Summative Coursework: Bioinformatics
CIS ID: cssg28

1.

a. Design an algorithm that takes a sequence of n letters as an input and outputs the most likely sequence of states (possibly silent) that generated the input. Prove the correctness of your algorithm and estimate the running time and the space used in terms of n and the number of states of the model k.

The Viterbi algorithm for finding the most probable state path can be extended to work on a HMM with silent states. A state *m* can be modeled as "remaining silent" by introducing a new silent state m_s , whereby *m* transitions to m_s (i.e. it "emits" a silent signal) with probability (1 – Σ(the emission probabilities of m)) and m_s does not emit any letters (i.e. it only transitions to other states). I noted that the probability of an observed sequence decreases when a silent state loops into itself; a transition probability multiplication is added to the product, however no symbol is emitted. Thus, looping within a single silent state will never occur in the most probable sequence of states, as the same path excluding the self-loops in silent states will always return a greater probability than if the loops were included. However, the proposed algorithm should taken into account the possibility of transitioning between a chain of consecutive states and remaining silent at each; for example, it may be more probable to remain silent at several consecutive states with high transition probability, and subsequently emit a desired symbol (say 'A') with very high probability after the last silent state, than to immediately emit A from the current state with very low probability (e.g. tending towards 0). Thus it may be possible that the most probable sequence of states is longer than the observed sequence, as remaining silent may increase the emission probability of a desired symbol later on. I also considered the case of cycles comprising multiple silent states; the most probable path should never require a completion of a cycle of silent state transitions, as by virtue of its properties a Markov chain only depends on the current state of the system, not its state at previous steps. It is more probable to emit the next letter in the sequence directly, than to proceed along a cycle of silent states and return to the same state. Going around the cycle of silent states will only add more probabilities to the total product, and no benefit is gained from the additional silent state transitions as we return to the same state from which we began the cycle; as each probability must be between 0 and 1, every additional multiplication decreases the total probability. Thus, in a most probable path, any silent states must follow in a chain (as opposed to a cycle).

"A run r generating the sequence s can be decomposed into the last transition of r from some state q to the end state and a preceding *partial run*, starting in the start state and ending in q and emitting s along the way" (Lyngso 2012). The preceding partial runs can be similarly decomposed via recursion, considering the last transition and final run preceding it; in practice, this is implemented via dynamic programming to store intermediate calculations. If the last state of the partial run remains silent and does not emit a letter, "the preceding partial run must have generated the prefix of s currently considered" (Lyngso 2012). Otherwise, if the last state in the partial run is non-silent, it must have emitted the last letter in the current prefix of s, with the remaining letters [0…s-1] being emitted by the previous partial run. This decomposition which includes the possibility of remaining silent can be reflected in an update to the recursion step in the Viterbi algorithm:

"If V(q, i) denotes the maximum probability of any partial run ending in state q and generating the sequence s[1..i]", then,

$V(q,i) = \mathbf{max}\ (a_{p,q} * e_{q,s[i]} * V(p, i-1)\ )$ if q is non-silent, *and* $(a_{p,q} * V(p, i-1))$ *if q is silent*,

where $a_{p,q}$ indicates the transition probability from state p to q, and $e_{q,s[i]}$ denotes the probability of emitting symbol i at state q. The silent case excludes an emission probability multiplication as no symbol is emitted.

However, if the model contains a cycle of potentially silent states (or an otherwise strongly connected component of silent states), a cyclic dependency arises in computing the values of V(q,i). For example, if there is a cycle of silent states $\{q_1, \ldots, q_k\}$ with $a_{qk,q0} > 0$ and $a_{qi,qi+1} > 0$ for $1 \leq i < k$, a cyclic dependency exists for any $V(q_i, j)$, as j may be emitted after any number of loops (potentially infinite) through $q_1, \ldots, q_k$ before breaking out. However, as we know that looping and returning to the initial "departure" state decreases the probability of the run, the algorithm should take care to only compute $V(q_i, j)$ of the shortest path from the ending state of the previous partial run and q_i, and not all subsequent $V'(q_i, j)$ which result from looping behavior in the cycle $\{q_1, \ldots, q_k\}$. Thus, if a cycle is identified in a given run $V'(q_i, j)$, the algorithm should immediately return $V(q_i, j)$, as its probability is by default maximal. Lyngso suggests that the shortest path computation can be performed by edge relaxation as in Djikstra's algorithm (2012), and $V(q_i, j)$ propagated for all subsequent $V'(q_i, j)$ (i.e. revisiting q_i in loops) and fixed for computing $V(q_k, j)$ of remaining members of the cycle $\{q_1, \ldots, q_k\}$. The edge relation step increases the time complexity of Viterbi, which is normally $O(K^2T)$ for K states and an observation sequence of length T, by a factor of m*log(n), where the graph contains n nodes and m edges.In the case of HMM, the number of nodes n equals the number of states K. Thus, the modified algorithm will have a time complexity of $O(K^2T * m(\log(K))$, where the number of edges m depends on the number of outbound transition probabilities from each state. In terms of space complexity, the algorithm must store intermediate computations from dynamic programming. The general Viterbi algorithm must a maintain a matrix of dimensions K*T, where K is the number of states and T is the observation length, hence O(T*K). The modified Viterbi algorithm will require O(c*2*T*K), as each state *m* must have a corresponding silent state m_s to which it transitions in the event that nothing is emitted (hence 2*). An additional constant factor c is used to represent the possibility that a silent state m_s may transition amongst other silent states in the HMM not directly related to the initial *m*.

i. Proving correctness and runtime

b. Write a brief report that first gives a self-contained description of the EM algorithm, and then provides a validation of your implementation.

i. Description of EM algorithm

Given observed data x and model parameters θ, the Expectation-Maximization (EM) algorithm aims to return a new set of model parameters λ* such that $p(x|\lambda^*) \geq p(x|\theta)$; in other words, the updated set of model parameters λ* increases the likelihood of the observed data, with equality only if $\lambda^*=\theta$ (i.e. parameters are not updated from the previous iteration). Iterating through updates to new model parameters λ* will eventually converge to a local maximum; this maximum represents the model parameters θ that (locally) maximize the probability of observing the data x. In the case of a HMM, the model parameters can be defined as the transition probabilities m_{kl} and emission probabilities $e_k(b)$; the observed data *x* is a sequence of letters of length *L*.

At each iteration, the generalized EM algorithm completes 2 steps: an expectation step and a maximization step. At the expectation step, the virtual sampling function $L_\theta(\lambda) = \prod_y p(y,x|\lambda)^{p(y|x,\theta)}$ is computed given model parameters θ and current λ. Next, at the maximization step, a new set of model parameters λ* is found such that $L_\theta(\lambda)$ is maximized; in practice, λ* need not be optimal but rather be an improvement from the current λ (i.e. $L_\theta(\lambda^*) > L_\theta(\lambda)$).

For HMM, the Baum-Welch algorithm is used to implement the expectation-maximization strategy. The expectation step is modified such that $L_\theta(\lambda) = \prod_s p(s,x|\lambda)^{p(s|x,\theta)}$, where x is the sequence of observed letters, s is the set of states, and λ are the new model parameters $\{m_{kl}, e_k(b)\}$. The probability $p(s,x|\lambda)$ can be computed as the product $\prod_{k,l} m_{kl}^{M_{s,kl}} \prod_{k,b} e_k(b)^{E_{s,k}(b)}$; thus, $L_\theta(\lambda)$ can be further reduced such that $L_\theta(\lambda) = \prod_s (\prod_{k,l} m_{kl}^{M_{s,kl}} \prod_{k,b} e_k(b)^{E_{s,k}(b)})^{p(s|x,\theta)} = \prod_{k,l} m_{kl}^{\sum_s M_{s,kl} p(s|x,\theta)} \prod_{k,b} e_k(b)^{\sum_s E_{s,k}(b) p(s|x,\theta)} = \prod_{k,l} m_{kl}^{\sum_s M_{kl}} \prod_{k,b} e_k(b)^{\sum_s E_k(b))}$, where M_{kl} represents the number of state transitions from *k* to *l* and $E_k(b)$ represents the number of emissions of symbol *b* from state *k*. At the maximization step, $L_\theta(\lambda)$ is

maximized when the m_{kl} and $e_k(b)$ terms are the relative frequencies of the corresponding variables; thus, $m_{kl} = M_{kl} / \Sigma_{l'} M_{kl'}$ and $e_k(b) = E_k(b) / \Sigma_{b'} E_k(b')$.

