Métodos Computacionais Aplicados à Biocomplexidade

Aluno: André Gustavo Dessoy Hubner - Matrícula: 00315569
IF-UFRGS

16 de agosto de 2024

1 Introdução

Esta tarefa tem como objetivo estudar a criação de uma árvore filogenética a partir de alinhamentos múltiplos e uma implementação do algoritmo de Kruskal a partir de um programa construído em Python e com a biblioteca Biopython.

2 Metodologia

2.1 Busca de genes de rRNA 16S

O primeiro passo na construção da árvore foi buscar por genes de RNA ribossômico 16S de pelo menos 10 espécies devido ao seu já bem estabelecido potencial em explicitar as relações filogenéticas entre os organismos com base na evolução de sua sequência. Esta busca foi construída de forma semelhante a tarefas passadas, utilizando o módulo "Entrez"do Biopython para primeiro realizar a busca como seria feito no site do NCBI com a função "esearch"e depois conseguir as sequências de fato com "efetch".

Mais detalhadamente, na primeira função buscamos no banco de dados "nucleotide"pelo termo "16S ribosomal RNA gene[Titl] NOT partial sequence[Titl] NOT uncultured[Titl] NOT clone[Titl]", que essencialmente retornará sequências quaisquer de gene de RNA ribossômico 16S que não sejam alguns tipos de sequência incompleta, obtendo assim sequências de boa qualidade, e retornando 100 resultados ao especificar o parâmetro "retmax"com o valor de 100.

Com os resultados dessa busca obteve-se então uma lista dos ids dos resultados, que para adicionar um elemento de aleatoriedade, foi submetida ao método "sample"da biblioteca "random". Esta função irá fazer, bem como diz seu nome, uma amostragem da lista passada de acordo com o valor k passado, que foi escolhido como 10 aqui. Esta lista amostrada é então passada ao efetch, buscando também em nucleotide e especificando o tipo de retorno como "fasta", retornando finalmente as sequências desejadas. Após um leve uso de manipulação de strings básica de Python, elas estão prontas para serem alinhadas entre si.

2.2 Alinhamento múltiplo e matriz de distância

Para o alinhamento múltiplo global, foi utilizado o módulo Align do Biopython. Embora tenha sido instruída a utilização do módulo pairwise2, esta dependência está marcada como obsoleta pela própria equipe do Biopython em sua documentação, e sua usabilidade é bem

pior do que a das funcionalidades usadas aqui. Ainda assim, tentou-se primeiramente ela, no entanto ficou claro rapidamente que suas funções de alinhamento necessariamente tomavam um tempo muito grande e pareciam alinhar várias vezes para apenas um alinhamento específicado, retornando o mesmo score para todos. A utilização do módulo Align pareceu não só melhor como ter resultados mais confiáveis, depois de verificar ambas as opções.

Apenas duas funcionalidades do Align foram necessárias aqui, sendo estas a classe "PairwiseAligner"e o método score() de objetos dessa classe. Através do primeiro, pode-se criar um objeto para realizar alinhamentos com grande capacidade e facilidade de customização, podendo especificar os valores de score a serem atribuídos a virtualmente cada situação. Neste projeto foi criada uma instância utilizando valores de match como 5, mismatch como -4, gap -10 e extensão de gap -1, além de especificar a realização de alinhamentos globais. Assim, utilizando dois laços de repetição, um aninhado no outro, foi possível usar o score() para realizar o alinhamento entre cada sequência das 10 anteriores, retornando apenas a informação que seria utilizada posteriormente, o score de cada alinhamento.

Esses scores, cada um informando a pontuação do alinhamento de uma das sequências com outra, foram armazenados em uma matriz de pontos utilizando listas de Python. Para transformar esta matriz em uma matriz de distâncias, precisava-se ainda aplicar um processamento para cada valor, de acordo com as seguintes regras:

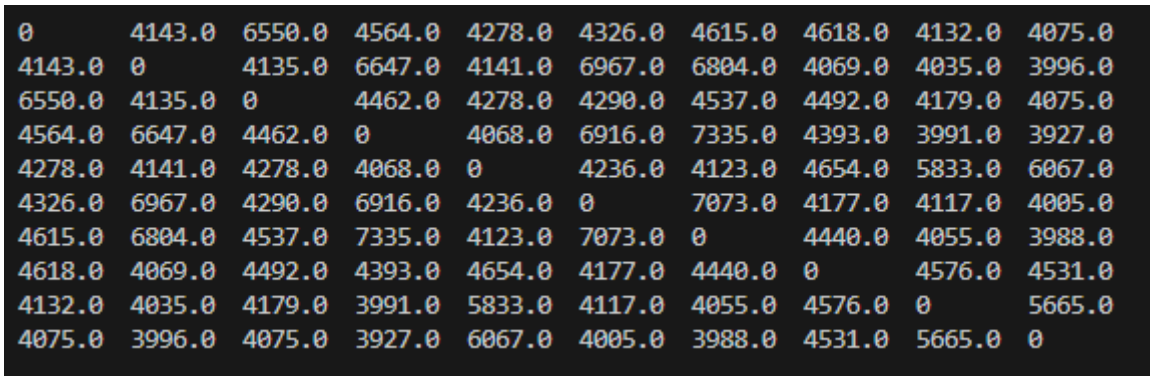
- Se ele fosse igual a zero, significava que era um alinhamento entre as mesmas sequências (nas iterações anteriores se aplicou a lógica para retornar 0 caso fossem as mesmas sequências), e então se retorna zero.
- Se fosse menor que zero, significava um score negativo (ou seja, fraco), portanto seria necessário convertê-lo a um valor grande, representando uma grande distância. Isto foi feito simplesmente invertendo o sinal do resultado.
- Não sendo nenhum dos dois, eleva-se o valor a -1, resultando em números positivos tão menores quanto o valor original era grande.

Foi feita ainda uma função simples para visualizar as duas matrizes geradas. As figuras 1 e 2 mostram esse resultado para as matrizes de pontos e de distância, respectivamente. As matrizes expostas não são dos mesmos dados usados para figuras posteriores devido à aleatoriedade garantida por execução do programa, no entanto são representativas do padrão de sequências que são obtidas.

2.3 Construção da árvore em formato Newick

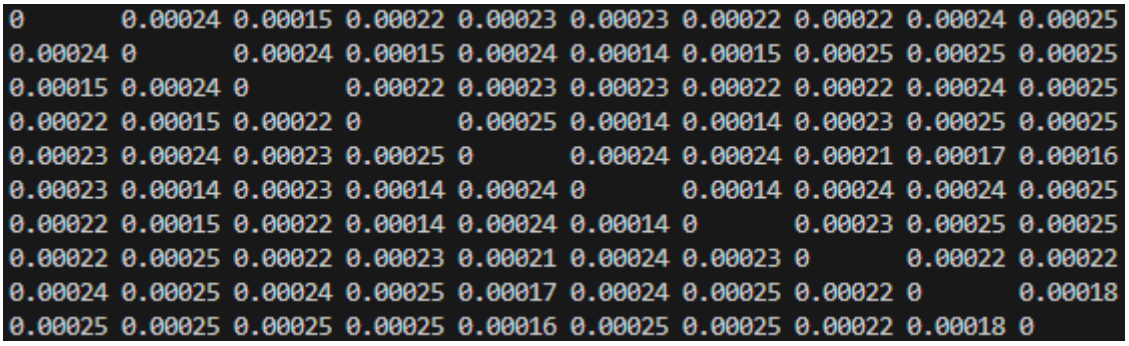
Construída a matriz de distâncias, já se havia todos os dados necessários para construir a árvore filogenética. Já tinha sido mencionado que isto seria feito implementando o algoritmo de Kruskal, entretanto para realizar cada inserção dessa árvore, seria necessário realizá-la em uma string que pudesse ser representada como uma árvore filogenética. Para isso serve o padrão de representação Newick, que sendo o modelo mais simples e também um dos mais utilizados para representar árvores filogenéticas, foi o padrão escolhido nesta ocasião.

