String Distance

$distance_{hamming}(x, y) = \sum_{i=1}^{n} cond(x_i = y_i, 0, 1)$.

$distance_{levenshtein}(x, y)$ is the minimum number of insertions, deletions and substitutions to transform x into y.

Given two strings an alignment is an assignment of gaps to positions in the strings so as to line up each letter in one sequence with either a corresponding letter or a gap in the other sequence. Paths in the edit graph are in bijection with alignments: $\rightarrow$ and $\downarrow$ are deletions and $\searrow$ are (mis)matches.

Needleman-Wunsch

Given match weight m, mismatch penalty s and gap penalty d:

1. $F(0,0) = 0$
2. $F(0,j) = -jd$
3. $F(i,0) = -id$
4. For each $i = 1 \dots m, j = 1 \dots n$:
   (a) $(F(i,j), Ptr(i,j)) = max \begin{cases} (F(i-1,j) - d, UP) \\ (F(i,j-1) - d, LEFT) \\ (F(i-1,j-1) + cond(x_i = y_i, m, -s), DIAG) \end{cases}$
5. $F_{OPT} = F(m,n)$

Space and time complexity of $\Theta(mn)$.

Needleman-Wunsch Overlap Detection

Does not penalize gaps at the start or the end. As Needleman-Wunsch but initialize with:

1. $F(i,0) = 0$
2. $F(0,j) = 0$

Upon termination, the optimal score is $F_{OPT} = max \begin{cases} max_i F(i,n) \\ max_j F(m,j) \end{cases}$.

Smith-Waterman

Ignores badly aligning regions, modified from Needleman-Wunsch:

1. $F(0,j) = 0$
2. $F(i,0) = 0$
3. For each $i = 1 \dots m, j = 1 \dots n$:
   (a) $(F(i,j), Ptr(i,j)) = max \begin{cases} (0, NIL) \\ (F(i-1,j) - d, UP) \\ (F(i,j-1) - d, LEFT) \\ (F(i-1,j-1) + cond(x_i = y_i, \end{cases}$
4. $F_{OPT} = max_{i,j} F(i,j)$

Needleman-Wunsch With Affine Gap

This uses a gap penalisation function $\gamma(n) = d + (n-1)e$ where n is the length of the gap. This reduces the penalty for long gaps. $F(i,j)$ represents the score of at alignment if x_i aligns to y_i. (gap not open) $G(i,j)$ represents the score if x_i or y_i aligns to a gap (gap open).

1. $F(0,0) = 0$
2. $F(0,j) = d + (j-1)e$
3. $F(i,0) = d + (i-1)e$
4. For each $i = 1 \dots m, j = 1 \dots n$:
   (a) $F(i,j) = max \begin{cases} F(i-1,j-1) + cond(x_i = y_i, m, -s) \\ G(i-1,j-1) + cond(x_i = y_i, m, -s) \end{cases}$
   (b) $G(i,j) = max \begin{cases} F(i-1,j) - d \\ F(i,j-1) - d \\ G(i,j-1) - e \\ G(i-1,j) - e \end{cases}$
5. $F_{OPT} = F(m,n)$

Banded

If we assume that that the optimal alignment has few gaps ($< k(max(n,m))$) then the path of the alignment will be close to diagonal. Space and time complexity of $\Theta(nk(max(n,m))) \ll \Theta(nm)$. This is done by searching just a diagonal band of the matrix.

1. $F(0,0) = 0$
2. $F(0,j) = -jd$
3. $F(i,0) = -id$
4. For each $i = 1 \dots m, j = max(1, i-k) \dots min(n, i+x)$:
   (a) $(F(i,j), Ptr(i,j)) = max \begin{cases} (F(i-1,j) - d, UP) \\ (F(i,j-1) - d, LEFT) \\ (F(i-1,j-1) + cond(x_i = y_i, \end{cases}$
5. $F_{OPT} = F(m,n)$

Space Efficient Needleman-Wunsch

Idea is to perform divide and conquer on the sequences to be aligned. To find the midpoint around which to partition the problem, we use a space efficient version of Needleman-Wunsch that computes alignment column-wise and only keeps around the previous two columns in memory hence using only $\Theta(min(m,n))$ space. This version does not allow us to reconstruct the optimal sequence but lets us compute the score of two sequences $d(x,y)$. So:

$F(i,j) = d(x[1:i], y[1:j])$

$B(i,j) = d(x[i+1:n], y[j+1,m])$

$F(i,j)+B(i,j)$ is the score of the best alignment through (i,j). This is computed in linear space and the best i found (since $F(m,n) = max_k(F(\frac{m}{2},k) + B(\frac{m}{2},k))$) in time $\Theta(mn)$. The divide and conquer algorithm can proceed in time $\Theta(mn)$ as well.