Programmatically, the Baum-Welch algorithm can be implemented via a dynamic programming approach which incorporates the forward and backward algorithms. The implementation accepts 3 parameters as input: a number of states *n*, alphabet size *a*, and observed input sequence *s*. Error handling is performed on the supplied inputs; namely, that all command line arguments are specified, that the number of states and alphabet length are positive integers, and that the observed input sequence is correctly formatted and only contains valid tokens from the alphabet. It is assumed that the observed sequence is formatted as a string where alphabet tokens are separated by commas; this allows for multi-character tokens when an alphabet is large (for a specified alphabet size of e.g. 15, '12' can be regarded as a distinct token due to comma delimiting, not '1' and '2') Next, the model parameters (initial, transition, and emission probabilities) are initialized randomly, ensuring that the randomized probabilities sum to 1 where appropriate.

First, the forward algorithm is used to compute the likelihood of a particular observed input sequence; the forward probability of the observed sequence can be calculated as the sum over probabilities of all possible hidden state paths. A dynamic programming approach is used to to progressively store intermediate values until the probability of the full sequence is computed. We initialize an alpha table to store these intermediate values at each timestep; a cell in row *t*, column *j* in the alpha table represents the probability of being in a given state after seeing the first *t* observations (Jurafsky 2020). The forward probability at time t, state j can be computed as $\alpha t(j) = \Sigma_{i,N} \alpha_{t-1}(i) a_{ij} b_j(o_t)$, where $\alpha_{t-1}(i)$ is the previous forward path probability from the previous time step, a_{ij} is the transition probability from previous state q_i to current state q_j *and* $b_j(o_t)$ is the state observation likelihood of the observation symbol o_t given the current state *j* (Jurafsky 2020). Rabiner scaling is implemented to normalize the alpha values at each time step; the corresponding normalization constants are recorded for re-use when scaling the beta table during the backward algorithm. If scaling is not implemented, the probabilities quickly tend to 0 when longer sequences and more states are passed as input.

Next, the backward algorithm is used to compute backward probabilities, which denote the conditional "probability of seeing the observations from time t + 1 to the end, given that we are in state i at time t" (Jurafsky 2020). The algorithm mimics the forward algorithm in that a dynamic programming approach is implemented to progressively compute intermediate values in the beta table; however, the matrix is populated in a backwards manner, with the last row in the beta table (i.e. at the last entry in the observed sequence) initialized to 1, and subsequent beta values computed following the recursion $\beta_t(i) = a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)$. Rabiner scaling is similarly used to normalize the beta values; the row vector β_t is multiplied by the corresponding normalization constant, $norm_t$, which was used to normalize alpha values at the same timestep.

Upon executing the forward and backward algorithms, the resulting alpha and beta table values can be used to compute an estimate of new transition and emission probabilities (i.e. to update the model parameters). An intermediate value ξ_t is computed for each (i,j) to denote the "probability of being in state *i* at time *t* and state *j* at time *t* + 1, given the observation sequence": $\xi_t(i,j) = [(\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)] / [\Sigma_{j=1,N} \alpha_t(j) \beta_t(j)]$ (Jurafsky 2020). $\xi_t(i,j)$ can subsequently be used to compute $\hat{a}_{ij}$, the new estimate of the transition probability from state *i* to state *j*, where $\hat{a}_{ij}$ = (the number of expected transitions from state *i* to state *j*) / (the total expected number of transitions from state i) = $(\Sigma_{t=1,T-1} \xi_t(i,j)) / (\Sigma_{t=1,T-1} \Sigma_{k=1,N} \xi_t(i,k))$. Next, the updated emission probability estimates $\hat{b}_j(v_k)$ are calculated, where $\hat{b}_j(v_k)$ equals the expected number of times in state *j* and emitting symbol v_k, divided by the total expected number of times in state *j* (Jiravsky 2020). This computation hinges on a second intermediate value, $\gamma_t(j)$, where gamma can be computed as the product of alpha and beta values = $(\alpha_t(j) * \beta_t(j)) / (\Sigma_{j=1,M} \alpha_t(j) * \beta_t(j))$. Using gamma, the updated emission probability $\hat{b}_j(v_k)$ can be computed as $(\Sigma_{t=1,T \text{ s.t. } O\_t = v\_k} \gamma_t(j)) / (\Sigma_{t=1,T} \gamma_t(j))$; the summation in the numerator can be interpreted as "the sum over all *t* for which the observation at time *t* was v_k" (Jiravsky 2020).

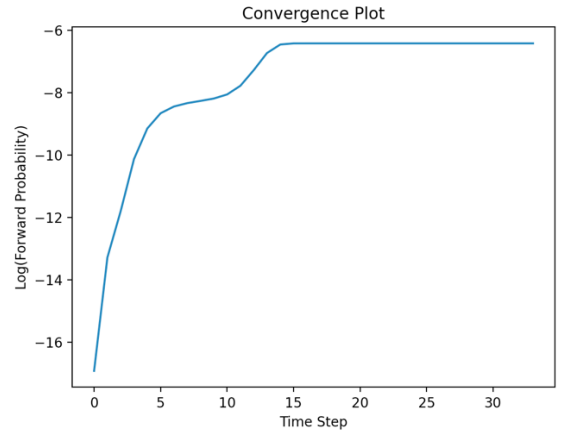
The model parameter estimates are updated iteratively; new alpha and beta values are computed at each expectation step, and the corresponding parameter updates are computed using these values in the maximization step. With each iteration, the log-likelihood of the model should improve (i.e. each updated estimate of the

model parameters is at least as good as the previous set) until the algorithm converges at a local maximum. The algorithm terminates when the maximum number of iterations (preset to 1000) is reached or a termination condition is satisfied. The implemented termination condition requires that the log-likelihood should fail to improve for at least 5 consecutive iterations before the algorithm halts. Upon terminating, the estimated model parameters (initial, transition, and emission probability matrices) are returned.

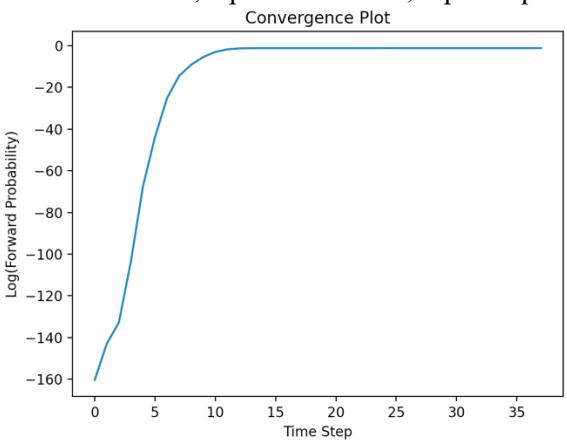
ii. Implementation validation

Convergence of the Baum-Welsh algorithm was evaluated by plotting log(forward probability) at each time step. Convergence can be visualized by a plateau in log-likelihood after a certain number of timesteps, indicating that the algorithm has converged to a local maximum. The plot() function included in the implementation can be used to generate a chart plotting log(forward probability) versus time. The following charts showcase the convergence of the algorithm under various initialization parameters:

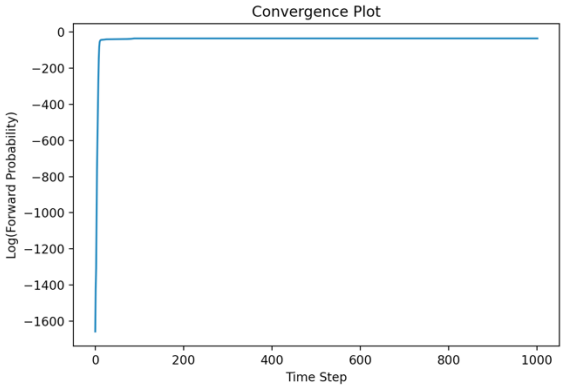
a. 5 states, alphabet size = 5, input sequence of length 10



b. 50 states, alphabet size = 5, input sequence of length 100



c. 50 states, alphabet size = 5, input sequence of length 1000



2.

a. Explain what the BUILD algorithm does and how it works in your own words. Do not use pseudocode.

The BUILD algorithm aims to solve the tree discovery problem, which asks whether, given a set of constraints, can one reconstruct a satisfying tree T, or determine that no such tree exists? The number of leaves on T, numbered 1,2,...,n, are known. The supplied constraints indicate lowest common ancestors of certain pairs of leaves; a valid constraint is of the form (i,j,) < (k,l), ij kl, where "the lowest common ancestor of i and j is a proper descendant of the lowest common ancestor of k and l". The lowest common ancestor (x,y) of two tree nodes x and y is defined as "the node a in T that is an ancestor of both x and y such that no proper descendant of a is also an ancestor of both x and y". The notion of 'proper descendant' implies that -.

The BUILD algorithm takes as input a set of leaf nodes S and set of constraints C. The algorithm is recursive in nature, partitioning the tree into further sets of nodes and recursively building trees for each set until the base case (a set containing a single node) is reached or a satisfying tree does not exist. If BUILD(S,C) returns a nonnull tree T, then T is a satisfying tree for the provided constraints C; otherwise, no such tree exists.

