Computational Approaches to Biological Challenges (algorithmic primer)

Shortest Common Superstring & Lander-Waterman Statistics

UNIVERSITY OF MARYLAND

What is Computer Science?

Not actually simple to define constructively

Still debate whether certain areas constitute CS

Computer science is the scientific and practical approach to computation and its applications. It is the systematic study of the feasibility, structure, expression, and mechanization of the methodical procedures (or algorithms) that underlie the acquisition, representation, processing, storage, communication of, and access to information* …

*http://www.cs.bu.edu/AboutCS/WhatIsCS.pdf

What is Computer Science?

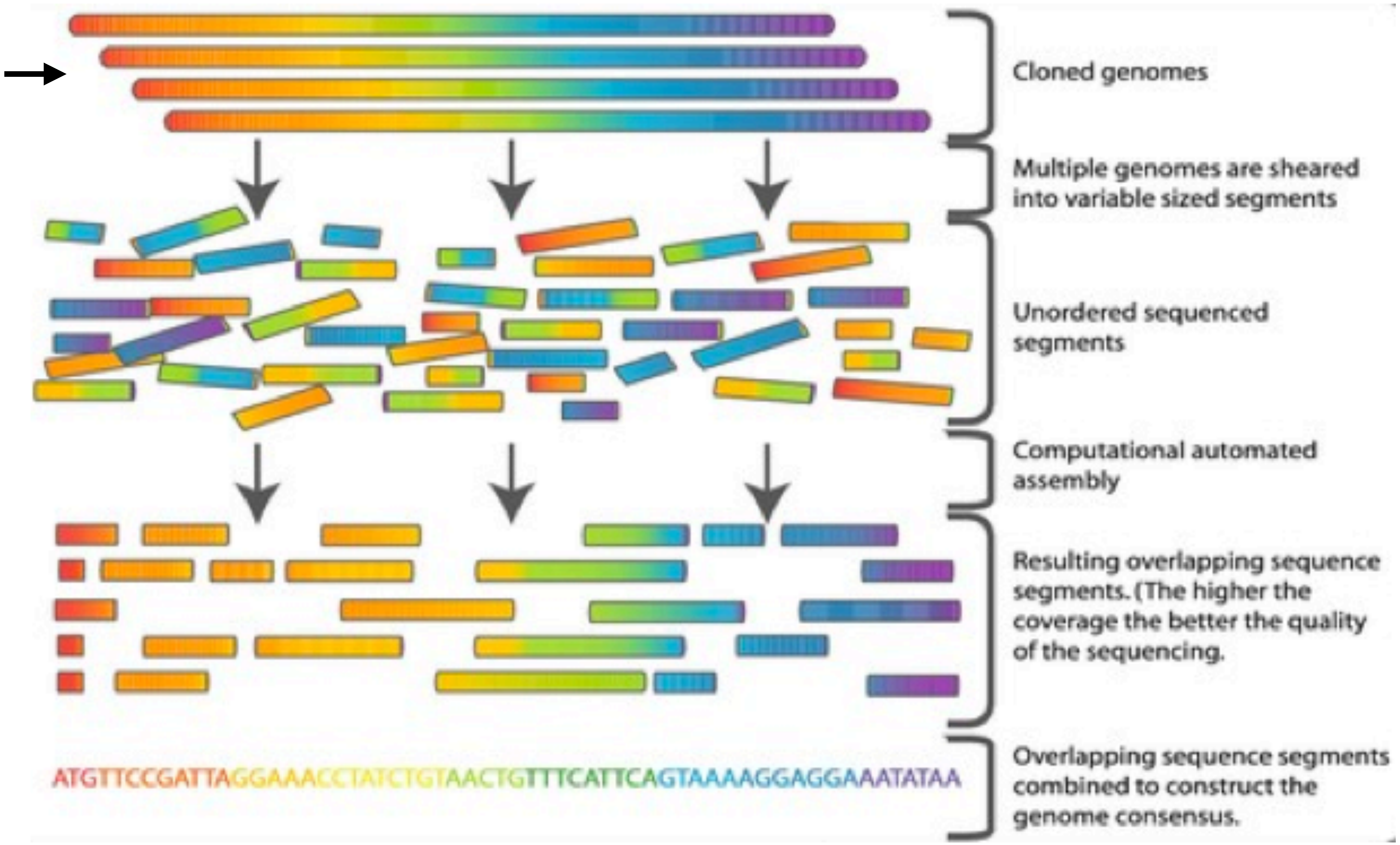
Concerned with the development of provably **correct** and **efficient** computational procedures (algorithms & data structures) to answer **well-specified** problems.

To answer a computational question, we first need a well-formulated problem.

It turns out that a **major challenge** in bioinformatics will simply be determining how to frame the *computational problem* corresponding to a *biological question* in a well-posed and meaningful way!

What is genome assembly: intuitively?

Why start with
many cloned genomes →
and not just one?



Cloned genomes

Multiple genomes are sheared into variable sized segments

Unordered sequenced segments

Computational automated assembly

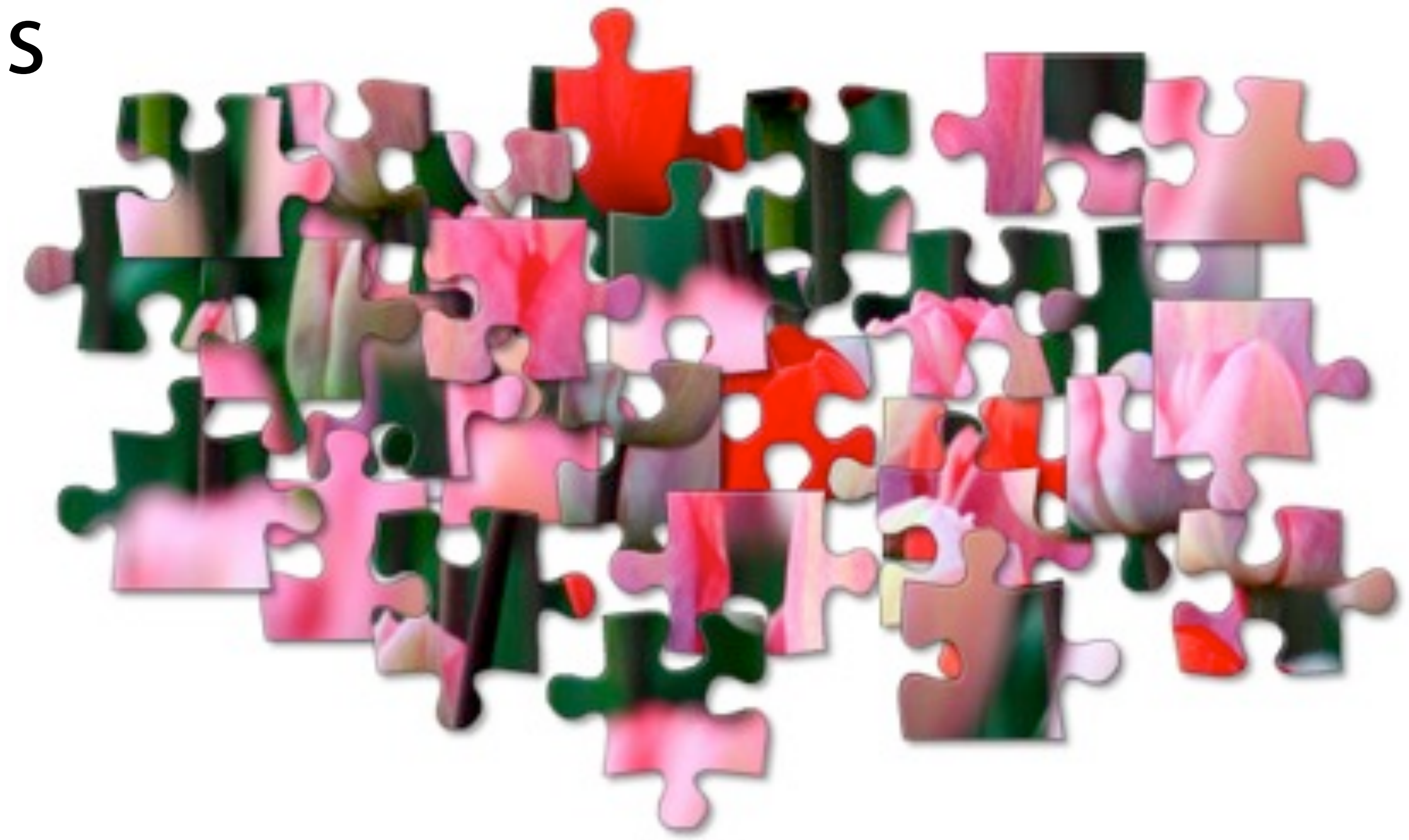
Resulting overlapping sequence segments. (The higher the coverage the better the quality of the sequencing.

Overlapping sequence segments combined to construct the genome consensus.

https://en.wikipedia.org/wiki/Shotgun_sequencing

Assembly

Reads



+

Reference genome

How to assemble puzzle without the benefit of knowing what the finished product looks like?

Input DNA

Next 5 slides courtesy of Ben Langmead

Assembly

Whole-genome "shotgun" sequencing starts by copying and fragmenting the DNA

("Shotgun" refers to the random fragmentation of the whole genome; like it was fired from a shotgun)

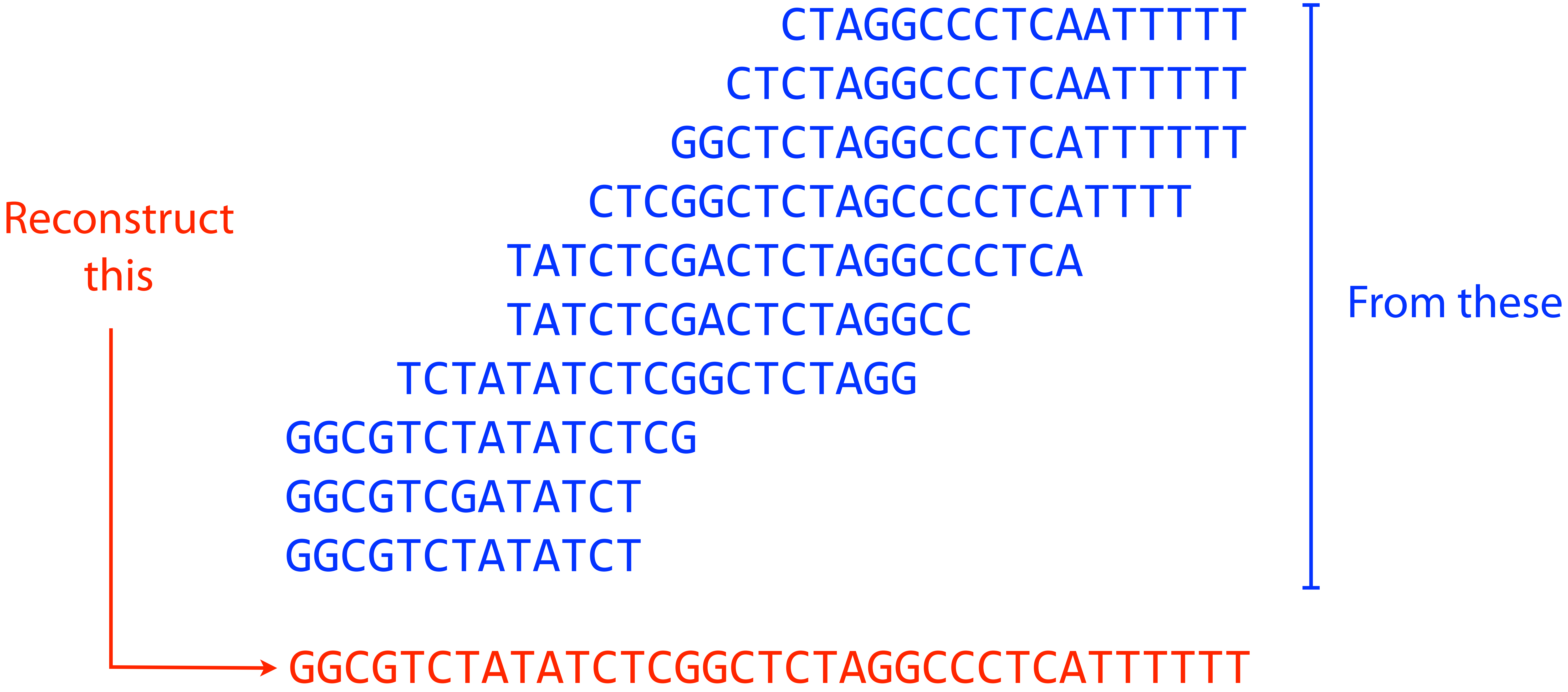
Input: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Copy: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

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

Assembly

Assume sequencing produces such a large # fragments that almost all genome positions are *covered* by many fragments...



