String Algorithms and Data Structures
Overlap Graphs

CS 199-225
Brad Solomon

December 9, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Please fill out ICES Evaluations

Feedback is important for the development of the class

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

Learning Objectives

Introduce State Diagrams and Markov Chains

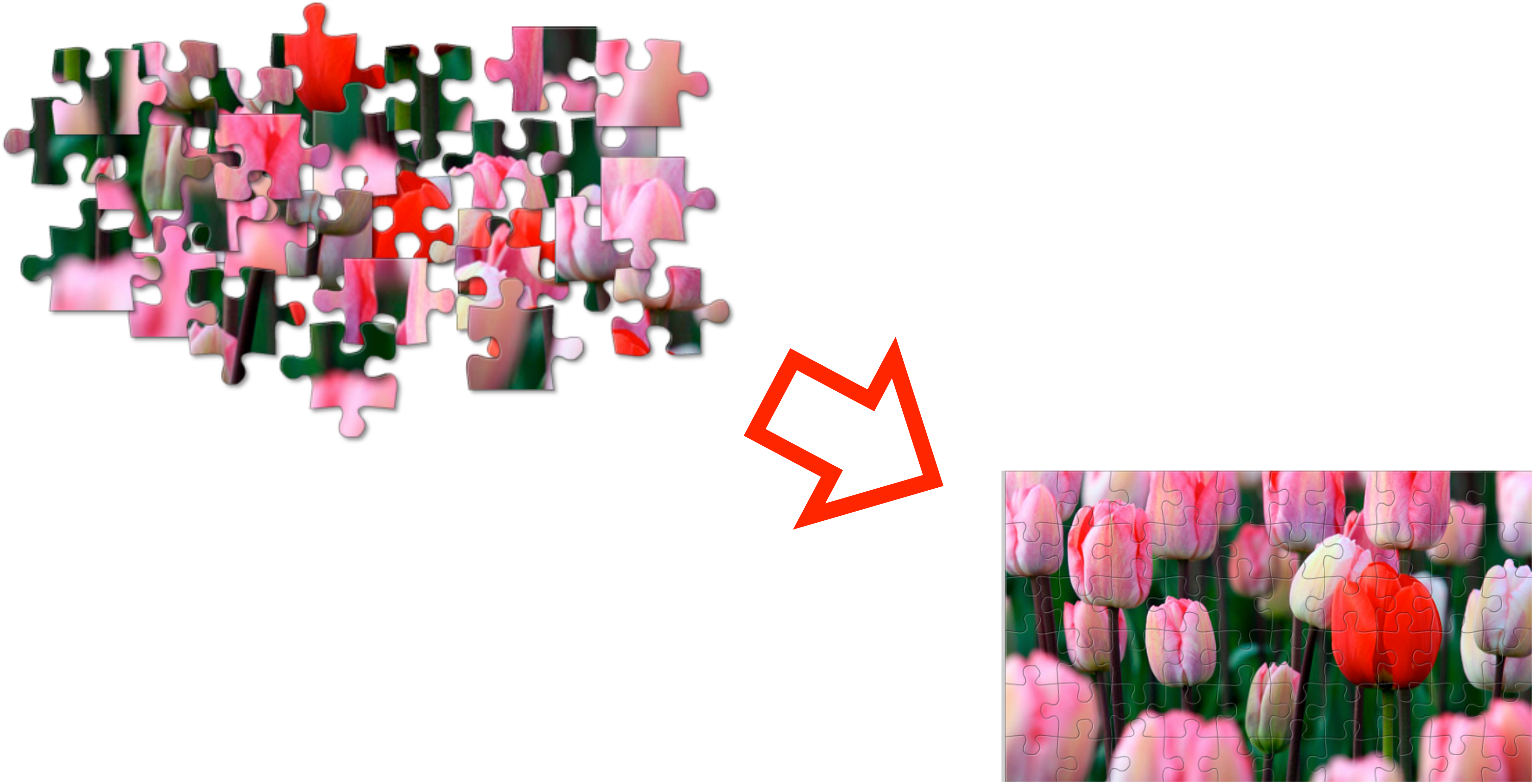
Identify how Markov chains can be used to:

- Estimate probabilities of sequences
- Identify more probable labels
- Predict future states

Define and determine stationary states

Introduce Hidden Markov Models

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

CTAGGCCCTCAATTTTT
CTCTAGGCCCTCAATTTTT
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT

From these

Reconstruct this

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

String Assembly

CTAGGCCCTCAATTTTT
CTCTAGGCCCTCAATTTTT
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT

From these

Reconstruct this

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

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 all possible overlaps

Given two strings, how can we find all overlaps?

X: `CTCTAGGCC`

Y: `TAGGCCCTC`

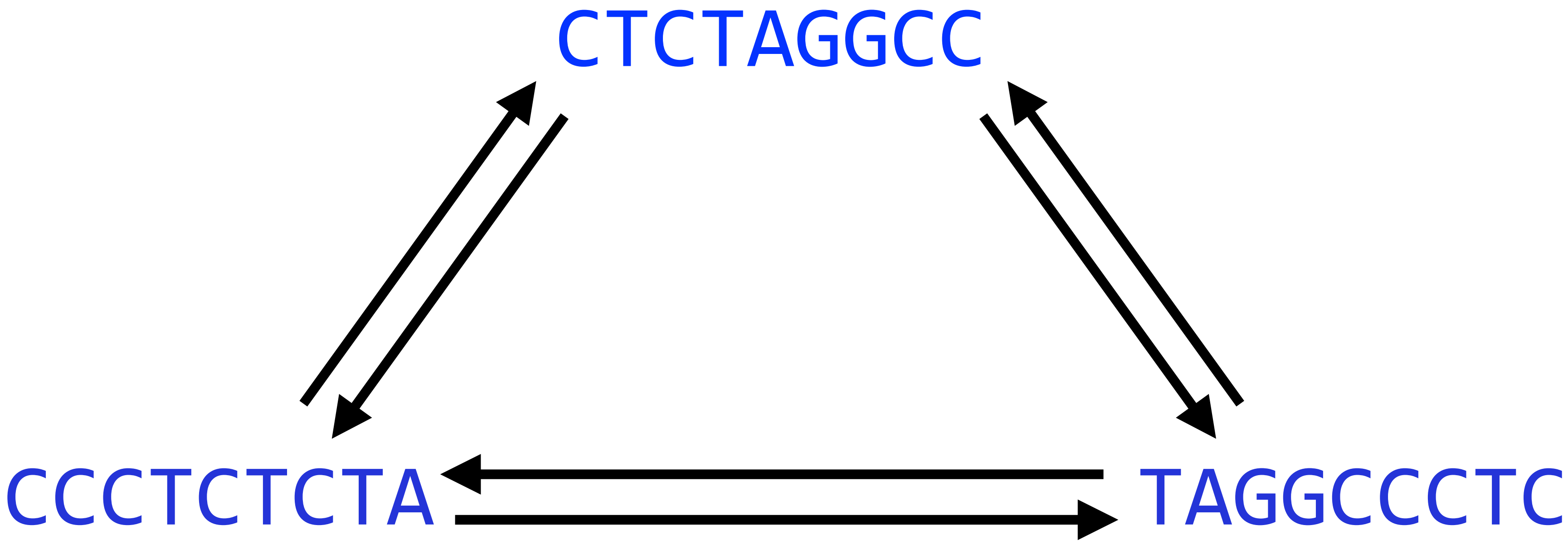
Identify all possible overlaps

How many unique strings do we need to search for?

