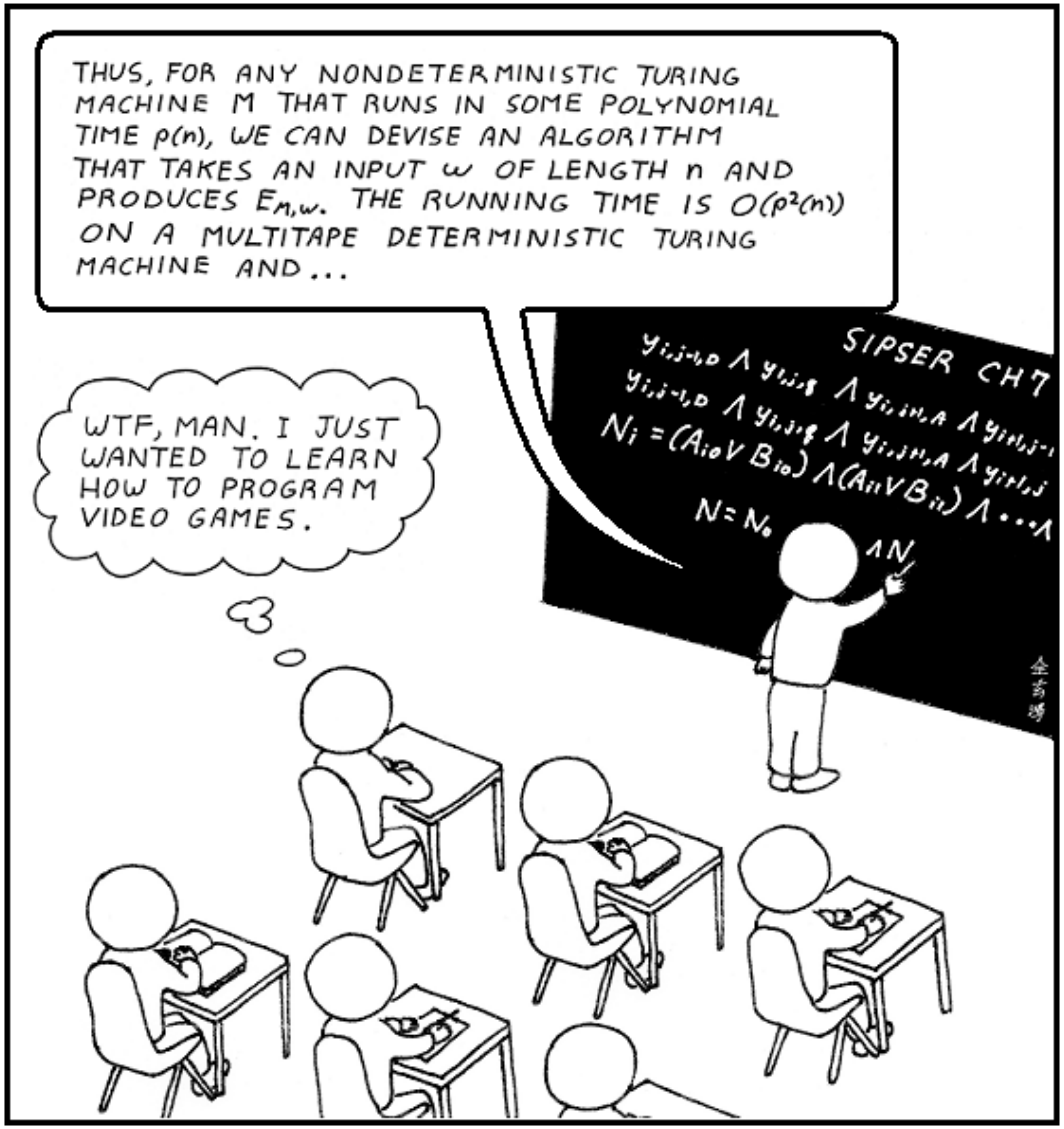
CSE 549: Computational Biology

Computer Science for Biology

Stony Brook University

What is Computer Science?


THUS, FOR ANY NONDETERMINISTIC TURING MACHINE M THAT RUNS IN SOME POLYNOMIAL TIME p(n), WE CAN DEVISE AN ALGORITHM THAT TAKES AN INPUT w OF LENGTH n AND PRODUCES E_{M,w}. THE RUNNING TIME IS O(p²(n)) ON A MULTITAPE DETERMINISTIC TURING MACHINE AND...
WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.
SIPSER CH7

http://people.cs.pitt.edu/~kirk/cs2110/computer_science_major.PNG

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* …

What isn't Computer Science?

Don't install operating systems (may develop them)

Don't set up the office network (may study / design network protocols)

Not about Hacking together a program or learning a web-framework — programming ≠ CS (may study formal languages and develop new programming languages, and programming is a skill many computer scientists learn / master.)

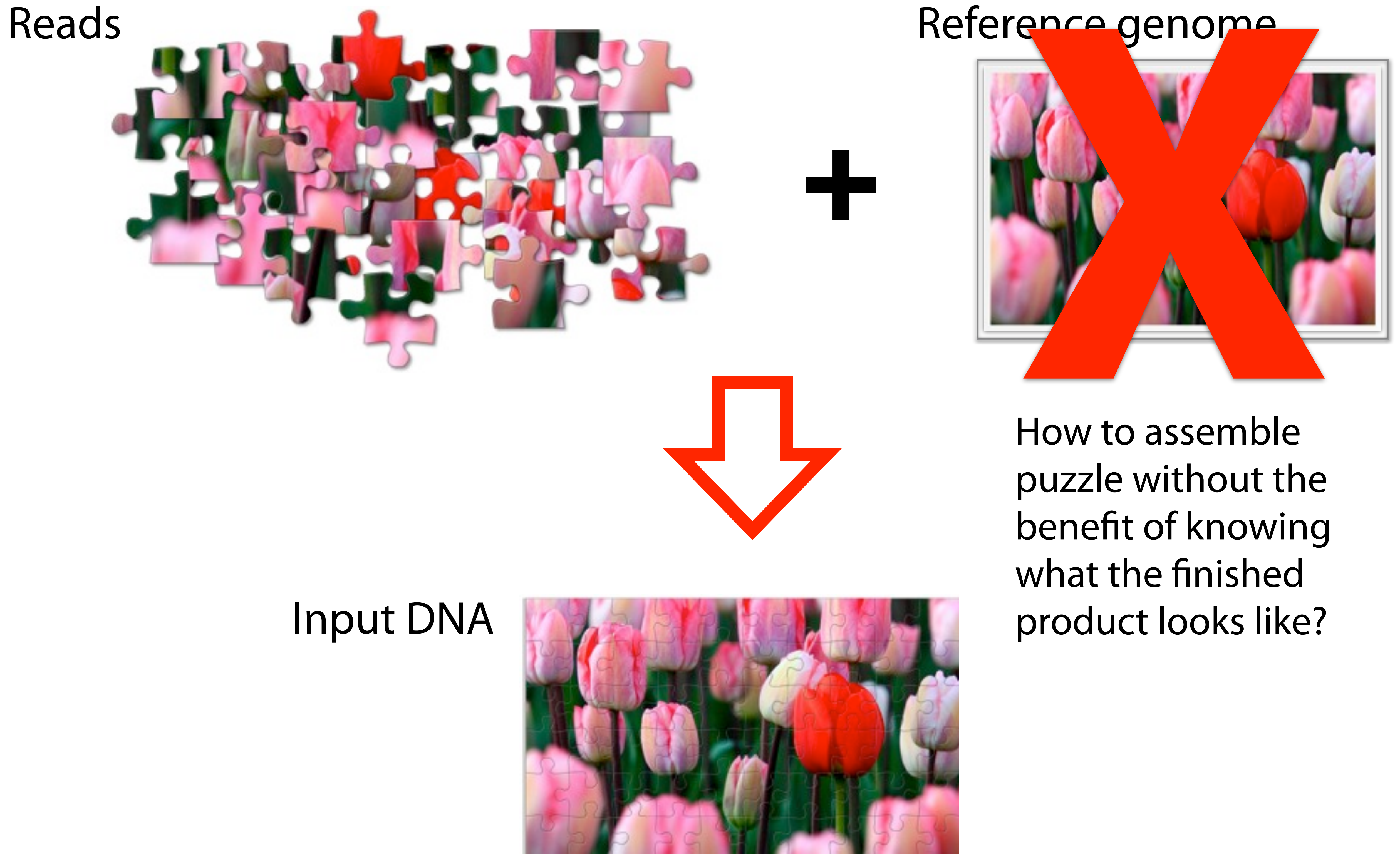
*http://www.cs.bu.edu/AboutCS/WhatIsCS.pdf

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.

Assembly



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

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
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
```

From these

Reconstruct this

```
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Assembly

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

Reconstruct this

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

From these

```
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Assembly

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

From these

Reconstruct this

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

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.

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

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

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.

~~**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?

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.

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

$\underbrace{\qquad\qquad\qquad\qquad}_{24}$

SCS(S): AAABBBABAA

$\underbrace{\qquad\qquad}_{10}$

```
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

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

AAA

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

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

AAAB

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

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

AAABA

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

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

AAABABB

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

order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB $\longleftarrow$ superstring 1

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 *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

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

Shortest common superstring

Can we solve it?

