Algorithms in Computational Biology

Shaojie Zhang
Computer Science

Contact Details:

Shaojie Zhang:

- Office: HEC 311
- Telephone: 407-823-6095
- Email: shzhang@cs.ucf.edu (No Webcourse Emails!!!)

Course home page: Webcourses@UCF

Misc.

- Time and Location
  - HEC-104
  - T/H 3:00-4:15 pm
  - Office hours: 2:00-2:50 pm

- Textbook (references):
  - Pavel Pevzner: Computational molecular biology: an algorithmic approach , MIT Press, 2000
  - Dan Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge university press, 1997
  - Phillip Compeau and Pavel Pevzner, Bioinformatics Algorithms: An Active Learning Approach, Volume 1 and Volume 2. Active Learning Publishers, 2015

Assessment:

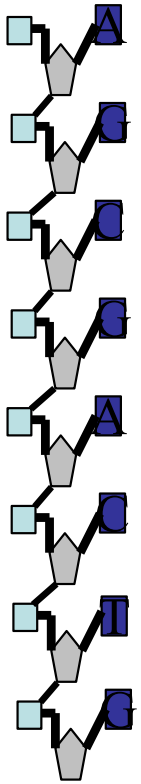
Element	Weight
4 Assignments: only accept answers in tex file (**Latex)**	40%
Midterm	20%
Final Exam	20%
Algorithm Presentation	15%
Class Participation	5%
TOTAL	100%

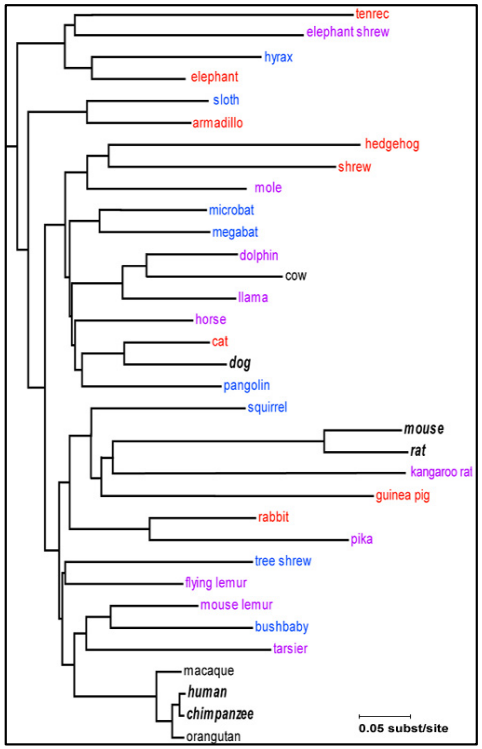
Getting started

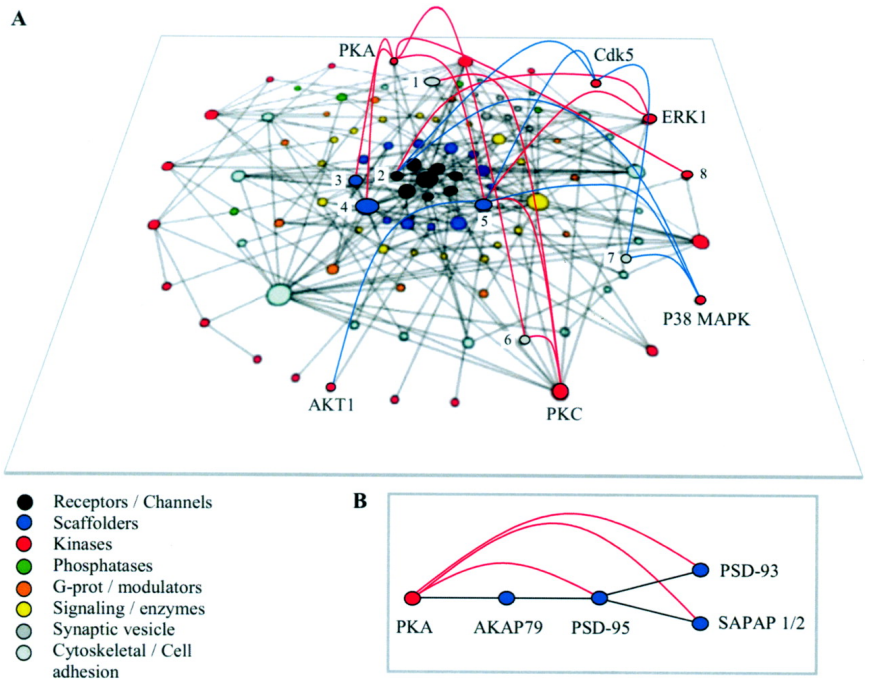
- Bioinformatics → solving biological problems by computers;

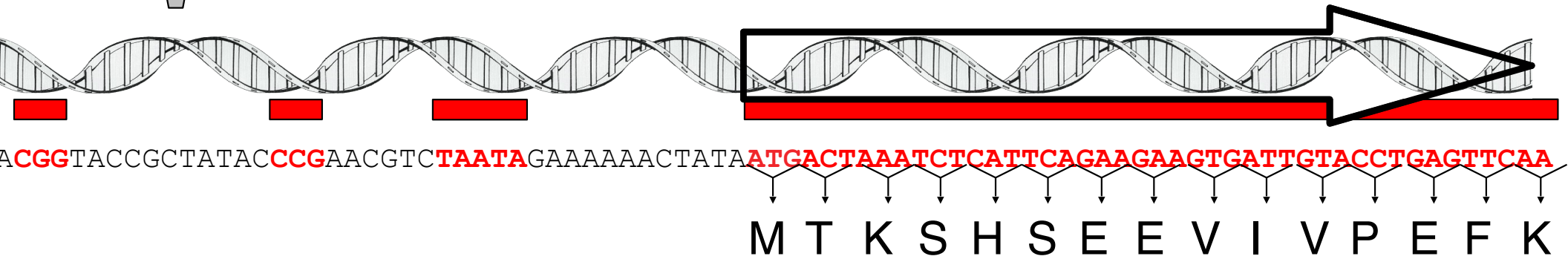
- Why computers?
  - Large amount of data
  - Complex computation

What kind of biological data?

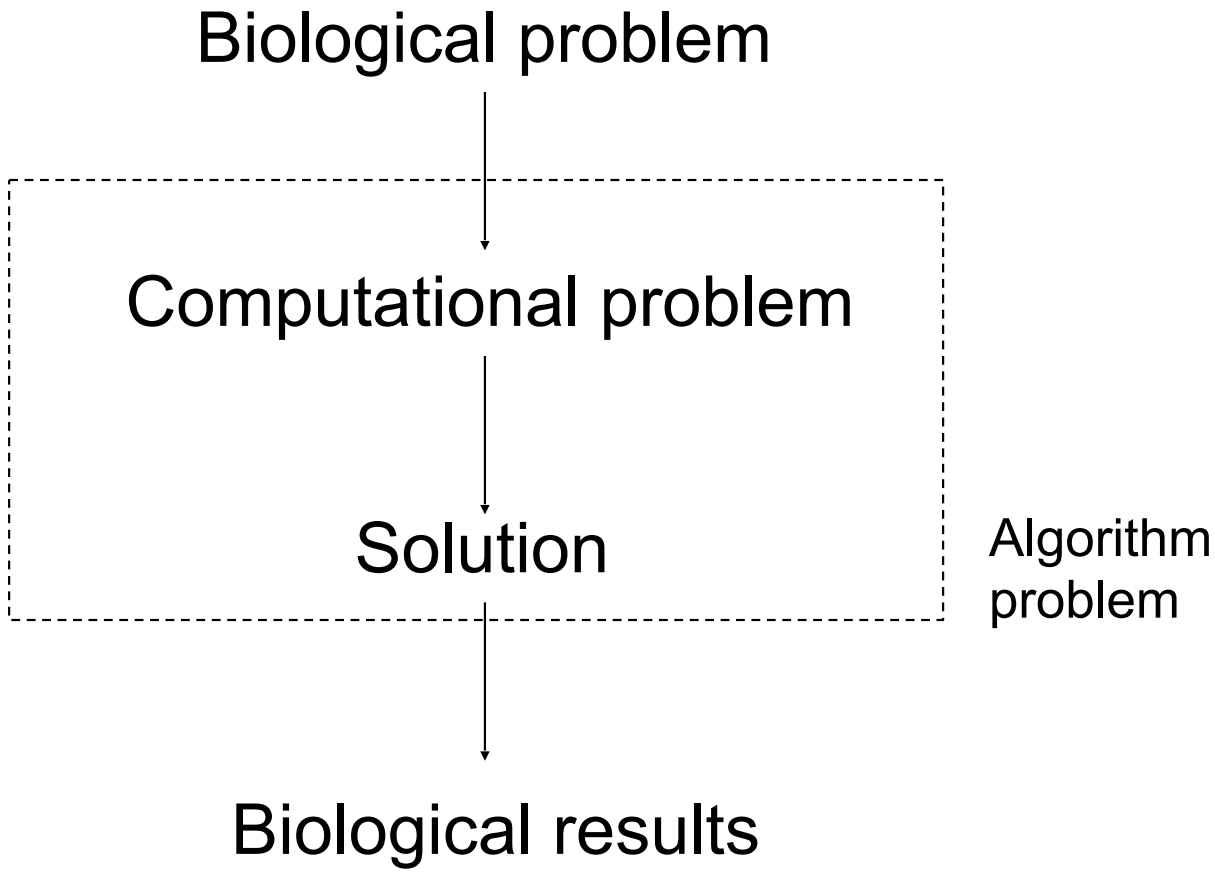




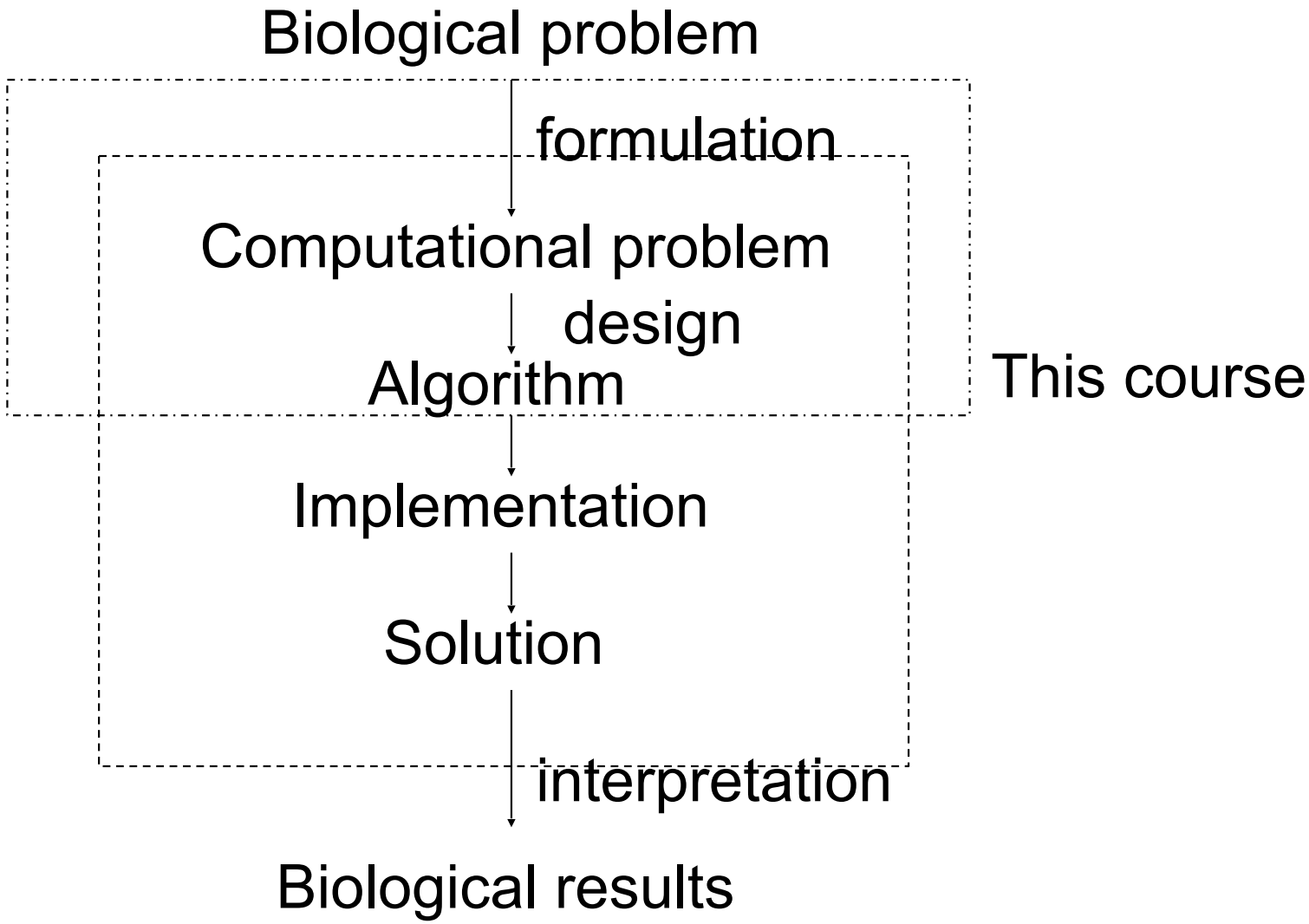




A general solution pipeline for bioinformatics problems



A general solution pipeline for bioinformatics problems



What is a (computer) algorithm?

- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required **output** for any legitimate **input** in a **finite** amount of time.

Features of Algorithm

- finiteness (otherwise: computational method)
  - termination
- definiteness
  - precise definition of each step
- input (zero or more)
- output (one or more)
- effectiveness

- Its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper. -- Knuth

Example of computational problem: sorting

- Statement of problem:
  - *Input:* A sequence of *n* numbers $<a_1, a_2, \ldots, a_n>$
  - *Output:* A reordering of the input sequence $<a'_1, a'_2, \ldots, a'_n>$ so that $a'_i \leq a'_j$, whenever $i < j$
- Instance:
  - Input <5, 3, 2, 8, 3>

Algorithms

- How to express algorithms
- How to design algorithms
- Proving correctness
- Efficiency
  - Theoretical analysis
  - Empirical analysis
- Optimality

How to express algorithms

- Idea
- Flowchart
- **Pseudocode**
- Program – implementation

How do we design an algorithm?

One step at a time – Iterative algorithms

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
codeC
<postCond>
```

Loop Invariants

Breaking problems down: recursive algorithms

Recursion: a subroutine which calls itself, with different parameters.

Compute n! = factorial(n)

factorial(n) = n× (n-1) ×…×2×1

= n×factorial(n-1)

Suppose routine factorial(p) can find factorial of p for all p < m. Then factorial(m+1) can be solved as follows:

factorial(m+1) =(m+1) ×factorial(m)

```
Factorial(m)
{
    If m = 1, Factorial(m) = 1;
    else  Factorial(m) = m×Factorial(m-1)
}
```

Rules:

Have a base case for which the subroutine need not call itself.
For general case, the subroutine does some operations, calls itself, gets result and does some operations with the result.

To get result, subroutine should progressively move towards the base case.

Recursion vs. iteration

```
Factorial_It(m)
{
prod = 1
For j=1  to m
  prod →prod *j
}
```

In general, iteration is more efficient than recursion because of maintenance of state information.

Towers of Hanoi

Source peg, Destination peg, Auxiliary peg

k disks on the source peg. Need to move all k disks to the destination peg using the auxiliary peg, without ever keeping a bigger disk on the smaller disk.

Why can this be solved by recursion?

We know how to move 1 disk from source to destination.

For two disks, move the top one to the auxiliary, bottom one to the destination, then first to the destination.

For three disks, move top two disks from source to auxiliary, using destination.

Then move the bottom one from the source to the destination.

Finally move the two disks from auxiliary to destination using source.

We know how to solve this for k=1

Suppose we know how to solve this for k-1 disks.

We will first move top k-1 disks from source to auxiliary, using destination.

Will move the bottom one from the source to the destination.

Will move the k-1 disks from auxiliary to destination using source.

```
TowerHanoi(k, source, auxiliary, destination)
  {
    If k=1 move disk from source to destination; (base case)
    else,
      {
          TowerHanoi(top k-1, source, destination, auxiliary);
          Move the kth disk from source to destination;
          Tower of Hanoi(k-1, auxiliary, source, destination);
      }
}
```

Get the biggest first – greedy approaches

The greedy algorithm used to give change.
Amount owed: 41 cents.

Subtract Quarter
41 - 25 = 16

Subtract Dime
16 - 10 = 6

Subtract Nickel
6 - 5 = 1

Subtract Penny
1 - 1 = 0

Coin change problem

Input: a number P (price)

Output: a way of paying P using bills ($5, $1) or coins (1, 5, 10, 25 cents)

Objective: minimize the number of coins used

8.37 = 5 + 1 + 1 + 1 + 0.25 + 0.10 + 0.01 + 0.01

8 bills/coins used

Coin change problem

Greedy algorithm: for simplicity we assume that the coin values are: 10,5,1 (cents)

```
GreedyCoinChange(N) {
   if N ≥ 10 then
      output(10), GreedyCoinChange(N-10)
   else if N ≥ 5 then
      output(5), GreedyCoinChange(N-5)
   else if N ≥ 1 then
      output(1), GreedyCoinChange(N-1)
}
```

Exact algorithm vs. heuristic algorithm

- Exact algorithm: guaranteed to report the correct (optimal) result
  - May be very expensive for some (intractable) problems
- Otherwise, heuristic algorithm
  - Many randomized algorithms
  - Judging by performance on real problems

Coin change problem

```
GreedyCoinChange(N) {
   if N ≥ 10 then
      output(10), GreedyCoinChange(N-10)
   else if N ≥ 5 then
      output(5), GreedyCoinChange(N-5)
   else if N ≥ 1 then
      output(1), GreedyCoinChange(N-1)
}
```

Theorem:
For coin values 10,5,1, the GreedyCoinChange algorithm outputs the minimal number of coins.

Coin change problem

Theorem: for coin values 10,5,1, the GreedyCoinChange algorithm outputs the minimal number of coins.

Proof:
Let N be the amount to be paid. Let the optimal solution be $P=A*10 + B*5 + C$. Clearly $B\leq 1$ (otherwise we can decrease B by 2 and increase A by 1, improving the solution). Similarly, $C\leq 4$.

Let the solution given by GreedyCoinChange be $P=a*10 + b*5 + c$. Clearly $b\leq 1$ (otherwise the algorithm would output 10 instead of 5). Similarly $c\leq 4$.

From $0\leq C\leq 4$ and $P=(2A+B)*5+C$ we have $C=P \bmod 5$.
Similarly $c=P \bmod 5$, and hence $c=C$. Let $Q=(P-C)/5$.

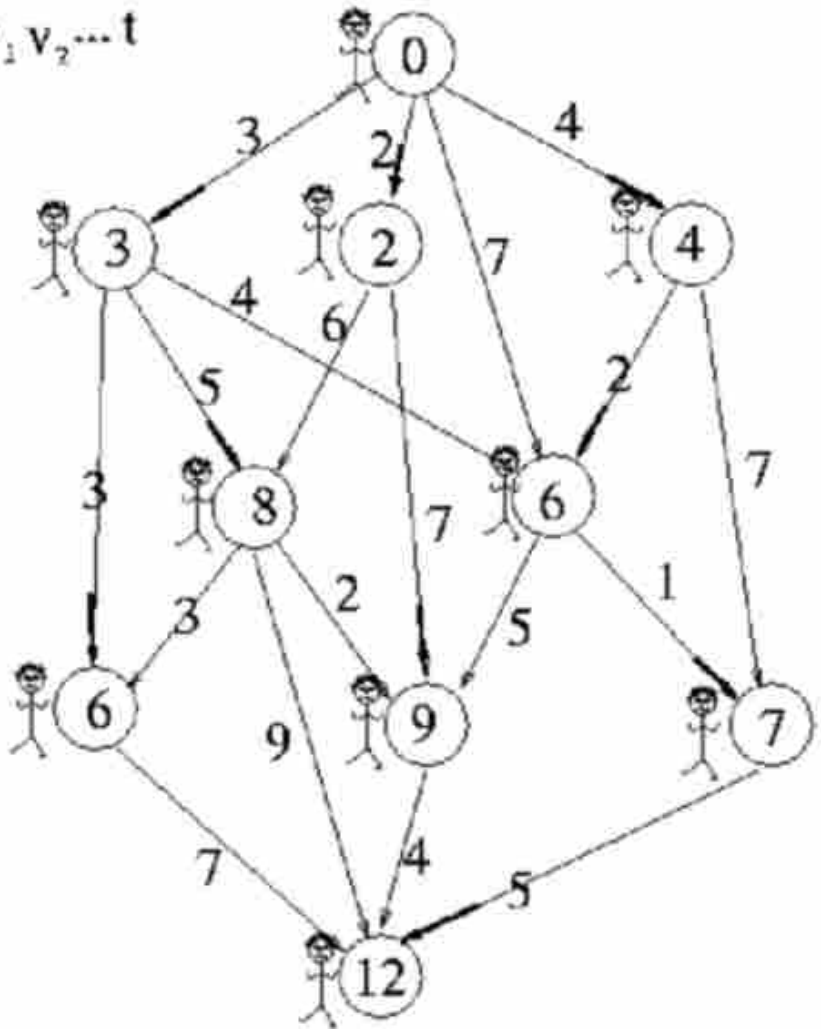
From $0\leq B\leq 1$ and $Q = 2A + B$ we have $B=Q \bmod 2$.
Similarly $b=Q \bmod 2$, and hence $b=B$.

Thus $a=A$, $b=B$, $c=C$, i.e., the solution given by GreedyCoinChange is optimal.

Breaking problems down: dynamic programming

Solve each subinstance

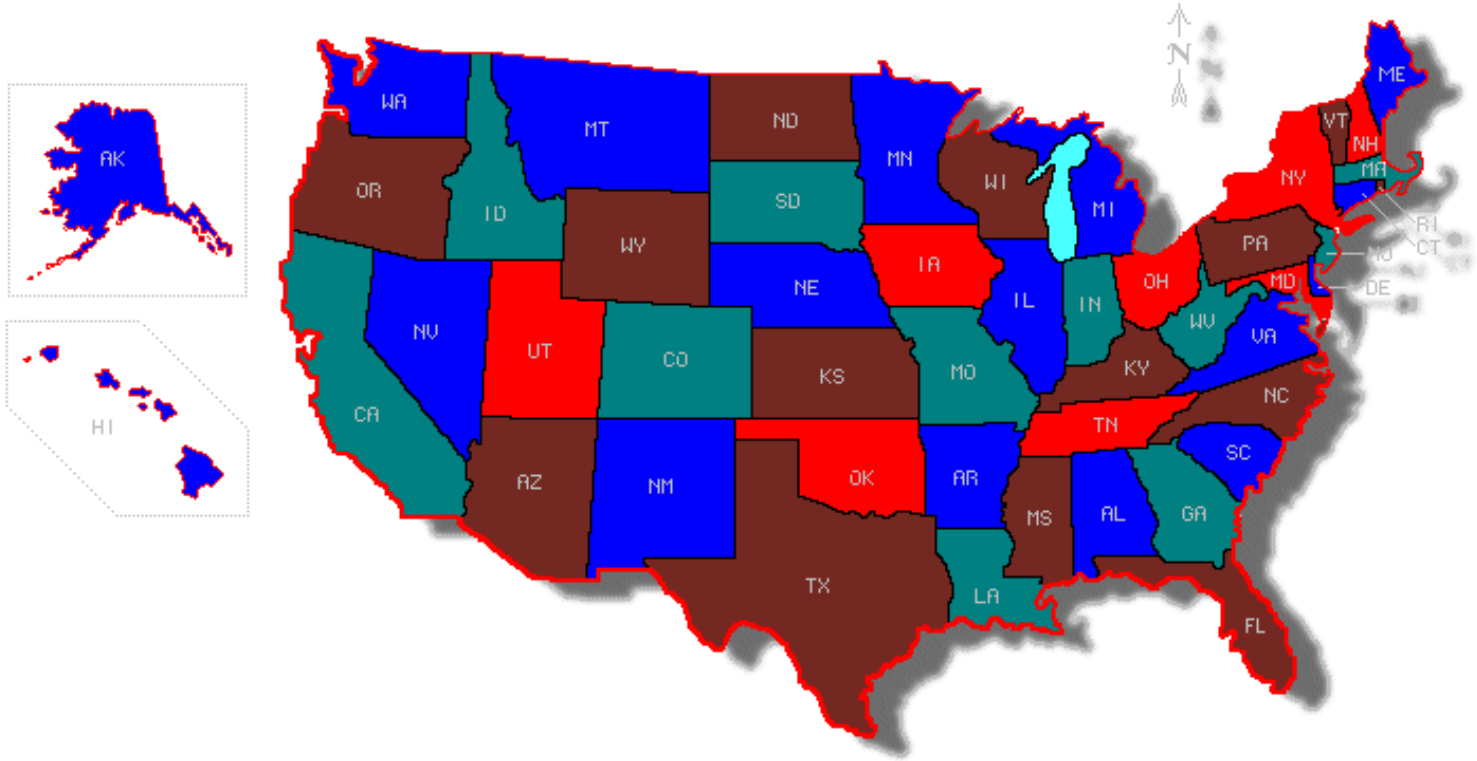
s v_1 v_2 ... t



	j	0	1	2	3	4	5	6	7	8
	y_j	0	1	0	1	1	0	1	0	
i	x_i									
0		0	0	0	0	0	0	0	0	0
1	1	0	0	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2	2
3	0	0	1	1	2	2	2	3	3	3
4	1	0	1	2	2	3	3	3	4	4
5	0	0	1	2	3	3	3	4	4	5
6	1	0	1	2	3	4	4	4	5	5
7	0	0	1	2	3	4	4	5	5	6

Abstract thinking: graph algorithms

Four Colored US Map



4-19-06

Analysis of Algorithms

- How good is the algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency

- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

Correctness

- Termination
  - Well-founded sets: find a non-negative quantity that always decreases as the algorithm is executed

- Partial Correctness
  - For recursive algorithms: induction
  - For iterative algorithms: axiomatic semantics, loop invariants

Complexity

- Space complexity

- Time complexity
  - For iterative algorithms: sums
  - For recursive algorithms: recurrence relations

Shortest path problem

A city has n view points

Buses go from one view point to another

A bus driver wishes to follow the shortest path
(travel time wise).

Every view point is connected to another by a road.

However, some roads are less congested than others.

Also, roads are one-way, i.e., the road from view point 1 to 2, is different from that from view point 2 to 1.

How to find the shortest path between any two pairs?

Naïve approach:

List all the paths between a given pair of view points

Compute the travel time for each.

Choose the shortest one.

How many paths are there between any two view points?

$$n! \cong (n/e)^n$$

Will be impossible to run your algorithm for n = 30.

Machine independent analysis

We assume that every basic operation takes constant time:

Example Basic Operations:

Addition, Subtraction, Multiplication, Memory Access

Non-basic Operations:

Sorting, Searching

Efficiency of an algorithm is measured by the number of basic operations it performs

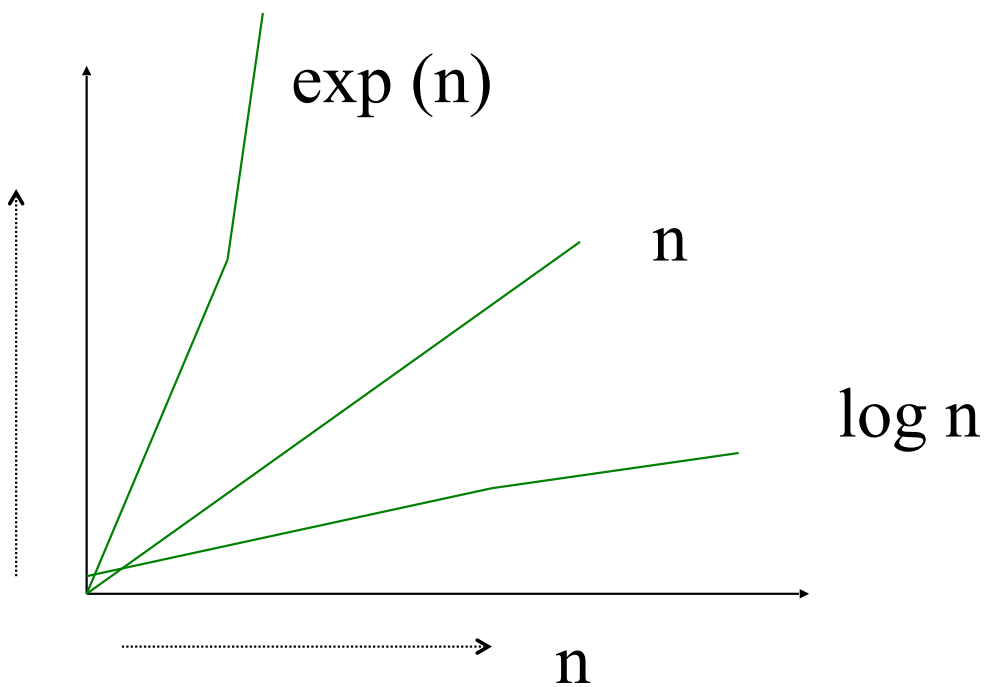
We do not distinguish between the basic operations.

Order of Increase

We care about the speed of our algorithms for large input sizes.

Note that for large n, log(n)/n , and n/exp(n) are very small.

However, n/2n is a constant for all n.



Function Orders

A function f(n) is O(g(n)) if "increase" of f(n) is not faster than that of g(n).

A function f(n) is O(g(n)) if there exists a number n_0 and a nonnegative constant *c* such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$.

If $\lim_{n \to \infty} f(n)/g(n)$ exists and is finite, then f(n) is O(g(n))

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , $n + n^2$

$\lim_{n\to\infty}$ sqrt(n) /n = 0, sqrt(n) is O(n)

$\lim_{n\to\infty}$ n/sqrt(n) = infinity, n is not O(sqrt(n))

$\lim_{n\to\infty}$ n /2n = 1/2, n is O(2n)

$\lim_{n\to\infty}$ 2n /n = 2, 2n is O(n)

$\lim_{n\to\infty} \ln(n)/n = 0,$	ln(n) is O(n)
$\lim_{n\to\infty} n/\ln(n) = \text{infinity},$	n is not O(ln(n))
$\lim_{n\to\infty} \exp(n)/n = \text{infinity},$	exp(n) is not O(n)
$\lim_{n\to\infty} n/\exp(n) = 0,$	n is O(exp(n))
$\lim_{n\to\infty} (n+\text{sqrt}(n))/n = 1,$	n + sqrt(n) is O(n)
$\lim_{n\to\infty} n/(\text{sqrt}(n)+n) = 1,$	n is O(n+sqrt(n))
$\lim_{n\to\infty} (n + n^2)/n = \text{infinity},$	$n + n^2$ is not O(n)
$\lim_{n\to\infty} n/(n + n^2) = 0,$	n is $O(n + n^2)$

Implication of O notation

Suppose we know that our algorithm uses at most O(f(n)) basic steps for any n inputs, and n is sufficiently large, then we know that our algorithm will terminate after executing at most constant times f(n) basic steps.

We know that a basic step takes a constant time in a machine.

Hence, our algorithm will terminate in a constant times f(n) units of time, for all large n.

Other Complexity Notation

Intuitively, (not exactly) f(n) is O(g(n)) means $f(n) \leq g(n)$
→g(n) is an upper bound for f(n).

Now a lower bound notation, $\Omega(n)$

f(n) is $\Omega(g(n))$, if $f(n) \geq cg(n)$ for some positive constant *c*, and all large n.

$$\lim_{n\to\infty} (f(n)/g(n)) > 0, \text{ if } \lim_{n\to\infty} (f(n)/g(n)) \text{ exists}$$

$f(n)$ is $\theta(g(n))$ if $f(n)$ is $O(g(n))$ and $\Omega(g(n))$

$\theta(g(n))$ is ``asymptotic equality''

$\lim_{n\to\infty} (f(n)/g(n))$ is a finite, positive constant, if it exists

$f(n)$ is $o(g(n))$ if $f(n)$ is $O(g(n))$ but not $\Omega(g(n))$

``asymptotic strict inequality''

$f(n)$ is $o(g(n))$ if given any positive constant c, there exists some m such that $f(n) < cg(n)$ for all $n \geq m$

$\lim_{n\to\infty} (f(n)/g(n)) = 0$, if $\lim_{n\to\infty} (f(n)/g(n))$ exists

Asymptotically less than or equal to	O
Asymptotically greater than or equal to	Ω
Asymptotically equal to	θ
Asymptotically strictly less	o

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , $n + n^2$

$\lim_{n\to\infty}$ sqrt(n) /n = 0, sqrt(n) is o(n)

$\lim_{n\to\infty}$ n/sqrt(n) = infinity, n is Ω(sqrt(n))

$\lim_{n\to\infty}$ n /2n = 1/2, n is θ(2n), Ω(2n)

$\lim_{n\to\infty}$ 2n /n = 2, 2n is θ(n), Ω(n)

Implication of the O, Ω Notation

Suppose, an algorithm has complexity O(f(n)) . This means that there exists a positive constant *c* such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at most *c*f(n) steps.

Suppose, an algorithm has complexity Ω(f(n)) . This means that there exists a positive constant *c* such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at least *c*f(n) steps.

Complexity of a problem vs. complexity of an algorithm

A problem is $O(f(n))$ means there is some $O(f(n))$ algorithm to solve the problem.

A problem is $\Omega(f(n))$ means every algorithm that can solve the problem is $\Omega(f(n))$

Worst case vs. average case

Suppose, an algorithm has complexity O(f(n)) . This means that there exists a positive constant c such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at most cf(n) steps.

Worse case complexity!

Much harder to analyze the average case complexity of an algorithm!

Algorithm: conclusions

- Finiteness
  - terminates after a finite number of steps
- Definiteness
  - rigorously and unambiguously specified
- Input
  - valid inputs are clearly specified
- Output
  - can be proved to produce the correct output given a valid input
- Effectiveness
  - steps are sufficiently simple and basic

Exact string matching

String vs. sequence

(Computer science vs. computational biology)

- String
  - a segment of *consecutive* characters.
  - usually called *sequence* in Biology.
- Sequence
  - need not be consecutive (in CS)
  - = String (in computational biology)
- Example:
  - *S* = “a t g a t g c a a t”
  - Substrings of *S*: “g a t g c”, “t g c a a t”.
  - Subsequences of *S*: “a g g t”, “a a a a”.

The exact string matching problem

- Input
  - a string *P* — the pattern
  - a string *S* — the text
- Output
  - all the occurrences of *P* in *S*.

Applications

- Computer Science
  - Dictionary, database
  - Search engines: Yahoo!, Google, …
- **Biology:**
  - **Biological sequence analysis**
- **Warm-up** for this course:
  - A well studied problem,
  - The idea/technique behind.

Notation for strings

- *S* is a string
  - |*S*| = the length of *S.*
  - substring: *S*[*i*…*j*].

A naïve algorithm

- Input: *S* and *P*.
- Output: all occurrences of *P* in *S*.

```
for i=1 to |S|
  if S[i…i+|P|-1] equals P
     output i;
```

Time complexity?

- $O(|S|^2 |P|)$
- $O(|S| |P|)$
- $O(|P| \log|S|)$
- $O(|S| + |P|)$

Tightness of the time complexity

- $O(|S| |P|)$ is tight.
  - Why?
  - S = 1 1 1 1 1 1 1 1 1 1 1
  - P = 1 1 1 1 1 1 1
- $O(|S|^2 |P|)$ is not tight.
  - Why?

Time complexity = $\Theta(|S| |P|)$

Knuth-Morris-Pratt

- [*SIAM J. Computing* 1977]
- Jumping as far as possible
  - far enough (for efficiency)
  - not too far (for correctness)

An example

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
                a b x y a b x z
```

mismatch = a signal for “jumping”.

More, …

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
      a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
```

What if there's no mismatch?

```
S = t t t t t t t t t
P = t t t t
      t t t t
        t t t t
          t t t t
            t t t t
              t t t t
```

Comments

- mismatch = a signal for “jumping”.
- Two types of jumps
  - Skipping several iterations
  - Skipping several comparisons in each iterations

For example, …

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
      a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
```

Comments

- mismatch = a signal for “jumping”.
- Two types of jumps
  - Skipping several iterations
  - Skipping several comparisons in each iterations
- How far is each jump?
  - Determinable merely from *P*.
  - That is, *S* is irrelevant.

For example, …

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
      a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
```

Strategy

Preprocessing *P* in $O(|P|)$ time to obtain the necessary information that correctly guides the "jumps".

Components of KMP algorithm

- The prefix function, Π

The prefix function,Π(i) for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern 'p'. In other words, this enables avoiding backtracking on the string 'S'.

- The KMP Matcher

With string 'S', pattern 'p' and prefix function 'Π' as inputs, finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrence is found.

The prefix function, Π

- P_k: the first k characters of P (the k length prefix of P).
  - P = *ababddc*
  - P_1= *a*
  - P_3= *aba*
- Given $P[1,m]$, define the prefix-function:

$$\pi[j] = \max\{k : k < j \text{ and } P_k \sqsupset P_j\}$$

Suffix

Matching P with itself after (the smallest) shifting j-k!

The prefix function, Π

Following pseudo-code computes the prefix function, Π:

```
Compute-Prefix-Function (p)
m ← length[p]            //' p' pattern to be matched
Π[1] ← 0
k ← 0
for q ← 2 to m
        do while k > 0 and p[k+1] != p[q]
            k ← Π[k]
        If p[k+1] = p[q]
            then k ← k +1
        Π[q] ← k
return Π
```

Example: compute Π for the pattern 'p' below:

a	b	a	b	a	c	a

p

Initially: m = length[p] = 7
Π[1] = 0
k = 0

Step 1: q = 2, k=0
Π[2] = 0

k+1 q

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 2: q = 3, k = 0,
Π[3] = 1

k+1 q

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 3: q = 4, k = 1
Π[4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step 4: q = 5, k =2
Π[5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: q = 6, k = 3
Π[6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	

Step 6: q = 7, k = 1
Π[7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

After iterating 6 times, the prefix function computation is complete: →

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

The KMP Matcher

The KMP Matcher, with pattern 'p', string 'S' and prefix function 'Π' as input, finds a match of p in S.

Following pseudocode computes the matching component of KMP algorithm:

```
KMP-Matcher(S,p)
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                  //number of characters matched
5 for i ← 1 to n                         //scan S from left to right
6     do while  q > 0 and p[q+1] != S[i]
7           q ← Π[q]                     //next character does not match
8      if p[q+1] = S[i]
9            then q ← q + 1              //next character matches
10     if q = m      {                   //is all of p matched?
11            print "Pattern occurs with shift" i – m
12            q ← Π[ q]                  // look for the next match
13     }
```

Note: KMP finds every occurrence of a 'p' in 'S'. That is why KMP does not terminate in step 12, rather it searches remainder of 'S' for any more occurrences of 'p'.

Illustration: given a String 'S' and pattern 'p' as follows:

S	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p	a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

For 'p' the prefix function, Π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1, q = 0
comparing p[1] with S[1]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

P[1] does not match with S[1]. 'p' will be shifted one position to the right.

Step 2: i = 2, q = 0
comparing p[1] with S[2]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p		a	b	a	b	a	c	a							

P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1
Comparing p[2] with S[3] p[2] does not match with S[3]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p			a	b	a	b	a	c	a						

Backtracking on p, comparing p[1] and S[3]

Step 4: i = 4, q = 0
comparing p[1] with S[4] p[1] does not match with S[4]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p				a	b	a	b	a	c	a					

Step 5: i = 5, q = 0
comparing p[1] with S[5] p[1] matches with S[5]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Step 6: i = 6, q = 1

Comparing p[2] with S[6] p[2] matches with S[6]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Step 7: i = 7, q = 2

Comparing p[3] with S[7] p[3] matches with S[7]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c					

Step 8: i = 8, q = 3

Comparing p[4] with S[8] p[4] matches with S[8]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Step 10: i = 10, q = 5

Comparing p[6] with S[10] p[6] doesn't match with S[10]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4

Comparing p[5] with S[11] p[5] matches with S[11]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b			a		

Step 12: i = 12, q = 5

Comparing p[6] with S[12] p[6] matches with S[12]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Step 13: i = 13, q = 6

Comparing p[7] with S[13] p[7] matches with S[13]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Pattern 'p' has been found to completely occur in string 'S'. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

Running - complexity analysis

matched

```
Π[1] ← 0
k ← 0
for q ← 2 to m
        do while k > 0 and p[k+1] != p[q]
                k ← Π[k]
        If p[k+1] = p[q]
                then k ← k +1
        Π[q] ← k
return Π
```

=Ω(|p|)

KMP Matcher

```
q ← 0
for i ← 1 to n
    do while  q > 0 and p[q+1] != S[i]
            q ← Π[q]
    if p[q+1] = S[i]
        then q ← q + 1
    if q = m          {
        print "Pattern occurs with shift" i – m
        q ← Π[ q]
    }
```

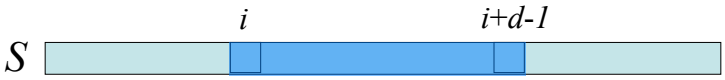
= Ω(|p|)
=O(|s|)

Another approach

Dan Gusfield's *Z* values

The Z values of a string S

- $Z(i)$ of a string S is the largest integer d such that $S[1\ldots d] = S[i\ldots i+d-1]$



Clearly, …

- If $Z(1)$, $Z(2)$, …, $Z(|S|)$ are the Z values of S, then
  - $Z(1) = |S|$;
  - $Z(i) \geq 0$ for each $i = 1, 2, \ldots, |S|$.

For example, …

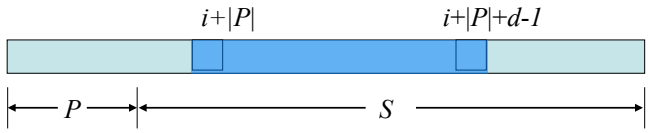
```
S =  a a g c a a t a a a g c
Z = 12 1 0 0 2 1 0 2 4 1 0 0
       a     a a   a a
             a     a a g c
                     a
```

Question

How do we find all occurrences of *P* in *S* using *Z* values?

Exact string matching with Z values

```
computing Z values of PS;
for i=1 to |S|
  if Z(i+|P|)≥ |P| then
    output i;
```



Time complexity?

```
computing Z values of PS;
for i=1 to |S|
  if Z(i+n)>=|P| then
    output i;
```

- $O(|S|)$ + time for computing the Z values of PS.

Z(i) can be naively computed in Ω(ISI) time

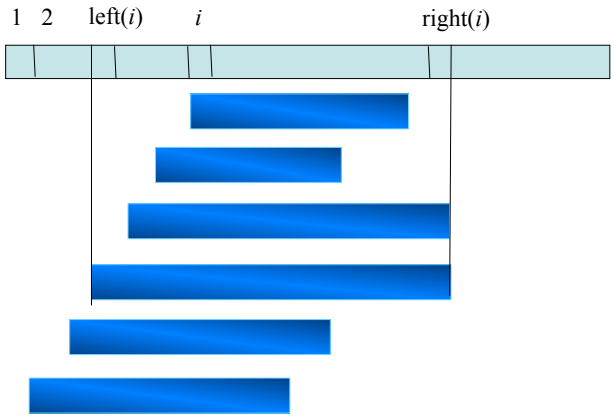
```
For i=1 to |S| {
  set j = i;
  set Z(i) = 0;
  while (S[j]==S[j-i+1]){
    Z(i)++;
    j++;
  }
}
```

Calculate Z values in linear time

Notation

- $\text{right}(i) = \max\{j + Z(j) - 1 \mid 1 < j \leq i\}$.
- $\text{left}(i) = \min\{j \mid \text{right}(j) = \text{right}(i)\}$.
- Observation: left(i) and right(i) nondecreasing.

Illustration



Strategy

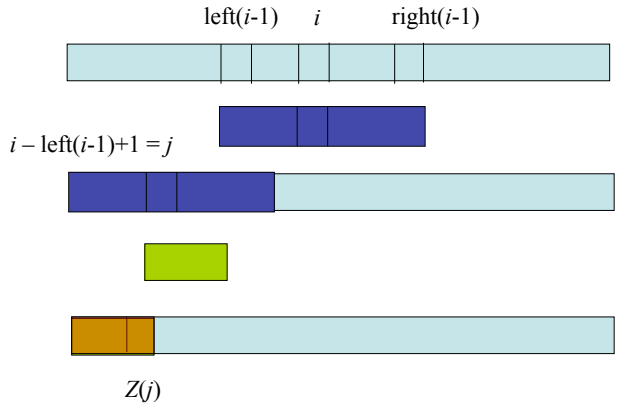
- Computing $Z(i)$, right(i), left(i) from
  - $Z(1)$, $Z(2)$, …, $Z(i-1)$;
  - right($i-1$);
  - left($i-1$).

Case 1: right(i-1) ≤ i-1.

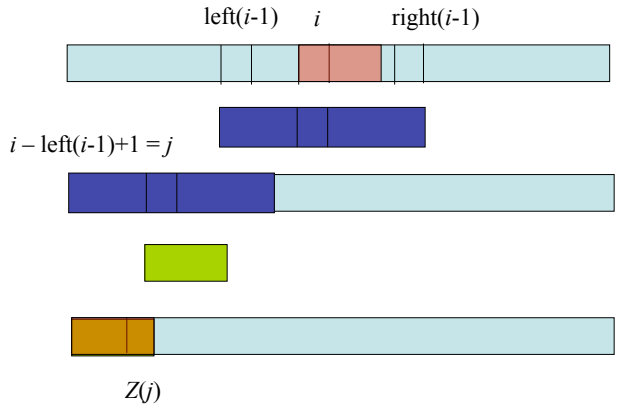
- right(i-1) does not cover i.
- Computing $Z(i)$ naively in $O(1+Z(i))$ time.
- left(i) = i.
- right(i) = $i + Z(i) - 1$.
- Observation

  $1+Z(i) = \text{right}(i) - i + 2 \leq \text{right}(i) - \text{right}(i-1) + 1$.

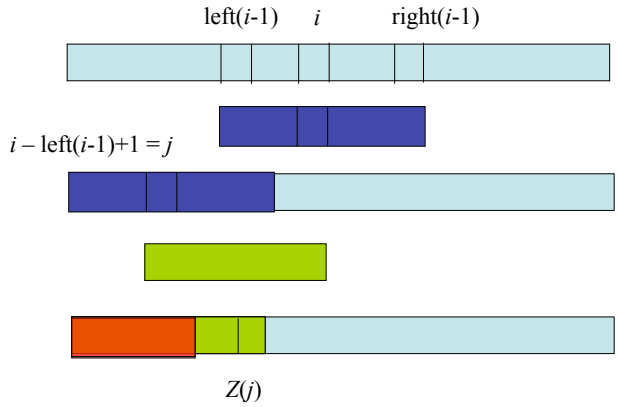
Case 2:right(i-1) ≥ i and Z(j) < right(i-1)-i+1.



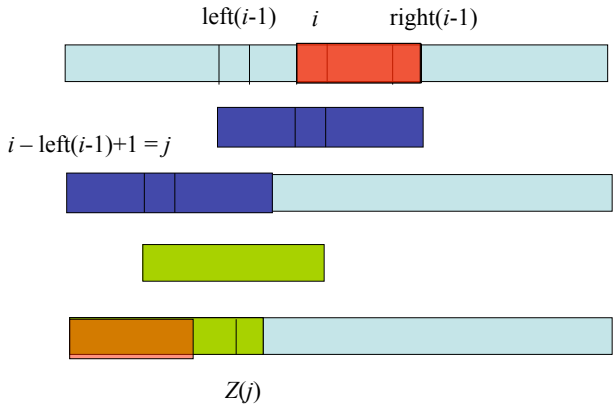
Z(i) = Z(j), left(i) = left(i-1), right(i) = right(i-1).



Case 3:right(i-1) ≥ i and Z(j) ≥ right(i-1)-i+1.

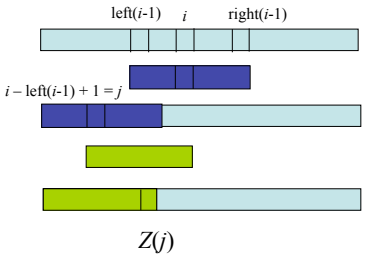


Finding Z(i) by comparisons starting from right(i-1)+1.



Computing left(i) and right(i).

- right(i) = i + Z(i) -1.
- left(i) = i.
- How many comparisons?
  - right(i)-right(i-1)+1.



Time complexity is linear

- Case 1:
  - $O(|Z(i)|+1) = O(\text{right}(i)-\text{right}(i-1)+1)$.
- Case 2:
  - $O(1) = O(\text{right}(i)-\text{right}(i-1)+1)$.
- Case 3:
  - $O(\text{right}(i)-\text{right}(i-1)+1)$.

Overall time complexity: O(ISI)

Z-value Pseudo code

```
int L = 0, R = 0;
for (int i = 1; i < n; i++) {
  if (i > R) {
    L = R = i;
    while (R < n && s[R-L] == s[R]) R++;
    z[i] = R-L; R--;
  } else {
    int k = i-L;
    if (z[k] < R-i+1) z[i] = z[k];
    else {
      L = i;
      while (R < n && s[R-L] == s[R]) R++;
      z[i] = R-L; R--;
    }
  }
}
```

Dr. Gusfield's lecture

- https://www.youtube.com/watch?v=MFK0WYeVEag

Suffix trees

Data structures for string pattern matching: Suffix trees

- Linear algorithms for exact string matching
  - KMP
  - Z-value algorithm
- What is suffix tree?
  - A tree-like **data structure** for solving problems involving strings.
  - Related data structures: Trie (re**trie**val) & PATRICIA (radix tree)
  - Allow the storage of **all substrings** of a given string in linear space
  - Simple algorithm to solve string pattern matching problem in linear time

Better than hash tables?

- Hash tables are certainly easier to understand. And, one can produce a hash table of all length *k* strings in O(*m*) time and look up a *k*-length string *x* in O(*k)* time, finding all *p* places where string *x* is found in O(*p*) time. This is the same as the bound for suffix trees.
- What if you don't know how long the string *x* is going to be?
- And most other string matching tricks don't work for it either.

Suffix Tree: definition

- A suffix tree *ST* for an m-character string *S* is a rooted directed tree with exactly *m* leaves numbered 1 to *m*.
- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of *S*.

Suffix tree: definition

- No two edges out of a node can have edge-labels beginning with the same character.
- The key feature of the suffix tree is that for any leaf *i*, the concatenation of the edge-labels on the path from the root to the leaf *i* exactly spells out the suffix of S that starts at position *i*.

Suffix Trees: Example

- Suffixes of ‘papua’
  - ‘papua’
  - ‘apua’
  - ‘pua’
  - ‘ua’
  - ‘a’
  - ‘’

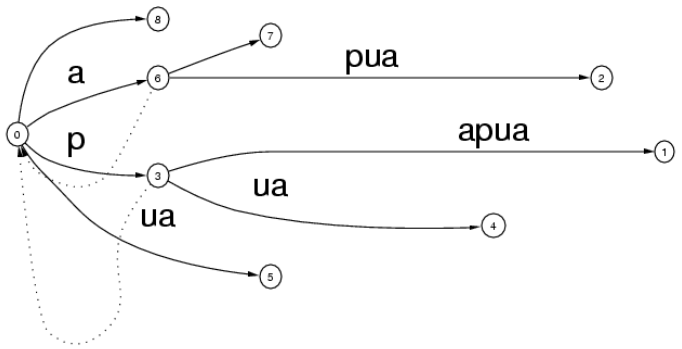
Suffix Trees: Example

- Suffixes of ‘papua’
  - ‘papua’
  - ‘apua’
  - ‘pua’
  - ‘ua’
  - ‘a’
  - ‘’

NOTE: Assume the string terminates with some character found nowhere else in the string. (eg. ‘\0’)

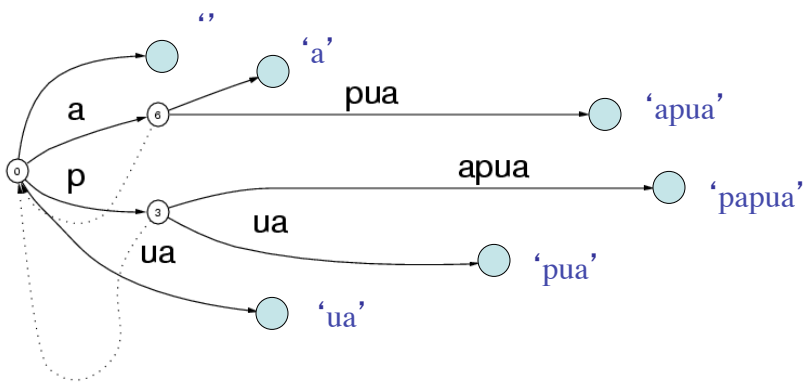
Suffix Trees: Example

- Suffix tree for ‘papua’

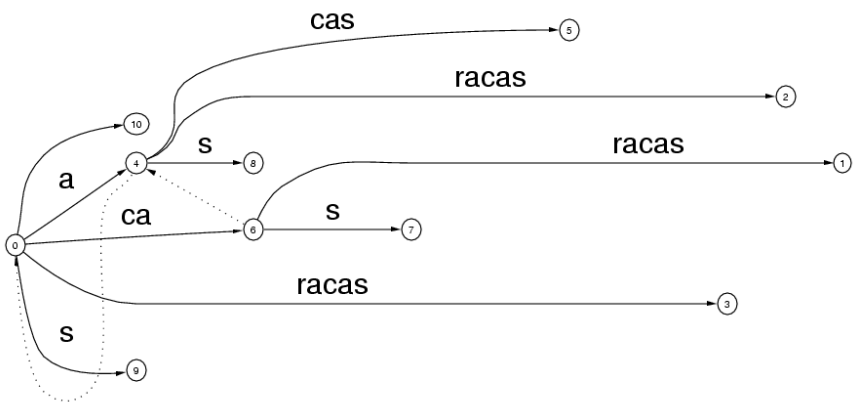


Suffix Trees: Example

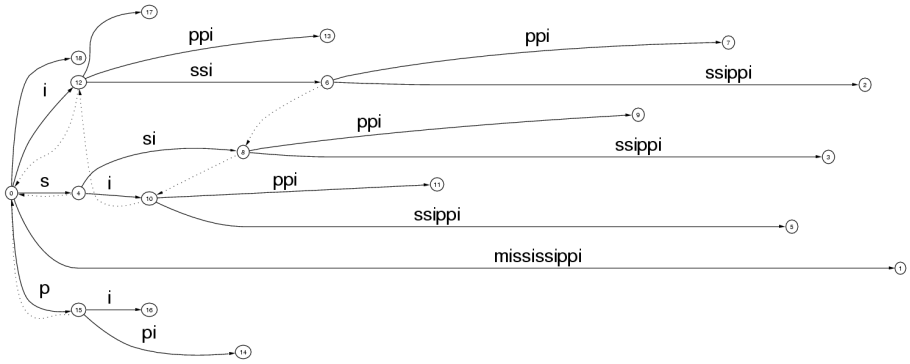
- Suffix tree for ‘papua’



caracas



mississippi



Suffix Trees

- Exact matching in linear time
- Many others
- “We know of no other single data structure that allows efficient solutions to such a wide range of complex string problems.” - Dan Gusfield

Exact string matching problem

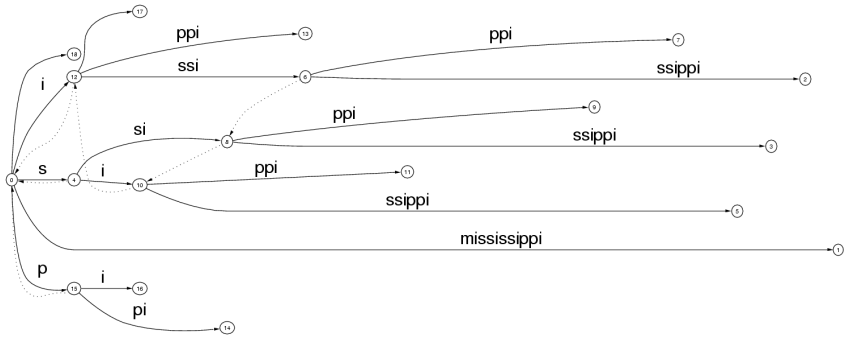
- Given a pattern *P* of length *m*, find all occurrences of *P* in text *T*
  - *O(n+m)* algorithm
- Solution: Build a suffix tree *ST* for text *T* in *O(m)* time. Then, match the characters of *P* along the unique path in *ST* until either *P* is exhausted or no more matches are possible.

Exact string matching problem

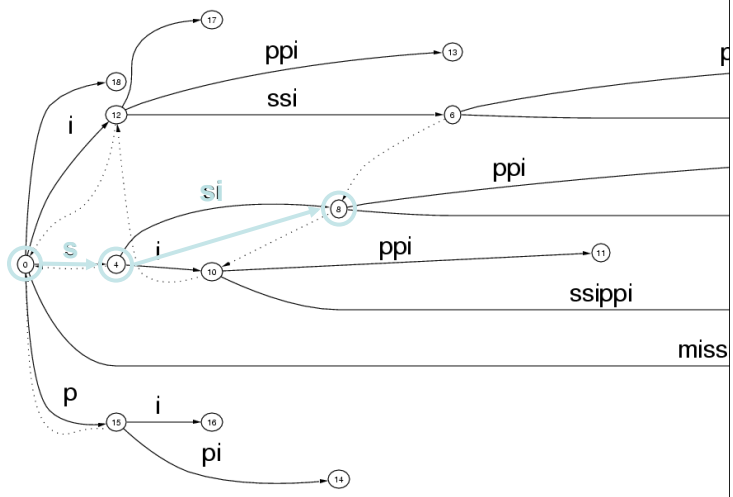
- Find 'ssi' in 'mississippi'

Exact string matching problem

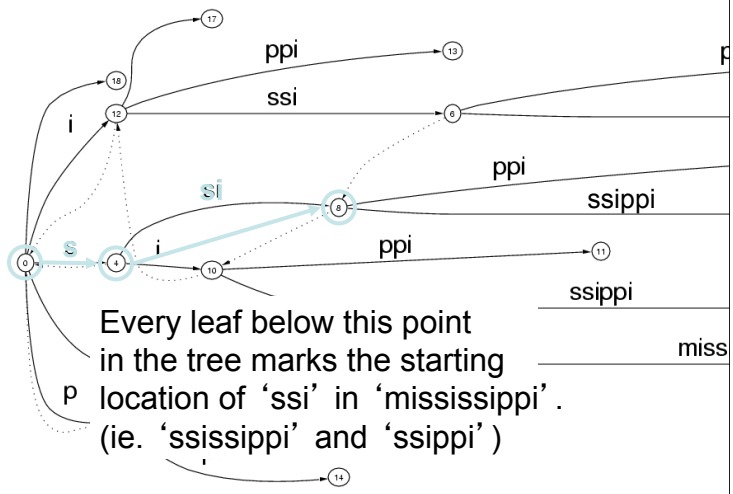
- Find ‘ssi’ in ‘mississippi’



Exact string matching problem



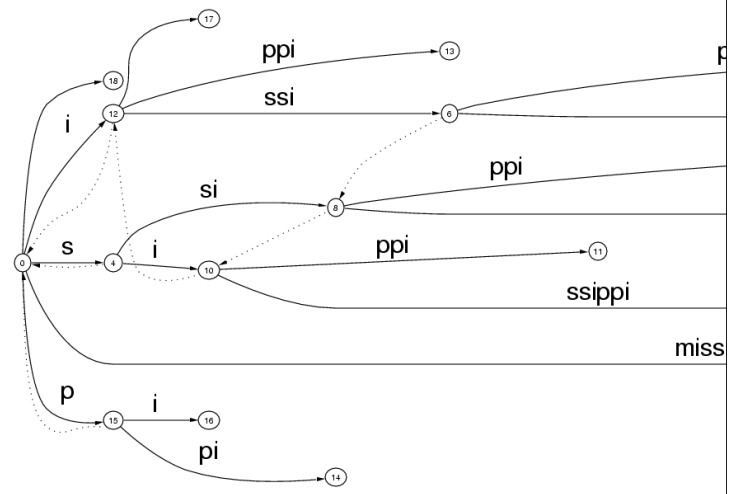
Exact string matching problem



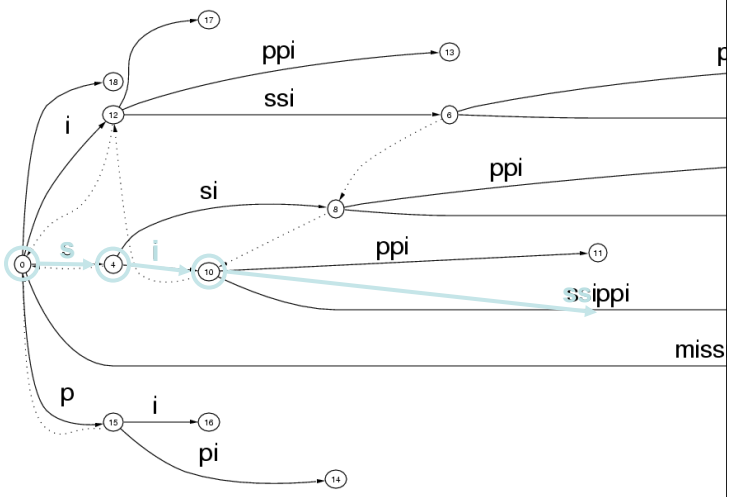
Exact string matching problem

- Find 'sissy' in 'mississippi'

Exact string matching problem



Exact string matching problem



Comparing to the other algorithms

- KMP and Boyer-Moore both achieve this worst case bound.
  - *O(m+n)* when the text and pattern are presented together.
- Suffix trees are <u>much</u> faster when the text is fixed and known first while the patterns vary.
  - *O(m)* for single time processing the text, then only *O(n)* for each new pattern.
- Based on suffix trees, is faster for searching a number of patterns at one time against a single text (**exact set matching problem**)
  - Aho-Corasick algorithm: preprocessing P instead of T.

Building the Suffix Tree

- How do we build a suffix tree?

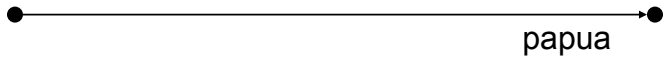
```
while suffixes remain:
   add next shortest suffix to the tree
```

Building the Suffix Tree

- papua

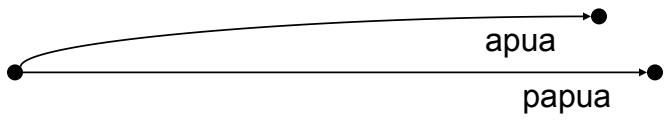
Building the Suffix Tree

- papua



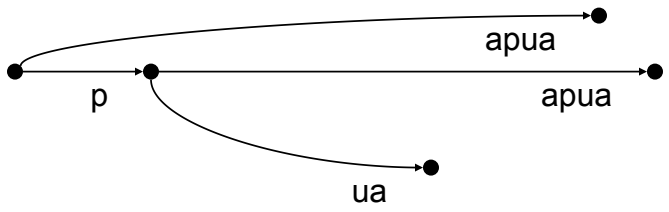
Building the Suffix Tree

- papua



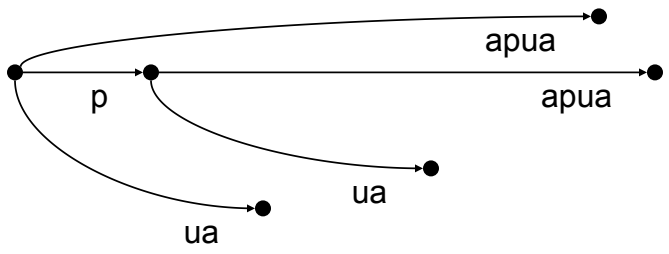
Building the Suffix Tree

- papua



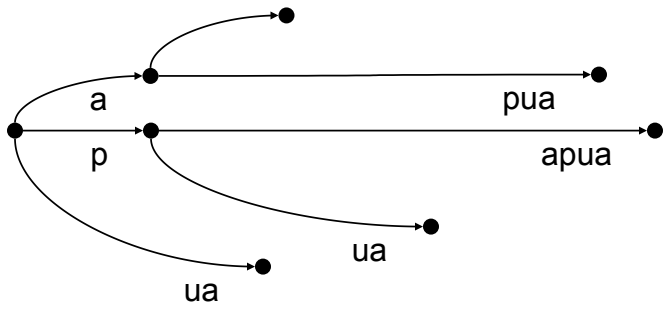
Building the Suffix Tree

- papua



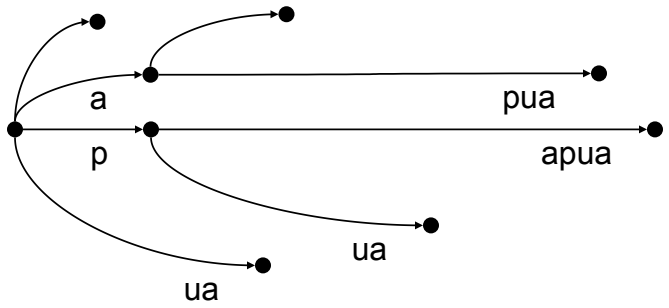
Building the Suffix Tree

- papua



Building the Suffix Tree

- papua



Building the Suffix Tree

- How do we build a suffix tree?

```
while suffices remain:
   add next shortest suffix to the tree
```

Naïve method - $O(m^2)$ (m = text size)

Building the Suffix Tree in O(m) time

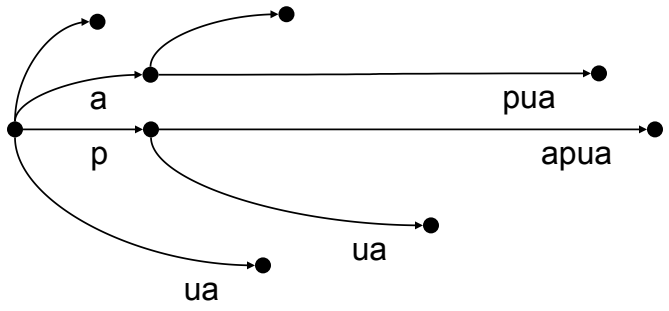
- In the previous example, we assumed that the tree can be built in *O(m)* time.
- Weiner showed original *O(m)* algorithm (Knuth is claimed to have called it "the algorithm of 1973")
- More space efficient algorithm by McCreight in 1976
- Simpler 'on-line' algorithm by Ukkonen in 1995

Ukkonen's Algorithm

- Build suffix tree *T* for string *S[1..m]*
  - Build the tree in *m* phases, one for each character. At the end of phase *i*, we will have tree T_i, which is the tree representing the prefix *S[1..i]* → online construction
    - In each phase *i*, we have *i* extensions, one for each character in the current prefix. At the end of extension *j*, we will have ensured that *S[j..i]* is in the tree T_i.

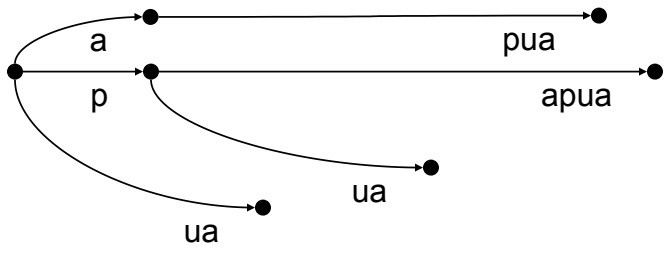
Implicit suffix tree

- papua



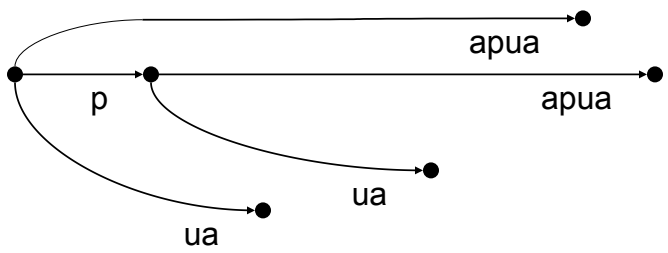
Implicit suffix tree

- papua



Implicit suffix tree

- papua



Implicit suffix tree can be transformed from/into regular suffix tree in O(n) time.

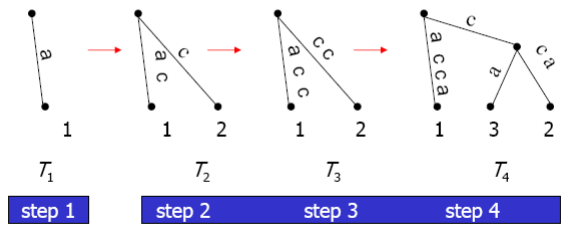
Ukkonen's Algorithm

Pseudo code for Ukk:

```
Construct tree T1.
for i = 1 to m–1 do
begin {phase i+1}
   for j = 1 to i +1 do
        begin {extension j}
            In the current tree find the end of the path from the root
            labeled t[j ... i]. If necessary, extend that path by adding
            character t[i+1], thus ensuring that string t[j...i+1] is in the
            tree.
         end;
end;
```

An Exmaple

t = a c c a $



Ukkonen's Algorithm

- This is an $O(m^3)$ time, $O(m^2)$ space algorithm.
- We need a few implementation speed-ups to achieve the $O(m)$ time and $O(m)$ space bounds.

Suffix extension rules

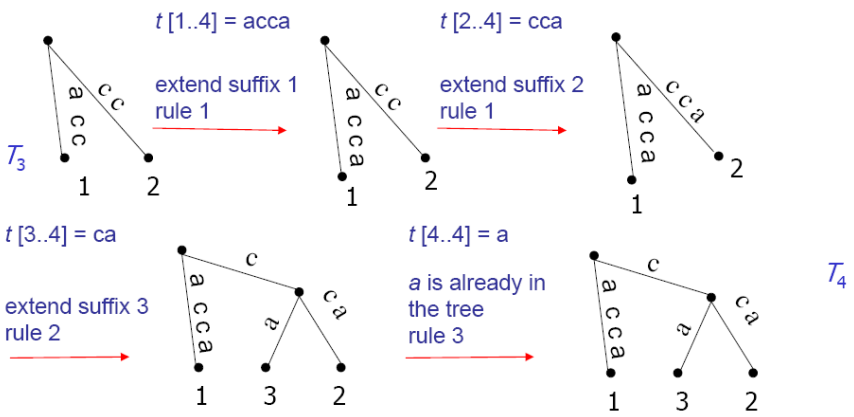
- 3 possible ways to extend *S[j..i]* with character *i +1.*
  1. *S[j..i]* ends at a leaf. Add the character *i+1* to the end of the leaf edge.
  2. No path from the end of *S[j..i]* starts with the *i+1* character. Split the edge and create a new node if necessary, then add a new leaf with character *i+1*.
     (This is the only extension that increases the number of leaves! The new leaf represents the suffix starting at position *j.)*
  3. There is already a path from the end of *S[j..i]* starts with the *i+1* character, or S[j..i+1] correspond to a path. Do nothing.

Ukkonen's Algorithms

t = a c c a $

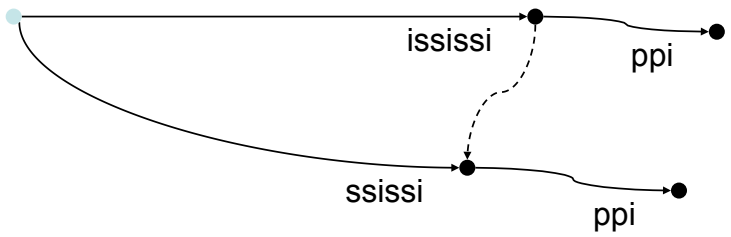
t [1...3] = acc

t [1...4] = acca



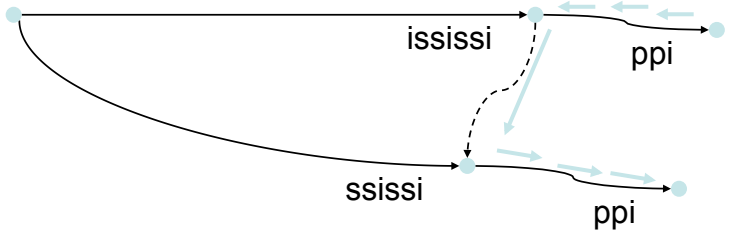
Ukkonen's Algorithm: Speed-up 1

- Suffix Links
  - speed up navigation to the next extension point in the tree



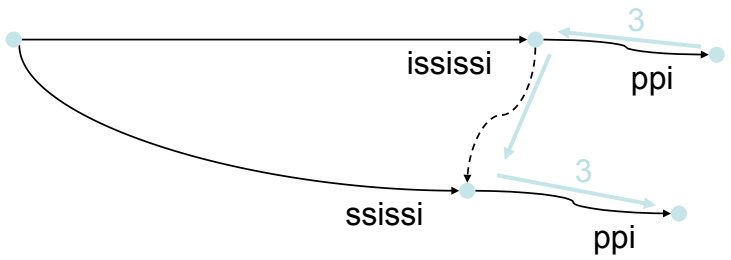
Ukkonen's Algorithm: Speed-up 2

- Skip/Count Trick
  - instead of stepping through each character, we know that we can just jump, as long as we're the right distance



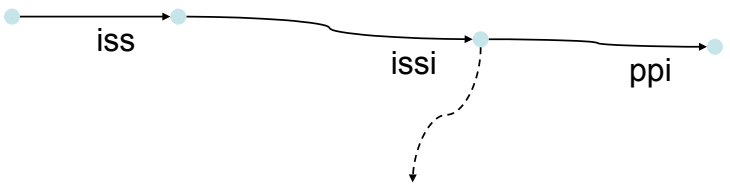
Ukkonen's Algorithm: Speed-up 2

- Skip/Count Trick
  - instead of stepping through each character, we know that we can just jump, as long as we're the right distance



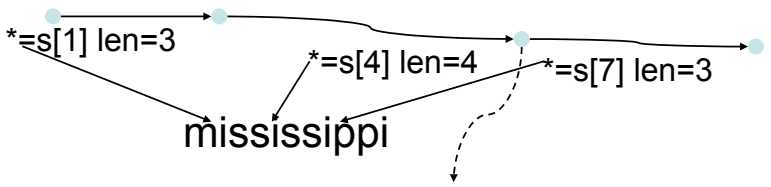
Ukkonen's Algorithm: Speed-up 3

- Edge-Label Compression
  - since we have a copy of the string, we don't need to store copies of the substrings for each edge



Ukkonen's Algorithm: Speed-up 3

- Edge-Label Compression
  - since we have a copy of the string, we don't need to store copies of the substrings for each edge
  - *O(m²)* space becomes *O(m)* space



Ukkonen's Algorithm: Speed-up 4

- A match is a show stopper.
  - If we find a match to our next character (rule 3 applies), we're done this phase.

Ukkonen's Algorithm: Speed-up 5

- Once a leaf, always a leaf (implicitly implement rule 1).
  - We don't need to update each leaf, since it will <u>always</u> be the end of the current string. We can get these updates for free.
  - Either 1)maintain a global *end-of-string* index or 2) insert the whole string for every leaf

Ukkonen's Algorithm: Speed-ups

- Because of speed-ups 4 and 5, we can pick up the next phase right where we ended the last one!

Ukkonen's Algorithm – mississippi with Speed-ups

```
void SuffixTree::update(char* s, int len) {
  …
  int i;
  int j;
  …
  for (i = 0, j = 0; i < len; i++) {
    while (j <= i) {
      … all the work …
    }
  }
}
```

Possible execution

Extension Case distribution

j=	1	2	3	4	5	6	7	8	9=m
i=1:	1	3							
i=2:	1	1	2						
i=3:	1	1	2	2					
i=4:	1	1	3	2	2				
i=5:	1	1	1	2	2	2			
i=6:	1	1	1	3	3	2	2		
i=7:	1	1	1	1	1	3	2	2	
i=8:	1	1	1	1	1	1	3	2	2

Need to compute:

At most one Case 2 per i

At most one Case 1 or 3 per j

Creating a true suffix tree

- Run another iteration of Ukkonen algorithm on S$
- No suffix is now a prefix of any other suffix.
- As a result, each suffix will end at a leaf.
- Replace each index on every leaf edge with the number m.

Total Algorithm time O(m)

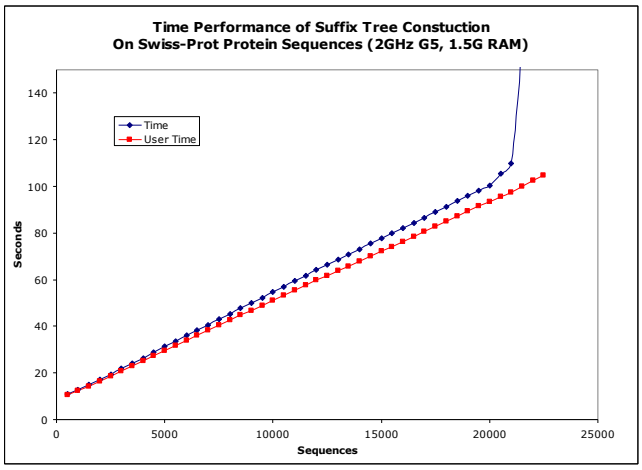
Ukkonen's Algorithm – The Punch Line

- By combining all of the speed-ups, we can now construct a suffix tree T_m representing the string *S[1..m]* in
  - *O(m)* time and in
  - *O(m)* space!

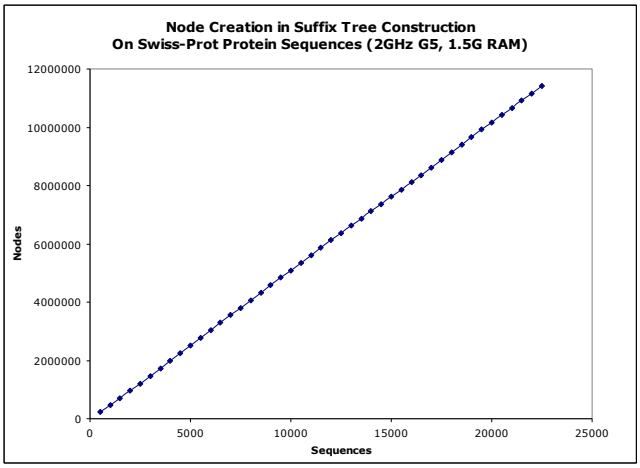
Exact string matching

- Both P (|P|=n) and T (|T|=m) are known:
  - Suffix tree method achieves same worst-case bound O(n+m) as KMP.
- T is fixed and build suffix tree, then P is input, k is the number of occurrences of P
  - Using suffix tree: O(n+k)
  - In contrast (KMP, preprocess P): O(n+m) for any single P
- P is fixed, then T is input
  - Selecting KMP rather than suffix tree
  - or Aho-Corasick algorithm (exact set matching problem)

Ukkonen's Algorithm – Time Performance



Ukkonen's Algorithm – Memory Usage



Applications

- Problems
  - linear-time longest common substring
  - constant-time least common ancestor
  - maximally repetitive structures
  - all-pairs suffix-prefix matching
  - compression
  - inexact matching
  - conversion to suffix arrays

Bioinformatics applications

- Applications
  - Sequence comparison
  - motif discovery
  - PST – probabilistic suffix trees
  - SVM string kernels
  - chromosome-level similarities and rearrangements

Burrows-Wheeler Transform (BWT)

Slides from Eleazar Eskin and Ben Langmead

Short Read Sequencing Problem (A Computer Science Problem)

Full DNA Sequence

```
AGAGCAGTCGAC
AGGTATAGTCTA
CATGAGATCGAC
ATGAGATCGGTA
GAGCCGTGAGAT
CGACATGATAGC
CAGAGCAGTCGA
CAGGTATAGTCT
ACATGAGATCGA
CATGAGATCGGT
AGAGCCGTGAGA
TCGACATGATAG
CCAGAGCAGTCG
ACAGGTATAGTC
TACATGAGATCG
ACATGAGATCGG
TAGAGCCGTGAG
ATCGACATGATA
GCCAGAGCAGTC
GACAGGTATAGT
CTACATGAGATC
```

- Short read sequencers generate random short substrings from the DNA sequence of a certain length.

