ALGORITMO NEIGHBOR-JOINING

Leonardo Azzi Martins

INF05018 - Biologia Computacional — Outubro de 2025

AGENDA

1. Pseudocódigo
2. Implementação
3. Resultados

Pseudocódigo

Algorithm 1: Neighbor-Joining

Input: Matriz de distâncias D, lista de labels L
Output: Árvore Newick N

Inicializar nodos para cada OTU; $n \leftarrow |L|$;
while $n > 2$ **do**
    $u_i \leftarrow \frac{1}{n-2} \sum_{j \neq i} D_{ij}$ para cada i;
    Escolher i, j com $D_{ij} - u_i - u_j$ mínimo, $i \neq j$;
    Criar nodo (ij) conectando i e j;
    $v_i = \frac{1}{2} D_{ij} + \frac{1}{2}(u_i - u_j)$;
    $v_j = \frac{1}{2} D_{ij} + \frac{1}{2}(u_j - u_i)$;
    Para cada $k \neq i, j$: $D_{(ij)k} = \frac{D_{ik} + D_{jk} - D_{ij}}{2}$;
    Remover linhas/colunas i, j de D e adicionar linha e coluna $D_{(ij)k}$;
    Atualizar $L \leftarrow (L \setminus \{i, j\}) \cup \{(ij)\}$;
    $N \leftarrow$ vértice entre nodos (ij) e nodos i e j com distância v_i e v_j;
    $n \leftarrow n - 1$;
end
$N \leftarrow$ vértice entre os dois últimos nodos (l, m) com distância D_{lm};
return N

Implementação

Tem como objetivo criar nodos iniciais para cada OTU, criar novos nodos nomeados e manter uma lista dos nodos criados durante o ciclo de vida do algoritmo.

- Atributos:
  - **nodes** (list[newick.Node])
    - Lista de nodos
- Métodos:
  - **__init__**(matrix: np.ndarray, labels:list)
    - Cria um nodo nomeado para cada label L
  - **create_node**(name: str) -> newick.Node
    - Retorna um novo nodo nomeado

```
1  class Newick:
2      def __init__(self, matrix: np.ndarray, labels:
           list):
3          if matrix.shape[0] != len(labels) or matrix.
               shape[0] != len(labels):
4              raise Exception("O n mero de labels
                   diferente do n mero de linhas/colunas
                   .")
5          self.nodes = [nw.Node.create() for _ in labels
               ]
6          for idx, node in enumerate(self.nodes):
7              node.name = f"{labels[idx]}"
8
9      def create_node(self, name: str):
10         new_node = nw.Node.create()
11         new_node.name = f"{name}"
12
13         return new_node
```

Retorna uma tupla com a matriz de distâncias a partir de um arquivo .txt e as labels a partir de um .txt separado por vírgulas.

```
1 def get_matrix(matrix_file: str, labels_file: str) ->
      tuple[np.ndarray, list]:
2     with open(labels_file, 'r') as f:
3         labels_str = f.read()
4
5     labels = [x.strip() for x in labels_str.split(',')
          ]
6     matrix = np.loadtxt(matrix_file)
7
8     return matrix, labels
```

Retorna uma lista de $u_i \leftarrow \sum_{j \neq i} \frac{D_{ij}}{n-2}$ para cada OTU i onde n é o número de OTUs

```
1  def compute_u(matrix: np.ndarray, clusters: list):
2      n = len(clusters)
3      u_list = []
4
5      for i, _ in enumerate(matrix[0]):
6          acc = 0
7          for j, _ in enumerate(matrix[1]):
8              if j != i:
9                  acc = acc + matrix[i][j] / (n - 2)
10         u_list.append(acc)
11
12     return u_list
```

Retorna i, j com $D_{ij} - u_i - u_j$ mínimo, com $i \neq j$

```
1  def find_min_ij(matrix: np.ndarray, u_list: list)->
       tuple[int, int]:
2      u = np.array(u_list, dtype=float)
3
4      # Broadcasting para computar os valores
           diretamente
5      D_minus_u = matrix - u[:, None] - u[None, :]
6
7      # Atribui infinito a diagonal i=j
8      np.fill_diagonal(D_minus_u, np.inf)
9
10     # Encontra indice com menores valores
11     i,j = np.unravel_index(np.argmin(D_minus_u),
           D_minus_u.shape)
12
13     return (i, j)
```

