N is the count of nodes)



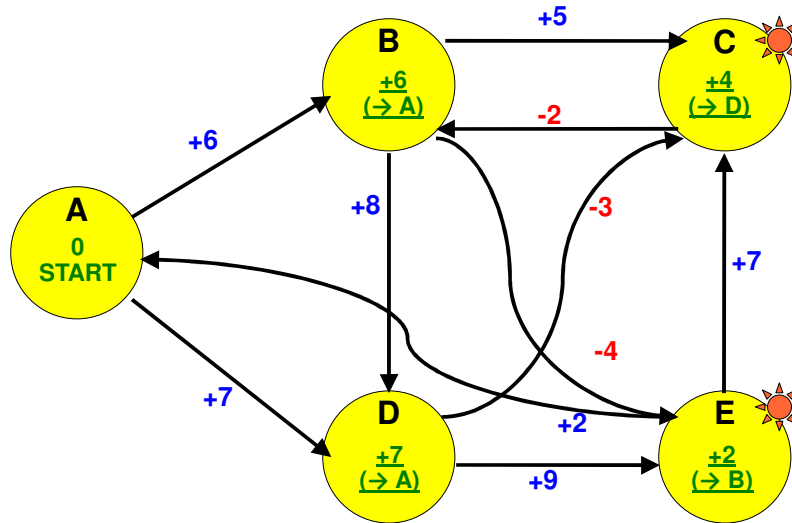
Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where N is the count of nodes)



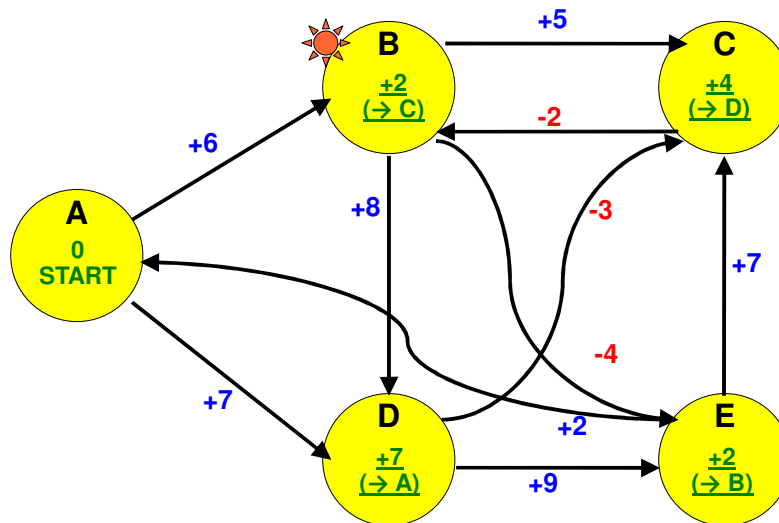
Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where N is the count of nodes)



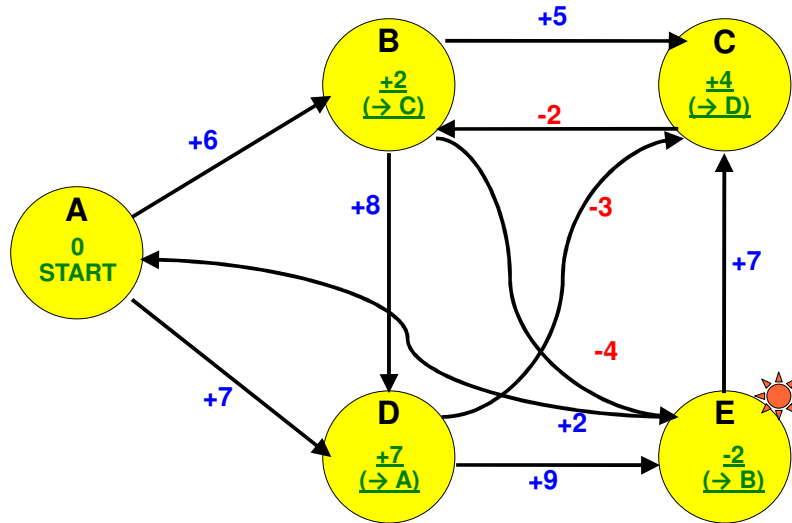
Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where N is the count of nodes)

