Ferramenta pmt

1. Identificação

A equipe é formada pelos alunos André Luiz Pereira da Silva e Lucas Vinícius Araújo da Silva.
As tarefas foram divididas da seguinte forma:
- André Luiz: Implementação da ferramenta de compressão e descompressão.
- Lucas Vinicius: Implementação dos algoritmos de busca indexada.
Ambos contribuíram de forma igualitária para a CLI e testes, assim como a construção do relatório.

2. Implementação

2.1 Algoritmos de indexação e casamento exato

O algoritmo escolhido para indexação foi o SA-IS, que faz o uso de induced sorting, e para o casamento exato o algoritmo Burrows-Wheeler Transform (BWT) foi escolhido.

2.2 Algoritmos de compressão e descompressão

O algoritmo escolhido para o compressão e descompressão foi a codificação de Huffman. Utilizamos uma estrutura de árvore binária para determinar um código binário para cada caractere do texto, dependendo da frequência em que aparecem.

2.3 Detalhes de implementação Relevantes

Os dois algoritmos utilizados para para indexação e casamento foram as formas ingênuas que aparecem na literatura, quase nada foi feito para limitar o uso de memória utilizada na criação do array de sufixo e das tabelas, além de mapear a quantidade de caracteres no alfabeto para um número exato que aparece no texto, em vez de utilizar os 256 caracteres do Latin 1. O algoritmo BWT não foi utilizado para comprimir o texto, o texto original está dentro do arquivo salvo.

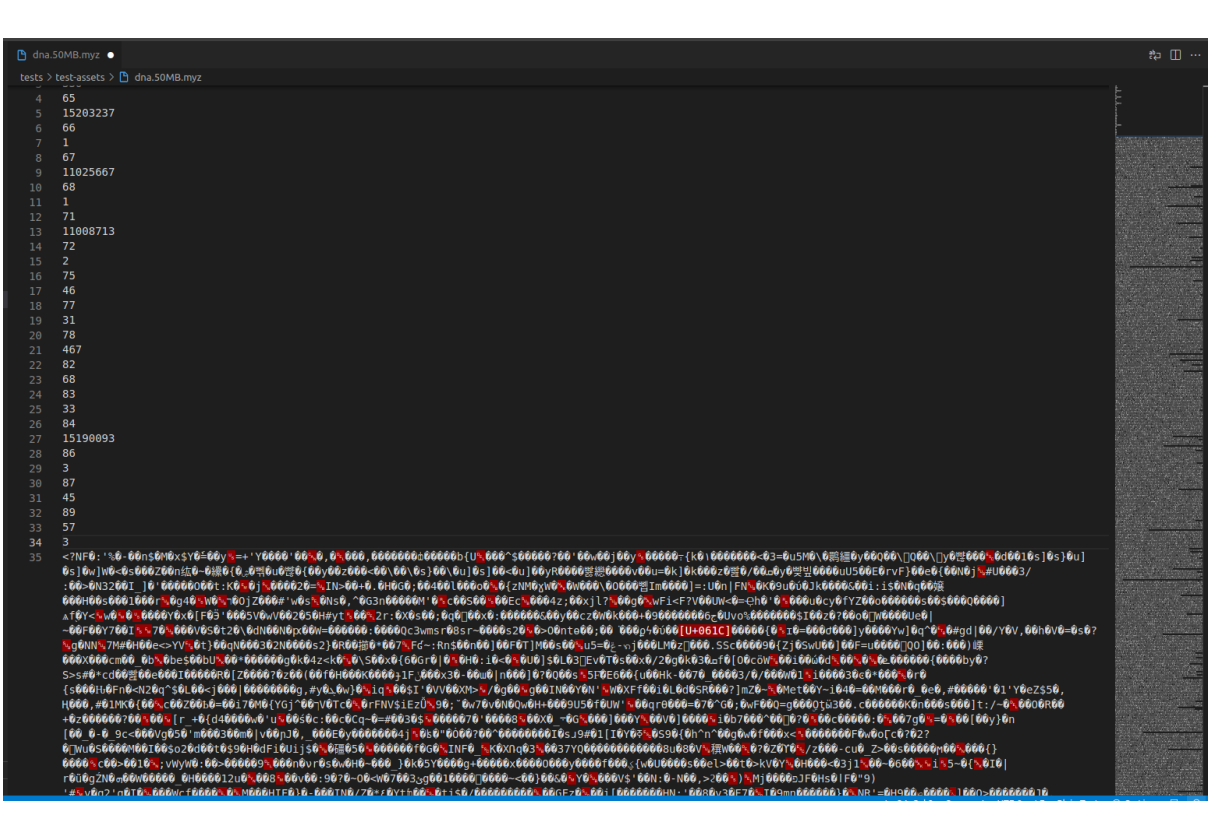
2 tabelas foram criadas no algoritmo BWT:
1. A primeira é uma tabela de caracteres, ordenada lexicograficamente.
2. A segunda é uma tabela que guarda a quantidade de ocorrências de um carácter no texto, em cada posição, até o final, e para isso é necessário um array do tamanho do texto para cada caracter no alfabeto, o que deixa o custo de memória muito grande para textos com alfabetos grandes.

Com essa implementação há uma escolha entre espaço utilizado e a velocidade do algoritmo, o algoritmo SA-IS é um dos mais rápidos da literatura para a construção do array de sufixo, com uma complexidade de O(n) para a sua velocidade, e o algoritmo BWT também trabalha com uma complexidade para a velocidade O(n), o que fica aparente é o uso de memória necessário para a construção da estrutura de dados do BWT, que precisa de [*caracteres únicos * tamanho do array de sufixo * sizeof(uint32_t)*] de memória para poder armazenar todos os detalhes relevantes para a busca exata.

Na indexação o array de sufixo e as tabelas são pré processadas e guardadas no arquivo.