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

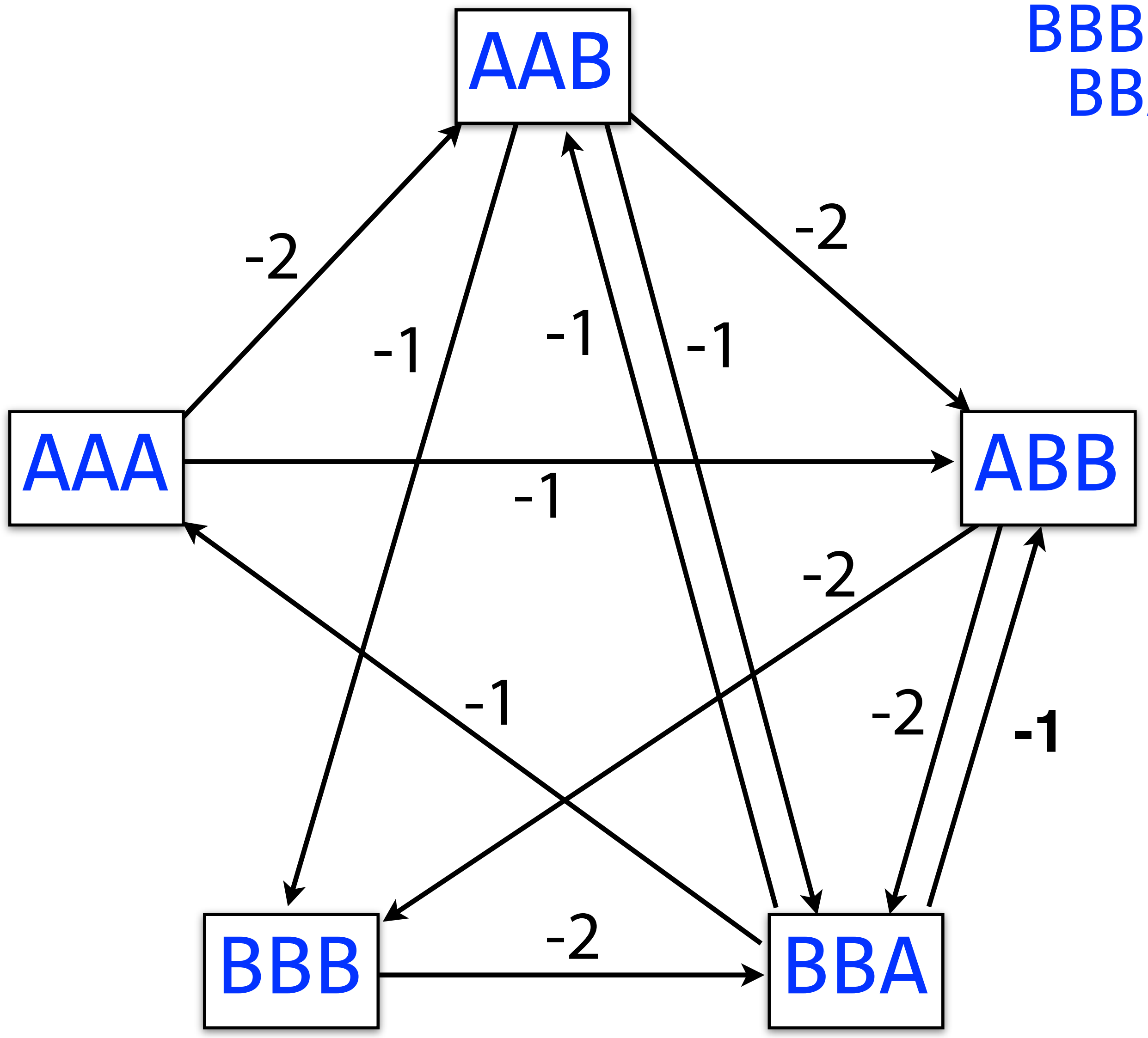
SCS corresponds to a path that visits every node once, minimizing total cost along path

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

S: AAA AAB ABB BBB BBA

SCS(S): AAABBBA

```
AAABBBA
AAA
 AAB
  ABB
   BBB
    BBA
```



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

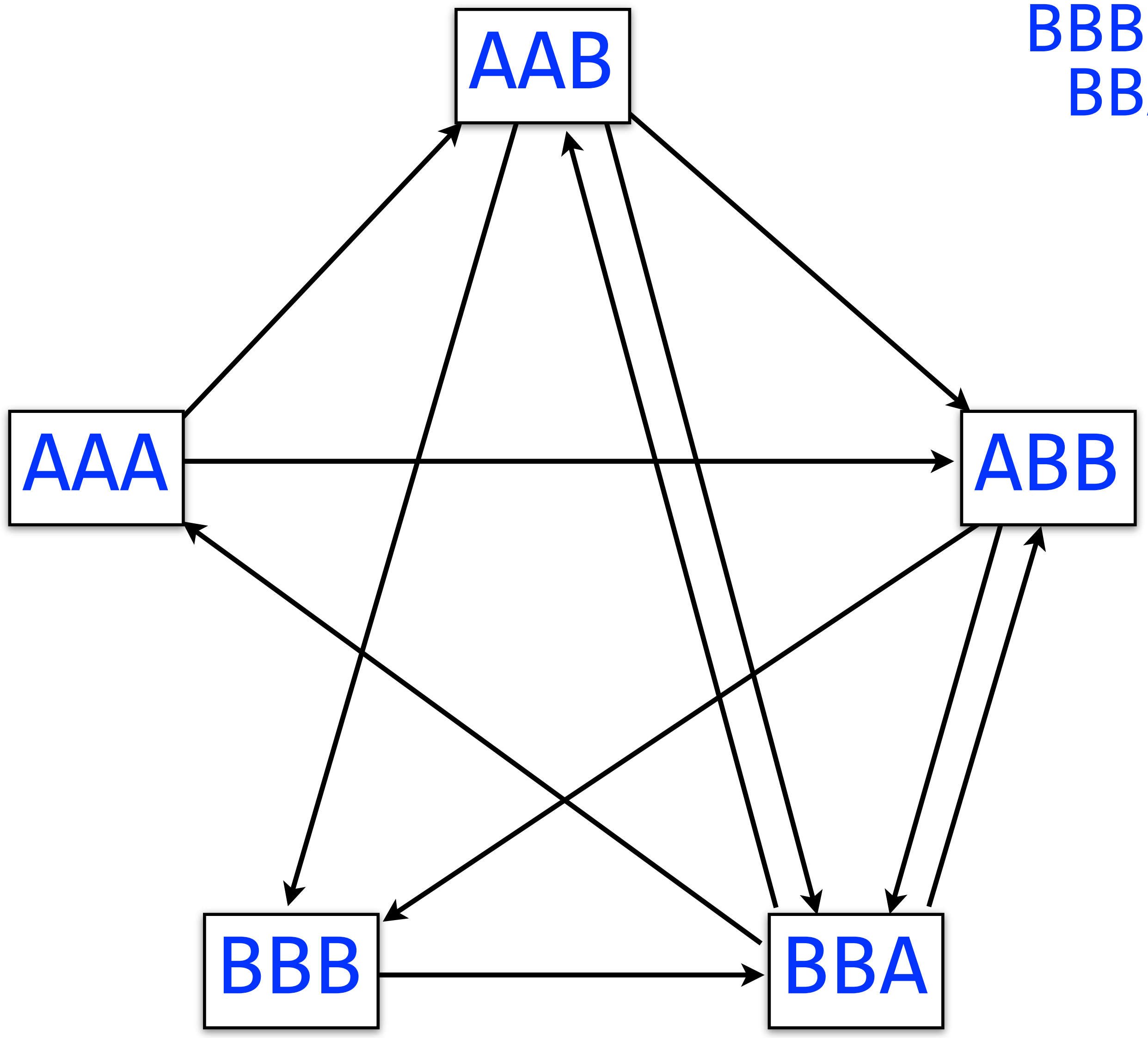
That's the *Hamiltonian Path* problem: NP-complete

