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 dos algoritmos Boyer-Moore e Sellers.
- Lucas Vinicius: Implementação dos algoritmos Shift-Or, Wu-Manber e o MakeFile.
Ambos contribuíram de forma igualitária para a CLI e demais módulos necessários, assim como a construção do relatório.

2. Implementação

2.1 Algoritmos de Casamento Exato

Os algoritmos escolhidos para o casamento exato foram:

- Boyer-Moore: Eficiente e considerado benchmark para processamento de cadeias de caracteres.
- Shift-Or: Bastante eficiente se o padrão não for maior que o tamanho de palavra padrão da máquina.

2.2 Algoritmos de Casamento Aproximado

Os algoritmos escolhidos para o casamento aproximado foram:

- Sellers: Implementado em cima do algoritmo de cálculo de distância de Levenshtein, de forma iterativa, preenchendo uma matriz de dimensões (tamanho do padrão x tamanho do texto). Foi considerado que utilizar essa matriz completa, ao invés das versões que precisam apenas armazenar as duas últimas linhas da matriz, seria interessante caso quiséssemos observar de onde os matches estavam vindo.

- Wu-Manber: Extensão do algoritmo Shift-Or e base do agrep.

2.3 Detalhes de implementação Relevantes

Recebemos o conjunto dos padrões como uma lista. Mesmo que seja apenas um padrão passado, esse é o comportamento. No caso do usuário não especificar um algoritmo específico que ele deseja, dado cada padrão da lista, analisamos seu tamanho para determinar qual o melhor algoritmo para essa situação: Devido às características do Shift-Or e Wu-Manber, se detectamos que o padrão tenha 8 ou menos caracteres, esses serão os algoritmos utilizados. Caso contrário, Boyer-Moore e Sellers serão utilizados.

Determinamos se uma busca é exata ou aproximada por meio da opção eMax (Custo máximo de edição): Se for 0 (valor padrão, caso o usuário não passe essa opção), é uma busca exata. Caso seja maior que 0, é uma busca aproximada.

No algoritmo de Sellers, foi considerado um custo de substituição de **1**.

O arquivo makefile foi feito de forma que ao usuário dar o comando “make”, o makefile compila o projeto e o insere na pasta bin e também cria um hard link para a pasta /usr/local/bin do linux, fazendo com que seja necessário apenas chamar o nome “pmt”, junto com a lista de opções opcionais e obrigatórias. O makefile também tem a opção de comando “make clean”, que remove esse arquivo na pasta bin e em sequência também remove o hard link criado.

A estrutura de dados que tivemos um cuidado maior na nossa implementação foi o std::vector. Em algumas partes do código é possível saber qual será a capacidade final do std::vector, na primeira versão do projeto essa estrutura sempre era definida com a capacidade default, que é zero, o que causou uma queda de performance muito grande por conta do overhead de alocações da capacidade. Isso foi resolvido ao implementar um tamanho inicial nos std::vector que eram usados em partes críticas do código.

3. Testes e resultados

Para o benchmark, determinamos duas formas de analisar a performance:

1. Tamanho do texto fixo e tamanho do padrão crescente.
2. Tamanho do texto crescente e tamanho do padrão fixo.

Em ambos os benchmarks, foram usados os argumentos -c, para o grep, e -l para os algoritmos de busca exata do pmt. Ambos tem como objetivo retornar o número de linhas onde houve ocorrência do padrão. Para o agrep e algoritmos de busca aproximada do pmt, foram usados o -c, que conta o número exato de ocorrências do padrão dentro de um custo de edição máximo. Decidimos fazer esses benchmarks com funções separadas pois o agrep não possui um argumento semelhante ao --count do grep.

Para o primeiro benchmark, determinamos um “pattern_source”:

GATCAATGAGGTGGACACCAGAGGCGGGGACTTGTAAATAACACTGGGCTGTAG
GAGTGATGGGGTTCACCTCTAATTCTAAGATGGCTAGATAATGCATCTTTCAGGGT
TGTGCTTCTATCTAGAAGGTAGAGCTGTGGTCGTTCAATAAAAGTCCTCAAGAGG
TTGGTTAATACGCATGTTTAATAGTACAGTATGGTGACTAT

