Universidade Federal de Pernambuco

Centro de Informática

Processamento de Cadeias e Caracteres

Relatório do Projeto 2

Aluno:	Arnaldo Rafael Morais Andrade	Data: 09/12/2018
	Marvson Allan Pontes de Assis	

1) Identificação

Nosso grupo inclui **Arnaldo Rafael Morais Andrade** e **Marvson Allan Pontes de Assis**. Arnaldo cuidou da parte do vetor de sufixos e interface, já Marvson, ficou com a compressão/descompressão e os testes.

2) Implementação

2.1) Funcionamento da Ferramenta

O uso é bastante simples, consiste de 4 partes independentes. Primeiramente faz-se um vetor de sufixos, a partir dele e do texto a ser procurado, construímos um arquivo .idx baseado na compressão LZ78. Para o modo de busca, fazemos o caminho inverso até termos os valores correspondidos pelo vetor de sufixo, ou seja, 3 vetores auxiliares e o próprio texto. Então é realizada a busca do padrão ao longo do texto, como visto em sala, foi utilizado do LCP-LR.

2.2) Detalhes Relevantes

a. Estratégias de Leitura

A leitura foi feita linha por linha, não foram feitas otimizações adicionais aqui.

b. Convenções

Usamos namespaces em vez de classes para ficar menos complexo e não precisar ficar instanciando classes. Qualquer estado é mantido pelo ipmt.

Escolhemos usar o namespace completo sempre e evitar o uso de “using namespace = …”, exceto quando isso prejudica a leitura.

c. Indexação

A fim de manter os vetores correspondentes do array de sufixos e o próprio texto, criamos uma string contendo os 4 principais elementos, separados por “$”. Foram concatenados um por um e o elemento final seria o próprio texto. A partir dessa nova variável, criamos o arquivo .idx.

d. Compressão e descompressão

O algoritmo de compressão e descompressão escolhido foi o LZ78. O mesmo foi implementado utilizando uma *trie* e ponteiros na forma *shared_ptr*, que inclui *garbage collector* do tipo que utiliza a contagem de dependências do ponteiro, e quando chega a zero efetua o *destroy*. O *shared_ptr* também é interessante por ser capaz de lidar facilmente com *Nulls* por padrão, de forma que nem precisamos se preocupar com essas coisas.

Uma preocupação era a maneira como percorrer a *trie*. Inicialmente foi feito com uma recursão, mas havia a preocupação de estourar a pilha, então modificamos para usar iteração mesmo. Algumas otimizações quanto a declaração de variáveis fora de laços e o uso de valores auxiliares foram conduzidas. A expectativa é que a compressão e descompressão pela *trie* seja mais rápida.

e. Busca

Uma vez o arquivo .idx descomprimido, podemos fazer a operação inversa para obter os vetores, úteis para busca e o texto. Basta percorrermos a super string e fazemos o processo de volta a partir do caractere conhecido "$". Com os itens em mãos, foi realizado a busca no vetor de sufixos, com o auxílio do LCP. Auxílio esse que reduz a forma que é feita busca binária do padrão ao longo do texto. Não foram efetuadas mudanças em relação ao algoritmo visto em sala.

2.3) Limitações

Para arquivos maiores que 100kB o tempo de compressão cresce bastante, inviabilizando os testes para tamanhos superiores. Porém é apenas uma limitação do tempo, a verdadeira limitação vem do número de padrões na compressão. Esse número não pode ser maior que *INT_MAX*.

Não é possível mostrar as linhas em que as ocorrências ocorrem.

3) Correções

Estávamos com um tempo absurdo na construção dos vetores Rlcp e Llcp, quando na verdade não deveria acontecer. Aconteceu que numa função recursiva (arquivo sarray.cpp , função fill_lcplr, linha 144), sua chamada estava sendo feita por valor e não referência, assim estava-se criando várias cópias dos parâmetros, desnecessariamente e assim, comprometendo o desempenho da construção dos vetores.

