String Algorithms and Data Structures

String Graph Assembly

CS 199-225

Brad Solomon

April 27, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Please fill out FLEX Evaluations

Always appreciate feedback on classes!

If not enough people fill it out, doesn't actually get recorded

Learning Objectives

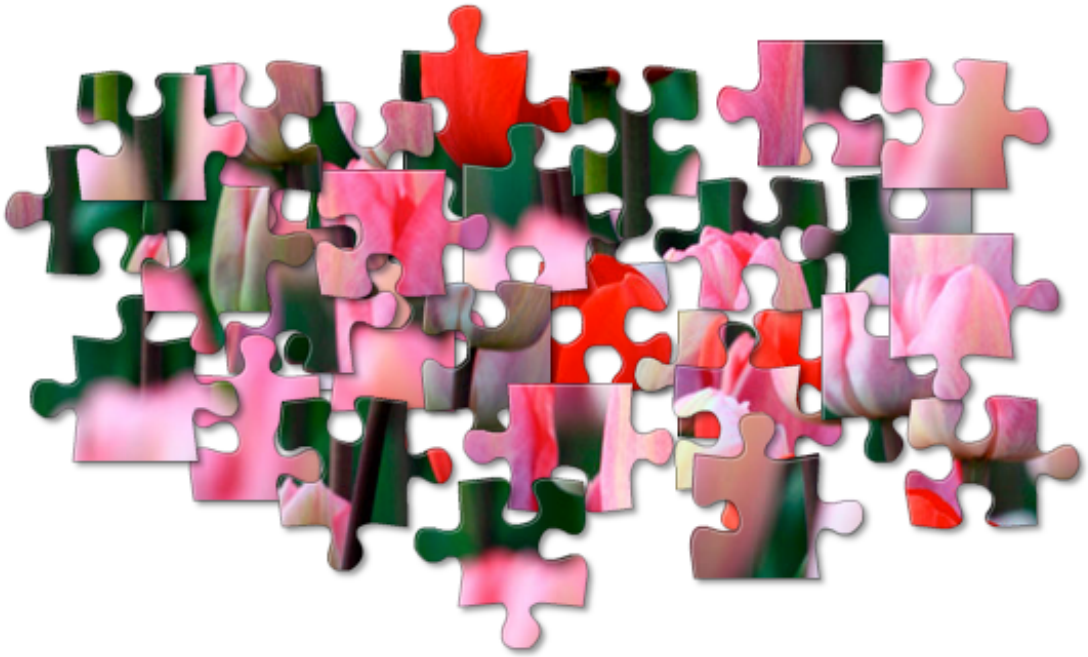
Introduce the string assembly problem

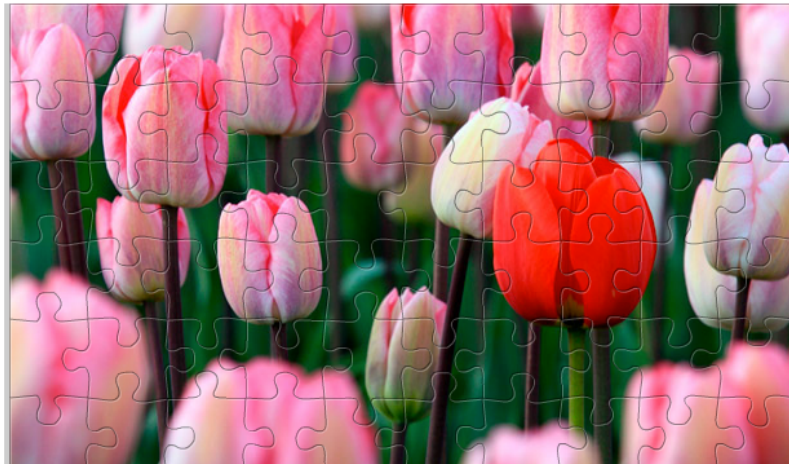
Address how exact / inexact overlaps can be computed “quickly”

Formalize the overlap-layout-consensus paradigm for assembly

Hint at the De Bruijn Graph

String Assembly





String Assembly

Whole-genome "shotgun" sequencing first copies the input DNA:

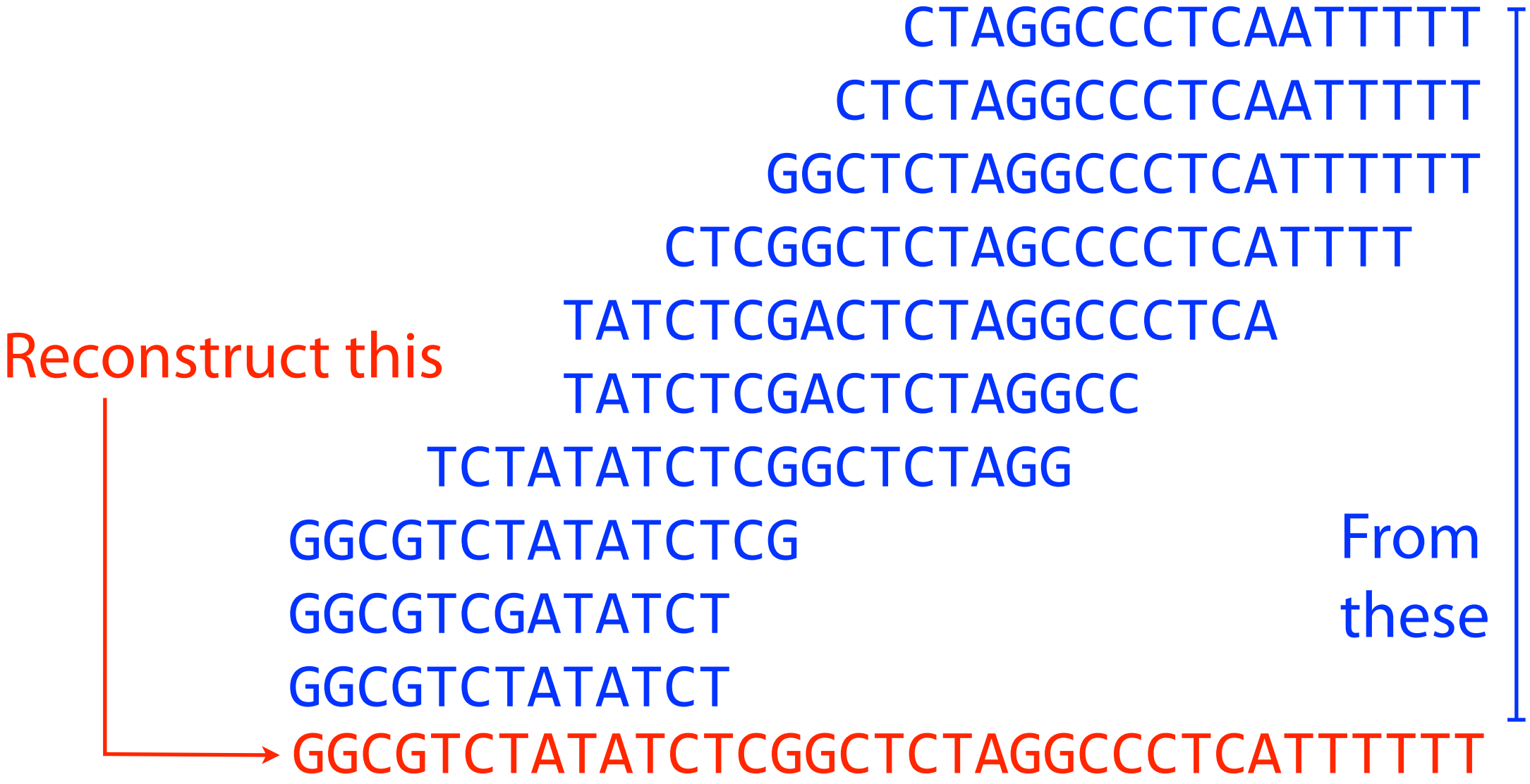
Input: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Copy: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

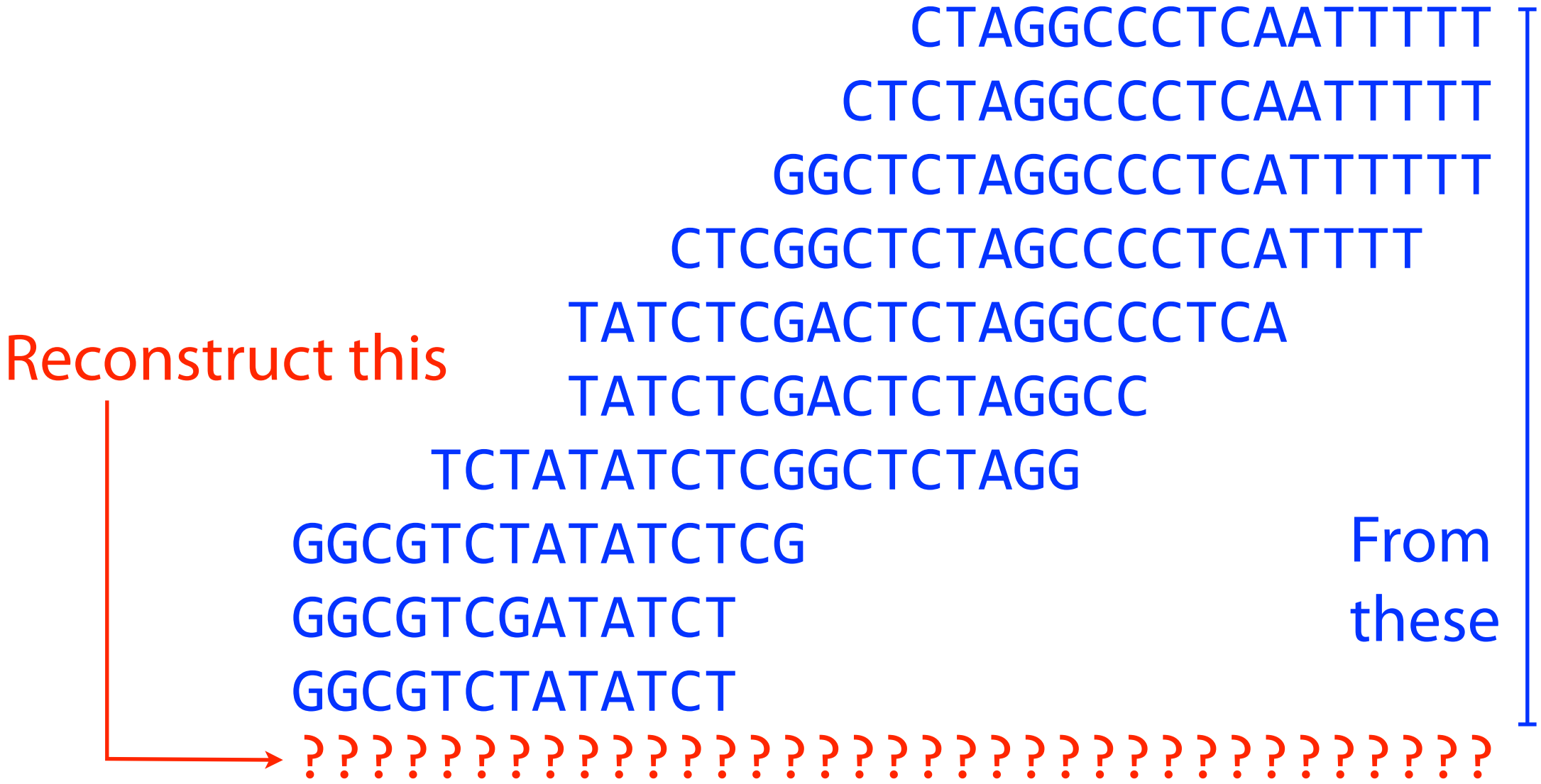
Then fragments it:

Fragment: GGCGTCTA TATCTCGG CTCTAGGCCCTC ATTTTTT
GGC GTCTATAT CTCGGCTCTAGGCCCTCA TTTTTT
GGCGTC TATATCT CGGCTCTAGGCCCT CATTTTTT
GGCGTCTAT ATCTCGGCTCTAG GCCCTCA TTTTTT

String Assembly



String Assembly



String Assembly

Input: A set of strings $S = \{s_1, s_2, \ldots, s_n\}$ assumed to be substrings of some underlying text *T*

Output: The 'best' approximation of *T*

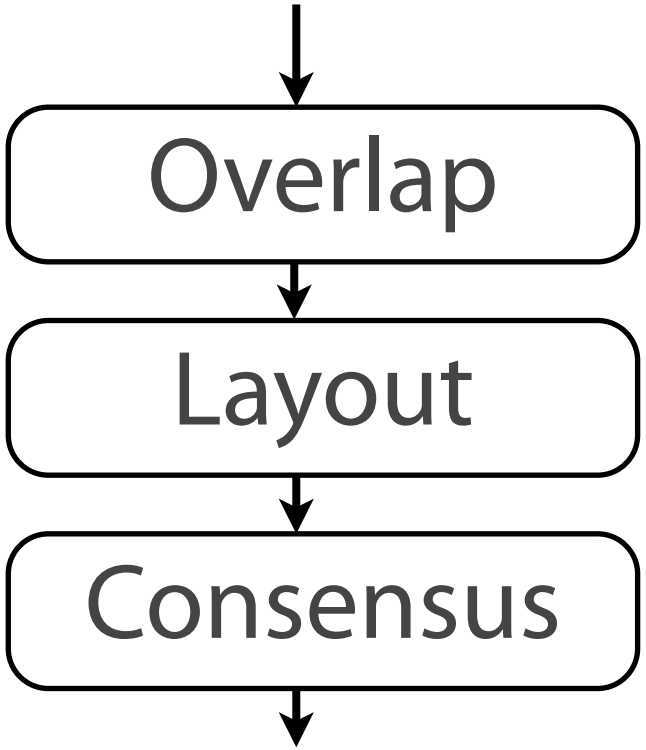
1) Identify all possible overlaps

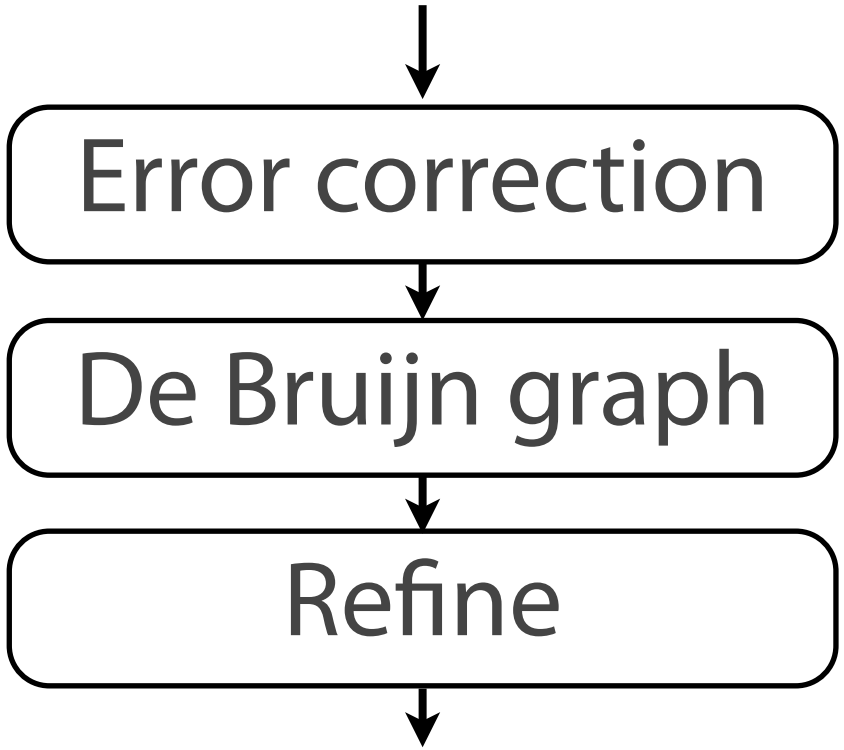
2) "Assemble" the best possible layout

3) Reconstruct *T* based on consensus

Assembly strategies

Two competing approaches using **graphs**!





Identify all possible overlaps

Given two strings, how can we find all overlaps?

X: `CTCTAGGCC`

Y: `TAGGCCCTC`

Identify all possible overlaps

How many unique strings do we need to search for?

