UNIVERSITÀ DEGLI STUDI DI PADOVA

NOTES OF

ALGORITHMS FOR BIOINFORMATICS

(Version 22/06/2020)

Edited by:
Stefano Ivancich
Luca Masiero

CONTENTS

1.Motif Finding

1.1. Basics

DNA: sequenza di lettere **ACGT**
Genoma: sequenza di DNA. Sequenza di $\{A, C, G, T\}$

- Uomo: 3 miliardi di caratteri
- Batteri: 600mila
- Virus: migliaia

Gene: sottostringa del DNA

- codifica (produce) le proteine.

Proteina: stringa su un alfabeto di **20 caratteri**

- è un mattoncino elementare per costruire la cellula.
- **Amminoacido**: singolo carattere della proteina. Composto da una 3-pleta di ACGT

Implanting Motif: si inserisce una stringa in diversi punti in un'altra stringa.

actgatactagatcatagacatg --AAAGGG----> actgaAAAGGGtactagatcataAAAGGGgacatg

- With mutations: si impianta la stringa e si cambiano dei caratteri a caso nella stringa impiantata:

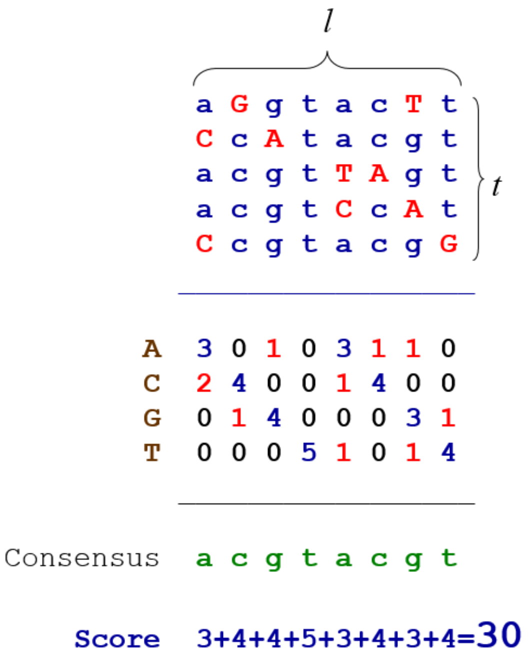
actgaAgAtGGtactagatcataAtAaGGgacatg

Challenge problem: trovare un Motif in un esempio che ha

- t sequenze
- ogni sequenza ha un pattern impiantato lungo n
- ogni pattern impiantato ha K mutazioni.
- (N,K)-motif

Supponiamo di conoscere gli indici di inizio del motif: $\boldsymbol{s} = (s_1, \dots, s_t)$

- Allineo le stringhe in una matrice (**Aligment Matrix**)
- Per ogni colonna conto quante volte si ripetono le ATGC (**Profile Matrix**)
- Consensus: prendo il massimo di ogni colonna.



l

	a	G	g	t	a	c	T	t
	C	c	A	t	a	c	g	t
	a	c	g	t	T	A	g	t
	a	c	g	t	C	c	A	t
	C	c	g	t	a	c	g	G
A	3	0	1	0	3	1	1	0
C	2	4	0	0	1	4	0	0
G	0	1	4	0	0	0	3	1
T	0	0	0	5	1	0	1	4
Consensus	a	c	g	t	a	c	g	t

t

Score 3+4+4+5+3+4+3+4=30

t: number of DNA sequences
n: length of each DNA sequence
DNA: $t \times n$ array
l: length of the motif (l-mer)
s_i: starting position of an l-mer in sequence i
$\boldsymbol{s} = (s_1, \dots, s_t)$: array of motif's starting positions

$$\text{Score}(\boldsymbol{s}, \boldsymbol{DNA}) = \sum_{i=1}^{l} \max_{k \in \{A,T,C,G\}} \text{count}(k, i)$$

Più grande è lo score più le stringhe sono correlate.

1.2. Motif Finding Problem

Motif Finding Problem: Given a set of DNA sequences, find a set of l-mers, one from each sequence, that maximizes the consensus score.
Input:

- $t \times n$ matrix of DNA
- l length of the pattern to find

Output: An array of t starting positions $\boldsymbol{s} = (s_1, \dots, s_t)$ **maximizing** $\mathrm{Score}(\boldsymbol{s}, \boldsymbol{DNA})$.

Brute force solution: $O(l(n-l+1)^t) = O(ln^t)$ compute the scores for each possible combination of starting positions $\boldsymbol{s}$

```
BruteForceMotifSearch($DNA, t, n, l$)
1  $bestScore \leftarrow 0$
2  for each $(s_1, \dots, s_t)$ from $(1, \dots, 1)$ to $(n-l+1, \dots, n-l+1)$
3      if $Score(\mathbf{s}, DNA) > bestScore$
4          $bestScore \leftarrow Score(\mathbf{s}, DNA)$
5          bestMotif $\leftarrow (s_1, s_2, \dots, s_t)$
6  return bestMotif
```

1.3. Median String Problem

Hamming Distance $d_H(\boldsymbol{v}, \boldsymbol{w})$ =# nucleotide pairs that do not match
$\mathrm{TotalDistance}(\boldsymbol{v}, \boldsymbol{DNA}) = \min_{\boldsymbol{s}=(s_1,\dots,s_t)} d_H(\boldsymbol{v}, \boldsymbol{s})$

- For each DNA sequence $\boldsymbol{i}$, compute all $d_H(\boldsymbol{v}, \boldsymbol{x})$ where $\boldsymbol{x}$ is an l-mers with starting position s_i
- Find minimum of $d_H(\boldsymbol{v}, \boldsymbol{x})$ among all l-mers in sequence $\boldsymbol{i}$

Median String Problem: Given a set of DNA sequences, find a median string.
Input:

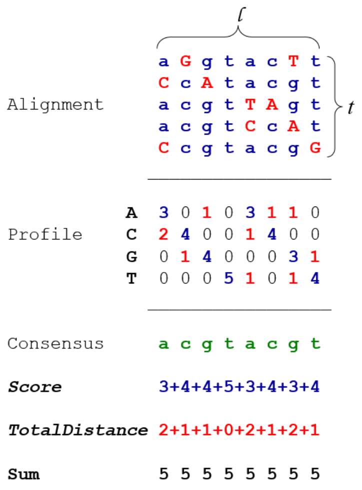
- $t \times n$ matrix of DNA
- l length of the pattern to find

Output: A string $\boldsymbol{v}$ of l nucleotides that minimizes $\mathrm{TotalDistance}(\boldsymbol{v}, \boldsymbol{DNA})$ over all strings of that length.

Brute force solution: $O(nt4^l)$ compute the scores for each possible combination of $\boldsymbol{v}$ (AAA…,…,TTT…)

```
BruteForceMedianSearch($DNA, t, n, l$)
1  $bestWord \leftarrow$ AAA$\cdots$AA
2  $bestDistance \leftarrow \infty$
3  for each $l$-mer $word$ from AAA...A to TTT...T
4      if TotalDistance($word, DNA$) $< bestDistance$
5          $bestDistance \leftarrow$ TotalDistance($word, DNA$)
6          $bestWord \leftarrow word$
7  return $bestWord$
```

Motif Finding Problem = Median String Problem: Maximizing Score = minimizing TotalDistance



		l
Alignment		a G g t a c T t
		C c A t a c g t
		a c g t T A g t
		a c g t C c A t
		C c g t a c g G
Profile	A	3 0 1 0 3 1 1 0
	C	2 4 0 0 1 4 0 0
	G	0 1 4 0 0 0 3 1
	T	0 0 0 5 1 0 1 4
Consensus		a c g t a c g t
Score		3+4+4+5+3+4+3+4
TotalDistance		2+1+1+0+2+1+2+1
Sum		5 5 5 5 5 5 5 5

- At any column ***i***
  $Score_i + TotalDistance_i = t$
- Because there are l columns
  $Score + TotalDistance = l * t$
- Rearranging:
  $Score = l * t - TotalDistance$
- $l * t$ is constant.
  Minimization of the right side is equivalent to the maximization of the left side

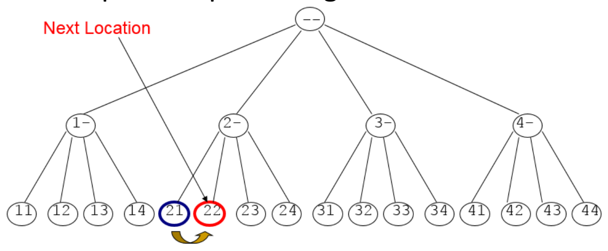
1.4. Search Tree

Search Tree: is used to implement these two lines:

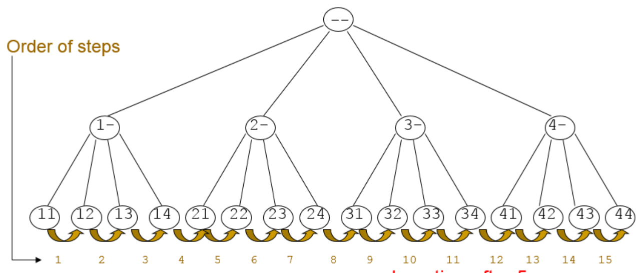
- Motif Finding problem: **for each** $(s_1, \ldots, s_t)$ **from** $(1, \ldots, 1)$ **to** $(n-l+1, \ldots, n-l+1)$
- Median String problem: **for** each l-mer *word* **from** AAA…A **to** TTT…T

And provide **4 moves** than can be used to skip fewer promising values:

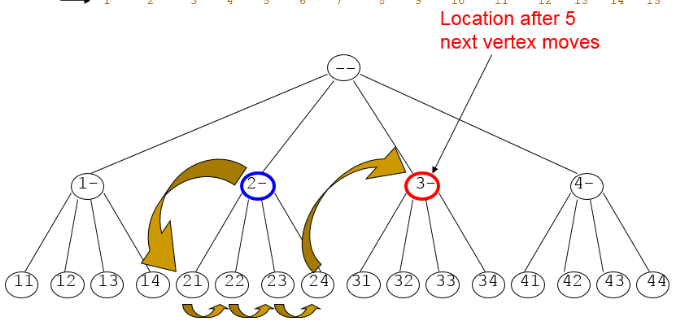
```
NEXTLEAF(a, L, k)
1  for $i \leftarrow L$ to 1
2      if $a_i < k$
3          $a_i \leftarrow a_i + 1$
4          return a
5      $a_i \leftarrow 1$
6  return a
```



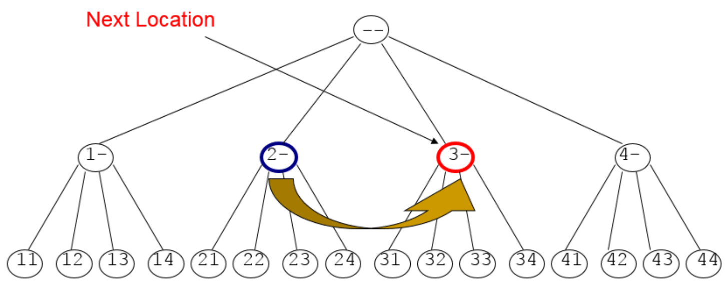
```
ALLLEAVES(L, k)
1  a ← (1, . . . , 1)
2  while forever
3      output a
4      a ← NEXTLEAF(a, L, k)
5      if a = (1, 1, . . . , 1)
6          return
```



```
NEXTVERTEX(a, i, L, k)
1  if $i < L$
2      $a_{i+1} \leftarrow 1$
3      return $(a, i+1)$
4  else
5      for $j \leftarrow L$ to 1
6          if $a_j < k$
7              $a_j \leftarrow a_j + 1$
8              return $(a, j)$
9  return $(a, 0)$
```

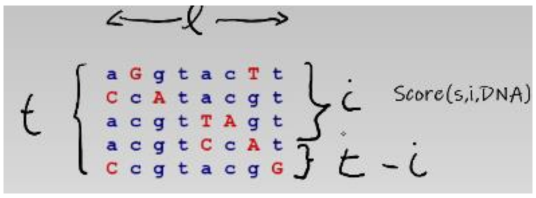


```
BYPASS(a, i, L, k)
1  for $j \leftarrow i$ to 1
2      if $a_j < k$
3          $a_j \leftarrow a_j + 1$
4          return $(a, j)$
5  return $(a, 0)$
```



1.5. Branch and Bound

Same Worst Case but average case is better.



Branch and Bound Motif Search: If we have analyzed the first i sequences and they provide a very bad score, assuming that the rest $t - i$ lines gives the best score possible, but this is less than the previous `BestScore` found, it has no sense to continue searching in that branch, so we skip directly to the next branch using `ByPass()`, otherwise we continue searching using `NextVertex()`

This saves us from looking at $(n - l + 1)^{t-i}$ leaves.

```
BRANCHANDBOUNDMOTIFSEARCH($DNA, t, n, l$)
s ← (1, . . . , 1)
$bestScore$ ← 0
$i$ ← 1
while $i > 0$
    if $i < t$          # Internal Nodes
        $optimisticScore$ ← $Score(s, i, DNA) + (t - i) \cdot l$
        if $optimisticScore < bestScore$
            (s, $i$) ← BYPASS(s, $i, t, n - l + 1$)
        else
            (s, $i$) ← NEXTVERTEX(s, $i, t, n - l + 1$)
    else                # Leaves
        if $Score(s, DNA) > bestScore$
            $bestScore$ ← $Score(s)$
            bestMotif ← $(s_1, s_2, \ldots, s_t)$
        (s, $i$) ← NEXTVERTEX(s, $i, t, n - l + 1$)
return bestMotif
```

```
BRUTEFORCEMOTIFSEARCHAGAIN($DNA, t, n, l$)
s ← (1, 1, . . . , 1)
$bestScore$ ← $Score(s, DNA)$
while forever
    s ← NEXTLEAF(s, $t, n - l + 1$)
    if $Score(s, DNA) > bestScore$
        $bestScore$ ← $Score(s, DNA)$
        bestMotif ← $(s_1, s_2, \ldots, s_t)$
    if $s = (1, 1, \ldots, 1)$
        return bestMotif
```

Branch and Bound Median String Search: if the total distance for a prefix is greater than that for the best word so far: TotalDistance (prefix, DNA) > BestDistance, there is no sense exploring the remaining part of the word. So we skip directly to the next branch using `ByPass()`, otherwise we continue searching using `NextVertex()`

```
BRANCHANDBOUNDMEDIANSEARCH($DNA, t, n, l$)
s ← (1, 1, . . . , 1)
$bestDistance$ ← ∞
$i$ ← 1
while $i > 0$
    if $i < l$
        $prefix$ ← nucleotide string corresponding to $(s_1, s_2, \ldots, s_i)$
        $optimisticDistance$ ← TOTALDISTANCE($prefix, DNA$)
        if $optimisticDistance > bestDistance$
            (s, $i$) ← BYPASS(s, $i, l$, 4)
        else
            (s, $i$) ← NEXTVERTEX(s, $i, l$, 4)
    else
        $word$ ← nucleotide string corresponding to $(s_1, s_2, \ldots s_l)$
        if TOTALDISTANCE($word, DNA$) < $bestDistance$
            $bestDistance$ ← TOTALDISTANCE($word, DNA$)
            $bestWord$ ← $word$
        (s, $i$) ← NEXTVERTEX(s, $i, l$, 4)
return $bestWord$
```

There are other techniques that uses more constraint on the bounds, and others that don't find the best solution but a good one. Motif Finding Problem 4

Planted Motif Search (PMS): $O\left(nm\binom{l}{d}3^d\frac{l}{w}\right)$ where d: hamming distance, w: word length of computer

Given the sequence S_i

- $C_i =$ collection of all possible l-mers
- $L_i =$ From C_i Generate all patterns at hamming distance d
- Sort L_i
- Eliminate duplicates from L_i
- Find motif common to all lists L_i

2. Randomized Algorithms for Motif Finding

Randomized quicksort: pick the pivot randomly enables to have $O(n \log n)$ expected run time.

Las Vegas Algorithms: always produce the correct solution (eg. Randomized quicksort), but they are often hard to come by.
Monte Carlo Algorithms: do not always produce the correct solution.

2.1. Greedy Profile Motif Search

Let $\boldsymbol{s} = (s_1, \dots, s_t)$ be the set of starting positions for l-mers in our t sequences.
The substrings corresponding to these starting positions will form:

- $t \times l$ **alignment matrix**
- $4 \times l$ **profile matrix** P, defined in terms of the frequency of letters, not as the count of letters.

$\Pr(\boldsymbol{a}|\boldsymbol{P}) = \prod_{i=1}^{n} p_{a_i,i}$ probability that an l-mer $\boldsymbol{a}$ was created by the Profile P.

- If $\boldsymbol{a}$ is very similar to the consensus string of P then $\Pr(\boldsymbol{a}|\boldsymbol{P})$ is high
- If $\boldsymbol{a}$ is very different to the consensus string of P then $\Pr(\boldsymbol{a}|\boldsymbol{P})$ is low

Given a profile: **P** =

A	**1/2**	7/8	**3/8**	0	**1/8**	0
C	1/8	0	1/2	**5/8**	3/8	0
T	1/8	**1/8**	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	**1/8**

The probability of the consensus string:
Prob(**aaacct**|**P**) = 1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8 = .033646

Probability of a different string:
Prob(**atacag**|**P**) = 1/2 x 1/8 x 3/8 x 5/8 x 1/8 x 1/8 = .001602

***P*-Most Probable *l*-mer** in a single sequence: is the l-mer in that sequence which has the highest probability of being created from the profile P.