Para compressão e descompressão, optamos trabalhar com um arquivo que registra as seguintes informações:
1. Cada caractere que aparece no texto ao menos uma vez e sua frequência: Essas informações foram registradas de forma mais simples e objetiva: Na primeira linha do arquivo guardamos o número de caracteres e frequências que virão em sequência, para que pudéssemos iterar nas linhas. Após isso, cada par de linhas subsequentes possui um número, que é lido como o código ASCII de um caractere, e outro número inteiro que é lido como a frequência do caractere lido na linha anterior. Cada uma dessas informações custa ao menos 1 Byte para ser armazenada.
2. Uma sequência de bits que representa o texto original codificado pela codificação de Huffman gerada. Aqui, escolhemos guardar cada ‘0’ ou ‘1’ como um bit de fato, e não como um caractere. Obviamente, se usássemos o formato do caractere para essa informação, todos os textos comprimidos acabariam com um tamanho ainda maior do que o texto original. Dito isso, o C++ exige que a unidade básica de informação seja um byte, então fomos acumulando bit por bit, até termos um conjunto de 8 bits, 1 Byte, e só aí inserimos esse byte no arquivo comprimido. Pensando nos casos em que há um “resto” de bits, insuficiente de formar um conjunto de 1 Byte, resolvemos também incluir, antes da sequência de bits, um número que indica quantos bits esse “resto” possui, assim, guardamos o “resto” num Byte e lemos ele até esse valor previamente informado.

No final, o arquivo fica parecido com algo do tipo:



É possível notar que a partir da linha 35 temos a sequência de bits. Às vezes os Bytes gerados significam algo em UTF-8 e são lidos dessa forma, mas a maior parte das vezes podemos ver símbolos vermelhos e de “?”, indicando que a “encoding” não conseguiu interpretar o Byte. Mas para nós, isso não faz diferença, pois o que importa é o conjunto inteiro de bits.

Para comprimir o arquivo, fazemos o cálculo das frequências dos caracteres, e usamos essa informação para gerar a árvore de Huffman. Dessa forma, atribuímos códigos menores para caracteres mais frequentes, e códigos maiores para caracteres menos frequentes.

Para descomprimir, retiramos do arquivo a informação das frequências dos caracteres. Se mantermos a mesma ordem das frequências, a árvore de Huffman gerada vai ser igual aquela gerada na etapa de compressão, então evitamos de serializar a árvore inteira. Com essa tabela de frequências, geramos a mesma árvore de Huffman novamente, e agora podemos usar ela para decodificar a sequência de bits presente no arquivo. Na etapa de testes, entramos em mais detalhes sobre a descompressão e como fizemos para deixá-la mais eficiente.

3. Testes e resultados

Indexação e busca exata

Para os testes de Indexação e busca exata foram utilizados 3 tipos de textos: DNA, Inglês e o de Shakespeare na pasta demo do github da cadeira. Apenas no texto de DNA foi possível utilizar um arquivo acima de 50MB, indo até 100MB, por ter um alfabeto pequeno ele foi o único que não ultrapassou a memória do computador de teste.

