# Computer Science for Biology (algorithmic primer)

UNIVERSITY OF MARYLAND

# Overview of Quiz 0

**Note**: I grabbed these stats last night, so the distribution may have changed by this morning.

From among the following answers, select the one that best describes how well you believe you know the topic of **asymptotic analysis**.

| | | | |
|---|---|---|---|
| **extremely well** | 3 respondents | 6 % | ✓ |
| **fairly well** | 16 respondents | 30 % | ✓ |
| **so-so** | 19 respondents | 36 % | ✓ |
| **I've heard of it before, but don't really know it** | 4 respondents | 8 % | ✓ |
| **new semester, who dis?** | 13 respondents | 25 % | ✓ |

# Overview of Quiz 0

From among the following answers, select the one that best describes how well you believe you know the topic of **dynamic programming**.

| | | | |
|---|---|---|---|
| **extremely well** | 2 respondents | **4 %** |  |
| **fairly well** | 10 respondents | **19 %** |  |
| **so so** | 28 respondents | **53 %** |  |
| **I've heard of it before, but don't really know it** | 14 respondents | **26 %** |  |
| **new semester, who dis?** | | **0 %** |  |

# Overview of Quiz 0

From among the following answers, select the one that best describes how well you believe you know the topic of **computational complexity**.

| | | | |
|---|---|---|---|
| **I'm pretty good at reductions, and am confident proving new problems NP-complete.** | 2 respondents | 4 % |  |
| **I understand the basic complexity classes, and can follow reduction proofs.** | 25 respondents | 48 % |  |
| **I generally understand what different complexity classes mean, but am not comfortable with the topic.** | 18 respondents | 35 % |  |
| **I am not at all comfortable with basic computational complexity topics.** | 8 respondents | 15 % |  |

# Overview of Quiz 0

Given a directed acyclic graph (DAG) G = (V,E), give a (tight) asymptotic upper bound for the complexity of finding the shortest path between two vertices s, t ∈ V.

Key knowledge: Every DAG has a *topological order* — a way you can visit the nodes so that when you see a node, you've already seen every node that has an edge to it.

# Overview of Quiz 0

Given a directed acyclic graph (DAG) G = (V,E), give a (tight) asymptotic upper bound for the complexity of finding the shortest path between two vertices s, t ∈ V.

Key knowledge: Every DAG has a *topological order* — a way you can visit the nodes so that when you see a node, you've already seen every node that has an edge to it.

One can find a topological order for a DAG in linear O(V+E) time.

Given the topological order, shortest path can easily be computed in O(V+E) time.

# Overview of Quiz 0

I am given two coins, which I will flip simultaneously.

Let X = H be the event that the first coin comes up heads and X = T be the event that the first coin comes up tails. Let Y = H and Y = T be defined similarly for the second coin.

If you know that Pr(X = H,Y = H) = 0.25 and Pr(X = T, Y = H) = 0.5 and Pr(X=H, Y =T)=0.05.

What is Pr(X=T, Y =T)?

| C1\C2 | H | T |
|---|---|---|
| H | 0.25 | 0.05 |
| T | 0.5 | ? |

# Overview of Quiz 0

I am given two coins, which I will flip simultaneously.

Let X = H be the event that the first coin comes up heads and X = T be the event that the first coin comes up tails. Let Y = H and Y = T be defined similarly for the second coin.

If you know that Pr(X = H,Y = H) = 0.25 and Pr(X = T, Y = H) = 0.5 and Pr(X=H, Y =T)=0.05.

What is Pr(X=T, Y =T)?

**Prob of all possible events must sum to 1 (unitary)**

| C1\C2 | H | T |
|---|---|---|
| H | 0.25 | 0.05 |
| T | 0.5 | 0.2 |

# Overview of Quiz 0

I am given two coins, which I will flip simultaneously.

Let X = H be the event that the first coin comes up heads and X = T be the event that the first coin comes up tails. Let Y = H and Y = T be defined similarly for the second coin.

If you know that Pr(X = H,Y = H) = 0.25 and Pr(X = T, Y = H) = 0.5 and Pr(X=H, Y =T)=0.05.

What is Pr(X = T)?

| C1\C2 | H | T |
|---|---|---|
| H | 0.25 | 0.05 |
| T | 0.5 | 0.2 |

# Overview of Quiz 0

I am given two coins, which I will flip simultaneously.

Let X = H be the event that the first coin comes up heads and X = T be the event that the first coin comes up tails. Let Y = H and Y = T be defined similarly for the second coin.

If you know that Pr(X = H,Y = H) = 0.25 and Pr(X = T, Y = H) = 0.5 and Pr(X=H, Y =T)=0.05.

What is Pr(X = T)?

The marginal prob. is the sum over all joint states where X has this value.

| C1\C2 | H | T | |
|---|---|---|---|
| H | 0.25 | 0.05 | 0.3 |
| T | 0.5 | 0.2 | 0.7 |

# Overview of Quiz 0

I am given two coins, which I will flip simultaneously.

Let X = H be the event that the first coin comes up heads and X = T be the event that the first coin comes up tails. Let Y = H and Y = T be defined similarly for the second coin.

If you know that Pr(X = H,Y = H) = 0.25 and Pr(X = T, Y = H) = 0.5 and Pr(X=H, Y =T)=0.05.

Are the two coin flips *statistically* independent (say why or why not)?

# Overview of Quiz 0

I am given two coins, which I will flip simultaneously.

Let X = H be the event that the first coin comes up heads and X = T be the event that the first coin comes up tails. Let Y = H and Y = T be defined similarly for the second coin.

If you know that Pr(X = H,Y = H) = 0.25 and Pr(X = T, Y = H) = 0.5 and Pr(X=H, Y =T)=0.05.

Are the two coin flips *statistically* independent (say why or why not)?

**No:** X and Y are *statistically* independent iff P(X=x,Y=y) = P(X=x)*P(Y=y)

# Overview of Quiz 0

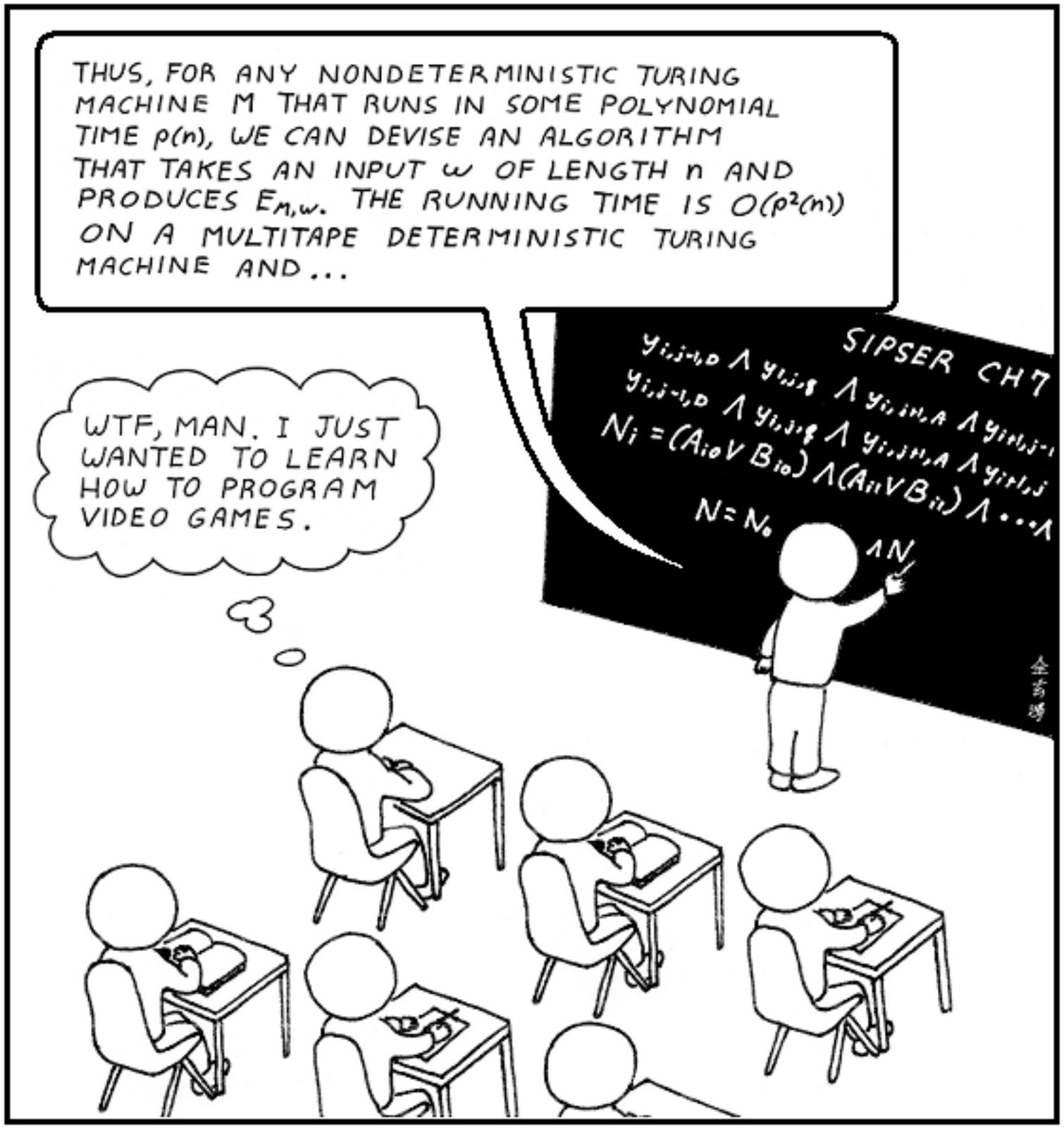
The program grep allows one to quickly find a target pattern in a file. In practice grep works quickly by using a combination of a well-engineered implementation, and an efficient algorithm for pattern search. Assume that you are given a large string T representing the file and a string P representing the pattern the user wishes to find.

