Relatório atividade sobre processamento de cadeia de caracteres

1. Introdução

Este trabalho tem como objetivo explorar e avaliar diferentes algoritmos de casamento de padrões e compressão de dados. Para isso, utilizamos o mesmo dataset da Parte 1 do trabalho de Hash, garantindo, também, como uma base consistente para análise comparativa caso seja necessário.

Na etapa de casamento de padrões, implementamos os algoritmos Boyer-Moore-Horspool (BMH) e Knuth-Morris-Pratt (KMP). Na etapa de compressão de dados, implementamos os algoritmos Run-Length Encoding (RLE) e Huffman, que representam abordagens distintas, e, após a compressão do dataset, calculamos a taxa de compressão para cada técnica e analisamos os resultados.

Além das implementações, este relatório apresenta uma comparação detalhada do tempo de execução dos algoritmos de busca e das taxas de compressão obtidas. Também discutimos as escolhas metodológicas, as ferramentas utilizadas.

Durante o desenvolvimento do projeto, não houve uma divisão específica de tarefas entre os integrantes do grupo. Em vez disso, todas as decisões, implementações e modificações foram realizadas de maneira colaborativa, com a participação conjunta dos dois membros. Esse modelo de trabalho permitiu uma troca constante de ideias e a contribuição coletiva em cada etapa do processo.

2. Metodologia

Algumas modificações foram feitas na primeira parte do trabalho, considerando que o arquivo tem aproximadamente 1.825.433 registros (excluindo o cabeçalho) o tamanho atual da tabela é muito grande: 10^9 + 7 (aproximadamente 1 bilhão de entradas), para uma tabela hash eficiente, usamos um tamanho diferente do tamanho indicado na primeira parte do trabalho, utilizando, dessa vez, um tamanho máximo de 2.607.773 registros.

Para a sessão de casamento de padrões, foram implementados os algoritmos Boyer-Moore-Horspool (BMH) e Knuth-Morris-Pratt (KMP).

O BMH consiste em pular múltiplas posições no texto sempre que ocorre um desencontro entre o padrão e o texto, construindo uma tabela de deslocamentos baseado nos caracteres do padrão de busca, durante a busca essa tabela é utilizada para encontrar mais eficientemente o padrão no texto, diminuindo o número de comparações.

O KMP é um algoritmo de casamento de padrões que evita comparações redundantes ao reutilizar informações sobre o padrão, antes da busca ele constroi uma um vetor de prefixos

que indica o deslocamento quando acontece um desencontro entre o padrão e o texto, usando esse vetor para comparar caracteres ainda não comparados.

Para a sessão de compressão de dados, foram implementados os algoritmos Run-Length Encoding (RLE) e Huffman.

O Run-Length Encoding (RLE) é um método simples de compressão que substitui sequências consecutivas de um mesmo símbolo por um par (símbolo, contagem). Esse método é altamente eficiente para textos ou imagens com grandes áreas de repetição, reduzindo significativamente o tamanho dos dados armazenados.

O algoritmo de Huffman é um método de compressão baseado em codificação prefixada, onde símbolos mais frequentes recebem códigos menores, e símbolos menos frequentes recebem códigos maiores. Ele constroi uma árvore de Huffman a partir das frequências dos símbolos, gerando uma codificação ótima para o conjunto de dados analisado.

Além disso, criamos o seguinte menu para facilitar a busca entre as implementações feitas no decorrer do trabalho:

Menu Principal

```
MENU PRINCIPAL:
1 - Buscar usando Tabela Hash
2 - Buscar usando Boyer-Moore-Horspool
3 - Buscar usando Knuth-Morris-Pratt
4 - Comprimir usando RLE
5 - Comprimir usando Huffman
6 - Sair
```

2.1 Casamento de Padrões

Para organizar a estrutura do código e facilitar a reutilização das funções de busca, foi criada uma classe específica para o casamento de padrões, chamada `Buscador`. Essa classe encapsula as duas funções de busca implementadas: `buscaBMH` e `buscaKMP`.