A call to BUILD(S,C) proceeds as follows. First, the size of S is checked. If S contains only a single node *i*, then a tree comprising only *i* is returned; this is the recursion base case or is otherwise encountered if a tree containing a single node is passed as input to the algorithm. Otherwise, the algorithm proceeds to compute the partition π_c = S,.....,, which, given constraints C, partitions the nodes of S into two or more sets. The partition π_c must satisfy the following 3 rules: if (i,j) < (k,l) is a constraint, then i and j are in one set of π_c; if (i,j) < (k,l) is a constraint, and k and l are in one set, then i,j, k, and l are all in the same set; no two leaves are in the same set of π_c unless it follows from the two prior rules. In the event that no constraints are applicable to the nodes in a block, they are partitioned into sets of singleton leaf nodes. If the resulting partition π_c contains at least 2 sets, the algorithm proceeds to recursively build a tree for each set by calling BUILD(S_m,C_m) on each set S_m in the partition. For each S_m, the set of constraints C is restricted to a subset C_m which contains those that involve members of S_m only. If at any point in the recursion a valid partition containing at least 2 sets cannot be obtained, the null tree is returned to indicate that no satisfying tree exists. Otherwise, once the base case(s) is reached, the resulting trees T_m are returned from each recursive call. A satisfying tree T can be formalized as one with a new node for its root and whose children are the roots of T_m (i.e. the trees returned by way of recursive calls when iterating from 1 to r for each m) , $1 \leq m, \leq r$, where r is the size of the partition of the set of input nodes S_r.

b. Expand the partition step in pseudocode.

The partition step is based on the following 3 rules:

1. If (i,j) < (k,l) is a constraint, then i and j are in one set of π_c
2. If (i,j) < (k,l) is a constraint, and k and l are in one set, then i,j, k, and l are all in the same set
3. No two leaves are in the same set of π_c unless it follows from the two prior rules

If the resulting partition does not contain at least 2 sets of nodes, then the null tree is returned.

The pseudocode for computing π_c follows, first ensuring that the partition fulfills Rule 1, and subsequently Rule 2:

```
#Compute the partition pi_c = S1, S2, ..., Sr given set of leaf nodes S and
constraints C

partition_sets = {}

#RULE 1
for constraint (i,j) < (k,l) in C:
    #Ensure all constraint elements in S
    if i not in S or j not in S or k not in S or l not in S:
        return null
    #Check if (i,j) already in the same set
    for set in partition_sets:
        if i in set and j in set:
            foundBoth = True
        #Check if either i or j is in at least one set
        else:
            if i in set:
                foundI = True
                setI_index = set
            else if j in set:
                foundJ = True
                setJ_index = set
    if foundBoth == True:
    #Both i and j already in the same set - satisfying rule 1
        pass
    else if foundI == True:
        #i already in a set - check if j exists elsewhere and merge.
Otherwise, append j.
        if foundJ == True:
            #Merge the distinct sets containing i and j
            partition_sets[setI_index] = partition_sets[setI_index] +
partition_sets[setJ_index]
            partition_sets.remove(partition_sets[setJ_index])
        else:
            #Add j to the set containing i
            partition_sets[setI_index].append(j)
    else if foundJ == True:
        #Add i to the set containing j
        partition_sets[setJ_index].append(i)
    else: #Neither i or j in a set - create a new set.
        new_set = {}
        new_set.append(i)
        new_set.append(j)
        partition_sets.append(new_set)

#Check for singletons; leaf nodes not explicitly mentioned in constraints
for leaf_node in S:
    for set in partition_sets:
        if leaf_node in set:
```

```
            foundLeaf = True
    if leaf_node == False:
        new_set = {}
        new_set.append(leaf_node)
        partition_sets.append(new_set)

#RULE 2
for constraint (i,j) < (k,l) in C:
    #Check if k and l in the same set
    for set in partition_sets:
        if k in set and l in set:
            foundBoth = True
            kl_set_index = set
    #If k and l in the same set, merge with the set containing i and j
    if foundBoth == True:
        for set in partition_sets:
            if i in set and j in set:
                set = set + partition_sets[kl_set_index]
                partition_sets.remove(kl_set_index)
```

c. Run the algorithm on the following set of constraints. You should show the partitioning and the recursive calls at each stage.

S: *a, b, c, d, e, f, g, h, i, j, k, l, m, n*

C:

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

$(d,i) < (k,n)$ $(d,i) < (g,i)$ $(c,l) < (g,k)$ $(g,b) < (g,i)$ $(g,i) < (d,m)$ $(c,h) < (c,a)$

$(e,f) < (h,l)$ $(j,l) < (j,a)$ $(k,m) < (e,i)$ $(j,n) < (j,f)$

Call to BUILD(S,C):

Compute the partition π_c:

Partition Sets: {}

Rule 1: If $(i,j) < (k,l)$ is a constraint, then *i* and *j* are in one set of π_c.

Iterating through constraints, applying Rule 1,

$(e,f) < (k,d)$; append *e,f* to the same set

Update partition sets: { [*e,f*] }

$(c,h) < (a,n)$; append *c,h* to the same set

Update partition sets: { [*e,f*], [*c,h*] }

$(j,n) < (j,l)$; append *j,n* to the same set

Update partition sets: { [*e,f*], [*c,h*], [*j,n*] }

(c,a) < (f,h); append a to the set [c,h]

Update partition sets: { [e,f], [c,h,a], [j,n] }

(j,l) < (e,n); append l to the set [j,n]

Update partition sets: { [e,f], [c,h,a], [j,n,l] }

(n,l) < (a,f); n and l already in the same set

(d,i) < (k,n); append d,i to the same set

Update partition sets: { [e,f], [c,h,a], [j,n,l], [d,i] }

(d,i) < (g,i); d and i already in the same set

(c,l) < (g,k); merge the sets containing c and l

Update partition sets: { [e,f], [c,h,a, j,n,l], [d,i] }

(g,b) < (g,i); append g,b to the same set

Update partition sets: { [e,f], [c,h,a, j,n,l], [d,i], [g,b] }

(g,i) < (d,m); merge the sets containing g and i

Update partition sets: { [e,f], [c,h,a, j,n,l], [d,i,g,b] }

(c,h) < (c,a); c and h already in the same set

(e,f) < (h,l); e and f already in the same set

(j,l) < (j,a); j and l already in the same set

(k,m) < (e,i); append k,m to the same set

Update partition sets: { [e,f], [c,h,a, j,n,l], [d,i,g,b] [k,m] }

(j,n) < (j,f); j and n already in the same set

Rule 2: If (i,j) < (k,l) is a constraint, and k and l are in one set, then i,j, k, and l are all in the same set

Iterating through constraints, applying Rule 2,

Current state of partition: { [e,f], [c,h,a, j,n,l], [d,i,g,b] [k,m] }

(e,f) < (k,d); k,d not in the same set

(c,h) < (a,n); c,h,a,n already in the same set

(j,n) < (j,l); j,n,l already in the same set

(c,a) < (f,h); f,h not in the same set

(j,l) < (e,n); e,n not in the same set

$(n,l) < (a,f)$; a,f not in the same set

$(d,i) < (k,n)$; k,n not in the same set

$(d,i) < (g,i)$; g,i,d already in the same set

$(c,l) < (g,k)$; g,k not in the same set

$(g,b) < (g,i)$; g,i,b already in the same set

$(g,i) < (d,m)$; d,m not in the same set

$(c,h) < (c,a)$; c,a,h already in the same set

$(e,f) < (h,l)$; merge the set $[e,f]$ with the set containing h,l

Update partition sets: { $[e,f,c,h,a,j,n,l]$, $[d,i,g,b]$ $[k,m]$ }

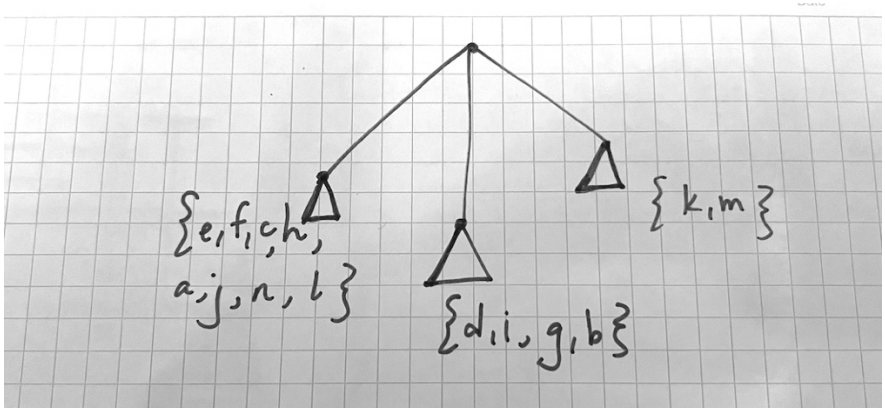
$(j,l) < (j,a)$; j,a,l already in the same set

