Bioinformatics Assignment

AMC: Z0132271

1 Question 1

1.1 Part A

1.1.1 Algorithm

The algorithm takes input of a sequence of n observations produced by some hidden Markov model (HMM), and outputs the corresponding sequence of T states, where $T \geq n$ (as some states may be silent). The succeeding process describes a method to generate a state sequence which maximises the probability of these observations, working atop a lattice of states analogously to the classic Viterbi algorithm. This lattice consists of layers of k states, where a new layer of the lattice is appended at the outermost dimension at every iteration i. At each lattice layer, a process of dynamic programming is utilised to pick the most likely state (and whether it emits or stays silent). This is done through three distributions: V, *ptr_states*, and *ptr_obs*.

Given the current layer, we consider each possible state s and determine the corresponding value in V. This is done through two maximisation steps. The first maximisation step determines the most likely transition to the current state s at iteration i (of value $t_{i,s}$) from some previous state. Dynamic programming is applied to achieve this, maximising over the values of V calculated for states in the preceding layer (and the transition matrix M). If we are at the first layer of the lattice (i.e. $i = 1$), meaning the preceding layer only consists of the artificial start state, the associated transition value $t_{1,s}$ for each state s is calculated as simply the initial probability of starting in that state s, namely π_s. If we are not in the initial layer, this transition value $t_{i,s}$ is determined as the maximum value (over all states r in the preceding lattice layer $i-1$) of the product between the associated value in V of that state, i.e. $V_r\left(i-1\right)$, and the transition probability between that state and the current state $m_{r,s}$: $t_{i,s} = \max_{r \in S}\left[V_r\left(i-1\right) m_{r,s}\right]$.

Once $t_{i,s}$ has been determined, with this maximal value coming from the specific preceding state r' (which would be the artificial start state if we are in the first lattice layer), we move onto the next maximisation step: determining the emission value $e_{i,s}$. This takes the maximum value of the probabilities of the current state s not emitting any observation, i.e. $e_s\left(\varepsilon\right)$, or that of s emitting the next symbol in the observation sequence x_j: $e_s\left(x_j\right)$.

Over this process, we maintain two pointers: a state pointer *ptr_states*, and an observation pointer *ptr_obs*. Beginning with the artificial start state, the state pointer references the preceding state r' from which we transitioned to the current state s, and the observation pointer keeps track of the observation emitted by each state, or ε for silence. Each element of these pointers corresponds to an element of the V distribution. When we are computing V in the current state s, we set the associated value of *ptr_states* equal to the predecessor r' from which we transitioned (when we maximised over preceding V values). Additionally, when an emission value is calculated and a state is decided to either be silent, or emit the current observation, this observation (or ε) is appended to the observation pointer so that we can always deduce the next observation in the sequence to be

emitted (when deciding whether a state emits or is silent). This is necessary to maintain as an each iteration in the algorithm doesn't necessarily correspond to a position in the observation sequence (as we may not emit at every iteration). Namely, when considering the emission value of the current state s, we use the observation pointer to deduce which state would be emitted at this point if s doesn't stay silent. This observation is deduced by traversing back through *ptr_obs* and finding the last state that was emitted (this could also be implemented simply by keeping a counter of the last emitted observation, which we update every time an emission occurs).

Once both local maxima of the transition value $t_{i,s}$ and emission value $e_s(x_j)$ are determined for the state s in the current layer i of the lattice (at iteration i), the value of the distribution V is updated such that we can maximise over this layer of states at the next iteration: $V_s(i) = e_s(x_j)\, t_{i,s}$. This is repeated over all of the states in the current layer i of the lattice. The sequential approach to dynamic programming, where we rely upon the computed value of V at iteration $i-1$ to deduce the following values at iteration i, is continued until the point at which the final observation x_L is selected to be emitted by some final state s_T, meaning the observation sequence is complete.

After completing this forward pass, the backward pass begins to construct the state sequence from the distributions V, *ptr_states*, and *ptr_obs*. First, we inspect the end of *ptr_obs*, finding the final element of the outermost dimension—i.e. the last row if we represent each of these final distributions as a $n \times T$ matrix, where n is the number of states and T the number of iterations it took to conclude the forward step. Within this row, we find the state s_T from which the final observation x_L was emitted. If multiple states correspond to x_L, we check the corresponding row of V and pick the state s_T with the maximum V value. This begins the process of reconstructing the state sequence, which we iteratively do from the end to the start: $s_T \to s_1$. Namely, upon determining the final state s_T, we inspect the corresponding value of *ptr_states* to determine the state s_{T-1} from which we transitioned to s_T. This state is selected as the next last state in the sequence, and we similarly query *ptr_states* again to find its predecessor s_{T-2}. This process is continued until the artificial start state is reached in *ptr_states*, which we discard and output the constructed state sequence $s_T \to s_1$.

