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:

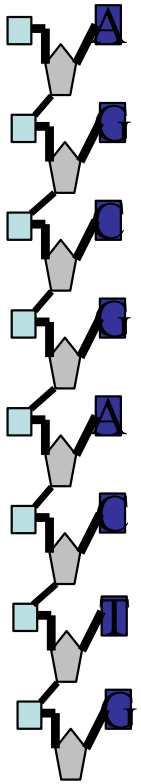
Element	Weight
4 Assignments: only accept answers in tex file (**Latex)**	40%
Midterm	20%
Final Exam	20%
Algorithm Presentation	15%
Class Participation	5%
TOTAL	100%

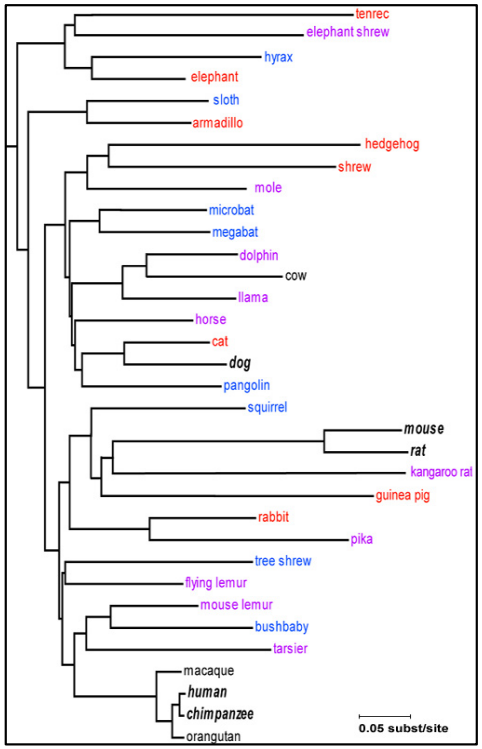
Getting started

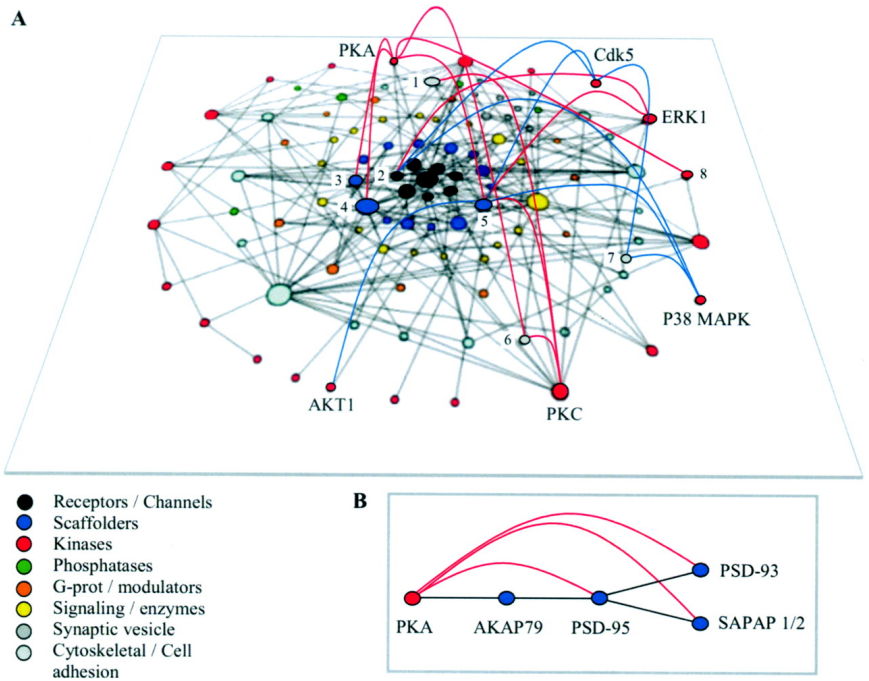
- Bioinformatics → solving biological problems by computers;

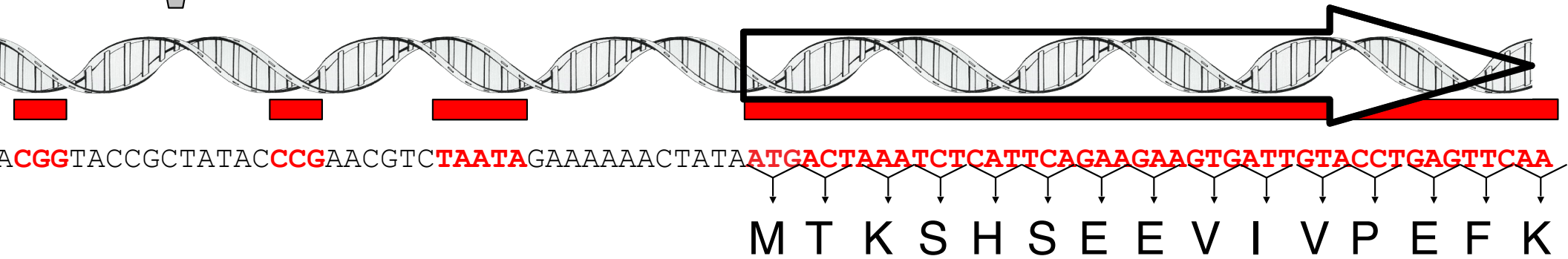
- Why computers?
  - Large amount of data
  - Complex computation

What kind of biological data?

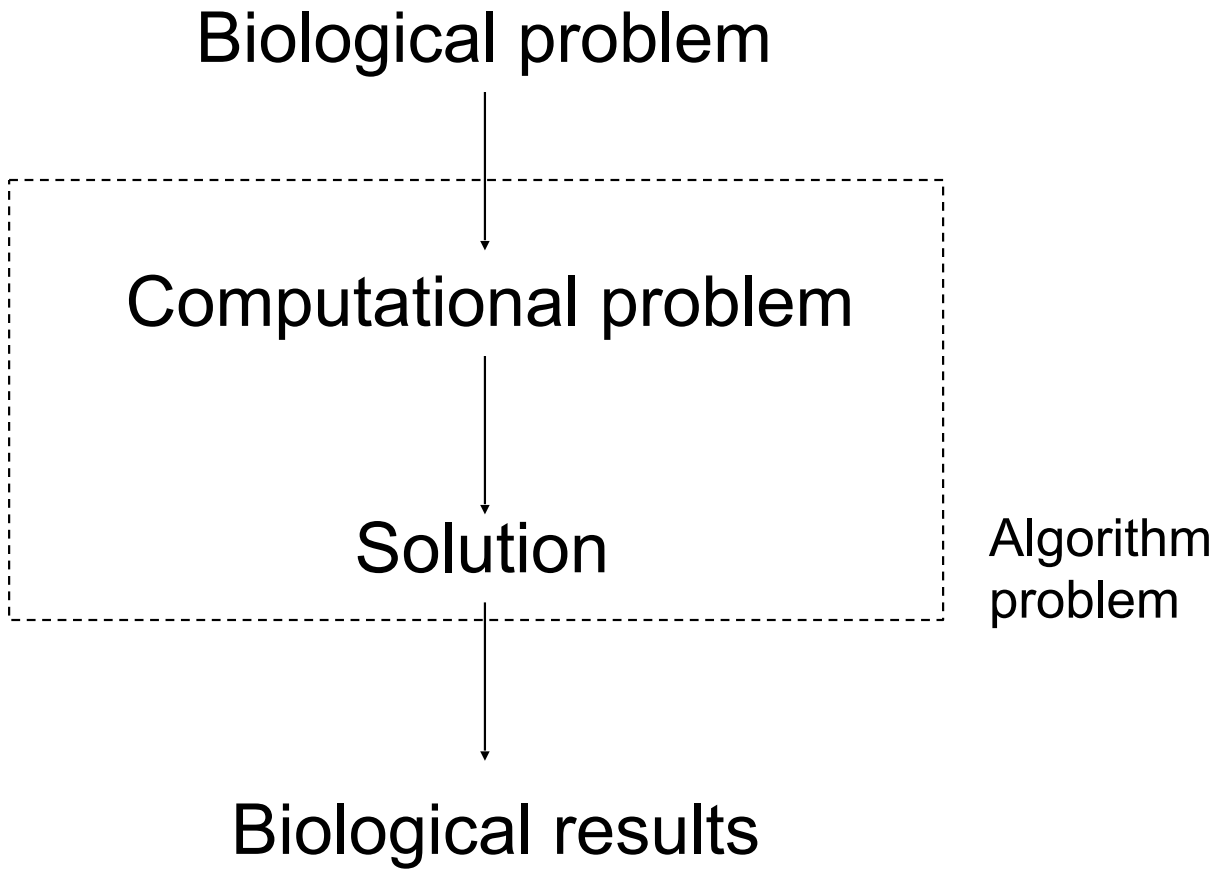




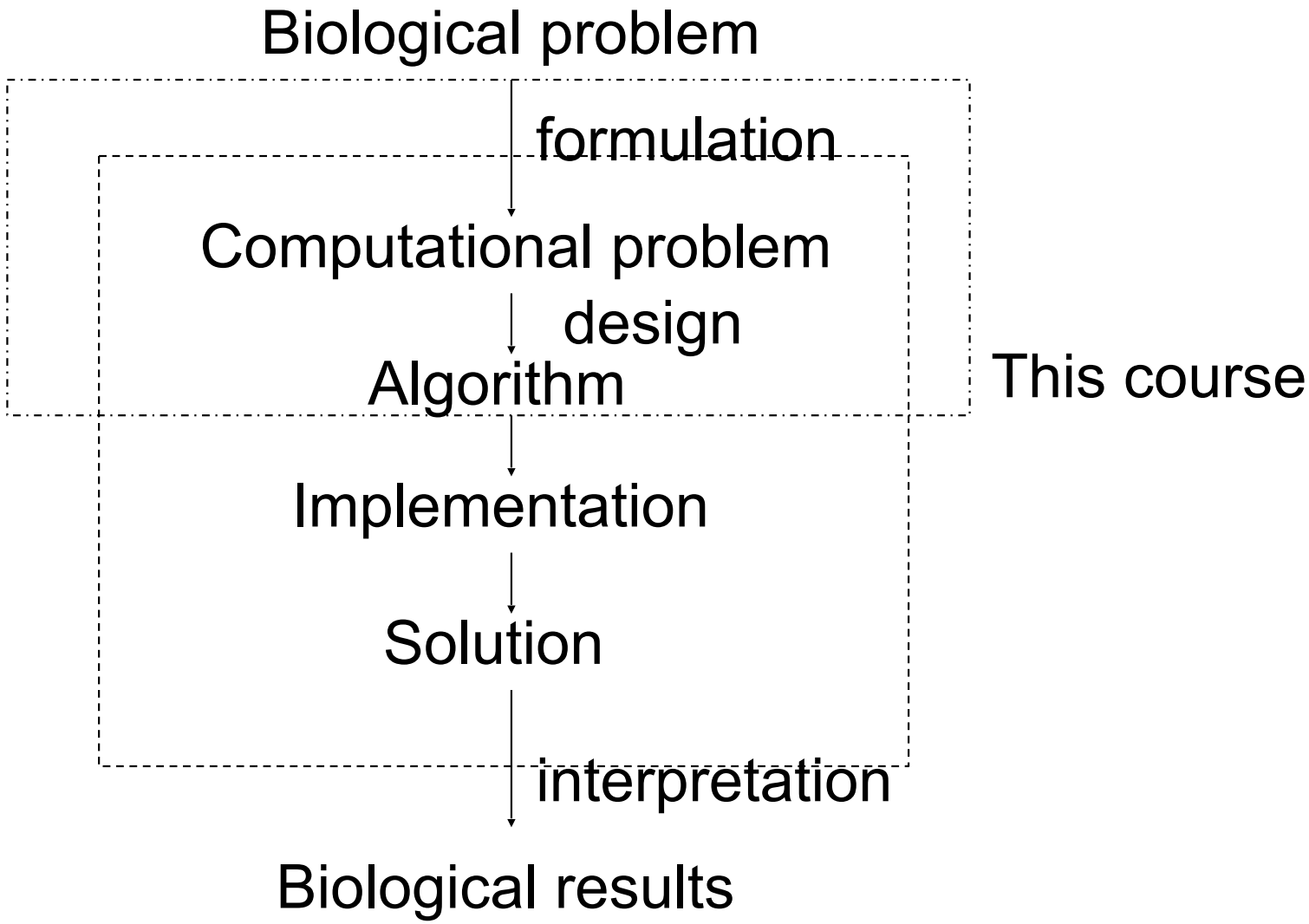




A general solution pipeline for bioinformatics problems



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 $Q=(P-C)/5$.

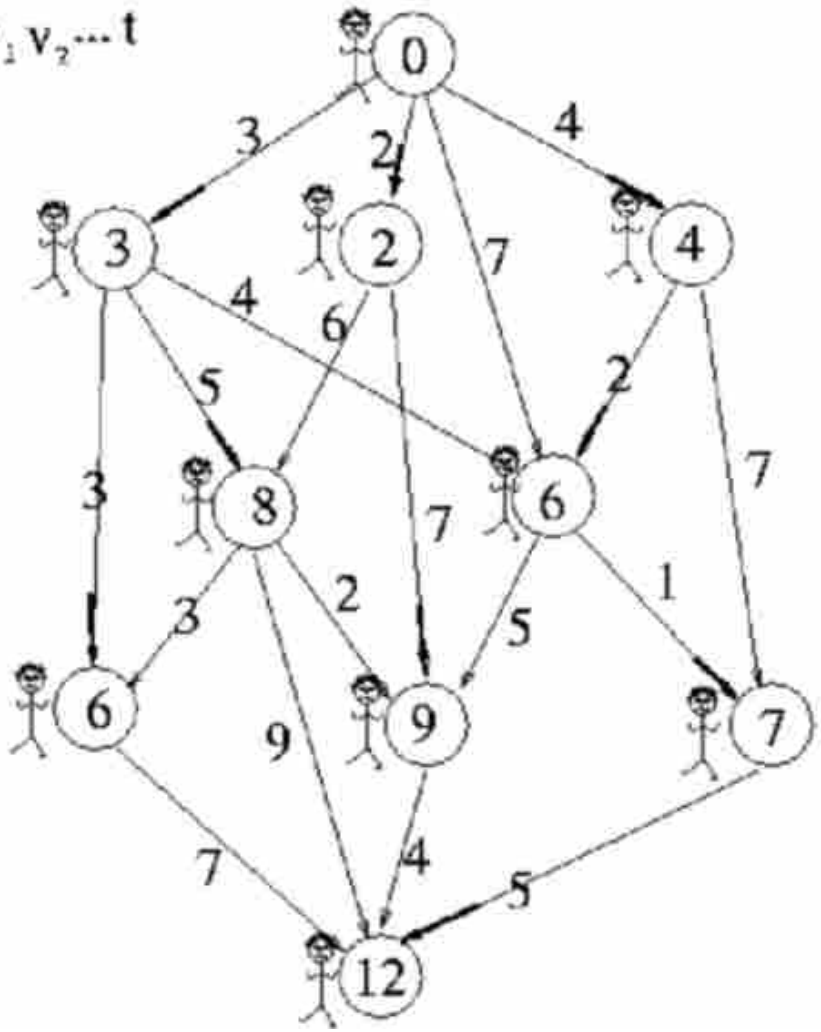
From $0\leq B\leq 1$ and $Q = 2A + B$ we have $B=Q \bmod 2$.
Similarly $b=Q \bmod 2$, and hence $b=B$.

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

Breaking problems down: dynamic programming

Solve each subinstance

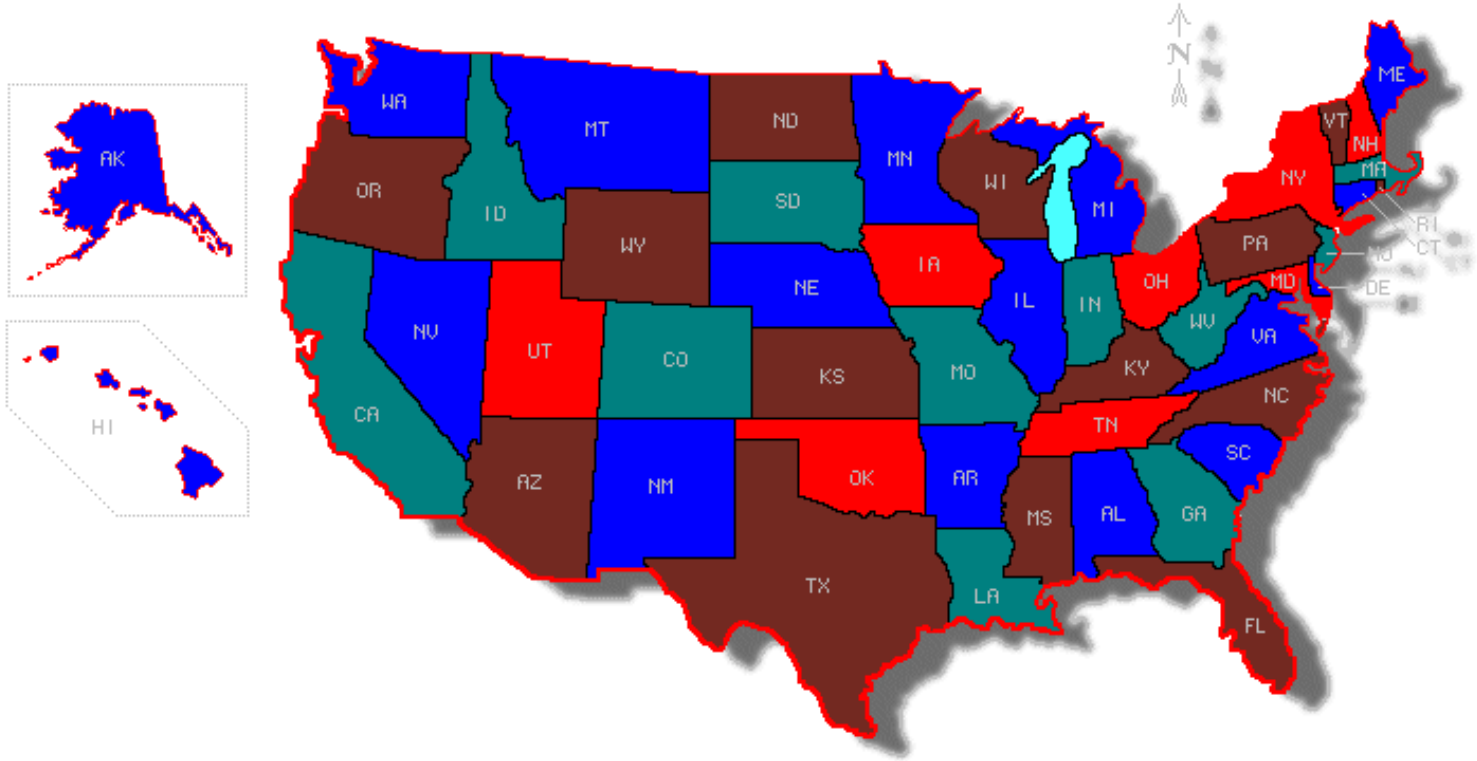
s v_1 v_2 ... t



	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

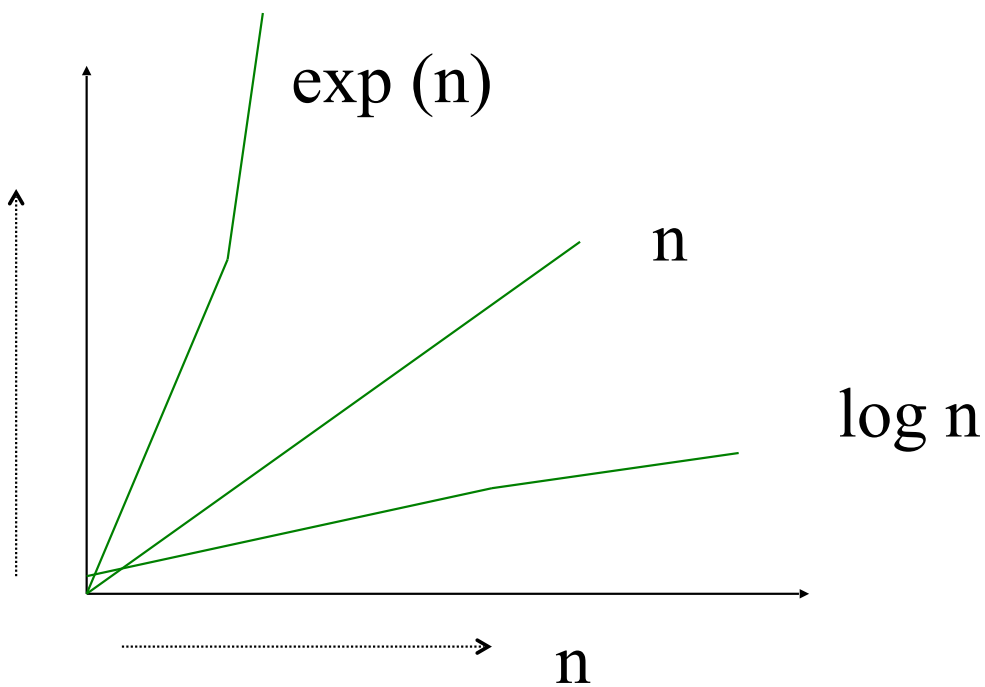
We do not distinguish between the basic operations.

