Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

# ÁRVORES FILOGENÉTICAS EM CYTHON

**FILIPE FREIRE ARAUJO**

Msc. André Chastel Lima (Orientador)

Dourados - MS
2024

# ÁRVORES FILOGENÉTICAS EM CYTHON

**FILIPE FREIRE ARAUJO**

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso, devidamente corrigida e defendida por Filipe Freire Araujo e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 15 de Outubro de 2024

Msc. André Chastel Lima
(Orientador)

A689a Araujo, Filipe Freire

Árvores filogenéticas em cython / Filipe Freire Araujo. – Dourados, MS: UEMS, 2024.
90p.

Trabalho de Conclusão de Curso (Graduação) – Ciência da Computação – Universidade Estadual de Mato Grosso do Sul, 2024.
Orientador: Me. André Chastel Lima

1. Árvores filogenéticas (biologia). 2. Cython (linguagem de programação) 3. Biologia computacional. 4. Algoritmos. 5. Evolução (biologia). I. Lima, André Chastel. II. Título

CDD 23 ed. 005.1

Ficha Catalográfica elaborada pela bibliotecária da Universidade Estadual de Mato Grosso do Sul (UEMS)
Aline Perdomo Soutelo, inscrita sob o CRB n. 3668 - 1ª Região.

Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

# ÁRVORES FILOGENÉTICAS EM CYTHON

**FILIPE FREIRE ARAUJO**

**Outubro de 2024**

## Banca Examinadora:

Prof. Msc. André Chastel Lima
Área de Computação - UEMS

Prof.ª Dra. Glaucia Gabriel Sass
Área de Computação - UEMS

Prof.ª Dra. Raquel Marcia Müller
Área de Computação - UEMS

Dedico este trabalho a todos que vieram antes, e a todos que estão por vir.

## AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Marta Freire Araujo e Elton de Araujo Lima, por todo o apoio que me deram para seguir o meu futuro.

Agradeço a minha irmã, Clarinda Freire Araujo por ser um raio de sol nos dias mais escuros.

Agradeço aos meus professores por terem me fornecido o conhecimento e a capacidade de usá-lo para seguir meu futuro na área de computação.

Agradeço aos meus amigos por torcerem pelo meu sucesso.

Agradeço ao meu orientador e professor André Chastel Lima por sua dedicação extraordinária aos seus alunos.

# RESUMO

Este trabalho tem como objetivo analisar a eficiência computacional na geração de árvores filogenéticas utilizando os algoritmos Neighbor-Joining, Neighbour-Joining Flexível e UPGMA, implementados nas linguagens python e Cython. As árvores filogenéticas são fundamentais para o estudo das relações evolutivas entre espécies biológicas, contribuindo para a compreensão da evolução e da biodiversidade. O uso de Cython é explorado como uma alternativa para otimizar a execução desses algoritmos, buscando aumentar o desempenho em comparação com implementações tradicionais em python.

Palavra-chave: Árvores Filogenéticas, Cython, Biologia Computacional

# ABSTRACT

This study aims to analyze computational efficiency in the generation of phylogenetic trees using the Neighbor-Joining, Flexible Neighbor-Joining, and UPGMA algorithms implemented in python and Cython. Phylogenetic trees are fundamental for studying evolutionary relationships between biological species, contributing to the understanding of evolution and biodiversity. Cython is explored as an alternative to optimize the execution of these algorithms, aiming to improve performance compared to traditional python implementations.



Key-word: Phylogenetic trees, Cython, Computational Biology

# SUMÁRIO

# Lista de tabelas

## Lista de figuras

## Lista de abreviaturas

| | |
|---|---|
| NJ | Neighbor-Joining |
| UPGMA | Unweighted Pair Group Method With Arithmetic Mean |
| FNJ | Flexible Neighbor-Joining |

# 1 Introdução

A filogenia é o estudo das relações evolutivas entre diferentes espécies ou grupos de organismos, e as árvores filogenéticas são representações gráficas dessas relações. A construção dessas árvores é fundamental em diversas áreas da biologia e bioinformática, pois permite uma melhor compreensão da evolução e das conexões entre os seres vivos. Um dos métodos amplamente utilizados para construir essas árvores é o algoritmo de Neighbor-Joining, que se destaca por ser um método eficiente para a construção de árvores filogenéticas com base em distâncias.

O algoritmo NJ com complexidade de $O(n^3)$ é particularmente atrativo, além de sua capacidade de lidar com grandes conjuntos de dados sem perder precisão significativa. O algoritmo FNJ é uma variação do NJ que usa uma flexibilidade para ganhar desempenho e diminuir a perda de precisão da árvore gerada. O UPGMA, com uma complexidade de $O(n^2)$ já oferece um método hierárquico de agrupamento que tem funcionamento rápido especialmente quando as taxas de evolução entre as espécies são constantes. No entanto, com o avanço das tecnologias de processamento de dados, novas abordagens vêm sendo desenvolvidas para otimizar ainda mais o tempo de execução desses algoritmos. Nesse contexto, uma das alternativas é o uso de linguagens de programação de baixo nível, como o Cython, que permite a otimização de código Python com a integração de bibliotecas C, aumentando a eficiência do processamento.

## 1.1 Objetivo

### 1.1.1 Objetivo Geral

O objetivo deste trabalho é realizar uma análise comparativa do desempenho de tempo entre os algoritmos Neighbor-Joining, Neighbor-Joining Flexível, e UPGMA em sua implementação padrão e uma versão otimizada em Cython, e também a qualidade da árvore gerada. Através dessa comparação, será possível verificar os ganhos em termos de tempo de execução. Dessa forma, este estudo busca contribuir para a escolha de técnicas de otimização adequadas para o uso em bioinformática, onde a construção rápida e precisa de árvores filogenéticas é essencial.

### 1.1.2 Objetivos Específicos

Os principais objetivos são:

- Implementar o código de NJ em python e em Cython
- Implementar o código de NJ flexível em python e em Cython
- Implementar o código UPGMA em python e em Cython
- Fazer comparações entre o tempo das implementações em python, e as implementações em Cython
- Conferir a qualidade das árvores geradas

## 1.2 Justificativa

A construção de árvores filogenéticas desempenha um papel importante em diversas áreas da biologia, como a taxonomia, genética, evolução, e aplicações na bioinformática. Com o aumento exponencial da quantidade de dados biológicos gerados por tecnologias de sequenciamento de nova geração, se tornou essencial ter métodos rápidos e eficientes para processar esses dados, e gerar representações precisas das relações evolutivas. Nesse sentido, o algoritmo Neighbor-Joining se destaca como um dos métodos mais utilizados devido à sua eficiência e simplicidade. No entanto, mesmo métodos consagrados como o NJ podem se tornar limitantes quando aplicados a conjuntos de dados massivos, especialmente no contexto de grandes estudos filogenéticos.

Diante desse cenário, explorar alternativas que trazem acelerações no processamento é fundamental. O uso de Cython, uma ferramenta que permite a integração de código python com C, oferece uma oportunidade para melhorar o desempenho do algoritmo Neighbor-Joining, mantendo sua precisão, e reduzindo significativamente o tempo de execução. A otimização do tempo de processamento é particularmente relevante quando se trata de dados filogenéticos de grande escala, onde a construção de árvores pode demandar recursos computacionais intensivos.

Assim, este trabalho se justifica pela necessidade de comparar o desempenho entre a versão padrão do algoritmo Neighbor-Joining e uma versão otimizada em Cython, permitindo identificar qual abordagem é mais eficiente para o processamento de grandes volumes de dados. Através dessa comparação, espera-se contribuir para o aprimoramento das ferramentas computacionais utilizadas na bioinformática,

proporcionando soluções mais rápidas e acessíveis para pesquisadores que trabalham com dados evolutivos.

### 1.3 Metodologia

A metodologia implementada no trabalho utiliza inicialmente um estudo bibliográfico para melhor entendimento dos algoritmos e suas aplicações. Após isso, serão desenvolvidos os algoritmos NJ, NJ Flexível, e UPGMA em python e em Cython. Será feito a comparação entre os resultados de tempo dos algoritmos em python e os algoritmos em Cython. Os testes serão realizados com múltiplas entradas, em formato de matrizes simétricas que modelam distâncias entre elementos em um espaço filogenético. As matrizes são de tamanho n por n, onde n corresponde ao número de elementos em análise. Para obter o tempo de execução de cada código será usado a biblioteca time que captura o tempo inicial e final do processamento do código e ao subtrair o tempo inicial do tempo inicial temos o tempo de execução.

### 1.4 Conceitos

A filogenética é o estudo das relações de espécies em biologia que segundo Ziemert e Jensen (2012) dentro dessa disciplina, a filogenética molecular utiliza dados de sequências para inferir essas relações, tanto para organismos quanto para os genes que mantêm. Com a grande quantidade de dados de sequência disponíveis publicamente, a inferência filogenética se tornou cada vez mais importante em todos os campos da biologia.

Uma das responsabilidades da computação biológica em relação a filogenética de acordo com Lima, Araujo, Stefanes, e Rozante (2022) é construir árvores

filogenéticas de um grupo de espécies com precisão mapeando a distância evolucionária entre cada par de espécies mapeadas.

Uma ferramenta útil para esse tipo de trabalho em grande escala é o Cython, que apresenta melhoria na eficácia de execução de códigos. De acordo com o Smith (2015), em seu livro *Cython A Guide for Python Programmers,* Cython demonstra um *speedup* muito bom em problemas como cálculo de números fibonacci em comparação com python puro obtendo melhoramento de tempo de até 50 vezes no cálculo do valor do número fibonacci na posição 90.

Usar essa ferramenta fica interessante ao ver o potencial de uso do algoritmo NJ, que de acordo com o Gascuel e o Steel (2006) forma iterativamente os pares de espécies para a formação de uma árvore filogenética de acordo com pares de espécies. A palavra chave é iterativamente, que contribui para o entendimento de que essas operações na prática são feitas com uma quantidade de dados enorme.

Esse desenvolvimento na área de computação biológica teve duas figuras importantes, Saitou e Nei, que em seu artigo em 1987 estabeleceu o algoritmo NJ que produz uma árvore de evolução mínima, uma solução eficaz em obter a topologia de árvore correta especialmente em comparação com os outros métodos populares da época.

# 2 Cython

De acordo com Smith (2015) Cython é uma linguagem híbrida posicionada entre python e C. Essa linguagem combina o melhor dos dois com a simplicidade do python e o desempenho otimizado de C.

Um dos principais benefícios do Cython é a velocidade. Ao utilizar Cython, é possível acelerar a execução de código Python. Isso ocorre porque ele converte o código interpretado do Python em código compilado, o que resulta em uma execução mais eficiente. Além disso, Cython oferece a flexibilidade de otimizar apenas partes do código que são críticas em termos de desempenho, sem a necessidade de reescrever toda a aplicação em C.

Cython traz a facilidade de desenvolvimento com python e a velocidade de execução do C.

## 2.1 Cythonização de Código Python

Para converter um código Python em Cython, o processo é relativamente simples:

1. **Instalação do Cython**: Primeiramente, o Cython deve ser instalado com o comando: pip install cython.
2. **Modificação do Código**: O próximo passo é modificar o código Python, alterando a extensão do arquivo de .py para .pyx. A tipagem estática pode ser adicionada para melhorar ainda mais o desempenho, utilizando a palavra-chave “cdef” para declarar variáveis com tipos específicos.

3. **Compilação**: Após as modificações, o código precisa ser compilado. Isso pode ser feito usando um arquivo setup.py, que permite a compilação via comando: python setup.py build_ext --inplace.
4. **Uso do Código Otimizado**: Após a compilação, o código Cython gerado pode ser importado e usado como se fosse um módulo python normal usando palavras chaves from e import.

Com isso o código python pode ser “cynthonizado” e essa otimização é útil em softwares que demandam alto desempenho, como bioinformática, e processamento de dados científicos.

# 3 Neighbor-Joining

O algoritmo Neighbor-Joining (NJ) é uma técnica amplamente utilizada para a construção de árvores filogenéticas com base em distâncias evolutivas entre diferentes espécies. Esse método foi desenvolvido por Saitou e Nei em 1987 e se destaca por sua eficiência computacional e simplicidade, especialmente ao lidar com grandes conjuntos de dados. O NJ pertence à classe dos métodos de agrupamento, onde o objetivo é encontrar a árvore não enraizada que melhor representa as distâncias entre as espécies, de acordo com uma matriz de distâncias.

O algoritmo Neighbor-Joining é frequentemente utilizado em bioinformática para inferir relações filogenéticas, com aplicações que vão desde a análise da evolução de genes até a comparação de genomas completos. Uma das principais vantagens do NJ é que ele é um método aglomerativo, que significa que ele constrói a árvore filogenética gradualmente, unindo pares de taxas que minimizam a distância total entre os nós da árvore.

## 3.1 Funcionamento do Algoritmo Neighbor-Joining

O funcionamento do NJ é um processo iterativo. A cada iteração, ele seleciona os dois nós (ou espécies) que são mais "próximos" com base na matriz de distâncias e traz juntos para formar um novo nó. A matriz de distâncias é atualizada, e o processo se repete até que todos os nós tenham sido unidos. O objetivo é minimizar a soma total das distâncias em toda a árvore.

O algoritmo pode ser descrito em termos dos seguintes passos:

1. Calcular a matriz de distâncias entre todas as espécies ou taxas.
2. A partir dessa matriz, calcular uma matriz auxiliar que ajusta as distâncias, levando em consideração a soma das distâncias para todos os outros nós.
3. Selecionar os dois nós com o menor valor ajustado na matriz auxiliar e unir eles em um nó intermediário.
4. Atualizar a matriz de distâncias, substituindo os nós unidos pelo nó intermediário e calculando as distâncias novas.
5. Repetir o processo até que todos os nós tenham sido unidos em uma única árvore.

