String Algorithms and Data Structures

String Graph Assembly

CS 199-225
Brad Solomon

April 18, 2022

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Assignment 11: a_edist due April 18 11:59 PM!

Last assignment!

String Assembly

READING THE BOOK OF LIFE: THE OVERVIEW

READING THE BOOK OF LIFE: THE OVERVIEW; Genetic Code of Human Life Is Cracked by Scientists

By NICHOLAS WADE

Published: June 27, 2000

Human Genome Project: 1990-2003

The New York Times | http://nyti.ms/1tcvLXq

SCIENCE

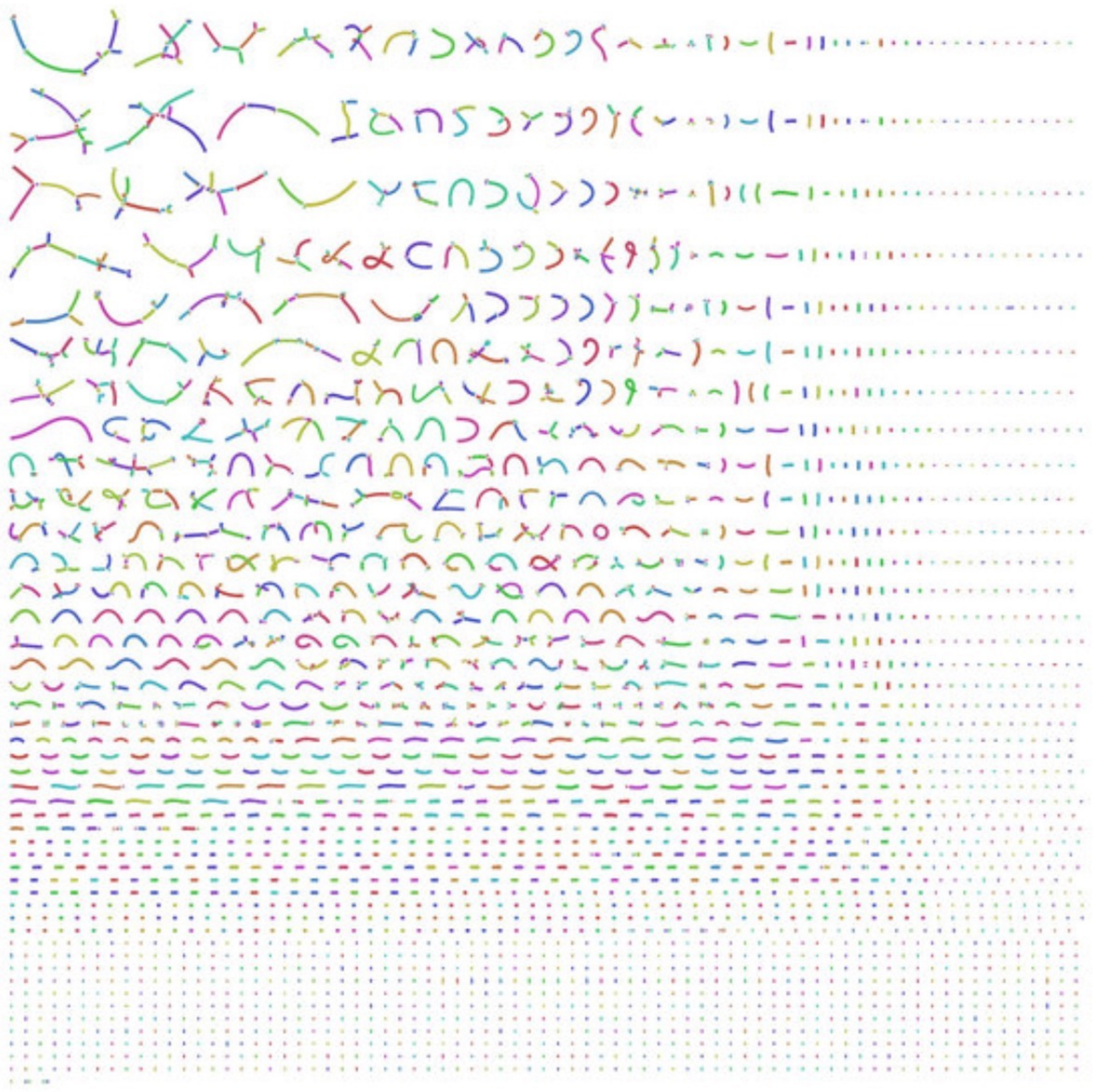
Man’s Genome From 45,000 Years Ago Is Reconstructed

OCT. 22, 2014

Carl Zimmer

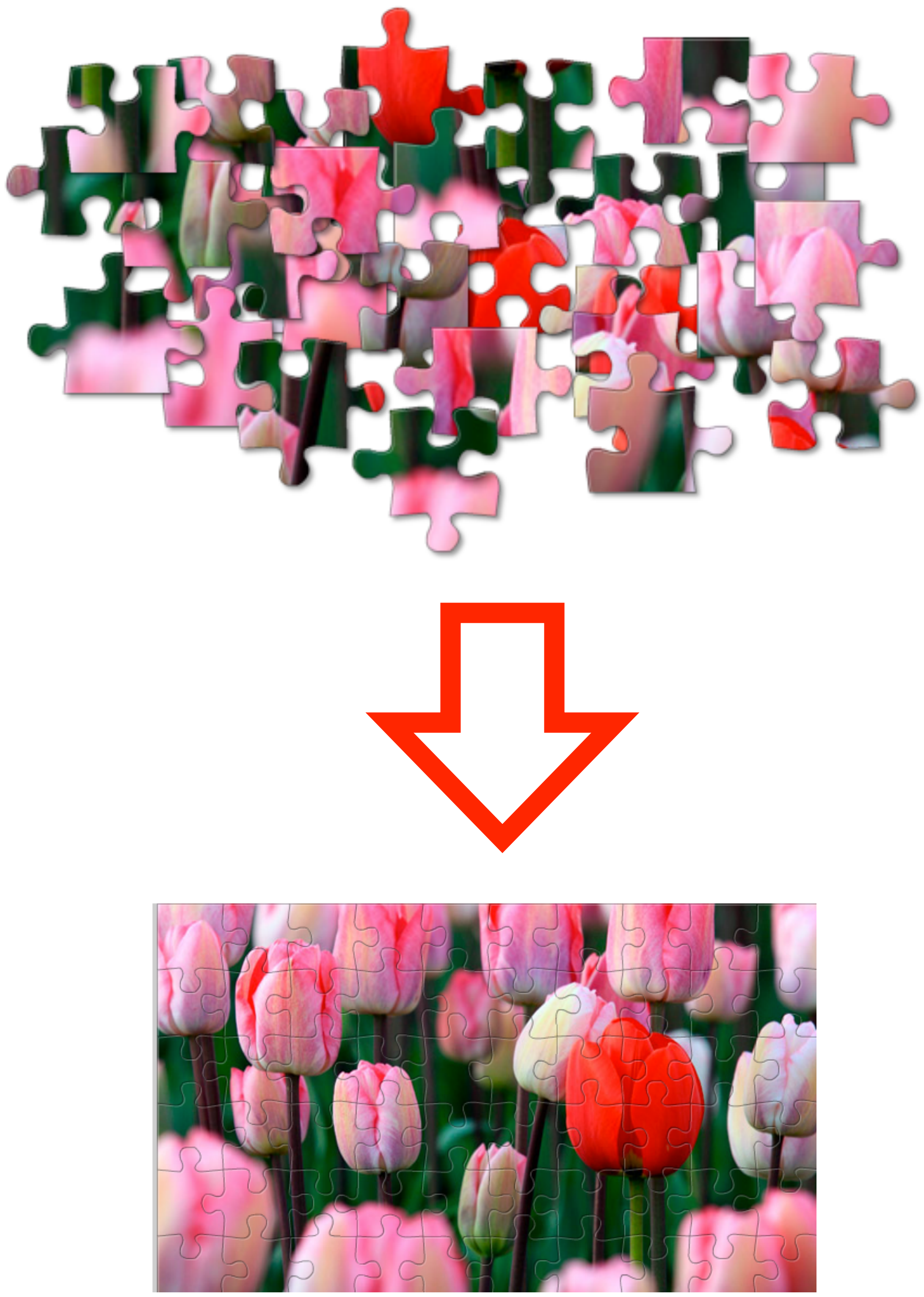
Team of Rival Scientists Comes Together to Fight Zika

By **AMY HARMON** MARCH 30, 2016



A visualization of the recently sequenced Aedes aegypti genome. Each of the 3,752 colored lines is a fragment of its three chromosomes that could not be fit together without the additional information that the Aedes Genome Working Group hopes to produce. A 2007 genome map for Aedes aegypti is fragmented into about 10 times as many pieces. Mark Kunitomi

