

# PAA – Relatório do Trabalho Prático 1

Danilo Ferreira e Silva<sup>1</sup>

<sup>1</sup>Departamento de Ciência da Computação – UFMG

danilofs@dcc.ufmg.br

## 1. Introdução

O objetivo deste trabalho foi a implementação em diferentes paradigmas de programação do problema de encontrar subsequências comuns. Mais especificamente, no problema proposto deseja-se descobrir o tamanho da maior subsequência comum entre pares de *strings*, com a restrição adicional de que essa subsequência deve ser composta de segmentos de tamanho mínimo  $k$ .

## 2. Formulação da solução

Antes de discutir a solução do problema, é necessário definir algumas notações usadas.

Seja  $w_n$  uma palavra formada pela sequência de  $n$  caracteres  $c_1, c_2, \dots, c_n$ . Definimos o sufixo  $w_i$  como uma subpalavra formada pelos últimos  $i$  caracteres. Ou seja, se  $w_n = abcd$ , então  $w_4 = abcd$ ,  $w_3 = bcd$ ,  $w_2 = cd$ , e assim por diante.

Seja também a função  $kmatch(k, w_n, v_m)$ , onde  $k$  é um número inteiro e  $w_n = c_1, c_2, \dots, c_n$ ,  $v_m = d_1, d_2, \dots, d_m$  são palavras, definida como:

$$kmatch(k, w_n, v_m) = \begin{cases} 1 & \text{se } c_i = d_i \text{ para os } k \text{ primeiros caracteres de } w_n \text{ e } v_m \\ 0 & \text{caso contrário} \end{cases}$$

Finalmente, definimos  $lcss(k, w_n, v_m) \rightarrow \mathbb{N}$  como a função que retorna o tamanho da maior subsequência comum entre as palavras  $w_n$  e  $v_m$ , e que sejam formadas de segmentos de tamanho  $k$  ou maior. Deve-se notar que  $lcss(k, w_n, v_m) = 0$  sempre que  $n = 0$  ou  $m = 0$ . Para simplificar as definições apresentadas posteriormente, este caso base será omitido.

Para o caso particular onde  $k = 1$ , o problema se torna mais simples, e pode ser expresso de forma recursiva:

$$lcss(1, w_n, v_m) = \max \left\{ \begin{array}{ll} \left\{ \begin{array}{ll} 1 + lcss(1, w_{n-1}, v_{m-1}) & \text{se } c_1 = d_1 \\ 0 & \text{se } c_1 \neq d_1 \end{array} \right. \\ lcss(1, w_{n-1}, v_m) \\ lcss(1, w_n, v_{m-1}) \end{array} \right\}$$

Ou seja, a função escolhe a maior dentre três possibilidades, casar os caracteres  $c_1$  e  $d_1$  assumindo que ambos participam da solução, pular o caracter  $c_1$ , ou pular o caracter  $d_1$ . No primeiro caso, só é possível fazer o casamento quando  $c_1 = d_1$ , caso contrário

a solução seria inválida e seu resultado é 0. Cada uma dessas possibilidades nos dá um subproblema menor, reduzindo ao menos um caracter.