X: CTCTAGGCC
T
TA
TAG
TAGG
TAGGC
TAGGCC
TAGGCCC
TAGGCCCT
TAGGCCCTC

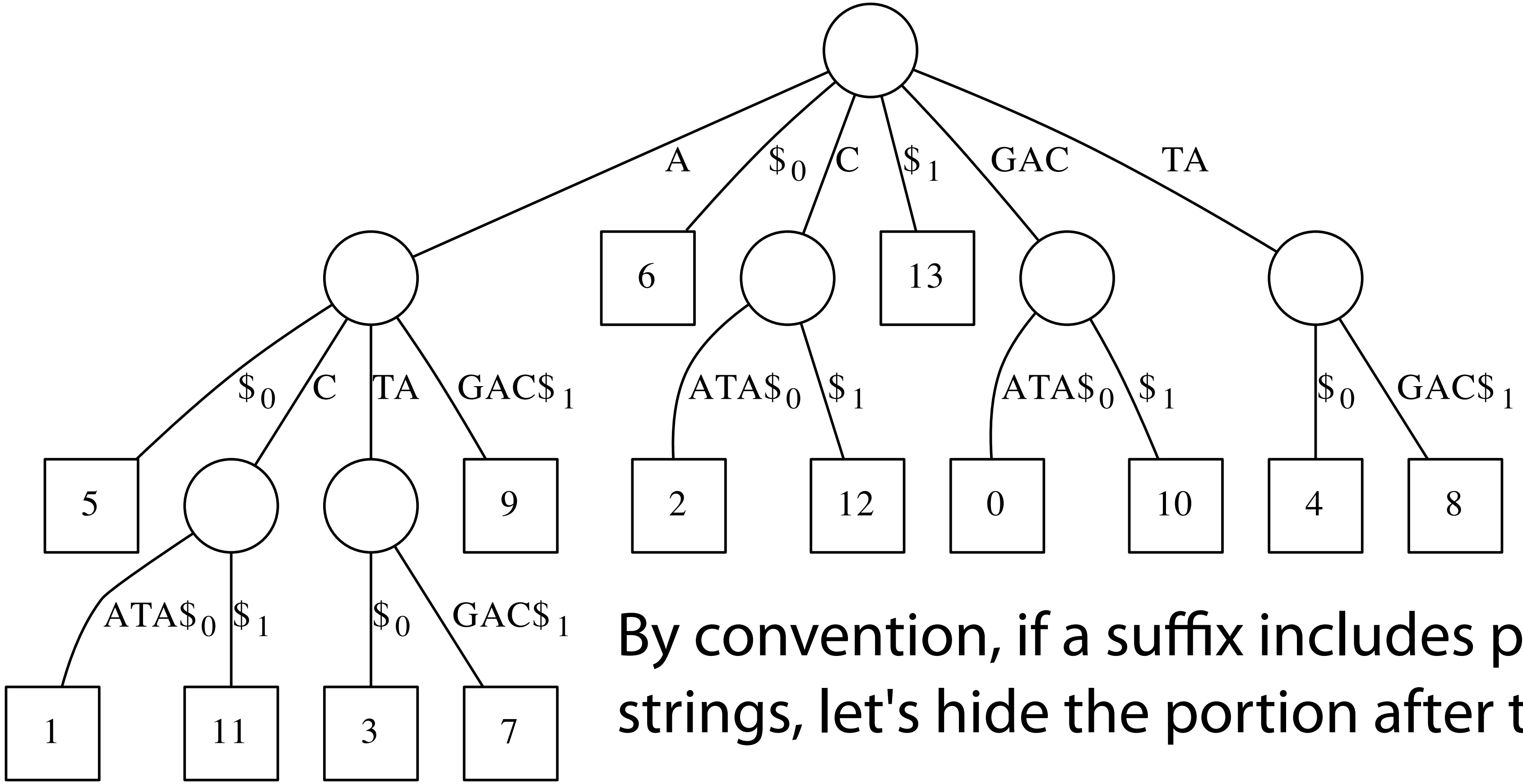
Identify all possible overlaps

If we are trying to find overlap between multiple strings?



Identify all possible overlaps

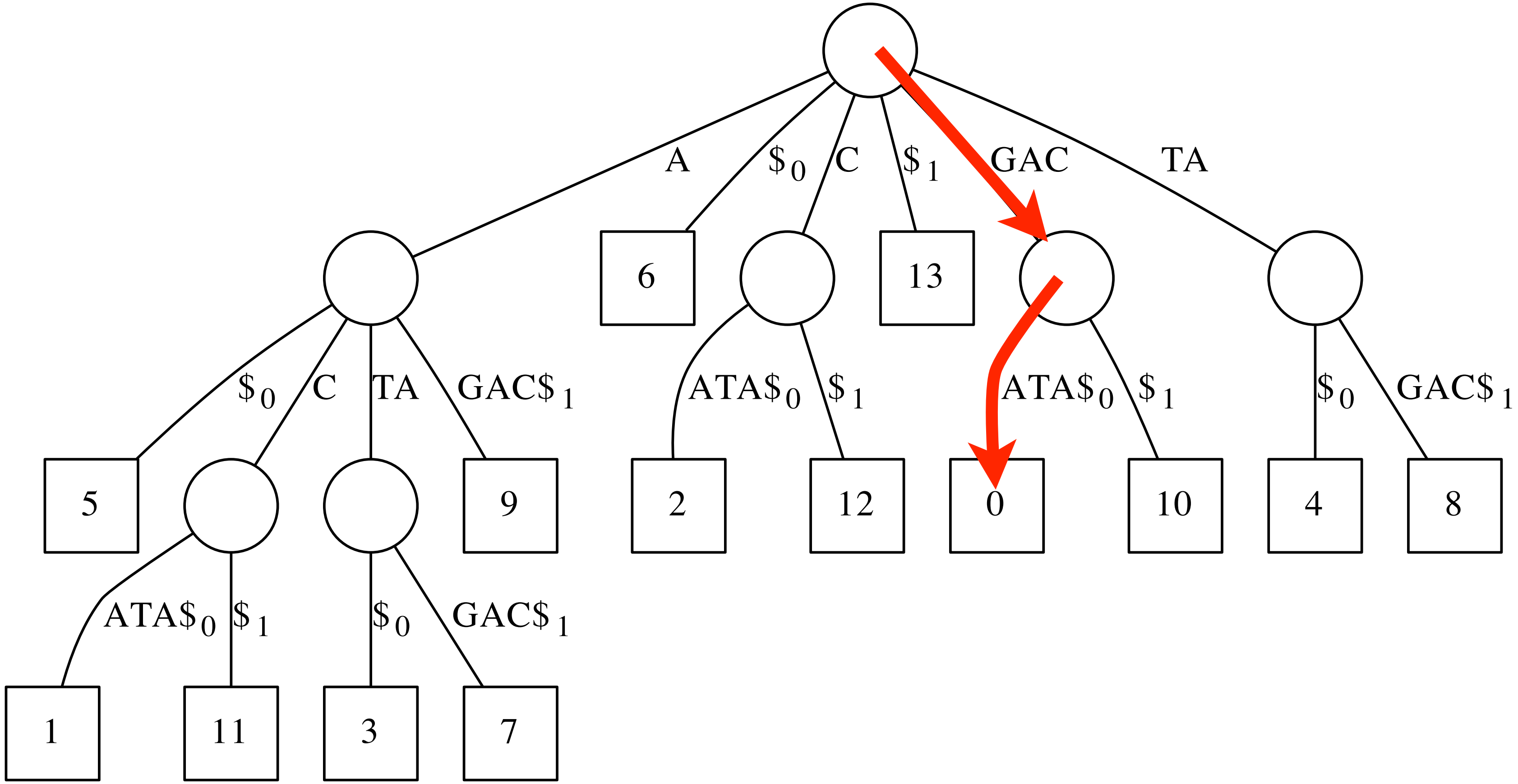
Generalized suffix tree for { "GACATA", "ATAGAC" } GACATA$\$_0$ATAGAC$\$_1$