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
```

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

String Assembly

From these

CTAGGCCCTCAATTTTT

CTCTAGGCCCTCAATTTTT

GGCTCTAGGCCCTCATTTTTT

CTCGGCTCTAGCCCCTCATTTT

TATCTCGACTCTAGGCCCTCA

TATCTCGACTCTAGGCC

TCTATATCTCGGCTCTAGG

GGCGTCTATATCTCG

GGCGTCGATATCT

GGCGTCTATATCT

Reconstruct this

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

String Assembly

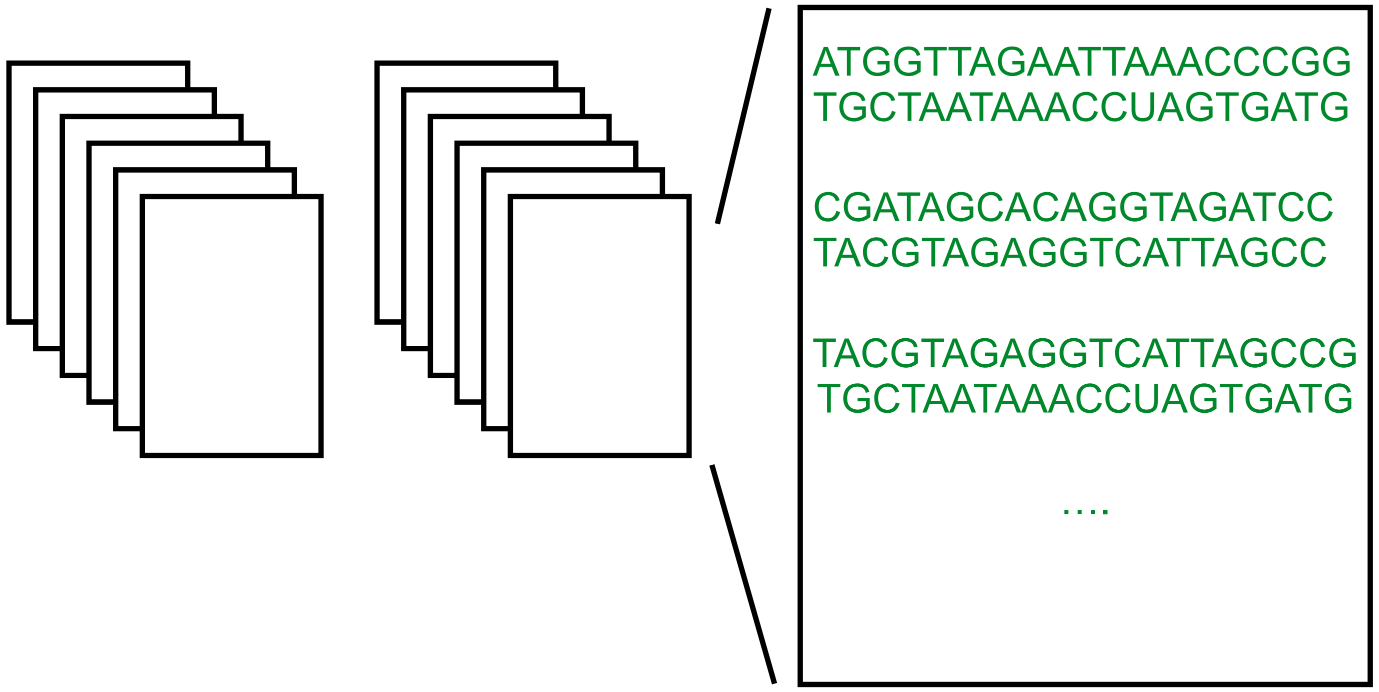
CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

From these

Reconstruct this

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

String Assembly

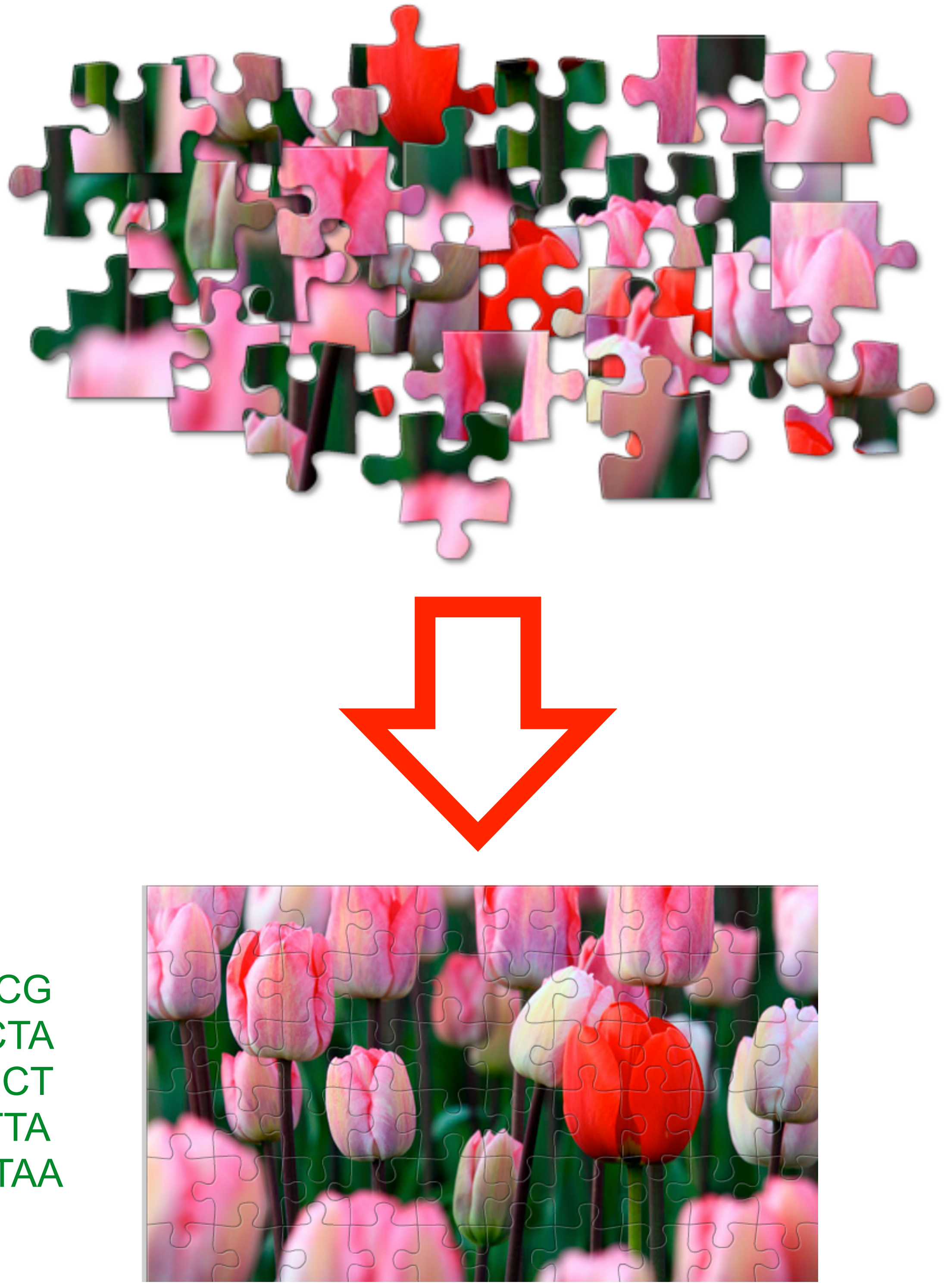


ATGGTTAGAATTAAACCCGG
TGCTAATAAACCUAGTGATG

CGATAGCACAGGTAGATCC
TACGTAGAGGTCATTAGCC

TACGTAGAGGTCATTAGCCG
TGCTAATAAACCUAGTGATG

....



ATGGTTAGAATTAAACCTGGATCTGCTAATAAACCUAGTGATGATGCG
ATAGCACAGGTAGATCCAGTTACGTAGAGGTCATTAGCCGTATTGCTA
ATAAACCTAGTGATGATTCGATAGCGTAGAGGTCATTAGCCTTGTGCT
AATAACAGGTAGATCCGTATACGTAGAGGTCATTACCAGAGGTCATTA
GTTGTGCTAATAAACCTAGTGTAGATGAAGAGGTCATTAGATCTGCTAA

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

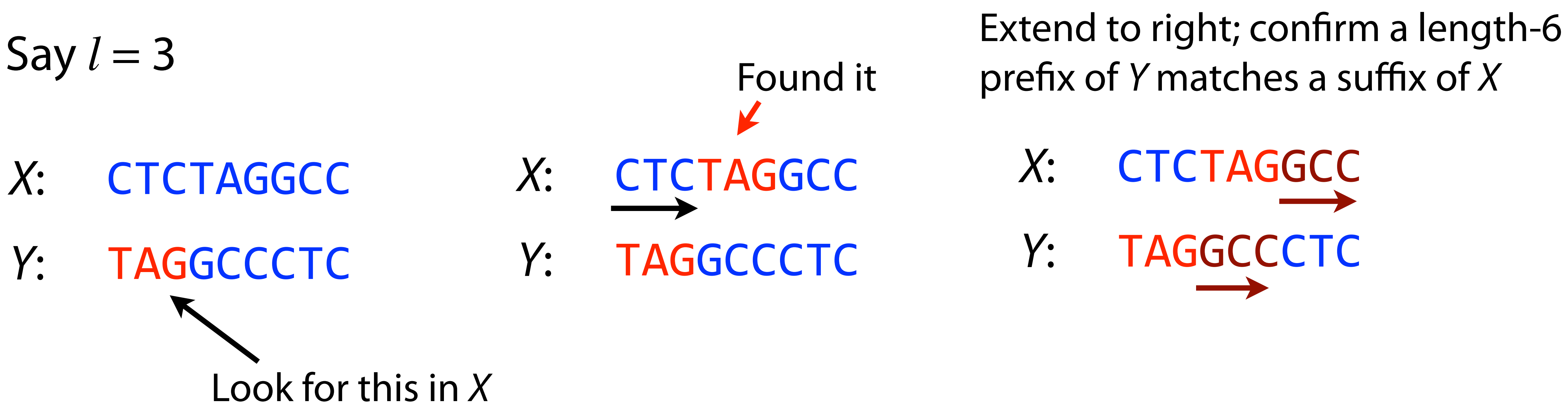
3) Reconstruct T based on consensus

Identify Overlaps

Length-l Overlap: Suffix of *X* of length $\geq l$ matches prefix of *Y*

Naive: look in *X* for occurrences of *Y*'s length-l prefix. Extend matches to the right to confirm if the suffix of *X* matches.