A lógica começa na função BuildTree, que recebe a matriz de distâncias e uma lista de códigos dos organismos alvos e retorna a representação em string da árvore gerada. Após a inicialização de variáveis que serão utilizadas ao longa da execução, já será realizada a primeira varredura na matriz de distâncias buscando pelo menor valor. É importante notar



0	4143.0	6550.0	4564.0	4278.0	4326.0	4615.0	4618.0	4132.0	4075.0
4143.0	0	4135.0	6647.0	4141.0	6967.0	6804.0	4069.0	4035.0	3996.0
6550.0	4135.0	0	4462.0	4278.0	4290.0	4537.0	4492.0	4179.0	4075.0
4564.0	6647.0	4462.0	0	4068.0	6916.0	7335.0	4393.0	3991.0	3927.0
4278.0	4141.0	4278.0	4068.0	0	4236.0	4123.0	4654.0	5833.0	6067.0
4326.0	6967.0	4290.0	6916.0	4236.0	0	7073.0	4177.0	4117.0	4005.0
4615.0	6804.0	4537.0	7335.0	4123.0	7073.0	0	4440.0	4055.0	3988.0
4618.0	4069.0	4492.0	4393.0	4654.0	4177.0	4440.0	0	4576.0	4531.0
4132.0	4035.0	4179.0	3991.0	5833.0	4117.0	4055.0	4576.0	0	5665.0
4075.0	3996.0	4075.0	3927.0	6067.0	4005.0	3988.0	4531.0	5665.0	0

Figura 1: Matriz de pontos obtida a partir de uma rodada de alinhamento múltiplo global entre 10 sequências aleatórias de genes de RNA ribossômico.



0	0.00024	0.00015	0.00022	0.00023	0.00023	0.00022	0.00022	0.00024	0.00025
0.00024	0	0.00024	0.00015	0.00024	0.00014	0.00015	0.00025	0.00025	0.00025
0.00015	0.00024	0	0.00022	0.00023	0.00023	0.00022	0.00022	0.00024	0.00025
0.00022	0.00015	0.00022	0	0.00025	0.00014	0.00014	0.00023	0.00025	0.00025
0.00023	0.00024	0.00023	0.00025	0	0.00024	0.00024	0.00021	0.00017	0.00016
0.00023	0.00014	0.00023	0.00014	0.00024	0	0.00014	0.00024	0.00024	0.00025
0.00022	0.00015	0.00022	0.00014	0.00024	0.00014	0	0.00023	0.00025	0.00025
0.00022	0.00025	0.00022	0.00023	0.00021	0.00024	0.00023	0	0.00022	0.00022
0.00024	0.00025	0.00024	0.00025	0.00017	0.00024	0.00025	0.00022	0	0.00018
0.00025	0.00025	0.00025	0.00025	0.00016	0.00025	0.00025	0.00022	0.00018	0

Figura 2: Matriz de distâncias obtida a partir de um processamento adicional da matriz anterior.

aqui que embora a matriz gerada anteriormente seja simétrica, a busca será realizada em apenas um dos lados simétricos. Assim, após uma varredura total desse lado o menor valor encontrado será usado para unir as sequências dessa posição na matriz na etapa seguinte. A lógica desta execução está na função SearchMinimum, que retorna não só esse valor como os dois índices representando a posição das sequências da linha e da coluna, respectivamente. Ela está representada na figura 3.

```python
def SearchMinimum(distMatrix: list):
    currentMinimum: int = 99999
    currentMinimumIndexes = []
    #i=linhas, j=colunas, k=alvo
    i=0
    for k in range(1,10):
        for j in range(k):
            if distMatrix[i][j] != 0 and distMatrix[i][j] < currentMinimum:
                currentMinimum = distMatrix[i][j]
                currentMinimumIndexes = [i, j]
        i=i+1
    #Última iteração
    for j in range(k):
        if distMatrix[i][j] != 0 and distMatrix[i][j] < currentMinimum:
            currentMinimum = distMatrix[i][j]
            currentMinimumIndexes = [i, j]


    return currentMinimum, currentMinimumIndexes
```

Figura 3: Função que performa uma busca inteira por um dos lados da matriz pelo valor mínimo, retornando-o junto dos índices das sequências da linha e da coluna da posição.

Logo após a execução dessa função a posição na qual foi obtida esse mínimo é modificada com o valor 99999, representando que aquela posição já foi usada. Inclusive antes mesmo desta modificação, é performada uma checagem de se o valor encontrado é igual a esse número, o que seria um dos pontos de retorno possíveis da árvore completa gerada. A figura 4 mostra a primeira execução dessa lógica.

