Ferramenta IPMT

1. Identificação

A equipe é composta por Bruno de Melo Costa (bmc4) e Renato Henrique Alpes Sampaio (rhas). Bruno implementou os algoritmos de indexação e busca e a função de leitura do texto. Renato implementou os algoritmos de compressão e descompressão. Os scripts de teste e apresentação dos dados, a CLI e o relatório foram feitos por ambos.

2. Implementação

2.1. Descrição da ferramenta

Foram implementados 4 algoritmos: SARR, que se divide em dois componentes: indexação e busca; e o lz77, que se decompõe em compressão e descompressão. O componente de busca do SARR suporta um flag "--count" que faz com que este imprima o número de ocorrências do padrão do arquivo ao invés de imprimir cada linha onde ocorre o padrão ao menos uma vez.

2.1.1. Alfabeto

O alfabeto consiste na tabela completa ASCII. Para a conversão, converte-se diretamente do *char* para o *uint8_t*, podendo utilizar o charactere como um número utilizando-se o melhor desempenho para a conversão (nativo).

2.1.2. Leitura

Para a indexação, o algoritmo lê o texto completo de uma única vez. Para a procura, o algoritmo lê em sequência o tamanho do texto, o vetor de sufixos, os vetores auxiliares *Llcp* e *Rlcp*, a primeira linha da matriz *P* e o vetor de quebra de linha. O algoritmo de compressão e descompressão leem o documento em formato binário, e a quantidade lida é variável dependendo de variáveis do algoritmo.

2.1.3. Detalhamento dos algoritmos

Detalharemos agora partes relativas dos algoritmos.

2.1.3.1. SARR - Indexação

Para o auxilio da construção da matriz *P*, cria-se uma nova estrutura *triple*, que tem comportamento semelhante à *pair*, porém só converte-se ao último elemento e possuem apenas as operações *!=* e <. Possui um vetor (*lineBreakers*) que salva as posições ordenadas das localizações do caractere de quebra de linha, para que seja mais eficiente a extração da linha, caso tenha sido requisitado a sua impressão. A *flag* do *lineBreakers* para determinar o fim do vetor é a inserção de um inteiro de tamanho igual ao do texto.

2.1.3.2. SARR - Busca

Com a implementação de (*lineBreakers*), o algoritmo, ao achar a posição da ocorrência, consegue, a partir de uma busca binária achar o início e fim de uma linha. Além disso, ele reconstrói o texto a partir da primeira linha da matriz *P* gerada em sua parte de Indexação.

2.1.3.2 Lz77 - Compressão

É especificado um tamanho para um buffer de busca e um tamanho para um buffer de lookahead. Então, preenche-se o buffer de lookahead com bytes do texto. Em seguida é realizada uma busca no buffer de busca, procurando a maior string que começa no início de lookahead e que aparece em alguma posição de busca. O algoritmo utilizado nessa busca é uma combinação de shift-or e boyer-moore, utilizando-se shift-or para buscas menores que 7 caracteres (as que ocorrem com maior frequência). A busca é encerrada quando alguma string não é encontrada no buffer de busca. Quando isso ocorre, são colocados no output a

posição onde a string foi encontrada, o tamanho da string e o caractere que causou a falha. Depois disso o processo se repete, avançado os buffers de busca e lookahead pelo tamanho da string encontrada.

2.1.3.2 Lz77 - Descompressão

Inicialmente o buffer de busca é inicializado vazio. Então, são lidos do arquivo dois inteiros: posição e tamanho. É colocada no output a string deste tamanho nesta posição do buffer de busca. Também é lido um caractere e ele é enviado para o output. Em seguida, o buffer de busca é atualizado com a string e o caractere e o processo se repete.

3. Testes e resultados

Foi utilizado um arquivo de texto em inglês (Arquivo de 200MB da fonte mencionada na especificação do projeto). As ferramentas utilizadas para comparação no benchmarking foram grep para casamento exato e gzip para compressão e descompressão. Todos os testes foram realizados com a flag --count.

Os testes foram realizados em um computador rodando Manjaro Linux 5.10.105-1 com CPU Intel Core i5-8250U @ 1.60GHz, memória DDR4 2400 MHz de 8GiB e SSD ST500LM021 de 6Gb/s. O compilador utilizado foi o GCC 11.2.0-4

Todos os scripts shell e python utilizados nos testes estão disponíveis na pasta /test do projeto.

