Projeto de algoritmos

Profa Mariana Kolberg

(material adaptado da apostila da prof. Luciana)

Programação Dinâmica

Seqüência comum mais longa

- O que é uma subseqüência?
  - Caracteres aparecem na mesma ordem
  - Não necessáriamente consecutivos
- O que é uma subseqüência comum mais longa?
  - É a maior subsequencia que aparece em ambas sequencias originais
- importância
  - Forma de medir a semelhança entre duas sequencias
    - Cadeias genéticas
    - Comparação entre arquivos

Programação Dinâmica

Seqüência comum mais longa

- Exemplo 1:
  - X= {A,B,K,D,J}
  - Y= {A,R,G,K,J}
  - Subsequencias comuns {A,K}, {A,J},{A,K,J}
  - Subsequencia comum mais longa {A,K,J}
- Exemplo 2:
  - X= {A,B,C,B,D,A,B}
  - Y= {B,D,C,A,B,A}
  - Subsequencia comum mais longa {B,C,A,B}
  - Subsequencia comum mais longa {B,C,B,A}

Programação Dinâmica
Seqüência comum mais longa

- A solução por força bruta seria
  - Enumerar todas as subseqüências de X
  - Conferir quais destas subseqüências são também subseqüências de Y
  - A medida que for encontrando estas subseqüências comuns, ir atualizando a mais longa
- Cada subsequência corresponde a um subconjunto de índices {1, 2, 3, ..., m}
  - Existem 2^m subseqüências de X
- Complexidade exponencial!

Programação Dinâmica

Seqüência comum mais longa

- Definição do problema

SCML

Instância Duas seqüencias $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$.
Solução Uma subseqüência comum Z de X, Y.
Objetivo Maximizar o comprimento de Z.

- Ao comparar x_1 e y_1 podemos ter caracteres:
  - Iguais – temos mais um elemento igual na nossa subsequencia, e podemos então analisar x_2 e y_2
  - Diferentes – não temos mais um elemento e podemos
    - Comparar x_1 e a sequencia de y_2 a y_n
    - Comparar y_1 e a sequencia de x_2 a x_n

Programação Dinâmica

Seqüência comum mais longa

- Idéia: prefixo
  - O i-ésimo prefixo de X é $X_i = \langle x_1, x_2, x_3, \ldots, x_i \rangle$
    - Dado $X = \langle A, B, C, B, D, A, B \rangle$
    - $X_3 = \langle A, B, C \rangle$
    - $X_5 = \langle A, B, C, B, D \rangle$

Teorema: Subestrutura Ótima de uma SCML

Sejam as seqüências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, e seja $Z = \langle z_1, z_2, \ldots, z_k \rangle$ qualquer SCML de X e Y

- Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma SCML de X_{m-1} e Y_{n-1}
- Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma SCML de X_{m-1} e Y
- Se $x_m \neq y_n$, então $z_k \neq y_m$ implica que Z é uma SCML de X e Y_{n-1}

Programação Dinâmica
Seqüência comum mais longa

$$c[i,j] = \begin{cases} 0 & \text{se } i=0 \text{ ou } j=0, \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j. \end{cases}$$

- A sequencia comum mais longa é zero quando uma das duas strings é vazia
- Quando os caracteres são iguais, somamos 1 a solução ótima da sequencia comum mais longa até então encontrada
- Quando os caracteres são diferentes, mantemos o valor da maior sequencia ótima encontrada:
  - Considerando o y em questão e apenas o caracter anteior ao x em questão
  - Considerando o x em questão e apenas o caracter anteior ao y em questão

Programação Dinâmica

Seqüência comum mais longa

- Cada elemento de c[i,j] representará o tamanho da maior subsequencia comum entre a subsequencia de X de índice 1 a i e da subsequencia de Y de indice 1 a j.
- Solução vai calculando por linha da matriz, da esquerda para direita, começando na posição 1,1

	-	A	B	C	B	D	A	B
-								
B								
D								
C								
A								
B								
A								

Programação Dinâmica

Seqüência comum mais longa

$$c[i,j] = \begin{cases} 0 & \text{se } i=0 \text{ ou } j=0, \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j. \end{cases}$$

	-	A	B	C	B	D	A	B
-								
B								
D								
C								
A								
B								
A								

Programação Dinâmica

Seqüência comum mais longa

	-	A	B	C	B	D	A	B
-	0	0	0	0	0	0	0	0
B	0							
D	0							
C	0							
A	0							
B	0							
A	0							

Programação Dinâmica

Seqüência comum mais longa

	-	A	B	C	B	D	A	B
-	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Programação Dinâmica

Seqüência comum mais longa

SCML

Entrada Dois strings X e Y e seus respectivos tamanhos m e n medidos em número de caracteres.