CTAGGCCCTCAATTTTT
CTCTAGGCCCTCAATTTTT
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT

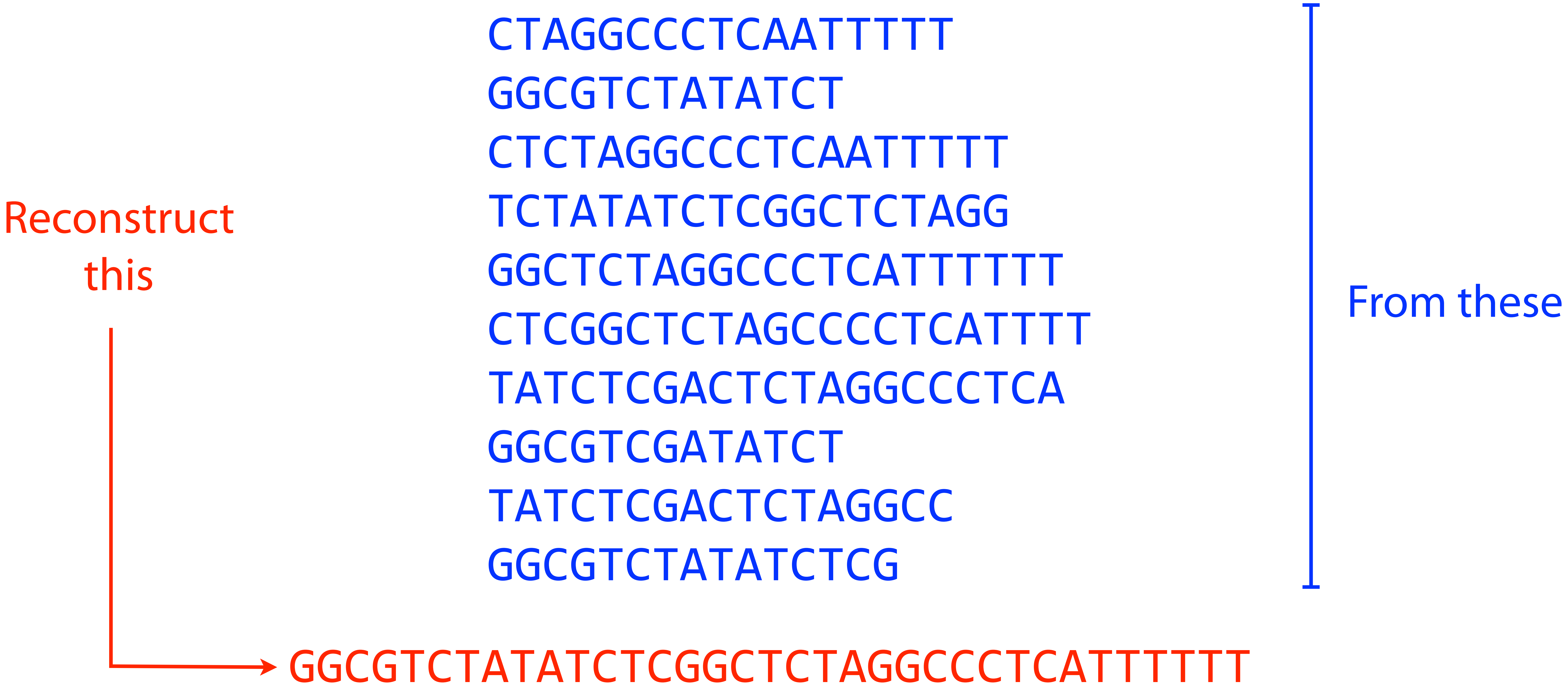
From these

Reconstruct this

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Assembly

...but we don't know what came from where



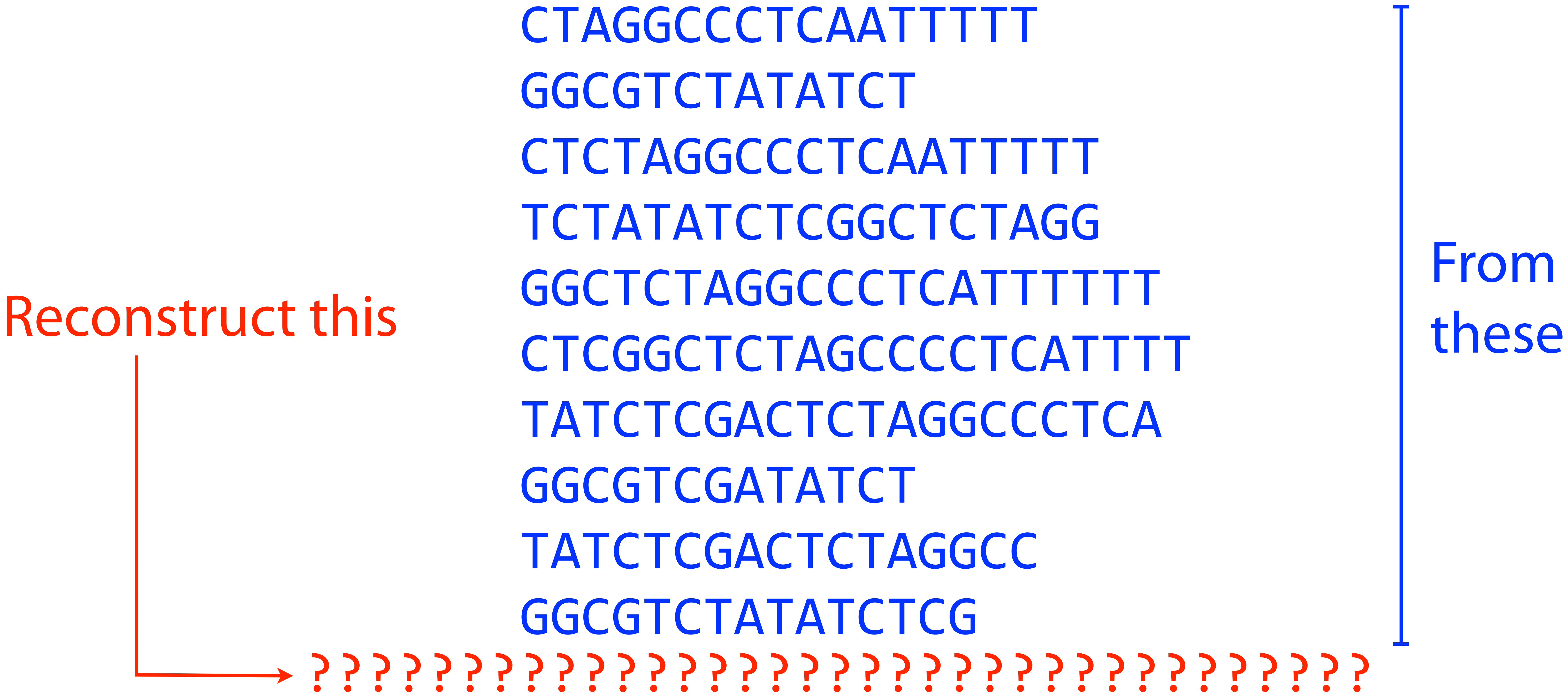
CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

From these

Reconstruct this

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Assembly



Reconstruct this

CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

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

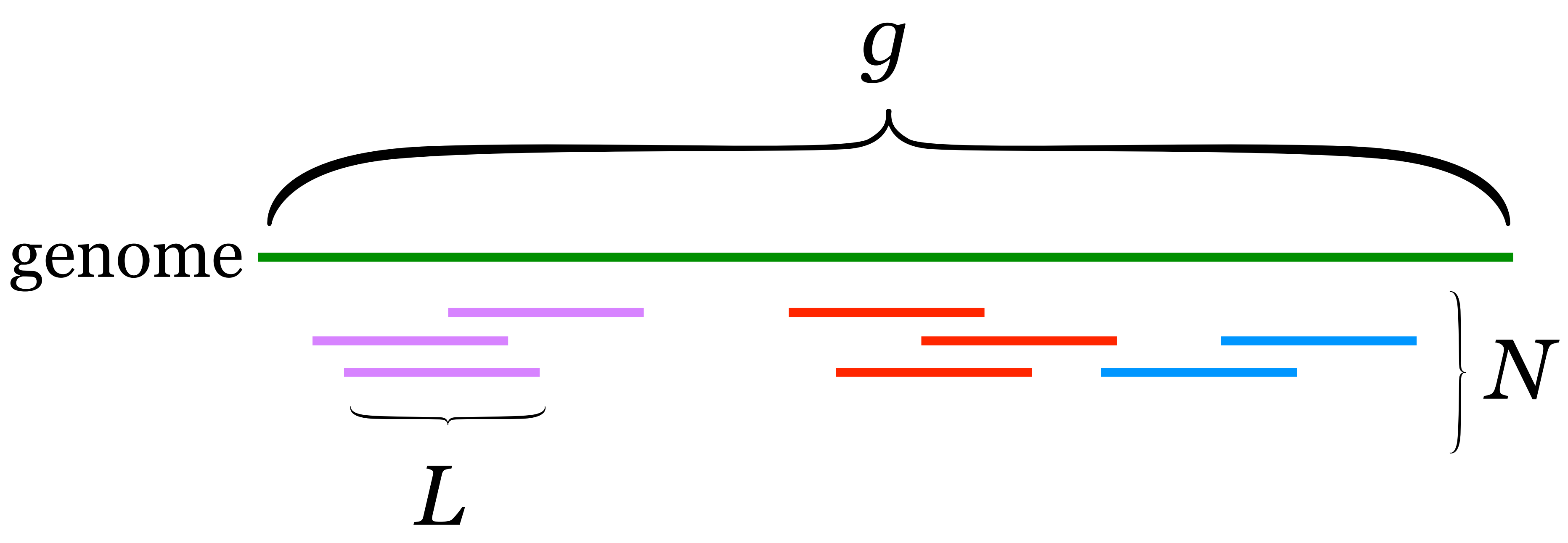
From these

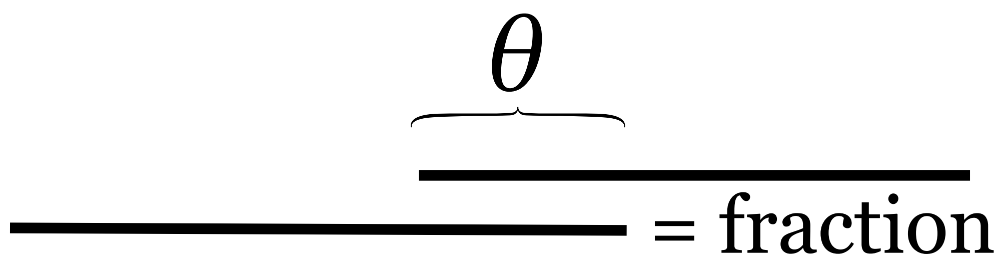
In general: we don't even know exactly how *long* the original string was!

Aside: How Much Coverage is Enough? Lander-Waterman Statistics

Lander ES, Waterman MS (1988). "Genomic mapping by fingerprinting random clones: a mathematical analysis". Genomics 2 (3): 231–239

How many reads to we need to be sure we cover the whole genome?





= fraction of L required to detect an overlap

An ***island*** is a contiguous group of reads that are connected by overlaps of length ≥ θL.
(Various colors above)

Want: Expression for expected # of islands given N, g, L, θ.

* Slide from Carl Kingsford

Expected # of Islands

$\lambda := N/g$ = probability a read starts at a given position (assuming random sampling)

Pr(k reads start in an interval of length x)

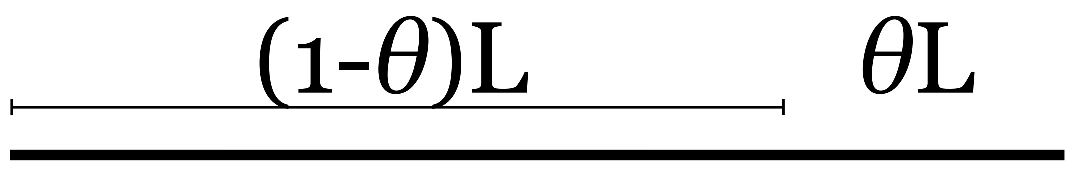
x trials, want k “successes”, small probability λ of success

Expected # of successes = λx

Poisson approximation to binomial distribution:

$$\Pr(k \text{ reads in length } x) = e^{-\lambda x}\frac{(\lambda x)^k}{k!}$$

Expected # of islands = $N \times$ Pr(read is at rightmost end of island)



= $N \times$ Pr(0 reads start in (1-θ)L)

$$= Ne^{-\lambda(1-\theta)L}\frac{\lambda^0}{0!} \quad \text{(from above)}$$

$$= Ne^{-\lambda(1-\theta)L}$$

$$= Ne^{-(1-\theta)LN/g}$$

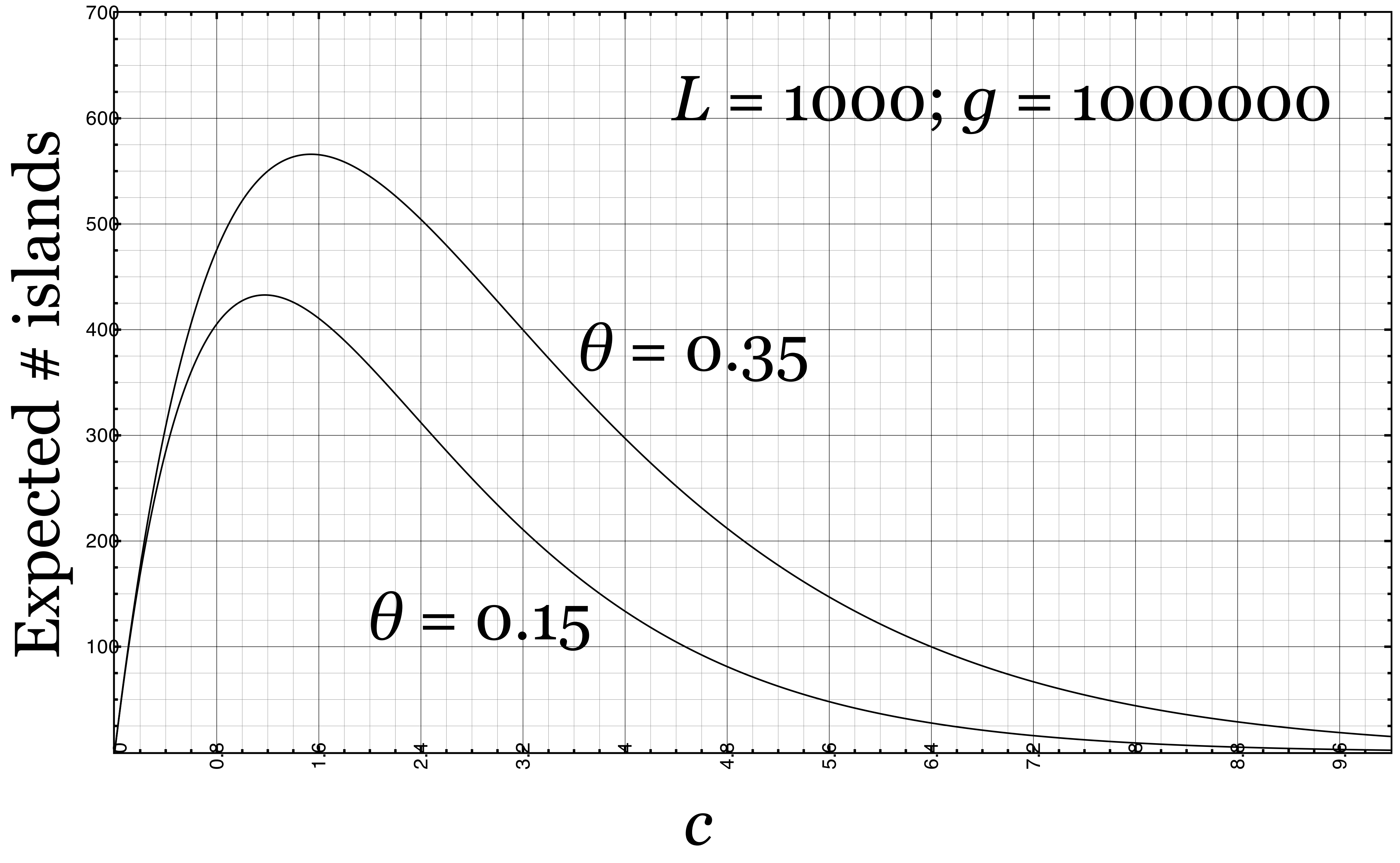
← LN/g is called the **coverage** c.

* Slide from Carl Kingsford

Expected # of Islands, 2

We can rewrite this expression to depend more directly on the things we can control: c and θ

$$
\begin{aligned}
\text{Expected \# of islands} &= Ne^{-(1-\theta)LN/g} \\
&= Ne^{-(1-\theta)c} \\
&= \frac{L/g}{L/g} Ne^{-(1-\theta)c} \\
&= \frac{g}{L} ce^{-(1-\theta)c}
\end{aligned}
$$



* Slide from Carl Kingsford

Formulating *a* genome assembly problem

Concerned with the development of provably **correct** and **efficient** computational procedures (algorithms & data structures) to answer **well-specified** problems.

To answer a computational question, we first need a well-formulated problem.

Given: a collection, R, of sequencing reads (strings)

Find: The genome (string), G, that generated them

Formulating *a* genome assembly problem

Concerned with the development of provably **correct** and **efficient** computational procedures (algorithms & data structures) to answer **well-specified** problems.

To answer a computational question, we first need a well-formulated problem.

~~**Given**: a collection, R, of sequencing reads (strings)~~

~~**Find**: The genome (string), G, that generated them~~

Not well-specified.
What makes one genome more likely than another?
What constraints do we place on the space of solutions?

Formulating *a* genome assembly problem

Concerned with the development of provably **correct** and **efficient** computational procedures (algorithms & data structures) to answer **well-specified** problems.

To answer a computational question, we first need a well-formulated problem.

Given: a collection, R, of sequencing reads (strings) ✔

Find: The shortest genome (string), G, that contains all of them

Shortest Common Superstring

Given: a collection, $S = \{s_1, s_2, \ldots, s_k\}$, of sequencing reads (strings)

Find*: The shortest possible genome (string), G, such that $s_1, s_2, \ldots, s_k$ are all substrings of G

How, might we go about solving this problem?

*for reasons we'll explore later, this isn't actually a great formulation for genome assembly.