Four Russians Heuristic

The n by n edit graph is partitioned into blocks of size t by t. A *block alignment* of two sequences is one in which alignment occurs through the insertion/deletion/alignment of blocks. A *block path* is one that traverses every block through its corners.

1. Precompute all pairwise alignments of sequences of length t with each other and store this in a lookup table. The table will be of size $4^t 4^t = 4^{2t}$.
2. Lookup the appropriate alignment scores for the problem in the table via binary search: this takes time $\Theta(\frac{n^2}{t^2} log_2(4^{2t}))$.
3. Use dynamic programming to find the optimal block alignment score based on metric: $s(i,j) = max \begin{cases} s(i-1,j) - d_{block} \\ s(i,j-1) - d_{block} \\ s(i-1,j-1) + m_{block}(i,j) \end{cases}$ where $m_{block}(i,j)$ is the result of the appropriate pairwise alignment. This takes time $\Theta(\frac{n^2}{t^2})$.

Let $t = \frac{log_2(n)}{4}$ and running time is bounded by $\Theta(\frac{n^2}{t^2} log_2(4^{\frac{log_2(n)}{2}})) = \Theta\left(\frac{n^2}{\left(\frac{log_2(n)}{4}\right)^2} log_2(n)\right) = \theta\left(\frac{n^2}{log_2(n)}\right)$

BLAST

Heuristic method to allow rapid sequence comparison of a query against a database. Smith-Waterman uses too much time and space.

1. Compile a list of words scoring at least a threshold T with the query word according to a substitution matrix δ.
2. Scan the database for entries containing any word from this list: call these the seeds.
3. The seeds are extended in both directions in an attempt to boost the alignment score, without considering insertions or deletions. Stop extending when score falls below a cutoff.
4. Gapped alignment between the query and database sequence using a variant of Smith-Waterman to find statistically significant matches.

Expected value $E(S)$ is the number of alignments with scores greater than or equal to S that are expected to occur by chance in a search. $E(S) = Kmne^{-\lambda S}$ where K is constant, λ scales for the matrix δ, m is query size and n is database size. Bit scores normalize the differences of database size and scoring matrices: $S' = \frac{(\lambda S - lnK)}{ln2}$, so $E(S') = mn2^{-S'}$.

Progressive Multiple Alignment

For k sequences there are $2^k - 1$ ways to extend a sequence, so dynamic programming takes time $\Theta(2^k n^k)$ and space $\Theta(n^k)$ for sequences of length n. This is prohibitively large, so instead:

1. Compute pairwise sequence Hamming distance normalized by sequence length.
2. Use this distance matrix to create a guide tree in which closer sequences are aligned first.

Inter-pair alignment is done by trying all different possibilities at a position (including gaps) and picking the best or just assume such things do not match.

Parsimony Trees

Rooted trees are *bifurcating* if all non-leaves have degree 3.

If the length of the path from the root to any leaf is the same, it is said to be ultrametric. The biological interpretation is that it obeys a molecular clock.

If the length of a tree edge is the hamming distance between its endpoints we can define *parsimony score* as the sum of edge weights. We seek trees with the lowest possible parsimony score, confusingly called the maximum parsimony tree.

Columns in sets of aligned sequences are *parsimony-informative* if they contain at least two nucleotides and at least two of those nucleotides occur at least twice!

Fitch's Algorithm

The small parsimony problem is to obtain a labelling of internal tree nodes that minimises hamming-distance parsimony with respect to the pre-labelled leaf nodes. This algorithm solves the problem for trees with leaves of length 1. To generalize it, run it on each sequence index separately and concatenate the resulting trees.

1. For leaf nodes i, set $R_i = \{label_i\}$
2. From leaves to the root consider internal node i with children j and k. $R_i = \begin{cases} R_j \cup R_k & R_j \cap R_k = \emptyset \\ R_j \cap R_k & \text{otherwise} \end{cases}$
3. From root to leaves consider node j with parent i. $r_i = \begin{cases} r_i & r_i \in R_j \\ \text{some element of } R_j & \text{if root node or } r_i \notin R_j \end{cases}$

If there are k possible values for each sequence position and n nodes, complexity is $\Theta(kn)$.