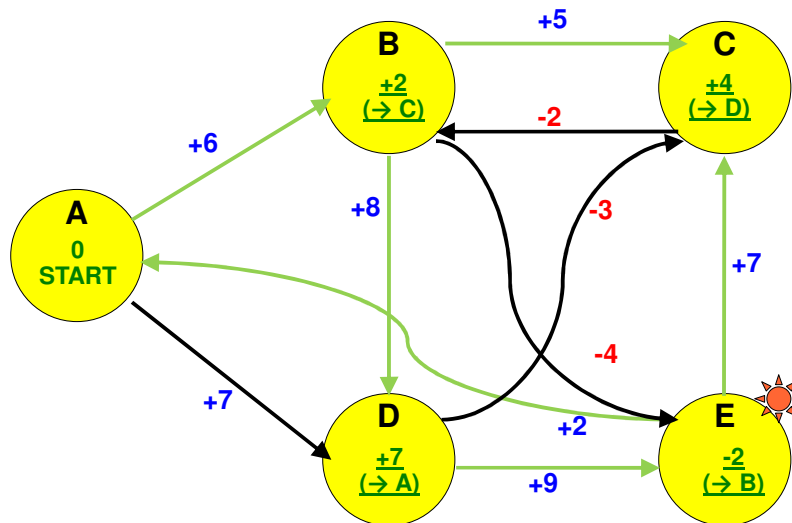


Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where N is the count of nodes)

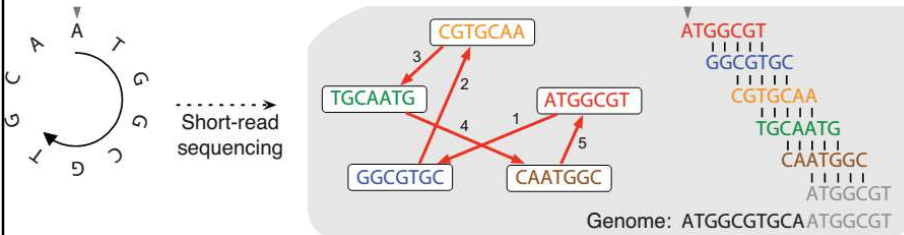


Answer: A-D-C-B-E



Modern assemblers now work a bit differently, using so-called **DeBruijn graphs**:

Here's what we saw before:

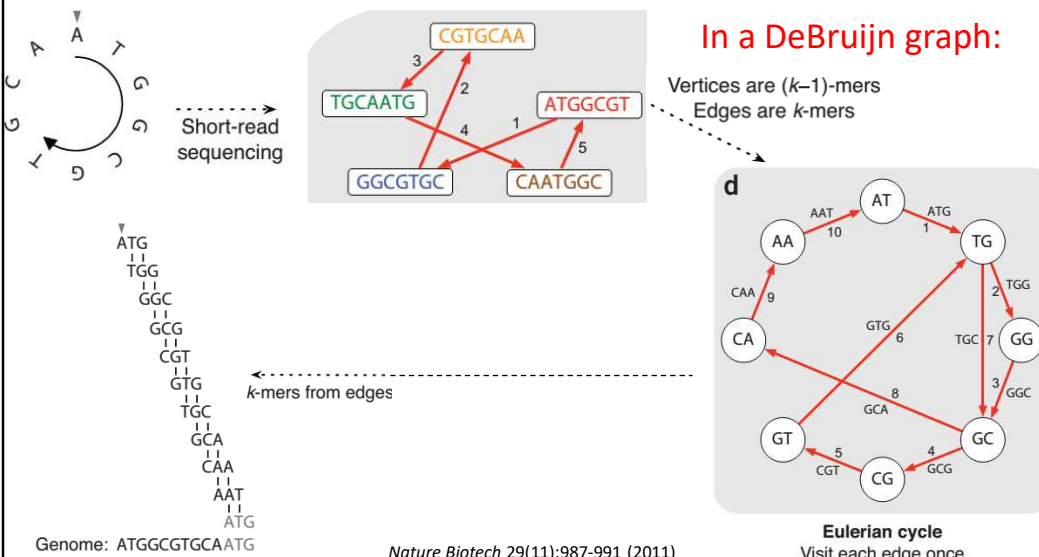


In Overlap-Layout-Consensus:

Nodes are reads
Edges are overlaps

Nature Biotech 29(11):987-991 (2011)

Modern assemblers now work a bit differently, using so-called **DeBruijn graphs**:

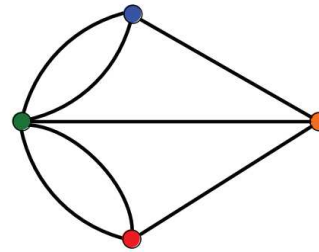


Nature Biotech 29(11):987-991 (2011)

Why Eulerian?

From Leonhard Euler's solution in 1735 to the 'Bridges of Königsberg' problem:

Königsberg (now Kaliningrad, Russia) had 7 bridges connecting 4 parts of the city. **Could you visit each part of the city, walking across each bridge only once, & finish back where you started?**

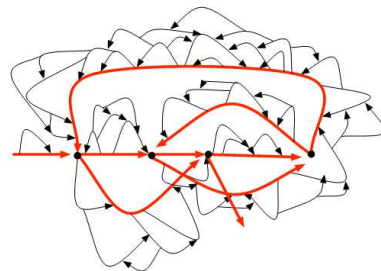
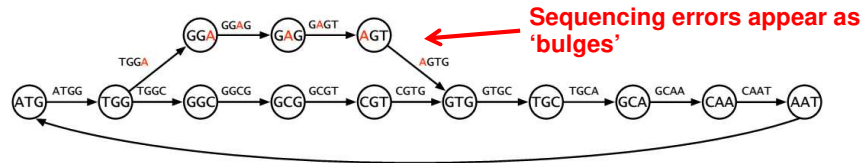
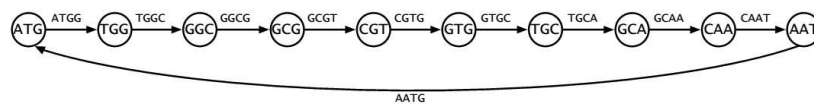


(Visiting every edge once = an *Eulerian* path)

Euler conceptualized it as a graph:
Nodes = parts of city
Edges = bridges

Nature Biotech 29(11):987-991 (2011)

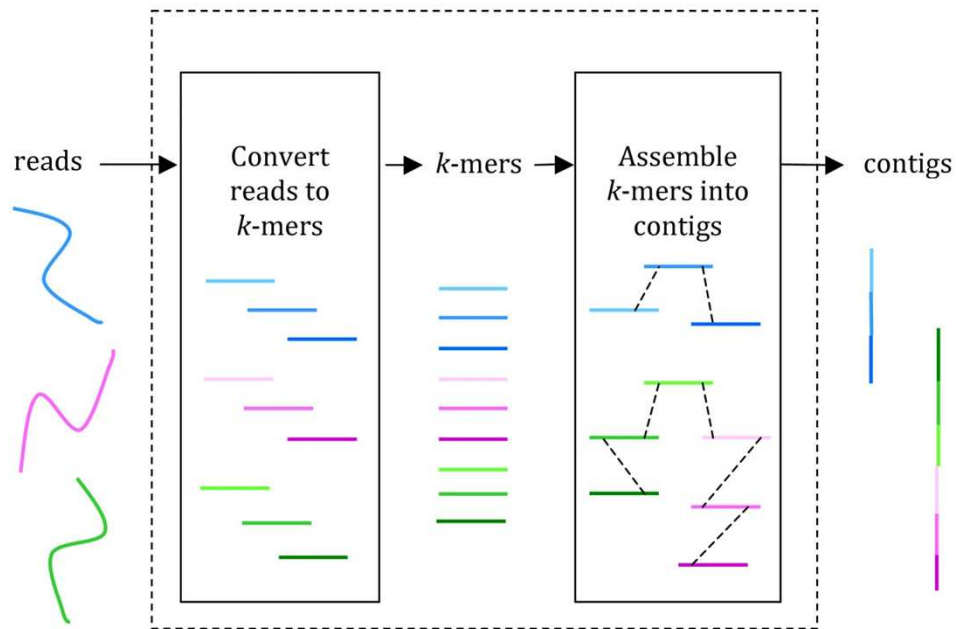
DeBruijn graph assemblers tend to have nice properties, e.g. correcting sequencing errors & handling repeats better