What algorithm might you use to search for the pattern P in T ?

What is the running time of your algorithm in terms of |T | and |P | (the lengths of strings T and P, respectively)?

Many possible answers, but an *efficient* algorithm will be linear in the length of the input O(|T|+|P|), we will learn about such algorithms next week.

# What is Computer Science?




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* …

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

# What is Computer Science?

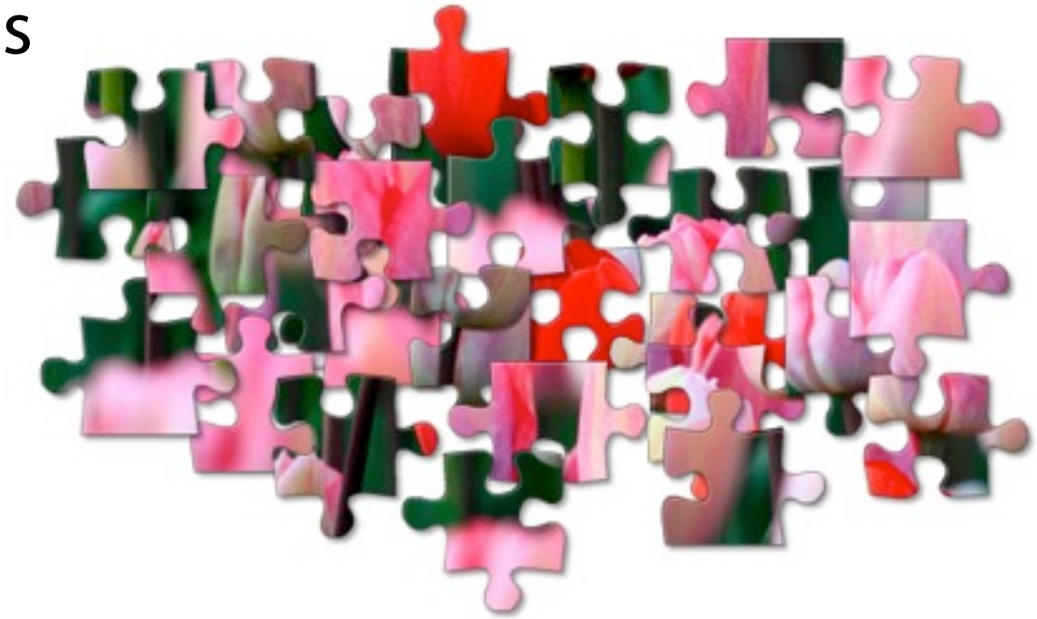
Concerned with the development of provably **correct** and **efficient** computational procedures (algorithms & data structures) to answer **well-specified** problems.

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!

# 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)

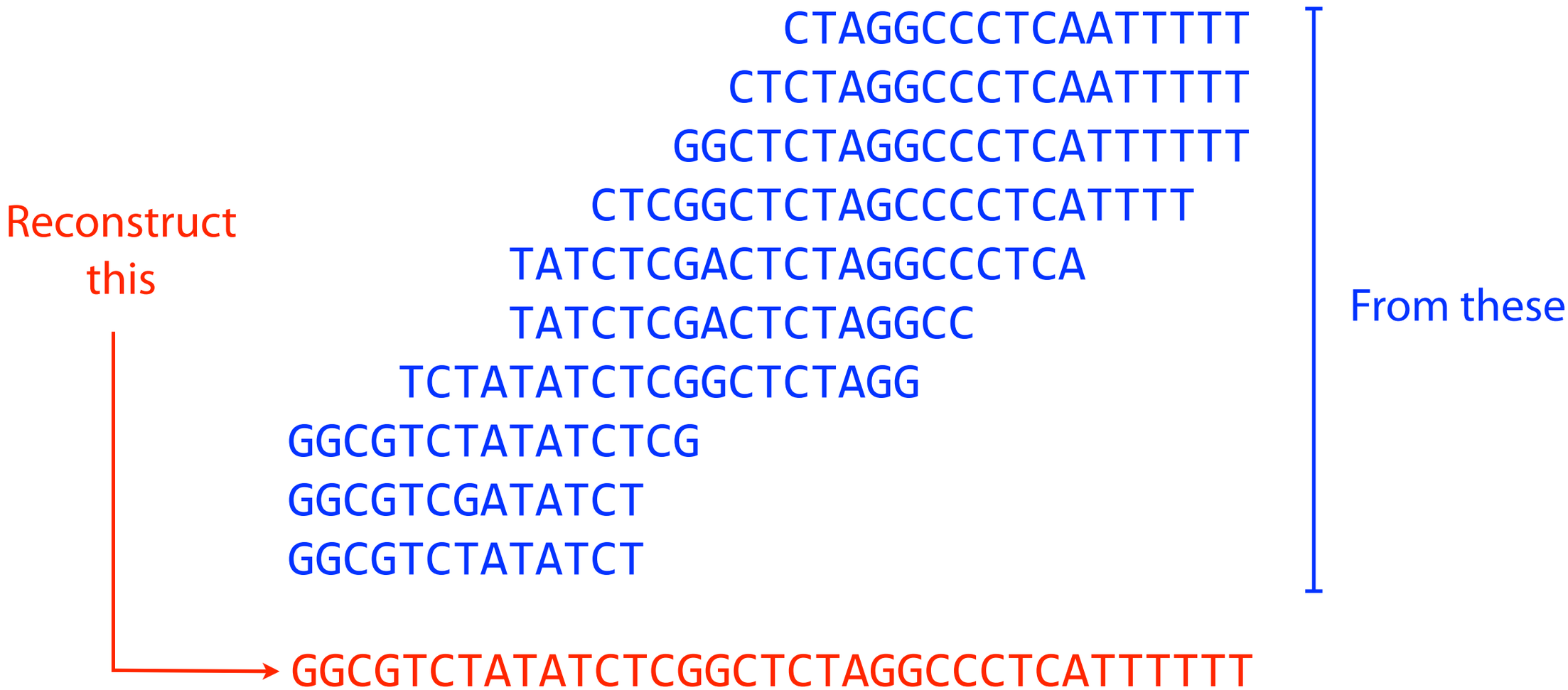
Input: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

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...



# 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

From these

CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

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

# 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: BAAAABBBAABAABBBBBBAAABAB

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

Can we solve it?

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

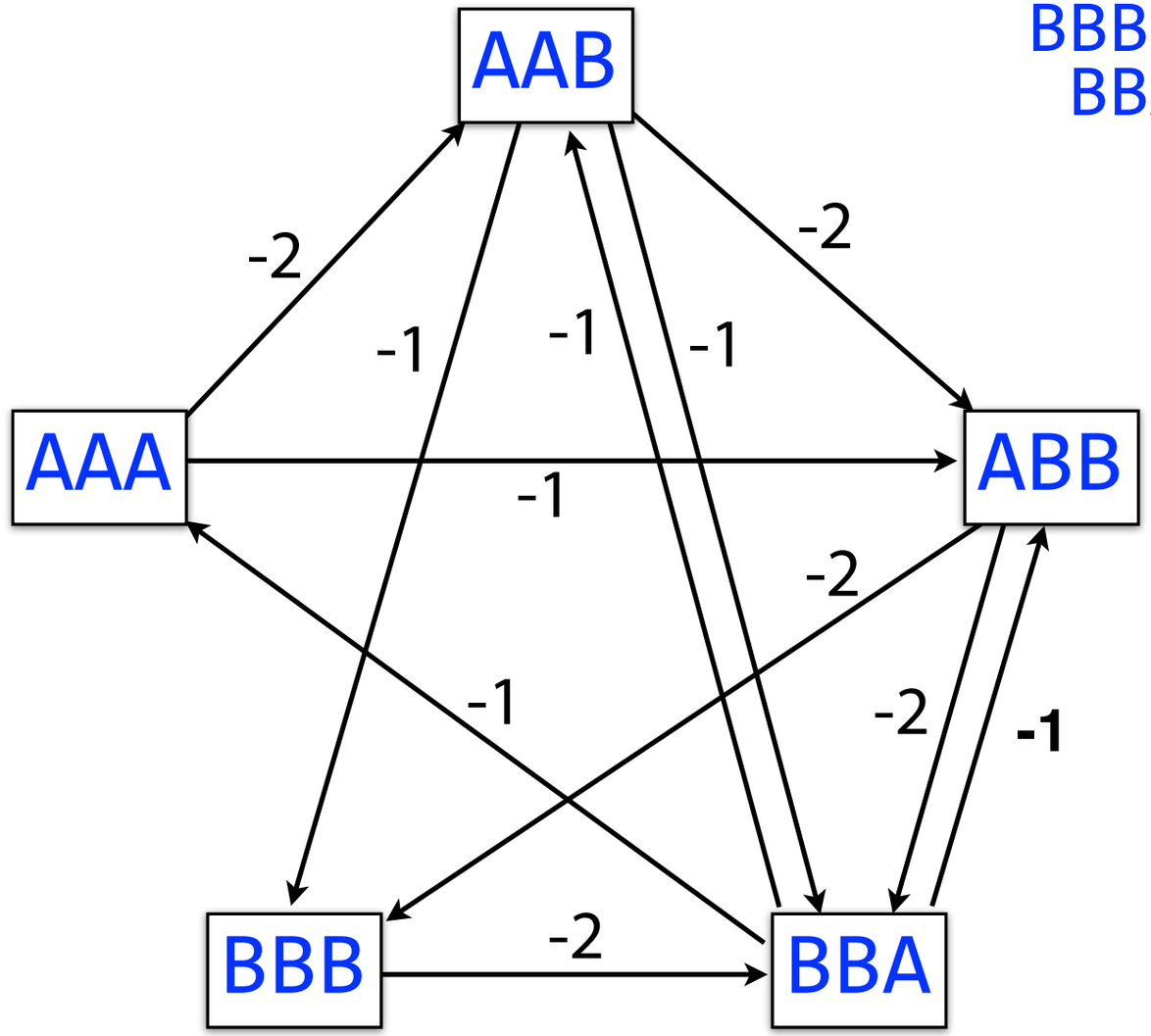
SCS corresponds to a path that visits every node once, minimizing total cost along path

