1 General Formula

2 Sequence Alignment

2.1 DNA Errors

Substitution: Flip one base for another (Mismatch Error)

Indel: Insert/Delete base in sequence (Gap Error)

Transition: Replace a Purine with a purine; replace pyrimidine with pyrimidine.

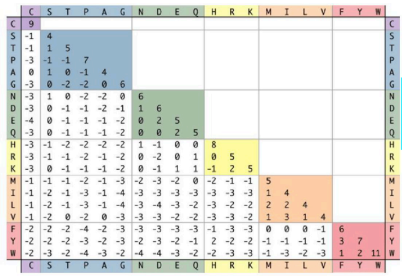
Transversion: Convert Purine to pyrimidine, vice versa. More expensive than transitions

Transposon: Delete/Insert whole region from sequence and insert to another

2.2 Similarity Heuristics

Hamming Distance: Number of different positions within string
Ex: ATCTCA vs ATCACA is 1

Normalized Hamming Distance: $\frac{\text{Hamming Distance}}{\text{len(Seq)}}$ (Assume sequences same length)
Edit Distance: Minimum-weight series of edit operations transforming a into b *Ex*: TGCAGT TCACAGT (Switch G to C, remove A, d= 2)
Substitution Matrix: Cost of switching one base to another (since transitions and transversions cost different amounts)
BLOSUM62: Most classical amino acid substitution matrix

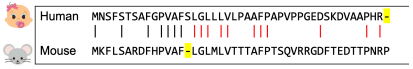


Levenshtein Distance: Number of edits (indels + substitutions)

Dynamic Programming: Reduce problems into subproblems to save results and reduce repeated computations.
- Requires optimal substructure (optimal solution contains optimal solutions to subproblems)
- Overlapping subproblems: Solution contains set of distinct subproblems repeated many times
- No Greedy Choice: Requires tracing back through many choices

2.3 Pairwise Sequence Alignment

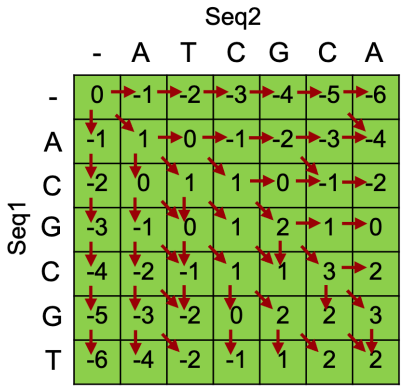
Pairwise Sequence Alignment: Arrange two sequences so similar regions are aligned to each other.

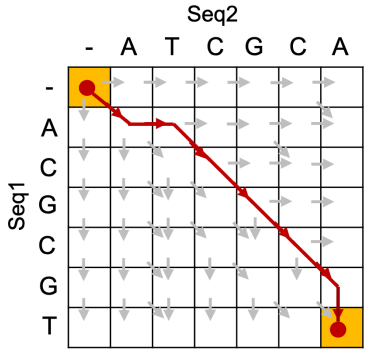


2.3.1 Global Alignment

Solves Global alignment (Overlap two entire sequences completely)
Time & Space Complexity: $O(mn)$
Needleman-Wunsch:

1. **Input**: Sequences seq1, seq2, match score (α), gap penalty (μ), mismatch penalty (σ)
2. **Output**: Alignment score, backtrack array
3. Initialize alignment + backtrack array of size seq1, seq2
4. Initialize top row and left column base cases
5. For every cell in matrix
6. if matching bases then add match score to score of left diagonal
7. if top/left cell + gap penalty is lowest score, add it
8. if left diagonal + mismatch penalty is lowest score, add it
9. put direction taken in backtrack array
10. output alignment score at bottom right cell, and use backtrack array to derive original alignment





Input: Sequences $seq1$ and $seq2$, match score(α), mismatch penalty (μ), gap penalty (σ)
Output: Alignment score and backtracking matrix

$n \leftarrow$ length of $seq1$
$m \leftarrow$ length of $seq2$
Initialize score matrix S with dimensions $(n+1) \times (m+1)$
Initialize backtrack matrix B with dimensions $(n+1) \times (m+1)$

for $i \leftarrow 0$ to n **do**
  $S[i][0] \leftarrow i \times \sigma$, $B[i][0] \leftarrow$ 'UP'
end for
for $j \leftarrow 0$ to m **do**
  $S[0][j] \leftarrow j \times \sigma$, $B[0][j] \leftarrow$ 'LEFT'
end for

for $i \leftarrow 1$ to m **do**
  for $j \leftarrow 1$ to n **do**
    match $\leftarrow S[i-1][j-1] + (\alpha$ if $seq1[i-1] = seq2[j-1]$ else $\mu)$
    delete $\leftarrow S[i-1][j] + \sigma$
    insert $\leftarrow S[i][j-1] + \sigma$
    $S[i][j] \leftarrow \max(\texttt{match}, \texttt{delete}, \texttt{insert})$
    if $S[i][j] =$ match **then**
      $B[i][j] \leftarrow$ 'DIAGONAL'
    else if $S[i][j] =$ delete **then**
      $B[i][j] \leftarrow$ 'UP'
    else
      $B[i][j] \leftarrow$ 'LEFT'
    end if
  end for
end for
return $S[n][m]$, B

