Ferramenta PMT

Antônio Harger Rivero - ahr
Vinicius Vieira Moreira - vvm

Introdução

A ferramenta *pmt* descrita neste relatório atende às funcionalidades básicas descritas nas especificações do projeto, com exceção de algumas funcionalidades relacionadas ao casamento aproximado de padrões, as quais descreveremos adiante na seção de Implementação.

De forma geral, durante a implementação dos algoritmos e da ferramenta em si, utilizamo-nos de *pair programming*, garantindo assim o entendimento dos algoritmos e suas estratégias pelos dois membros da dupla, além de aumentar a velocidade de implementação. Durante a fase de testes inicialmente escrevemos alguns scripts em *bash* e posteriormente em Python.

Implementação

Descrição funcionamento ferramenta

Dado que o uso e funcionamento da ferramenta devem seguir o padrão POSIX, primeiramente tínhamos que garantir que o nosso *pmt* teria o seguinte funcionamento básico:

$ pmt [options] pattern textfile [textfile...]

Sendo *options*: -e, --edit; -p, --pattern; -a, --algorithm_name; -c, --count e por último -h, --help

Uma vez que implementamos a ferramenta em C#, não foi possível utilizar a *GNU C Library* (getopt(), por exemplo). Para isso, implementamos as classes *CommandDescription.cs* e *Command.cs*, onde a primeira realiza o *parsing* dos comandos e opções e a segunda constroi um objeto do tipo *Command*, baseando-se na descrição do mesmo. Para além disso também criamos uma classe *ResultHandler*, que lida com os resultados retornados pelos algoritmos.

Algoritmos de Casamento Exato

Os algoritmos que implementamos para essa modalidade foram o **KMP** e o **Aho-Corasick**. Iniciamos as implementações pelo KMP por ser mais intuitivo e sua estratégia

ser de fácil visualização durante a execução, já o Aho-Corasick foi escolhido devido a sua facilidade de lidar com um conjunto grande de padrões a serem casados, pois uma vez que o Aho-Corasick é treinado (a construção de sua *triee*/*HashSet)* a execução é muito rápida.

Algoritmos de Casamento Aproximado

Na modalidade de casamento aproximado a ferramenta tem como única opção o algoritmo Sellers.

Decisões relevantes de implementação

Uma vez que os únicos algoritmos de casamento exato implementados foram o KMP e o Aho-Corasick, a maior diferença que temos entre os dois são suas capacidades e eficiência para lidar com poucos padrões (KMP) ou uma grande quantidade de padrões a serem casados (Aho-Corasick). Podemos assim nos basear na quantidade de padrões a serem casados para decidir qual algoritmo será usado.

Estruturas de Dados

- **KMP**
    - No KMP utilizamos apenas um Array de inteiro para armazenas as ocorrências de sufixos que também eram prefixos durante o casamento de padão.
- **Aho-Corasick**
    - Para armazenarmos o alfabeto de caracteres presentes nos padrões de entrada utilizados pelo Aho-Corasick, utilizamos um *HashMap* da biblioteca padrão do C#, uma vez que precisávamos saber se determinado caractere estava presente no padrão, tendo a *HashTable* eficiência maior em detrimento a uma simples lista, por exemplo.
- **Sellers**
    - Utilizamos apenas de listas para fazer a tabela. Não foram tomadas decisões muito relevantes.

Bugs conhecidos e limitações

Para além das limitações da nossa implementação do Sellers, a *CLI* que implementamos não atende completamente os padrões POSIX, como o uso de *wildcards* e o *autocomplete*, uma vez que codificamos o nosso próprio *parser*.

Situações que os algoritmos foram empregados

Testes e Resultados

Os testes descritos a seguir foram conduzidos em um computador com sistema operacional Ubuntu Gnome 16.04, 4GB de memória RAM e processador Intel Core i5 2.2 GHz, alem disso utilizamos a plataforma Mono Runtime para executar os binários.