1.1.2 Correctness and Runtime

The presented algorithm must output (one of) the maximum likelihood sequence of states corresponding to the given sequence of observations due to the Markov properties of dependence—meaning the dynamic programming approach of maximum likelihood over local computations must generate a maximum likelihood sequence at the global level. As we are dealing with a HMM, we are permitted two assumptions. Firstly, the probability of a particular state appearing at the current point in the sequence depends only upon the previous state: $P(s_i | s_1 \to s_{i-1}) = P(s_i | s_{i-1})$. This is known as the Markov assumption. The second assumption is that the probability of some observation x_j depends only upon the state s_i which emitted it, not any other state or observation: $P(x_j | s_1 \to s_T, x_1 \to x_n) = P(x_j | s_i)$. This is known as output independence.

As we know the sequence of observations being input into the algorithm was produced by a HMM, we know that the Markov assumption holds, and so when deducing which state occurs at point i in the state sequence, we need only consider the state in the preceding position $i-1$. Given the second assumption, we know the emission of some sole observation x depends only upon the the set of states which could have produced that specific observation. Thus, to determine the state which emitted x we can simply consider the emission probabilities over the set of states, and pick the most likely. This means

that finding the highest likelihood state-observation pair at the local level (i.e. through considering each state/observation at each iteration) results in finding the global sequence with maximum likelihood.

The Viterbi algorithm, upon which this implementation is based, relies upon both of these assumptions. Namely, when we are deducing the next state in the state sequence, this depends only upon the likelihood of this state following the preceding state (the Markov assumption), and the likelihood of this state emitting the current observation (output independence). To deduce state transitions, the presented algorithm utilises the same forwards and backwards dynamic programming method as the Viterbi algorithm, and thus generates the correct maximum likelihood state transitions at a local level—generating a maximum likelihood sequence at the global level—due to the Markov assumption. For the case of emissions, the assumption of output independence still holds. Therefore, we know that the likelihood of a given observation in the sequence depends only upon a given state in the state sequence which emitted it. Similarly, if no observation is emitted at some point within the state sequence, this depends only upon the current state (that stayed silent). Thus, we can deduce whether a state emits the current observation, or stays silent, on a local per-state basis as the forward pass is underway, and this will correspond to the generation of a global maximum likelihood sequence of emitting and silent states corresponding to the given observation sequence.

Therefore, as the local maximisation of both components (i.e. the transition and emission values) upon which the selection of states is based generates the maximum likelihood global sequence, the algorithm is correct.

When considering the complexity of the implemented approach we must first consider the complexity of the Viterbi algorithm, upon which the process is based. Viterbi has a time complexity of $O\left(k^2 n\right)$, and space complexity also of $O\left(k^2 n\right)$, where k is the number of states of the HMM and n is the number of observations in the input sequence. The presented algorithm doesn't have a set number of iterations over which a state sequence is guaranteed to be output—as is true of the n iterations needed for the Viterbi algorithm—as upon commencing there is no way to tell the number of silent states of the output sequence. However, the algorithm must terminate (due to the diminished likelihood of the current state being silent if the preceding state was silent) to generate a sequence of states $s_1 \rightarrow s_T$, where $T \geq n$. Thus, the number of iterations must be some multiple m of the number of iterations of Viterbi (i.e. $T = n * m$), and so linear in the length of the observation sequence n. At each iteration, the algorithm implements a double loop over the set of k states, similarly to Viterbi, and thus is quadratic in the number of states (k^2). However, for each the states in the innermost loop, we additionally loop over the probability of emitting or silence, which for each state is 2 variables, so we have $2k^2$. Overall, this means the time complexity of the algorithm is $O\left(2k^2 nm\right)$, or simply $O\left(k^2 nm\right)$. When it comes to the space complexity, the structure of the V distribution is identical to that of Viterbi (as we maximise over the emission value before storing the value of V), simply longer in the outermost dimension, as the number of iterations is increased to $T = n * m$. The same complexity is witnessed thrice for V, *ptr_states*, and *ptr_obs*. Hence, the space complexity is $O\left(3k^2 nm\right)$, or simply $O\left(k^2 nm\right)$.

1.2 Part B

1.2.1 Expectation-Maximisation Algorithm Description

The expectation-maximisation (EM) algorithm is a method through which, on input of a sequence of T symbols $X = (x_1, \ldots, x_{T-1})$ and the number of hidden states k, the

parameters of the corresponding Hidden Markov Model (HMM) can be estimated. The parameterisation $\theta = \{M, E, \pi\}$ consists of the transition (M), emission (E), and initial (π) distributions of the HMM, and is deduced through maximum likelihood estimation over the probability $P_\theta(X)$ of the observed sequence. This is conducted iteratively over two steps. Initially, the expectation step (E-step) computes the loss function L_θ of the joint probability of the observations X (and hidden states S) under the current parameters θ. This is followed by the maximisation step (M-step), where new parameters θ^* are generated that maximise L_θ, thus increasing the likelihood of X under this parameterisation.