Shortest common superstring

Given a collection of strings *S*, find *SCS*(*S*): the shortest string that contains all strings in *S* as substrings

Without requirement of "shortest," it's easy: just concatenate them

Example: *S*: BAA AAB BBA ABA ABB BBB AAA BAB

Concatenation: BAAAABBBAABAABBBBBAAABAB

24

SCS(S): AAABBBABAA

10

```
AAA
 AAB
  ABB
   BBB
    BBA
     BAB
      ABA
       BAA
```

Slide courtesy of Ben Langmead

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

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAA

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

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

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAAB

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

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

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABA

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

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

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABB

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

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

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB ⟵ superstring 1

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

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

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

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

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

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring

S: AAA AAB ABB BBB BBA

SCS(S): AAABBBA

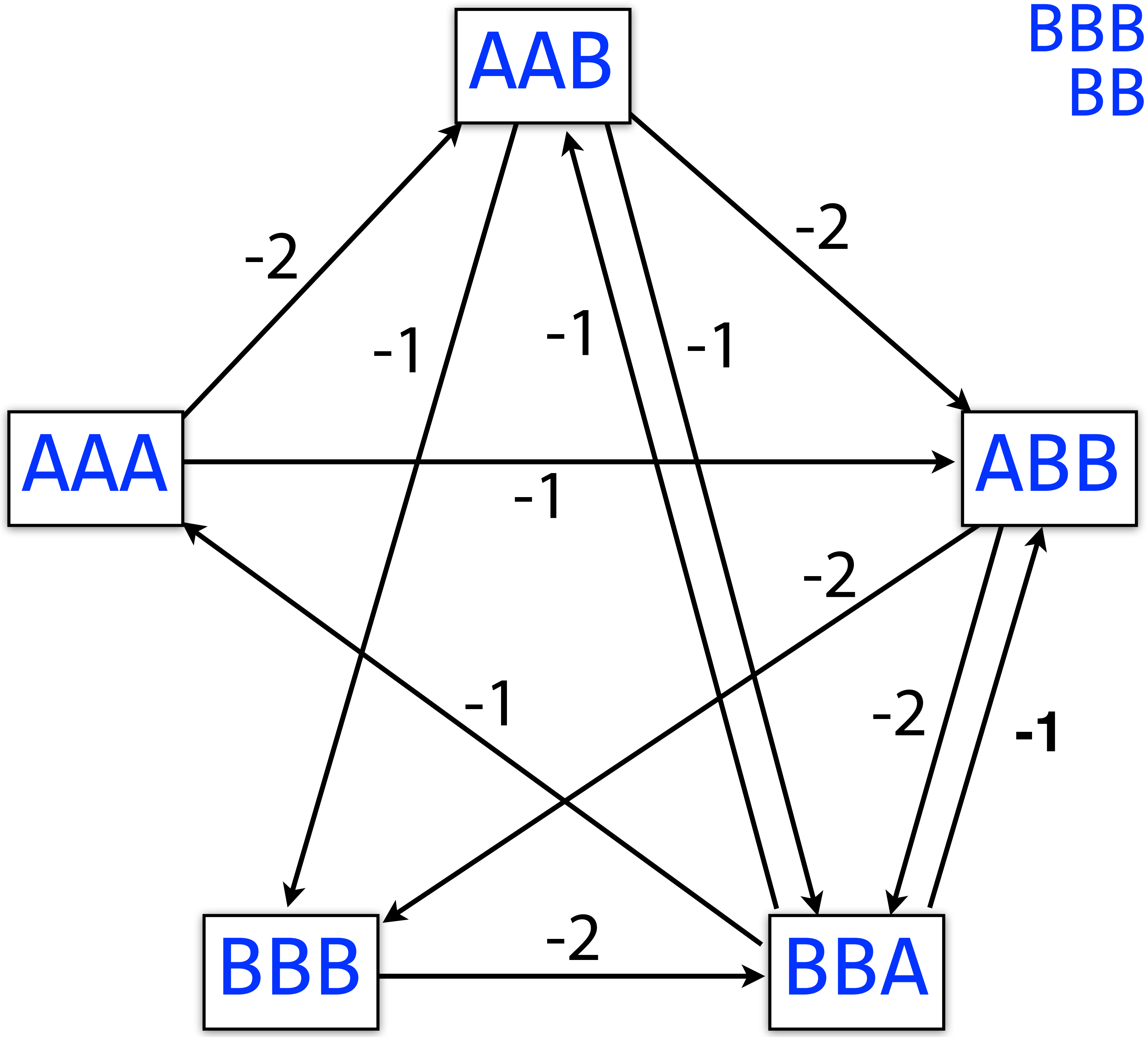
AAA
 AAB
  ABB
   BBB
    BBA

Can we solve it?

Imagine a modified overlap graph where each edge has cost = - (length of overlap)

SCS corresponds to a path that visits every node once, minimizing total cost along path

That's the *Traveling Salesman Problem* (*TSP*), which is NP-hard!



Slide courtesy of Ben Langmead

Shortest common superstring

Say we disregard edge weights and just look for a path that visits all the nodes exactly once

That's the *Hamiltonian Path* problem: NP-complete

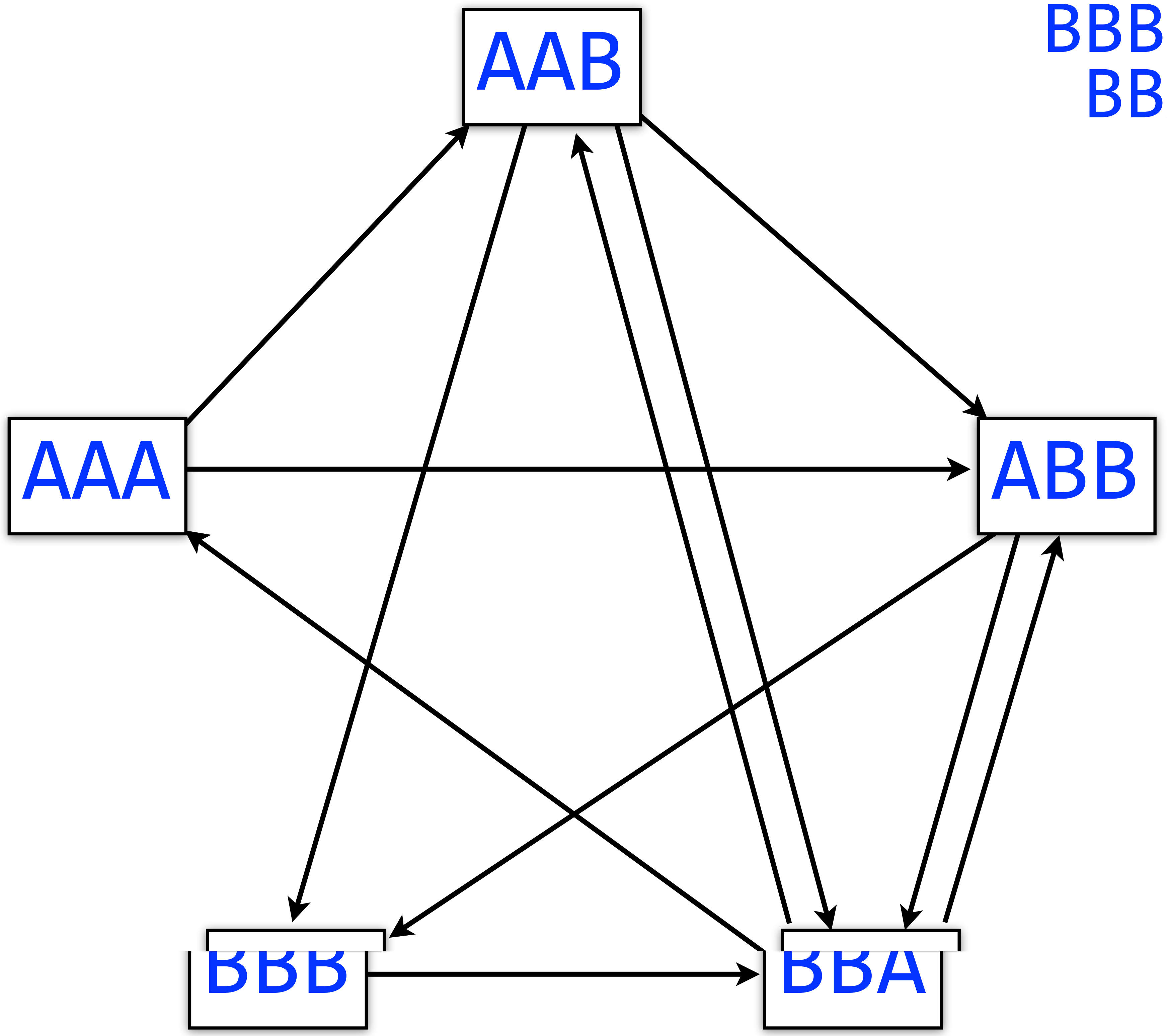
So, it's not even the weights that make visiting all nodes once hard

Indeed, it's well established that SCS is NP-hard

S: AAA AAB ABB BBB BBA

SCS(S): AAABBBA

```
AAA
 AAB
  ABB
   BBB
    BBA
```



Slide courtesy of Ben Langmead

Shortest common superstring & friends

Traveling Salesman, Hamiltonian Path, and Shortest Common Superstring are all NP-hard

For refreshers on Traveling Salesman, Hamiltonian Path, NP-hardness and NP-completeness, see Chapters 34 and 35 of "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein, or Chapters 8 and 9 of "Algorithms" by Dasgupta, Papadimitriou and Vazirani (free online: http://www.cs.berkeley.edu/~vazirani/algorithms)

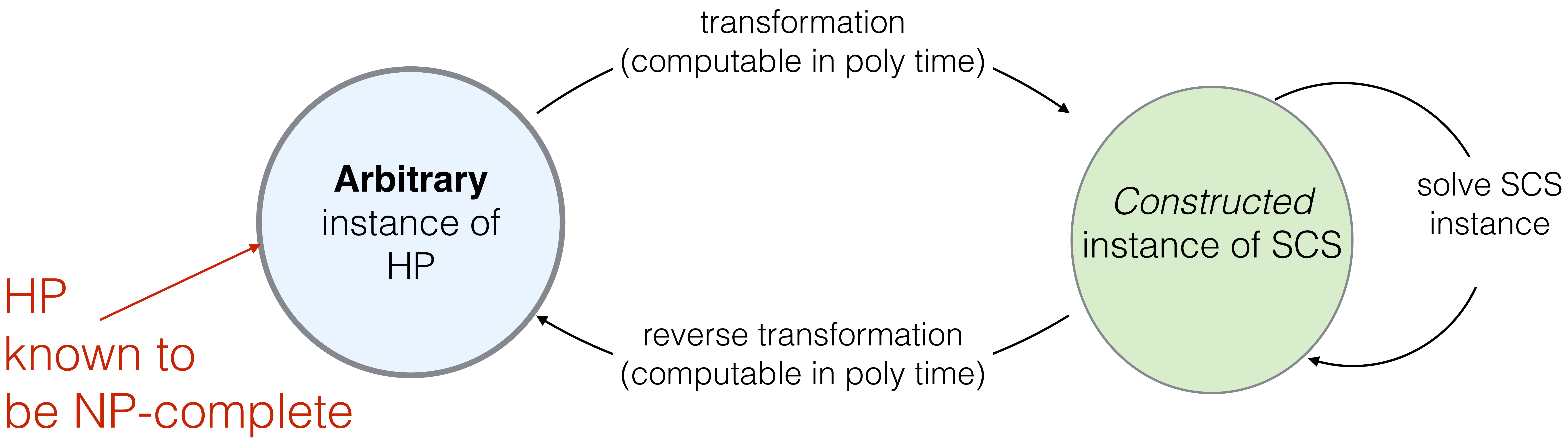
Who remembers reductions from 351?

Slide courtesy of Ben Langmead

Important note: The fact that we modeled SCS as NP-hard problems (TSP and HP) **does not** prove that (the decision version of) SCS is NP-complete. To do that, we must **reduce** a known NP-complete problem to **SCS**.

Given an instance I of a known hard problem, generate an instance I' of SCS such that if we can solve I' in polynomial time, then we can solve I in polynomial time. This *implies* that SCS is *at least* as hard as the hard problem.

This can be done e.g. with HAMILTONIAN PATH


transformation
(computable in poly time)
Arbitrary instance of HP
Constructed instance of SCS
solve SCS instance
reverse transformation
(computable in poly time)
HP known to be NP-complete

Shortest Common Superstring

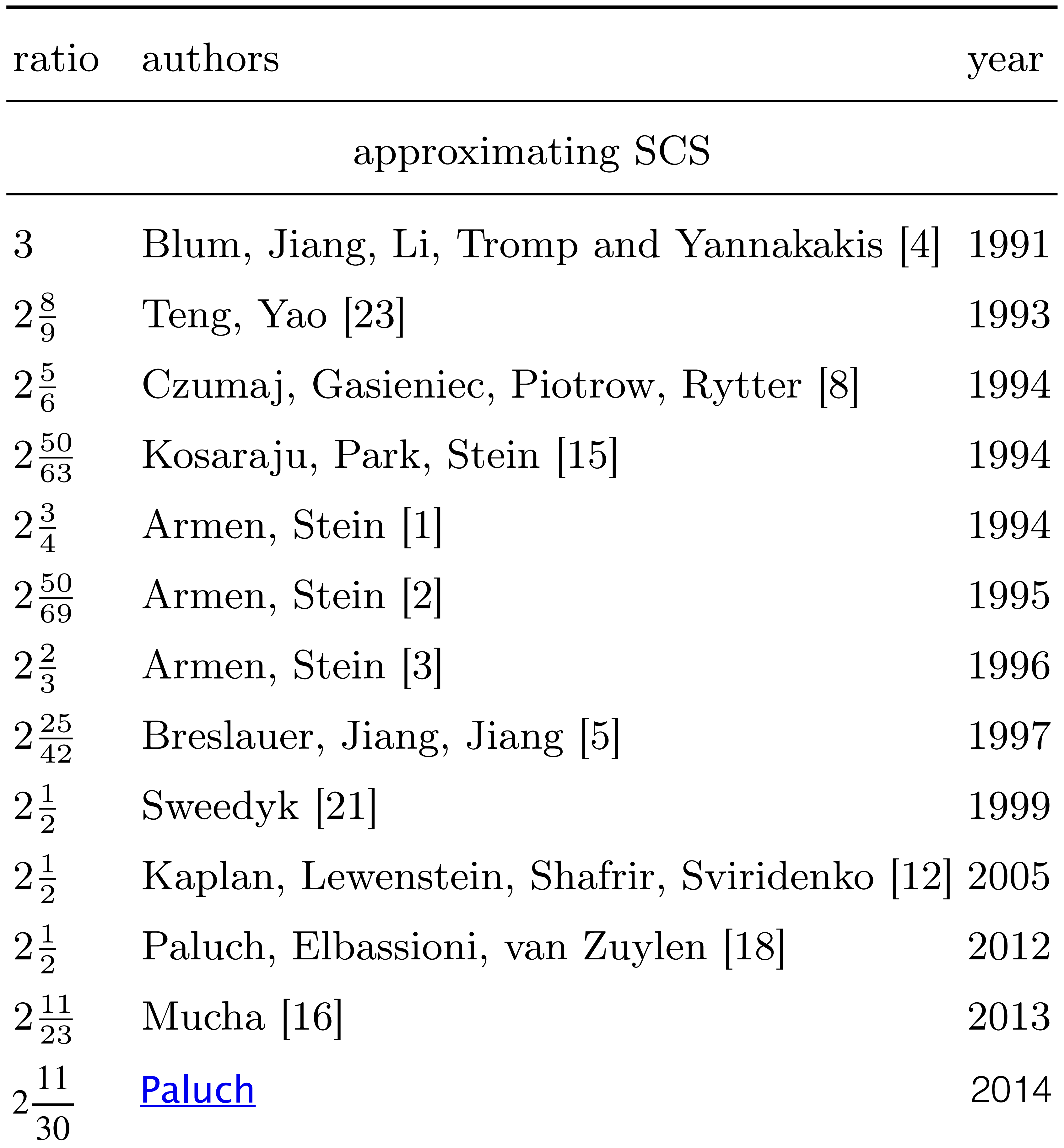
The fact that (the decision version of) SCS is **NP-complete** means that it is unlikely that there exists *any* algorithm that can solve a general instance of this problem in time polynomial in n — the number of input strings (i.e. reads in the case of genome assembly).

