Bio informatics

- Sequence alignment, Hamming distance, Edit distance, edit graph
- LCS problem, global alignment, local alignment
- Needleman-Wunsch algorithm, Smith-Waterman algorithm, affine gap penalty, Hirschberg algorithm
- Nussinov algorithm
- BLAST, spaced seed, PH
- Multiple pairwise alignment, progressive alignment, CLUSTAL
- BWT
- Hamiltonian graph, De Bruijn graph, Eulerian path
- Phylogeny, Fitch parsimony, Sankoff parsimony, score matrix
- Distance-based phylogeny, additivity property, UPGMA, ultrametric property, Neighbor-Joining
- Micro array data, clustering algorithm, Lloyd's algorithm, progressive greedy K-means, Manor Clustering.
- HMM, Viterbi algorithm, Forward-Backward algorithm, HMM performance
- Gibbs sampling
- Accessibility list, adjacency list, Wagner algorithm
- Gillespie algorithm, assumptions

Bioinformatics

1. What is sequence alignment?

Alignment is a way of arranging two DNA or protein sequences ~~that~~ to identify regions of similarity that are conserved among species. The differences in these sequences are because of mutations.
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.

2. Describe with one example the difference between Hamming and Edit distances. [2007 P8 Q13 (a)]

The Hamming distance between two sequences is simply the number of mismatches if we align the ith symbol of one sequence with the ith symbol of the other.
e.g. 1010 has Hamming distance 4
0101

The Edit distance is the minimum number of operations required to make the sequences match perfectly.
The operations typically are insertion, deletion, and substitution.
e.g. 1010- has Edit distance 2, where - is an insertion
-0101

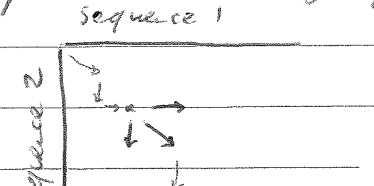
3. What is the idea behind Dynamic Programming?

DP is a bottom-up mechanism: we solve all possible small (sub) problems and then combine them to obtain solutions for bigger

problems. We rely on the fact that we can reduce a complex problem to a set of identical sub-problems which can be solved independently, and then combined. to solve the complex problem.
Our increase in efficiency comes from the fact that with a naive approach (e.g. recursion; top-down) we may solve the same small subproblem multiple times, but with DP we store the results of subproblems to avoid this unnecessary recomputation.

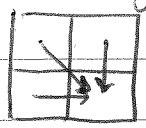
4. Why do we use dynamic programming for sequence alignment problems? [2012 Pg Q1(a)(i)]

In sequence alignment we effectively try to find the shortest path from one end of a graph to the other; this is called the edit graph. We try to find the alignment with the lowest cost, where at each point in the graph we can either:



- go right (gap in sequence 2)
- go down (gap in sequence 1)
- go down-and-right, diagonally
  (either a match, if the bases or amino acids of sequence 1 and 2 are the same in that entry we reach; or a mismatch otherwise).

Different costs for gaps, matches, and mismatches have different results.
We can see that the result of any entry in the edit graph depends on at most three other entries: we set the value of an entry to the minimum of:

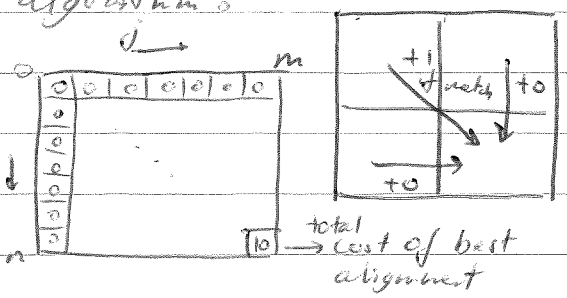


the value of the entry left of it + the cost of a gap,
the value of the entry above it + the cost of a gap,
or the value of the entry diagonally to the left-and-above

+ the cost of a match/mismatch, depending on whether the bases or amino acids of the two sequences match for that square. As a result, we can easily compute a new entry's value in constant time by storing these three other entries in the edit graph. This is exactly what dynamic programming is! As a result, we get a time complexity of $O(n \cdot m)$, where n and m are the lengths of sequences 1 and 2.

5. What is the Longest Common Subsequence problem?

The LCS problem is the simplest form of sequence alignment, allowing only insertions and deletions as operations (no mismatches; — equivalently, the cost of a mismatch is ∞). We score 1 for matches, and 0 for gaps (= insertions and deletions).
We can solve it using the Edit graph in $O(n \cdot m)$ time and space, using the following (DP) algorithm:




```
for i = 0 to n:   S_{i,0} = 0
for j = 0 to m:   S_{0,j} = 0
for i = 1 to n:
    for j = 1 to m:
```

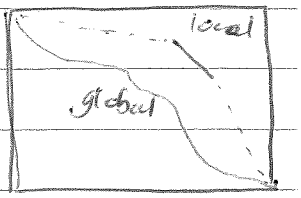
$$S_{i,j} = \max \begin{cases} S_{i-1,j} + 0 & \text{(gap in vertical sequence)} \\ S_{i,j-1} + 0 & \text{(gap in horizontal sequence)} \\ S_{i-1,j-1} + 1 & \text{if } v_i = w_j \text{ (match)} \end{cases}$$

where v_i and w_j are the ith base or amino acid of the two sequences.
To recover the alignment from the edit graph we remember where we "came" from using a matrix of backpointers:

$$b_{i,j} = \begin{cases} \uparrow & \text{if } S_{i,j} = S_{i-1,j} \\ \leftarrow & \text{if } S_{i,j} = S_{i,j-1} \\ \nwarrow & \text{if } S_{i,j} = S_{i-1,j-1} + 1 \text{ and } v_i = v_j \end{cases}$$

And we return both matrices.

6. What are the global and local alignment problems?



The <u>global alignment problem</u> tries to find the longest path between vertices $(0,0)$ and (n,m) in the edit graph; i.e. a globally optimum path.

The <u>local alignment problem</u> tries to find the longest path among paths between arbitrary vertices (i,j) and (i',j') in the edit graph; i.e. a locally optimum path.

We can find the global alignment using e.g. the Needleman-Wunsch algorithm, and the local alignment using e.g. the Smith-Waterman algorithm.

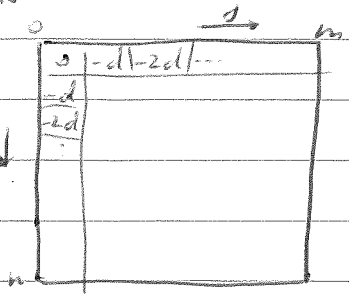
7. Discuss the <u>Needleman-Wunsch algorithm</u>.

This algorithm computes the global alignment of two sequences of lengths n and m in $O(n \cdot m)$ time and space, using dynamic programming.

We allow gaps, matches, and mismatches, at custom costs. Let d be the cost of a gap, and $c(x_i, y_j)$ the cost of a match if $x_i = y_j$, and the cost of a mismatch otherwise.

where x_i is the ith base or amino acid in the vertical sequence, and y_j the jth in the horizontal sequence

We then get the following pseudocode:



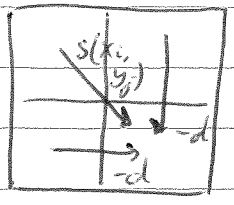
for $i = 0$ to n: $S_{i,0} = -i \cdot d$

for $j = 0$ to m: $S_{0,j} = -j \cdot d$

for $i = 1$ to n:

  for $j = 1$ to m:

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

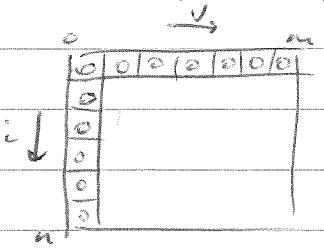


$$b_{i,j} = \begin{cases} \uparrow & \text{if } S_{i,j} = S_{i-1,j} - d \\ \leftarrow & \text{if } S_{i,j} = S_{i,j-1} - d \\ \nwarrow & \text{if } S_{i,j} = S_{i-1,j-1} + c(x_i, y_j) \end{cases}$$

As before, $S_{n,m}$ gives the total cost of the best global alignment, and we can recover the alignment using b.

