Relatório Ferramenta PMT

1. Identificação

1.1 Equipe

Davi Yan Macedo Brandão — dymb
Edson de Melo Neto — emn2

1.2 Descrição das contribuições dos membros da equipe

- Davi focou mais na implementação da CLI, na utilização das threads, auxiliou Edson a debuggar e otimizar os algoritmos implementados, além de implementar o algoritmo aho-corasick.
- Edson focou mais na implementação dos outros algoritmos utilizados, e ajudou Davi a otimizar a forma com que era realizado o parsing, os pré-processamentos e as estruturas de dados em geral.

2. Implementação

2.1. Algoritmos de Casamento Exato

Para o Casamento Exato, resolvemos implementar os algoritmos de **Brute Force, KMP**, **Boyer-Moore** e **Aho-Corasick**.

Após analisarmos os resultados dos testes feitos, confirmamos que o algoritmo de **Boyer-Moore** obteve um melhor desempenho em todos os testes realizados. Um problema que nos deparamos foi a performance do algoritmo de **KMP**, e não conseguimos descobrir o motivo pelo qual ele estava com um resultado inferior inclusive quando comparado com o algoritmo de **Brute Force**. Portanto, decidimos temporariamente utilizar sempre o algoritmo de **Boyer-Moore.**

2.2. Algoritmos de Casamento Aproximado

Nessa etapa do projeto, resolvemos implementar o algoritmo de **Sellers** e o algoritmo de **Wu-Manber**.

Nossa implementação do algoritmo de ***Wu-Manber*** é restrito a padrões com tamanho menor ou igual a 64, pois utilizamos inteiros de 64-bits para representar as máscaras de caracteres utilizadas no algoritmo.

Portanto, a ferramenta prioriza o uso do ***Wu-Manber*** quando o tamanho do padrão é menor ou igual a 64, e utiliza o algoritmo de **Sellers** caso contrário.

2.3. Detalhes Importantes

2.3.1 Threads:

Decidimos implementar a ferramenta fazendo uso da API POSIX Threads (“pthread”), de modo a paralelizar as execuções e aproveitar o paralelismo de hardware dos computadores modernos. A ferramenta cria uma thread para cada arquivo de texto, para cada padrão. Para o caso de algoritmos de múltiplos padrões (como o *Aho-Corasick*), foi necessário um tratamento especial. É possível, entretanto, tirar proveito de ainda mais paralelismo que o que fizemos, e daremos mais detalhes sobre isso na parte de possíveis melhorias.

2.3.2 Pré Processamento:

Para evitar que múltiplas threads realizem o mesmo pré processamento sobre um mesmo padrão, chegamos à conclusão de que seria necessário realizar todo o pré processamento requerido pelos algoritmos usados na instância antes de fazer a distribuição das threads, assim, evitamos o custo redundante de realizar a mesma computação em todas as linhas de todos os textos. Também garantimos que todo pré processamento realizado e acessado pelas threads não fosse alterado em tempo de execução, para evitar condições de corrida.

2.3.3 Estruturas de dados implementadas e utilizadas:

Para um uso efetivo de threads, foi necessário que implementássemos estruturas de dados que lidassem com o envio dos parâmetros para as threads e com o retorno. Fizemos questão que apenas o necessário embarcasse na estrutura, e para um melhor desempenho demos total preferência para os tipos e as funções de C.

Porém, para algumas funções no parsing, pré processamento e gerenciamento das threads, achamos adequado fazer uso de estruturas da biblioteca padrão de C++, como “*vector*” e “*queue*”. Por preencherem uma pequena parte da ferramenta se comparada à execução dos algoritmos, achamos razoável o uso de tais estruturas.

Também achamos interessante implementar a ferramenta de tal maneira que melhorias futuras não sejam difíceis de se implementar, separando os algoritmos e as funções principais por arquivo.

2.3.4 Parsing:

Todo o parsing dos comandos e argumentos da CLI foi implementado utilizando a função “geotpt” da biblioteca GNU, além de uma estrutura de dados para organizar as opções e os parâmetros enviados à ferramenta.

2.3.5 Leitura do texto

Sabemos que um dos maiores gargalos em qualquer programa moderno é o tempo de espera limitado por leitura, principalmente de discos rígidos. O uso de threads pode aliviar um pouco essa penalidade, mas sua existência ainda permanece.