Tipos de dados utilizados

Para a maioria dos testes utilizamos textos em inglês de tamanhos 50MB, 100MB e 200MB. Para as entradas retiramos de listas das palavras mais comuns do inglês divididas em três tamanhos (curtas, médias e longas). [Além disso rodamos sobre padrões de proteínas]****

Ferramentas e algoritmos utilizados para benchmark

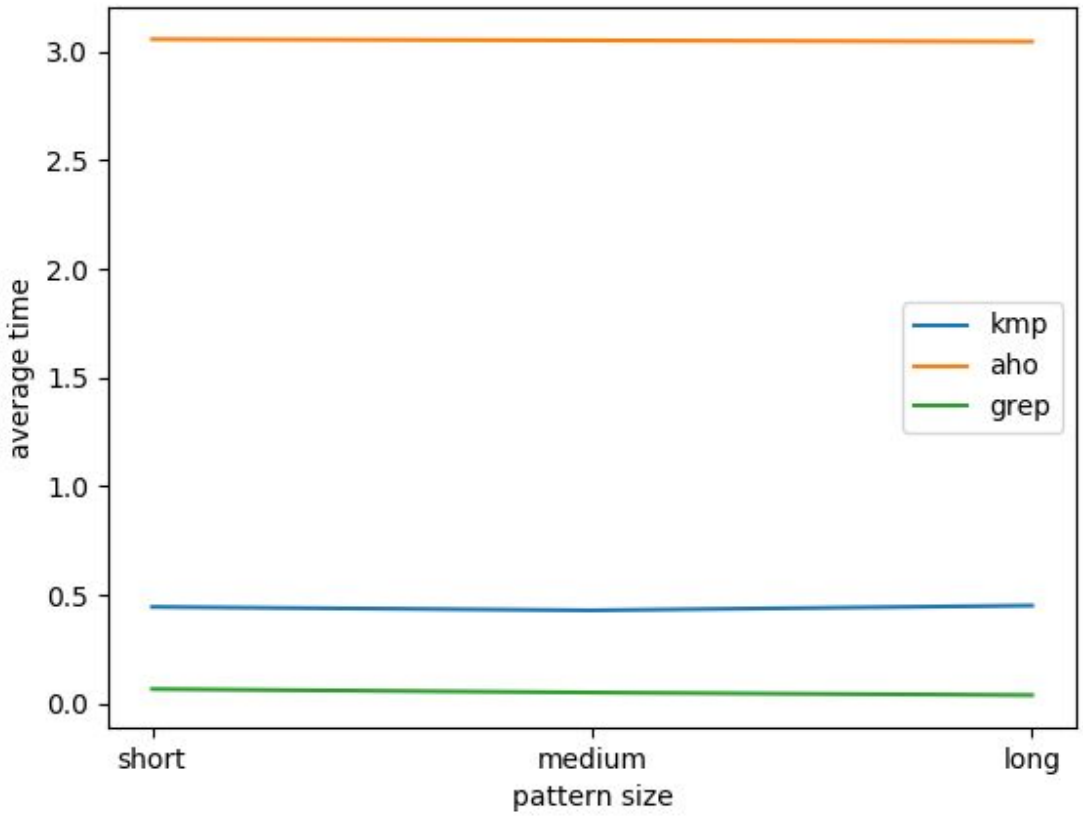
Para o *benchmark* utilizamos a ferramenta *time* da suite GNU, que nos retornava o tempo de execução dos comandos.

Ambiente de testes

Os testes foram realizados através de scripts em python, que faziam chamadas na linha de comando, armazenavam os tempos médios de execução em arquivo e geravam os gráficos dos resultados.

