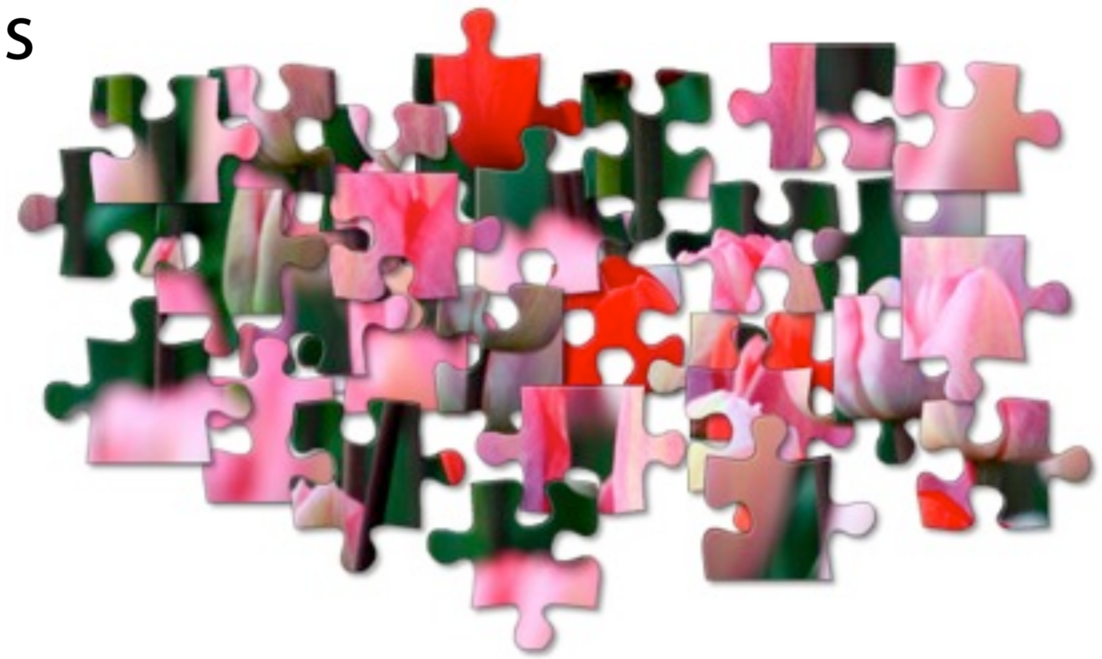
Assembly & shortest common superstring

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

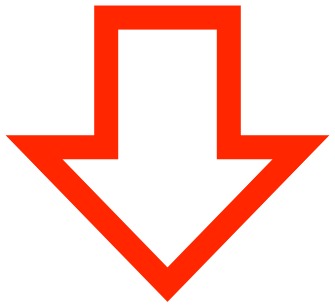
Assembly

Reads



+

Reference genome



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

Input DNA

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...

From these

```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
```

Reconstruct this

```
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Assembly

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

CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

From these

Reconstruct this

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Assembly

Key term: *coverage*. Usually it’s short for *average coverage*: the average number of reads covering a position in the genome.

```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA                  177 nucleotides
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT     35 nucleotides
```

Average coverage = 177 / 35 ≈ 7x

Assembly

Coverage could also refer to the number of reads covering a particular position in the genome:

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

Coverage at this position = 6

Assembly

Basic principle: the more similarity there is between the end of one read and the beginning of another...

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

...the more likely they are to have originated from overlapping stretches of the genome:

```
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Assembly

Say two reads truly originate from overlapping stretches of the genome. Why might there be differences?

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

1. Sequencing error

2. Difference between inhereted *copies* of a chromosome

   E.g. humans are diploid; we have two copies of each chromosome, one from mother, one from father. The copies can differ:

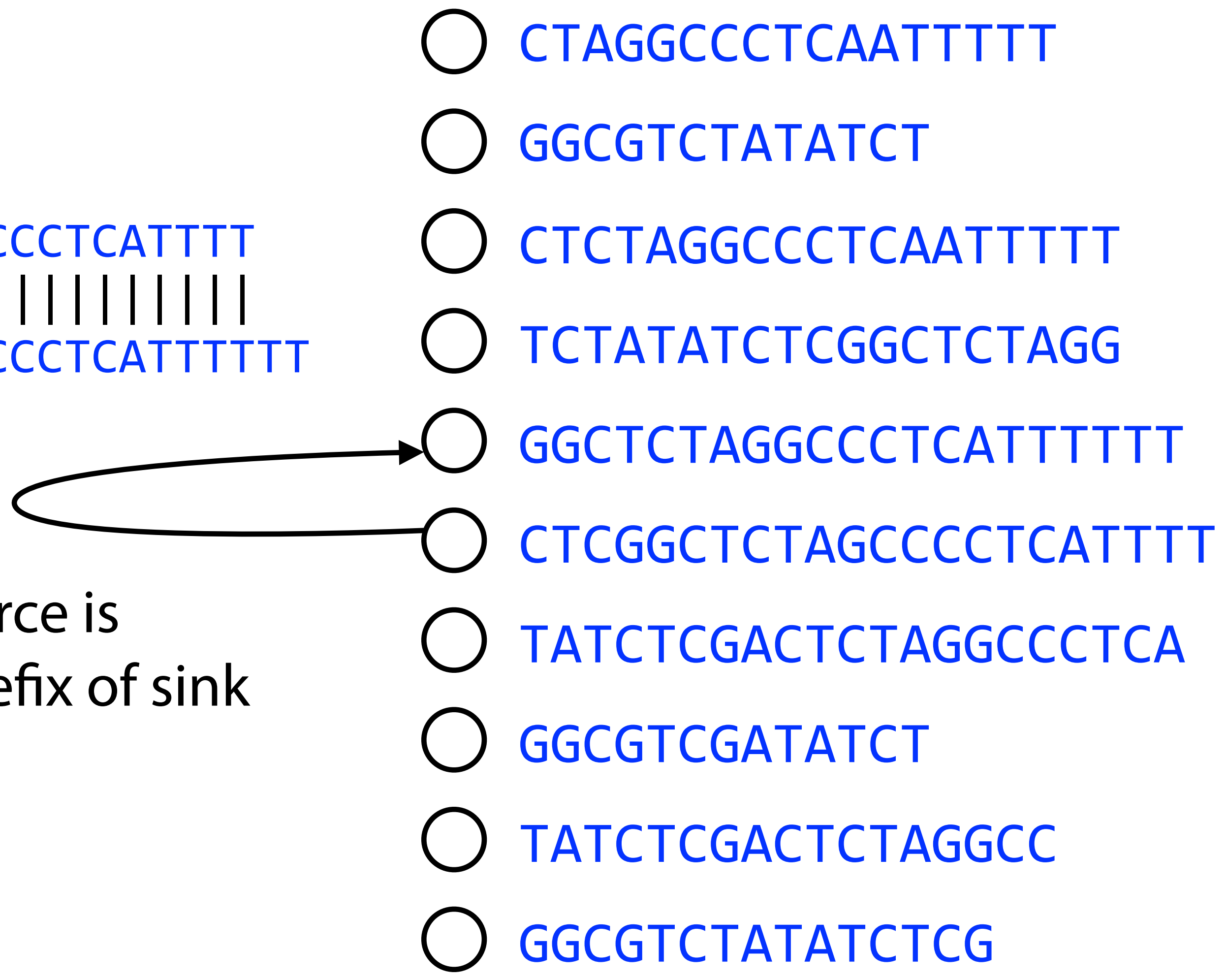
```
Read from Mother:         TATCTCGACTCTAGGCC
                          ||||||| |||||||
Read from Father:     TCTATATCTCGGCTCTAGG

Sequence from Mother: TCTATATCTCGACTCTAGGCC
Sequence from Father: TCTATATCTCGGCTCTAGGCC
```

