Ferramenta PMT

Márcio Barbosa de Oliveira Filho - mbof@cin.ufpe.br

1 Implementação

1.1 Opções Extras

A ferramenta *pmt* descrita neste relatório possui a interface básica definida na especificação do projeto. Além das opções de linha de comando lá explicitadas, ela possui as seguintes opções:

- **-c, --count-only**: A saída é apenas a contagem das ocorrências dos padrões. Como evita operações de entrada e saída, essa opção torna a ferramenta mais eficiente.
- **-s, --show-alignment**: Aplicável apenas a buscas aproximadas. Exibe *um* alinhamento possível para cada ocorrência dos padrões. Utiliza o algoritmo Sellers.
- **-a, --algorithm**: Permite ao usuário especificar o algoritmo de busca a ser utilizado. Os valores possíveis são **u** (Ukkonen), **se** (Sellers) e **w** (WuMamber) para buscas aproximadas e **k** (Kmp), **sh** (ShiftOr), **b** (BoyerMoore) e **a** (AhoCorasick) para buscas exatas.

Nenhuma das configurações acima é obrigatória. As duas primeiras opções são desabilitadas por padrão e a terceira, quando não especificada, tem seu valor escolhido automaticamente pela ferramenta. Essa escolha é feita de acordo com parâmetros considerados relevantes para o desempenho dos algoritmos e determinados através de experimentos (detalhados mais adiante).

1.2 Estratégia de Combinação de Algoritmos para Busca Exata

Através dos testes realizados pudemos concluir que o algoritmo *Boyer-Moore* tem o melhor desempenho para padrões *maiores*, sendo isso ainda mais perceptível quando textos *grandes* ($>$ 500MB) são utilizados. No entanto, ele possui uma perda significativa de desempenho quando padrões e alfabetos pequenos são utilizados. De maneira geral, um alfabeto pequeno representa um desafio para a maioria dos algoritmos implementados porque aumenta a quantidade de casamentos parciais. No entanto, esse parâmetro exerce maior influência sobre o *BoyerMoore* porque tende a aproximá-lo de seu pior caso, o qual é equivalente

ao da força bruta. Por outro lado, consideramos que os cenários capazes de explorar essa característica do *BoyerMoore* não são comuns. Além disso, toda a informação que temos sobre o alfabeto é aquela inferida através do padrão, de modo que não vale a pena sermos muito restritivos a respeito desse parâmetro sob pena de não tirarmos proveito da eficiência do *BoyerMoore*.

Existem três situações em que procuramos evitar o uso do *BoyerMoore*. A primeira delas é quando buscamos mais de um padrão simultaneamente. Nesse caso, fazemos uso do *AhoCorasick*. A segunda, como mencionamos, é quando o usuário fornece um padrão pequeno. Nessa situação utilizamos a força bruta para padrões unitários e o *ShiftOr* para demais padrões pequenos. O terceiro caso em que evitamos o uso do *BoyerMoore* está relacionado ao tamanho do alfabeto do padrão. Evitamos o *BoyerMoore* quando avaliamos que o alfabeto do padrão é pequeno e o tamanho do padrão é no máximo 64. Nesse caso nós utilizamos nossa implementação mais rápida do *ShiftOr*, que chamamos de *ShiftOr2*. Ela tem um suporte limitado ao tamanho do padrão porque armazena as máscaras usadas pelo algoritmo em inteiros de 64 *bits*. A nossa implementação do *ShiftOr* com suporte a qualquer tamanho de padrão possui um desempenho consideravelmente pior.

Resumimos a estratégia de combinação de algoritmos para busca exata a seguir:

- Se mais de um padrão é fornecido, usamos o *AhoCorasick*,

Seja m o tamanho do padrão a ser procurado e α o tamanho do alfabeto do padrão.

- Se $m = 1$ usamos o algoritmo de força bruta,
- Se $m \leq 64$ e $\alpha \leq 5$ usamos o *ShiftOr*, *(Note que isso é sempre verdade se $m \leq 5$.)*
- Caso contrário: usamos o *BoyerMoore*

1.3 Estratégia de Combinação de Algoritmos para Busca Aproximada

No caso da busca aproximada, nossa estratégia de combinação de algoritmos teve como principal objetivo utilizar o algoritmo *Ukkonen* sempre que possível. Devido à construção do autômato, o *Ukkonen* tem um tempo de preprocessamento bastante elevado quando comparado aos demais. No entanto, seu processamento do texto é bastante eficiente e, quando esse texto é suficientemente grande, o tempo de preprocessamento pode ser compensado.

Como o crescimento do número de estados do autômato é exponencial, precisamos ter muita atenção aos parâmetros que o influenciam. No entanto, fazemos uma consideração semelhante a que fizemos em relação ao *BoyerMoore*. Ou seja, acreditamos que as restrições que fizemos são suficientes para impedir casos ruins em geral, embora eles ainda possam acontecer. De maneira geral,

usamos o *Ukkonen* quando o padrão é pequeno ou quando ele é razoavelmente grande mas o erro máximo é pequeno.