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




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

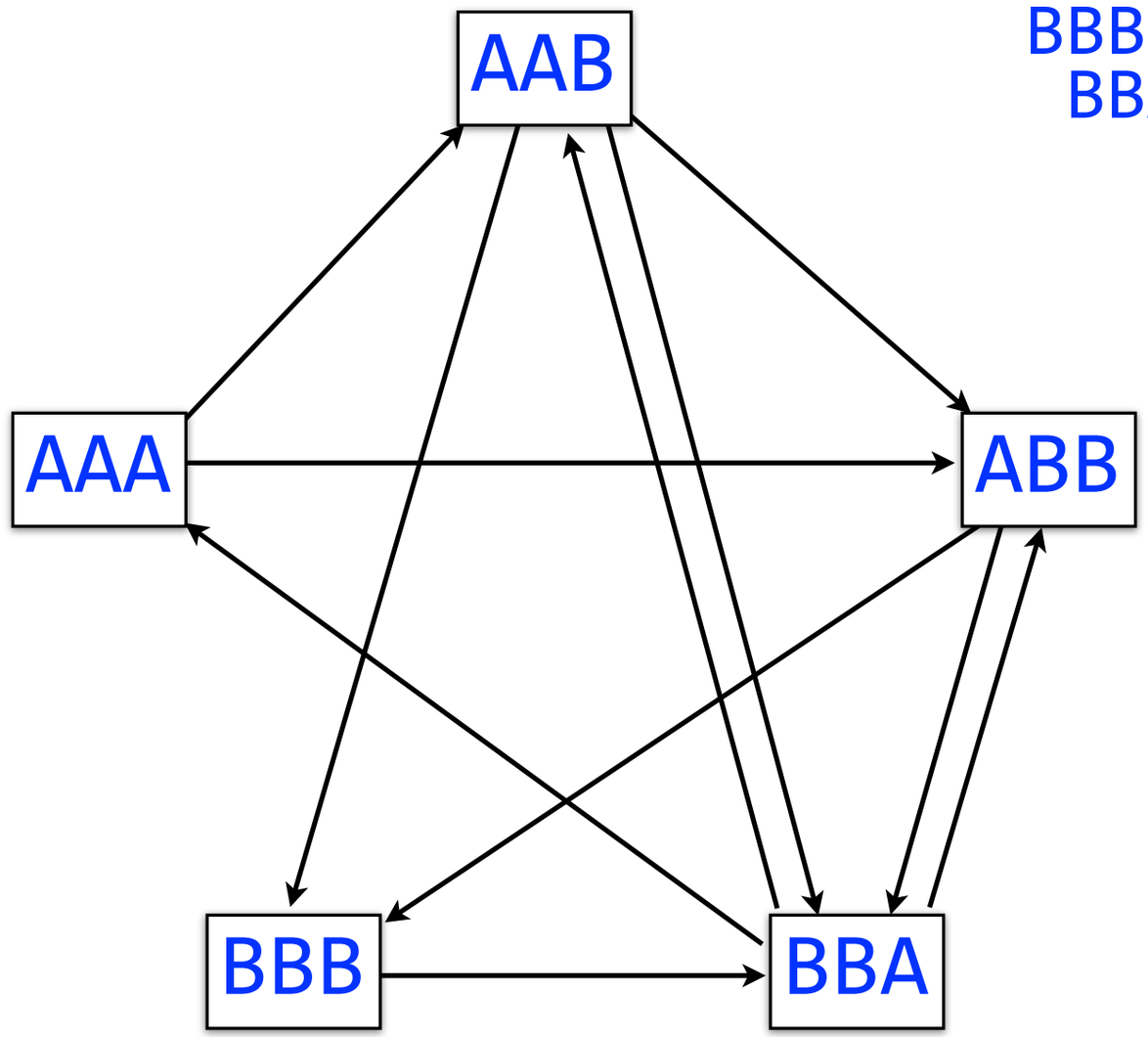
That's the *Hamiltonian Path* problem: NP-complete

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

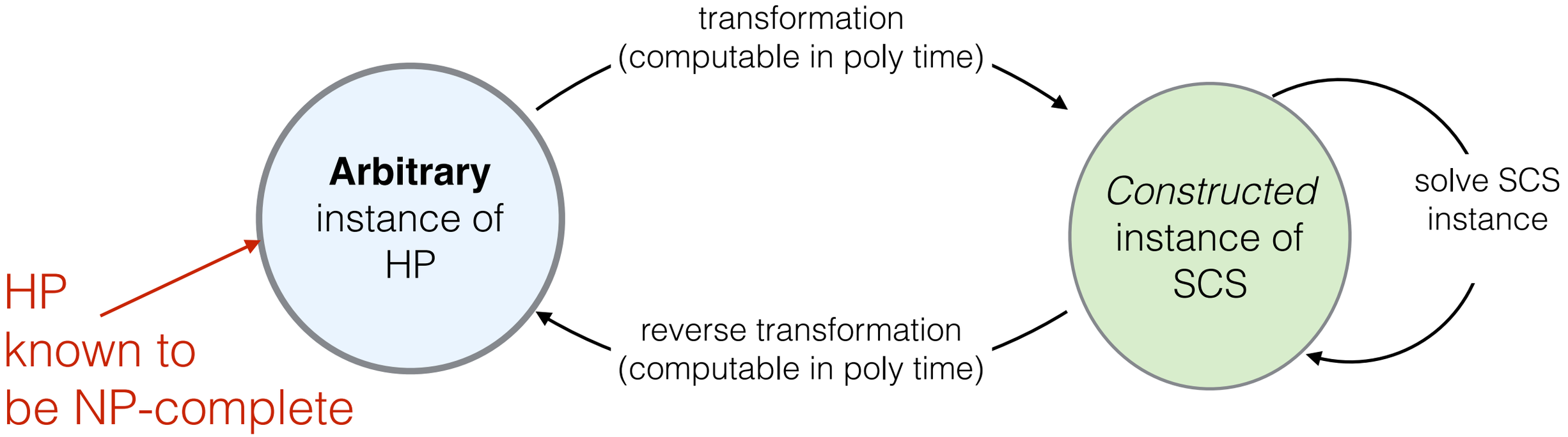
For refreshers on Traveling Salesman, Hamiltonian Path, NP-hardness and NP-completeness, see Chapters 34 and 35 of "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein, or Chapters 8 and 9 of "Algorithms" by Dasgupta, Papadimitriou and Vazirani (free online: http://www.cs.berkeley.edu/~vazirani/algorithms)

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



# 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.

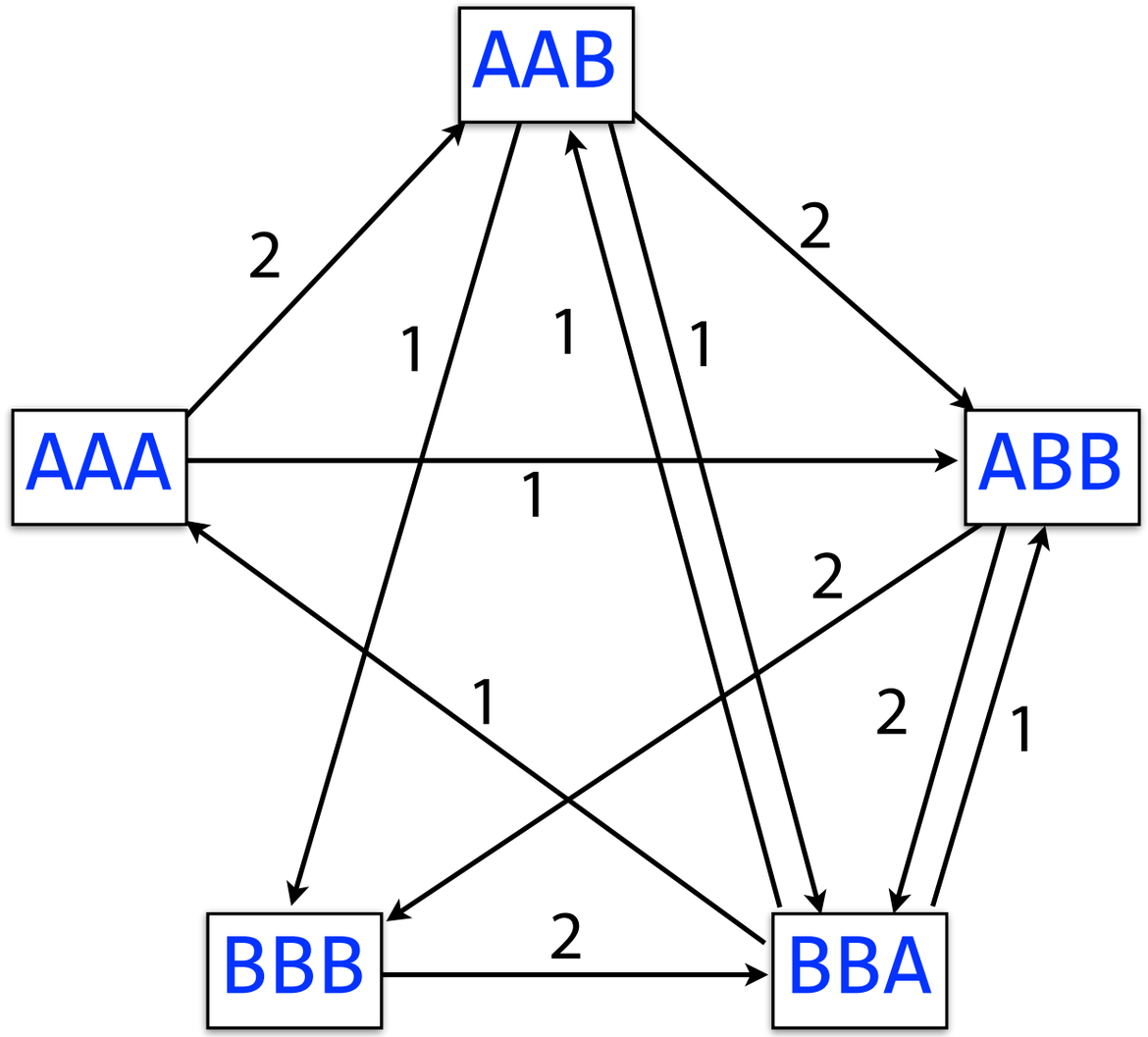
Greedy conjecture factor of 2-OPT *is* the worst case

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.