Input: Sequences $seq1$ and $seq2$, Backtracking matrix B
Output: Aligned sequences and alignment score
function RecoverAlignment(i, j)
  if $i \leftarrow 0$ and $j \leftarrow 0$ **then**
    return '', ''
  end if
  if $B[i][j] \leftarrow$ 'DIAGONAL' **then**
    (aligned_seq1, aligned_seq2) $\leftarrow$ RecoverAlignment$(i-1, j-1)$
    return aligned_seq1 + $seq1[i-1]$, aligned_seq2 + $seq2[j-1]$
  else if $B[i][j] \leftarrow$ 'UP' **then**
    (aligned_seq1, aligned_seq2) $\leftarrow$ RecoverAlignment$(i-1, j)$
    return aligned_seq1 + $seq1[i-1]$, aligned_seq2+'-'
  else if $B[i][j] \leftarrow$ 'LEFT' **then**
    (aligned_seq1, aligned_seq2) $\leftarrow$ RecoverAlignment$(i, j-1)$
    return aligned_seq1+ '-', aligned_seq2 + $seq2[j-1]$
  end if
end function

(aligned_seq1, aligned_seq2) $\leftarrow$ RecoverAlignment(m, n)
return aligned_seq1, aligned_seq2, $S[m][n]$

2.3.1.1 NCBI BLASTn Scoring scheme

Because whole regions get inserted/deleted, having n bases indeled is less costly than n indels. So we make gap opening a high cost, and continuing gap lesser cost.

Consant: Gap Penalty = Constant for all size n
Linear: Gap Penalty = Gap Length x Gap opening
Affine: Gap Penalty = Gap Opening + Gap Extension x (Gap Length - 1)
Convex: Gap Penalty <= Affine Gap Penalty

NCBI Blastn Scoring Matrix:

Event	Score
Match	+2
Mismatch	–3
Gap opening	–5
Gap extension	–2

2.3.2 Local Alignment

Smith-Waterson: Same as **Needleman-Wunsch** but we can start and end anywhere, and the goal is to find the lowest cost path overall. 0 is minimum, so initialize every cell as minimum of a continuation of path and 0, and the highest score is the ending of the alignment. To retrieve, we start at highest cell and go back until we reach 0.

2.3.3 Comparison

Global Alignment: Needleman–Wunsch Algorithm

- **Initialization**: Top-left, i.e. $M(0,0) = 0$
- **Iteration**:

$$M(i,j) = \max \begin{cases} M(i-1,j) + \text{Gap Pen} \\ M(i,j-1) + \text{Gap Pen} \\ M(i-1,j-1) + S(i,j) \end{cases}$$

- **Termination**: Bottom-right
- **Traceback**: Start from bottom-right and end at top-left

Local Alignment: Smith–Waterman Algorithm

- **Initialization**: Top row/left column ($M(0,j) = 0$ and $M(i,0) = 0$)
- **Iteration**:

$$M(i,j) = \max \begin{cases} 0 \\ M(i-1,j) + \text{Gap Pen} \\ M(i,j-1) + \text{Gap Pen} \\ M(i-1,j-1) + S(i,j) \end{cases}$$

- **Termination**: Anywhere
- **Traceback**: Start from the highest-scored cell and end when $M(i,j)$ is 0.

2.3.4 Database Search

k-mer: Continuous block of k characters (def. 11 for nucl and 3 for prot)

High-Scoring Segment Pair: Alignment found by BLAST word match

BLAST:

1. Build sequence into **k-mers** (continuous block of k characters)
2. For each k-mer build index hash table of occurrences of all k-mers in target string
3. In construction of k-mers, allow some deviation along threshold of difference
4. Given a query, break it into k-mers and look up each k-mer as seed for sequence alignment
5. For each seed, extend seq until alignment falls below threshold

2.3.4.1 Variations

Gapped BLAST: Extensions are in BLAST are allowed to use gaps. (So take gaps and continue aligning after the gaps)
Two-hit Seeding: You can only extend a sequence if there is another sequence nearby it.
- Finding two-length words is more likely than a full word, so it's more sensitive.

2.3.4.2 Scoring

Raw Score: Alignment score (S) of string

P value: Probability of alignment score $\geq S$ given random strings

E value: Expected number of alignments with score $\geq S$ given random strings

2.3.4.3 Alignment Scoring

- Used to compare alignments of same query-reference pair
- Used to compare alignments of diff query-reference pair
  - Determine whether (Q1, R1) or (Q2, R2) is more likely a homology
- Used to indicate confidence of alignment
  - Indicates whether alignment (Q1, R1) is actually homologous

2.3.4.3.1 Comparing Alignment of same query-reference pair

Define match (p), substitution (q), indel (r) probability.
Probability is $p^m q^s r^i$, log-prob = $m \log(p) + s \log(q) + i(\log)r$
m, s, i are number of matches, substitutions, indels respectively.