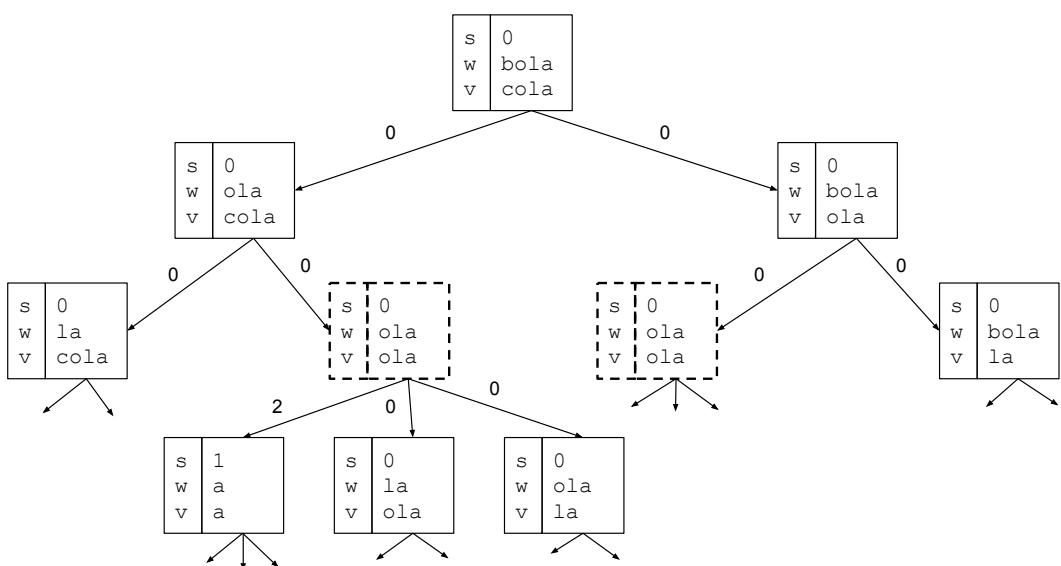
Para tratar o restrição do tamanho mínimo do segmento, é necessário introduzir um parâmetro auxiliar booleano  $s$  na função para indicar se o subproblema deve assumir que já foi casado um segmento de tamanho mínimo  $k$ . Neste caso, a definição de  $lcss(k, w_n, v_m, s)$  é:

$$\max \left\{ \begin{array}{ll} k + lcss(1, w_{n-k}, v_{m-k}, 1) & \text{se } s = 0 \text{ e } kmatch(k, w_n, v_m) = 1 \\ 1 + lcss(1, w_{n-1}, v_{m-1}, 1) & \text{se } s = 1 \text{ e } kmatch(1, w_n, v_m) = 1 \\ 0 & \text{caso contrário} \\ lcss(1, w_{n-1}, v_m, 0) & \\ lcss(1, w_n, v_{m-1}, 0) & \end{array} \right.$$

## 2.1. Solução por força bruta

Baseando-se na formulação recursiva do problema foi implementado um algoritmo recursivo que testa exaustivamente todas as possibilidades.

Na figura 1 é ilustrada o início da árvore de recursão para uma instância do problema com  $k = 2$ ,  $w_n = bola$ ,  $v_m = cola$ . Nela pode-se notar que quando existe a possibilidade de casamento, divide-se em três ramos, quando não existe a divisão é feita em dois ramos. Expandindo totalmente a árvore testa-se exaustivamente qualquer combinação possível de se formar uma subsequência, portanto a solução ótima é encontrada. É interessante observar que há repetição de subproblemas, como demarcado nos dois nodos pontilhados. Essa propriedade será explorada na solução em programação dinâmica, que será discutida posteriormente.



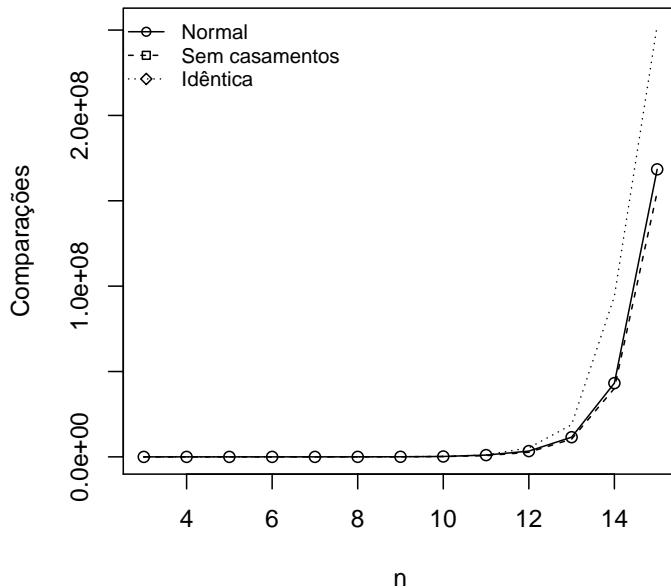
**Figura 1. Árvore de recursão para o problema  $k = 2$ ,  $w_n = bola$ ,  $v_m = cola$**

