



UNIVERSIDADE FEDERAL DA BAHIA – UFBA  
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO  
PROGRAMA DE GRADUAÇÃO

ANTÔNIO ROBERTO PAOLI

**UM ESTUDO AVANÇADO DO PROBLEMA  
DA MAIOR SUBSEQUÊNCIA COMUM**

Salvador - BA, Brasil

31 de maio 2016

Antônio Roberto Paoli

**Um estudo avançado do problema da Maior  
Subsequência Comum**

Monografia apresentada para obtenção do  
Grau de Bacharel em Ciência da Computação  
pela Universidade Federal da Bahia.

Universidade Federal da Bahia – UFBA  
Departamento de Ciência da Computação  
Programa de Graduação

Orientador: Maurício Pamplona Segundo

Salvador - BA, Brasil

31 de maio 2016

Antônio Roberto Paoli

Um estudo avançado do problema da Maior Subsequência Comum/ Antônio Roberto Paoli. – Salvador - BA, Brasil, 31 de maio 2016-

70 p. : il. (algumas color.) ; 30 cm.

Orientador: Maurício Pamplona Segundo

Monografia – Universidade Federal da Bahia – UFBA

Departamento de Ciência da Computação

Programa de Graduação, 31 de maio 2016.

1. LCS. 2. longest common subsequence problem. 2. bit-parallel LCS-length. 3. comparação de cadeias de caracteres. I. Maurício Pamplona Segundo. II. Universidade Federal da Bahia. III. Departamento de Ciências da Computação. IV. Longest Common Subsequence Problem

Antônio Roberto Paoli

## **Um estudo avançado do problema da Maior Subsequência Comum**

Monografia apresentada para obtenção do  
Grau de Bacharel em Ciência da Computação  
pela Universidade Federal da Bahia.

Trabalho aprovado. Salvador - BA, Brasil, 31 de maio de 2016:

---

**Maurício Pamplona Segundo**  
Orientador

---

**Rubisley de Paula Lemes**  
Avaliador

---

**Jesus Ossian da Cunha Silva**  
Avaliador

Salvador - BA, Brasil  
31 de maio 2016

*“orandum est ut sit mens sana in corpore sano”*

*Satira 10 - Juvenal*

# Resumo

O problema da Maior Subsequência Comum é um problema clássico da Ciência da Computação que consiste em encontrar a mais longa subsequência de caracteres comuns a duas strings. A solução genérica é obtida pelo uso de uma matriz que combina estes caracteres dois a dois, com alto custo de tempo e uso de memória. Este trabalho busca soluções para o cálculo mais eficiente do comprimento da Maior Subsequência Comum. Para isso, propõe dois novos algoritmos denominados limpeza de matriz e diagonal. Para confirmar os resultados destes novos algoritmos são apresentadas e avaliadas as implementações de diversos trabalhos bem estudados na literatura que são comparados nas mesmas condições de ambiente. Adicionalmente, as soluções são submetidas ao repositório do Sphere online Judge (SPOJ) para confirmar os resultados pelo confronto com diferentes soluções de programadores de todo o mundo.

**Palavras-chave:** Maior Subsequência Comum; bit-paralelo; paralelização; diagonal; limpeza de matriz.

# Abstract

The problem of Longest Common subsequence (LCS) is a classic problem of computer science which is to find the longest subsequence of characters common to two strings. The general solution is obtained by use of a matrix that combines these two by two characters, with a high time cost and memory usage. This work seeks solutions for more efficient calculation of the length of the LCS. To this end, proposes two new algorithms called matrix cleaning and diagonal. To confirm the results of these new algorithms are presented and evaluated the implementation of several works well studied in the literature and compared in the same environmental conditions. In addition, these solutions are submitted to Sphere online Judge (SPOJ) repository to confirm the results by confrontation with different programmer solutions worldwide.

**Keywords:** longest common subsequence; bit-parallel; parallelization; diagonal, matrix cleaning.

# Lista de ilustrações

Figura 1 – Configuração do computador utilizado nos experimentos. . . . .	18
Figura 2 – Matriz LCS preenchida por programação dinâmica, considerando a lógica de recursão por sufixo, ordenada por prefixo. Os valores são calculados sequencialmente da esquerda para a direita e de cima para baixo. . . . .	21
Figura 3 – Matriz LCS preenchida por programação dinâmica, considerando a lógica de recursão por prefixo, ordenada por sufixo. Os valores são calculados sequencialmente da direita para a esquerda e de baixo para cima. . . . .	22
Figura 4 – Mapa de coincidências comum às strings $X$ e $Y$ . . . . .	25
Figura 5 – Mapa de coincidências de caracteres das strings $X$ e $Y$ . O ponto em (4,6) domina os pontos dentro do quadrante 4, que são candidatos a dar continuidade à LCS. . . . .	26
Figura 6 – Matriz de Programação Dinâmica e mapa de coincidências com pontos dominantes marcados com círculos e os pontos irrelevantes com x. . . . .	27
Figura 7 – Mapa de distribuição do conjunto de pontos. Conjunto $D$ em círculos azuis e conjunto $K$ demarcados com círculo laranja. . . . .	28
Figura 8 – Matriz preenchida por programação dinâmica delineando as fronteiras de mudança de uma unidade no comprimento da LCS. . . . .	29
Figura 9 – Sequência de análise linha a linha do algoritmo última linha. O número de fronteiras é o comprimento da LCS. . . . .	30
Figura 10 – Lista encadeada para a string $X$ . . . . .	33
Figura 11 – Sequência de exemplo da solução limpeza de matriz. . . . .	35
Figura 12 – A diagonal caracteriza um eixo de semelhança na matriz LCS. . . . .	37
Figura 13 – Sequência de exemplo da solução diagonal. . . . .	41
Figura 14 – Representação binária da matriz de casamentos de caracteres. Bits 1 marcam os casamentos dos caracteres, sendo que os bits em azul são relevantes e delineiam as fronteiras. . . . .	45
Figura 15 – Representação da matriz de casamentos de caracteres em reverso para uso da técnica de bit-paralelo. . . . .	46
Figura 16 – Representação da evolução do vetor $L_i$ através do algoritmo bit-paralelo. . . . .	51
Figura 17 – Representação de diagonais e triângulos de corte sobre no processamento da LLCS. . . . .	52
Figura 18 – <i>Ranking</i> do problema LCS0 no SPOJ <a href="http://www.spoj.com/ranks/LCS0/">http://www.spoj.com/ranks/LCS0/</a> em 15/Mai/16. . . . .	59



Figura 19 – <i>Ranking</i> do problema AIBOHP em <a href="http://www.spoj.com/ranks/AIBOHP/">www.spoj.com/ranks/AIBOHP/</a> em 15/Mai/2016. . . . .	61
Figura 20 – <i>Ranking</i> do problema IOIPALIN em <a href="http://www.spoj.com/ranks/IOIPALIN/">www.spoj.com/ranks/IOIPALIN/</a> em 15/Mai/2016. . . . .	63
Figura 21 – Exemplo para o problema XMEN. (a) matriz de permutações e (b) matriz reindexada. . . . .	65

# Lista de tabelas

Tabela 1 – Casos de teste. . . . .	17
Tabela 2 – Estimativa de tempo de execução para string com 50.000 caracteres. . .	22
Tabela 3 – Tabela comparativa entre implementações Allison e Dix (1986), Crochemore et al. (2000) e Hyyrö (2004). . . . .	49
Tabela 4 – Resumo dos tempos de execução e das complexidades das soluções apresentadas. . . . .	56
Tabela 5 – Problema LCS0 do SPOJ. . . . .	58
Tabela 6 – Problema AIBOHP do SPOJ. . . . .	60
Tabela 7 – Problema IOIPALIN do SPOJ. . . . .	62
Tabela 8 – Problema XMEN do SPOJ. . . . .	64
Tabela 9 – Tempo em ms das soluções aplicadas ao Problema XMEN e resultado no SPOJ. . . . .	66

# Lista de abreviaturas e siglas

LCS	Longest Common Subsequence Problem
LLCS	Length of the Longest Common Subsequence
MLCS	Multiple Longest Common Subsequence Problem
RFLCS	Repetition Free Longest Common Subsequence Problem
SPOJ	Sphere Online Judge <i>"www.spoj.com"</i>
LIS	Longest Increasing Subsequence Problem

# Lista de símbolos

$\alpha$	Letra grega minúscula alpha
$\Sigma$	Letra grega maiúscula sigma
$\lambda$	Letra grega minúscula lambda
$\subseteq$	Está contido ou é igual
$\supseteq$	Contém ou é igual
$\in$	Pertence
$ A $	Cardinalidade do conjunto A

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>13</b>
<b>2</b>	<b>REFERENCIAL TEÓRICO</b>	<b>15</b>
<b>3</b>	<b>AMBIENTE</b>	<b>17</b>
<b>4</b>	<b>IMPLEMENTAÇÕES PARA O PROBLEMA DA LCS</b>	<b>19</b>
<b>4.1</b>	<b>Soluções <math>\geq O(N^2)</math></b>	<b>19</b>
4.1.1	Solução recursiva	19
4.1.2	Solução matricial	20
4.1.3	Solução matricial em espaço linear	23
<b>4.2</b>	<b>Soluções <math>&lt; O(N^2)</math></b>	<b>24</b>
4.2.1	Solução última linha	29
4.2.2	Solução limpeza de matriz	33
4.2.3	Solução diagonal	37
4.2.4	Solução bit-paralelo	44
4.2.5	Solução bit-paralelo com limites	51
<b>4.3</b>	<b>Discussão</b>	<b>55</b>
<b>5</b>	<b>RESULTADOS PRÁTICOS</b>	<b>57</b>
<b>5.1</b>	<b>Problema Longest Common Subsequence - LCS0</b>	<b>58</b>
<b>5.2</b>	<b>Problema Aibohphobia - AIBOHP</b>	<b>60</b>
<b>5.3</b>	<b>Palindrome 2000 - IOIPALIN</b>	<b>62</b>
<b>5.4</b>	<b>Problema X-Men - XMEN</b>	<b>64</b>
<b>6</b>	<b>CONCLUSÃO</b>	<b>67</b>
	<b>REFERÊNCIAS</b>	<b>69</b>

# 1 Introdução

A essência do problema de encontrar a maior subsequência comum (LCS, *Longest Common Subsequence*) entre os elementos de um conjunto de sequências de símbolos de qualquer natureza é determinar o grau de similaridade existente entre elas. Talvez o exemplo mais impactante para sua utilização seja a comparação de sequenciamentos genéticos (ALSMADI; NUSER, 2012), mas a LCS tem outras várias aplicações, tais como a comparação de textos (MYERS, 1986), comparação de códigos de programas de computador (HUNT; MCILROY, 1976), similaridade de imagens ou texturas (WANG, 2003), correção ou sinalização de erros de digitação e busca de documentos (KENT; SALIM, 2010). Outros problemas da área de Ciência da Computação derivam soluções a partir da LCS ou estão intimamente relacionados, como por exemplo: distância de Levenshtein ou distância de edição (LEVENSHTEIN, 1966; WAGNER; FISCHER, 1974), RFLCS (*Repetition Free Longest Common Subsequence Problem*) (TJANDRAATMADJA, 2010) e LIS (*Longest Increasing Subsequence*) (CROCHEMORE; PORAT, 2008).

Definimos como sequência o encadeamento ordenado de objetos naturais representados por símbolos, comumente caracteres alfa-numéricos. Como exemplo, as letras A, C, G e T representam os quatro nucleotídeos de uma cadeia de DNA (*i.e.* as bases adenina, citosina, guanina e timina), e sequências como ACCCGGTTT representam uma sequência de DNA. Uma subsequência qualquer pode ser obtida extraindo-se zero ou mais caracteres da sequência original, mantendo seu ordenamento. Por exemplo, as sequências ACCCGGTTT, ACC, AGTT e ACGT são todas subsequências de ACCCGGTTT. Por extensão, a sequência vazia é subsequência de todas as sequências da natureza, e toda sequência é subsequência de si própria.

Definimos subsequência comum a duas sequências como uma sequência que é subsequência de ambas as sequências. Uma subsequência comum máxima é aquela que, dentre todas as subsequências comuns, tem o maior comprimento. Para efeito do estudo da LCS, podemos abstrair completamente o significado de cada símbolo em uma sequência qualquer e manter o foco somente em encontrar a subsequência máxima entre sequências de caracteres. Por esta razão, é comum definir o LCS como o problema de determinar a maior subsequência de caracteres comuns a duas sequências de caracteres, também chamadas de strings.

O problema da LCS tem duas vertentes: *Multiple Longest Common Subsequence* (MLCS) e *Length of the Longest Common Subsequence* (LLCS). MLCS é o problema de encontrar a maior subsequência entre um conjunto de subsequências (WANG; KORKIN; SHANG, 2009). LLCS é o problema de encontrar apenas o comprimento da maior sub-

sequência entre duas sequências, e é particularmente útil para se medir numericamente o grau de similaridade entre duas sequências.

O problema da LCS também possui duas variantes: o caso em que as sequências comparadas tem um alfabeto finito e limitado (CHIN; POON, 1991), e o caso em que o alfabeto é finito, mas não conhecido previamente (CROCHEMORE; PORAT, 2008). O primeiro caso permite a otimização de códigos explorando particularidades da sequência e do alfabeto, enquanto o segundo caso requer a uma solução mais genérica para o problema.

O problema da LCS é um problema de natureza combinatória com complexidade de tempo quadrática, por essa razão, este problema possui dificuldade crescente de realização prática com o crescimento do tamanho das strings comparadas. Em Agosto do ano de 2012 foi proposto no repositório *Sphere online Judge (SPOJ)* o problema de calcular o comprimento da LCS de duas strings com até 50.000 caracteres minúsculos em menos de 676 ms usando um computador de uso popular. Em fins do ano 2014, apenas 18 programadores em todo o mundo, num total da ordem de 3800 tentativas, tinham realizado a tarefa. Este fato ressaltou que as soluções tradicionais apontadas pela literatura não eram capaz de lidar com o problema de forma satisfatória e que novos algoritmos são necessários para o problema da LCS para strings muito longas. Este trabalho foi motivado pelo desafio proposto no *SPOJ* e consiste no estudo da LCS pela implementação de algoritmos que busquem melhorar a eficiência do cálculo do seu comprimento, buscando comprovar resultados teóricos apresentados em diversos trabalhos já publicados. Ao longo do desenvolvimento deste trabalho, o estudo minucioso das propriedades da matriz LCS conduziu à criação de dois novos algoritmos cujas implementações denominamos de limpeza de matriz e de diagonal. O primeiro teve como princípio o mapeamento de pontos de casamento dos caracteres em uma estrutura de dados em listas e a redução sucessiva por eliminação sistemática destes dados associado-os às camadas da LCS e o segundo é o resultado otimizado da aplicação do primeiro algoritmo em duas metades da matriz LCS dividida através da sua diagonal.

O restante deste trabalho está organizado da seguinte maneira: no [Capítulo 2](#) apresentamos um referencial teórico para sustentar o desenvolvimento dos capítulos seguintes; no [Capítulo 3](#) delineamos o ambiente utilizado para as implementações; no [Capítulo 4](#) apresentamos as implementações realizadas, seus resultados e considerações particulares; no [Capítulo 5](#) apresentamos resultados práticos obtidos ao submeter os códigos ao *Sphere Online Judge (SPOJ)*, visando comprovar o desempenho obtido quando comparado com soluções apresentadas por usuários de todo o mundo; por fim, no [Capítulo 6](#) apresentamos a conclusão do trabalho e apontamos uma possível continuidade na pesquisa buscando soluções cada vez mais eficientes.

## 2 Referencial Teórico

Definimos  $S = \{S_1, S_2, S_3, \dots, S_m\}$  como uma sequência de caracteres de comprimento  $m$ , onde  $S_k$  representa o  $k$ -ésimo elemento de  $S$ ,  $S_{i:j}$  representa a subsequência dos elementos  $i$  até  $j$  de  $S$ ,  $S_{k:m}$  representa o sufixo de  $S$  a partir do elemento  $k$ , e  $S_{1:k}$  representa o prefixo da sequência  $S$  até o elemento  $k$ .

Dadas duas sequências  $X$  e  $Y$  de comprimento  $m$  e  $n$ , respectivamente, onde  $X = \{X_1, X_2, \dots, X_m\}$  e  $Y = \{Y_1, Y_2, \dots, Y_n\}$  definimos  $LCS(X, Y)$  como a máxima subsequência comum entre  $X$  e  $Y$ . A equação de recorrência para o cálculo de  $LCS(X, Y)$  pode ser deduzida a partir de duas propriedades:

1. Se  $X_1 = Y_1$ , então  $LCS(X, Y)$  é a concatenação de  $X_1$  com  $LCS(X_{2:m}, Y_{2:n})$ .
2. Se  $X_1 \neq Y_1$ , então  $LCS(X, Y) = \max(LCS(X_{2:m}, Y), LCS(X, Y_{2:n}))$ .

Estas propriedades constituem a sub-estrutura ótima para a resolução continuada do primeiro caractere de cada string com a aplicação recursiva dessas mesmas propriedades ao sufixo remanescente. As mesmas propriedades são válidas quando as strings são analisadas do fim para o começo, resolvendo o último caractere e aplicando a recursão ao prefixo remanescente:

1. Se  $X_m = Y_n$ , então  $LCS(X, Y)$  é a concatenação de  $LCS(X_{1:m-1}, Y_{1:n-1})$  com  $X_m$ .
2. Se  $X_m \neq Y_n$ , então  $LCS(X, Y) = \max(LCS(X_{1:m-1}, Y), LCS(X, Y_{1:n-1}))$ .

As definições matemáticas das recursões da  $LCS$  por sufixo e prefixo são respectivamente apresentadas nas Equações 2.1 e 2.2:

$$LCS(X, Y) = \begin{cases} \emptyset & \text{se } m = 0 | n = 0 \\ X_1 + LCS(X_{2:m}, Y_{2:n}) & \text{se } X_1 = Y_1 \\ \max(LCS(X, Y_{2:n}), LCS(X_{2:m}, Y)) & \text{se } X_1 \neq Y_1 \end{cases} \quad (2.1)$$

$$LCS(X, Y) = \begin{cases} \emptyset & \text{se } m = 0 | n = 0 \\ LCS(X_{1:m-1}, Y_{1:n-1}) + X_m & \text{se } X_m = Y_n \\ \max(LCS(X_{1:m-1}, Y), LCS(X, Y_{1:n-1})) & \text{se } X_m \neq Y_n \end{cases} \quad (2.2)$$



As Equações 2.1 e 2.2 podem ser adaptadas para a recursão da LLCS, como apresentado nas equações 2.3 e 2.4, respectivamente:

$$|LCS(X, Y)| = \begin{cases} 0 & \text{se } m = 0 | n = 0 \\ 1 + |LCS(X_{2:m}, Y_{2:n})| & \text{se } X_1 = Y_1 \\ \max(|LCS(X_{2:m}, Y)|, |LCS(X, Y_{2:n})|) & \text{se } X_1 \neq Y_1 \end{cases} \quad (2.3)$$

$$|LCS(X, Y)| = \begin{cases} 0 & \text{se } m = 0 | n = 0 \\ |LCS(X_{1:m-1}, Y_{1:n-1})| + 1 & \text{se } X_1 = Y_1 \\ \max(|LCS(X_{1:m-1}, Y)|, |LCS(X, Y_{1:n-1})|) & \text{se } X_1 \neq Y_1 \end{cases} \quad (2.4)$$

A prova destas recursões é apresentada por [Cormen et al. \(2009\)](#).

## 3 Ambiente

As implementações realizadas neste trabalho tem o protótipo "*int LCS(char \* X, char \* Y)*", onde *X* e *Y* são as entradas da função *LCS* e representam duas sequências de caracteres alfabéticos minúsculos, sem espaços. A [Tabela 1](#) descreve os casos de teste de entrada da função. O retorno da função é um número inteiro com o valor do comprimento da LCS correspondente. O [Código 3.1](#) apresenta o programa elementar para a chamada da função *LCS(X, Y)*.

Teste	Arquivo	Tamanho <i>X</i>	Tamanho <i>Y</i>	Descrição
1	lcs50krnd.txt	50000	50000	<i>X</i> e <i>Y</i> aleatórios
2	lcs50kid.txt	50000	50000	<i>X</i> = <i>Y</i> aleatório
3	lcs50krev.txt	50000	50000	<i>Y</i> = palíndromo de <i>X</i>
4	lcs50kaa.txt	50000	50000	<i>X</i> = $a^m$ <i>Y</i> = $a^n$
5	lcs50kab.txt	50000	50000	<i>X</i> = $a^m$ <i>Y</i> = $b^n$
6	lcs25krnd.txt	25000	50000	<i>X</i> e <i>Y</i> aleatórios

Tabela 1 – Casos de teste.

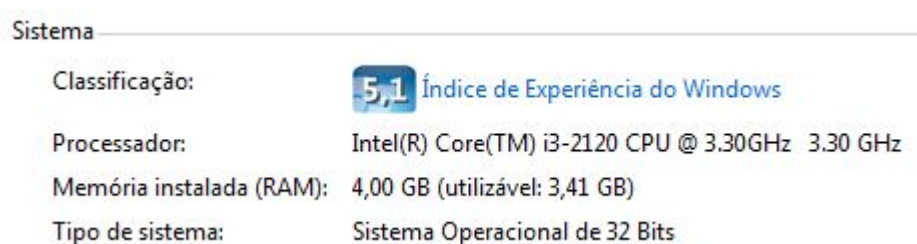
```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAXs 50001           // tamanho maximo da string
5 #define INF 0x7FFFFFFF     // infinito, maximo int 32 bits
6 #define max(a,b) a<b?b:a   // macro max(a,b)
7 #define min(a,b) a<b?a:b   // macro min(a,b)
8
9 typedef unsigned int uint;  // tipo inteiro 32 bits sem sinal
10
11 int LCS(char *X, char *Y) {
12     ...
13 }
14
15 int main() {
16     char X[MAXs],Y[MAXs];
17     scanf("%s %s",X,Y);
18     printf("%d\n",LCS(X,Y));
19     return 0;
20 }

```

Código 3.1 – Programa elementar para chamada da função *LCS(X, Y)*.