X: CTCTAGGCC

T

TA

TAG

TAGG

TAGGC

TAGGCC

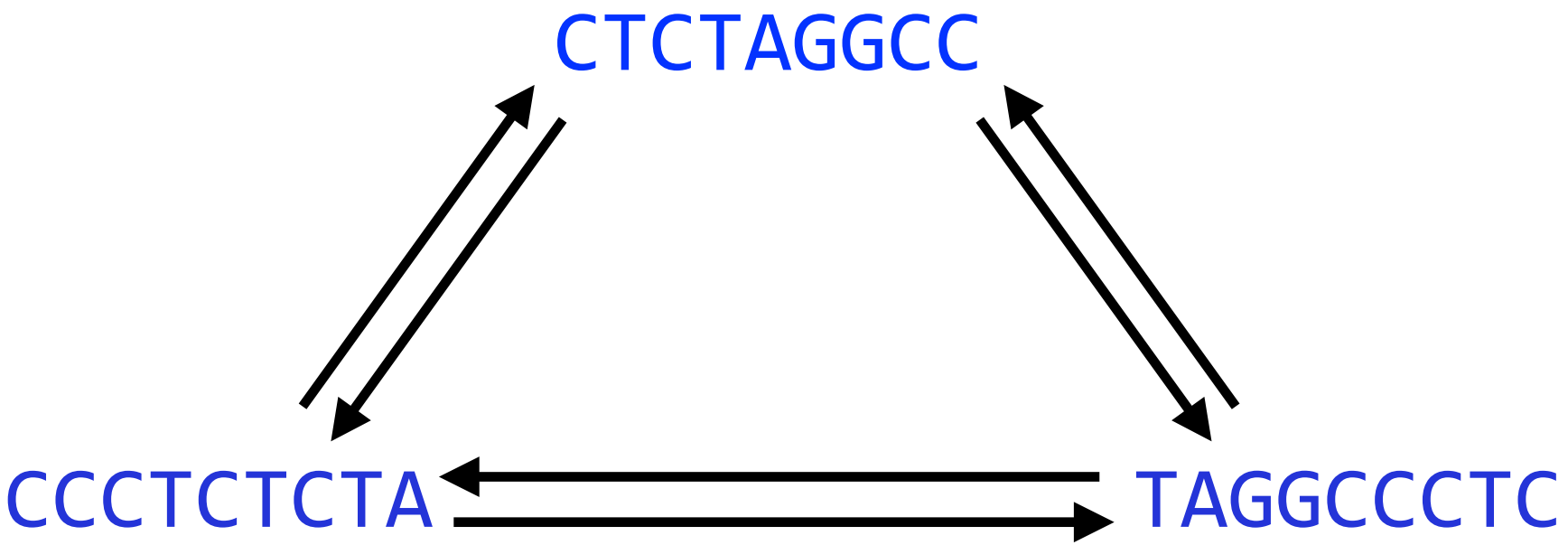
TAGGCCC

TAGGCCCT

TAGGCCCTC

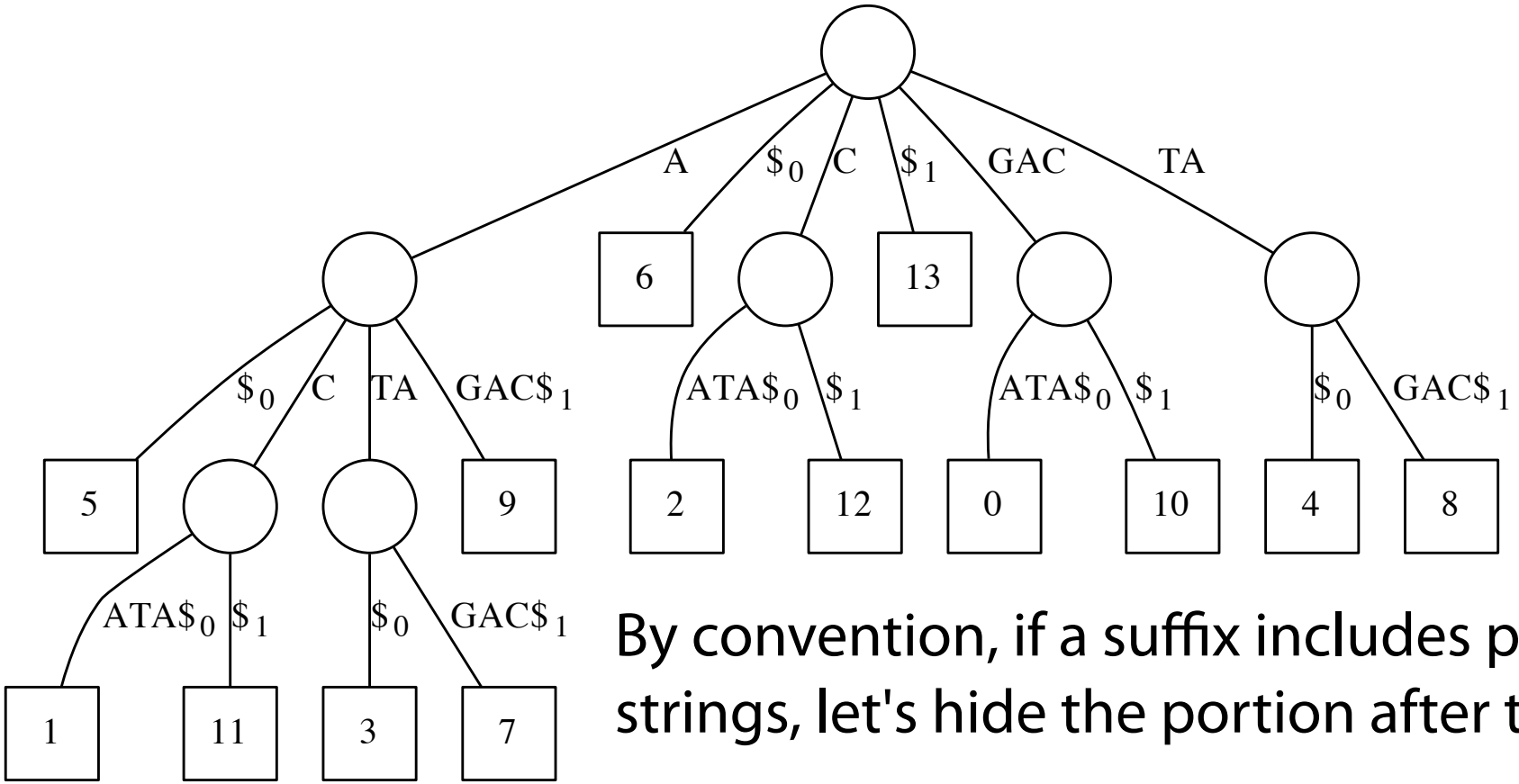
Identify all possible overlaps

If we are trying to find overlap between multiple strings?



Identify all possible overlaps

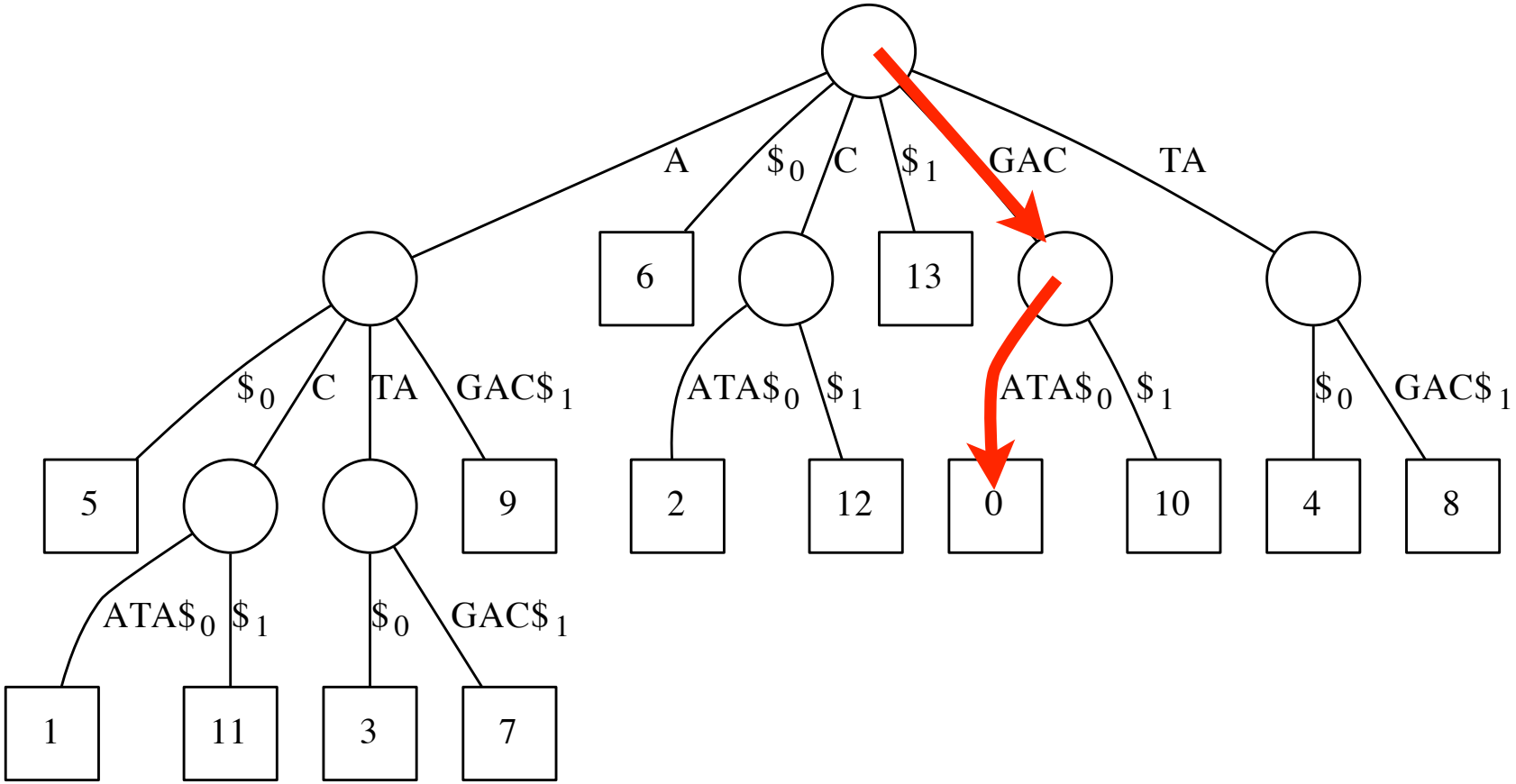
Generalized suffix tree for {"GACATA", "ATAGAC"} GACATA$_0$ATAGAC$_1$



By convention, if a suffix includes part of both strings, let's hide the portion after the first $.

Identify all possible overlaps

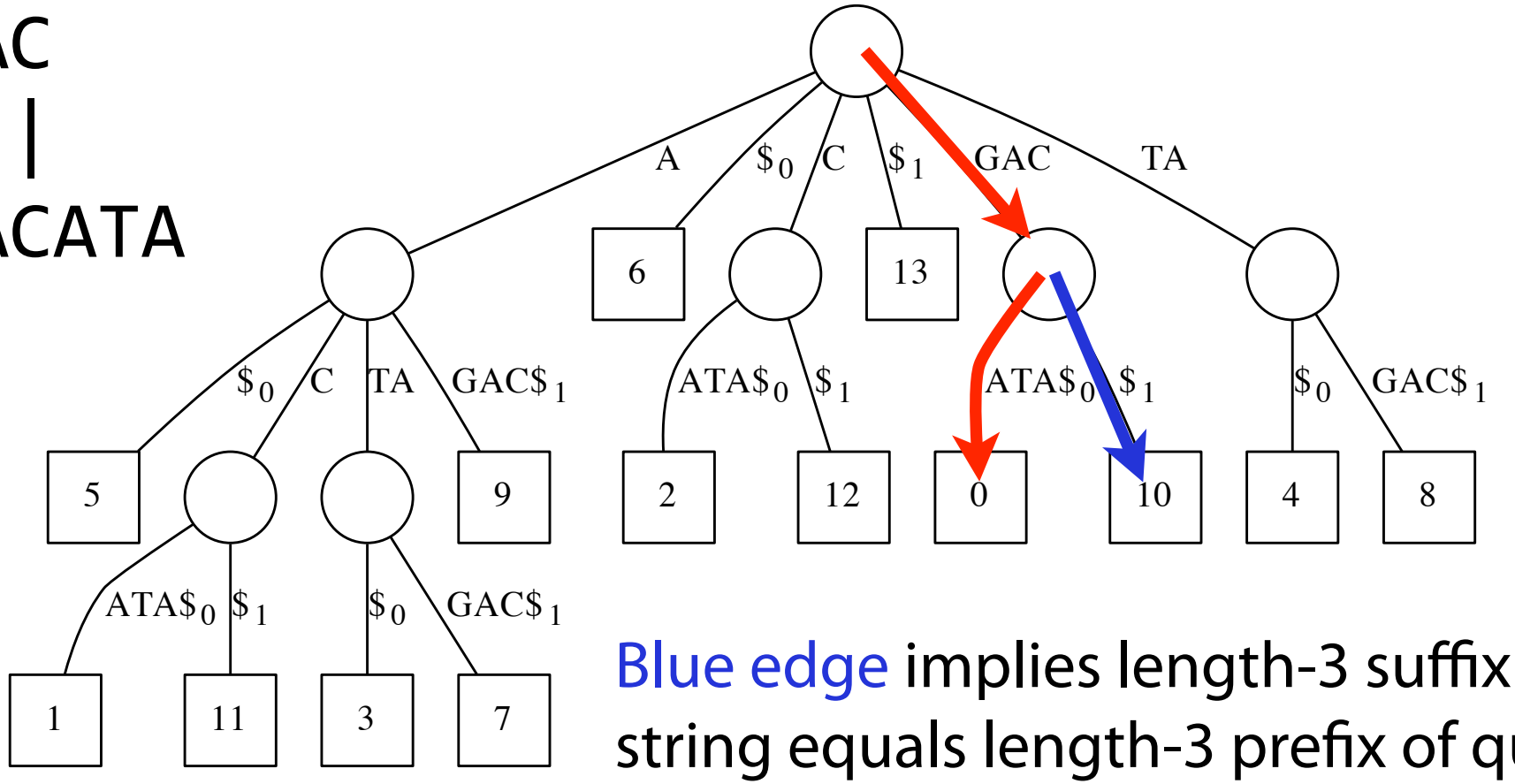
Let query = GACATA. From root, follow path labeled with query.



Identify all possible overlaps

Let query = GACATA. From root, follow path labeled with query.

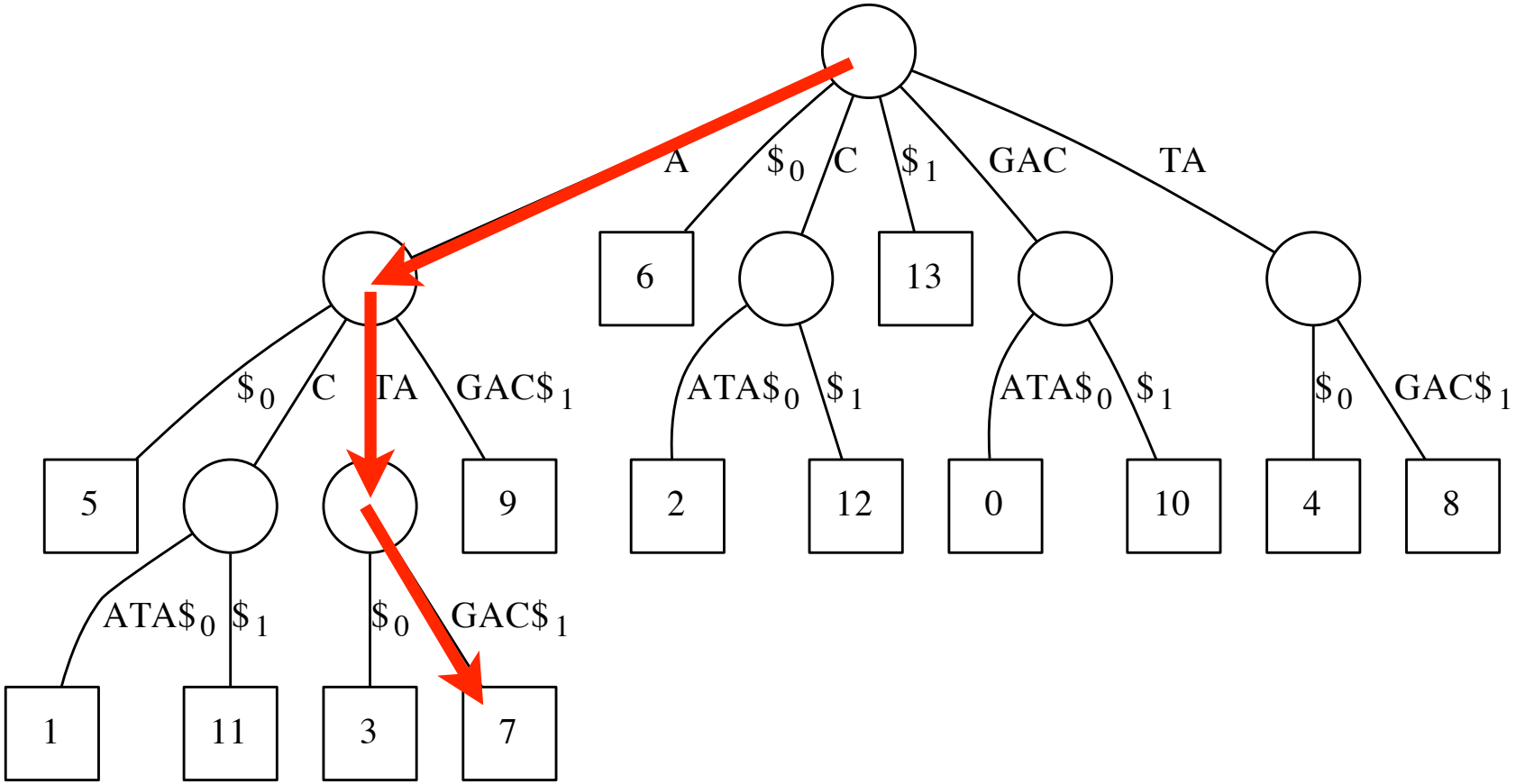
```
ATAGAC
  |||
GACATA
```



Blue edge implies length-3 suffix of second string equals length-3 prefix of query

Identify all possible overlaps

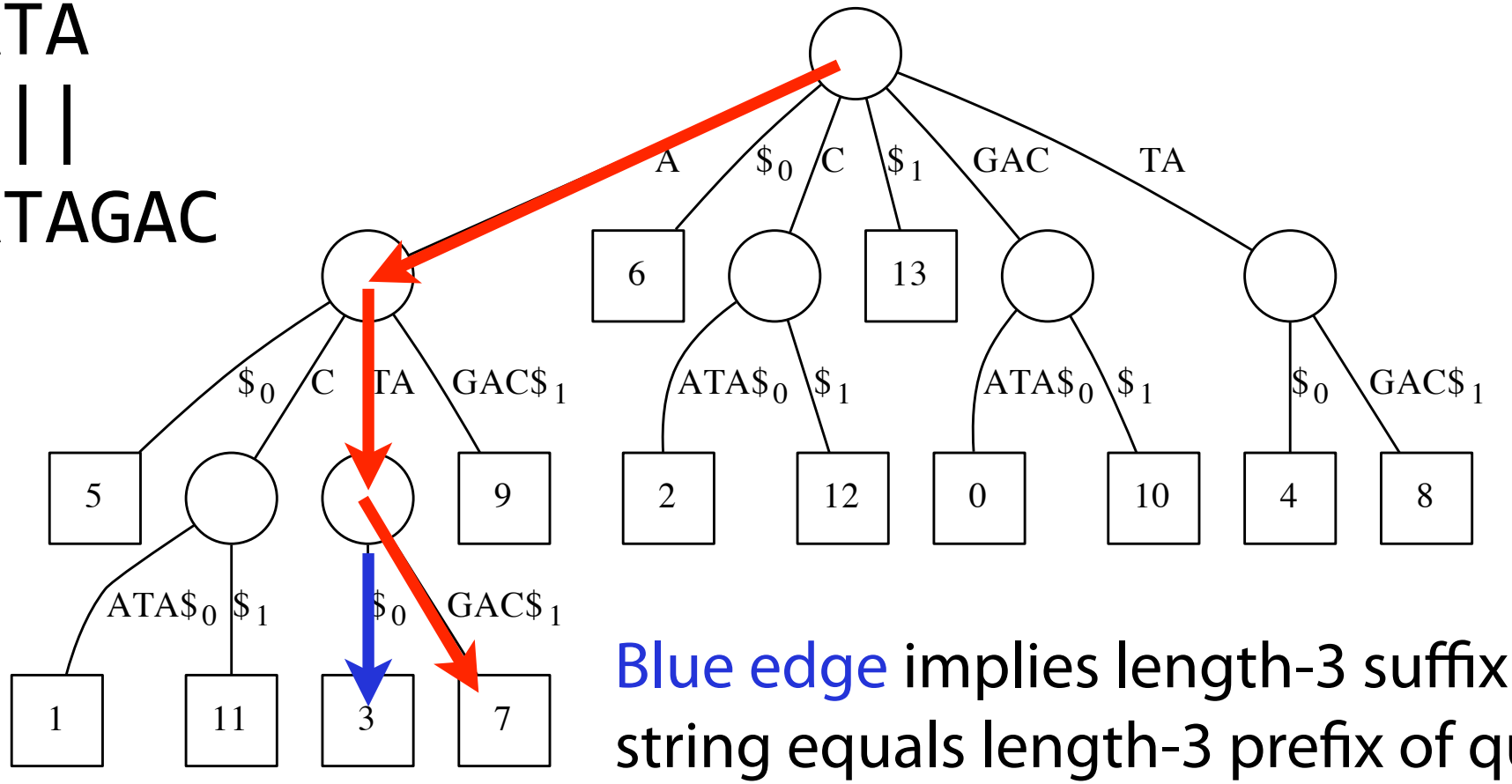
Let query = ATAGAC. From root, follow path labeled with query.



