Universidade Federal de Pernambuco – UFPE

Centro de Informática – CIn

Recife – 2021.2

Processamento de

Cadeias de Caracteres

Relatório PMT

Escrito por:

Eidson Sá (ejas),

Vinícius Dantas (vdj)

Entrega do relatório: 07/04/2022

Sumário

Identificação

A equipe é formada por Eidson Jacques Almeida de Sá e Vinícius Dantas Januário. Vinícius fez os algoritmos de casamento e a leitura dos arquivos de texto processados pela aplicação. Eidson fez o parsing dos argumentos e a lógica, tratamento de exceções, seleção de algoritmos e fluxo da aplicação, bem como a realização dos scripts de teste em shell script e processamento do relatório em python.

Implementação

Descrição da ferramenta

A pmt é uma ferramenta de uso por linha de comando (CLI) que permite buscar um ou mais padrões em um arquivo de texto fornecido como entrada.

Na classe de algoritmos de casamento exato, implementamos: Sliding Window (Brute Force), KMP, Boyer Moore e Aho-Corasick. Para os algoritmos de casamento aproximado utilizamos o Sellers e Wu-Manber. O algoritmo Shift-Or não foi implementado mas é usado indiretamente através do Wu-Manber com distância de edição = 0 durante os testes.

A ferramenta permite o usuário escolher o algoritmo a ser utilizado através da opção --algorithm, mas caso o usuário não especificar o algoritmo a ser utilizado, a ferramenta usa a seguinte regra:

emax == 0:
- o pattern está em um arquivo: Aho-Corasick
- o pattern não está em um arquivo:
  - o pattern tem tamanho 1: Wu-Manber (Equivalente ao Shift Or)
  - o pattern tem tamanho maior que 1: Boyer-Moore

emax != 0:
- o tamanho do pattern é menor ou igual a 64: Wu-Manber
- o tamanho do pattern é maior que 64: Sellers

Tendo como base os resultados que obtivemos com os testes de desempenho.

Detalhes de implementação relevantes

Para estratégia de leitura de arquivos linha a linha, utilizamos as funções padrão de C (fgets), armazenando as linhas em um buffer que suporta cerca de 10mb. Também vimos que esse processo poderia ser acelerado em arquivos muito grandes com o uso de técnicas mais avançadas como memory mapping.

Tentamos ao máximo estruturar as classes do projeto, criando uma classe abstrata como base para os algoritmos de busca de um só padrão (SinglePatternSearch.h). Porém encontramos dificuldades para encontrar uma solução que englobasse o algoritmo Aho Corasick e que não trouxesse uma perda de desempenho geral da aplicação, tendo em vista seu comportamento

diferente com relação aos demais algoritmos. Optamos por mantê-lo em uma classe com métodos e atributos estáticos.

Criamos nossa própria classe para representar as cadeias de caracteres, buscando uma forma de chegar num meio termo entre usar char* e o overhead da classe std::string.

De forma geral buscamos re-implementar os algoritmos vistos em sala de aula, se atentando aos detalhes de implementação ao se trabalhar com C++. Decidimos colocar o vetor (std::vector) de ocorrências dentro da classe de busca com atributo público e retornar apenas um número de ocorrências, visando diminuir o número de criações desse objeto com cada chamada da função de busca, mesmo que seu retorno já seja otimizado com a move semantics de C++. No algoritmo Aho Corasick representamos um nó da FSM como um struct englobando todos os seus atributos (goto, fail e ocorrências). O algoritmo Wu Manber é limitado a 64 caracteres, devido ao limite do long int para representar as máscaras binárias. Tentar utilizar um std::bitset se mostrou uma perda de performance que não valeria a pena. Além disso, criamos uma classe auxiliar (BenchMarkTimer) para nos ajudar na medição de performance de um determinado escopo da aplicação.

Por fim aplicamos uma opção extra na aplicação: -n --noacopt, que não recebe argumentos e desabilita uma otimização que amortiza a constante de reportar as posições das ocorrências no Aho Corasick, limitando-se apenas a somar a quantidade de ocorrências. Essa otimização mostra ter bons ganhos de desempenho na busca de muitos padrões e com número elevado de ocorrências dentro do texto, como descrito no quarto teste da próxima seção.

A aplicação não possui nenhum bug conhecido, porém falta uma verificação automática do output que obtivemos dos testes de desempenho para validar com mais rigor a corretude do programa. Nos limitamos a testar manualmente a aplicação e comparar os resultados com o grep, brute force e código python da disciplina, nenhum erro foi encontrado até o momento. Para os algoritmos de casamento aproximado, não conseguimos testar a corretude com a ferramenta agrep, pois sua contagem de linhas que possui padrões é diferente. Nesse caso comparamos os resultados entre os dois algoritmos e os scripts vistos em sala de aula.