Como é possível ver, temos as seguintes variáveis, junto de suas funções:

- tree: é a árvore sendo construída, crescendo a cada iteração.
- minimumIndicesDict: serve para ter controle de quais índices (sequências) já foram adicionadas à árvore.
- currentMinimum: receberá o valor mínimo encontrado em SeachMinimum; inicializada com 100000 para que qualquer valor encontrado seja menor.
- currentMinimumIndexes: índices dos mínimos encontrados na última execução de SearchMinimum; são adicionados em minimumIndexesDict logo após essa execução.
- secondaryBranchesSet: representa um conjunto de ramos (partes da árvore) que não fazem parte do ramo principal ainda, e que terão uma lógica de inserção diferente quando forem inseridos posteriormente.

```python
def BuildTree(distMatrix: list, organismsCodes: list) -> tuple[str, dict]:
    tree:str = ""
    minimumIndexesdict:dict = dict()
    currentMinimum:int = 100000
    currentMinimumIndexes:list = []
    secondaryBranchsSet:list = list()

    #Faz primeira iteração fora do loop pois é previsível, e para não atribuir a um galho secundário
    currentMinimum, currentMinimumIndexes = SearchMinimum(distMatrix)

    distMatrix[currentMinimumIndexes[0]][currentMinimumIndexes[1]] = 99999
    tree = tree+'('+organismsCodes[currentMinimumIndexes[0]]+','+organismsCodes[currentMinimumIndexes[1]]+'),'

    minimumIndexesdict[currentMinimumIndexes[0]] = organismsCodes[currentMinimumIndexes[0]]
    minimumIndexesdict[currentMinimumIndexes[1]] = organismsCodes[currentMinimumIndexes[1]]

    counter=2
    while currentMinimum != 99999:
        firstIndexIsRepeated=False
        secondIndexIsRepeated=False
```

Figura 4: Função que performa uma busca inteira por um dos lados da matriz pelo valor mínimo, retornando-o junto dos índices das sequências da linha e da coluna da posição.

O ramo secundário mencionado nesta última variável é uma limitação da abordagem utilizada aqui que precisa ser tratada, mas para entender sua função, primeiro é necessário saber mais da implementação do algoritmo usada aqui. De acordo com o algoritmo de Kruskal, cada vez que vai ser adicionado um dos nós representando o alinhamento de dois organismos à árvore, deve-se saber se nenhum ou um deles já está inserido na árvore. Se um já está, simplesmente formamos um clado entre o organismo que não está e todo o clado que contém o outro organismo que já estava na árvore. Entretanto, se ambos não estavam na árvore, forma-se primeiro apenas um clado entre esses dois.

Como se sabe que todos os clados formados serão imediatamente ou eventualmente inseridos em um clado "principal"e que a lógica de inserção desse último caso é diferente (são dois ou mais organismos sendo inseridos na árvore principal ao invés de um), primeiro é realizada uma iteração fora do laço de repetição para definir uma "árvore principal", e posteriormente todas as vezes que clados são formados desta forma eles são tratados como "galhos secundários", que apesar de serem imediatamente adicionados à árvore através do controle de "minimumIndicesDict", precisam de um controle adicional através de "secondaryBranchesSet"para realizar a lógica apropriada de inserção à árvore principal quando os índices do mínimo encontrado representarem um organismo já na árvore principal e outro em um "galho"secundário.

A primeira execução mencionada é o que vemos na figura 4. A partir de então, começa um laço de repetição que só quebrará ao todos os organismos serem inseridos na árvore e nenhuma árvore secundária sobrar, ou o valor mínimo encontrado ser igual a 99999, o valor usado para descartar posições da matriz. Este último critério por sua vez nunca deve ser alcançado, uma vez que em toda iteração ocorre a inserção apropriada, e se não são atingidos os critérios para isso, verifica-se primeiro se o tamanho do dicionário de índices contém o mesmo número de elementos que o número de linhas na matriz e não há nada restando em galhos secundários, conforme explicado anteriormente, retornando então a árvore finalizada e terminando a execução de BuildTree caso essas condições sejam verdadeiras. O código com a lógica para qual tipo de modificação aplicar está demonstrado nas figuras 5 e 6.

