Bioinformatics Assignment

zrlr73

1 Markov Models

a) HMMs with silent states

Since the Viterbi algorithm is already popular for solving this problem without silent states, I decided to adapt it for the inclusion of silent states. The Viterbi algorithm iterates using the observed output of the modelled HMM, constructing a trellis of possible states for each observed output.

Viterbi iterates over the observed sequence, making additional silent state computations difficult to include, given that they inherently do not create an observed symbol. Also, including additional logic to handle silent states inside the main loop will create significant time overheads.

Therefore, my proposed solution precomputes the most likely silent state transitions and includes them in the transition matrix. Essentially, for each state pair (a, b) it computes the sequence of silent states that is most likely to start at state a and finish at state b. If this most likely sequence is more likely than the 'standard' $a \to b$ transition, it will replace that transition in the transition matrix. It stores the sequence of silent states that created the corresponding item in the transition matrix as an ordered list of states, with a direct transition (ie, directly from $a \to b$) represented by an empty array.

The pseudocode is below (on a new landscape page), but is also included separately as the file 'q1a.pseudo'. In order to avoid verbose implementation details (which got out of hand quickly when I started writing things out in a more concrete manner!) the more abstract processes are described with comments instead of pseudocode.

```
function Viterbi_Silent(..., transitions, states, ...):
  // Parameters structured as follows:
  //  - transitions: a square 2D array with edge length equal to number of states
  //        where transitions[i][j] contains the probability of transitioning from
  //        state 'i' to state 'j'
  //  - states: an array of items representing states, where each item can be used
  //        to identify a slot in an array indexed by state
  // All other parameters omitted for brevity here!

  transitionStateMatrix = []

  for a in states:
    transitionStateMatrix[a] += []

    for b in states:

      // Run a Viterbi algorithm, modified as follows:
      //  - For the first row of the trellis, assume previous state was 'a'
      //  - Use an observed sequence of infinite silent states
      //  - For each sequence length, store the sequence with the highest
      //        probability of finishing by appending state 'b', and the
      //        corresponding probability
      //  - Stop iterating once every probability in the current sequence
      //        length's row is less than transitions[a][b]
      //  - Take the sequence of silent states with the highest probability over
      //        all lengths and store it in bestSequence, with its probability
      //        in bestProb

      if (bestProb > transitions[a][b]):
        transitions[a][b] = bestProb
        transitionStateMatrix[a][b] = bestSequence

  // Finally, run the standard Viterbi algorithm using the updated transition probability
  //      matrix, backtracking using transitionStateMatrix to identify the states
  //      traversed between each non-silent state
```

Correctness & Complexity

Since the Viterbi algorithm finds the most probable path, and one of the assumptions of an HMM is that each state transition is considered to be independent, it is clear that every time the transition $a \to b$ occurs (where both a and b are non-silent), we assume the most probable 'path' from a to b was used. Under the new assumptions, this may involve taking a 'longer route' via other silent states, or directly transitioning from the former to the latter. As such, if the most probable 'path' from a to b is via the silent states c, d, e, then in all cases where a transitions to b the most probable path is $a \to c \to d \to e \to b$. This is what allows us to completely replace the normal $a \to b$ transition with a single series of silent states, if that series is more likely than a direct transition.

The modified Viterbi algorithm designed to find these most likely silent sequences between a pair of non-silent states should also be correct, because it is essentially taking a maximum probability over all sequence lengths and all possible final states. Halting when the whole row is less than `transitions[a][b]` is valid because we are working with probabilities, so each new transition is only going to reduce the total probability of a given state sequence. Therefore, once every possible state sequence has become less likely than the normal transition probability, we can know that it is no longer possible to beat that probability with a longer sequence of silent states.

Finally, the condition after each modified Viterbi algorithm ensures that the silent state sequence will only replace the usual transition probability (and empty list of intervening states `[]`) if that sequence of states constitutes an improved transition probability when compared to the direct transition. All of this means we can be sure that the transition matrix takes into account the most likely possible sequence of silent states.

Since the main body of the algorithm still uses the unmodified Viterbi algorithm, the complexity of that section is $O(k^2 \times n)$ as standard. The pre-computing of possible silent state sequences is the main difference from the normal procedure, so we will now study that.

Technically, the modified Viterbi being run for each state transition could run forever in a model where both emission and transition probabilities of 1 are present. However, in the real world, HMMs that incorporate probabilities of 1 are not very useful or likely, so I believe this is not a problem. In a more realistic scenario, with a reasonable number of states with reasonable transition and emission probabilities, it is unlikely that it will reach sequences with greater lengths than one or two orders of magnitude. Testing an extreme situation with silent state transition & emission probabilities of 0.99 and a target transition probability of 0.1 yielded only 230 iterations.