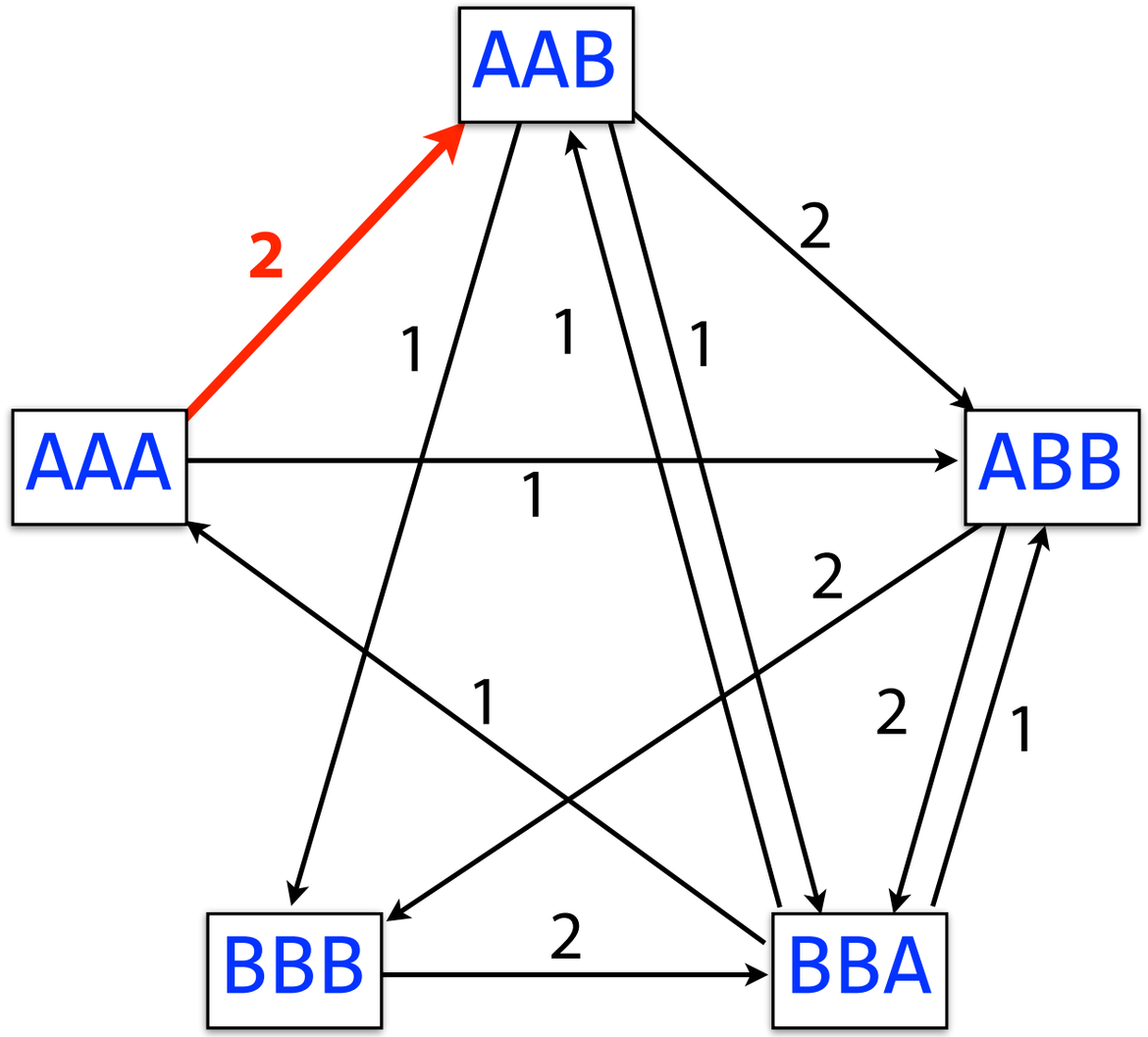
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

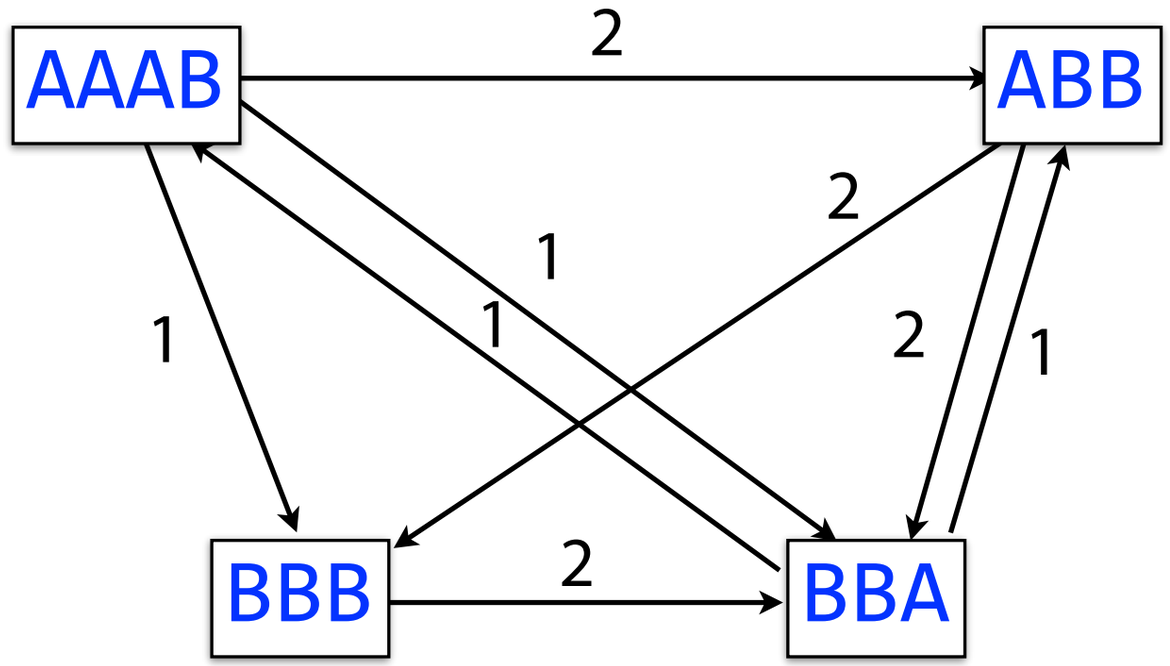
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

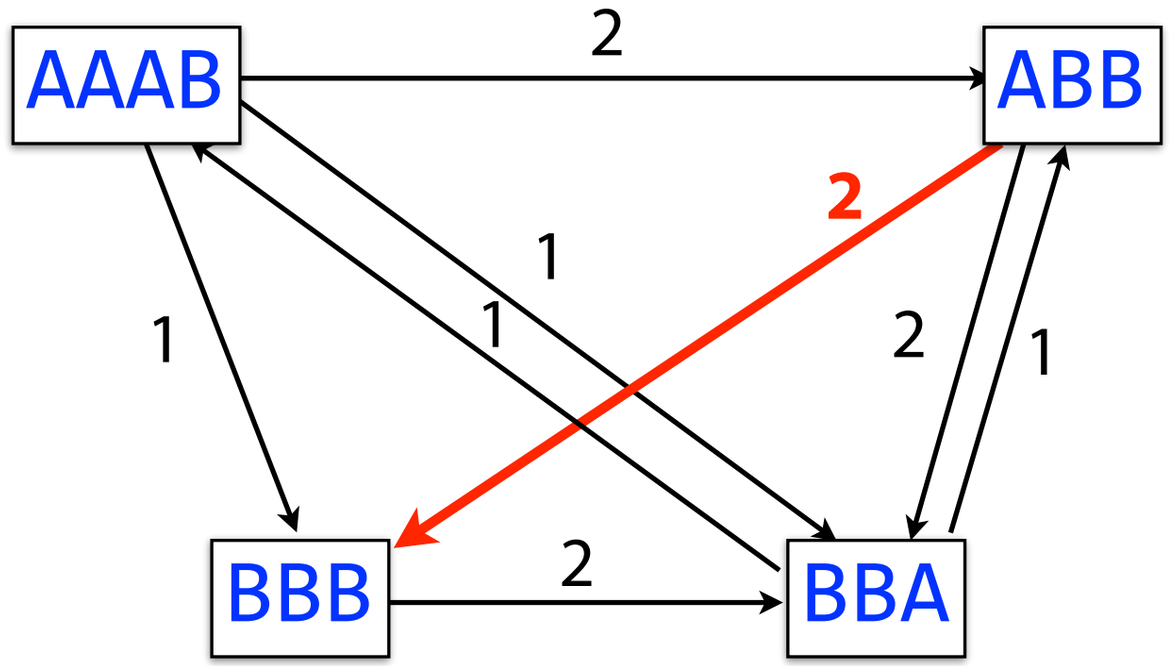
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

