Bioinformatics

1 Question 1

We first discuss the Baum-Welch algorithm in general and then our implementation choices later.

1.1 Baum-Welch

A hidden Markov model (HMM) is made up of a Markov chain where the states X cannot be directly observed (we describe them as hidden states). However, we are able to infer information about this Markov chain indirectly, as at every time step a symbol Y_t (from the set of symbols Y) is emitted with a probability that depends only upon the hidden state at that time. To describe this property formally we say

$$P(Y_t|X_1 = x_1, .., X_t = t) = P(Y_t|X_t = x_t)$$

Furthermore, the hidden state X_{t+1} at $t+1$, is only dependent on the previous hidden state. Formally we say

$$P(X_{t+1} = x|X_1 = x_1, .., X_t) = P(X_{t+1} = x|X_n = x_n)$$

. This is the Markov property.

In term of notation we let X_t be a discrete hidden random variable and the observation variable is denoted by Y_t. We note that this overview has called on multiple sources which are all acknowledges in the references section. We follow the notation used in [1]

A HMM is parameterised by the emission probabilities, the Markov chain transition probabilities and the initial probabilities distribution. A transition probability (denoted by a_{ij}) refers to the probability of being in some hidden state i at time t and transitioning to some hidden state j at time $t+1$ and these, as discussed, obey the Markov property. If we have N hidden states, we represent this parameter by the matrix A of size $N \times N$ such that $a_{ij} = P(X_{t+1} = j|X_t = i)$. Note that we cannot assume that $a_{ij} = a_{ji}$ and we allow transitions to the same states (a_{ii} is not necessarily zero). The initial state distribution is the probability, over all N hidden states, of being in that hidden states at time $t = 1$. Formally we say $p(X_1 = i) = a_{s_0 i}$ where s_0 is the artificial state. We denote these probabilities as the N dimensional vector π. Finally, we denote the emission probabilities by a matrix B such that $b_j(y_t) = P(Y_t = y_i|X_t = j)$. This describes the probability of seeing the observation symbol y_i at time t given that the hidden state, at time t, is j.

We denote a HMM θ with parameters A, B, π by $\theta = (A, B, \pi)$. Often, the parameters of a HMM are unknown. If a HMM, with unknown parameters θ, produces a sequence of observations of length T that depend on hidden quantities, we can use the expectation-maximisation (EM) algorithm to estimate these parameters. An EM algorithm tries to estimate these parameters by finding a set of parameters $\theta*$ that maximises the likelihood of see the observations given a set of parameters. We can formally say that the EM attempts to find a set of parameters $\theta*$ which maximise $p(O|\theta) = \sum_x p(O, x|\theta)$ where O is the observed sequence of length N and x is the hidden sequence of states that produced O. The Baum-Welch algorithm is the standard EM algorithm. It computes a set of parameter values that are are guaranteed to be, at least, locally optimal.

At a high level, we start with initial (random) probability estimates for the parameters, we then compute expectations of how frequently each transition and emission is used and then we re-estimate the probabilities based on those expectations. We then repeat this procedure until convergence.

In more detail, we first initialise the parameters A, B and π randomly. We then execute the forward and backward procedure.

In the forward procedure, we calculate the probability $\alpha_i(t)$ of being in state i and seeing the sequence of observations $y_1, ..y_(t)$ given our model parameters. We define α recursively:

$$\alpha_i(1) = \pi_i b_i(y_1)$$

$$\alpha_i(t+1) = b_i(y_{t+1}) \sum_{j=1}^{N} \alpha_j(t) a_{ji}$$

Where $\alpha_i(t)$ is the previous forward path probability at the time step t. a_{ji} is the transition probability from previous state j to the current state i. $b_i(y_t)$ is the probability of seeing the observation symbol y_t given the current state i. The forwards procedure consists of calculating α for all states up to the end time T.

In the backwards procedure, we calculate the probability $\beta_i(t)$ of being in state i at time t and then seeing the sequence of observation $y_{t+1}, .., y_T$. Formally we say

$$\beta_i(t) = P(y_{t+1}, .., y_T | X_t = i, \theta)$$

. We define β using backward recursion as shown:

$$\beta_i(T) = 1$$

$$\beta_i(t) = \sum_{j=1}^{N} \beta_j(t+1) a_{ij} b_j(y_{t+1})$$

The backwards procedure consists of calculating $\beta_i(t)$ for all states from $t = 1$.

