A Maior Subsequência em Comum e suas Aplicações na Modelagem de uma Urna Eletrônica

Rodrigo T. M. O. Lima – Grupo E

Centro Universitário IESB
Brasília – DF – Brasil

rodrigo.t.lima@gmail.com

Resumo. *Este artigo descreve uma modelagem de urna eletrônica e propõe o uso de um algoritmo baseado no problema da maior subsequência em comum (LCS) para aprimorar os resultados de consultas aos dados em arquivo. O modelo teórico providencia todas funções típicas de uma urna eletrônica e também contém um módulo específico para uso administrativo. Este módulo possui funções de busca de informações sobre votos, de busca de informações sobre candidatos e de gerenciamento dos dados. O algoritmo de LCS é baseado nos conceitos de Hirschberg (1975) e soluciona o problema em tempo quadrático e espaço linear.*

1. Introdução

O processo eleitoral é de suma importância para um país, definindo suas lideranças nos anos seguintes e ditando a estrutura administrativa em vários níveis do Estado. A urna eletrônica, método de votação que vem sendo usado no Brasil desde 1994 (TSE, 2014), introduziu aspectos tecnológicos ao cenário eleitoral que permitissem uma apuração rápida e eficiente.

A urna eletrônica habilita, como funcionalidade básica, a realização do voto de um eleitor. O sistema, usado por milhões de brasileiros nos dias de votação (TSE, 2013), deve ser performático, confiável e estável. Ao lidar com grandes massas de dados, o sistema deve manter sua integridade e o sigilo das informações trabalhadas.

O trabalho visa, por meio de uma modelagem teórica, simular as principais funcionalidades da urna, além de integrar funções de busca, que são disponibilizadas aos administradores responsáveis pela realização de uma votação. Os procedimentos de busca são auxiliados por funções voltadas à correção de *strings*.

Como foco do projeto, é implementado um algoritmo baseado no problema da maior subsequência em comum, também conhecido como *Longest Common Subsequence* (LCS). Este algoritmo é responsável por comparar os termos de busca do usuário aos dados disponíveis no sistema, gerando um valor que retrata o quão próximo uma *string* de caracteres está à outra (WAGNER e FISCHER, 1974).

Caso os termos de busca não são completamente equivalentes aos dados disponíveis, os resultados mais semelhantes à busca original são divulgados como sugestões de busca, ranqueados de acordo com o valor do LCS. Quanto mais próximo, maior é este valor, até que se chegue em um máximo igual ao comprimento das *strings*. Isto permite um certo grau de erro ou ambiguidade nas consultas.

2. Referencial Teórico

2.1. The String-to-String Correction Problem

Wagner e Fischer (1974) detalham um algoritmo que calcula o número de operações necessárias para transformar uma cadeia de caracteres em outra. São considerados três tipos de operações, conforme definidos por Morgan (1970 apud WAGNER e FISCHER, 1974): (1) alterar um caractere para outro; (2) deletar um caractere; (3) inserir um caractere.

A proposta do algoritmo é de computar este resultado com tempo de execução e espaço $O(m \times n)$, onde m e n representam o número de caracteres nas duas cadeias consideradas. O emprego desta técnica pode ser usado para solucionar o problema do *Longest Common Subsequence* (WAGNER e FISCHER, 1974).

Considere duas cadeias finitas de caracteres A com comprimento m ($a_1a_2...a_m$) e B com comprimento n ($b_1b_2...b_n$). Para os autores, uma operação pode ser expressa como um par ordenado de caracteres denotado por (a, b). Se $a \neq \varepsilon$ e $b \neq \varepsilon$, então é uma operação que altera o caractere a para o caractere b. Caso $a \neq \varepsilon$ e $b = \varepsilon$, é uma operação que deleta o caractere a. Se $a = \varepsilon$ e $b \neq \varepsilon$, é uma operação que insere o caractere b.

