Universidade Federal de Minas Gerais
DCC- Departamento de Ciência da Computação
Engenharia de Sistemas

PEDRO CAROLINO MALAQUIAS - 2016070115

TRABALHO PRÁTICO 2
ALGORITMOS E ESTRUTUDAS DE DADOS 3

BELO HORIZONTE

2018

1 INTRODUÇÃO

O objetivo desse trabalho era realizar o desenvolvimento de um algoritmo para auxiliar os usuários na hora da digitação, o que se mostra ser de grande ajuda para pessoas que digitam rápido.

A ideia principal é implementar um sistema para recomendar palavras semelhantes a qual o usuário digitou, dado um limite de mudanças. Para isso, foi dado um conjunto de palavras D que serve como dicionário, foi dado também uma palavra Q para realizar a consulta, e um inteiro N para informar o máximo de operações para transformar uma palavra fonte na palavra destino.

Somente 3 operações são validas e tem custo unitário, são elas:

- Inserção de Caractere: um caractere qualquer pode ser inserido na palavra de consulta.
- Remoção de Caractere: um caractere qualquer pode ser removido na palavra de consulta.
- Substituição de Caractere: um caractere qualquer da palavra de consulta pode ser alterado por outro caractere.

Tendo essas informações, o algoritmo deve retornar todas as palavras do dicionário que podem ser reconstruídas a partir da palavra de consulta utilizando o no máximo o número de operações dado no problema. Esse retorno deve ser feito ordenando as palavras, e utilizando como critério o numero de operações gasta para transformar uma string na outra. Caso de empate, deve-se ordenar por ordem léxica.

Problemas como estes são facilmente perceptíveis no nosso dia a dia, tendo em vista que todos os smartphones tem essa funcionalidade na hora da digitação, com os verificadores ortográficos, seja no aplicativo do mensager do facebook, ou no aplicativo do whatsapp.

Desta forma, cheguei à conclusão que a melhor forma de resolução desse problema, tendo em vista os meus conhecimentos prévios, seria aplicar o algoritmo de “Levenshtein Distance”, esse algoritmo calcula o mínimo de operações de inserção, remoção e substituição para transformar uma string padrão, em outra, utilizando o paradigma de Programação Dinamica.

2 MODELAGEM DO PROBLEMA

A visualização gráfica do problema pode ser representada como um autômato para casamento aproximado, onde a aresta vertical insere um caractere, a aresta diagonal solida substitui um caractere, e a aresta diagonal tracejada retira um caractere.

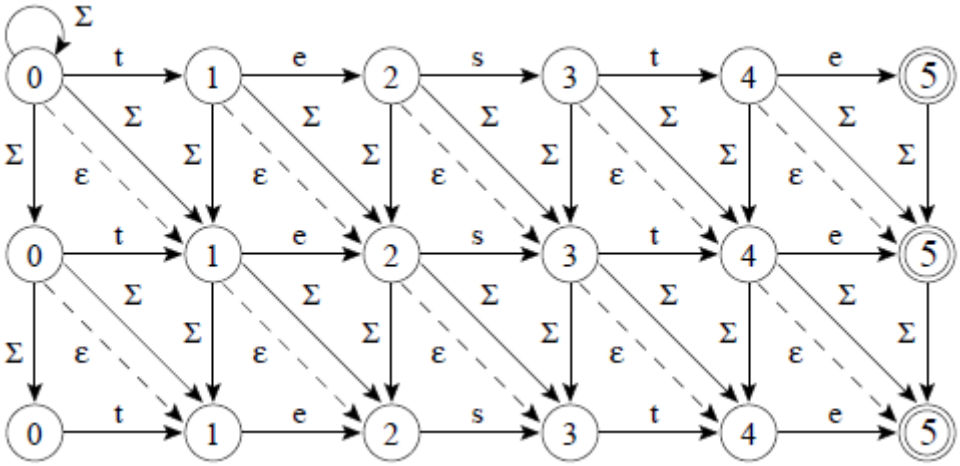


Figura 2-1: Exemplo de Autômato para Casamento Aproximado

Para a solução do problema, utilizamos como ideia base o algoritmo conhecido como “Levenshtein Distance”, onde esse algoritmo calcula o mínimo de operações validas para transformar uma string em outra.

```
while(exit != entrance_dic)
{
    //Levenshtein Distance
    scanf("%s",vet_word);
    size_word = strlen(vet_word)+1;

    //STEP 2
    matrix[0][0] = 0;
    for(i = 1; i < size_pattern; i++)
    {
        matrix[i][0] = i;
    }

    for(j = 1; j < size_word; j++)
    {
        matrix[0][j] = j;
    }

    //STEP 3 e STEP 4
    for(i = 1; i < size_pattern; i++)
    {
        for(j = 1; j <  size_word; j++)
        {

            //STEP 5
            if(vet_pattern[i-1] == vet_word[j-1])
            {
                comparation = 0;
            }
            else
            {
                comparation = 1;
            }

            //STEP 6
            above = matrix[i][j-1];
            diag = matrix[i-1][j-1];
            left = matrix[i-1][j];

            minimo = calc_minimo(above+1,diag+comparation,left+1);
            matrix[i][j] = minimo;
        }
    }

    //STEP 7
    n = size_pattern;
    m = size_word;
    result = matrix[n-1][m-1];

    if(result <= change)
    {
        count_change++;
        vetOrdena[k].qnt_change = result;
```