```
ATGAGATCGGTAGAGCCGTGAGAT
GAGCAGTCGACAGGTATAGTCTAC
AGAGCAGTCGACAGGTATAGTCTA
TGAGATCGACATGATAGCCAGAGC
TAGCCAGAGCAGTCGACAGGTATA
GATAGCCAGAGCAGTCGACAGGTA
GAGATCGACATGATAGCCAGAGCA
GCAGTCGACAGGTATAGTCTACAT
AGCAGTCGACAGGTATAGTCTACA
TCGACATGAGATCGGTAGAGCCGT
CAGTCGACAGGTATAGTCTACATG
GAGATCGACATGATAGCCAGAGCA
GTAGAGCCGTGAGATCGACATGAT
```

Short Reads Difficulties

```
ATGAGATCGGTAGAGCCGTGAGAT
GAGCAGTCGACAGGTATAGTCTAC
AGAGCAGTCGACAGGTATAGTCTA
TGAGATCGACATGATAGCCAGAGC
TAGCCAGAGCAGTCGACAGGTATA
GATAGCCAGAGCAGTCGACAGGTA
GAGATCGACATGATAGCCAGAGCA
GCAGTCGACAGGTATAGTCTACAT
AGCAGTCGACAGGTATAGTCTACA
TCGACATGAGATCGGTAGAGCCGT
CAGTCGACAGGTATAGTCTACATG
GAGATCGACATGATAGCCAGAGCA
GTAGAGCCGTGAGATCGACATGAT
```

- We don't know where each read comes from!
- Can't identify where the mutations are!
- What do we do?

Key Idea: "Re"-Sequencing

We know that my genome is very close to the Human genome.

My Genome:
TACATGAGATC**G**ACATGAGATC**G**GTAGAGC**C**GTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

The Human Genome:
TACATGAGATC[C]ACATGAGATC[T]GTAGAGC[T]GTGAGATC
TC[G]ACATGAGATC[G]GTAGAGC[C]GT

Recovered Sequence:
TACATGAGATC**G**ACATGAGATC**G**GTAGAGC**C**GTGAGATC

"Re"-Sequencing Output

Resequencing provides a list of changes to make from the reference to change it to the target. Similar to unix "diff".

My Genome:
TACATGAGATCGACATGAGATCGGTAGAGCCGTGAGATC

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
G G C

A list of changes

Recovered Sequence:
TACATGAGATCGACATGAGATCGGTAGAGCCGTGAGATC

Algorithmic "Re"-Sequencing Challenges

- Sequences are long!
  - **Human Genome is 3,000,000,000 long.**
- Sequencers generate many reads!
  - **A sequencer generates over 1,000,000,000 reads.**
- We need efficient algorithms to "map" each read to its location in the genome.

There are other challenges which we are not mentioning.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

- We can slide our read along the genome and count the total mismatches between the read and the genome.
- If the mismatches are below a threshold, we say that it is a match.

```
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
TCGACATGAGATCGGTAGAGCCGT
```

Total of 18 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

```
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
 TCGACATGAGATCGGTAGAGCCGT
```

Total of 15 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

```
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
   TCGACATGAGATCGGTAGAGCCGT
```

Total of 23 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

```
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
          TCGACATGAGATCGGTAGAGCCGT
```

Total of 23 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

```
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
          TCGACATGAGATCGGTAGAGCCGT
```

Total of 3 mismatches. Below threshold. A match!

Complexity of Trivial Algorithm

- 3,000,000,000 length genome (N)
- 300,000,000 reads to map (M)
- Reads are of length 30 (L)
- Number of mismatches allowed is 2 (D).
- Each comparison of match vs. mismatch takes 1/1,000,000 seconds (t).

Total Time = N*M*L*t = 27,000,000,000,000 seconds or 864,164 years!

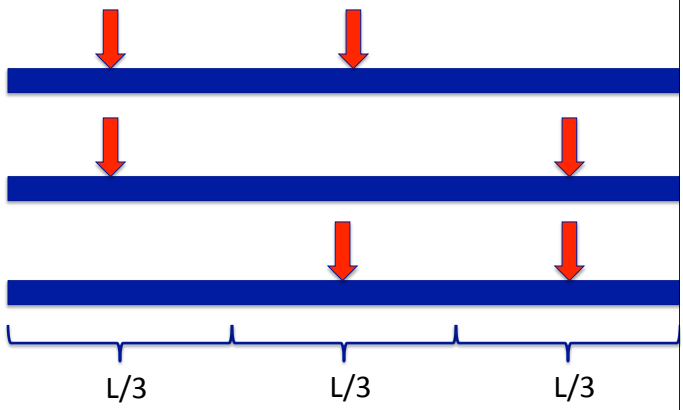
- Important: Trivial algorithm only solves problem under assumptions.

Some observations

- Most positions in the genome match very poorly.
- We are looking for only a few mismatches. (D is small)
- A substring of our read will match perfectly.

Perfect Matching Read Substrings

Three "worst" possible cases for placement of mutations.



- In each case, there is a perfect match of L/3.

Finding a perfect match of length L/3

- Intuition: Create an index (or phone book) for the genome.
- We can look up an entry quickly.

Sequence	Positions
AAAAAAAAAA	32453, 64543, 76335
AAAAAAAAAC	64534, 84323, 96536
AAAAAAAAAG	12352, 32534, 56346
AAAAAAAAAT	23245, 54333, 75464
AAAAAAAACA	
AAAAAAAACC	43523, 67543
…	
CAAAAAAAAA	32345, 65442
CAAAAAAAAC	34653, 67323, 76354
…	
TCGACATGAG	54234, 67344, 75423
TCGACATGAT	11213, 22323
…	
TTTTTTTTTG	64252
TTTTTTTTTT	64246, 77355, 78453

If L=30, each entry will have a key of length 10. Each entry will contain on average $N/4^{10}$ positions. (Approximately 3,000).

If L=45, each entry will have a key of length 15. Each entry will contain on average 3 positions.

Complexity of Indexing Algorithm

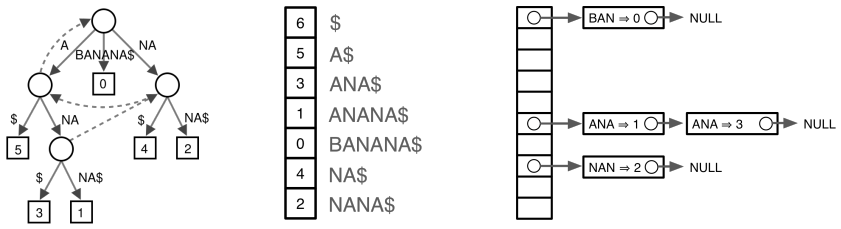
- We need to look up each third of the read in the index.
- For L=30, our index will contain entries of length 10. Each entry will contain on average $N/(4^{L/3})$ or 3,000 positions.
- For each position, we need to compute the number of mismatches.
- Our running time is $L^* M^*3^*N/(4^{L/3})^*T$=81,000,000 seconds or 937 days.
- If L=45, then the time is 81,000 seconds or 22.5 hours.

Indexing a genome

- To find exactly matching substrings, we need to build an index for the whole genome.
- Problem: The genome is BIG!

Indexing

- Genome indices can be big. For human:



> 35 GBs > 12 GBs > 12 GBs

- Large memory requirement implications
  - **Requires large memory machine (expensive)**
  - **Partition genome and index each part (slow)**

Burrows-Wheeler Transform

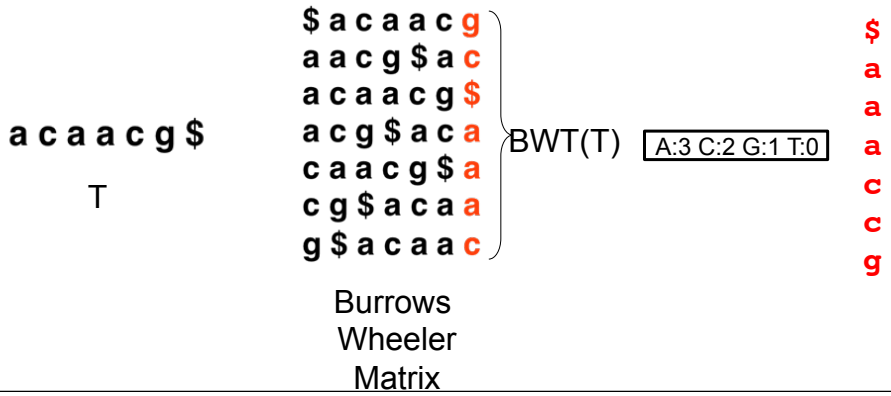
- http://en.wikipedia.org/wiki/Burrows-Wheeler_transform
- Reversible permutation used originally in compression

- Once BWT(T) is built, *all else shown here is discarded*
  - **Matrix will be shown for illustration only**

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

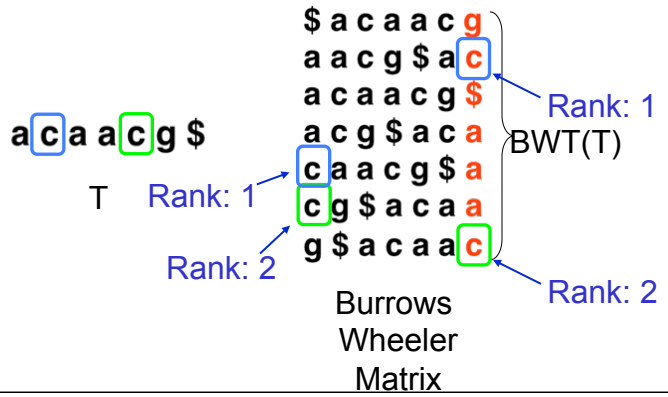
Burrows-Wheeler Transform

- Store only last column
- First column can be recovered by counting symbols in last column because it is sorted



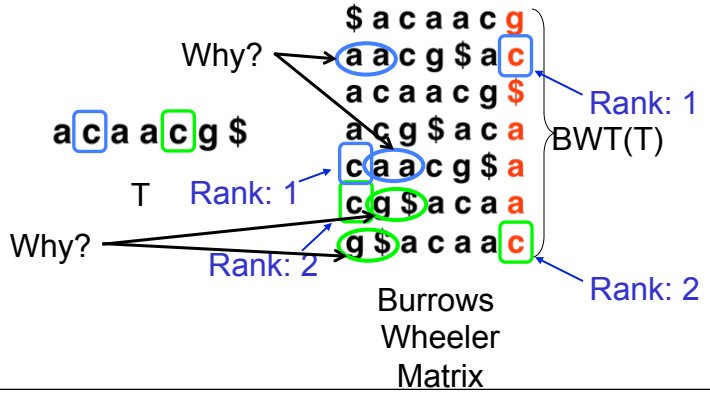
Burrows-Wheeler Transform

- Property that makes BWT(T) reversible is "LF Mapping"
  - **i^{th} occurrence of a character in Last column is same *text* occurrence as the ith occurrence in First column**



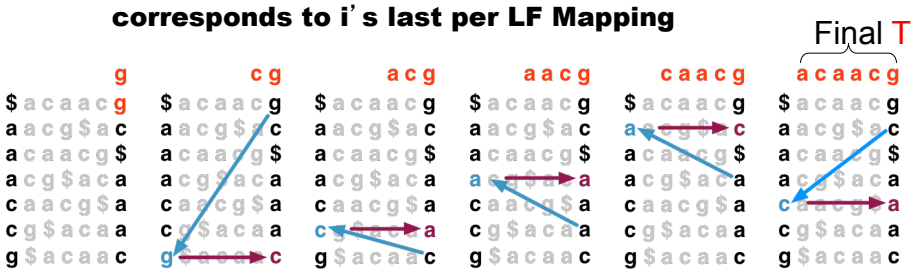
Burrows-Wheeler Transform

- Property that makes BWT(T) reversible is "LF Mapping"
  - **i^{th} occurrence of a character in Last column is same *text* occurrence as the ith occurrence in First column**



Burrows-Wheeler Transform

- To recreate T from BWT(T), repeatedly apply rule:
  - **T = BWT[LF(i)] + T; i = LF(i)**
  - **Where LF(i) maps row i to row whose first character corresponds to i's last per LF Mapping**



- Could be called "unpermute" or "walk-left" algorithm

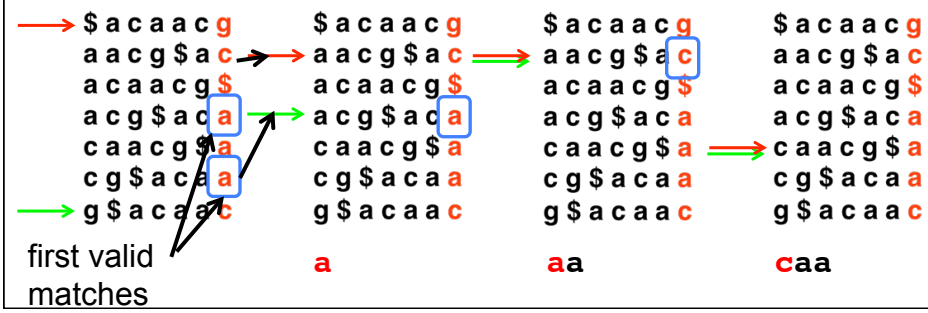
FM Index

- Ferragina & Manzini propose "FM Index" based on BWT
- Observed:
  - **LF Mapping also allows *exact matching* within T**
  - **LF(i) can be made fast with *checkpointing***
  - **...and more (see FOCS paper)**

- Ferragina P, Manzini G: Opportunistic data structures with applications. *FOCS. IEEE Computer Society; 2000.*
- Ferragina P, Manzini G: An experimental study of an opportunistic index. *SIAM symposium on Discrete algorithms*. Washington, D.C.; 2001.

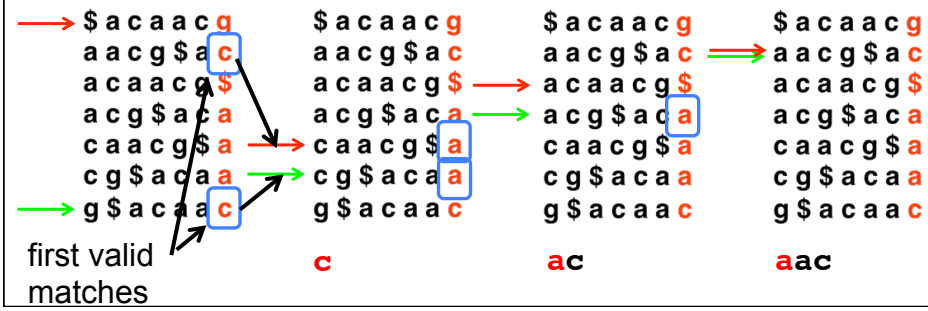
Exact Matching with FM Index

- Look up pattern in reverse.
- Use 2 pointers to represent range of matches.
  - **All matches will be next to each other in matrix.**
- Find first valid match for next symbol in range.
  - **Example: searching for "caa"**



Exact Matching with FM Index

- Look up pattern in reverse.
- Use 2 pointers to represent range of matches.
- Find first valid match for next symbol in range.
  - **Example: searching for "aac"**



Exact Matching with FM Index

- If no match…
  - **Example: searching for "gac"**

```
$acaacg    $acaacg    $acaacg    $acaacg
aacg$ac    aacg$ac    aacg$ac    aacg$ac
acaacg$    acaacg$    acaacg$ ?  acaacg$
acg$aca    acg$aca    acg$aca    acg$aca
caacg$a    caacg$a    caacg$a    caacg$a
cg$acaa    cg$acaa    cg$acaa    cg$acaa
g$acaac    g$acaac    g$acaac    g$acaac

           c          ac         gac
```

- Pointers will get lost.
- FM index can quickly check for a match.

Where in sequence is the match?

- Use "walk-left" to build sequence to start
- Count number of sequences
  - **Example: searching for "aac"**

```
$acaacg    $acaacg    $acaacg    $acaacg
aacg$ac    aacg$ac    aacg$ac    aacg$ac
acaacg$    acaacg$    acaacg$    acaacg$
acg$aca    acg$aca    acg$aca    acg$aca
caacg$a    caacg$a    caacg$a    caacg$a
cg$acaa    cg$acaa    cg$acaa    cg$acaa
g$acaac    g$acaac    g$acaac    g$acaac

aac        caac       acaac      $acaac
           1 move     2 moves    end!
```

- Number of moves back is start position of match
  - **Example: "aac" is in position 2.**

Where in sequence is match?

- "walk-left" to start of sequence is slow
- Requires on average N/2 steps to reach start.
- Alternate strategy: keep index of positions.
  - **Example: searching for "aac"**

$ a c a a c g	6
a a c g $ a c	2
a c a a c g $	0
a c g $ a c a	3
c a a c g $ a	1
c g $ a c a a	4
g $ a c a a c	5

aac

- Problem: requires as much storage as hashtable!

Where in sequence is match?

- Key Idea: Store fraction of array (sampling)
- Only store some positions and "walk-left"
- Combines two previous strategies
  - **Example: searching for "aac"**

$ a c a a c g	6
a a c g $ a c	
a c a a c g $	
a c g $ a c a	
c a a c g $ a	1
c g $ a c a a	
g $ a c a a c	

aac

$ a c a a c g	6
a a c g $ a c	
a c a a c g $	
a c g $ a c a	
c a a c g $ a	1
c g $ a c a a	
g $ a c a a c	

caac

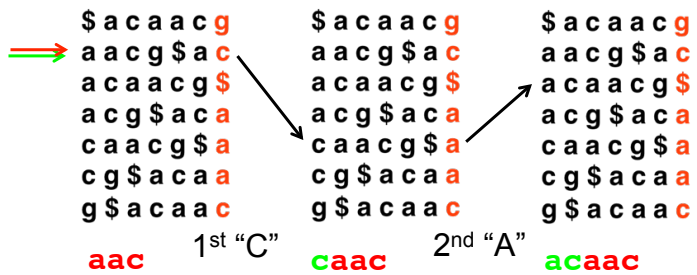
Position = number of moves + position in array

For "aac" = 1 + 1 = 2

- How many values to store provides defines time/space tradeoff.

"walk-left" optimization

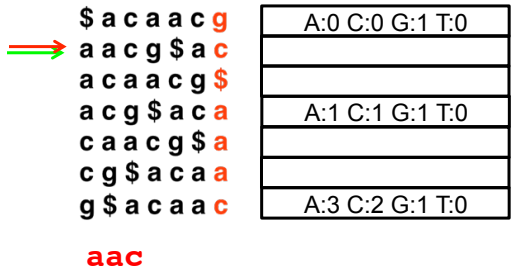
- Each "walk-left" requires counting previous occurrences of symbol in BWT
  - **Example: searching for "aac"**



- Requires counting occurrences in N/2 length string
- Really slow!

"walk-left" optimization

- Idea: use checkpoints to store previous counts
  - **Example: searching for "aac"**



- Requires counting occurrences only until checkpoint.
- Really fast!

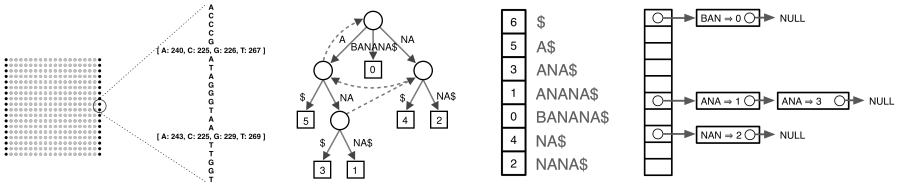
FM Index is Small (Bowtie)

- Entire FM Index on DNA reference consists of:
  - **BWT (same size as T)**
  - **Checkpoints (~15% size of T)**
  - **SA sample (~50% size of T)**
- Total: ~1.65x the size of T

Assuming 2-bit-per-base encoding and no compression, as in Bowtie

Assuming a 16-byte checkpoint every 448 characters, as in Bowtie

Assuming Bowtie defaults for suffix-array sampling rate, etc



~1.65x >45x >15x >15x

Reference Paper

- Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology 10:R25.
  - (Some slides from paper)

Dynamic programming

Coin change problem

Input: a number P (price)

Output: a way of paying P using bills ($5, $1) or coins (1, 5, 10, 25 cents)

Objective: minimize the number of coins used (greedy approach)

8.37 = 5 + 1 + 1 + 1 + 0.25 + 0.10 + 0.01 + 0.01

8 bills/coins used

The Change Problem

<u>Goal</u>: Convert some amount of money ***M*** into given denominations, using the fewest possible number of coins

<u>Input</u>: An amount of money ***M***, and an array of ***d*** denominations $\boldsymbol{c} = (c_1, c_2, \ldots, c_d)$, in a decreasing order of value ($c_1 > c_2 > \ldots > c_d$)

<u>Output</u>: A list of d integers $i_1, i_2, \ldots, i_d$ such that

$$c_1 i_1 + c_2 i_2 + \ldots + c_d i_d = \boldsymbol{M}$$

and $i_1 + i_2 + \ldots + i_d$ is minimal

Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

	Value	1	2	3	4	5	6	7	8	9	10
Min # of coins		**1**		**1**		**1**					

Initialization: only one coin is needed to make change for the values 1, 3, and 5

Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

	Value	1	2	3	4	5	6	7	8	9	10
	Min # of coins	**1**	**2**	**1**	**2**	**1**	**2**		**2**		**2**

However, two coins are needed to make change for the values 2, 4, 6, 8, and 10.

Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

	Value	1	2	3	4	5	6	7	8	9	10
	Min # of coins	**1**	**2**	**1**	**2**	**1**	**2**	**3**	**2**	**3**	**2**

Lastly, three coins are needed to make change for the values 7 and 9

Change Problem: Recurrence

This example is expressed by the following recurrence relation:

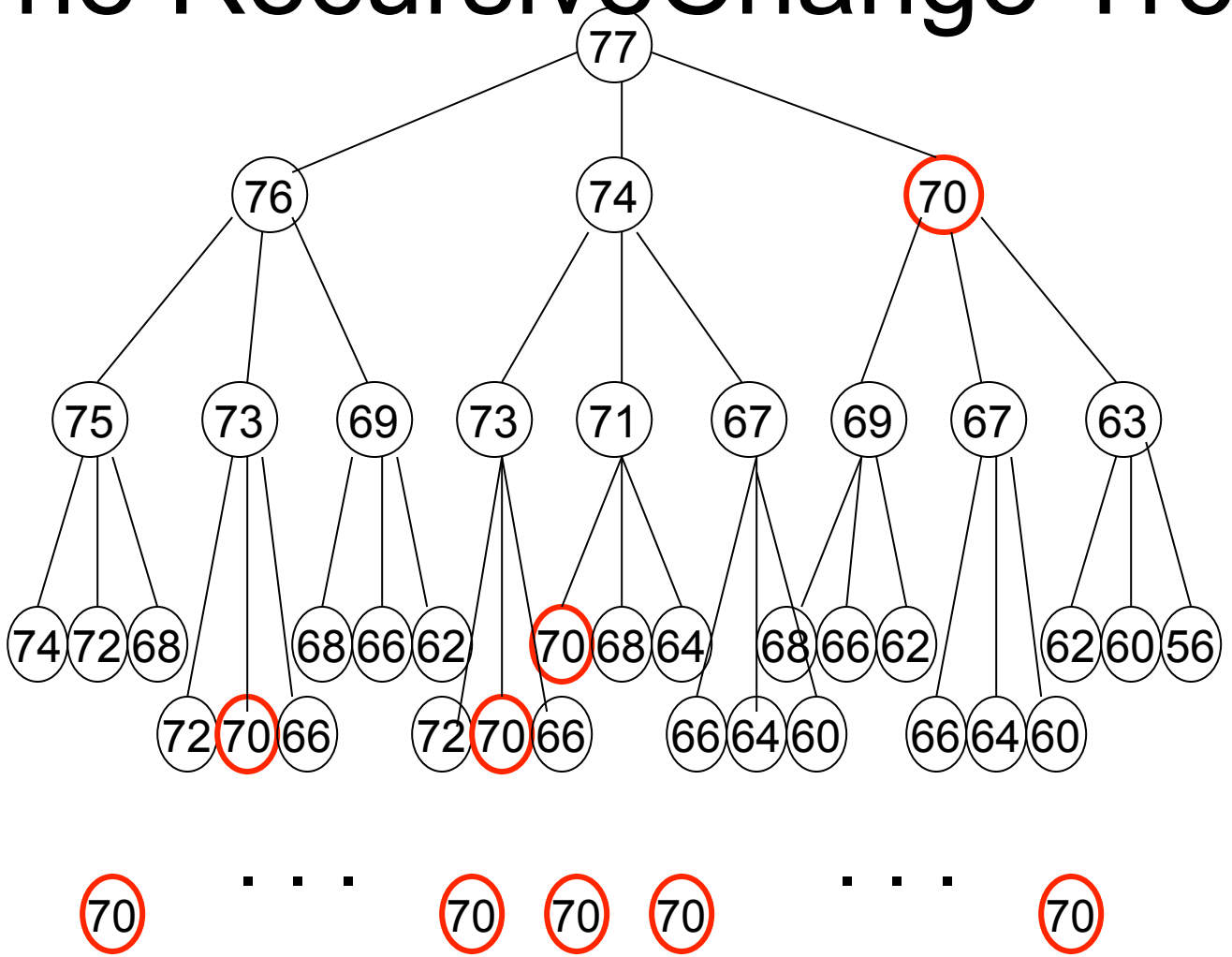
$$minNumCoins(M) = \mathbf{min} \begin{cases} minNumCoins(M-1) + 1 \\ minNumCoins(M-3) + 1 \\ minNumCoins(M-5) + 1 \end{cases}$$

Change Problem: Recurrence
(cont'd)

Given the denominations ***c***: c_1, c_2, …, c_d, the recurrence relation is:

$$minNumCoins(M) = \mathbf{min} \begin{cases} minNumCoins(M-c_1) + 1 \\ minNumCoins(M-c_2) + 1 \\ \dots \\ minNumCoins(M-c_d) + 1 \end{cases}$$

The RecursiveChange Tree



We can do better

- We're re-computing values in our algorithm more than once
- Save results of each computation for 0 to ***M***
- This way, we can do a reference call to find an already computed value, instead of re-computing each time
- Running time ***M********d***, where ***M*** is the value of money and ***d*** is the number of denominations

The change problem: dynamic programming

```
DPChange(M,c,d)
  bestNumCoins_0 ← 0
  for m ← 1 to M
    bestNumCoins_m ← infinity
    for i ← 1 to d
      if m ≥ c_i
        if bestNumCoins_(m - c_i) + 1 < bestNumCoins_m
          bestNumCoins_m ← bestNumCoins_(m - c_i) + 1
  return bestNumCoins_M
```

DPChange: example

0	0

0	1	
0	1	

0	1	2	
0	1	2	

0	1	2	3	
0	1	2	1	

0	1	2	3	4	
0	1	2	1	2	

0	1	2	3	4	5	
0	1	2	1	2	3	

0	1	2	3	4	5	6
0	1	2	1	2	3	2

0	1	2	3	4	5	6	7
0	1	2	1	2	3	2	1

0	1	2	3	4	5	6	7	8
0	1	2	1	2	3	2	1	2

0	1	2	3	4	5	6	7	8	9
0	1	2	1	2	3	2	1	2	3

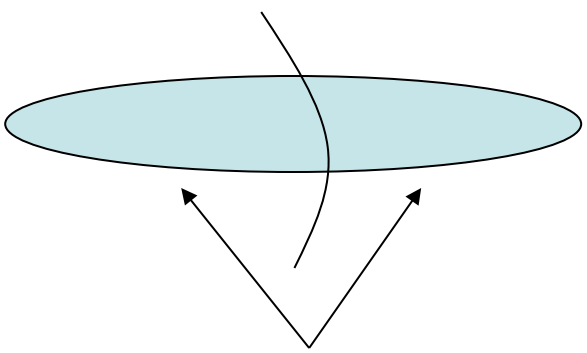
$\boldsymbol{c} = (1,3,7)$

$\boldsymbol{M} = 9$

DP: two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic-programming solution:

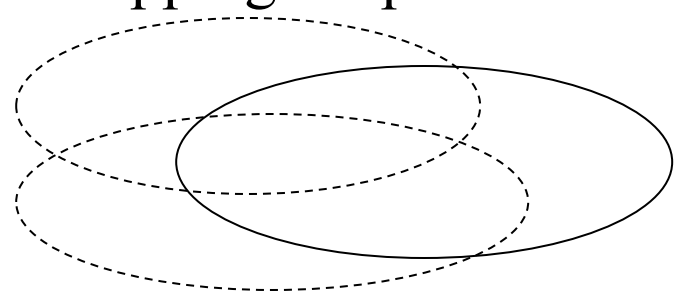
1. optimal substructures



Each substructure is optimal.

(Principle of optimality)

2. overlapping subproblems



Subproblems are dependent.

(otherwise, a divide-and-conquer approach is the choice.)

Three steps

- A DP algorithm has three basic steps:
  - The recurrence relation (for defining the value of an optimal solution);
  - The tabular computation (for computing the value of an optimal solution, while memoizing and avoiding recomputation);
  - The traceback (for delivering an optimal solution).

DPChange: example

0	0

0	1	
0	1	

0	1	2	
0	1	2	

0	1	2	3	
0	1	2	1	

0	1	2	3	4	
0	1	2	1	2	

0	1	2	3	4	5	
0	1	2	1	2	3	

0	1	2	3	4	5	6
0	1	2	1	2	3	2

0	1	2	3	4	5	6	7
0	1	2	1	2	3	2	1

0	1	2	3	4	5	6	7	8
0	1	2	1	2	3	2	1	2

0	1	2	3	4	5	6	7	8	9
0	1	2	1	2	3	2	1	2	3

$\boldsymbol{c} = (1,3,7)$

$\boldsymbol{M} = 9$

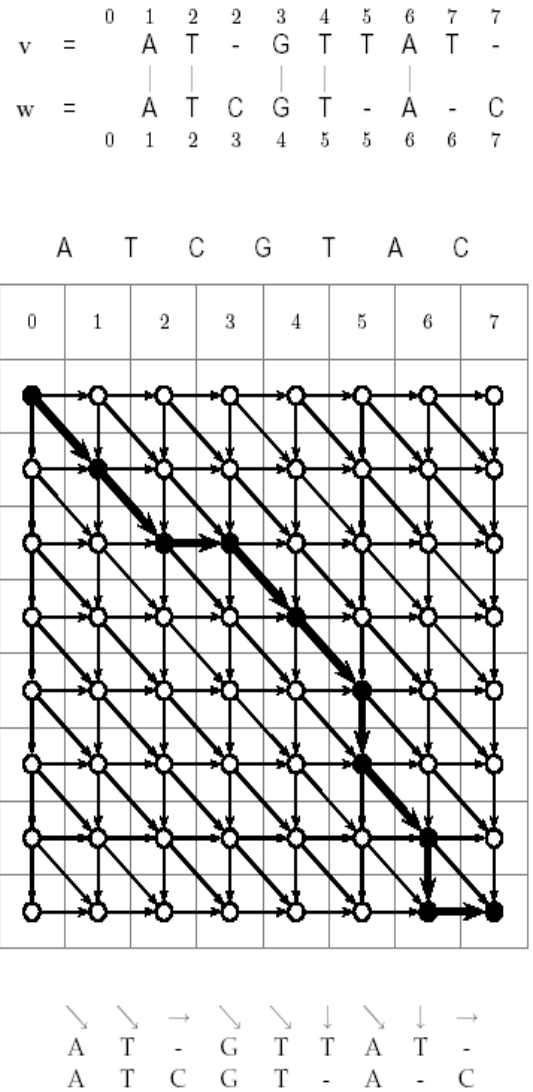
Aligning sequences with insertions and deletions

TGCATAT → ATCCGAT in 4 steps

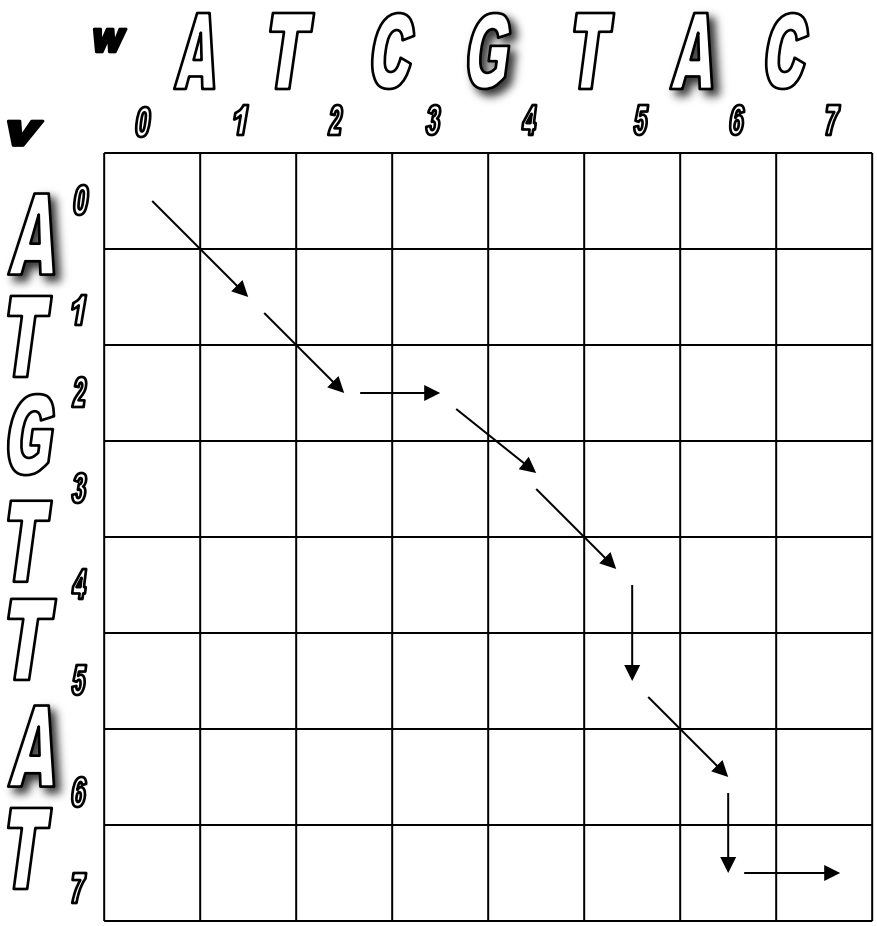
TGCATAT → (insert A at front)
ATGCATAT → (delete 6th T)
ATGCATA → (substitute G for 5th A)
ATGCGTA → (substitute C for 3rd G)
ATCCGAT (Done)

The alignment graph

– Every alignment path is from source to sink



Alignment as a path in the alignment graph

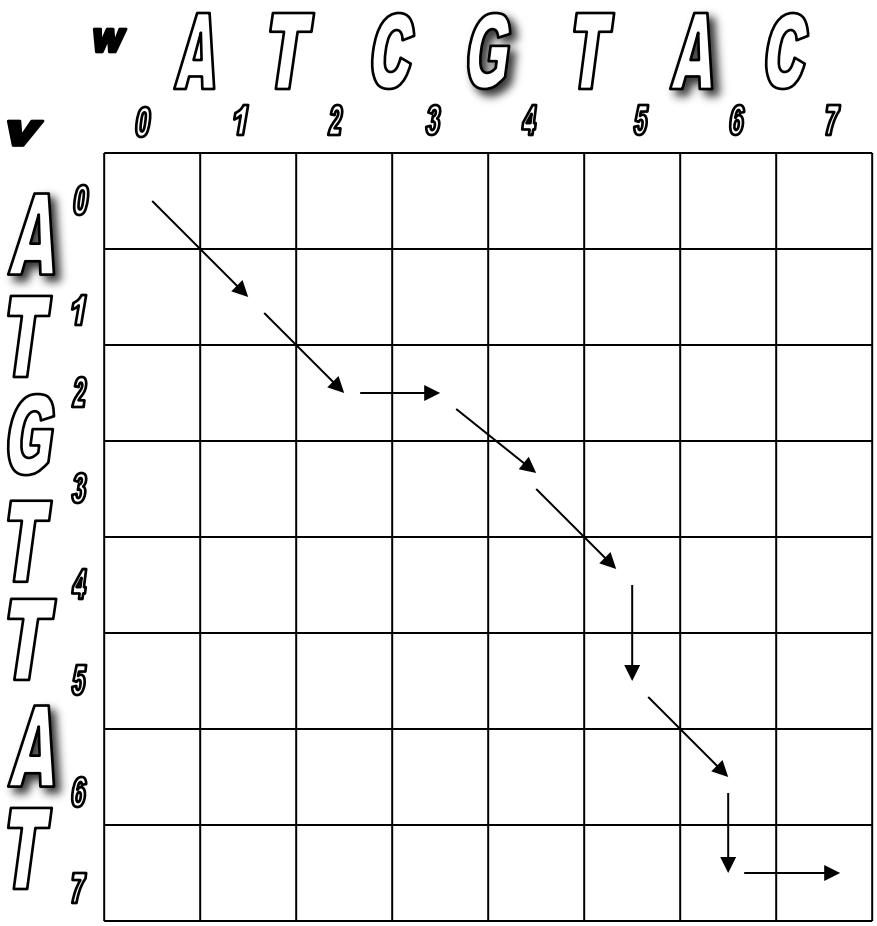


0	1	2	2	3	4	5	6	7	7
	A	T	_	G	T	T	A	T	_
	A	T	C	G	T	_	A	_	C
0	1	2	3	4	5	5	6	6	7

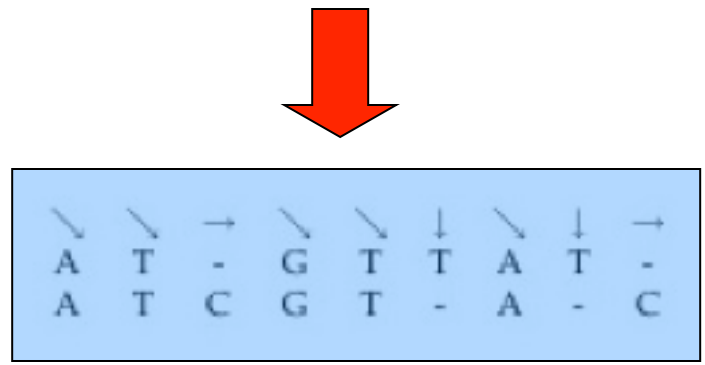
- Corresponding path -

(0,0) , (1,1) , (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7)

Alignment as a Path in the Edit Graph



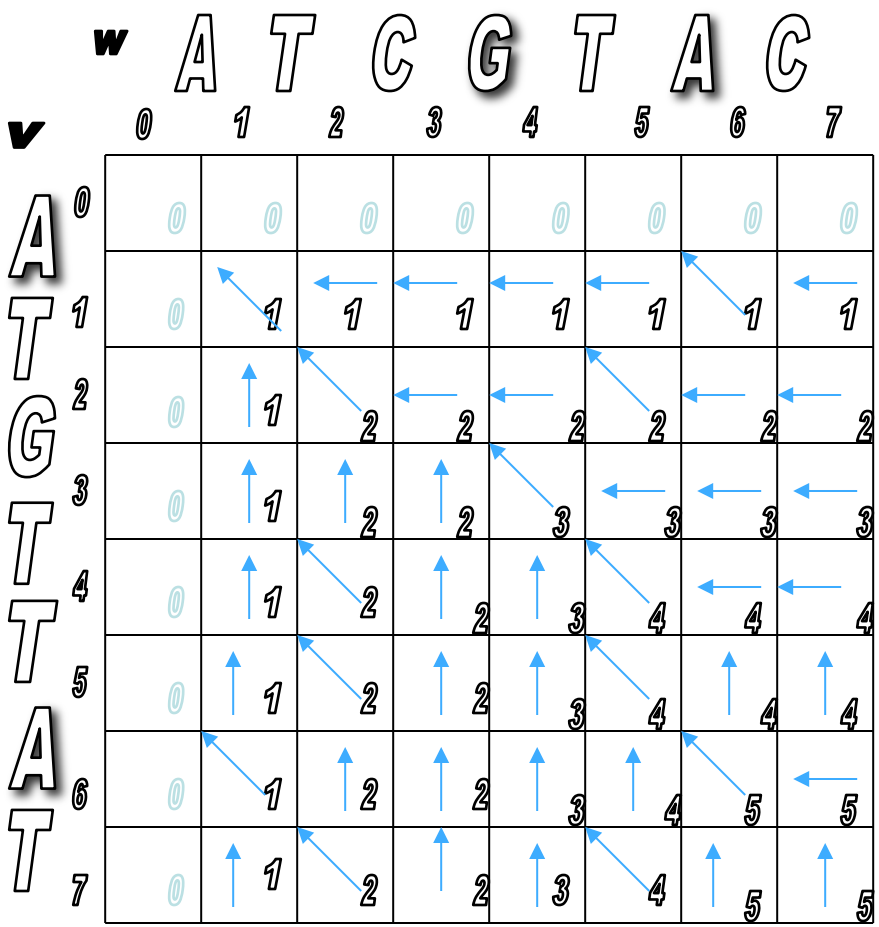
Every path in the edit graph corresponds to an alignment:



Alignment: Dynamic Programming

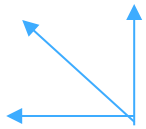
$$s_{i,j} = \max \begin{cases} s_{i-1,\, j-1} + 1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} & \downarrow \\ s_{i,\, j-1} & \rightarrow \end{cases}$$

Backtracking Example



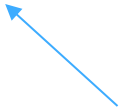
Continuing with the dynamic programming algorithm gives this result.

Alignment: Backtracking

Arrows  show where the score originated from.

if from the top

if from the left

 if $v_i = w_j$

Dynamic programming: why it is working?

$$s_{i,j} = \min \begin{cases} s_{i-1,\,j-1} \text{ if } v_i = w_j & \searrow \\ s_{i-1,\,j} + 1 & \downarrow \\ s_{i,\,j-1} + 1 & \rightarrow \end{cases}$$

Longest common subsequence (LCS) – Alignment without mismatches

- Given two sequences

$$\boldsymbol{v} = v_1\, v_2 \ldots v_m \text{ and } \boldsymbol{w} = w_1\, w_2 \ldots w_n$$

- The LCS of ***v*** and ***w*** is a sequence of positions in

$$\boldsymbol{v}\text{: } 1 \leq i_1 < i_2 < \ldots < i_t \leq m$$

and a sequence of positions in

$$\boldsymbol{w}\text{: } 1 \leq j_1 < j_2 < \ldots < j_t \leq n$$

such that i_t *-th* letter *of* ***v*** *equals to* j_t*-letter of* ***w*** and ***t*** is maximal

Dynamic programming

$$s_{i,j} = \max \begin{cases} s_{i-1,\, j-1} + 1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} & \downarrow \\ s_{i,\, j-1} & \rightarrow \end{cases}$$

From edit distance to sequence alignment

- The Edit Distance problem (or LCS) represents the simplest form of sequence alignment – allows only simple penalty for insertions, deletions and mismatches.
- In the LCS Problem, we scored 1 for matches and 0 for indels and mismatches
- For Edit distances, we scored 0 for matches and 1 for indels and mismatches
- Consider penalizing indels and mismatches with negative scores

Scoring Matrices

To generalize scoring, consider a (4+1) x(4+1) **scoring matrix** δ.

In the case of an amino acid sequence alignment, the scoring matrix would be a (20+1)x(20+1) size. The addition of 1 is to include the score for comparison of a gap character "-".

This will simplify the algorithm as follows:

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Scoring Indels: Naive Approach

- A fixed penalty σ is given to every indel:
  - $-\sigma$ for 1 indel,
  - -2σ for 2 consecutive indels
  - -3σ for 3 consecutive indels, etc.

Can be too severe penalty for a series of 100 consecutive indels

Affine Gap Penalties

- In nature, a series of *k* indels often come as a single event rather than a series of *k* single nucleotide events:

$$ATA___GC \qquad\qquad ATAG_GC$$

$$ATATTGC \qquad\qquad AT_GTGC$$

This is more likely.

Normal scoring would give the same score for both alignments

This is less likely.

Accounting for Gaps

- *Gaps*- contiguous sequence of spaces in one of the rows

- Score for a gap of length *x* is:

$$\text{gap}(x) = -(\rho + \sigma x)$$

where $\rho > 0$ is the penalty for introducing a gap:

gap opening penalty

ρ will be large relative to σ:

gap extension penalty

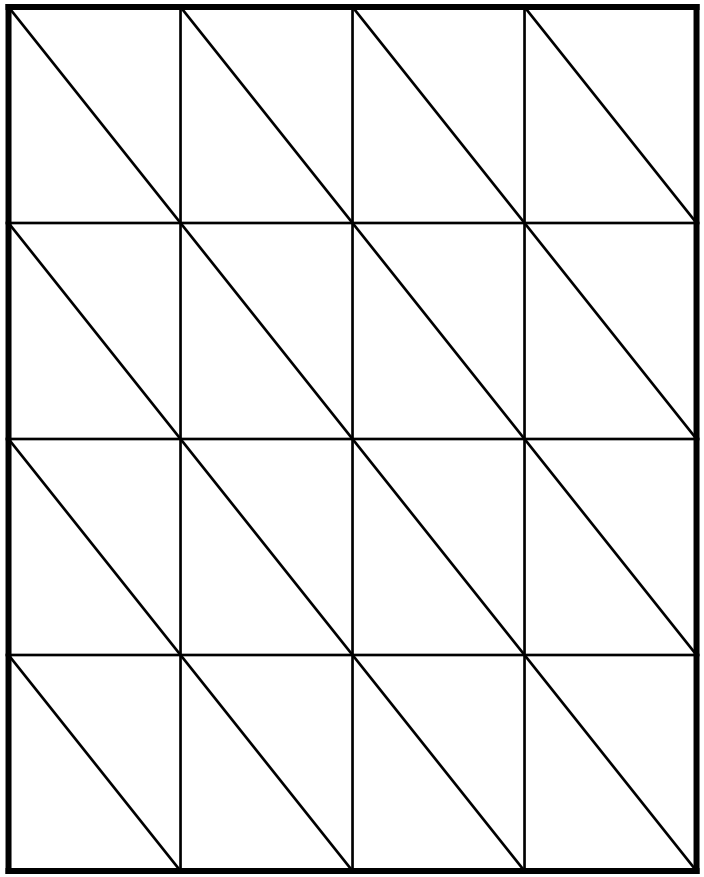
because you do not want to add too much of a penalty for extending the gap.

Can we still use the same recursion?

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

$$\delta(v_i, -) = \delta(-, w_j) = \rho + \sigma \ ?$$

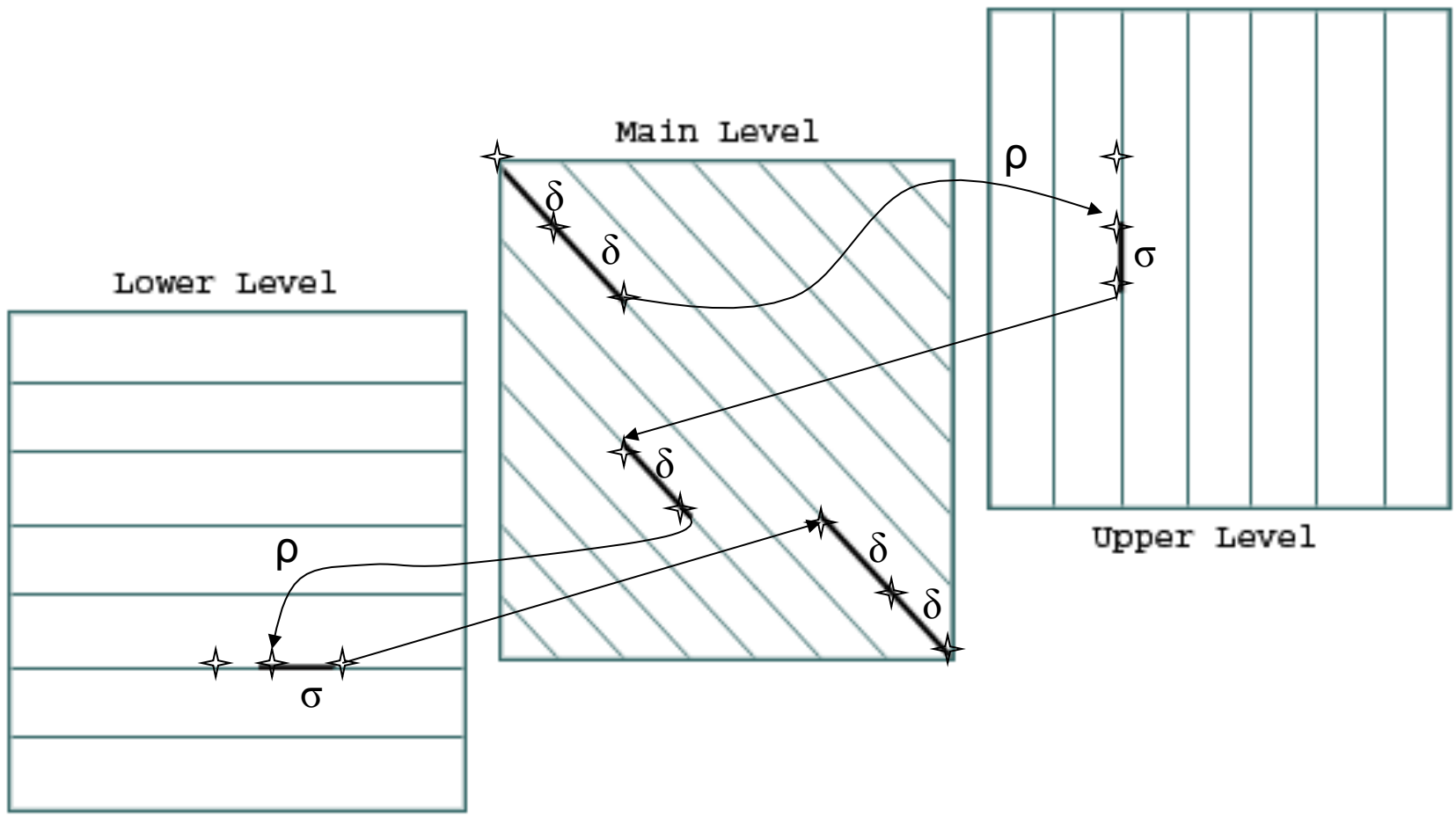
Affine gap penalties and alignment graph



To reflect affine gap penalties we have to add “long” horizontal and vertical edges to the edit graph. Each such edge of length *x* should have weight

$$-\rho - x * \sigma$$

Alignment graph in 3 Layers



Affine Gap Penalty Recurrences

$$\overset{\downarrow}{s}_{i,j} = \max \begin{cases} \overset{\downarrow}{s}_{i-1,j} - \sigma \\ s_{i-1,j} - (\rho+\sigma) \end{cases}$$

Continue Gap in *w* (deletion)

Start Gap in *w* (deletion): from middle

$$\vec{s}_{i,j} = \max \begin{cases} \vec{s}_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho+\sigma) \end{cases}$$

Continue Gap in *v* (insertion)

Start Gap in *v* (insertion):from middle

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \overset{\downarrow}{s}_{i,j} \\ \vec{s}_{i,j} \end{cases}$$

Match or Mismatch

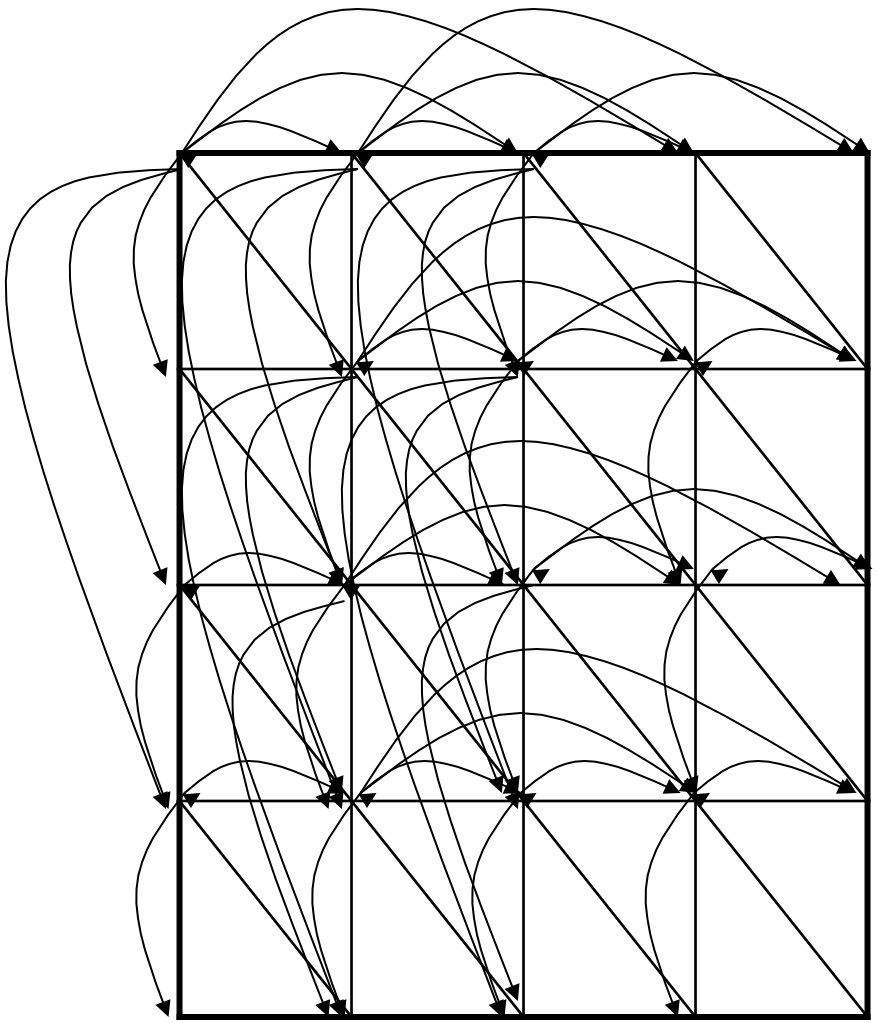
End deletion: from top

End insertion: from bottom

Arbitrary gap penalty

- Score for a gap of length *x* by gap(x)

“gap penalty” edges to the alignment graph



There are many such edges!

So the complexity increases from $O(n^2)$ to $O(n^3)$

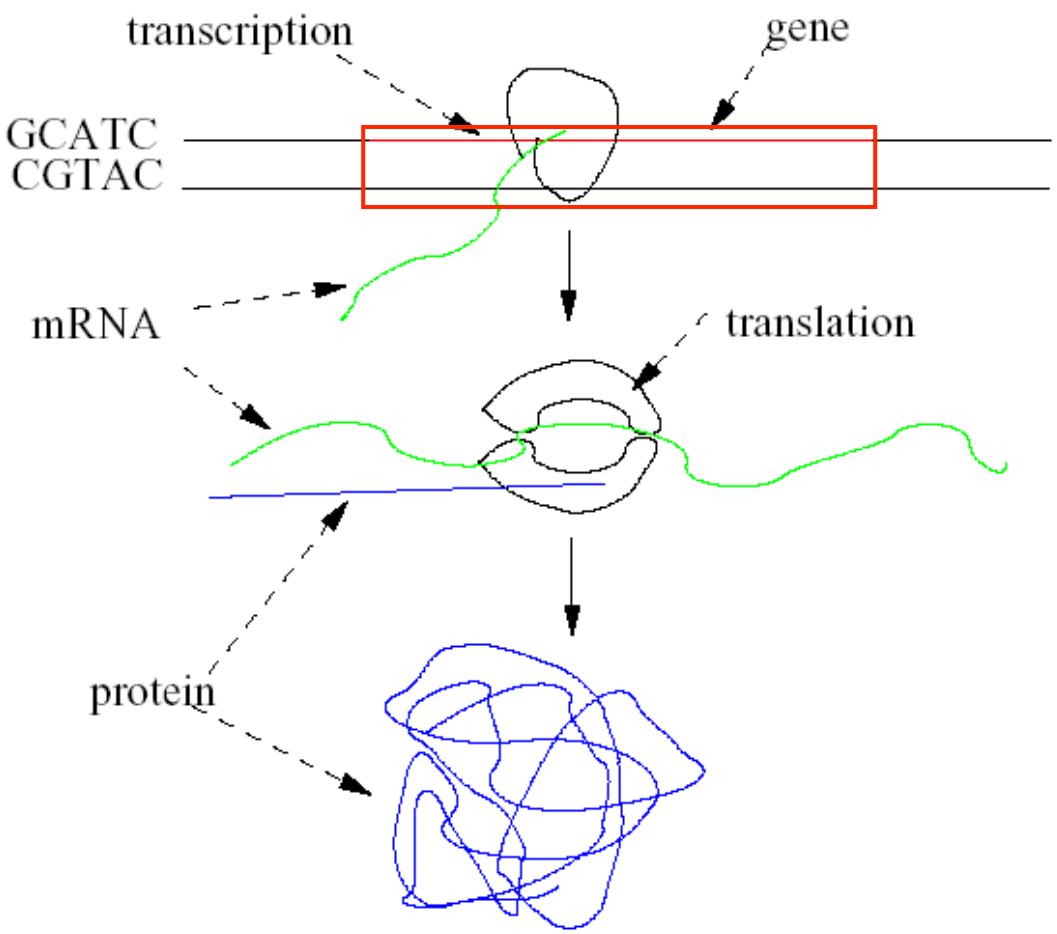
Take-home Problem: the number of alignments

- Let A(n,m) = the number of alignments of one sequence with length i and one sequence with length j.
- A(n,m)=A(n-1,m)+A(n-1,m-1)+A(n,m-1).
- $$A(m,n) = \sum_{k=0}^{\min(m,n)} \frac{(m+n-k)!}{k!(m-k)!(n-k)!}$$

		n										
		0	1	2	3	4	5	6	7	8	9	10
m	0	1										
	1	1	3									
	2	1	5	13								
	3	1	7	25	63							
	4	1	9	41	129	321						
	5	1	11	61	231	681	1683					
	6	1	13	85	377	1289	3653	8989				
	7	1	15	113	575	2241	7183	19825	48639			
	8	1	17	145	833	3649	13073	40081	108545	265729		
	9	1	19	181	1159	5641	22363	75517	224143	598417	1462563	
	10	1	21	221	1561	8361	36365	134245	433905	1256465	3317445	8097453

Non-coding RNA gene finding problems

Central dogma cont'd

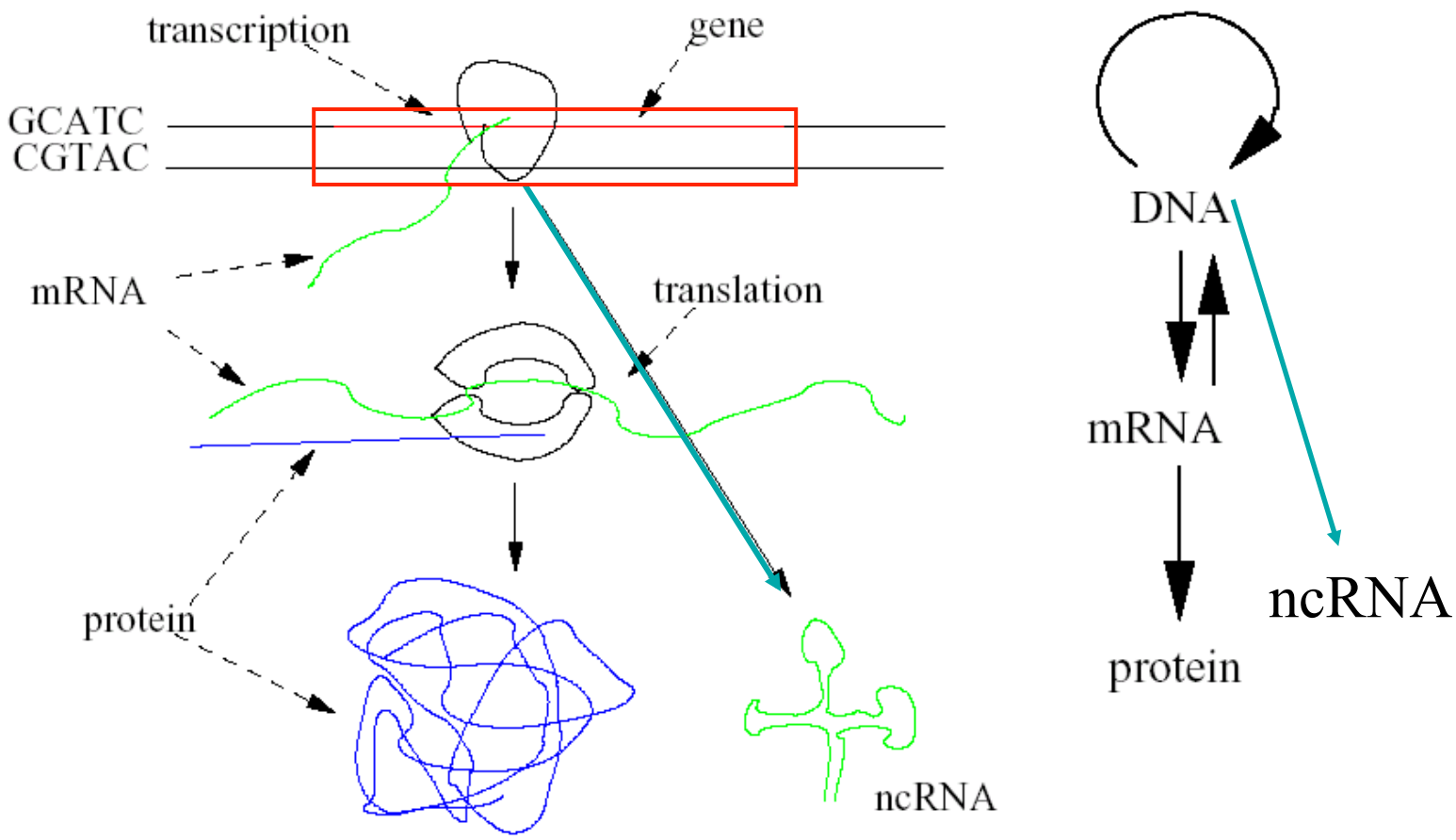


Central dogma cont'd

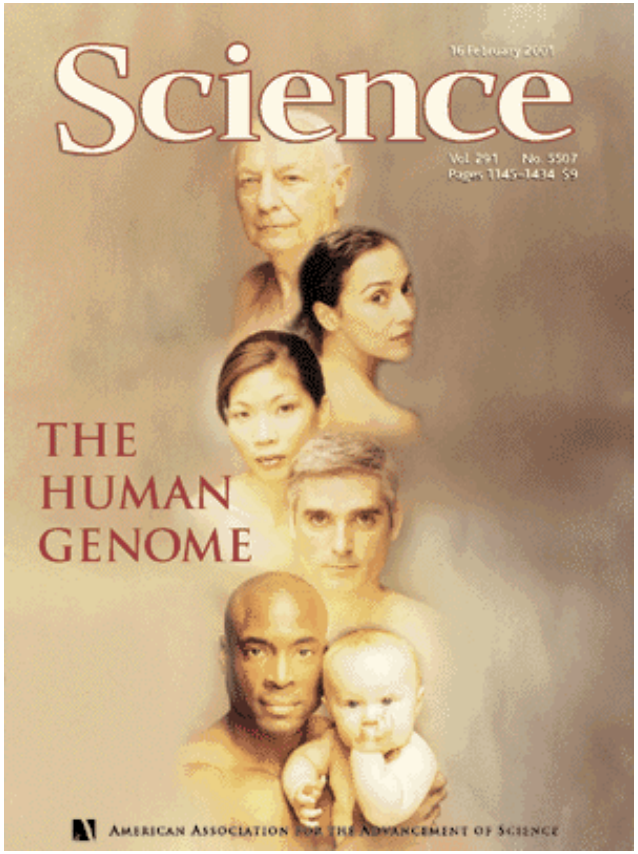
- Non-coding RNA (ncRNA)
  - RNA acting as functional molecule.
  - Not translated into protein.

- Non-coding RNA gene
  - The region of DNA coding ncRNA.

Central dogma cont'd



Human genome



How many genes do we have?

- Only about 30,000 to 40,000 protein-coding genes in the human genome [Lander et al. *Nature* (2001), Venter et al. *Science* (2001)] .

- Total protein coding gene length is only about 1.5 percent of the human genome. ($3*10^9$ bases)

What did we miss out?

- Current gene prediction methods only work well for protein coding genes.
- Non-coding RNA genes are undetected because they do not encode proteins.
- Modern RNA world hypothesis:
  - There are many unknown but functional ncRNAs. [Eddy *Nature Reviews* (2001)]
  - Many ncRNAs may play important role in the unexplained phenomenon.[Storz *Science* (2002)]

Question:

If there are many ncRNAs, what are they doing?

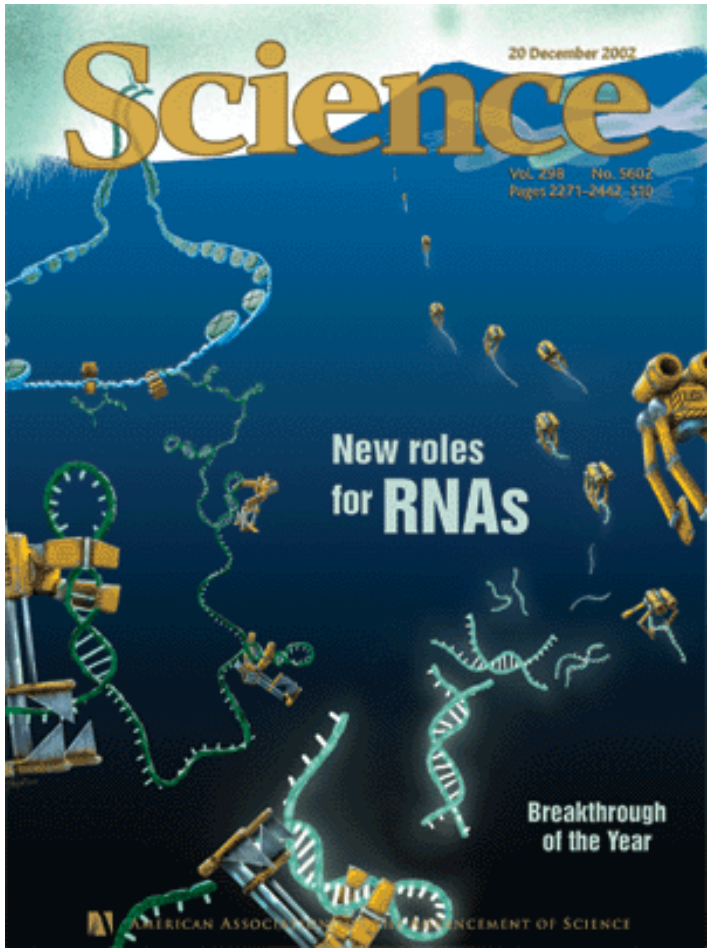
Question:

Biologically, why do we need functional ncRNAs in addition to protein?

Why do we need ncRNAs?

- ncRNAs involve sequence specific recognition of other nucleic acids (e.g. mRNAs, DNAs).
- ncRNA is an ideal material for this role.
  - DNA is big and packaged and can do this job.
- *Base complementary* allows ncRNA to be sequence specific!
- For example:
  - small interfering RNAs (siRNA) is used to protect our genome.
  - It recognizes invading foreign RNAs/DNAs based on the sequence specificity.
  - And helps to degrade the foreign RNAs.

Science
17 May 2002
Vol. 296 - No. 5571
Pages 1189–1352 $9
RNA Silencing and
Noncoding RNA
American Association for the Advancement of Science


Science
20 December 2002
Vol. 298 No. 5602
Pages 2271–2442 $10
New roles
for RNAs
Breakthrough
of the Year

How can we find such ncRNA genes in the genome?

RNA secondary structure

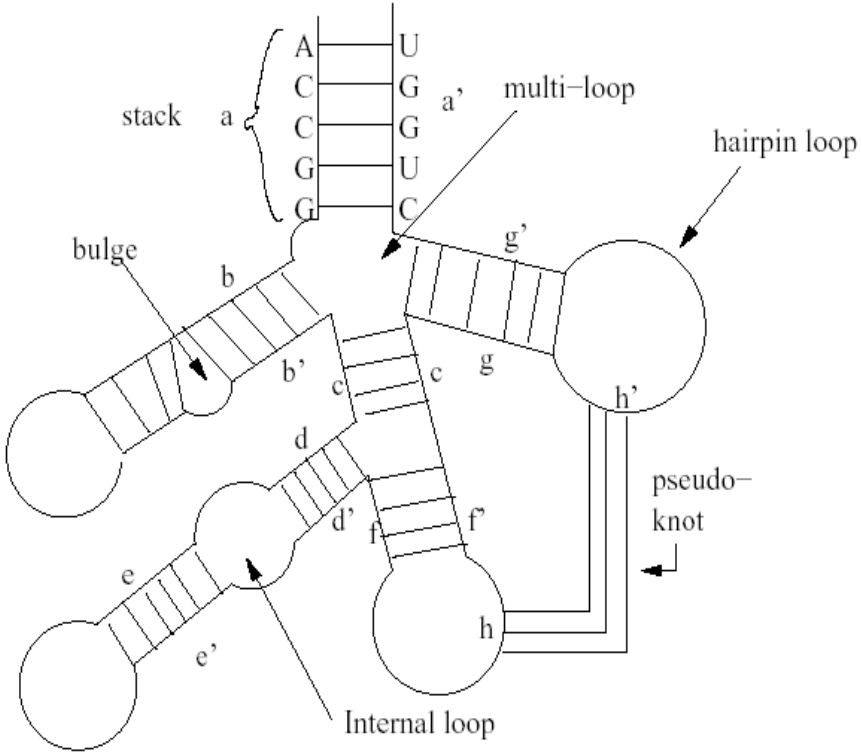
- ncRNA is not a random sequence.
- Most RNAs fold into particular base-paired secondary structure.

- Canonical basepairs:
  - Watson-Crick basepairs:
    - G - C
    - A - U
  - Wobble basepair:
    - G – U

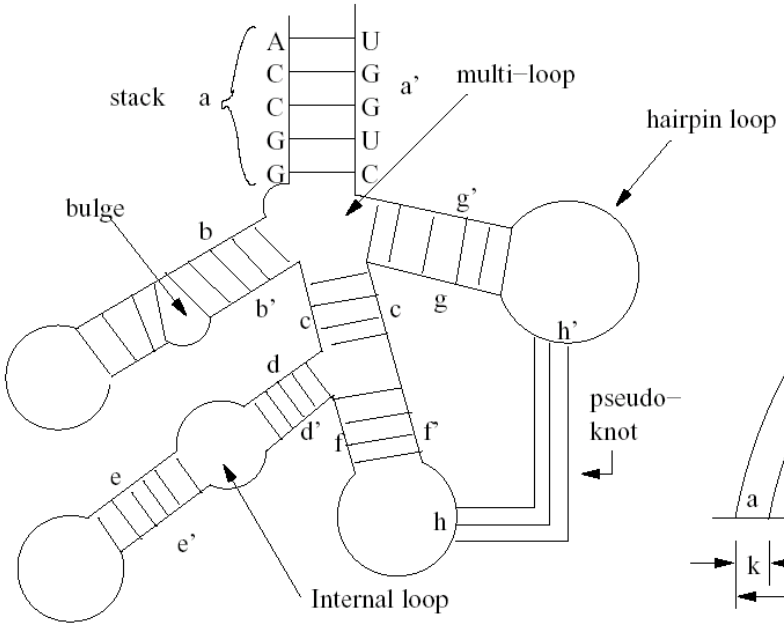
RNA secondary structure cont'd

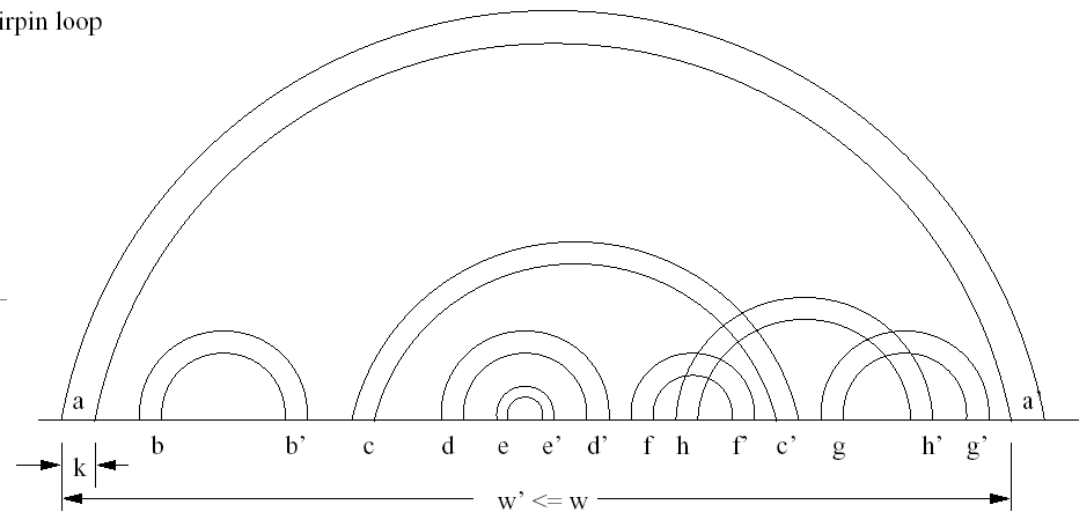
- Stacks: continuous nested basepairs. (energetically favorable)

- Non-basepaired loops:
  - Hairpin loop.
  - Bulge.
  - Internal loop.
  - Multiloop.



RNA secondary structure cont'd

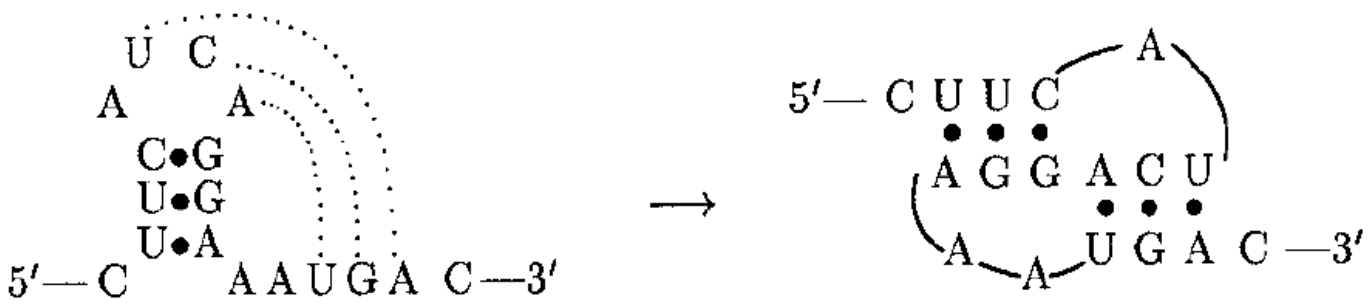




- Most basepairs are non-crossing basepairs.
  - Any two pairs (i, j) and (i', j') ➔ $i < i' < j' < j$ or $i' < i < j < j'$
- Pseudoknots are the crossing basepairs.

Pseudoknots

- Pseudoknots are important for certain ncRNAs
- Violate the non-crossing assumption.
- Pseudoknots make most problems harder
- We assume there are no pseudoknots otherwise noted.



[Rivas and Eddy (1999)]

RNA secondary structure prediction

- It is a basic issue in ncRNA analysis
- It is important information to the biologists.
- Searching and alignment algorithms are based on these models.
- RNA secondary structure -- a set of non-crossing base pairs.

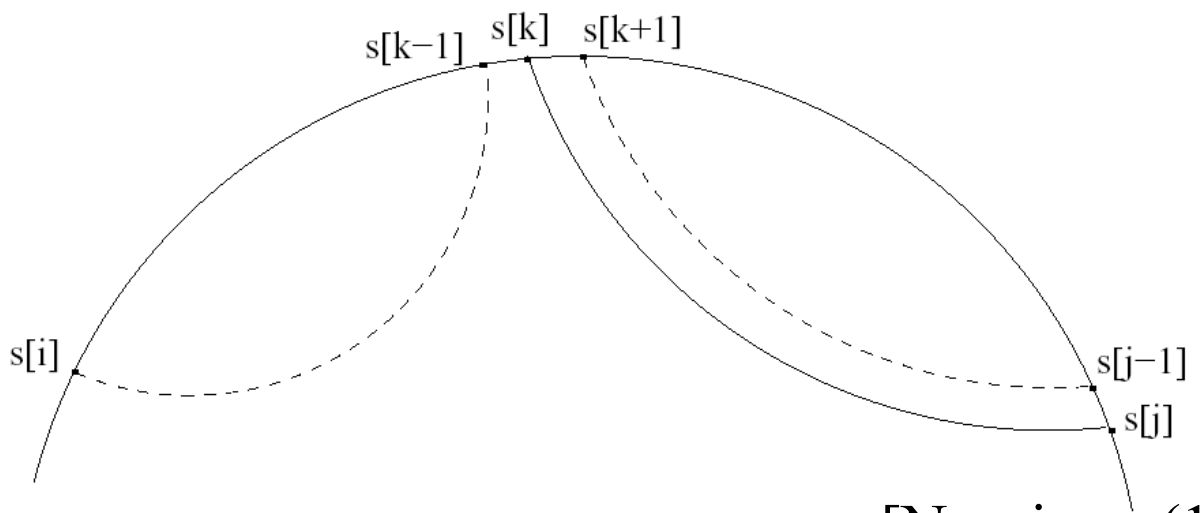
Base pair maximization problem

- A simple energy model is to maximize the number of basepairs to minimize the free energy. [Waterman (1978), Nussinov et al (1978), Waterman and Smith (1978)]
- G – C, A – U, and G – U are treated as equal stability.
- Contributions of stacking are ignored.

Problem 1: [Base pair maximization problem]
Given an RNA sequence, determine a set of base pairs in a RNA sequence such that the number of base pairs is maximal and no base pairs cross each other.

A dynamic programming solution

- Let $s[1 \ldots n]$ be an RNA sequence.
- $\delta(i,j) = 1$ if $s[i]$ and $s[j]$ form a complementary base pair, else $\delta(i,j) = 0$.
- $M(i,j)$ is the maximum number of base pairs in $s[i \ldots j]$.



[Nussinov (1980)]

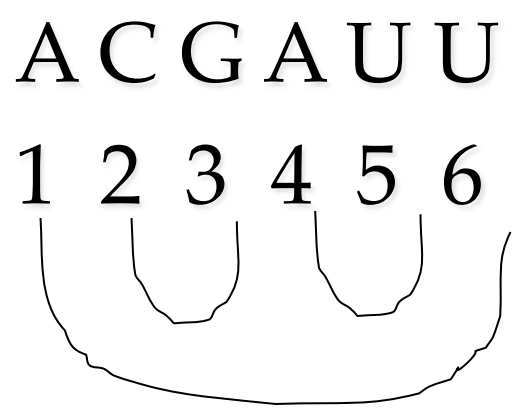
A dynamic programming solution

$$M(i,j) = max \begin{cases} M(i,j-1), \\ M(i,k-1) + M(k+1,j-1) + \delta(k,j) \\ \qquad\qquad \text{for } i \leq k < j. \end{cases}$$

- *M*(*1,n*) is the number of base pairs in the optimal basepaired structure for *s*[*1…n*].
- All these basepairs can be found by tracing back through the matrix *M*.
- Filling *M* needs $O(n^3)$ time.

RNA structure: example

j \ i	1	2	3	4	5	6
2	0					
3	1	1				
4	1	1	0			
5	2	2	1	1		
6	3	2	1	1	0	

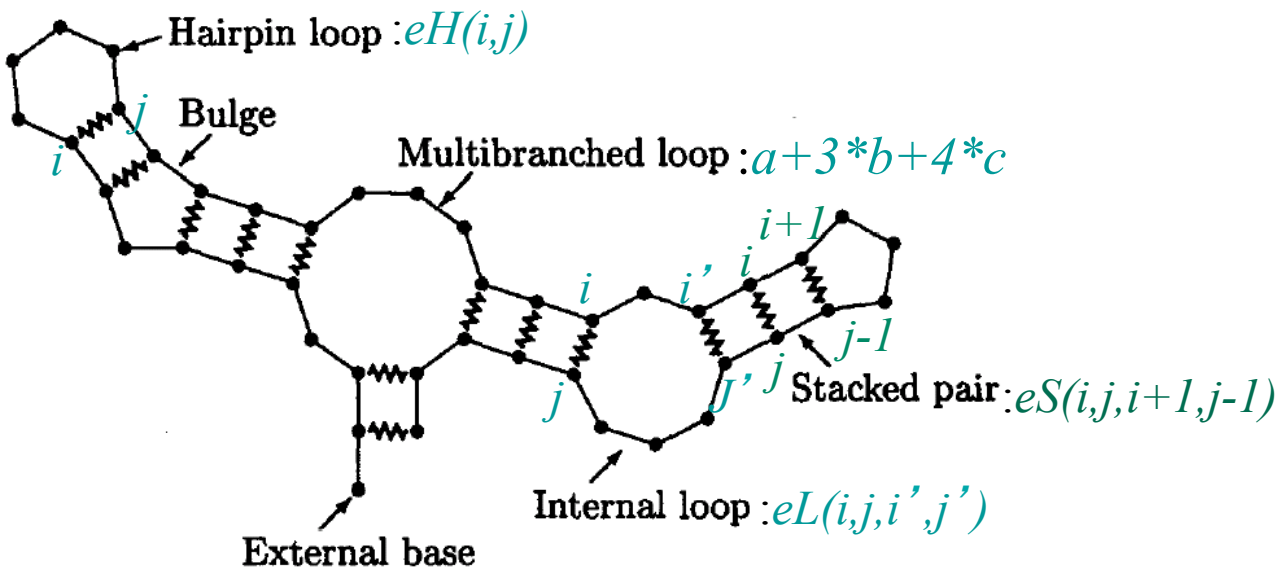


Zuker-Sankoff minimum energy model

- Stacks (contiguous nested base pairs) are the dominant stabilizing force – contribute the negative energy
- Unpaired bases form loops contribute the positive energy.
  - Hairpin loops, bulge/internal loops, and multiloops.

- Zuker-Sankoff minimum energy model. [Zuker and Sankoff (1984), Sankoff (1985)]
- `Mfold` and `ViennaRNA` are all based on this model. (this model is also called mfold model)

Zuker-Sankoff minimum energy model



[Lyngsø (1999)]

RNA minimum energy problem

Problem 2: [RNA minimum energy problem] Given an RNA sequence $s[1...n]$, and four energy functions eH, eS, eL, a, b, and c, determine a noncrossing secondary structure for this RNA sequence such that the sum of the energy over all the loops and stacks in the secondary structure is minimized.

- This problem can be solved by a dynamic programming algorithm in $O(n^4)$ time.
- Lyngsø et al. (1999) revise the energy function for internal loop, proposed an $O(n^3)$ time solution.

Zuker-Sankoff model

- **Hairpin loop:** $eH(i,j)$ is the energy of the hairpin loop from $i+1$ to $j-1$, which *closed* by base pair (i,j).

- **Stacked base pairs:** $eS(i,j,i+1,j-1)$ is the energy of the stacking base pairs (i,j) and $(i+1,j-1)$.

- **Bulge and internal loop:** $eL(i,j,i',j')$ is the energy of the the bulge or internal loop starting from $i+1$ to $i'-1$ and from $j'+1$ to $j'-1$ which is *closed* by base pairs (i,j) and (i',j').

- **Multi-loop:** a is the energy of generating a multi-loop, b is the energy of one base pair that *closes* the multi-loop, and c is the energy of one unpaired base in the multi-loop.

Recursive functions

- *W*(*i*) holds the minimum energy of a structure on *s*[*1…i*].

$$W(i) = \min\{W(i-1), \min_{0 \le k < i}\{W(k) + V(k+1, i)\}\}.$$

- *V*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] with *s*[*i*] and *s*[*j*] forming a basepair.

$$\begin{aligned} V(i,j) = \min\{ & eH(i,j), \\ & eS(i,j,i+1,j-1) + V(i+1,j-1), \\ & \min_{i<i'<j'<j \text{ and } i'-i+j-j'>2}\{eL(i,j,i',j') + V(i',j')\}, \\ & \min_{i+1<k<j}\{WM(i+1,k-1) + WM(k,j-1) + a\}\}, \end{aligned}$$

- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] that is part of multiloop.

$$\begin{aligned} WM(i,j) = \min\{ & V(i,j) + b, \\ & WM(i,j-1) + c, \\ & WM(i+1,j) + c, \\ & \min_{i<k\le j}\{WM(i,k-1) + WM(k,j)\}\}, \end{aligned}$$

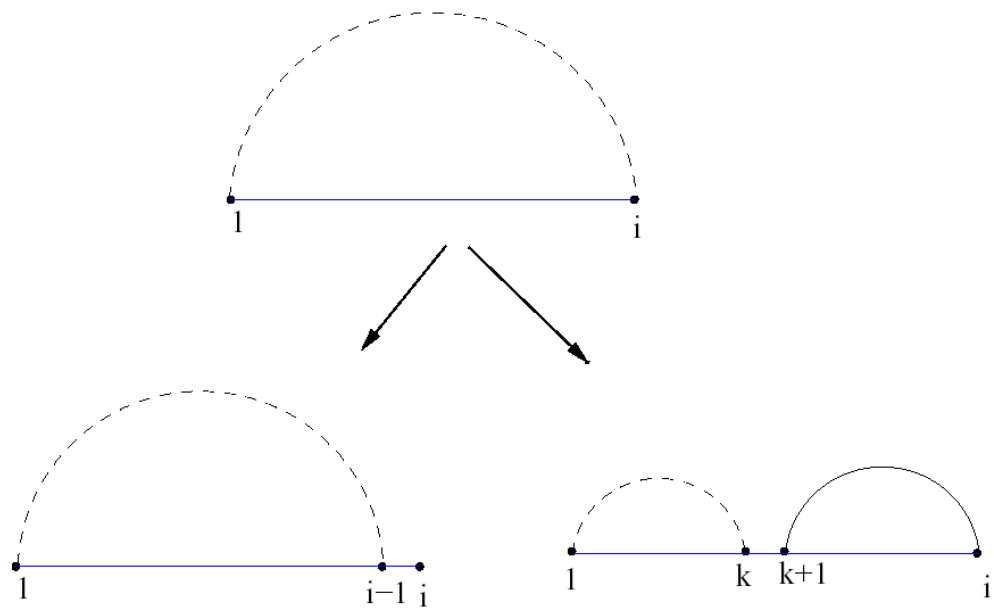
Recursive functions (Zuker)

- *W*(*i*) holds the minimum energy of a structure on *s*[*1…i*].

$$W(i) = \min\{W(i-1), \min_{0 \le k < i}\{W(k) + V(k+1, i)\}\}.$$

- *V*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] with *s*[*i*] and *s*[*j*] forming a basepair.

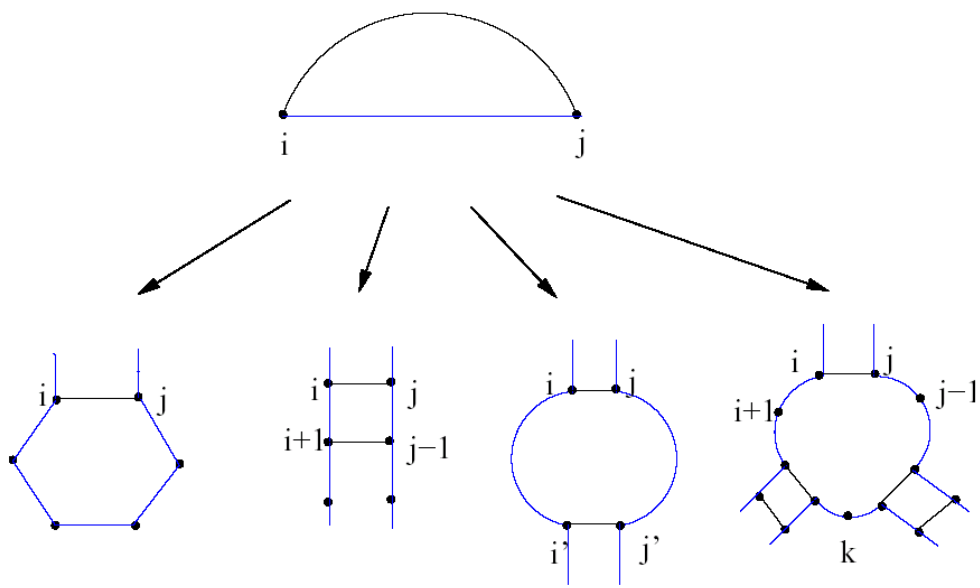
- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] that is part of multiloop.



A recursive solution

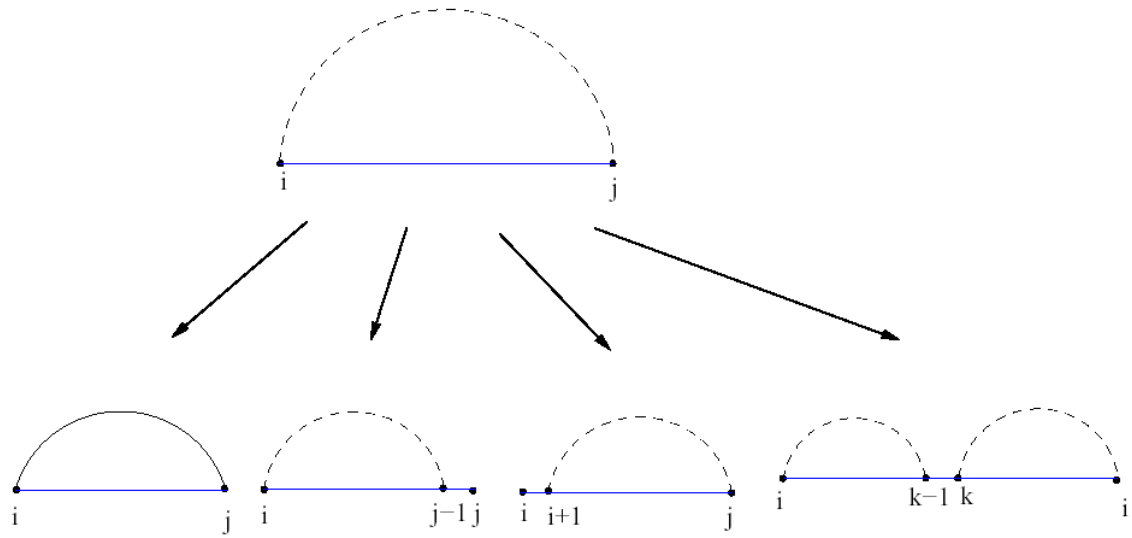
- *W*(*i*) holds the minimum energy of a structure on *s*[*1…i*].
- *V*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] with *s*[*i*] and *s*[*j*] forming a basepair.
- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] that is part of multiloop.

$$
\begin{aligned}
V(i,j) = \min\{ & eH(i,j), \\
& eS(i,j,i+1,j-1) + V(i+1,j-1), \\
& \min_{i<i'<j'<j \text{ and } i'-i+j-j'>2} \{eL(i,j,i',j') + V(i',j')\}, \\
& \min_{i+1<k<j} \{WM(i+1,k-1) + WM(k,j-1) + a\}\},
\end{aligned}
$$



A recursive solution

- *W*(*i*) holds the minimum energy of a structure on *s*[*1…i*].
- *V*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] with *s*[*i*] and *s*[*j*] forming a basepair.
- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] that is part of multiloop.



$$
\begin{aligned}
WM(i,j) = \min\{ & V(i,j) + b, \\
& WM(i,j-1) + c, \\
& WM(i+1,j) + c, \\
& \min_{i<k\leq j} \{WM(i,k-1) + WM(k,j)\}\},
\end{aligned}
$$

A recursive solution

- *W*(*i*) holds the minimum energy of a structure on *s*[*1…i*].

$$W(i) = \min\{W(i-1), \min_{0 \le k < i}\{W(k) + V(k+1, i)\}\}.$$

- *V*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] with *s*[*i*] and *s*[*j*] forming a basepair.

$$\begin{aligned} V(i,j) = \min\{ & eH(i,j), \\ & eS(i,j,i+1,j-1) + V(i+1,j-1), \\ & \min_{i<i'<j'<j \text{ and } i'-i+j-j'>2}\{eL(i,j,i',j') + V(i',j')\}, \\ & \min_{i+1<k<j}\{WM(i+1,k-1) + WM(k,j-1) + a\}\}, \end{aligned}$$

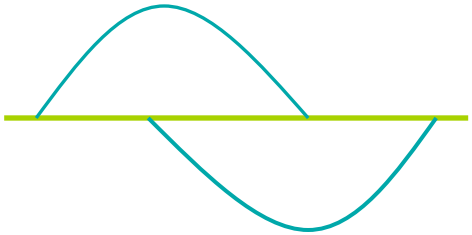
- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*I…j*] that is part of multiloop.

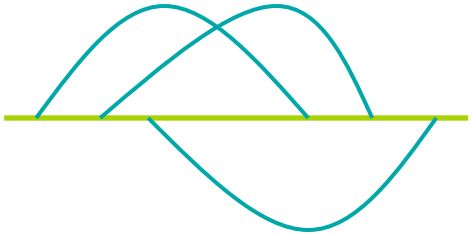
$$\begin{aligned} WM(i,j) = \min\{ & V(i,j) + b, \\ & WM(i,j-1) + c, \\ & WM(i+1,j) + c, \\ & \min_{i<k\le j}\{WM(i,k-1) + WM(k,j)\}\}, \end{aligned}$$

Prediction with pseudoknots

- Base pair maximization allowing crossing pairs can be solved in polynomial time.

- Ieong et al. (2003) proved that base pairing maximization problem allowing crossing pairs in a *planar* secondary structure is NP-hard.





Prediction with pseudoknots

- Prediction allowing generalized pseudoknots with energy functions depending on adjacent basepairs is NP-hard.
  - Akutsu (2000) (longest common subsequence for multiple sequences (LCS)).
  - Lyngsø and Pedersen (2000) (3SAT).
  - similar to Zuker-Sankoff minimum energy model.
- Pseudoknots in structure-known RNAs.
  - Biologists are not interested in the approximation solutions.
  - Most pseudoknots are planar.
  - Not too many variations.
- Rivas and Eddy (1999) presented a $O(n^6)$ solution allowing most types of pseudoknots in known ncRNAs.

RNA sequence-structure alignment

- RNA alignments:
  - sequence-sequence alignment.
  - sequence-structure alignment.
    - It's most useful for ncRNA finding.
  - structure-structure alignment.

What is an alignment?

- Alignment of two sequences

```
ACCCG-UUAAU
| ||| ||*||
A-CCGUUUCAU
```

- Sequence-structure alignment:
  - One RNA sequence with its known secondary structure.
  - One just sequence
  - Take into account both structure similarity and sequence similarity.