Então foi feito uma otimização em todas as chamadas da função, passando, sempre que possível, as chamadas por referência e valores imutáveis (const &).

Outro ponto importante foi na interpretação do index (leitura após ter sido criado o arquivo .idx), em que convertendo uma string para um vetor de inteiros, estava-se armazenando o valor de cada char. E.g: “156” -> “[1][5][6]”. Ou seja, falhando em casos em que o número continha 2 dígitos. Esse fator comprometeu no cálculo de ocorrências do padrão no texto.

A estrutura do index foi mantida (armazenava-se em uma string os 3 vetores separados por “$” e por fim o texto) e sua escrita foi em binário.

Já a sua leitura, foi modificada para atender todos os casos. Apenas mudou-se o fator multiplicativo de acordo com a casa do dígito. E.g: “156” -> 1*100+5*10+6*1 -> “[156]”.

4) Testes e Resultados

4.1) Metodologia

As comparações foram feitas manualmente e executadas com um arquivo shell unix. Foram conduzidos 30 testes para cada combinação de parâmetros, para que exista minimamente propriedade estatística nos testes. Nos gráficos na seção de resultados, é possível visualizar o desvio padrão das medidas como um traço vertical. Em alguns gráficos, o desvio padrão é insignificante para ser visualizado. De cada 30 testes, foi aplicada a média e desvio padrão para demonstrar graficamente os resultados.

Os testes foram feitos em uma máquina com sistema operacional Ubuntu 16.04, Processador Intel® Core™ i7-3770 CPU @ 3.40GHz × 8 , Memória 7,7 GiB, Gráficos Intel® Ivybridge Desktop. Compilador utilizado GCC/G++ 5.4.0.