Quando decidirmos não utilizar o *Ukkonen*, precisamos escolher entre os algoritmos *Sellers* e *WuMamber*. O algoritmo *Sellers* é influenciado apenas pelo tamanho do padrão, enquanto que o *WuMamber* apenas pelo erro máximo. De maneira geral, identificamos que o *WuMamber* é cerca de duas vezes mais rápido que o *Sellers* quando esses parâmetros são semelhantes. Por isso, utilizamos o *WuMamber* quando o tamanho do padrão é pelo menos metade do erro máximo e o tamanho do padrão é no máximo 64 (pelas mesmas razões do *ShiftOr*). Para os demais casos utilizamos o *Sellers*.

Ressaltamos que apenas a nossa implementação do *Sellers* suporta a reconstrução de alinhamentos. Por isso ele é o algoritmo utilizado sempre que o usuário escolhe essa opção.

Se mais de um padrão é fornecido, a escolha do algoritmo é feita individualmente.

Resumimos a estratégia de utilização dos algoritmos para busca aproximada abaixo.

Seja m o tamanho do padrão a ser procurado, e_{max} o erro máximo e T o tamanho total em bytes do(s) arquivo(s) de texto(s).

- Se $T > 100$MB e ($m < 20$ ou ($m < 70$ e $e_{max} \leq 5$) ou ($m < 100$ e $e_{max} \leq 3$)) usamos o *Ukkonen*,
- Se $(e_{max}/2) < m \leq 64$ usamos o *WuMamber*,
- Caso contrário usamos o *Sellers*

1.4 Estruturas de Dados e Outras Decisões Relevantes

A seguir fazemos um breve registro de algumas decisões de implementação que tiverem impacto relevante no desempenho geral da ferramenta.

- **Lista de padrões no autômato do AhoCorasick**. Implementamos uma lista duplamente ligada[1] de modo a permitir a operação de concatenação exigida pelo algoritmo em tempo $O(1)$.
- **Árvore ternária de colunas e autômato do Ukkonen**. Cada uma das folhas dessa árvore representa uma coluna da matriz de PD. No entanto, armazenamos essas folhas em separado, aproveitando-as como nós do autômato. Dessa maneira, diminuimos o tamanho total da árvore e evitamos colocar em todos os nós informações pertinentes apenas às folhas. Procuramos, com isso, economizar memória.[2]
- **Leitura da entrada**. Os arquivos que contém os textos a serem analizados são processados em blocos de 16MB. Verificamos que um tamanho

[1] Ver *src/auxiliar/AhoList.{h,cpp}*

[2] Ver *src/auxiliar/UkkonenTreeNode.{h,cpp}* e *src/auxiliar/UkkonenAutomatonNode.{h,cpp}*.

menor de bloco aumenta consideravelmente o tempo de execução da ferramenta, enquanto que um valor maior não traz maiores ganhos.

- **Determinação do alfabeto**. O alfabeto é definido como o conjunto das letras dos padrões mais um símbolo que não ocorre em nenhum deles (caracter nulo, ASCII 0). Dessa forma, os caracteres ASCII são mapeados nesses valores, em particular, todo caracter que não ocorre no padrão é mapeado para o caracter nulo. Essa estratégia favorece os algoritmos que dependem *diretamente* do alfabeto, como o *AhoCorasick* e o *Ukkonen*.

- **Reconstrução de alinhamento**. Para possibilitar ao algoritmo *Sellers* reconstruir os alinhamentos das ocorrências encontradas armazenamos apenas as $(m + e_{max})$ últimas colunas da matriz de programação dinâmica. Com isso a opção de imprimir os alinhamentos não causa grande impacto no consumo de memória da aplicação. Quando essa opção está desligada, utilizamos uma implementação do *Sellers* que armazena apenas as 2 últimas colunas.

2 Testes e Resultados

Todos os testes conduzidos, tanto da ferramenta de modo geral quanto dos algoritmos individualmente, foram controlados por *scripts* escritos em *python*. A função desses *scripts* era a de invocar aplicações através da linha de comando com parâmetros adequados, armazenar e manipular os resultados. A principal métrica utilizada foi a média dos tempos de execução. Os tempos em si foram coletados a partir do tempo *real* de execução fornecido pela ferramenta *time* do *Linux*. A ferramenta *gnuplot* foi utilizada para que os dados pudessem ser visualizados e comparados graficamente.

Os testes que serão descritos a seguir foram conduzidos em um computador com sistema operacional *Ubuntu 14.04*, 4GB de memória *RAM* e processador *Intel Core* i5 2.53 GHz.

2.1 Testes para Escolha dos Algoritmos de Busca Exata

Para escolher quais algoritmos seriam utilizados na busca exata e em quais situações, vários testes foram realizados com diferentes configurações de padrões e textos. Em todos eles, os algoritmos tinham como saída apenas a quantidade de ocorrências dos padrões. Para fins de comparação, incluímos também a ferramenta *grep* nos testes. Para que ela tivesse saída compatível com os demais algoritmos nós a executamos da seguinte maneira: *grep -e pattern -o -F textfile | wc -l*. Esse comando resulta na contagem da quantidade de ocorrências encontradas. Desse modo, os testes também serviram para identificar erros de implementação.