Saída O tamanho da maior subseqüência comum entre X e Y.

```
 1  m := comprimento(X)
 2  n := comprimento(Y)
 3  for i := 0 to m do c[i,0] := 0;
 4  for j := 1 to n do c[0,j] := 0;
 5  for i := 1 to m do
 6      for j := 1 to n do
 7          if x_i = y_j then
 8              c[i,j] := c[i-1,j-1] + 1
 9          else
10              c[i,j] := max(c[i,j-1], c[i-1,j])
11          end if
12      end for
13  return c[m,n]
```

Programação Dinâmica
Seqüência comum mais longa

- É possivel fazer otimizações no código
  - Se a única informação que precisamos saber é o comprimento da seqüência comum mais longa, é possível reduzir o espaço de armazenamento utilizado:
    - Os valores de cada linha dependem apenas dos valores da linha ou coluna anterior
- Se queremos guardar qual é a seqüência comum mais longa
  - Idéia: quardar em uma matriz auxiliar qual informação foi utilizada para achar cada valor da matriz de comprimento (cada subproblema) da seqüência mais longa

Programação Dinâmica

Seqüência comum mais longa

Entrada Dois strings X e Y e seus respectivos tamanhos m e n medidos em número de caracteres.

Saída O tamanho da maior subseqüência comum entre X e Y e o vetor b para recuperar uma SCML.

```
1  m := comprimento[X]
2  n := comprimento[Y]
3  for i := 0 to m do c[i,0] := 0;
4  for j := 1 to n do c[0,j] := 0;
5  for i := 1 to m do
6      for j := 1 to n do
7          if x_i = y_j then
8              c[i,j] := c[i-1,j-1] + 1
9              b[i,j] := ↖
10         else if c[i-1,j] ≥ c[i,j-1] then
11                 c[i,j] := c[i-1,j]
12                 b[i,j] := ↑
13             else
14                 c[i,j] := c[i,j-1]
15                 b[i,j] := ←
16 return c e b
```

Programação Dinâmica

Seqüência comum mais longa

	.	B	D	C	A	B	A
.	0	0	0	0	0	0	0
A	0	↑0	↑0	↑0	↖1	←1	↖1
B	0	↖1	←1	←1	↑1	↖2	←2
C	0	↑1	↑1	↖2	←2	↑2	↑2
B	0	↖1	↑1	↑2	↑2	↖3	←3
D	0	↑1	↖2	↑2	↑2	↑3	↑3
A	0	↑1	↑2	↑2	↖3	↑3	↖4
B	0	↖1	↑2	↑2	↑3	↖4	↑4

Programação Dinâmica

Seqüência comum mais longa

- Algoritmo para impressão da SCML

PRINT-SCML

Entrada Matriz $b \in \{\leftarrow, \nwarrow, \uparrow\}^{m \times n}$.

Saída A maior subseqüência Z comum entre X e Y obtida a partir de b.

```
1  if i = 0 or j = 0 then return
2  if b[i, j] = ↖ then
3     Print-SCML(b, X, i 1, j 1)
4     print x_i
5  else if b[i, j] = ↑ then
6     Print-SCML(b, X, i 1, j)
7  else
8     Print-SCML(b, X, i, j 1)
```

Programação Dinâmica

Seqüência comum mais longa

SCML

Entrada Dois strings X e Y e seus respectivos tamanhos m e n medidos em número de caracteres.

Saída O tamanho da maior subseqüência comum entre X e Y.

```
1  m := comprimento(X)
2  n := comprimento(Y)
3  for i := 0 to m do c[i,0] := 0;
4  for j := 1 to n do c[0,j] := 0;
5  for i := 1 to m do
6      for j := 1 to n do
7          if x_i = y_i then
8              c[i,j] := c[i-1,j-1] + 1
9          else
10             c[i,j] := max(c[i,j-1], c[i-1,j])
11         end if
12     end for
13 return c[m,n]
```

Programação Dinâmica

Seqüência comum mais longa

- Análise da complexidade

Programação Dinâmica
Similaridade entre strings

- Calcular o número mínimo de operações básicas necessárias para transformar uma sequencia de caracteres em outra.
- Importância
  - Verificação ortográfica em textos
  - Similaridade entre palavras
  - Sugestão de pesquisa
- Exemplo
  - [Estalar, estrela] = 3
    - Quais??

Programação Dinâmica

Similaridade entre strings

Considere o problema de determinar o número mínimo de operações que transformam um string s em um string t, se as operações permitidas são a inserção de um caracter, a deleção de um caracter ou a substituição de um caracter para um outro. O problema pode ser visto como um alinhamento de dois strings da forma

```
sonhar
vo--ar
```

em que cada coluna com um caracter diferente (inclusive a "falta" de um caracter -) tem custo 1 (uma coluna $(a, -)$ corresponde à uma deleção no primeiro ou uma inserção no segundo string, etc.).