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]
    }
```

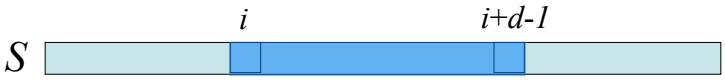
= Ω(|p|)
=O(|s|)

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, …

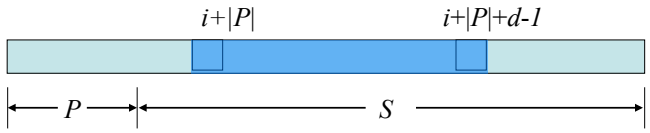
```
S =  a a g c a a t a a a g c
Z = 12 1 0 0 2 1 0 2 4 1 0 0
       a     a a   a a
             a     a a g c
                     a
```

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

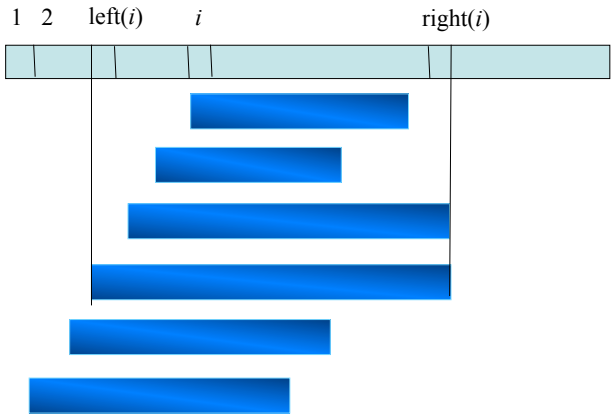
```
For i=1 to |S| {
  set j = i;
  set Z(i) = 0;
  while (S[j]==S[j-i+1]){
    Z(i)++;
    j++;
  }
}
```

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

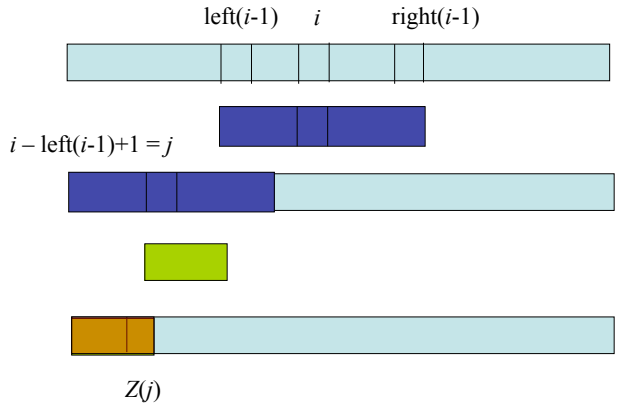
- Computing $Z(i)$, right(i), left(i) from
  - $Z(1)$, $Z(2)$, …, $Z(i-1)$;
  - right($i-1$);
  - left($i-1$).

Case 1: right(i-1) ≤ i-1.

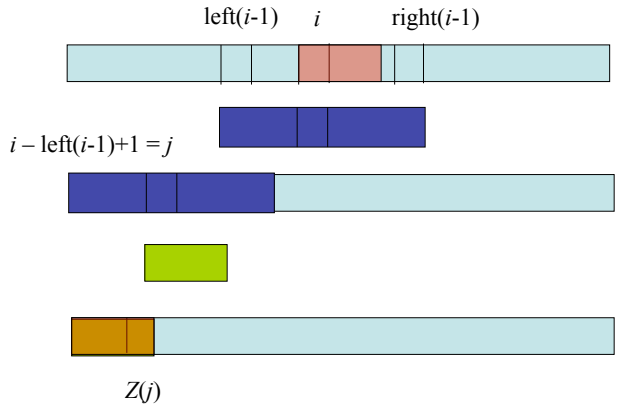
- right(i-1) does not cover i.
- Computing $Z(i)$ naively in $O(1+Z(i))$ time.
- left(i) = i.
- right(i) = $i + Z(i) - 1$.
- Observation

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

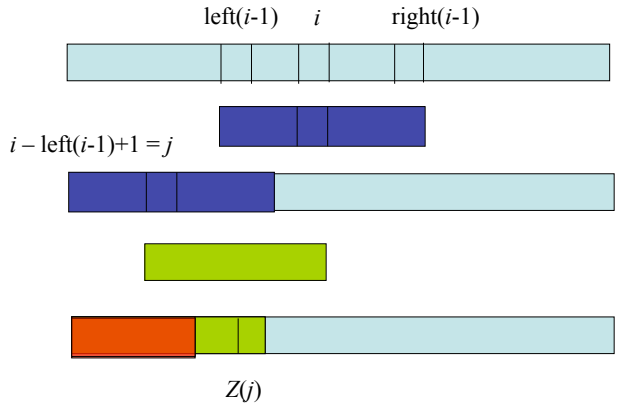
Case 2:right(i-1) ≥ i and Z(j) < right(i-1)-i+1.



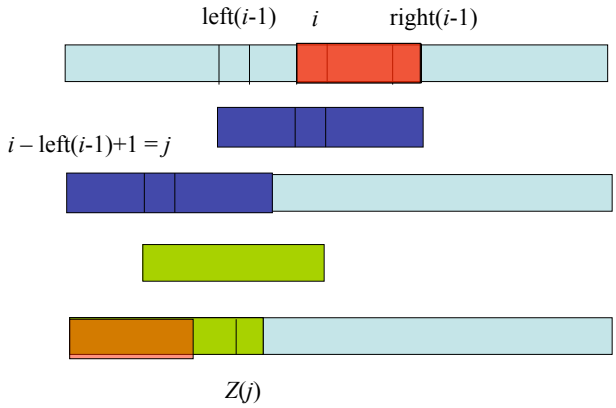
Z(i) = Z(j), left(i) = left(i-1), right(i) = right(i-1).



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

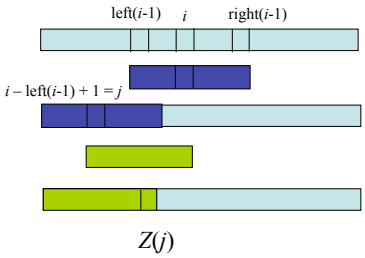


Finding Z(i) by comparisons starting from right(i-1)+1.



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’
  - ‘’

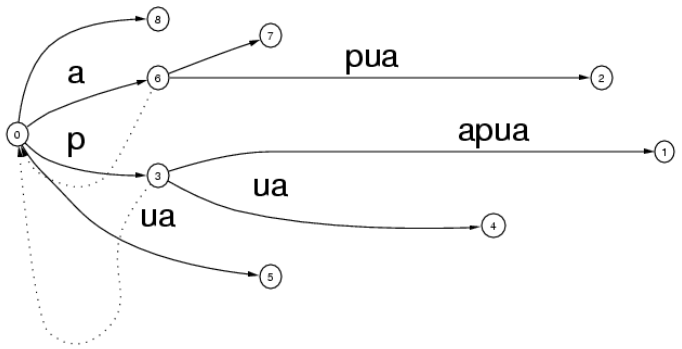
Suffix Trees: Example

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

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

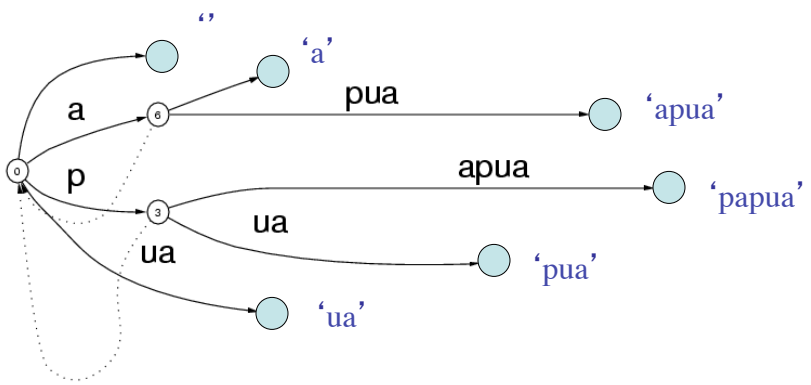
Suffix Trees: Example

- Suffix tree for ‘papua’

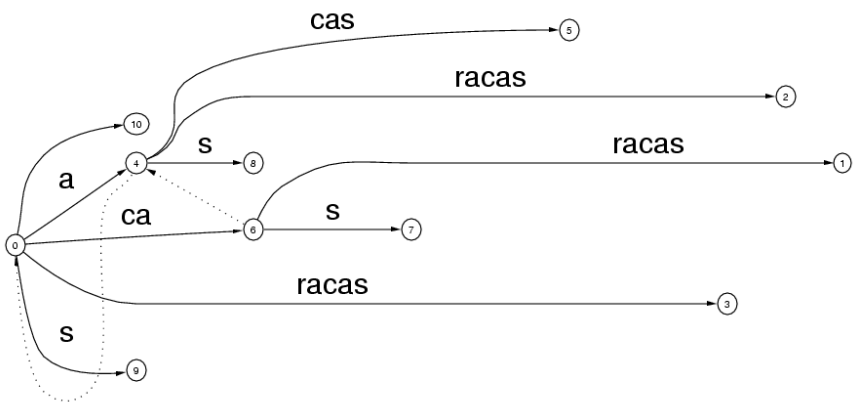


Suffix Trees: Example

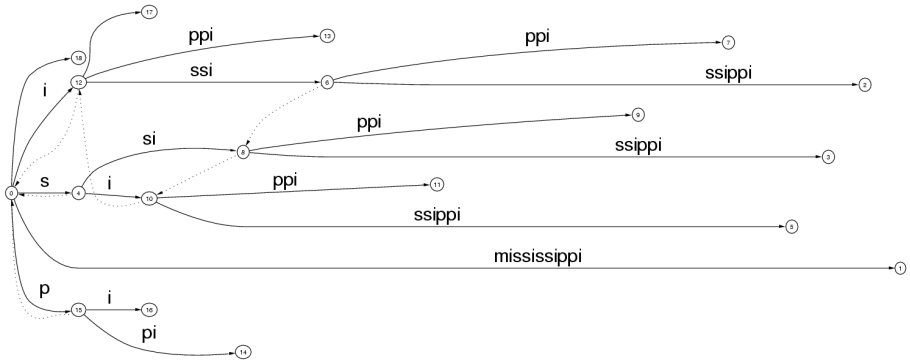
- Suffix tree for ‘papua’



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

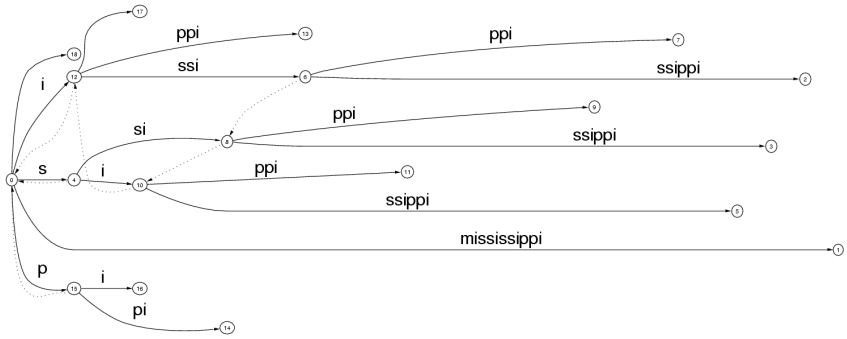
- Given a pattern *P* of length *m*, find all occurrences of *P* in text *T*
  - *O(n+m)* algorithm
- Solution: Build a suffix tree *ST* for text *T* in *O(m)* time. Then, match the characters of *P* along the unique path in *ST* until either *P* is exhausted or no more matches are possible.

Exact string matching problem

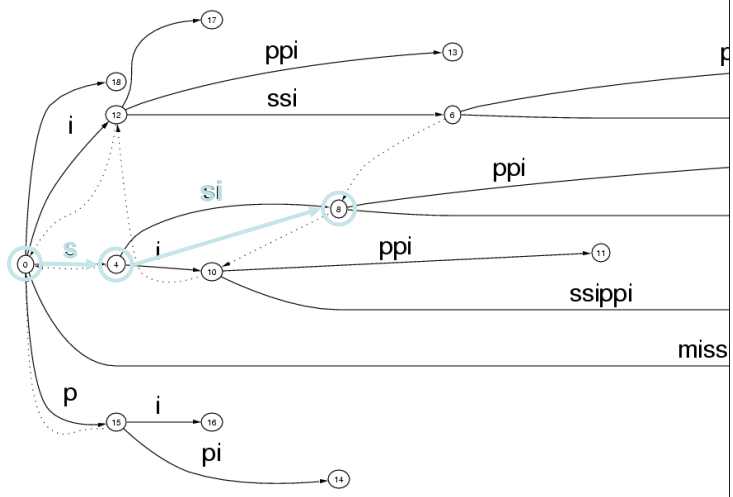
- Find 'ssi' in 'mississippi'

Exact string matching problem

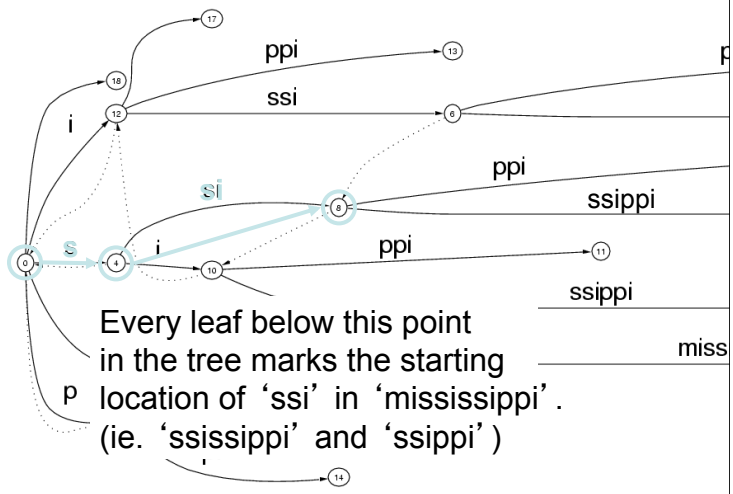
- Find ‘ssi’ in ‘mississippi’



Exact string matching problem



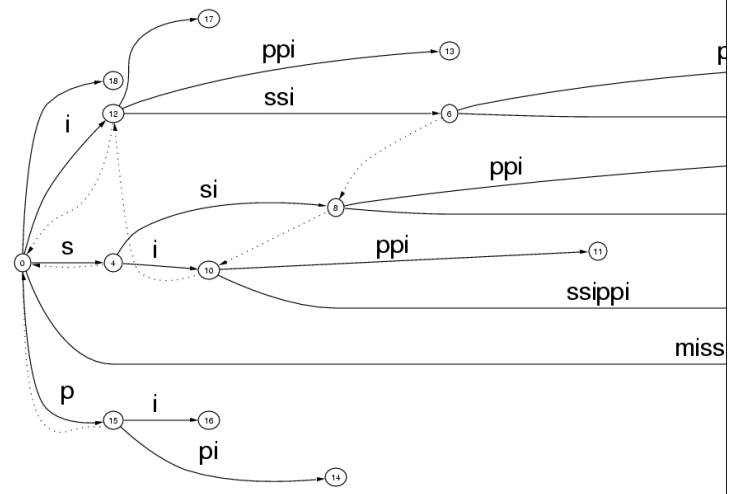
Exact string matching problem



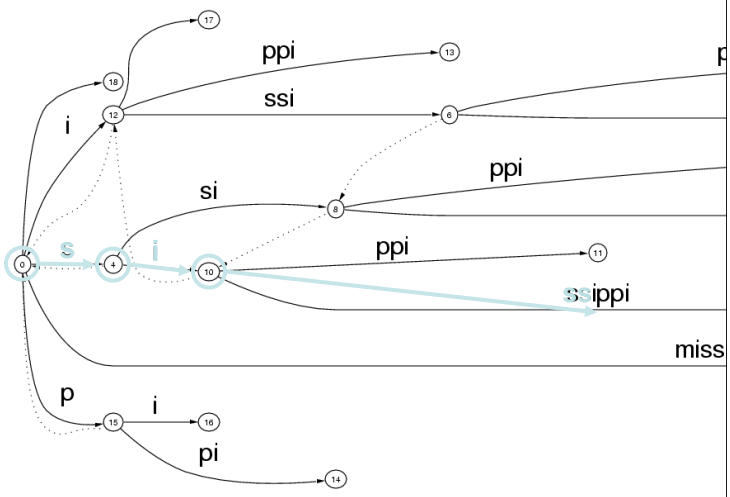
Exact string matching problem

- 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?

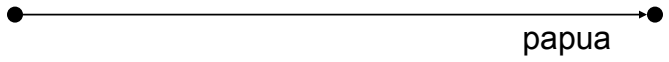
```
while suffixes remain:
   add next shortest suffix to the tree
```

Building the Suffix Tree

- papua

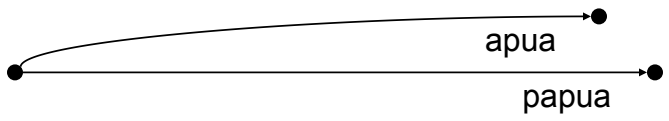
Building the Suffix Tree

- papua



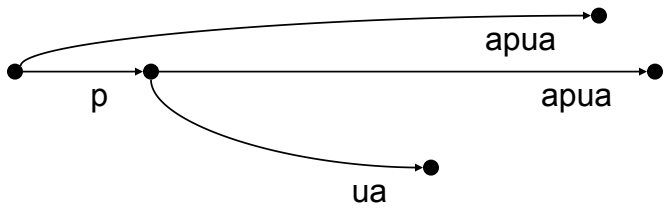
Building the Suffix Tree

- papua



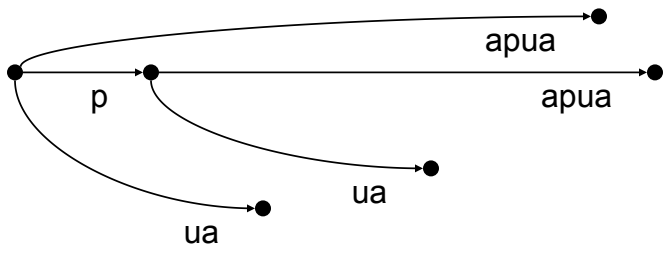
Building the Suffix Tree

- papua



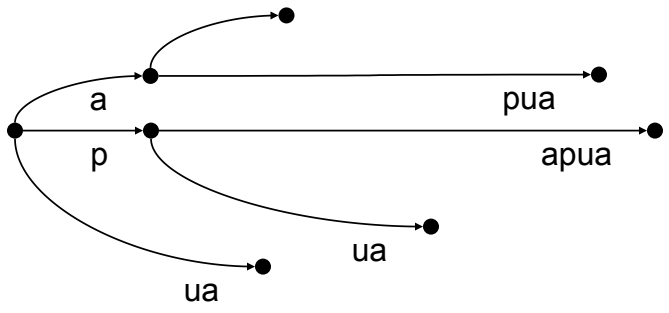
Building the Suffix Tree

- papua



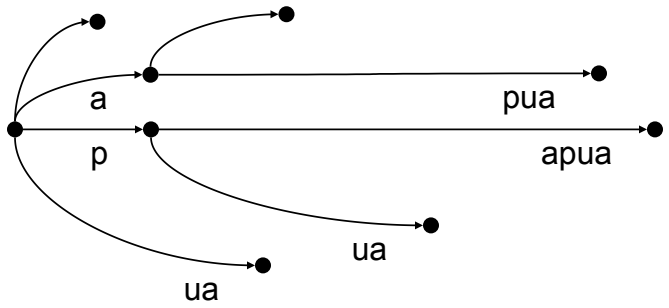
Building the Suffix Tree

- 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

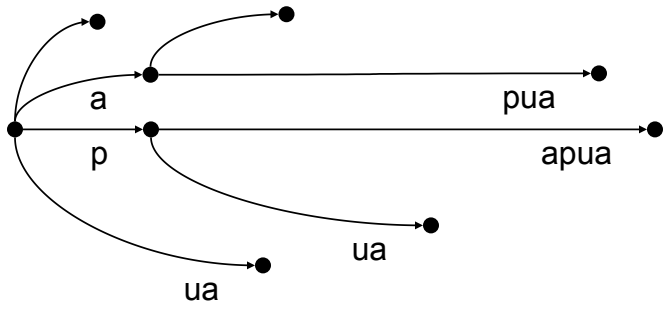
- In the previous example, we assumed that the tree can be built in *O(m)* time.
- Weiner showed original *O(m)* algorithm (Knuth is claimed to have called it "the algorithm of 1973")