Problems:
- Longer spotty seq's are penalized more than shorter perfect seq's
  - TGCAT_AG_GAT is very close to CCGTA_AG_AT, whereas TGCAGTAGGAT is scored worse than CCCACGAG-AT
- Repeated information is penalized (even though it's functionally the same)
  - TCAGT aligning to TCGGT is better than TCAGTTCAGT aligning to TCGGTTCGGT

2.3.4.3.2 Comparing Alignment of diff query-reference pair

Homology Model: Assume alignment reflects evolution
Random Model: Assume alignment spurred randomly

2.3.4.3.2.1 Homology Model

Probability: $\Pr(x, y \mid H)$
Assume p_{ab} is probability evolution gave char a in x and char b in y $P(x, y \mid H) = \prod_{i=1}^{n} p_{x_i y_i}$

2.3.4.3.2.2 Random Model

Probability: $\Pr(x, y \mid R)$
Assume p_a is probability to randomly get char a $P(x, y \mid R) = \prod_{i=1}^{n} p_{x_i} \prod_{j=1}^{n} p_{y_j}$

2.3.4.3.2.3 Likelihood Ratio

Likelihood Ratio: $P = \frac{\Pr(x,y \mid H)}{\Pr(x,y \mid R)} = \sum_i \log\left(\frac{p_{x_i y_i}}{p_{x_i} p_{y_i}}\right)$
Alignment score: $\log(P)$

2.3.4.3.2.4 Substitution Matrix

A matrix used to contain the unique probabilities of switching out one character for another.
BLOSUM62 is empirically found values on actual probabilities of switching out amino acids.

To calculate $\frac{p_{x_i y_i}}{p_{x_i} p_{y_i}}$ take a toy database of values, and calculate probability of getting pair (x, y) over all possible pairs in sequences (Number of pairs (x, y) over number of all pairs) and then divide that by probability of getting x and y individually (number of $\frac{x}{y}$ over all base pairs in list). Then substitution matrix only takes log values, so take the log.

How is the substitution matrix calculated?

- A toy database consisting conserved regions aligned into blocks:

```
AVQRLPECVAKPLWNVSNDLGLKPVLTYGDVCLTNCR
ACDTIPESVAAPLLKVSEALGLPPLATYAGLVLWNFC
PAEVLPRNLALPFVEVSRNLGLPPILVHSDLVLTNWT
```

  - Summary: 3 rows, 37 columns, $3 \times 37 = 111$ amino acids, $\binom{3}{2} \times 37 = 111$ pairs
  - For example: $s(I, L) = \log\left(\frac{p_{IL}}{p_I p_L}\right) = \log p_{IL} - \log p_I - \log p_L = \log \frac{3}{111} - \log \frac{2}{111} - \log \frac{21}{111}$
  - In BLOSUM matrices these values are rounded to the nearest integer

2.3.4.3.2.5 Derivation of BLOSUM62

- Some protein families (like hemoglobin) are well-studied and will be overrepresented in our databases, so to create a

matrix that properly reflects priors, we calculate similariy scores between sequences, and those with similarity scores greater than 62% are grouped together. The total weight of a group is 1, so if we have x, x', y where x, x' pass similarity threshold, but y doesn't, then x, x' each have weight 0.5 but y has weight 1. This is useful for creating a more representative protein database.

We can use substitution matrix to simplify calculating values $\sum_{x_i,y_i \in \text{ Sequences}} s(x_i, y_i) = \log(P)$

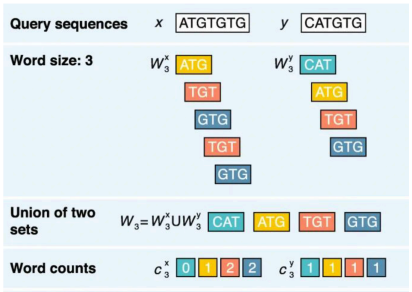
2.3.5 FASTA Format

> Title (Usually accession ID, Product, Organism) Sequence

- Sequences are standard IUPAC codes
- Lowercase allowed but mapped to uppercase
- Numbers not allowed in sequence

2.3.5.1 Alignment-Free Sequence Comparison

Given 2 sequences, chop them up into kmers. Then create a union between these two kmer lists. Then, create a word count of each kmer into a vector of the combined kmer list. Then take Euclidean distance between the two for a score.



2.3.5.2 Open Vocabulary Problem (Byte-Pair encoding)

To increase ability to scan sequence, take index of k-mers, sorted by frequency, and add that as a new "character". Iteratively, we reduce the number of tokens to scan.

2.4 Multi-sequence Alignment

Align k sequences

2.4.1 Sum-of-Pairs

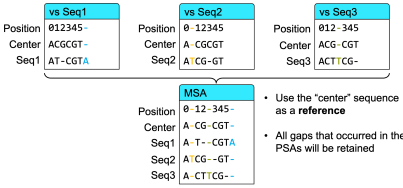
Do pairwise sequence alignment for every pair of sequences, and take the sum for a global alignment score, then optimize over that MSA for the alignment. Takes $O(n^2k^2)$ time, because there are $\binom{k}{2} = \frac{1}{2}k(k-1)$ pairs of alignments

2.4.2 Centre-Star Method

Choose one sequence as center sequence by doing Pairwise Sequence Alignment on all pairs and find one sequence that has maximum of all alignment scores to be the center. This takes $O(k^2n^2)$ time.