If we give up on finding a *shortest* possible superstring G, and instead look for one that's "near-shortest", how does the situation change?

Shortest Common Superstring

There's a "greedy" *heuristic* that turns out to be an *approximation algorithm* (provides a solution within a constant factor of the the optimum)

Different approx. (**not all greedy**)



ratio	authors	year
	approximating SCS	
3	Blum, Jiang, Li, Tromp and Yannakakis [4]	1991
$2\frac{8}{9}$	Teng, Yao [23]	1993
$2\frac{5}{6}$	Czumaj, Gasieniec, Piotrow, Rytter [8]	1994
$2\frac{50}{63}$	Kosaraju, Park, Stein [15]	1994
$2\frac{3}{4}$	Armen, Stein [1]	1994
$2\frac{50}{69}$	Armen, Stein [2]	1995
$2\frac{2}{3}$	Armen, Stein [3]	1996
$2\frac{25}{42}$	Breslauer, Jiang, Jiang [5]	1997
$2\frac{1}{2}$	Sweedyk [21]	1999
$2\frac{1}{2}$	Kaplan, Lewenstein, Shafrir, Sviridenko [12]	2005
$2\frac{1}{2}$	Paluch, Elbassioni, van Zuylen [18]	2012
$2\frac{11}{23}$	Mucha [16]	2013
$2\frac{11}{30}$	Paluch	2014

At each step, chose the pair of strings with the maximum overlap, merge them, and return the merged string to the collection.

Greedy conjecture factor of 2-OPT *is* the worst case

Open conjecture! We can prove 3.5, but many believe the factor is actually 2.

Golovnev, Kulikov, & Mihajlin. "Approximating Shortest Superstring Problem Using de Bruijn Graphs." Combinatorial Pattern Matching. Springer Berlin Heidelberg, 2013.

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

Algorithm in action (l = 1):

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

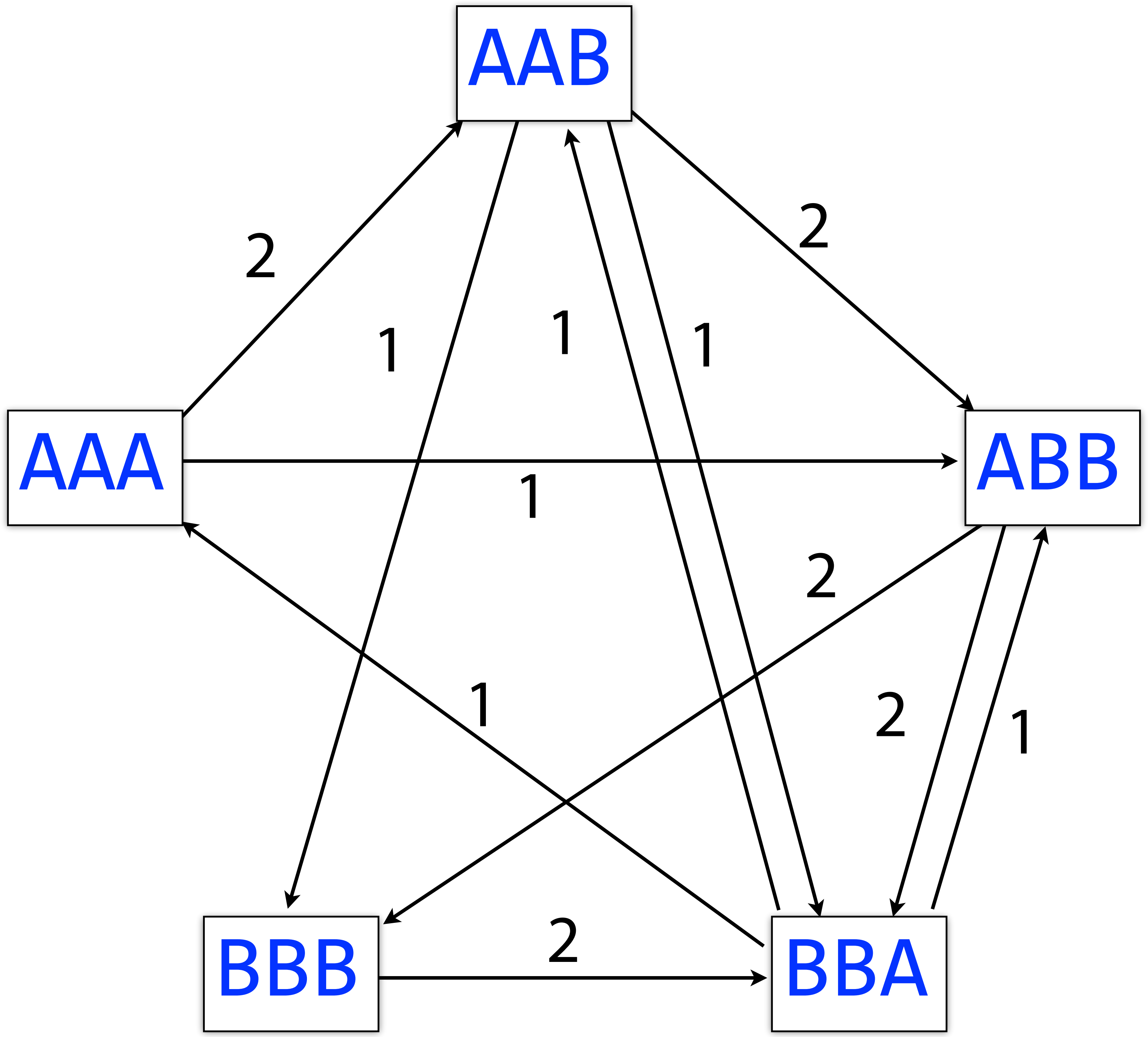
Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

Algorithm in action ($l = 1$):

├──── Input strings ────┤

AAA AAB ABB BBB BBA



Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

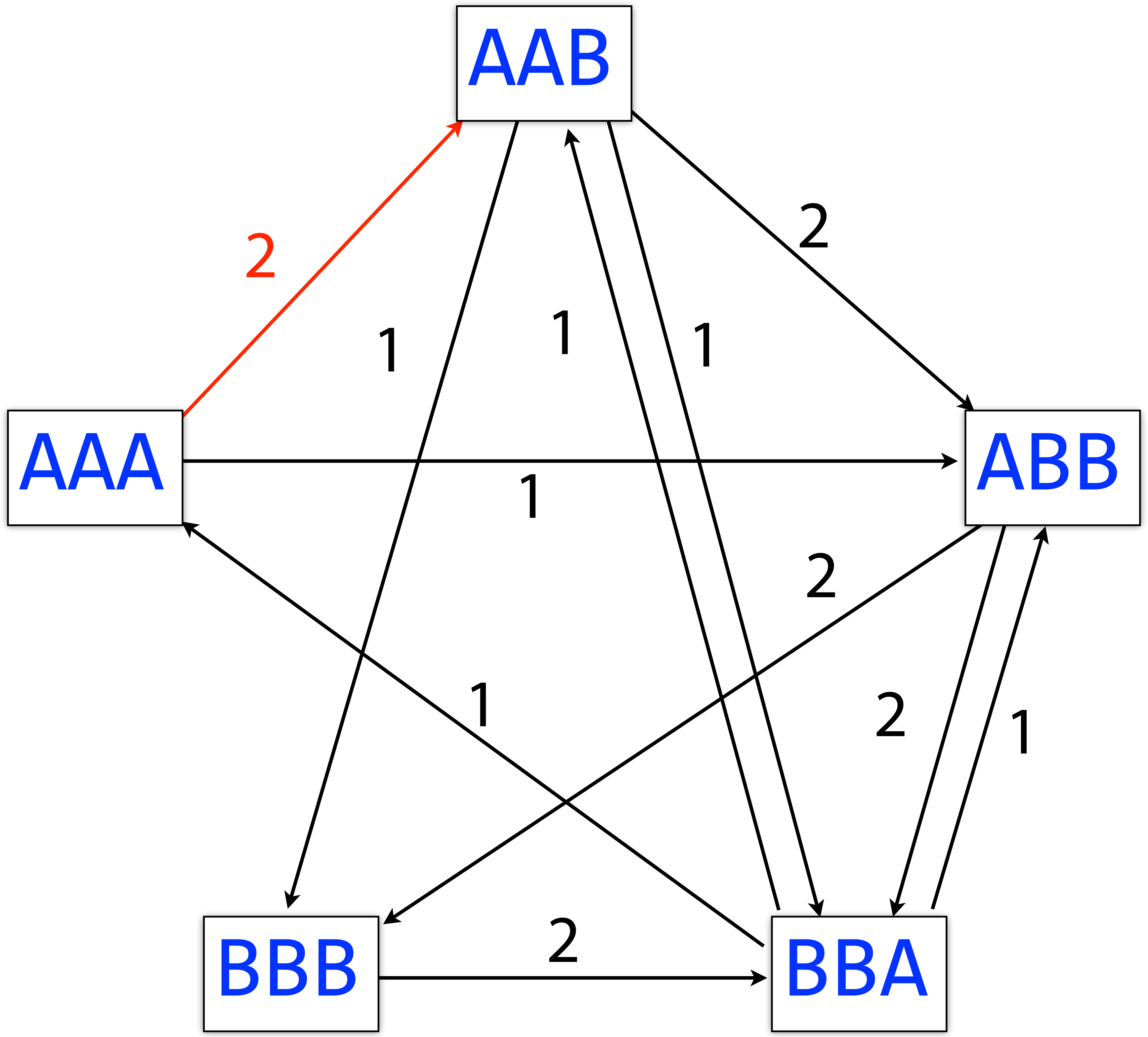
Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA



Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

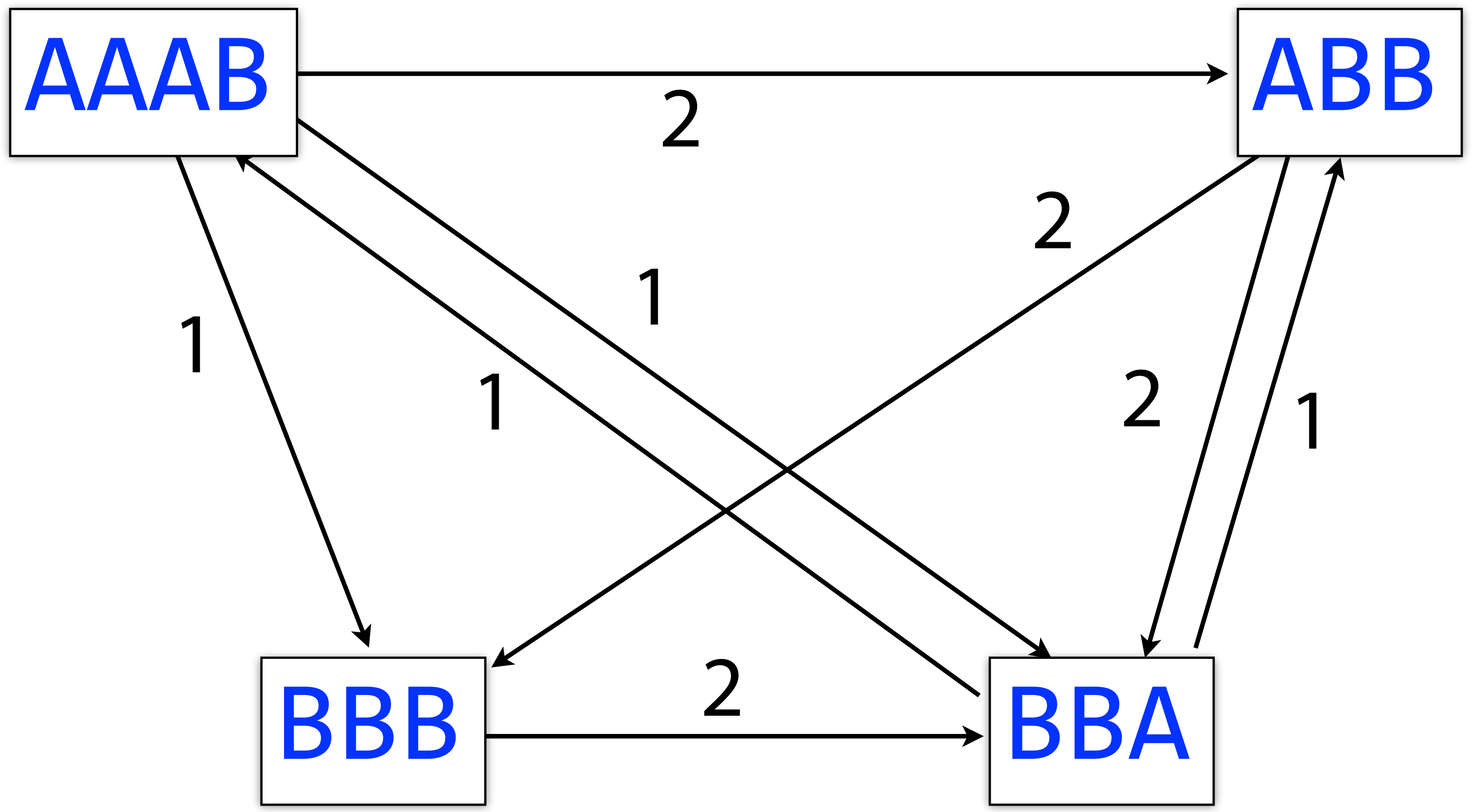
Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

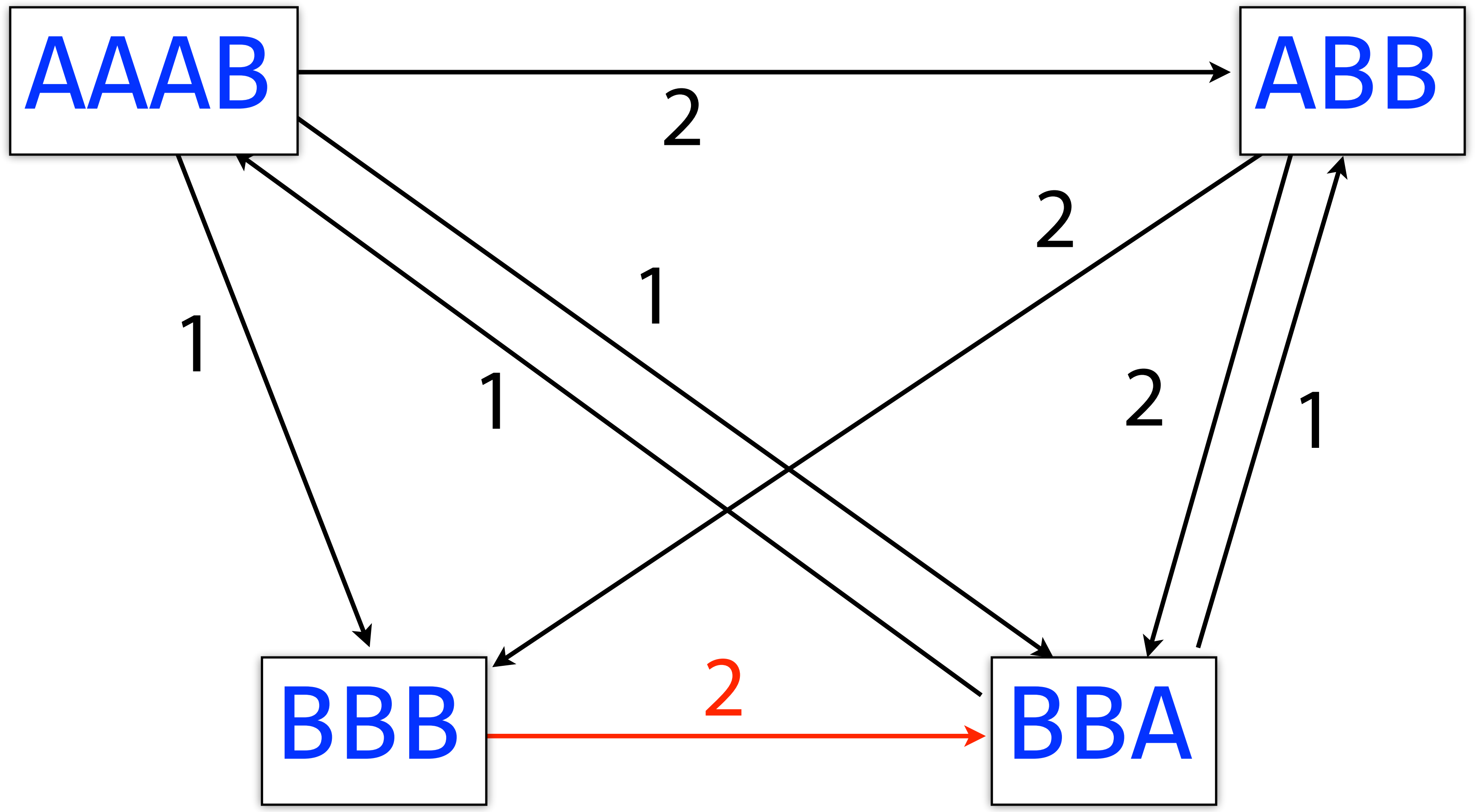
Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