Todos os resultados experimentais apresentados neste trabalho utilizam programas desenvolvidos na linguagem C e foram executados em um computador doméstico com o Sistema Operacional Windows 7, cujas configurações são apresentadas na [Figura 1](#). Os tempos de execução são obtidos em segundos tomando-se o menor valor mais recorrente para minimizar a interferência da característica multi-tarefa do sistema operacional. Este programa em linguagem C executa a leitura das strings  $X$  e  $Y$  e em seguida faz uma chamada à função  $LCS(X, Y)$ , que calcula o comprimento da LCS e retorna o seu valor para ser impresso no terminal. Esta estrutura se repetirá em todos os experimentos, substituindo-se apenas o conteúdo da função  $LCS(X, Y)$ .



Sistema	
Classificação:	5,1 Índice de Experiência do Windows
Processador:	Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz 3.30 GHz
Memória instalada (RAM):	4,00 GB (utilizável: 3,41 GB)
Tipo de sistema:	Sistema Operacional de 32 Bits

Figura 1 – Configuração do computador utilizado nos experimentos.

## 4 Implementações para o Problema da LCS

As soluções apresentadas neste trabalho foram ordenadas da mais simples para as mais complexas. Algumas delas poderiam ser simplesmente omitidas, dado que a inviabilidade das mesmas já está exaustivamente comprovada. Entretanto, para efeito didático, achamos conveniente introduzi-las neste capítulo a fim de se obter números comparativos em termos de tempo de execução.

Usaremos como exemplo ao longo do texto as strings CAGAXTCACGACGTA e TCGTGTTGCGTYACT, que possuem duas LCS de comprimento 8, uma delas CGTGCGTA e a outra CGTTCGACT.

### 4.1 Soluções $\geq O(N^2)$

As soluções a seguir são convencionais por serem fundamentadas diretamente nas Equações 2.1 a 2.4, cujo princípio é a comparação de caracteres. Estas soluções apresentam um limite assintótico inferior de  $O(N^2)$  pois realizam ao menos uma vez a comparação, dois a dois, de todos os caracteres das strings de entrada.

#### 4.1.1 Solução recursiva

O Código 4.1 apresenta a solução recursiva da LCS, que nada mais é do que a transcrição direta da Equação 2.3, apresentada no Capítulo 2. Embora matematicamente correta, é exageradamente custosa. É fácil perceber que em strings sem correspondência, quando a recursão chamar  $L(X_{2:m}, Y)$  e  $L(X, Y_{2:n})$ , ambas chamarão  $LCS(X_{2:m}, Y_{2:n})$  causando uma superposição desnecessária que se propagará exponencialmente até o fim da execução (CORMEN et al., 2009). Apenas no melhor caso, quando uma das strings é prefixo da outra, haverá uma quantidade de chamadas recursivas igual ao comprimento da menor string. A complexidade desta solução é  $O(2^{m+n})$  ou  $O(2^{2N})$ .

```

1 int LCS(char *X, char *Y){
2     if (*X==0 || *Y==0) return 0;
3     if (*X==*Y) return (1+LCS(X+1, Y+1));
4     else return max(LCS(X, Y+1), LCS(X+1, Y));
5 }

```

Código 4.1 – Solução recursiva.

Para um par de strings de apenas 12 caracteres diferentes entre si (*i.e.*  $LCS=0$ ), esta solução levou 45 segundos para gerar o resultado. Para efeito comparativo, consideraremos o tempo de resposta como infinito para os casos de testes descritos no [Capítulo 3](#).

### 4.1.2 Solução matricial

A solução matricial se utiliza da técnica de programação dinâmica, que consiste em armazenar em tabelas, numa sequência lógica apropriada, os resultados das recursões previamente obtidos para que sejam reutilizados quando necessário ([CORMEN et al., 2009](#)). Considerando que uma string qualquer com  $n$  caracteres tem  $n + 1$  prefixos, incluindo o prefixo vazio, a combinação dois a dois de todos os prefixos de duas strings  $X$  e  $Y$  com  $m$  e  $n$  caracteres possui  $(m + 1) \times (n + 1)$  possibilidades. Desta forma, os valores de LCS calculados para cada combinação de prefixos podem ser armazenado em uma matriz  $L$  com  $m + 1$  colunas e  $n + 1$  linhas, totalizando  $(n + 1) \times (m + 1)$  elementos. Considerando-se que o comprimento da LCS na combinação de qualquer prefixo com o prefixo vazio é igual a zero, tais combinações são previamente preenchidas restando calcular apenas  $m \times n$  possibilidades. O pressuposto elementar da programação dinâmica é que os novos elementos a serem calculados tenham sua dependência previamente calculada, o que pode ser obtido seguindo uma ordem específica de acesso à matriz.

Desta forma, ao calcular o elemento  $L[i][j]$  numa matriz organizada por prefixos, o resultado dependerá de  $L[i - 1][j]$ ,  $L[i - 1][j - 1]$  e  $L[i][j - 1]$ , conforme [Equação 2.3](#). O último elemento a ser calculado estará na posição inferior direita e conterá o comprimento da LCS.

Como exemplo, observamos na [Figura 2](#) que os prefixos  $X_{1:5}$  e  $Y_{1:3}$  possuem LCS com comprimento igual a 2, previamente calculado e armazenado no elemento  $L[3][5]$ . No momento do cálculo do elemento  $L[4][6]$  é verificado que  $X_6$  e  $Y_4$  são iguais, e assim vale a segunda linha da [Equação 2.3](#), portanto  $L[4][6] = 1 + L[3][5]$ . Já no momento do cálculo do elemento  $L[6][10]$ , é verificado que  $X_{10}$  e  $Y_6$  são diferentes. Assim, vale a terceira linha da [Equação 2.3](#), e  $L[6][10]$  deve receber o maior valor entre  $L[6][9]$  e  $L[5][10]$ . Neste exemplo,  $L[6][10]$  recebe o valor 4 previamente calculado e armazenado no elemento  $L[5][10]$ . Ao fim do processo, o elemento  $L[15][15]$  conterá o comprimento da  $LCS(X, Y)$ .

O mesmo processo pode ser igualmente repetido seguindo a lógica por prefixo da [Equação 2.4](#), onde os elementos da matriz  $L$  são calculados da direita para a esquerda e de baixo para cima, e o resultado do comprimento da LCS é armazenado em  $L[0][0]$ .

A implementação da solução matricial em linguagem C é ligeiramente mais eficiente se for adotada a recursão por prefixo da [Equação 2.4](#), pois strings em C são indexadas a partir de 0 e o fim de uma string é marcada com o caractere especial NULL, com isso, teremos naturalmente a simulação do sufixo vazio neste caso.

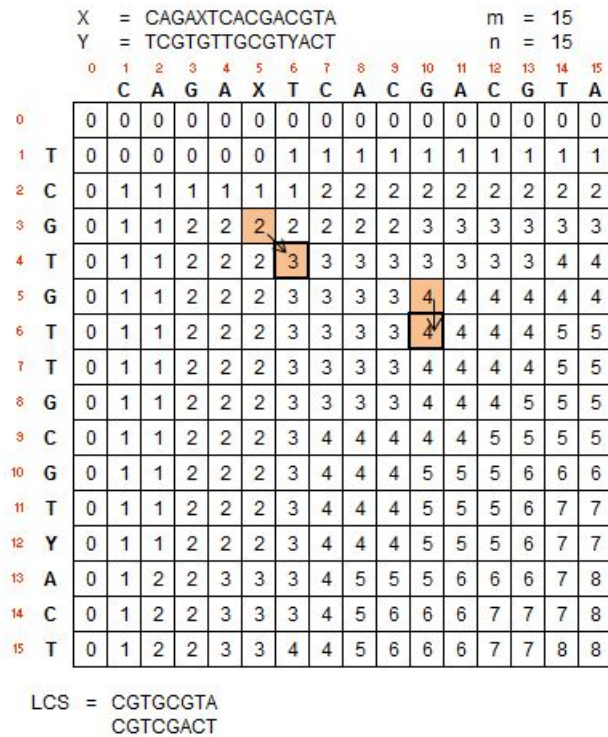


Figura 2 – Matriz LCS preenchida por programação dinâmica, considerando a lógica de recursão por sufixo, ordenada por prefixo. Os valores são calculados sequencialmente da esquerda para a direita e de cima para baixo.

A solução matricial tem complexidade de tempo e de espaço  $O(mn)$  ou  $O(N^2)$ , e por essa razão não é viável para a comparação de strings muito longas. Ao considerarmos  $m$  e  $n$  iguais a 50.000 caracteres, a memória necessária ultrapassaria 4GB, o que é impossível de se endereçar em um Sistema Operacional de 32 bits. Por não ser possível resolver uma LCS com 50000 caracteres com este algoritmo no ambiente utilizado, truncamos as strings de teste em diversos tamanhos e as medições do tempo de execução foram utilizadas para estimar por interpolação o tempo que seria necessário para o tamanho desejado.

Na máquina de referência (Figura 1), o comprimento máximo viável para cada string foi de 22.000 caracteres, consumindo toda a memória disponível em alocação estática global e executando em 4,45 segundos. Por interpolação quadrática estimamos que o tempo necessário, caso fosse possível a execução nesta máquina, seria de 21,81 segundos, conforme apresentado na Tabela 2.

A implementação da solução matricial é apresentada no Código 4.2. Para viabilizar o uso dos casos de teste foi necessário efetuar pequenas modificações no código: introduzimos uma variável LIMs (Linha 1) para limitar o tamanho da matriz de programação dinâmica alocada (Linha 2) e truncar as strings de entrada (Linha 9). O restante do código é essencialmente a programação dinâmica que preencherá todos os elementos dessa matriz (CORMEN et al., 2009).

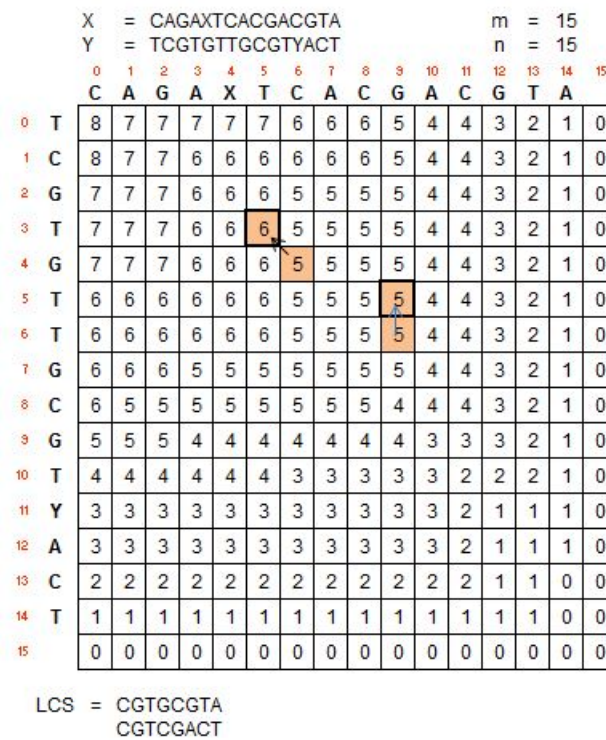


Figura 3 – Matriz LCS preenchida por programação dinâmica, considerando a lógica de recursão por prefixo, ordenada por sufixo. Os valores são calculados sequencialmente da direita para a esquerda e de baixo para cima.

Comprimento	tempo real(s)	tempo interpolado(s)
1000	0,01	0,01
5000	0,20	0,22
10000	0,81	0,87
15000	2,01	1,96
20000	3,25	3,49
22000	4,45	4,22
50000	-	21,81

Tabela 2 – Estimativa de tempo de execução para string com 50.000 caracteres.

```

1 #define LIMs 22000
2 int L[LIMs+1][LIMs+1];
3 typedef unsigned int uint;
4
5 int LCS(char *X, char *Y) {
6     register int i,j;
7     uint k,m,n;
8
9     X[LIMs]=0;Y[LIMs]=0;

```

```

10
11     for (m=0; X[m]; m++);
12     for (n=0; Y[n]; n++);
13
14     for (i=n; i>=0; i--) {
15         for (j=m; j>=0; j--) {
16             if (X[j]=='\0' || Y[i]=='\0') L[i][j]=0;
17             else if (X[j]==Y[i])
18                 L[i][j]=L[i+1][j+1]+1;
19             else
20                 L[i][j]=max(L[i+1][j], L[i][j+1]);
21         }
22     }
23     return L[0][0];
24 }

```

Código 4.2 – Solução Matricial

### 4.1.3 Solução matricial em espaço linear

Analisando a solução matricial anterior, observamos que cada elemento calculado depende apenas de duas linhas da matriz, a anterior e a atual. Na verdade, cada elemento  $L[i][j]$  depende apenas de outros três elementos, graças à ordem de preenchimento da matriz: o elemento anterior  $L[i][j+1]$ , o elemento adjacente inferior  $L[i+1][j]$  e o elemento anterior ao adjacente inferior  $L[i+1][j+1]$ .

Diante deste fato, com o uso de duas variáveis auxiliares, apenas uma única linha é necessária para armazenar os resultados. Para cada novo elemento a ser calculado, o resultado será armazenado temporariamente nestas variáveis auxiliares, que por sua vez irão gradativamente sobrescrever a linha atual. Ao final do processo, restará apenas uma linha equivalente à primeira linha da matriz no algoritmo matricial e o resultado estará na primeira posição desta linha. A programação dinâmica precisará apenas de um vetor, que poderá ser alocado diretamente na função, melhorando a solução em termos de portabilidade e modularidade. Enquanto o espaço necessário para os casos de teste na solução matricial é de 10GB, a solução de espaço linear requer apenas 200KB. Esta economia de espaço permite a aplicação do algoritmo em strings maiores, mas como a complexidade ainda é  $O(N^2)$ , o tempo de execução permanece insatisfatório. Apresentamos a implementação da solução matricial em espaço linear no [Código 4.3](#).

```

1 int LCS(char *X, char *Y) {
2     int i, j, k1, k2, m, n;
3

```



```
4     for (m=0; X[m]; m++);
5     for (n=0; Y[n]; n++);
6
7     int L[m+1];
8
9     for (i=n; i>=0; i--) {
10        k1=0;
11        for (j=m; j>=0; j--) {
12            if (X[j]=='\0' || Y[i]=='\0') k2=0;
13            else if (X[j]==Y[i])
14                k2=1+L[j+1];
15            else
16                k2=max(L[j], k1);
17            L[j+1]=k1;
18            k1=k2;
19        }
20        L[0]=k1;
21    }
22    return L[0];
23 }
```

Código 4.3 – Solução matricial em espaço linear.

Para os casos de teste, este programa executou em 13,05 segundos. Este tempo é bem menor do que o tempo estimado da solução matricial plena, pois o uso de memória de forma intensiva força o processador e o sistema operacional a gerenciar memória *cache* e memória virtual com mais frequência, atrasando a execução do programa em si.

## 4.2 Soluções $< O(N^2)$

As soluções apresentadas até o momento requerem que todos os caracteres que compõe as strings sejam comparados dois a dois, impondo o limitador intransponível de  $O(N^2)$  no tempo de execução. Como a LCS é composta apenas de caracteres que ocorrem simultaneamente nas duas strings, algoritmos que considerarem somente o conjunto de pontos de coincidência na matriz serão mais eficientes, como mostrado na [Figura 4](#), em que apenas 41 dos 225 pontos existentes precisam ser considerados na busca da LCS.

Denominamos  $L$  o conjunto de pontos da matriz da programação dinâmica, e  $R$  o conjunto de coincidências de caracteres em um par de strings qualquer. O tamanho do conjunto  $L$  é dependente apenas do comprimento das strings comparadas e é igual a  $m \times n$ , onde  $m$  e  $n$  são o comprimento das strings  $X$  e  $Y$ , respectivamente. O tamanho do conjunto  $R$  é variável e dependente do conteúdo destas strings. Strings formadas por

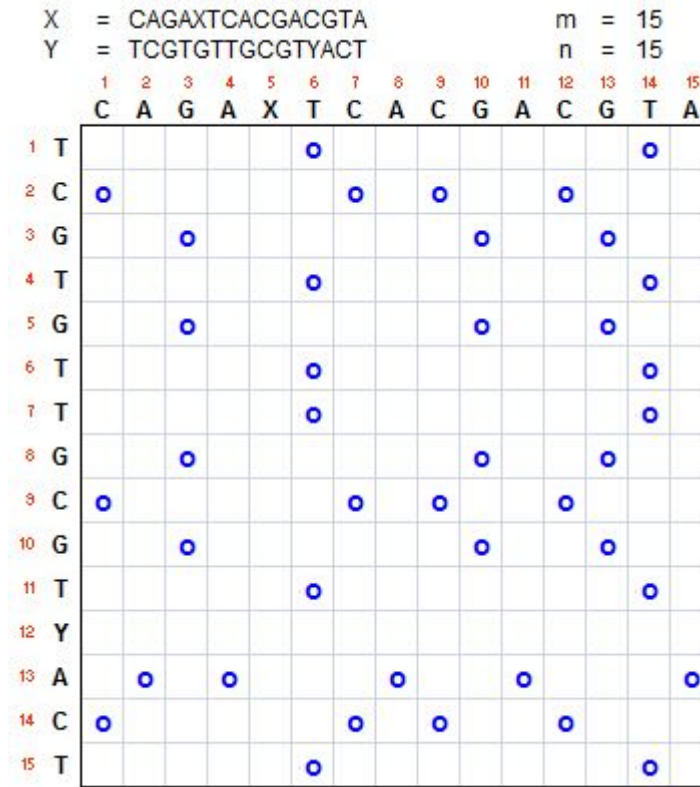


Figura 4 – Mapa de coincidências comum às strings X e Y.

alfabeto grande e distribuição uniforme dos caracteres possuem conjunto  $R$  muito pequeno em relação ao conjunto  $L$ , enquanto que strings com alfabeto pequeno ou com distribuição não uniforme geram conjuntos  $R$  maiores (CROCHEMORE; ILIOPOULOS; PINZON, 2003). Em qualquer destas hipóteses, o conjunto  $R$  é subconjunto próprio do conjunto  $L$ , e esta característica é explorada no desenvolvimento de algoritmos eficientes para a busca da LCS. Sejam  $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_{|\Sigma|}\}$  o alfabeto que compõe as strings  $X$  e  $Y$ ,  $Q_X^{\sigma_i}$  a quantidade de ocorrência do caractere  $\sigma_i$  em  $X$  e  $Q_Y^{\sigma_i}$  a quantidade de ocorrência do caractere  $\sigma_i$  em  $Y$ , o tamanho do conjunto  $R$  é calculado pela Equação 4.1 que é igual ao somatório dos produtos da quantidade de ocorrência de cada caractere em cada uma das strings.

$$|R| = \sum_{\alpha=\sigma_1}^{\sigma_{|\Sigma|}} Q_X^\alpha \times Q_Y^\alpha \tag{4.1}$$

Para os casos de strings que possuem distribuição uniforme de caracteres do alfabeto, podemos estimar o tamanho de  $R$  pela Equação 4.2.

$$|R| \sim \frac{|L|^2}{|\Sigma|} \tag{4.2}$$

A utilização do conjunto  $R$  como base para a solução do problema da LCS substitui

a abordagem da comparação de caracteres pela abordagem gráfica. Nesta nova abordagem, as coincidências de caracteres das strings  $X$  e  $Y$  agora são apresentadas como pares ordenados  $(i, j)$  sobre o plano, mantendo-se uma correspondência posicional com a matriz da programação dinâmica original. Uma subsequência somente pode ser formada por pontos do conjunto  $R$  ordenados de forma crescente, ou seja, dados dois pontos  $R_1[i_1][j_1]$  e  $R_2[i_2][j_2]$ ,  $R_2 > R_1$  se e somente se  $i_2 > i_1$  e  $j_2 > j_1$ . Se  $i_1 = i_2$  ou  $j_1 = j_2$ , os mesmos estarão dispostos na mesma linha ou mesma coluna, respectivamente, e não podem pertencer a uma subsequência ao mesmo tempo. Nestes casos, aquele ponto que tiver o índice oposto menor, obstrui o maior tornando-o irrelevante.

Hirschberg (1977) demonstrou que, escolhido um ponto qualquer na matriz de coincidências, este ponto dividirá a matriz em 4 quadrantes, como mostrado na Figura 5. Na busca da LCS, todos os pontos dos quadrantes 1, 2 e 3 são irrelevantes em relação a este ponto porque são menores ou iguais ao ponto escolhido e podem ser desprezados. Caso o ponto escolhido seja parte da LCS, somente os pontos dentro do quadrante 4 devem ser considerados para sua continuidade, seguindo a ordem crescente mencionada anteriormente.