Durante os testes duas implementações do *ShiftOr* foram executadas. A que chamamos de *ShiftOr2* suporta apenas padrões de tamanho até 64, pois utiliza

inteiros de 64 *bits* como máscaras. Já a que chamamos simplesmente de *ShiftOr* é geral e suporta padrões de todos os tamanhos.

2.1.1 Teste 1: Texto em inglês e padrões de tamanhos variados

No primeiro teste, nós utilizamos um arquivo de texto de 1.1GB[3,4] e padrões com tamanhos entre 1 e 90. Tanto o texto quanto os padrões estavam em inglês. A Figura 1 registra o resultado desse teste, mostrando, para cada algoritmo, o tempo médio de execução em função do tamanho do padrão.

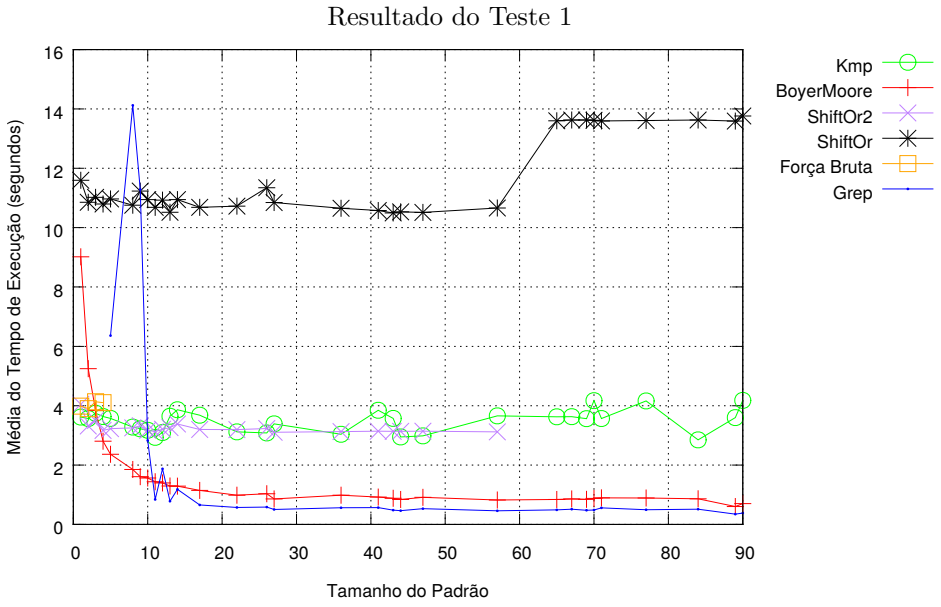

Figura 1: Cada algoritmo foi individualmente testado com um texto em inglês de 1.1G. Foram procurados padrões com tamanhos entre 1 e 90. A ferramenta grep também foi executada.

Constatamos que o algoritmo *ShiftOr* generalizado não consegue bons resultados. Em particular, quando o tamanho do padrão é maior que de 64 torna-se necessário mais de 1 inteiro para representar cada máscara utilizada pelo algoritmo. Isso tem um impacto perceptível no tempo de execução do *ShiftOr*.

O *Kmp* mostra-se relativamente estável, com tempos de execução em torno de 4 segundos. O *BoyerMoore*, por sua vez, se torna bastante eficiente para padrões maiores.

A seguir, na Figura 2, nós restringimos o gráfico da Figura 1 a padrões com tamanho até 15. Dessa forma podemos obter uma melhor visualização desses casos.

[3]Disponível em *http://pizzachili.dcc.uchile.cl/texts/nlang/english.1024MB.gz*.

[4]O arquivo original continha um caracter nulo no meio do texto. Nós o retiramos para evitar comportamentos estranhos que interferissem nos testes.

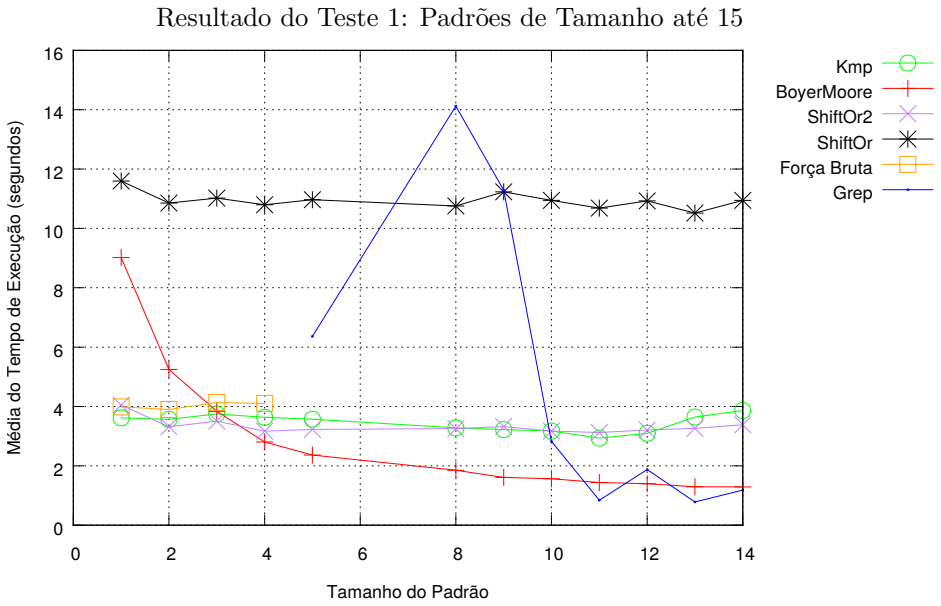

Figura 2: O mesmo teste, mas com padrões de tamanho até 15.