```
Stru1: >>>>    <<<<
Seq1:  GGGGCAACCCC
       ++++*||++++
Seq2:  AUCCGAAGGAU
```

RNA sequence-structure alignment

- Given two RNA sequence $s[1 \ldots n]$ and $t[1 \ldots m]$, and s has a known secondary structure S, where (i,j) in S implies $s[i]$ is basepaired with $s[j]$.
- Score for the alignment is the sum of scores γ of each column plus the sum of scores δ of each basepair in S.
- Scores for each column:
  - $\gamma(a,b)$ for all a, b *in* $\{A, C, G, U, -\}$.
- Scores for two basepair columns:
  - $\delta(a, b, c, d)$ for all $a, b, c,$ and d *in* $\{A, C, G, U\}$.
  - For example $\delta(a, b, c, d) = 1$ if (a,b) *in* S and c and d form basepair, otherwise $-\infty$.

Problem 3: [**RNA sequence-structure alignment problem**] Given two RNA sequence $s[1..n]$ and $t[1...m]$, where s has a known secondary structure without pseudoknots, compute the highest scoring alignment.

Analysis: alignment solutions

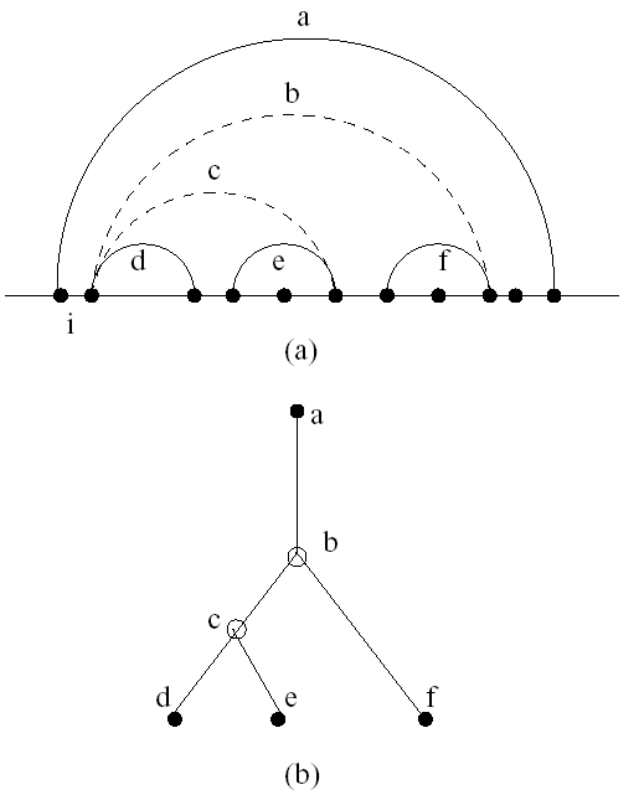
- Similar to Nussinov's base pair maximization problem's solution, the obvious dynamic programming solution is in $O(n^6)$ time.

- Bafna et al. (1995) improved the dynamic programming algorithm to a running time of $O(n^4)$ by using binarized tree for structure *S*.

- We further improve the algorithm by building a binarized tree for whole sequence $s[1 \ldots n]$ with its structure, and reduce the computation time into $O(n^3 m_1)$, where m_1 is the number of the branches in the tree (typically very small constant).

The method of Bafna et al. (1995)

- Adding spurious (dashed) edges to *S*, so that each node in *S'* has at most 2 children.
- We use the dashed edges (void nodes) to fix certain interval doing recursions, when we need find the branch points.
- It is in $O(n^2m^2+nm^3)$ time.



Binarizing an RNA structure tree (Bafan et al. 1995)

Procedure *Binarize*(i, j)
(* Assume that $(i, j) \in S$ has k children $\{(i_1, j_1), \ldots, (i_k, j_k)\}$ *)
begin
  for $1 \leq u < k$ do
    Binarize(i_u, j_u)
    $S' = S' \cup \{(i_1, j_u)\}$
    if $(u > 1)$
      parent$((i_1, j_{u-1})) = (i_1, j_u)$
      parent$((i_u, j_u)) = (i_1, j_u)$
  parent$(i_1, j_k) = (i, j)$
end

Alignment Algorithm in $O(m^2n^2+nm^3)$ (bafna et al. 1995)

Procedure *InferStructure*()
begin
for intervals (i_1, j_1), $1 \leq i_1 < j_1 \leq n$
and intervals (i_2, j_2), $1 \leq i_2 < j_2 \leq m$
(* Assume that the intervals are examined in lexicographically increasing order of widths*)

$$Align[i_1, j_1, i_2, j_2] = \max \begin{cases} Align[i_1+1, j_1, i_2, j_2] + \gamma(s[i_1], '-') \\ Align[i_1, j_1, i_2+1, j_2] + \gamma('-', t[i_2]) \\ Align[i_1+1, j_1, i_2+1, j_2] + \gamma(s[i_1], t[i_2]) \\ Align[i_1, j_1-1, i_2, j_2] + \gamma(s[j_1], '-') \\ Align[i_1, j_1, i_2, j_2-1] + \gamma('-', t[j_2]) \\ Align[i_1, j_1-1, i_2, j_2-1] + \gamma(s[j_1], t[j_2]) \end{cases}$$

if $(i_1, j_1) \in S$ and
$t[i_2]$ and $t[j_2]$ are complementary base-pairs

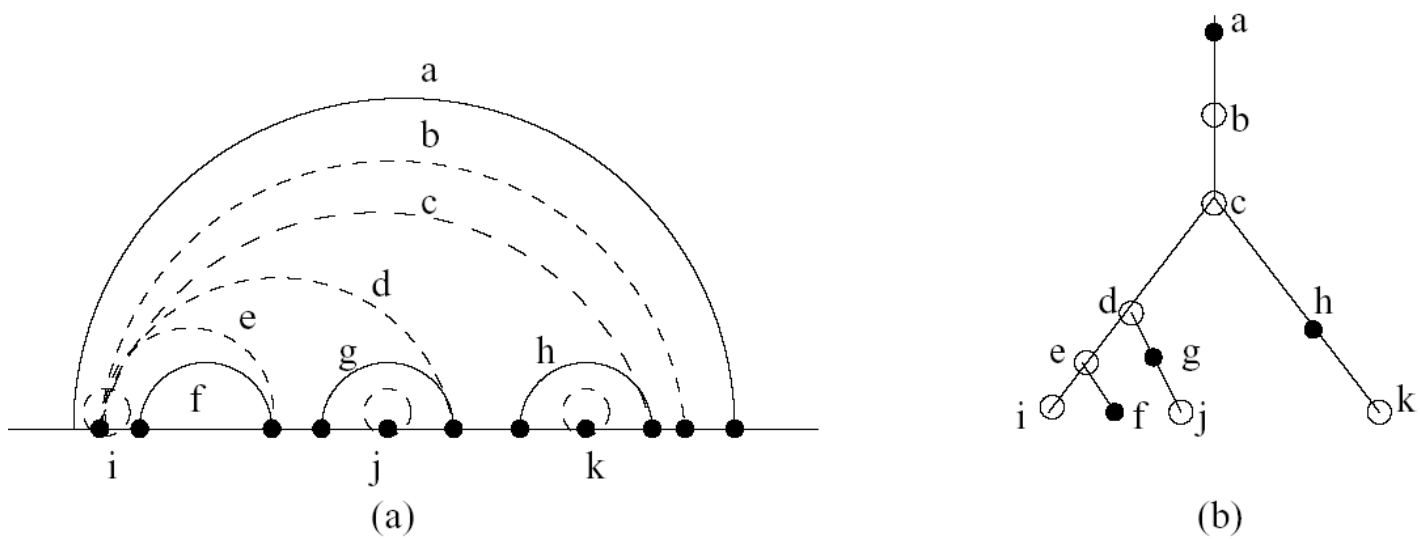
$$Align[i_1, j_1, i_2, j_2] = \max \begin{cases} Align[i_1, j_1, i_2, j_2], \\ \delta(i_1, j_1, i_2, j_2) + \gamma(s[i_1], t[i_2]) \\ +\gamma(s[j_1], t[j_2]) + Align[i_1+1, j_1-1, i_2+1, j_2-1] \end{cases}$$

else if $(i_1, j_1) \in S' - S$ and
$(k, j_1)) = rightchild(i_1, j_1)$

$$Align[i_1, j_1, i_2, j_2] = \max \begin{cases} Align[i_1, j_1, i_2, j_2], \\ \max_{i_2 < l < j_2} \{Align[i_1, k-1, i_2, l-1] + Align[k, j_1, l, j_2]\} \end{cases}$$

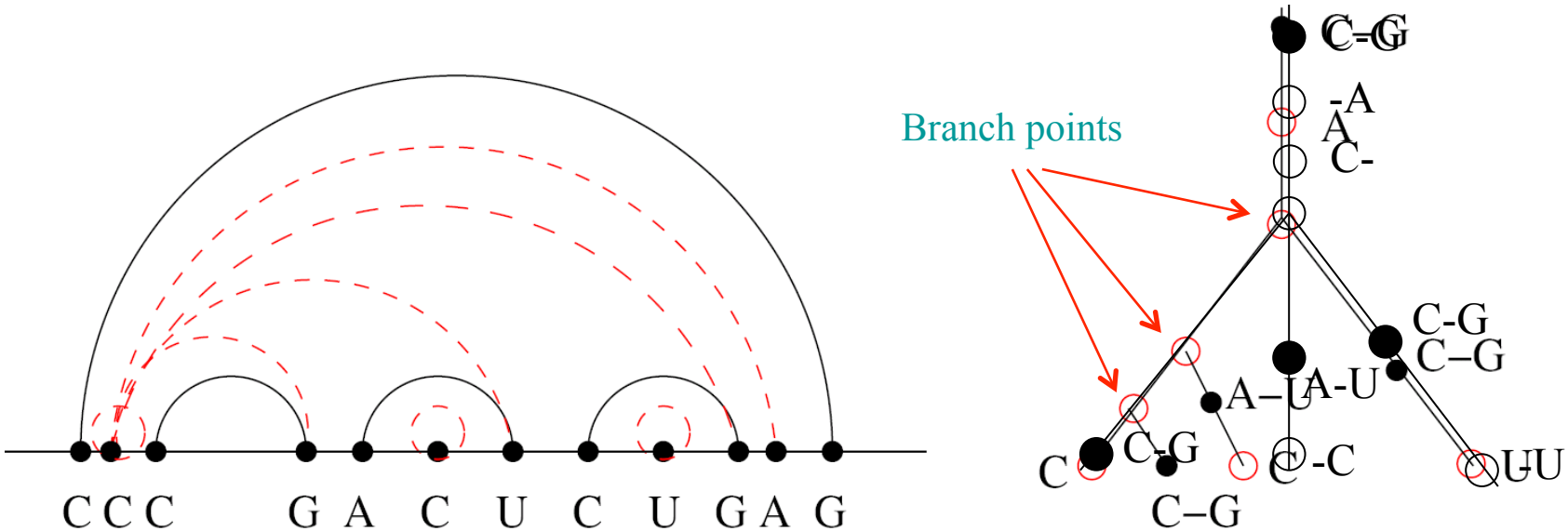
Extend the binary tree into the whole sequence s[1…n]

- Solid nodes represent the basepairs.
- Dotted nodes represent either branch point or unpaired bases.
- Now the s[1…n] with its structure is changed into a binary tree.
- The dynamic programming compares each nodes against an interval.
- It is in $O(n^3 m_1)$ time. (m_1 the number of branch points)



Using a binary tree to represent an RNA

- Solid nodes represent the basepairs.
- Void nodes represent either branch point or unpaired bases.



The dynamic programming aligns each subtree rooted at node *v* against an subinterval *(i,j)*

If root v is a solid node (first and last bases of the subsequence represented by the subtree are paired to each other):

If v is a void node (the subsequence's last base is unpaired):

If v is a void node (it is a branch point for multi-loop.):

procedure alignRNA

(* S is the set of base-pairs in RNA structure of s. S' is the augmented set. *)

for all intervals (i,j), $1 \le i < j \le n$, all nodes $v \in S'$

if $v \in S$

$$A[i,j,v] = \max \begin{cases} A[i+1,j-1,\text{child}(v)] + \gamma(s[l_v],t[i]) + \gamma(s[r_v],t[j]) \\ \quad +\delta(t[i],t[j],s[l_v],s[r_v]), \\ A[i,j-1,v] + \gamma('-',t[j]), \\ A[i+1,j,v] + \gamma('-',t[i]), \\ A[i+1,j,\text{child}[v]] + \gamma(s[l_v],t[i]) + \gamma(s[r_v],'-'), \\ A[i,j-1,\text{child}[v]] + \gamma(s[l_v],'-') + \gamma(s[r_v],t[j]), \\ A[i,j,\text{child}[v]] + \gamma(s[l_v],'-') + \gamma(s[r_v],'-'), \end{cases}$$

else if $v \in S' - S$, and v has one child

$$A[i,j,v] = \max \begin{cases} A[i,j-1,\text{child}[v]] + \gamma(s[r_v],t[j]), \\ A[i,j,\text{child}[v]] + \gamma(s[r_v],'-'), \\ A[i,j-1,v] + \gamma('-',t[j]), \\ A[i+1,j,v] + \gamma('-',t[i]), \end{cases}$$

else if $v \in S' - S$, and v has two children

$$A[i,j,v] = \max_{i \le k \le j}\{A[i,k-1,\text{left_child}[v]] + A[k,j,\text{right_child}[v]]\}$$

end if

Modifications of the formulation

- Banded alignment. [$O(n^2\delta_n)$]

- Complicated scoring functions:
  - Affine gap penalties in both stacks and loops.
  - Learning scoring matrix from handmade alignments.

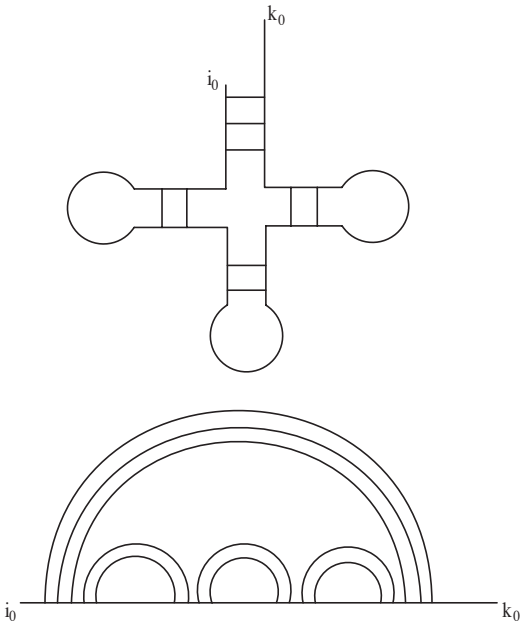
Structural Alignment of Pseudo-knotted RNA

RNA pseudo-knotted structures

RNA alignment problem has been solved for RNAs with a regular structure,
i.e. non-pseudo-knotted structures.

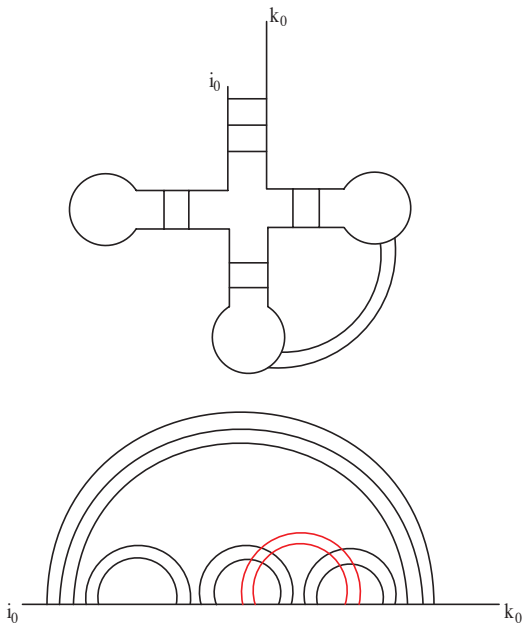
Regular structure:

All base pair are non-crossing.



Pseudo-knotted structure:

Some of the base pairs are crossing.



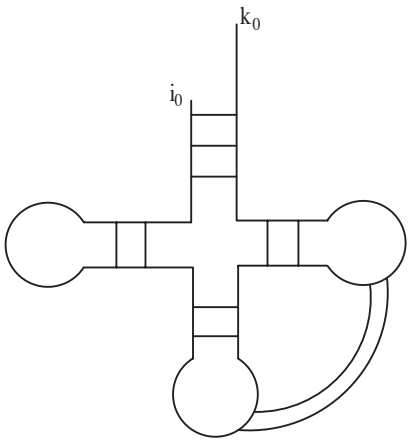
Solving problem for pseudo-knotted RNAs

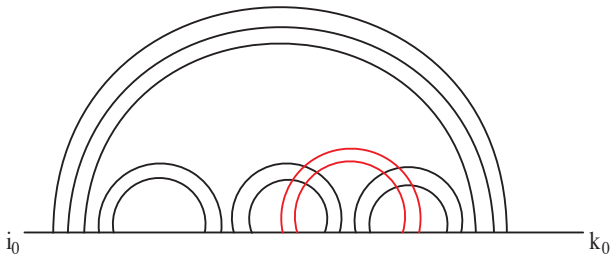
Dynamic programming technique used to align subsequences.

Challenge: Aligning RNA with general pseudoknot sturctures is hard. (Jiang et. al JCB 2002).

Formal definition of pseudo-knots such that

- To classify the pesudoknot strcutres so that most common pseudoknot is compuatable.
- computation is not very expensive
- biologically important

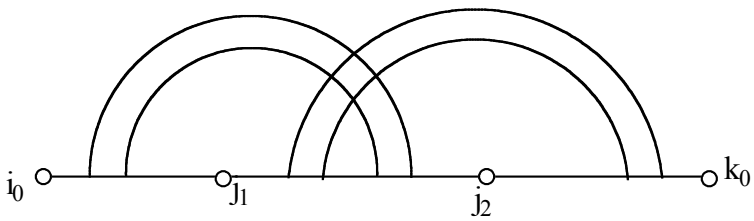


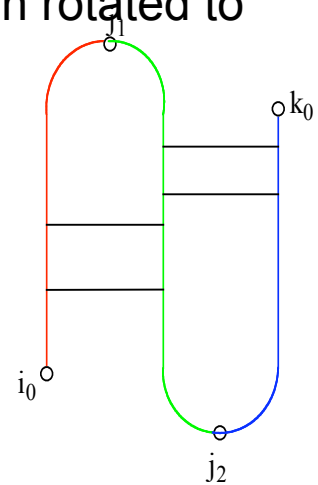


Definition: simple pseudo-knot

- How can we define a pseudo-knot?
- There are many pseudo-knot definitions: Akutsu [journal 2002?], Rivas&Eddy, ….
- For pediction.

- We start with Akutsu's simple pseudo-knot formalism:
- All base pairs non-crossing and horizontal when rotated to form 2 loops.

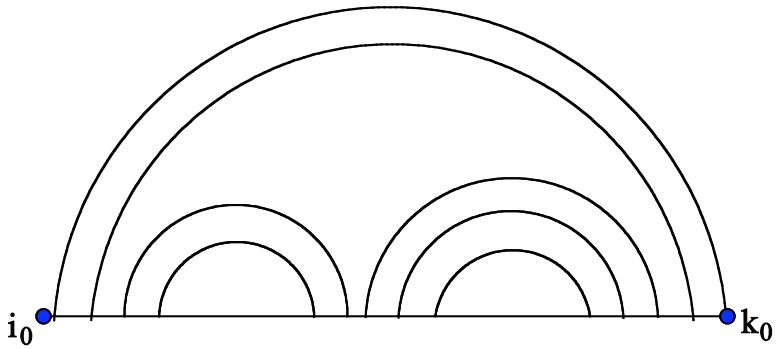




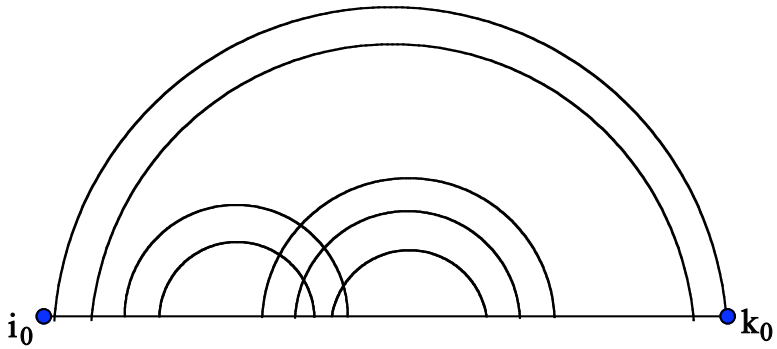
Sub-structure for a simple pseudo-knot

For DP algorithm, how to define sub-structure?

- Regular structure:
- continuous subintervals as
- substructure of recursion.



- Simple Pseudo-knot:
- can not use this substructure
- due to interweaving base pairs.



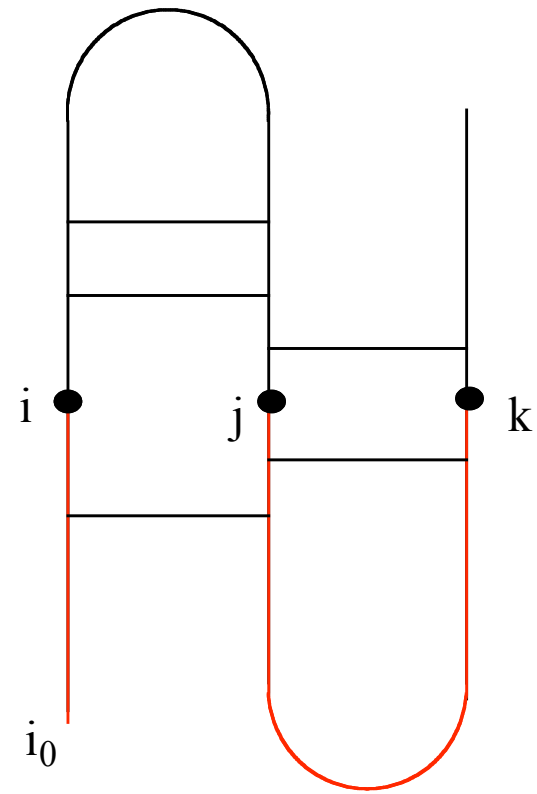
Sub-structure for a simple pseudo-knot

sub-pseudoknot P(i, j, k) as

the union of two subintervals

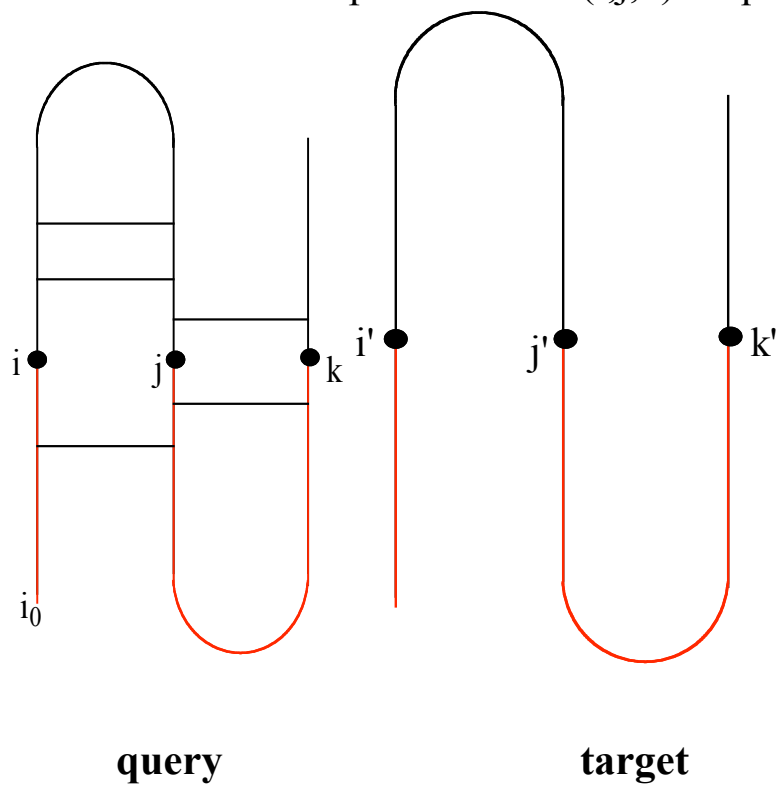
$P(i, j, k) = [i_0, i] \cup [j, k]$

frontier (i.j.k)



Naive approach

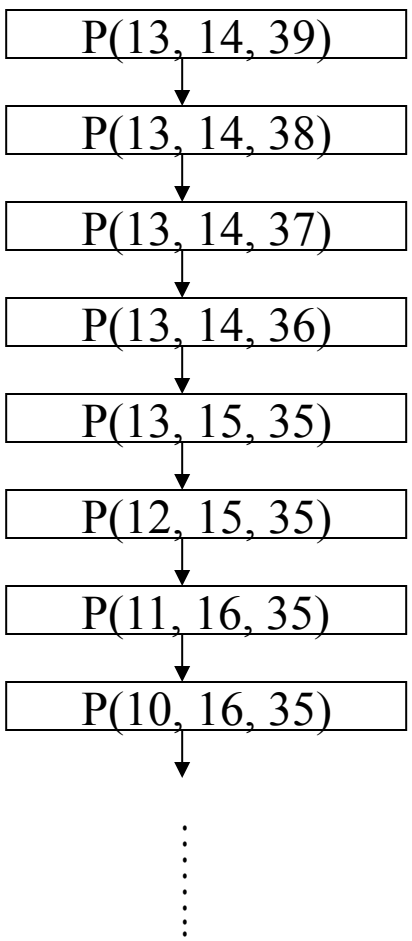
B[i, j, k, i', j', k']: Optimal score of the alignment of the sub-pseudoknot P'(i',j',k') in target to sub-pseudoknot P(i,j,k) in query.

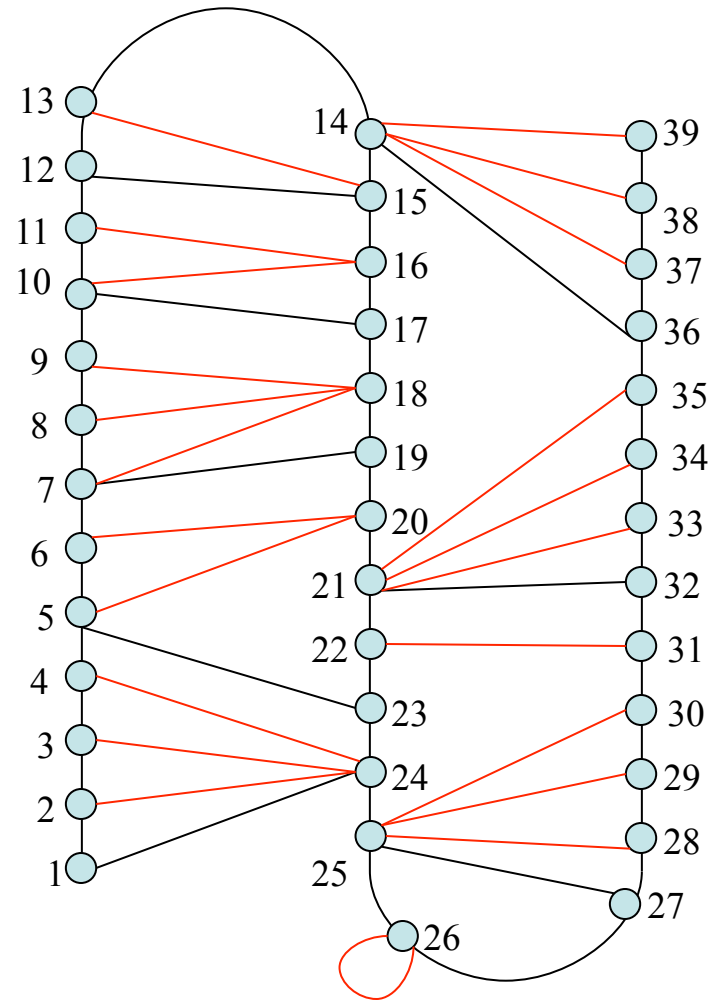


Compute B[i, j, k, i', j', k']
$\Rightarrow O(m^3n^3)$ scores.
(m:query, n:target)

Instead of all triplets in the query, consider only the valid sub-pseudo-knots that will represent the simple pseudo-knot.

Use a chain of sub-pseudoknots to represent simple pseudo-knot

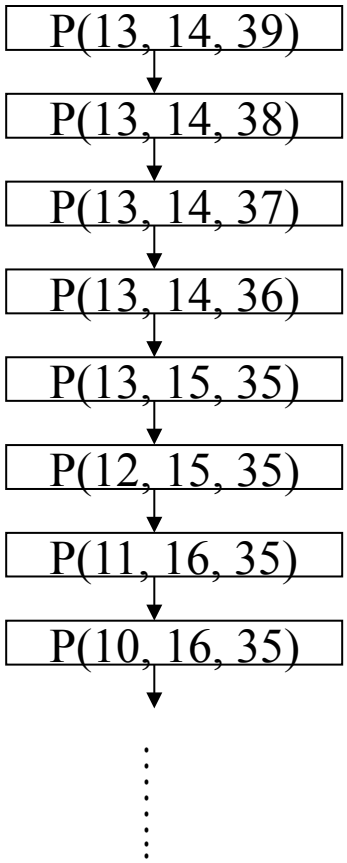


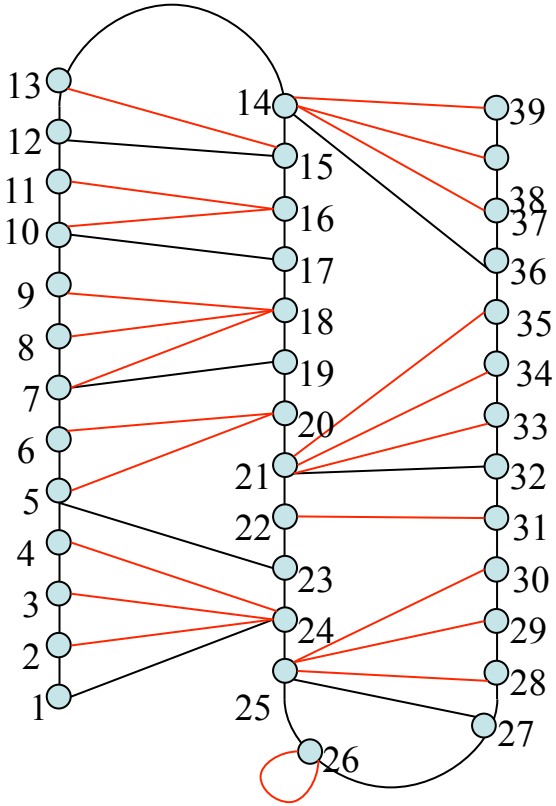


Why Chaining?

- DP: use sub-optimal solution of the child sub-structure to compute optimal score at each step.

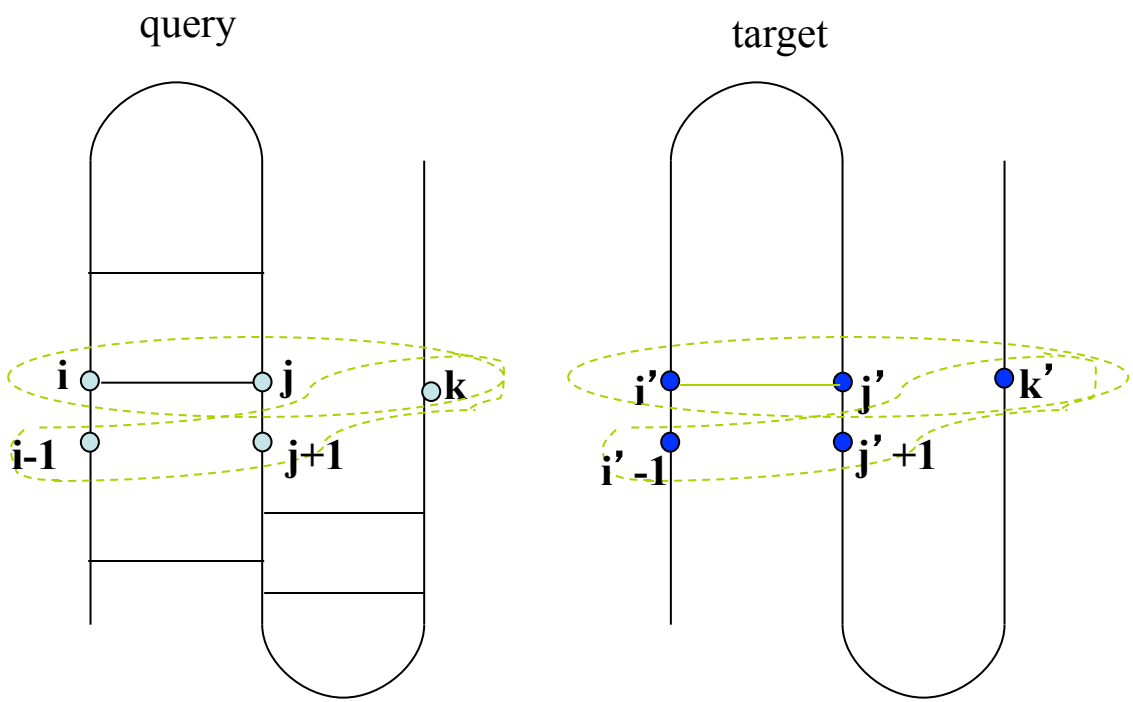
- compute B[i,j,k, i',j', k']
- => $O(mn^3)$ scores
- (m:query, n:target)





Alignment Algorithm Recursions: (i,j) is a base pair case

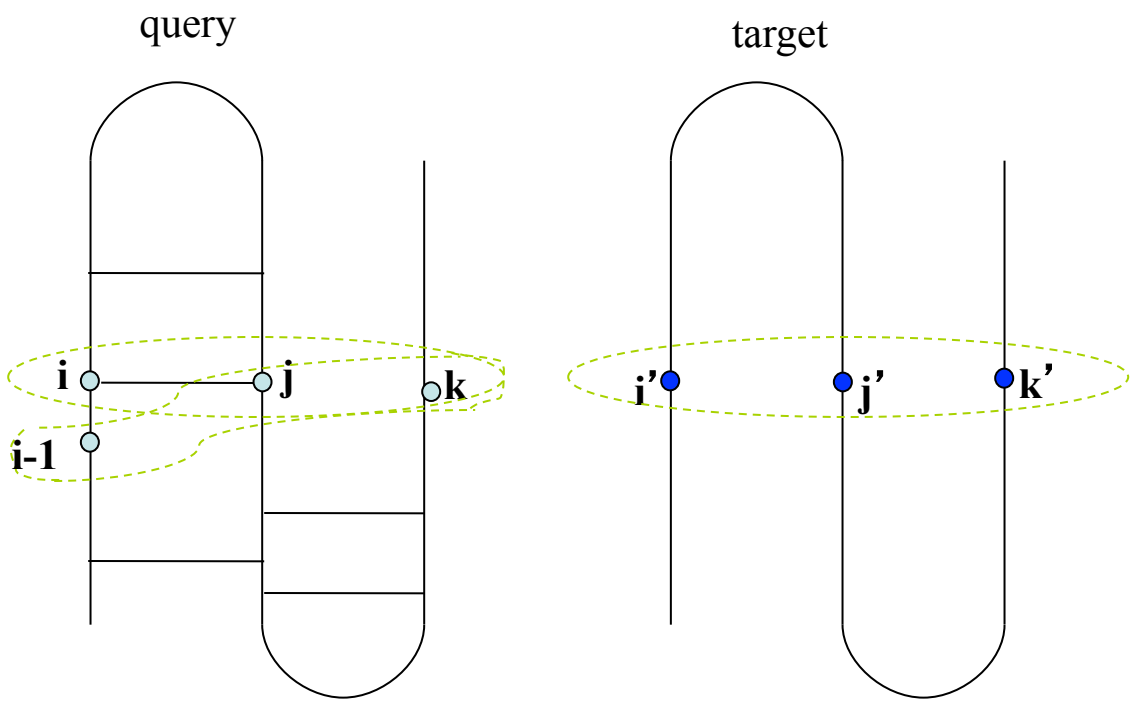
MATCH: (i,j) and (i’,j’) are corresponding pairs



- B[i, j, k , i’ , j’ , k’] = max {MATCH, INSERT, DELETE}

Alignment Algorithm Recursions: (i,j) is a base pair case

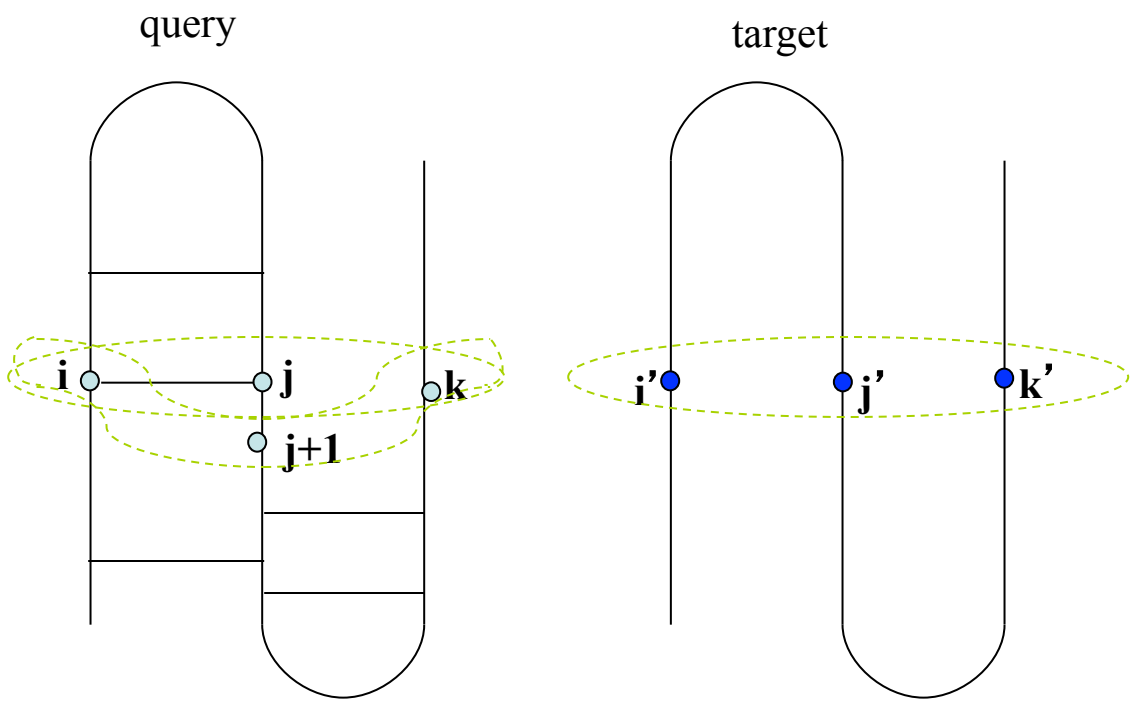
DELETION: i is deleted



- B[i, j, k , i' , j' , k'] = max {MATCH, INSERT, DELETE}

Alignment Algorithm Recursions: (i,j) is a base pair case

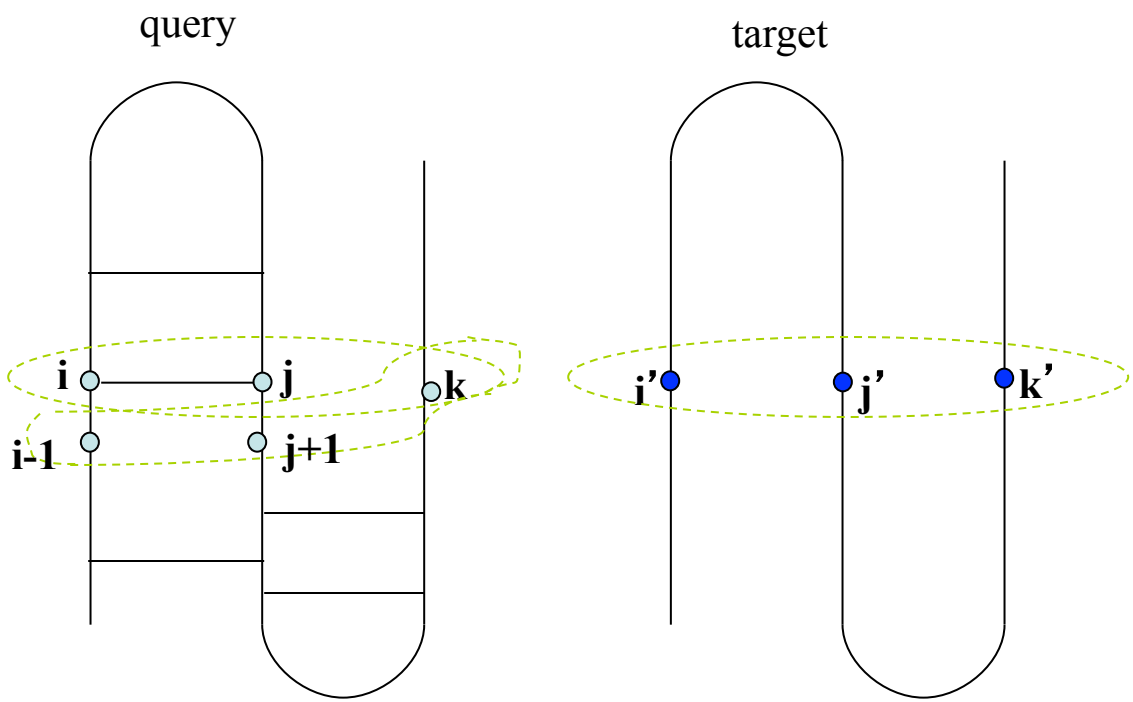
DELETION: j is deleted



- B[i, j, k , i' , j' , k'] = max {MATCH, INSERT, DELETE}

Alignment Algorithm Recursions: (i,j) is a base pair case

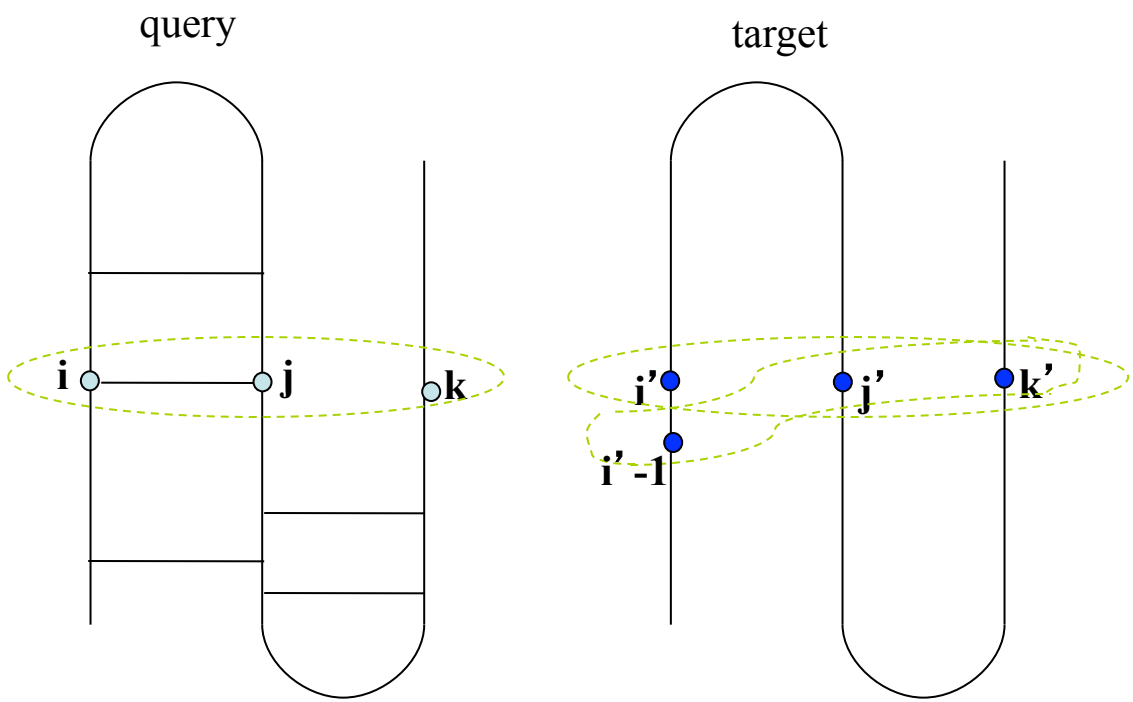
DELETION: i and j are deleted



- B[i, j, k , i' , j' , k'] = max {MATCH, INSERT, DELETE}

Alignment Algorithm Recursions: (i,j) is a base pair case

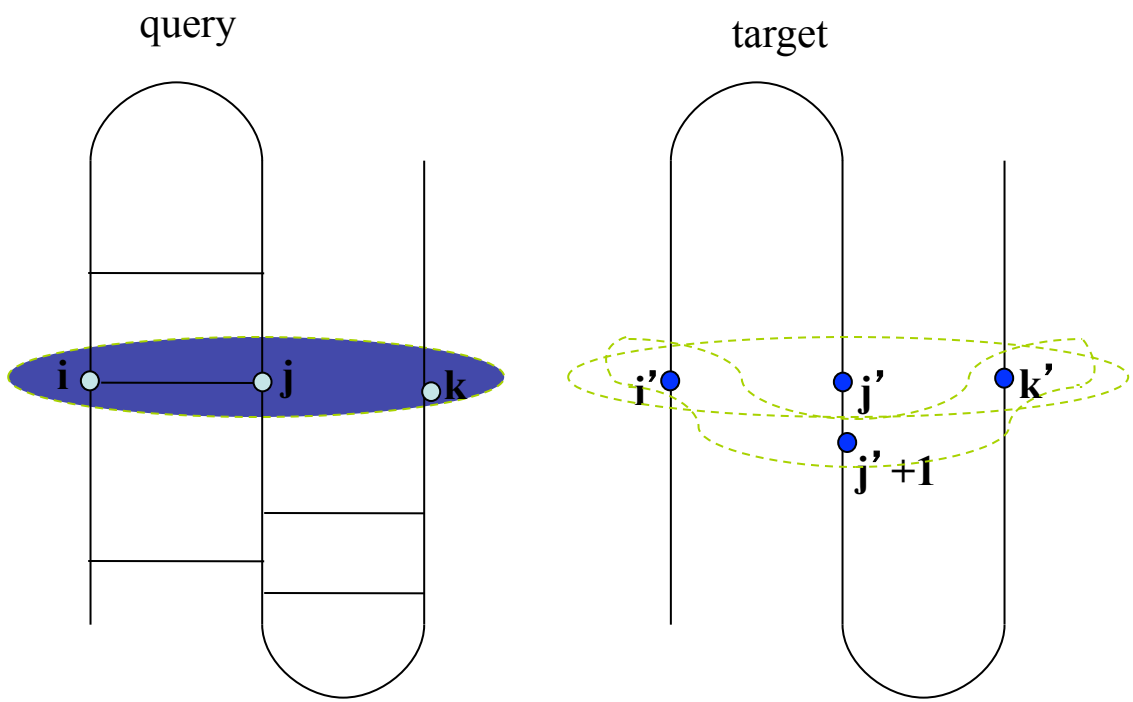
INSERTION: i' is inserted



- B[i, j, k , i' , j' , k'] = max {MATCH, INSERT, DELETE}

Alignment Algorithm Recursions: (i,j) is a base pair case

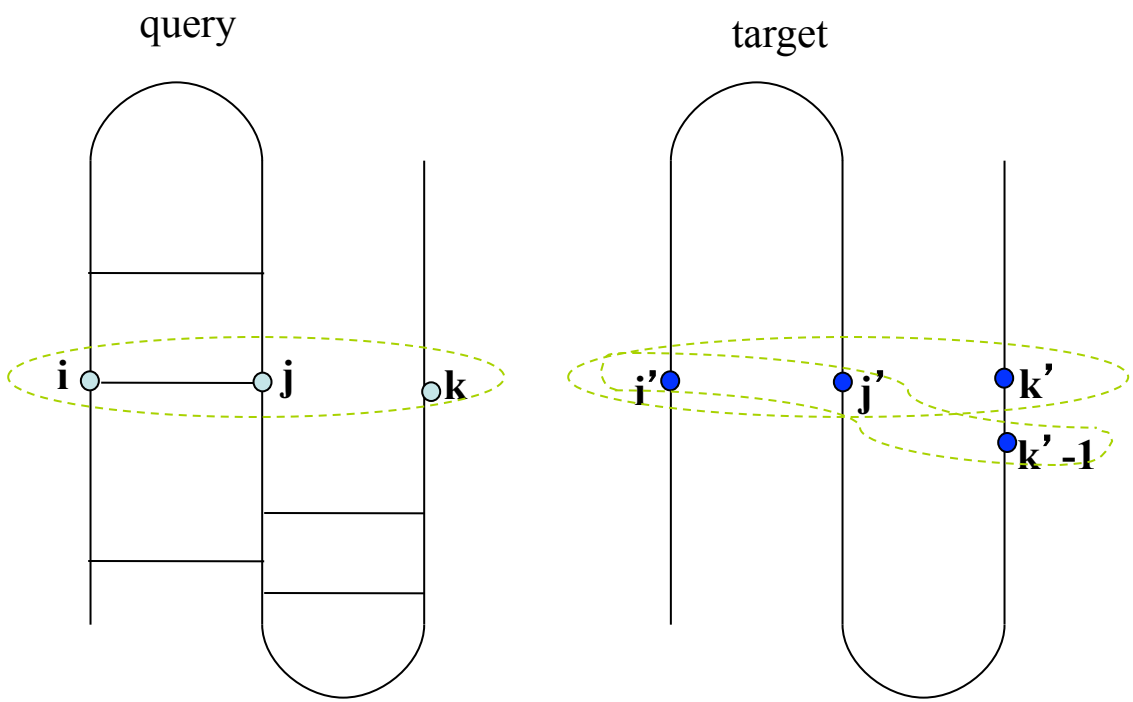
INSERTION: j' is inserted



- B[i, j, k , i' , j' , k'] = max {MATCH, INSERT, DELETE}

Alignment Algorithm Recursions: (i,j) is a base pair case

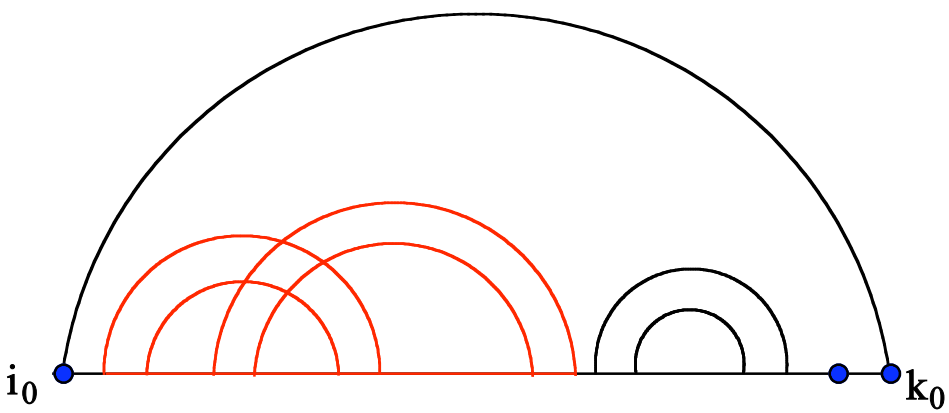
INSERTION: k' is inserted

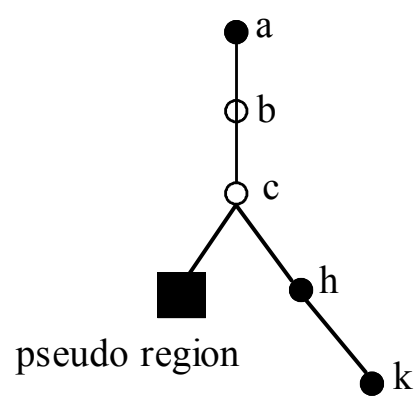


- B[i, j, k , i' , j' , k'] = max {MATCH, INSERT, DELETE}

Simple Pseudo-knot in a Regular Structure: S in R

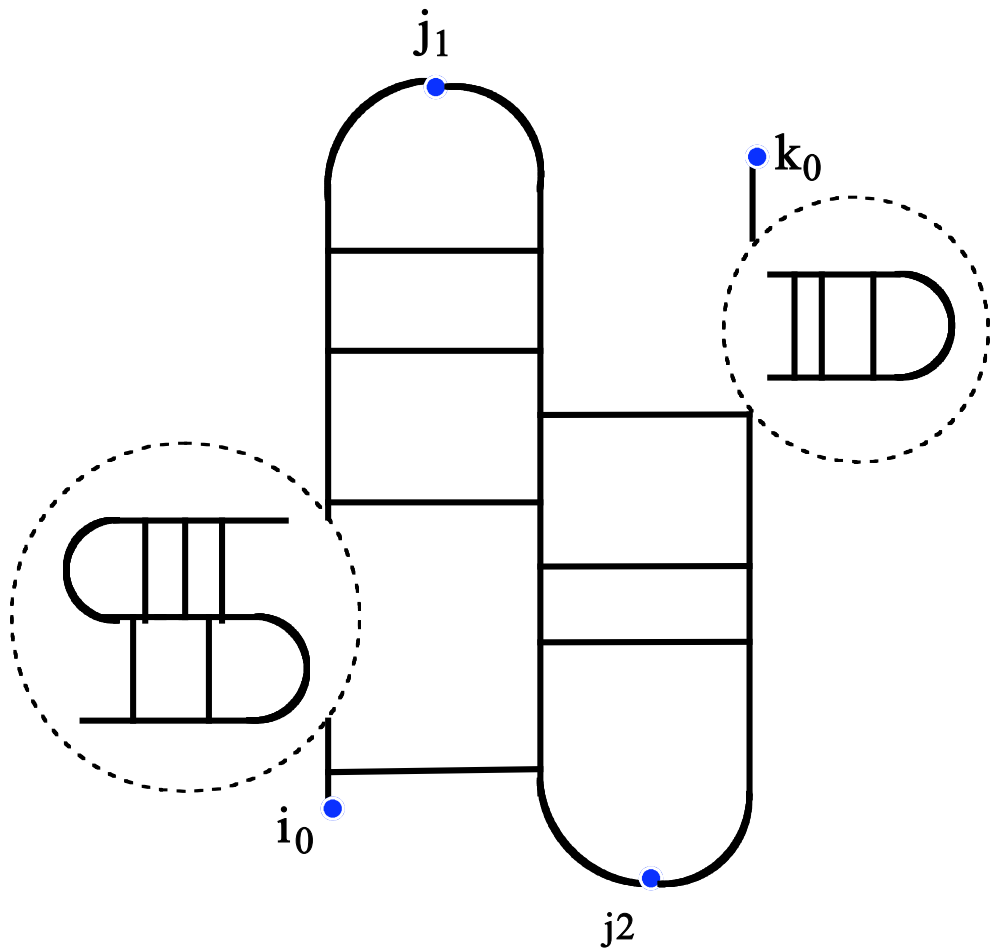
Use a binary tree to represent RNA
Solid circular nodes correspond to the actual base pairs.
Empty circular nodes correspond to unpaired bases.
Rectangular node correspond to sub-tree representing pseudo-knotted region





Simple pseudo-knot in a simple pseudo-knot: recursive simple pseudo-knot

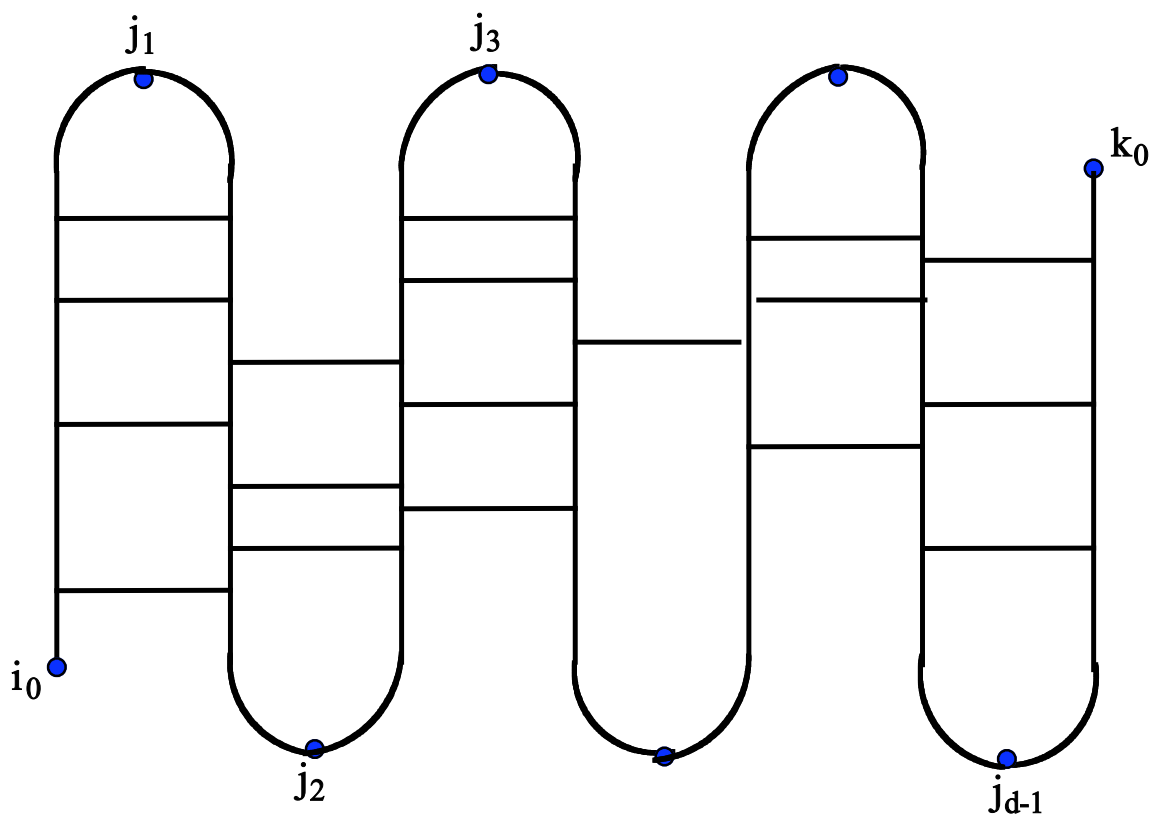
- S in S
- R in S



Which structures can we handle?

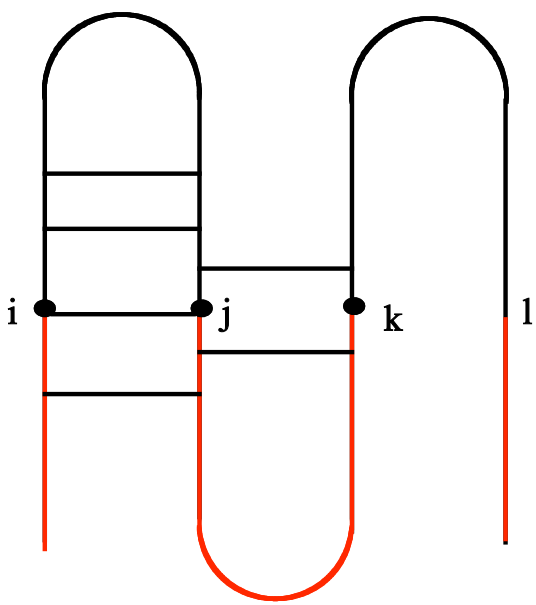
- Time complexity increases with the recursion depth of the pseudo-knotted region!
-
- R: regular structure
- S: simple pseudo-knot
-
- R: $O(mn^3)$
- S: $O(mn^4)$
- S in R: $O(mn^4)$
- R in S: $O(mn^5)$
- R in S in R: $O(mn^5)$ = S in S in R: $O(mn^5)$
- R in S in S in R = $O(mn^6)$
- ……..

Can we handle simple pseudo-knots with higher degree: standard pseudo-knots?

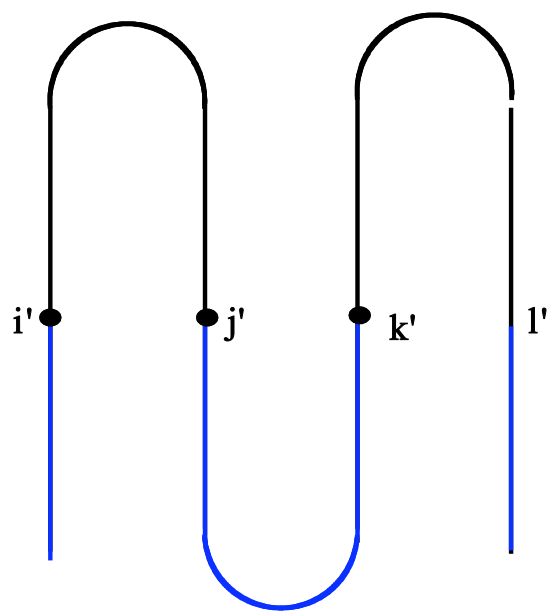


Can we handle simple pseudo-knots with higher degree: standard pseudoknots?

- Yes! By revising the sub-pseudoknot structure and the recursion cases accordingly



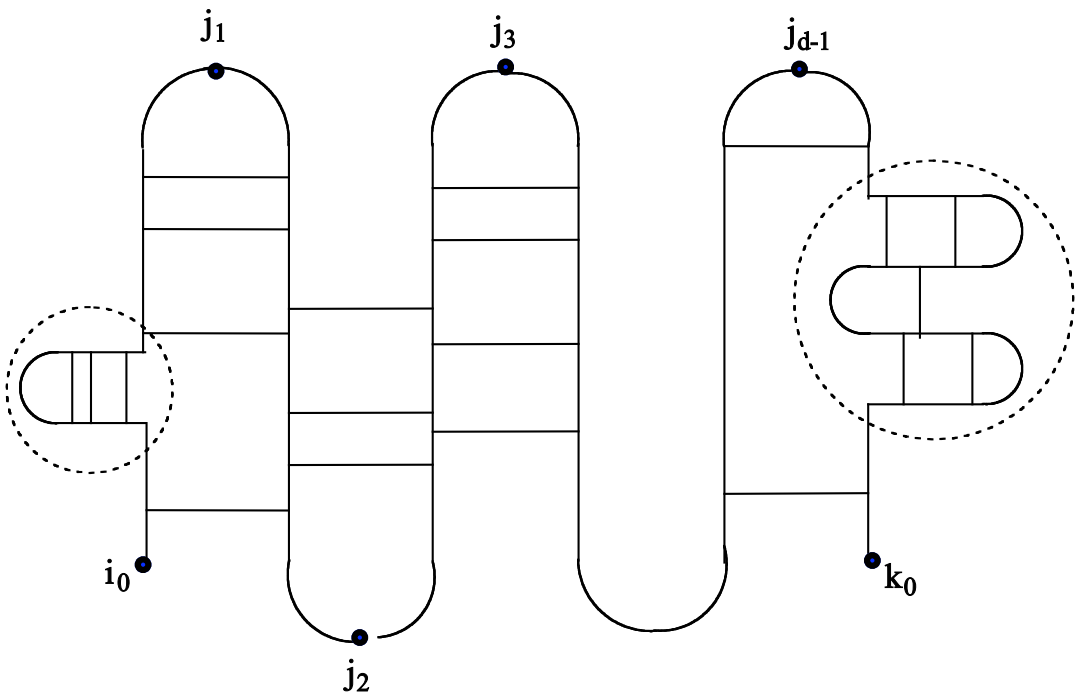
query



target

Can we handle recursive standard pseudoknots?

Yes! Same reasoning with recursive simple pseudoknots.

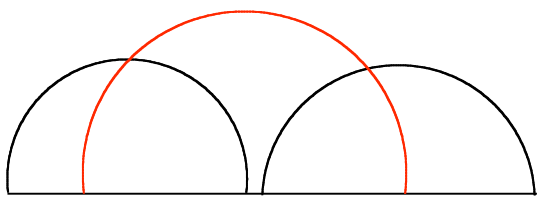


What is left? What can we NOT handle?

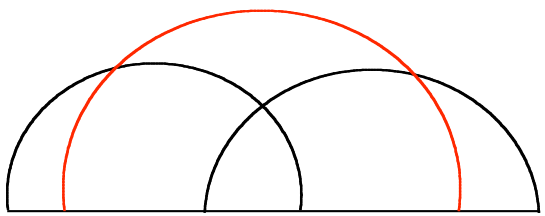
?We can handle the class of pseudoknots defined by Akutsu which is the second largest class currently defined. We can additionally handle standard and recursive standard pseudoknots which are defined by us.

A&U $\subseteq$ **A&U U {standard/recursive standard pseudoknots}** $\subseteq$ **R&E**

The largest class is defined by Rivas and Eddy. An example from this class we can not handle:



We can handle this!
(Standard pseudo-knot of degree 4)



We can NOT handle this!

Implementation: PAL

- C++ implementation of our algorithm.
  - input:
    - a query sequence with known structure (R/S/S in R)
    - a target sequence
  - output:
    - all high scoring local alignments in the target sequence

Testing

- Test Data:
- RFAM database, 6 RNA families with simple pseudo-knotted structures. (simple pseudo-knots in regular structure)

    - UPSK
    - Antizyme
    - Corona FSE
    - Corona pk3
    - Parecho CRE
    - IFN gamma

Test 1: Structure Prediction

- How good is PAL in inferring structure of the target sequence?
  - Pick 2 seed members of an RNA family as query and target.
  - Align them.
  - Compare the inferred structure of target with annotated structure in Rfam.

Test 1: Structure Prediction Results

- TP, FP, FN, Sensitivity, Specificity

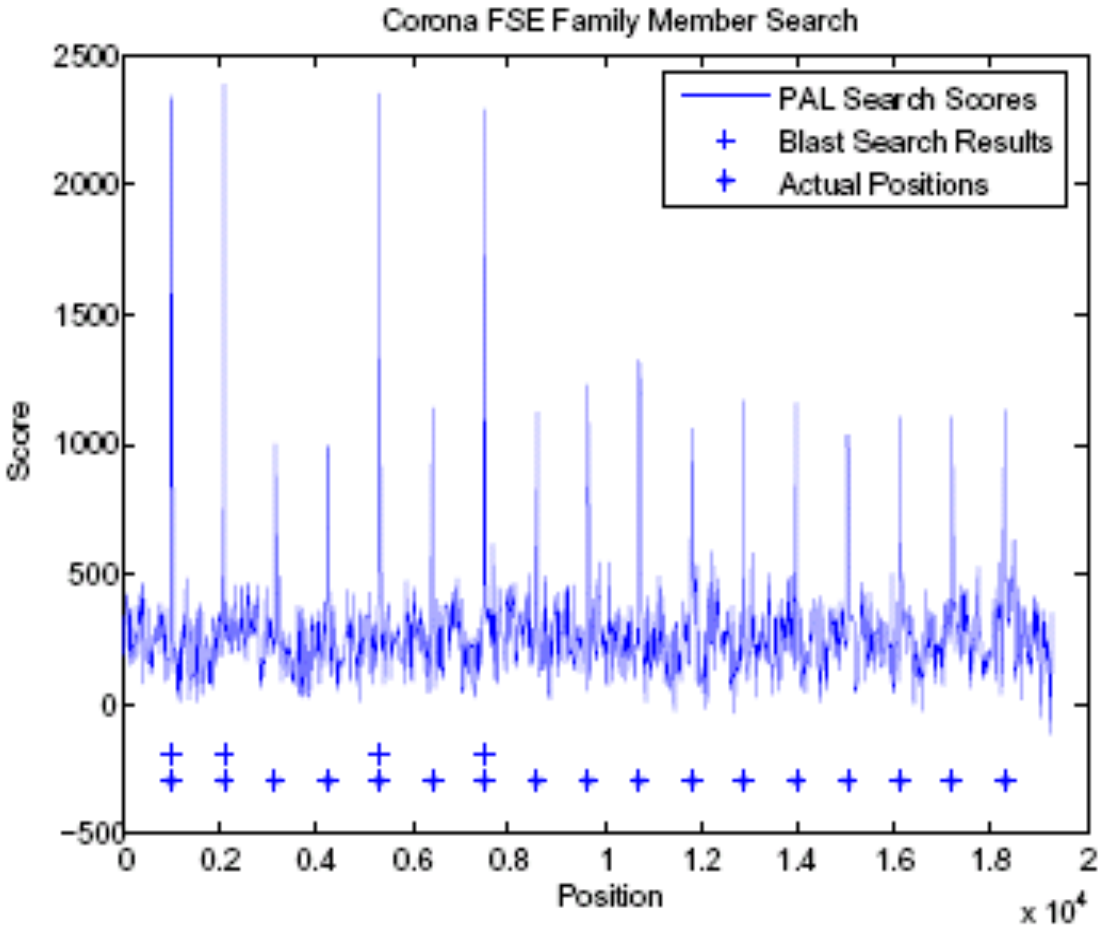
RNA Family	Specificity (Mean)	Sensitivity (Mean)
UPSK	1	1
Antizyme	0.99	0.99
Parecho	0.95	0.94
Corona FSE	0.94	0.94
Corona pk3	0.97	0.97
IFN Gamma	0.93	0.93

- Specificity = TP/(TP+FP)
- Sensitivity = TP/(TP+FN)
- Both measure is ~0.95
- PAL is a strong predictor of structure

Test 2: Homologue Search

- How well is PAL in finding the homologues of an RNA sequence?
  - Generate a random genome.
  - Insert the members of an RNA family.
  - Pick one of the members as a query.
  - Search for the homologues of the query.
  - Can we locate the members?

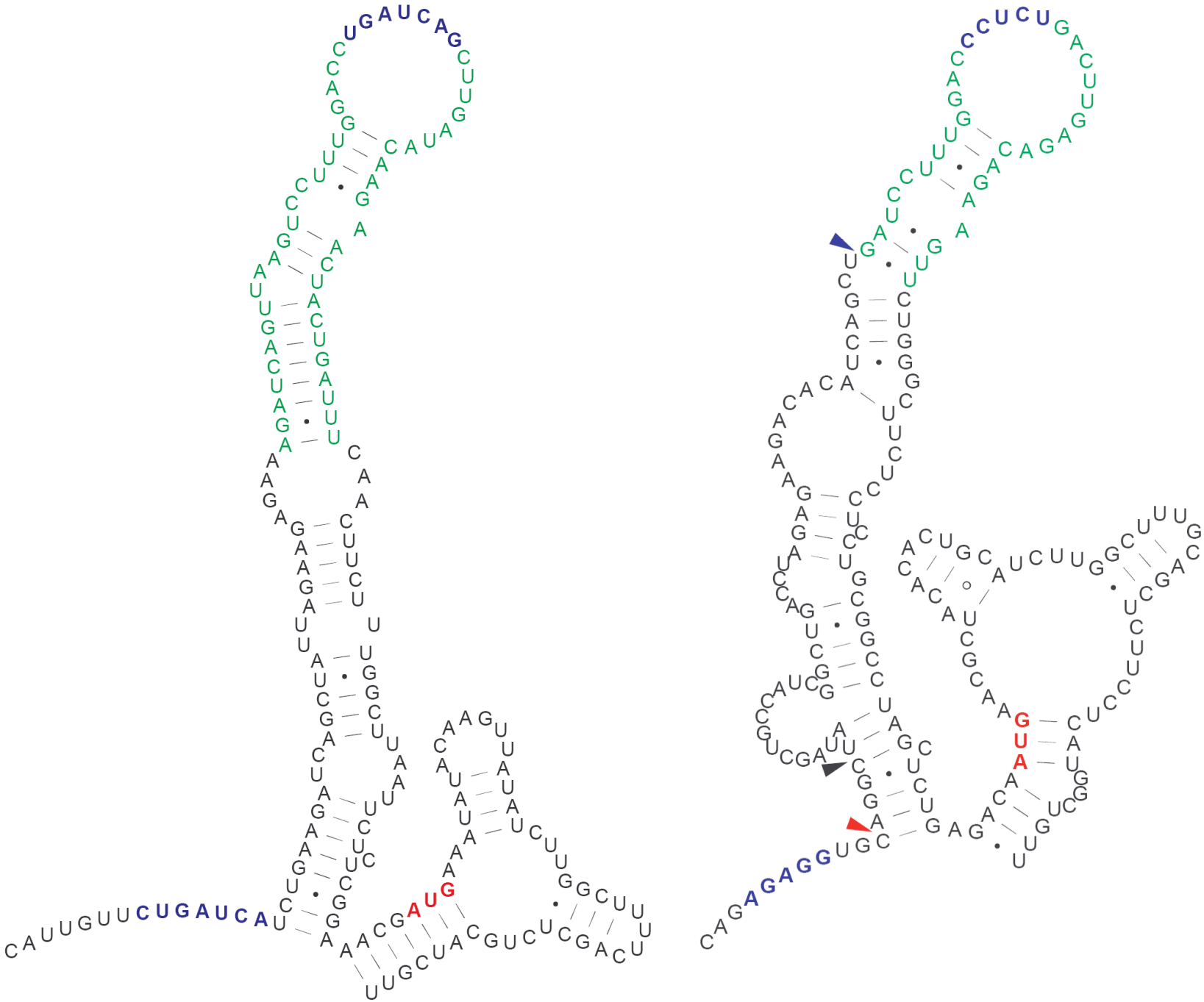
Test 2: Homologue Search Results



RNA Family	# Found	
	BLAST	PAL
UPSK	3	3
Antizyme	12	12
Parecho CRE	4	4
Corona-FSE	4	17
Corona-pk3	5	13
IFN-gamma	4	4

Novel Homolologues Search

- Searched mouse, rat and gerbil genomes for homologues of
- IFN-gamma RNA family.



Conclusion

- PAL is a viable tool in finding novel homologues and inferring structure.

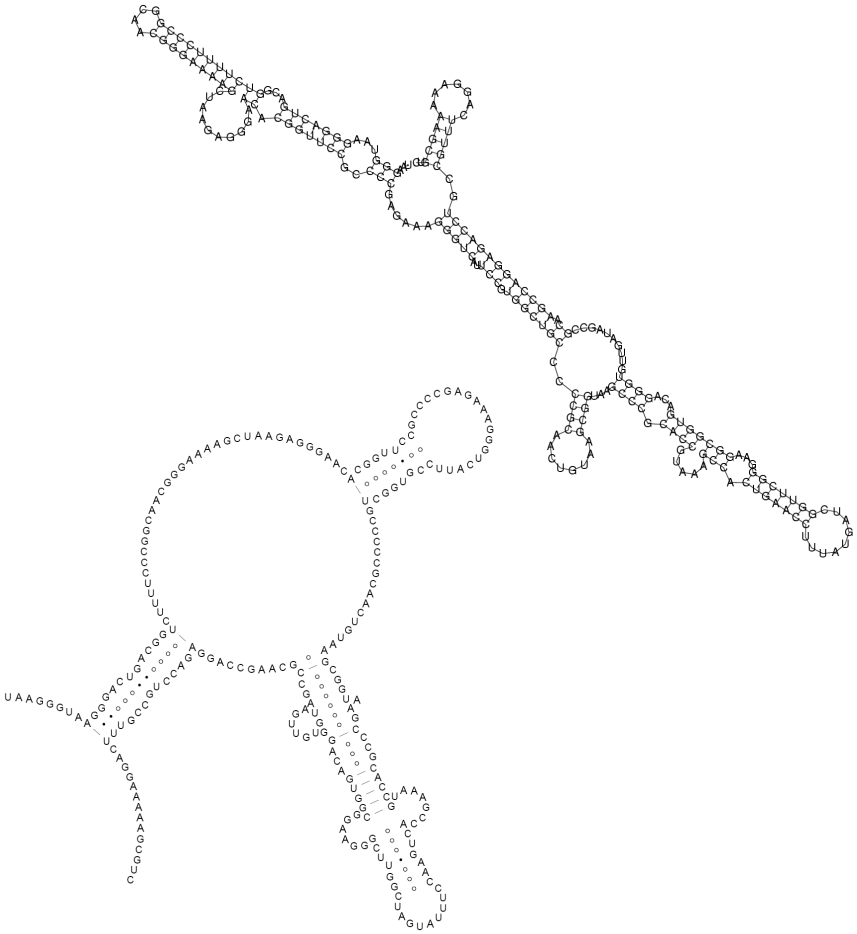
- We hope PAL will help to understand and explore the impact of
- pseudo-knotted RNAs in cellular function.

Reformulate the folding problem

RNA secondary structure prediction problem

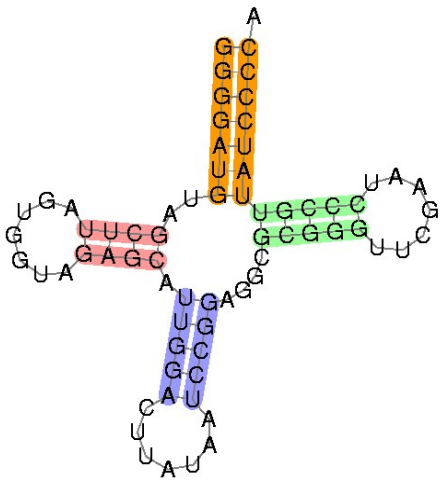
- Algorithms/programs to compute the minimum energy:
  - Nussinov et al (1978), Waterman (1978), Smith and Waterman (1978), and Zuker and Sankoff (1984).
  - **Mfold** (Zuker 2003) and **RNAfold (ViennaRNA)** (Hofacker 2003).

- RNA folding via energy minimization has its shortcomings:
  - Prediction depends on correct energy parameters.
  - Sometimes, the true structure does not have the minimum energy.



How to find ncRNA genes from a multiple alignment

- RNAs with similar functions often have similar structures.
- Sequence changes can be tolerated by covarying mutations.



