ICC (SV) – Mini-project

Barbara Jobstmann, Jamila Sam, Samuel Teeporten

Amir Bouchaoui, Miguel Crespo

(Version 1.1)

This document uses color and contains clickable links. It is best viewed in digital format. It is very important to read it carefully and **entirely** and to **understand the data structures provided** as well as the role of each function before you start coding.

1 Introduction

A phylogenetic tree aims to convey the relationships between different entities over time. It shows the evolutionary proximity between different entities, in terms of hereditary characteristics (such as the morphologic traits, DNA/RNA sequences, amino acid sequences, etc.). Figure 1 shows a phylogenetic tree of living things, based on RNA data and proposed by Carl Woese. It shows the separation of bacteria, archaea, and eukaryotes. The entities that are close to each other in this tree have similar RNA sequences.

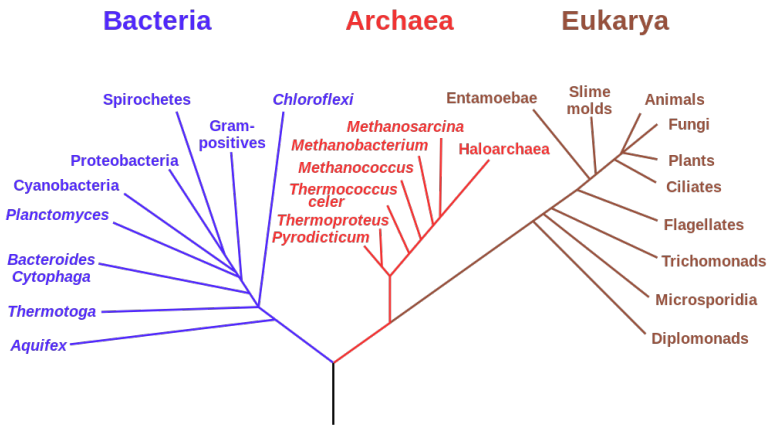

Figure 1: Phylogenetic tree of living things https://en.wikipedia.org/wiki/File:Phylogenetic_tree.svg

The phylogenetic trees are important as they help us understand how different organisms are linked by evolution. They show how species have changed and branched out over time. By examining these trees, scientists can determine which species have common ancestors, follow gene evolution, and observe adaptation schemes. These trees are used

in many domains like biology, ecology, and medical research to study the diversity and the connections in the living world.

For instance, in epidemiology, such trees help track how diseases spread and evolve over time. By analyzing the genetic relationships between different strains of a virus or bacteria, scientists can understand how infections spread in communities and identify potential sources of outbreaks.

In cancer research, phylogenetic trees help researchers understand the genetic changes that lead to tumor development and progression. By studying the evolutionary history of cancer cells within a tumor, scientists can identify key mutations and pathways involved in cancer growth, which can inform the development of targeted therapies and personalized treatment approaches.

2 Objectives and Example

This project aims to create phylogenetic trees using the two following methods of hierarchical regrouping:

1. WPGMA (Weighted Pair Group Method with Arithmetic Mean)
2. UPGMA (Unweighted Pair Group Method with Arithmetic Mean)

We illustrate the WPGMA algorithm using a small example. The UPGMA algorithm is very similar and will be explained later. Imagine we have four organisms with different genetic codes, i.e., DNA sequences consisting of the four DNA bases Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). Since the genetic code of an entity can be very large one usually focuses on a part of interest to compare the organisms (e.g., one gene or less). We will call a classifiable element like a part of a DNA sequence a **taxon**. Assume we would like to analyze the relationship of the following four DNA fragments (T1, T2, T3, T4).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T1	C	A	T	A	G	A	C	C	T	G	A	C	G	C	C	A	G	C	T	C
T2	C	A	T	A	G	A	C	C	C	G	C	C	A	T	G	A	G	C	T	C
T3	C	G	T	A	G	A	C	T	G	G	G	C	G	C	C	A	G	C	T	C
T4	C	C	T	A	G	A	C	G	T	C	G	C	G	G	C	A	G	T	C	C

The starting point of this clustering method is a distance matrix that shows the distance between any two entities. We compute the distance between two entities by comparing them letter by letter. E.g., the distance between Taxon T1 and Taxon T2 is 5 because they differ in Positions 8, 10, 12, 13, and 14. Computing the pairwise distances for the four taxa gives the distance matrix shown in Table 1.