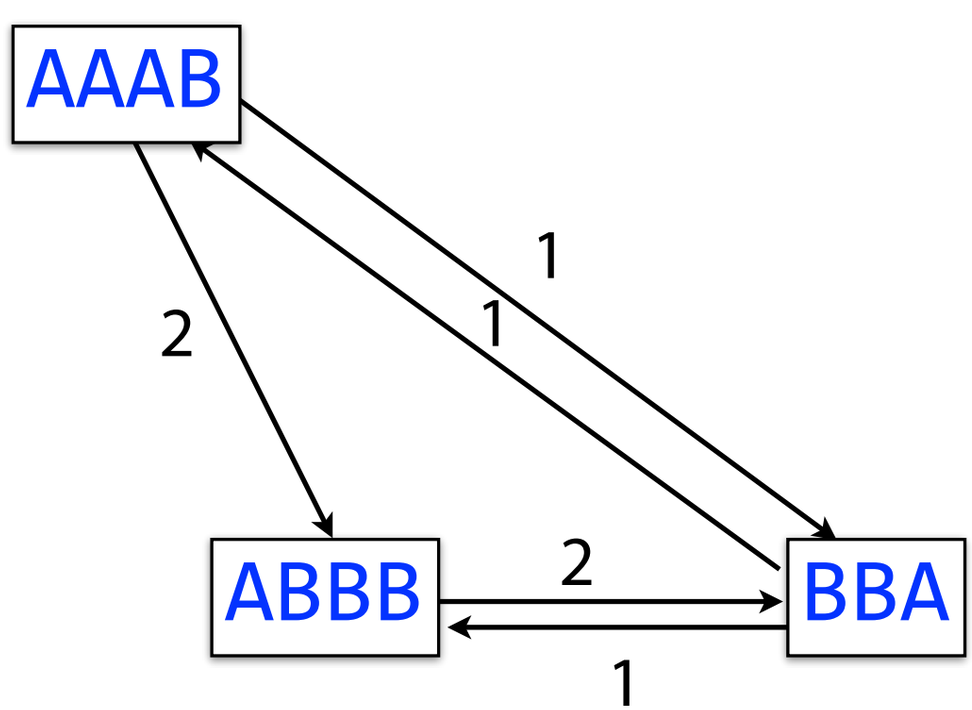
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

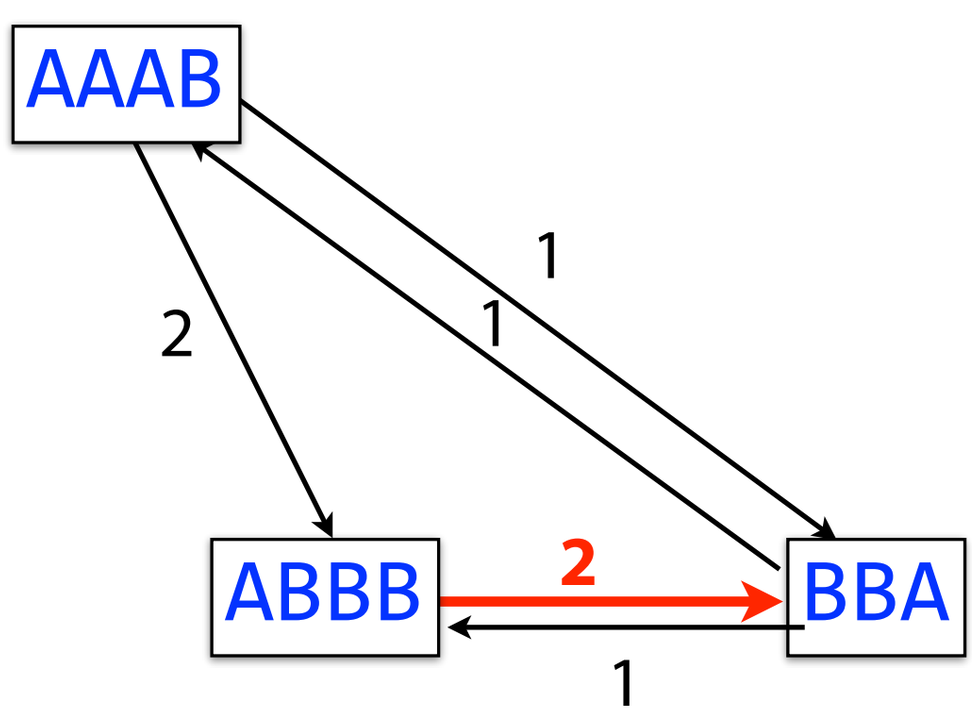
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

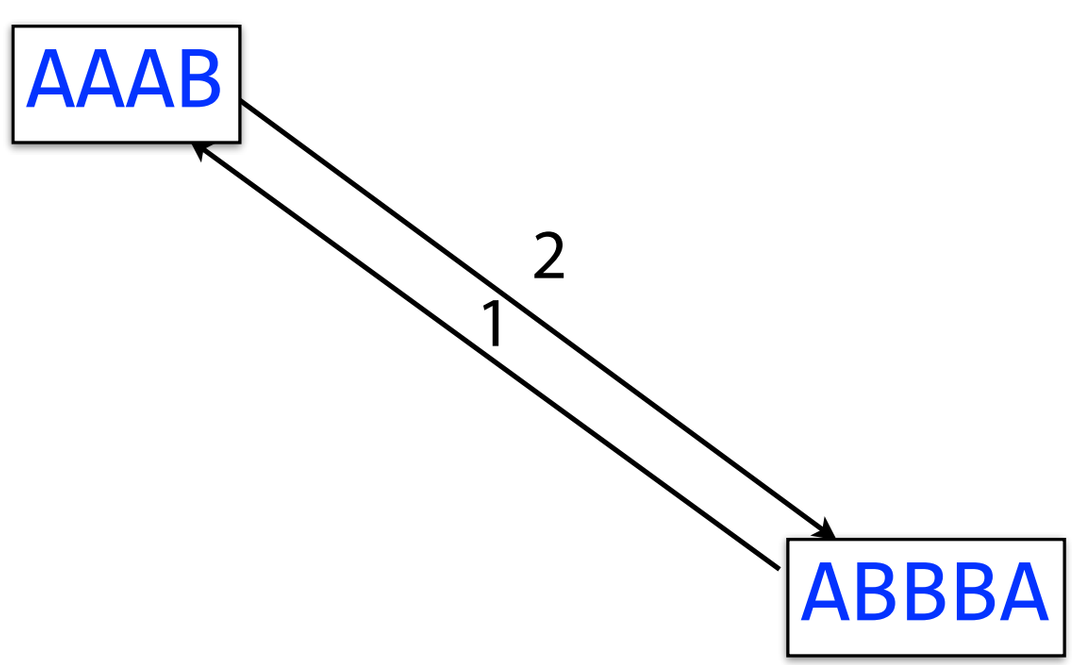
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

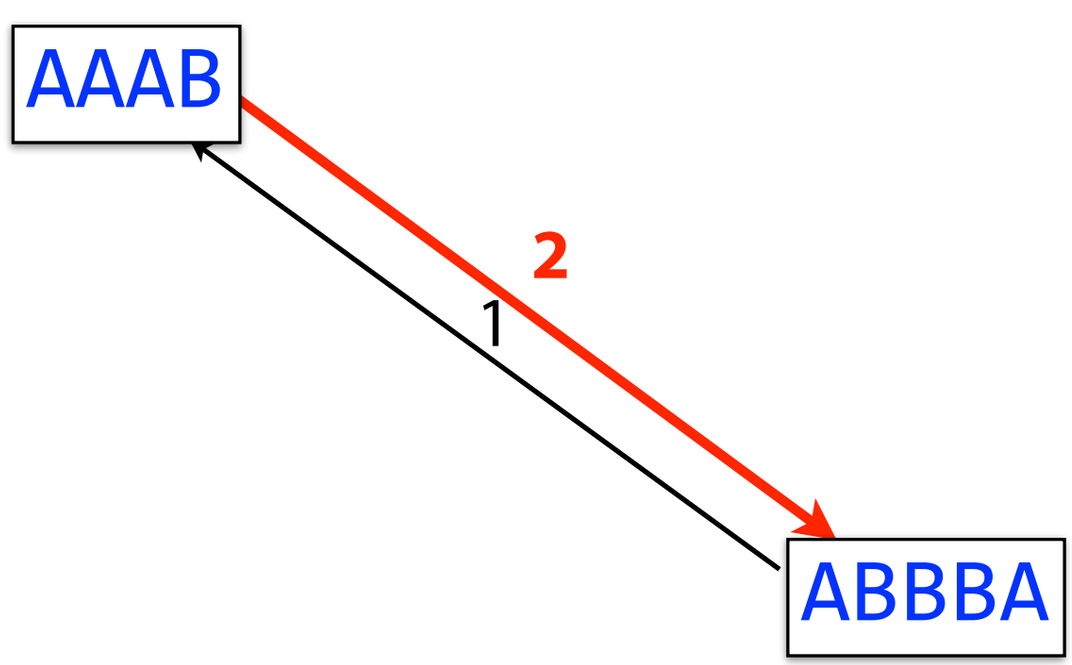
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