O algoritmo de Wagner e Fischer (1974) constrói uma tabela D com m+1 linhas e n+1 colunas (sem considerar linha e coluna para rótulos). Considere $1 \leq j \leq m$ e $1 \leq i \leq n$ os índices dos caracteres de A e B, respectivamente. Considere também o custo individual de cada operação (alterar, deletar, inserir) como sendo igual à 1. Então, o método para computar o valor de cada célula da tabela D é:

(1) Se $i = j = 0$, então D[i][j] = 0
(2) Senão, se $j = 0$, então D[i][0] = D[i-1][0] + 1
(3) Senão, se i = 0, então D[0][j] = D[0][j-1] + 1
(4) Senão, se A[i] ≠ B[j] então, D[i][j] = min(D[i-1][j-1], D[i-1][j], D[i][j-1]) + 1
(5) Senão, se A[i] = B[j] então, D[i][j] = min(D[i-1][j-1], D[i-1][j], D[i][j-1])

O algoritmo computa então a quantidade mínima de etapas necessárias para esta transformação, com o resultado final expresso na célula da última linha com a última coluna (D[m+1][n+1]). Como exemplo, seja A = 'bd' e B = 'abcd':

Tabela 1. Menor custo para a transformação de cada subsequência

	ε (vazio)	**'a'**	**'b'**	**'c'**	**'d'**
ε (vazio)	**0**	**1**	**2**	**3**	**4**
'b'	**1**	**1**	**1**	**2**	**3**
'd'	**2**	**2**	**2**	**2**	**2**

No exemplo acima, a tabela diz que o menor custo para transformar a cadeia 'bd' na cadeia 'abcd' é igual à 2. Quanto menor for o número de operações necessárias para transformar a cadeia A na cadeia B, maior a semelhança entre elas. Logo, uma adaptação deste algoritmo gera a solução para o problema do *Longest Common Subsequence* usando a diferença entre o comprimento da cadeia mais longa e o menor custo de operações necessárias para transformar a cadeia A na cadeia B (WAGNER e FISCHER, 1974):

$$\text{LCS} = max(A.\text{comprimento}, B.\text{comprimento}) - D[m+1][n+1]$$

2.2. A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg (1975) propõe uma solução para o problema da maior subsequência em comum (LCS) por meio de um algoritmo que computa o resultado em tempo $O(m \times n)$ e espaço $O(n)$. Outros algoritmos desenvolvidos anteriormente resolvem o problema do LCS em tempo quadrático e espaço também quadrático.

Uma cadeia de caracteres A é considerada uma subsequência em comum de outra cadeia de caracteres B se todo caractere de A está em B, na mesma sequência. Não é necessário, portanto, que os caracteres de A em B sejam contíguos. Seja $A = a_1a_2...a_n$ e $B = b_1b_2...b_m$. A é subsequência de B se, e somente se: (1) para todo caractere a_i em A, a_i também pertence a B; (2) se a_i for igual a b_u e a_j for igual a b_v, necessariamente se $i < j$, então $u < v$ (HIRSCHBERG, 1975).

A solução de Hirschberg mantém as mesmas ideais centrais propostas por outros autores como Wagner e Fischer (1974) na concepção do algoritmo, que tem seu fundamento na programação dinâmica, onde uma matriz solução é usada para armazenar valores de subproblemas (*divisão e conquista*) e a computação é realizada fileira por fileira. No entanto, Hirschberg (1974) introduz uma variação que possibilita o reaproveitamento de espaço em memória, diminuindo consideravelmente seu consumo.