8. I Discuss the Smith-Waterman algorithm [2007 P8 Q13(b)]

The Smith-Waterman algorithm solves the local alignment problem of finding local alignments between two sequences of bases (for DNA) or amino acids (for protein sequences). It is a simple modification of the Needleman-Wunsch algorithm that resets any score below 0 to 0 (i.e. adding a fourth term to the max-term). As a result, we ignore badly aligned regions, and find locally well-aligned sequences. The pseudocode is:




```
for i = 0 to n: S_{i,0} = 0
for j = 0 to m: S_{0,j} = 0

for i = 1 to n:
    for j = 1 to m:
```

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

$$b_{i,j} = \begin{cases} \text{null (end of local alignment)} & \text{if } S_{i,j} = 0 \\ \uparrow & \text{if } S_{i,j} = S_{i-1,j} - d \\ \leftarrow & \text{if } S_{i,j} = S_{i,j-1} - d \\ \nwarrow & \text{if } S_{i,j} = S_{i-1,j-1} + s(x_i, y_j) \end{cases}$$

This algorithm also runs in $O(n \cdot m)$ time and space.

If we want to find the best local alignment, we find the highest entry, and then trace back using $b_{i,j}$ until we get null.
If we want all alignments with a score of at least t, we find all i, j with $S_{i,j} \geq t$, and then trace back similarly to before.

9. Discuss how the alignment score and the quality of the results depends on the match score, mismatch, and gap penalty [2013 P7 Q3 (b)]

The two types of alignments are:
- Global (an alignment from the starts of the sequences to the ends)
- Local (identifying local areas inside the sequences with particularly high similarity).

By changing the score and penalties we can get different results for the algorithms computing these two types of alignments.
If we have a high mismatch penalty we tend to get more gaps in our alignments as mismatches are discouraged.
If we have a high gap penalty we will get more mismatches and fewer gaps.
Finally, having a high match score results in a more selective use of mismatches, resulting in more gaps.
The alignment score itself does not tell us much, as we can simply scale it up or down by scaling the three scores.
Taken to an extreme, having an infinite (or very high) match score results in solving the Longest Common Subsequence problem, as we insert as many gaps as necessary to match two bases or amino acids.
For a very low gap score we can identify good local alignments as we introduce a lot of gaps to reach well-aligning regions of the sequences.

10. Describe what needs to be taken into account for gaps in DNA sequence alignment. [2012 P9 Q1(a)(ii)]

Mutations in DNA can cause deletions, insertions, but also base changes (e.g. G → C). As a result, where we previously only considered gaps and matches with the LCS problem, we now also need to consider the possibility of a mismatch arising from a mutation in the DNA.
We need to penalise each expected mutation in the DNA sensibly, so we can properly identify the alignment of two sequences. This is done by setting the gap penalty, match score, and mismatch penalty.
For longer gaps, as with DNA sequences, we need to use an affine gap penalty.

11. Compare the use of the affine gap penalty with the constant gap penalty. [2006 P8 Q13(b)]

With the constant gap penalty, we have a set penalty for each gap we introduce in the alignment, even for consecutive gaps in the same alignment.
However, if there are many gaps we may not want to penalise long gap sequences too much. Therefore we introduce a gap penalty for starting a gap, and another (smaller) penalty for continuing a penalty:

$$f(n) = \underbrace{d}_{\text{gap open}} + (n-1) \cdot \underbrace{e}_{\text{gap extend}}$$

This is called the <u>affine gap penalty</u>.
In order to use it in our sequence alignments, we now need to remember whether our best score of a previous vertice in the edge graph just had a gap in the same direction as well. We therefore need an algorithm keeping track of four matrices. As before, our space and time complexity is $O(n \cdot m)$ for the trivial implementation.

The matrices we keep track of are:
- the best cost matrix V
- the match/mismatch matrix F
- the matrix for affine gaps in the i-direction (vertical) G
- the matrix for affine gaps in the j-direction (horizontal) H.

The pseudocode is as follows:

for $i = 1$ to n: $V_{i,0} = d + (i-1) \cdot e$
for $j = 1$ to m: $V_{0,j} = d + (j-1) \cdot e$
$V_{0,0} = 0$

for $i = 1$ to n:
  for $j = 1$ to n:
    $F_{i,j} = V_{i-1,j-1} + S(x_i, y_j)$

$$G_{i,j} = \max \begin{cases} V(i-1,j) - d & \text{(start new gap)} \\ G(i-1,j) - e & \text{(extend gap)} \end{cases}$$

$$H_{i,j} = \max \begin{cases} V(i,j-1) - d & \text{(start new gap)} \\ G(i,j-1) - e & \text{(extend gap)} \end{cases}$$

$$V_{i,j} = \max \{ F_{i,j}, G_{i,j}, H_{i,j} \}$$

$$b_{i,j} = \begin{cases} \leftarrow & \text{if } V_{i,j} = H_{i,j} \\ \uparrow & \text{if } V_{i,j} = G_{i,j} \\ \nwarrow & \text{if } V_{i,j} = F_{i,j} \end{cases}$$

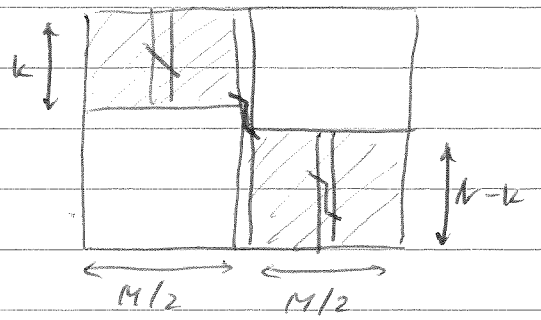
12. [Discuss the space-time complexity of dynamic algorithms in sequence alignment. [2010 P7 Q5 (a)]]

Dynamic programming algorithms rely on storing the results of sub-problems, and combining these to give the solutions to bigger problems.

In sequence alignment, this involves building an $(n+1) \times (m+1)$ matrix, for two sequences of lengths n and m, and gradually filling this matrix. We also store pointers in this "graph" that tell us the path of the alignment in the graph.
The above approach results in a space and time complexity of $O(n \cdot m)$, because we do a constant-time computation for each entry in the graph (= matrix), and store a constant amount of data for each entry as well.

We can improve on the space complexity, while having the same time complexity, using the Hirschberg algorithm, which reduces the time complexity to $O(n)$, the size of a single column. We notice that, to compute the value of an entry, we only need the values left of it, above it, and left-and-above (diagonally). Therefore, to compute the values of one column we only need the values of that column so far, plus the values of the previous column, costing $O(n)$ space. (or $O(m)$, swapping the two sequences).
The algorithm works as follows:

- We use a divide-and-conquer strategy to repeatedly split the problem in two.
- For each (slab) problem, we compute the values in the matrix one-by-one, until we reach the middle column.
- We then work backwards, now starting in the last column (rather than the first column), until we reach the same middle column.
- We add up the values of these two columns, and find the highest score. Our path definitely passes through this point, as well as any points on the path we can trace using the back pointer of this point. We remember these points.
- The place where this trace leaves and enters the middle column gives us the subproblem, which we recursively solve. By then merging the paths of the two subgraphs, we find our overall path.