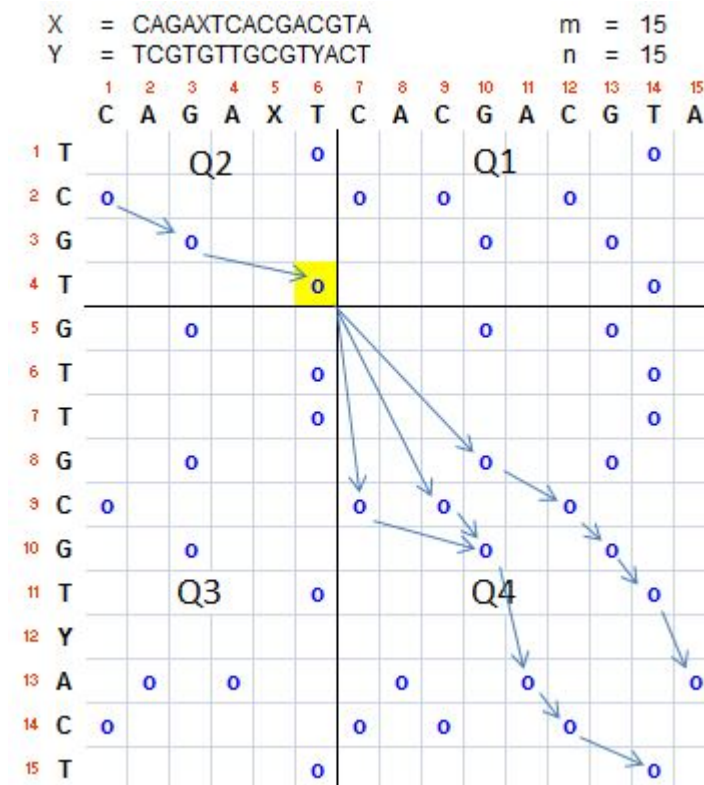
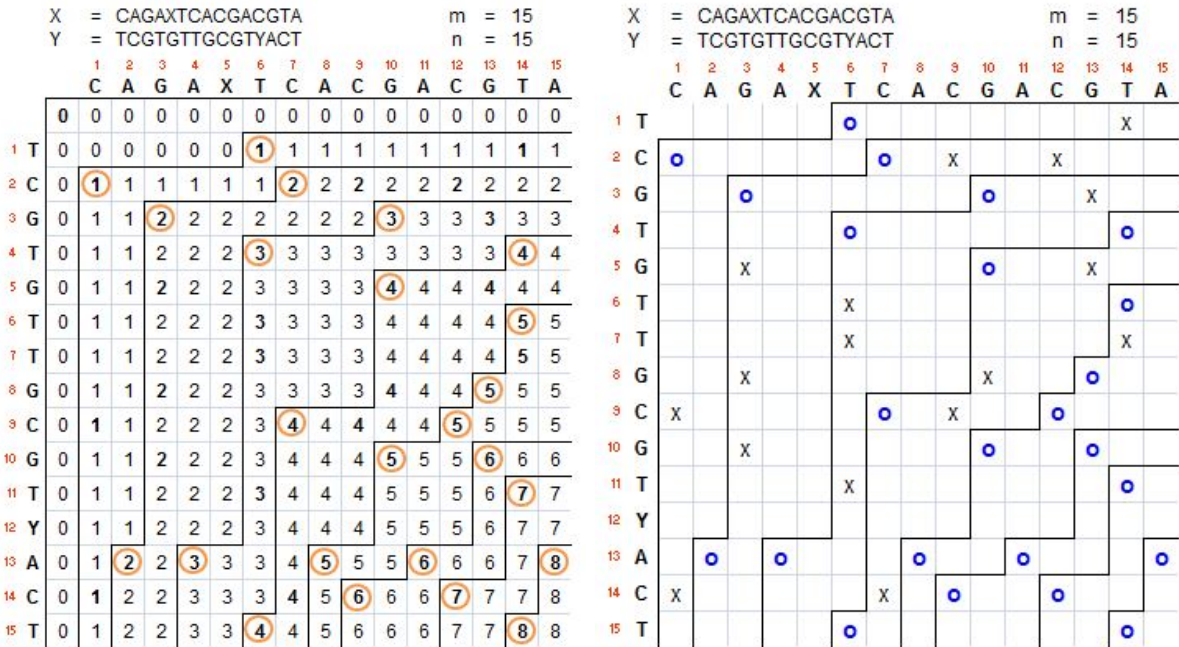


Figura 5 – Mapa de coincidências de caracteres das strings  $X$  e  $Y$ . O ponto em  $(4,6)$  domina os pontos dentro do quadrante 4, que são candidatos a dar continuidade à LCS.

Chamamos de conjunto dominante  $D$ , os pontos do conjunto  $R$  que são relevantes

na formação da LCS. Ao demarcar linhas de fronteira na matriz de programação dinâmica, verificamos que o conjunto  $D$  é exatamente composto pelos pontos que marcam os vértices externos destas linhas, como ilustrado na Figura 6a. Na Figura 6b verificamos que apenas 24 dos 41 pontos do conjunto  $R$  são relevantes para a busca da LCS.



(a) Matriz Programação Dinâmica.

(b) Conjunto D.

Figura 6 – Matriz de Programação Dinâmica e mapa de coincidências com pontos dominantes marcados com círculos e os pontos irrelevantes com x.

Ao numerarmos as linhas de fronteiras através de um índice  $k$ , chamamos os pontos dominantes que delimitam a fronteira  $k$  de  $k$ -Dominantes ou  $D^k$ . A distribuição do conjunto  $D$  sobre a matriz é resultado das seguintes propriedades: somente 0 ou 1 fronteira é acrescentada a cada linha da matriz, e o ponto relevante  $D^k$  a cada nova linha é sempre aquele que está mais próximo e à direita da fronteira  $k - 1$  na linha anterior. Consideramos a existência universal da fronteira  $D^0$  que representa a subsequência vazia e é observada na Figura 6a.

Assim, podemos verificar na Figura 6b que:

- O ponto  $R[1][6]$  delimitou a fronteira 1 na primeira linha por ser o mais a direita da fronteira  $D^0$  e descartou o ponto  $R[1][14]$ ;
- O ponto  $R[2][1]$  moveu a fronteira 1 para a coluna 1;
- O ponto  $R[2][7]$  delimitou a fronteira 2 e descartou os pontos  $R[2][9]$  e  $R[2][12]$ ;
- A cada nova linha, as fronteiras existentes podem ser movidas para a esquerda, e no máximo uma nova fronteira pode ser criada.

A região da distribuição dos pontos  $D$  começa no canto superior esquerdo e é delimitada por duas arestas, simétricas à diagonal, com inclinação calculada pela razão  $\frac{N}{|\Sigma|}$ . Na Figura 7 vemos as duas retas que delimitam a região onde o conjunto  $D$  é mais provável de estar distribuído.

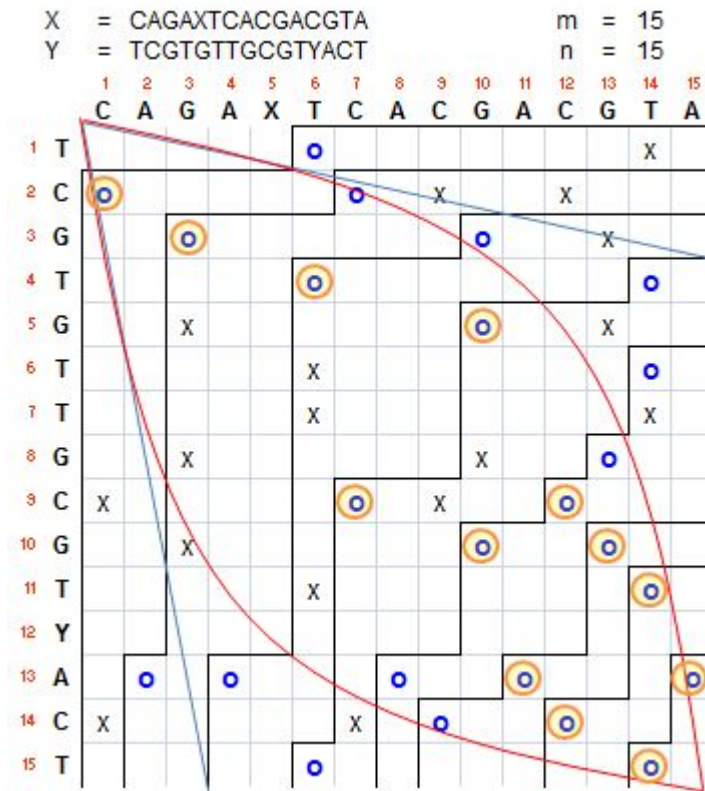


Figura 7 – Mapa de distribuição do conjunto de pontos. Conjunto  $D$  em círculos azuis e conjunto  $K$  demarcados com círculo laranja.

As seqüências de pontos dominantes e crescentes que rapidamente atingem a lateral direita ou a base da matriz podem ser interpretadas como seqüências comuns que foram abortadas por não poderem mais ser prolongadas. Todos os pontos  $D^k$  que não podem dominar pontos  $D^{k+1}$  poderiam ser desconsiderados, restando apenas o conjunto  $K$  com todos os pontos que podem pertencer a uma LCS. A área delimitada pelos dois arcos mostrados na Figura 7 representa, empiricamente, a área provável que contém o conjunto  $K$ .

A Equação 4.3 mostra a relação entre estes diversos conjuntos.

$$LCS \subseteq K \subseteq D \subseteq R \subseteq L \tag{4.3}$$

As implementações a seguir se utilizam das propriedades delineadas nesta seção, buscando identificar o menor subconjunto de  $R$  que contém  $K$  na determinação do comprimento da LCS.

### 4.2.1 Solução última linha

A solução matricial em espaço linear chama a atenção pelo fato da última linha calculada possuir uma relevância especial. Todas as fronteiras de transição da LCS cortam a última linha calculada. As transições de uma unidade na última linha tem uma correlação direta com o momento que ocorre uma igualdade entre os caracteres das strings. Ao observar as transições de uma unidade nos comprimentos da LCS em todas as linhas, percebemos que existem linhas de fronteira separando claramente as regiões de transição, como mostrado na [Figura 8](#).

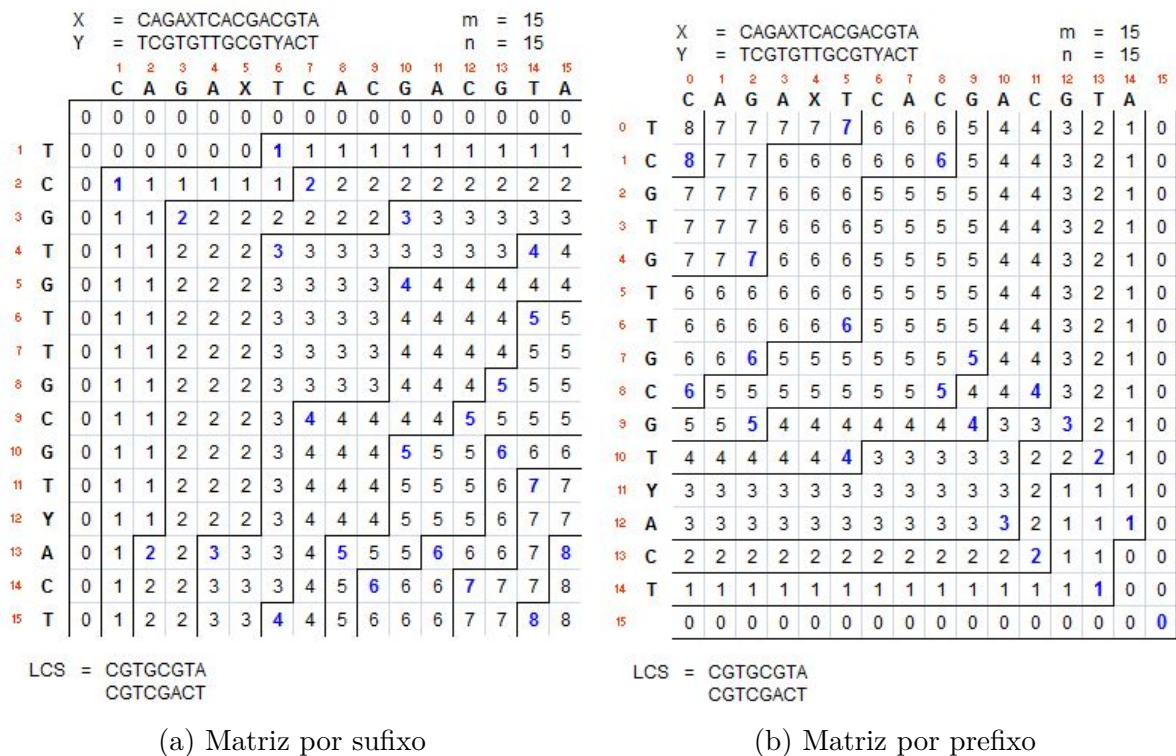


Figura 8 – Matriz preenchida por programação dinâmica delineando as fronteiras de mudança de uma unidade no comprimento da LCS.

A solução última linha explora a possibilidade de se obter a sequência da última linha da matriz, inferindo suas transições a partir das coincidências relevantes dos caracteres, linha após linha ([HUNT; SZYMANSKI, 1977](#)).

A implementação usa listas encadeadas para indexar a posição dos caracteres nas strings e vetores auxiliares para armazenar a primeira posição de um caractere do alfabeto na lista encadeada sem que seja necessário a comparação entre eles. A cada linha, verificamos quais fronteiras podem ser melhoradas e qual é melhor candidato para uma nova fronteira, descartando todas as demais coincidências até o fim da linha. Quando uma fronteira não pode ser mais melhorada, esta posição pode ser descartada das listas encadeadas e fronteiras são atualizadas somente no intervalo que ainda pode ser melhorado. Em suma, o algoritmo inicializa com todas as fronteiras situadas no infinito e vai puxando-

as em direção ao início da matriz conforme ocorrem coincidências dos caracteres nas duas strings comparadas. É importante reiterar que, a cada linha processada, zero ou uma fronteira é trazida do infinito e adicionada a lista de fronteiras. Naturalmente o número final de fronteiras refletirá o comprimento da LCS.

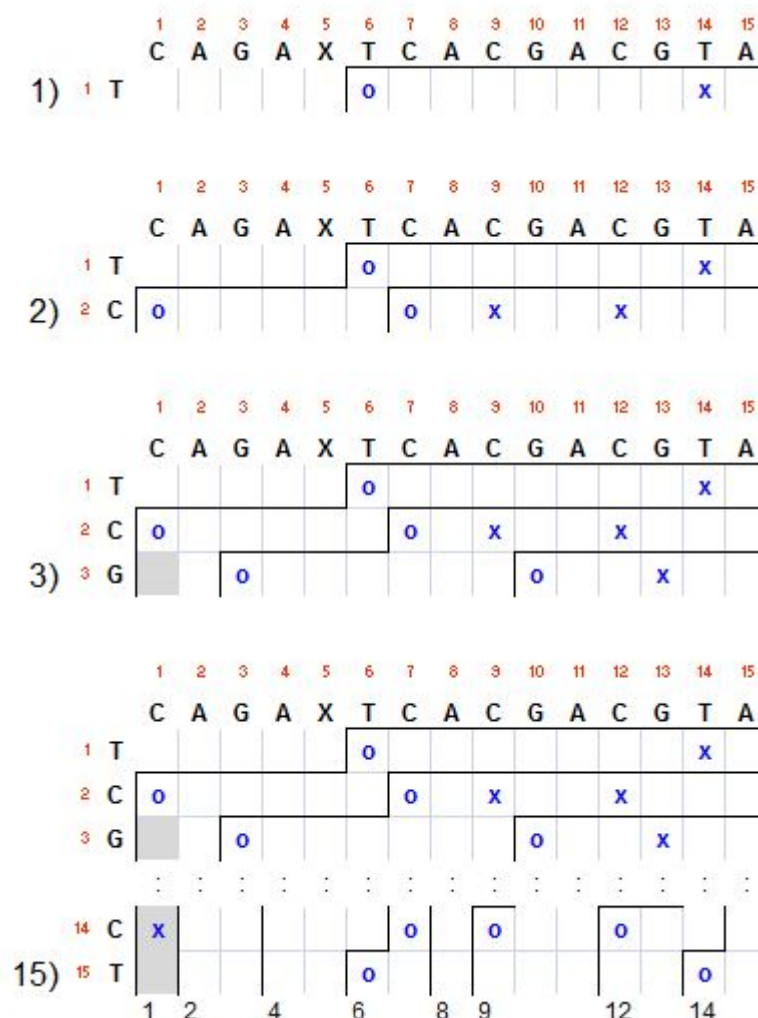


Figura 9 – Sequência de análise linha a linha do algoritmo última linha. O número de fronteiras é o comprimento da LCS.

Na Figura 9, mostramos passo a passo como definir as fronteiras presentes na Figura 8a. Na linha 1, o ponto (1,6) descarta o ponto (1,14) colocando a coluna 6 como melhor candidata a fronteira de valor 1. Ao analisar a linha 2, verificamos que o ponto (2,1) é melhor que (1,6) para a fronteira 1 e que (2,7) é melhor candidato à fronteira 2, descartando assim os pontos (2,9) e (2,12). Além disso, a fronteira do ponto (2,1) não pode mais ser melhorada, pois está no limite esquerdo da matriz, e por isso todas as linhas subsequentes podem desconsiderar esta fronteira. Na linha 3, a fronteira 2 foi melhorada para coluna 3 pelo ponto (3,3) e o ponto (3,10) é melhor candidato a fronteira 3. Ao final do algoritmo, teremos o conjunto {1,2,4,6,8,9,12,14} das colunas onde ocorrem os limites

das fronteiras. O número de elementos deste conjunto é o comprimento da LCS.

O [Código 4.4](#) apresenta a implementação da solução última linha. As listas encadeadas são codificadas com vetores lineares  $Qx$  e  $Qy$ , que representam coordenadas do conjunto  $R$  mencionado anteriormente. Usamos  $\text{SigmaX}$  e  $\text{SigmaY}$  para apontar a posição inicial de cada caractere do alfabeto nas strings de entrada. Estes vetores são criados e inicializados nas Linhas 15 a 31. Nas Linhas 33 e 34 criamos um vetor  $L$  de tamanho  $m$  que inicialmente assume que todas as possíveis colunas de fronteiras situam-se no infinito. A partir disso, para cada elemento de  $Y$  verificamos se ele ainda contém coincidências em  $X$ . Se não houver, marcamos qual a primeira fronteira que ainda podemos melhorar e descartamos este caractere de  $Qy$  (Linhas 42 e 43); verificamos então se a fronteira da ocorrência em  $X$  do caractere sendo analisado em  $Y$  melhorou ao máximo possível e marcamos esta posição para não mais ser atualizada (Linhas 45 a 50); atualizamos as posições das fronteiras já conhecidas (Linhas 51 a 58); verificamos se uma nova fronteira foi encontrada (Linhas 59 a 62); e finalmente retornamos  $K$  como o número de colunas válidas encontradas (Linha 64). O vetor  $L$  conterà as colunas da matriz onde passam as fronteiras divisórias do comprimento da LCS na última linha processada. A complexidade do pré-processamento para indexar as strings é  $O(m+n)$ . O processamento da LCS ocorre sobre o conjunto  $R$ , então a complexidade de tempo é  $O(R) + O(m+n)$ . A utilização de espaço é linear com comprimento das strings e portanto é  $O(N)$ . O tempo de execução foi de 2,4 segundos, comprovando a melhora de eficiência ao reduzir significativamente o conjunto de dados a serem processados.

```

1  int LCS(char *X, char *Y) {
2      int i, j, k, l, m, n, o, K, P, INF;
3
4      for(m=0; X[m]; m++);
5      for(n=0; Y[n]; n++);
6
7      if (m==0 || n==0) return 0;
8
9      if (m<n) {
10         char *T;
11         T=X; X=Y; Y=T;
12         k=m; m=n; n=k;
13     }
14
15     int SigmaX[256], Qx[m];
16     int SigmaY[256], Qy[n];
17
18     for (i=0; i<256; i++) {

```



```
19     SigmaX[i]=INF;
20     SigmaY[i]=INF;
21 }
22
23 for (i=m-1;i>=0;i--) {
24     Qx[i]=SigmaX[X[i]];
25     SigmaX[X[i]]=i;
26 }
27
28 for (i=n-1;i>=0;i--) {
29     Qy[i]=SigmaY[Y[i]];
30     SigmaY[Y[i]]=i;
31 }
32
33 int L[m];
34 for (i=0;i<m;i++) L[i]=MAXs;
35
36 P=INF;K=0;i=0;j=0;k=0;l=0;
37 int ini=0;
38 int mel=0;
39 for (i=0;i<n;i++)
40 {
41     k=ini;
42     while (SigmaY[X[mel]]==INF) mel++;
43     SigmaY[Y[i]]=Qy[SigmaY[Y[i]]];
44
45     P=SigmaX[Y[i]];
46     if(P==mel) {
47         SigmaX[Y[i]]=Qx[SigmaX[Y[i]]];
48         mel++;
49         ini++;
50     }
51     while (P<INF&&k<K) {
52         if (P<=L[k]) {
53             o=P;
54             while (P<=L[k]&&P<INF) P=Qx[P];
55             L[k]=o;
56         }
57         k++;
58     }
59     if(P<L[k]) {
60         L[k]=P;
```

```

61         K++;
62     }
63 }
64 return K;
65 }

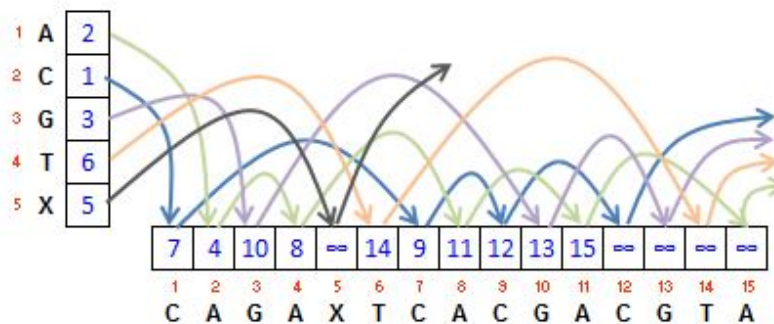
```

Código 4.4 – Solução última linha

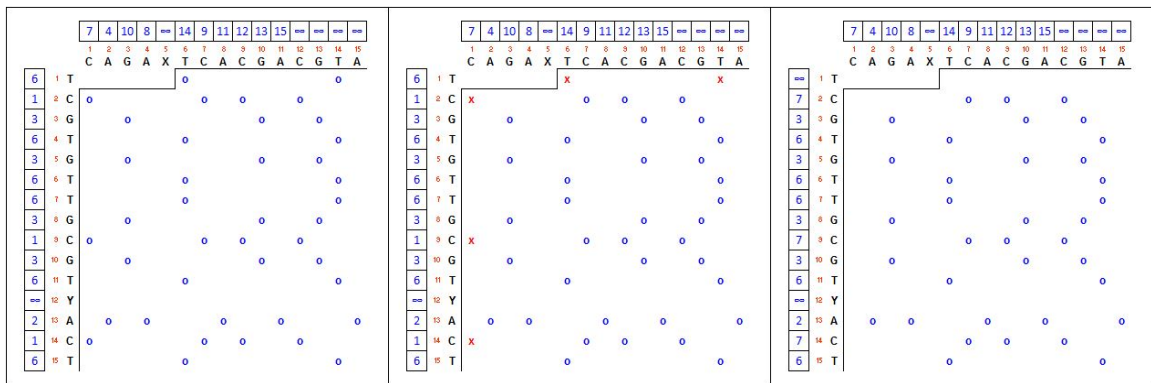
### 4.2.2 Solução limpeza de matriz

A solução limpeza de matriz é uma solução própria derivada da simplificação da solução última linha. O objetivo é simplesmente contar a quantidade de camadas que correspondem a cada fronteira, e assim obter o comprimento da LCS, sem o custo da manutenção de qualquer outra informação. Esta contagem é realizada eliminando-se de uma matriz de coincidências (*i.e. conjunto  $R$* ) todos os pontos pertencentes a uma determinada camada  $k$ , sequencialmente, até a matriz ficar vazia.