Removing the 'bulges' corrects the errors (e.g. leaves the red path)

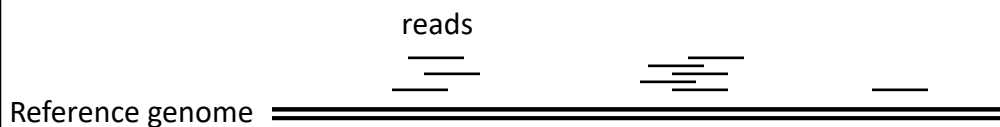
Nature Biotech 29(11):987-991 (2011)

e.g. Velvet, an example algorithm using DeBruijn graphs



Beginner's guide to comparative bacterial genome analysis using next-generation sequence data
Microb Informatics Exp (2013) doi:10.1186/2042-5783-3-2

Once a reference genome is assembled,
new sequencing data can 'simply' be
mapped to the reference.



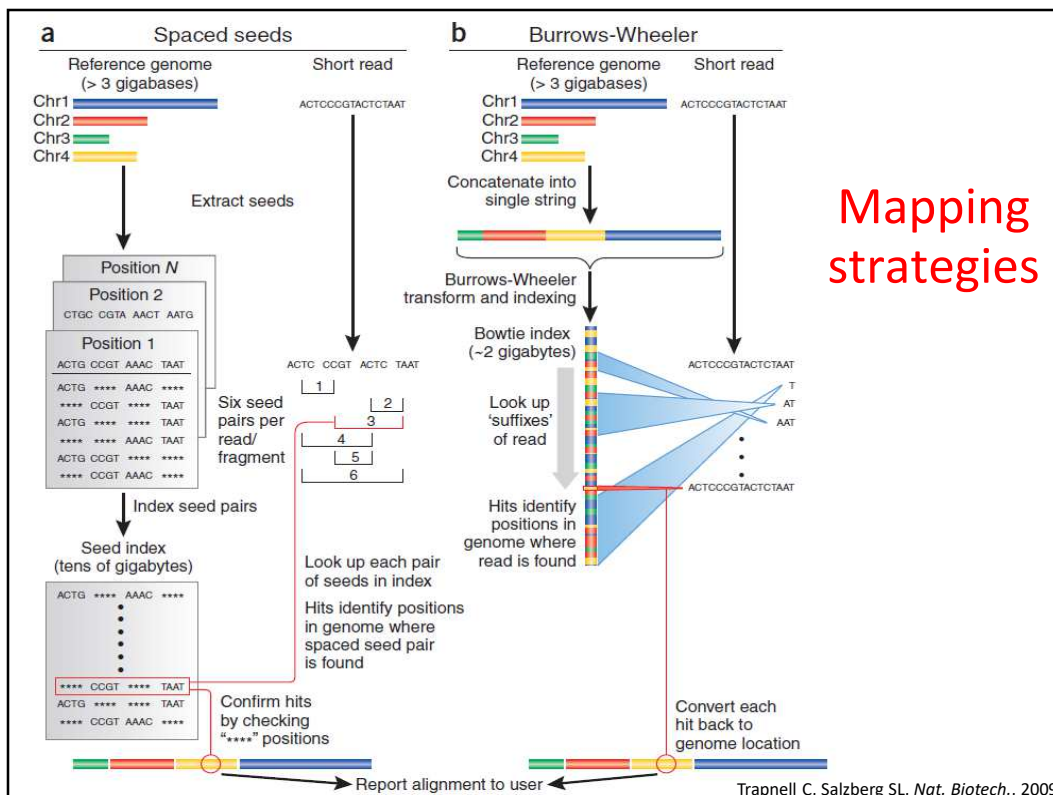
Mapping reads to assembled genomes

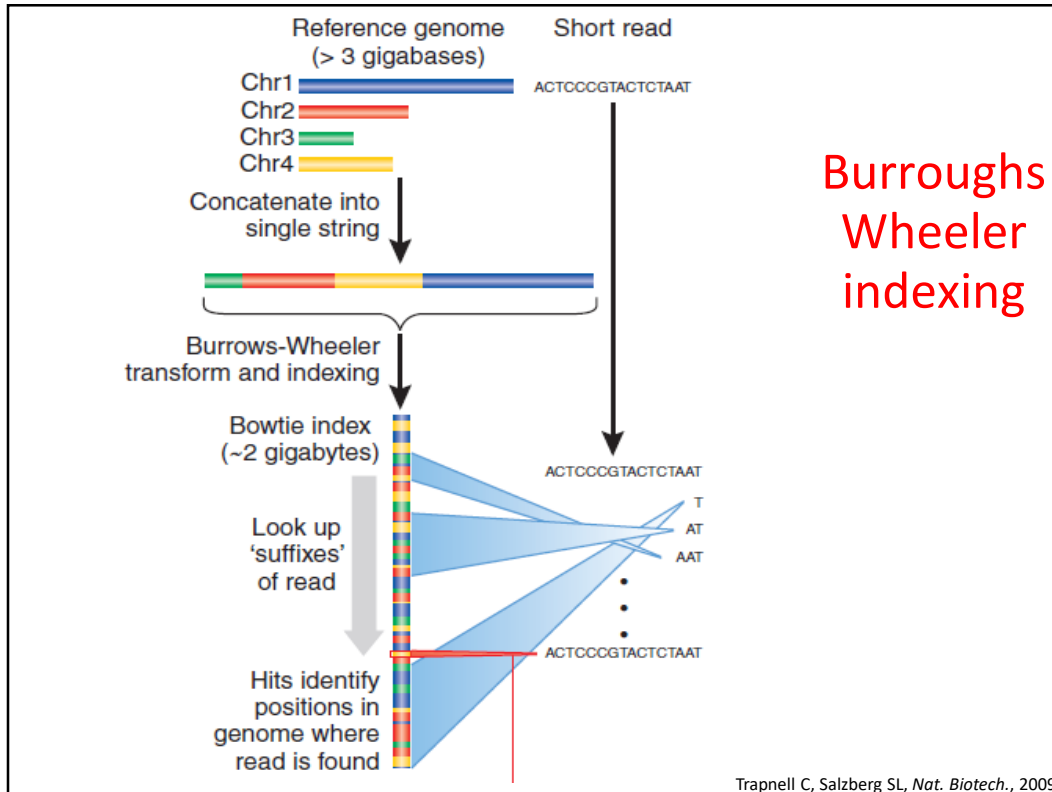
Table 1 A selection of short-read analysis software

Program	Website	Open source?	Handles ABI color space?	Maximum read length