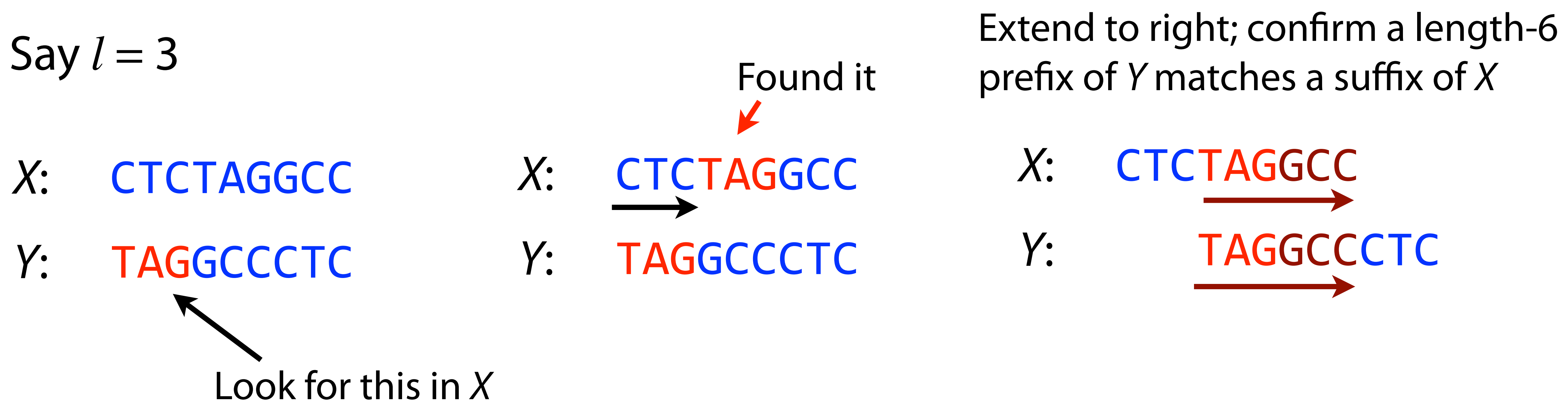
Say $l = 3$



Identify Overlaps

Length-l Overlap: Suffix of *X* of length $\geq l$ matches prefix of *Y*

Naive: look in *X* for occurrences of *Y*'s length-l prefix. Extend matches to the right to confirm if the suffix of *X* matches.

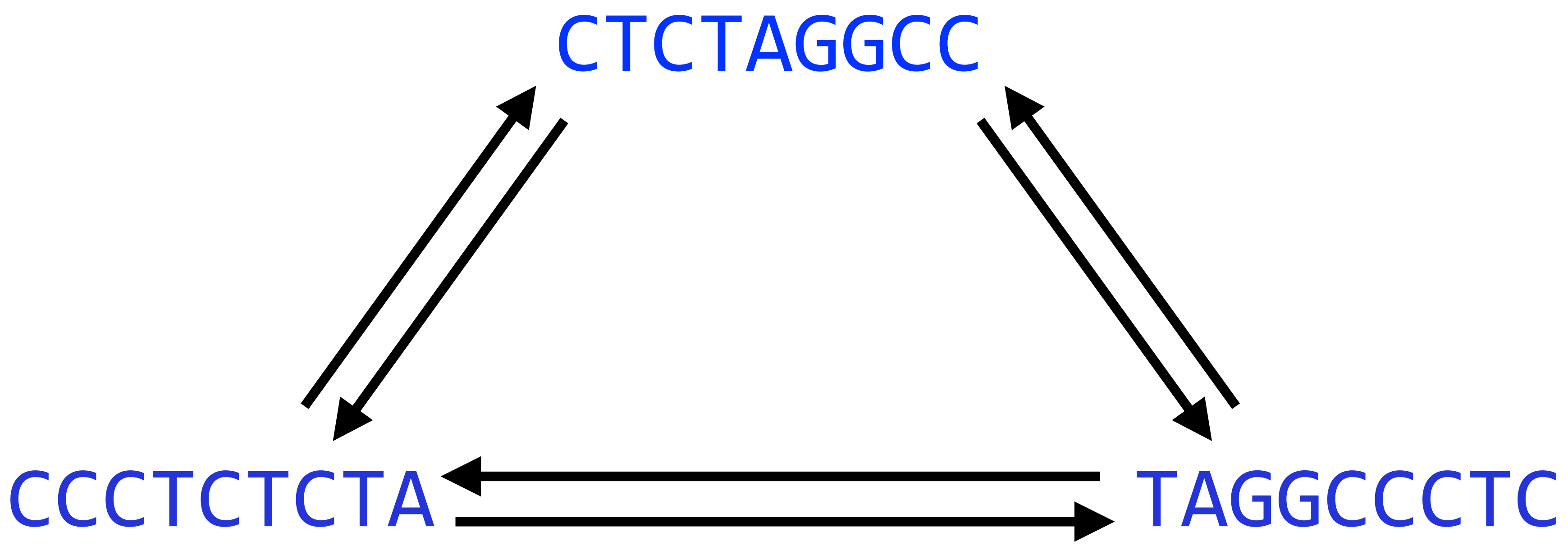


Identify Overlaps

Length-l Overlap: Suffix of *X* of length $\geq l$ matches prefix of *Y*

Naive: look in *X* for occurrences of *Y*'s length-l prefix. Extend matches to the right to confirm if the suffix of *X* matches.

For three strings, how many overlaps must be calculated?



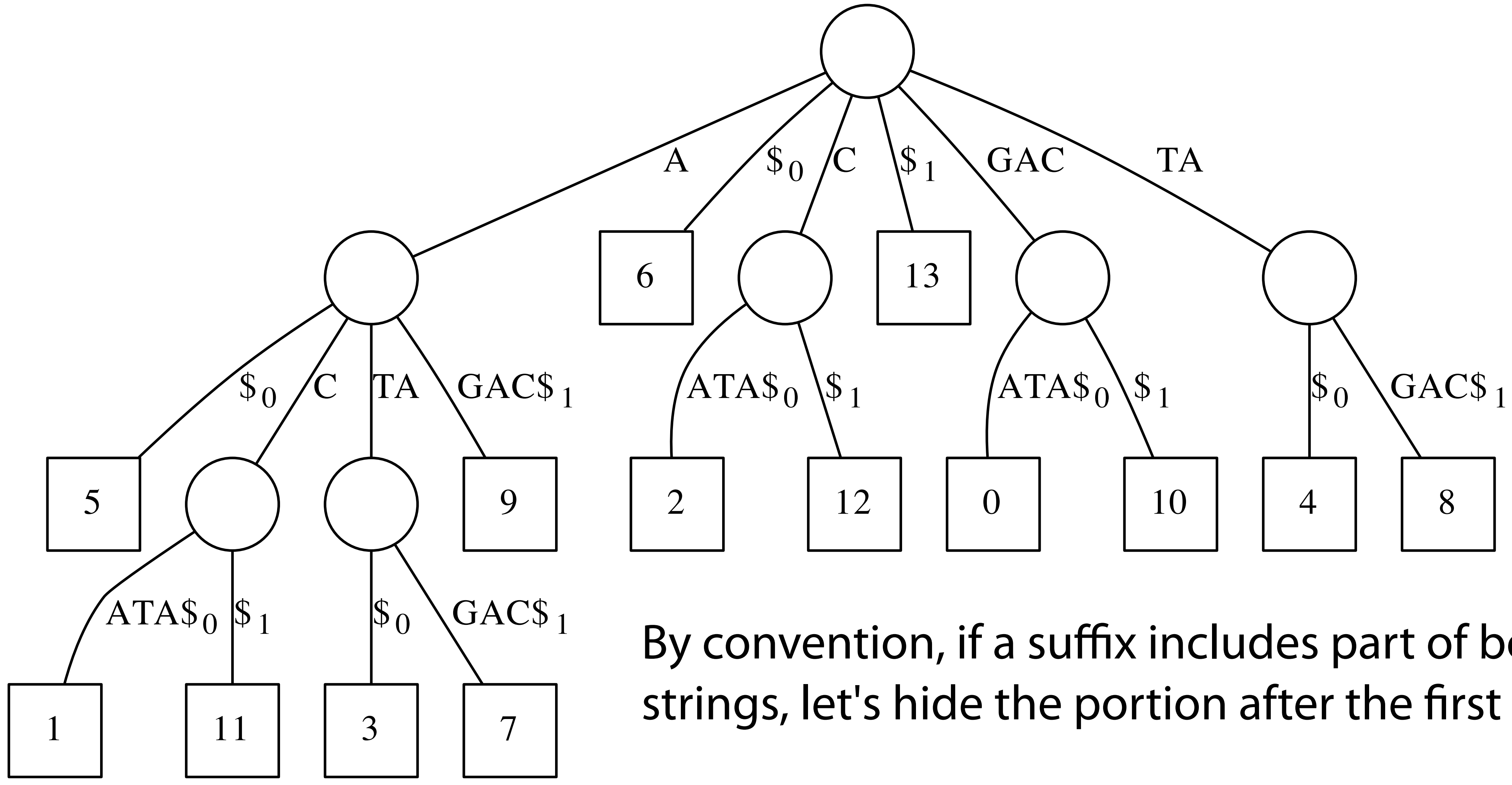
In bulk, there are better ways to do this…

Identify Overlaps: Generalized Suffix Tree

To build a suffix tree from two strings ***X*** and ***Y***, make a new string ***X*$_0$*Y*$_1$** where **$_0$, $_1$** are both terminal symbols. Build a suffix tree for ***X*$_0$*Y*$_1$** .

Generalized suffix tree for { "GACATA", "ATAGAC" }

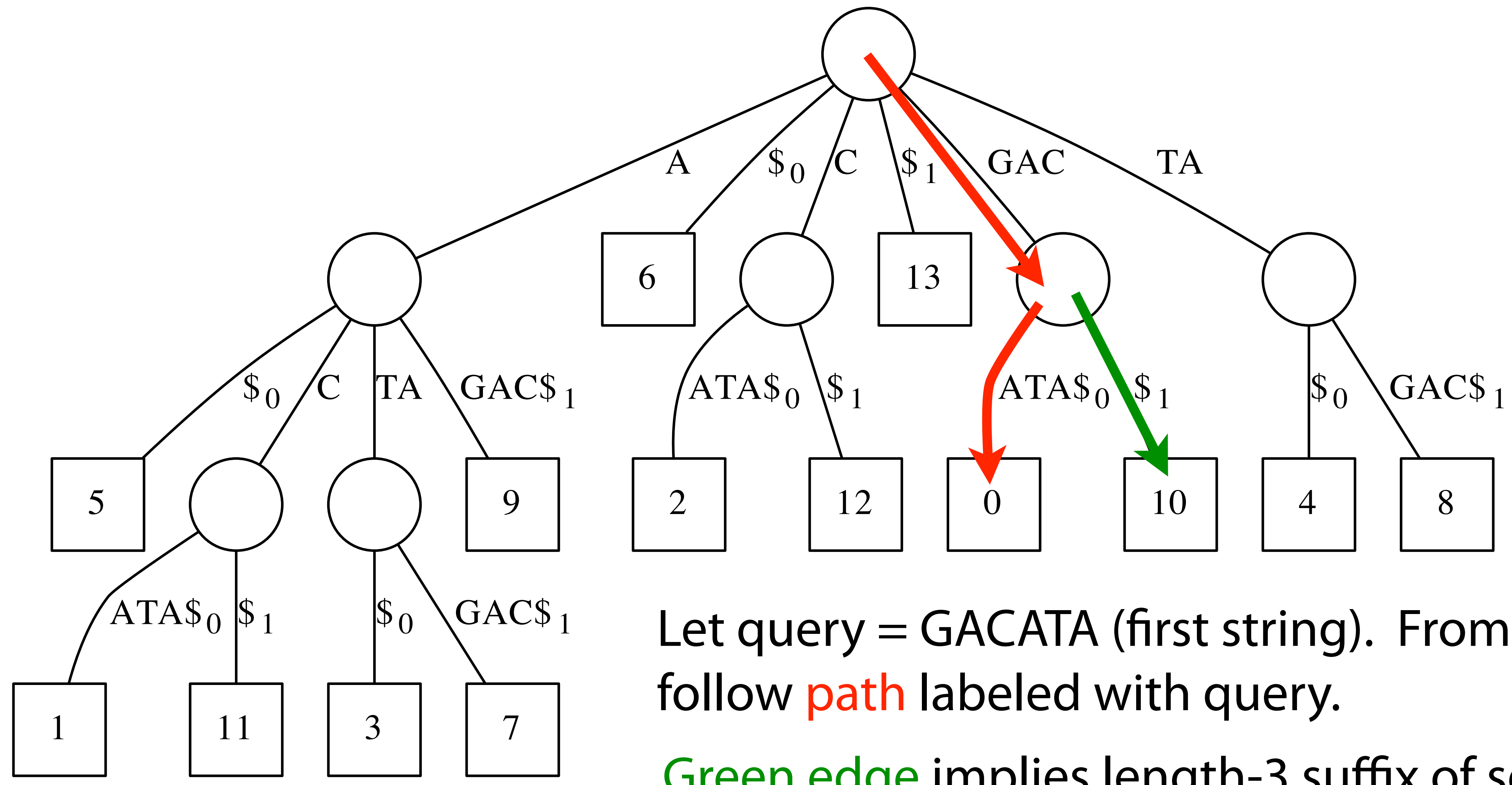
GACATA$_0$ATAGAC$_1$



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

Identify Overlaps: Generalized Suffix Tree

Generalized suffix tree for { "GACATA", "ATAGAC" } $GACATA\$_0ATAGAC\$_1$



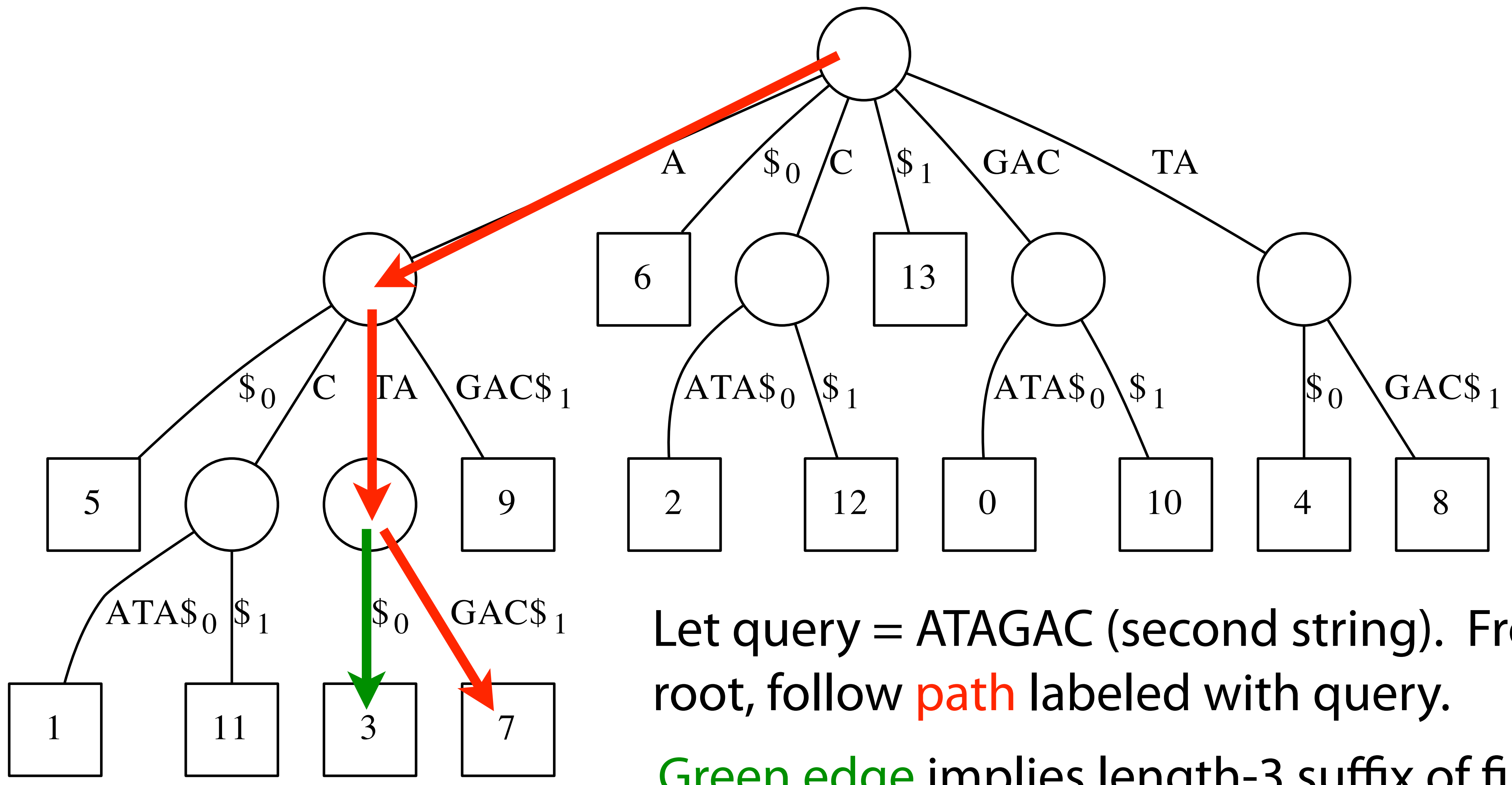
Let query = GACATA (first string). From root, follow path labeled with query.

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