EM can be implemented through the Baum-Welch algorithm, splitting the E-step into a forward and backward phase. The forward phase recursively constructs the function α over the states $i \in S$ and observations $x \in X$, considering the sequence from start to finish $x_0 \to x_{T-1}$. Each element $\alpha_i(t+1)$ represents the probability of the sequence of symbols $x_0 \to x_{t+1}$ given it concludes with the state i corresponding to the observation x_{t+1}. The base of recursion $\alpha_i(0)$ for each state i is simply the probability of seeing that state correspond to the initially observed symbol x_0: $\alpha_i(0) = e_i(x_0)\pi_i$. Recursive steps are then computed as the product of the emission probability $e_i(x_{t+1})$ and the sum (over states $j \in S$) of the preceding recursive call (i.e. $\alpha_j(t)$), and the probability of the transition $i \to j$ (m_{ij}):

$$\alpha_i(t+1) = e_i(x_{t+1}) \sum_{j \in S} \alpha_j(t) m_{ij}. \tag{1}$$

The backwards phase constructs the distribution β over the states $i \in S$ and observations $x \in X$, recursively from the end of the sequence to the start $x_{T-1} \to x_0$. Each element $\beta_i(t)$ represents the probability of the sequence $x_{t+1} \to x_{T-1}$ given it is preceded by the state i emitting observation x_t. At the base of recursion, this probability is simply one: $\beta_i(T-1) = 1$. This is computed as the sum (over states $j \in S$) of the preceding recursive call (i.e. $\beta_j(t+1)$), the probability of j emitting the observation x_{t+1} ($e_j(x_{t+1})$), and the probability of transitioning $i \to j$ (m_{ij}):

$$\beta_i(t) = \sum_{j \in S} \beta_j(t+1) e_j(x_{t+1}) m_{ij}. \tag{2}$$

Together, the forward and backward phases constitute the E-step, then the α and β distributions are maximised over in the succeeding M-step. This consists of three stages. First, we compute the probability distribution $\gamma_i(t)$ of each state $i \in S$ emitting the observation x_t (at sequence position t) given the entire observation sequence $x_0 \to x_{T-1}$:

$$\gamma_i(t) = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j \in S} \alpha_j(t)\beta_j(t)}. \tag{3}$$

Secondly, we compute the distribution $\xi_{ij}(t)$ of the joint probability of each state $i \in S$ emitting x_t at position t and another state $j \in S$ emitting x_{t+1} at $t+1$, given all observations $x_0 \to x_{T-1}$. This is computed based upon α and β, and the transition and emission matrices computed during the previous iteration's M-step:

$$\xi_{ij}(t) = \frac{\alpha_i(t)\beta_j(t+1)m_{ij}e_j(x_{t+1})}{\sum_{p \in S}\sum_{q \in S} \alpha_p(t)\beta_q(t+1)m_{pq}e_q(x_{t+1})}. \tag{4}$$

To conclude the M-step, the values of the transition, emission, and initial matrices are updated using γ and ξ. The initial distribution π^* is calculated by employing γ on each state and the initial position: $\pi_i^* = \gamma_i(0)$. Elements of the transition matrix $m_{ij} \in M^*$

(for each pair of states $i, j \in S$) are updated to the ratio between the total probability of transitioning $i \to j$ over any two consecutive observations $x_t \to x_{t+1}$ at positions $t \to t+1$, and the total probability of being in i over all positions $t = 0 \to T-1$:

$$m_{ij}^* = \frac{\sum_{t=0}^{T-2} \xi_{ij}(t)}{\sum_{t=0}^{T-2} \gamma_i(t)}. \tag{5}$$

Finally, each element of the emission matrix $e_i(x') \in E^*$ is updated to the ratio between the total probability of being in state i over all observation sequence positions $t = 0 \to T-1$ and that state x_t having the value x', and the same probability over any value of x_t:

$$e_i(x') = \frac{\sum_{t=0}^{T-2} \gamma_i(t)\, x_t == x'}{\sum_{t=0}^{T-2} \gamma_i(t)}. \tag{6}$$

It is proven that under the new parameter set $\theta^* = \{M^*, E^*, \pi^*\}$, the probability $P_\theta(X)$ of the observed sequence is at least as high as under the preceding parameters. Thus, this process is repeated iteratively until equality—the point of convergence—where the likelihood of X is maximised.