To implement these procedures, our implementation exploits the recursive nature of the definitions (noting that we first check if the value has been computed before recursing).

We now use these procedures to complete the expectation step. To achieve this, we first compute (γ) which is the probability of being in state i at time t given both the model parameters and the sequence of observations $y_1, .., y_T$. Which we can formally write by

$$\gamma_i(t) = P(X_t = i | Y, \theta)$$

We compute γ for all N hidden states from $t = 1$ to $t = T$. To calculate this, we employ Bayes theorem giving:

$$\gamma_i(t) = \frac{P(X_t = i, Y | \theta)}{P(Y | \theta} = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}$$

.

We also need to calculate the probability of being in state i at time t transiting to be in state j at time $t + 1$, given both the model and sequence of observations. Formally we can write this as:

$$\xi_{ij}(t) = P(X_t = i, X_{t+1} = j | Y, \theta)$$

We can again calculate this by using Bayes theorem to give:

$$\xi_{ij}(t) = \frac{P(X_t = i, X_{t+1} = j, Y, \theta)}{P(Y | \theta}) = \frac{\alpha_i(t) a_{ij} \beta_j(t+1) b_j(y_{t+1})}{\sum_{k=1}^{N} \sum_{w=1}^{N} \alpha_k(t) a_{kw} \beta_w(t+1) b_w(y_{t+1})}$$

. We compute ξ for all state pairs from time $t = 1$ to $t = T$.

In practice, when computing γ and ξ we can exploit the fact that the the denominators of $\gamma_i(t)$ $\xi_{ij}(t)$ are the same to save on computation.

We now complete the maximisation step by updating the model parameters to maximise the values of the paths that are used a lot (while still respecting the stochastic constraints). We update a as follows:

$$\hat{a}_{ij} = \frac{Expected\,Number\,of\,Transitions\,from\,State\,i\,to\,State\,j}{Expected\,Number\,of\,Transitions\,from\,State\,i}$$

$$\hat{b}_{ij} = \frac{Expected\,Number\,of\,Times\,in\,State\,j\,and\,Seeing\,Symbol\,v_k}{Expected\,Number\,of\,Times\,in\,State\,j}$$

Which using:

$$\sum_{t=1}^{T} \gamma_i(t) = Expected\,Number\,of\,Transitions\,from\,x_i$$

$$\sum_{t=t}^{T} \xi_t(i, j) = Expected\,Number\,of\,Transitions\,from\,x_i\,to\,x_j$$

We can define as:

$$\pi*_i = \gamma_i(1)$$

$$a*_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

$$b*_i(v_k) = \frac{\sum_{t=1\, y_t=v_k}^{T} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)}$$

In the event we have multiple sequences (call the number of sequences R) we then combine all

$$\pi_i^* = \frac{\sum_{r=1}^{R} \gamma_{ir}(1)}{R}$$

$$a_{ij}^* = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T-1} \xi_{ijr}(t)}{\sum_{r=1}^{R}\sum_{t=1}^{T-1} \gamma_{ir}(t)}, a_{ij}^* = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T-1} \xi_{ijr}(t)}{\sum_{r=1}^{R}\sum_{t=1}^{T-1} \gamma_{ir}(t)},$$

$$b_i^*(v_k) = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T} 1_{y_{tr}=v_k} \gamma_{ir}(t)}{\sum_{r=1}^{R}\sum_{t=1}^{T} \gamma_{ir}(t)}, b_i^*(v_k) = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T} 1_{y_{tr}=v_k} \gamma_{ir}(t)}{\sum_{r=1}^{R}\sum_{t=1}^{T} \gamma_{ir}(t)},$$

We repeat the expectation and maximisation procedure until the desired level of convergence. At every iteration, the algorithm returns parameters with a increasing likelihood of producing the observation sequence O. We say for each iteration that produces a new set of parameters λ from an old set of parameters θ that $p(O|\lambda) \geq p(O|\theta)$ where we have equality only if the parameters are the same (the model has converged).

