BINF6000 A2

Part A: Retrieving entries from biological sequence databases.

Exercise 1: Retrieving information for the protein TMM25_HUMAN (2 marks)

Change the above snippet of code so that it prints the actual sequence that makes up the signal peptide found in TMM25_HUMAN. You can assume that the start position is 1. You should end up with a print-out similar to what we got for RNS1_ARATH, which is shown below. Currently processing the sequence with the name UNIPROT:RNS1_ARATH. There is a signal peptide ending at position 22. The signal peptide looks like this: MKILLASLCLISLLVILPSVFS

Submit: Provide the code (1 mark) to extract the signal peptide from the sequence, and the new output (1 mark).

```
from guide import *

# Fetching the detailed annotations and sequence for TMM25_HUMAN
myprotein = getSequence('TMM25_HUMAN', format='txt')
mysequence = getSequence('TMM25_HUMAN', format='fasta')
# Extracting the header from the detailed annotations to get the UNIPROT code
header = myprotein.split('\n', 1)[0]
print("Currently processing the sequence with name UNIPROT:%s" %
header.split()[1])
# Initialize variables for start and end of the signal peptide
sp_start, sp_end = None, None
# Loop through each line of the detailed annotation to find the signal peptide
annotation
for line in myprotein.splitlines():
    words = line.split() # Split each line by whitespace
    if words[0] == 'FT' and words[1] == 'SIGNAL':
        sp_start, sp_end = map(int, words[2].split(".."))
# If a signal peptide is found, extract and print its sequence
if sp_start is not None and sp_end is not None:
    # Adjust indices for Python's 0-based indexing (subtract 1 from each)
    signal_peptide = mysequence.sequence[sp_start-1:sp_end]
    print("There is a signal peptide ending at position %d" % sp_end)
    print("The signal peptide looks like this: %s" % signal_peptide)
else:
    print("There is no signal peptide")
```

Output

Currently processing the sequence with name UNIPROT:TMM25_HUMAN

There is a signal peptide ending at position 26

The signal peptide looks like this: MALPPGPAALRHTLLLLPALLSSGWG

Exercise 2: Retrieve and process information about multiple members of the same protein family (2 marks)

Submit: Provide your code (1 mark) and the number of examples found in *E. coli* (1 mark).

```
from guide import *

seqs = [] # Initialize an empty list to store sequence objects
names = searchSequences('family:DAO AND reviewed:true') # Search for sequences in a specific family that are also reviewed
cnt = 0 # Initialize a counter for the number of Escherichia coli exemplars

# Loop through each name returned from the search
for name in names:
    seq = getSequence(name) # Retrieve the sequence for each name
    seqs.append(seq) # Append the sequence object to the list
    start_index = seq.annot.find("OS=") # Start index for species name
    # Check if the species name is found within the annotation
    if start_index != -1:
        end_index = seq.annot.find("=", start_index + 3) # Find the next occurrence of "=" after "OS=" which typically starts the next field

        # If no other "=" is found, use the length of the annotation
        if end_index == -1:
            end_index = len(seq.annot)
        else:
            # Otherwise, find the space before the next "=" to end the species name
            end_index = seq.annot.rfind(" ", start_index, end_index)

        species = seq.annot[start_index + 3:end_index].strip() # Extract the species name from the annotation
        print(seq.name, '\t', species) # Print the sequence name and its species

        # Check if the species is Escherichia coli
        if "Escherichia coli" in species:
            print('*')
            cnt += 1 # Increment the counter for Escherichia coli
    else:
        print(seq.name, '\tno species information available')

# Write all the sequences to a FASTA file named 'dao.fa'
writeFastaFile('dao.fa', seqs)
# Print the number of Escherichia coli exemplars found
if cnt > 0:
    print(f"Found {cnt} exemplars in E. coli")
else:
        print("No Escherichia coli exemplars found.")
```

Found 11 exemplars in E. coli

Exercise 3: Processing the FASTA format (2 marks)

It is helpful to know what the FASTA format is, so take a quick look in your new file. Find the functions `readFastaFile` and `readFastaString` in `guide.py` and try to understand how a FASTA file like the one shown below is read.

Submit: Briefly describe what the FASTA format is (1 mark). Explain why readFastaFile('fail.fa', Protein_Alphabet) will fail with the above FASTA file called fail.fa (1 mark).

FASTA format is a text-based format for representing nucleotide or peptide sequences, where sequences are represented in lines of text preceded by a header line. The header line starts with a ">" character, followed by the sequence identifier and optionally additional description or annotations. The sequence itself appears on subsequent lines right after the header and can span multiple lines.

The FASTA format uses > to indicate the start of a new sequence entry, followed by an identifier and optional description. Normally, this character is split to separate the identifier and description for further processing. If the > character is left in the line processed by parts = line[1:].split(), the first item in the parts list will start with >, e.g., >THIOG_NOSS1. When storing sequence identifiers or using them to reference sequences in data structures (like dictionaries or databases), having an unexpected > could lead to inconsistencies or lookup failures because the identifier stored ('>THIOG_NOSS1') would not match a cleanly queried identifier ('THIOG_NOSS1').

Part B: Sequence alignment

Exercise 4: Find two putative homologs to *your* D-amino-acid oxidase (2 marks)

Using the D-amino-acid oxidase set, find the two closest sequences to yours in terms of sequence *identity* (the fraction of amino acids that agree). You are provided with code that will perform this task, but you need to (a) comment the code line-by-line to illustrate that you understand it, and (b) print out the resulting sequences with their percent similarity. Once known, look up the putative homologs in Uniprot (http://uniprot.org) with your web browser.

