Bioinformatics

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Computer Science Tripos, Part II

Michaelmas Term 2012/13

Dr Pietro Liò

Copies of slides

William Gates Building
15 JJ Thomson Avenue
Cambridge
CB3 0FD

http://www.cl.cam.ac.uk/

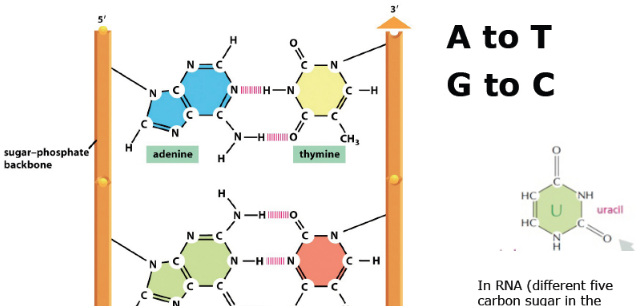
Overview of the course

The course focuses on algorithms for DNA sequence and gene expression data analysis. First we learn how to analyse one, two or more sequences. Searching a database for nearly exact matches (using Blast algorithm) is the most important routine work in a Bioinformatics labs. We learn how to build trees to study sequences relationship. We use hidden Markov models to infer properties such as the exon/intron arrangements in a gene or the structure of a protein. The second part of the course is about clustering gene expression data using K-means or the Markov clustering algorithm; then we can reconstruct the genetic networks (Wagner algorithm). Finally, a network of biochemical reactions could be simulated using the Gillespie algorithm. Key examples at the end of each lecture (see links at the end of this course material); figures sources acknowledged during the lectures.

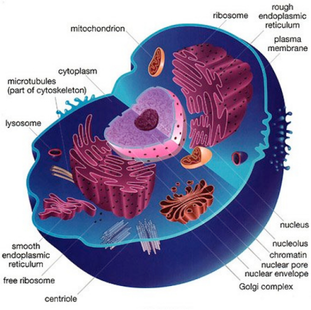
Topics and List of algorithms

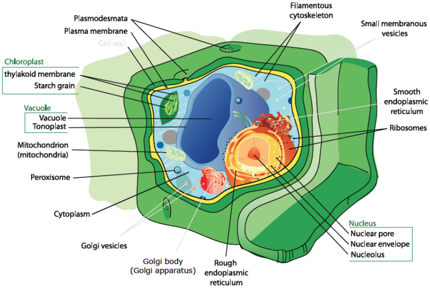
- Basic concepts in genetics and genomics.
- Dynamic programming (Longest Common Subsequence, DNA, RNA alignment, linear space alignment).
- Progressive alignment (Clustal).
- Alignment of Short reads to a reference genome: the Burrows-Wheeler transform
- Homology database search (Blast, Patternhunter).
- Phylogeny - parsimony-based - (Fitch, Wagner, Sankoff).
- Phylogeny - distance based - (UPGMA,Neighbour Joining).
- Phylogeny (consensus tree, tree rearrangements).
- Clustering (K-means, Markov Clustering)
- Hidden Markov Models applications in Bioinformatics (Genscan, TMHMM).
- Pattern search in sequences (Gibbs sampling).
- Biological Networks reconstruction (Wagner) and simulation (Gillespie).

1. DNA could be thought as a string of symbols from a 4-letter (bases) alphabet, A (adenine), T (thymine), C (cytosine) and G (guanine). In the double helix A pairs with T, C with G. A gene is a string of DNA that contains information for a cell function. The Genome is the entire DNA in a cell.
2. RNA is same as DNA but T → U (uracil); proteins are strings from an alphabet of 20 amino acids. The proteins have also a 3D shape which could be described as a 3 D graph. The genetic code is a map between DNA and proteins (3 DNA bases, i.e. 1 triplet, correspond to one amino acid).

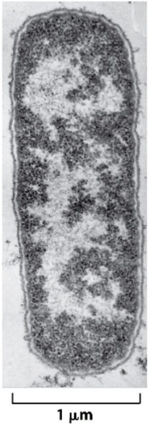


Top: a human cell (it measures 10μm across); bottom: a plant cell

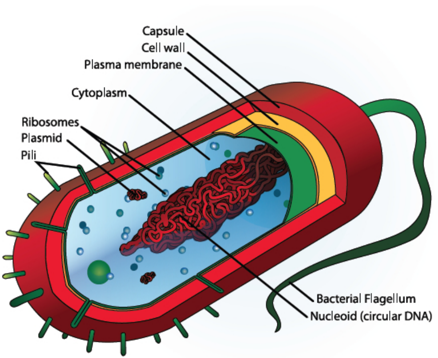




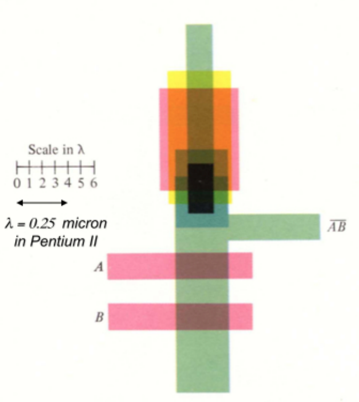
A bacterial cell (for example E. coli) measures about 2μm in length, yet it contains about $1,600\mu$m (1.6 mm) of circular double strands DNA (5 x10^6 DNA bases in E. coli).



Electron micrograph of E.coli (DNA in light stained region)



In eucaryotes the genetic information is distributed over different DNA molecules. A human cell contains 24 different such chromosomes. If all DNA of a human cell would be laid out end-to-end it would reach approximately 2 meters. The nucleus however measures only 6μm. Equivalent of packing 40 km of fine thread into a tennis ball with a compression ratio of 10000.

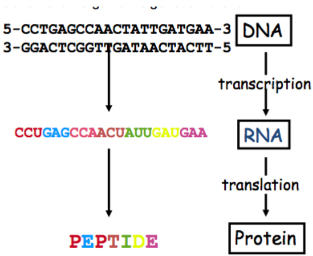


(a) NAND gate layout geometry.

Bacterium

Human chromosome.

DNA makes RNA (also called mRNA) makes proteins (the 3D graph below); given the pairing rule in a DNA double strands molecule, all the information is in each single strand. The RNA is termed mRNA and is translated by triplet of bases into a chain of amino acids (the protein).



5-CCTGAGCCAACTATTGATGAA-3

3-GGACTCGGTTGATAACTACTT-5

DNA

transcription

CCUGAGCCAACUAUUGAUGAA

RNA

translation

PEPTIDE

Protein

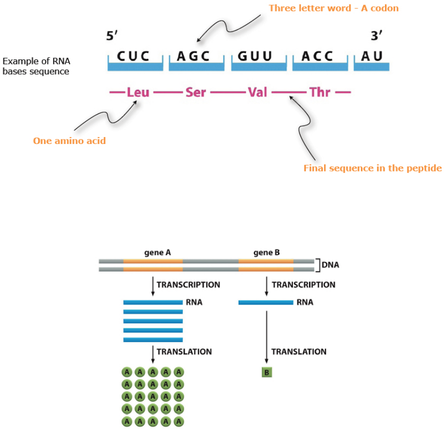
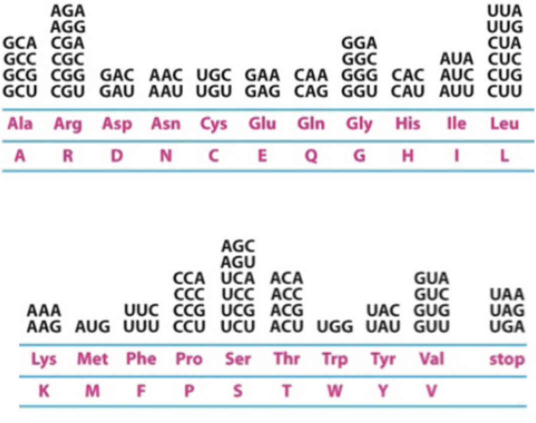


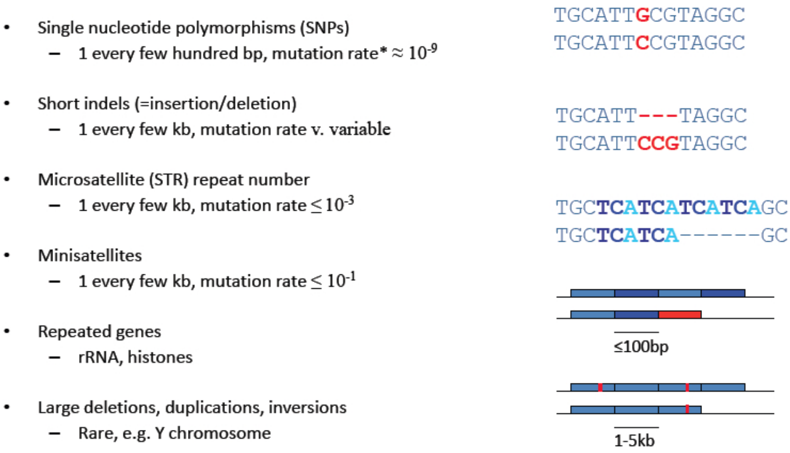
Figure: The central dogma of molecular biology is that DNA is transcribed to RNA which is translated to protein. The amount of RNA depends on gene activity which is influenced by other proteins binding before the start of the gene; different tissues contain cells with different amount of RNA for each gene



GCA GCC GCG GCU	AGA AGG CGA CGC CGG CGU	GAC GAU	AAC AAU	UGC UGU	GAA GAG	CAA CAG	GGA GGC GGG GGU	CAC CAU	AUA AUC AUU	UUA UUG CUA CUC CUG CUU
Ala	Arg	Asp	Asn	Cys	Glu	Gln	Gly	His	Ile	Leu
A	R	D	N	C	E	Q	G	H	I	L

AAA AAG	AUG	UUC UUU	CCA CCC CCG CCU	AGC AGU UCA UCC UCG UCU	ACA ACC ACG ACU	UGG	UAC UAU	GUA GUC GUG GUU	UAA UAG UGA
Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val	stop
K	M	F	P	S	T	W	Y	V	

Figure: The genetic code provides the information for the translation of codons (triplets of bases, in black) into amino acids (single and triple letter code in red) that are chained together to form a protein; 61 codons make 20 amino acids; 3 special codons say "stop message"

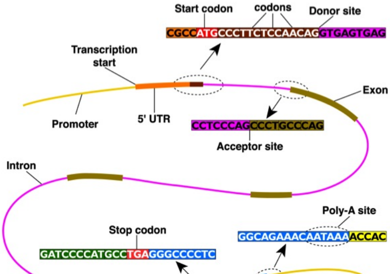


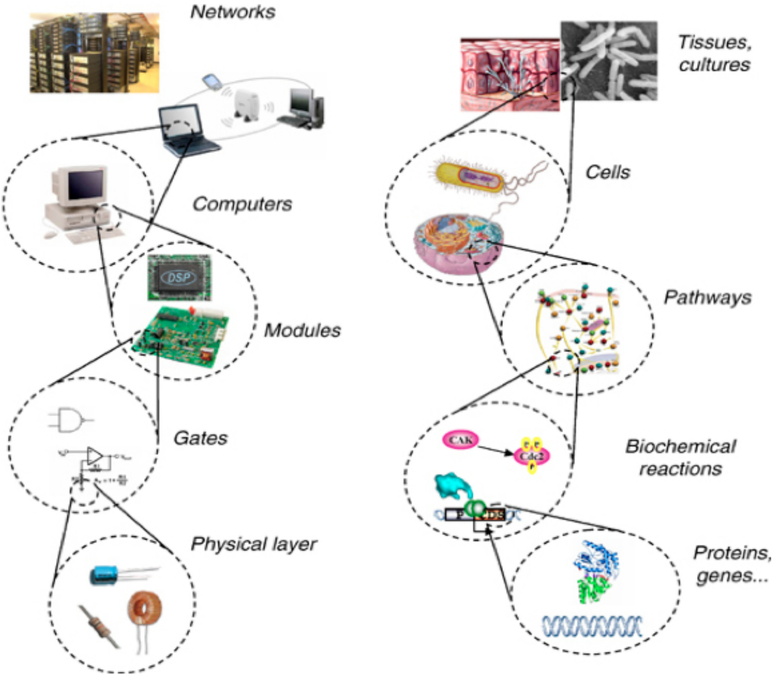
- Single nucleotide polymorphisms (SNPs)
  - 1 every few hundred bp, mutation rate* $\approx 10^{-9}$
- Short indels (=insertion/deletion)
  - 1 every few kb, mutation rate v. variable
- Microsatellite (STR) repeat number
  - 1 every few kb, mutation rate $\leq 10^{-3}$
- Minisatellites
  - 1 every few kb, mutation rate $\leq 10^{-1}$
- Repeated genes
  - rRNA, histones
- Large deletions, duplications, inversions
  - Rare, e.g. Y chromosome

Figure: Type and frequency of mutations in the human genome per generation

The structure of a human gene

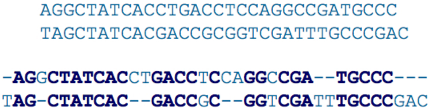
A gene starts with the promoter region, which is followed by a transcribed but non-coding region called 5' untranslated region (5' UTR). Then follows the initial exon which contains the start codon which is usually ATG. There is an alternating series of introns and internal exons, followed by the terminating exon, which contains the stop codon. It is followed by another non-coding region called the 3' UTR; at the end there is a polyadenylation (polyA) signal, i.e. a repetition of Adenine (example AAAAA). The intron/exon and exon/intron boundaries are conserved short sequences and called the acceptor and donor sites.




Networks
Computers
Modules
DSP
Gates
Physical layer
Tissues, cultures
Cells
Pathways
Biochemical reactions
CAK
Cdc2
P
CDS
Proteins, genes...

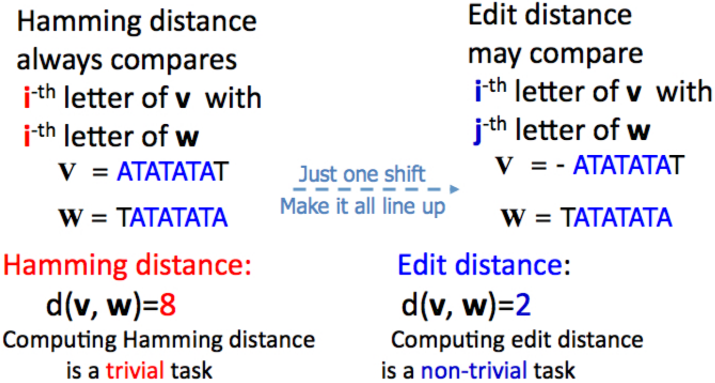
⋆TOPIC: The Biological information we extract by aligning 2 sequences

Alignment is a way of arranging two DNA or protein sequences to identify regions of similarity that are conserved among species. Each aligned sequence appears as a row within a matrix. Gaps are inserted between the residues (=amino acids) of each sequence so that identical or similar bases in different sequences are aligned in successive positions. Each gap spans one or more columns within the alignment matrix. Given two strings $x = x_1, x_2, , x_M$, $y = y_1, y_2, , y_N$, an alignment is an assignment of gaps to positions $0, , M$ in x, and $0, , N$ in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence.



```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC

-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```



Hamming distance always compares i-th letter of **v** with i-th letter of **w**

v = ATATATAT

w = TATATATA

Just one shift

Make it all line up

Edit distance may compare i-th letter of **v** with j-th letter of **w**

v = - ATATATAT

w = TATATATA

Hamming distance:

d(**v**, **w**)=8

Computing Hamming distance is a trivial task

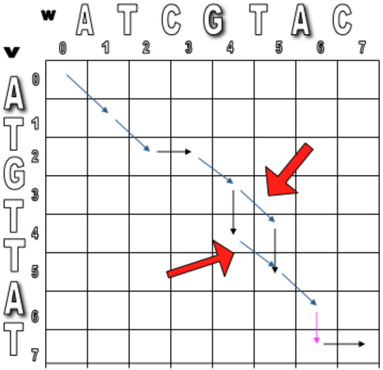
Edit distance:

d(**v**, **w**)=2

Computing edit distance is a non-trivial task

Figure: The Hamming distance is a column by column number of mismatches; the Edit distance between two strings is the minimum number of operations (insertions, deletions, and substitutions) to transform one string into the other

Alignment as a Path in the Edit Graph



first alignment

```
     0122345677
v=   AT_GTTAT_
w=   ATCGT_A_C
     0123455667
```

second alignment

```
     0122345677
v=   AT_GTTAT_
w=   ATCG_TA_C
     0123445667
```

The score of the alignment paths are 5.

Figure: Create a matrix M with one sequence as row header and the other sequence as column header Assign a 1 where the column and row site matches (diagonal segments), zero otherwise (horizontal or vertical segments); Sequence alignment can be viewed as a Path in the Edit Graph. The edit graph is useful to introduce the dynamic programming technique

Dynamic programming

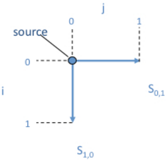
1. A method for reducing a complex problem to a set of identical sub-problems
2. The best solution to one sub-problem is independent from the best solution to the other sub-problem
3. Consider the Fibonacci Series: $F(n) = F(n-1) + F(n-2)$ where $F(0) = 0$ and $F(1) = 1$.
4. A recursive algorithm will take exponential time to find F(n) while a Dynamic Programming solution takes only n steps (linear time)
5. A recursive algorithm is likely to be polynomial if the sum of the sizes of the subproblems is bounded by kn.
6. If, however, the obvious division of a problem of size n results in n problems of size n-1 then the recursive algorithm is likely to have exponential growth.

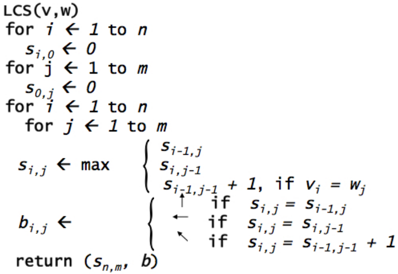
1. Dynamic programming can be thought of as being the reverse of recursion. Recursion is a top-down mechanism, we take a problem, split it up, and solve the smaller problems that are created.
2. Dynamic programming is a bottom-up mechanism: we solve all possible small problems and then combine them to obtain solutions for bigger problems.
3. The reason that this may be better is that, using recursion, it is possible that we may solve the same small subproblem many times. Using dynamic programming, we solve it once.
4. Needleman-Wunsch (global alignment, see later) algorithm turns string alignment into a problem in dynamic programming

The Longest Common Subsequence (LCS)

- The Longest Common Subsequence (LCS) problem is the simplest form of sequence alignment allows only insertions and deletions (no mismatches).
- Given two sequences $v = v_1\ v_2\ , v_m$ and $w = w_1\ w_2\ , w_n$. The LCS of v and w is a sequence of positions in v: $1 < i_1 < i_2 < < i_t < m$ and a sequence of positions in w: $1 < j_1 < j_2 < < j_t < n$ such that i_t letter of v equals to j_t-letter of w and t is maximal
- In the LCS problem, we score 1 for matches and 0 for indels
- In alignment: Consider penalising indels and mismatches with negative scores

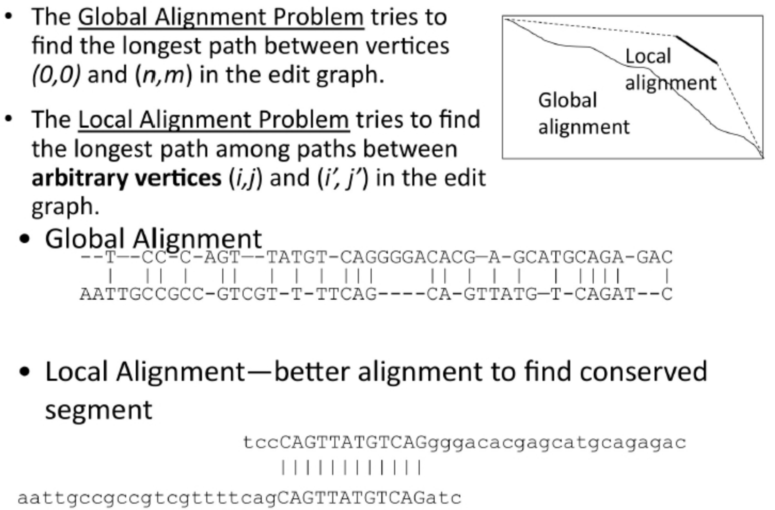
The Longest Common Subsequence





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

Figure: It takes O(nm) time to fill in the n by m dynamic programming matrix. The pseudocode consists of two nested for loops to build up a n by m matrix.


Local
alignment
Global
alignment

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

- Global Alignment

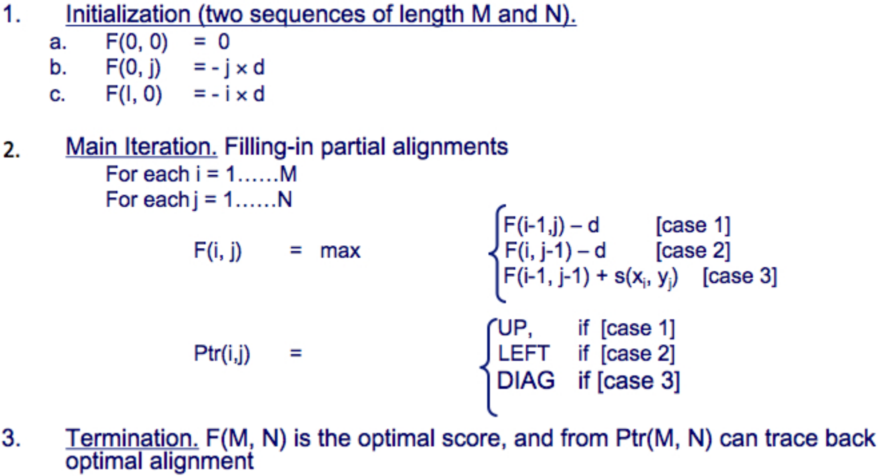
```
--T--CC-C-AGT--TATGT-CAGGGACACG-A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || | | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG-T-CAGAT--C
```

- Local Alignment—better alignment to find conserved segment

```
                  tccCAGTTATGTCAGggacacgagcatgcagagac
                     ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

Figure: The same sequences could be used in both alignments; we need to set the match score, the mismatch and gap penalties (next slide, d)

Needleman-Wunsch algorithm (Global alignment)



1. Initialization (two sequences of length M and N).
   a. $F(0, 0) = 0$
   b. $F(0, j) = - j \times d$
   c. $F(I, 0) = - i \times d$

2. Main Iteration. Filling-in partial alignments

   For each i = 1……M

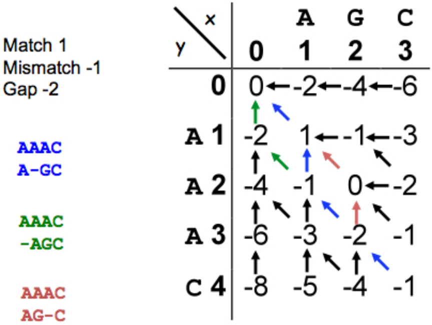
   For each j = 1……N

$$F(i, j) = \max \begin{cases} F(i-1,j) - d & \text{[case 1]} \\ F(i, j-1) - d & \text{[case 2]} \\ F(i-1, j-1) + s(x_i, y_j) & \text{[case 3]} \end{cases}$$

$$Ptr(i,j) = \begin{cases} \text{UP,} & \text{if [case 1]} \\ \text{LEFT} & \text{if [case 2]} \\ \text{DIAG} & \text{if [case 3]} \end{cases}$$

3. Termination. F(M, N) is the optimal score, and from Ptr(M, N) can trace back optimal alignment

Example



Match 1
Mismatch -1
Gap -2

AAAC
A-GC

AAAC
-AGC

AAAC
AG-C

y \ x	0	A 1	G 2	C 3
0	0	-2	-4	-6
A 1	-2	1	-1	-3
A 2	-4	-1	0	-2
A 3	-6	-3	-2	-1
C 4	-8	-5	-4	-1

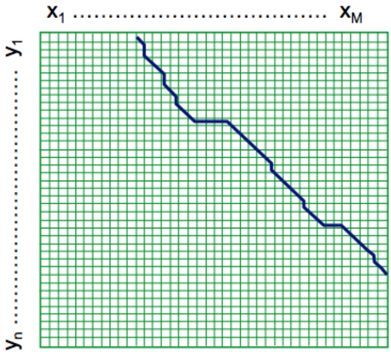
Figure: Given a m x n matrix, the overall complexity of computing all sub-values is $O(nm)$. The final optimal score is the value at position n,m. In this case we align the sequences AGC and AAAC.