1.2 Implementation Choices

We make a few notable implementation choices. Firstly, the α and β values can get very small therefore to avoid (potential) under or overflow (due to numerical precision) we normalise α and β (based on [3]) by

$$Z(t) = \sum_{i=1}^{N} \alpha_i(t)$$

$$\alpha_i(t)* = \frac{\alpha_i(t)}{Z(t)}$$

$$\beta_i(t)* = \frac{\beta_i(t)}{Z(t)}$$

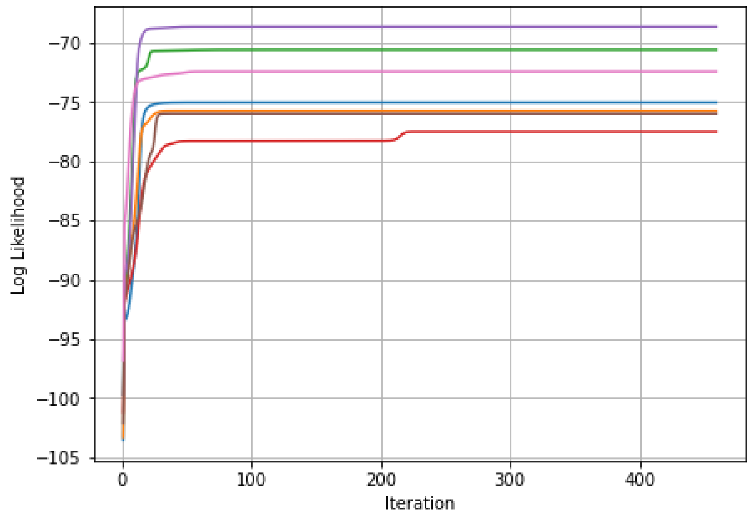
We note that a popular way to mitigate numerical error is to work in log space. However, having implemented both, we found that the log implementation was slower and we did not find any significant issues with numerical accuracy when verifying our implementation. For that reason, we have chosen to submit the normal implementation but have left the log implementation in the comments as evidence of this implementation.

In our implementation, we stop iterating when there is no increase in the log likelihood. We chose log likelihood as the likelihood can be quite small and as the log function effectively 'stretches out' small numbers it is ideal for this task.

As the Baum-Welch only guarantees finding a local maximum, it may be the case that this local maximum is not global. To find a higher (potentially global) maximum we can conduct a greater search over the parameter space by running the algorithm multiple times, with the parameters initialised different. Evidently, for longer sequences this would take too long so we run the algorithm in parallel and output the solution with the highest probability of observing the sequence given the model parameters. This helps when verifying.

1.2.1 Verifying correctness

The correctness of our implementation was verified in different ways. Firstly, for each iteration, the Baum-Welch produces news parameters such that $p(O|\lambda) \geq p(O|\theta)$. We verified this by printing the value of $p(O|\theta)$ for every iteration and tested that it was increasing. We have included the convergence graphs $p(O|\theta)$ for one of the tests we completed as evidence of this.



Convergence plot for HMM model

As an additional check, as the Baum-Welch algorithm converges to a local maximum the parameters were checked to be a local maximum by verifying $P(Y_r|\theta)$ was a maximum for those parameters in a neighbourhood of θ.

The most complete test was creating a hidden Markov model (by initialising the a, b, π matrices) and samples sequences from this. And then running our algorithm on 50 test cases each with 1-7 different sequences. The number of hidden states and number of observation symbols was varied from 2 to 20. We used sequences of random length from 20 to 10,000. We deliberately made sure to test it on both ergodic and non-ergodic HMM. We ran these experiments on the NCC. In all cases, our implementation was able to recover the true parameters or returned the same parameters as the hmmlearn library when we initialised in the same way. We have included this verification code at the bottom of our code as evidence.

2 Question 2

2.1 Question 2a

Given a rooted tree, we define the lowest common ancestor of a pair of nodes x, y to be the node a that is furthest from the root of the tree such that the set of nodes that descent from a contain both nodes x, y. We will denote the lowest common ancestor of two nodes x and y using the notation (x, y). The lowest common ancestor is symmetric so $(x, y) = (y, x)$. We can define constraints on nodes in a rooted tree of the form $(i, j)(k, l)$ which signifies that the lowest common ancestor of the nodes i, j is a descendant of the lowest common ancestor of the nodes k, l.

Given a set of labelled leaves and a set of constraints for sets of these leaves, the $BUILD$ algorithm establishes if there exists a tree where all these leaves are contained in the tree and the constraints are satisfied and if so, it rebuilds this tree. Given a bioinformatics context, if we assume that the tree is a phylogeny, the constraints passed in describe how closely related pairs of species are. For example if we consider the constraint $(i, j)(k, l)$, we expect all species i, j, k, l to share a common ancestor however i, j are more closely related.