We'll mostly ignore ploidy, but real tools must consider it

Overlaps

Finding all overlaps is like building a *directed graph* where directed edges connect overlapping nodes (reads)

```
CTCGGCTCTAGCCCCTCATTTT
   |||||||| ||||||||||
   GGCTCTAGGCCCTCATTTTTT
```



Suffix of source is similar to prefix of sink

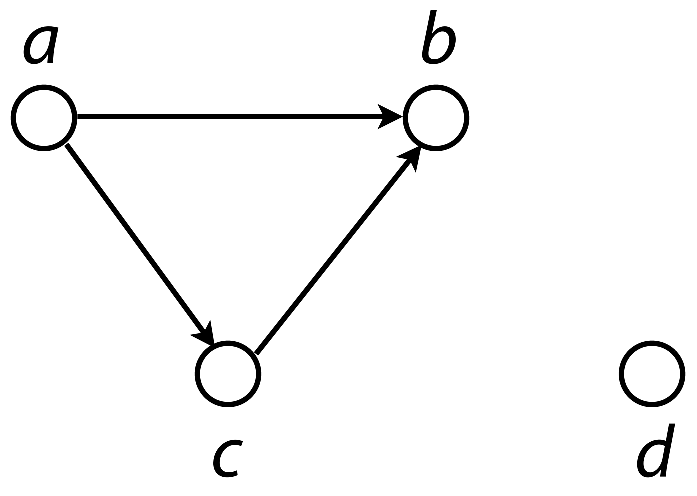
Directed graph review

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

Directed edge is an *ordered pair* of vertices.
First is the *source*, second is the *sink*.

Vertex is drawn as a circle

Edge is drawn as a line with an arrow connecting two circles



Vertex also called *node* or *point*

Edge also called *arc* or *line*

Directed graph also called *digraph*

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

$E = \{ (\underset{\substack{\uparrow \\ \text{Source}}}{a}, \underset{\substack{\uparrow \\ \text{Sink}}}{b}), (a, c), (c, b) \}$

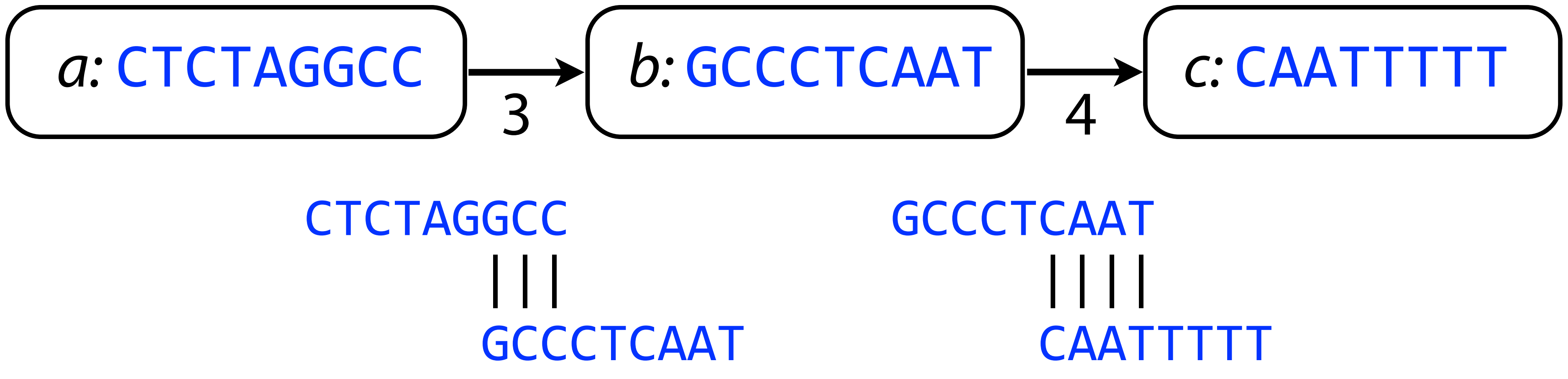
Overlap graph

Below: overlap graph, where an overlap is a suffix/prefix match of at least 3 characters

A vertex is a read, a directed edge is an overlap between suffix of source and prefix of sink

