Softwarewerkzeuge der Bioinformatik

Prof. Dr. Volkhard Helms
PD Dr. Michael Hutter, Markus Hollander,
Andreas Denger, Marie Detzler, Velik Velikov
Winter Semester 2021/2022

Saarland University
Center for Bioinformatics

Exercise Sheet 3

Sequence Analysis: Multiple Sequence Alignment (MSA) and Phylogeny

Learning objective: *The goal is to learn how to generate multiple sequence alignments, how to interpret them e.g. regarding sequence conservation and how to generate phylogenetic trees. Additionally, you are going to apply the Sankoff algorithm and learn more about Python.*

Exercise 3.1: Homologous sequences, conserved domains and phylogenetic trees

a) Apply ProteinBLAST to the protein **Q38856**. Remember to select the correct database. Select the first 10 sequences and save them in multi–fasta–format by clicking on "Download" → "FASTA (complete Sequence)". Analogously, save the first 50 sequences in another file.

b) Click on "Distance tree of results" and look at the phylogenetic tree of the first 50 sequences. Select "Taxonomic Name" under "Sequence Label" to see the species labels. Find the tree biological groups (plants, fungi and animals).

c) On the website http://www.ebi.ac.uk/Tools/msa you can find different tools for generating multiple sequence alignments. Select a tool and apply it with default parameters to the file with 10 sequences. Completely conserved positions are marked with "*", high conservation with similar attributes as ":", and less conserved positions with ".". Find contiguous and **highly** conserved parts in the alignment.

d) Lets assume you want to find the active centre of a protein but only have the protein sequence and not the structure. How can a multiple sequence alignment help to solve that problem?

e) Generate an MSA of the 50 sequences with the same tool. Which differences can you see between the two alignments?

Exercise 3.2: Outgroup

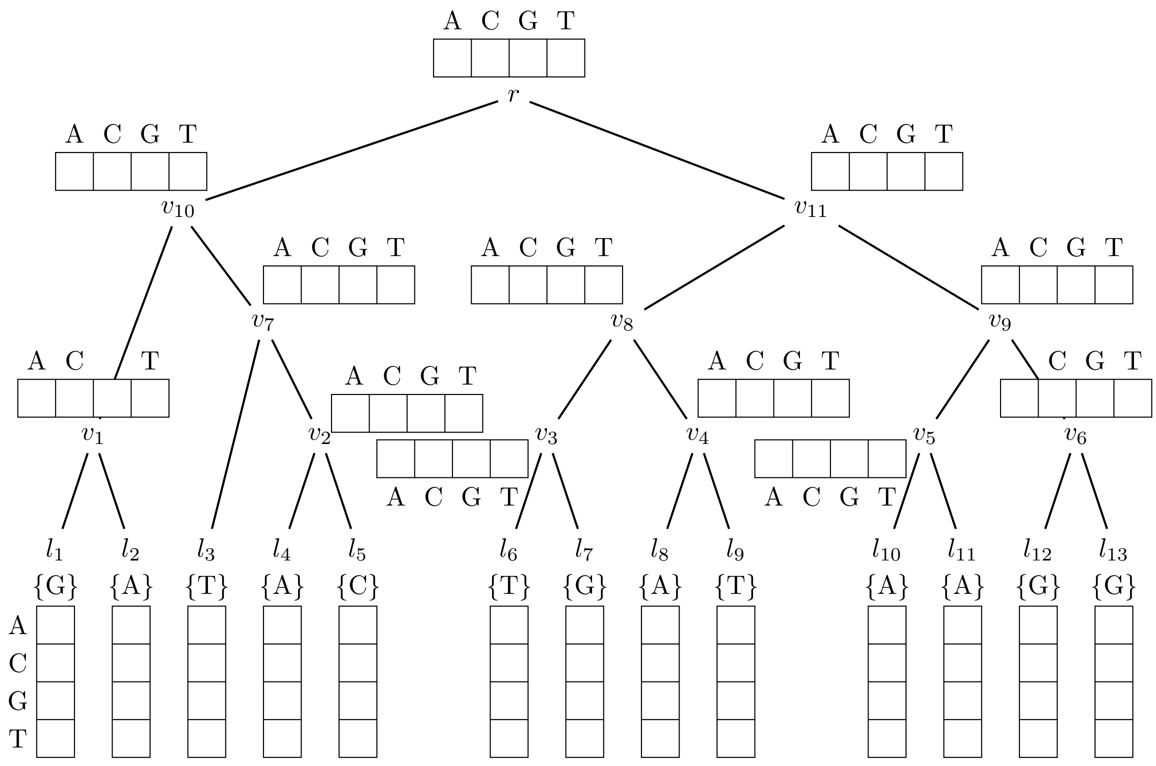
a) Generate an MSA of the sequences provided on the lecture website (sequences.fasta). Is everything conserved?

b) The sequence of which species differs the most from the rest when you just look at the alignment?

c) Use the alignment tool to construct a phylogenetic tree. The sequence of which species differs the most from the others according to the tree?