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

Identify all possible overlaps

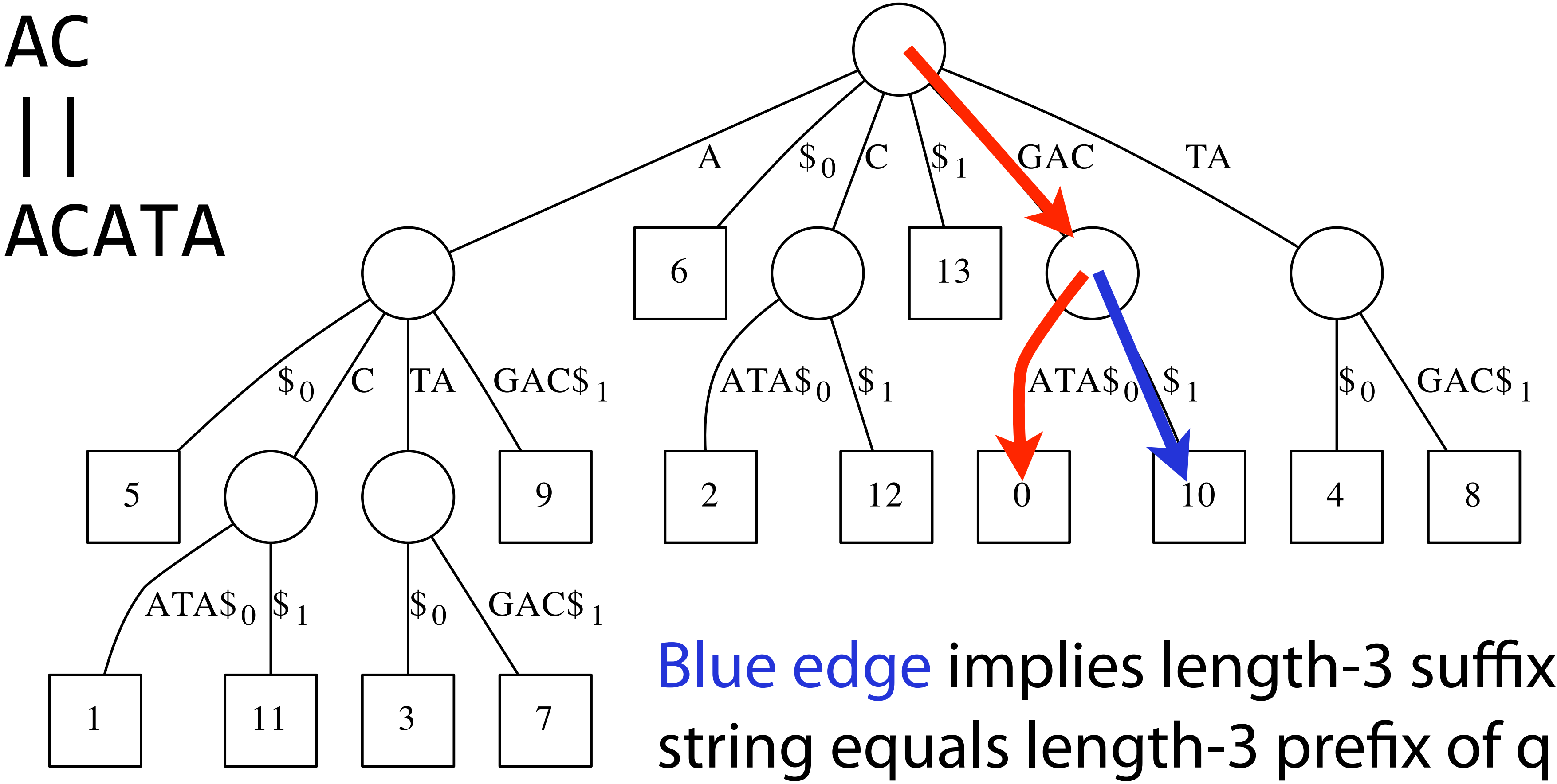
Let query = GACATA. From root, follow path labeled with query.



Identify all possible overlaps

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

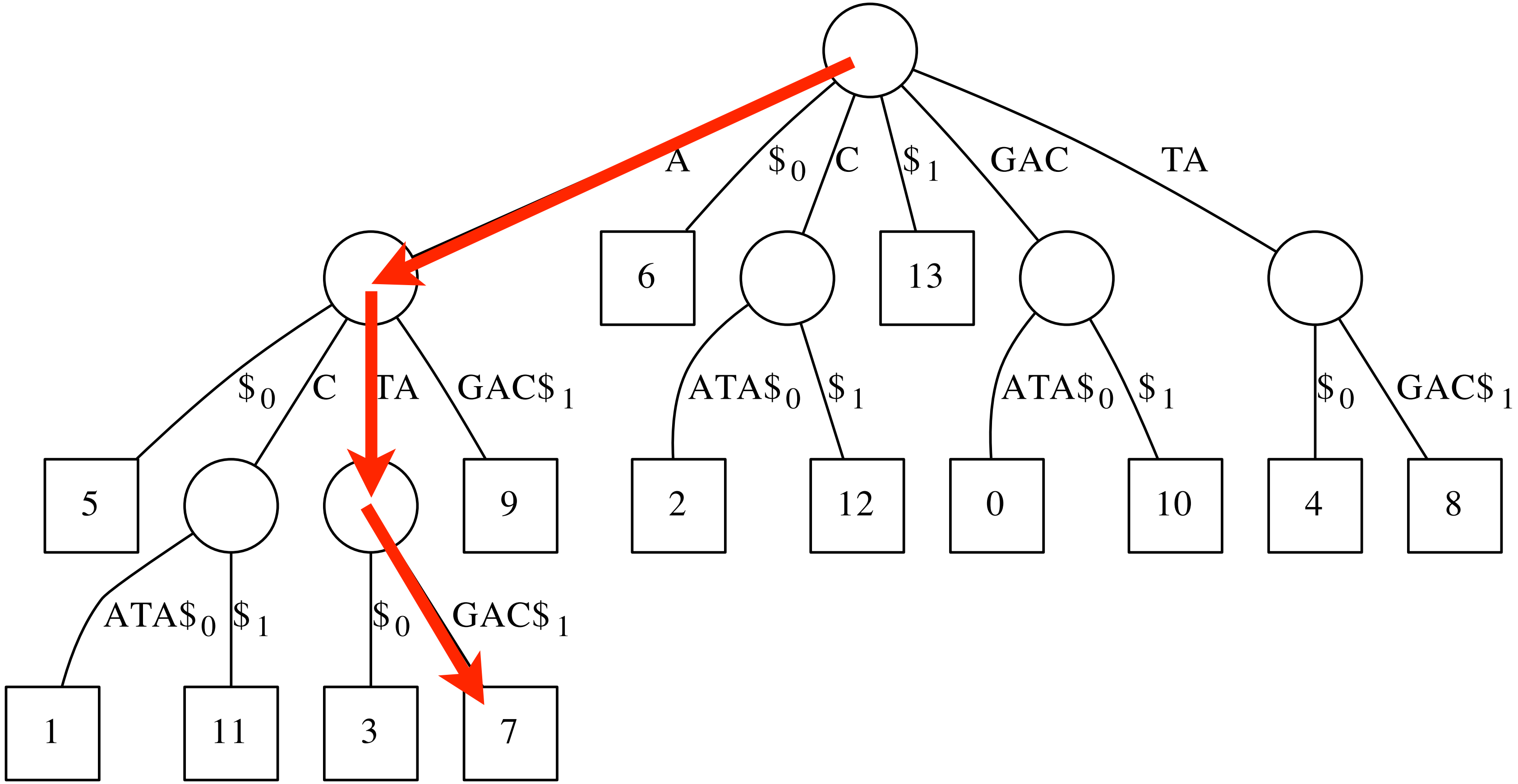
```
ATAGAC
   |||
   GACATA
```



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

Identify all possible overlaps

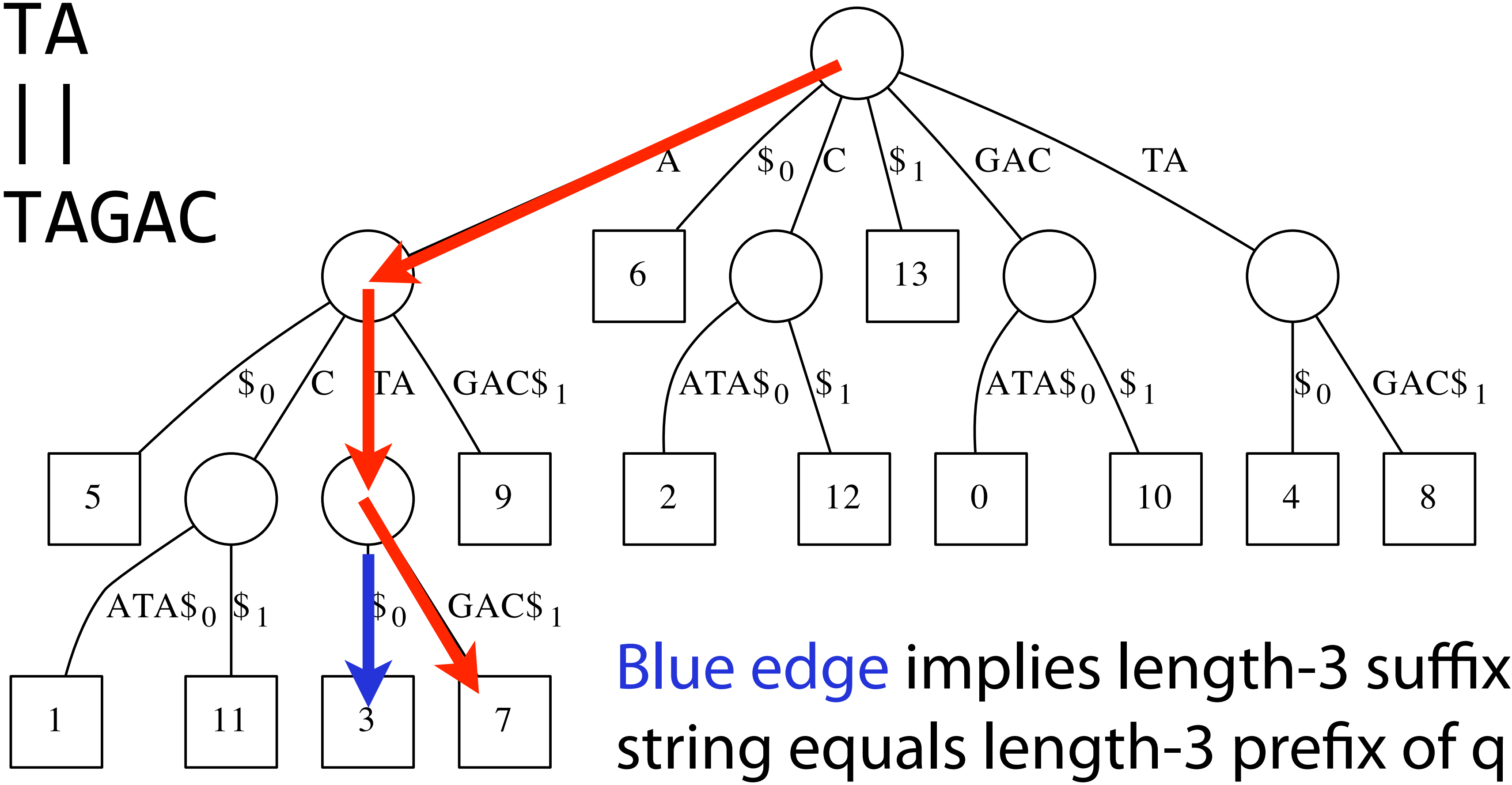
Let query = ATAGAC. From root, follow path labeled with query.