A Figura 2 sugere que o *BoyerMoore* é o mais eficiente com padrões de tamanho pelo menos igual a 4. Com padrões menores os demais algoritmos possuem tempo de execução simulares, com exeção do *ShiftOr* generalizado. Cortamos os resultados do *grep* para padrões com tamanho inferior a 4 porque o tempo de execução era muito alto e dificultava a visualização do gráfico.

Ainda utilizando o mesmo arquivo de texto, experimentamos padrões ainda maiores para testar se o comportamento dos algoritmos se mantinha dentro do esperado. Para isso, no entanto, excluímos o algoritmo *ShiftOr*, por ser muito lento e também o *ShiftOr2*, por não suportar o tamanho dos padrões. O resultado pode ser visto na Figura 3.

Quando comparamos os tempos de execução registrados nos gráficos das Figuras 1 e 2 com os da Figura 3, percebemos que tanto o *Kmp* quanto o *BoyerMoore* mantiveram-se praticamente estáveis. O *BoyerMoore*, portanto, continuou a mostrar-se superior ao lidar com padrões maiores.

2.1.2 Teste 2: Texto e Padrões Aleatórios e Alfabetos de Tamanhos Variados

Para encerrar os testes com algoritmos para busca exata, nós investigamos textos e padrões uniformemente gerados e com alfabetos de diferentes tamanhos. Nosso principal objetivo era definir a estratégia para lidar com padrões pequenos, uma vez que já estávamos convencidos de que o *BoyerMoore* tratava eficientemente os maiores.

Realizamos dois testes com características similares. Procuramos padrões

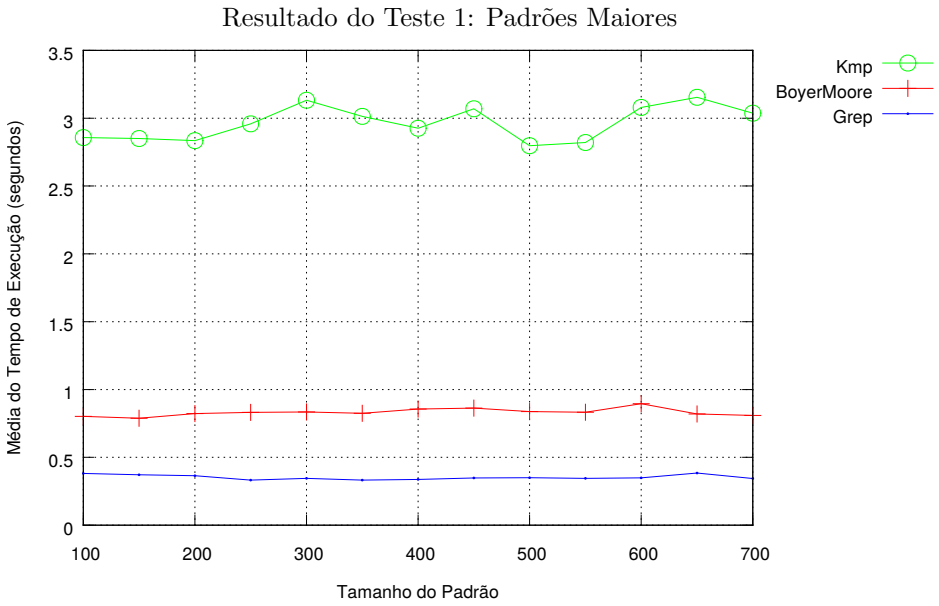

Figura 3: Padrões maiores foram buscados no mesmo arquivo de texto do teste anterior. Esses padrões eram frases em inglês.

uniformemente gerados com tamanhos entre 1 e 20 em um arquivo de texto de 1.5GB também uniformemente gerado. Utilizamos alfabetos de tamanho 5 (primeiro teste) e 10 (segundo teste). Os resultados estão registrados nas Figuras 4 e 5. Excluímos o algoritmo *ShiftOr* generalizado dos testes e, no primeiro teste, também a ferramenta *grep*. Ela apresentou alguns tempos bastante elevados e dificultaria a visualização gráfica dos resultados.