Algorithm in action (l = 1):

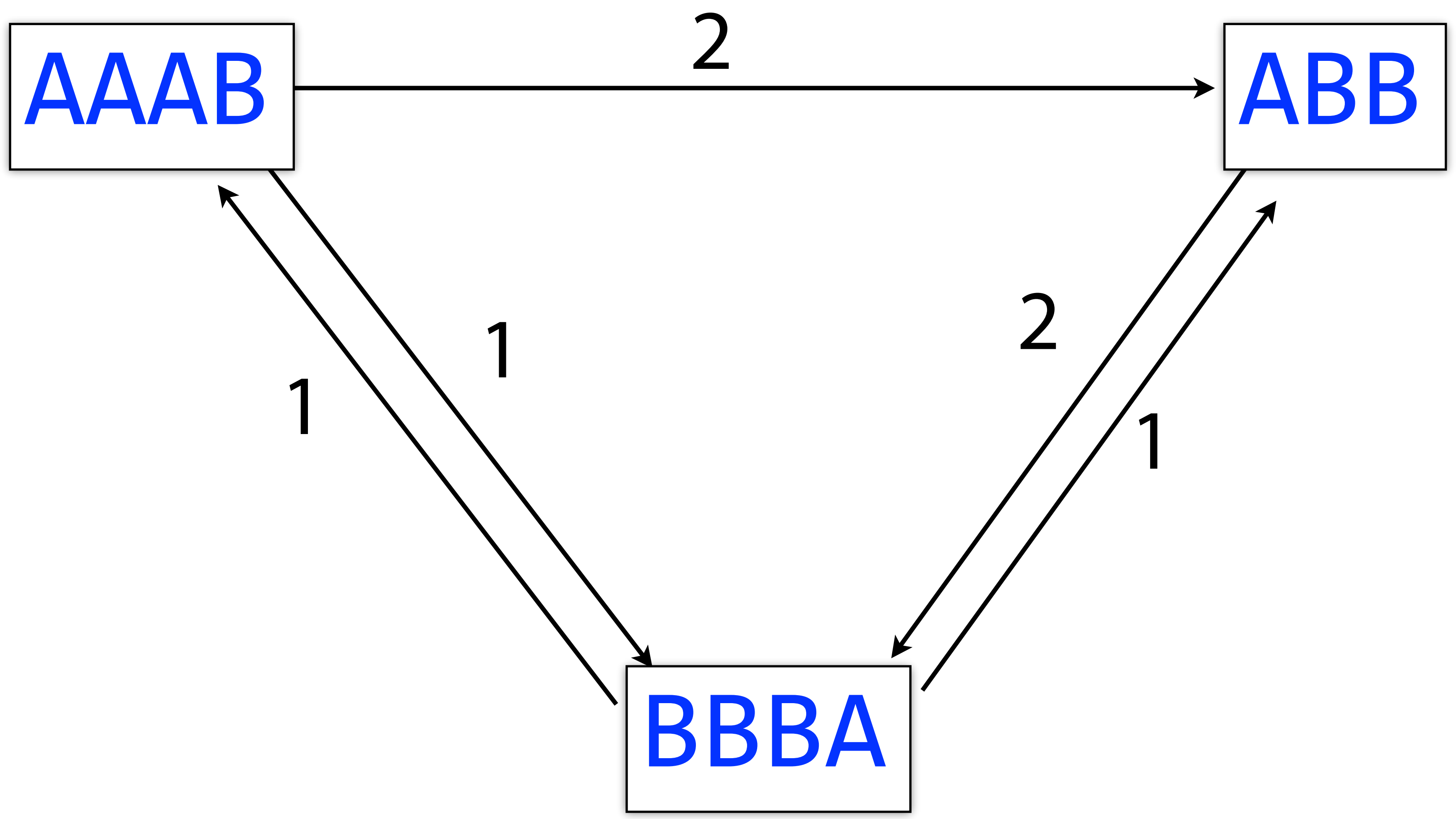
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB



Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

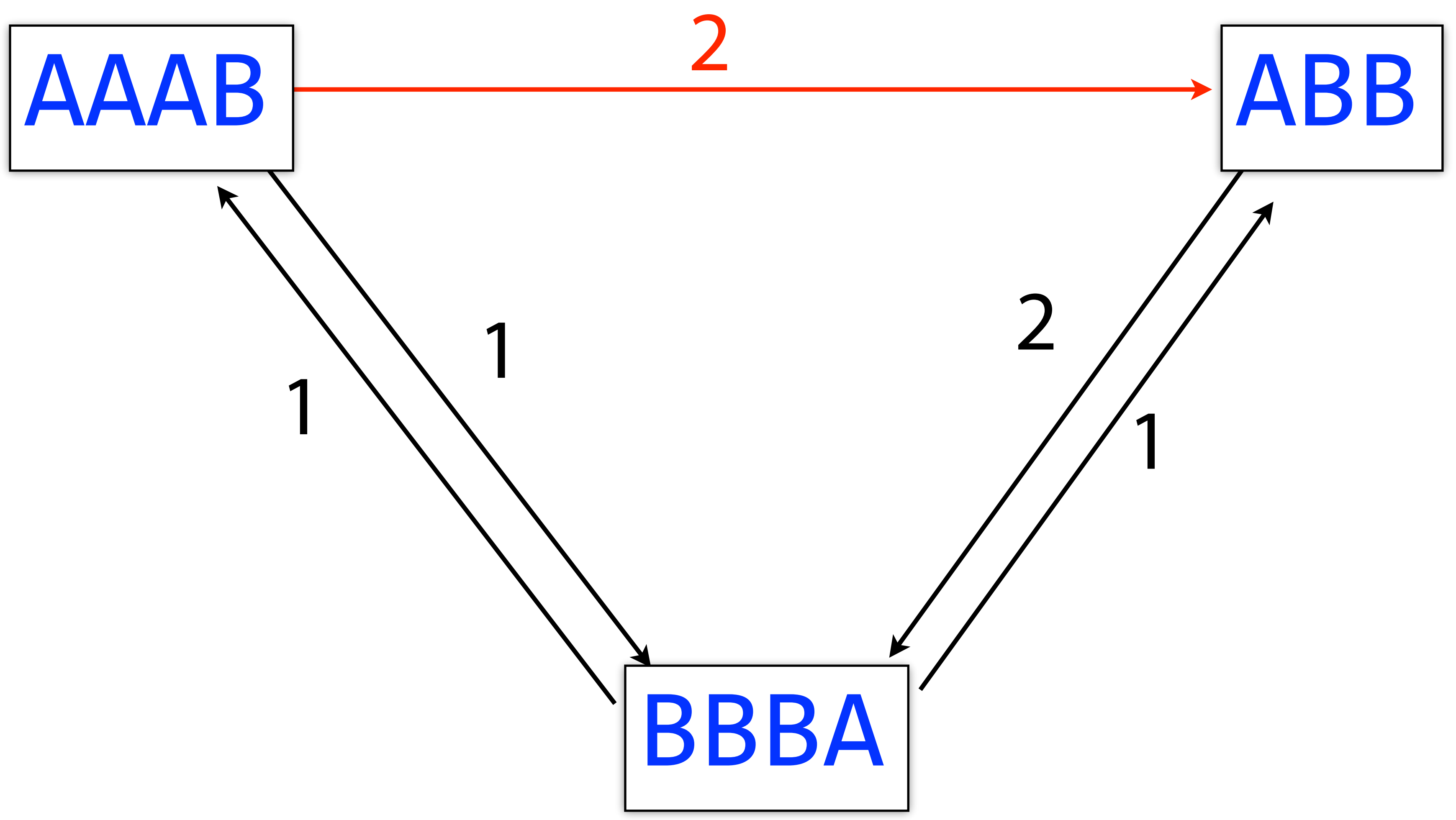
Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. 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



Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. l = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

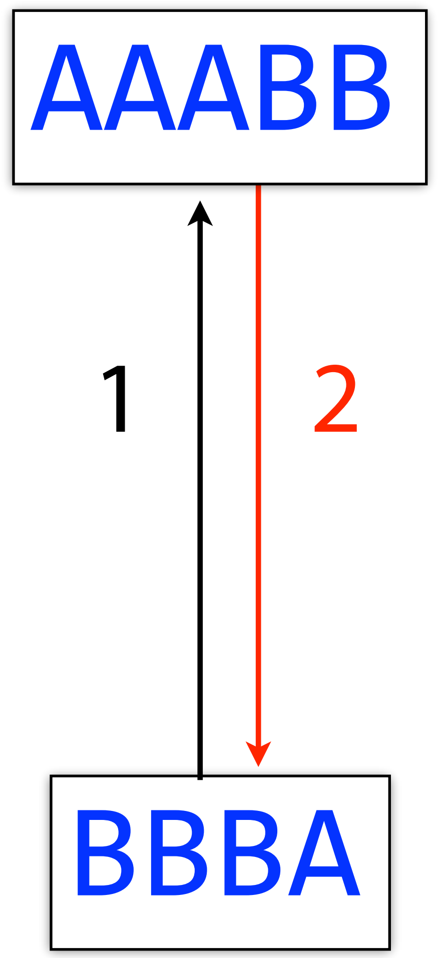
AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

AAABB BBBA



Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. $l =$ 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

AAABBBA

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Greedy-SCS: in each round, merge pair of strings with maximal overlap.

Stop when no more overlaps exist. Concatenate resulting strings. $l =$ 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

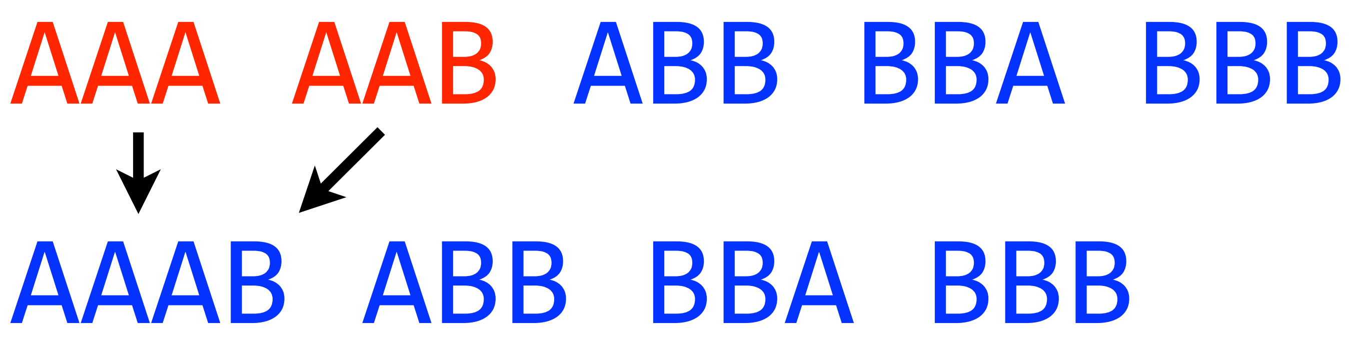
AAABBBA

That's the SCS

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Greedy shortest common superstring (when is it not optimal?)



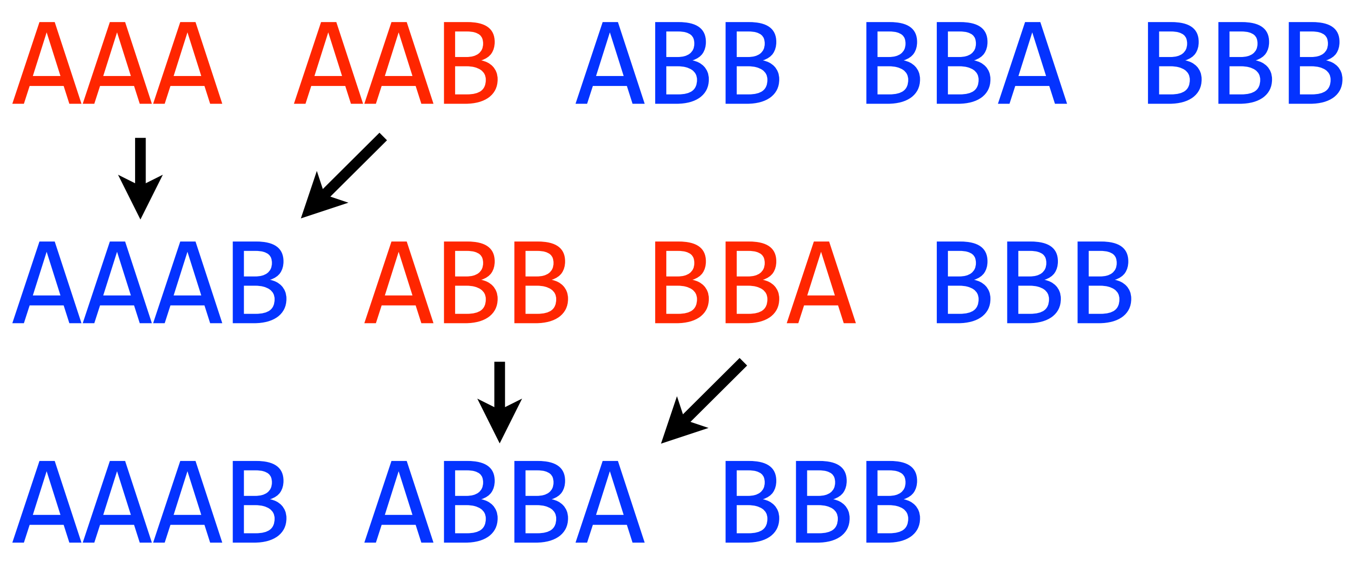
AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Greedy shortest common superstring (when is it not optimal?)