```
ATAGAC
   |||
   GACATA
```

Identify Overlaps: Generalized Suffix Tree

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



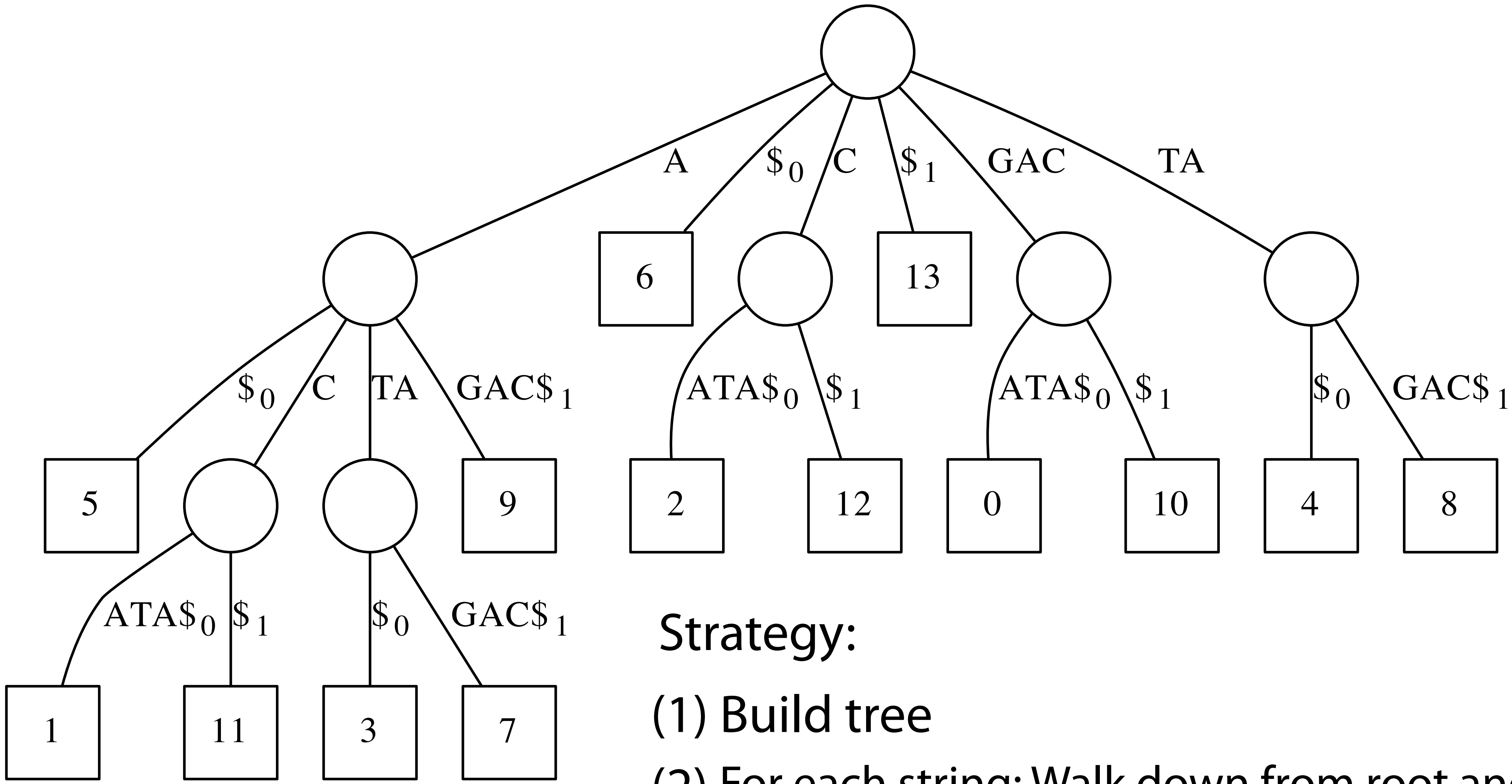
```
GACATA
   |||
   ATAGAC
```

Let query = ATAGAC (second string). From root, follow path labeled with query.

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

Identify Overlaps: Generalized Suffix Tree

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

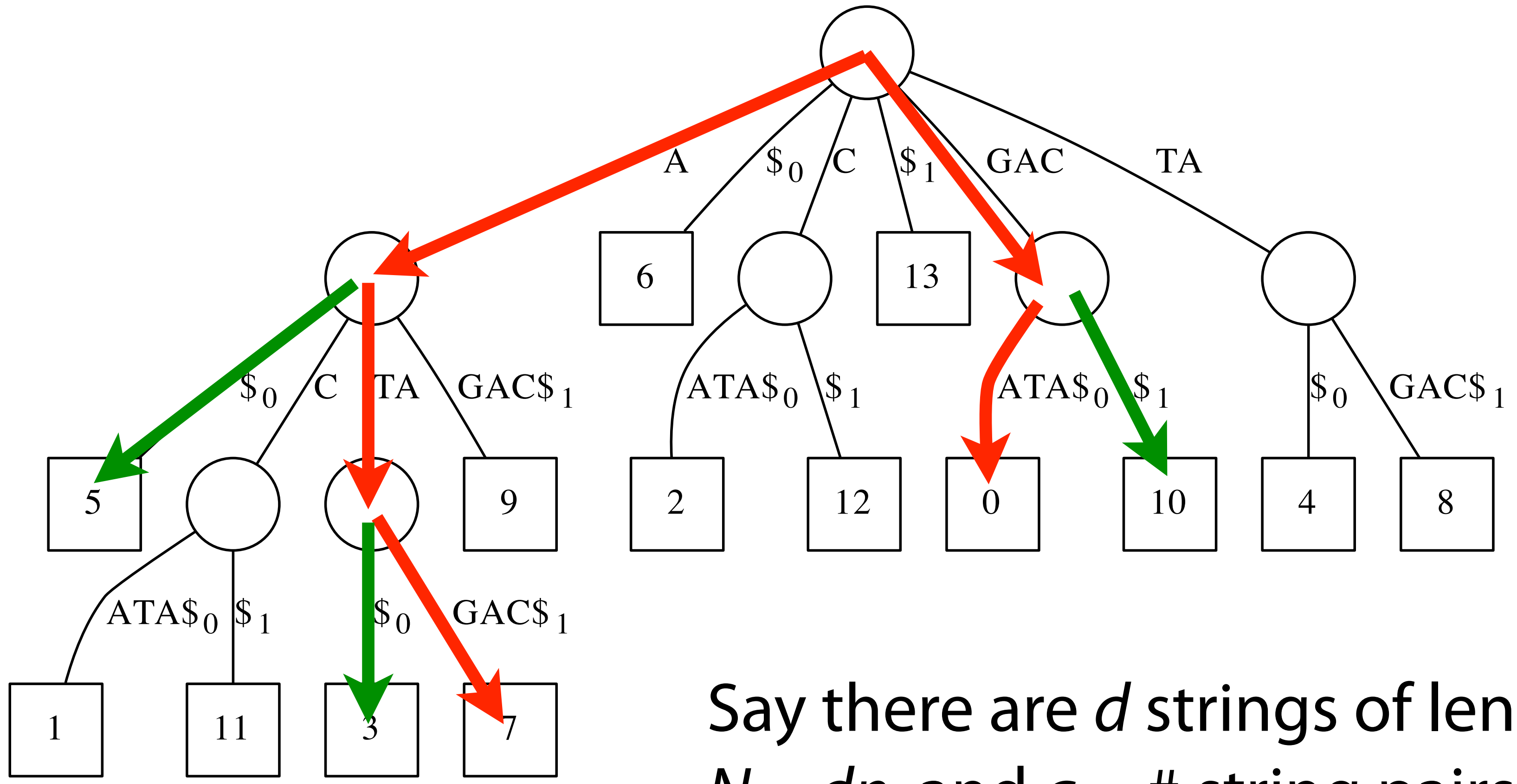


Strategy:

(1) Build tree

(2) For each string: Walk down from root and report any outgoing edge labeled with a separator. Each corresponds to a prefix/suffix match involving prefix of query string and suffix of string ending in the separator.

Identify Overlaps: Generalized Suffix Tree



Say there are *d* strings of length *n*, total length *N* = *dn*, and *a* = # string pairs that overlap

Assume for given string pair we report only the longest suffix/prefix match

Time to build generalized suffix tree:	O(*N*)
... to walk down red paths:	O(*N*)
... to find & report overlaps (green):	O(*a*)
Overall:	O(*N* + *a*)

Identify Overlaps: Dynamic Programming

What about *approximate* suffix/prefix matches?

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

Use *approximate matching* recurrence relationship

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1]+\delta(x[i-1],y[j-1]) \end{cases}$$

How do we search for prefix / suffix matches between X and Y?

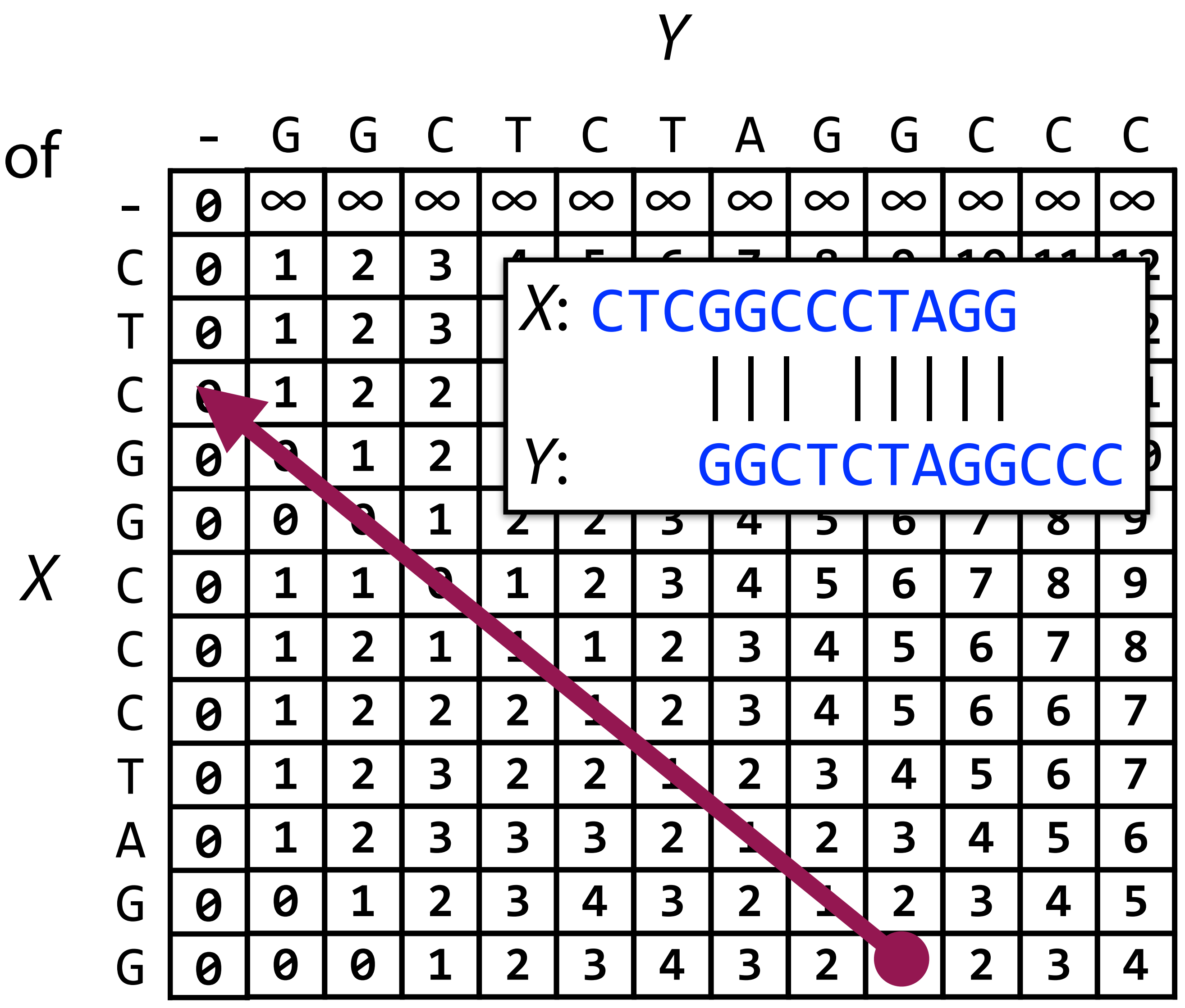
Identify Overlaps: Dynamic Programming

How to adjust our matrix so suffix of *X* aligns to prefix of *Y*?

First column gets 0s

First row gets ∞s

Backtrace from last row



	-	G	G	C	T	C	T	A	G	G	C	C	C
-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	0	1	2	3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
T	0	1	2	3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
C	0	1	2	2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
G	0	0	1	2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
G	0	0	0	1	2	2	3	4	5	6	7	8	9
C	0	1	1	0	1	2	3	4	5	6	7	8	9
C	0	1	2	1	1	1	2	3	4	5	6	7	8
C	0	1	2	2	2	1	2	3	4	5	6	6	7
T	0	1	2	3	2	2	1	2	3	4	5	6	7
A	0	1	2	3	3	3	2	1	2	3	4	5	6
G	0	0	1	2	3	4	3	2	1	2	3	4	5
G	0	0	0	1	2	3	4	3	2	●	2	3	4

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

Identify Overlaps: Dynamic Programming

Say there are d strings of length n, total length $N = dn$, and a is total number of pairs with an overlap

# overlaps to try:	$O(d^2)$
Size of each DP matrix:	$O(n^2)$
Overall:	$O(d^2n^2)$, or $O(N^2)$

Contrast $O(N^2)$ with suffix tree: $O(N + a)$, but where a is worst-case $O(d^2)$

Real-world overlappers mix the two; index filters out vast majority of non-overlapping pairs, dynamic programming used for remaining pairs

There are other approaches too!

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

How do we store them?

2) "Assemble" the best possible layout

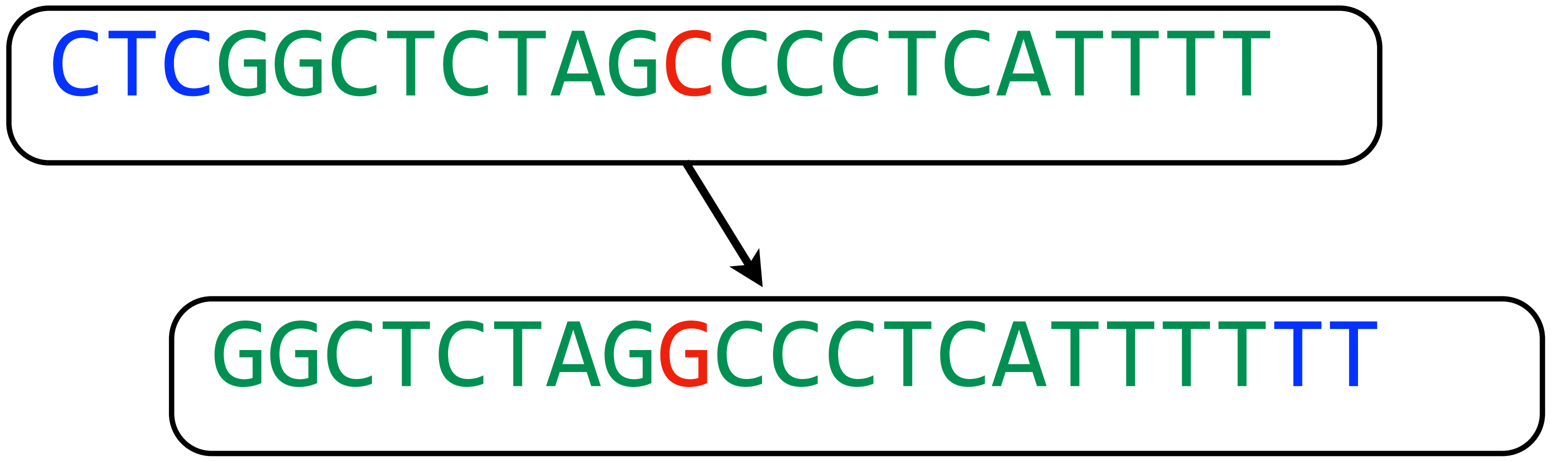
3) Reconstruct T based on consensus

Overlap graph

Each node is a string

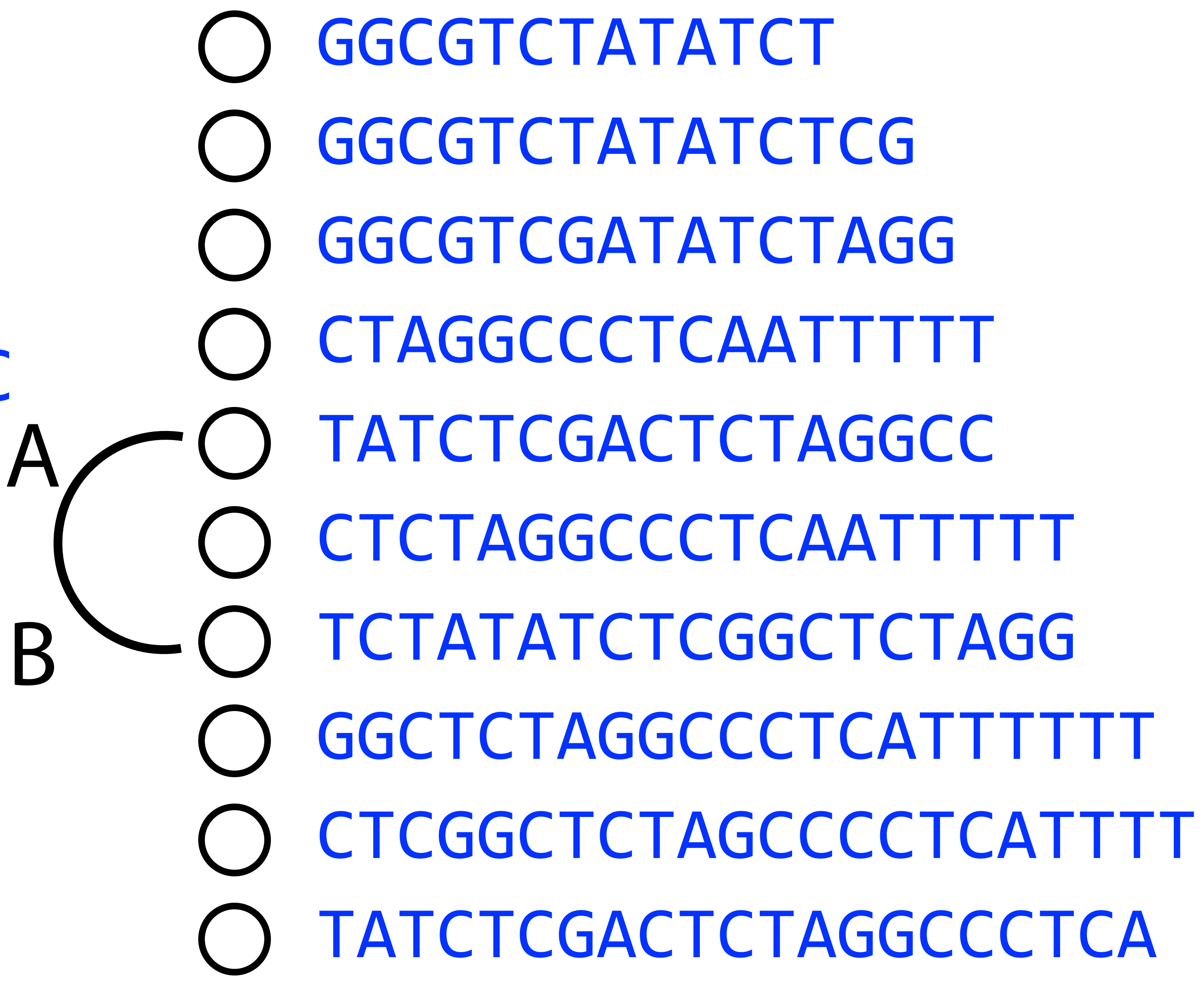
CTCGGCTCTAGCCCCTCATTTT

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



Overlap graph

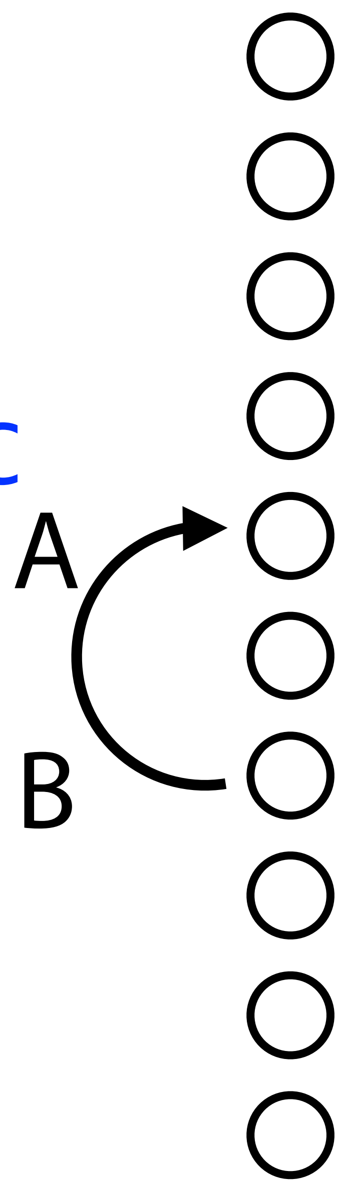
```
TCTATATCTCGGCTCTAGG
    ||||||| |||||||
    TATCTCGACTCTAGGCC
```



Which direction is this edge?

Overlap graph

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

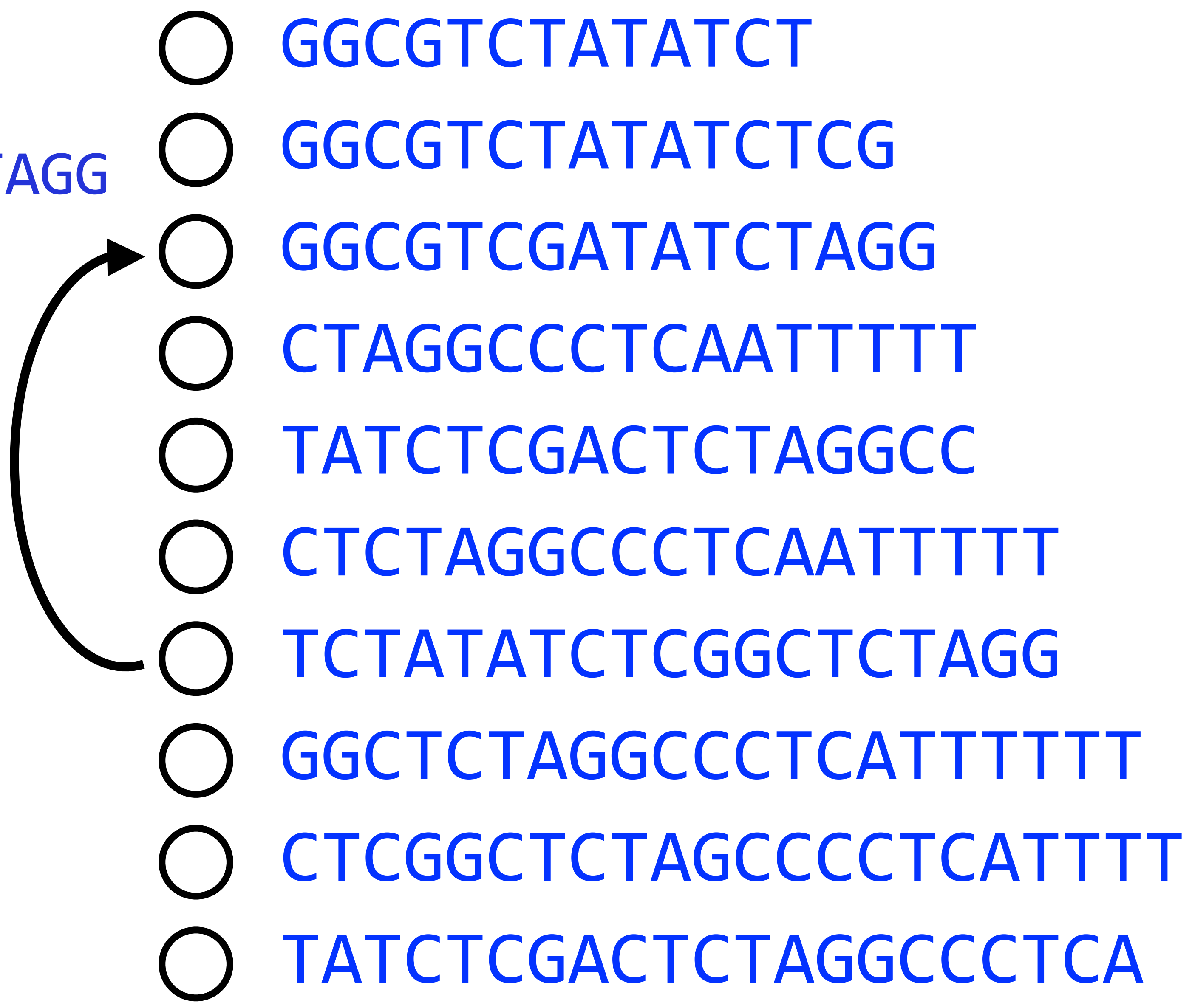