Initially, each entity is in its own cluster. The algorithm proceeds in steps. In each step, two clusters with a minimal distance are merged to form a new cluster. The distance of the new cluster to any other cluster C is the average between the distances of the two merged clusters to C. For instance, in our example, initially, each taxon is in its own cluster, and the minimal distance between two clusters is 4, which is the distance between

Table 1: Distance between T1, T2, T3, and T4

	T1	T2	T3	T4
T1	0	5	4	7
T2	5	0	7	10
T3	4	7	0	7
T4	7	10	7	0

Table 2: Distances between T1T3, T2, and T4

	T1T3	T2	T4
T1T3	0	6	7
T2	6	0	10
T4	7	10	0

Table 3: Distances between T1T3T2 and T4

	T1T3T2	T4
T1T3T2	0	8.5
T4	8.5	0

T1 and T3. We create a new cluster merging T1 and T3. The distance between the new cluster T1T3 and T2 is the average of the distance between T1 and T2, which is 5, and the distance between T3 and T2, which is 7. So, the distance between T1T3 and T2 is $(5 + 7)/2 = 6$. The distance between T1T3 and T4 is $(7 + 7)/2 = 7$. Table 2 shows the new distance matrix after merging T1 and T3. In the next step, we merge Cluster T1T3 with T2 and obtain the distance matrix shown in Table 3. We will continue merging clusters until we obtain a single cluster.

The phylogenetic tree is built by following the clustering steps (grouping the closest clusters) and distributing the distances over the different branches. Merging T1 and T3 gives us a tree with two branches, one for T1 and one for T3. The distance between T1 and T3 is 4, which will be evenly distributed over these two branches, so each branch is assigned the value 2. Figure 2 shows the tree corresponding to the first merging step. The next step, which merges T1T3 with T2 extends the tree by one level with T1 and T3 on one side and T2 on the other (see Figure 3). The distance between T1T3 and T2 is 6, so each of these two new branches is assigned the value 3. Since the left subtree (with T1 and T3) already has value 2, we only need to add 1 to get to a total of 3. In the final step, the cluster T1T3T2 is merged with T4 which gives another layer in the tree. The distance between these two clusters is 8.5, which gives 4.25 for each new branch. Figure 4 shows the final tree generated for T1, T2, T3, and T4 following the WPGMA algorithm.

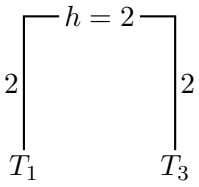

Figure 2: Tree after stage 1

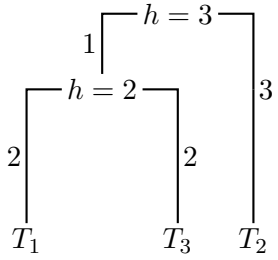

Figure 3: Tree after stage 2

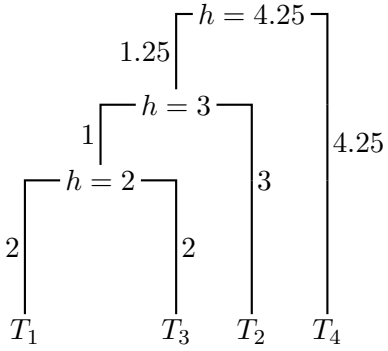

Figure 4: Final tree

3 Structure and code

This project is divided into three parts:

1. Implementation of debugging functions.
2. Initializing the data structures representing a phylogenetic tree and computing the distance matrix.
3. Building the phylogenetic trees.

Within each part of the project, you need to implement several functions. Utility types and functions are provided in the files `utils.[hpp][.cpp]`. The data types and structures to be used are defined in the file `phylogenetics.hpp`.

It is important to review the types and functions included in the file `phylogenetics.hpp` and to understand their roles.

Note that if we do not use the directive `using namespace std;` we must change the identifiers `cout`, `vector` etc. by adding `std::` as a prefix (i.e., `std::cout`, `std::vector`, `std::endl`). This is commonly done in C++ and the provided material is written this way (we will come back to this in the next semester).

You need to fill in the code in the file `phylogenetics.cpp`. Precise instructions are given in this document. The prototypes of the functions to be implemented are in the file `phylogenetics.hpp`. They **must not be modified**.

Note that in addition to the mandatory functions, you are free to define any additional functions or data structures that you consider relevant.

Be careful: any additional function or data type must be written in the file **`phylogenetics.cpp` because this is the only file that will be submitted**.

It is a good practice to create clean and modular code. In the appendix (Section A), at the end of the document, you can find a description of the difference between UPGMA and WGPMA.

Note that in the examples of code (or program executions) in this document, you may see an arrow like this ↪ . It refers to a line break added only in this document to make the example easier to read. The new line does not exist in the original snippet.

3.1 Utility functions

A taxon, such as a protein, can be a very long sequence of characters. It can therefore become tedious to enter as program input at every execution.

The file `utils.hpp` provides the utility function `readSequencesFromFile(std::string filename)` for reading character sequences from a file named `filename`. You can find examples of how to use this function in the provided file `sources/main.cpp`.

The file `phylogenetics.hpp` also includes the type `Taxon`, which is a `string`, and the enumeration type `AlgoType`, which is used to select the algorithm that we want to use for building the phylogenetic tree.

3.2 Included test files

We include some data in the folder `data` to test your code. Each text file (`.txt`) in this directory will correspond to a group of taxa (taxons) to be classified. You are free to add your own files for testing purposes but they will not be submitted.