AAA AAB ABB BBA BBB

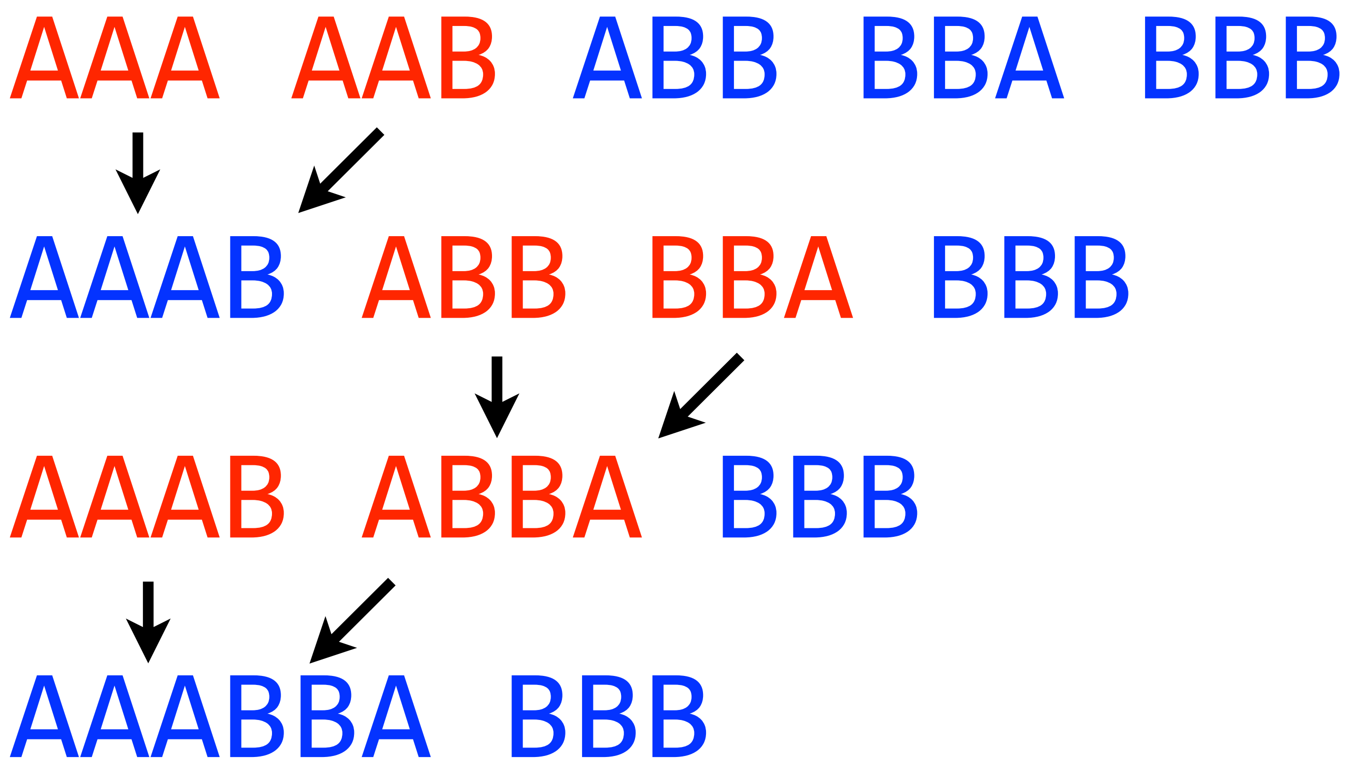
AAAB ABB BBA BBB

AAAB ABBA BBB

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Greedy shortest common superstring (when is it not optimal?)



AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

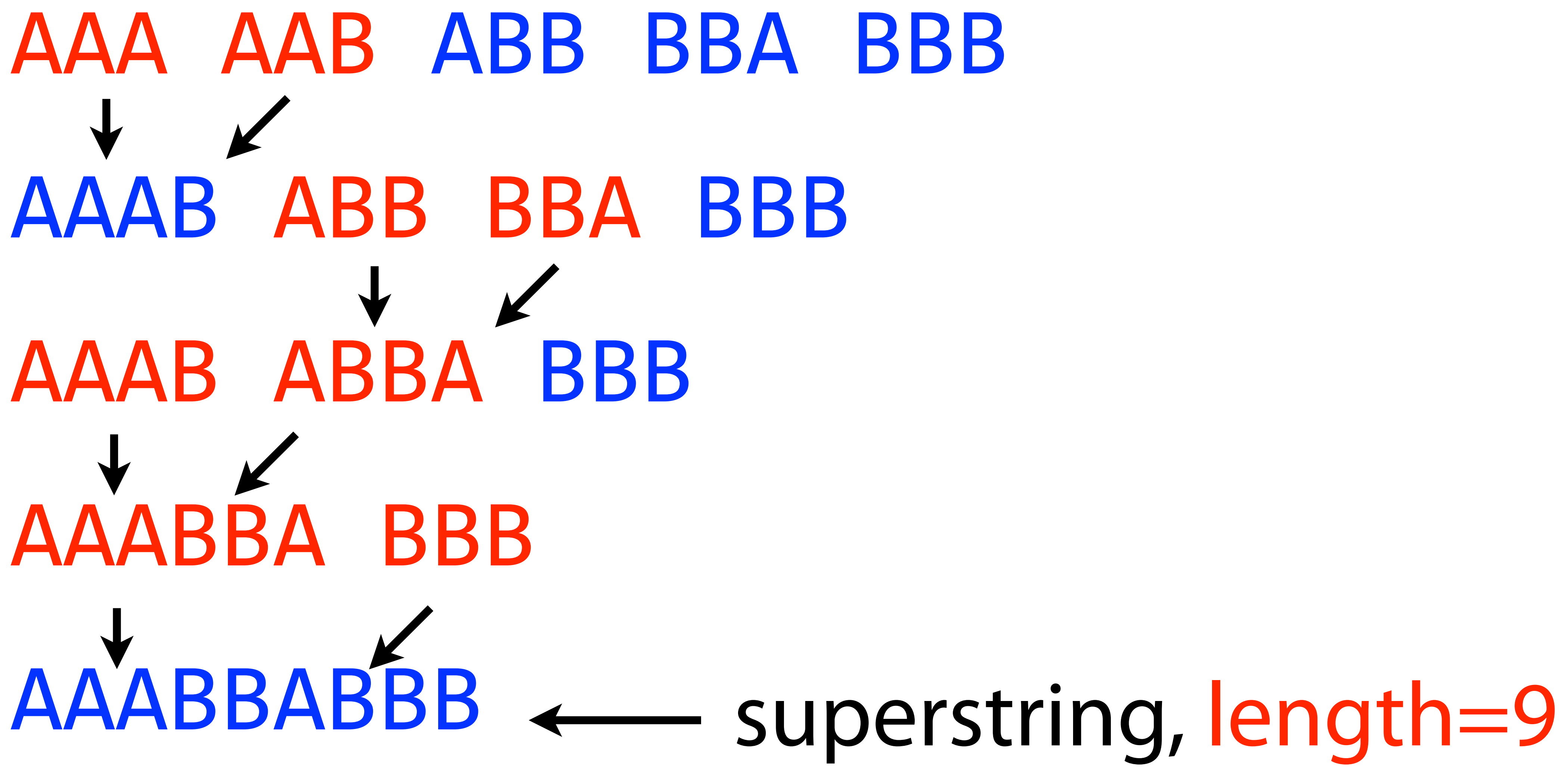
AAAB ABBA BBB

AAABBA BBB

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Greedy shortest common superstring (when is it not optimal?)



AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

AAAB ABBA BBB

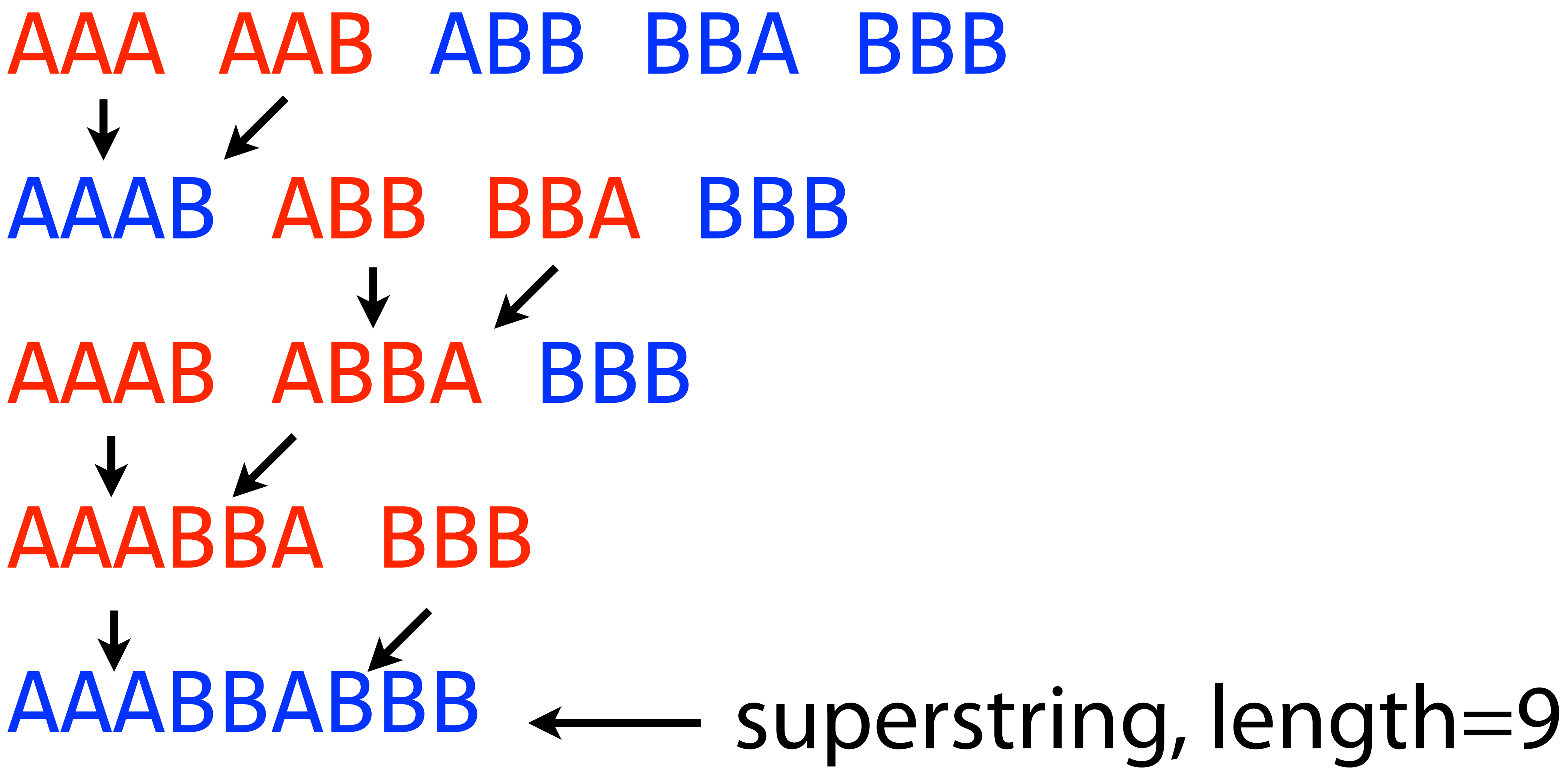
AAABBA BBB

AAABBABBB ← superstring, length=9

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Greedy shortest common superstring (when is it not optimal?)



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

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

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

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Why else might it not be a good model for assembly?

Greedy-SCS assembling all substrings of length 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
```

↑

Foiled by repeat!

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: greedy

Same example, but increased the substring length 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`

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: 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
```

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Shortest common superstring: 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

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Third law of assembly

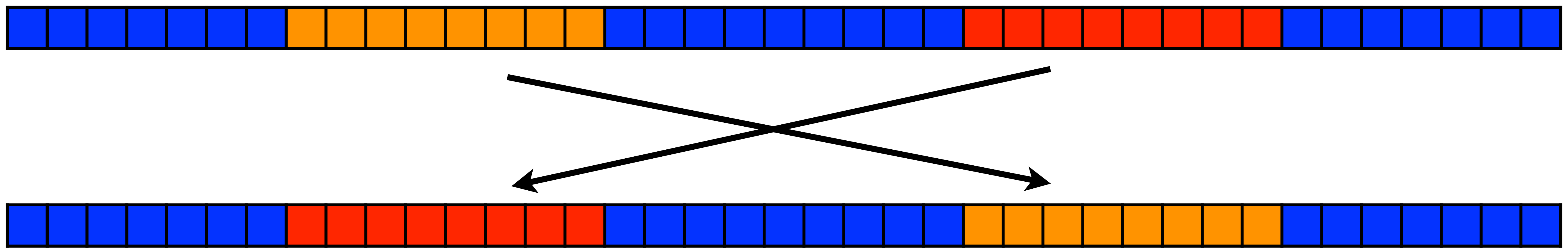
Repeats make assembly difficult; whether we can assemble without mistakes depends on length of reads and repetitive patterns in genome

Collapsing a tandem repeat: a_long_long_long_time

↓

a_long_long_time

Spurious rearrangement:

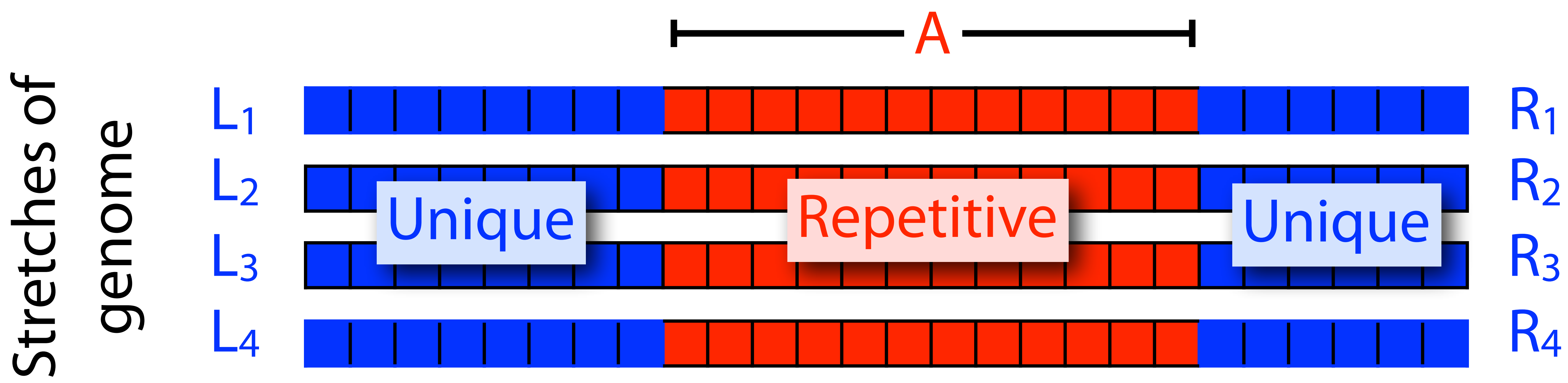


Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Repeats foil assembly

Portion of overlap graph involving repeat family A



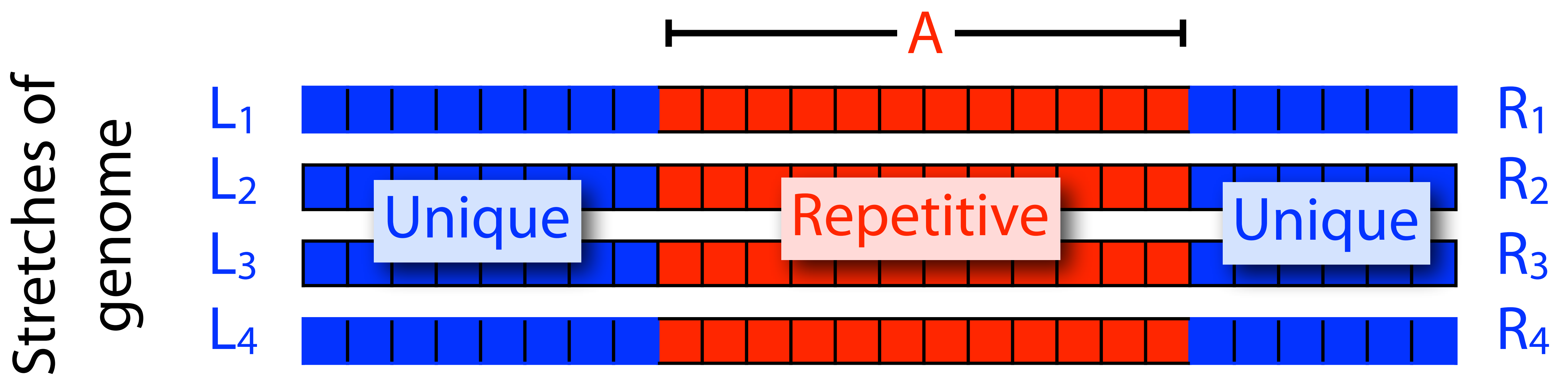
As are longer than read length

Slide courtesy of Ben Langmead

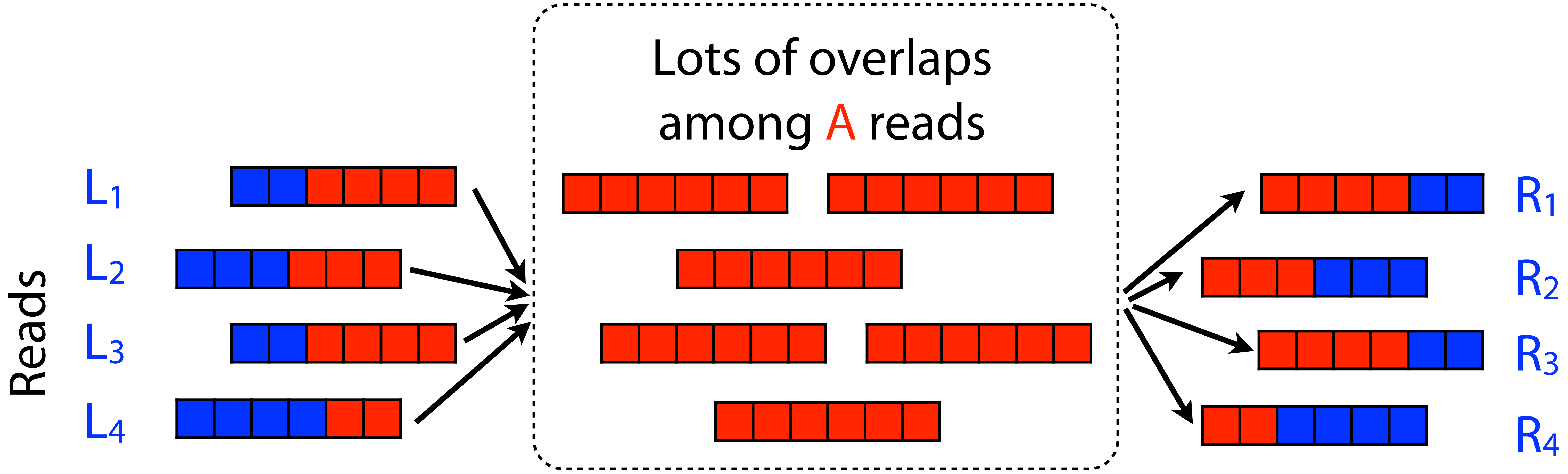
http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Repeats foil assembly

Portion of overlap graph involving repeat family A



As are longer than read length

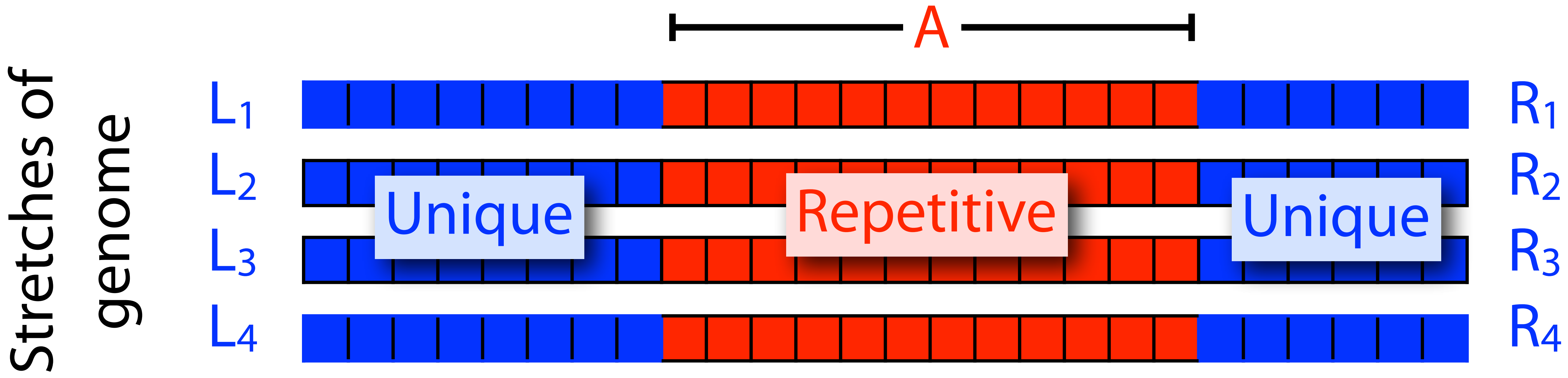


Slide courtesy of Ben Langmead

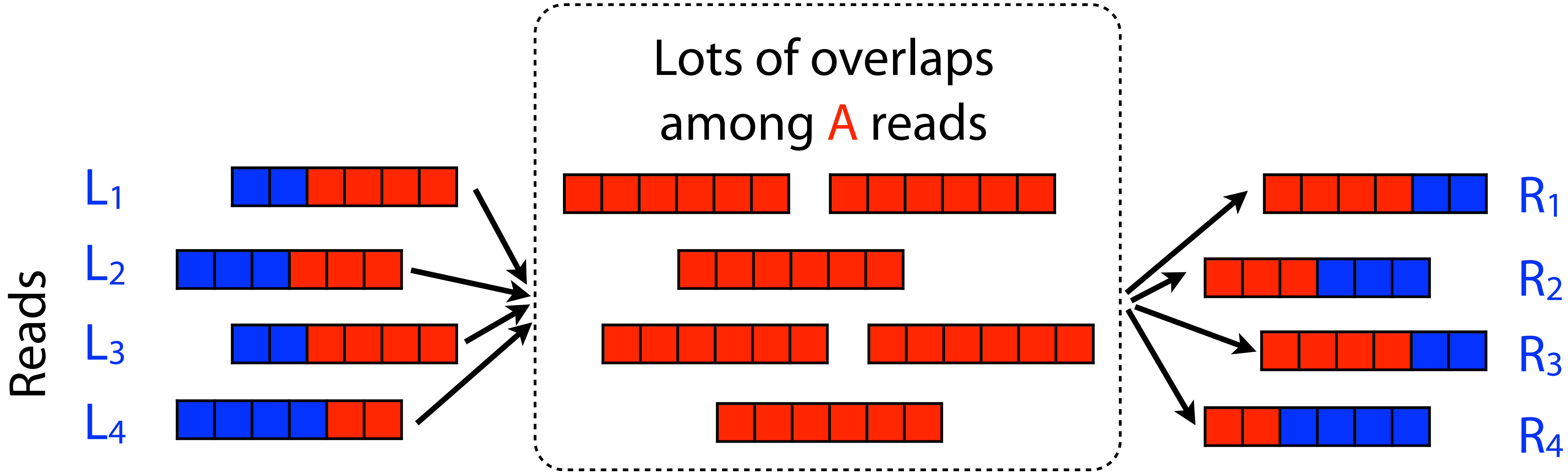
http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Repeats foil assembly

Portion of overlap graph involving repeat family A



As are longer than read length



Even if we avoid collapsing copies of A, we can't know which paths in correspond to which paths out

Slide courtesy of Ben Langmead

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

A practical implementation of SCS

Two components of the SCS implementation make things slow from a practical perspective on large data

1) All vs. all string comparison makes computing the initial edge set slow.

2) Searching / scanning for the highest-scoring edge in the overlap graph requires O(E) work in each iteration

A practical implementation of SCS

Two components of the SCS implementation make things slow from a practical perspective on large data

1) All vs. all string comparison makes computing the initial edge set slow.

Use a hash table from length ℓ prefix/suffix strings to their containing string to speed up edge discovery

2) Searching / scanning for the highest-scoring edge in the overlap graph requires O(E) work in each iteration

Use a priority queue to minimize the work done in each iteration

A practical implementation of SCS

Use a hash table from length ℓ prefix/suffix strings to their containing string to speed up edge discovery

1:ACAGTTA 3:AGAGTCG 4:CCAAGAG 6:AGCGCGC

7:GCGCGCA

2:GTTACCA 5:TAGCGCG

Make hash map from each first/last l-mer to string containing it

```
{ACA: 1, TTA: 1,
 GTT: 2, CCA: 2,
 AGA: 3, TCG: 3,
 CCA: 4, GAG: 4,
 TAG: 5, GCG: 5,
 AGC: 6, CGC: 6,
 GCG: 7, GCA: 7 }
```

Iterate over each input string and query *all* l-mers in the hash. If you find a hit, *check* if it induces a suffix/prefix overlap

A practical implementation of SCS

Iterate over each input string and query *all* l-mers in the hash. If you find a hit, *check* if it induces a suffix/prefix overlap

1:ACAGTTA

3:AGAGTCG

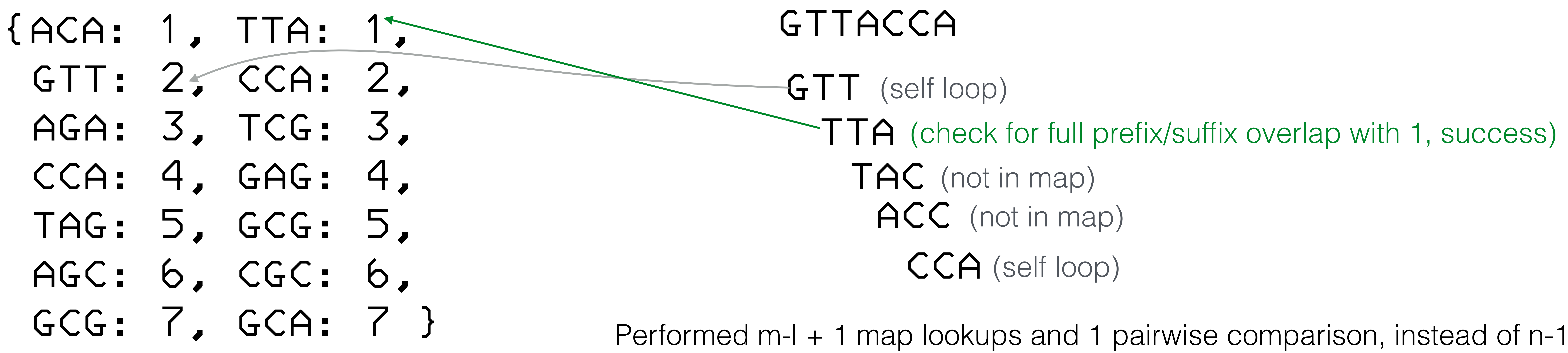
4:CCAAGAG

6:AGCGCGC

7:GCGCGCA

5:TAGCGCG

2:GTTACCA



```
{ACA: 1, TTA: 1,
 GTT: 2, CCA: 2,
 AGA: 3, TCG: 3,
 CCA: 4, GAG: 4,
 TAG: 5, GCG: 5,
 AGC: 6, CGC: 6,
 GCG: 7, GCA: 7 }
```

GTTACCA

GTT (self loop)

TTA (check for full prefix/suffix overlap with 1, success)

TAC (not in map)

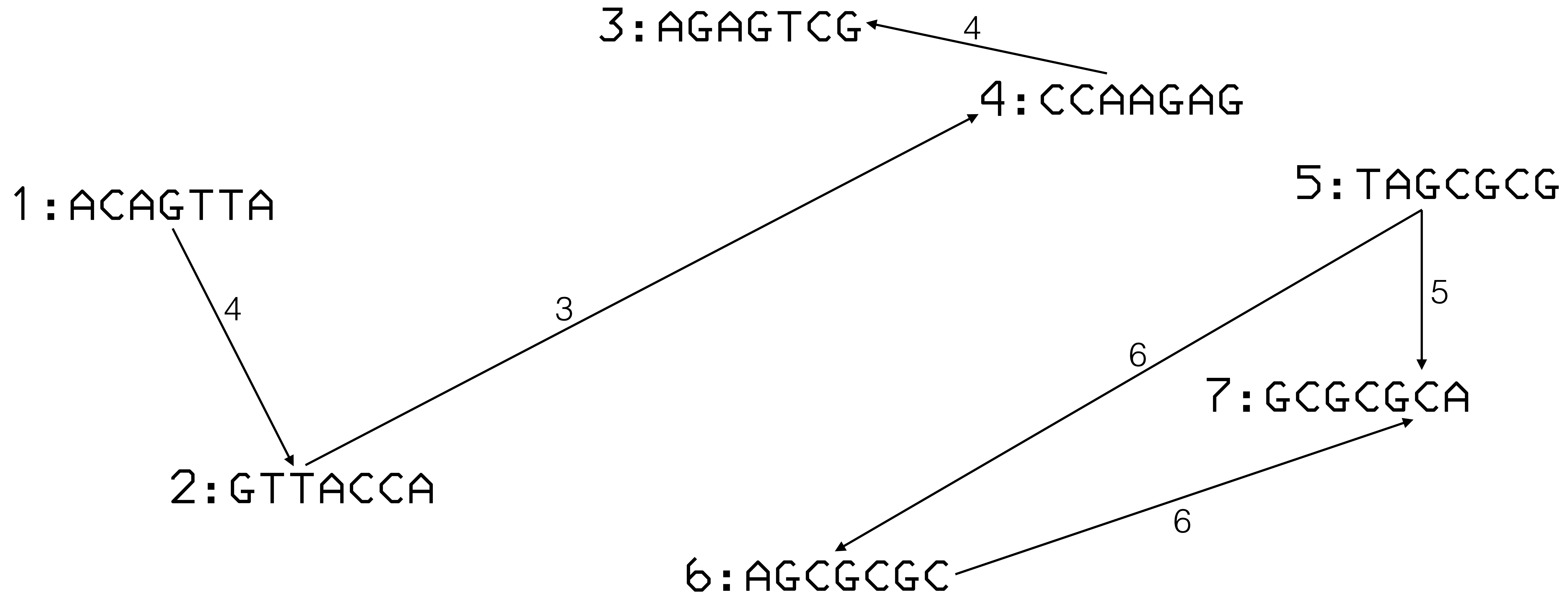
ACC (not in map)

CCA (self loop)

Performed m-l + 1 map lookups and 1 pairwise comparison, instead of n-1 pairwise comparisons!