Sankoff's Algorithm

The weighted small parsimony problem generalizes the small parsimony problem beyond hamming distance to include a matrix δ indicating the cost of pairwise mutations. This algorithm solves the problem for trees with leaves of length 1, but can be generalized as before. Quantity $s_t(v)$ denotes the minimum parsimony score of the sub-tree rooted at v if v has character t.

1. For leaf nodes i, set $s_t(i) = cond(label_i = t, 0, \infty)$
2. From leaves to the root consider internal node i with children j and k. $s_t(i) = min_i(s_i(j) + \delta_{i,t}) + min_i(s_i(k) + \delta_{i,t})$
3. The root l gets $r_l = argmin_i(s_i(l))$ and sub-trees recursively obtain the characters that lead to that minimum score

If there are k possible values for each sequence position and n nodes, complexity is $\Theta(kn)$.

Generalized Parsimony Method

Given n aligned sequences containing c parsimony-informative columns:

1. $S(g) = 0$
2. For each distinct graph g with n nodes:
   (a) For each of the c columns:
      i. Label the graph leaves with the letters of the column
      ii. Use Fitch's or Sankoff's algorithm to compute a minimum parsimony tree and add its parsimony score to the total score for this graph $S(g)$
3. The best graph is $argmax_g(S(g))$

Assume there are k possible values for a sequence position and g suitable graphs with n nodes. Time complexity is $\Theta(gckn)$.

Ultrametrics

An ultrametric space has a distance measure d that satisfies $d(x, y) \geq 0$, $d(x, y) = 0 \iff x = y$, $d(x, y) = d(y, x)$ and crucially $d(x, z) \leq max(d(x, y), d(y, z))$.

UPGMA

Unweighted Pair Group Method with Arithmetic Mean. This computes the distance between clusters using average pairwise distance. Its weakness is that it produces an ultrametric tree that implicitly assumes a constant molecular clock (differences between two sequences are proportional to the time to the last common ancestor).

1. For each sequence x_i, create $C_i = \{x_i\}$
2. Compute pairwise cluster distance matrix D based on sequence distance
3. While more than one cluster remains:
   (a) Find two clusters C_i,C_j with minimum distance d_{ij}
   (b) Add a vertex to the tree connecting those of C_i and C_j with height $\frac{d_{ij}}{2}$
   (c) Replace them with $C_k = C_i \cup C_j$ such that $D_{kl} = \frac{D_{il}|C_i| + D_{jl}|C_j|}{|C_i| + |C_j|}$: this ensures the new cluster distance is the unweighted arithmetic mean of the distances to the constituent sequences.

3-Way Tree Reconstruction

We wish to determine inter-sequence tree distances that match the observed inter-sequence string distances. For 3 leaved trees (i,j,k with common node c) this can be done exactly using the string distance matrix D:

$$d_{ic} = \frac{D_{ij} + D_{ik} - D_{jk}}{2}$$

$$d_{jc} = \frac{D_{ij} + D_{jk} - D_{ik}}{2}$$

$$d_{kc} = \frac{D_{ki} + D_{kj} - D_{ij}}{2}$$

Fitch Margoliash

Computes an unrooted tree with sequences at its leaves that satisfies distance constraints, but does not assume a molecular clock:

1. Compute pairwise cluster distance matrix D based on sequence distance
2. While more than three sequences remain:
   (a) Find the closest sequences i and j according to D
   (b) Clump all other sequences together to form K such that $D_{aK} = \frac{\sum_{k \in K} D_{ak}}{|K|}$
   (c) Solve this 3-sequence problem using exact reconstruction where the distance to K is the distance from a new internal node to the sequences of K
   (d) Replace i and j in D with node (i, j) where $D_{(i,j)k} = \frac{D_{ik}+D_{jk}}{2}$
3. Output the graph built up

Neighbour Joining

Similar to cluster analysis but also relaxes the assumption of equal rates of molecular change among branches.