Diferentemente de propostas anteriores, o algoritmo de Hirschberg utiliza apenas dois vetores X e Y de tamanho igual ao comprimento da segunda *string* + *1* para armazenar as computações, visto que os valores de uma fileira dependem dos valores da outra. Logo, os valores da fileira que está sendo trabalhada são definidos com base nos valores já computados da fileira anterior. Esta característica permite uma alternância no uso das fileiras, diminuindo o espaço usado pelo algoritmo para grau linear (HIRSCHBERG, 1974).

```
int LCS_DP(char* A, char* B, int m, int n)
{
    int X[n+1], Y[n+1];
    int i, j;

    for (i = m; i >= 0; i--)
    {
        for (j = n; j >= 0; j--)
        {
            if (A[i] == '\0' || B[j] == '\0')
            {
                X[j] = 0;
            }
            else if (A[i] == B[j])
            {
                X[j] = 1 + Y[j+1];
            }
            else
            {
                X[j] = max(Y[j], X[j+1]);
            }
        }
        Y = X;
    }
    return X[0];
}
```

3. Modelagem da urna eletrônica

Por oferecer alta performance e maior facilidade na manipulação de ponteiros em memória, a ferramenta de modelagem é a linguagem de programação C. Esta linguagem, sendo compilada e possuindo pouco *overhead*, possibilita implementações eficientes, critério necessário ao se trabalhar com grandes volumes de dados (CELES, CERQUEIRA, RANGEL, 2004).

O modelo da urna eletrônica é composto por três partes principais: Módulo Administrador, Módulo Eleitor e arquivos com os dados. Também, o projeto disponibiliza em um arquivo texto todas as instruções necessárias para seu uso correto. O Módulo Administrador, contendo funções que permitem a visualização dos dados, é acessível apenas pelas equipes técnicas responsáveis pela votação.

3.1. Acesso ao Módulo Administrador

O acesso ao Módulo Administrador é condicionado à digitação das credenciais corretas. Para tanto, o programa recebe a tentativa de senha do usuário e a compara com a senha que está armazenada de forma criptografada em um arquivo separado. Este arquivo armazena em uma única linha o seguinte registro:

```
struct chave{
      char senha[N];// Senha, criptografada, de acesso; tamanho N
      int exp;// Expoente de criptografia: d × e ≡ 1 mod ϕ(n)
      int mod;// Módulo de criptografia: n = p × q
};
```

Tal registro do tipo ‘chave’ é gerado por outro programa à parte usando o algoritmo de criptografia RSA, que faz uso de duas chaves, uma pública e outra privada. Este algoritmo se baseia no problema da fatoração, da teoria dos números, para fortificar a integridade e a confidencialidade das informações (RIVEST, SHAMIR, ADLEMAN, 1978).

O programa principal recebe o *input* do usuário como tentativa de acesso ao Módulo Administrador. Este *input* é criptografado usando o expoente e o módulo de criptografia localizados no registro. Finalmente, o programa compara o *input* criptografado com a senha criptografada armazenada no arquivo. Por motivos de segurança, em nenhum momento a senha correta é descriptografada.

3.2. Operações do Módulo Administrador

No Módulo Administrador, o usuário tem acessso às seguintes funções administrativas: (1) Consultar candidato; (2) Listar candidatos; (3) Incluir candidato; (4) Remover candidato; (5) Consultar voto; (6) Listar votos; (7) Simular eleição.

As funções 1-4 possibilitam que a equipe técnica gerencie os dados sobre os candidatos. Estas funções trabalham com informações armazenadas em um arquivo texto denominado *candidatos.txt*. As informações sobre os candidatos seguem a seguinte estrutura:

```
struct candidato{
      char codigo_do_candidato[5]; // Código identificador
      char nome_candidato[30]; // Nome completo, 30 caracteres
      char codigo_do_partido[3]; // Código do partido
};
```

As funções 5 e 6 lidam com as informações dos votos já registrados. Tais dados estão armazenados em um arquivo texto chamado *votos.txt* e estão organizados como segue:

```
struct voto{
      char regiao; // Região do votante
      char cpf[15]; // Cadastro de Pessoa Física
      char numero_sequencial_de_voto[9]; // Define a ordem dos votos
      char codigo_do_municipio[8]; // Código fornecido pelo IBGE
      char codigo_candidato_federal[3]; // Voto federal
      char codigo_do_partido_federal[3]; // Partido federal
      char codigo_do_candidato_regional[5]; // Voto regional
      char codigo_do_partido_regional[3]; // Partido regional
};
```

A função 7 tem como propósito popular os arquivos *candidatos.txt* e *votos.txt* com milhões de dados artificiais, possibilitando testes rápidos do sistema e de todas suas operações. As duas funções de consulta, 1 e 5, fazem uso do algoritmo de LCS onde as *strings* comparadas pertencem aos membros: nome do candidato e CPF do eleitor, respectivamente.

