Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

DCC005 - ALGORITMOS E ESTRUTURAS DE DADOS III - TURMA TE

Trabalho Prático 4 – Erros de Português

Guilherme Saulo Alves

guisaulo@hotmail.com

Professores - Jussara Marques, Olga Nikolaevna,

Marcos Augusto, Wagner Meira

1. Introdução

Este trabalho descreve uma solução para o problema de correção ortográfica de palavras. Um certo professor aplicou uma prova discursiva e percebeu erros de português nas respostas dos alunos. O professor decidiu verificar as palavras onde os erros eram cometidos e penalizar o aluno de acordo com o "tamanho" dos erros de grafia. A solução proposta foi implementado baseada no *algoritmo de Distância de Levenshtein,* capaz de calcular a distância de edição entre duas palavras.

O método de correção compara as respostas dos alunos com um dicionário de palavras aceitas. Feita a comparação, o algoritmo indica qual o custo mínimo associado a um conjunto de operações capaz de transformar as palavras incorretas nas palavras do dicionário. Quatro operações são realizadas sobre as palavras erradas para transformá-las em palavras corretas: *Inserção, Remoção, Substituição* e *Troca.*

O algoritmo implementado se baseia no paradigma de programação dinâmica para calcular a distância entre duas palavras. Esse problema é foco de estudos em Ciência da Computação, especialmente na área de processamentos de cadeias de caracteres. O método tem grande aplicabilidade em editores de texto utilizados em diversas áreas. A proposta da implementação do trabalho consiste em três formas distintas. A primeira de programação dinâmica sequencial, a segunda de programação dinâmica com paralelização de palavras e a terceira de programação dinâmica com computações paralelas internas. Mais detalhes da implementação são abordados ao longo da documentação.

2. Modelagem e Solução Proposta

Abaixo seguem as implementações das soluções propostas para o problema. Os algoritmos foram explicados de forma clara, através de pseudocódigos e esquemas ilustrativos.

2.1. Considerações Iniciais

O ambiente de desenvolvimento do código foi via compilador GNU GCC Compiler via linha de comando no Sistema Operacional Ubuntu Linux 13.04.

Para gerar o programa executável, utiliza-se o comando make no shell do Linux. O programa gerado possui nome tp4 e deve aceitar, independente da ordem, os seguintes argumentos:

```
./tp4 -a <algoritmo> -t <número threads> - r <respostas>
-d <dicionário> -o <saída>
```

2.2. Modularidade

O código está dividido em nove arquivos principais:

- **main.c:** Arquivo principal do programa, contém a chamada de execução da função principal;
- **pds.c:** Implementação sequencial do algoritmo por meio do uso de técnicas de programação dinâmica;
- **pds.h:** Possui as declarações da Implementação sequencial do algoritmo;
- **pdpp.c:** Implementação paralela do algoritmo por meio do uso de técnicas de paralelismo de palavras;
- **pdpp.h:** possui as declarações da Implementação paralela do algoritmo
- **pdcpi.c:** Implementação paralela do algoritmo por meio do uso de técnicas de paralelismo de computações Internas;
- **pdcpi.h:** possui as declarações da Implementação paralela do algoritmo;
- **lista.c:** possui as funções usadas por uma estrutura de dados básica lista;
- **lista.h:** possui o TAD e a declaração das funções usadas por uma estrutura de dados básica lista.

2.3. Funções, Estruturas e Algoritmos

As variáveis e funções possuem nomes que facilitam a análise do código. Vale ressaltar que, apesar de não serem concluídas, as propostas das soluções paralelizadas também serão apresentadas e comentadas no código.

2.3.1. Modelagem

Modelamos o trabalho com um TAD lista que é responsável pela gravação das respostas de cada aluno. A lista foi uma ferramenta para poder imprimir a distância total antes das palavras no arquivo de saída. A complexidade de espaço é *O(n)* em que *n* é a quantidade de respostas de um aluno. A célula da lista é composta pelos seguintes itemns:

```
typedef struct {
        float Distancia; //guarda a distância para uma resposta
        char *Palavra;  //guarda a resposta corrigida
} TipoResposta;
```

2.3.1. Programa Principal

- `int main(int argc, char *argv[]);`

Contém a chamada da função principal. Para facilitar a compressão, o programa principal possui duas partes. Na primeira parte, o programa verifica os argumentos de execução e inicializa algumas variáveis de controle. Na segunda parte, o programa principal chama o algoritmo correspondente, passado nos argumentos de execução, para umas das 3 formas de implementação da solução proposta: 1 – Programação dinâmica sequencial, 2 - Programação Dinâmica com Paralelização de Palavras, 3 - Programação Dinâmica com Computação Paralelas Internas.