Then combine all the sequences by determining the gaps in each sequence at which position, and then insert them in every sequence. Ex: AC-GA, TC-AG, A-TCG => A-C–GA, T-C–AG, A-T–CG

This doesn't really concern alignment between non-centre sequences.



2.4.3 Progressive Alignment

Greedy algorithm that is a heuristic that doesn't visit all solutions but still gives good answer.

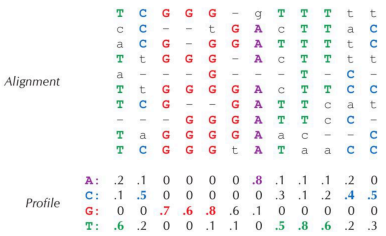
Do pairwise alignment on all sequence pairs, $O(k^2n^2)$. Then build distance matrix among sequences via some heuristic. Then build a guided tree by some clustering algorithm and then select some starting sequence according to some optimal criteria (e.g. sequence most similar to rest), and align between node pairs, merging recursively for $k-1$ alignments.

2.4.3.1 Distance Matrix

Matrix with zero on diagonal, and non-negative values, symmetric, and satisfies triangle inequality. The values at coordinate i, j are just distances between sequence i and sequence j. Distance matrix is built from some heuristic: Hamming dist, Levenshtein dist, Substitution matrix-based dist.

2.4.4 Aligning new sequence

Then, if we have a new sequence to add, we gather our alignment of all the sequences, and capture the frequency of each possible character at each position for each character. Then the new "sequence" is the greatest frequency at each position, which you align to.



2.4.5 Applications of MSA

- Motif Finding: Detect conserved regions of a protein family as important motifs
- Structure Prediction: Alignment to protein family help to predict protein structure better
- Phylogeny: Phylogeny helps better examine evolution history among organisms

3 Phylogeny

MRCA: Most Recent Common Ancestor (Root of Tree)

Internal Nodes: Ancient hypothetical species
Leaf Nodes: Extant species

Neighbour: Two species that are most closely related on tree

Clade: Group of organisms with common ancestor

Outgroup: Set of organisms that are not in the "ingroup", and are distant in the tree.

Homolog: Descendant of common ancestor

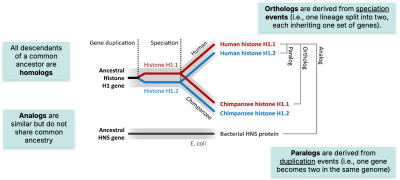
Analog: Similar but don't share ancestry

Ortholog: Speciation event, where common ancestors break into two species

Paralog: One gene breaks into two separate versions in same genome

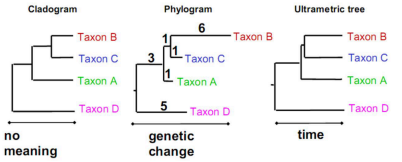
Phylogeny: Construction of evolutionary tree by evolutionary relationship

Morphology: Construction of evolutionary tree by physical featuers



3 problems: **Topology**, **Root**, **Branch Lengths**

Cladogram: Tree where branch length has no meaning
Phylogram: Tree where branch length conveys genetic change
Ultrametric tree: Tree where branch length conveys time



Hierarchical Clustering: Arrange sequences based on pairwise distance in hierarchical manner

Well-behaved tree: All nodes have equal distance from root to leaves (ultrametric), the distance is defined as "time". Also, the three-point condition is met, where for all distances, x, y, z, $d(x, y) \leq \max(d(x, z), d(z, y))$

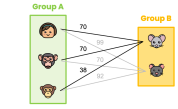
3.1 Distance-based methods

3.1.1 UPGMA

UPGMA: Unweighted Pair Group Method with Arithmetic Mean

Linkage Criterion: Distance metric between two groups (not just sequences, but 2 separate groups of sequences)

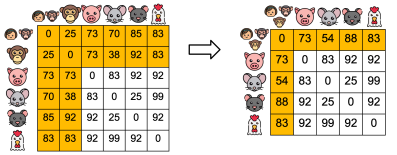
UPGMA (unweighted pair group method with arithmetic mean)



UPGMA: The average pairwise distance

➢ UPGMA = $\frac{\text{Sum of pairwise distances}}{\text{Number of pairs}}$
= (70 + 70 + 38 + 99 + 70 + 92) / 6 = 73

	0	14	25	73	70	99	83
	14	0	25	73	70	70	83
	25	25	0	73	38	92	83
	73	73	73	0	83	92	92
	70	70	38	83	0	25	99
	99	70	92	92	25	0	92
	83	83	83	92	99	92	0



	0	25	73	70	85	83
	25	0	73	38	92	83
	73	73	0	83	92	92
	70	38	83	0	25	99
	85	92	92	25	0	92
	83	83	92	99	92	0

	0	73	54	88	83
	73	0	83	92	92
	54	83	0	25	99
	88	92	25	0	92
	83	92	99	92	0

Note that distance from node to leaf is half of distance value in the matrix.

