Trabalho Prático 2

Sugestões Doodle

Algoritmos e Estrutura de Dados III

Álvaro Augusto Alves de Carvalho

2015430231

1 Introdução

O objetivo deste trabalho prático é a resolução do problema proposto entitulado "Sugestões Doodle". O problema consiste em determinar, a partir de uma string de entrada e um dicionário de strings fornecido, todas as strings que podem ser alcançadas através da string de origem, em ordem do número de edições necessárias e utilizando a ordem lexicográfica das palavras para desempate. Este é um problema de programação dinâmica muito conhecido, e é um problema encontrado por softwares de aproximação de caracteres em softwares de busca ou de correção ortográfica (Ex.: Google, teclado de um smartphone). Desta forma, deve-se implementar um algoritmo para encontrar a Mínima Distância de Edição, dada a palavra de origem e o dicionário, assim como o número máximo de operações permitidas, todas com custo O(1).

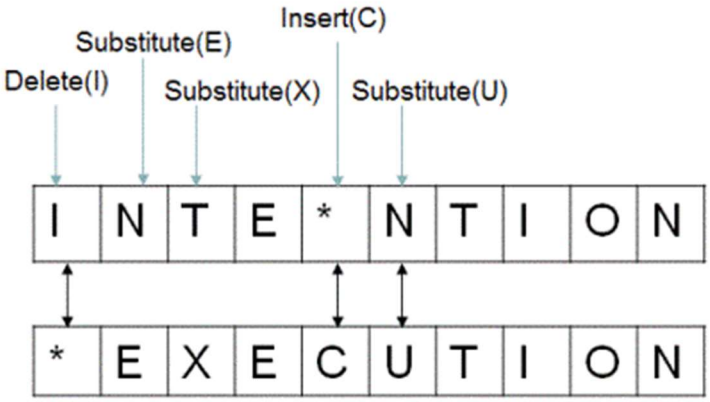

Figura 1: Exemplo do algoritmo de mínima distância de edição. Considere 2 palavras e 3 operações de edição (inserção, remoção e substituição). Qual o mínimo número de edições que devem ser feitas para chegar na palavra 2 a partir da palavra 1? No exemplo acima, partindo de INTE NTION, são feitas 5 operações para chegar na palavra EXECUTION.

Com isto em mente, e utilizando conceitos de programação dinâmica aprendidos na disciplina de Algoritmos e Estrutura de Dados III, chegou-se na conclusão de que a melhor maneira de abordar o problema seria utilizando o algoritmo de mínima distância de Levenshtein, calculando esta distância mínima para cada palavra do dicionário em relação à palavra de consulta e ordenando esta lista de palavras. Utilizando este método de Levenshtein junto com algumas simples manipulações para tratamento da entrada e da saída, conseguimos obter a resposta do problema.

2 Modelagem do Problema e Implementação

		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Figura 2: Exemplo de uma matriz de Levenshtein. Esta matriz é construída a partir do algoritmo de distância mínima de Levenshtein. A construção da Matriz não é a única maneira de implementar o algoritmo, mas é relativamente eficiente e de simples implementação.

A distância de Levenshtein é uma medida de similaridade entre duas strings. Esta distância é justamente o número de operações individuais que podem ser feitas na string, inserção, deleção e substituição. Este algoritmo foi desenvolvido por Vladimir Levenshtein, um cientista da informação russo, em 1965. O algoritmo é bem conhecido e tem aplicações em várias áreas de grande importância para a evolução da informática e medicina, tais como: Reconhecimento da fala; Análise de DNA (comparação de cadeias de DNA); Correção ortográfica (importante para o funcionamento eficiente de ferramentas de busca como o Google).

A implementação de Levenshtein utilizada neste trabalho consiste em criar uma matriz de distâncias d[n+1][m+1], sendo ‘n’ o número de caracteres da string fonte, e ‘m’ o número de caracteres da string destino. Esta matriz contém o número de edições necessárias em cada célula para obter uma sub-string a partir de outra. Por exemplo, na célula da matriz d[i][j], o valor contido nesta célula representa o número de edições necessárias para obter a string s1[1,2,3...,j] a partir da string s2[1,2,3,...i]. Observando a figura 2 acima, e a célula d(0,3), nós temos as sub-

strings s1 = “mei” e s2 = “l”, e temos o valor 3. Ou seja, é necessário 3 operações para obter “mei” a partir de “l” e vice-versa.