Test and verification of edge cases

It is common practice to check the input parameters of a function; for example, to verify that a table is not empty, and/or has correct dimensions,...These tests generally make debugging easier, and help you to reason about the behavior of a function.

We will suppose that the arguments of the functions are correct by default and that it is not necessary to make such verifications. In the cases in which it is necessary, you should use the following type of C++ instruction to stop the program if a parameter is invalid:

```
if (condition) throw std::invalid_argument(" an error message ");
```

If the condition is `true`, the code will display the error message in the terminal and will stop the program. Otherwise, it will continue its execution.

As an example, if a pointer parameter called `key` of a function called `f` should not be `nullptr` for the function to work correctly, you can write

```
if (key == nullptr) throw std::invalid_argument(" key must not be
    ↪ nullptr in function f ");
```

at the beginning of the body of `f`. Note that, in general, this type of instruction can be placed anywhere in the program. This mechanism is called *exception handling* and will be studied in detail in the next semester.

> To use the exceptions mechanism, it is necessary to include the library `exception`.

It is your responsibility to verify that the program behaves correctly. It is crucial to **check carefully at each stage that you are producing the correct data before moving on to the next stage, which will use that data.**

To help you with the verification, we provide a few examples of tests in the files `main.cpp` and `unit_test.cpp` (the tests of the latter file are invoked in the file `main.cpp`). You can use these tests and fill them in to ensure that everything works properly. Note that the code in `main.cpp` and `unit_test.cpp` will not be graded. Modify them as you need.

To test the code that you have written for the different parts, you can call the function `run_unit_tests(part)` in which `part` is replaced by the number of the part that you want to check. For example, if you want to test your code for Part 3 of the project, call the function `run_unit_tests(3)` in the file `main.cpp`.

IMPORTANT The included tests are not exhaustive. It is recommended that you fill them in, especially for handling edge cases. Finally, it is highly recommended to familiarize yourself from the beginning with the usage of the debugger (see the tutorial of week 8). Add breakpoints inside the functions to stop the code during the execution and to examine the involved values.

4 Implementation

This project has three parts. In each of them, you need to implement some functions in the file `phylogenetics.cpp`.

4.1 Part 1

The aim of the first part of the project is to implement the functions that will be used to print the content of the required data structures.

A cluster is a node in a phylogenetic tree. For example, the tree of Figure 3, will be represented using the following five clusters:

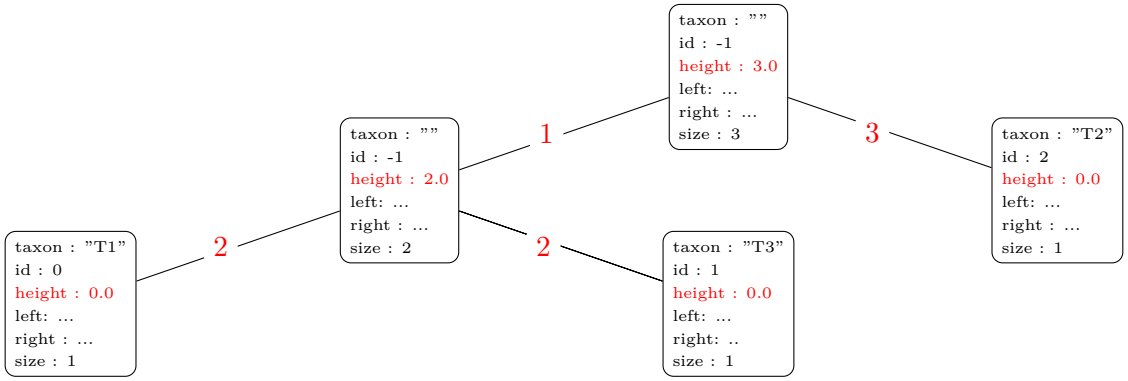

Figure 5: Phylogenetic tree : detailed view of the nodes (the information in red on the links between the nodes is not part of the representation of the `Cluster`)

The data structure `Cluster` represents a cluster. **It must not be modified** and contains the following information:

- **`id`**: An integer identifier of the node; this identifier is important only in the terminal nodes (leaves); its value will be -1 otherwise;
- **`taxon`**: A string of type `Taxon` identifying the leaf nodes. Concretely, the value of this field is:
  - the name of the taxon (e.g., an RNA sequence) for the leaf nodes;
  - an empty string for the non-leaf nodes.
- **`left`** & **`right`**: Pointers to the two nodes aggregated by the current node, a pointer for the left node and another for the right one;
- **`height`**: the height of the node that gives its distance to the leaf nodes, as explained in the introduction example. For example, the distance separating T1 and T3 is four (according to the distance matrix in Table 1), and the height of the node T1+T3 is four divided by two (distributing the distance equally on the two branches).
- **`size`**: the size of the node is defined as the number of nodes it aggregates. In Figure 3, the node aggregating T1 (id=0) and T3 (id=1) has a size of two; the one

which aggregates this node (of size 2) with the node T2 (of size 1) has as size of three. Note that the size cannot be negative and is an integer.

