

Strings e Programação Dinâmica

Edit Distance

Prof. Edson Alves - UnB/FGA

1. *Edit Distance*
2. Variações de $edit(S, T)$

Edit Distance

Falha no algoritmo de busca em string

- Sejam S e T duas strings, com $S \neq T$. Quando comparadas no que diz respeito a igualdade, há três motivos comuns que levam ao veredito falso
- O primeiro é que ambas strings tem mesmo tamanho, mas diferem em um ou mais símbolos
- Por exemplo, se $S = \text{"banana"}$ e $T = \text{"bacana"}$, $|S| = |T|$, mas $S[3] \neq T[3]$
- O segundo ocorre quando a primeira string é mais longa que a segunda, e poderiam se tornar iguais se removidos os caracteres excedentes
- Por exemplo, se $S = \text{"aspectos"}$ e $T = \text{"seco"}$, segue que $|S| > |T|$, mas teríamos $S = T$ se removidos os caracteres das posições 1, 3, 6 e 8 de S

Falha no algoritmo de busca em string

- O terceiro motivo é o fato da primeira string é mais curta do que a segunda, e poderiam se igualar se adicionados os caracteres ausentes
- Por exemplo, se $S = \text{"fga"}$ e $T = \text{"formigas"}$, temos $|S| < |T|$, e ambas se tornariam iguais com a adição dos caracteres **"ormis"** em S , nas devidas posições
- Na prática, é possível obter qualquer string T a partir de uma string S dada, usando uma sequência de operações baseado nos três motivos listados
- Estas operações são: alterar um caractere, adicionar um caractere ou remover um caractere

- O problema denominado *edit distance* consiste em determinar o número mínimo de operações a serem feitas para transformar a string S na string T
- Em termos mais gerais, o problema é computar o custo mínimo desta transformação, se a cada operação for associado um determinado custo
- Este custo mínimo é denotado $edit(S, T)$ e tem as seguintes propriedades:
 1. $edit(S, T) \geq 0$
 2. $edit(S, T) = 0$ se, e somente se, $S = T$
 3. $edit(S, T) = edit(T, S)$ (simetria)
 4. $edit(S, T) \leq edit(S, U) + edit(U, T)$ (desigualdade triangular)

$edit(S, T)$ – Caso base

- Considere que $|S| = m, |T| = n$. Para determinar $edit(S, T)$ deve-se construir uma tabela auxiliar de estados st , onde $st(i, j) = edit(S[1..i], T[1..j])$, com $0 < i \leq m, 0 < j \leq n$
- Os casos bases acontecem quando uma das duas strings é vazia: nestes casos, o mínimo de operações a serem feitas é igual um número de inserções correspondente ao tamanho da string não vazia
- Em notação simbólica,

$$st(0, j) = j$$

$$st(i, 0) = i$$

- Se os custos de inserção, de remoção e de alteração forem c_i, c_r, c_s , respectivamente, então os casos bases devem ser

$$st(0, j) = j \times c_i \quad \# \text{ } j \text{ inserções}$$

$$st(i, 0) = i \times c_r \quad \# \text{ } i \text{ remoções}$$

$edit(S, T)$ – Transições

- A transição entre os estados será, dentre as três operações, a de menor custo
- Uma transição por inserção seria dada por

$$st(i, j) = st(i, j - 1) + c_i, \quad \# \text{ adicionar } T[j]$$

- Uma transição por remoção seria igual a

$$st(i, j) = c_r + st(i - 1, j), \quad \# \text{ remover } S[i]$$

- Uma transição por alteração corresponde a

Mantém $S[i]$ ou substitui $S[i]$ por $T[j]$

$$st(i, j) = st(i - 1, j - 1) + \begin{cases} 0, & \text{se } S[i] = T[j] \\ c_s, & \text{caso contrário} \end{cases}$$