Conforme é possível ver nas figuras, os códigos para inserir um galho secundário à árvore

```
counter=2
while currentMinimum != 99999:
    firstIndexIsRepeated=False
    secondIndexIsRepeated=False

    currentMinimum, currentMinimumIndexes = SearchMinimum(distMatrix)
    if(currentMinimum) == 99999:
        return tree
    distMatrix[currentMinimumIndexes[0]][currentMinimumIndexes[1]] = 99999

    if currentMinimumIndexes[0] in minimumIndexesdict:
        firstIndexIsRepeated=True

    if currentMinimumIndexes[1] in minimumIndexesdict:
        secondIndexIsRepeated=True

    if firstIndexIsRepeated and secondIndexIsRepeated:
        for item in secondaryBranchsSet:
            for index in item:
                if index == currentMinimumIndexes[0]:
                    tree = ConnectBranches(tree, currentMinimumIndexes[0], currentMinimumIndexes[1])
                    secondaryBranchsSet.remove(item)
                if index == currentMinimumIndexes[1]:
                    tree = ConnectBranches(tree, currentMinimumIndexes[1], currentMinimumIndexes[0])
                    secondaryBranchsSet.remove(item)

        if len(minimumIndexesdict) == len(distMatrix) and len(secondaryBranchsSet) == 0:
            if tree[-1] == ',':
                tree = tree[:-1]
            tree = tree+';'
            return tree, minimumIndexesdict
        else:
            continue
```

Figura 5: Primeira metade do código dentro do laço de repetição, contendo a lógica para verificar se vai haver uma inserção de um galho secundário na árvore principal e se a árvore está terminada.

```
        if len(minimumIndexesdict) == len(distMatrix) and len(secondaryBranchsSet) == 0:
            if tree[-1] == ',':
                tree = tree[:-1]
            tree = tree+';'
            return tree, minimumIndexesdict
        else:
            continue

    elif not firstIndexIsRepeated and not secondIndexIsRepeated:
        tree = tree+'('+organismsCodes[currentMinimumIndexes[0]]+','+organismsCodes[currentMinimumIndexes
        minimumIndexesdict[currentMinimumIndexes[0]] = organismsCodes[currentMinimumIndexes[0]]
        minimumIndexesdict[currentMinimumIndexes[1]] = organismsCodes[currentMinimumIndexes[1]]

        secondaryBranchsSet.append([max(minimumIndexesdict), min(minimumIndexesdict)])

    elif (firstIndexIsRepeated and not secondIndexIsRepeated):
        tree, minimumIndexesdict = ModifyTree(tree, minimumIndexesdict, currentMinimumIndexes[0], current


    elif (not firstIndexIsRepeated and secondIndexIsRepeated):
        tree, minimumIndexesdict = ModifyTree(tree, minimumIndexesdict, currentMinimumIndexes[1], current


    if firstIndexIsRepeated or secondIndexIsRepeated:
        counter+=1
    else:
        counter+=2

return tree
```

Figura 6: Segunda metade do código dentro do laço de repetição, contendo primero a condição de inserir uma galho secundário e depois os dois casos de inserção de um organismo à árvore principal.

principal e apenas um organismo a ela estão encapsulado pelos métodos "ConnectBranches"e "ModifyTree", respectivamente, que foram omitidos aqui por envolverem muito mais questões técnicas do que algo que já não foi explicado do funcionamento do algoritmo. Chegado o fim da função, temos por fim uma string representando uma árvore em formato Newick e seguindo a distância filogenética estimada de acordo com o algoritmo de Kruskal.

3 Resultados

3.1 Árvore gerada a partir do algoritmo e base de comparação

Ainda dentro do código, foram utilizadas duas funções do módulo Phylo do Biopython para validar e visualizar a árvore gerada, sendo estas a "draw_ascii"e a "draw". No entanto, como ambas esperam um argumento no formato de árvore do mesmo módulo, a string da árvore foi primeiro convertida para um objeto StringIO através de conversão explícita direta, e logo depois para uma árvore do Biopython com Phylo.read(), passando a árvore e o formato "newick"como argumentos. A partir da variável resultante foi possível visualizar a árvore com ambas as funções draw, e o resultado da mais detalhada está na figura 7.