Before continuing, take a look at the provided type aliases (in `phylogenetics.hpp`). You will see that a tree (`Tree`) is a vector of `Cluster`. The data structure `Tree` is used jointly with the data structure `DistanceMatrix`, which stores the distances between any two `Cluster` in the tree vector. **The order in these data structures is important**: the entry `[i][j]` in a `DistanceMatrix` gives the distance between the entry `i` and the entry `j` of the corresponding `Tree`.

You are then required to implement some useful functions for printing the contents of data structures. Their use will help you when debugging your program. Each of the following functions returns a *string* formatted according to a precise specification. These strings can then be printed to see the contents of the corresponding data structures.

Please make sure that your output matches the one provided in the handout and in the unit tests.

Functions not yet implemented contain the line `NotImplemented();` which will cause an error if the function is executed. Remove this line when you implement a function.

Here is the list of functions to implement:

- `std::string toString(const DistanceMatrix& matrix, bool verbose = false ↪ );` When `verbose` is `false`, the return string is constructed so that the values of the distance matrix are separated by a space, and a line break is made between each line (the character `'\n'` represents a line break). When `verbose = true`, the string represents the distance matrix as a set of lines with the following format: $i - j = dist_{i,j}$, where i is a row number, j a column number, and $dist_{i,j}$ is the value of the entry $[i][j]$ of the matrix. This entry must be converted to a `string` using the provided function `double_to_string` (which limits the number of decimal points).

  The part of the function `test_part1` dedicated to this function and called in `main` must produce the following output for the matrix `dist` (the `toString` method is called twice, once with the `verbose` parameter set to `false` and again with it set to `true`):

```
 ======= Testing toString for DistanceMatrix =======
 ======= printing in non verbose mode by default ====
0 17 21 31 23
17 0 30 34 21
21 30 0 28 39
31 34 28 0 43
23 21 39 43 0
 ======= printing in verbose mode ====
0 - 0 = 0
0 - 1 = 17
0 - 2 = 21
0 - 3 = 31
0 - 4 = 23
1 - 0 = 17
1 - 1 = 0
1 - 2 = 30
1 - 3 = 34
1 - 4 = 21
2 - 0 = 21
2 - 1 = 30
2 - 2 = 0
2 - 3 = 28
2 - 4 = 39
3 - 0 = 31
3 - 1 = 34
3 - 2 = 28
3 - 3 = 0
3 - 4 = 43
4 - 0 = 23
4 - 1 = 21
4 - 2 = 39
4 - 3 = 43
4 - 4 = 0
 ======= End testing toString for DistanceMatrix =======
```

- `std::string toString(const ClusterIdPair& pair);` returns a string representation of a `ClusterIdPair` using the format `id1-id2\n`, where `id1` is the first element of `pair`, and `id2` the second. The `'\n'` character is a line break in C++. For example, given that `pair` is initialized with 2 and 3, printint its corresponding `string` should produce:

  ```
  2-3
  ```

  followed by a line break.

- `std::string clusterToString(const Cluster* cluster, bool verbose = false ↪ );` this function creates recursively a `string` representation of a `Cluster`: if it is a terminal node, it will output the corresponding taxon, if not, it will output the `string` representation of its left child followed by its right one, separated by

a comma and between parenthesis. If `verbose` is `true`, the taxon will be printed in its entirety followed by the values of the fields `id`, `height`, and `size`, separated by commas and surrounded by brackets (`[]`) (see the execution trace below for an exact description of the expected format). The value of the `height` field must be converted to a string using the provided function `double_to_string`. If `verbose` is `false`, a shorter representation will be produced by concatenating the word `Taxon` with the integer identifier of the taxon. If `cluster` is a `nullptr`, the function will return an empty string. The function `test_part1` called in `main` provides the expected result for a network similar to the one in Figure 3 :

```
======= Testing clusterToString =======
======= printing in non verbose mode by default ====
((Taxon_0,Taxon_2),Taxon_1)
======= printing in verbose mode ====
((ACGTAACCTTGGG[i:0,h:0,s:1],GTAGTAGTAGTAG[i:2,h:0,s:1])[i:-1,h:2,s
   ↪ :2],AGGGTCTATATGT[i:1,h:0,s:1])[i:-1,h:3,s:3]
======= End testing clusterToString =======
```

