Trabalho II: Filogenia

INF05018 - Biologia Computacional (2025/2)

Descrição da implementação do algoritmo UPGMA.

Leonardo Azzi Martins

Instituto de Informática, UFRGS

Descrição do Algoritmo

1. Inicialização:

- Atribuir cada x_i em seu respectivo cluster C_i
- Definir uma folha por sequência, cada uma com altura 0 (não foi implementado, por simplificação)

2. Iteração:

- Encontrar dois clusters C_i e C_j tal que a distância $d_{i,j}$ na matriz de distâncias seja mínima
- Agrupar os clusters: $C_k = C_i \cup C_j$
- Adicionar um vértice conectando C_i e C_j, e adicionar na altura $d_{i,j}/2$

Descrição do Algoritmo

3. Terminação:

- Quando resta apenas um cluster.

Implementação em Python

Classe: Cluster

```
(method) def __init__(
    self: Self@Cluster,
    dist_matrix: matrix
) -> None

Cria um cluster para cada linha ou coluna da matriz de distâncias"

Args
    dist_matrix : np.matrix -> matriz de distâncias
```

Implementação em Python

Classe: Cluster

```
(method) def group_clusters(
    self: Self@Cluster,
    c_a,
    c_b
) -> None

Agrupa dois clusters Newick.

Args
    c_a : tuple -> primeiro cluster
    c_b : tuple -> segundo cluster
```

Implementação em Python

Classe: Cluster

```
(method) def complement_clusters(
    self: Self@Cluster,
    c_a,
    c_b
) -> list[int]

Retorna uma lista com os clusters complementares a c_a e c_b.

Args
    c_a : tuple -> primeiro cluster
    c_b : tuple -> segundo cluster

Returns
    list -> lista com clusters existentes diferentes de c_a e c_b.
```

Implementação em Python

```
(function) def run(d_matrix: ndarray) -> tuple
Retorna a árvore filogenética obtida com o algoritmo UPGMA.

Args
    d_matrix : np.ndarray -> matriz de distâncias

Returns
    tuple -> árvore filogenética em formato Newick (sem distâncias)
```

Implementação do algoritmo: run()

```
# 1. Inicialização
    # - Atribuir cada x_i ao seu cluster C_i
    C = Cluster(d_matrix)

    while len(C.clusters) > 1:
        old_cluster = C.clusters.copy()
```

- Cria um objeto da classe Cluster a partir da matriz de distâncias;
- Laço `while`: terminação quando resta apenas um cluster.

Implementação do algoritmo: run()

```
# 2. Iteração
# - Encontrar dois clusters C_i e C_j cujo d_ij é mínimo
d_matrix_tri = d_matrix.copy()

# Aplica matriz triangular, com diagonal e triângulo superior em np.inf,
# para que seja encontrado o valor mínimo do triângulo inferior.
d_matrix_tri[np.triu_indices_from(d_matrix_tri, k=0)] = np.inf
d_min = np.unravel_index(np.argmin(d_matrix_tri, axis=None), d_matrix.shape)

C_i, C_j = d_min # Índices do valor mínimo. Vai indicar quais clusters manipular

# - C_k = C_i U C_j
C.group_clusters(C.clusters[C_i], C.clusters[C_j])
```

- Encontra clusters onde a distância é mínima, obtendo seus índices na matriz
- Agrupa os clusters

Implementação do algoritmo: run()

```
# Criar cópia da matriz original, deletando C_i e C_j
d_matrix_new = d_matrix.copy()
d_matrix_new = np.delete(d_matrix_new, C_i, axis=0) # Deletar linhas C_i e C_j
d_matrix_new = np.delete(d_matrix_new, C_j, axis=0)
d_matrix_new = np.delete(d_matrix_new, C_i, axis=1) # Deletar colunas C_i e C_j
d_matrix_new = np.delete(d_matrix_new, C_j, axis=1)

# Adicionar uma coluna e linha C_k
d_matrix_new = np.c_[np.ones(d_matrix_new.shape[0]) * np.inf, d_matrix_new]
d_matrix_new = np.r_[[np.ones(d_matrix_new.shape[1]) * np.inf], d_matrix_new]
d_matrix_new[0][0] = 0 # mantém diagonal zerada
```

- Manipulação da matriz de distâncias:
  - Cria uma matriz auxiliar
  - Deleta as linhas e colunas relativas a C_i e C_j na matriz auxiliar
  - Adiciona uma nova linha e coluna para C_k na matriz auxiliar

Implementação do algoritmo: run()

```
# Obtém clusters "complementares", ou seja, que não são C_i ou C_j
C_comp = C.complement_clusters(C.clusters[C_i], C.clusters[C_j])


for c_alvo in C_comp:
    # Busca distâncias na matriz original entre C_i e C_j com outros clusters c_k
    c_k = old_cluster.index(c_alvo) # Obtém índice do cluster na matriz original
    d_ik = d_matrix[C_i][c_k]
    d_jk = d_matrix[C_j][c_k]
    d_mk = (d_ik + d_jk) / 2.0

    # Atualiza distâncias na nova matriz
    c_k = C.clusters.index(c_alvo) # Obtém índice do cluster na matriz nova

    # - Atualizar matriz com d_kz = (d_iz + d_jz) / 2
    d_matrix_new[0][c_k] = d_mk
    d_matrix_new[c_k][0] = d_mk
```

- Computa novas distâncias para o cluster C_k

Implementação do algoritmo: Fórmula de Atualização

Cálculo da Nova Distância:

- Fórmula:

$$d_{km} = \frac{d_{im} + d_{jm}}{2}$$

Onde:

$$d_{im}: \text{ Distancia entre o cluster } i \text{ e um cluster } m.$$
$$d_{jm}: \text{ Distancia entre o cluster } j \text{ e um cluster } m.$$

Implementação do algoritmo: run()

```
    # Atualiza a matriz principal
    d_matrix = d_matrix_new

# 3. Terminação
# - Quando restar apenas um cluster C_i
return C.clusters[0]
```

- Atualiza a matriz de distâncias
- Se o número de clusters for > 1, continua no laço. Se não, termina e retorna a tupla final em formato Newick simplificado.

Implementação do algoritmo: simplificações

- Demorei um pouco até conseguir traduzir as demonstrações de aulas e pseucodódigos em estruturas de dados que fizessem sentido. Por isso, construí a implementação com várias simplificações:
    - As "labels" de cada OTU são seus índices na matriz de distâncias. Ficou assim pois estava pensando muito na manipulação dos índices na matriz após os agrupamentos. Esta foi a parte mais complicada para abstrair e implementar.
    - Por isto, minha representação em Newick considera as OTUs como estes índices numéricos (depentende da forma como a matriz de entrada foi gerada), e sem considerar as distâncias. Pois utilizei listas com as tuplas para controlar os índices da matriz. Não tive tempo pra formatar isso como uma string Newick completa.

Representação das árvores no formato Newick

- Representação da árvore filogenética como tuplas, sem representação de distâncias.

Exemplo:

$$((A, B, (C, D));$$

onde A, B, C, D são índices para linhas ou colunas da matriz de distâncias, relativas a sequências.

Fim!