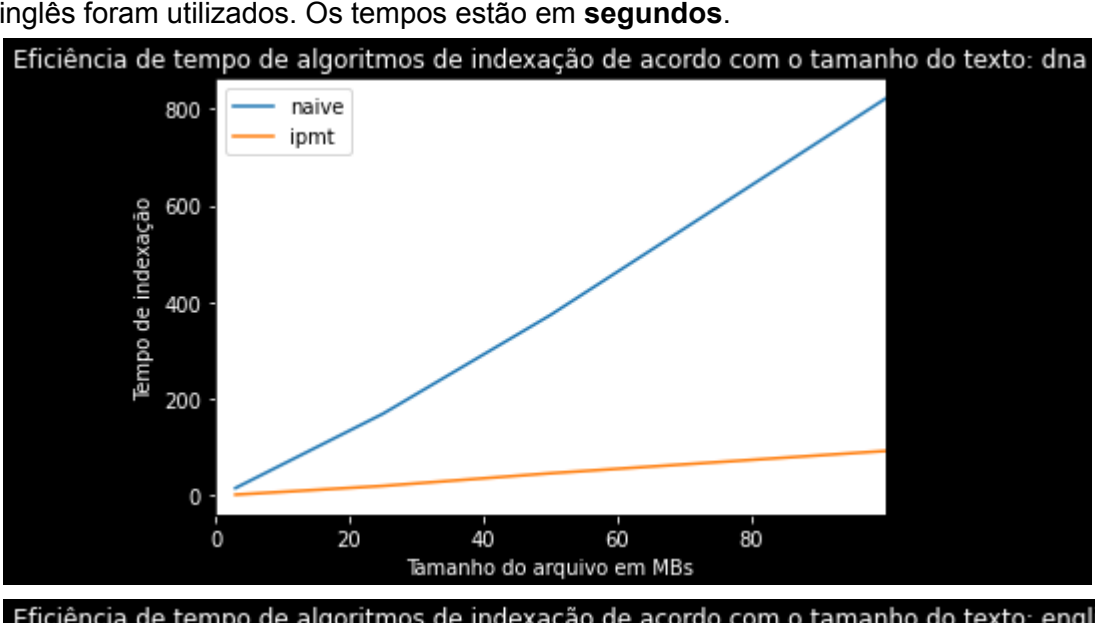
Compressão e descompressão

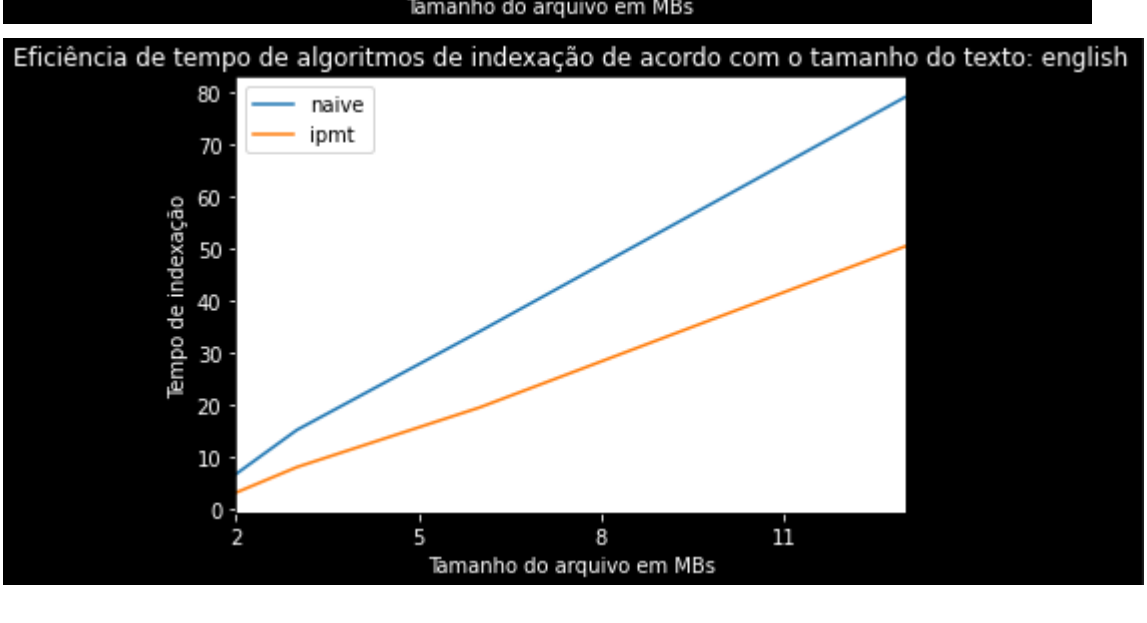
Para os testes de compressão e descompressão, foram usados 3 tipos de textos: DNA, Inglês e código-fonte. Dentro de cada uma dessas categorias, foram usados arquivos de 50MB, 100MB e 200MB.

Realizamos os testes via Python, usando a biblioteca subprocesses para fazer chamadas a programas via linha de comando por meio do script python. Foram marcados o tempo do sistema antes de fazer a chamada ao ipmt, grep e zip, e depois de executarem. Também utilizamos funções do python para registrar o tamanho dos arquivos. Para os gráficos, foi usada a biblioteca matplotlib, relacionando tempos de execução ou tamanho de arquivos e o fator variável dos testes (tamanho do padrão, tipo de arquivo e tamanho do arquivo).

3.1 Indexação

Para a indexação foram testados 2 códigos, o código antigo, feito a partir do código visto em aula, utilizando um algoritmo de ordenação com comparações (std::sort em vez de radix sort) contra o algoritmo SA-IS. As duas coisas que podem impactar em algo do teste é o tamanho do arquivo, para os dois algoritmos, e o tamanho do alfabeto, para o algoritmo SA-IS+BWT, portanto apenas os arquivos de DNA e o de inglês foram utilizados. Os tempos estão em **segundos**.