Comparando-se os gráficos da Figura 4 com o da Figura 2 percebemos que o *Kmp* e o *BoyerMoore* quase dobraram seus tempos de execução. Em parte, isso poderia ser justificado pelo aumento do tamanho do arquivo (de 1.1GB para 1.5GB). No entanto, note que, por exemplo, o *ShiftOr2* manteve seu tempo de execução praticamente inalterado. Por isso, concluímos que essa perda considerável de desempenho é causada, principalmente, pela diminuição do tamanho do alfabeto. Pudemos confirmar isso ao observar o teste registrado na Figura 5, em que os algoritmos foram alimentados com um arquivo de mesmo tamanho (1.5GB), mas gerado a partir de um alfabeto (duas vezes) maior. Novamente essa variação no tamanho do alfabeto não teve influência no *ShiftOr2*, como era esperado[5].

Ainda assim, verificamos que, em ambos os cenários, o *BoyerMoore* é bastante eficiente para padrões maiores. O *ShiftOr2*, por sua vez, se mostrou estável, apresentando bom desempenho onde o *BoyerMoore* e o *Kmp* aumenta-

[5]Isso é verdade porque estamos apenas contando as ocorrências. Se elas fossem reportadas, naturalmente *todo* algoritmo seria afetado pela diminuição do alfabeto.

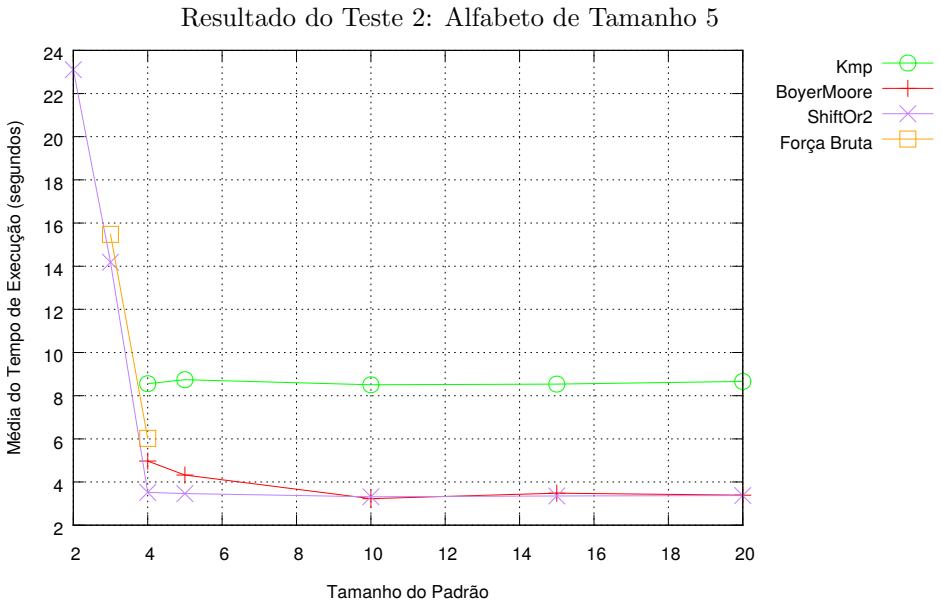

Figura 4: Teste feito buscando-se padrões uniformemente gerados em um arquivo também uniformemente gerado de 1.5GB. O alfabeto utilizado tem tamanho 5.

ram seus tempos de execução.

Notamos que a ferramenta *grep* também tem seu desempenho bastante afetado pela diminuição do alfabeto. Tanto que tivemos que remover alguns de seus resultados para não comprometer a visualização do gráfico. No entanto, ressaltamos que essa comparação não é completamente justa porque o *grep* sempre imprime as ocorrências dos padrões, enquanto nossos algoritmos apenas imprimem a quantidade dessas ocorrências. Ainda assim, o *grep* tem um desempenho muito bom e estável para padrões maiores.

2.1.3 Conclusões

Com os testes que apresentamos nessa seção, pudemos tirar as seguintes conclusões:

- O tempo de execução para padrões muito pequenos é bastante variável. Claro que, para padrões unitários, nenhum algoritmo faz menos operações que o força bruta. Para padrões até tamanho 5 percebemos que o *ShiftOr2* é bastante estável e eficiente, mesmo em diferentes cenários.

- A partir de padrões de tamanho 5 o *BoyerMoore* é, em geral, mais eficiente. Vimos cenários em que o *ShiftOr2* teve tempo de execução compatível com o *BoyerMoore* mesmo para padrões maiores. Mas mesmo nesses casos a diferença foi pouca, de modo que parece ser melhor escolher o *BoyerMoore* como o algoritmo para padrões grandes.