### 2.1.1. Custo de espaço e tempo

Se defirmos  $N = n + m$ , podemos expressar o custo da solução pela equação de recorrência quando  $k = 1$ :

$$T(N) = 2T(N - 1) + T(N - 2) + O(k)$$

Expandindo essa equação é possível verificar que a complexidade é  $O(k3^N)$ . Note que o fator  $O(k)$  é um custo relacionado a operação *kmatch*, que no pior caso compara  $k$  caracteres. É importante ressaltar que a equação de recorrência dada assume um pior caso onde sempre é possível fazer um casamento. Na prática isso nem sempre irá ocorrer e algumas recursões se dividirão em apenas dois ramos.



**Figura 2. Número de comparações por tamanho da entrada do algoritmo de força bruta**

Com relação a espaço, a solução implementada usa um método recursivo, portanto é necessário manter em memória a pilha com os registros de ativação das chamadas. Como a profundidade da árvore é  $N$ , o custo de memória é  $O(N)$ .

No gráfico 2 podemos ver o número de comparações, que é a operação mais significativa, para variados tamanhos de entrada. O gráfico apresenta três linhas para três casos diferentes de entrada, um para o caso de palavras idênticas, outra pra o caso de palavras sem nenhum casamento de caracteres e, finalmente, um caso onde duas palavras são geradas com caracteres aleatórios entre a-h, distribuídos uniformemente.

### 2.2. Solução usando programação dinâmica

Partindo da definição recursiva descrita na solução por força bruta, podemos resolver o problema usando programação dinâmica aproveitando-se das proprieda-

des de subestrutura ótima e sobreposição de subproblemas. A solução do problema  $lcss(k, w_n, v_m, s)$  depende da solução dos subproblemas,  $lcss(k, w_{n-1}, v_m, 0)$ ,  $lcss(k, w_n, v_{m-1}, 0)$ ,  $lcss(k, w_{n-1}, v_{m-1}, 1)$ ,  $lcss(k, w_{n-k}, v_{m-k}, 1)$ . Mais do que isso, o número total de subproblemas distintos na formulação dada é o número possível de combinações de  $w_i$ ,  $v_j$  e  $s$ , ou seja,  $n \times m \times 2$ .