3.3. *Longest Common Subsequence* (LCS)

Como ênfase do projeto, o algoritmo que providencia as sugestões de busca é baseado na solução do problema do LCS desenvolvida por Hirschberg (1975). De modo geral, este algoritmo compara duas sequências: “termos de busca” e “dado em arquivo”. Os “termos de busca” são uma cadeia de caracteres digitada pelo usuário quando efetuando uma busca no sistema. Estes “termos” podem ser caracteres alfabéticos (nome do candidato) ou caracteres numéricos (CPF). O “dado em arquivo” é o registro do arquivo sendo lido pelo programa. Cada registro está em linha separada no arquivo e o programa lê, linha a linha, até que se gere uma condição de parada.

O algoritmo compara as duas sequências e retorna um valor inteiro: o número de caracteres na maior subsequência em comum. A função armazena os três melhores resultados e ao final de sua execução demonstra ao usuário as opções que mais se aproximam de seus “termos de busca”. As duas condições de parada são: (1) O tamanho da maior subsequência, o comprimento dos “termos de busca” e o comprimento do “dado em arquivo” são todos iguais (*strings* comparadas são idênticas); (2) Final de arquivo (EOF).

Para análise de desempenho, são implementadas duas soluções do problema de LCS. O primeiro algoritmo, baseado na solução desenvolvida por Hirschberg (1975), faz uso da programação dinâmica (DP).

O segundo algoritmo, conhecida como *Naive*, gera todas as possíveis subsequências de forma recursiva e as analisa uma a uma. Esta solução tem complexidade de tempo $O(2^n)$, se tornando aceleradamente um método ineficiente na medida que a quantidade de dados aumenta.

```
int LCS_Naive(char* A, char* B, int m, int n)
{
    if(m == 0 || n == 0)
    {
        return 0;
    }
    else if(A[m-1] == B[n-1])
    {
        return (1 + LCS_Naive(A, B, m-1, n-1));
    }
    else
    {
        return maior(LCS_Naive(A, B, m, n-1), LCS_Naive(A, B, m-1, n));
    }
}
```

3.4. Módulo "Eleitor"

O acesso ao módulo "Eleitor" é condicionado à identificação do usuário. Para tanto, o programa pede as informações referentes aos campos contidos no registro *voto* (com exceção do número sequencial que é preenchido automaticamente) descrito anteriormente e solicita a confirmação do usuário.

Após esta confirmação, o Módulo Eleitor possibilita que o usuário vote em algum candidato cadastrado em arquivo. Para que o eleitor vote nulo, é preciso preencher o campo "voto" com o algarismo '0'. Caso o eleitor deseja votar em branco, deve preencher o campo "voto" com o algarismo '-1'. Este procedimento é feito primeiro para os candidatos regionais e depois para os candidatos federais.

Ao preencher todos os campos, o programa solicita uma confirmação (sim/não) do usuário. Apenas com as informações confirmadas que o programa as grava no arquivo. Com isso, o programa imprime na tela os detalhes do voto do usuário e encerra sua atividade.

4. Metodologia

O estudo considera dois métodos para calcular a maior subsequência em comum, *Naive* e programação dinâmica (DP). Para analisar o desempenho assintótico de cada algoritmo, usa-se crescentes quantidades de dados. Os dados escolhidos para realizar a pesquisa são do tipo *voto*, descrito anteriormente, localizados em um arquivo contendo 10 milhões de registros únicos. O conjunto de dados trabalhados representam votos de eleitores dos três maiores colégios eleitorais do país: São Paulo (SP), Minas Gerais (MG) e Rio de Janeiro (RJ).