Portanto, nos inspiramos na ferramenta GREP, e decidimos realizar a leitura do texto linha por linha. Isso nos possibilitou algumas vantagens, como:

- Apenas a linha precisa ser alocada na memória principal.
- Podemos retornar informações baseadas nas linhas, como o número de ocorrências em uma linha específica.

Porém, isso se torna um problema em arquivos com linhas muito grandes. Na parte de melhorias possíveis, falaremos um pouco mais sobre o que poderia ter sido feito a respeito.

2.3.6 Alfabeto:

Por praticidade, decidimos utilizar o alfabeto ASCII.

2.3.7 Limitações Conhecidas:

Por conta do uso de threads, as saídas da ferramenta, quando existem mais de uma thread em execução, são assíncronas. Isto é, se existem duas threads lendo dois arquivos de texto diferentes, ou procurando padrões diferentes no mesmo arquivo de texto, nada garante que, por exemplo, a thread 1 irá printar na tela todas as linhas e suas respectivas ocorrências antes que a thread 2 o faça, sendo muito mais provável que os outputs se intercalem. Sabemos que esse detalhe pode se tornar um incômodo, e por isso implementamos a opção -o, que separa os outputs devidamente. É possível resolver esse problema dentro da ferramenta, mas todas as soluções que conseguimos imaginar requerem algum tipo de perda drástica no paralelismo, e julgamos que não valeria a pena.

Por conta da natureza da distância de Levenshtein, é difícil pôr na tela exatamente todas as posições em que o casamento aproximado ocorreu. Por isso, os resultados da nossa função que printa as linhas e suas ocorrências quando lidando com algoritmos de casamento aproximado pode ser estranho.

No momento, a ferramenta escolhe o mesmo algoritmo para todos os textos em que um padrão é buscado. Porém, fizemos o código de uma certa forma de que alterações futuras nesse detalhe não sejam difíceis.

A opção "-m, –max" que para a execução assim que um número máximo de ocorrências é encontrado, não funciona com o algoritmo Aho-Corasick. Caso seja necessário encontrar um número máximo de ocorrências para todos os padrões de uma lista de padrões, recomendamos o uso da opção -f para forçar a ferramenta a utilizar os algoritmos de padrão único.

2.4. Opções Extras

As opções oferecidas pela ferramenta são:

-h, --help; -e, --edit; -a, --algorithms; -c, --count; -s, --statistics;
-p, --pattern; -m, --max; -i, --ignore; -n, --show; -f, --force; -o, --output;

Um detalhamento sobre o uso de cada uma delas pode ser encontrado tanto no README do projeto quanto no uso do comando "-h, –help" na CLI.

Fizemos o possível para que nenhuma opção interfira com o funcionamento de outra, a não ser no caso supra-citado.

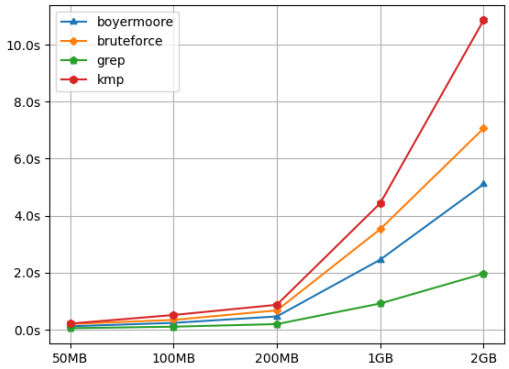
É importante citar que alguns comandos podem ajudar o desempenho, enquanto outros podem atrapalhar.

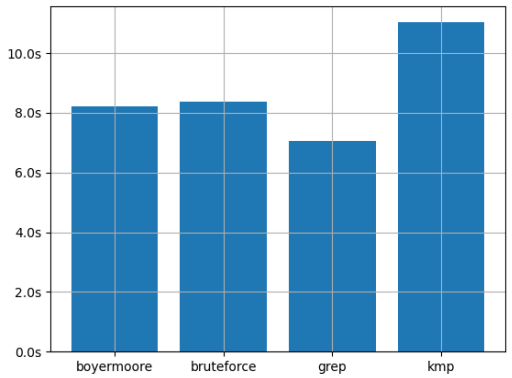
3. Testes e Resultados

3.1. Casamento Exato

Caso Teste 1:
Único padrão curto,
Tamanhos variáveis de arquivos de texto.