Experimentos Realizados

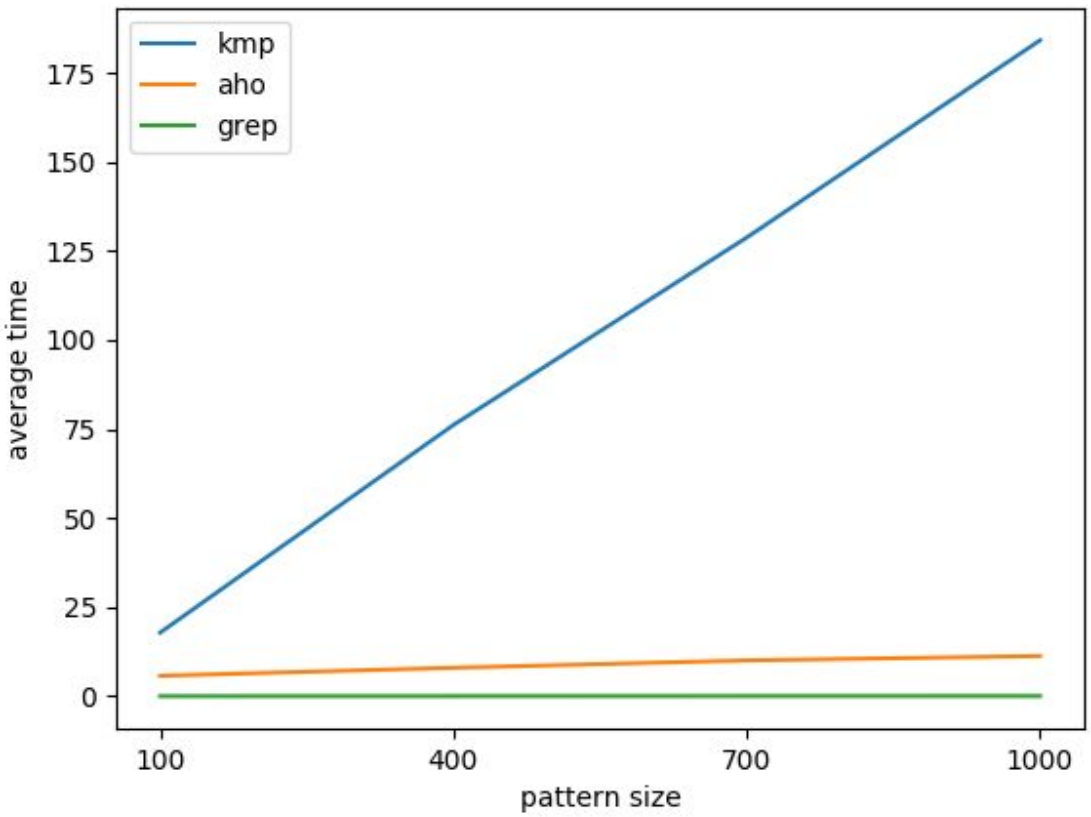
Teste 0



O teste acima foi realizado iterando 100 vezes os algoritmos recebendo cada uma dessas iterações um padrão aleatório, retirado de um de nossos arquivos de padrões (curtos, médios e longos) contendo 20.000 padrões cada. É importante salientar que o utilizamos neste teste um arquivo de 50MB de textos em inglês (Pizza&Chili) e padrões em inglês mais recorrentes de acordo com uma pesquisa da Google™.

Como este foi um teste inicial e de certa forma experimental, além do arquivo de texto ser de apenas 50MB, não pudemos identificar muitas diferenças entre os algoritmos em função do tamanho da entrada, apenas a vantagem do grep em cima das outras implementações e a maior velocidade do KMP em relação ao Aho-Corasick, uma vez que o cenário ‘ótimo’ para o Aho-Corasick é o casamento de muitos padrões.