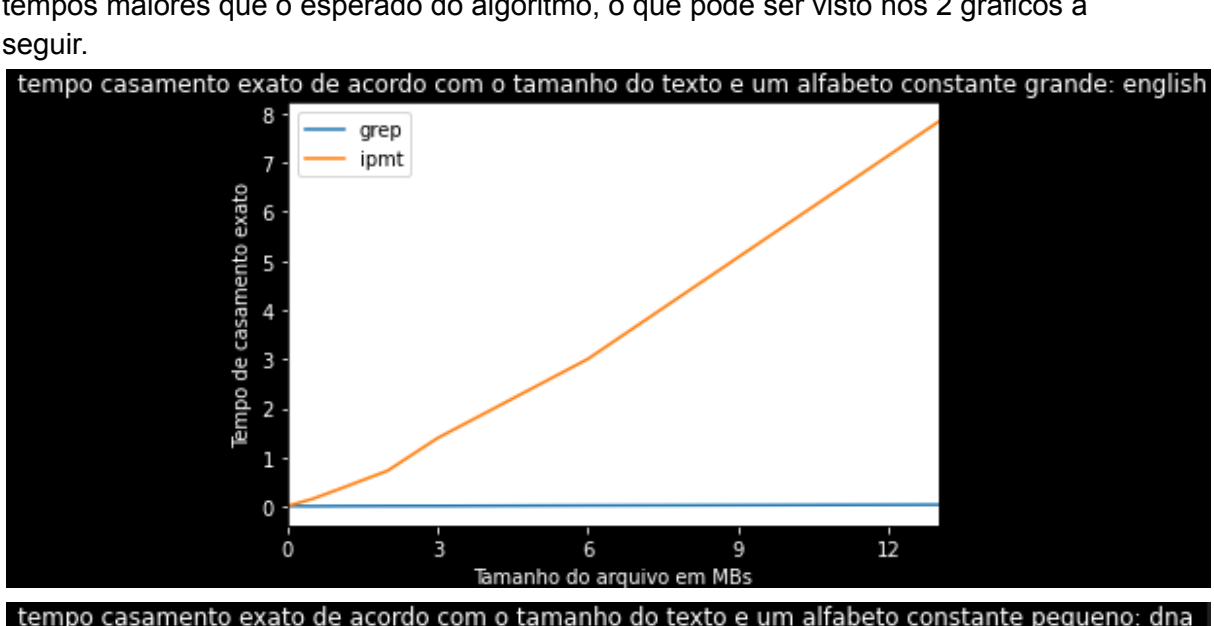


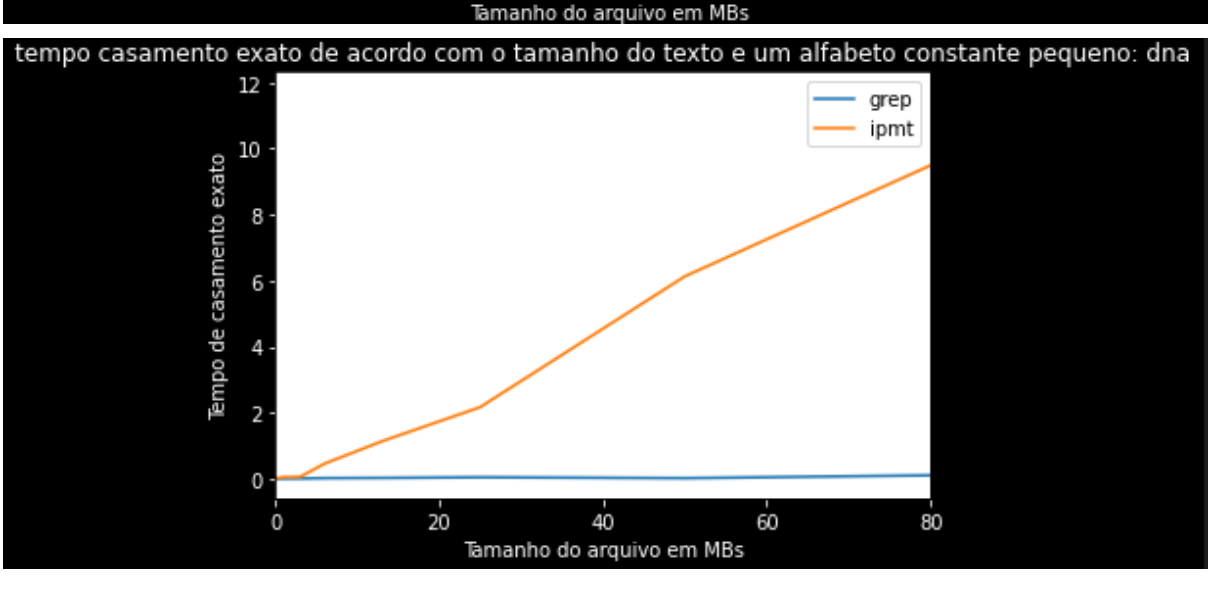
O resultado no teste do DNA é o esperado, em um alfabeto pequeno o SAIS+BWT é bem mais eficiente que o algoritmo ingênuo. O ponto interessante de se ver é que no teste dos textos de inglês a maior parte do tempo ocorreu no algoritmo BWT, mostrando que mesmo que o algoritmo BWT tenha uma complexidade bem mais baixa que o algoritmo ingênuo, em um alfabeto com algoritmo grande as constantes do algoritmo ficam bem mais pesadas que o esperado.

3.2 Casamento Indexado

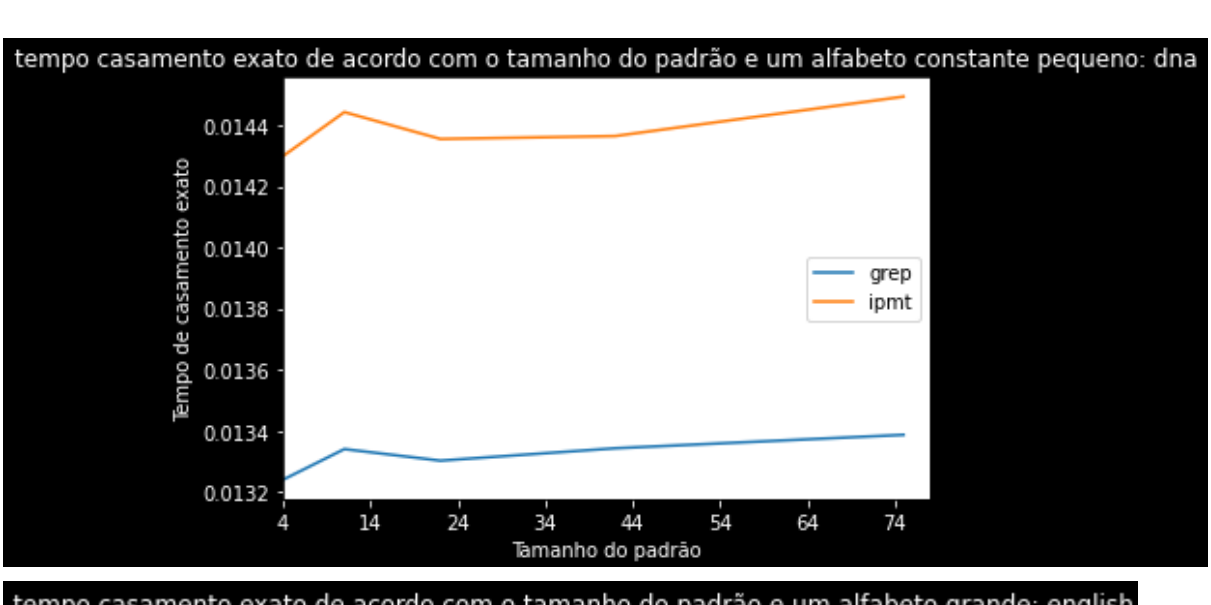
Para o casamento exato, 3 coisas que podem influenciar:
1. O tamanho do alfabeto do arquivo.
2. O tamanho do padrão.
3. O tamanho do texto do arquivo.

