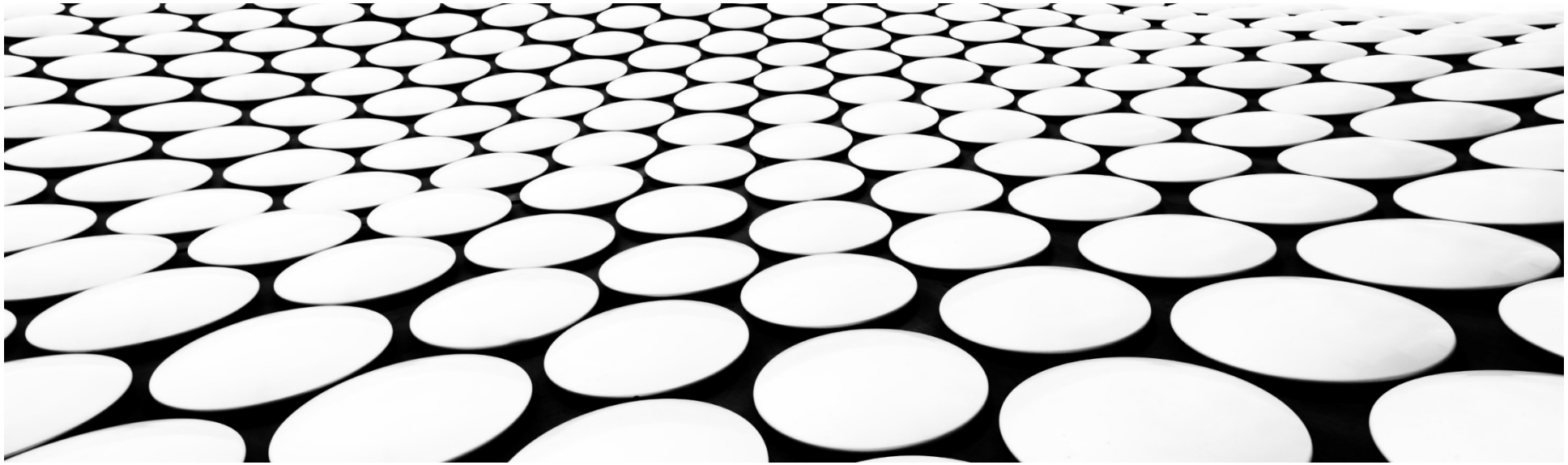
BIOINFORMATICS(BIOCOMPUTING)
(6)
Assembly

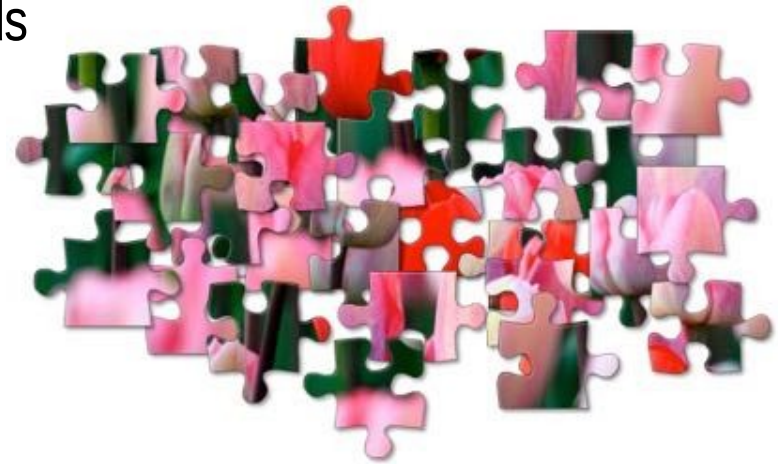
DR. IBRAHIM ZAGHLOUL



Assembly & Shortest Common Superstring

Assembly

Reads



+

Refere enom

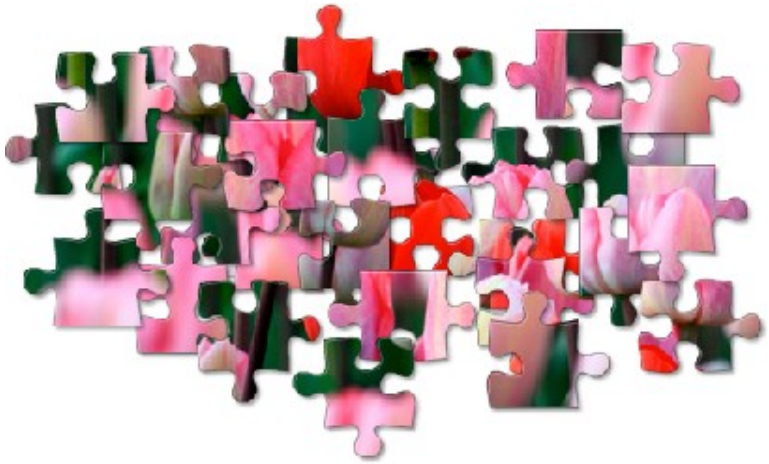
How do we assemble puzzle without the benefit of knowing what the finished product should look like?

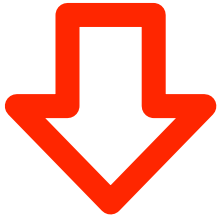
Input DNA

(That's what the Human Genome Project had to do!)

De novo shotgun assembly

De Novo: From scratch





Assembly

```
                                  CTAGGCCCTCAATTTTT
                                CTCTAGGCCCTCAATTTTT
                              GGCTCTAGGCCCTCATTTTTT
                          CTCGGCTCTAGCCCCTCATTTT
                      TATCTCGACTCTAGGCCCTCA
Reconstruct this      TATCTCGACTCTAGGCC
                  TCTATATCTCGGCTCTAGG
              GGCGTCTATATCTCG
              GGCGTCGATATCT
              GGCGTCTATATCT
              GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

From these

Assembly

Reconstruct this

CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

??????????????????????????????????????

From these

Coverage

The amount of redundant information that we have about the genome

```
                CTAGGCCCTCAATTTTT
               CTCTAGGCCCTCAATTTTT
             GGCTCTAGGCCCTCATTTTTT
         CTCGGCTCTAGCCCCTCATTTT
     TATCTCGACTCTAGGCCCTCA
     TATCTCGACTCTAGGCC
  TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Coverage = 5 (This position is covered by 5 reads)

Coverage

```
                    CTAGGCCCTCAATTTTT
                  CTCTAGGCCCTCAATTTTT
                GGCTCTAGGCCCTCATTTTTT
            CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Coverage = 5

Overall coverage: the coverage averaged over all positions of the genome

$$\frac{\text{Total Reads len.}}{\text{Total genome len.}}$$

CTAGGCCCTCAATTTTT
CTCTAGGCCCTCAATTTTT
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

177 bases

35 bases

Average coverage = 177 / 35 ≈ 5-fold

Given these 2 reads from the same genome

TCTATATCTCGGCTCTAGG

TATCTCGACTCTAGGCC

```
TCTATATCTCGGCTCTAGG
       ||||||| ||||||||
       TATCTCGACTCTAGGCC
```

- A suffix of one read is very similar to a prefix in the other read.
- This gives a hint that these reads might have originated form overlapping portions from the genome.

First law of assembly

If a suffix of read A is similar to a prefix of read B...

```
TCTATATCTCGGCTCTAGG
      ||||||| |||||||
      TATCTCGACTCTAGGCC
```

...then A and B might *overlap* in the genome

```
      TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
            TATCTCGACTCTAGGCC
```

```
TCTATATCTCGGCTCTAGG
      ||||||| ||||||||
     TATCTCGACTCTAGGCC
```

Why the differences?

1. Sequencing errors
2. Polyploidy: e.g. humans have 2 copies of each chromosome, and copies can differ

Second law of assembly

More coverage leads to more and longer overlaps

```
                    CTAGGCCCTCAATTTTT
          CTCGGCTCTAGCCCCTCATTTT
     TCTATATCTCGGCTCTAGG
GGCGTCGATATCT
```

less coverage

```
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
                    CTAGGCCCTCAATTTTT
               GGCTCTAGGCCCTCATTTTTT
          CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
     TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCTATATCT
```

more coverage

Second law of assembly

More coverage leads to more and longer overlaps

```
                    CTAGGCCCTCAATTTTT
            CTCGGCTCTAGCCCCTCATTTT
      TCTATATCTCGGCTCTAGG
GGCGTCGATATCT
```

less coverage

```
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
                    CTAGGCCCTCAATTTTT
                GGCTCTAGGCCCTCATTTTTT
            CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
      TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCTATATCT
```

more coverage

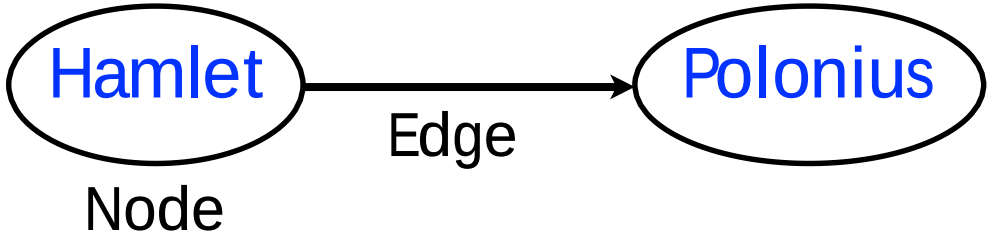
```
TCTATATCTCGGCTCTAGG
      ||||||| |||||||
     TATCTCGACTCTAGGCC
```

Representing all overlaps in one structure

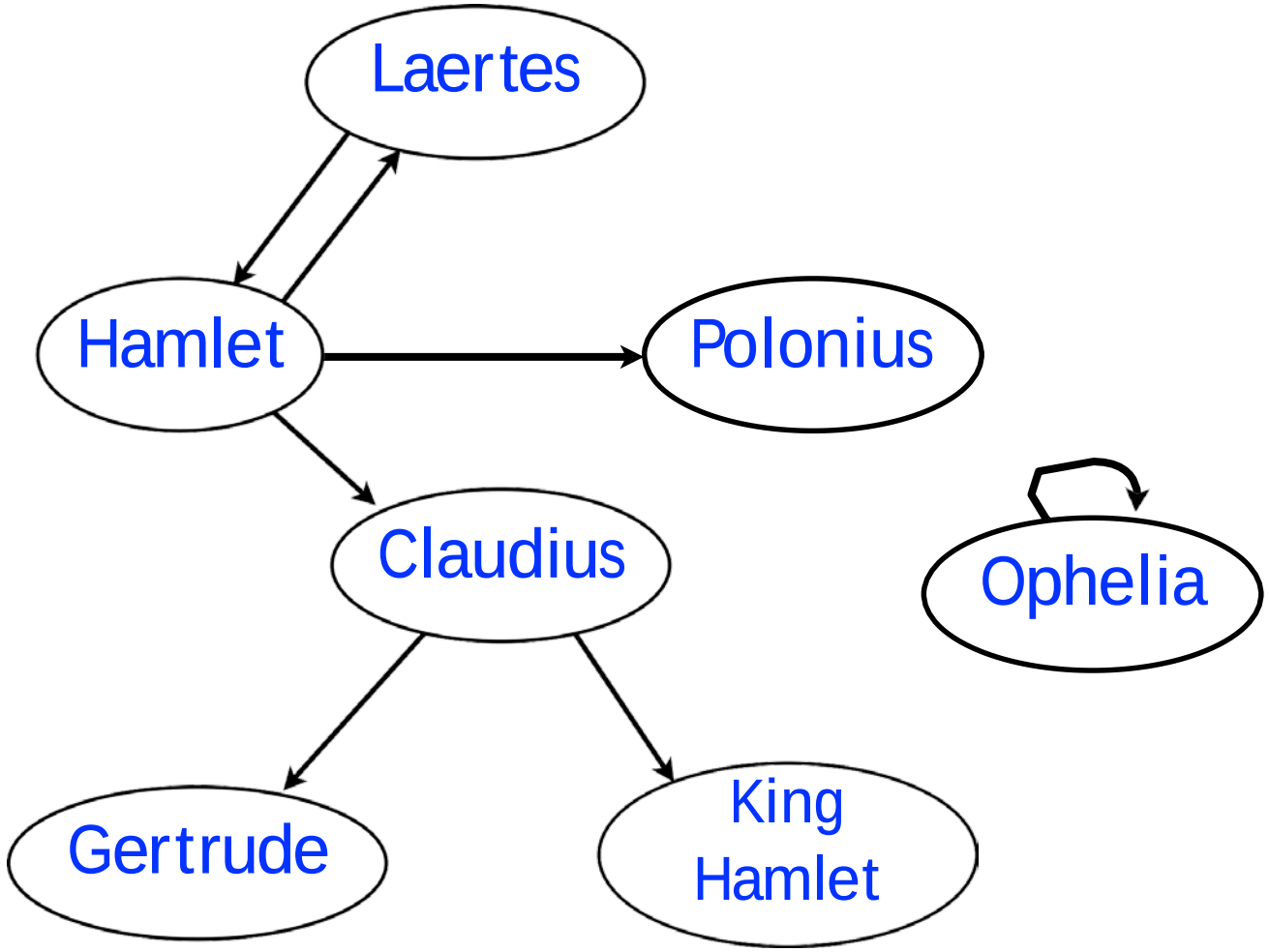
```
TCTATATCTCGGCTCTAGG
    ||||||| |||||||
   TATCTCGACTCTAGGCC
```

