Processamento de Cadeia de Caracteres - Projeto II

Ferramenta IPMT (Versão Corrigida)

1. Alunos

José Nilton de Oliveira Lima Júnior <jnolj>, implementou o algoritmo de indexação e de busca. Construiu o frontend da ferramenta.
Júlia Feitosa <mjfc>, implementou o algoritmo de compressão LZ78. Realizou os testes.

2. Implementação

De antemão, estabeleceremos as notações a serem utilizadas ao longo do relatório, descritas abaixo:

$$m = \text{tamanho do padrão}$$
$$n = \text{tamanho do texto}$$
$$\alpha = \text{tamanho do alfabeto}$$
$$r = \text{taxa de compressão (compression ratio)}$$

A ferramenta *ipmt* possui dois modos de operação: modo de indexação e modo de busca. O modo de indexação recebe como argumento o nome de um arquivo de texto, criando um arquivo de índices comprimido. Esse arquivo permite o casamento offline de padrões. O modo de busca recebe como parâmetro um arquivo de índices (.idx) comprimido e um padrão ou arquivo de padrões que se deseja buscar. A linguagem escolhida pela equipe foi C++. Os algoritmos desenvolvidos foram o Suffix Array e o LZ78.

Para as operações de I/O com os arquivos, utilizamos as funções padrão das bibliotecas ifstream e ostream. A equipe optou por não reutilizar a classe FileReader desenvolvida no projeto anterior, visto que a classe não apresentou um ganho de performance relevante.

2.1 Algoritmo de Indexação e Busca

O modo de indexação recebe como entrada um arquivo de texto e, utilizando o algoritmo Suffix Array, gera os índices a partir do texto. Em seguida, o compressor LZ78 é acionado para comprimir e salvar no formato .idx.

A implementação do algoritmo de indexação utiliza a estrutura auxiliar *SuffixInfo* para otimização e clareza do código. A estrutura contém o índice de início do sufixo, seu rank lexicográfico entre os sufixos e o rank da do sufixo *j bits* a frente do atual. A construção do *Suffix Array* utiliza um algoritmo $O(n*\log^2(n))$ para sua construção, ordenando iterativamente o *Suffix Array* considerando os primeiros *j bits* do sufixo. Além do *Suffix Array,* o algoritmo também gera os vetores auxiliares *leftLCP* e *rightLCP*, que correspondem ao maior prefixo em comum entre os sufixos da extremidade esquerda e direita,

respectivamente. Também é gerado um vetor *prefix* auxiliar, que guarda a ordenação dos sufixos considerando os j primeiros bits. O vetor *prefix* é utilizado para a construção de *leftLCP* e *rightLCP*, porém não é comprimido pois não é necessário para a busca de padrões.

2.2 Algoritmo de Compressão

O algoritmo de compressão escolhido foi o LZ78, ele foi implementado armazenando o dicionário num *unordered_map<string, int>* e o código gerado em um *vector<pair<int, char>>*.

Ademais, o dicionário utilizado no LZ78 não é robusto para textos pequenos, considerando que não há muita repetição de padrões, o que pode resultar num aumento do texto codificado. Para textos com tamanho >= 5MB já ocorre uma compressão satisfatória ao utilizar o LZ78.

3. Testes

Os testes foram executados a partir de scripts bash, que fazem a chamada à ferramenta ipmt, com diferentes parâmetros inseridos pelo usuário dependendo do que se deseja testar. Os scripts encaminham para arquivos de texto informações sobre o tempo de execução do programa (determinado pelo valor "real" do comando *time*), Os valores apresentados consideram a média aritmética do tempos de execução da ferramenta para parâmetros distintos 5 vezes. Para o teste dos algoritmos foram utilizados as bases de dados do Pizza&Chilli (http://pizzachili.dcc.uchile.cl/texts.html). Do corpus utilizamos estes arquivos:

- Pitches (Sequência de arquivos MIDI) 50MB
- Sources (Arquivo de código C /Java) 50, 100, 200MB
- DBLP.xml (Arquivo estruturado XML) 50, 100, 200MB
- English (Conjunto de arquivos em inglês) 50, 100, 200MB

Os padrões foram obtidos de uma lista com as 1000 palavras mais comuns da língua inglesa (https://www.ef.com/english-resources/english-vocabulary/top-1000-words/) e também de trechos retirados diretamente dos textos, com não mais do que 200 caracteres.

Os testes foram realizados numa máquina virtual VMWARE Workstation, com Ubuntu 18.04.1 LTS 64-bit, 3.8GB de RAM, Intel® Core™ i7-7500U CPU @ 2.70GHz × 4, GNOME 3.28.2.