Submit: Provide in your response,
(a) the numbers and names of the three sequences and the percent identity for the two alignments (0.6 marks)
(b) your code that is commented line-by-line to illustrate that you understand it and the output generated by your code (0.8 marks), and
(c) a sentence or two that (attempt to) explain their similarity, e.g. in terms of taxonomy, function, or evolution more generally (0.6 marks).

I will be using sequence number 167, which is sp|Q7VM59|MNMC_HAEDU

Best Match 1: sp|A3N0L8|MNMC_ACTP2 with 73.77% identity

Best Match 2: sp|B0BPE2|MNMC_ACTPJ with 73.77% identity

```
from guide import *

# Load the substitution matrix and sequence data
b62 = readSubstMatrix('blosum62.matrix', Protein_Alphabet)
seqs = readFastaFile('dao.fa', Protein_Alphabet)
# Use student number to pick a specific sequence to compare against others
my_student_number = 45367115
my_sequence_number = my_student_number % (len(seqs) - 1)
print('I will be using sequence number %d, which is %s' % (my_sequence_number,
seqs[my_sequence_number].name))
#I will be using sequence number 167, which is sp|Q7VM59|MNMC_HAEDU

# Initialize variables to store the highest identity values and their indices
best1 = 0
idx1 = 0
best2 = 0
idx2 = 0

# Iterate over all sequences in the dataset
for idx in range(len(seqs)):
    # Ensure the current sequence is not the sequence of interest
    if idx != my_sequence_number:
        seq = seqs[idx]
        # Align the sequence of interest with the current sequence using the
BLOSUM62 substitution matrix and a gap penalty of -7
        aln = align(seqs[my_sequence_number], seq, b62, -7)
        # Calculate the percent identity of the alignment
        percent = scoreAlignment(aln) / aln.alignlen;

        # If this percent identity is higher than the best recorded, update
the first and second best matches
        if best1 < percent:
            best2 = best1  # Move the best match to second best
            idx2 = idx1  # Update the index of the second best match
            best1 = percent  # Update the best percent identity
            idx1 = idx  # Update the index of the best match

        # If this percent identity is not better than the best but better than
the second best, update the second best
        elif best2 < percent:
            best2 = percent  # Update the second best percent identity
            idx2 = idx  # Update the index of the second best match

# Output the best matching sequences and their percent identities
print(f"Best Match 1: {seqs[idx1].name} with {best1 * 100:.2f}% identity")
print(f"Best Match 2: {seqs[idx2].name} with {best2 * 100:.2f}% identity")
```

The high sequence similarity between the D-amino-acid oxidases from Haemophilus and Actinoplanes, despite their differing taxonomic classifications in the Gammaproteobacteria and Actinobacteria respectively, highlights a conserved functional role of these enzymes across diverse bacterial groups. This conservation suggests these enzymes might perform essential roles related to amino acid metabolism that are critical across varied environmental niches and evolutionary pressures." This reflects the evolutionary conservation of the enzyme's function despite taxonomic differences, suggesting that its role in metabolism is critical enough to be preserved across diverse bacterial lineages.

Exercise 5: Determine the consensus sequence for the D-amino-acid oxidase MSA (2 marks)Outline

Write a new function `getConsensus` that takes an alignment as an argument. It should construct and return a new `Sequence` from the consensus characters determined for each column in the alignment. You are recommended to repeatedly call `getConsensusForColumn` above.