Os métodos escolhidos foram selecionados por sua relevância na literatura de algoritmos de casamento de padrão, apresentando um bom equilíbrio entre eficiência e complexidade de implementação. O BMH é especialmente eficiente para buscas em textos longos, enquanto o KMP minimiza comparações desnecessárias ao utilizar um vetor de prefixos para guiar a busca.

2.1.1 Implementação

A princípio, a implementação se iniciou pelo código de Boyer-Moore-Horspool, e depois pelo código de Knuth-Morris-Pratt.

A implementação do `buscaBMH` foi totalmente baseada no material disponibilizado em aula, incluindo os slides e explicações da professora. Para isso, utilizamos uma abordagem adaptada para trabalhar diretamente com arquivos CSV, otimizando algumas condições de parada para melhorar a eficiência em relação à versão original dos materiais de apoio. Além disso, a função auxiliar `desloca` foi implementada para calcular os saltos no texto.

Algoritmo 1: Vetor deslocamento

```
void desloca(const string& P, vector<int>& d) {
    int m = P.size();
    for (int i = 0; i < 256; i++) d[i] = m;
    for (int i = 0; i < m - 1; i++) d[P[i]] = m - 1 - i;
}
```

Algoritmo 2: Busca BMH

```
vector<registro*> buscaBMH(const string& padrao) {
    vector<registro*> resultados;
    ifstream arquivo(arquivo_csv);
    string linha;

    // Pula o cabeçalho
    getline(arquivo, linha);

    vector<int> d(256);
    desloca(padrao, d);

    while (getline(arquivo, linha)) {
        stringstream ss(linha);
        string nome;
        getline(ss, nome, ','); // Pula o ID
        getline(ss, nome, ','); // Pega o nome

        int pos = 0;
        if (nome == padrao) {
            registro* reg = criaRegistro(linha);
            if (reg) resultados.push_back(reg);
        }
    }
    arquivo.close();
    return resultados;
}
```

Para a implementação do `buscaKMP`, juntamente com `computaPrefixo`, seguimos o material apresentado em aula, garantindo que a busca ocorresse de forma eficiente ao evitar comparações desnecessárias. A adaptação do código permitiu que a função realizasse a busca diretamente no arquivo CSV, melhorando sua aplicabilidade ao contexto do trabalho.

Algoritmo 3: Computa prefixo

```
vector<int> computaPrefixo(const string& P) {
    int m = P.size();
    vector<int> pi(m, 0);
    int k = 0;
    for (int q = 1; q < m; q++) {
        while (k > 0 && P[k] != P[q]) k = pi[k - 1];
        if (P[k] == P[q]) k++;
        pi[q] = k;
    }
    return pi;
}
```

Algoritmo 4: Busca KMP

```
vector<registro*> buscaKMP(const string& padrao) {
    vector<registro*> resultados;
    ifstream arquivo(arquivo_csv);
    string linha;

    // Pula o cabeçalho
    getline(arquivo, linha);

    vector<int> pi = computaPrefixo(padrao);

    while (getline(arquivo, linha)) {
        stringstream ss(linha);
        string nome;
        getline(ss, nome, ','); // Pula o ID
        getline(ss, nome, ','); // Pega o nome

        int q = 0;
        for (int i = 0; i < (int)nome.size(); i++) {
            while (q > 0 && padrao[q] != nome[i]) q = pi[q - 1];
            if (padrao[q] == nome[i]) q++;
            if (q == (int)padrao.size() && nome.size() == padrao.size()) {
                registro* reg = criaRegistro(linha);
                if (reg) resultados.push_back(reg);
                break;
            }
        }
    }

    arquivo.close();
    return resultados;
}
```

2.1.2 Resultados e Análises

Para a análise dos resultados, pesquisamos em uma IA generativa para entender como poderíamos calcular o tempo de execução na linguagem C++, com isso obtivemos a função `high_resolution_clock` , disponível na biblioteca `chrono` do C++, para medir com precisão o tempo de execução dos algoritmos implementados. Dessa forma, para as opções um