```
GCAUCGGugGUUCagu--gguaGAAU---//---CCGAUGCa
UCUAAUAugGCAUauu-----aGUGC---//---UAUUAGAa
GGGGAUGuaGCUUagu--gguaGAGC---//---UAUCCCCa
GCCGCCGuaGCUCagcccgggaGAGC---//---CGGCGGCa
GGGCCCGuaGCUUagcucgguaGAGC---//---CGGGCCCa
```

RNAalifold

- If we have correct multiple alignments, looking for covarying mutations and finding consensus structure is a good way to do structure prediction.
  - **RNAalignfold** (Hofacker et al. 2002)
  - The consensus structure prediction is more accurate.
  - To find energetically stable consensus structure is more statistically significant.
  - Still compute the MFE.
  - Covariance information is incorporated into the energy model by rewarding compensatory and consistent mutations.

$$E_{i,j} = \min\left\{ E_{i,j-1};\ \min_{\substack{k:\ i+m<k\leq j \\ \Pi_{ik}=1}} E_{i+1,k-1} + E_{k+1,j} + \beta_{ik} \right\}$$

Problem:

- Correctly aligning multiple and divergent RNA sequences without taking into account the structural information is difficult.
- To get covarying mutation, we need an alignment.
- Do the multiple alignment and structure prediction at the same time.

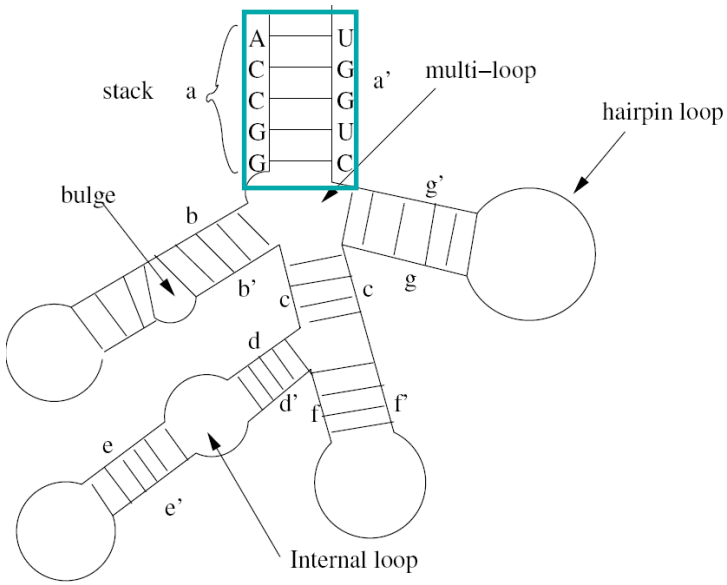
RNA consensus folding problem

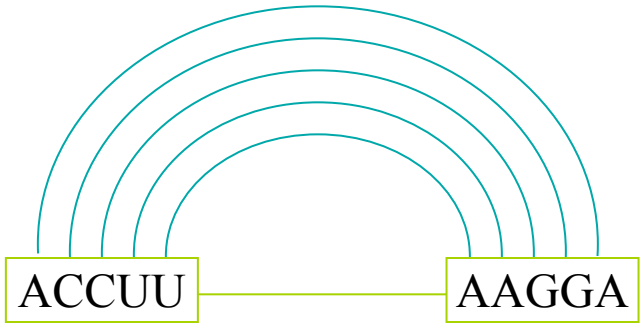
RNA consensus folding problem: computing the common secondary structure for a set of unaligned RNA sequences

- Sankoff (1985) first proposed an algorithm simultaneously align RNA sequences and find the optimal common fold.
  - Time complexity is $0(n^6)$ for two sequences with length n.
  - Implemented as **Dynalign** (Mathews and Turner 2002)
  - It's not practical for multiple sequences.

- Eddy and Durbin (1994) and other groups used stochastic context-free grammars to predict the consensus structure.
  - Start from a seed alignment.
  - Stochastic iteration to improve the alignment and predict structure.
  - Need a good seed alignment.

Motivation to a new approach

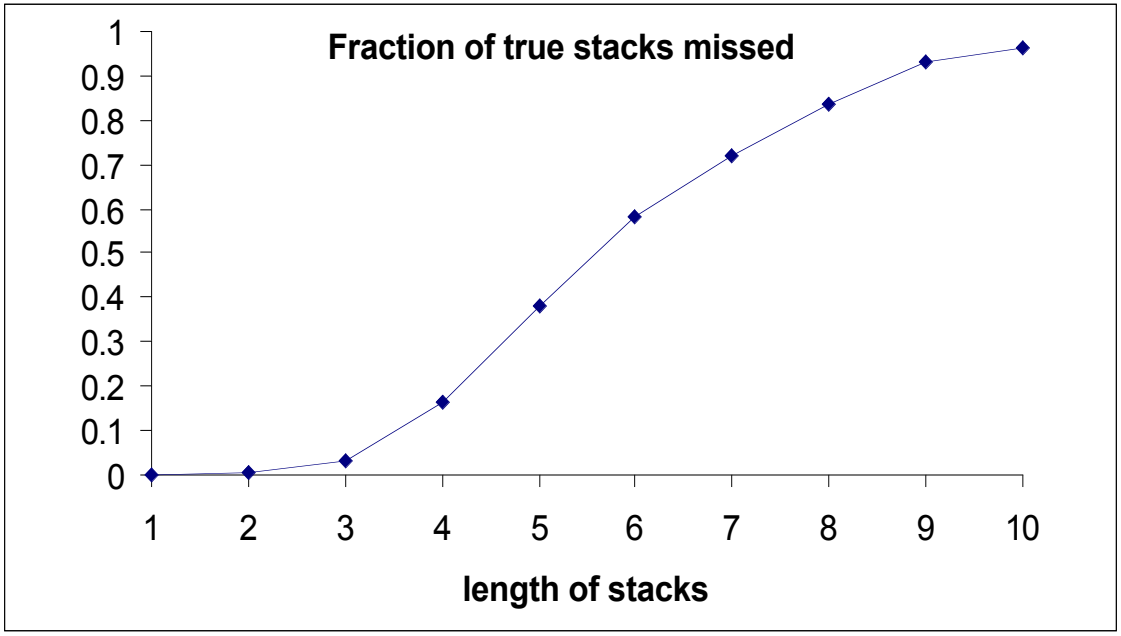




$p = (1/4)^5 < 0.001.$

- Base-pairs appear in ‘clusters’: we call them stacks, which is energetically favorable.
- Most of the stability of the RNA secondary structure is determined by stacks.
- Stacks are much less likely to occur by chance.

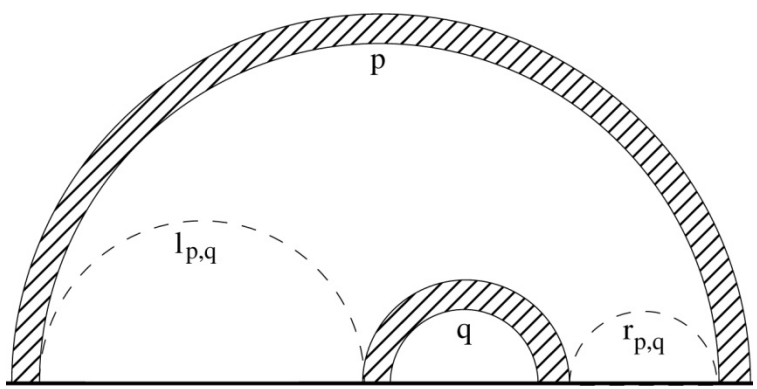
Statistics of the stacks in Rfam database



The *Stack-based Folding* algorithm based on Nussinov model

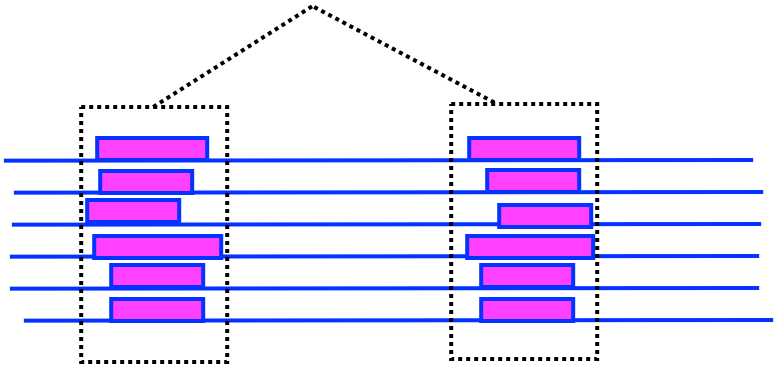
- The Nussinov model (simplified model)
  - Basically counts the number of base pairs
- The *Stack-based Folding* algorithm
  - N(p) : the maximal number of base pairs of all stack configurations within *p* (including *p* itself)

$$N(p) = p_l + \max_{\forall q \in \mathcal{F}_I(p)} \{N(q) + N(l_{p,q})\}$$



Using stacks as anchors for predictions

- The idea of anchors as constraints has been used in multiple genomic sequence alignment.
  - MAVID (Bray and Pachter, 2004)
  - TBA (Blanchette et al., 2004)



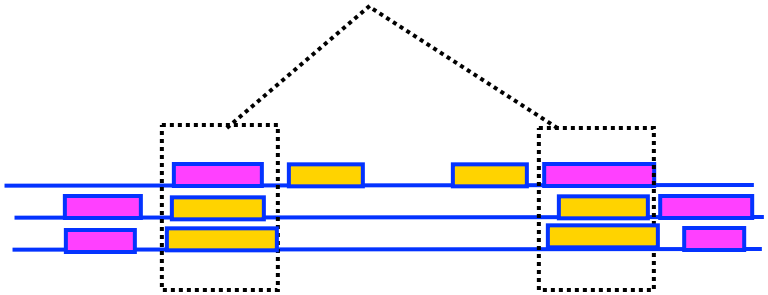
Several heuristic methods have been developed by finding anchored stacks:

- Waterman (1989) used a statistical approach to choose conserved stacks within fixed-size windows.
- Ji and Stormo (2004) and Perriquet et al. (2003) use primary sequence conservation of the stacks and the length of loop regions to reduce the searching space.

- stack anchor has low sequence similarity.
- It's hard to find correct anchors

Problem:

- Selecting one stack at a time may cause wrong matching stacks.



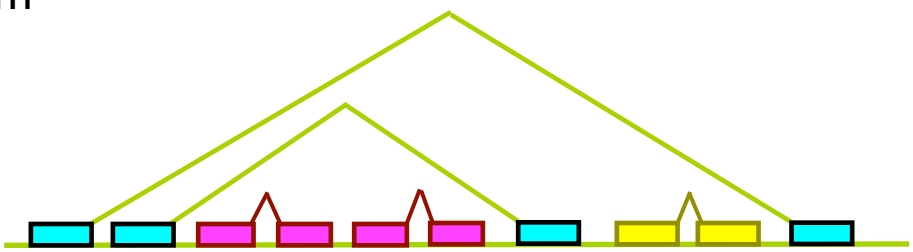
A global approach: configuration of stacks

- RNA secondary structure can be viewed as stacks plus unpaired loops. (no individual base-pairs)

- The energy of the structure is the sum of the energies of stacks and loops.

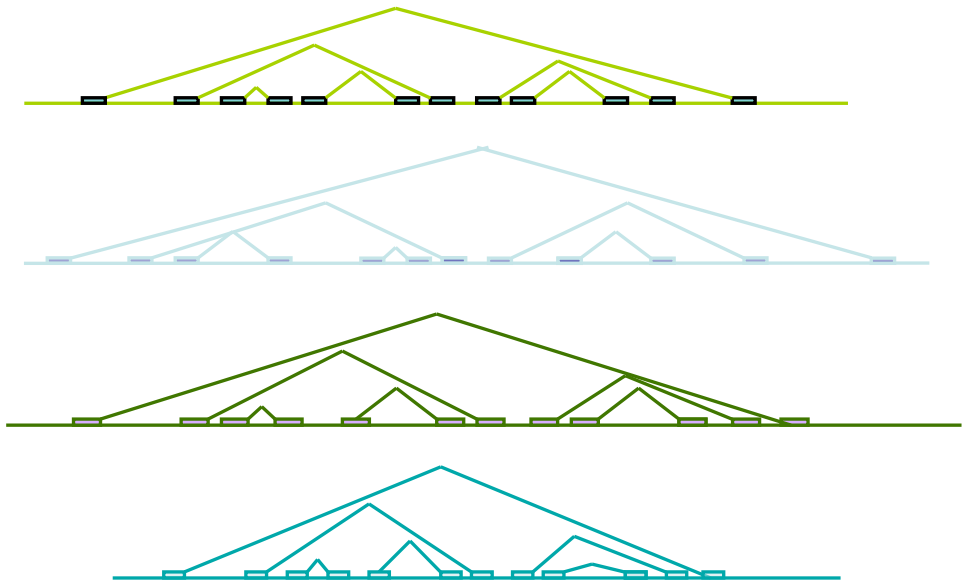
- Stack configuration:
  - Nested stacks
  - Parallel stacks
  - Crossing stacks (pseudo knots)

- More generalized stacks can include mismatches in the stacks.

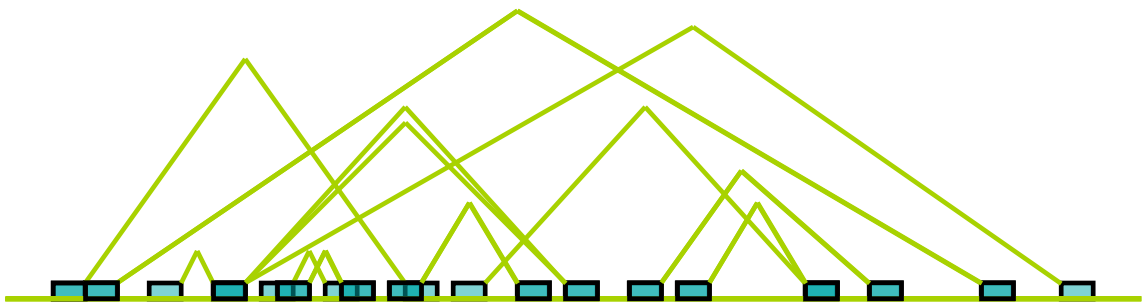


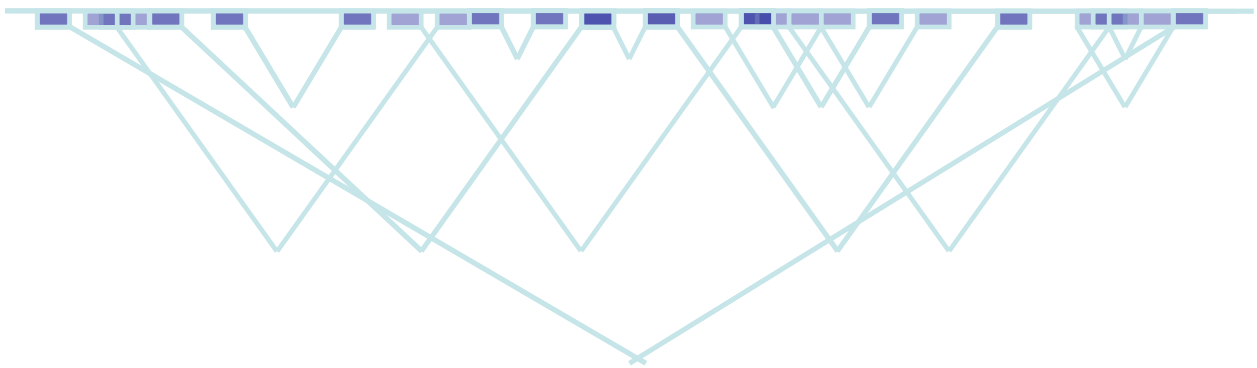
RNA Stack-based Consensus Folding (RNAscf) problem

- Find conserved stack configurations for a set of unaligned RNA sequence.
- Optimize both stability (free energy) of the structure and sequence similarity computed based on these common stacks as anchors.

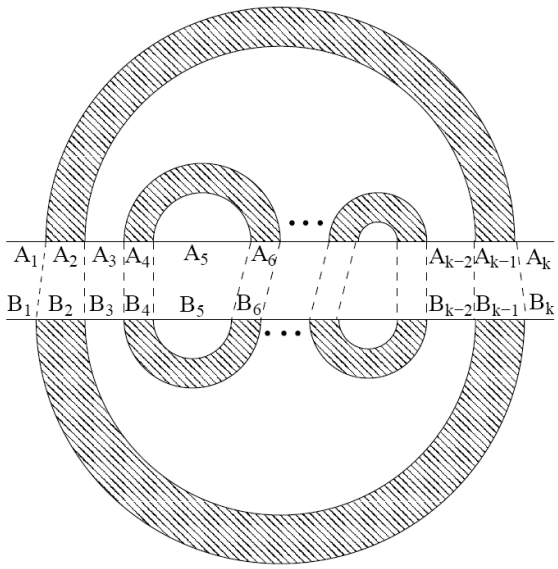


RNA stack-based consensus folding for pairwise sequences





A matching stack-configurations on two sequences



$$M(\mathcal{P}(A,B)) = w_1 \Phi(\mathcal{P}(A,B)) + w_2 \sum_{i \in \mathcal{P}(A,B)} \mathcal{S}(A_i, B_i) + w_3 \sum_{i \notin \mathcal{P}(A,B)} \mathcal{S}(A_i, B_i)$$

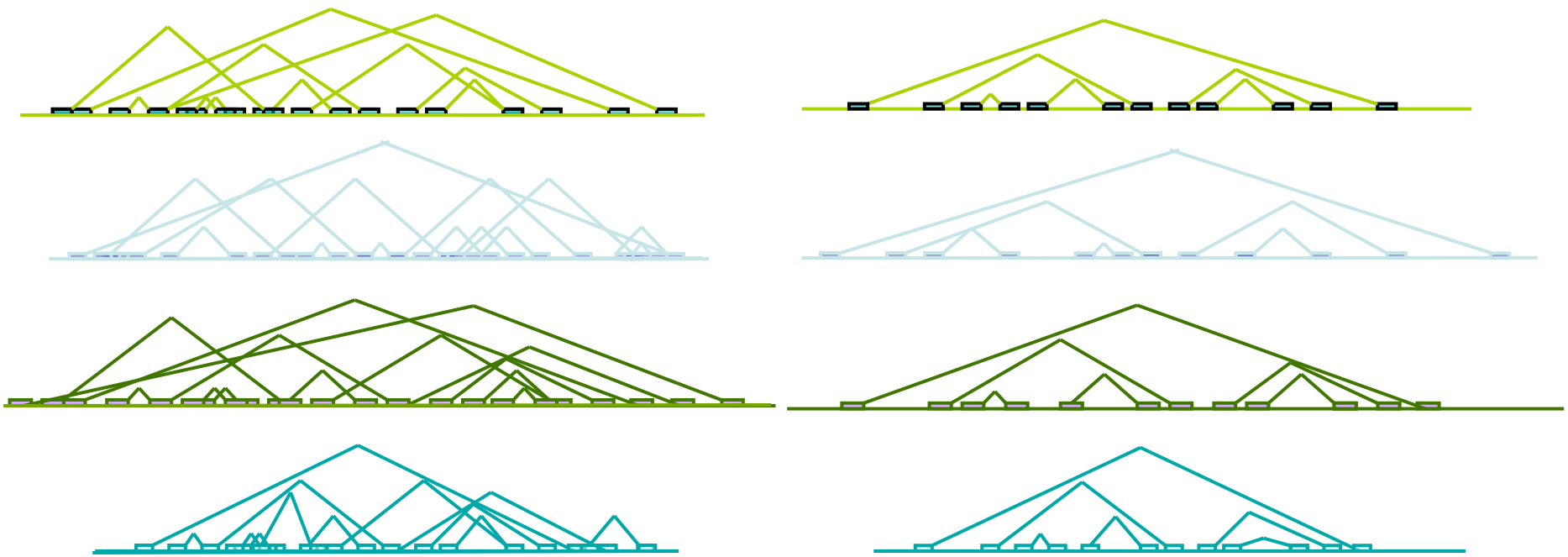
Weights of different costs

Energy of the consensus structure

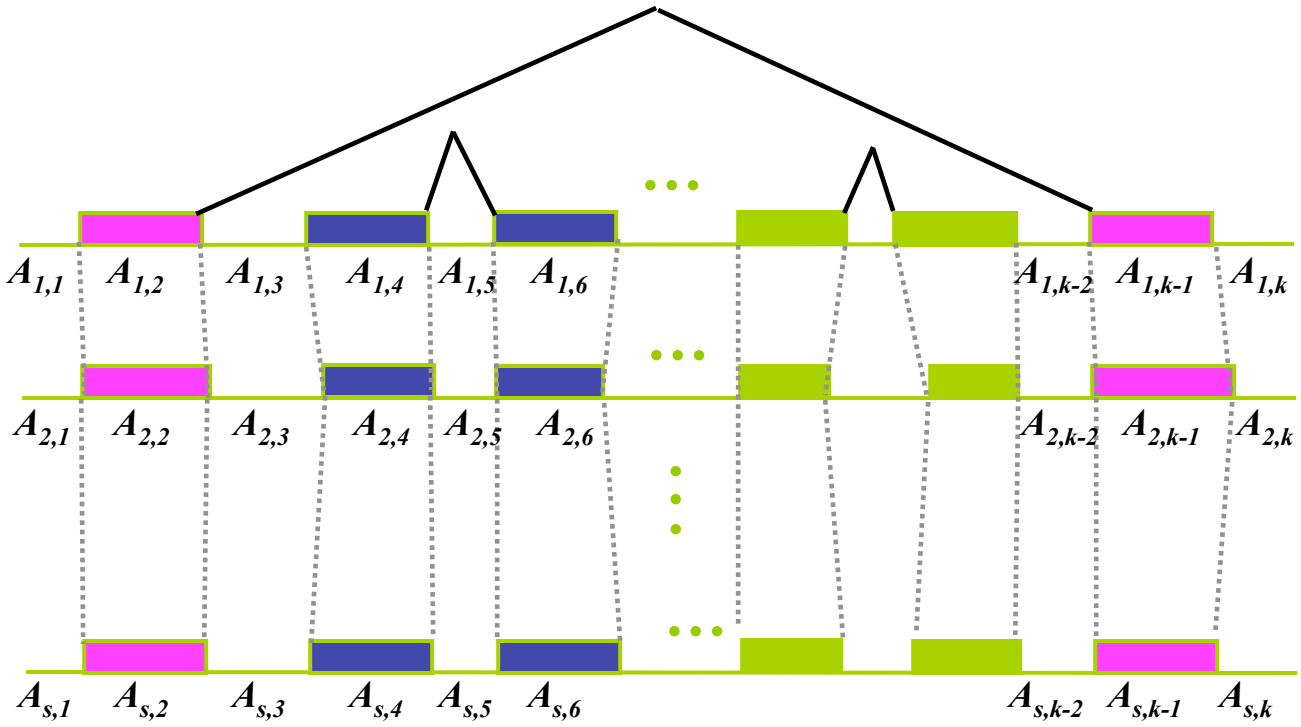
Sequence similarity of unpaired regions

Sequence similarity of stacks

RNA Stack-based Consensus Folding for multiple sequences



Cost function for multiple sequences

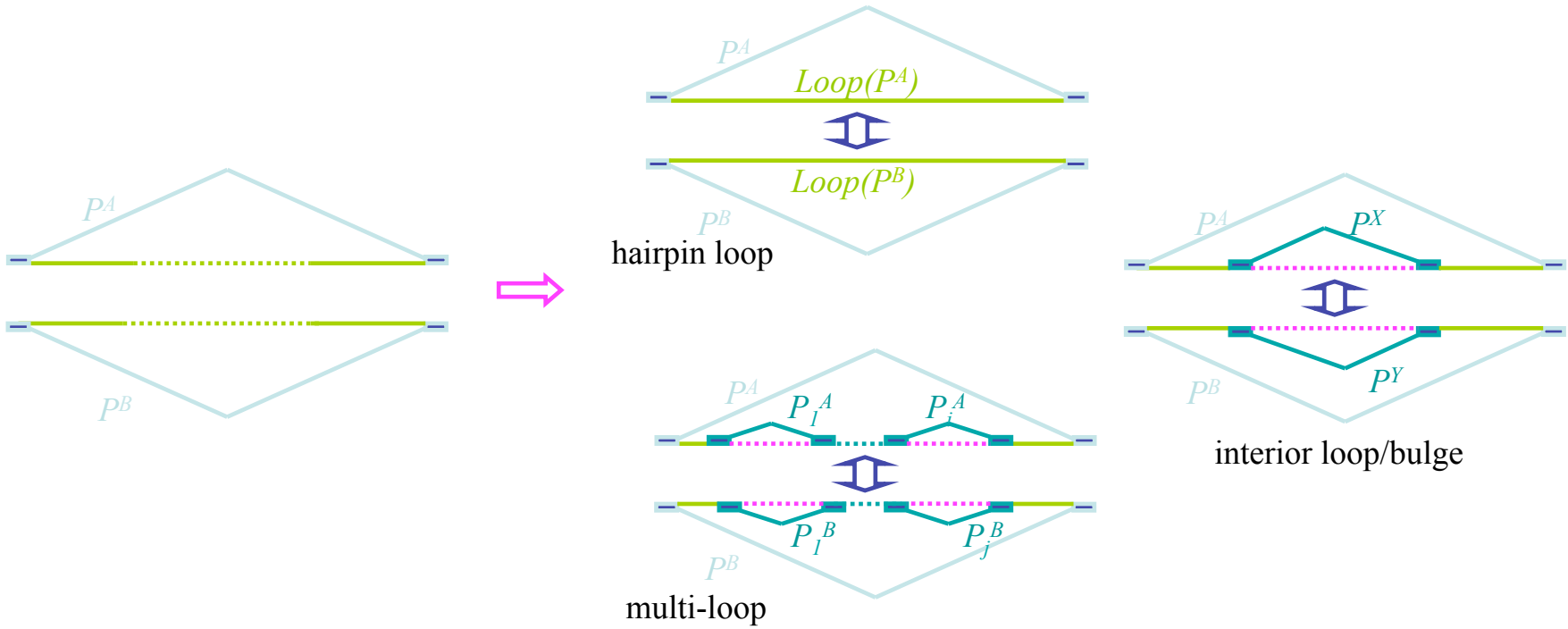


$$M(\mathcal{P}(A_1,\ldots,A_s)) = w_1\Phi(\mathcal{P}(A_1,\ldots,A_s)) + w_2\sum_{j\in\mathcal{P}}\mathcal{S}\begin{pmatrix}A_{1,j},\\ A_{2,j},\\ \ldots,\\ A_{s,j}\end{pmatrix} + w_3\sum_{j\notin\mathcal{P}}\mathcal{S}\begin{pmatrix}A_{1,j},\\ A_{2,j},\\ \ldots,\\ A_{s,j}\end{pmatrix}$$

Compute an optimal stack configuration for two sequences

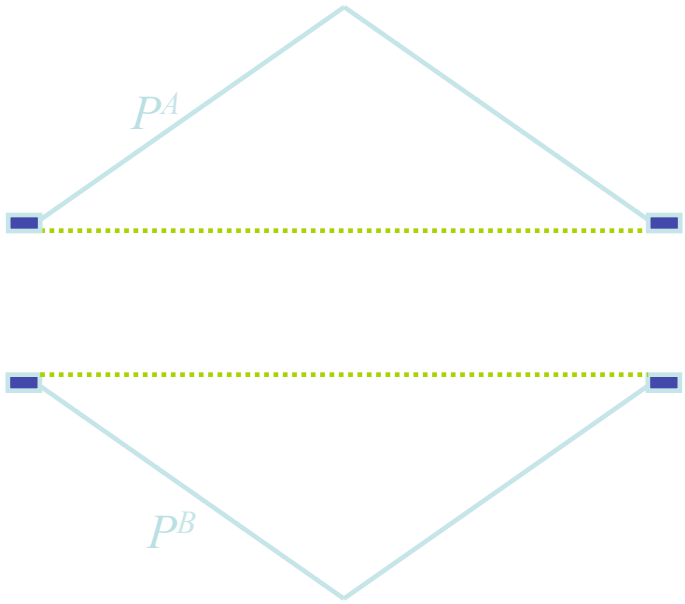
- Dynamic programming algorithm is used to align RNA sequences and find an optimal configuration at the same time.
  - The algorithm is similar to prior work (Sankoff 1985, Bafna et al. 1995)
  - Differences:
    - We use stacks as the basic structural elements.
    - Prior work used individual base pairs.
  - The computational time is $O(n^4)$ (n is the number of stacks).
    - Sankoff's algorithm is $O(m^6)$, (m is the length of the sequences).
    - The number of possible stacks (size >= 4) is much smaller than the length of the sequence.
    - It's much faster.

For any pair of stacks, there are three choices:



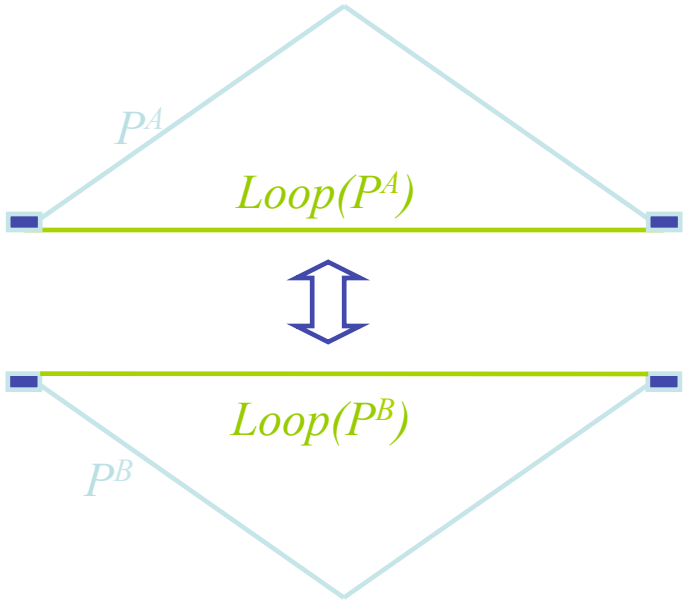
$$M[P^A, P^B] = M_s[P^A, P^B] + \min \left\{ \begin{array}{ll} M_h[P^A, P^B], & \text{(* hairpin loop *)} \\ M_b[P^A, P^B], & \text{(* interior loop/bulge *)} \\ M_m[P^A, P^B] & \text{(* multi-loop *)} \end{array} \right\}$$

The score of matching stacks:



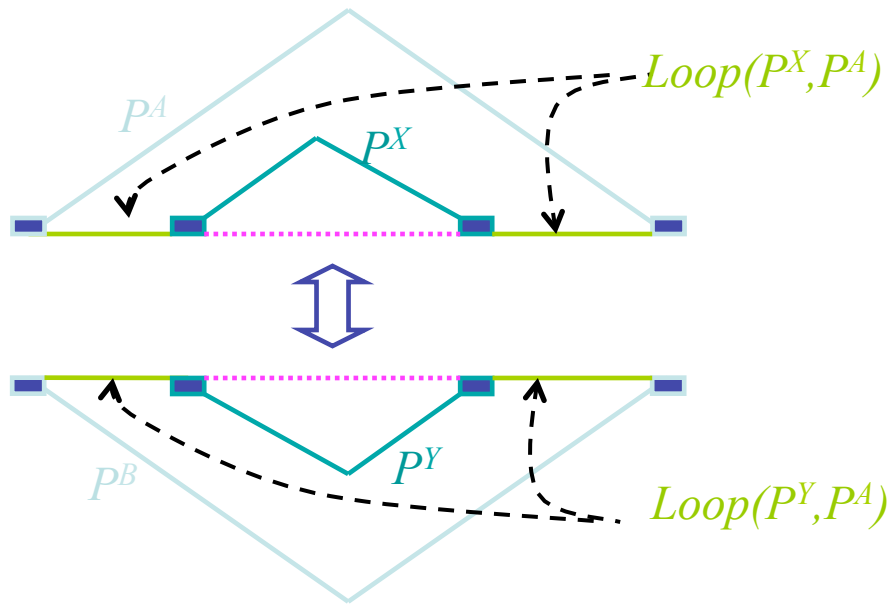
$$M_s[P^A, P^B] = w_1 \max \left\{ \begin{array}{c} \mathcal{E}_s(P^A), \\ \mathcal{E}_s(P^B) \end{array} \right\} + w_2 \mathcal{S} \left(\begin{array}{c} \mathrm{Seq}(P^A) \\ \mathrm{Seq}(P^B) \end{array} \right)$$

The score of matching hairpin loops:



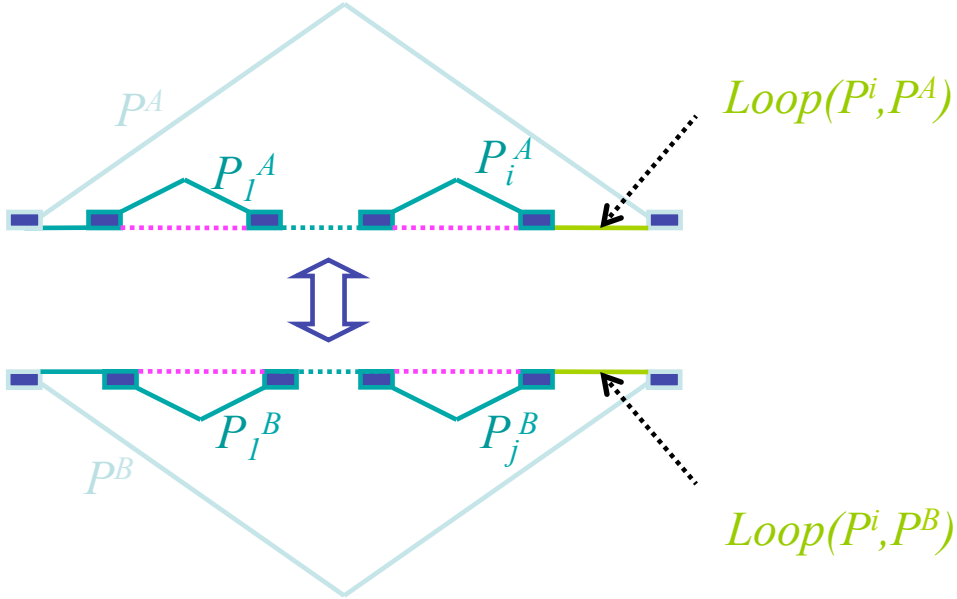
$$M_h[P^A, P^B] = \boxed{w_1 \max \left\{ \begin{array}{c} \mathcal{E}_h(|\mathrm{Loop}(P^A)|), \\ \mathcal{E}_h(|\mathrm{Loop}(P^B)|) \end{array} \right\} + w_3 \mathcal{S} \left(\begin{array}{c} \mathrm{Loop}(P^A), \\ \mathrm{Loop}(P^B) \end{array} \right)}$$

The score of matching interior loops or bulges:



$$M_b[P^A, P^B] = \min_{\substack{P^x <_I P^A, \\ P^y <_I P^B}} \left\{ \boxed{w_1 \max \left\{ \begin{array}{l} \mathcal{E}_b(|\text{Loop}(P^x, P^A)|), \\ \mathcal{E}_b(|\text{Loop}(P^y, P^B)|) \end{array} \right\} + w_3 \mathcal{S} \left(\begin{array}{l} \text{Loop}(P^x, P^A), \\ \text{Loop}(P^y, P^B) \end{array} \right)} + \boxed{M[P^x, P^y]} \right\}$$

The score of matching two multi-loops:



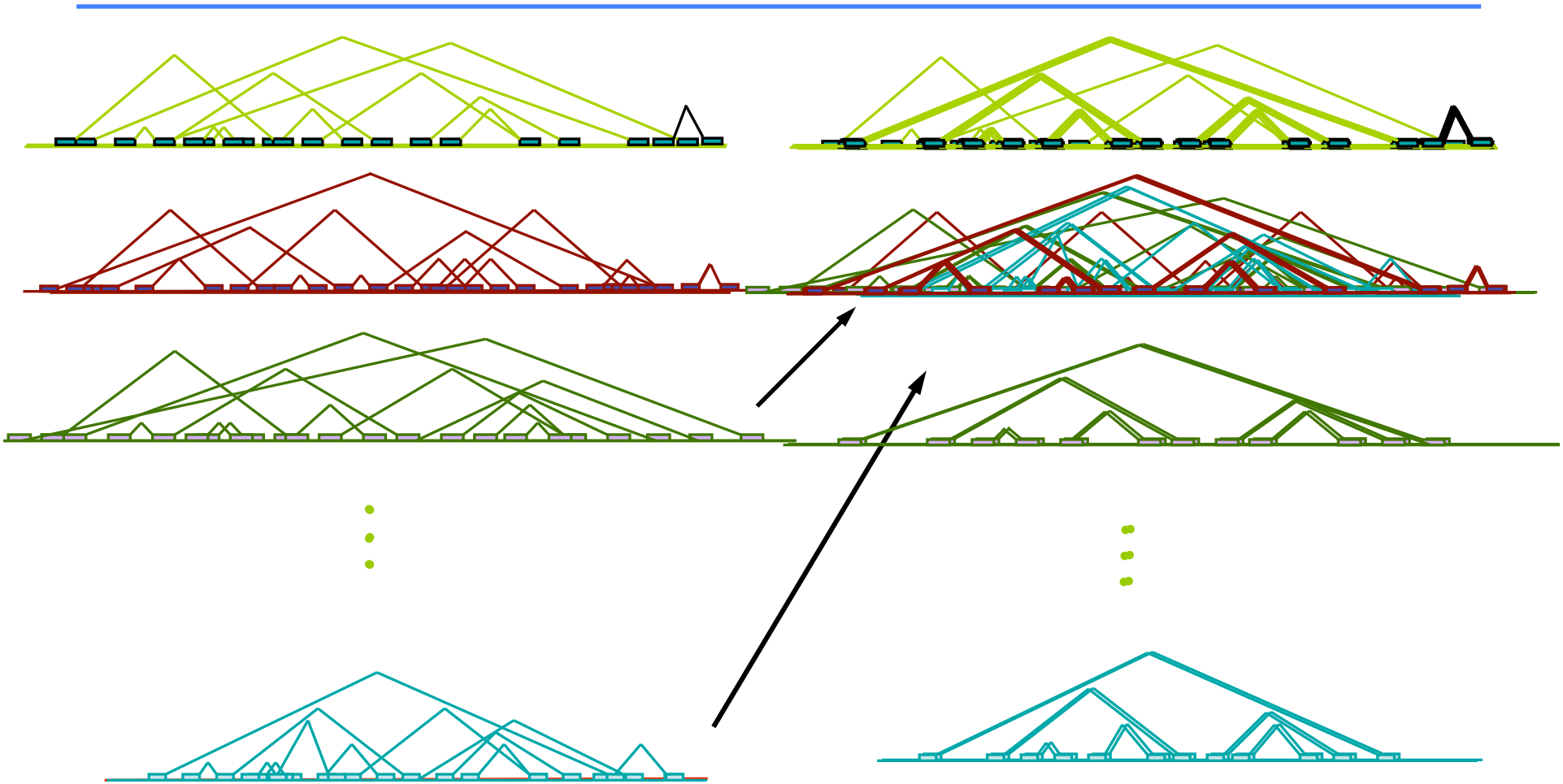
$$M_m[P^A, P^B] = \min_{i,j} \left\{ \boxed{M_c[P_i^A, P_j^A]} + \boxed{w_1 \max \left\{ \begin{array}{l} \mathcal{E}_m(|\mathrm{Loop}(P^i, P^A)|), \\ \mathcal{E}_m(|\mathrm{Loop}(P^j, P^B)|) \end{array} \right\} + w_3 \mathcal{S} \left(\begin{array}{l} \mathrm{Loop}(P^i, P^A), \\ \mathrm{Loop}(P^j, P^B) \end{array} \right)} \right\}$$

Consensus folding for multiple sequences

$$M(\mathcal{P}(A_1,\ldots,A_s)) = w_1\Phi(\mathcal{P}(A_1,\ldots,A_s)) + w_2\sum_{j\in\mathcal{P}}\mathcal{S}\begin{pmatrix}A_{1,j},\\ A_{2,j},\\ \ldots,\\ A_{s,j}\end{pmatrix} + w_3\sum_{j\notin\mathcal{P}}\mathcal{S}\begin{pmatrix}A_{1,j},\\ A_{2,j},\\ \ldots,\\ A_{s,j}\end{pmatrix}$$

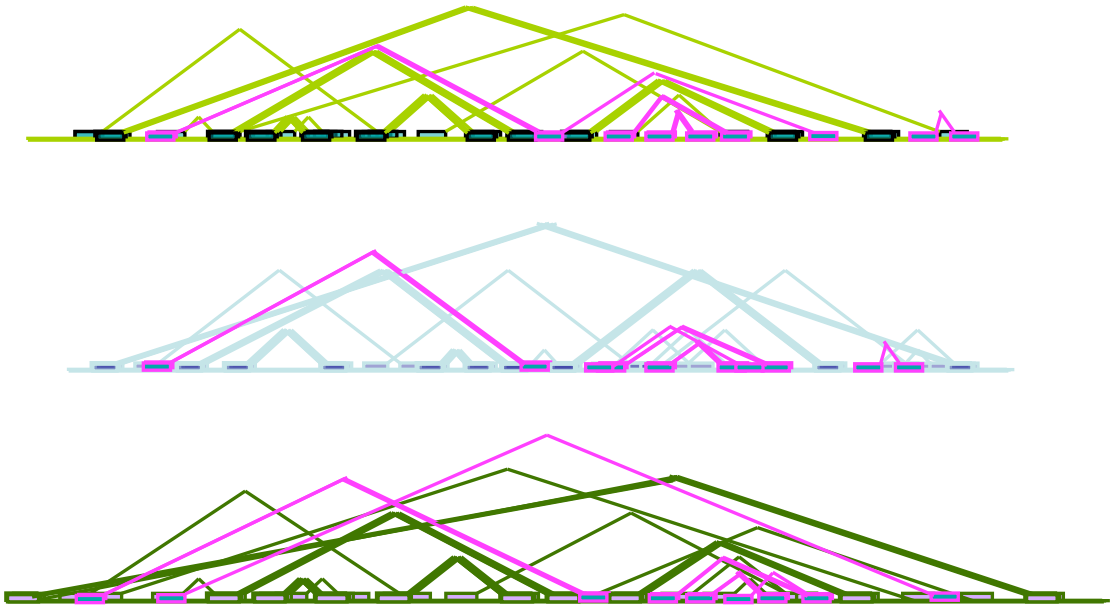
- We use a heuristic method based on the notion of star-alignment.
  - Compute an optimal configuration from a random seed pair.
  - Align all individual sequences to this configuration.
  - Choose the conserved stack configuration in all sequences.
  - Allow some stacks to be partially conserved (at least appear in a certain fraction of the sequences).

Compute the stack configuration for multiple sequences:
RNAscf(k,h,f)



Iterative procedure for **RNAscf**

1. P = RNAscf(k, h, f).
2. In each sequence, extract the unpaired regions according to the loop regions in P.
3. Predict additional putative stacks that are not crossing with P using smaller k' and h'.
4. Recompute the alignment for with additional putative stacks using RNAscf(k',h',f).



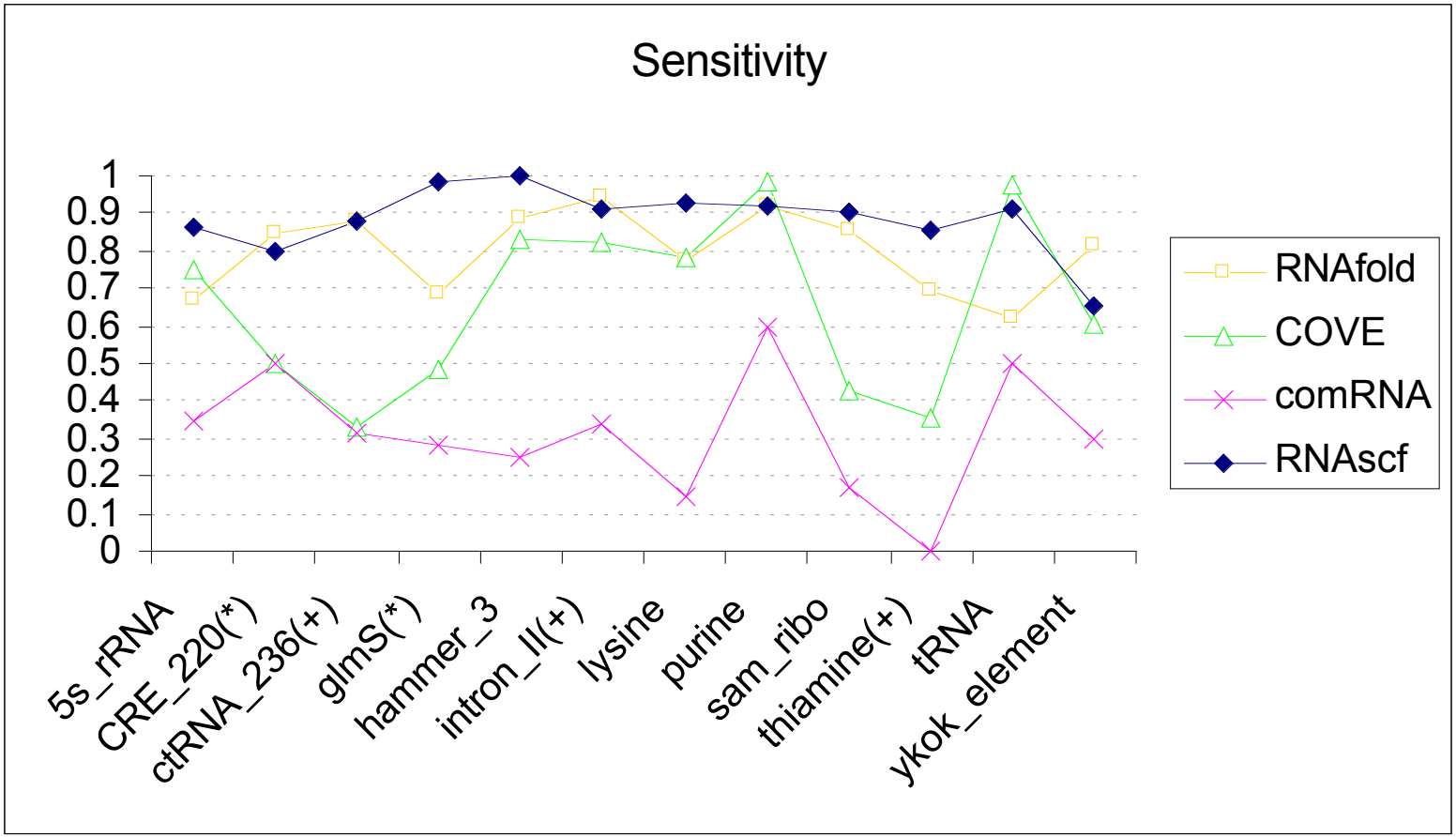
Test dataset

- We choose a set of 12 RNA families from Rfam database:
  - 20 sequences chosen from the families. (except for CRE and glms, we choose 10 sequences) with annotated structures.
  - There are 953 stacks.
  - We compare **RNAscf** with 3 other programs that are available online for RNA folding:
    - **RNAfold** (energy based minimization) (Hofacker 2003)
    - **COVE** (covariance model) (Eddy and Durbin 1994)
      - Cove need a staring seed alignment which is produced by **ClustalW**.
    - **comRNA** (computing anchors in multiple sequences) (Ji, Xu and Stormo 2004).
  - *Sensitivity*: the fraction of true stacks that overlapped with predicted stacks.
  - *Accuracy*: the fraction of predicted stacks that overlapped with true stacks

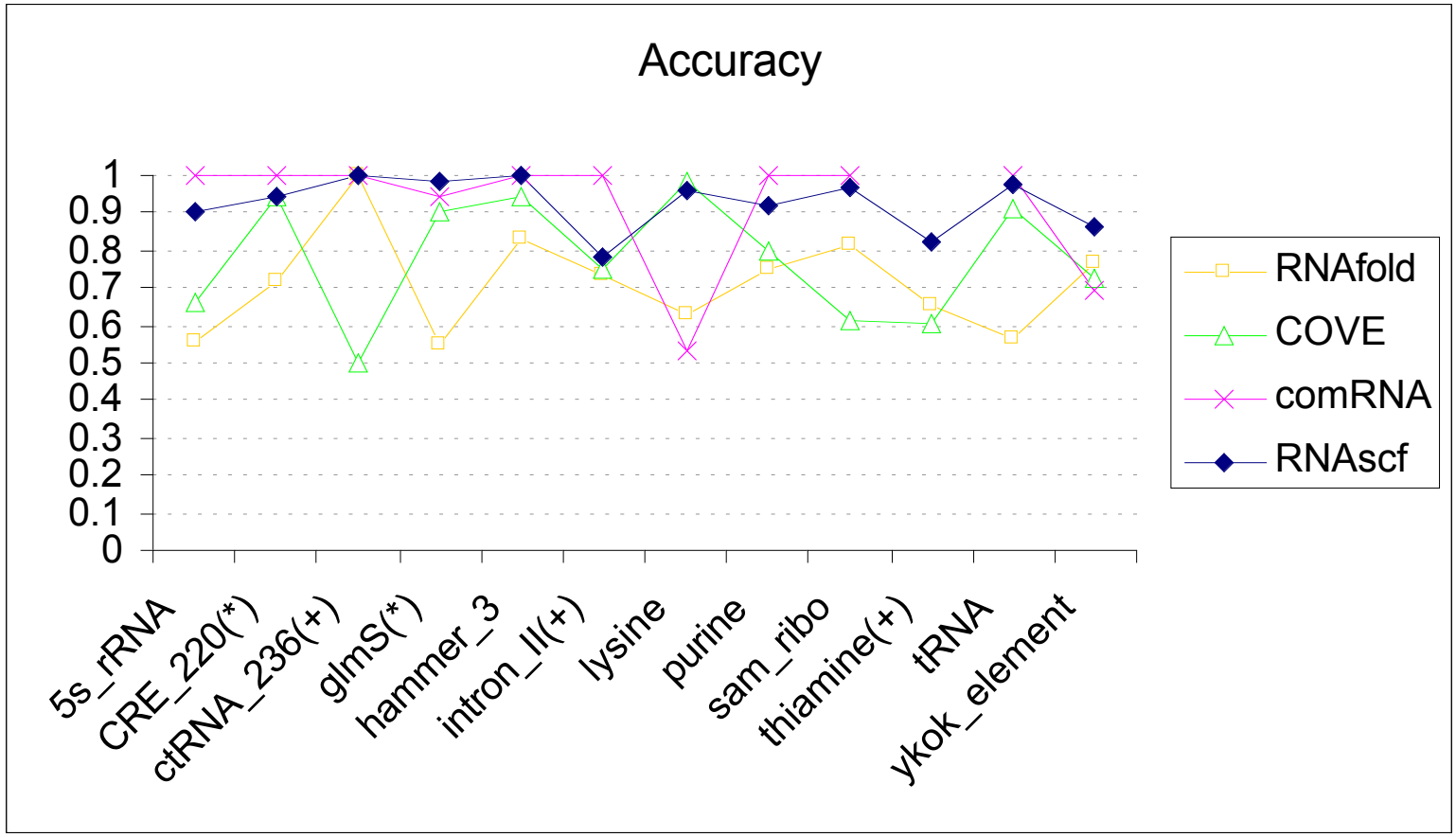
Test results

Name (Rfam_id)	Stacks	Sensitivity				Accuracy			
		RNAscf	RNAfold	Cove	comRNA	RNAscf	RNAfold	Cove	comRNA
5s_rRNA (RF00001)	100	0.86	0.67	0.75	0.35	0.9	0.558	0.658	1
Rhino_CRE (RF00220)	20	0.8	0.85	0.5	0.5	0.941	0.72	0.941	1
ctRNA_pGA1 (RF00236)	51	0.882	0.882	0.333	0.313	1	1	0.5	1
glmS(RF00234)	60	0.983	0.683	0.483	0.283	0.983	0.552	0.906	0.944
Hammerhead_3 (RF00008)	60	1	0.883	0.833	0.25	1	0.828	0.943	1
Intron_gpII (RF00029)	56	0.91	0.946	0.821	0.339	0.782	0.731	0.754	1
Lysine (RF00168)	120	0.925	0.775	0.783	0.142	0.958	0.633	0.984	0.531
Purine (RF00167)	60	0.917	0.917	0.983	0.6	0.917	0.753	0.797	1
Sam_riboswitch (RF00162)	113	0.903	0.858	0.425	0.168	0.966	0.813	0.613	1
Thiamine (RF00059)	80	0.858	0.690	0.354	0	0.824	0.654	0.606	-
tRNA (RF00005)	113	0.912	0.625	0.975	0.5	0.973	0.567	0.910	1
ykok (RF00380)	180	0.656	0.817	0.606	0.3	0.863	0.762	0.727	0.692
Average		**0.884**	0.8	0.654	0.312	**0.926**	0.714	0.778	0.924

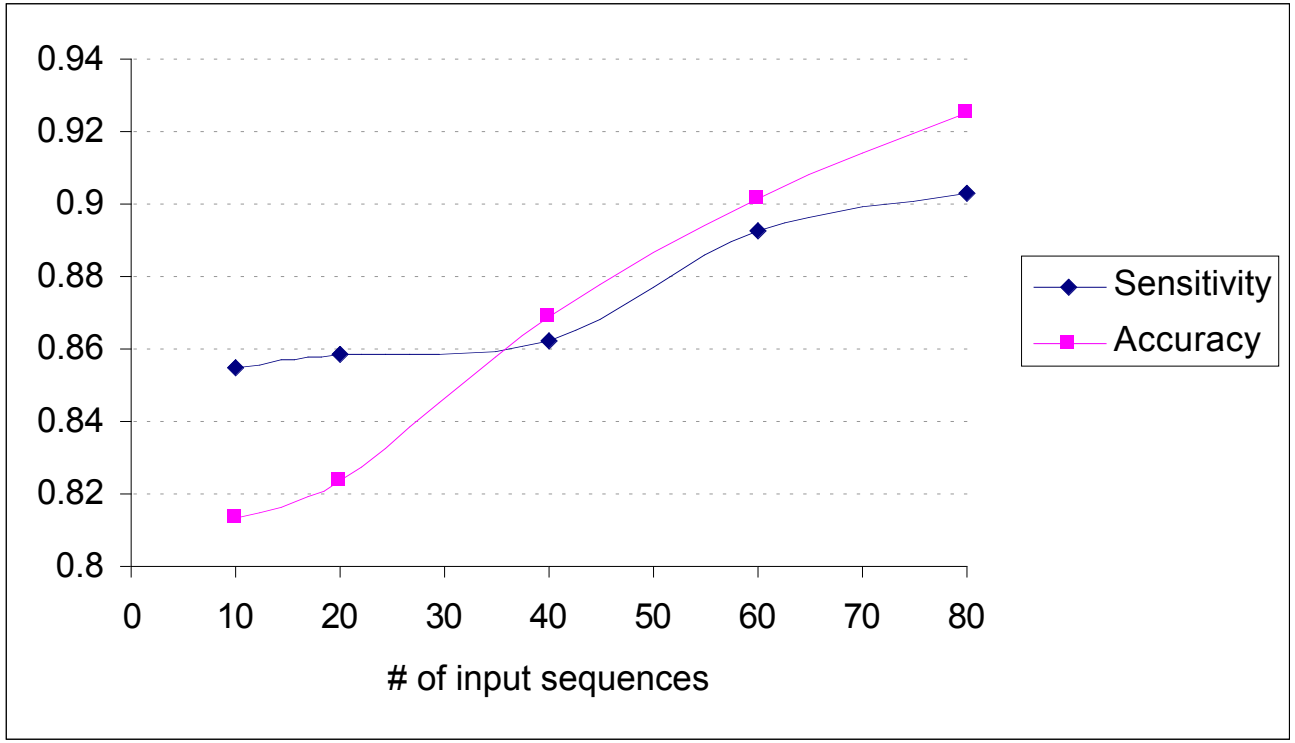
Test results



Test results

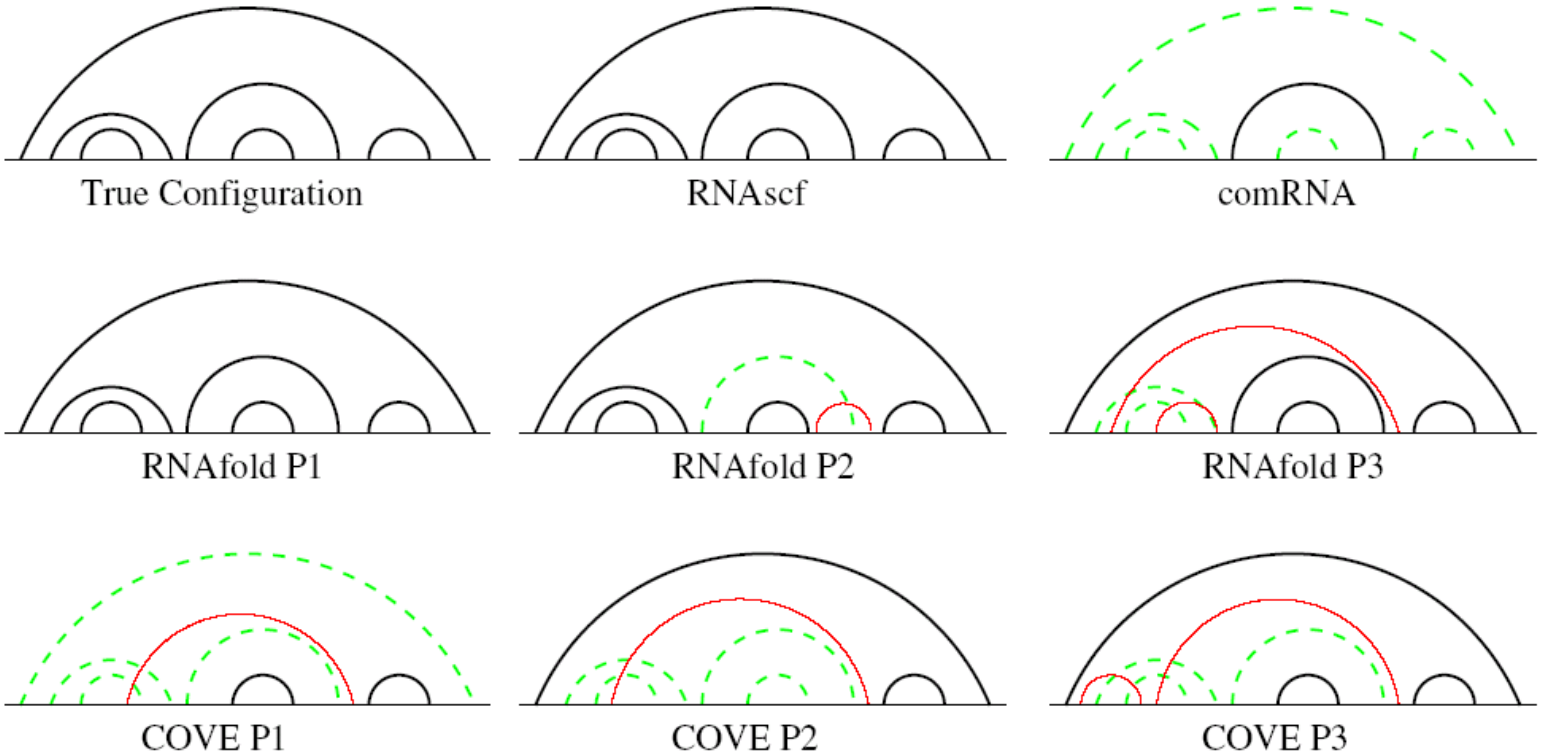


Performance improves when the number of sequences increases



(Using Thiamine riboswitch subfamily (RF00059))

RNAscf always finds the right consensus stack configuration.

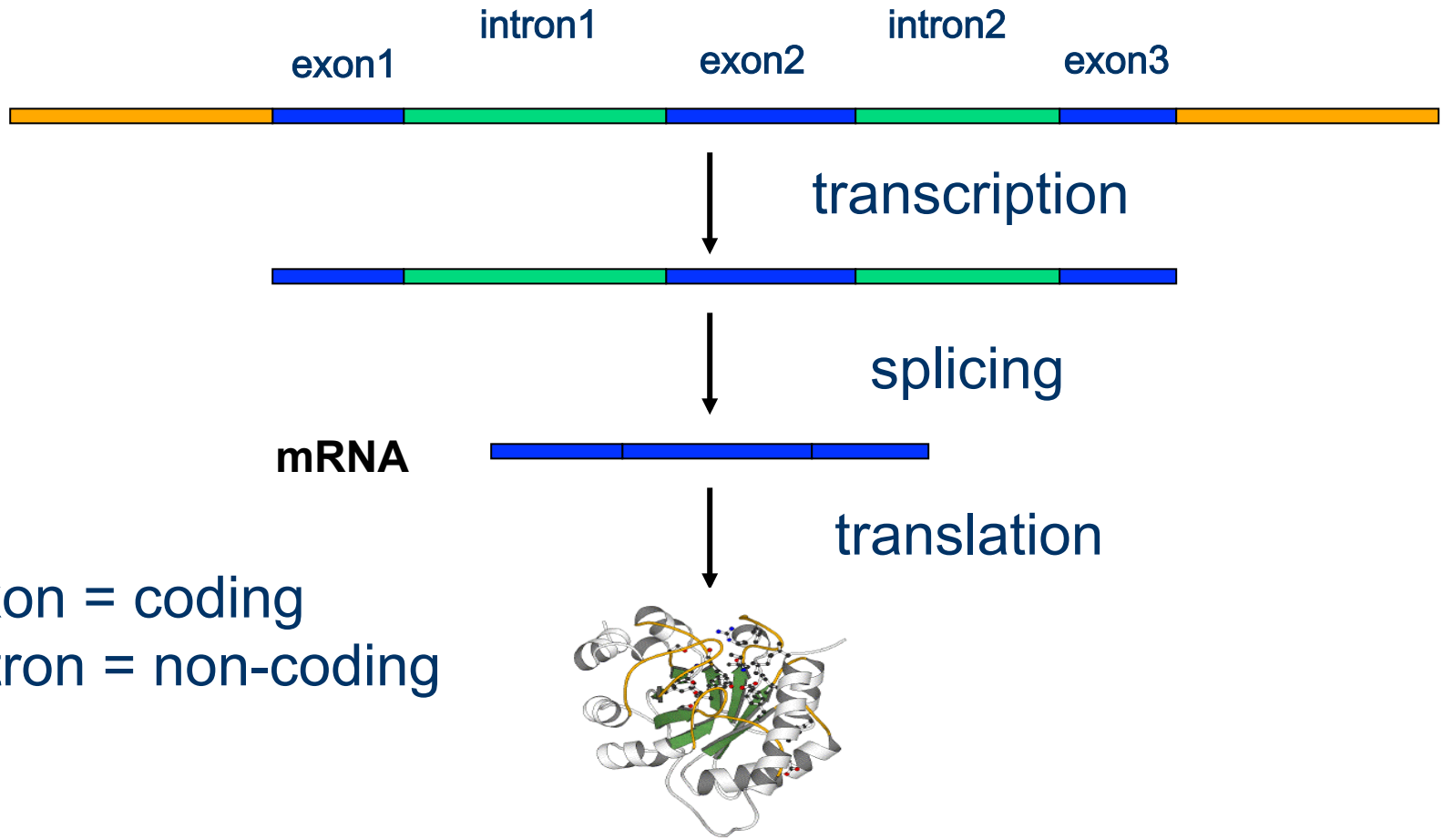


(Sam riboswitch (RF00162))

Conclusion and future work

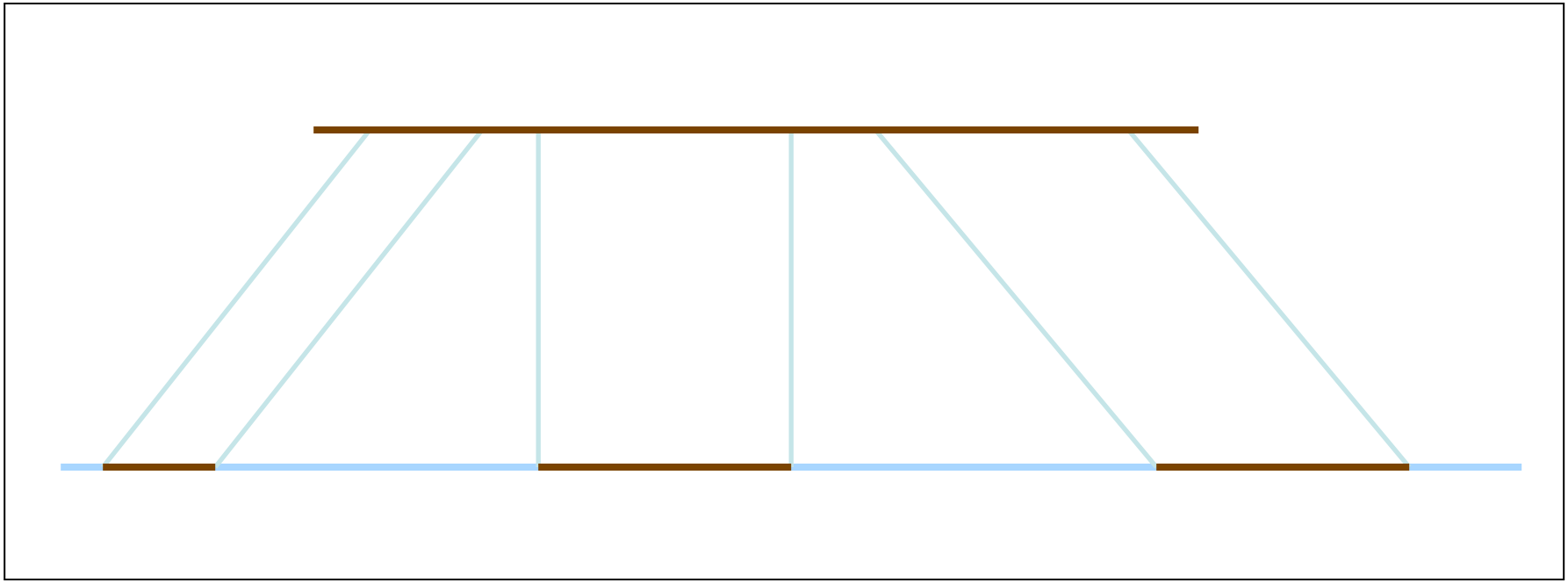
- **RNAscf** is a valid approach to RNA consensus structure prediction.
  - Use stack configuration to represent RNA secondary structure.
  - Propose a dynamic programming algorithm to find optimal stack configuration for pairwise sequences.
  - Use both primary sequence information and energy information.
  - Use a star-alignment-like heuristic method to get the consensus structure for multiple sequences.
- Future work:
  - Correcting errors by using a stochastic iterative scheme (such as Gibbs sampling).
  - Provide P-value for each prediction.
  - Use RNAscf to find new families of ncRNAs.
  - Perform constraint folding to refine the predicted structure by adding some minor basepairs.

Central Dogma and Splicing



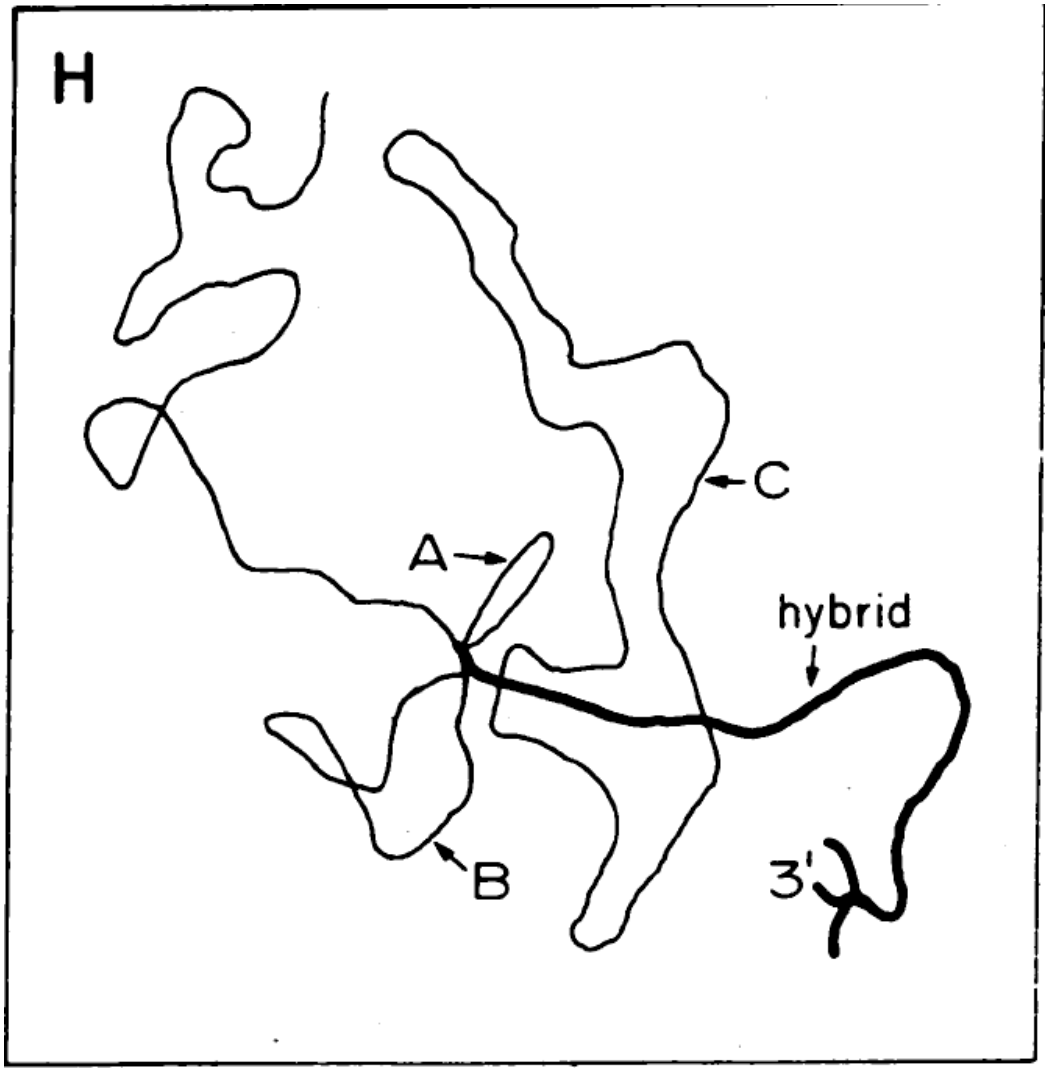
exon = coding
intron = non-coding

Comparing genes in two genomes



- Small islands of similarity corresponding to similarities between exons

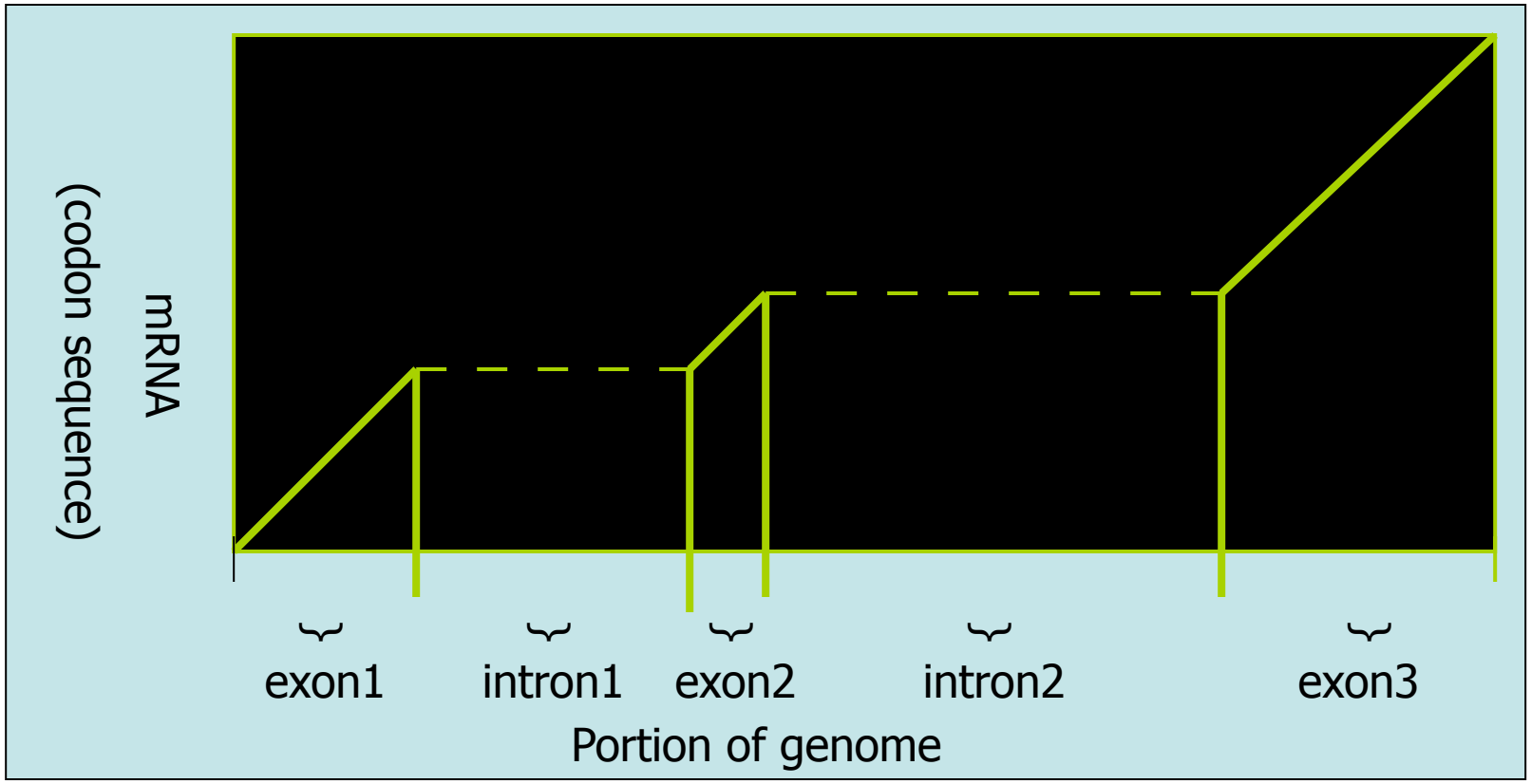
Berget et al. (1977)



Gene finding

- **Problem:** Given a known mRNA sequence (e.g. in mouse) and an un-annotated genome sequence (e.g. in human), find a set of substrings of the genomic sequence whose concatenation best fits the mRNA (which is the homologous gene of the mouse gene in human).

Spliced sequence alignment

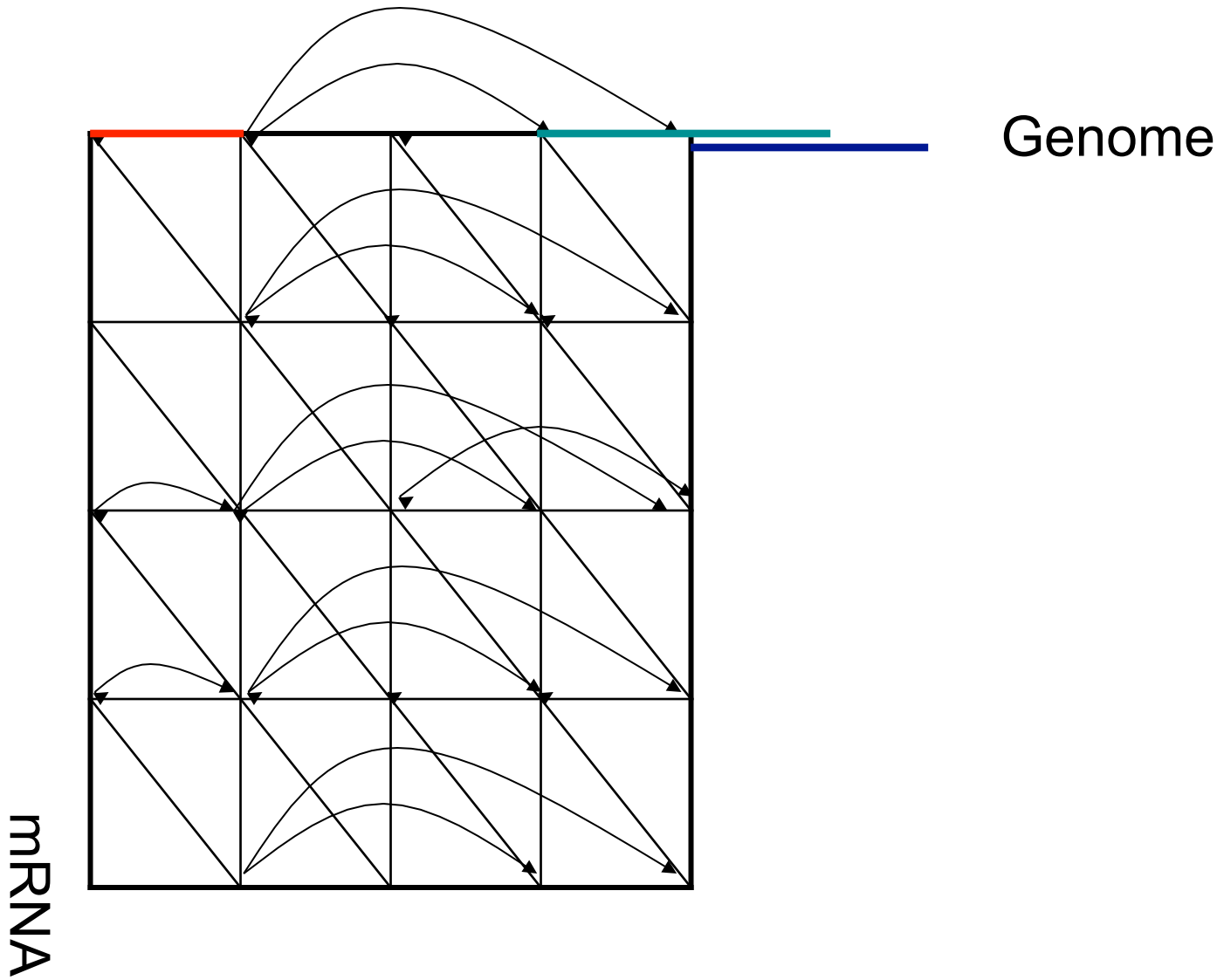


Spliced alignment problem: formulation

- **Goal**: Find a chain of blocks (putative exons) in a genomic sequence that best fits a target sequence
- **Input**: Genomic sequences *G*, target mRNA sequence *T*, and a set of putative exons *B*.
- **Output**: A chain of exons Γ such that the global alignment score between Γ* and T is maximum among all chains of blocks from *B*.

Γ* - concatenation of all exons from chain Γ

“Splicing gap” edges to the alignment graph



Spliced alignment recurrence

If *i* is not the starting vertex of block *B*:

- $S(i, j, B) =$

$$\max \{ \; S(i-1, j, B) - \textit{indel penalty} \quad S(i, j-1, B) - \textit{indel penalty} \quad S(i-1, j-1, B) + \delta(g_i, t_j) \}$$

If *i* is the starting vertex of block *B*:

- $S(i, j, B) =$

$$\max \{ \; S(i, j-1, B) - \textit{indel penalty} \quad \max_{\text{all blocks } B' \text{ preceding block } B} S(end(B'), j, B') - \textit{indel penalty} \quad \max_{\text{all blocks } B' \text{ preceding block } B} S(end(B'), j-1, B') + \delta(g_i, t_j) \}$$

Spliced Alignment Solution

- After computing the three-dimensional table *S(i, j, B)*, the score of the optimal spliced alignment is:

$$\max_{\text{all blocks } B} S(end(B), length(T), B)$$

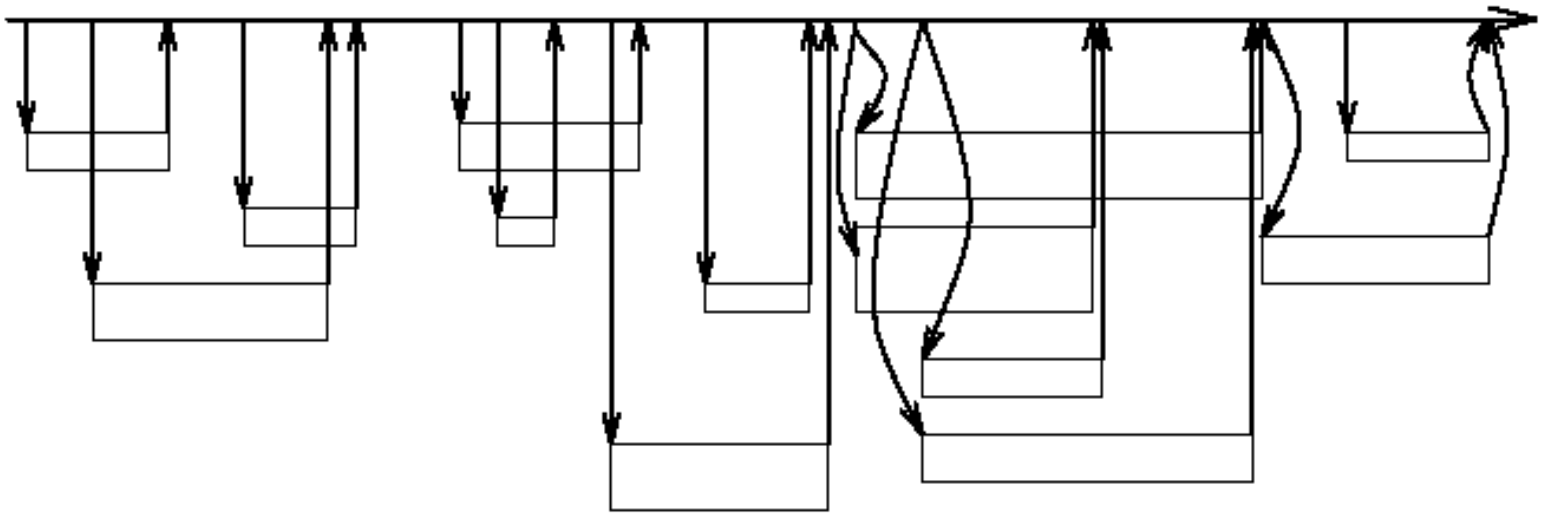
Spliced alignment: complexity

- Considering multiple *i*-prefixes leads to slow down. running time:

$$O(mn^2 |B|)$$

where m is the target length, n is the genomic sequence length and $|B|$ is the number of blocks.

Spliced alignment: a special case of network matching problem



- Input: a Directed Acyclic Graph (DAG, or network), G, with edges labeled by strings; and a string S.
- Output: a path in G, with the labeling sequence best fits S.

Spliced alignment of two genome sequences

- Input: two Directed Acyclic Graphs (DAG, or network), G_1 and G_2.
- Output: paths P_1 and P_2 in two graphs, respectively, with the labeling sequences best fits each other.

- Example: align entire human and mouse genomes while predicting genes in both sequences simultaneously as chains of aligned blocks (exons)

- This approach does not assume any annotation of either human or mouse genes.

Hard problems

Generalizing pairwise alignment

- Alignment of 2 sequences is represented as a 2-row matrix
- In a similar way, we represent alignment of 3 sequences as a 3-row matrix