3.1 Indexação:

O algoritmo foi testado em trechos de diferentes tamanhos de um texto em inglês (1Mb, 5Mb, 10Mb, 50Mb) e foram observados o tempo de execução e o tamanho final do índice. Foram feitos 3 testes para cada tamanho do texto, tomando-se a média dos três testes como

resultado final do tamanho. Foi encontrada uma ferramenta candidata a ser usada como comparação (cindex), mas não foi possível encontrá-la nos repositórios do manjaro, no AUR ou em arquivo .tar.gz para instalação manual.

3.2 Busca:

Para o teste de busca foram realizadas duas séries de testes, uma com tamanho de texto variável e uma com tamanho de padrão variável. Para o testo de texto variável foi realizada a busca em índices gerados a partir de trechos de um texto em inglês (1Mb, 5Mb, 10Mb e 50Mb). Foi utilizada a palavra “test” como padrão para a busca e foram feitos 3 testes para cada tamanho do texto, tomando-se a média como resultado final para o tamanho. Foi encontrada uma ferramenta candidata a ser usada como comparação (csearch), mas não foi possível encontrá-la nos repositórios do manjaro, no AUR ou em arquivo .tar.gz para instalação manual. Portanto, a comparação foi feita com a ferramenta grep, sem indexação.

Para o teste de padrão variável, foi realizada a busca em um índice gerado a partir de um texto em inglês de 50MB, com padrões de 1 a 100 caracteres (inclusivo) que podem ser encontrados aqui. Os padrões foram retirados deste arquivo, utilizando-se este script para escolher 3 padrões aleatórios para cada tamanho, e para gerar padrões de tamanho apropriado a partir de padrões já existentes quando não era possível selecionar três padrões do tamanho. Foi realizado um teste para cada padrão (três para cada tamanho), tomando-se a média como resultado final do tamanho. Da mesma forma que no teste anterior, foi utilizada a ferramenta grep sem indexação para comparação.

3.3 Compressão

Inicialmente foi realizado um teste grid search em um arquivo de texto em inglês de 10Mb para definir os melhores valores para o tamanho do buffer de busca e de lookahead. Os resultados podem ser observados na tabela a seguir.

	Busca = 2^8	Busca = 2^{16}
Lookahead = 2^8	0m3,141s / 9,6MiB	4m20,941s / 5,4MiB
Lookahead = 2^{16}	0m14,343s / 6,7MiB	4m26,058 / 12,7MiB

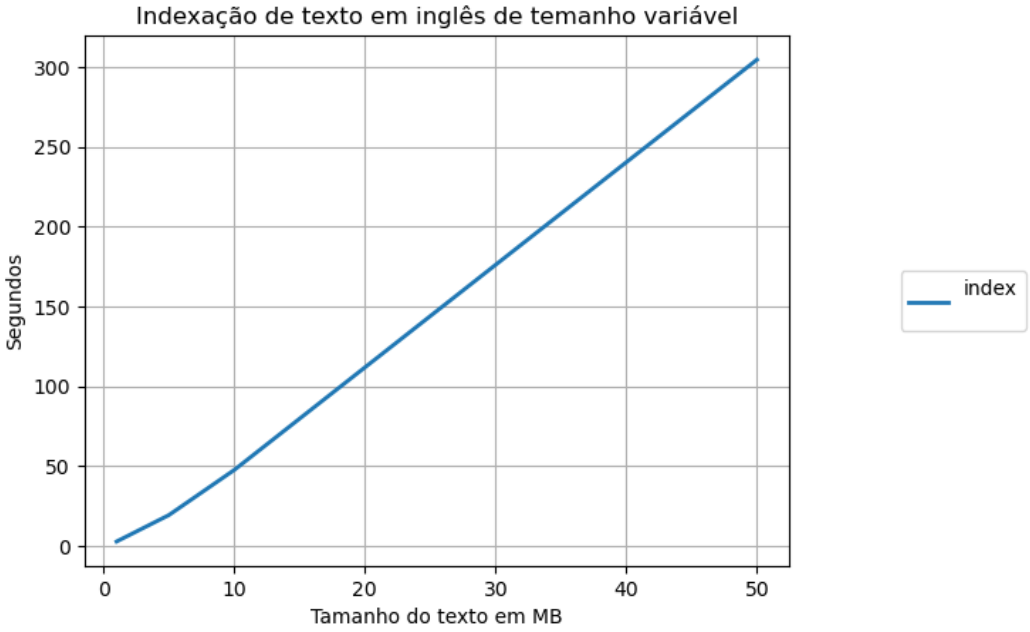
Esses resultados levaram a decisão de utilizar 2^8 como tamanho para os dois buffers, já que aumentar o lookahead sempre piorou o resultado e aumentar a busca aumenta o tempo em um fator igual ao aumento.