# 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

AAABBBA ⟵ superstring, length=7

Slide courtesy of Ben Langmead

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

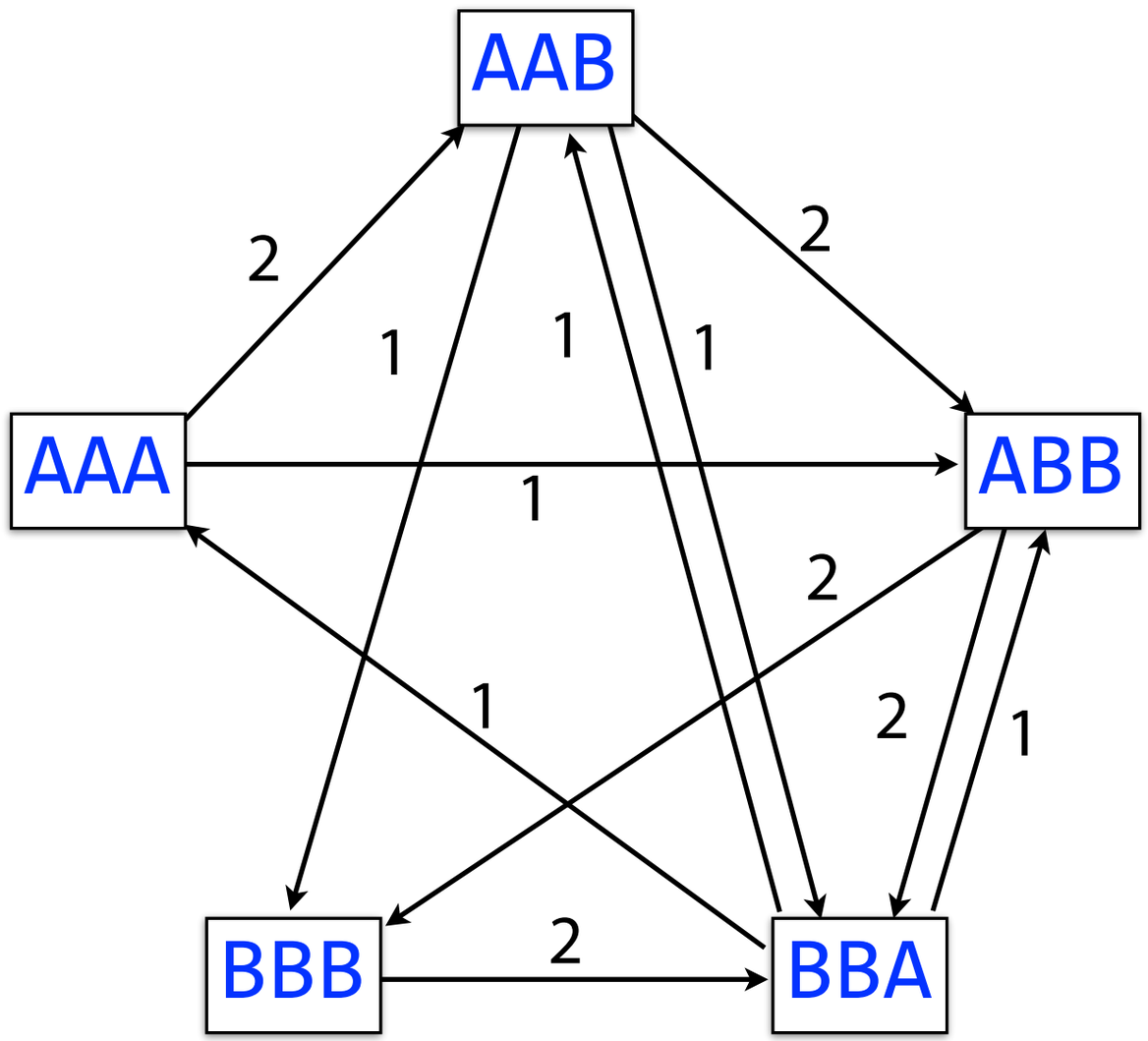
# Shortest common superstring: greedy

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




Slide courtesy of Ben Langmead

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

# Shortest common superstring: greedy

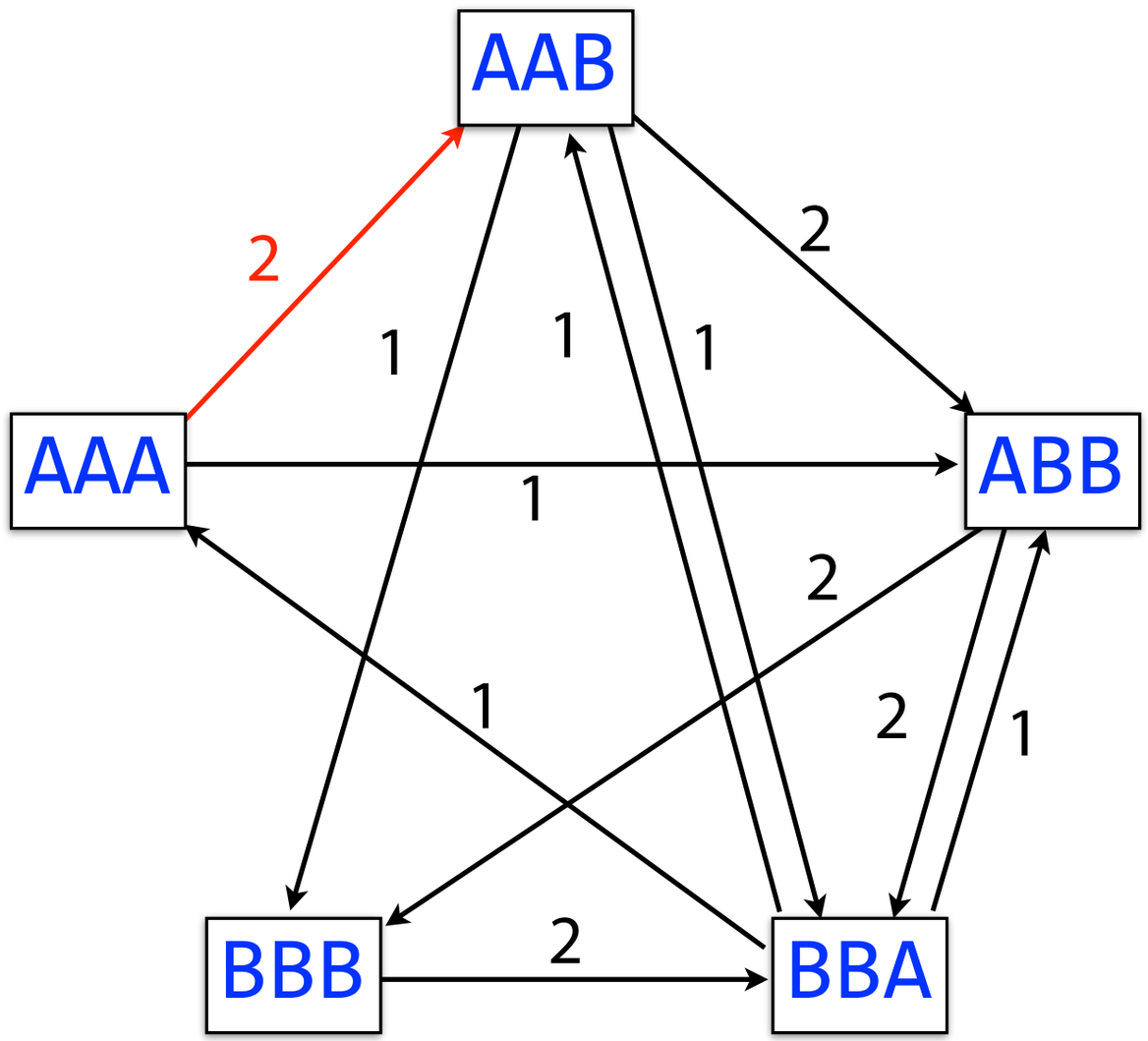
Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. $l$ = minimum overlap.

Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA




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.

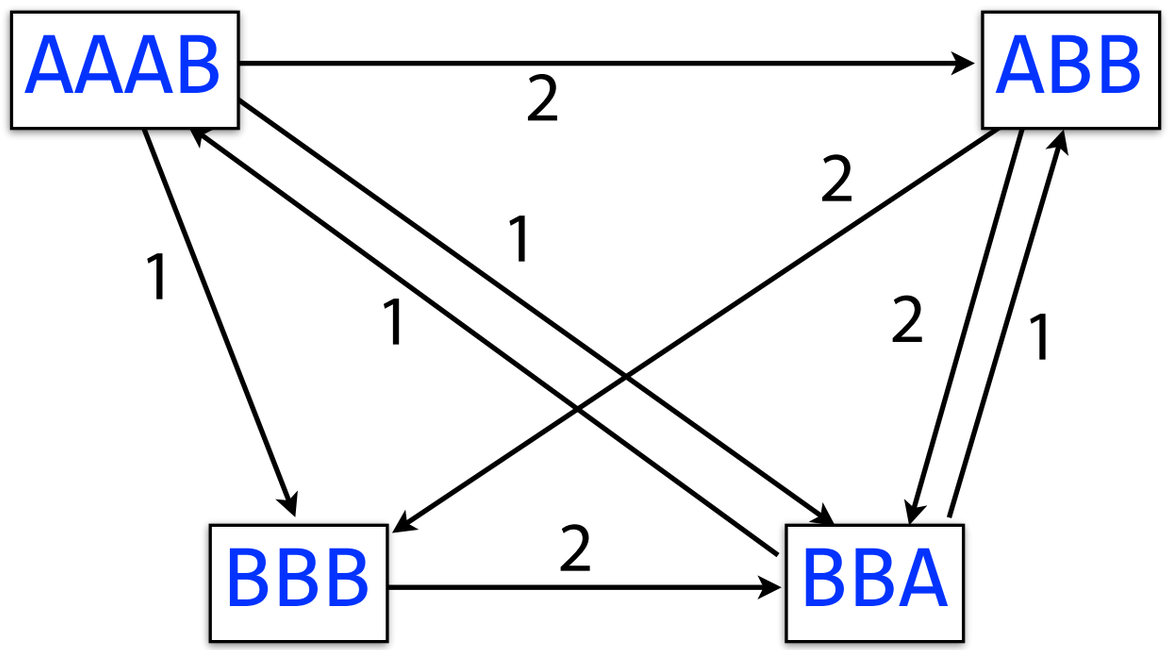
Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

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 there's 1 string left. $l$ = minimum overlap.