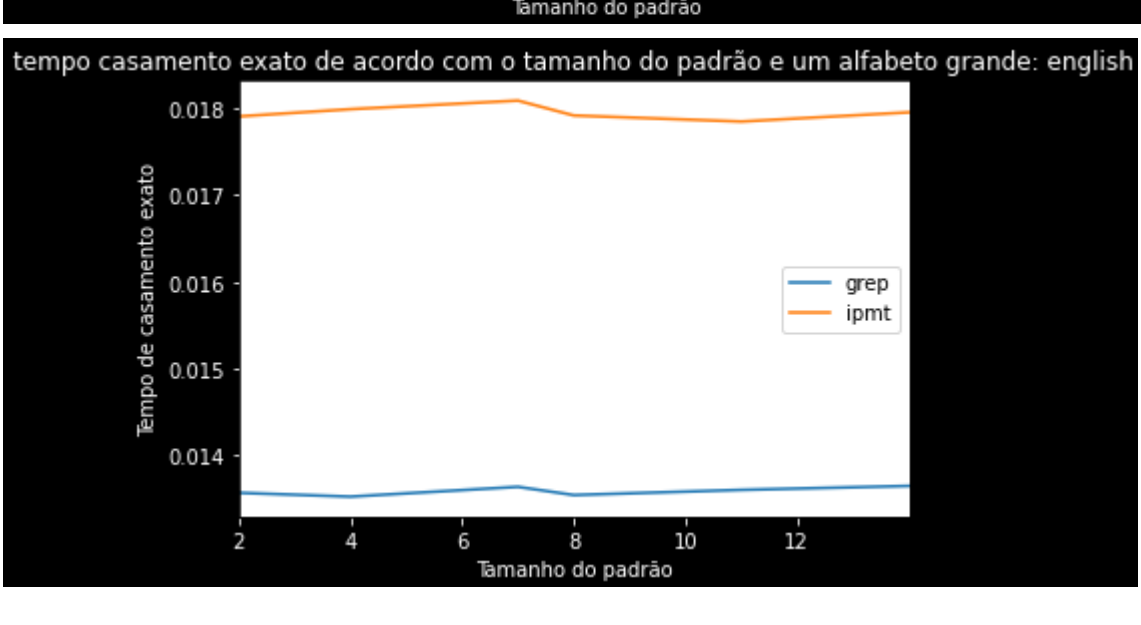
Para o ponto 1 e 3, os arquivos que o BWT tem que carregar está em ordem de magnitudes maiores que os arquivos que o Grep tem que carregar, causando tempos maiores que o esperado do algoritmo, o que pode ser visto nos 2 gráficos a seguir.





Quanto maior o texto, maior o tempo para carregar, e quanto maior o alfabeto mais rápido a duração de execução aumenta.





Esse teste foi rodado 1000 vezes e finalmente foi feito uma média dos resultados, dá para ver que há uma mudança não significativa com o aumento do tamanho dos padrões.

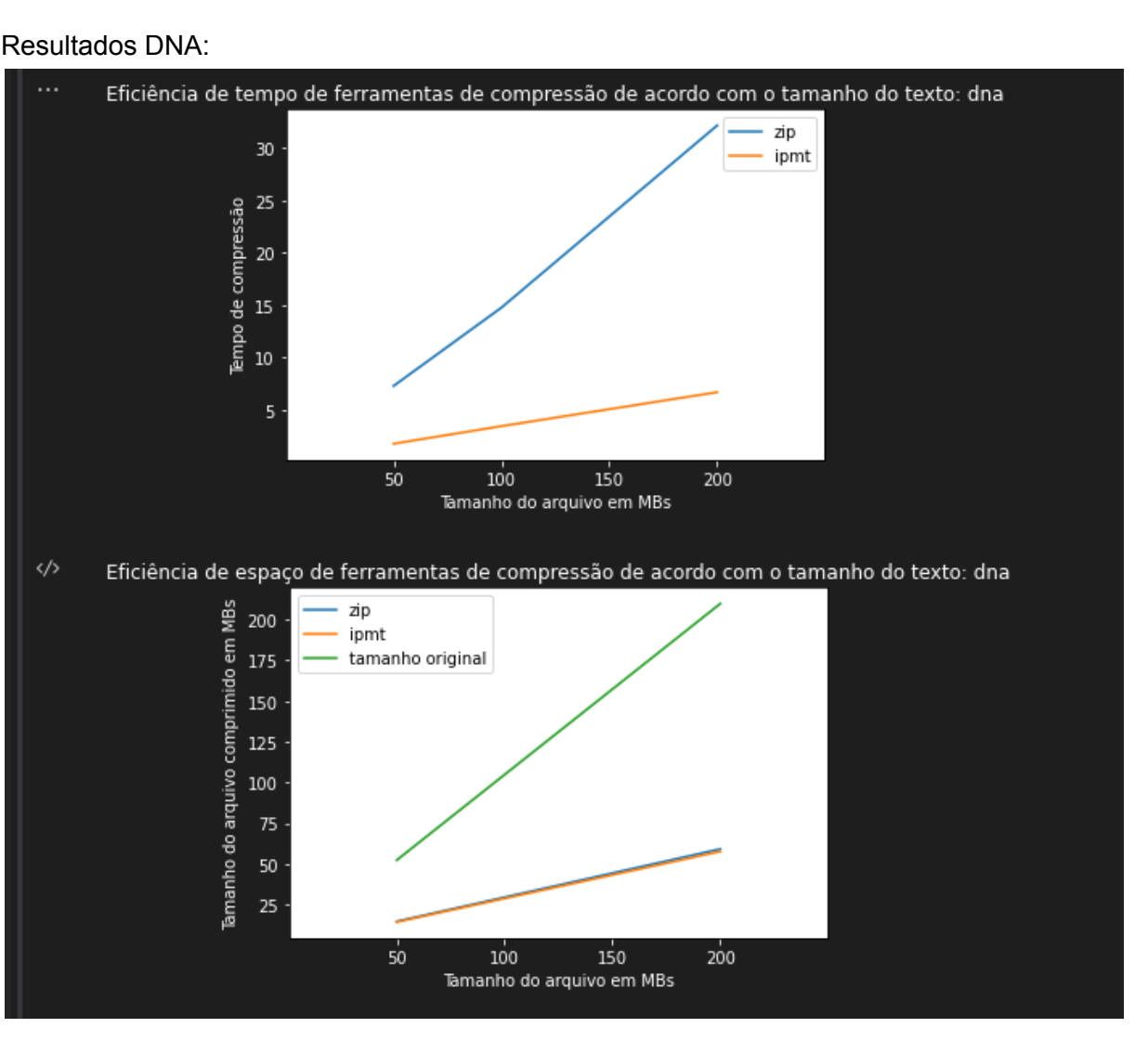
Nesses testes os resultados para o array de sufixos não está tão bem quanto esperado, o motivo que foi visto para isso é por conta do tamanho dos arquivos que foram indexados. Ao fazer um teste onde apenas pegamos o tempo de pesquisa do padrão dentro do texto, com 70 padrões diferentes e 10000 iterações de cada pesquisa, obtivemos a seguinte média:

```
matching 70 patterns, median elapsed time is: 3 microseconds
```