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

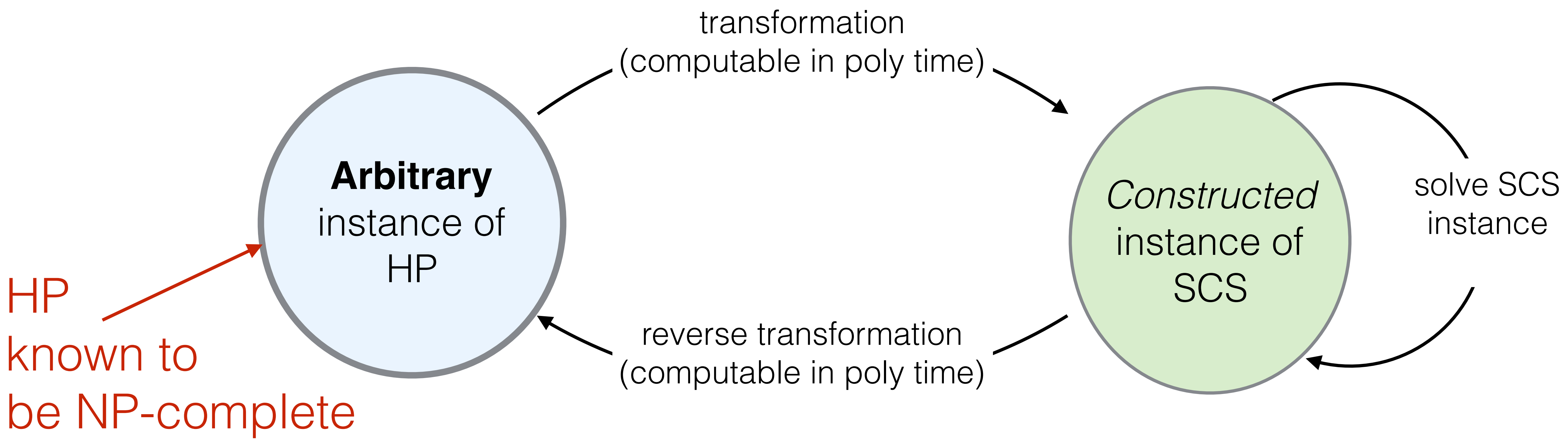
Traveling Salesman, Hamiltonian Path, and Shortest Common Superstring are all NP-hard

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)

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 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 strings.

If we give up on finding the *shortest* possible superstring G, how does the situation change?

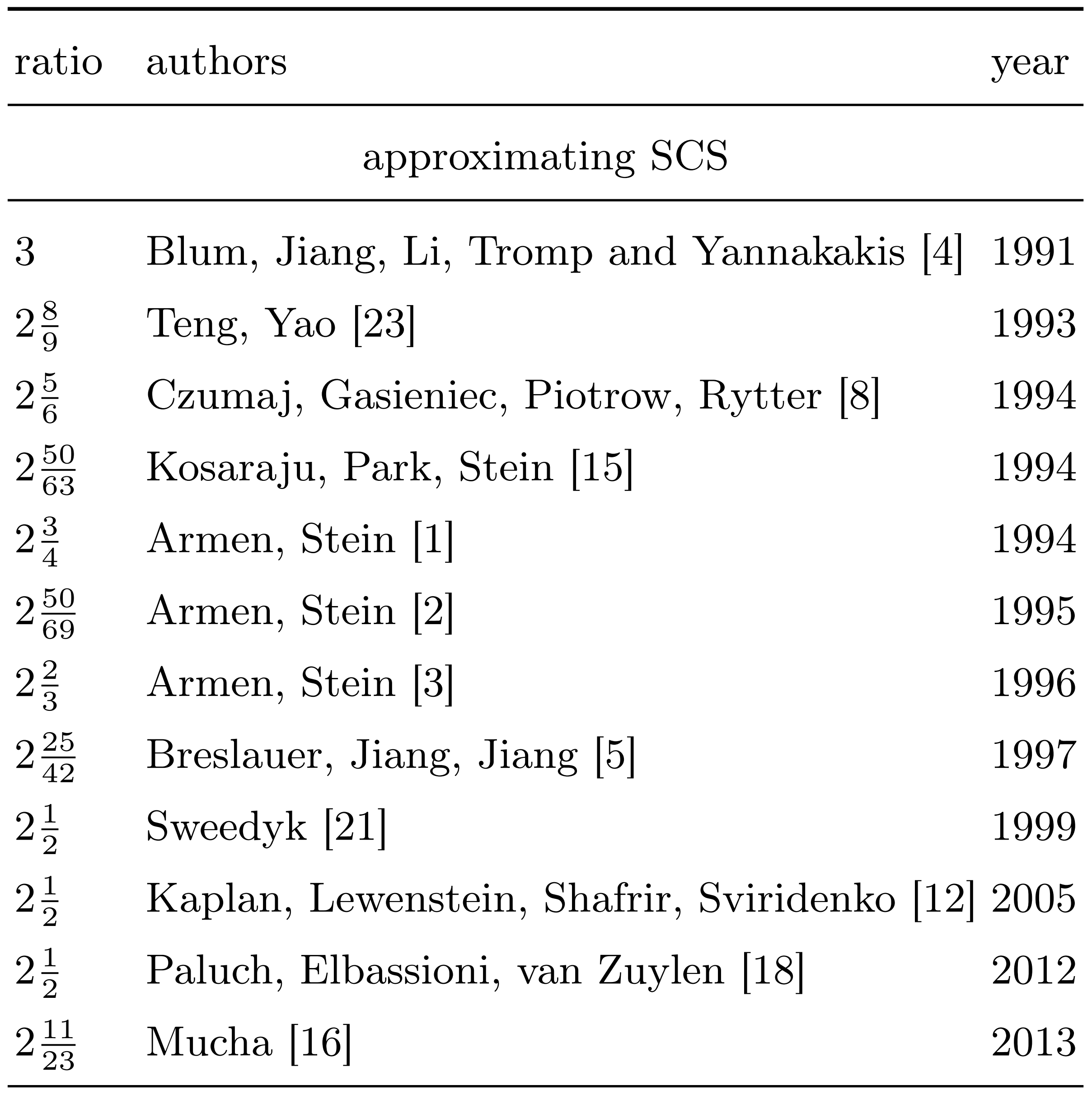
Shortest Common Superstring

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

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

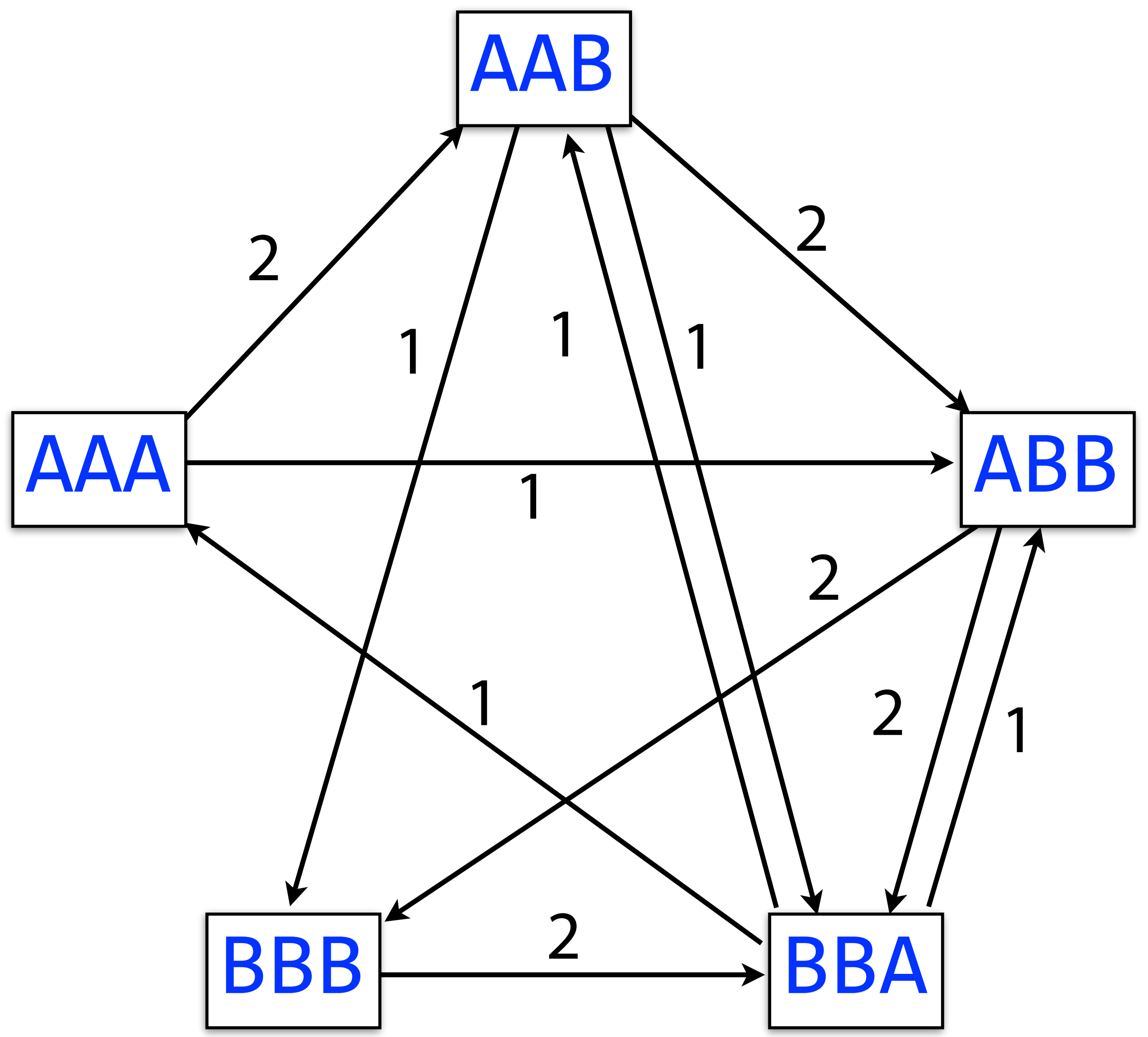
Different approx. (**not all greedy**)