4.2) **Resultados**

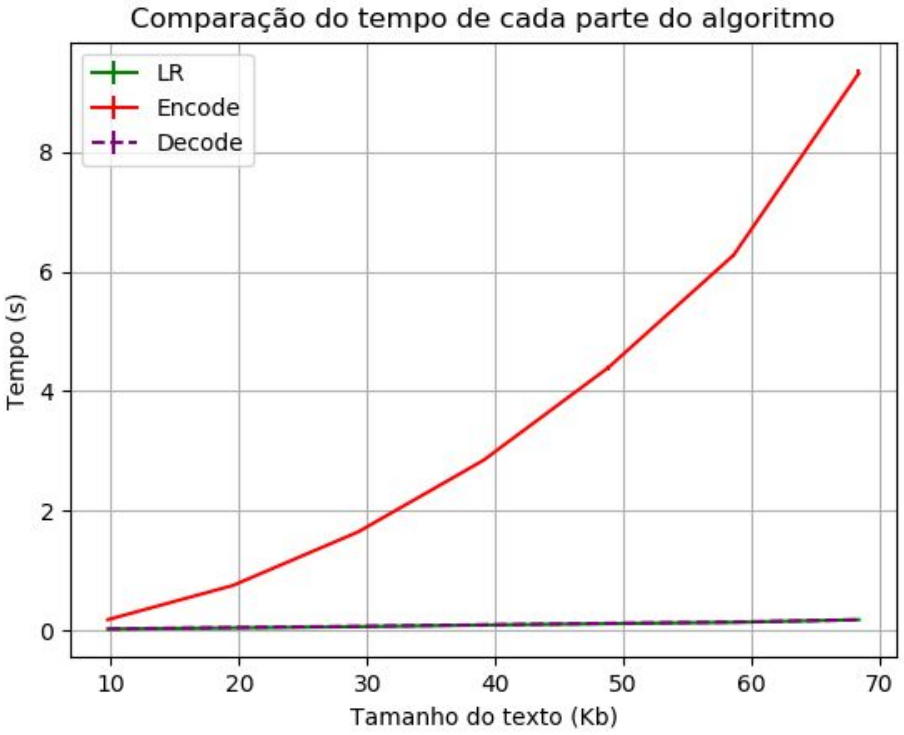

Figura 1 - Comparação do tempo de cada parte diferente algoritmo

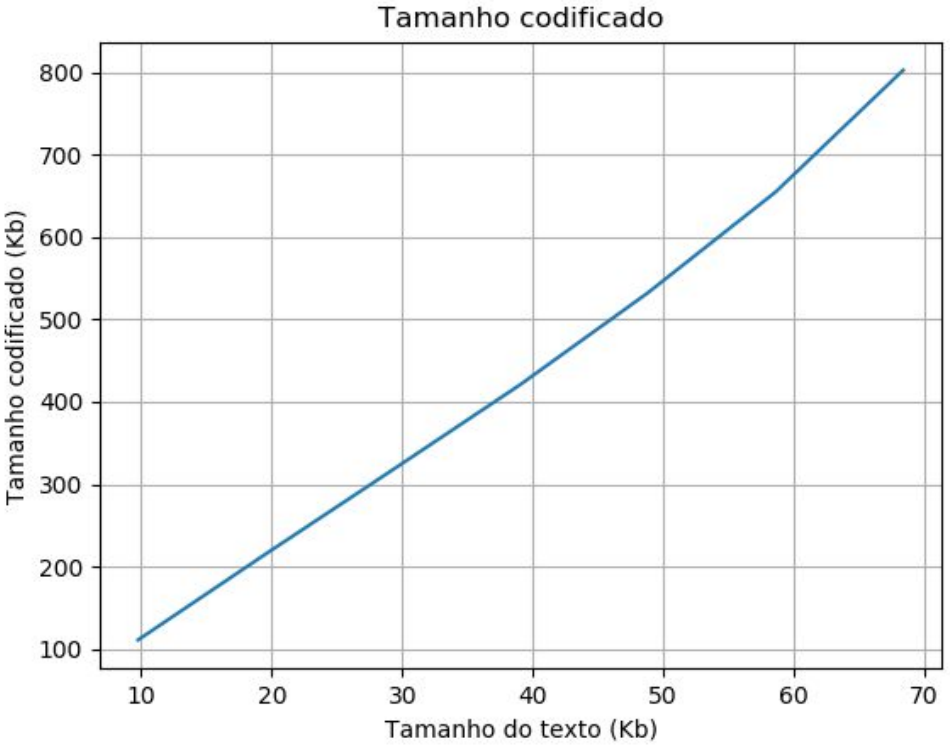