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**?

D: x y

	ϵ	G	G	C	T	C	T	A	G	G	C	C	C
ϵ													
C													
T													
C													
G													
G													
C													
C													
C													
T													
A													
G													
G													

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

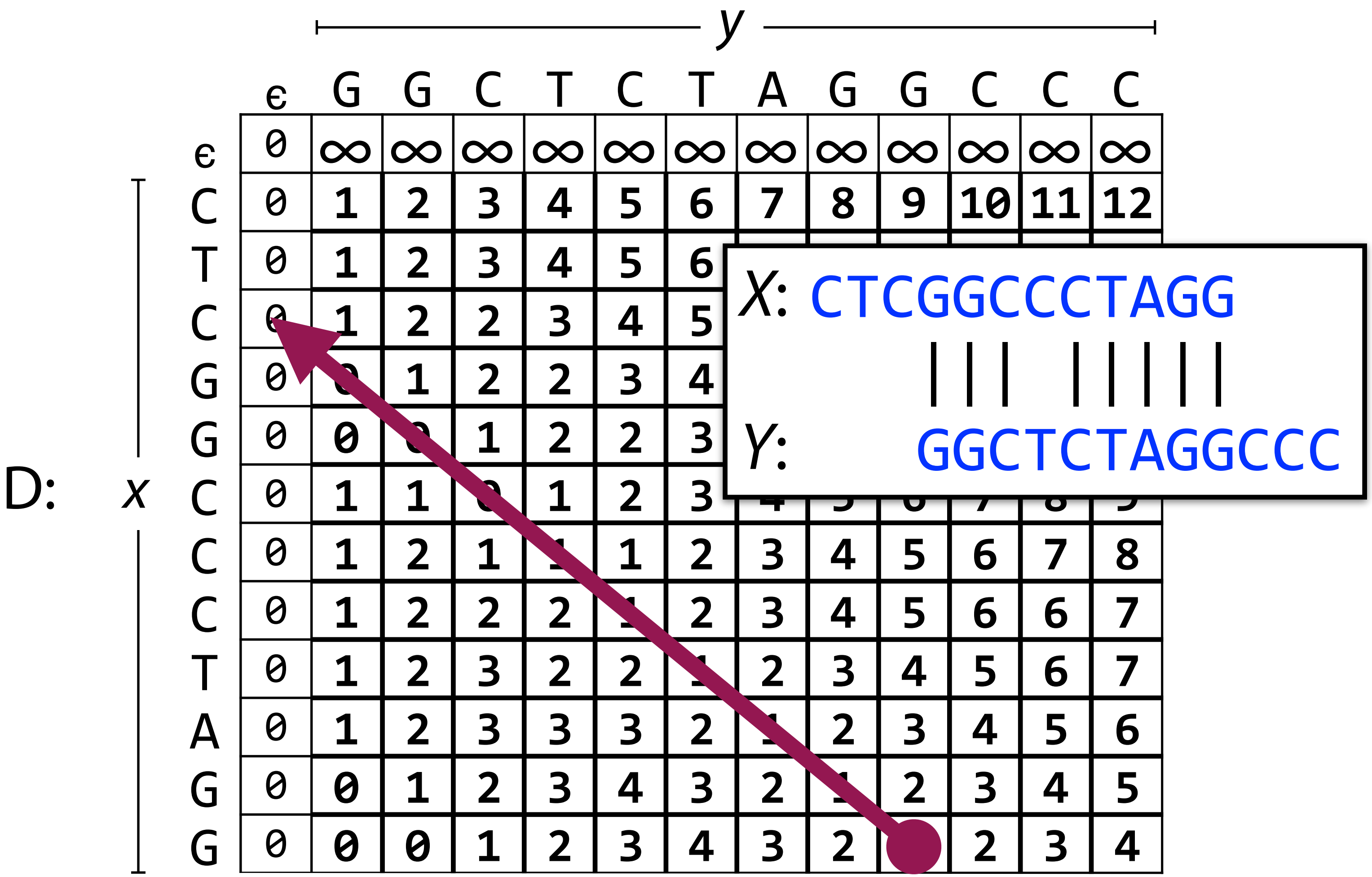
Identify all possible overlaps

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

First row gets ∞s

First column gets 0s

Backtrace from last row



	ϵ	G	G	C	T	C	T	A	G	G	C	C	C
ϵ	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	0	1	2	3	4	5	6	7	8	9	10	11	12
T	0	1	2	3	4	5	6	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
C	0	1	2	2	3	4	5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
G	0	0	1	2	2	3	4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
G	0	0	0	1	2	2	3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
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	[illegible]	2	3	4

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

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?

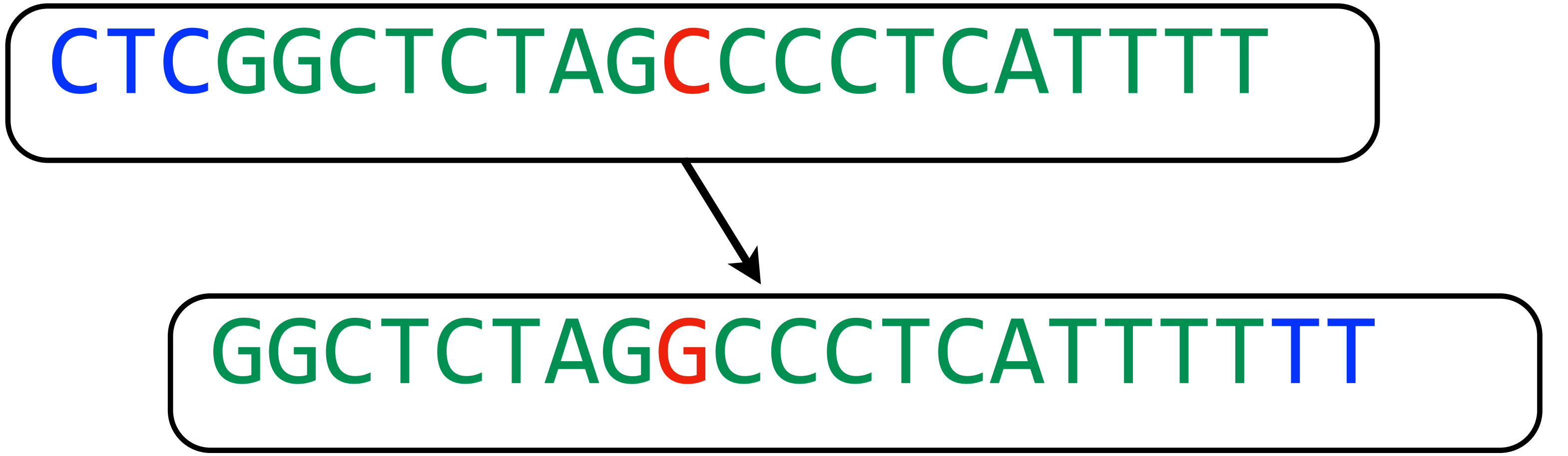
Storing and assembling overlaps

How do we store all our overlaps?