- `std::string toString(const Tree& tree, bool verbose = false);` returns the `string` representations of all the clusters of the given tree separated by a line break. If `verbose` is `true`, taxa (taxons) in the tree will be displayed in verbose mode. The function `test_part1` called by `main` must generate the following display for the trees `tree1` and `tree2` :

```
======= Testing toString for Tree =======
printing tree1:
---------------
 ======= printing in verbose mode ====
ACGTAACCTTGGG[i:0,h:0,s:1];
AGGGTCTATATGT[i:1,h:0,s:1];
 ======= printing in non verbose mode ====
Taxon_0;
Taxon_1;
printing tree2:
---------------
 ======= printing in verbose mode ====
((ACGTAACCTTGGG[i:0,h:0,s:1],GTAGTAGTAGTAG[i:2,h:0,s:1])[i:-1,h:2,s
   ↪ :2],AGGGTCTATATGT[i:1,h:0,s:1])[i:-1,h:3,s:3];
 ======= printing in non verbose mode ====
((Taxon_0,Taxon_2),Taxon_1);
 ======= End testing toString for Tree =======
```

Tests

To test this part, you can use the provided main program. Open the file `main.cpp` and you will see the provided tests. **Uncomment the relevant lines to test your code**. The function `test_part1` gives you an example of simple tests that you can complete as you wish for the first part of the project. In addition, the lines towards the end of the function `main` execute the function `run_unit_tests`, which is defined in `unit_test.cpp` Open the file `unit_test.cpp` and review the function `run_unit_tests`. You will see that it calls different provided tests for each part of the project.

For the first part of the project, you can call the function `run_unit_tests(1)`, which will execute just the tests for this part. The execution must display the message `[Passed]` for all tests. If a test fails, you will see a message saying `[Failed]` and the difference between the expected and the computed value.

For example:

```
---------------
test_distanceMatrixToString
------------------------
[Passed]
[Passed]
[Passed]
[Passed]
```

showing that the function `toString(const DistanceMatrix&)` has passed the tests.

If not, it will display something like:

```
--------------
test_distanceMatrixToString
------------------------

[Failed]
 expected:
 0 17 21
 17 0 30
 21 30 0
 found:
 0 17 21 17 0 30 21 30 0
```

indicating that `toString` has produced the output shown after `found`, instead of the correct one shown after `expected`.

4.2 Part 2

Note: if `ptr_c` is a pointer on a cluster and `m` is the name of one of the cluster's fields, then the notations `(*ptr_c).m` and `ptr_c->m` are equivalent in C++.

In the second part of the project, you will implement the functions needed to initialize the data structure representinga phylogenetic tree and to compute the distance matrix. These functions are:

- `int calculateDistance(const Taxon& seq1, const Taxon& seq2);` compute the distance between the taxon `seq1` and the taxon `seq2`. This distance will be the number of times `seq1[i]` is different from `seq2[i]` for all `i`. The two taxa (the two taxons) can have different lengths: the extra characters will also be counted as differences.
- `Tree initTree(std::vector<Taxon> taxa);` returns an initial tree built using the taxa passed as a parameter. For each taxon of the group `taxa`, a dynamically created cluster will be added to the tree. The clusters will be numbered in the order of construction. This number will be the integer identifier of a cluster. The numbering starts with zero.
- `void deleteTree(Tree& tree);` which empties a given tree and **frees the memory associated to all of its clusters**. This function should work with no crash even if some of the `clusters` in `tree` are `nullptr`. Be careful, an incomplete deletion can lead to errors later, in the last part of the project.
- `DistanceMatrix initDistanceMatrix(const Tree& tree);` builds and returns the distance matrix corresponding to the tree passed as a parameter. For each pair `(i,j)` of clusters in the tree, the entry `[i][j]` of the matrix will be computed as the distance between the taxa (between the taxons) in the clusters i and j. The distance is computed using the function `calculateDistance`. This function only makes sense if the tree passed as a parameter has only leaf nodes and if none of the leaves is `nullptr`, which must be ensured using the exception mechanism. You should write a function `isLeafNodeTree` that tests this first condition. This function will return `true` only if the size of all the clusters inside the tree is one.
- `void eraseColumn(DistanceMatrix& distances, size_t c)` erases the column with the index `c` from a distance matrix; `c` should be compatible with the actual size of the matrix, otherwise an exception should be thrown;
- `void eraseRow(DistanceMatrix& distances, size_t r)` erases the row with the index `r` from the distance matrix; `r` should be compatible with the actual size of the matrix, otherwise an exception should be thrown.

Note: To remove an element at position `p` in a `vector v` use the following instruction:

```
v.erase(v.begin() + p);
```

Attention, if the instruction `v.erase(v.begin() + p);` is used in an iteration, it must be a classic iteration (`for (size_t i(..);..;..)`) and not an iteration over a set of values (`for (auto val :...)`). In a `for (auto val :...)` loop, if the size of the collection is changed while iterating over it, the program will crash (explanations will be provided on next semester!).

Tests

Use the function `test_part2` to test this part of the project. The expected outputs for this part are given in the Appendix A.2.1. You can also run the corresponding unit tests of this part by calling `run_unit_tests(2)`
If your code is correct, you should only see `[Passed]` messages.

4.3 Part 3

In the third part, you should implement the phylogenetic tree building algorithms. The enumerated type `AlgoType` will be used to indicate the variant to be used (`UGPMA` or `WGPMA`).