### 3.2 Algoritmo Neighbor-Joining

```
1. Inicializar uma matriz de distâncias D entre as n espécies.
2. Enquanto n > 2:
   3. Para cada par de espécies (i, j), calcular a matriz auxiliar Q:
      Q(i,j) = (n-2) * D(i,j) - Σ D(i, k) - Σ D(j, k)
   4. Selecionar o par de espécies (i,j) com o menor valor de Q.
   5. Unir i e j em um novo nó U.
   6. Calcular as distâncias de U para todas as outras espécies:
      D(U,k) = (D(i,k) + D(j,k) - D(i,j)) / 2
   7. Atualizar a matriz de distâncias D para refletir o novo nó U.
   8. Remover i e j da matriz D.
9. Quando restarem apenas duas espécies, unir as últimas duas e
formar a árvore final.
```

### 3.3 Exemplo

O Wikipedia oferece um exemplo de como o NJ utiliza uma matriz de distâncias entre pares de espécies e, a cada iteração, une as duas espécies mais próximas,

criando um novo nó que representa essa união. Depois, a matriz é atualizada para incluir esse nó, repetindo o processo até que todas as espécies sejam conectadas, formando a árvore filogenética.

1. Temos uma matriz simétrica D 4x4 com espécies a, b, c, e d:

| | **a** | **b** | **c** | **d** |
|---|---|---|---|---|
| **a** | 0 | 5 | 9 | 9 |
| **b** | 5 | 0 | 10 | 10 |
| **c** | 9 | 10 | 0 | 8 |
| **d** | 9 | 10 | 8 | 0 |

D[i, j] é a distância entre as espécies i e j, logo D[a, b]=D[0, 1] e para D[0, 1]=5

2. Para inicializar temos o calculo da divergencia total, onde calculamos a soma da distancia de cada especie para todas as outras especies, por exemplo:

   Para espécie a, temos distância 5 para espécie b, 9 para c, e 9 para d, logo temos uma divergência para espécie a no total de 23.

   Logo temos:

$$divergência\_total[a] = 5 + 9 + 9 = 23$$

$$divergência\_total[b] = 5 + 10 + 10 = 25$$

$$divergência\_total[c] = 9 + 10 + 8 = 27$$

$$divergência\_total[d] = 9 + 10 + 8 = 27$$

$$divergência\_total = [23, 25, 27, 27]$$

3. Para calcular matriz Q usamos a formula:

$$Q[i,j] = (n-2)\cdot D[i,j] - divergência\_total[i] - divergência\_total[j]$$

$$Q[a,\ b] = (4-2) * D[a,\ b] - 23 - 25 = 2 * 5 - 23 - 25 = -38$$

$$Q[a,\ c] = (4-2) * D[a,\ c] - 23 - 27 = 2 * 9 - 23 - 27 = -32$$

$$Q[a,\ d] = (4-2) * D[a,\ d] - 23 - 27 = 2 * 9 - 23 - 27 = -32$$

$$Q[b,\ c] = (4-2) * D[b,\ c] - 25 - 27 = 2 * 10 - 25 - 27 = -42$$

$$Q[b,\ d] = (4-2) * D[b,\ d] - 25 - 27 = 2 * 10 - 25 - 27 = -42$$

$$Q[c,\ d] = (4-2) * D[c,\ d] - 27 - 27 = 2 * 8 - 27 - 27 = -38$$

Onde n é o número de espécies e D[i, j] é a distância entre as espécies i e j. Fazendo esse cálculo para a matriz toda temos a matriz Q:

| | **a** | **b** | **c** | **d** |
|---|---|---|---|---|
| **a** | 0 | -38 | -32 | -32 |
| **b** | -38 | 0 | -42 | -42 |
| **c** | -32 | -42 | 0 | -38 |
| **d** | -32 | -42 | -38 | 0 |

4. Seleção de par:

   Selecionamos o menor valor de Q[i, j] que é Q[1, 2] =−42, então o par (b,c) será fundido.

5. Fusão do par selecionado:

   Depois de selecionar o par (c, d) calculamos os comprimentos dos ramos usando as seguintes fórmulas:

$$delta = (divergência\_total[c] - divergência\_total[d]) / (n - 2)$$

$$delta = (25 - 27) / (4 - 2) = -1$$

O comprimento do ramo é calculado para c e para d:

$$ramo\_b = 0.5 * (D[b, c] + delta)$$
$$ramo\_b = 0.5 * (10 - 1) = 4.5$$

$$ramo\_c = D[b, c] - ramo\_b$$
$$ramo\_c = 10 - 4.5 = 5.5$$

E agora as espécies b e c se fundem e se tornam uma espécie única bc

6. A matriz é atualizada e as novas distâncias entre o novo nó cd e as outras espécies são calculadas usando:

$$D[bc, a] = (D[b, a] + D[c, a]) / 2$$

$$(5 + 9) / 2 = 7$$

$$D[bc, d] = (D[b, d] + D[c, d]) / 2$$

$$(10 + 8) / 2 = 9$$

Isso resulta na nova matriz:

| | **a** | **bc** | **d** |
|---|---|---|---|
| **a** | 0 | 7 | 9 |
| **bc** | 7 | 0 | 9 |
| **d** | 9 | 9 | 0 |

7. Calculamos a nova matriz Q usando:

$$Q[i, j] = (n - 2) \cdot D[i, j] - divergência\_total[i] - divergência\_total[j]$$
$$divergência\_total[a] = 7 + 9 = 16$$

$$divergência\_total[c] = 7 + 9 = 16$$

$$divergência\_total[d] = 9 + 9 = 18$$

Onde n=3 para as espécies a, bc, d. A divergência total soma as distâncias de cada espécie para as outras espécies restantes.

Calculamos a matriz Q:

$$Q[a, bc] = (3 - 2) * D[a, bc] - 16 - 16 = 1 * 7 - 16 - 16 = -25$$
$$Q[a, d] = (3 - 2) * D[a, d] - 16 - 18 = 1 * 9 - 16 - 18 = -25$$
$$Q[bc, d] = (3 - 2) * D[bc, d] - 16 - 18 = 1 * 9 - 16 - 18 = -25$$

| | **a** | **bc** | **d** |
|---|---|---|---|
| **a** | 0 | -25 | -25 |
| **bc** | -25 | 0 | -25 |
| **d** | -25 | -25 | 0 |

8. Fusão final:

Os valores da matriz Q são todos iguais, logo escolhemos o par a e bc. Fazemos os cálculos de cumprimento de ramo.

O comprimento do ramo e calculado para c e para d:

$$ramo\_a = 0.5 * (D[a, bc] + delta)$$

$$ramo\_a = 0.5 * 7 = 3.5$$

$$ramo\_bc = D[c, d] - ramo\_a$$

$$ramo\_bc = 7 - 3.5 = 3.5$$

E agora as espécies a e bc se fundem e se tornam uma espécie única abc.

9. A matriz é atualizada e as novas distâncias entre o novo no cd e as outras espécies são calculadas usando:

$$D[abc,\ d]\ =\ (D[a,\ d]\ +\ D[bc,\ d])\ /\ 2$$
$$(9\ +\ 9)/2\ =\ 9$$

Isso resulta na nova matriz:

| | **abc** | **d** |
|---|---|---|
| **abc** | 0 | 9.5 |
| **d** | 9.5 | 0 |

10. Fusão final:

Agora, fundimos os dois nós restantes ab e cd, e calculamos o comprimento do ramo:

$$ramo\_abc\ =\ 0.5{\cdot}D[abc, d]\ =\ 0.5{\cdot}9\ =\ 4.5$$

$$ramo\_d\ =\ 0.5{\cdot}D[abc, d]\ -\ ramo\_abc\ =\ 9\ -\ 4.5\ =\ 4.5$$

- O comprimento do ramo de abc até a raiz é 4.5.
- O comprimento do ramo de d até a raiz é 4.5.

11. Resultado final:

Agora temos a árvore completa:

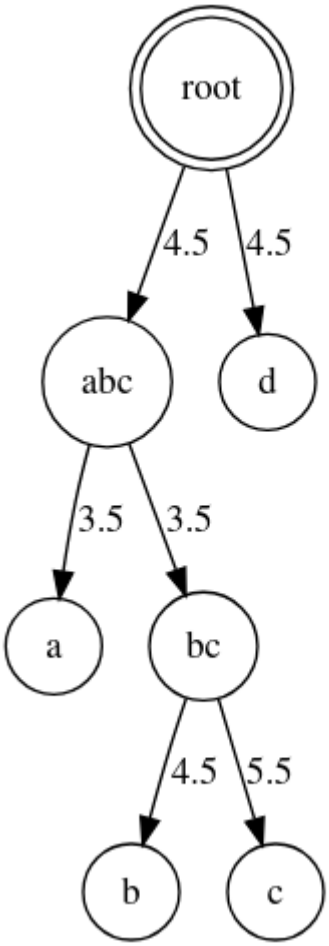



Figura 1: árvore gerada para exemplo de NJ

# 4 Neighbor-Joining Flexível

A versão padrão do NJ tem uma complexidade cúbica no pior caso, $O(n^3)$, o que pode ser inviável para grandes volumes de dados.

No entanto, para lidar com grandes conjuntos de dados, foi proposto o Neighbor-Joining flexível. O FNJ introduz uma modificação importante ao permitir que, em cada iteração, um conjunto de múltiplos pares de nós seja processado simultaneamente, em vez de apenas um par como no NJ original. Esse processo acelera a construção da árvore, resultando em uma complexidade $O(n^2 \log n)$, uma melhoria significativa em termos de eficiência em comparação ao NJ.

Além da eficiência, a flexibilidade do FNJ permite ajustar a precisão e a velocidade do algoritmo através de parâmetros controláveis (p/q). À medida que o fator de achatamento (flattening factor) p/q aumenta, o tempo de execução é reduzido, mas a precisão da árvore gerada pode diminuir como resultado. A escolha adequada desse parâmetro é crucial para encontrar um equilíbrio entre desempenho e corretude.

## 4.1 Algoritmo Neighbor-Joining Flexível

1. Inicializar a matriz de distâncias D e os parâmetros p e q.
2. Enquanto N > 2:
   3. Calcular a matriz auxiliar Q.
   4. Selecionar ⌈N·p/q⌉ pares disjuntos que minimizam Q.
   5. Unir esses pares e atualizar a matriz D.
   6. Reduzir o número de elementos N em cada iteração.
7. Retornar a árvore filogenética gerada.

## 4.2 Exemplo

O FNJ é uma variante flexível do algoritmo NJ. Ele utiliza dois parâmetros, p e q, para ajustar o processo de união de espécies com maior flexibilidade, permitindo ajustar a precisão e a velocidade do algoritmo de acordo com as necessidades:

1. Temos uma matriz simétrica D 4x4 com espécies a, b, c, e d:

| | **a** | **b** | **c** | **d** |
|---|---|---|---|---|
| **a** | 0 | 5 | 9 | 9 |
| **b** | 5 | 0 | 10 | 10 |
| **c** | 9 | 10 | 0 | 8 |
| **d** | 9 | 10 | 8 | 0 |

D[i, j] é a distância entre as espécies i e j, logo para D[0, 1]=5

2. Para inicializar temos o calculo da divergencia total, onde calculamos a soma da distancia de cada especie para todas as outras especies, por exemplo:

   Para espécie a, temos distância 5 para espécie b, 9 para c, e 9 para d, logo temos uma divergência para espécie a no total de 23.

   Logo temos:

$$divergência\_total[a] = 5 + 9 + 9 = 23$$

$$divergência\_total[b] = 5 + 10 + 10 = 25$$

$$divergência\_total[c] = 9 + 10 + 8 = 27$$

$$divergência\_total[d] = 9 + 10 + 8 = 27$$

$$divergência\_total = [23, 25, 27, 27]$$

3. Para calcular matriz Q usamos a formula:

$$Q[i,j] = (n-2)\cdot D[i,j] - divergência\_total[i] - divergência\_total[j]$$

$$Q[a,\ b] = (4-2) * D[a,\ b] - 23 - 25 = 2 * 5 - 23 - 25 = -38$$

$$Q[a,\ c] = (4-2) * D[a,\ c] - 23 - 27 = 2 * 9 - 23 - 27 = -32$$

$$Q[a,\ d] = (4-2) * D[a,\ d] - 23 - 27 = 2 * 9 - 23 - 27 = -32$$

$$Q[b,\ c] = (4-2) * D[b,\ c] - 25 - 27 = 2 * 10 - 25 - 27 = -42$$

$$Q[b,\ d] = (4-2) * D[b,\ d] - 25 - 27 = 2 * 10 - 25 - 27 = -42$$

$$Q[c,\ d] = (4-2) * D[c,\ d] - 27 - 27 = 2 * 8 - 27 - 27 = -38$$

Onde n é o número de espécies e D[i, j] é a distância entre as espécies i e j. Fazendo esse cálculo para a matriz toda temos a matriz Q:

| | **a** | **b** | **c** | **d** |
|---|---|---|---|---|
| **a** | 0 | -38 | -32 | -32 |
| **b** | -38 | 0 | -42 | -42 |
| **c** | -32 | -42 | 0 | -38 |
| **d** | -32 | -42 | -38 | 0 |

4. Parametros p e q:

Para esse exemplo atribuímos p=1 e q=20, e com isso o número de pares selecionados para fusão é dado por:

$$n * (p/q)$$

$$4 * \lceil 1/20 \rceil = 1$$

Então, selecionamos um par por iteração. O menor valor de Q[i, j] é Q[2,3]=−42, então o par (b,c) será fundido.