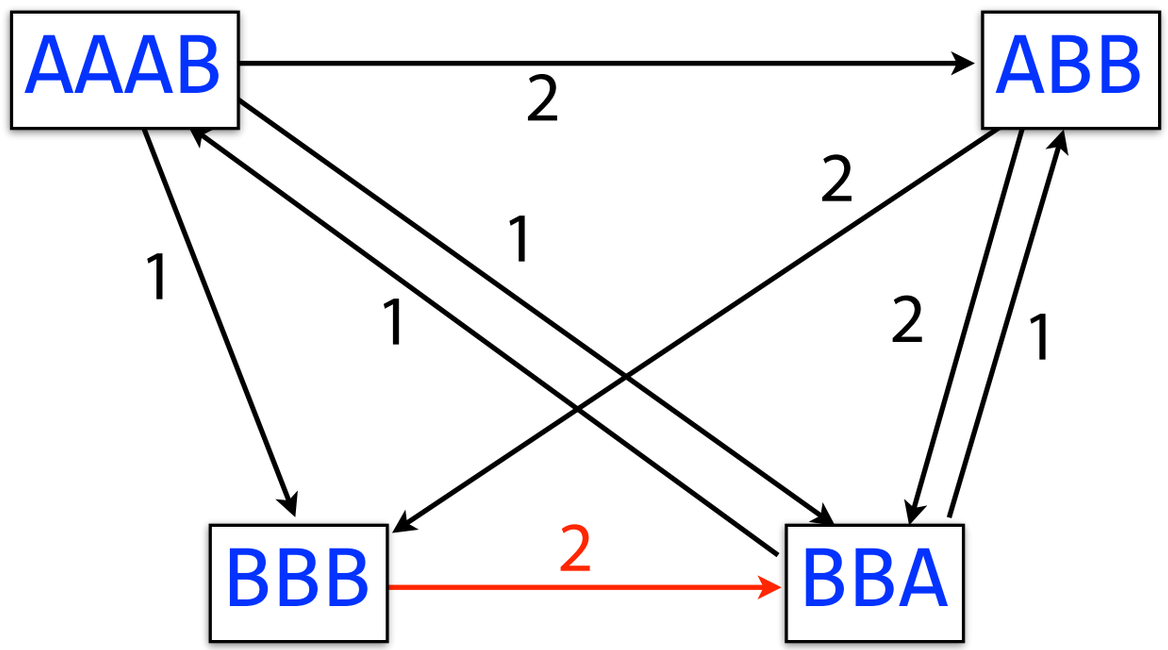
Algorithm in action ($l = 1$):

Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA




Slide courtesy of Ben Langmead

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 ($l = 1$):

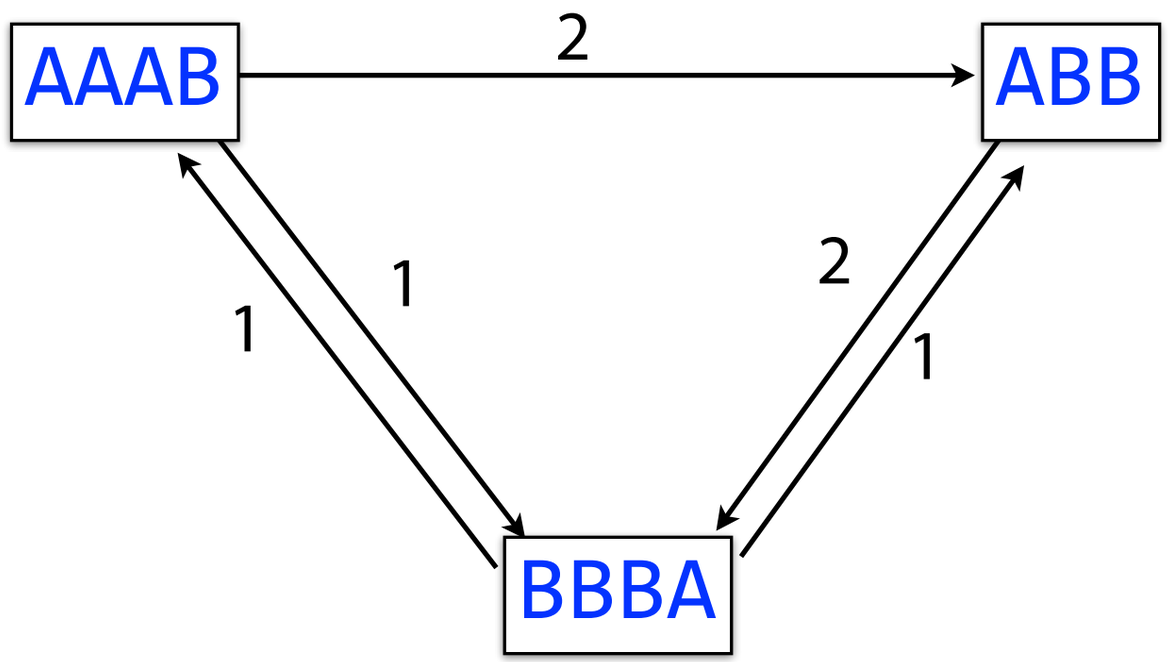
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

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 there's 1 string left. $l$ = minimum overlap.

Algorithm in action ($l = 1$):

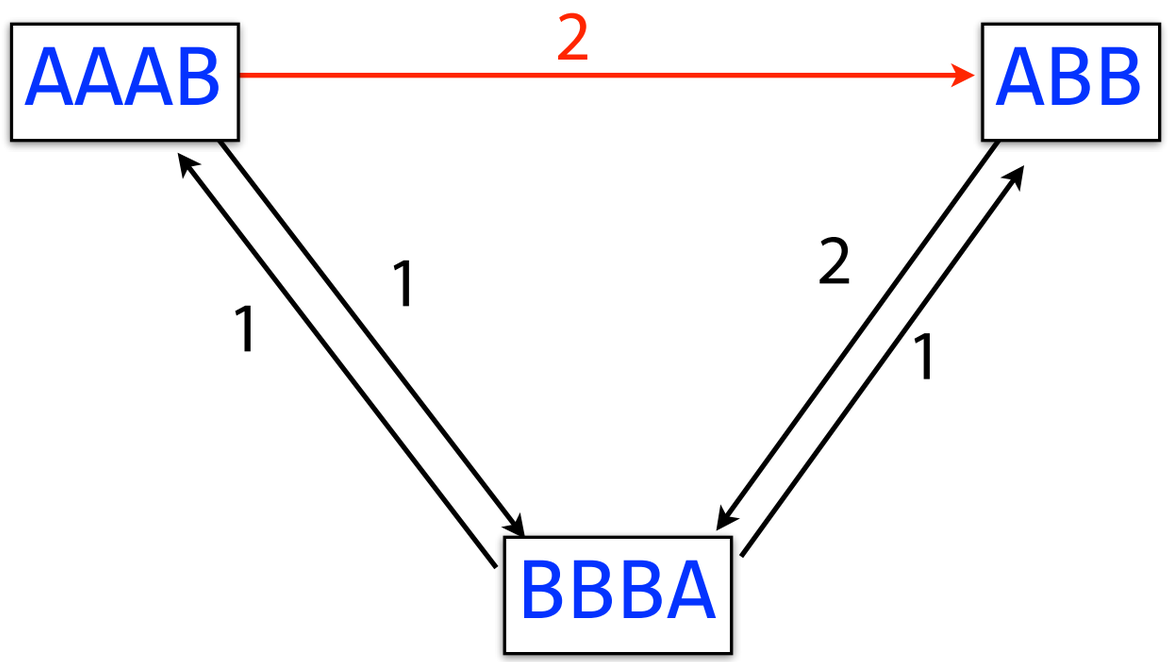
Input strings

AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

AAAB ABB BBB BBA

AAAB BBBA ABB




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

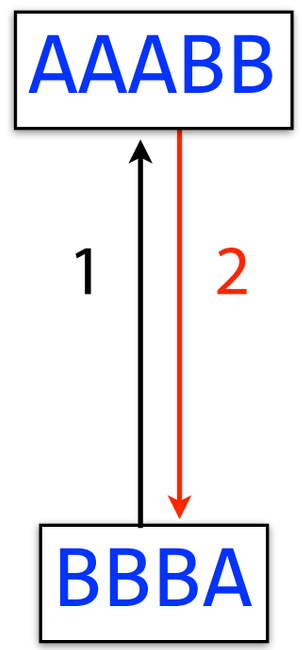
AAA AAB ABB BBB BBA

AAA AAB ABB BBB BBA

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 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

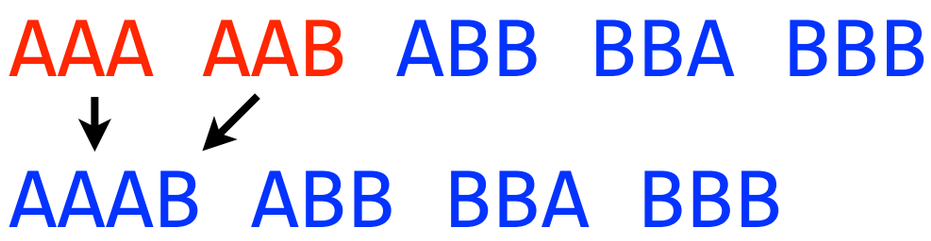
AAABBBA

That's the SCS

Slide courtesy of Ben Langmead

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

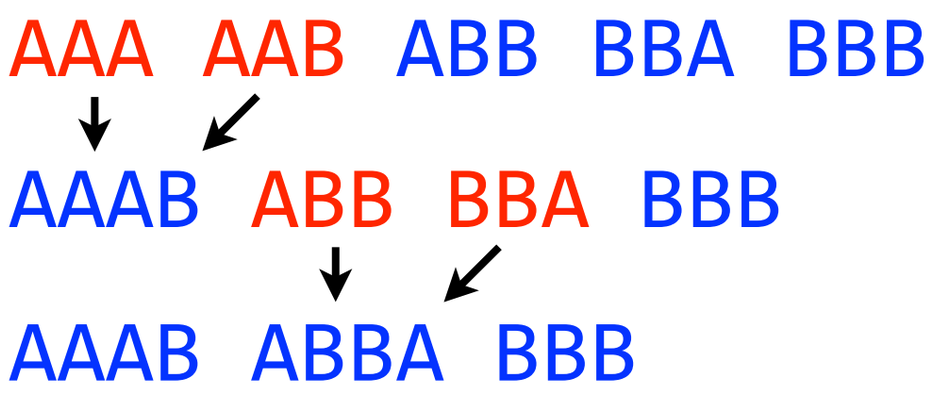
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