2.3.2. Programação Dinâmica Sequencial

- `void PPSequencial(char *NomArqRespostas, char*NomArqDicionario, char *NomArqSaida);`

A função lê caracter por caracter dos arquivos de entrada formando as strings. Quando as strings estão prontas, elas são passadas para a função levenshtein. A função também imprime os custos de edição no arquivo de saída. O processo repete até as respostas acabarem. Veja o pseudocódigo abaixo:

Pseudocódigo 1 Programação Dinâmica Sequencial

```
1: FazListaVazia (Lista);
2: while char1 ∈ ArqResposta != EOF do {lê caracter por caracter}
3:              if char1 = '\n' then {indica que as respostas de um aluno acabaram}
4:                      FimDaLinha ← TRUE
5:                      lock1 ← TRUE {libera a string à função levenshtein}
6:              end if
7:              if char1 = ' ' then {indica que uma string se formou}
8:                      lock1 ← TRUE
9:              end if
10:             if lock1 then {só é liberado quando uma palavra já esta formada}
11:                     while char2 ∈ ArqDicionario != EOF do
12:                             if char2 = '\n' then
13:                                     lock2 ← TRUE {libera a string à função levenshtein}
14:                             end if
15:                             if lock1 && lock2 {armazena a menor distância e a strCerta}
16:                                     Distancia ← levenshtein (strCerta, strErrada);
17:                                     if Distancia < MenorDistancia
18:                                             MenorDistancia ← Distancia
19:                                     end if
20:                                     lock2 ← FALSE;
21:                             end if
22:                             else if char2 is valid then {continua formar a string}
23:                                     strCerta ← char2
24:                             end if
25:                     end while
26:                     if !FimDaLinha {Insere uma resposta e distancia de uma aluno na lista}
27:                             DistanciaTotal ← MenorDistancia
28:                             InsereResposta (MenorDistancia, strCerta, Lista);
29:                     end if
30:                     else if FimDaLinha {Imprime a lista no arquivo e esvazia a lista}
31:                             DistanciaTotal ← MenorDistancia
32:                             InsereResposta (MenorDistancia, strCerta, Lista);
33:                             ImprimeRespostaAluno (Lista);
34:                             RetiraResposta (Lista);
35:                             DistanciaTotal = 0;
36:                     end if
37:                     lock1 ← FALSE
38:             end if
39:             else if char1 is valid then {continua formar a string}
40:                     strErrada ← char1
41:             end if
42:             FimDaLinha ← FALSE
43: end while
```

- `float levenshtein(const char *palavra1, const char *palavra2);`

Contém o algoritmo de *distância de Levenstein* que calcula a distância de edição entre duas strings. A distância é dada pelo número mínimo de operações necessárias para transformar uma string na outra. Essas operações são inserção, remoção, substituição e troca. O algoritmo é uma implementação do paradigma de programação dinâmica onde há ganho de desempenho em problemas que possuem superposição de subproblemas e substrutura ótima. Segue as operações e sua formulação alternativa abaixo. Observe que a operação de troca equivale ao custo de uma inserção e uma remoção, sendo facultativo implementar a troca ou não.

Inserção = M[i][j-1]+1; (custo 1)

Remoção = M[i-1][j] + 1; (custo 1)

Substituição = [i-1][j-1] + 1.5; (custo 1.5)

Troca = (M[i][j-1] + 1) + (M[i-1][j] + 1); (custo 2)

M[i][0] ←i M[0][j] ←j

$$M[i][j] \leftarrow \begin{cases} M[i-1][j-1] \leftarrow 0 & \text{se } xi = yj \\ Min\ (Inserção, Remoção, Substituição, Troca) & \text{caso contrário} \end{cases}$$

A técnica utiliza uma matriz *M[n+1][m+1]*, na qual *n* e *m* são os números de caracteres das duas strings. A matriz *M[i][j]* representa o menor número de operações de edição necessárias para casar *x1..i* com *y1..j*, onde *x* e *y* são caracteres das strings.