5. Fusão do par selecionado:

   Depois de selecionar o par (c, d) calculamos os comprimentos dos ramos usando as seguintes fórmulas:

$$delta = (divergência\_total[b] - divergência\_total[c]) / (n - 2)$$

$$delta = (25 - 27) / 2 = -1$$

O comprimento do ramo é calculado para c e para d:

$$ramo\_b = 0.5 * (D[b, c] + delta)$$
$$ramo\_b = 0.5 * (10 - 1) = 4.5$$

$$ramo\_c = D[b, c] - ramo\_b$$
$$ramo\_c = 10 - 4.5 = 5.5$$

E agora as espécies b e c se fundem e se tornam uma espécie única bc

6. A matriz é atualizada e as novas distâncias entre o novo no cd e as outras espécies são calculadas usando:

$$D[bc, a] = (D[b, a] + D[c, a]) / 2$$

$$D[bc, a] = (5 + 9) / 2 = 7$$

$$D[bc, d] = (D[b, d] + D[c, d]) / 2$$

$$D[bc, a] = (10 + 8) / 2 = 9$$

   Isso resulta na nova matriz:

| | **a** | **b** | **cd** |
|---|---|---|---|
| **a** | 0 | 7 | 9 |
| **b** | 7 | 0 | 9 |
| **cd** | 9 | 9 | 0 |

7. Calculamos a nova matriz Q usando:

$$Q[i,j] = (n-2)\cdot D[i,j] - divergência\_total[i] - divergência\_total[j]$$

Onde n=3 para as espécies a, b, cd. A divergência total soma as distâncias de cada espécie para as outras espécies restantes.

$$divergência\_total[a] = 7 + 9 = 16$$
$$divergência\_total[bc] = 7 + 9 = 16$$
$$divergência\_total[d] = 9 + 9 = 18$$

Calculamos a matriz Q:

$$Q[a,bc] = (3-2)\cdot D[a,bc] - 16 - 16 = 1 * 7 - 16 - 16 = -25$$
$$Q[a,d] = (3-2)\cdot D[a,d] - 16 - 18 = 1 * 9 - 16 - 18 = -25$$
$$Q[bc,d] = (3-2)\cdot D[bc,d] - 16 - 18 = 1 * 7 - 16 - 18 = -25$$

| | **a** | **b** | **cd** |
|---|---|---|---|
| **a** | 0 | -25 | -25 |
| **b** | -25 | 0 | -25 |
| **cd** | -25 | -25 | 0 |

8. Seleção de pares:

Com n=3, p=1, e q=20 temos $3 * [1/20] = 1$. Logo selecionamos o menor valor de Q, Q=-25. Vamos selecionar pares a e b para fusão.

Fazemos os cálculos de cumprimento de delta e cumprimento de ramo:

$$delta = (divergência\_total[a] - divergência\_total[b]) / (n - 2)$$

$$delta = (16 - 16) / 1 = 0$$

O comprimento do ramo e calculado para c e para d:

$$ramo\_a = 0.5 * (D[a, bc] + delta)$$

$$ramo\_a = 0.5 * (7 + 0) = 3.5$$

$$ramo\_bc = D[a, bc] - ramo\_a$$

$$ramo\_bc = 7 - 3.5 = 3.5$$

Agora fundimos a e bc para criar nó abc com tamanho de ramo de 3.5 para a e 3.5 para bc.

9. A matriz é atualizada e as novas distâncias entre o novo no cd e as outras espécies são calculadas usando:

$$D[abc, d] = (D[b, d] + D[b, d] + D[c, d]) / 3$$
$$(9 + 10 + 8)/3 = 9$$

Isso resulta na nova matriz:

| | **abc** | **d** |
|---|---|---|
| **abc** | 0 | 9 |
| **d** | 9 | 0 |

10. Fusão final:

Agora, fundimos os dois nós restantes abc e d, e calculamos o comprimento do ramo:

- O comprimento do ramo de abc até a raiz é 4.5.
- O comprimento do ramo de d até a raiz é 4.5.

11. Resultado final:

Agora temos a árvore completa:

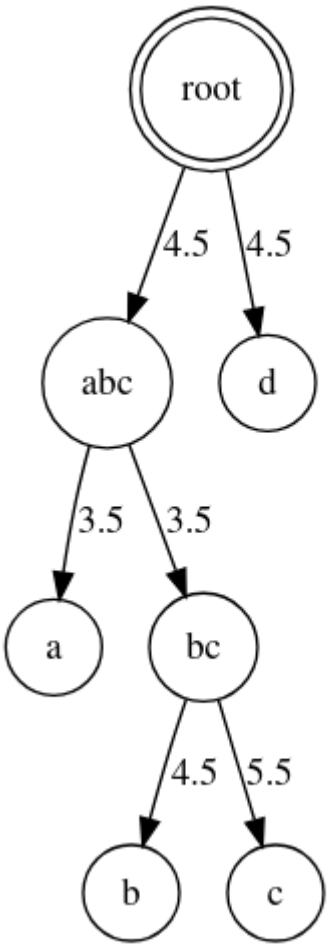



Figura 2: árvore gerada para exemplo de FNJ

# 5 UPGMA

O método UPGMA (Unweighted Pair Group Method with Arithmetic Mean) é um algoritmo de agrupamento hierárquico utilizado na construção de árvores filogenéticas. Desenvolvido por Sneath e Sokal em 1973, ele constrói árvores filogenéticas enraizadas com base em distâncias evolutivas. A principal premissa do UPGMA é que a taxa de evolução é constante ao longo de todas as ramificações da árvore, o que implica que o algoritmo assume uma velocidade de evolução homogênea entre os organismos.

O UPGMA é utilizado para encontrar pares de taxa (espécies ou grupos de espécies) com a menor distância e, em seguida, combinar eles em um único nó, representando seu ancestral comum. Este processo é repetido até que todas as espécies tenham sido agrupadas em uma única árvore enraizada.

A vantagem do UPGMA está em sua simplicidade e eficiência, com uma complexidade temporal de $O(n^2)$, onde n é o número de espécies. No entanto, uma limitação do método é sua suposição de que as taxas de evolução são constantes entre todas as linhagens, o que nem sempre é verdadeiro em cenários biológicos complexos. Como resultado, o UPGMA pode produzir árvores menos precisas quando as espécies em questão evoluíram em velocidades diferentes.

## 5.1 Algoritmo UPGMA

1. Inicializar a matriz de distâncias D entre as espécies.
2. Inicializar um cluster para cada espécie.
3. Enquanto houver mais de um cluster:
   4. Encontrar os dois clusters i e j com a menor distância D(i, j).
   5. Unir i e j em um novo cluster k, calculando a distância média entre k e os outros clusters.
   6. Atualizar a matriz de distâncias D para refletir a união de i e j.
   7. Remover i e j da matriz de distâncias.
8. Repetir até que apenas um cluster permaneça.

## 5.2 Exemplo

O UPGMA utiliza uma matriz de distâncias entre pares de espécies para construir uma árvore filogenética que envolve o agrupamento de espécies com base nas suas distâncias médias. O método assume que a taxa de evolução é constante , isso de acordo com Michener (1957) é chamado de relógio molecular, o que significa que as distâncias observadas são proporcionais ao tempo desde a divergência.

Considere cinco espécies: a, b, c, d, e com suas respectivas distâncias. A matriz de distâncias D é apresentada abaixo:

1. Temos uma matriz simétrica D 4x4 com espécies a, b, c, e d:

| | **a** | **b** | **c** | **d** |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| **a** | 0 | 5 | 9 | 9 |
| **b** | 5 | 0 | 10 | 10 |
| **c** | 9 | 10 | 0 | 8 |
| **d** | 9 | 10 | 8 | 0 |

D[i, j] é a distância entre as espécies i e j, logo para D[0, 1]=5

2. Selecionamos as espécies mais próximas:

No UPGMA, selecionamos o par de espécies (i, j) com a menor distância em D[i, j]. Neste caso, a menor distância é D[a, b]=5. Então, vamos fundir as espécies a e b em um novo nó ab.

3. Cálculo dos Ramos

No UPGMA, a distância do nó ab para as espécies a e b é simplesmente metade da distância entre elas:

$$ramo\_a \ = \ ramo\_b \ = \ D[a, \ b] \ / \ 2 \ = \ 5 \ / \ 2 \ = \ 2.5$$

Agora temos o nó ab com os ramos a e b com cumprimento de ramo de 2.5 para cada ramo.

Agora que fundimos a e b em ab, recalculamos a matriz de distâncias para as três espécies restantes ab, c, d. A nova distância entre ab e as outras espécies é a média aritmética das distâncias de a e b para as outras espécies:

$$D[ab, \ c] \ = \frac{D[a, \ c]+D[b, \ c]}{2} = \frac{9+10}{2} = \ 9.5$$

$$D[ab,\ d] = \frac{D[a,\ d]+D[b,\ d]}{2} = \frac{9+10}{2} = 9.5$$

4. Atualizamos a matriz

| | **ab** | **c** | **d** |
|---|---|---|---|
| **ab** | 0 | 9.5 | 9.5 |
| **c** | 9.5 | 0 | 8 |
| **d** | 9.5 | 8 | 0 |

5. Seleção do próximo par

Agora, selecionamos o próximo par com a menor distância, que é D[c, d]=8. Vamos fundir as espécies c e d em um novo nó cd.

6. Cálculo dos ramos

$$ramo\_c\ =\ ramo\_d\ =\ D[c,\ d]\ /\ 2$$

$$8\ /\ 2\ =\ 4$$

Agora temos no cd com ramos para c e d, com distância 4 para ambos que é metade da distância entre c e d.

7. Atualização da matriz

Agora, restam os nós ab e cd. Calculamos a nova distância entre ab e cd como a média das distâncias de ab para c e d:

$$D[ab,\ cd] = \frac{D[ab,\ c]+D[ab,\ d]}{2} = \frac{9+10}{2} = 9.5$$

Atualizamos a matriz:

| | **ab** | **cd** |
|---|---|---|
| **ab** | 0 | 9.5 |
| **c** | 9.5 | 0 |

8. Fusao final

Agora, fundimos os nós ab e cd. O ramo de ab e cd até a raiz da árvore será:

$$ramo\_ab = ramo\_cd = D[ab, cd] / 2$$

$$ramo\_ab = ramo\_cd = 9.5/2 = 4.75$$

9. Resultado final:

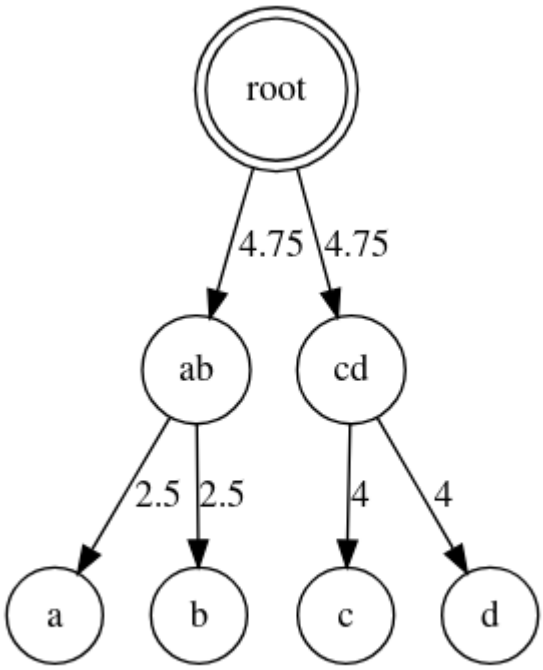



Figura 3: árvore gerada para exemplo de FNJ

# 6 Testes

Os testes foram feitos utilizando matrizes simétricas de diferentes tamanhos como entrada, com o objetivo de medir o tempo de execução e comparar esse tempo junto com a qualidade de formação da árvore gerada dos códigos NJ, FNJ, e UPGMA em python e em Cython.

## 6.1 Configuração dos Testes

Os testes foram realizados em uma máquina linux com arquitetura x86_64 e um processador Intel Core i5-8265U. Para a realização dos testes, foram geradas matrizes simétricas de tamanhos variados, que serviram como entrada para os códigos. As matrizes simétricas representam distâncias evolutivas entre diferentes espécies.

Os tamanhos das matrizes foram definidos em 1024x1024, 2048x2048 e 4096x4096. Para cada um desses tamanhos, foram geradas três matrizes distintas, para garantir uma variação nos dados de entrada. As matrizes simétricas foram geradas com valores aleatórios flutuantes representando as distâncias entre os pares de espécies.

O tempo de execução foi medido pela função time.time().

## 6.2 Procedimento dos Testes

A sequência dos testes foram executado dessa forma:

1. Geração de uma matriz simétrica de tamanho pré-definido (1024, 2048, ou 4096).
2. Execução do código de construção de árvore filogenética usando a matriz como entrada.
3. Registrar medida de tempo de execução.
4. Repetir para as outras duas matrizes do mesmo tamanho.

Este procedimento foi repetido tanto para a implementação em python quanto em Cython, para os três tamanhos de matrizes.

### 6.3 Valores dos Parâmetros p e q

O FNJ introduz dois parâmetros ajustáveis, p e q, para melhorar a precisão e desempenho. Esses parâmetros influenciam tanto a qualidade do resultado quanto a velocidade de execução do algoritmo. No artigo *"A Flexible Variant of the Neighbor-Joining for Building Phylogenetic Trees"* escrito por Lima, Araujo, e Stefanes (2022), o valor p/q foi denominado “fator de achatamento” e esse fator tem efeito diretamente tanto na velocidade quanto a qualidade da árvore gerada.

Um fator de achatamento alto, aumenta a velocidade de execução, porém a qualidade da árvore gerada pode ser menor. Um fator de achatamento baixo traz uma maior precisão mas a um tempo de execução maior. Os valores p e q então oferecem a possibilidade de equilíbrio.

O tamanho da matriz de distâncias é um fator na escolha dos parâmetros p e q, para matrizes maiores pode-se evitar tempos mais prolongados de execução enquanto mantém um grau de precisão sem sacrificar totalmente para o ganho de tempo.

A implementação de FNJ do artigo de Lima, Araujo, Stefanes, e Rozante (2022) explica que o valor de p controla a extensão da busca por pares de nós para serem unidos, logo um p menor resulta em uma busca mais detalhada e precisa, mas mais lenta. Já o q determina a influência da estrutura já formada da árvore nas junções, com um q maior dando mais peso à estrutura existente, enquanto um q menor foca mais nas distâncias imediatas entre os pares.

Para a implementação do FNJ, os parâmetros p e q desempenham papéis cruciais no controle da flexibilidade e eficiência do algoritmo.