Testes e Resultados

Para os testes de desempenho da aplicação foram utilizados 3 datasets, disponíveis em http://pizzachili.dcc.uchile.cl/texts.html. Foram eles:

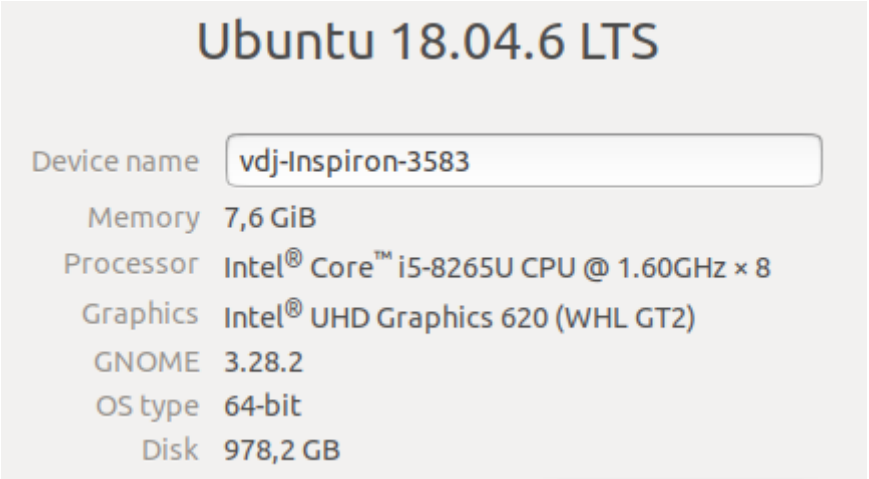
- english.1024MB: Texto em língua inglesa de 1gb.
- sources.200MB: Código fonte na linguagem C de 200mb.
- dna.200MB: Sequências de DNA, com alfabeto limitado, com 200bmb.

Para os dois primeiros datasets foi necessário fazer um pré processamento do texto, devido a estarem no formato ISO-8859-1. Utilizamos a ferramenta *iconv* do linux para converter os arquivos em ASCII.

Para comparação de desempenho de casamento exato e aproximado foram utilizadas as ferramentas grep e agrep, respectivamente. Para busca em múltiplos padrões também se utilizou o grep, tanto passando um arquivo de padrões como opção ou individualmente para cada padrão.

No total foram realizados 7 conjuntos de testes, cada teste foi realizado 3 vezes e o tempo de execução considerado foi a média aritmética das 3 execuções.

Todos os testes foram realizados em uma máquina com disco rígido na seguinte configuração:

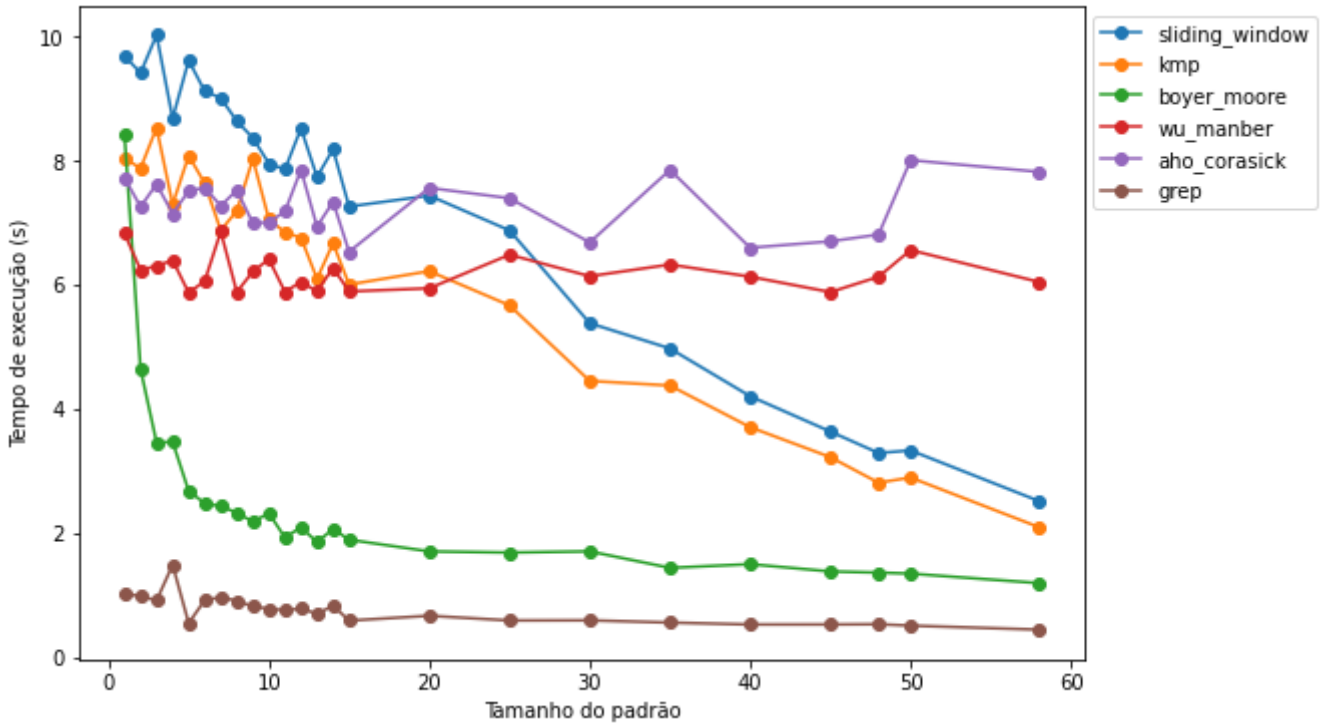


Casamento Exato

Para o casamento exato foram realizados 4 testes, 3 para busca de um único padrão e um para busca de múltiplos padrões.