We only ever remember at most two columns, at $O(n)$ space cost, and the path through the graph, which is of length at most $2n$, i.e. $O(n)$ space as well.

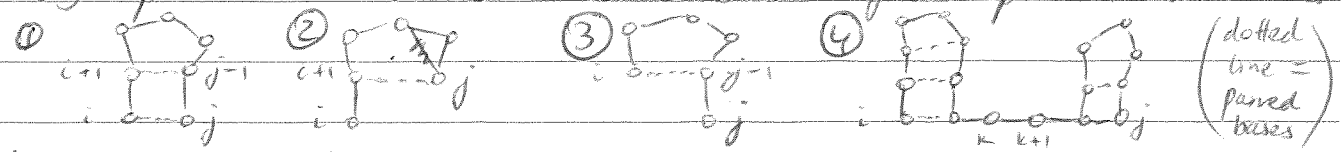
To see why this is $O(n \cdot m)$ time still, notice that at each step we reduce the problem size by half (grey area in figure). Therefore, our computational cost is:

$$n \cdot m + \frac{n \cdot m}{2} + \frac{n \cdot m}{4} + \frac{n \cdot m}{8} + \ldots < 2 \cdot n \cdot m$$

times the cost of computing one entry ($O(1)$), so overall a time complexity of $O(n \cdot m)$.

13. Discuss, with one example, the complexity of the Nussinov algorithm for RNA folding.

Many molecules of RNA do not translate into proteins, but instead fold up (using pairings A-T, C-G) into 2D-shapes, and then into 3-D shapes. They then regulate cell processes by interacting with proteins.
The intra-chain folding of RNA (i.e. with itself) reveals the RNA secondary structure, telling us which bases are paired.
Every optimal structure can be built from optimal substructures:



i, j paired i unpaired j unpaired bifurcation

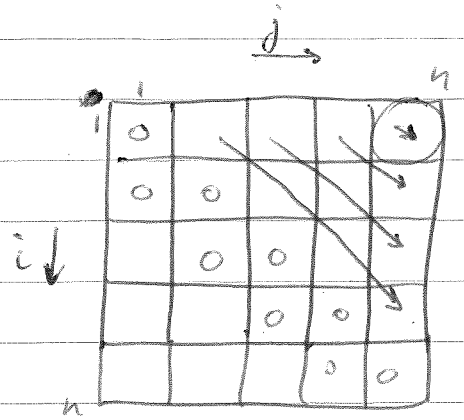
The final example shows two optimal substructures, with in-between two connected bases k and $k+1$

The Nussinov algorithm is a dynamic algorithm that uses the above four cases to compute the intra-chain RNA folding.

We use $\gamma(i,j)$ to denote the maximum number of base pairs in a folding of subsequence $S[i, i+1, \ldots, j]$.

Let $\delta(x_i, y_j) = \begin{cases} 1 & \text{if } x_i \text{ and } y_j \text{ are a complementary base pair, i.e. } (A,U) \text{ or } (C,G) \\ 0 & \text{otherwise} \end{cases}$

We get the following pseudocode:




```
for i = 1 to n+1:
    γ(i, i) = 0
    γ(i+1, i) = 0
γ(n, n) = 0
for k = 1 to n:
    for i = 1 to (n-k+1):
        j = i + k
```

$$\gamma(i,j) = \begin{cases} \gamma(i+1,j) & (2) \\ \gamma(i,j-1) & (3) \\ \gamma(i+1,j-1) + \delta(x_i, y_j) & (1) \\ \max_{i<k<j} \left(\gamma(i,k) + \gamma(k+1,j) \right) & (4) \end{cases}$$

We can define a backvector as before as well to recover the path. The final value is given by $\gamma(0,n)$.

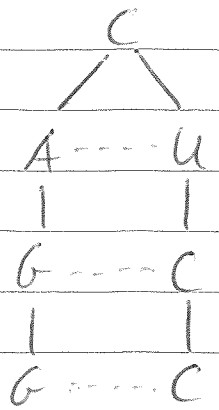
The space complexity is $O(n^2)$, since we have $n \times n$ entries, in the matrix, each requiring constant space.
The time complexity again is $O(n^2)$ times the cost of calculating a single term. Note now that we have a $\max_{i<k<j}$ term, which takes $O(n)$ time to compute.

Thus, the overall time complexity is $O(n^3)$.

Small example: folding of GGACUCC

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

Giving:



14. I Describe how BLAST works. [2005 P8 Q9(a)(i)]

Homology is the existence of shared ancestry between a pair of structures, or genes, in different species. This is exhibited by observing a strong sequence similarity between a protein and its counterpart in another species. 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 program for finding local alignments between a query sequence, and sequences in a target database. It uses heuristics to achieve a time complexity of $O(n)$, where n is the size of the database. It returns all (good) local alignments with a score of above a certain threshold. The system sacrifices sensitivity for a gain in speed, by using the fact that we are normally only interested in near-perfect alignments (99% or so, not 70%), so good alignments will have long substrings that match perfectly.

Note that BLAST is not guaranteed to find all hits (HSPs) due to this sacrifice in sensitivity.

⊛ An example is the Blosum matrix, a symmetric amino acid replacement matrix used as a scoring matrix in Blast search and phylogeny. If p_{ij} is the probability of amino acids i and j replacing each other, and p_i is the probability of finding amino acid i in any protein sequence, then $score_{ij} = \frac{1}{k} \log \frac{p_{ij}}{p_i p_j}$ (k = scaling factor)

It works roughly as follows:

- We identify all W-mers (substrings of size W) in the query.
- In a pre-processing step, we create a database of all occurrences of each possible W-mer in the target sequence; this forms the target database, which can be used again for later queries as well
- For each W-mer in the query we find its neighborhood ~~words~~: all W-mers with a similarity score of at least the neighborhood score threshold T with the W-mer in the query. (e.g. using a similarity matrix of amino acids ⊛).
  A higher T means we achieve better speeds, but lower sensitivity.
- For each of the neighborhood words s that were above the threshold, we look up in a hash table (created in the preprocessing of the database) all occurrences of this word in the database. Each such hit t gives us a seed (s, t).
- For each seed (s, t), we expand them on both sides, and see how well they match using a substitution score $\sigma(s, t)$. We keep expanding them until their substitution score falls a certain distance below the best score found for shorter extensions of this seed.
- We then report all extended segment pairs (s, t) that are locally maximum, and have a substitution score $\sigma(s, t) \geq x$, where x is the minimum match score. These are called the high-scoring segment pairs (HSP).
- Finally, we calculate the statistical significance of each HSP, and report those as well.

Gap BLAST / BLAST2 also merge seeds if they are within a distance A from each other, but the original BLAST did not allow indels (more efficient).

The BLAST algorithm has three parameters: the word size W, the word similarity threshold T, and the minimum match score X. (or neighborhood score threshold).

15. [I] Describe the output of a BLAST search. [2005 P8 Q9(a)(ii)]

For each returned HSP with score S, BLAST reports its E-value, which is the expected number of HSPs with a score of at least S.

$$E(S) = W m n e^{-\lambda S}$$

where m and n are the sizes of the query and database, respectively, and W and λ are scaling factors for the search space and scoring scheme, respectively.

i.e. the E-value corresponds to the expected number of pairwise alignments with a score of at least S. Thus, a lower E-value suggests a better alignment.