- p controla a quantidade de pares candidatos a serem considerados para junção em cada iteração. Um valor maior de p faz com que mais pares sejam inicialmente avaliados, o que pode acelerar a execução do algoritmo ao aumentar a quantidade de opções disponíveis para fusão em cada etapa.
- q, por outro lado, limita o número de pares que serão analisados em profundidade. Um valor menor de q torna o algoritmo mais rápido, pois restringe o número de pares que passam por uma avaliação detalhada. Assim, apenas uma fração dos pares candidatos identificados pelo p é realmente considerada para fusão.

Percebe-se com essas partes definidas que o fator de achatamento p/q controla então a porcentagem de pares a serem selecionados a cada iteração.

Com isso em mente os parâmetros p e q selecionados para os testes foram:

- Matriz 1024: p=10, q=100, atigindo valor p/q=0.1
- Matriz 2048: p=10, q=200, atigindo valor p/q=0.05
- Matriz 4096: p=10, q=400, atigindo valor p/q=0.025

O valor p controla a seleção inicial, enquanto o valor q limita a análise.

### 6.4 Verificação de qualidade

Para verificar a qualidade da árvore gerada por os códigos utilizamos o formato de Newick e a métrica Robinsons-Foulds. O formato de Newick de acordo com Khalafvand (2015) é utilizado na representação de árvores filogenéticas, que descrevem as relações evolutivas entre espécies. Em uma árvore Newick, os pares entre parênteses representam as espécies, enquanto as separações feitas por vírgulas representam eventos de divergência evolutiva.

Como exemplo: (A,(B,(C,D))); representa uma árvore com quatro espécies A, B, C, e D. Onde B, C, e D são mais próximos em termos de evolução.

De acordo com Robinson (1981) a métrica de Robinson-Foulds é uma forma popular de comparar árvores filogenéticas. Ela calcula a distância entre duas árvores ao medir as diferenças nas divisões dos nós da árvore. Essa distância é representada como uma fração x/y onde x é o número de divisões que são diferentes entre as árvores e y é o número de divisões étnicas entre as árvores. Isso significa que, por exemplo, uma distância de RF de 0/14 significa que as árvores são idênticas e um valor de 14/14 significa que as árvores são totalmente diferentes.

# 7 Resultados e Discussão

Ao executar os algoritmos e suas respectivas versões em Cython encontramos uma diferença em desempenho. A versão em Cython, assim como o estudo referencial teórico apontou é mais rápida.

**Resultados em python:**

| | Matriz 1 Tamanho 1024 (segundos) | Matriz 2 Tamanho 1024 (segundos) | Matriz 3 Tamanho 1024 (segundos) | Média Tamanho 1024 (segundos) |
|---|---|---|---|---|
| Neighbor-Joining | 321.7044 | 329.6095 | 334.5948 | 328.63623 |
| Neighbor-Joining Flexivel (10/100) | 11.9186 | 11.0263 | 10.6779 | 11.2076 |
| UPGMA | 109.0346 | 106.4726 | 107.9301 | 107.8124 |

Tabela 1: resultados de tempo de execução para matrizes de tamanho 1024 em python

| | Matriz 1 Tamanho 2048 (segundos) | Matriz 2 Tamanho 2048 (segundos) | Matriz 3 Tamanho 2048 (segundos) | Média Tamanho 2048 (segundos) |
|---|---|---|---|---|
| Neighbor-Joining | 2373.7686 | 2372.8345 | 2382.4895 | 2376.3642 |
| Neighbor-Joining Flexivel (10/200) | 71.7922 | 71.7788 | 73.1727 | 72.2479 |
| UPGMA | 828.5729 | 831.6155 | 839.8741 | 833.3542 |

Tabela 2: resultados de tempo de execução para matrizes de tamanho 2048 em python

| | Matriz 1 Tamanho 4096 (segundos) | Matriz 2 Tamanho 4096 (segundos) | Matriz 3 Tamanho 4096 (segundos) | Média Tamanho 4096 (segundos) |
|---|---|---|---|---|
| Neighbor-Joining | 17,003.4270 | 16,992.9844 | 17,011.3462 | 17,002.5859 |
| Neighbor-Joining Flexivel (10/400) | 568.9844 | 536.8383 | 499.1776 | 535.0001 |
| UPGMA | 6,376.9611 | 6,402.0386 | 6,390.4435 | 6,389.8144 |

Tabela 3: resultados de tempo de execução para matrizes de tamanho 4096 em python

## Resultados em Cython:

| | Matriz 1 Tamanho 1024 (segundos) | Matriz 2 Tamanho 1024 (segundos) | Matriz 3 Tamanho 1024 (segundos) | Média Tamanho 1024 (segundos) |
|---|---|---|---|---|
| Neighbor-Joining | 83.7238 | 85.7153 | 86.2100 | 85.2164 |
| Neighbor-Joining Flexivel (10/100) | 3.1751 | 3.8925 | 3.5090 | 3.5255 |
| UPGMA | 5.0231 | 4.8207 | 4.9080 | 4.9173 |

Tabela 4: resultados de tempo de execução para matrizes de tamanho 1024 em Cython

| | Matriz 1 Tamanho 2048 (segundos) | Matriz 2 Tamanho 2048 (segundos) | Matriz 3 Tamanho 2048 (segundos) | Média Tamanho 2048 (segundos) |
|---|---|---|---|---|
| Neighbor-Joining | 722.4881 | 729.3451 | 724.0042 | 725.2791 |
| Neighbor-Joining Flexivel (10/200) | 21.9379 | 22.6856 | 22.0037 | 22.2091 |
| UPGMA | 29.8637 | 27.2100 | 29.0129 | 28.6955 |

Tabela 5: resultados de tempo de execução para matrizes de tamanho 1024 em Cython

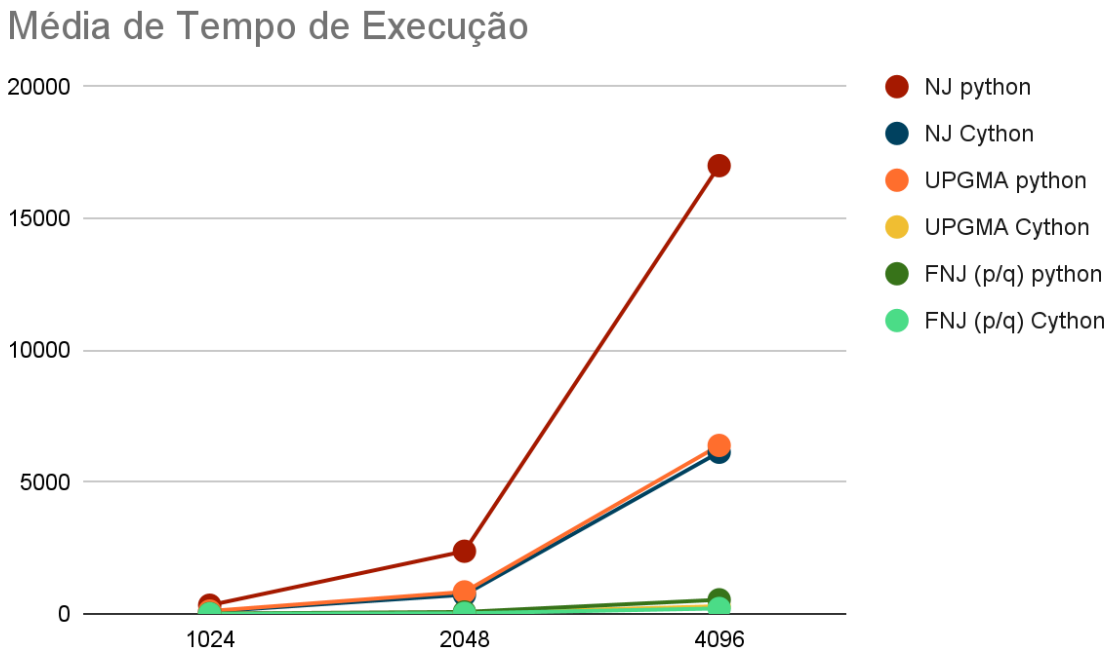
| | Matriz 1 Tamanho 4096 (segundos) | Matriz 2 Tamanho 4096 (segundos) | Matriz 3 Tamanho 4096 (segundos) | Média Tamanho 4096 (segundos) |
|---|---|---|---|---|
| Neighbor-Joining | 6,132.3610 | 6,148.9063 | 6,111.5325 | 6,130.9333 |
| Neighbor-Joining Flexivel (10/400) | 205.1842 | 204.9913 | 205.0572 | 205.0776 |
| UPGMA | 283.1899 | 299.4281 | 292.0305 | 291.5495 |

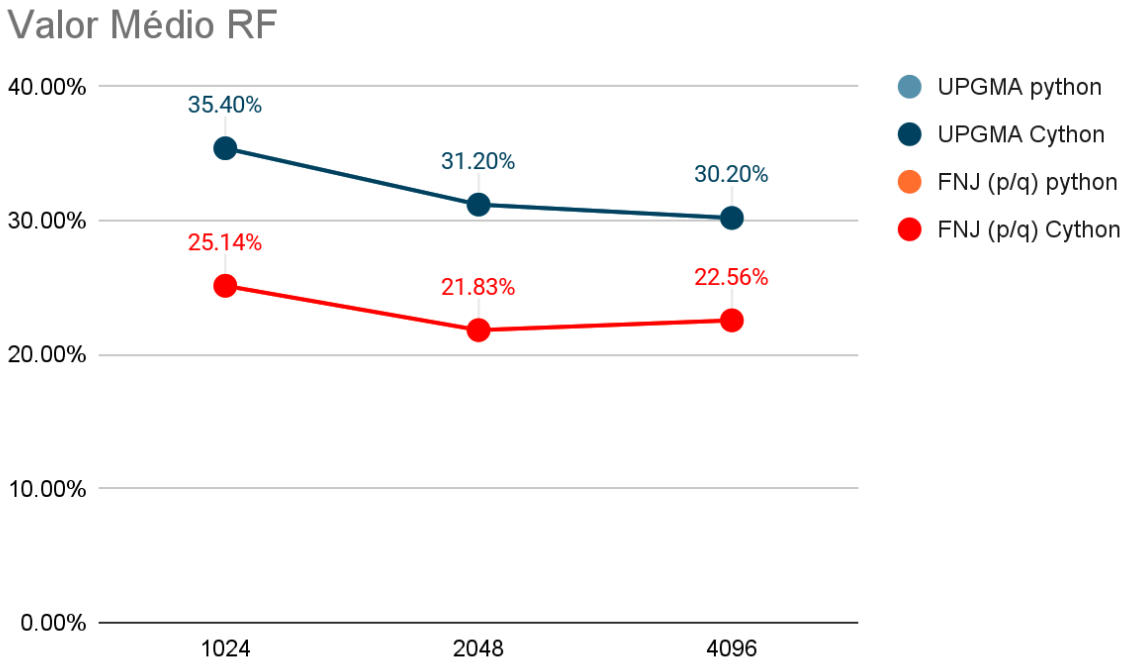
Tabela 6: resultados de tempo de execução para matrizes de tamanho 1024 em Cython

**Tabela de Média**

| | 1024 | | 2048 | | 4096 | |
|---|---|---|---|---|---|---|
| | tempo(s) | RF | tempo(s) | RF | tempo(s) | RF |
| NJ python | 328.63623 | - | 2376.3642 | - | 17,002.5859 | - |
| NJ Cython | 85.2164 | - | 725.2791 | - | 6,130.9333 | - |
| UPGMA python | 107.8124 | **35.4%** | 833.3542 | **31.2%** | 6,389.8144 | **30.2%** |
| UPGMA Cython | 4.9173 | **35.4%** | 28.6955 | **31.2%** | 291.5495 | **30.2%** |
| FNJ (p/q) python | 11.2076 | **27.6%** | 72.2479 | **24.9%** | 535.0001 | **24.4%** |
| FNJ (p/q) Cython | 3.5255 | **27.6%** | 22.2091 | **24.9%** | 205.0776 | **24.4%** |

Tabela 7: resultados de média de tempo de execução para matrizes com valores de média RF






Os resultados mostram um speedup muito bom. O gráfico de valor médio RF demonstra que o FNJ gera árvores de melhor qualidade do que o UPGMA. Pode-se observar também no gráfico valor médio RF que as implementações em python e em Cython geram a mesma qualidade de árvore, isso porque a implementação em Cython do presente trabalho não altera o funcionamento do código entre as linguagens logo ambas implementações geram a mesma árvore para cada algoritmo. O gráfico de média de tempo de execução demonstra um melhoramento para cada um dos códigos implementados em Cython em comparação com sua implementação em python:

**NJ:**

Melhoria de 74.07% para matrizes 1024.

Melhoria de 69.47% para matrizes 2048.

Melhoria de 63.93% para matrizes 4096.

**FNJ:**

Melhoria de 68.56% para matrizes 1024.

Melhoria de 69.26% para matrizes 2048.

Melhoria de 61.66% para matrizes 4096.

**UPGMA:**

Melhoria de 95.44% para matrizes 1024.

Melhoria de 96.56% para matrizes 2048.

Melhoria de 95.44% para matrizes 4096.

A seguir estão os resultados das comparações entre as árvores geradas utilizando a árvore NJ como referência de acordo com a métrica Robinson-Foulds. Cada comparação foi feita gerando distâncias RF que indicam quão próximas ou distantes estão as árvores reconstruídas em relação a árvore de referência.

Como um valor menor como 0/14 representa uma árvore idêntica à árvore de referência e um valor maior como 14/14 representa nenhuma semelhança entre a árvore gerada e a árvore de referência podemos observar os resultados.