No primeiro teste usamos o dataset english.1024MB, nele buscamos palavras da língua inglesa e trechos mais longos retirados diretamente do texto. Seu resultado se encontra na seguinte figura:

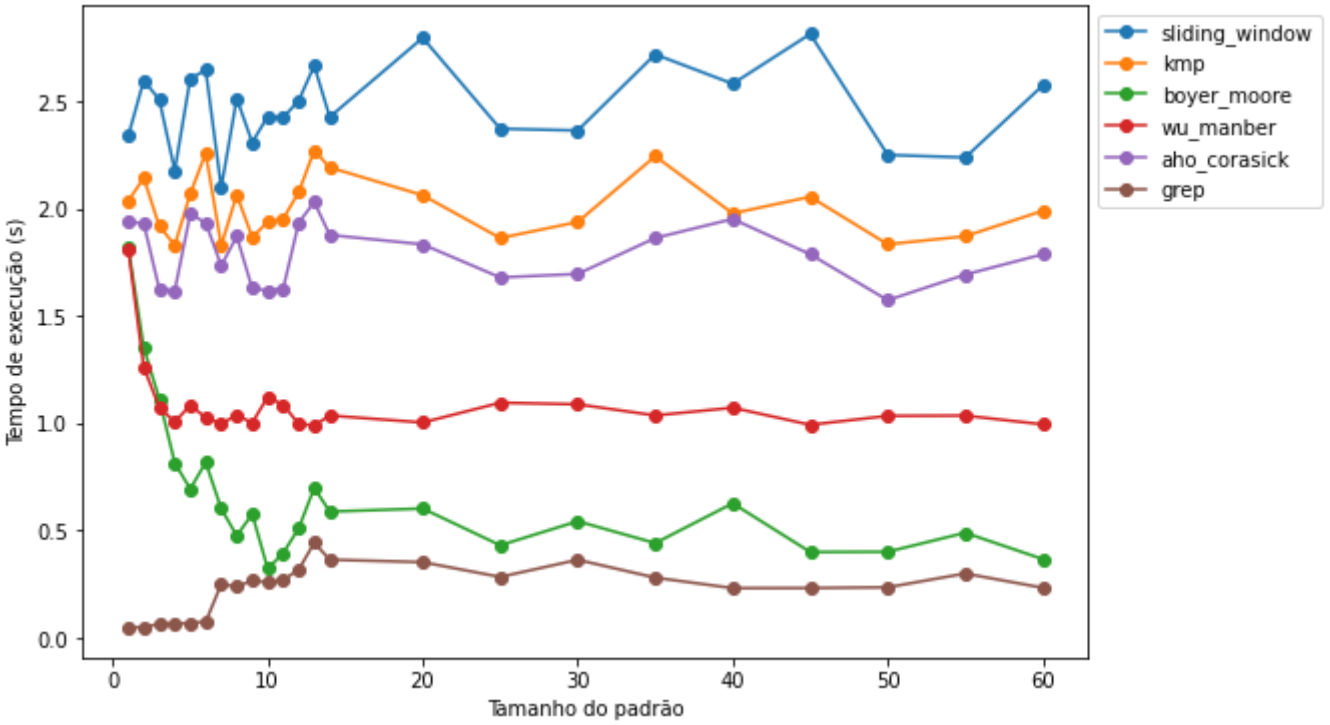
Palavras em inglês, english.1024MB



Nota-se que o Boyer Moore não desempenha bem para o padrão de tamanho 1, mas que vai ganhando desempenho na medida que o tamanho do padrão cresce. O Sliding Window acompanha a mesma curva de desempenho que a do KMP, sendo um pouco pior que este. Tanto o Wu Manber quanto o Aho Corasick apresentaram desempenho mais uniformes e que não melhoram à medida que o tamanho do padrão aumenta. O grep desempenha muito bem em todos os casos.