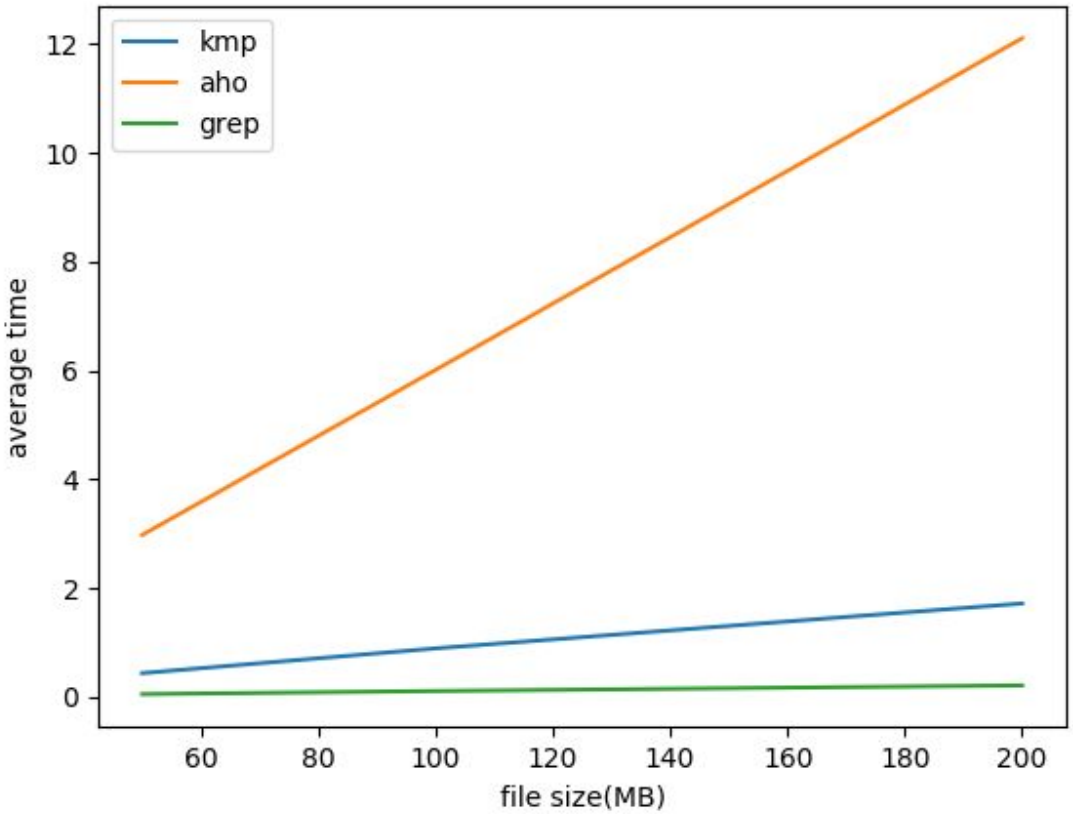
Teste 1



Errata: onde lê-se pattern size é quantidade de padrões

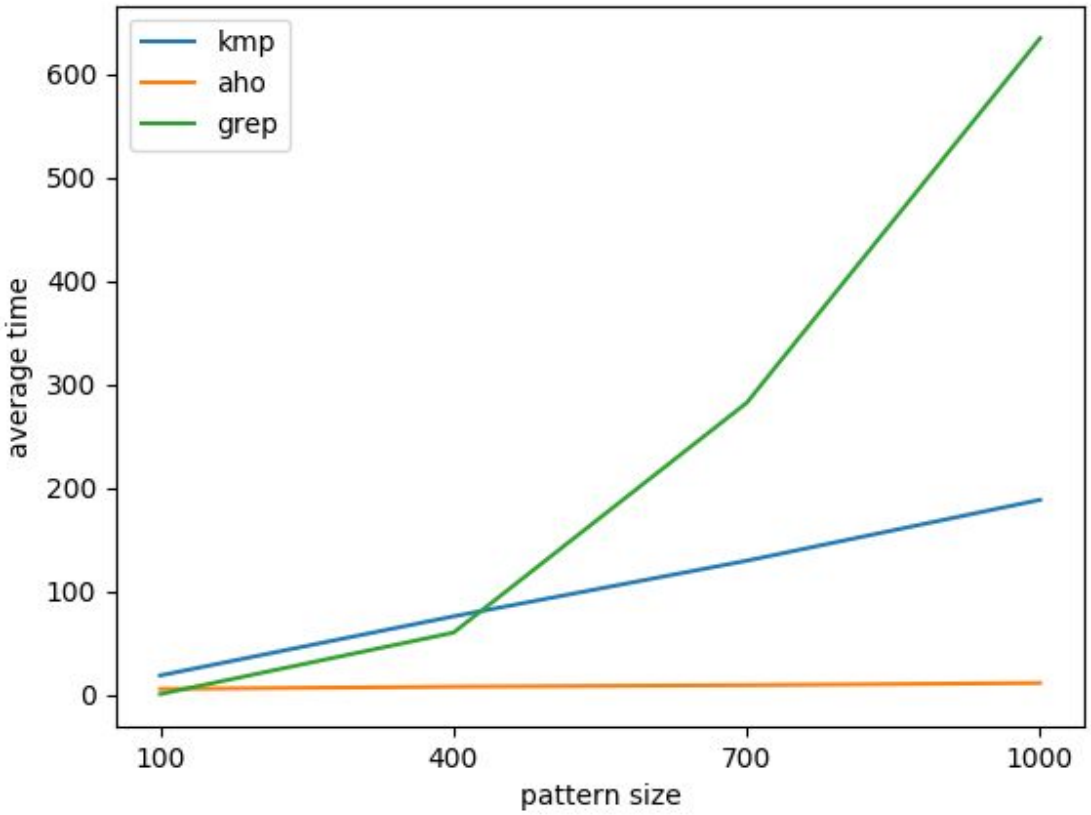
Neste teste executamos a ferramenta recebendo uma determinada quantidade de padrões (vide gráfico), sendo esses padrões de tamanhos variados. O arquivo de texto utilizado também foi em de textos em inglês e de tamanho 50MB. Neste teste pudemos verificar a eficiência do Aho-Corasick em lidar com múltiplos padrões, mantendo sua performance próxima a do *grep*, além disso, podemos perceber a escalada do tempo de execução do KMP quando executado com uma maior quantidade de padrões.

Teste 2



Neste teste realizamos 20 iterações de cada algoritmo, recebendo padrões de tamanho médio aleatórios contidos em um arquivo com 20000 padrões em inglês. Os arquivos em que os padrões foram buscados também continham texto em inglês. Pudemos perceber certa constâncias e eficiência no nosso KMP, em se tratando de um único ou poucos de tamanho médio, como de costumo pudemos verificar ótima performance do *grep* e também notamos um aumento acentuado em função do tamanho dos textos.