Submit: Provide the commented code you wrote (1 mark) and the consensus sequence (1 mark) for your D-amino-acid oxidase alignment.

```
from guide import *

# Read the alignment from a Clustal format file into an alignment object.
aln = readClustalFile('dao.aln', Protein_Alphabet)
# Print the number of sequences loaded and the width of the alignment.
print('Loaded %d sequences into the alignment, which is %d columns wide' % (len(aln), aln.alignlen))
aln.writeHTML('dao.html') # Write the alignment to an HTML file for visualization.

def getConsensusForColumn(aln, colidx):
    symcnt = {} # Dictionary to count occurrences of each symbol in the column.
    for seq in aln.seqs: # Loop through each sequence in the alignment.
        mysym = seq[colidx] # Fetch the symbol at the current column index.
        # Increment the count for this symbol in the dictionary.
        if mysym in symcnt:
            symcnt[mysym] += 1
        else:
            symcnt[mysym] = 1
    # Variables to track the most common symbol and its count.
    consensus = None
    maxcnt = 0
    # Determine the symbol with the highest count.
    for mysym in symcnt:
        if symcnt[mysym] > maxcnt:
            maxcnt = symcnt[mysym]
            consensus = mysym
    return consensus # Return the most common symbol for this column.

def getConsensus(aln):
    """
```

```
    Constructs a consensus sequence from a given alignment.

    Args:
    aln (Alignment): The alignment from which to derive the consensus.

    Returns:
    Sequence: A sequence object representing the consensus of the alignment.
    """
    consensus_string = ''  # Initialize an empty string to store consensus characters

    # Iterate over each column index in the alignment's width
    for colidx in range(aln.alignlen):
        # Get the most common character in each column
        consensus_char = getConsensusForColumn(aln, colidx)
        print(colidx)
        print(consensus_char)
        # Append this character to the consensus string
        consensus_string += consensus_char

    # Create a new Sequence object with the consensus string
    consensus_sequence = Sequence(consensus_string, Protein_Alphabet, name='Consensus', gappy=True)
    return consensus_sequence

# Load the alignment
aln = readClustalFile('dao.aln', Protein_Alphabet)
consensus_seq = getConsensus(aln) # Obtain the consensus sequence
print(consensus_seq) # Print the consensus sequence to verify the output
```

```
Consensus: --------------------SIQPATL----EWNE---D----
GTPVSRQFDDVYFSNDNGLEETRYVFLGGNGLPERWAEH----------RLFVIAETGFGTGLNFLALWQAFRQFRPA----
---------------------LQRLHFISFEKFPLTRADLARAHQ-----------------------HW---------P--
--ELAP---LAEQLLAQWP---A-LL----PGCHRLLFDEGRVTLDLWFGDINELLPQ--------------------
LNARVDAWFLDGFAPAKNPD-------MWTP-----------------------NL------FNAMARLARPGATLATF--
---TSAGFV------------R--RGLQEAGFTVQKVK----------------------GFGRKREMLCGEMEQRL-----
--------------------PWF-RP----------------------
KRDAAIIGGGIAGAALALALARRGWQVTLYCADEAP-AQGASGNPQGALYPLLSKDDNALSRFFRAAFLFARRFYDALL----
----------G-AFDHDWCGVLQLAWDEKSAERLAKML---ALGLPA----------ELASALDAEEAEQL---
AGLPLACGGLFYPQGGWLCPAELTRALLALA----G---TLHYGTEVQRL--ER------------------D-GW-------
---------------QLLDAQG-------------------ASAPVVVLA--NGHQITRFS-------------QTAHLP---
-LYPVRGQVSHIPTT--------------PAL--LKT-VLCYDGYLTPA-----------------NG--HHC-----
IGASYDRGDED-----TAFREADQQ----ENLQRLQECLPD--------W---------------------------------
----------EVD-----------VSDLQARVGVRCATRDHLPMVGPVPDYAATLAEYA-L--------------------A-
PAPVYPGLYV-LGGLG----SRGLCSAPLCAELLAAQICGEPLPLDADLLAALNPNRFWVRKLLKGKA---------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------
```

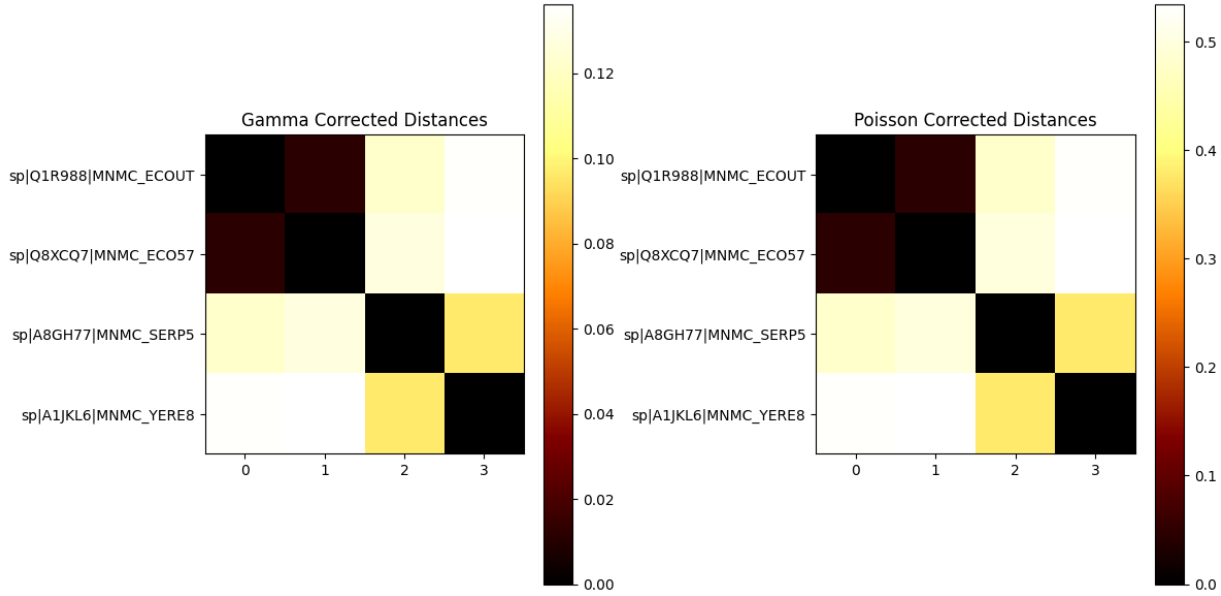
Exercise 6: Add the Poisson and Gamma distance metrics to `calcDistances` (3 marks)

Add two other evolutionary distance metrics, specifically the Poisson and Gamma corrected distances as described by Zvelebil and Baum (p267-276), to the method `calcDistances`.