$(k,m) < (e,i)$; e,i not in the same set

$(j,n) < (j,f)$; j,f,n already in the same set

Final computed partition: { $[e,f,c,h,a,j,n,l]$, $[d,i,g,b]$ $[k,m]$ }

Current state of the tree:



Recurse into set $[e,f,c,h,a,j,n,l]$, calling BUILD(S,C):

S: e,f,c,h,a,j,n,l

C (updated to contain only those pertaining to leaves in S):

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

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

Compute the partition π_c:

Partition Sets: {}

Iterating through constraints, applying Rule 1,

(c,h) < (a,n); append c,h to the same set

Update partition sets: { [c,h] }

(j,n) < (j,l); append j,n to the same set

Update partition sets: { [c,h], [j,n] }

(c,a) < (f,h); append a to the set [c,h]

Update partition sets: { [c,h,a], [j,n] }

(j,l) < (e,n); append l to the set [j,n]

Update partition sets: { [c,h,a], [j,n,l] }

(n,l) < (a,f); n and l already in the same set

(c,h) < (c,a); c and h already in the same set

(e,f) < (h,l); append e,f to the same set

Update partition sets: { [c,h,a], [j,n,l], [e,f] }

(j,l) < (j,a); j and l already in the same set

(j,n) < (j,f); j and n already in the same set

Iterating through constraints, applying Rule 2,

Current state of partition: { [c,h,a], [$j,n,$l], [e,f] }

(j,n) < (j,l); j,n,l already in the same set

(c,a) < (f,h); f,h not in the same set

(j,l) < (e,n); e,n not in the same set

(n,l) < (a,f); a,f not in the same set

(c,h) < (c,a); c,a,h already in the same set

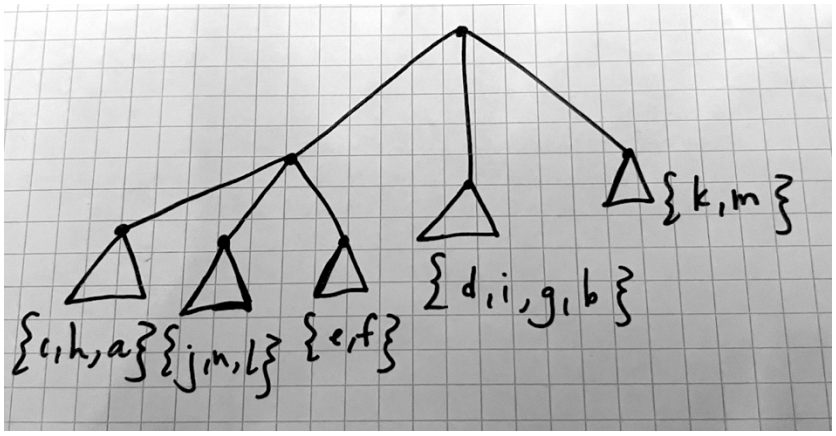
(e,f) < (h,l); h,l not in the same set

(j,l) < (j,a); j,a not in the same set

(j,n) < (j,f); j,f not in the same set

Final computed partition: { [c,h,a], [[j,n,l], [e,f] } …. (IN MEMORY) { [d,i,g,b], [k,m] }

Current state of the tree:



Recurse into set [*c,h,a*], calling BUILD(S,C):

S: *c, h, a*

C (updated to contain only those pertaining to leaves in S):

(*c*,*h*) < (*c*,*a*)

Partition Sets: {}

Iterating through constraints, applying Rule 1,

(*c*,*h*) < (*c*,*a*); append *c,h* to the same set

Update partition sets: { [*c,h*] }

No constraint applies to *a*; append as a singleton.

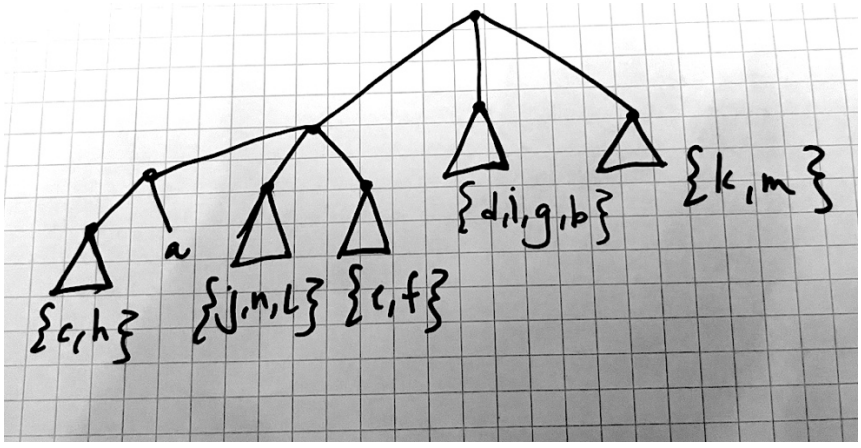
Update partition sets: { [*c,h*], [*a*] }

Iterating through constraints, applying Rule 2,

(*c*,*h*) < (*c*,*a*); *c,a* not in the same set

Final computed partition: { [*c,h*], [*a*] } …. (IN MEMORY) { [*j,n,l*] [*e,f*] ,[*d,i,g,b*], [*k,m*] }

Current state of the tree:



Recurse into set [*c,h*], calling BUILD(S,C):

S: *c, h*

C (updated to contain only those pertaining to leaves in S): no relevant constraints

As no applicable constraints, append nodes as singleton sets.

Final computed partition: { [*c*], [*h*] } …. (IN MEMORY) { [*a*], [*j,n,l*] [*e,f*] ,[*d,i,g,b*], [*k,m*] }

Recurse into set [*c*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *c*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*h*], [*a*], [*j,n,l*] [*e,f*], [*d,i,g,b*], [*k,m*] }

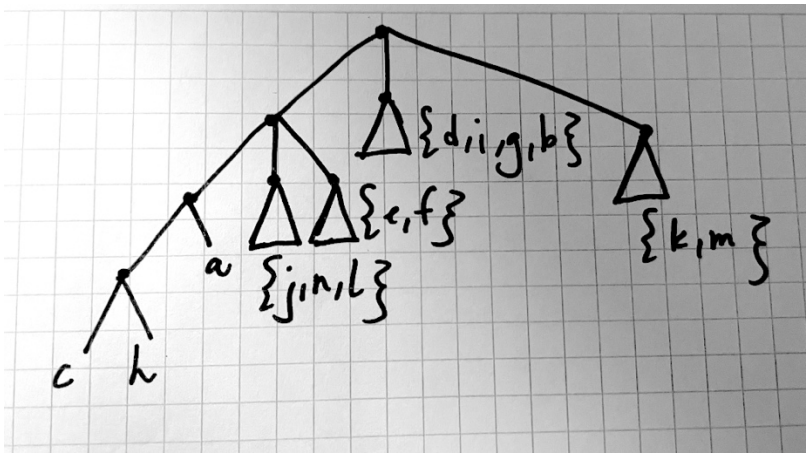
Recurse into set [*h*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *h*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*a*], [*j,n,l*] [*e,f*], [*d,i,g,b*], [*k,m*] }

Recurse into set [*a*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *a*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*j,n,l*] [*e,f*], [*d,i,g,b*], [*k,m*] }

Current state of the tree:



Recurse into set [*j,n,l*], calling BUILD(S,C):

S: *j, n, l*

C (updated to contain only those pertaining to leaves in S):

(*j*,*n*) < (*j*,*l*)

Partition Sets: {}

Iterating through constraints, applying Rule 1,

(*j*,*n*) < (*j*,*l*); append *j,n* to the same set

Update partition sets: { [*j,n*] }

No constraint applies to *l*; append as a singleton.

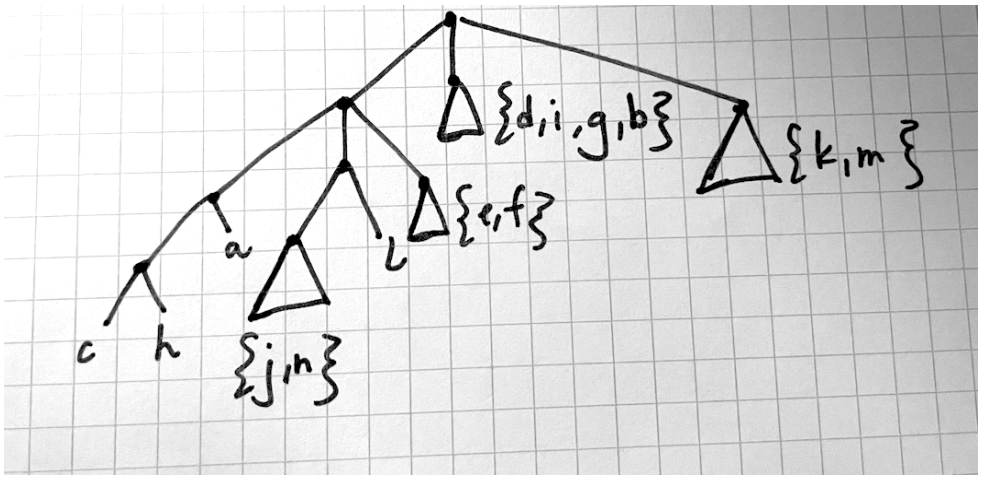
Update partition sets: { [*j,n*], [*l*] }

Iterating through constraints, applying Rule 2,

(*j*,*n*) < (*j*,*l*); *j,l* not in the same set

Final computed partition: { [*j,n*], [*l*] } …. (IN MEMORY) { [*e,f*], [*d,i,g,b*], [*k,m*] }

Current state of the tree:



Recurse into set [*j,n*], calling BUILD(S,C):

S: *j,n*

C: no relevant constraints

As no applicable constraints, append nodes to singleton sets.