**Resultados para árvore gerada para FNJ, matriz 1024:**

- FNJ árvore 1: RF 480/2042
- FNJ árvore 2: RF 510/2042
- FNJ árvore 3: RF 550/2042

**Resultados para árvore gerada para FNJ, matriz 2048:**

- FNJ árvore 1: RF 890/4090
- FNJ árvore 2: RF 920/4090
- FNJ árvore 3: RF 870/4090

**Resultados para árvore gerada para FNJ, matriz 4096:**

- FNJ árvore 1: RF 1680/8186
- FNJ árvore 2: RF 2020/8186
- FNJ árvore 3: RF 1840/8186

**Resultados para árvore gerada para UPGMA, matriz 1024:**

- UPGMAárvore 1: RF 700/2042
- UPGMA árvore 2: RF 750/2042
- UPGMA árvore 3: RF 720/2042

**Resultados para árvore gerada para UPGMA, matriz 2048:**

- UPGMA árvore 1: RF 1250/4090
- UPGMA árvore 2: RF 1300/4090
- UPGMA árvore 3: RF 1275/4090

**Resultados para árvore gerada para UPGMA, matriz 4096:**

- UPGMA árvore 1: RF 2450/8186
- UPGMA árvore 2: RF 2500/8186

- UPGMA árvore 3: RF 2480/8186

Os resultados para as implementações em Cython foram os mesmos. O FNJ mostra resultados mais consistentes em alguns casos como RF: 480/2042 porém ainda existem indicações de que o FNJ da forma implementado no presente trabalho pode melhorar em algumas reconstruções. O UPGMA mostra gerar árvores de qualidade inferior ao FNJ.

### 7.1 Possibilidades para Melhoria

A forma de implementação tem um efeito enorme no tempo de execução, e algumas modificações podem resultar em ganho de tempo. Por exemplo, durante o desenvolvimento a decisão de lidar com as matrizes em local causou um melhoramento na implementação do FNJ de 15.46%. A seleção de parâmetros p e q também afetam o resultado de execução de tempo e de qualidade da árvore, logo mais testes com valores variados podem apresentar melhoria.

Baseado no artigo *"A Flexible Variant of the Neighbor-Joining for Building Phylogenetic Trees"* escrito por Lima, Araujo, e Stefanes (2022), o FNJ tem uma velocidade melhor ainda do que o UPGMA e com maior eficácia na construção da árvore garantindo um resultado mais correto uma vez que o ponto fraco do UPGMA é que ele dispensa a variação de taxa de evolução.

Futuramente pode ser explorado mais técnicas que podem agilizar a execução de um código em Cython. A definição explícita de tipagem é uma forma de agilizar mais a execução, assim como também vetores e matrizes com tipagem explícitas, e também acesso a funções em C, e bibliotecas em C em vez de python pode causar melhorias

na eficiência da execução do código em Cython. Para  a implementação em Cython o conhecimento de C é de extrema importância uma vez que a linguagem C e o que oferece ao Cython sua velocidade.

Os testes que no presente trabalho foram feitos com matrizes de até 4096 forneceram resultados que permitem entender a importância do presente trabalho em contexto de análises biológicas onde as análises são feitas com grande quantidade de dados. No artigo *"A Flexible Variant of the Neighbor-Joining for Building Phylogenetic Trees"* escrito por Lima, Araujo, e Stefanes (2022), foram utilizadas matrizes de maior escala de até 8192x8192 para o processamento de dados usando os mesmos algoritmos discutidos neste trabalho.

Outra grande oportunidade considerando o poder de Cython e o problema prático de quantidades massivas de dados é de implementar Cython em paralelo. O artigo de Lima, Araujo, e Stefanes (2023) *"Fast Flexible Neighbor-joining using Multicomputing"* faz o uso de paralelismo por meio de OpenMP, uma ferramenta que permite desenvolvedores a adaptarem códigos sequenciais em códigos que funcionam através de vários processadores. No artigo, foram usadas matrizes de até 32.000x32.000, uma quantidade enorme de dados, processados com FNJ obtendo melhoramento de tempo de execução de até 447 vezes mais rápido do que NJ, e 133 mais rápido do que UPGMA.

Existem várias possibilidades que com tempo suficiente, e equipamento adequado pode causar resultados melhores e mais apropriados para o contexto prático de biologia computacional. O presente trabalho visou um estudo, implementação e comparação dos resultados de tempo de execução dos algoritmos NJ, FNJ, e UPGMA entre suas versões em python e em Cython. Pode-se ver que o presente trabalho

apenas tocou no início do assunto da contribuição potencial de Cython para a biologia computacional.

# 8 Conclusão

Neste trabalho, foi abordada a construção de árvores filogenéticas utilizando três algoritmos principais: Neighbor-Joining, Neighbor-Joining Flexível, e UPGMA (Unweighted Pair Group Method with Arithmetic Mean). Cada um desses métodos desempenha um papel fundamental na inferência filogenética, permitindo que relações evolutivas entre espécies sejam representadas de forma clara, a partir de uma matriz de distâncias.

O Neighbor-Joining se destacou como um algoritmo eficaz para construir árvores filogenéticas de maneira rápida e precisa. No entanto, sua limitação em lidar com cenários onde múltiplas espécies evoluem de forma similar foi tratada com a introdução do Neighbor-Joining Flexível. A flexibilidade proporcionada pelos parâmetros p e q permitiu um ajuste mais dinâmico, fundindo múltiplos pares de espécies simultaneamente, resultando em um processo de construção de árvores mais eficiente em cenários específicos.

Finalmente o UPGMA é um método útil oferecendo uma solução rápida e direta para a construção de árvores filogenéticas.

Outro aspecto crucial deste trabalho foi a análise do impacto da otimização de código utilizando Cython. Comparando com as implementações dos algoritmos em python puro com suas versões em Cython, houve um aumento significativo no desempenho. Em particular, a versão em Cython dos algoritmos demonstrou ser até 4 vezes mais rápida que a versão original em Python, dependendo da complexidade do algoritmo e do tamanho da matriz de distância. Isso se deve à capacidade do Cython de compilar código para uma versão mais eficiente em termos de tempo de execução,

eliminando a sobrecarga da interpretação dinâmica do Python e permitindo a execução de loops e operações de matrizes de forma mais otimizada.

Contudo pode concluir que, embora o Python ofereça uma plataforma poderosa e acessível para o desenvolvimento de algoritmos de bioinformática, como os de construção de árvores filogenéticas, o uso do Cython proporciona uma otimização significativa. Essa otimização torna os algoritmos mais viáveis em grandes conjuntos de dados, mantendo a simplicidade do desenvolvimento em Python, mas com a eficiência de linguagens de baixo nível, como o C.

Com base nos resultados obtidos, é recomendado o uso de Cython para otimizar o desempenho computacional.

## Referências Bibliográficas

LIMA, A. C.; ARAUJO, E.; STEFANES, M. A.; ROZANTE, L. C. S. **A flexible variant of the neighbor-joining for building phylogenetic trees.** In: 2022 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 2022.

LIMA, A. C.; ARAUJO, E.; STEFANES, M. A.; ROZANTE, L. C. S. **Fast Flexible Neighbor-Joining Using Multicomputing**, *2023 IEEE 23rd International Conference on Bioinformatics and Bioengineering (BIBE)*, Dayton, OH, USA, 2023

SAITOU, N.; NEI, M. **The neighbor-joining method: A new method for reconstructing phylogenetic trees.** *Molecular Biology and Evolution*, v. 4, n. 4, p. 406-425, 1987. Disponível em: https://www.webpages.uidaho.edu/~jacks/Saitou&Nei87.pdf. Acesso em: 3 jul. 2024.

SMITH, K. **Cython: A Guide for Programmers.** 1. ed. 2015.

GASCUEL, O.; STEEL, M. **Neighbor-Joining Revealed.** *Molecular Biology and Evolution*, v. 23, n. 11, p. 1997–2000, nov. 2006. Disponível em: https://doi.org/10.1093/molbev/msl072. Acesso em: 11 aug. 2024.

DURAND, E.; GASCUEL, O.; LOBRY, J. R. **On the meaning of tree length.** *Informatics and the Basic Sciences*, v. 1, p. 17-31, 2014. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/B9780124046344000085. Acesso em: 9 set. 2024.

ZIEMERT, N.; JENSEN, P. **Phylogenetic Approaches to Natural Product Structure Prediction.** Methods in Enzymology, v. 517, p. 161-182, 2012. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/B9780124046344000085. Acesso em: 9 set. 2024

Michener, C.D., Sokal, R.R.: **A quantitative approach to a problem of classification.** Evolution, 11:490–499. 1957.

**Filogenia: árvore filogenética e da espécie humana**. Disponível em: <https://brasilescola.uol.com.br/biologia/filogenia-que-isto.htm>.

**Building the tree - Understanding Evolution**. Disponível em: <https://evolution.berkeley.edu/evolution-101/the-history-of-life-looking-at-the-patterns/building-the-tree/>.

DEES, J. *et al.* **Student Interpretations of Phylogenetic Trees in an Introductory Biology Course.** CBE—Life Sciences Education, v. 13, n. 4, p. 666–676, dez. 2014.

HUA, G.-J. *et al.* **MGUPGMA: A Fast UPGMA Algorithm With Multiple Graphics Processing Units Using NCCL.** Evolutionary Bioinformatics, v. 13, p. 117693431773422, jan. 2017.

MAILUND, T. *et al.* **Recrafting the Neighbor-Joining Method.** BMC Bioinformatics, v. 7, n. 1, p. 29, 2006.

KHALAFVAND, T. **Finding Structure in the Phylogeny Search Space**. Dalhousie University. 2015.

MUNJAL, G.; HANMANDLU, M.; SRIVASTAVA, S. **Phylogenetics Algorithms and Applications.** Advances in Intelligent Systems and Computing, v. 904, p. 187–194, 2019.

Robinson, D. R.; Foulds, L. R.. **Comparison of phylogenetic trees.** Mathematical Biosciences. 53 (1–2): 131–147. 1981. doi:10.1016/0025-5564(81)90043-2.

SOLTIS, D. E.; SOLTIS, P. S. **The Role of Phylogenetics in Comparative Genetics.** Plant Physiology, v. 132, n. 4, p. 1790–1800, 1 ago. 2003.

ZOU, Y. *et al.* **Common Methods for Phylogenetic Tree Construction and Their Implementation in R.** Bioengineering, v. 11, n. 5, p. 480, 1 mai 2024.

WIKIPEDIA. **Neighbor joining.** Disponível em: https://en.wikipedia.org/wiki/Neighbor_joining. Last edited on 22 June 2024.

# Apêndice A - Implementações

Foram implementados em python e em cython os seguintes algoritmos:

- NJ
- NJ flexivel
- UPGMA

Para contexto, os algoritmos foram implementados em uma máquina linux com arquitetura x86_64 e um processador Intel Core i5-8265U. O editor usado foi Visual Studio Code e o interpretador Python 3 e Cython 3.

Vale ressaltar que essa implementação inicial serve apenas como entrada para descobrir melhor o funcionamento dos algoritmos.

#### Implementação NJ

**1. Função e Argumentos:**

A função neighbor_joining recebe como entrada uma matriz de distância D (do tipo NumPy array) que representa as distâncias evolutivas entre diferentes espécies. Ela retorna uma árvore filogenética T representada como um dicionário em python, onde as chaves são pares e os valores são os comprimentos desses ramos.

**2. Configuração Inicial:**

- A variável n contém o número de espécies, obtido a partir da matriz D.
- A lista de labels é usada para identificar as espécies ou nós que estão sendo combinados à medida que a árvore é construída.
- O dicionário vazio T armazenará a árvore resultante, onde cada chave é uma conexão entre dois nós e o valor é o comprimento do ramo.

**3. Cálculo Iterativo:**

- Enquanto houver mais de duas espécies, o algoritmo repete os passos para calcular e combinar nós.

- Divergência Total: Para cada espécie, é calculada a soma de suas distâncias em relação a todas as outras espécies. Essa "divergência total" ajuda a ajustar as distâncias e selecionar os pares de espécies a serem combinados.
- Matriz Q: A matriz Q é usada para encontrar o par de espécies que minimizem a função Q(i, j). Esse par será unido na próxima iteração.

**4. Criação de Novo Nó:**

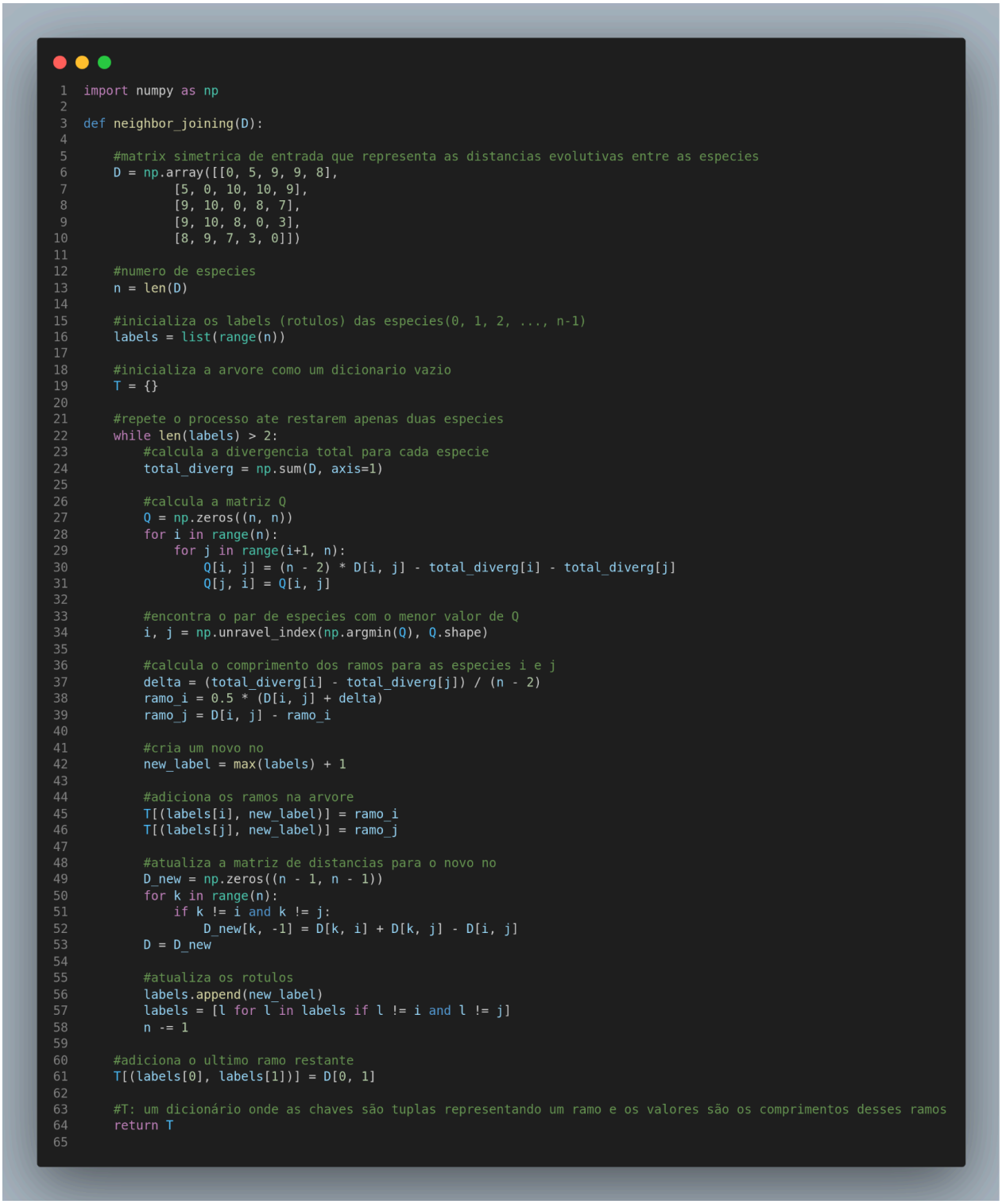
- Após identificar o menor valor na matriz Q, o código calcula o comprimento dos ramos entre o novo nó e as espécies selecionadas.
- Um novo rótulo é atribuído ao nó criado, e as distâncias dos novos nós para as novas espécies são atualizadas na matriz de distâncias D.