Each node is a string

CTCGGCTCTAGCCCCTCATTTT

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

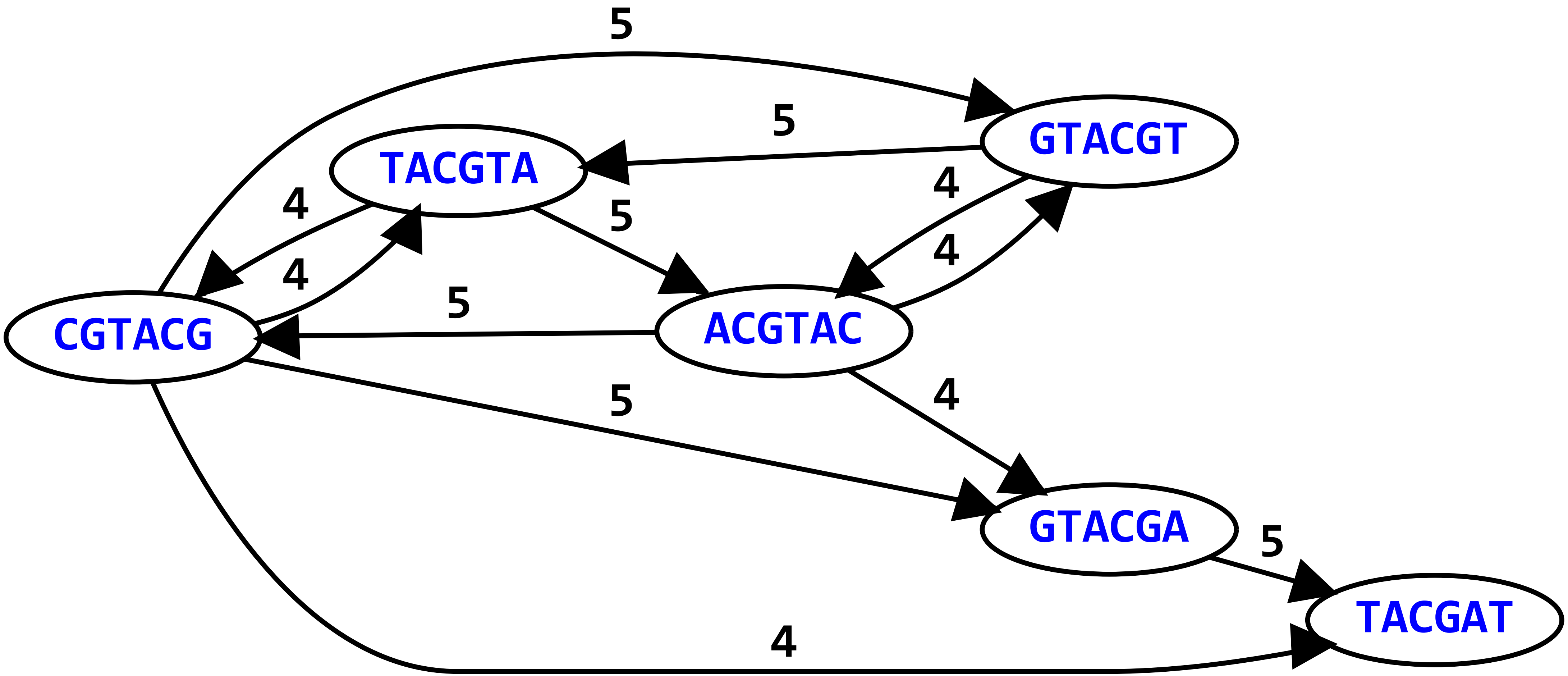


CTCGGCTCTAGCCCCTCATTTT

GGCTCTAGGCCCTCATTTTTT

Overlap Graph Assembly

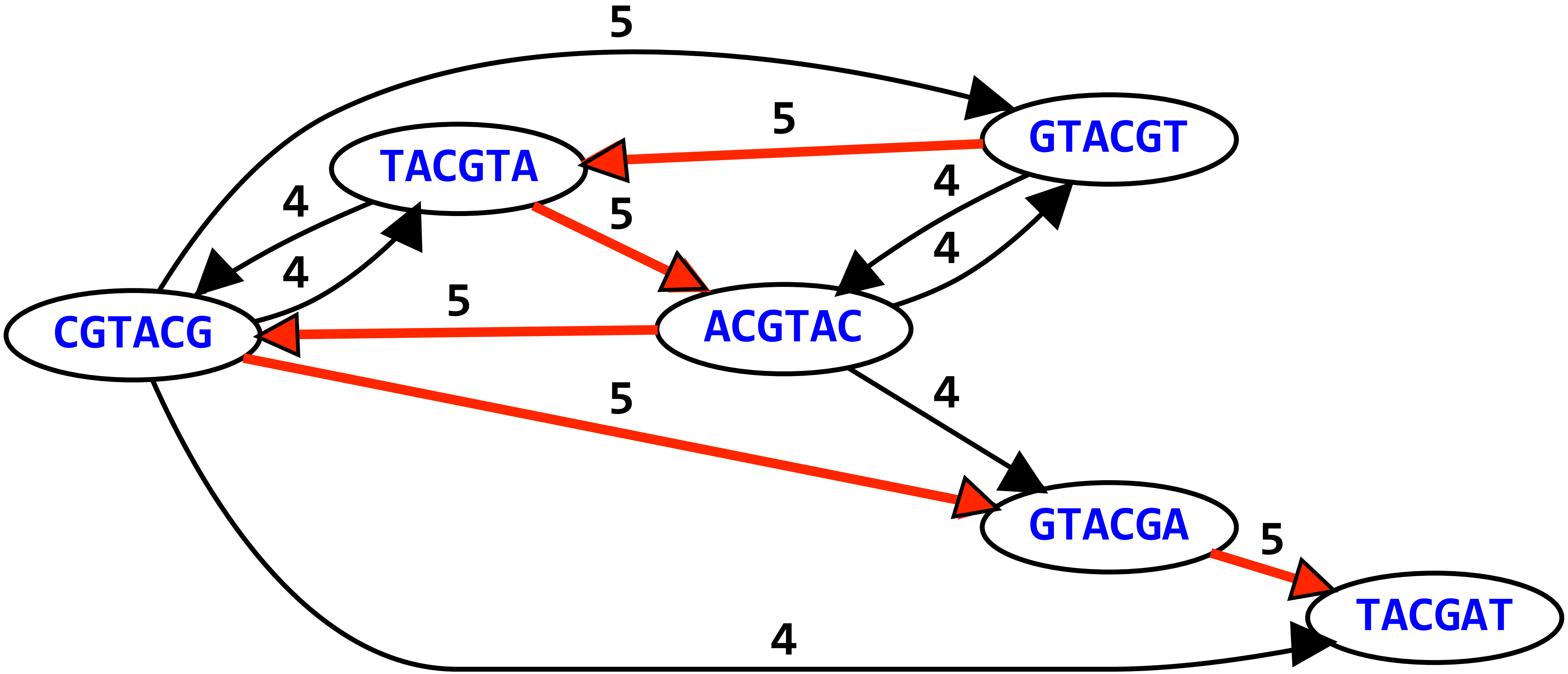
Our best assembly *should* be a path through all nodes in the graph!