O computador host possui a seguinte configuração: Windows 10 Single Home Language 1703 64-bit, 8 GB de RAM , Intel® Core™ i7-7500U CPU @ 2.70GHz - 2.90GHz. O compilador utilizado foi gcc version 7.3.0 (Ubuntu 7.3.0-27 ubuntu1~18.04).

3.1 Compressão

O script do primeiro teste utiliza lê um arquivo contendo os nomes de diversos textos , lendo um por vez e executando compressão para cada um. O tempo de execução do ipmt foi redirecionado para um arquivo, e após o cálculo da média, a ferramenta nativa de gráficos do Google Docs foi utilizada para a construção dos gráficos. Neste teste, procuramos relacionar

de forma confiável o tamanho do texto e o tempo de compressão. A ferramenta foi invocada para três arquivos diferentes, porém com o mesmo tamanho. O tempo de execução final é a média aritmética do tempo de compressão desses arquivos. Após executar essa tarefa cinco vezes, o valor final foi obtido. O resultado após a execução do primeiro teste encontra-se na Figura 3.1.

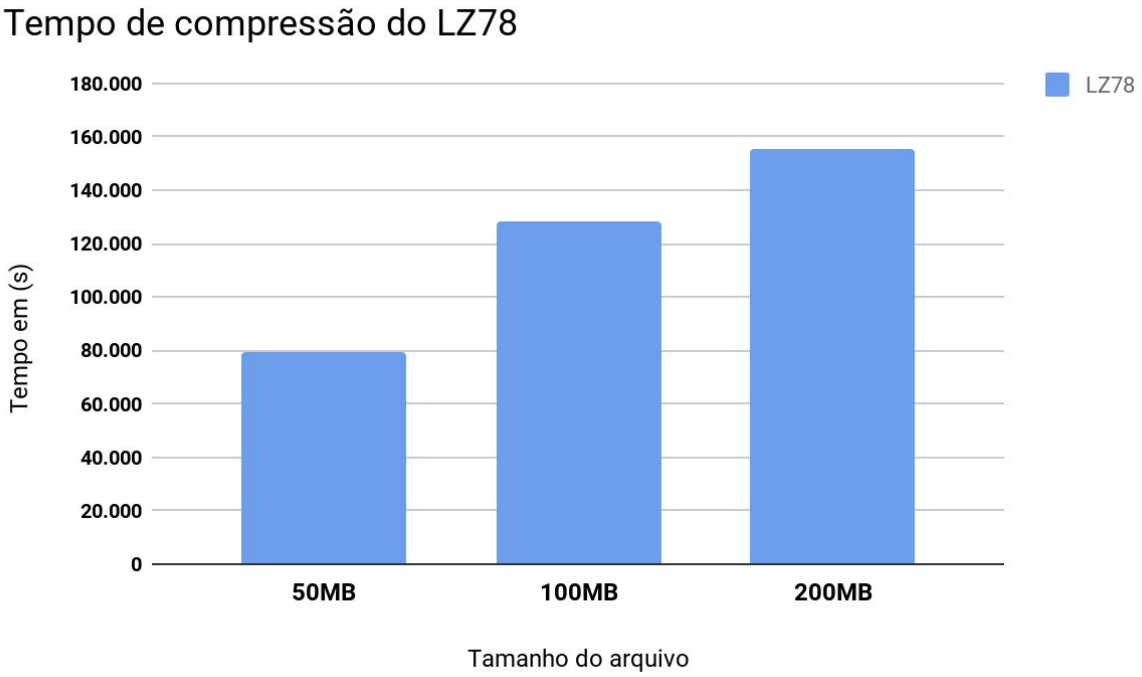

Figura 3.1 - Teste 1: Comparação do desempenho do LZ78 para diferentes tamanhos de arquivos

O segundo teste permitiu analisar melhor o desempenho dos algoritmo de compressão levando em consideração o conteúdo de cada texto. Ele repete a mecânica do primeiro teste, da leitura e compressão de uma lista de arquivos, porém não somamos o tempo de compressão dos arquivos. Porém isso permitiu mostrar como o tempo de compressão varia de acordo com o tipo de texto. Os três arquivos usados foram um código fonte em C/Java, um arquivo XML e um arquivo em inglês. Os resultados foram agrupados de acordo com o tamanho de cada arquivo. Os resultados podem ser visto na Figura 3.2, logo abaixo.