As the E-value depends on the choice of the parameters W and λ, we cannot compare E-values from different BLAST searches. We can convert our raw score S for the HSP to a bit-score S' that can be compared between different BLAST searches:

$$S' = \frac{\lambda \cdot S - \ln W}{\ln 2}$$

Note that: $E = m \cdot n \cdot 2^{-S'}$

~~Finally, the probability of finding~~

Finally, the number of HSPs (s,t) with $\sigma(s,t) \geq x$ can be described by a Poisson process. Hence the probability of finding exactly k HSPs with score $\geq S$ is given by $P(k) = \frac{E^k}{k!} e^{-E}$.

The probability of finding as many HSPs of score S and E-value E as we did is given by the P-value. ~~A lower value~~ is again a better alignment.

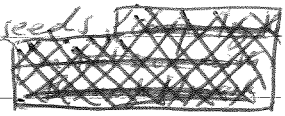
$$P = 1 - e^{-E} \text{ for one HSP (or more)}, \quad P_c = 1 - e^{-E} \sum_{i=0}^{c-1} \frac{E^i}{i!} \text{ for } c \text{ HSP's (or more)}$$

BLAST reports E-values rather than P-values as it is easier to compare e.g. an E-value of 5 and 10, than a P-value of 0.993 and 0.99995.

16. Discuss why the use of spaced seeds in sequence database search is better than the use of consecutive seeds. [2010 PYQ Q3 (a)]

The biggest problem with BLAST is its low sensitivity (and low speed). To address this, we can use spaced seeds. A seed (s,t) is a pair of sequences that have an alignment score above a certain threshold x, where s is a subsequence of the query, ~~and~~ and t of the database.

A spaced seed (s,t) is again a pair of sequences, but they are now matching exactly at certain positions, specified by a model, e.g. 111010010100110111, where "1" means "exact match" and "0" means "don't care". We then expand each spaced seed in the same way as consecutive seeds. Consecutive seeds are effectively all 1's: 11111...



Before, if we wanted to speed things up we had to use a longer seed, but that would mean losing even more sensitivity. By using spaced seeds, we increase both our sensitivity and speed!

Our score previously was simply the number of matches and mismatches; we now only count matches and mismatches where the mask is 1!

 The seeds have to be carefully designed to get good performance. Pattern Hunter (PH) was the first method that used carefully designed spaced seeds to improve the sensitivity of DNA local alignment.
PH II uses multiple optimal seeds.

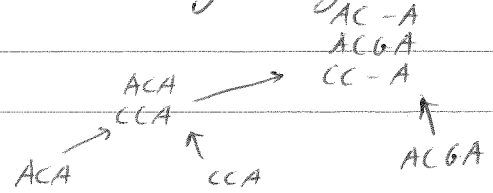
17. Why do we use dynamic programming algorithms for pairwise sequence alignment problems but not for multiple pairwise alignment? [2006 P8 Q13 (a)]

Dynamic programming algorithms work relatively well for pairwise sequence alignment problems because the space and time complexities are reasonable: $O(n \cdot m)$ time and space, and even $O(n)$ space with the Hirschberg algorithm, where n and m are the lengths of the two sequences.

When we have k different sequences of lengths $n_1, n_2, \ldots, n_k$, using dynamic programming for alignment would involve a k-dimensional matrix, and calculating each value would require $O(2^k)$ surrounding values. This gives a space complexity of $O(n_1 \cdot n_2 \cdots n_k)$, and a time complexity of $O(2^k \cdot n_1 \cdot \ldots \cdot n_k)$. The Hirschberg algorithm would only get rid of one n_i factor from the space complexity. These complexities are simply infeasible for any reasonable sequence alignment.

Note that we might still use dynamic programming algorithms to compare pairs of alignments and then combine them later; it is just infeasible to compute the alignment in one go.
For example, in CLUSTAL we compute pairwise alignments (using dynamic programming or BLAST or PH or ...) to give a similarity matrix, and then create a guide tree from this matrix (e.g. using UPGMA or Neighbor Joining). We then build up the multiple sequence alignment (MSA) using progressive alignment, which uses the guide tree to progressively (sequentially) add alignments to the growing MSA.

* e.g.



As we merge subalignments, we add all possible combinations, and have a - (don't care) where they do not match.

18. I Describe the aims of the Burrows-Wheeler transform.

The current sequencing techniques have highly parallel operations and low costs per base, but produce several millions of reads (short stretches of DNA bases - usually around 35-400 base pairs).

The Burrows-Wheeler Transform (BWT) is a text compression technique that can be used to reduce the memory requirements for sequence alignment.
It is a reversible transform that rearranges the strings so that there will be a lot of consecutive equivalent characters (e.g. bases) in a row, (e.g. tatatatata$ into attttt aaaaa$), which can then be stored more compactly (e.g. a1t5a4)

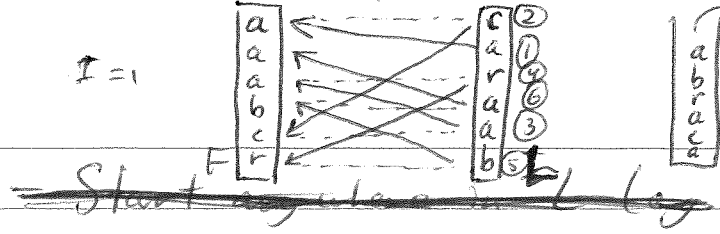
BWT(T): $O(|T|^2 \lg |T|)$
- Input string T; add EOF char $.
- Calculate all cyclic shifts of T, and sort lexographically their order
- Output L, the array consisting of the last letters in each of the cyclic shifts of T.
  Also output I, the index of the original string T in the sorted order

e.g. T = "abraca" ⟹

	0	a	a	b	r	a	c
T→	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

⟹ L = caraab, I = 1

Inverse BWT(L, I): $O(|T| \lg |T|)$
- Input L and I
- Calculate the array F, consisting of all the first letters of all cyclic shifts, by simply sorting L.
  This works because each character in F is also in L:
  if a character is the (a,t of) the ith cyclic shift of T, then that character will be the first in the (i+1)th shift of T, and therefore in F and L.



- Starting from $L[I]$: (set $i = I$)
  - → While we do not get back to $L[I]$:
    - ⇒ Add $L[i]$ to the output stack
    - ⇒ Say that this char $c = L[i]$ is the jth occurrence of c in the array L.
    - ⇒ Now let k be the index of the jth occurrence of c in the array F
    - ⇒ Our next entry then is $L[k]$; i.e. set $i = k$
- To obtain the original string, pop off the characters from the output stack, the order in which they come off is the order of the original string.

19. Describe how we can assemble a DNA sequence from multiple reads of the sequence.

Next Generation Sequencing?

We are presented with lots of small strings of DNA called reads, which are part of a bigger DNA string. These reads overlap, so we want to reconstruct the original DNA string (which the reads are from) by maximizing the overlap. As these reads may be of different lengths, we might want to split them up into K mers by taking all k-length substrings of each read.

There are several approaches to solving this problem. Firstly, we could construct a Hamiltonian graph using the reads, where each node is a read and we get a directed edge between two nodes if they have some kind of overlap (e.g. at least five bases).
Similarly, we can construct a Hamiltonian graph using the k-mers, where we get a node for each k-mer, and

a directed edge for each overlap of $(k-1)$ bases.
In both cases, we then try to look for a Hamiltonian cycle, a path travelling through every node exactly once and ending at the starting node, to reconstruct the original sequence.
Unfortunately, finding a Hamiltonian cycle is NP-complete.