- More space efficient algorithm by McCreight in 1976
- Simpler 'on-line' algorithm by Ukkonen in 1995

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.

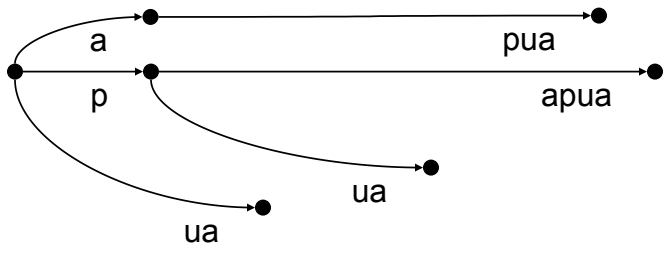
Implicit suffix tree

- papua



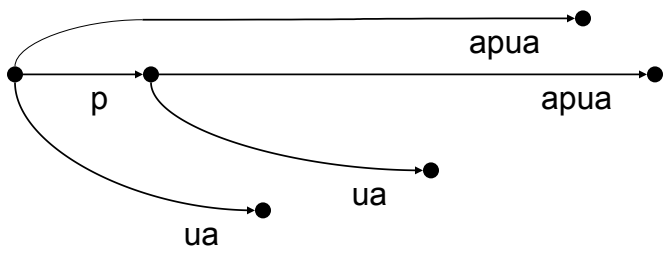
Implicit suffix tree

- papua



Implicit suffix tree

- papua



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

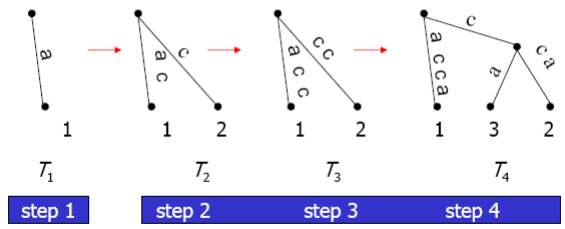
Ukkonen's Algorithm

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

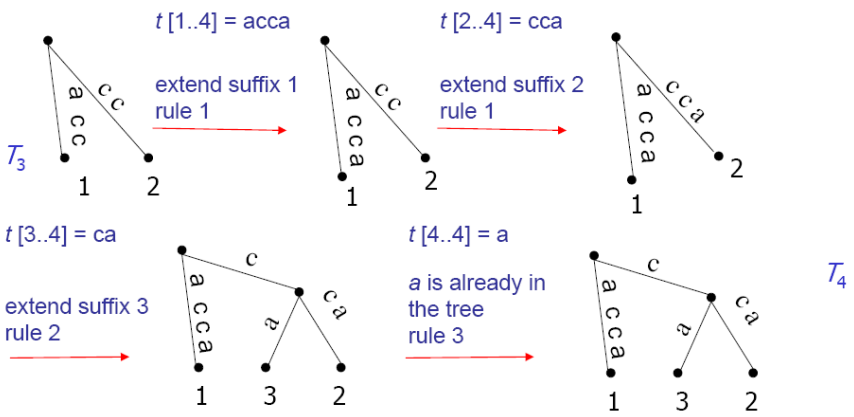
- 3 possible ways to extend *S[j..i]* with character *i +1.*
  1. *S[j..i]* ends at a leaf. Add the character *i+1* to the end of the leaf edge.
  2. No path from the end of *S[j..i]* starts with the *i+1* character. Split the edge and create a new node if necessary, then add a new leaf with character *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 $

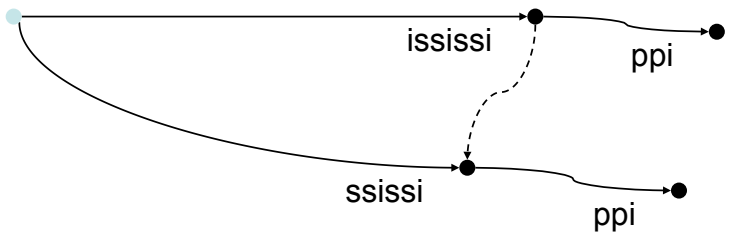
t [1...3] = acc

t [1...4] = acca



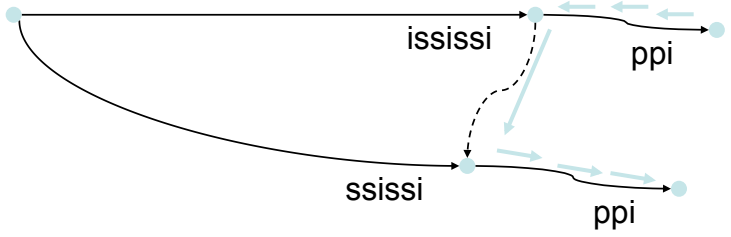
Ukkonen's Algorithm: Speed-up 1

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



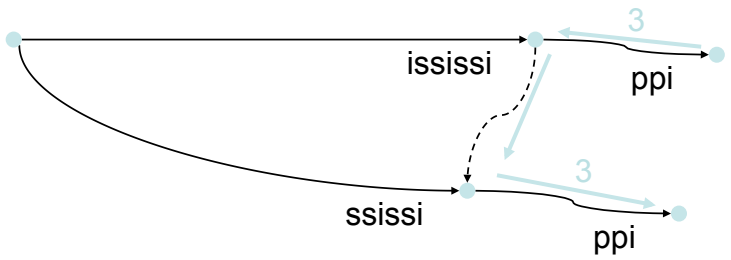
Ukkonen's Algorithm: Speed-up 2

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



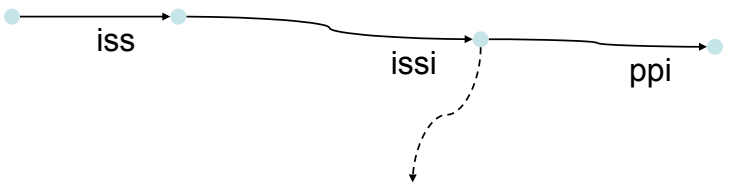
Ukkonen's Algorithm: Speed-up 2

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



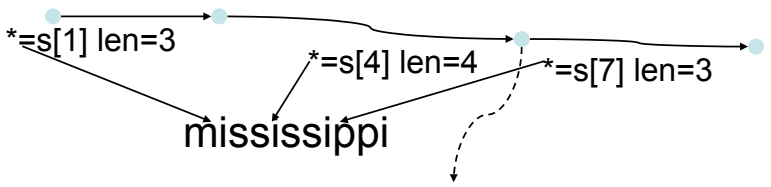
Ukkonen's Algorithm: Speed-up 3

- 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)

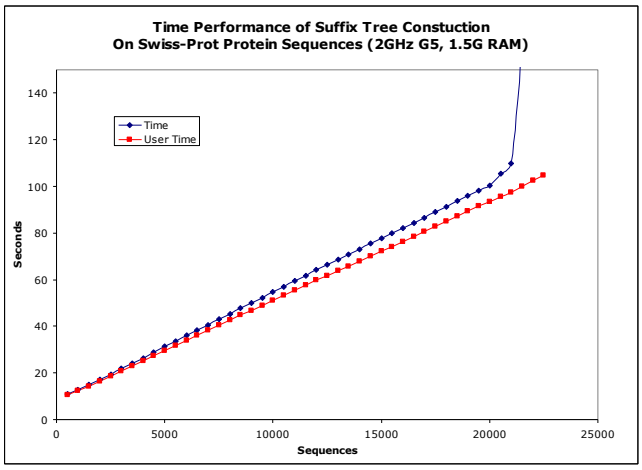
Ukkonen's Algorithm – The Punch Line

- 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!

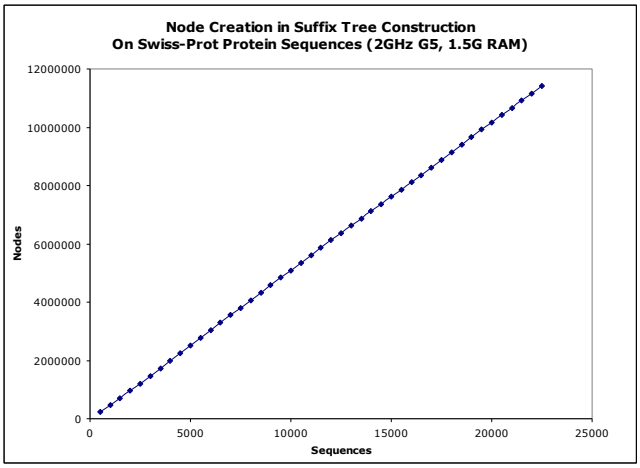
Exact string matching

- 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!

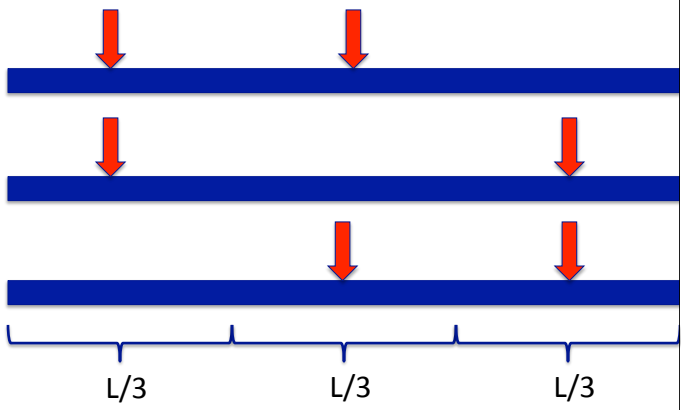
- Important: Trivial algorithm only solves problem under assumptions.

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

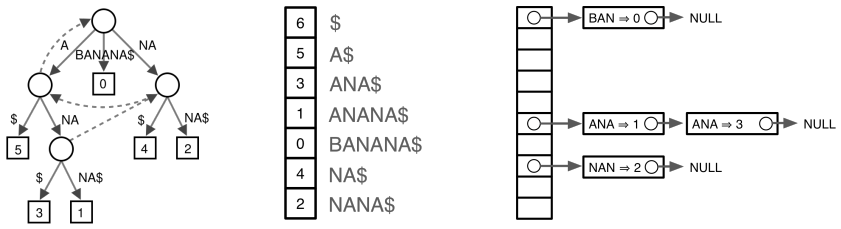
- We need to look up each third of the read in the index.
- For L=30, our index will contain entries of length 10. Each entry will contain on average $N/(4^{L/3})$ or 3,000 positions.
- For each position, we need to compute the number of mismatches.
- Our running time is $L^* M^*3^*N/(4^{L/3})^*T$=81,000,000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

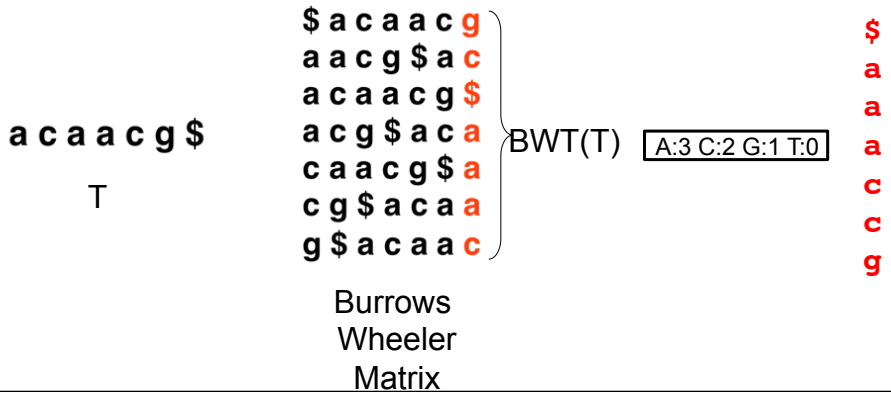
- http://en.wikipedia.org/wiki/Burrows-Wheeler_transform
- Reversible permutation used originally in compression

- 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

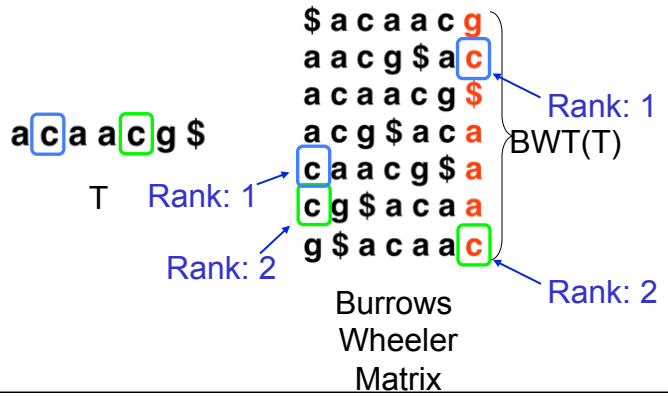
Burrows-Wheeler Transform

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



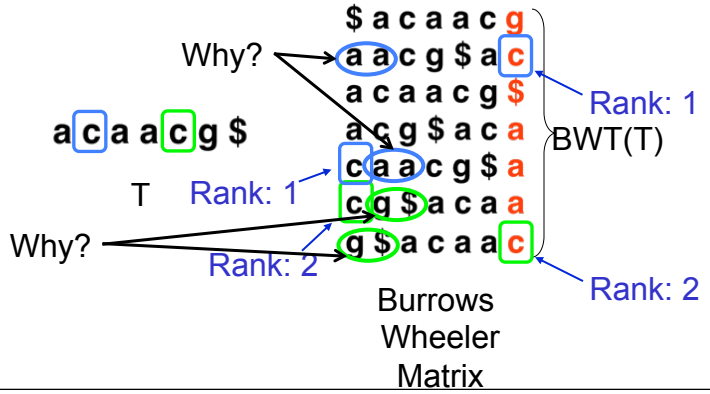
Burrows-Wheeler Transform

- 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**



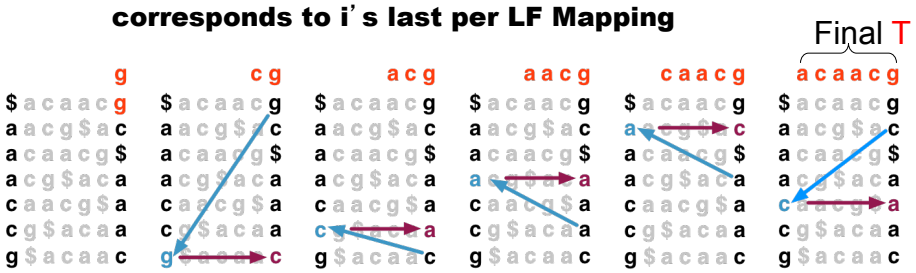
Burrows-Wheeler Transform

- 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

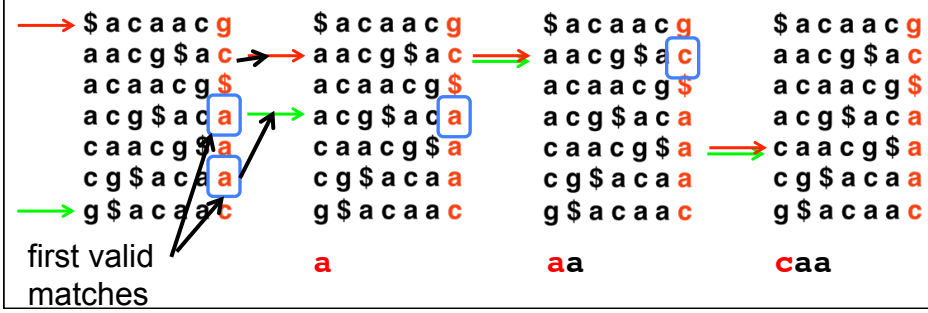
FM Index

- 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.

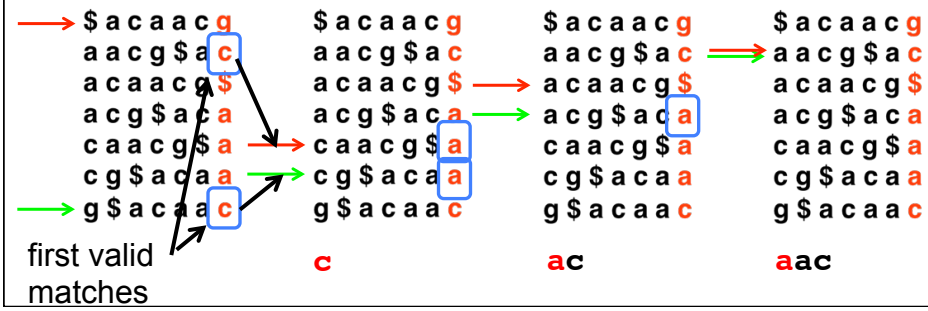
Exact Matching with FM Index

- 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

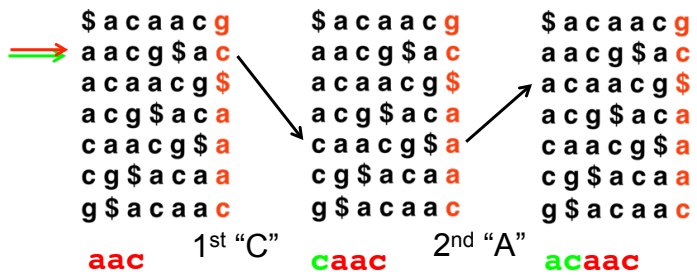
Position = number of moves + position in array

For "aac" = 1 + 1 = 2

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

"walk-left" optimization

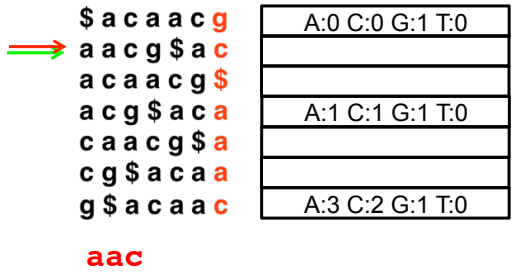