Identify all possible overlaps

Let query = ATAGAC. From root, follow path labeled with query.

```
GACATA
  |||
ATAGAC
```



Blue edge implies length-3 suffix of first string equals length-3 prefix of query

Identify all possible overlaps

What is the Big O for a generalized suffix tree solution?

Let n be the number of strings and m the length of each string

Time to build generalized suffix tree:

To walk down each string in the tree:

... to find & report overlaps:

Overall:

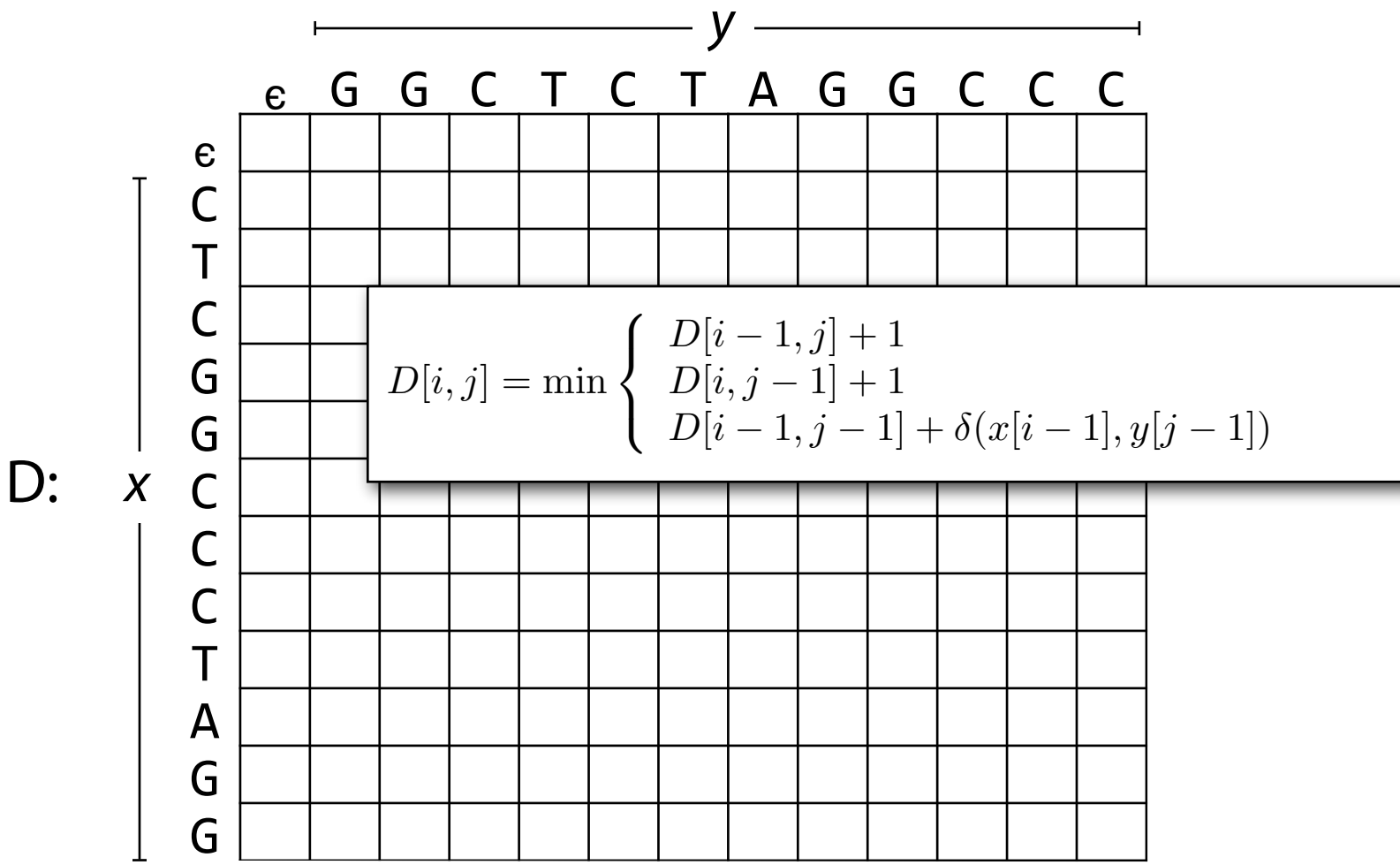
Identify all possible overlaps

What about *approximate* overlaps?

```
X: CTCGGCCCTAGG
      |||  |||||
Y:    GGCTCTAGGCCC
```

Identify all possible overlaps

How can we adjust this dynamic program for **overlaps**?



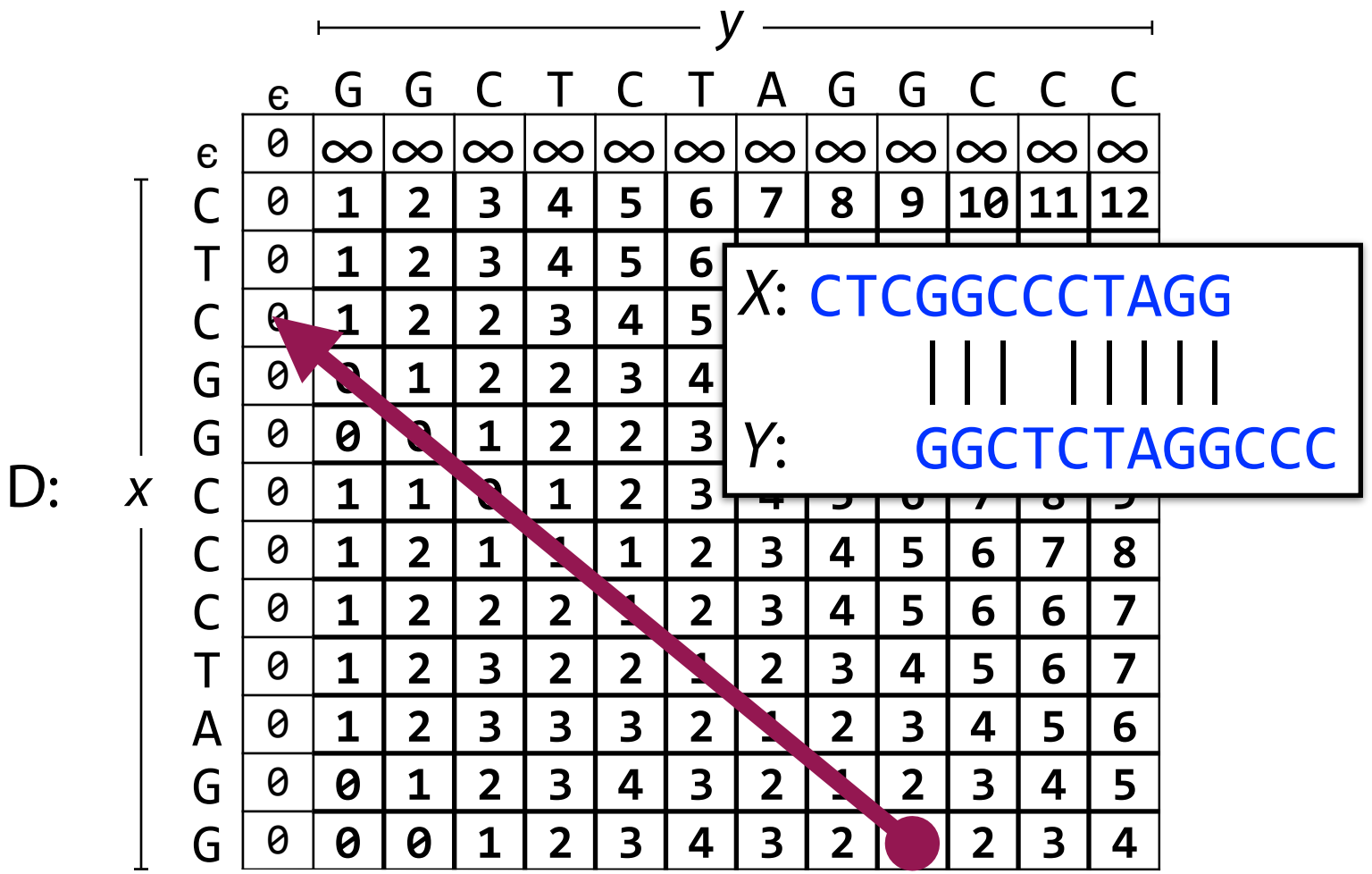
Identify all possible overlaps

How can we adjust this dynamic program for **overlaps**?

First row gets ∞s

First column gets 0s

Backtrace from last row



Identify all possible overlaps

What is the Big O for a dynamic program solution?

Let n be the number of strings and m the length of each string

Number of overlap pairs:

Size of each matrix:

Overall:

Identify all possible overlaps

What is the Big O for a dynamic program solution?

Let n be the number of strings and m the length of each string

Suffix Tree: $O(nm + \alpha)$.

Dynamic Program: $O(n^2m^2)$

True solutions use both! Filter with tree and solve with dynamic

Wajid, Bilal, and Erchin Serpedin. "Review of general algorithmic features for genome assemblers for next generation sequencers." *Genomics, proteomics & bioinformatics* 10.2 (2012): 58-73.

Sohn, Jang-il, and Jin-Wu Nam. "The present and future of de novo whole-genome assembly." *Briefings in bioinformatics* 19.1 (2018): 23-40.

String Assembly

Input: A set of strings $S = \{s_1, s_2, \ldots, s_n\}$ assumed to be substrings of some underlying text *T*

Output: The 'best' approximation of *T*

1) Identify all possible overlaps

Solved with suffix tree / dynamic programming!

2) "Assemble" the best possible layout

3) Reconstruct *T* based on consensus

Storing and assembling overlaps

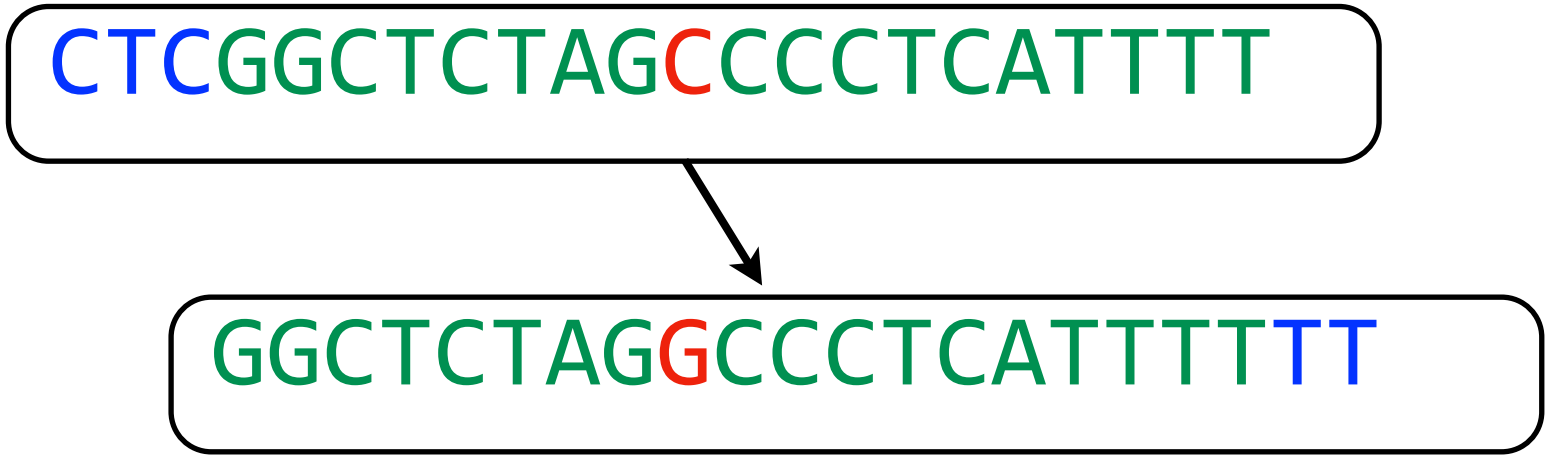
How do we store all our overlaps?

Storing and assembling overlaps

How do we store all our overlaps?

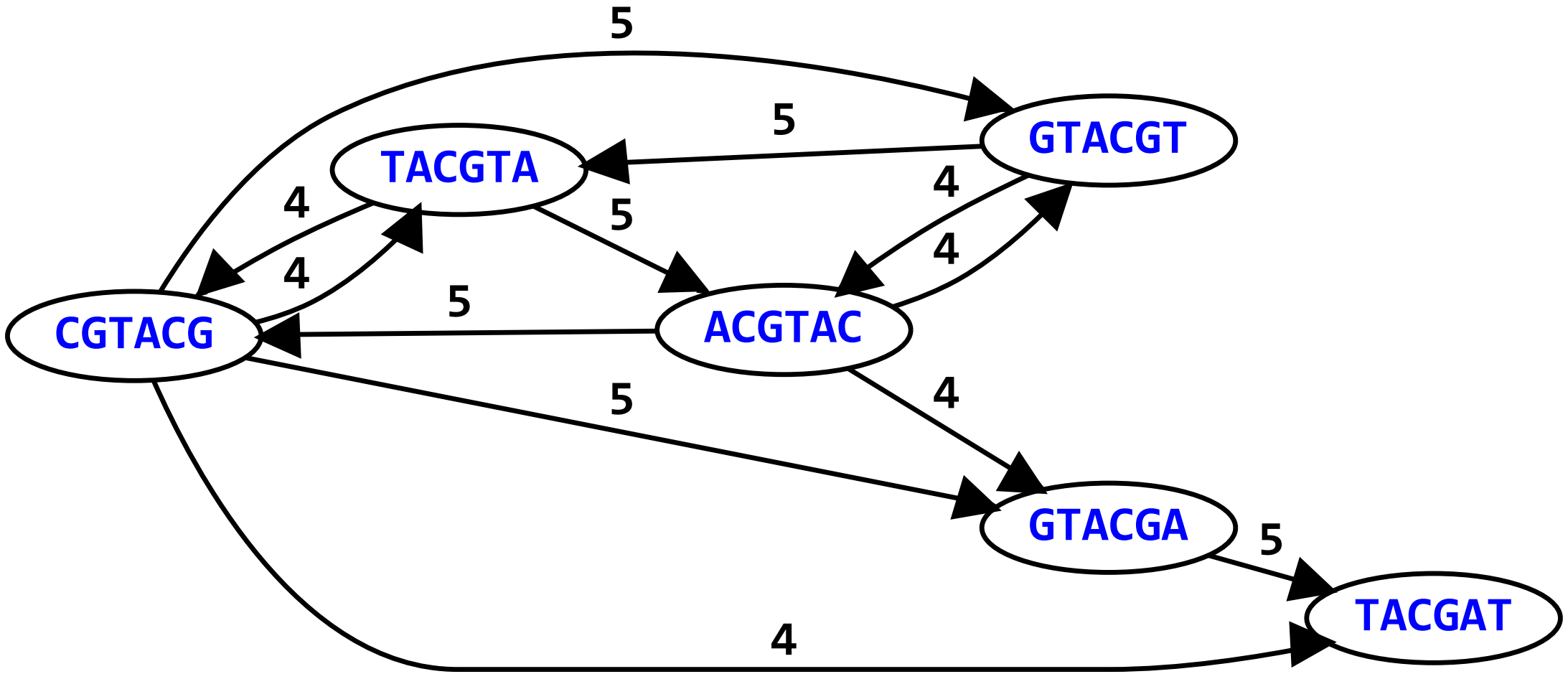
Each node is a string

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



Overlap Graph Assembly

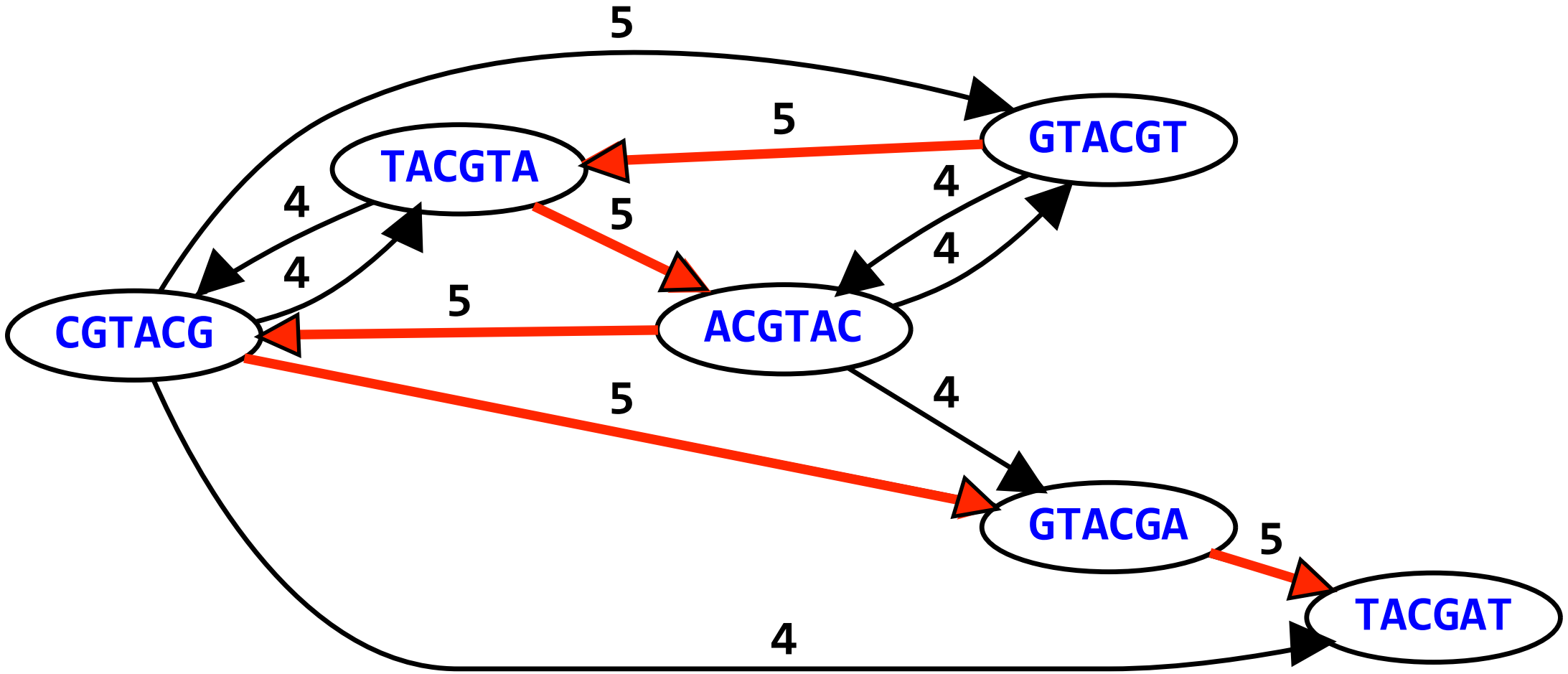
Our best assembly *should* be a path through all nodes in the graph!