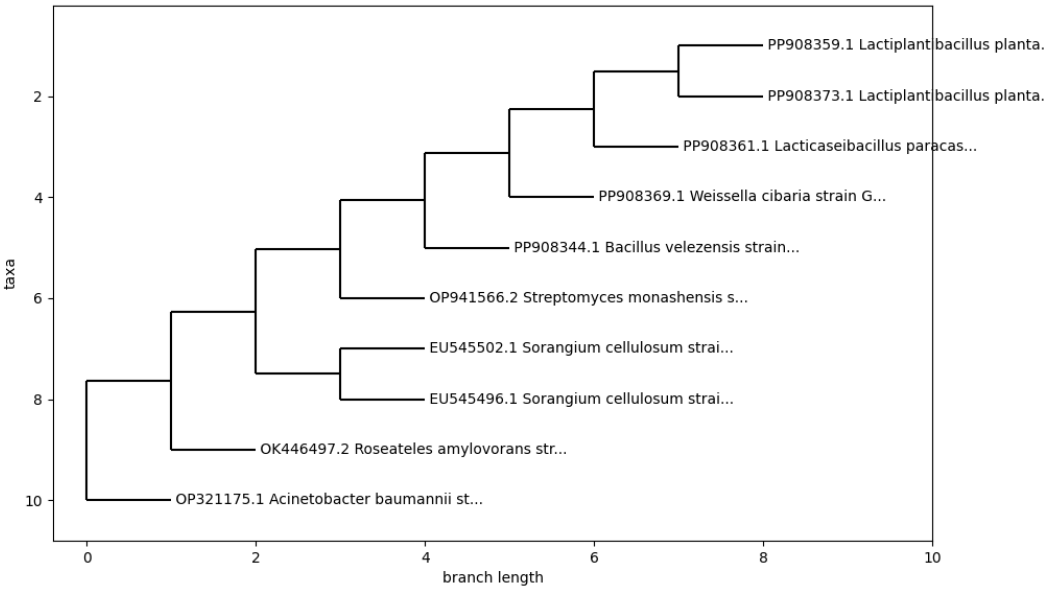


Figura 7: Árvore filogenética resultante do algoritmo aplicado e de sua visualização através do método draw.

Nota-se que a visualização tem uma qualidade limitada, porém, na própria documentação do módulo há poucos detalhes sobre a configuração da chamada ao método draw. Como se sabe que essa função ,na verdade, usa funcionalidades da biblioteca Matplotlib, foi tentada uma configuração direta com esta, o que não foi bem sucedida. Apesar disso, a árvore gerada claramente tem a estrutura de uma árvore filogenética, assim como apresenta organismos de nomes semelhantes, classificados mais proximamentes na taxonomia, mais próximos um dos outros, o que é um indício visual claro de um bom resultado gerado.

Todavia, para fazer reflexões relevantes sobre se a árvore gerada com este algoritmo é válida ou não, primeiro é necessário termos uma fonte de informação confiável sobre a

distância filogenética dos organismos. Sendo assim, a partir de uma das execuções do código, foram coletados todos os códigos das sequências utilizadas na mesma execução, os quais foram então usados para gerar uma árvore filogenética com a ferramenta de árvore comum da seção de taxonomia do NCBI. Essa ferramenta estima as árvores puramente conforme as distâncias das classicações dos organismos, tal como informa sua página "help", logo, não subsitui análises filogenéticas minuciosas, mas usa uma base suficientemente credível.

Ademais, como não é possível selecionar mais de um organismo de um mesmo tipo e há um número limitado de linhagens selecionáveis, a árvore gerada contém 10-N nós, em que N é o número de organismos da mesma espécie que vieram no resultado (aqui houve uma repetição de *Sorangium cellulosum* e de *Lactiplantibacillus plantarum*). A árvore resultante está disponível na figura 8.