```
TATCTCGACTCTAGGCC
   |||| |||||| ||
  CTCGGCTCTAGCCCCTCAT
```

Directed graph



Directed graph

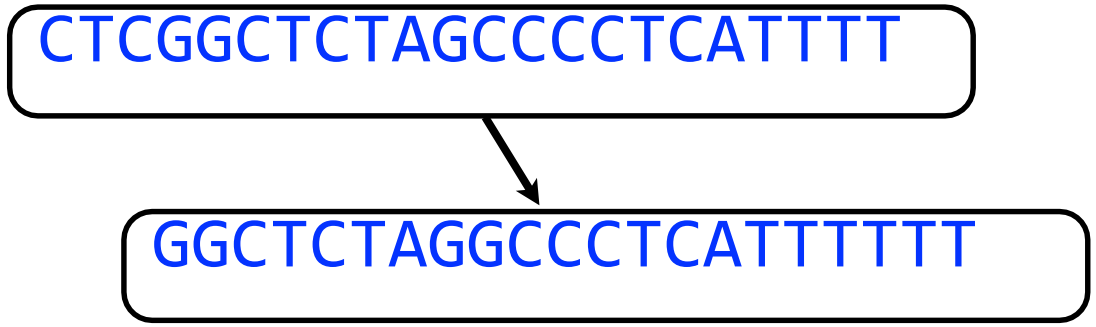


Overlap graph

Each node is a read

CTCGGCTCTAGCCCCTCATTTT

Draw edge A -> B when suffix of A overlaps prefix of B



Overlap graph

- Nodes: all 6-mers from GTACGTACGAT
  - GTACGT
  - TACGTA
  - ACGTAC
  - CGTACG
  - GTACGA
  - TACGAT

Decide a threshold for overlap to consider.

- Edges: overlaps of length ≥ 4

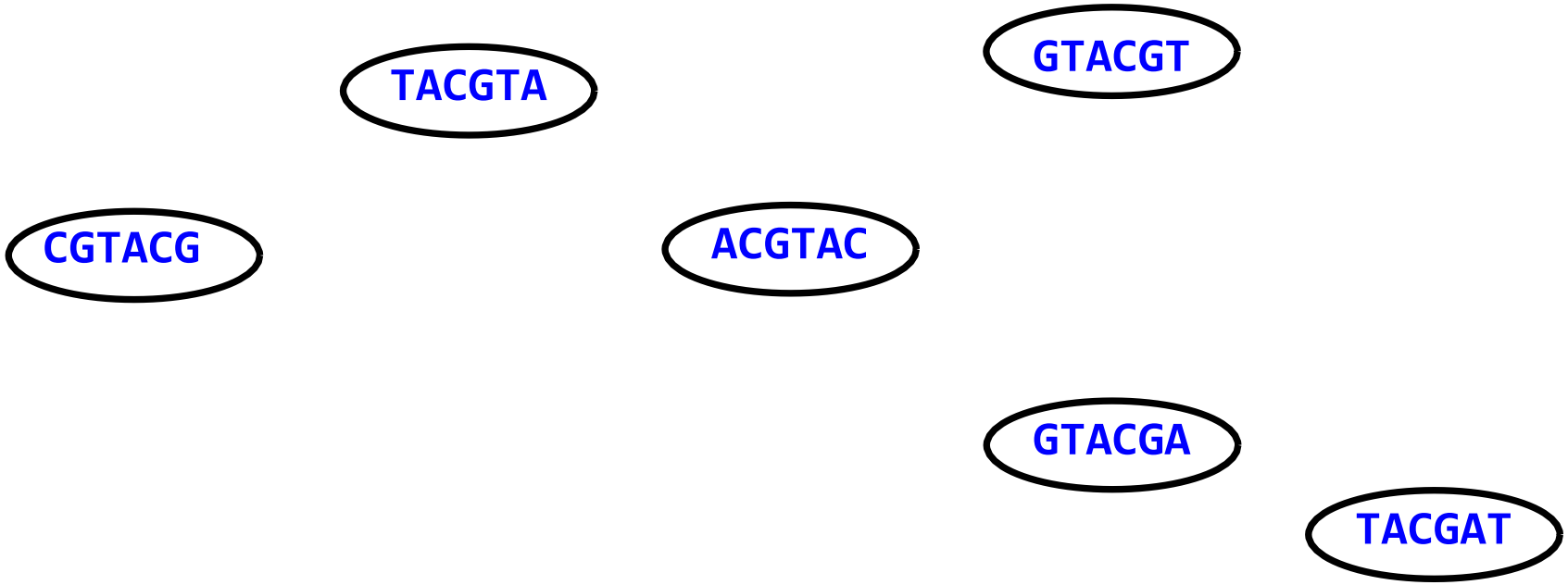
Overlap graph

- Nodes: all 6-mers from GTACGTACGAT

Decide a threshold for overlap to consider.

- Edges: overlaps of length ≥ 4

- GTACGT
- TACGTA
- ACGTAC
- CGTACG
- GTACGA
- TACGAT



Note: Only exact matches are allowed in this example

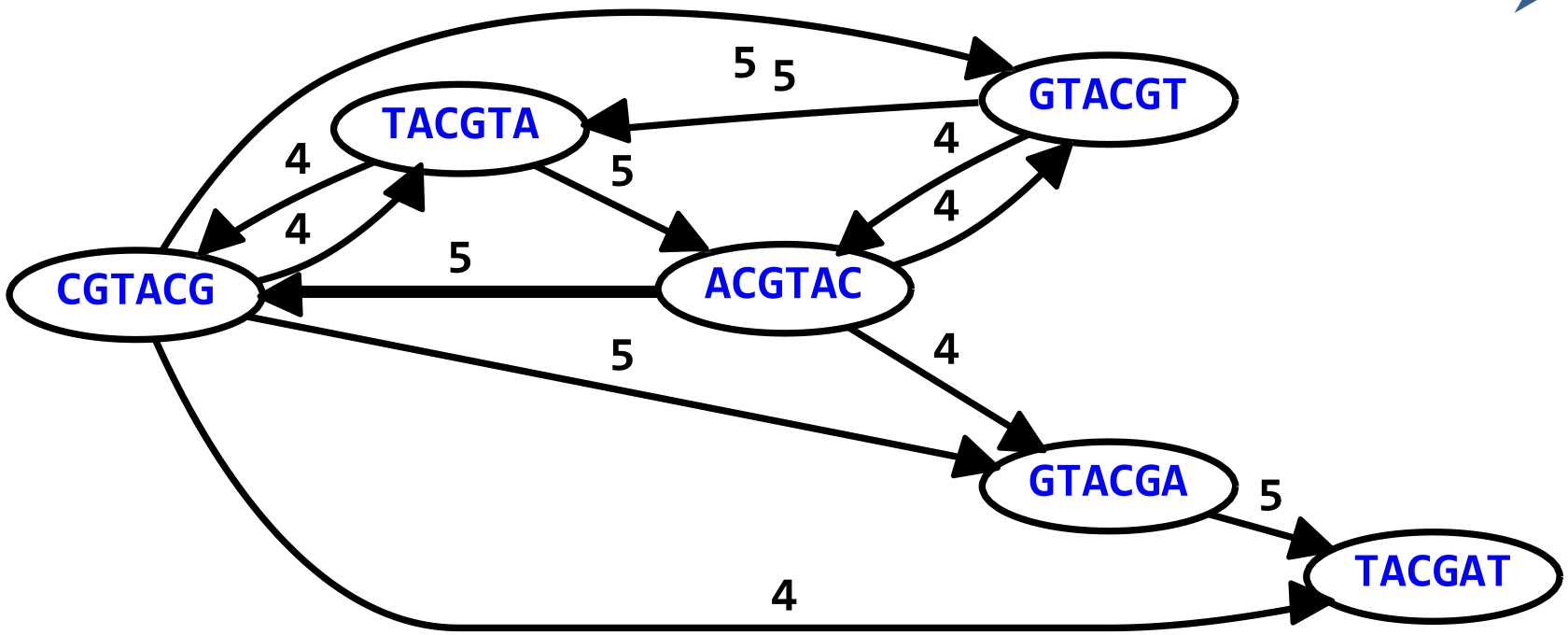
Overlap graph

- Nodes: all 6-mers from GTACGTACGAT

Decide a threshold for overlap to consider.

- Edges: overlaps of length ≥4

- GTACGT
- TACGTA
- ACGTAC
- CGTACG
- GTACGA
- TACGAT

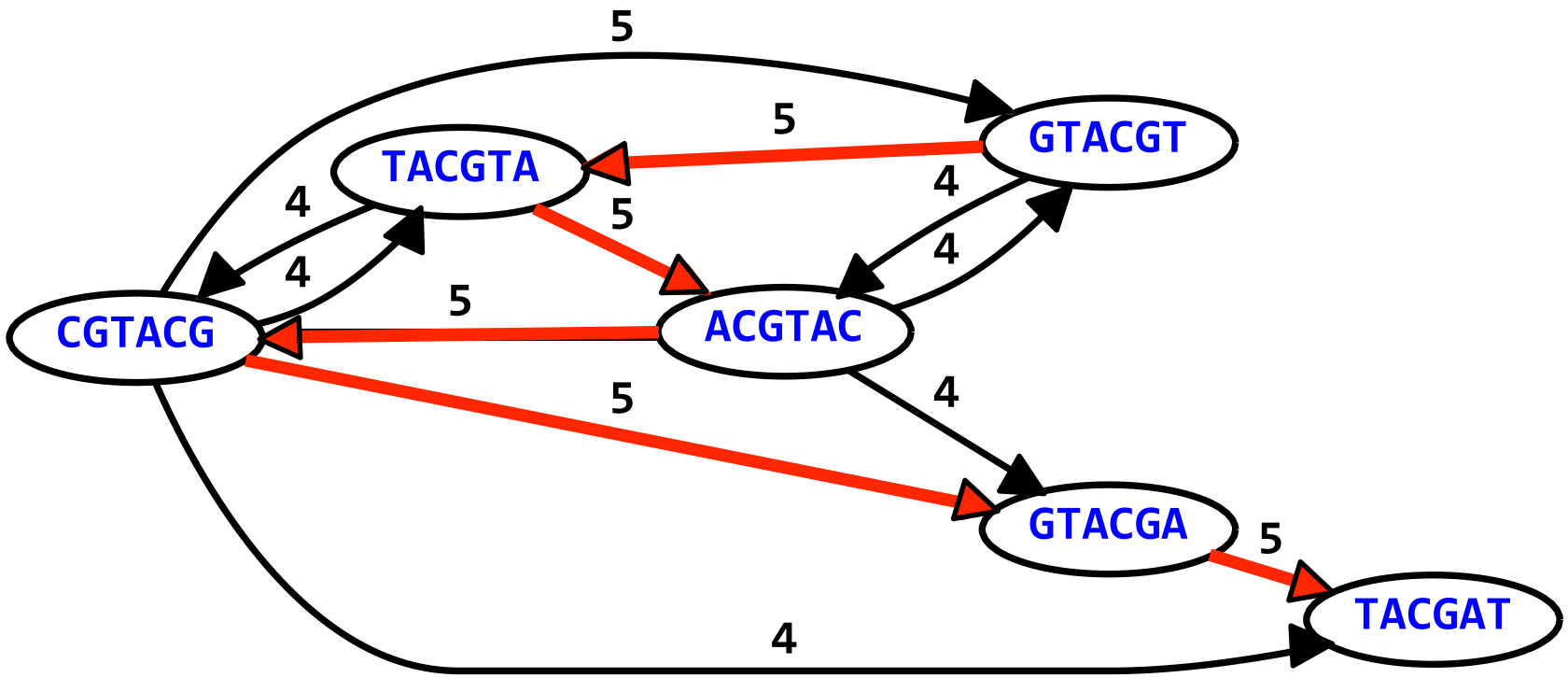


Note: Only exact matches are allowed in this example

Overlap graph

Nodes: all 6-mers from GTACGTACGAT

Edges: overlaps of length ≥ 4



Shortest common superstring

Given set of strings S, find $SCS(S)$: shortest string containing the strings in S as substrings

S: BAA AAB BBA ABA ABB BBB AAA BAB

$Concat(S)$: BAAAABBBAABAABBBBBAAABAB

24

$SCS(S)$: AAABBBABAA

10

Shortest common superstring

NP-complete: no efficient algorithms for large inputs

Idea: pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAA

Idea: pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAAB

Idea: pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABA

Idea: pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABB

Idea: pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB ⟵ superstring 1

Idea: pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB ⟵ superstring 1

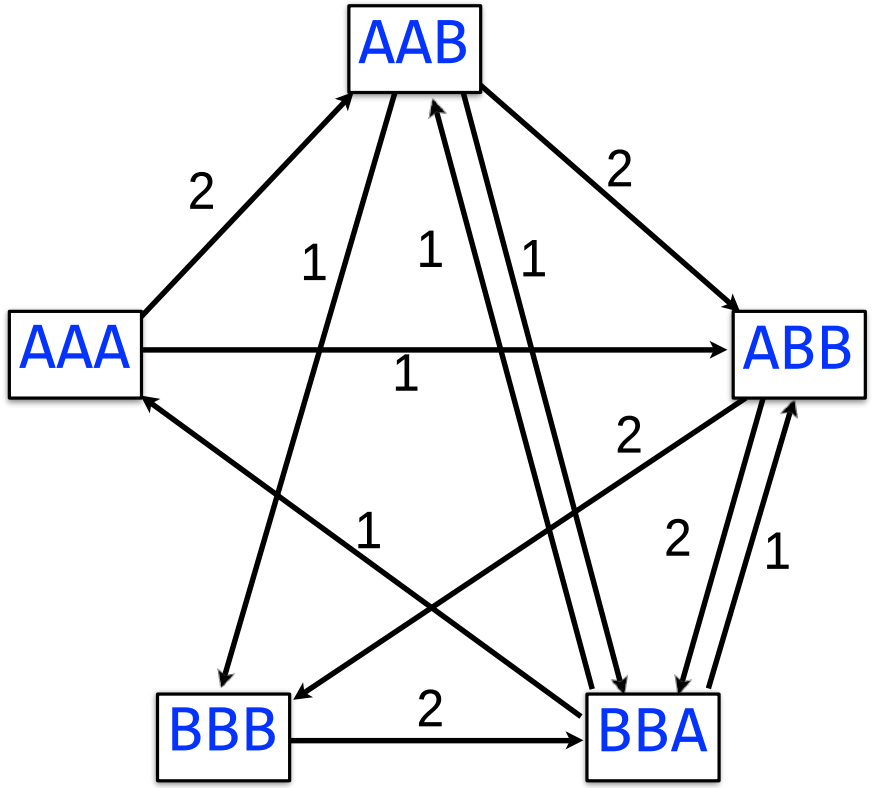
order 2: AAA AAB ABA BAB ABB BBB BAA BBA

AAABABBBAABBA ⟵ superstring 2

Try all possible orderings and pick shortest superstring

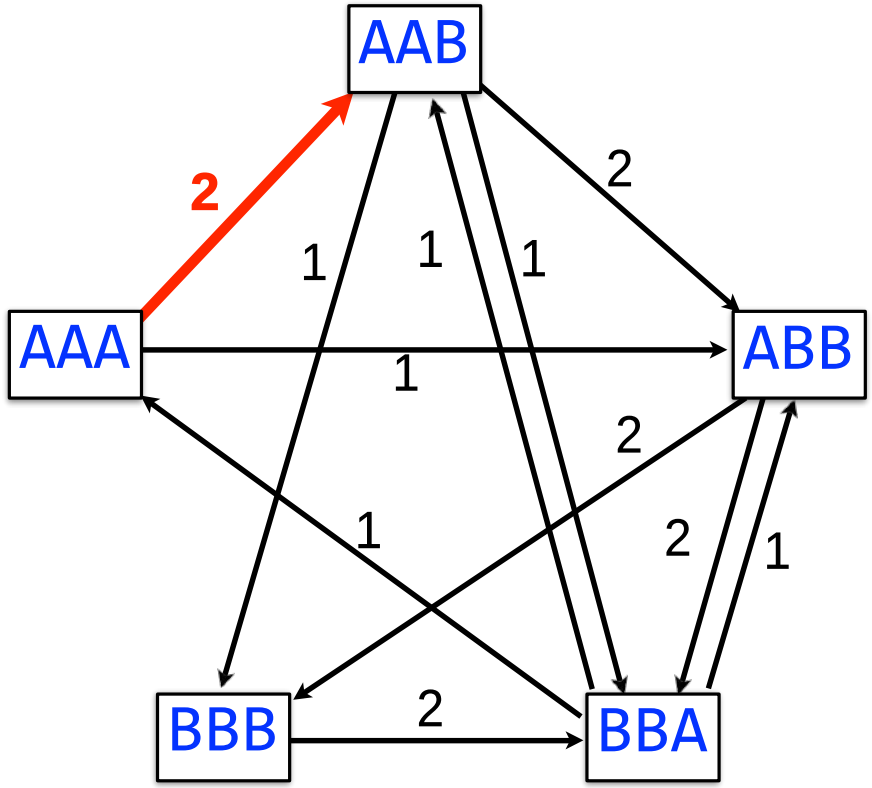
If S contains n strings, $n!$ (n factorial) orderings possible

Greedy shortest common superstring

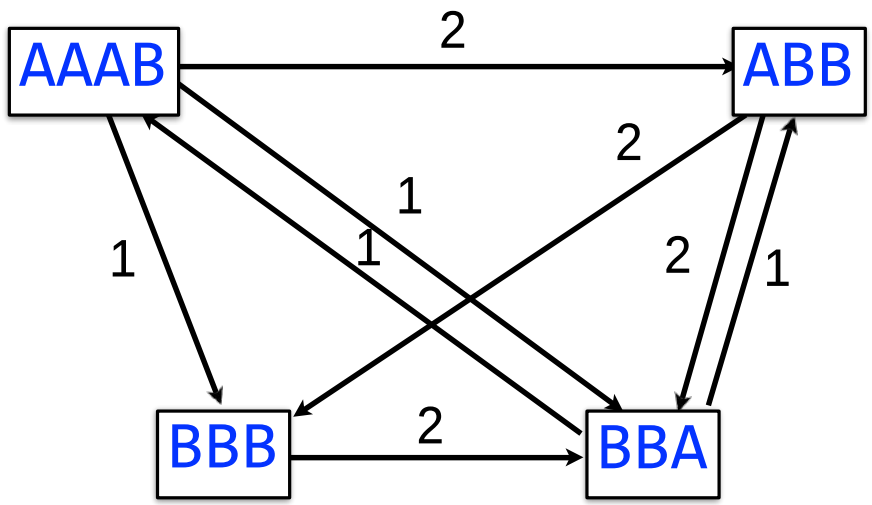


Greedy: It takes decisions that give the most reduction of the superstring length

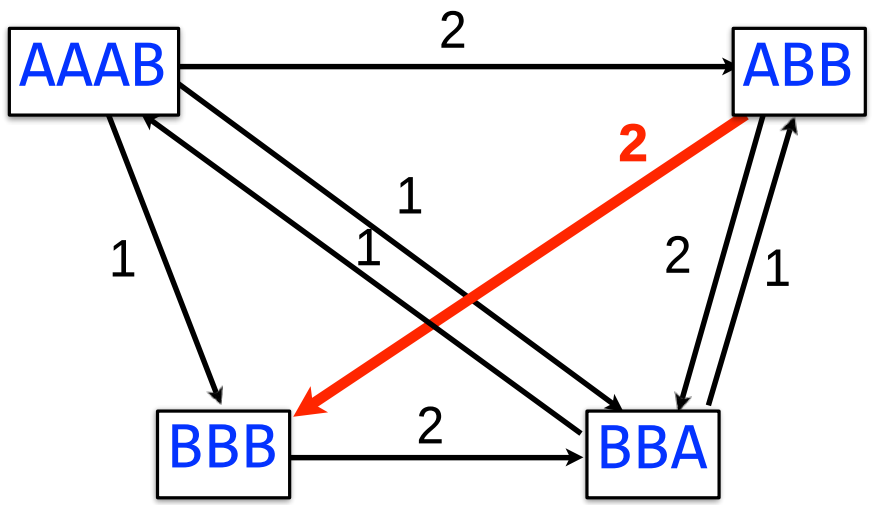
Greedy shortest common superstring



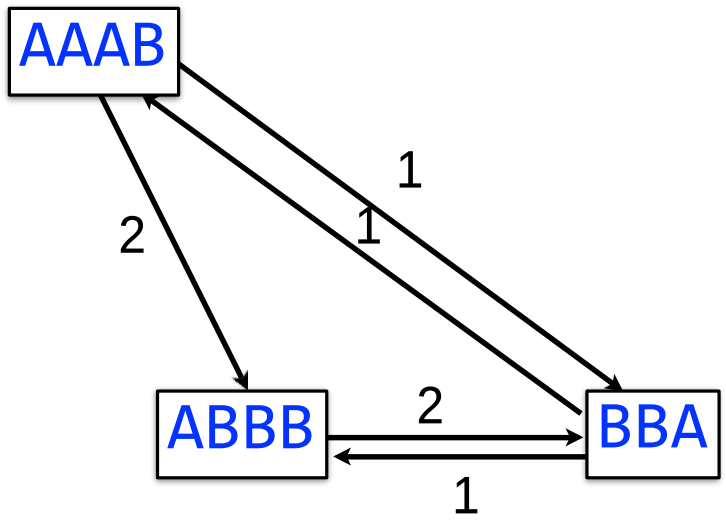
Greedy shortest common superstring



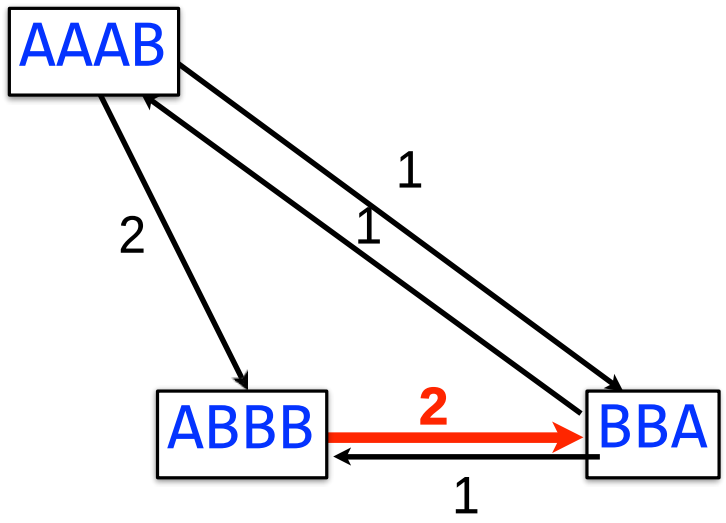
Greedy shortest common superstring



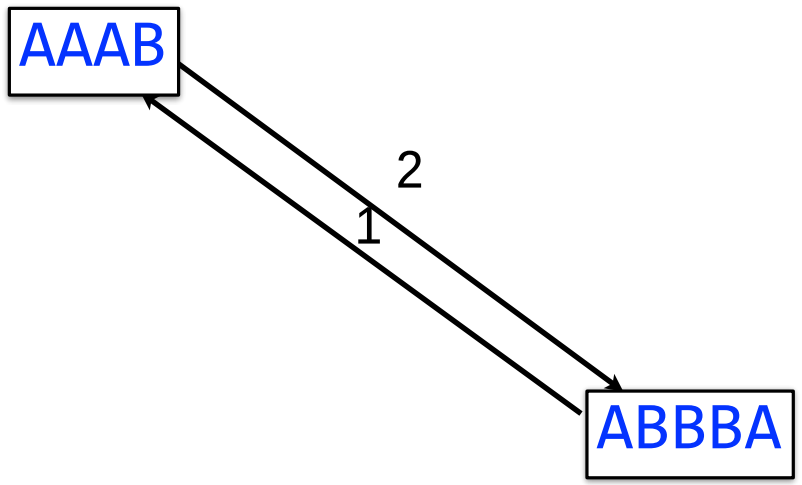
Greedy shortest common superstring



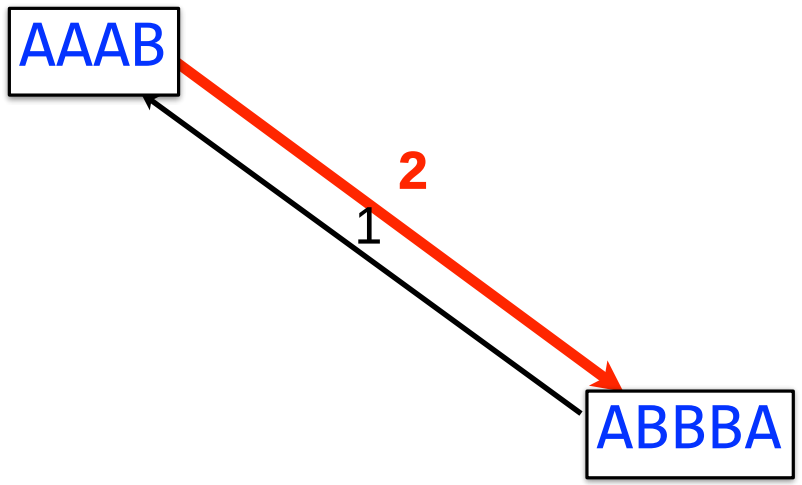
Greedy shortest common superstring



Greedy shortest common superstring



Greedy shortest common superstring



Greedy shortest common superstring

AAABBBA ← superstring, length=7

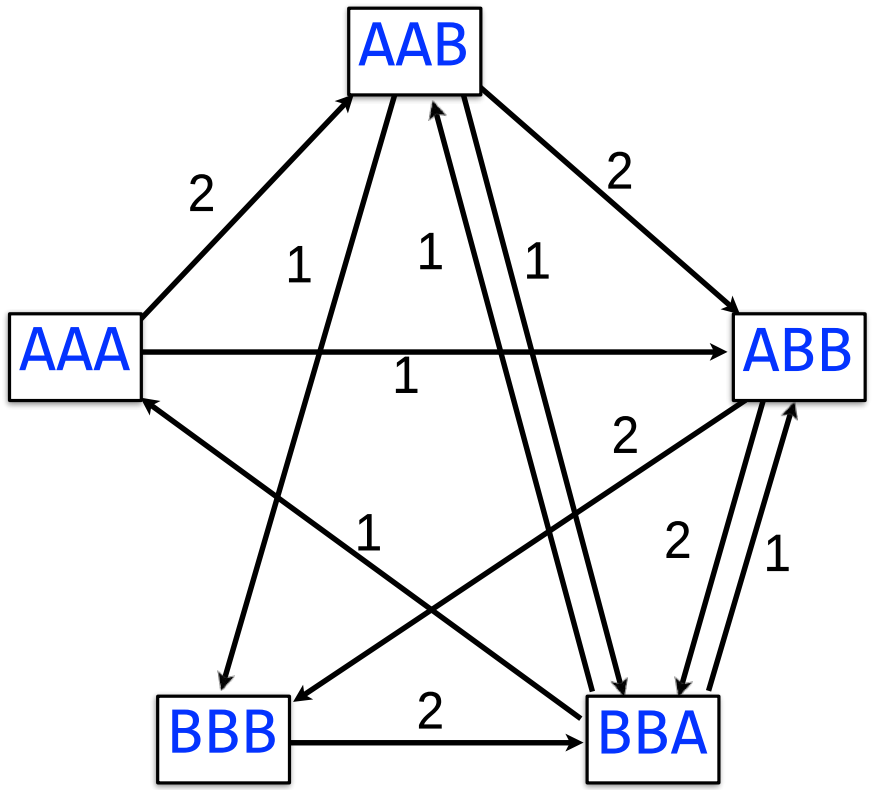
Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action (l = 1):

Input strings

AAA AAB ABB BBB BBA



Shortest common superstring: greedy

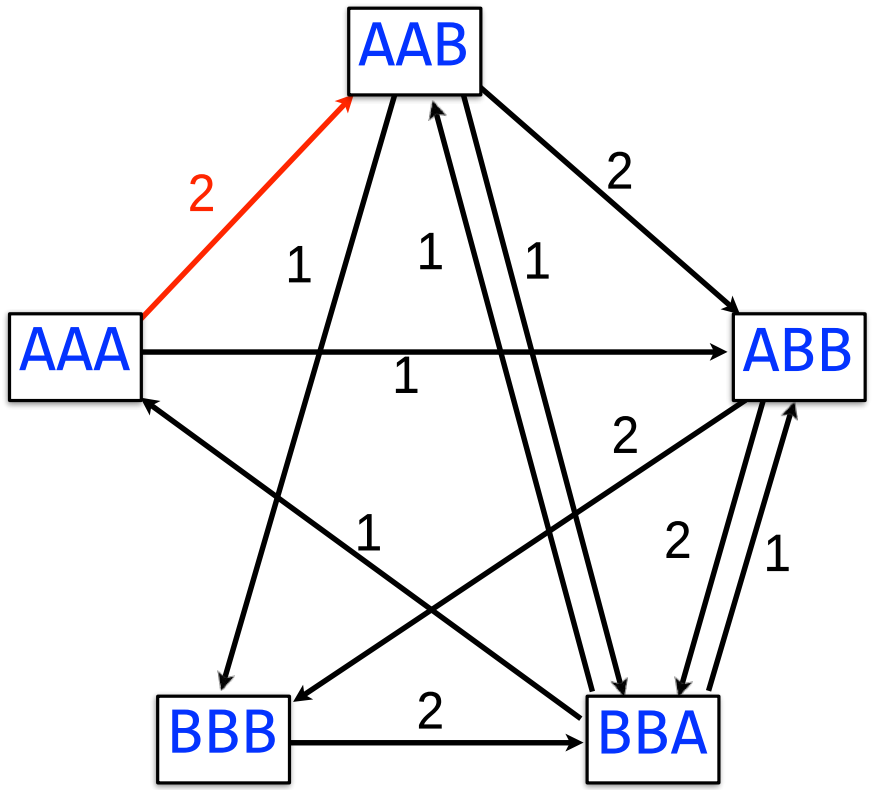
Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action (l = 1):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