- Each "walk-left" requires counting previous occurrences of symbol in BWT
  - **Example: searching for "aac"**



- 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!

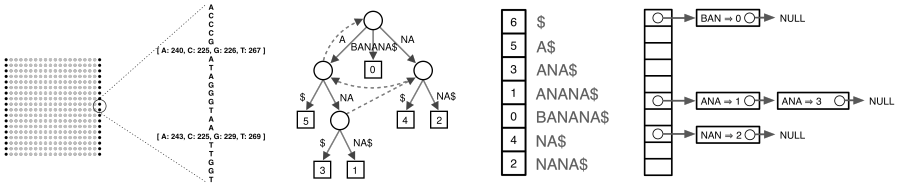
FM Index is Small (Bowtie)

- 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:

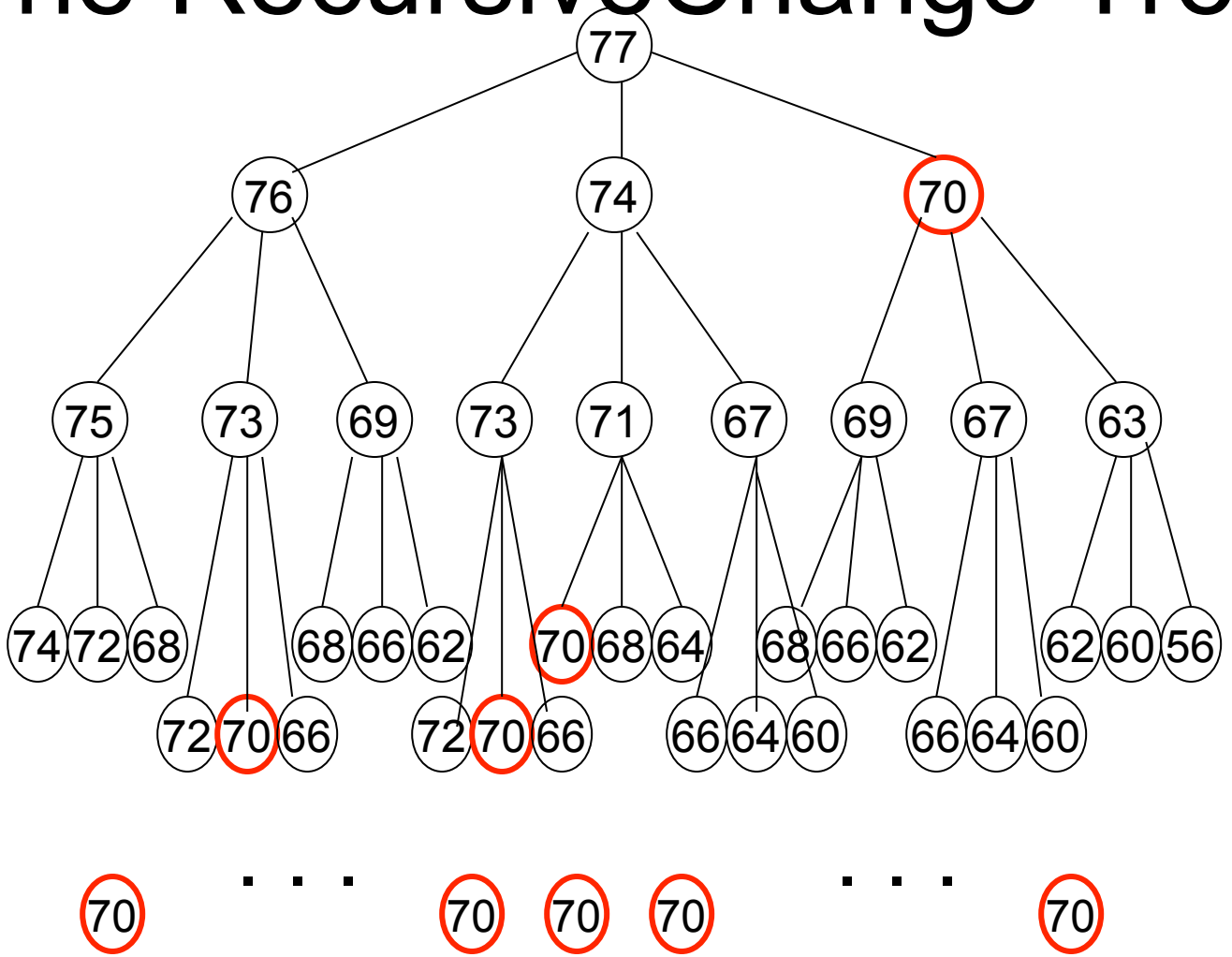
$$minNumCoins(M) = \mathbf{min} \begin{cases} minNumCoins(M-1) + 1 \\ minNumCoins(M-3) + 1 \\ minNumCoins(M-5) + 1 \end{cases}$$

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

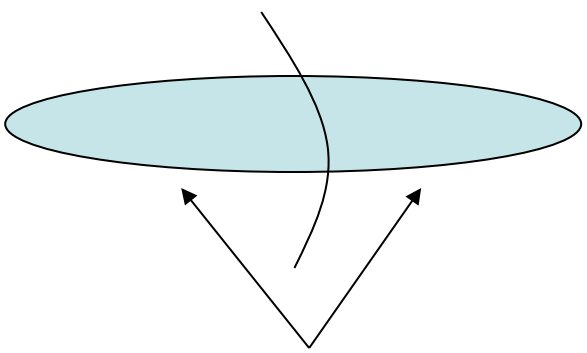
$\boldsymbol{c} = (1,3,7)$

$\boldsymbol{M} = 9$

DP: two key ingredients

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

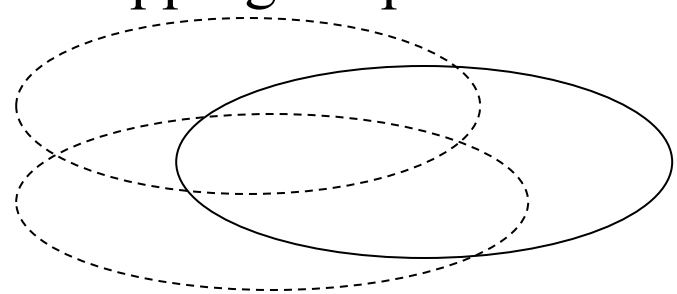
1. optimal substructures



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

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

$\boldsymbol{M} = 9$

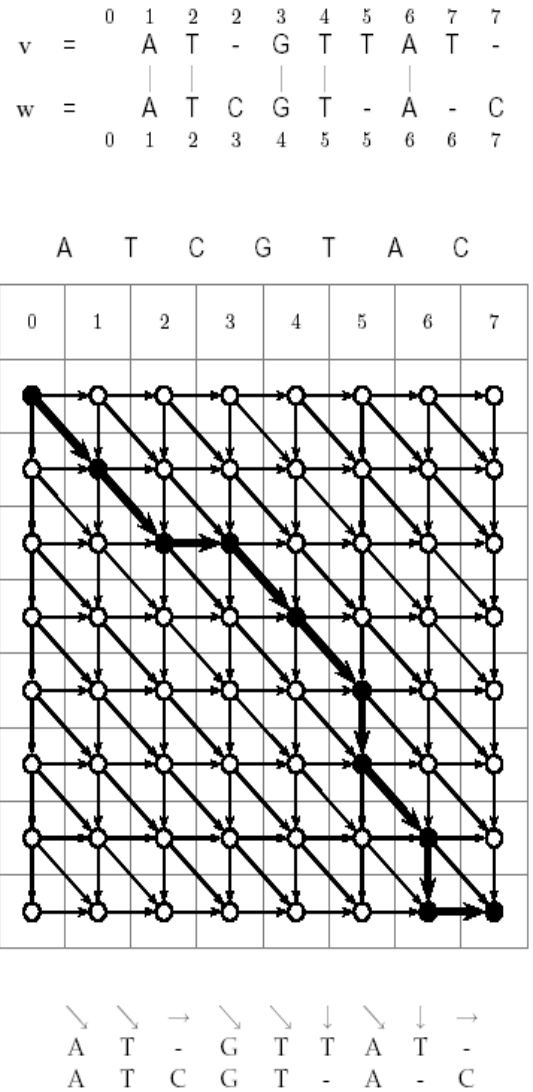
Aligning sequences with insertions and deletions

TGCATAT → ATCCGAT in 4 steps

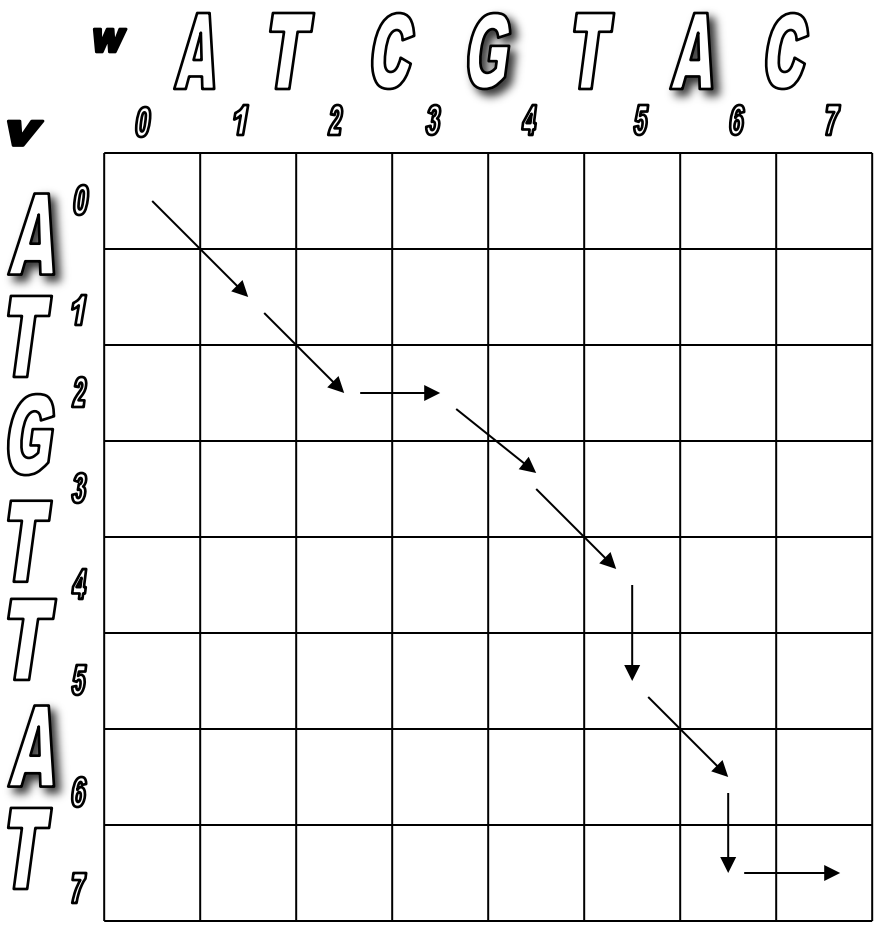
TGCATAT → (insert A at front)
ATGCATAT → (delete 6th T)
ATGCATA → (substitute G for 5th A)
ATGCGTA → (substitute C for 3rd G)
ATCCGAT (Done)

The alignment graph

– Every alignment path is from source to sink



Alignment as a path in the alignment graph

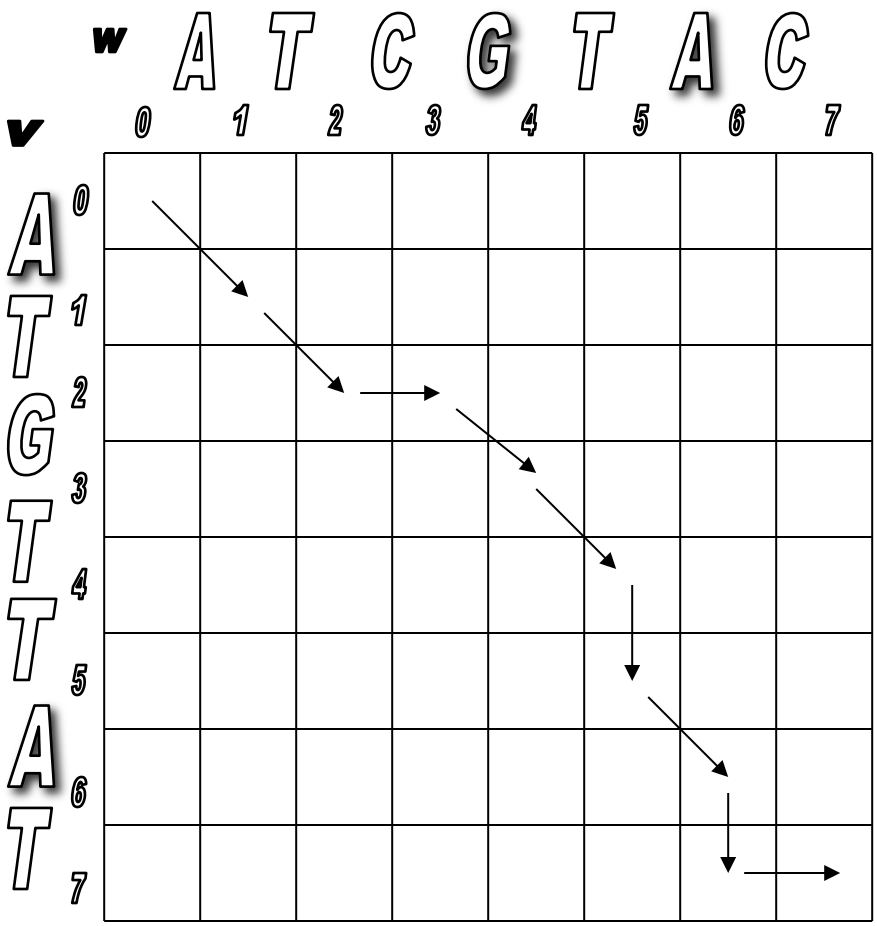


0	1	2	2	3	4	5	6	7	7
	A	T	_	G	T	T	A	T	_
	A	T	C	G	T	_	A	_	C
0	1	2	3	4	5	5	6	6	7

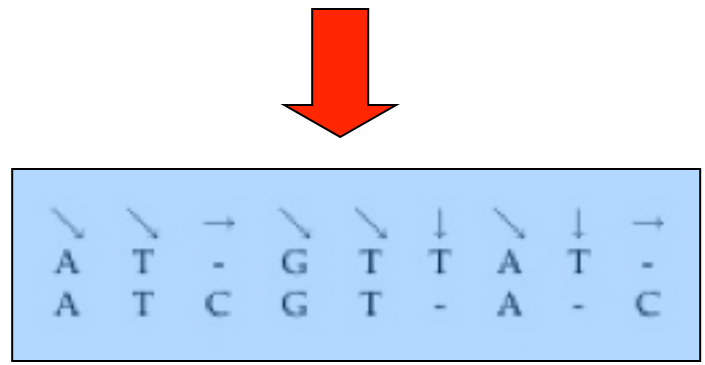
- Corresponding path -

(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



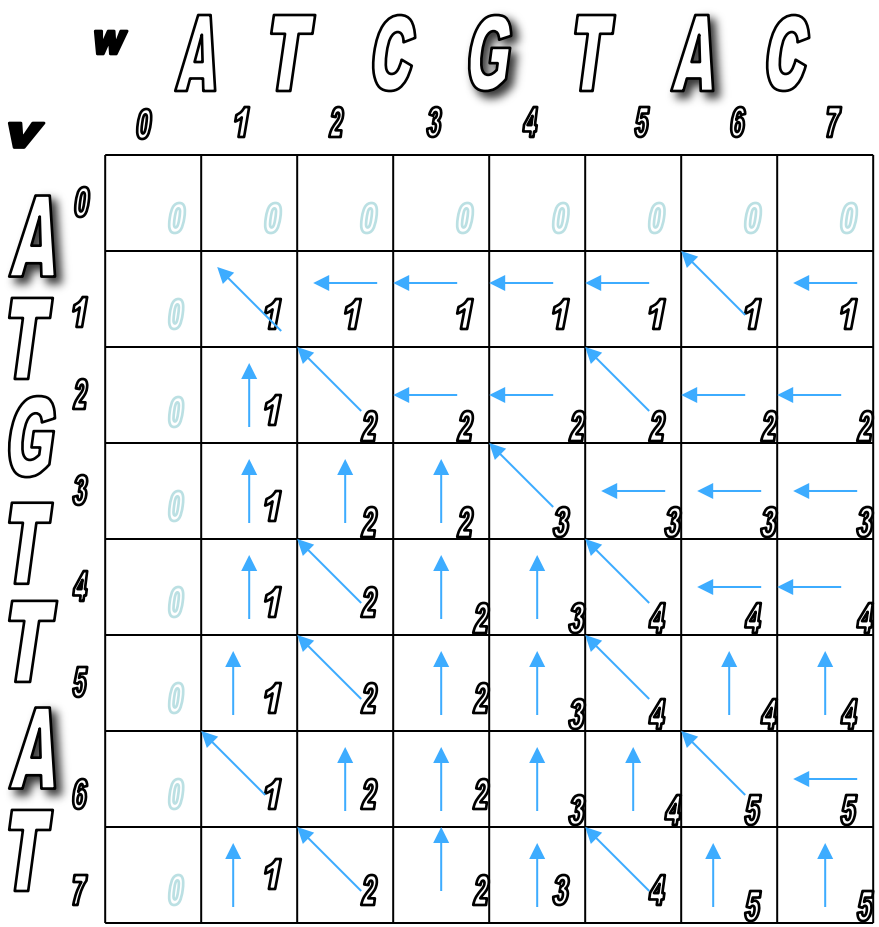
Every path in the edit graph corresponds to an alignment:



Alignment: Dynamic Programming

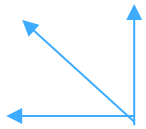
$$s_{i,j} = \max \begin{cases} s_{i-1,\, j-1} + 1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} & \downarrow \\ s_{i,\, j-1} & \rightarrow \end{cases}$$

Backtracking Example



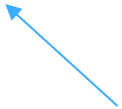
Continuing with the dynamic programming algorithm gives this result.

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

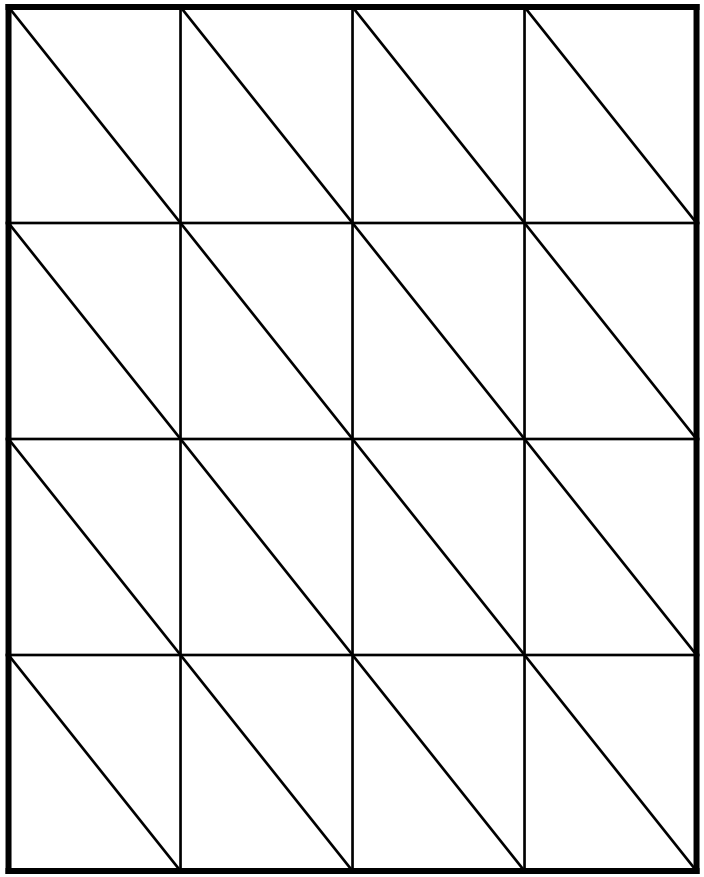
because you do not want to add too much of a penalty for extending the gap.

Can we still use the same recursion?

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

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

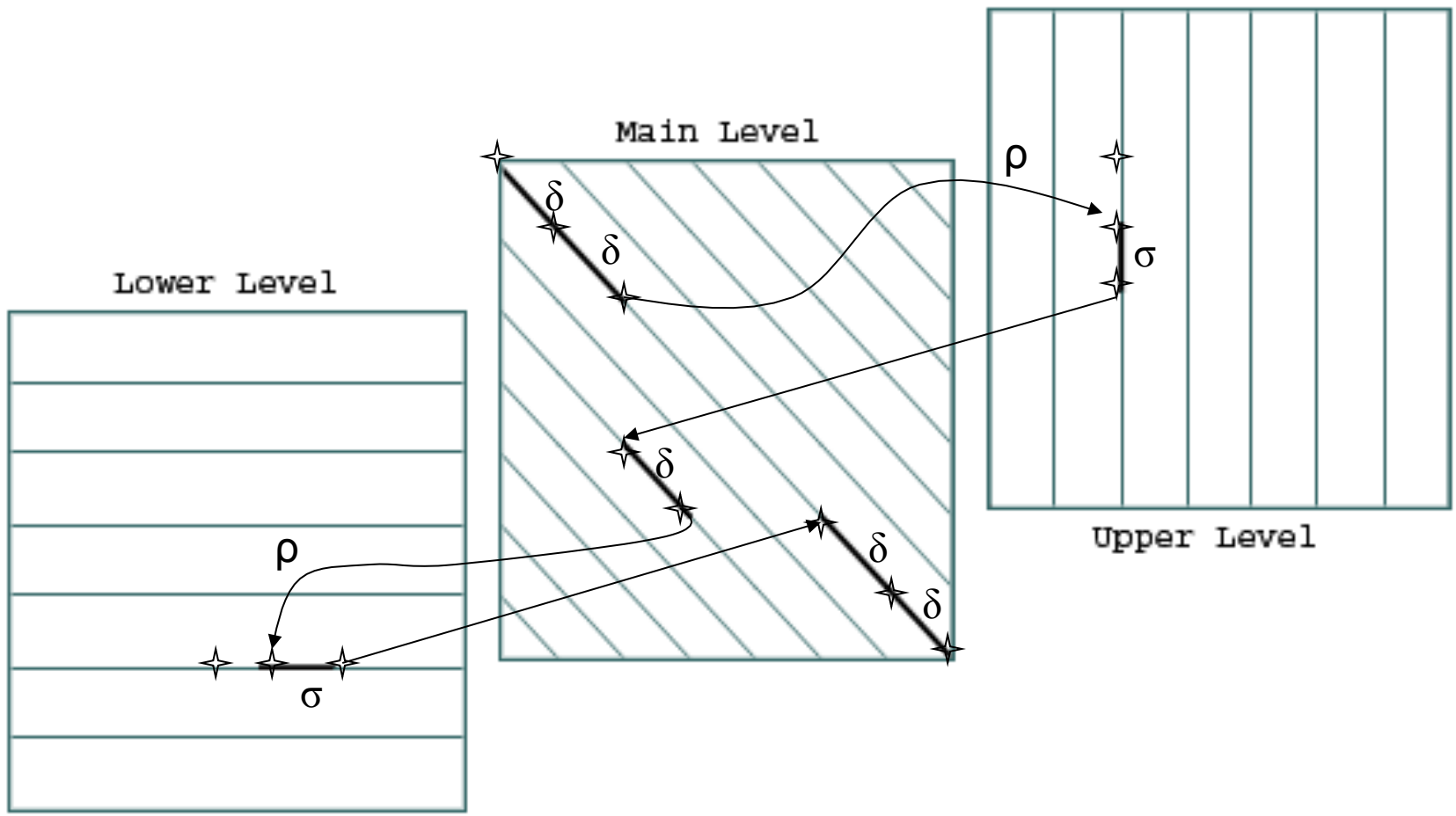
Affine gap penalties and alignment graph



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

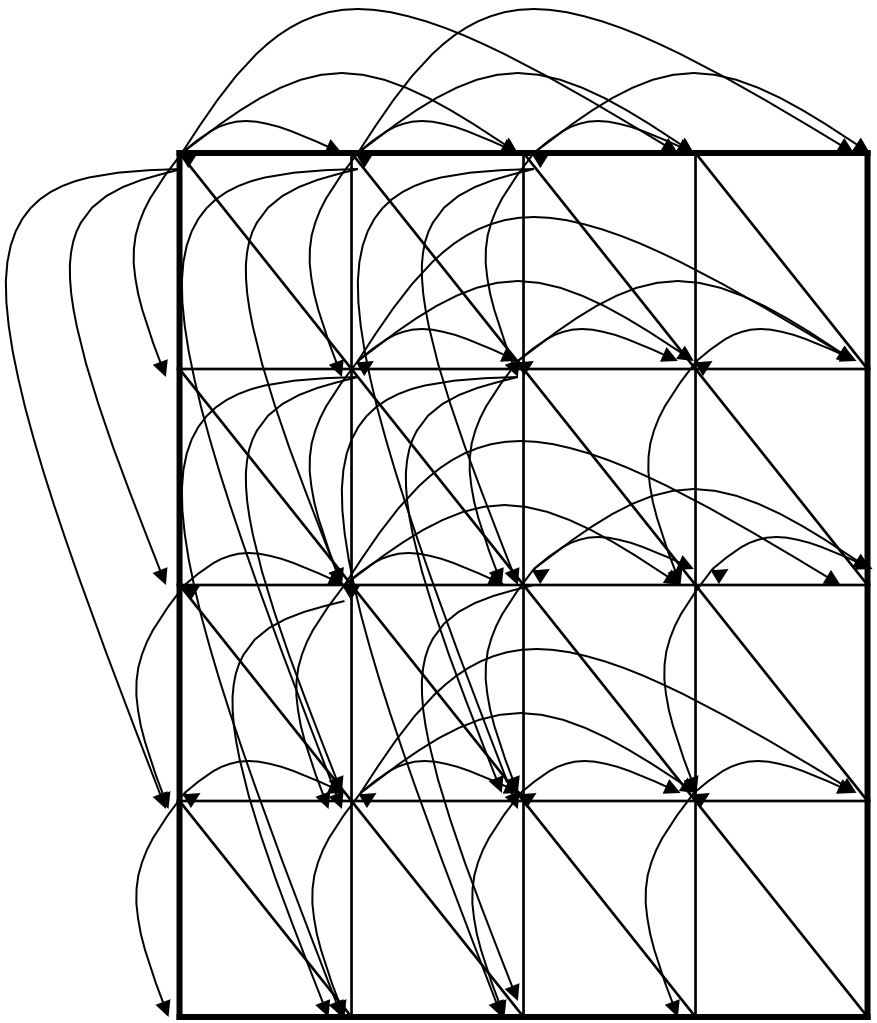
End deletion: from top

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 $O(n^3)$

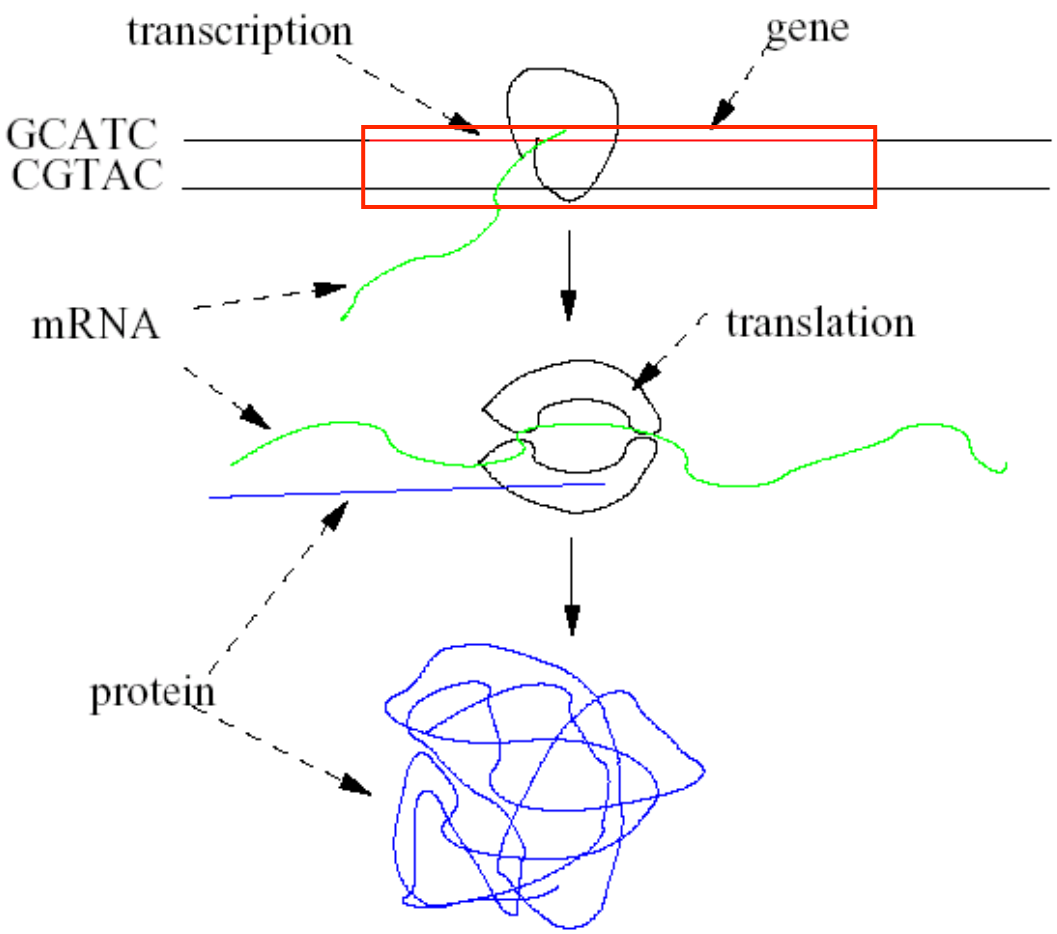
Take-home Problem: the number of alignments

- 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

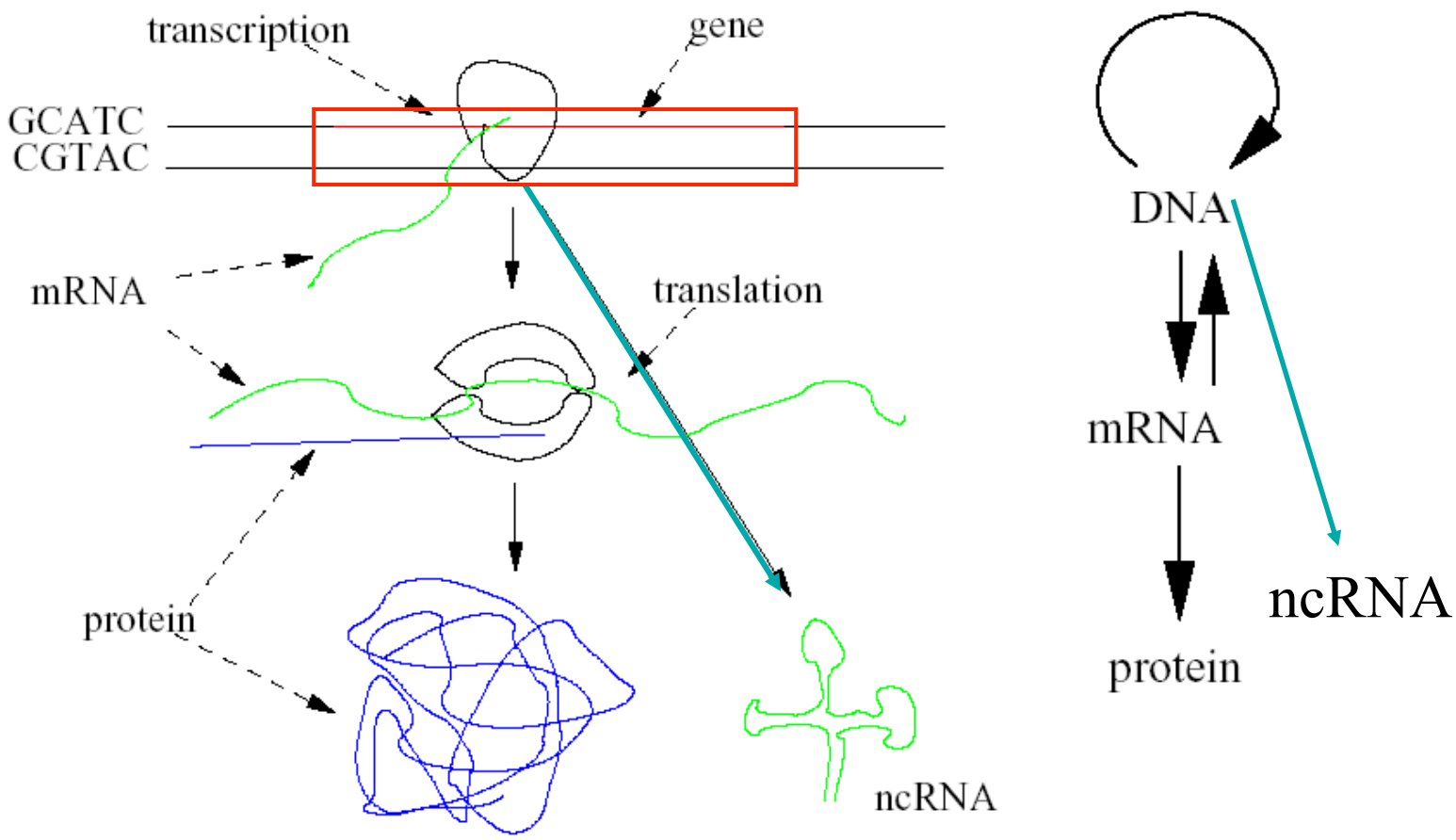


Central dogma cont'd

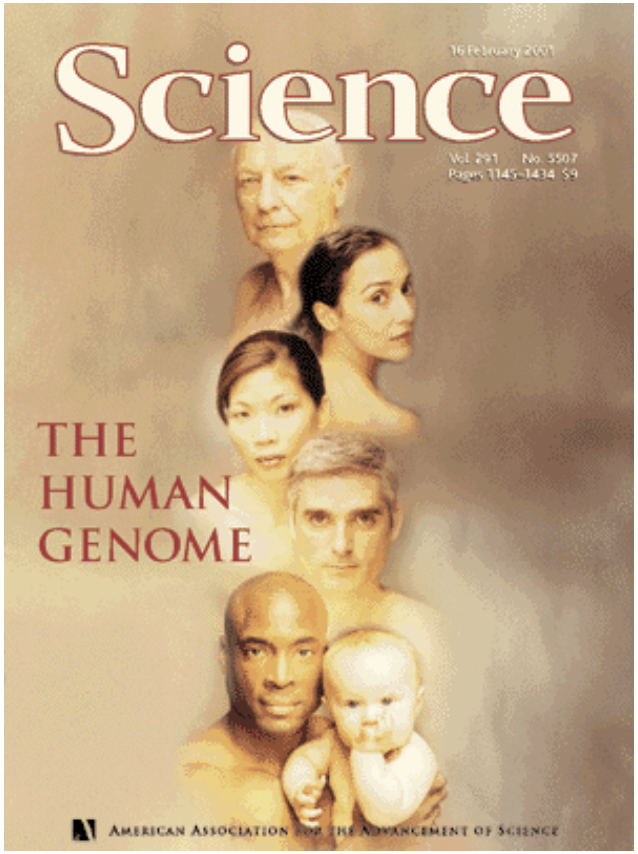
- Non-coding RNA (ncRNA)
  - RNA acting as functional molecule.
  - Not translated into protein.

- 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?

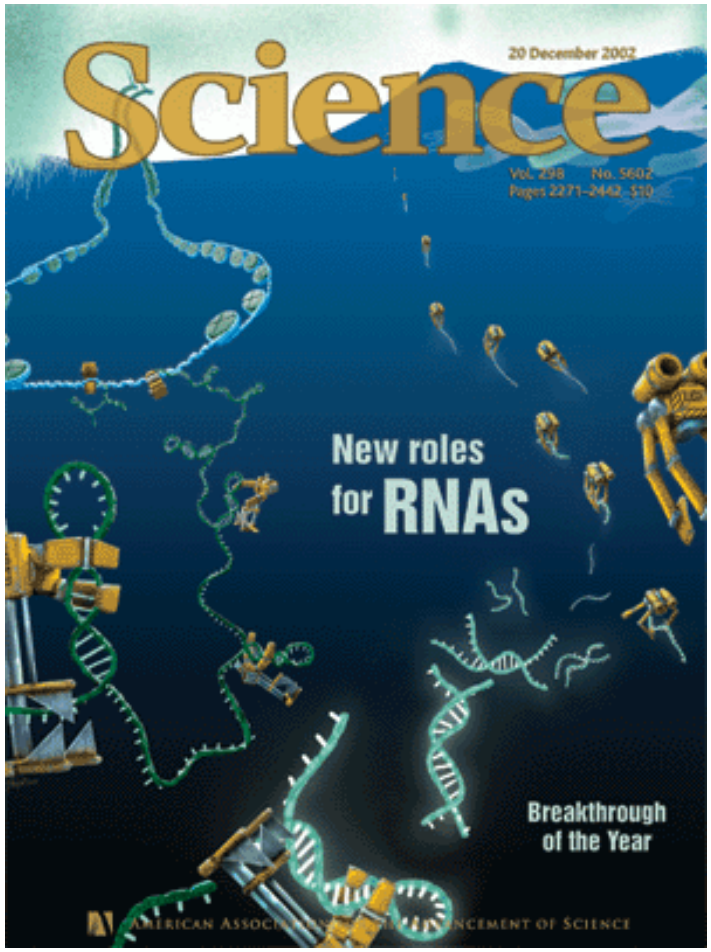
Question:

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

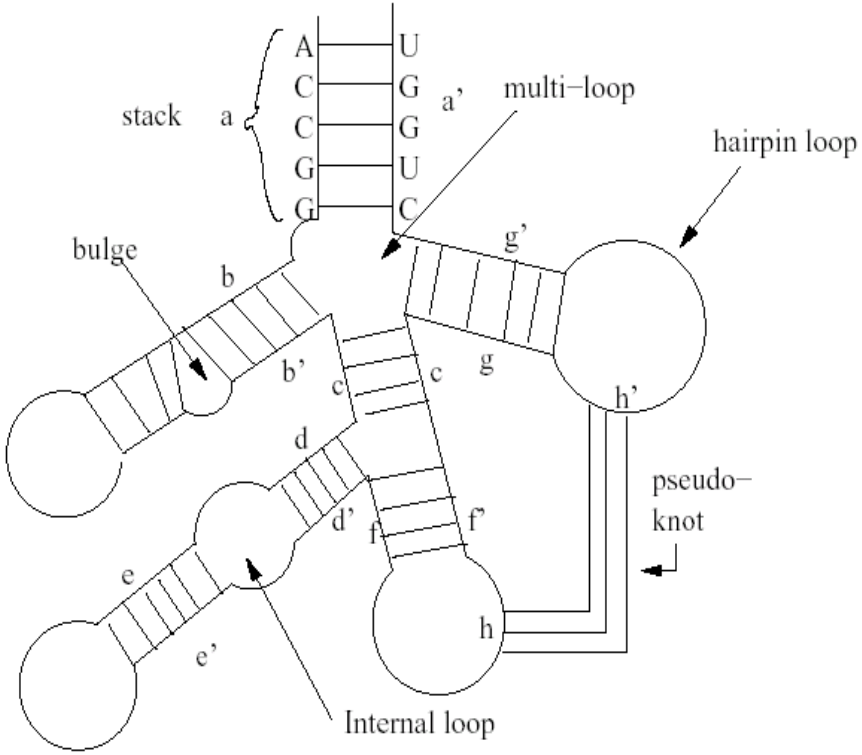
- ncRNA is not a random sequence.
- Most RNAs fold into particular base-paired secondary structure.

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

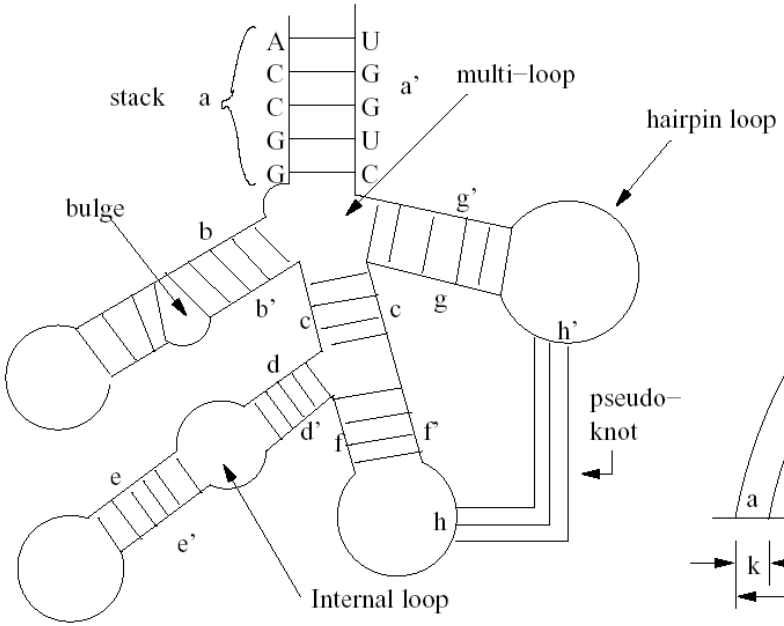
RNA secondary structure cont'd

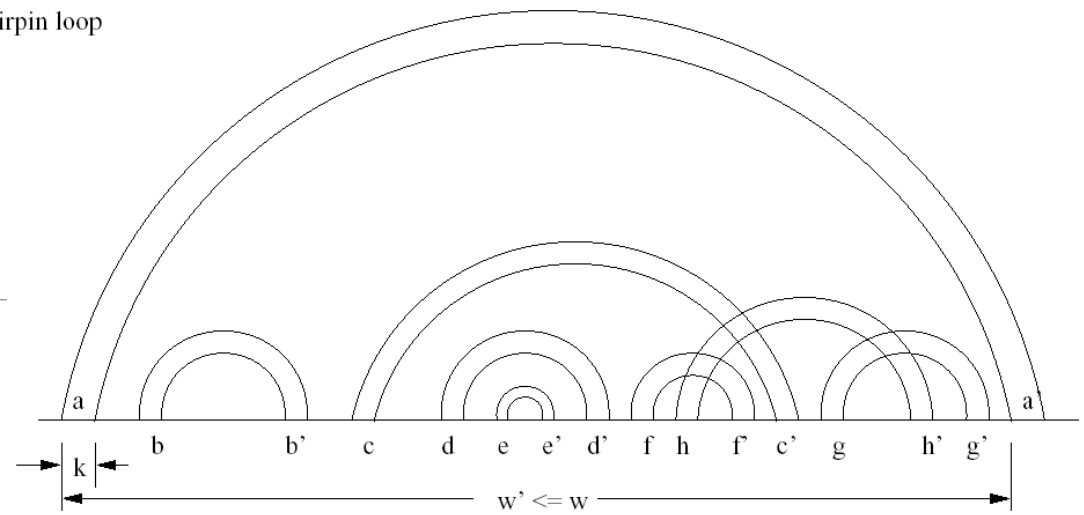