We now outline how Build works. We first call Build on a set of leaves S and on a set of constraints on these leaves C. We note that if the leaf set S consists of a single node we return the tree consisting of that single node. Else, we then partition these leaves into blocks by applying three rules using the constraint set C. Given a constraint in the form $(i, j)(k, l)$, the first rule is that i and j have to be in the same block. If we did not require this, then (i, j) would be the root of the tree, which cannot be a descendent of (k, l). The second rule is if k and l are in the same block then i, j, k, l have to be in that same block, else k, l should be in separate blocks. This is necessary because otherwise it would not be possible for (i, j) to be a descendent of (k, l). The final rule is that no two leaves may be in the same block unless it is a consequence of the first two rules.

Having computed a partition π_c, we now check that it has at least two blocks. If it has less than 2 blocks then we return the null tree.

If the partition has two or more blocks we proceed as follows. For each block $S_1, .., S_r$ in the partition, we construct a set of constraints, that is a subset of the original constraints C, by adding all the constraints that only contain leaves from that block.

We then recursively call the Build algorithm on each block and each constructed set of constraints for that block. If these recursive calls return the null tree then no such tree, that satisfies the original constraints, exist. Else we construct the tree by starting with a new root node and letting the children of this node be the tree returned by the recursive calls.

2.2 Question 2b

To run BUILD on the set of constraints this has been shown below. However, we discuss, in detail, how the first set of partitions were computed for clarity. By applying rule 1 and 3 on the first column of constraints, this yields the partitions [e,f], [c,h,a], [j,n,l], [d,i]. However, the constraint $(c, l)l(g, k)$ requires us to merge the second and third partitions by rule 2. We then proceed by creating an additional block for the constraint $(g, b)l(g, i)$ by rule 1 and 3 to give the partitions [e,f], [c,h,a,j,n,l], [d,i], [g,b]. However for the next constraint $(g, i)l(d, m)$ we need to apply rule 2 to merge the third and fourth partitions to give [e,f], [c,h,a,j,n,l], [d,i,g,b]. The constraint $(e, f)l(h, l)$ requires us, by rule 2, to merge the first and second partitions to give [e,f,c,h,a,j,n,l], [d,i,g,b]. Finally by applying rule 1 and rule 3 for the constraint $(k, m)l(e, i)$ we obtain the first partition blocks: [e,f,c,h,a,j,n,l], [d,i,g,b], [k,m].

To be very clear we label each block or constraint by the recursion depth using the upper index and then use what number call this is (at that recursion depth) for the lower index.

The depth of recursion is: 0

Partition Blocks:

S_1^1: [e,f,c,h,a,j,n,l], S_2^1: [d,i,g,b], S_3^1: [k,m]

Constraints relevant to each block:

C_1^1	C_2^1	C_3^1
$(c,h) < (a,n)$,	$(d,i) < (g,i)$,	$\emptyset$
$(j,n) < (j,l)$,	$(g,b) < (g,i)$,	
$(c,a) < (f,h)$,		
$(j,l) < (e,n)$,		
$(n,l) < (a,f)$,		
$(c,h) < (c,a)$,		
$(e,f) < (h,l)$,		
$(j,l) < (j,a)$,		
$(j,n) < (j,f)$,		

Recursion Calls made (in the form BUILD(Block, Constraints):

BUILD(S_1^1, C_1^1)
BUILD(S_2^1, C_2^1)
BUILD(S_3^1, C_3^1)

The depth of recursion is: 1

BUILD(S_1^1, C_1^1):

Partition Blocks:

S_1^2: [c, h, a], S_2^2: [j, n, l], S_3^2: [e, f]

Constraints relevant to each block:

C_1^2	C_2^2	C_3^2
$(c,h) < (c,a)$	$(j,n) < (j,l)$	$\emptyset$

Recursion Calls made:

BUILD(S_1^2 , C_1^2)
BUILD(S_2^2 , C_2^2)
BUILD(S_3^2 , C_3^2)

BUILD(S_2^1, C_2^1):

Partition Blocks:
S_4^2: [d,i], S_5^2: [g,b]
Constraints relevant to each block:

C_4^2	C_5^2
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_4^2 , C_4^2)
BUILD(S_5^2 , C_5^2)

BUILD(S_3^1, C_3^1):

Partition Blocks:
S_6^2: [k], S_7^2: [m]
Constraints relevant to each block:

C_6^2	C_7^2
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_6^2 , C_6^2)
BUILD(S_7^2 , C_7^2)

The depth of recursion is: 2

BUILD(S_1^2, C_1^2):

Partition Blocks:
S_1^3: [c,h], S_2^3: [a]
Constraints relevant to each block:

C_1^3	C_2^3
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_1^3 , C_1^3)
BUILD(S_2^3 , C_2^3)

BUILD(S_2^2, C_2^2):

Partition Blocks:
S_3^3: [j,n], S_4^3: [l]
Constraints relevant to each block:

C_3^3	C_4^3
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_3^3 , C_3^3)
BUILD(S_4^3 , C_4^3)

BUILD(S_3^2, C_3^2):

Partition Blocks:
S_5^3: [e], S_6^3: [f]
Constraints relevant to each block:

C_5^3	C_6^3
$\emptyset$	$\emptyset$

Recursion Calls made:
Block
Constraints

BUILD(S_5^3 , C_5^3)
BUILD(S_6^3 , C_6^3)

BUILD(S_4^2, C_4^2):

Partition Blocks:
S_7^3: [d], S_8^3: [i]
Constraints relevant to each block:

C_7^3	C_8^3
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_7^3 , C_7^3)
BUILD(S_8^3 , C_8^3)

BUILD(S_5^2, C_5^2):

Partition Blocks:
S_9^3: [g], S_{10}^3: [b]
Constraints relevant to each block:

C_9^3	C_{10}^3
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_9^3 , C_9^3)
BUILD(S_{10}^3 , C_{10}^3)

BUILD(S_6^2, C_6^2):

return k

BUILD(S_7^2, C_7^2):

return m

The depth of recursion is: 3

BUILD(S_1^3, C_1^3):

Partition Blocks:
S_1^4: [c], S_2^4: [h]
Constraints relevant to each block:

C_1^4	C_2^4
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_1^4 , C_1^4)
BUILD(S_2^4 , C_2^4)

BUILD(S_2^3, C_2^3):

Return a

BUILD(S_3^3, C_3^3):

Partition Blocks:
S_1^4: [j], S_2^4: [n]
Constraints relevant to each block:

C_3^4	C_4^4
$\emptyset$	$\emptyset$

Recursion Calls made:
BUILD(S_3^4 , C_3^4)
BUILD(S_4^4 , C_4^4)

BUILD(S_4^3, C_4^3):

return l

BUILD(S_5^3, C_5^3):

return e

BUILD(S^3_6, C^3_6):

return f

BUILD(S^3_7, C^3_7):

return d

BUILD(S^3_8, C^3_8):

return i

BUILD(S^3_9, C^3_9):

return g

BUILD(S^3_{10}, C^3_{10}):

return b

The depth of recursion is: 4

BUILD(S^4_1, C^4_1):

return c

BUILD(S^4_2, C^4_2):

return h

BUILD(S^4_3, C^4_3):

return j

BUILD(S^4_4, C^4_4):

return n

If we now return these values and travel back up the tree to give (in the form call, value returned):

The depth of recursion is: 4

BUILD(S^4_1, C^4_1)	[c]
BUILD(S^4_2, C^4_2)	[h]
BUILD(S^4_3, C^4_3)	[j]
BUILD(S^4_4, C^4_4)	[n]

The depth of recursion is: 3

BUILD(S^3_1, C^3_1)	[c, h]
BUILD(S^3_2, C^3_2)	[a]
BUILD(S^3_3, C^3_3)	[j, n]
BUILD(S^3_4, C^3_4)	[l]
BUILD(S^3_5, C^3_5)	[e]
BUILD(S^3_6, C^3_6)	[f]
BUILD(S^3_7, C^3_7)	[d]
BUILD(S^3_8, C^3_8)	[i]
BUILD(S^3_9, C^3_9)	[g]
BUILD(S^3_{10}, C^3_{10})	[b]

The depth of recursion is: 2

BUILD(S_1^2, C_1^2)	[[c, h], [a]]
BUILD(S_2^2, C_2^2)	[[j, n], l]
BUILD(S_3^2, C_3^2)	[e, f]
BUILD(S_4^2, C_4^2)	[d, i]
BUILD(S_5^2, C_5^2)	[g, b]
BUILD(S_6^2, C_6^2)	[k]
BUILD(S_7^2, C_7^2)	[m]