1.2.2 Implementation Validation

The first validation sequence will test the algorithm on the most basic, predictable input: a sequence of n observations where each observation is the same. It is clear the output that should correspond to an input of any number of states k. Each row of the $k \times k$ transition matrix should be the same, and each element within said row should be the same, as there is an equal likelihood of transitioning between any states due to the observation sequence being completely uniform. For the same reason, the initial distribution should infer an equal probability to starting in each of the states (as there is no way to discern between them given the uniform sequence). Finally, the emission matrix should be one one dimensional (due to there only being a single letter in the observation alphabet), with the probability of each state being one (as there is only one possible observation for each state to emit). Upon input of this example into the implemented algorithm (e.g. $obs =$ '$aaaaaaaa$', $k = 5$), the associated output is a $k \times k$ transition matrix of identical values (e.g. 0.2 in the case of $k = 5$), a $k \times 1$ initial matrix of identical values (again 0.2), and a $1 \times k$ emission with each element having value 1. This validates that the algorithm can correctly resolve a uniform sequence of identical observations.

The next test case will focus upon the inputs $obs =$ '$abababa$', $k = 2$. As there is only two states, the observation alphabet only consists of two symbols ('a' and 'b'), and all observation transitions are 'a' $\to$ 'b' or 'b' $\to$ 'a', it is clear upon what the parameters should converge. Due to presence of only two possible observation transitions, and two states, the transition matrix should indicate the only possible state transitions are $0 \to 1$ (corresponding to 'a' $\to$ 'b') and $1 \to 0$ (for the inverse). The transition matrix generated by this implementation achieves this through setting $m_{ij} = 1$ for $i = 1, j = 0$ and $i = 0, j = 1$, and all other transitions having zero probability. As there is only two observation symbols, and two states, the emission matrix should represent that one of these states can emit only 'a' and the other only 'b'. This true of the output of the implemented algorithm, which sets $e_0(\text{'}b\text{'}) = 1$ and $e_1(\text{'}a\text{'}) = 1$, and keeps all emission probabilities zero. Finally, since all patterns of observations within the sequence are always of the form 'a' $\to$ 'b' $\to$ 'a' $\to \ldots$, it is clear that the initial distribution over the two states should signify that the sequence always starts with an 'a'. Since in the parameters thus far state

1 has been designated as the sole emitter of this observation, the initial distribution of this implementation correctly demonstrates this by setting the initial probability of state 1 equal to 1 (and state 0 has zero probability). Thus, the solution produces a valid output to this problem; in fact, this argument should hold true for any sequence of the form '$abcd\ldots$' $* L$ (i.e. there is a pattern of distinct observations which is repeated through the whole sequence), validating the solution further through confirming this is the case for the algorithm's outputs.

2 Question 2

2.1 Part A

The algorithm $BUILD$ presented by Aho et al. focuses on the algorithmic goal of reconstructing a tree T that satisfies an input set of constraints C. The lowest common ancestor of two nodes x and y, denoted $a = LCA(x, y)$, is a node a such that no proper descendant of a (i.e. no node b with a as an ancestor where b is not a) which is an ancestor of both x and y. This concept is used to formulate constraints for the structure of a tree; the constraint $(i, j) < (k, l)$ over the set of leaves $\{i, j, k, l\}$ specifies that the node $LCA(i, j)$ is a proper descendant of $LCA(k, l)$.

The $BUILD$ algorithm implements a recursive process, upon input of a set of nodes S and a set of constraints C (of the aforementioned structure) over these nodes. The base of the recursion is where S only contains a single node; in this case the algorithm outputs the singleton tree T purely consisting of this node as the root. Otherwise, we compute a partition upon the nodes of S with respect to the constraints C. A partition π_C is the subdivision of a set of nodes into subsets $S_1, S_2, \ldots, S_r$ such that the descendants of each child m of the root node of T constitutes the set S_m. To satisfy C, for each constraint $(i, j) < (k, l) \in C$ the corresponding partition $\pi_C = S_1, \ldots, S_r$ must satisfy two conditions. Firstly, both i and j must reside within the same set. Secondly, if k and l lie within the same set, this must imply all nodes i, j, k and l lie within the same set. All sets in the partition must satisfy these conditions; namely, no two nodes may reside within the same set unless specified by either of the prior rules.

For a tree T to exist satisfying C, we must be able to find a satisfactory partition $\pi_C = S_1, \ldots, S_r$ where $r \geq 2$—as the existence of at least two subsets to recurse into is a necessary condition when constructing a tree where each non-leaf node has at least two children. Thus, after computing the partition we must check it contains a minimum of two sets; if not, we output a null tree.