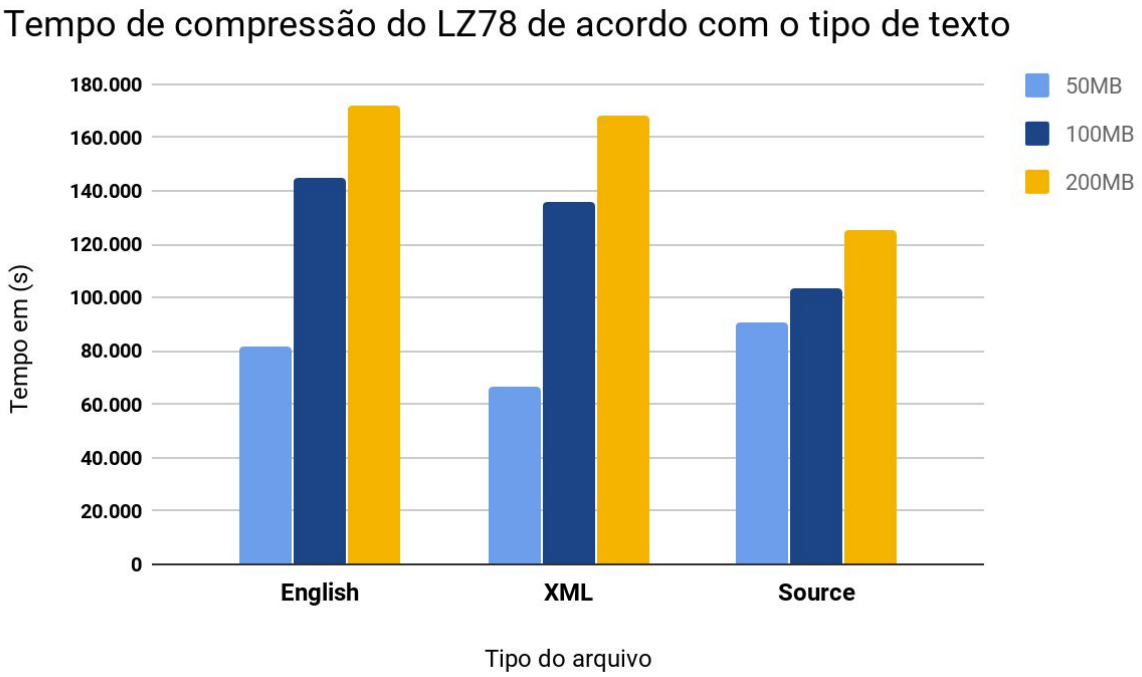

Figura 3.2 - Teste 2: Comparação do desempenho do LZ78 de acordo com o tipo de arquivo

O terceiro teste serviu para mostrar como a taxa de compressão do LZ78 se altera de acordo com o tamanho do arquivo. O procedimento do Teste 1 foi repetido, dessa vez calculando a taxa de compressão, 1 - (*tamanho_comprimido*/*tamanho_original)* .