```
GGCGTCTATATCT
GGCGTCTATATCTCG
GGCGTCGATATCTAGG
CTAGGCCCTCAATTTTT
TATCTCGACTCTAGGCC
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
```

Which direction is this edge?

Overlap graph

```
TCTATATCTCGGCTCTAGG
                 ||
                 GGCGTCGATATCTAGG
```

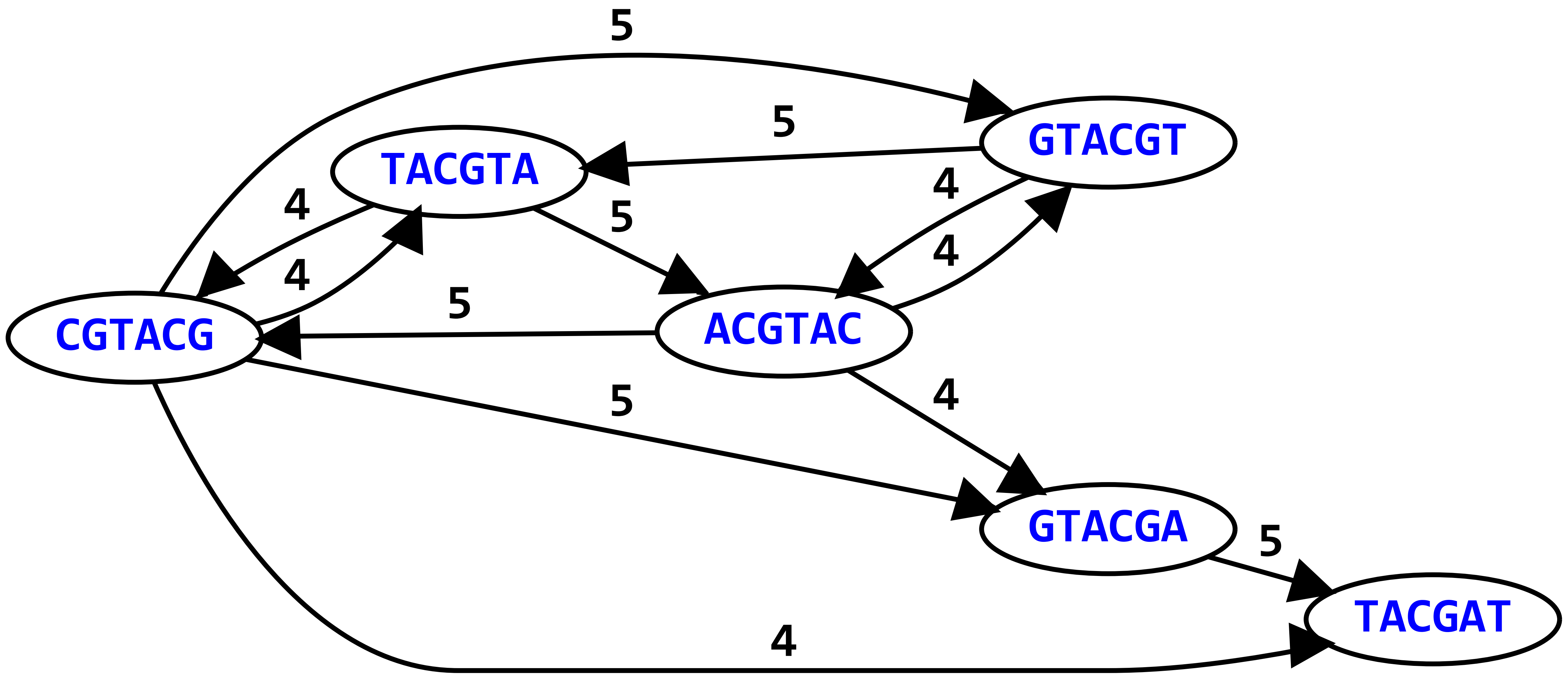


Not every overlap is ‘meaningful’

Overlap graph

Nodes: all 6-mers from GTACGTACGAT

Edges: overlaps of length *l* ≥4



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 ✓

Build an overlap graph

2) "Assemble" the best possible layout

3) Reconstruct T based on consensus

Assemble best possible layout

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

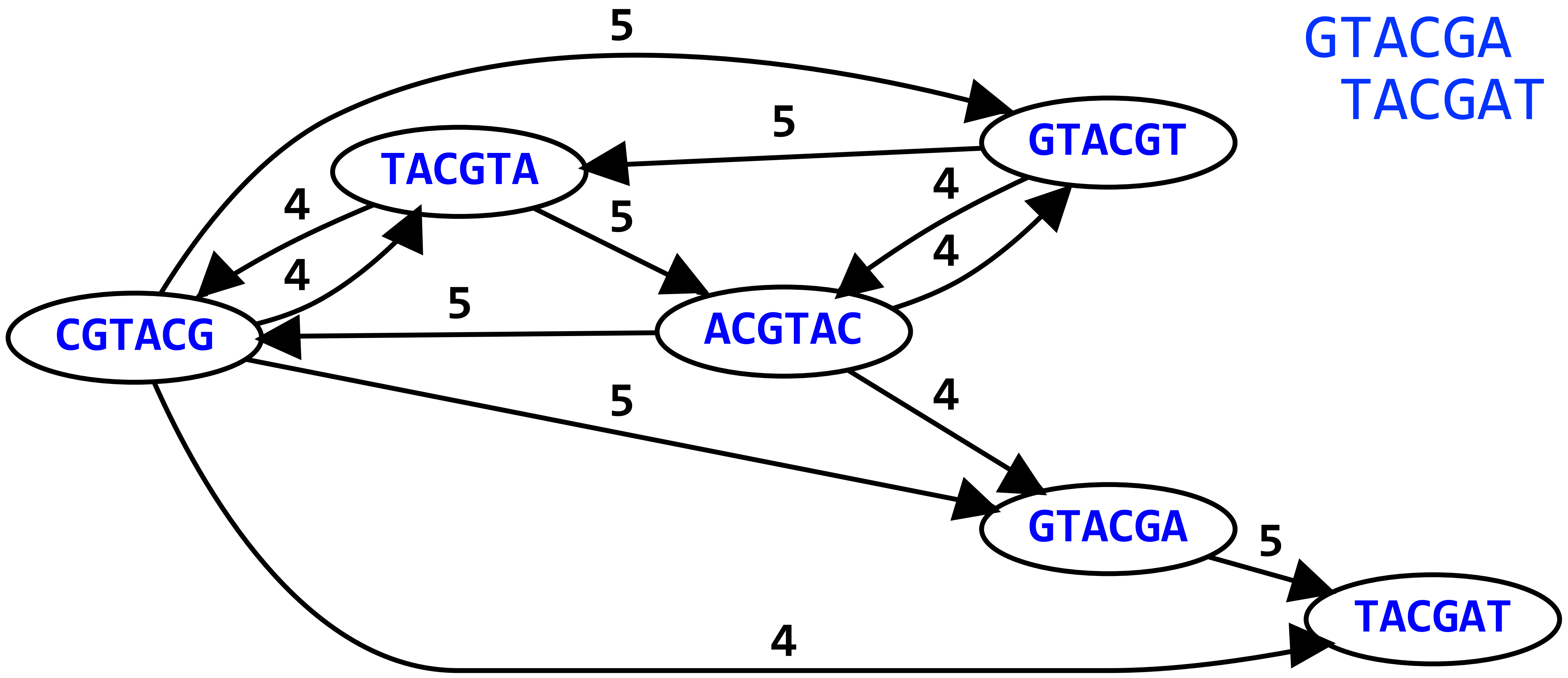
```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
```

Assemble best possible layout

Nodes: all 6-mers from GTACGTACGAT

Edges: overlaps of length $l \geq 4$

```
GTACGT
 TACGTA
  ACGTAC
   CGTACG
    GTACGA
     TACGAT
```

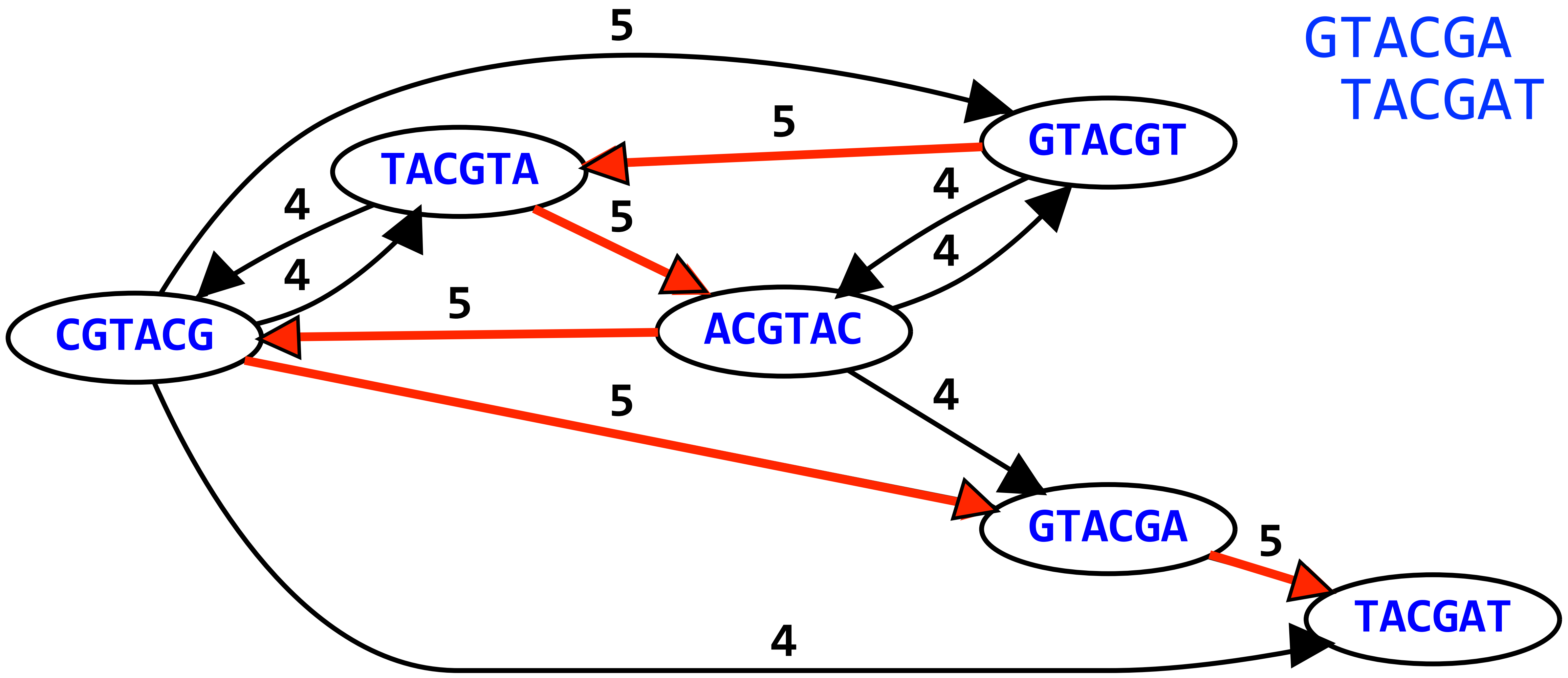


Assemble best possible layout

Nodes: all 6-mers from GTACGTACGAT

Edges: overlaps of length $l \geq 4$

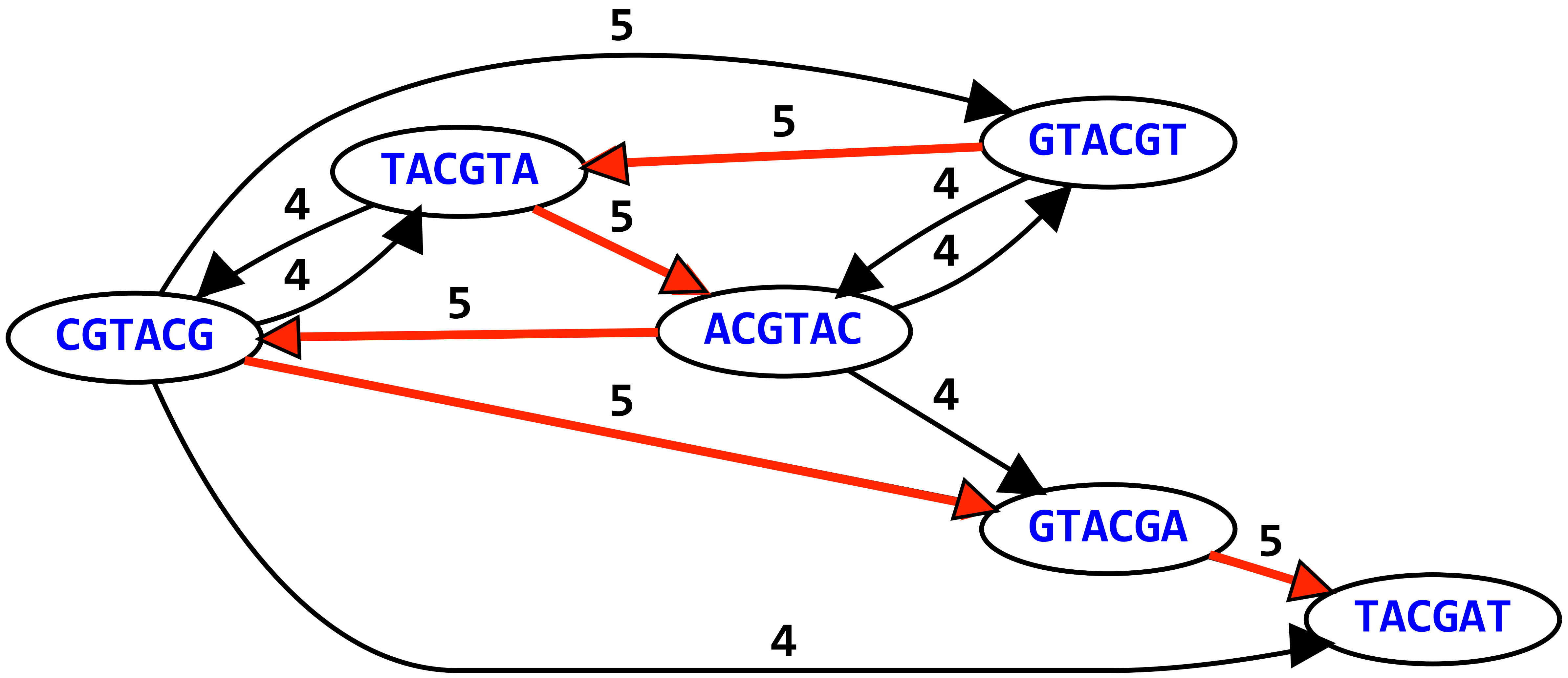
```
GTACGT
 TACGTA
  ACGTAC
   CGTACG
    GTACGA
     TACGAT
```



Our layout is a path through our graph that touches all nodes

Assemble best possible layout

Given overlap graph, how can we find the "best" path through this graph?



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

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

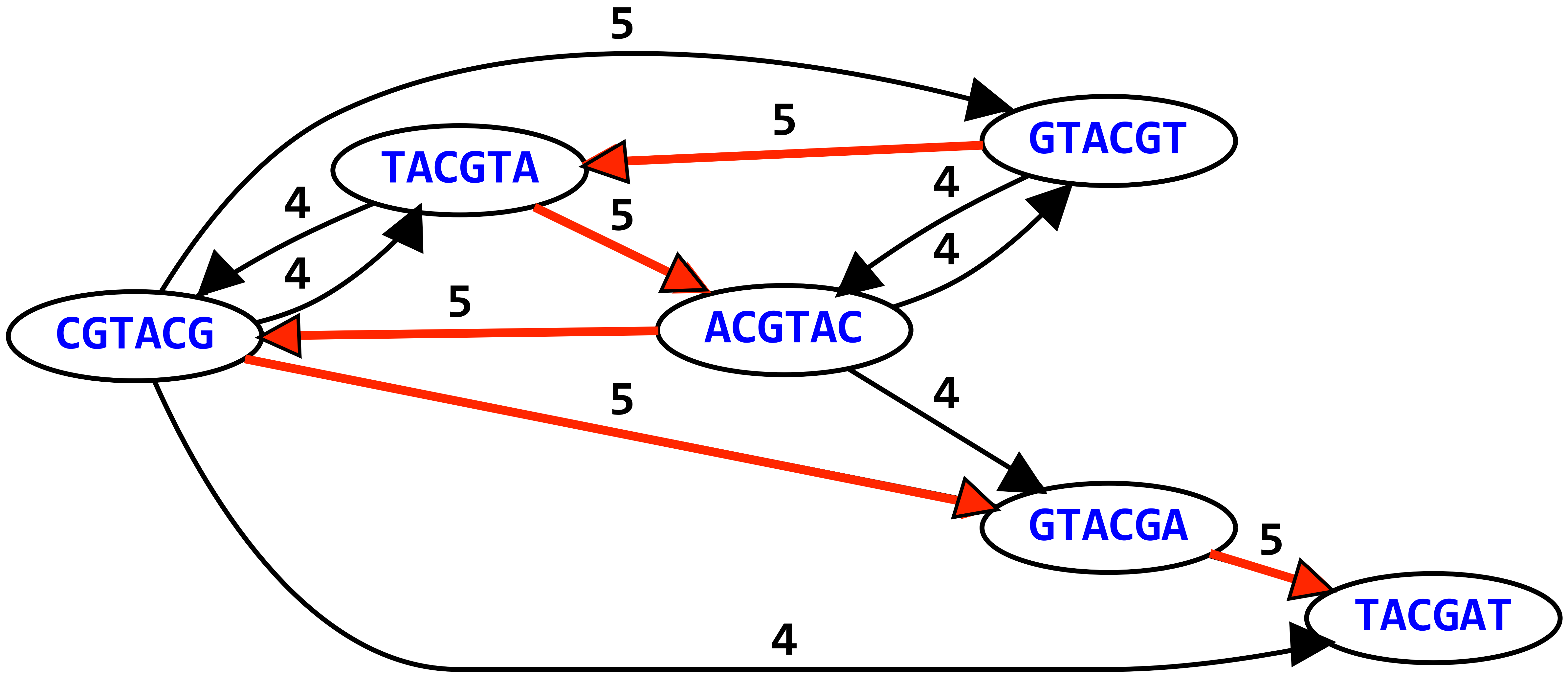
$\underbrace{\hspace{10em}}_{24}$

SCS(S): AAABBBABAA

$\underbrace{\hspace{5em}}_{10}$

Original example courtesy of Ben Langmead

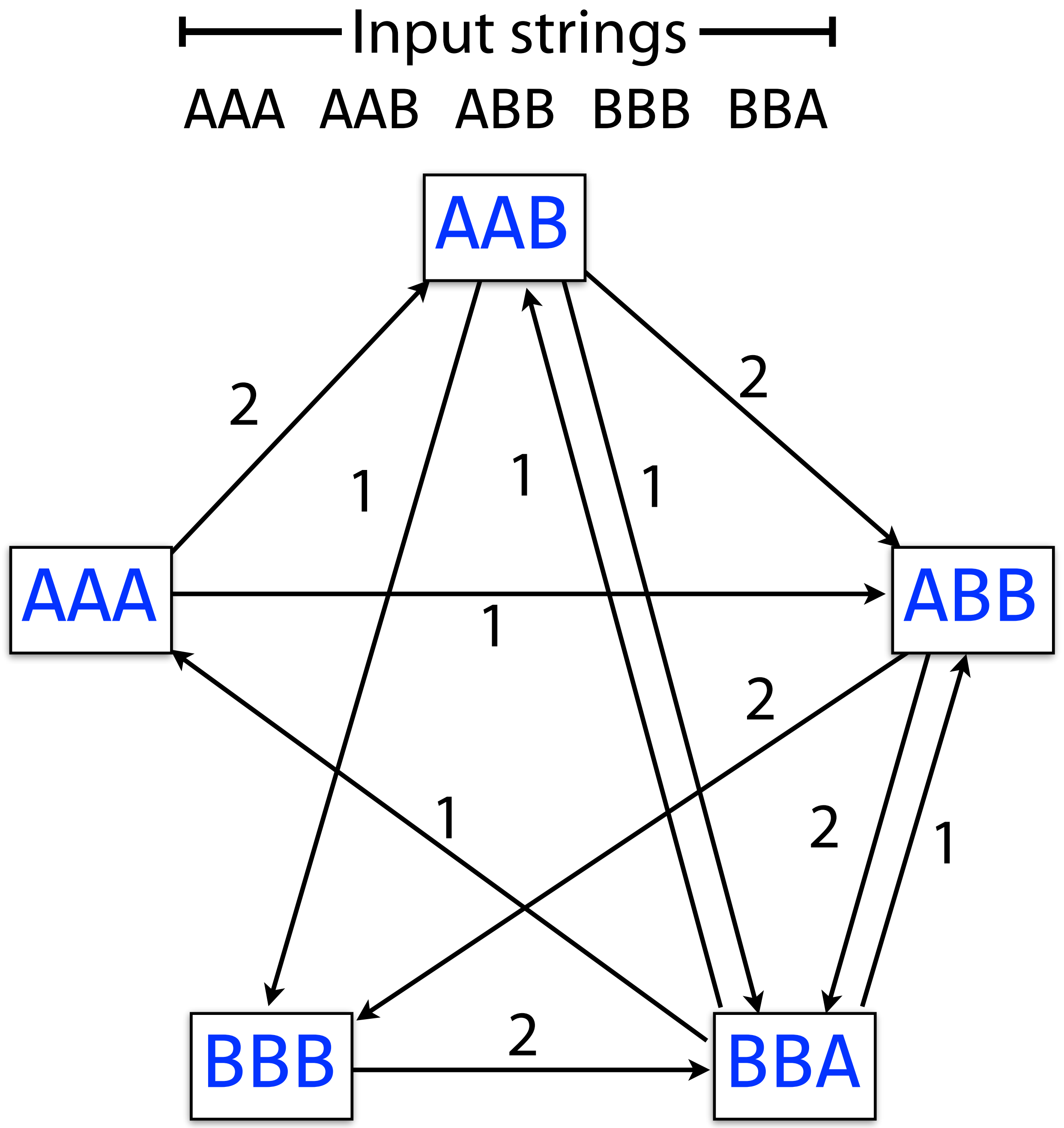
Shortest Common Superstring