- Assim,

$$st(i, 0) = i \times c_r$$

$$st(0, j) = j \times c_i$$

$$st(i, j) = \min\{st(i, j-1) + c_i, st(i-1, j) + c_r, st(i-1, j-1) + k\}$$

onde $k = 0$ se $S[i] = T[j]$, ou $k = c_s$, caso contrário

- Portanto, $edit(S, T) = st(m, n)$
- Como a tabela tem $(m+1)(n+1)$ estados, e cada transição é feita em $O(1)$, o algoritmo tem complexidade $O(mn)$

Visualização de $edit(S, T)$

$edit(i, j)$		'A'	'C'	'C'	'E'	'P'	'T'	'E'	'D'
	0	1	2	3	4	5	6	7	8
'T'	1	1	2	3	4	5	5	6	7
'E'	2	2	2	3	3	4	5	5	6
'P'	3	3	3	3	4	3	4	5	6

Visualização de $edit(S, T)$

$edit(i, j)$		'A'	'C'	'C'	'E'	'P'	'T'	'E'	'D'
	<u>0</u>	1	2	3	4	5	6	7	8
'T'	1	1	2	3	4	5	5	6	7
'E'	2	2	2	3	3	4	5	5	6
'P'	3	3	3	3	4	3	4	5	6

Substituição de 'T' por 'A'

$$edit(1, 1) = edit(0, 0) + 1$$

Visualização de $edit(S, T)$

$edit(i, j)$		'A'	'C'	'C'	'E'	'P'	'T'	'E'	'D'
	0	1	2	3	4	5	6	7	8
'T'	1	1	2	<u>3</u>	4	5	5	6	7
'E'	2	2	2	3	3	4	5	5	6
'P'	3	3	3	3	4	3	4	5	6

Mantém o caractere 'E' comum

$$edit(2, 4) = edit(1, 3) + 0$$

Visualização de $edit(S, T)$

$edit(i, j)$		'A'	'C'	'C'	'E'	'P'	'T'	'E'	'D'
	0	1	2	3	4	5	6	7	8
'T'	1	1	2	3	4	5	5	6	7
'E'	2	2	2	3	<u>3</u>	4	5	5	6
'P'	3	3	3	3	4	3	4	5	6

Remove o caractere 'P' na string "TEP"

$$edit(3, 4) = edit(2, 4) + 1$$

Visualização de $edit(S, T)$

$edit(i, j)$		'A'	'C'	'C'	'E'	'P'	'T'	'E'	'D'
	0	1	2	3	4	5	6	7	8
'T'	1	1	2	3	4	5	5	6	7
'E'	2	2	2	3	3	4	5	5	6
'P'	3	3	3	3	4	<u>3</u>	4	5	6

Insere o caractere 'T' na string "ACCEP"

$$edit(3, 6) = edit(3, 5) + 1$$

Implementação *bottom-up* de $edit(S, T)$

```
9 int edit(const string& s, const string& t)
10 {
11     const int c_i = 1, c_r = 1, c_s = 1;    // Custos iguais a um
12     int m = s.size(), n = t.size();
13
14     for (int i = 0; i <= m; ++i)
15         st[i][0] = i*c_r;
16
17     for (int j = 1; j <= n; ++j)
18         st[0][j] = j*c_i;
19
20     for (int i = 1; i <= m; ++i)
21         for (int j = 1; j <= n; ++j) {
22             int insertion = st[i][j - 1] + c_i, deletion = st[i - 1][j] + c_r;
23             int change = st[i - 1][j - 1] + c_s*(s[i - 1] == t[j - 1] ? 0 : 1);
24             st[i][j] = min({ insertion, deletion, change });
25         }
26
27     return st[m][n];
28 }
```

Variações de $edit(S, T)$

Implementação de $\text{edit}(S, T)$ com memória $O(n)$

- A implementação anterior tem complexidade $O(nm)$ tanto para o tempo de execução quanto para a memória
- Isto se deve à tabela de estados st , que tem dimensões $m \times n$
- É possível implementar o mesmo algoritmo usando apenas $O(n)$ de memória, uma vez que é necessário apenas a linha anterior para computar os valores da próxima linha
- Esta segunda implementação pode ser necessária em competições ou ambientes com restrição de memória
- A complexidade do tempo de execução, porém, se mantém igual