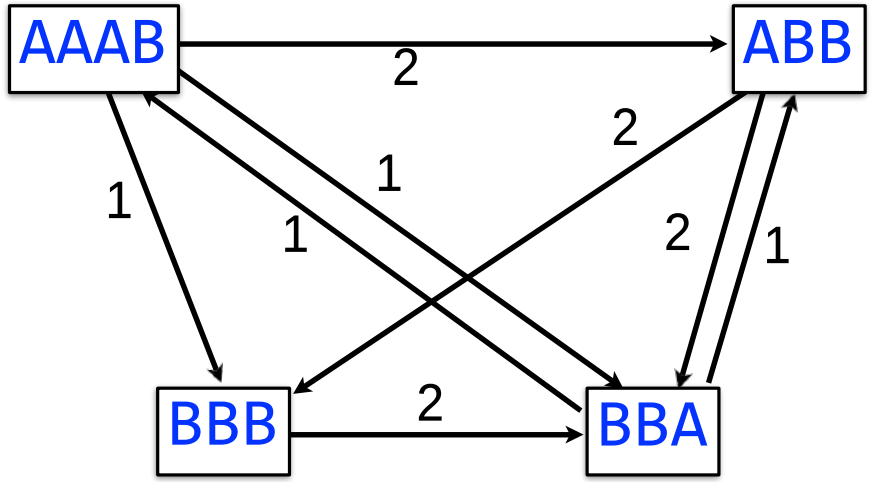
Algorithm in action (l = 1):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

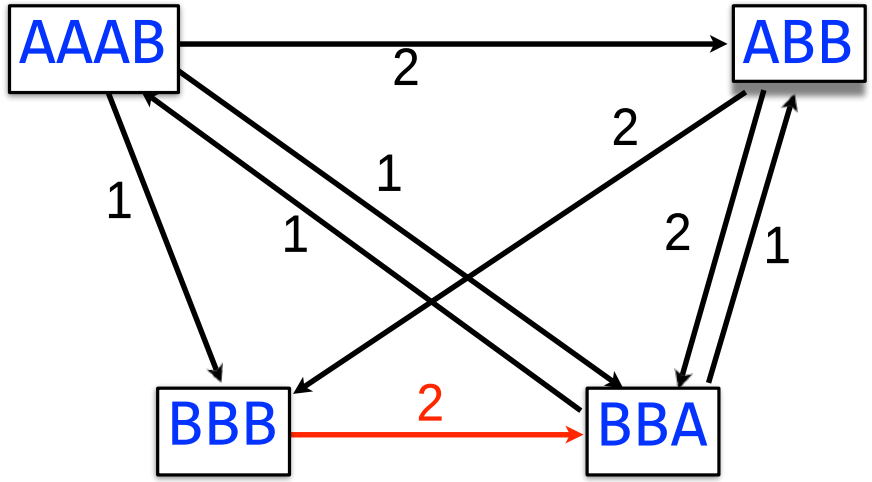
Algorithm in action (l = 1):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action (l = 1):

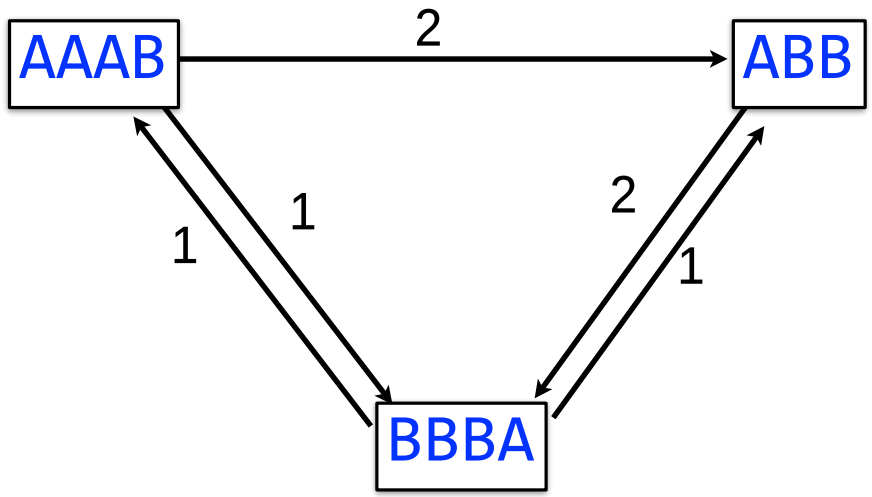
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action (l = 1):

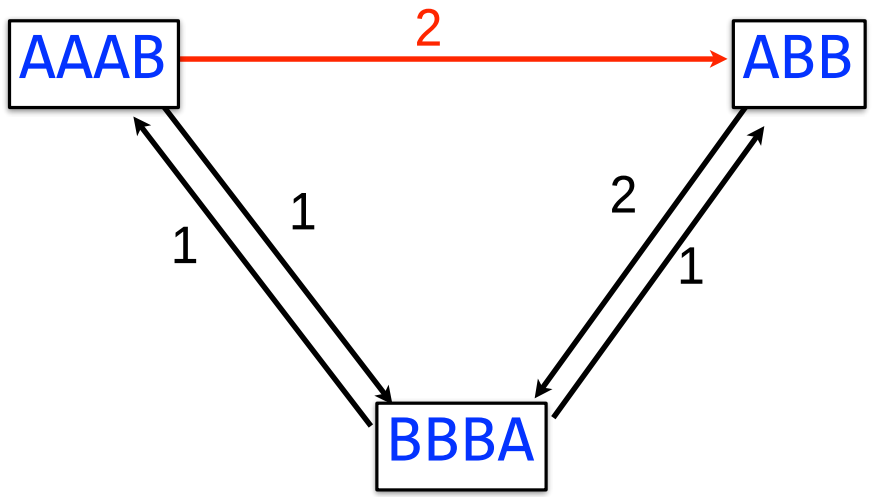
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action (l = 1):

Input strings

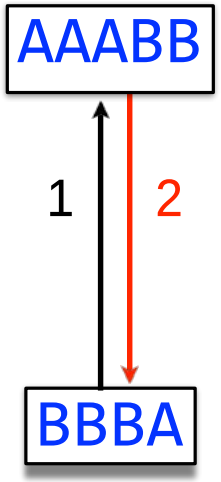
AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

AAABB BBBA



Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

Algorithm in action (l = 1):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

AAABB BBBA

AAABBBA

$\boxed{\text{AAABBBA}}$

That's the SCS

Greedy shortest common superstring

AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

Greedy shortest common superstring

AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

AAAB ABBA BBB

Greedy shortest common superstring

AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

AAAB ABBA BBB

AAABBA BBB

Greedy shortest common superstring

AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

AAAB ABBA BBB

AAABBA BBB

AAABBABBB ← superstring, length=9

Greedy shortest common superstring

AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

AAAB ABBA BBB

AAABBA BBB

AAABBABBB ⟵ superstring, length=9

AAABBBA ⟵ superstring, length=7

- Greedy answer *isn't necessarily optimal*
- *It doesn't necessarily return the SCS*

