

Exercício 6.30 (Papadimitriou)

Alice Duarte Scarpa, Bruno Lucian Costa

2015-06-23

1 Enunciado

Reconstruindo árvores filogenéticas pelo método da máxima parcimônia

Uma árvore filogenética é uma árvore em que as folhas são espécies diferentes, cuja raiz é o ancestral comum de tais espécies e cujos galhos representam eventos de especiação.

Queremos achar:

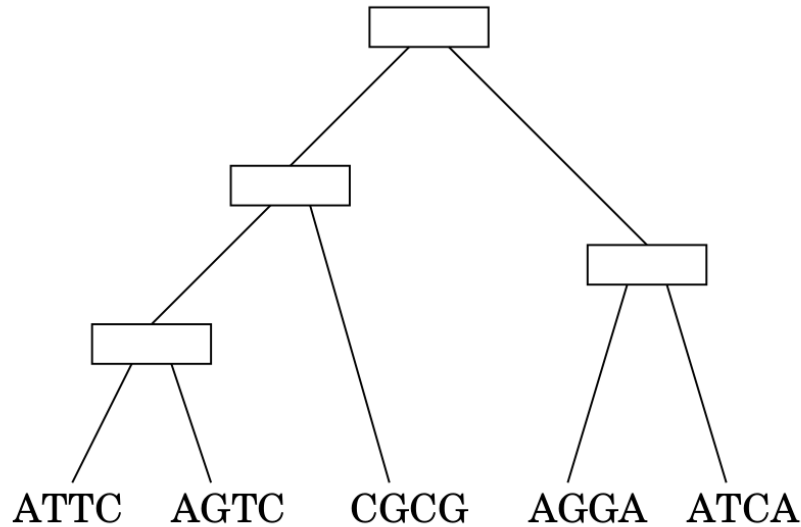
- Uma árvore (binária) evolucionária com as espécies dadas
- Para cada nó interno uma string de comprimento k com a sequência genética daquele ancestral.

Dada uma árvore acompanhada de uma string $s(u) \in \{A, C, G, T\}^k$ para cada nó $u \in V(T)$, podemos atribuir uma nota usando o método da máxima parcimônia, que diz que menos mutações são mais prováveis:

$$\text{nota}(T) = \sum_{(u,v) \in E(T)} (\text{número de posições em que } s(u) \text{ e } s(v) \text{ diferem}).$$

Achar a árvore com nota mais baixa é um problema difícil. Aqui vamos considerar um problema menor: Dada a estrutura da árvore, achar as sequências genéticas $s(u)$ para os nós internos que dêem a nota mais baixa.

Um exemplo com $k = 4$ e $n = 5$:



1. Ache uma reconstrução para o exemplo seguindo o método da máxima parcimônia.
2. Dê um algoritmo eficiente para essa tarefa.

2 Solução

A nota final de uma árvore é a soma da nota de cada letra. Podemos calcular a resposta para cada letra independentemente e depois concatenar as respostas para obter a árvore final.

Nós vamos usar um algoritmo de programação dinâmica para encontrar o valor das folhas intermediárias em uma árvore P em que cada folha tem valor A, G, T ou C.

Vamos representar a nossa árvore como um objeto:

```

class Arvore:
    def __init__(self, pai):
        self.filhos = []
        self.valor = ""
        self.pai = pai
  
```

Vamos computar $melhor_nota[v, \ell]$ como a melhor maneira de preencher os nós da sub-árvore enraizada em v , dado que o pai de v tem valor ℓ .

Também preencheremos $melhor_letra[v, \ell]$ com um valor possível de uma configuração otimal. Guardaremos tais valores em dicionários.

```
melhor_nota = {}
melhor_letra = {}
```

Vamos computar $melhor_nota$ de baixo para cima. Então, o caso base para esse algoritmo é a resposta para as folhas, isto é, $melhor_nota[folha][\ell]$.

Uma sub-árvore que contém apenas uma folha e seu pai vai ter nota 0 se a folha e o pai tiverem ambos o mesmo valor (A, G, T ou C) ou nota 1 se os dois tiverem valores diferentes:

$$melhor_nota[folha][\ell] = \begin{cases} 0 & \text{se } folha.valor = \ell \\ 1 & \text{caso contrário} \end{cases}$$

Além disso, não temos escolha para o valor otimal:

$$melhor_valor[folha][\ell] = folha.valor$$

Tendo o caso base, podemos computar $melhor_nota[v][\ell]$ assumindo que $melhor_nota[w][\ell]$ já foi computado para todo w filho de v e $\ell \in \{A, G, T, C\}$.

Dado que o pai de v tem valor ℓ , a melhor nota para a sub-árvore enraizada em v quando o valor de v é igual a m é:

$$[\ell \neq m] + \sum_{w \text{ filho de } v} melhor_nota[w][m]$$

Onde

$$[\ell \neq m] = \begin{cases} 0 & \text{se } m = \ell \\ 1 & \text{caso contrário} \end{cases}$$

Queremos escolher um valor $m \in \{A, G, T, C\}$ para v que minimize a nota final da sub-árvore. Então:

$$melhor_nota[v][\ell] = \min_{m \in \{A, G, T, C\}} \left([\ell \neq m] + \sum_{w \text{ filho de } v} melhor_nota[w][m] \right)$$

e $melhor_letra[v][\ell]$ é um valor de m que atinge o mínimo acima.