(Tabela Hash), dois (BMH) e três (KMP) do menu principal, o tempo de processamento é registrado e impresso da seguinte maneira:

Algoritmo 5: Calculando tempo execução

```
auto inicio = high_resolution_clock::now();

\***************
Utiliza a implementação para determinado método de busca,
Podendo ser Tabela Hash, BMH, KMP.
*****************\

auto fim = high_resolution_clock::now();
auto duracao = duration_cast<milliseconds>(fim - inicio);
cout << "\nTempo de busca: " << duracao.count() << " ms" << endl;
```

Inicialmente, foram selecionados nomes presentes no início e no final do arquivo, além de um nome ausente no conjunto de dados. Em seguida, foi calculado o tempo de execução dos três métodos de busca utilizados – Tabela Hash, Boyer-Moore-Horspool (BMH) e Knuth-Morris-Pratt (KMP). É importante dizer que o tempo da Tabela Hash considera o tempo de construção da tabela nesse primeiro caso. Os resultados obtidos estão apresentados na tabela abaixo.

Para a busca no início, utilizamos o nome **Mary**, que é o primeiro da lista. Já para a busca no final, o nome escolhido foi **Zyrin**. Além disso, para cada método de busca, foi calculado o tempo médio considerando 10 buscas.

Tempo médio de execução (ms)			
	Busca no início	Busca no final	Busca inexistente
Tabela Hash	3091.8	2884.3	2889.1
Boyer-Moore-Horspool	578.3	591.7	593.8
Knuth-Morris-Pratt	702.7	738.8	781.7

Tabela 1 - Construindo e buscando na Tabela Hash

Já a segunda tabela, mostra os resultados para as buscas com a Tabela Hash já criada, utilizando as mesmas buscas

	Tempo médio de execução (ms)		
Método	Busca no início	Busca no final	Busca inexistente
Tabela Hash	0	0	0
Boyer-Moore-Horspool	578.3	591.7	593.8
Knuth-Morris-Pratt	702.7	738.8	781.7

Tabela 2 - Buscando com Tabela Hash construída

A Tabela Hash, na Tabela 1, apresentou tempos de execução significativamente mais altos em comparação com os outros métodos. Isso evidentemente acontece pelo fato dos resultados da Tabela considerarem o tempo de criação da Hash, que aumenta o tempo total de execução, já na Tabela 2 os resultados são tão pequenos que tornam a busca quase instantânea, visto que envolve apenas um cálculo de hash e uma verificação direta na estrutura.

Entre os métodos de Casamento de Padrão, o BMH se mostrou mais eficiente no caso apresentado, o KPM foi menos eficiente possivelmente pelo fato desse método ser mais utilizado para textos com muitos caracteres repetidos, que não é o caso do nosso arquivo teste.

Isso implica que, se houver muitas buscas subsequentes, a Tabela Hash é extremamente vantajosa, no entanto, se o foco for apenas algumas buscas, pode não valer a pena gastar tempo construindo a estrutura, e métodos diretos como Boyer-Moore-Horspool (BMH) podem ser mais eficientes.

2.2 Compressão de dados

A compressão de dados é um processo essencial para a otimização do armazenamento e transmissão de informação, reduzindo o tamanho dos arquivos sem perda de informação relevante. Para organizar a estrutura do código e facilitar a reutilização das funções de compressão, foi criada uma classe específica chamada `Compressor`. Essa classe encapsula as duas principais funções de compressão implementadas: `compressaoRLE` e `compressaoHuffman`.

Esses métodos foram selecionados estrategicamente para destacar suas diferenças fundamentais, uma vez que cada um possui um princípio de funcionamento distinto e é mais eficiente em cenários específicos. Enquanto um deles é altamente eficaz para compressão de textos que apresentam muitas repetições consecutivas, o outro é projetado para lidar com distribuições variadas de caracteres, otimizando a compactação com base na frequência de ocorrência. Essa escolha permitirá uma análise detalhada de suas características, evidenciando como cada técnica se comporta diante de diferentes tipos de entrada e proporcionando comparações mais claras e significativas nos resultados obtidos.