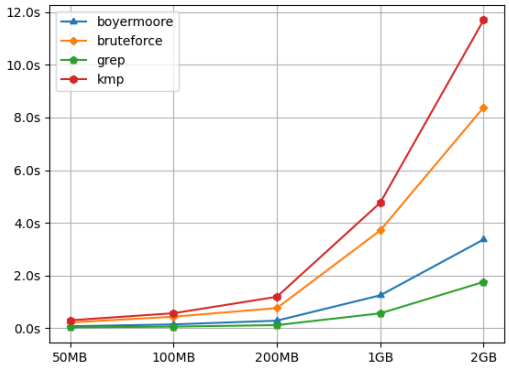
O uso de threads para leitura de múltiplos arquivos

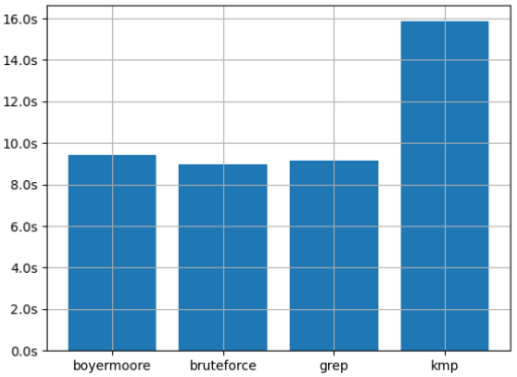




Caso Teste 2:
Único padrão médio,
Múltiplos tamanhos de arquivos.

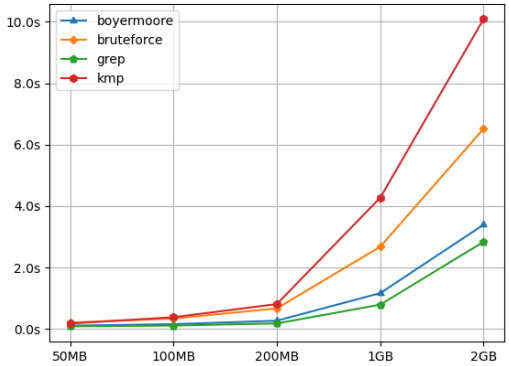
O uso de threads para leitura de múltiplos arquivos

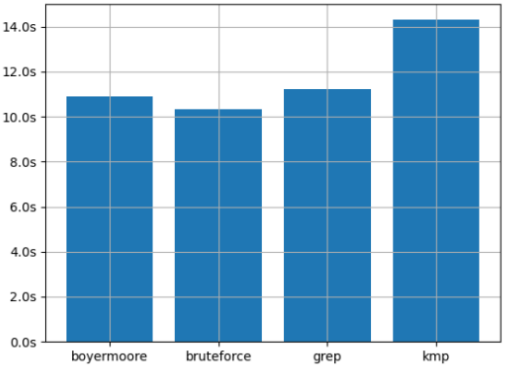




Caso Teste 3:
Único padrão longo,
Tamanhos variáveis de arquivos.