Figura 2-2: Algoritmo Levenshtein Distance

Esse algoritmo tem 7 etapas que são elas:

- 1ª Etapa: Construir uma Matrix (nxm), onde n é a quantidade de caracteres da palavra de consulta, e m é a quantidade de caracteres da palavra do dicionário.
- 2ª Etapa: Inicializar a primeira linha e a primeira coluna da matriz, de 0 a n ,e 0 a m respectivamente.
- 3ª Etapa e 4ª Etapa: Caminhar tanto na linha quanto na coluna

- 5ª Etapa: Se as letras comparadas na linha e na coluna forem iguais, comparation é 0, se forem diferentes comparation e 1.
- 6ª Etapa: Salvar os valores, acima, na diagonal e a esquerda da posição verificada, comparar qual deles e o menor, e colocar o menor na posição verificada.
- 7ª Etapa: Te mostra o resultado, que estará na posição nxm da matriz, após o termino do algoritmo.

Segue abaixo a ilustração da matriz logo após a execução do algoritmo.

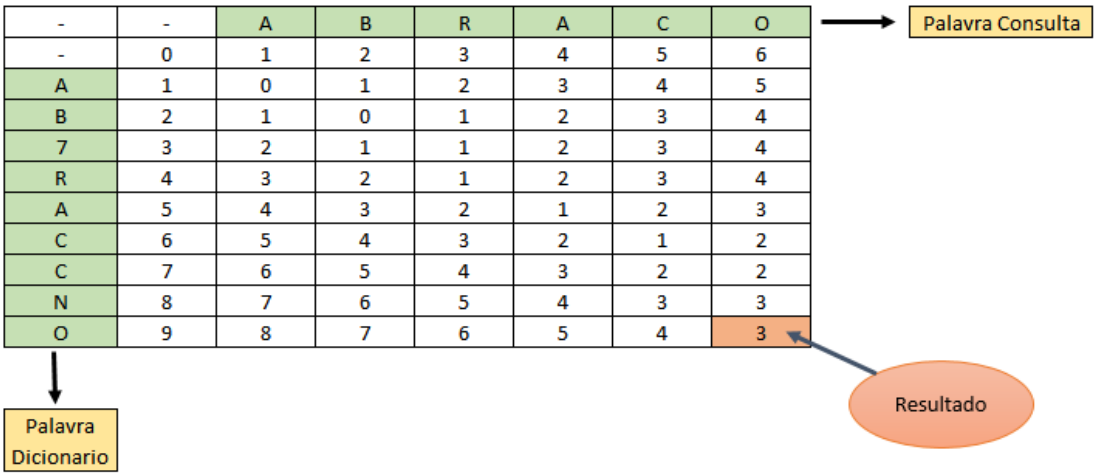


Figura 2-3: Ilustração da Matriz

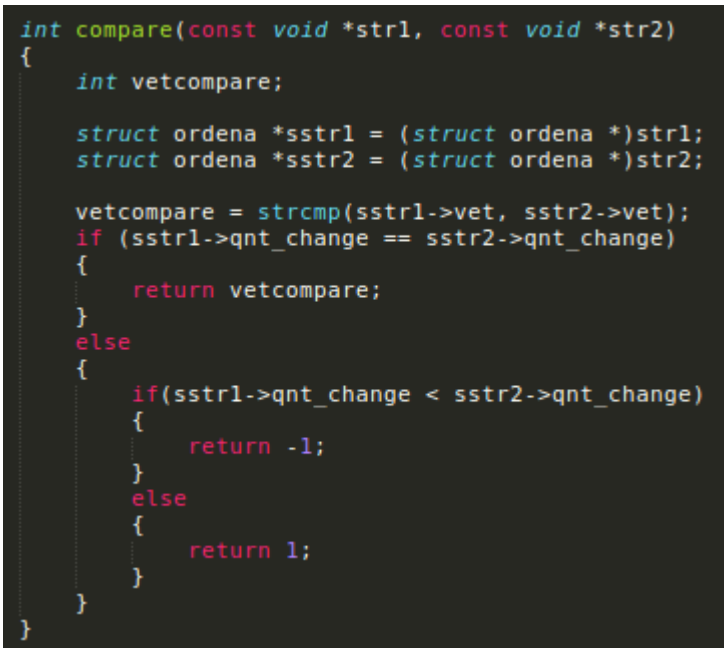
Para a resolução do problema, rodamos esse algoritmo o número de palavras do dicionário vezes. E logo em seguida ordenamos os resultados validos, por ordem crescente das palavras que sofreram menos alterações para as que sofreram mais alterações, e dando empate na quantidade de alterações, o desempate ocorre por ordem léxica.

```
typedef struct ordena{

    int qnt_change;
    char vet[6000];

}ordena;
```

Figura 2-4: Struct para Ordenar

Para isso foi criado uma Struct para o armazenamento das palavras validas e da sua quantidade de mudança. E em seguida fizemos o ordenamento das structs utilizando a função qsort().