Por exemplo, para converter a palavra *matrandsa* na palavra *saturadas* o custo é 5.5, pois temos uma substituição de *m* por *s* (custo 1.5), retirada do *n* (custo 1), inserção do *u* depois do *t* (custo 1) e uma troca do *a* pelo *s* no final da cadeia (custo 2).

No final, o elemento do fundo ao lado direito da matriz contém a resposta. A sua complexidade é O(mn), onde m é o tamanho da primeira cadeia de caracteres e n o tamanho da segunda. Veja o exemplo a seguir que calcula a distância de edição igual a 3 para transformar a palavra *SUNDAY* em *SATURDAY* preenchendo a matriz com os custos*:*

		S	***A***	***T***	***U***	***R***	***D***	***A***	***Y***
	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
S	1.0	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0
U	2.0	1.0	1.5	2.5	2.0	3.0	4.0	5.0	6.0
N	3.0	2.0	2.5	3.0	3.0	3.5	4.5	5.5	6.5
D	4.0	3.0	3.5	4.0	4.0	4.5	3.5	4.5	5.5
A	5.0	4.0	3.0	4.0	5.0	5.5	4.5	3.5	4.5
Y	6.0	5.0	4.0	4.5	5.5	6.5	5.5	4.5	3.5

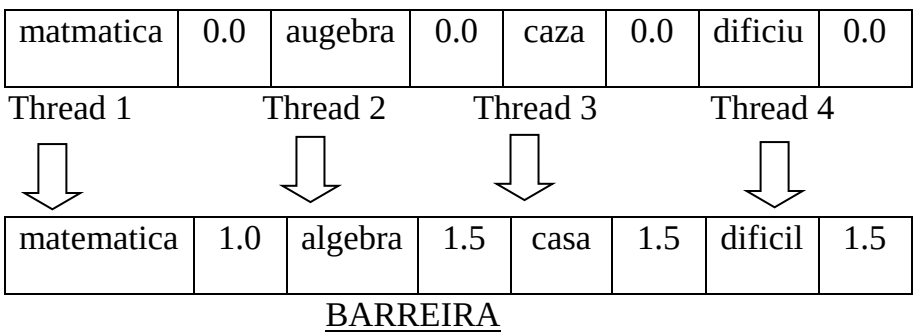
Pseudocódigo 2 Algoritmo de Levenshtein

```
1: Levenstein ← str1, str2, m, n
2: for i ← 0 to m
3:       M[i][0] ← i
4: end for
5: for j ← 0 to n
6:       M[0][j] ← j
7: end for
8: for j ← 1 to n
9:       for i ← 1 to m
10:            if str1[i] == str2[j]
11:                 M[i][j] ← M[i-1][j-1]
12:            end if
13:            else
14:                 Inserção ← M[i][j-1]+1;
15:                 Remoção ← M[i-1][j] + 1;
16:                 Substituição ← M[i-1][j-1] + 1.5;
17:                 Troca ← (M[i][j-1] + 1) + (M[i-1][j] + 1);
17:                 Min ← Inserção
18:                 if Remoção <Min
19:                      Min ←Remoção
20:                 end if
21:                 if Substituição < Min
22:                      Min ←Substituição
23:                 end if
24:                 if Troca < Min
25:                      Min ← Troca
26:                 end if
27:                 M[i][j] ← Min
28:            end else
29:      end for
30: end for
31: return M[n][m] {resposta}
```

2.3.3. Programação Dinâmica com Paralelização de Palavras

```
void PPPalavras(char *NomArqRespostas, char *NomArqDicionario,
                char *NomArqSaida, int numThreads);
```

A solução proposta consiste em utilizar a mesma implementação do algoritmo de programação dinâmica sequencial, porém, mais de um par de palavras deve ser comparada simultaneamente. A estratégia utiliza a lista com as palavras erradas de um dado aluno. Veja a ilustração abaixo:



Cada palavra errada na lista é comparada com as palavras do dicionário ao mesmo tempo, distribuídas entre threads. Teremos uma barreira para indicar que todas as palavras do aluno foram comparadas. Com a barreira liberada, podemos imprimir as palavras corrigidas e suas distancias no arquivo de saída. No fim do paralelismo, um procedimento com um novo aluno pode ser realizado. A estratégica minimiza a comunicação entre threads, pois cada comparação entre as palavras é executada de forma independente. Porém, uma desvantagem da implementação seria o desbalanceamento de carga, pois uma palavra muito grande pode demorar ser processada por um thread, aumentando assim o tempo de espera de outros threads que já estejam prontas.