O uso de threads para leitura de múltiplos arquivos

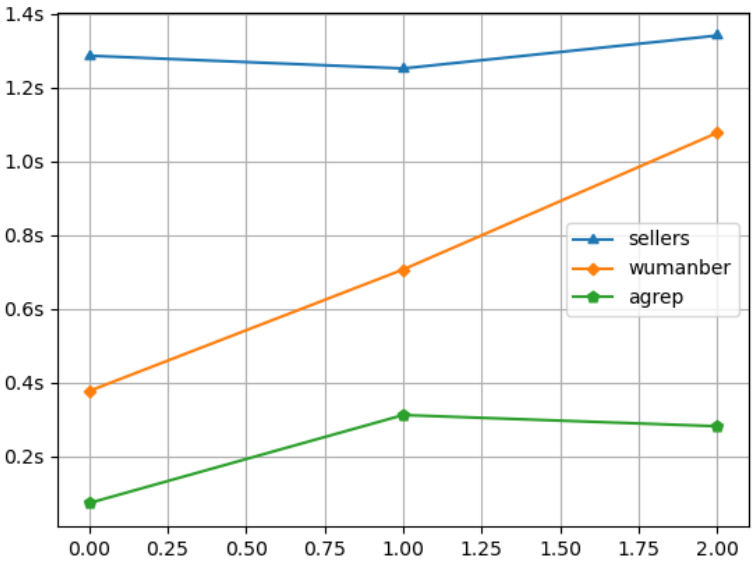




3.2. Casamento Aproximado

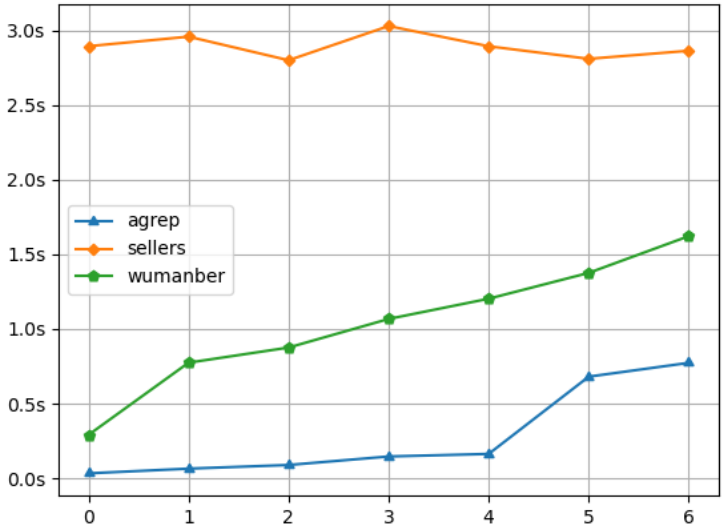
Caso Teste 4:
Único padrão curto,
Único arquivo (50MB),
Múltiplas distâncias.

Gráfico Segundos por Distância



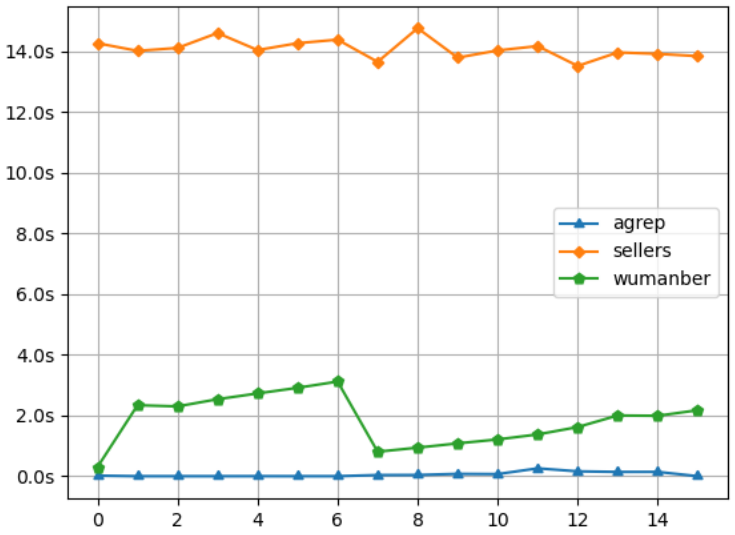
Caso Teste 5:
Único padrão curto,
Único arquivo (50MB),
Múltiplas distâncias.

Gráfico Segundos por Distância



Caso Teste 6:
Único padrão longo,
Único arquivo (50MB),
Múltiplas distâncias.