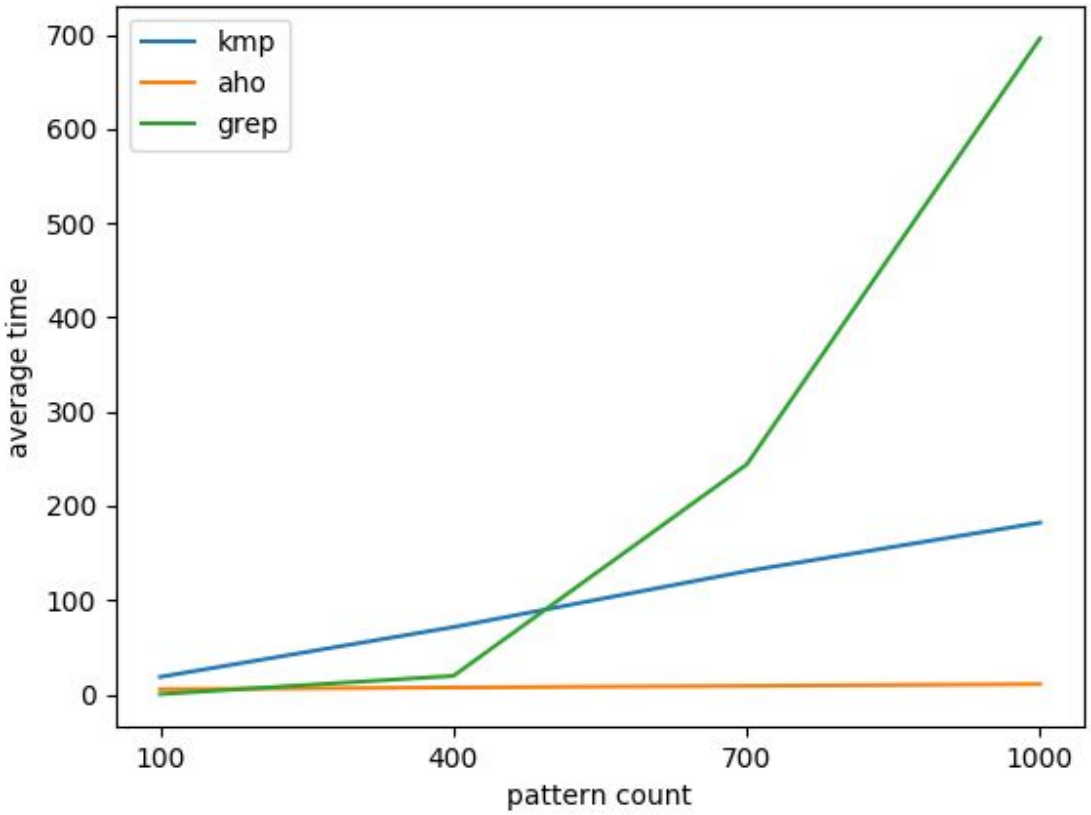
Teste 3



Errata: onde lê-se pattern size é quantidade de padrões

O teste acima foi realizado iterando 20 vezes os algoritmos recebendo cada uma dessas iterações um padrão aleatório de tamanho grande. É importante apontar que esse aumento no tempo de execução do *grep* em função da quantidade de padrões quando executamos *grep -c*. No mais, o desempenho do KMP e Aho-Corasick foi bem próximo de outros testes em cenários semelhantes que realizamos.