How good is an alignment?

The score of an alignment is calculated by summing the rewarding scores for match columns that contain the same bases and the penalty scores for gaps and mismatch columns that contain different bases. A scoring scheme specifies the scores for matches and mismatches, which form the scoring matrix, and the scores for gaps, called the gap cost. There are two types of alignments for sequence comparison. Given a scoring scheme, calculating a global alignment is a kind of global optimization that forces the alignment to span the entire length of two query sequences, whereas local alignments just identify regions of high similarity within two sequences. The method of computing the entropy, explained in the multiple sequence alignment section could be used also for pairwise alignment.

Maybe it is OK to have an unlimited # of gaps in the beginning and end:

```
----------CTATCACCTGACCTCCAGGCCGATGCCCCTTCCGGC
GCGAGTTCATCTATCAC--GACCGC--GGTCG--------------
```



Changes:

1. <u>Initialization</u>

   For all i, j,

   $F(i, 0) = 0$

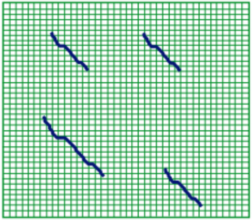
   $F(0, j) = 0$

2. <u>Termination</u>

$$F_{OPT} = \max \begin{cases} \max_i F(i, N) \\ \max_j F(M, j) \end{cases}$$

The local alignment: the Smith-Waterman algorithm

Idea: Ignore badly aligning regions: Modifications to Needleman-Wunsch



e.g. x = aaaacc**ccgggg**

y = **cccggg**aaccaacc

Initialization: $F(0, j) = F(i, 0) = 0$

Iteration:

$$F(i, j) = \max \begin{cases} 0 \\ F(i-1, j) - d \\ F(i, j-1) - d \\ F(i-1, j-1) + s(x_i, y_j) \end{cases}$$

Termination:

1. If we want the best local alignment…

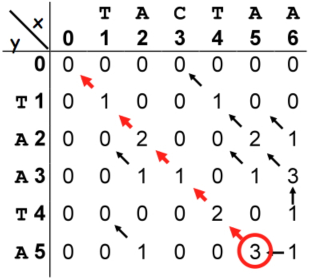
$$F_{OPT} = \max_{i,j} F(i, j)$$

2. If we want all local alignments scoring > t

For all i, j find $F(i, j) > t$, and trace back

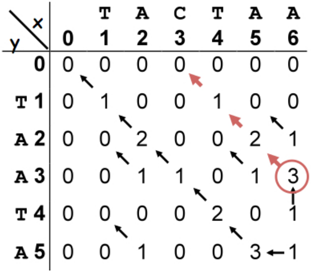
Example, Local alignment TAATA vs TACTAA

y = TAATA
x = TACTAA



y \ x	0	T 1	A 2	C 3	T 4	A 5	A 6
0	0	0	0	0	0	0	0
T 1	0	1	0	0	1	0	0
A 2	0	0	2	0	0	2	1
A 3	0	0	1	1	0	1	3
T 4	0	0	0	0	2	0	1
A 5	0	0	1	0	0	3	1

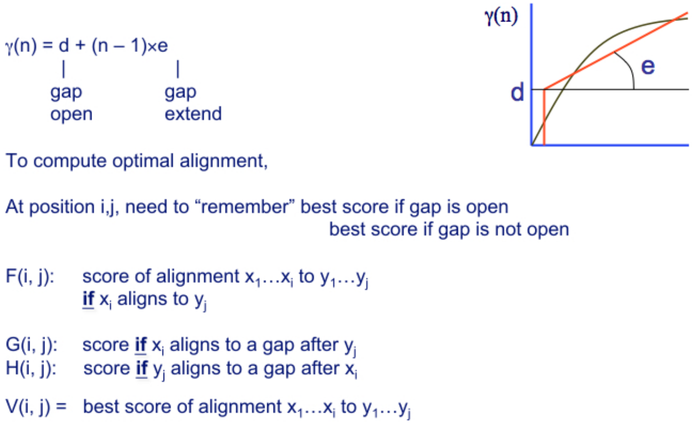
```
y =    TAATA
x = TACTAA
```



y \ x	0	T 1	A 2	C 3	T 4	A 5	A 6
0	0	0	0	0	0	0	0
T 1	0	1	0	0	1	0	0
A 2	0	0	2	0	0	2	1
A 3	0	0	1	1	0	1	3
T 4	0	0	0	0	2	0	1
A 5	0	0	1	0	0	3	1

Affine: two penalties for gap insertion

if there are many gaps we do not want to penalise too much; so we think at due penalties: one for the first gap (opening) and one, smaller, for the following required gaps.



$$\gamma(n) = \underset{\text{gap open}}{d} + (n-1)\times\underset{\text{gap extend}}{e}$$

To compute optimal alignment,

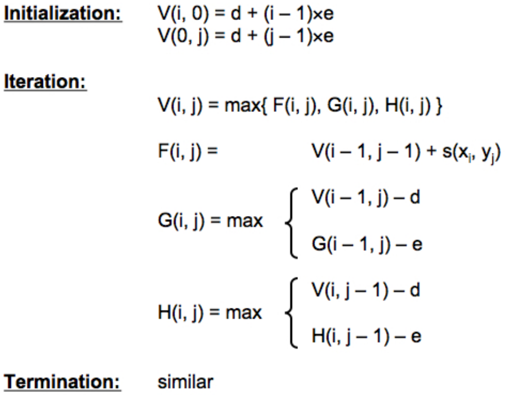
At position i,j, need to “remember” best score if gap is open
best score if gap is not open

F(i, j): score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$
<u>if</u> x_i aligns to y_j

G(i, j): score **<u>if</u>** x_i aligns to a gap after y_j
H(i, j): score **<u>if</u>** y_j aligns to a gap after x_i

V(i, j) = best score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$

Time complexity - As before O(nm), as we only compute four matrices instead of one. Space complexity - There's a need to save four matrices (for F, G, H and V respectively) during the computation. Hence, O(nm) space is needed, for the trivial implementation.



Initialization: $V(i, 0) = d + (i - 1)\times e$

$V(0, j) = d + (j - 1)\times e$

Iteration:

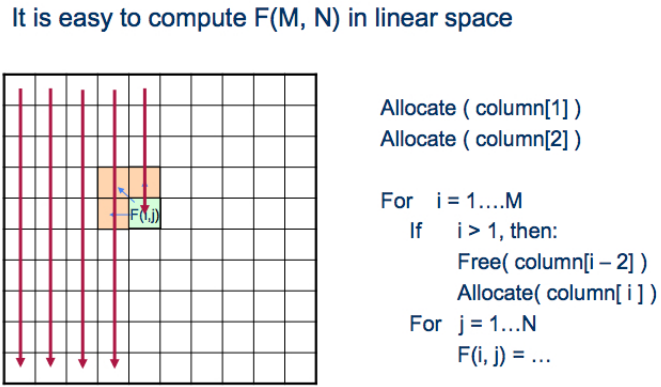
$$V(i, j) = \max\{ F(i, j), G(i, j), H(i, j) \}$$

$$F(i, j) = V(i - 1, j - 1) + s(x_i, y_j)$$

$$G(i, j) = \max \begin{cases} V(i - 1, j) - d \\ G(i - 1, j) - e \end{cases}$$

$$H(i, j) = \max \begin{cases} V(i, j - 1) - d \\ H(i, j - 1) - e \end{cases}$$

Termination: similar



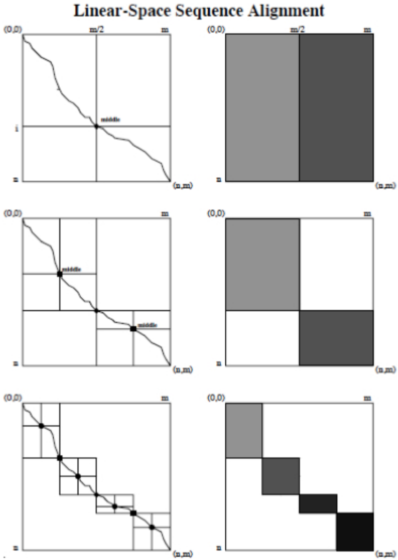
It is easy to compute F(M, N) in linear space

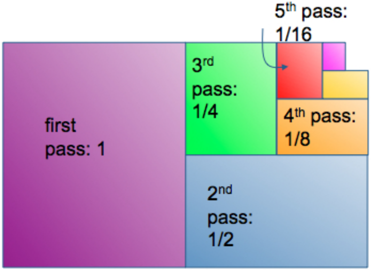
```
Allocate ( column[1] )
Allocate ( column[2] )

For   i = 1….M
    If     i > 1, then:
           Free( column[i – 2] )
           Allocate( column[ i ] )
    For   j = 1…N
           F(i, j) = …
```

Figure: Space complexity of computing just the score itself is O(n); we only need the previous column to calculate the current column, and we can then throw away that previous column once we have done using it

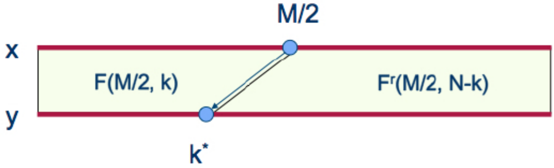
Alignment in linear space, Hirschberg algorithm



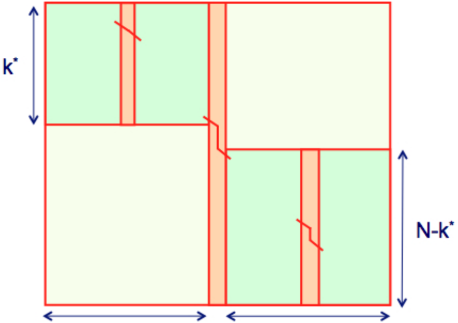


Define (m/2,k) as the vertex where the longest path crosses the middle column

$F(M, N) = \max_{K=0,N} \left(F\left(M/2, K\right) + F^r\left(M/2, N - K\right)\right)$

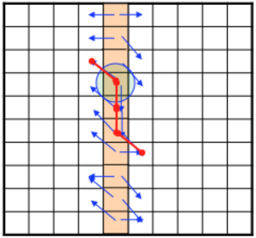


- Iterate this procedure to the left and right!



Now, we can find k^* maximizing $F(M/2, k) + F^r(M/2, k)$

Also, we can trace the path exiting column M/2 from k^*



<u>Conclusion:</u> In O(NM) time, O(N) space,
we found optimal alignment path at column M/2

Hirschberg's Linear-space algorithm:

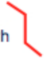
```
MEMALIGN(l, l', r, r'):                    (aligns $x_l \ldots x_{l'}$ with $y_r \ldots y_{r'}$)

1.   Let $h = \lceil (l'-l)/2 \rceil$
2.   Find in Time $O((l' - l) \times (r'-r))$, Space $O(r'-r)$
     the optimal path,      $L_h$, at column h
     Let $k_1$ = pos'n at column h – 1 where $L_h$ enters
         $k_2$ = pos'n at column h + 1 where $L_h$ exits

1.   MEMALIGN(l, h-1, r, $k_1$)

2.   Output $L_h$

3.   MEMALIGN(h+1, l', $k_2$, r')
```



Time, Space analysis of Hirschberg's algorithm:

To compute optimal path at middle column,

For box of size M × N,

Space: 2N

Time: cMN, for some constant c

Then, left, right calls cost $c(M/2 \times k^* + M/2 \times (N-k^*)) = cMN/2$

All recursive calls cost

Total Time: $cMN + cMN/2 + cMN/4 + \ldots.. = 2cMN = O(MN)$

Total Space: $O(N)$ for computation,
$O(N+M)$ to store the optimal alignment

⋆Topic: Biology of RNA

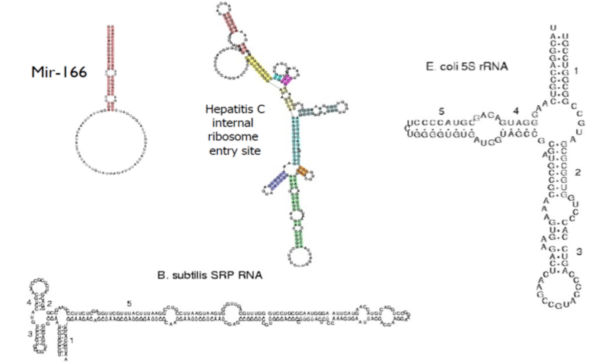


Figure: Examples of RNA molecules in nature; many molecules of RNA do not translate into proteins; the molecules fold into 2d (secondary) and 3d (tertiary) structures and regulate cell processes by interacting among each other and with proteins

Folding i.e. intra chain alignment of a RNA molecule

The intrachain folding of RNA reveals RNA Secondary Structure
This tells which bases are paired in the subsequence from x_i to x_j
Every optimal structure can be built by extending optimal substructures.

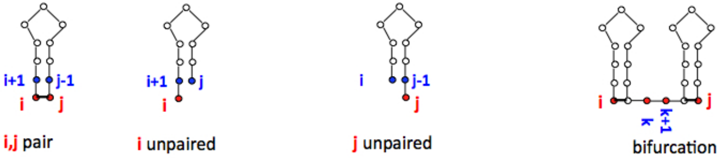


Figure: Set of paired positions on interval [i,j]. Suppose we know all optimal substructures of length less than $j - i + 1$. The optimal substructure for $[i, j]$ must be formed in one of four ways: i,j paired; i unpaired; j unpaired; combining two substructures. Note that each of these consists of extending or joining substructures of length less than $j - i + 1$

Nussinov dynamic programming algorithm for RNA folding

1. Let $\gamma(i,j)$ be the maximum number of base pairs in a folding of subsequence S[i . . . j].
2. for $1 \leq i \leq n$ and $i < j \leq n$: $\gamma(i,i) = 0$;
   for $i = 1, ..., n$ $\gamma(i, i-1) = 0$
3. starting from $i = 2, ..., n$

$$\gamma\left(i,j\right) = max \begin{cases} \gamma\left(i+1,j\right) \\ \gamma\left(i,j-1\right) \\ \gamma\left(i+1,j-1\right) + \delta\left(i,j\right) \\ max_{i<k<j}\left[\gamma\left(i,k\right) + \gamma\left(k+1,j\right)\right] \end{cases}$$

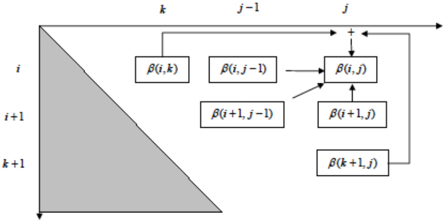
4. Where $\delta(i,j) = 1$ if x_i and x_j are a complementary base pair i.e. (A, U) or (C, G), and $\delta(i,j) = 0$, otherwise.

There are $O(n^2)$ terms to be computed, each requiring calling of $O(n)$ already computed terms for the case of bifurcation. Thus overall complexity is $O(n^3)$ time and $O(n^2)$ space.

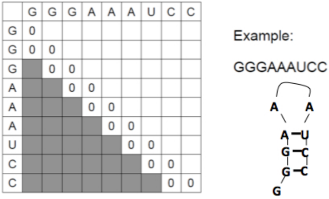
Nussinov algorithm for RNA folding

Note that only the upper (or lower) half of the matrix needs to be filled. Therefore, after initialization the recursion runs from smaller to longer subsequences as follows:

```
for l = 1 to n do
    for i = 1 to $(n+1-l)$ do
        j = i + l
        compute $\gamma(i,j)$
    end for
end for
```



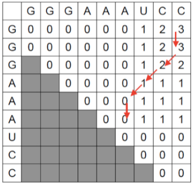
Nussinov algorithm: example



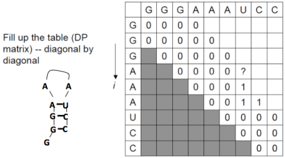
	G	G	G	A	A	A	U	C	C
G	0								
G	0	0							
G		0	0						
A			0	0					
A				0	0				
A					0	0			
U						0	0		
C							0	0	
C								0	0

Example:

GGGAAAUCC



	G	G	G	A	A	A	U	C	C
G	0	0	0	0	0	0	1	2	3
G	0	0	0	0	0	0	1	2	3
G		0	0	0	0	0	1	2	2
A			0	0	0	0	1	1	1
A				0	0	0	1	1	1
A					0	0	1	1	1
U						0	0	0	0
C							0	0	0
C								0	0



Fill up the table (DP matrix) -- diagonal by diagonal

A A
A–U
G–C
G–C
G
i

	G	G	G	A	A	A	U	C	C
G	0	0	0	0					
G	0	0	0	0	0				
G		0	0	0	0	0			
A			0	0	0	0	?		
A				0	0	0	1		
A					0	0	1	1	
U						0	0	0	0
C							0	0	0
C								0	0

Figure: order: top left, bottom left, right: a matrix will be filled along the diagonals and the solution can be recovered through a traceback step.

⋆Topic: Homology search algorithms: The Biological problem

The sequence structures of genes and proteins are conserved in nature. It is common to observe strong sequence similarity between a protein and its counterpart in another species that diverged hundreds of millions of years ago. Accordingly, the best method to identify the function of a new gene or protein is to find its sequence- related genes or proteins whose functions are already known. The Basic Local Alignment Search Tool (BLAST) is a computer program for finding regions of local similarity between two DNA or protein sequences. It is designed for comparing a query sequence against a target database. It is a heuristic that finds short matches between query and database sequences and then attempts to start alignments from these seed hits. BLAST is arguably the most widely used program in bioinformatics. By sacrificing sensitivity for speed, it makes sequence comparison practical on huge sequence databases currently available.

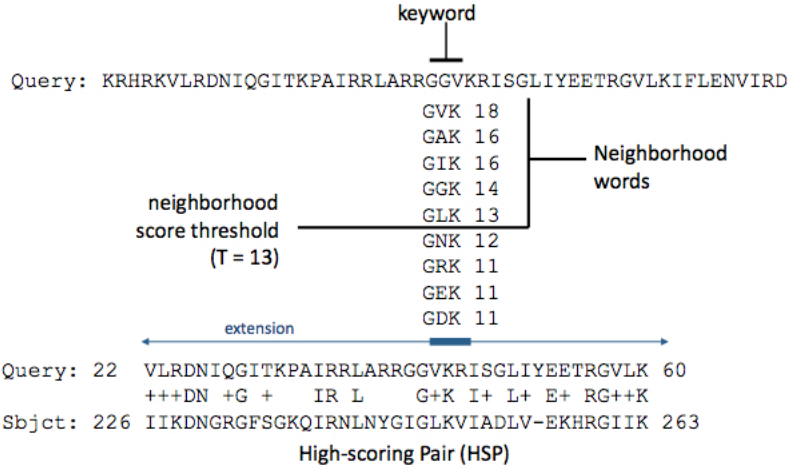
BLAST programs (Basic Local Alignment Search Tools

While Dynamic Programming (DP) is a nice way to construct alignments, it will often be too slow. Since the DP is $O(n^2)$, matching two 3, 000, 000, 000 length sequences would take about $9x10^{18}$ operations. BLAST is an alignment algorithm which runs in O(n) time. For sequences of length 3, 000, 000, 000, this will be around 3, 000, 000, 000 times faster. The key to BLAST is that we only actually care about alignments that are very close to perfect. A match of 70% is worthless; we want something that matches 95% or 99% or more. What this means is that correct (near perfect) alignments will have long substrings of nucleotides that match perfectly. Most popular Blast-wise algorithms use a seed-and-extend approach that operates in two steps: 1. Find a set of small exact matches (called seeds) 2. Try to extend each seed match to obtain a long inexact match.

The steps are as follows:

1. Pre-processing step of BLAST is to make sure that all substrings of W consecutive nucleotides will be included in a database (or in a hash table). These are called the W-mers of the database.
2. Split query into overlapping words of length W (the W-mers)
3. Find a neighborhood of similar words for each word (see below)
4. Lookup each word in the neighborhood in a hash table to find where in the database each word occurs. Call these the seeds, and let S be the collection of seeds.
5. Extend the seeds in S until the score of the alignment drops off below a threshold.
6. Report matches with overall highest scores

BLAST permits a trade off between speed and sensitivity, with the setting of a "threshold" parameter T. A higher value of T yields greater speed, but also an increased probability of missing weak similarities



```
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
```

To speed up the homology search process, BLAST employs a filtration strategy: It first scans the database for length-w word matches of alignment score at least T between the query and target sequences and then extends each match in both ends to generate local alignment (in the sequences) whose alignment score is larger than a threshold S. The matches are called high-scoring segment pairs (HSPs). BLAST outputs a list of HSPs together with E-values that measure how frequent such HSPs would occur by chance. A HSP has the property that it cannot be extended further to the left or right without the score dropping significantly below the best score achieved on part of the HSP. The original BLAST algorithm performs the extension without gaps. Variants are gapped Blast, psi-blast and others.

Statistical significance in Blast

- Assume that the length m and n of the query and database respectively are sufficiently large; a segment-pair (s, t) consists of two segments, one in m (say the amino acid string: VALLAR) and one in n (say PAMMAR), of the same length. We think of s and t as being aligned without gaps and score this alignment using a substitution score; the alignment score for (s, t) is denoted by $\sigma(s, t)$.
- Given a cutoff score x, a segment pair (s, t) is called a high-scoring segment pair (HSP), if it is locally maximal and $\sigma(s, t) \geq x$ and the goal of BLAST is to compute all HSPs.
- The BLAST algorithm has three parameters: the word size W, the word similarity threshold T and the minimum match score x.

For protein sequences, BLAST operates as follows

The list of all words of length W that have similarity $\geq$ T to some word in the query sequence m is generated. The database sequence n is scanned for all hits t of words s in the list. Each such seed (s, t) is extended until its score $\sigma(s, t)$ falls a certain distance below the best score found for shorter extensions and then all best extensions are reported that have score $\geq$ x. In practice, W is around 4 for proteins.

The list of all words of length W that have similarity $\geq$ T to some word in the query sequence m can be produced in time proportional to the number of words in the list. These are placed in a keyword tree and then, for each word in the tree, all exact locations of the word in the database n are detected in time linear to the length of n. The original version of BLAST did not allow indels, making hit extension very fast.

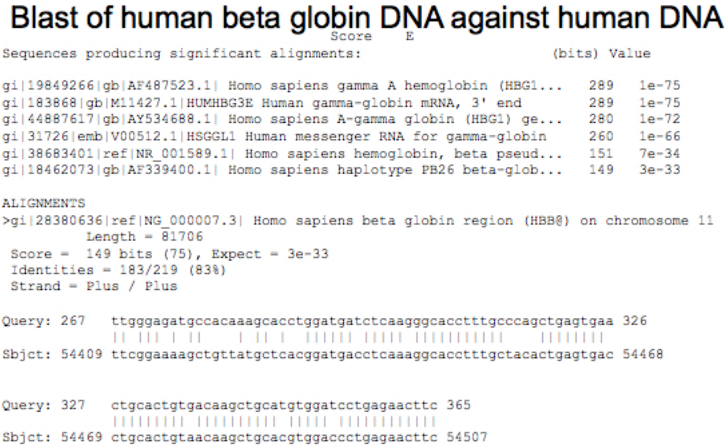
Note that the use of seeds of length W and the termination of extensions with fading scores are both steps that speed up the algorithm, but also imply that BLAST is not guaranteed to find all HSPs.