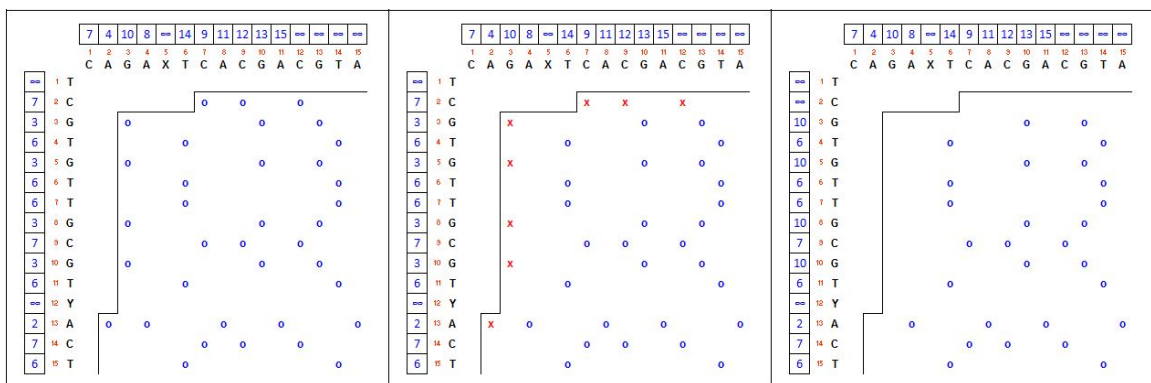
O algoritmo utiliza listas encadeadas para representar a matriz. A eliminação dos pontos de cada camada  $k$  é realizada recalculando os ponteiros da lista para que os pontos que estão sendo eliminados não sejam mais apontados. A Figura 10 ilustra o esquema da lista encadeada onde o espaço ocupado tem o mesmo comprimento da string representada. A Figura 11 ilustra os passos necessários para limpar totalmente a matriz. O número de passos é igual ao comprimento da LCS.

Figura 10 – Lista encadeada para a string  $X$ .

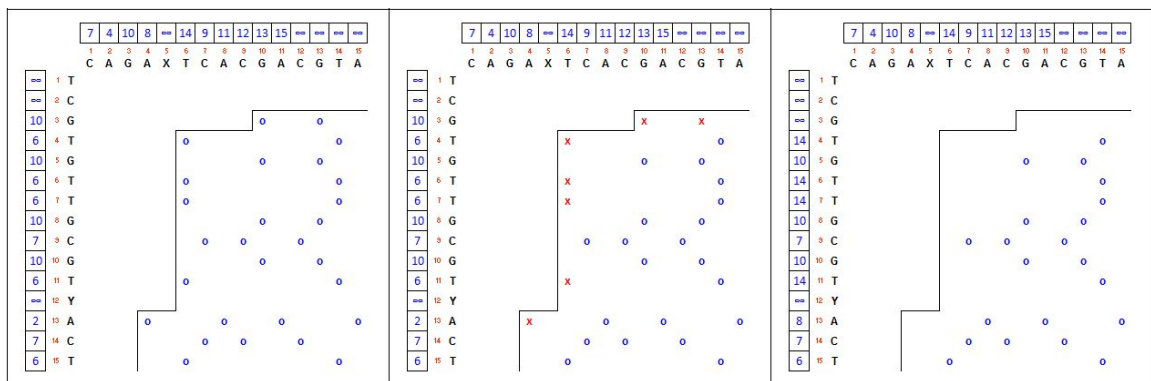
O código Código 4.5 apresenta a implementação da solução limpeza de matriz. O tempo de execução desta solução foi de 2,387 segundos, praticamente igual ao da solução última linha. A melhora obtida com a eliminação do custo de manutenção do vetor de posições das fronteiras foi totalmente perdida pela falta da indexação da string  $Y$ . A estrutura de representação do conjunto  $R$  por listas encadeadas somente para a string  $X$ , ou seja, somente nas linhas da matriz, não permite que sejam eliminados numa única operação toda uma coluna de pontos irrelevantes. Por conta disso, o ganho sobre a base  $R$  foi pequeno e a solução degenera para alguns casos como por exemplo o de duas strings



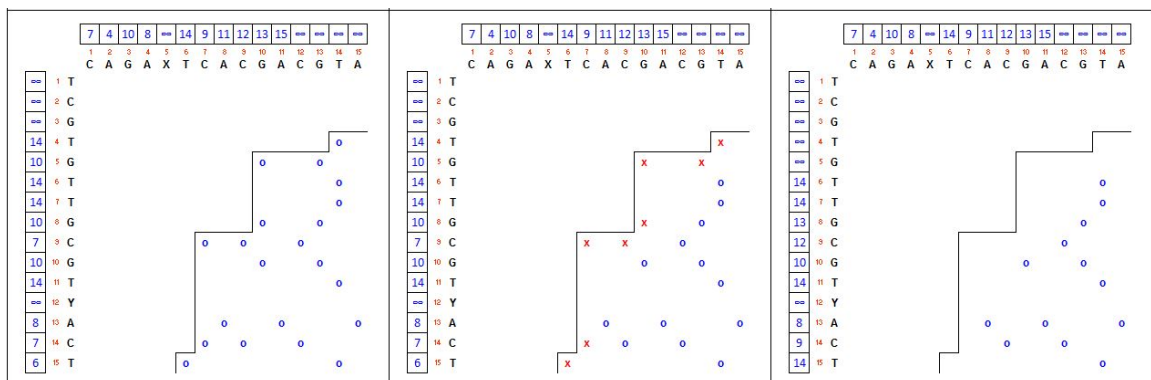
(a) Passo 1



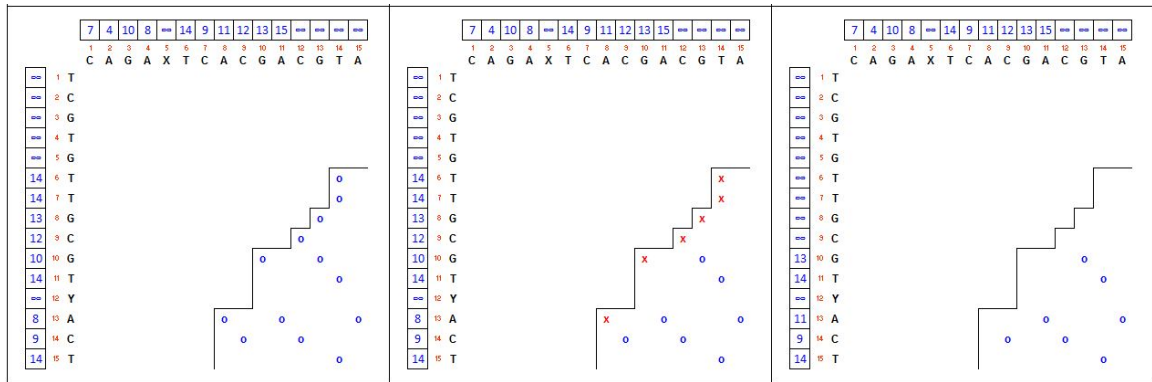
(b) Passo 2



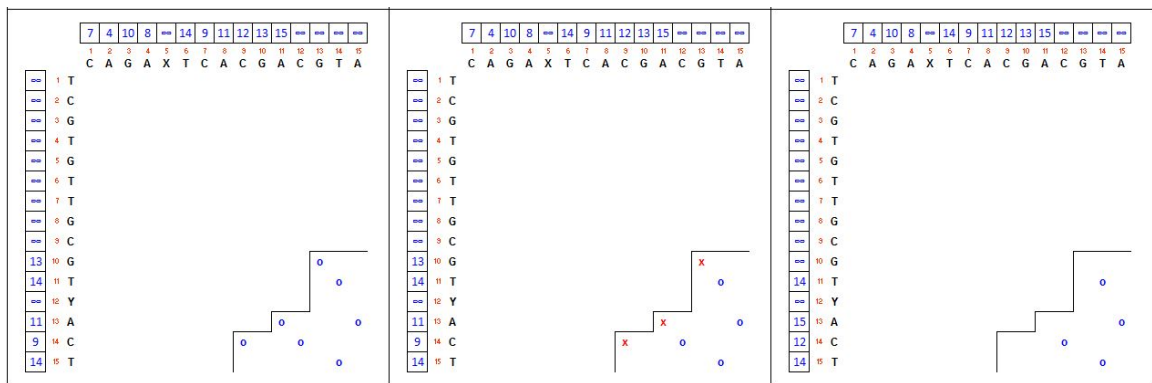
(c) Passo 3



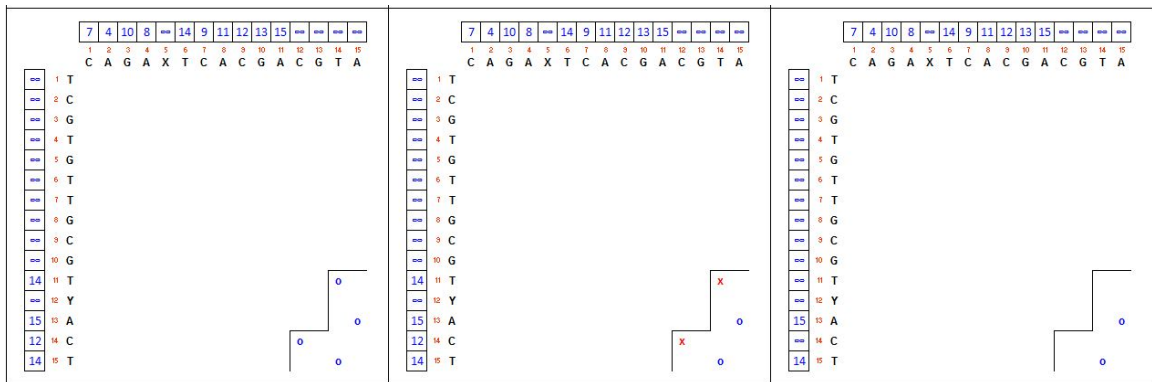
(d) Passo 4



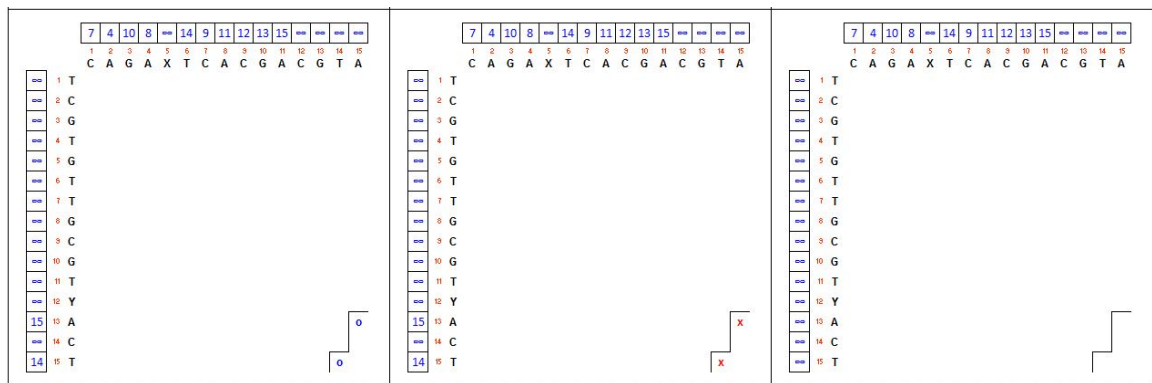
(e) Passo 5



(f) Passo 6



(g) Passo 7



(h) Passo 8

Figura 11 – Sequência de exemplo da solução limpeza de matriz.

idênticas com alfabeto de tamanho unitário. A complexidade de tempo no pior caso é  $O(N^2/2) + O(m + n)$ . E no caso médio é  $O(R) + O(m + n)$ .

```
1 int LCS(char *X, char *Y)
2 {
3     int i,j,k,l,m,n,o,K;
4
5     for(m=0;X[m];m++);
6     for(n=0;Y[n];n++);
7     if (m==0||n==0) return 0;
8
9     if (m<n){
10        char *T;
11        T=X;X=Y;Y=T;
12        k=m;m=n;n=k;
13    }
14    int Sigma[256], Q[m], Mx[n];
15
16    for (i=0;i<256;i++)
17        Sigma[i]=INF;
18    for (i=m-1;i>=0;i--) {
19        Q[i]=Sigma[X[i]];
20        Sigma[X[i]]=i;
21    }
22    for (i=0;i<n;i++)
23        Mx[i]=Sigma[Y[i]];
24
25    for (K=0,i=0;i<n;i++) {
26        if (Mx[i]<INF) {
27            j=Mx[i];
28            for (k=i+1;k<n;k++) {
29                if (Mx[k]<=j) {
30                    o=Mx[k];
31                    while (Mx[k]<=j&&Mx[k]<INF)
32                        Mx[k]=Q[Mx[k]];
33                    j=o;
34                }
35            }
36            K++;
37        }
38    }
```

```

39     return K;
40 }

```

Código 4.5 – Solução limpeza de matriz

### 4.2.3 Solução diagonal

A Solução diagonal é uma solução própria. Durante a fase de pesquisa, não foi encontrado nenhum trabalho utilizando esta abordagem, donde podemos supor ser esta solução uma contribuição inédita ou rara. Foi desenvolvida a partir da solução limpeza de matriz para explorar melhor o uso das listas encadeadas. As listas apresentam a vantagem de poderem ser abandonadas ao atingir-se um elemento limite a partir do qual todos os demais são irrelevantes, porém, na solução limpeza de matriz, esta vantagem somente é aproveitada na porção superior da diagonal principal e se mostra pouco efetiva na porção inferior. A ideia da solução diagonal é dividir a matriz em duas metades através da diagonal principal e representar a metade superior em listas baseada em linhas e a metade inferior em listas baseada em colunas. Como pode ser visto na [Figura 12](#), as linhas de fronteira mudam de direção próximo de cruzarem a diagonal da matriz sugerindo que a porção superior é melhor representada por listas ao longo da string *X* e a porção inferior por listas ao longo da string *Y*.

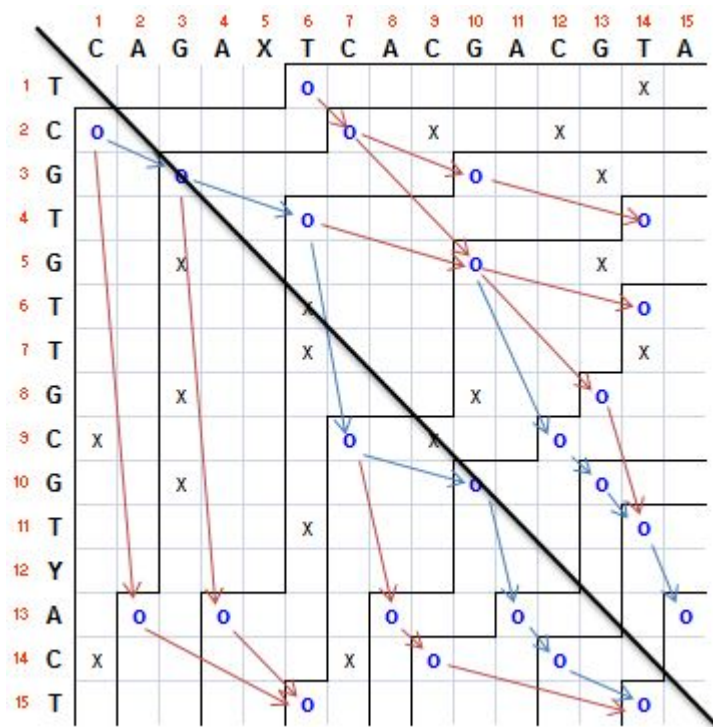
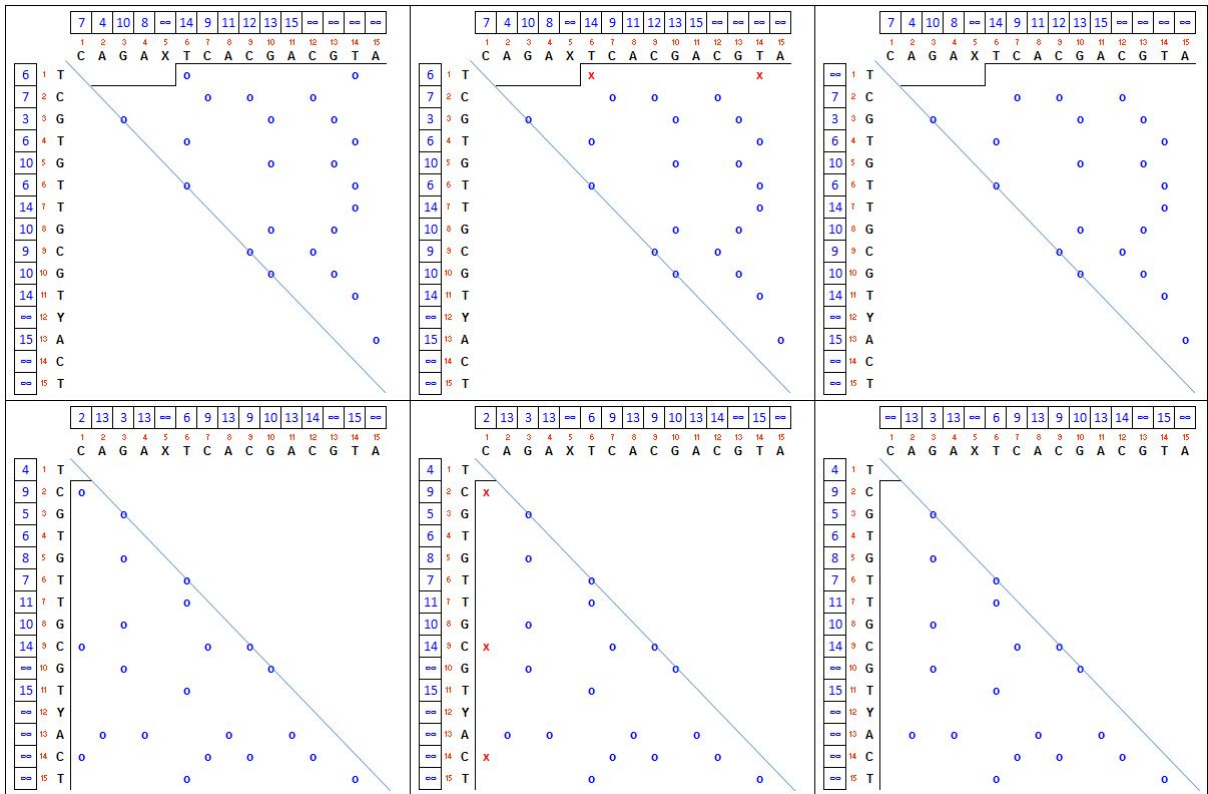
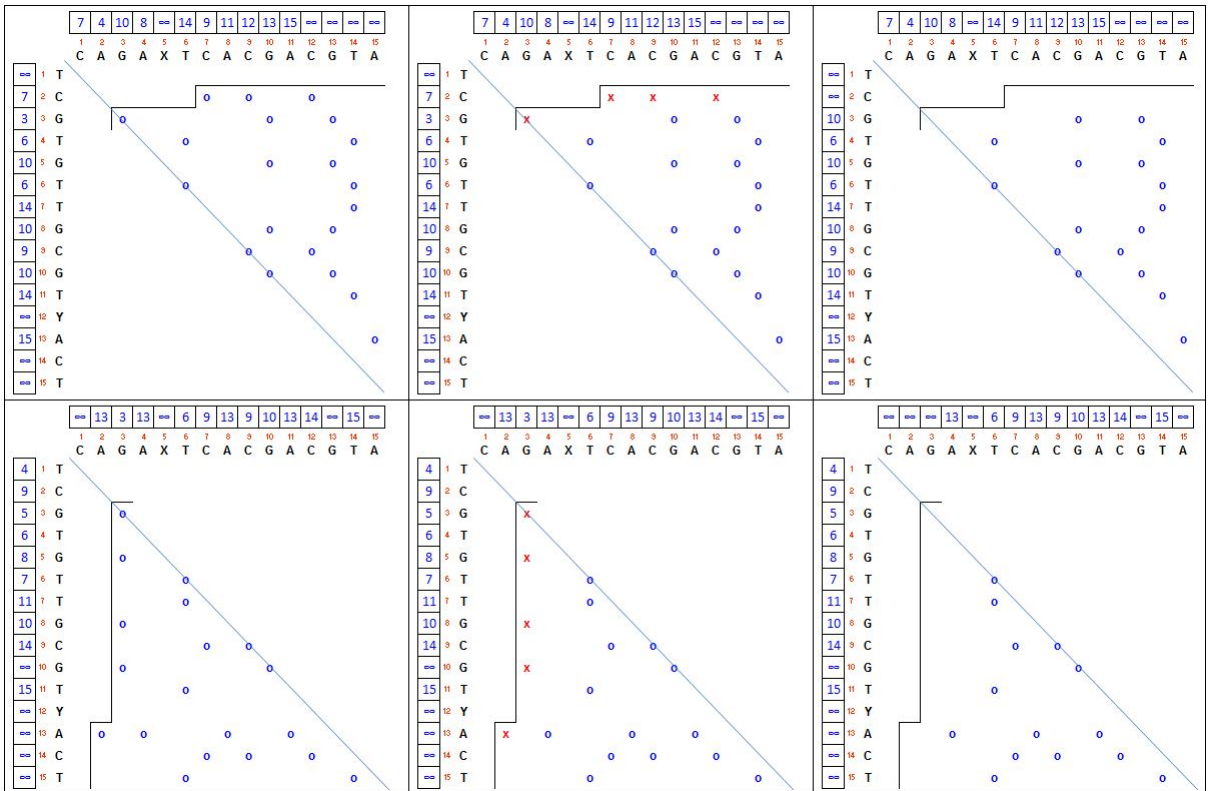


Figura 12 – A diagonal caracteriza um eixo de semelhança na matriz LCS.