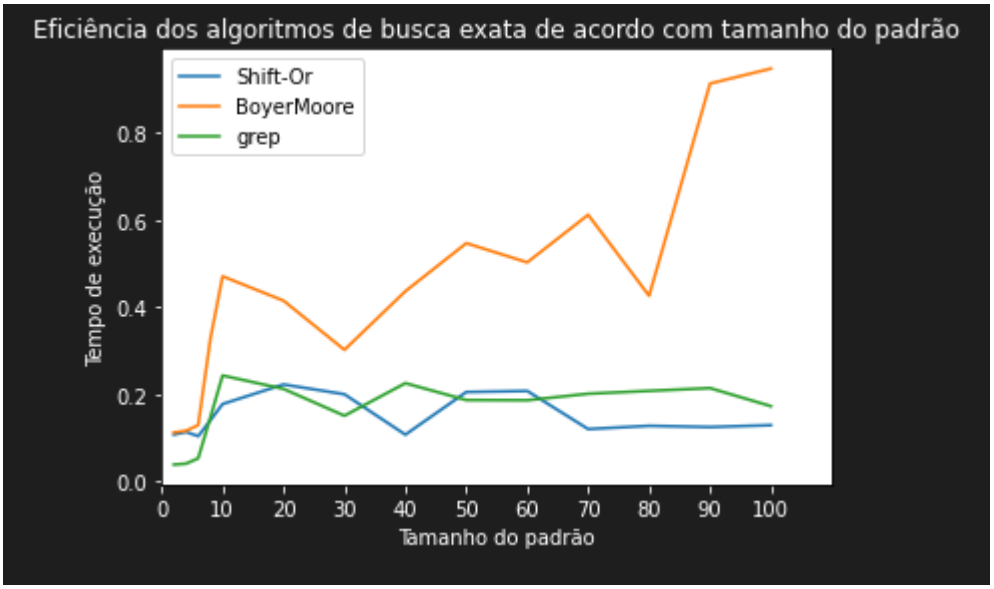
Rodamos os algoritmos de busca implementados e o grep e agrep em looping, com patterns cada vez maiores (intervalo entre 2 e 100 caracteres) proveniente do pattern_source, e buscando no texto dna.200mb. Registramos os tempos decorridos para cada busca e plotamos gráficos para melhorar a visualização.

Já para o segundo, mantemos a ideia da sequência genética, mas usando os arquivos de 50mb, 100mb, 200mb e 400mb presentes no repositório. O padrão fixo escolhido foi o subtexto composto pelos 8 caracteres iniciais do pattern_source: “GATCAATG”.

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 pmt, grep ou agrep, e depois de executarem. Para os gráficos, foi usada a biblioteca matplotlib, relacionando tempos de execução e o fator variável dos testes (tamanho do padrão ou tamanho do arquivo).

3.1 Casamento Exato

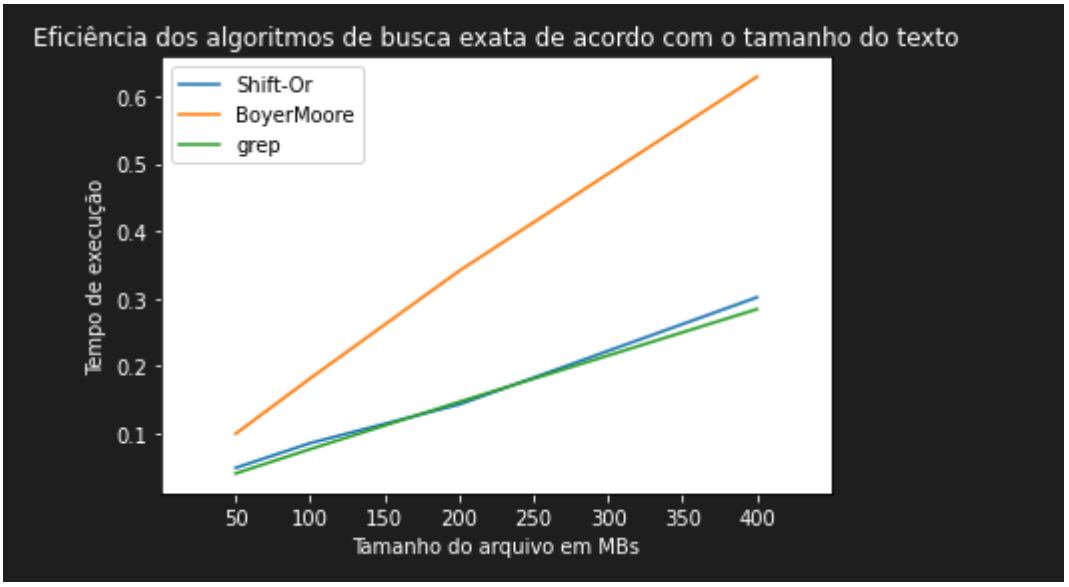
Primeiro benchmark:



Observação: O Shift-Or só realiza a busca se o padrão for menor ou igual a 64

caracteres. Acima disso, ele só emite um aviso de que o algoritmo não é eficiente para esse tamanho de padrão e solicitando ao usuário que tente outro. É por isso que, no gráfico, próximo do tamanho 70, o tempo de execução do shift-or parece ter diminuído bastante.