For DNA sequences, BLAST operates as follows

- For DNA sequences, BLAST operates as follows: The list of all words of length W in the query sequence m is generated. The database n is scanned for all hits of words in this list. Blast uses a two-bit encoding for DNA. This saves space and also search time, as four bases are encoded per byte. In practice, W is around 12 for DNA.
- HSP scores are characterized by two parameters, W and λ. The expected number of HSPs with score at least S is given by the E-value, which is: $E(S) = Wmne^{-\lambda S}$.
- Essentially, W and λ are scaling-factors for the search space and for the scoring scheme, respectively.
- As the E-value depends on the choice of the parameters W and λ, one cannot compare E-values from different BLAST searches.

- For a given HSP (s, t) we transform the raw score $S = \sigma(s, t)$ into a bit-score thus: $S' = \frac{\lambda S - lnW}{ln2}$. Such bit-scores can be compared between different BLAST searches. To see this, solve for S in the previous equation and then plug the result into the original E-value.
- E-values and bit scores are related by $E = mn2^{-S'}$
- The number of random HSPs (s, t) with $\sigma(s, t) \geq x$ can be described by a Poisson distribution. Hence the probability of finding exactly k HSPs with a score $\geq S$ is given by $P(k) = \frac{E^k}{k!}e^{-E}$
- The probability of finding at least one HSP by chance is $P = 1 - P(X = 0) = 1 - e^{-E}$, called the P-value, where E is the E-value for S.
- BLAST reports E-values rather than P-values as it is easier, for example, to interpret the difference between an E-value of 5 and 10, than to interpret the difference between a P-value of 0.993 and 0.99995. For small E-values < 0.01, the two values are nearly identical.

Example of Blast



Blast of human beta globin DNA against human DNA

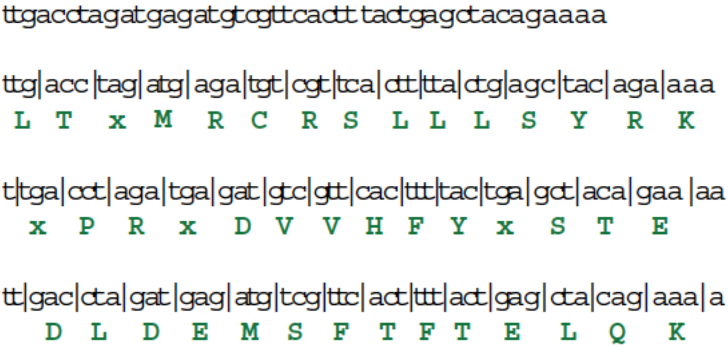
```
                                                                 Score    E
Sequences producing significant alignments:                      (bits) Value

gi|19849266|gb|AF487523.1| Homo sapiens gamma A hemoglobin (HBG1...   289   1e-75
gi|183868|gb|M11427.1|HUMHBG3E Human gamma-globin mRNA, 3' end        289   1e-75
gi|44887617|gb|AY534688.1| Homo sapiens A-gamma globin (HBG1) ge...   280   1e-72
gi|31726|emb|V00512.1|HSGGL1 Human messenger RNA for gamma-globin     260   1e-66
gi|38683401|ref|NR_001589.1| Homo sapiens hemoglobin, beta pseud...   151   7e-34
gi|18462073|gb|AF339400.1| Homo sapiens haplotype PB26 beta-glob...   149   3e-33

ALIGNMENTS
>gi|28380636|ref|NG_000007.3| Homo sapiens beta globin region (HBB@) on chromosome 11
          Length = 81706
 Score =  149 bits (75), Expect = 3e-33
 Identities = 183/219 (83%)
 Strand = Plus / Plus

Query: 267   ttgggagatgccacaaagcacctggatgatctcaagggcacctttgcccagctgagtgaa 326
             || ||| | ||    | || |  |||||| ||||| |||||||||||    ||||||||
Sbjct: 54409 ttcggaaaagctgttatgctcacggatgacctcaaaggcacctttgctacactgagtgac 54468


Query: 327   ctgcactgtgacaagctgcatgtggatcctgagaacttc 365
             ||||||||| |||||||||| ||||| ||||||||||||
Sbjct: 54469 ctgcactgtaacaagctgcacgtggaccctgagaacttc 54507
```



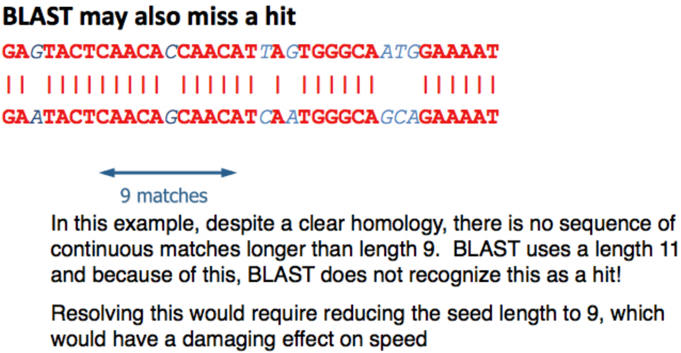
```
ttgacctagatgagatgtcgttcactt tactgagctacagaaaa

ttg|acc |tag|atg|aga|tgt|cgt|tca|ctt|tta|ctg|agc|tac|aga|aaa
 L   T   x   M   R   C   R   S   L   L   L   S   Y   R   K

t|tga|cct|aga|tga|gat |gtc|gtt|cac|ttt|tac|tga|gct|aca|gaa |aa
   x   P   R   x   D    V   V   H   F   Y   x   S   T   E

tt|gac|cta|gat |gag|atg|tcg|ttc|act|ttt|act|gag|cta|cag|aaa|a
    D   L   D    E   M   S   F   T   F   T   E   L   Q   K
```

Figure: Blast DNA query (top) against a database of proteins will process all the potential triplets forming codons



BLAST may also miss a hit

```
GAGTACTCAACACCAACATTAGTGGGCAATGGAAAAT
|| ||||||||| |||||| | ||||||   ||||||
GAATACTCAACAGCAACATCAATGGGCAGCAGAAAAT
```

In this example, despite a clear homology, there is no sequence of continuous matches longer than length 9. BLAST uses a length 11 and because of this, BLAST does not recognize this as a hit!

Resolving this would require reducing the seed length to 9, which would have a damaging effect on speed

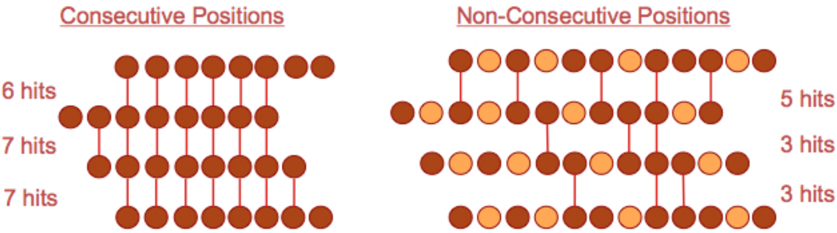
Figure: Example of Blast Pitfalls

Patternhunter

The biggest problem for BLAST is low sensitivity (and low speed). Massive parallel machines are built to do Smith Waterman exhaustive dynamic programming. A spaced seed is formed by two words, one from each input sequence, that match at positions specified by a fixed pattern and one don't care symbol respectively. For example, the pattern 1101 specifies that the first, second and four-th positions must match and the third one contain a mismatch. PatternHunter (PH) was the first method that used carefully designed spaced seeds to improve the sensitivity of DNA local alignment. Spaced seeds have been shown to improve the efficiency of lossless filtration for approximate pattern matching, namely for the problem of detecting all matches of a string of length m with q possible substitution errors.

Blast vs PH vs PH II

If you want to speed up, have to use a longer seed. However, we now face a dilemma: increasing seed size speeds up, but looses sensitivity; decreasing seed size gains sensitivity, but looses speed. How do we increase sensitivity and speed simultaneously? Spaced Seed: nonconsecutive matches and optimized match positions. Represent BLAST seed by 11111111111; Spaced seed: 111010010100110111 where 1 means a required match and 0 means dont care position. This simple change makes a huge difference: significantly increases hit to homologous region while reducing bad hits. Spaced seeds give PH a unique opportunity of using several optimal seeds to achieve optimal sensitivity, this was not possible by BLAST technology. PH II uses multiple optimal seeds; it approaches Smith-Waterman sensitivity while is 3000 times faster. Example: Smith-Waterman (SSearch): 20 CPU-days, PatternHunter II with 4 seeds: 475 CPU-seconds: 3638 times faster than Smith-Waterman dynamic programming at the same sensitivity



On a 70% conserved region:

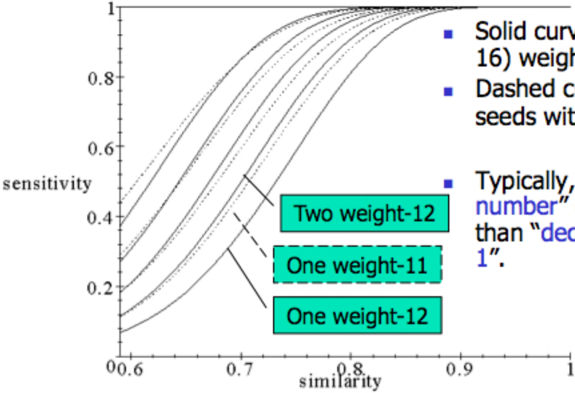
	Consecutive	Non-consecutive
Expected # hits:	1.07	0.97
Prob[at least one hit]:	0.30	0.47

- 111010010100110111 (called a model)
  - Eleven required matches (weight=11)
  - Seven "don't care" positions

```
GAGTACTCAACACCAACATTAGTGGCAATGGAAAAT…
|| ||||||||| ||||| || |||||   ||||||
GAATACTCAACAGCAACACTAATGGCAGCAGAAAAT…
       111010010100110111
```

- Hit = all the required matches are satisfied.
- BLAST seed model = 11111111111

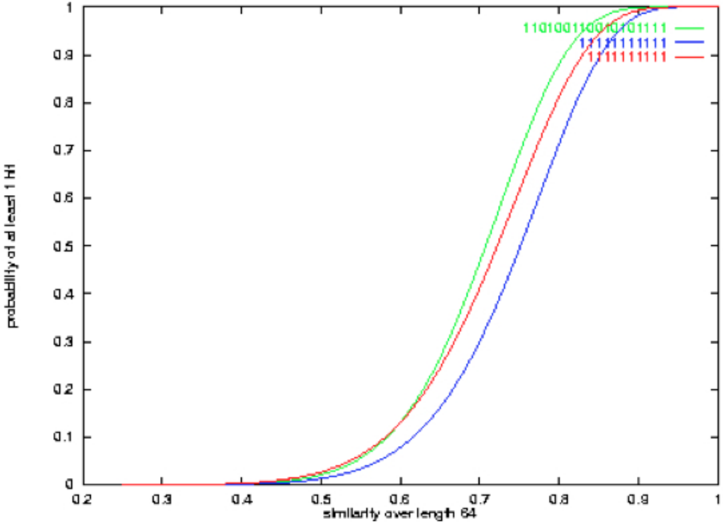
```
111010010100110111
 111010010100110111
  111010010100110111
   111010010100110111
    111010010100110111
     111010010100110111
      111010010100110111
        ......
```



- Solid curves: Multiple (1, 2, 4, 8, 16) weight-12 spaced seeds.
- Dashed curves: Optimal spaced seeds with weight = 11, 10, 9, 8.

- Typically, "Doubling the seed number" gains better sensitivity than "decreasing the weight by 1".

Sensitivity: PH weight 11 seed vs BLAST 11 & 10



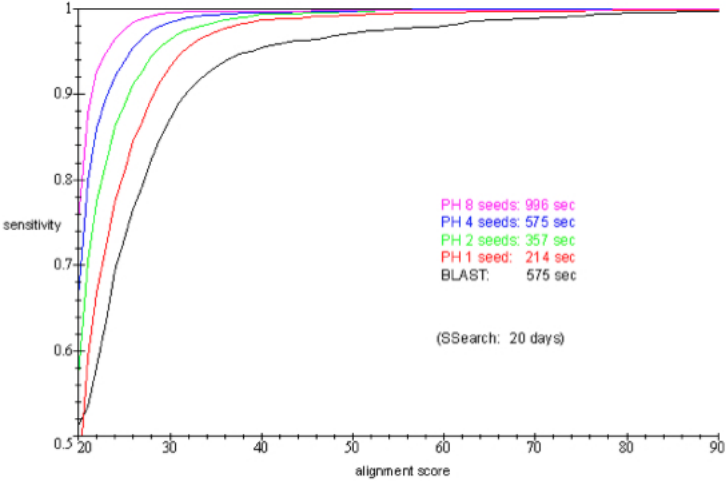
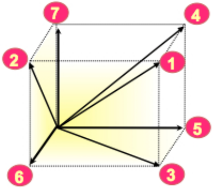


Figure: sensitivity versus alignment score

Difficulties of extending dynamic programming to n sequences

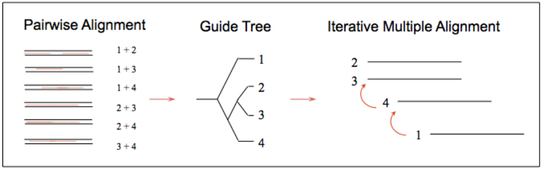
- For two sequences, there are three ways to extend an alignment
- for n sequences, a n-dimensional dynamic programming hypercube has to be computed and for each entry we have to evaluate $(2^n - 1)$ predecessors.
- Given 3 sequences, the figure below shows a three-dimensional alignment path matrix: there are $= (2^3 - 1) = 7$ ways to extend an alignment.



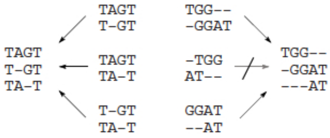
1	2	3	4	5	6	7
ABC	AB-	ABC	ABC	ABC	AB-	AB-
ABC	ABC	AB-	ABC	AB-	AB-	ABC
ABC	ABC	ABC	AB-	AB-	ABC	AB-

Progressive alignment

- Progressive alignment methods are heuristic in nature. They produce multiple alignments from a number of pairwise alignments.
- Perhaps the most widely used algorithm of this type is CLUSTALW.
- Given N sequences, align each sequence against each other and obtain a similarity matrix; Similarity = exact matches / sequence length (percent identity)
- Create a guide tree using the similarity matrix; the tree is reconstructed using clustering methods such as UPGMA or neighbor-joining (explained later).
- Progressive Alignment guided by the tree.



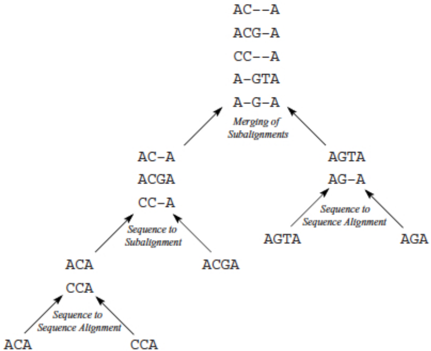
Not all the pairwise alignments build well into multiple sequence alignment; the progressive alignment greedily builds a final alignment along the guide tree using a given method to merge sub-alignments.



```
TAGT      TGG--
T-GT      -GGAT

TAGT      -TGG         TGG--
T-GT      AT--         -GGAT
TA-T                   ---AT
TAGT      GGAT
TA-T      --AT

T-GT
TA-T
```



```
AC--A
ACG-A
CC--A
A-GTA
A-G-A
```

Merging of Subalignments

```
AC-A      AGTA
ACGA      AG-A
CC-A
```

Sequence to Subalignment

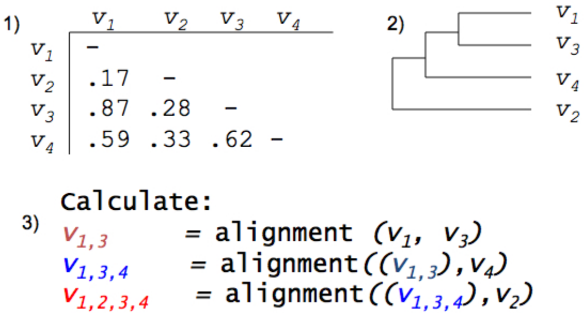
Sequence to Sequence Alignment

```
ACA       ACGA      AGTA      AGA
CCA
```

Sequence to Sequence Alignment

```
ACA       CCA
```

Progressive alignment



1)

	v_1	v_2	v_3	v_4
v_1	–			
v_2	.17	–		
v_3	.87	.28	–	
v_4	.59	.33	.62	–

3) Calculate:

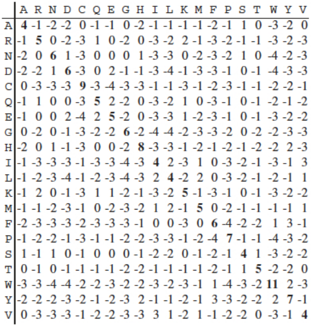
$v_{1,3}$ = alignment (v_1, v_3)

$v_{1,3,4}$ = alignment$((v_{1,3}),v_4)$

$v_{1,2,3,4}$ = alignment$((v_{1,3,4}),v_2)$

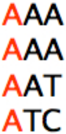
Figure: Progressive alignment of 4 sequences: 1) pairwise alignment; 2) pairwise alignment score analysis; tree showing the best order of progressive alignment, 3) building up the alignment

Blosum is a symmetric amino acid replacement matrix used as scoring matrix in Blast search and in phylogeny. Starting from a MSA of conserved portions of protein sequences we compute p_{ij} the probability of two amino acids i and j replacing each other in each column, and p_i and p_j are the background probabilities of finding the amino acids i and j in any protein sequence. Then we compute: $Score_{ij} = (k^{-1}) log(p_{ij}/p_i p_j)$ where the k is a scaling factor.



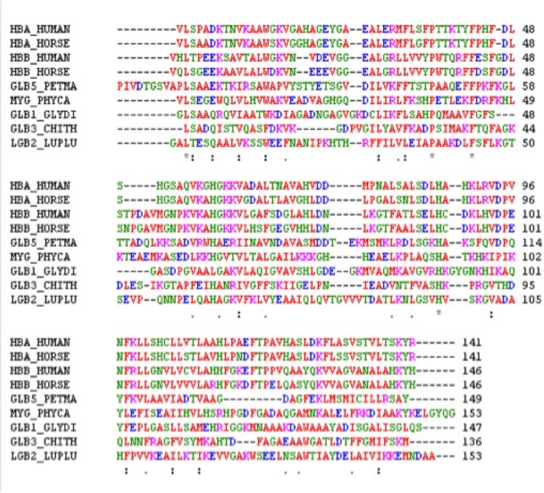
	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**4**	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	**5**	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	**6**	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	**6**	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	**9**	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	**5**	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	**5**	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	**6**	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	**8**	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	**4**	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	**4**	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	**5**	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	**5**	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	**6**	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	**7**	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	**4**	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	**5**	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	**11**	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	**7**	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	**4**

Entropy measure of a multiple alignment of 4 sequences of three bases each



```
AAA
AAA
AAT
ATC
```

Compute the frequencies for the occurrence of each letter in each column of multiple alignment pA = 1, pT=pG=pC=0 (1st column)
pA = 0.75, pT = 0.25, pG=pC=0 (2nd column)
pA = 0.50, pT = 0.25, pC=0.25 pG=0 (3rd column)
Compute entropy of each column: $E = -\sum_{X=A,C,G,T} p_x log\,(p_x)$
Entropy for a multiple alignment is the sum of entropies of its columns



```
HBA_HUMAN       ---------VLSPADKTNVKAAWGKVGAHAGEYGA--EALERMFLSFPTTKTYFPHF-DL 48
HBA_HORSE       ---------VLSAADKTNVKAAWSKVGGHAGEYGA--EALERMFLGFPTTKTYFPHF-DL 48
HBB_HUMAN       --------VHLTPEEKSAVTALWGKVN--VDEVGG--EALGRLLVVYPWTQRFFESFGDL 48
HBB_HORSE       --------VQLSGEEKAAVLALWDKVN--EEEVGG--EALGRLLVVYPWTQRFFDSFGDL 48
GLB5_PETMA      PIVDTGSVAPLSAAEKTKIRSAWAPVYSTYETSGV--DILVKFFTSTPAAQEFFPKFKGL 58
MYG_PHYCA       ---------VLSEGEWQLVLHVWAKVEADVAGHGQ--DILIRLFKSHPETLEKFDRFKHL 49
GLB1_GLYDI      ---------GLSAAQRQVIAATWKDIAGADNGAGVGKDCLIKFLSAHPQMAAVFGFS--- 48
GLB3_CHITH      ----------LSADQISTVQASFDKVK------GDPVGILYAVFKADPSIMAKFTQFAGK 44
LGB2_LUPLU      --------GALTESQAALVKSSWEEFNANIPKHTH--RFFILVLEIAPAAKDLFSFLKGT 50
                          *:  :   :   :  .             :  .:   *     *

HBA_HUMAN       S-----HGSAQVKGHGKKVADALTNAVAHVDD-----MPNALSALSDLHA--HKLRVDPV 96
HBA_HORSE       S-----HGSAQVKAHGKKVGDALTLAVGHLDD-----LPGALSNLSDLHA--HKLRVDPV 96
HBB_HUMAN       STPDAVMGNPKVKAHGKKVLGAFSDGLAHLDN-----LKGTFATLSELHC--DKLHVDPE 101
HBB_HORSE       SNPGAVMGNPKVKAHGKKVLHSFGEGVHHLDN-----LKGTFAALSELHC--DKLHVDPE 101
GLB5_PETMA      TTADQLKKSADVRWHAERIINAVNDAVASMDDT--EKMSMKLRDLSGKHA--KSFQVDPQ 114
MYG_PHYCA       KTEAEMKASEDLKKHGVTVLTALGAILKKKGH-----HEAELKPLAQSHA--TKHKIPIK 102
GLB1_GLYDI      -----GASDPGVAALGAKVLAQIGVAVSHLGDE--GKMVAQMKAVGVRHKGYGNKHIKAQ 101
GLB3_CHITH      DLES-IKGTAPFEIHANRIVGFFSKIIGELPN-----IEADVNTFVASHK---PRGVTHD 95
LGB2_LUPLU      SEVP--QNNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHV---SKGVADA 105
                           .   .  :   .                  .  .   *       :

HBA_HUMAN       NFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR------ 141
HBA_HORSE       NFKLLSHCLLSTLAVHLPNDFTPAVHASLDKFLSSVSTVLTSKYR------ 141
HBB_HUMAN       NFRLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH------ 146
HBB_HORSE       NFRLLGNVLVVVLARHFGKDFTPELQASYQKVVAGVANALAHKYH------ 146
GLB5_PETMA      YFKVLAAVIADTVAAG---------DAGFEKLMSMICILLRSAY------- 149
MYG_PHYCA       YLEFISEAIIHVLHSRHPGDFGADAQGAMNKALELFRKDIAAKYKELGYQG 153
GLB1_GLYDI      YFEPLGASLLSAMEHRIGGKMNAAAKDAWAAAYADISGALISGLQS----- 147
GLB3_CHITH      QLNNFRAGFVSYMKAHTD---FAGAEAAWGATLDTFFGMIFSKM------- 136
LGB2_LUPLU      HFPVVKEAILKTIKEVVGAKWSEELNSAWTIAYDELAIVIKKEMNDAA--- 153
                 :  .   :   :            . .       .   :
```

Figure: The globin proteins from different species could be aligned because they have many similar substrings

Insight into protein structure (3D graph) from MSA analysis

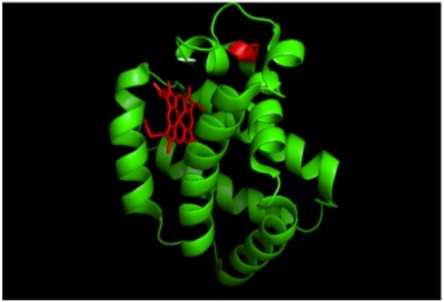


Figure: Human globin 3D structure. Although DNA sequences mutate and diverge, because of the small amount of changes all the globin sequences in the alignment of the previous slide are likely to have the same or very similar structure. Columns rich of gaps often correspond to unstructured regions (loops); conserved regions often correspond to binding sites or regions where one protein interacts with a DNA site or with another protein

Aligning short reads against a reference genome

- The current sequencing procedures are characterized by highly parallel operations, much lower cost per base, but (unfortunately) they produce short stretches of DNA bases, called reads (usually 35-400bp). Todays machines are commonly referred to as short-read sequencers or next-generation sequencers (NGS)
- Given a reference genome and a set of reads, we need to find at least one "good" local alignment for each read
- Genomes are too large and reads are too many for direct approaches like dynamic programming
- to map reads on the genome, the Burrows- Wheleer transform, BWT (a text compression method) is used

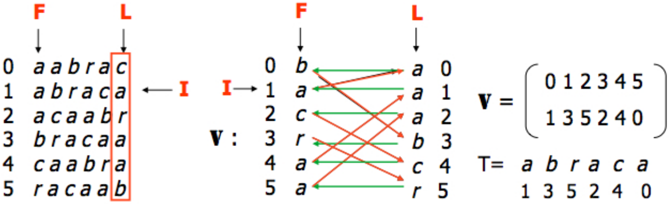
Burrows-Wheeler Transform

INPUT (example): T = "abraca"; then we sort lexicographically all the cyclic shifts of T

For all i ≠ I, the character L[i] is followed in T by F[i]; for any character ch, the i-th occurrence of ch in F corresponds to the i-th occurrence of ch in L.