Exercise 3.3: Sankoff algorithm

Which base was likely in the ancestor sequence at the given position of the alignment? Use the Sankoff algorithm and the given cost function.



$\longrightarrow$ Base in the ancestor sequence:

Cost function:

	A	C	G	T
A	0	2	1	2
C	2	0	2	1
G	1	2	0	2
T	2	1	2	0

Exercise 3.4: Amino acid sequences (Python)

a) Go to kaggle.com and log in. Create a new notebook, rename it and delete the already existing cell.

b) **Strings:** Strings are sequences of letters, symbols and numbers and thus represent text. In the last tutorial we displayed the string 'Hello World!' with the *print*() function. Strings can be assigned to variables and also be easily combined with "+":

```
part1 = 'MNPTLILAAFCLGIASATLTFDHSLEAQWTKWK'
part2 = 'AMHNRLYGMNEEGWRRAVWEKNMKMIELHNQEYREGK'
complete = part1 + part2
```

Copy the example to an empty code cell. Add a line in which you display the content of *complete* with the *print*() function.

c) You can target individual letters in a string by putting its position (*index*) in square brackets. The index is 0–based, which means that the first letter of a string has index 0 and not 1. To get the first letter of the sequence in b), you therefore write: $complete[0]$. You can also specify the position from the end, e.g. to get the last letter, you write: $complete[-1]$.

Use the *print*() function to display the first, tenth, last and second last letter of *complete*.

d) You can also target sections of a string by putting its start and end index in square brackets. The index is 0–based here as well. The given end position is not included. To target positions 10 to (including) 20 of the sequence, you write: $complete[9:20]$.

Use the *print*() function to display the first 10 letters of the sequence.

e) **Dictionaries:** So called dictionaries are data structures suited to assign values to keys. The keys can be used to retrieve the values. The values can be single words, texts or numbers, and also more complex data types like lists.

For example, here an (incomplete) dictionary with amino acid symbols is assigned to the variable *symbols*. The 1-letter symbols are the keys and the 3-letter symbols the values:

```
symbols = {'A': 'Ala', 'R': 'Arg', 'N': 'Asn', 'D': 'Asp', 'C': 'Cys',
           'Q': 'Gln', 'E': 'Glu', 'G': 'Gly', 'H': 'His', 'I': 'Ile',
           'L': 'Leu', 'K': 'Lys', 'M': 'Met', 'F': 'Phe', 'P': 'Pro',
           'S': 'Ser', 'T': 'Thr', 'W': 'Trp'}
```

Copy the example to an empty code cell and complete it with the missing symbols for tyrosine and valine.

f) Similar to strings, you can retrieve values from a dictionary by putting the key in square brackets. For example, $symbols['A']$ returns the 3-letter code 'Ala'.

Use the *print*() function to display the 3-letter codes of serine and threonine.

g) **For–Loops:** So called for–loops are useful to automatically execute an operation repeatedly. In this example, we iterate with the key word *for* over all amino acids *aa* in the sequence *complete* and use our dictionary to display the 3-letter code of each amino acid.

```
for aa in complete:
    print(symbols[aa])
```

This is also the first example in which we can see that python uses indentation to recognise code sections. Everything that is indented to the right beneath the for–loop is exectued in each iteration.

Copy th example into an empty cell and run it.

h) **Functions:** You already used the built–in *print*() function which display things. Functions are very useful if we want to do something often with different input. In the following example we create with the key word *def* a new function *transform* that takes a sequence (*sequence*) as input. The function iterates with a for–loop over all amino acids in the input sequence and adds the 3–letter code to the varaible *new_sequence*. At the end, the function returns the new sequence with the key word *return*.

Everything that is not indented to the right does no longer belong to the function. The function can now be used just like *print*() and you can assign the result to variables, like e.g. *transformed_sequence*.

```
def transform(sequence):
    new_sequence = ''
    for aa in complete:
        new_sequence += symbols[aa]
    return new_sequence

transformed_sequence = transform(complete)
print(transformed_sequence)
```

Copy the example to an empty cell and run it. Apply the function to some made-up amino acid sequences.

Have fun!