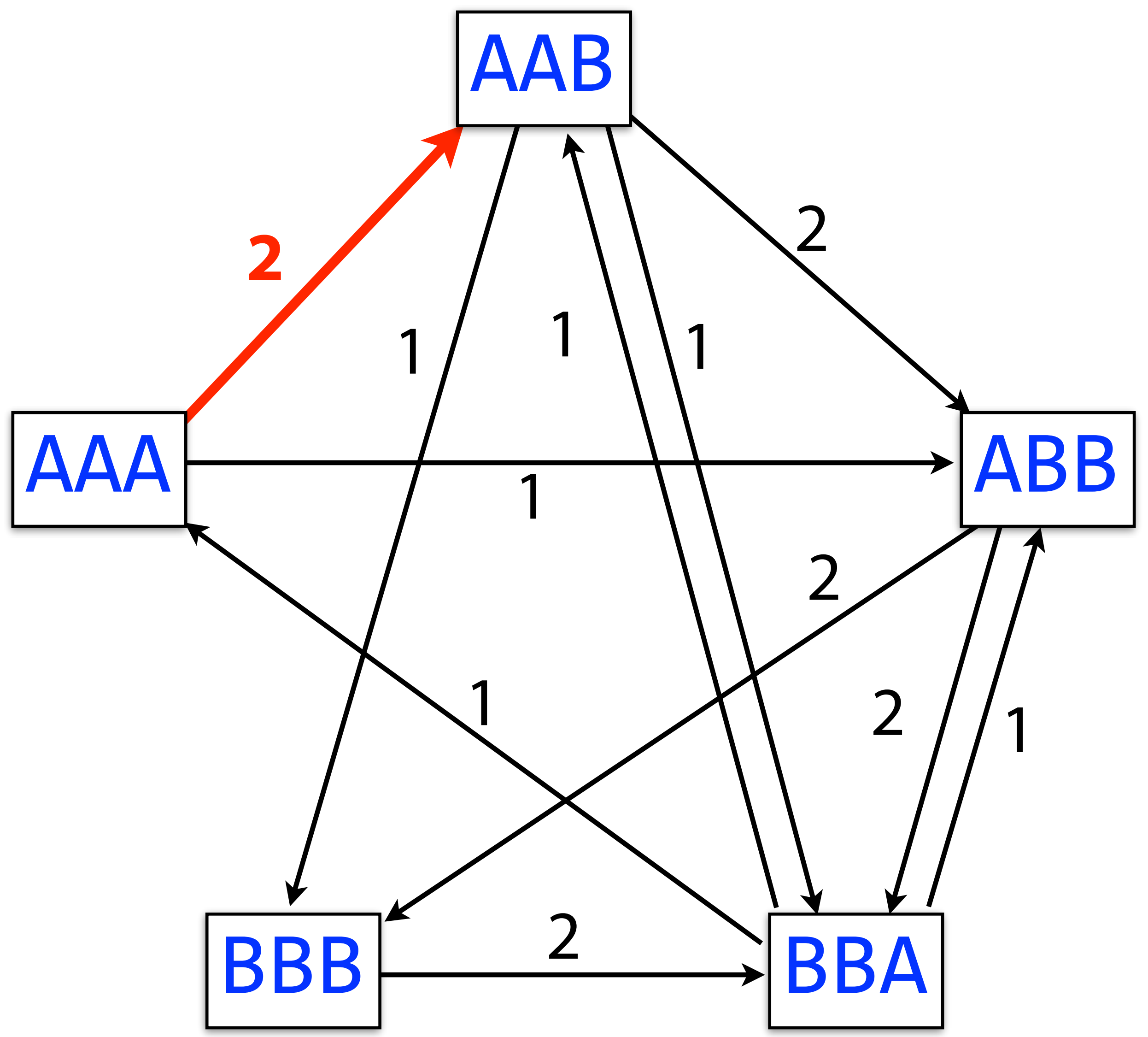
ratio	authors	year
	approximating SCS	
3	Blum, Jiang, Li, Tromp and Yannakakis [4]	1991
$2\frac{8}{9}$	Teng, Yao [23]	1993
$2\frac{5}{6}$	Czumaj, Gasieniec, Piotrow, Rytter [8]	1994
$2\frac{50}{63}$	Kosaraju, Park, Stein [15]	1994
$2\frac{3}{4}$	Armen, Stein [1]	1994
$2\frac{50}{69}$	Armen, Stein [2]	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