OUTPUT: BWT(T)=caraab and the index I=1, that denotes the position of the original word T after the lexicographical sorting.

The Burrows-Wheeler Transform is reversible, in the sense that, given BWT(T) and an index I, it is possible to recover the original word T.



	F				L
0	a	a	b	r	a c
1	a	b	r	a	c a
2	a	c	a	a	b r
3	b	r	a	c	a a
4	c	a	a	b	r a
5	r	a	c	a	a b

$$\mathbf{V} = \begin{pmatrix} 0\,1\,2\,3\,4\,5 \\ 1\,3\,5\,2\,4\,0 \end{pmatrix}$$

T= a b r a c a
1 3 5 2 4 0

Burrows-Wheeler Transform in alignment: example

Reversible permutation used originally in compression
Once BWT(T) is built, all else shown here is discarded

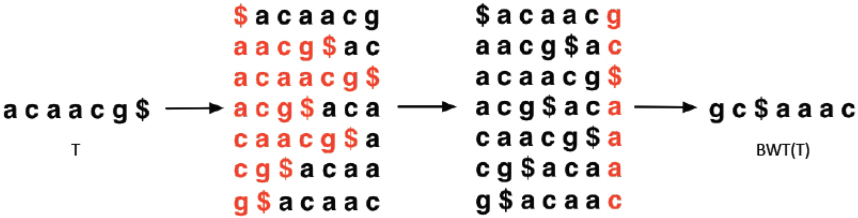
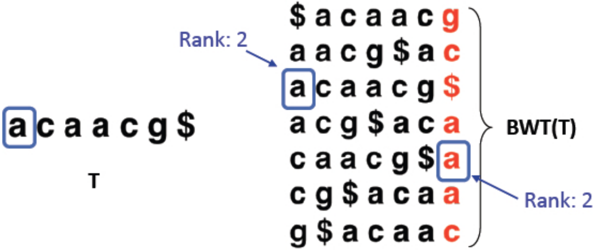


Figure: in red the analogy with the suffix array (from Wall lab in Harvard)

Burrows-Wheeler Transform in alignment: example

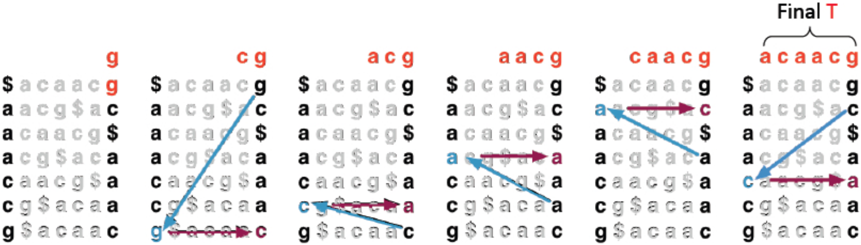
Property that makes BWT(T) reversible is LF Mapping
ith occurrence of a character in **L**ast column is same text occurrence as the ith occurrence in the **F**irst column



```
$ a c a a c g
a a c g $ a c
a c a a c g $
a c g $ a c a
c a a c g $ a
c g $ a c a a
g $ a c a a c
```

Burrows-Wheeler Transform in alignment: example

To recreate T from BWT(T), repeatedly apply rule: T = BWT[LF(i)] + T; i = LF(i).
Where LF(i) maps row i to row whose first character corresponds to i”s last per LF Mapping



⋆Topic: Phylogeny (remember the guide tree)

The reconstruction of the history of speciation could be done by comparison of DNA and amino acid sequences. A phylogeny is a tree where the leaves (existing species) are labeled and no internal node (ancestor) has degree 2 except for the root. Phylogenies may be rooted or unrooted. Here we use the terms species and taxa in a synonymous way. A clade is a group of species that includes all descendants of one common ancestor.

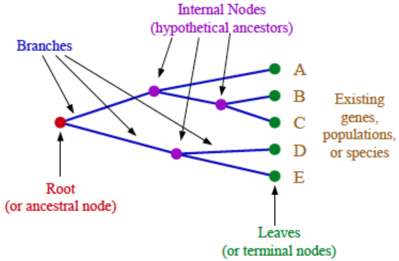
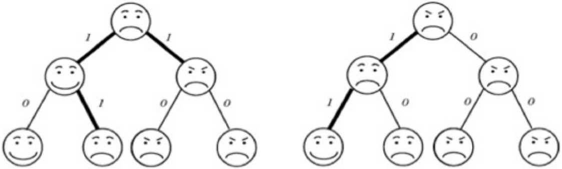


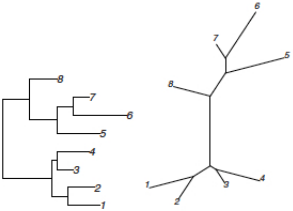
Figure: tree representation: $((a, (b, c)), (d, e))$; trees could also be unrooted

Phylogeny using parsimony

Biological aims: from sequence alignment to phylogeny (a tree) by minimising the number of changes (mutations). Parsimony means economy; there are three main algorithms (Fitch,Wagner,Sankoff); the output trees are rooted (below the difference between rooted, left, and unrooted, right)



(a) *Parsimony Score=3* (b) *Parsimony Score=2*



Fitch parsimony model for DNA characters

Fitch downpass algorithm (on the left the pseudo code, right details)

Bottom-up phase: Determine set of possible states for each internal node; top-down phase: Pick states for each internal node. If the descendant state sets S_q and S_r overlap, then the state set of node p will include the states present in the intersection of S_q and S_r. If the descendant state sets do not overlap, then the state set of p will include all states that are the union of S_q and S_r. States that are absent from both descendants will never be present in the state set of p.

```
1. $S_p \leftarrow S_q \bigcap S_r$
2. if $S_p = 0$ then
3. $S_p \leftarrow S_q \bigcup S_r$
4. $l \leftarrow l + 1$
5. end if
```

Initialization: $R_i = [\ s_i\ ]$; Do a post-order (from leaves to root) traversal of tree Determine R_i of internal node i with children j, k:

$$R_i = \begin{cases} R_j \bigcap R_k & \textit{if } R_j \bigcap R_k \neq 0 \\ R_j \bigcup R_k & \textit{otherwise} \end{cases}$$

Assume that we have the final state set F_a of node a, which is the immediate ancestor of node p (S_p) that has two children q (S_q) and r (S_r).

```
1. $F_p \leftarrow S_p \bigcap F_a$
2. if $F_p \neq F_a$ then
3. if $S_q \bigcap S_r \neq 0$ then
4. $F_p \leftarrow ((S_q \bigcup S_r) \bigcap F_a) \bigcup S_p$
5. else
6. $F_p \leftarrow S_p \bigcup F_a$
7. end if
8. end if
```

$$R_i(s) = \begin{cases} 0 & if\ s_i = s \\ \infty & otherwise \end{cases}$$

$$R_i(s) = min_{s'}\{R_j(s') + S(s', s)\} + min_{s'}\{R_k(s') + S(s', s)\}$$

If the downpass state set of p includes all of the states in the final set of a, then each optimal assignment of final state to a can be combined with the same state at p to give zero changes on the branch between a and p and the minimal number of changes in the subtree rooted at p. If the final set of a includes states that are not present in the downpass set of p, then there is a change on the branch between a and p.

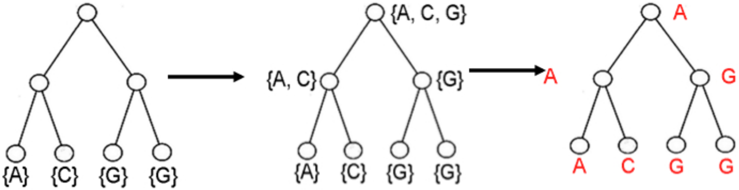


Figure: Fitch

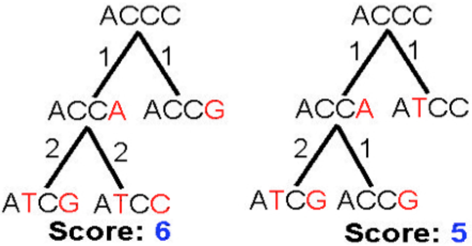


Figure: Parsimony-score = number of union operations

Wagner Algorithm

Wagner downpass algorithm

Assume that the state set of a node p is a set of continuous elements $S = x, x+1, x+2, ..., y$ where $min(S) = x$ and $max(S) = y$ (we can also call this set an interval). Now define the operation $S_i \sqcap S_j$ as producing the set of continuous elements from $max(min(S_i), min(S_j)) to min(max(S_i), max(S_j))$.

```
given a node p and its two daughters q and r
$S_p \leftarrow S_q \bigcap S_r$
if $S_p = 0$ then
    $S_p \leftarrow S_q \sqcap S_r$
    $l \leftarrow l + (\|S_p\| - 1)$
end if
```

If S_i and S_j overlap, then this operation simply produces their intersection, but if they do not overlap, the result is a minimum spanning interval connecting the two sets. For instance, $2, 3, 4 \sqcap 6, 7, 8 = 4, 5, 6$.

Wagner uppass algorithm

```
$F_p \leftarrow S_p \bigcap F_a$
if $F_p \neq F_a$ then
    if $S_q \bigcap S_r \neq 0$ then
        $F_p \leftarrow ((S_q \bigsqcup S_r) \bigcap F_a) \bigcup S_p$
    end if
end if
```

let us define the operation $S_i \bigsqcup S_j$ as producing the set of continuous elements from $min(min(Si), min(Sj))$ to $max(max(Si), max(Sj))$. If the two intervals overlap, the result is simply their union, but if they are disjoint then the operation will produce an interval including all the values from the smallest to the largest. For example, $3, 4 \bigsqcup 6, 7 = 3, 4, 5, 6, 7$.

Sankoff general parsimony or Sankoff optimisation

Sankoff downpass algorithm

```
1. for all i do
2. $h_i^{(q)} \leftarrow min_j(c_{ij} + g_j^{(q)})$
3. $h_i^{(r)} \leftarrow min_j(c_{ij} + g_j^{(r)})$
4. end for
5. for all i do
6. $g_i^{(p)} \leftarrow h_i^{(q)} + h_i^{(r)}$
7. end for
```

Sankoff parsimony is based on a cost matrix $C = c_{ij}$, the elements of which define the cost c_{ij} of moving from a state i to a state j along any branch in the tree. The cost matrix is used to find the minimum cost of a tree and the set of optimal states at the interior nodes of the tree.

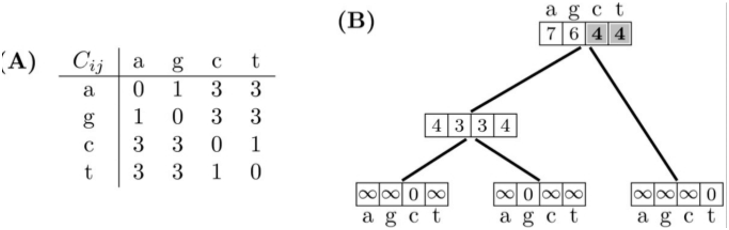
Sankoff finding optimal states and uppass algorithm

```
$F_p \leftarrow 0$
for all i in $F_a$ do
    $m \leftarrow c_{i1} + g_1^{(p)}$
    for all $j \neq 1$ do
        $m \leftarrow min(c_{ij} + g_j^{(p)}, m)$
    end for
    for all j do
        if $c_{ij} + g_j^{(p)} = m$ then
            $F_p \leftarrow F_p \bigcup j$
        end if
    end for
end for
```

```
for all j do
    $f_j^{(p)} \leftarrow min_i(f_i^{(a)} - h_i^{(p)} + c_{ij})$
end for
```

Complexity: if we want to calculate the overall length (cost) of a tree with m taxa, n characters, and k states, it is relatively easy to see that the Fitch and Wagner algorithms have complexity $O(mnk)$ and the Sankoff algorithm is of complexity $O(mnk^2)$.

Sankoff: example of downpass



(A)

C_{ij}	a	g	c	t
a	0	1	3	3
g	1	0	3	3
c	3	3	0	1
t	3	3	1	0

Figure: If the leaf has the character in question, the score is 0; else, score is ∞ Each mutation $a->b$ costs the same in Fitch and Wagner and differently in Sankoff parsimony algorithm (weighted matrix in A). An example of a weighted matrix for Sankoff (for proteins) is the Blosum, presented before in this course

example of uppass

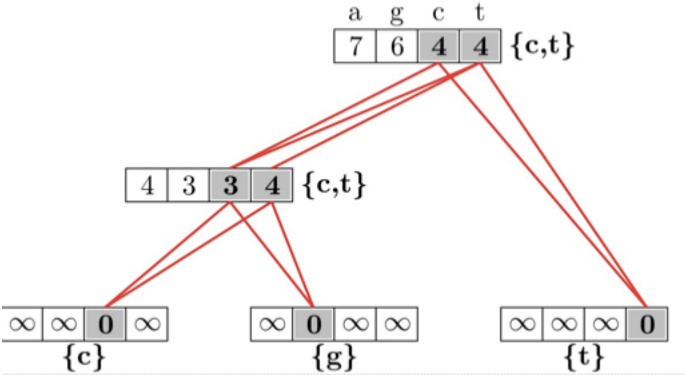
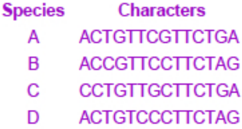
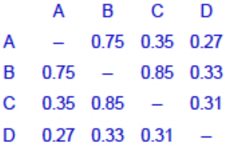


Figure: Example of Sankoff algorithm

Distance methods use a distance (dissimilarity matrix= 1 - similarity) matrix to construct a tree and are kin to clustering methods. We can use the same matrix we use for Blast search, for example the Blosum matrix. The UPGMA outputs a rooted tree while the neighbour joining outputs an unrooted tree.



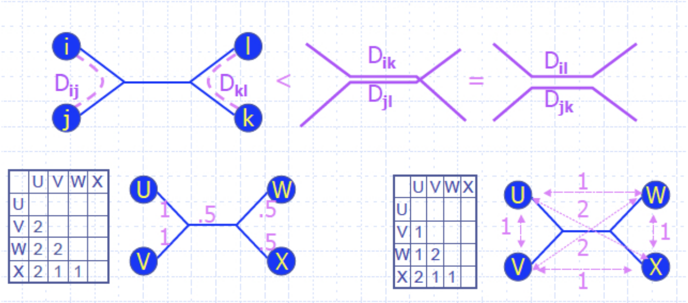
Species	Characters
A	ACTGTTCGTTCTGA
B	ACCGTTCCTTCTAG
C	CCTGTTGCTTCTGA
D	ACTGTCCCTTCTAG



	A	B	C	D
A	–	0.75	0.35	0.27
B	0.75	–	0.85	0.33
C	0.35	0.85	–	0.31
D	0.27	0.33	0.31	–

Additivity: when a distance matrix turns into a tree

A matrix D is additive if and only if : for every four indices i,j,k,l the maximum and median of the three pairwise sums are identical: $D_{ij} + D_{kl} \leq D_{ik} + D_{jl} = D_{il} + D_{jk}$ Suggests how to connect 4 points into a tree to fit D

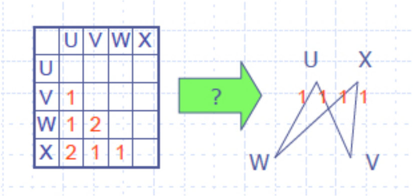


	U	V	W	X
U				
V	2			
W	2	2		
X	2	1	1	

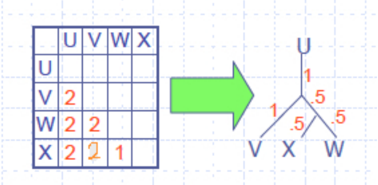
	U	V	W	X
U				
V	1			
W	1	2		
X	2	1	1	

The additivity property

Top: distance matrix does not turn into a tree; Bottom: the distance matrix turns into a tree.



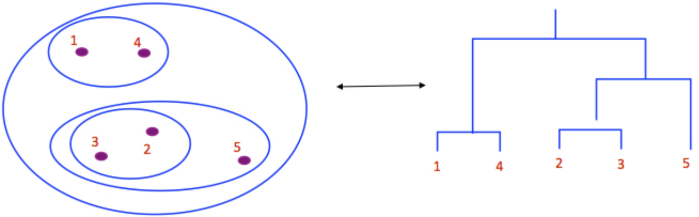
	U	V	W	X
U				
V	1			
W	1	2		
X	2	1	1	



	U	V	W	X
U				
V	2			
W	2	2		
X	2	2	1	

UPGMA: Unweighted Pair Group Method with Arithmetic Mean

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. The algorithm produces an ultrametric tree : the distance from the root to any leaf is the same (this corresponds to a constant molecular clock: the same proportion of mutations in any pathway root to leaf). Input is a distance matrix of distances between species; the iteration combines the two closest species until we reach a single cluster.



UPGMA is also hierarchical clustering

1. Initialization: Assign each species to its own cluster C_i
2. Each such cluster is a tree leaf
3. Iteration:
4. Determine i and j so that $d(C_i, C_j)$ is minimal
5. Define a new cluster $C_k = C_i \bigcup C_j$ with a corresponding node at height $d(C_i, C_j)/2$
6. Update distances to C_k using weighted average
7. Remove C_i and C_j
8. Termination: stop when just a single cluster remains

Neighbor Joining, NJ

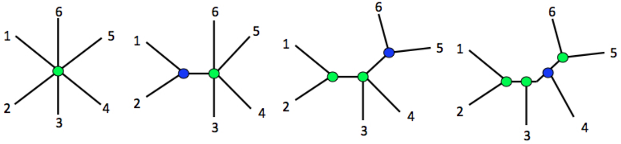
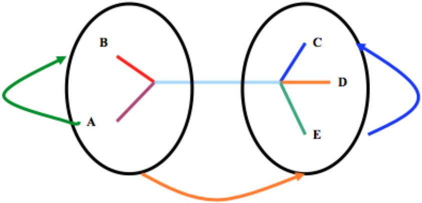


Figure: NJ starts with a star topology (i.e. no neighbors have been joined) and then uses the smallest distance in the distance matrix to find the next two pairs to move out of the multifurcation then recalculate the distance matrix that now contains a tip less.

$$S_{mn} = \frac{\sum d_{im} + d_{in}}{2(N-2)} + \frac{d_{mn}}{2} - \frac{\sum d_{ij}}{N-2}$$

Figure: The sequences are chosen to give best least-squares estimate of branch length (joining m and n; i are the other nodes)

1. Identify i,j as neighbour if their distance is the shortest.
2. Combine i,j into a new node u.
3. Update the distance matrix.
4. Distance of u from the rest of the tree is calculated
5. If only 3 nodes are left finish.



Neighbor Joining

1. If N represents the number of leaves at each stage, we compute S_{12}, S_{13}, S_{14},, $S_{(N-1,N)}$, which is about N^2 computations.
2. We have N stages (we start off with a matrix of N x N, and at each stage the matrix is reduced by 1), therefore, N x $N^2 = N^3$.
3. Each S_{ij} we compute, requires us to sum over all of the elements in the matrix once again, N^2 computations, so we've reached a complexity of N x N^2 X $N^2 = N^5$.

in the next slide we will operate so that Stage 1 and 2 remain with the same complexity $O(N^3)$, while Stage 3 is reduced to $O(1)$, and thus the complexity is $O(N^3)$

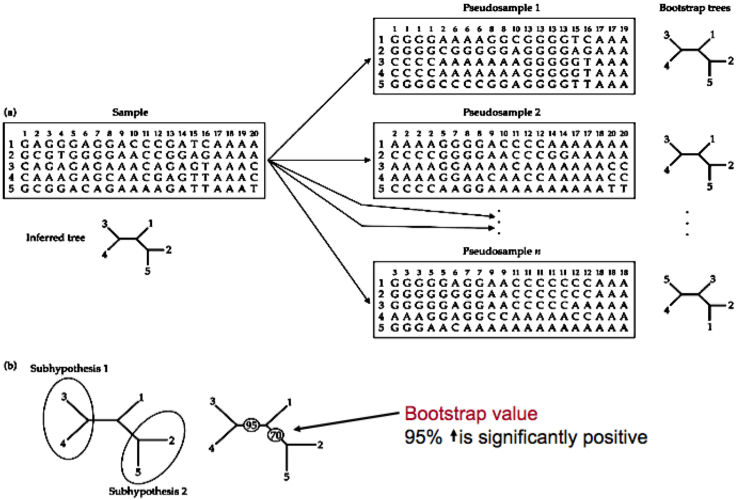
Neighbor Joining with complexity of $O(N^3)$

1. Give a matrix of pairwise distances (d_{ij}), for each terminal node i calculate its net divergence r_i from all the other species using the formula $r_i = \sum_{k=1}^{N} d_{ji}$ where N is the number of terminal nodes in the current matrix.
2. Create a rate corrected distance matrix M in which the elements are defined as $M_{ij} = d_{ij} - (r_i - r_j) / (N - 2)$ only states $i \neq j$ are interesting, even only the minimum needs to be known.
3. define a new node u whose three branches join nodes i, j and the rest of the tree.
4. Define the length of the tree branches from u to i to j as $v_{iu} = \frac{\frac{d_{ij}}{2} + (r_i - r_j)}{2(N-2)}$ and $v_{ju} = d_{ij} - v_{iu}$
5. Define the distance from u to each other terminal node $d_{ku} = (d_{ik} + d_{jk} + d_{ij}) / 2$
6. Remove distance to nodes i and j from the data matrix and decrease N by 1.
7. If more than two nodes remaining, go back to step 1. Otherwise the tree is full defined except for the last branch

The bootstrap algorithm

If there are m sequences, each with n nucleotides, a phylogenetic tree can be reconstructed using some tree building methods.

1. From each sequence, n nucleotides are randomly chosen with replacements, giving rise to m rows of n columns each. These now constitute a new set of sequences.
2. A tree is then reconstructed with these new sequences using the same tree building method as before.
3. Next the topology of this tree is compared to that of the original tree. Each interior branch of the original tree that is different from the bootstrap tree is given a score of 0; all other interior branches are given the value 1.
4. This procedure of resampling the sites and tree reconstruction is repeated several hundred times, and the percentage of times each interior branch is given a value of 1 is noted. This is known as the bootstrap value. As a general rule, if the bootstrap value for a given interior branch is 95% or higher, then the topology at that branch is considered "correct".



