Practical 3: Ultrametric and Additive Trees

Ximena Moure Eliya Tiram

03/10/2023 , submission deadline 09/10/2023

1. **Given a file ultrametric.txt containing a matrix of evolutionary distances between species, write a Python script to determine whether the distance matrix is ultrametric**
   The matrix is ultrametric. The following code reads the file given in the exercise and answer the questions within it.

```
    # Read matrix from file
with open('ultrametric.txt', 'r') as f:
    matrix = [list(map(int, line.split())) for line in f]


def is_ultrametric(matrix):
    n = len(matrix)

    # Check if diagonal is all zeros
    for i in range(n):
        if matrix[i][i] != 0:
            return False

    # Check if matrix is symmetric
    for i in range(n):
        for j in range(n):
            if matrix[i][j] != matrix[j][i]:
                return False

    # Check ultrametric triangle inequality
    for i in range(n):
        for j in range(n):
            for k in range(n):
                # Determine the maximum of the three distances
                max_val = max(matrix[i][j], matrix[j][k], matrix[i][k])
                # Check if this maximum value
    appears at least twice among the three distances (for any three points,
                # the largest distance between any two of the points should

                # equal the distance between the third point and the other two).
                # If not, the ultrametric triangle inequality is violated

                if [matrix[i][j], matrix[j][k], matrix[i][k]].count(max_val) < 2:
                    return False

    return True


result_is_ultra = is_ultrametric(matrix)
if result_is_ultra:
    print("The matrix is ultrametric.")
else:
    print("The matrix is not ultrametric.")
```

Listing 1: Python code ex1

2. **What is the running time of your script, as a function of the number n of species?**
   Checking Diagonal is Zeros: $O(n)$
   Checking if the matrix is Symmetric: $O(n^2)$
   Checking Ultrametric Triangle Inequality: $O(n^3)$
   So, the overall time complexity is: $O(n) + O(n^2) + O(n^3) = O(n^3)$

3. **What is the best possible running time of an algorithm to test for an ultrametric distance matrix?**
   We have to consider every possible triplet of species, and this results in a time complexity of $O(n^3)$.

4. **Given a file additive.txt containing a matrix of evolutionary distances between species, write a Python script to determine whether the distance matrix is additive.**
   The matrix is additive. The code below shows the implemented algorithm that led to the result.

```python
from itertools import combinations

# Read matrix from file and store it in 'matrix' as rows
with open('additive.txt', 'r') as f:
    matrix = [list(map(int, line.split())) for line in f]

print("matrix", matrix)


# Checks whether the matrix is additive using the Four-Point Condition
def is_additive(matrix):
    for comb in combinations(range(len(matrix)), 4):
        i, j, k, l = comb
        # Check Four-Point Condition for each combination of four points
        sum1 = matrix[i][j] + matrix[k][l]
        sum2 = matrix[i][k] + matrix[j][l]
        sum3 = matrix[i][l] + matrix[j][k]
        # Finding the two largest sums
        sums = [sum1, sum2, sum3]
        max1 = max(sums)
        sums.remove(max1)
        max2 = max(sums)
        # Check if two largest sums are equal
        if max1 != max2:
            return False
    return True


result_is_additive = is_additive(matrix)

if result_is_additive:
    print("The matrix is additive.")
else:
    print("The matrix is not additive.")

```

Listing 2: Python code ex4

5. **What is the running time of your script, as a function of the number n of species?**
   Given a matrix of size n×n, the number of ways to select 4 species out of n is given by the binomial coefficient $\binom{n}{4}$. This can be calculated as:

$$\binom{n}{4} = \frac{n!}{4!(n-4)!}$$

   In big O notation, the number of combinations is $O(n^4)$.
   For each of these combinations, we're doing constant work (i.e., calculating some distances and comparisons which takes O(1) time).
   Thus, the overall time complexity for the algorithm is: $O(n^4)$×O(1)=$O(n^4)$

6. **What is the best possible running time of an algorithm to test for an additive distance matrix?**
   The best known running time for checking if a distance matrix is additive using the Four-Point Condition is $O(n^4)$for a matrix of size n×n. This is because you have to check the Four-Point Condition for every possible combination of four species, which is $O(n^4)$ combinations.