Final computed partition: { [*j*], [*n*] } …. (IN MEMORY) { [*l*], [*e,f*], [*d,i,g,b*], [*k,m*] }

Recurse into set [*j*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *j*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*n*], [*l*] [*e,f*], [*d,i,g,b*], [*k,m*] }

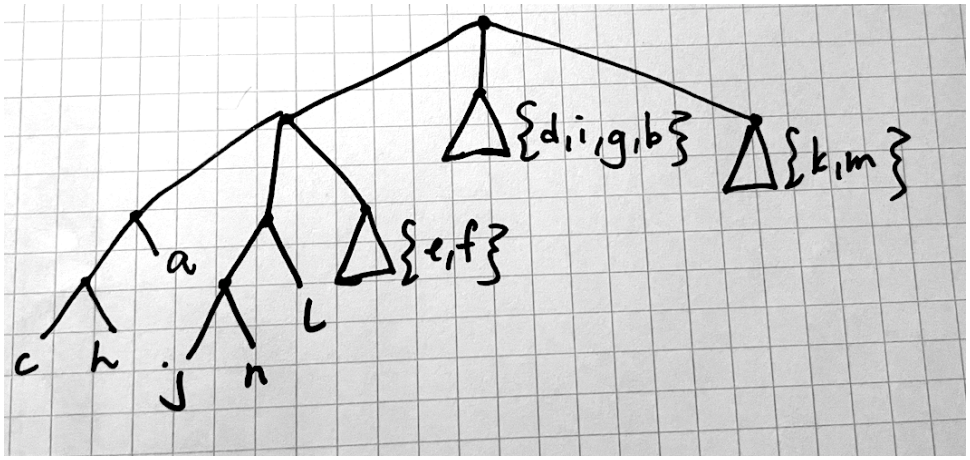
Recurse into set [*n*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node n. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*l*] [*e,f*], [*d,i,g,b*], [*k,m*] }

Recurse into set [*l*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *l*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*e,f*], [*d,i,g,b*], [*k,m*] }

Current state of the tree:



Recurse into set [*e,f*], calling BUILD(S,C):

S: *e,f*

C: no relevant constraints

As no applicable constraints, append nodes to singleton sets.

Final computed partition: { [*e*] [*f*] } …. (IN MEMORY) { [*d,i,g,b*], [*k,m*] }

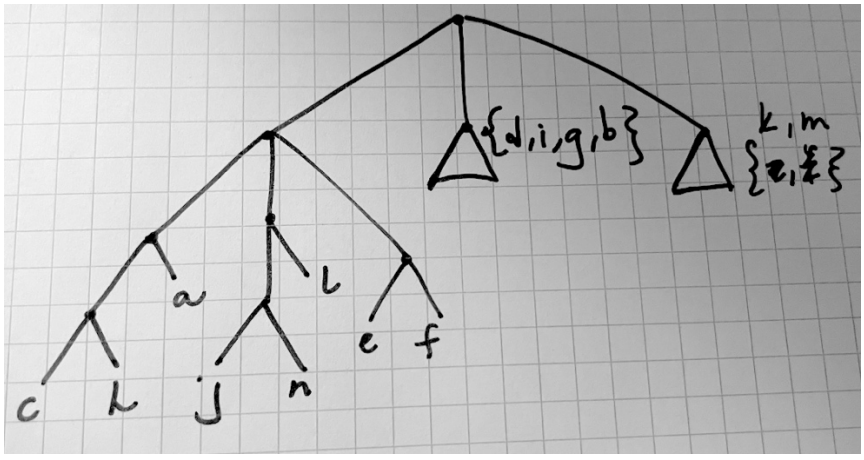
Recurse into set [*e*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *e*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*f*], [*d,i,g,b*], [*k,m*] }

Recurse into set [*f*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *f*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*d,i,g,b*], [*k,m*] }

Current state of the tree:



Recurse into set [*d,i,g,b*], calling BUILD(S,C):

S: *d, i, g, b*

C:

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

Partition Sets: {}

Iterating through constraints, applying Rule 1,

$(d,i) < (g,i)$; append d,i to the same set

Update partition sets: { [d,i] }

$(g,b) < (g,i)$; append g,b to the same set

Update partition sets: { [d,i], [g,b] }

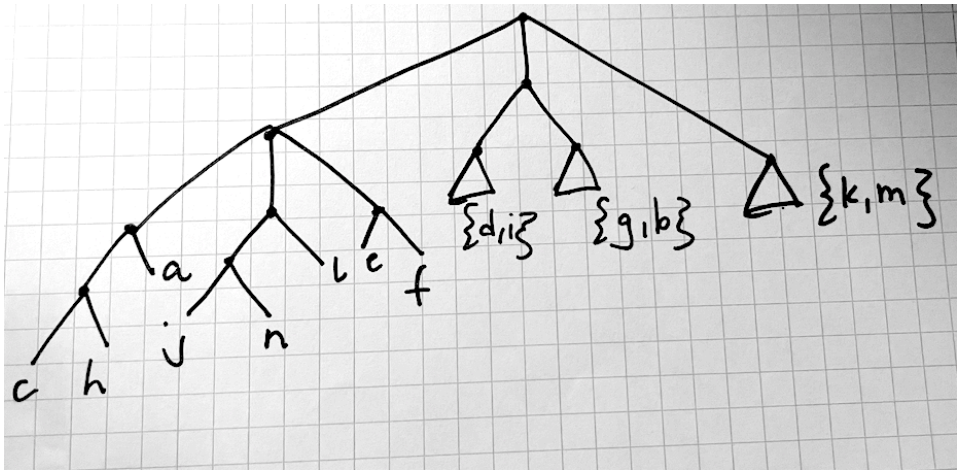
Iterating through constraints, applying Rule 2,

$(d,i) < (g,i)$; g,i not in the same set

$(g,b) < (g,i)$; g,i not in the same set

Final computed partition: { [d,i], [g,b] …. (IN MEMORY) [k,m] }

Current state of the tree:



Recurse into set [d,i], calling BUILD(S,C):

S: d, i

C: no relevant constraints

As no applicable constraints, append nodes as singleton sets.

Computed partition: { [d], [i] …. (IN MEMORY) [g,b], [k,m] }

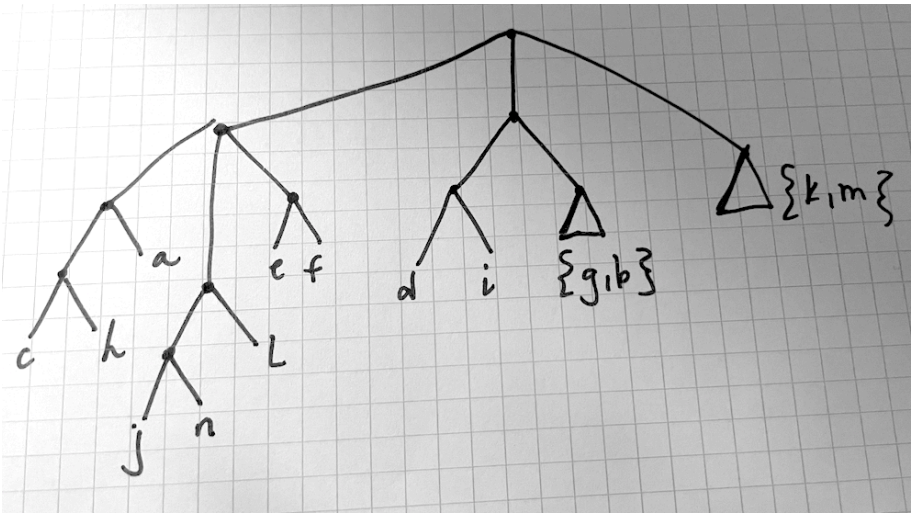
Recurse into set [d], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node d. Proceed to backtrack up the recursion tree. (IN MEMORY) { [i], [g,b], [k,m] }

Recurse into set [*i*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *i*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*g,b*] [*k,m*] }

Current state of tree:



Recurse into set [*g,b*], calling BUILD(S,C):

S: *g, b*

C: no relevant constraints

As no applicable constraints, append nodes as singleton sets.