Figura 2 - Tamanho do texto codificado dado o tamanho do texto original

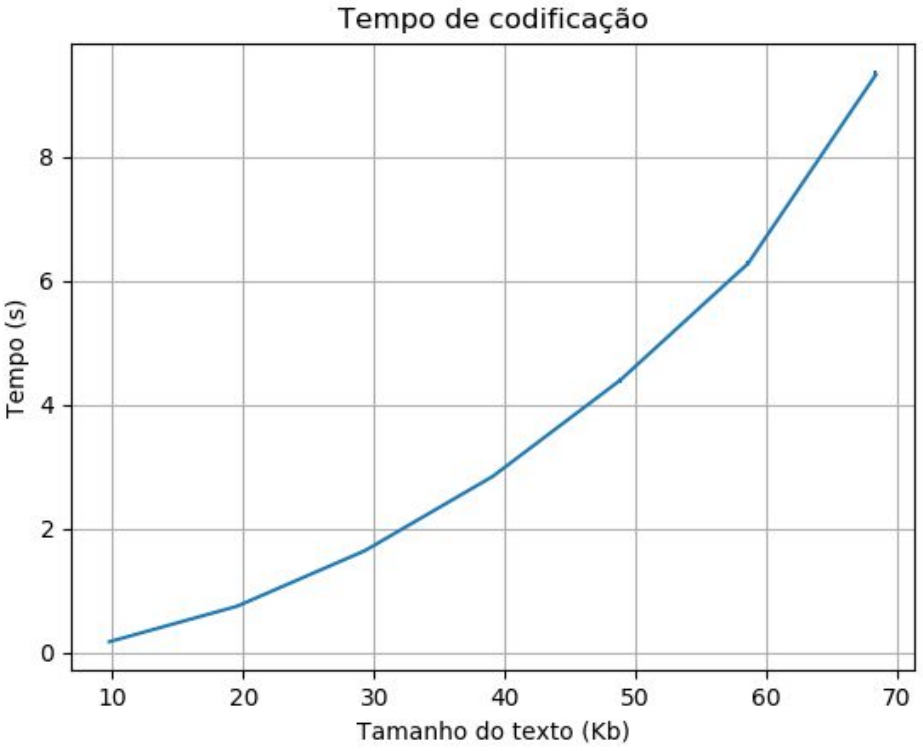

Figura 3 - Tempo de codificação

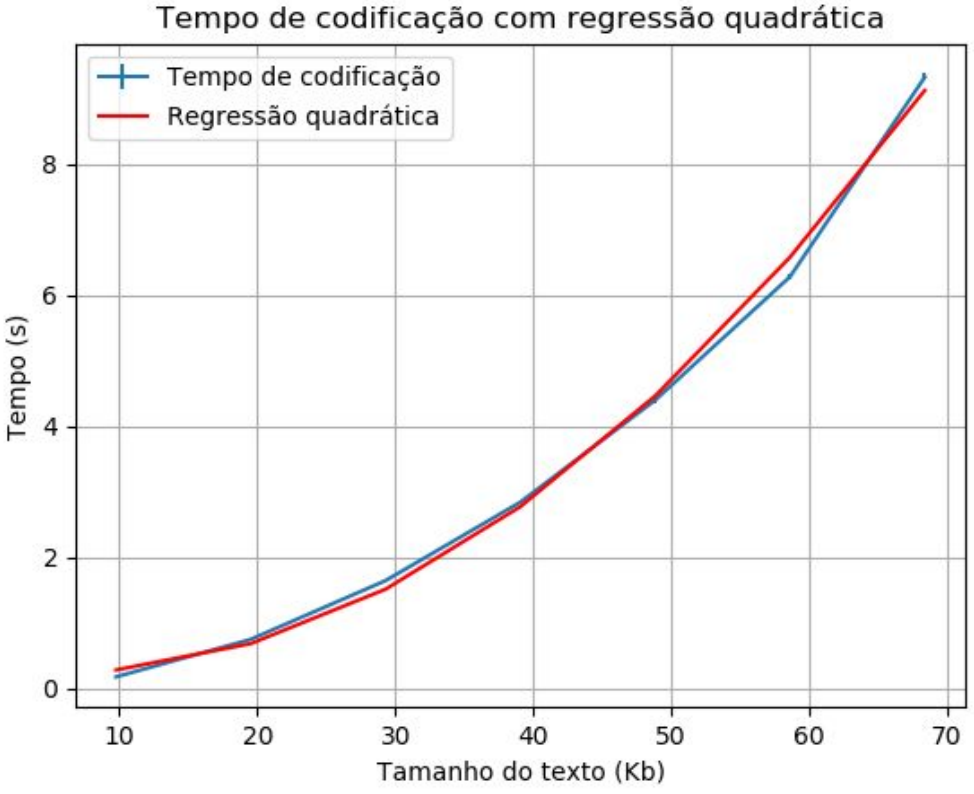