Segundo benchmark:



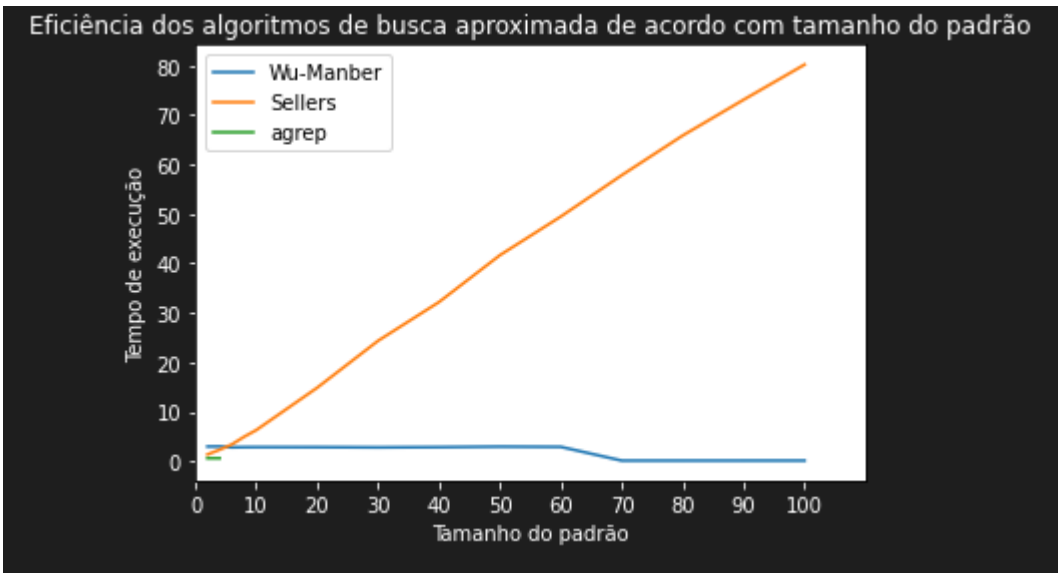
3.2 Casamento Aproximado

O erro máximo considerado para o benchmarking dos algoritmos de busca foi de **1**.

Primeiro benchmark:
Primeiramente, não foi possível determinar o tempo de execução para o agrep nesse benchmarking completo, pois a partir de um pattern de tamanho 6, ele estourava a pilha do sistema (mas apenas nesse teste):

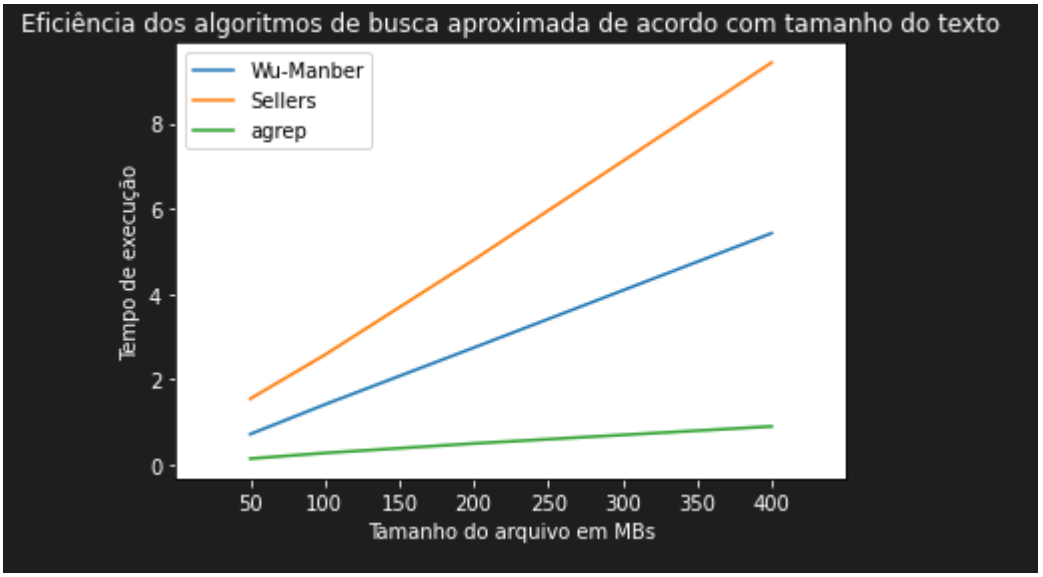
```
1001855

*** stack smashing detected ***: terminated

---------------------------------------------------------------------------
CalledProcessError                        Traceback (most recent call last)
/tmp/ipykernel_30136/346103084.py in <module>
     20     pattern = content[:n]
     21     t1 = t.timeit()
---> 22     subp.check_call(["agrep", "-c", "-1", pattern, "dna.200MB"])
     23     t2 = t.timeit()
     24     y_agrep.append(t2 - t1)

/usr/lib/python3.8/subprocess.py in check_call(*popenargs, **kwargs)
    362         if cmd is None:
    363             cmd = popenargs[0]
--> 364         raise CalledProcessError(retcode, cmd)
    365     return 0
    366

CalledProcessError: Command '['agrep', '-c', '-1', 'GATCAA', 'dna.200MB']' died with <Signals.SIGABRT: 6>.
```