Here are the functions that you should implement (for each one of them, you will find an execution example in the function `test_part3`):

- `ClusterIdPair minimumDistance(const DistanceMatrix& distances);` returns the pair of distinct `Clusters` separated by the smallest distance in the matrix `distances`. The matrix must be traversed line by line and for each line, column by column. If the minimal distance is present several times, only the pair with the smallest indexes (the first encountered) will be returned. For example, for the distance matrix:

  ```
  {
    {0, 1, 2},
    {1, 0 , 1},
    {2, 1 , 0}
  }
  ```

  The pair `{0,1}` should be returned. This means that the two closest `Clusters` (in terms of distance) are those occupying Position 0 and Position 1 in the phylogenetic tree corresponding to this distance matrix. This function will only work correctly if the parameter is a non-empty square matrix. Use the exception mechanism to guarantee this condition.

  Indication: the largest possible value for a `double` is given by `std::numeric_limits<double>::max()`.

- `void mergeClusters(const ClusterIdPair& pair, Tree& tree, DistanceMatrix & distances, AlgoType algorithm = WPGMA);` merges the pair of `Clusters`

identified by `pair` in the tree `tree` and adapts the distance matrix `distances` accordingly. The cluster obtained by the merging operation will have the data as described by the example in Figure 5.
You should use the same way of coding the identifiers (either the integer identifier or the one of type `Taxon`) as in this example. The merge is done so that in the tree `tree`, the node with the position `pair[0]` is replaced with the new node obtained by merging, and the node in the position `pair[1]` disappears from the tree. The matrix of distance will be adapted consequently: for each possible index `p`, the entries `[pair[0]][p]` and `[p][pair[0]]` will be re-evaluated according to the specific modalities of the algorithm used (average of distances or average of distances weighted by node size, depending on whether `WPGMA` or `UPGMA` is used, see the difference between the two in the appendix). For example, the first call of the function `test_merge_clusters`, called by `test_part3`, should produce the following output:

```
======= Testing mergeCluster (WPGMA) =======

The pair to merge:
0-1

Before merging:
-----------------
Tree:
------
ACGTAACCTTGGG[i:0,h:0,s:1];
ACGGTCTATTGGA[i:1,h:0,s:1];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 6 11
6 0 11
11 11 0

After merging clusters:
------------------------------
Tree:
------
(ACGTAACCTTGGG[i:0,h:0,s:1],ACGGTCTATTGGA[i:1,h:0,s:1])[i:-1,h:3,s
    ↪ :2];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 11
11 0

 ======= End testing mergeCluster (WPGMA) =======
```

This function works correctly only if the parameters supplied have a correct structure: elements in `pair` must be different and compatible with the size of the matrix, `tree` must contain no `nullptr`, the size of the tree must be the same as the size of the distance matrix, and the latter must be square. These conditions must be guaranteed using the exception mechanism.

- `void buildPhylogeneticTree(Tree& tree, DistanceMatrix& distances, AlgoType algorithm = WPGMA);` applies the algorithm that merges the nodes closest in terms of distance, until only one node remains in the tree. The tree will then represent the desired phylogenetic tree. For this function to work properly, `tree` must contain no `nullptr`, the size of the tree must be the same as the size of the distance matrix, and the latter must be square. If any of theses conditions is not verified an exception must be thrown.

- `std::string phylogeneticTreeToString(const Cluster* root, bool verbose = false);` allows you to display a phylogenetic tree more visually, starting from its root and displaying the height of each cluster (see below for an example of the desired display). If `root` is `nullptr`, an empty string will be returned.

  Indication: use a recursive helper function taking an additonal parameter representing the "depth" of a given recursive call (this depth will give the number of `'|'` symbols in a given line).

The function `test` called by `test_part3` , should produce the following display for the Wikipedia example :

```
----- Wikipedia Example with UPGMA ----
(((Taxon_0,Taxon_1),Taxon_4),(Taxon_2,Taxon_3));
 (16.5)
| +--- (11)
| | +--- (8.5)
| | | +---a
| | | +---b
| | +---e
| +--- (14)
| | +---c
| | +---d
----- Wikipedia Example with WPGMA ----
(((a[i:0,h:0,s:1],b[i:1,h:0,s:1])[i:-1,h:8.5,s:2],e[i:4,h:0,s:1])[i:-1,h
    ↪ :11,s:3],(c[i:2,h:0,s:1],d[i:3,h:0,s:1])[i:-1,h:14,s:2])[i:-1,h
    ↪ :17.5,s:5];
 (17.5)
| +--- (11)
| | +--- (8.5)
| | | +---a
| | | +---b
| | +---e
| +--- (14)
| | +---c
| | +---d
```

Tests

To test this part, you can call the function `test_part3`. The expected outputs for this part are given in Appendix A.2.2. You can also call the unit test function as follows: `run_unit_tests(3)`
If your code is correct, you should get only the messages `[Passed]`