Given a sequence = ctataaaccttacatc, find the P-most probable *l*-mer

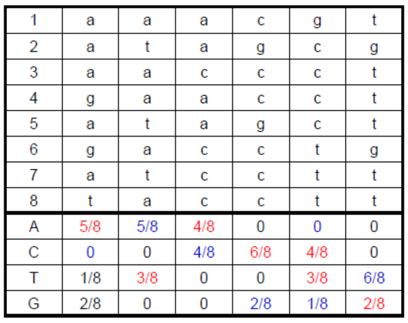
Compute *prob*(**a**|**P**) for every possible 6-mer:

String, Highlighted in Red	Calculations	*Prob*(**a**|**P**)
ctataaaccttacat	1/8 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 7/8 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 7/8 x 3/8 x 0 x 3/8 x 0	0
ctataaaccttacat	**1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8**	**.0336**
ctataaaccttacat	1/2 x 7/8 x 1/2 x 5/8 x 1/4 x 7/8	.0299
ctataaaccttacat	1/2 x 0 x 1/2 x 0 1/4 x 0	0
ctataaaccttacat	1/8 x 0 x 0 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 0 x 0 x 3/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 3/8 x 5/8 x 1/8 x 7/8	.0004

To avoid many entries with prob $\Pr(\boldsymbol{a}|\boldsymbol{P}) = 0$, there exist techniques to equate zero to a very small number so that one zero does not make the entire probability of a string zero.

***P*-Most Probable *l*-mer** in Many Sequences:

- Find the P-most probable l-mer in each of the sequences.
- Align those l-mers in a matrix.
- Calculate a new Profile matrix
- Compare it to the old Profile matrix. If the score had increased it ok, otherwise we are in the wrong direction.



1	a	a	a	c	g	t
2	a	t	a	g	c	g
3	a	a	c	c	c	t
4	g	a	a	c	c	t
5	a	t	a	g	c	t
6	g	a	c	c	t	g
7	a	t	c	c	t	t
8	t	a	c	c	t	t
A	5/8	5/8	4/8	0	0	0
C	0	0	4/8	6/8	4/8	0
T	1/8	3/8	0	0	3/8	6/8
G	2/8	0	0	2/8	1/8	2/8

Greedy Profile Motif Search:

- Select random starting positions.
- Create a profile $\boldsymbol{P}$ from the substrings at these starting positions.
- Find the $\boldsymbol{P}$-most probable l-mer $\boldsymbol{a}$ in each sequence and change the starting position to the starting position of $\boldsymbol{a}$.
- Compute a new profile based on the new starting positions after each iteration and proceed until we cannot increase the score anymore.

```
GREEDYPROFILEMOTIFSEARCH($DNA, t, n, l$)
1  Randomly select starting positions $\mathbf{s} = (s_1, \ldots, s_t)$ in $DNA$
2  Form profile P from s
3  $bestScore \leftarrow 0$
4  while $Score(\mathbf{s}, DNA) > bestScore$
5      $bestScore \leftarrow Score(\mathbf{s}, DNA)$
6      for $i \leftarrow 1$ to $t$
7          Find a P-most probable $l$-mer a from the $i$th sequence
8          $s_i \leftarrow$ starting position of a
9  return $bestScore$
```

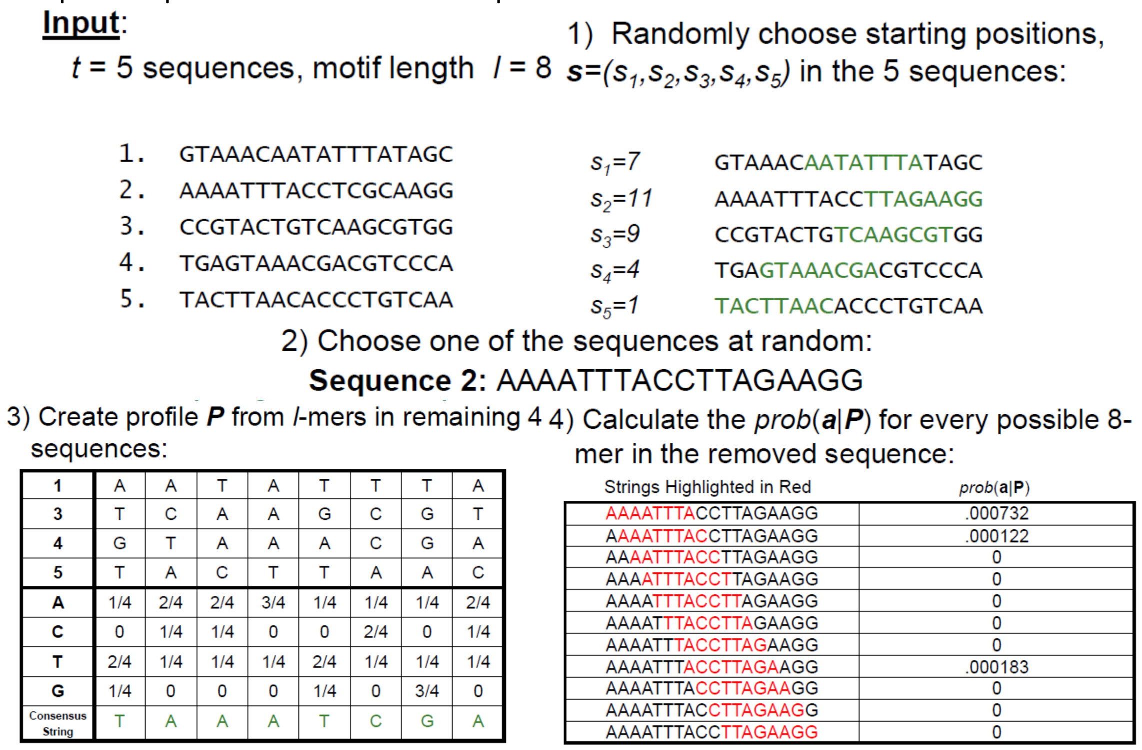
Since we choose starting positions randomly, there is little chance that our guess will be close to an optimal motif, meaning it will take a very long time to find the optimal motif. It is unlikely that the random starting positions will lead us to the correct solution at all. In practice, this algorithm is run many times with the hope that random starting positions will be close to the optimum solution simply by chance.

2.2. Gibbs Sampling

Greedy Profile Motif Search changes starting positions $(s_1, \ldots, s_t)$ between every iteration, and may change as many as all t positions in a single iteration. Gibbs sampling is an iterative procedure that at each iteration discards one l-mer from the alignment and replaces it with a new one. In other words**, it changes at most one position in s in each iteration** and thus moves with more caution in the space of all starting positions.

Gibbs Sampling:

- Randomly select starting positions $\boldsymbol{s} = (s_1, \ldots, s_t)$ in DNA and form the set of l-mers starting at these positions.
- Randomly choose one of t sequences.
- Create a profile P from the l-mers in the remaining $t-1$ sequences.
- For each position in the removed sequence, calculate the probability that the l-mer starting at this position is generated by profile $\boldsymbol{P}$
- Choose the new starting position for the removed sequence randomly, according to the probabilities calculated in step 4.
- Repeat steps 2-5 until there is no improvement.



Input:

t = 5 sequences, motif length l = 8

1. GTAAACAATATTTATAGC
2. AAAATTTACCTCGCAAGG
3. CCGTACTGTCAAGCGTGG
4. TGAGTAAACGACGTCCCA
5. TACTTAACACCCTGTCAA

1) Randomly choose starting positions, $\boldsymbol{s}=(s_1,s_2,s_3,s_4,s_5)$ in the 5 sequences:

$s_1=7$	GTAAACAATATTTATAGC
$s_2=11$	AAAATTTACCTTAGAAGG
$s_3=9$	CCGTACTGTCAAGCGTGG
$s_4=4$	TGAGTAAACGACGTCCCA
$s_5=1$	TACTTAACACCCTGTCAA

2) Choose one of the sequences at random:

Sequence 2: AAAATTTACCTTAGAAGG

3) Create profile ***P*** from *l*-mers in remaining 4 sequences:

1	A	A	T	A	T	T	T	A
3	T	C	A	A	G	C	G	T
4	G	T	A	A	A	C	G	A
5	T	A	C	T	T	A	A	C
A	1/4	2/4	2/4	3/4	1/4	1/4	1/4	2/4
C	0	1/4	1/4	0	0	2/4	0	1/4
T	2/4	1/4	1/4	1/4	2/4	1/4	1/4	1/4
G	1/4	0	0	0	1/4	0	3/4	0
Consensus String	T	A	A	A	T	C	G	A

4) Calculate the *prob*(***a***|***P***) for every possible 8-mer in the removed sequence:

Strings Highlighted in Red	*prob*(**a**|**P**)
AAAATTTACCTTAGAAGG	.000732
AAAATTTACCTTAGAAGG	.000122
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	.000183
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0

Gibbs sampling needs to be modified when applied to samples with unequal distributions of nucleotides.
Gibbs sampling often converges to locally optimal motifs rather than globally optimal motifs.
Needs to be run with many randomly chosen seeds to achieve good results.

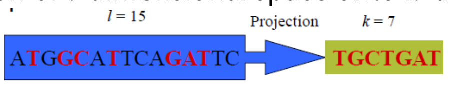
2.3. Random Projections

We randomly select a subset of positions in the pattern creating a projection of the pattern. Search for that projection in a hope that the selected positions are not affected by mutations in most instances of the motif.

Projection:

- Choose k positions in string of length l
- Concatenate nucleotides at chosen k positions to form k-tuple.

This can be viewed as a projection of l-dimensional space onto k-dimensional subspace.

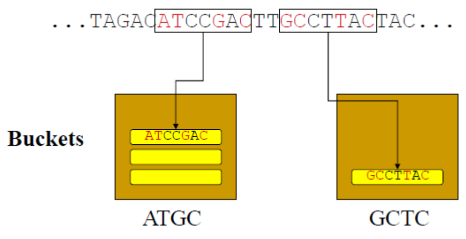


Projection = (2, 4, 5, 7, 11, 12, 13)

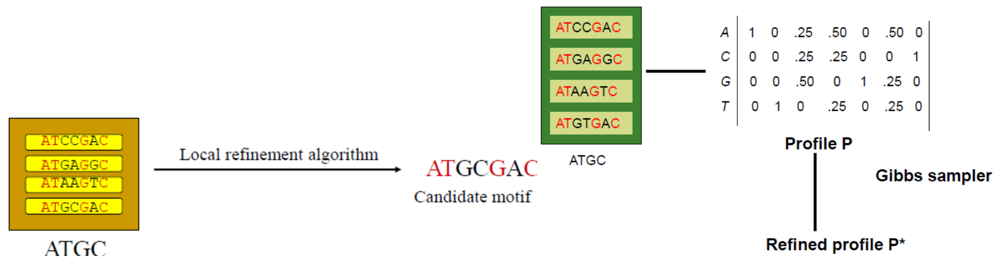
Random Projections Algorithm: (single iteration)

- Select k out of l positions uniformly at random.
- For each l-tuple x in input sequences, hash it into a bucket labeled by $h(x)$ (the label is composed by the letter at k selected positions).

Choose projection (1,2,5,7)



- For each bucket h containing more than s sequences (enriched buckets), extract a motif using a local refinement algorithm.
  For example, using Gibbs Sampler: form a profile $\boldsymbol{P}(h)$, then use $\boldsymbol{P}(h)$ as starting point to obtain refined profile $\boldsymbol{P}^*$.



A	1	0	.25	.50	0	.50	0
C	0	0	.25	.25	0	0	1
G	0	0	.50	0	1	.25	0
T	0	1	0	.25	0	.25	0

- Candidate motif is best found by selecting the best motif among refinements of all enriched buckets.

Some projections will fail to detect motifs but if we try many of them (run multiple times the algorithm) the probability that one of the buckets fills in is increasing.

Random Projection is a procedure for finding good starting points: every enriched bucket is a potential starting point. Feeding these starting points into existing algorithms (like Gibbs sampler) provides good local search in vicinity of every starting point.

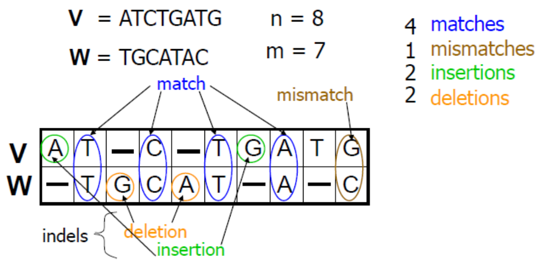
Choosing Projection Size k:

- small enough so that several motif instances hash to the same bucket.
- large enough to avoid contamination by spurious l-mers.

3. Dynamic Programming

Computing a similarity score between two genes tells how likely it is that they have similar functions. Dynamic programming is a technique for revealing similarities between genes.

3.1. Edit Distance

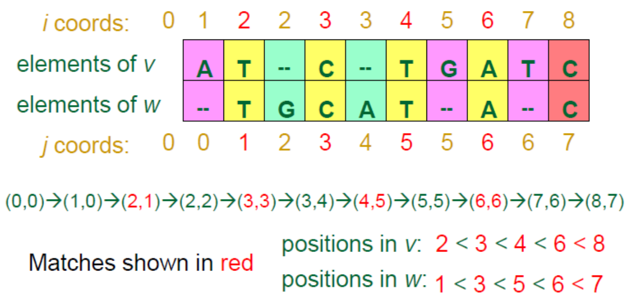


Alignment: $2 * k$ matrix where $k > m, n$
Same characters are aligned, then insert and delete some characters.

Longest Common Subsequence (LCS): is the sequence of positions in $\boldsymbol{v}$: $1 \leq i_1 < \cdots < i_t \leq m$ and $\boldsymbol{w}$: $1 \leq j_1 < \cdots < j_t \leq n$ such that the i_t-th character of $\boldsymbol{v}$ = j_t-th character of $\boldsymbol{w}$ and t is maximal.
Every common subsequence is a path in a 2-D Manhattan grid:

- diagonals are alignable characters (matches)
- others identify insertions/deletions

Solution: find the path that maximize the number of diagonals (we want to maximize the matches).

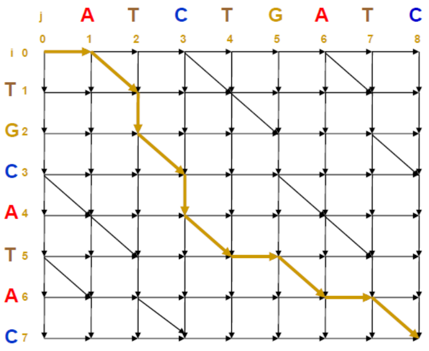


i coords:	0	1	2	2	3	3	4	5	6	7	8
elements of *v*		A	T	--	C	--	T	G	A	T	C
elements of *w*		--	T	G	C	A	T	--	A	--	C
j coords:	0	0	1	2	3	4	5	5	6	6	7

(0,0)→(1,0)→(2,1)→(2,2)→(3,3)→(3,4)→(4,5)→(5,5)→(6,6)→(7,6)→(8,7)

Matches shown in red

positions in *v*: 2 < 3 < 4 < 6 < 8
positions in *w*: 1 < 3 < 5 < 6 < 7



$O(nm)$

```
LCS(v, w)
for $i \leftarrow 0$ to $n$
    $s_{i,0} \leftarrow 0$
for $j \leftarrow 1$ to $m$
    $s_{0,j} \leftarrow 0$
for $i \leftarrow 1$ to $n$
    for $j \leftarrow 1$ to $m$
        $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1}+1, & \text{if } v_i = w_j \end{cases}$
        $b_{i,j} \leftarrow \begin{cases} \text{"}\uparrow\text{"} & \text{if } s_{i,j} = s_{i-1,j} \\ \text{"}\leftarrow\text{"} & \text{if } s_{i,j} = s_{i,j-1} \\ \text{"}\nwarrow\text{"}, & \text{if } s_{i,j} = s_{i-1,j-1}+1 \end{cases}$
return $(s_{n,m}, \mathbf{b})$
```

```
PRINTLCS(b, v, i, j)
if $i = 0$ or $j = 0$
    return
if $b_{i,j} = \text{"}\nwarrow\text{"}$
    PRINTLCS(b, v, $i-1$, $j-1$)
    print $v_i$
else
    if $b_{i,j} = \text{"}\uparrow\text{"}$
        PRINTLCS(b, v, $i-1$, $j$)
    else
        PRINTLCS(b, v, $i$, $j-1$)
```

Edit distance $d(\boldsymbol{v}, \boldsymbol{w})$ = MIN number of elementary operations (insertions, deletions, and substitutions) to transform $\boldsymbol{v}$ in $\boldsymbol{w}$.
Is calculated according to the initial conditions $d_{i,0} = i, d_{0,j} = j$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$ and the following recurrence:

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

3.2. Sequence Alignment

3.2.1. Global Alignment

LCS allows only insertions and deletions (no mismatches), awards 1 for matches and does not penalize indels.

Simplest scoring schema:

- $+1$: match premium
- μ: mismatch penalty
- σ: indel penalty

Global Alignment: Find the best alignment between two strings under a given scoring matrix. $O(n^2)$

- **Input**: Strings $\boldsymbol{v}, \boldsymbol{w}$ and a scoring matrix $\boldsymbol{\delta}$
- **Output**: An alignment of $\boldsymbol{v}$ and $\boldsymbol{w}$ whose score is maximal among all possible alignments of $\boldsymbol{v}$ and $\boldsymbol{w}$.

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

$\text{Score} = \#\text{matches} - \mu * \#\text{mismatches} - \sigma * \#\text{indels}$

LCS problem is the Global Alignment problem with the parameters $\mu = 0, \sigma = 0$.

Scoring Matrices

Scoring techniques:

- Identity: matches/sequence size
- Conservation: combine matches, mismatches and indels

Scoring matrix $\boldsymbol{\delta}$ size= $(|\Sigma| + 1) \times (|\Sigma| + 1)$ (where +1 is for the gap character "-")

Scoring matrices are created based on biological evidence. Alignments can be thought of as two sequences that differ due to mutations. Some of these mutations have little effect on the protein's function, therefore some penalties in δ will be less harsh than others.

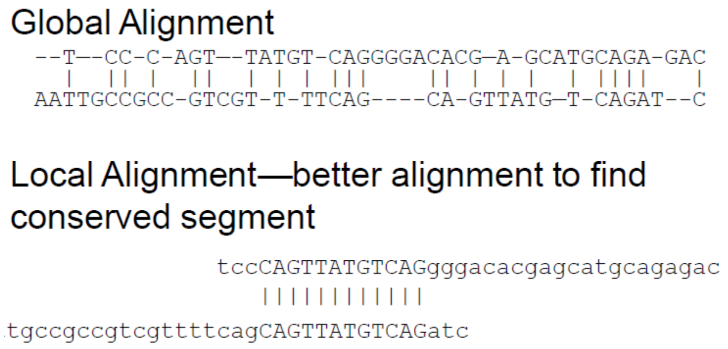
Common Matrices for protein sequence comparison:

- point accepted mutations (**PAM**): 1 PAM = 1% of all amino acid positions are changed.
- Block substitution (**BLOSUM**): Scores derived from observations of the frequencies of substitutions in blocks of local alignments in related proteins.