```
A T _ G C G _
A _ C G T _ A
A T C A C _ A
```

- Score: more conserved columns, better alignment

Sum of pairs (SP) score

- Consider pairwise alignment of sequences

  a_i and a_j

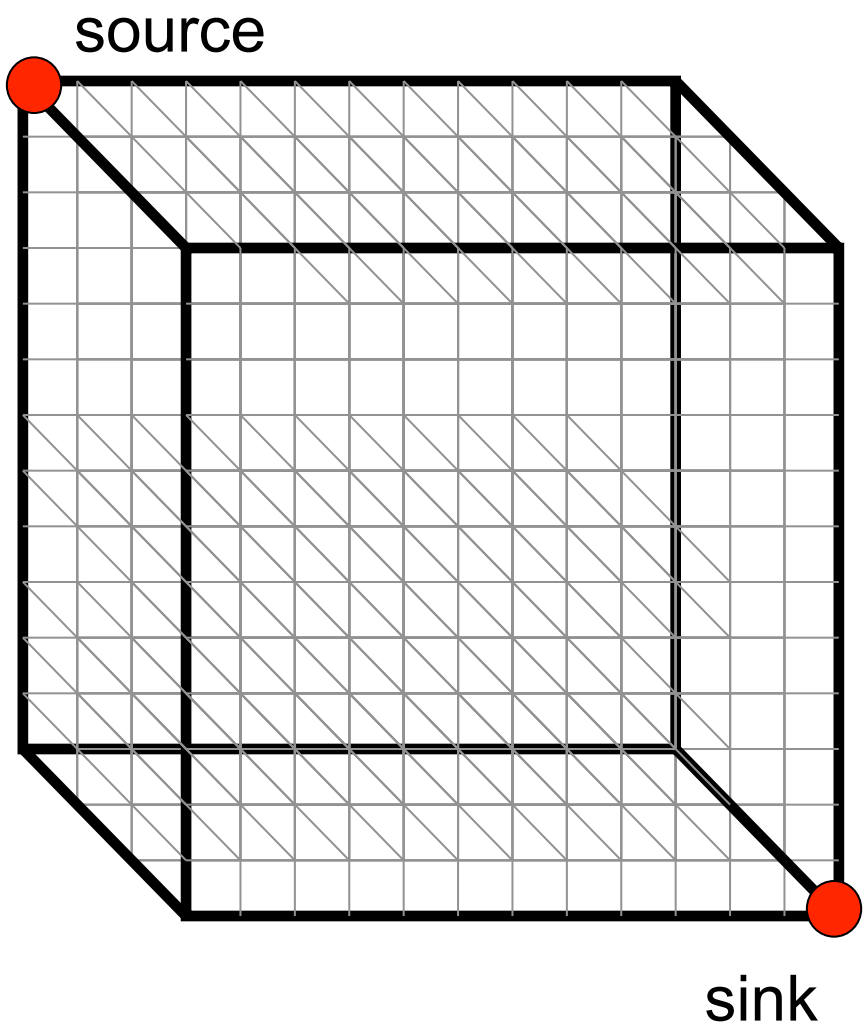
  imposed by a multiple alignment of k sequences
- Denote the score of this suboptimal (not necessarily optimal) pairwise alignment as

  $$s^*(a_i, a_j)$$
- Sum up the pairwise scores for a multiple alignment:

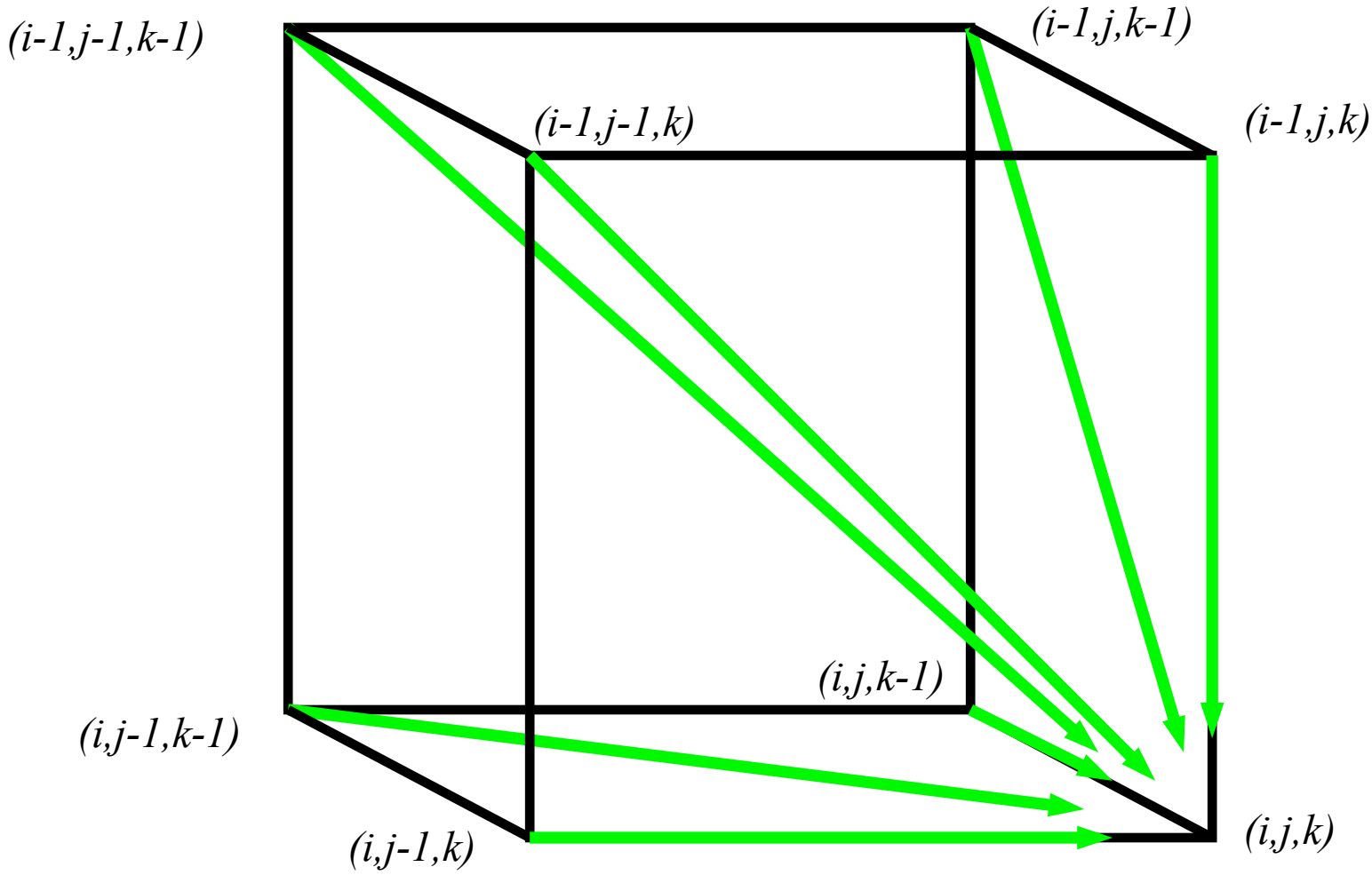
  $$s(a_1, \ldots, a_k) = \Sigma_{i,j}\, s^*(a_i, a_j)$$

Aligning three sequences

- Same strategy as aligning two sequences
- Use a 3-D alignment graph, with each axis representing a sequence to align



Architecture of 3-D Alignment Cell



Multiple alignment: dynamic programming

- $$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(v_i, w_j, u_k) & \text{cube diagonal: no indels} \\ s_{i-1,j-1,k} + \delta(v_i, w_j, _) & \\ s_{i-1,j,k-1} + \delta(v_i, _, u_k) & \text{face diagonal: one indel} \\ s_{i,j-1,k-1} + \delta(_, w_j, u_k) & \\ s_{i-1,j,k} + \delta(v_i, _, _) & \\ s_{i,j-1,k} + \delta(_, w_j, _) & \text{edge diagonal: two indels} \\ s_{i,j,k-1} + \delta(_, _, u_k) & \end{cases}$$

- $\delta(x, y, z)$ is an entry in the 3-D scoring matrix

Multiple alignment: complexity

- For 3 sequences of length $\boldsymbol{n}$, the run time is $7\boldsymbol{n}^3$; $O(\boldsymbol{n}^3)$
- For $\boldsymbol{k}$ sequences, build a $\boldsymbol{k}$-dimensional Manhattan, with run time $(2^k-1)(\boldsymbol{n}^k)$; $O(2^k\boldsymbol{n}^k)$
- Conclusion: dynamic programming approach for alignment between two sequences is easily extended to $\boldsymbol{k}$ sequences but it is exponential to $\boldsymbol{k}$.
- Classification of hard problems.

Decision problems vs. optimization problems

- *Decision problems*: determine whether a given statement is true or false
- *Optimization problems*: find the solution with the best possible score according to some scoring scheme
  - Minimization problems
  - Maximization problems

Hamiltonian cycles

Given a directed graph, we want to decide whether or not there is a *Hamiltonian cycle* in this graph. This is a decision problem.

Hamiltonian cycle: a circular path that visits each node once and exactly once.

TSP - Traveling Salesman Problem

Given a complete graph and an assignment of weights to the edges, find a Hamiltonian cycle of minimum weight.

This is the *optimization version* of the problem. In the *decision version*, we are given a weighted complete graph and a real number *c,* and we want to know whether or not there exists a Hamiltonian cycle whose combined weight of edges does not exceed *c.*

Minimum spanning tree problem

Given a connected graph and an assignment of weights to the edges, find a spanning tree of minimum weight.

Minimum spanning tree: a subgraph which is a tree and connects all the vertices together.

This is the *optimization version* of the problem. In the *decision version*, we are given a weighted connected graph and a real number *c,* and we want to know whether or not there exists a spanning tree whose combined weight of edges does not exceed *c.*

Important observation

- Each optimization problem has a corresponding decision problem.

The class P

A decision problem *D* is *solvable in polynomial time* (or *in the class P*), if there exists an algorithm *A* such that

- *A* takes instances of *D* as inputs.
- *A* always outputs the correct answer “Yes” or “No”.
- There exists a polynomial *p* such that the execution of *A* on inputs of size *n* always terminates in *p(n)* or fewer steps.

The class P

EXAMPLE: The Minimum Spanning Tree Problem is in the class P. → The greedy algorithm is a correct algorithm.

The class *P* is often considered as synonymous with the class of computationally feasible problems, although in practice some polynomial algorithms are unrealistic.

Witnesses for decision problems

Witness: the instance to a decision problem is "yes", e.g., in the Hamiltonian cycle problem, any permutation $v_1, v_2, \ldots, v_n$ of the vertices of the input graph is a *potential witness.* This potential witness is a *true witness* if v_1 is adjacent to v_2 , … and v_n is adjacent to v_1.

The class NP

A decision problem is *non-deterministically polynomial-time solvable* or *in the class NP* if there exists an algorithm *A* such that

- *A* takes as inputs potential witnesses for "yes" answers to problem *D.*
- *A* correctly distinguishes true witnesses from false witnesses.
- There exists a polynomial *p* such that for each potential witnesses of each instance of size *n* of *D,* the execution of the algorithm *A* takes at most *p(n)* steps.

The class NP

Note that if a problem is in the class *NP*, then we are able to verify “yes”-answers in polynomial time, if we are provided a true witness.

The P=NP Problem

Are the classes *P* and *NP* identical? This is an open problem. It is not hard to show that every problem in *P* is also in *NP,* but it is unclear whether every problem in *NP* is also in *P.*

The P=NP Problem

Thousands of computer scientists have been unsuccessful for decades to design polynomial-time algorithms for some problems in the class *NP.*

This constitutes overwhelming *empirical* evidence that the classes *P* and *NP* are indeed distinct, but no formal mathematical proof of this fact is known.

Polynomial-time reducibility

Let *E* and *D* be two decision problems. We say that *D* is *polynomial-time reducible to E* if there exists an algorithm *A* such that

- *A* takes instances of *D* as inputs and always outputs the correct answer “Yes” or “No” for each instance of *D.*
- *A* uses as a subroutine a hypothetical algorithm *B* for solving *E.*
- There exists a polynomial *p* such that for every instance of *D* of size *n* the algorithm *A* terminates in at most *p(n)* steps *if each call of the subroutine B is counted as only m steps, where m is the size of the actual input of B.*

An example of polynomial-time reducibility

Theorem: The Hamiltonian cycle problem is polynomial-time reducible to the decision version of TSP.

Proof: Given an instance G with vertices $v_1, \ldots, v_n$ of the Hamiltonian cycle problem, let H be the weighted complete graph on $v_1, \ldots, v_n$ such that the weight of an edge $\{v_i, v_j\}$ in H is 1 if $\{v_i, v_j\}$ is an edge in G, and is 2 otherwise. Now the correct answer for the instance G of the Hamiltonian cycle problem can be obtained by running an algorithm on the instance $(H,n+1)$ of the TSP.

NP-complete problems

A decision problem *E* is *NP-complete* if *every* problem in the class *NP* is polynomial-time reducible to *E.* The Hamiltonian cycle problem, the decision versions of the TSP and literally hundreds of other problems are known to be NP-complete.

To show a decision problem E is NP-complete, you need to show E is in class NP, and one problem known to be NP-complete is polynomial-time reducible to *E.*

NP-hard problems

Optimization problems whose decision versions are NP-complete are called *NP-hard.*

Theorem: If there exists a polynomial-time algorithm for finding the optimum in any *NP*-hard problem, then *P* = *NP.*

NP-hard problems

Proof: Let *E* be an *NP*-hard optimization (let us say minimization) problem, and let *A* be a polynomial-time algorithm for solving it. Now an instance *J* of the corresponding decision problem *D* is of the form *(I,c),* where *I* is an instance of *E,* and *c* is a number. Then the answer to *D* for instance *J* can be obtained by running *A* on *I* and checking whether the cost of the optimal solution exceeds *c.* Thus there exists a polynomial-time algorithm for *D,* and *NP*-completeness of the latter implies *P*= *NP.*

Duality

- For many optimization problems, there is a ***dual*** problem associated with the original (***primal***) problem.
- The relationship between the primal and dual problems are useful in a variety of ways, e.g.
  - helps to find efficient algorithms for solving the primal problem;
  - provides arguments for proving the optimality (or close-to-optimality) of a primal solution.

Consequences for bioinformatics

In view of the overwhelming empirical evidence against the equality *P*= *NP* it seems that no *NP*-hard optimization problem is solvable by an algorithm that is *guaranteed* to:

- run in polynomial time *and*
- *always* produce a optimal solution.

Unfortunately, many, perhaps most, of the important optimization problems in bioinformatics are *NP*-hard. To make matters worse, the instances of interest in bioinformatics are typically of large size.

What can we do about these problems?

Hard problems in bioinformatics

- Multiple sequence alignment problem
- Protein threading / design problem
- Map / sequence assembly problem
- Many others…

Alternative performance measures?

So far we have been talking about algorithms that

- run in polynomial time on *all* instances
- always find the solution with the *best* score/cost.

Worst case vs. average performance

For many practical purposes, it may be sufficient to have an algorithm whose *average* running time for instances of size n is bounded by a polynomial. Such an algorithm may still be unacceptably slow for some particularly bad instances, but such bad instances will necessarily be very rare and may be of little practical relevance.

Approximation algorithms

While optimal solutions to optimization problems are clearly best, “reasonably” good solutions are also of value.

Let us say that an algorithm for a minimization problem D has a *performance guarantee of* $1 + \varepsilon$ if for each instance I of the problem it finds a solution whose cost is at most $(1 + \varepsilon)$ times the cost of the optimal solution for instance I. While D may be *NP*-hard, it may still be possible to find, for some $\varepsilon > 0$, polynomial-time algorithms for D with performance guarantee $1 + \varepsilon$. Such algorithms are called *approximation algorithms.*

Polynomial-time approximation schemes

- A minimization problem *D* has a *polynomial-time approximation scheme (PTAS)* if for every $\varepsilon > 0$ there exists a polynomial-time algorithm for *D* with performance guarantee $1 + \varepsilon$.

- *Even if D* may be *NP*-hard, it may still have a PTAS.

Heuristic algorithms

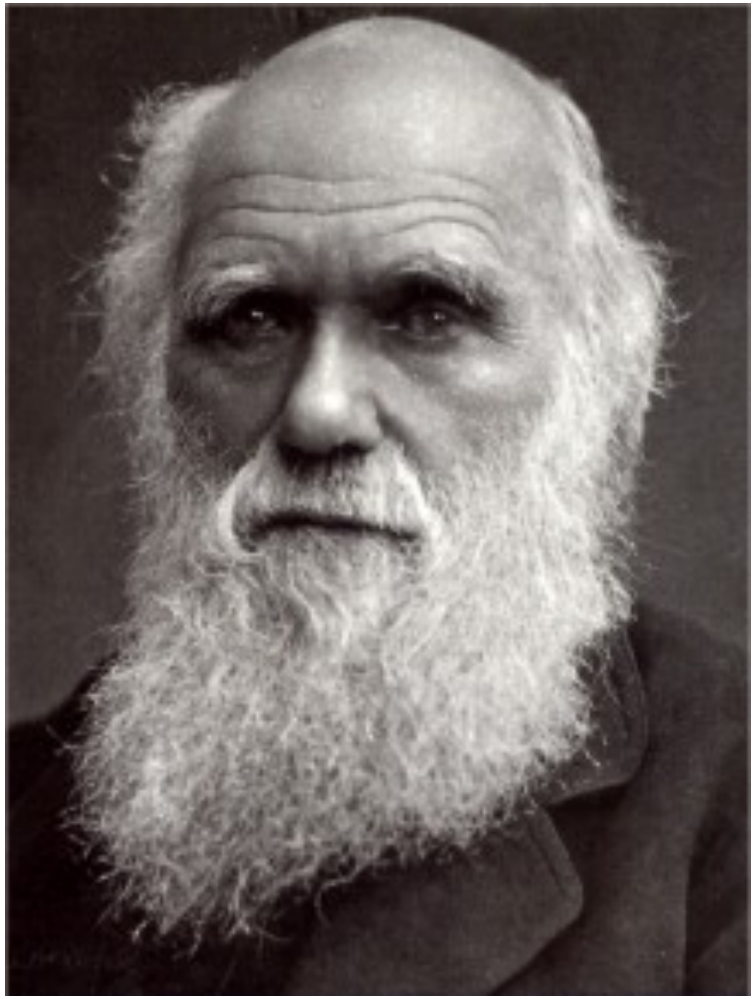
- reasonably fast on the average instance
- most of the time find solutions within $(1 + \varepsilon)$ of optimum for reasonably small ε
- It is not always easy or possible to mathematically analyze the performance of a heuristic algorithm

Algorithmic problems related to phylogenetic trees

Evolution

Evolution of new organisms is driven by

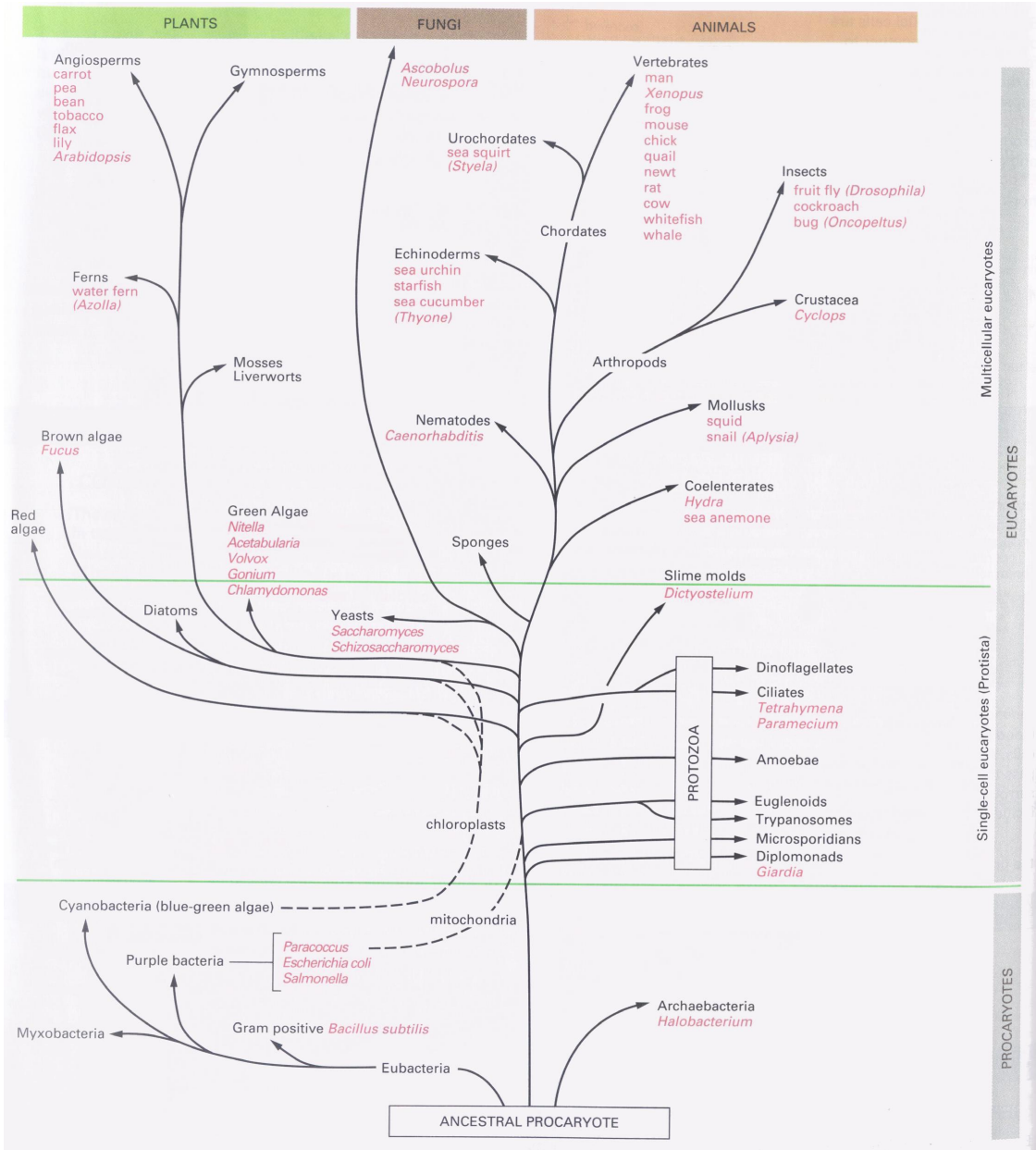
- **Mutations**
  - The DNA sequence can be changed due to single base changes, deletion/insertion of DNA segments, etc.
- **Selection bias**



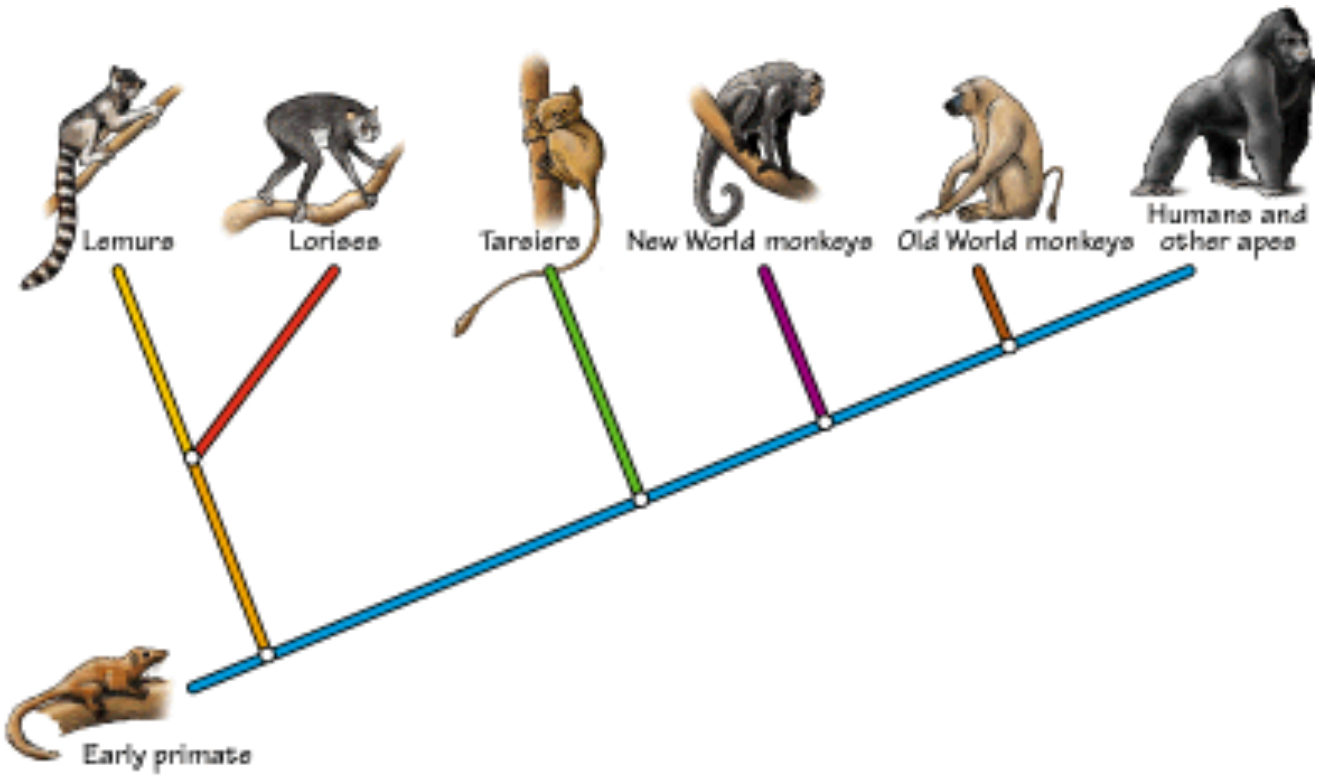
Theory of evolution

- Basic idea
  - **speciation** events lead to creation of different species.
  - Speciation caused by physical separation into groups where different genetic variants become dominant
- Any two species share a (possibly distant) common ancestor

The Tree of Life



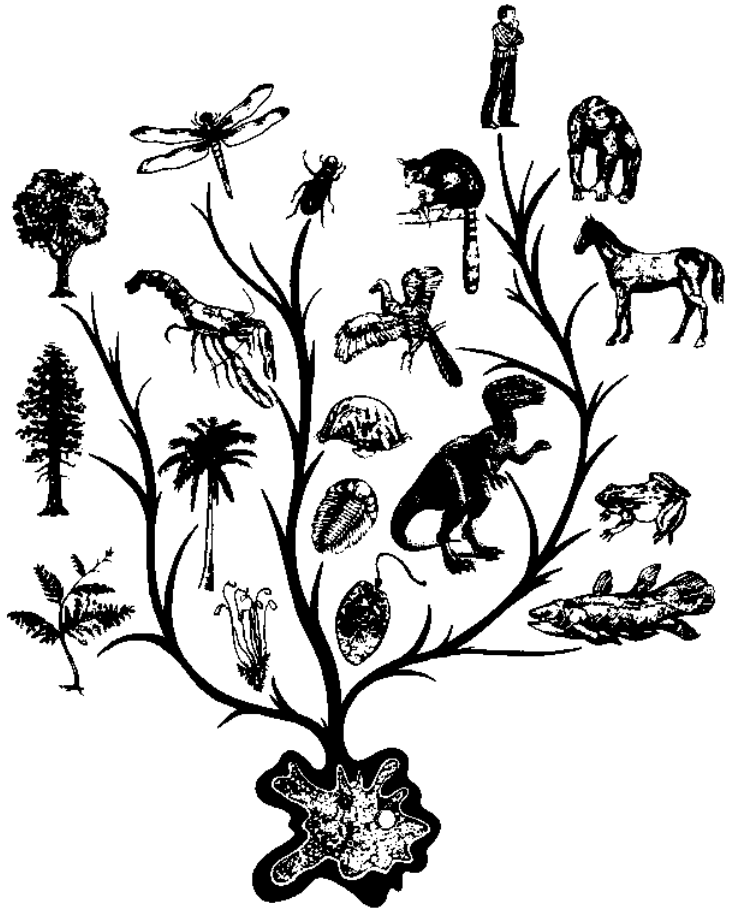
Primate evolution



A **phylogeny** is a tree that describes the sequence of speciation events that lead to the forming of a set of current day species; also called a **phylogenetic tree.**

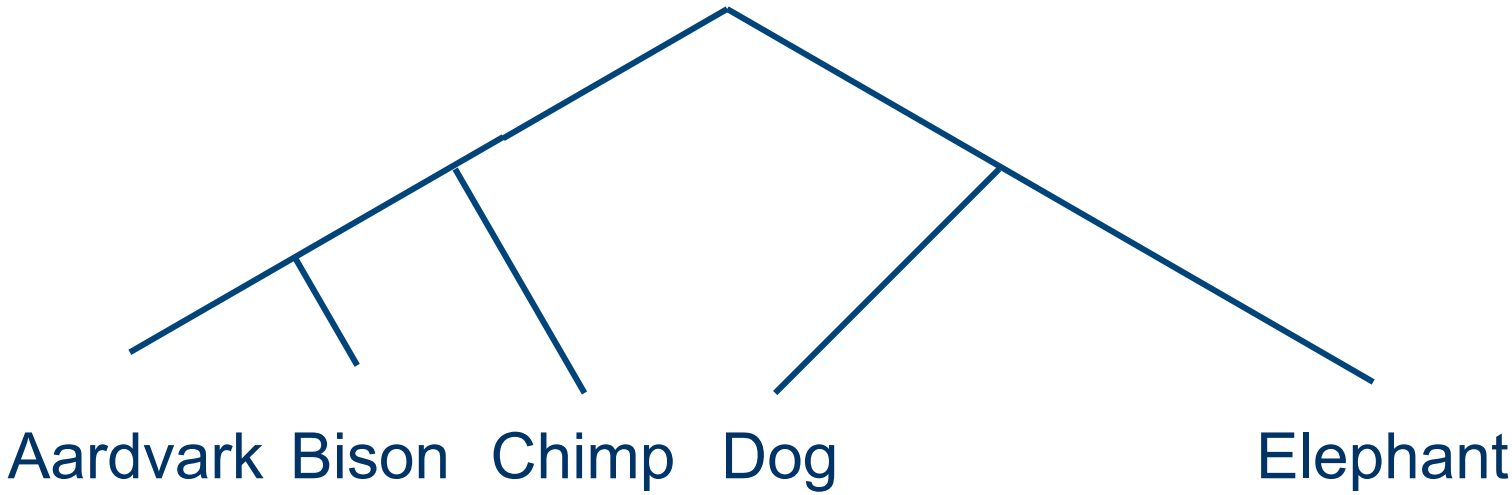
From sequences to a phylogenetic tree