Seguindo esta lógica, a matriz pode ser inicializada conforme a figura 3, e a partir disso, consegue-se obter todos os outros elementos da matriz de maneira iterativa, ou recursiva.

		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1											
e	2											
v	3											
e	4											
n	5											
s	6											
h	7											
t	8											
e	9											
i	10											
n	11											

Figura 3: Inicialização da matriz de Levenshtein.

Com esta matriz inicializada, o resto da matriz é percorrida, seguindo as instruções:

- Examina cada caractere de s1 (s1[1]...s1[n]) e cada caractere de s2 (s2[1]...s2[m])
- Verifica a célula d[i][j]:
    - Se s1[i] != s2[j], a célula d[i][j] receberá o mínimo de:
        - A célula imediatamente acima + 1: d[i-1][j] + 1
        - A célula imediatamente à esquerda + 1: d[i][j-1] + 1
        - A célula na diagonal à esquerda e acima + 1: d[i-1][j-1] + 1
    - Se s1[i] == s2[j], a célula d[i][j] receberá o valor da célula na diagonal à esquerda e acima: d[i-1][j-1].
    - Exemplo: Na figura 3, olhando para a célula d[1][1], s1[i] != s2[1], então considera-se os 3 valores cercando a célula à esquerda e acima, e pega o mínimo deles, e finalmente soma 1. O valor de d[1][1] será 2.
    - Considerando d[2][2], temos que s1[2] == s2[2]. Desta forma, temos que: d[2][2] = d[1][1] = 2.

Quando todos os passos forem concluídos e a matriz for completamente preenchida, a resposta final para a menor distância de edição das duas strings será o valor em d[n][m].

Em posse desta ferramenta, o trabalho está quase concluído. Basta fazer o tratamento da entrada e depois da saída. Para isso, optou-se por armazenar as palavras do dicionário em uma estrutura de dados que contém a palavra em si e o valor da distância de edição em relação à palavra de consulta.

```
struct dictionary {
    char word[MAX_SIZE];
    int distance;
};
```

Declaração da estrutura “dictionary” que representa uma palavra do dicionário lido pela entrada padrão.

No momento de receber como entrada as palavras que compõem o dicionário, uma lista de estruturas “dictionary” é feita para acomodar todas as strings. No entanto, para auxiliar e aumentar a eficiência do algoritmo, visto que existe um tamanho máximo N de edições permitidas, o algoritmo faz também um filtro inicial, desconsiderando todas as strings que possuem tamanho maior que a string de consulta + N e tamanho menor que a string de consulta – N. A lógica por trás disso é que, se entre uma string e outra há um gap de mais de N caracteres, não é possível chegar na string do dicionário a partir da string de consulta com menos de N operações, e portanto não é relevante para o trabalho.

```
for each Entry {
    if (isValidSizeEntry) {
        dictionaryList[i].word = Entry;
    }
}
```

Pseudo-código da verificação de quem entrará na lista de palavras a serem processadas. Para cada palavra lida, somente será inserida na lista se o tamanho dessa palavra estiver entre (tamanho da palavra original - N) e (tamanho da palavra original + N)

Com estas considerações, a lista de palavras é criada com uma distância inicial -1. Após isso, utiliza-se o algoritmo de Levenshtein entre a palavra de consulta e cada palavra da lista de palavras, atualizando a distância de edição em cada palavra. É importante notar aqui que após cada execução do algoritmo de Levenshtein, a matriz é destruída e desalocada.

```
void getMinimumEditDistance(char *q, struct dictionary * dict, int n) {
    int i;
    for (i=0;i<n;i++) {
        dict[i].distance = levenshteinDistIterativo(q, dict[i].word);
    }
}
```

Código para pegar a distância de Levenshtein para cada palavra da lista em relação à palavra de consulta “q”

Após esse processamento, basta ordenar a lista de acordo com a distância em ordem crescente, e para critério de desempate, utilizar a ordem lexicográfica das strings. Para isso foi utilizada a função qsort da biblioteca stdlib do C.

Com isso, o programa por fim imprime a lista ordenada na saída padrão, destrói a lista de estruturas criada e encerra a execução. Problema resolvido.

3 Análise de Complexidade (teórica)

3.1 Análise de Complexidade de Tempo

Podemos realizar a análise de complexidade temporal baseando nas complexidades de cada algoritmo apresentado.