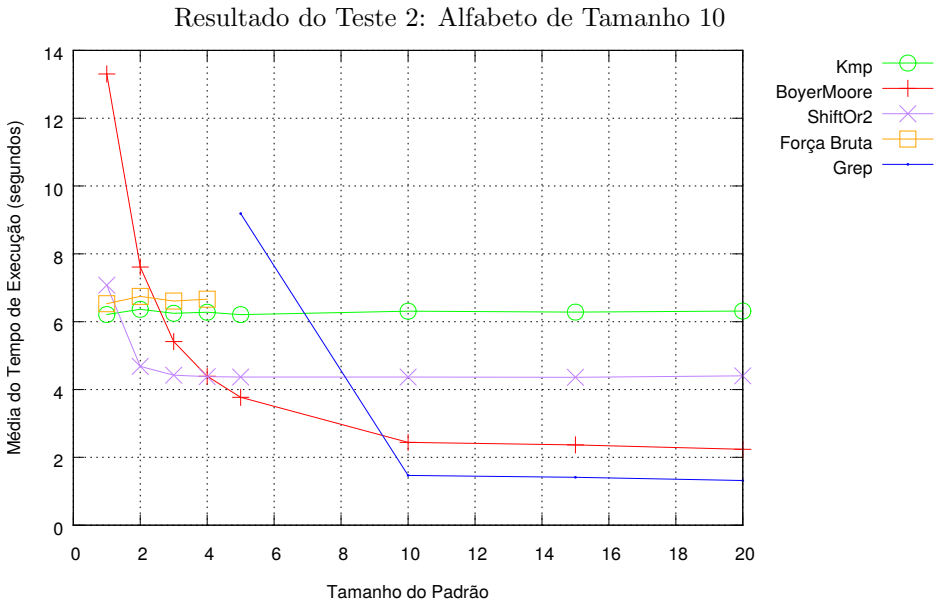

Figura 5: Teste feito buscando-se padrões uniformemente gerados em um arquivo também uniformemente gerado de 1.5GB. O alfabeto utilizado tem tamanho 10.

- Vimos, no entanto, que não só o tamanho do padrão importa. Também o tamanho do alfabeto e o número de ocorrências e casamentos parciais influencia o *Kmp* e o *BoyerMoore*. De modo geral, não podemos prever esses cenários sem conhecer o alfabeto do texto. Podemos, no entanto, utilizar o tamanho do alfabeto do *padrão* como heurística para ajudar a determinar quando 'desistir' de usar o *BoyerMoore*. Isso não é ideal, uma vez que não leva em conta a heurística do mau caracter, a qual pode aumentar bastante o desempenho do algoritmo mesmo para um alfabeto pequeno. Além disso, vimos que, nos casos em que o *BoyerMoore* é afetado pelo tamanho do alfabeto, apenas o *ShiftOr2* é uma opção a ser considerada. Por essa razão, vamos desistir do *BoyerMoore* em função do tamanho do alfabeto *somente se* o tamanho do padrão for suportado pelo *ShiftOr2* (≤ 64). Com isso, naturalmente, não evitamos todos os cenários ruins do *BoyerMoore*, nem o seu pior caso, em particular. No entanto, consideramos que cobrimos os cenários mais comuns e menos artificiais.

2.2 Testes para Escolha dos Algoritmos de Busca Aproximada

De maneira semelhante ao caso da busca exata, realizamos testes variados para determinar quais algoritmos utilizar e em quais situações. Para isso consideramos implementações dos algoritmos *WuMamber*, *Ukkonen* e *Sellers*.

Preparamos três cenários para comparar os algoritmos. Neles procuramos forçar o pior caso de todos os algoritmos, utilizando padrões de todos os tamanhos $m \in [1..20]$ e erro máximo $e = m - 1$. Em cada teste utilizamos um arquivo com texto em inglês (100MB no primeiro, 500MB no segundo e 1.1GB no terceiro). Apresentamos os resultados nas Figuras 6, 7 e 8 a seguir.

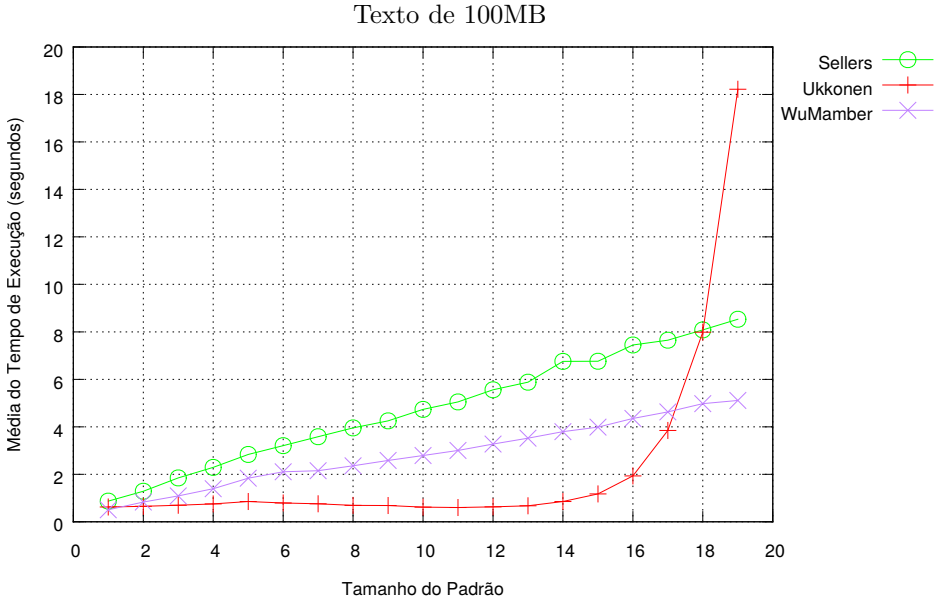

Figura 6: Texto em inglês de 100MB. Para cada tamanho m do padrão, o erro máximo foi definido como $m - 1$.