Bowtie	http://bowtie.cbcb.umd.edu	Yes	No	None
BWA	http://maq.sourceforge.net/bwa-man.shtml	Yes	Yes	None
Maq	http://maq.sourceforge.net	Yes	Yes	127
Mosaik	http://bioinformatics.bc.edu/marthlab/Mosaik	No	Yes	None
Novoalign	http://www.novocraft.com	No	No	None
SOAP2	http://soap.genomics.org.cn	No	No	60
ZOOM	http://www.bioinform.com	No	Yes	240

The list is a little longer now! e.g. see https://en.wikipedia.org/wiki/List_of_sequence_alignment_software#Short-Read_Sequence_Alignment

Trapnell C, Salzberg SL, *Nat. Biotech.*, 2009

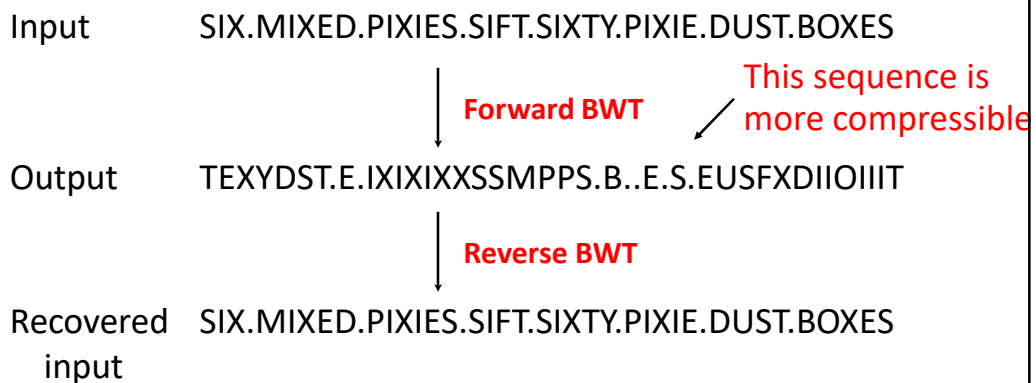




Burroughs-Wheeler transform indexing

BWT is often used for file compression (like bzip2), here used to make a fast 'lookup' index in a genome