- O algoritmo utilizado para ler as entradas e filtrar as que não seriam válidas percorre por toda a entrada padrão e tem complexidade fixa em O(N+3), onde N é o número de palavras do dicionário, e é necessário mais 3 passadas para ler o número de entradas do dicionário, a palavra de consulta e o número máximo de operações aceitas.
- O algoritmo de Levenshtein cria e trabalha em cima de uma matriz de ordem NxM, onde N é o tamanho da string s1 (palavra de consulta) e M é o tamanho da string s2 (palavra do dicionário). Este algoritmo é executado um número de vezes que representa D – k, onde D é o número de entradas do dicionário e k é o número de palavras filtradas, desta forma, temos que o algoritmo têm uma complexidade:
$$\sum_{1}^{D-k} O(N * M_i)$$
- Os outros algoritmos implementados referem-se à destruição da matriz em si (O(N) onde N = número de linhas) e impressão na tela (O(D) onde D = número de palavras no dicionário). A implementação da filtragem das palavras inadequadas ocorre no mesmo loop de leitura e portanto acontece em O(1).

Por fim, temos que a complexidade do programa em si, considerando o algoritmo de maior complexidade, é:

$$\sum_{1}^{D-k} O(N * M_i)$$

.

3.2 Análise de Complexidade Espacial

Para a análise de complexidade espacial, vamos abordar a estrutura de dados do dicionário em si, assim como os algoritmos utilizados.

- A estrutura utilizada é uma lista de estruturas, onde cada estrutura é composta por uma string e um int, com complexidade espacial equivalente a:

$$\sum_{1}^{D} sizeof(char) * S_i + sizeof(int)$$

Onde S_i é cada palavra do dicionário lido, e D é o número de entradas no dicionário. O algoritmo aloca uma lista de tamanho D antes de realizar o filtro.

- O algoritmo de Levenshtein realiza operações em uma matriz, tal que a complexidade espacial do algoritmo é equivalente à maior matriz que ele precisa construir para a consulta (alocação dinâmica, a matriz é destruída e reconstruída para cada palavra do dicionário). A maior matriz será construída para a string do dicionário de maior tamanho. Considerando o tamanho desta string específica como sendo S_M, temos que a complexidade é: $O(N * S_M)$, onde N é o tamanho da string de consulta.

Por fim, temos uma complexidade espacial do programa, considerando as estruturas e algoritmos abordados: $O_{max}\big(O(N * (S + sizeof(int)), O(N * S_M)\big) = O(N * S_M)$

4 Análise dos Experimentos

Para a análise experimental do trabalho, foi utilizado o comando *time* (para análise de complexidade temporal), e o *valgrind* (para análise do consumo de memória do programa). Foram utilizados os testes fornecidos pelo moodle da disciplina na análise. Todos os testes foram realizados em uma máquina virtual Ubuntu através do Oracle VM VirtualBox, com 4gb de RAM alocados. A máquina física dispõe de um processador i7-4700MQ @2.40GHz, com 16gb de RAM.

4.1 Análise Temporal

Para a análise temporal, cada teste foi executado 10 vezes utilizando o comando *time*. Ex.: time ./tp2 < teste_i.txt > output_i.txt

Os valores obtidos seguem na tabela 1.

Teste / Execucoes	1	2	3	4	5	6	7	8	9	10
1	0.001	0.002	0.002	0.002	0.002	0.001	0.001	0.001	0.001	0.001
2	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
3	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
4	0.014	0.008	0.008	0.008	0.008	0.01	0.008	0.007	0.009	0.008
5	0.032	0.026	0.024	0.021	0.027	0.026	0.024	0.026	0.022	0.026
6	0.072	0.07	0.067	0.06	0.062	0.064	0.063	0.062	0.061	0.062
7	0.073	0.067	0.068	0.065	0.067	0.071	0.078	0.073	0.071	0.076
8	0.227	0.191	0.186	0.196	0.19	0.191	0.191	0.189	0.19	0.189
9	0.341	0.298	0.307	0.312	0.305	0.298	0.303	0.303	0.317	0.3
10	0.423	0.397	0.404	0.4	0.417	0.396	0.388	0.392	0.391	0.405

Tabela 1: Valores de tempo de execução obtido por teste, em um total de 10 execuções. Valores expressos em segundos.

A partir da média dos valores obtidos (tabela 2), e do número de entradas no dicionário (D)em cada teste, o gráfico da figura 3 foi construído.