Submit: Provide the amended function of your code (2 marks) and a new heatmap (1 mark). Comment the rows you modified.

```
from guide import *
import numpy as np
import matplotlib.pyplot as plt

class AdvancedAlignment(Alignment):
    def calcDistances(self, measure='fractional'):
        distmat = np.zeros((len(self.seqs), len(self.seqs))) # Initialize a matrix to store
distances.
        for i in range(len(self.seqs)): # Iterate over each sequence.
            for j in range(i + 1, len(self.seqs)): # Compare with every other sequence.
                seqA = self.seqs[i]
                seqB = self.seqs[j]
                L = D = 0 # Initialize counters for total positions and differences.
                for k in range(self.alignlen): # Iterate over alignment length.
                    if seqA[k] != '-' and seqB[k] != '-': # Only consider positions without
gaps.
                        L += 1
                        if seqA[k] != seqB[k]: # Increment difference counter if residues
differ.
                            D += 1
                if L == 0:
                    p = 0
                else:
                    p = D / L # Fraction of differing positions.

                if measure == 'poisson':
                    # Poisson correction as per Zvelebil and Baum
                    dist = -0.75 * np.log(1 - (4/3) * p) if p < 0.75 else 10
                elif measure == 'gamma':
                    # Gamma correction with alpha = 2.0
                    alpha = 2.0
                    dist = (1 / alpha) * ((1 - p)**(-1/alpha) - 1) if p > 0 else 0
                else:
                    dist = p # Default to fractional distance if no other measure is
specified.

                distmat[i, j] = distmat[j, i] = dist # Symmetrically fill the matrix.
        return distmat

# Load the full alignment from the file
aln = readClustalFile('dao.aln', Protein_Alphabet)
```

```
# Specify a list of the identifiers for the sequences you want to include
selected_names = [
    "sp|Q1R988|MNMC_ECOUT", "sp|Q8XCQ7|MNMC_ECO57", "sp|A8GH77|MNMC_SERP5",
    "sp|A1JKL6|MNMC_YERE8", "sp|B1JKLM|MNMC_ABCD1", "sp|C2DEFG|MNMC_HIJK2",
    "sp|D3HIJK|MNMC_LMNOP", "sp|E4KLMN|MNMC_QRSTU", "sp|F5NOPQ|MNMC_UVWXZ",
    "sp|G6QRST|MNMC_XYZAB"
]

# Filter the alignment to only include the specified sequences
filtered_seqs = [seq for seq in aln.seqs if seq.name.split()[0] in selected_names]
selected_aln = AdvancedAlignment(filtered_seqs)
d2_gamma = selected_aln.calcDistances('gamma') # Calculate distances using the Gamma
correction
d2_poisson = selected_aln.calcDistances('poisson') # Calculate distances using the Poisson
correction

# Plotting the heatmaps for visualizing distances
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))  # Creates a figure with two subplots

# Plot Gamma corrected distances
cax1 = ax1.imshow(d2_gamma, cmap=plt.cm.afmhot, interpolation='nearest')
fig.colorbar(cax1, ax=ax1) # Add a colorbar to the plot.
ax1.set_yticks(np.arange(len(selected_aln.seqs))) # Set y-ticks to sequence names.
ax1.set_yticklabels([s.name for s in selected_aln.seqs])
ax1.set_title('Gamma Corrected Distances')

# Plot Poisson corrected distances
cax2 = ax2.imshow(d2_poisson, cmap=plt.cm.afmhot, interpolation='nearest')
fig.colorbar(cax2, ax=ax2)
ax2.set_yticks(np.arange(len(selected_aln.seqs)))
ax2.set_yticklabels([s.name for s in selected_aln.seqs])
ax2.set_title('Poisson Corrected Distances')

plt.tight_layout()  # Adjust layout to not overlap
plt.show() # Display the plots.
```



Darker Colors represent smaller distances, indicating that the sequences are more closely related. Brighter or lighter colors (towards yellow or white in the colormap) represent larger distances, suggesting that the sequences are less closely related or more evolutionarily distant from each other.

Part C: Phylogenetic analysis

Exercise 7: Curate the MalS.fa dataset (2 marks)

You are provided with a file `MalS.fa`, which contains the amino acid sequences of a set of 50 MalS proteins. If you look in the file, you will see that the first part of each FASTA header (e.g. 'S.cerevisiae' for 'S.cerevisiae S288c IMA2') is the name of the yeast species the MalS protein belongs to. This is the text-string that is used to assign a `.name` to the sequence by default in most programs, including our Python implementation `readFastaFile`.

Because of this, you will have sequences with identical names. We need to make sure that each sequence has a unique name as many alignment programs cannot handle duplicate names. For example, we can achieve this by replacing the space in 'S.cerevisiae S288c IMA2' with an underscore to get 'S.cerevisiae_S288c_IMA2'. The `str` method `'split'` might be useful here. You can read more about it with the below code:

As a result of repeated gene duplication and speciation, some species of yeasts contain many homologs of this enzyme (e.g. *S. cerevisiae* contains 7 genes, *S.mikatae* has three, and *S. bayanus* has only one). Using the set of 50 MalS proteins, you should:

- Assign a new name to each sequence via concatenation of each word in the FASTA header, separated by the underscore-character `_`.
- Remove any proteins from the `MalS.fa`, which are **not** from one of the yeast types in the table above. The keys of the `yeasts` dictionary include all species recorded in the table.
- Save the subset of sequences into a file `select.fa`