Different kind of graph

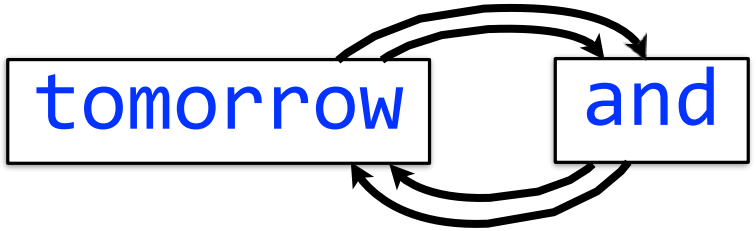
"tomorrow and tomorrow and tomorrow"

tomorrow

and

Different kind of graph

"tomorrow and tomorrow and tomorrow"



An edge represents an ordered pair of adjacent words in the input

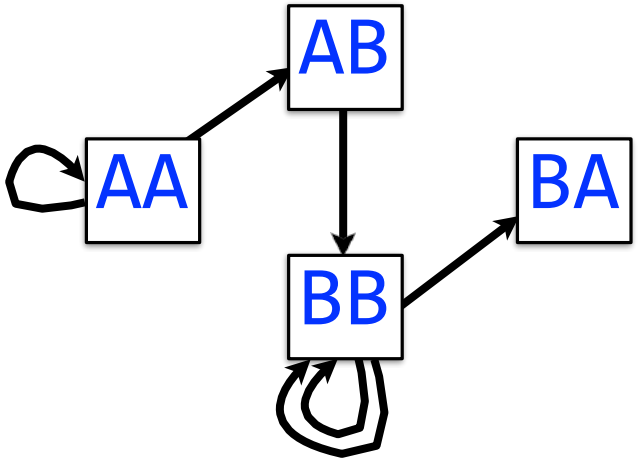
Multigraph: there can be more than one edge from node A to node B

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

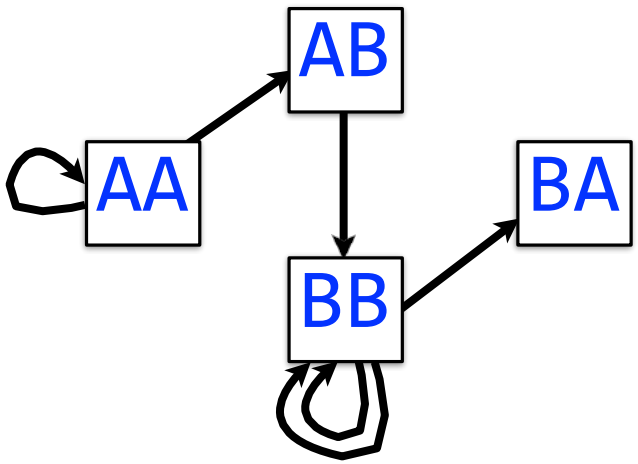
L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB BB, BA



One edge per *k*-mer

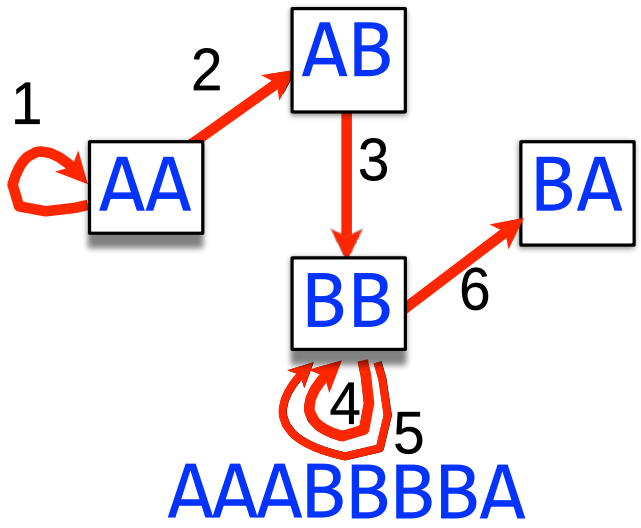
One node per distinct *k*-1-mer

De Bruijn graph



Walk crossing each edge exactly once gives a reconstruction of the genome

De Bruijn graph



Walk crossing each edge exactly once gives a reconstruction of the genome. This is an *Eulerian walk*.

Eulerian walk: A walk through the grapgh that groes from node to node and crosses each of the edges exactly once

De Bruijn graph

Aside: how do you pronounce "De Bruijn"?

There is debate:

https://www.biostars.org/p/7186/

I still don't quite know. I say "De Broin" (rhymes with "groin")

I asked a Dutch person once; his pronunciation sounded more like "De Brown"

Nicolaas Govert de Bruijn
1918 -- 2012

Directed multigraph

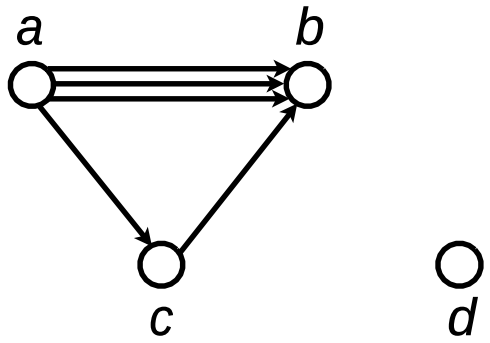
Directed **multigraph** $G(V, E)$ consists of set of *vertices, V* and **multiset** of *directed edges, E*

Otherwise, like a directed graph

Node's *indegree* = # incoming edges

Node's *outdegree* = # outgoing edges

De Bruijn graph is a directed multigraph



$V = \{a, b, c, d\}$