A chave de busca usada é do membro *cpf*. Afim de assegurar cenários iguais na execução de cada um dos algoritmos, pesquisa-se as chaves contidas em registros específicos do arquivo. Por exemplo, a chave *'293.474.547-58'* pertence ao registro na linha 10.000 do corpo do arquivo. Pesquisa-se este exato registro e, ao encontrá-lo, as soluções calculam um valor da maior subsequência em comum igual ao comprimento das duas strings comparadas, caracterizando um resultado de busca 100% preciso. Chegando a esta condição, ambos algoritmos realizam uma parada e gera-se o tempo de execução total.

Os resultados são tabelados e aplicados a um gráfico, onde uma curva é ajustada aos pontos, possibilitando uma visualização das tendências performáticas. Para garantir maior precisão na mensuração dos tempos de execução, cada teste é repetido em um total de cinco iterações e calcula-se uma média aritmética simples a partir destes valores.

5. Resultados e discussão

O desempenho dos algoritmos diferiu acentuadamente na medida que a quantidade de dados aumentasse. Após o tamanho da amostra ultrapassar 10.000 votos, tornou-se inviável a mensuração de desempenho do algoritmo *Naive*.

Tabela 2. Tempo de execução dos algoritmos DP e Naive com diferentes N

Quantidade de dados (N)	DP (segundos)	Naive (segundos)
100	0	1,781
1.000	0,002	18,375999
10.000	0,018	187,606995
100.000	0,161	N/A
500.000	0,768	N/A
1.000.000	1,546	N/A
10.000.000	15,464	N/A

A característica do algoritmo *Naive* de usar a recursividade sem aproveitar soluções previamente calculadas tornou sua execução demasiadamente lenta, sendo irrealizável sua implementação em contextos reais onde a quantidade de dados trabalhada é algo substancial. Por meio desta solução, são feitas chamadas recursivas em escala exponencial, aonde muitas delas se repetem, desperdiçando memória e aumentando o tempo de processamento.

Por outro lado, o desempenho da solução usando programação dinâmica (DP) manteve um nível performática satisfatório com crescentes quantidades de dados. No teste envolvendo a maior quantidade de dados (N = 10.000.000), o algoritmo foi capaz de produzir uma resposta em aproximadamente 15 segundos.

A evidente eficiência da solução DP se dá pela estrutura algorítmica, que faz uso de técnicas de *memoization*. Isto garante que não se desperdice ciclos de execução, aumentando a eficiência do processamento e reduzindo drasticamente a quantidade de memória necessária para sua implementação.