Overlap Graph Assembly

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

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

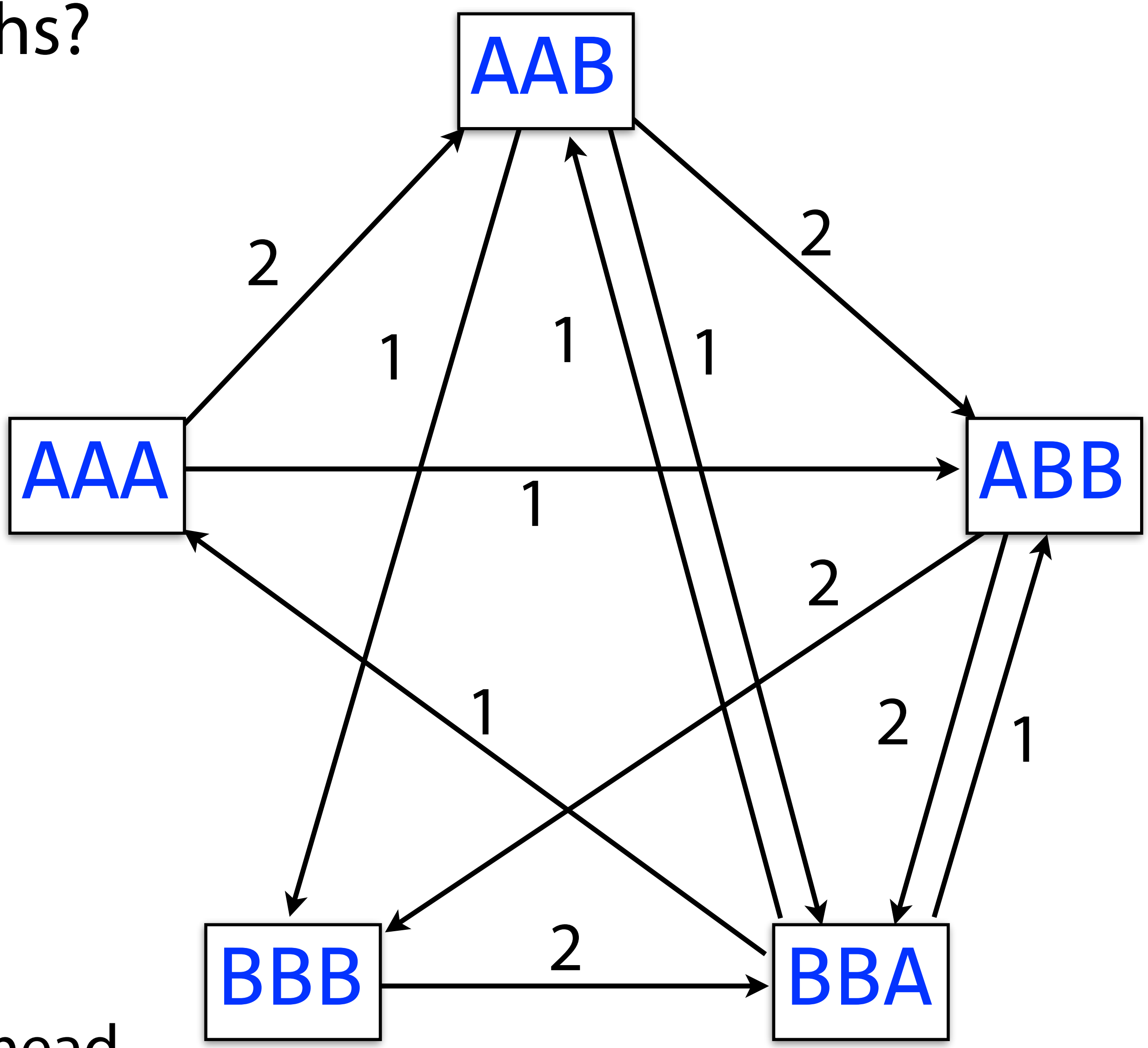


Shortest Common Superstring

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

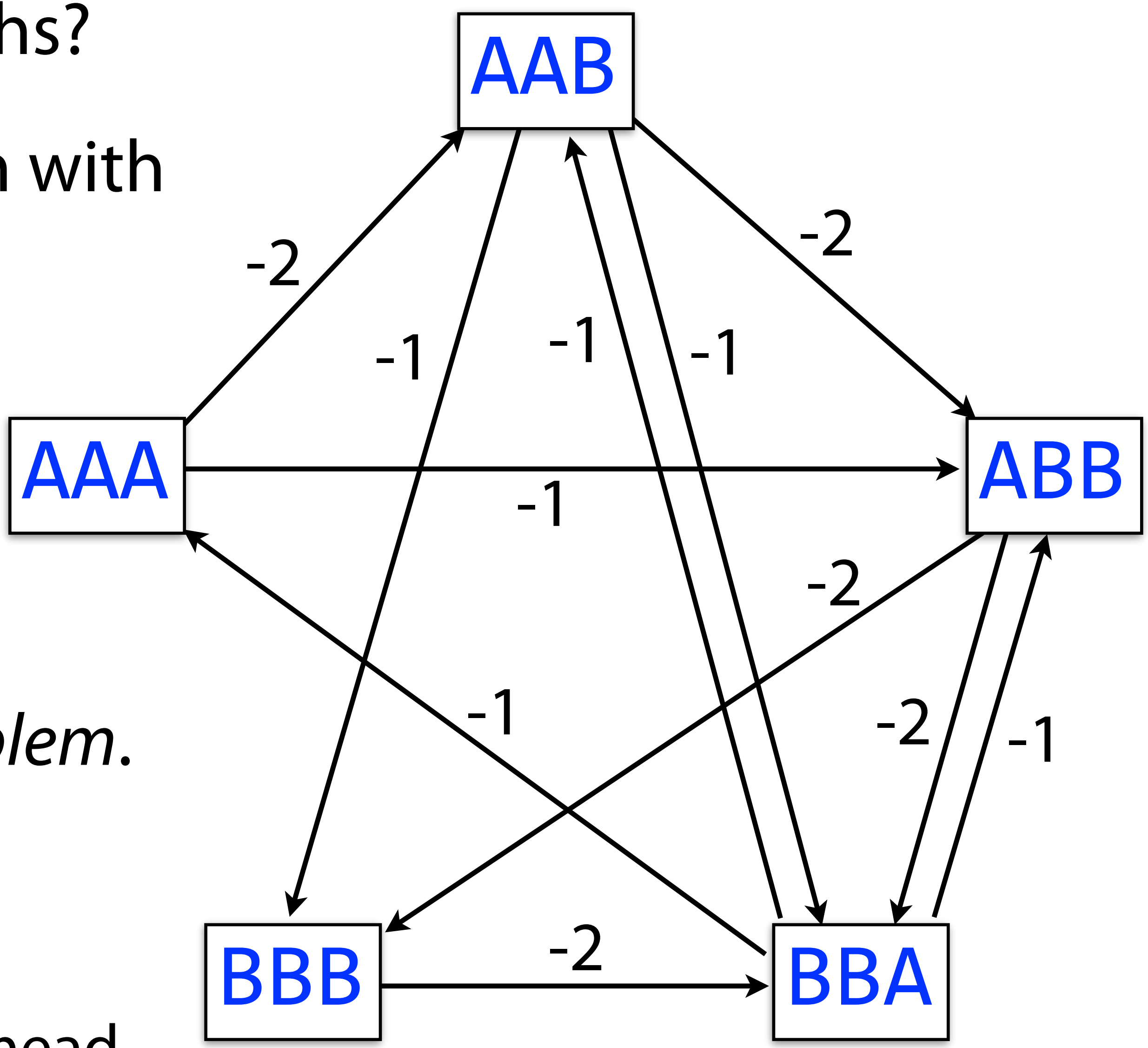
AAA AAB ABB BBB BBA

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

SCS: Greedy

Repeatedly merge pair of strings with maximal overlap.