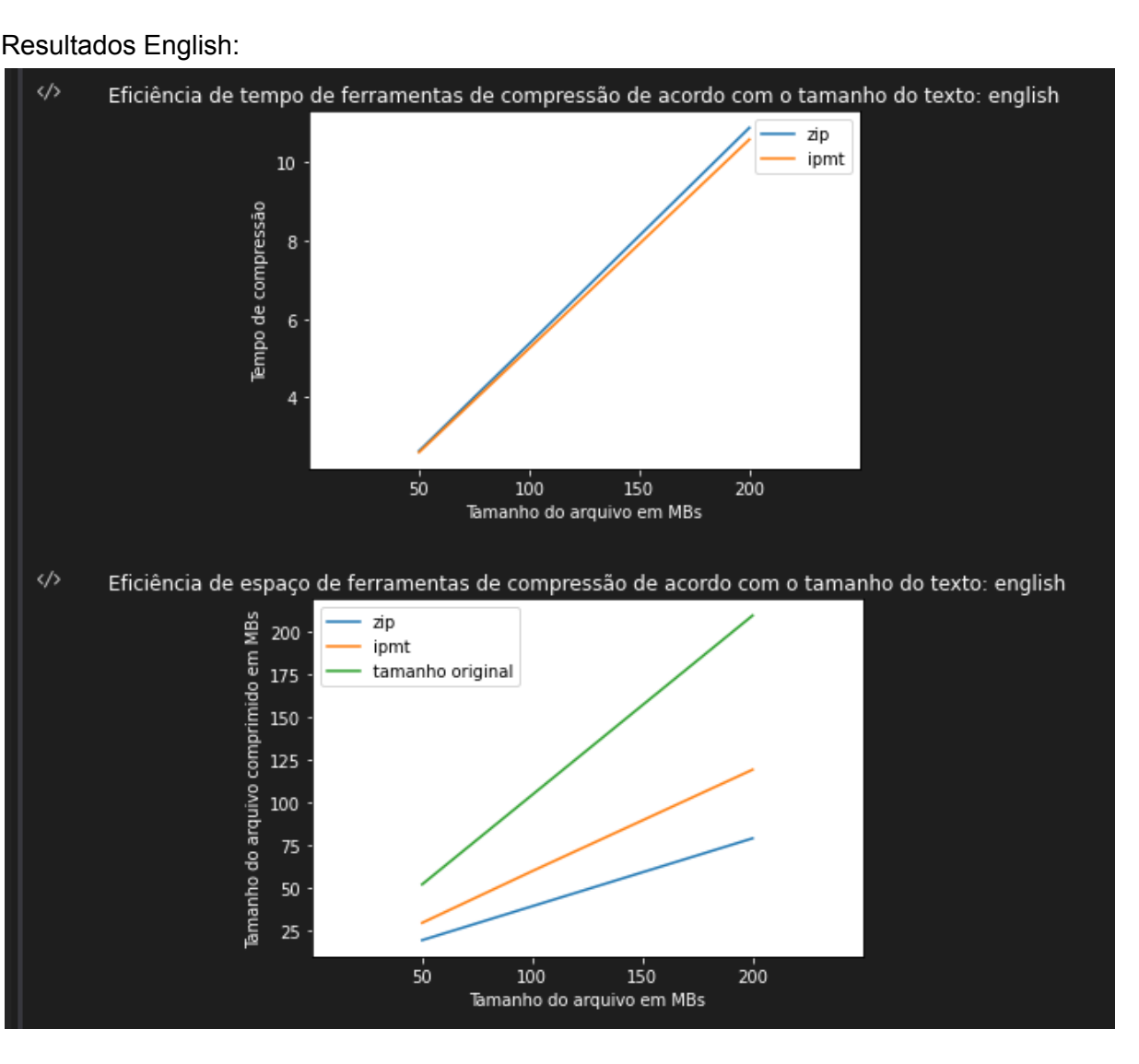
3.3 Compressão

Para os testes de compressão, analisamos a velocidade em que os programas conseguem compactar os arquivos de texto, assim como o tamanho final do arquivo compacto. Em quesito de velocidade, comparamos a eficiência do ipmt com a ferramenta “zip”, presente no linux. Quanto ao espaço, além do zip, incluímos também o tamanho original dos arquivos comprimidos para melhor visualização do processo de compressão.