The modified Viterbi algorithm will need to iterate over [all states, for each state, for each sequence length], and is being run for [all end states, for each start state]. Giving the silent state sequence length before [all probabilities $\leq$ transition probability] for any start/end pair the name s, its time complexity is a function of:

$$k^4 \times \text{Average}(s)$$

The value of s, and its average, are difficult to estimate, as it relies on nuances relating to the distributions of transition and emission probabilities. However, as mentioned above, I believe that it will not reach unreasonable magnitude for a real-world HMM, so it is not a huge problem. Also, any model with an excessively large number of iterations for one state pair will necessarily have an extremely uneven distribution of probabilities. Therefore, some state pairs would need zero (or otherwise a very small number of) iterations to evaluate, which goes some way to offset this individual complexity.

To summarise, both k and the average sequence length required to evaluate the silent state sequences will be relatively small for most reasonable cases. It is worth noting that the value that is usually largest, the overall sequence length n, has no bearing on this process - meaning this preprocessing section will take the same time even with extreme observed sequence lengths.

The space complexity of the first section is also fairly reasonable. The matrix that stores the most likely silent state sequences for each transition will be of size $k^2 \times$ (Average OR Maximum)(s), depending on whether a linked list or fixed array is used for the state sequences. The main trellis of probabilities used for the modified Viterbi algorithm will never exceed a factor of $k \times$ Maximum(s).

For the main Viterbi section, we once again have the usual space complexity of $k \times n$ for the main trellis, and the usual HMM parameters: the emission matrix at $k \times$ [Number of symbols], the transition matrix at k^2 and the initial probability distribution at k. However, we also need to store the silent state sequences for each transition for backtracking purposes, at the above size of $k^2 \times$ (Average OR Maximum)(s).

Therefore, to summarise:

The time complexity of the pre-computation is $k^4 \times$ Average(s), with its most space-complex structure at a complexity of $k^2 \times$ (Average OR Maximum)(s).

The second half of the algorithm, the standard Viterbi process, requires time of $O(k^2 \times n)$ and space $O(k \times n)$ with a small number of additional structures up to $k^2 \times$ (Average OR Maximum)(s).

It is worth noting that n, for a typical HMM, will be far larger than k or s, so the apparent complexity of the pre-computation is less extreme than it may seem at a glance.

b) The Expectation-Maximisation Algorithm

My implementation of EM for HMMs (aka the Baum-Welch algorithm) has been included in the file 'q1b.py'. Expectation-Maximisation works as follows:

> Given the observed sequence, the alphabet, and the number of states, generate some arbitrary model parameters for the initial distribution of state likelihoods, transition probabilities and emission probabilities.
>
> Run the forward algorithm on the sequence, given the initial generated parameters. The forward algorithm generates a trellis of probabilities with width `num_states` and height `sequence_length`. Each probability in this trellis corresponds to the likelihood of arriving in each state at each point in the observed sequence.
>
> Run the backward algorithm on the sequence. The backward algorithm generates a similar trellis of probabilities to the forward algorithm: the only difference is that the probabilities in the trellis correspond to the likelihood of producing the rest of the sequence starting from the given state at the given point in the sequence.
>
> With these metrics computed, we can compute the estimated number of times that each transition or emission will occur given our current model parameters. We can then divide these by the total number of transmissions & emissions to find a new set of model parameters, and also compute a new distribution for the starting states based on values from the first row of the trellises. Finally, we compute the log likelihood of the sequence being observed under our new set of parameters. This is used to measure progress towards the maximum.
>
> These new parameters are then substituted for the old ones, and the algorithm repeats again, starting with the forward algorithm. This happens until the likelihood of the new parameters stops appreciably increasing, at which point the algorithm terminates and returns the final set of parameters. For termination I used the point at which two consecutive likelihoods are identical - at this point the increase in likelihood is less than machine precision.

Since we are multiplying lots of probabilities here, which tends to result in very small numbers and will never exceed 1.0, it is a good idea to work in log space in order to maximise precision and minimise underflows. Since the log function effectively 'stretches out' small numbers and 'squashes' large numbers in its output space, it is ideal for this task. In log space, operations such as multiplication and division can be easily done with addition and subtraction. Numbers do need to be converted out of log space for a 'logsumexp' operation when sums are required, but this is not a significant problem as there will always be numbers of significant enough magnitude that results are useful.