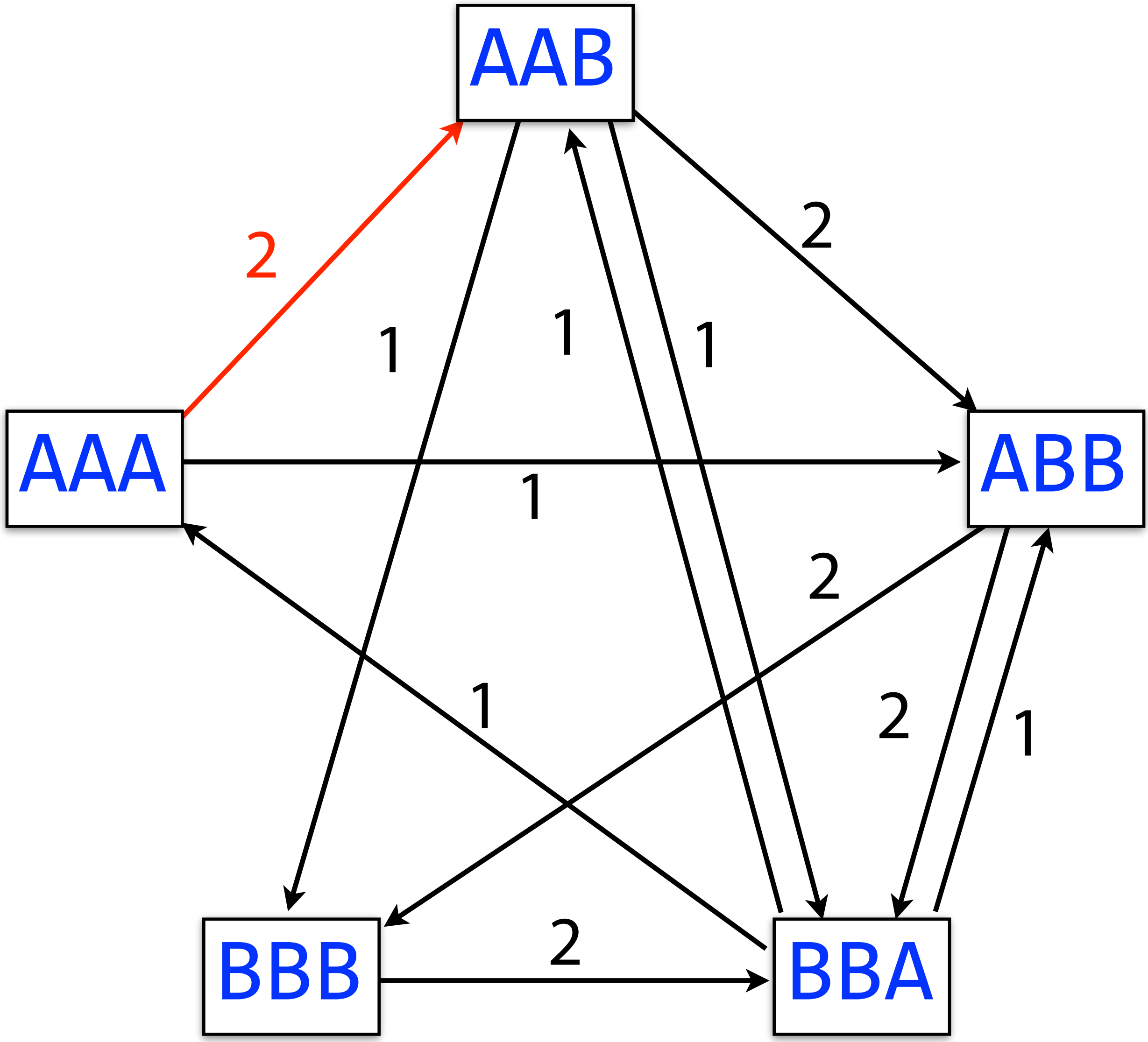
Stop when there's 1 string left.

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.

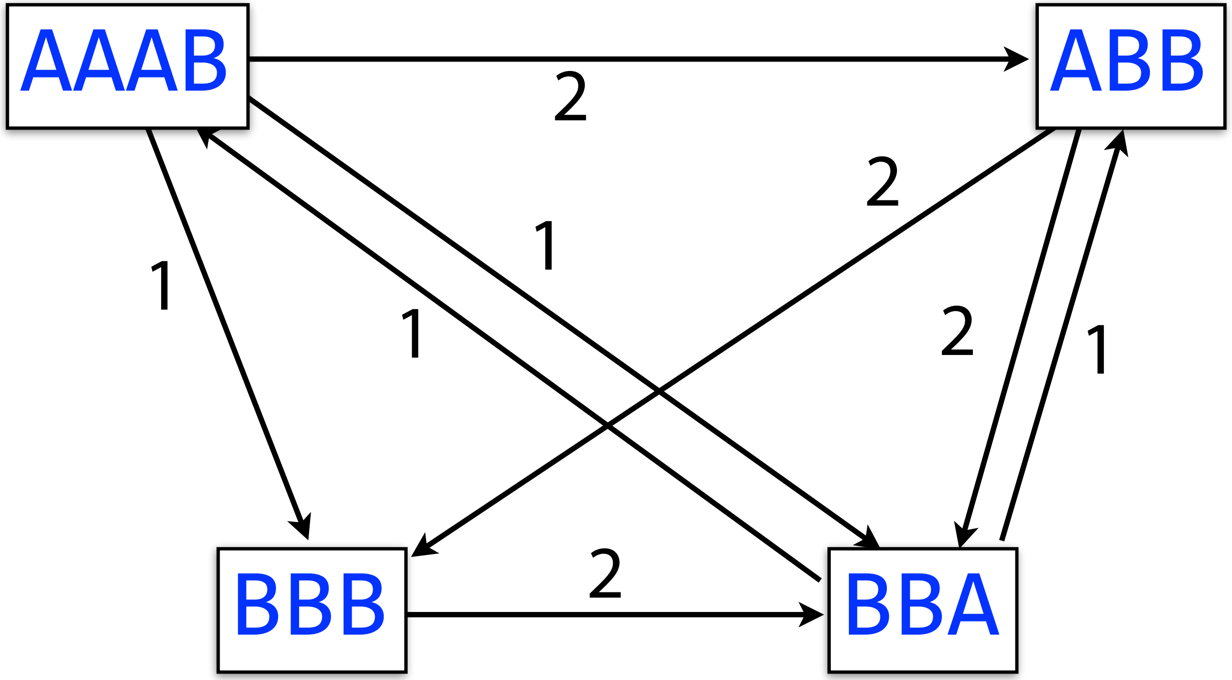
l = minimum overlap.

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.

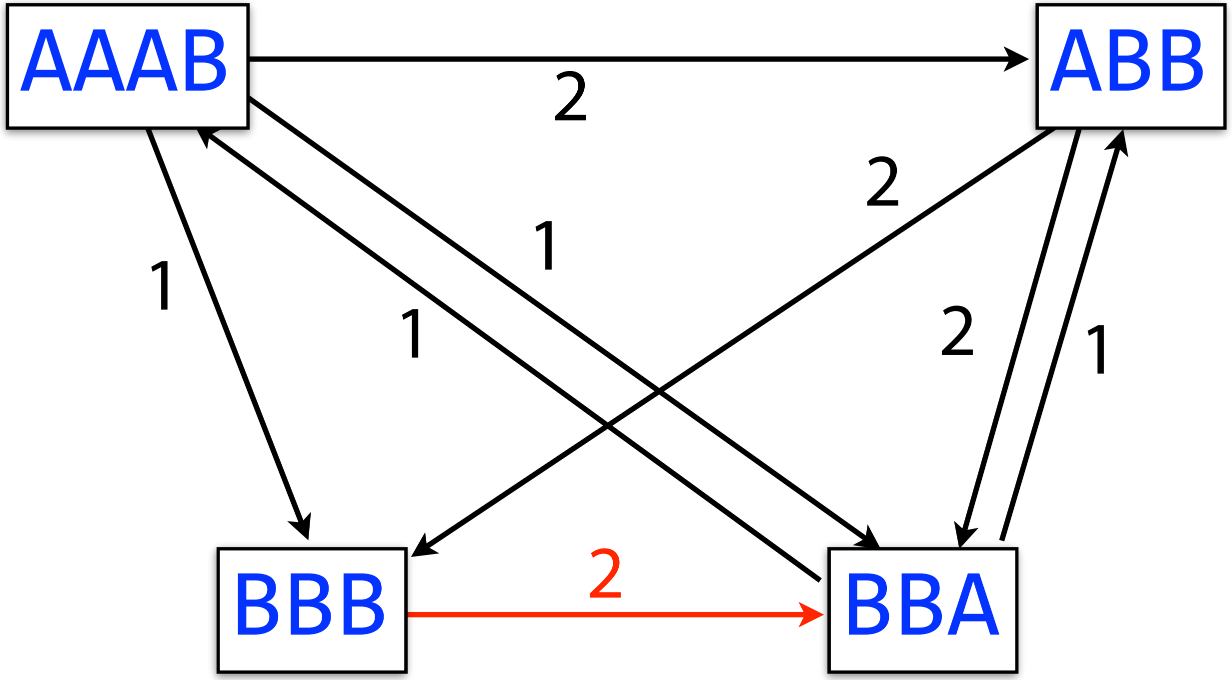
l = minimum overlap.

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.