- Stacks: continuous nested basepairs. (energetically favorable)

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



RNA secondary structure cont'd

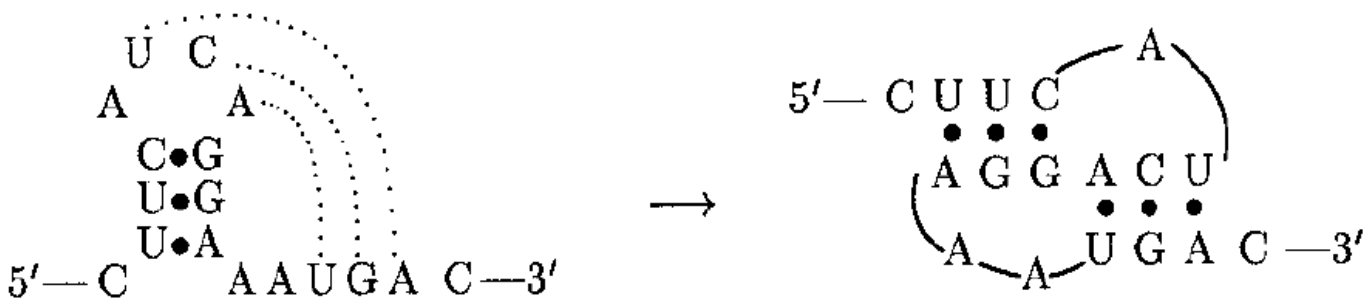




- Most basepairs are non-crossing basepairs.
  - Any two pairs (i, j) and (i', j') ➔ $i < i' < j' < j$ or $i' < i < j < j'$
- Pseudoknots are the crossing basepairs.

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.

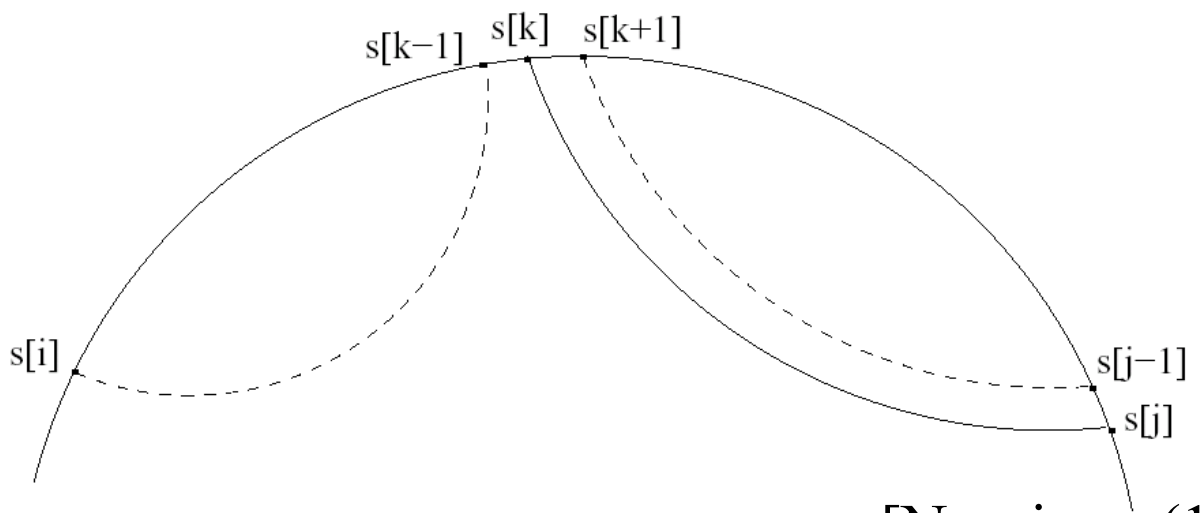
Base pair maximization problem

- 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)]

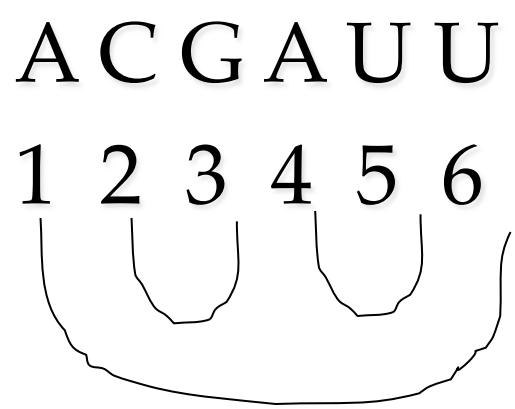
A dynamic programming solution

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

- *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	

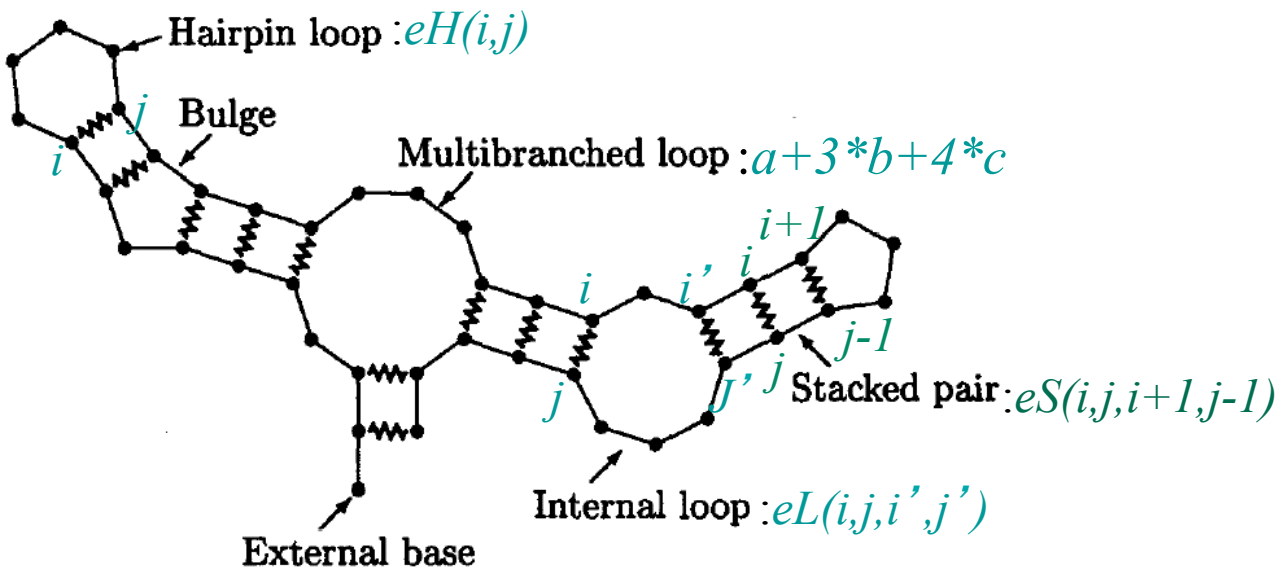


Zuker-Sankoff minimum energy model

- 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.

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

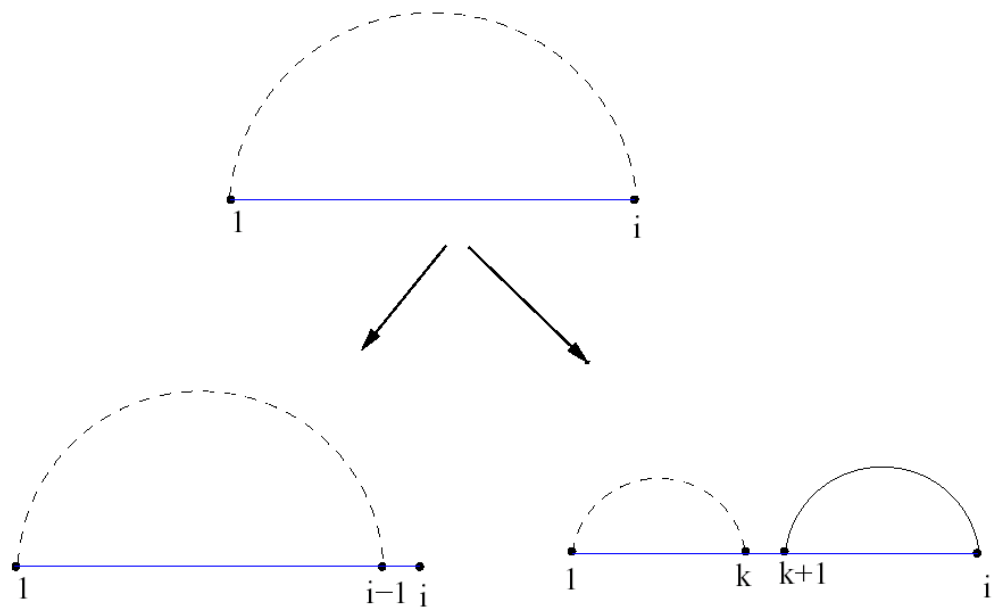
Recursive functions (Zuker)

- *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.

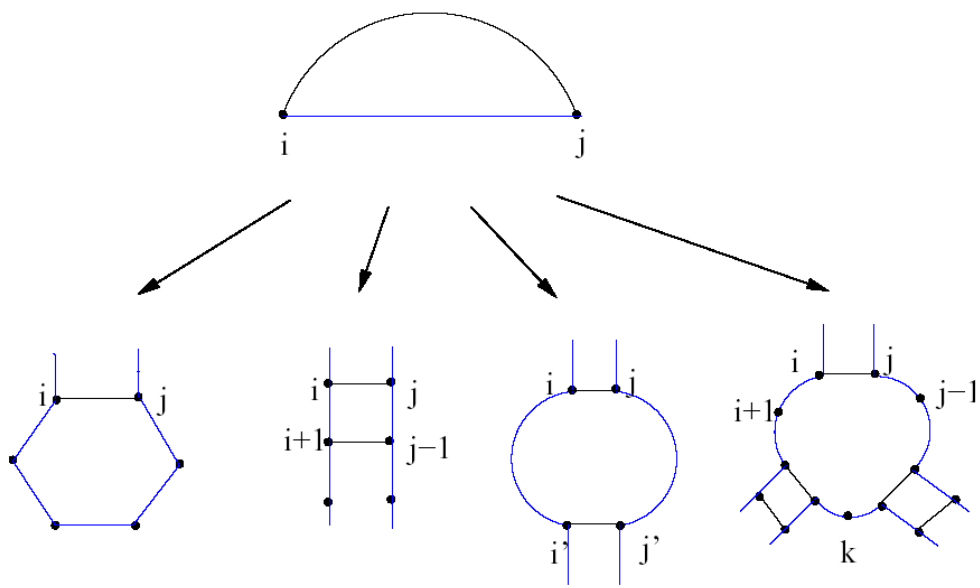
- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] that is part of multiloop.



A recursive solution

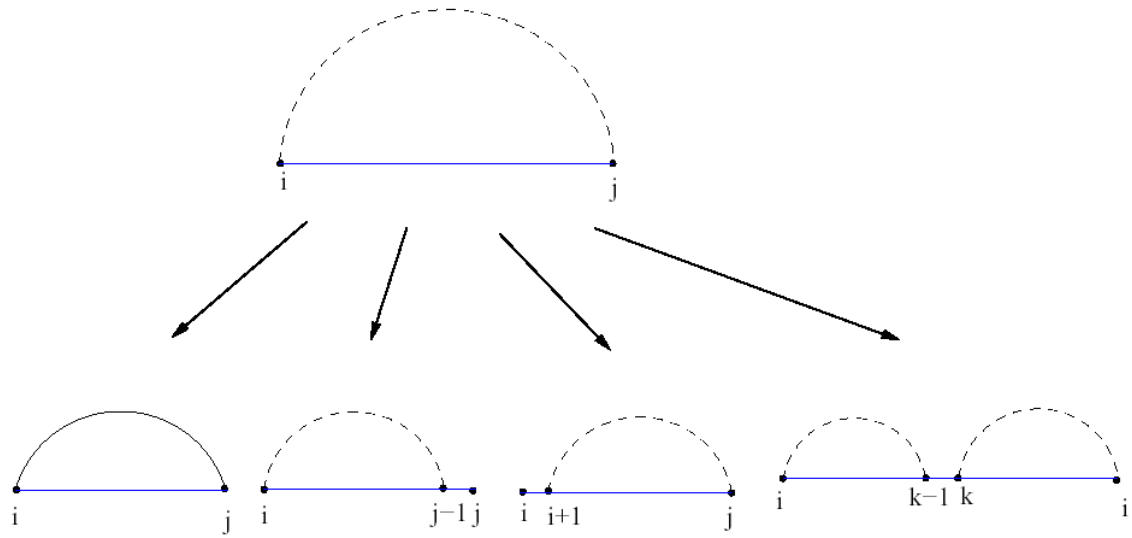
- *W*(*i*) holds the minimum energy of a structure on *s*[*1…i*].
- *V*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] with *s*[*i*] and *s*[*j*] forming a basepair.
- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*i…j*] that is part of multiloop.

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



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.

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

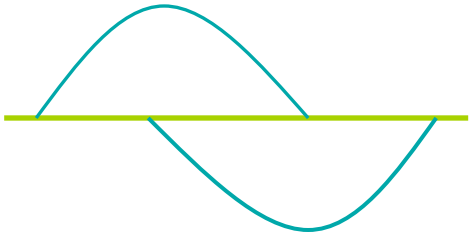
- *WM*(*i,j*) holds the minimum energy of a structure on *s*[*I…j*] that is part of multiloop.

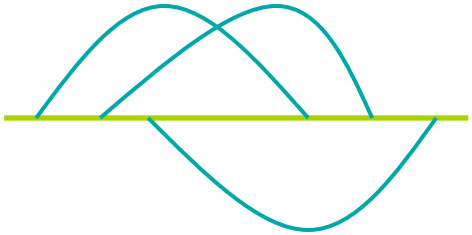
$$\begin{aligned} WM(i,j) = \min\{ & V(i,j) + b, \\ & WM(i,j-1) + c, \\ & WM(i+1,j) + c, \\ & \min_{i<k\le j}\{WM(i,k-1) + WM(k,j)\}\}, \end{aligned}$$

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

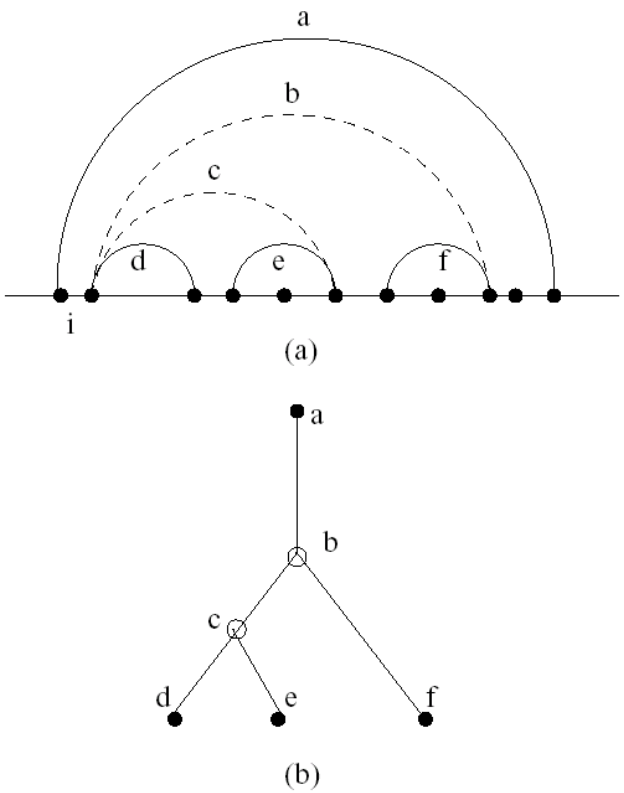
- Similar to Nussinov's base pair maximization problem's solution, the obvious dynamic programming solution is in $O(n^6)$ time.

- 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

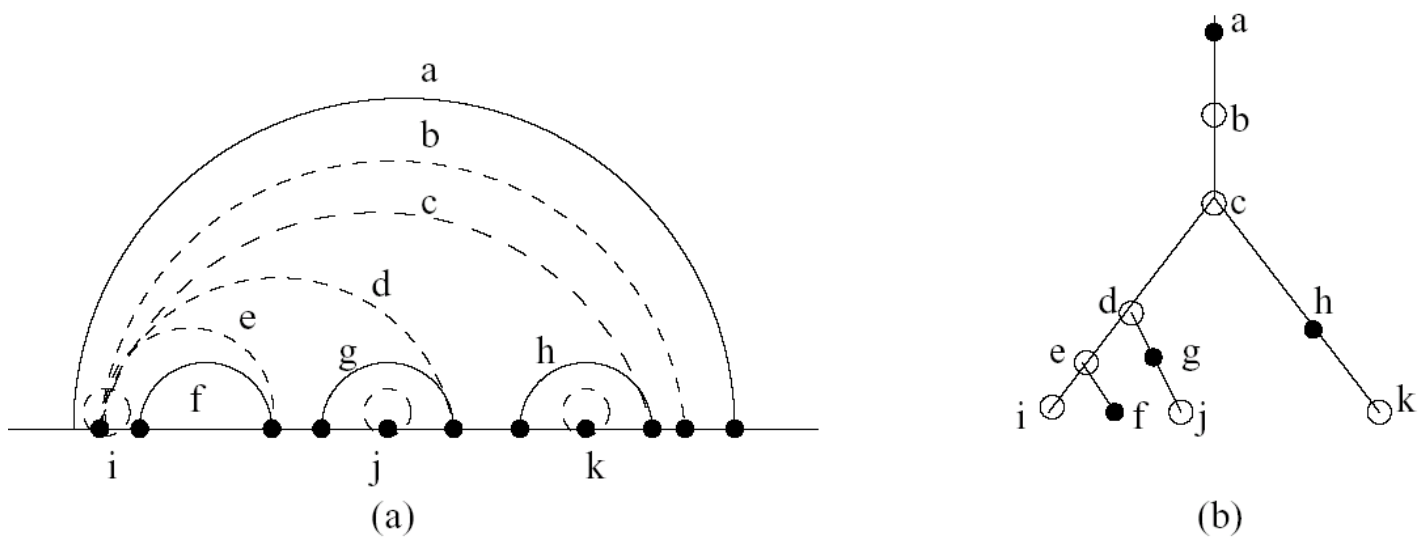
$$Align[i_1, j_1, i_2, j_2] = \max \begin{cases} Align[i_1, j_1, i_2, j_2], \\ \delta(i_1, j_1, i_2, j_2) + \gamma(s[i_1], t[i_2]) \\ +\gamma(s[j_1], t[j_2]) + Align[i_1+1, j_1-1, i_2+1, j_2-1] \end{cases}$$

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

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

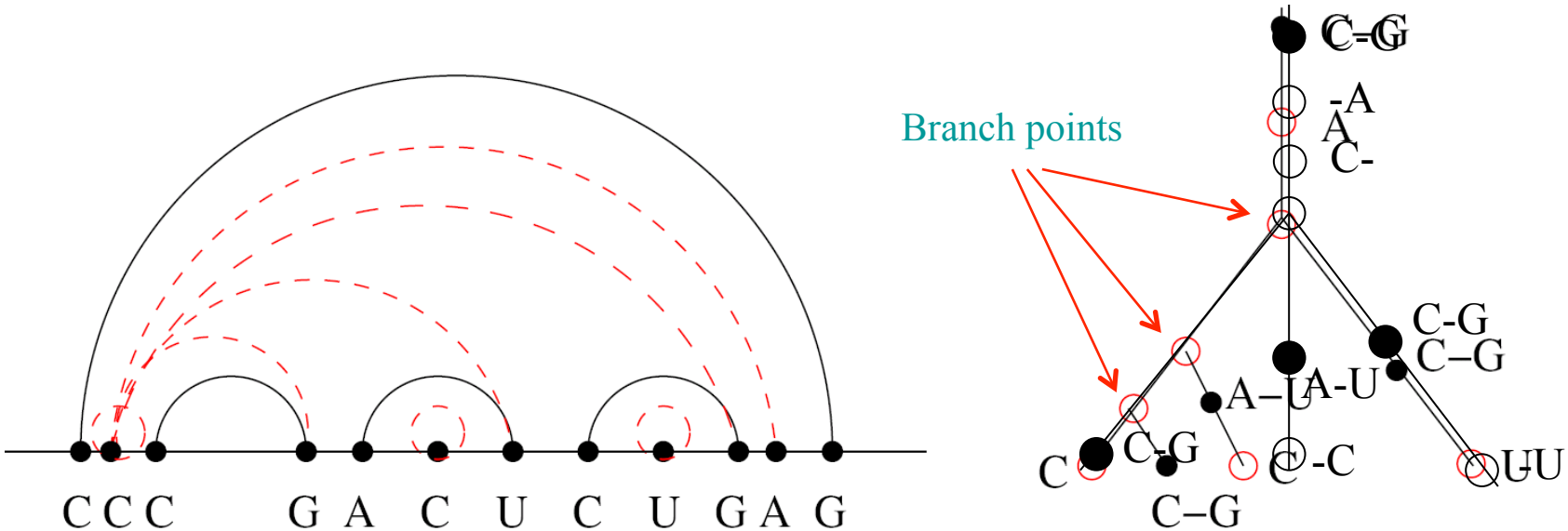
Extend the binary tree into the whole sequence s[1…n]

- 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.

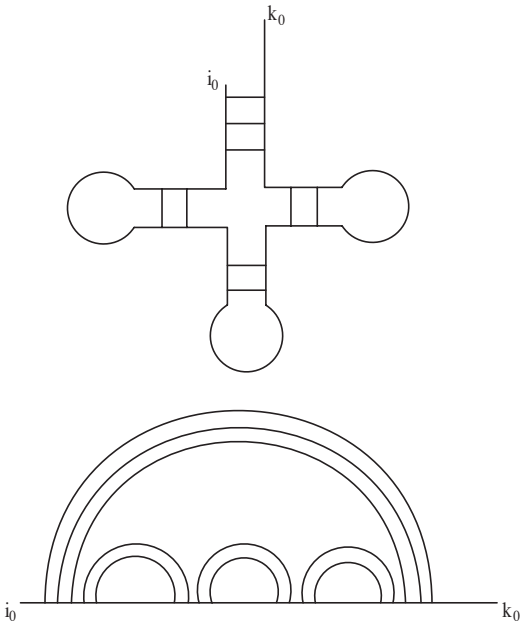
Structural Alignment of Pseudo-knotted RNA

RNA pseudo-knotted structures

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

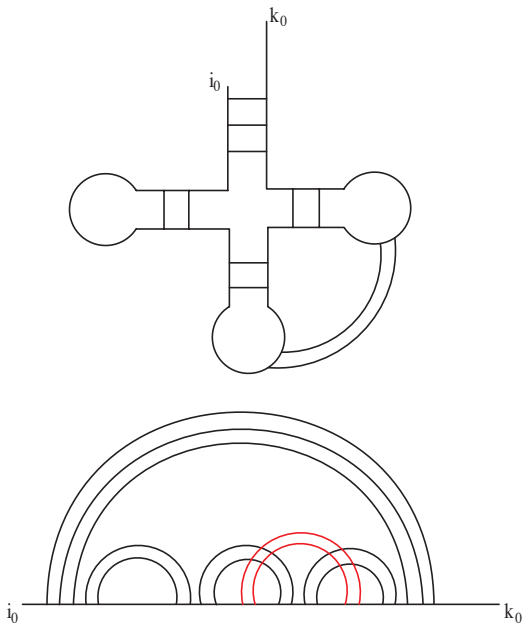
Regular structure:

All base pair are non-crossing.



Pseudo-knotted structure:

Some of the base pairs are crossing.



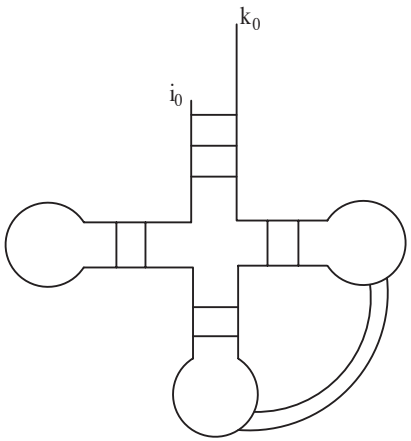
Solving problem for pseudo-knotted RNAs

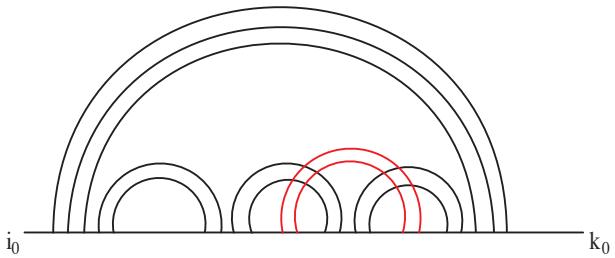
Dynamic programming technique used to align subsequences.

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

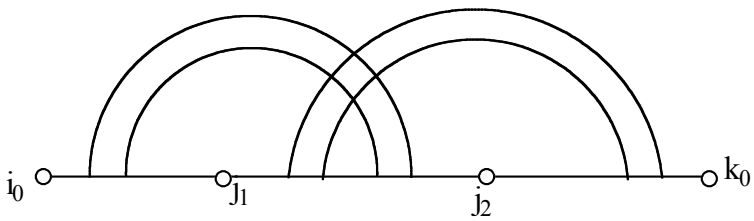