```
>>> scs(['GTACGT', 'TACGTA', 'ACGTAC',
         'CGTACG', 'GTACGA', 'TACGAT'])
'GTACGTACGAT'
```

Shortest Common Superstring

How can we solve SCS using graphs?



Original example courtesy of Ben Langmead

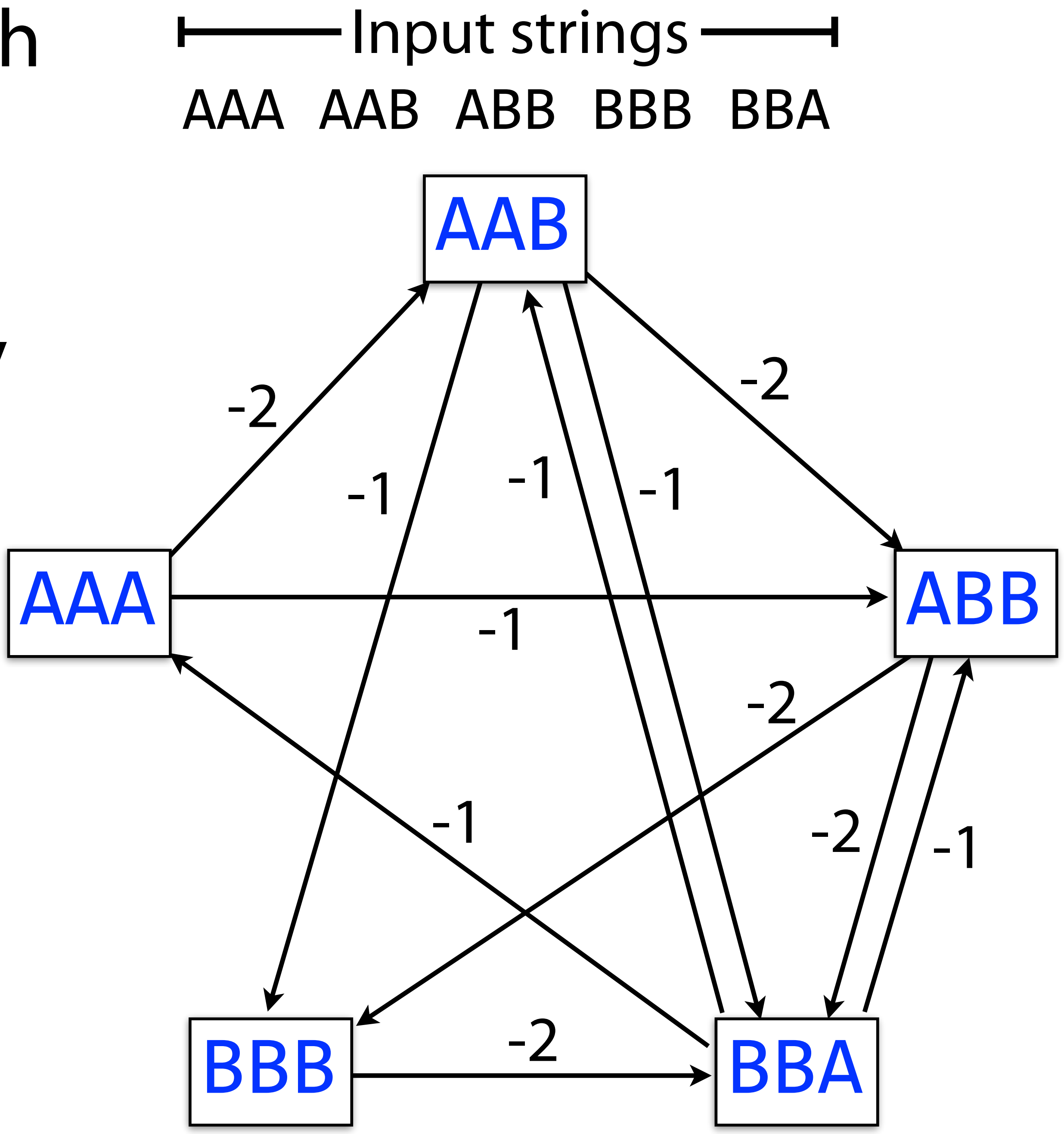
Shortest Common Superstring

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: $\underbrace{\text{AAA AAB}}$ 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

$$\underbrace{\text{AAA AAB}}_{\text{AAAB}}$$

Take into account overlap whenever possible

Shortest Common Superstring: Exhaustive

Pick order for strings in *S and* construct superstring

order 1: AAA $\underbrace{\text{AAB}\quad\text{ABA}}$ 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 $\underline{\text{ABB}\quad\text{BAA}}$ BAB BBA BBB

AAABABBAA

Shortest Common Superstring: Exhaustive

Pick order for strings in *S* *and* construct superstring

order 1: AAA AAB ABA ABB $\underbrace{\text{BAA BAB}}$ 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 $\longleftarrow$ 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 $\longleftarrow$ superstring 1

order 2: AAA AAB ABA BAB ABB BBB BAA BBA

AAABABBBAABBA $\longleftarrow$ superstring 2

Try all possible orderings and pick shortest superstring

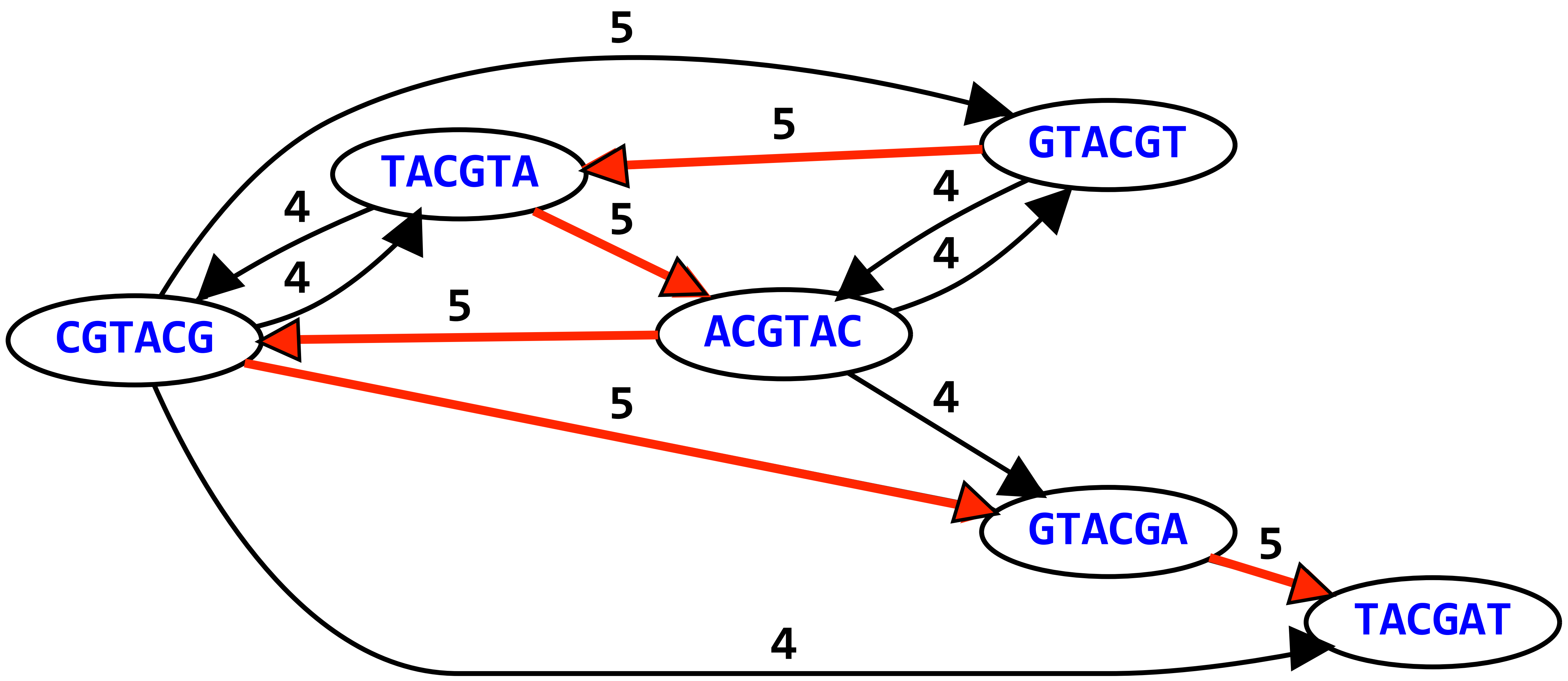
If *S* contains *n* strings, how many orderings are are possible?

n ! (*n* factorial) orderings possible

Assemble best possible layout

We want the "best" path through our graph:

SCS is not viable (NP-Hard)



Maybe we don't need the optimal path…

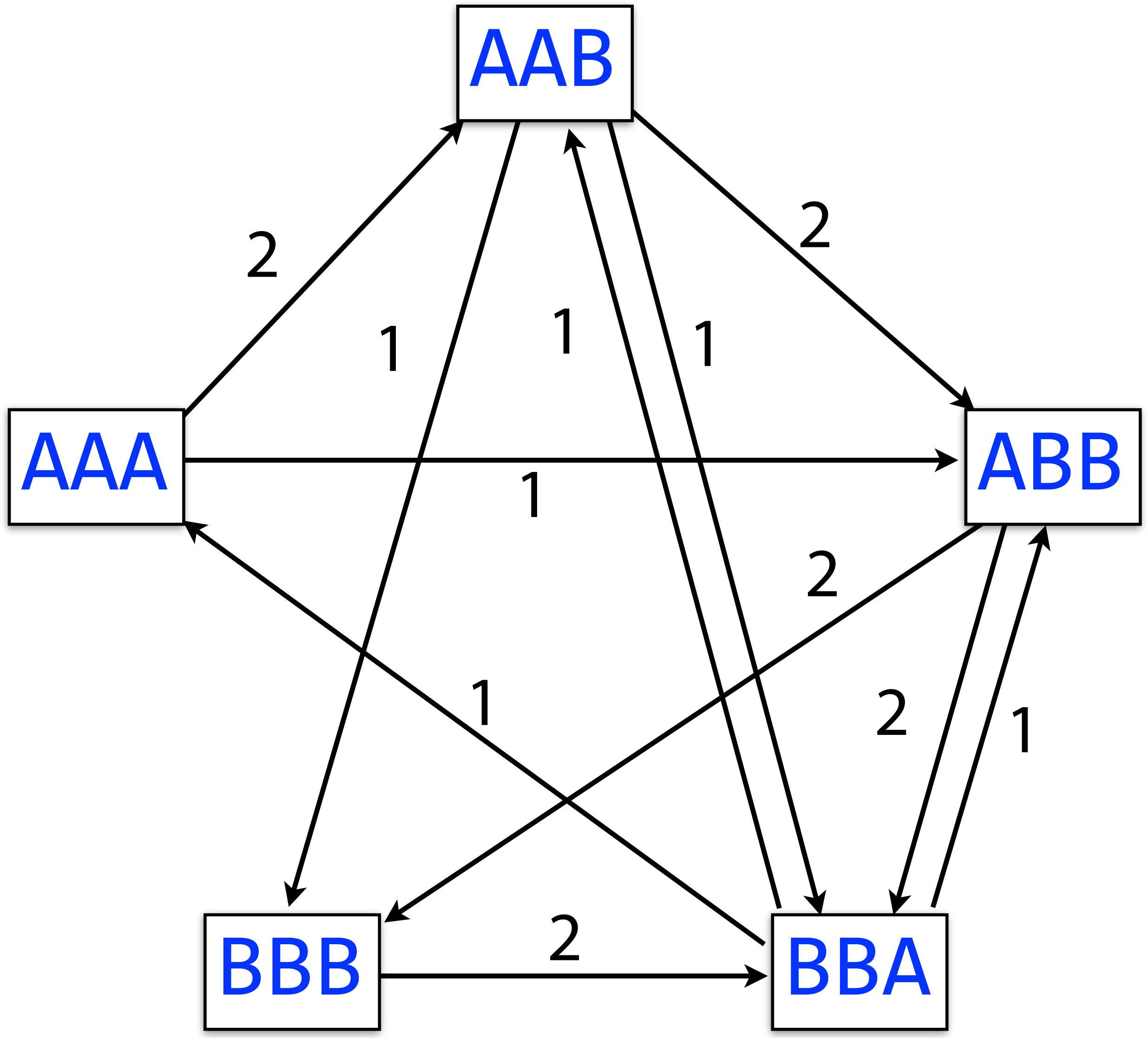
Shortest Common Superstring: Greedy

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



Original example courtesy of Ben Langmead

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.

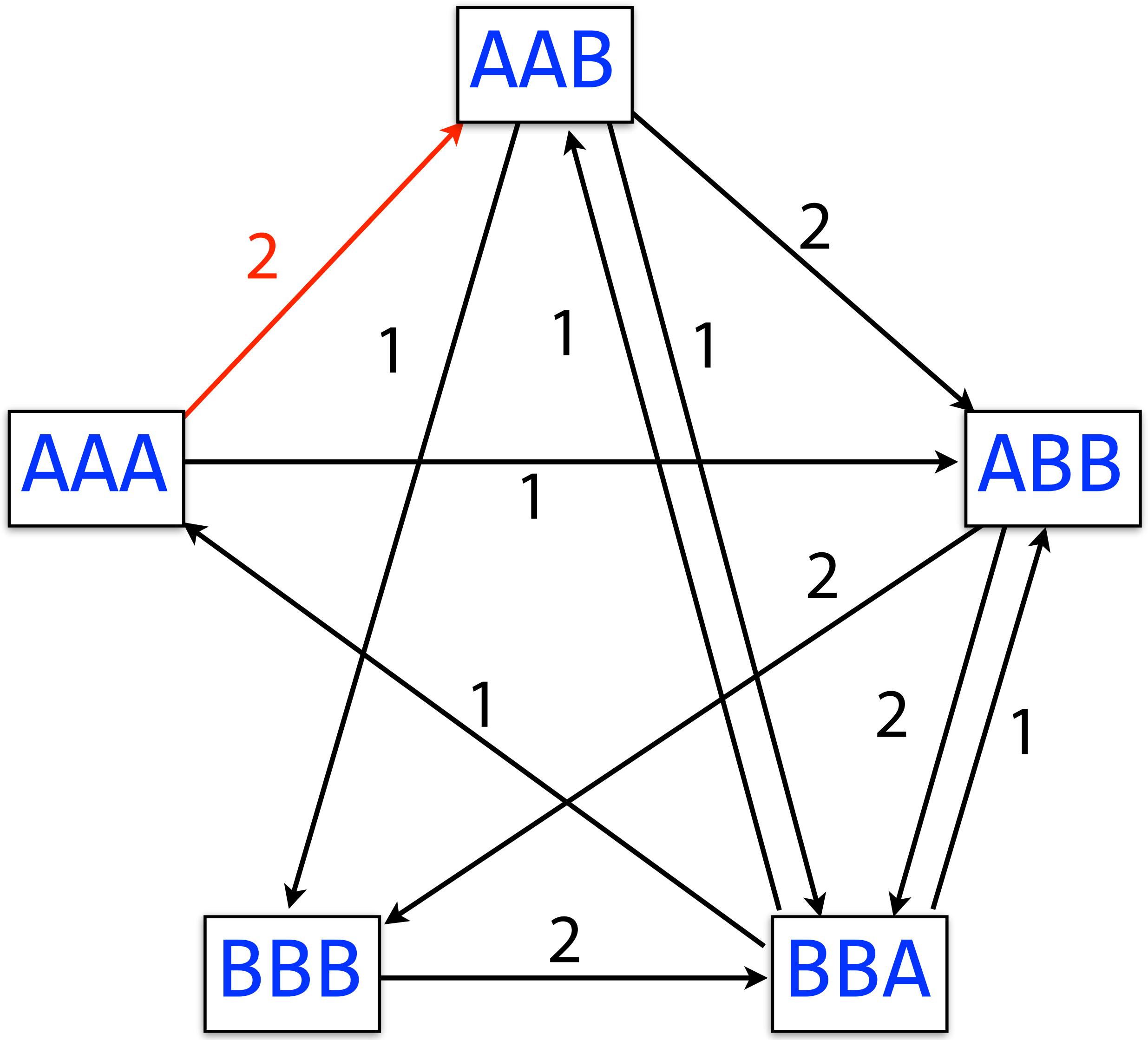
Algorithm in action ($l = 1$):

|——— Input strings ———|

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

Pick the highest weight overlap



Original example courtesy of Ben Langmead

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 ($l = 1$):

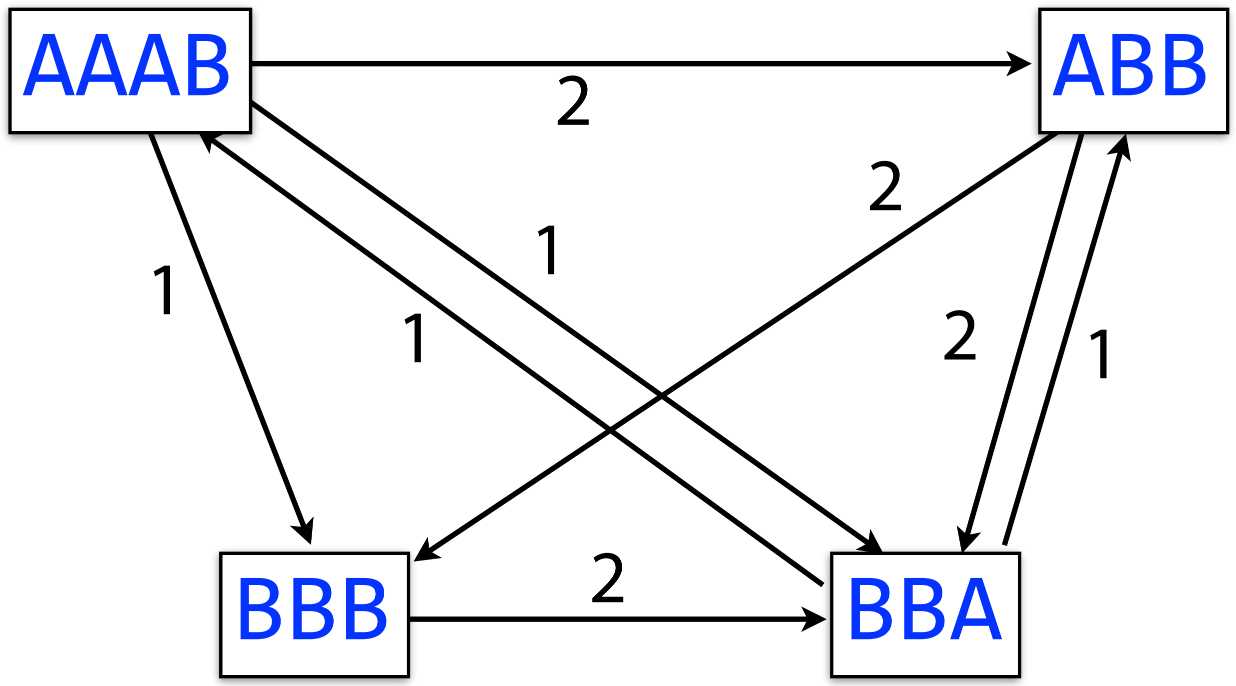
├── Input strings ──┤

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

Merge to create a new node



Original example courtesy of Ben Langmead

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 ($l = 1$):

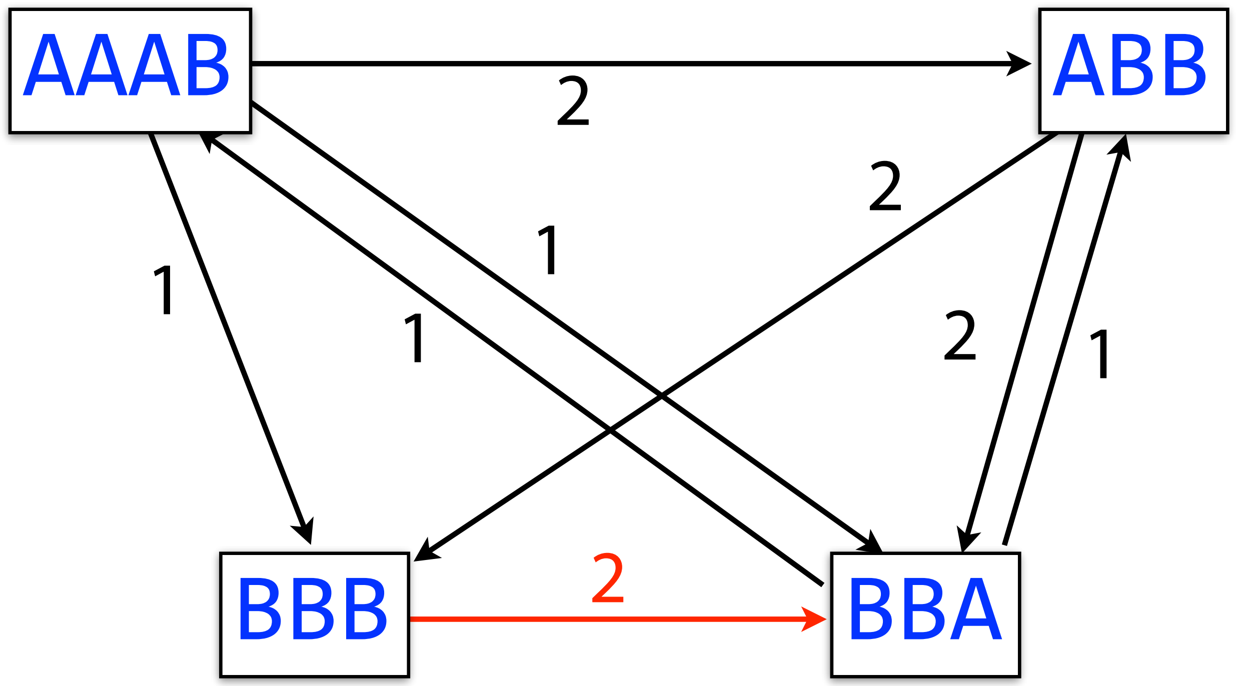
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

Pick the highest weight overlap


$$\begin{tikzcd}
AAAB \arrow[rr, "2"] \arrow[d, "1"'] \arrow[drr, "1"] & & ABB \arrow[dll, "2"'] \arrow[d, shift right, "2"'] \\
BBB \arrow[rr, "2"] & & BBA \arrow[ull, "1"] \arrow[u, shift right, "1"']