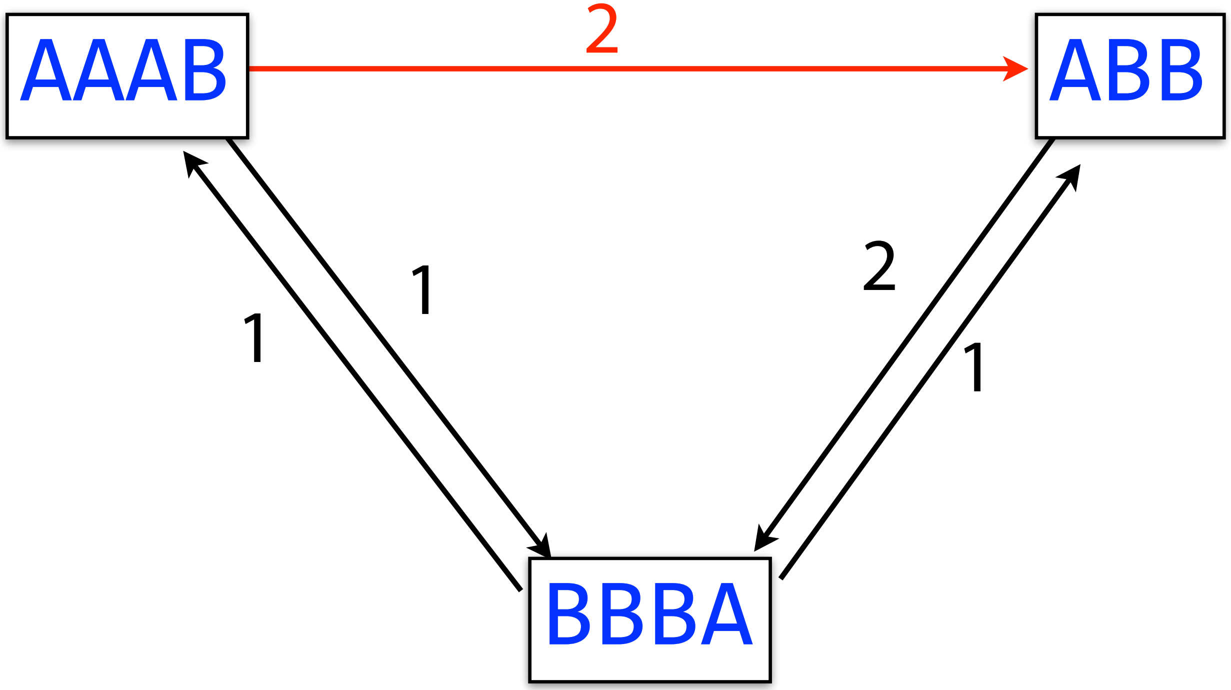
Algorithm in action ($l = 1$):

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 ($l = 1$):

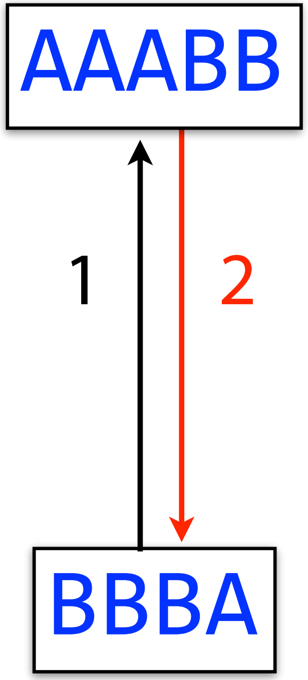
⊢—— Input strings ——⊣

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.

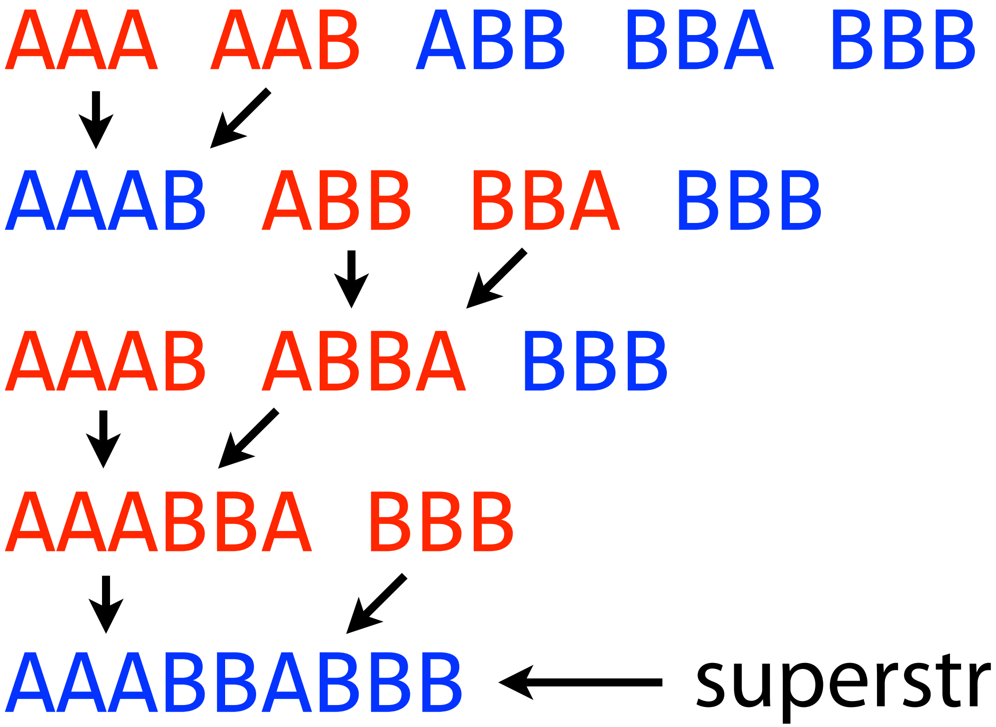
Algorithm in action ($l = 1$):

⊢—— Input strings ——⊣

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

AAABBBA

SCS: Greedy



AAA AAB ABB BBA BBB

AAAB ABB BBA BBB

AAAB ABBA BBB

AAABBA BBB

AAABBABBB ← superstring, length=9

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

`a_long_long_long_time`

↓ *collapse*

`a_long_long_time`

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

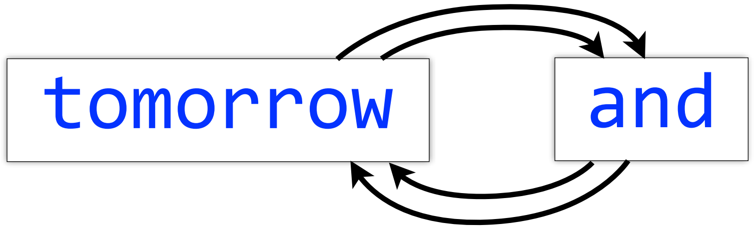
Build contigs over what we know for certain, ignore the rest

If we had another week…

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

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

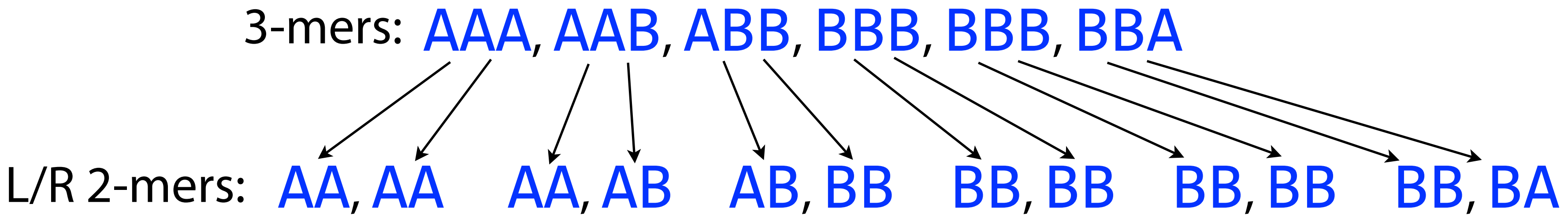
"`tomorrow and tomorrow and tomorrow`"



If we had another week…

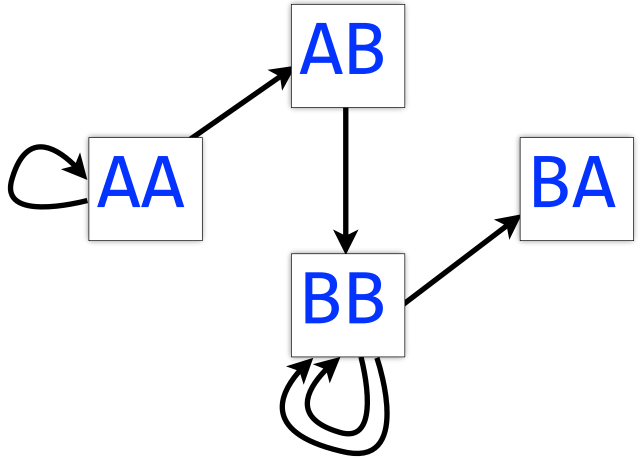
The **De Bruijn Graph!**

Text *T*: AAABBBBA



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

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



One edge per *k*-mer

One node per distinct *k*-1-mer

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

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