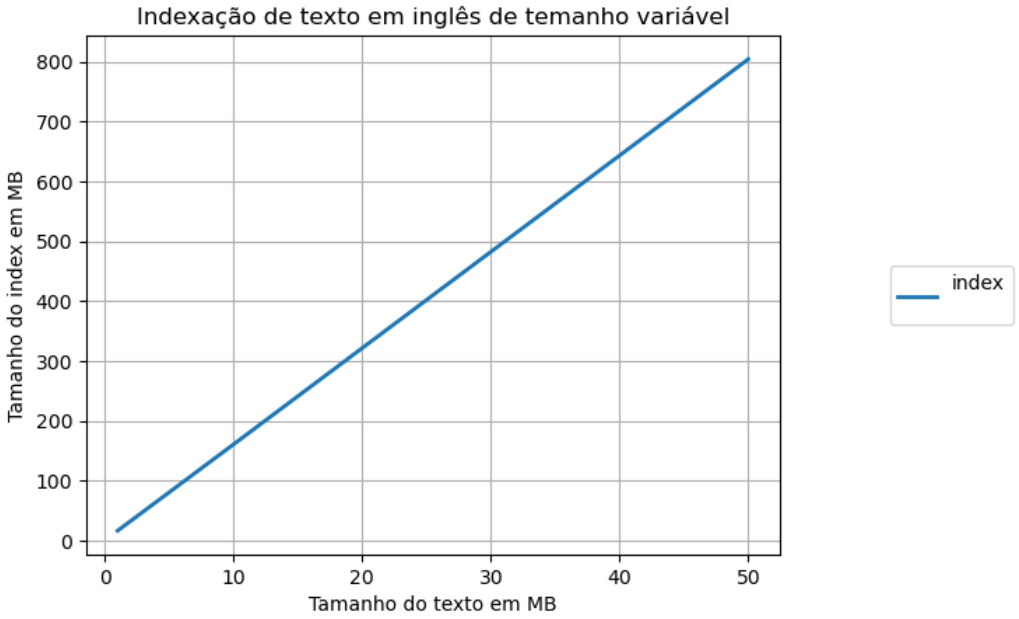
Para o teste de compressão foi realizada uma série de testes em trechos de um texto em inglês de tamanho variável (1Mb, 5Mb, 10Mb, 50Mb e 100Mb) e foram observados o tempo de execução e o tamanho do arquivo comprimido em Mb. Foram feitos três testes para cada tamanho do texto, tomando-se a média como resultado final do tamanho. Foi utilizada a ferramenta gzip (algoritmo deflate) para comparação.