**Tabela 1. Tabela de solução de subproblemas**

	$s = 0$				$s = 1$			
	B	O	L	A	B	O	L	A
C	3	3	2	0	3	3	2	0
O	3	3	2	0	3	3	2	0
L	2	2	2	0	2	2	2	0
A	0	0	0	0	0	0	0	1

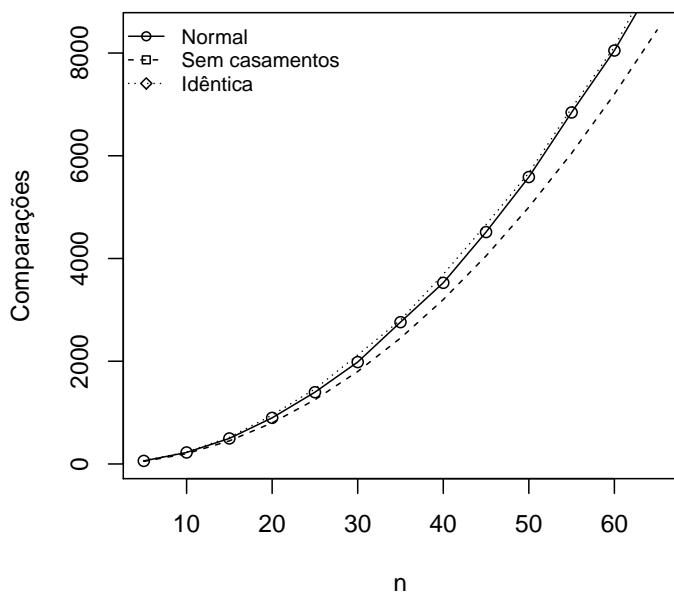
Dessa forma, é possível resolver o problema computando o valor de cada subproblema em duas matrizes  $n \times m$ , uma para  $s = 0$  e outra para  $s = 1$ . A matriz é preenchida da direita para a esquerda, de cima para baixo, partindo do menor subproblema para o maior, até chegar na solução final. A tabela 1 mostra o resultado da computação de cada posição das tabelas usadas no algoritmo de programação dinâmica para o problema com  $k = 2$ ,  $w_n = bola$ ,  $v_m = cola$ .

### 2.2.1. Custo de tempo e espaço

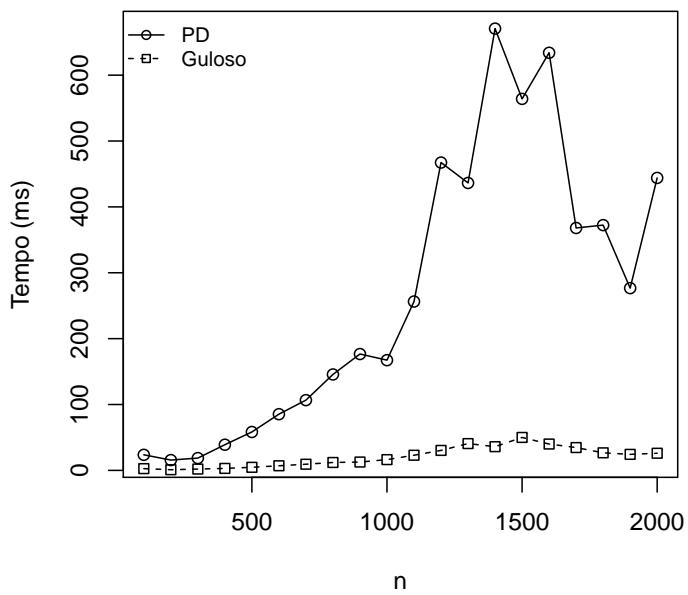
O custo de tempo e espaço do algoritmo usando programação dinâmica é  $O(n \times m \times 2)$ , pois toda a matriz deve ser preenchida. No entanto, é possível melhorar o custo de espaço observando o fato que de que nem todas as células da tabela são necessárias durante todo a computação. De fato, quando se computa a célula  $(i, j)$ , dependemos apenas das células  $(i + k, j + k)$ ,  $(i + 1, j + 1)$ ,  $(i + 1, j)$  e  $(i, j + 1)$ . No entanto, por simplicidade, essa melhoria não foi implementada.

Na figura 3 temos um gráfico que mostra o número de comparações efetuadas para entradas de tamanho variado, e com  $k = 3$ . O algoritmo dinâmico tem pouca variação com relação a similaridade das palavras de entrada, mas ainda sim o seu pior caso ocorre quando elas são identicas, pois a operação *kmatch* irá custar  $k$  em mais casos.

Na figura 4 temos um gráfico comparando o tempo de execução do algoritmo dinâmico com o guloso, que é discutido na próxima sessão. Neste caso foi usado  $k = 8$  e palavras de entrada também geradas aleatoriamente, de forma similar aos demais experimentos.