Vertices (reads): { *a:* CTCTAGGCC, *b:* GCCCTCAAT, *c:* CAATTTTT }

Edges (overlaps): { (*a*, *b*), (*b*, *c*) }

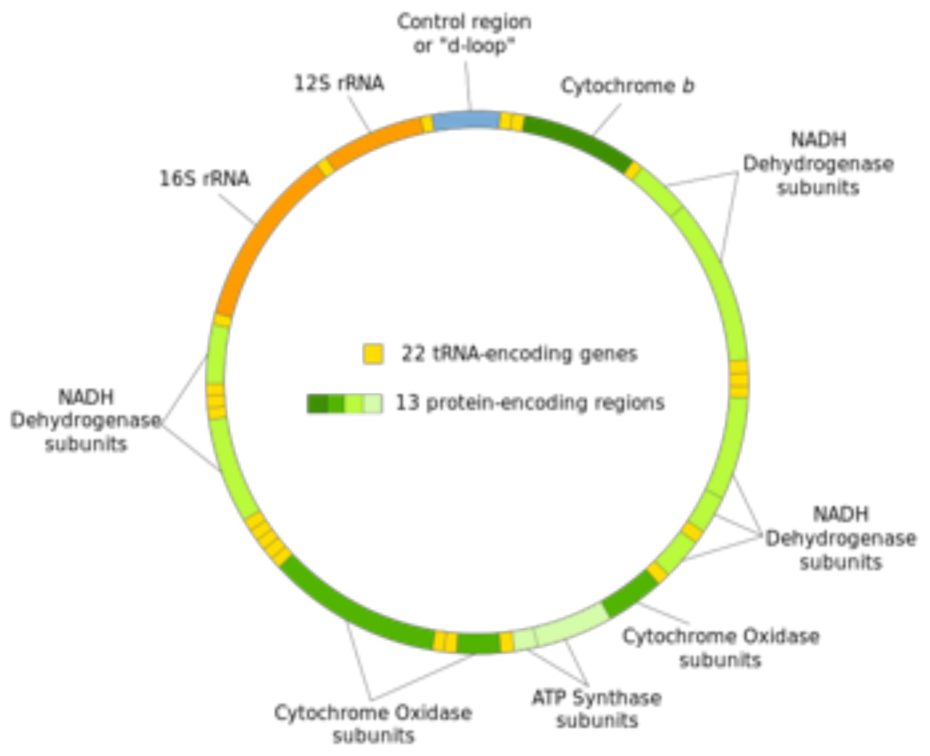


Overlap graph

Overlap graph could contain *cycles*. A cycle is a path beginning and ending at the same vertex.

$$\begin{tikzcd}
\boxed{\textit{a:}\ \texttt{CTCTAGGCC}} \arrow[r, "3"'] & \boxed{\textit{b:}\ \texttt{GCCCTCACT}} \arrow[r, "4"'] & \boxed{\textit{c:}\ \texttt{CACTCTAGG}} \arrow[ll, bend right, "7"']
\end{tikzcd}$$

These happen when the DNA string itself is circular. E.g. bacterial genomes are often circular; mitochondrial DNA is circular.



Control region or "d-loop"
12S rRNA
Cytochrome *b*
16S rRNA
NADH Dehydrogenase subunits
22 tRNA-encoding genes
13 protein-encoding regions
NADH Dehydrogenase subunits
NADH Dehydrogenase subunits
Cytochrome Oxidase subunits
ATP Synthase subunits
Cytochrome Oxidase subunits

Cycles could also be due to *repetitive* DNA, as we'll see

Finding overlaps

$$\begin{tikzcd}\boxed{a\text{: CTCTAGGCC}} \arrow[r] & \boxed{b\text{: GCCCTCAAT}} \arrow[r] & \boxed{c\text{: CAATTTTT}}\end{tikzcd}$$

How do we build the overlap graph?

What constitutes an overlap?

Assume for now an "overlap" is when a suffix of *X* of length ≥ l exactly matches a prefix of *Y*, where *k* is given

Finding overlaps

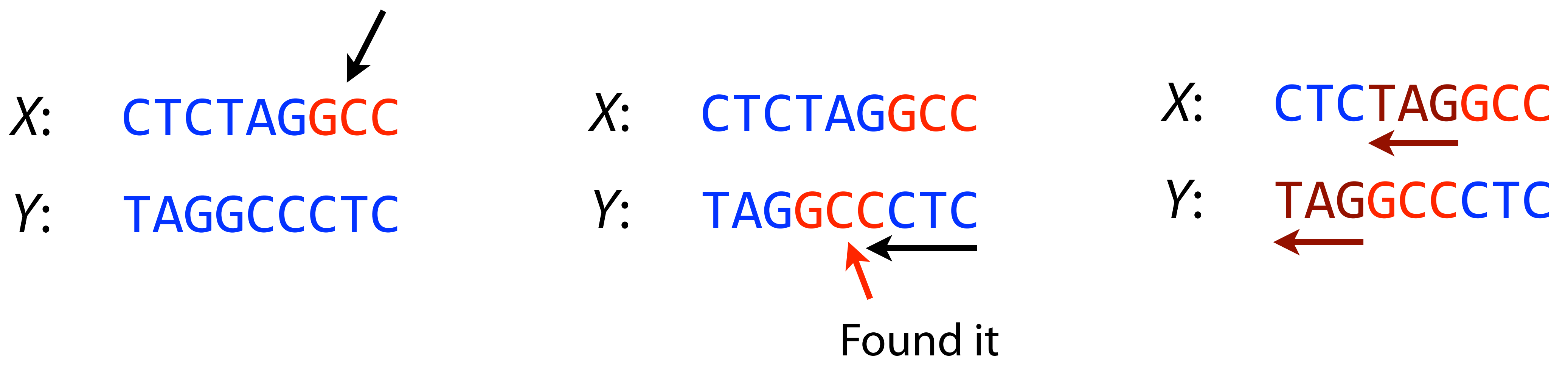
Overlap: length-l suffix of X matches length-l prefix of Y, where l is given

Simple idea: look in Y for occurrences of length-l suffix of X. Extend matches to the left to confirm whether entire prefix of Y matches.