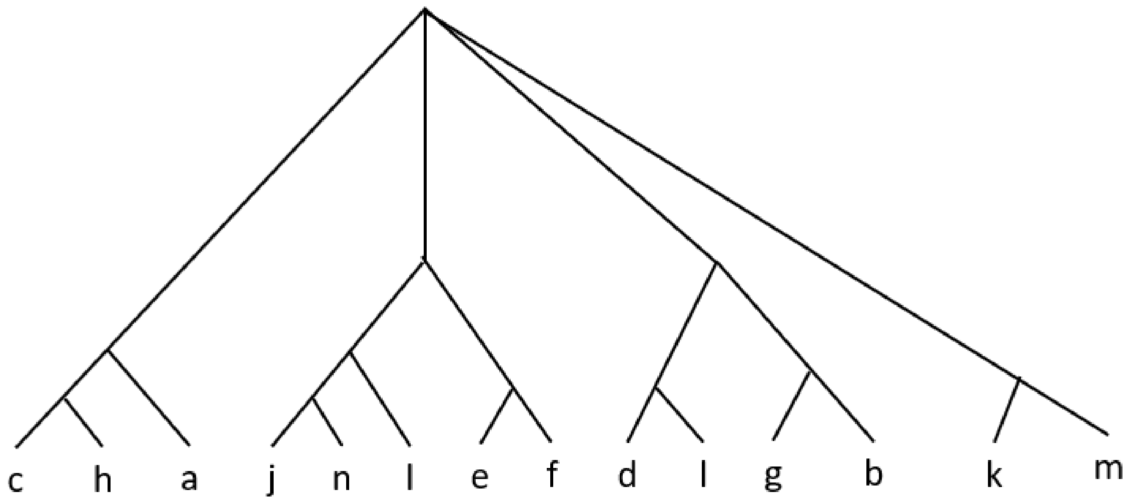
The depth of recursion is: 1

BUILD(S_1^2, C_1^2)	[[[c, h], a], [[j, n], l], [e, f]]
BUILD(S_2^2, C_2^2)	[[d, i], [g, b]]
BUILD(S_3^2, C_3^2)	[k, m]

The depth of recursion is: 0

BUILD(S, C)	[[[[c, h], a], [[j, n], l], [e, f]], [[d, i], [g, b]], [k, m]]

This results in the final tree:



2.3 Question 2c

One advantageous property of MinCutSupertree is that is preserves the nesting and sub-trees that are present and shared by the input trees.

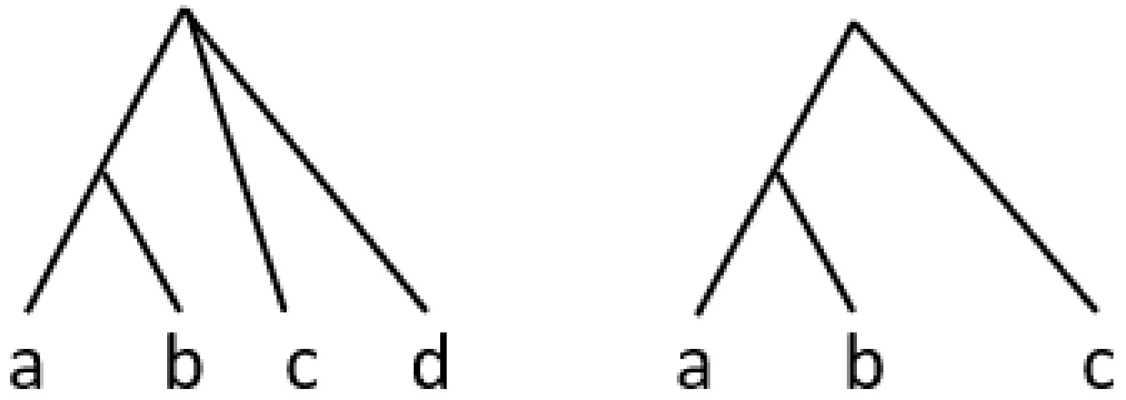
To describe precisely where this is achieved we first define some notation. $\mathcal{T}$ is the input set of rooted phylogenetic trees such that $\mathcal{T} = T_1, .., T_n$. We note that these trees can be weighted. If so we define s to be the weight function, that takes a tree from $T_1, .., T_n$ and returns a weight (that is a rational number). We denote S to be $S \cup_{i=1}^{k} L(T_i)$ where $L(T_i)$ is the leaf set for tree T_i. Given an S, we denote the minimal subtree of T that contains S by $T(S)$. We define $T|S$ to be the rooted phylogenetic tree on S obtained from T(S). We construct $T|S$ by suppressing every vertex with degree two apart from the vertex in T(S) that is nearest to the root of T.