A different approach is to construct a De Bruijn graph, where each directed edge is a k-mer between a node for its prefix (size $k-1$) and its suffix (size $k-1$). We reconstruct the sequence by finding a Eulerian path, a path visiting each edge exactly once. It can be shown that in our construction we can always find a Eulerian path.
The time required to find this path is roughly proportional to the number of edges in the De Bruijn graph.
The strength of this approach (and why it is not NP-complete) is due to the fact that we merge a lot of information by combining nodes with the same $(k-1)^{\text{th}}$ length substring, and that the overlaps between reads are implicit in the graph, so a lot of comparisons are not required. It does, however, mean that information is lost in this representation.

20. [What is the reason behind researching phylogeny? [2012 P7 Q3(a)(i)]]

By comparing DNA sequences from different species and building a phylogeny (tree) from these similarities, we can reconstruct the evolutionary history of different biological species. The heights in the tree where different (groups of) species are combined gives an indication of when the species diverged; i.e. the height in the tree gives a molecular clock of evolution.

21. Describe two algorithms for doing phylogeny using parsimony

Phylogeny using parsimony works by constructing a hypothesis phylogeny (tree) of the evolution, and then evaluating how good this tree is, allowing us to compute its relative quality compared to other hypothesis trees.

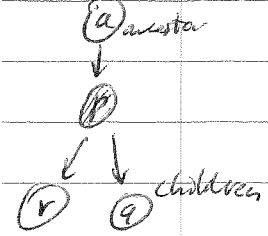
Say that we have a hypothesis tree structure, with the values of the leaf nodes (e.g. a base value; we have a (small) tree for each base in the DNA sequences) already known (i.e. these are the DNA sequences we are comparing). We now want to know the best values for the internal nodes leading up to the root node.
By then computing the (total) number of differences between parent- and child-node values, we get the parsimony score of the tree.

This score allows us to compare different hypothesis trees, and see which one is the best one.

There are two main algorithms to compute the values for the internal nodes: Fitch parsimony, and Sankoff parsimony.

The Fitch parsimony model for DNA sequences consists of two stages:

1. Bottom-up stage: determine the set of possible states for each internal node.
   → For an internal node p, if the states of its descendents q and r overlap, $S_p = S_q \cap S_r$; otherwise it is their union: $S_p = S_q \cup S_r$
2. Top-down stage: pick states for each internal node.
   → Let F_a be the final set of states for an ancestor a of an internal node p (states S_p) with children r and q (states S_r, S_q). For the root node, $F_{root} = S_{root}$.
   If the final states of a are included in S_p, $F_p = F_a$.

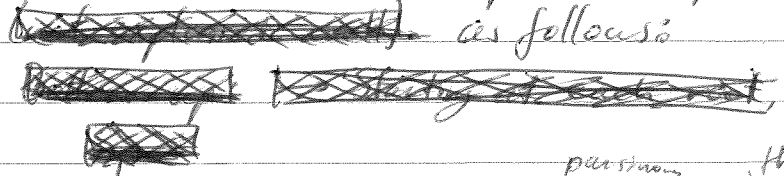


Otherwise, if $S_q \cap S_r = \emptyset$, $F_p = S_p \cup F_a$; if $S_q \cap S_r \neq \emptyset$, $F_p = ((S_q \cup S_r) \cap F_a) \cup S_p$

→ In other words, if S_p includes all possible states of its ancestor F_a, it is the same as its ancestor.

→ If there is a state in F_a that is not in S_p, and p's children r and q have no overlap ($S_q \cap S_r = \emptyset$), F_p includes all states either in F_a or S_p (recall $S_p = S_q \cup S_r$ then). If there is a state in F_a that is not in S_p, and p's children do have overlap ($S_q \cap S_r \neq \emptyset$), F_p includes all states in both F_a and (S_q or S_r). The extra S_p term (recall $S_p = S_q \cap S_r$ then) adds the overlap of its children that is not in F_a

We can then try to compute the parsimony score from the final sets. as follows:



→ Let $R_i(s)$ denote the parsimony score of node i (the subtree with i as root) if we choose state s for node i.

→ Initialize

$$R_i(s) = \begin{cases} 0 & \text{if } s = S_i, \text{ the state of this leaf node} \\ \infty & \text{otherwise} \end{cases}$$

for each leaf node i with state S_i

→ Go up into the tree (leaves to root), and compute:

$$R_p(s) = \min_{s'} \{ R_q(s') + S(s', s) \} + \min_{s''} \{ R_r(s'') + S(s'', s) \}$$

where q and r are the children of internal node p, and $S(s', s)$ is the cost of moving from state s' to s.

→ $\min_s R_{root}(s)$ then gives us the minimum parsimony score, and by then going down the tree (root to leafs), always choosing the state s minimizing $R_p(s)$, we get the best state value for each node p. (if two $R_p(s)$ have the same value, and are both minimal, we can choose either state).

Sankoff parsimony also has two stages: At each node p, we keep an array g_p with the minimum parsimony score if we chose a specific state (e.g. base value) for node p, i.e. one entry for each state

1. Bottom-up stage: compute the cheapest mutation to each of the possible states in a node p, using the parsimony scores of its children q and r, and a score matrix C_{ij} = cost of mutating from i to j.

Then

$$g_i^{(p)} = \min_j \left(C_{ij} + g_j^{(r)}\right) + \min_j \left(C_{ij} + g_j^{(q)}\right)$$

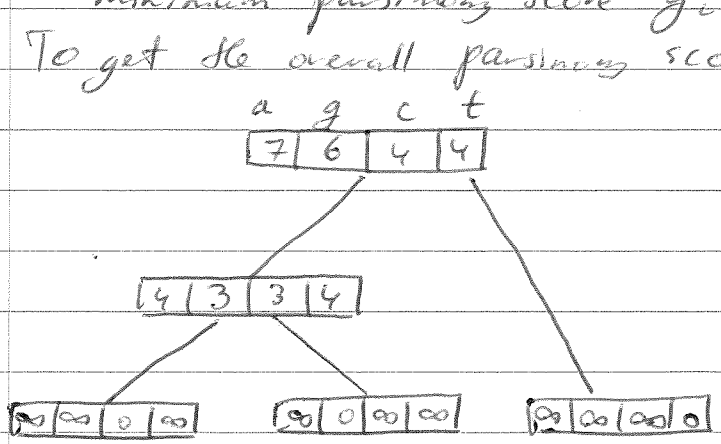
also store a backpointer with which states for q and r gave you this minimum

where we initialize the leaf nodes using:

$$g_i^{(\text{leaf})} = \begin{cases} 0 & \text{if } i = s_i, \text{ the state of the leaf node} \\ \infty & \text{otherwise} \end{cases}$$

- Top-down stage: to get the state of each internal node p, simply choose the state s_i with the minimum parsimony score $g_i^{(\text{root})}$, and follow the backpointers.

To get the overall parsimony score, we simply find $\min_i g_i^{(\text{root})}$.



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

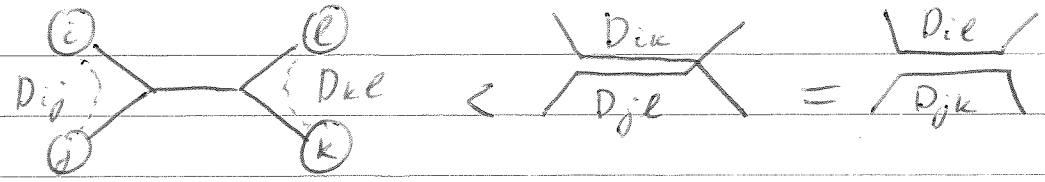
We build a tree for each character in the sequences.

Both approaches use a cost matrix C_{ij}, although Fitch parsimony only uses this cost matrix to compute the parsimony. An example of a weighted matrix is Blosum.