Implementando o que descrevemos recursivamente, obtemos a seguinte função:

```

def calcula_melhor_nota(v, l):
    if not v.filhos:
        melhor_nota[v, l] = 1 if l != v.valor else 0
        melhor_letra[v, l] = v.valor
        return melhor_nota[v, l]

    melhor_nota[v, l] = 100000

    for m in ['A', 'G', 'T', 'C']:
        nota_atual = sum(calcula_melhor_nota(w, m) for w in v.filhos)
        if m != l:
            nota_atual += 1

        if nota_atual < melhor_nota[v, l]:
            melhor_nota[v, l] = nota_atual
            melhor_letra[v, l] = m

    return melhor_nota[v, l]

```

Sabendo calcular $melhor_nota[v][\ell]$ para todos os vértices exceto a raiz podemos encontrar a nota da árvore como o mínimo entre os possíveis valores para a raiz:

$$\min_{\ell \in \{A, G, T, C\}} \sum_{v \text{ filho da raiz}} melhor_nota[v][\ell]$$

Um valor ótimo para a raiz é um valor de ℓ para o qual o mínimo acima é atingido. Preencheremos $raiz.label$ com tal valor, como abaixo:

```

melhor_nota_raiz = 100000
for l in ['A', 'G', 'T', 'C']:
    nota_atual_raiz = sum(calcula_melhor_nota(w, m) for w in v.filhos)

    if nota_atual_raiz < melhor_nota_raiz:
        raiz.valor = l
        melhor_custo_raiz = nota_atual_raiz

```

Tendo o valor da raiz já determinado e os valores de $melhor_letra$, podemos preencher os valores dos nós internos:

```

def preenche_dado_pai(v):
    v.valor = melhor_letra[v, v.pai.valor]
    for w in v.filhos:

```

```

        preenche_dado_pai(w)

for w in raiz.filhos:
    preenche_dado_pai(w)

```

3 Rodando o algoritmo

3.1 Formato Newick

Um formato muito usado para árvores em bioinformática é o formato Newick. Assim como as *s-expressions* do LISP, ele usa o fato de que parênteses podem ser usados para especificar uma árvore.

TODO: especificar o formato, referência do formato

3.1.1 Parseando o formato Newick

O primeiro passo é notar que (gato, rato) é equivalente a (gato)(rato), então podemos transformar uma estrutura com vírgulas em uma estrutura que só contém parênteses.

TODO: explicar o código

```

def parseia_newick(string):
    string = string.replace(',', ' ').replace(';', ' ')

    em_construcao = collections.deque()
    em_construcao.append(Arvore(None))

    for ch in string:
        if ch == '(':
            pai_atual = em_construcao[-1]
            filho_novo = Arvore(pai_atual)
            pai_atual.filhos.append(filho_novo)
            em_construcao.append(filho_novo)
        elif ch == ')':
            em_construcao.pop()
        else:
            em_construcao[-1].valor += ch

    assert len(em_construcao) == 1
    return em_construcao[0]

```

3.1.2 Separando e concatenando árvores

As árvores no nosso algoritmo só tem uma letra por nó, mas nós recebemos apenas uma árvore com toda a string de DNA.

Precisamos de um método para capaz de criar uma árvore para cada carácter. A seguinte DFS cria a árvore das i -ésimas letras:

```
def separa_arvore(indice, origem):
    copia_origem = Arvore(None)
    if len(origem.valor):
        copia_origem.valor = origem.valor[indice]

    for filho in origem.filhos:
        copia_filho = separa_arvore(indice, filho)
        copia_filho.pai = copia_origem
        copia_origem.filhos.append(copia_filho)

    return copia_origem
```

Depois de rodar o algoritmo, vamos querer juntar as árvores para encontrar os valores dos nós intermediários. Podemos fazer isso com uma DFS e reduce.

```
def concatena_arvores(arvores):
    fusao = Arvore(None)
    fusao.valor = reduce(lambda string, arv: string + arv.valor,
                        arvores, "")

    for i in xrange(len(arvores[0].filhos)):
        fusao_filho = concatena_arvores(
            map(lambda arvore: arvore.filhos[i], arvores))
        fusao_filho.pai = fusao
        fusao.filhos.append(fusao_filho)

    return fusao
```

3.2 Rodando o algoritmo com os dados do problema

3.3 Rodando o algoritmo com dados reais

Obtemos os dados no formato Newick do Rosalind, uma plataforma de ensino de bioinformática.

4 Extensões

Ao fazer esse exercício, notamos que a árvore já é uma entrada do problema. Como é possível obter a árvore de menor valor a partir das espécies

Esse problema é NP-completo [TODO: colocar referência] e o melhor algoritmo conhecido é [TODO]