Computed partition: { [*g*] , [*b*] …. (IN MEMORY) [*k,m*] }

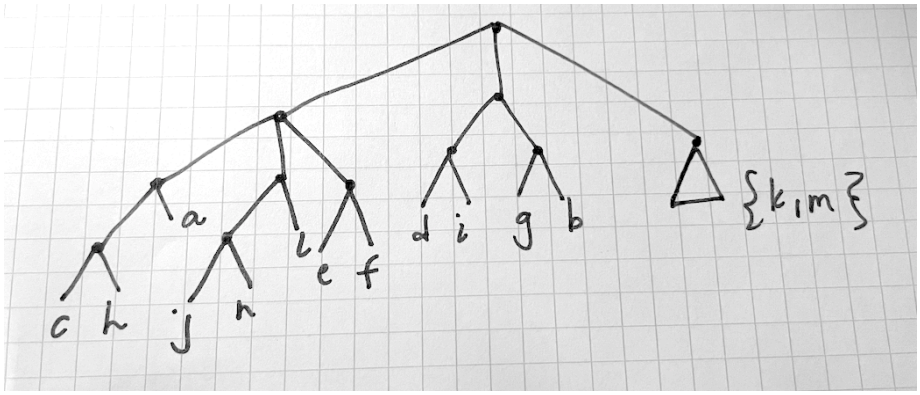
Recurse into set [*g*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *g*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*b*] [*k,m*] }

Recurse into set [*b*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *b*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*k,m*] }

Current state of tree:



Recurse into set [*k, m*], calling BUILD(S,C):

S: *k, m*

C: no relevant constraints

As no applicable constraints, append nodes to singleton sets.

Final computed partition: { [*k*], [*m*] } …. (IN MEMORY) { }

Recurse into set [*k*], calling BUILD(S,C):

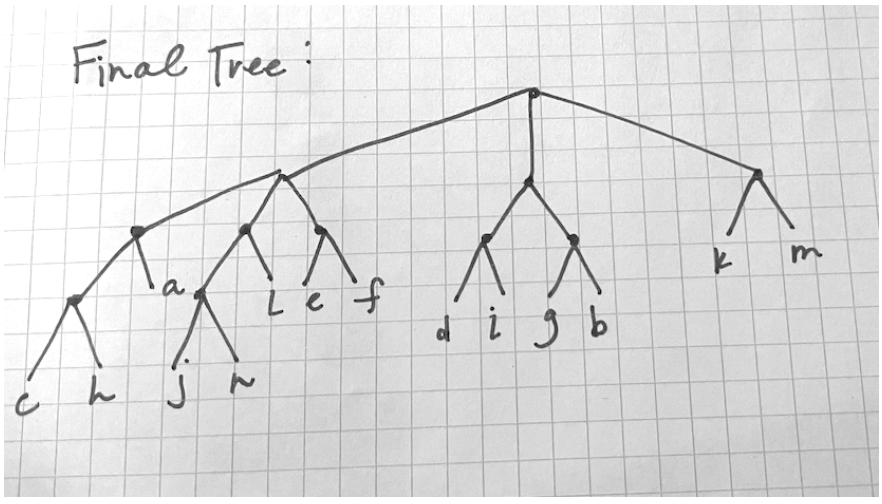
Reached base case, set contains only one leaf node. Return a tree of node *k*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [*m*] }

Recurse into set [*m*], calling BUILD(S,C):

Reached base case, set contains only one leaf node. Return a tree of node *m*. Proceed to backtrack up the recursion tree. (IN MEMORY) { [ ] }

Completed series of recursive calls; a satisfying tree exists. Assemble the final tree *T* with a new node for its root and whose children are the roots of T_m $1 \leq m \leq r$ (trees formed through recursive calls). For clarity, the state of the tree was maintained while running the algorithm.

Final tree:



d. "Reverse" the Build algorithm, i.e. design an algorithm that takes a tree with labeled leaves as an input, and produces a set of constraints of the form (i,j) < (k,l), such that when Build runs on that set, the result is (an isomorphic copy of) the input tree. Prove the correctness of your algorithm. Also, a smaller output (a set of constraints) would give you a better mark.

i. Description of the algorithm and pseudocode

The proposed ReverseBuild algorithm operates by performing a DFS search on the input tree, stopping at intermediate nodes. At each intermediate node, the algorithm first examines whether the intermediate node s has at least 2 children. If yes, a standardized procedure is followed to locate 3 suitable left, right, and upper nodes such that the constraint (left, right) < (left, upper) can be appended. An upper node is defined as a descendant of the parent of s, which is not in the same subtree as s. The upper node is found by inspecting the parent of s, and performing a DFS on its children (apart from s) to locate the first leaf node. A similar procedure is followed to find the left and right nodes, where a DFS is performed on the left and right children of s to locate a suitable leaf node (i.e. in the event that they are intermediate nodes and not immediately leafs). These internal DFS searches are performed within the larger tree traversal using a DFS helper function, DFS_helper_search(). If s has more than 2 children, this procedure is repeated for each additional child to the right such that a constraint (left, right') < (left, upper) is added.

Otherwise, if the intermediate node s has only one child, a different procedure is followed. A DFS helper search is performed to find a "left" node (i.e. a leaf that is a descendant of the single child – we term it a "left" node because the same left search procedure is followed as when 2+ children are present). Next, an upper node is found as per the standard procedure. However, as no right node is present, a secondary upper node (upper2) which is itself an ancestor of the designated upper node, must be found. This is accomplished by looking at the children of s.parent.parent (excluding s.parent itself) and performing a DFS helper search for a descendant leaf node. Once found, the constraint (left, upper) < (left, upper2) is appended.

Following this procedure traverses the entire input tree, visiting every node. The root and leaf nodes are ignored for purposes of developing constraints; only the intermediate nodes are of interest. At each intermediate node, the relevant level of hierarchy in the tree is considered, such that the resulting list of constraints produces a suitable partition at each step during the Build() procedure.

Pseudocode for ReverseBuild:

```
REVERSEBUILD(input_tree):

#Follow a DFS of the tree, stopping at the intermediary nodes and
adding constraints if relevant conditions are met

#DFS helper function
def DFS_helper_search(n):
   mark n as inspected
   if n is a leaf node:
      return n

   for node in n.children:
      DFS_helper_search(node)

   return None

#Main DFS tree traversal to generate constraints
def DFS-recursive(G, s):
```

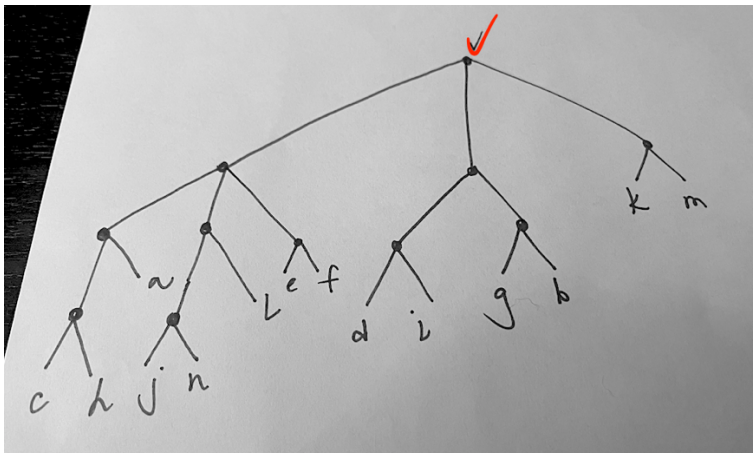
```
mark s as visited

if s !isLeafNode:
#s is an intermediary node
    if not root:
       #Find a suitable upper ancestor of the intermediate node
       if len(s.parent.leaf_children) >= 1:
            upper = s.parent.leaf_children[0]
       else:
            upper = None
            nodes_to_search = s.parent.children
            #Remove the current node from the list to search
            nodes_to_search.remove(s)
            for node in nodes_to_search:
                upper = DFS_helper_search(node)
                if upper != None:
                     break
            #No suitable upper node was found, return an error
            if upper == None:
                return Error
       #Inspect the child nodes of s
       #Check if s has more than one child
       if len(s.children) > 1:
       #Locate L and R nodes for constraint
             left = s.children[0]
             if left !isLeafNode:
                 #Find a leaf node in the left sub-tree
                 left = DFS_helper_search(leaf)
                 if left == None:
                     return Error
             #Right nodes (if more than 2 children)
             right_toiterate = s.leaf_children[1:]
             for node in right_toiterate:
                if right !isLeafNode:
                     right = DFS_helper_search(right)
                     if right == None:
                        return Error
                else:
                     right = node
                constraint = {(left,right)<(left,upper)}
                #Check the constraint is not a duplicate
                   if constraint not in constraints:
                     constraints.append(constraint)

        #If s has only one child, need to find 2 upper ancestors
        else:
            Assign a left node
            if s.parent.parent != root:
                if s.children[0] isLeaf:
                    left = s.children[0]
```

```
                    else:
                        left = DFS_helper_search(s.children[0])
                        if left == None:
                              return Error
                    #Find the second upper ancestor
                    if len(s.parent.parent.leaf_children) >= 1:
                       upper2 = s.parent.parent.leaf_children[0]
                    else:
                         upper2 = None
                         nodes_to_search = s.parent.parent.children
                         #Remove the current node from the list to
search
                          nodes_to_search.remove(s)
                          for node in nodes_to_search:
                              upper2 = DFS_helper_search(node)
                              if upper2 != None:
                                  break
                          if upper == None:
                              return Error
                    constraint = {(left,upper)<(left,upper2)}
                    #Check the constraint is not a duplicate
                    if constraint not in constraints:
                         constraints.append(constraint)

          #Continue the DFS on the children of s
          for node in s.children in Tree:
              if node is not visited:
                  DFS_recursive(Tree, node)

#Calling the main DFS function on the root to begin searching the
tree and generating constraints
    #Check if tree is empty
    if input_tree isEmpty:
        return []
    else:
        curr_node = input_tree.root
        global constraints = []
        DFS_recursive(input_tree, root)
        return constraints
```