A partir desses cenários pudemos perceber claramente o comportamento linear dos algoritmos *WuMamber* e *Sellers*, bem como o comportamento exponencial do *Ukkonen*. Pudemos perceber também que, de maneira geral, quando o arquivo de texto é grande, o preprocessamento pesado feito pelo *Ukkonen* é compensado pela leitura rápida do texto. No entanto, existe sempre um ponto em que o tempo de execução do *Ukkonen* 'explode'. Antes desse ponto, vemos que o *Ukkonen* é muito eficiente, considerando que os testes exploram casos ruins dos três algoritmos.

Outro ponto importante a ser percebido é que o *Sellers* tem quase o dobro do tempo de execução quando comparado com o *WuMamber*. Ou seja, o tempo de execução do *Sellers* para um padrão de tamanho m é similar ao tempo de execução do *WuMamber* com erro máximo $2m$.

Nos testes realizamos, o *Ukkonen* deu um salto no tempo de execução com padrões de tamanho em torno de 20. Na prática, padrões maiores podem ser suportados desde que o erro máximo e o tamanho do alfabeto não sejam muito grandes.

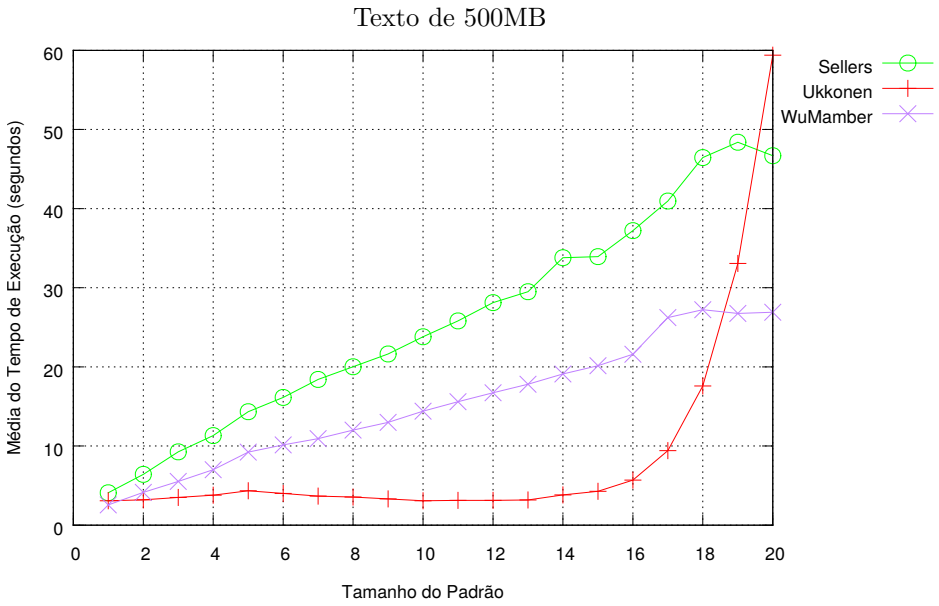

Figura 7: Texto em inglês de 500MB. Para cada tamanho m do padrão, o erro máximo foi definido como $m - 1$.

2.2.1 Conclusões

Com os testes apresentados e outros similares que não registramos aqui, pudemos chegar às seguintes conclusões:

- O algoritmo *Ukkonen* é bastante eficiente para arquivos grandes, desde que utilizado para com um tamanho de padrão e erro máximo que não provoque uma 'explosão' do número de estados do autômato.
- O algoritmo *WuMamber* é mais eficiente do que o *Sellers* para parâmetros semelhantes
- A busca aproximada é consideravelmente mais custosa que a busca exata. Em particular, não conseguimos uma implementação razoável para o algoritmo *WuMamber generalizado*, isto é, com suporte a qualquer tamanho de padrão.

2.3 Teste da Ferramenta

Nessa seção vamos apresentar alguns testes feitos com a ferramenta em si, quando não especificamos qual algoritmo deve ser utilizado. O ambiente de execução continua o mesmo que já descrevemos.