Given distance matrix, find closest pair of species, then combine them to a "new node" and recalculate the values for it and the remaining sequences by UPGMA = $\frac{\text{Sum of pairwise distances}}{\text{Number of pairs}}$ and continue until you're done.

This best describes topology and distance of species.

Ultrametric trees are represented by symmetric 0-diagonal matrices, where indices represent divergence events. Given ultrametric matrix, an ultrametric tree for that matrix exists if:

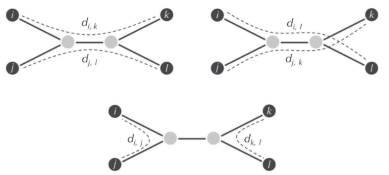
- There are n leaves, one for each row and column of matrix
- Each internal node is labeled by a time in D and has exactly two-children
- Along any path from root to a leaf, the divergence times at the internal nodes strictly decrease.
- For any triplet of nodes, the distances $D_{i,j}, D_{j,k}, D_{i,k}$ are either equal, or two are equal and the remaining one is smaller. (By 4-point theorem)

3.1.2 Neighbour Joining

Additive Matrix: There exists an additive tree fitting the matrix

Additive Tree: Given symmetric $n \times n$ 0-diagonal matrix, for any neighbouring leaves i, j, with parent m, the distance from m to any other leaf k is $d_{k,m} = \frac{d_{i,k}+d_{j,k}-d_{j,i}}{2}$

Four-Point Theorem: Distance matrix is additive iff two of the sums are equal and third is less than or equal to other sums:



3.1.2.1 Procedure

To avoid the previous limitations, the trick is to subtract the averaged distances to all other leaves to compensates for long edges. We define can define a new matrix such that:

$D_{ij} = d_{ij} - (r_i + r_j), \quad r_i = \frac{1}{|L|-2}\sum_{k \in L} d_{ik},$

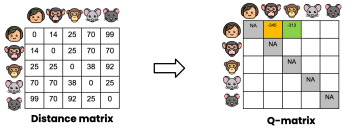
Algorithm: Neighbour-joining

Initialisation:
Define T to be the set of leaf nodes, one for each given sequence, and put $L = T$.
Iteration:
Pick a pair i, j in L for which D_{ij}, defined by (7.4), is minimal.
Define a new node k and set $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$, for all m in L.
Add k to T with edges of lengths $d_{ik} = \frac{1}{2}(d_{ij} + r_i - r_j), d_{jk} = d_{ij} - d_{ik}$, joining k to i and j, respectively.
Remove i and j from L and add k.
Termination:
When L consists of two leaves i and j add the remaining edge between i and j, with length d_{ij}.

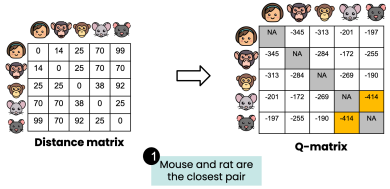
Procedure:

1. Create auxiliary Q-matrix where $Q(i,j) = (n-2)d_{i,j} - \sum_{k \in L} d_{i,k} - \sum_{k \in L} d_{j,k}$.
2. Merge closest pair wrt Q-matrix (lowest number in matrix)
3. Estimate branch length between chosen node and new node. $l(i,u) = \frac{1}{2}d(i,j) + \frac{1}{2(n-2)}\left[\sum_{k=1}^{n} d(i,k) - \sum_{k=1}^{n} d(j,k)\right]$. Note taht $l(i,u) + l(j,u) = d(i,j)$ Sometimes these branch lengths can be negative, so we can add a fixed constant to all branch lengths for non-negativity.
4. Update unrooted tree with new parent node between neighbours, and have respective lengths between them.
5. Update distance matrix so $d(u,k) = \frac{1}{2}[d(i,k) + d(j,k) - d(i,j)]$

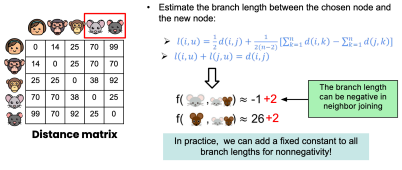
Step 1: Build a Q-matrix to **choose the closest pair to merge**

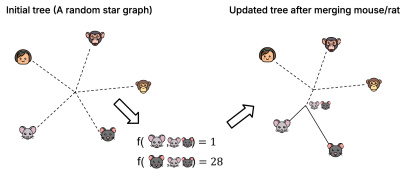


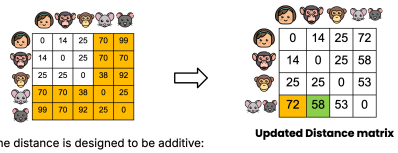
- The Q-metric considers the distance relative to other species
  - ➢ $Q(i,j) = (n-2)d(i,j) - \sum_{k=1}^{n} d(i,k) - \sum_{k=1}^{n} d(j,k)$
  - ➢ $= (5-2) \times 14 - (14+25+70+99) - (14+25+70+70) = -345$
  - ➢ $= (5-2) \times 25 - (14+25+70+99) - (25+25+38+92) = -313$



- Create a new internal node u:
- Estimate the branch length between the chosen node and the new node:
  - ➢ $l(i,u) = \frac{1}{2}d(i,j) + \frac{1}{2(n-2)}\left[\sum_{k=1}^{n} d(i,k) - \sum_{k=1}^{n} d(j,k)\right]$
  - ➢ $l(i,u) + l(j,u) = d(i,j)$







The distance is designed to be additive:

- $d(u,k) = \frac{1}{2}[d(i,k) + d(j,k) - d(i,j)]$
- $= \frac{1}{2}[70 + 99 - 25] = 72$
- $= \frac{1}{2}[70 + 70 - 25] = 58$

3.1.3 Jukes-Cantor

Jukes-Cantor is a tool to recompute distances based on back-mutations for more accurate phylogenetic trees.