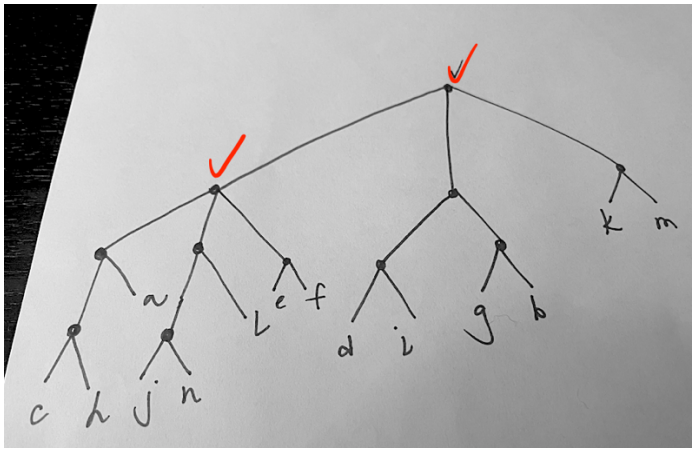
ii. Example: executing the algorithm to generate constraints for the tree from problem 2c

Given the tree produced in problem 2c, generating a set of constraints using ReverseBuild():

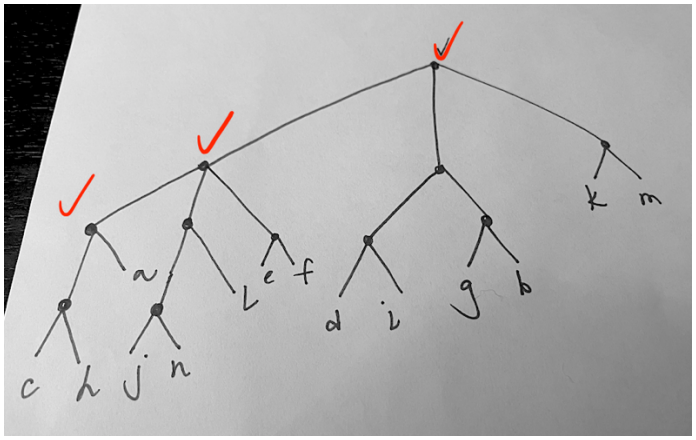
The DFS first inspects the root; no constraints are generated as the statement ‘if not root’: is not satisfied to proceed further.



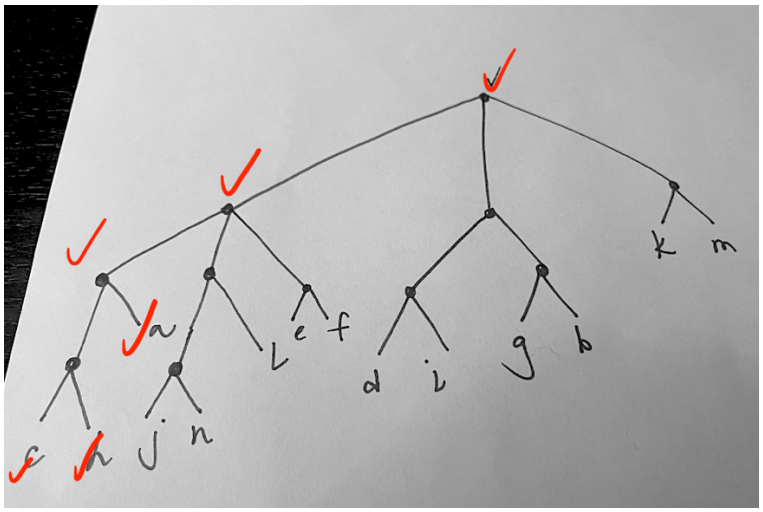
DFS helper search is executed on the next intermediate node. First, an ‘upper’ node is located for the right half of the constraint; in this case, d. Next, left and right descendants are obtained; in this case, by DFS helper search, c, and j. (c,j) < (c,d) is appended to the list of constraints. As this intermediate node has more than 2 children, another right node is found (e) and a second constraint is appended: (c,e) < (c,d).



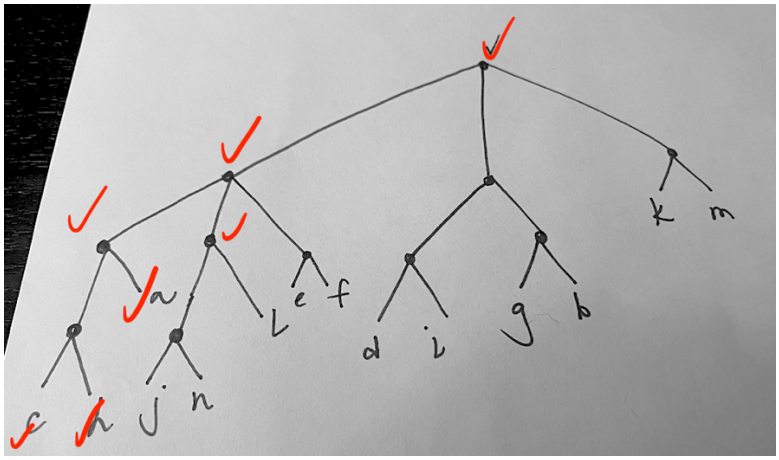
Again, upper, left, and right nodes are found using DFS helper search. A suitable upper leaf node is j. c is a suitable left node. The only suitable right node is a. (c,a) < (c,j) is appended to the constraints.



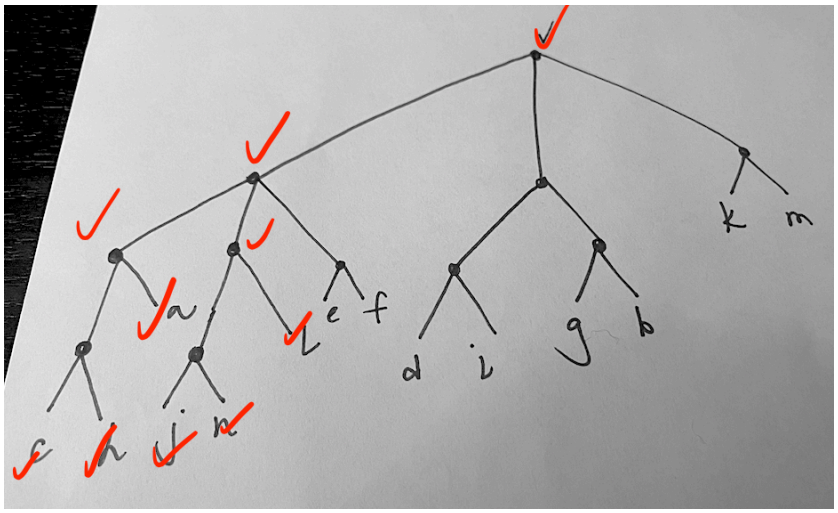
Next, by the same reasoning, (c,h) < (c,a) is appended to the constraints. As c,h , a are leaf nodes, they are marked as visited but no further action is taken.