3.4 Descompressão

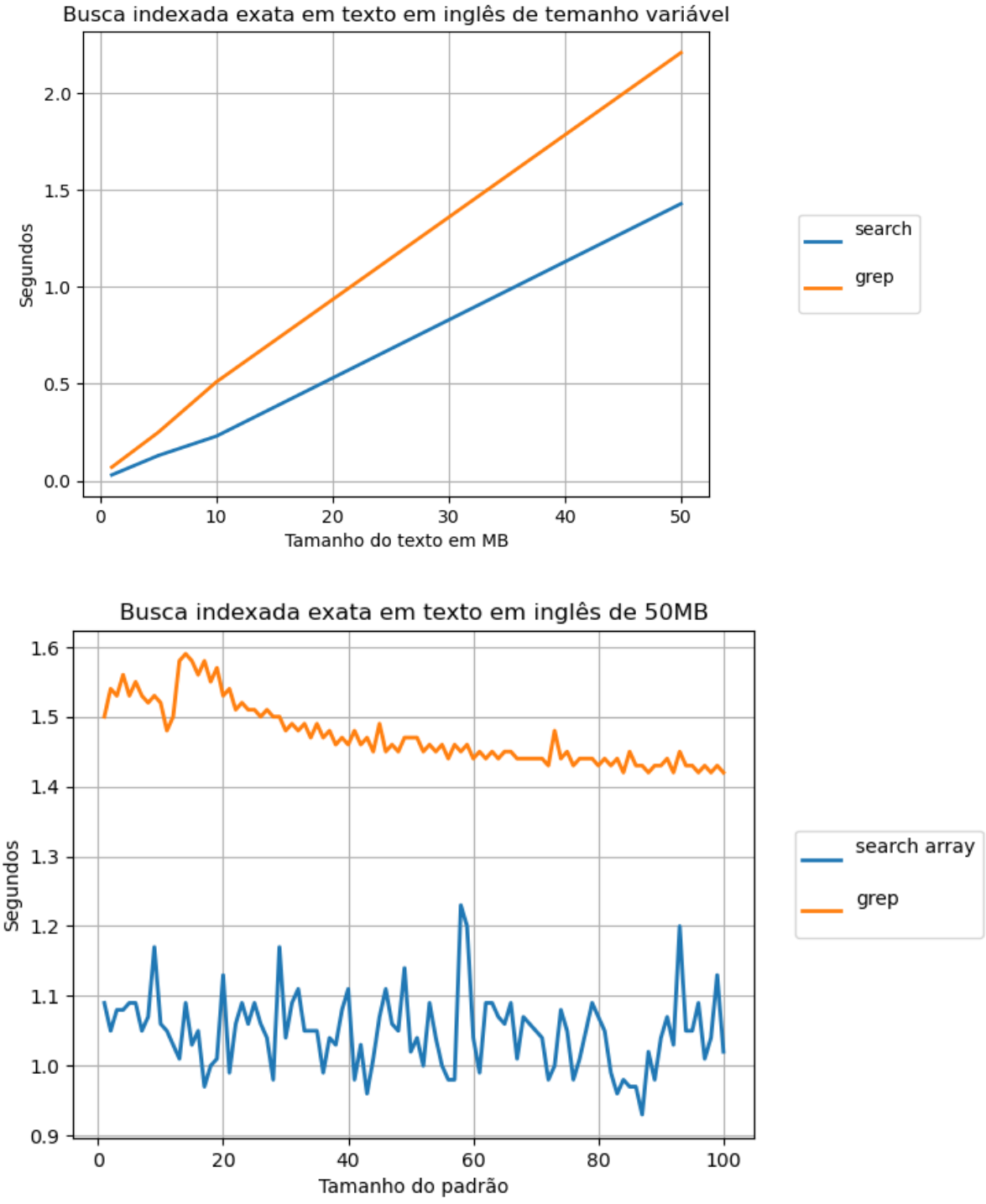
Para o teste de descompressão foi realizada uma série de testes em arquivos comprimidos gerados a partir de trechos de um texto em inglês de tamanho variável (1Mb, 5Mb, 10Mb, 50Mb e 100Mb). Foram feitos três testes para cada tamanho do texto, tomando-se a média como resultado final do tamanho. Foi utilizada a ferramenta gzip (algoritmo deflate) para comparação.