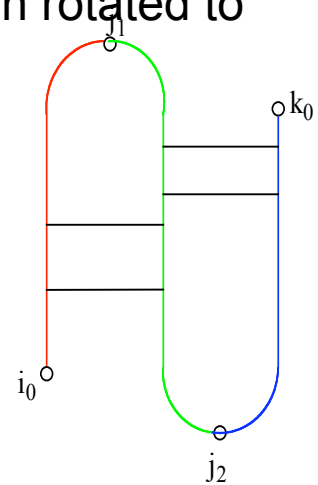


Definition: simple pseudo-knot

- 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.

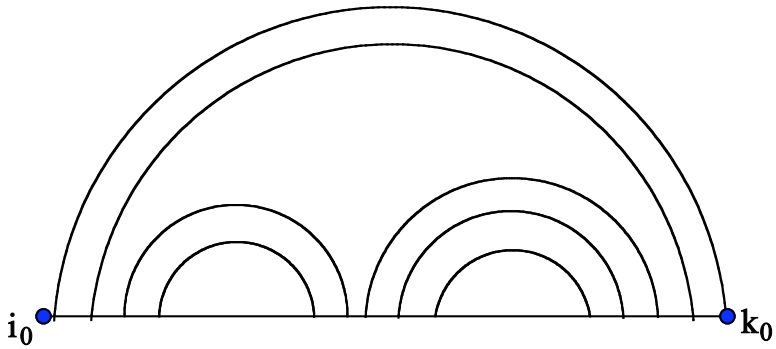




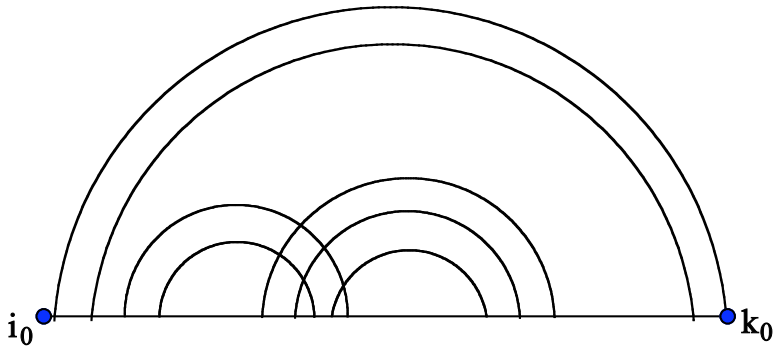
Sub-structure for a simple pseudo-knot

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.



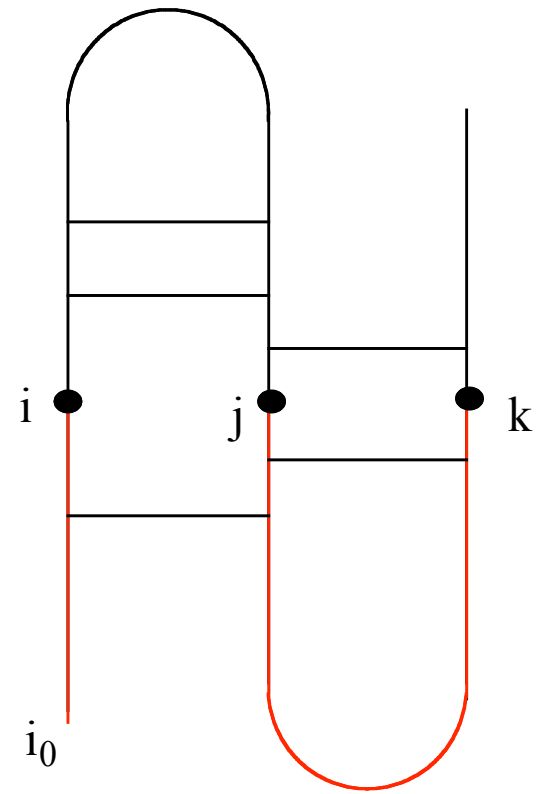
Sub-structure for a simple pseudo-knot

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

the union of two subintervals

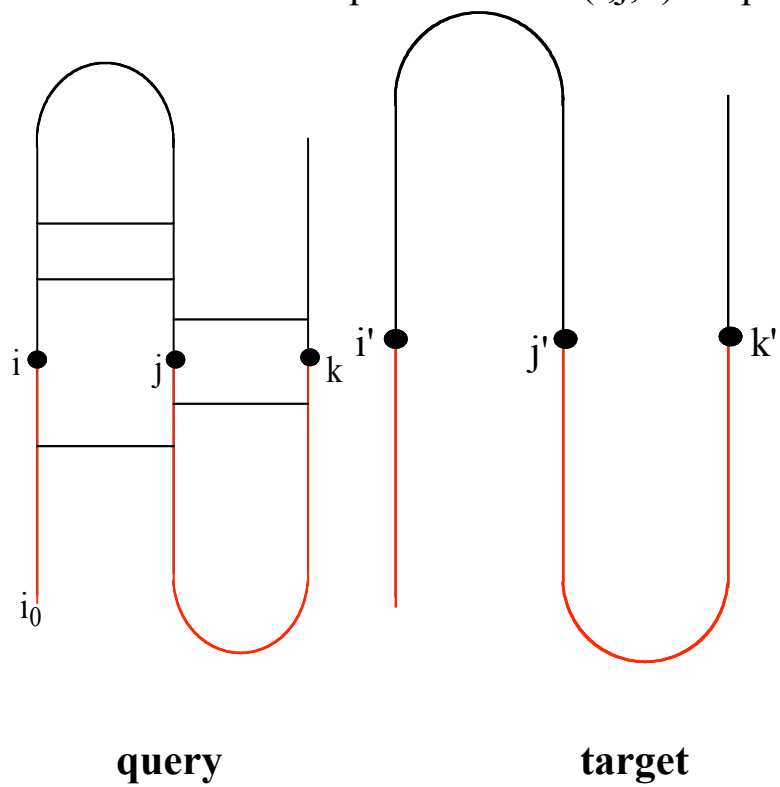
$P(i, j, k) = [i_0, i] \cup [j, k]$

frontier (i.j.k)



Naive approach

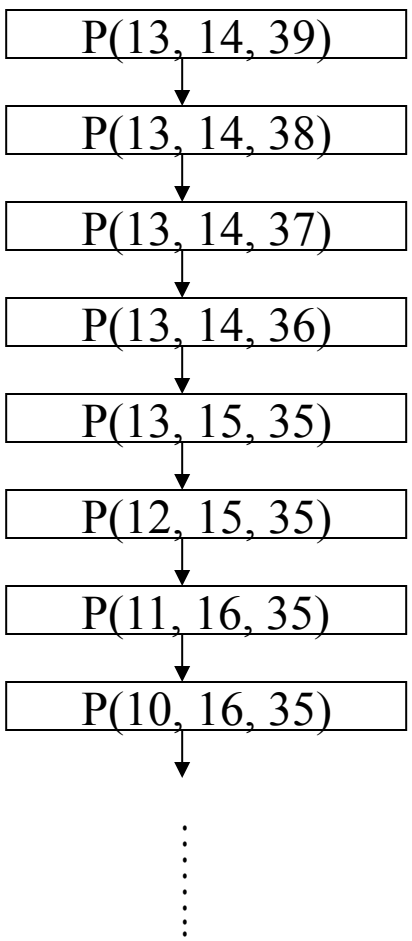
B[i, j, k, i', j', k']: Optimal score of the alignment of the sub-pseudoknot P'(i',j',k') in target to sub-pseudoknot P(i,j,k) in query.

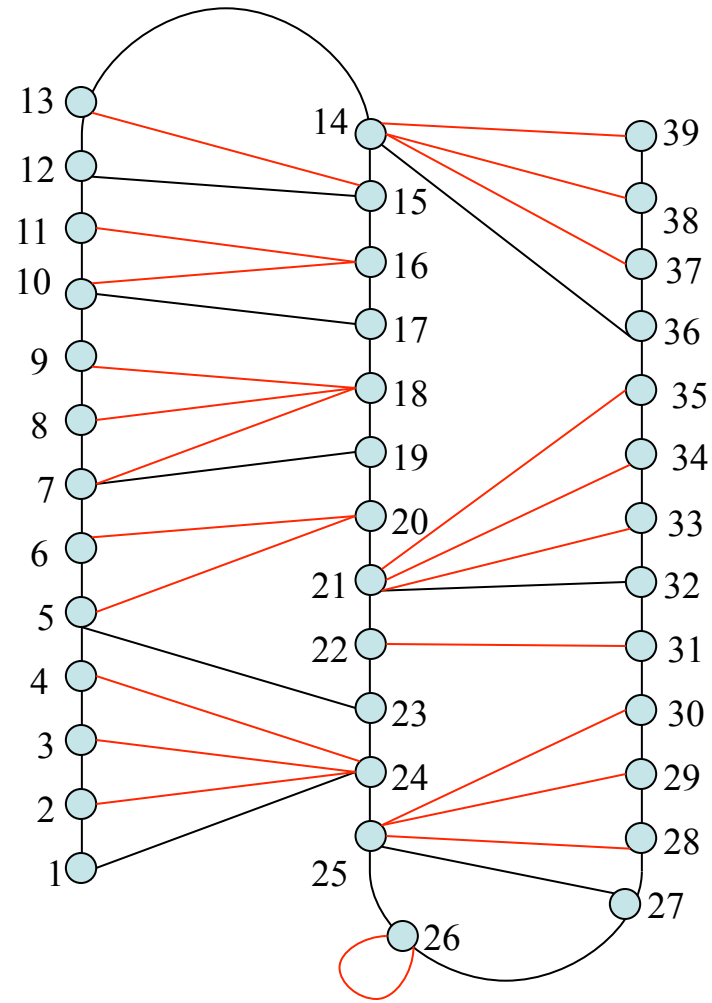


Compute B[i, j, k, i', j', k']
$\Rightarrow O(m^3n^3)$ scores.
(m:query, n:target)

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

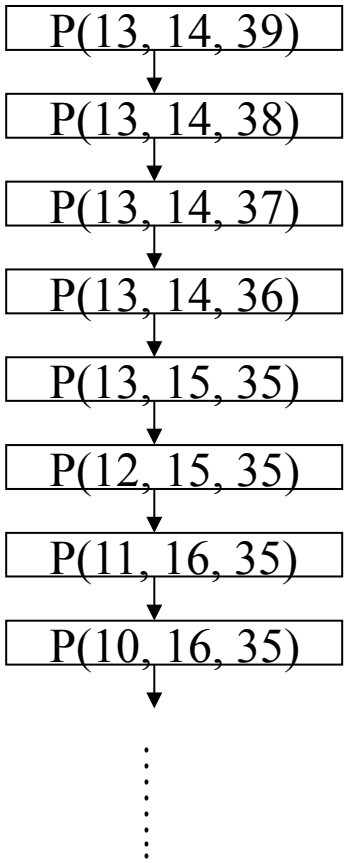


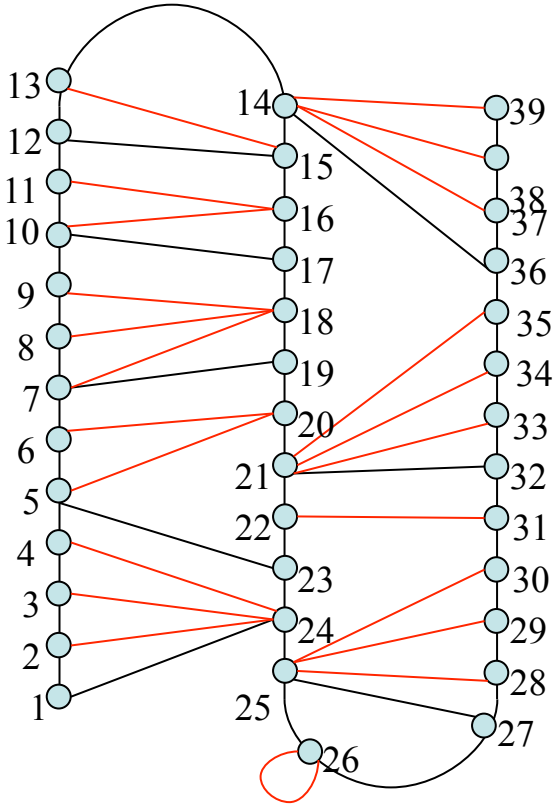


Why Chaining?

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

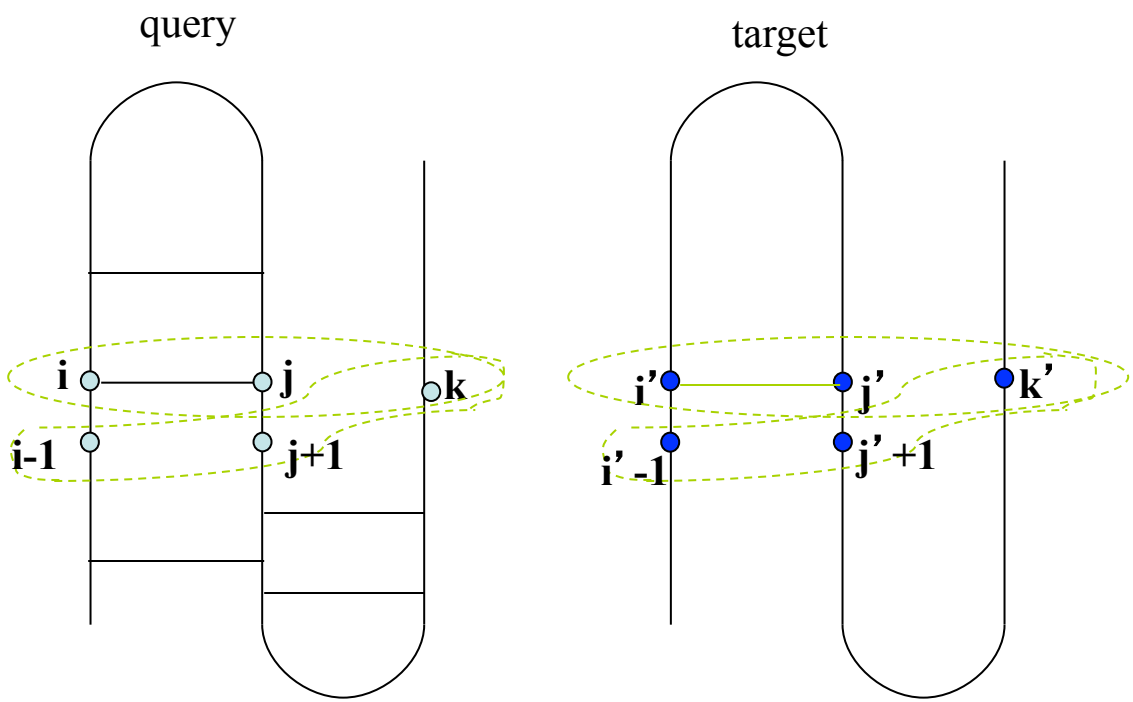
- compute B[i,j,k, i',j', k']
- => $O(mn^3)$ scores
- (m:query, n:target)





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

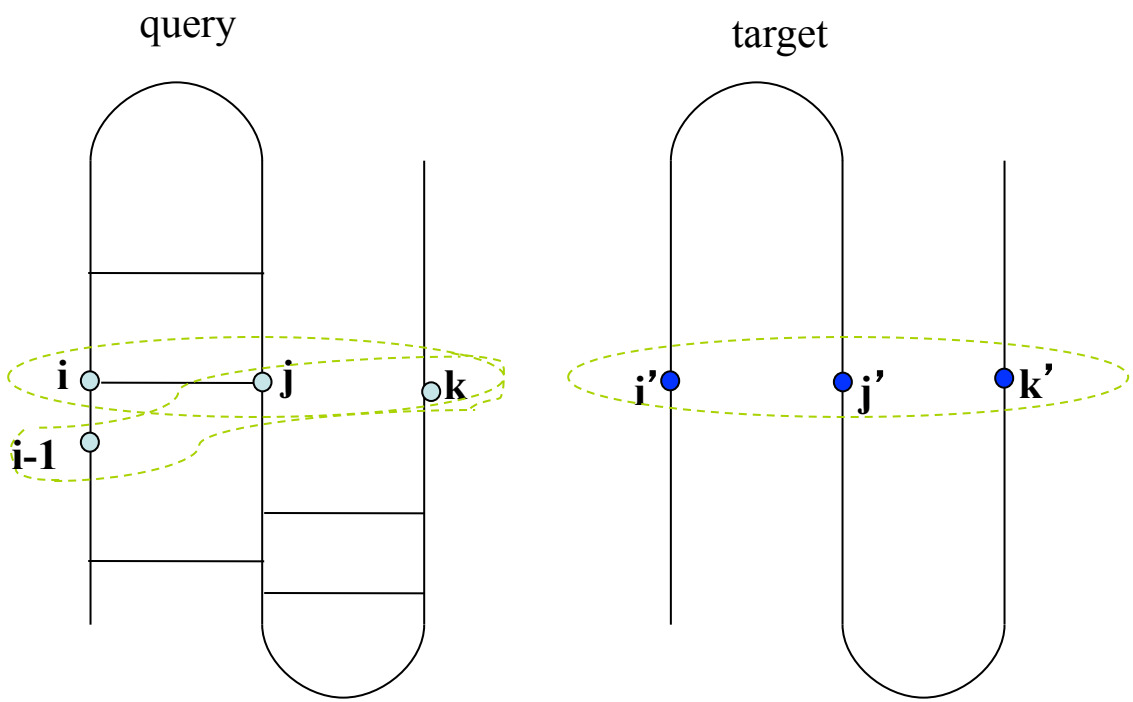
MATCH: (i,j) and (i’,j’) are corresponding pairs



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

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

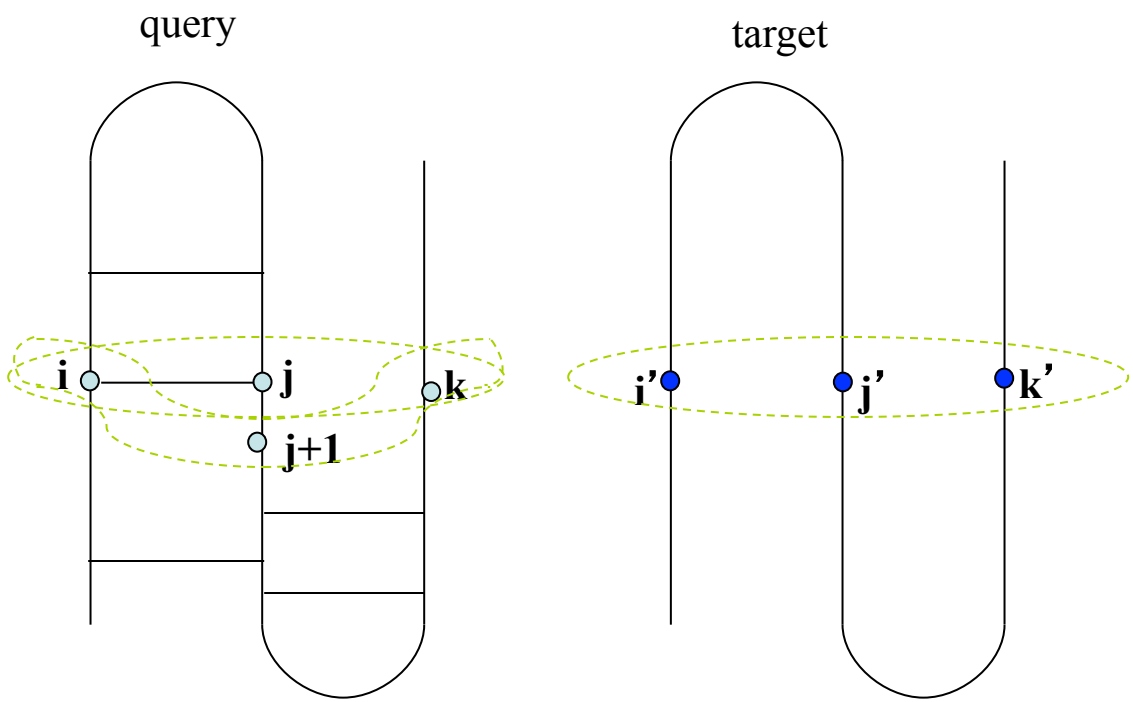
DELETION: i is deleted



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

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

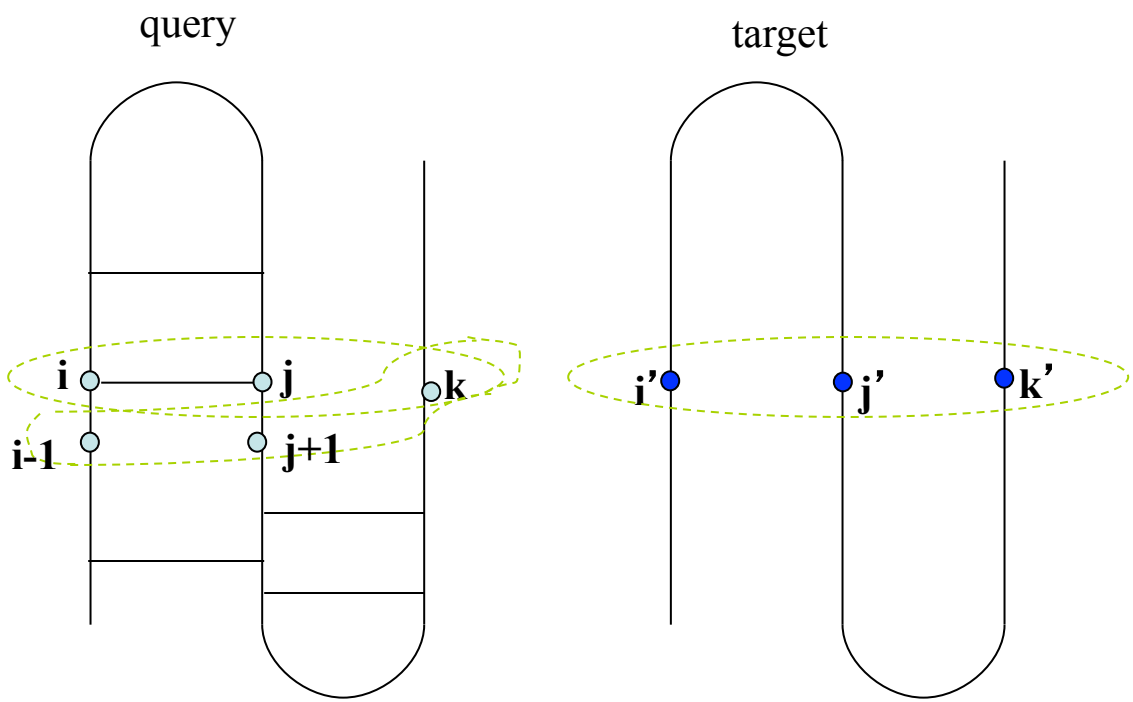
DELETION: j is deleted



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

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

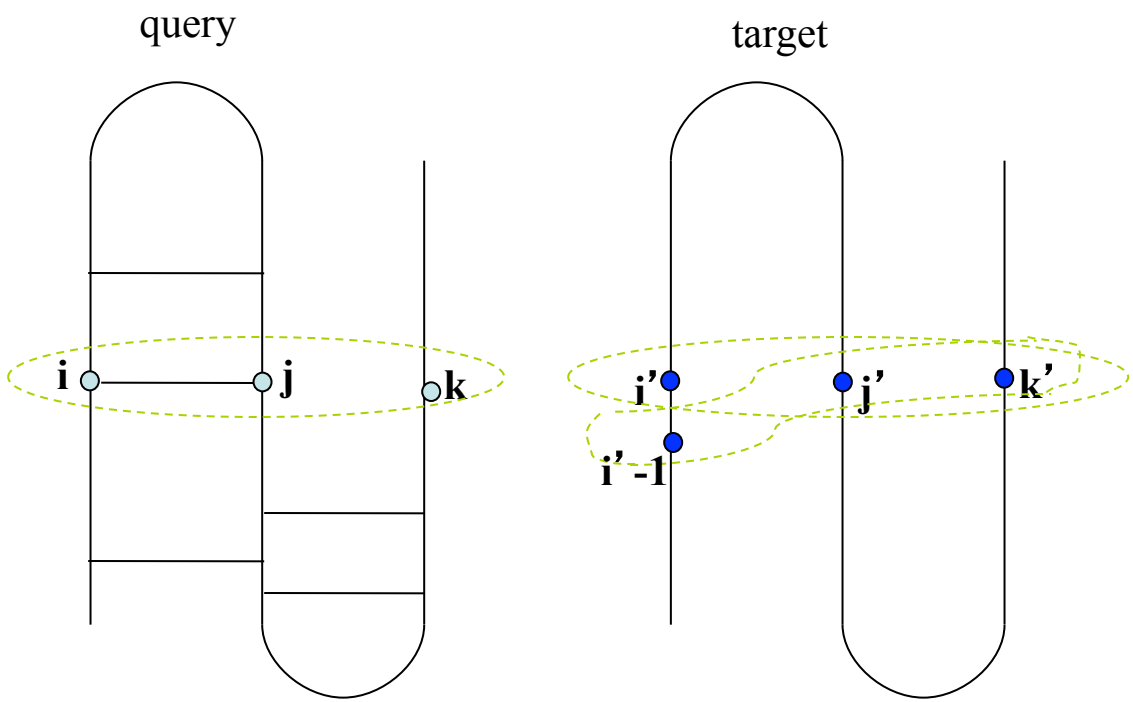
DELETION: i and j are deleted



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

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

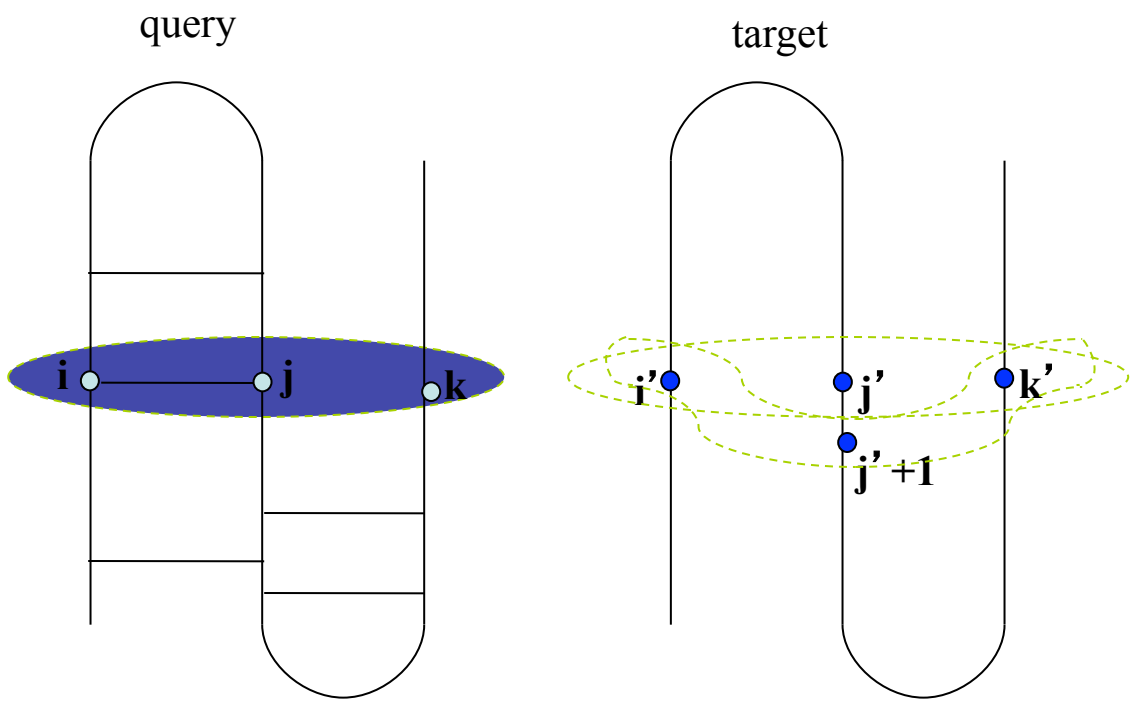
INSERTION: i' is inserted



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

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

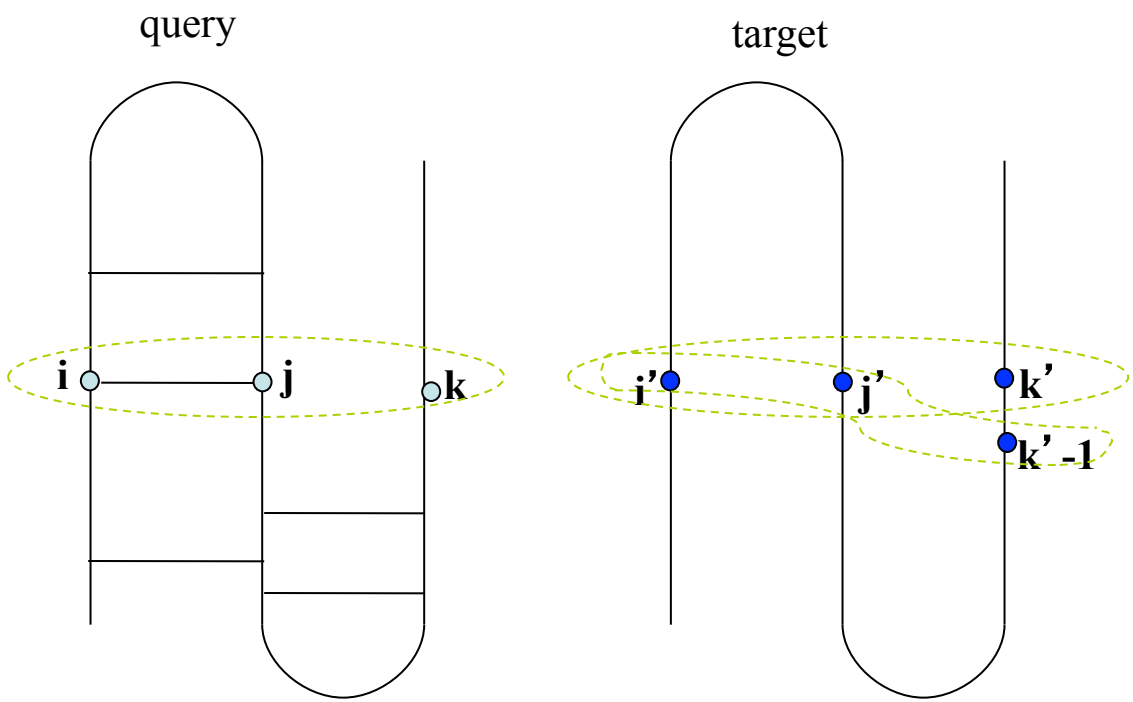
INSERTION: j' is inserted



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

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

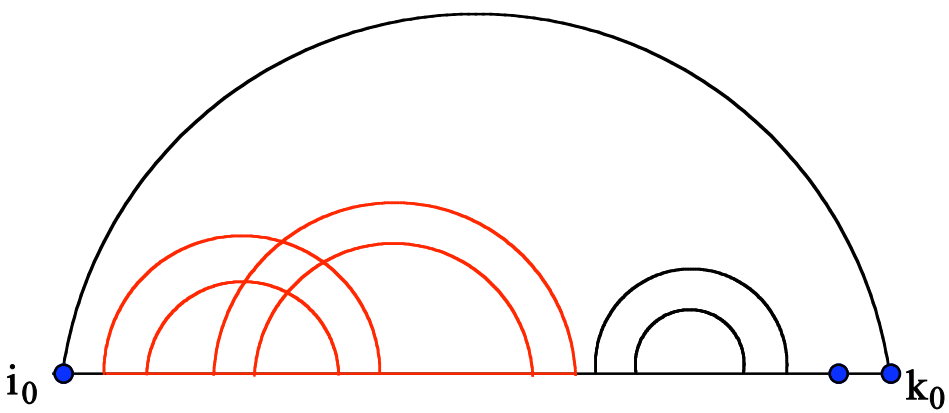
INSERTION: k' is inserted

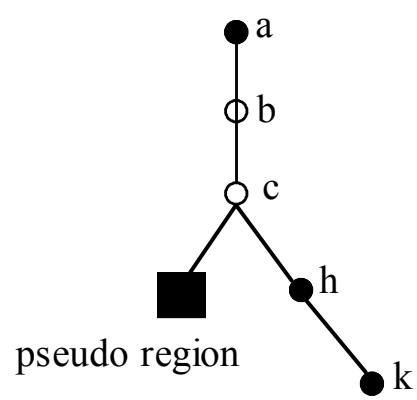


- B[i, j, k , i' , j' , k'] = max {MATCH, INSERT, DELETE}

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

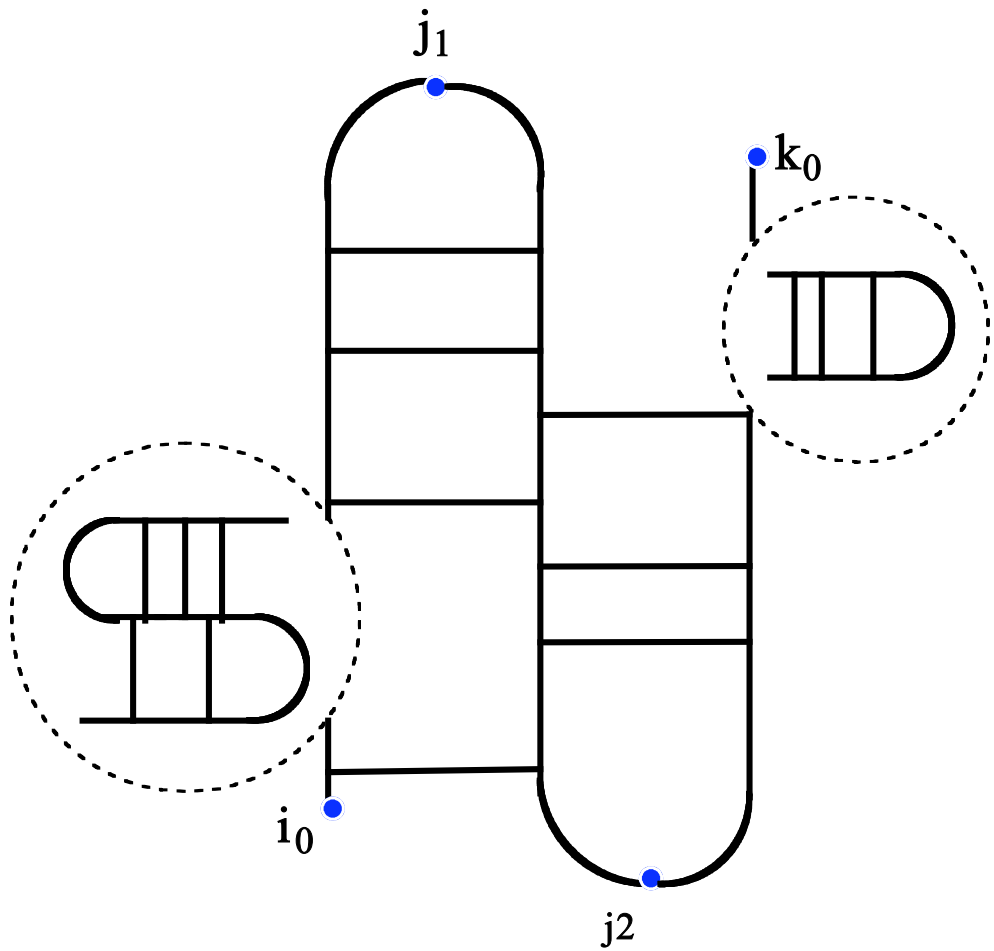
Use a binary tree to represent RNA
Solid circular nodes correspond to the actual base pairs.
Empty circular nodes correspond to unpaired bases.
Rectangular node correspond to sub-tree representing pseudo-knotted region





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

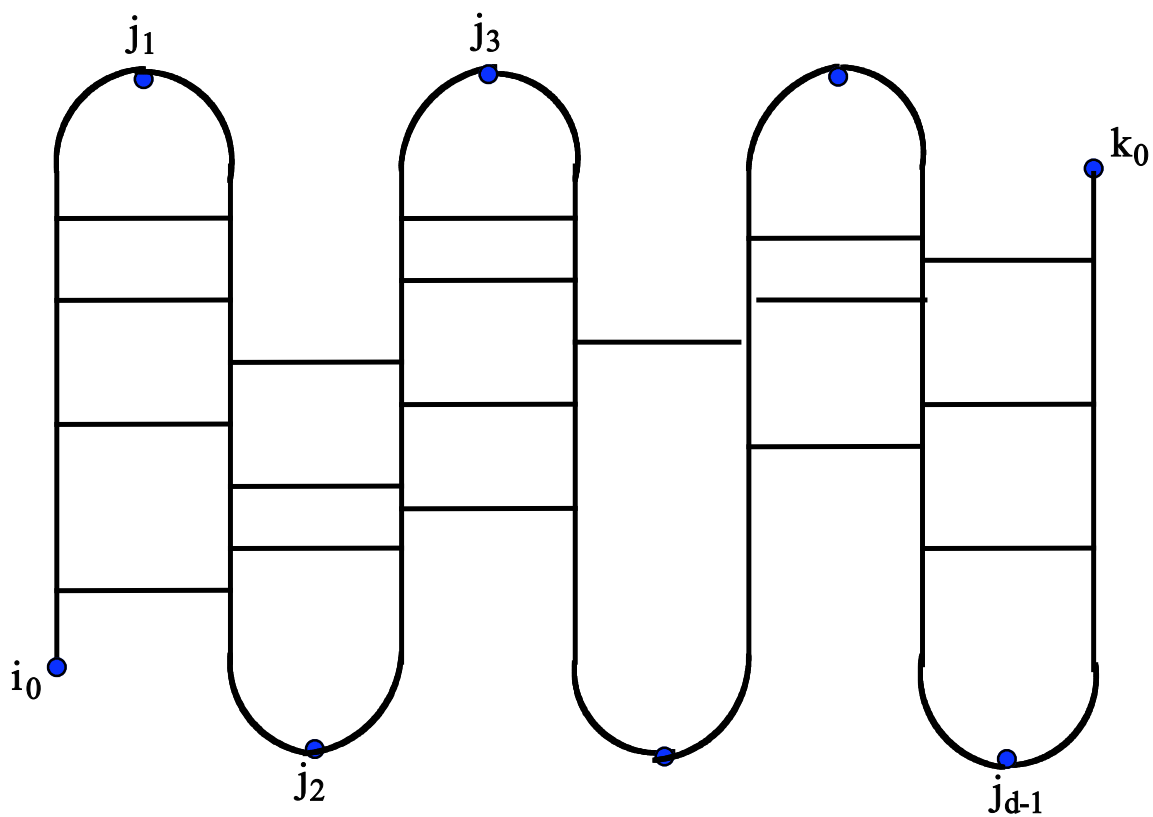
- S in S
- R in S



Which structures can we handle?

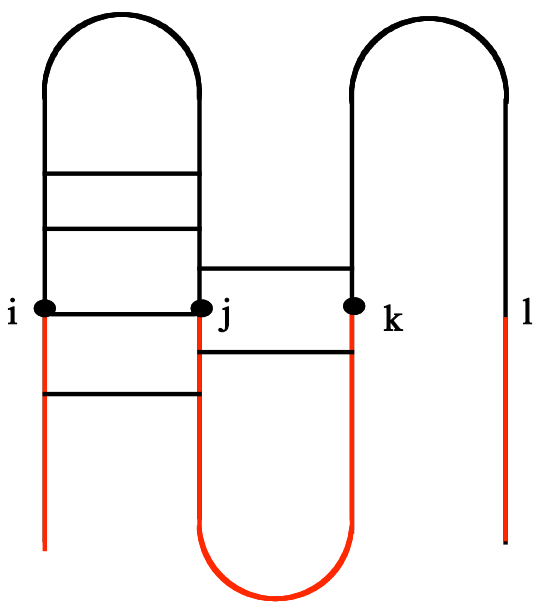
- Time complexity increases with the recursion depth of the pseudo-knotted region!
-
- R: regular structure
- S: simple pseudo-knot
-
- R: $O(mn^3)$
- S: $O(mn^4)$
- S in R: $O(mn^4)$
- R in S: $O(mn^5)$
- R in S in R: $O(mn^5)$ = S in S in R: $O(mn^5)$
- R in S in S in R = $O(mn^6)$
- ……..

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

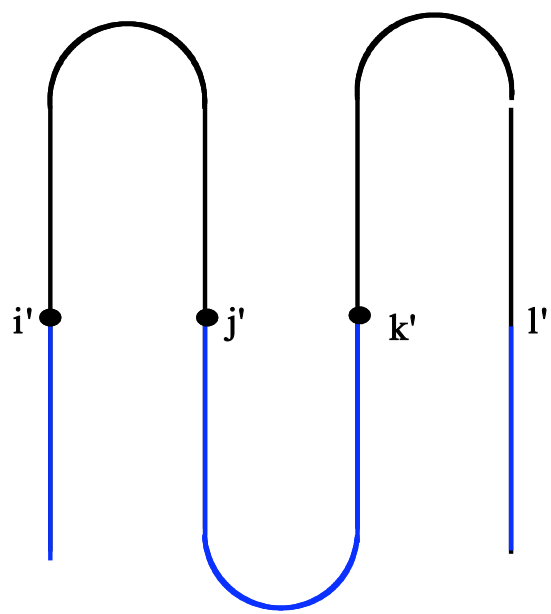