Computa o tamanho das arestas entre o nodo (ij) e os nodos i e j:

$$v_i = \frac{1}{2}D_{ij} + \frac{1}{2}(u_i - u_j)$$
$$v_j = \frac{1}{2}D_{ij} + \frac{1}{2}(u_j - u_i)$$

```
1 def compute_edges(matrix: np.ndarray, u_list: list, i:
      int, j: int) -> tuple[float, float]:
2     v_i = 1/2 * matrix[i,j] + 1/2 * (u_list[i] -
          u_list[j])
3     v_j = 1/2 * matrix[i,j] + 1/2 * (u_list[j] -
          u_list[i])
4
5     return v_i, v_j
```

FUNÇÃO COMPUTE_DISTANCES()

Para cada $k \neq i, j$, computa $D_{(ij)k} = \frac{D_{ik}+D_{jk}-D_{ij}}{2}$ como uma matriz

```
1  def compute_distances(matrix: np.ndarray, i: int, j:
       int) -> np.ndarray:
2      # Cria uma mascara para obter os indices
           diferentes de i e j
3      n = matrix.shape[0]
4      mask = np.ones(n, dtype=bool)
5      mask[[i,j]] = False
6      k_indices = np.arange(n)[mask]
7
8      # Computa D_ijk com broadcasting, gerando um array
           com elementos ij_k
9      D_ijk = (matrix[i, k_indices] + matrix[j,
           k_indices] - matrix[i, j]) / 2
10
11     return D_ijk
```

Remover linhas/colunas i, j e adicionar (ij) na matriz D

```
1  def update_distances(matrix: np.ndarray, D_ijk: np.
       ndarray, i:int, j:int) -> np.ndarray:
2
3      upd_matrix = np.delete(matrix, [i, j], axis=0)
4      upd_matrix = np.delete(upd_matrix, [i, j], axis=1)
5      n = upd_matrix.shape[0]
6      new_matrix = np.zeros((n+1, n+1))
7
8      # Adiciona zero da diagonal
9      D_ijk = np.append(D_ijk, 0.0)
10
11     # Copia a matriz original
12     new_matrix[:n, :n] = upd_matrix
13     new_matrix[-1, :n+1] = D_ijk # ultima linha
14     new_matrix[:n+1, -1] = D_ijk # ultima coluna
15
16     return new_matrix
```

Fluxo principal de controle do algoritmo.

```
1  def run(d_matrix: np.ndarray, labels: list):
2      # 0. Inicializar nodos para cada OTU
3      tree = Newick(d_matrix, labels)
4      clusters = tree.nodes.copy()
5
6      while len(clusters) > 2:
7          # 1. Calcular u_i para cada OTU
8          u_list = compute_u(d_matrix, clusters)
9          # 2. Encontrar i e j com os menores valores
               D_ij - u_i - u_j
10         i, j = find_min_ij(d_matrix, u_list)
```

```
11          # 3. Criar novo nodo (ij) que conecta i e j
12          i_name = clusters[i].name
13          j_name = clusters[j].name
14          node_ij = tree.create_node("")
15          # "" representa label vazia
16
17          # 3.1. Calcular o tamanho da aresta entre os
                nodos i e j com o nodo (ij)
18          v_i, v_j = compute_edges(d_matrix, u_list, i,
                j)
19
20          # 4. Estimar distancias entre nodo (ij) e os k
                restantes
21          D_ijk = compute_distances(d_matrix, i, j)
```