2.2.1 Implementação

Para a compressão Run-Length Encoding (RLE) começamos pela leitura do arquivo original e pela contagem de repetições do mesmo caractere, após contar a quantidade de repetições, escreve-se o caractere seguido do número de ocorrências. Como é comum representar a contagem com um byte, o código impede que a contagem ultrapasse 255 e inicia um novo bloco se necessário.

Todas as etapas necessárias para a implementação da compressão RLE foram realizadas com sucesso. A função `compressaoRLE` foi desenvolvida garantindo a correta identificação e

compactação de sequências repetitivas, otimizando a manipulação de strings para garantir eficiência na execução.

Algoritmo 6: Compressão RLE

```
double compressaoRLE(const string& arquivo_saida) {
    ifstream entrada(arquivo_entrada, ios::binary);
    ofstream saida(arquivo_saida, ios::binary);
    size_t tamanho_original = 0;
    size_t tamanho_comprimido = 0;
    char atual, anterior = 0;
    unsigned contador = 0;
    bool primeiro = true;

    while (entrada.get(atual)) {
        tamanho_original++;

        if (primeiro) {
            anterior = atual;
            contador = 1;
            primeiro = false;
            continue;
        }

        if (atual == anterior && contador < 255) {
            contador++;
        } else {
            saida.put(anterior);
            saida.put(contador);
            tamanho_comprimido += 2;
            anterior = atual;
            contador = 1;
        }
    }
    // Escreve o último grupo
    if (!primeiro) {
        saida.put(anterior);
        saida.put(contador);
        tamanho_comprimido += 2;
    }
    entrada.close();
    saida.close();

    return calcularTaxaCompressao(tamanho_original, tamanho_comprimido);
}
```

Para a implementação do algoritmo de Huffman, foi necessário compreender o funcionamento do método, estruturando sua lógica em funções auxiliares. A função `construirArvoreHuffman` é responsável por construir a árvore de Huffman baseada na frequência dos caracteres, enquanto `gerarCodigosHuffman` gera a tabela de códigos binários para cada caractere presente no texto.

Primeiramente, Um mapa (`map<char, unsigned> frequencias`) é criado para armazenar a quantidade de ocorrências de cada caractere, a árvore é construída utilizando uma fila de

prioridade (`priority_queue`), onde os nós menos frequentes são combinados para formar novos nós internos após construir a árvore, percorre-se ela para atribuir códigos binários a cada caractere. O texto original é substituído pelos códigos binários correspondentes, armazenando o resultado compactado no arquivo de saída que, por fim, é salvo.

Algoritmo 7: Compressão Huffman

```
double compressaoHuffman(const string& arquivo_saida) {
    ifstream entrada(arquivo_entrada, ios::binary);

    map<char, unsigned> frequencias;
    string dados;
    char c;
    size_t tamanho_original = 0;

    while (entrada.get(c)) {
        frequencias[c]++;
        dados += c;
        tamanho_original++;
    }
    entrada.close();

    auto raiz = construirArvoreHuffman(frequencias);
    auto codigos = gerarCodigosHuffman(raiz);

    // Salva o arquivo comprimido
    ofstream saida(arquivo_saida, ios::binary);

    // Salva a tabela de códigos
    salvarCodigosHuffman(codigos, saida);

    // Salva os dados comprimidos
    salvarDadosComprimidos(dados, codigos, saida);

    size_t tamanho_comprimido = saida.tellp();
    saida.close();

    return calcularTaxaCompressao(tamanho_original, tamanho_comprimido);
}
```

Assim, a implementação do algoritmo de compressão Huffman foi concluída com sucesso. A função `compressaoHuffman` foi desenvolvida para construir a árvore de Huffman com base na frequência dos caracteres, gerando códigos binários eficientes e garantindo a compressão.

2.2.2 Resultados e Análises

Para avaliar o desempenho dos métodos de compressão, foram realizadas medições do tempo de execução e da taxa de compressão. O tempo de execução foi calculado utilizando a biblioteca chrono, enquanto a taxa de compressão foi determinada com base na razão entre o tamanho original e o tamanho do arquivo compactado.