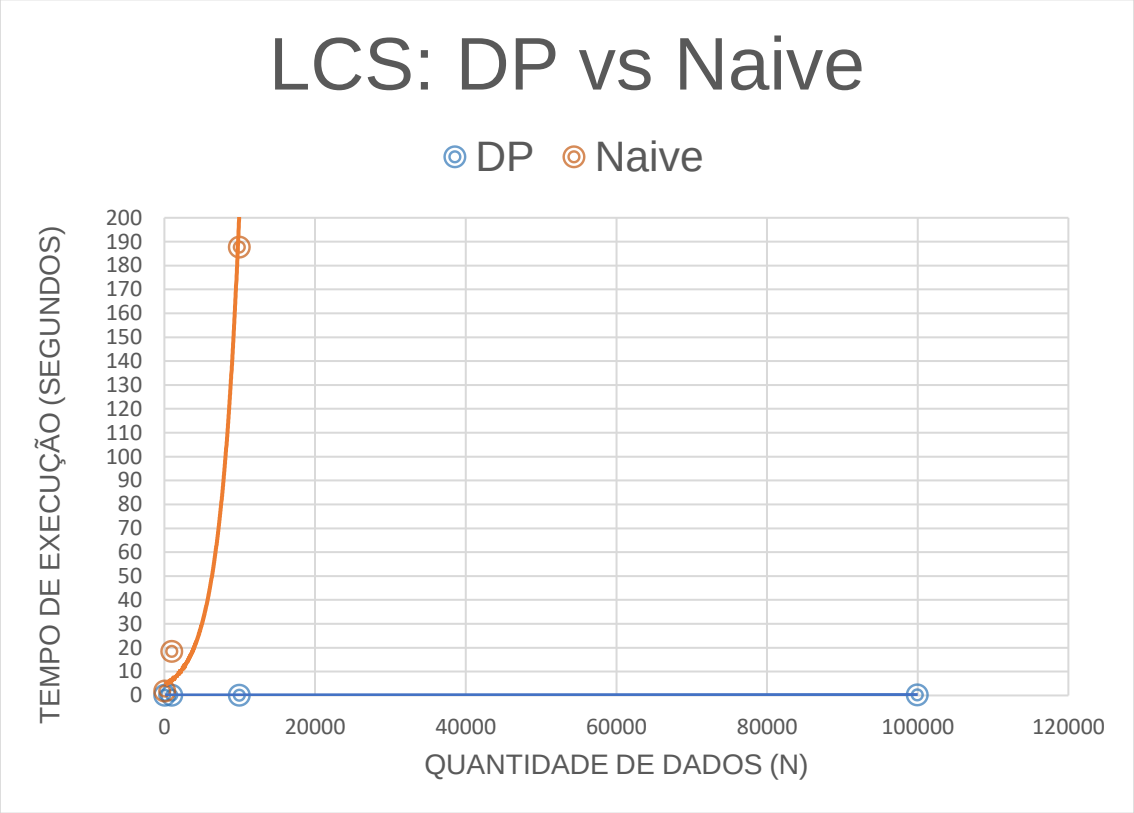

Figura 1. Tempo de execução dos algoritmos de LCS

6. Conclusão

Os avanços tecnológicos experimentados na área da votação propiciaram funcionalidades imprescindíveis para os profissionais responsáveis pela condução do processo eleitoral. O gerenciamento dos dados dos candidatos e dos eleitores é um aspecto fundamental na condução de uma eleição.

O modelo proposto neste artigo, com função de busca auxiliada por um algoritmo de LCS baseado na solução de Hirschberg (1975) fornece melhores opções de pesquisa aos usuários e auxilia no controle das informações. A implementação fundamentada na programação dinâmica demonstrou eficiência performática radicalmente melhor que a implementação *Naive*.

Para trabalhos futuros, sugere-se uma análise feita a partir de uma massa de dados ultrapassando 100 milhões de registros, buscando se aproximar o máximo possível de condições reais em uma eleição de um país populoso como o Brasil.

Referências

A. Wagner, R. and J. Fischer, M. (1974). The String-to-String Correction Problem. *Journal of the Association for Computing Machinery*, 21(1), pp.168-173.

Celes, W., Cerqueira, R. and Rangel, J. (2004). *Introdução a Estruturas de Dados*. 11th ed. Rio de Janeiro: Elsevier Editora Ltda.

Morgan, H. (1970). Spelling correction in systems programs. *Communications of the ACM*, 13(2), pp.90-94.

Rivest, R., Shamir, A. and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), pp.120-126.

S. Hirschberg, D. (1975). A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6), pp.341-343.

Tse.jus.br. (2014). Urna eletrônica — Tribunal Superior Eleitoral. [online] Available at: http://www.tse.jus.br/eleicoes/urna-eletronica/urna-eletronica [Accessed 21 Oct. 2018].

Tse.jus.br. (2013). Série urna eletrônica: da máquina de votar ao voto informatizado — Tribunal Superior Eleitoral. [online] Available at: http://www.tse.jus.br/imprensa/noticias-tse/2013/Setembro/serie-urna-eletronica-da-maquina-de-votar-ao-voto-informatizado [Accessed 20 Oct. 2018].