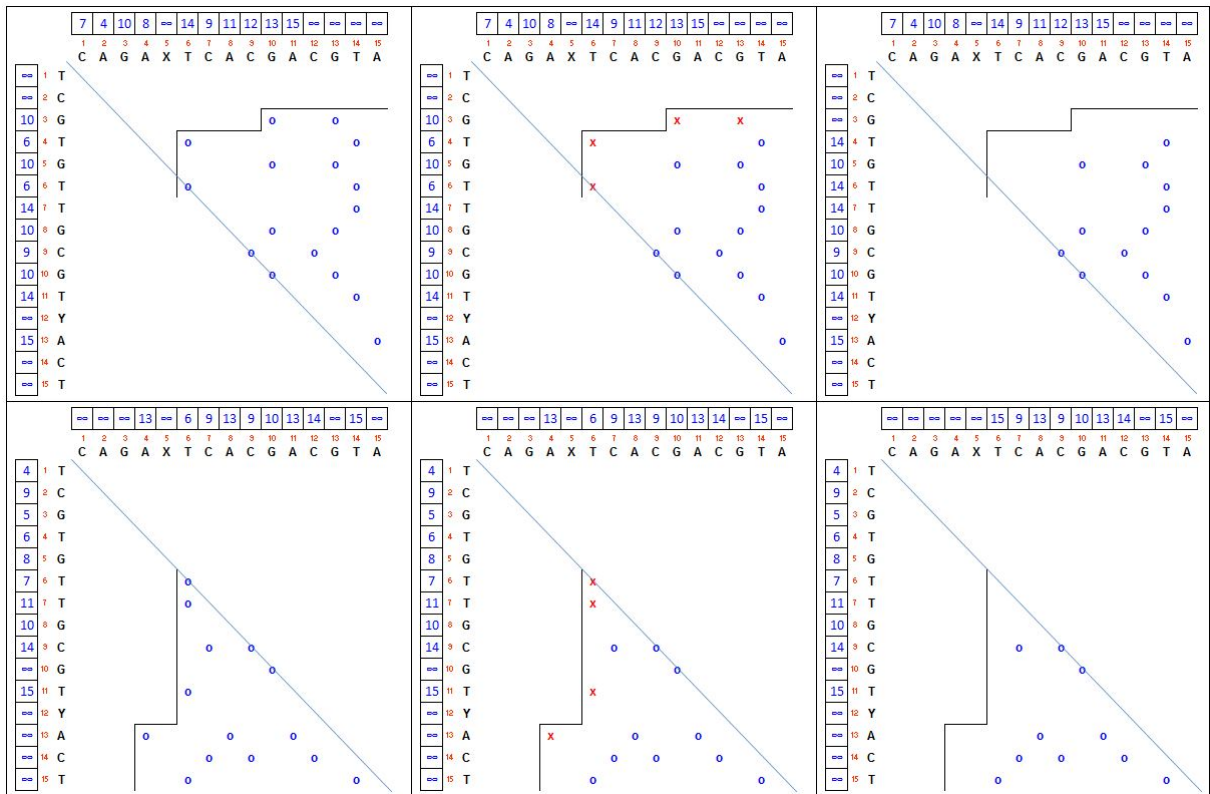
A [Figura 13](#) ilustra os passos necessários para limpar totalmente a matriz. O número de passos é igual ao comprimento da LCS.



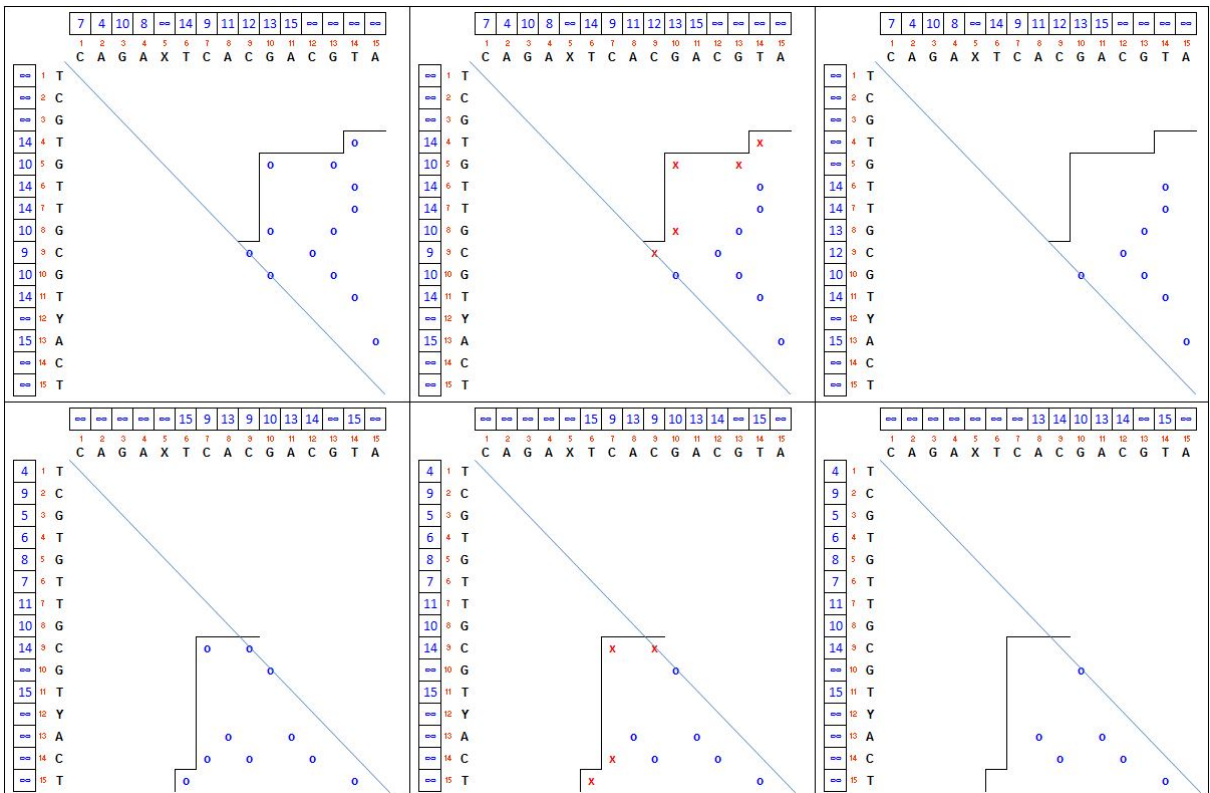
(a) Passo 1



(b) Passo 2

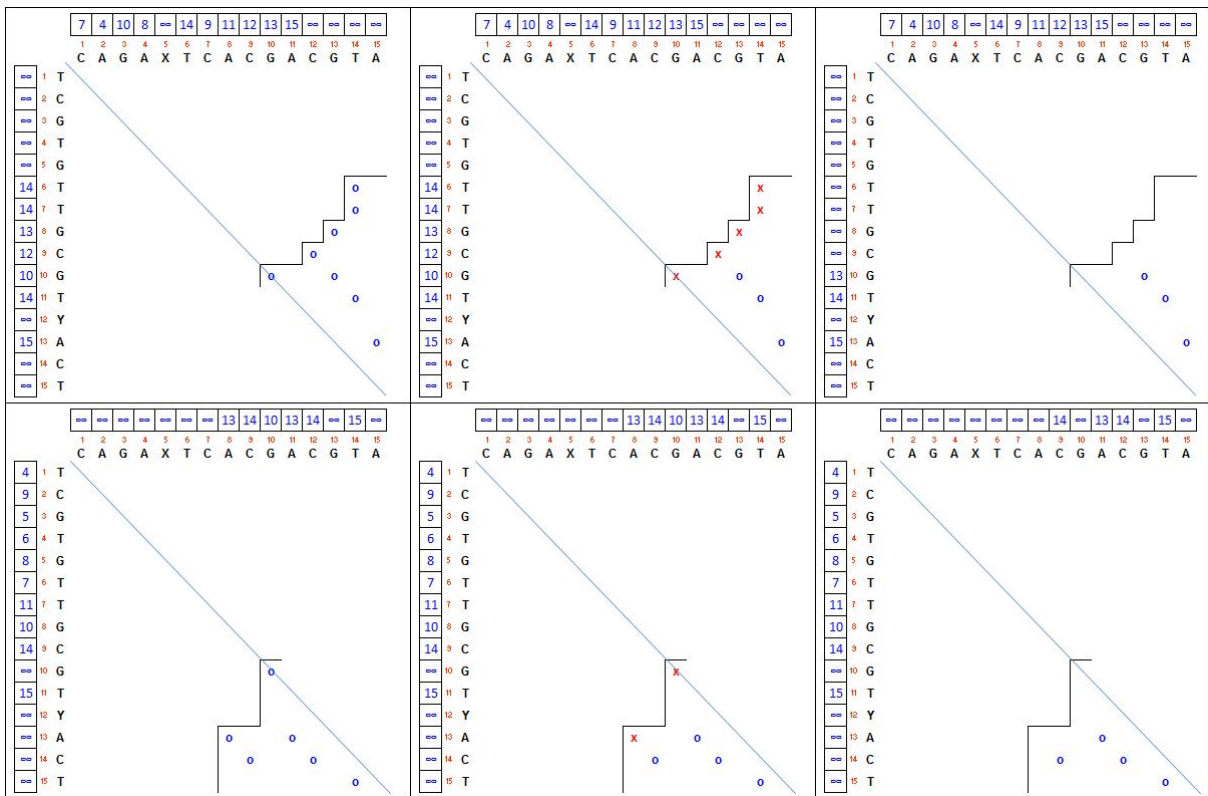


(c) Passo 3

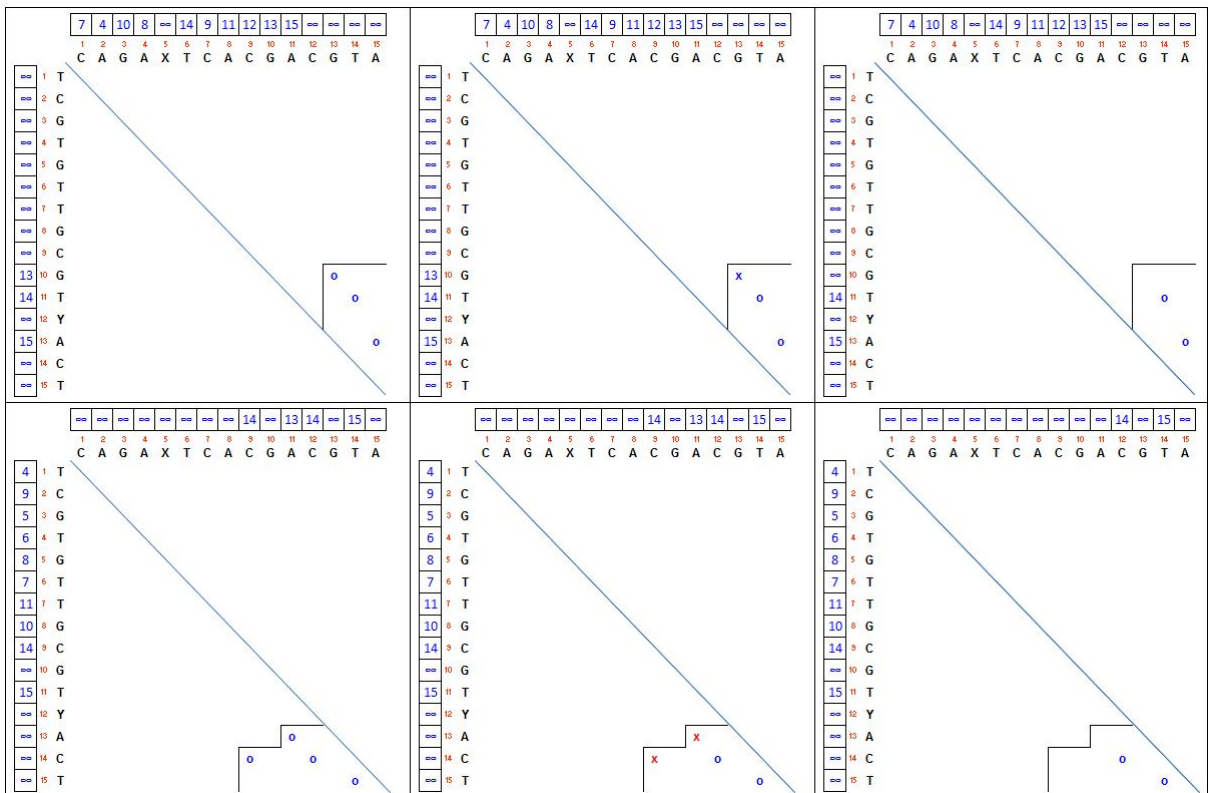


(d) Passo 4

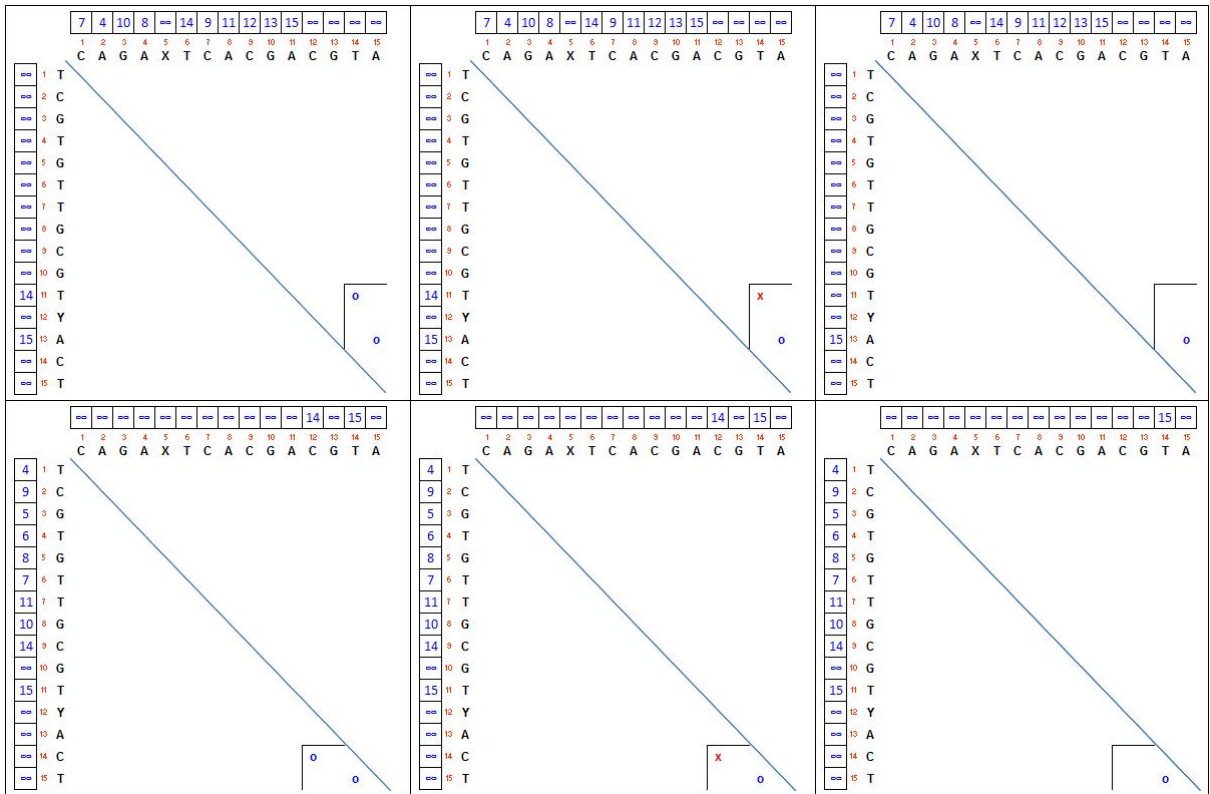




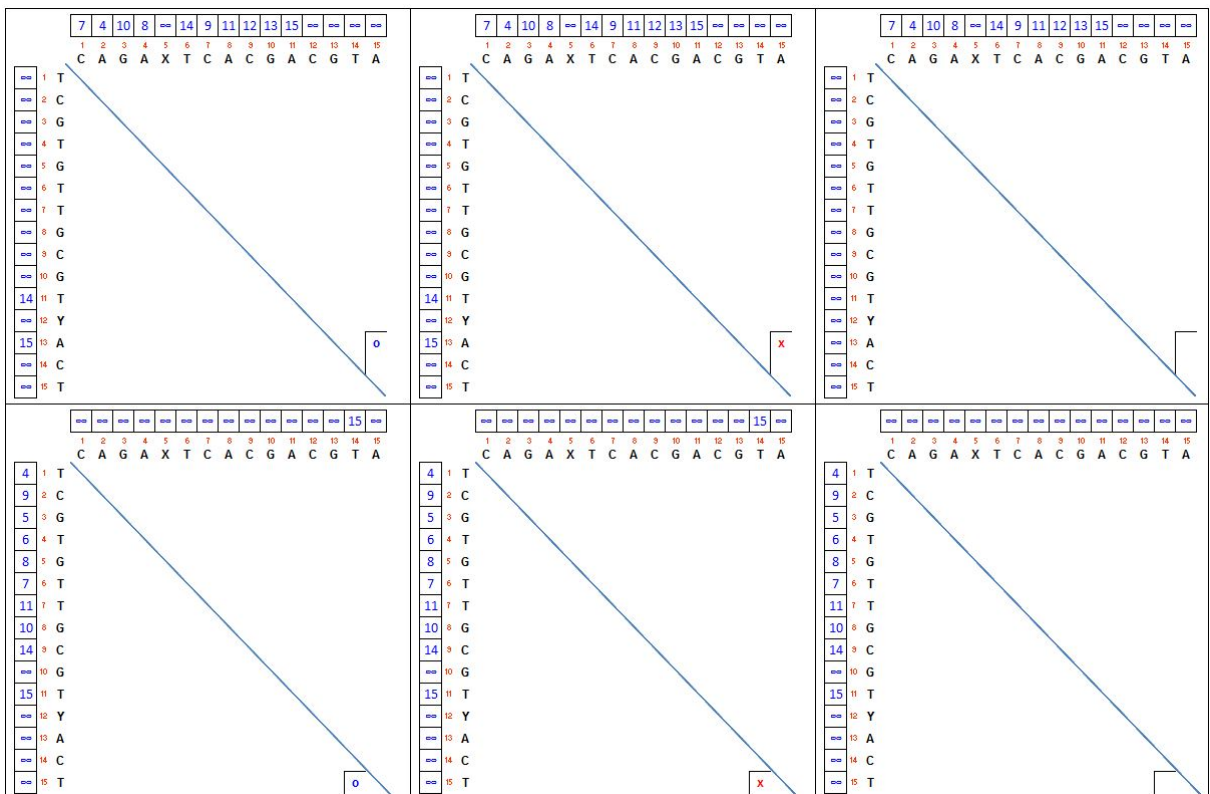
(e) Passo 5



(f) Passo 6



(g) Passo 7



(h) Passo 8

Figura 13 – Sequência de exemplo da solução diagonal.

O processamento é realizado de forma síncrona para linhas e colunas, e consiste em eliminar sequencialmente os grupos de pontos pertencentes a cada fronteira, até que as listas fiquem vazias. Num primeiro passo, selecionamos as linha e coluna mais externas de cada fronteira  $k$ , detectamos o primeiro ponto dominante de cada uma através de sua lista e descartamos o restante, que é constituído totalmente por pontos irrelevantes. Em seguida verificamos sequencialmente se este ponto pode ser melhorado delineando a continuidade da linha de fronteira até tocar a diagonal. Como a LCS desenvolve-se alinhada próxima da diagonal, este segundo processamento é interrompido muito cedo, contribuindo para a eficiência do código. O [Código 4.6](#) apresenta a implementação da solução diagonal.

Neste código criamos as listas encadeadas exatamente como na solução última linha (Linhas 15 a 31). Em seguida, atualizamos dois novos vetores  $MX$  e  $MY$  para dividir a matriz em duas através da sua diagonal. Estes vetores apontam para a primeira ocorrência do primeiro ponto das listas a partir da diagonal (Linhas 33 a 43). Por fim, as camadas são contadas e eliminadas das duas metades sincronizadamente (Linhas 45 a 67), onde, nas Linhas 46 a 49, são eliminados os pontos mais externos. E nas linhas 50 a 64 são eliminados os pontos mais internos da fronteira  $k$ , em cada metade, até tocarem a diagonal.