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration

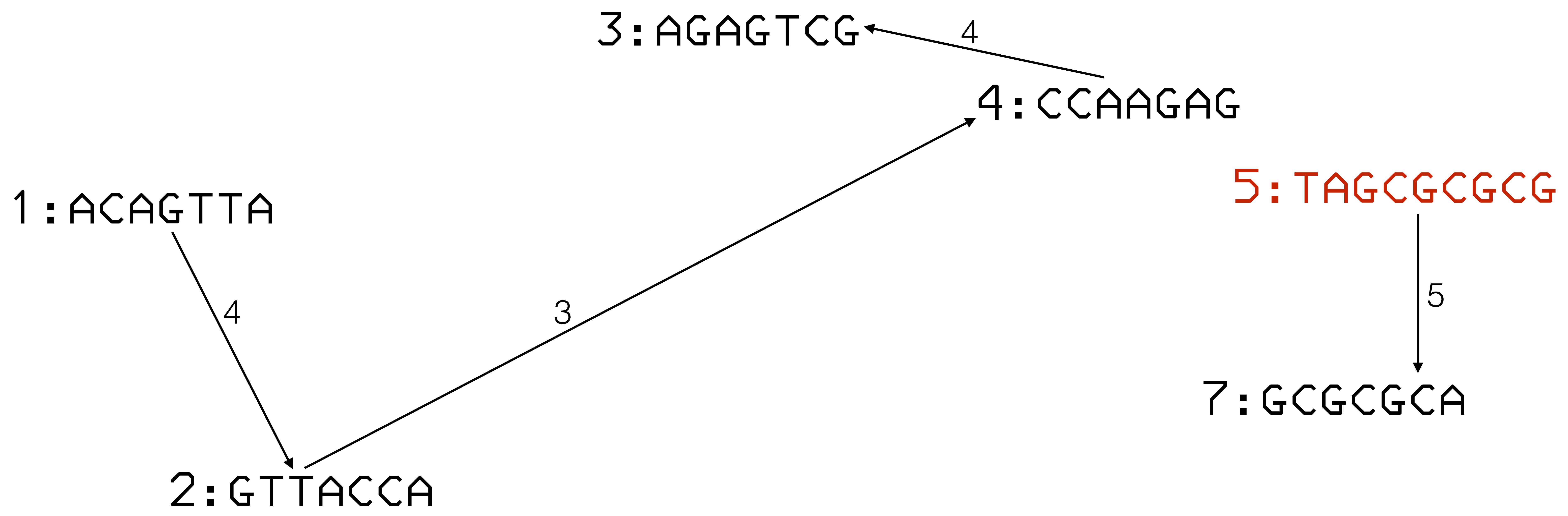


PQ edges: ['(5, 6, -6)', '(5, 7, -5)', '(6, 7, -6)', '(2, 4, -3)', '(1, 2, -4)', '(4, 3, -4)']

* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration



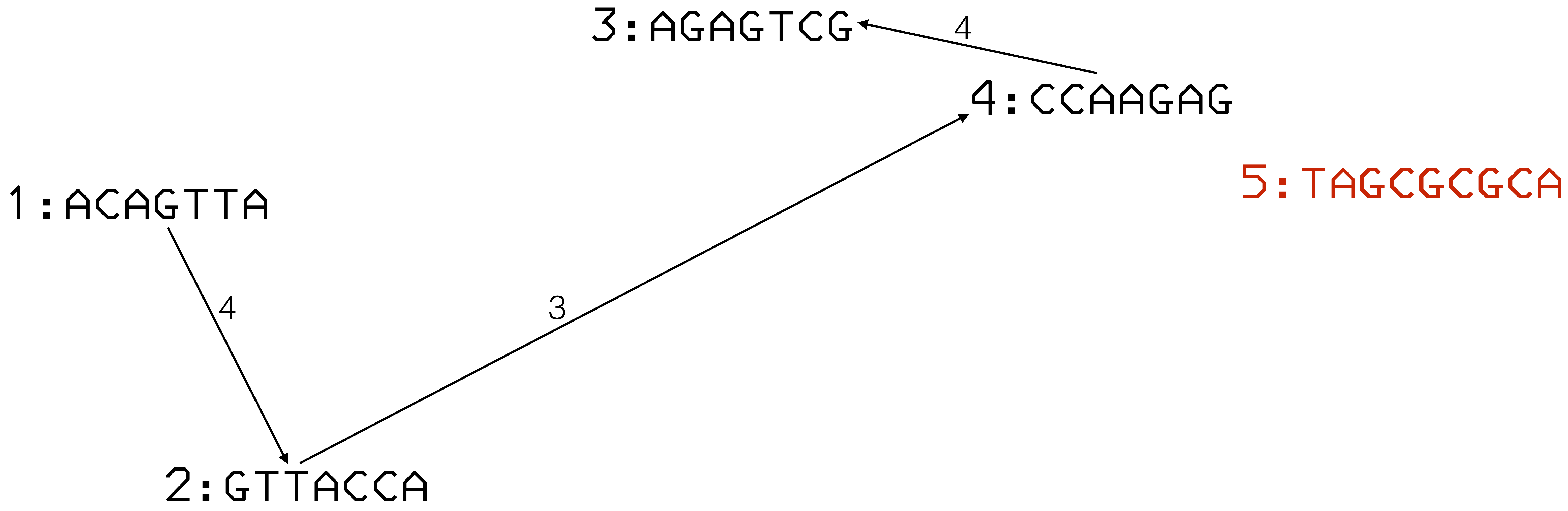
Since 7 touched 5&6 (which now belong in 5) we must compute overlap of 7,5.
Here, no new overlaps occur.

PQ edges: ['(5, 6, -6)', '(5, 7, -5)', '(6, 7, -6)', '(2, 4, -3)', '(1, 2, -4)', '(4, 3, -4)']

* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration



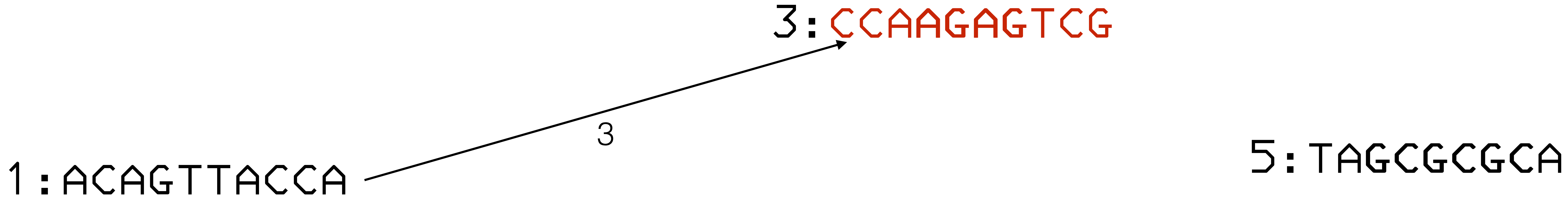
Merge 5,7 to make new 5

PQ edges: ['(5, 7, -5)', '(2, 4, -3)', '(1, 2, -4)', '(4, 3, -4)']

* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration



3:CCAAGAGTCG

3

1:ACAGTTACCA

5:TAGCGCGCA

Merge 4,3 to make new 3, which forces us to evaluate the 1->3 edge

PQ edges: ['(4, 3, -4)','(1, 4, -3)']

* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration

1:ACAGTTACCAAGAGTCG

5:TAGCGCGCA

Merge 1,3 to make new 1

PQ edges:[’(1, 4, -3)']

* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration

1:ACAGTTACCAAGAGTCG

5:TAGCGCGCA

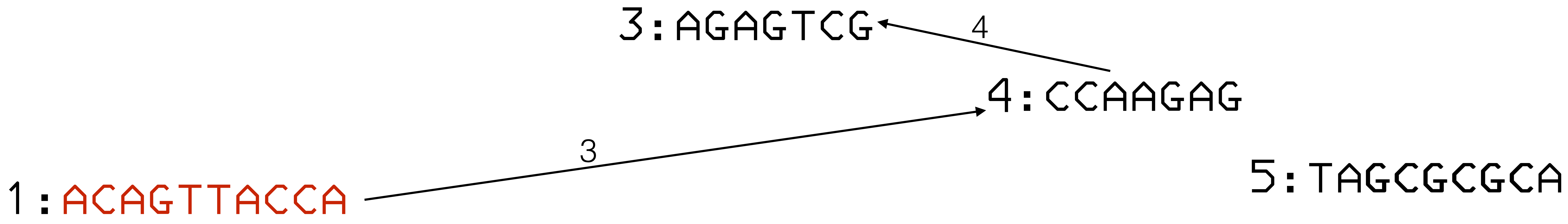
These are the final greedy superstrings! There's only one "trick" left.

PQ edges:[]

* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration



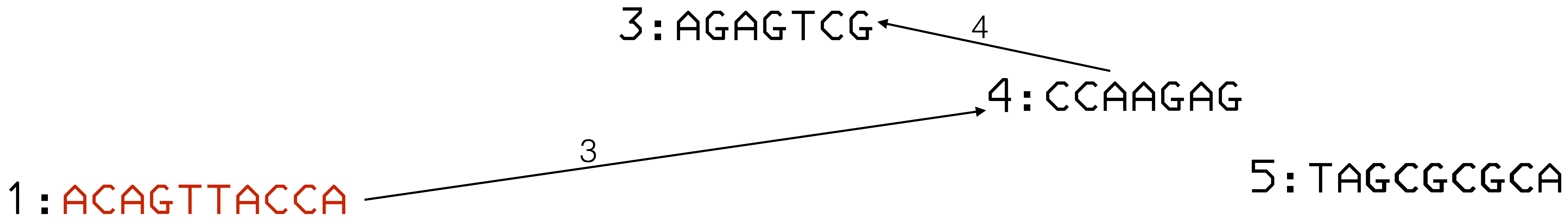
Note here, once we pop (1,2,-4) we still have a pair (2,4,-3) in the PQ. But at this point, string “2” doesn’t exist anymore! If we have to relabel in the queue each time we perform a merge, this will be slow. How do we handle this?

PQ edges: ['(1, 2, -4)', '(2, 4, -3)', '(4, 3, -4)']

* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration



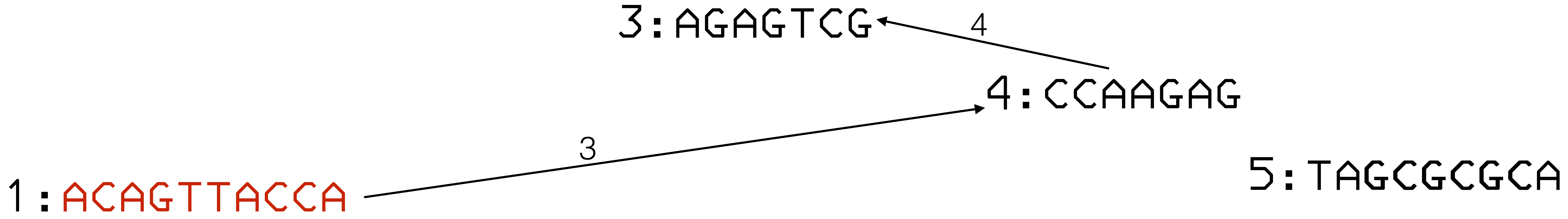
Solution: We maintain a “union-find” data structure. Each string is initially its own parent. When we merge strings a,b we make a the parent of b in the union find data structure. When we pop an element from the PQ, the first thing we do is find the “roots” of the union-find components where they belong. Those are the strings whose overlaps we must evaluate, and that we may choose to merge.

PQ edges: ['(1, 2, -4)', '(2, 4, -3)', '(4, 3, -4)']

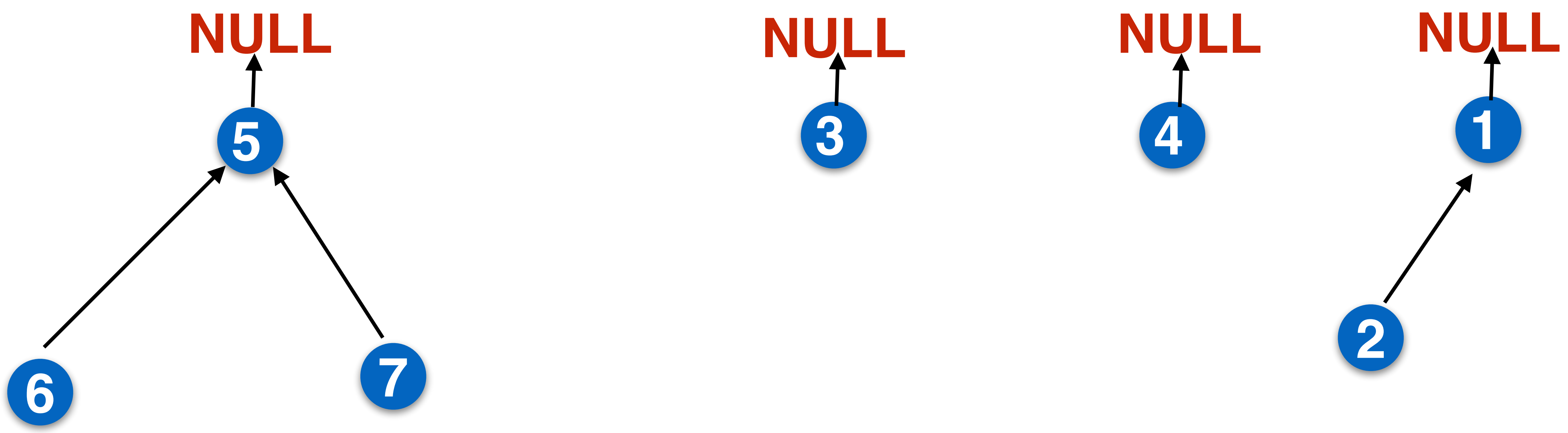
* read as source, target, - length of overlap

A practical implementation of SCS

Use a priority queue to minimize the work done in each iteration



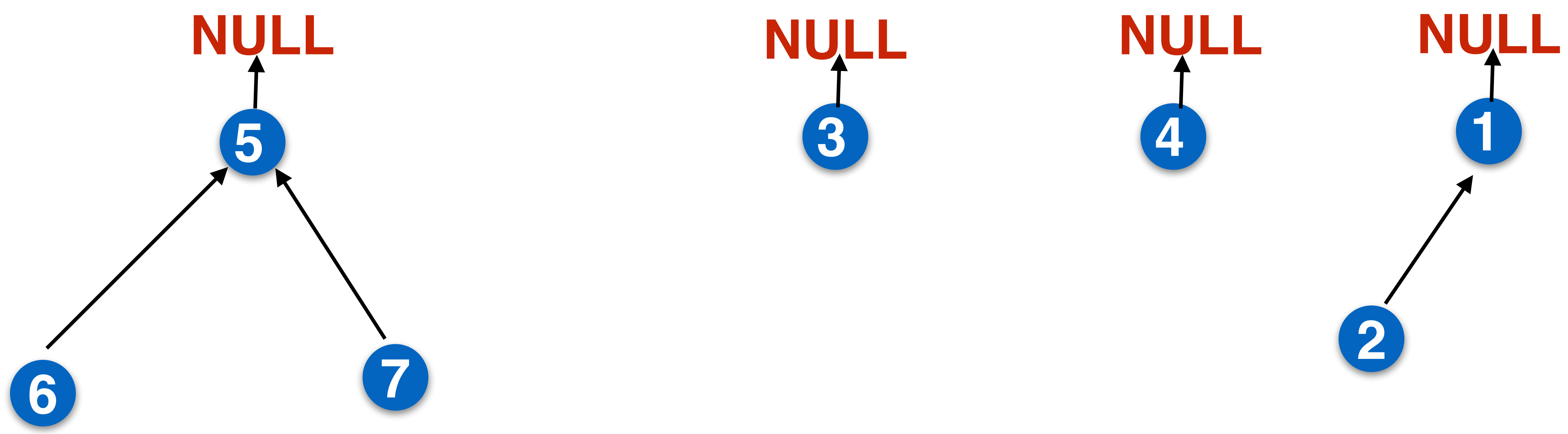
At this point, the union find looks like this:



As you can see, it directly encodes the series of merges performed. Further, for each original string, we can see what "current" string it resides in by walking up its tree until we hit the root.

A practical implementation of SCS

At this point, the union find looks like this:



As you can see, it directly encodes the series of merges performed. Further, for each original string, we can see what "current" string it resides in by walking up its tree until we hit the root.

For our purposes "basic" union-find will suffice. Note, however it can be made *much* more efficient.

See e.g.: "path compression", "union by rank" — both *easy* ways to make the data structure much faster.

Take-home message:

We are interested in *correct and efficient algorithms* for solving *well-specified* problems.

We must be careful about how we *pose* the problems.

Actually, shortest common superstring is a rather poor model for sequence assembly, due to repeats and errors.