Assumptions:
1. All nucleotide bases occur with equal probability
2. Each base has equal probability of mutating into any other with $r = 0.25$
3. Substitutions occur independently at each site over time

Let $p_{A(t)}, p_{C(t)}, p_{G(t)}, p_{T(t)}$ be probabilities that a given site contains nucleotide A, C, G, T at time t.

Since we assume all nucleotides are equally likely, $p_{X(t)}$ is probability that a site is $X(C, G, T)$ at time t

$\sum_{\delta \in A,C,G,T} p_\delta(t) = 1$ and if a nucleotide is A at time $t = 0$, then $p_{A(0)} = 1, p_{C(0)} = p_{G(0)} = ... = 0$

Define total mutation velocity away from A as α, then mutaiton velocity from A to C is $\frac{\alpha}{3}$, same with all other nucleotides.

For infinitesimal time step dt, $p_{A(t+dt)} =$ probability of staying A $= p_{A(t)} \cdot -p_{A(t)}\alpha dt + \left(p_{C(t)} + p_{G(t)} + p_{T(t)}\right) \cdot \left(\frac{\alpha}{3}\right) dt$

Similarly, $p_{X(t+dt)} =$ probability of becoming C, G, or T $= p_{X(t)} - p_{X(t)}\alpha dt + p_{A(t)}\left(\frac{\alpha}{3}\right) dt + 2p_{X(t)}\left(\frac{\alpha}{3}\right) dt$

Taking the limit as $dt \to 0$, $\frac{dp_A}{dt} = -\frac{4\alpha}{3} p_A + \frac{\alpha}{3}$ and $\frac{dp_X}{dt} = \frac{\alpha}{3} p_A - \frac{\alpha}{3} p_X$

Then we solve to get $p_{A(t)} = \frac{1}{4} + \frac{3}{4} e^{-\frac{4\alpha}{3} t}$ and $p_{X(t)} = \frac{1}{4} - \frac{1}{4} e^{-\frac{4\alpha}{3} t}$

Thus, the expected number of substitutions per site is: $d = -\frac{3}{4} \ln\left(1 - \frac{4}{3} p\right)$ which is our new distance function. $p = 3P_x$ is the observed fraction of nucleotide differences between sequences, and d is assumed to be αt to agree with p

3.2 Character-based Method

3.2.1 Maximum Parsimony

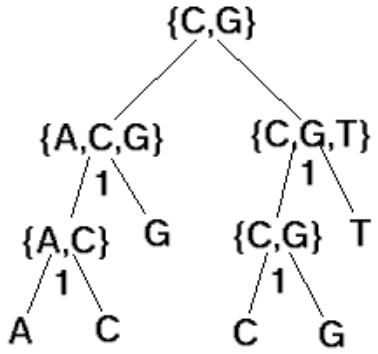
Score trees based on the minimum number of substitutions required to convey the found tree.

Parsimony: Simplest explanation is usually best.

Pseudocode: Given tree and alignment column u, label interal nodes to minimize substitutions. **Initialization**: Set $C = 0$ and root node $k = 2N - 1$ **Iteration**: If k is a leaf, set $R_k = \{x_{k(u)}\}$ If k is an internal node with children i, j: If $R_i \cap R_j \neq \emptyset$, set $R_k = R_i \cap R_j$ Else $R_k = R_i \cup R_j$ and increment C

To score a tree using maximum parsimony we do the following:
1. Assign nucleotide to root node:
   - Choose arbitrarily from R_{2N-1}
2. For each internal node k, assign nucleotides recursively
   - if parent k has state r and $r \in R_i$, assign r to descendant i
   - Otherwise choose arbitrarily from R_j

Note that $x_{k(u)}$ means the u'th base pair for node k. Then, for internal nodes, you take the intersection as the list of possible values that wouldn't require a substitution. If there is nothing, then join them, so that you only require one substitution.



Note this doesn't build a tree, but scores it.

3.2.2 Maximum Likelihood

The likelihood of a tree T with branch lengths t and given sequence alignment is $L(T, t) = P(x_1, ..., x_n \mid T, t)$

We want to find the most likely tree given sequence alignments. Note, $L(x_1, x_2) = \sum_X \pi_X P(x_1 \mid X, t_1) P(x_2 \mid X, t_2)$

π_X is stationary probability of nucleotide X, X is possible ancestral states at root. $P(x_i \mid X, t_i)$ is probability of transition from X to x_i in time t_i

$L_{k(X)}$ means likelihood of subtree rooted at k, given it has state X $P(X \to Y, t)$ is probability that state X at a node changes to Y over time t π_x is stationary probability of nucleotide X at the root.

