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:

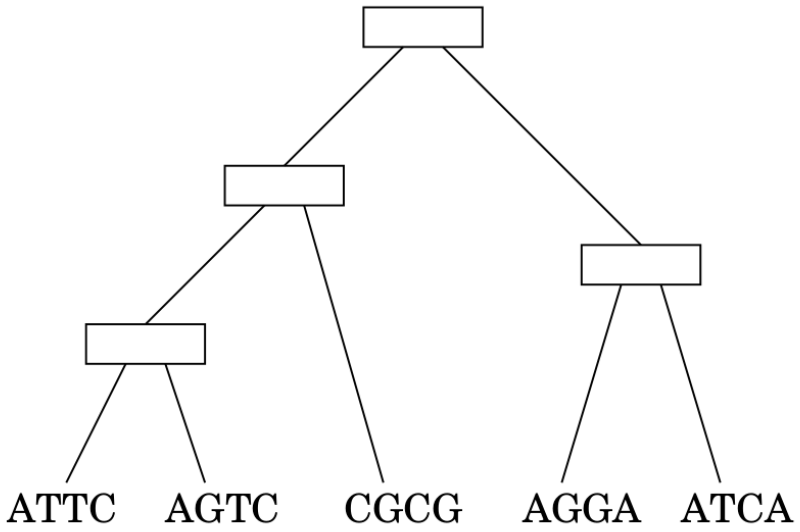
- Uma árvore (binária) evolucionária com as espécies dadas
- Para cada nó interno uma string de comprimento k com a sequência genética daquele ancestral.

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. Como em dados reais de bioinformática é muito comum representar uma deleção como o carácter extra '-', também vamos suportar essa opção na nossa solução.

Vamos representar a nossa árvore como um objeto:

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

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, $melhor\_nota[\text{folha}, \ell]$.

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 (v, l) in melhor_nota:
        return 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.

Podemos agora preencher toda a árvore:

```
def preenche_tudo(raiz):
    melhor_nota_raiz = 100000
    for l in ['A', 'G', 'T', 'C', '-']:
        nota_atual_raiz = sum(calcula_melhor_nota(w, l) for w in raiz.filhos)

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

```
    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)

    return raiz, melhor_nota_raiz
```

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.

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.

Depois construimos a árvore adicionando vértices novos conforme os parênteses.

```
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].nome += ch

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

3.2 Formato FASTA

Outro formato muito comum em bioinformática é o formato FASTA. É um formato bem simples:

```
>Nome
Sequência de DNA
que pode estar em
várias linhas
```

Vamos escrever uma função para converter dados no formato FASTA em um dicionário:

```
def fasta_para_dict(linhas):
    nome = ""
    dna = ""
    saida = {}

    for linha_atual in linhas:
        if linha_atual[0] == ">":
            if nome != "":
                saida[nome] = dna
                dna = ""
            nome = linha_atual[1:]
        else:
            dna += linha_atual

    saida[nome] = dna

    return saida
```

3.3 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)
    copia_origem.nome = origem.nome

    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, "")
    fusao.nome = arvores[0].nome

    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
```

3.4 Imprimindo os resultados

Vamos usar o formato FASTA para imprimir as sequências de DNA que encontrarmos para todos os nós internos.

```
def imprime_resposta(arvore):
    if arvore.filhos:
        print '>%s' % arvore.nome
        print arvore.valor
```

```
    for w in arvore.filhos:
        imprime_resposta(w)
```

3.5 Rodando o algoritmo com os dados do problema

Usando os formatos Newick e FASTA para descrever os dados do problema, obtemos o seguinte:

```
(((folha1,folha2)interno1,folha3)interno2,(folha4,folha5)interno3)interno4;
>folha1
ATTC
>folha2
AGTC
>folha3
CGCG
>folha4
AGGA
>folha5
ATCA
```

Salvamos esta entrada no arquivo `livro.in`.
Podemos agora rodar o código para descobrir os valores dos nós internos:

```
with open(arquivo_entrada, 'r') as f:
    entrada = map(lambda s: s.strip(), f.readlines())

raiz_grande = parseia_newick(entrada[0])
dnas = fasta_para_dict(entrada[1:])

def preenche_folhas(arvore, dnas):
    if arvore.nome in dnas:
        arvore.valor = dnas[arvore.nome]

    for w in arvore.filhos:
        preenche_folhas(w, dnas)

preenche_folhas(raiz_grande, dnas)

n = len(dnas.values()[0])
```

```
arvores_sep = [separa_arvore(i, raiz_grande) for i in range(n)]
pares_preenchidos = [preenche_tudo(a) for a in arvores_sep]

nota_total = sum(nota for (_, nota) in pares_preenchidos)
resposta = concatena_arvores([raiz for (raiz, _) in pares_preenchidos])

print "Nota:", nota_total
imprime_resposta(resposta)
```

Executando o código acima descrito, obtemos a seguinte saída.

```
Nota: 7
>interno4
AGCA
>interno2
AGCA
>interno1
AGTC
>interno3
AGCA
```

3.6 Rodando o algoritmo com dados mais complexos

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

O arquivo rosalind.in contém um banco de dados de 221 espécies, cada uma com 266 caracteres.

Como o resultado é muito longo, deixamos o arquivo de saída disponível em rosalind.out. As primeiras linhas da saída são:

```
Nota: 16880
>sibiricus_plumipes
TGGCATTGCAATGTACAAGTGGTCTAACGTCAGTGATCATACTTGCTAAGAATAGAAAAC
CAACCTAAATAAAACCGACACTAAAGACCTCAGCGTTATGCGACATAACTATGCCTCGTT
AAGAATCTGCGTAATACGGCACGCGGACAACATATGACTAGGGGCATAGGAAGTATGAAG
CGTTCGGCAGTCTTGGGATCAAAAAGAAGCGTAGATCAATTTCTATGTAAAACTATACGG
CCGACAAATTTAAAAACGCGAACGATGGAAT
```