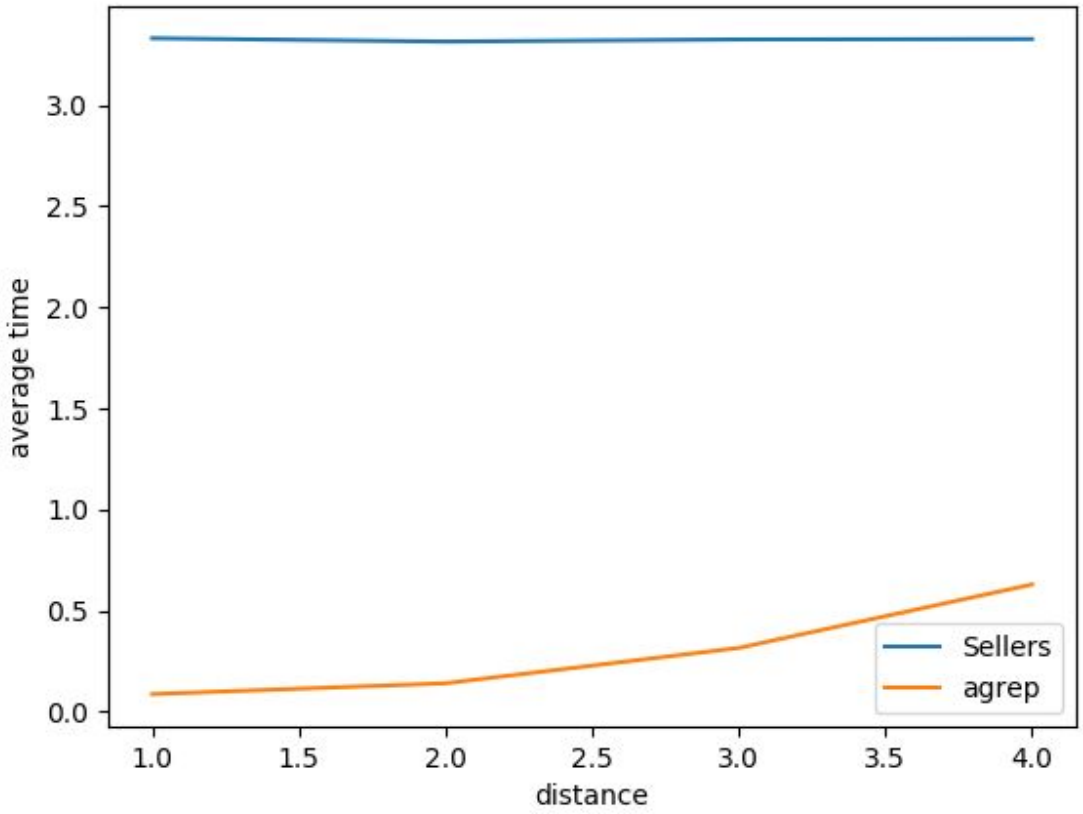
Teste 4



Errata: onde lê-se pattern size é números de padrões

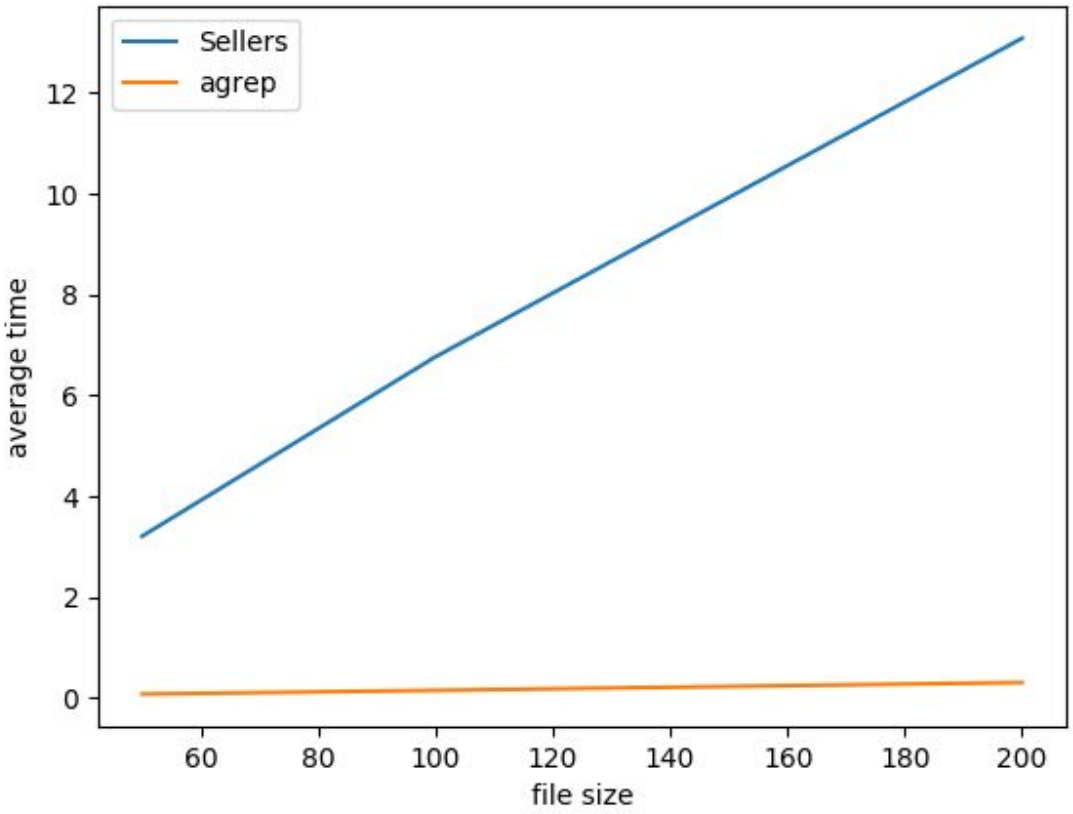
Este teste foi realizado exatamente nas mesmas condições do anterior, com exceção que aqui utilizamos padrões grandes.