Say $k = 3$

Look for this in Y, going right-to-left

Extend to left; in this case, we confirm that a length-6 prefix of Y matches a suffix of X



Finding overlaps: implementation

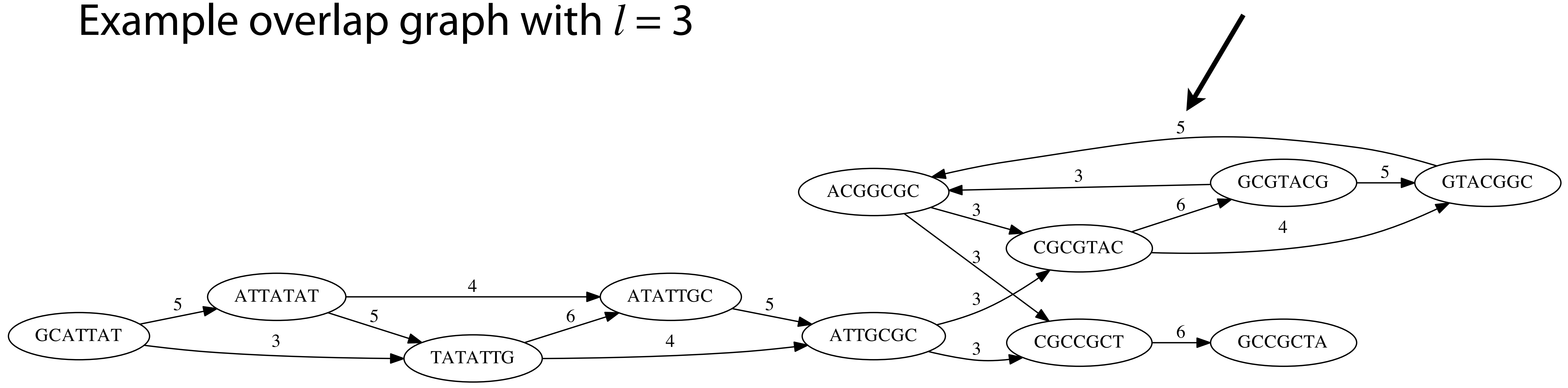
```
def suffixPrefixMatch(x, y, k):
    ''' Return length of longest suffix of x of length at least k that
        matches a prefix of y.  Return 0 if there no suffix/prefix
        match has length at least k. '''
    if len(x) < k or len(y) < k:
        return 0
    idx = len(y) # start at the right end of y
    # Search right-to-left in y for length-k suffix of x
    while True:
        hit = string.rfind(y, x[-k:], 0, idx)
        if hit == -1: # not found
            return 0
        ln = hit + k
        # See if match can be extended to include entire prefix of y
        if x[-ln:] == y[:ln]:
            return ln # return length of prefix
        idx = hit + k - 1 # keep searching to left in Y
    return -1
```

Python example: http://nbviewer.ipython.org/7089885

Finding overlaps

Edge label is overlap length

Example overlap graph with *l* = 3



Original string: GCATTATATATTGCGCGTACGGCGCCGCTACA

Formulating the assembly problem

Finding overlaps is important, and we'll return to it, but our ultimate goal is to recreate (assemble) the genome

How do we formulate this problem?

First attempt: the *shortest common superstring* (*SCS*) problem

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: $\underbrace{\text{BAAAABBBAABAABBBBBAAABAB}}_{24}$

SCS(S): $\underbrace{\text{AAABBBABAA}}_{10}$

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

Shortest common superstring

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!

S: AAA AAB ABB BBB BBA

SCS(S): AAABBBA

```
AAA
 AAB
  ABB
   BBB
    BBA
```

$$\begin{tikzcd}
 & & AAB \arrow[drr, "-2"] \arrow[ddl, "-1"'] \arrow[ddr, bend left=10, "-1"] & & \\
AAA \arrow[urr, "-2"] \arrow[rrrr, "-1"'] & & & & ABB \arrow[dlll, "-2"'] \arrow[dl, bend left=10, "-2"'] \\
 & BBB \arrow[rr, "-2"] & & BBA \arrow[uul, bend left=10, "-1"] \arrow[ur, bend left=10, "-2"'] \arrow[ulll, "-1"] &
\end{tikzcd}$$

Shortest common superstring

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

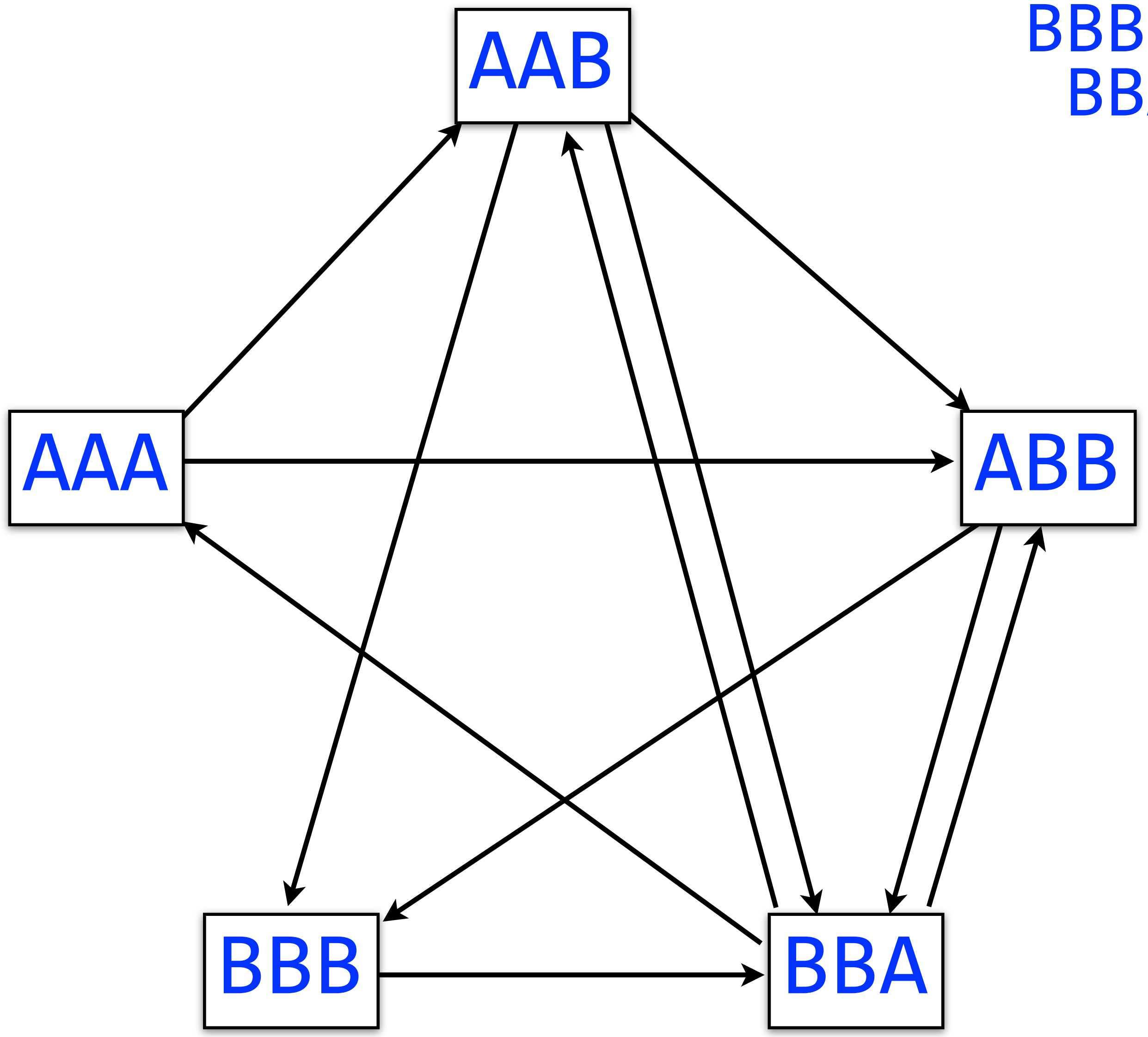
That's the *Hamiltonian Path* problem: NP-complete