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

Greedy shortest common superstring



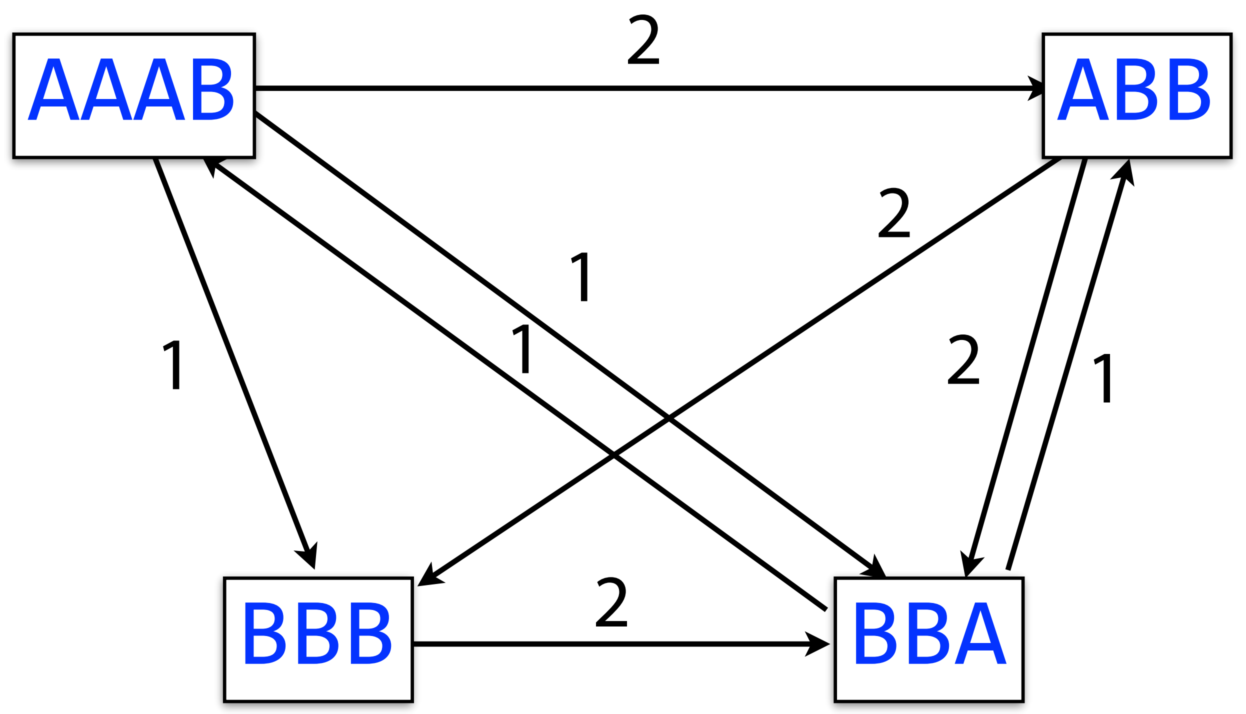
Greedy shortest common superstring



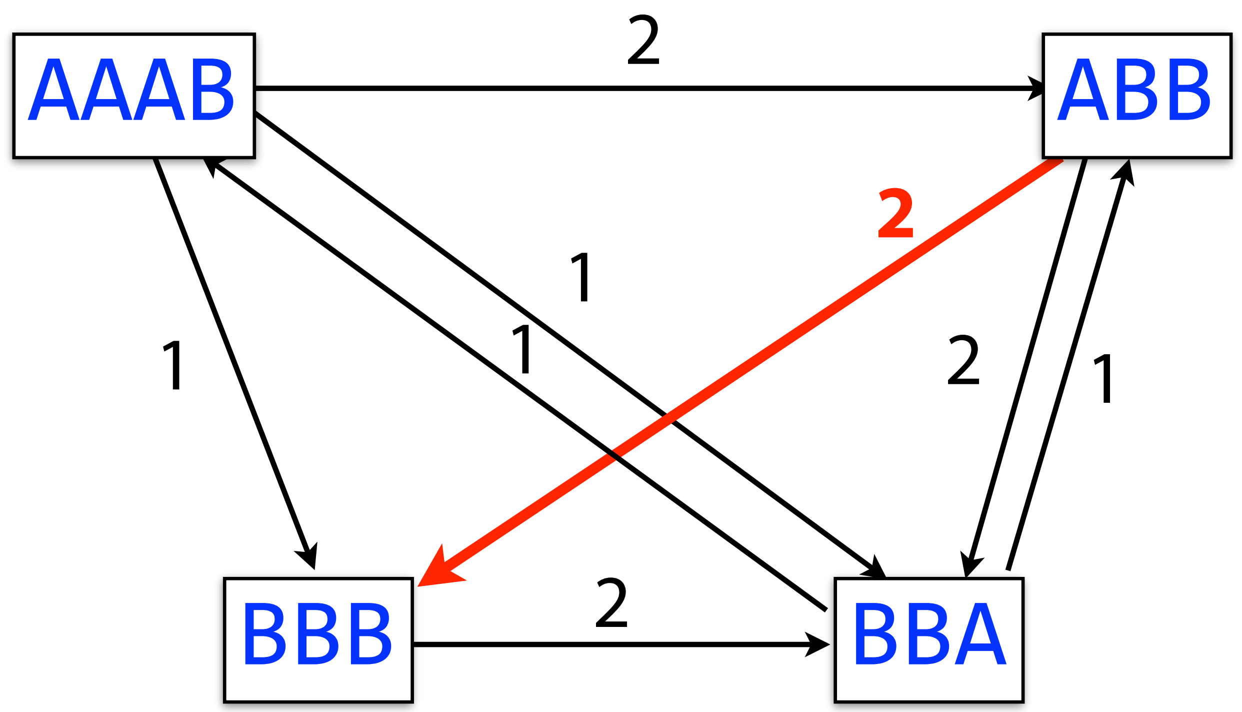
Slide courtesy of Ben Langmead

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

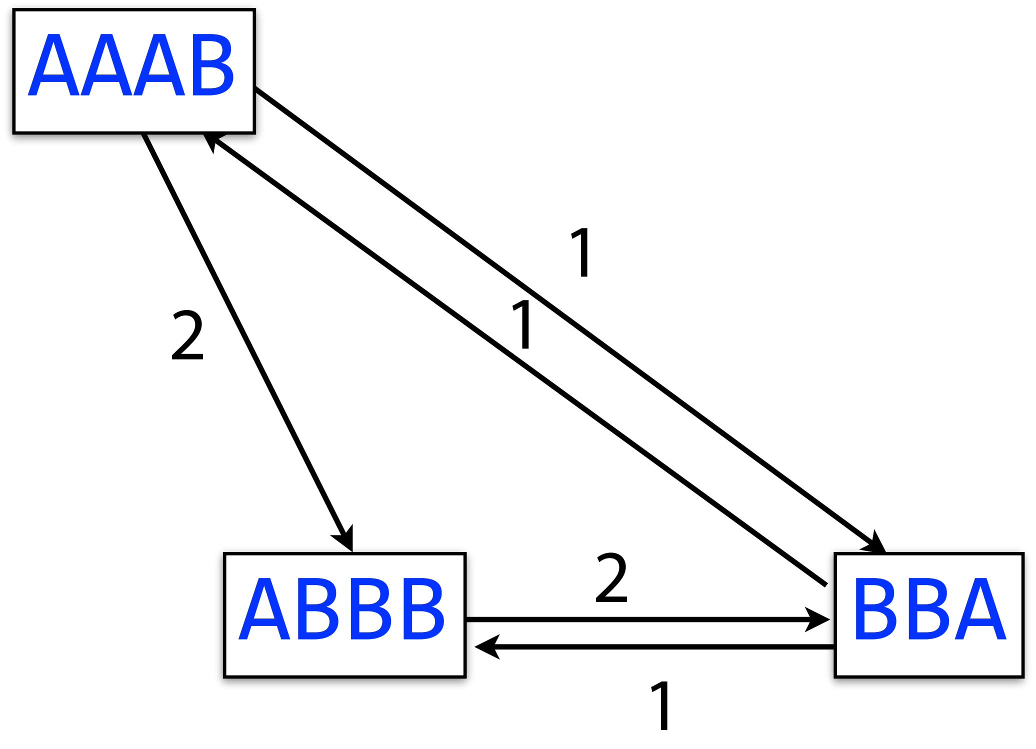
Greedy shortest common superstring



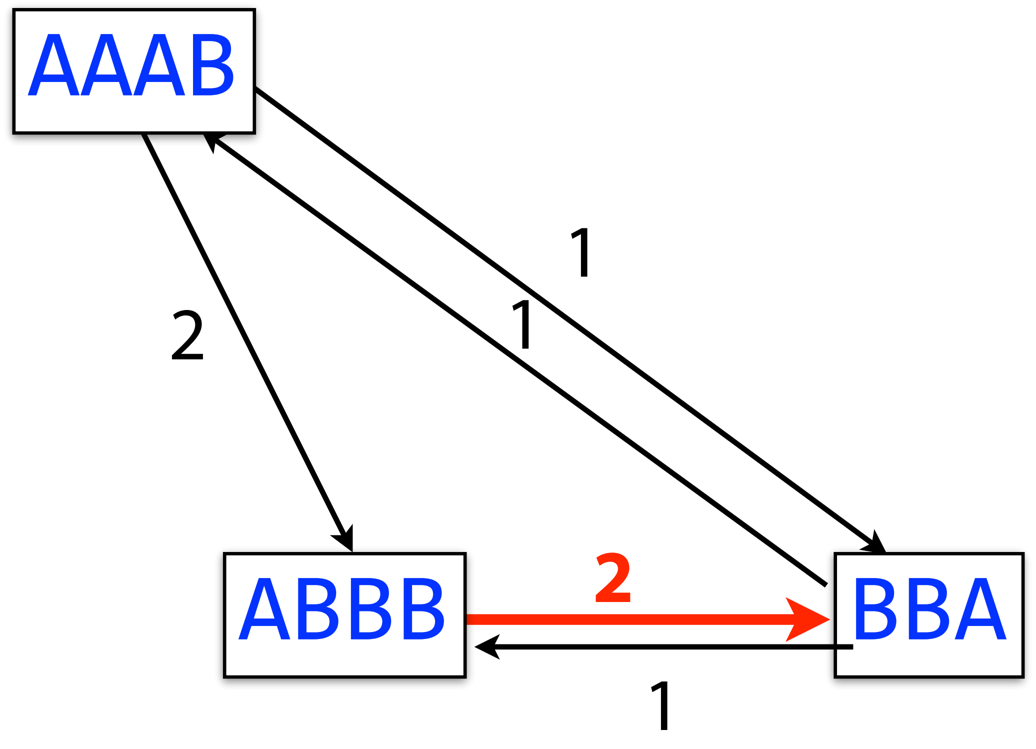
Greedy shortest common superstring


$$\begin{tikzcd}
AAAB \arrow[rr, "2"] \arrow[d, "1"'] \arrow[drr, bend left=10, "1"] & & ABB \arrow[dll, "2"'] \arrow[d, bend right=10, "2"'] \\
BBB \arrow[rr, "2"] & & BBA \arrow[ull, bend left=10, "1"] \arrow[u, bend right=10, "1"']
\end{tikzcd}$$

