Strings

String e Programação Dinâmica – *Edit Distance*

Prof. Edson Alves - UnB/FGA

2019

Sumário

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 estão
- O primeiro é que ambas strings tem mesmo tamanho, mas diferem em um ou mais símbolos
- Por exemplo, se $S =$ `"banana"` e $T =$ `"bacana"`, $|S| \neq |T|$, mas $S[3] \neq T[3]$
- O segundo ocorre quando a primeira string é mais longa que a segunda, e poderíam se tornar iguais se removidos os caracteres excedentes
- Por exemplo, se $S =$ `"aspectos"` e $T =$ `"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 caracateres ausentes
- Por exemplo, se $S =$ `"fga"` e $T =$ `"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 listas
- Esta operações são: alterar um caractere, adicionar um caractere ou remover um caractere

Edit Distance

- O problema denominado *edit distance* consiste em determinar o número mínimo de operações a serem feitas
- Em termos mais gerais, 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 $0 < i \leq m, 0 < j \leq n$
- O 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

$$st(0,j) = j \times c_i \qquad \texttt{\# } j \texttt{ inserções}$$
$$st(i,0) = i \times c_r \qquad \texttt{\# } i \texttt{ remoções}$$

$edit(S, T)$ – Transições

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

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

- Uma transição por remoção seria igual a

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

- Uma transição por alteração corresponde a

$$\texttt{\# Mantém } S[i] \texttt{ ou substitui } S[i] \texttt{ por } T[j]$$

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

$edit(S, T)$

- 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'
	$\underline{0}$	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

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	$\underline{3}$	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

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	$\underline{3}$	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	$\underline{3}$	**4**	5	6

Insere o caractere 'T' na string "ACCEP"

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

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

```cpp
int edit(const string& s, const string& t)
{
    const int c_i = 1, c_r = 1, c_s = 1;      // Custos iguais a um
    int m = s.size(), n = t.size();

    for (int i = 0; i <= m; ++i)
        st[i][0] = i*c_r;

    for (int j = 1; j <= n; ++j)
        st[0][j] = j*c_i;

    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j) {
            int insertion = st[i][j - 1] + c_i;
            int deletion = st[i-1][j] + c_r;
            int change = st[i-1][j-1] + c_s*(s[i-1] == t[j-1] ? 0 : 1);
            st[i][j] = min({ insertion, deletion, change });
        }

    return st[m][n];
}
```

Variações de $edit(S, T)$

Implementação de $edit(S, T)$ 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 necesário 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 $O(n)$

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAX { 1000 };

int a[MAX], b[MAX];

int edit(const string& s, const string& t)
{
    const int c_i = 1, c_r = 1, c_s = 1;      // Custos iguais a um
    int m = s.size(), n = t.size();
    int *prev = a, *line = b;

    for (int j = 1; j <= n; ++j)
        prev[j] = j*c_i;

    for (int i = 1; i <= m; ++i)
    {
        line[0] = i*c_r;

```

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

```
        for (int j = 1; j <= n; ++j) {
            int insertion = line[j - 1] + c_i;
            int deletion = prev[j] + c_r;
            int change = prev[j-1] + c_s*(s[i-1] == t[j-1] ? 0 : 1);
            line[j] = min({ insertion, deletion, change });
        }

        swap(line, prev);
    }

    return prev[n];
}

int main()
{
    string s { "TEP" }, t { "ACCEPTED" };

    cout << edit(s, t) << '\n';

    return 0;
}
```

Sequência de operações ótima

- Uma variante do problema *edit distance* é retorna um conjunto de operações $O = \{o_1, o_2, \ldots, o_s\}$ tal que $s = edit(S, T)$ 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

```
// -        Deletion
// c        Insertion of char c
// =        Keep
// [c->d]   Change (c to d)
string edit_operations(const string& s, const string& t)
{
    const int c_i = 1, c_r = 1, c_s = 1;      // Custos iguais a um
    int m = s.size(), n = t.size();

    for (int i = 0; i <= m; ++i)
    {
        st[i][0] = i*c_r;
        ps[i][0] = 'r';
    }

    for (int j = 1; j <= n; ++j)
    {
        st[0][j] = j*c_i;
        ps[0][j] = 'i';
    }

```

Implementação da sequência de operações ótima

```
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j) {
            int insertion = st[i][j - 1] + c_i;
            int deletion = st[i-1][j] + c_r;
            int change = st[i-1][j-1] + c_s*(s[i-1] == t[j-1] ? 0 : 1);
            st[i][j] = min({ insertion, deletion, change });

            ps[i][j] = (insertion <= deletion and insertion <= change) ?
                'i' : (deletion <= change ? 'r' : 's');
        }

    int i = m, j = n;
    ostringstream os;

    while (i or j)
    {
        switch (ps[i][j]) {
        case 'i':
            os << t[j - 1];
            --j;
            break;
```

Implementação da sequência de operações ótima

```
        case 'r':
            os << '-';
            --i;
            break;

        case 's':
            if (s[i-1] == t[j-1])
                os << '=';
            else
                os << "]" << t[j - 1] << ">-" << s[i - 1] << "[";

            --i;
            --j;
        }
    }

    auto ops = os.str();
    reverse(ops.begin(), ops.end());

    return ops;
}
```

Referências

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