Overlap Graph Assembly

Our best assembly *should* be a path through all nodes in the graph!

One reasonable idea: *shortest common superstring* (SCS)

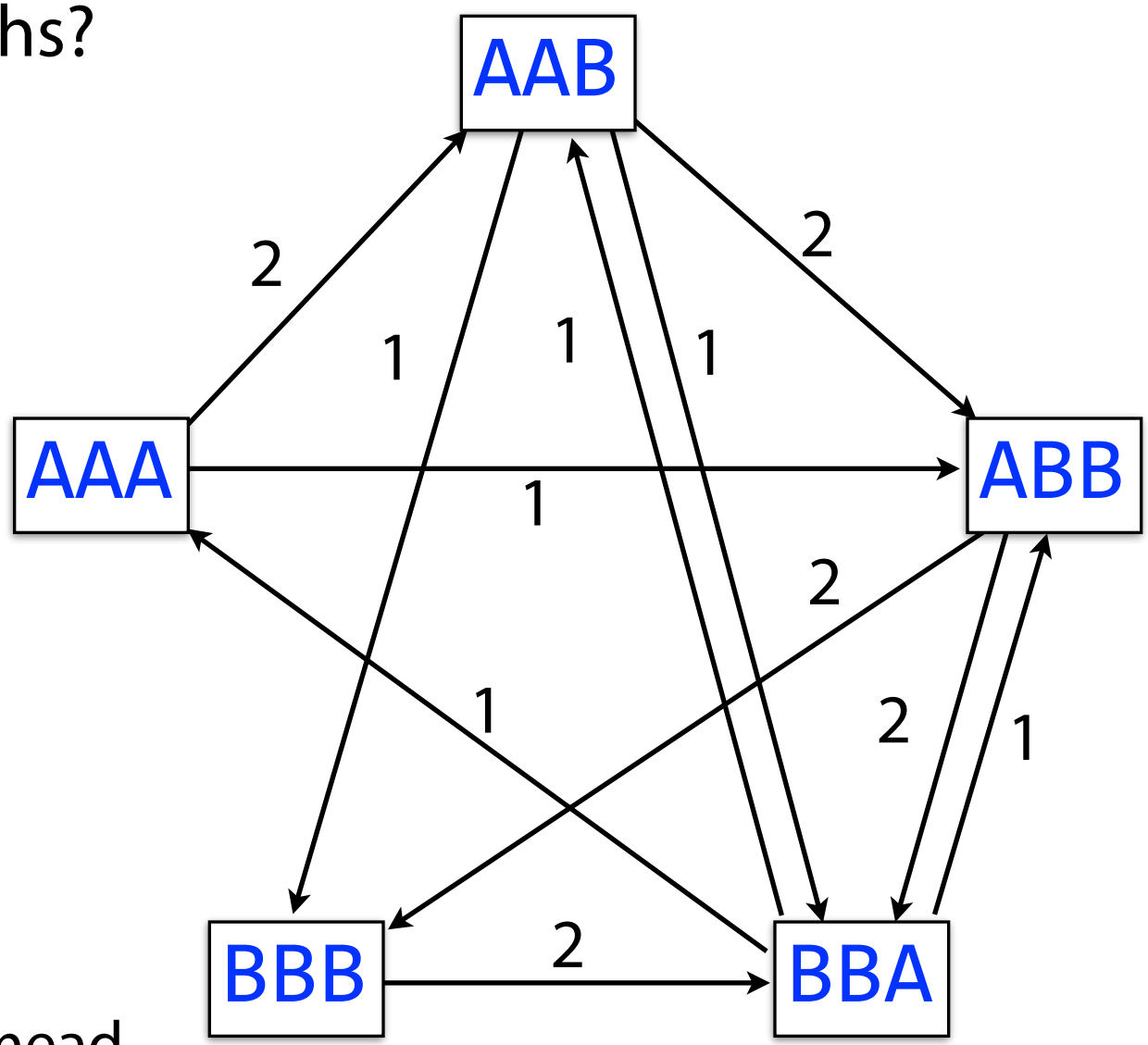


Shortest Common Superstring

Input strings

AAA AAB ABB BBB BBA

How can we solve SCS using graphs?



Original example courtesy of Ben Langmead

Shortest Common Superstring

Input strings

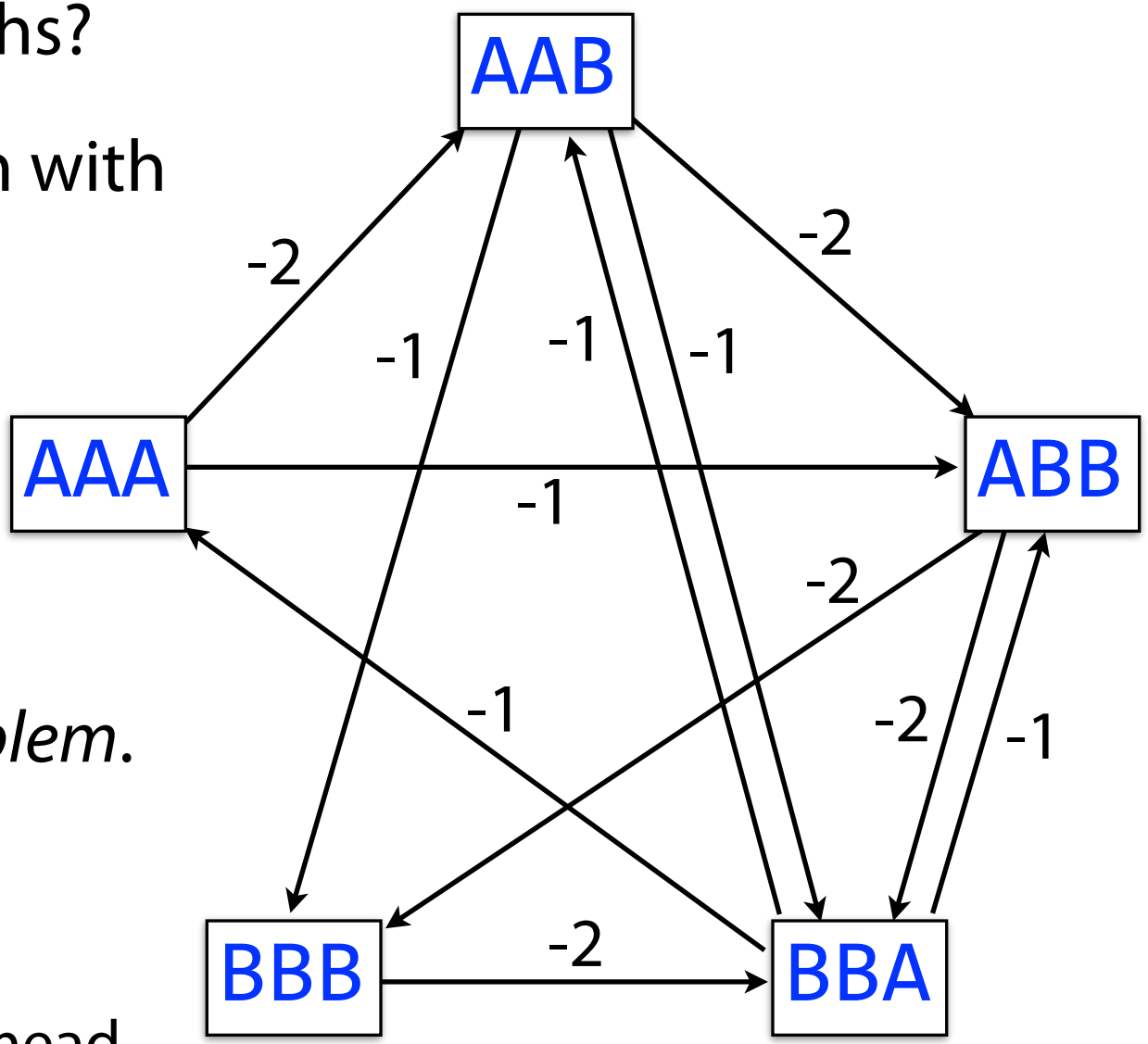
AAA AAB ABB BBB BBA

How can we solve SCS using graphs?

Imagine a modified overlap graph with edge weight = - (overlap)

The SCS is a path that visits every node once, minimizing total cost

That's the *Traveling Salesman Problem*. **NP-Hard!**



Original example courtesy of Ben Langmead

Shortest Common Superstring: Exhaustive

Pick order for strings in *S and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAA

Shortest Common Superstring: Exhaustive

Pick order for strings in *S and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAAB

Take into account overlap whenever possible

Shortest Common Superstring: Exhaustive

Pick order for strings in *S and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABA

Shortest Common Superstring: Exhaustive

Pick order for strings in *S and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABB

Shortest Common Superstring: Exhaustive

Pick order for strings in S *and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAA

Shortest Common Superstring: Exhaustive

Pick order for strings in *S and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABAB

Concatenate full string when no overlap

Shortest Common Superstring: Exhaustive

Pick order for strings in *S and* construct superstring

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB ⟵ superstring 1

Shortest Common Superstring: Exhaustive

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, how many orderings are are possible?

Shortest Common Superstring: Exhaustive

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, how many orderings are are possible?

n ! (*n* factorial) orderings possible

SCS: Greedy

Repeatedly merge pair of strings with maximal overlap.

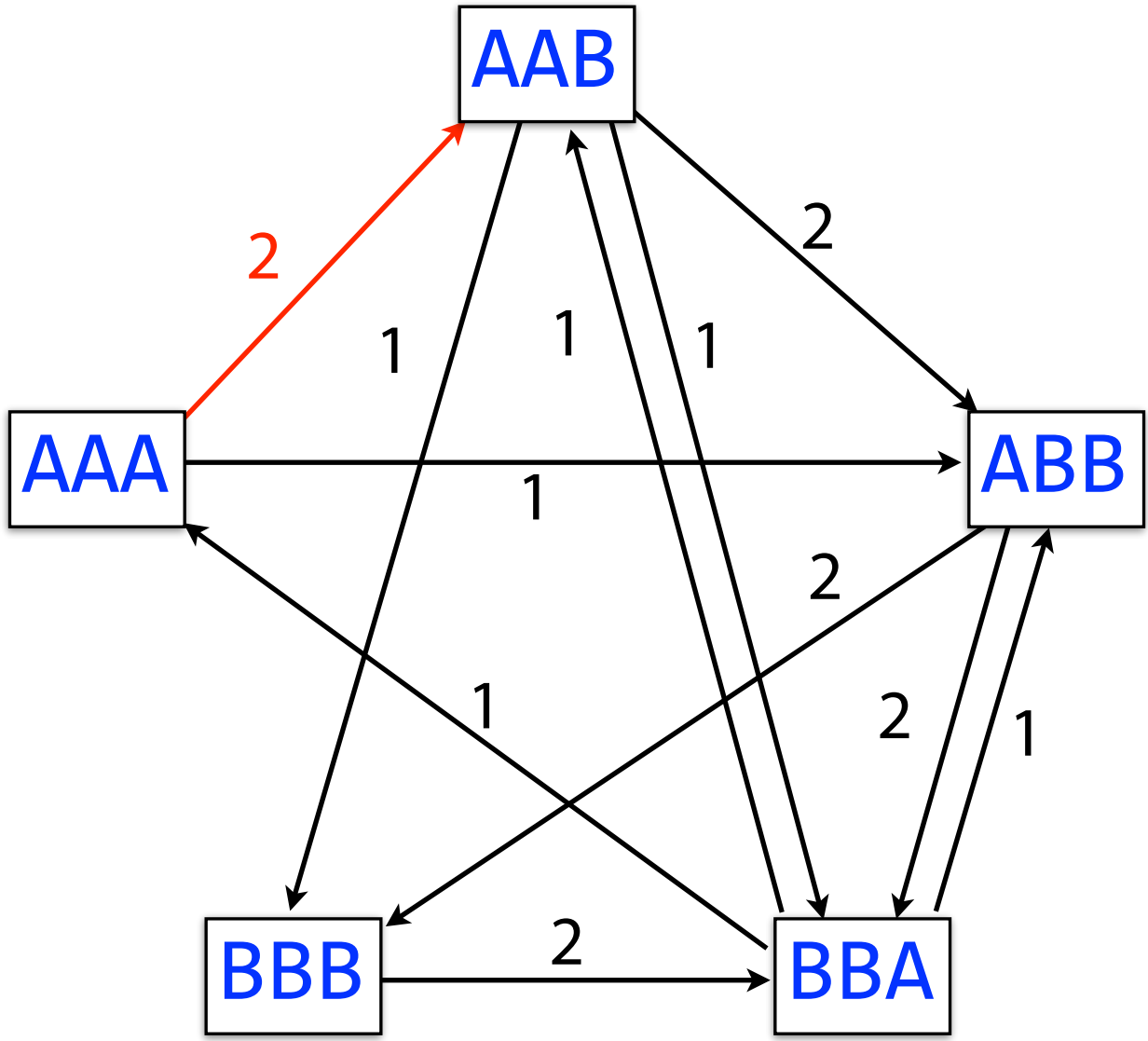
Stop when there's 1 string left.

l = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA



SCS: Greedy

Repeatedly merge pair of strings with maximal overlap.

Stop when there's 1 string left.

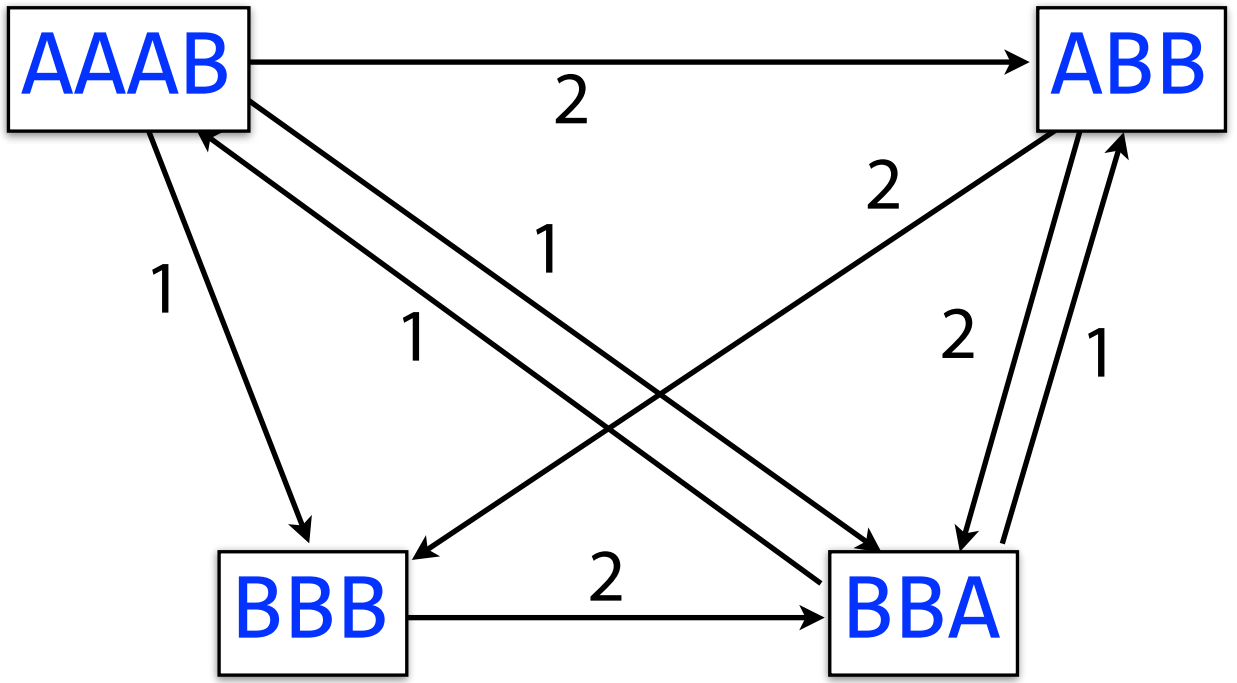
l = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



SCS: Greedy

Repeatedly merge pair of strings with maximal overlap.