Local vs. Global Alignment

- **Global**: tries to find the longest path between vertices $(0,0)$ and (n, m) in the edit graph.
- **Local**: tries to find the longest path among paths between arbitrary vertices (i, j) and (i', j') in the edit graph.

In the edit graph with negatively scored edges, Local Alignment may score higher than Global Alignment



Global Alignment

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C
```

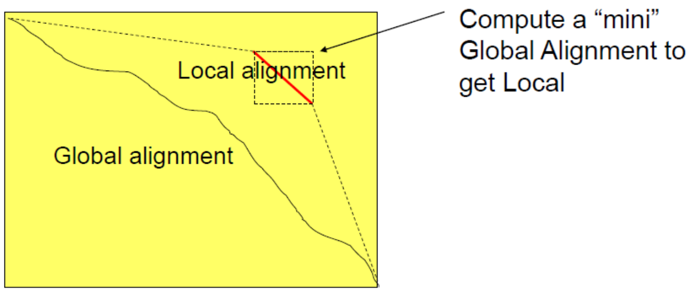
Local Alignment—better alignment to find conserved segment

```
               tccCAGTTATGTCAGggacacgagcatgcagagac
                  ||||||||||||
tgccgccgtcgttttcagCAGTTATGTCAGatc
```

3.2.2. Local Alignment

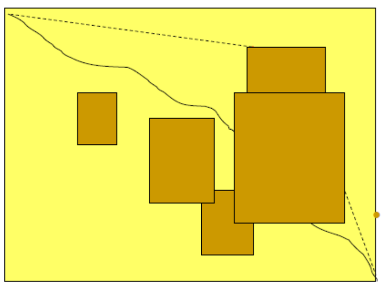
Local Alignment: Find the best local alignment between two strings. $O(n^4)$

- **Input**: Strings v, w and scoring matrix δ
- **Output**: Alignment of substrings of v and w whose alignment score is maximum among all possible alignment of all possible substrings.



Solution: with dynamic programming $O(n^2)$

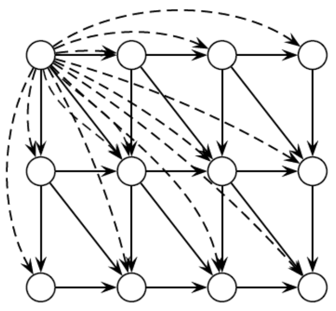
Imagine that exist another arc, with 0 weight, that from the origin go to every other node. So we can go from the origin to any other vertex without penalties.



- Long run time *O(n⁴)*:

  - In the grid of size *n x n* there are ~n^2 vertices *(i,j)* that may serve as a source.

  - For each such vertex computing alignments from *(i,j)* to *(i',j')* takes *O(n²)* time.
- This can be remedied by giving free rides



$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Notice there is only this change from the original recurrence of a Global Alignment

K-best local alignments: Several local alignments might have biological significance

Output best k non overlapping alignments:

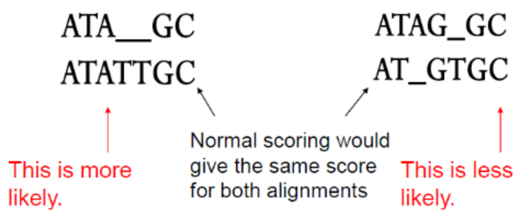
- A particular local alignment can be specified by the edges that it uses during the traceback.
- Two local alignments are said to be disjoint if they do not use any of the same edges.

A Simple algorithm O(knm):

- ForbiddenEdges= []
- For i= 1 .. k:
  - Fill in the dynamic programming matrix, disallowing the use of any ForbiddenEdges
  - Traceback, adding every visited edge to ForbiddenEdges

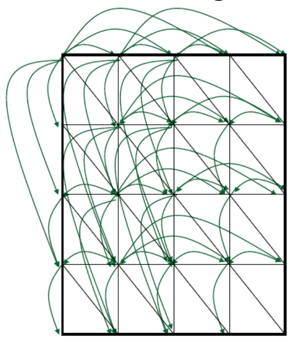
3.2.3. Alignment with Affine Gap Penalties

In nature, a series of k indels often come as a single event rather than a series of k single nucleotide events. So applying the penalty σ k consequent times, it's too severe.



Gap: contiguous sequence of spaces in one of the rows
Score for a gap of length x is: $-(\rho + \sigma x)$
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.

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 $-(\rho + \sigma x)$



There are many such edges!

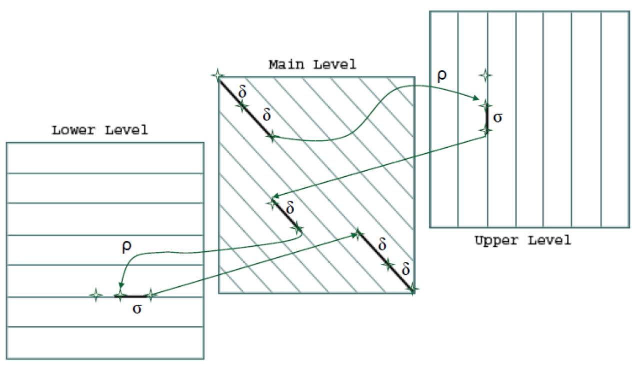
Adding them to the graph increases the running time of the alignment algorithm by a factor of ***n*** (where ***n*** is the number of vertices)

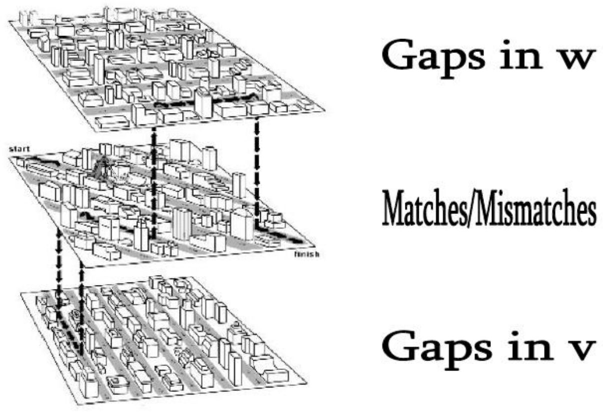
So the complexity increases from O(n^2) to O(n^3)

So we use 3 Manhattan grids: each has arcs only in 1 direction

- The main level is for diagonal edges. Extends matches and mismatches.
- The lower level is for horizontal edges. Creates/extends gaps in sequence v.
- The upper level is for vertical edges. Creates/extends gaps in the sequence w.

Jumping penalty from the main level to either the upper level or the lower level $-\rho - \sigma$
Gap extension penalty for each continuation on a level other than the main level $-\sigma$





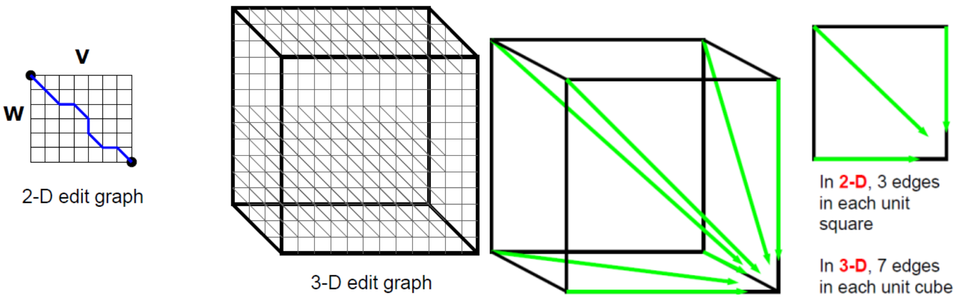
$$\overset{\downarrow}{s}_{i,j} = \max \begin{cases} \overset{\downarrow}{s}_{i-1,j} - \sigma & \text{Continue Gap in } w \text{ (deletion)} \\ s_{i-1,j} - (\rho+\sigma) & \text{Start Gap in } w \text{ (deletion): from middle} \end{cases}$$

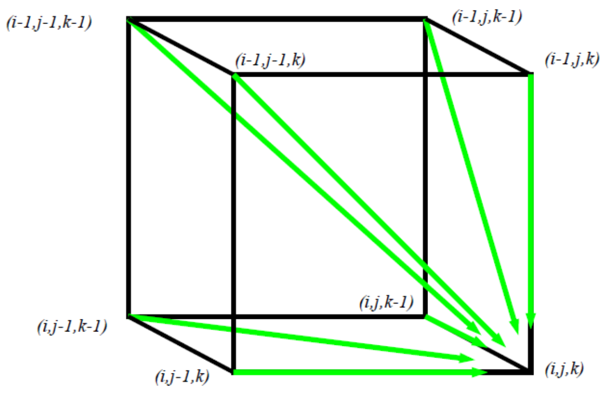
$$\overrightarrow{s}_{i,j} = \max \begin{cases} \overrightarrow{s}_{i,j-1} - \sigma & \text{Continue Gap in } v \text{ (insertion)} \\ s_{i,j-1} - (\rho+\sigma) & \text{Start Gap in } v \text{ (insertion):from middle} \end{cases}$$

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) & \text{Match or Mismatch} \\ \overset{\downarrow}{s}_{i,j} & \text{End deletion: from top} \\ \overrightarrow{s}_{i,j} & \text{End insertion: from bottom} \end{cases}$$

3.3. Multiple Alignment

Alignment between more than 2 sequences.
Alignment of k sequences is represented as a k-row matrix
The path is k-dimension Manhattan grid space.
Run time: $O(2^k n^k)$ (k sequences each of length n)


v
w
2-D edit graph
3-D edit graph
In 2-D, 3 edges in each unit square
In 3-D, 7 edges in each unit cube



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

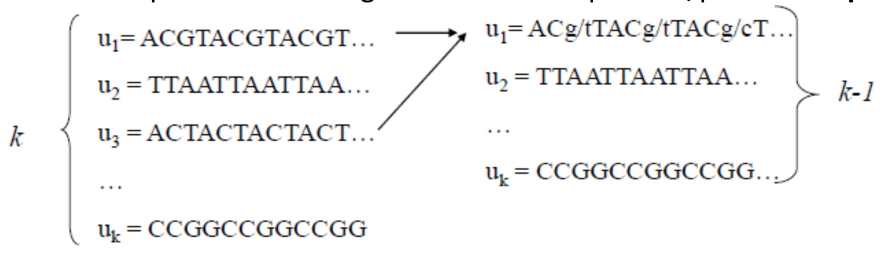
$\delta(x, y, z)$ is an entry in the 3-D scoring matrix

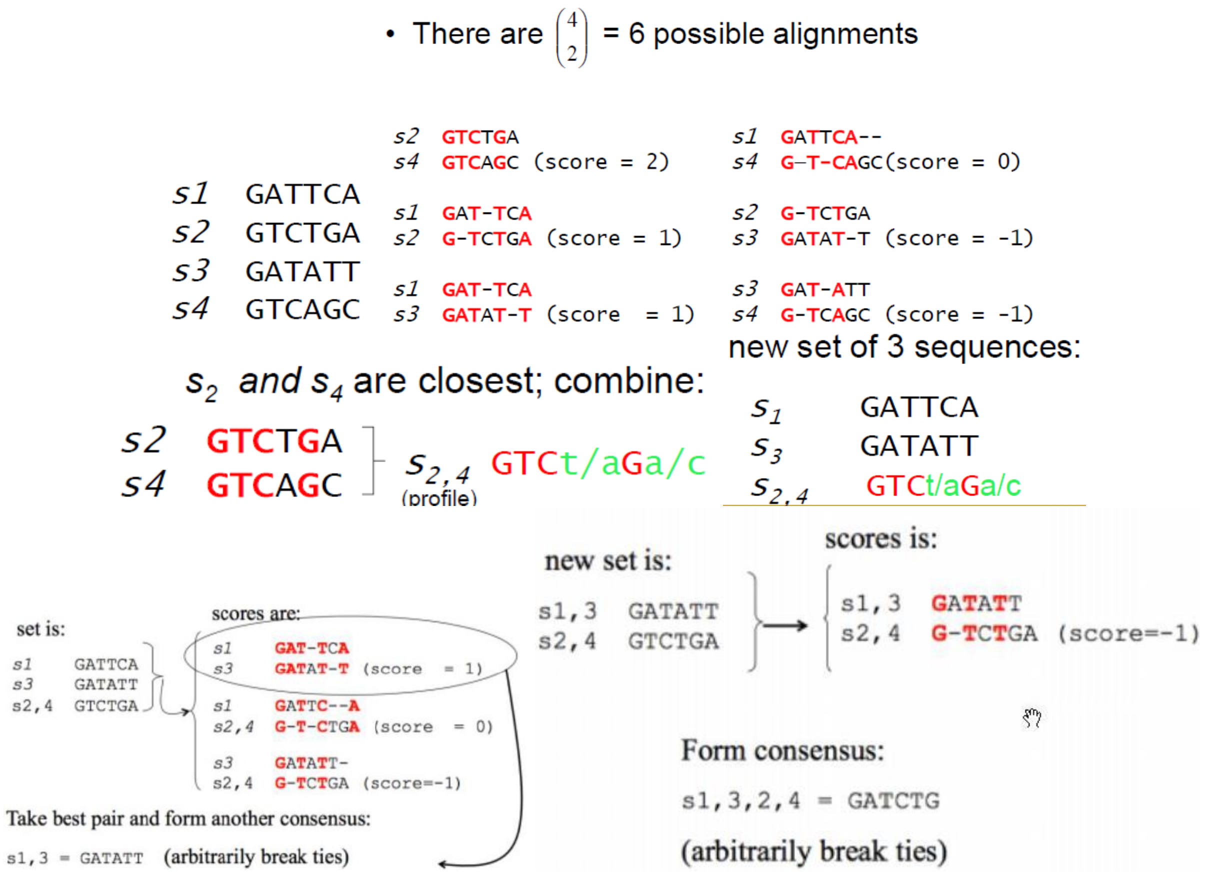
Run time for the exact solution is impractical.

Every multiple alignment induces pairwise alignments.
Reverse Problem: Constructing Multiple Alignment from Pairwise Alignments. Not always possible.
From an optimal multiple alignment, we can infer pairwise alignments between all pairs of sequences, but they are not necessarily optimal
It is difficult to infer a ``good" multiple alignment from optimal pairwise alignments between all sequences

Aligning alignments: do it by aligning the profiles.

Greedy Approach: Choose most similar pair of strings and combine into a profile, in this way reducing alignment of k sequences to an alignment of k-1 sequences/profiles. **Repeat**





- There are $\binom{4}{2}$ = 6 possible alignments

```
s1  GATTCA
s2  GTCTGA
s3  GATATT
s4  GTCAGC

s2  GTCTGA
s4  GTCAGC (score = 2)

s1  GAT-TCA
s2  G-TCTGA (score = 1)

s1  GAT-TCA
s3  GATAT-T (score  = 1)

s1  GATTCA--
s4  G—T-CAGC(score = 0)

s2  G-TCTGA
s3  GATAT-T (score = -1)

s3  GAT-ATT
s4  G-TCAGC (score = -1)
```

s_2 *and* s_4 are closest; combine:

```
s2  GTCTGA
s4  GTCAGC   }  s2,4 GTCt/aGa/c
                (profile)
```

new set of 3 sequences:

```
s1     GATTCA
s3     GATATT
s2,4   GTCt/aGa/c
```

set is:

```
s1     GATTCA
s3     GATATT
s2,4   GTCTGA
```

scores are:

```
s1     GAT-TCA
s3     GATAT-T  (score  = 1)

s1     GATTC--A
s2,4   G-T-CTGA  (score  = 0)

s3     GATATT-
s2,4   G-TCTGA  (score=-1)
```

Take best pair and form another consensus:

s1,3 = GATATT (arbitrarily break ties)

new set is:

```
s1,3   GATATT
s2,4   GTCTGA
```

scores is:

```
s1,3   GATATT
s2,4   G-TCTGA  (score=-1)
```

Form consensus:

s1,3,2,4 = GATCTG

(arbitrarily break ties)

Progressive alignment: is a variation of greedy algorithm with a somewhat more intelligent strategy for choosing the order of alignments.
Works well for close sequences, but deteriorates for distant sequences.

- Gaps in consensus string are permanent
- Use profiles to compare sequences

Star approach: Given k sequences

- Pick one sequence x_c as the center
- For each $x_i \neq x_c$ determine an optimal alignment between x_i and x_c
- Merge pairwise alignments
- Return: multiple alignment resulting from aggregate

Two possible approaches:

- Try each sequence as a center, return the best multiple alignment
- Compute all pairwise alignments and select the string x_c that maximizes: $\sum_{x_i \neq x_c} sim\ (x_i, x_c)$

Tree approach: organize multiple sequence alignment using a guide tree

- Leaves represent sequences
- Internal nodes represent alignments
- Determine alignments from the bottom of the tree upward
- Return the multiple alignment represented by the root of the tree

Scoring: techniques to evaluate the quality of a multiple alignment

- **Number of matches** (multiple longest common subsequence score): A column is a “match” if all the letters in the column are the same. Only good for very similar sequences.
- **Entropy score:** idea: try to minimize the entropy of each column. Columns that can be described using few bits are good.
  Entropy for a multiple alignment is the sum of entropies of its columns:

$$\sum_{\text{over all columns}} \sum_{X=A,T,G,C} p_X \log p_X$$

$$entropy\begin{pmatrix} A \\ A \\ A \\ A \end{pmatrix} = 0$$ Best case

column entropy:
$$-(\ p_A \log p_A + p_C \log p_C + p_G \log p_G + p_T \log p_T)$$

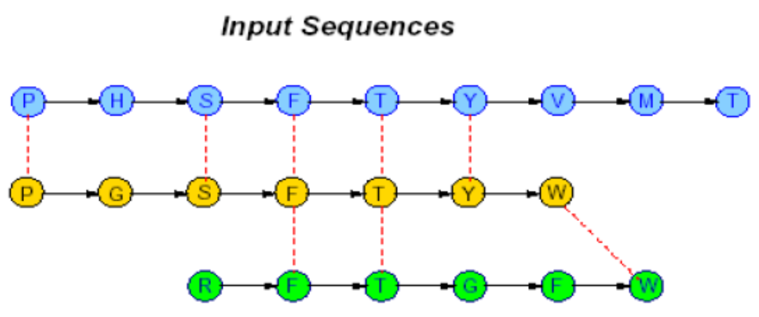
Worst case $$entropy\begin{pmatrix} A \\ T \\ G \\ C \end{pmatrix} = -\sum \frac{1}{4}\log\frac{1}{4} = -4(\frac{1}{4} * -2) = 2$$

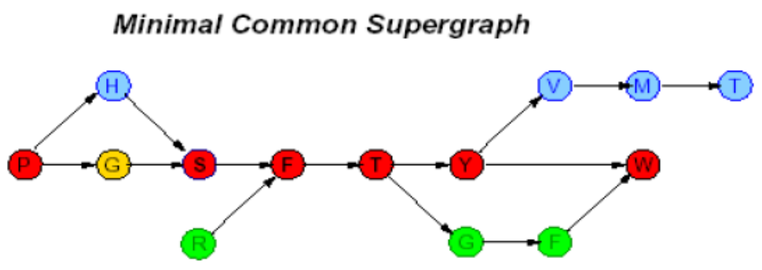
A	A	A
A	C	C
A	C	G
A	C	T

- Column 1 = -[1*log(1) + 0*log0 + 0*log0 +0*log0]
  = 0
- Column 2 = -[($^1/_4$)*log($^1/_4$) + ($^3/_4$)*log($^3/_4$) + 0*log0 + 0*log0]
  = -[($^1/_4$)*(-2) + ($^3/_4$)*(-.415)] = +0.811
- Column 3 = -[($^1/_4$)*log($^1/_4$)+($^1/_4$)*log($^1/_4$)+($^1/_4$)*log($^1/_4$) +($^1/_4$)*log($^1/_4$)]
  = 4* -[($^1/_4$)*(-2)] = +2.0
- Alignment Entropy = 0 + 0.811 + 2.0 = +2.811

- **Sum of pairs (SP Score)**
  - From a multiple alignment, we can infer pairwise alignments between all sequences, but they are not necessarily optimal. A 3D alignment can be projected onto the 2D plane to represent an alignment between a pair of sequences.
  - $s^*(a_i, a_j)$: score of this suboptimal pairwise alignment
  - SP-Score: $s(a_1, \ldots, a_k) = \sum_{i,j} s^*(a_i, a_j)$

Alignment as a Graph:





4. Divide & Conquer Algorithms

Divide problem into sub problems
Conquer by solving sub problems recursively. If the sub problems are small enough, solve them directly.
Combine the solutions of sub problems into a solution of the original problem (tricky part)

Divide and Conquer Approach to LCS:

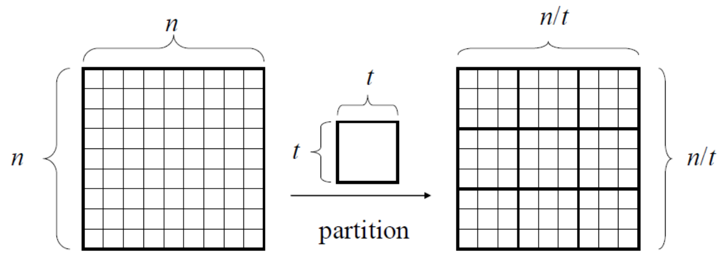
```
Path(source, sink)
if(source & sink are in consecutive columns)
  output the longest path from source to sink
else
  middle ← middle vertex between source & sink
  Path(source, middle)
  Path(middle, sink)
```

To find the middle vertex ……………………………………

Block Alignment & Four Russians Speedup

Partition the nxn grid into blocks of size txt

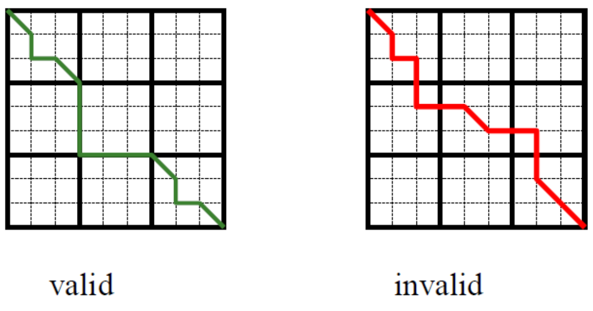
each sequence is sectioned off into chunks, each of length t
Sequence $\boldsymbol{u} = u_1 \dots u_n$ becomes $|u_1 \dots u_t|\ |u_{t+1} \dots u_{2t}| \dots |u_{n-t+1} \dots u_n|$
Sequence $\boldsymbol{v} = v_1 \dots v_n$ becomes $|v_1 \dots v_t|\ |v_{t+1} \dots v_{2t}| \dots |v_{n-t+1} \dots v_n|$



Block alignment of sequences $\boldsymbol{u}$ and $\boldsymbol{v}$:

- An entire block in u is aligned with an entire block in v
- An entire block is inserted
- An entire block is deleted

Block path: a path that traverses every txt square through its corners

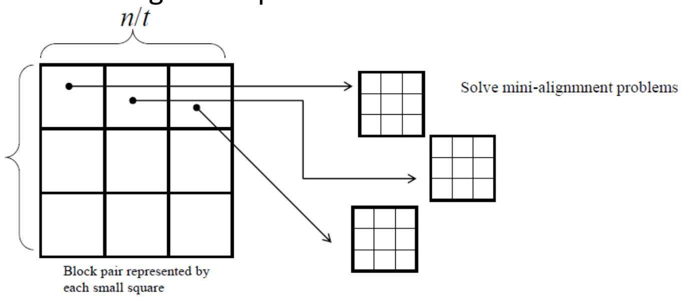


Block Alignment Problem: Find the longest block path through an edit graph.

- **Input**: Two sequences, $\boldsymbol{u}$ and $\boldsymbol{v}$ partitioned into blocks of size t.
- **Output**: The block alignment of u and v with the maximum score (i.e., the longest block path through the edit graph).

Solution: compute alignment score $ß_{i,j}$ for each pair of blocks $|u_{(i-1)*t+1} \dots u_{i*t}|$ and $|v_{(j-1)*t+1} \dots v_{j*t}|$

For each block pair, solve a mini alignment problem of size txt



n/t
Solve mini-alignment problems
Block pair represented by each small square

Optimal block alignment score between the first i blocks of $\boldsymbol{u}$ and first j blocks of $\boldsymbol{v}$: $O(n^2)$

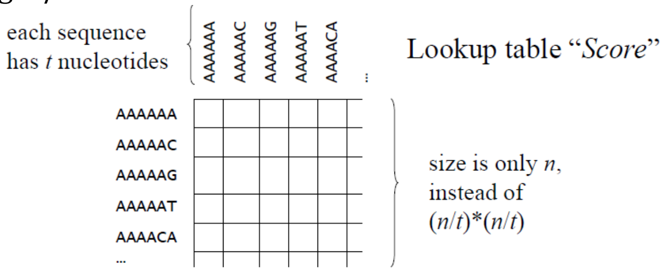
$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma_{\text{block}} \\ s_{i,j-1} - \sigma_{\text{block}} \\ s_{i-1,j-1} - \beta_{i,j} \end{cases}$$

σ_{block} is the penalty for inserting or deleting an entire block

$\beta_{i,j}$ is score of pair of blocks in row i and column j.

Four Russians Technique: speeds up the block alignment by using a lookup table Score to eliminate the time of computing $ß_{i,j}$.

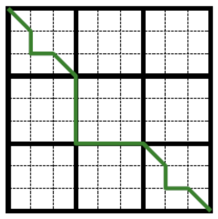
Lookup table: stores the precomputed the scores $ß_{i,j}$ for all possible pair of sequences. Its size is $4^t \times 4^t$, if we set $t = \log n / 4$ then the size is n

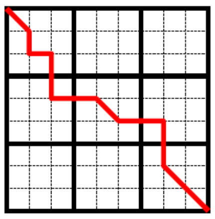


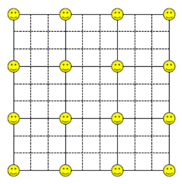
$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} + Score(i\text{th block of v}, j\text{th block of u}) \end{cases}$$

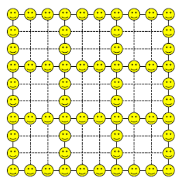
Final time complexity: $O(n^2 / \log n)$

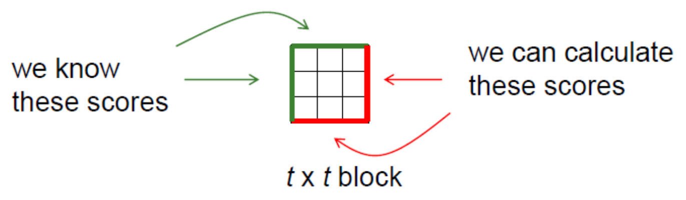
Problem: In block alignment, we only care about the corners of the blocks. In LCS, we care about all points on the edges of the blocks, because those are points that the path can traverse.


block alignment

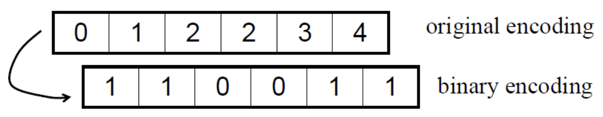

longest common subsequence


block alignment has $(n/t)*(n/t) = (n^2/t^2)$ points of interest


LCS alignment has $O(n^2/t)$ points of interest



- Build a lookup table for all possible values of the four variables:
  1. all possible scores for the first row $s_{*,j}$
  2. all possible scores for the first column $s_{*,j}$
  3. substring of sequence *u* in this block (4^t possibilities)
  4. substring of sequence *v* in this block (4^t possibilities)
- For each quadruple we store the value of the score for the last row and last column.
- This will be a huge table, but we can eliminate alignments scores that don't make sense
- Instead of recording numbers that correspond to the index in the sequences *u* and *v*, we can use binary to encode the differences between the alignment scores



- Alignment scores in LCS are monotonically increasing, and adjacent elements can't diffe
  by more than 1
- Example: 0,1,2,2,3,4 is ok; 0,1,**2**,**4**,5,8, is not because 2 and 4 differ by more than 1 (and so do 5 and 8)
- Therefore, we only need to store quadruples whose scores are monotonically increasing and differ by at most 1
- 2^t possible scores (*t* = size of blocks)
- 4^t possible strings
  - Lookup table size is $(2^t * 2^t)*(4^t * 4^t) = 2^{6t}$
- Let *t* = (log*n*)/4;
  - Table size is: $2^{6((\log n)/4)} = n^{(6/4)} = n^{(3/2)}$
- Time = O([n^2/t^2]*log*n*)
- O([$n^2/\{\log n\}^2$]*log*n*) ≤ O($n^2/\log n$)

5.Greedy Algorithms and Genome Rearrangements

Reversal $p(i,j)$ reverses (flips) the elements from i to j in π
Gene order is represented by a permutation π

$$\begin{tikzcd}
\pi = \pi_1 \text{------} \pi_{i-1} \underline{\pi_i \pi_{i+1} \text{------} \pi_{j-1} \pi_j} \pi_{j+1} \text{-----} \pi_n \arrow[d, "\rho(i,j)"'] \\
\pi_1 \text{------} \pi_{i-1} \pi_j \pi_{j-1} \text{------} \pi_{i+1} \pi_i \pi_{j+1} \text{-----} \pi_n
\end{tikzcd}$$

$$\begin{tikzcd}
\pi = 1\ 2\ \underline{3\ 4\ 5}\ 6\ 7\ 8 \arrow[d, "\rho(3,5)"'] \\
1\ 2\ 5\ 4\ 3\ 6\ 7\ 8
\end{tikzcd}$$

Reversal Distance Problem: Given two permutations, find the shortest series of reversals that transforms one into another.

- **Input**: Permutations π and σ
- **Output**: A series of reversals $\rho_1, \dots, \rho_t$ transforming π into σ such that t is minimum

Sorting by Reversals Problem: Given a permutation, find a shortest series of reversals that transforms it into the identity permutation (1 2 … n)

- **Input**: Permutation $\boldsymbol{\pi}$
- **Output**: A series of reversals $\rho_1, \dots, \rho_t$ transforming π into the identity permutation such that t is minimum.

Example :

$$\begin{array}{rl}
\boldsymbol{\pi} = & \underline{3\ 4}\ 2\ 1\ 5\ 6\ 7\ 10\ 9\ 8 \\
 & 4\ 3\ 2\ 1\ 5\ 6\ 7\ \underline{10\ 9\ 8} \\
 & \underline{4\ 3\ 2\ 1}\ 5\ 6\ 7\ 8\ 9\ 10 \\
 & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10
\end{array}$$

So $d(\boldsymbol{\pi}) = 3$

Pancake Flipping Problem: Given a stack of n pancakes, what is the minimum number of flips to rearrange them into perfect stack?

- **Input**: Permutation $\boldsymbol{\pi}$
- **Output**: A series of prefix reversals $\rho_1, \dots, \rho_t$ transforming $\boldsymbol{\pi}$ into the identity permutation such that t is minimum

```
SimpleReversalSort($\pi$)
1 for  $i \leftarrow 1$ to $n - 1$
2   $j \leftarrow$ position of element $i$ in $\pi$ (i.e., $\pi_j = i$)
3   if $j \neq i$
4      $\pi \leftarrow \pi * \rho(i, j)$
5      output $\pi$
6   if $\pi$ is the identity permutation
7     return
```

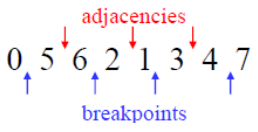
Not optimal.

Approximation ratio: $A(\pi)/OPT(\pi)$

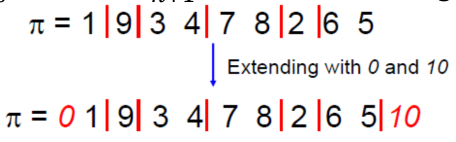
- $A(\pi)$ solution produced by algorithm A
- $OPT(\pi)$ optimal solution of the problem

Minimization algorithm: $\max_{|\pi|=n} A(\pi)/OPT(\pi)$

Adjacency: a pair of adjacent elements that are consecutive. Eg. (2,1) or (3,4)
Breakpoint: a pair of adjacent elements that are not consecutive. Eg. (6,2) or (1,3)
$b(\pi)$: # of breakpoints



Extending Permutations: put $\pi_0 = 0$ and $\pi_{n+1} = n+1$ at the beginning and at the end.


π = 1 | 9 | 3 4 | 7 8 | 2 | 6 5
Extending with 0 and 10
π = 0 1 | 9 | 3 4 | 7 8 | 2 | 6 5 | 10

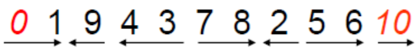
Each reversal eliminates at most 2 breakpoints.
This implies: reversal distance ≥ #breakpoints / 2

Sorting by Reversals: A Better Greedy Algorithm

```
BreakPointReversalSort($\pi$)
1 while $b(\pi) > 0$
2   Among all possible reversals,
    choose reversal $\rho$ minimizing $b(\pi \cdot \rho)$
3   $\pi \leftarrow \pi \cdot \rho(i, j)$
4   output $\pi$
5 return
```

Strip: an interval between two consecutive breakpoints in a permutation

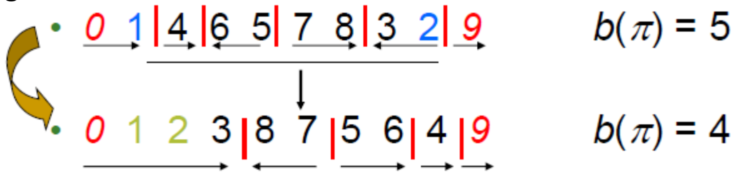
- Decreasing strip: strip of elements in decreasing order (e.g. 6 5 and 3 2).
- A single element strip is considered decreasing with exception of the strips with 0 and n+1



Theorem 1: If permutation π contains at least one decreasing strip, then there exists a reversal ρ which decreases the number of breakpoints.

Best reversal:

- Choose decreasing strip with the smallest element k in π (k = 2 in this case)
- Find $k-1$ in the permutation
- Reverse the segment between k and $k-1$



If there is no decreasing strip, create one by reversing a increasing strip.