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

S: AAA AAB ABB BBB BBA

SCS(S): AAABBBA

```
AAA
 AAB
  ABB
   BBB
    BBA
```



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)

Shortest common superstring

Let’s take the hint give up on finding the *shortest possible* superstring

Non-optimal superstrings can be found with a *greedy* algorithm

At each step, the greedy algorithm “greedily” chooses longest remaining overlap, merges its source and sink

Shortest common superstring: greedy

Greedy-SCS algorithm in action ($l = 1$):

```
     |——— Input strings ———|
     ABA  ABB  AAA  AAB  BBB  BBA  BAB  BAA
  2  BAAB  ABA  ABB  AAA  BBB  BBA  BAB
  2  BABB  BAAB  ABA  AAA  BBB  BBA
  2  BBAAB  BABB  ABA  AAA  BBB
  2  BBBAAB  BABB  ABA  AAA
  2  BBBAABA  BABB  AAA
  2  BABBBAABA  AAA
  1  BABBBAABAAA
     BABBBAABAAA
     |— Superstring —|
```

In red are strings that get merged before the next round

Greedy answer:
BABBBAABAAA

Actual SCS:
AAABBBABAA

Rounds of merging, one merge per line.
Number in first column = length of overlap merged before that round.

Shortest common superstring: greedy

Greedy algorithm is *not* guaranteed to choose overlaps yielding SCS

But greedy algorithm is a good *approximation*; i.e. the superstring yielded by the greedy algorithm won't be more than ~2.5 times longer than true SCS (see Gusfield 16.17.1)

Shortest common superstring: greedy

Greedy-SCS algorithm in action again ($l = 3$):

```
  |—————————————— Input strings ——————————————|
  ATTATAT CGCGTAC ATTGCGC GCATTAT ACGGCGC TATATTG GTACGGC GCGTACG ATATTGC
6 TATATTGC ATTATAT CGCGTAC ATTGCGC GCATTAT ACGGCGC GTACGGC GCGTACG
6 CGCGTACG TATATTGC ATTATAT ATTGCGC GCATTAT ACGGCGC GTACGGC
5 CGCGTACG TATATTGCGC ATTATAT GCATTAT ACGGCGC GTACGGC
5 CGCGTACGGC TATATTGCGC ATTATAT GCATTAT ACGGCGC
5 CGCGTACGGCGC TATATTGCGC ATTATAT GCATTAT
5 CGCGTACGGCGC GCATTATAT TATATTGCGC
5 CGCGTACGGCGC GCATTATATTGCGC
3 GCATTATATTGCGCGTACGGCGC
  GCATTATATTGCGCGTACGGCGC
  |—— Superstring ——|
```

Shortest common superstring: greedy

Another setup for Greedy-SCS: assemble all substrings of length 6 from string a_long_long_long_time. $l = 3$.