Can we handle simple pseudo-knots with higher degree: standard pseudoknots?

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



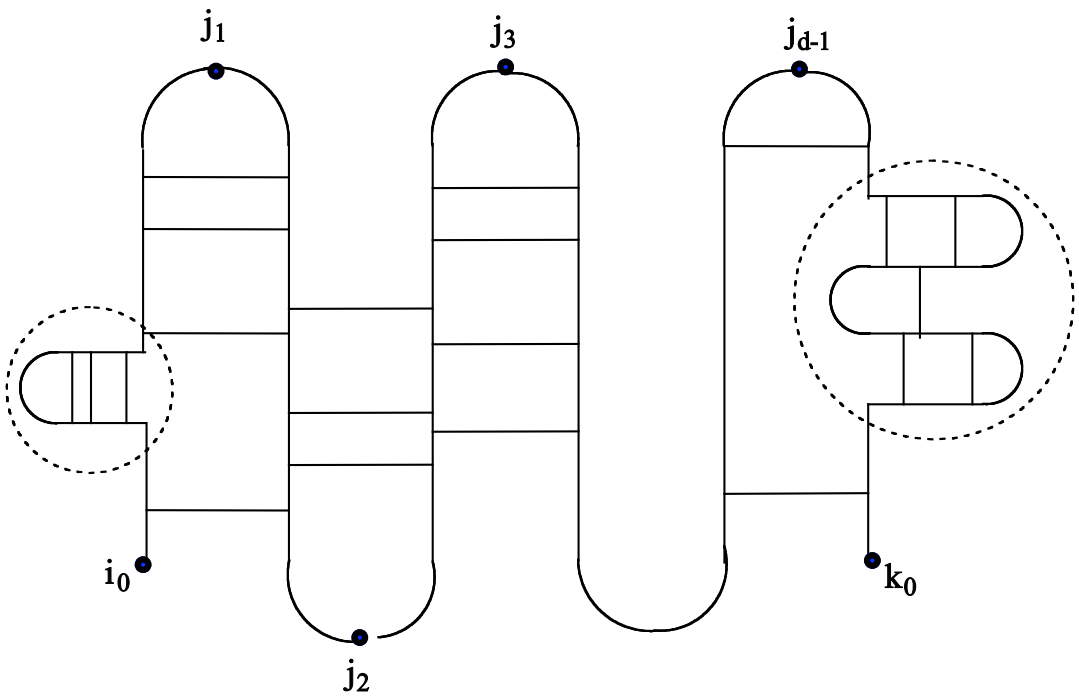
query



target

Can we handle recursive standard pseudoknots?

Yes! Same reasoning with recursive simple pseudoknots.

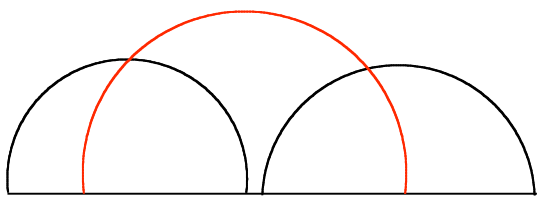


What is left? What can we NOT handle?

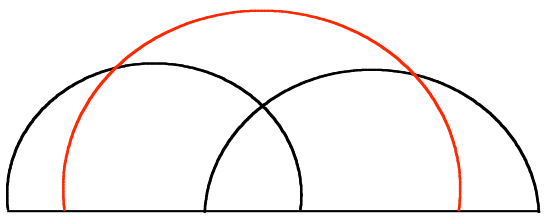
?We can handle the class of pseudoknots defined by Akutsu which is the second largest class currently defined. We can additionally handle standard and recursive standard pseudoknots which are defined by us.

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

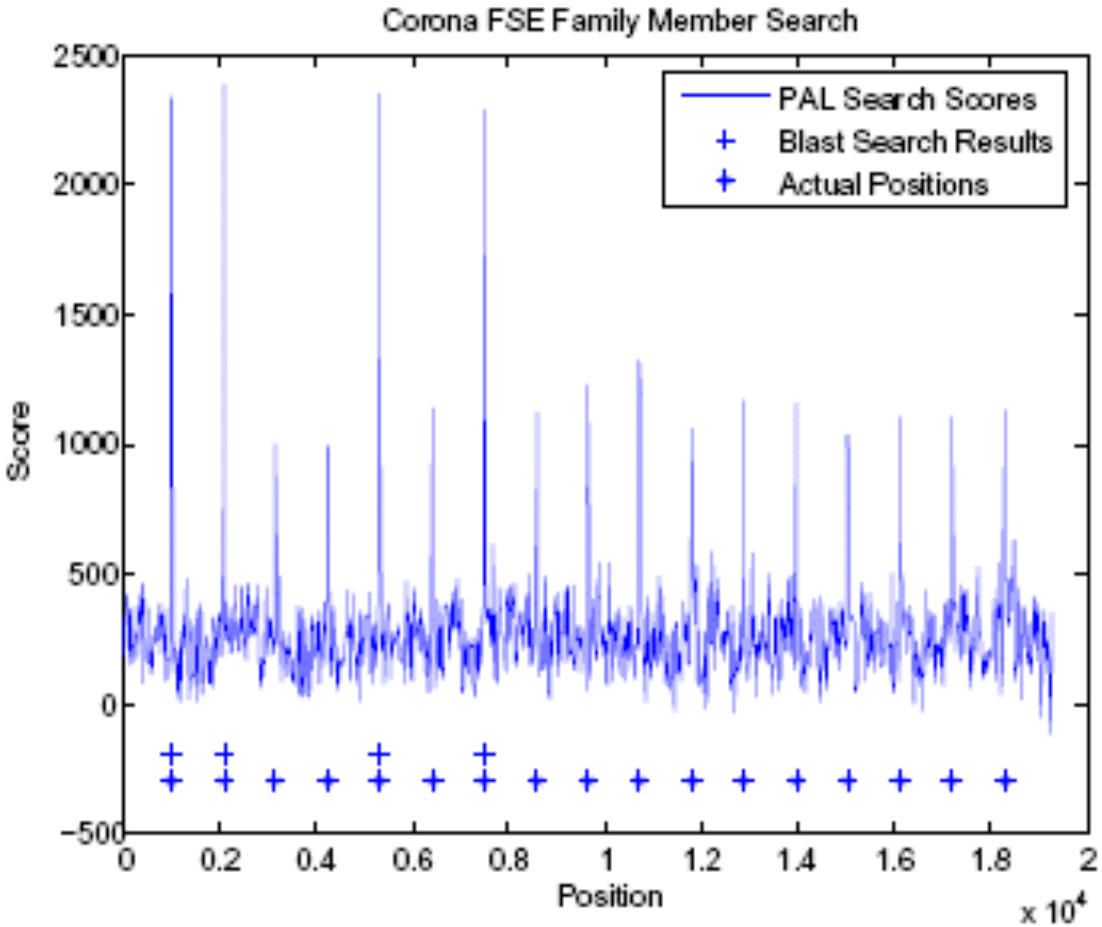
RNA Family	Specificity (Mean)	Sensitivity (Mean)
UPSK	1	1
Antizyme	0.99	0.99
Parecho	0.95	0.94
Corona FSE	0.94	0.94
Corona pk3	0.97	0.97
IFN Gamma	0.93	0.93

- 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?

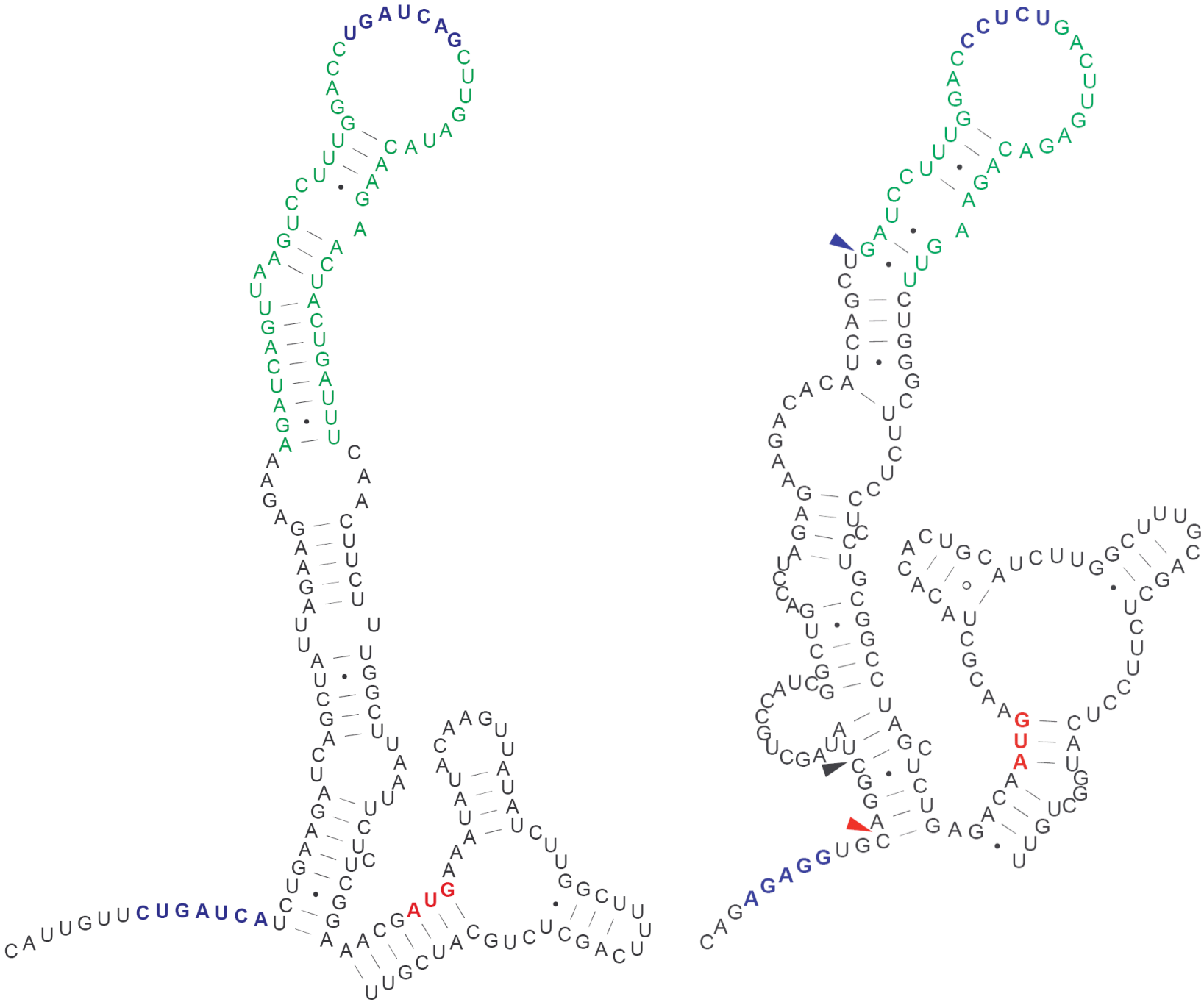
Test 2: Homologue Search Results



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.

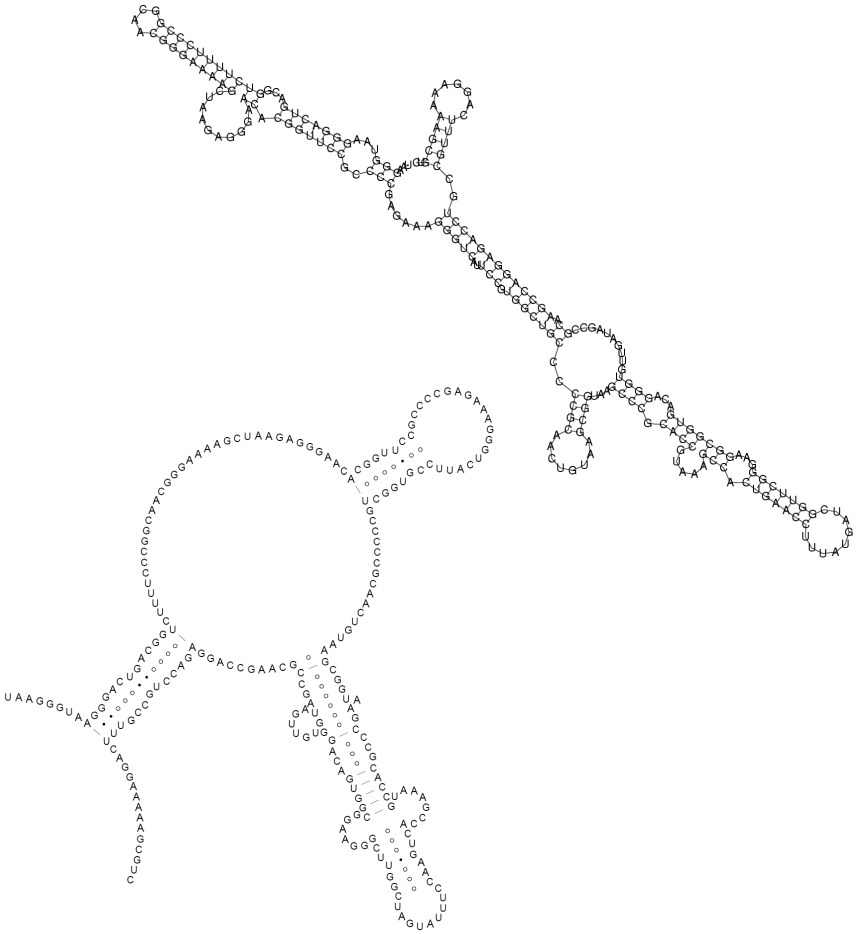
- We hope PAL will help to understand and explore the impact of
- pseudo-knotted RNAs in cellular function.

Reformulate the folding problem

RNA secondary structure prediction problem

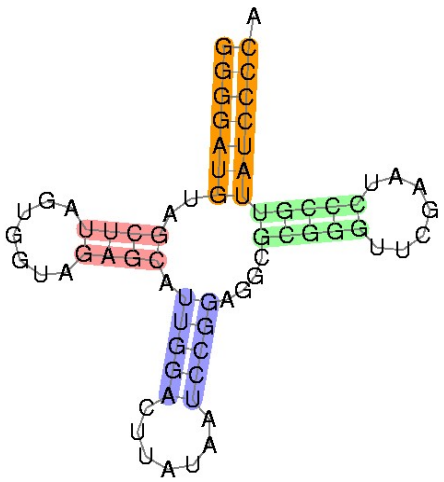
- Algorithms/programs to compute the minimum energy:
  - Nussinov et al (1978), Waterman (1978), Smith and Waterman (1978), and Zuker and Sankoff (1984).
  - **Mfold** (Zuker 2003) and **RNAfold (ViennaRNA)** (Hofacker 2003).

- 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.

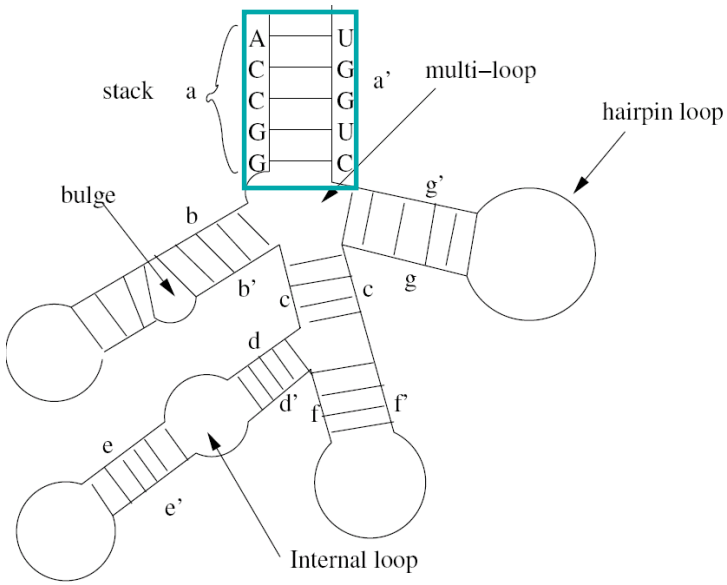
RNA consensus folding problem

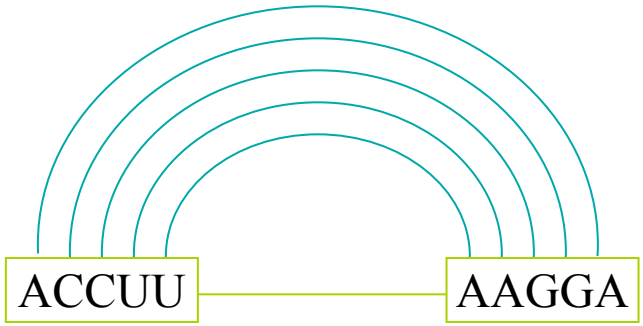
RNA consensus folding problem: computing the common secondary structure for a set of unaligned RNA sequences

- 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

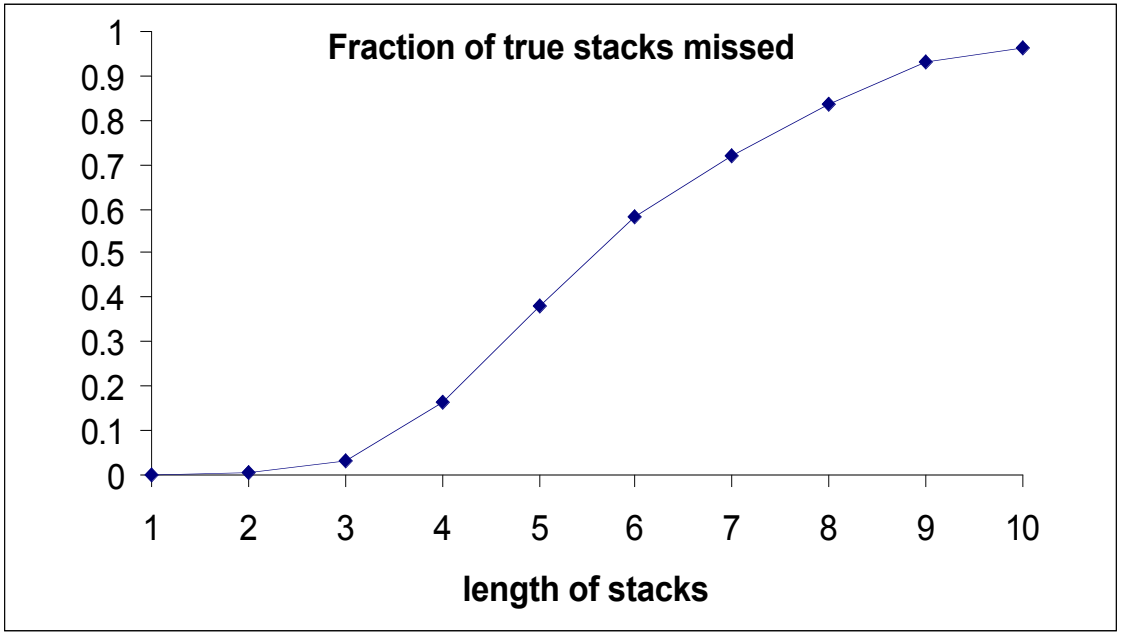




$p = (1/4)^5 < 0.001.$

- 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.

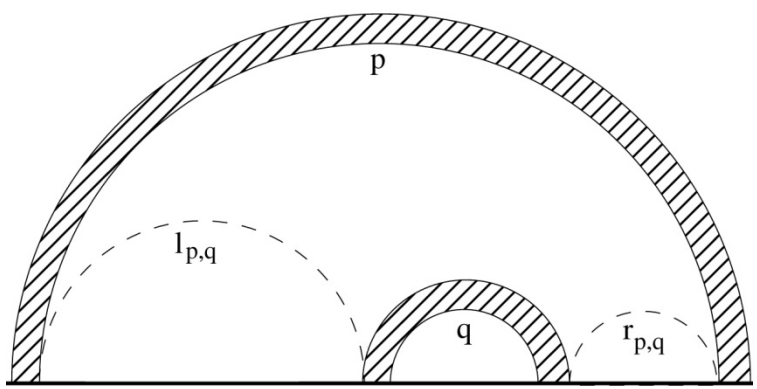
Statistics of the stacks in Rfam database



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

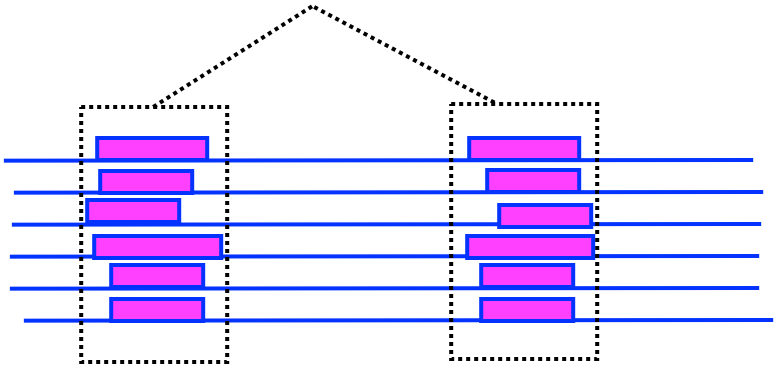
- The Nussinov model (simplified model)
  - Basically counts the number of base pairs
- The *Stack-based Folding* algorithm
  - N(p) : the maximal number of base pairs of all stack configurations within *p* (including *p* itself)

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



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)



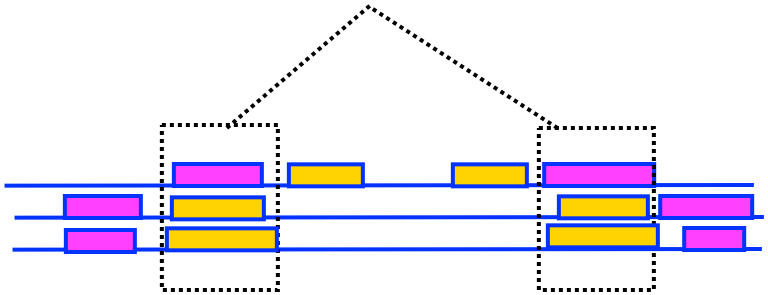
Several heuristic methods have been developed by finding anchored stacks:

- 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.



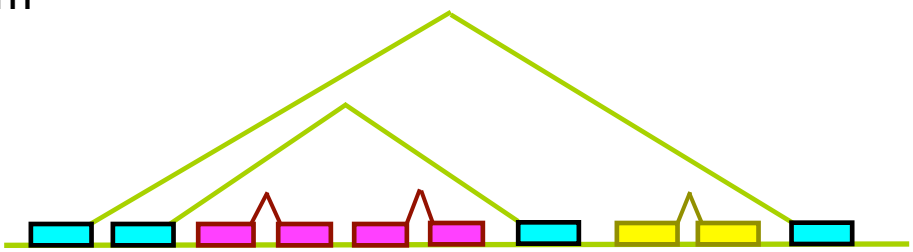
A global approach: configuration of stacks

- 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.

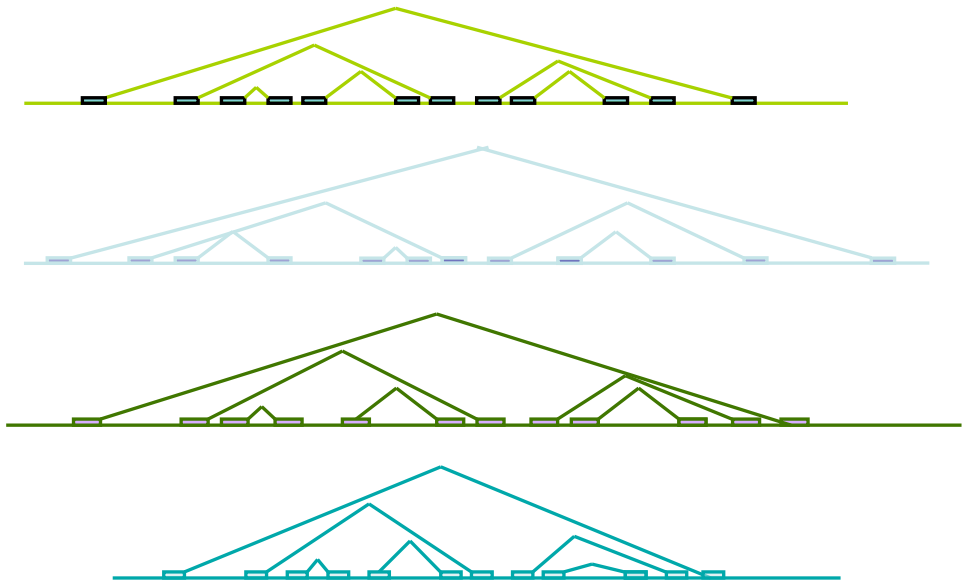
- Stack configuration:
  - Nested stacks
  - Parallel stacks
  - Crossing stacks (pseudo knots)

- More generalized stacks can include mismatches in the stacks.

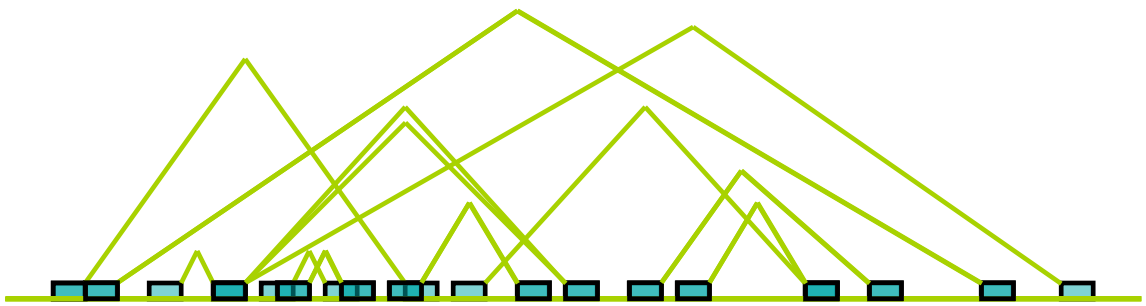


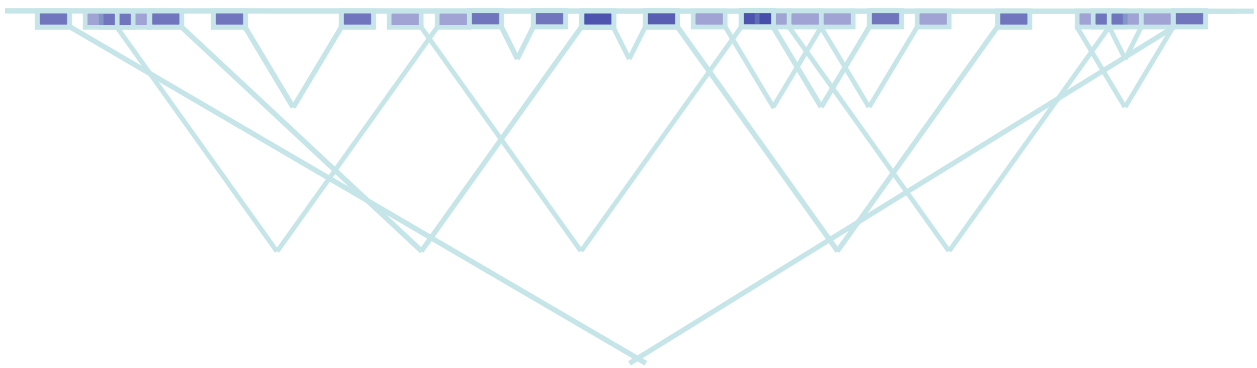
RNA Stack-based Consensus Folding (RNAscf) problem

- 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.

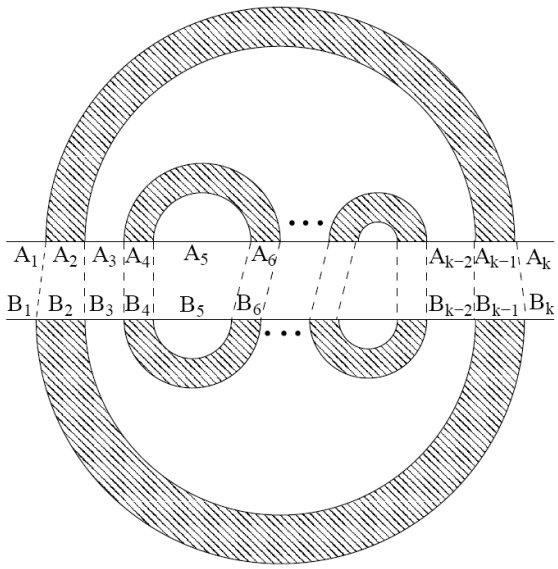


RNA stack-based consensus folding for pairwise sequences





A matching stack-configurations on two sequences