Algoritmo 8: Calcular compressão

```
double Compressor::calcularTaxaCompressao(size_t tamanho_original, size_t
tamanho_comprimido) {
    return (1.0 - static_cast<double>(tamanho_comprimido) / tamanho_original) * 100.0;
}
```

Algoritmo 9: Calculando tempo de execução

```
auto inicio = high_resolution_clock::now();

\****************
Utiliza a implementação para determinado método de compressão
seja Run-Length Encoding, seja Hulffman
*****************\

auto fim = high_resolution_clock::now();
auto duracao = duration_cast<milliseconds>(fim - inicio);
cout << "\nTempo de compressão: " << duracao.count() << " ms" << endl;
```

Run-Length Encoding		Huffman	
Taxa de compressão	Tempo de execução (ms)	Taxa de compressão	Tempo de execução (ms)
-89.18%	1694	41.85%	25444

Tabela 3 - Taxa de compressão e tempo de execução para cada método

Os resultados obtidos indicam que a compressão Run-Length Encoding (RLE) pode, em alguns casos, resultar em um grande aumento do tamanho do arquivo, como evidenciado pela taxa de compressão negativa de -89,18%. Isso ocorre porque o RLE é eficiente apenas quando há muitas repetições consecutivas de caracteres, mas pode ser ineficaz em dados com pouca redundância.

Já a compressão de Huffman apresentou uma taxa de compressão positiva de 41,85%, demonstrando sua capacidade de reduzir significativamente o tamanho do arquivo, especialmente quando há uma distribuição desigual de caracteres. No entanto, isso veio ao custo de um tempo de execução maior (25.444 ms) em comparação com o RLE (1.694 ms).

Com base nesses resultados, pode-se concluir que a escolha do algoritmo de compressão depende do tipo de dado que está sendo compactado. O RLE é vantajoso para arquivos com muitas repetições consecutivas, enquanto o Huffman é mais adequado para compressão eficiente em arquivos com distribuições variadas de caracteres e isso ficou evidente no nosso teste.

3. Considerações finais

O presente trabalho abordou duas técnicas fundamentais no processamento e manipulação de grandes volumes de dados textuais: compressão de dados e casamento de padrões. Ambas desempenham papeis complementares no contexto da otimização de armazenamento e eficiência na busca por informações dentro de arquivos extensos.

O casamento de padrões foi utilizado para realizar buscas eficientes dentro dos arquivos CSV. A implementação dos algoritmos Boyer-Moore-Horspool (BMH) e Knuth-Morris-Pratt (KMP) possibilitou uma análise comparativa entre abordagens distintas. O BMH demonstrou um desempenho superior para buscas diretas em textos não altamente repetitivos, enquanto o KMP mostrou-se mais eficiente em cenários onde há muitas repetições de caracteres no padrão de busca.

A análise experimental revelou, também, que, embora a Tabela Hash tenha um custo inicial de construção elevado, sua eficiência em buscas subsequentes é notável, tornando-a ideal para cenários onde múltiplas consultas são realizadas no mesmo conjunto de dados.

A compressão foi utilizada como um meio de reduzir o espaço ocupado pelos arquivos, facilitando o armazenamento e a transmissão de dados. Através dos métodos implementados, foi possível observar que técnicas como Huffman apresentam bons resultados, especialmente em arquivos contendo grandes quantidades de repetições ou padrões previsíveis. Todavia, Run-Length Encoding (RLE) demonstrou limitações significativas, especialmente quando aplicada a arquivos CSV. O método, que comprime sequências repetidas de caracteres, mostrou-se ineficaz para esse tipo de dado, pois os arquivos não continham padrões longos e contínuos de repetição

Os experimentos mostraram que a taxa de compressão depende fortemente da redundância nos dados, sendo que arquivos altamente repetitivos apresentam melhor desempenho. No entanto, um dos desafios enfrentados na compactação é o custo computacional envolvido na codificação e decodificação, que pode impactar o tempo de processamento dependendo do tamanho do arquivo.