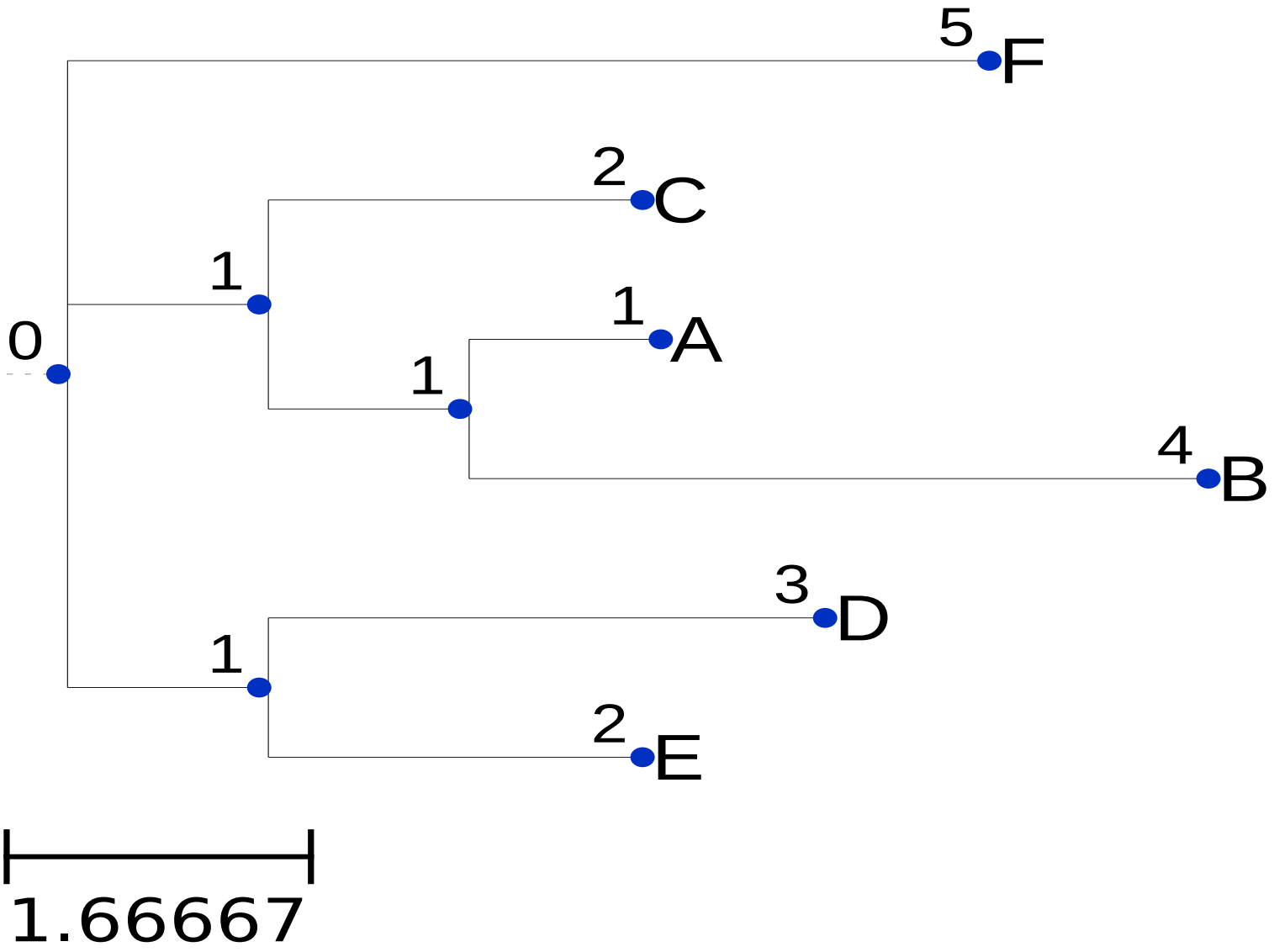
```
22          # 5. Atualizar matriz D removendo i e j, e
                adicionando D_ijk
23          d_matrix = update_distances(tree, d_matrix,
                D_ijk, i, j)
24
25          # 5.1 Atualizando  ndice  de nodos
26          node_i = clusters[i]
27          node_j = clusters[j]
28          clusters = [v for idx, v in enumerate(clusters
                ) if idx not in [i,j]]
```

```
29          # 6. Cria aresta entre (ij) e os nodos i e j
30          node_ij.add_descendant(node_i)
31          node_ij.add_descendant(node_j)
32
33          # Atualiza distancias de i e de j para (ij)
34          node_i.length = v_i
35          node_j.length = v_j
36
37          ## Adiciona nodo ij aos clusters e a arvore
38          clusters.append(node_ij)
39          tree.nodes.append(node_ij)
```

```
40      # 7. Criar nodo lm conectando os ultimos
41      node_l = clusters[0]
42      node_m = clusters[1]
43      node_m.add_descendant(node_l)
44
45      print(node_m.ascii_art())          # Viz ASCII
46      print(nw.dumps(node_ij))           # Arvore Newick
47      newick_tree = nw.dumps(node_ij)
48      t = Tree(newick_tree, format=0)
49      t.show()                           # GUI visualizacao
```

Resultados

EXEMPLO DE AULA



Leishmania é um gênero de protozoários da família *Trypanosomatidae*, que inclui os parasitas causadores das leishmanioses.
Trypanosoma cruzi é uma espécie de protozoário flagelado da família *Trypanosomatidae*. É o agente etiológico da doença de Chagas.
A seguir são apresentadas sequências no gene GD2 em espécies de Leishmania (L.) e Trypanosoma (T.)