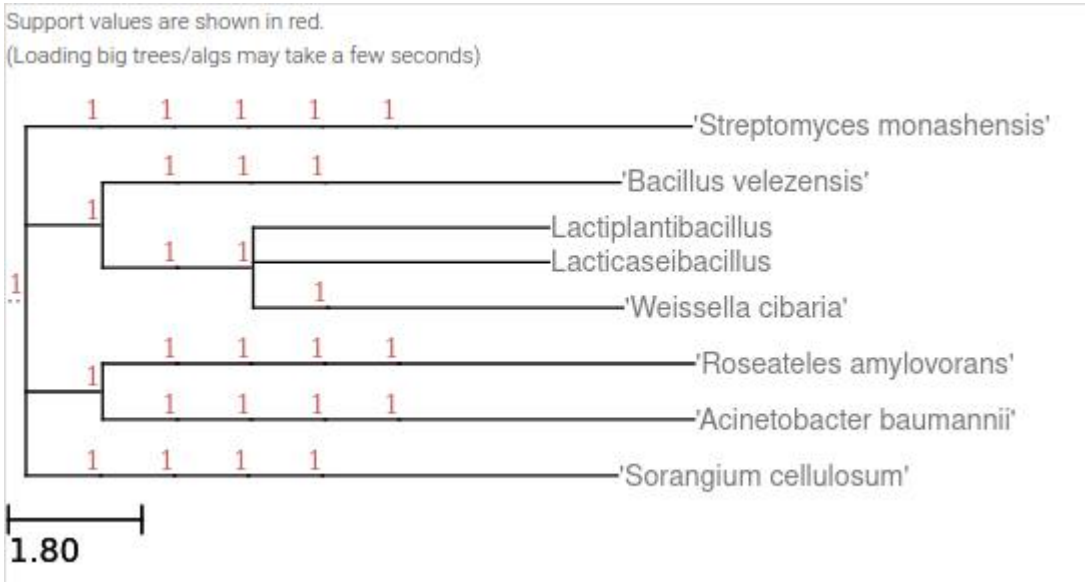


Figura 8: Árvore obtida ao informar ao passar os mesmos organismos da execução do algoritmo de Kruskal à visualização de árvore comum do NCBI. Nota-se que há 8 nodos terminais ao invés de 10, devido a execução ter sorteado 10 organismos havendo duas repetições de espécie

3.2 Comparação entre as duas árvores

Observando as duas árvores e considerando os nodos de *Lactiplantibacillus plantarum* e de *Sorangium cellulosum* na árvore externa como sendo na verdade um clado contendo as duas linhagens da mesma espécie conforme os dados da execução, são bastante notáveis alguns pontos positivos na árvore construída. Primeiramente, o caso mais óbvio é que tanto as repetições de *Lactiplantibacillus plantarum* quanto de *Sorangium cellulosum* estão mais próximas da sequência da outra linhagem da mesma espécie do que qualquer outra. O segundo ponto que é extremamente positivo é o fato de os dois *Lactiplantibacillus plantarum*, o *Lacticaseibacillus paracasei* e o *Weissella cibaria*, todos estes pertencentes a gêneros da família *Lactobacillaceae*, estarem mais próximos de si do que qualquer outro. E não só isso, como *Bacillus velezensis*, que também é um bacilo, está mais próximo desses do que de outros. *Roseateles amylovorans* e *Acinetobacter baumannii*, pertencentes ao mesmo filo Pseudomonadota, também estão próximos.

Por outro lado, o principal ponto negativo é o fato de que vários desses organismos deveriam estar no mesmo nível de hierarquia, conforme a árvore base. O clado representando todos os bacilos, por exemplo, está uma hierarquia abaixo de *Streptomyces monashensis* e várias em relação aos outros, quando, na verdade, deveria estar no mesmo nível da árvore de acordo com a base (por todos serem bactérias). Outro caso disso é o fato de *Roseateles amylovarans* estar um nível abaixo de *Acinetobacter baumannii*, quando claramente ambas deveriam estar no mesmo nível, por ambos terem o mesmo filo em comum. Esta é sem dúvidas a principal limitação da abordagem utilizada aqui, o que limitaria bastante a utilização do algoritmo para gerar árvores úteis em maiores cenários. Um outro ponto negativo, de grau bem menor, porém, é que claramente o algoritmo utilizado não leva em conta as distâncias evolutivas para formar o tamanho dos galhos assim como é feito na árvore gerada pelo NCBI taxonomy, o que levaria um maior refinamento para implementar.

Resolvido o problema dos níveis da árvore, contudo, pelo menos com os dados obtidos aqui e com a observação rápida de outras execuções, sería um recurso muito útil para gerar árvores com o intuito de análisar as relações filogenéticas entre os organismos de uma forma mais geral, agrupando organismos semelhantes geneticamente mais perto um do outro através de suas sequências de gene de RNA 16S. Na implementação atual já é bem clara, após uma observação maior da árvore, o padrão com que ele forma as relações entre os organismos mais próximos, podendo ser útil para certas investigações mais rápidas.