Once a sufficient partition π_C has been constructed, for each constituent set $S_m \in \pi_C$ we generate a corresponding subset of constraints $C_m \subseteq C$ such that C_m contains constraints only involving the nodes of S_m. We then recurse, implementing the algorithm upon the inputs S_m and C_m for all $1 \leq m \leq r$. If a tree output by any of these lower recursions is *null*, the null tree is output at the current level of recursion. Otherwise, on construction of the collection of non-null trees $T_1, \ldots, T_r$, we output the composite tree T consisting of a new node as the root with r children, where each child node $1 \leq m \leq r$ is the root of the tree T_m (with the corresponding full tree expanded below it).

Thus, on conclusion of the highest level of recursion one of two outputs is observed: the tree T with leaves S and a structure satisfying the constraints C, or the null tree, indicating no tree exists satisfying the given constraints.

2.2 Part B

Algorithm 1 Question 2B: Compute $\pi_C = S_1, S_2, \ldots, S_r$

Input: constraint set C upon node set S

Output: partition π_C

initialise collection of sets $\pi_C \leftarrow \emptyset$
if C is empty **then**
  return π_C = the collection of singleton sets each containing one node from S
else
  for all constraints $(i, j) < (k, l) \in C$ **do**
    if i is in some set S_m **and** j is not in any set **then**
      allocate j to S_m
      add S_m to the collection of sets π_C
    else if j is in some set S_m **and** i is not in any set **then**
      allocate i to S_m
      add S_m to π_C
    else if i is in set S_m **and** j is in a different set S_n **then**
      merge S_m and S_n to form a new set $S_l = S_m \cup S_n$ within π_C
    else if neither i or j is in any set **then**
      create a new set $S_m = \{i, j\}$
      add S_m to π_C
    end if
  end for
  for all nodes $n \in S$ which are not in any set **do**
    create a new singleton set $S_m = \{n\}$
    add S_m to π_C
  end for
  for all constraints $(i, j) < (k, l) \in C$ **do**
    if k and l are both in set S_m **and** i and j are in a different set S_n **then**
      merge S_m and S_n to form a new set $S_l = S_m \cup S_n$ within π_C
    end if
  end for
  return π_C = the collection of all remaining sets
end if

2.3 Part C

Figure 1: Step 1

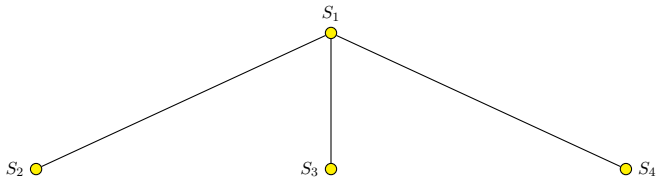
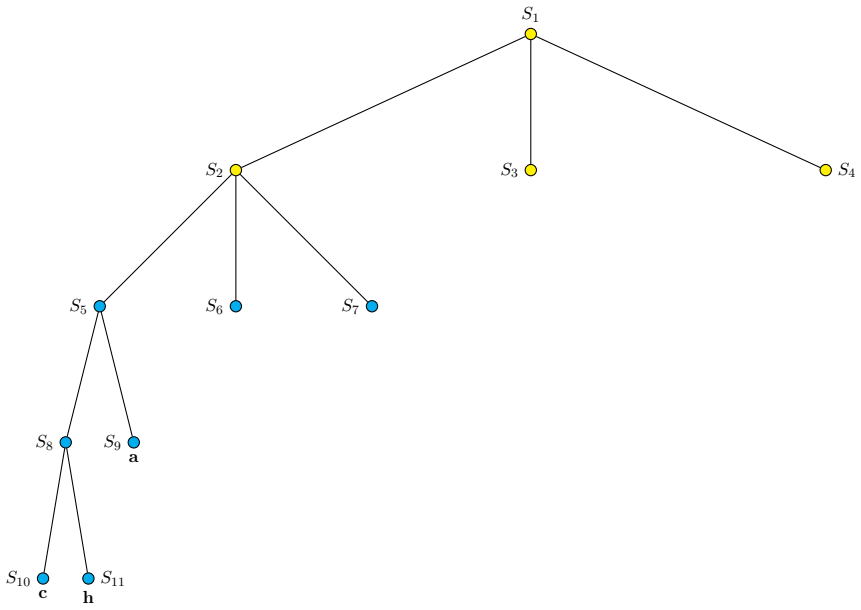