Stop when there's 1 string left.

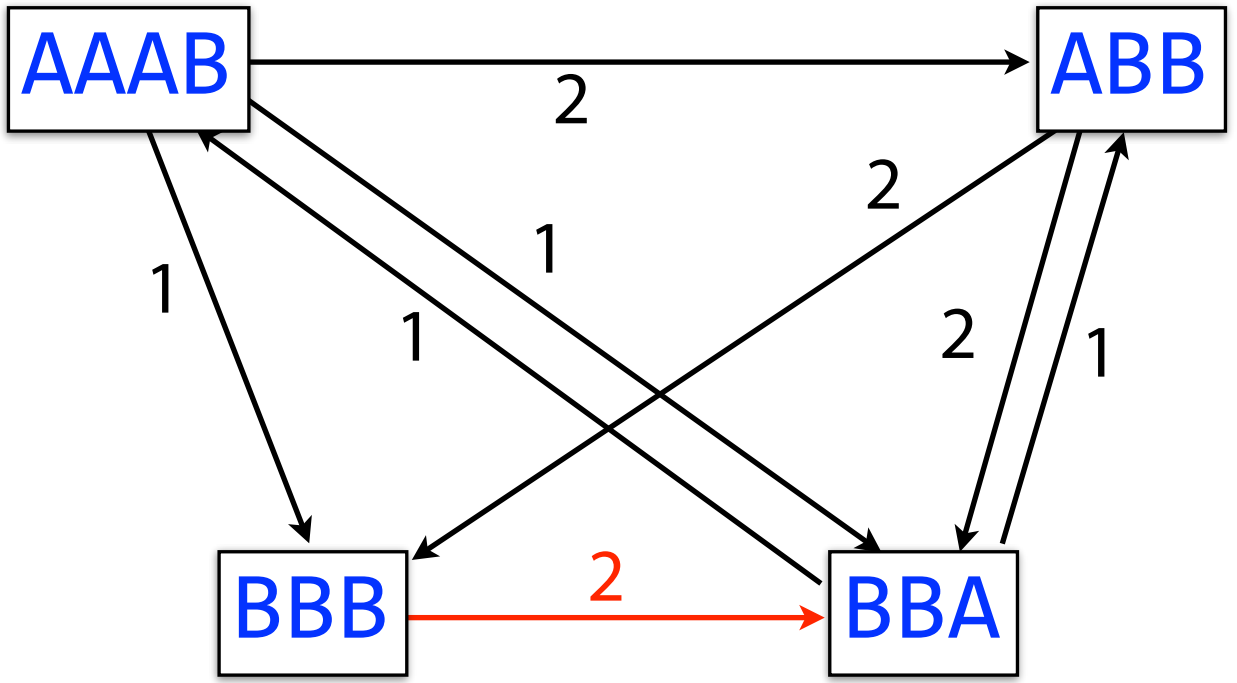
l = minimum overlap.

Algorithm in action ($l = 1$):

⊢—— Input strings ——⊣

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



SCS: Greedy

Repeatedly merge pair of strings with maximal overlap.

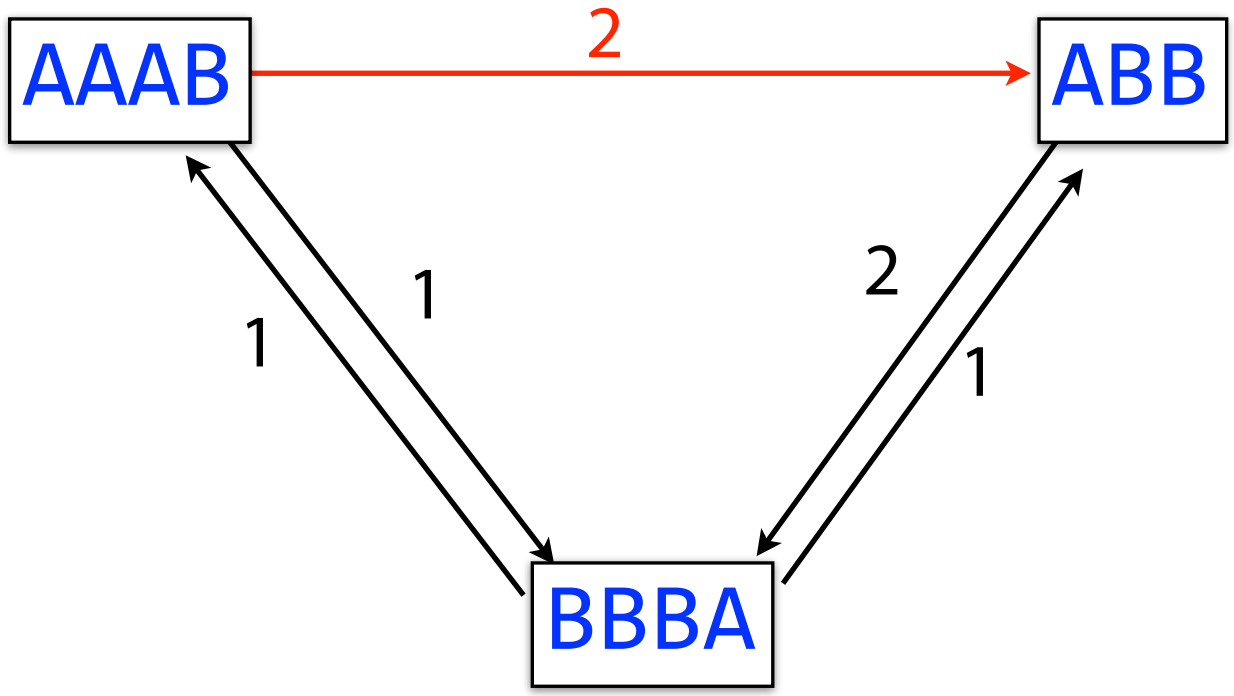
Stop when there's 1 string left.

l = minimum overlap.

Algorithm in action ($l = 1$):

⟵ Input strings ⟶

AAA AAB ABB BBB BBA
AAAB ABB BBB BBA
AAAB BBBA ABB



SCS: Greedy

Repeatedly merge pair of strings with maximal overlap.

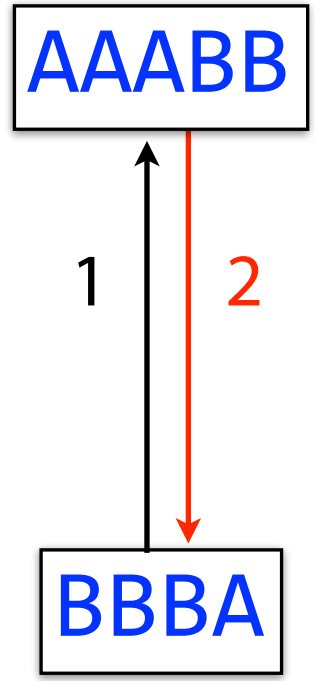
Stop when there's 1 string left.

l = minimum overlap.

Algorithm in action ($l = 1$):

⟵ Input strings ⟶

AAA AAB ABB BBB BBA
AAAB ABB BBB BBA
AAAB BBBA ABB
AAABB BBBA



SCS: Greedy

Repeatedly merge pair of strings with maximal overlap.

Stop when there's 1 string left.

l = minimum overlap.

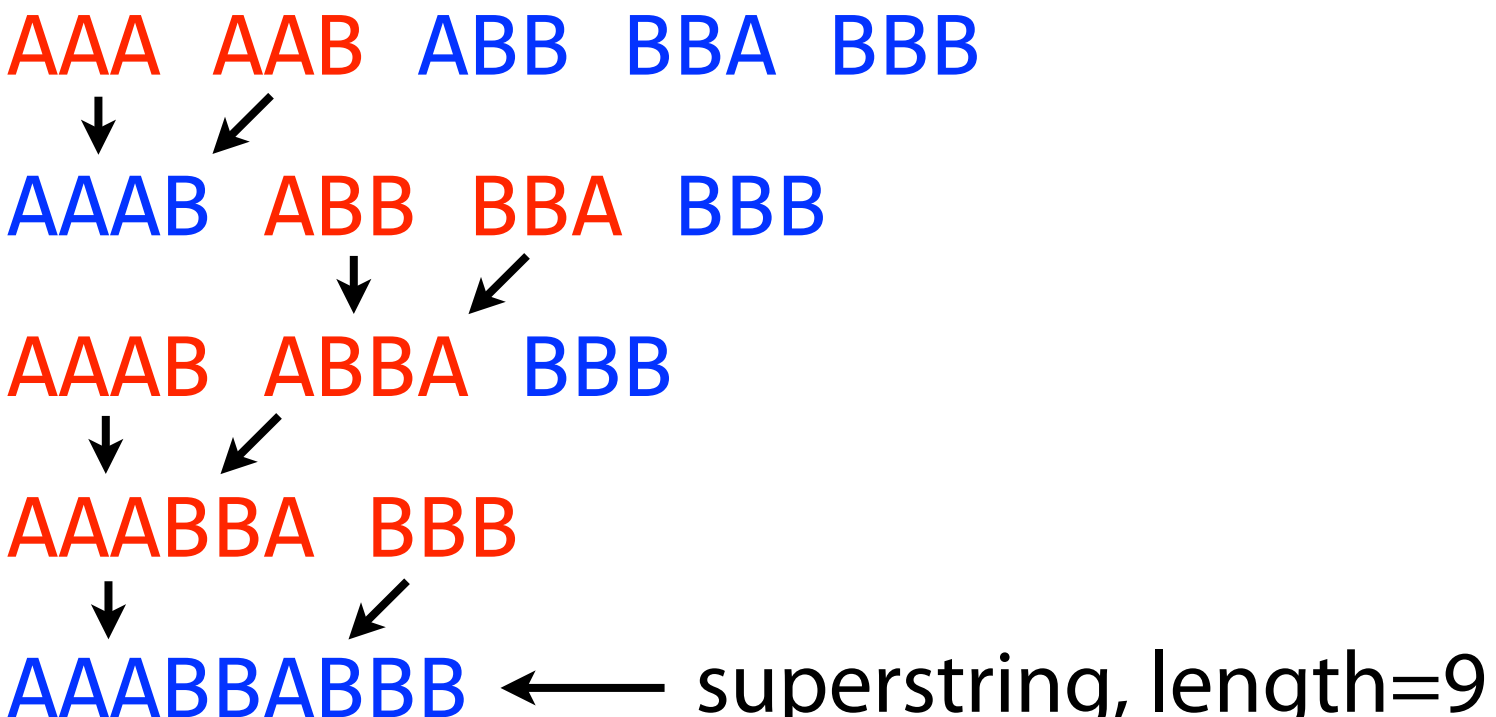
Algorithm in action ($l = 1$):

⟵ Input strings ⟶

AAA AAB ABB BBB BBA
AAAB ABB BBB BBA
AAAB BBBA ABB
AAABB BBBA
AAABBBA

AAABBBA

SCS: Greedy



AAABBBA ⟵ superstring, length=7

Problem 1: Greedy answer *isn't necessarily optimal*

SCS: Greedy

Greedy-SCS assembling all substrings of length $k = 6$ from:
a_long_long_long_time. $l = 3$.

```
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
```

What happened?

SCS: Greedy

Same example, but increased the substring length, *k*, from 6 to 8

```
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
```

Got the whole thing: `a_long_long_long_time`

Why is this different?

SCS: Greedy

Why are substrings of length 8 long enough for Greedy-SCS to figure out there are 3 copies of `long`?

SCS: Greedy

Why are substrings of length 8 long enough for Greedy-SCS to figure out there are 3 copies of `long`?

```
a_long_long_long_time
```

```
g_long_l
```

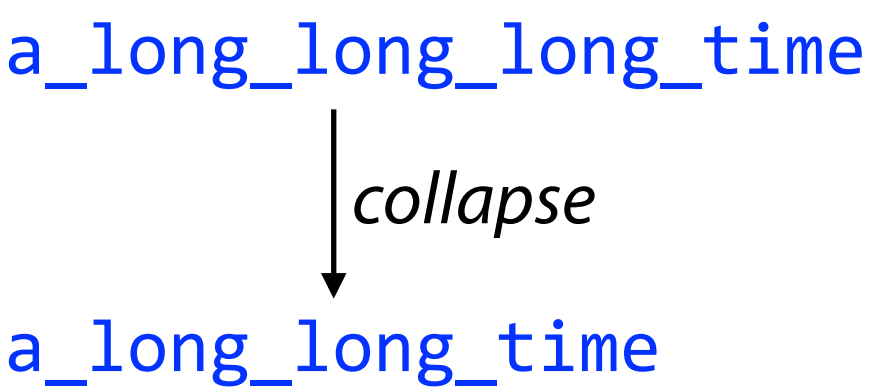
One length-8 substring spans all three `long`s

SCS: Greedy

Problem 2: *repeats foil assembly*

SCS can't handle repeats at all (the 'shortest' is not the best)!

More generally, algorithms that aren't very careful about repeats may *collapse* them



SCS: Greedy

Problem 2: *repeats foil assembly*

Solution: Identify repeats and **ignore** them!

Build ***contigs*** — contiguous fragments we can solve

`a_long` `long_time`

Fun trivia: This is particularly bad for genomics. The human genome is ~50% repetitive!

String Assembly

Input: A set of strings $S = \{s_1, s_2, \ldots, s_n\}$ assumed to be substrings of some underlying text *T*

Output: The 'best' approximation of *T*

1) Identify all possible overlaps

Solved with suffix tree / dynamic programming!

2) "Assemble" the best possible layout

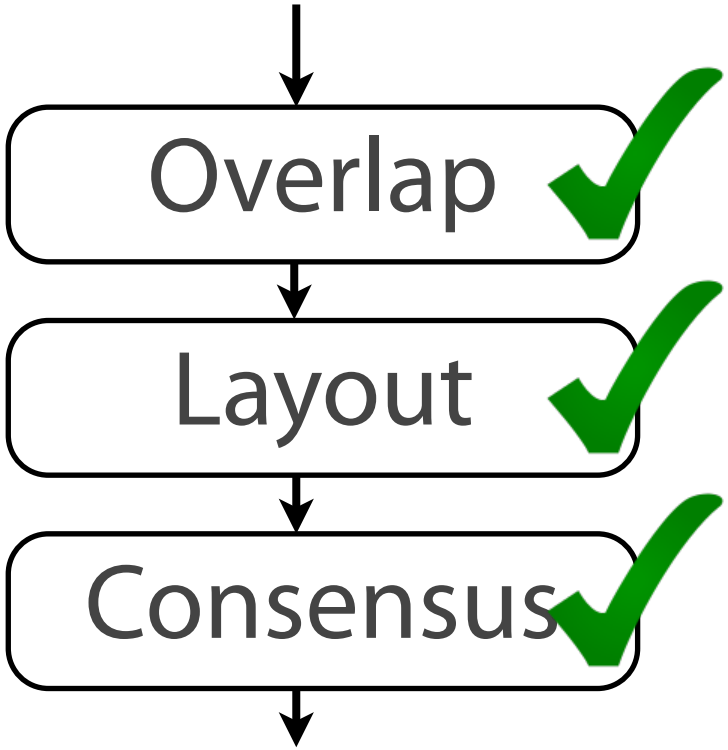
NP-Hard! Heuristics have trouble with repeats

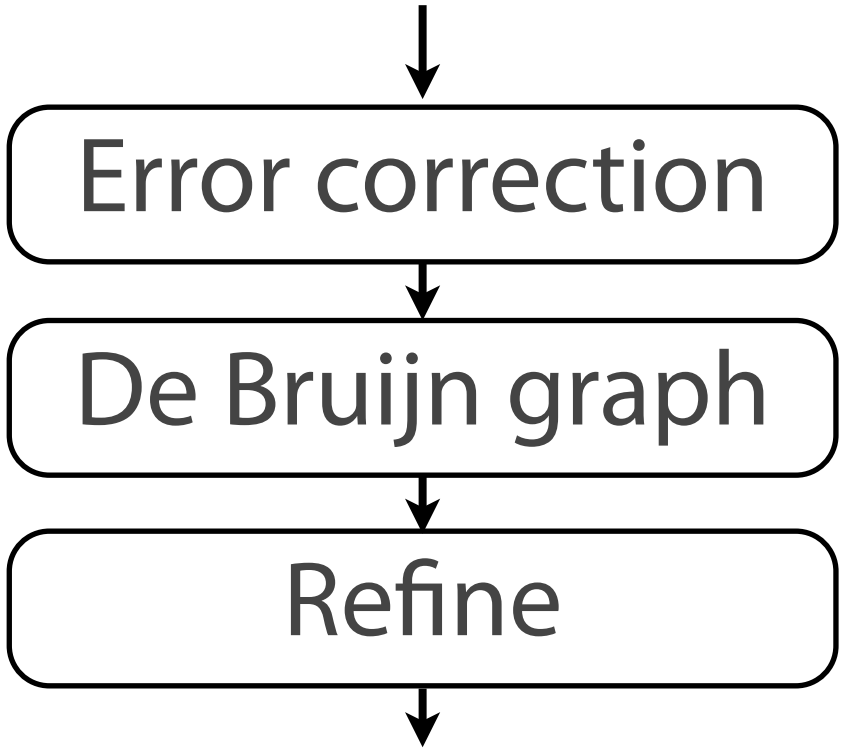
3) Reconstruct *T* based on consensus

Build contigs over what we know for certain, ignore the rest

Assembly strategies

Two competing approaches using **graphs**!



