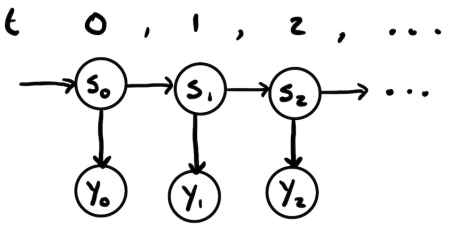
Bioinformatics Assignment Report

wrjx35

March 13, 2022

Hidden Markov Models

In a Hidden Markov Model we consider a set of hidden states S that generates an observed sequence $Y_{0:T}$ which consists of values from a finite alphabet Σ.



The Baum-Welch algorithm aims to estimate the following parameters for the HMM:

1. The **Transition Matrix** A which stores the probabilities of moving from state i to state j.
$$A_{ij} = P[X_t = j | X_{t-1} = i] \tag{1}$$

2. The **Emission Matrix** B which stores the probability of observing a value j when in state i.
$$B_{ij} = P[Y_t = j | X_t = i] \tag{2}$$

3. The **Initial Probabilities** $\pi 0$ which stores the probabilities of staring in each state.
$$\pi 0(i) = P[X_0 = i] \tag{3}$$

To begin calculating estimates for these parameters, the algorithm first initialises them randomly, such that every row in A and B sum to 1 as do the values in $\pi 0$.

The rest of the algorithm can be split into 3 stages:

1 Forward algorithm

The forward algorithm aims to predict the probability that, for time t and state i, the observed sequence at t ($Y_{0:t}$) will end in state i.

$$\alpha_t(i) = P[Y_{0:t}|X_t = i] \tag{4}$$

The Forward algorithm is recursive in nature. The base case can be defined by:

$$\alpha_0(i) = \pi 0(i) P\left(Y_0 \mid X_t = i\right) \tag{5}$$

In plain English, this is the probability of starting at state i and the probability of giving the first value of the observed sequence whilst in state i.

The recursive case of the function can be defined as:

$$\alpha_t(i) = \sum_j \alpha_{t-1}\left(j\right) P\left(X_t = i \mid X_{t-1} = j\right) P\left(Y_t \mid X_t = i\right) \tag{6}$$

i.e. the sum of the forward probabilities at the previous time step for j multiplied by the probability of transitioning to state i from state j and the probability of observing the value in the sequence at the current time step whilst in state i, for all possible previous states j. The latter two probabilities are given in the transition and emission matrices respectively.

In the implementation created for this assignment, a bottom-up dynamic programming approach is used to tabulate the values of α at each time t for each state i, storing them in an $m \times T$ matrix. This is a more computationally efficient method as it means that values do not need to be re-computed every time they are re-used.

2 Backward algorithm

The backward algorithm aims to calculate the probability that when starting in state i at time t, the rest of the observed sequence will be generated ($Y_{t+1:T}$).

$$\beta_t(i) = P[Y_{0:t}|X_t = i] \tag{7}$$

Similarly to the forwards algorithm, the backwards algorithm is also recursive. The base case can be defined as:

$$\beta_T(i) = 1 \tag{8}$$

This is due to the fact that at this point there are no more values in the observed sequence at this point, as time T represents the end of the sequence.

The recursive case for the function is defined as:

$$\beta_t(i) = \sum_j \beta_{t+1}\left(j\right) P\left(X_{t+1} = j \mid X_t = i\right) P\left(Y_{t+1} \mid X_{t+1} = j\right) \tag{9}$$

i.e. the sum of the backwards probabilities at the next time step for j multiplied by the probability of transitioning to state j from state i and the probability of observing the value in the sequence at the next time step when in the next state j, for all possible next states j. In the same way as with the Forwards Algorithm, the latter two probabilities are given in the transition and emission matrices respectively.

Similarly to with the forwards function, in the implementation attached to this assignment used a bottom-up dynamic programming approach is used to calculate all the values of β at each time t and for each state i, storing them in an $m \times T$ matrix. In this case the values at time T (the end of the observed sequence) are calculated first, as opposed to in the forward function where the values at time 0 are calculated first. Again this is more computationally efficient than re-computing the values each time they are reused.