Figura 4 - Tempo de codificação com regressão quadrática

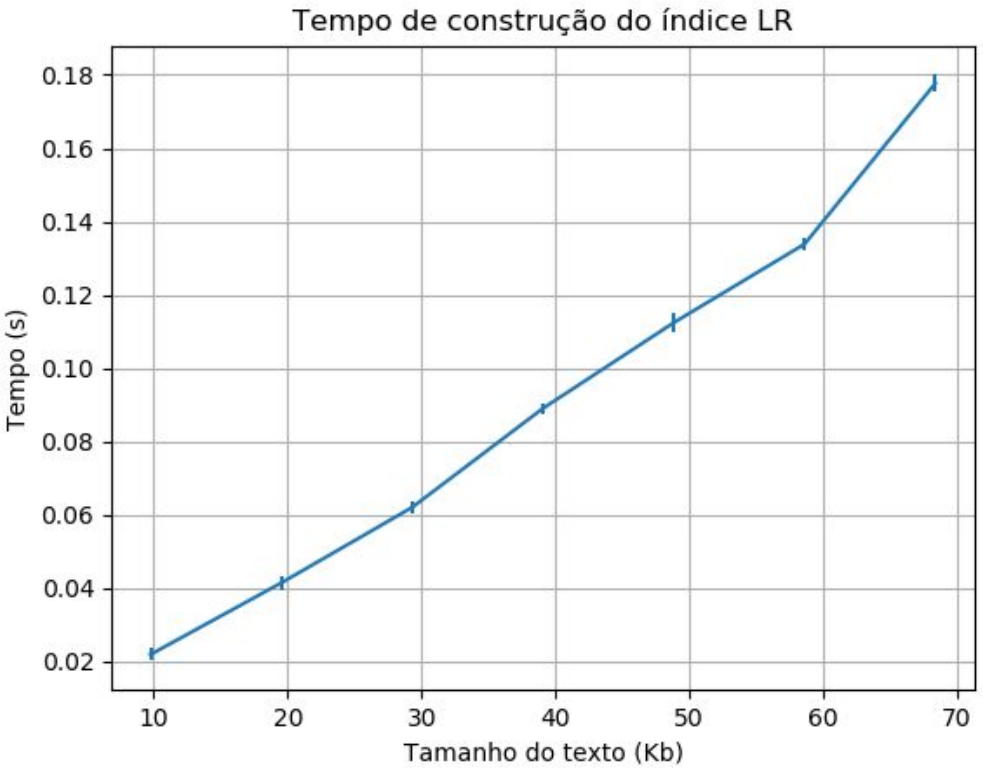

Figura 5 - Tempo de construção do índice LR

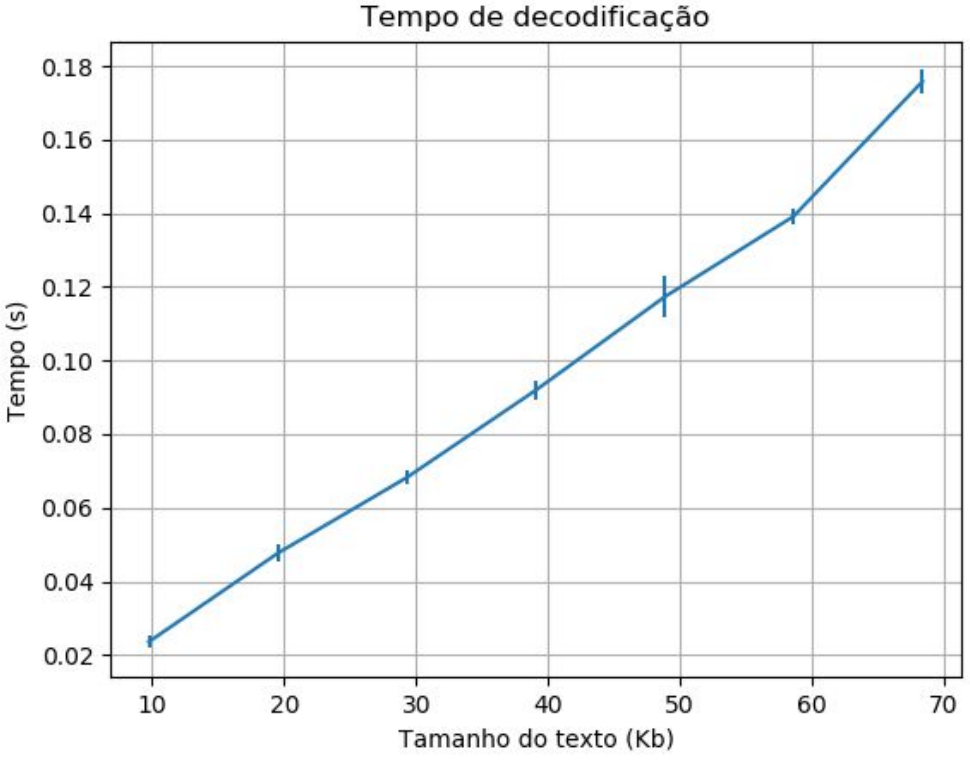