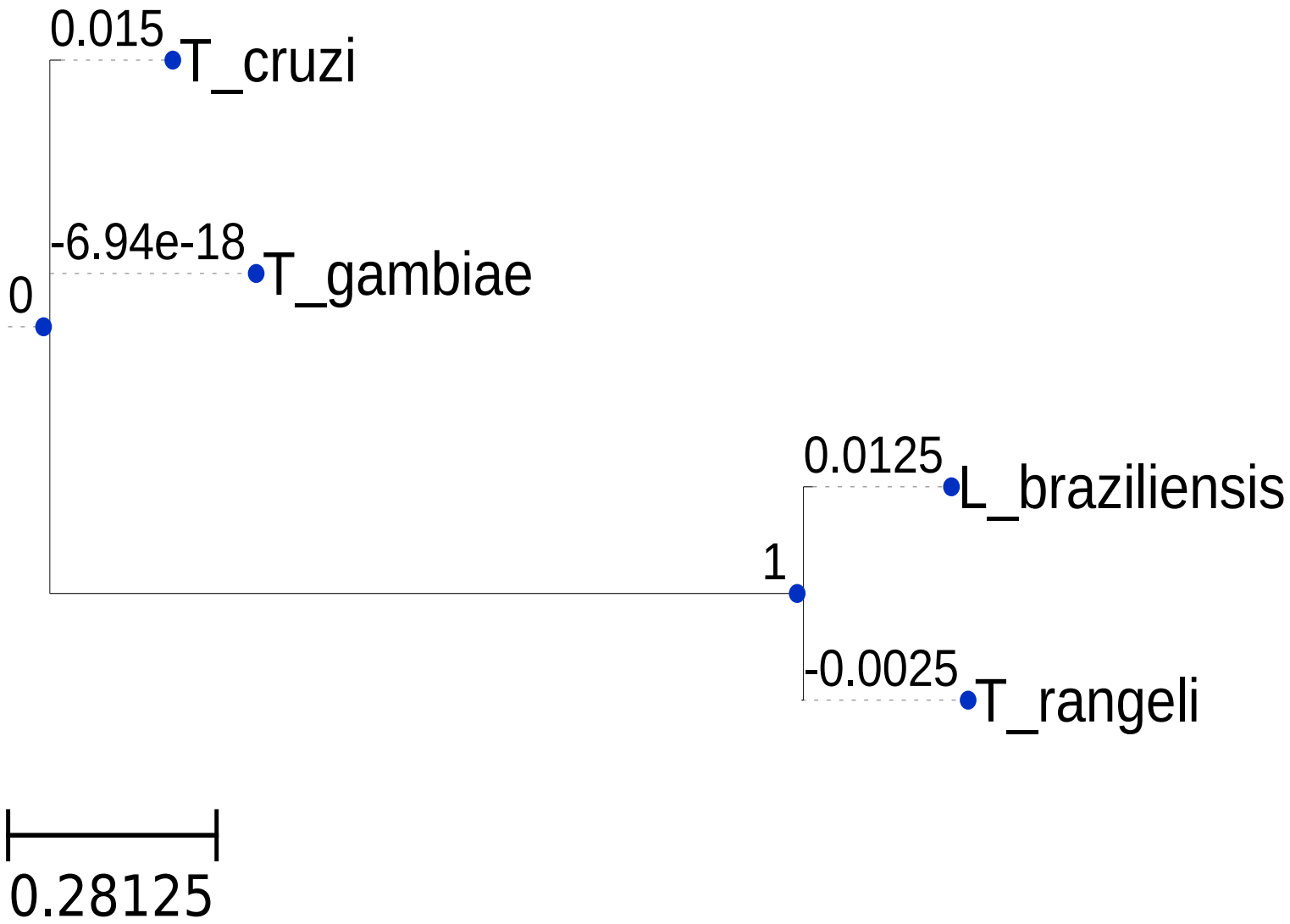
L. braziliensis	ACCTCTCTCGATCTAAATTGATAGCCTTAAATAT
T. rangeli	ACCTCCCTCGATCTAAATTGATAGCCTTAAATAT
T. cruzi	ACCTCCCTCGATCTGAATTGATAGCCTTAAACAT
T. gambiae	ACCTCCCTCGATCTGAATTGATAGCCTCGAATAT

Figura: Sequências no gene GD2.

Dada a matriz de distâncias gerada a partir da sequência de DNA sequências no gene GD2 em espécies de Leishmania (L.) e Trypanosoma (T.), implemente o algoritmo NJ (Neighbor joining) para realizar a construção da árvore filogenética das seguintes espécies:

	L. braziliensis	T. rangeli	T. cruzi	T. gambiae
L. braziliensis	0.000	0.010	0.300	0.280
T. rangeli	0.010	0.000	0.280	0.270
T. cruzi	0.300	0.280	0.000	0.015
T. gambiae	0.280	0.270	0.015	0.000

RESULTADOS DO TRABALHO III



OBRIGADO!

Leonardo Azzi Martins
Instituto de Informática — UFRGS
lamartins@inf.ufrgs.br