A Appendix

A.1 UPGMA vs WGPMA

The two algorithms WPGMA and UPGMA differ only in the way they compute the distances for a newly created cluster. Recall that in WPGMA, if cluster C_1 and C_2 are merged into cluster C_1C_2 then the distance between C_1C_2 and a third cluster C_3 is the average between the distances, i.e.,

$$d(C_1C_2, C_3) = \frac{d(C_1, C_3) + d(C_2, C_3)}{2}.$$

In UPGMA, the distance depends also on the sizes of the clusters, i.e., if a cluster includes more elements, it has more influence on the distance. More precisely, the distance between the merged cluster C_1C_2 and a third cluster C_3 is given by the proportional average, i.e.,

$$d(C_1C_2, C_3) = \frac{|C_1| \cdot d(C_1, C_3) + |C_2| \cdot d(C_2, C_3)}{|C_1| + |C_2|},$$

where $|C_1|$ and $|C_2|$ refer to the size of C_1 and C_2, respectively.

In the following, we will apply the UPGMA clustering method to the example from Section 2. We start with the distance matrix given in Table 4. The minimal distance is between T_1 and T_3, which leads to a new cluster T_1T_3. Initially, the size of all clusters is 1 because each of them includes only one taxon. Therefore, the distances between T_1T_3 and the other clusters are the same for both algorithms (WPGMA and UPGMA). They are shown in Table 5. In the next step, we merge the clusters T_1T_3 and T_2. Note that T_1T_3 has a size of 2 and T_2 has a size of 1. Using this information the distance between the new cluster $T_1T_3T_2$ and T_4 is computed as follows:

$$d(T_1T_3T_2, T_4) = \frac{2 \cdot d(T_1T_3, T_4) + 1 \cdot d(T_2, T_4)}{2+1} = \frac{2 \cdot 7 + 1 \cdot 10}{2+1} = 8$$

This leads to the final distance matrix shown in Table 6 and the corresponding tree shown in Figure 6.

Table 4: Distance between T_1, T_2, T_3, and T_4

	T_1	T_2	T_3	T_4
T_1	0	5	4	7
T_2	5	0	7	10
T_3	4	7	0	7
T_4	7	10	7	0

Table 5: Distances between T_1T_3, T_2, and T_4

	T_1T_3	T_2	T_4
T_1T_3	0	6	7
T_2	6	0	10
T_4	7	10	0

Table 6: Distances between $T_1T_3T_2$ et T_4

	$T_1T_3T_2$	T_4
$T_1T_3T_2$	0	8
T_4	8	0

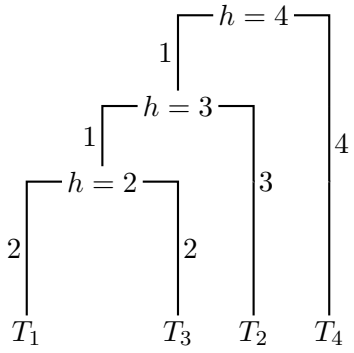

Figure 6: Final tree for UPGMA

A.2 Outputs of the provided tests

A.2.1 Part 2

```
 ========= TESTING PART 2 ================
 ======= Testing calculateDistance =======
Distance between CGTAACCTTGGG and CGTGAGCTTA: 5
 ======= end Testing calculateDistance =======
 ======= Testing initTree =======
The initial vector of taxa is: {"a", "b", "c", "d", "e"}
Tree constructed by initTree, printed in non verbose mode:
Taxon_0;
Taxon_1;
Taxon_2;
Taxon_3;
Taxon_4;
Same tree printed in verbose mode:
a[i:0,h:0,s:1];
b[i:1,h:0,s:1];
c[i:2,h:0,s:1];
d[i:3,h:0,s:1];
e[i:4,h:0,s:1];
 ======= end testing initTree =======
 ======= Testing initDistanceMatrix =======
Input data: a tree constructed using the taxa ACGTAACCTTGGG,
    ↪ ACGGTCTATTGGA and GTAGTAGTAGTAG
```

```
Distance matrix constructed by initDistanceMatrix:
0 6 11
6 0 11
11 11 0
 ======= end testing initDistanceMatrix =======
 ======= Testing eraseColumn and eraseRow =======
Initial distance matrix:
0 17 21 31 23
17 0 30 34 21
21 30 0 28 39
31 34 28 0 43
23 21 39 43 0
The previous matrix after erasing column 3:
0 17 21 23
17 0 30 21
21 30 0 39
31 34 28 43
23 21 39 0
The previous matrix after erasing row 3:
0 17 21 23
17 0 30 21
21 30 0 39
23 21 39 0
 ======= end testing eraseColumn and eraseRow =======
```