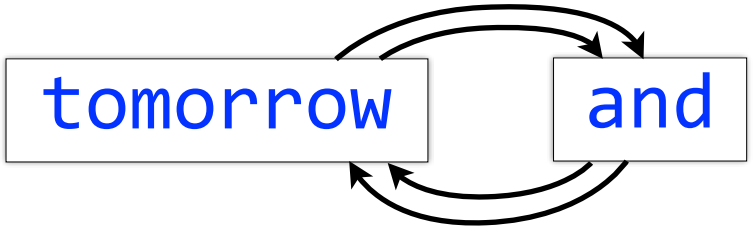


Alternative Graph Solution

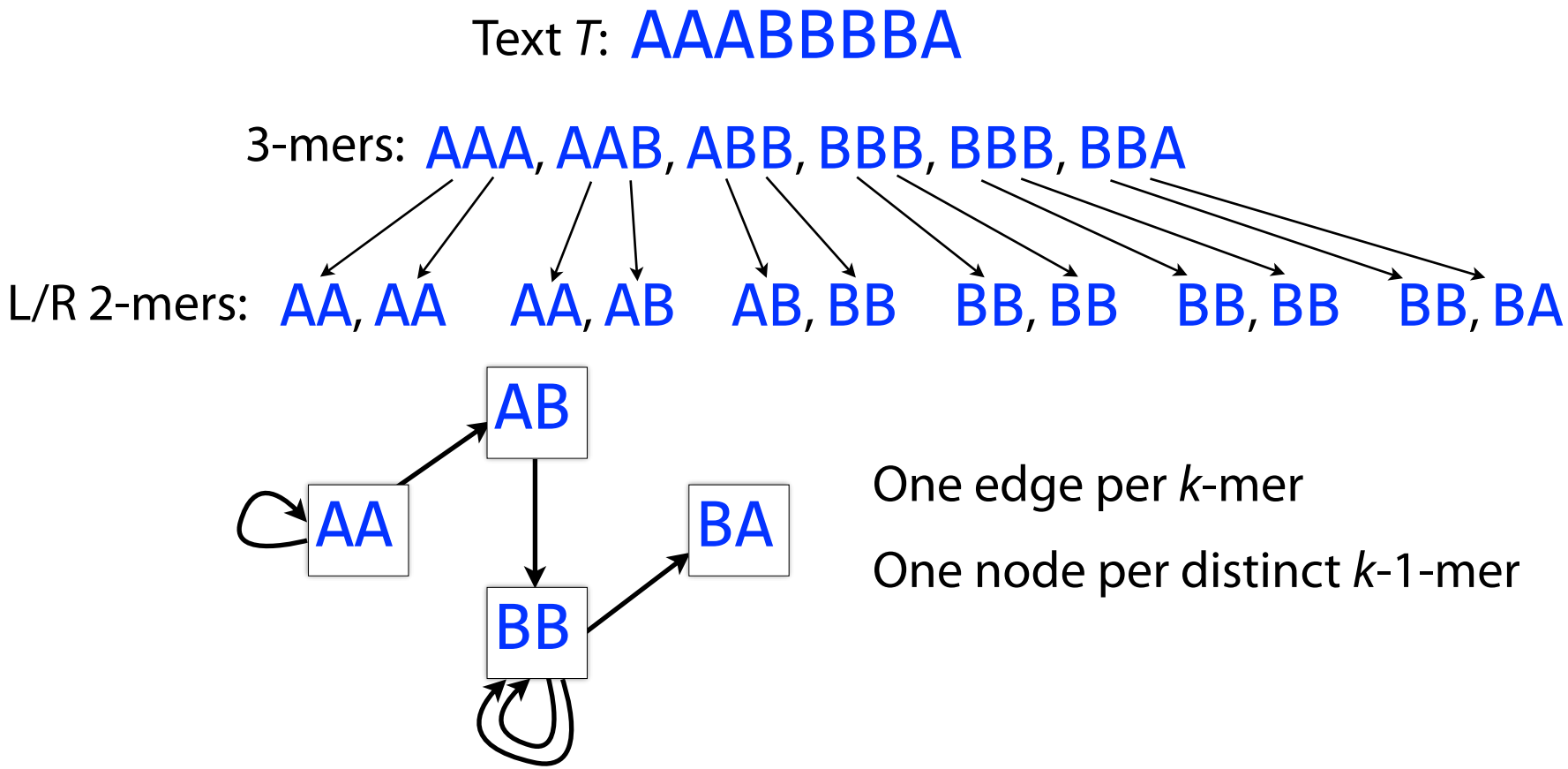
If ignoring the problem bothers you, there's another class of graphs…

This graph class keeps track of the **number of repeats!**

"`tomorrow and tomorrow and tomorrow`"



De Bruijn Graph

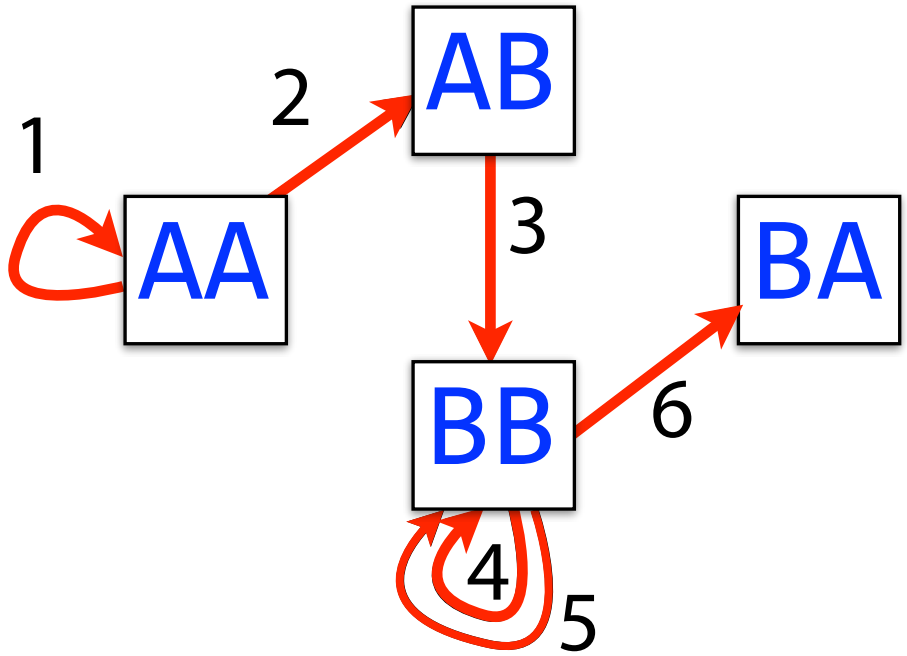


De Bruijn Graph

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

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

This is an *Eulerian walk*.



De Bruijn Graph

How much work to build graph for total length *N*?

For each *k*-mer, add 1 edge and up to 2 nodes

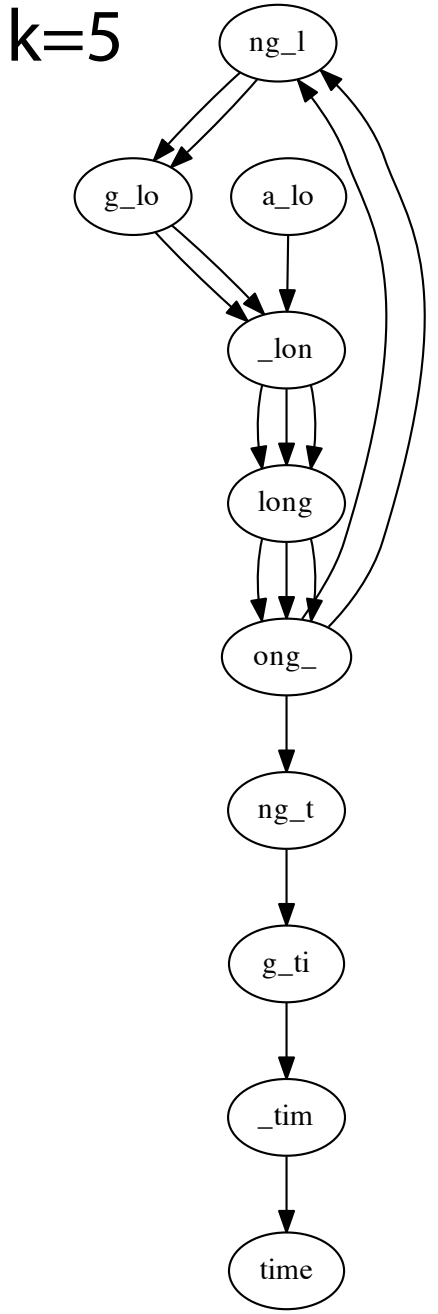
Reasonable to say this is O(1) expected work

Say hash map holds nodes & edges

Say *k*-1-mers fit in O(1) machine words, and hashing O(1) words is O(1) work

Querying / adding a key is O(1) expected work

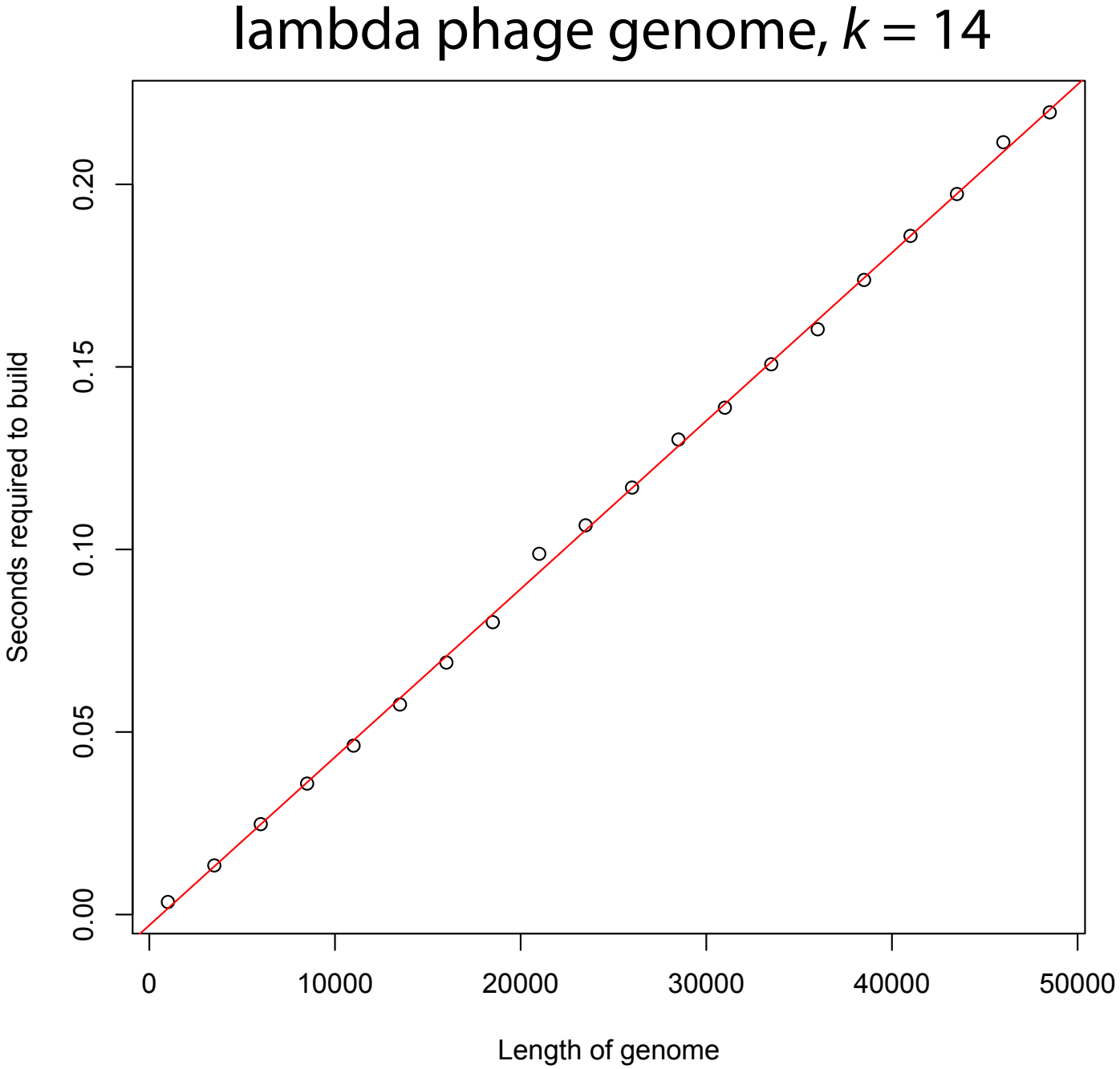
O(1) expected work for 1 *k*-mer, **O(*N*) overall**



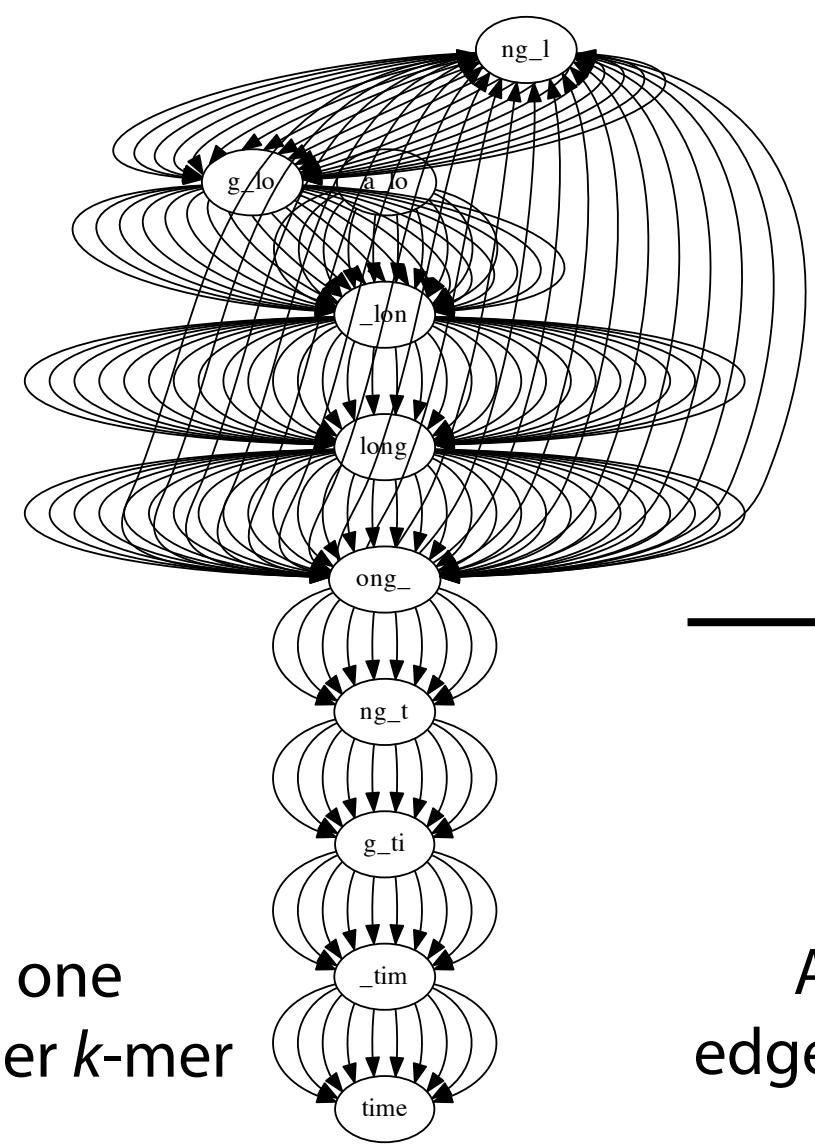
De Bruijn Graph

O(*N*) expectation works in practice

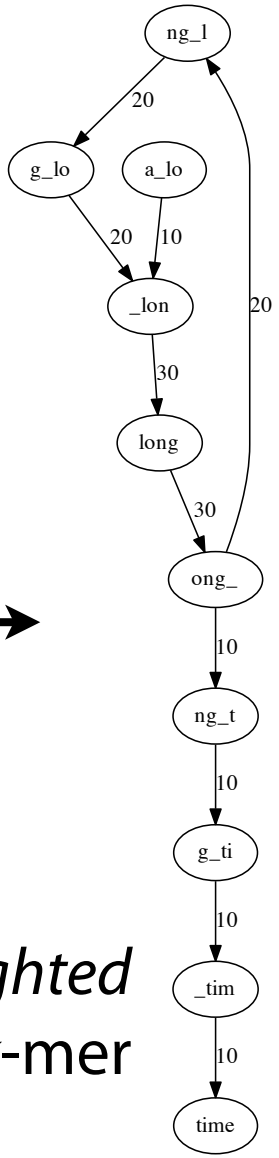
(in this case at least)



De Bruijn Graph as weighted graph



Before: one edge per *k*-mer



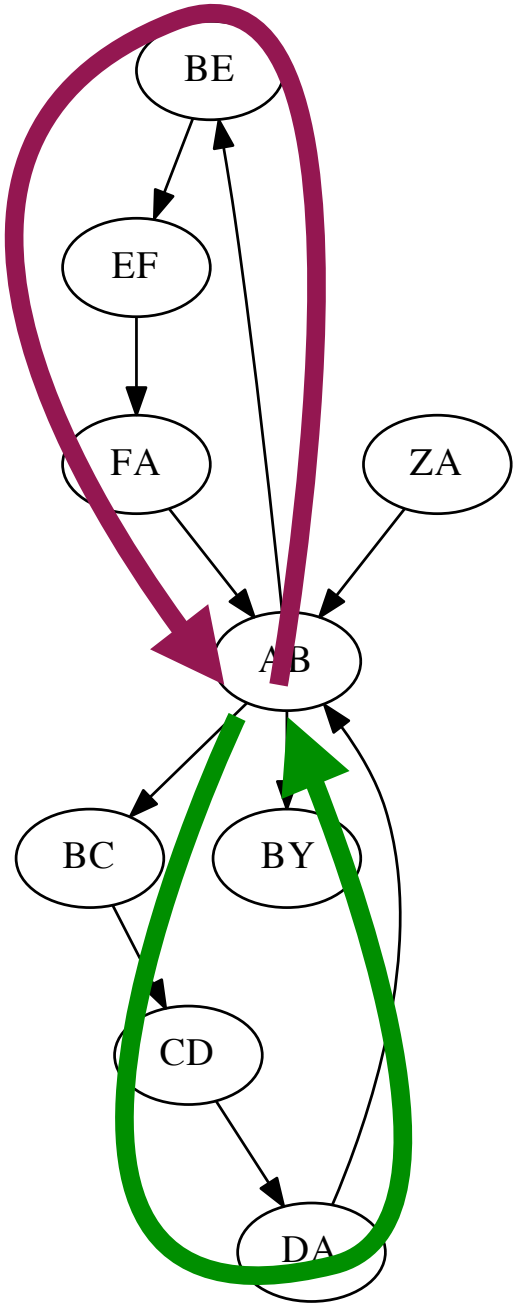
After: one *weighted* edge per *distinct k*-mer