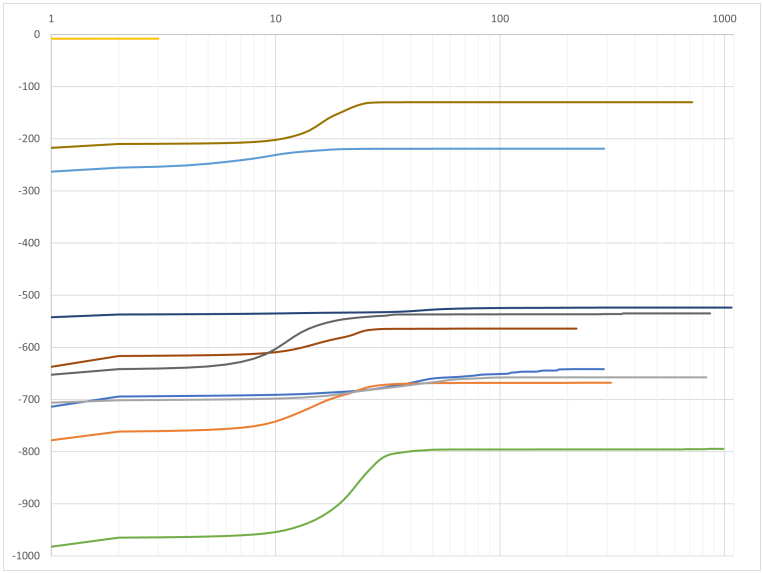
One issue I did encounter was that there were instances where probabilities of 0 were reached: this would land out of the range of the log function. To solve this, I implemented wrapper functions for `math.exp()` and `math.log()` which allowed the conversion of 0 into log space as negative infinity. Since the log space is entirely internal and the values returned are in conventional space, this does not affect the program's I/O.

I also employed a scaling method to keep numbers within reasonable bounds, based on section (V)(A) of Rabiner's 1989 paper on HMMs. This involved dividing each row in the forward trellis by its sum to ensure it added up to 1, and using the same forward sums to also scale the corresponding rows of the backward trellis. This causes the log likelihood in its normal form to be lost, but as shown in the paper it instead becomes equal to the sum of the logs of the row sums.

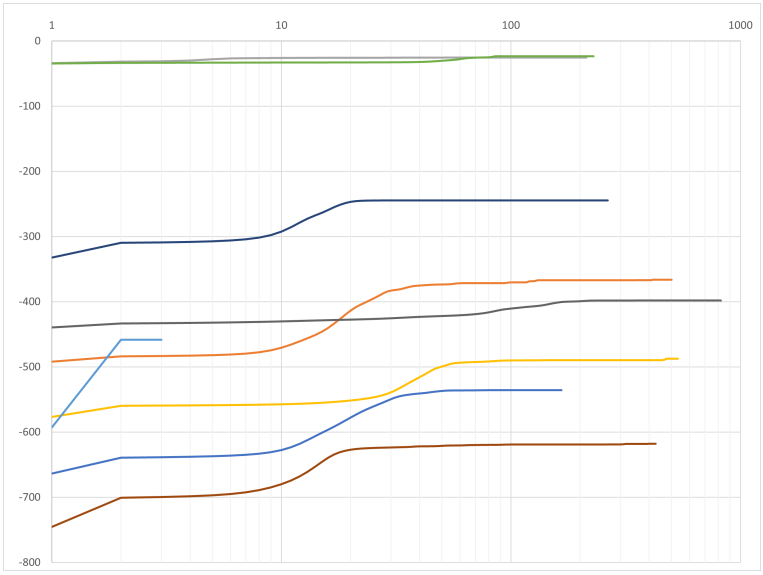
I also used the `randpd2` algorithm from The House Carpenter on WordPress to generate random probability distributions.

Validation

Following a number of tests on randomised sequences, I am confident that my implementation consistently converges on a maximum likelihood. Initially, I randomly selected letters from an alphabet, but I also created a basic HMM program with randomly generated model parameters. Convergence plots showing the results can be seen below.