A.2.2 Part 3

```
 ========= TESTING PART 3 ================
 ======= Testing minimumDistance =======
Input data: the distance matrix
Input data: the distance matrix {0, 1, 2}, {1, 0, 1}, {2, 1, 0}
The pair with the minimum distance is:
0-1
 ======= End testing minimumDistance =======
 ======= Testing mergeCluster (WPGMA) =======

The pair to merge:
0-1

Before merging:
-----------------
Tree:
------
ACGTAACCTTGGG[i:0,h:0,s:1];
ACGGTCTATTGGA[i:1,h:0,s:1];
GTAGTAGTAGTAG[i:2,h:0,s:1];
```

```
Distance matrix:
-----------------
0 6 11
6 0 11
11 11 0

After merging clusters:
------------------------------
Tree:
------
(ACGTAACCTTGGG[i:0,h:0,s:1],ACGGTCTATTGGA[i:1,h:0,s:1])[i:-1,h:3,s:2];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 11
11 0

 ======= End testing mergeCluster (WPGMA) =======
 ======= Testing mergeCluster (UPGMA) =======
Input data: a tree with three leaf nodes "ACGTAACCTTGGG", "ACGGTCTATTGGA
    ↪ " and "GTAGTAGTAGTAG"

The pair to merge:
0-1

Before merging:
-----------------
Tree:
------
ACGTAACCTTGGG[i:0,h:0,s:1];
ACGGTCTATTGGA[i:1,h:0,s:1];
GTAGTAGTAGTAG[i:2,h:0,s:1];

Distance matrix:
-----------------
0 6 11
6 0 11
11 11 0

After merging clusters:
------------------------------
Tree:
------
(ACGTAACCTTGGG[i:0,h:0,s:1],ACGGTCTATTGGA[i:1,h:0,s:1])[i:-1,h:3,s:2];
GTAGTAGTAGTAG[i:2,h:0,s:1];
```

```
Distance matrix:
-----------------
0 11
11 0

 ======= End testing mergeCluster (UPGMA) =======
 ======= Testing build and print phylogenetic trees =======
Input data:
Tree:
------
Verbose printing:
CATAGACCTGACGCCAGCTC[i:0,h:0,s:1];
CATAGACCCGCCATGAGCTC[i:1,h:0,s:1];
CGTAGACTGGGCGCCAGCTC[i:2,h:0,s:1];
CCTAGACGTCGCGGCAGTCC[i:3,h:0,s:1];
Non verbose printing:
Taxon_0;
Taxon_1;
Taxon_2;
Taxon_3;

Distance matrix:
-----------------
0 5 4 7
5 0 7 10
4 7 0 7
7 10 7 0
----- Build Phylogenetic Tree with WPGMA ----

After calling buildPhylogeneticTree:
Tree:
------
(((Taxon_0,Taxon_2),Taxon_1),Taxon_3);

Distance matrix:
-----------------
0

Calling phylogeneticTreeToString:

Verbose printing:
 (4.25)
| +--- (3)
| | +--- (2)
| | | +---CATAGACCTGACGCCAGCTC
| | | +---CGTAGACTGGGCGCCAGCTC
| | +---CATAGACCCGCCATGAGCTC
```

```
| +---CCTAGACGTCGCGGCAGTCC
Non verbose printing:
 (4.25)
| +--- (3)
| | +--- (2)
| | | +---Taxon_0
| | | +---Taxon_2
| | +---Taxon_1
| +---Taxon_3

----- Build Phylogenetic Tree with UPGMA ----

After calling buildPhylogeneticTree:

Tree:
------
(((Taxon_0,Taxon_2),Taxon_1),Taxon_3);

Calling phylogeneticTreeToString:

Verbose printing:
 (4)
| +--- (3)
| | +--- (2)
| | | +---CATAGACCTGACGCCAGCTC
| | | +---CGTAGACTGGGCGCCAGCTC
| | +---CATAGACCCGCCATGAGCTC
| +---CCTAGACGTCGCGGCAGTCC
Non verbose printing:
 (4)
| +--- (3)
| | +--- (2)
| | | +---Taxon_0
| | | +---Taxon_2
| | +---Taxon_1
| +---Taxon_3
----- Wikipedia Example with UPGMA ----
(((Taxon_0,Taxon_1),Taxon_4),(Taxon_2,Taxon_3));
 (16.5)
| +--- (11)
| | +--- (8.5)
| | | +---a
| | | +---b
| | +---e
| +--- (14)
| | +---c
| | +---d
```

```
----- Wikipedia Example with WPGMA ----
(((a[i:0,h:0,s:1],b[i:1,h:0,s:1])[i:-1,h:8.5,s:2],e[i:4,h:0,s:1])[i:-1,h
    ↪ :11,s:3],(c[i:2,h:0,s:1],d[i:3,h:0,s:1])[i:-1,h:14,s:2])[i:-1,h
    ↪ :17.5,s:5];
 (17.5)
| +--- (11)
| | +--- (8.5)
| | | +---a
| | | +---b
| | +---e
| +--- (14)
| | +---c
| | +---d
======= end testing build and print phylogenetic trees =======
```