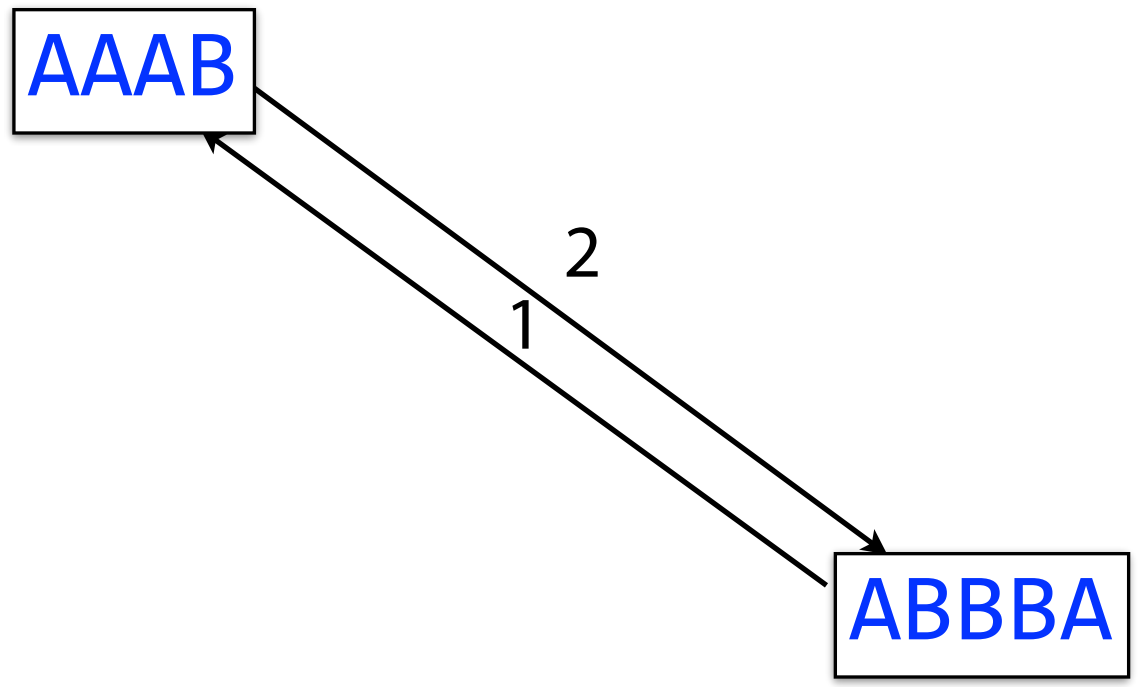
Greedy shortest common superstring



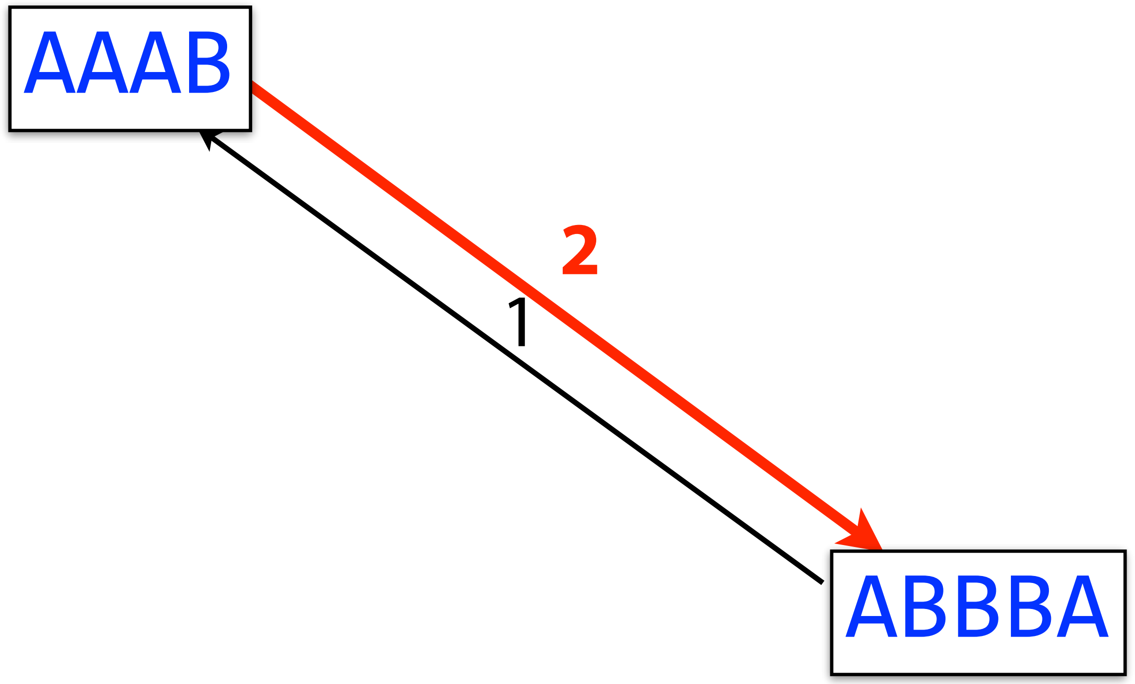
Greedy shortest common superstring



Greedy shortest common superstring



Greedy shortest common superstring



Greedy shortest common superstring

$\boxed{\text{AAABBBA}} \longleftarrow$ superstring, length=7

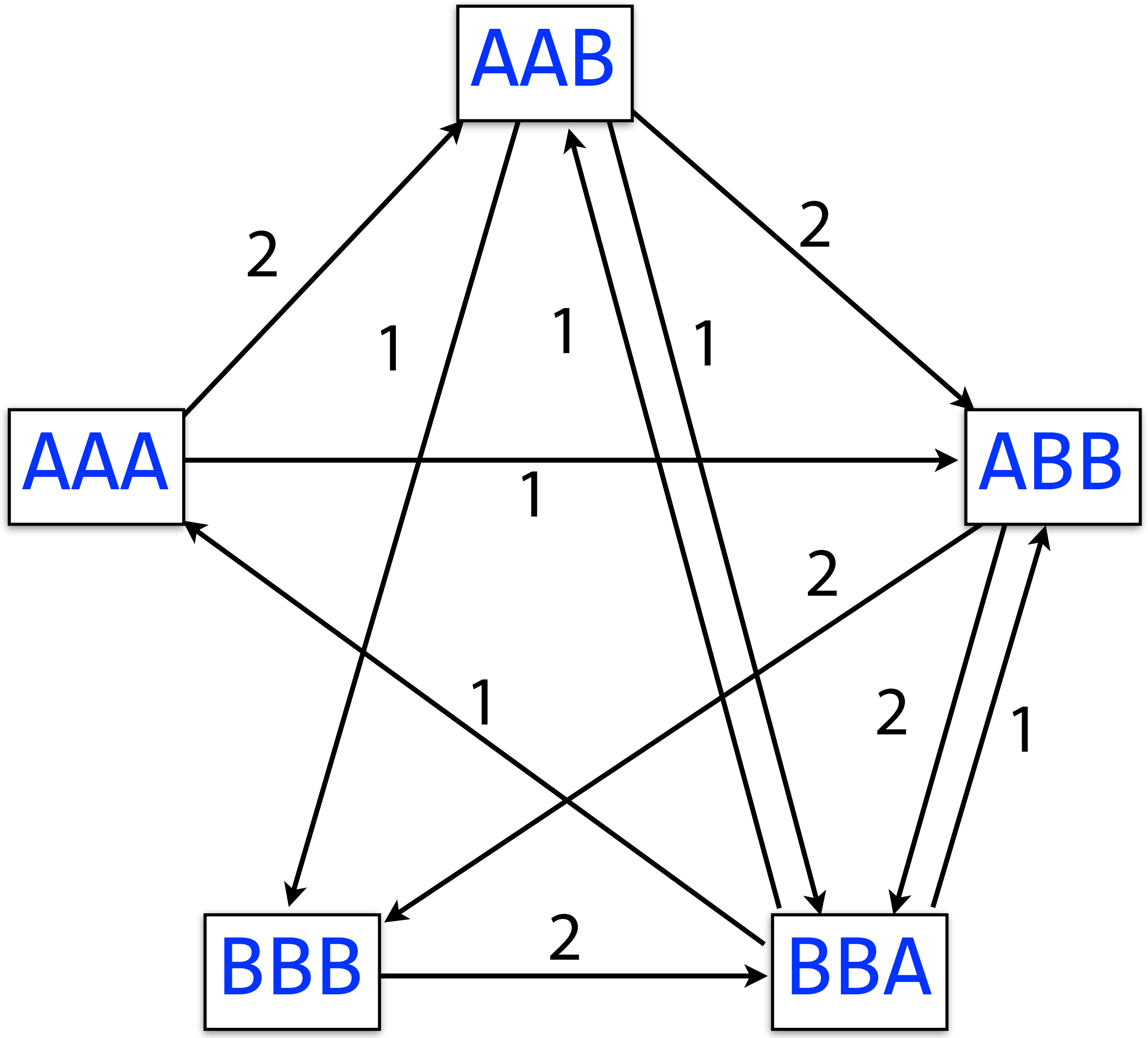
Shortest common superstring: greedy

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



Slide courtesy of Ben Langmead

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

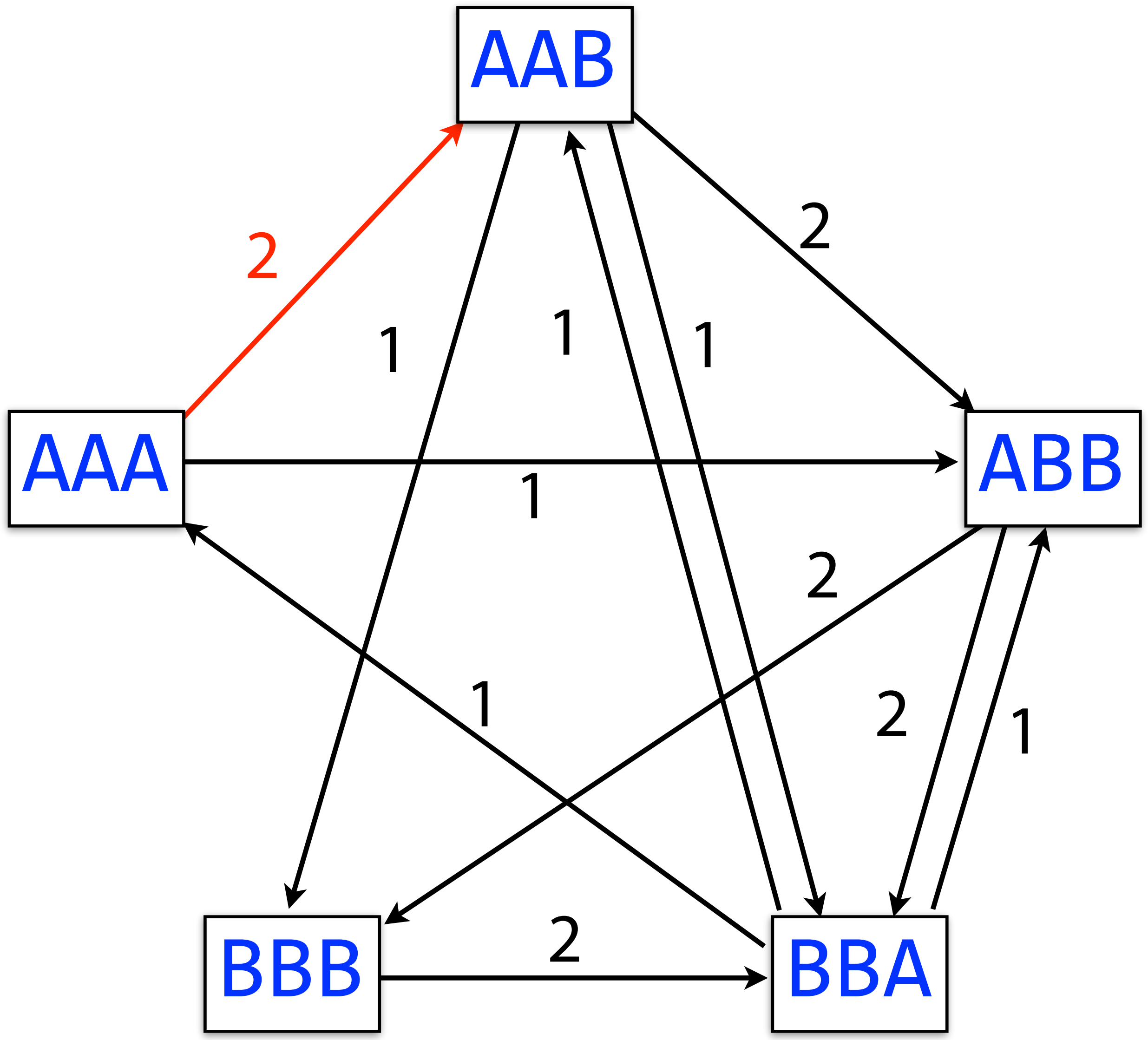
Shortest common superstring: greedy