**Figura 3.** Número de comparações por tamanho da entrada do algoritmo dinâmico



**Figura 4.** Comparaçāo dos tempos de execuāo dos algoritmos dinâmico e gulosso para entradas grandes

## 2.3. Solução usando algoritmo guloso

Por último, foi implementado uma solução gulosa, baseada no seguinte algoritmo:

```
1: result ← 0
2: s ← 0
3: jmin ← 1
4: for i = 1 → n do
5:   for j = jmin → m do
6:     km ← if s = 0 then k else 1
7:     if kmatch(km, wi, vj) = 1 then
8:       jmin ← j + 1
9:       result ← result + km
10:      s ← 1
11:      break inner loop and update i
12:    else
13:      s ← 0
14:    end if
15:  end for
16: end for
```

Este algoritmo basicamente encontra de forma gulosa um casamento de  $k$  caracteres percorrendos os caracteres sequencialmente com dois *loops* aninhados. Ao encontrar um casamento, o algoritmo assume que ele faz parte da solução e simplesmente continua a buscar outros casamentos sem nunca voltar atrás.

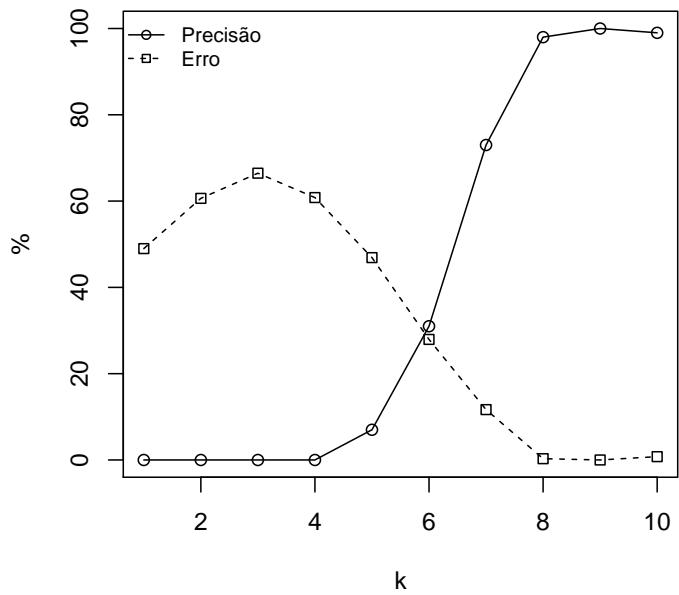
Naturalmente esse algoritmo não é capaz de encontrar a solução ótima. Com a entrada  $k = 1$ ,  $w_n = \text{lovxxelyxxxxx}$ ,  $v_m = \text{xxxxxxxxlovely}$  a resposta deveria ser 7, mas o algoritmo guloso retorna 6. Isso acontece pois o primeiro casamento encontrado por ele é do caracter  $l$ , o que faz com que ele descarte o casamento de  $x$ , que gera uma subsequência maior.

Na figura 5 temos um gráfico que mostra duas curvas, uma com a precisão e outra com o erro para valores variados de  $k$ . A precisão é medida como a porcentagem de execuções onde o erro é zero, executando o algoritmo 100 vezes para cada  $k$ . A curva de erro mostra a média do percentual de quão menor é a solução gulosa comparada a solução ótima, mais uma vez para as mesmas 100 para cada  $k$ . As palavras de entradas são de tamanho fixo 100, geradas aleatoriamente de forma semelhante aos demais experimentos.