Teste 5

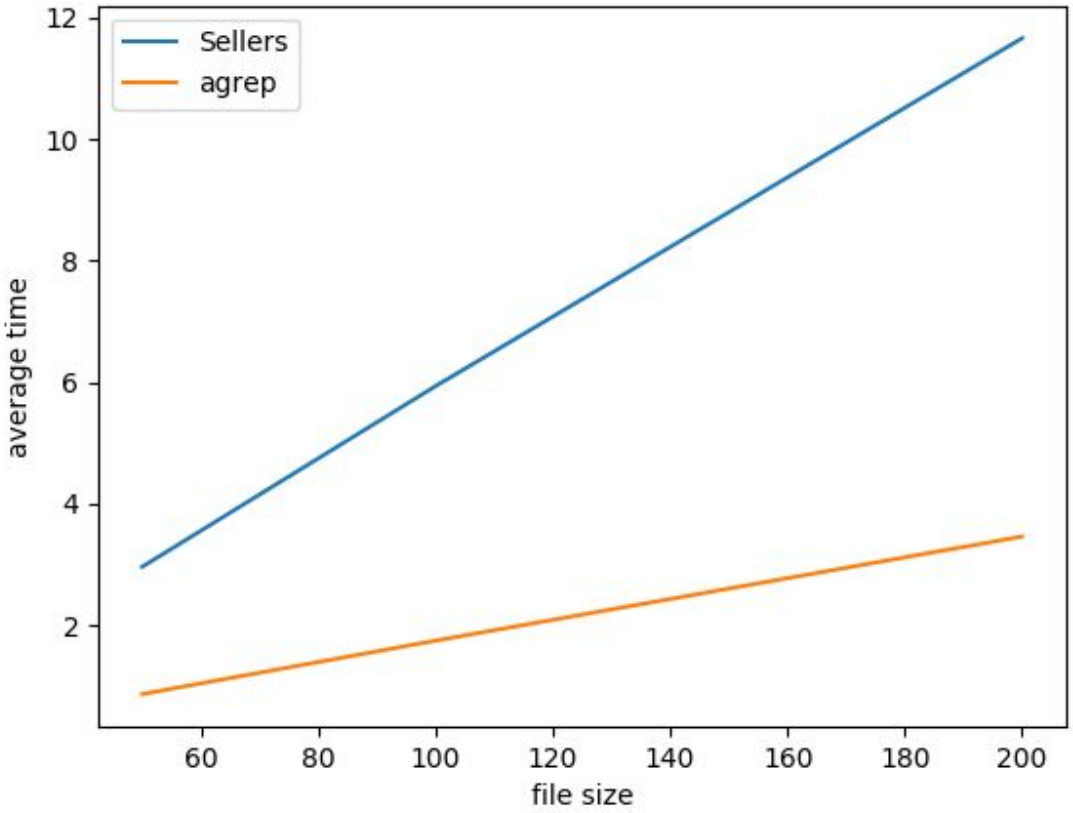


Este teste foi realizado selecionando 20 palavras longas aleatórias em um texto de 50MB e buscando-as com distâncias de 1 a 4. Na comparação entre nossa ferramenta e o agrep, verificamos uma diferença significativa de tempo de busca. Uma observação interessante, no entanto, é que o algoritmo sellers não varia de acordo com a distância.

Teste 6


Sellers
agrep
average time
file size
0
2
4
6
8
10
12
60
80
100
120
140
160
180
200

Neste teste utilizamos padrões longos, distância de edição 1. Fica evidente nesse a ineficiência do Sellers em relação ao *agrep.*



Ao utilizarmos distância de edição 5, observamos uma melhora do Sellers em comparação ao *agrep*.

Discussão e conclusão

Como já verificamos durante o corpo deste relatório, a nossa ferramenta *pmt* demonstra um desempenho razoável em relação ao *grep* em alguns dos cenários testados. Além disso, com a arquitetura que utilizamos em nosso código C#

- Pontos positivos:
    - Apesar da implementação em C#, a ferramenta apresenta um bom desempenho em muitos cenários.
    - Código de fácil manutenção e depuração
- Pontos negativos:
    - Desempenho consideravelmente inferior ao *grep*