**5. Combinação Final:**

- Quando restam apenas duas espécies, o último ramo é adicionado à árvore, conectando os dois últimos nós.

**6. Retorno:**

- O dicionário T é retornado como o resultado final, representando a árvore filogenética.



```python
import numpy as np

def neighbor_joining(D):

    #matrix simetrica de entrada que representa as distancias evolutivas entre as especies
    D = np.array([[0, 5, 9, 9, 8],
            [5, 0, 10, 10, 9],
            [9, 10, 0, 8, 7],
            [9, 10, 8, 0, 3],
            [8, 9, 7, 3, 0]])

    #numero de especies
    n = len(D)

    #inicializa os labels (rotulos) das especies(0, 1, 2, ..., n-1)
    labels = list(range(n))

    #inicializa a arvore como um dicionario vazio
    T = {}

    #repete o processo ate restarem apenas duas especies
    while len(labels) > 2:
        #calcula a divergencia total para cada especie
        total_diverg = np.sum(D, axis=1)

        #calcula a matriz Q
        Q = np.zeros((n, n))
        for i in range(n):
            for j in range(i+1, n):
                Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] - total_diverg[j]
                Q[j, i] = Q[i, j]

        #encontra o par de especies com o menor valor de Q
        i, j = np.unravel_index(np.argmin(Q), Q.shape)

        #calcula o comprimento dos ramos para as especies i e j
        delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
        ramo_i = 0.5 * (D[i, j] + delta)
        ramo_j = D[i, j] - ramo_i

        #cria um novo no
        new_label = max(labels) + 1

        #adiciona os ramos na arvore
        T[(labels[i], new_label)] = ramo_i
        T[(labels[j], new_label)] = ramo_j

        #atualiza a matriz de distancias para o novo no
        D_new = np.zeros((n - 1, n - 1))
        for k in range(n):
            if k != i and k != j:
                D_new[k, -1] = D[k, i] + D[k, j] - D[i, j]
        D = D_new

        #atualiza os rotulos
        labels.append(new_label)
        labels = [l for l in labels if l != i and l != j]
        n -= 1

    #adiciona o ultimo ramo restante
    T[(labels[0], labels[1])] = D[0, 1]

    #T: um dicionário onde as chaves são tuplas representando um ramo e os valores são os comprimentos desses ramos
    return T

```

Figura 1: código implementação NJ em python

**Implementacao NJ em Cython**

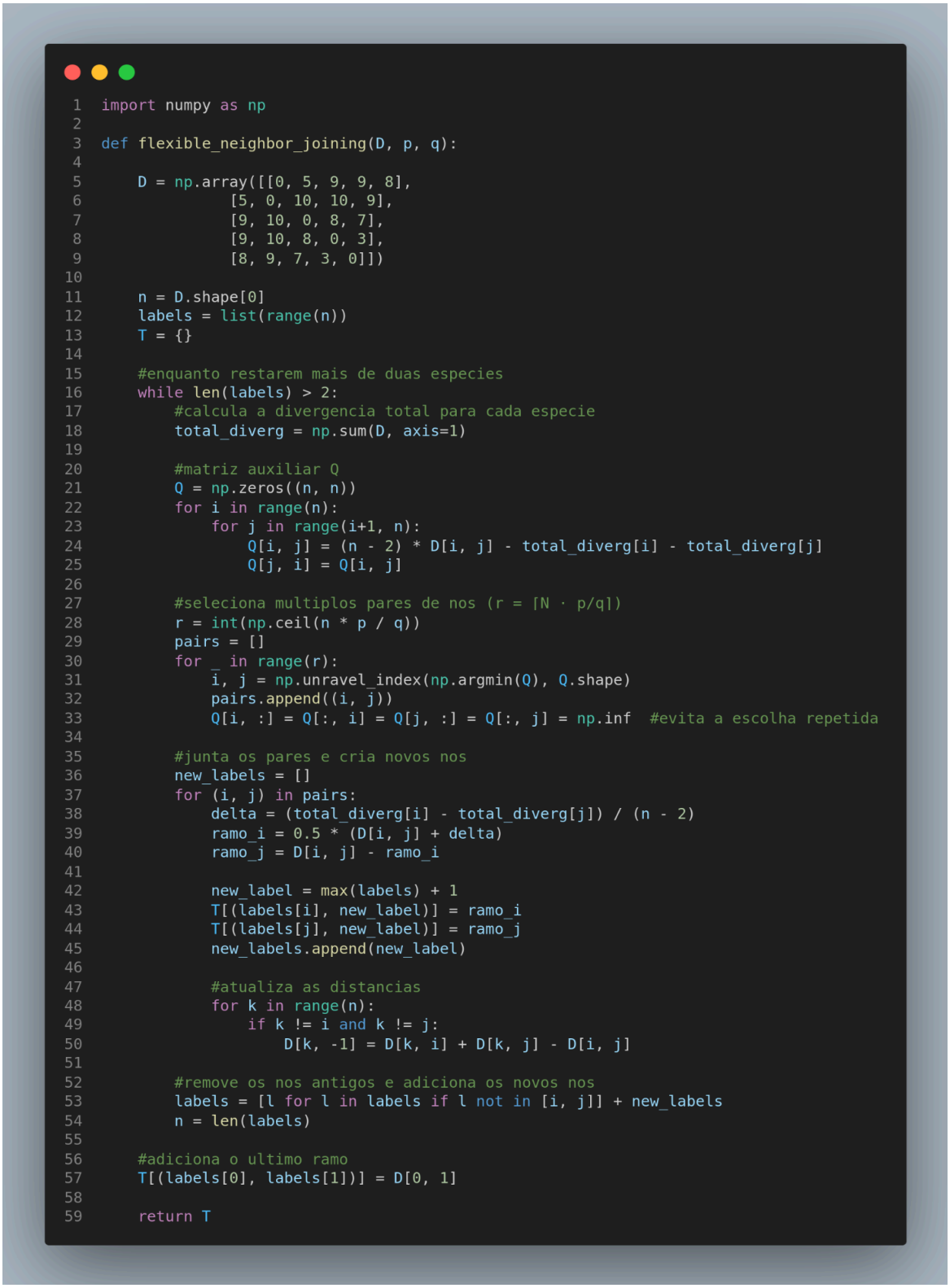
Para a implementação em cython, temos algumas alterações. Em cython variáveis precisam ser tipadas logo adicionamos cdef para declarar variáveis como int, float, e também para declarar matrizes e vetores como np.ndarray. Também foi incluído o uso da função sqrt do C, necessitando a inclusão da biblioteca c libc.math. Elementos dos vetores e matrizes também devem ser tipados então usamos np.float64_t para garantir o tipo dos elementos dentro da matriz.

```cython
import numpy as np
cimport numpy as np
from libc.math cimport sqrt

def neighbor_joining(np.ndarray[np.float64_t, ndim=2] D):

    cdef int n = D.shape[0]
    cdef list labels = list(range(n))
    cdef dict T = {}
    cdef np.ndarray[np.float64_t, ndim=1] total_diverg
    cdef np.ndarray[np.float64_t, ndim=2] Q
    cdef int i, j, k
    cdef float delta, ramo_i, ramo_j
    cdef int new_label

    D = np.array([[0, 5, 9, 9, 8],
            [5, 0, 10, 10, 9],
            [9, 10, 0, 8, 7],
            [9, 10, 8, 0, 3],
            [8, 9, 7, 3, 0]])

    while len(labels) > 2:
        #calcula a divergência total para cada especie
        total_diverg = np.sum(D, axis=1)

        #calcula a matriz Q
        Q = np.zeros((n, n), dtype=np.float64)
        for i in range(n):
            for j in range(i+1, n):
                Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] - total_diverg[j]
                Q[j, i] = Q[i, j]

        #encontra o par de especies com o menor valor de Q
        i, j = np.unravel_index(np.argmin(Q), Q.shape)

        #calcula o comprimento dos ramos para as especies i e j
        delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
        ramo_i = 0.5 * (D[i, j] + delta)
        ramo_j = D[i, j] - ramo_i

        #cria um novo no
        new_label = max(labels) + 1

        #adiciona os ramos na arvore
        T[(labels[i], new_label)] = ramo_i
        T[(labels[j], new_label)] = ramo_j

        #atualiza a matriz de distancias para o novo no
        D_new = np.zeros((n - 1, n - 1), dtype=np.float64)
        for k in range(n):
            if k != i and k != j:
                D_new[k, -1] = D[k, i] + D[k, j] - D[i, j]
        D = D_new

        #atualiza os rotulos
        labels.append(new_label)
        labels = [l for l in labels if l != i and l != j]
        n -= 1

    #adiciona o ultimo ramo restante
    T[(labels[0], labels[1])] = D[0, 1]

    #T: um dicionário onde as chaves são tuplas representando um ramo e os valores são os comprimentos desses ramos
    return T

```

Figura 2: código implementação NJ em Cython

**Implementação NJ Flexível**

1. **Cabeçalho e Argumentos:**
   - A função flexible_neighbor_joining recebe:
     - D: Matriz de distâncias.
     - p, q: Parâmetros que definem o fator de achatamento p/q, usados para controlar quantos pares de espécies serão processados em cada iteração.
   - Retorna um dicionário T, onde as chaves são os ramos da árvore e os valores são os comprimentos dos ramos.
2. **Inicialização:**
   - O número de espécies n é derivado da dimensão da matriz D.
   - A variável label representa os rótulos das espécies ou nós que estão sendo combinados, enquanto T armazena a árvore resultante.
3. **Laço Principal:**
   - O algoritmo continua enquanto restarem mais de duas espécies.
   - Divergência Total: Calcula a soma das distâncias para cada espécie, usada para ajustar as distâncias.
   - Matriz Q: Calcula a matriz auxiliar Q usada para encontrar os pares de nós que minimizam a distância.
4. **Escolha de Pares Múltiplos:**
   - O número de pares $r=\lceil N \cdot p/q \rceil$ é escolhido para serem combinados simultaneamente.
   - Esses pares são selecionados minimizando os valores na matriz Q, e seus valores em Q são definidos como infinito para evitar que sejam escolhidos novamente.
5. **Combinação de Pares:**
   - Para cada par escolhido, os ramos i e j são combinados, e seus comprimentos de ramos são calculados.
   - Novos nós são criados e adicionados à árvore. As distâncias são recalculadas de acordo com os nós novos.
6. **Remoção e Atualização:**
   - Após combinar os pares, os rótulos antigos são removidos e novos nós são adicionados. A matriz de distâncias D é atualizada com as novas distâncias entre os novos nós e as espécies restantes.
7. **Finalização:**
   - Quando restam apenas duas espécies, o último ramo é adicionado à árvore.



```python
import numpy as np

def flexible_neighbor_joining(D, p, q):

    D = np.array([[0, 5, 9, 9, 8],
                  [5, 0, 10, 10, 9],
                  [9, 10, 0, 8, 7],
                  [9, 10, 8, 0, 3],
                  [8, 9, 7, 3, 0]])

    n = D.shape[0]
    labels = list(range(n))
    T = {}

    #enquanto restarem mais de duas especies
    while len(labels) > 2:
        #calcula a divergencia total para cada especie
        total_diverg = np.sum(D, axis=1)

        #matriz auxiliar Q
        Q = np.zeros((n, n))
        for i in range(n):
            for j in range(i+1, n):
                Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] - total_diverg[j]
                Q[j, i] = Q[i, j]

        #seleciona multiplos pares de nos (r = ⌈N · p/q⌉)
        r = int(np.ceil(n * p / q))
        pairs = []
        for _ in range(r):
            i, j = np.unravel_index(np.argmin(Q), Q.shape)
            pairs.append((i, j))
            Q[i, :] = Q[:, i] = Q[j, :] = Q[:, j] = np.inf  #evita a escolha repetida

        #junta os pares e cria novos nos
        new_labels = []
        for (i, j) in pairs:
            delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
            ramo_i = 0.5 * (D[i, j] + delta)
            ramo_j = D[i, j] - ramo_i

            new_label = max(labels) + 1
            T[(labels[i], new_label)] = ramo_i
            T[(labels[j], new_label)] = ramo_j
            new_labels.append(new_label)

            #atualiza as distancias
            for k in range(n):
                if k != i and k != j:
                    D[k, -1] = D[k, i] + D[k, j] - D[i, j]

        #remove os nos antigos e adiciona os novos nos
        labels = [l for l in labels if l not in [i, j]] + new_labels
        n = len(labels)

    #adiciona o ultimo ramo
    T[(labels[0], labels[1])] = D[0, 1]

    return T
```