Sample

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	G	A	G	G	G	A	G	G	A	C	C	C	G	A	T	C	A	A	A	A
2	G	C	G	T	G	G	G	G	A	A	C	C	G	G	A	G	A	A	A	A
3	C	A	G	A	G	A	G	A	A	A	C	A	G	A	G	T	A	A	A	C
4	C	A	A	A	G	A	G	C	A	A	C	G	A	G	T	T	A	A	A	C
5	G	C	G	G	A	C	A	G	A	A	A	A	G	A	T	T	A	A	A	T

Pseudosample 1

	1	1	1	1	2	6	6	6	8	8	10	13	13	13	13	15	16	17	17	19
1	G	G	G	G	A	A	A	A	G	G	C	G	G	G	G	T	C	A	A	A
2	G	G	G	G	C	G	G	G	G	G	A	G	G	G	G	A	G	A	A	A
3	C	C	C	C	A	A	A	A	A	A	A	G	G	G	G	G	T	A	A	A
4	C	C	C	C	A	A	A	A	A	A	A	G	G	G	G	G	T	A	A	A
5	G	G	G	G	C	C	C	C	G	G	A	G	G	G	G	T	T	A	A	A

Pseudosample 2

	2	2	2	2	5	7	8	8	9	10	11	12	12	14	14	17	17	18	20	20
1	A	A	A	A	G	G	G	G	A	C	C	C	C	A	A	A	A	A	A	A
2	C	C	C	C	G	G	G	G	A	A	C	C	C	G	G	A	A	A	A	A
3	A	A	A	A	G	G	A	A	A	A	C	A	A	A	A	A	A	A	C	C
4	A	A	A	A	G	G	A	A	C	A	A	C	C	A	A	A	A	A	C	C
5	C	C	C	C	A	A	G	G	A	A	A	A	A	A	A	A	A	A	T	T

Pseudosample *n*

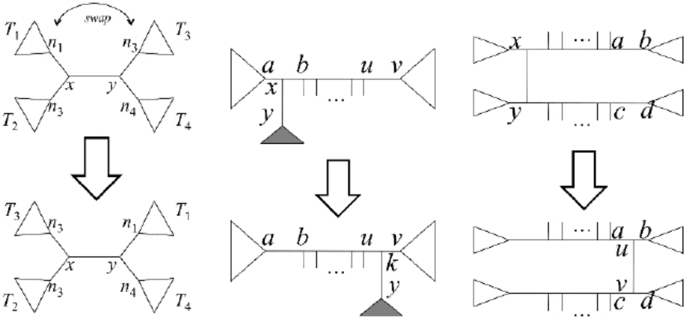
	3	3	3	5	5	6	7	7	9	9	11	11	11	11	11	12	12	18	18	18
1	G	G	G	G	G	A	G	G	A	A	C	C	C	C	C	C	C	A	A	A
2	G	G	G	G	G	G	G	G	A	A	C	C	C	C	C	C	C	A	A	A
3	G	G	G	G	G	A	G	G	A	A	C	C	C	C	C	A	A	A	A	A
4	A	A	A	G	G	A	G	G	C	C	A	A	A	A	A	C	C	A	A	A
5	G	G	G	A	A	C	A	A	A	A	A	A	A	A	A	A	A	A	A	A

Bootstrap value

95% ↑is significantly positive

Topology rearrangement: three methods are often employed

Tree-topology changing operations: (left) nearest neighbor interchange (NNI), (middle) subtree pruning regrafting (SPR), (right) tree bisection reconnection (TBR). NNI is a special case of SPR, which in turn is a special case of TBR. Let n be the number of taxa in the phylogeny; the number of distinct NNI, SPR, and TBR operations are $O(n)$, $O(n^2)$, and $O(n^3)$



Algorithms for NNI,SPR, TBR

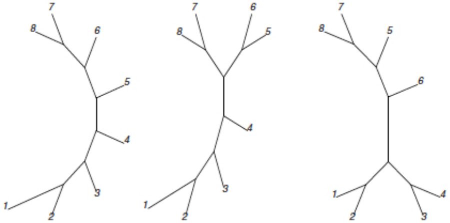
NNI first picks an internal edge (x,y). Let the other two nodes adjacent to them be n_1, n_2, and, n_3,n_4. Pick one of n_1 or n_2, and pick one of n_3 or n_4; say n_1 and n_3 are picked. Remove edges (x, n_1), (y, n_3) from the phylogeny, and add edges (x, n_3) and (y, n_1). In other words, we obtain the new phylogeny by swapping the two clade rooted at n_1 and n_3.

SPR picks two edges (x,y), and (u,v). The edge (u,v) is bisected to create edges (u,w) and (w,v). Pick one of the end points for edge (x,y), say x. The edge (x,y) is first removed from the phylogeny, and the edge (w,y) added to the phylogeny. This makes x a degree-2 node, which has to be suppressed: let the two nodes adjacent to x be a,b; remove edges (x,a) and (x,b), remove node x, then add edge (a,b). This operation detaches the clade rooted at y and reattaches it to the edge (u,v).

TBR removes an edge (x,y), then suppresses the two degree-2 nodes x and y. This creates two disconnected subtrees; choose one edge from each of the two trees. Bisect the two edges by adding nodes u and v, and add edge (u,v) to reconnect the two subtrees.

Tree consensus: three methods often used

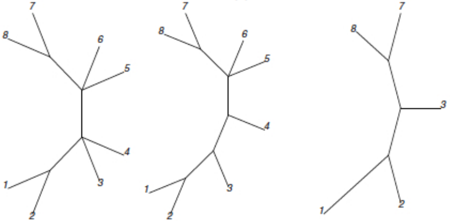
The strict consensus of an input set of phylogenies is the phylogeny such that its every bipartition is in every input phylogeny. Algorithm: pick any input phylogeny, mark every edge whose bipartition is missing in at least one input phylogeny, and contract it by joining its two endpoints together. A relaxation of the strict consensus tree is the p-consensus: it is the phylogeny whose bipartitions are in proportion $\geq pN$ of all the N input phylogenies. It can be shown that if $p > 0.5$ then such a phylogeny exists and is unique, but this is not necessarily so when $p \leq 0.5$. Strict consensus is simply the case where $p = 1$. When we require every bipartition in the consensus to be in $> N/2$ input trees (i.e., $p = (N/2 + 1)/N$ for even N and $p = 1/2$ for odd N), this is called the majority consensus. The third type of consensus tree is the maximum agreement subtree: the goal is to find a largest subset of input taxa such that the input phylogenies all have the same topology when we restrict them to this subset.



(a1) (a2) (a3)

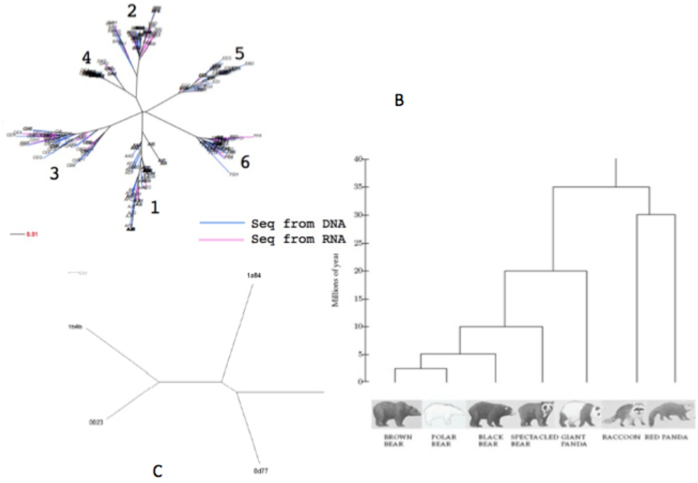
Bipartition		a1	a2	a3
12	345678	*	*	*
123	45678	*	*	
1234	5678	*	*	*
12345	678	*		
123456	78	*	*	*
123478	56		*	
125678	34			*
12346	578			*

(b)

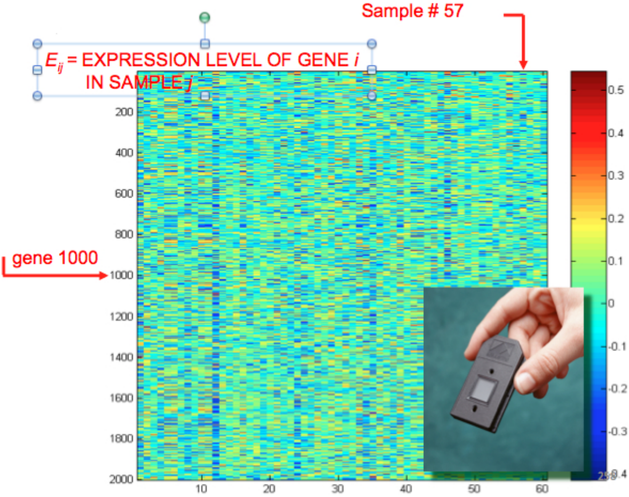


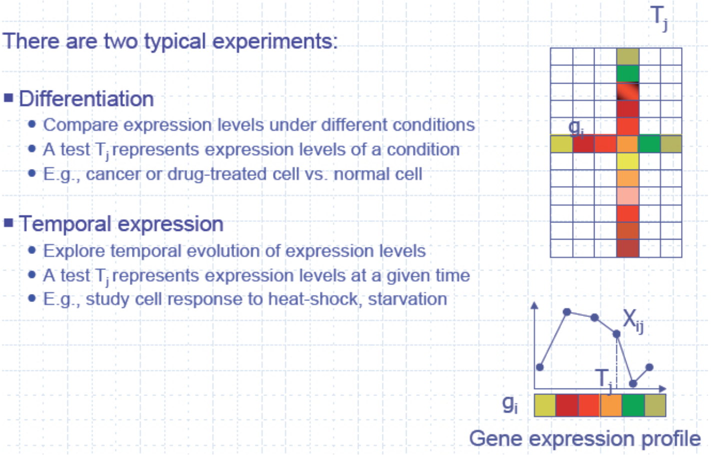
(c) strict consensus (d) majority consensus (e) maximum agreement subtree

Examples of trees: a) hiv virus sampled at different times from 6 patients (1-6); b) phylogeny of bears and panda; c) phylogeny of computer viruses (the FakeAV-DO function f1 was first coded into an alphabet and aligned using Clustal).



⋆Topic: Clustering gene expression data





There are two typical experiments:

- Differentiation
  - Compare expression levels under different conditions
  - A test T_j represents expression levels of a condition
  - E.g., cancer or drug-treated cell vs. normal cell
- Temporal expression
  - Explore temporal evolution of expression levels
  - A test T_j represents expression levels at a given time
  - E.g., study cell response to heat-shock, starvation

Figure: The color of the spot indicates activation with respect to control (red) or repression with respect to the control (green) or absence of regulation (yellow) of a gene, or error in the technological process (black). The genes can be all the genes of an organism (example the 6000 genes of yeast), or a selection of genes of interest (+ control genes).

Aims: clustering gene expression: visualising and analyzing vast amounts of biological data as a whole set can be difficult. It is easier to interpret the data if they are partitioned into clusters combining similar data points.

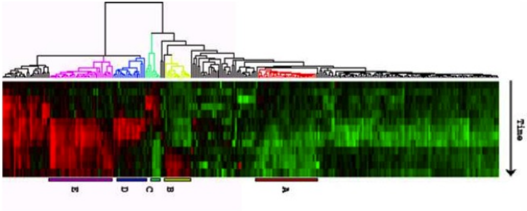


Figure: Hierarchical clustering (UPGMA) could be used to investigate whether the genes belonging to the same cluster share a common function or are co-regulated by a common protein which binds before the gene acting as repressor or activator of the gene function. The clusters are coloured differently in the hierarchical clustering added to the microarray

Clustering for Gene expression data

Microarrays measure the activity (expression level) of the genes under varying conditions/time points. Expression level is estimated by measuring the amount of RNA for that particular gene. A gene is active if it is being transcribed. More mRNA usually indicates more gene activity. Microarray data are usually transformed into an intensity matrix. The analysis allows scientists to make correlations between different genes (even if they are dissimilar) and to understand how genes functions might be related. Plot each datum as a point in N-dimensional space; Make a distance matrix for the distance between every two gene points in the N-dimensional space; Genes with a small distance share the same expression characteristics and might be functionally related or similar. Clustering reveal groups of functionally related genes.

K-Means Clustering: Lloyd Algorithm

1. Arbitrarily assign the k cluster centers
2. while the cluster centers keep changing
3. Assign each data point to the cluster Ci corresponding to the closest cluster representative (center) $(1 \leq i \leq k)$
4. After the assignment of all data points, compute new cluster representatives according to the center of gravity of each cluster, that is, the new cluster representative is $\sum v \setminus |C|$ for all v in C for every cluster C

Progressive greedy K-means Algorithm

```
Select an arbitrary partition P into k clusters
while forever
    bestChange $\leftarrow$ 0
    for every cluster C
        for every element i not in C
            if moving i to cluster C reduces its clustering cost
                if cost(P) $-$ cost($P_{i \rightarrow C}$) $>$ bestChange
                    bestChange $\leftarrow$ cost(P) $-$ cost($P_{i \rightarrow C}$)
                    $i^{'}$ $\leftarrow$ i
                    $C^{'}$ $\leftarrow$ C
    if bestChange $>$ 0
        Change partition P by moving $i^{'}$ to $C^{'}$
    else
        return P
```

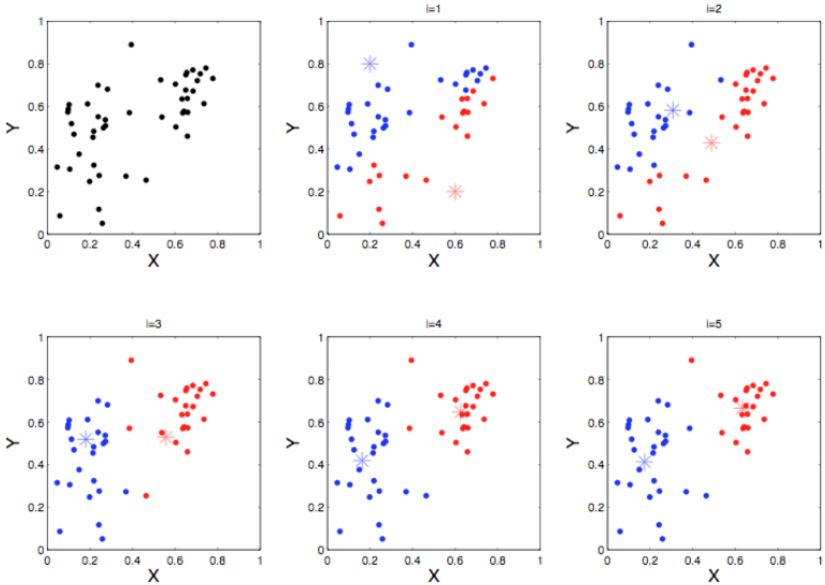
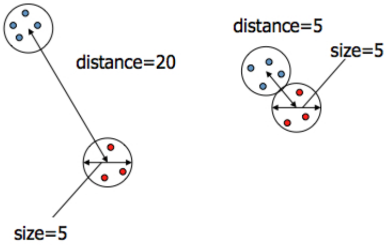


Figure: K-means progression from left to right and top to bottom

The quality of cluster could be assessed by ratio of distance to nearest cluster and cluster diameter. A cluster can be formed even when there is no similarity between clustered patterns. This occurs because the algorithm forces k clusters to be created. Linear relationship with the number of data points; Complexity is $O(nKI)$ where n = number of points, K = number of clusters, I = number of iterations.



Markov Clustering algorithm, MCL

We take a random walk on the graph described by the similarity matrix, but after each step we weaken the links between distant nodes and strengthen the links between nearby nodes.
Unlike most clustering algorithms, the MCL does not require the number of expected clusters to be specified beforehand. The basic idea underlying the algorithm is that dense clusters correspond to regions with a larger number of paths.
A random walk has a higher probability to stay inside the cluster than to leave it soon. The crucial point lies in boosting this effect by an iterative alternation of expansion and inflation steps.
The inflation parameter is responsible for both strengthening and weakening of current. (Strengthens strong currents, and weakens already weak currents). The expansion parameter, r, controls the extent of this strengthening / weakening (In the end, this influences the granularity of clusters).

MCL Algorithm

1. Input is an un-directed graph, power parameter e (usually =2), and inflation parameter r (usually =2).
2. Create the associated matrix
3. Normalize the matrix; $M'_{pq} = \frac{M_{pq}}{\sum_i M_{iq}}$
4. Expand by taking the e-th power of the matrix; for example, if $e = 2$ just multiply the matrix by itself.
5. Inflate by taking inflation of the resulting matrix with parameter r : $M_{pq} = \frac{(M_{pq})^r}{\sum_i (M_{iq})^r}$
6. Repeat steps 4 and 5 until a steady state is reached (convergence).

MCL Algorithm complexity and entropy analysis

The number of steps to converge is not proven, but experimentally shown to be 10 to 100 steps, and mostly consist of sparse matrices after the first few steps. There are several distinct measures informing on the clustering and its stability such as the following clustering entropy:
$S = -1/L \sum_{ij}(P_{ij} log_2 P_{ij} + (1 - P_{ij}) log_2(1 - P_{ij}))$ where the sum is over all edges and the entropy is normalized by the total number of edges. This might be used to detect the best clustering obtained after a long series of clusterings with different granularity parameters each time.
The expansion step of MCL has time complexity $O(n^3)$. The inflation has complexity $O(n^2)$. However, the matrices are generally very sparse, or at least the vast majority of the entries are near zero. Pruning in MCL involves setting near-zero matrix entries to zero, and can allow sparse matrix operations to improve the speed of the algorithm vastly.

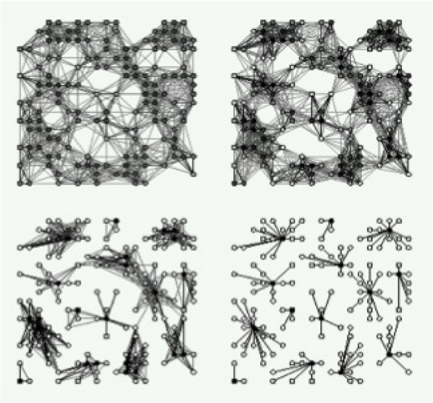
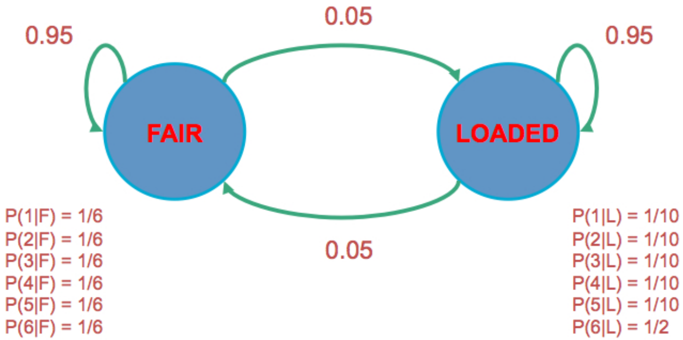


Figure: progression from left to right and top to bottom

⋆Topic: Hidden Markov Models in Bioinformatics

HMMs form a useful class of probabilistic graphical models used to find genes, predict protein structure and classify protein families.
Definition: A hidden Markov model (HMM) has an Alphabet = $b_1, b_2, , b_M$, set of states Q = 1, ..., K , and transition probabilities between any two states
a_{ij} = transition prob from state i to state j
$a_{i1} + + a_{iK} = 1$, for all states i = 1,K
Start probabilities a_{0i}
$a_{01} + + a_{0K} = 1$
Emission probabilities within each state $e_i(b) = P(x_i = b|\pi_i = k)$
$e_i(b1) + + e_i(bM) = 1$, for all states i = 1,K
A Hidden Markov model is Memoryless: $P(\pi_{t+1} = k|$ whatever happened so far) $= P(\pi_{t+1} = k|\pi_1, \pi_2, , \pi_t, x_1, x_2, , x_t) = P(\pi_{t+1} = k|\pi_t)$ at each time step t, only matters the current state π_t

The dishonest casino model

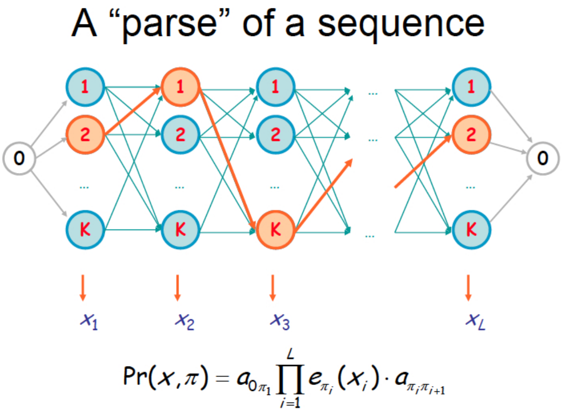


P(1|F) = 1/6
P(2|F) = 1/6
P(3|F) = 1/6
P(4|F) = 1/6
P(5|F) = 1/6
P(6|F) = 1/6

P(1|L) = 1/10
P(2|L) = 1/10
P(3|L) = 1/10
P(4|L) = 1/10
P(5|L) = 1/10
P(6|L) = 1/2

The dishonest casino

- Known:
- The structure of the model
- The transition probabilities
- Hidden: What the casino did (ex FFFFFLLLLLLLFFFF)
- Observable: The series of die tosses, es 3415256664666153...
- What we must infer:
- When was a fair die used?
- When was a loaded one used?