If we have n characters, k states (for bases, k=4), and m taxa (species' sequences we are comparing), Fitch parsimony has complexity $O(m \cdot n \cdot k)$, and Sankoff $O(m \cdot n \cdot k^2)$

22. Describe the input and the output of a distance-based algorithm. [2005 P9 Q9(a)]

Distance methods get as input a distance matrix (dissimilarity matrix = 1 − similarity), and use it to construct a tree showing the phylogeny of the species.
The distance matrix has to be <u>additive</u> (satisfy the four-points property) in order to turn 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:



$$D_{ij} + D_{kl} < D_{ik} + D_{jl} = D_{il} + D_{jk}$$

A distance matrix without the additivity property does not turn into a tree.
The tree can either be rooted (e.g. UPGMA), or unrooted (e.g. Neighbor Joining).

23. Describe the <u>UPGMA algorithm</u>. [2006 P8 Q13(d)]

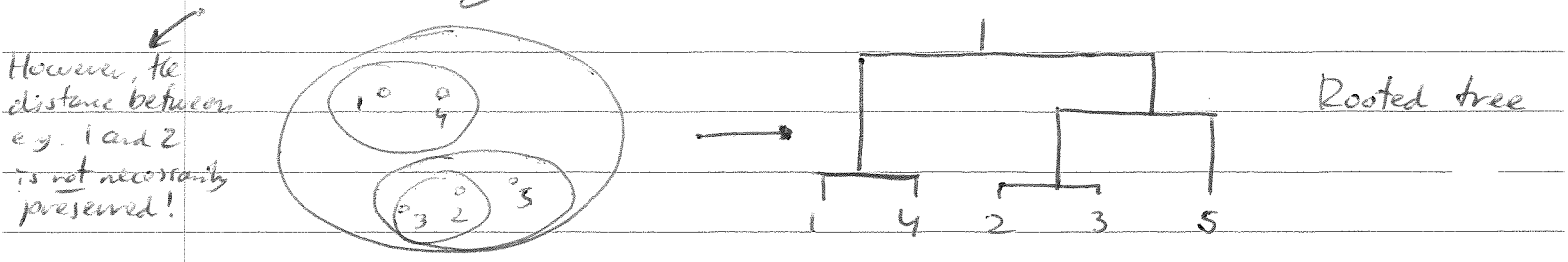
Unweighted Pair Group Method with Arithmetic mean

The UPGMA algorithm is an algorithm for constructing the phylogenetic tree of different species.
The input is a distance matrix of distances between species. We then keep iterating until we reach a single cluster, at each stage computing the average pairwise distances between each cluster's species to find the distance between two clusters, and merging the two

with the additivity property!

closest / most similar ones. We also "date" each vertex in the tree by this ~~cluster~~ distance score, which effectively gives us a molecular clock of when two species diverged in time.

A rooted tree is produced, with the ultrametric property: the distance ~~to~~ from root to each leaf is the same. This is why ~~the~~ we can use the tree as a molecular clock



Algorithm:

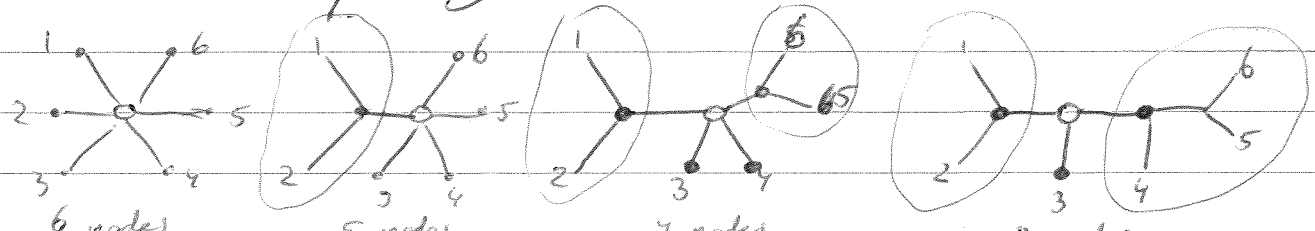
- Assign each species its own cluster C_i and leaf node in the tree
- We have a distance matrix d
- While we have more than ~~one~~ a single cluster:
  - → Determine i and j s.t. $d(C_i, C_j)$ is minimal
  - → Define a new cluster $C_k = C_i \cup C_j$ with a corresponding node at height $d(C_i, C_j)/2$ and edges (C_k, C_i), (C_k, C_j)
  - → Update distances from C_k to all other clusters using the weighted average of its pairwise members' distances
  - → Remove C_i and C_j

If we have m species, the complexity is $O(m^2)$: there are $(m-1)$ iterations, and each time we find the minimum distance (can keep track of this and update it when we merge clusters), update the distances ($O(m)$), and remove the clusters, so overall $O(m)$ cost per iteration.

24. Discuss the Neighbor-Joining algorithm, and its complexity. [2005 P9 Q9 (a)(iii)]

The NJ algorithm is an algorithm for constructing an unrooted phylogenetic tree of different species, using a distance matrix with the additivity property.

We start with a star topology of the nodes, with the center node being "symbolic" (does not belong to a species), and all other nodes belonging to a species.
We then repeatedly identify the two nodes i, j with the shortest distance in the distance matrix, and combine them into a new node u. We then update the distance matrix (cost $O(n^2)$ for n species) by computing the distance of u to the rest of the tree. We repeat this until three nodes are left.
The overall complexity is $O(n^3)$.



The distance metric used is the Q-criterion:

$$Q(i,j) = (r-2)\,d(i,j) - \sum_{k=1}^{r}\left[d(i,k) + d(j,k)\right]$$

where we sum over the r [redacted] groups of species left.
We update the distances using:

[redacted]

$$d(i,u) = \frac{1}{2}\,d(i,j) - [\text{redacted}]$$

$$- \frac{1}{2(r-2)}\left[\sum_{k=1}^{r} d(i,k) - d(j,u)\right]$$

and $d(j,u)$ similarly. Finally, we replace i and j by u in the distance matrix using

$$d(u,k) = \frac{1}{2}\left[d(i,k) - d(i,u)\right] + \frac{1}{2}\left[d(j,k) - d(j,u)\right]$$

NJ still reconstructs the correct tree when the distance matrix is perturbed by small amounts of noise. NJ is optimal regarding tolerable noise amplitude.
NJ also preserves distances between two nodes (in contrast to UPGMA, which does not).

25. I Describe the format of microarray data. [2005 P8 Q9(b)(c)]

Microarray data (collected using a DNA chip) records the activity (expression level) of genes in different cells/tissues at different times, and under different conditions (e.g. normal, cancer, after taking a drug), so that we can compare the different conditions. Expression level is estimated by measuring the amount of mRNA for that particular gene (more mRNA indicates more gene activity).
Microarray data is usually transformed into a set of large matrices, over which we can run clustering algorithms to identify changes of activity in genes and functional similarity among genes.

26. I Describe how a cluster algorithm works. [2005 P8 Q9(b)(ii)]

A cluster algorithm takes in a set of data points (e.g. microarray data) and assigns each data point to a "cluster", a set of data points that are more similar to each other than to data points in different clusters.
Algorithms include Lloyd's Algorithm, progressive greedy k-means clustering, and Manor Clustering.

There are two typical experiments we wish to analyze using clustering: differentiation (compare expression levels under different conditions), and temporal expression (explore temporal evolution of expression levels).