$E = \{(a, b), (a, b), (a, b), (a, c), (c, b)\}$

Repeated

Eulerian walk definitions and statements

Node is *balanced* if indegree equals outdegree

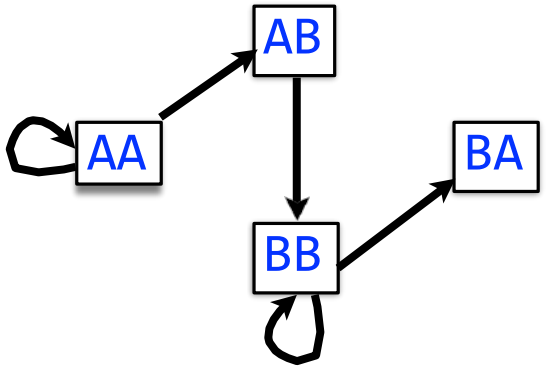
Node is *semi-balanced* if indegree differs from outdegree by 1

Graph is *connected* if each node can be reached by some other node

Eulerian walk visits each edge exactly once

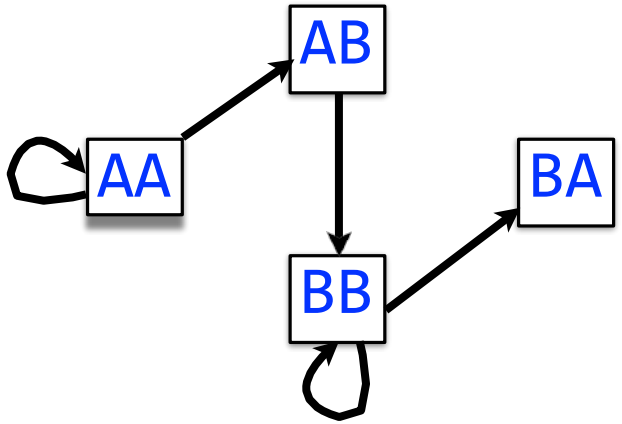
Not all graphs have Eulerian walks. Graphs that do are *Eulerian.*
(For simplicity, we won't distinguish Eulerian from semi-Eulerian.)

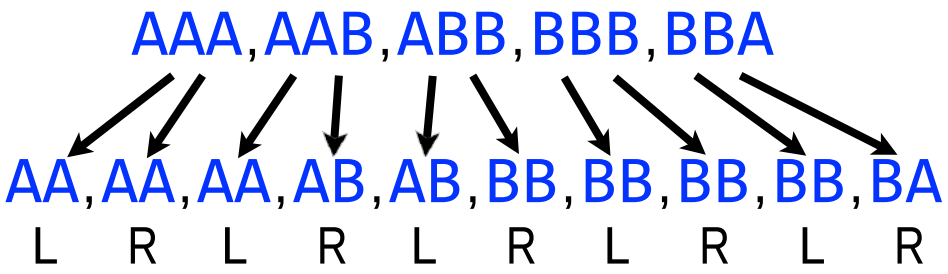
A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced



De Bruijn graph

Back to 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": each genome *k*-mer is sequenced exactly once with no errors

Pick a substring length *k*: 5

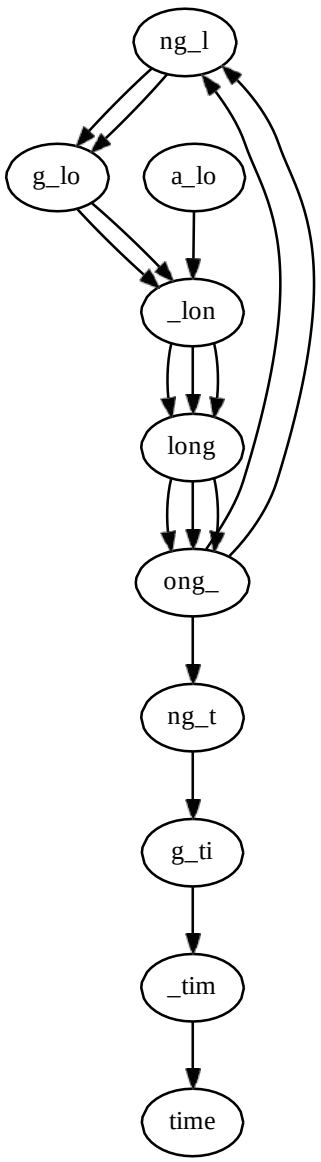
Start with an input string: `a_long_long_long_time`

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

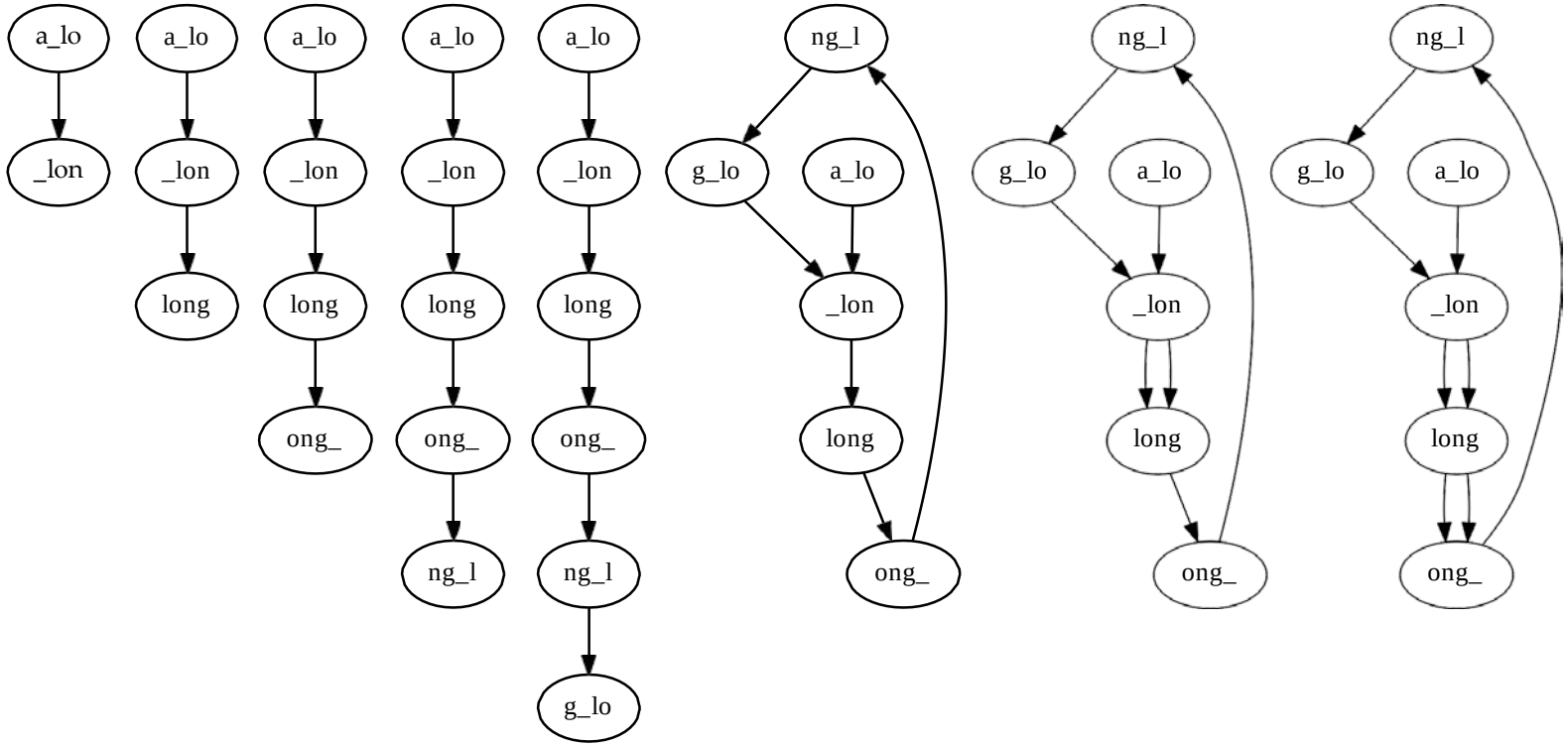
`long_`

`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



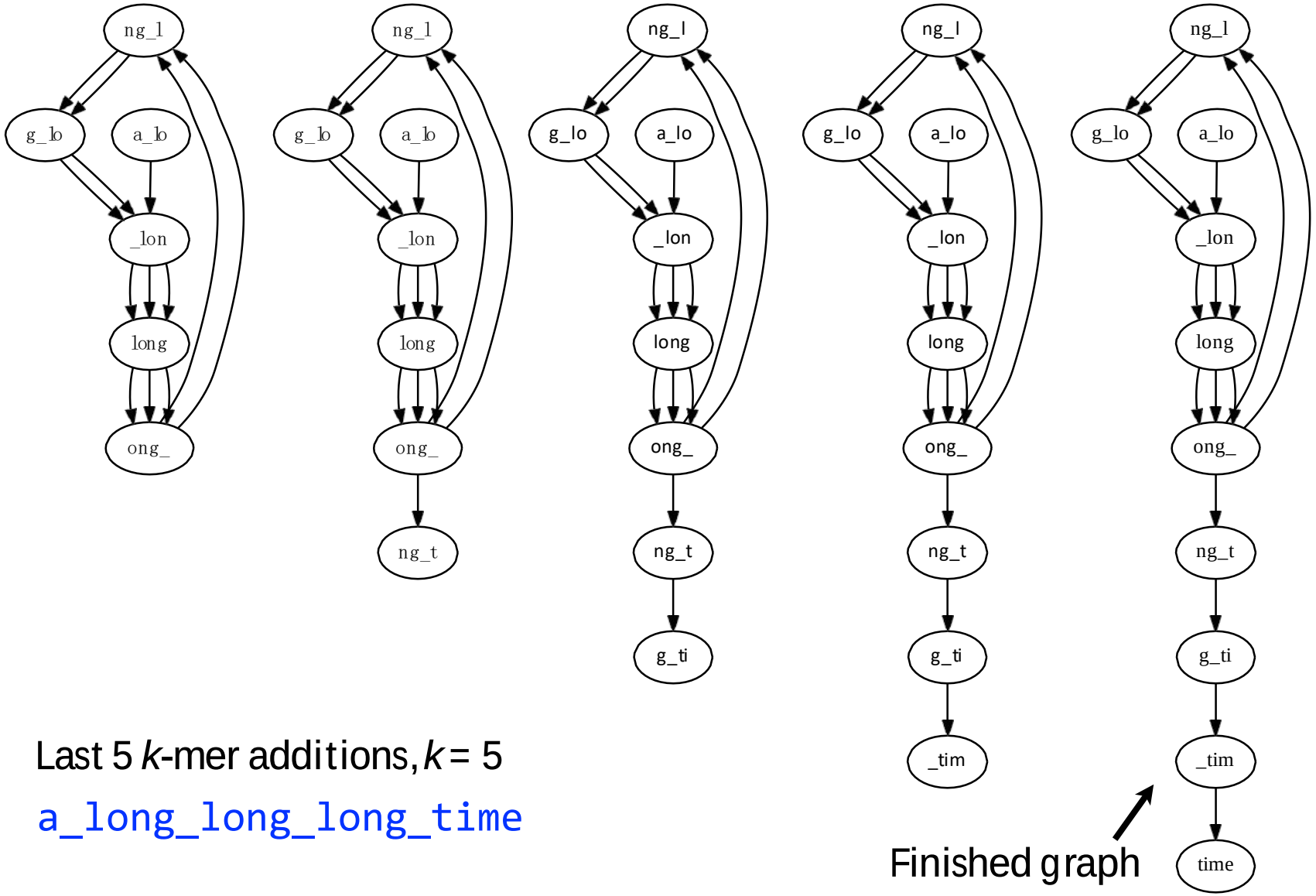
De Bruijn graph



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

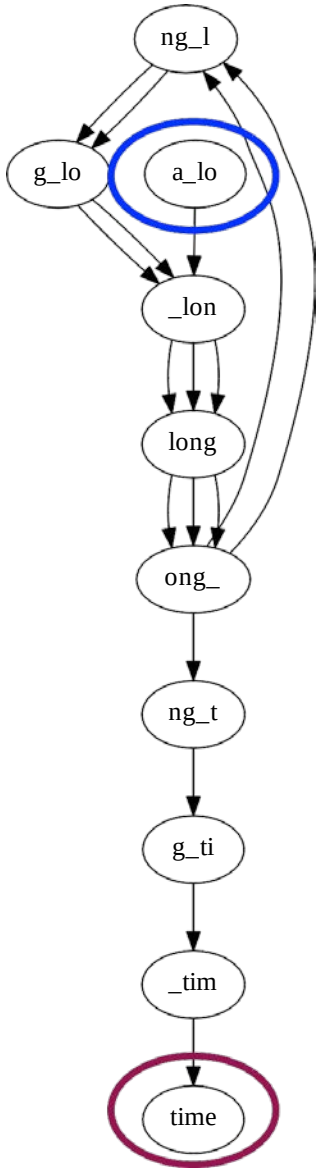
De Bruijn graph

Procedure yields 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 left- and right-most *k*-1-mers are equal

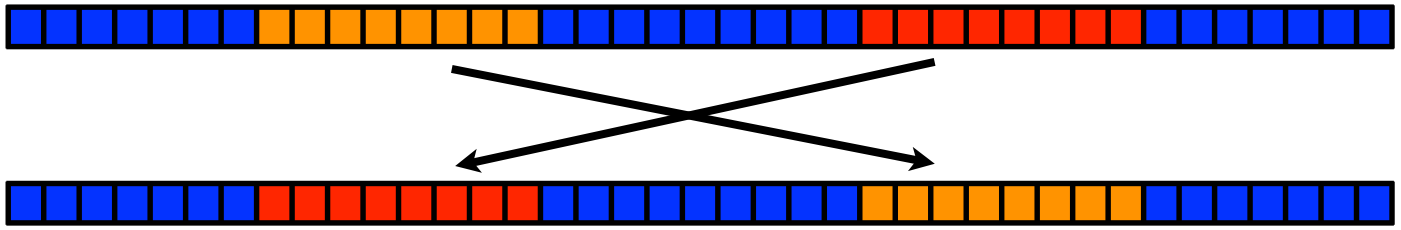
Third law of assembly

Repeats make assembly difficult; whether we can assemble without mistakes depends on length of reads and repetitive patterns in genome

Collapsing:

```
a_long_long_long_time
        ↓
a_long_long_time
```

Shuffling:



De Bruijn graph

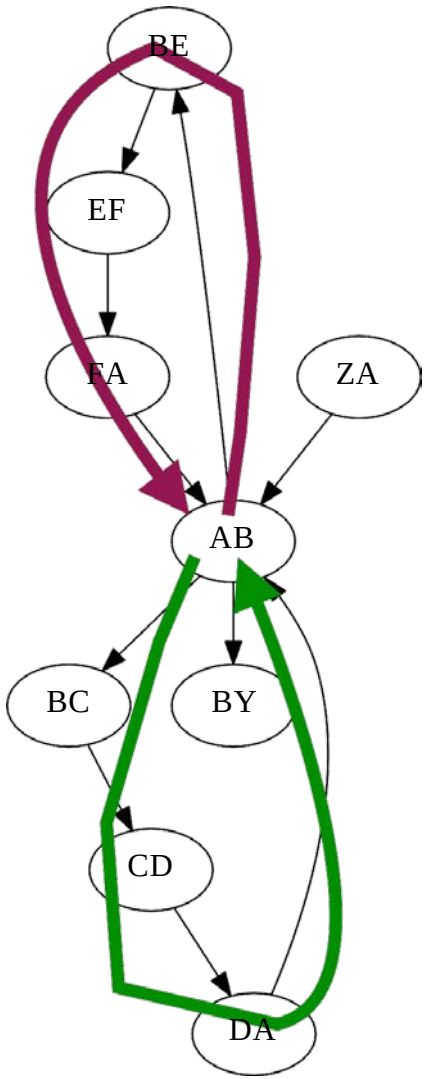
Right: graph for **ZABCDABEFABY**, $k = 3$

Problem 1: Repeats still cause misassembles

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

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

Only one walk correspond to the correct sequences, others correspond to incorrect shuffling.



Problem 2:

We've been building DBGs assuming "perfect" sequencing: each *k*-mer reported exactly once, no mistakes. Real datasets aren't like that.

De Bruijn graph

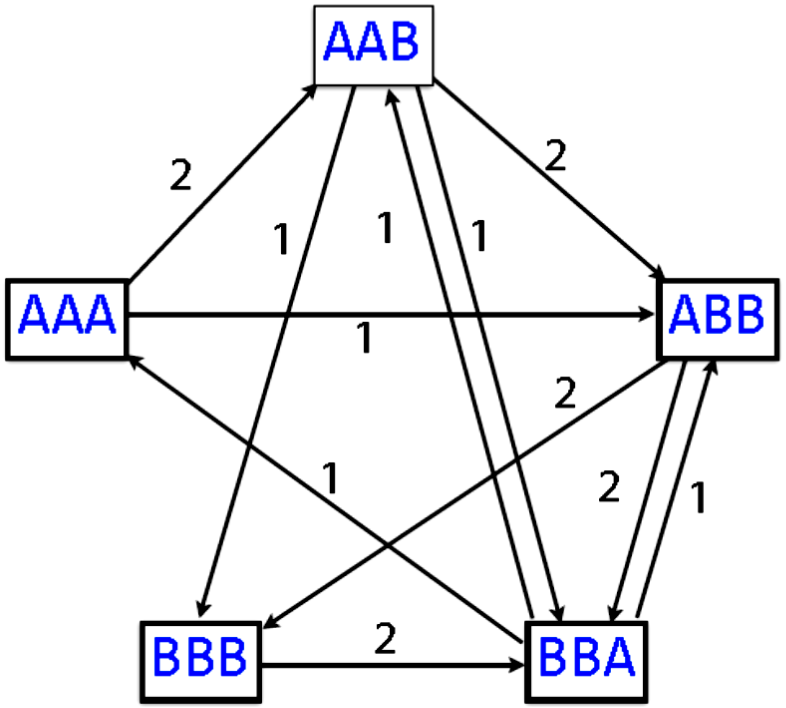
Casting assembly as Eulerian walk is appealing, but not practical

- 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.

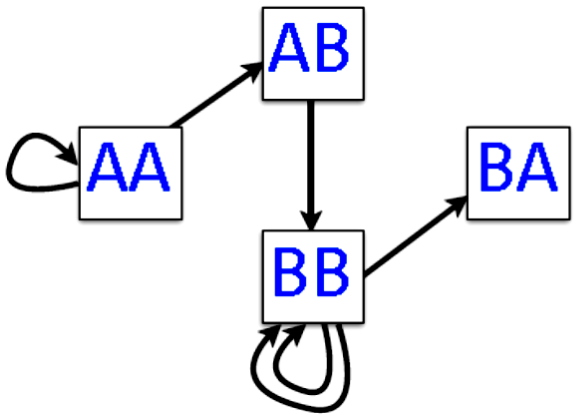
Assembly in Reality

Overlap graph



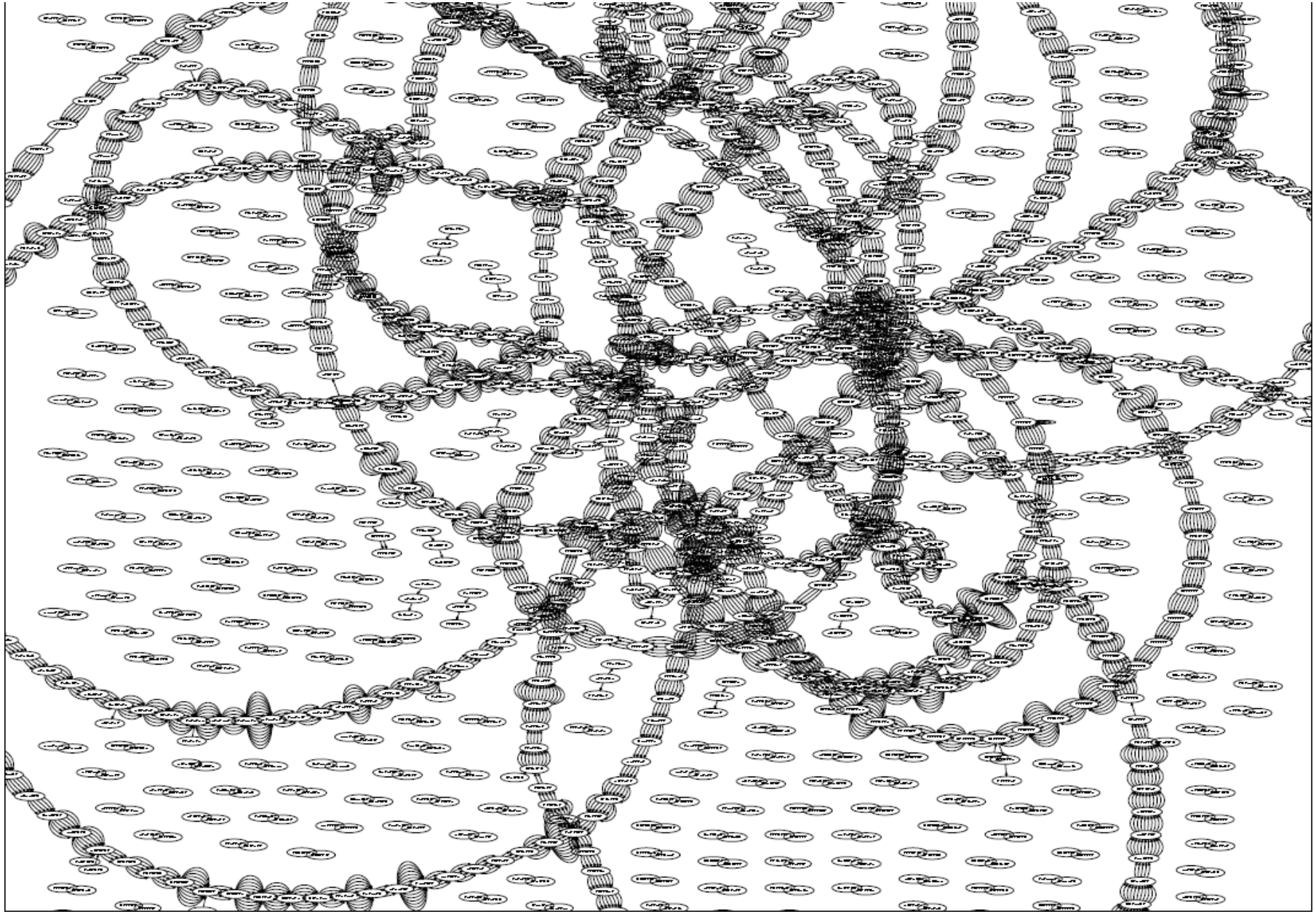
Overlap-Layout-Consensus (OLC) assembly

De Bruijn graph



De Bruijn Graph based (DBG) assembly

A real Genome will look like:



De Bruijn graph

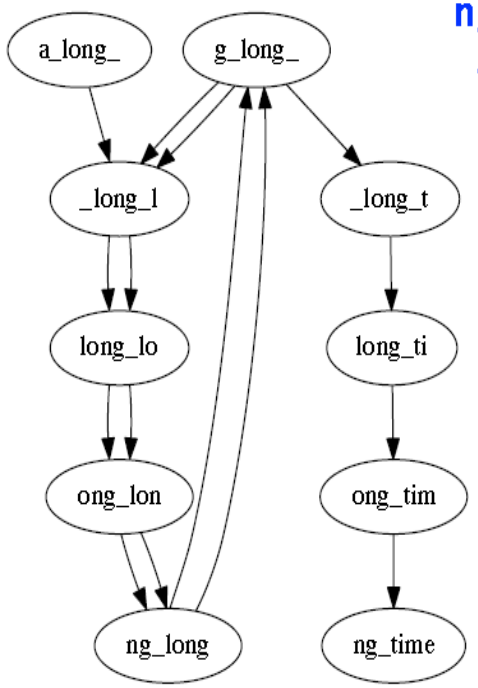
Longer reads, less repeats ambiguity

$k = 8$ Genome: a_long_long_long_time

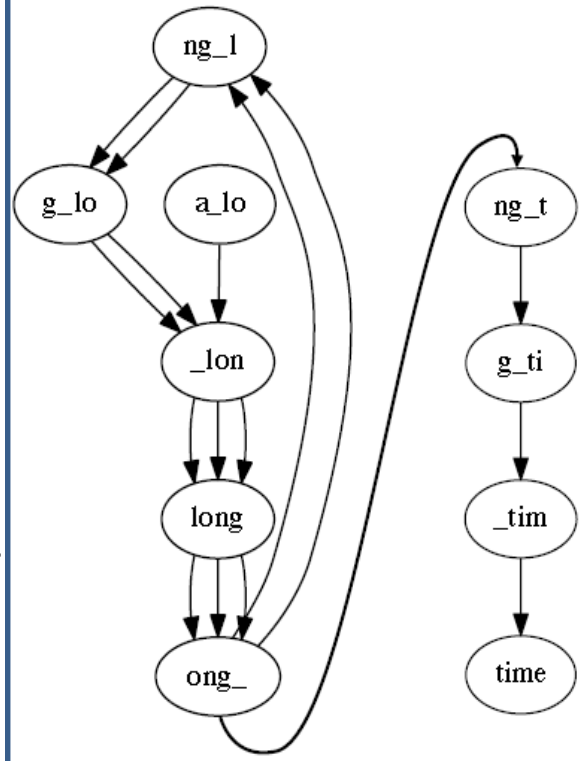
Reads: a_long_long_long, ng_long_l, g_long_time

k-mers:

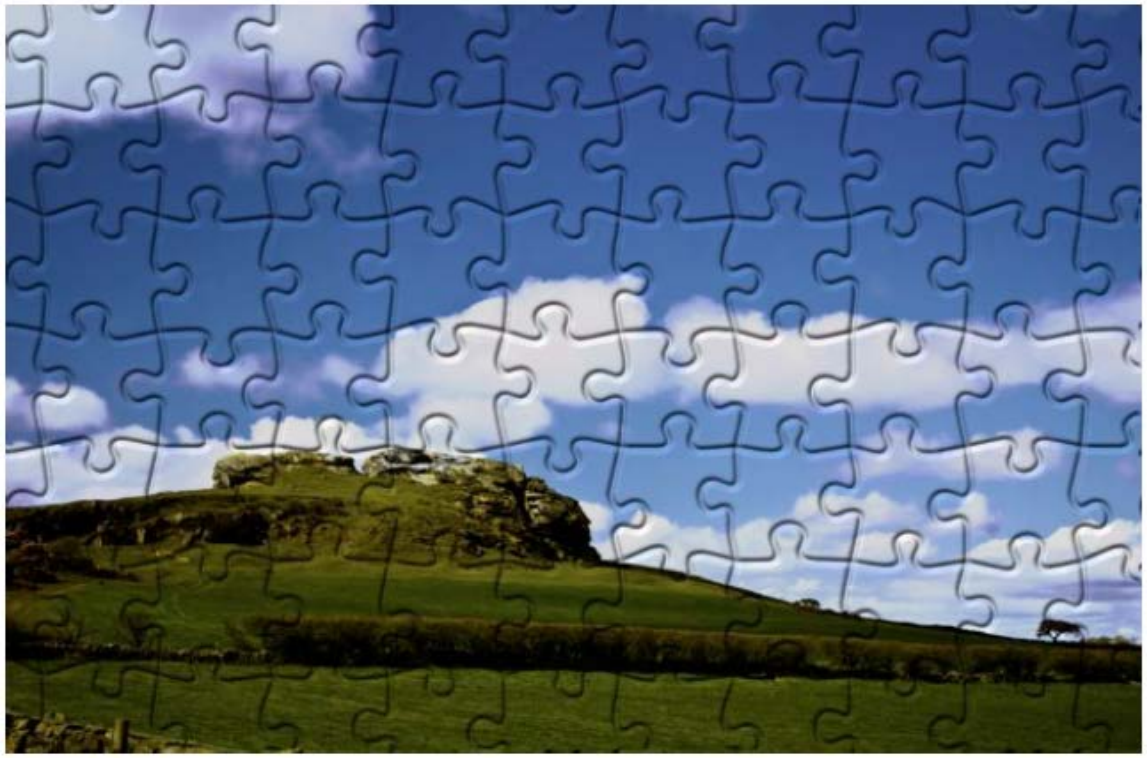
```
a_long_l          ng_long_   g_long_t
 _long_lo          g_long_l   _long_ti
  long_lon                    long_tim
   ong_long                    ong_time
    ng_long_
     g_long_l
      _long_lo
       long_lon
        ong_long
```



Graph for a_long_long_long_time, k = 5:



Repeats makes assembly difficult



Potential solution is to have longer reads, technically challenging

Greedy SCS on **6**-mers of
a_long_long_long_time