De Bruijn Graph

Problem 1: Repeats can 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

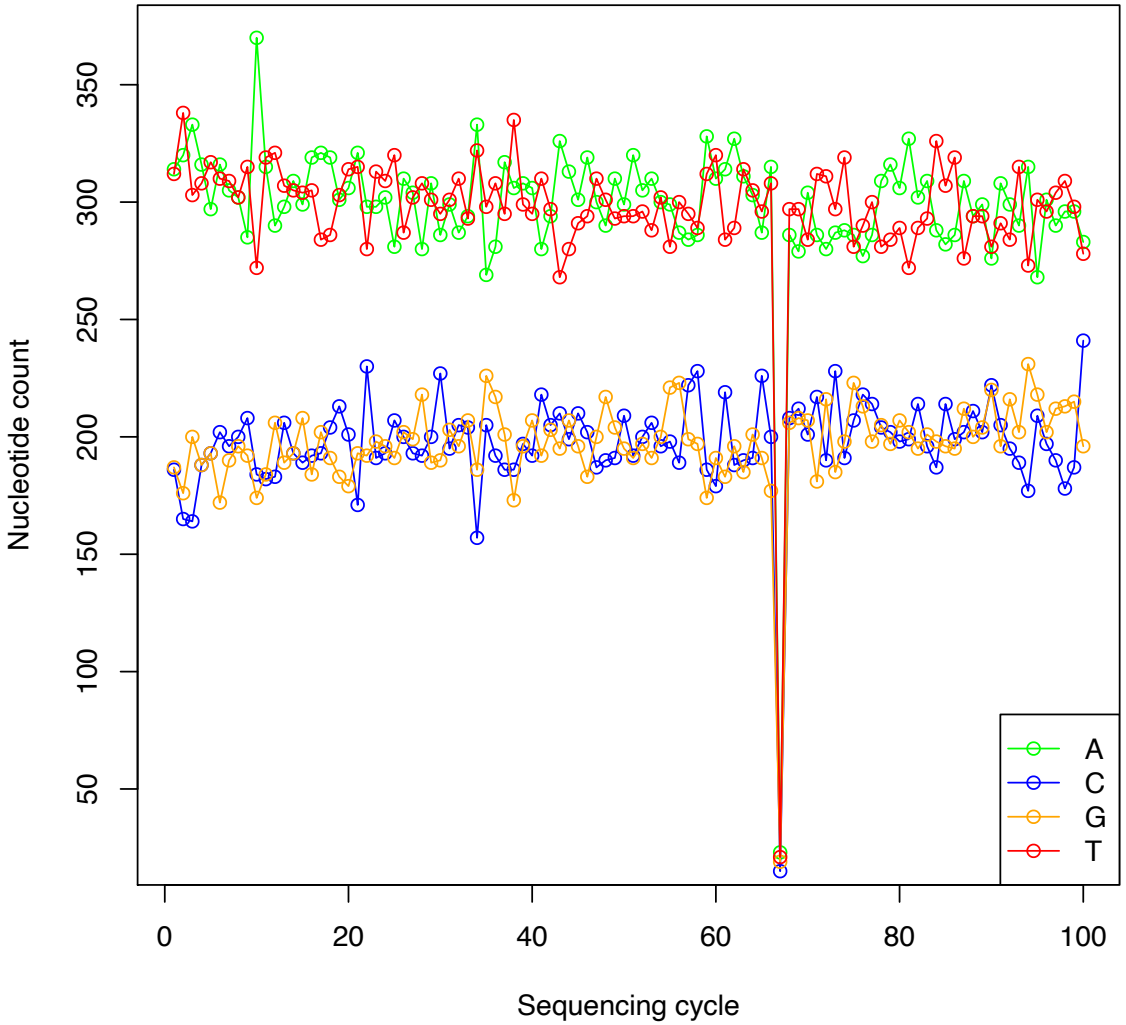


De Bruijn Graph

Problem 2: Data is not often 'perfect'!

We've been building DBGs assuming "perfect" data: each *k*-mer reported exactly once, no mistakes.

Real datasets aren't like that.

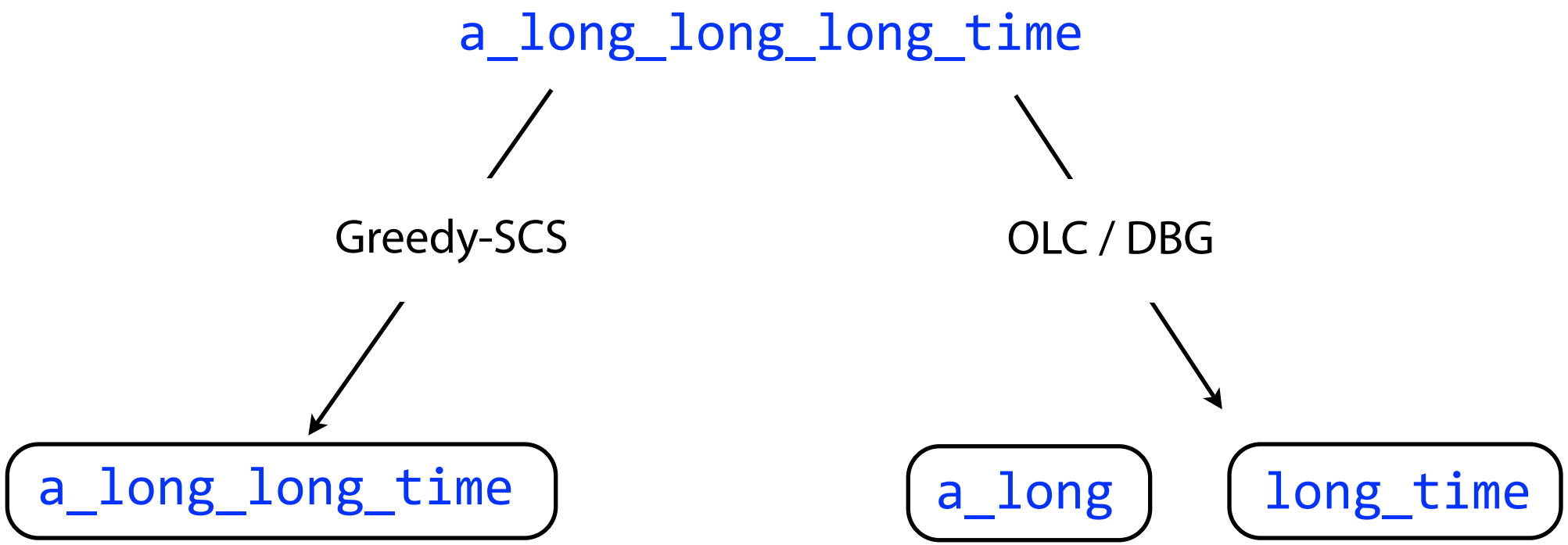


Solution to repeats: Ignore them!

Handle unresolvable repeats by *leaving them out*

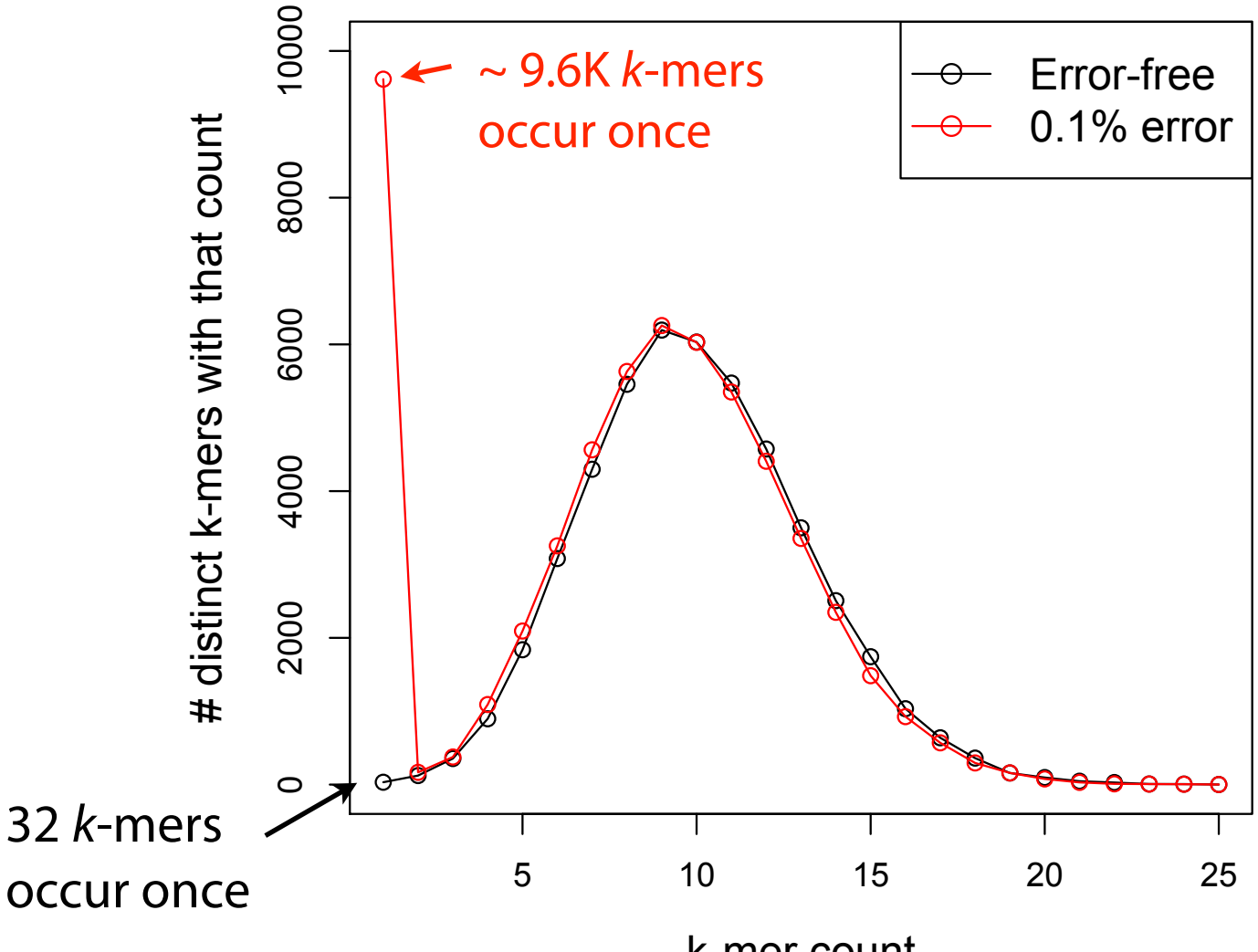
This breaks the assembly into fragments

Fragments called *contigs* (short for *contiguous*)



De Bruijn Graph Error Correction

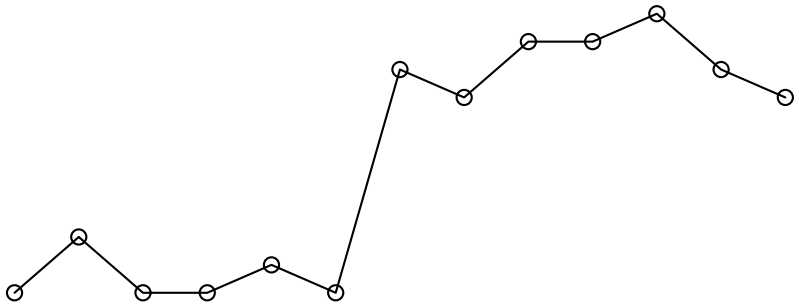
k-mers with errors occur fewer times than error-free *k*-mers



Single errors cause drop in kmer counts:

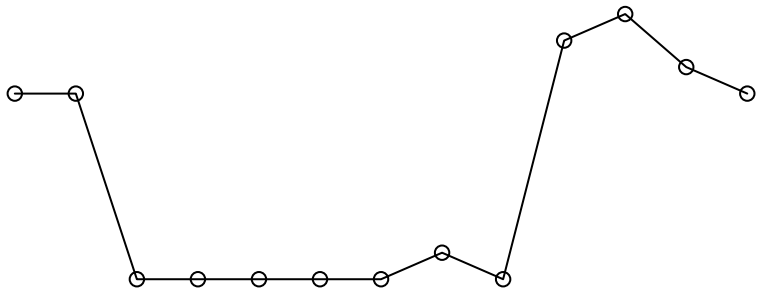
GCGTACTACGCGTCTGGCCT

GCGTACTA: 1
CGTACTAC: 3
GTACTACG: 1
TACTACGC: 1
ACTACGCG: 2
CTACGCGT: 1
TACGCGTC: 9
ACGCGTCT: 8
CGCGTCTG: 10
GCGTCTGG: 10
CGTCTGGC: 11
GTCTGGCC: 9
TCTGGCCT: 8



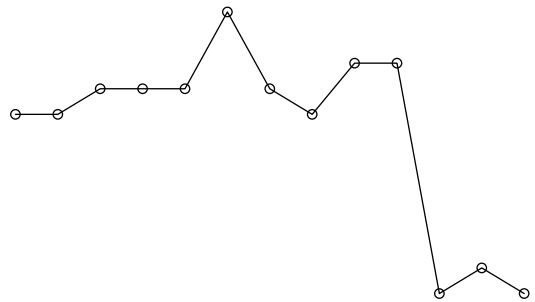
GCGTATTACACGTCTGGCCT

GCGTATTA: 8
CGTATTAC: 8
GTATTACA: 1
TATTACAC: 1
ATTACACG: 1
TTACACGT: 1
TACACGTC: 1
ACACGTCT: 2
CACGTCTG: 1
ACGTCTGG: 1
CGTCTGGC: 11
GTCTGGCC: 9
TCTGGCCT: 8



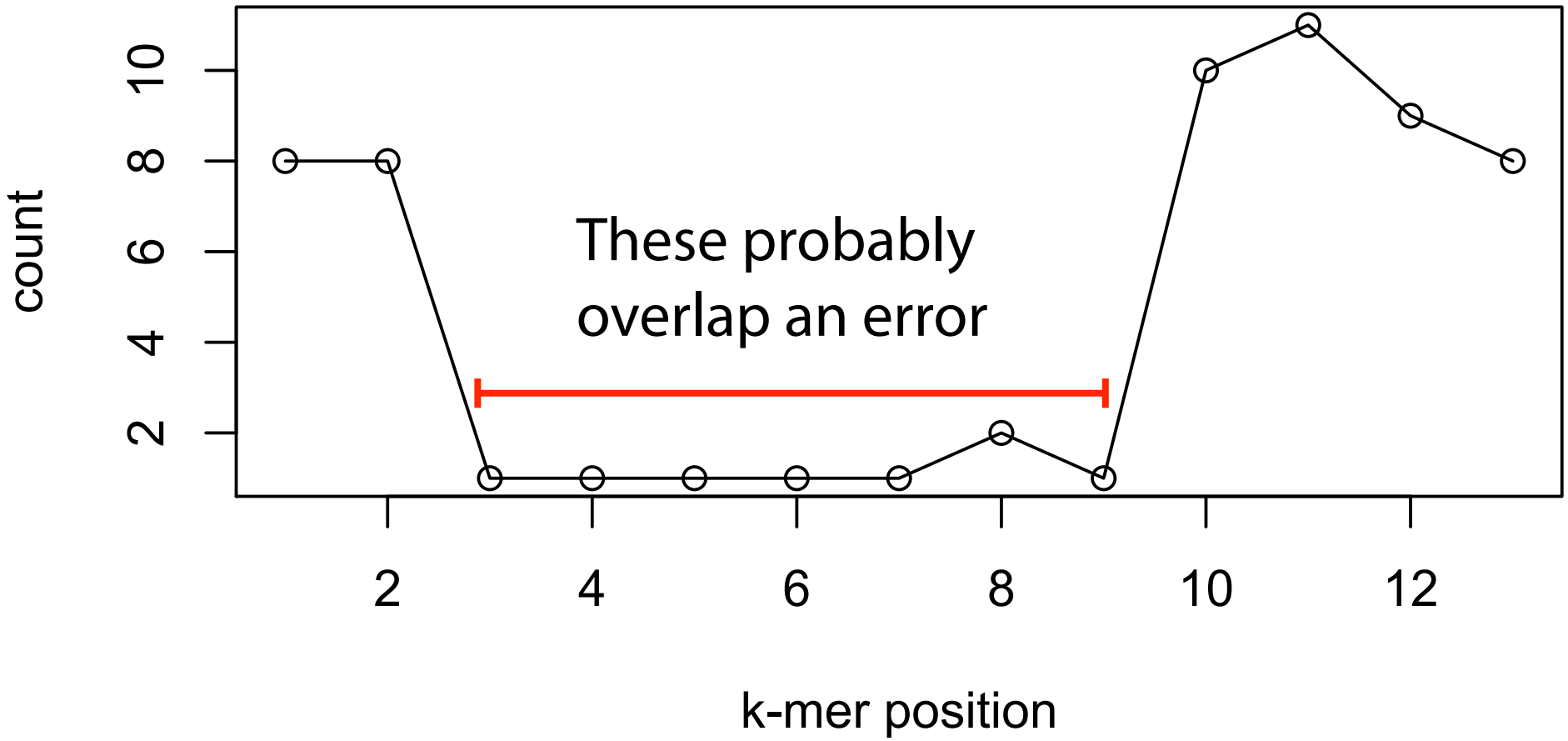
GCGTATTACGCGTCTGGTCT

GCGTATTA: 8
CGTATTAC: 8
GTATTACG: 9
TATTACGC: 9
ATTACGCG: 9
TTACGCGT: 12
TACGCGTC: 9
ACGCGTCT: 8
CGCGTCTG: 10
GCGTCTGG: 10
CGTCTGGT: 1
GTCTGGTC: 2
TCTGGTCT: 1