Figura 6 - Tempo de decodificação

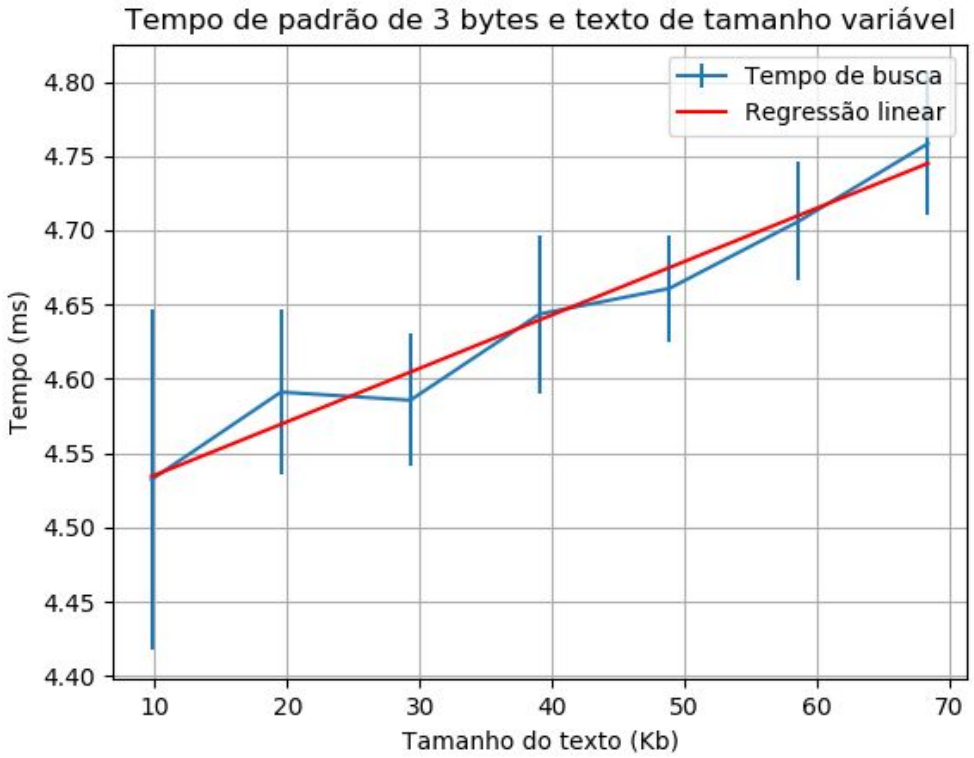

Figura 7 - Tempo de busca com um padrão de tamanho 3 (3 caracteres) e texto de tamanho variável