Convergence plot for sequences generated randomly

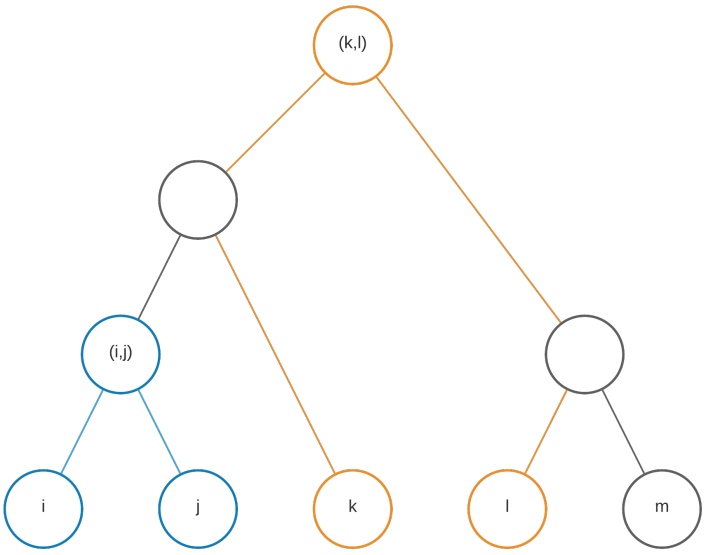


Convergence plot for sequences generated by randomised HMMs

2 Tree Reconstruction

a) The BUILD Algorithm

If we call (i, j) the lowest common ancestor of the leaves i and j of a tree, we can define an operator $<$ which denotes the hierarchy of lowest common ancestors. For example, the constraint $(i, j) < (k, l)$ tells us that the lowest common ancestor of k and l is itself an ancestor of the lowest common ancestor of i and j - see figure below.



The purpose of the BUILD algorithm is to construct a tree solely from these constraints. In a bioinformatics context, assuming the tree is a phylogeny, the constraints would be describing how closely related pairs of species are. As an example, we would specify the above constraint if we knew that the four species $\{i, j, k, l\}$ all had a common ancestor, but i and j were more closely related to each other than to the others.

In BUILD, the structure of a tree is inferred using a technique known as 'partitioning'. This is when the set of leaves is split into groups known as 'blocks' at the root - for example, assuming descriptive enough

constraints, the above tree would be split into the two sets $\{i, j, k\}$ and $\{l, m\}$ at the root. This partitioning can be done fairly easily using logical properties of the constraints:

From any constraint $(i, j) < (k, l)$ we can infer that:

1. i and j are in the same block.
2. If we already know that k and l are in the same block, then i and j must also be in that same block.

We also assume that no two leaves are in the same block unless otherwise specified by the constraints. Essentially, this means that any leaves t that are not added to blocks as a result of constraints will be put in their own block $\{t\}$. In the original paper, this is called the third rule.

With the set of leaves split into at least two blocks, we recurse into each block, partitioning every block until there is only one leaf left in the set. At this point, the algorithm's recursive structure exactly reflects the tree being constructed, so it simply needs to construct and return a tree. Instances with just one leaf return it, and instances that partitioned items join the returned structures with a new node until the outer layer is reached and the tree is complete.

b) Partitioning

Partitioning could occur as follows:

```
// a 2D array: items in this array will be referred to as 'blocks'
π_C = []

For each constraint (a, b) < (c, d):
  If neither a nor b already exists in π_C:
    Add the new block [a,b] to π_C

  Else if a and b already exist in separate blocks in π_C:
    Replace those two blocks with their union

  Else if a already exists in a block in π_C:
    Add b to that block
  Else if b already exists in a block in π_C:
    Add a to that block

For each constraint (a, b) < (c, d):
  If c and d share a block but a and b are in another block:
    Replace those two blocks with their union

For each leaf x that is not a member of an existing block:
  Add the new block [x] to π_C
```

The result of this algorithm is the 2D array π_C, with structure as follows:

```
[
  [i, j, k],
  [l, m, o],
  [n]
]
```

Each sub-array (each depicted here on its own line) corresponds to a block S_i, with letters denoting the leaves in that block. The entry `[n]` represents a leaf `n` that is a direct child of the current node.

c) Example

For clarity, I have named each block and set of constraints with their recursion depth and an arbitrary number to identify them at that depth. This is notated $S^{\text{depth}}_{\text{number}}$.