2.3.4. Programação Dinâmica com Computações Paralelas Internas

- `void PPCompInternas(char *NomArqRespostas, char*NomArqDicionario, char *NomArqSaida, int numThreads);`

A solução proposta consiste em processar cada elemento da matriz gerada pelo algoritmo de programação dinâmica. Para isso, temos que estudar as dependências de dados que existem para o cálculo de cada um dos custos de edição. Veja a ilustração abaixo:

1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13

A Matriz representa as dependências do algoritmo de Levenshtein. Uma vez que a célula 1 foi concluída, ela flui informação para as células leste, sul e sudeste. Assim a célula 2 só pode ser processada depois da célula 1 ser processada. Uma vez que a célula 2 fluir suas informações, a célula 3 pode ser tradada, assim sucessivamente. Esse tipo de comportamento será a estratégia de paralelismo para calcular diferentes células em threads separadas.

A estratégica tenta minimizar a comunicação entre threads (sincronização de threads para comunicação de dados) para minimizar o tempo de espera entre as threads. Além disso, fornecemos equilíbrio de carga de modo que cada thread faça a mesma quantidade de trabalho. A técnica consiste em dividir a matriz em blocos de tamanho variável, de forma que cada thread opere em um bloco especifico. Veja o esquema ilustrativo abaixo, para 3 threads disponíveis:

		S	***A***	***T***	***U***	***R***	***D***	***A***	***Y***
	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
S	1.0	T1		T2		T3		T1	
U	2.0								
N	3.0	T2		T3		T1		T2	
D	4.0								
A	5.0	T3		T1	T2	T3	T1	T2	T3
Y	6.0			T1	T2	T3	T1	T2	T3

Utilizamos blocos de tamanho grande primeiro depois os pequenos, devido ao aumento dos custos de processamento. Isso nos permite equilibrar a carga sobre todos os cálculos e ainda obter as vantagens de minimizar necessidade de comunicação.

2.4. Entrada e Saída

Para este trabalho, a entrada consiste em dois arquivos de textos separados. O primeiro contendo um dicionário de palavras corretas, uma por linha, e o segundo arquivo contendo as respostas dos alunos, um aluno por linha. As respostas do aluno estão separadas por um espaço em branco. Para cada palavra do arquivo de respostas e necessário calcular a distância de edição entre essa palavra e todas as demais do dicionário, selecionando a palavra do dicionário que tiver menor distância de edição.

Em caso de empate, quando a palavra tiver a mesma distância de edição para duas palavras distintas do dicionário, foi utilizada a palavra que aparece primeiro no dicionário. A seguir temos um exemplo de entrada:

Dicionario:

```
aspecto
matematica
algebra
posicao
profissao
fourier
casa
calculo
dificil
```

Respostas:

```
aepscto caculo
matmatica augebra caza dificiu
posissao proficao furier cauculo
```

No arquivo de saída é impresso a diferença total entre as palavras escritas pelos alunos e as palavras corretas no dicionário, juntamente com sua distância para cada palavra escrita originalmente, na ordem das palavras na resposta. Veja a saída do exemplo anterior:

Saída:

```
4.0: aspecto,3.0 calculo,1.0
5.5: matematica,1.0 algebra,1.5 casa,1.5 dificil,1.5
7.5: posicao,2.5 posicao,2.5 fourier,1.0 calculo,1.5
```

- Foi considerado que as palavras terão somente letras, e as letras maiúsculas são diferentes de letras minúsculas.
- Como os termos dos arquivos podem ter tamanho variável, a leitura dos arquivos foi feita caractere/caractere formando uma palavra dinamicamente;
- Para seguir o formato do arquivo de saída descrito na especificação, utilizou-se variáveis float e TAD lista para guarda as repostas dos alunos e imprimir posteriormente, tomando que a solução caiba na memória principal.