Implementação de $edit(S, T)$ com memória $O(n)$

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const int MAX { 1000 };
6
7 int a[MAX], b[MAX];
8
9 int edit(const string& s, const string& t)
10 {
11     const int c_i = 1, c_r = 1, c_s = 1;      // Custos iguais a um
12     int m = s.size(), n = t.size(), *prev = a, *line = b;
13
14     for (int j = 1; j <= n; ++j)
15         prev[j] = j*c_i;
16
17     for (int i = 1; i <= m; ++i)
18     {
19         line[0] = i*c_r;
```

Implementação de $edit(S, T)$ com memória $O(n)$

```
21     for (int j = 1; j <= n; ++j) {
22         int insertion = line[j - 1] + c_i;
23         int deletion = prev[j] + c_r;
24         int change = prev[j-1] + c_s*(s[i-1] == t[j-1] ? 0 : 1);
25         line[j] = min({ insertion, deletion, change });
26     }
27
28     swap(line, prev);
29 }
30
31 return prev[n];
32 }
33
34 int main()
35 {
36     string s { "TEP" }, t { "ACCEPTED" };
37     cout << edit(s, t) << '\n';
38
39     return 0;
40 }
```

Sequência de operações ótima

- Uma variante do problema *edit distance* é retornar um conjunto de operações $O = \{o_1, o_2, \dots, o_s\}$ tal que $s = \text{edit}(S, T)$ e o_j é uma das três operações: inserção, remoção ou alteração
- Para obter tal sequência, na implementação com memória $O(nm)$, basta manter um registro da operação responsável pela atualização de cada elemento da tabela st
- Ao final do algoritmo, uma sequência de operações que leva ao custo mínimo pode ser reconstruída por meio da recursão
- A construção da tabela tem complexidade $O(nm)$, e a identificação da sequência de operações tem complexidade $O(n + m)$

Implementação da sequência de operações ótima

```
9 // -      Deletion
10 // c      Insertion of char c
11 // =      Keep
12 // [c->d]  Change (c to d)
13 string edit_operations(const string& s, const string& t)
14 {
15     const int c_i = 1, c_r = 1, c_s = 1;      // Custos iguais a um
16     int m = s.size(), n = t.size();
17
18     for (int i = 0; i <= m; ++i)
19     {
20         st[i][0] = i*c_r;
21         ps[i][0] = 'r';
22     }
23
24     for (int j = 1; j <= n; ++j)
25     {
26         st[0][j] = j*c_i;
27         ps[0][j] = 'i';
28     }
```

Implementação da sequência de operações ótima

```
30  for (int i = 1; i <= m; ++i)
31      for (int j = 1; j <= n; ++j) {
32          int insertion = st[i][j - 1] + c_i, deletion = st[i-1][j] + c_r;
33          int change = st[i-1][j-1] + c_s*(s[i-1] == t[j-1] ? 0 : 1);
34          st[i][j] = min({ insertion, deletion, change });
35
36          ps[i][j] = (insertion <= deletion and insertion <= change) ?
37              'i' : (deletion <= change ? 'r' : 's');
38      }
39
40  int i = m, j = n;
41  ostringstream os;
42
43  while (i or j)
44  {
45      switch (ps[i][j]) {
46          case 'i':
47              os << t[j - 1];
48              --j;
49              break;
```

Implementação da sequência de operações ótima

```
51     case 'r':
52         os << '-';
53         --i;
54         break;
55
56     case 's':
57         if (s[i-1] == t[j-1])
58             os << '=';
59         else
60             os << "]" << t[j - 1] << ">-" << s[i - 1] << "[";
61
62         --i; --j;
63     }
64 }
65
66 auto ops = os.str();
67 reverse(ops.begin(), ops.end());
68
69 return ops;
70 }
```

1. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.