```
Rat       QEPGGLVVPPTDA
Rabbit    QEPGGMVVPPTDA
Gorilla   QEPGGLVVPPTDA
Cat       REPGGLVVPPTEG
```



There are many possible types of sequences to use (e.g. Mitochondrial vs Nuclear proteins).

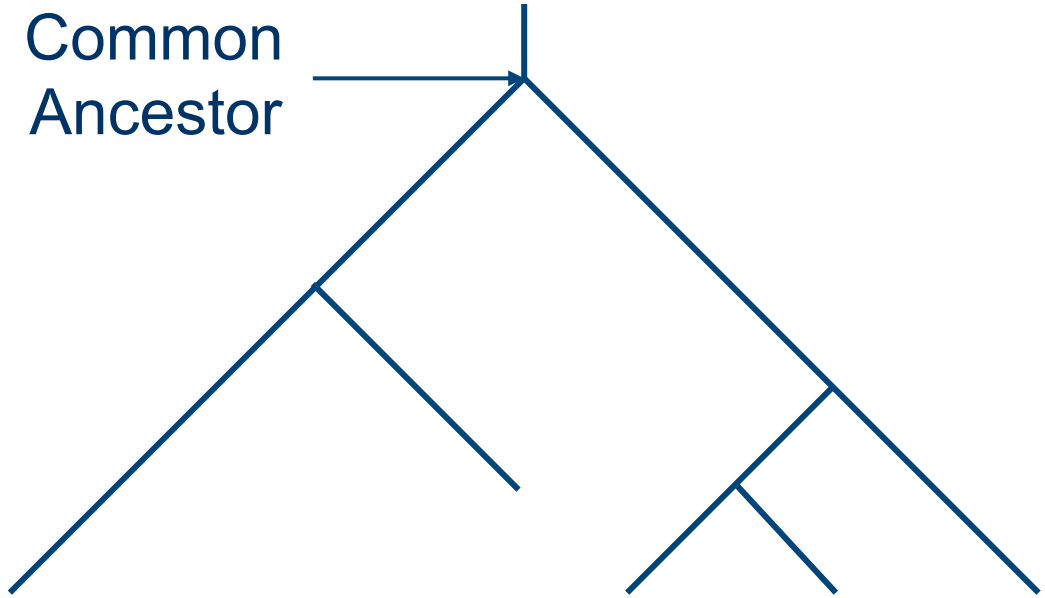
Phylogenenetic trees



- **Leaves** - current day species (or taxa – plural of taxon)
- **Internal vertices** - hypothetical common ancestors
- **Edges length** - “time” from one speciation to the next

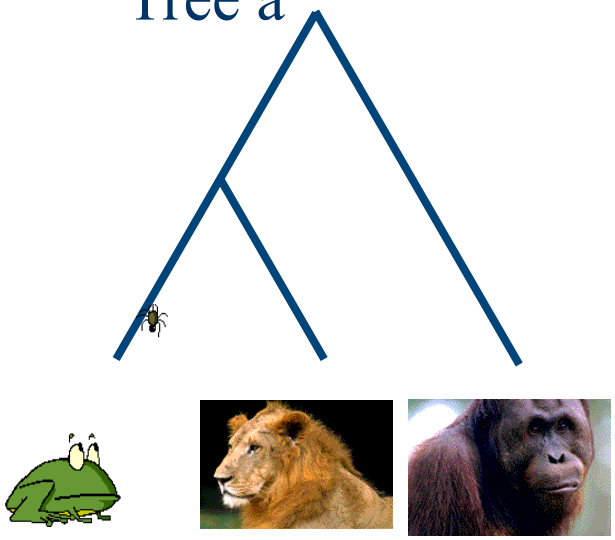
Types of Trees

A natural model to consider is that of **rooted** trees

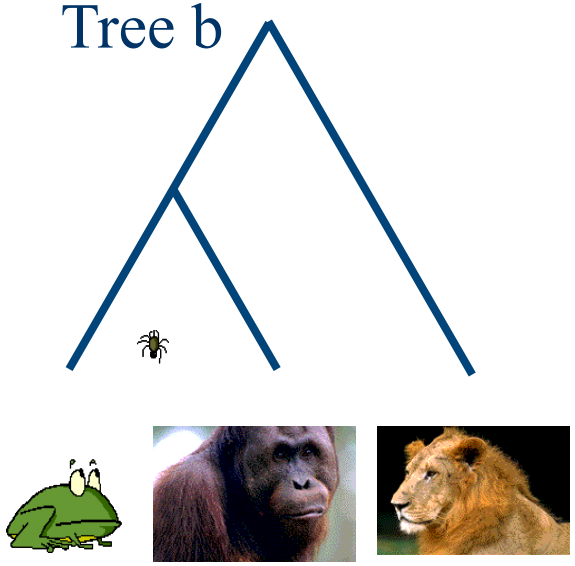


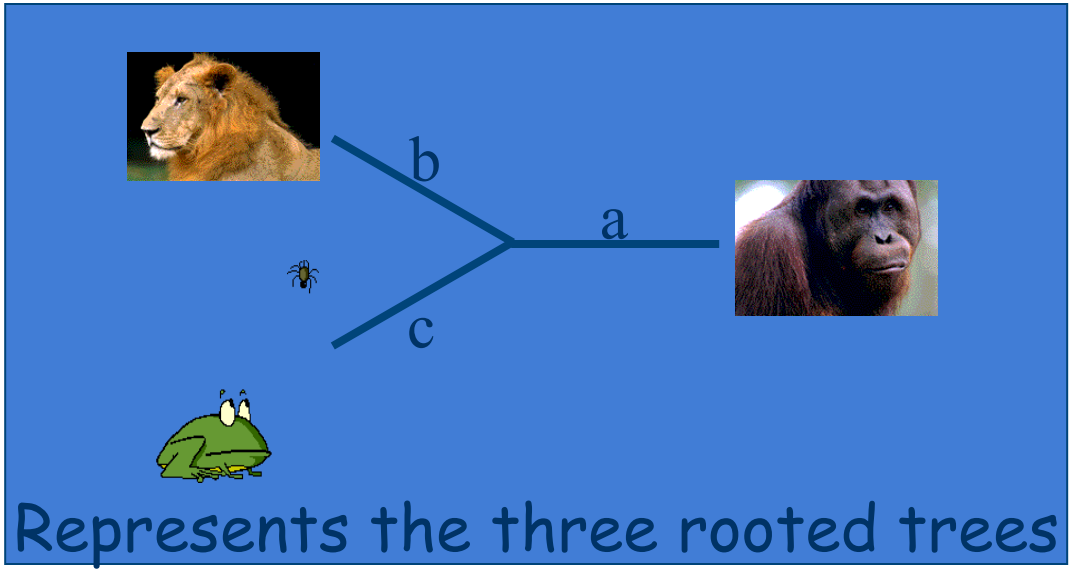
Rooted versus unrooted trees

Tree a



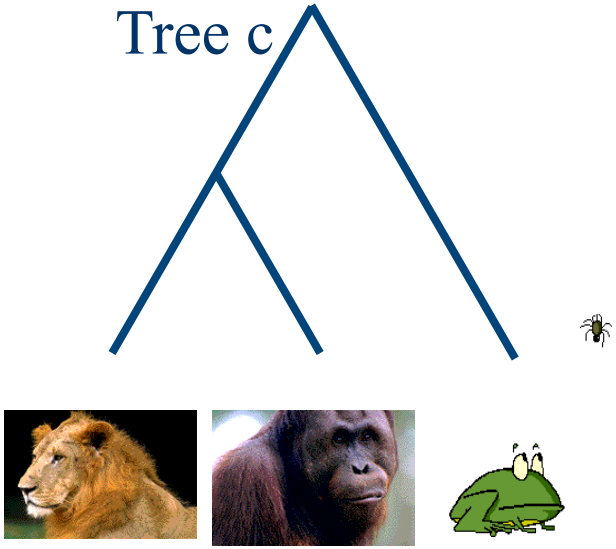
Tree b





Represents the three rooted trees

Tree c



Total numbers of trees

- For N taxa,
  - Rooted bifurcating trees:
    - $(2n-3)!! = (2n-3)!/2^{n-2}(n-2)!$
  - Unrooted bifurcating trees
    - $(2n-5)!!$
  - Tree shapes (with n leaves rooted bifurcating tree): $C(2n-1,n)/(2n-1)$

Type of Data

- **Distance**-based
  - Input is a matrix of distances between species
  - Can be fraction of residue they disagree on, or alignment score between them, or …
- **Character**-based
  - Examine each character (e.g., residue) separately

Character-based methods for constructing phylogenies

In this approach, trees are constructed by comparing the characters of the corresponding species. Characters may be morphological (teeth structures) or molecular (nucleotides in homologous DNA sequences). One common approach is Maximum Parsimony

Common Assumptions:

- Independence of characters (no correlations)
- Best tree is one where minimal changes take place

Character based methods: Input

species	C_1	C_2	C_3	C_4	…													C_m
dog	A	A	C	A	G	G	T	C	T	T	C	G	A	G	G	C	C	C
horse	A	A	C	A	G	G	C	C	T	A	T	G	A	G	A	C	C	C
frog	A	A	C	A	G	G	T	C	T	T	T	G	A	G	T	C	C	C
human	A	A	C	A	G	G	T	C	T	T	T	G	A	T	G	A	C	C
pig	A	A	C	A	G	T	T	C	T	T	C	G	A	T	G	G	C	C
	*	*	*	*	*			*	*			*	*				*	*

- Each character (column) is processed independently.
- The green character will separate the human and pig from frog, horse and dog.
- The red character will separate the dog and pig from frog, horse and human.
- We seek for a tree that will **best explain** all characters simultaneously.

Maximum Parsimony

A Character-based method

Input:

- h sequences (one per species), all of length k.

Goal:

- Find a tree with the input sequences at its leaves,

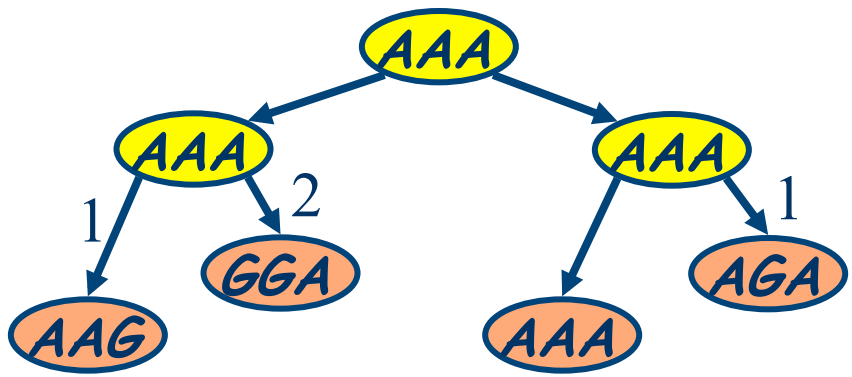
and an assignment of sequences to internal nodes,

such that the total number of substitutions is minimized.

Example

Input: four nucleotide sequences: AAG, AAA, GGA, AGA taken from four species.

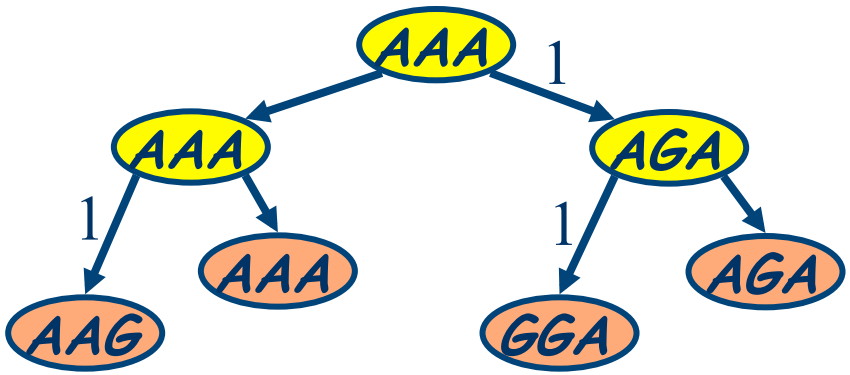
By the parsimony principle, we seek a tree that has a minimum total number of substitutions of symbols between species and their originator in the phylogenetic tree. Here is one possible tree.



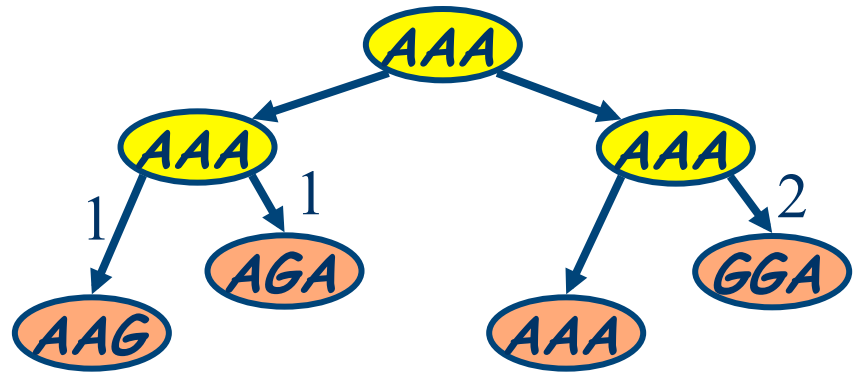
Total #substitutions = 4

Example

There are many assignments for this tree. For example:



Total #substitutions = 3



Total #substitutions = 4

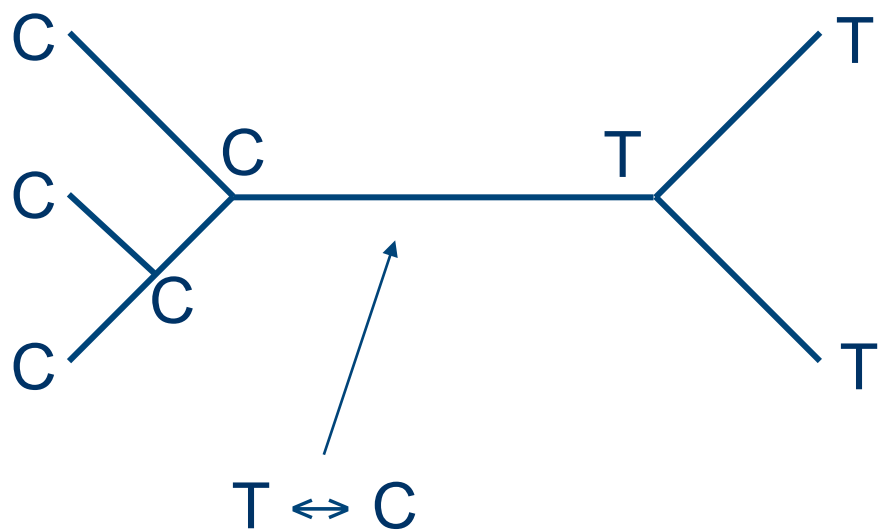
The left tree is preferred over the right tree.

The total number of changes is called the **parsimony score**.

Example with one letter sequences

- Suppose we have five species, such that three have ‘C’ and two ‘T’ at a specified position

- Minimal tree has only one evolutionary change:



Parsimony based reconstruction

- Two separate components:

1. A procedure to find the minimum number of changes needed to explain the data for a given tree topology, where species are assigned to leaves.

2. A search through the space of trees.

Finding the right tree: Perfect phylogeny problem

The algorithms of Fitch and Sankoff assume that the tree is known. Finding the optimal tree is harder.

Recall the general problem:

Input: A set of species, specified by strings of characters.

Output: A tree T, and assignment of species to the leaves of T, with minimum parsimony score.

The Perfect Phylogeny Problem

Basic assumption for the perfect phylogeny problem:

A ***character*** is a significant property, which distinguishes between species (e.g. dental structure).

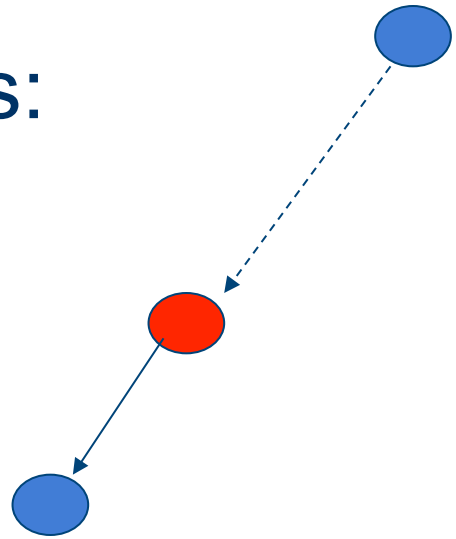
Hence, characters in evolutionary trees should be "Homoplasy free", as we define next.

Homoplasy-free characters 1

Characters in Phylogenetic Trees should avoid:

reversal transitions

- A species regains a state it's direct ancestor has lost.
- Famous known reversals:
  - Teeth in birds.
  - Legs in snakes.

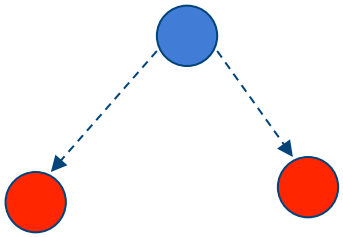


Homoplasy-free characters 2

…and also avoid

convergence transitions

- Two species possess the same state while their least common ancestor possesses a different state.
- Famous known convergence: The marsupials.



Characters as colorings

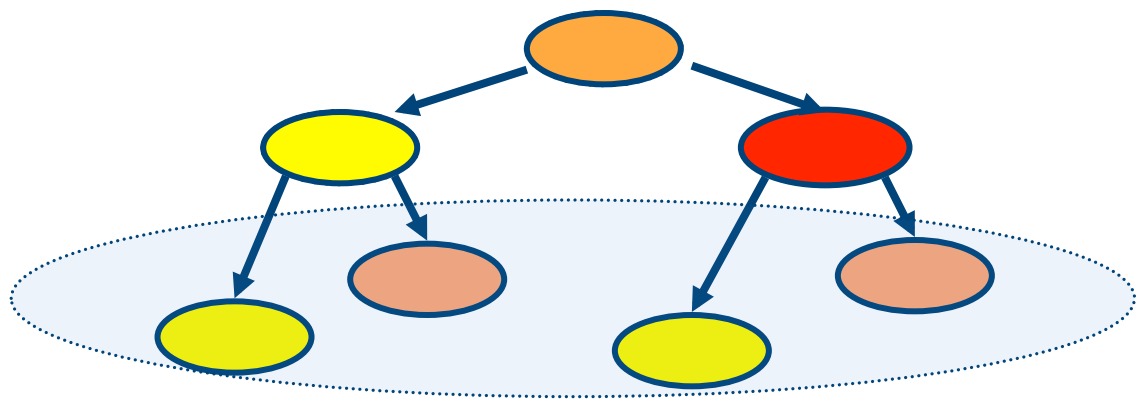
A ***coloring*** of a tree $T=(V,E)$ is a mapping

$$C:V \rightarrow \text{[set of colors]}$$

A p***artial coloring*** of T is a mapping defined on a subset of the vertices $U \subseteq V$:

$$C:U \rightarrow \text{[set of colors]}$$

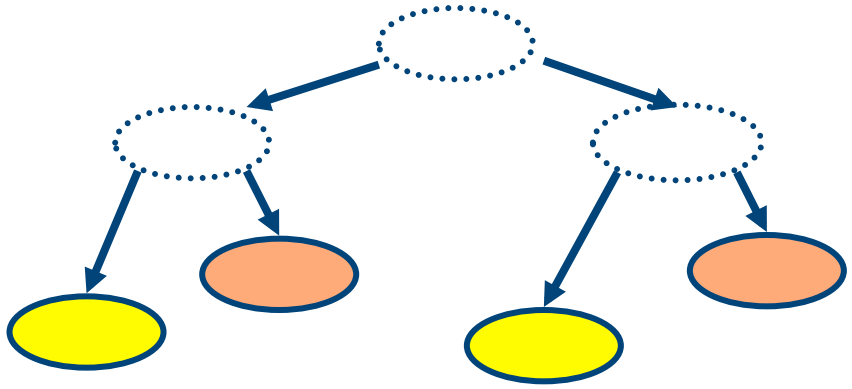
$U=$



Characters as Colorings

Each character defines a (partial) coloring of the corresponding phylogenetic tree:

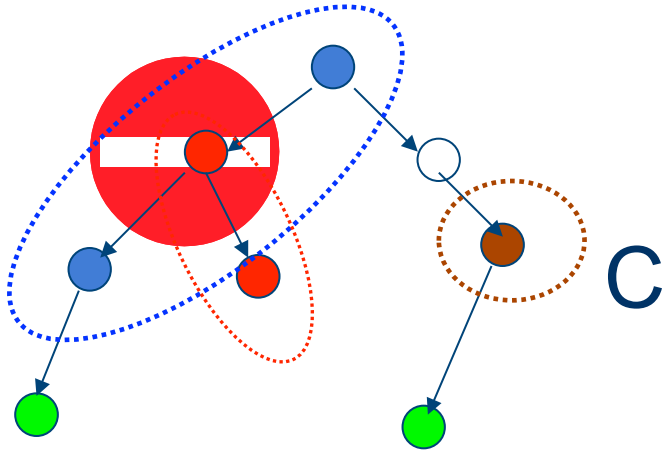
Species ≡ Vertices
States ≡ Colors



Convex colorings (characters)

Let $T=(V,E)$ be a colored tree, and d be a color.
The d-***carrier*** is the minimal subtree of T containing all vertices colored d

<u>Definition:</u> A (partial/total) coloring of a tree is ***convex*** iff all *d-carriers* are disjoint

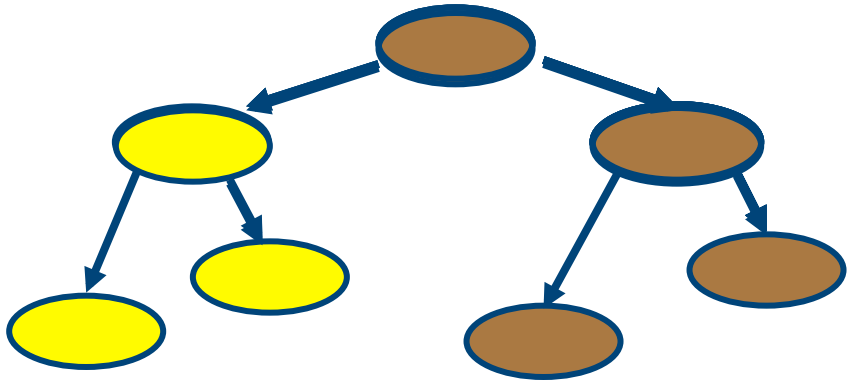


Convexity $\Leftrightarrow$ Homoplasy Freedom

A character is Homoplasy free (avoids reversal and convergence transitions)

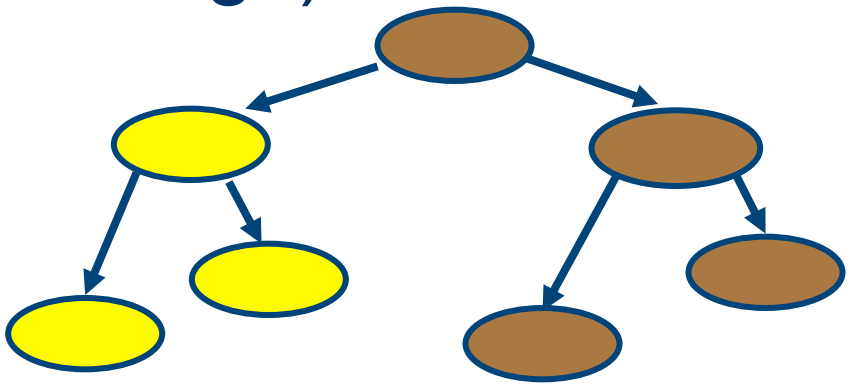
$\updownarrow$

The corresponding (partial) coloring is ***convex***



Perfect phylogeny problem

- Input: a set of species, and ***many*** characters.
- Question: is there a tree *T* containing the species as vertices, in which ***all*** the characters (colorings) are convex?



The Perfect phylogeny problem (pure graph theoretic setting)

<u>Input</u>: Partial colorings $(C_1,\ldots,C_k)$ of a set of vertices U (in the example: 3 total colorings: left, center, right, each by two colors).

<u>Problem</u>: Is there a tree $T=(V,E)$, s.t. $U \subseteq V$ and for $i=1,\ldots,k$,, C_i is a convex (partial) coloring of T?

NP-Hard In general, in P for some special cases. Next we show a polynomial time algorithm for the case of *binary characters*.

Directed binary perfect phylogeny

Directed binary characters:

- **0** – property exists
- **1** – property doesn't exist
- Initially (at the root) all propertied do not exist.

Input: binary coloring ($C_1,\ldots,C_m$) of a set S (nxm binary matrix M)

Problem: Find a phylogenetic tree T over S (if one exists), s.t.

1. For $j=1,\ldots,m$, the partial coloring induced by C_j is convex in T.
2. The root has state **0** in all characters.

➢ We will present a polynomial-time solution

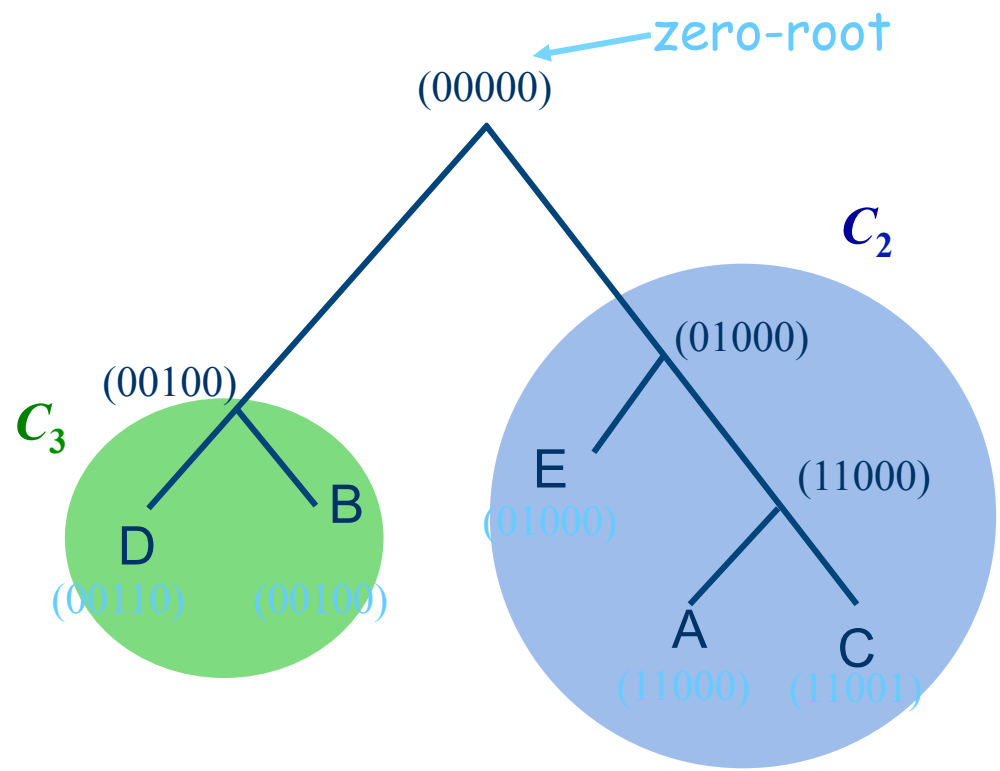
Example

Input:

m characters

n species	C_1	C_2	C_3	C_4	C_5
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	0	1
D	0	0	1	1	0
E	0	1	0	0	0

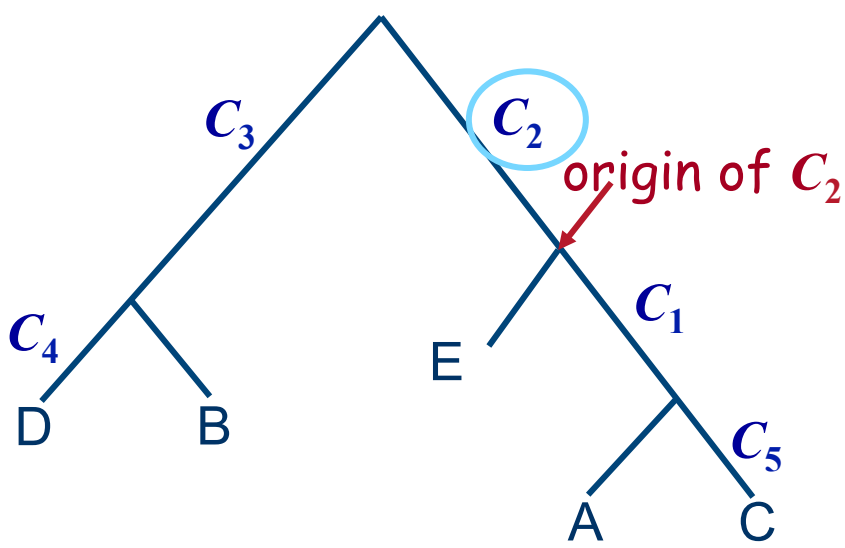
Possible output:



An important observation

A tree is a directed perfect phylogeny for a given **0/1** matrix
iff
we can map each character to an edge/vertex on which this character was "turned on".

Example:



	C_1	C_2	C_3	C_4	C_5
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	0	1
D	0	0	1	1	0
E	0	1	0	0	0

Laminar matrices

Definitions:

- O_j – set of objects that have character C_j ($O_j=\{i : M_{ij}=1\}$).
- A collection of sets $\{S_1, \ldots, S_k\}$ is ***laminar*** if
  - for all i, j, either S_i and S_j are disjoint, or one includes the other.

Theorem: A binary matrix M has a perfect phylogenetic tree iff the collection $\{O_1, \ldots, O_m\}$ is ***laminar***.

Laminar

	C_1	C_2	C_3	C_4	C_5
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	0	1
D	0	0	1	1	0
E	0	1	0	0	0

Not Laminar

	C_1	C_2	C_3	C_4	C_5
A	1	1	0	0	0
B	0	0	1	0	1
C	1	1	0	0	1
D	0	0	1	1	0
E	0	1	0	0	1

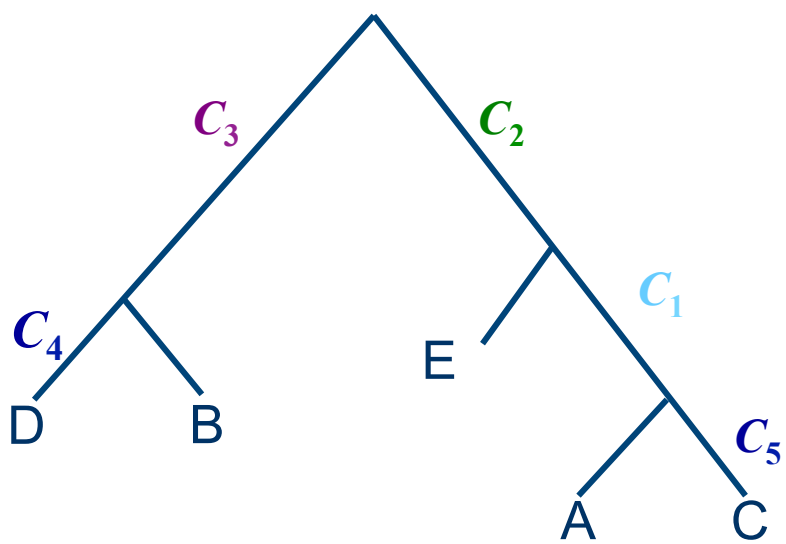
Proof

➔ Assume M has a perfect phylogeny.

Consider the edges labeled C_i and C_j:

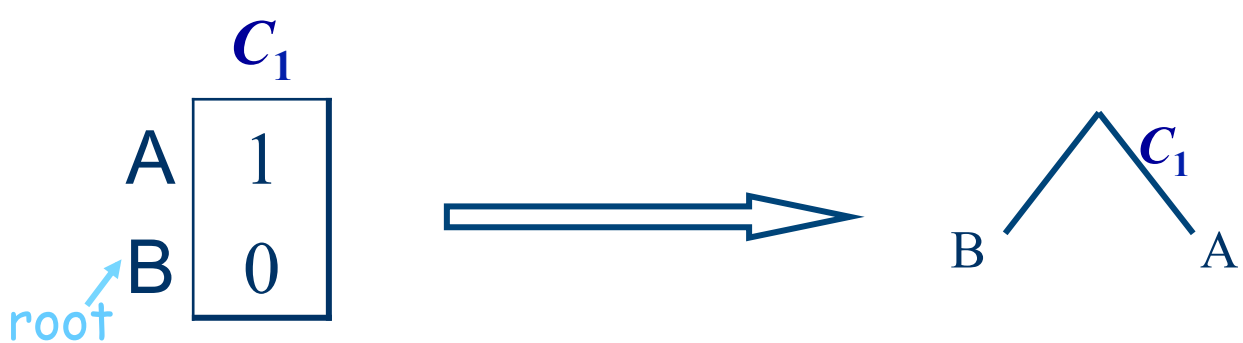
If there is a ***root-to-leaf*** path containing both edges (C_1,C_2 below), then O_i includes O_j or vice-versa.

Otherwise, O_i and O_j are disjoint (C_1,C_3 below).



🡸 Assume that the collection $\{O_1, \ldots, O_k\}$ is laminar.
We prove by induction on the number of characters k that M has a perfect phylogenetic tree.

Basis: one character. There are at most two (distinct) objects, one with and one without this character.



← Assume that the collection $\{O_1, \ldots, O_k\}$ is laminar.

Induction step: assume correctness for n-1 characters.

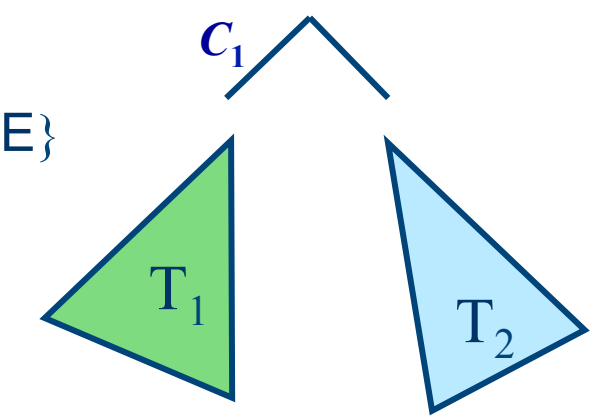
Consider a matrix with n characters (non-zero columns), and assume WLOG that O_1 is not contained in O_j for all $j > 1$.

- S_1 – the set of objects i for which $M_{i1} = 1$. S_2 – the remaining objects.
- **Claim:** each character belongs to objects in S_1 or S_2 , but not to both.

By induction there are trees T_1 and T_2 for S_1 and S_2.

	C_1	C_2	C_3	C_4	C_5
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	0	1
D	0	0	1	1	0
E	1	0	0	0	0

$S_1 = \{A,C,E\}$
$S_2 = \{B,D\}$



Efficient Implementation

1. Sort the columns (characters) according to decreasing binary value.

Claim: If the binary value of column i is larger than that of column j, then O_i is not a proper subset of O_j.

Proof: $O_i > O_j$ means the 1's in O_i are not covered by the 1's in O_j.

	C_1	C_2	C_3	C_4	C_5
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	0	1
D	0	0	1	1	0
E	0	1	0	0	0

	C_2	C_1	C_3	C_5	C_4
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	0	0

2. Make a backwards linked list of the 1's in each row

Claim: If the columns are sorted, then the set of columns is laminar iff for each column i, all the links leaving column i point at the same column.

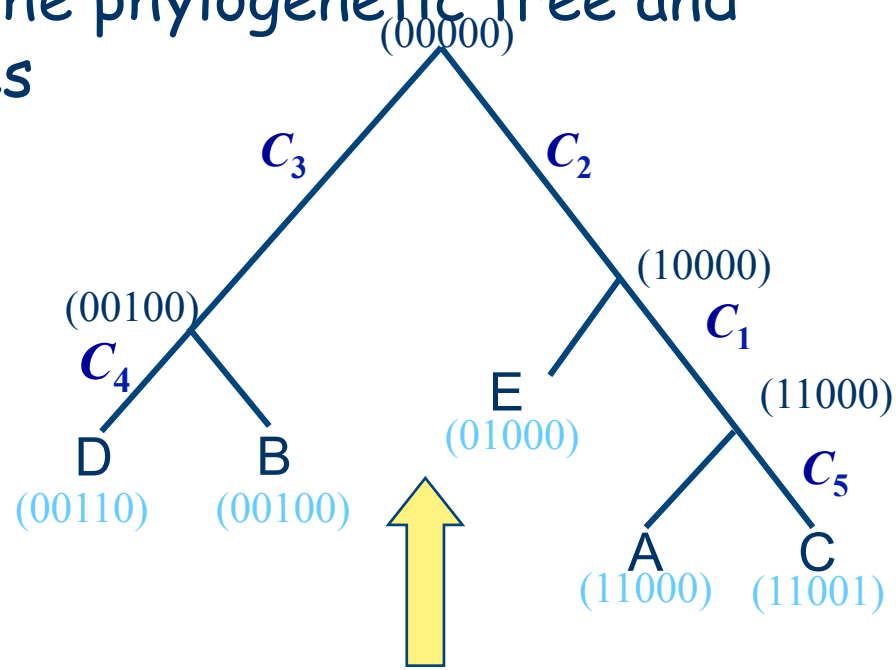
➔ If the matrix is laminar then these pointers define the inclusion hierarchy

	C_2	C_1	C_3	C_5	C_4
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	0	0

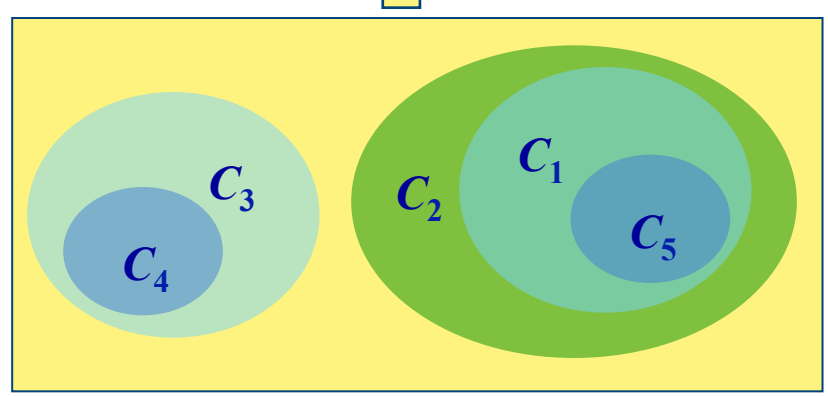
	C_2	C_1	C_3	C_5	C_4
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	0	0	1	1	0

3. If the matrix is laminar, compute the inclusion hierarchy

4. Reconstruct topology of the phylogenetic tree and ancestral character states



	C_2	C_1	C_3	C_5	C_4
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	0	0



1. Sort the columns (characters) according to decreasing binary value.
2. Make a backwards linked list of the **1**'s in each row
3. If the matrix is laminar, compute the inclusion hierarchy
4. Reconstruct topology of the phylogenetic tree and ancestral character states

Complexity: ***O(mn)*** – use radix (bucket) sort in stage 1.

	c_1	c_2	c_3	c_4	c_5
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	0	1
D	0	0	1	1	0
E	0	1	0	0	0

	c_2	c_1	c_3	c_5	c_4
A	1	1	0	0	0
B	0	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	0	0

Genotypes and haplotypes

Each individual has two "copies" of each chromosome.

At each site, each chromosome has one of two alleles (states) denoted by 0 and 1 (motivated by SNPs)

0 1 1 1 0 0 1 1 0

1 1 0 1 0 0 1 0 0

Two haplotypes per individual

Merge the haplotypes

2 1 2 1 0 0 1 2 0 Genotype for the individual

SNP Data

- A SNP is a Single Nucleotide Polymorphism - a site in the genome where two different nucleotides appear with sufficient frequency in the population (say each with 5% frequency or more).
- SNP maps in human have been compiled with a density of about 1 site per 1000.
- SNP data is what is mostly collected in populations - it is much cheaper to collect than full sequence data, and focuses on variation in the population, which is what is of interest.

Haplotype Map Project: HAPMAP

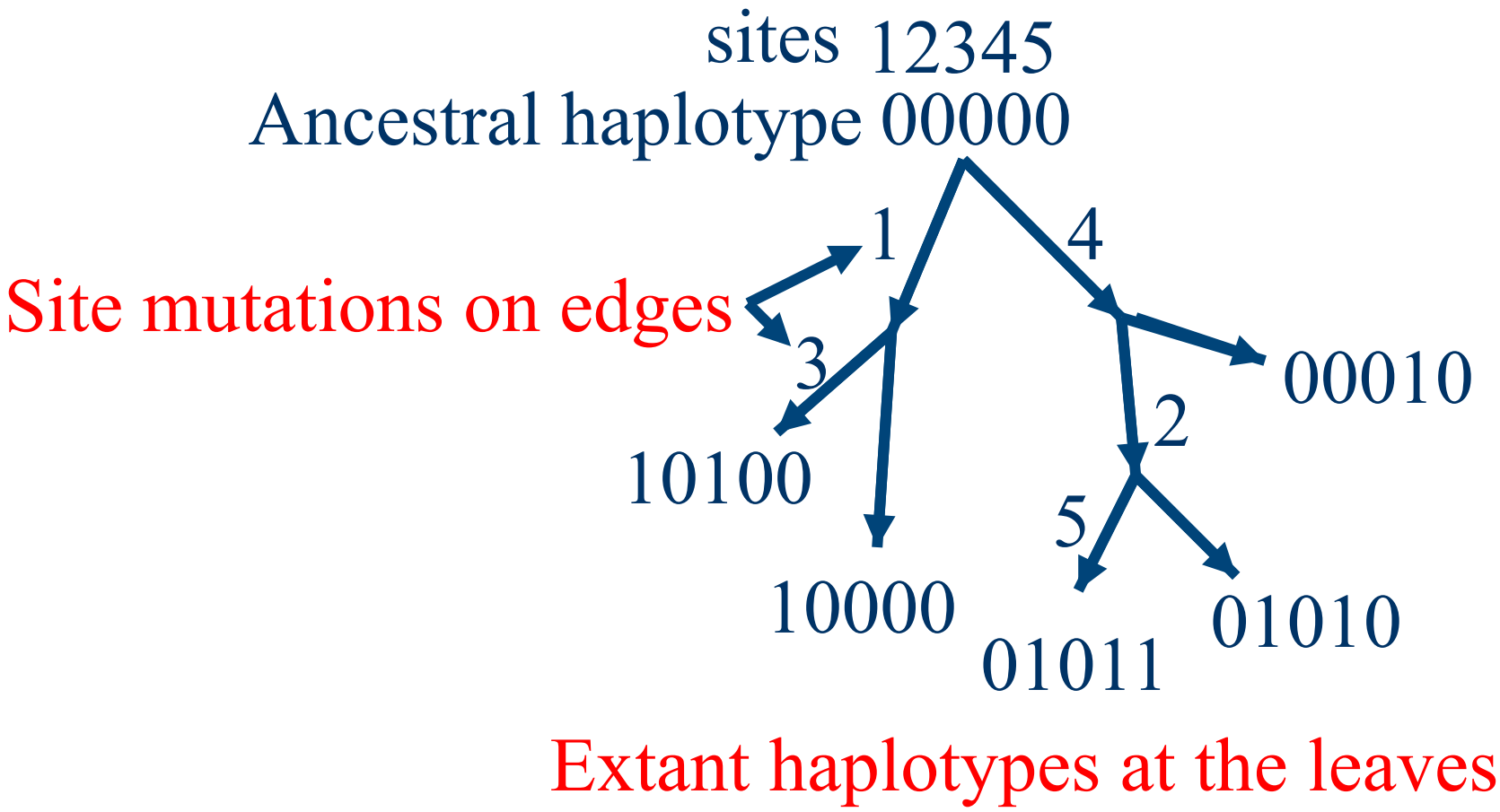
- NIH lead project ($100M) to find common haplotypes in the Human population.
- Used to try to associate genetic-influenced diseases with specific haplotypes, to either find causal haplotypes, or to find the region near causal mutations.
- Haplotyping individuals is expensive.

Haplotyping problem

- Biological Problem: for disease association studies, haplotype data is more valuable than genotype data, but haplotype data is hard to collect. Genotype data is easy to collect.

- Computational problem: given a set of n genotypes, determine the original set of n haplotype pairs that generated the n genotypes. This is hopeless without a genetic model.

Perfect phylogeny model of haplotype evolution



The Perfect phylogeny model

We assume that the evolution of extant haplotypes can be displayed on a rooted, directed tree, with the all-0 haplotype at the root, where each site changes from 0 to 1 on exactly one edge, and each extant haplotype is created by accumulating the changes on a path from the root to a leaf, where that haplotype is displayed.

In other words, the extant haplotypes evolved along a perfect phylogeny with all-0 root.

Justification for perfect phylogeny model

- In the absence of recombination each haplotype of any individual has a single parent, so tracing back the history of the haplotypes in a population gives a tree.
- Recent strong evidence for long regions of DNA with no recombination. Key to the NIH haplotype mapping project.
- Mutations are rare at selected sites, so are assumed non-recurrent.
- Connection with coalescent models.

Perfect Phylogeny Haplotype (PPH)

Given a set of genotypes S, find an explaining set of haplotypes that fits a perfect phylogeny.

sites

S

	1	2
a	2	2
b	0	2
c	1	0

Genotype matrix

A haplotype pair explains a genotype if the merge of the haplotypes creates the genotype. Example: The merge of 0 1 and 1 0 explains 2 2.

The PPH Problem

Given a set of genotypes, find an explaining set of haplotypes that fits a perfect phylogeny

	1	2
a	2	2
b	0	2
c	1	0

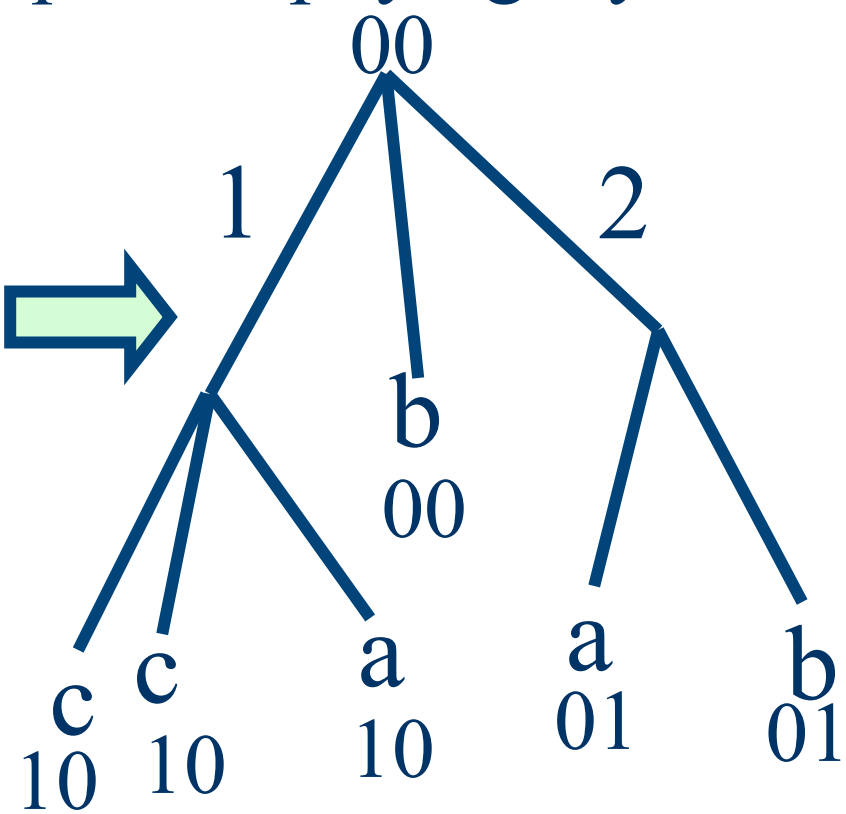
	1	2
a	1	0
a	0	1
b	0	0
b	0	1
c	1	0
c	1	0

The Haplotype Phylogeny Problem

Given a set of genotypes, find an explaining set of haplotypes that fits a perfect phylogeny

	1	2
a	2	2
b	0	2
c	1	0

	1	2
a	1	0
a	0	1
b	0	0
b	0	1
c	1	0
c	1	0



The Alternative Explanation

	1	2
a	2	2
b	0	2
c	1	0

	1	2
a	1	1
a	0	0
b	0	0
b	0	1
c	1	0
c	1	0

No tree possible for this explanation

Efficient solutions to the PPH problem - n genotypes, m sites

- Reduction to a graph realization problem (GPPH) - build on Bixby-Wagner or Fushishige solution to graph realization $O(nm\ \alpha(nm))$ time. (Gusfield 2002)
- Reduction to graph realization - build on Tutte's graph realization method $O(nm^2)$ time.
- Direct, from scratch combinatorial approach - $O(nm^2)$ (Bafna et al . 2003, Eskin et al. 2003)
- EHK approach - specialize the Tutte solution to the PPH problem - $O(nm^2)$ time.

The case of the 1's

1) For any row i in S, the set of 1 entries in row i specify the exact set of mutations on the path from the root to the least common ancestor of the two leaves labeled i, in every perfect phylogeny for S.
2) The order of those 1 entries on the path is also the same in every perfect phylogeny for S, and is easy to determine by "leaf counting".

S

	1	2	3	4	5	6	7
a	1	0	1	0	0	0	0
b	0	1	0	1	0	0	0
c	1	2	0	0	2	0	2
d	2	2	0	0	0	2	0

Leaf Counting

In any column c, count two for each 1, and count one for each 2. The total is the number of leaves below mutation c, in every perfect phylogeny for S. So if we know the set of mutations on a path from the root, we know their order as well.

S

	1	2	3	4	5	6	7
a	1	0	1	0	0	0	0
b	0	1	0	1	0	0	0
c	1	2	0	0	2	0	2
d	2	2	0	0	0	2	0
Count	5	4	2	2	1	1	1

So Assume

The columns are sorted by leaf-count, largest to the left.

Similarly

In any perfect phylogeny, the edge corresponding to the leftmost 2 in a row must be on a path just after the 1's for that row.

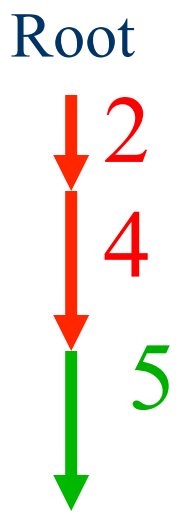
Simple conclusions

sites

1 2 3 4 5 6 7

i:0 1 0 1 2 2 2

Subtree for row i data



The order is known for the red mutations together with the leftmost blue mutation.

But what to do with the remaining blue entries (2’s) in a row?

More simple tools

3) For any row i in S, and any column c, if S(i,c) is 2, then in every perfect phylogeny for S, the path between the two leaves labeled i, must contain the edge with mutation c.

Further, every mutation c on the path between the two i leaves must be from such a column c.

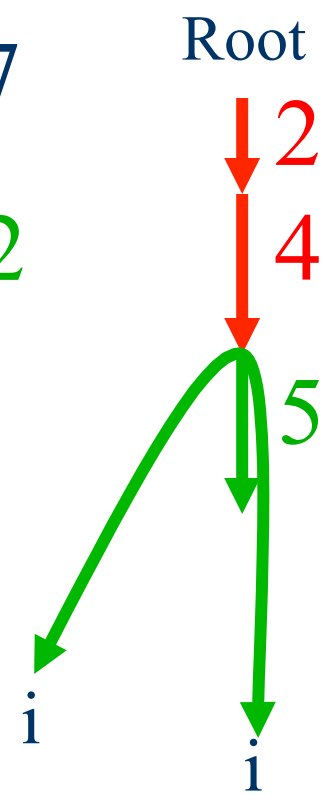
From row data to tree constraints

sites

1 2 3 4 5 6 7

i:0 1 0 1 2 2 2

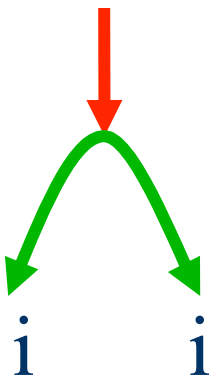
Subtree for row i data



Edges 5, 6 and 7 must be on the blue path, and 5 is already known to follow 4, but we don't where to put 6 and 7.

The graph theoretic problem

Given a genotype matrix S with n sites, and a red-blue subgraph for each row i,



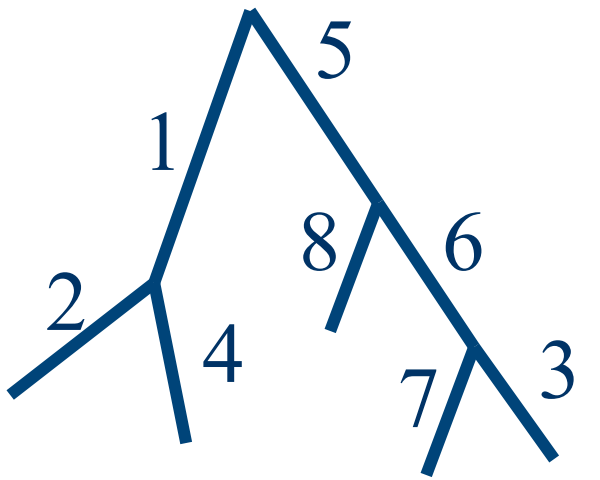
create a directed tree T where each integer from 1 to n labels exactly one edge, so that each subgraph is contained in T.

Powerful tool: graph realization

- Let Rn be the integers 1 to n, and let P be an unordered subset of Rn. P is called a path set.
- A tree T with n edges, where each is labeled with a unique integer of Rn, realizes P if there is a contiguous path in T labeled with the integers of P and no others.
- Given a family P1, P2, P3…Pk of path sets, tree T realizes the family if it realizes each Pi.
- The graph realization problem generalizes the consecutive ones problem, where T is a path.

Graph realization example

P1: 1, 5, 8
P2: 2, 4
P3: 1, 2, 5, 6
P4: 3, 6, 8
P5: 1, 5, 6, 7



Realizing Tree T

Graph realization

Polynomial time (almost linear-time) algorithms exist for the graph realization problem – Whitney, Tutte, Cunningham, Edmonds, Bixby, Wagner, Gavril, Tamari, Fushishige, Lofgren 1930's - 1980's

The algorithms are not simple; none implemented before 2002.

Recognizing graphic Matroids

- The graph realization problem is the same problem as determining if a binary matroid is graphic, and the algorithms come from that literature.
- The fastest algorithm is due to Bixby and Wagner
- Representation methods due to Cunningham et al.

Reducing PPH to graph realization

We solve any instance of the PPH problem by creating appropriate path sets, so that a solution to the resulting graph realization problem leads to a solution to the PPH problem instance.

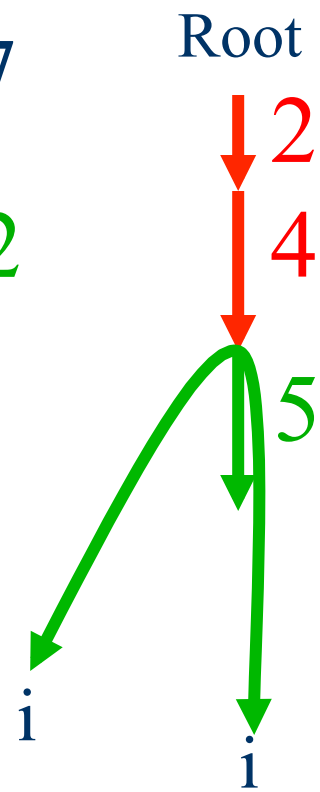
The key issue: How to encode the needed subgraph for each row, and glue them together at the root.

From raw data to tree constraints

sites
1 2 3 4 5 6 7

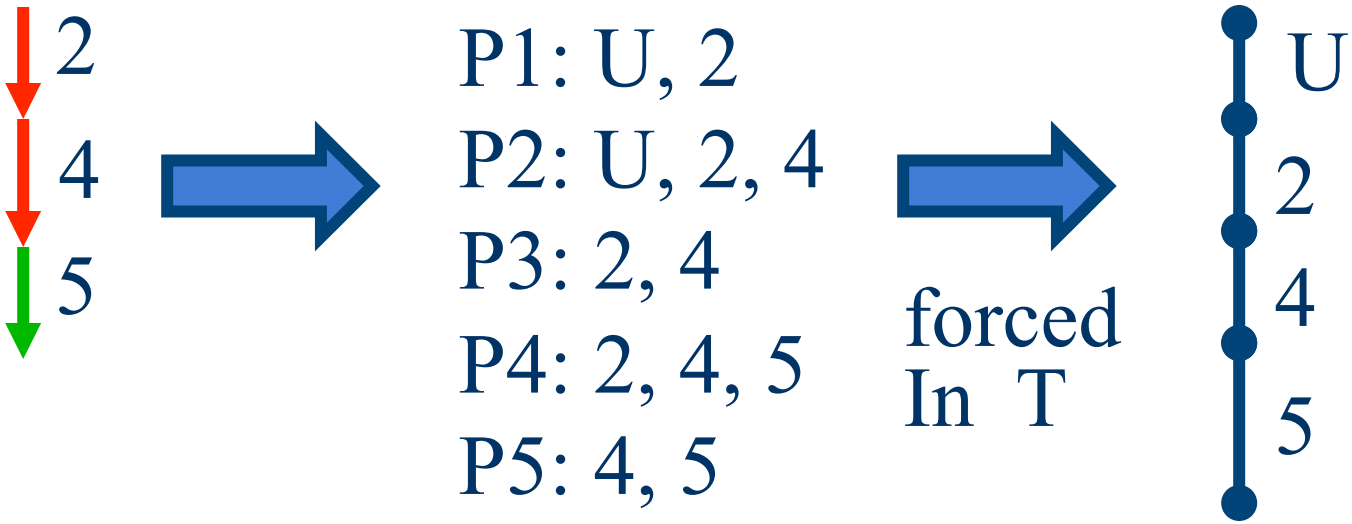
i:0 1 0 1 2 2 2

Subtree for row i data



Edges 5, 6 and 7 must be on the blue path, and 5 is already known to follow 4.

Encoding a Red-Blue directed path



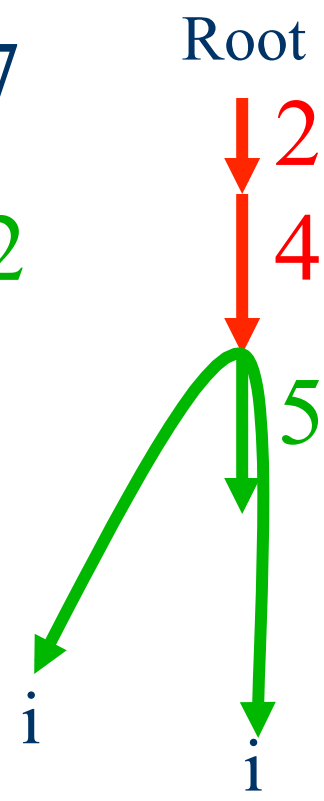
U is a glue edge used to glue together the directed paths from the different rows.

Now add a path set for the blues in row i.

sites

1 2 3 4 5 6 7

i:0 1 0 1 2 2 2



P: 5, 6, 7

That's the reduction

The resulting path-sets encode everything that is known about row i in the input.

The family of path-sets are input to the graph-realization problem, and every solution to the that graph-realization problem specifies a solution to the PPH problem, and conversely.

But how is graph realization solved?

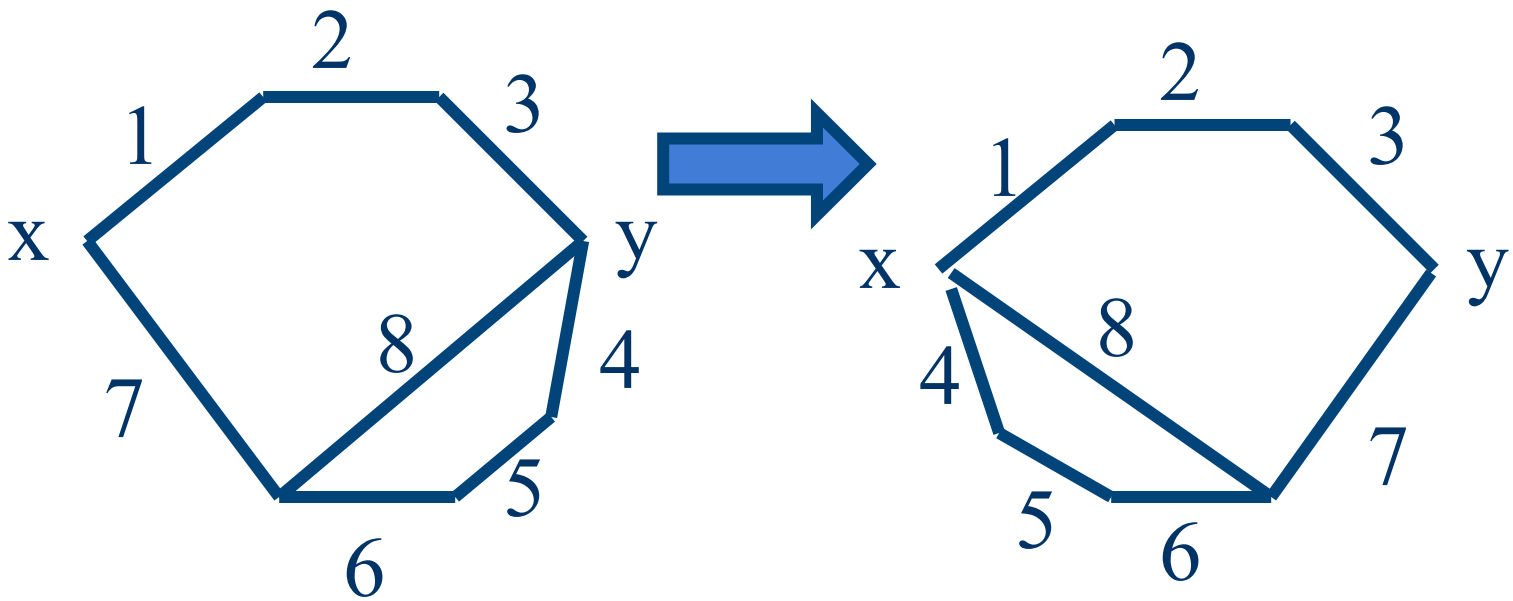
Tutte's Algorithm for Graph Realization, given a partial solution T.

- Pick an unpicked edge e.
- Determine any other edges that must be on one particular side of e or the other.
- Determine any pair of edges that must be on opposite sides of e. Form a graph G with an edge between any such pair - test if bipartite. If so, assign one side of G to one side of e, and the other side of G to the other side of e.
- Apply the decisions, modifying T, and recursion.

Multiple solutions

- In 1933, Whitney showed how multiple solutions to the graph realization problem are related.
- Partition the edges of G into two connected graphs, each with at least two edges, such that the two graphs have exactly two nodes in common. Then twist one graph around those two nodes.
- Any solution can be transformed to another via a series of such twists.

Twisting Example



All the cycle sets are preserved by twisting

Representing all the solutions

All the solutions to the PPH problem can be implicitly represented in a data structure which can be built in linear time, once one solution is known. Then each solution can be generated in linear time per solution. Method is a small modification of ones developed by Cunningham and Edwards, and by Hopcroft and Tarjan.

The DPPH Method

- Bafna et al. $O(nm^2)$ time
- Based on deeper combinatorial observations about the PPH problem.
- A matrix-centric approach (rather than tree-centric), although a graph is used in the algorithm.

First, we need to understand why some sets of haplotypes have a perfect phylogeny, and some do not.

When does a set of haplotypes fit a perfect phylogeny?

Arrange the haplotypes in a matrix, two haplotypes for each individual. Then (with no duplicate columns), the haplotypes fit a unique perfect phylogeny if and only if no two columns contain all three pairs:

0,1 and 1,0 and 1,1

This is called the 3-Gamete Test

The Alternative Explanation

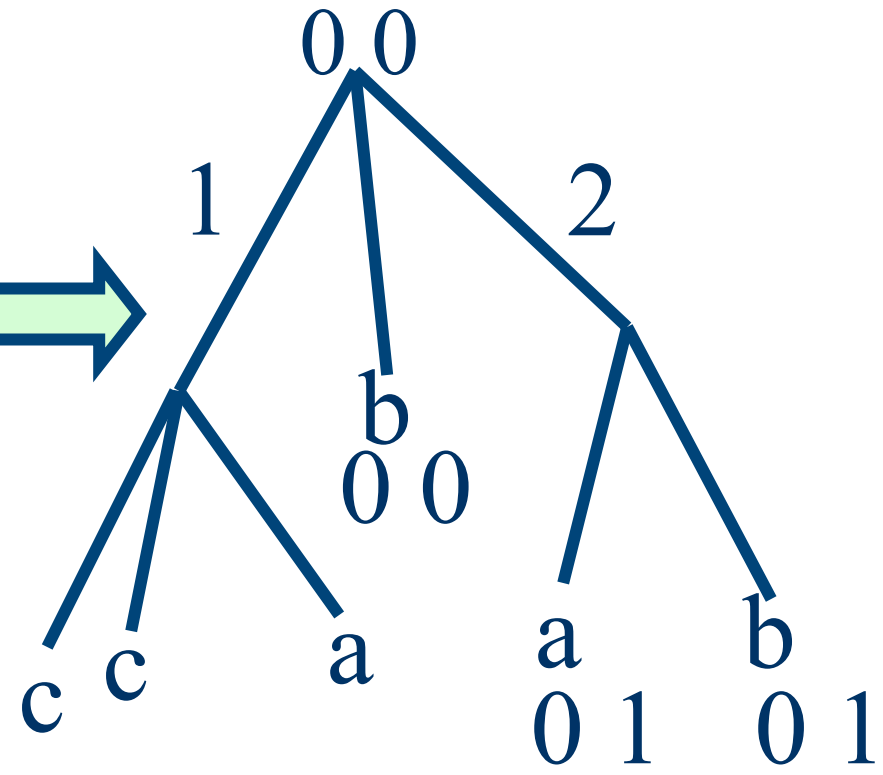
	1	2
a	2	2
b	0	2
c	1	0

	1	2
a	1	1
a	0	0
b	0	0
b	0	1
c	1	0
c	1	0

No tree possible for this explanation

	1	2
a	2	2
b	0	2
c	1	0

	1	2
a	1	0
a	0	1
b	0	0
b	0	1
c	1	0
c	1	0



PPH: The combinatorial problem

Input: A ternary matrix (0,1,2) M with 2N rows partitioned into N pairs of rows, where the two rows in each pair are identical.

Def: If a pair of rows (r,r') in the partition have entry values of 2 in a column j then positions (r,j) and (r' ,j) are called Mates.

Output: A binary matrix M' created from M by replacing each 2 in M with either 0 or 1, such that

a) A position is assigned 0 if and only if its Mate is assigned 1.

b) M' passes the 3-Gamete Test, i.e., does not contain a 3x2 submatrix (after row and column permutations) with all three combinations 0,1; 1,0; and 1,1

Initial observations

If two columns of M contain the following rows

2 0
2 0 mates

0 2
0 2 mates

then M' will contain a row with 1 0 and a row with 0 1 in those columns.

This is a forced expansion.

Initial observations

Similarly, if two columns of M contain the mates

2 1

2 1

then M' will contain a row with 1 1 in those columns.

This is a forced expansion.

If a forced expansion of two columns
creates 0 1 in those columns, then any 2 2
1 0 2 2
in those columns must be set to be
0 1
1 0
We say that two columns are forced out-of-phase.

If a forced expansion of two columns
creates 1 1 in those columns, then any 2 2
2 2
in those columns must be set to be
1 1
0 0
We say that two columns are forced in-phase.

	1	2	3
a	1	2	2
a	1	2	2
b	2	0	2
b	2	0	2
c	1	2	2
c	1	2	2
d	1	2	2
d	1	2	2
e	2	2	0
e	2	2	0

Example:

Columns 1 and 2, and 1 and 3 are forced in-phase.
Columns 2 and 3 are forced out-of-phase.

Immediate failure

It can happen that the forced expansion of cells creates a 3x2 submatrix that fails the 3-Gamete Test. In that case, there is no PPH solution for M.

Example:

20
20
11
11
02
02

Will fail the 3-Gamete Test

An $O(nm^2)$-time Algorithm

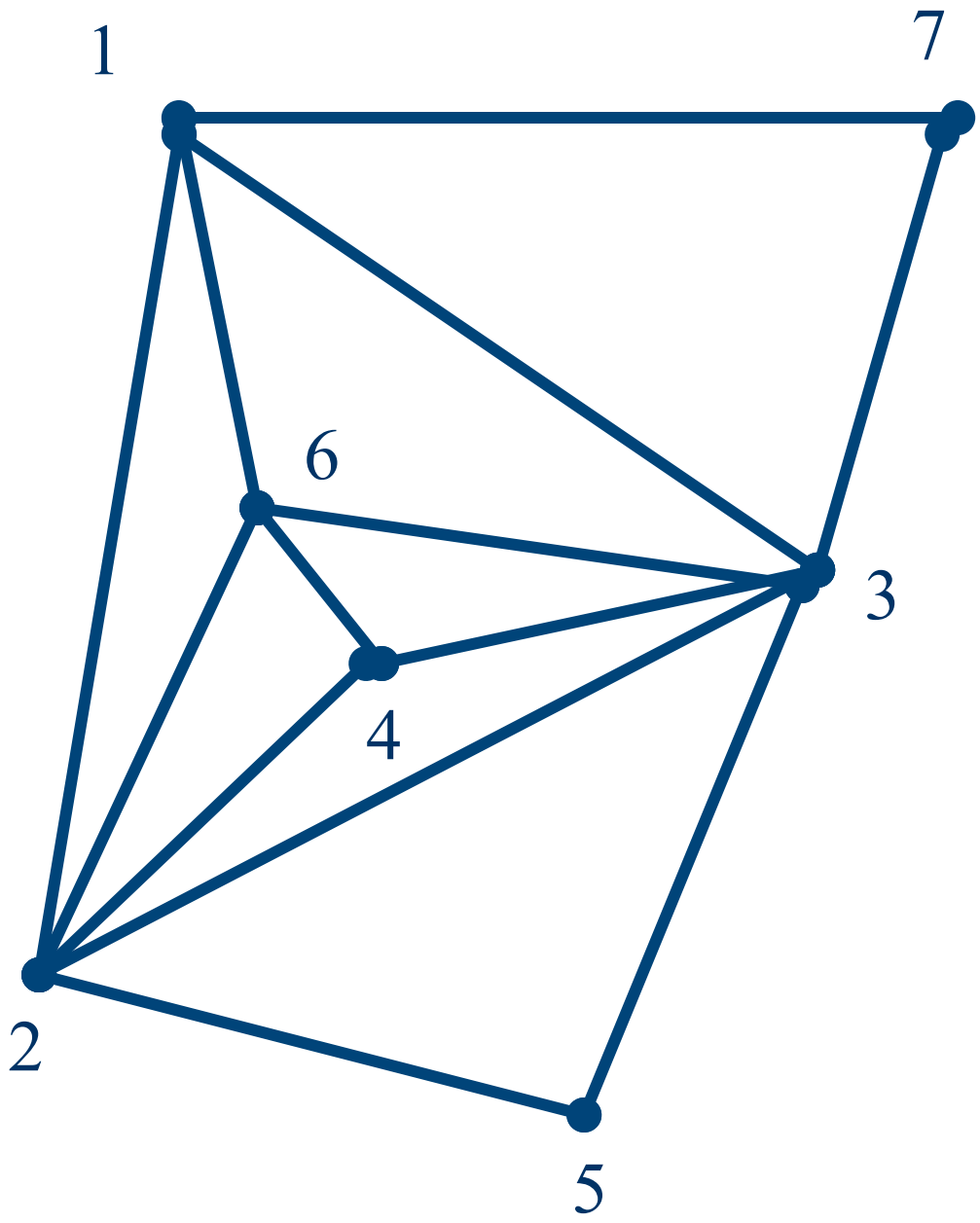
- Find all the forced phase relationships by considering columns in pairs.
- Find all the inferred, invariant, phase relationships.
- Find a set of column pairs whose phase relationship can be arbitrarily set, so that all the remaining phase relationships can be inferred.
- Result: An implicit representation of all solutions to the PPH problem.

	1	2	3	4	5	6	7
a	1	2	2	2	0	0	0
a	1	2	2	2	0	0	0
b	2	0	2	0	0	0	2
b	2	0	2	0	0	0	2
c	1	2	2	2	0	2	0
c	1	2	2	2	0	2	0
d	1	2	2	0	2	0	0
d	1	2	2	0	2	0	0
e	2	2	0	0	0	2	0
e	2	2	0	0	0	2	0

A running example.

Overview of Bafna et al. algorithm

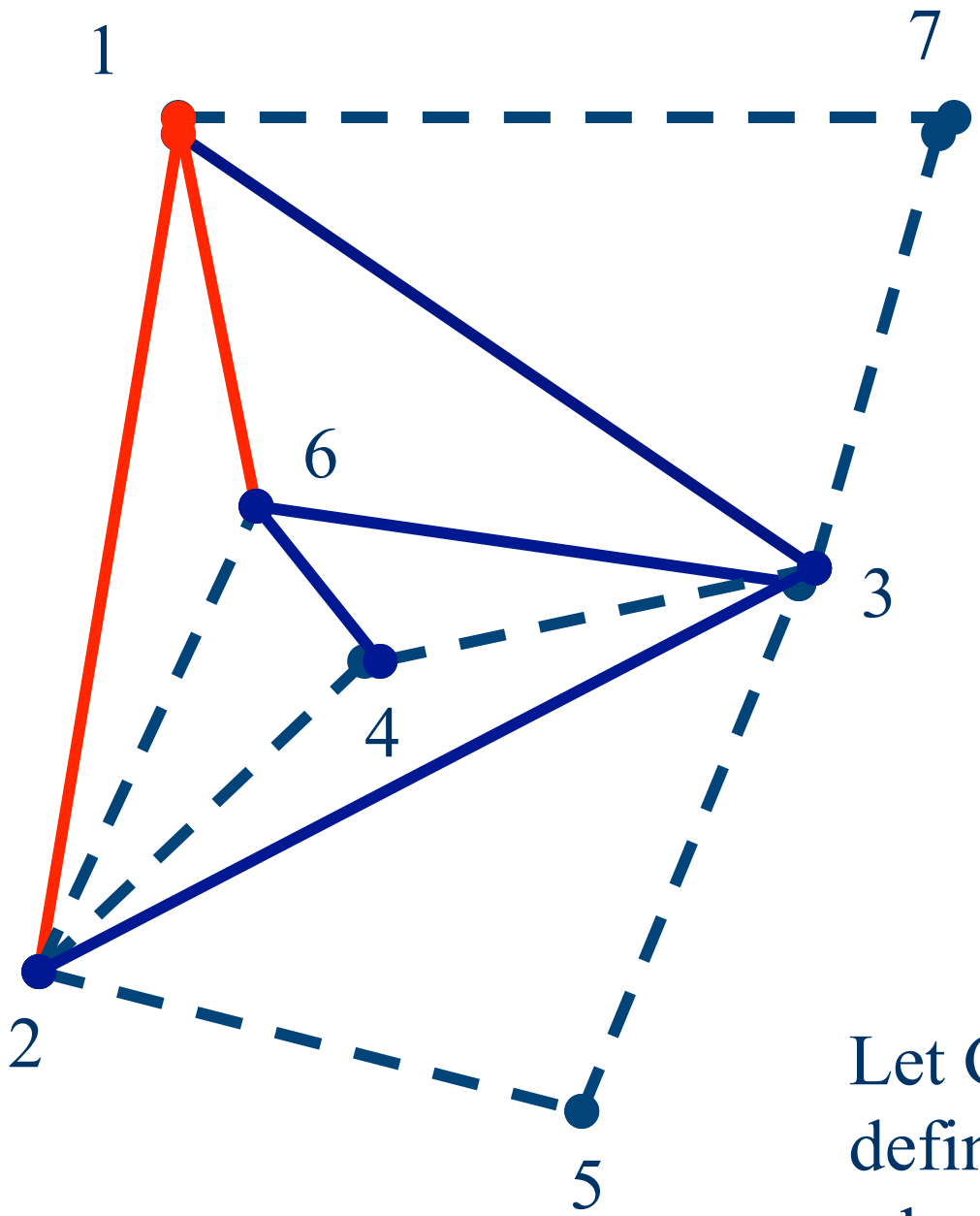
First, represent the forced phase relationships, and the needed decisions, in a graph G.



Graph G

Each node represents a column in M, and each edge indicates that the pair of columns has a row with 2's in both columns.

The algorithm builds this graph, and then checks whether any pair of nodes is forced in or out of phase.

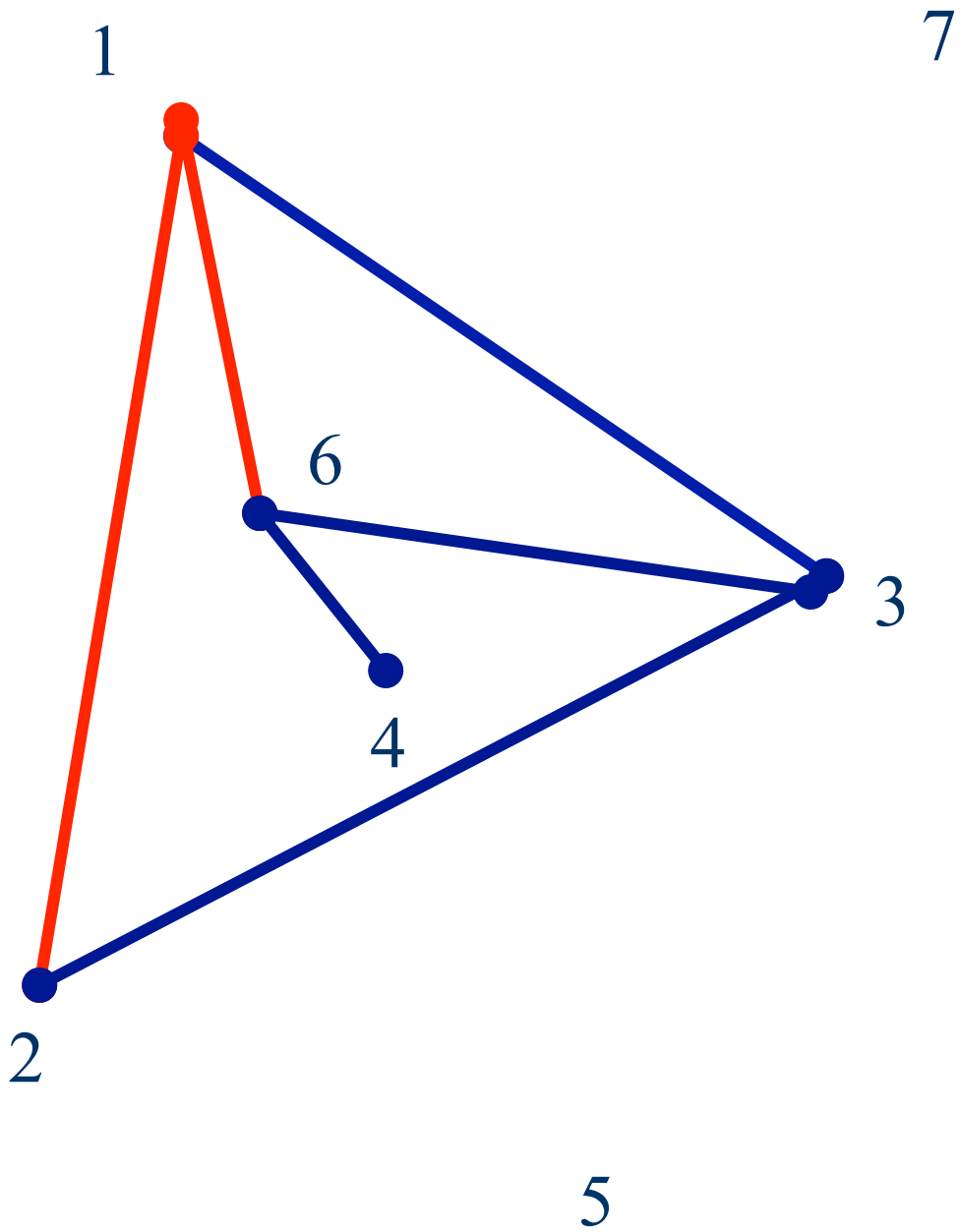


Graph Gc

Each Red edge indicates that the columns are forced in-phase.

Each Blue edge indicates that the columns are forced out-of-phase.

Let Gf be the subgraph of Gc defined by the red and blue edges.



Graph Gf has three connected components.

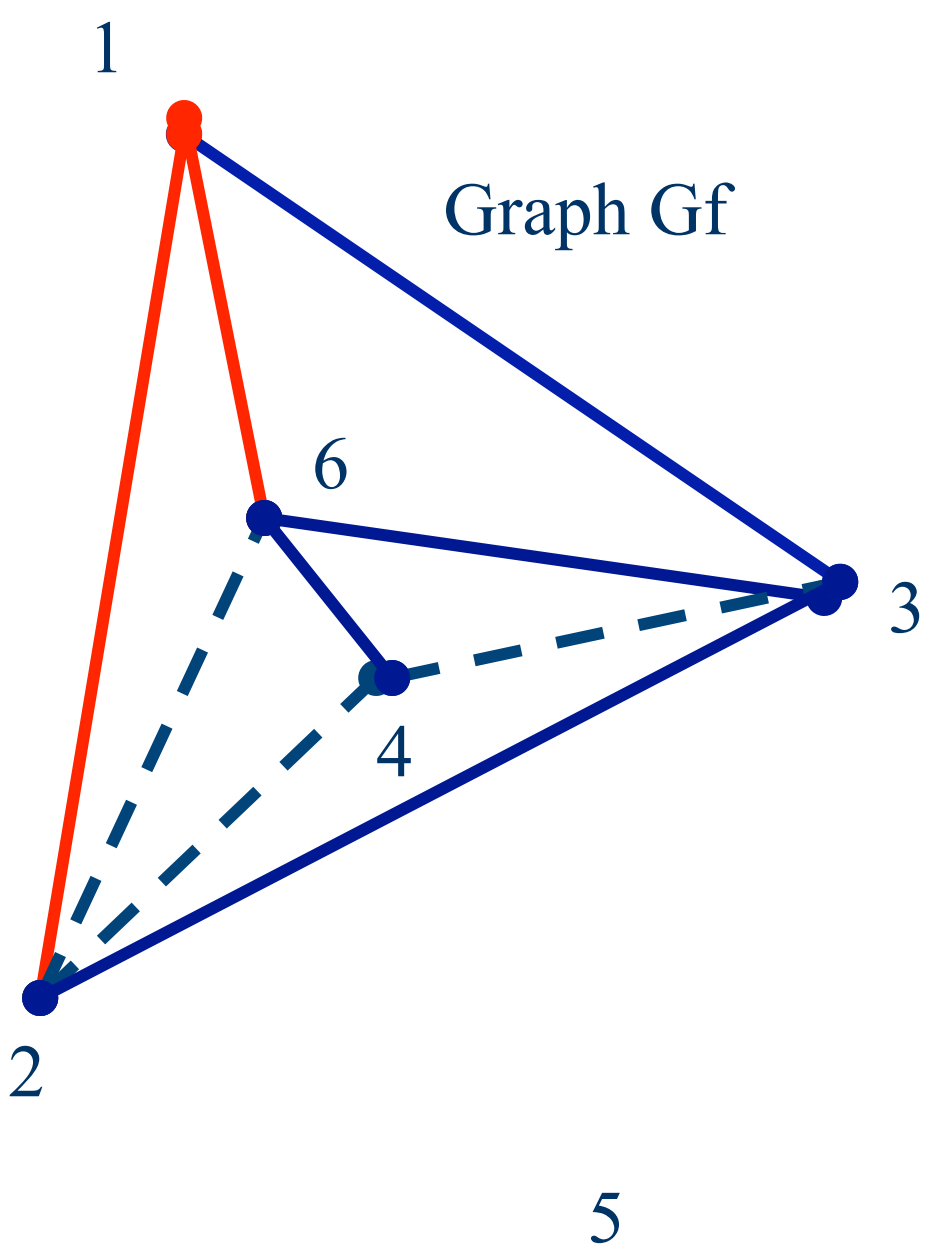
The central theorem

There is a solution to the PPH problem for M if and only if there is a coloring of the dashed edges of Gc with the following property:

For any triangle (i,j,k) in Gc, where there is one row containing 2's in all three columns i,j and k (any triangle containing at least one dashed edge will be of this type), the coloring makes either 0 or 2 of the edges blue (out-of-phase).

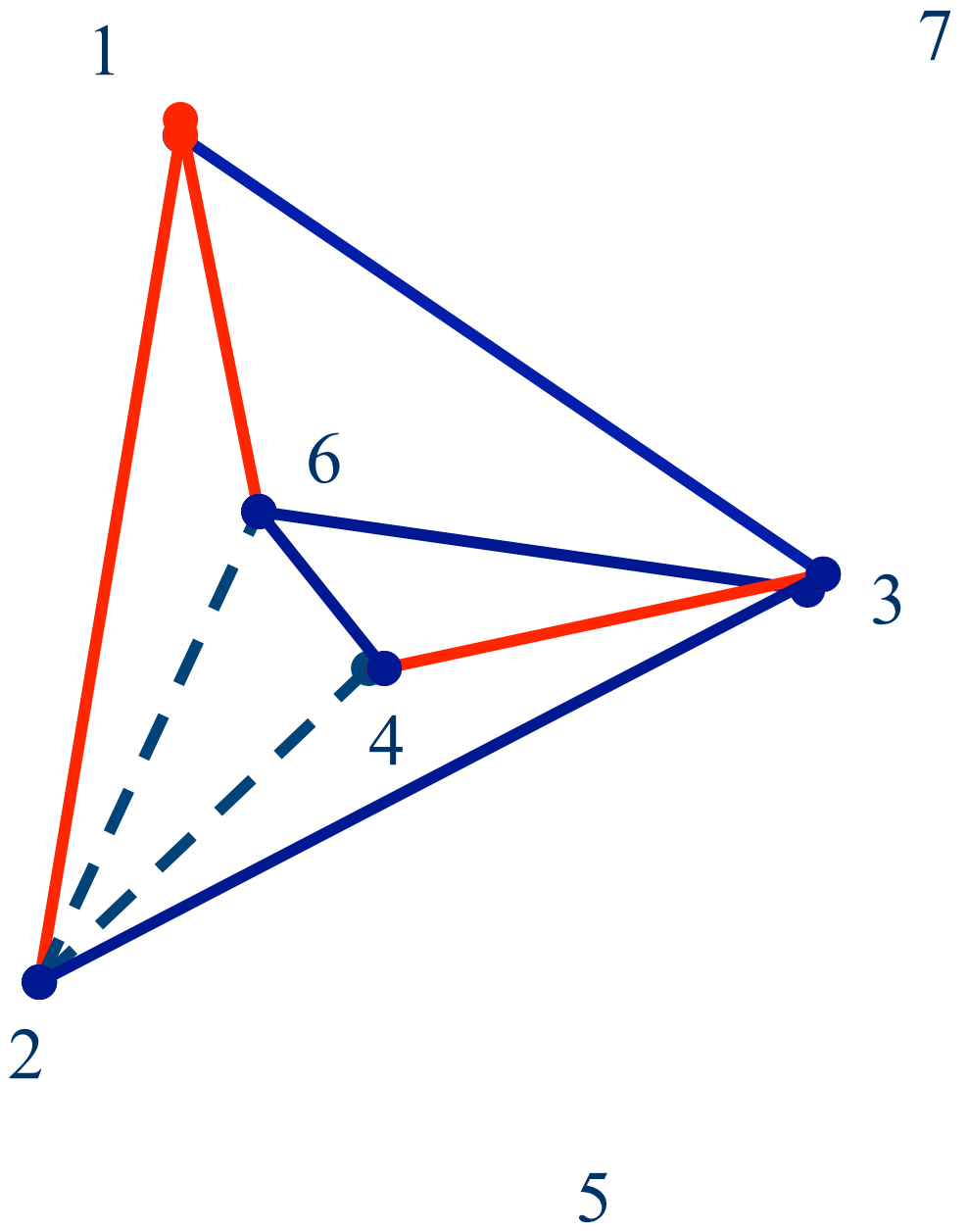
Nice, but how do we find such a coloring?

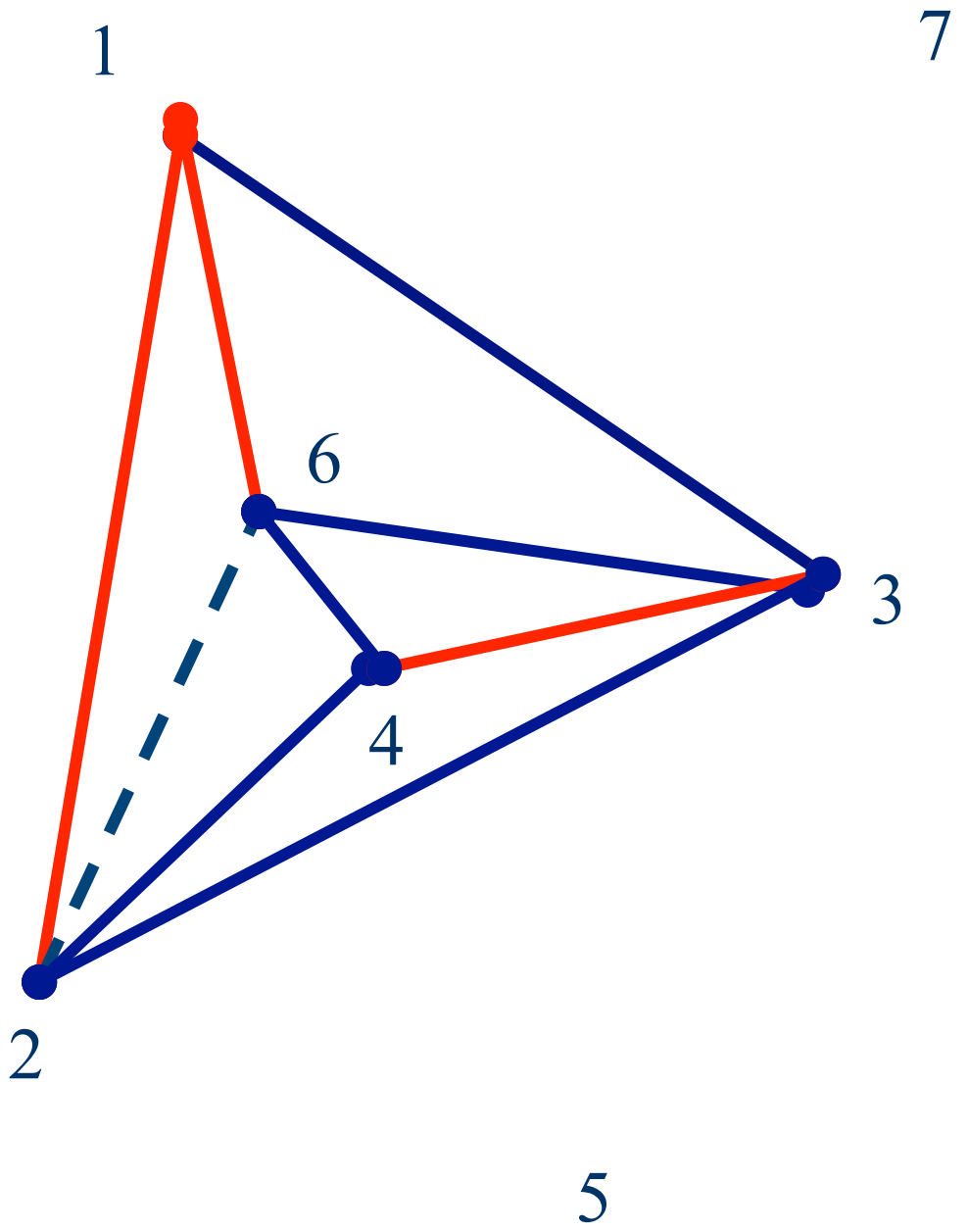
Triangle Rule

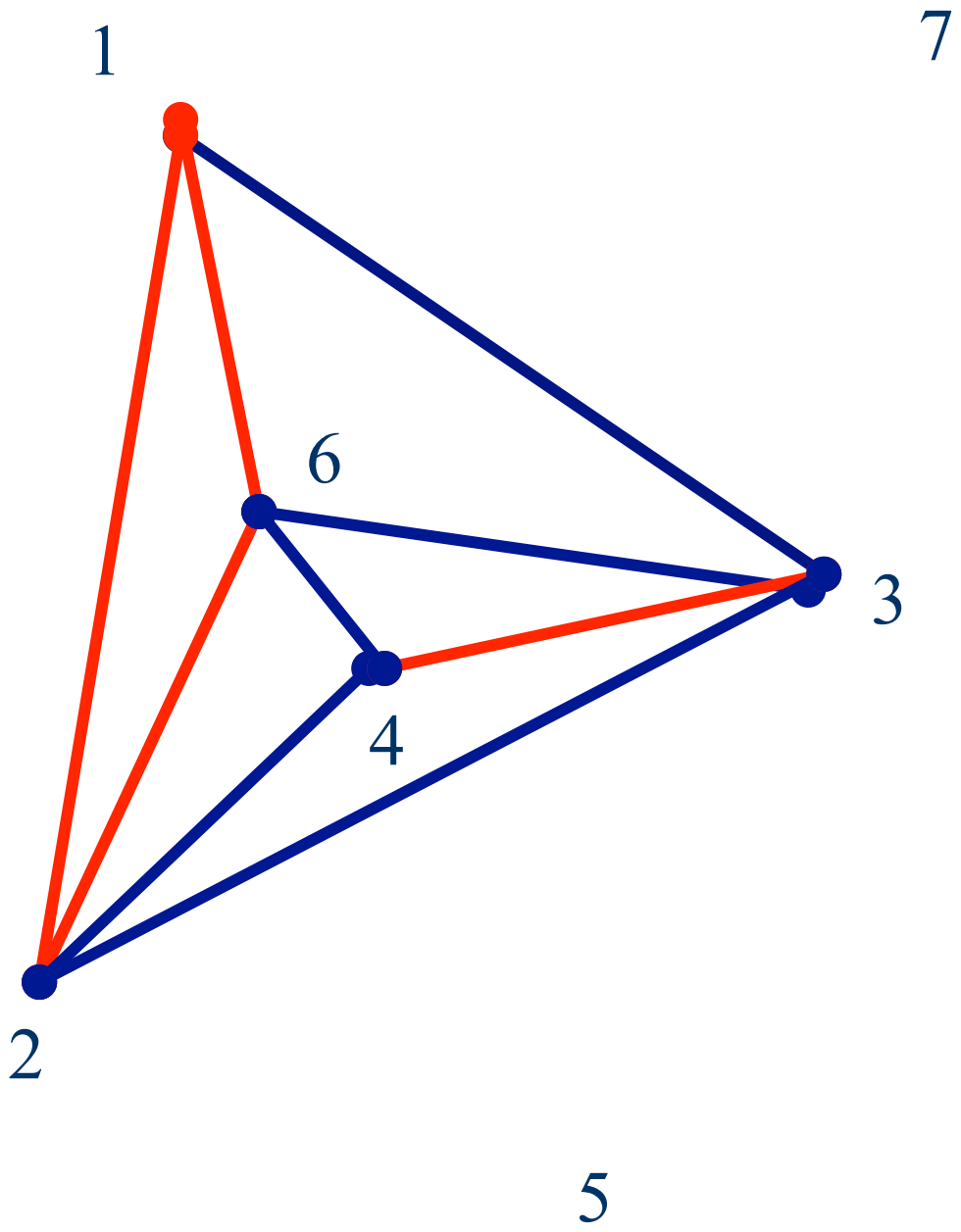


Theorem 1: If there are any dashed edges whose ends are in the same connected component of Gf, at least one edge is in a triangle where the other edges are not dashed, and in every PPH solution, it must be colored so that the triangle has an even number of Blue (out of Phase) edges.

This is an "inferred" coloring


1
7
6
3
4
2
5

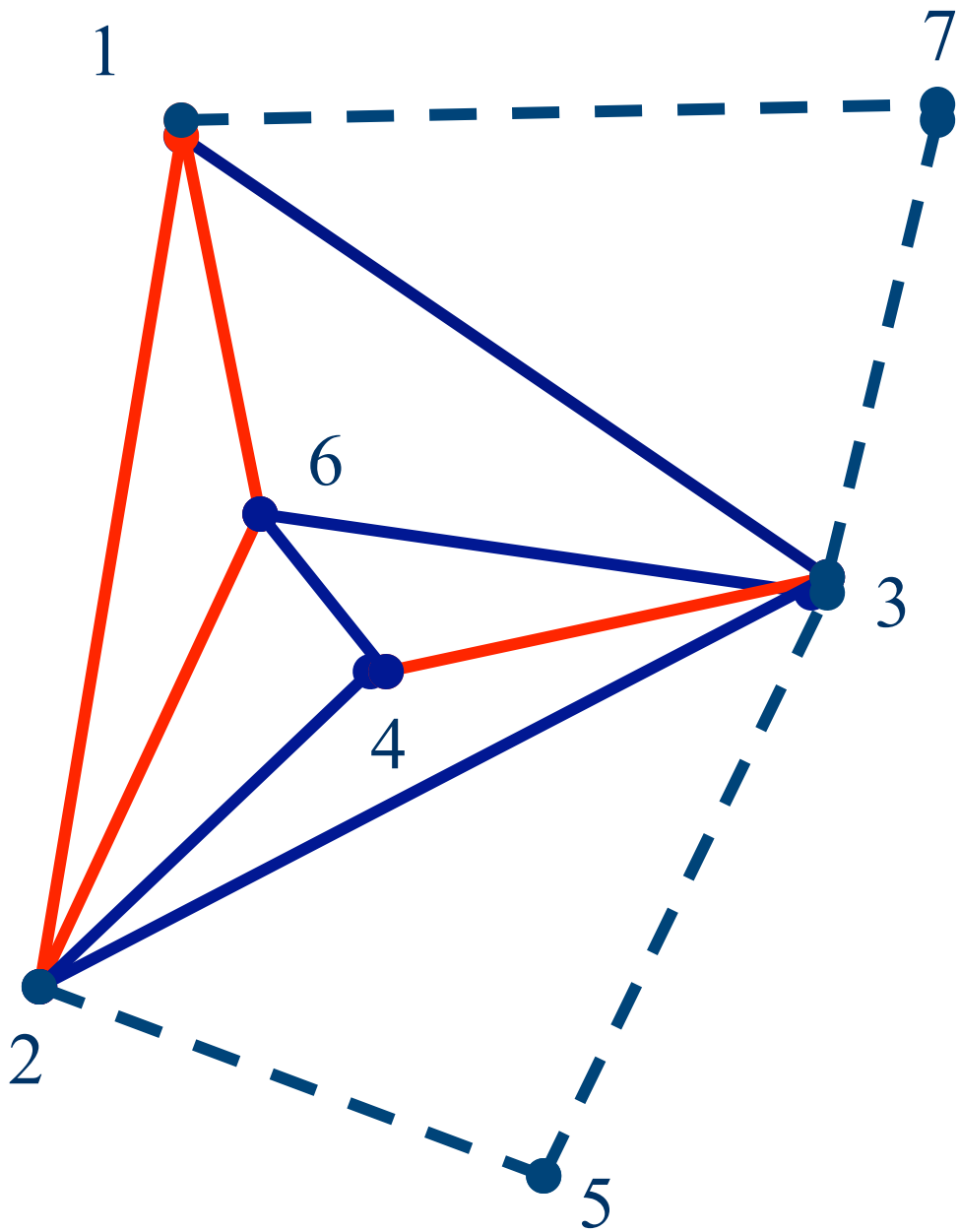

1
7
6
3
4
2
5


1
7
6
3
4
2
5

Corollary

Inside any connected component of Gf, ALL the phase relationships on edges (columns of M) are uniquely determined, either as forced relationships based on pairwise column comparisons,
or by triangle-based inferred colorings.

Hence, the phase relationships of all the columns in a connected component of Gf are INVARIANT over all the solutions to the PPH problem.



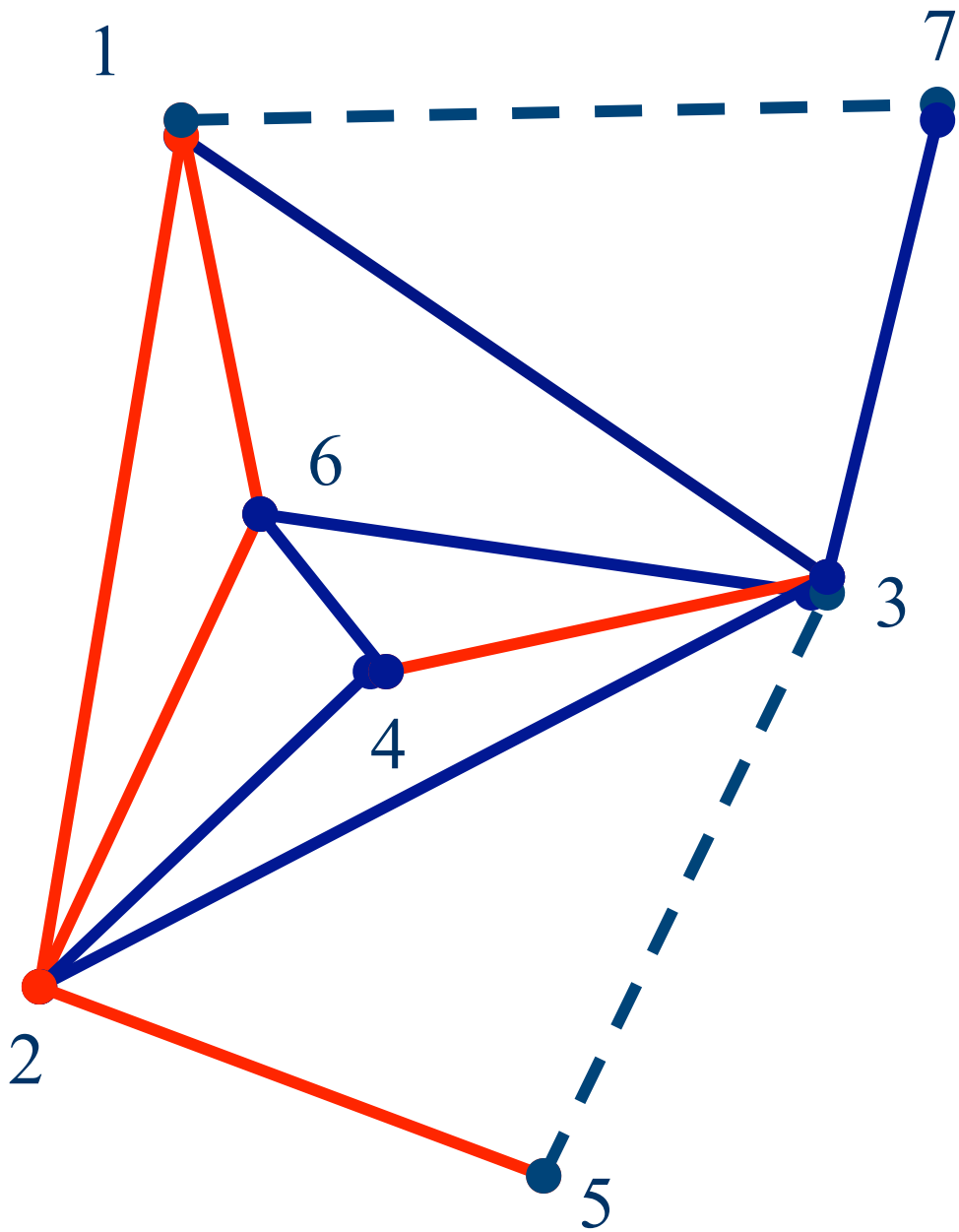
How should we color the remaining dashed edges in a connected component C of Gc?

Answer

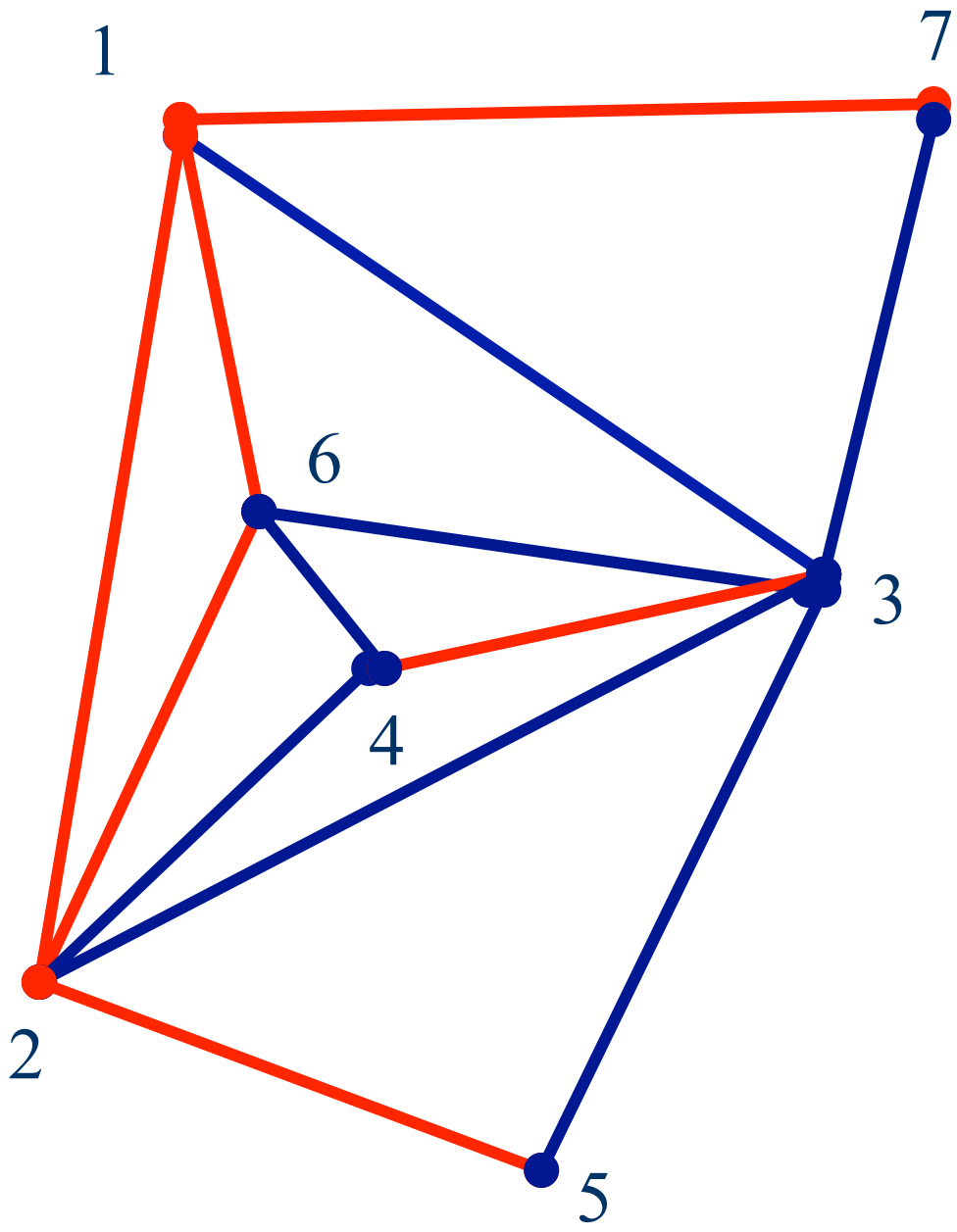
For a connected component C of G with k connected components of Gf, select any
subset S of k-1 dashed edges in C, so that S
together with the red and blue edges span all the nodes of C.

Arbitrarily, color each edge in S either red or blue.

Infer the color of any remaining dashed edges by successive use of the triangle rule.



Pick and color edges (2,5) and (3,7)
The remaining dashed edges are colored by using the triangle rule.


1
7
6
3
4
2
5

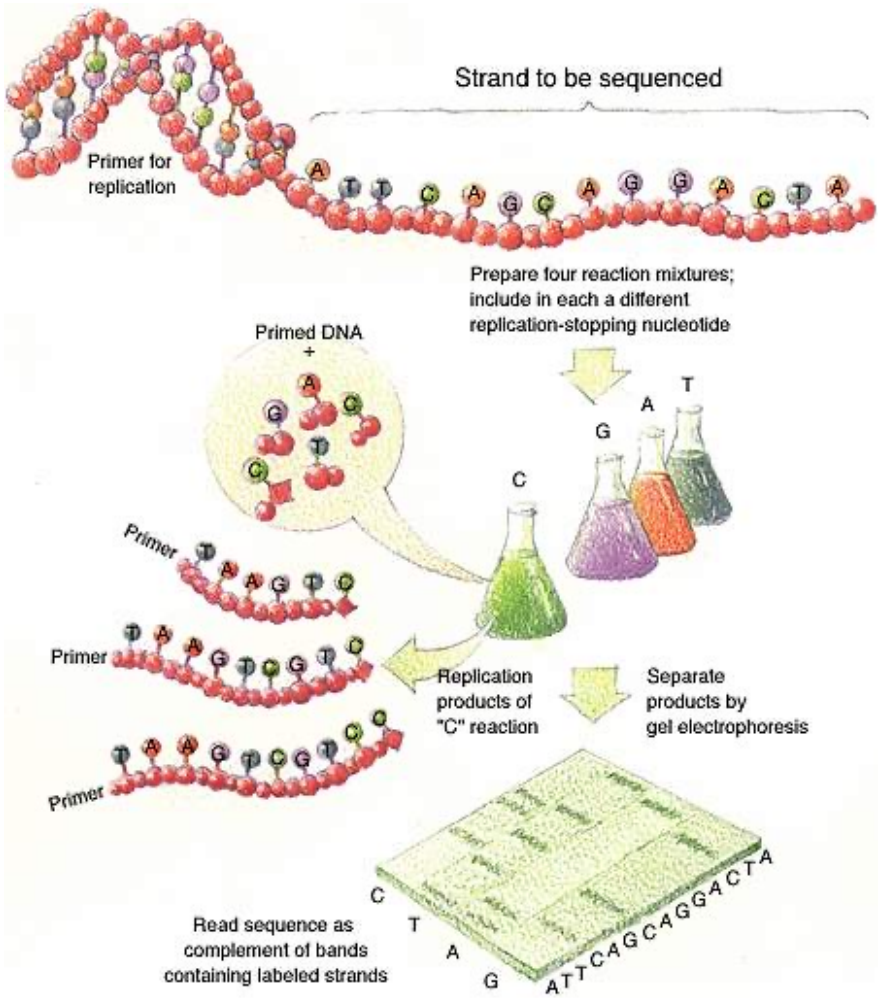
Theorem 2

- Any selected S works (allows the triangle rule to work) and any coloring of the edges in S determines the colors of any remaining dashed edges.
- Different colorings of S determine different colorings of the remaining dashed edges.
- Each different coloring of S determines a different solution to the PPH problem.
- All PPH solutions can be obtained in this way, i.e. using just one selected S set, but coloring it in all 2^{k-1} ways.

Eulerian graph

DNA Sequencing

- Shear DNA into millions of small fragments
- Read 500 – 700 nucleotides at a time from the small fragments (Sanger method)



Fragment Assembly

- **<u>Computational Challenge</u>:** assemble individual short fragments (reads) into a single genomic sequence (“superstring”)

- Until late 1990s the shotgun fragment assembly of human genome was viewed as intractable problem

Shortest superstring problem

- <u>Problem:</u> Given a set of strings, find a shortest string that contains all of them
- <u>Input</u>: Strings $s_1, s_2, \ldots, s_n$
- <u>Output</u>: A string s that contains all strings $s_1, s_2, \ldots, s_n$ as substrings, such that the length of s is minimized

- **Complexity:** NP–hard
- **Note:** this formulation does not take into account sequencing errors

Shortest superstring problem: example

The Shortest Superstring problem

Set of strings: {000, 001, 010, 011, 100, 101, 110, 111}

Concatenation
Superstring 000 001 010 011 100 101 110 111

010
110
011
000

Shortest
superstring 0 0 0 1 1 1 0 1 0 0

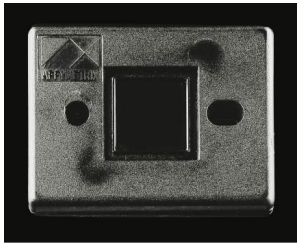
001
111
101
100

Sequencing by Hybridization (SBH): History

- **1988:** SBH is suggested as an an alternative sequencing method.

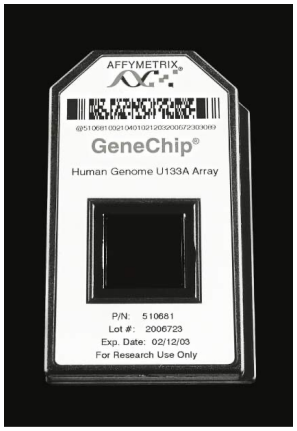
*First microarray prototype **(1989)***

*First commercial DNA microarray prototype w/16,000 features **(1994)***



- **1991:** Light directed polymer synthesis developed by Steve Fodor and colleagues.

*500,000 features per chip **(2002)***



- **1994:** Affymetrix develops first 64-kb DNA microarray

How SBH works

- Attach all possible DNA probes of length *l* to a flat surface, each probe at a distinct and known location. This set of probes is called the DNA array.
- Apply a solution containing fluorescently labeled DNA fragment to the array.
- The DNA fragment hybridizes with those probes that are complementary to substrings of length *l* of the fragment.

How SBH works

- Using a spectroscopic detector, determine which probes hybridize to the DNA fragment to obtain the *l*–mer composition of the target DNA fragment.

- Apply the combinatorial algorithm (below) to reconstruct the sequence of the target DNA fragment from the *l* – mer composition.

Hybridization on DNA Array

Universal DNA Array

	AA	AT	AG	AC	TA	TT	TG	TC	GA	GT	GG	GC	CA	CT	CG	CC
AA																
AT			ATAG													
AG																
AC												ACGC				
TA											TAGG					
TT																
TG																
TC																
GA																
GT																
GG													GGCA			
GC	GCAA															
CA	CAAA															
CT																
CG																
CC																

DNA target TATCCGTTT (complement of ATAGGCAAA)

hybridizes to the array of all 4-mers:

```
A T A G G C A A A
A T A G
  T A G G
    A G G C
      G G C A
        G C A A
          C A A A
```

l-mer composition

- ***Spectrum (s, l)*** - *unordered* multiset of all possible *(n – l + 1)* *l*-mers in a string *s* of length *n*
- The order of individual elements in *Spectrum (s, l)* does not matter
- For *s* = TATGGTGC all of the following are equivalent representations of *Spectrum (s, 3):*

  {TAT, ATG, TGG, GGT, GTG, TGC}

  {ATG, GGT, GTG, TAT, TGC, TGG}

  {TGG, TGC, TAT, GTG, GGT, ATG}

l-mer composition

- ***Spectrum (s, l)*** - *unordered* multiset of all possible *(n – l + 1)* *l*-mers in a string *s* of length *n*
- The order of individual elements in *Spectrum (s, l)* does not matter
- For *s* = TATGGTGC all of the following are equivalent representations of *Spectrum (s, 3):*

  {TAT, ATG, TGG, GGT, GTG, TGC}

  {ATG, GGT, GTG, TAT, TGC, TGG}

  {TGG, TGC, TAT, GTG, GGT, ATG}
- We usually choose the lexicographically maximal representation as the canonical one.

Observations: different sequences may have the same spectrum

- Different sequences may have the same spectrum:

  Spectrum(GTATCT,2)=

  Spectrum(GTCTAT,2)=

  {AT, CT, GT, TA, TC}

The SBH problem

- <u>Goal</u>: Reconstruct a string from its *l*-mer composition

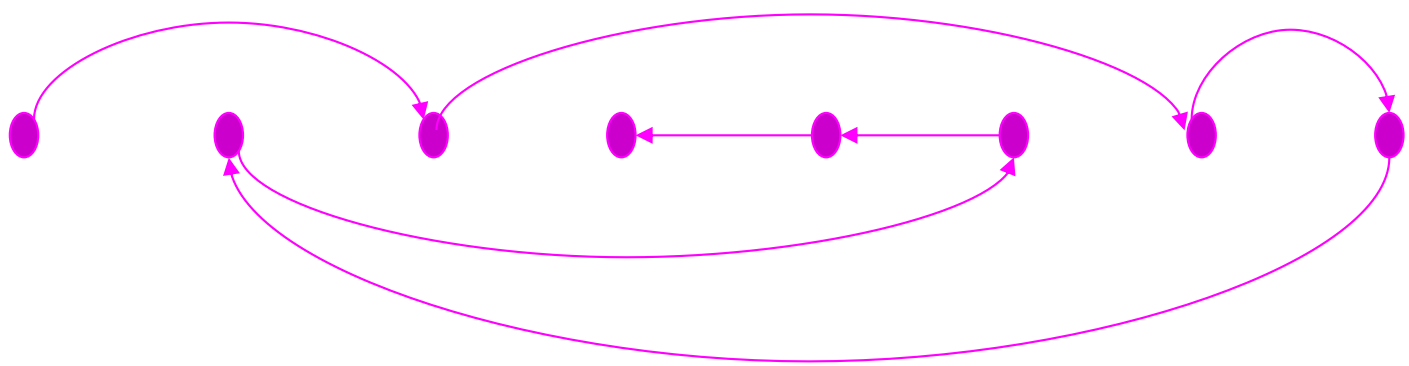
- <u>Input</u>: A set *S*, representing all *l*-mers from an (unknown) string *s*

- <u>Output</u>: String *s* such that *Spectrum (s,l)* = *S*

SBH: Hamiltonian path approach

S = { ATG AGG TGC TCC GTC GGT GCA CAG }

H ATG AGG TGC TCC GTC GGT GCA CAG



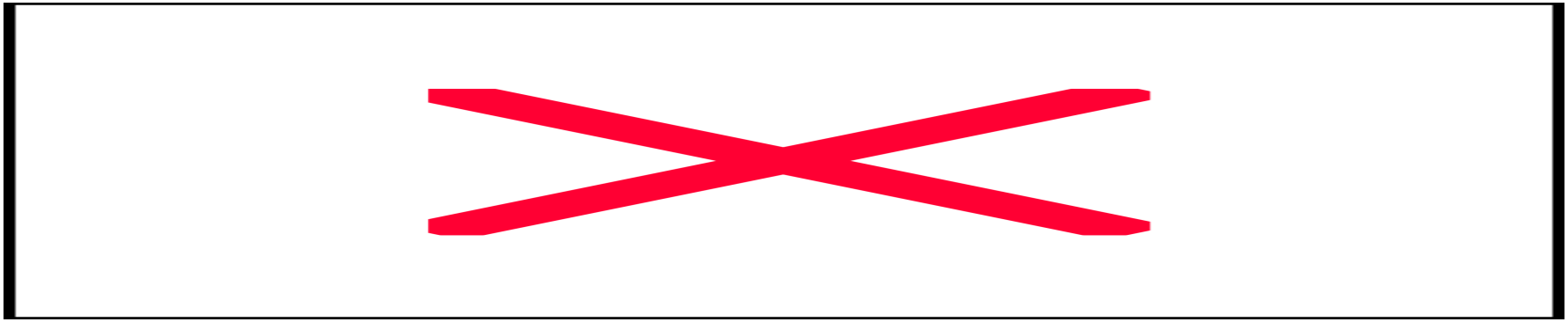
ATG CAG GTC C

Path visited every VERTEX once

SBH: Hamiltonian path approach

A more complicated graph:

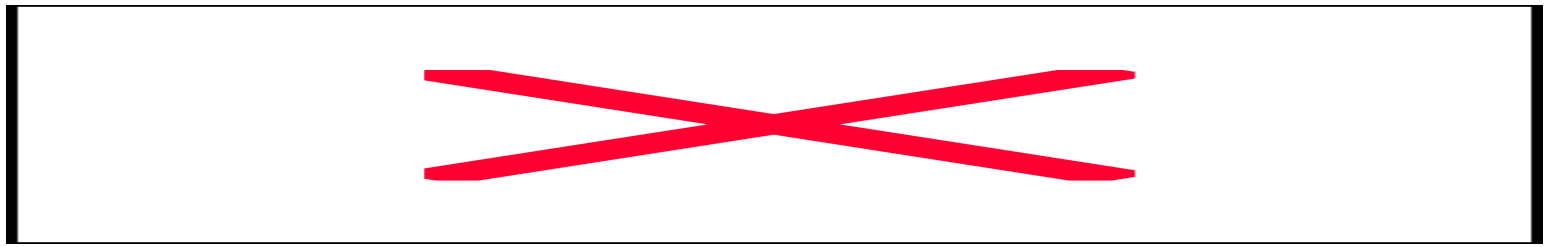
S = { ATG TGG TGC GTG GGC GCA GCG CGT }



SBH: Hamiltonian path approach

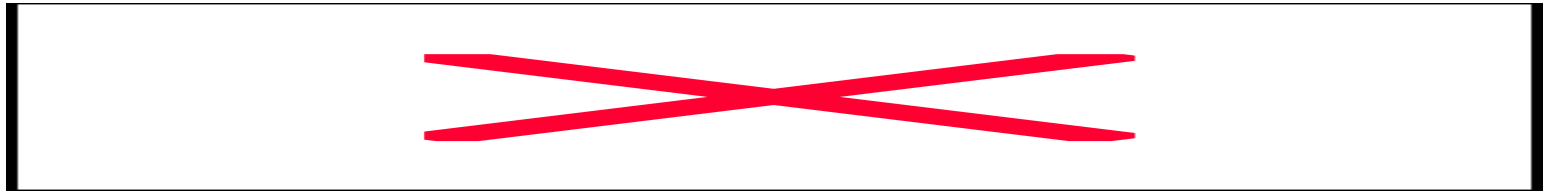
S = { ATG TGG TGC GTG GGC GCA GCG CGT }

Path



ATGCGTGGCA

Path 2:

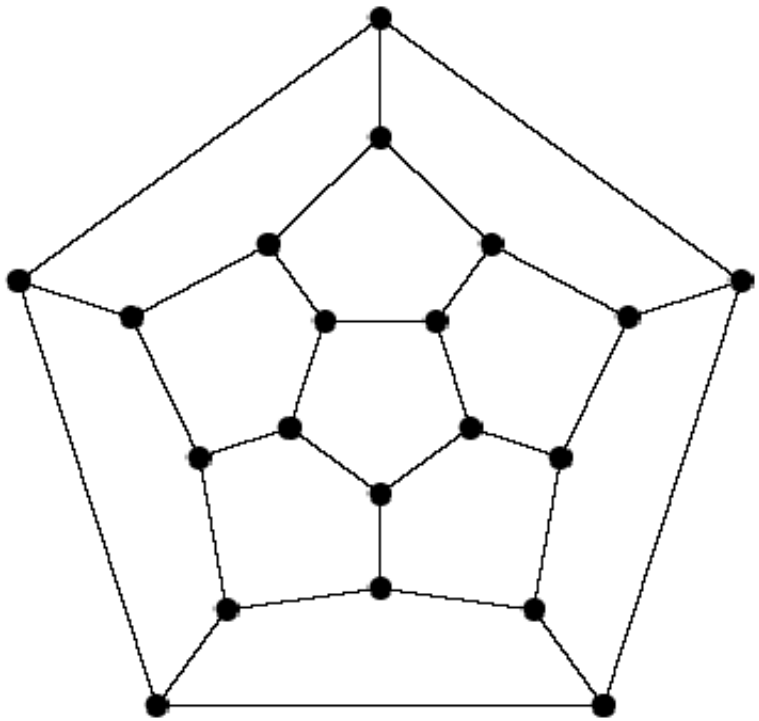


ATGGCGTGCA

Hamiltonian cycle problem

- Find a cycle that visits every ***vertex*** exactly once

- NP–complete

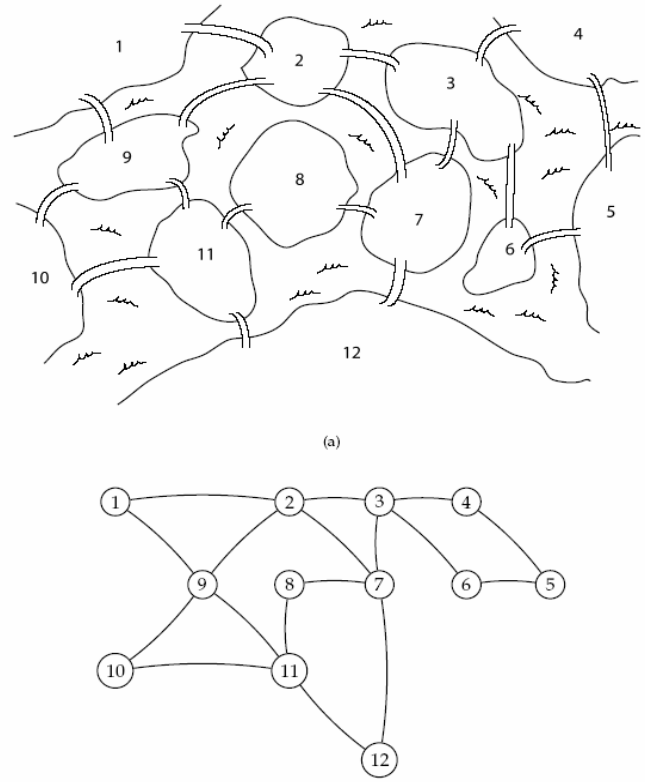


Game invented by Sir William Hamilton in 1857

Eulerian cycle problem

- Find a cycle that visits every ***edge*** exactly once

- Linear time

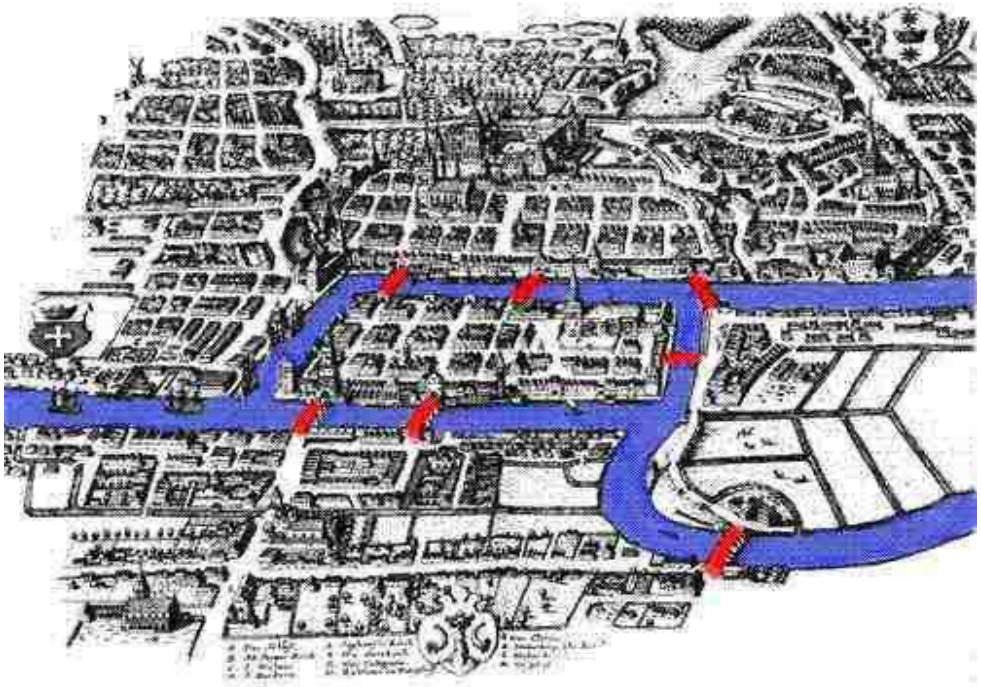


More complicated Königsberg

The Bridge Obsession Problem

Find a tour crossing every bridge just once

Leonhard Euler, 1735



Bridges of Königsberg

Euler Theorem

- A graph is balanced if for every vertex the number of incoming edges equals to the number of outgoing edges:

$$in(v)=out(v)$$

- **Theorem**: *A connected graph is Eulerian if and only if each of its vertices is balanced.*

Euler Theorem: proof

- Eulerian → balanced

  for every edge entering *v* (incoming edge) there exists an edge leaving *v* (outgoing edge). Therefore

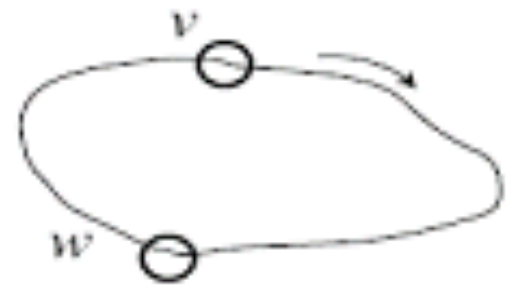
$$in(v)=out(v)$$

- Balanced → Eulerian

  ???

Algorithm for Constructing an Eulerian Cycle

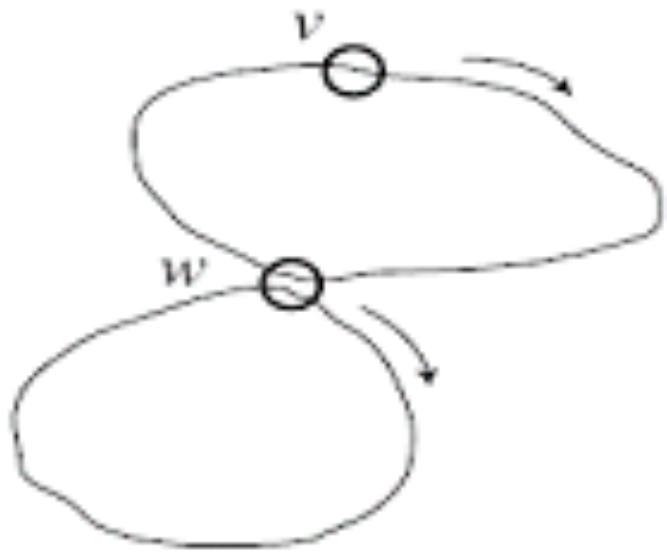
a. Start with an arbitrary vertex *v* and form an arbitrary cycle with unused edges until a dead end is reached. Since the graph is Eulerian this dead end is necessarily the starting point, i.e., vertex *v*.



(a)

Algorithm for Constructing an Eulerian Cycle (cont'd)

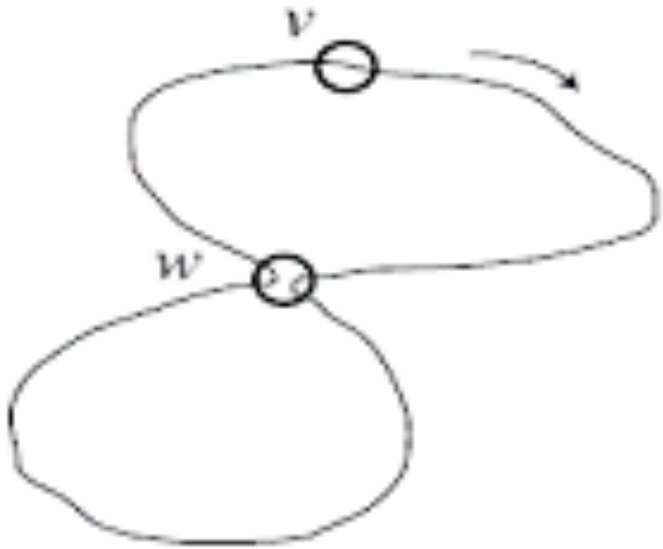
b. If cycle from (a) above is not an Eulerian cycle, it must contain a vertex *w*, which has untraversed edges. Perform step (a) again, using vertex *w* as the starting point. Once again, we will end up in the starting vertex *w.*



(b)

Algorithm for Constructing an Eulerian Cycle

c. Combine the cycles from (a) and (b) into a single cycle and iterate step (b).



(c)

Euler Theorem: extension

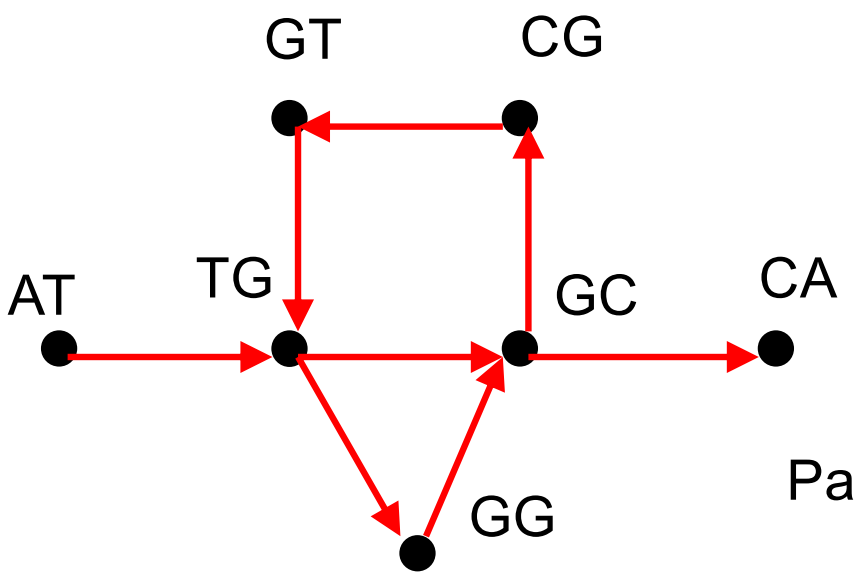
- **Theorem**: *A connected graph has an Eulerian path if and only if it contains at most two semi-balanced vertices and all other vertices are balanced.*

SBH: Eulerian path approach

S = { ATG, TGC, GTG, GGC, GCA, GCG, CGT , TGG }

Vertices correspond to (l – 1) – mers : { AT, TG, GC, GG, GT, CA, CG }

Edges correspond to *l* – mers from *S*

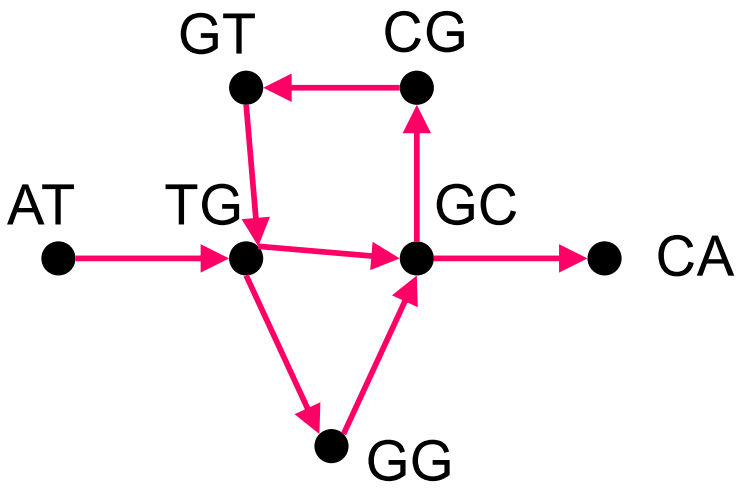


De Bruijn graph

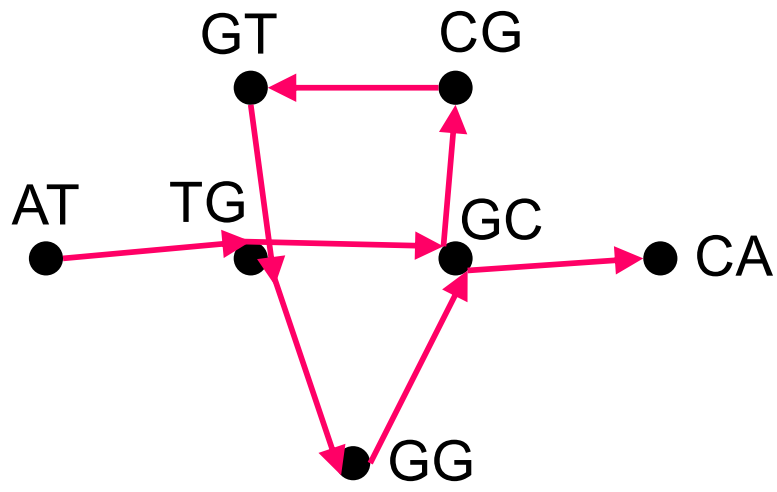
Path visited every EDGE once

SBH: multiple solutions

S = { AT, TG, GC, GG, GT, CA, CG } corresponds to two different paths:



ATGGCGTGCA



ATGCGTGGCA

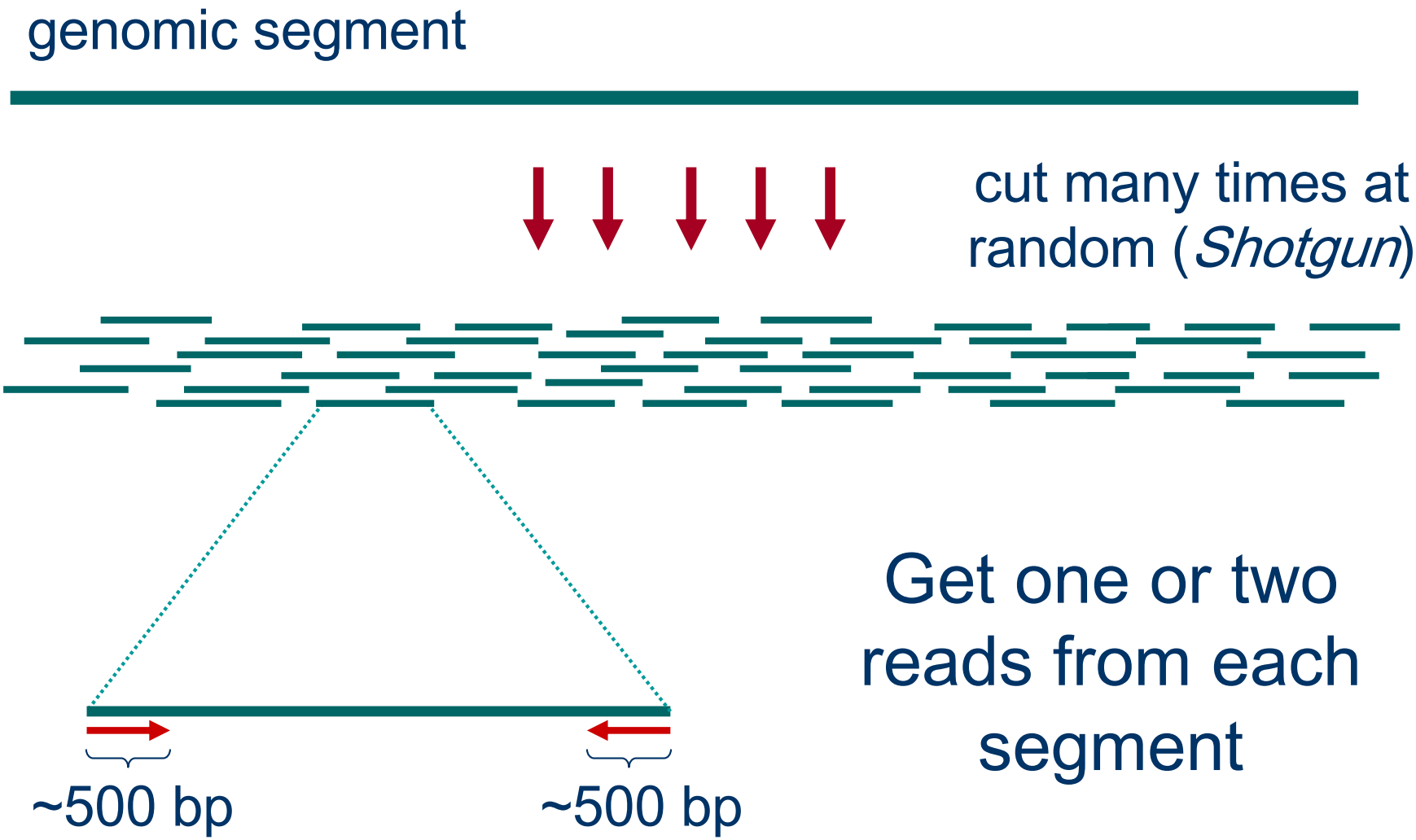
Representing alternative SBH solutions by Eulerian graphs

- # of solutions ←→ # of Eulerian cycles
  - BEST theorem

- Obtains all solutions from one solution → Cassette transformations

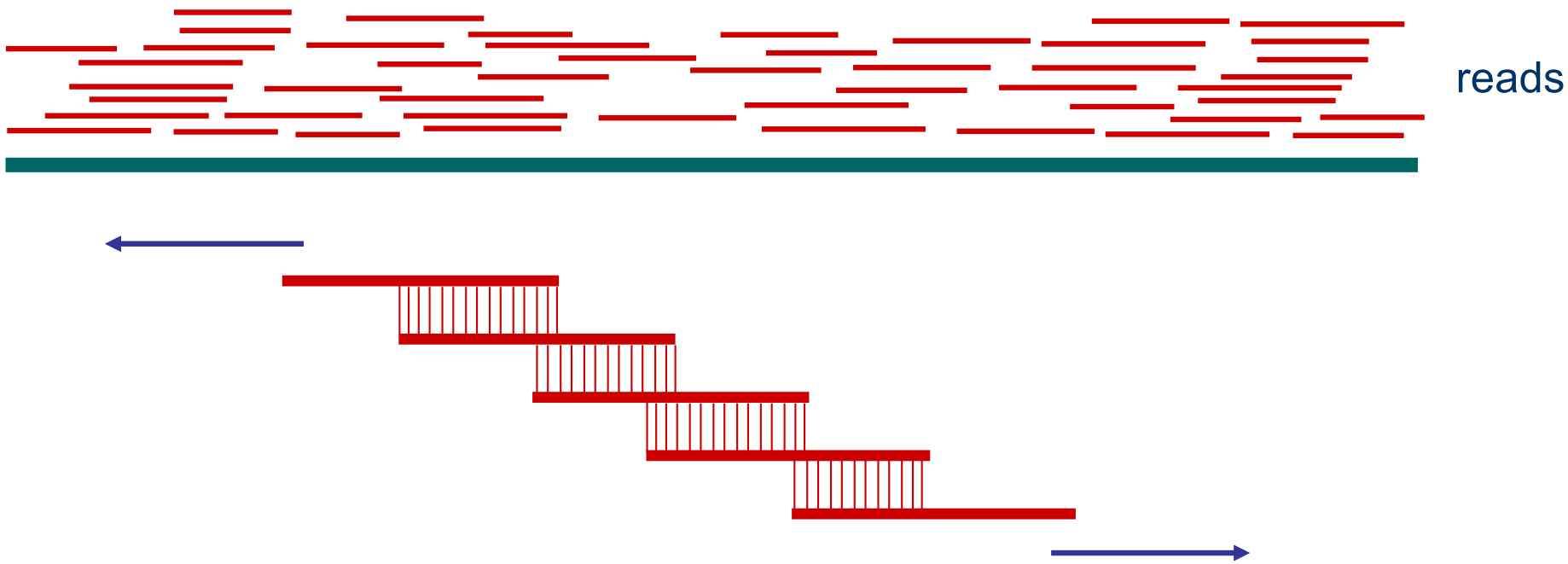
Reads assembly

Shotgun Sequencing



Get one or two reads from each segment

Fragment Assembly



Cover region with ~7-fold redundancy

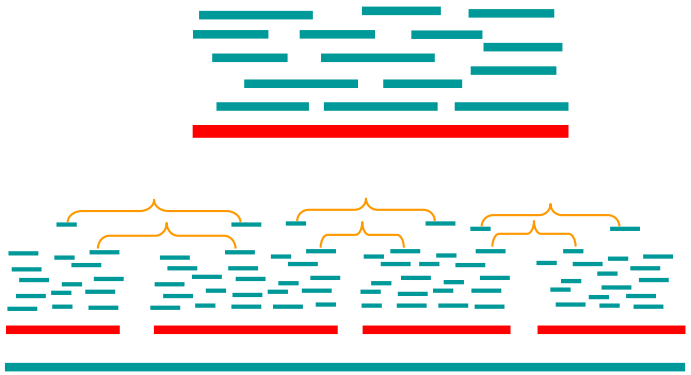
Overlap reads and extend to reconstruct the original genomic region

Overlap-Layout-Consensus

Assemblers: ARACHNE, PHRAP, CAP, TIGR, CELERA

Overlap: find potentially overlapping reads

Layout: merge reads into contigs and contigs into supercontigs



Consensus: derive the DNA sequence and correct read errors

..ACGATTACAATAGGTT..

Overlap

- Find the best match between the suffix of one read and the prefix of another
- Due to sequencing errors, need to use dynamic programming to find the optimal *overlap alignment*
- Apply a filtration method to filter out pairs of fragments that do not share a significantly long common substring

Overlapping Reads

- Sort all *k*-mers in reads (*k* ~ 24)
- Find pairs of reads sharing a k-mer
- Extend to full alignment – throw away if not >95% similar

Overlapping Reads and Repeats

- A *k*-mer that appears N times, initiates N^2 comparisons

- For an *Alu* that appears 10^6 times → 10^{12} comparisons – too much

- **<u>Solution:</u>**

  Discard all *k*-mers that appear more than

  $t \times$ Coverage, ($t \sim 10$)

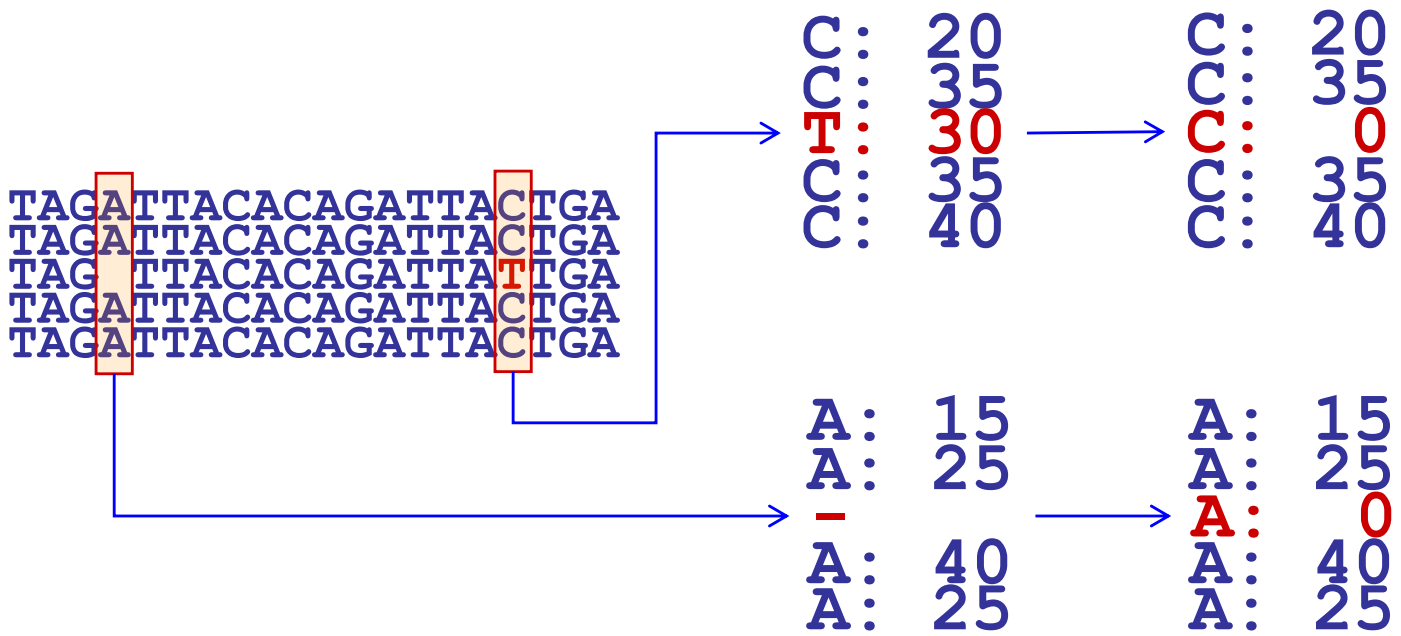
Finding Overlapping Reads

Create local multiple alignments from the overlapping reads

Finding Overlapping Reads

(cont'd)

- Correct errors using multiple alignment

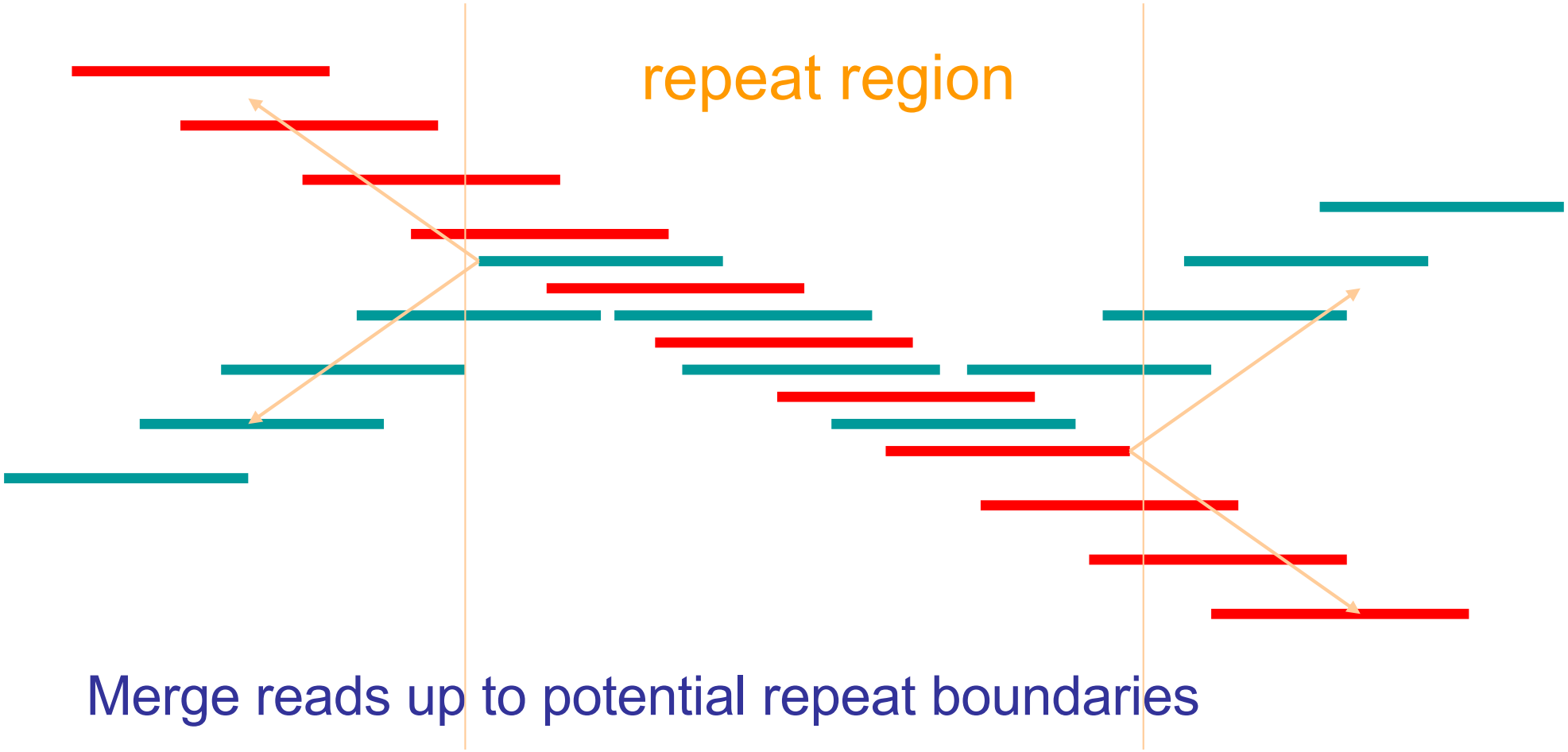


- Score alignments
- Accept alignments with good scores

Layout

- Repeats are a major challenge
- Do two aligned fragments really overlap, or are they from two copies of a repeat?
- Solution: repeat masking – hide the repeats!!!
- Masking results in high rate of misassembly (up to 20%)
- Misassembly means alot more work at the finishing step

Merge Reads into Contigs



Merge reads up to potential repeat boundaries

Repeats, Errors, and Contig Lengths

- Repeats shorter than read length are OK
- Repeats with more base pair differencess than sequencing error rate are OK
- To make a smaller portion of the genome **appear** repetitive, try to:
  - Increase read length
  - Decrease sequencing error rate

Error Correction

Role of error correction:

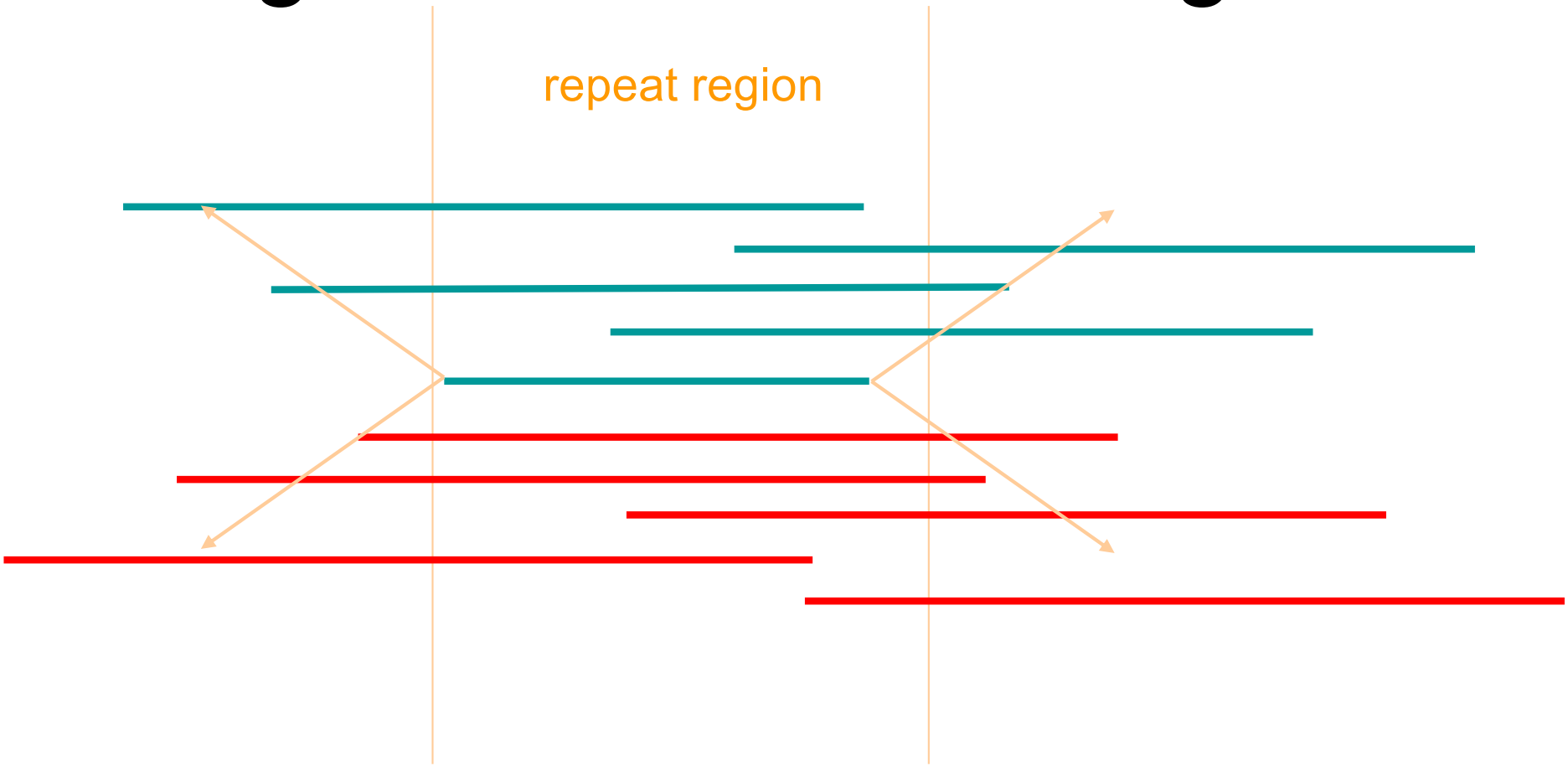
Discards ~90% of single-letter sequencing errors

decreases error rate

$\Rightarrow$ decreases effective repeat content

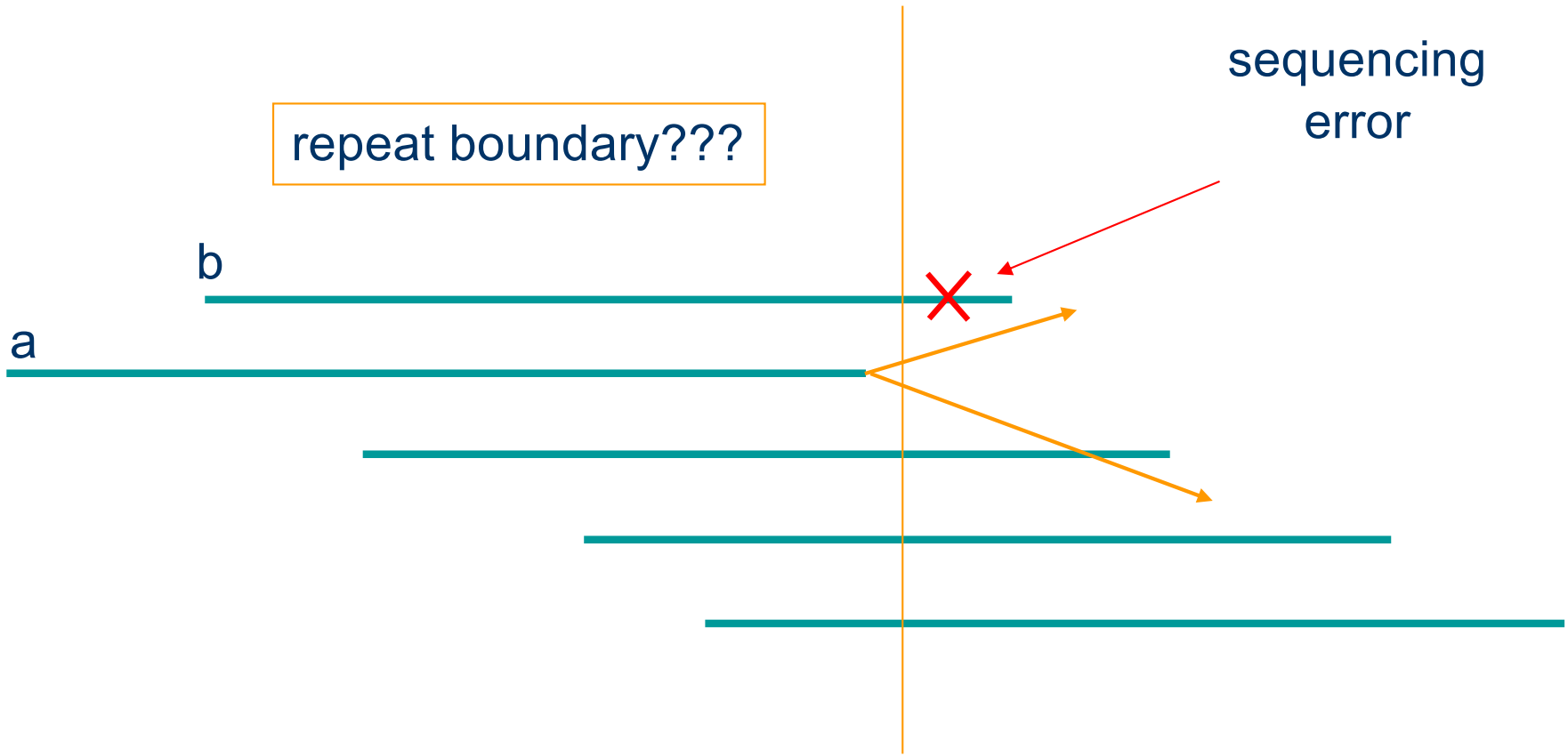
$\Rightarrow$ increases contig length

Merge Reads into Contigs (cont'd)



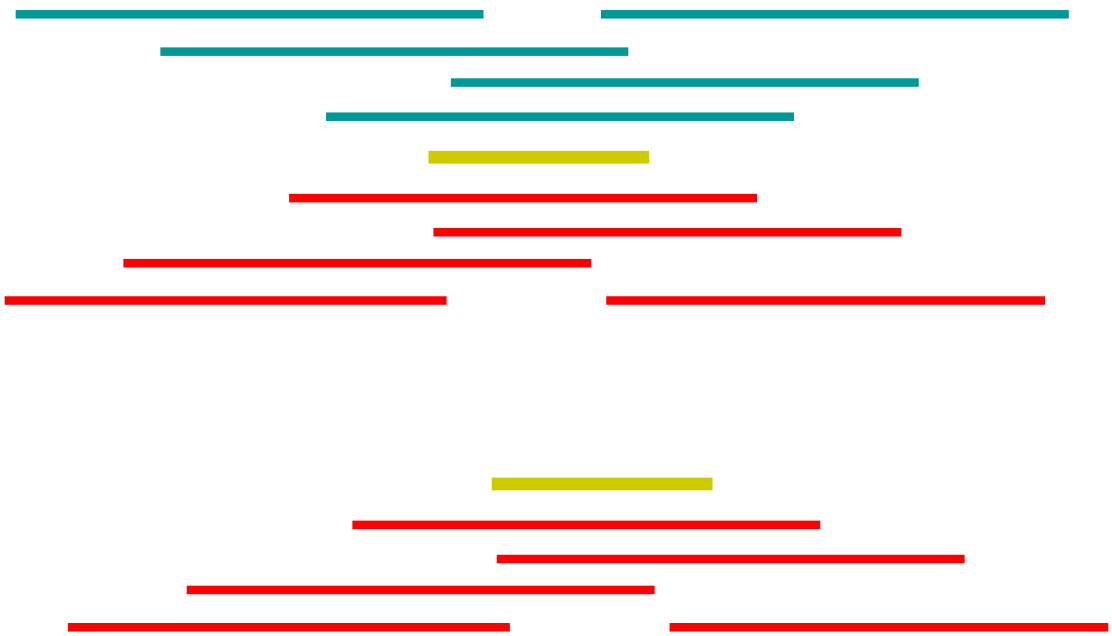
- Ignore non-maximal reads
- Merge only maximal reads into contigs

Merge Reads into Contigs (cont'd)



- Ignore "hanging" reads, when detecting repeat boundaries

Merge Reads into Contigs (cont'd)



?????

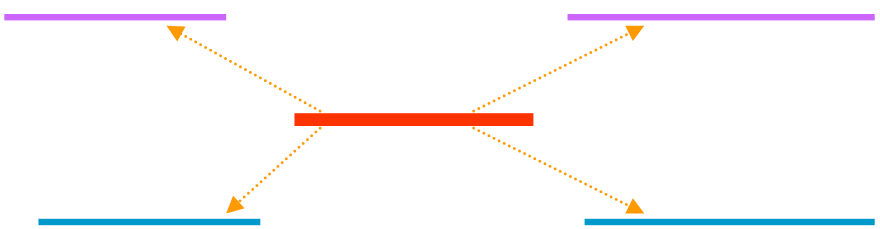
Unambiguous

- Insert non-maximal reads whenever unambiguous

Link Contigs into Supercontigs

Normal density

Too dense:
Overcollapsed?

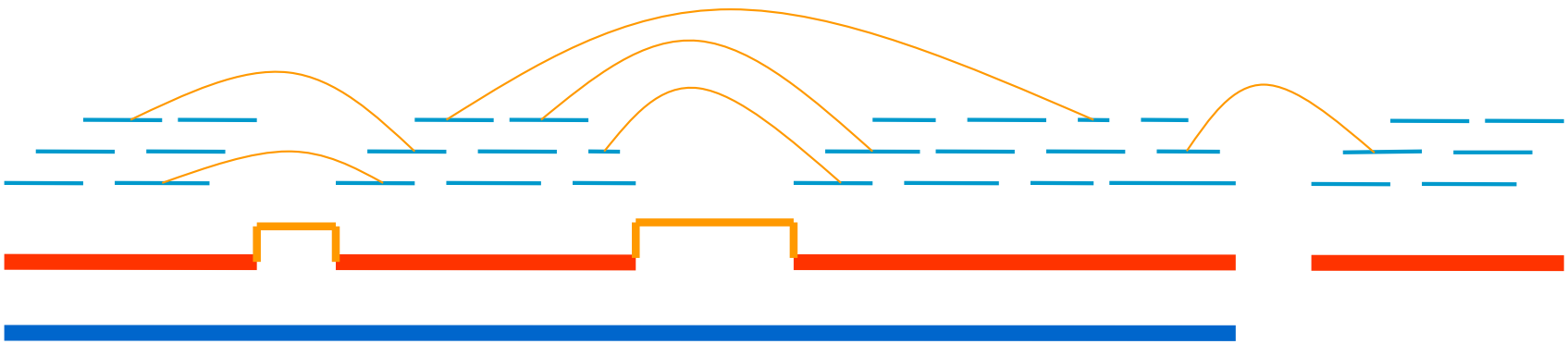


Inconsistent links:
Overcollapsed?

Link Contigs into Supercontigs
(cont'd)

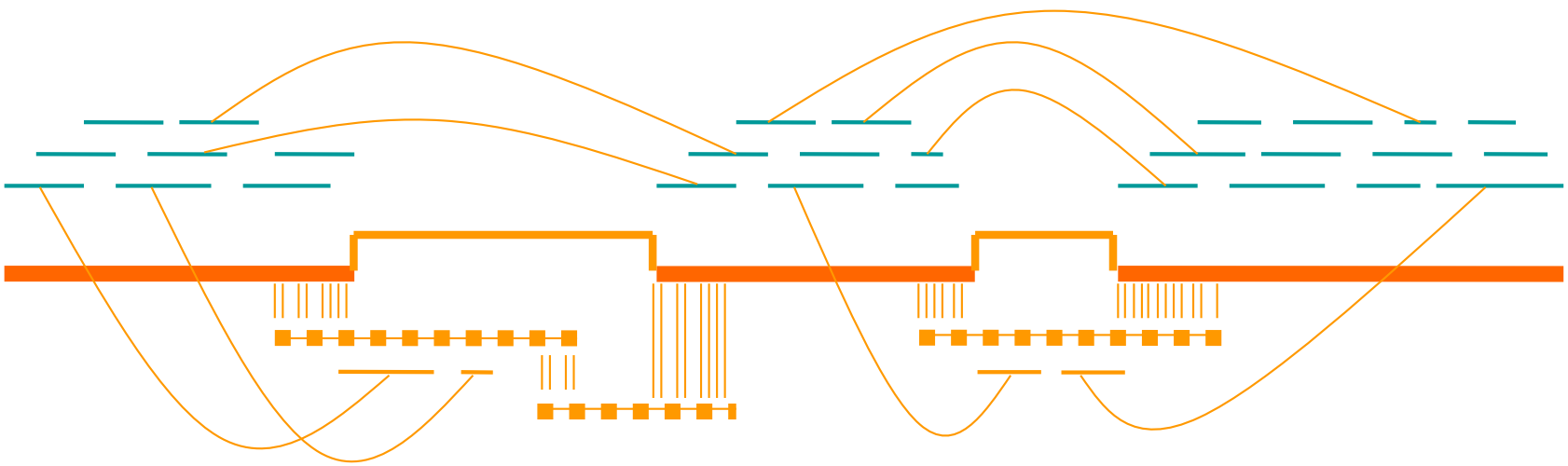
Find all links between unique contigs

Connect contigs incrementally, if ≥ 2 links

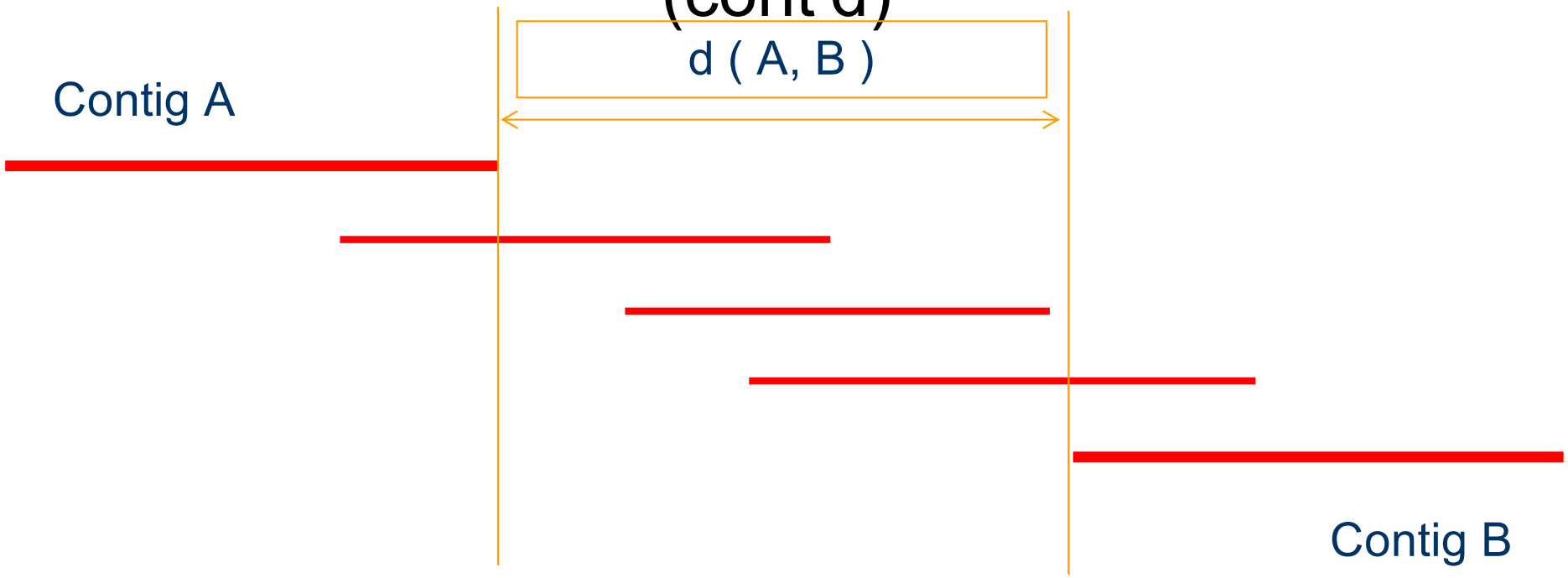


Link Contigs into Supercontigs
(cont'd)

Fill gaps in supercontigs with paths of overcollapsed contigs



Link Contigs into Supercontigs (cont'd)



Define G = (V, E)

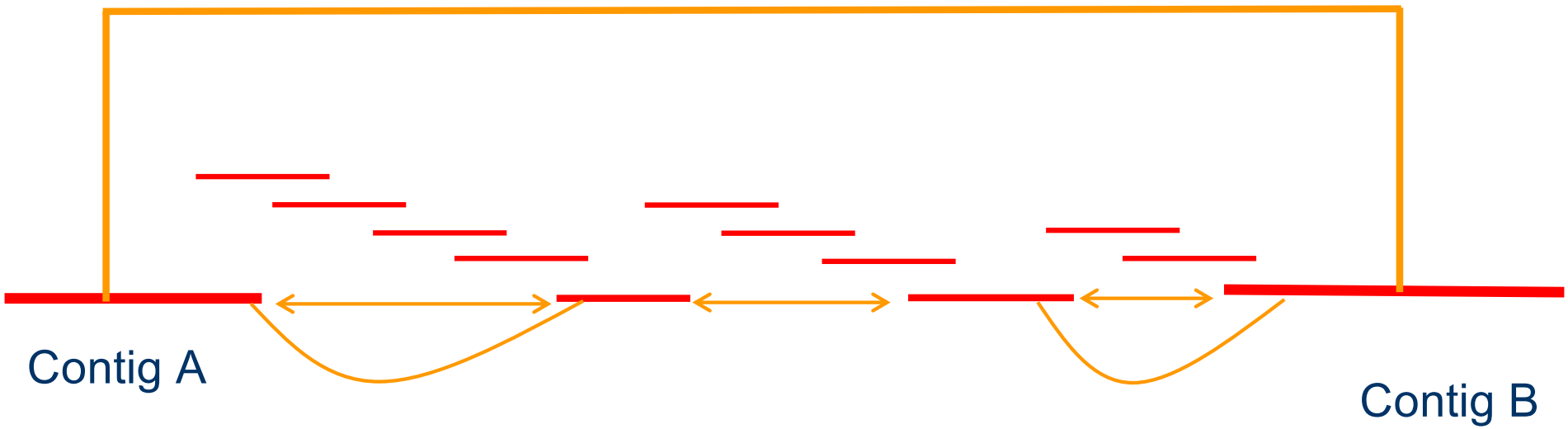
V := contigs

E := (A, B) such that d(A, B) < C

<u>Reason to do so</u>: Efficiency; full shortest paths cannot be computed

Link Contigs into Supercontigs

(cont'd)



Define T: contigs linked to either A or B

Fill gap between A and B if there is a path in G passing only from contigs in T

Consensus

- A consensus sequence is derived from a profile of the assembled fragments
- A sufficient number of reads is required to ensure a statistically significant consensus
- Reading errors are corrected

Derive Consensus Sequence

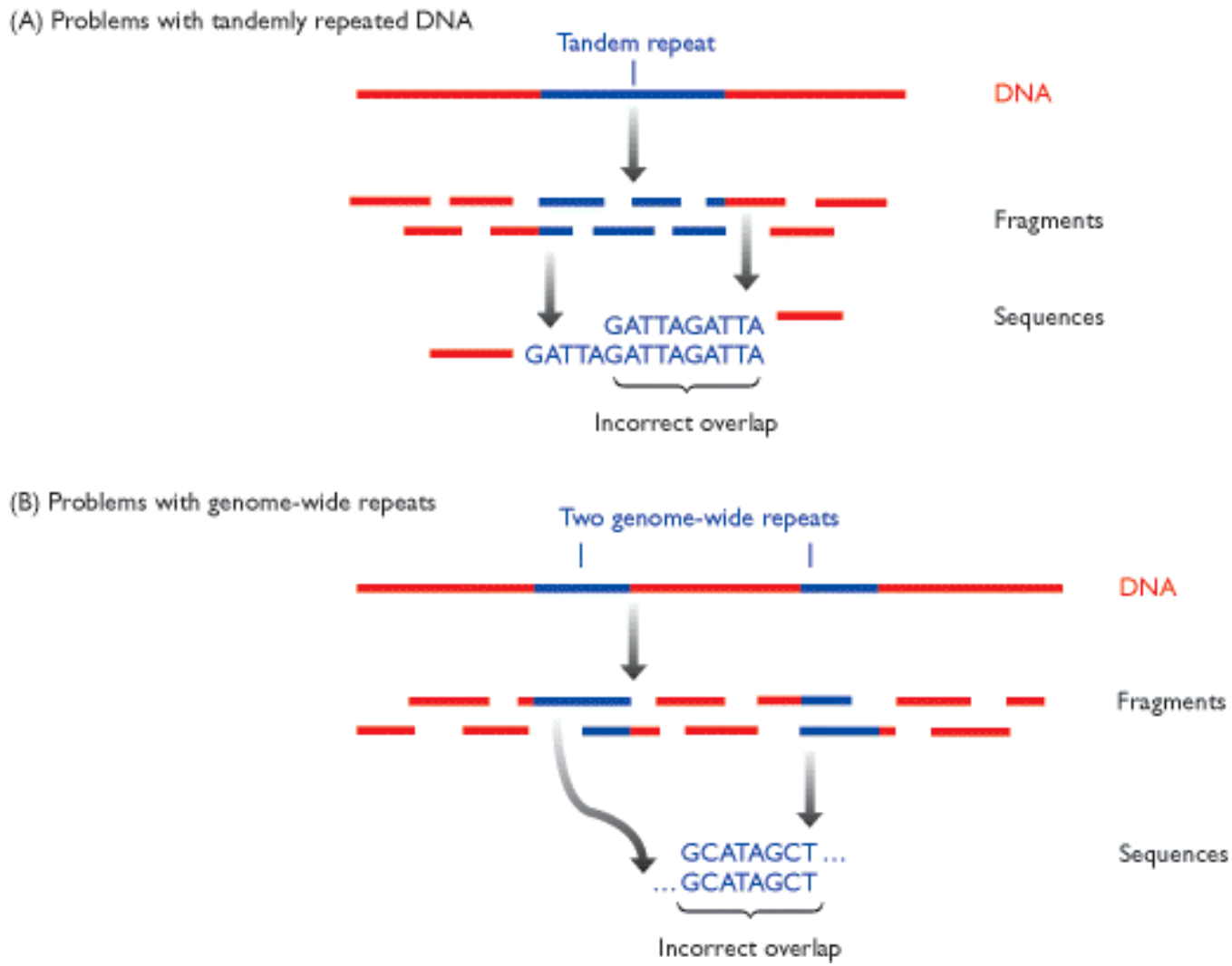
```
TAGATTACACAGATTACTGA TTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAAACTA
TAG TTACACAGATTATTGACTTCATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGGGTAA CTA
```

```
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
```

Derive multiple alignment from pairwise read alignments

Derive each consensus base by weighted voting

Problems with the shotgun approach

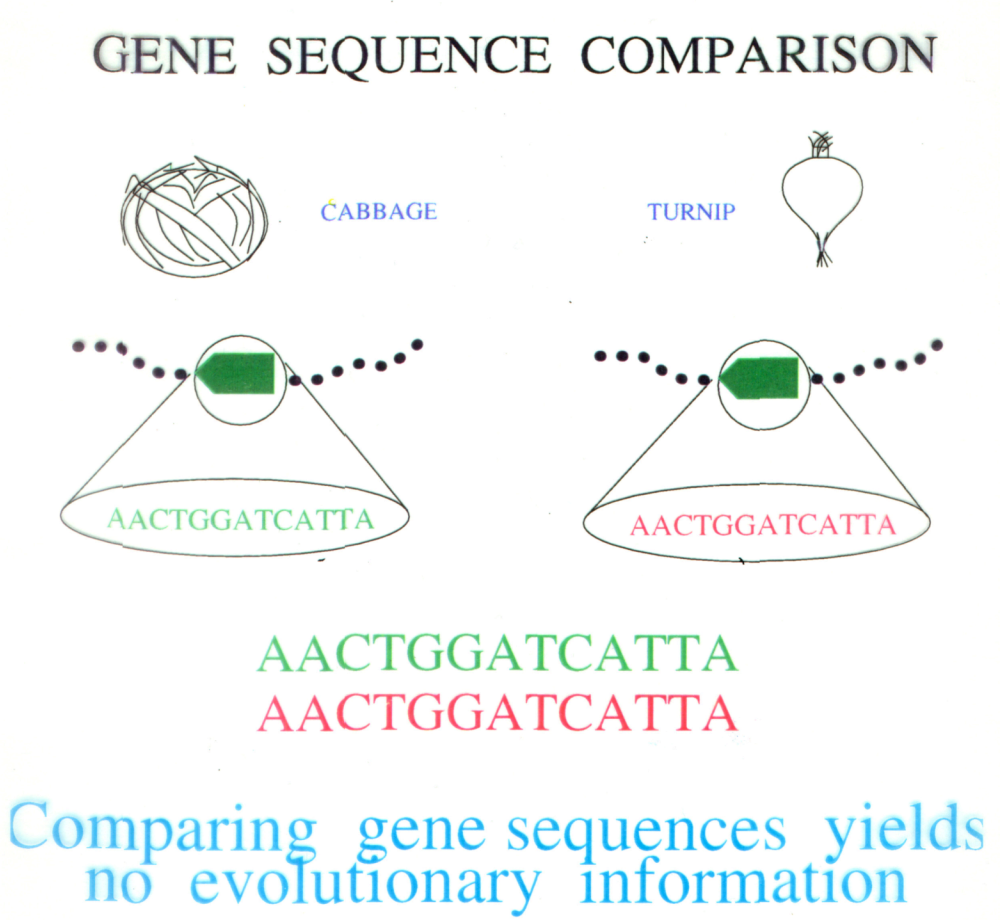


Genome Rearrangements

Turnip vs Cabbage: Look and Taste Different

- Although cabbages and turnips share a recent common ancestor, they look and taste different

Turnip vs Cabbage: Comparing Gene Sequences Yields No Evolutionary Information

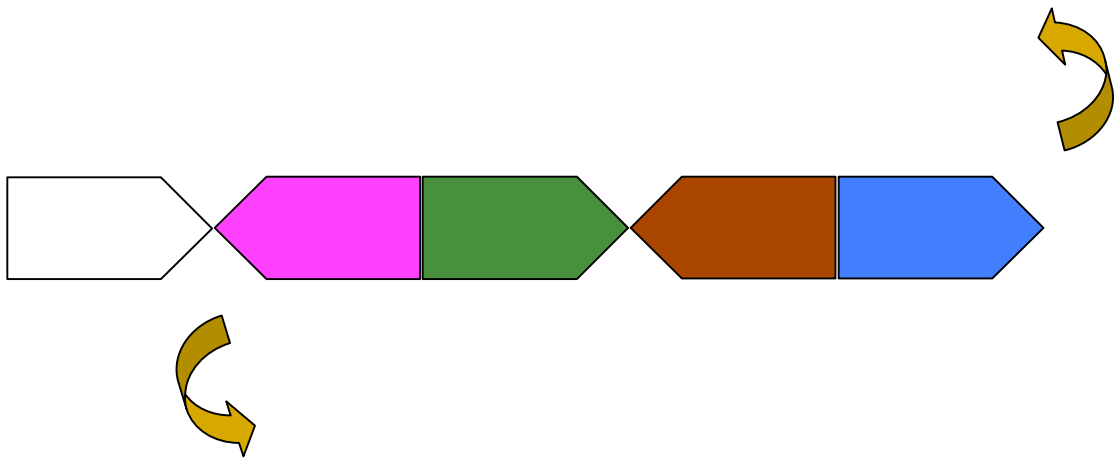


Turnip vs Cabbage: Almost Identical mtDNA gene sequences

- In 1980s Jeffrey Palmer studied evolution of plant organelles by comparing mitochondrial genomes of the cabbage and turnip
- 99% similarity between genes
- These surprisingly identical gene sequences differed in gene order
- This study helped pave the way to analyzing genome rearrangements in molecular evolution

Turnip vs Cabbage: Different mtDNA Gene Order

- Gene order comparison:



Turnip vs Cabbage: Different mtDNA Gene Order

- Gene order comparison:

Turnip vs Cabbage: Different mtDNA Gene Order

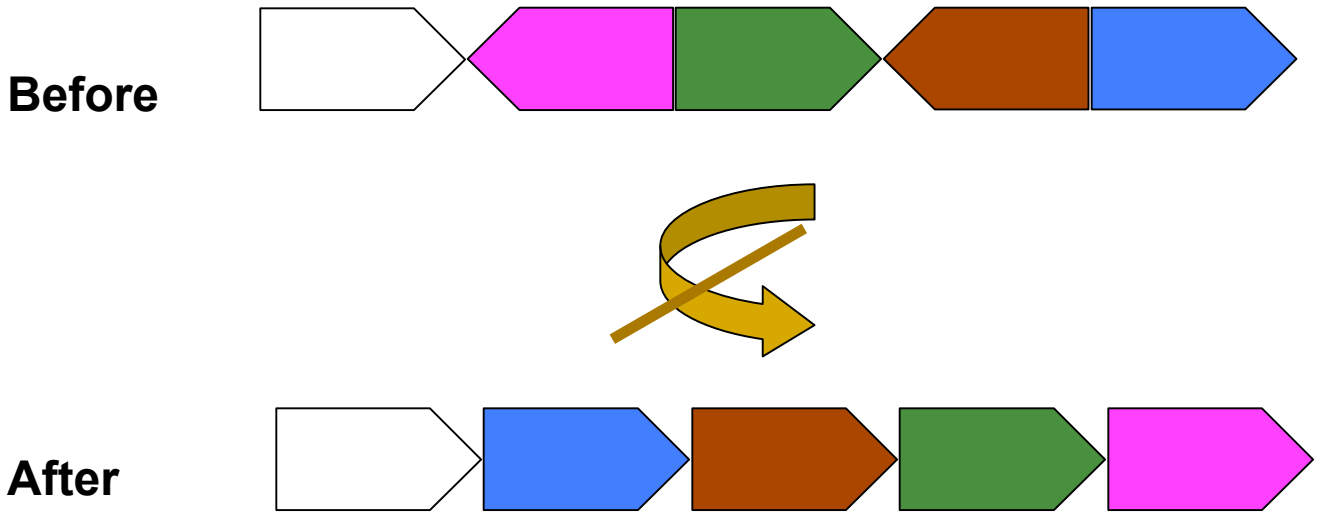
- Gene order comparison:

Turnip vs Cabbage: Different mtDNA Gene Order

- Gene order comparison:

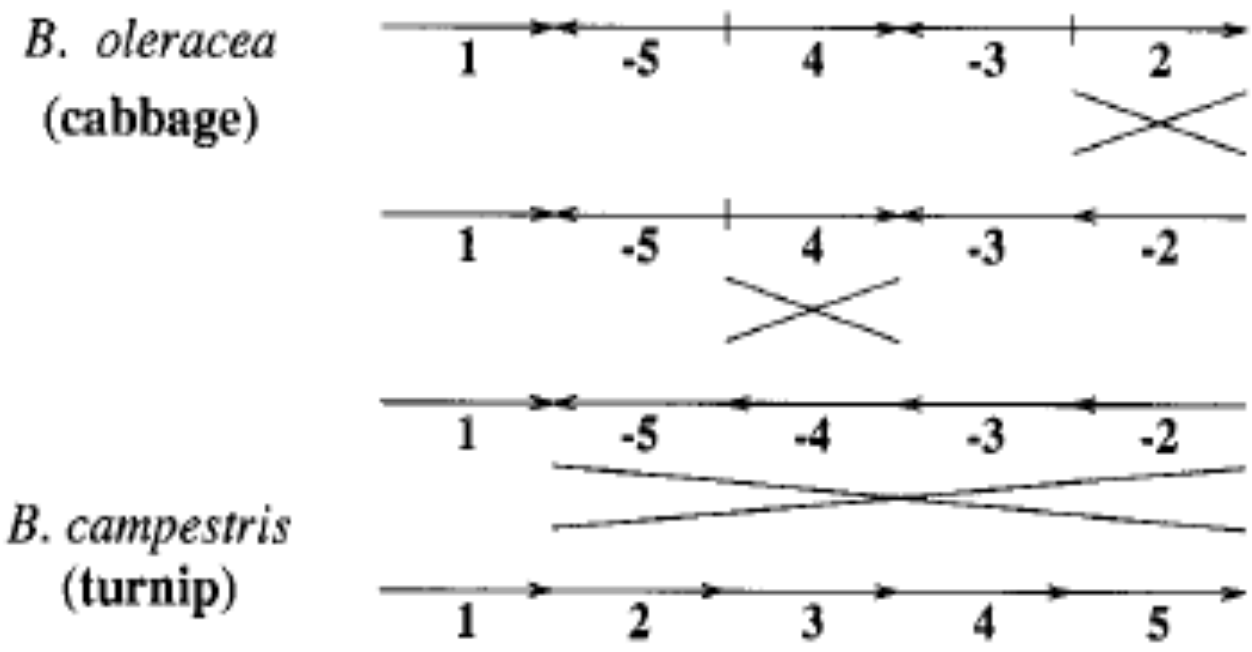
Turnip vs Cabbage: Different mtDNA Gene Order

- Gene order comparison:

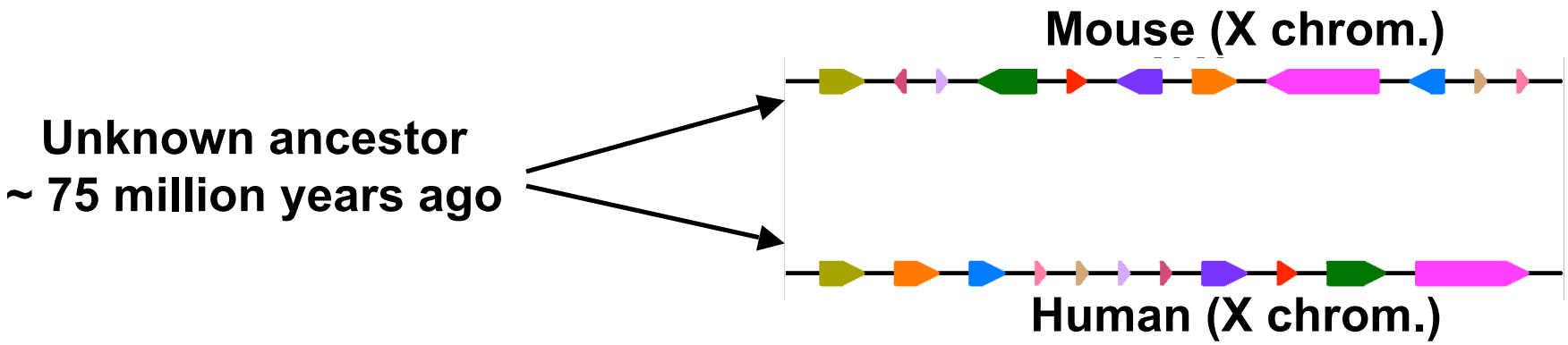


Evolution is manifested as the divergence in gene order

Transforming Cabbage into Turnip

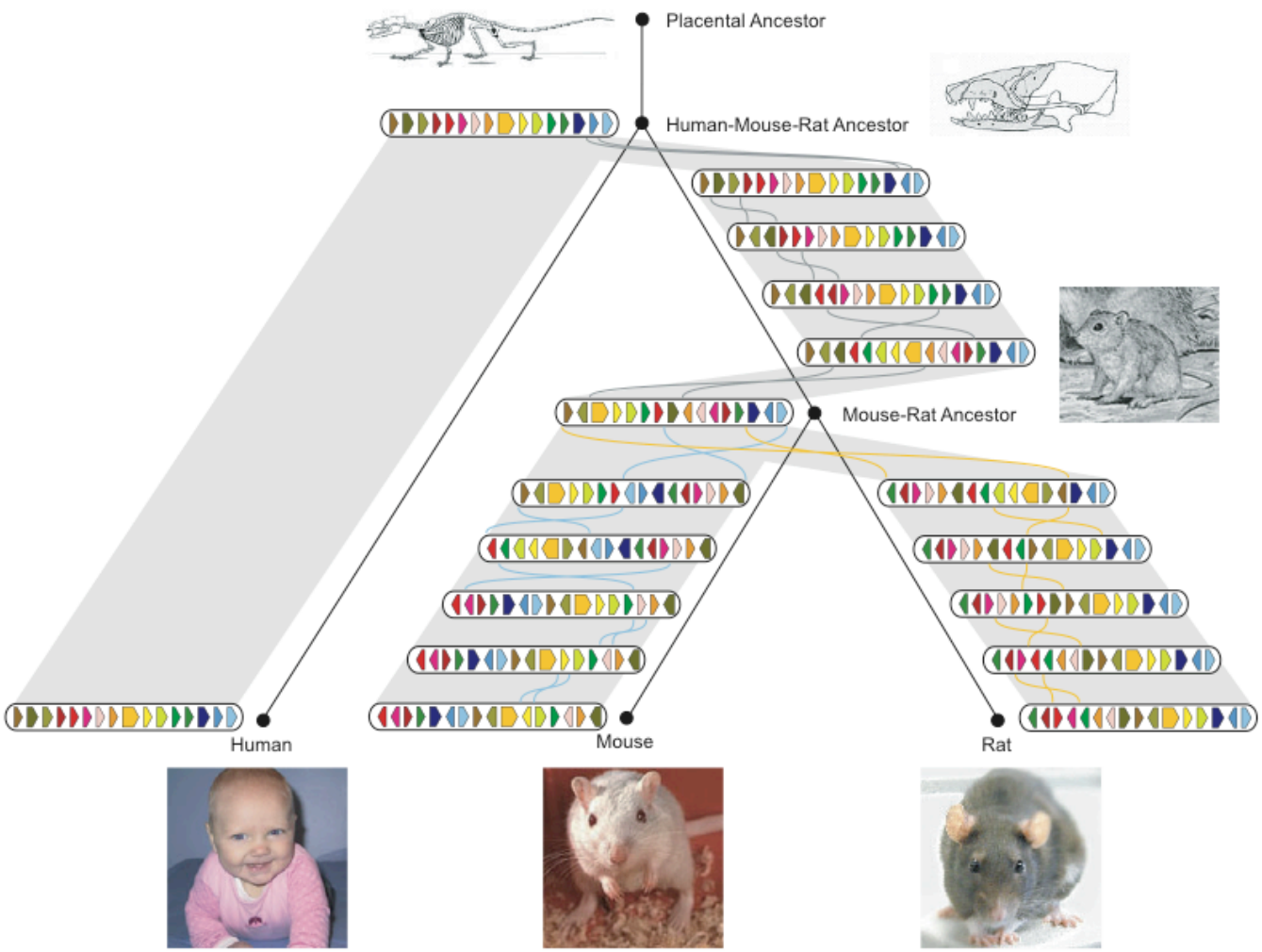


Genome rearrangements



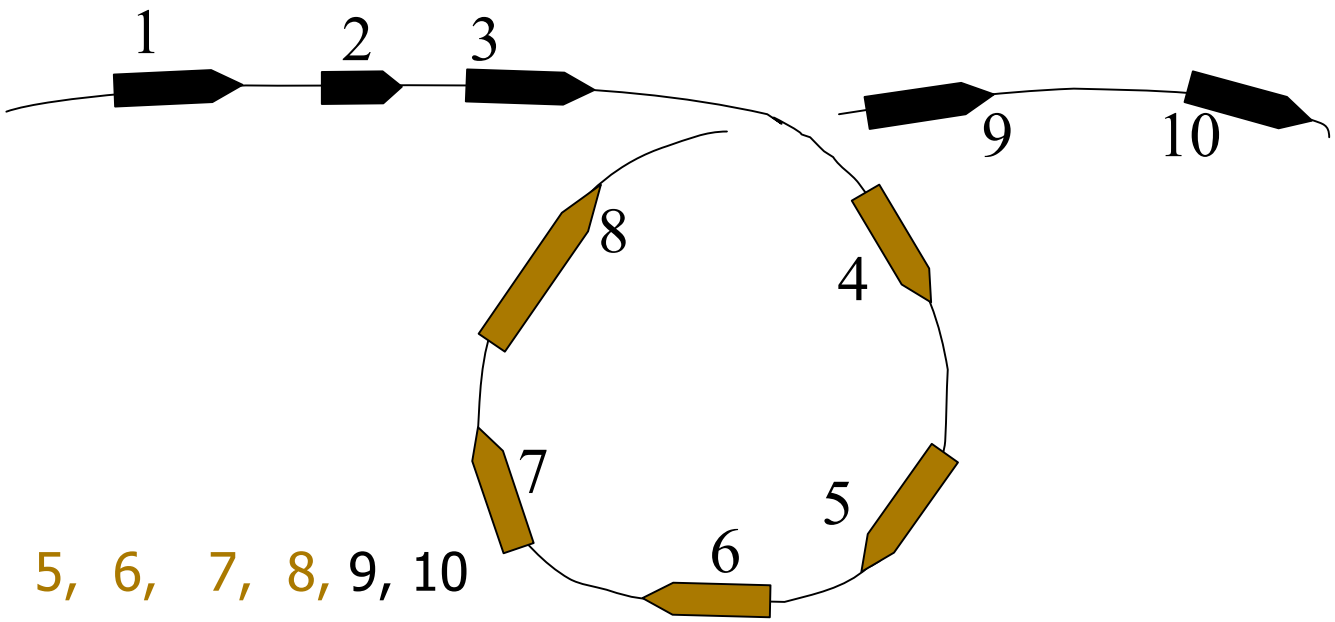
- What are the similarity blocks and how to find them?
- What is the architecture of the ancestral genome?
- What is the evolutionary scenario for transforming one genome into the other?

History of Chromosome X



Rat Consortium, *Nature*, 2004

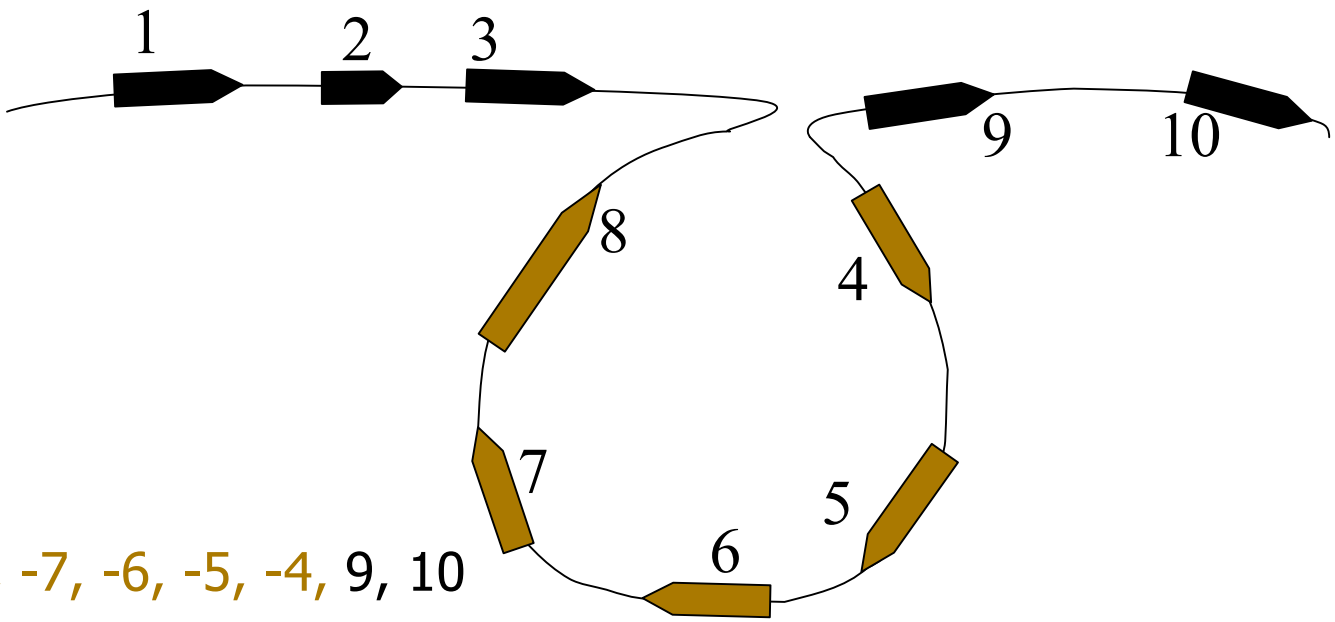
Reversals



1, 2, 3, 4, 5, 6, 7, 8, 9, 10

- Blocks represent conserved genes.

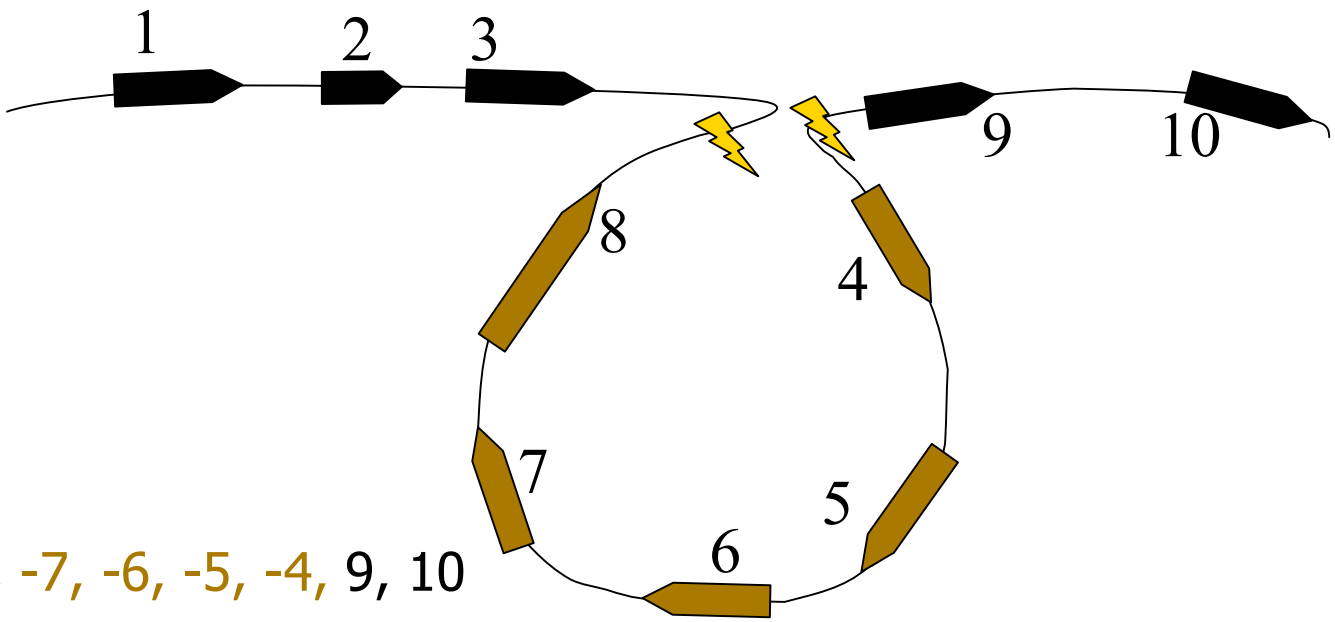
Reversals



1, 2, 3, -8, -7, -6, -5, -4, 9, 10

- Blocks represent conserved genes.
- In the course of evolution or in a clinical context, blocks 1,…,10 could be misread as 1, 2, 3, -8, -7, -6, -5, -4, 9, 10.

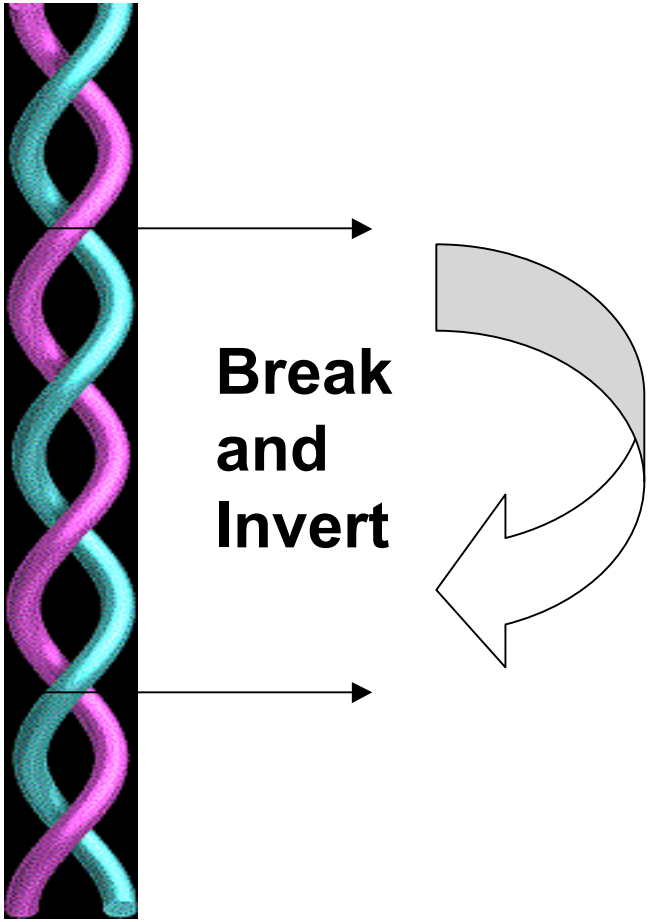
Reversals and Breakpoints



1, 2, 3, -8, -7, -6, -5, -4, 9, 10

The reversion introduced two *breakpoints* (disruptions in order).

Reversals: Example



Break and Invert

5’ ATGCCTGTACTA 3’

3’ TACGGACATGAT 5’

5’ ATGTACAGGCTA 3’

3’ TACATGTCCGAT 5’

Types of Rearrangements

Reversal

1 2 3 4 5 6 → 1 2 -5 -4 -3 6

Translocation

1 2 3
4 5 6 → 1 2 6
4 5 3

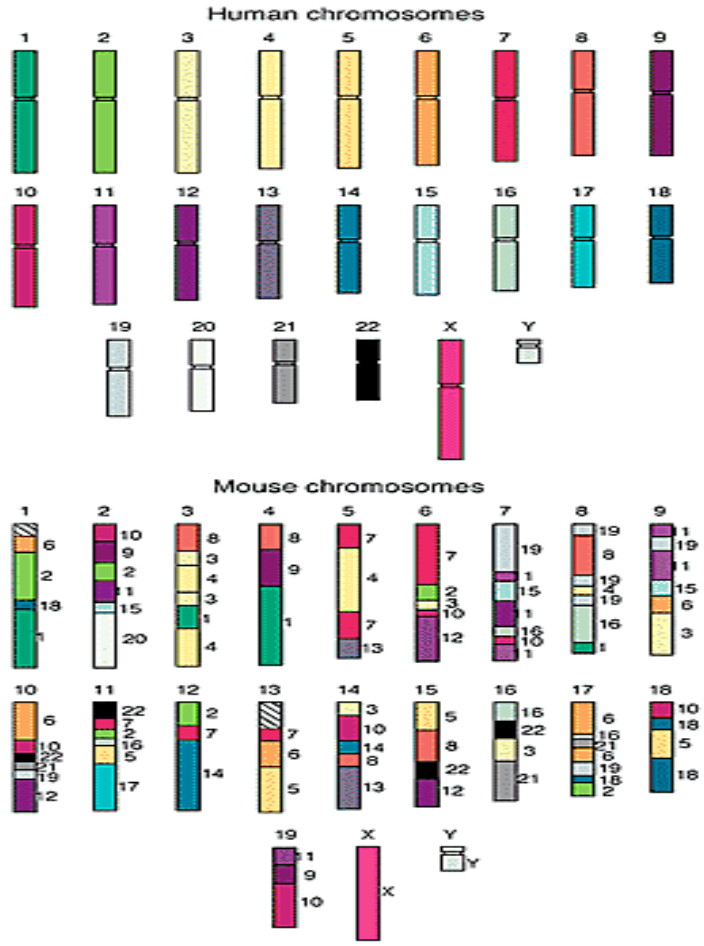
Fusion

1 2 3 4
5 6 ⇄ 1 2 3 4 5 6

Fission

Comparative Genomic Architectures: Mouse vs Human Genome

- Humans and mice have similar genomes, but their genes are ordered differently
- ~245 rearrangements
  - Reversals
  - Fusions
  - Fissions
  - Translocation



Waardenburg's Syndrome: Mouse Provides Insight into Human Genetic Disorder

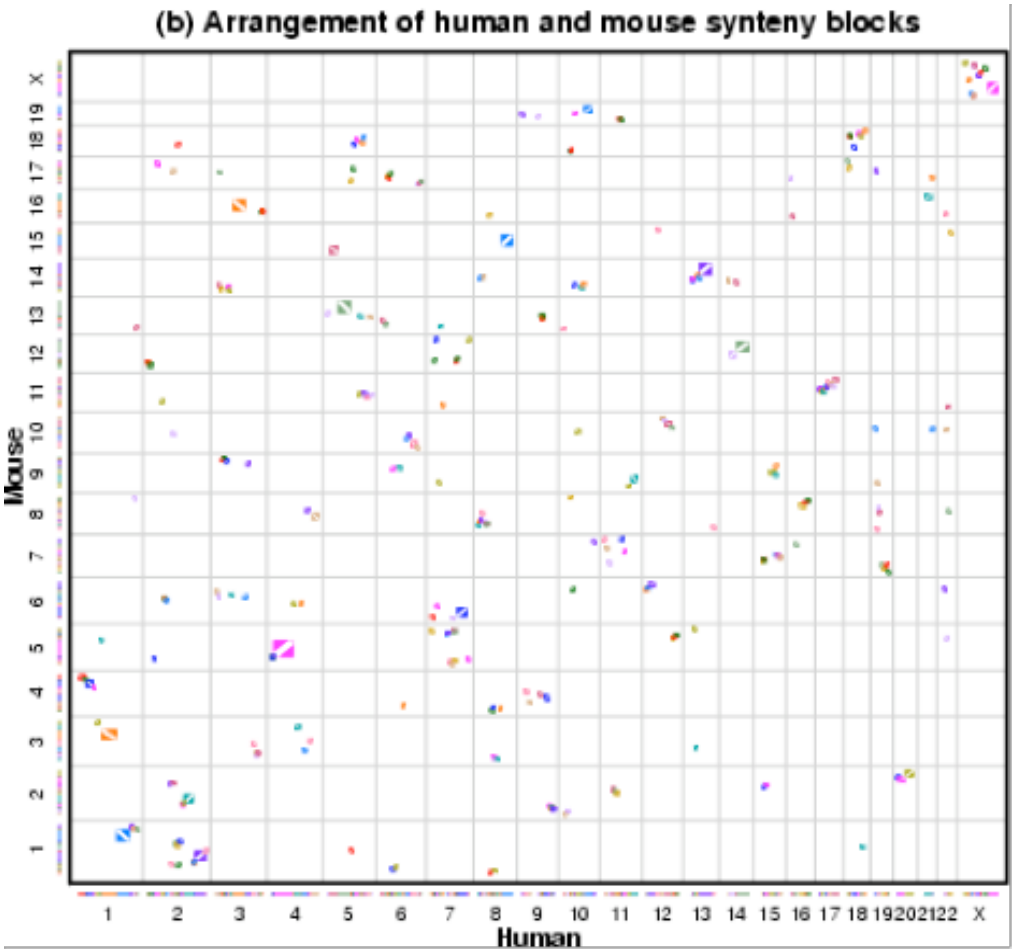
- Waardenburg's syndrome is characterized by pigmentary dysphasia
- Gene implicated in the disease was linked to human chromosome 2 but it was not clear where exactly it is located on chromosome 2

Waardenburg's syndrome and splotch mice

- A breed of mice (with splotch gene) had similar symptoms caused by the same type of gene as in humans
- Scientists succeeded in identifying location of gene responsible for disorder in mice
- Finding the gene in mice gives clues to where the same gene is located in humans

Comparative Genomic Architecture of Human and Mouse Genomes

To locate where corresponding gene is in humans, we have to analyze the relative architecture of human and mouse genomes



Reversals: Example

π = 1 2 3 4 5 6 7 8

$\rho(3,5)$ ↓

1 2 5 4 3 6 7 8

Reversals: Example

π = 1 2 3 4 5 6 7 8

$\rho(3,5)$ ↓

1 2 5 4 3 6 7 8

$\rho(5,6)$ ↓

1 2 5 4 6 3 7 8

Reversals and Gene Orders

- Gene order is represented by a permutation π:

$$\pi = \pi_1 \text{ ------ } \pi_{i-1}\ \underline{\pi_i\ \pi_{i+1} \text{ ------ } \pi_{j-1}\ \pi_j}\ \pi_{j+1} \text{ ----- } \pi_n$$

$$\Big\downarrow \rho(i,j)$$

$$\pi_1 \text{ ------ } \pi_{i-1}\ \pi_j\ \pi_{j-1} \text{ ------ } \pi_{i+1}\ \pi_i\ \pi_{j+1} \text{ ----- } \pi_n$$

- Reversal $\rho(i, j)$ reverses (flips) the elements from i to j in π

Reversal Distance Problem

- Goal: Given two permutations, find the shortest series of reversals that transforms one into another

- Input: Permutations $\boldsymbol{\pi}$ and $\boldsymbol{\sigma}$

- Output: A series of reversals $\rho_1,\ldots\rho_t$ transforming $\boldsymbol{\pi}$ into $\boldsymbol{\sigma}$, such that t is minimum

- ***t*** - reversal distance between π and σ
- ***d***(π, σ) - smallest possible value of t, given π and σ

Sorting By Reversals Problem

- <u>Goal</u>: Given a permutation, find a shortest series of reversals that transforms it into the identity permutation ($1\ 2\ \ldots\ n$)

- <u>Input</u>: Permutation π

- <u>Output</u>: A series of reversals $\rho_1, \ldots\ \rho_t$ transforming π into the identity permutation such that t is minimum

Sorting By Reversals: Example

- $t = d(\pi)$ - reversal distance of π
- Example :

$$\pi = \underline{3\ \ 4}\ \ 2\ \ 1\ \ 5\ \ 6\ \ 7\ \ 10\ \ 9\ \ 8$$

$$4\ \ 3\ \ 2\ \ 1\ \ 5\ \ 6\ \ 7\ \ \underline{10\ \ 9\ \ 8}$$

$$\underline{4\ \ 3\ \ 2\ \ 1}\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10$$

$$1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10$$

So $d(\pi) = 3$

Sorting by reversals: 5 steps

Step 0:	π	2	-4	-3	5	-8	-7	-6	1
Step 1:		2	3	4	5	-8	-7	-6	1
Step 2:		2	3	4	5	6	7	8	1
Step 3:		2	3	4	5	6	7	8	-1
Step 4:		-8	-7	-6	-5	-4	-3	-2	-1
Step 5:	γ	1	2	3	4	5	6	7	8

Sorting by reversals: 4 steps

Step 0:	π	2	-4	-3	5	-8	-7	-6	1
Step 1:		2	3	4	5	-8	-7	-6	1
Step 2:		-5	-4	-3	-2	-8	-7	-6	1
Step 3:		-5	-4	-3	-2	-1	6	7	8
Step 4:	γ	1	2	3	4	5	6	7	8

Sorting by reversals: 4 steps

Step 0:	π	2	-4	-3	5	-8	-7	-6	1
Step 1:		2	3	4	5	-8	-7	-6	1
Step 2:		-5	-4	-3	-2	-8	-7	-6	1
Step 3:		-5	-4	-3	-2	-1	6	7	8
Step 4:	γ	1	2	3	4	5	6	7	8

What is the reversal distance for this permutation? Can it be sorted in 3 steps?

Pancake Flipping Problem

- The chef is sloppy; he prepares an unordered stack of pancakes of different sizes
- The waiter wants to rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom)
- He does it by flipping over several from the top, repeating this as many times as necessary

Christos Papadimitrou and Bill Gates flip pancakes

Pancake Flipping Problem: Formulation

- <u>Goal</u>: Given a stack of *n* pancakes, what is the minimum number of flips to rearrange them into perfect stack?
- <u>Input</u>: Permutation π
- <u>Output</u>: A series of prefix reversals $\rho_1, \ldots \rho_t$ transforming π into the identity permutation such that *t* is minimum

Pancake Flipping Problem: Greedy Algorithm

- Greedy approach: 2 prefix reversals at most to place a pancake in its right position, $2n - 2$ steps total

  at most

- William Gates and Christos Papadimitriou showed in the mid-1970s that this problem can be solved by at most $5/3\ (n + 1)$ *prefix reversals*

Sorting By Reversals: A Greedy Algorithm

- If sorting permutation π = 1 2 3 6 4 5, the first three elements are already in order so it does not make any sense to break them.
- The length of the already sorted prefix of π is denoted *prefix*(π)
  - *prefix*(π) = 3
- This results in an idea for a greedy algorithm: increase *prefix*(π) at every step

Greedy Algorithm: An Example

- Doing so, π can be sorted

$$1\ 2\ 3\ \underline{6\ 4}\ 5$$

$$\downarrow$$

$$1\ 2\ 3\ 4\ \underline{6\ 5}$$

$$\downarrow$$

$$1\ 2\ 3\ 4\ 5\ 6$$

- Number of steps to sort permutation of length *n* is at most *(n – 1)*

Greedy Algorithm: Pseudocode

<u>SimpleReversalSort(π)</u>