\end{tikzcd}$$

Original example courtesy of Ben Langmead

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 ──┤

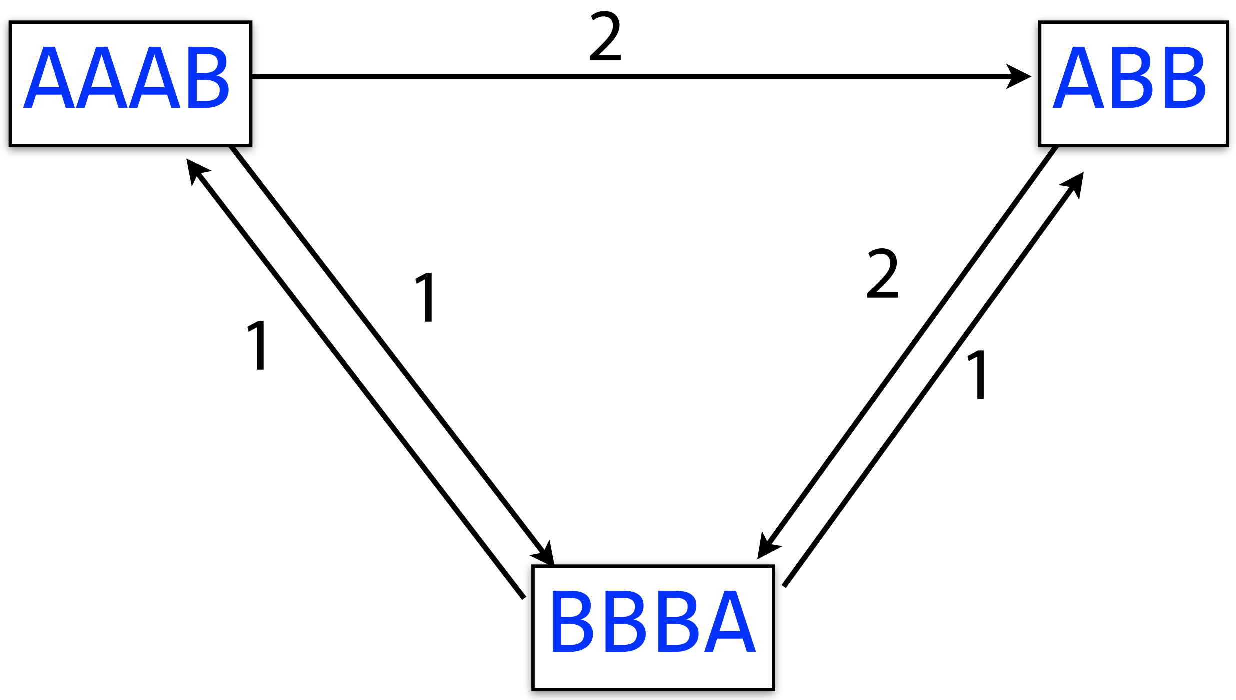
AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

Merge to create a new node



Original example courtesy of Ben Langmead

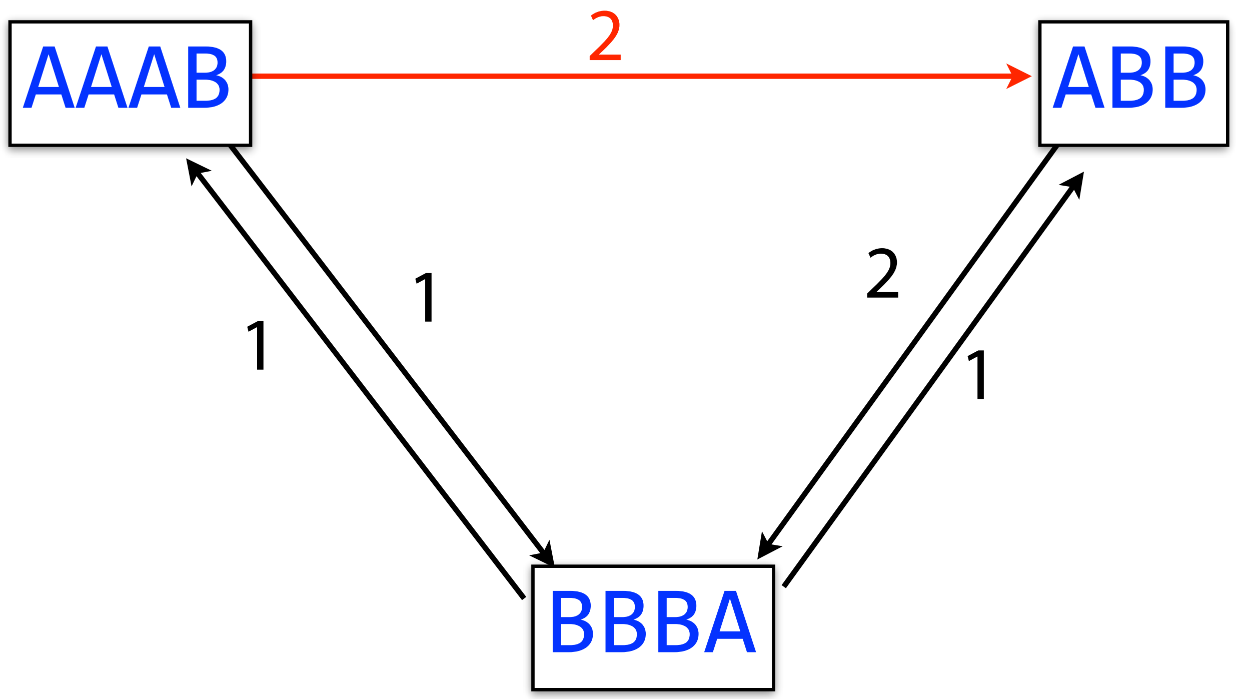
Shortest Common Superstring: Greedy

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



Original example courtesy of Ben Langmead

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

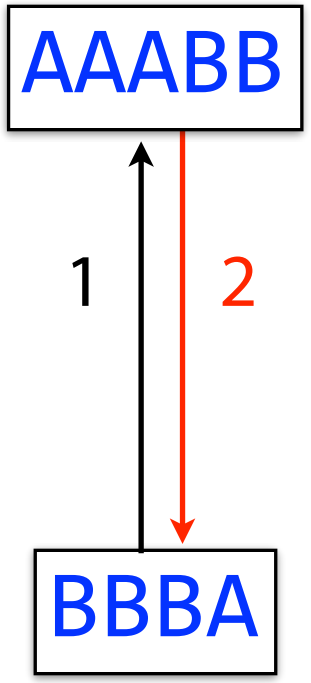
AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

AAABB BBBA



Original example courtesy of Ben Langmead

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):

$\vdash$—— Input strings ——$\dashv$

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

$\boxed{\text{AAABBBA}}$

That's the SCS

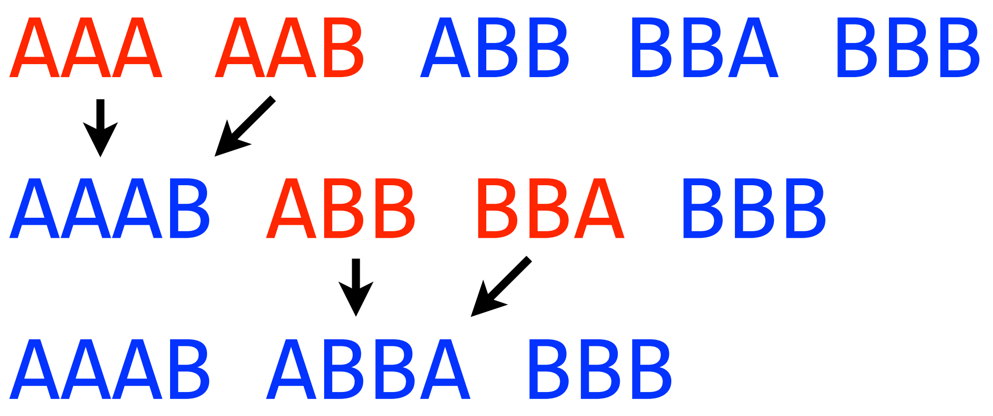
Is Greedy-SCS optimal?

Original example courtesy of Ben Langmead

Shortest Common Superstring: Greedy

$$\begin{tikzcd}
\text{AAA} \arrow[d] & \text{AAB} \arrow[dl] & \text{ABB} & \text{BBA} & \text{BBB} \\
\text{AAAB} & \text{ABB} & \text{BBA} & \text{BBB} &
\end{tikzcd}$$

Shortest Common Superstring: Greedy



Shortest Common Superstring: Greedy

$$\begin{tikzcd}
AAA \arrow[d] & AAB \arrow[dl] & ABB & BBA & BBB \\
AAAB & ABB \arrow[d] & BBA \arrow[dl] & BBB & \\
AAAB \arrow[d] & ABBA \arrow[dl] & BBB & & \\
AAABBA & BBB & & &
\end{tikzcd}$$

Shortest Common Superstring: Greedy

$$\begin{tikzcd}
\text{AAA} \arrow[d] & \text{AAB} \arrow[dl] & \text{ABB} & \text{BBA} & \text{BBB} \\
\text{AAAB} & \text{ABB} \arrow[d] & \text{BBA} \arrow[dl] & \text{BBB} & \\
\text{AAAB} \arrow[d] & \text{ABBA} \arrow[dl] & \text{BBB} & & \\
\text{AAABBA} \arrow[d] & \text{BBB} \arrow[dl] & & & \\
\text{AAABBABBB} & \text{superstring, length=9} \arrow[l] & & &
\end{tikzcd}$$

Shortest Common Superstring: Greedy

$$\begin{tikzcd}
\text{AAA} \arrow[d] & \text{AAB} \arrow[dl] & \text{ABB} & \text{BBA} & \text{BBB} \\
\text{AAAB} & \text{ABB} \arrow[d] & \text{BBA} \arrow[dl] & \text{BBB} & \\
\text{AAAB} \arrow[d] & \text{ABBA} \arrow[dl] & \text{BBB} & & \\
\text{AAABBA} \arrow[d] & \text{BBB} \arrow[dl] & & & \\
\text{AAABBABBB} & \text{superstring, length=9} \arrow[l] & & &
\end{tikzcd}$$

AAABBBA $\longleftarrow$ superstring, length=7

Greedy answer *isn't necessarily optimal*

Shortest Common Superstring: 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?

Original example courtesy of Ben Langmead

Shortest Common Superstring: 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
```