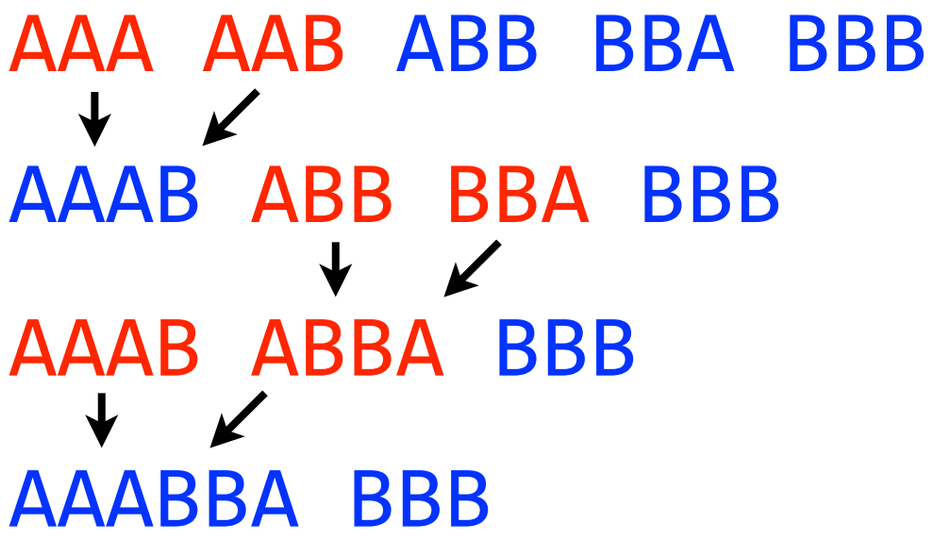
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

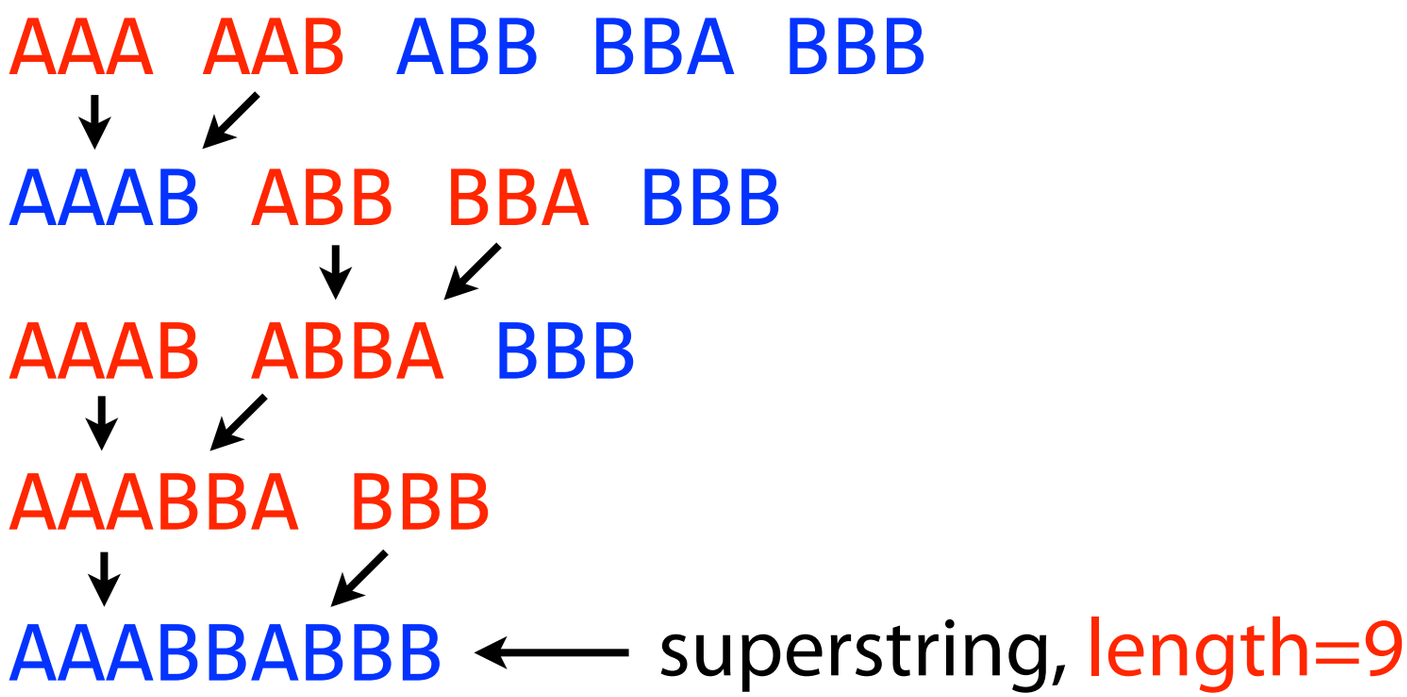
# Greedy shortest common superstring




Slide courtesy of Ben Langmead

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

# Greedy shortest common superstring

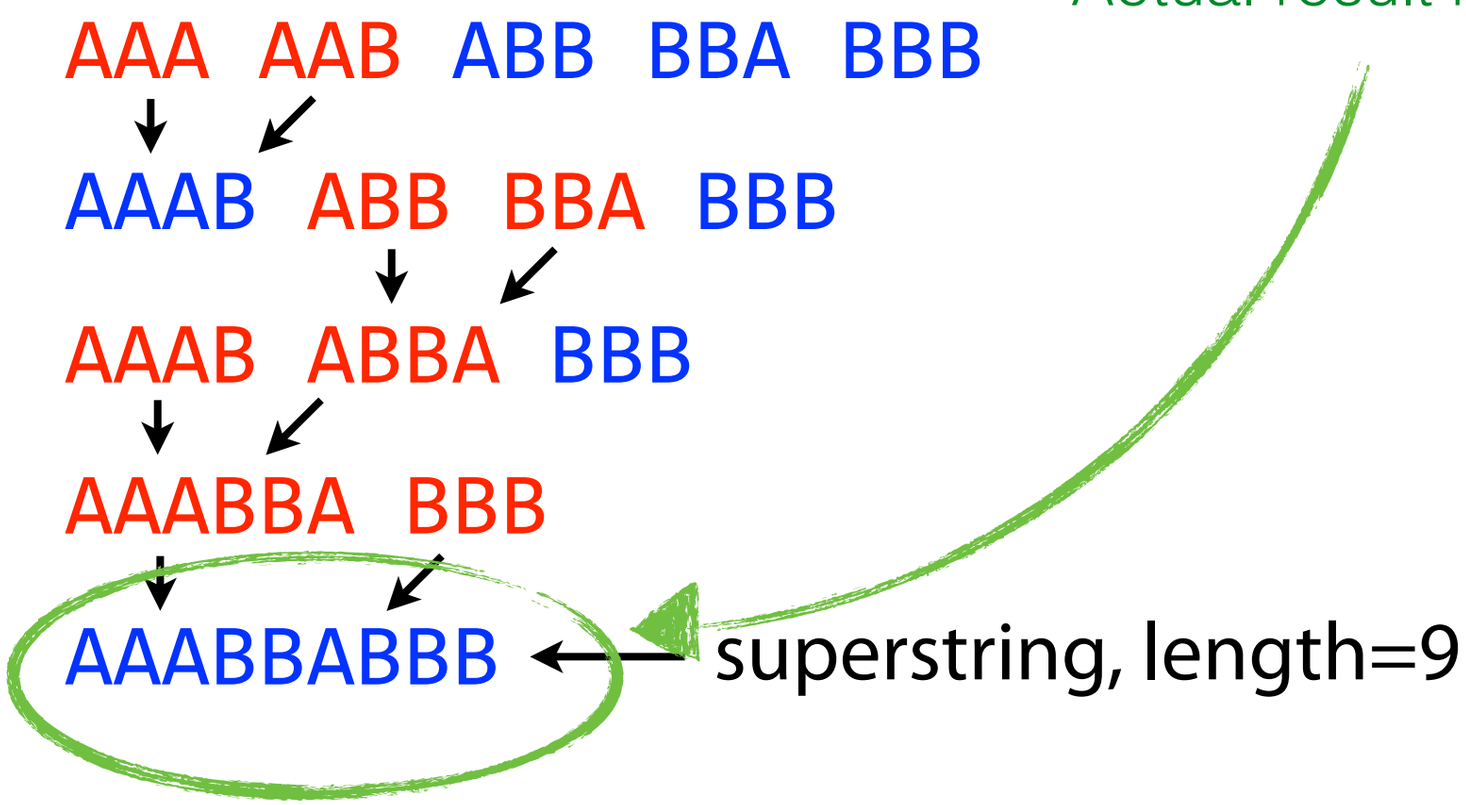



Slide courtesy of Ben Langmead

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

# Greedy shortest common superstring

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




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

# 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.