3.3 Resultados e discussão

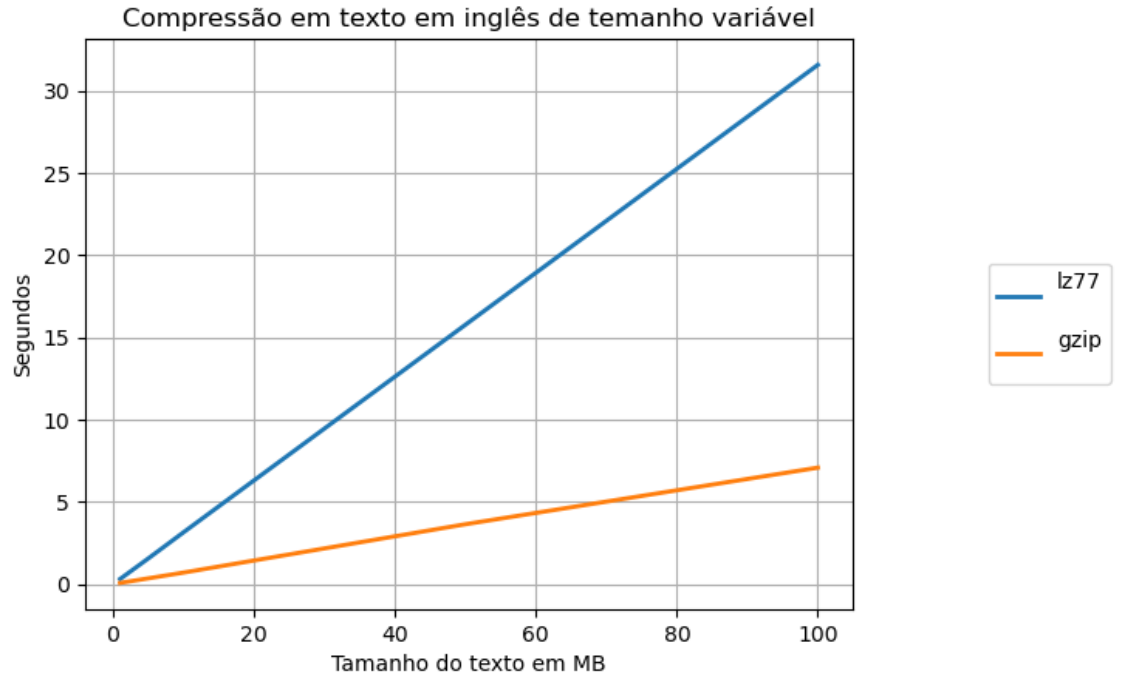


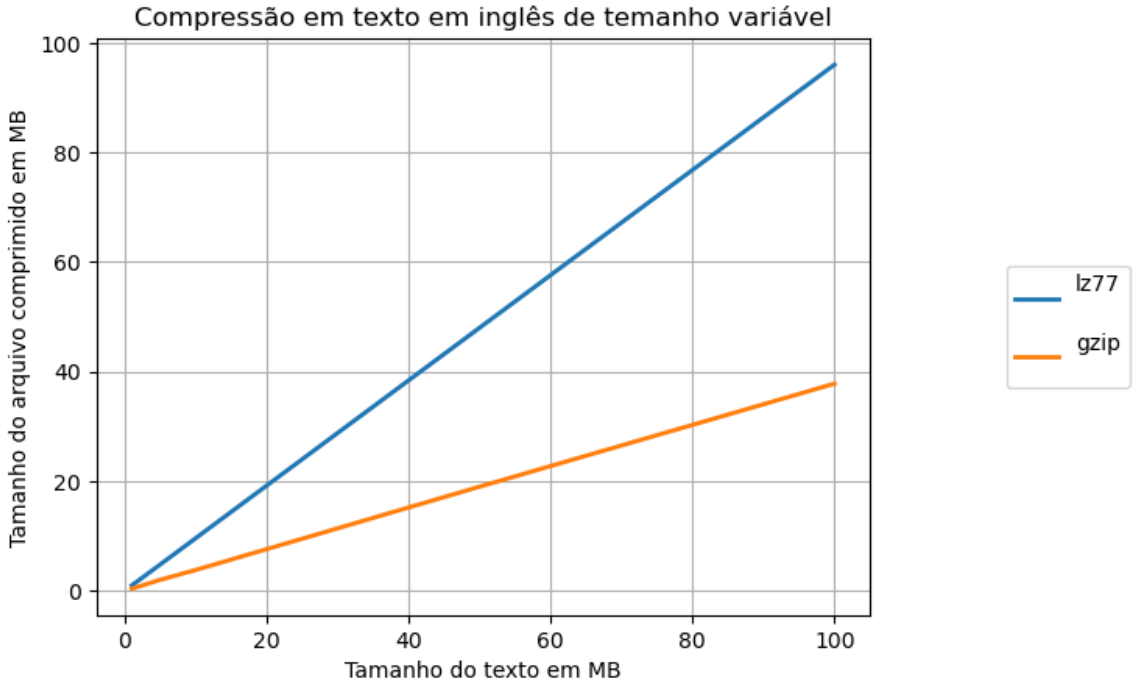


Com esses resultados, é possível perceber que o algoritmo de indexação (SARR) tem uma custo linear em tempo e espaço. Ele possui um grande custo de tempo comparado a buscas sem indexação, porém, posteriormente, é possível ver que esse tempo tem o seu efeito reduzido com o aumento de padrões pesquisados.