Teste	Tempo Médio	D
1	0.0014	15
2	0.002	60
3	0.002	90
4	0.0088	2000
5	0.0254	2000
6	0.0643	4000
7	0.0709	6000
8	0.194	8000
9	0.3084	10000
10	0.4013	10000

Tabela 2: Tempo médio de execução x número de vértices, por teste

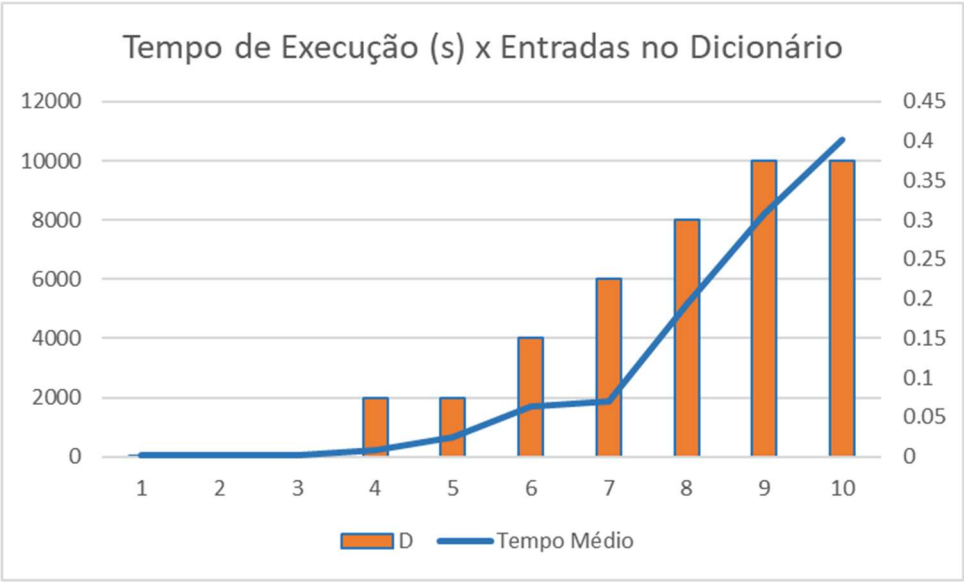

Figura 4: Gráfico de tempo de execução (em segundos) x Número de entradas no dicionário, por teste

Podemos ver, principalmente nos últimos testes, que a complexidade do programa depende linearmente do número de entradas no dicionário na entrada, mas o tamanho das palavras do dicionário possuem um peso maior. Vemos que o teste 9 possui um tempo de execução significativamente menor que o teste 10, com o mesmo número de entradas.

4.2 Análise Espacial

Para a análise espacial foi utilizado o Valgrind para cada caso de teste apresentado, e foi criada uma tabela com os valores de memória utilizados em cada execução em relação ao número de vértices. Posteriormente, foi criada uma representação gráfica dos dados.

Testes	kBytes	D
1	105.98	15
2	409.91	60
3	562.67	90
4	13200.15	2000
5	13545.17	2000
6	32197.48	4000
7	40620.28	6000
8	73232.8	8000
9	129701.8	10000
10	160060.8	10000

Tabela 3: Espaço de memória utilizado (kBytes) x Número de entradas no dicionário, por teste

Figura 4: Gráfico de consumo de memória (kBytes) x Número de entradas no dicionário, por teste

Através dos dados apresentados, podemos verificar novamente, e aqui talvez até com mais facilidade, que a complexidade espacial do programa depende não só do número de entradas do dicionário, mas também do tamanho das palavras no teste.

5 Conclusão

O trabalho proposto teve como objetivo familiarizar o aluno com conceitos de programação dinâmica vistos na disciplina de Algoritmos e Estrutura de Dados III. Para a resolução do problema, foi implementada uma solução simples que depende principalmente do algoritmo de distância mínima de Levenshtein, com algumas simples implementações para processamento dos dados, de forma a se adequar ao problema apresentado.

Esta solução aqui apresentada, apesar de não ser a mais eficiente, acredito que teve uma boa relação de “complexidade de implementação x eficiência do código”. Acredito que este foi um problema bastante direto, e a abordagem utilizada foi adequada à realidade do aluno, que ainda possui algumas limitações para implementação de algoritmos mais sofisticados, principalmente considerando o tempo disponível para realização do trabalho junto com as dificuldades encontradas em outras disciplinas durante o curso. As análises experimentais e teóricas, a princípio, também parecem condizentes com o problema proposto e a solução implementada.