O segundo teste foi feito no sources.200MB. As palavras buscadas foram geradas aleatoriamente de maneira uniforme dentro do alfabeto. O resultado está na figura a seguir:

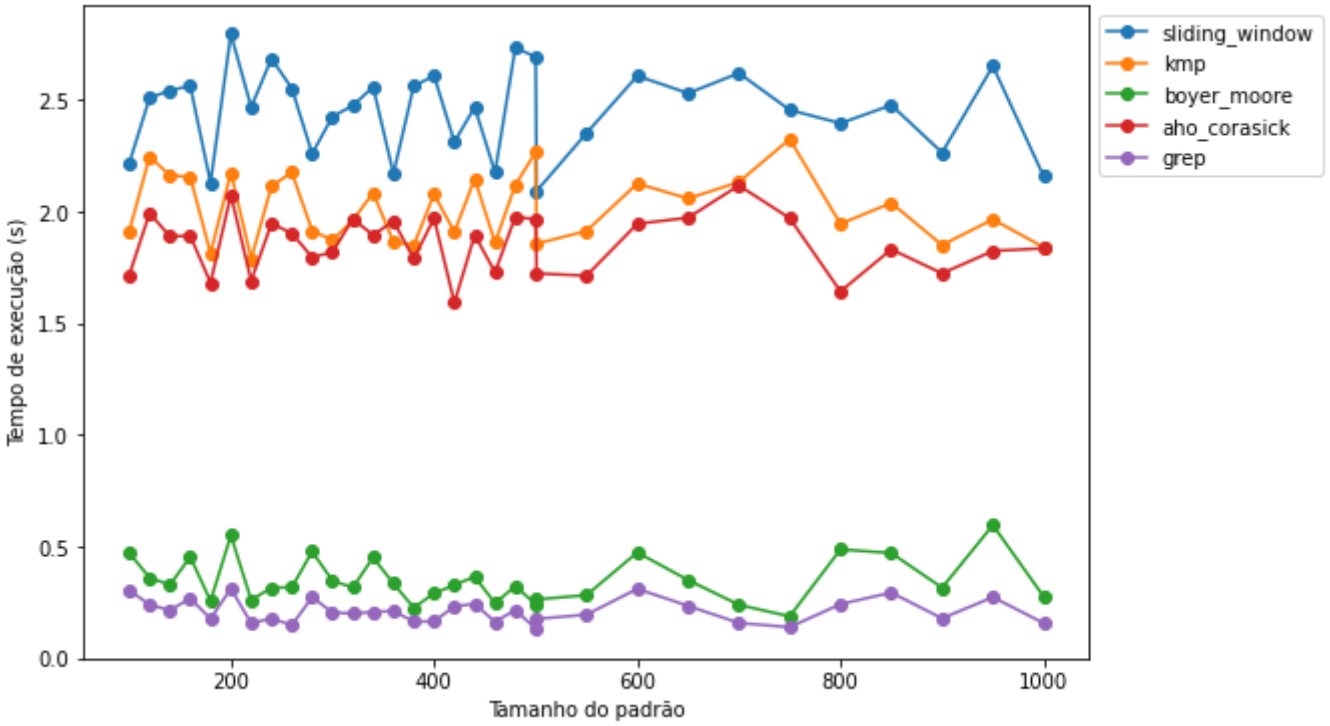
Sequências de dna em dna.200MB



Nesse caso podemos notar que o Aho Corasick manteve um desempenho melhor que o KMP e o Sliding Window para o texto em alfabeto pequeno, que nesse caso não ganharam desempenho com o tamanho do padrão. O Wu Manber se manteve estável, porém apenas a partir do terceiro padrão testado. O Boyer Moore não conseguiu um desempenho melhor que o Wu Manber para esses 3 casos e conseguiu acompanhar bem a curva do grep a partir dos padrões de tamanho 10.

O terceiro teste foi feito no mesmo dataset dna.200MB, sendo que com padrões muito longos também gerados aleatoriamente dentro do alfabeto, com o resultado a seguir:

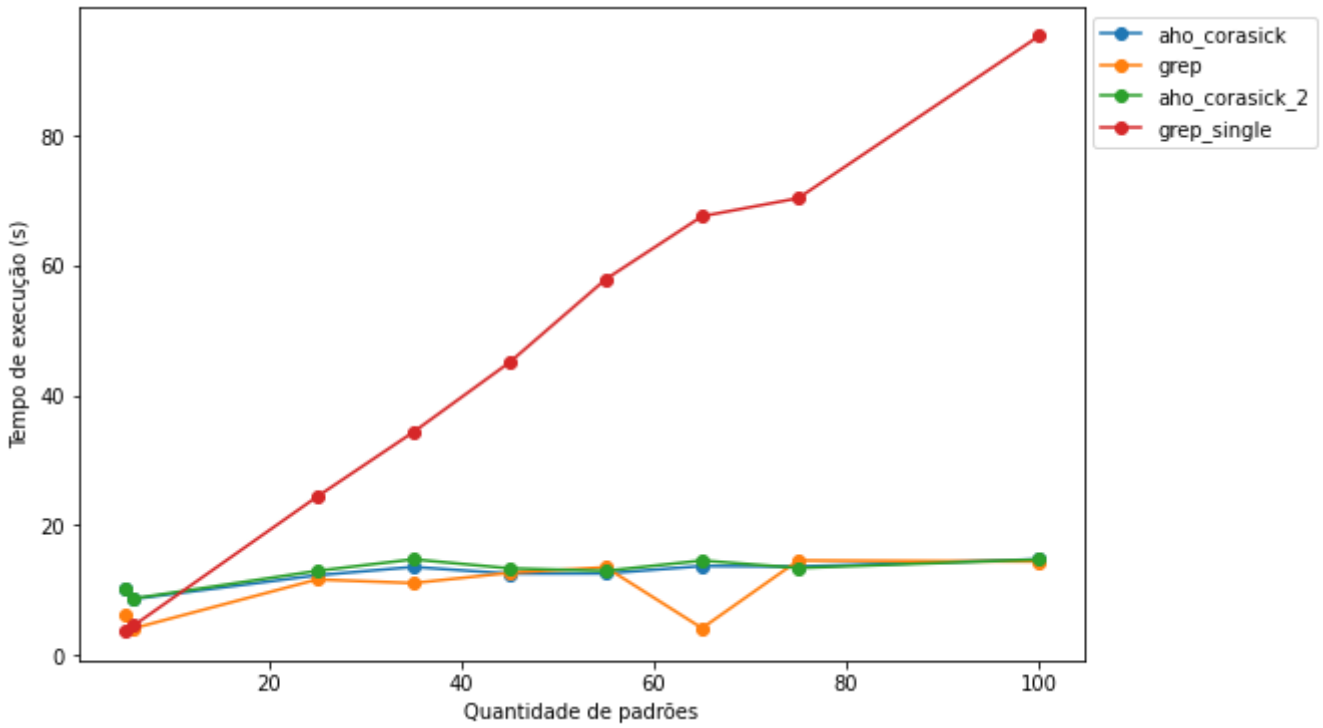
Sequências longas de dna em dna.200MB



Nele podemos observar um comportamento continuado do teste anterior, temos duas classes de desempenho onde cada algoritmo segue a mesma curva que o outro, com o Aho Corasick e o grep liderando em cada classe.

O Quarto teste foi feito com busca com múltiplos padrões, utilizando as duas versões do Aho Corasick, a aho_corasick sendo a versão sem a opção -n e aho_corasick_2 a versão com o -n. Notamos que o grep possuía um desempenho bastante elevado passando um arquivo de padrões, para fins de ilustração decidimos também testar ele sendo rodado individualmente para cada padrão (grep_single). O teste foi feito no english.200MB e todas as palavras buscadas foram sorteadas dentro de um conjunto de 3000 palavras comuns da língua inglesa, com o número de palavras progressivamente aumentando. Se obteve o seguinte resultado:

Múltiplas palavras em inglês, english.1024MB



O gráfico ilustra bem que os algoritmos de busca de um padrão não são páreos para os algoritmos de busca de múltiplos padrões nesse caso de uso. Podemos notar que o grep também possui uma implementação para esse caso de uso, e que em alguns pontos nosso algoritmo conseguiu ser melhor. Nota-se também que em alguns pontos o Aho Corasick com otimização possui uma leve vantagem sobre a versão sem otimizações. Essa diferença fica ainda mais evidente quando testamos um caso com as 3000 palavras do conjunto de palavras nesse mesmo dataset, onde a quantidade de ocorrências das palavras era muito grande e o overhead de percorrer e apontar a lista de ocorrências é muito grande, como ilustrado na tabela abaixo:

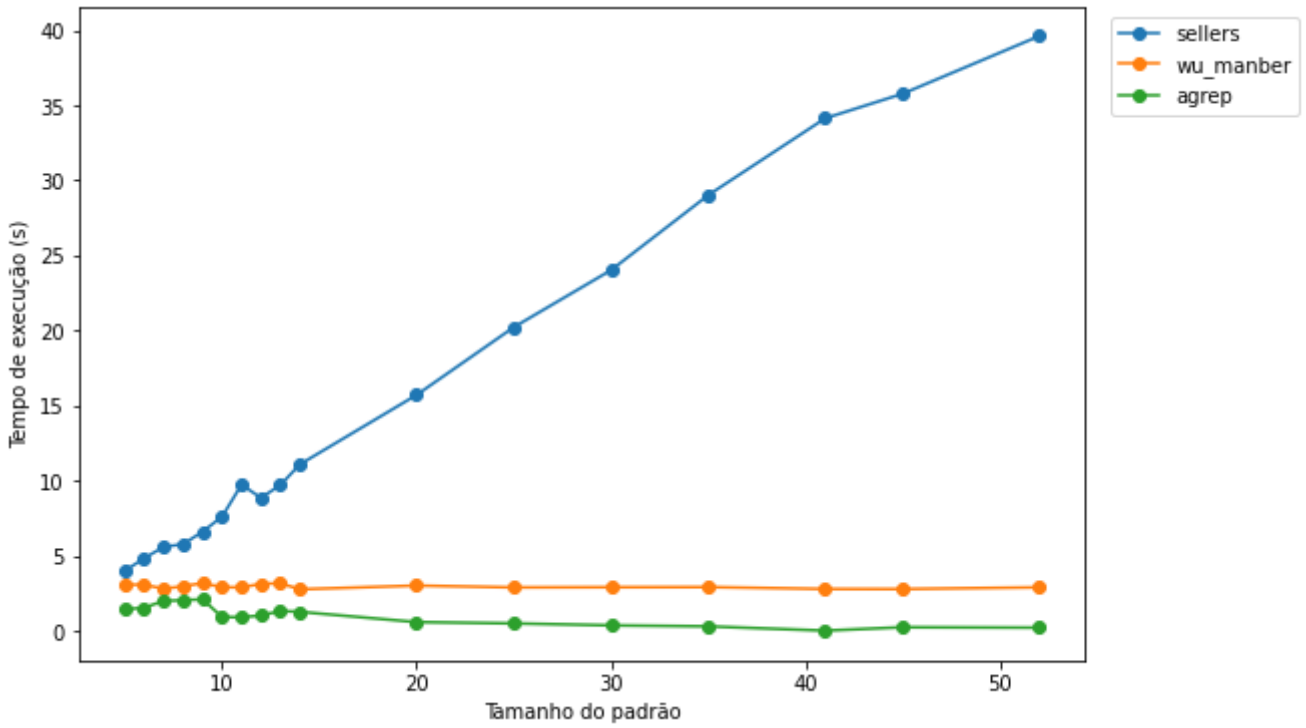
	aho_corasick	aho_corasick_2	grep
Tempo de execução(s)	15,641	21,088	1,997

Com esse teste mais extremo, onde o número de ocorrências chega a ser de 372474919, vemos que o ganho de performance por não reportar as posições de ocorrências dos padrões é muito grande, com ganho de cerca de 5,5 segundos de performance. Mas em um primeiro momento o que mais nos impressionou foi o enorme ganho de desempenho que o grep teve. Lendo um pouco a respeito sobre a possibilidade de usar multi-threading no Aho Corasick, suspeitamos que o grep deva usar esse tipo de estratégia para um número elevado de padrões.

Casamento Aproximado

Para o casamento aproximado foram feitos três testes, utilizando os algoritmos Sellers, Wu Manber e a ferramenta agrep nos dois primeiros. O primeiro foi feito no dataset sources.200MB, com emax fixo em 4 e buscando palavras selecionadas de um conjunto de palavras da língua inglesa. Obtivemos o seguinte resultado:

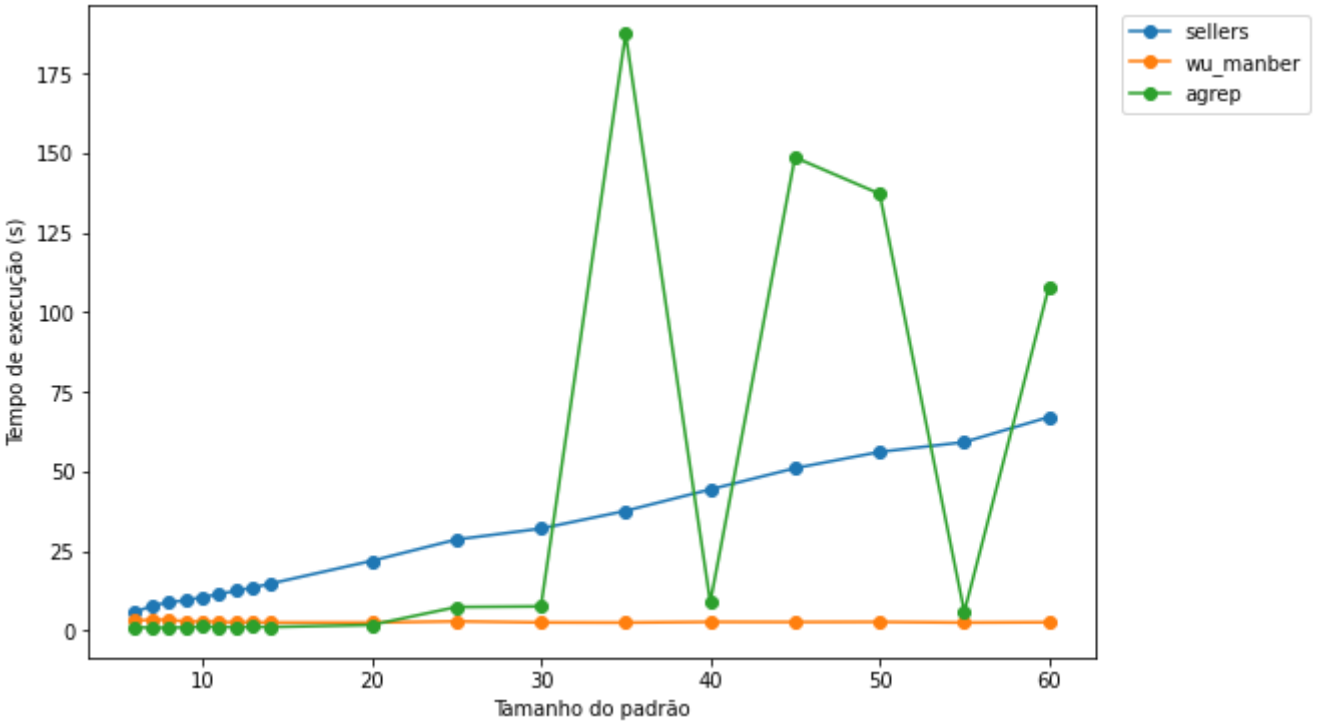
Palavras em inglês com emax = 4, sources.200MB