Figure 2: Step 2

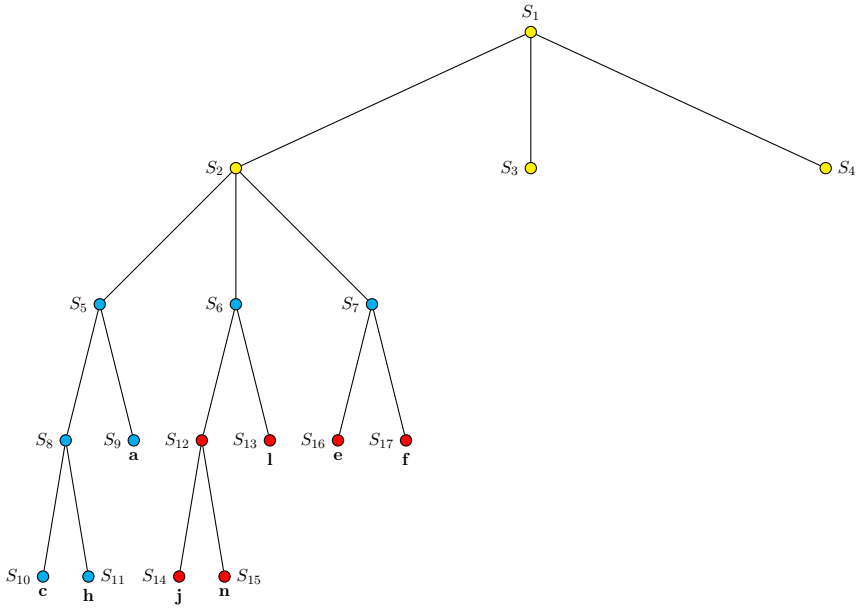


Firstly, we shall label each constraint such that they can subsequently be referred to by index only:

1. $(e, f) < (k, d)$
2. $(c, h) < (a, n)$
3. $(j, n) < (j, l)$
4. $(c, a) < (f, h)$
5. $(j, l) < (e, n)$
6. $(n, l) < (a, f)$
7. $(d, i) < (k, n)$
8. $(d, i) < (g, i)$
9. $(c, l) < (g, k)$
10. $(g, b) < (g, i)$
11. $(g, i) < (d, m)$
12. $(c, h) < (c, a)$
13. $(e, f) < (h, l)$
14. $(j, l) < (j, a)$
15. $(k, m) < (e, i)$
16. $(j, n) < (j, f)$

The initial inputs into the top layer of recursion are the set of nodes $S = \{a, b, \ldots, n\}$ and the set of constraints $C = \{1, 2, \ldots, 16\}$. We will refer to these as S_1 and C_1. The first partition we compute is π_{C_1}. Following the fist condition outlined in Section 2.1 and detailed in Algorithm 1 lines 5–18, we generate the initial partition sets $\pi_{C_1} = \{e, f\}, \{c, h, a, j, n, l\}, \{d, i, g, b\}, \{k, m\}$. The second condition (lines 23–26) means we must compute merges, generating the partition $\pi_{C_1} = \{a, c, e, f, h, j, l, n\}, \{b, d, g, i\}, \{k, m\}$.

Figure 3: Step 3



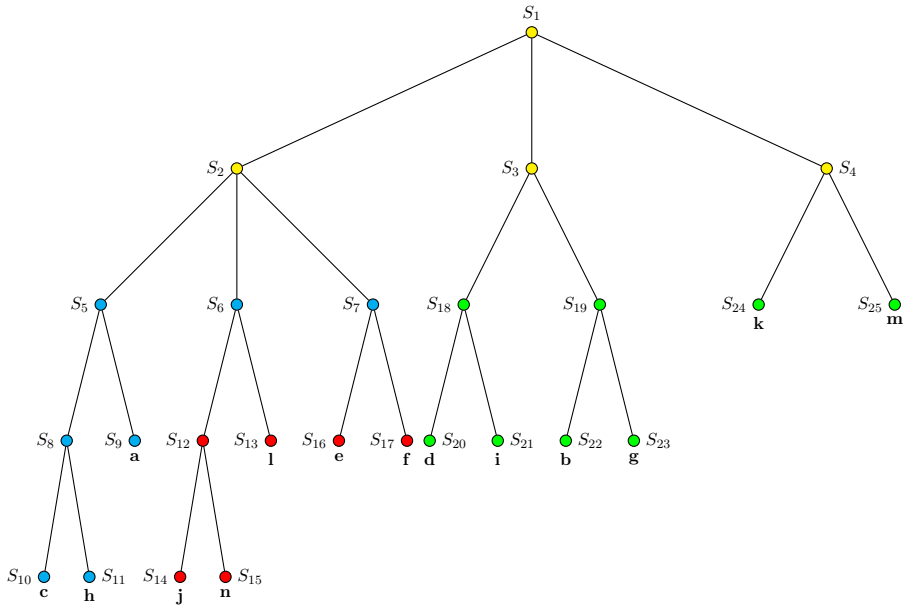
We will denote these subsets S_2, S_3, and S_4. This process is shown through the yellow nodes in Figure 1.