De Bruijn Graph Error Correction

Count profile indicates where errors are



De Bruijn Error Correction

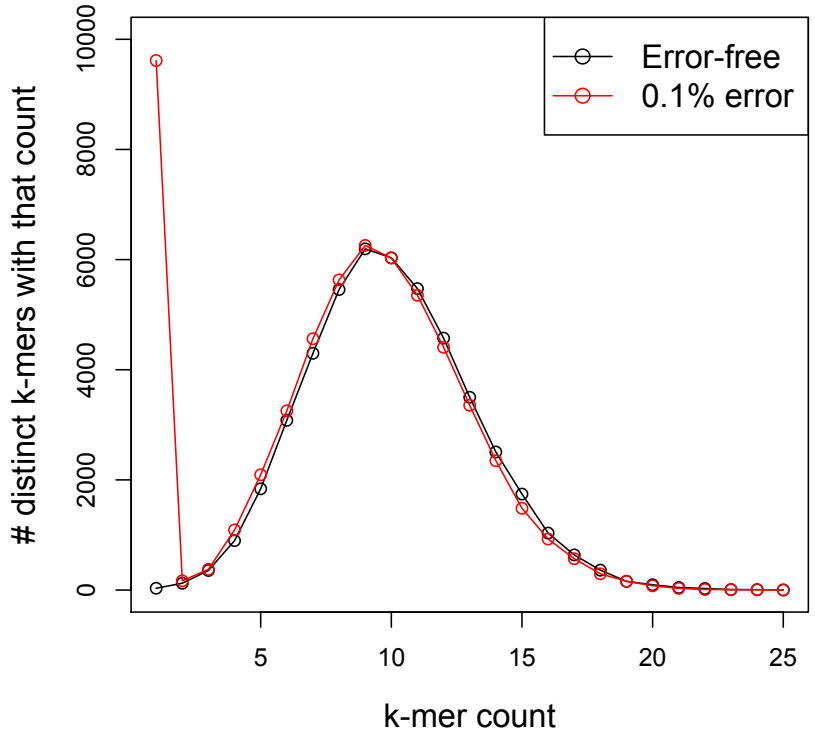
Simple algorithm, given a count threshold *t*:

For each read:

For each k-mer:

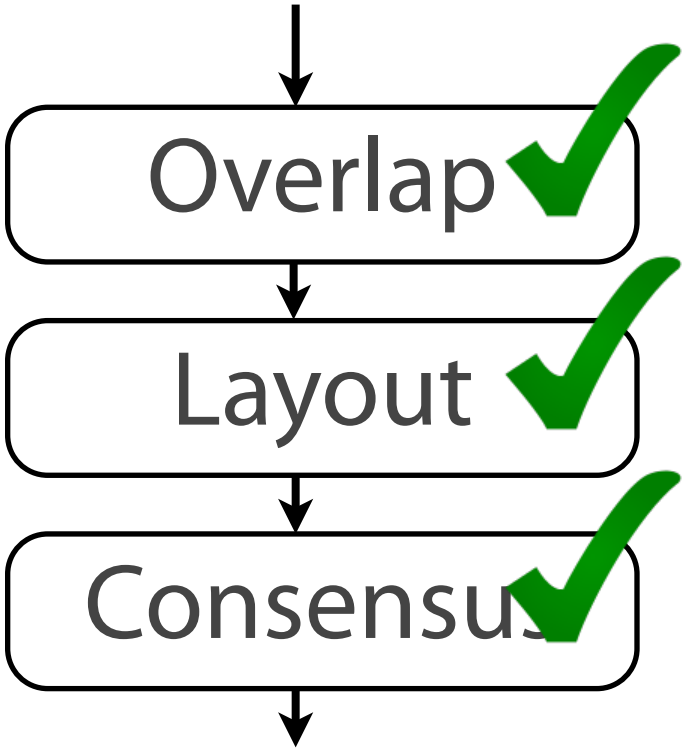
If *k*-mer count < *t*:

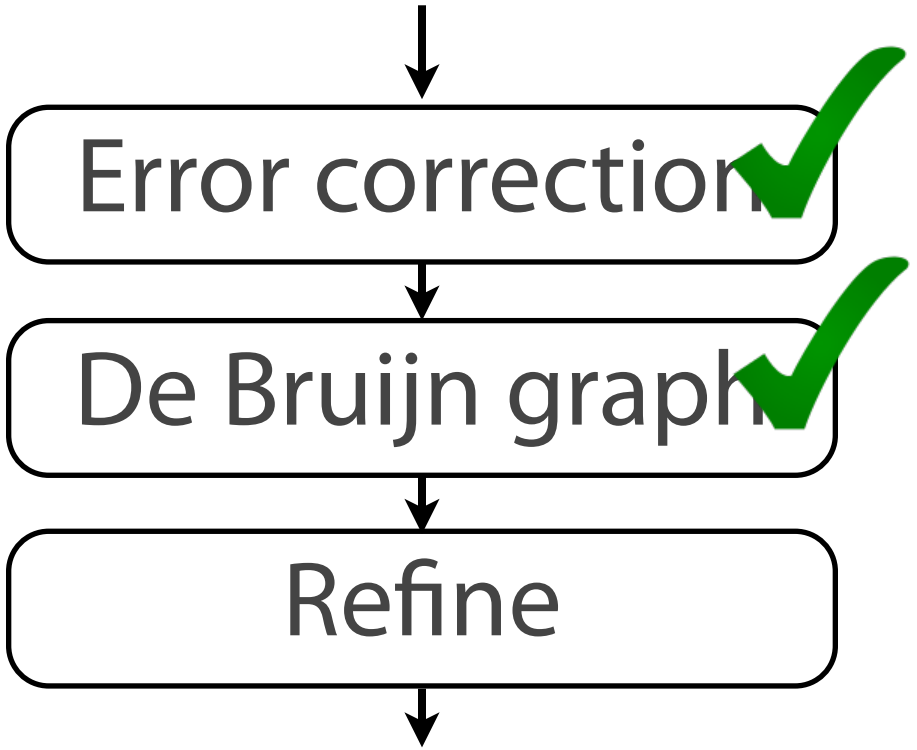
Examine *k*-mer's neighbors within some Hamming/edit distance.
If neighbor has count ≥ *t*, replace old *k*-mer with neighbor.



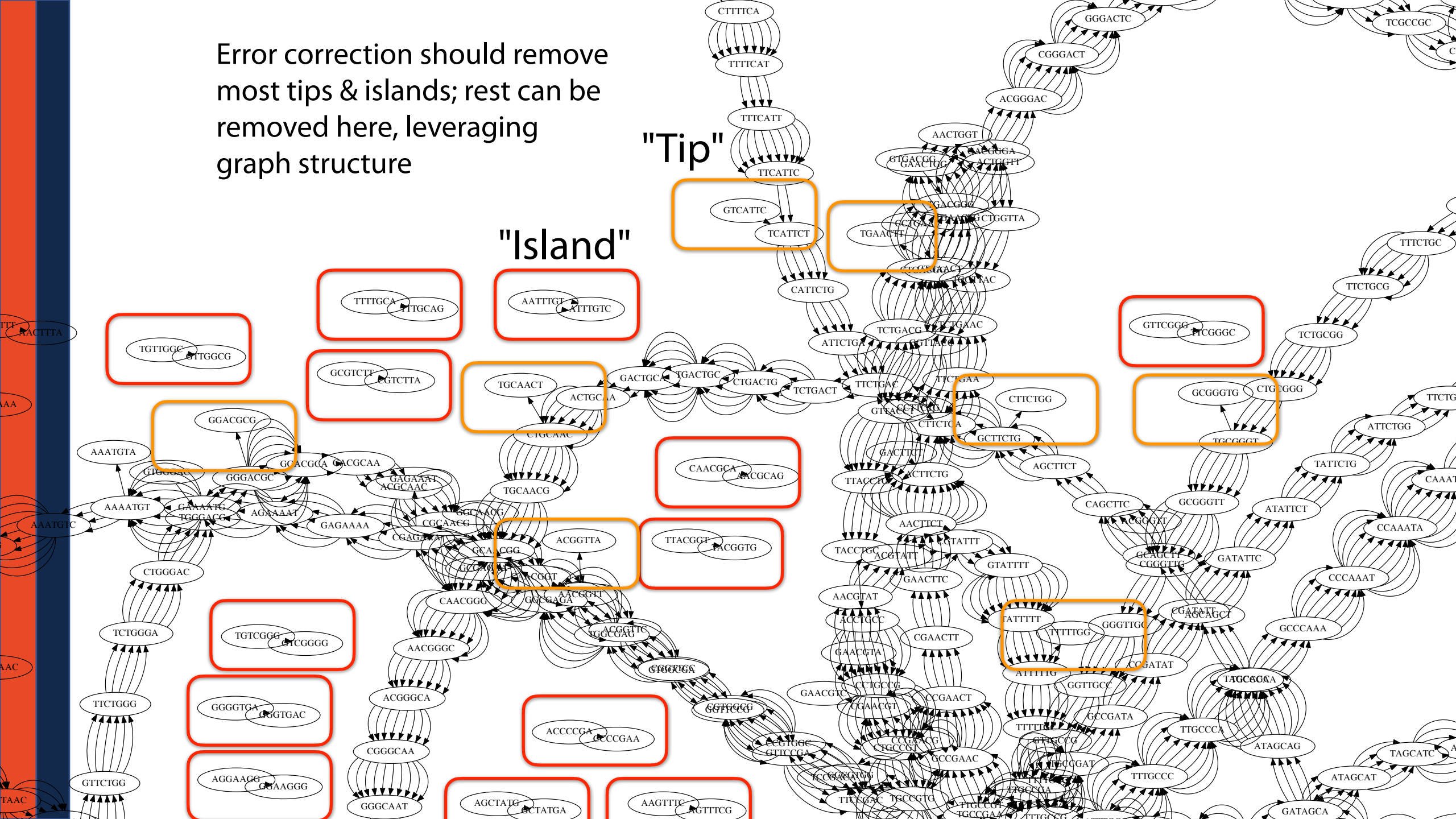
Pick *t* based on the expectation of errors (usually 1)

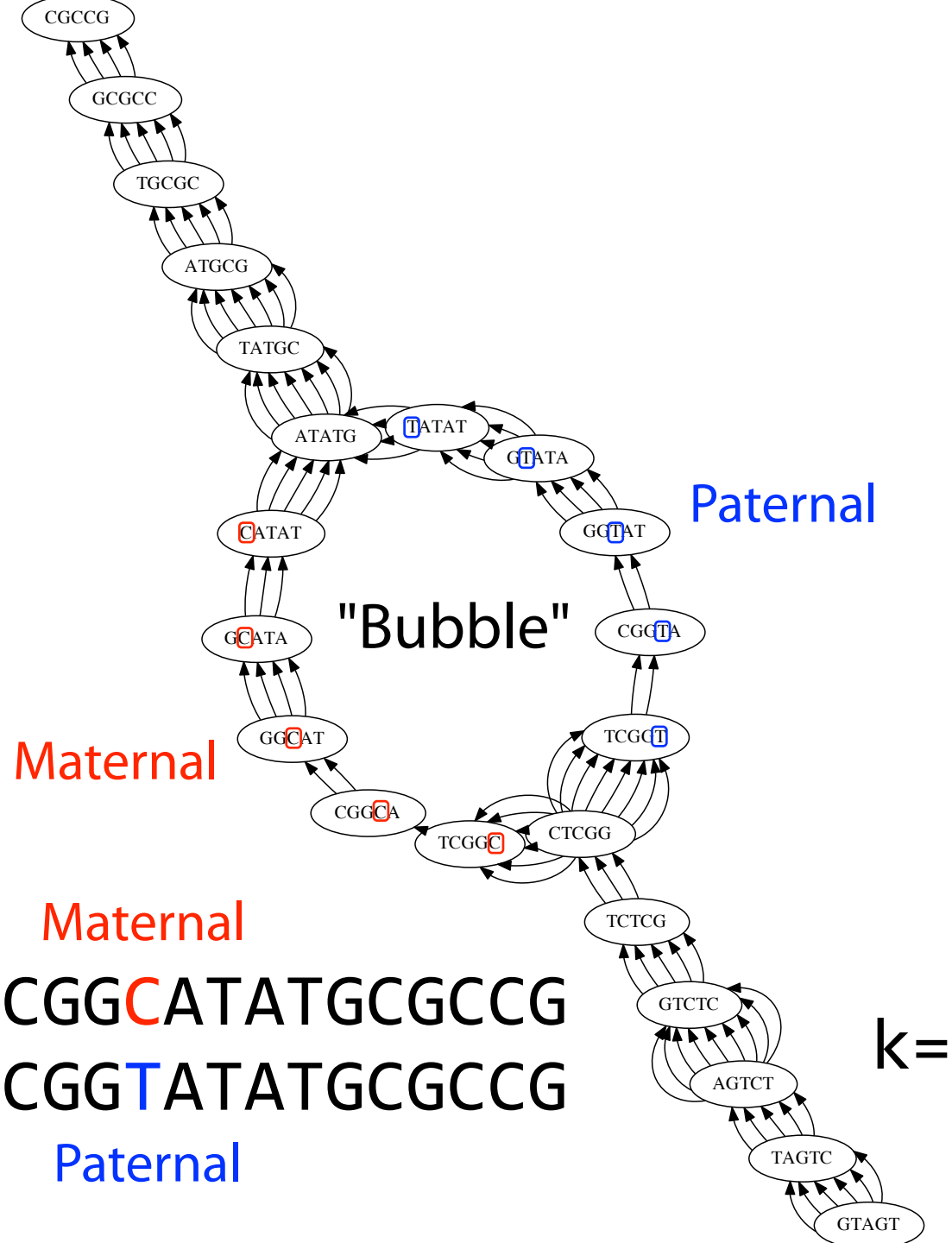
Assembly strategies





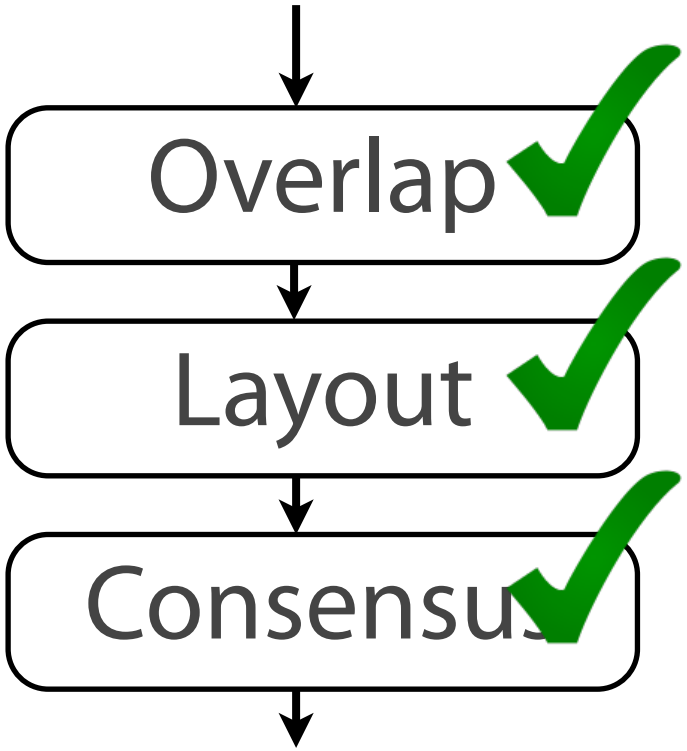
Error correction should remove most tips & islands; rest can be removed here, leveraging graph structure

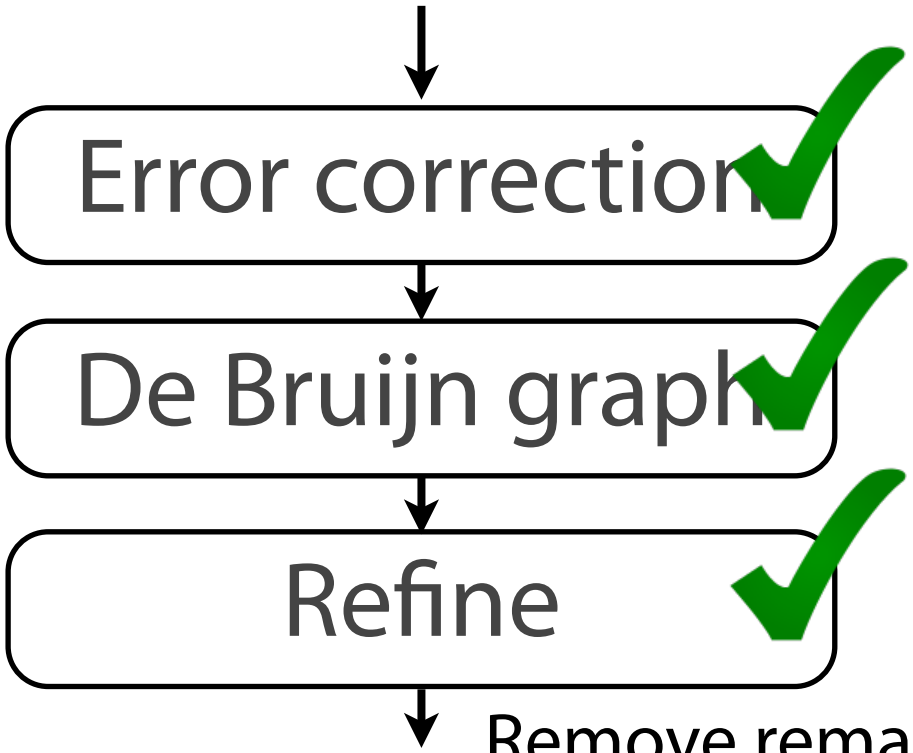



CGCCG
GCGCC
TGCGC
ATGCG
TATGC
ATATG
TATAT
GTATA
CATAT
GCTAT
GCATA
CGCTA
GGCAT
TCGCT
CGGCA
TCGGC
CTCGG
TCTCG
GTCTC
AGTCT
TAGTC
GTAGT
"Bubble"
Paternal
Maternal
Maternal
GTAGTCTCGGCATATGCGCCG
GTAGTCTCGGTATATGCGCCG
Paternal
k=6

Assembly strategies

Both produce *contigs* — more work needed to get *T*!





Remove remaining "islands" “tips” and “bubbles” so that contig are more obvious

This concludes CS 199-225!

Material Covered:

Exact Search: Z-algorithm Boyer-Moore

Suffix Trie Suffix Tree Suffix Array

BWT FM Index

Inexact Search: Pigeonhole Principle

Edit Distance FM Index

String Assembly: OLC Assembly / De Bruijn Graph

Machine Learning: Markov Chain / Hidden Markov Models

This concludes CS 199-225!

Learning Objectives:

Understand fundamental string algorithms

Experience applying data structures, algorithms, and algorithm design principles to real world problems

Justify implementation choices based on theoretical or practical considerations

Build a foundation for future data science projects