```
ImprovedBreakpointReversalSort($\pi$)
1 while $b(\pi) > 0$
2   if $\pi$ has a decreasing strip
3     Among all possible reversals, choose reversal $\rho$
          that minimizes $b(\pi \cdot \rho)$
4   else
5     Choose a reversal $\rho$ that flips an increasing strip in $\pi$
6   $\pi \leftarrow \pi \cdot \rho$
7   output $\pi$
8 return
```

6. Molecular Evolution

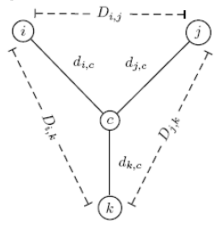
Evolutionary Tree:

- leaves represent existing species
- internal vertices represent ancestors
- root represents the oldest evolutionary ancestor

Tree distance between i and j $d_{ij}(T)$: length of a path between leaves i and j
Distance Matrix D $n \times n$: each cell is the edit distance between a gene (DNA string) in species i and species j, where the gene of interest is sequenced for all n species.

Given n species, Evolution of these genes is described by a tree that we don't know.
We need an algorithm to construct a tree that best fits the distance matrix D_{ij}
Fitting means: $D_{ij} = d_{ij}(T)$

Reconstructing a 3 Leaved Tree: given the 3x3 matrix



Observe:

$d_{ic} + d_{jc} = D_{ij}$

$d_{ic} + d_{kc} = D_{ik}$

$d_{jc} + d_{kc} = D_{jk}$

$$\mathbf{d_{ic} = (D_{ij} + D_{ik} - D_{jk})/2}$$

arly,

$$\mathbf{d_{jc} = (D_{ij} + D_{jk} - D_{ik})/2}$$

====> $$\mathbf{d_{kc} = (D_{ki} + D_{kj} - D_{ij})/2}$$

Trees with >3 Leaves:

- $O(n^2)$ equations
- $2n - 3$ variables (edges)
- We have more constrains than variables => not always possible to solve for $n > 3$
- The matrix must be ADDITIVE

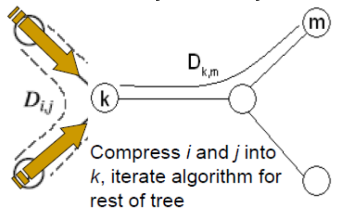
Distance Based Phylogeny Problem: Reconstruct an evolutionary tree from a distance matrix

- **Input**: $n \times n$ distance matrix D_{ij}
- **Output**: weighted tree T with n leaves fitting D

If D is additive, this problem has a solution and there is a simple algorithm to solve it.

Using Neighboring Leaves to Construct the Tree

- Find neighboring leaves i and j with parent k
- Remove the rows and columns of i and j
- Add a new row and column corresponding to k, where the distance from k to any other leaf m can be computed as: $D_{km} = \left(D_{im} + D_{jm} - D_{ij}\right)/2$



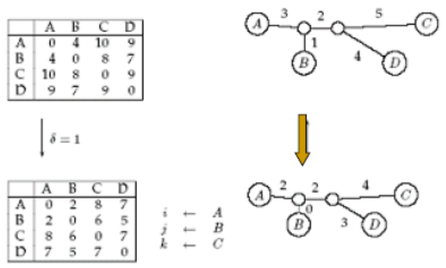
m
$D_{k,m}$
$D_{i,j}$
k
Compress i and j into k, iterate algorithm for rest of tree

Neighbor Joining Algorithm: Finds a pair of leaves that are close to each other but far from other leaves: implicitly finds a pair of neighboring leaves.
Works well for additive and other non-additive matrices.

6.1. Additive Phylogeny Algorithm

Degenerate Triples: 3 distinct elements $1 \leq i, j, k \leq n$ where $D_{ij} + D_{jk} = D_{ik}$
i, j, k lies on the evolutionary path from i to k (or is attached to this path by an edge of length 0).
If distance matrix D

- **has a degenerate triple** i, j, k then j can be "removed" from D thus reducing the size of the problem.
- **does not have** a degenerate triple i, j, k then we "create" a degenerative triple in D by shortening all hanging edges by the same amount δ (so that all pair wise distances in the matrix are reduced by 2δ) until a degenerate triple is found.
- The attachment point for j can be recovered in the reverse transformations by saving D_{ij} for each collapsed leaf.



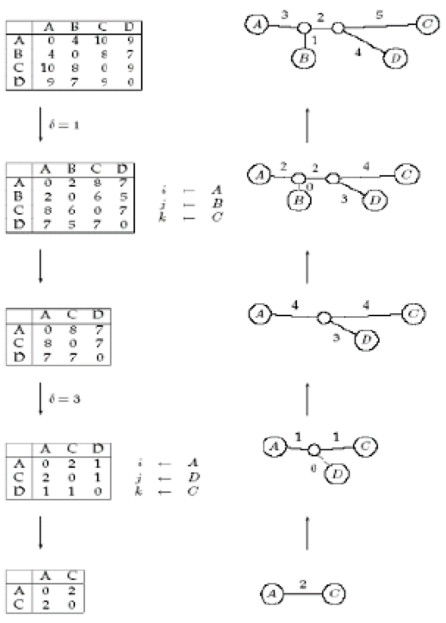
	A	B	C	D
A	0	4	10	9
B	4	0	8	7
C	10	8	0	9
D	9	7	9	0

$\delta = 1$

	A	B	C	D
A	0	2	8	7
B	2	0	6	5
C	8	6	0	7
D	7	5	7	0

$i \leftarrow A$
$j \leftarrow B$
$k \leftarrow C$

Additive Phylogeny Algorithm



	A	B	C	D
A	0	4	10	9
B	4	0	8	7
C	10	8	0	9
D	9	7	9	0

$\delta = 1$

	A	B	C	D
A	0	2	8	7
B	2	0	6	5
C	8	6	0	7
D	7	5	7	0

$i \leftarrow A$
$j \leftarrow B$
$k \leftarrow C$

	A	C	D
A	0	8	7
C	8	0	7
D	7	7	0

$\delta = 3$

	A	C	D
A	0	2	1
C	2	0	1
D	1	1	0

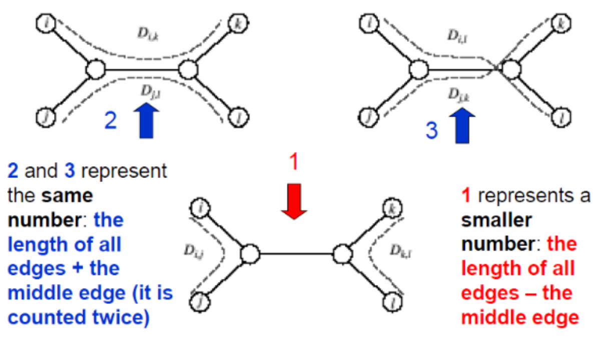
$i \leftarrow A$
$j \leftarrow D$
$k \leftarrow C$

	A	C
A	0	2
C	2	0