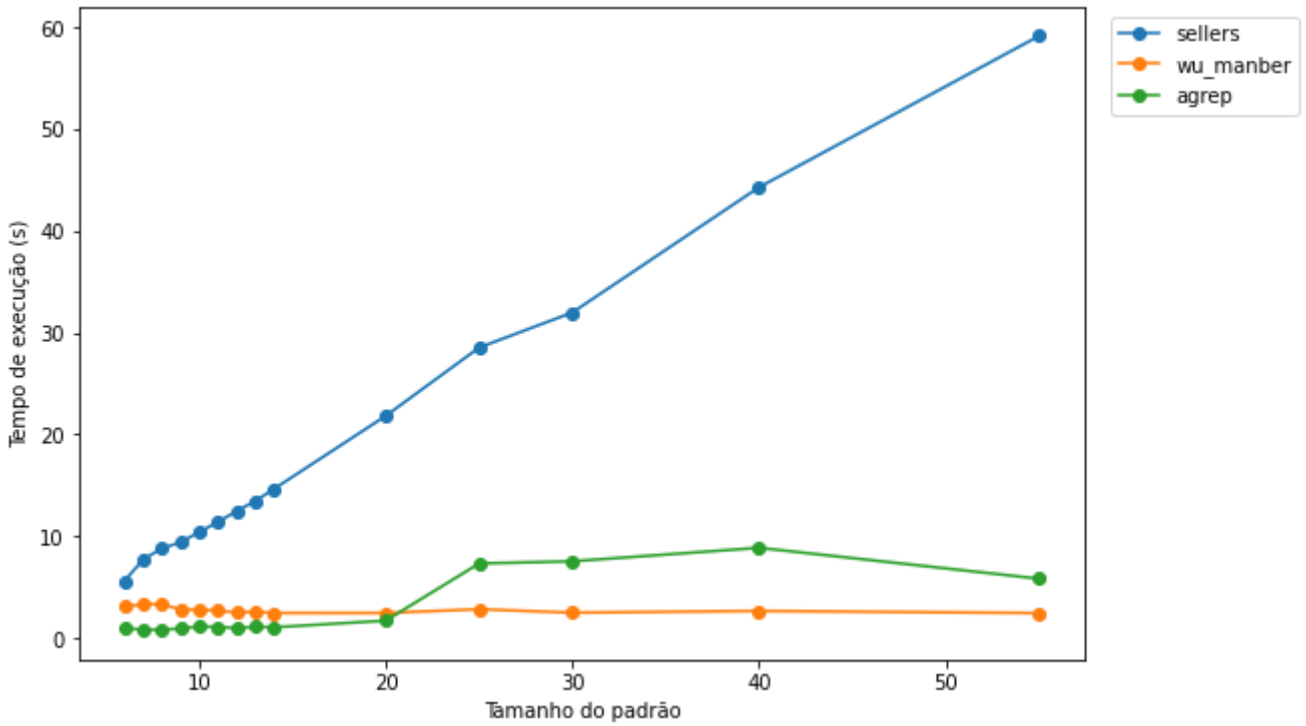
Com erro fixo podemos notar que o tempo de execução do Wu Manber se mantém estável, assim como o do agrep, tendo em vista que a complexidade do Wu Manber é quadrática no erro emax. Como a complexidade do Sellers é quadrática no tamanho do padrão, seu tempo de execução naturalmente vai aumentar com o tamanho do padrão.

Já o segundo teste foi feito nas sequências de dna, também com erro fixo emax = 4 e com palavras geradas aleatoriamente dentro do alfabeto. O que mais salta os olhos é o comportamento não uniforme da ferramenta agrep, como mostra a figura a seguir. Incluímos duas versões do gráfico sem os pontos fora da curva do agrep.