A “parse” of a sequence

$$\Pr(x,\pi) = a_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i) \cdot a_{\pi_i \pi_{i+1}}$$

Given a sequence x = $x_1 x_N$, A parse of x is a sequence of states π = π_1, , π_N

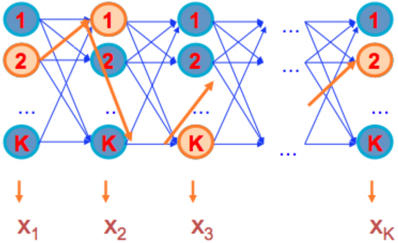
Likelihood of a parse

Given a sequence $x = x_1 \ldots\ldots x_N$

and a parse $\pi = \pi_1, \ldots\ldots, \pi_N$,

To find how likely is the parse:

(given our HMM)



$P(x, \pi) = P(x_1, \ldots, x_N, \pi_1, \ldots\ldots, \pi_N) =$

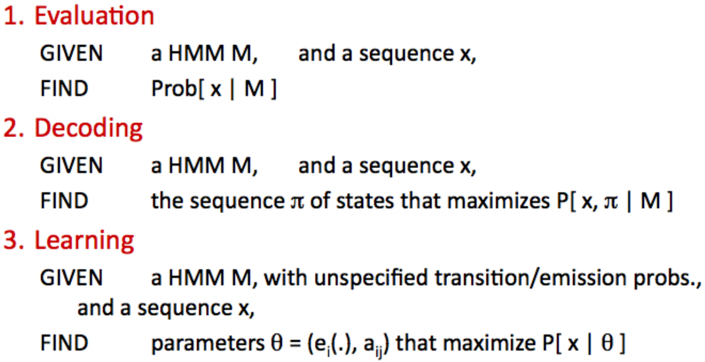
$P(x_N, \pi_N \mid \pi_{N-1})\, P(x_{N-1}, \pi_{N-1} \mid \pi_{N-2}) \ldots\ldots P(x_2, \pi_2 \mid \pi_1)\, P(x_1, \pi_1) =$

$P(x_N \mid \pi_N)\, P(\pi_N \mid \pi_{N-1}) \ldots\ldots P(x_2 \mid \pi_2)\, P(\pi_2 \mid \pi_1)\, P(x_1 \mid \pi_1)\, P(\pi_1)$

$=$

$a_{0\pi 1}\, a_{\pi 1 \pi 2} \ldots\ldots a_{\pi N-1 \pi N}\, e_{\pi 1}(x_1) \ldots\ldots e_{\pi N}(x_N)$

The three main questions on HMMs



1. Evaluation

GIVEN a HMM M, and a sequence x,

FIND Prob[x | M]

2. Decoding

GIVEN a HMM M, and a sequence x,

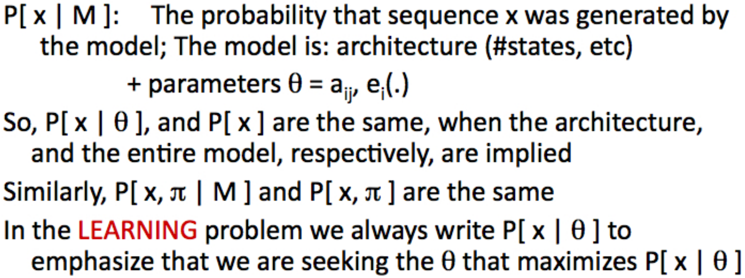
FIND the sequence π of states that maximizes P[x, π | M]

3. Learning

GIVEN a HMM M, with unspecified transition/emission probs., and a sequence x,

FIND parameters $\theta = (e_i(.), a_{ij})$ that maximize P[x | θ]

Lets not be confused by notation



P[x | M]: The probability that sequence x was generated by the model; The model is: architecture (#states, etc)

+ parameters $\theta = a_{ij}, e_i(.)$

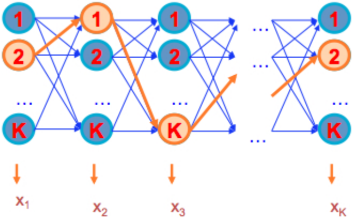
So, P[x | θ], and P[x] are the same, when the architecture, and the entire model, respectively, are implied

Similarly, P[x, π | M] and P[x, π] are the same

In the LEARNING problem we always write P[x | θ] to emphasize that we are seeking the θ that maximizes P[x | θ]

GIVEN $x = x_1x_2......x_N$

We want to find $\pi = \pi_1, ..., \pi_N$, such that P[x, π] is maximized

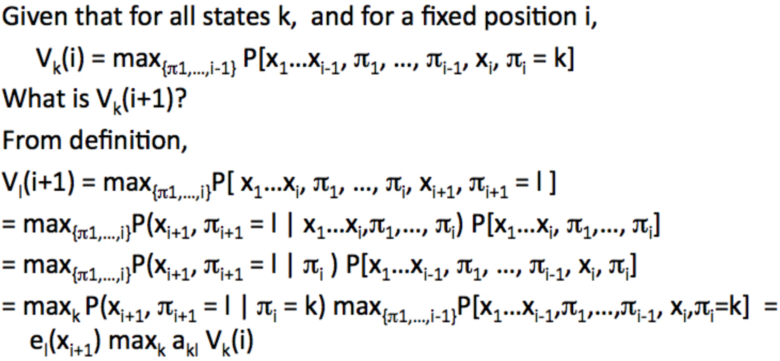


$\pi^* = \text{argmax}_\pi \text{ P[} x, \pi \text{]}$

We can use dynamic programming!

Let $V_k(i) = \max_{\{\pi 1,...,i-1\}} P[x_1...x_{i-1}, \pi_1, ..., \pi_{i-1}, x_i, \pi_i = k]$
= Probability of most likely sequence of states ending at state $\pi_i = k$

Decoding main idea



Given that for all states k, and for a fixed position i,

$V_k(i) = \max_{\{\pi 1,\dots,i-1\}} P[x_1 \dots x_{i-1}, \pi_1, \dots, \pi_{i-1}, x_i, \pi_i = k]$

What is $V_k(i+1)$?

From definition,

$V_l(i+1) = \max_{\{\pi 1,\dots,i\}} P[\, x_1 \dots x_i, \pi_1, \dots, \pi_i, x_{i+1}, \pi_{i+1} = l \,]$

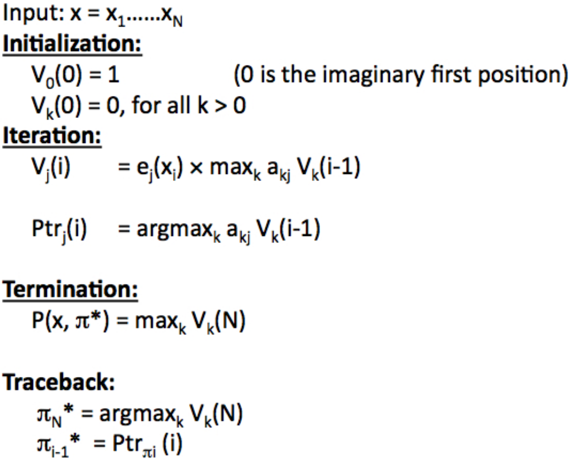
$= \max_{\{\pi 1,\dots,i\}} P(x_{i+1}, \pi_{i+1} = l \mid x_1 \dots x_i, \pi_1, \dots, \pi_i)\, P[x_1 \dots x_i, \pi_1, \dots, \pi_i]$

$= \max_{\{\pi 1,\dots,i\}} P(x_{i+1}, \pi_{i+1} = l \mid \pi_i)\, P[x_1 \dots x_{i-1}, \pi_1, \dots, \pi_{i-1}, x_i, \pi_i]$

$= \max_k P(x_{i+1}, \pi_{i+1} = l \mid \pi_i = k) \max_{\{\pi 1,\dots,i-1\}} P[x_1 \dots x_{i-1}, \pi_1, \dots, \pi_{i-1}, x_i, \pi_i = k] =$

$e_l(x_{i+1}) \max_k a_{kl} V_k(i)$

The Viterbi Algorithm



Input: $x = x_1 \ldots\ldots x_N$

Initialization:

$V_0(0) = 1$ (0 is the imaginary first position)

$V_k(0) = 0$, for all $k > 0$

Iteration:

$V_j(i) = e_j(x_i) \times \max_k a_{kj} V_k(i-1)$

$Ptr_j(i) = \text{argmax}_k\, a_{kj} V_k(i-1)$

Termination:

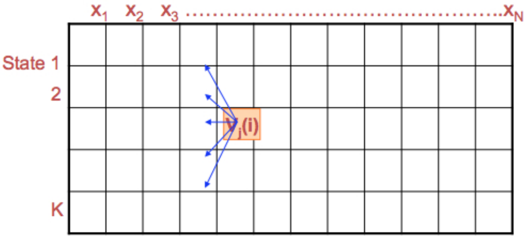
$P(x, \pi^*) = \max_k V_k(N)$

Traceback:

$\pi_N^* = \text{argmax}_k\, V_k(N)$

$\pi_{i-1}^* = Ptr_{\pi i}\,(i)$

The Viterbi Algorithm



Similar to "aligning" a set of states to a sequence

Time:

$O(K^2N)$

Space:

$O(KN)$

Generating a sequence by the model

Given a HMM, we can generate a sequence of length n as follows:

1. Start at state π_1 according to prob $a_{0\pi_1}$
2. Emit letter x_1 according to prob $e_{\pi_1}(x_1)$
3. Go to state π_2 according to prob $a_{\pi_1\pi_2}$
4. until emitting x_n

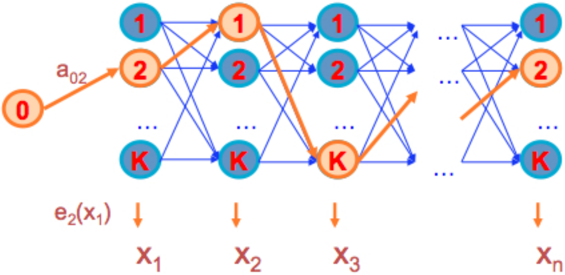
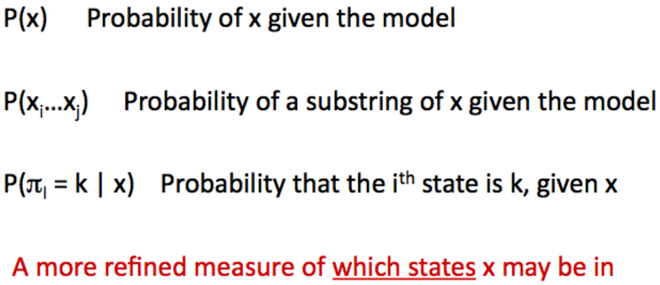


Figure:

Evaluation



$P(x)$ Probability of x given the model

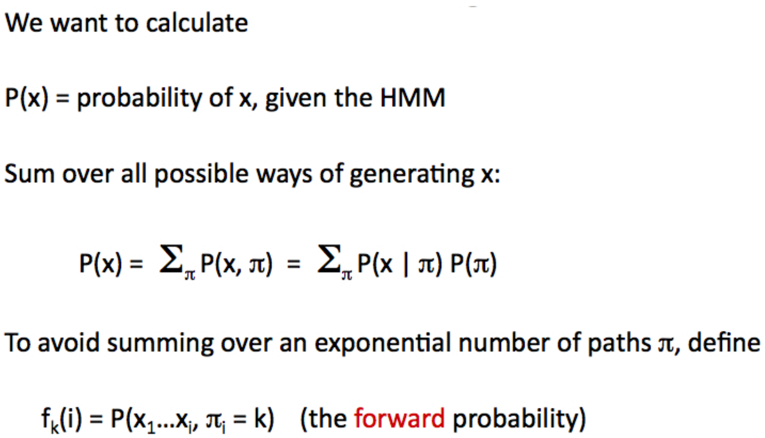
$P(x_i \ldots x_j)$ Probability of a substring of x given the model

$P(\pi_l = k \mid x)$ Probability that the i^{th} state is k, given x

A more refined measure of which states x may be in

The Forward Algorithm

We will develop algorithms that allow us to compute:



We want to calculate

P(x) = probability of x, given the HMM

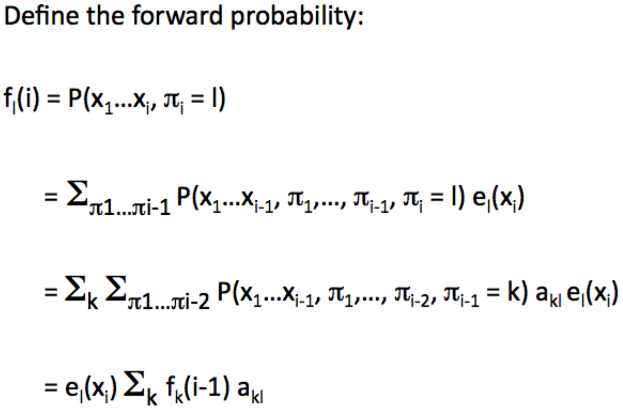
Sum over all possible ways of generating x:

$$P(x) = \Sigma_{\pi} P(x, \pi) = \Sigma_{\pi} P(x \mid \pi) P(\pi)$$

To avoid summing over an exponential number of paths π, define

$f_k(i) = P(x_1 \ldots x_i, \pi_i = k)$ (the forward probability)

The Forward Algorithm derivation



Define the forward probability:

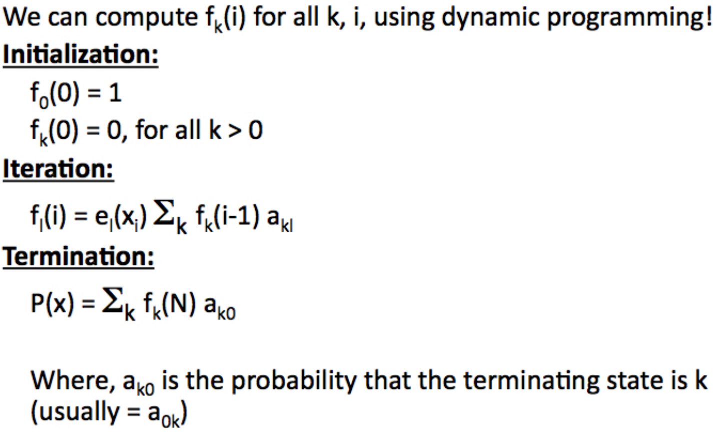
$$f_l(i) = P(x_1 \ldots x_i, \pi_i = l)$$

$$= \Sigma_{\pi 1 \ldots \pi i-1} P(x_1 \ldots x_{i-1}, \pi_1, \ldots, \pi_{i-1}, \pi_i = l)\ e_l(x_i)$$

$$= \Sigma_k \Sigma_{\pi 1 \ldots \pi i-2} P(x_1 \ldots x_{i-1}, \pi_1, \ldots, \pi_{i-2}, \pi_{i-1} = k)\ a_{kl}\ e_l(x_i)$$

$$= e_l(x_i)\ \Sigma_k\ f_k(i-1)\ a_{kl}$$

The Forward Algorithm



We can compute $f_k(i)$ for all k, i, using dynamic programming!

Initialization:

$f_0(0) = 1$

$f_k(0) = 0$, for all $k > 0$

Iteration:

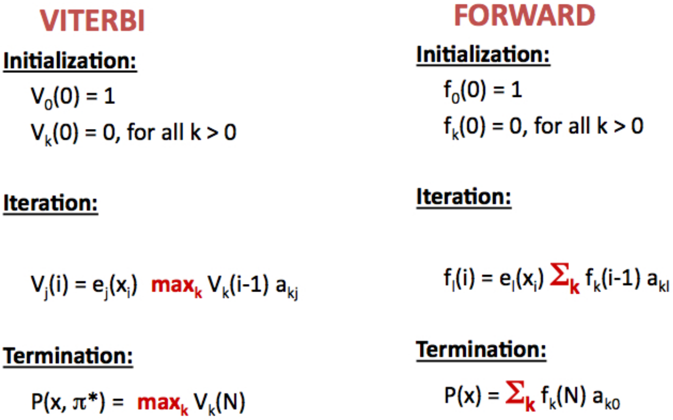
$f_l(i) = e_l(x_i) \sum_k f_k(i-1)\, a_{kl}$

Termination:

$P(x) = \sum_k f_k(N)\, a_{k0}$

Where, a_{k0} is the probability that the terminating state is k
(usually $= a_{0k}$)

Comparison between Viterbi and Forward



VITERBI

Initialization:

$V_0(0) = 1$

$V_k(0) = 0$, for all $k > 0$

Iteration:

$$V_j(i) = e_j(x_i)\ \max_k V_k(i-1)\, a_{kj}$$

Termination:

$$P(x, \pi^*) = \max_k V_k(N)$$

FORWARD

Initialization:

$f_0(0) = 1$

$f_k(0) = 0$, for all $k > 0$

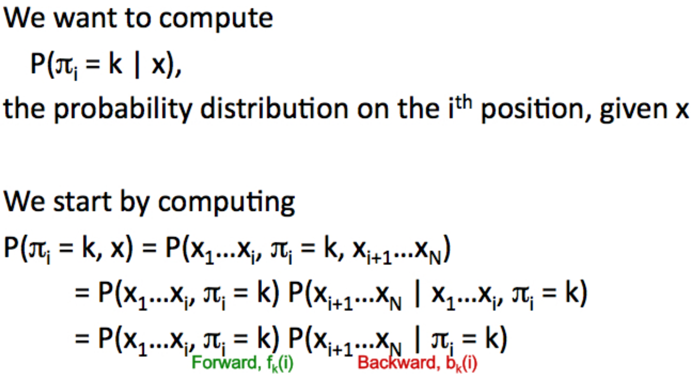
Iteration:

$$f_l(i) = e_l(x_i) \sum_k f_k(i-1)\, a_{kl}$$

Termination:

$$P(x) = \sum_k f_k(N)\, a_{k0}$$

Motivation for the Backward Algorithm



We want to compute

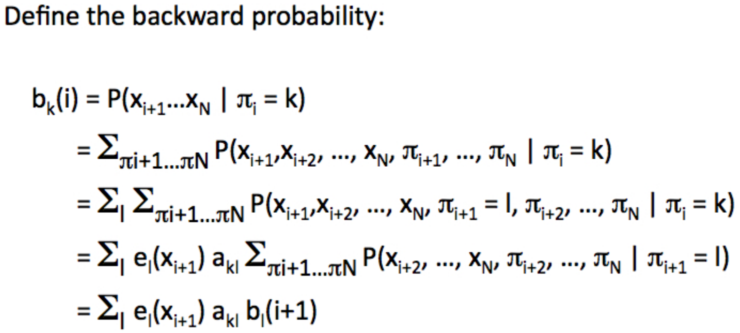
$P(\pi_i = k \mid x)$,

the probability distribution on the ith position, given x

We start by computing

$$
\begin{aligned}
P(\pi_i = k, x) &= P(x_1 \ldots x_i, \pi_i = k, x_{i+1} \ldots x_N) \\
&= P(x_1 \ldots x_i, \pi_i = k)\, P(x_{i+1} \ldots x_N \mid x_1 \ldots x_i, \pi_i = k) \\
&= \underbrace{P(x_1 \ldots x_i, \pi_i = k)}_{\text{Forward, } f_k(i)}\, \underbrace{P(x_{i+1} \ldots x_N \mid \pi_i = k)}_{\text{Backward, } b_k(i)}
\end{aligned}
$$

The Backward Algorithm derivation



Define the backward probability:

$$b_k(i) = P(x_{i+1} \ldots x_N \mid \pi_i = k)$$

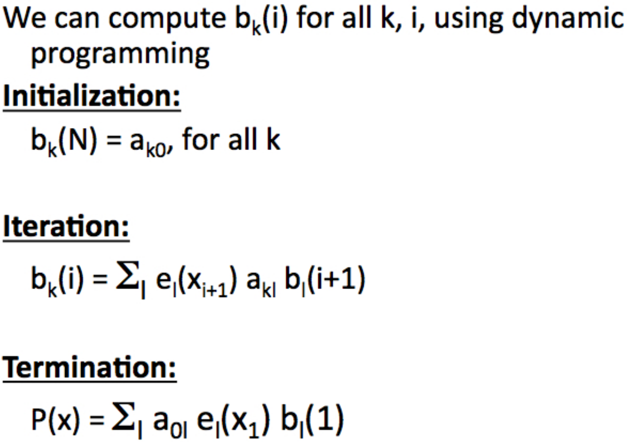
$$= \sum_{\pi i+1 \ldots \pi N} P(x_{i+1}, x_{i+2}, \ldots, x_N, \pi_{i+1}, \ldots, \pi_N \mid \pi_i = k)$$

$$= \sum_l \sum_{\pi i+1 \ldots \pi N} P(x_{i+1}, x_{i+2}, \ldots, x_N, \pi_{i+1} = l, \pi_{i+2}, \ldots, \pi_N \mid \pi_i = k)$$

$$= \sum_l e_l(x_{i+1})\, a_{kl} \sum_{\pi i+1 \ldots \pi N} P(x_{i+2}, \ldots, x_N, \pi_{i+2}, \ldots, \pi_N \mid \pi_{i+1} = l)$$

$$= \sum_l e_l(x_{i+1})\, a_{kl}\, b_l(i+1)$$

The Backward Algorithm



We can compute $b_k(i)$ for all k, i, using dynamic programming

Initialization:

$b_k(N) = a_{k0}$, for all k

Iteration:

$b_k(i) = \Sigma_l \, e_l(x_{i+1}) \, a_{kl} \, b_l(i+1)$

Termination:

$P(x) = \Sigma_l \, a_{0l} \, e_l(x_1) \, b_l(1)$

What is the running time, and space required, for Forward, and Backward?

Time: $O(K^2N)$

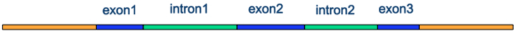
Space: $O(KN)$

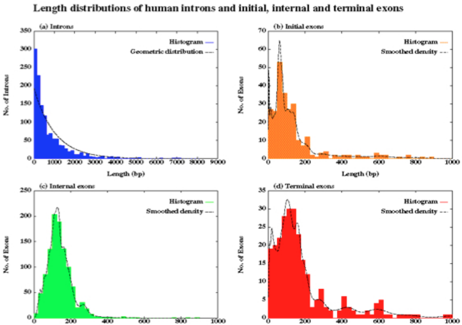
Useful implementation technique to avoid underflows

Viterbi: sum of logs

Forward/Backward: rescaling at each position by multiplying by a constant

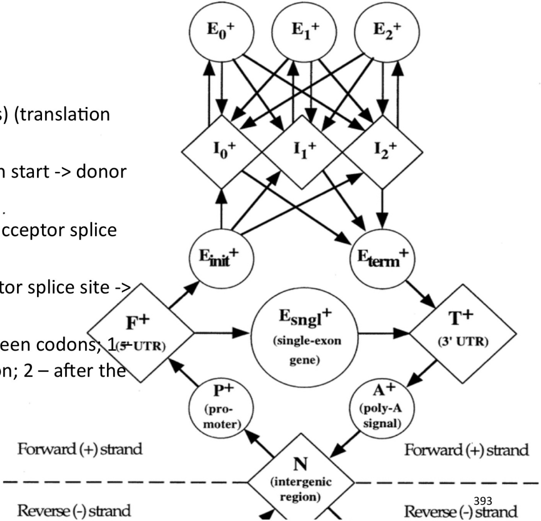
In order to identify genes and their parts (exons and introns) we need to know their length distribution (see example in figures below). Human genes comprise about 3% of the human genome; Average gene length: $\sim 8,000$ DNA base pairs (bp); Average of 5-6 exons/gene; Average exon length: $\sim$ 200 bp; Average intron length: $\sim 2,000$ bp; $\sim$ 8% genes have a single exon Some exons can be as small as 1 or 3 bp. Example HUMFMR1S (http://www.ncbi.nlm.nih.gov/nuccore/1668818) is not atypical: 17 exons 40-60 bp long, comprising 3% of a 67,000 bp gene.