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



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.

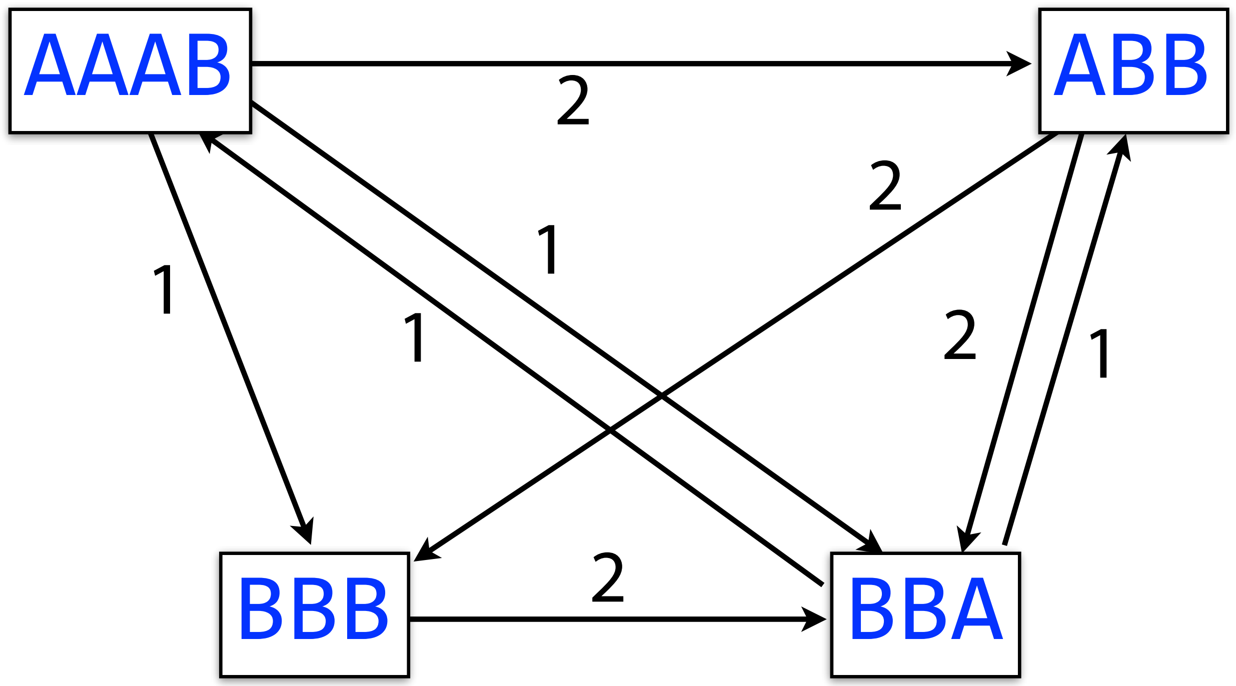
Algorithm in action ($l = 1$):

|———Input strings———|

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



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.

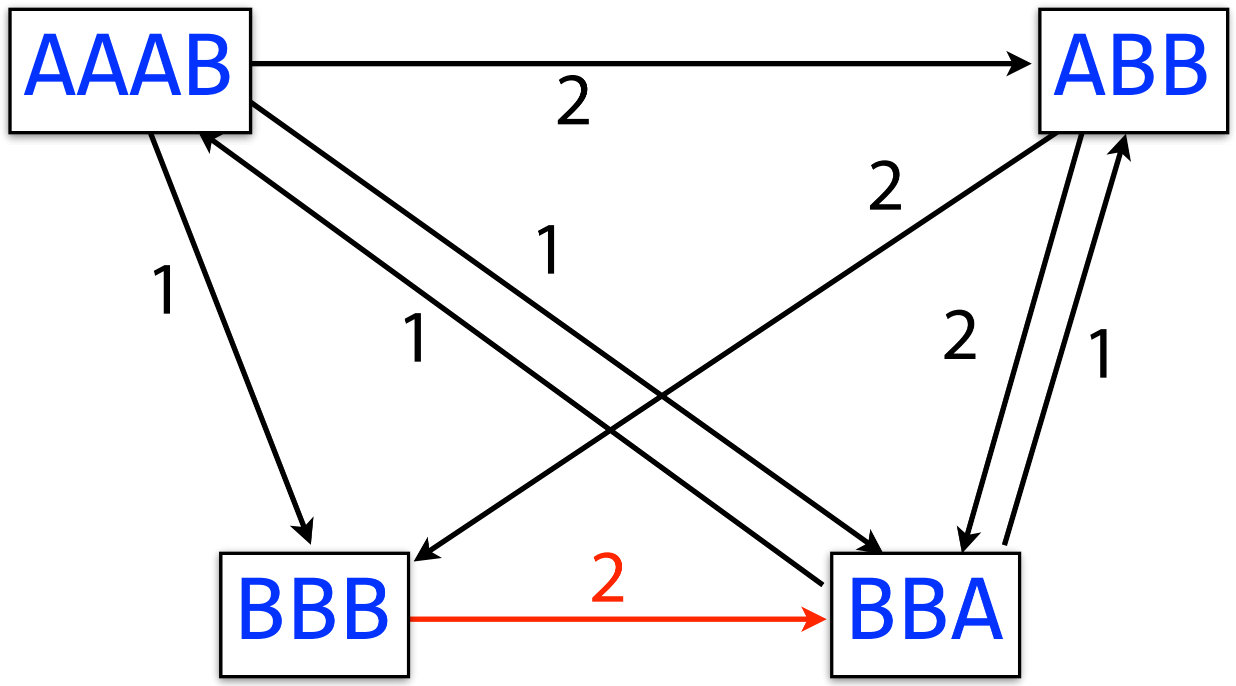
Algorithm in action ($l = 1$):

|—— Input strings ——|

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA



Slide 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 ($l = 1$):

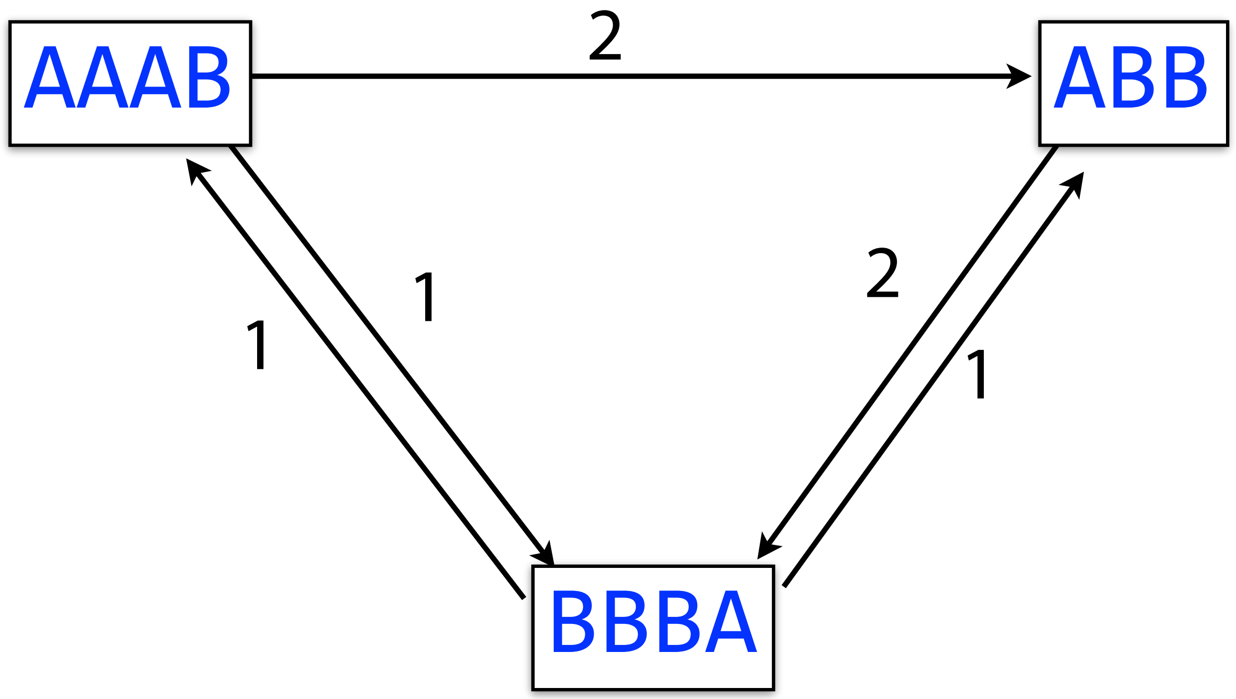
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB



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 ($l = 1$):

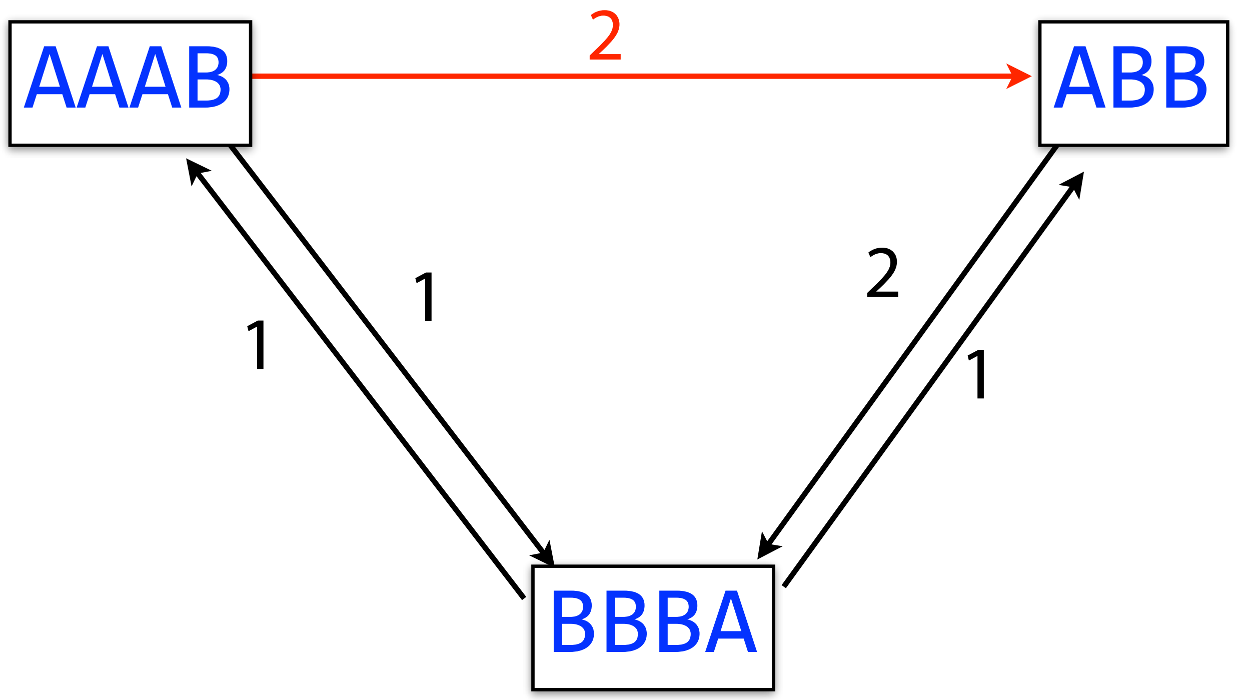
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB



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 ——⊣

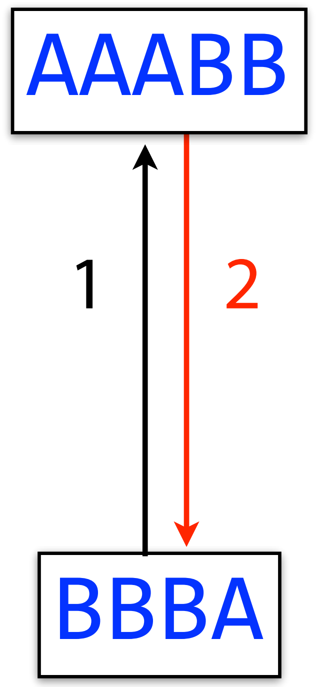
AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB

AAABB BBBA



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

AAABBBA

$\boxed{\text{AAABBBA}}$

That's the SCS

Greedy shortest common superstring

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

Greedy shortest common superstring

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

Greedy shortest common superstring

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

Slide courtesy of Ben Langmead

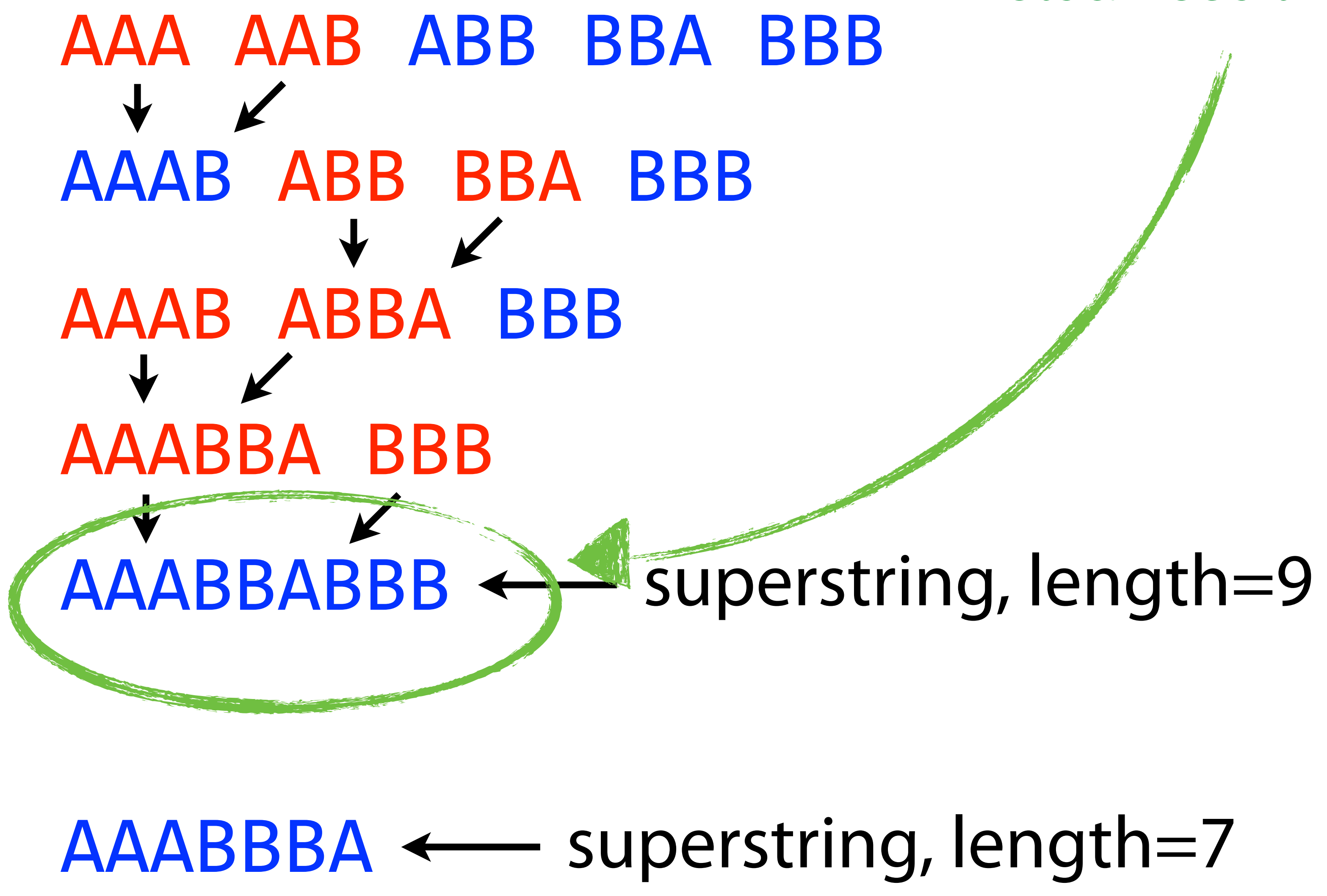
http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf

Greedy shortest common superstring

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

Greedy shortest common superstring

Note: approx. guarantee
is on length of the superstring
Actual result may be very different.



Greedy answer *isn't necessarily optimal*

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.