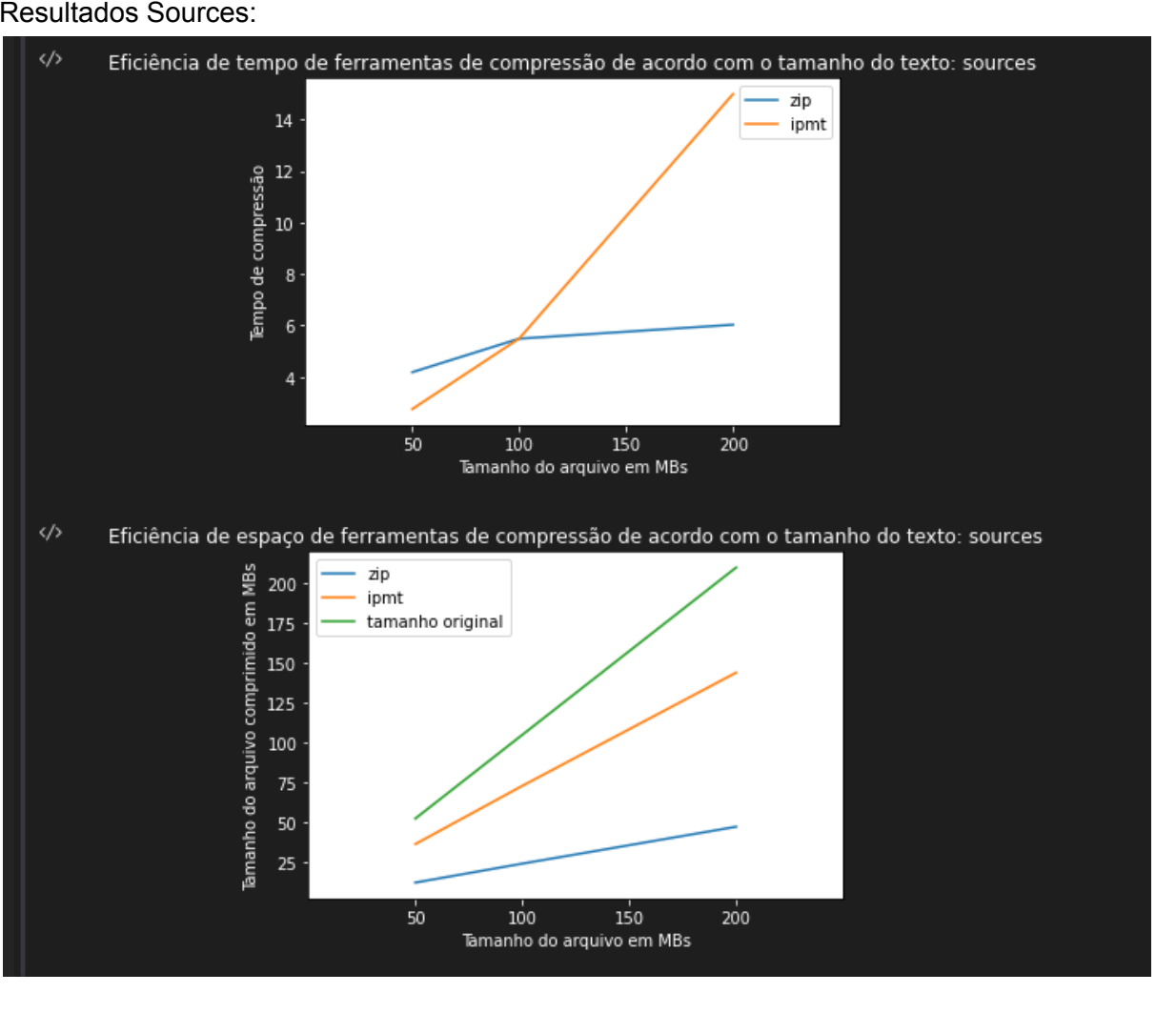
Resultados DNA:



Resultados English:

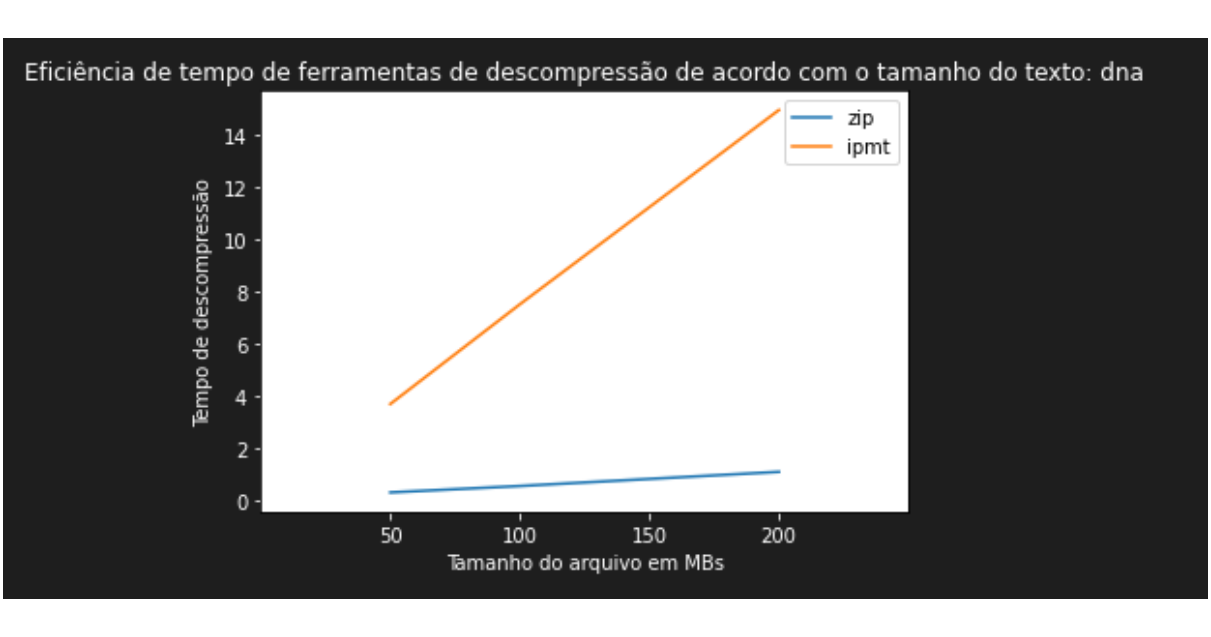


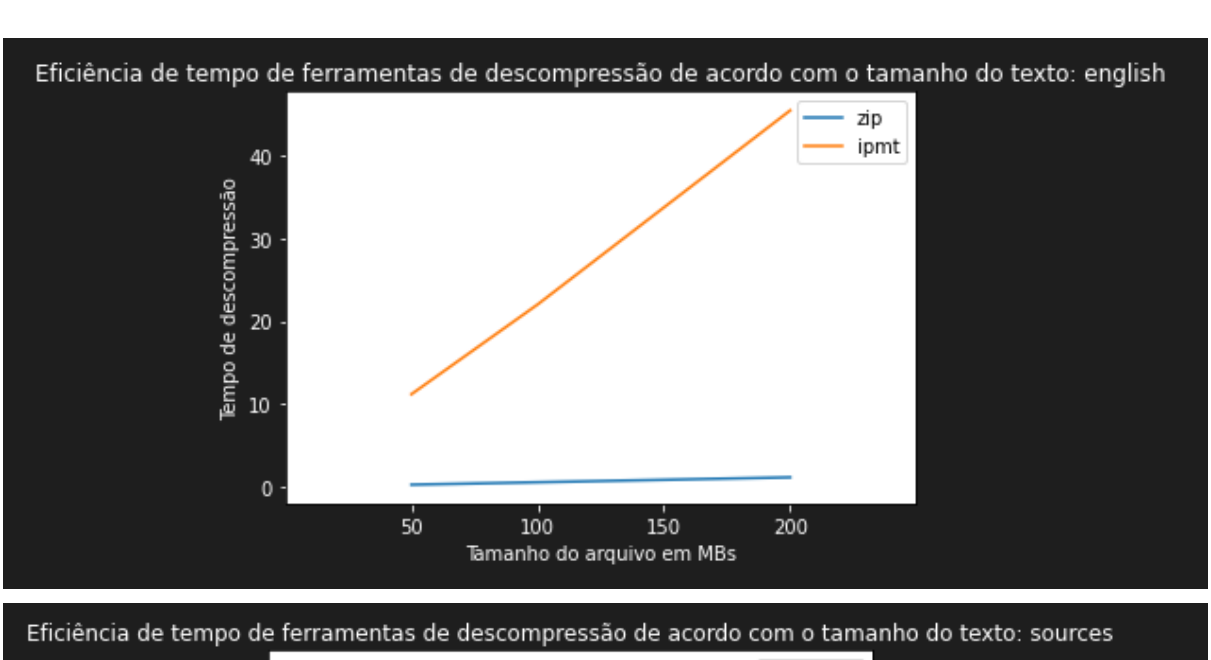
Resultados Sources:

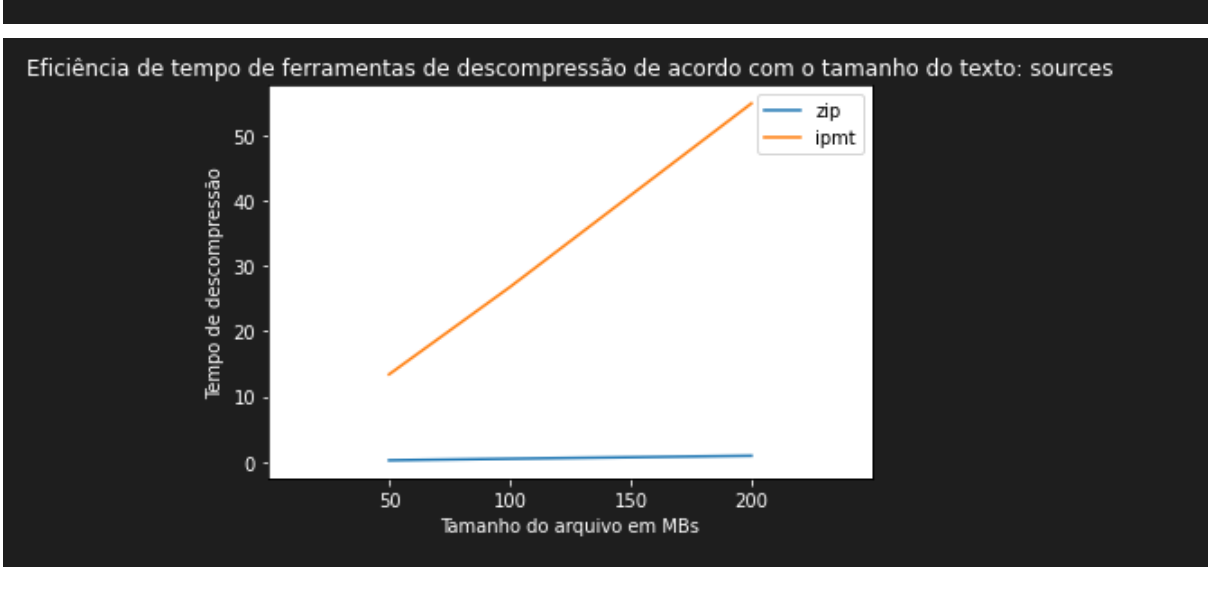


Conseguimos resultados relativamente próximos ao do zip, em especial nos tipos de texto “DNA”, que possuem um menor alfabeto, portanto menores códigos de Huffman. Acreditamos também, que, já que o zip funciona com vários tipos de arquivos diferentes, ele possui maiores overheads, o que nos permitiu alcançá-lo em algumas situações. Dito isso, a velocidade ainda deixa a desejar e, novamente, acreditamos que a maior parte do tempo excedente de execução se concentra nas operações de entrada e saída.

3.4 Descompressão







Aqui, já podemos ver um resultado bem melhor para zip. Originalmente, o algoritmo de descompressão navegava pela árvore de huffman para decodificar cada caractere, mas essas operações eram muito custosas, fazendo com que uma simples descompressão de 200MB demorasse horas. Assim, resolvemos reformular a maneira com que decodificamos os códigos. Com apenas um pré-processamento da árvore, fomos capazes de gerar um tabela de códigos que relaciona cada caractere com seu devido código, em formato de dicionário, o que tornou a execução da decodificação bem mais eficiente e possível de ser usada em tempo hábil. Ainda assim, dada a operação de varrer o texto codificado, adicionando o próximo bit ao código corrente e checando para ver se ele existe no dicionário, ainda não conseguimos equipar a eficiência dessa etapa aquela do unzip.