O tempo de execução foi de 1,326 segundos. O processamento ocorre parcialmente sobre o conjunto  $R$  e a complexidade de tempo é  $O(R) + O(m + n)$ .

```
1 int LCS(char *X, char *Y)
2 {
3     register int i, j, k, l, m, n, o, K;
4
5     for(m=0; X[m]; m++);
6     for(n=0; Y[n]; n++);
7     if (m==0 || n==0) return 0;
8
9     if (m<n) {
10         char *T;
11         T=X; X=Y; Y=T;
12         k=m; m=n; n=k;
13     }
14
15     int SigmaX[256], SigmaY[256];
16     int QX[m], QY[n], MX[m], MY[n];
17
18     for (i=0; i<256; i++) {
19         SigmaX[i]=INF;
20         SigmaY[i]=INF;
21     }
```

```
22
23     for (i=m-1;i>=0;i--) {
24         QX[i]=SigmaX[X[i]];
25         SigmaX[X[i]]=i;
26     }
27
28     for (i=n-1;i>=0;i--) {
29         QY[i]=SigmaY[Y[i]];
30         SigmaY[Y[i]]=i;
31     }
32
33     for (i=0;i<m;i++) {
34         while (SigmaY[X[i]]<i)
35             SigmaY[X[i]]=QY[SigmaY[X[i]]];
36         MX[i]=SigmaY[X[i]];
37     }
38
39     for (i=0;i<n;i++) {
40         while (SigmaX[Y[i]]<i)
41             SigmaX[Y[i]]=QX[SigmaX[Y[i]]];
42         MY[i]=SigmaX[Y[i]];
43     }
44
45     for (K=0,k=0;k<n;k++) {
46         if (MY[k]<INF||MX[k]<INF) {
47             l=k;
48             i=MY[l]; MY[l]=INF;
49             j=MX[l]; MX[l]=INF;
50             while (l<i&&l<j) {
51                 l++;
52                 if(MY[l]<=i) {
53                     o=MY[l];
54                     while (MY[l]<=i&&MY[l]<INF)
55                         MY[l]=QX[MY[l]];
56                     i=o;
57                 }
58                 if(MX[l]<=j) {
59                     o=MX[l];
60                     while (MX[l]<=j&&MX[l]<INF)
61                         MX[l]=QY[MX[l]];
62                     j=o;
63             }
```

```

64         }
65         K++;
66     }
67 }
68 return K;
69 }

```

Código 4.6 – Solução diagonal

#### 4.2.4 Solução bit-paralelo

Sabendo que um ponto da matriz LCS precisa de um único bit para representar a ocorrência de um casamento de caractere (Figura 14), pode-se obter ganho de velocidade de processamento ao aproveitar-se da arquitetura da máquina na construção da solução. De fato, a solução tradicional  $O(mn)$  pode ser reduzida por um fator  $w$  correspondente ao comprimento em bits da palavra suportada pelo computador, uma vez que operações lógicas afetarão  $w$  bits paralelamente. Nota-se, entretanto, que a solução passa a ser dependente da máquina e, idealmente, o Sistema Operacional e o compilador deveriam utilizar plenamente o potencial do processador igualando o comprimento da sua palavra binária ao comprimento da palavra binária do processador.

O algoritmo é similar à solução última linha apresentada na subseção 4.2.1, mantendo-se a ideia de processar e identificar o início das sucessivas fronteiras da matriz LCS. Mas agora no domínio binário, em vez de listas encadeadas no domínio dos inteiros Figura 14, os bits marcados em azul são os casamentos de caracteres relevantes e marcam o limite de uma fronteira mais a esquerda na sua linha.

O princípio utilizado na movimentação do bit marcador mais para a esquerda é fundamentado na operação de decrementar um número binário. Esta operação marca todos os bits 0 menos significativos com valor 1 até encontrar o primeiro bit 1 menos significativo, que é marcado com 0, deixando demais bits mais significativos inalterados. Este bit pode ser isolado com as operações abaixo:

$$\begin{array}{r|l}
 x & 1000101110010100 \\
 -1 & -0000000000000001 \\
 \hline
 (x-1) & 1000101110010011 \\
 \neg(x-1) & 0111010001101100 \\
 \hline
 r = x \& \neg(x-1) & 0000000000000100
 \end{array}$$

Se entendermos que o 1 a ser subtraído é uma referência para encontrar o bit

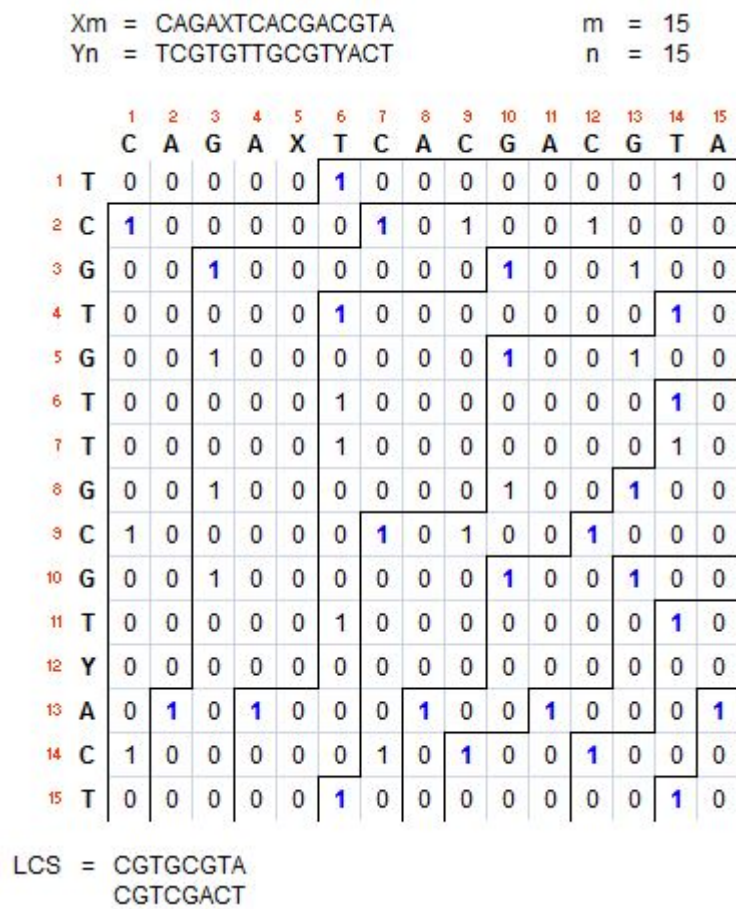


Figura 14 – Representação binária da matriz de casamentos de caracteres. Bits 1 marcam os casamentos dos caracteres, sendo que os bits em azul são relevantes e delimitam as fronteiras.

menos significativo mais próximo desta referência, podemos aplicar o mesmo princípio, em paralelo, conforme o exemplo abaixo:

$$\begin{array}{r|l}
 x & 1000101110010100 \\
 d & -0000101000100001 \\
 \hline
 (x-d) & 1000000101110011 \\
 \neg(x-d) & 0111111010001100 \\
 \hline
 r = x \& \neg(x-d) & 0000101010000100
 \end{array}$$

É como se estivéssemos decrementando diversos números binários simultaneamente em paralelo e houvesse um espaço fictício entre eles. O exemplo redesenhado a seguir demonstra 4 operações simultâneas realizadas em um registrador fictício de 16 bits.

$$\begin{array}{r|l}
 x & 10001\ 01\ 1100\ 10100 \\
 d & -00001\ 01\ 0001\ 00001 \\
 \hline
 (x-d) & 10000\ 00\ 1011\ 10011 \\
 \neg(x-d) & 01111\ 11\ 0100\ 01100 \\
 \hline
 r = x \& \neg(x-d) & 00001\ 01\ 0100\ 00100
 \end{array}$$

Para aplicar este princípio na resolução do problema LCS devemos fazer os seguintes ajustes:

- Organizar a string  $X$  em ordem reversa para que a marcação dos casamentos de caracteres estejam na mesma direção da movimentação dos bits da operação binária, que é no sentido do bit mais significativo para o menos significativo, conforme Figura 15.
- Pré-processar o alfabeto da string para criar uma representação binária dos casamentos deste caractere sobre a string  $X$ .
- Cuidar para que *carries* e *borrows* sejam propagados além do limite do comprimento da palavra de bits do computador utilizado.
- Para que o bit já detectado como elemento pertencente a LCS não seja considerado em duplicidade, a descoberta do próximo bit representando o próximo elemento da LCS se faz através da rotação para a direita de 1 bit do resultado atual corrente.

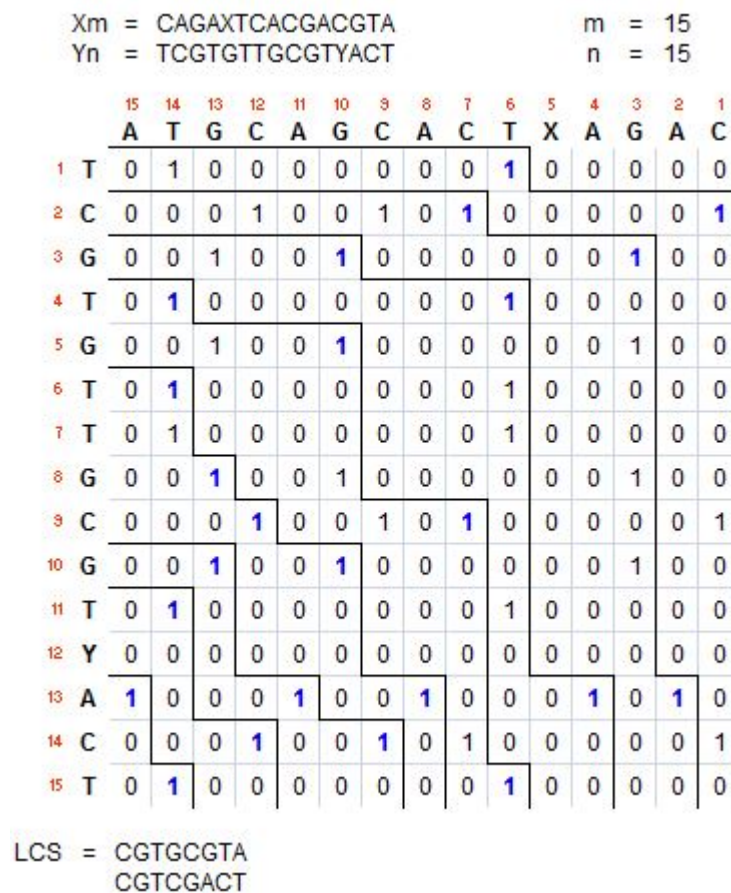


Figura 15 – Representação da matriz de casamentos de caracteres em reverso para uso da técnica de bit-paralelo.

Para as formulações a seguir, considere que  $L_i$  é a marcação das fronteiras na  $i$ -ésima linha da matriz e  $L_{i-1}$  é a mesma marcação para a linha anterior,  $U$  e  $V$  são variáveis auxiliares, e que  $T_i$  é a string binária dos casamentos do caractere  $Y_i$  sobre a string  $X$ .

A solução bit-paralelo levou à experimentações visando reduzir o número de operações de máquina e assim obter maior eficiência. Allison e Dix (1986) são os precursores desta metodologia, e em seu trabalho apresentaram as Equações 4.4 e 4.5:

$$U = L_{i-1} \vee T_i \tag{4.4}$$

$$L_i = U \wedge \neg(U - ((L_{i-1} \ll 1) \vee 1)) \tag{4.5}$$

retornando a quantidade de bits 1 em  $L_n$  como tamanho da LCS.

Crochemore et al. (2000) apresentaram a versão complementar, como mostrado na Equação 4.6:

$$\overline{L}_i = (\overline{L_{i-1}} + (\overline{L_{i-1}} \wedge T_i)) \vee (\overline{L_{i-1}} \wedge (\neg(T_i))) \tag{4.6}$$

retornando o número de bits 0 em  $L_n$  como tamanho da LCS.

Hyyrö (2004) alega que sua formulação apresenta melhor eficiência que as anteriores. As Equações 4.7 e 4.8 apresentam a mesma.

$$U = (\overline{L_{i-1}} \wedge T_i) \tag{4.7}$$

$$\overline{L}_i = (\overline{L_{i-1}} + U) \vee (\overline{L_{i-1}} - U) \tag{4.8}$$

retornando o número de bits 0 em  $L_n$  como tamanho da LCS.

Para exemplificar, segue o cálculo para  $L_{13}$  utilizando Allison e Dix (1986):

$L_{12}$	011001001100101
$T_{13}$	100010010001010
$U = L_{12} \vee T_{13}$	111011011101111
$(L_{12} \ll 1) \vee 1$	110010011001011
$V = U - (L_{12} \ll 1) \vee 1$	001001000100100
$\neg V$	110110111011011
$L_{13} = U \wedge \neg V$	110010011001011

A implementação desta solução bit-paralelo é apresentada no código Código 4.7.



```

1 int LCS(char *X, char *Y) {
2     register int i,j,k,m,n;
3     for(m=0;X[m];m++);
4     for(n=0;Y[n];n++);
5     if (m==0||n==0) return 0;
6
7     if (m<n) {
8         char *T;
9         T=X;X=Y;Y=T;
10        k=m;m=n;n=k;
11    }
12
13    int w=(m+31)>>5; // ceil[m/32]
14
15    uint S[256][w]; // Sigma {all char}
16    for (i=0;i<256;i++)
17        for (j=0;j<w;j++)
18            S[i][j]=0;
19
20    uint set=1;
21    for (i=0,j=0;i<m;i++) {
22        S[X[i]][j]=set;
23        set<<=1;
24        if (!set){set++;j++;}
25    }
26
27    uint L[w]; // Vetor L(i)
28    uint b1,b2,c; // borrows and carries
29
30    for (i=0;i<w;i++) L[i]=0;
31
32    register uint U,V;
33    for (i=0;i<n;i++) {
34        b1=1;b2=0;
35        for (j=0;j<w;j++) {
36            U = L[j] | S[Y[i]][j];
37            c = L[j]>>31;
38            V = U-(L[j]<<1|b1+b2);
39            b1=c; b2=(V>U);
40            L[j] = U & (~V);
41        }
42    }

```

```

43     for (k=0, i=0; i<w; i++)
44         for (; L[i]; k++) L[i]&=L[i]-1;
45     return k;
46 }

```

Código 4.7 – Solução bit-paralelo.

Esta solução apresenta variáveis contadoras operando no registrador para ganhar velocidade (Linha 2); detecção do comprimento das strings (Linhas 3 e 4); escolha da maior string para representação binária (Linhas 7 a 11); cálculo do número inteiro de palavras de 32 bits necessárias para representar o mapa de casamentos de caracteres da string  $Y$  sobre a string  $X$  (Linha 13); inicialização das strings binárias de ocorrências para cada caractere do alfabeto utilizado (Linhas 15 a 25); inicialização do vetor linha  $L$  da matriz LCS com bits 0 em toda sua extensão (Linhas 27 a 30); implementação do código segundo o algoritmo de [Allison e Dix \(1986\)](#) (Linhas 32 a 42); e finalmente a contagem dos bits 1 do vetor  $L[w]$  usando o método de [Kernighan e Ritchie \(1988\)](#), que representa o comprimento da LCS (Linhas 43 a 46). A complexidade de tempo é naturalmente  $O(mn/w)$ , e o algoritmo apresentado acima executou em 453 ms para os casos de teste considerados.

[Hyyrö \(2004\)](#) apresenta referências para os algoritmos de [Allison e Dix \(1986\)](#) e [Crochemore et al. \(2000\)](#), sustentando que seu algoritmo é o mais veloz entre as soluções de bit-paralelo por usar menos operações de máquina no núcleo do algoritmo. Entretanto, as nossas implementações destas 3 versões, mantendo uma equivalência nos recursos e técnicas usadas, não confirmaram as afirmações de [Hyyrö \(2004\)](#).

Como mostrado na [Tabela 3](#), a implementação de [Allison e Dix \(1986\)](#) ainda é a mais veloz. Embora as soluções de [Allison e Dix \(1986\)](#) e [Crochemore et al. \(2000\)](#) usem 5 operações básicas no núcleo do método, contra 4 da solução de [Hyyrö \(2004\)](#), a necessidade de propagação de *carries* adiciona operações binárias que deixam todas as soluções marginalmente idênticas. Concluimos que a solução de [Allison e Dix \(1986\)](#) foi mais eficiente por usar um número maior de operações computacionais unárias, e que a solução de [Hyyrö \(2004\)](#) vence marginalmente a de [Crochemore et al. \(2000\)](#) por eliminar um acesso a variável indexada.

Solução	Tempo (ms)	Op Método	OP carry	Total	Unary
Allison	453	5	4	9	3
Crochemore	530	5	4	9	1
Hyyro	514	4	6	10	0

Tabela 3 – Tabela comparativa entre implementações [Allison e Dix \(1986\)](#), [Crochemore et al. \(2000\)](#) e [Hyyrö \(2004\)](#).

Os fragmentos de código a seguir, se substituídos nas linhas 32 a 45 do Código 4.7,

alteram o mesmo para se obter as soluções [Crochemore et al. \(2000\)](#) e [Hyyrö \(2004\)](#). Vale lembrar que o comprimento da LCS nestas soluções é obtido pela contagem de bits 0. Por razões práticas é melhor apenas acrescentar uma linha na contagem de bits para negar  $L_m$  e contar os bits 1 usando o algoritmo de [Kernighan e Ritchie \(1988\)](#), que conta bits em  $O(k)$ , onde  $k$  é o número de bits 1 da palavra binária.

```

1   register uint U,V;
2   for (i=0;i<n;i++) {
3       c=0;
4       for (j=0;j<w;j++) {
5           U = L[j]+(L[j]&S[Y[i]][j])+c;
6           c = U<c|U<L[j];
7           L[j]=U|L[j]&(~S[Y[i]][j]);
8       }
9   }
10  for (k=0,i=0;i<w;i++) {
11      L[i]=~L[i];
12      for (;L[i];k++) L[i]&=L[i]-1;
13  }
14  return k;

```

Código 4.8 – Fragmento para obter a solução [Crochemore et al. \(2000\)](#) para bit-paralelo.

```

1   register uint U,V;
2   for (i=0;i<n;i++) {
3       c=0;b=0;
4       for (j=0;j<w;j++) {
5           V = L[j]&S[Y[i]][j];
6           U = L[j]+V+c;
7           c = U<c|(U<L[j]);
8           V = L[j]-(V+b);
9           b = (V>L[j]);
10          L[j] = U|V;
11      }
12  }
13  for (k=0;i=0;i<w;i++) {
14      L[i]=~L[i];
15      for (;L[i];k++) L[i]&=L[i]-1;
16  }
17  return k;

```

Código 4.9 – Fragmento para obter a solução de [Hyyrö \(2004\)](#) para bit-paralelo.

### 4.2.5 Solução bit-paralelo com limites

As soluções que consideram a matriz de casamentos de caracteres entre as strings exploram os limites das fronteiras de transição da LCS. Estas fronteiras apresentam a característica fundamental de que, a cada linha processada, zero ou uma fronteira é acrescentada, representando o crescimento parcial da LCS em 0 ou 1 caractere no seu comprimento. A solução bit-paralelo atua, a cada linha processada, movendo os limites das fronteiras já descobertas da direita para a esquerda, e acrescentando zero ou uma nova fronteira na posição mais a esquerda possível após todas as fronteiras já descobertas.

A Figura 16 mostra um exemplo do desenvolvimento do vetor  $L_i$  para as strings de exemplo utilizando o algoritmo bit-paralelo. Simulamos nesta figura um hipotético processador cuja palavra tem 4 bits de comprimento. Para efeito didático, a string X foi alongada para ocupar 10 palavras de processador de 4 bits.

		w10	w9	w8	w7	w6	w5	w4	w3	w2	w1
		TACG	GGTT	GGAC	AAAC	GTCC	AGTC	XATG	CAGE	ACTX	AGAC
		0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
1	T	0000	0000	0000	0000	0000	0000	0000	0000	0010	0000
2	C	0000	0000	0000	0000	0000	0000	0000	0000	0100	0001
3	G	0000	0000	0000	0000	0000	0000	0000	0010	0000	0101
4	T	0000	0000	0000	0000	0000	0000	0010	0000	0010	0101
5	G	0000	0000	0000	0000	0000	0100	0000	0010	0010	0101
6	T	0000	0000	0000	0000	0100	0000	0010	0010	0010	0101
7	T	0000	0000	0000	0000	0000	0010	0010	0010	0010	0101
8	G	0000	0000	0000	0000	0000	0110	0001	0010	0010	0101
9	C	0000	0000	0000	0000	0001	0101	0000	1000	0110	0101
10	G	0000	0000	0000	0000	1001	0100	0001	0010	0110	0101
11	T	0000	0000	0000	0000	0101	0000	0011	0010	0110	0101
12	Y	0000	0000	0000	0000	0101	0000	0011	0010	0110	0101
13	A	0000	0000	0000	0010	0100	0000	0110	0100	1100	1011
14	C	0000	0000	0001	0001	0000	0001	0100	1001	1100	1011
15	T	0000	0000	0001	0000	0000	0011	0010	1001	1010	1011
16	:	:	:	:	:	:	:	:	:	:	:

w=4 bits

Figura 16 – Representação da evolução do vetor  $L_i$  através do algoritmo bit-paralelo.

Para observar a Figura 16 definimos que o início do vetor está mais a esquerda e o bit menos significativo representa o primeiro caractere da string X. O fim do vetor se estende até a última palavra binária do lado direito. Duas importantes observações podem ser feitas nesta figura: a primeira é que a solução de bit-paralelo processa toda a linha até o seu final para buscar o bit 1 mais relevante, mas o bit 1 mais relevante é encontrado muito antes do processamento atingir o final do vetor de bits. Logo, o restante do processamento resulta em nenhuma alteração no vetor  $L_i$ , pois o objetivo do algoritmo é justamente detectar apenas o bit 1 mais relevante e mais a esquerda, desprezando todos os demais até o fim do vetor, que serão mantidos com o valor 0. A segunda observação é que os bits relevantes que já tenham sido localizados e que não podem ser mais movidos

são recorrentemente repetidos no Vetor  $L_i$ , linha após linha, e como são movidos para a direita, estes bits 1 vão se acumulando do lado direito do vetor. Assim, não tendo mais espaço para mover estes bits, as palavras binárias que os representam não se modificam mais, embora a solução bit-paralelo reprocessem-nas inutilmente a cada linha.

Estas duas observações sugerem que o processamento binário pode ser feito dentro de uma janela limitada. Porém os limites desta janela são variáveis e ainda não desenvolvemos um algoritmo que permita estabelecer precisamente este contorno. O que fazemos é uma estimativa do comprimento da LCS em função do conteúdo das strings e do tamanho do alfabeto utilizado.

Se definirmos que um par de strings, cada uma com comprimento  $N$  tenha uma LCS de tamanho  $\lambda$ , é fácil perceber que geometricamente tal LCS não pode estar situada, nem em parte, dentro dos dois triângulos retos opostos à diagonal principal de hipotenusa de comprimento  $\lambda$ . Vamos chamar a hipotenusa deste triângulo de diagonal de corte e o triângulo de triângulo de corte, ambos ilustrados na Figura 17.