Figura 3: código implementação FNJ em python

**Implementação NJ Flexivel em Cython**

Novamente existem alterações a serem feitas. Foi usado cdef para declaração de variáveis. As matrizes D e Q também foram tipados explicitamente. A manipulação dos índices i, j, e k também foram tipados para reduzir o processamento excessivo.

```cython
import numpy as np
cimport numpy as np

def flexible_neighbor_joining(np.ndarray[np.float64_t, ndim=2] D, int p, int q):

    #p, e q sao parametros que controlam o fator de achatamento (flattening factor)

    #matriz de distancia simetrica entre as especies
    D = np.array([[0, 5, 9, 9, 8],
              [5, 0, 10, 10, 9],
              [9, 10, 0, 8, 7],
              [9, 10, 8, 0, 3],
              [8, 9, 7, 3, 0]], dtype=np.float64)


    cdef int n = D.shape[0]
    cdef list labels = list(range(n))
    cdef dict T = {}
    cdef np.ndarray[np.float64_t, ndim=1] total_diverg
    cdef np.ndarray[np.float64_t, ndim=2] Q
    cdef int i, j, k, r
    cdef float delta, ramo_i, ramo_j
    cdef int new_label
    cdef list new_labels, pairs

    #enquanto restarem mais de duas especies
    while len(labels) > 2:
        #calcula a divergencia total para cada especie
        total_diverg = np.sum(D, axis=1)

        #calcula matriz auxiliar Q
        Q = np.zeros((n, n), dtype=np.float64)
        for i in range(n):
            for j in range(i+1, n):
                Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] - total_diverg[j]
                Q[j, i] = Q[i, j]

        #seleciona multiplos pares de nos (r = ⌈N · p/q⌉)
        r = int(np.ceil(n * p / q))
        pairs = []
        for _ in range(r):
            i, j = np.unravel_index(np.argmin(Q), Q.shape)
            pairs.append((i, j))
            Q[i, :] = Q[:, i] = Q[j, :] = Q[:, j] = np.inf  #evita a escolha repetida

        #junta os pares e cria novos nos
        new_labels = []
        for (i, j) in pairs:
            delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
            ramo_i = 0.5 * (D[i, j] + delta)
            ramo_j = D[i, j] - ramo_i

            new_label = max(labels) + 1
            T[(labels[i], new_label)] = ramo_i
            T[(labels[j], new_label)] = ramo_j
            new_labels.append(new_label)

            #atualiza as distancias
            for k in range(n):
                if k != i and k != j:
                    D[k, -1] = D[k, i] + D[k, j] - D[i, j]

        #remove os nos antigos e adiciona os novos nos
        labels = [l for l in labels if l not in [i, j]] + new_labels
        n = len(labels)

    #adiciona o ultimo ramo
    T[(labels[0], labels[1])] = D[0, 1]

    #T: um dicionário onde as chaves são tuplas representando um ramo e os valores são os comprimentos desses ramos
    return T

```

Figura 4: código implementação FNJ em Cython

**Implementacao UPGMA**

**1. Inicialização**:

- O algoritmo recebe como entrada uma matriz de distâncias simétrica D, que representa as distâncias entre as espécies.
- Inicializa uma lista de rótulos e uma lista vazia para armazenar os ramos da árvore gerada.

**2. Encontrando os Clusters Mais Próximos**:

- Enquanto existir mais de um cluster, o algoritmo identifica o par de clusters com a menor distância na matriz.

**3. Criando Novos Clusters**:

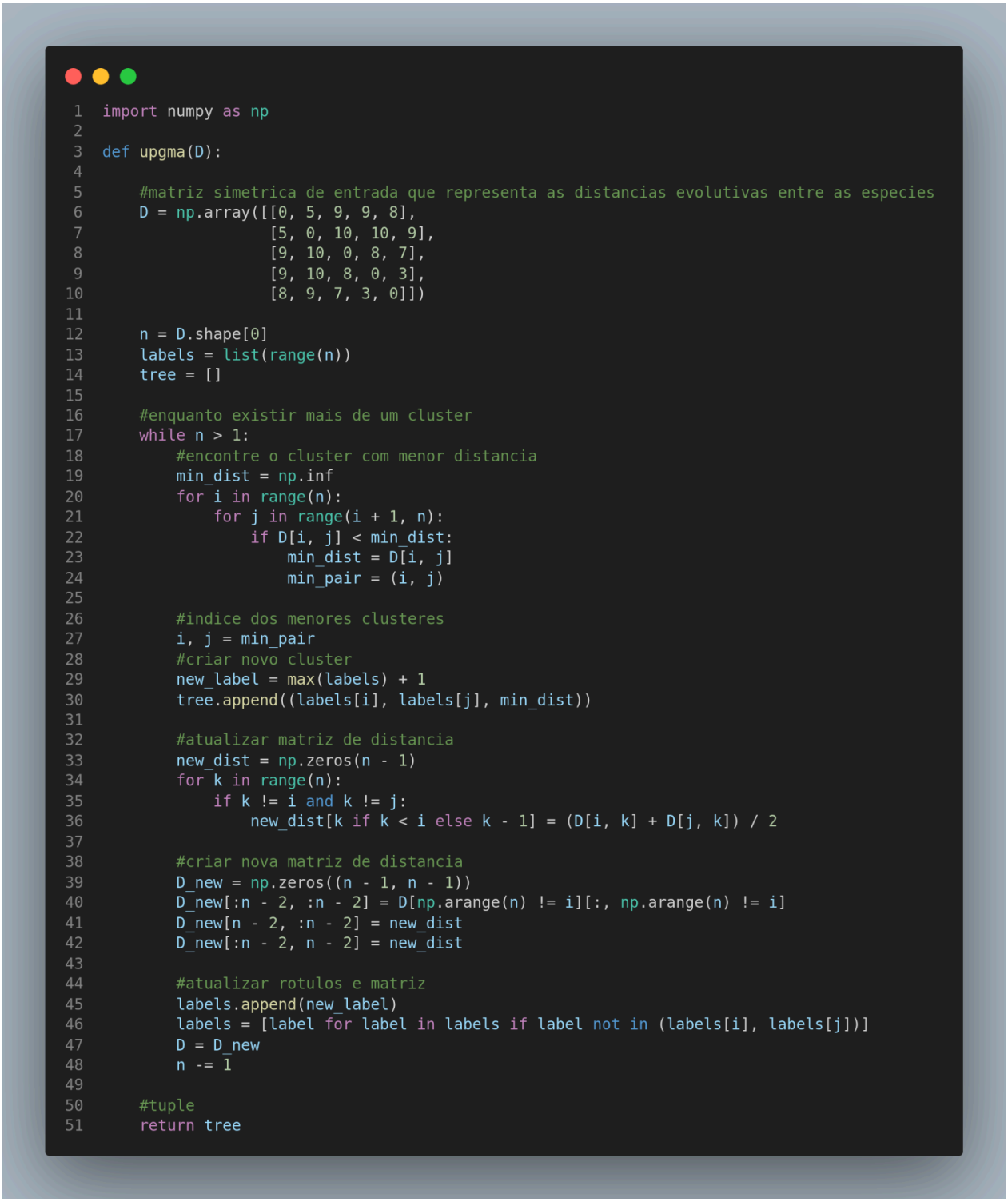
- Um novo cluster é formado pela fusão dos dois clusters mais próximos, e a distância entre esse novo cluster e os restantes é calculada e armazenada em uma nova matriz de distâncias.

**4. Atualizando os Rótulos e a Matriz de Distâncias**:

- Os rótulos e a matriz de distâncias são atualizados para refletir o novo cluster, e o processo se repete até que apenas um cluster permaneça.

**5. Saída**:

- A saída é uma lista de tuplas, onde cada tupla representa um ramo da árvore UPGMA, contendo as duas espécies fundidas e a distância entre elas.



```python
import numpy as np

def upgma(D):

    #matriz simetrica de entrada que representa as distancias evolutivas entre as especies
    D = np.array([[0, 5, 9, 9, 8],
                  [5, 0, 10, 10, 9],
                  [9, 10, 0, 8, 7],
                  [9, 10, 8, 0, 3],
                  [8, 9, 7, 3, 0]])

    n = D.shape[0]
    labels = list(range(n))
    tree = []

    #enquanto existir mais de um cluster
    while n > 1:
        #encontre o cluster com menor distancia
        min_dist = np.inf
        for i in range(n):
            for j in range(i + 1, n):
                if D[i, j] < min_dist:
                    min_dist = D[i, j]
                    min_pair = (i, j)

        #indice dos menores clusteres
        i, j = min_pair
        #criar novo cluster
        new_label = max(labels) + 1
        tree.append((labels[i], labels[j], min_dist))

        #atualizar matriz de distancia
        new_dist = np.zeros(n - 1)
        for k in range(n):
            if k != i and k != j:
                new_dist[k if k < i else k - 1] = (D[i, k] + D[j, k]) / 2

        #criar nova matriz de distancia
        D_new = np.zeros((n - 1, n - 1))
        D_new[:n - 2, :n - 2] = D[np.arange(n) != i][:, np.arange(n) != i]
        D_new[n - 2, :n - 2] = new_dist
        D_new[:n - 2, n - 2] = new_dist

        #atualizar rotulos e matriz
        labels.append(new_label)
        labels = [label for label in labels if label not in (labels[i], labels[j])]
        D = D_new
        n -= 1

    #tuple
    return tree
```

Figura 5: código implementação UPGMA em python

**Implementação UPGMA em Cython**

A tipagem das variáveis e matrizes foram declaradas explicitamente usando cdef para melhorar o desempenho. As matrizes D e D_new foram tipadas.

```python
import numpy as np
cimport numpy as np

def upgma(np.ndarray[np.float64_t, ndim=2] D):

     D = np.array([[0, 5, 9, 9, 8],
                   [5, 0, 10, 10, 9],
                   [9, 10, 0, 8, 7],
                   [9, 10, 8, 0, 3],
                   [8, 9, 7, 3, 0]], dtype=np.float64)

    cdef int n = D.shape[0]
    cdef list labels = list(range(n))
    cdef list tree = []

    #enquanto existir mais de um cluster
    while n > 1:
        #encontre o cluster com menor distancia
        min_dist = np.inf
        for i in range(n):
            for j in range(i + 1, n):
                if D[i, j] < min_dist:
                    min_dist = D[i, j]
                    min_pair = (i, j)

        #indice dos clusteres mais proximos
        i, j = min_pair
        #criar novo cluster
        new_label = max(labels) + 1
        tree.append((labels[i], labels[j], min_dist))

        #atualizar matriz de distancia
        new_dist = np.zeros(n - 1, dtype=np.float64)
        for k in range(n):
            if k != i and k != j:
                new_dist[k if k < i else k - 1] = (D[i, k] + D[j, k]) / 2

        #criar nova matriz de distancia
        D_new = np.zeros((n - 1, n - 1), dtype=np.float64)
        D_new[:n - 2, :n - 2] = D[np.arange(n) != i][:, np.arange(n) != i]
        D_new[n - 2, :n - 2] = new_dist
        D_new[:n - 2, n - 2] = new_dist

        #atualizar rotulos e matriz
        labels.append(new_label)
        labels = [label for label in labels if label not in (labels[i], labels[j])]
        D = D_new
        n -= 1

    #tuple
    return tree