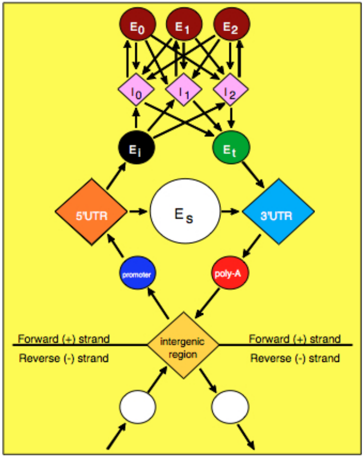


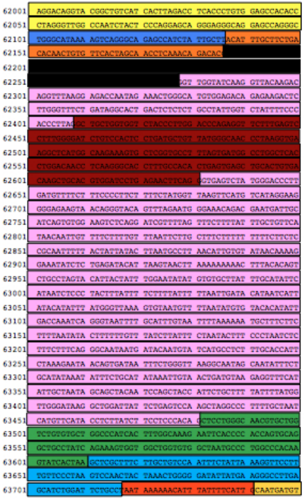
GenScan

- N - intergenic region
- P - promoter
- F - 5’ untranslated region
- E_{sngl} – single exon (intronless) (translation start -> stop codon)
- E_{init} – initial exon (translation start -> donor splice site)
- E_k – phase k internal exon (acceptor splice site -> donor splice site)
- E_{term} – terminal exon (acceptor splice site -> stop codon)
- I_k – phase k intron: 0 – between codons; 1 – after the first base of a codon; 2 – after the second base of a codon



GENSCAN (Burge & Karlin)

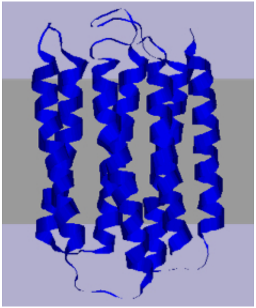




```
62001 AGGACAGGTA CGGCTGTCAT CACTTAGACC TCACCCTGTG GAGCCACACC
62051 CTAGGGTTGG CCAATCTACT CCCAGGAGCA GGGAGGGCAG GAGCCAGGGC
62101 TGGGCATAAA AGTCAGGGCA GAGCCATCTA TTGCTTACAT TTGCTTCTGA
62151 CACAACTGTG TTCACTAGCA ACCTCAAACA GACAC
62201
62251                                    GT TGGTATCAAG GTTACAAGAC
62301 AGGTTTAAGG AGACCAATAG AAACTGGGCA TGTGGAGACA GAGAAGACTC
62351 TTGGGTTTCT GATAGGCACT GACTCTCTCT GCCTATTGGT CTATTTTCCC
62401 ACCCTTAGGC TGCTGGTGGT CTACCCTTGG ACCCAGAGGT TCTTTGAGTC
62451 CTTTGGGGAT CTGTCCACTC CTGATGCTGT TATGGGCAAC CCTAAGGTGA
62501 AGGCTCATGG CAAGAAAGTG CTCGGTGCCT TTAGTGATGG CCTGGCTCAC
62551 CTGGACAACC TCAAGGGCAC CTTTGCCACA CTGAGTGAGC TGCACTGTGA
62601 CAAGCTGCAC GTGGATCCTG AGAACTTCAG GCTGAGTCTA TGGGACCCTT
62651 GATGTTTTCT TTCCCCTTCT TTTCTATGGT TAAGTTCATG TCATAGGAAG
62701 GGGAGAAGTA ACAGGGTACA GTTTAGAATG GGAAACAGAC GAATGATTGC
62751 ATCAGTGTGG AAGTCTCAGG ATCGTTTTAG TTTCTTTTAT TTGCTGTTCA
62801 TAACAATTGT TTTCTTTTGT TTAATTCTTG CTTTCTTTTT TTTTCTTCTC
62851 CGCAATTTTT ACTATTATAC TTAATGCCTT AACATTGTGT ATAACAAAAG
62901 GAAATATCTC TGAGATACAT TAAGTAACTT AAAAAAAAAC TTTACACAGT
62951 CTGCCTAGTA CATTACTATT TGGAATATAT GTGTGCTTAT TTGCATATTC
63001 ATAATCTCCC TACTTTATTT TCTTTTATTT TTAATTGATA CATAATCATT
63051 ATACATATTT ATGGGTTAAA GTGTAATGTT TTAATATGTG TACACATATT
63101 GACCAAATCA GGGTAATTTT GCATTTGTAA TTTTAAAAAA TGCTTTCTTC
63151 TTTTAATATA CTTTTTTGTT TATCTTATTT CTAATACTTT CCCTAATCTC
63201 TTTCTTTCAG GGCAATAATG ATACAATGTA TCATGCCTCT TTGCACCATT
63251 CTAAAGAATA ACAGTGATAA TTTCTGGGTT AAGGCAATAG CAATATTTCT
63301 GCATATAAAT ATTTCTGCAT ATAAATTGTA ACTGATGTAA GAGGTTTCAT
63351 ATTGCTAATA GCAGCTACAA TCCAGCTACC ATTCTGCTTT TATTTTATGG
63401 TTGGGATAAG GCTGGATTAT TCTGAGTCCA AGCTAGGCCC TTTTGCTAAT
63451 CATGTTCATA CCTCTTATCT TCCTCCCACA GCTCCTGGGC AACGTGCTGG
63501 TCTGTGTGCT GGCCCATCAC TTTGGCAAAG AATTCACCCC ACCAGTGCAG
63551 GCTGCCTATC AGAAAGTGGT GGCTGGTGTG GCTAATGCCC TGGCCCACAA
63601 GTATCACTAA GCTCGCTTTC TTGCTGTCCA ATTTCTATTA AAGGTTCCTT
63651 TGTTCCCTAA GTCCAACTAC TAAACTGGGG GATATTATGA AGGGCCTTGA
63701 GCATCTGGAT TCTGCCTAAT AAAAAACATT TATTTTCATT GCAATGATGT
```

Figure: The model (left) and the output (right) of Genscan prediction of a genomic region; the result is a segmentation of a genome sequence, i.e. the colours map the HMM states with the predicted functional genomic segments

Membrane proteins are important for cell import/export. TMHMM is a program to predict membrane protein topology (i.e. which parts are outside, inside and in the membrane) Trained with a database of experimentally determined transmembrane helices Prediction method: Posterior decoding, the program computes the position with respect to the membrane for each amino acid of the sequence. The figure below describes a 7 helix membrane protein forming a sort of a cylinder (3d graph) across the cell membrane.



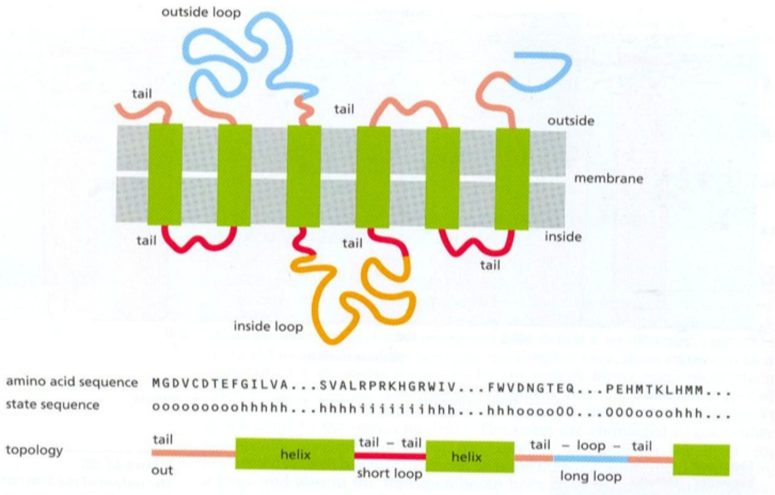

outside loop
tail
tail
outside
membrane
tail
tail
inside
tail
inside loop
amino acid sequence MGDVCDTEFGILVA...SVALRPRKHGRWIV...FWVDNGTEQ...PEHMTKLHMM...
state sequence oooooooooohhhhh...hhhhiiiiiiihhh...hhhoooo00...000ooooohhh...
topology
tail
out
helix
tail – tail
short loop
helix
tail – loop – tail
long loop

Figure: top: the 3D graph previous figure could be represented as a 2D graph; bottom, 3 state prediction: each amino acid could be in the membrane segment (h), outside the cell (o) or inside the cell (i)

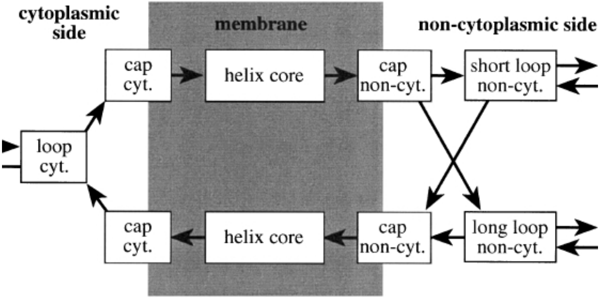
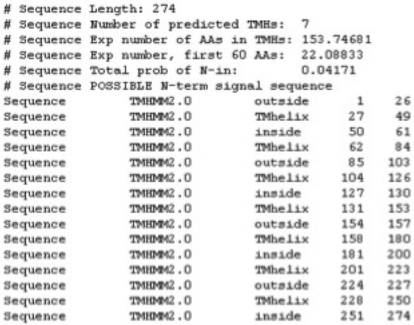
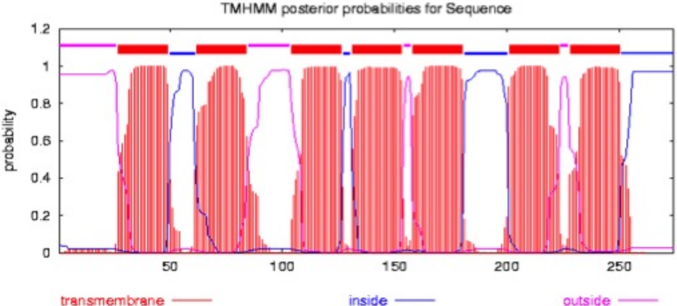


Figure: The THMM model: a three state prediction model (h,o,i) could be then refined adding more states, for example caps, i.e. the boundary between outside and membrane and inside and membrane. This refinement improves the prediction of the topology of the protein.

TMHMM http://www.cbs.dtu.dk/services/TMHMM/



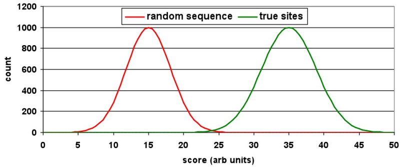
```
# Sequence Length: 274
# Sequence Number of predicted TMHs:  7
# Sequence Exp number of AAs in TMHs: 153.74681
# Sequence Exp number, first 60 AAs:  22.08833
# Sequence Total prob of N-in:        0.04171
# Sequence POSSIBLE N-term signal sequence
Sequence        TMHMM2.0        outside      1    26
Sequence        TMHMM2.0        TMhelix     27    49
Sequence        TMHMM2.0        inside      50    61
Sequence        TMHMM2.0        TMhelix     62    84
Sequence        TMHMM2.0        outside     85   103
Sequence        TMHMM2.0        TMhelix    104   126
Sequence        TMHMM2.0        inside     127   130
Sequence        TMHMM2.0        TMhelix    131   153
Sequence        TMHMM2.0        outside    154   157
Sequence        TMHMM2.0        TMhelix    158   180
Sequence        TMHMM2.0        inside     181   200
Sequence        TMHMM2.0        TMhelix    201   223
Sequence        TMHMM2.0        outside    224   227
Sequence        TMHMM2.0        TMhelix    228   250
Sequence        TMHMM2.0        inside     251   274
```



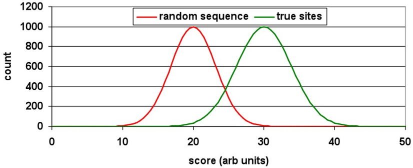
Assessing performances: Sensitivity and specificity

1. be predicted to occur: Predicted Positive (PP)
2. be predicted not to occur: Predicted Negative (PN)
3. actually occur: Actual Positive (AP)
4. actually not occur: Actual Negative (AN)
5. True Positive $TP = PP \bigcap AP$
6. True Negative $TN = PN \bigcap AN$
7. False Negative $FN = PN \bigcap AP$
8. False Positive $FP = PP \bigcap AN$
9. Sensitivity: probability of correctly predicting a positive example Sn = TP/(TP + FN)
10. Specificity: probability of correctly predicting a negative example Sp = TN/(TN + FP) or
11. probability that positive prediction is correct Sp = TP/(TP + FP)

Specificity/Sensitivity Tradeoffs



Ideal Distribution of Scores



More Realistically…

$$Sn = \frac{TruePositive}{AllTrue} = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$Sp = \frac{TruePositive}{AllPositive} = \frac{TruePositive}{TruePositive + FalsePositive}$$

Correlation Coefficient

$$CC = \frac{[(TP)(TN) - (FP)(FN)]}{\sqrt{(AN)(PP)(AP)(PN)}}$$

$$AN = TN + FP; AP = TP + FN;$$

$$PP = TP + FP; PN = TN + FN$$

Gibbs Sampling is an example of a Markov chain Monte Carlo algorithm, it is an iterative procedure that discards one l-mer after each iteration and replaces it with a new one. Gibbs Sampling proceeds slowly and chooses new l-mers at random increasing the odds that it will converge to the correct solution. It could be used to identify short strings, motifs, common to all co-regulated genes which are not co-aligned..

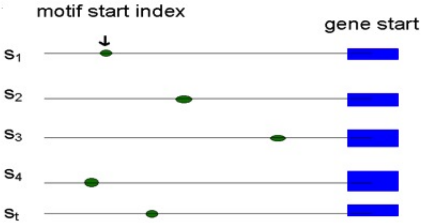
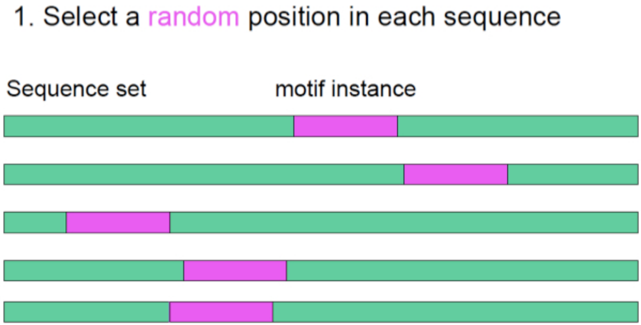


Figure: Several genes are co-regulated (activated or repressed) by same protein that binds before the gene start (transcription factor)

The Biological problem: given a set of sequences, find the motif shared by all or most sequences, while its starting position in each sequence is unknown; Each motif appears exactly once in one sequence, the motif has fixed length.

1. Randomly choose starting positions s = $(s_1,\ldots,s_t)$ and form the set of l-mers associated with these starting positions.
2. Randomly choose one of the t sequences
3. Create a profile p from the other t -1 sequences.
4. For each position in the removed sequence, calculate the probability that the l-mer starting at that position was generated by p.
5. Choose a new starting position for the removed sequence at random based on the probabilities calculated in step 4.
6. Repeat steps 2-5 until there is no improvement

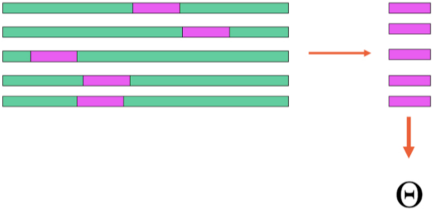
Considering a set of unaligned sequences, we choose initial guess of motifs



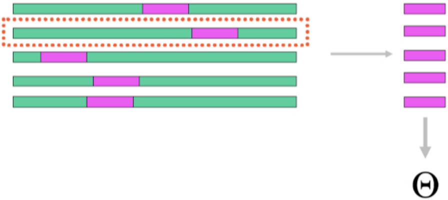
1. Select a random position in each sequence

Figure: motifs in purple, the rest of the sequences in green; next figures: theta is the weight matrix i.e. the frequency of each base in the aligned set of motifs; red the best fitting motif; in y axis the likelihood of each motif with respect to the current weight matrix.

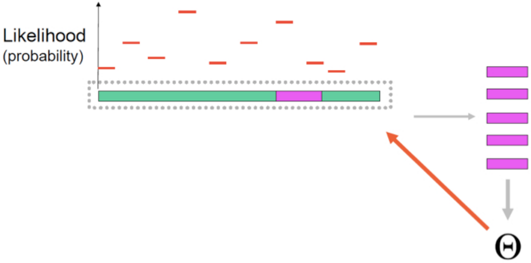
2. Build a weight matrix



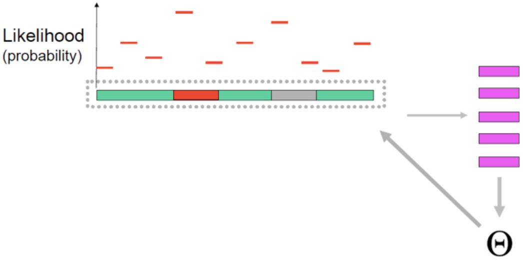
3. Select a sequence at random



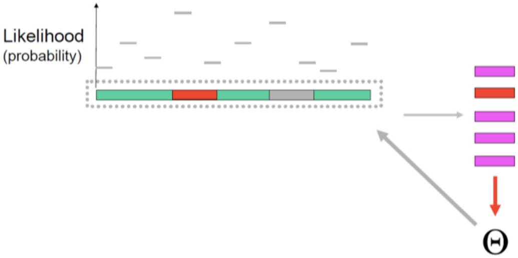
4. Score possible sites in seq using weight matrix



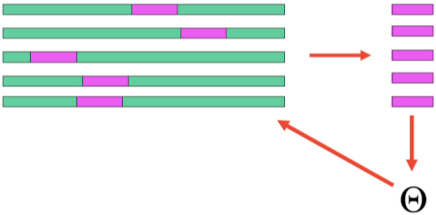
5. Sample a new site proportional to likelihood



6. Update weight matrix



7. Iterate until convergence (no change in sites/Θ)



Properties of Biological Networks

Let assume that there are two related genes, B and D neither is expressed initially, but E causes B to be expressed and this in turn causes D to be expressed the addition of CX by itself may not affect expression of either B or D both CX and E will have elevated levels of $mRNA_B$ and low levels of $mRNA_D$

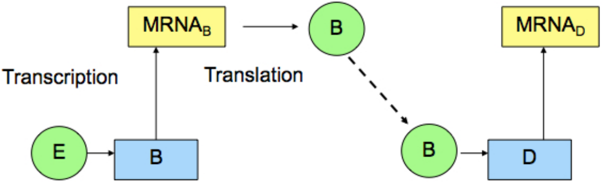


Figure: We have E only; B is a Primary Target of E; Production of $mRNA_B$ is enhanced by E; D is a Secondary Target of E; Production of $mRNA_D$ is enhanced by B

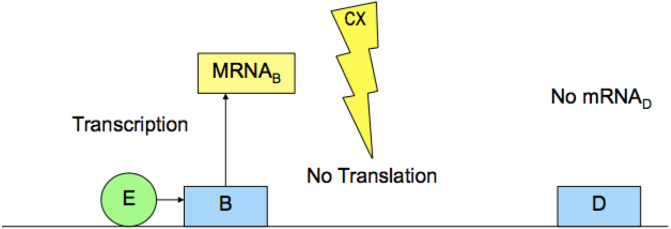


Figure: E and CX both present; B is a Primary Target; Production of RNA_B is enhanced by E; Production of RNA_D is decreased (prevented)

What is a genetic network?

A genetic network is a group of genes in which individual genes can influence the activity of other genes. What, then, is gene activity? Gene activity can include many different things. Most definitions revolve around gene expression, whether a gene is expressed or not, as RNA or as protein.

What is a genetic perturbation?

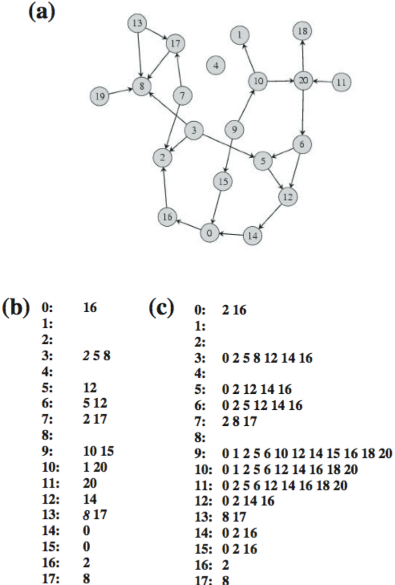
it is an experimental manipulation of gene activity by manipulating either a gene itself or its product. Such perturbations include point mutations, gene deletions, overexpression, inhibition of translation, or any other interference with the activity of the product.

Network reconstruction: direct and indirect effects

Network reconstruction: direct and indirect effects.
When manipulating a gene and finding that this manipulation affects the activity of other genes, the question often arises as to whether this is caused by a direct or indirect interaction?
An algorithm to reconstruct a genetic network from perturbation data should be able to distinguish direct from indirect regulatory effects.
Consider a series of experiments in which the activity of every single gene in an organism is manipulated. (for instance, non-essential genes can be deleted, and for essential genes one might construct conditional mutants.) The effect on mRNA expression of all other genes is measured separately for each mutant.

- How to reconstruct a large genetic network from n gene perturbations in fewer than n^2 steps?
- Motivation: perturb a gene network one gene at a time and use the effected genes in order to discriminate direct vs. indirect gene-gene relationships
- Perturbations: gene knockouts, over-expression, etc.
- Method: For each gene g_i , compare the control experiment to perturbed experiment and identify the differentially expressed genes Use the most parsimonious graph that yields the graph as its reachable graph
- Reference A. Wagner Bioinformatics 17, 1193-1197, 2001

The nodes of the graph correspond to genes, and two genes are connected by a directed edge if one gene influences the activity of the other.



(b)

0:	16
1:	
2:	
3:	2 5 8
4:	
5:	12
6:	5 12
7:	2 17
8:	
9:	10 15
10:	1 20
11:	20
12:	14
13:	8 17
14:	0
15:	0
16:	2
17:	8

(c)

0:	2 16
1:	
2:	
3:	0 2 5 8 12 14 16
4:	
5:	0 2 12 14 16
6:	0 2 5 12 14 16
7:	2 8 17
8:	
9:	0 1 2 5 6 10 12 14 15 16 18 20
10:	0 1 2 5 6 12 14 16 18 20
11:	0 2 5 6 12 14 16 18 20
12:	0 2 14 16
13:	8 17
14:	0 2 16
15:	0 2 16
16:	2
17:	8

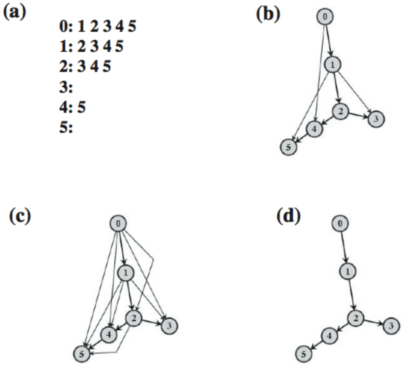


Figure: The figure illustrates three graphs (Figs. B,C,D) with the same accessibility list Acc (Fig. A). There is one graph (Fig. D) that has Acc as its accessibility list and is simpler than all other graphs, in the sense that it has fewer edges. Lets call Gpars the most parsimonious network compatible with Acc.

Figure A shows a graph representation of a hypothetical genetic network of 21 genes. Figure B shows an alternative representation of the network shown in A. For each gene i, it simply shows which genes activity state the gene influences directly. In graph theory, a list like that shown in Fig. B is called the adjacency list of the graph. We will denote it as Adj(G), and will refer to $Adj(i)$ as the set of nodes (genes) adjacent to (directly influenced by) node i. One might also call it the list of nearest neighbors in the gene network, or the list of direct regulatory interactions.

When perturbing each gene in the network shown in Figure A, one would get the list of influences on the activities of other genes shown in Figure C.

Starting from a graph representation of the network in Figure A, one arrives at the list of direct and indirect causal interactions in Figure C by following all paths leaving a gene. That is, one follows all arrows emanating from the gene until one can go no further.

The adjacency list completely defines the structure of a gene network

In graph theory, the list Acc(G) is called the accessibility list of the graph G, because it shows all nodes (genes) that can be accessed (influenced in their activity state) from a given gene by following paths of direct interactions.

In the context of a genetic network one might also call it the list of perturbation effects or the list of regulatory effects.

Acc(i) is the set of nodes that can be reached from node i by following all paths of directed edges leaving i. Acc(G) then simply consists of the accessibility list for all nodes i

The adjacency matrix of a graph G, $A(G) = (a_{ij})$ is an n by n square matrix, where n is the number of nodes (genes) in the graph. An element aij of this matrix is equal to one if and only if a directed edge exists from node i to node j. All other elements of the adjacency matrix are zero.

The accessibility matrix $P(G) = p_{ij}$ is also an n by n square matrix. An element p_{ij} is equal to one if and only if a path following directed edges exists from node i to node j. otherwise p_{ij} equals zero.

Adjacency and accessibility matrices are the matrix equivalents of adjacency and accessibility lists.

Lets first consider only graphs without cycles, where cycles are paths starting at a node and leading back to the same node.

Graphs without cycles are called acyclic graphs.

Later generalize to graphs with cycles.

An acyclic directed graph defines its accessibility list, but the converse is not true.

In general, if Acc is the accessibility list of a graph, there is more than one graph G with the same accessibility list

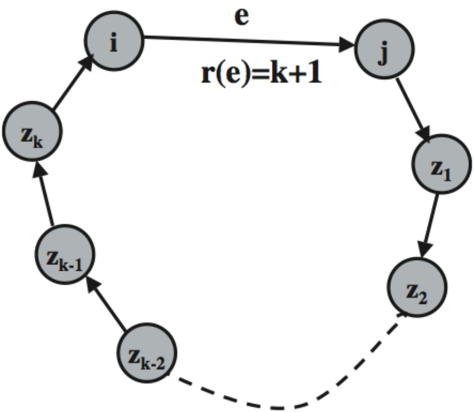
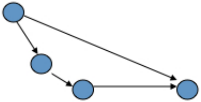


Figure: A shortcut is an edge connecting two nodes, i and j that are also connected via a longer path of edges. The shortcut e is a shortcut range k+1. That is, when eliminating e, I and j are still connected by a path of length k+1.