Pseudocode: **Initialization**: At each leaf node, define $L_{i(X)} = 1$ if $X = x_i$, 0 otherwise. **Recursion**: For each internal node with children i, j $L_{k(X)} = \sum_{Y,Z} P(X \to Y, t_i) L_{i(Y)} \cdot P(X \to Z, t_j) L_{j(Z)}$ **Final Step**: $L(T) = \sum_X \pi_X L_{\text{root}}(X)$

3.2.3 Bayesian Inference

Like maximum likelihood, based on probabilities, but tree topology is not single tree, but probability distribution of all trees. Uses Bayes' theorem to calculate posterior probability of trees based on prior probability of trees and observed data. Samples from probability distribution and updates model states iteratively

3.2.4 Tree Space Search

Stepwise Addition: Start from minimum tree (3 taxa) and add taxa, then optimize topology.
Use starting tree: A tree with all taxa but is less accurate. Either use a cheap method for construction or randomly create.

3.3 Summary

Methods	Optimality Criterion
Distance Methods	Distance
Maximum Parimsony	Parimsony
Maximum Likelihood	Likelihood

Note top-to-bottom is increasing accuracy and decreasing efficiency.

3.3.1 Newick Tree Format:

Newick Tree format is a text file like this:

```
(((monkey:0.01572,chimp:0.01038):0.27040,chicken:0.37215):0.12011,
(pig:0.08412,(rat:0.03239,mouse:0.03430):0.19210):0.06343,human:0.03265);
```

- `()` -- group (hierarchy)
- `,` -- siblings
- `:` -- branch length (optional)
- `;` -- end of tree

- **For example:** `((A),B),C);`

4 Genome Mapping

4.1 Sequencing

We get sequencing from either Sanger sequencing, Next Generation Sequencing, to produce DNA fragments.

Long (>10000 bp) Reads (Pacbio/Nanopore) typically have a high error rate, and short reads (Illumina) have a bigger computational task, but higher accuracy.

4.2 Assembly

The goal is to derive the original string from a collation of fragments. The original string is first copied multiple times, and then fragmented, so there's no guarantee that the fragments are one after another.

Coverage: Number of fragments overlapping for a specific position of the original sequence.

Note: If there is a lot of overlap between the end of one fragment and start of another, it is likely that they were overlapping in the actual genome.

4.2.1 Long Read

1. Get Reads
2. Build overlap graph
3. Bundle stretches of the overlap graph into **contigs**
4. Pick most likely nucleotide sequence for each contig
5. Output Contigs

Contig: Set of DNA segments that overlap in a way that provides contiguous representation of genomic region.

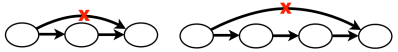
k-mer composition: Collection of all k-mer substrings of the Text, including repeats.

k-mer Prefix/Suffix: First/Last $k-1$ nucleotides of k-mer

Then connect all k-mers of the text in a graph, where for given (Text) nodes with Suffix(Text) = Prefix(Text') should have an edge between them.

Then, to solve this problem, we want a Hamiltonian path (going through every node) to solve, so we can use heuristics to solve.

For instance, remove transitively-inferrible edges, edges skipping one or two nodes.



Then, create a contig for all non-branching stretches.

Then, derive a consensus by taking all reads tha tmake up a contig and line them up and take consensus (i.e. through majority vote).

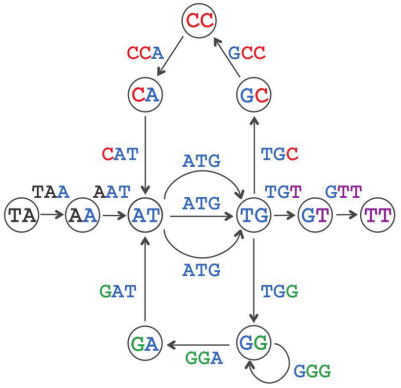
4.2.2 Short Read Assembly

1. Error Correction
2. Graph Construction
3. Graph Cleaning
4. Contig Assembly
5. Scaffolding
6. Gap Filling

Error Correction means preprocessing reads to find errors and only accepting exact matches and try to replace rare k-mers with common k-mers.

4.2.2.1 De Brujin

Graph Consruction: Given collection of k-mers P, define nodes as all unique $k-1$-mers that are prefixes or suffixes of k-mers in P and for each k-mer in P create a directed edge from its prefix to its suffix.



Euler's Theorem: Every balanced, strongly connected directed graph has a cycle that visits every edge exactly once.

Form a cycle by randomly walking in a graph, and not revisiting edges. While unexplored edges remain, select a node, newStart, in Cycle with unexplored edges and form a new cycle by traversing Cycle starting at newStart, then randomly walk to include new unexplored edges.

Note that you can turn an Eulerian path into an Eulerian cycle by adding an edge between two unbalanced nodes making it a balanced graph.

Graph Cleaning: Remove "sink" nodes in De Brujin graph and combine divergence structures that converge to a single node later through the graph.

Contig Assembly: Use Euler algorithm to identify strongly connected regions in the genome and use that path to determine the full contig.