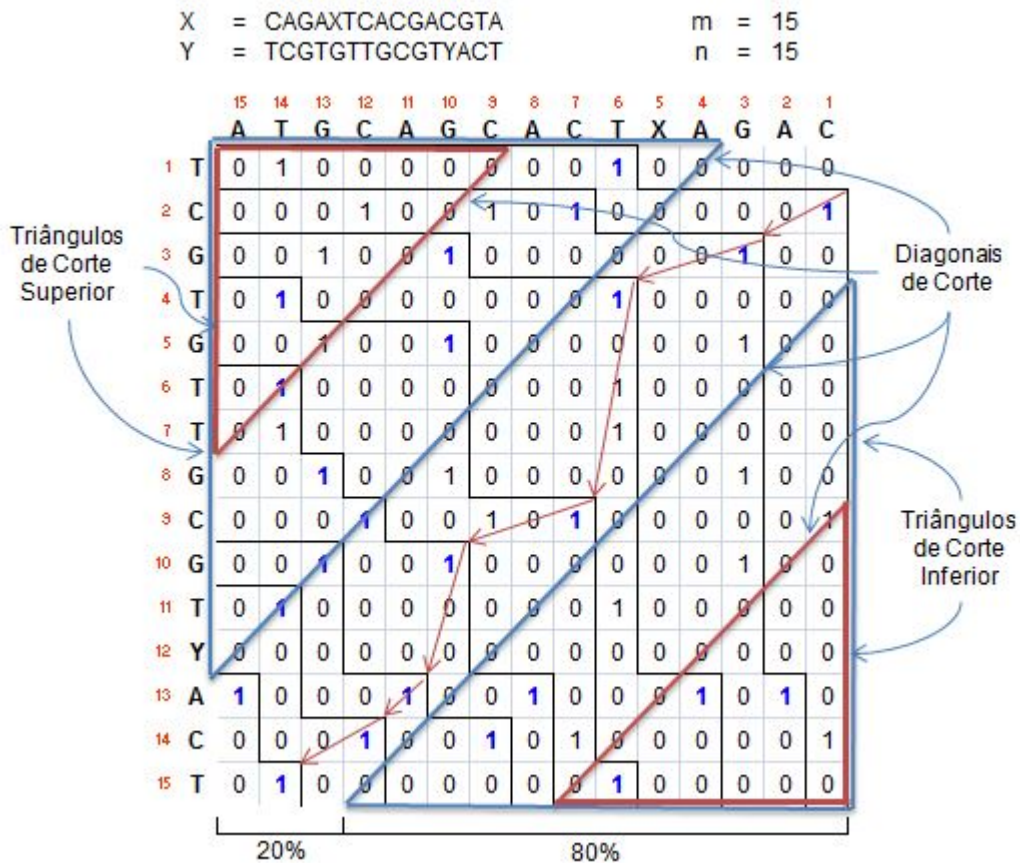


Figura 17 – Representação de diagonais e triângulos de corte sobre no processamento da LLCS.

Estabelecido que buscamos uma LCS com tamanho maior ou igual a  $\lambda$ , o algoritmo bit-paralelo não precisará processar as palavras binárias dentro dos triângulos de corte

com hipotenusa de comprimento igual a  $\lambda$ , pois certamente a LCS buscada não caberia dentro desta área, reduzindo assim o tempo de processamento. Este argumento tem aplicação clara quando estabelecemos um limiar para verificar se duas strings são similares. Digamos por exemplo que se a LCS for menor que 20% as strings não são similares, estamos estabelecendo um limiar de corte e podemos eliminar definitivamente toda a região da matriz onde este limiar de 20% não caberia. Isto não quer dizer que o algoritmo, ajustado para um limite de corte maior que 20%, aplicado a duas strings cuja LCS tenha o comprimento de 20%, não entregaria o valor correto do comprimento da LCS, pois há uma grande probabilidade de que sua LCS esteja realmente fora dos triângulos de corte.

Em algumas aplicações, onde o tempo de processamento é crítico, podemos traçar uma diagonal de corte mais longa. Neste caso, o algoritmo pode entregar um valor com erro, mas este erro será devido aos pontos da LCS que se desviaram da região diagonal central e caíram dentro dos triângulos de corte. Este tipo de erro pode ser controlado com estudo estatístico da repetição dos caracteres nas strings comparadas. Assim, a região diagonal útil pode ser um múltiplo pequeno do tamanho do alfabeto das strings.

No exemplo da [Figura 17](#), as strings tem duas LCS de comprimento igual a 8. O traçado da diagonal de corte em vermelho elimina toda a região onde certamente não haverá subsequência de comprimento maior ou igual a 8 e, conseqüentemente, o algoritmo entregará corretamente o comprimento da LCS. No entanto, as diagonais de corte em azul, ajustadas para cortar 80% do comprimento das strings comparadas, ainda assim entregam corretamente o comprimento da LCS, embora o método tenha processado apenas uma das LCS existentes. O Código [Código 4.10](#) apresenta a implementação do método bit-paralelo com Limites.

```

1  int LCS(char *X, char *Y) {
2      register int i, j, k, m, n;
3      for(m=0; X[m]; m++);
4      for(n=0; Y[n]; n++);
5      if (m==0 || n==0) return 0;
6
7      if (m<n) {
8          char *T;
9          T=X; X=Y; Y=T;
10         k=m; m=n; n=k;
11     }
12
13     int w=(m+31)>>5;           // ceil[m/32]
14
15     uint S[26][w];           // Sigma {a..z}
16     for (i=0; i<26; i++)

```

```

17     for (j=0;j<w;j++)
18         S[i][j]=0;
19
20     uint set=1;
21     for (i=0,j=0;i<m;i++) {
22         S[X[i]-97][j]=set;
23         set<<=1;
24         if (!set) {set++; j++;}
25     }
26
27     int li=0,ls=w;           // words limits
28     int x=w*0.20;          // length of w tested
29     float r=(float)m/n;    // relation length/height
30
31     uint L[w];             // Vector L(i)
32     uint b1,b2,c;         // borrows and carries
33
34     for (i=0;i<w;i++) L[i]=0;
35
36     register uint U,V;
37     for (i=0;i<n;i++){
38         k=(int)(i*r+31)>>5; // ceil k/32)
39         li=max(k-x,0);
40         ls=min(k+x,w);
41         b1=1;b2=0;
42         for (j=li;j<ls;j++) {
43             U = L[j] | S[Y[i]-97][j];
44             c = L[j]>>31;
45             V = U-(L[j]<<1|b1+b2);
46             b1=c; b2=(V>U);
47             L[j] = U & (~V);
48         }
49     }
50     for (k=0,i=0;i<w;i++)
51         for (;L[i];k++)
52             L[i]&=L[i]-1;
53     return k;
54 }

```

Código 4.10 – Solução de implementação bit-paralelo com limites.

Neste código, os limites foram estabelecidos nas Linhas 27 a 29 para acelerar o laço de processamento nas Linhas 42 a 48. O tempo de execução para um corte de 80% foi de

156 ms e para um corte de 95% foi de 46 ms. Ambos entregam o comprimento correto da LCS das strings dos casos de teste. A complexidade é  $O((mn/w) \times (1 - x^2))$  onde  $w$  é o comprimento em bits da palavra e  $x$  é o fator de redução no tamanho da string para definir a diagonal de corte, neste caso,  $w$  é igual a 32 e  $x$  é igual a 0,8.

### 4.3 Discussão

Embora o problema da LCS seja um problema do tipo combinatório e tenha uma fórmula de recorrência imutável, como visto na fundamentação teórica no [Capítulo 2](#), podemos obter soluções eficientes pela utilização de outros algoritmos que explorem um subconjunto menor dos dados. Este trabalho buscou apresentar soluções explorando características nas três vertentes conhecidas do problema: comparação de caracteres; análise gráfica; e poder da arquitetura. A [Tabela 4](#) apresenta o resumo de todas as soluções apresentadas.

As soluções que utilizam comparação de caracteres por programação dinâmica tem uso restrito para strings pequenas ou onde é desejado obter a própria LCS. As soluções que dependem da arquitetura do computador são muito rápidas, mas utilizam mais espaço pois necessitam de uma tabela para armazenar o padrão de bits para cada símbolo do alfabeto enquanto que as soluções gráficas levam grande vantagem em utilização de espaço.

A solução bit-paralelo pode ficar inviável para casos onde o alfabeto seja muito grande. Por exemplo, se considerarmos somente os 256 símbolos ASCII, nossos casos de teste ocupariam 1,6 MB. Por esta razão a solução apresentada utilizou a restrição do alfabeto nos 26 caracteres minúsculos. Como veremos no [Capítulo 5](#) dos Resultados Práticos, a solução bit-paralelo não foi adequada para o problema XMEN por causa da matriz ser muito esparsa e o alfabeto ser muito extenso, comprovando sua inviabilidade nestes casos. Observa-se que a solução bit-paralelo tem complexidade de tempo fixa de  $O(N^2/w)$  e independente do conteúdo. Como a solução diagonal tem complexidade de  $O(R)$  que é aproximadamente  $O(N^2/|\Sigma|)$  é fácil perceber que em termos de complexidade a solução diagonal ultrapassa a solução bit-paralelo quando o tamanho do alfabeto for maior que o tamanho da palavra binária do processador. Na prática, a solução bit-paralelo ainda leva alguma vantagem sobre a solução diagonal por usar códigos de máquina de nível mais baixo mas é superada em algum momento quando  $|\Sigma|$  cresce além de  $w$ .

A solução diagonal é a mais rápida das soluções que não dependem da arquitetura, e apresenta a grande vantagem de utilizar espaço linear independente do conteúdo das strings e do tamanho do alfabeto. O critério de limitação apresentado na solução bit-paralelo também pode ser aplicado nesta solução levando a trabalhar muito mais próximo do conjunto  $K$  que qualquer outra apresentada.



$n^\circ$	Solução	Tempo (s)	O(tempo)	O(espço)
1	Recursiva	infinito	$O(2^{2N})$	$O(N)$
2	Matricial	21,810	$O(mn)$	$O(mn)$
3	Espaço linear	13,050	$O(mn)$	$O(m)$
4	Última linha	2,400	$O(R)+O(m+n)$	$O(m+n)$
5	Limpeza de matriz	2,387	$O(N^2/2)+O(m+n)$	$O(m+n)$
6	Diagonal	1,326	$O(R)+O(m+n)$	$O(m+n)$
7	Bit-paralelo	0,421	$O(mn/w)$	$O(m \Sigma /w)$
8	Bit-paralelo limitado 80%	0,156	$O((mn/w).(1-x^2))$	$O(m \Sigma /w)$
9	Bit-Paralelo limitado 95%	0,046	$O((mn/w).(1-x^2))$	$O(m \Sigma /w)$

Tabela 4 – Resumo dos tempos de execução e das complexidades das soluções apresentadas.

## 5 Resultados Práticos

As soluções apresentadas foram submetidas ao *Sphere Online Judge* (SPOJ) com o objetivo de comparar seu desempenho com as soluções submetidas por usuários de todo o mundo em igualdade de condições. O SPOJ é um repositório de problemas da área da Ciência da Computação, e está disponível a qualquer usuário que deseja testar suas habilidades como programador. O SPOJ tem atualmente mais de 300.000 usuários representando mais de 6.000 universidades de todo o mundo. Mais de 30.000 problemas de diferentes níveis de complexidade estão disponíveis para serem solucionados. Os problemas do SPOJ são apresentados pela sua descrição e suas restrições de tempo, uso de memória e linguagem de programação. É importante destacar que os usuários não tem acesso aos casos de teste, evitando fraudes na solução. Os usuários, ao submeterem suas soluções, recebem como resposta *Accepted* (AC) se resolverem corretamente os casos de teste e atenderem as suas restrições ou podem ser ter suas soluções rejeitadas recebendo uma mensagem de erro. As mais comuns são *Time Limit Exceeded* (TLE), caso a solução exceda o tempo limite para a execução, e *Wrong Answer* (WA), caso a resposta de algum dos casos de teste esteja errada. Um tutorial detalhado pode ser encontrado em <http://www.spoj.com/tutorials/>. As soluções aceitas são listadas em um *ranking* e ordenadas pelo desempenho em tempo. O usuário recebe uma pontuação inversamente proporcional à dificuldade do problema, sendo que a dificuldade do problema é estabelecida pelo número de usuários que apresentam soluções aceitas, isto é, quanto mais usuários apresentarem soluções para um mesmo problema, menor é a pontuação distribuída entre os mesmos. A pontuação acumulada de cada usuário compõe o *ranking* individual mundial e contribui para os *rankings* por instituição e por país. No SPOJ, o Brasil ocupa o 8º lugar e a UFBA ocupa o 327º lugar. A Índia lidera o *ranking* por países e tem quase a metade dos usuários inscritos. Apresentamos em seguida os problemas do SPOJ relacionados com o problema da LCS que foram utilizados para testar as diversas soluções apresentadas.

## 5.1 Problema Longest Common Subsequence - LCS0

O Enunciado do problema LCS0 está descrito na [Tabela 5](#). O problema Longest Common Subsequence - LCS0 ([SPOJ-LCS0, 2012](#)) é bastante objetivo e consiste em implementar um algoritmo que calcule a LLCS de duas strings com até 50.000 caracteres, considerando um alfabeto restrito as letras minúsculas e o tempo de execução limitado em 676 ms.

Enunciado LCS0	
Input:	You will be given two lines. The first line will contain the string A, the second line will contain the string B. Both strings consist of no more than 50000 lowercase Latin letters.
Output:	Output the length of the longest common subsequence of strings A and B.
Example:	
Input:	abababab bcbb
Output:	3
Time limit:	0.676 s
Source limit:	50000B
Memory limit:	1536MB

Tabela 5 – Problema LCS0 do SPOJ.

Este problema foi lançado como desafio durante a disciplina de Laboratório de Programação II (MATA80) no semestre 2014.2 da UFBA, dando origem a este trabalho. A primeira solução foi realizada em 08/Nov/2014, mas o resultado *Accepted* foi obtido somente após a 43ª submissão, em 04/Ago/2015, obtendo o terceiro lugar no *ranking* de usuários, com a solução bit-paralelo. Todas as demais soluções não passaram no limite de tempo para este problema. Posteriormente, submetendo a solução bit-paralelo com limites, aplicando limitador de 80% do comprimento da string, obtive o primeiro lugar em 18/Fev/16, conforme apresentado na [Figura 18](#).

Após algumas otimizações para melhor tratar as matrizes retangulares, aquelas em que os comprimentos das strings comparadas são diferentes, e parametrizando o código para que o mesmo ajuste a limitação para qualquer caso de teste, obtivemos tempo de 90 ms para todos os casos de testes deste problema, utilizando limitação de 95%.

Conforme pode ser visto pela [Figura 18](#), o problema recebeu um total de 3.889 submissões, e que apenas 110 submissões entre 20 usuários foram aceitas, demonstrando que este problema é realmente desafiador. Dentre os usuários listados, destacam-se: Michael Kharitonov (Universidade de Moscow), 259º lugar no *ranking* mundial; Damian Straszak (University of Wrocław - Polônia), 27º no ranking mundial; Takanori Maehara (Academic Researcher at Shizuoka University - Japão), 322º no *ranking* mundial e Mark Gordon

### Longest Common Subsequence statistics & best solutions

Users accepted	Submissions	Accepted	Wrong Answer	Compile Error	Runtime Error	Time Limit Exceeded
20	3889	110	304	529	1605	1336

All ADA ASM ASMGCC AWK BASH BF clang\_c C# C++ 5 clang\_cpp C++ 4.3.2 C++14 C99 strict CLPS CLOJ COB LISP clisp LISP sbcl D dmd clang\_d elixir ERL F# fantom FORT GO gosu 1.6.1 GROOVY HASK ICON ICK JAR JAVA JS J52 LUA NEM NICE nim NODEJS objc clang\_objc CAML PAS gpc PAS fpc PERL PERL 6 PHP pico PIKE PRLG PYTH 2.7 PYPY PYTH 3.4 PYTH 3.2.3 n RUBY rust SCALA SCM guile SCM chicken SCM qobi SED ST TCL VB.net WSPC

RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2016-05-15 17:33:15	Antonio Roberto Paoli	accepted	0.09	2.2M	C
2	2013-06-22 15:24:46	Michael Kharitonov	accepted	0.63	2.2M	C
3	2013-03-10 14:58:51	K_operafan	accepted	1.32	3.1M	C++ 4.3.2
4	2013-08-05 03:31:46	popopolong	accepted	1.62	4.3M	C++ 4.3.2
5	2013-07-28 21:44:16	Arunkumar	accepted	2.18	1.8M	C
6	2016-05-15 20:32:23	Antonio Roberto Paoli	accepted	2.63	4.1M	C++ 5
7	2012-09-04 11:13:23	Damian Straszak	accepted	2.69	3.1M	C++ 4.3.2
8	2016-02-21 10:12:15	Geoffry Song	accepted	2.81	6.6M	HASK
9	2014-07-22 21:34:54	Mark Gordon	accepted	2.95	2.9M	C++ 4.3.2
10	2013-03-09 08:47:39	Takanori MAEHARA	accepted	3.03	2.9M	C++ 4.3.2

Figura 18 – *Ranking* do problema LCS0 no SPOJ <http://www.spoj.com/ranks/LCS0/> em 15/Mai/16.

(USA) 439<sup>o</sup> no *ranking* mundial.

Também pode ser visto na [Figura 18](#) que a posição 6 no *ranking* é o resultado da submissão da solução diagonal com limitação de 95%, demonstrando a potencialidade da solução. A solução diagonal está implementada considerando a sincronia das linhas e colunas e está otimizada para strings do mesmo tamanho. Como é muito provável que os casos de testes deste problema tenham matrizes retangulares, a solução diagonal pode ser melhorada para comportar estas variantes.

## 5.2 Problema Aibohphobia - AIBOHP

O enunciado do problema Aibohphobia (AIBOHP) é apresentado na [Tabela 6](#). Este problema busca obter a mínima distância de edição de uma string com até 6100 caracteres para transformá-la num Palíndromo. Trata-se não da distância de Levenshtein ([LEVENSHTEIN, 1966](#)), mas sim de um caso particular onde só é permitida a inserção e a remoção de caracteres, mas não a substituição. Este caso particular pode ser resolvido pela fórmula  $m - LLCS(X, Y)$ , onde  $X$  é a string de entrada,  $m$  é o comprimento de  $X$ ,  $Y$  é a string reversa de  $X$  e  $LLCS(X, Y)$  é o algoritmo que retorna o comprimento da LCS entre  $X$  e  $Y$ . Este problema não especifica o alfabeto usado, portanto deve ser implementado para receber qualquer caractere ASCII.

Enunciado AIBOHP	
Text:	BuggyD suffers from AIBOHPHOBIA - the fear of Palindromes. A palindrome is a string that reads the same forward and backward. To cure him of this fatal disease, doctors from all over the world discussed his fear and decided to expose him to large number of palindromes. To do this, they decided to play a game with BuggyD. The rules of the game are as follows: BuggyD has to supply a string S. The doctors have to add or insert characters to the string to make it a palindrome. Characters can be inserted anywhere in the string. The doctors took this game very lightly and just appended the reverse of S to the end of S, thus making it a palindrome. For example, if S = "fft", the doctors change the string to "fftff". Nowadays, BuggyD is cured of the disease (having been exposed to a large number of palindromes), but he still wants to continue the game by his rules. He now asks the doctors to insert the minimum number of characters needed to make S a palindrome. Help the doctors accomplish this task. For instance, if S = "fft", the doctors should change the string to "tfft", adding only 1 character.
Input:	The first line of the input contains an integer t, the number of test cases. t test cases follow. Each test case consists of one line, the string S. The length of S will be no more than 6100 characters, and S will contain no whitespace characters.
Output:	For each test case output one line containing a single integer denoting the minimum number of characters that must be inserted into S to make it a palindrome.
Example:	
Input:	1 fft
Output:	1
Time limit:	1,94 s
Source limit:	30000B
Memory limit:	1536MB

Tabela 6 – Problema AIBOHP do SPOJ.

O algoritmo bit-paralelo superou todas as demais soluções obtendo o topo do

ranking entre 3853 usuários que obtiveram sucesso em 17649 submissões, como atesta a Figura 19. Não foram aplicados limites na solução submetida. Vale ressaltar que as soluções matriciais também são aceitas no problema.

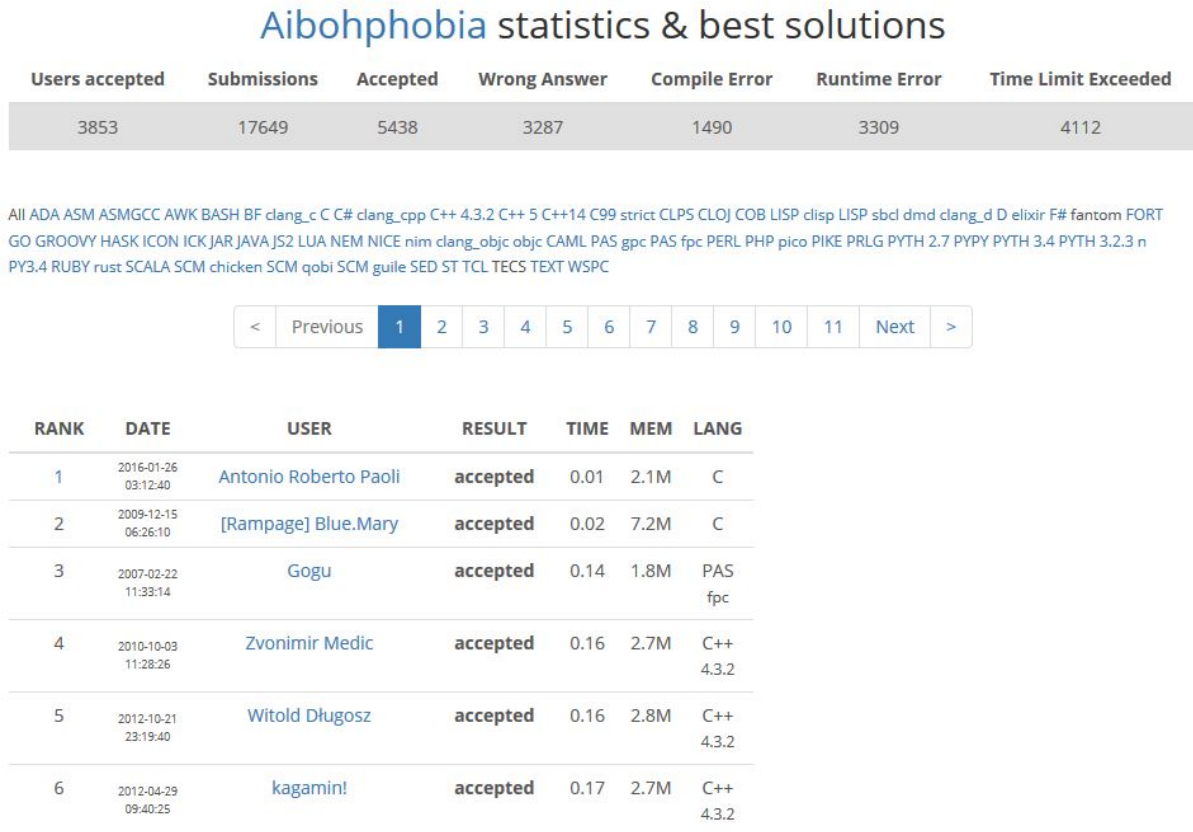


Figura 19 – *Ranking* do problema AIBOHP em [www.spoj.com/ranks/AIBOHP/](http://www.spoj.com/ranks/AIBOHP/) em 15/Mai/2016.