```
AdditivePhylogeny($D$)
  if $D$ is a 2 x 2 matrix
    $T$ = tree of a single edge of length $D_{1,2}$
    return $T$
  if $D$ is non-degenerate
    $\delta$ = trimming parameter of matrix $D$
    for all $1 \leq i \neq j \leq n$
      $D_{ij} = D_{ij} - 2\delta$
  else
    $\delta = 0$
  Find a triple $i, j, k$ in $D$ such that $D_{ij} + D_{jk} = D_{ik}$
  $x = D_{ij}$
  Remove $j^{th}$ row and $j^{th}$ column from $D$
  $T$ = AdditivePhylogeny($D$)
  Add a new vertex $v$ to $T$ at distance $x$ from $i$ to $k$
  Add $j$ back to $T$ by creating an edge $(v,j)$ of length 0
  for every leaf $l$ in $T$
    if distance from $l$ to $v$ in the tree $\neq D_{l,j}$
      output "matrix is not additive"
      return
  Extend all "hanging" edges by length $\delta$
  return $T$
```

The Four Point Condition: efficient additivity check.
Let $1 \leq i, j, k, l \leq n$ be 4 distinct leaves in a tree.
The condition is satisfied If 2 of the following sums are the same and the third is smaller:

- $D_{ij} + D_{kl}$
- $D_{ik} + D_{jl}$
- $D_{il} + D_{jk}$



2 and 3 represent the same number: the length of all edges + the middle edge (it is counted twice)

1 represents a smaller number: the length of all edges – the middle edge

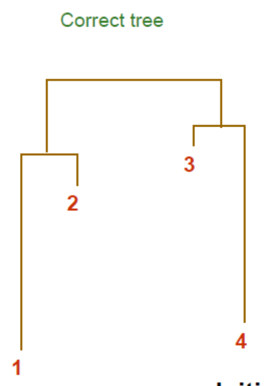
An $n \times n$ matrix D is additive if and only if the four-point condition holds for every quartet $1 \leq i, j, k, l \leq n$

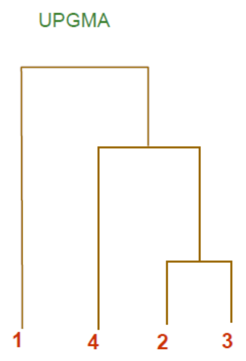
6.2. NOT Additive distance matrix

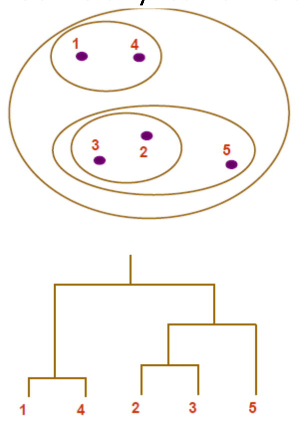
Least Squares Distance Phylogeny Problem: If the distance matrix D is NOT additive, then we look for a tree T that approximates D the best. NP-HARD. $\sum_{i,j}\left(d_{ij}(T) - D_{ij}\right)^2$

UPGMA is a clustering algorithm that:

- computes the distance between clusters using average pairwise distance
- assigns a height to every vertex in the tree, effectively assuming the presence of a molecular clock and dating every vertex.
- Problem: produces an ultrametric tree: the distance from the root to any leaf is the same.







Initialization:

Assign each x_i to its own cluster C_i

Define one leaf per sequence, each at height 0

Iteration:

Find two clusters C_i *and* C_j such that d_{ij} is min

Let $C_k = C_i \cup C_j$

Add a vertex connecting C_i, C_j *and* place it at height $d_{ij}/2$

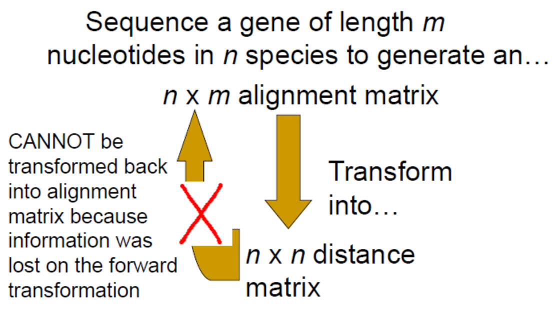
Delete C_i *and* C_j

Termination:

When a single cluster remains

6.3. Weighted Small Parsimony Problem

Alignment Matrix vs. Distance Matrix


Sequence a gene of length *m* nucleotides in *n* species to generate an…
n x *m* alignment matrix
CANNOT be transformed back into alignment matrix because information was lost on the forward transformation
Transform into…
n x *n* distance matrix

Character Based Tree Reconstruction: determine what character strings at internal nodes would best explain the character strings for the n observed species

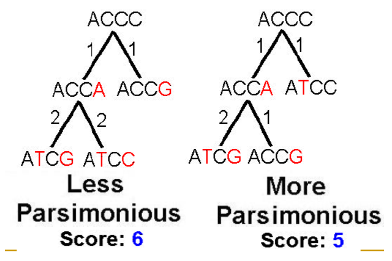
Use the nxm alignment matrix (n = # species, m = #characters) directly instead of using distance matrix.

Characters may be nucleotides. Others may be the # of eyes or legs or the shape of a beak or a fin.

Parsimony score of a tree: sum of the lengths (weights) of the edges. Where the length of an edge in the tree is the Hamming distance.

Assumes observed character differences resulted from the fewest possible mutations.

Seeks the tree that yields lowest possible parsimony score sum of cost of all mutations found in the tree.



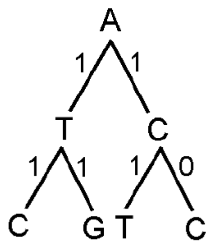
Small Parsimony Problem

- **Input**: Tree T with each leaf labeled by an m character string.
- **Output**: Labeling of internal vertices of the tree T minimizing the parsimony score.

Assume that every leaf is labeled by a single character, because the characters in the string are independent.

Weighted Small Parsimony Problem: Input includes a $k \times k$ scoring matrix describing the cost of transformation of each of k states into another one.

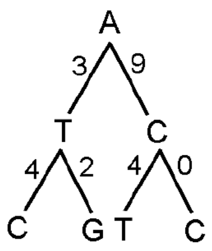
- **Input**: Tree T with each leaf labeled by elements of a k letter alphabet and a $k \times k$ scoring matrix δ_{ij}
- **Output**: Labeling of internal vertices of the tree T minimizing the weighted parsimony score.



Small Parsimony Scoring Matrix

	A	T	G	C
A	0	1	1	1
T	1	0	1	1
G	1	1	0	1
C	1	1	1	0

Small Parsimony Score: 5



Weighted Parsimony Scoring Matrix

	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

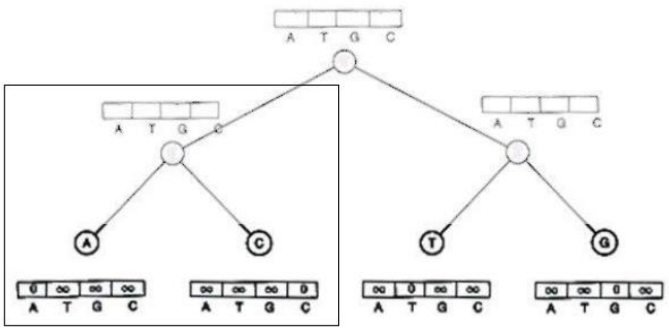
Weighted Parsimony Score: 22

6.3.1. Sankoff's Algorithm

The score at each vertex is based on scores of its children: $s_t(parent) = \min\limits_i\{s_i(leftchild) + \delta_{i,t}\} + \min\limits_j\{s_j(rightchild) + \delta_{j,t}\}$

Begin at leaves:

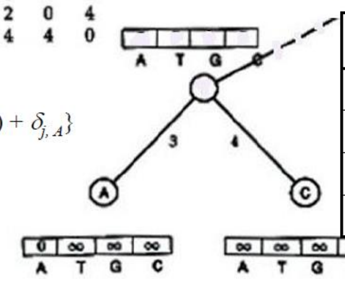
- If leaf has the character in question, score is 0
- Else, score is ∞



δ	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

$s_t(v) = \min_i \{s_i(u) + \delta_{i,t}\} + \min_j\{s_j(w) + \delta_{j,t}\}$

$s_A(v) = 0 + \min_j\{s_j(w) + \delta_{j,A}\}$

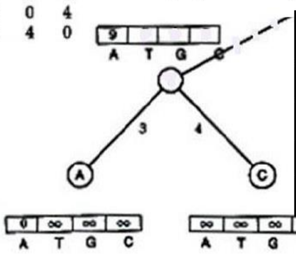


	$s_i(u)$	$\delta_{i,A}$	sum
A	0	0	0
T	∞	3	∞
G	∞	4	∞
C	∞	9	∞

δ	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

$s_t(v) = \min_i \{s_i(u) + \delta_{i,t}\} + \min_j\{s_j(w) + \delta_{j,t}\}$

$s_A(v) = 0 + 9 = \mathbf{9}$

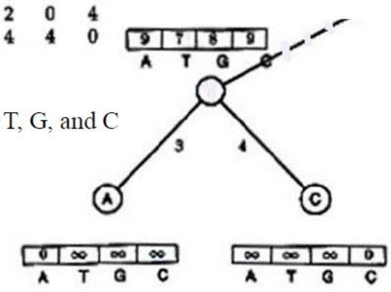


	$s_j(u)$	$\delta_{j,A}$	sum
A	∞	0	∞
T	∞	3	∞
G	∞	4	∞
C	0	9	**9**

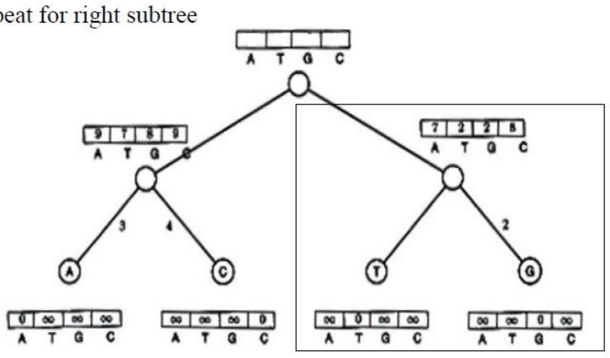
δ	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

$s_t(v) = \min_i \{s_i(u) + \delta_{i,t}\} + \min_j\{s_j(w) + \delta_{j,t}\}$

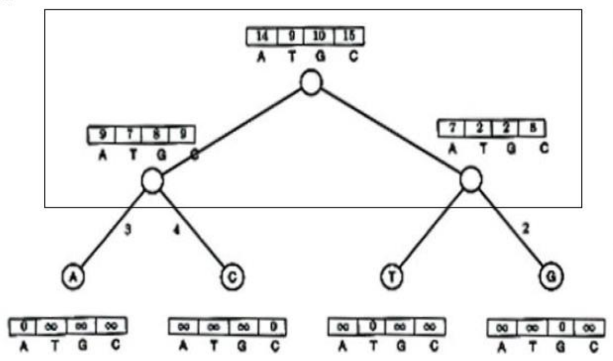
Repeat for T, G, and C



Repeat for right subtree

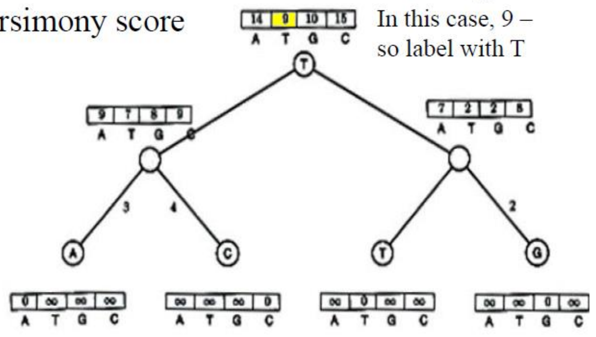


Repeat for root

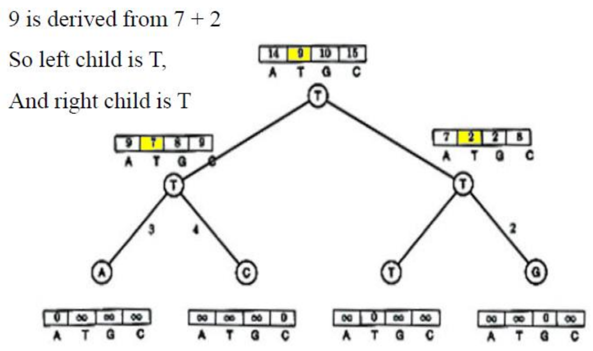


Smallest score at root is minimum weighted parsimony score

In this case, 9 – so label with T



After the scores at root vertex are computed the Sankoff algorithm moves down the tree and assign each vertex with optimal character.



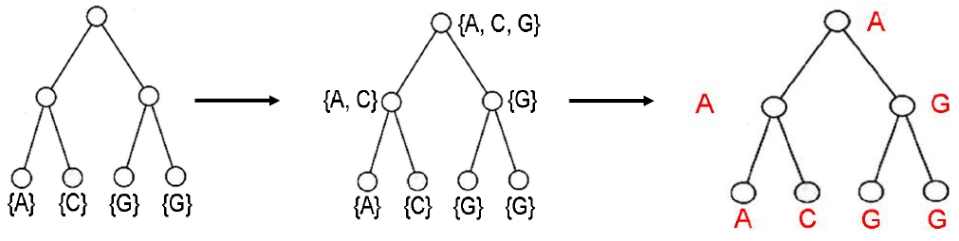
6.3.2. Fitch's Algorithm

Step 1) Assigns a set of letters to every vertex in the tree.
The father is:

- If the two children's sets of character overlap, it's the common set of them
- If not, it's the union set of them.

Step 2) Assign labels to each vertex, traversing the tree from root to leaves
Assign root arbitrarily from its set of letters
For all other vertices:

- If its parent's label is in its set of letters, assign it its parent's label
- Else, choose an arbitrary letter from its set as its label



$O(nk)$
The Sankoff algorithm gives the same set of optimal labels as the Fitch algorithm

6.3.3. Large Parsimony Problem

Input: An $n \times m$ matrix M describing n species, each represented by an m character string
Output: A tree T with n leaves labeled by the n rows of matrix M, and a labeling of the internal vertices such that the parsimony score is minimized over all possible trees and all possible labelings of internal vertices

Search space is huge, NP-complete, it's solvable for $n < 10$

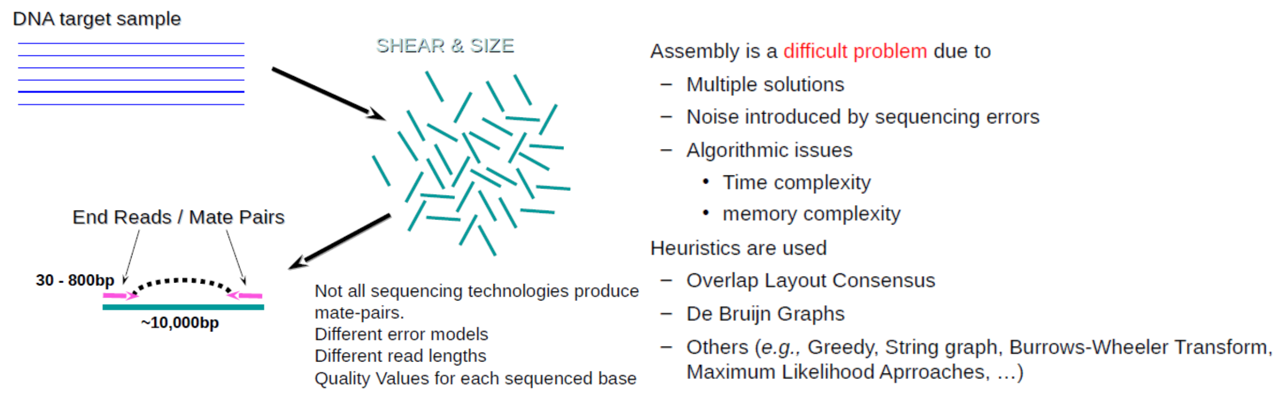
Some greedy techniques based on Tree:

- Nearest Neighbor Interchange
- Subtree Pruning and Recrafting
- Tree Bisection and Reconnection

Problems with Parsimony: reliance on one method provides incomplete picture
When different methods all give same result, more likely that the result is correct

7. DNA Assembly

It’s not possible to read the DNA in one-shot. But we read many fragments called **reads**.



Not all sequencing technologies produce mate-pairs.
Different error models
Different read lengths
Quality Values for each sequenced base

Assembly is a difficult problem due to
- Multiple solutions
- Noise introduced by sequencing errors
- Algorithmic issues
  - Time complexity
  - memory complexity

Heuristics are used
- Overlap Layout Consensus
- De Bruijn Graphs
- Others (*e.g.*, Greedy, String graph, Burrows-Wheeler Transform, Maximum Likelihood Aprroaches, …)

De-novo assembly: do everything from scratch
- Easier with long reads
- Needs good coverage (~10x min)
- Generally, produces fragmented assemblies (i.e., contigs)
- Only option when you don’t have a closely related (and correctly assembled) reference genome

Comparative assembly: we have a “reference” (related) genome.
- Easier
- Basic idea:
  - Take the reads and map them against the reference genome allowing for some small mismatches (Alignment)
  - Collect all overlapping reads, perform multiple sequence alignment, and produce consensus sequence
- Short reads can map to several places
- Needs close reference genome
- Repeats are problematic
- Can be highly accurate even when reads have errors

Simplest scenario:
- Reads have no errors
- Reads are long enough that each of them appears exactly once in the genome
- Each read has the same orientation
- CSP

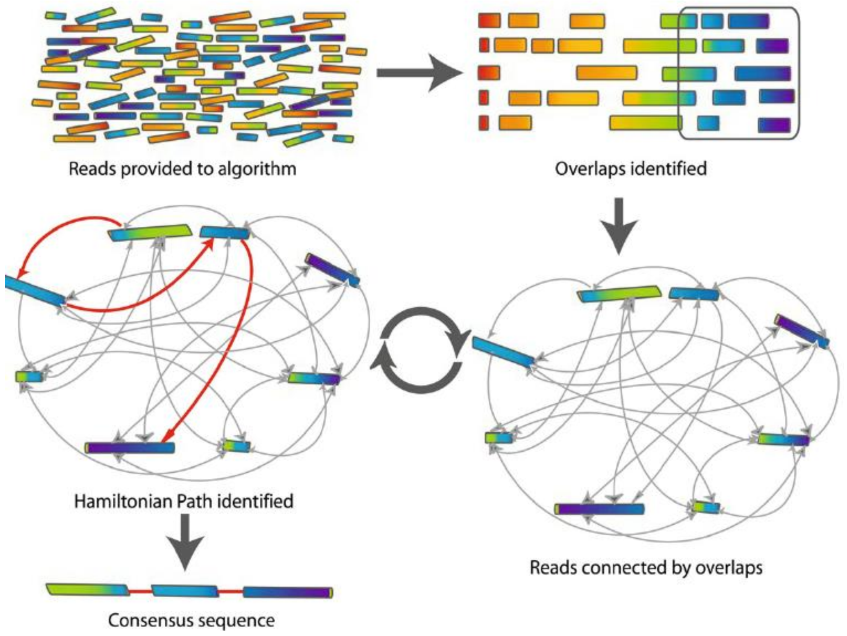
Common Superstring Problem (CSP): Given a set R of strings find a string S such that r is a substring of S for all r in R

Trivial solution: concatenate all the strings in R.

Overlap Graph
- Each read is a node
- directed edge from u to v if the two reads have sufficient overlap
- Objective: Find a Hamiltonian Path (for linear genomes) or a Hamiltonian Circuit (for circular genomes)

Hamiltonian Path Approach

- NP-hard
- produce multiple solutions
- Tends to produce fragmented assemblies


Reads provided to algorithm
Overlaps identified
Hamiltonian Path identified
Reads connected by overlaps
Consensus sequence

Identifying overlaps: Computing all-pairs overlap is computationally expensive, especially for NGS datasets, which can have millions of short reads.

***k*-mer Graph (de Bruijn graph):**

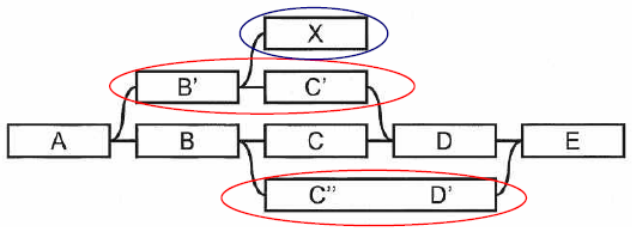
- Vertices are k-mers (substrings of length k) that appear in some read
- edges are defined by overlap of $k-1$ nucleotides
- number of nodes is $O(4^k)$
- Does not require all-pairs overlap calculation
- loss of information about reads can lead to incorrect assemblies
- produces fragmented assemblies
- assemblies are constructed by finding **Eulerian paths**

Eulerian path is a path going through each of the edge exactly once.
If the k-mer set comes from a sequence and each k-mer appears exactly once in the sequence, then the de Bruijn graph has a Eulerian path.
Sequencing errors and repeats may lead to non Eulerian graphs or to graphs with multiple Eulerian paths. To overcome this problem, we use **pruning**.

Errors on reads can cause "*irregularities*" to the graph, mainly in two forms

Bubbles: errors inside a read involving k-1 k-mers

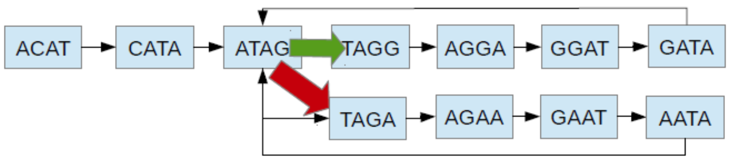
Dead ends: errors on the edge of the reads



Using de Bruijn Graphs:

- Given: set of k-mers from a DNA sequence
- Algorithm:
  - Construct the de Bruijn graph
  - Find an Eulerian path in the graph
  - The path defines a sequence with the same set of k-mers as the original, but this may not be the original DNA sequence

Seqeunce: **ACATAGGATAGAATAG**



Chimeric Sequence: ACATAGAATAGGATAG

Correct Sequence: ACATAGATAGAATAG

Velvet: tool to constrict the k-mer graph

- For each k-mer, create an hash value for it and its reverse complement
- In a Hash Table store the hash values and the list of all reads that contains either it or its reverse complement
- Construct the de Bruijn graph
- Collapse linear paths into a single node
- Remove tips chains of nodes disconnected to one end
- Remove bubbles which are two paths starting and ending on the same node (only one is hold)
- Finally find an Eulerian path

Main problem with Velvet is the huge amount of memory

Abyss distributed the hash tables to several computational nodes in order to "evenly" spread the memory requirements.

Handling repeats:

- **Repeat detection**
  - **pre-assembly:** find fragments that belong to repeats
    - statistically (most existing assemblers
    - repeat database (RepeatMasker)
  - **during assembly:** detect "tangles" indicative of repeats
  - **post-assembly:** find repetitive regions and potential misassemblies.
    - Reputer, RepeatMasker
    - "unhappy" mate-pairs (too close, too far, mis-oriented)
- **Repeat resolution**
  - find DNA fragments belonging to the repeat