Programação Dinâmica
Similaridade entre strings

Uma solução ótima contém uma solução ótima do subproblema sem a última coluna, senão podemos obter uma solução de menor custo. Existem quatro casos possíveis para a última coluna:

$$\begin{bmatrix} a \\ - \end{bmatrix}; \begin{bmatrix} - \\ a \end{bmatrix}; \begin{bmatrix} a \\ a \end{bmatrix}; \begin{bmatrix} a \\ b \end{bmatrix}$$

com caracteres a, b diferentes. O caso (a, a) somente se aplica, se os últimos caracteres são iguais, o caso (a, b) somente, se eles são diferentes.

Programação Dinâmica
Similaridade entre strings

- inicialização
  - Para criar uma palavra a partir do zero deve-se inserir os caracteres um a um da esquerda para direita
    - Custo para inserir o primeiro é 1
    - Custo para inserir o segundo é 1 + 1
    - E assim por diante
- Iteração
  - Combinar as formas de passar de uma palavra para a outra
  - Considerando cada prefixo, procurar aquela que gere o menor número de transformações
- Se o custo de transformação para o subproblema é o menor, então quando calculamos o custo para o problema maior, este também será o menor possível

Programação Dinâmica

Similaridade entre strings

$$d(s,t) = \begin{cases} \max(|s|,|t|) & \text{se } |s \models 0 \text{ ou } |t \models 0 \\ \min(d(s',t)+1, d(s,t')+1, d(s',t')+[c_1 \neq c_2]) & \text{se } s = s'c_1 \text{ e } t = t'c_2 \end{cases}$$

Essa distância está conhecida como *distância de Levenshtein* (Levenshtein, 1966). Uma implementação direta é

	-	E	S	T	R	E	L	A
-								
E								
S								
T								
A								
L								
A								
R								

Programação Dinâmica

Similaridade entre strings

	-	E	S	T	R	E	L	A
-	0	1	2	3	4	5	6	7
E	1	0	1	2	3	4	5	6
S	2	1	0	1	2	3	4	5
T	3	2	1	0	1	2	3	4
A	4	3	2	1	1	2	3	3
L	5	4	3	2	2	2	2	3
A	6	5	4	3	3	3	3	2
R	7	6	5	4	3	4	4	3

Programação Dinâmica

Similaridade entre strings – força bruta

DISTÂNCIA

Entrada Dois strings s, t e seus respectivos tamanhos n e m medidos em número de caracteres.

Saída A distância mínima entre s e t.

```
1  distância(s,t,n,m):=
2    if (n=0) return m
3    if (m=0) return n
4    if (s_n = t_m) then
5      sol_0 = distância(s,t,n-1,m-1)
6    else
7      sol_0 = distância(s,t,n-1,m-1) + 1
8    end if
9    sol_1 = distância(s,t,n,m-1) + 1
10   sol_2 = distância(s,t,n-1,m) + 1
11   return min(sol_0, sol_1, sol_2)
```

Programação Dinâmica

Similaridade entre strings

PD-DISTÂNCIA

Entrada Dois strings s e t, e n e m, seus respectivos tamanhos medidos em número de caracteres.

Saída A distância mínima entre s e t.

Comentário O algoritmo usa uma matriz $M = (m_{i,j}) \in \mathbb{N}^{(n+1)\times(m+1)}$ que armazena as distâncias mínimas $m_{i,j}$ entre os prefixos $s[1 \ldots i]$ e $t[1 \ldots j]$.

```
1  PD-distância(s,t,n,m):=
2     for i := 0,...,n do m_{i,0} := i
3     for i := 1,...,m do m_{0,i} := i
4     for i := 1,...,n do
5        for j := 1,...,m do
6           if (s_i = t_j) then
7              sol_0 := m_{i-1,j-1}
8           else
9              sol_0 := m_{i-1,j-1} + 1
10          end if
11          sol_1 := m_{i,j-1} + 1
12          sol_2 := m_{i-1,j} + 1
13          m_{i,j} := min(sol_0, sol_1, sol_2);
14       end for
15    return m_{i,j}
```

Programação Dinâmica

Similaridade entre strings

Distância entre textos

Valores armazenados na matriz M para o cálculo da distância entre ALTO e LOIROS

	·	L	O	I	R	O	S
·	0	1	2	3	4	5	6
A	1	1	2	3	4	5	6
L	2	1	2	3	4	5	6
T	3	2	2	3	4	5	6
O	4	3	2	3	4	4	5

```
-ALTO-
LOIROS
```

Programação Dinâmica

Similaridade entre strings

```
1  PD-distância(s,t,n,m):=
2     for i := 0,...,n do m_{i,0} := i
3     for i := 1,...,m do m_{0,i} := i
4     for i := 1,...,n do
5        for j := 1,...,m do
6           if (s_i = t_j) then
7              sol_0 := m_{i-1,j-1}
8           else
9              sol_0 := m_{i-1,j-1} + 1
10          end if
11          sol_1 := m_{i,j-1} + 1
12          sol_2 := m_{i-1,j} + 1
13          m_{i,j} := min(sol_0, sol_1, sol_2);
14       end for
15    return m_{i,j}
```

Programação Dinâmica

Similaridade entre strings

- Análise da complexidade