Submit: Provide your commented code (1 mark) and report how many sequences were selected (1 mark).

```
from guide import *

# Load the CSV file to create the yeasts dictionary
import csv
yeasts = dict() # Initializes an empty dictionary to store yeast metabolism data.
with open('sugars.csv', 'rt') as csvfile: # Opens the 'sugars.csv' file in text mode.
    reader = csv.reader(csvfile)  # Creates a CSV reader object to iterate over lines in the CSV file.
    for row in reader:  # Iterates over each row in the CSV file.
        yeasts[row[0]] = [y == 'True' for y in row[1:]] # Converts each row into a dictionary entry where the key is the yeast species and the value is a list of Booleans indicating sugar metabolism.
print (yeasts) # Print this out to get an idea of the structure
for yeast in yeasts: # Iterates over keys in the 'yeasts' dictionary.
    print (yeast) # Prints each yeast species (key) from the dictionary.

sugars = ['Maltose','Sucrose','Turanose','Maltotriose','Maltulose','Melizitose','Isomaltose','Palatinose','Me-a-Glu']
sugar_index = {} # Dictionary to hold index positions of sugars.
cnt = 0 # Counter initialized to 0.
for sugar in sugars: # Loops through each sugar in the list.
    sugar_index[sugar] = cnt # Assigns an index to each sugar based on its position in the list.
    cnt += 1 # Increments the counter by one.

# for example...
yeasts['S.kluyverii'][sugar_index['Maltulose']] # Accesses whether 'S.kluyverii' metabolizes 'Maltulose' by using the sugar_index to locate the right Boolean value in the list.
# In the output dictionary, 'S.kluyverii' is associated with the list [True, True, False, False, True, False, False, True, True]. Using the sugar_index:
# yeasts['S.kluyverii'][4] would thus access the fifth element in the list for 'S.kluyverii', which is True, indicating that 'S.kluyverii' does metabolize 'Maltulose'.

mals = readFastaFile('MalS.fa', Protein_Alphabet) # Reads the 'MalS.fa' FASTA file and returns a list of Sequence objects.
select = [] # List to store selected sequences.
cnt = 0 # Counter to track the number of selected sequences.
for seq in mals: # Iterates over each sequence in 'mals'.
    if (seq.name in yeasts): # Checks if the sequence's yeast species is listed in the 'yeasts' dictionary.
        for annot in seq.annot.split(): # Splits the annotation part of the sequence name and iterates over each element.
            seq.name += '|' + annot # Appends each annotation to the sequence name separated by a '|'.
        select.append(seq) # Adds the modified sequence to the 'select' list.
        cnt += 1 # Increments the counter.
writeFastaFile('select.fa',select) # Writes the selected sequences to a new FASTA file 'select.fa'.
print('Selected', cnt, 'sequences') # Prints the number of selected sequences.
```

Selected 31 sequences

Exercise 8: Identify the consensus sequence for the MalS alignment (2 marks)

You will now perform a multiple sequence alignment on your subset of MALS proteins so that you can observe how similar they are and hypothesise their evolutionary relationships. Use Clustal Omega to create a multiple sequence alignment of your subset of MALS proteins and determine the consensus sequence.

Submit: Provide the consensus sequence (2 marks).

```
from guide import *

# Function to calculate consensus for each column in the alignment
def getConsensusForColumn(aln, colidx):
    symcnt = {}  # Dictionary to count occurrences of each symbol in the column
    for seq in aln.seqs: # Iterate over each sequence in the alignment
        mysym = seq[colidx] # Get the symbol at the current column index
        if mysym in symcnt:
            symcnt[mysym] += 1 # Increment count if symbol already encountered
        else:
            symcnt[mysym] = 1 # Initialize count for new symbol
    consensus = None # Variable to hold the consensus symbol for the column
    maxcnt = 0 # Variable to track the highest count of any symbol
    for mysym, count in symcnt.items(): # Iterate over symbol counts
        if count > maxcnt: # Check if the current symbol count is the highest encountered
            maxcnt = count # Update the highest count
            consensus = mysym # Set the consensus symbol to the one with the highest count
    return consensus # Return the consensus symbol for the column

# Read the alignment file
aln = readClustalFile('MalS.aln', Protein_Alphabet)

# Calculate the consensus sequence
consensus_seq = '' # Initialize an empty string to build the consensus sequence
for colidx in range(aln.alignlen): # Iterate over each column index in the alignment
    consensus_seq += getConsensusForColumn(aln, colidx) # Append the consensus symbol of each column to the sequence

# Output the calculated consensus sequence
print("Consensus Sequence:", consensus_seq)
aln = readClustalFile('MalS.aln', Protein_Alphabet)
# Write the alignment to an HTML file for visual inspection
aln.writeHTML('MalS.html')
```

```
Consensus Sequence: ---MTISSAHPETEPKWWKEATIYQIYPASFKDSN----------
NDGWGDLKGIASKLEYIKELGVDAIWICPFYDSPQDDMGYDIANYEKVWPTYGTNEDCFALIEKTHKLGMKFITDLVINHCSS
EHEWFKESRSSKTNPKRDWFFWRPPKGYDAEGKPIPPNNWRSFFGGSAWTFDEKTQEFYLRLFASTQPDLNWENEDCRKAIYE
SAVGYWLDHGVDGFRIDVGSLYSKVPGLPDAPVTDENSKWQHSDPFTMNGPRIHEFHQEMNKFMRNRV-
KDGREIMTVGEVQHGSDETKRLYTSASRHELSELFNFSHTDVGTSPKFRYNLVPFELKDWKVALAELFRFINGTDCWSTIYLE
NHDQPRSITRFGDDSPKNRVISGKLLSVLLVSLTGTLYVYQGQELGQINF-KNWPIEKYEDVEVRNNYKAIKEEHGENSK---
EMKKFLEGIALISRDHARTPMPWTKEEPNAGFSG---
PDAKPWFYLNESFREGINAEDESKDPNSVLNFWKEALQFRKAHKDITVYGYDFEFIDLDNKKLFSFTKK--Y-
DNKTLFAALNFSSDEIDFTIPNDSASFKLEFGNYPDKEVDASSRTLKPWEGRIYISE--
```