Então, registramos as informações do agrep até onde possível e continuamos com os outros:



Observação: O Shift-Or só realiza a busca se o padrão for menor ou igual a 64 caracteres. Acima disso, ele só emite um aviso de que o algoritmo não é eficiente para esse tamanho de padrão e solicitando ao usuário que tente outro. É por isso que, no gráfico, próximo do tamanho 70, o tempo de execução do shift-or parece ter diminuído bastante.

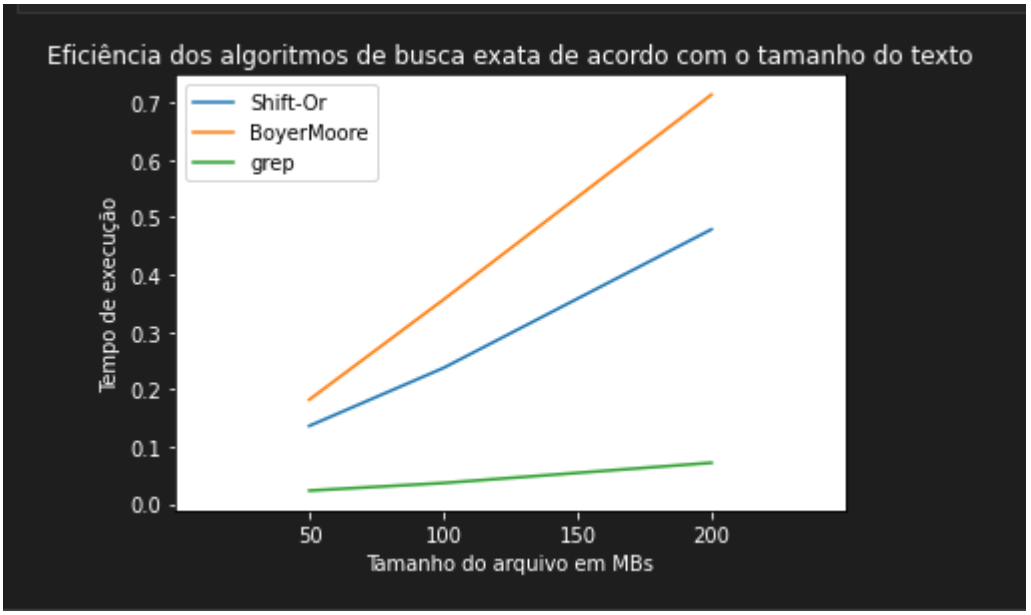
Segundo benchmark:



Conclusões

De forma geral, não conseguimos bater o desempenho do grep, nem mesmo do agrep, para esses testes. Acreditamos, inicialmente, que o alfabeto curto dos textos de DNA (“ACTG “) pudesse ter fatorado nisso. É de se esperar, por exemplo, que o algoritmo de Boyer-Moore tenha menos chances de dar um grande salto, pois como só há essencialmente 4 caracteres no alfabeto, eles repetem com bastante frequência. Sendo assim, criamos ainda um terceiro benchmark, dessa vez usando textos com palavras comuns em inglês, seguindo a mesma metodologia do benchmark 2, isso é, mantendo um padrão fixo “book” e aumentando o tamanho do

texto (usamos as versões de 50MB, 100MB e 200MB presentes no repositório. Porém, o resultado foi bastante similar ao gráfico do texto de DNA:



Sendo assim, a nossa teoria mais recente para a performance tão abaixo do grep e agrep é a maneira que fazemos a operação da leitura de dados. Numa segunda versão, tentaremos dar atenção especial a esse aspecto do projeto.

Já sobre os resultados de busca aproximada, podemos concluir:
Como o algoritmo de Sellers necessita de uma matriz bidimensional onde uma dimensão é o texto onde a busca ocorre e a outra é o padrão, nota-se que o seu desempenho deteriorou bastante à medida que o padrão e texto foram crescendo.