One issue encountered when calculating the forward and backwards probabilities of the HMM for large sequences was numerical underflow. To remedy this, the values of α and β were scaled as such:

$$\hat{\alpha}_t(i) = \frac{1}{G_t}\alpha_t(i) \tag{10}$$

$$\hat{\beta}_t(i) = \frac{1}{G_t}\beta_t(i) \tag{11}$$

where,

$$G_t = \sum_j \alpha_t(j) \tag{12}$$

3 Update parameters

The values calculated by the the forwards and backwards functions are used to update the parameters of the HMM. There are two values that are needed to do this both which use the forward and backwards probabilities as components. The first is the probability distribution of a state i at time t given the value of the complete observed sequence denoted as η.

$$\eta_t(i) = P[X_t = i|Y_{0:T}] = \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{\sum_j \hat{\alpha}_t(j)\hat{\beta}_t(j)} \tag{13}$$

The second is the joint probability distribution of two consecutive states at time t given the observed sequence denoted as ξ.

$$\xi_t(i,j) = P[X_t = i, X_{t+1} = j|Y_{0:T}] \tag{14}$$

$$= \frac{\hat{\alpha}_t(i)\hat{\beta}_{t+1}(j)P[X_{t+1} = j|X_t = i]P[Y_{t+1}|X_{t+1} = j]}{\sum_{k,l} \hat{\alpha}_t(k)\hat{\beta}_{t+1}(l)P[X_{t+1} = l|X_t = k]P[Y_{t+1}|X_{t+1} = l]}$$

These can then be used to update the values of A, B and $\pi 0$ as follows:

$$A'_{ij} = \frac{\sum_t \xi_t(i,j)}{\sum_t \eta_{t-1}(i)} \tag{15}$$

$$B'_{ij} = \frac{\sum_t \eta_t(i) \times 1_{Y_t}}{\sum_t \eta_t(i)} \tag{16}$$

where 1_{Y_t} is 1 if Y_t is the actual value observed in the sequence, otherwise is 0.

$$\pi 0'(i) = \eta_0(i) \tag{17}$$

Calculating the values for η and ξ is done in a similar way to that of α and β by storing them in a matrix (instead of computing them on the fly). For η an $m \times T$ matrix is used storing the value $\eta_t(i)$ for all possible time values t and states i, and for ξ a $T \times m \times m$ matrix is used to store the value $\xi_t(i,j)$ for all possible time values t and states i and j.

The matrices A, B and π_0 are then iterated through, using the computed matrices for η and ξ as per the equations above. When calculating values in A, B, η and ξ, the denominators were pre-computed as early as possible to reduce the overall amount of computation.

The algorithm runs for either a user defined number of iterations or until it detects that the content of A, B and π_0 has not changed by a notable degree over one iteration.

Validation of Algorithm

To validate that the implementation did indeed work, a pair of simple test cases were run, predicting parameters for strings of lengths 5 and 7. The resulting matrices were then used to generate a strings sampling the probabilities they contained to see whether the generated string(s) matched the input string.

The experiment used an input string of $ABCCB$ and consisted of 5 states.

```
A, B, pi0 = baum_welch(5, "ABCCB", "ABC")
```

Which generated the following parameter values:

$$A = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix},$$

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \\ 1.0 & 0.0 & 0.0 \end{bmatrix},$$

$$\pi_0 = \begin{bmatrix} 0.67796112 & 0.0 & 0.0 & 0.0 & 0.32203888 \end{bmatrix}$$

Starting with the state with the highest probability in π_0 results in the following sequence of states `[1,2,3,4,2]` and results in the values `[A,B,C,C,B]` being observed, as expected.

Starting in state 5 (which has the only other non-zero value in π_0) results in the sequence of states; `[5,2,3,4,5]` and also results in the expected observed values; `[A,B,C,C,B]`.

The second experiment used an input string of $ABAACBC$ and consisted of 7 states.

```
A, B, pi0 = baum_welch(7, "ABAACBC", "ABC")
```

Which generated the following parameter values:

$$A = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.143 & 0.857 & 0.0 \end{bmatrix},$$

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \\ 1.0 & 0.0 & 0.0 \end{bmatrix},$$

$$\pi_0 = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

The two possible paths followed by using these matrices to generate an observed sequence of length 7 both result in the desired string $ABAACBC$, providing further validation of the implementation.

Phylogenetic Trees

Question 2(a)

The BUILD algorithm [1] is an algorithm designed to reconstruct a tree T when given its set of its leaves and a set of constraints on the lowest common ancestors between the leaves.