Now we recurse into the subset $S_2 = \{a, c, e, f, h, j, l, n\}$ with the corresponding constraints subset $C_2 = \{2, 3, 4, 5, 6, 12, 13, 14, 16\}$, applying $BUILD(S_2, C_2)$. This recursive step produces the partition $\pi_{C_2} = \{a, c, h\}, \{j, l, n\}, \{e, f\}$. We will denote these as S_5, S_6, and S_7. Next we recurse further into $S_5 = \{a, c, h\}$ with $C_5 = \{12\}$. This generates the partition $\pi_{C_5} = \{c, h\}, \{a\} = S_8, S_9$. When recursing into S_8 we find $C_8 = \emptyset$ and so output the partition $\pi_{C_8} = \{c\}, \{h\} = S_{10}, S_{11}$, thus meaning the nodes c and h are the leftmost deepest leaves. We can also see that S_9 is a singleton set, and is thus the leaf node a at the above level of recursion. This is visualised through the blue nodes in Figure 2.

As we go up the levels of recursion, we next need to compute $BUILD(S_6, C_6)$ where $S_6 = \{j, l, n\}$ and $C_6 = \{6\}$ to generate the partition $\pi_{C_6} = \{j, n\}, \{l\} = S_{12}, S_{13}$. We first recurse into $BUILD(S_{12}, C_{12})$, where $C_{12} = \emptyset$, resulting in two singleton sets S_{14} and S_{15} indicating j and n are leaves at this level. On observation of the singleton set S_{13} at the level above we find that l is a leaf here. The next lowest recursion, $BUILD(S_7 = \{e, f\}, C_7 = \emptyset)$, gives the singleton sets $S_{16} = \{e\}$ and $S_{17} = \{f\}$, indicating e and f are leaves at this level. Figure 3 demonstrates this process through the red nodes.

Next we recurse back to the top level, applying $BUILD(S_3 = \{b, d, g, i\}, C_3 = \{8, 10\})$, to generate the partition $\pi_{C_3} = \{d, i\}, \{b, g\} = S_{18}, S_{19}$. Recursing upon the first set of this partition, $BUILD(S_{18}, C_{18} = \emptyset)$, gives the partition $\pi_{C_{18}} = \{d\}, \{i\} = S_{20}, S_{21}$ and thus leaves d and i at this level. Similarly, $BUILD(S_{19}, C_{19} = \emptyset)$ gives $\pi_{C_{19}} = \{b\}, \{g\} = S_{22}, S_{23}$ and leaves b and g. Once again, we return to the top level of recursion and compute $BUILD(S_4 = \{k, m\}, C_4 = \emptyset)$, giving the partition $\pi_{C_{C_4}} = \{k\}, \{m\} = S_{24}, S_{25}$, which when we recurse into give the final two leaves k and m. This concludes the algorithm, and thus the full tree is output, shown through in Figure 4 (the green nodes indicating the additions from this part).

Figure 4: Step 4—full tree



2.4 Part D

2.4.1 *Reverse-BUILD* Algorithm Implementation

The proposed algorithm *Reverse-BUILD* is shown through the pseudocode in Algorithm 2. It implements an iterative method for building a set of constrains C from a given tree T through which we could apply the $BUILD$ algorithm and construct a tree isomorphic to T. This is done through splitting T into distinct layers, where each layer contains all nodes at the same depth, and considering each of these layers L_i through a bottom-up approach. We start at the deepest layer and assign each of the leaves its own singleton set; sets on sibling leaves are then merged and assigned to label the corresponding parent node p.

We then consider the next layer, where similarly any leaves are assigned to singleton sets. Then, we construct a partition over all sibling nodes in layer L_i (with the same parent p in layer L_{i+1}), consisting of the sets associated with each child (if a child is a leaf the associated set will be a singleton set). Namely, for the parent node p in layer L_{i+1} with N children in layer L_i, the following partition is constructed over the sets $S_1 \to S_N$ of the children $1 \to N$ of p: $\pi_p = S_1, S_2, \ldots, S_N$. For each set $S_m \in \pi_p$, we then loop over all other sets S_n in the same partition and ensure there is a constraint which links the elements of S_m and S_n, following the two constraint rules described in Section 2.1.

The construction of this constraint (in absence of one existing already) is implemented through the lines 8–13 in Algorithm 2, which is done adhering to both rules. We need to ensure this is the case between every set in the partition so that the correct dependencies between nodes are exhibited in the full constraint set. If this is followed, it maintains that when the $BUILD$ algorithm is applied to the full set we can once again generate the same partitions (when we reach this level of the tree).