  - determine correct tiling across the repeat

Evaluate assembly quality with Contig N50 is the length of the smallest contig in the set that contains the fewest (largest) contigs whose combined length represents at least 50% of the assembly

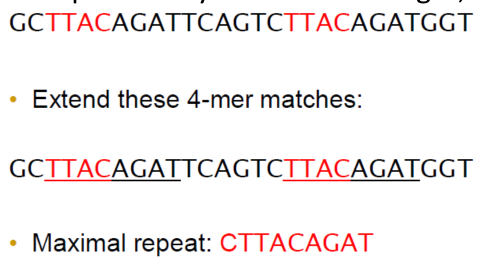
8. Pattern Matching

8.1. Genomic Repeats

Genomic Repeats: string that repeats with mutations ATGGTCTAGGACCTAGTGTTC

l −mer Repeats

- Short repeats are easy to find (e.g., hashing)
- Long repeats are difficult to find
  - Find exact repeats of short l-mers (l is usually 10 to 13)
  - Use l-mer repeats to potentially extend into longer, **maximal repeats**



GCTTACAGATTCAGTCTTACAGATGGT

- Extend these 4-mer matches:

GCTTACAGATTCAGTCTTACAGATGGT

- Maximal repeat: CTTACAGAT

Hashing DNA sequences:

- Each l −mer can be translated into a binary string (A, T, C, G can be represented as 00, 01, 10, 11)
- After assigning a unique integer per l-mer it is easy to get all start locations of each l-mer in a genome

To find repeats in a genome:

- For all l-mers in the genome, note the start position and the sequence
- Generate a hash table index for each unique l-mer sequence
- In each index of the hash table, store all genome start locations of the l-mer which generated that index
- Extend l-mer repeats to maximal repeats

8.2. Pattern Matching

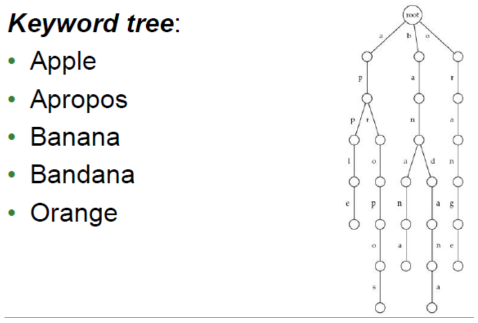
Pattern Matching Problem: Find all occurrences of a pattern in a text

- **Input**: Pattern $\boldsymbol{p} = p_1 \dots p_n$ and text $\boldsymbol{t} = t_1 \dots t_m$
- **Output**: All positions $1 \leq i \leq (m - n + 1)$ such that the n-letter substring of $\boldsymbol{t}$ starting at i matches $\boldsymbol{p}$

Brute force: $O(nm)$
Suffix trees: $O(m)$

Keyword Tree:

- Stores a set of keywords in a rooted labeled tree
- Each edge labeled with a letter from an alphabet
- Any two edges coming out of the same vertex have distinct labels
- Every keyword stored can be spelled on a path from root to some leaf



Keyword tree:
- Apple
- Apropos
- Banana
- Bandana
- Orange

Multiple Pattern Matching Problem: Given a set of patterns and a text, find all occurrences of any of patterns in text

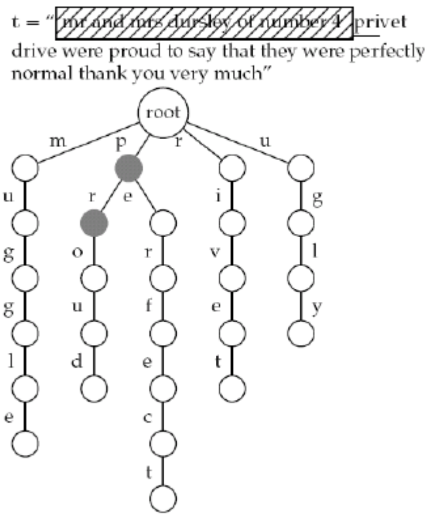
- **Input**: k patterns $\boldsymbol{p^1}, \dots, \boldsymbol{p^k}$ and text $\boldsymbol{t} = t_1 \dots t_m$
- **Output**: All positions $1 \leq i \leq m$ where substring of $\boldsymbol{t}$ starting at i matches $\boldsymbol{p_j}$ for $1 \leq j \leq k$

Keyword Tree Approach

- Build keyword tree in $O(N)$ time; N is total length of all patterns
- With naive threading: $O(N + nm)$
- Aho Corasick algorithm: $O(N + m)$

To match patterns in a text using a keyword tree:

- Build keyword tree of patterns
- Thread" the text through the keyword tree
- Threading is "complete" when we reach a leaf in the keyword tree
- When threading is "complete," we've found a pattern in the text


t = "mr and mrs dursley of number 4 privet drive were proud to say that they were perfectly normal thank you very much"
root
m
p
r
u

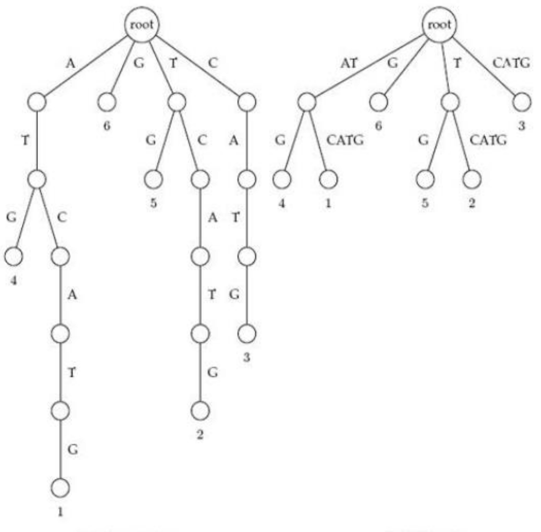
Suffix Trees=Collapsed Keyword Trees

- except edges that form paths are collapsed
- Each edge is labeled with a substring of a text
- All internal edges have at least two outgoing edges
- Leaves labeled by the index of the pattern.
- Suffix trees of a text is constructed for all its suffixes
- Builds in $O(m)$ time for text of length m

To find any pattern of length n in a text:

- Build suffix tree for text
- Thread the pattern through the suffix tree
- Can find pattern in text in $O(n)$

$O(n + m + n_{occ})$ time for "Pattern Matching Problem"

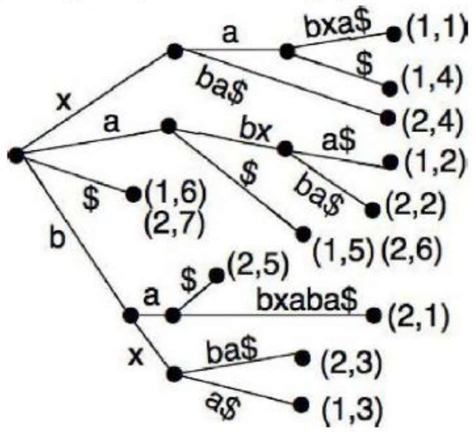

(a) Keyword tree (b) Suffix tree

Generalized Suffix Tree: be used to represent all suffixes of a set of strings

Construction: Apply ordinary suffix tree construction to a concatenation of the strings: $C = S_1\$_1 \dots S_k\$_k$

- $\$_i$ is a unique end-marker for each S_j => each leaf corresponds to a start position of C
- These can be converted to pairs (i, j), where i identifies the string, and j is the position in string S;
- Total time is $O(|C|)$
- Synthetic that span boundaries of original strings can be eliminated

Generalized suffix tree for $S_1 = xabxa$ and $S_2 = babxba$
(after completing all phases for $S_2\$$):



Longest common substrings

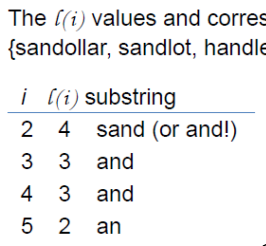
For each node v of a suffix tree, define the string depth of v to be the length of the path label $L(v)$ of v

Finding maximal substrings common to strings S1 and S2

- Build a generalized suffix tree for S1 and S2
- Mark each internal node v by 1 (resp. 2) if the subtree below v contains a leaf for a suffix of S1 (resp.S2)
- Traverse the tree to find nodes marked by both 1 and 2, and choose any u of them with a maximal string depth => $L(u)$ is a maximal common substring

Frequent Common Substring

- Let S be a set of strings $\{S_1, \dots, S_k\}$, of total length n. Define $I(i)$, for each $i = 2, \dots, k$, as the length of a maximal substring that is common to at least i strings of S
- We want to compute for each $i = 2, \dots, k$ the value $I(i)$

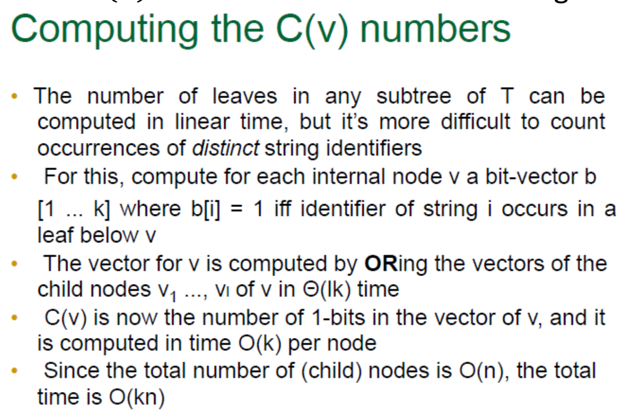


The $\ell(i)$ values and corresponding substrings for «S= {sandollar, sandlot, handler, grand, pantry}:

i	$\ell(i)$	substring
2	4	sand (or and!)
3	3	and
4	3	and
5	2	an

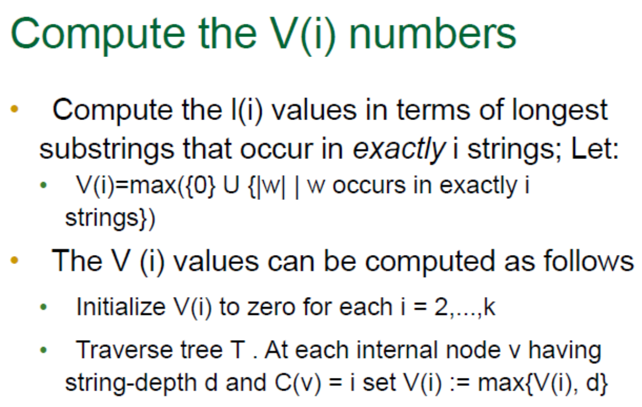
$$O(kn)$$

- Compute a generalized suffix tree T for strings $S_1\$_1 \dots S_k\$_k$
- Compute how many distinct string identifiers occur below any internal node of T
- $C(v)$ be the number of distinct string identifiers in the leaves of T below node v



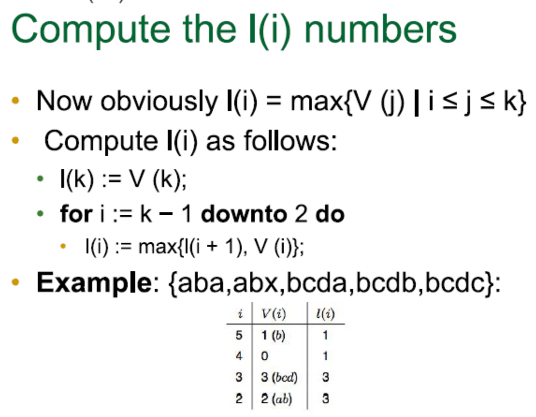
Computing the C(v) numbers

- The number of leaves in any subtree of T can be computed in linear time, but it's more difficult to count occurrences of *distinct* string identifiers
- For this, compute for each internal node v a bit-vector b [1 ... k] where b[i] = 1 iff identifier of string i occurs in a leaf below v
- The vector for v is computed by **OR**ing the vectors of the child nodes v_1 ..., v_l of v in Θ(lk) time
- C(v) is now the number of 1-bits in the vector of v, and it is computed in time O(k) per node
- Since the total number of (child) nodes is O(n), the total time is O(kn)



Compute the V(i) numbers

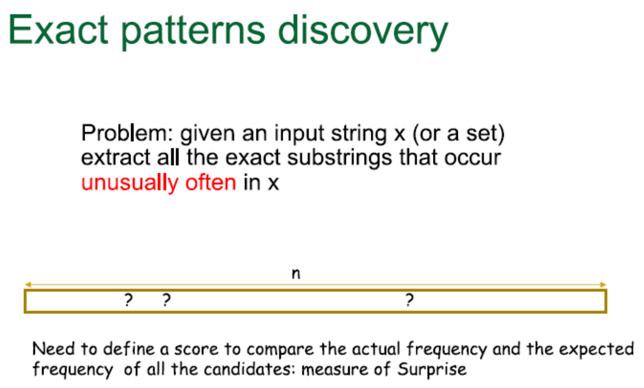
- Compute the I(i) values in terms of longest substrings that occur in *exactly* i strings; Let:
  - V(i)=max({0} U {|w| | w occurs in exactly i strings})
- The V (i) values can be computed as follows
  - Initialize V(i) to zero for each i = 2,...,k
  - Traverse tree T . At each internal node v having string-depth d and C(v) = i set V(i) := max{V(i), d}



Compute the I(i) numbers

- Now obviously **I**(i) = max{V (j) **|** i ≤ j ≤ k}
- Compute **I**(i) as follows:
  - I(k) := V (k);
  - **for** i := k − 1 **downto** 2 **do**
    - I(i) := max{I(i + 1), V (i)};
- **Example**: {aba,abx,bcda,bcdb,bcdc}:

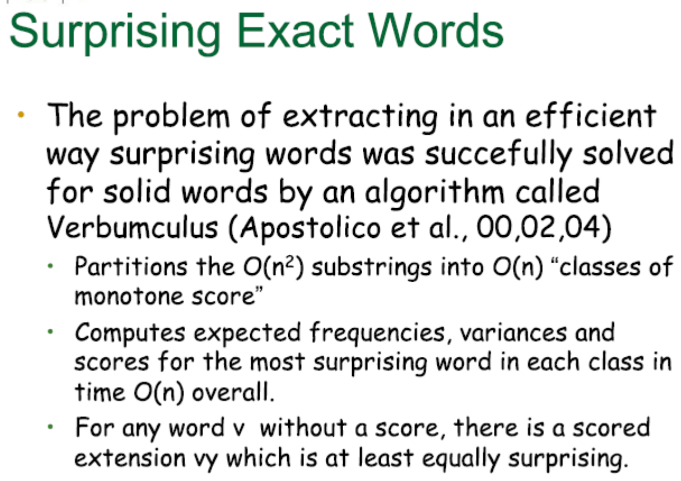
i	$V(i)$	$\ell(i)$
5	1 (b)	1
4	0	1
3	3 (bcd)	3
2	2 (ab)	3



Exact patterns discovery

Problem: given an input string x (or a set) extract all the exact substrings that occur unusually often in x

Need to define a score to compare the actual frequency and the expected frequency of all the candidates: measure of Surprise



Surprising Exact Words

- The problem of extracting in an efficient way surprising words was succefully solved for solid words by an algorithm called Verbumculus (Apostolico et al., 00,02,04)
  - Partitions the O(n^2) substrings into O(n) "classes of monotone score"
  - Computes expected frequencies, variances and scores for the most surprising word in each class in time O(n) overall.
  - For any word v without a score, there is a scored extension vy which is at least equally surprising.

8.3. Approximate Pattern Matching

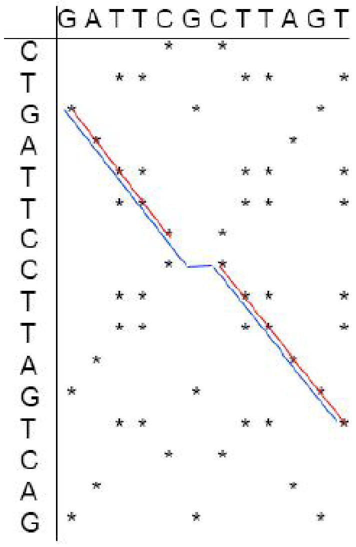
Usually, because of mutations, it makes much more biological sense to find approximate pattern matches.

Heuristic Similarity Searches

- Find short exact matches, and use them as seeds for potential match extension
- "Filter" out positions with no extendable matches

FASTA:

- Identify diagonals above a threshold length
- Diagonals in the dot matrix indicate exact substring matching.
- Extend diagonals and try to link them together, allowing for minimal mismatches/indels
- Linking diagonals reveals approximate matches over longer substrings



Approximate Pattern Matching Problem: Find all approximate occurrences of a pattern in a text.

Input: A pattern **p** = $p_1 \ldots p_n$, text **t** = $t_1 \ldots t_m$, and k, the maximum number of mismatches

Output: All positions $1 \leq i \leq (m - n + 1)$ such that $t_i \ldots t_{i+n-1}$ and $p_1 \ldots p_n$ have at most k mismatches (i.e., Hamming distance between $t_i \ldots t_{i+n-1}$ and **p** $\leq k$)

Query Matching Problem: Find all substrings of the query that approximately match the text.

Input: Query **q** = $q_1 \ldots q_w$,
text **t** = $t_1 \ldots t_m$,
n (length of matching substrings),
k (maximum number of mismatches)

Output: All pairs of positions (i, j) such that the
n-letter substring of **q** starting at i
approximately matches the
n-letter substring of **t** starting at j,
with at most k mismatches

Query Matching: Approximately matching strings share some perfectly matching substrings.

Search for perfectly matching substrings (easy).
Filtration in query matching:

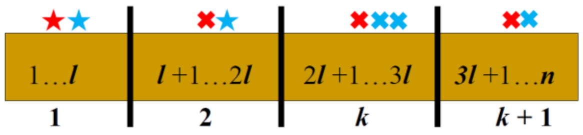
If $x_1 \ldots x_n$ and $y_1 \ldots y_n$ match with at most k mismatches, they must share an ℓ-tuple that is perfectly matched, with $\ell = \lfloor n/(k+1) \rfloor$

Break string of length n into k+1 parts, each each of length $\lfloor n/(k+1) \rfloor$

- k mismatches can affect at most k of these k+1 parts
- At least one of these k+1 parts is perfectly matched

- Suppose k = 3. We would then have $l=n/(k+1)=n/4$:

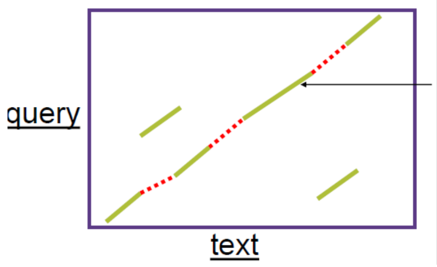
Gli errori posso o meno apparire uniformemente in queste suddivisioni. Per esempio:



- There are at most k mismatches in n, so at the very least there must be one out of the k+1 ℓ–tuples without a mismatch

Cerco gli l-mer che appaiono in maniera esatta nel testo, filtro tutto il resto perché questi saranno i punti di partenza di un possibile match che deve contenere un l mer esatto e poi cercherò di estendere l'l-mer per trovare la soluzione (allineamento locale tra la query e il testo).

For each ℓ-match we find, try to extend the match further to see if it is substantial

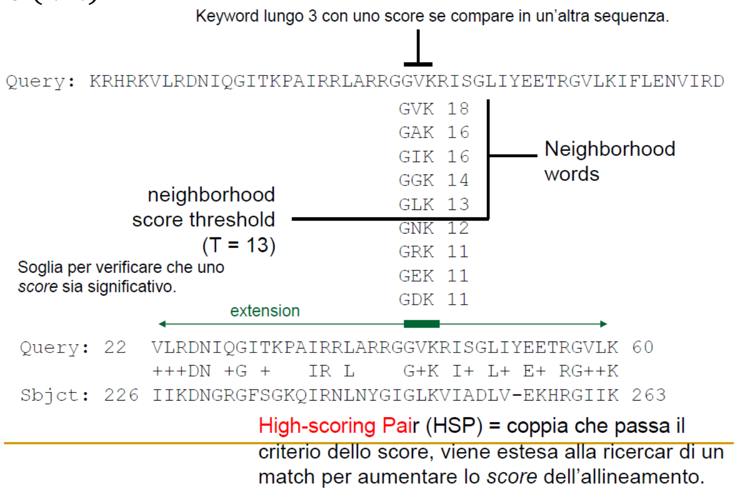


Extend perfect match of length ℓ until we find an approximate match of length n with k mismatches

Local alignment is to slow: Quadratic local alignment is too slow while looking for similarities between long strings.

BLAST

- Great improvement in speed, with a modest decrease in sensitivity
- Minimizes search space instead of exploring entire search space between two sequences
- Finds short exact matches ("seeds"), only explores locally around these "hits"
- Phase 1: Keyword search of all words of length w from the query of length n in database of length m with score above threshold.
- Phase 2: Local alignment extension for each found keyword
- Extend result until longest match above threshold is achieved
- Running time $O(nm)$



Keyword lungo 3 con uno score se compare in un'altra sequenza.
Neighborhood words
neighborhood score threshold (T = 13)
Soglia per verificare che uno score sia significativo.
extension
High-scoring Pair (HSP) = coppia che passa il criterio dello score, viene estesa alla ricercar di un match per aumentare lo score dell'allineamento.

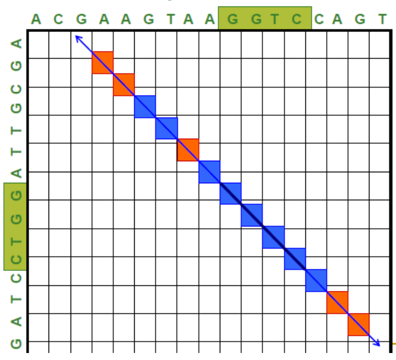
```
Query: KRHRKVLRDNIQGITKPAIRRLARRGGVKRISGLIYEETRGVLKIFLENVIRD
                                          GVK 18
                                          GAK 16
                                          GIK 16
                                          GGK 14
                                          GLK 13
                                          GNK 12
                                          GRK 11
                                          GEK 11
                                          GDK 11