1. Initialize graph to a star topology with all r sequences hanging off a common node
2. Compute pairwise cluster distance matrix D based on pairwise sequence distance
3. If pairs remain to be joined:
   (a) Based on the current distance matrix compute $Q_{ij} = (r-2)D_{ij} - \sum_{k=1}^{r} D_{ik} - \sum_{k=1}^{r} D_{jk}$
   (b) Find the pair of sequences i, j that is closest according to Q and create a node e in the graph joining these two
   (c) Calculate the distance of sequences i and j to this new node: $D_{ie} = \frac{1}{2}D_{ij} + \frac{1}{2(r-2)}\left(\sum_{k=1}^{r} D_{ik} - \sum_{k=1}^{r} D_{jk}\right)$ and symmetrically for D_{je}
   (d) Calculate the distance of all other sequences to the new node: $D_{ek} = \frac{1}{2}(D_{ik} - D_{ie}) + \frac{1}{2}(D_{jk} - D_{je})$
   (e) Recursively join neighbours after removing i and j from the distance matrix as they are covered by the consolidated node e
4. Otherwise output just the two pairs joined by an edge in the obvious way

To make this fast the sub-terms $R_i = \sum_{k=1}^{r} D_{ik}$ can be computed and cached at the start of an iteration. This leads to a time complexity of $\Theta(r^3)$.

Tree Improvements

Can greedily attempt to improve the initial estimate for a genetic tree by branch swapping and testing if we have created a more parsimonious tree. This can be done by tree bisection and reconnection:

1. Bisect a tree along a branch to obtain two sub-trees
2. Connect the sub-trees by joining a pair of branches, one per sub-tree

May also use sub-tree pruning and regrafting or nearest neighbour interchanges.

Hypothesis Testing

Bootstrapping techniques can be used to check the reliability of an inferred tree. A tree is inferred from the sample data and instances of identical trees are looked for to support their correctness. Data generation techniques include sampling with replacement (choose some columns and repeat them a number of times and sampling without replacements (take a subset of a permutation of the input sample columns).

Multiple possible internally labelled trees can be selected from based on the maximum likelihood criterion. Likelihood L is defined such that $L \propto P(\text{data on tree}|\text{hypothesis})$. Tree likelihood is computed recursively:

1. For nodes i above leaf nodes labelled with symbols j and k with distances t_1 and t_2 respectively, $L(i) = P_{label(i),j}(t_1)P_{label(i),k}(t_2)$
2. For other nodes i above child nodes j and k with distances t_1 and t_2 respectively, $L(i) = \sum_{y \in Labels} \sum_{z \in Labels} L(relabel(j, y))L(relabel(k, z))P_{label(i)y}(t_1)P$

We can also evaluate the likelihood of an unlabelled tree with respect to some sequence of length n:

1. Assign the symbols of the sequence to the leaves of the tree.

2. For each sequence index i:
   (a) $L_{unlabelled}(i) = 0$
   (b) For each assignment of letters to internal node labellings of the tree:
      i. Increment $L_{unlabelled}(i)$ by the likelihood of this labelled tree
3. Final likelihood $L_{unlabelled} = \prod_{i=1}^{n} L_{unlabelled}(i)$ by assumption of symbol independence

It may be useful to evaluate the probabilities in logarithmic space for practical reasons as they are typically quite small.

Markov Models

Some Markov processes may not allow us to observe the states transitioned through directly. We can infer information from these based on their emissions. The state space is given by the set K and the emission set by D. Transition probabilities are stored in the matrix A of size $|K| \times |K|$ and emission probabilities in E of size $|K| \times |D|$.

A *parse* is a sequence of states π recovered from a sequence of emissions x. Parse likelihood is given by $P(\pi, x) = \prod_{i=1}^{N} a_{\pi_{i-1}\pi_i} e_{\pi_i x_i}$.

Viterbi Algorithm

To solve the decoding problem we need to obtain, given knowledge of the HMM, the sequence of states π that maximises the probability of generating the observed emissions x of length n. The essential trick is use dynamic programming with $V_k(i)$ storing the probability of the most likely sequence of states ending at state $\pi_i = k$:

1. $V_0(0) = 1$ (where 0 is an imaginary first position)
2. $\forall k > 0. V_k(0) = 0$
3. $Ptr_j(i) = Nil$
4. For each index i from 1 to n and state j:
   (a) $V_j(i) = e_j(x_i) max_k(a_{kj} V_k(i-1))$
   (b) $Ptr_j(i) = argmax_k(a_{kj} V_k(i-1))$
5. Now the optimal probability is $max_k(V_k(n))$ and the chain of states that give that sequence begins in $argmax_k(V_k(n))$

Time complexity is $\Theta(|K|^2 n)$ and space complexity is $\Theta(|K|n)$. Again in practice you will need to work in logarithmic space to prevent small values experiencing loss of precision.