Once we have ensured all pairs of sets in π_p have a corresponding constraint in C satisfying the aforementioned rules, all sets in the partition are merged into a single superset containing all values within partition. This set is then linked to the node p such that

Algorithm 2 Reverse-BUILD

Input: tree T with labelled leaves S

Output: a corresponding the set of constraints C

1: initialise $C \leftarrow \emptyset$
2: **for all** non-root layers L_i of T (from deepest leaves to the layer before the root) **do**
3: assign each leaf l at layer L_i the singleton set containing its value $S_l = \{v_l\}$
4: **for all** non-leaf nodes p in layer L_{i+1} (above L_i) **do**
5: form the partition $\pi_p = S_1, S_2, \ldots, S_N$ consisting of the sets assigned to each of the N children of p
6: **for all** sets $S_m \in \pi_p$ containing more than one element **do**
7: **for all** other sets $S_n \in \pi_p \setminus S_m$ **do**
8: **if** no constraint $(a, b) < (c, d) \in C$ exists where $a, b \in S_m$ and either $c \in S_n$ or $d \in S_n$ **then**
9: sample two distinct leaves from the first set $a, b \in S_m$
10: set c equal to either of these leaves a or b
11: sample another leaf from the second set set $d \in S_n$
12: add a new constraint over these values: $C \leftarrow (a, b) < (c, d)$
13: **end if**
14: **end for**
15: merge the sets of π_p and assign this to the corresponding parent node p
16: **end for**
17: **end for**
18: **end for**
19: **return** the set of constraints C

upon the next iteration, when we are considering the nodes on its level to construct the partition, this set will be used as the set S_p associated with node p. After this has been performed for all collections of siblings in level L_i (i.e. for all parents in L_{i+1}), we move up to layer L_{i+1} and repeat.

2.4.2 Proof of Correctness

For the base case of *Reverse-BUILD*, we have the leaves a and b, which will be given the singleton sets $\{v_a\}$ and $\{v_b\}$. As these do not have more than one element, no constraint is constructed and we merge them into a set of two at their parent c. When we arrive back to node c during the forward *BUILD* algorithm, and wish to construct the partition of the set $\{v_a, v_b\}$, these nodes will be split into two singleton leaf nodes holding each value (as the set is of length 2 and thus can wholly contain no constraints). Thus, for leaves, the same tree structure is maintained over the forward and backwards passes.

Next, consider the case further into the application of *Reverse-BUILD*, where we are considering the parent node f, where we constructed the partition $\pi_f = \{v_a, v_b\}, \{v_d, v_e\}$ (by collating the sets of the children of f). It is clear that upon application of the forwards *BUILD*, for the constructed tree to split in this manner (i.e. produce the two children linked to these sets), there must be a constraint which requires this. Namely, there must be some constraint $(w, x) < (y, z)$ where w and x are distinct nodes in the same set (following the first rule) and y and z are in different sets (second rule). Thus, the constraint set must contain both $(v_a, v_b) <$ (v_a or v_b, v_c or v_d) and $(v_c, v_d) <$ (v_c or v_d, v_a or v_b). This is implemented through lines 8–13 of Algorithm 2, which arbitrarily selects the first set being considered to take the first two values (w, x), samples one of these for position y,

and a value from the second set for position z. This preserves the presence of a constraint in C such that when the forwards *BUILD* algorithm is applied, the partition splits the total set of values splits into the correct subsets to propagate into each subtree of the current node, maintaining the same overall structure. Specifically, upon the partition $\pi_f = \{v_a, v_b\}, \{v_d, v_e\}$, the algorithm *Reverse-BUILD* would construct constraints analogous to $(v_a, v_b) < (v_a, v_c)$ and $(v_c, v_d) < (v_c, v_a)$ (as all pairs of sets in the partition are looped through). The two sets would then be merged into the superset $\{v_a, v_b, v_c, v_d\}$ at their parent. When this parent is considered whilst applying the forwards *BUILD*, we will be tasked with generating a partition over the set $\{v_a, v_b, v_c, v_d\}$. Because the constraint set contains both $(v_a, v_b) < (v_a, v_c)$ and $(v_c, v_d) < (v_c, v_a)$, we know v_a, v_b must be in the same set, v_c, v_d must be in the same set, and v_a, v_c must be in different sets. Hence, the only way these values can be split is through the partition $\{v_a, v_b\}, \{v_d, v_e\}$, propagating the same sets of values into the subtrees as seen in the original tree (upon which we applied *Reverse-BUILD*).

If this is repeated on every node (for each partition), from the bottom of the tree upwards, this ensures that when that node is being considered during the application of *BUILD*, the same partitions are constructed, and thus the same values are propagated into each subtree, resulting in the same overall tree structure. Additionally, further into the application of *Reverse-BUILD*, as we consider nodes further up the tree, the partition sets get bigger and bigger. Thus, the likelihood that, when we are considering two sets X and Y in said partition, a constraint will exist between these two sets increases, and so the number of constraints that have to be constructed at every node decreases over the algorithmic process. Hence, only the necessary amount of constrains are added to the set to reconstruct the tree structure exactly, and if a constraint exists which already fulfils this criteria then redundant constraints are not further added (which would overfill the total constraint set).