Exercise 9: Locate the MalS active site in the alignment (2 marks)

Voordeckers et al. (2012) mapped an alignment of MalS onto a structure of Ima1 (which is a member of the MalS family; it should be part of your selected subset; see Yamamoto et al. 2010 for the structure). Voordeckers and colleagues identified 9 columns that corresponded to the site at which sugars are metabolised. They appear as columns 173, 231-234, 294-295, 324 and 437 in the Voordeckers et al. alignment.

Submit: Identify "by eye" and provide the column numbers (2 marks) that make up the active site, i.e. map the columns in Voordeckers alignment to columns in your alignment from Exercise 8.

```
from guide import *

def getConsensus(aln):
    """Calculates the consensus sequence for the given alignment."""
    consensus = []
    for i in range(aln.alignlen):
        col = [seq.sequence[i] for seq in aln.seqs if seq.sequence[i] != '-']
        if col:
            consensus.append(max(set(col), key=col.count))
        else:
            consensus.append('-')
    return ''.join(consensus)

def find_closest_sequence(aln, consensus_seq):
    """Finds the sequence in the alignment that is closest to the consensus sequence."""
    min_diff = float('inf')
    closest_seq = None
    for seq in aln.seqs:
        diff = sum(1 for a, b in zip(seq.sequence, consensus_seq) if a != b)
        if diff < min_diff:
            min_diff = diff
            closest_seq = seq
    return closest_seq

# Load the alignment
aln = readClustalFile('MalS.aln', Protein_Alphabet)

# Specify Voordeckers et al. column indices (adjust to zero-index)
voordeckers_columns = [172, 230, 231, 232, 233, 293, 294, 323, 436]  # Zero-indexed

# Function to fetch a column from the alignment
def get_column(alignment, col_index):
    return ''.join(seq.sequence[col_index] for seq in alignment.seqs if col_index < len(seq.sequence))

# Iterate over the specified columns and print their contents
for col in voordeckers_columns:
    column_content = get_column(aln, col)
    print(f"Column {col + 1}: {column_content}")
```

```
Column 173: FFFFFFFFFFFFFFFFFFFFF
Column 231: GGGGGGGGGGGAAAAAAGGGGAA
Column 232: SSSSSSSSSSSGGGGGGSSSSGG
Column 233: LLLLLLLLLLLLLLLLMMMMLL
Column 234: YYYYYYYYYYYYYYYYYYYYY
Column 294: QQQQQQQQRRQQAAAAAAGGGGGG
Column 295: HHHHHHHHHHHHHHHHHVIVFFQF
Column 324: GGGGGGGGGGGGGGGGGGGGGG
Column 437: EEEEEEEEEEEKEKKKRRREEED
```

Conserved residues across different sequences indicate a functional, structural, or evolutionary significance. High conservation suggests these residues are crucial for the protein's function or structural integrity.

Column 173 (F): Phenylalanine (F) being conserved across all sequences suggests it could play a critical role in maintaining structural stability or partaking in a specific function like substrate binding or maintaining the protein's integrity.

Column 231 (G and A): Glycine (G) and Alanine (A) are small amino acids found in regions where the protein needs flexibility or tight packing. A shift from G to A might indicate minor variations in flexibility or steric requirements.

Columns 232-233 (S, G, L, M): Serine (S), Glycine (G), Leucine (L), and Methionine (M) showing up suggests variability in a region that might be involved in binding or active site mechanics due to the side chain properties of these amino acids (polarity, flexibility, and hydrophobicity).

Column 234 (Y): Tyrosine (Y) conservation across sequences strongly points to a critical role in catalytic activity, involved as a part of the binding site due to its ability to participate in hydrogen bonding and aromatic stacking interactions.

Columns 294-295 (Q, R, H): The presence of Glutamine (Q), Arginine (R), and Histidine (H) highlights a region crucial for enzymatic activity, as these residues are often found in active sites where they might contribute to substrate binding or catalysis, especially given their polar and charged nature.

Column 324 (G): Glycine's suggests flexibility required in this part of the protein; a loop region necessary for the enzyme's functional motion.