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

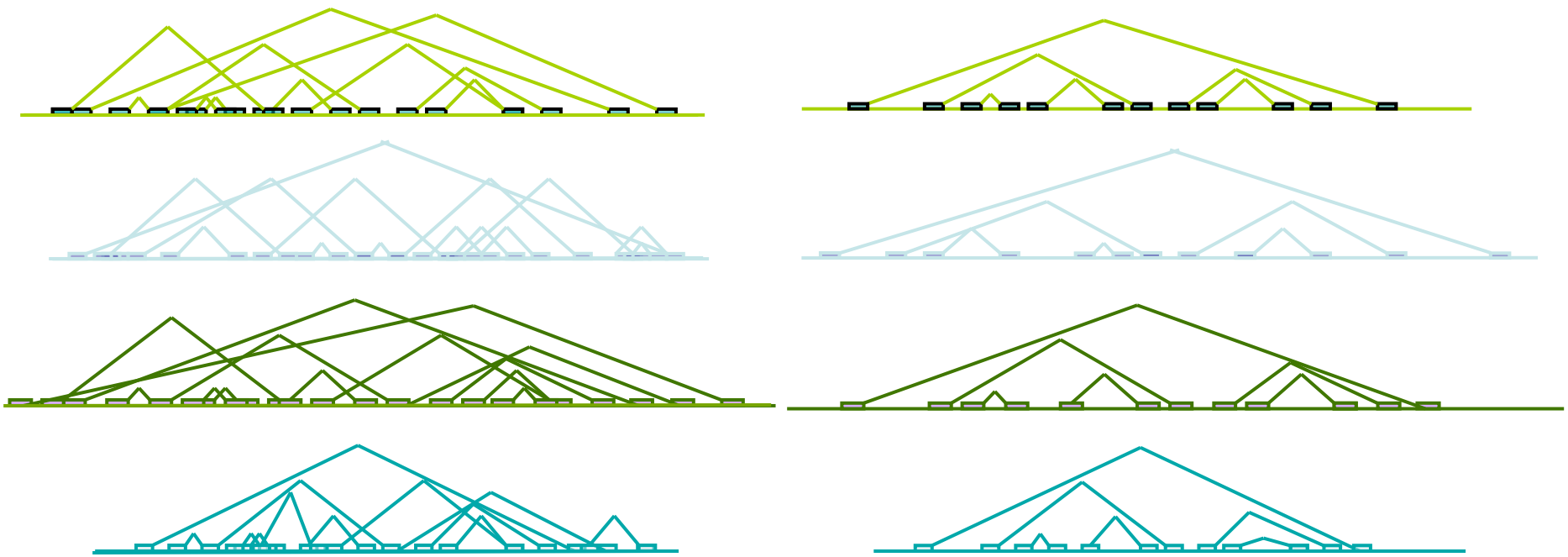
Weights of different costs

Energy of the consensus structure

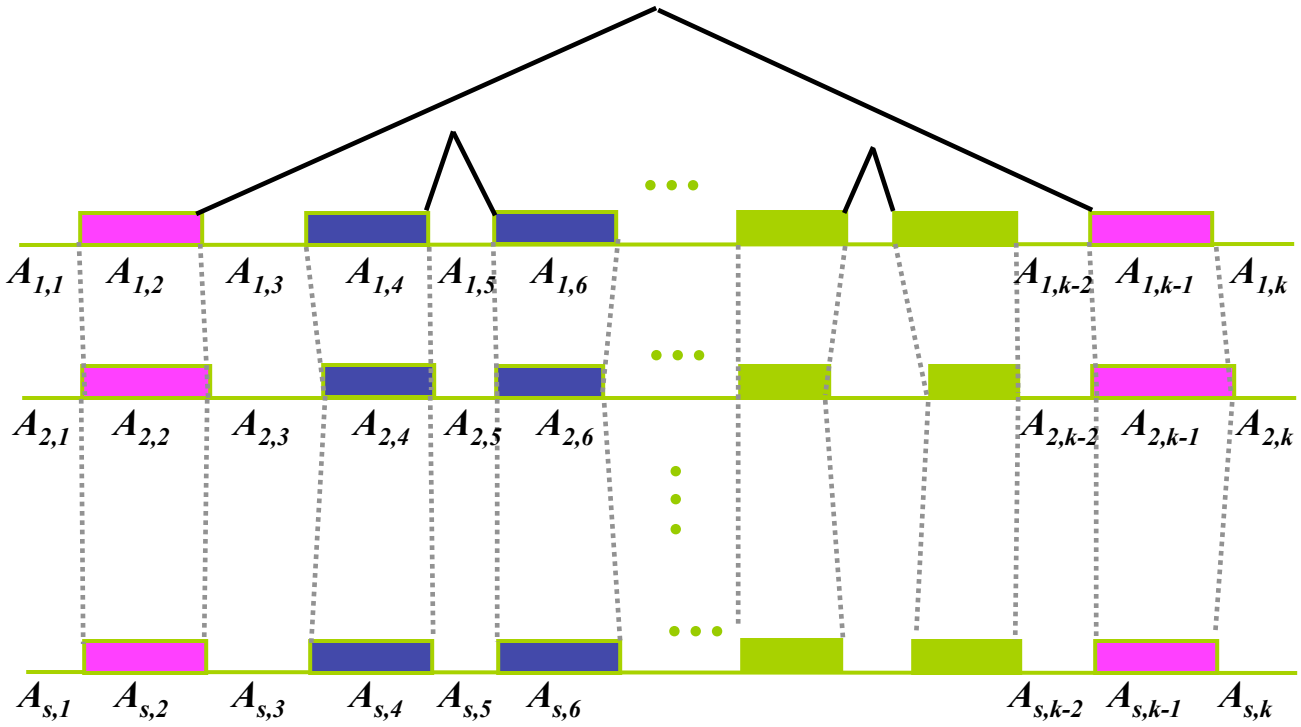
Sequence similarity of unpaired regions

Sequence similarity of stacks

RNA Stack-based Consensus Folding for multiple sequences



Cost function for multiple sequences

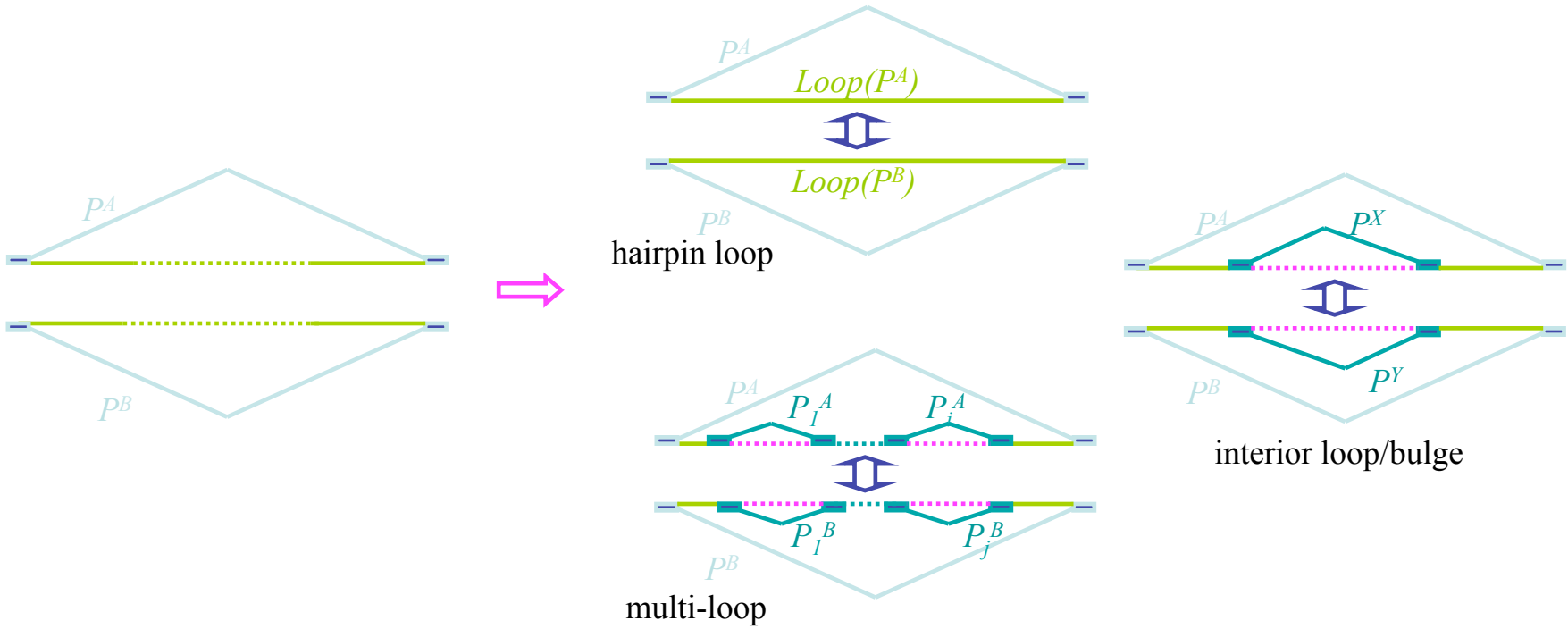


$$M(\mathcal{P}(A_1,\ldots,A_s)) = w_1\Phi(\mathcal{P}(A_1,\ldots,A_s)) + w_2\sum_{j\in\mathcal{P}}\mathcal{S}\begin{pmatrix}A_{1,j},\\ A_{2,j},\\ \ldots,\\ A_{s,j}\end{pmatrix} + w_3\sum_{j\notin\mathcal{P}}\mathcal{S}\begin{pmatrix}A_{1,j},\\ A_{2,j},\\ \ldots,\\ A_{s,j}\end{pmatrix}$$

Compute an optimal stack configuration for two sequences

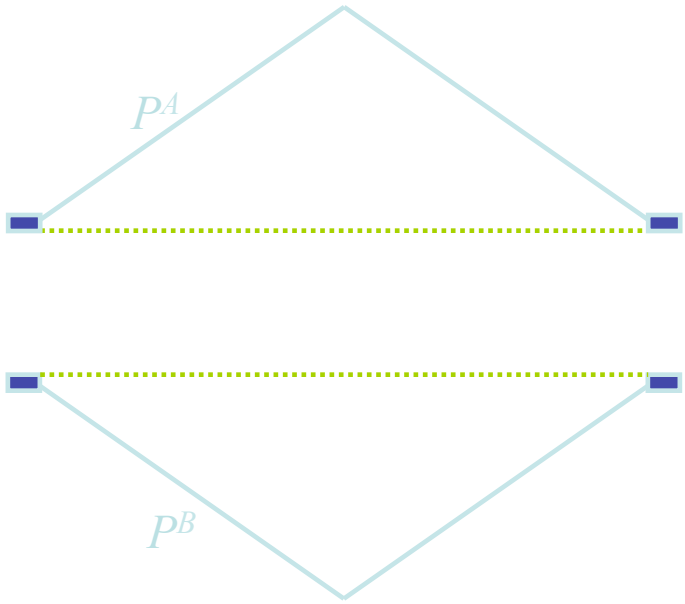
- Dynamic programming algorithm is used to align RNA sequences and find an optimal configuration at the same time.
  - The algorithm is similar to prior work (Sankoff 1985, Bafna et al. 1995)
  - Differences:
    - We use stacks as the basic structural elements.
    - Prior work used individual base 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:



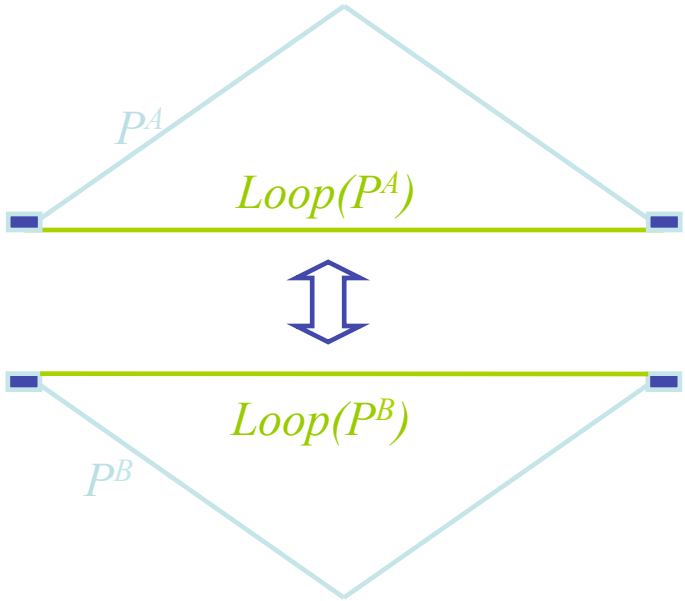
$$M[P^A, P^B] = M_s[P^A, P^B] + \min \left\{ \begin{array}{ll} M_h[P^A, P^B], & \text{(* hairpin loop *)} \\ M_b[P^A, P^B], & \text{(* interior loop/bulge *)} \\ M_m[P^A, P^B] & \text{(* multi-loop *)} \end{array} \right\}$$

The score of matching stacks:



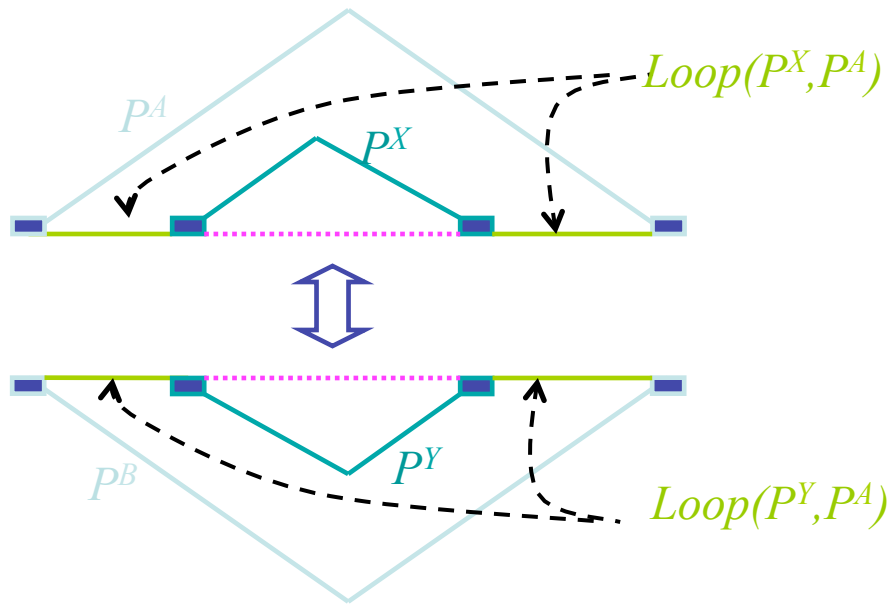
$$M_s[P^A, P^B] = w_1 \max \left\{ \begin{array}{c} \mathcal{E}_s(P^A), \\ \mathcal{E}_s(P^B) \end{array} \right\} + w_2 \mathcal{S} \left(\begin{array}{c} \mathrm{Seq}(P^A) \\ \mathrm{Seq}(P^B) \end{array} \right)$$

The score of matching hairpin loops:



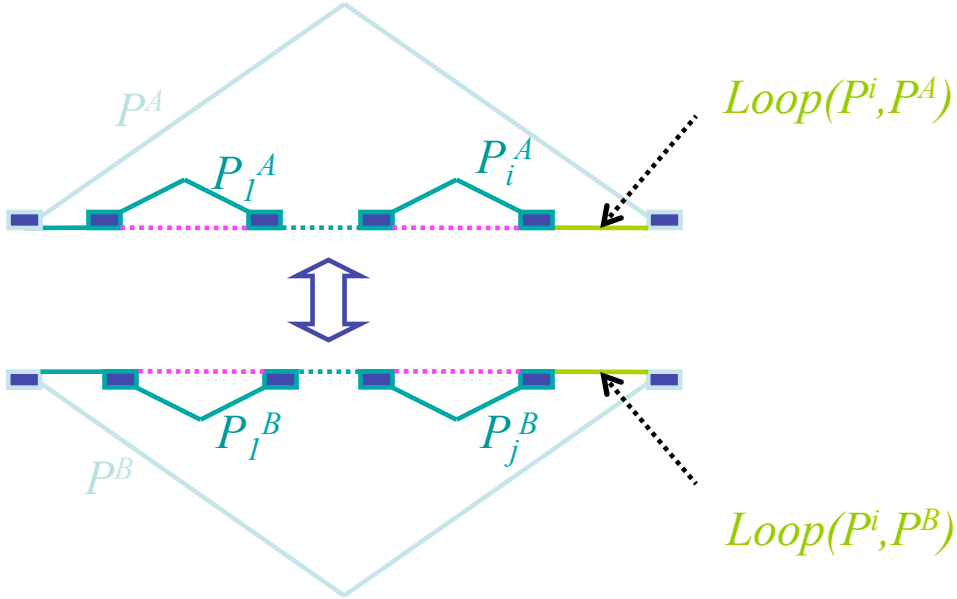
$$M_h[P^A, P^B] = \boxed{w_1 \max \left\{ \begin{array}{c} \mathcal{E}_h(|\mathrm{Loop}(P^A)|), \\ \mathcal{E}_h(|\mathrm{Loop}(P^B)|) \end{array} \right\} + w_3 \mathcal{S} \left(\begin{array}{c} \mathrm{Loop}(P^A), \\ \mathrm{Loop}(P^B) \end{array} \right)}$$

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:



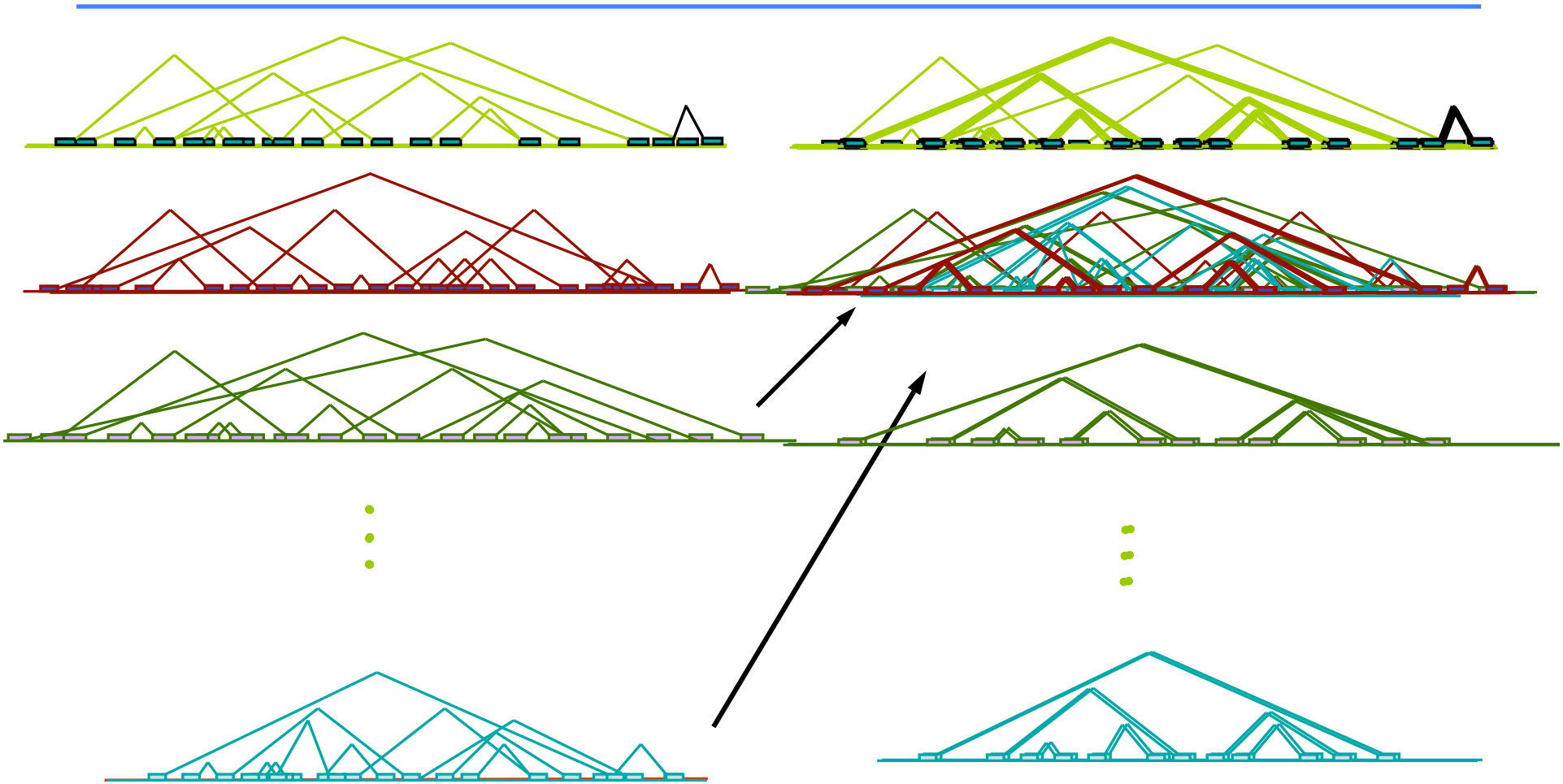
$$M_m[P^A, P^B] = \min_{i,j} \left\{ \boxed{M_c[P_i^A, P_j^A]} + \boxed{w_1 \max \left\{ \begin{array}{l} \mathcal{E}_m(|\mathrm{Loop}(P^i, P^A)|), \\ \mathcal{E}_m(|\mathrm{Loop}(P^j, P^B)|) \end{array} \right\} + w_3 \mathcal{S} \left(\begin{array}{l} \mathrm{Loop}(P^i, P^A), \\ \mathrm{Loop}(P^j,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

Consensus folding for multiple sequences

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

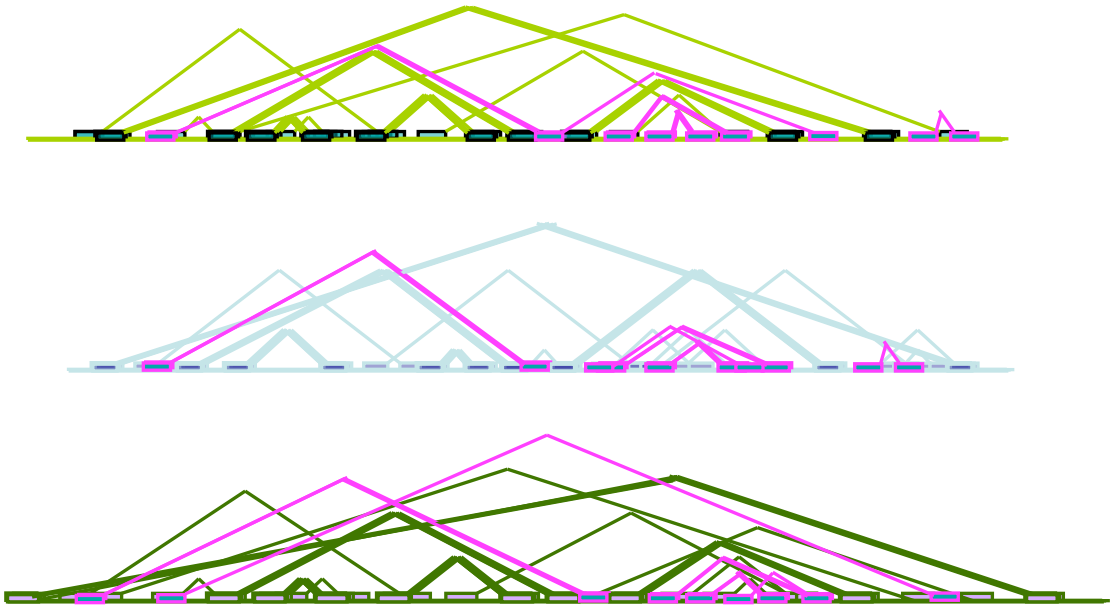
- We use a heuristic method based on the notion of star-alignment.
  - Compute an optimal configuration from a random seed pair.
  - Align all individual sequences to this configuration.
  - Choose the conserved stack configuration in all sequences.
  - Allow some stacks to be partially conserved (at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



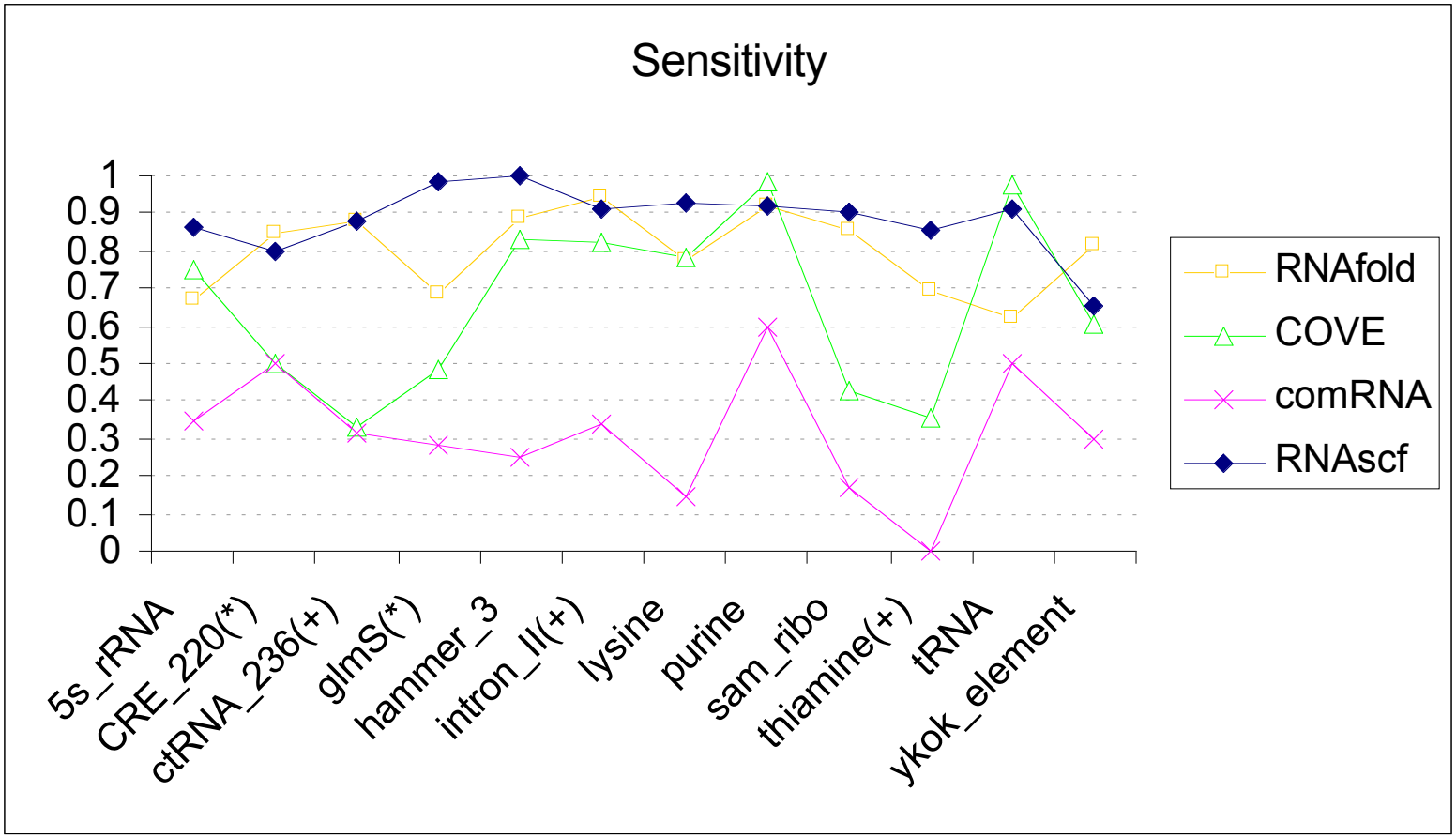
Test dataset

- 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

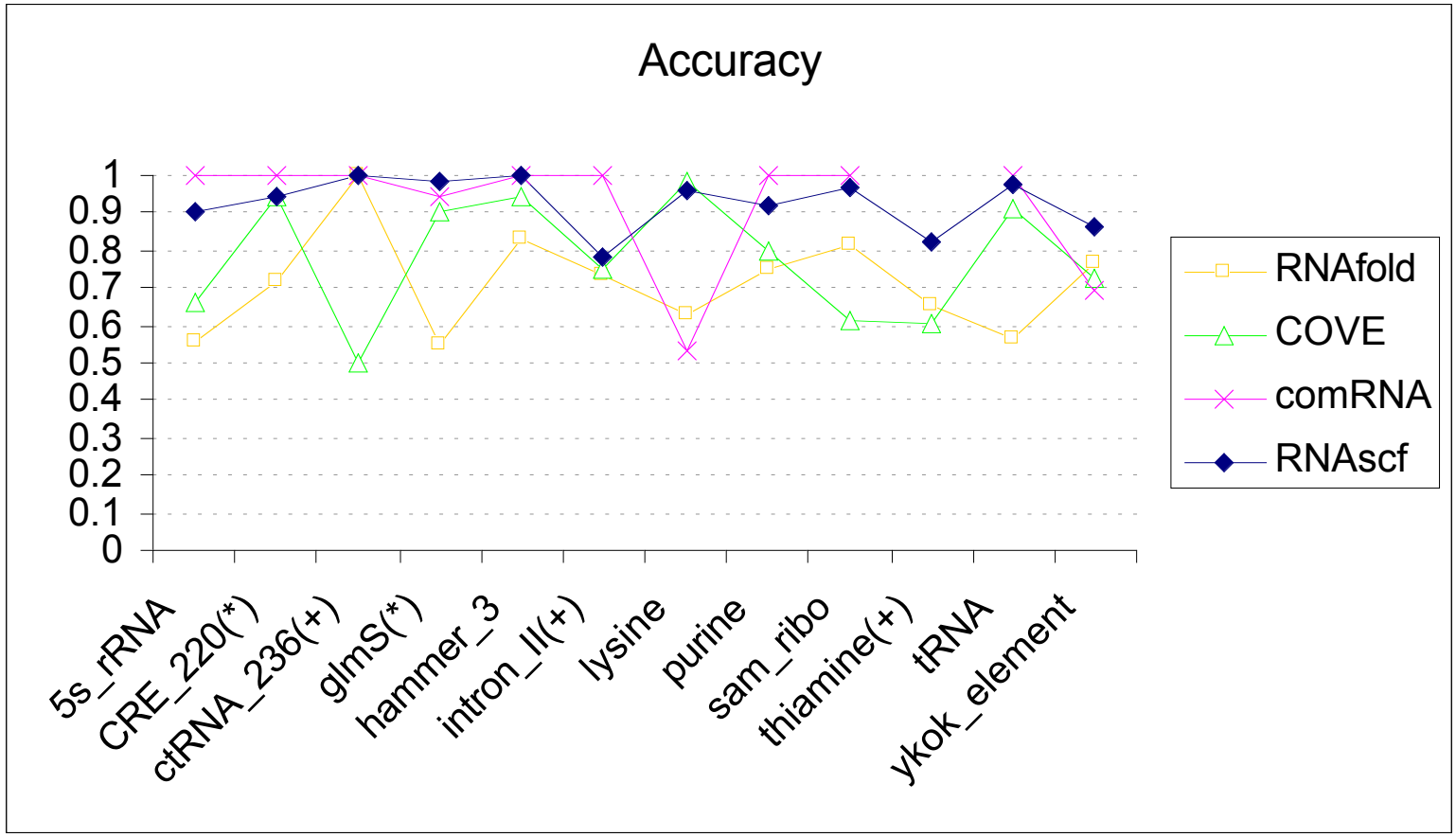
Test results

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

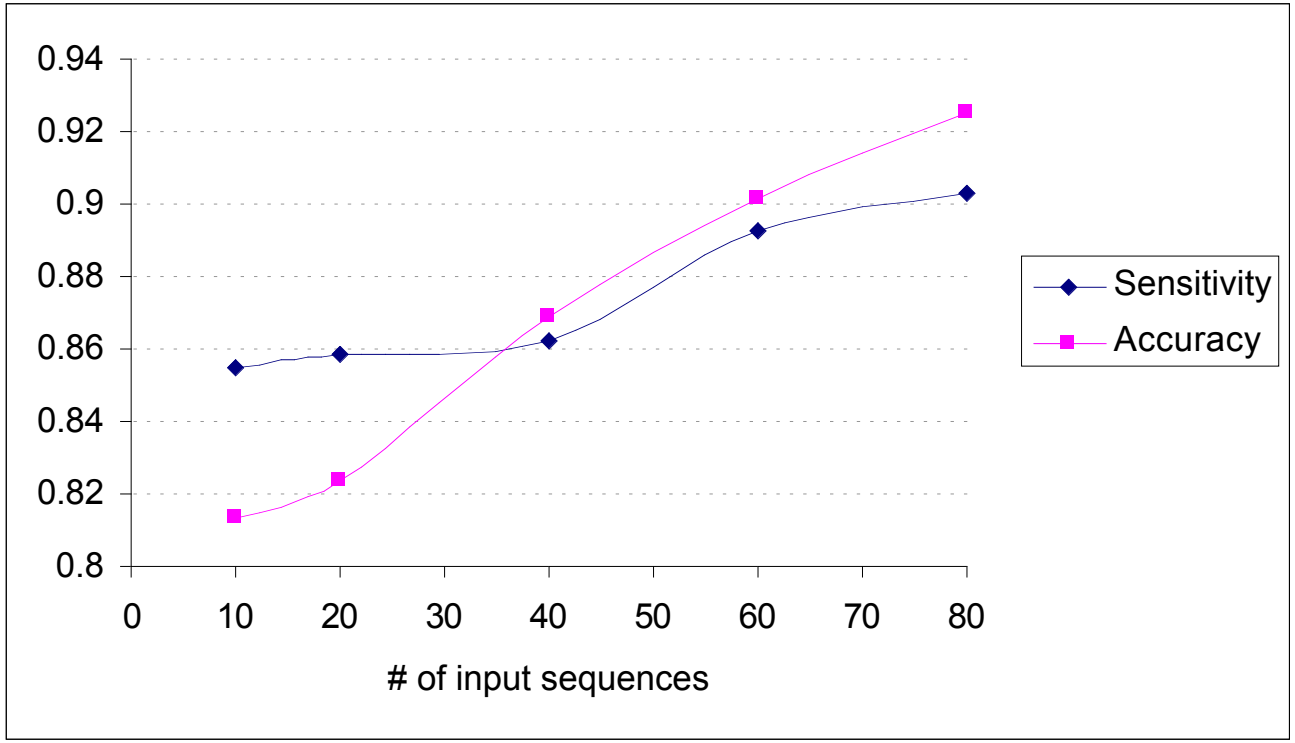
Test results



Test results

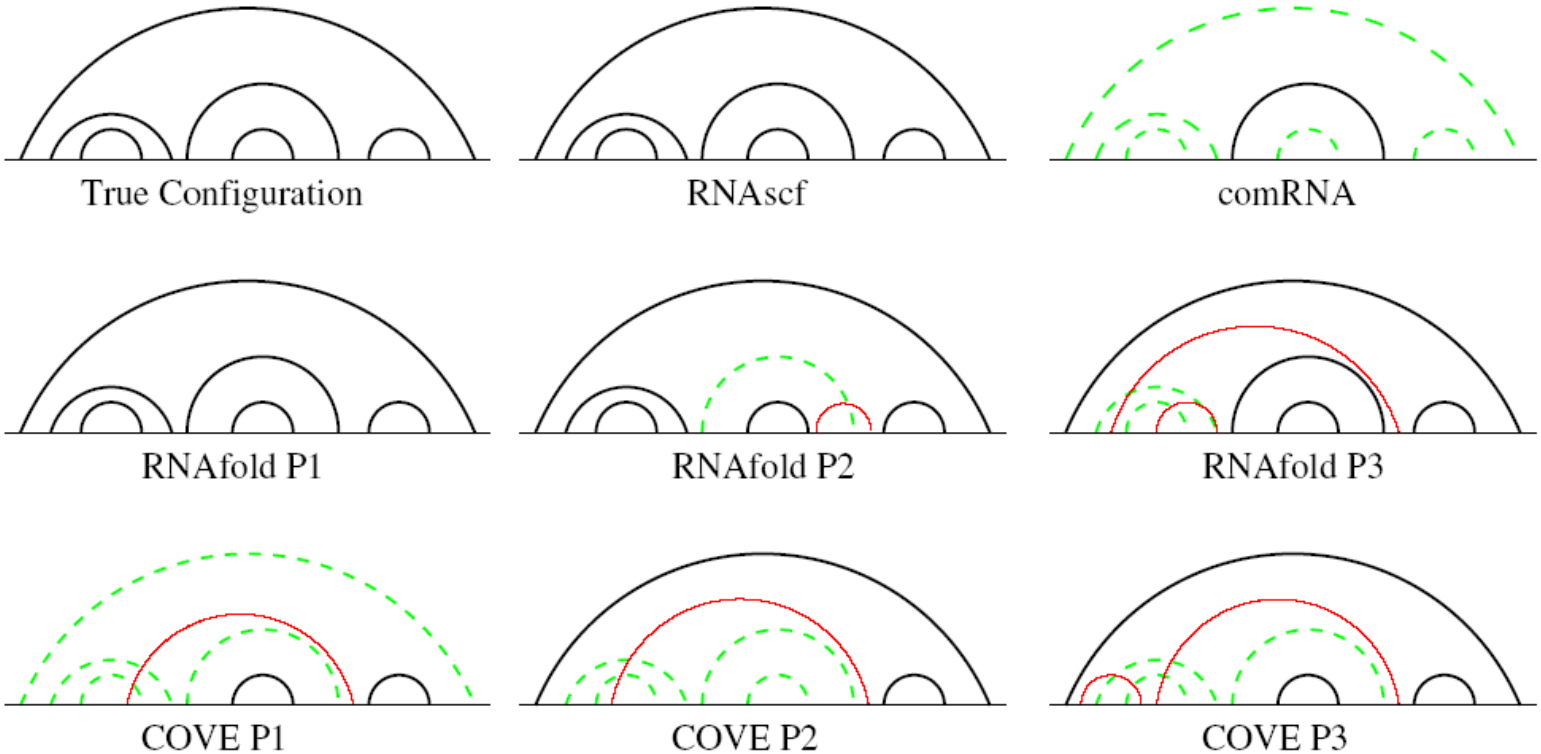


Performance improves when the number of sequences increases



(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.