### 5.3 Palindrome 2000 - IOIPALIN

O enunciado do problema Palindrome 2000 - IOIPALIN é apresentado na [Tabela 7](#). Este problema é essencialmente idêntico ao problema anterior, exceto por usar strings de comprimento de até 5.000 caracteres como casos de teste e apresentar um alfabeto maior, que inclui dígitos. O algoritmo bit-paralelo foi bastante eficiente, entretanto não obteve o topo do ranking porque o critério de colocação é tempo de execução e ordem de submissão, ficando então no quinto lugar. Observa-se, entretanto, que o uso de memória foi o menor entre as submissões vencedoras, conforme verifica-se na [Figura 20](#).

Enunciado IOIPALIN	
Text:	A palindrome is a symmetrical string, that is, a string read identically from left to right as well as from right to left. You are to write a program which, given a string, determines the minimal number of characters to be inserted into the string in order to obtain a palindrome. As an example, by inserting 2 characters, the string "Ab3bd" can be transformed into a palindrome ("dAb3bAd" or "Adb3bdA"). However, inserting fewer than 2 characters does not produce a palindrome.
Input:	The first line contains one integer: the length of the input string $N$ , $3 \leq N \leq 5000$ . The second line contains one string with length $N$ . The string is formed from uppercase letters from A to Z, lowercase letters from a to z and digits from 0 to 9. Uppercase and lowercase letters are to be considered distinct.
Output:	The first line contains one integer, which is the desired minimal number.
Example:	
Input:	5 Ab3bd
Output:	2
Time limit:	0,158 s
Source limit:	50000B
Memory limit:	1536MB

Tabela 7 – Problema IOIPALIN do SPOJ.

## Palindrome 2000 statistics & best solutions

Users accepted	Submissions	Accepted	Wrong Answer	Compile Error	Runtime Error	Time Limit Exceeded
1446	7782	1878	1331	450	861	3258

All ADA ASM ASMGCC AWK BASH BF clang\_c C C# clang\_cpp C++ 4.3.2 C++ 5 C++14 C99 strict CLPS CLOJ COB LISP clisp LISP sbcl dmd clang\_d D elixir ERL F# fantom FORT GO GROOVY HASK ICON ICK JAR JAVA JS JS2 LUA NEM NICE nim clang\_objc CAML PAS gpc PAS fpc PERL PHP pico PIKE PRLG PYTH 2.7 PYPY PYTH 3.4 PYTH 3.2.3 n PY3.4 RUBY rust SCALA SCM chicken SCM qobi SCM guile SED ST TCL TECS TEXT WSPC

<	Previous	1	2	3	4	5	6	7	8	9	10	11	Next	>
---	----------	---	---	---	---	---	---	---	---	---	----	----	------	---

RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2010-08-18 15:23:36	[Rampage] Blue.Mary	accepted	0.00	7.0M	C
2	2011-07-22 12:19:54	YAMADA	accepted	0.00	4.3M	C++ 4.3.2
3	2012-08-25 22:07:09	Tooru Ichii	accepted	0.00	8.7M	C++ 4.3.2
4	2012-08-25 22:10:29	Tooru Ichii	accepted	0.00	15M	C++ 5
5	2016-01-26 03:40:44	Antonio Roberto Paoli	accepted	0.00	2.1M	C
6	2015-08-17 08:00:31	arunpatala	accepted	0.14	2.2M	C
7	2012-01-14 19:58:05	Aleksandar Ivanović	accepted	0.15	3.1M	C++ 5

Figura 20 – Ranking do problema IOIPALIN em [www.spoj.com/ranks/IOIPALIN/](http://www.spoj.com/ranks/IOIPALIN/) em 15/Mai/2016.

O código submetido foi o bit-paralelo sem nenhum limite. Foi também submetido um código bit-paralelo com limite em 20%, que recebeu *Wrong Answer*. Nota-se que o problema pede soluções para strings contendo letras maiúsculas, letras minúsculas e números, sendo provável que as strings dos casos de testes sejam preparadas especificamente para resoluções de palíndromos, com distribuições de caracteres com regularidades próprias para casos limite. Este é um caso onde o uso de limites não se aplica. As demais soluções não passam neste problema porque o limite de tempo do problema é bastante restrito.



## 5.4 Problema X-Men - XMEN

O enunciado do problema XMEN é apresentado na [Tabela 8](#). O problema XMEN não é um problema de LCS, e sim um problema de Subsequência Crescente Máxima (LIS, *Longest Increasing Subsequence*). A distribuição dos símbolos em cada string deste problema é uma permutação de um alfabeto finito, contínuo e ordenado, de tamanho igual ao comprimento da string. Para resolver este problema como LIS é preciso reindexar as duas permutações, renomeando seus símbolos, para que tenhamos uma string ordenada a ser comparada com uma permutação, e calcular a mais longa subsequência crescente desta permutação. A complexidade da solução tradicional para a LIS é  $O(N \log N)$  existindo, para o caso particular da permutação uma solução com complexidade de tempo de  $O(N \log \log N)$ , não sendo este o objeto deste trabalho.

Enunciado XMEN	
Text:	Dr. Charles Xavier is trying to check the correlation between the DNA samples of Magneto and Wolverine. Both the DNAs are of length N, and can be described by using all integers between 1 to N exactly once. The correlation between two DNAs is defined as the Longest Common Subsequence of both the DNAs. Help Dr. Xavier find the correlation between the two DNAs.
Input:	First line of input contains number of Test Cases T. Each test case starts with an integer N, size of DNA. Next two lines contain N integers each, first line depicting the sequence of Magneto's DNA and second line depicting Wolverine's DNA.
Output:	For each test case print one integer, the correlation between the two DNAs.
Example:	
Input:	2 2 1 2 2 1 3 1 2 3 1 3 2
Output:	1 2
Constraints:	$1 \leq T \leq 10$ $1 \leq N \leq 100000$
Time limit:	1 s
Source limit:	50000B
Memory limit:	1536MB

Tabela 8 – Problema XMEN do SPOJ.

Na [Figura 21](#) apresentamos do lado esquerdo, um modelo do problema XMEN com duas permutações e do lado direito o mesmo problema após a reindexação resumindo o problema a uma única permutação. A reindexação é equivalente a uma renomeação dos símbolos e a sequência a ser analisada com a utilização de um algoritmo LIS é  $\{5, 7, 10, 2, 14, 15, 4, 6, 11, 9, 8, 12, 13, 1, 3\}$ , conforme apresentado nesta figura. A matriz que

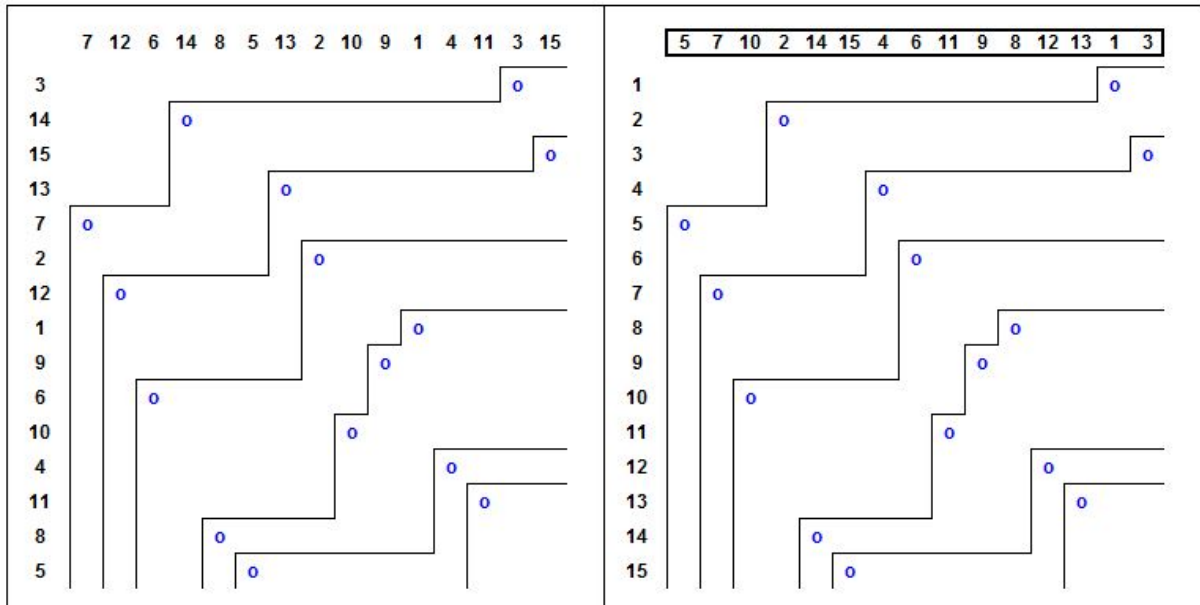


Figura 21 – Exemplo para o problema XMEN. (a) matriz de permutações e (b) matriz reindexada.

combina duas permutações é bastante esparsa, contendo apenas um ponto em cada linha e um ponto em cada coluna, ou seja, o conjunto  $R$  tem tamanho igual a  $N$ .

Embora este problema seja apresentado como LCS, não é para ser resolvido como tal, pois existem algoritmos mais eficientes para tratar permutações, mas, aqui a solução diagonal surpreende e mostra todo seu potencial. De fato, exceto a solução diagonal, todas as demais soluções apresentadas neste trabalho falharam ao tratar o problema como LCS, inclusive a solução bit-paralelo.

A razão do sucesso da solução diagonal pode ser facilmente explicada pela complexidade de tempo da solução que é  $O(R)$  ou  $O(N^2/N)$ , assim  $R$  é igual a  $N$ . A solução bit-paralelo falha por ter complexidade constante de  $O(N^2/w)$  e neste caso  $w \ll N$ .

Como não temos acesso aos casos de teste do SPOJ, criamos 3 casos especiais de teste com tamanho de 100.000 símbolos para comparar o desempenho das soluções em ambiente controlado. Os resultados foram resumidos na Tabela 9. A primeira coluna lista a solução usada, a segunda coluna apresenta o tempo para um caso de teste em que as strings  $X$  e  $Y$  são idênticas, a terceira coluna mostra um caso de teste onde  $Y$  é o reverso de  $X$ , a quarta coluna mostra o tempo para um caso de teste com strings geradas aleatoriamente, que possui LCS de comprimento 820, e a quinta coluna o resultado da submissão no SPOJ.

Algumas conclusões podem ser tiradas desta tabela. (1) A solução Limpeza de Matriz degenera-se para  $O(N^2/2)$  quando as strings são iguais, devido ao fato do algoritmo testar, a cada linha, todas as linhas seguintes para verificar se as fronteiras podem ser

Solução	Caso Dir	Caso Rev	Caso Rnd	SPOJ
Limpeza de matriz	9189	31	140	TLE
Bit-paralelo	1310	1310	1310	TLE
Bit-paralelo com limite	203	202	202	WA
Diagonal	31	31	46	ACC 210 ms
LIS	31	31	31	ACC 180 ms

Tabela 9 – Tempo em ms das soluções aplicadas ao Problema XMEN e resultado no SPOJ.

melhoradas, o que nunca ocorre, mas possui desempenho ótimo para strings reversas, onde todos os testes são positivos. (2) A solução bit-paralelo obteve tempo de execução idêntico para todos os casos de teste, provando que é independente do conteúdo. A solução bit-paralelo é inviável para permutações, pois calcula desnecessariamente toda a linha para mover um único bit. (3) A solução bit-paralelo com limites poderia ter acelerado o tempo de execução, mas com uma matriz extremamente esparsa, um único ponto que não seja considerado implicará necessariamente em uma resposta errada, como ocorreu no SPOJ. Isto mostra que a limitação somente se aplica a problemas de alfabeto pequeno em relação ao comprimento das strings comparadas. (4) A solução diagonal foi a única aceita, com desempenho ótimo nos dois casos extremos de teste, deteriorando-se levemente no caso randômico em que alguns pontos dominantes se afastaram da diagonal principal.

Para comparação, a solução LIS em  $O(N \log N)$  foi submetida e verificou-se que esta solução depende integralmente do tamanho do conjunto  $R$ , mas não da sua distribuição. Com este problema mostramos que a solução diagonal proposta neste trabalho é muito promissora para a LLCS pois os seus resultados, pelo menos para permutações, são comparáveis aos obtidos pela LIS e superaram a solução bit-paralelo.

## 6 Conclusão

O problema LCS é um problema combinatório e envolve operar direta ou indiretamente nos pares ordenados  $(x, y)$  que duas strings geram ao combinar seus caracteres alinhados sobre os eixos  $x$  e  $y$ , respectivamente. É um problema bidimensional em sua essência e uma abordagem convencional sempre nos levará a soluções de complexidade  $O(N^2)$  de tempo e espaço, pois o conjunto destes pares contém  $(m \times n)$  elementos.

Soluções viáveis para o uso cotidiano envolvem a diminuição do tamanho deste universo tomando como base somente os dados essencialmente necessários, pré-processando e descartando imediatamente todos elementos que não contribuirão para a solução. Três subconjuntos são bem definidos no universo de pares da matriz  $L$ : o subconjunto  $R$  dos pares ordenados que possuem coordenadas cujos caracteres correspondentes possuem uma coincidência; O subconjunto  $D$  de  $R$  com os pares que possuem maior relevância no traçado das zonas de fronteira da LCS; e o subconjunto de pares ordenados  $K$  cujos elementos pertencem a pelo menos uma das LCS possíveis. Desta forma,  $LCS \subseteq K \subseteq D \subseteq R \subseteq L$ .

A solução recursiva é inviável para qualquer aplicação porque seu tempo de execução cresce exponencialmente. A solução matricial opera totalmente em  $L$  e não é eficiente, pois o seu tempo de execução cresce quadraticamente com crescimento do comprimento das strings. Porém, a solução é útil para strings pequenas e para aqueles casos onde é preciso recuperar a própria LCS. As soluções última linha, limpeza de matriz e diagonal operam parcialmente em  $R$  e nem mesmo utilizam uma matriz como apoio de dados, porém sua execução depende do conteúdo das strings. Destas três soluções, a diagonal é a que opera mais próximo do conjunto  $D$ . As soluções bit-paralelo operam completamente em  $L$ , obtendo ganho de velocidade por meio de paralelismo de acordo com o tamanho em bits da palavra do processador. Estas soluções não tem dependência direta do conteúdo das strings mas tem dependência do tamanho do alfabeto, além da dependência do hardware empregado.

Embora os casos de teste tivessem um comprimento longo, de 50.000 caracteres, permitindo experimentações práticas, não foi abordado neste trabalho os casos de busca da LCS para strings verdadeiramente longas. Estes casos requerem processamento em linha, ou seja, sem o pré-processamento utilizado em algumas das soluções apresentadas. A abordagem utilizada nas soluções bit-paralelo com limites e diagonal são mais promissoras para resolução do problema da LCS em linha. A solução diagonal apresentada é promissora, pois obteve o menor tempo de execução entre as soluções independentes da arquitetura e tem complexidade de espaço independente do tamanho do alfabeto, por isso, a solução diagonal supera a solução bit-paralelo quando o alfabeto se torna muito grande. Como

demonstrado no exemplo prático *XMEN*, a solução bit-paralelo tem complexidade de espaço de  $O(|\Sigma| \times N)$  e mesmo que o processamento do alfabeto seja feito em linha, a execução é em  $O(L/w)$  que é muito maior que  $O(R)$  da solução diagonal. Por outro lado, acreditamos que é promissor pesquisar o uso do bit-paralelismo na solução diagonal. Durante a etapa de pesquisa, não foram encontrados trabalhos que dividam a matriz LCS pela diagonal principal, sendo esta a principal das contribuições deste trabalho. Assim concluimos que a análise das propriedades da LCS ao longo da diagonal de sua matriz abre uma nova linha de pesquisa para este problema.

# Referências

ALLISON, L.; DIX, T. I. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, v. 23, n. 5, p. 305 – 310, 1986. ISSN 0020-0190. Citado 3 vezes nas páginas 9, 47 e 49.

ALSMADI, I.; NUSER, M. String matching evaluation methods for dna comparison. *International Journal of Advanced Science and Technology*, v. 47, n. 1, p. p13–32, 2012. Citado na página 13.

CHIN, F. Y. L.; POON, C. K. A fast algorithm for computing longest common subsequences of small alphabet size. *J. Inf. Process.*, Information Processing Society of Japan, Tokyo, Japan, Japan, v. 13, n. 4, p. 463–469, abr. 1991. ISSN 0387-6101. Disponível em: <<http://dl.acm.org/citation.cfm?id=105582.105592>>. Citado na página 14.

CORMEN, T. H. et al. *Introduction to Algorithms, Third Edition*. 3rd. ed. [S.l.]: The MIT Press Cambridge, London, England, 2009. ISBN 0262033844, 9780262033848. Citado 4 vezes nas páginas 16, 19, 20 e 21.

CROCHEMORE, M.; ILIOPOULOS, C. S.; PINZON, Y. J. *Speeding-up Hirschberg and Hunt-Szymanski LCS Algorithms*. 2003. Citado na página 25.

CROCHEMORE, M. et al. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, v. 80, p. 279–285, 2000. Citado 4 vezes nas páginas 9, 47, 49 e 50.

CROCHEMORE, M.; PORAT, E. Computing a longest increasing subsequence of length  $k$  in time  $o(n \log \log k)$ . In: *Proceedings of the 2008 International Conference on Visions of Computer Science: BCS International Academic Conference*. Swinton, UK, UK: British Computer Society, 2008. (VoCS'08), p. 69–74. Disponível em: <<http://dl.acm.org/citation.cfm?id=2227536.2227543>>. Citado 2 vezes nas páginas 13 e 14.

HIRSCHBERG, D. S. Algorithms for the longest common subsequence problem. *J. ACM*, 1977. Citado na página 26.

HUNT, J. W.; MCILROY, M. D. *An Algorithm for Differential File Comparison*. Murray Hill, NJ, 1976. Citado na página 13.

HUNT, J. W.; SZYMANSKI, T. G. A fast algorithm for computing longest common subsequences. *Commun. ACM*, ACM, New York, NY, USA, v. 20, n. 5, p. 350–353, maio 1977. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/359581.359603>>. Citado na página 29.

HYRÖ, H. Bit-parallel lcs-length computation revisited. In: *In Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA)*. [S.l.: s.n.], 2004. p. 16–27. Citado 4 vezes nas páginas 9, 47, 49 e 50.

KENT, C. K.; SALIM, N. Features based text similarity detection. *CoRR*, abs/1001.3487, 2010. Disponível em: <<http://arxiv.org/abs/1001.3487>>. Citado na página 13.

- KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language 2nd Ed.* [S.l.: s.n.], 1988. Citado 2 vezes nas páginas 49 e 50.
- LEVENSHTAIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, Vol. 10, No. 8. (1966), pp. 707-710, 1966. Citado 2 vezes nas páginas 13 e 60.
- MYERS, E. W. An o(nd) difference algorithm and its variations. *Algorithmica*, v. 1, p. 251–266, 1986. Citado na página 13.
- NAKATSU, N.; KAMBAYASHI, Y.; YAJIMA, S. A longest common subsequence algorithm suitable for similar text strings. *Acta Inf.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 18, n. 2, p. 171–179, nov. 1982. ISSN 0001-5903. Disponível em: <<http://dx.doi.org/10.1007/BF00264437>>. Citado na página 68.
- SPOJ-LCS0. *Longest Common Subsequence*. 2012. <<http://http://www.spoj.com/ranks/LCS0/>>. Accessed: 2016-02-22. Citado na página 58.
- TJANDRAATMADJA, C. Problema da subsequência máxima comum sem repetições. 2010. Citado 2 vezes nas páginas 13 e 68.
- ULLMAN, J. D.; AHO, A. V.; HIRSCHBERG, D. S. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, ACM, New York, NY, USA, v. 23, n. 1, p. 1–12, jan. 1976. ISSN 0004-5411. Citado na página 68.
- WAGNER, R. A.; FISCHER, M. J. The string-to-string correction problem. *J. ACM*, ACM, New York, NY, USA, v. 21, n. 1, p. 168–173, jan. 1974. ISSN 0004-5411. Disponível em: <<http://doi.acm.org/10.1145/321796.321811>>. Citado na página 13.
- WANG, Q.; KORKIN, D.; SHANG, Y. *Efficient Dominant Point Algorithms for the Multiple Longest Common Subsequence (MLCS) Problem*. 2009. Disponível em: <<https://www.aaai.org/ocs/index.php/IJCAI/IJCAI-09/paper/view/293>>. Citado na página 13.
- WANG, Y.-H. Image indexing and similarity retrieval based on spatial relationship model. *Inf. Sci. Inf. Comput. Sci.*, Elsevier Science Inc., New York, NY, USA, v. 154, n. 1-2, p. 39–58, ago. 2003. ISSN 0020-0255. Disponível em: <[http://dx.doi.org/10.1016/S0020-0255\(03\)00005-7](http://dx.doi.org/10.1016/S0020-0255(03)00005-7)>. Citado na página 13.