Wagner Algorithm

- Step1: Graphs without cycles only (acyclic directed graph)
- Step2: Graphs with cycles

- Step 1: Shortcut:



- A shortcut-free graph compatible with an accessibility list is a unique graph with the fewest edges among all graphs compatible with the accessibility list, i.e, a shortcut-free graph is the most parsimonious graph.

Theorem 1 (step 1)

- Let Acc be the accessibility list of an acyclic digraph. Then there exists exactly one graph Gpars that has Acc as its accessibility list and that has fewer edges than any other graph G with Acc as its accessibility list.
- This means that for any list of perturbation effects there exists exactly one genetic network G with fewer edges than any other network with the same list of perturbation effects.
- Definition: An accessibility list Acc and a digraph G are compatible if G has Acc as its accessibility list. Acc is the accessibility list induced by G.
- Definition: Consider two nodes i and j of a digraph that are connected by an edge e. The range r of the edge e is the length of the shortest path between i and j in the absence of e. If there is no other path connecting i and j, then r: $= \infty$.

Theorem 2 (step1)

Let Acc(G) be the accessibility list of an acyclic directed graph, Gpars its most parsimonious graph, and V(Gpars) the set of all nodes of Gpars . Then the following equation (1):
$\forall i \in V\left(G_{pars}\right) \ldots Adj\left(i\right) = Acc\left(i\right) \setminus \cup_{j \in Acc(i)} Acc\left(j\right)$
In words, for each node i the adjacency list Adj(i) of the most parsimonious genetic network is equal to the accessibility list Acc(i) after removal of all nodes that are accessible from any node in Acc(i).

Example

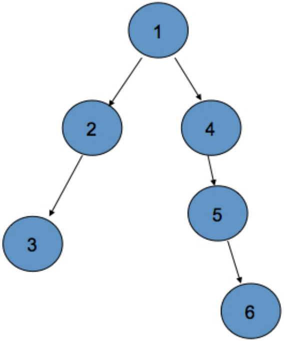


Figure: $Adj(1) = Acc(1)$ – $(Acc(2) + Acc(3) + Acc(4) + Acc(5) + Acc(6)) \ = (2, 3, 4, 5, 6)$ - $(3 \cup (5, 6) \cup 6) = (2, 4)$

Proof: I will first prove that every node in Adj(i) is also contained in the set defined by the right hand side of (1).
Let x be a node in Adj(i). This node is also in Acc(i). Now take, without loss of generality any node j $\in$ Acc(i). Could x be in Acc(j)? If x could be in Acc(j) then we could construct a path from i to j to x. But because x is also in Adj(i), there is also an edge from i to x. This is a contradiction to Gpars being shortcut-free. Thus, for no j $\in$ Acc(i) can x be in Acc(j). x is therefore also not an element of the union of all Acc(j) shown on the right-hand side of (1). Thus, subtracting this union from Acc(i) will not lead to the difference operator in (1) eliminating x from Acc(i). Thus x is contained in the set defined by the right-hand side of (1).

Next to prove: Every node in the set of the right-hand side of (1) is also in Adj(i).

Let x be a node in the set of the right-hand side of (1). Because x is in the right hand side of (1), x must a fortiori also be in Acc(i). That is, x is accessible from i. But x can not be accessible from any j that is accessible from i.

For if it were, then x would also be in the union of all Acc(j). Then taking the complement of Acc(i) and this union would eliminate x from the set in the right hand side of (1). In sum, x is accessible from i but not from any j accessible from i. Thus x must be adjacent to i.

The algorithm itself will use the following corollary to Theorem 2.

Corollary 2: Let i, j, and k be any three pairwise different nodes of an acyclic directed shortcut-free graph G. If j is accessible from i, then no node k accessible from j is adjacent to i.

Proof: Let j be a node accessible from node i. Assume that there is a node k accessible from j, such that k is adjacent to i. That is, j $\in$ Acc(i), k $\in$ Acc(j) and k $\in$ Adj(i). That k is accessible from j implies that there is a path of length at least one from j to k. For the same reason, there exists a path of length at least one connecting i to j. In sum, there must exist a path of length at least two from i to k. However, by assumption, there also exists a directed edge from i to k. Thus, the graph G can not be short-cut free.

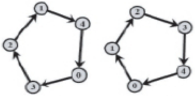
Step 2: How about graphs with cycles?

Two different cycles have the same accessibility list

Perturbations of any gene in the cycle influences the activity of all other genes in the same cycle

Cant decide a unique graph if cycle happens

Not an algorithmic but an experimental limitation



0:	3	0:	1
1:	4	1:	2
2:	1	2:	3
3:	2	3:	4
4:	0	4:	0

0:	1 2 3 4
1:	0 2 3 4
2:	0 1 3 4
3:	0 1 2 4
4:	0 1 2 3

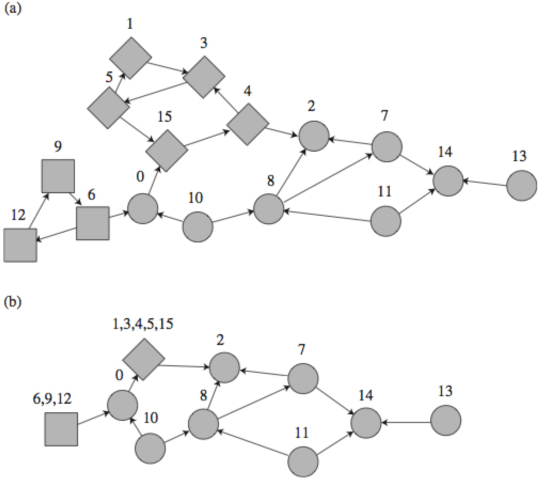


Figure: Basic idea: Shrink each cycles (strongly connected components) into one node and apply the algorithm of step 1. A graph after shrinking all the cycles into nodes is called a condensation graph

How good is this algorithm?

1. Unable to resolve cycled graphs
2. Require more data than conventional methods using gene expression correlations.
3. There are many networks consistent with the given accessibility list. The algorithm construct the most parsimonious one.
4. The same problem was proposed around 1980 which is called transitive reduction.
5. The transitive reduction of a directed graph G is the directed graph G' with the smallest number of edges such for every path between vertices in G, G' has a path between those vertices.
6. An O(V) algorithm for computing transitive reduction of a planar acyclic digraph was proposed by Sukhamay Kundu. (V is the number of nodes in G)

Complexity

- Measures of algorithmic complexity are influenced by the average number of entries in a nodes accessibility list. Let $k < n - 1$ be that number.
- For all practical purposes, there will be many fewer entries than that, not only because accessibility lists with nearly n entries are not accessibility lists of acyclic digraphs, but also because most real-world graphs are sparse.
- During execution, each node accessible from a node j induces one recursive call of PRUNEACC, after which the node accessed from j is declared as visited.
- Thus, each entry of the accessibility list of a node is explored no more than once.
- However, line 15 of the algorithm loops over all nodes k adjacent to j. If $a = |Adj(j)|$, on average, then overall computational complexity becomes $O(nka)$.

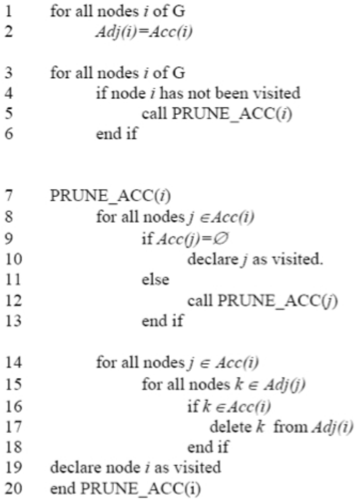
Comments on the code

The algorithm itself takes the accessibility list of a graph and eliminates entries inconsistent with Theorem 2 and Corollary 2.

It does so recursively until only the adjacency list of the shortcut-free graph is left.

The algorithm is shown as pseudocode. Because it operates on lists, programming languages such as perl or library extensions of other languages permitting list operations will facilitate its implementation.

(In Appendix a perl implementation of the algorithm, where accessibility and adjacency list are represented by a two-dimensional hashing array.)



```
for all nodes $i$ of G
    $Adj(i)=Acc(i)$

for all nodes $i$ of G
    if node $i$ has not been visited
        call PRUNE_ACC($i$)
    end if


PRUNE_ACC($i$)
    for all nodes $j \in Acc(i)$
        if $Acc(j)=\varnothing$
            declare $j$ as visited.
        else
            call PRUNE_ACC($j$)
        end if

    for all nodes $j \in Acc(i)$
        for all nodes $k \in Adj(j)$
            if $k \in Acc(i)$
                delete $k$ from $Adj(i)$
            end if
declare node $i$ as visited
end PRUNE_ACC(i)
```

The algorithm needs an accessibility list for each node *i, Acc(i)*, which would be obtained from gene perturbation data and subsequent gene activity measurements for a genetic network.

In lines one and two, for each no
i the adjacency list *Adj(i)* is initialized as equal to the accessibility list.

The algorithm will delete elemen
from this *Adj(i)* until the adjacenc
list of the most parsimonious network of *Acc(G)* is obtained.

```
1    for all nodes $i$ of G
2        $Adj(i)=Acc(i)$

3    for all nodes $i$ of G
4        if node $i$ has not been visited
5            call PRUNE_ACC($i$)
6        end if

7    PRUNE_ACC($i$)
8        for all nodes $j \in Acc(i)$
9            if $Acc(j)=\varnothing$
10               declare $j$ as visited.
11           else
12               call PRUNE_ACC($j$)
13           end if

14       for all nodes $j \in Acc(i)$
15           for all nodes $k \in Adj(j)$
16               if $k \in Acc(i)$
17                   delete $k$ from $Adj(i)$
18               end if
19   declare node $i$ as visited
20   end PRUNE_ACC(i)
```

The master loop in lines 3-6 cycles over all nodes of *G*, and calls the routine PRUNE_ACC for each node *i*.

In the last statement of this routine (line 19) the calling node is declared as visited.

A visited node is a node whose adjacency list *Adj(i)* needs not be modified any further.

This is the purpose of the conditional statement in the master loop (line 4), which skips over nodes that have already been visited.

```
1     for all nodes $i$ of G
2           $Adj(i)=Acc(i)$

3     for all nodes $i$ of G
4           if node $i$ has not been visited
5                 call PRUNE_ACC($i$)
6           end if


7     PRUNE_ACC($i$)
8           for all nodes $j \in Acc(i)$
9                 if $Acc(j)=\varnothing$
10                      declare $j$ as visited.
11                else
12                      call PRUNE_ACC($j$)
13                end if

14          for all nodes $j \in Acc(i)$
15                for all nodes $k \in Adj(j)$
16                      if $k \in Acc(i)$
17                         delete $k$  from $Adj(i)$
18                      end if
19    declare node $i$ as visited
20    end PRUNE_ACC(i)
```

Aside from storing *Acc* and *Adj*, the algorithm thus also needs to keep track of all visited nodes.

In an actual implementation, *Acc*, *Adj*, and any data structure that keeps track of visited nodes would need to be either global variables or passed into the routine PRUNE_ACC, preferably by reference.

In contrast, the calling node *i* needs to be a local variable because of the recursivity of PRUNE_ACC.

```
for all nodes $i$ of G
    $Adj(i)=Acc(i)$

for all nodes $i$ of G
    if node $i$ has not been visited
        call PRUNE_ACC($i$)
    end if


PRUNE_ACC($i$)
    for all nodes $j \in Acc(i)$
        if $Acc(j)=\varnothing$
            declare $j$ as visited.
        else
            call PRUNE_ACC($j$)
        end if

    for all nodes $j \in Acc(i)$
        for all nodes $k \in Adj(j)$
            if $k \in Acc(i)$
              delete $k$ from $Adj(i)$
            end if
declare node $i$ as visited
end PRUNE_ACC(i)
```

Function PRUNE_ACC

It contains of two loops. The first loop (lines 8-13) cycles over all nodes *j* accessible from the calling node *i*. If there exists a node accessible from *j*, then PRUNE_ACC is called from *j*. If no node is accessible from *j*, that is, if *Acc(j)* =∅, then *j* is declared as visited.

Because its accessibility list is empty, its adjacency list must be empty as well (*Adj(i)* ⊆*Acc(i))*, and needs no further modification.

```
1     for all nodes $i$ of G
2         $Adj(i)=Acc(i)$

3     for all nodes $i$ of G
4         if node $i$ has not been visited
5             call PRUNE_ACC($i$)
6         end if


7     PRUNE_ACC($i$)
8         for all nodes $j \in Acc(i)$
9             if $Acc(j)=\varnothing$
10                declare $j$ as visited.
11            else
12                call PRUNE_ACC($j$)
13            end if

14        for all nodes $j \in Acc(i)$
15            for all nodes $k \in Adj(j)$
16                if $k \in Acc(i)$
17                    delete $k$  from $Adj(i)$
18                end if
19    declare node $i$ as visited
20    end PRUNE_ACC(i)
```

Thus, through the first loop PRUNE_ACC calls itself recursively until a node is reached whose accessibility list is empty.

There always exists such a node, otherwise the graph would not be acyclic.

This also means that infinite recursion is not possible for an acyclic graph. Thus, the algorithm always terminates.

More precisely, the longest possible chain of nested calls of PRUNE_ACC is *(n-1)* if G has ***n*** nodes.

For any node ***i*** calling PRUNE_ACC, the number of nested calls is at most equal to the length of the longest path starting at ***i***.

```
for all nodes $i$ of G
    $Adj(i)=Acc(i)$

for all nodes $i$ of G
    if node $i$ has not been visited
        call PRUNE_ACC($i$)
    end if


PRUNE_ACC($i$)
    for all nodes $j \in Acc(i)$
        if $Acc(j)=\emptyset$
            declare $j$ as visited.
        else
            call PRUNE_ACC($j$)
        end if

    for all nodes $j \in Acc(i)$
        for all nodes $k \in Adj(j)$
            if $k \in Acc(i)$
               delete $k$ from $Adj(i)$
            end if
declare node $i$ as visited
end PRUNE_ACC(i)
```

The second loop of PRUNE_ACC (lines 14-18) only starts once the algorithm has explored all nodes accessible from the calling node *i*, that is, as the function calls made during the first loop return.

In the second loop the principle of Corollary 2 is applied.

Specifically, the second loop cycles over all nodes *j* accessible from *i* in line 14.

```
1     for all nodes $i$ of G
2         $Adj(i)=Acc(i)$

3     for all nodes $i$ of G
4         if node $i$ has not been visited
5             call PRUNE_ACC($i$)
6         end if


7     PRUNE_ACC($i$)
8         for all nodes $j \in Acc(i)$
9             if $Acc(j)=\varnothing$
10                declare $j$ as visited.
11            else
12                call PRUNE_ACC($j$)
13            end if

14        for all nodes $j \in Acc(i)$
15            for all nodes $k \in Adj(j)$
16                if $k \in Acc(i)$
17                    delete $k$ from $Adj(i)$
18                end if
19    declare node $i$ as visited
20    end PRUNE_ACC(i)
```

In a slight deviation from what Corollary 2 suggests, line 15 cycles not over all nodes $k \in Acc(j)$, but only over $k \in Adj(j)$.

All nodes $k \in Adj(j)$ are deleted from $Adj(i)$ in lines 16-18. Cycling only over $k \in Adj(j)$ saves time, but does not compromise the requirement that all nodes $k \in Adj(i)$ be removed, because line 14 covers all nodes *j* accessible from *i*.

Because of the equality proven in Theorem 2, once this has been done, the adjacency list need not be modified further. This is why upon leaving this routine, the calling node is declared as visited.

Notice also that if a node *j* with $Acc(j) = \in$is encountered, the loop in line 15 is not executed.

```
1     for all nodes $i$ of G
2         $Adj(i)=Acc(i)$

3     for all nodes $i$ of G
4         if node $i$ has not been visited
5             call PRUNE_ACC($i$)
6         end if

7     PRUNE_ACC($i$)
8         for all nodes $j \in Acc(i)$
9             if $Acc(j)=\varnothing$
10                declare $j$ as visited.
11            else
12                call PRUNE_ACC($j$)
13            end if

14        for all nodes $j \in Acc(i)$
15            for all nodes $k \in Adj(j)$
16                if $k \in Acc(i)$
17                   delete $k$ from $Adj(i)$
18                end if
19    declare node $i$ as visited
20    end PRUNE_ACC(i)
```

```
for all nodes $i$ of $G$
    if $component[i]$ has not been defined
        create new node $x$ of $G*$
        $component[i]=x$
        for all nodes $j \in Acc(i)$
            if $i \in Acc(j)$
                $component[j]=x$
            end if
    end if

for all nodes $i$ of $G*$
    $Acc_{G*}(i)=\emptyset$
for all nodes $i$ of $G$
    for all nodes $j \in Acc(i)$
        if $component[i] \neq component[j]$
            if $component[j] \notin Acc_{G*}(component[i])$
                add $component[j]$ to $Acc_{G*}(component[i])$
            end if
        end if
```

$\star$Topic: Gillespie algorithm

Consider a system of N molecular species $S_1, , S_N$ interacting through M elemental chemical reactions $R_1, , R_M$.
We assume that the system is confined to a constant volume W and is well stirred and at a constant temperature. Under these assumptions, the state of the system can be represented by the populations of the species involved.
We denote these populations by $X(t)X_1(t), , X_N(t)$, where $Xi(t)$ is the number of molecules of species S_i in the system at time t. The well stirred condition is crucial. For each reaction R_j, a propensity function a_j, such that $a_j(x)dt$ the probability, given $X(t) = x$, that one R_j reaction will occur in time interval $[t, t + dt)$. State change vector v_j, whose ith component is defined by $v_{j,i}$ the change in the number of S_i molecules produced by one R_j reaction.

The most important method to simulate a network of biochemical reactions is Gillespies stochastic simulation algorithm (SSA)

1. The Gillespie algorithm is widely used to simulate the behavior of a system of chemical reactions in a well stirred container
2. The key aspects of the algorithm is the drawing of two random numbers at each time step, one to determine after how much time the next reaction will take place, the second one to choose which one of the reactions will occur.
3. Each execution of the Gillespie algorithm will produce a calculation of the evolution of the system. However, any one execution is only a probabilistic simulation, and the chances of being the same as a particular reaction is vanishingly small.
4. Therefore to garner any useful information from the algorithm, it should be run many times in order to calculate a stochastic mean and variance that tells us about the behaviour of the system.
5. the complexity of the Gillespie algorithm is $O(M)$ where M is the number of reactions.

Gillespie Algorithm

1. Initialise: set the initial molecule copy numbers, set time $t = 0$.
2. Calculate the propensity function a_i for each reaction, and the total propensity according to equation $a_0\left(x\right) \equiv \sum_{j=1}^{M} a_j\left(x\right)$, i = 1,...,M.
3. Generate two uniformly distributed random numbers r_1 and r_2 from the range (0, 1).
4. Compute the time τ to the next reaction using equation $\tau = \frac{1}{a_{0(x)}} ln\left(\frac{1}{r_1}\right)$.
5. Decide which reaction R_μ occurs at the new time using equation $r_2 > \sum_{k=1}^{\mu-1} a_k \ldots and \ldots r_2 < \frac{1}{a_0} \sum_{k=1}^{\mu-1} a_k$.
6. Update the state vector v by adding the update vector : $v(t+\tau) = v(t) + (\nu)_\mu$
7. Set $t = t + \tau$. Return to step 2 until t reaches some specified limit t_{MAX}.

In each step, the SSA starts from a current state $x(t) = x$ and asks two questions: When will the next reaction occur? We denote this time interval by t . When the next reaction occurs, which reaction will it be? We denote the chosen reaction by the index j. To answer the above questions, one needs to study the joint probability density function $p(\tau, j \mid x, t)$ that is the probability, given $X(t) = x$, that the next reaction will occur in the infinitesimal time interval $[t + \tau, t + \tau + dt]$. The theoretical foundation of SSA is given by $p(\tau, j \mid x, t) = a_j(x)\, exp(-a_0(x)\,\tau)$, where $a_0(x) \equiv \sum_{j=1}^{M} a_j(x)$ It implies that the time t to the next occurring reaction is an exponentially distributed random variable with mean $1/a_0(x)$, and that the index j of that reaction is the integer random variable with point probability $a_j(x)/a_0(x)$. The τ is $\tau = \frac{1}{a_{0(x)}} ln\left(\frac{1}{r_1}\right)$

The system state is then updated according to $X(t + \tau) = x + \nu_j$ and this process is repeated until the simulation final time or until some other terminating condition is reached.

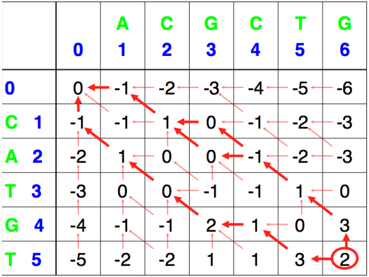
Interesting websites for practicals

Tutorials for Molecular Biology (accessible to computer science students) http://www.thomas-schlitt.net/Bioproject.html; http://www.biostat.wisc.edu/ craven/hunter.pdf
Data Repository: http://www.ncbi.nlm.nih.gov/ ; Human Genome Browser Gateway http://genome.ucsc.edu/ www.ensembl.org ; http://www.ebi.ac.uk
Progressive alignment:
http://www.ebi.ac.uk/Tools/msa/clustalw2/;
ftp://ftp.ebi.ac.uk/pub/software/.
Phylogenetic software repository:
http://evolution.genetics.washington.edu/phylip/software.html
HMM: http://www.cbs.dtu.dk/services/TMHMM/;
http://genes.mit.edu/GENSCAN.html
Various libraries to help with Bio data BioJava www.biojava.org ; BioPerl www.bioperl.org ; BioPython www.biopython.org ; BioCorba www.biocorba.org ; C++
www.ncbi.nlm.nih.gov/IEB/ToolBox/

Examples of Exam Questions

- Align the two strings: ACGCTG and CATGT, with match score =1 and mismatch, gap penalty equal -1
- Describe with one example the difference between Hamming and Edit distances
- Discuss the complexity of an algorithm to reconstruct a genetic network from microarray perturbation data
- Discuss the properties of the Markov clustering algorithm and the difference with respect to the k-means and hierarchical clustering algorithms

Examples of Answers Align the two strings: ACGCTG and CATGT, with match score =1 and mismatch, gap penalty equal -1



			A	C	G	C	T	G
		0	1	2	3	4	5	6
	0	0	-1	-2	-3	-4	-5	-6
C	1	-1	-1	1	0	-1	-2	-3
A	2	-2	1	0	0	-1	-2	-3
T	3	-3	0	0	-1	-1	1	0
G	4	-4	-1	-1	2	1	0	3
T	5	-5	-2	-2	1	1	3	2

Describe with one example the difference between Hamming and Edit distances *TGCATAT* → *ATCCGAT* in 4 steps; TGCATAT (insert A at front); ATGCATAT (delete 6th T); ATGCATA (substitute G for 5th A); ATGCGTA (substitute C for 3rd G); ATCCGAT (Done).

Examples of Answers

Discuss the complexity of an algorithm to reconstruct a genetic network from microarray perturbation data

Reconstruction: O(nka) where n is the number of genes, k is the average number of entries in the accession list; a is the average number of entries in adjacency list. Large scale experimental gene perturbations in the yeast Saccharomyces cerevisiae (n=6300) suggests that $k < 50$, $a < 1$, and thus that $nka << n^2$.

Discuss the properties of the Markov clustering algorithm and the difference with respect to the k-means and hierarchical clustering algorithms

MCL algorithm: We take a random walk on the graph described by the similarity matrix and after each step we weaken the links between distant nodes and strengthen the links between nearby nodes.

The k-means algorithm is composed of the following steps: 1) Place K points into the space represented by the objects that are being clustered. These points represent initial group centroids. 2) Assign each object to the group that has the closest centroid. 3) When all objects have been assigned, recalculate the positions of the K centroids. 4) Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.

Hierarchical clustering: Start with each point its own cluster. At each iteration, merge the two clusters; with the smallest distance. Eventually all points will be linked into a single cluster. The sequence of mergers can be represented with a rooted tree.