Recursion depth 0

Generating blocks from first halves of constraints:

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

Consolidating blocks with second halves of constraints:

S^1_1	S^1_2	S^1_3
{e,f,c,h,a,j,n,l}	{d,i,g,b}	{k,m}

Finding constraints for each block:

C^1_1	C^1_2	C^1_3
$(c,h) < (a,n)$	$(d,i) < (g,i)$	$\emptyset$
$(j,n) < (j,l)$	$(g,b) < (g,i)$	
$(c,a) < (f,h)$		
$(j,l) < (e,n)$		
$(n,l) < (a,f)$		
$(c,h) < (c,a)$		
$(e,f) < (h,l)$		
$(j,l) < (j,a)$		
$(j,n) < (j,f)$		

Recursion:

	Block		Constraints	
BUILD(	S^1_1	,	C^1_1	)
BUILD(	S^1_2	,	C^1_2	)
BUILD(	S^1_3	,	C^1_3	)

Recursion depth 1

BUILD(S^1_1, C^1_1):

Generating & consolidating blocks from constraints:

S^2_1	S^2_2	S^2_3
{c,h,a}	{j,n,l}	{e,f}

Finding constraints for each block:

C^2_1	C^2_2	C^2_3
$(c,h) < (c,a)$	$(j,n) < (j,l)$	$\emptyset$

Recursion:

	Block		Constraints	
BUILD(	S^2_1	,	C^2_1	)
BUILD(	S^2_2	,	C^2_2	)
BUILD(	S^2_3	,	C^2_3	)

BUILD(S^1_2, C^1_2):

Generating & consolidating blocks from constraints:

S^2_4	S^2_5
{d,i}	{g,b}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_4^2	,	C_4^2	)
BUILD(	S_5^2	,	C_5^2	)

BUILD(S_3^1, C_3^1):

Generating & consolidating blocks from constraints:

S_6^2	S_7^2
{k}	{m}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_6^2	,	C_6^2	)
BUILD(	S_7^2	,	C_7^2	)

Recursion depth 2

BUILD(S_1^2, C_1^2):

Generating & consolidating blocks from constraints:

S_1^3	S_2^3
{c,h}	{a}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_1^3	,	C_1^3	)
BUILD(	S_2^3	,	C_2^3	)

BUILD(S_2^2, C_2^2):

Generating & consolidating blocks from constraints:

S_3^3	S_4^3
{j,n}	{l}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_3^3	,	C_3^3	)
BUILD(	S_4^3	,	C_4^3	)

BUILD(S_3^2, C_3^2):

Generating & consolidating blocks from constraints:

S_5^3	S_6^3
{e}	{f}

Finding constraints for each block:

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

Recursion:

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

BUILD(S_4^2, C_4^2):

Generating & consolidating blocks from constraints:

S_7^3	S_8^3
{d}	{i}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_7^3	,	C_7^3	)
BUILD(	S_8^3	,	C_8^3	)

BUILD(S_5^2, C_5^2):

Generating & consolidating blocks from constraints:

S_9^3	S_{10}^3
{g}	{b}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_9^3	,	C_9^3	)
BUILD(	S_{10}^3	,	C_{10}^3	)

BUILD(S_6^2, C_6^2):

Return k

BUILD(S_7^2, C_7^2):

Return m

Recursion depth 3

BUILD(S_1^3, C_1^3):

Generating & consolidating blocks from constraints:

S_1^4	S_2^4
{c}	{h}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_1^4	,	C_1^4	)
BUILD(	S_2^4	,	C_2^4	)

BUILD(S_2^3, C_2^3):

Return a

BUILD(S_3^3, C_3^3):

Generating & consolidating blocks from constraints:

S_1^4	S_2^4
{j}	{n}

Finding constraints for each block:

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

Recursion:

	Block		Constraints	
BUILD(	S_3^4	,	C_3^4	)
BUILD(	S_4^4	,	C_4^4	)

BUILD(S_4^3, C_4^3):

Return l

BUILD(S_5^3, C_5^3):

Return e

BUILD(S_6^3, C_6^3):

Return f

BUILD(S_7^3, C_7^3):

Return d

BUILD(S_8^3, C_8^3):

Return i

BUILD(S_9^3, C_9^3):

Return g

BUILD(S_{10}^3, C_{10}^3):

Return b

Recursion depth 4

BUILD(S_1^4, C_1^4):

Return c

BUILD(S_2^4, C_2^4):

Return h

BUILD(S_3^4, C_3^4):

Return j

BUILD(S_4^4, C_4^4):

Return n

Travelling back up the recursion tree, with each internal node represented as a new set containing its children:

Recursion depth 4

Call	Returned Value
BUILD(S_1^4, C_1^4)	c
BUILD(S_2^4, C_2^4)	h
BUILD(S_3^4, C_3^4)	j
BUILD(S_4^4, C_4^4)	n