Query: 22  VLRDNIQGITKPAIRRLARRGGVKRISGLIYEETRGVLK 60
           +++DN +G +    IR L     G+K I+ L+ E+ RG++K
Sbjct: 226 IIKDNGRGFSGKQIRNLNYGIGLKVIADLV-EKHRGIIK 263
```

Original BLAST

- Dictionary: All words of length *w*
- Alignment: Ungapped extensions until score falls below some statistical threshold
- Output: All local alignments with score > threshold

Original BLAST: Example

- *w* = 4
- Exact keyword match of GGTC
- Extend diagonals with mismatches until score is under 50%
- Output result
  GTAAGGTCC
  GTTAGGTCC

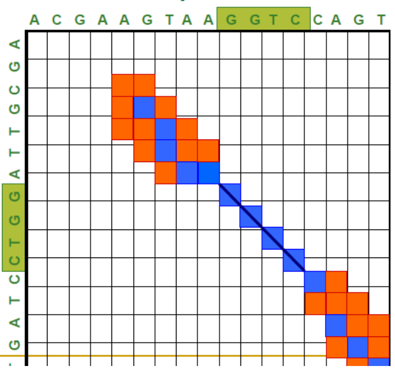


Gapped BLAST : Example

Original BLAST exact keyword search, THEN:

Extend with gaps around ends of exact match until score < *threshold*

Output result
GTAAGGTCCAGT
GTTAGGTC-AGT



BLAST: Segment Score

- BLAST uses scoring matrices (δ) to improve on efficiency of match detection
  - Some proteins may have very different amino acid sequences, but are still similar
- For any two ℓ-mers $x_1 \ldots x_l$ and $y_1 \ldots y_l$:
  - Segment pair: pair of ℓ-mers, one from each sequence
  - Segment score: $\Sigma^{\ell}_{i=1}\ \delta(x_i, y_i)$

BLAST: Locally Maximal Segment Pairs

- A segment pair is maximal if it has the best score over all segment pairs
- A segment pair is locally maximal if its score can’t be improved by extending or shortening
- Statistically significant locally maximal segment pairs are of biological interest.
- BLAST finds all locally maximal segment pairs with scores above some threshold. A significantly high threshold will filter out some statistically insignificant matches

BLAST: matches short consecutive sequences (consecutive seed)
PatternHunter: matches short non consecutive sequences (spaced seed)
Is PH better: Higher hit probability. Lower expected number of random hits.

BLAT (BLAST Like Alignment Tool): Same idea as BLAST locate short sequence hits and extend.
Builds an index of the database and scans linearly through the query sequence, whereas BLAST builds an index of the query sequence and then scans linearly through the database
Index is stored in RAM which is memory intensive, but results in faster searches

9.Alignment Free

Basic Axiom of Computational Biology: A high similarity among objects, measured by mathematical functions, is strong indication of functional relatedness and/or common ancestry.

Basic Problems:

- Definition of good similarity/distance functions
- Development of efficient algorithms for their computation

Alignment-Free Methods: Similarity of two strings is assessed based only on the DICTIONARY of substrings that appear in the strings, irrespective of their relative position.

Computational Approaches:

- **Explicit Collection** and Use of Word Statistics, either exact or approximate
- **Implicit Collection** and Use of Word Statistics

Explicit Collection of Word Statistics: Intuition: two strings are similar if they are composed of the same basic building blocks.

Example: abracadabra , abracaabrad,

- **Exact L tuple statistics:**

  Example for L=2 and Binary Alphabet

  - ababaab
  - Dictionary D =(aa, ab, ba,bb)
  - Char Vector =(1 , 3 , 2 , 0)

  Once Characteristic vector is known, similarity and distance can be computed with dozens of formulae.

- **Approximate L tuple statistics:**

  Example for L=2, K=1 and Binary alphabet

  - ababaab
  - Dictionary D=(aa, ab,ba,bb)
  - Char Vector =(6 , 4 , 3 , 5)

Implicit Collection of Word Statistics: Similarity is captured by quantifying "how easy" it is to describe x, given y. **Kolmogorov Complexity**

Example: abraabraabra | abra

- **Kolmogorov Complexity** $K(x)$ of a string x is defined as the length of the shortest binary program that produces x
- **Conditional Kolmogorov Complexity** $K(x|y)$ represents the minimum amount of information required to generate x by an effective computation when y is given as an input to the computation
- **Kolmogorov Complexity** $K(x,y)$ of a pair objects x and y is the length of the shortest binary program that produces x and y and a way to tell them apart.

- **Universal Similarity metric (USM):** is a lower bound, of any computable distance/similarity function. Based on Kolmogorov Complexity.

$$USM(x,y)=\frac{\max\{K(x\mid y^*),K(y\mid x^*)\}}{\max\{K(x),K(y)\}}$$

  K(x) can be approximated via data compression by using its relationship with Shannon Information Theory.
  Given compression algorithm C, K(x) can be approximated by $|C(x)|$, $K(x,y)$ by $|C(xy)|$ and $K(x|y^*)$ by $|C(xy)-C(x)|$.
  In practice, USM become a methodology that depends critically on the choice of compression algorithm.
  Given compression algorithm, three general formulas to approximate USM:

$$UCD(x,y)=\frac{\max\{C(xy)-C(x),C(yx)-C(y)\}}{\max\{C(x),C(y)\}}$$

where

$$NCD(x,y)=\min\{NCD_1(x,y),NCD_1(y,x)\}\qquad NCD_1(z,w)=\frac{C(zw)-\min\{C(z),C(w)\}}{\max\{C(z),C(w)\}}$$

$$CD(x,y)=\frac{\min\{C(xy),C(yx),C(x)+C(y)\}}{C(x)+C(y)}$$

Evaluation Methodology: How good is an alignment free similarity/distance functions:
- Data Sets with a trustworthy classification
- Statistical Tools to establish the intrinsic ability of the similarity/distance to discriminate relatedness of strings ROC Analysis
- Statistical Tools to establish how well standard classification algorithms

Application 1: Comparison of Regulatory Sequences
Similar binding site contents are expected to drive similar expression patterns.
Identification of enhancing sequences that regulate the same cell type.
- Transcription factors binding sites often occur in clusters, also called cis regulatory modules (CRMs)
- The position and orientation of binding sites in CRMs may vary, making an alignment often impossible.

Alignment Free Statistics:
- **D_2 statistic:** is the correlation between the number of occurrences of k-mers appearing in two sequences A and B. Can be biased by the stochastic noise in each sequence

$$D_2=\sum_{w\in\Sigma^k}A_wB_w$$

A_w is the number of times w appears in A

- To standardize D_2 and to account for different k-mers distributions several statistics have been introduced, e.g. D_2^S and D_2^*

$$\tilde{A}_w=A_w-(n-k+1)*p_w \qquad D_2^*=\sum_{w\in\Sigma^k}\frac{\tilde{A}_w\tilde{B}_w}{(n-k+1)p_w} \qquad D_2^s=\sum_{w\in\Sigma^k}\frac{\tilde{A}_w\tilde{B}_w}{\sqrt{\tilde{A}_w^2+\tilde{B}_w^2}}$$

- $\boldsymbol{N_2}$ **statistic:** very similar to D_2^* except it counts k-mers with at most one mismatch and considers also the reverse complement.

Major Problems with Alignment Free Statistics:

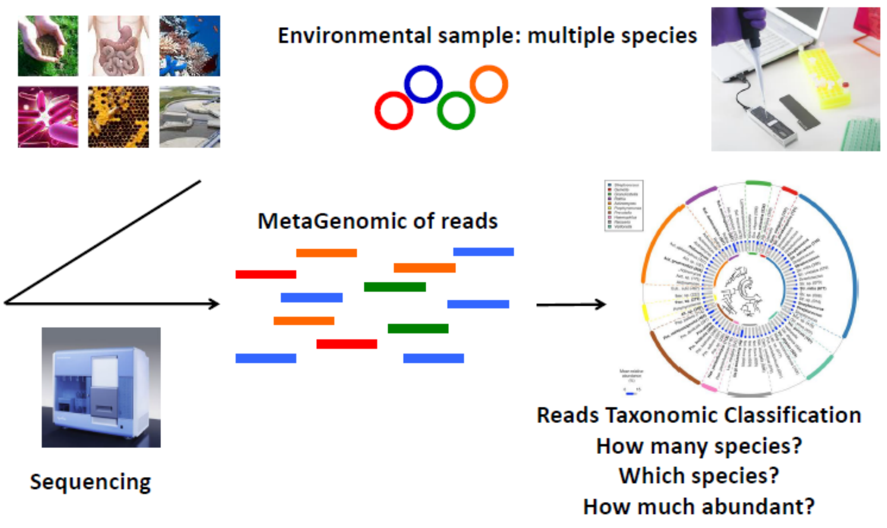
- are influenced by the length / resolution k of k-mers.
- For CRMs, where multiple binding sites with different lengths are present, a fixed resolution k will never capture the statistics of all binding sites
- The presence of repeats can alter the occurrence profile of some k mers.

Reads quality: generate tons of reads, filter out low quality reads, and work with the rest, assuming it is by and large error free.

10. Metagenomics

Genetics: the study of individual genes and their roles in inheritance.
Genomics: it is an interdisciplinary field of science focusing on the structure, function, evolution, mapping, and editing of genomes.

Metagenomics: is the study of genomic sequences obtained directly from an environment where multiple microorganisms coexists.
Permette di sequenziare tutto il materiale genetico presente in un campione.



MetaGenomic Reads Analysis:

- **Reference Based** (Supervised): Use reference database to assign reads to a given species
- **Reference Free** (Binning)

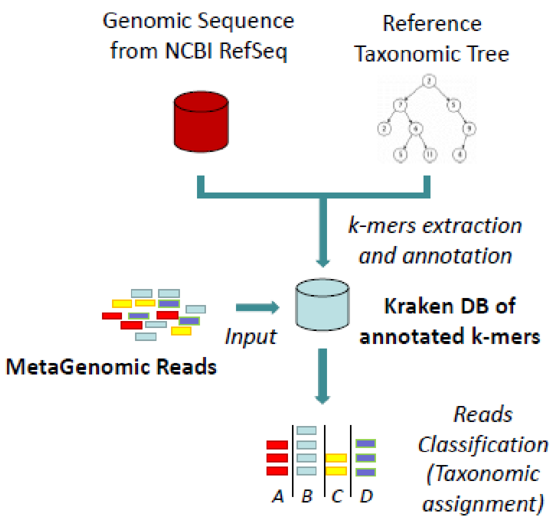
Kraken (Reference based)

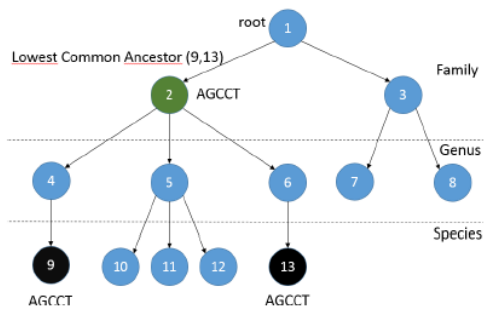
K-mer DB Creation
Input:
- ALL genome sequences (NCBI RefSeq)
- Taxonomic Tree

Output:
- Kraken DataBase of annotated k-mers

Classification
Input:
- Metagenomic reads
- Kraken DB of annotated k-mers

Output:
- Taxonomic classification of each reads


Genomic Sequence from NCBI RefSeq
Reference Taxonomic Tree
k-mers extraction and annotation
Input
Kraken DB of annotated k-mers
MetaGenomic Reads
A
B
C
D
Reads Classification (Taxonomic assignment)

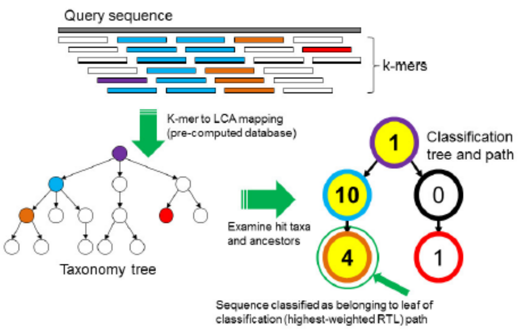


For all Bacterial and Archaeal complete genomes in NCBI RefSeq, Kraken DB contains 5.8 billion k-mers, about 65GB.

Augmented Taxonomic Tree:

- At every node in taxonomic tree is associated a list of k-mers
- Scan all k-mers of each genomic sequence in the dataset.
- If a k-mer appears only in a given genome, than it is associated to the leaf node representing the species of that genome
- If a k-mer appears in more than one species then its moved to the Lowest Common Ancestor (LCA) of these nodes.
- Every k-mers is associated only to one node in the taxonomic tree.

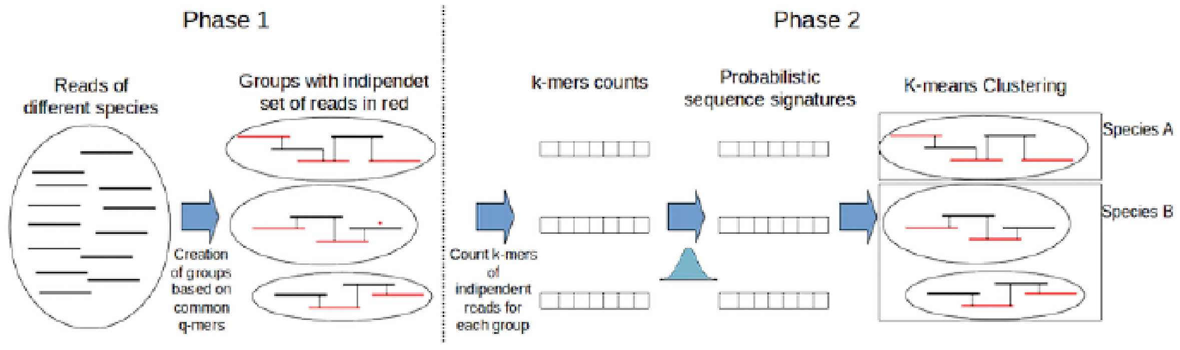
- A read is decomposed into the list of its k-mers
- Each k-mer is searched in the augmented taxonomic tree, and in the corresponding node a counter is incremented for every hit.
- The node's counters are used to classify the read by searching the highest weighted path, from the root to a leaf (RTL), in the tree


Query sequence
k-mers
K-mer to LCA mapping
(pre-computed database)
Taxonomy tree
Examine hit taxa and ancestors
Classification tree and path
1
10
0
4
1
Sequence classified as belonging to leaf of classification (highest-weighted RTL) path

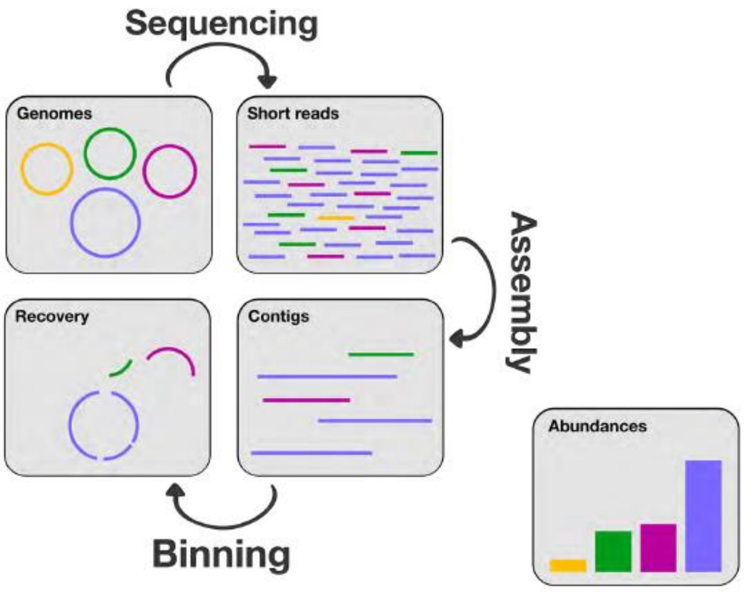
MetaProb (Reference based): is a novel reference free assembly assisted tool for metagenomic reads binning.

- Phase 1: Reads are grouped based on overlap information between reads , similarly as in de novo assembly. In each group we identify a set of independent reads
    - The construction of the reads overlap graph can be prohibitive
    - We assume that two reads overlaps if they share at least m q mers q =30
    - Reads are progressively merged into groups
    - The k-mer counts of a group can be artificially inflated by overlaps
    - To avoid overcounting overlaps, for each group, we define a subset of independent reads (in red) that do not overlap with each other.
- Phase 2: Extract the probabilistic sequence signature from each group based on k mers counts. Groups of reads are clustered based on their sequence signatures

MetaProb can also estimate of the number of species.
MetaProb can deal with short and long reads in a novel probabilistic framework, by using probabilistic sequence signatures.


Phase 1
Phase 2
Reads of different species
Groups with indipendet set of reads in red
k-mers counts
Probabilistic sequence signatures
K-means Clustering
Species A
Species B
Creation of groups based on common q-mers
Count k-mers of indipendent reads for each group

Binning (reference free): is the process of sorting DNA sequences into group that might represent an individual genome or genomes from closely related organisms.
Si analizzano i contig anzi che le read.



Domande di ripasso

Motif Finding

- Cos'è un Alignment Matrix, Profile Matrix, consensous e score?
- Definizione di Motif Finding Problem e soluzione.
  Definizione di Hamming distance tra stringhe.
  Definizione di Total Distance tra stringa e DNA.
  Definizione di Median String Problem e soluzione.
- Che relazione c'è tra Motif Finding Problem e Median String Problem?
- Che miglioramenti si possono appurare per ottenere una migliore complessità temporale?

Randomized Algorithm for Motif Finding

- Cos’è un algoritmo randomizzato?
- Cosa sono gli algoritmi Las Vegas e Monte Carlo?
- Come si trovano i P-Most Probable l-mer in a single sequence?
  Come si trovano i P-Most Probable l-mer in many sequences?
  Come funziona il Greedy Profile Motif Search?
- Come funziona il Gibbs Sampling?
- Come funziona il Random Projection?

Dynamic programming

- A cosa serve il similarity score? Cosa ha a che fare coN IL dp?
- Cos'è l'allinemento?
- Definizione del Longest Common Subsequence (LCS) e soluzione.
- Definizione di edit distance e sua ricorrenza
- Definizione di global Alignment e sua ricorrenza
- Che relazione c'è tra LCS e Global Aligment?
- Definizione di Scoring Matrix, a cosa serve, quali sono le più comuni.
- Che differenza c'è tra Global e Local Alignment?
- Definizione di Local Alignment e sua ricorrenza
- K-best local alignments
- Definizione di Alignment with Affine Gap Penalties
- Definizione di Multiple Alignment e sue possibili soluzioni.

D&C Algorithms

- Come funziona un algoritmo D&C?
- Cos'è il block alignement?
- Definizione del Block Alignment Problem e sua soluzione
- In che modo si può velocizzare il block alignment?
- Qual è il problema principale del block alignment? E come si può risolvere?

Genomic Rearrangements

- Cosa fa un’operazione di reversal?
- Definizione di Reversal Distance Problem.
- Definizione di Sorting by Reversals Problem.
- Definizione di Pancake Flipping Problem.

- Cos'è un Adjacency e un Breakpoint?
- Come funziona l'algoritmo Sorting by Reversals?
- Cos'è uno strip?
- Come funziona l'algoritmo ImprovedBreakpointReversalSort?

Molecolar Evolution
- Come è fatto un Evolutionary Tree?
- Definizione di Tree Distance e Distance Matrix
- Di cosa si occupa la Molecolar Evolution?
- Come si ricostruisce un albero? Qual'è il maggior problema?
- Definizione di Distance Based Phylogeny Problem.
- Come funziona il Neighbor Joining Algorithm?
- Additive Phylogeny Algorithm
    - Cosa sono le Degenerate Triples? Come possono essere usate?
    - Come funziona l'Additive Phylogeny Algorithm?
    - Cosa dice e a cosa serve la The Four Point Condition?
- Che algoritmi possiamo usare in caso di NOT Additive distance matrix?
- Parsimony problems
    - Cosa fa il Character Based Tree Reconstruction?
    - Come is calvola il Parsimony score of a tree e a cosa serve?
    - Definizione di Small Parsimony Problem
    - Definizione di Weighted Small Parsimony Problem
    - Come funziona il Sankoff’s Algorithm?
    - Come funziona il Fitch’s Algorithm?
    - Definizione di Large Parsimony Problem e suoi algoritmi risolutivi.
- Qual'è il problema del Parsimony? E come può essere risolto?

DNA Assembly
- What is the main problem when reading DNA?
- What are the 2 ways to read DNA?
- Definizione di Common Superstring Problem (CSP) e sua soluzione
- Com'è fatto un overlap graph, a cosa serve? (Hamiltonian Path Approach)
- Com'è fatto un k-mer Graph (de Bruijn graph)? Come si usa?
- Cos'è velvet? A cosa serve?
- Come si gestiscono i repeats?
- Come si valuta la qualità degli assebly?

Pattern Matching:
- Come si usano i l-mer per trovare i repeats?
- Come funziona l'hashing delle sequenze di DNA? E come viene usato per trovare repeats in un genoma?
- Pattern Matching
    - Definizione di Pattern Matching Problem, e come si risolve?
    - Cos'è un Keyword tree?
    - Definizione di Multiple Pattern Matching Problem,e come si risolve?
    - Cos'è un suffix tree? Come si usa per il Pattern Matching Problem?
    - Cos'è un Generalized suffix tree? Come si costruisce?

    - Come si risolve il Longest common substring problem tra 2 stringhe?
    - Definizione di Frequent Common Substring e come si risolve.
- Approximate Pattern Matching
    - Approximate Pattern Matching: perché si usa?
    - Definizione di Approximate Pattern Matching Problem
    - Definizione di Query Matching Problem.
    - Come funziona la Filtration in query matching?
    - Come funziona BLAST? E le sue modifiche?
    - PatternHunter, BLAT

Alignment Free:
- Qual'è il Basic Axiom of Computational Biology?
- Quali sono i problemi principali?
- Cosa fanno gli Alignment-Free Methods e quali sono Computational Approaches?
- Come si valutano le metodologie?
- Qual'è una applicazzione dell'alignment free?
- Quali sono le Alignment Free Statistics? E quali problemi hanno?

Metagenomics:
- Cosa sono la genetica, la genomica e la metogenomica?
- Quali metodi esistono per analizzare le Reads?