```
ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
ng_time ong_lon long_ti g_long_ a_long long_l
ong_lon long_time g_long_ a_long long_l
long_lon long_time g_long_ a_long
long_lon g_long_time a_long
long_long_time a_long
a_long_long_time
```

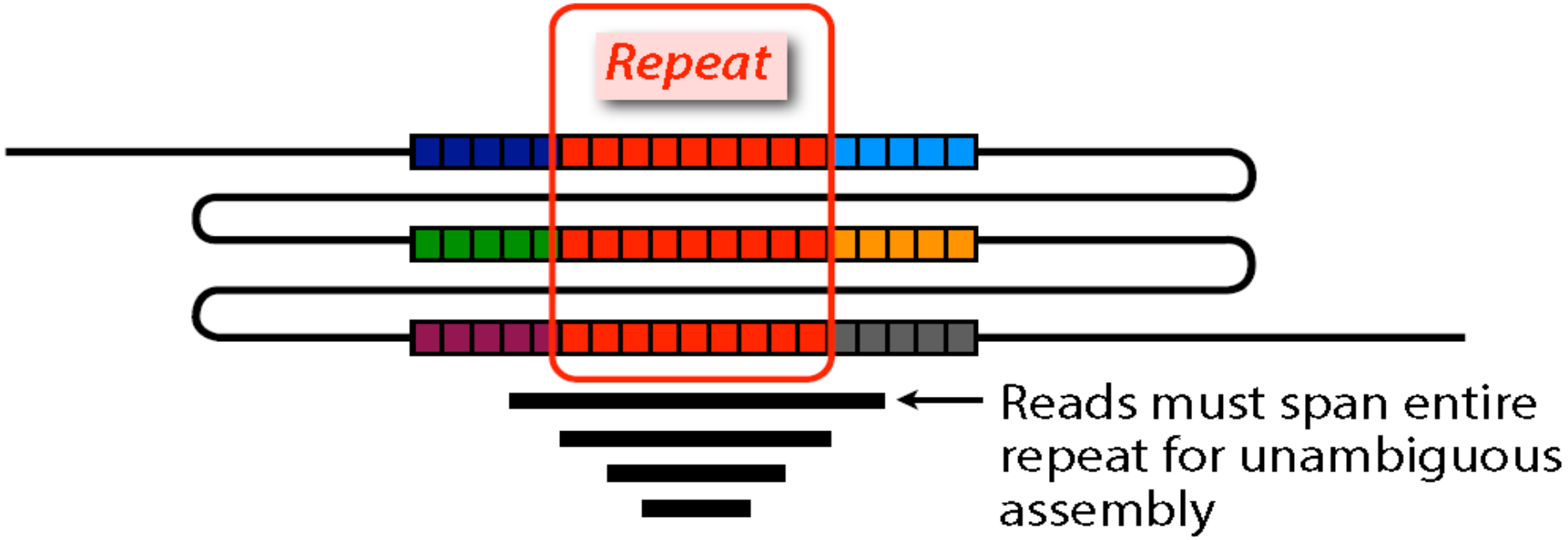
Greedy SCS on **8**-mers of
a_long_long_long_time

```
long_lon ng_long_ _long_lo g_long_t ong_long g_long_l ong_time a_long_l _long_ti long_tim
long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l _long_ti
_long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l
_long_time a_long_lo long_lon ng_long_ g_long_t ong_long g_long_l
_long_time ong_long_ a_long_lo long_lon g_long_t g_long_l
g_long_time ong_long_ a_long_lo long_lon g_long_l
g_long_time ong_long_ a_long_lon g_long_l
g_long_time ong_long_l a_long_lon
g_long_time a_long_long_l
a_long_long_long_time
```

a_long_long_long_time

g_long_l

- Longer reads may contain parts of non-repetitive portions.



- The reason that longer reads can counteract the problem of repetitive DNA is that they anchor repetitive sequences to their surrounding non-repetitive context.
- If the reads are long enough to extend through the repetitive sequence and overlap the non-repetitive sequence on either side, then that allows us to recreate the genome sequence unambiguously.
- Longs reads are technically difficult.
- Three are still ongoing efforts to solve repetitive issues.