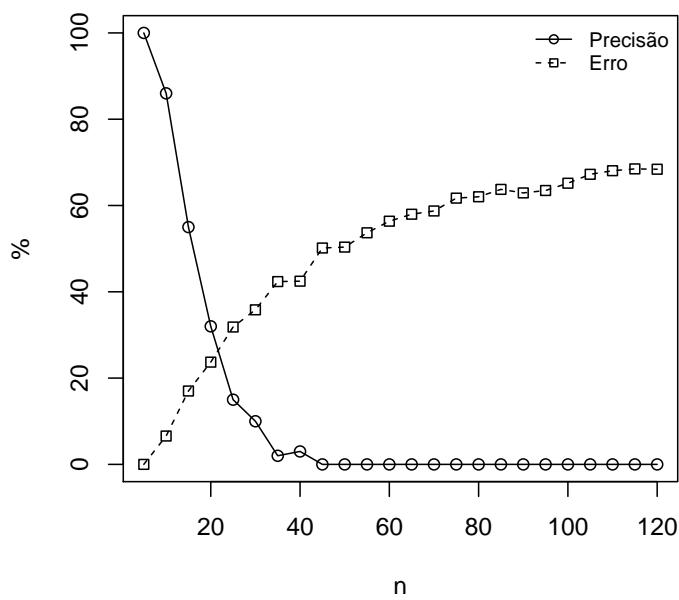
Já na figura 6 temos um gráfico semelhante, mas variando o tamanho da entrada com  $k = 3$ . Em ambos os gráficos notamos uma variação grande do erro e precisão por estes dois fatores. No entanto, de forma geral o erro tende a ser alto, ultrapassando 50%.

### 2.3.1. Custo de tempo e espaço

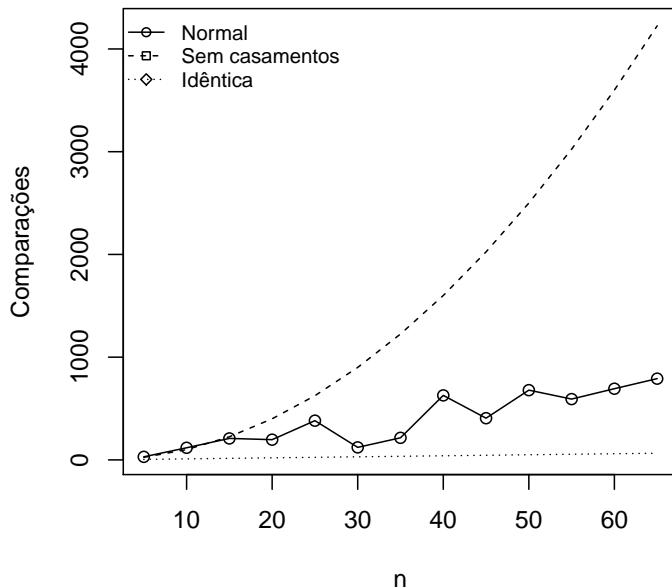
No pior caso, o algoritmo guloso é  $O(knm)$ , pois compara cada caracter de  $w_n$  com cada caracteres de  $v_m$ . Isso ocorre quando as palavras são totalmente distintas. Em casos mais típicos, porém, quando existem casamentos, o número de comparações feitas é mais próximo de linear, como podemos ver no gráfico 7. Em um caso extremo de palavras idênticas o custo é linear.



**Figura 5. Precisão do algoritmo guloso variando o valor de  $k$**



**Figura 6. Precisão do algoritmo guloso variando o tamanho da entrada**



**Figura 7. Número de comparações por tamanho da entrada do algoritmo gúloso**

### 3. Conclusão

Neste trabalho foram implementados três algoritmos para o problema de subsequências comuns usando três paradigmas distintos, exercitando os conceitos de paradigmas de programação. O primeiro deles usou a abordagem de força bruta, que encontra a solução ótima mas torna-se inviável na prática por seu custo exponencial, como constatado pelos experimentos.

Uma segunda abordagem usando programação dinâmica foi feita, cujo custo de espaço e tempo é quadrático. Isso viabiliza a execução com entradas de tamanhos bem maiores, com a desvantagem do espaço necessário para memorização de soluções.

Por último, foi implementado um algoritmo gúloso com custo quadrático no pior caso, mas que na prática pode ser próximo de linear. Esse algoritmo no entanto não é ótimo e sua precisão é altamente dependente de fatores como o tamanho da entrada e do valor de  $k$ .