Sequências de dna com emax = 4, dna.200MB



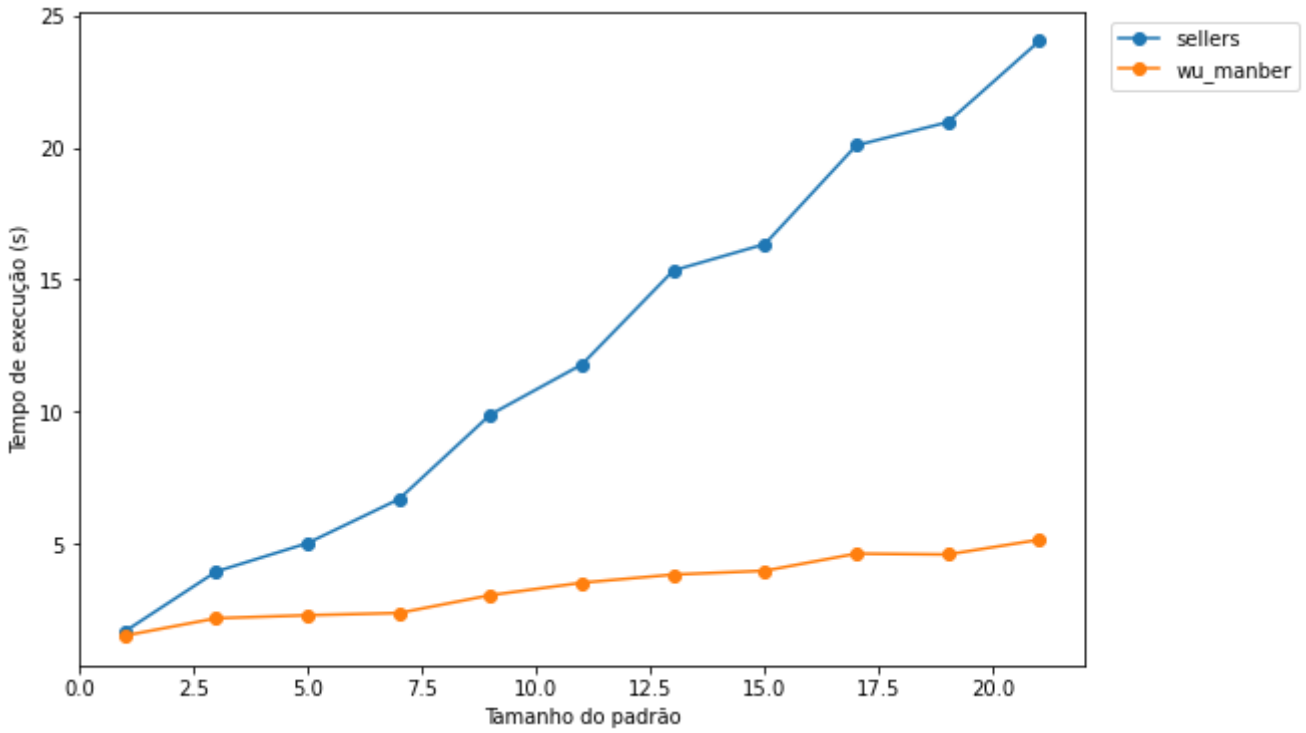
Sequências de dna com emax = 4, dna.200MB



Carecemos de uma explicação do por que o agrep teve esse desempenho, sabemos que a partir do padrão 11 não há mais ocorrências aproximadas dos padrões no texto. No mais vimos que os algoritmos implementados demonstraram o mesmo padrão descrito no teste anterior.

No último teste usamos também as sequências de dna, só que agora variando o erro emax = tamanho do padrão - 1, obtendo assim o seguinte resultado:

Sequências de dna com emax = len(patt) - 1, dna.200MB



Podemos ver que agora a curva de desempenho do Wu Manber começa a aumentar, seguindo sua complexidade quadrática com relação ao erro. Porém vemos que mesmo no pior caso a constante do Wu Manber é bem inferior que a do Sellers.

Conclusões

Realizado os testes, vimos que para a categoria de casamento exato de um único padrão o algoritmo Boyer Moore foi o que obteve melhor desempenho em boa parte dos casos, com exceção em palavras de tamanho 1 onde o Wu Manber (emax = 0) se sobressaiu. Em linguagem natural, todos os algoritmos de janela deslizante (Boyer Moore, KMP, Sliding Window) ganham performance à medida que o padrão aumenta, o que não é o caso quando testamos o algoritmo em um texto de alfabeto reduzido, principalmente para o KMP e o Sliding Window. Ambos Wu Manber e Aho Corasick se mostraram opções estáveis em ambos os alfabetos, com o último obtendo melhor performance que o KMP e o Brute Force em um alfabeto reduzido.

Para a categoria de casamento exato múltiplo, fica evidente que a abordagem do Aho Corasick é a melhor opção, com esse desempenho podendo ainda ser melhorado com o paralelismo.

Nos algoritmos de casamento aproximado a superioridade do Wu Manber em relação ao Sellers é evidente, tanto em complexidade quanto em constante, porém a grande desvantagem deste algoritmo é sua limitação à quantidade de bits do long int (64). Algo que pode ser feito é implementar ou usar um bitset, porém fica um pouco mais difícil de otimizar o código e inevitavelmente tem que se arcar com um overhead envolvido.