$S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$ graph. We first construct the graph $(S, E_{\mathcal{T}})$, which we denote by $S_{\mathcal{T}}$. For every node pair (a, b) in S, we add edge between the two nodes in $S_{\mathcal{T}}$ if, in at least one tree in $\mathcal{T}|S$, the lowest common ancestor of (a, b) is not the root. In this case, we say a and b and b are in the same proper cluster. To construct $S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$ we then weight the edge (a, b) in $S_{\mathcal{T}}$ by the sum of the weights of the input trees that have a and b in a proper cluster and do this for all edges in $S_{\mathcal{T}}$. We then define the graph E^{max} which is the set of edges that have a weight equal to $w_{sum} := \sum_{T \in \mathcal{T}} w(T)$ (the sum of the weights of the input trees). To obtain $S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$ we merge any nodes in $S_{\mathcal{T}}$ that have an edge in E^{max}. We then proceed to remove all loops and if there are any parallel edges we replace these with a single edge of weight equal to the sum of the trees in $\mathcal{T}|S$ that have a proper cluster containing either one or both of one of the endpoints of the parallel edges. It is the construction of $S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$ that preserves nestings and subtrees in the input trees. If alternatively, we removed every edge that was in the union of all minimum-weight cut sets of $S_{\mathcal{T}}$ we would not preserve the nestings and subtrees. Alternatively, by merging the nodes with edges in E^{max}, we treat these nodes together in the minimum cut so that when we recurse when we hit step two we return the tree with these two nodes as leaves in the same tree.

2.4 Question 2d

We note that the MinCutSupertree is more general than Build as we can weight the input trees. Now we argue that given a Build instance (can a tree be constructed given a set of constraints), we can solve this instance using MinCutSupertree.

Firstly, a constraint can be represented as a tree in one of two ways. Firstly, if a constraint is of the form $(a, b) < (c, d)$, where a, b, c, d are all unique we would graph this as the first tree in the picture below. Alternatively, if a constraint has a repeated leaf (for example $(a, b) < (a, c)$) we would graph this as the second tree.



We note that these are the only two possible ways to represent a constraint as a tree.

To solve a Build instance, using the MinCutSupertree, for each constraint (in the set of constraints passed in) we would construct the appropriate tree (as outlined above). We then pass all the trees (one for each constraint) into the MinCutSupertree, weighting each tree with the same equal constant weight (one). If the MinCutSupertree algorithm outputted a tree we return this, else if it outputs the null tree we output this (which says that given these constraints we cannot construct the tree).

We now argue that if the MinCutSupertree does return a tree, it would be the same tree returned by Build. When creating the S_T graph, by connecting nodes that are in a proper cluster, this corresponds to rule 1 in the build algorithm. We can look at two nodes being connected (in the S_T graph in MinCutSupertree) as the same thing as being in the same partition (in Build). This is because for a constraint $(a, b) < (c, d)$ we know that a, b are in a proper cluster as otherwise their lowest common ancestor would be the root, which could not be descendent of the lowest common ancestor of c, d. Finally, rule 3 in Build is met because initially S_T is disconnected so unless they are in the same proper cluster (rule 1) or are merged (rule 2) they are in a separate set. By recursing on the different nodes sets in the mincut (in MinCutSupertree), this is the same as recursing on the different partitions (as we have shown that the two sets are the same). Furthermore, we also pass in the input trees, with any species not in the node set (we are recursing) removed. This is the same as recursing on only the relevant constraints. Therefore as we have followed the three rules and recurse in the same way it follows that MinCutSupertree would return the same tree as Build if the tree exists. However, if we recurse on empty node sets in the MinCutSupertree this is the same as returning the null tree in the build as it indicates that one set of constraints is not able to be met.

Therefore as we have shown that an instance of Build can be solved by MinCutSupertree and MinCutSupertree extends Build (by allowing weighted input trees), we have shown that MinCutSupertree is a generalisation of build.

References

[1] https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm

[2] https://people.cs.umass.edu/~mccallum/courses/inlp2004a/lect10-hmm2.pdf

[3] https://ieeexplore.ieee.org/document/18626