4.2.3 Computational Problems in Sequencing Data

- Read Mapping/Alignments: Map/align reads/fragments back to known genome
- Variant Calling: Detect positions varying from reference population
- Genome Assembly: Reconstruct full chromosome from short/long sequencing reads/fragments

Seed-and-extend: To find target regions on reference genome that are at most k mutations between read and target build alignment from seed regions instead of global/local alignment.

4.3 Indexing Data Structures

4.3.1 K-mer index

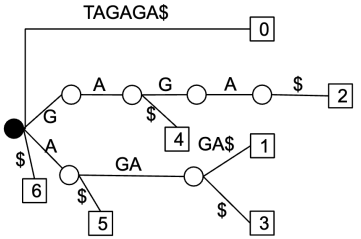
Generate list of words of length k in genome string. For each kmer build index table with all occurrences in the reference, and for each kmer of query, find its hits in the index table.

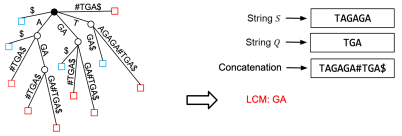
4.3.2 Suffix Tree

Suffix: Substring S_i of s starting at position i.

- Start with full string S and empty root.
- Add each suffix successively and label leaf with position.
- Use existing edges where possible.
- Merge edges that have no branches and concatenate labels for better space efficiency.

Pattern Matching: To see if query string Q is substring of S, start at root and traverse tree according to query.
Longest Common Substring: Build substring for concatenated string $S\#Q\$$ and label leaves depending on whether suffix belongs to S or Q. Common substrings, as internal nodes, are prefixes of two suffixes from both S and Q, so find node with longest path to the root.





4.3.3 Suffix Array

Sorted list of all suffixes of string S.

- Made by generating list of suffixes, sorting them in lexicographical order. $A[k]$ is start position of k-th least-sorted suffix.

```
def pattern_matching_with_suffix_array(S, Q, suffix_array):
    suffixes = [S[i:] for i in suffix_array]

    left, right = 0, len(suffixes)
    # First Binary Search for L_Q (First occurrence)
    while left < right:
        mid = (left + right) // 2
        if suffixes[mid] < Q:  # Move right if suffix is smaller
            left = mid + 1
        else:
            right = mid  # Move left otherwise
    first = left
    right = len(suffixes

    # Second Binary Search for U_Q (Last occurrence)
    while left < right:
        mid = (left + right) // 2
        if suffixes[mid].startswith(Q):  # If Q is a prefix, move right
            left = mid + 1
        else:  # Otherwise, move left
            right = mid
    last = right-1  # Adjust to return last valid match
    if first > last:
        return f'"{Q}" does not appear in S'
    else:
        return (first, last)  # Now properly inclusive
```

4.3.4 Burrows-Wheeler Transformation

Lossless string compression algorithm, made by generating all rotations of string S, sorting rotations lexicographically, and keeping only last column as output. Can be further reduced by run-length encoding.

Note that we rotate to the left, so TAGAGA$ becomes AGAGA$T

Three ways of inverting:
Sort and add: Given Iteration i sort lexicographically, then add original BWT to beginning. Continue until you reach same length as input string, and take string with $ at the end. Each iteration takes $O(n \log n)$, so total complexity is $O(n^2 \log n)$

4.3.4.1 LF-Mapping

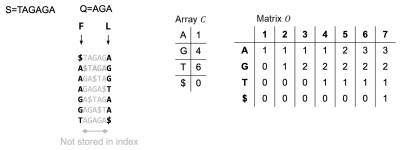
First-last property: k-th occurrence of a symbol in First Column and k-th occurrence of symbol in Last Column correspond to the same position of this symbol in Text.

Last column is the BWT, first column is the sorted characters of BWT.

4.3.4.2 FM-Index Pattern Matching

Combines BWT for compression and suffix array for efficient search.

C array has $C[c]$ = total number of occurrences of characters $< c$ in F (sorted characters) O matrix has $O[c, k]$ = number of times c occurs in $L[1 : k]$ (Start from index 1)



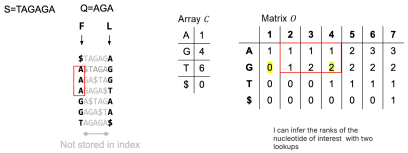
Array C

A	1
G	4
T	6
$	0

Matrix O

	1	2	3	4	5	6	7
A	1	1	1	1	2	3	3
G	0	1	2	2	2	2	2
T	0	0	0	1	1	1	1
$	0	0	0	0	0	0	1

To pattern match, do the following:



Array C

A	1
G	4
T	6
$	0

Matrix O

	1	2	3	4	5	6	7
A	1	1	1	1	2	3	3
G	0	1	2	2	2	2	2
T	0	0	0	1	1	1	1
$	0	0	0	0	0	0	1

I can infer the ranks of the nucleotide of interest with two lookups

4.3.5 Comparison

	kmer index	suffix tree	suffix array	BWT & FM Index
Space	$O(R)$	$O(R)$ with pointers	$O(R)$	$O(R)$
Search Time	$O(q)$	$O(q)$	$O(q \log R)$	$O(q)$