Segue abaixo a função de comparação das structs:



```c
int compare(const void *str1, const void *str2)
{
    int vetcompare;

    struct ordena *sstr1 = (struct ordena *)str1;
    struct ordena *sstr2 = (struct ordena *)str2;

    vetcompare = strcmp(sstr1->vet, sstr2->vet);
    if (sstr1->qnt_change == sstr2->qnt_change)
    {
        return vetcompare;
    }
    else
    {
        if(sstr1->qnt_change < sstr2->qnt_change)
        {
            return -1;
        }
        else
        {
            return 1;
        }
    }
}
```

Figura 2-5: Função para Comparação de Struct

Essa função é utilizada como parâmetro da função qsort().

Logo após a finalização da ordenação das structs é impresso na tela as palavras validas.

3 ANÁLISE TEORICA DO CUSTO ASSINTÓTICO

Nesta seção analisaremos os custos teóricos de tempo e espaço do algoritmo, tanto para a matriz de adjacência, quanto para a busca em profundidade e pilha.

3.1 ANÁLISE TEORICA DO CUSTO ASSINTOTICO DE TEMPO

- Levenshtein Distance: O algoritmo possui uma complexidade de tempo O(nm), quando é implementado utilizando programação dinâmica. Pode ser facilmente verificado e o tempo de percorrer e preencher a matriz. Como isso é feito numero de palavras vezes D , o custo ficaria sendo O(Dnm).
- Ordenação: A ordenação é feita pela função qsort() , que representa o algoritmo de ordenação Quicksort, então seu custo assintótico depende do caso em que ele cai, se é melhor caso O(nlogn), se for o pior caso $O(n^2)$.

Portanto o custo assintótico de tempo do algoritmo, varia em relação ao caso da ordenação, podendo ser: $O(Dnm + n^2)$ para o pior caso, ou O(Dnm + nlogn) para o melhor caso.

3.2 ANÁLISE TEORICA DO CUSTO ASSINTOTICO DO ESPACO

- Levenshtein Distance: O algoritmo possui uma complexidade de espaço também O(nm), que é o tamanho da matriz, utilizada na execução do algoritmo.
- Ordenação: A ordenação na função qsort(), a sua complexidade depende do que está se ordenando, no nosso caso, foi ordenado n structs, sendo n a quantidade de palavras validas. Então o seu custo de espaço é a própria struct.

Portanto o custo assintótico de espaço do algoritmo, é O(nm + tamanho da struct).

4 ANÁLISE DOS EXPERIMENTOS

A análise experimental da implementação pode ser visualizada pela Tabela 1. Para realizar o experimento foi gerado um arquivo MAKEFILE onde, ele já faz a compilação com o atalho, digitado no terminal, “make tp2”, faz a limpeza do compilado com “make clean”, e também executa dos os testes com o atalho, “make teste”.

Para medir o tempo de execução do código, foi utilizada a biblioteca "<time.h>" para contar o número de clocks necessários para executar o código. Cada instancia foi gerada 4 vezes e foi retirada a média para obtermos um tempo com maior confiabilidade e precisão. Tendo em vista que os testes foram realizados em um computador com processador Core i5-7200 8GB de RAM e 2.71GHz.

Tabela de Resultado do Algoritmo		
Testes	Quantidade de Palavras	Tempo(s)
Teste 1	15	0,27
Teste 2	60	0,24
Teste 3	90	0,27
Teste 4	2000	0,67
Teste 5	2000	15,43
Teste 6	4000	20,73
Teste 7	6000	27,3
Teste 8	8000	151,2
Teste 9	10000	267,45
Teste 10	10000	211,23

Tabeça 1-1: Resultados em relação ao tempo e entrada

A Tabela 1, fornece o tempo médio do algoritmo em relação a quantidade de palavras que os testes forneciam, percebe que tem um aumento a medida que a quantidade de palavra vai aumentando, o que mais chama atenção são os resultados dos testes 9 e 10, pois ambos tem a mesma quantidade de entrada, mas o algoritmo de ordenação faz com que um seja mais rápido que o outro.

Isso pode ser verificado também nos testes 4 e 5, em que o teste 4 entrou no melhor caso O(nlogn) da ordenação, e o teste 5 entrou no pior caso $O(n^2)$.

5 CONCLUSÃO

Nesse trabalho, foi realizado a implementação do algoritmo “Levenshtein Distance” utilizando o paradigma de programação dinâmica, onde foi resolvido o problema, transformando o problema em subproblemas e a medida que o subproblema eram resolvidos o problema inicial também era.

Os resultados foram satisfatórios, tendo em vista que, todos os testes realizados obtivemos o resultado esperado.

O que se pode notar visualizando os tempos obtidos, foi que dependendo da quantidade de palavras fornecidas, o algoritmo demorava a mais para encontrar o resultado esperado, tendo em vista que, teríamos que criar mais matrizes e ordenar uma maior quantidade de structs o que é totalmente compreensível tendo em vista a analise assintótica do algoritmo.