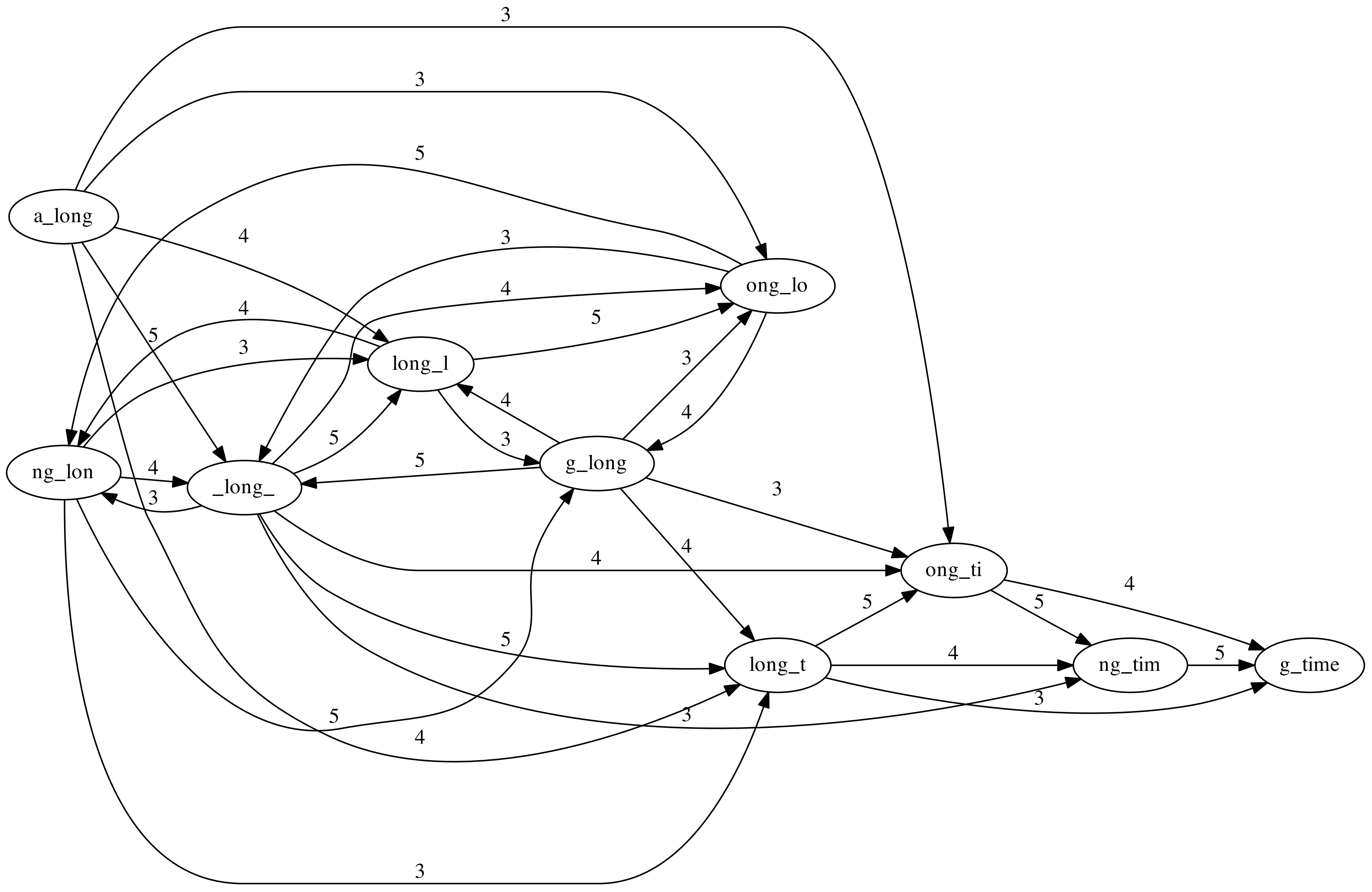
```
  ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
5 ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
5 ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
5 ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
5 ng_time ong_lon long_ti g_long_ a_long long_l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

I only got back: a_long_long_time (missing a _long)

What happened?

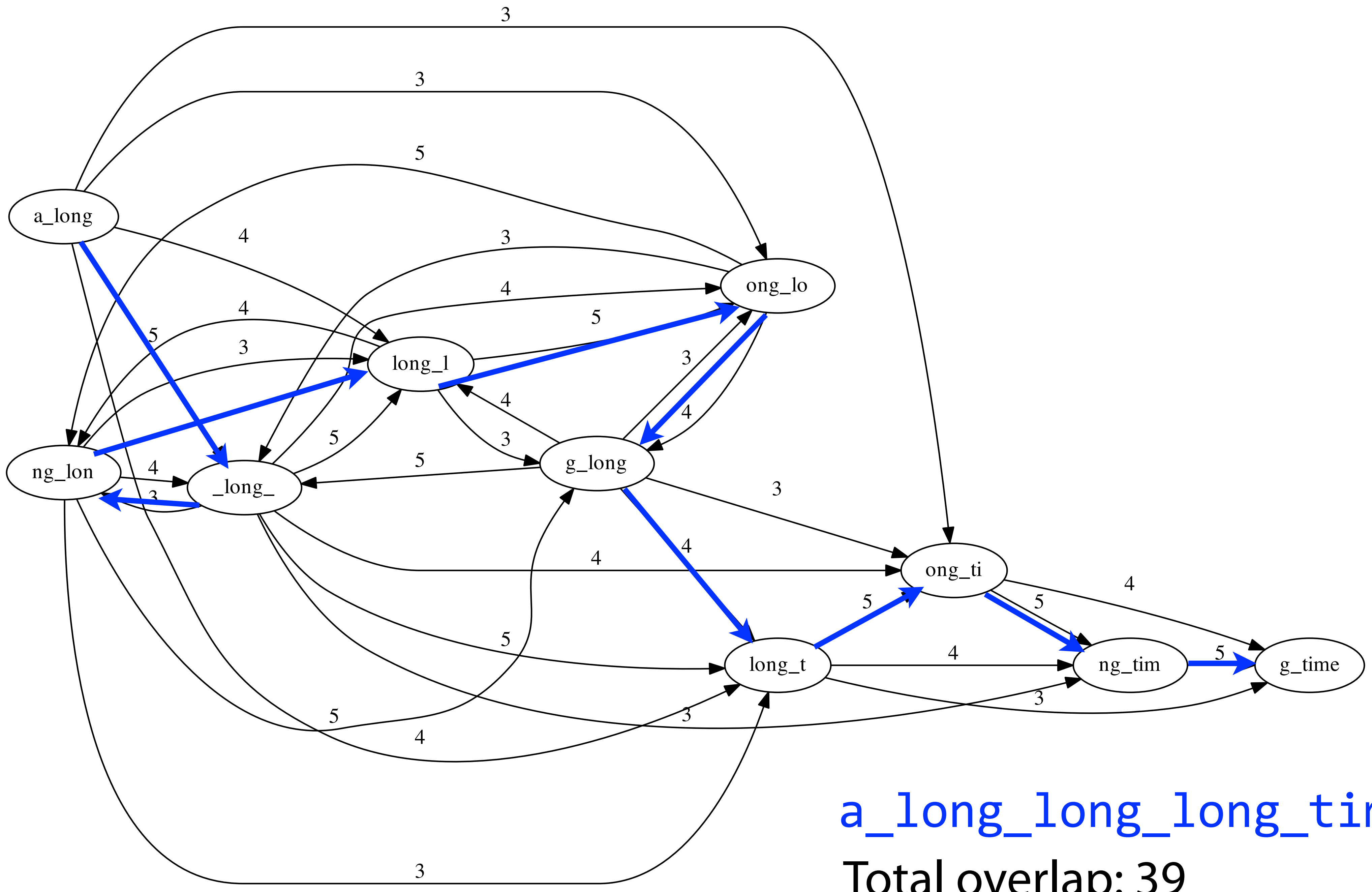
Shortest common superstring: greedy

The overlap graph for that scenario ($l = 3$):



Shortest common superstring: greedy

The overlap graph for that scenario ($l = 3$):

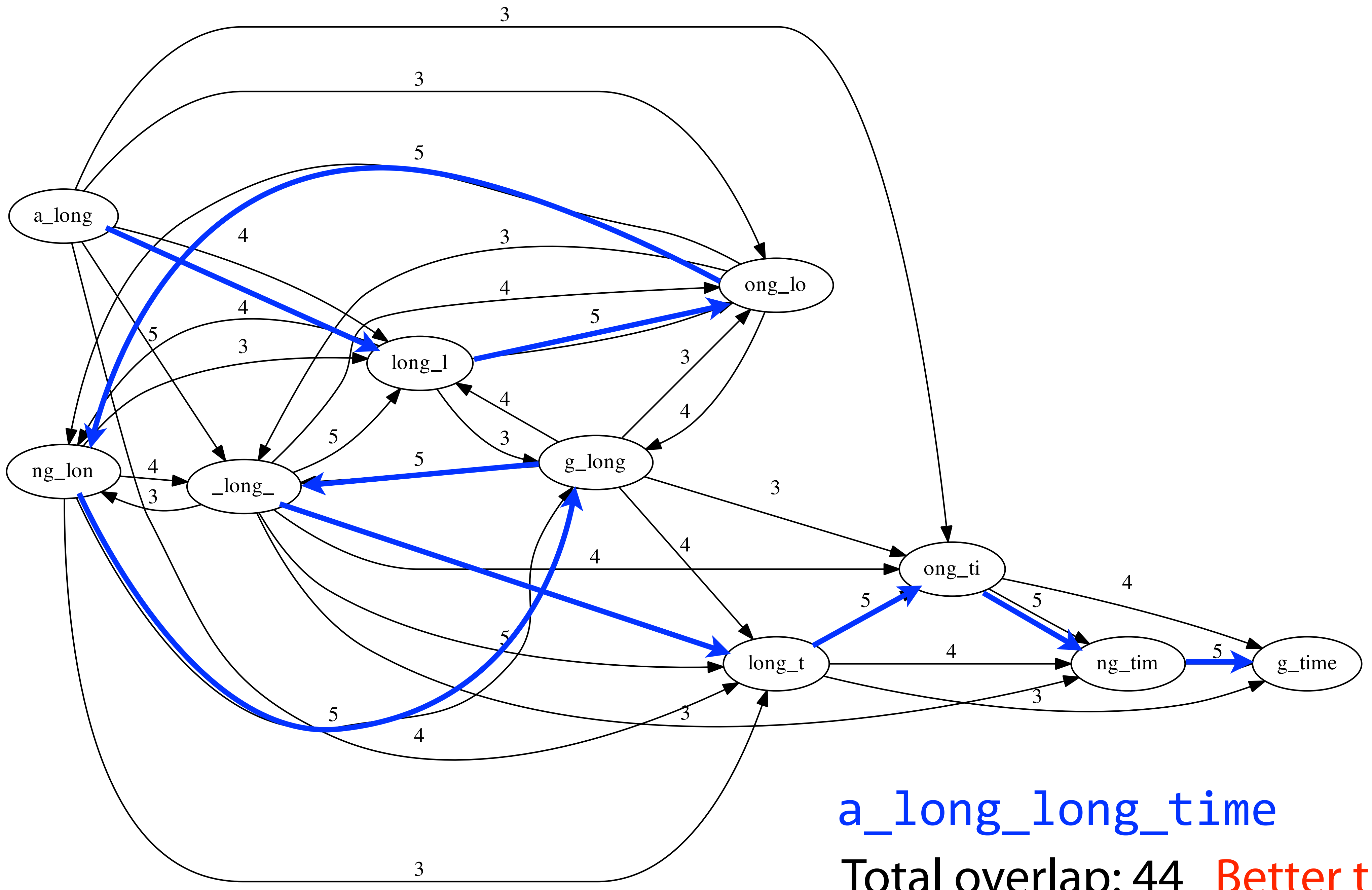


a_long_long_long_time

Total overlap: 39

Shortest common superstring: greedy

The overlap graph for that scenario ($l = 3$):



a_long_long_time

Total overlap: 44 Better than the correct path!

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

Got the whole thing: `a_long_long_long_time`

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

Repeats

Repeats often foil assembly. They certainly foil SCS, with its "shortest" criterion!

Reads might be too short to "resolve" repetitive sequences. This is why sequencing vendors try to increase read length.

Algorithms that don't pay attention to repeats (like our greedy SCS algorithm) might *collapse* them

$$\begin{tikzcd}
\texttt{a\_long\_long\_long\_time} \arrow[d, "\textit{collapse}"] \\