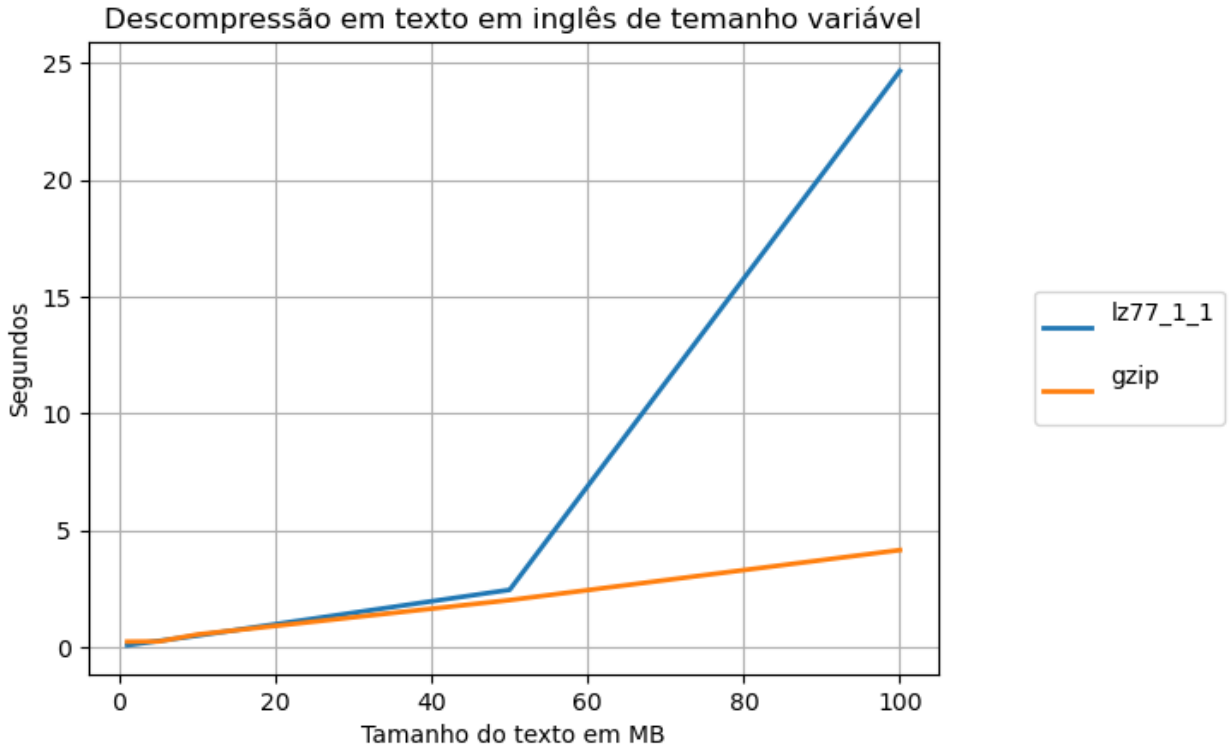


Nesses resultados, podemos ver que, mesmo que o tempo de construção do arquivo de indexação seja grande, há uma grande melhoria no tempo de busca por padrões, o que é ideal para quando se quer buscar vários padrões em um único arquivo de texto.





Nesse teste é possível observar que os dois algoritmos (deflate e lz77) são lineares tanto em tempo de execução quanto em espaço ocupado pelo arquivo comprimido, embora o deflate tenha um poder de compressão muito maior pelo uso de huffman coding no resultado gerado pelo lz77 e na própria árvore gerada pelo huffman.



O algoritmo de descompressão implementado pelo grupo e o utilizado pelo gzip tem tempos de execução muito similares e lineares até 50Mb, no entanto a implementação do grupo de lz77 tem um aumento considerável no tempo para 100Mb. Examinando o dataset é possível observar que este resultado não é causado por um outlier, todos os três resultados estão bem próximos.

4. Conclusões

Para o algoritmo de SARR, não foi possível comparar o resultado com outras ferramentas disponíveis na internet, porém, mesmo que tenha uma demora no processo de indexação, a busca de padrões se mostrou ser mais rápida do que o grep. Logo, a utilização do algoritmo de SARR mostra-se mais eficiente se o usuário pretende realizar várias buscar por padrões em um texto ao invés de uma única procura.

O lz77 se comporta linearmente da forma esperada e de forma também esperada, é pior que a implementação do deflate utilizada pelo gzip, uma vez que essa implementação se utiliza do lz77 no primeiro passo de sua compressão. Infelizmente não foi possível implementar no tempo definido o algoritmo huffman para ser possível o uso do deflate para uma melhor comparação.