A constraint on a subset of leaves can be given by the formula $(i, j) < (k, l)$, where i, j, k and l are individual leaves and $i \neq j$ and $k \neq l$. This constraint

means that the lowest common ancestor of i and j is a proper descendant of the lowest common ancestor of k and l.

The BUILD algorithm [1] works recursively in a depth-first manner. In each recursive call, the algorithm splits the set of leaves into multiple partitions according to a pair of rules:

1. i and j must be in the same partition.
2. if k and l are put in the same partition, then i and j must also be in this partition.

Then for each generated partition the algorithm calls itself along with a subset of constraints whereby only leaves present in the subset are represented in the constraints. If the sub-set contains only a single leaf (i.e. the algorithm reaches the bottom of the recursive branch), then the algorithm returns a tree containing just that single leaf. If the algorithm is unable to partition the (sub)set of leaves any further and the size of the (sub)set is > 1, the algorithm returns a null tree. The tree returned for each partition call are combined to share the same root. This tree is returned from the function. If the the returned tree is not a null tree, then we know that the tree satisfies the given constraints, otherwise no tree can exist which satisfies the given constraints.

Question 2(b)

The solution to $2b$ can be seen as handwritten notes on pages 7-13.

{a, b, c, d, e, f, g, h, i, j, k, l, m, n}

Calculate partitions

apply Rule 1

{e, f} {c, h, a, l, j, n}

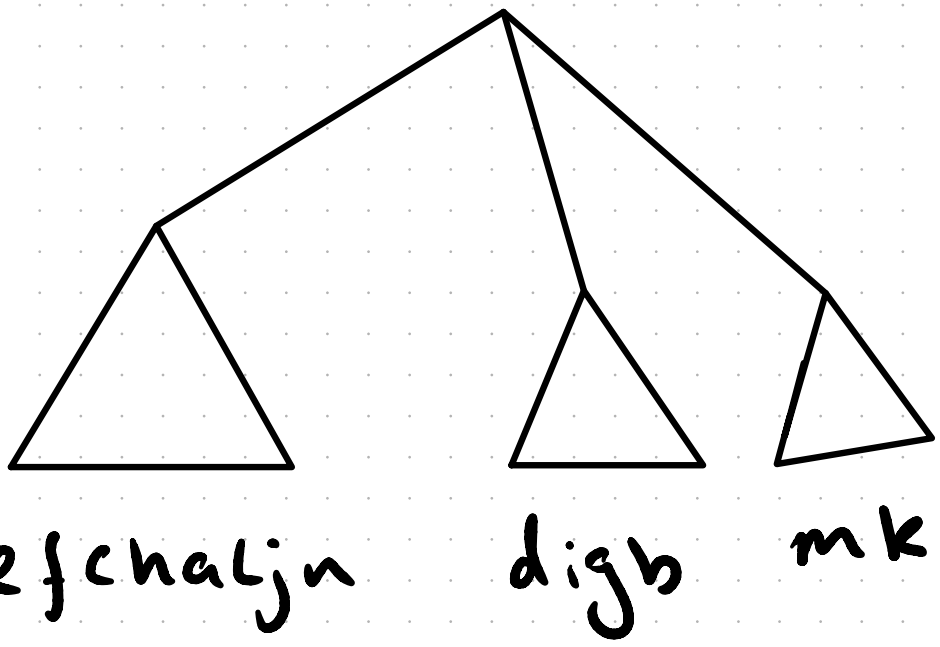
{d, i, g, b} {m, k}

apply rule 2

S_1 = {e, f, c, h, a, l, j, n}

S_2 = {d, i, g, b}

S_3 = {m, k}



Recurse into S_1.