\texttt{a\_long\_long\_time}
\end{tikzcd}$$

The human genome is ~ 50% repetitive!

Repeats

Basic principle: *repeats foil assembly*

Another example using Greedy-SCS:

Input: `it_was_the_best_of_times_it_was_the_worst_of_times`

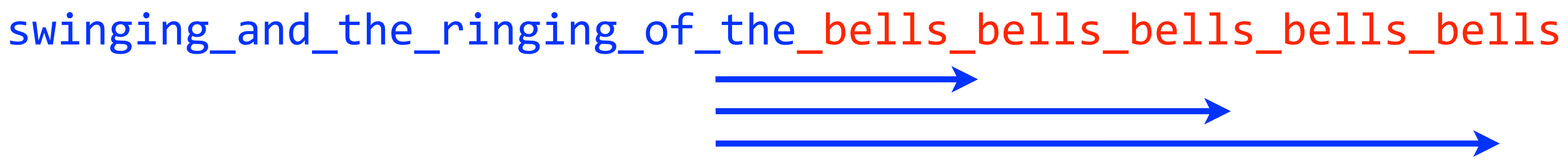
Extract every substring of length k, then run Greedy-SCS.
Do this for various l (min overlap length) and k.

l, k	output
3, 5	`the_worst_of_times_it_was_the_best_o`
3, 7	`s_the_worst_of_times_it_was_the_best_of_t`
3, 10	`_was_the_best_of_times_it_was_the_worst_of_tim`
3, 13	`it_was_the_best_of_times_it_was_the_worst_of_times`

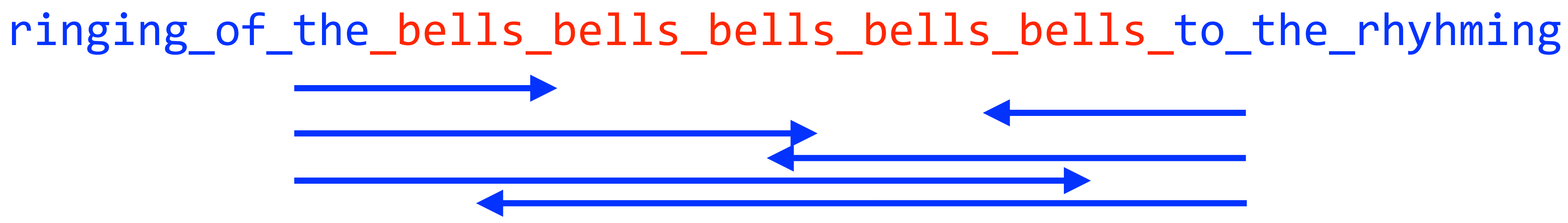
Repeats

Basic principle: *repeats foil assembly*

Longer and longer substrings allow us to “anchor” more of the repeat to its non-repetitive context:


swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells

Often we can “walk in” from both sides. When we meet in the middle, the repeat is resolved:


ringing_of_the_bells_bells_bells_bells_bells_to_the_rhyhming

Repeats

Basic principle: *repeats foil assembly*

Yet another example using Greedy-SCS:

Input: `swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells`

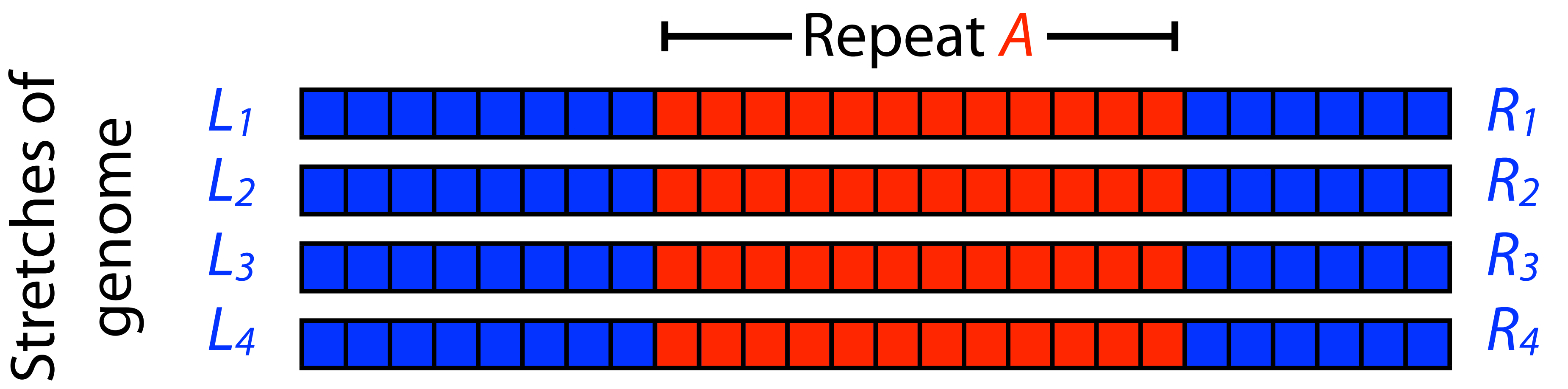
l, k	output
3, 7	`swinging_and_the_ringing_of_the_bells_bells`
3, 13	`swinging_and_the_ringing_of_the_bells_bells_bells`
3, 19	`swinging_and_the_ringing_of_the_bells_bells_bells_bells_b`