3. Analise de Complexidade

A solução proposta consiste na implementação das funções de Programação Dinâmica para ler as palavras do arquivo e o Algoritmo de Levenstein. Para análise de complexidade de tempo e espaço considere os seguintes aspectos:

- Suponha *p* a quantidade de palavras do arquivo de respostas, *q* a quantidade de palavras do arquivo dicionário, *m* é o tamanho da primeira cadeia de caracteres e *n* o tamanho da segunda cadeia de caractere;

3.1. Programação Dinâmica Leitura Arquivos

Complexidade de Espaço: O principal uso de memória consiste alocar espaço para uma lista que contém as respostas e suas distâncias dos alunos. Logo a complexidade de espaço é ***O(n).***

Complexidade de Tempo: O principal consumo de tempo consiste nos 2 loops que leem as palavras dos arquivos. Logo a complexidade de tempo é ***O(pq).***

3.2. Algoritmo de Levenshtein

Complexidade de Espaço: O principal uso da memória consiste alocar uma matriz *M[m][n]* para armazenar os custos de distância entre duas palavras. Logo a complexidade de espaço é ***O(mn)***.

Complexidade de Tempo: O principal consumo de tempo consiste na comparação de cada palavra errada com todas palavras do dicionário. Logo a complexidade de tempo é ***O(mn)***.

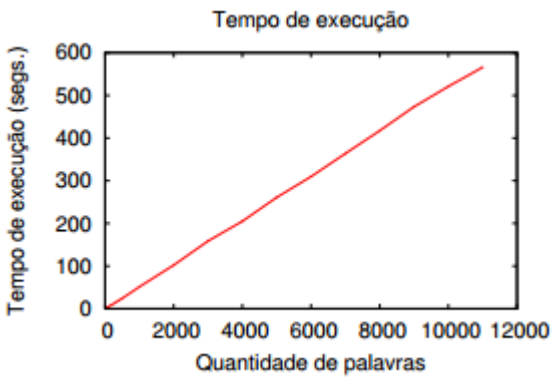
4. Experimentos

Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2Gb de memória. Testes com dicionário muito extenso acarretou em falhas de segmentação.

4.1. Tabela e Gráfico

Para avaliar o funcionamento do programa, utilizou-se vários tamanhos de entradas com palavras erradas aleatórias, mantendo-se constante os tamanhos do dicionário. Não foi possível finalizar a partes das implementações paralelas, por isso apresentamos apenas testes da parte sequencial. Segue os esquemas com as respectivas analises, verificando o tempo de execução. Note que o tempo de execução é linearmente crescente, como esperado da complexidade do algoritmo sequencial.

Palavras	Tempo (segs)
10	0,004
50	2,782
100	4,616
200	9,203
300	13,983
400	19,123
500	24,135



5. Conclusão

Neste trabalho foi implementado um algoritmo para encontrar o casamento aproximado de cadeias de caracteres, utilizando o algoritmo de distância Levenshtein. O trabalho possibilitou aprofundar os estudos em programação dinâmica e processamento de cadeias de caracteres. Uma das dificuldades encontradas durante o desenvolvimento desse algoritmo, foi a leitura das strings dos arquivos caractere por caractere, o qual formava as strings dinamicamente. Outra dificuldade foi a impressão no arquivo de saída, que necessitou de uma lista para gravar as respostas de um aluno, calcular a distância total e imprimir corretamente no formato de saida.

Infelizmente não foi possível completar o trabalho com êxito. Seria interessante ter tido mais tempo para codificar a parte de programação paralela. A grande complexidade para aprender as funções de threads da biblioteca POSIX aliada ao final de semestre com muitas provas e trabalhos, impossibilitou a implementação da mesma.

6. Referências

[1] Cormen, T. (2001). Introduction to algorithms. Edição 2001. MIT press.

[2] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 2ª Edição, Editora Thomson, 2004.

[3] Material preparados pelo Prof. Marcos e Meira e disponibilizados para os alunos da disciplina de Algoritmos e Estruturas de Dados III.

[4] Algoritmo Distância de Edição - Distância de Levenshtein – UFSJ - https://www.youtube.com/watch?v=JYzx0z7TmTM

[5] Parallel Patterns: WaveFront Pattern - Levenshtein distance - http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=wavefrontPattern

[6] Edwards, William - Edit Distance on massive strings - http://williamedwardscoder.tumblr.com/post/24208897480/edit-distance-on-massive-strings