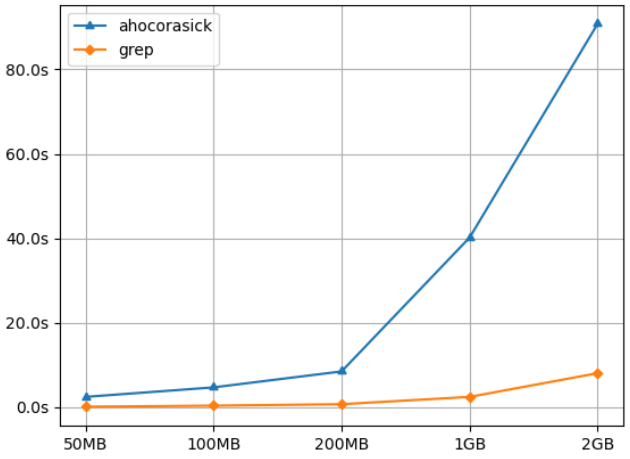
Gráfico Segundos por Distância

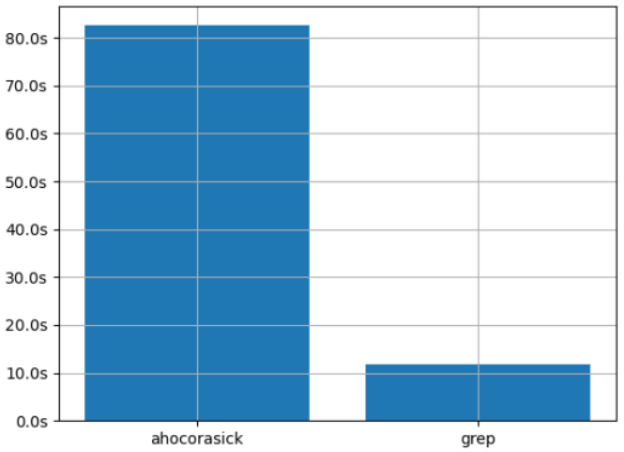


3.3. Casamento Exato – Múltiplos padrões

Caso Teste 7:
500 padrões, de tamanhos aleatórios.
Tamanhos variáveis de arquivos.

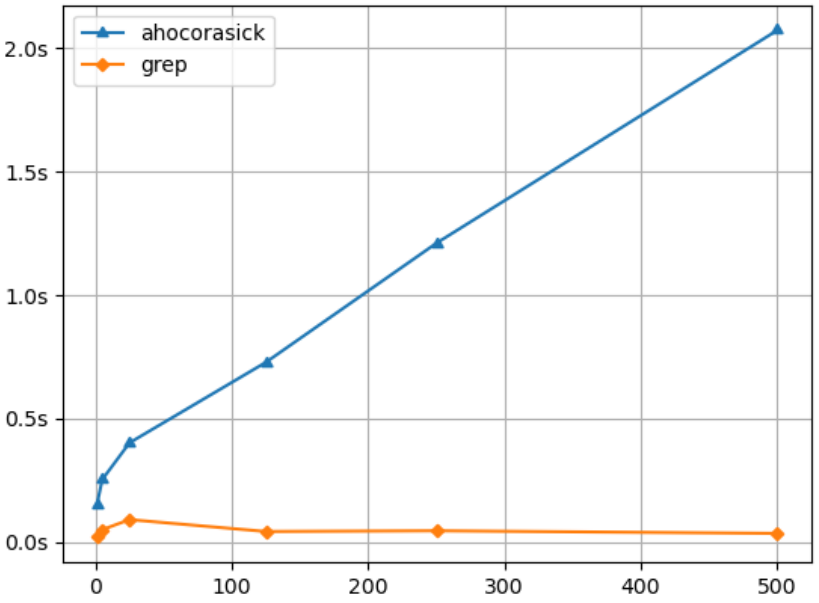
O uso de threads para leitura de múltiplos arquivos





Caso teste 8:
Múltiplas quantidades de padrões (1, 5, 25, 125, 250, 500),
Único arquivo (50MB).

Gráfico segundos por # padrões



4. Conclusões e Possíveis Melhorias

Com base nos nossos testes, podemos confirmar que o uso de threads foi uma melhoria positiva no desempenho da nossa aplicação, que chegou a apresentar resultados superiores ao ***grep*** ao processar múltiplos arquivos. Porém, é possível ir além. Decidimos criar threads para cada padrão, em cada texto, mas é possível criar uma thread para cada **linha,** mitigando o gargalo do tempo de leitura. Com isso, porém, diversos problemas de concorrência e uso exaustivo da memória poderiam surgir, por isso, devido à complexidade da ideia e tempo limitado para a realização do projeto, decidimos evitar e implementar algo mais simples.

Quando os textos são consideravelmente grandes (>50MB), o tempo que é "gasto" ao preparar as threads e os retornos é muito menor (em ordens de magnitude) que o tempo de processamento realizado pelo algoritmo, que acaba por ser praticamente constante. Por isso, fizemos questão de garantir que isso acontecesse, pois é importante evitar perdas desnecessárias no desempenho.

No geral, o desempenho das ferramentas ***agrep*** e ***grep*** são melhores que as da pmt, porém, em alguns casos (como no uso do algoritmo de **Boyer-Moore,** por exemplo) nossa implementação ficou atrás apenas por um fator linear, um bom sinal de que a complexidade está razoável.

Tivemos uma experiência interessante, com desafios que nos ajudaram a entender melhor os algoritmos da disciplina assim como suas implementações. A experiência de montar uma CLI também foi altamente enriquecedora.