3, 25	`swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells`

$\longrightarrow$

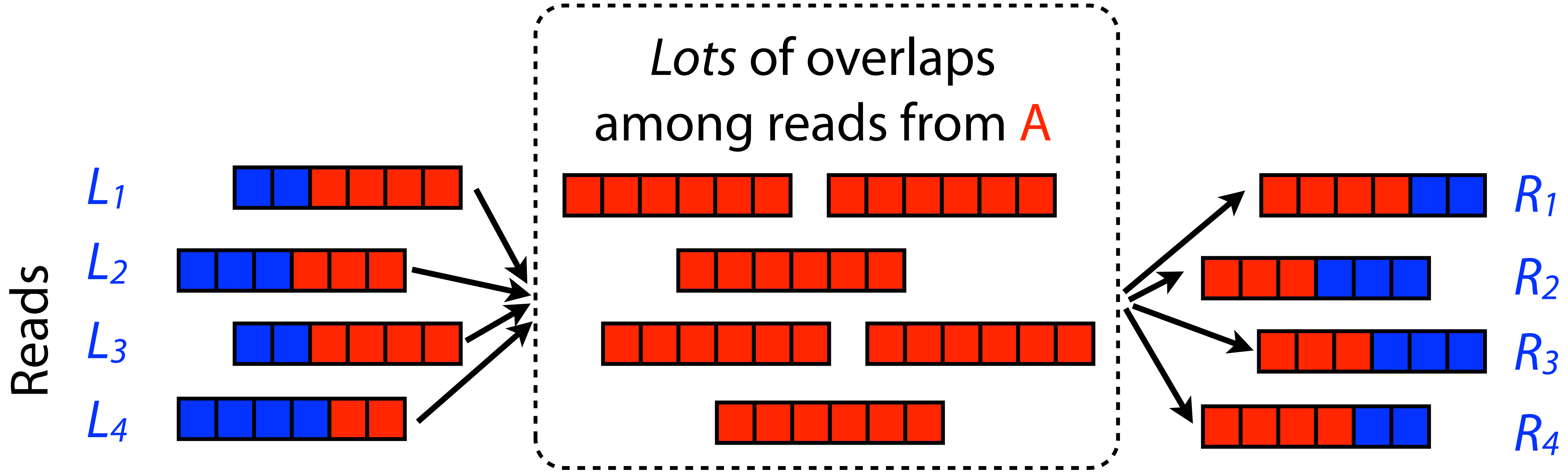
longer and longer substrings allow us to "reach" further into the repeat

Repeats

Picture the portion of the overlap graph involving repeat *A*



Assume *A* is longer than read length



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

Shortest common superstring: post mortem

SCS is flawed as a way of formulating the assembly problem

- No tractable way to find optimal SCS
  - Had to use Greedy-SCS. Answers might be too long.
- SCS spuriously collapses repetitive sequences
  - Answers might be too short, by a lot!

Need formulations that are (a) tractable, and (b) handle repeats as gracefully as possible

Remember: repeats foil assembly no matter the algorithm. This is a property of read length and repetitiveness of the genome.

Taxonomy of assembly approaches

Search for most parsimonious explanation of the reads (shortest superstring)

> Exact solutions are intractable (e.g. TSP), but a greedy approximation is possible
>
> Any solution will collapse repeats spuriously

Search for "maximum likelihood" explanation of the reads; i.e. force solution to be consistent with uniform coverage

> *No* solutions (that I know of) are tractable

Give up on unresolvable repeats and use a tractable algorithm to assemble the resolvable portions. **This is what real tools do.**