Forward Algorithm

To solve the evaluation problem we need to obtain, given knowledge of the HMM, the likelihood of the observed sequence of emissions x of length n. Again we use dynamic programming where $f_k(i)$ stores the probability of the emission sequence such that $\pi_i = k$:

1. $f_0(0) = 1$ (where 0 is an imaginary first position)
2. $\forall k > 0. f_k(0) = 0$
3. For each index i from 1 to n and state j:
   (a) $f_j(i) = e_{j,x_i} \sum_k f_k(i-1) a_{k,j}$
4. Now the final probability is $\sum_k f_k(n) a_{k,0}$ where $a_{k,0}$is the probability that the terminating state is k.

Time complexity is $\Theta(|K|^2 n)$ and space complexity is $\Theta(|K|n)$. Again in practice you will work in logarithmic space.

K-Means Clustering (Lloyd's Algorithm)

1. Choose k random center points in n dimensional cluster space
2. Until data points stop moving between clusters:
   (a) Put all points into the cluster they are nearest according to some distance metric
   (b) Calculate new cluster centre points from clustered points X by $CentrePoint(X)_{k \in \mathbb{Z}_n} = \frac{\sum_{x \in X} x_k}{|X|}$ and use them as new centre point seeds

This has a time complexity of only $O(m)$ for m points on average but has a worst case of $2^{\Omega(\sqrt{m})}$. You need to choose k and it gives non-deterministic heuristic results. You can measure cluster quality with cluster diameter vs. inter-cluster distance, distance between the cluster members and the cluster centre or by repeating runs and checking how often cluster reoccur.

Hierarchical Clustering

1. Put every point to be clustered in its own cluster
2. While more than k clusters remain, merge those clusters with the smallest inter-cluster distance

Again you need to know k. Time complexity is $\Theta(m)$ for m points but there is no statistical foundation to this algorithm.

Markov Clustering

As input this algorithm takes a m vertex problem represented as a $n \times n$ adjacency matrix A and normalises each column to obtain the stochastic matrix M. Now:

1. While the network entropy is changing:
   (a) $M := M^k$ (expansion) and $m_{ij} := (m_{ij})^r$ (inflation): this simulates a random walk

Network Reconstruction

For acyclic digraphs, there is exactly one minimal graph that satisfies an accessibility list Acc: this is called the most parsimonious network compatible with the list. It is defined by $V(G_{parsimonous}) = \{v|\exists i.v \in Acc(i)\}$ and $\forall i \in V(G_{parsimonous}).Adj(i) = Acc(i) \setminus \cup_{j \in Acc(i)} Acc(j)$. This can be defined as an algorithm in the obvious way with time complexity $\Theta(nE_j(|Adj(j)|)E_j(|Acc(j)|))$ for graphs of n nodes.

In order to extend this to cyclic graphs, we first apply a strongly connected components analysis to identify cycles and cycle-like things and replace them with single aggregated nodes. An algorithm based on the corollary that two nodes i and j are in the same strong component iff $i \in Acc(j)$ and $j \in Acc(i)$ is as follows:

1. For each node i in $V(G)$:
   (a) If $Component(i)$ has been defined continue
   (b) Add a node x with $Acc_{G*}(x) = \emptyset$ to $G*$
   (c) For each node j in $\{x\} \cup \{j|\forall j \in Acc(i).i \in Acc(j)\}$:
      i. Set $Component(j) = x$
2. For each node i in $V(G*)$:
   (a) Set $Acc_{G*}(i) = \emptyset$
   (b) For each node j in $Acc(i)$:
      i. If $Component(i) \neq Component(j) \wedge Component(j) \notin Acc_{G*}(Component(i))$:
         A. Add $Component(j)$ to $Acc_{G*}(Component(i))$

The second step just recovers the accessibility list of $G*$from that of G. The first step is responsible for assigning the actual strongly connected component nodes.

Gillespie Algorithm

This is a stochastic simulation algorithm for reactions. Given a list of reaction rates $a_i(x)$ in terms of a vector x that stores the quantity of reactant molecules in the environment:

1. Set x to the initial reactant quantities.
2. Repeatedly:
   (a) Find $a_0 = \sum_j a_j(x)$
   (b) Pick a random quantity δt from $Exp(a_0)$
   (c) Pick a reaction number r from $Uniform(0, 1)$
   (d) Pick reaction j such that $\sum_{i=1}^{j-1} a_i < ra_0 \leq \sum_{i=1}^{j} a_i$
   (e) Execute reaction j, updating x and increasing simulated time by δt

Stochastic simulation gives you more detailed information about how a system evolves than analytic methods, and is often more tractable.