BWT = 'reversible block-sorting'



http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Input

^BANANA |

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

**All
Rotations**

^BANANA |
| ^BANANA
A | ^BANAN
NA | ^BANA
ANA | ^BAN
NANA | ^BA
ANANA | ^B
BANANA | ^

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Sorting All Rows in Alphabetical Order

```
ANANA | ^B
ANA | ^BAN
A | ^BANAN
BANANA | ^
NANA | ^BA
NA | ^BANA
^BANANA |
| ^BANANA
```

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

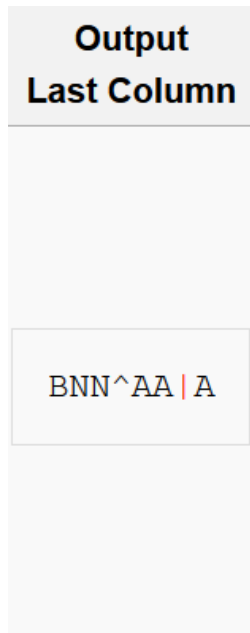
Burroughs-Wheeler transform indexing

Taking Last Column

```
ANANA | ^B
ANA | ^BAN
A | ^BANAN
BANANA | ^
NANA | ^BA
NA | ^BANA
^BANANA |
| ^BANANA
```

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing



http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Transformation				
Input	All Rotations	Sorting All Rows in Alphabetical Order	Taking Last Column	Output Last Column
^BANANA	^BANANA ^BANANA A ^BANAN NA ^BANA ANA ^BAN NANA ^BA ANANA ^B BANANA ^	ANANA ^B ANA ^BAN A ^BANAN BANANA ^ NANA ^BA NA ^BANA ^BANANA ^BANANA	ANANA ^ B ANA ^ BAN A ^ BANAN BANANA ^ NANA ^ BA NA ^ BANA ^BANANA ^ BANANA	BNN^AA A

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

BWT is remarkable because it is
reversible.

Any ideas as how you might reverse it?

Burroughs-Wheeler transform indexing

Input

BNN^AA | A

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Add 1	Sort 1	Add 2	Sort 2
B	A	BA	AN
N	A	NA	AN
N	A	NA	A
^	B	^B	BA
A	N	AN	NA
A	N	AN	NA
	^	^	^B
A		A	^
Write the sequence as the last column	Sort it...	Add the columns...	Sort those...

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Add 3	Sort 3	Add 4	Sort 4
BAN	ANA	BANA	ANAN
NAN	ANA	NANA	ANA
NA	A ^	NA ^	A ^B
^BA	BAN	^BAN	BANA
ANA	NAN	ANAN	NANA
ANA	NA	ANA	NA ^
^B	^BA	^BA	^BAN
A ^	^B	A ^B	^BA
Add the columns...	Sort those...	Add the columns...	Sort those...

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Add 5	Sort 5	Add 6	Sort 6
BANAN NANA NA ^B ^BANA ANANA ANA ^ ^BAN A ^BA	ANANA ANA ^ A ^BA BANAN NANA NA ^B ^BANA ^BAN	BANANA NANA ^ NA ^BA ^BANAN ANANA ANA ^B ^BANA A ^BAN	ANANA ANA ^B A ^BAN BANANA NANA ^ NA ^BA ^BANAN ^BANA
Add the columns...	Sort those...	Add the columns...	Sort those...