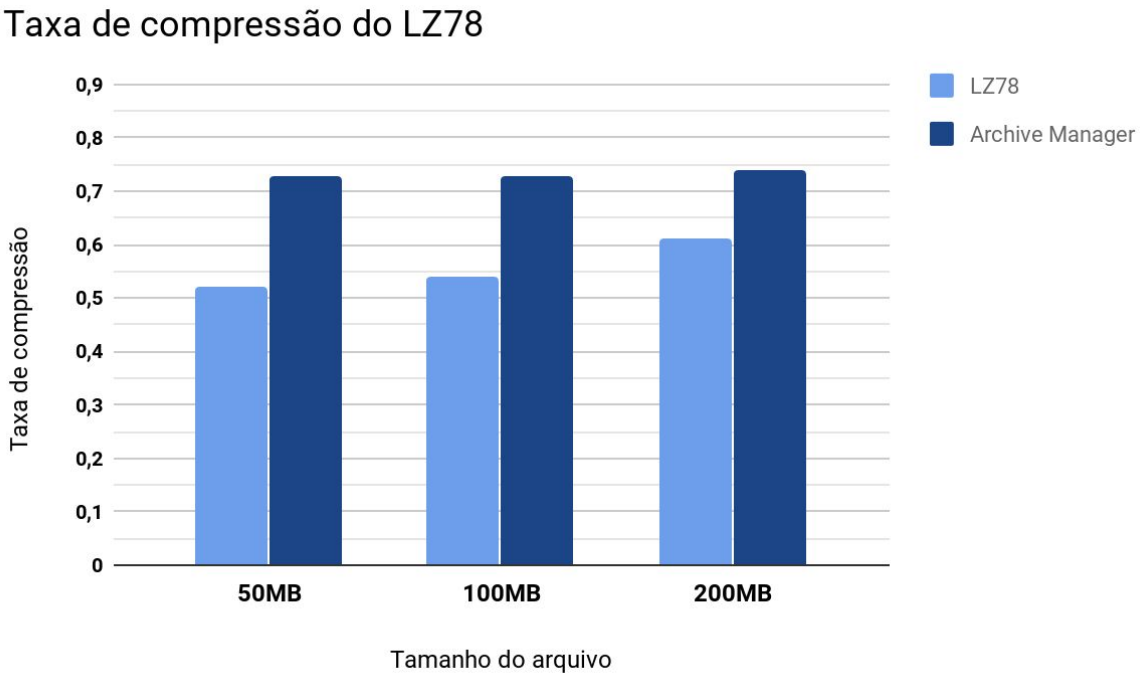

Figura 3.3 - Teste 3: Taxa de compressão do desempenho do LZ78 de acordo com o tamanho de arquivo

O quarto teste auxiliou a relacionar a taxa de compressão com o conteúdo de cada texto. O procedimento do Teste 2 foi repetido, calculando a taxa de compressão, 1 - (*tamanho_comprimido*/*tamanho_original)* . O desempenho foi comparado com o da ferramenta Archive Manager.

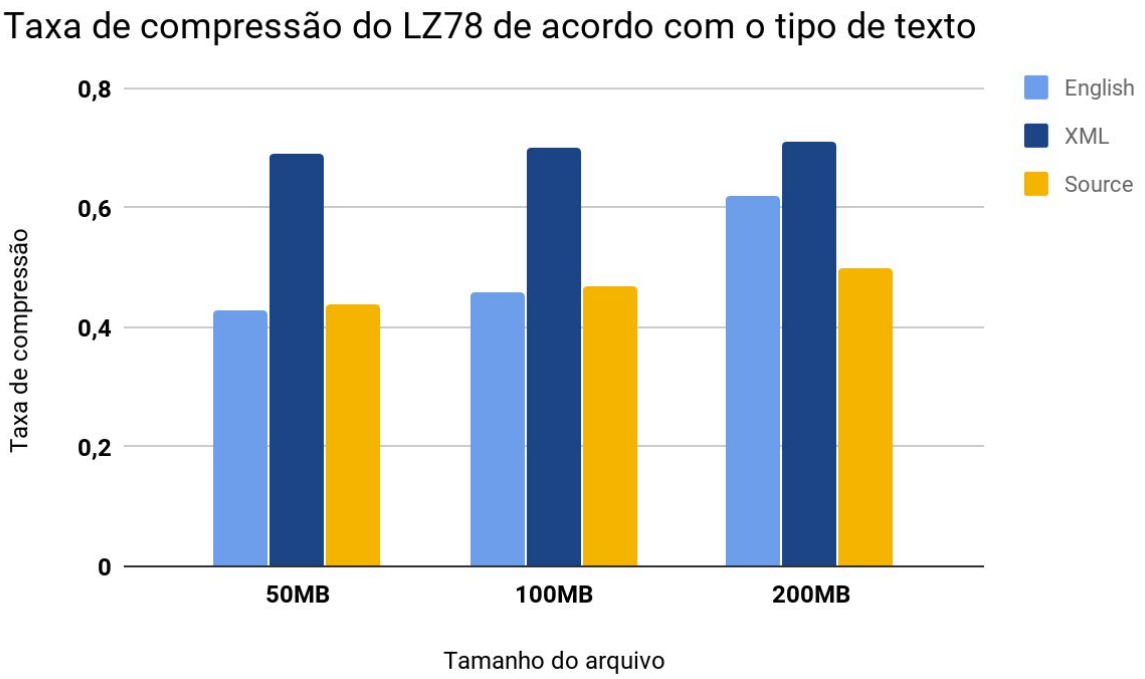

Figura 3.4 - Teste 4: Comparação da taxa de compressão do LZ78 de acordo com o tipo de arquivo

3.2 Indexação e Busca

O quinto teste mostra o desempenho da busca com o *SuffixArray* para um padrão de um determinado tamanho. Para realizar esse teste, quatro padrões do mesmo tamanho foram utilizados, e o resultado final é a média aritmética do tempo de busca desses padrões. Esses padrões são palavras da língua inglesa. Os textos utilizados também são em inglês.