27. Discuss how Lloyd's algorithm, the progressive K-means algorithm, and the Manor Clustering algorithm work.

Lloyd's algorithm is a K-means algorithm. It has one parameter K, the number of clusters.
The algorithm is very simple:
- We arbitrarily assign the k cluster centers
- Then while there is a change in the cluster centers
  → Assign each data point to the closest cluster (e.g. using Euclidean distance)
  → Update the cluster centers C_i to the "center of gravity" of all data points v that belong to it:

$$C_i = \sum_{v \in C_i} \frac{v}{|C_i|}$$ where $|C_i|$ is the number of data points in cluster i

This algorithm finds a local maximum (so not necessarily the best solution). Its time complexity is $O(n \cdot k \cdot I)$, for k clusters, n data points, and I iterations until convergence

With the progressive greedy K-means algorithm, instead of moving all data points each iteration, we move the single point that gives us the greatest "gain" by moving it.
We can define the cost function for a data point p in cluster C_j as the Euclidean distance from p to the cluster center C_j.
The gain is then the difference in cost between the current cost with $p \in C_j$, and the cost of p in a different cluster, $p \in C_i$
Our algorithm then is:
- Arbitrarily select a partitioning P of the data points into k clusters $C_1, \ldots, C_K$.
- While we can make a change that reduces our total cost:
  → For each data point p in cluster C_i, calculate the gain of moving p to a different cluster C_j: $d_{ip} - d_{jp}$

→ We remember the p and C_i that gives the greatest gain
→ If this gain > 0, we move p to C_i.
Otherwise, we return P and terminate.

This algorithm tends to give a better results than Lloyd's algorithm, but as we move only one data point at a time we typically need more iterations to converge (still $O(n \cdot k \cdot I)$, but larger I).

The Markov Clustering algorithm does not require the number of expected clusters to be specified beforehand.
The idea behind MCL is:
- The input is the matrix of micro array data
- We regard this matrix as a graph, and we take a random walk on the graph.
- There will be many strong links within a cluster, and fewer between clusters, so by travelling along a random edge we are more likely to stay within the cluster than leave it.
- Thus, by analyzing where the flow tends to gather, we can identify where clusters are.
- We boost this effect by an iterative alternation of expansion and inflation steps.

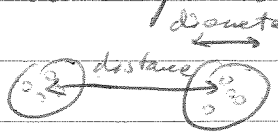
The MCL algorithm has two parameters, a power parameter e, and an inflation parameter r. In the expansion step, we take the e^{th} power of the adjacency matrix M (which we normalized at the start to make it stochastic: $M_{pq}' = \frac{M_{pq}}{\sum_i M_{iq}}$), and then in the inflation step we take the r^{th} power of each element: $M_{pq} = \frac{(M_{pq})^r}{\sum_i (M_{pq})^r}$.

The expansion parameter e is usually taken as 2, and then the inflation parameter tunes the granularity of the clusters.

Each iteration of the algorithm costs $O(n^3)$ for an $n \times n$ adjacency matrix. We normally require 10 to 100 iterations until a steady state is reached.
(expansion $O(n^3)$, inflation $O(n^2)$).
We can also prune in the MCL algorithm, setting near-zero entries to zero, creating a sparser graph and speeding up the algorithm.

28. How can you evaluate the results obtained using clustering algorithms? [2013 PG Q1(c)]

For the K-means algorithms, the quality of the cluster results could be assessed by the ratio of the distance to the nearest cluster and the cluster diameter.
We need to assess this because the algorithms forces k clusters to be created, so a cluster can be formed even when there is no similarity between clustered data points.



For the Markov Clustering algorithm, we could e.g. measure the clustering entropy:

$$S = -\frac{1}{L}\sum_{ij}\left[M_{ij}\log_2 M_{ij} + (1-M_{ij})\log_2(1-M_{ij})\right]$$

L: no. of edges

This informs us of the clustering and its stability, and we could use it to detect the best clustering obtained from different granularity parameters each time.

Or clustering cost.

$$C = \sum_{n \to c_i} (x_n - c_i)^2$$

2g. Describe how you would build a Hidden Markov Model to identify membrane segments in amino acid sequences. [2012 P7 Q3(b)(i)]

We can build a HMM that identifies genes and their parts (exons and introns) in a genome by training a model with a database of experimentally determined genes / transmembrane helices and validating the model using another database. We can then identify membrane segments by using the HMM to compute the most likely sequence of states (states here being what part of the genome we are in) using the Viterbi algorithm, and then seeing whether any states correspond to a membrane segment.

A HMM consists of a set of n hidden states π_i, a set of K possible outputs x_j, state transition probabilities $a_{ij} = \mathbb{P}(\pi_{t+1} = j \mid \pi_t = i)$, and emission probabilities $e_k(b) = \mathbb{P}(x_i = b \mid \pi_i = k)$.

Also start probabilities a_{0i} of starting in state i.

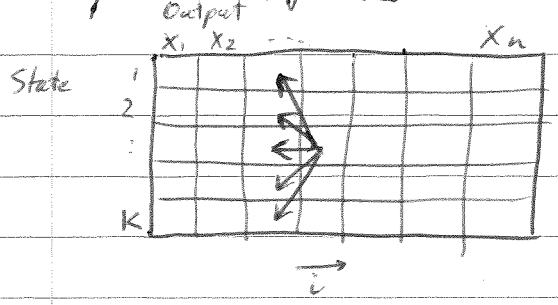
The states are hidden, but the outputs are observed.

A HMM is memoryless: $\mathbb{P}(\pi_{t+1} = k \mid \pi_1, \pi_2, \ldots, \pi_t, x_1, \ldots, x_t) = \mathbb{P}(\pi_{t+1} = k \mid \pi_t)$ at each time step t; i.e. the transition probabilities (and emission probabilities too) only depend on the current state.

The first thing we have to do when building the HMM is deciding on our set of states and outputs. From there we have three main problems for HMM's:

1. Learning - we have to learn the transition and emission probabilities that best fit the training database. (e.g. Baum-Welch algorithm)
2. Decoding - given a sequence of outputs, we use our trained model to find the most likely sequence of hidden states (e.g. Viterbi algorithm).
3. Evaluation - given a sequence of outputs, we use our trained model to find the probability of observing this sequence of outputs (e.g. forward-backward algorithm).

The Viterbi algorithm is a dynamic algorithm that computes the most likely sequence of states, given a sequence of observed outputs, in $O(nk^2)$ time and $O(nk)$ space (for k states and n observed outputs).



We build up a matrix V computing at each column the most likely previous state for each possible current state, given the likelihood of being in the previous state before, transitioning to the current state, and emitting the value we did in our state. In other words,

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

where $e_j(x_i)$ is the probability of emitting value x_i at time i, while being in state j; $V_k(i-1)$ is the k^{th} value in the $(i-1)^{th}$ column, and a_{kj} is the probability of transitioning from state k to state j.

Pseudocode:

```
V_0(0) = 1
for j = 1 to k:   V_j(0) = 0
for i = 1 to n:
    for j = 1 to k:
        V_j(i) = e_j(x_i) · max_k [a_kj · V_k(i-1)]
        Ptr_j(i) = argmax_k [a_kj · V_k(i-1)]
```

Traceback:

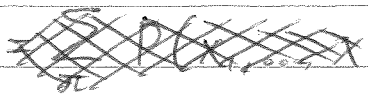
$$\pi_n = \operatorname{argmax}_k V_k(n), \quad \pi_{i-1} = Ptr_{\pi_i}(i)$$

We use an imaginary start position to incorporate the start probabilities a_{0i}.

To avoid underflow in these computations we use sums of logs rather than multiplication.