```
for i ← 1 to n – 1
    j ← position of element i in π (i.e., π_j = i)
    if j ≠ i
        π ← π * ρ(i, j)
        output π
    if π is the identity permutation
        return
```

Analyzing SimpleReversalSort

- SimpleReversalSort does not guarantee the smallest number of reversals and takes five steps on π = 6 1 2 3 4 5 :
  - Step 1: 1 6 2 3 4 5
  - Step 2: 1 2 6 3 4 5
  - Step 3: 1 2 3 6 4 5
  - Step 4: 1 2 3 4 6 5
  - Step 5: 1 2 3 4 5 6

Analyzing SimpleReversalSort (cont'd)

- But it can be sorted in two steps:

  π = 6 1 2 3 4 5

  - Step 1: 5 4 3 2 1 6
  - Step 2: 1 2 3 4 5 6
- So, SimpleReversalSort(π) is not optimal

- Optimal algorithms are unknown for many problems; approximation algorithms are used

Approximation Algorithms

- These algorithms find approximate solutions rather than optimal solutions
- The approximation ratio of an algorithm A on input π is:

$$A(\pi) / OPT(\pi)$$

where

$A(\pi)$ -solution produced by algorithm A
$OPT(\pi)$ - optimal solution of the problem

Approximation Ratio/Performance Guarantee

- Approximation ratio (performance guarantee) of algorithm A: max approximation ratio of all inputs of size *n*
  - For algorithm A that minimizes objective function (minimization algorithm):
    - $\max_{|\pi| = n} A(\pi) / OPT(\pi)$

Approximation Ratio/Performance Guarantee

- Approximation ratio (performance guarantee) of algorithm A: max approximation ratio of all inputs of size *n*
  - For algorithm A that minimizes objective function (minimization algorithm):
    - $\max_{|\pi| = n} A(\pi) / OPT(\pi)$
  - For maximization algorithm:
    - $\min_{|\pi| = n} A(\pi) / OPT(\pi)$

Adjacencies and Breakpoints

$$\pi = \pi_1\pi_2\pi_3\ldots\pi_{n-1}\pi_n$$

- A pair of elements π_i and π_{i+1} are adjacent if

$$\pi_{i+1} = \pi_i \pm 1$$

- For example:

$$\pi = 1\ \ 9\ \ \underline{3\ \ 4}\ \ \underline{7\ \ 8}\ \ 2\ \ \underline{6\ \ 5}$$

- (3, 4) or (7, 8) and (6,5) are adjacent pairs

Breakpoints: An Example

There is a breakpoint between any adjacent element that are non-consecutive:

$$\pi = 1 \mid 9 \mid 3 \;\; 4 \mid 7 \;\; 8 \mid 2 \mid 6 \;\; 5$$

- Pairs (1,9), (9,3), (4,7), (8,2) and (2,5) form breakpoints of permutation π
- $b(\pi)$ - # breakpoints in permutation π

Adjacency & Breakpoints

- An adjacency - a pair of adjacent elements that are consecutive
- A breakpoint - a pair of adjacent elements that are not consecutive

---

$\pi = 5 \;\; 6 \;\; 2 \;\; 1 \;\; 3 \;\; 4 \longrightarrow$ Extend π with $\pi_0 = 0$ and $\pi_7 = 7$

adjacencies

0 5 6 2 1 3 4 7

breakpoints

Extending Permutations

- We put two elements $\pi_0 = 0$ and $\pi_{n+1} = n+1$ at the ends of π

  Example:

$$\pi = 1 \,|\, 9 \,|\, 3 \;\; 4 \,|\, 7 \;\; 8 \,|\, 2 \,|\, 6 \;\; 5$$

Extending with *0* and *10*

$$\pi = 0 \;\; 1 \,|\, 9 \,|\, 3 \;\; 4 \,|\, 7 \;\; 8 \,|\, 2 \,|\, 6 \;\; 5 \,|\, 10$$

Note: A new breakpoint was created after extending

Reversal Distance and Breakpoints

- Each reversal eliminates at most 2 breakpoints.

π = 2 3 1 4 6 5

0 | 2 3 | 1 | 4 | 6 5 | 7 $\qquad b(\pi) = 5$

0 1 | 3 2 | 4 | 6 5 | 7 $\qquad b(\pi) = 4$

0 1 2 3 4 | 6 5 | 7 $\qquad b(\pi) = 2$

0 1 2 3 4 5 6 7 $\qquad b(\pi) = 0$

Reversal Distance and Breakpoints

- Each reversal eliminates at most 2 breakpoints.
- This implies:

  reversal distance ≥ #breakpoints / 2

π = 2 3 1 4 6 5

0 | 2 3 | 1 | 4 | 6 5 | 7	$b(\pi) = 5$
0 1 | 3 2 | 4 | 6 5 | 7	$b(\pi) = 4$
0 1 2 3 4 | 6 5 | 7	$b(\pi) = 2$
0 1 2 3 4 5 6 7	$b(\pi) = 0$

Sorting By Reversals: A Better Greedy Algorithm

<u>BreakPointReversalSort(π)</u>

1 **while** $b(\pi) > 0$

2 Among all possible reversals, choose reversal ρ minimizing $b(\pi \bullet \rho)$

3 $\pi \leftarrow \pi \bullet \rho(i, j)$

4 **output** π

5 **return**

Sorting By Reversals: A Better Greedy Algorithm

<u>BreakPointReversalSort(π)</u>

1 **while** $b(\pi) > 0$

2 Among all possible reversals, choose reversal ρ minimizing $b(\pi \bullet \rho)$

3 $\pi \leftarrow \pi \bullet \rho(i, j)$

4 **output** π

5 **return**

Problem: this algorithm may work forever

Strips

- Strip: an interval between two consecutive breakpoints in a permutation
  - Decreasing strip: *strip* of elements in decreasing order (e.g. 6 5 and 3 2).
  - Increasing strip: *strip* of elements in increasing order (e.g. 7 8)

0 1 9 4 3 7 8 2 5 6 *10*

- A single-element strip can be declared either increasing or decreasing. We will choose to declare them as decreasing with exception of the strips with *0* and *n+1*

Reducing the Number of Breakpoints

Theorem 1:

If permutation π contains at least one decreasing strip, then there exists a reversal ρ which decreases the number of breakpoints (i.e. $b(\pi \bullet \rho) < b(\pi)$)

Things To Consider

- For π = 1 4 6 5 7 8 3 2

  0 1 | 4 | 6 5 | 7 8 | 3 2 | *9* $b(\pi) = 5$

  - Choose decreasing strip with the smallest element k in π (k = 2 in this case)

Things To Consider (cont'd)

- For π = 1 4 6 5 7 8 3 2

  0 1 | 4 | 6 5 | 7 8 | 3 2 | *9* $b(\pi) = 5$

  - Choose decreasing strip with the smallest element k in π (k = 2 in this case)

Things To Consider (cont'd)

- For π = 1 4 6 5 7 8 3 2

  0 1 | 4 | 6 5 | 7 8 | 3 2 | *9* $b(\pi) = 5$

  - Choose decreasing strip with the smallest element *k* in π (*k* = 2 in this case)
  - Find *k – 1* in the permutation

Things To Consider (cont'd)

- For π = 1 4 6 5 7 8 3 2

  0 1 | 4 | 6 5 | 7 8 | 3 2 | *9* $\quad b(\pi) = 5$

  - Choose decreasing strip with the smallest element *k* in π (*k* = 2 in this case)
  - Find *k – 1* in the permutation
  - Reverse the segment between *k* and *k-1*:
  - *0* 1 | 4 | 6 5 | 7 8 | 3 2 | *9* $\quad b(\pi) = 5$
  - *0* 1 2 3 | 8 7 | 5 6 | 4 | *9* $\quad b(\pi) = 4$

Reducing the Number of Breakpoints Again

- If there is no decreasing strip, there may be no reversal ρ that reduces the number of breakpoints (i.e. $b(\pi \bullet \rho) \geq b(\pi)$ for any reversal ρ).
- By reversing an increasing strip (# of breakpoints stay unchanged), we will create a decreasing strip at the next step. Then the number of breakpoints will be reduced in the next step (theorem 1).

Things To Consider (cont'd)

- There are no decreasing strips in π, for:

$$\pi = 0\ \ 1\ \ 2 \mid 5\ \ 6\ \ 7 \mid 3\ \ 4 \mid 8 \qquad b(\pi) = 3$$

$$\pi \bullet \rho(6,7) = 0\ \ 1\ \ 2 \mid 5\ \ 6\ \ 7 \mid 4\ \ 3 \mid 8 \qquad b(\pi) = 3$$

- ✓ $\rho(6,7)$ does not change the # of breakpoints
- ✓ $\rho(6,7)$ creates a decreasing strip thus guaranteeing that the next step will decrease the # of breakpoints.

ImprovedBreakpointReversalSort

<u>ImprovedBreakpointReversalSort(π)</u>

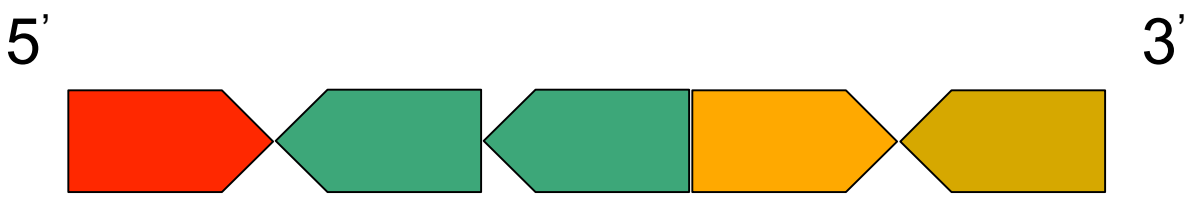
```
while b(π) > 0
    if π has a decreasing strip
        Among all possible reversals, choose reversal ρ
                                that   minimizes b(π • ρ)
    else
        Choose a reversal ρ that flips an increasing strip in π
  π ← π • ρ
    output π
return
```

ImprovedBreakpointReversalSort: Performance Guarantee

- *ImprovedBreakPointReversalSort* is an approximation algorithm with a performance guarantee of at most 4
  - It eliminates at least one breakpoint in every two steps; at most $2b(\pi)$ steps
  - Approximation ratio: $2b(\pi) \ / \ d(\pi)$
  - Optimal algorithm eliminates at most 2 breakpoints in every step: $d(\pi) \geq b(\pi) / 2$
  - Performance guarantee:
    - $(2b(\pi) / d(\pi)) \geq [2b(\pi) / (b(\pi) / 2)] = 4$

Signed Permutations

- Up to this point, all permutations to sort were unsigned
- But genes have directions… so we should consider signed permutations

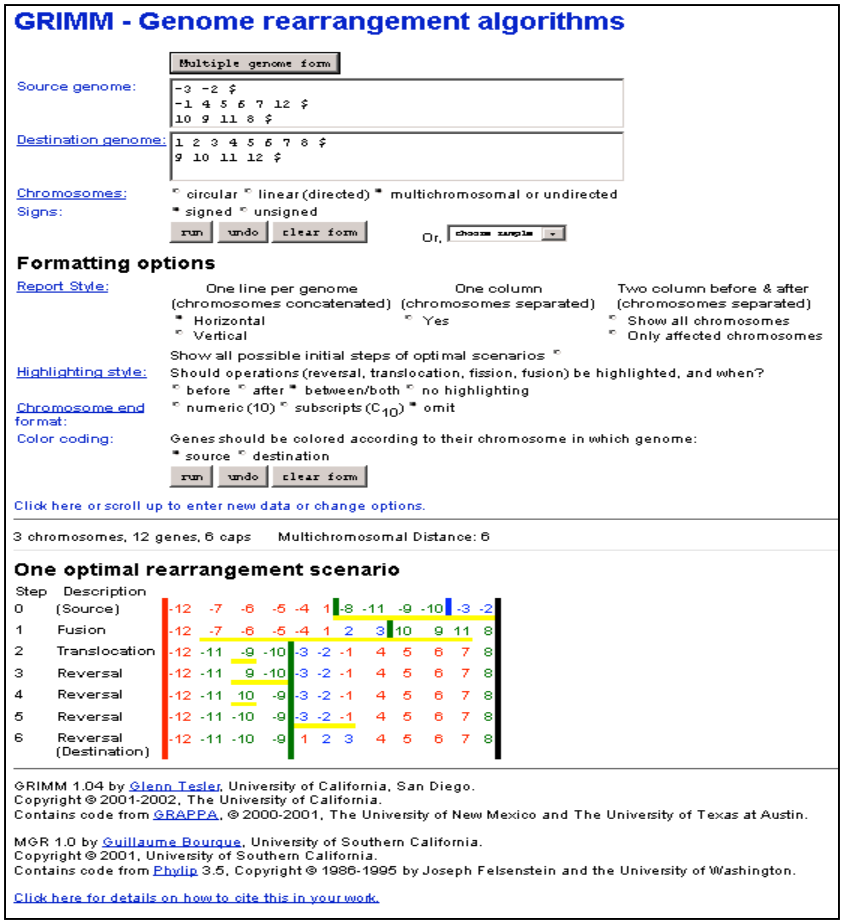


π = 1 -2 -3 4 -5

GRIMM Web Server

- Real genome architectures are represented by signed permutations
- Efficient algorithms to sort signed permutations have been developed
- GRIMM web server computes the reversal distances between signed permutations:

GRIMM Web Server



GRIMM - Genome rearrangement algorithms

Multiple genome form

Source genome: -3 -2 $
-1 4 5 6 7 12 $
10 9 11 8 $

Destination genome: 1 2 3 4 5 6 7 8 $
9 10 11 12 $

Chromosomes: circular linear (directed) multichromosomal or undirected
Signs: signed unsigned

run | undo | clear form Or, choose sample

Formatting options

Report Style: One line per genome (chromosomes concatenated): Horizontal, Vertical; One column (chromosomes separated): Yes; Two column before & after (chromosomes separated): Show all chromosomes, Only affected chromosomes

Show all possible initial steps of optimal scenarios

Highlighting style: Should operations (reversal, translocation, fission, fusion) be highlighted, and when? before after between/both no highlighting

Chromosome end format: numeric (10) subscripts (C_{10}) omit

Color coding: Genes should be colored according to their chromosome in which genome: source destination

run | undo | clear form

Click here or scroll up to enter new data or change options.

3 chromosomes, 12 genes, 6 caps Multichromosomal Distance: 6

One optimal rearrangement scenario

Step	Description	
0	(Source)	-12 -7 -6 -5 -4 1 -8 -11 -9 -10 -3 -2
1	Fusion	-12 -7 -6 -5 -4 1 2 3 10 9 11 8
2	Translocation	-12 -11 -9 -10 -3 -2 -1 4 5 6 7 8
3	Reversal	-12 -11 9 -10 -3 -2 -1 4 5 6 7 8
4	Reversal	-12 -11 10 -9 -3 -2 -1 4 5 6 7 8
5	Reversal	-12 -11 -10 -9 -3 -2 -1 4 5 6 7 8
6	Reversal (Destination)	-12 -11 -10 -9 1 2 3 4 5 6 7 8

GRIMM 1.04 by Glenn Tesler, University of California, San Diego.
Copyright © 2001-2002, The University of California.
Contains code from GRAPPA, © 2000-2001, The University of New Mexico and The University of Texas at Austin.

MGR 1.0 by Guillaume Bourque, University of Southern California.
Copyright © 2001, University of Southern California.
Contains code from Phylip 3.5, Copyright © 1986-1995 by Joseph Felsenstein and the University of Washington.

Click here for details on how to cite this in your work.

http://www-cse.ucsd.edu/groups/bioinformatics/GRIMM

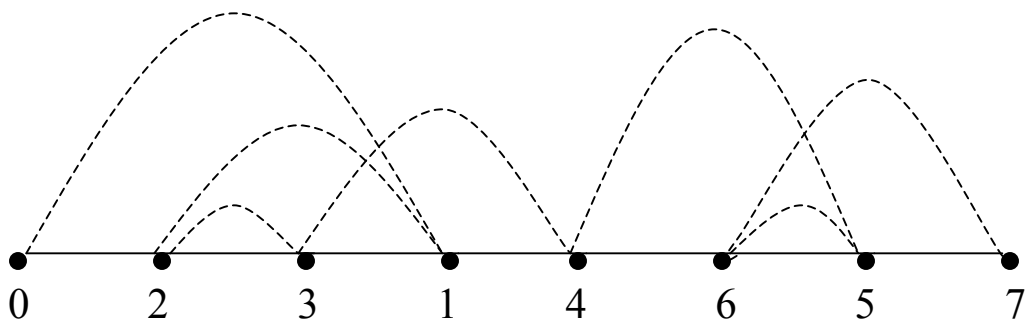
Breakpoint Graph

1) Represent the elements of the permutation $\pi = 2\ 3\ 1\ 4\ 6\ 5$ as vertices in a graph (ordered along a line)

2) Connect vertices in order given by π with black edges (black path)

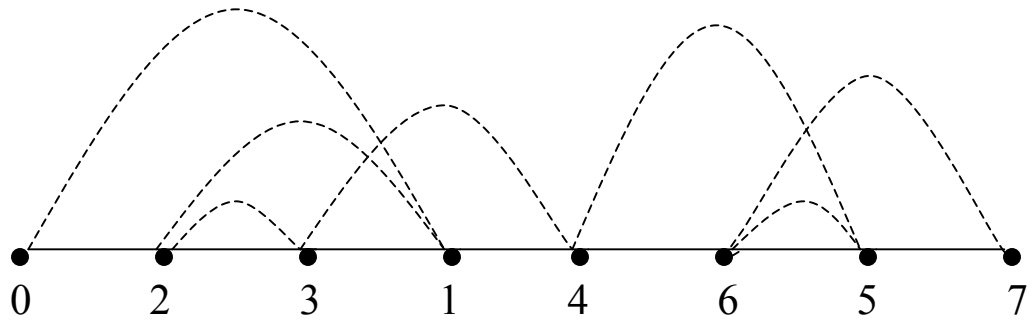
3) Connect vertices in order given by 1 2 3 4 5 6 with grey edges (grey path)

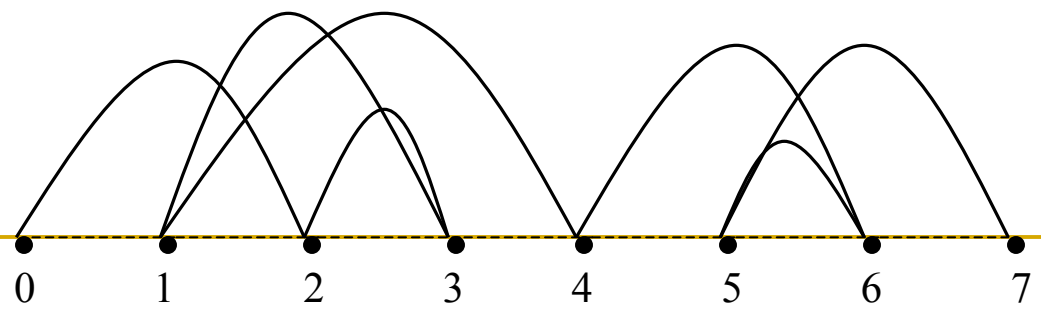
4) Superimpose black and grey paths



Two Equivalent Representations of the Breakpoint Graph

- Consider the following Breakpoint Graph
- If we line up the gray path (instead of black path) on a horizontal line, then we would get the following graph
- Although they may look different, these two graphs are the same



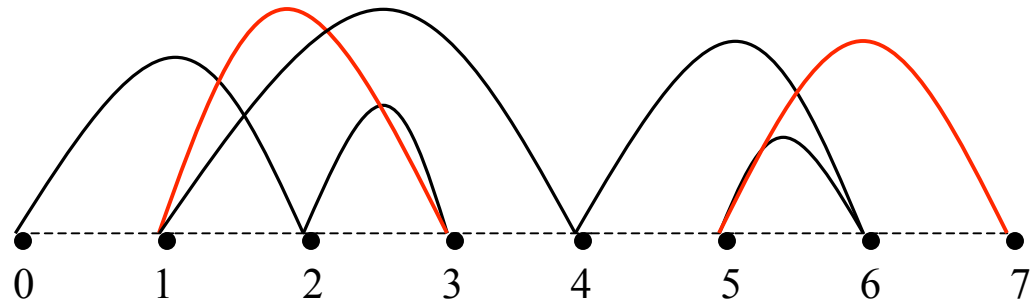


What is the Effect of the Reversal ?

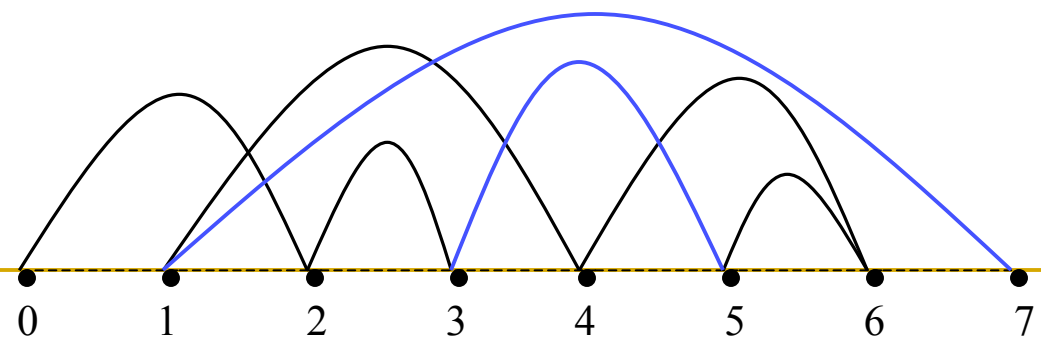
How does a reversal change the breakpoint graph?

- The gray paths stayed the same for both graphs
- There is a change in the graph at this point
- There is another change at this point
- The black edges are unaffected by the reversal so they remain the same for both graphs

Before: 0 2 3 1 4 6 5 7

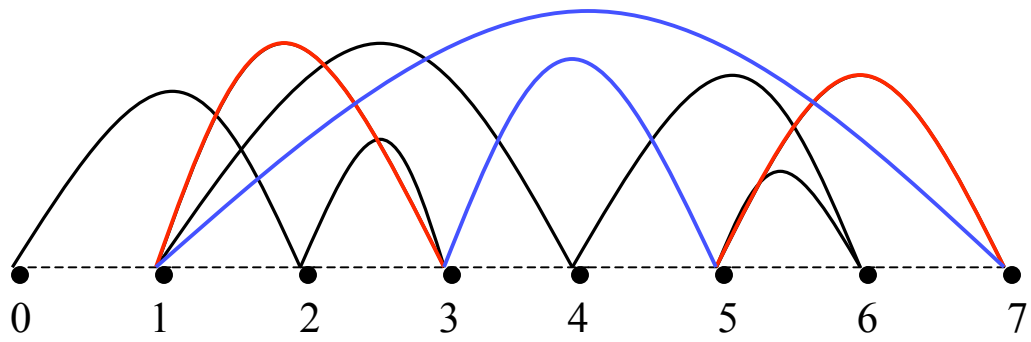


After: 0 2 3 5 6 4 1 7



A reversal affects 4 edges in the breakpoint graph

- A reversal removes 2 edges (red) and replaces them with 2 new edges (blue)



Effects of Reversals

Case 1:

Both edges belong to the same cycle

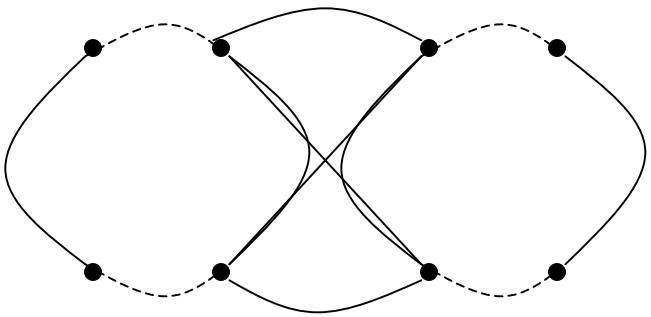
- Remove the center black edges and replace them with new black edges (there are two ways to replace them)
- (a) After this replacement, there now exists 2 cycles instead of 1 cycle
- (b) Or after this replacement, there still exists 1 cycle

Therefore, after the reversal

$$c(\pi\rho) - c(\pi) = 0$$

$c(\pi\rho) - c(\pi) = 0$ or 1

This is called a proper reversal since there's a cycle increase after the reversal.



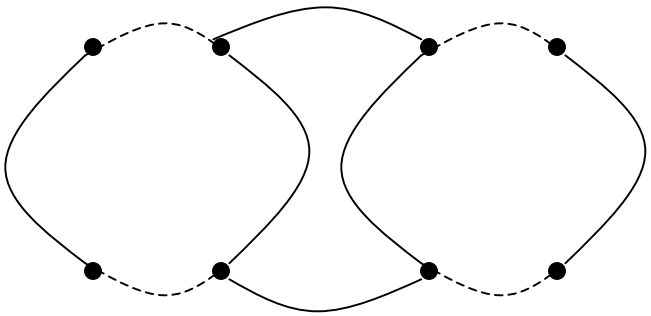
Effects of Reversals (Continued)

<u>Case 2</u>:

Both edges belong to different cycles

- Remove the center black edges and replace them with new black edges
- After the replacement, there now exists 1 cycle instead of 2 cycles

$\Rightarrow$ $c(\pi\rho) - c(\pi) = -1$



Therefore, for every permutation π and reversal ρ, $c(\pi\rho) - c(\pi) \leq 1$

Reversal Distance and Maximum Cycle Decomposition

- Since the identity permutation of size n contains the maximum cycle decomposition of $n+1$, $c(identity) = n+1$

- $c(identity) - c(\pi)$ equals the number of cycles that need to be "added" to $c(\pi)$ while transforming π into the identity

- Based on the previous theorem, at best after each reversal, the cycle decomposition could increased by one, then:

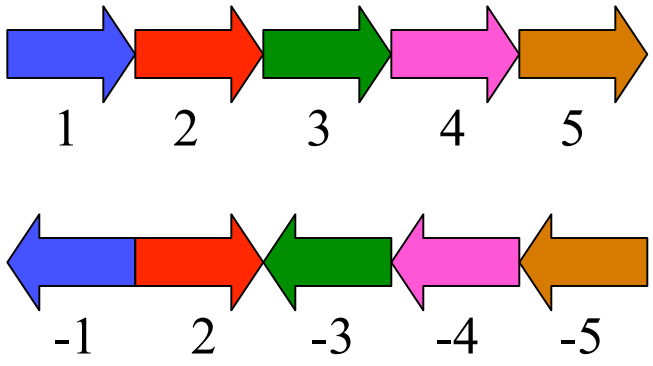
$$d(\pi) = c(identity) - c(\pi) = n+1 - c(\pi)$$

- Yet, not every reversal can increase the cycle decomposition

➡ *Therefore,* $d(\pi) \geq n+1 - c(\pi)$

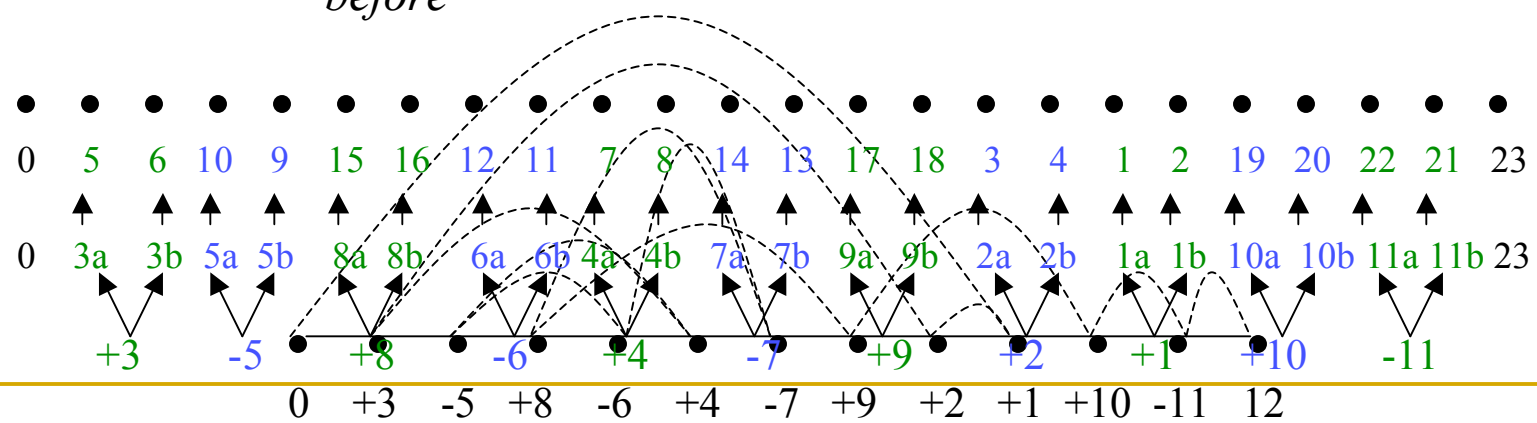
Signed Permutation

- Genes are *directed* fragments of DNA and we represent a genome by a signed permutation
- If genes are in the same position but there orientations are different, they do not have the equivalent gene order
- For example, these two permutations have the same order, but each gene's orientation is the reverse; therefore, they are not equivalent gene sequences



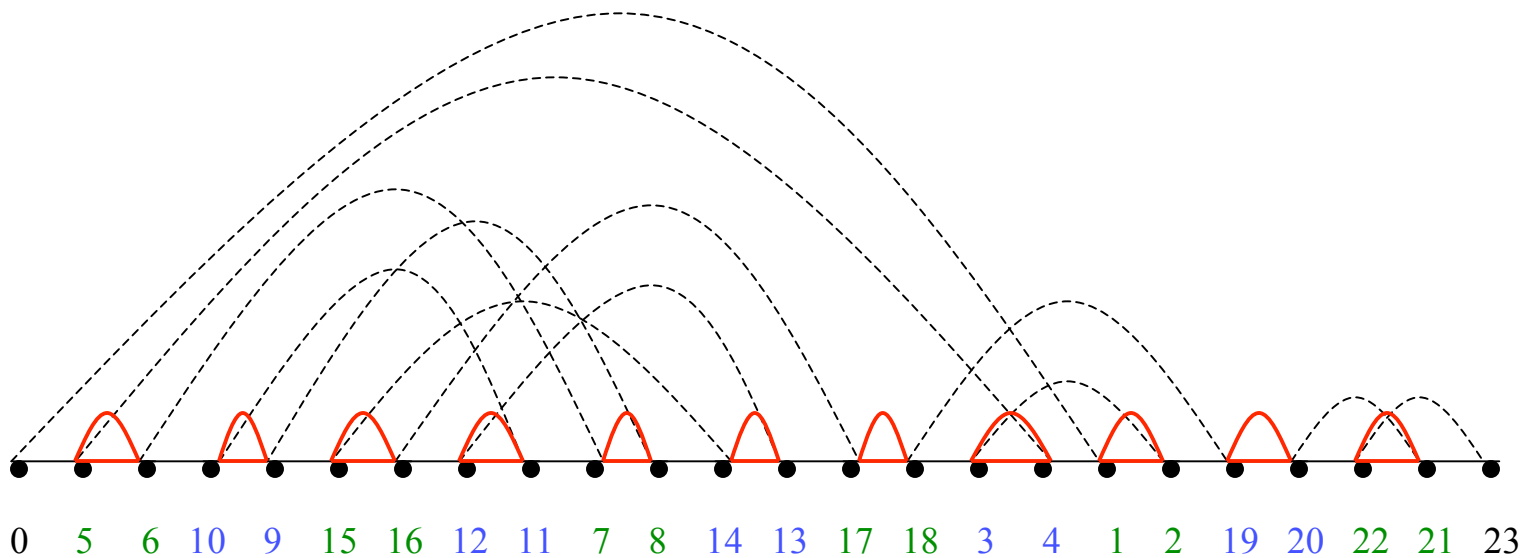
From Signed to Unsigned Permutation

- Begin by constructing a normal signed breakpoint graph
- Redefine each vertex x with the following rules:
  - *If vertex x is positive, replace vertex x with vertex 2x-1 and vertex 2x in that order*
  - *If vertex x is negative, replace vertex x with vertex 2x and vertex 2x-1 in that order*
  - *The extension vertices x = 0 and x = n+1 are kept as it was before*



From Signed to Unsigned Permutation (Continued)

- Construct the breakpoint graph as usual
- Notice the alternating cycles in the graph between every other vertex pair
- Since these cycles came from the same signed vertex, we will not be performing any reversal on both pairs at the same time; therefore, these cycles can be removed from the graph

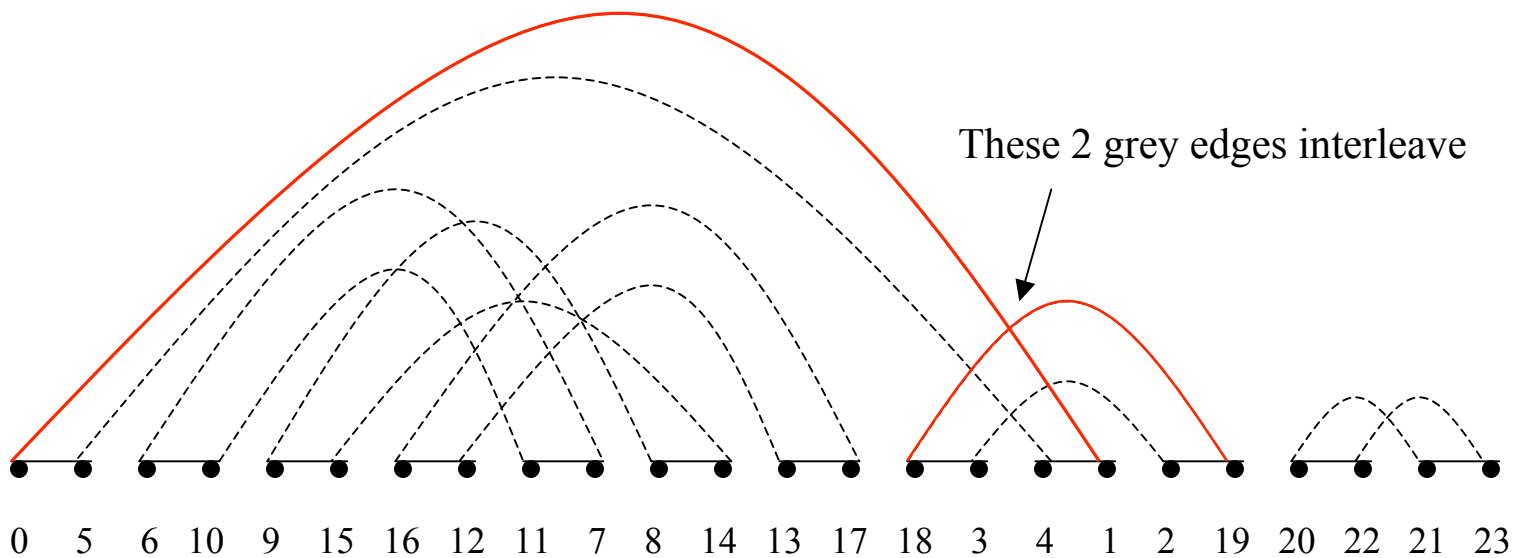


Interleaving Edges

- Interleaving edges are grey edges that cross each other

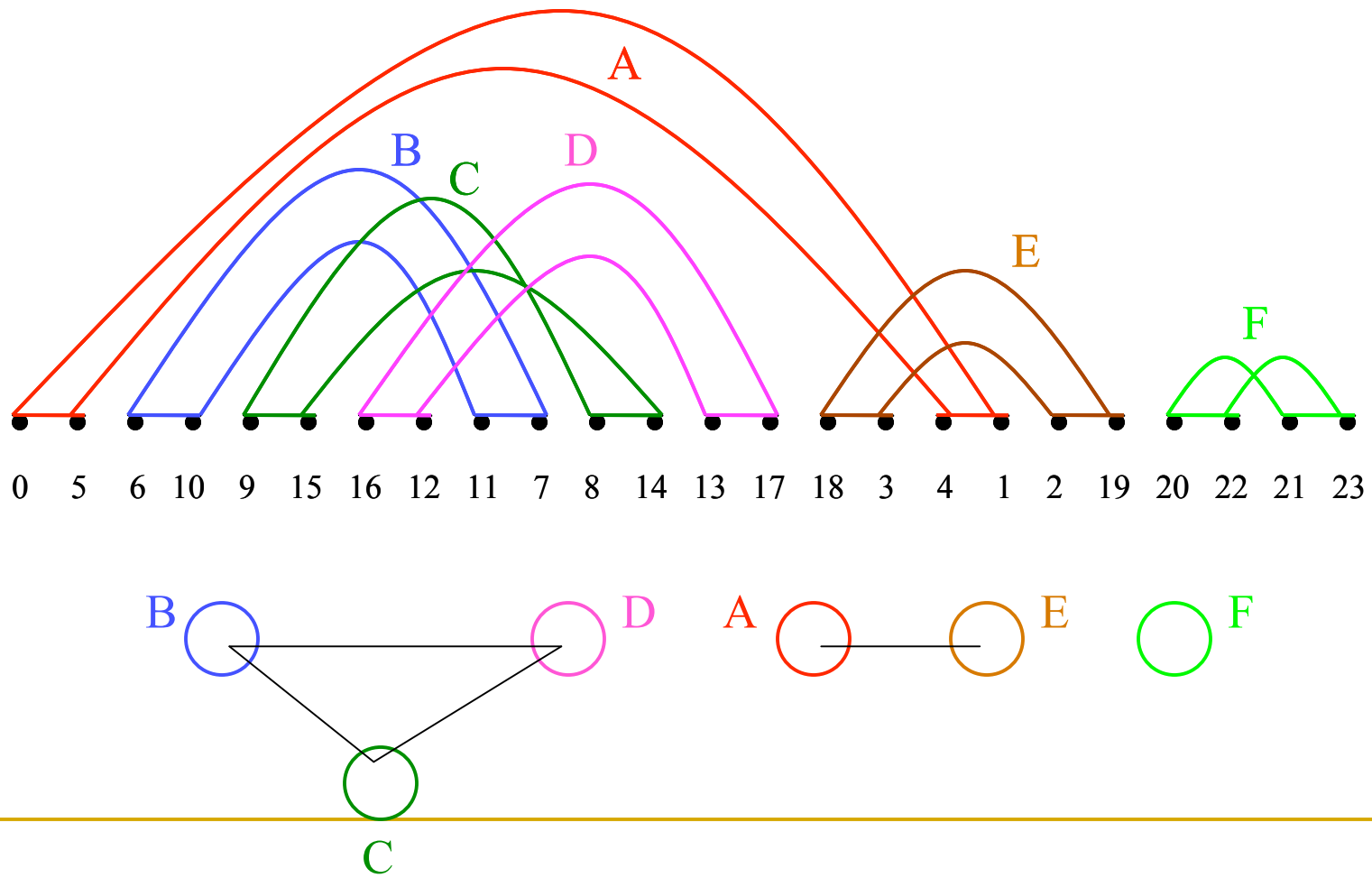
  Example: Edges (0,1) and (18, 19) are interleaving

- Cycles are interleaving if they have an interleaving edge



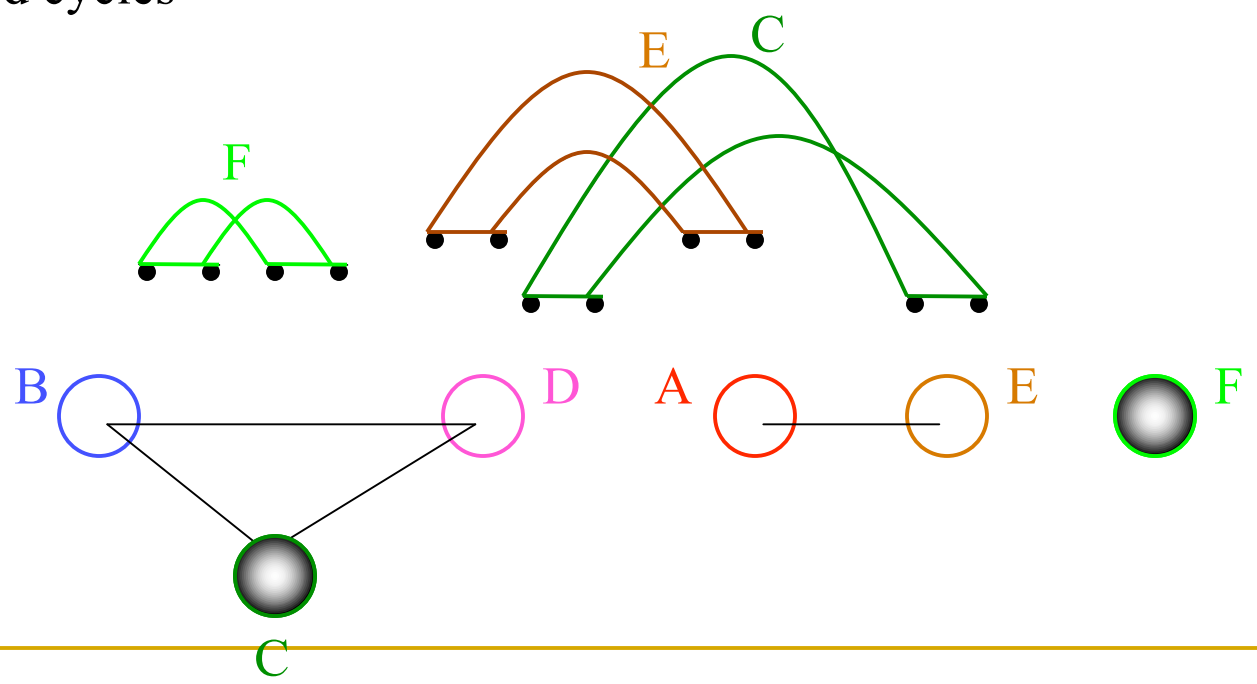
Interleaving Graphs

- An Interleaving Graph is defined on the set of cycles in the Breakpoint graph and are connected by edges where cycles are interleaved



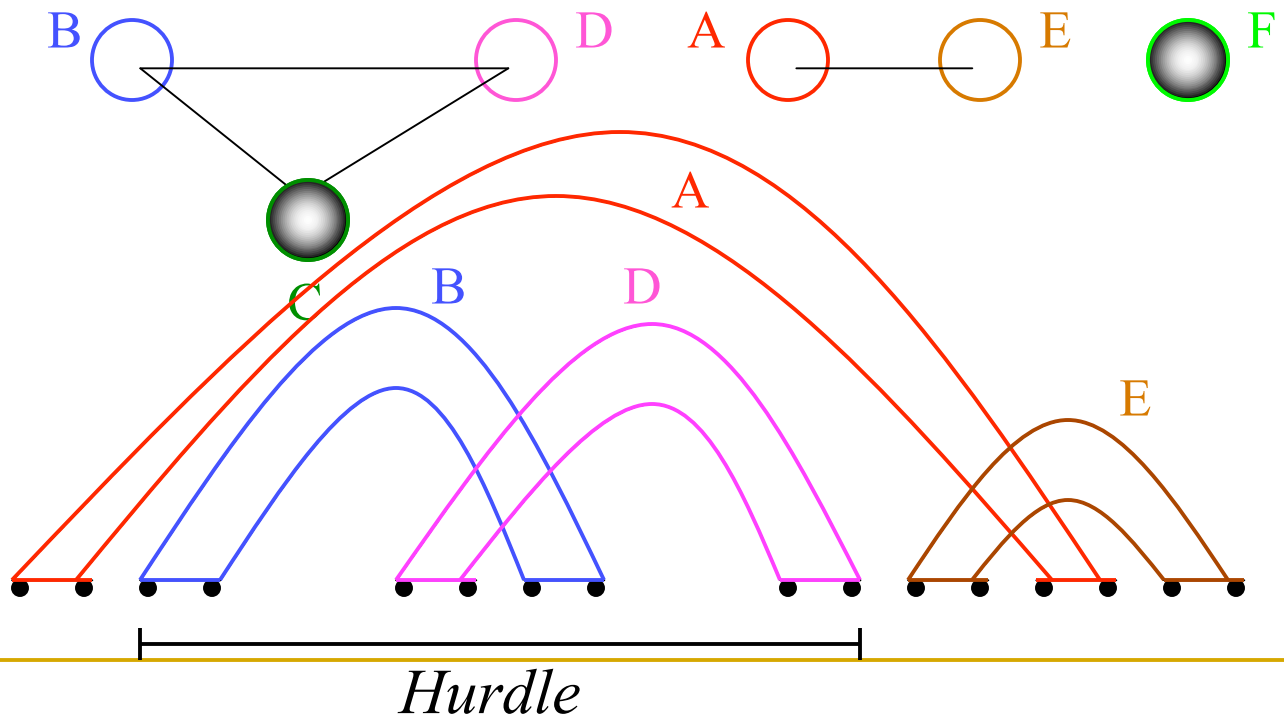
Interleaving Graphs (Continued)

- Oriented cycles are cycles that have the following form
- Mark them on the interleave graph
- Unoriented cycles are cycles that have the following form
- In our example, A, B, D, E are unoriented cycles while C, F are oriented cycles



Hurdles

- Remove the oriented components from the interleaving graph
- The following is the breakpoint graph with these oriented components removed
- Hurdles are connected components that do not contain any other connected components within it



Reversal Distance with Hurdles

- Hurdles are obstacles in the genome rearrangement problem
- They cause a higher number of required reversals for a permutation to transform into the identity permutation
- Let $h(\pi)$ be the number of hurdles in permutation π
- Taking into account of hurdles, the following formula gives a tighter bound on reversal distance:

$$\Rightarrow \quad d(\pi) \geq n+1 - c(\pi) + h(\pi)$$

Integer / linear programming

“Programming”

- A class of mathematical techniques (e.g. dynamic programming) to solve the general optimization problems as

$$Max\ f(x), \qquad x \in S \subseteq R^n$$

R^n*:* the set of *n*-dimensional vectors of real numbers

f(x): objective function, a real-valued function defined on *S*

S the *constraint set*.

- By choosing f and S appropriately, we can model a wide variety of real-life problems in this way.
- Hard problems: $|S| \sim O(k^n)$, $k > 1$

Feasibility and optimality

- Any $x \in S$ is called a *feasible solution*
- If the there is an $x_0 \in S$ such that $f(x) \leq f(x_0)$ for all $x \in S$ then x_0 is called an *optimal solution*

- The aim is to find an optimal solution for a given f and S
- S can be defined by *constraints*

Example

MAX: $350X_1 + 300X_2$ → f

S.T.: $X_1 + X_2 \leq 200$

$9X_1 + 6X_2 \leq 1566$ → constraints

$12X_1 + 16X_2 \leq 2880$

$X_1, X_2 \geq 0$

X_1, X_2 must be integers → integer programming

Integer (linear) programming

- *f* & *S* are restricted by linear form (functions)
  - Linear programming
- *S* is restricted to have only integer values
  - Integer programming (IP), often referred to as integer linear programming (ILP)
- mixed integer programming problem: some elements of *S* are restricted to integers
- ILP is often harder than the corresponding LP problem

Linear programming

- $f(x) = c^{\mathrm{T}}x$, $S = \{ x \mid \boldsymbol{A}x = \boldsymbol{b}, x \geq 0 \}$
  - c is an n x 1 vector, A is an m x n matrix and b is an $m \times 1$ vector

- For general x, these problems can be solved exactly (e.g. simplex technique). For integer x, the problem is *NP*-complete.

Inequality

- Inequality constraints can easily be introduced by adding an extra variable
  - max $2x_1 + 3x_2$ subject to $x_1 + x_2 \leq 10$ is equivalent to max $2x_1 + 3x_2$ subject to $x_1 + x_2 + x_3 = 10$
  - For "≥", we would insert (-x3) into the constraint
  - The extra variable is called a slack variable – it does not appear in the objective function. Because this is so straight-forward, many ILP solving programs allow you to express constraints with inequality directly.

Example: capital budgeting

- A firm has n projects that it would like to undertake, but due to budget limitations, not all can be selected. In particular, project j has a value of c_j, and requires an investment of a_{ij} in the time period i, $i = 1,\ldots,m$. The capital available in time period i is b_i.
- Objective: maximize the total value, subject to given budget constraints

Example: capital budgeting

A set of variables x_j, which we interpret as:

- $x_j = 1$, project j is selected
- $x_j = 0$, project j is not selected

Then the objective function can be formulated as

$$\sum_{j=1}^{n} c_j x_j$$

The constraints are

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i,\ i = 1,\ldots,m; \quad x_j \le 1,\ j = 1,\ldots,n$$

Linear programming problems

$$
\begin{array}{lrl}
\textbf{maximize} & z = & -4x_1 + x_2 - x_3 + x_4 \\
\textbf{subject to} & & -7x_1 + 5x_2 + x_3 + x_4 = 8 \\
& & -2x_1 + 4x_2 + 2x_3 - x_4 = 10 \\
& & x_1, \quad x_2, \quad x_3, \quad x_4 \geq 0
\end{array}
$$

We will describe LPs that start with the following (canonical) form:

- **equality constraints**
- **nonnegative (right hand side, RHS) variables**

Fundamental theorem

Extreme point (or Simplex filter) theorem:

If the maximum or minimum value of a linear function defined over a polygonal convex region exists, then it is to be found at the boundary of the region

Boundary points: basic feasible solutions

What does the extreme point theorem imply?

- A finite number of extreme points (bfs) implies a finite number of solutions!
- Hence, search is reduced to a finite set of points
- However, a finite set can still be too large for practical purposes
- **Simplex method provides an** efficient systematic search guaranteed to converge in a finite number of steps.

Basic feasible solutions

- Each corner point solution of the polyhedron is a basic feasible solution.

- The simplex method is a systematic way of moving from one basic feasible solution to another, always improving the solution, until the optimum solution is obtained.

Basic Feasible Solutions

$$
\begin{array}{llrrrrrl}
\textbf{maximize} & z = & -4x_1 & + & x_2 & - & x_3 & + & x_4 & \\
\textbf{subject to} & & -7x_1 & + & 5x_2 & + & x_3 & + & x_4 & = 8 \\
 & & -2x_1 & + & 4x_2 & + & 2x_3 & - & x_4 & = 10 \\
 & & x_1, & & x_2, & & x_3, & & x_4 & \geq 0
\end{array}
$$

Suppose there are m constraints, n variables

A basic solution is found by setting n-m variables to 0 and solving the remaining system with m variables and m constraints.

- **The n – m variables are called non-basic variables**
- **The m variables are called basic variables**

Basic feasible solutions

$$
\begin{array}{llrrrrrl}
\textbf{maximize} & z = & -4x_1 & + & x_2 & - & x_3 & + & x_4 & \\
\textbf{subject to} & & -7x_1 & + & 5x_2 & + & x_3 & + & x_4 & = 8 \\
 & & -2x_1 & + & 4x_2 & + & 2x_3 & - & x_4 & = 10 \\
 & & x_1, & & x_2, & & x_3, & & x_4 & \geq 0
\end{array}
$$

$$
\begin{array}{ccccc|c}
-z & x_1 & x_2 & x_3 & x_4 & \\
\hline
1 & -4 & 1 & -1 & 1 & = \; 0 \\
0 & -7 & 5 & 1 & 1 & = \; 8 \\
0 & -2 & 4 & 2 & -1 & = \; 10
\end{array}
$$

Basic feasible solutions

$-z$	x_1	x_2	x_3	x_4		
1	-11	6	0	2	=	8
0	-7	5	1	1	=	8
0	12	6	0	-3	=	-6

Example: Suppose we want the solution with basic variables x_3 and x_4, and thus x_1 and x_2 are non-basic.

We then perform pivot operations.

Basic Feasible Solutions

$-z$	x_1	x_2	x_3	x_4		
1	-3	2	0	0	=	4
0	-3	3	1	0	=	6
0	-4	2	0	1	=	2

Next pivot on the –3.

Basic Feasible Solutions

Canonical form: basic variables have a single one in the column.

reduced costs →

$-z$	x_1	x_2	x_3	x_4		
1	-3	2	0	0	=	4
0	-3	3	1	0	=	6
0	-4	2	0	1	=	2

The basic solution is found by setting non-basic variables to 0. We get $x_1=0$, $x_2=0$, $x_3=6$, $x_4=2$.

This solution also satisfies $x \geq 0$. It is called a basic feasible solution.

Simplex algorithm

$z = -3x_1 + 2x_2 - 4$

$-z$	x_1	x_2	x_3	x_4		
1	-3	2	0	0	=	4
0	-3	3	1	0	=	6
0	-4	2	0	1	=	2

The entering variable for a max problem is a variable with positive reduced cost.

The pivot element is chosen uniquely in the column of the entering variable so that the next basis is feasible.

Simplex algorithm

The pivot element is chosen to leave the basis according to pivot rules (e.g. a min ratio rule).

$-z$	x_1	x_2	x_3	x_4		
1	1	0	0	-1	=	-2
0	3	0	1	-1.5	=	3
0	-2	1	0	.5	=	1

A pivot is carried out, leading to the next bfs.

Variable x_4 has left the basis.
The new basis consists of x_2 and x_3.

Simplex algorithm

$-z$	x_1	x_2	x_3	x_4		
1	0	0	-1/3	-1/2	=	-3
0	1	0	1/3	-1/2	=	1
0	0	1	2/3	-1/2	=	3

Optimality conditions for a maximization problem: all reduced costs are non-positive.

Simplex algorithm

Pivots are carried out until the bfs is optimal.

$-z$	x_1	x_2	x_3	x_4		
1	0	0	-1/3	-1/2	=	-3
0	1	0	1/3	-1/2	=	1
0	0	1	2/3	-1/2	=	3

$$z = -x_3/3 - x_4/2 + 3$$

This new bfs is optimal. Increasing x_3 or x_4 makes the solution worse.

Duality in linear programming

- Every primal problem there exists a corresponding dual problem
  - Primal Maximization → Dual Minimization
  - Primal Minimization → Dual Maximization

Duality in linear programming

- Primal has *n* choice variable and *m* constraints and dual has *m* variables and *n* constraints
- Right hand side elements (b_i) in the primal correspond to coefficient of the objective function of the dual
- The a_{ij} constraint coefficients become a_{ji} in dual

Primal and dual in matrix form

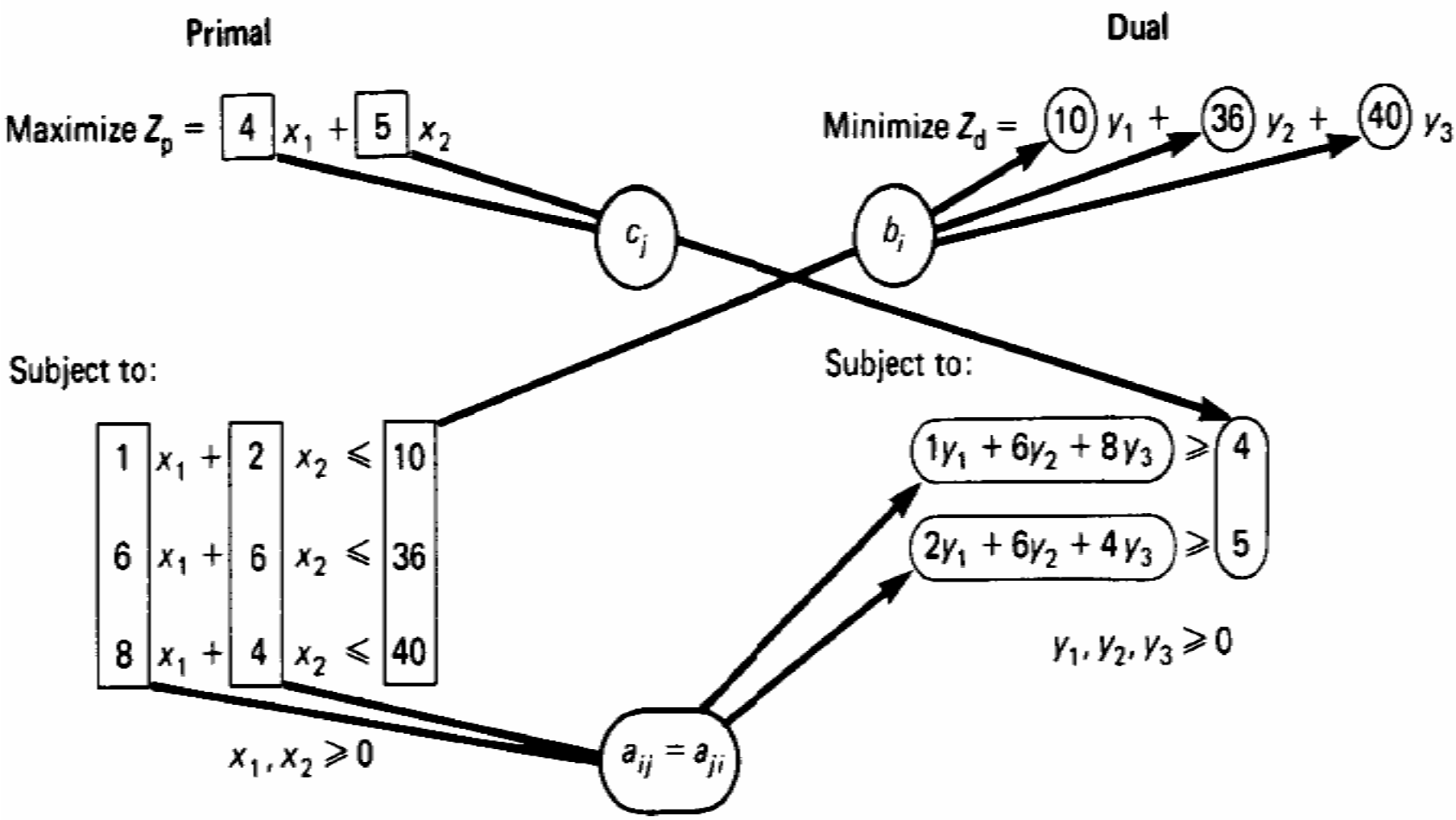
- Primal

$$\max Z = c'x$$
$$\text{s. t. } Ax \leq r$$
$$x \geq 0$$

- Dual

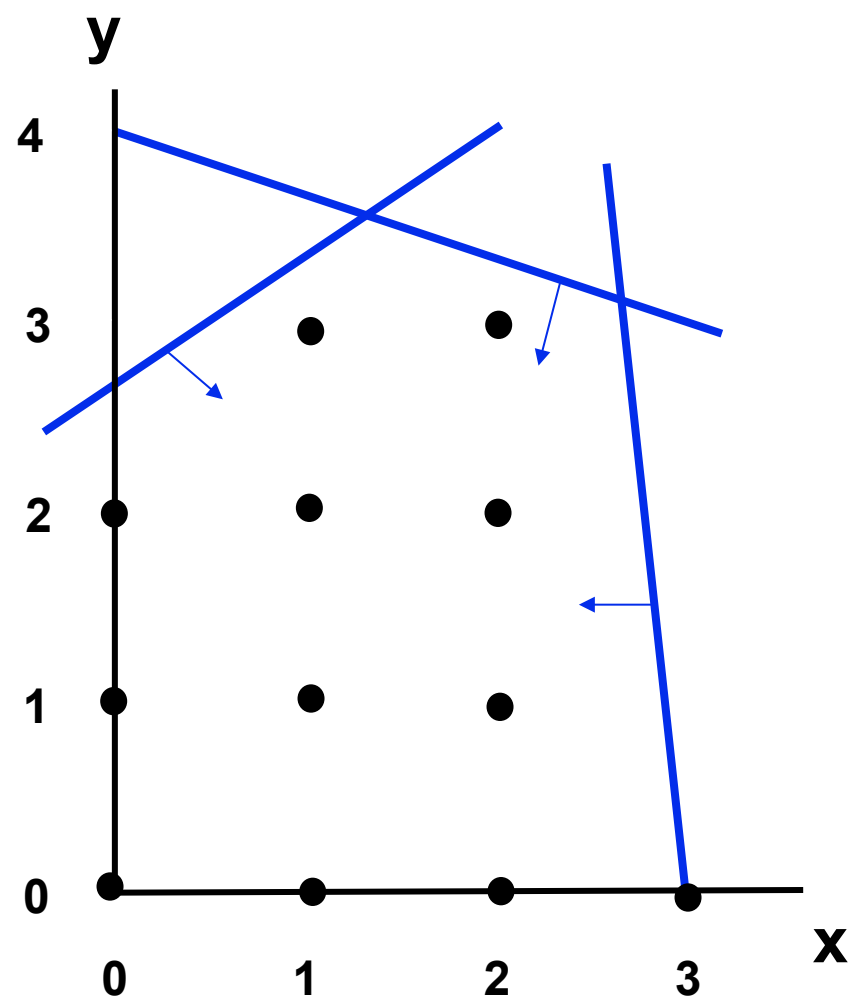
$$\max Z^* = r'y$$
$$\text{s. t. } A'y \geq c$$
$$y \geq 0$$

Primal and dual relationship



LP → Integer Programming

- Feasible region is a set of discrete points.
- Can't be assured a corner point solution.
- IP is NP-hard problem (By reduction from Satisfiability)
- Solving it as an LP provides a relaxation and a bound on the solution.



Duality theory

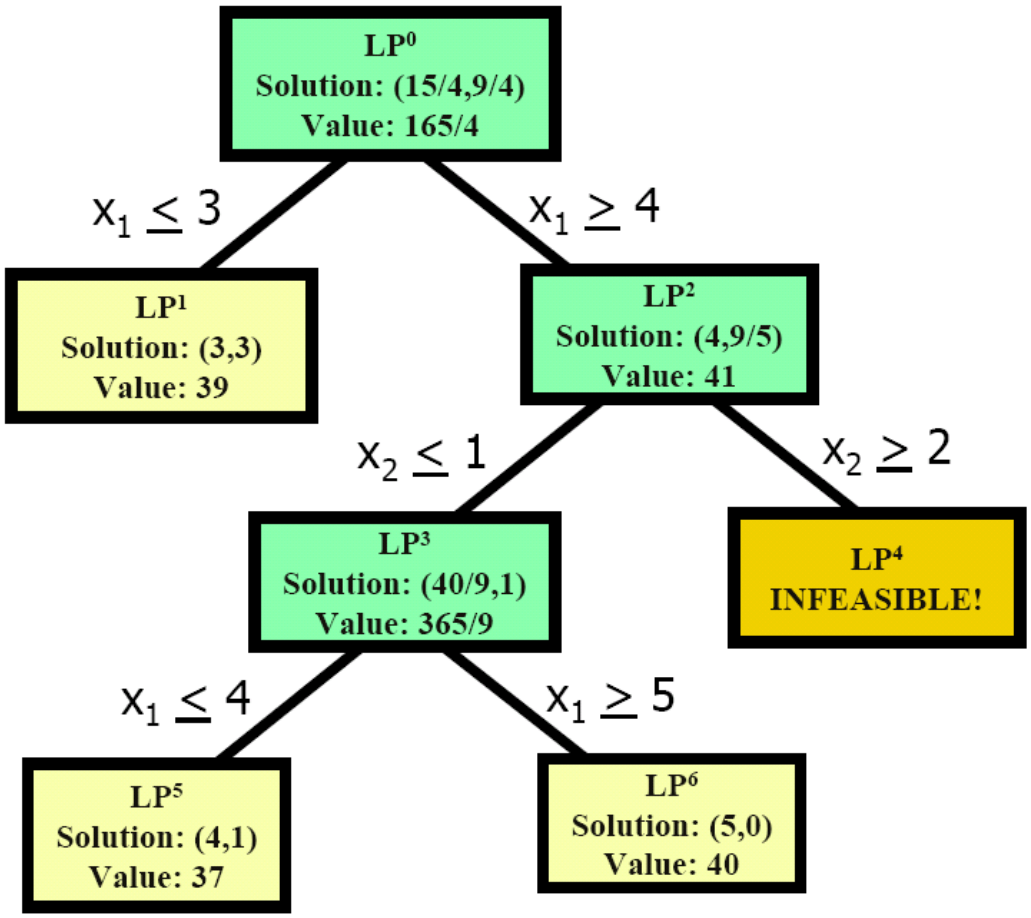
- (**Weak duality theorem**) the objective function value of the dual (*min*) at any feasible solution ≥ the objective function value of the primal (*max*) at any feasible solution.
- (**Strong duality theorem**) if the primal has an optimal solution, x^*, then the dual also has an optimal solution, y^*, such that $c' x^* = r' y^*$.

Solutions to IP

- Package
  - GNU Linear Programming Kit (GLPK) http://www.gnu.org/software/glpk/ → free
  - Cplex: mathematical optimizer http://www.ilog.com/products/cplex/ → commercial
- Enumeration
  - Small size of problems
- Branch and bound (tree-like search)
- Specially designed algorithms
  - LP relaxation

IP Solver = LP Solver + B&B

- Solve the LP relaxation of the IP
- If the LP solution has a non-integer variable, branch on that variable and solve the 2 resultant LPs
- Traverse the tree recursively until:
  - All terminals exposed (unsolvable LPs are terminal); or
  - A sub-LP solution is worse than an existing integer solutio



IP:

Maximize $8x_1 + 5x_2$

S.t. $x_1 + x_2 \leq 6$

$9x_1 + 5x_2 \leq 45$

$x_1, x_2 \in Z^+$

$\Downarrow$

LP Relaxation:

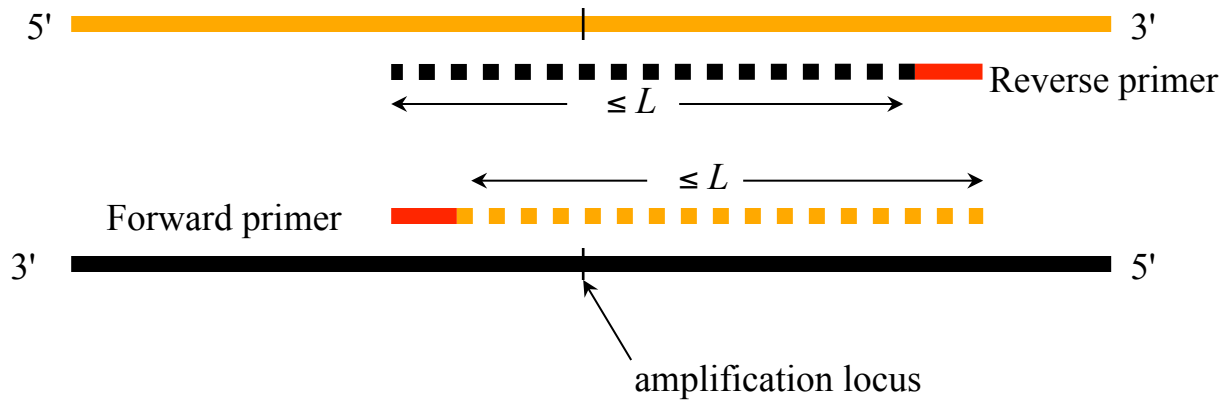
Maximize $8x_1 + 5x_2$

S.t. $x_1 + x_2 \leq 6$

$9x_1 + 5x_2 \leq 45$

$x_1, x_2 \geq 0$

Primer pair selection problem



- **Given:**
  - Genomic sequence around amplification locus
  - Primer length k
  - Amplification upper-bound L
- **Find:** Forward and reverse primers of length k that hybridize within a distance of L of each other and optimize amplification efficiency **(melting temperatures, secondary structure, cross hybridization, etc.)**

Primer set selection: multiplex experiment

- Spotted microarray synthesis [Fernandes and Skiena' 02]
  - Need unique pair for each amplification product, but primers can be re-used to minimize cost
  - Potential to reduce #primers from $O(n)$ to $O(n^{1/2})$ for n products

Primer set selection: application

- SNP Genotyping
  - Thousands of SNPs that must genotyped using hybridization based methods (e.g., SBE)
  - Selective PCR amplification needed to improve accuracy of detection steps (whole-genome amplification not appropriate)
  - No need for unique amplification!
  - Primer minimization is critical
    - Fewer primers to buy
    - Fewer multiplex PCR reactions

Primer set selection problem

- **Given:**
  - Genomic sequences around each amplification locus
  - Primer length k
  - Amplification upperbound L
- **Find:**
  - Minimum size set of primers S of length k such that, for each amplification locus, there are two primers in S hybridizing to the forward and reverse sequences within a distance of L of each other
  - Uniqueness constraint: S should contain a unique pair of primers amplifying each locus

Selection with uniqueness Constraints

- Can be modeled as minimum multicolored subgraph problem:
  - Vertices of the graph correspond to candidate primers
  - add edge colored by color *i* between two primers if they amplify i-th SNP and do not amplify any other SNP
  - Goal is to find minimum size set of vertices inducing edges of all colors
- NP-hard problem
- Trivial approximation algorithm: select 2 primers for each SNP
  - $O(n^{1/2})$ approximation since at least $n^{1/2}$ primers required by every solution

Integer program formulation

- Variable x_u for every vertex (candidate primer) u
  - x_u set to 1 if u is selected, and to 0 otherwise
- Variable y_e for every edge e
  - y_e set to 1 if corresponding primer pair selected to amplify one of the SNPs
- Objective: minimize sum of x_u
- Constraints:
  - for each i, sum of $\{y_e : e \text{ amplifying SNP } i\} \geq 1$
  - $2y_e \leq x_u + x_v$ for every e incident to u & v