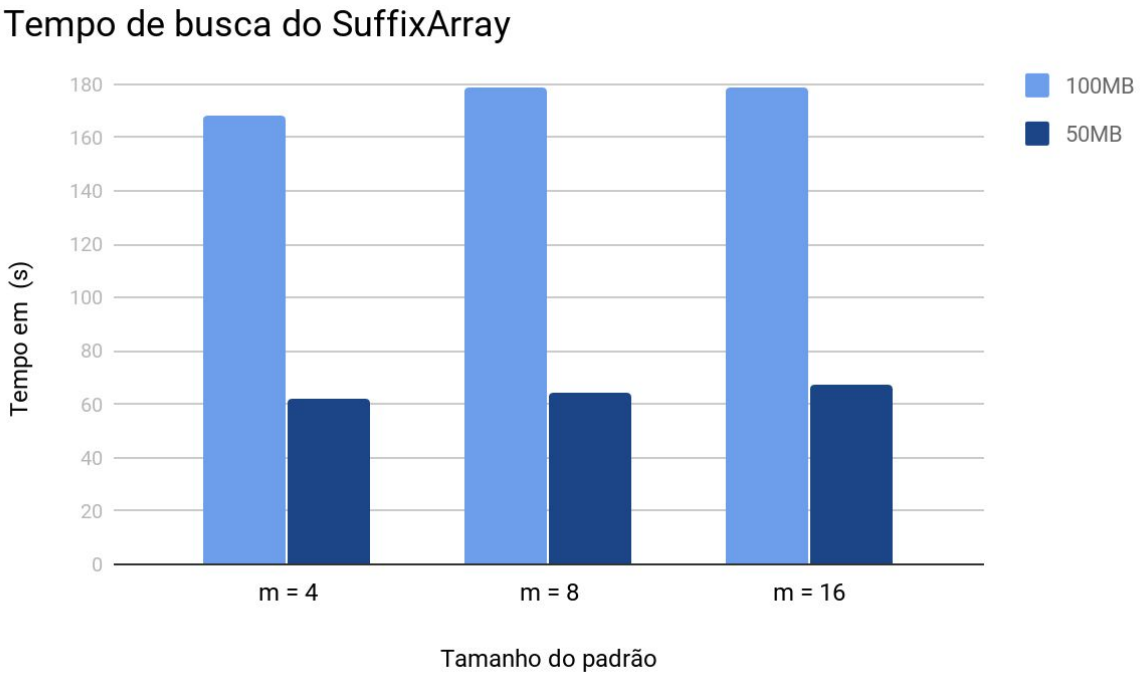

Figura 3.5 - Teste 5: Comparação do tempo de busca do SuffixArray com o tamanho do arquivo

Neste teste é possível perceber que o tamanho do padrão não influencia muito no tempo de busca do *SuffixArray*, então uma possível conclusão é que o tamanho do texto a ser processado é muito mais relevante para o tempo de execução.

4. Bugs

Para arquivos de tamanho >= 500MB, podem ocorrer erros na execução do algoritmo de compressão. Isso se deve à limitação da implementação do dicionário do *LZ78*, que dependendo do tamanho e do conteúdo do texto, pode crescer demasiadamente, ocasionando erros de memória. Uma estratégia que havia sido adotada para corrigir esse erro mas foi descartada, foi a renovação do dicionário a cada 256 entradas, porém isso estava ocasionando que o arquivo comprimido se tornasse maior que o arquivo original.

5. Correções

A versão inicial do programa apresentava erros para arquivos com tamanho >= 500kb. Após um estudo do código, descobrimos existiam dois problemas. O primeiro problema estava presente na classe *Suffix Array*, onde um *array* estático de inteiros era criado a partir do tamanho do texto. A criação de um array com um n muito grande causa *segmentation fault* pois o programa não teria como alocar estaticamente esse *array*. Para corrigir isso, todos os vetores do programa foram transformados para *std::vector* e, por precaução e otimização, as utilizações de *int* foram trocadas para *long long int.* Outro problema detectado foi que o caractere \0 estava sendo considerado como parte do texto após a descompressão, ocasionando erros na busca. Esse caractere é inserido no texto codificado quando o loop do

algoritmo LZ78 termina mas ainda resta uma string a ser codificada. Portanto, ele se torna necessário para a descompressão correta do texto, mas não deve ser parte do mesmo. Então, a função do suffix array remove esse caractere do fim do texto, antes de processá-lo. Após essas alterações, o programa funciona como especificado.

6. Conclusões

O desempenho do algoritmo de compressão foi satisfatório, atingindo boas taxas de compressão, apesar da limitação do tamanho do arquivo do texto. O uso do LZ78 seria indicado para arquivos com tamanho >= 5MB. Para o *SuffixArray*, o desempenho é proporcional ao tamanho do arquivo e não se altera significativamente com o aumento do tamanho do padrão, o que é condizente com o que foi aprendido em sala.