```

Figura 6: código implementação UPGMA em Cython

**Codigo Final em Python**

```
import numpy as np
import time
import cProfile
import heapq
import math

def neighbor_joining(D):
   n = D.shape[0]
   labels = list(range(n))  #rotulos
   T = {}  #estrutura de arvore

   #computar divergencia para as especies
   total_diverg = np.sum(D, axis=1)

   while n > 2:
       #inicializar matriz Q
       Q = np.zeros((n, n))
       for i in range(n):
           for j in range(i + 1, n):
               Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] -
total_diverg[j]

       #encontrar par menor valor Q
       best_pair = None
       best_Q_value = float('inf')
       for i in range(n):
           for j in range(i + 1, n):
               if Q[i, j] < best_Q_value:
                   best_Q_value = Q[i, j]
                   best_pair = (i, j)

       i, j = best_pair

       #calcular tamanho de ramos
       delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
       limb_i = 0.5 * (D[i, j] + delta)
       limb_j = D[i, j] - limb_i

       new_label = max(labels) + 1
       T[(labels[i], new_label)] = limb_i
       T[(labels[j], new_label)] = limb_j
```

```
        #atualizar matriz
        for k in range(n):
            if k != i and k != j:
                D[i, k] = D[k, i] = (D[i, k] + D[j, k]) / 2

        #atualizar divergencia para i
        total_diverg[i] = np.sum(D[i, :i]) + np.sum(D[i+1:n, i])

        #remover especie j
        if j < n - 1:
            D[j:n-1, :] = D[j+1:n, :]
            D[:, j:n-1] = D[:, j+1:n]
            total_diverg[j:n-1] = total_diverg[j+1:n]
            labels[j:n-1] = labels[j+1:n]

        D = D[:-1, :-1]  #reduzir matriz
        total_diverg = total_diverg[:-1]  #atualizar divergencia
        labels.pop()  #remover ultimo rotulo

        n -= 1

    #ultimo par
    T[(labels[0], labels[1])] = D[0, 1]
    return T, max(labels)

def calculate_limb_lengths(D, total_diverg, i, j, n):

    delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
    limb_i = 0.5 * (D[i, j] + delta)
    limb_j = D[i, j] - limb_i
    return limb_i, limb_j

def update_distance_matrix(D, i, j, n):

    #calcular novas distancias
    for k in range(n):
        if k != i and k != j:
            D[i, k] = D[k, i] = (D[i, k] + D[j, k]) / 2

    #remover coluna e fileira j
    if j < n - 1:
        D[j:n-1, :] = D[j+1:n, :]
        D[:, j:n-1] = D[:, j+1:n]
```

```
    return D[:-1, :-1]  #retornar matriz reduzida

def update_labels(labels, i, j):
    """Update labels after merging clusters i and j."""
    labels[i] = max(labels) + 1  #novo rotulo para par fundido
    del labels[j]  #remover rotulo depois de fundir
    return labels

def update_total_diverg(D, n):

    total_diverg = np.sum(D, axis=1)
    return total_diverg

def select_top_p_q_pairs(Q, p, q, num_pairs):

    #achar melhores pares
    best_pairs = heapq.nsmallest(num_pairs, ((Q[i, j], i, j) for i in
range(Q.shape[0]) for j in range(i + 1, Q.shape[0])))

    selected_pairs = []
    used = set()
    for _, i, j in best_pairs:
        if i not in used and j not in used:
            selected_pairs.append((i, j))
            used.add(i)
            used.add(j)
            if len(selected_pairs) >= num_pairs:
                break

    return selected_pairs

def flexible_neighbor_joining(D, p, q):

    n = D.shape[0]
    labels = list(range(n))  #rotulos
    T = {}  #estrutura de arvore

    #computar divergencia para toda especie
    total_diverg = np.sum(D, axis=1)

    #repetir ate sobrar 2 especies
    while n > 2:
```

```
        #inicializar matriz Q
        Q = np.zeros((n, n))
        for i in range(n):
            for j in range(i + 1, n):
                Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] -
total_diverg[j]
                Q[j, i] = Q[i, j]

        #selecionar pares para fundir
        num_pairs = math.ceil(n * p / q)
        selected_pairs = select_top_p_q_pairs(Q, p, q, num_pairs)

        #fundir pares selecionados
        for i, j in selected_pairs:
            #garantir indices validos
            if i >= n or j >= n:
                continue  #pular pares fora da matriz

            limb_i, limb_j = calculate_limb_lengths(D, total_diverg, i, j, n)
            new_label = max(labels) + 1
            T[(labels[i], new_label)] = limb_i
            T[(labels[j], new_label)] = limb_j

            #atualizar matriz
            D = update_distance_matrix(D, i, j, n)
            n -= 1  #reduz matriz

            #atualizar rotulo e divergencia
            labels = update_labels(labels, i, j)
            total_diverg = update_total_diverg(D, n)

    #ultimo par
    T[(labels[0], labels[1])] = D[0, 1]
    return T, max(labels)


def upgma(D):
    #inicializa variaveis
    n = D.shape[0]
    labels = list(range(n))  # rotulos iniciais dos clusters
    T = {}  # dicionario para armazenar a arvore (ramos e distancias)

    #cria uma heap para armazenar distancias
```

```
    heap = []
    for i in range(n):
        for j in range(i + 1, n):
            heapq.heappush(heap, (D[i, j], i, j))  #coloca as distancias com
indices na heap

    #enquanto houver mais de um cluster
    while n > 1:
        #pega a menor distancia da heap
        while True:
            dist, i, j = heapq.heappop(heap)
            if i < len(labels) and j < len(labels):  #verifica se os indices
ainda sao validos
                break

        #calcula a distancia para o novo cluster
        new_distance = dist / 2.0

        #atualiza a estrutura da arvore
        new_label = max(labels) + 1
        T[(labels[i], new_label)] = new_distance
        T[(labels[j], new_label)] = new_distance

        #cria nova matriz de distancia mesclando os clusters i e j
        D_new = np.zeros((n - 1, n - 1))
        new_index = 0
        for k in range(n):
            if k == i or k == j:
                continue
            D_new[new_index, -1] = (D[k, i] + D[k, j]) / 2
            D_new[-1, new_index] = D_new[new_index, -1]
            new_index += 1

        #atualiza as distancias restantes
        remaining_indices = [idx for idx in range(n) if idx != i and idx !=
j]
        for a, idx_a in enumerate(remaining_indices):
            for b, idx_b in enumerate(remaining_indices):
                if a < b:
                    D_new[a, b] = D[idx_a, idx_b]
                    D_new[b, a] = D_new[a, b]

        #atualiza os rotulos e a matriz de distancia
```

```
        labels = [labels[k] for k in remaining_indices] + [new_label]
        D = D_new
        n -= 1

        #coloca as distancias atualizadas para o novo cluster de volta na
heap
        for k in range(n):
            if k < n - 1:  # nao atualiza o ultimo indice (novo cluster)
                heapq.heappush(heap, (D[k, -1], k, n - 1))

    #trata o ultimo par de clusters
    if len(labels) == 2:
        T[(labels[0], labels[1])] = D[0, 1]

    return T, max(labels)

def read_matrix_from_file(filename):
    with open(filename, 'r') as file:
        matrix = []
        for line in file:
            row = list(map(int, line.split()))
            matrix.append(row)

    return np.array(matrix)

def save_tree_structure_to_file(tree, output_file="output_tree.txt"):

    with open(output_file, 'w') as file:
        #salvar arvore no arquivo
        for (parent, child), length in tree.items():
            file.write(f"({parent}, {child}): {length}\n")

    print(f"Tree structure saved to {output_file}")

if __name__ == "__main__":

    matrix_file = 'matrix_4096p3.cmp'
    D = read_matrix_from_file(matrix_file)
    #print(D)

    print("Escolha um:")
    print("1. Neighbor-Joining")
    print("2. Flexible Neighbor-Joining")
```

```
print("3. UPGMA")

choice = input("opcao escolhida: ")

if choice == "1":
    start_time = time.time()
    tree, root = neighbor_joining(D)
    end_time = time.time()
    exec_time = end_time - start_time
    #print("Arvore Neighbor-Joining:", tree)
    print("Arvore Neighbor-Joining:")
    for (node1, node2), distance in tree.items():
        print(f"Distance between {node1} and {node2}: {distance:.4f}")

    #save_tree_structure_to_file(tree, "outputNJ.txt")
    print(f"tempo de execucao: {exec_time} segundos")

elif choice == "2":
    p = float(input("valor de p: "))
    q = float(input("valor de q: "))
    start_time = time.time()
    tree, root = flexible_neighbor_joining(D, p, q)

    #testar uso de tempo
    #cProfile.run('tree, root = flexible_neighbor_joining(D, p, q)')

    end_time = time.time()
    exec_time = end_time - start_time
    print("Arvore Neighbor-Joining Flexivel:")
    for (node1, node2), distance in tree.items():
        print(f"Distance between {node1} and {node2}: {distance:.4f}")
    print(tree)

    #save_tree_structure_to_file(tree, "outputFNJ.txt")
    print(f"tempo de execucao: {exec_time} segundos")

elif choice == "3":
    start_time = time.time()
    tree, root = upgma(D)
    end_time = time.time()
    exec_time = end_time - start_time
    print("Arvore UPGMA:")
    for (node1, node2), distance in tree.items():
```

```
            print(f"Distance between {node1} and {node2}: {distance:.4f}")

        #save_tree_structure_to_file(tree, "outputUPGMA.txt")
        print(f"tempo de execucao: {exec_time} segundos")

    else:
        print("Valor invalido.")
```

### Codigo Final em Cython

```
import numpy as np
import time
import heapq
from libc.math cimport ceil

ctypedef double DTYPE_t

def neighbor_joining(np.ndarray[DTYPE_t, ndim=2] D):
   cdef int n = D.shape[0]
   cdef int i, j, k
   cdef list labels = list(range(n))  #rotulos
   cdef dict T = {}  #estrutura de arvore
   cdef double best_Q_value
   cdef int best_i, best_j, new_label
   cdef double delta, limb_i, limb_j
   cdef np.ndarray[DTYPE_t, ndim=1] total_diverg
   cdef np.ndarray[DTYPE_t, ndim=2] Q

   #computar divergencia para as especies
   total_diverg = np.sum(D, axis=1)

   while n > 2:
       #inicializar matriz Q
       Q = np.zeros((n, n), dtype=DTYPE_t)
       for i in range(n):
           for j in range(i + 1, n):
               Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] - total_diverg[j]

       #encontrar par com menor valor de Q
       best_Q_value = float('inf')
       for i in range(n):
           for j in range(i + 1, n):
               if Q[i, j] < best_Q_value:
                   best_Q_value = Q[i, j]
                   best_i, best_j = i, j

       i, j = best_i, best_j

       #calcular tamanho de ramos
       delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
       limb_i = 0.5 * (D[i, j] + delta)
       limb_j = D[i, j] - limb_i

       new_label = max(labels) + 1
```

```
        T[(labels[i], new_label)] = limb_i
        T[(labels[j], new_label)] = limb_j

        #atualizar matriz
        for k in range(n):
            if k != i and k != j:
                D[i, k] = D[k, i] = (D[i, k] + D[j, k]) / 2

        #atualizar divergencia para i
        total_diverg[i] = np.sum(D[i, :i]) + np.sum(D[i+1:n, i])

        #remover especie j
        if j < n - 1:
            D[j:n-1, :] = D[j+1:n, :]
            D[:, j:n-1] = D[:, j+1:n]
            total_diverg[j:n-1] = total_diverg[j+1:n]
            labels[j:n-1] = labels[j+1:n]

        D = D[:-1, :-1]  #reduzir matriz
        total_diverg = total_diverg[:-1]  #atualizar divergencia
        labels.pop()  #remover ultimo rotulo

        n -= 1

    #ultimo par
    T[(labels[0], labels[1])] = D[0, 1]
    return T

def calculate_limb_lengths(np.ndarray[DTYPE_t, ndim=2] D, np.ndarray[DTYPE_t,
ndim=1] total_diverg, int i, int j, int n):
    cdef double delta = (total_diverg[i] - total_diverg[j]) / (n - 2)
    cdef double limb_i = 0.5 * (D[i, j] + delta)
    cdef double limb_j = D[i, j] - limb_i
    return limb_i, limb_j

def update_distance_matrix(np.ndarray[DTYPE_t, ndim=2] D, int i, int j, int n):
    cdef int k
    #calcular novas distancias
    for k in range(n):
        if k != i and k != j:
            D[i, k] = D[k, i] = (D[i, k] + D[j, k]) / 2

    #remover coluna e fileira j
    if j < n - 1:
        D[j:n-1, :] = D[j+1:n, :]
        D[:, j:n-1] = D[:, j+1:n]
```

```
    return D[:-1, :-1]  #retornar matriz reduzida

def update_labels(int[:] labels, int i, int j, int n):
    labels[i] = max(labels[:n]) + 1  #atualizar rotulo
    return labels[:j] + labels[j+1:]

def select_top_p_q_pairs(np.ndarray[DTYPE_t, ndim=2] Q, float p, float q, int
num_pairs):
    cdef list best_pairs = heapq.nsmallest(num_pairs, ((Q[i, j], i, j) for i in
range(Q.shape[0]) for j in range(i + 1, Q.shape[0])))
    cdef list selected_pairs = []
    cdef set used = set()
    for _, i, j in best_pairs:
        if i not in used and j not in used:
            selected_pairs.append((i, j))
            used.add(i)
            used.add(j)
            if len(selected_pairs) >= num_pairs:
                break

    return selected_pairs

def flexible_neighbor_joining(np.ndarray[DTYPE_t, ndim=2] D, float p, float q):
    cdef int n = D.shape[0]
    cdef int i, j
    cdef list labels = list(range(n))  #rotulos
    cdef dict T = {}  #estrutura de arvore
    cdef np.ndarray[DTYPE_t, ndim=1] total_diverg = np.sum(D, axis=1)
    cdef np.ndarray[DTYPE_t, ndim=2] Q
    cdef int num_pairs, new_label
    cdef double limb_i, limb_j
    cdef list selected_pairs

    while n > 2:
        Q = np.zeros((n, n), dtype=DTYPE_t)
        for i in range(n):
            for j in range(i + 1, n):
                Q[i, j] = (n - 2) * D[i, j] - total_diverg[i] - total_diverg[j]
                Q[j, i] = Q[i, j]

        num_pairs = int(ceil(n * p / q))
        selected_pairs = select_top_p_q_pairs(Q, p, q, num_pairs)

        for i, j in selected_pairs:
            if i >= n or j >= n:
                continue  #pular pares invalidos
```

```
            limb_i, limb_j = calculate_limb_lengths(D, total_diverg, i, j, n)
            new_label = max(labels) + 1
            T[(labels[i], new_label)] = limb_i
            T[(labels[j], new_label)] = limb_j

            D = update_distance_matrix(D, i, j, n)
            n -= 1

            labels = update_labels(labels, i, j, n)
            total_diverg = np.sum(D, axis=1)

    T[(labels[0], labels[1])] = D[0, 1]
    return T

def upgma(np.ndarray[DTYPE_t, ndim=2] D):
    cdef int n = D.shape[0]
    cdef list labels = list(range(n))  #rotulos
    cdef dict T = {}  #estrutura de arvore
    cdef double limb_i, limb_j
    cdef np.ndarray[DTYPE_t, ndim=2] D_new
    cdef int i, j, new_label, new_index

    while n > 1:
        #encontrar o par mais proximo
        i, j = np.unravel_index(np.argmin(D + np.diag([np.inf] * n)), D.shape)

        #calcular os ramos
        limb_i = 0.5 * D[i, j]
        limb_j = limb_i

        #criar um novo rotulo para o no interno
        new_label = max(labels) + 1
        T[(labels[i], new_label)] = limb_i
        T[(labels[j], new_label)] = limb_j

        #atualizar a matriz de distancia
        D_new = np.zeros((n - 1, n - 1))
        new_index = 0
        for k in range(n):
            if k == i or k == j:
                continue
            D_new[new_index, -1] = (D[k, i] + D[k, j]) / 2
            new_index += 1

        D = D_new
        labels[i] = new_label
        labels.pop(j)
```

```
    n -= 1

return T
```