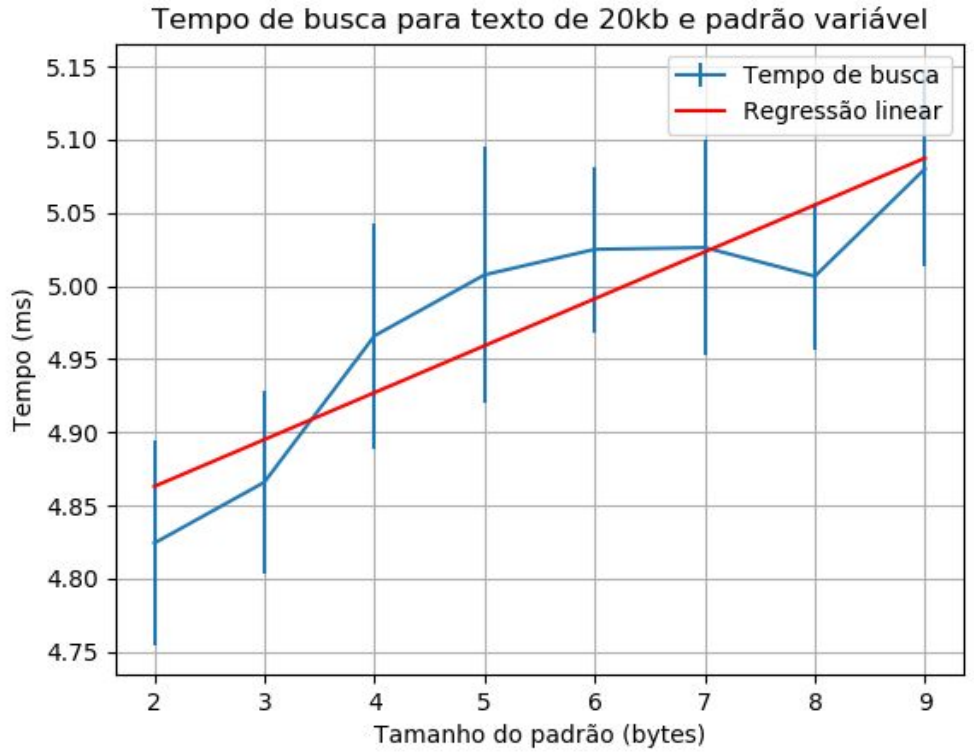

Figura 8 - Tempo de busca com texto de 20 Kb e tamanho do padrão variável

5) Conclusões

Observando os resultados, concluímos da **Figura 1** que em comparação com o tempo de codificação do algoritmo LZ_78, o tempo de execução das outras etapas é quase constante.

Além disso, a partir das figuras **Figura 2**, **Figura 5** e **Figura 6**, nota-se que o desvio padrão das medidas para cada tamanho do texto é pequeno, o que mostra que a tendência linear do tempo da construção do índice, e do tempo da decodificação é consistente.

Ainda assim, o comportamento da codificação assusta. Imaginamos que ele pudesse ser exponencial, mas a **Figura 4** nos mostra uma regressão quadrática que se ajusta perfeitamente a curva. No fim das contas, o LZ_78 teve uma implementação de tempo quadrático no tamanho do padrão. Mas devemos atentar ao fato de que antes do LZ_78, há a construção do índice que tem uma complexidade de espaço que não é linear, podendo ser o culpado do comportamento quadrático. Mesmo assim, a constante de tempo do LZ_78 continua grande. Por conta disso, construímos uma implementação de um string_view (ou string_ref) para armazenar somente referências de partes de strings em vez de criar novas substrings, mas o processo revelou-se ser mais complicado do que parece. Construir um novo container STL requer que respeitemos uma série de padrões. As maiores dificuldades foram em relação a variáveis constantes, ponteiros e operadores de c++. Por fim, não houve tempo o bastante para corrigir todos os problemas do *string_ref*.

Quanto à busca, a **Figura 7** e a **Figura 8** apontam uma relação linear entre tamanho do texto, tamanho do padrão e tempo de busca. Apesar de ser visivelmente linear, dado a reta de regressão, ainda assim temos um alto desvio padrão para cada valor do parâmetro que variou.