$\{e, f, c, h, a, l, j, n\}$

calculate partitions

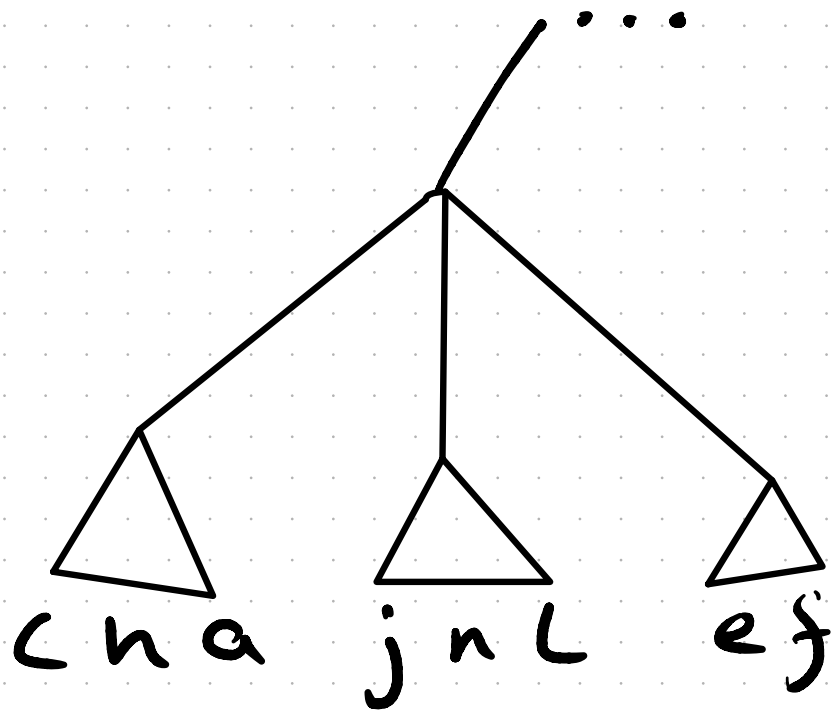
$S_1' = \{c, h, a\}$

$S_2' = \{j, n, l\}$

$S_3' = \{e, f\}$

Constraints germane to S_1:

$(c,h) < (a,h)$
$(j,n) < (j,l)$
$(c,a) < (f,h)$
$(j,l) < (e,h)$
$(h,l) < (a,f)$
$(c,h) < (c,a)$
$(e,f) < (h,l)$
$(j,l) < (j,a)$
$(j,h) < (j,f)$



Recurse into S'_1

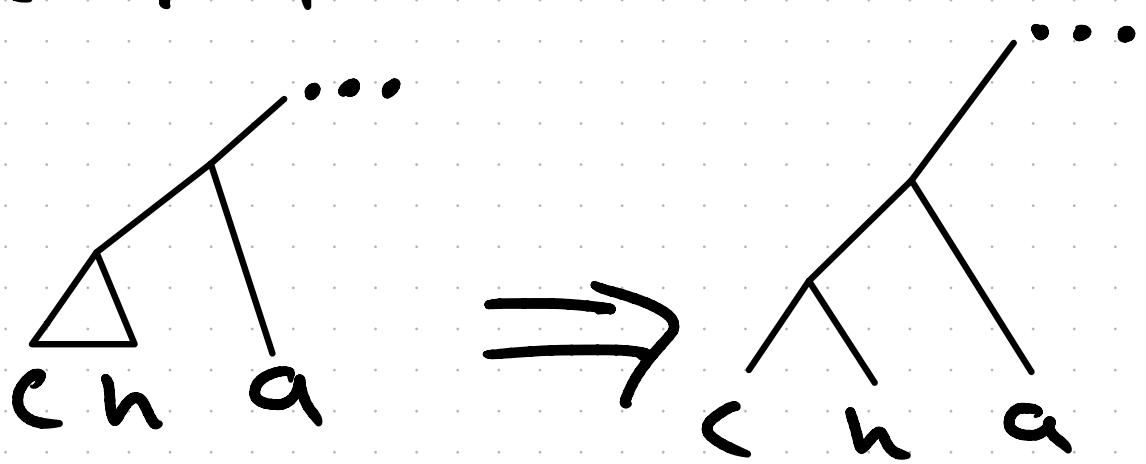
$\{c, h, a\}$

Calculate π_c

| Constraints germane to S'_1:
| $(c,h) < (c,a)$

$S''_1 = \{c, h\}$

$S''_2 = \{a\}$



(recursion into S''_1 is trivial)

Recurse into S_2'