Recursion depth 3

Call	Returned Value
BUILD(S_1^3, C_1^3)	{c, h}
BUILD(S_2^3, C_2^3)	a
BUILD(S_3^3, C_3^3)	{j, n}
BUILD(S_4^3, C_4^3)	l
BUILD(S_5^3, C_5^3)	e
BUILD(S_6^3, C_6^3)	f
BUILD(S_7^3, C_7^3)	d
BUILD(S_8^3, C_8^3)	i
BUILD(S_9^3, C_9^3)	g
BUILD(S_{10}^3, C_{10}^3)	b

Recursion depth 2

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

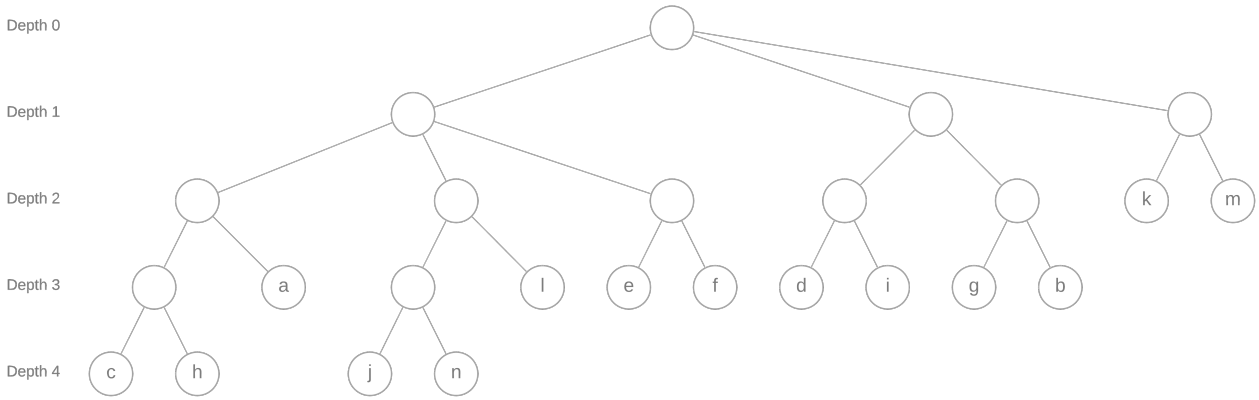
Recursion depth 1

Call	Returned Value
BUILD(S_1^2, C_1^2)	{{{c, h}, a}, {{j, n}, l}, {e, f}}
BUILD(S_2^2, C_2^2)	{{d, i}, {g, b}}
BUILD(S_3^2, C_3^2)	{k, m}

Recursion depth 0

Call	Returned Value
BUILD(S, C)	{{{{c, h}, a}, {{j, n}, l}, {e, f}}, {{d, i}, {g, b}}, {k, m}}

To summarise, the below tree is constructed:



d) Reversing BUILD

This algorithm has been implemented in Python 3.9 (specifically, 3.9.2), and is also included as 'q2d.py' alongside this PDF.

```
class Constraint:
  """Class representing a tree constraint such as those used by the BUILD algorithm.
  Stored constraint becomes (a, b) < (c, d) where a, b, c, d are the four arguments provided.
  Access Constraint.text to get the constraint as a human-readable string.
  """

  def __init__(self, a: str, b: str, c: str, d:str):
    self.items = [a, b, c, d]
    self.text = f'({a}, {b}) < ({c}, {d})'


class TreeNode:
  """Class representing a node in a tree. Takes two arguments:
  - children: A list of TreeNodes representing children (provide [] for a leaf)
  - name: The node's name (required for a leaf, optional otherwise)
  """

  def __init__(self, children: list, name: str = ''):
    self.children = children
    self.name = name
    self.isLeaf = children == []

  def traverse(self) -> (list, list, list):
    """Returns three values in a tuple:
    - A list of constraints representing the subtree below the current node
    - A 'flattened' list of the named nodes beneath the current node in the tree
      ie, a node with children (a node connected to (a, b), c) would return [a, b, c]
    - A list of pairs of nodes that would need to be linked together by constraints created
      by any parent of the current node
    """

    # Leaves return just themselves
```

```
if (self.isLeaf):
  return ([], [self.name], [])

numChildren = len(self.children)


# Traverse children
leaves = []
internals = []

childFlatLists = []
constraintLists = []

for i in self.children:
  (constraints, flatList, pairsToConnect) = i.traverse()

  if (len(flatList) == 1):
    leaves.append(flatList[0])

  else:
    internals.append((constraints, flatList, pairsToConnect))
    childFlatLists.append(flatList)
    constraintLists.append(constraints)

numLeaves = len(leaves)
numInternals = len(internals)


# Generate constraints from data returned by children
myConstraints = []
myFlatList = leaves
myPairsToConnect = []
childPairsToConnect = []

# If we're only connected to leaves, just return the flattened list of leaves,
```

```
#   making sure the parent connects them
if (numInternals == 0):
  return ([], leaves, [(leaves[i], leaves[i+1]) for i in range(numLeaves-1)])

# If there are multiple internal children, link them all with constraints
elif (numInternals > 1):
  # As we are connecting two internal children at a time, making one constrant for
  #   each would result in an unnecessary 'loop' - therefore we can omit one
  for i in range(numInternals-1):

    # Generate constraint, adding the right hand side to the list of pairs to connect for any parent
    thisFlatList = internals[i][1]
    nextFlatList = internals[i+1][1]
    myPairsToConnect.append((thisFlatList[0], nextFlatList[0]))

    # Add any remaining pairsToConnect from the current internal to a central list
    childPairsToConnect.extend(internals[i][2])

# Add pairsToConnect for any children not covered above
#   (either a single internal child skipped by the if, or the final one which was not covered by the loop)
childPairsToConnect.extend(internals[numInternals-1][2])

# Link any leaves to an internal
# (If we've got this far without returning, we have at least one internal child)
firstFlatList = internals[0][1]
for i in leaves:
  # If we have pairsToConnect from internal children, use those
  if (len(childPairsToConnect) > 0):
    connectingPair = childPairsToConnect.pop(0)

  else:
    # Otherwise, just use the first two leaves in the first internal
    connectingPair = firstFlatList

  # Generate constraint, adding the right hand side to the list of pairs to connect for any parent
```

```
      myConstraints.append(Constraint(connectingPair[0], connectingPair[1], connectingPair[0], i))
      myPairsToConnect.append((connectingPair[0], i))

    # If there are any more pairsToConnect from children
    while (len(childPairsToConnect) > 0):
      toConnect = childPairsToConnect.pop(0)

      # If we have any leaves, just connect to the first one
      if (numLeaves != 0):
        toConnectTo = leaves[0]

      else:
        # Otherwise, find a pair from any internal that doesn't include the pair we're connecting
        # There will always be at least two children in a valid tree, so
        #   if there are no leaves there will be enough internals
        toConnectTo = next(
          flatList[0]
          for flatList in childFlatLists
          if ((toConnect[0] not in flatList) and (toConnect[1] not in flatList))
        )

      # Generate the constraint
      #   (no need to add to myPairsToConnect as the right hand side is superfluous)
      myConstraints.append(Constraint(toConnect[0], toConnect[1], toConnect[0], toConnectTo))

    # Prepare output & return
    for i in childFlatLists:
      myFlatList.extend(i)

    for i in constraintLists:
      myConstraints.extend(i)

    return (myConstraints, myFlatList, myPairsToConnect)

  def getConstraints(self):
```

```
    """Returns a list of constraints representing the (sub)tree under the current node
    (Executes traverse() and returns only the constraints)
    """

    return self.traverse()[0]
```

Proof of correctness

The key to this algorithm is the `TreeNode.traverse()` function. This function returns three things:

- A list of constraints defining the subtree under the current node
- A 'flattened' list of all the leaves beneath the current node
- A list of pairs of leaves that occupy different branches underneath the current node (such that those branches will need to be connected up in the parent to ensure that they are all placed in the block representing the current node when constructing). This is referred to as `pairsToConnect` in the code and below.

The `TreeNode.traverse()` function calls itself recursively on the children, and uses the output of each child to construct the information needed for the current node. Each child is treated as a 'black box' that generates the constraints needed to describe its own structure and returns any pairs of nodes that need connecting together to form a coherent block at the parent level. As such, as long as the algorithm generates correct constraints for shallow ($<$ ≈3-4-layer) trees, it will be correct for any tree.

In the following section I will be referring to 'internal children': these are taken to be direct children of the current node that are not leaves. Each internal child will have at least two leaves somewhere beneath it in a valid tree.

The constraints returned are decided as follows:

- No constraints if the current node has no internal children. Otherwise:
- One for each directly-attached leaf 'l': (a, b) $<$ (a, l) where [a, b] come from the same internal child (and willl be preferentially selected from child pairsToConnect sets before arbitrarily picking them)
- Any further constraints (a, b) $<$ (a, x) where [a, b] are from child pairsToConnect sets and x is chosen arbitrarily from children's flat lists (these constraints simply serve to merge any remaining unmerged pairsToConnect)
- Any constraints passed up from children

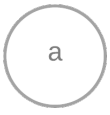
The flattened list is a simple list of all leaves below the current node - defined recursively, it's [a] for any leaf a and the union of all children's flattened lists for all internal nodes.