$\uparrow$

Foiled by repeat!

Original example courtesy of Ben Langmead

Shortest Common Superstring: 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`

Original example courtesy of Ben Langmead

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

String Repeats

Basic principle: *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

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

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

String Repeats

Basic principle: *repeats foil assembly*

Another example using Greedy-SCS:

Input: `swinging_and_the_ringing_of_the_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_b`
3, 25	`swinging_and_the_ringing_of_the_bells_bells_bells_bells`

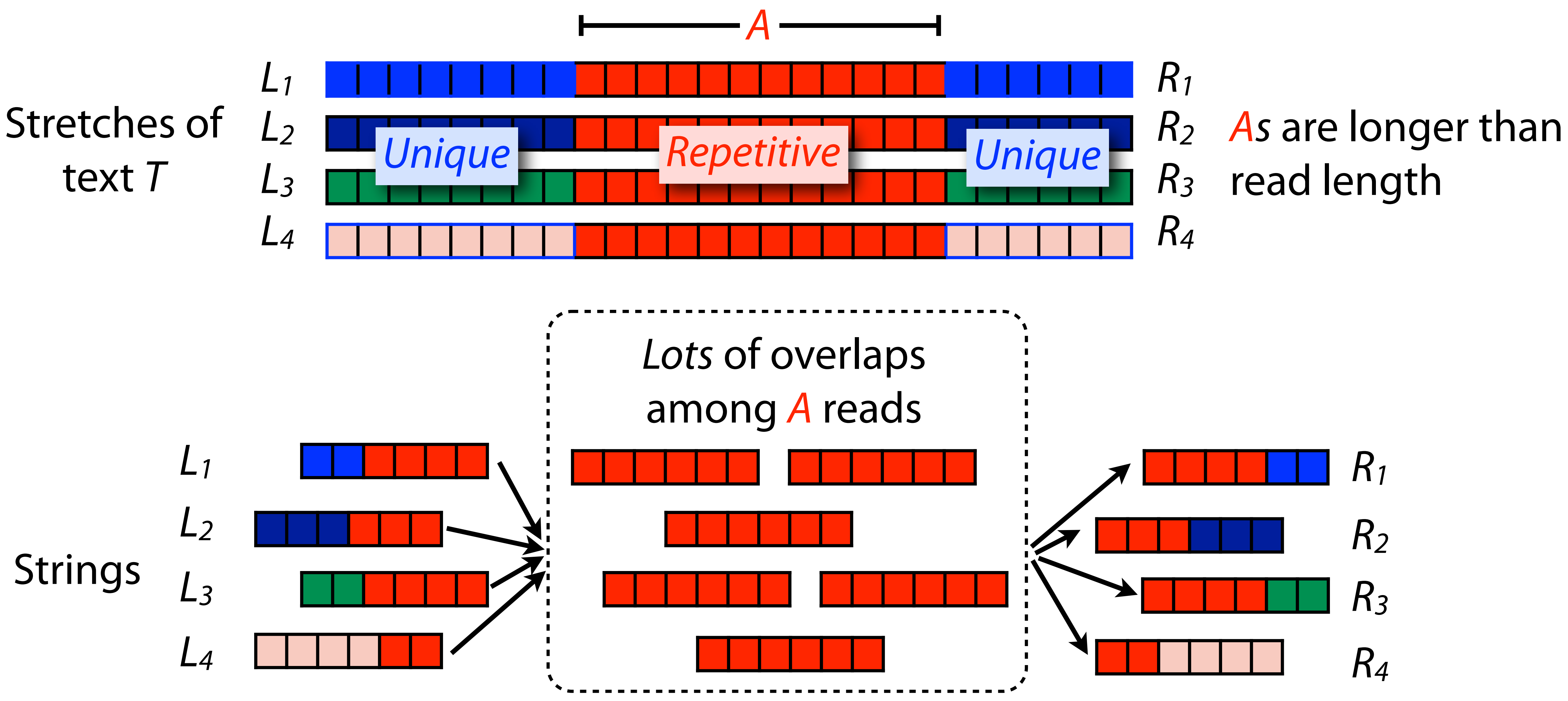
$\longrightarrow$

longer and longer substrings 'reach' further into repeat

Original example courtesy of Ben Langmead

String Repeats

Portion of overlap graph involving repeat family *A*

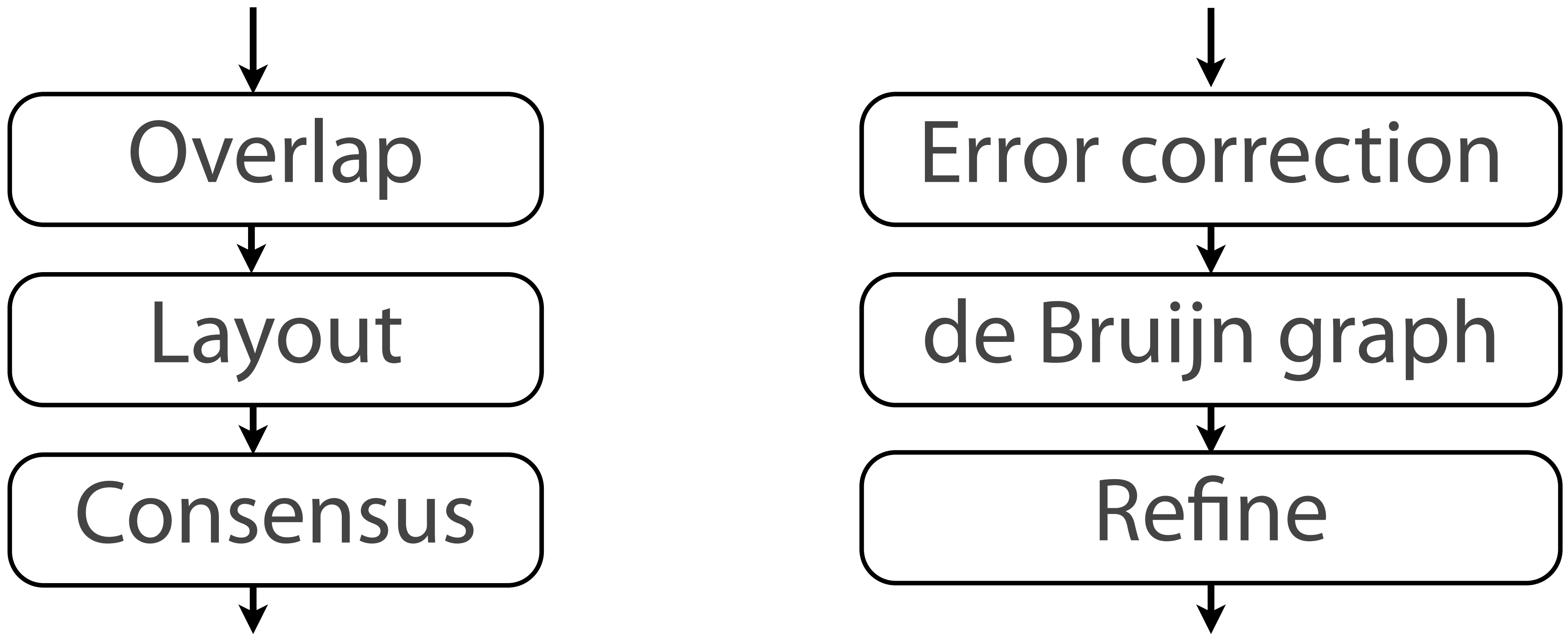


Even if we avoid collapsing copies of *A*, we can't know which paths *in* correspond to which paths *out*

Real-world Assembly

Alternative 1: Overlap-Layout-Consensus (OLC) assembly

Alternative 2: De Bruijn graph (DBG) assembly


Overlap
Layout
Consensus
Error correction
de Bruijn graph
Refine