$\{j, n, L\}$

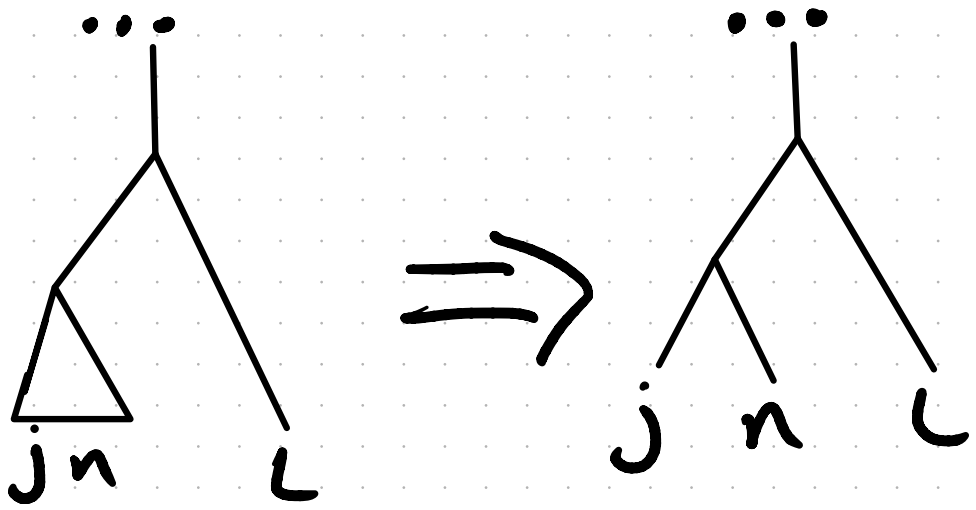
Constraints germane to S_2':

$(j, n) < (j, L)$

Calculate π_C

$S_1'' = \{j, n\}$

$S_2'' = \{L\}$



(recursion into S_1'' is trivial)

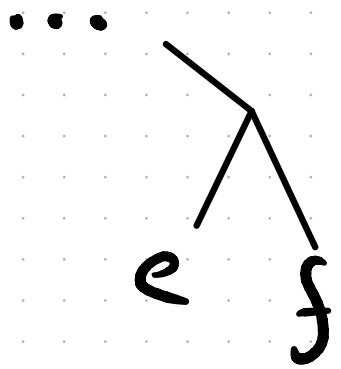
Recursion into S_3' | no constraints

$\{e, f\}$

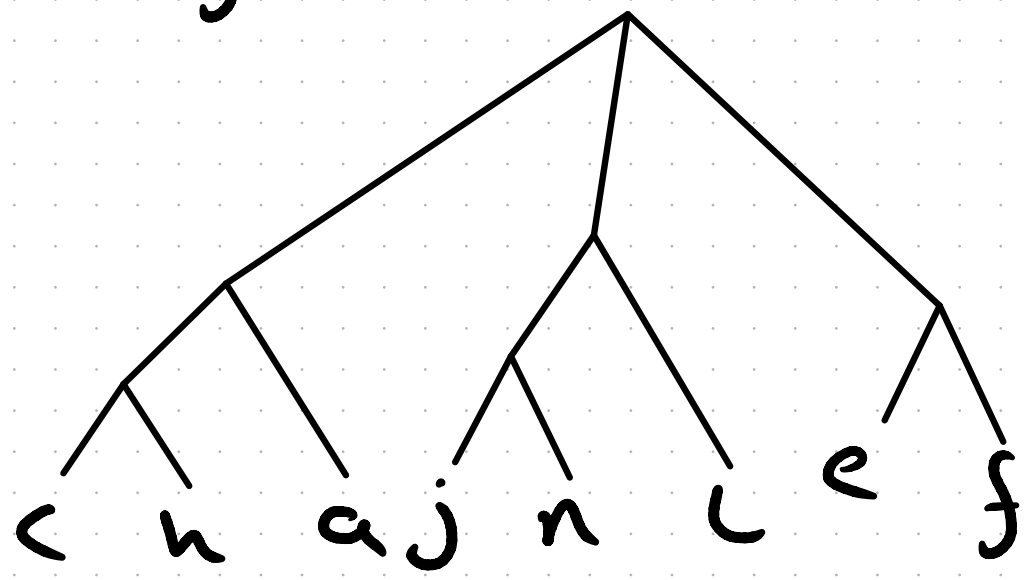
Calculate π_c

$S_1'' = \{e\}$

$S_2'' = \{f\}$



$\therefore T_1 =$



Recurse into S_2

$\{d, i, g, b\}$

Calculate π_c

Constraints germane to S_2:

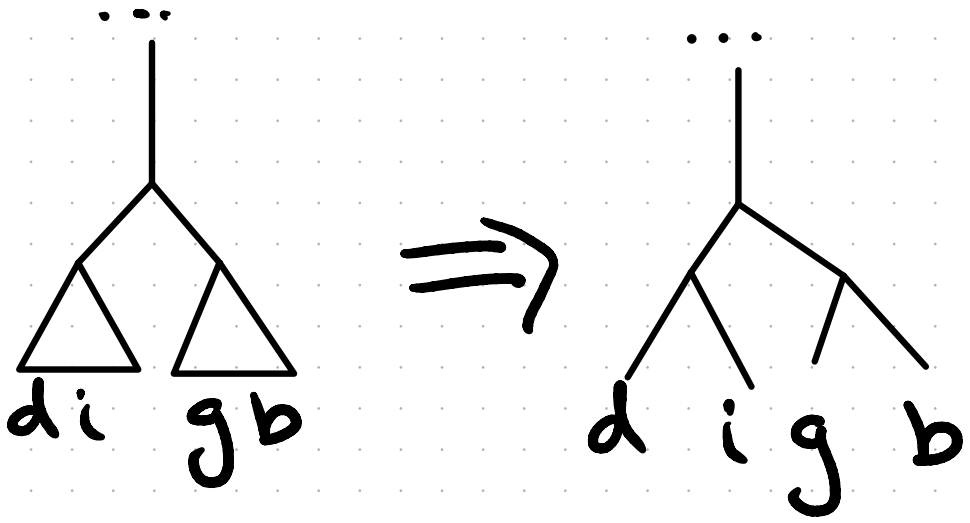
$(d, i) < (g, i)$

$(g, b) < (g, i)$

$S_1' = \{d, i\}$

$S_2' = \{g, b\}$

$T_2 =$



(recursion into S_1' and S_2' is trivial)