The pairsToConnect are decided as follows:

- No pairs if the current node is a leaf
- Pairs of leaves beneath n_i and n_{i+1} for each internal child n_i except the last
- (a, l) for each leaf l where a is a leaf beneath an internal child

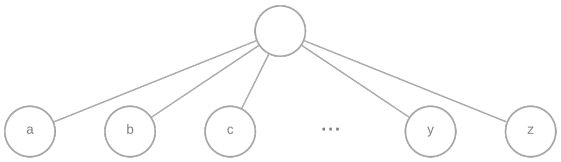
Abstractly, the aim of pairsToConnect is to ensure every branch underneath the current node is linked to at least one other branch in the parent, forming a fully connected block without using more constraints than are needed.

For the following structures, `TreeNode.traverse()` on the root might return:



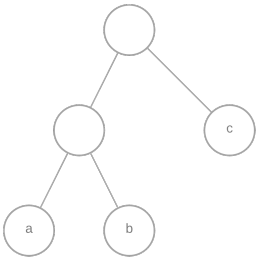
([], ['a'], [])

This is a trivial case and contains just one leaf. As such it returns no constraints or pairs to connect; simply itself inside the flattened child list.



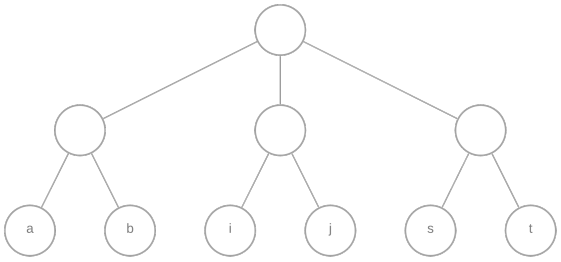
([], ['a', 'b', ..., 'z'], [('a', 'b'), ('b', 'c'), ..., ('y', 'z')])

This case demonstrates how an internal node with leaves for children would return itself. It still generates no constraints. This is correct: it is impossible to correctly describe a tree of this structure with constraints of the format used here. However, it does return pairs of leaves to be connected into one block at the parent.



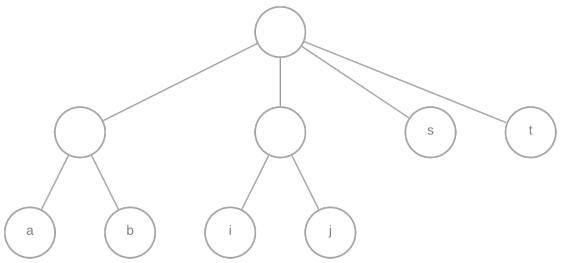
(['(a, b) < (a, c)'], ['a', 'b', 'c'], [('a', 'c')])

This is the simplest tree for which a constraint can be generated. It is represented by a single constraint. At this level we cannot generate constraints that ensure that the items of the (a, b) branch and the (c) branch are put together in the parent's blocks, which is why it returns the pair (a, c) for connecting with a constraint in the parent.



(['(a, b) < (a, i)', '(i, j) < (i, s)', '(s, t) < (s, a)'],
['a', 'b', 'i', 'j', 's', 't'], [('a', 'i'), ('i', 's')])

Each of the nodes in the middle layer has instructed the root here to join their leaves into one block, which is done with each constraint returned here. The internal nodes' sets will not be joined by the parent as things stand here though, so pairs of single leaves from each one are returned to the parent for connecting at the next layer.



(['(a, b) < (a, s)', '(i, j) < (i, t)'], ['a', 'b', 'i', 'j', 's', 't'],
[('a', 'i'), ('i', 's'), ('a', 't')])

This is a larger version of the third example. Each constraint links an internal node to a leaf, and serves two purposes: to define internal children's blocks and to state the existence of leaves. Returned for connection

by the parent are pairs of single nodes from each child.

While I have not conducted any formal proofs, I am confident that this algorithm is close to being optimal (if not optimal already). As a benchmark, it describes the graph in question 2c in only 11 constraints. For all small, contrived graphs on which I have tested it, I have found its result to be accurate (and, as far as I can tell, optimal), and see no particular reason that this would not apply to larger graphs too.