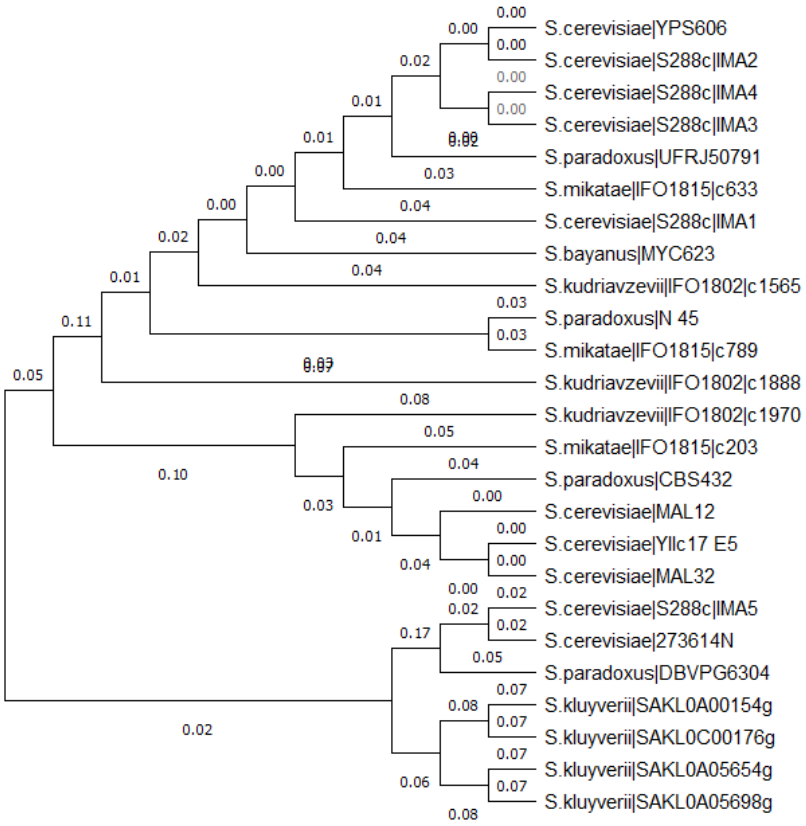
Column 437 (E, K): Glutamate (E) and Lysine (K) variation indicates a region crucial for maintaining the protein’s charge balance and interactions, contributing to substrate binding or structural stability at the active site.

Exercise 10: Infer the phylogenetic relationships among select MalS proteins (2 marks)

You will now use the MSA to infer a phylogenetic tree that describes their evolutionary relationships. Infer a phylogenetic tree using UPGMA and the Poisson corrected evolutionary distance from above, i.e. *not* the fractional distance.

Submit: Provide the code you used (1 mark) and the tree (1 mark) with tips labelled with the names of the sequences (species and other annotations as described previously).

```
from guide import *

# Load the alignment from the MalS protein sequences
aln = readClustalFile('MalS.aln', Protein_Alphabet)

# Calculate the Poisson corrected distances for the sequences
distances = aln.calcDistances(measure='poisson')

# Generate the phylogenetic tree using the UPGMA algorithm
tree = runUPGMA(aln, 'poisson')

# Print and save the tree in Newick format with sequence labels
print("Phylogenetic Tree in Newick Format:")
print(tree)
writeNewickFile('MalS_phylogenetic_tree_poisson.nwk', tree)
```



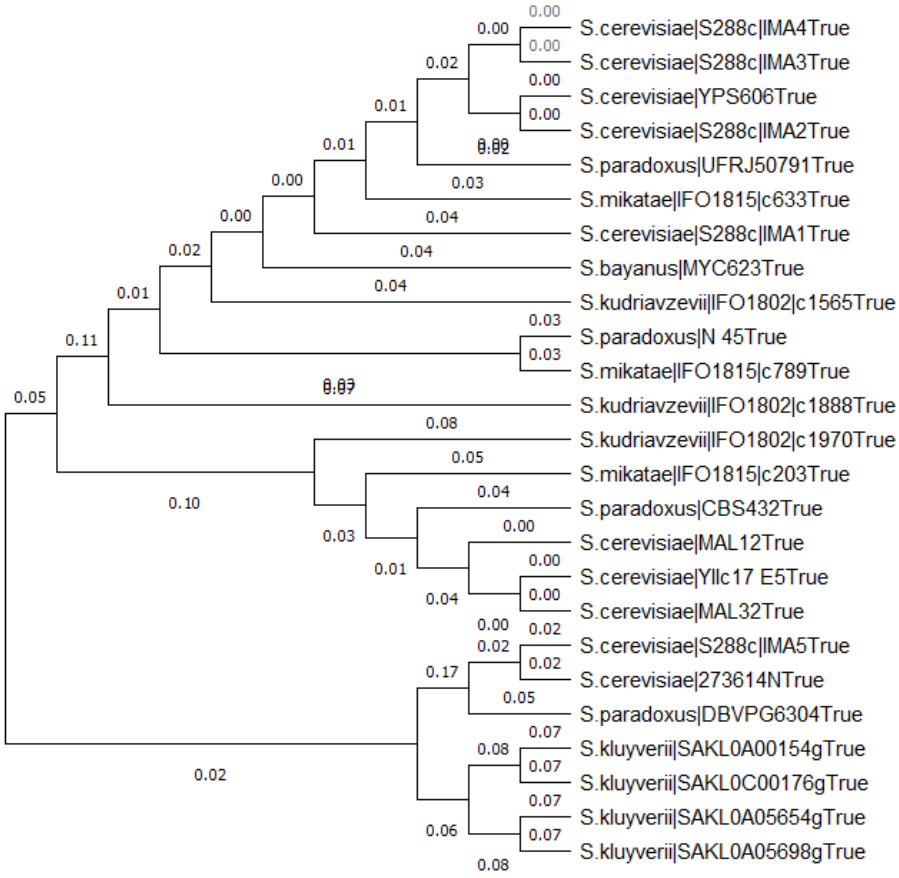
Exercise 11: Annotate your tree with metabolised sugars (2 marks)

From the MalS table, we know that one or more proteins in a given species will be able to metabolise a certain sugar. Since we do not know which, we assume that all proteins in a species will be able to metabolise all sugars with which the species is associated.For each sugar, visualise the phylogenetic tree, now with tips labelled with `True` or `False` depending on whether the protein could be metabolising it. Identify the sugar which binds to a subset of the yeasts in your data set, and where `True` and `False` appear to be in some agreement with the tree.