Recursion into S_3

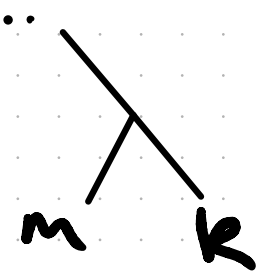
$\{M, K\}$

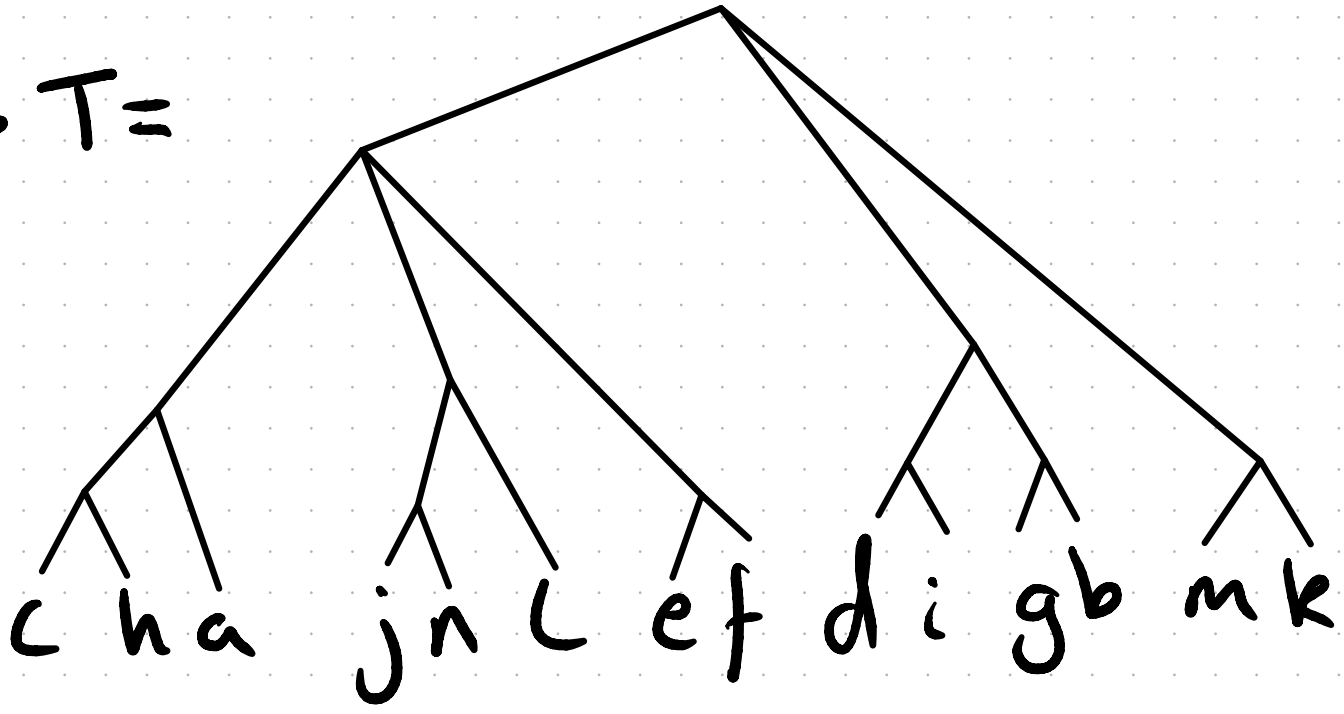
no constraints

Calculate π_c

$S_1' = \{M\}$

$S_2' = \{K\}$

$T_3 =$ 

$\therefore T =$ 

Question 2(c)

The authors of [3] provide a proof that if there are two subsets, A and B, of $\bigcap_{i=1}^{k} \mathcal{L}(T_i)$ whereby A is nesting in B for all $i \in \{1, 2, ..., k\}$, then in the tree returned by the MinCutSupertree algorithm must also have A nesting in B.

This property is maintained when constructing the tree $S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$ as the subset A will be represented as a single element. This is due to the fact that the sub-tree $S_{\mathcal{T}}$ induced by A will be a clique of size $|A|$ and, as such, in the final step of obtaining $S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$ from $S_{\mathcal{T}}$ by deleting all the loops remaining in $S_{\mathcal{T}}$ and replacing parallel edges with a single edge, the result will be a single element.

When the final tree is constructed from the the sub-trees constructed at this recursive level, A, and the elements it contains, will be treat as a single node, meaning that the sub-graph will become adjacent to the the new root node of all vertices listed in $S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$ via its own root. As a result, the position of A, and its elements, within the larger tree will remain unaffected, thus maintaining its nesting.

Question 2(d)

It can be argued that the MinCutSupertree algorithm is a generalisation of the BUILD algorithm. The BUILD algorithm aims to reconstruct a tree when given a set of leaves and a set of constraints on the lowest common ancestors between its leaves, such that the tree satisfies these constraints.

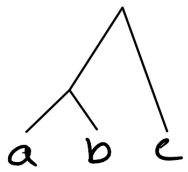


Figure 1: T, a rooted triple $ab|c$

The MinCutSupertree attempts to amalgamate a set of rooted phylogenetic trees into one larger rooted phylogenetic. This is essentially the same thing as the aim of the BUILD algorithm, just with re-defined inputs. The rooted phylogenetic trees themselves define a set of constraints on their leave due to their design. For example take Figure 1 which depicts a phylogenetic tree T. This is a tree that can be used as part of the input for the MinCutSupertree algorithm such that we may define $\mathcal{L}(T) = \{a, b, c\}$. From this we can deduce that the inputs to the BUILD algorithm that would produce the same output are the leaves $\{a, b, c\}$ and the set of constraints $\{(a, b) < (a, c)\}$. While this is a very simple case, the logic would apply to much larger and more complex rooted phylogenetic trees. Given a set of rooted phylogenetic trees we would be able to combine the leaf set for each tree to get the input to the BUILD algorithm

and combining the set of constraints constraints would defined by the structure of each tree.

The MinCutSupertree algorithm becomes a generalisation of BUILD due to the fact that it works on more input cases. The BUILD algorithm returns a null tree if and only if the there is not a tree which satisfies the given constraints. Therefore it only returns a tree if all the constraints can be satisfied. MinCutSupertree on the other hand is not restricted in this way. If it finds the input subtrees are incompatible (i.e. that a tree cannot be constructed based on the constraints defined by the input rooted phylogenetic trees) then then it aims to remove the incompatibilities by deleting the minimal amount of information from the input trees such that the algorithm is able to continue [2]. They achieve this by constructing the a subset of edges in $S_{\mathcal{T}}/E_{\mathcal{T}}^{max}$, E', and deleting the edges corresponding to those in this set from the set $E_{\mathcal{T}}$ if $S_{\mathcal{T}}$ is connected. Thus MinCutSupertree always returns something, whereas BUILD does not.

References

[1] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, 10(3):405–421, 1981.

[2] Malte Brinkmeyer, Thasso Griebel, and Sebastian Böcker. Polynomial supertree methods revisited. *Adv. Bioinformatics*, 2011:524182, December 2011.

[3] Charles Semple and Mike Steel. A supertree method for rooted trees. *Discrete Applied Mathematics*, 105:147–158, 07 2000.