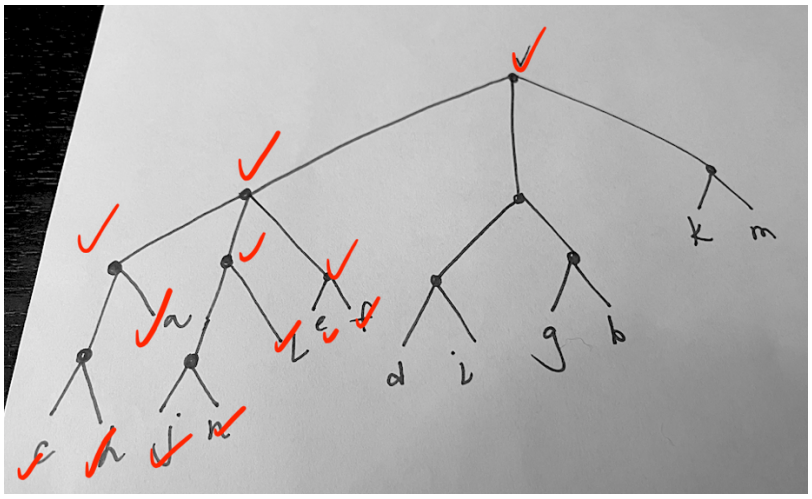
The DFS recurses back up and proceeds to search the next part of the sub-tree. A suitable upper node is e. j and l are suitable left and right nodes, respectively. (j,l) < (j,e) is added to the list of constraints.



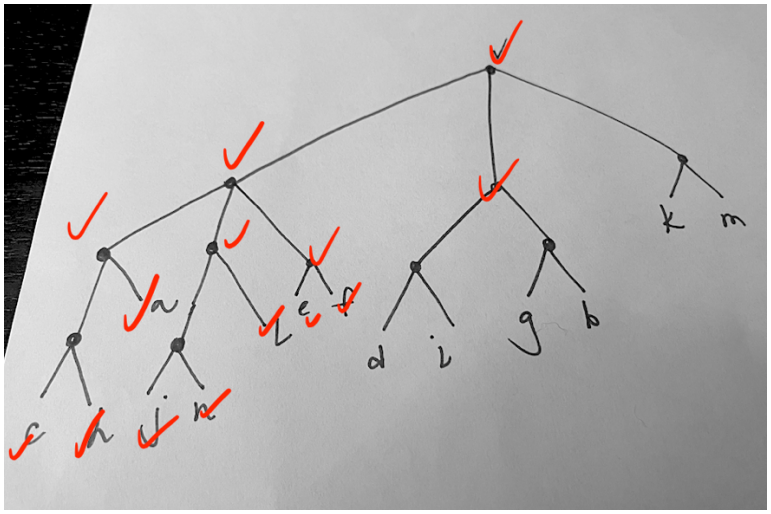
Next, upper is denoted l, and left and right are denoted j and n, respectively. (j,n) < (j,l) is added to the constraints. As j, n, and l are leaf nodes, the DFS marks these nodes as visited but no additional constraints are added.



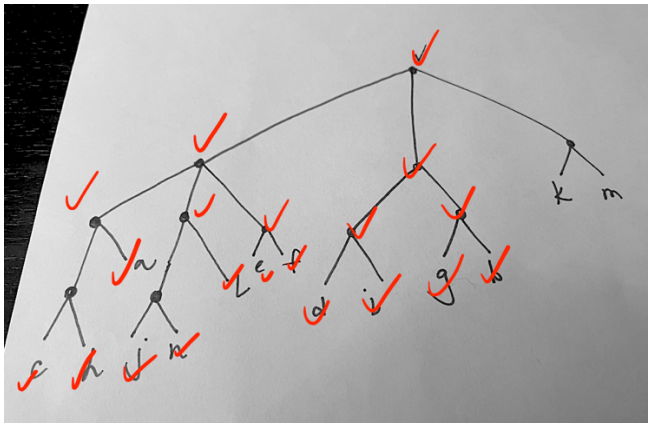
The DFS backtracks and again recurses into the third subtree. A suitable upper node is c. Left and right nodes are set to e and f. (e,f) < (e,c) is appended to the constraints. The DFS proceeds to inspect nodes e and f, but as they are leaf nodes, no further action is taken.



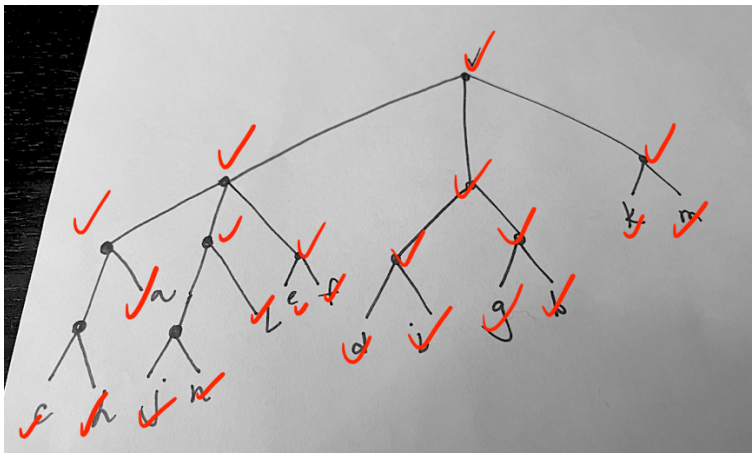
DFS backtracks and recurses into the next subtree. A suitable upper node is k, left node is d, and right node is g. We append (d,g) < (d,k) to the constraints.



It follows that (d,i) < (d,g) and (g, b) < (g, d) are added to the constraints.



Lastly, we examine the final subtree. A suitable upper node is c, and k and m are the left and right nodes, respectively. Thus (k, m) < (k,c) is added to the constraints.



Thus, the final generated list of constraints is:

(c,j) < (c,d)

(c,e) < (c,d)

(c,a) < (c, j)

(c,h) < (c,a)

(j, l) < (j, e)

(j, n) < (j, l)

(e,f) < (e,c)

(d,g) < (d, k)

(d, i) < (d, g)

(g, b) < (g, d)

(k, m) < (k,c)

Running Build() on this set of constraints returns an isomorphic copy of the input tree.

iii. Proof of correctness

The correctness of a depth first search can be established by proving 2 claims: first, that DFS is called exactly once for each node in the tree, and second, that the main for loop (for node in s.children) is executed exactly once for each edge in the graph. The first claim can be proved by referring to the 'visited' marked at each node; DFS is called on the node iff node.visited == False, and node.visited is set to True once DFS has been executed at that node. Thus, DFS cannot be called more than once on any single node as the Boolean update from node.visited = False to node.visited = True may only occur once. Further, the for loop "for node in s.children" ensures that DFS is called on each node at least once, as the search begins at the root of the tree, of which all remaining nodes in the tree are children.

The second claim can be proven by referring to the validity of Claim 1 (i.e. that DFS is called once for each node in the tree), and observing that the loop "for node in s.children" is executed once for each outgoing edge to a child of s. By virtue of the correctness of a DFS search we can be confident that each node in the supplied input tree will be examined at least once to determine whether a new constraint must be appended to the constraints array.

The validity of the constraints themselves can be established by observing the relationship between intermediate nodes and their position in the tree. As each intermediate node is visited by the DFS search, a constraint is appended for each level of hierarchy (i.e. depth) in the tree. As the algorithm appends at least one constraint for *each* intermediate node, ancestral information is conveyed for each level of hierarchy in the tree. This ensures that the partition will remain valid throughout the duration of the Build() algorithm, from the initial partitioning step until the singleton leafs are reached. To generate each constraint, a helper DFS function is used to find suitable left, right, and upper nodes; by virtue of the correctness of DFS, the correctness of finding a suitable node is established (i.e. the first leaf node found by DFS is used). Further, the (left, right) < (left, upper) constraint structure ensures that the intermediate node in question is "cradled" by a lowest common ancestor immediately above it; as the intermediate node itself is the LCA of the left and right nodes immediately below it, and the intermediate node's parent is the LCA of the left and upper node, the intermediate node and the upper LCA are always separated by a single edge. As the DFS proceeds, all edges of the tree will be similarly addressed such that a constraint is appended to represent each intermediate node and its parent (i.e. intermediate node < parent). As no intermediate node is left unaddressed and each constraint concerns the LCA relationship between an intermediate node and its immediate ancestor, the algorithm will convey the complete tree hierarchy from the root to the leaves. Further, as the search for a 'right' descendant node proceeds through all of the intermediate node's non-leftmost children (i.e. if 2+ children) we account for constraints needed to group more than 2 "sibling" nodes in a partition. As such all of the leaf nodes are contained in at least one constraint.

Report References (code references are included as comments)

A. V. Aho, Y. Sagiv, T. G. Szymanski, J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. SIAM Journal on Computing, 10:405—421, 1981.

Daniel Jurafsky & James H. Martin, 2020. *Hidden Markov Models.* Accessible at: https://web.stanford.edu/~jurafsky/slp3/A.pdf

Dantchev, Stefan. Bioinformatics Lecture Slides, Maximum-Likelihood for HMM and Expectation-Maximization Algorithm for HMM.

Lyngso, Rune. Hidden Markov Models. http://www.stats.ox.ac.uk/~mcvean/DTC/STAT/Lectures/Weds_wk2/hidden_markov_models.pdf