Submit: Provide your commented code (0.8 marks) along with your annotated tree (0.6 marks) for the sugar you identified. Explain what may have happened to that metabolic function over evolutionary time (0.6 marks), by referring to the species and proteins in your tree.

```
from guide import *
import csv

# Load the CSV file to create the yeasts dictionary
yeasts = {}
with open('sugars.csv', 'rt') as csvfile:
    reader = csv.reader(csvfile)
    next(reader)  # Skip header line
    for row in reader:
        yeasts[row[0]] = [y == 'True' for y in row[1:]]

# Define the sugars and create a sugar index dictionary
sugars = ['Maltose', 'Sucrose', 'Turanose', 'Maltotriose', 'Maltulose', 'Melizitose', 'Isomaltose',
'Palatinose', 'Me-a-Glu']
sugar_index = {sugar: idx for idx, sugar in enumerate(sugars)}

# Load the alignment and generate the tree
aln = readClustalFile('MalS.aln', Protein_Alphabet)
tree = runUPGMA(aln, 'poisson')

# Function to annotate the tree with sugar metabolism
def annotate_tree(node, yeasts, sugar_index, sugar):
    # Check if it's a leaf node (has no children)
    if node.left is None and node.right is None:
        species = node.label.split('|')[0]  # Extract the species name from the label
        # Add sugar metabolism data to the node label
        if species in yeasts and yeasts[species][sugar_index[sugar]]:
            node.label += ' (True)'
        else:
            node.label += ' (False)'
    else:
        # Recursive calls for non-leaf nodes
        if node.left is not None:
            annotate_tree(node.left, yeasts, sugar_index, sugar)
        if node.right is not None:
            annotate_tree(node.right, yeasts, sugar_index, sugar)
```

```
# Annotate the tree for each sugar
for sugar in sugars:
    # Start at the root and annotate the whole tree
    annotate_tree(tree.root, yeasts, sugar_index, sugar)
    # Save the annotated tree to a Newick file
    writeNewickFile(f'annotated_MalS_tree_{sugar}.nwk', tree)

    # Print the annotated tree in Newick format for verification
    print(f"Annotated Phylogenetic Tree for {sugar} in Newick Format:")
    print(tree)
```



The annotations (True) across multiple species such as Saccharomyces cerevisiae, S. paradoxus, S. mikatae, and S. bayanus suggest that the ability to metabolize maltose is highly conserved among these closely related species. This indicates that maltose metabolism is a fundamental and possibly advantageous trait maintained through speciation events within this clade. Different strains of the same species (e.g., different S. cerevisiae strains like IMA1, IMA2, IMA3, IMA4, and others) consistently show the ability to metabolize maltose, which underscores the nature of this trait for the species' ecological fitness. The widespread presence of maltose metabolism capability among different yeast species suggests that this trait was present in their common ancestor and has been retained due to its evolutionary advantage.

Exercise 12: Identify the active sites at inferred ancestral sequences (2 marks)

Submit: Provide your commented code (1 mark) and the labelled phylogenetic tree (1 mark).

```
from guide import *

# Load the alignment including gaps
aln = readClustalFile('select.aln', Protein_wGAP)
tree = runUPGMA(aln, 'poisson')
# Associate the alignment data with the tree, preparing for sequence
operations like parsimony.
tree.putAlignment(aln)
# Perform a maximum parsimony analysis to infer ancestral sequences for the
internal nodes of the tree.
tree.parsimony()

def strSites(node, columns):
    """Annotate the node with a string representing active sites."""
    if node.left is None and node.right is None:  # It's a leaf node
        active_sites = "".join(node.sequence[pos] for pos in columns)
        node.label += f":{active_sites}"
    else:  # Internal node
        active_sites = "".join(node.sequence[pos] for pos in columns)
        # Append this string to the current node's label for display.
        node.label += f":{active_sites}"
        # Recurse for child nodes if they exist, passing the same active site
columns.
        if node.left:
            strSites(node.left, columns)
        if node.right:
            strSites(node.right, columns)

def toNewick(node):
    """Convert a tree node to a Newick string format with active site
annotations."""
    if node is None: # Handle the base case where the node is None.
        return ""
    # Format the node's label and distance to its parent, if distance is
available.
    label = f"{node.label}:{node.dist:.4f}" if node.dist is not None else
node.label
    if node.left is None and node.right is None: # If the node is a leaf,
return its label only.
        return label
    else:
        # Recursively call toNewick for the left and right children and
combine them in the proper format.
```

```
        left_str = toNewick(node.left)
        right_str = toNewick(node.right)
        return f"({left_str},{right_str}){label}"

# Define columns corresponding to active sites
active_site_columns = [172, 230, 231, 232, 233, 293, 294, 323, 436]
# Annotate the entire tree with active sites starting from the root.
strSites(tree.root, active_site_columns)
# Convert the entire annotated tree into a Newick format string.
newick_str = toNewick(tree.root) + ";"
print("Newick format with active sites:", newick_str) # Output the Newick
string to console.
with open("annotated_tree.nwk", "w") as file: # Write the Newick string to a
file for external analysis.
    file.write(newick_str)
```