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Add 7	Sort 7	Add 8
BANANA NANA ^B NA ^BAN ^BANANA ANANA ^ ANA ^BA ^BANAN A ^BANA	ANANA ^ ANA ^BA A ^BANA BANANA NANA ^B NA ^BAN ^BANANA ^BANAN	BANANA ^ NANA ^BA NA ^BANA <u>^BANANA </u> ANANA ^B ANA ^BAN ^BANANA A ^BANAN
Add the columns...	Sort those...	Add the columns...

The row with the "end of file" character at the end is the original text

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

Burroughs-Wheeler transform indexing

Output
^BANANA

The row with the "end of file" character at the end is the original text

http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

The Burroughs-Wheeler transform leads naturally to a suffix array

0	googol\$	String Sorting	0	6	\$googol
1	oogol\$g	→	1	3	gol\$go o
2	ogol\$go		2	0	googol \$
3	gol\$goo		3	5	l\$goog o
4	ol\$goog		4	2	ogol\$g o
5	l\$googo		5	4	ol\$goo g
6	\$googol		6	1	oogol\$ g

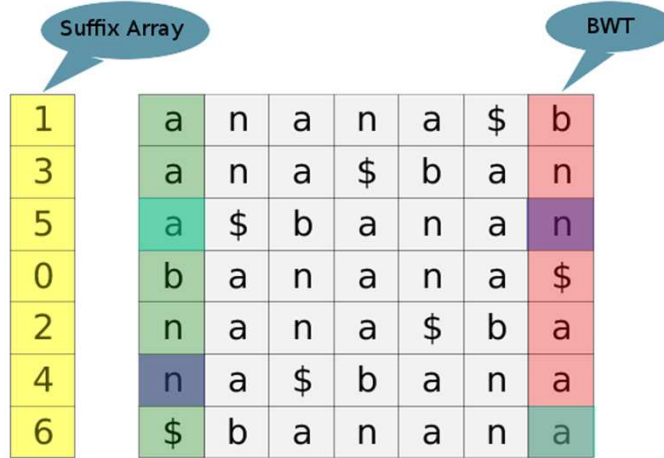
Pos

X = googol\$

i	S(i)	B[i]
↓	↓	↓
		lo\$ogg
		(6,3,0,5,2,4,1)

Li & Durbin, doi:10.1093/bioinformatics/btp324/

The Burroughs-Wheeler transform leads naturally to a suffix array

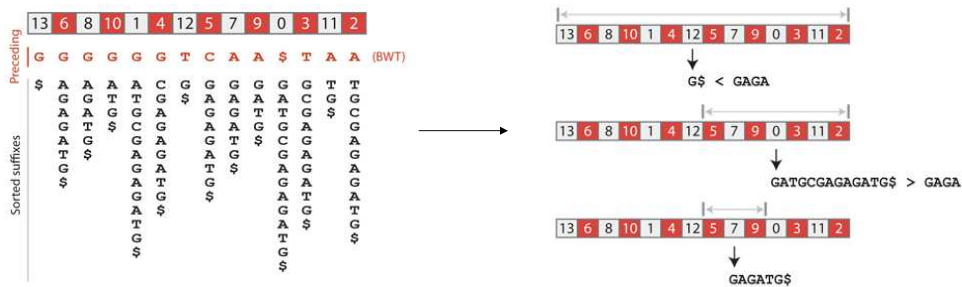


<http://blog.avadis-ngs.com/2012/04/elegant-exact-string-match-using-bwt-2/> (& wikipedia)

“If string W is a substring of X, the position of each occurrence of W in X will occur in an interval in the suffix array. This is because all the suffixes that have W as prefix are sorted together.”

Li & Durbin, doi:10.1093bioinformatics/btp324/

e.g. applying BWT to construct the suffix array of **GATGCGAGAGATG**



The search can be even more efficient by using compression & various other extensions

<http://blog.thegrandlocus.com/2016/07/a-tutorial-on-burrows-wheeler-indexing-methods>

Why is this efficient?

Searching a suffix array in this way cuts the search space in half at each step, so...

A suffix array of the human genome (3.2 billion bases) takes at most

$$\log_2(3.2 \text{ billion}) + 1 = 32 \text{ steps}$$

to determine if a query sequence is present or not

There are few more steps to find all the occurrences, build an efficient real-world implementation, use compression to reduce memory and storage space, etc., but this still illustrates the massive savings in time and memory from constructing an index