The Forward-Backward algorithm computes the probability of observing a certain sequence of outputs x. We use dynamic programming to compute:

$$P(x) = \sum_{\pi} P(x, \pi) = \sum_{\pi} P(x|\pi) P(\pi)$$



We do this in two stages:

1. Compute the forward probabilities

$$f_\ell(i) = P(x_1, \ldots, x_i, \pi_i = \ell)$$
$$= e_\ell(x_i) \sum_k f_k(i-1)\, a_{k\ell}$$

similar to Viterbi, but now $\sum$ rather than max

going forward in time.

2. Compute the backward probabilities

$$b_\ell(i) = \sum_k e_k(x_{i+1}) \cdot a_{\ell k}\, b_k(i+1) = P(x_{i+1}, \ldots, x_n | \pi_i = k)$$

going backwards in time

We can then use these values to compute

$$P(\pi_i = k, x) = \underbrace{P(x_1, \ldots, x_i, \pi_i = k)}_{\text{forward probability}} \underbrace{P(x_{i+1}, \ldots, x_n | \pi_i = k)}_{\text{backward probability}}$$

And using this compute

$$P(x) = \sum_{i,k} P(\pi_i = k, x)$$

the probability of being in a state k at time i, given the observed sequence of emissions.

Contrast this with the Viterbi algorithm: we compute the likelihood of a single state at each time $0 \le i \le n$, rather than the most likely sequence of states.

30. How would you assess the sensitivity and specificity performance of your HMM? [2012 P7 Q 3(b)(ii)]

We define the following values:

- Predicted Positive (PP) is what we predicted to occur
- Predicted Negative (PN) is what we predicted not to occur
- Actual Positive (AP) is what actually occurred
- Actual Negative (AN) is what actually did not occur

- True Positive $TP = PP \cap AP$ is what we correctly predicted to occur
- True Negative $TN = PN \cap AN$ is what we correctly predicted not to occur
- False Negative $FN = PN \cap AP$ is what we incorrectly predicted not to occur
- False Positive $FP = PP \cap AN$ is what we incorrectly predicted to occur

- Sensitivity is the probability of correctly predicting a positive (i.e. recall)

example:

$$S_p = \frac{|TP|}{|TP|+|FN|}$$

- Specificity is the probability of correctly predicting a negative

example:

$$S_n = \frac{|TN|}{|TN|+|FP|}$$

- The probability that a positive prediction is correct, (precision) is

$$P = \frac{|TP|}{|TP|+|FP|}$$

31. Explain how Gibbs sampling works.

Given a set of sequences, we want to find the motif (subsequence) shared by most or all of the sequences by finding its start index in each of the sequences.
The motif we are looking for is of a fixed width I.
At each iteration, we discard one of the start indices at random, and update it to a different index that best aligns with the other sequences' motifs.
The algorithm roughly is:
- Randomly choose starting positions $s_1, \ldots, s_t$ for each of the t sequences, and form the set of I-mers with these starting positions.
- While we make an improvement:
  - → Randomly choose one of the t sequences.
  - → Create a profile p from the other t-1 I-mers
  - → Using this profile, compute the probability that an I-mer was created by the profile, for each starting index in the chosen sequence, with the I-mer starting at that index
  - → Choose a new starting index at random using these probabilities.

32. Describe an algorithm for reconstructing a biological network.

A biological network is a group of genes in which individual genes can influence the activity of other genes, e.g. if a gene B is expressed it can cause another gene D to be expressed as well.
By performing genetic perturbations (experimentally manipulating gene activity by manipulating either a gene itself or its product) and measuring the effect on mRNA expression of all genes (giving microarray data) we can see which genes' production

is influenced. We now wish to find out which genes directly interact with others, and which interactions were indirect.

Say we have performed an experiment in which the activity of every single gene in an organism is manipulated, and we have measured the effect on mRNA expression of all other genes for each mutant.
We can then easily construct a genetic network, where nodes are genes, and we get an directed edge between genes 1 and 2 if gene 1 causes 2 to be expressed (directly or indirectly) by simply seeing whether the expression level of 2 changed when we manipulated 1.
We can store this information of (in)direct manipulation in the accessibility list $Acc(G)$, showing all genes that can be "accessed" from a given gene.

However, we really want to construct the network of direct interactions. To do this we must compute the adjacency list $Adj(G)$ of the graph, showing all genes that are directly adjacent to each gene.
We can reconstruct the adjacency list using the Wagner algorithm.
If there are no cycles, the algorithm works as follows:
- Let $Acc(G)$ be the accessibility list of an acyclic directed graph. We want to compute its most parsimonious graph G_{pars}, by constructing the adjacency list $Adj(G)$. $V(G)$ is the set of nodes in G
- $\forall i \in V(G) . \quad Adj(i) = Acc(i) \setminus \bigcup_{j \in Acc(i)} Acc(j)$

A parsimonious graph is the simplest network with a certain accessibility list $Acc(G)$. This graph only contains "direct" edges (no transitive ones), which is what we are looking for.

For graphs with cycles, all nodes in the cycle have the same accessibility list, meaning the algorithm does not work. We cannot decide a unique graph if a cycle happens anyways (cycle could go other way as well for example)
The Wagner algorithm therefore shrinks each cycle into one node, and then solves the graph as before. In other words, it is unable to resolve cyclic graphs.

Complexity:
For n nodes, with k entries in a node's accessibility list on average, and let $a = |Adj(j)|$ be the average number of nodes in the adjacency list; then the complexity is $O(n \cdot k \cdot a)$.
This is because for each of the n nodes we have to follow its k (on average) nodes j that are accessible, and then follow its accessible nodes as well. By naively visited nodes, we visit on average $|Adj(j)|$ other nodes. Thus, $O(n \cdot k \cdot a)$.

33. Discuss the utility of the Gillespie algorithm in system biology [2009 Pg Q 3(d)]

The Gillespie algorithm is a method for simulating systems, according to the populations of the different parts in the system (e.g. species, or chemical substances) and certain reactions that change the populations.
For each reaction we have a change vector, $\vec{v}_i$, stating the change in populations if the reaction takes place, and a propensity function $a_i(x)$ giving the probability of an event taking place in a short time period given the populations X (vector).
Our simulation then goes as follows:
- We start at time $t = 0$
- We pick a random amount of time τ to advance by, which we model as an exponentially distributed RV, with scale parameter the total propensity of all reactions:

$$\tau = \frac{1}{a_0(x)} \ln \frac{1}{r} \quad \text{for } a_0(x) = \sum_{j=1}^{M} a_j(x),\ r \sim U(0,1)$$

- At time $t + \tau$, our next reaction takes place.
  Choose a random reaction R_i, proportionate to its propensity function $a_i(x)$ (x = current populations), scaled by the total propensity $a_0(x)$
- Update the population: $x = x + v_i$, and update the time: $t = t + \tau$
- Repeat until t reaches some specified limit t_{max}.

The Gillespie algorithm is widely used to simulate the behaviour of a system of chemical reactions in a well stirred container.
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, so in order to garner 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.

The complexity of the Gillespie algorithm is $O(M)$, where M is the number of reactions, although t_{max} and the propensity functions also influence it greatly (since $a_0(x)$ is used as the scale parameter for the event times).

~~Discuss the assumptions of the Gillespie algorithm (one sentence)~~

This method tends to be better than for example solving a set of differential equations corresponding to the reactions, because in reality there is a lot of noise, which the Gillespie algorithm can model, but differential equations can only study the steady state.

34. [Discuss the assumptions of the Gillespie algorithm. [2010 P3 Q23(d)]

The most important assumption is that our system is well-stirred (not compartmentalised). Furthermore, we assume that our system is confined to a constant volume and at a constant temperature, and that we know all the molecular reactions that take place.