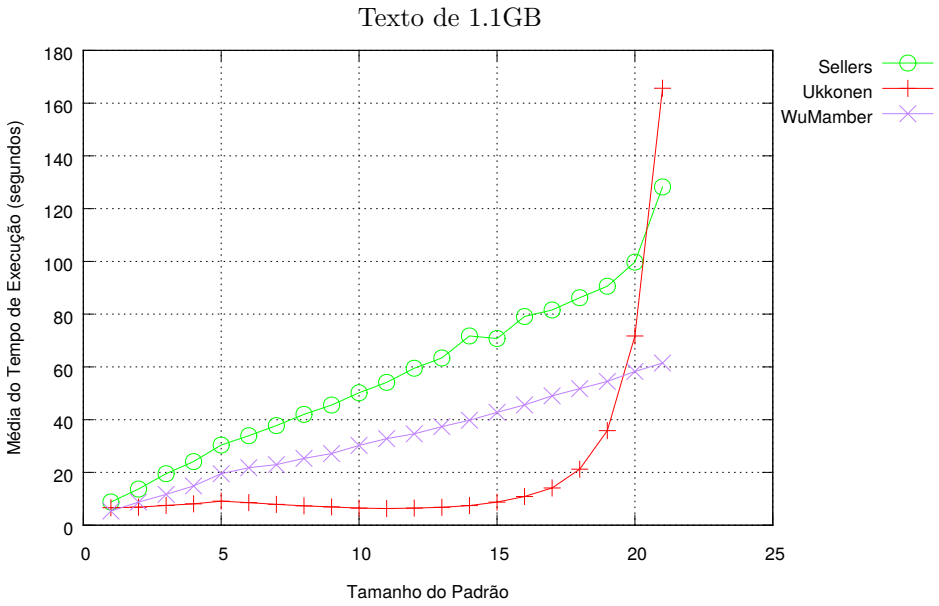

Figura 8: Texto em inglês de 1.1GB. Para cada tamanho m do padrão, o erro máximo foi definido como $m - 1$.

2.3.1 Teste 1: Múltiplos Padrões

O primeiro teste consiste em fornecer à ferramenta vários padrões (através da opção *-p*). Separamos esses padrões em três arquivos:

- **p1**: Arquivo contendo padrões com muitas ocorrências no texto,
- **p2**: Arquivo contendo padrões com poucas ocorrências no texto,
- **p3**: Arquivo grande contendo muitos padrões.

No primeiro teste procuramos cada um desses arquivos de padrões em um texto na língua inglesa com 1GB. Testamos nossa ferramenta de duas maneiras diferentes: com a *flag* -c ligada e desligada. Além disso, incluímos no teste também a ferramenta *grep*, também de duas maneiras. Na primeira, utilizamos o comportamento padrão da imprimir as ocorrências e depois contamos o número de linhas impressas (comando: *grep -o -f pattern-file text-file — wc -l*). A segunda maneira foi utilizar a *flag* -c da ferramenta *grep* para impedir que ela imprimisse ocorrências. Essa *flag*, no entanto, não conta extamente o que queremos. Ela contabiliza o número de *linhas* em que *alguma* ocorrência foi encontrada. Apenas usamos o resultado dessa chamada como referência. O resultado desse teste é apresentado a seguir.

Nesse teste, os aquivos p1 e p2 tinham 23 padrões e menos de 300 *bytes* cada. Já o arquivo p3 tinha 10000 padrões e 1MB.

Tabela 1: Resultado do Teste 1

Arquivo	N. de ocorrências	**pmt -c (tempo)**	**pmt (tempo)**	**grep (tempo)**	**grep -c (tempo)**
p1	2.10^8	25s	1min 40s	>5min	14s
p2	10^4	24s	25s	13s	13s
p3	2.10^5	22s	20s	>5min	>5min

Resultado da busca de cada um dos padrões em p1, p2 e p3 em um arquivo de texto com 1GB. p1 e p2 continham 23 padrões e p3 continha 10000.

Observamos que a quantidade de padrões e o tamanho total deles não tem grande influência quando apenas contamos as ocorrências. Isso já era esperado porque nesse caso utilizamos o algoritmo *AhoCorasick*. Esse algoritmo, por sua vez, só depende dos padrões para uma etapa de preprocessamento (de custo linear), a qual é compensada pelo processamento eficiente do texto. Naturalmente que quando a ferramenta imprime as ocorrências encontradas o tempo de execução é superior. Mas note que, ainda assim, o desempenho da ferramenta só foi sensivelmente degradado quando uma quantidade muito grande de ocorrências teve que ser impressa (mais de 10^8).

2.3.2 Teste 2: Busca Exata de um Único Padrão

Como primeiro teste, executamos a ferramenta *pmt* no modo apenas de contagem (*flag* -c) para procurar padrões de variados tamanhos em um texto na língua inglesa com 1.1GB. O resultado é mostrado na Figura 9 a seguir.

Nesse teste percebemos que nossa ferramenta consegue manter um bom comportamento mesmo com padrões pequenos. Com padrões maiores ela se aproxima do desempenho do *grep*. Os resultados do *grep* estavam acima dos 30 segundos para padrões pequenos. Apenas para não prejudicar a visualização do gráfico nós diminuimos esses valores.

Em seguida, repetimos o teste preliminar que fizemos para investigar o impacto do tamanho do alfabeto nos algoritmos. O resultado pode ser visto na Figura 10 a seguir.

Nesse teste nós utilizamos um arquivo de texto com 1.5GB gerado uniformemente a partir de um alfabeto de tamanho 10. Note que, comparado com a Figura 9 nossa ferramenta apresenta uma perda de desempenho. Isso acontece porque ela escolheu o algoritmo *BoyerMoore* para tratar a maior parte dos padrões.

2.3.3 Teste 3: Busca Aproximada de um Único Padrão

Para testar a busca aproximada, nós utilizamos um arquivo de texto de 1.1GB na língua inglesa. Os padrões utilizados também era palavras ou frases da língua inglesa. Os erros máximos foram gerados aleatoriamente. Utilizamos a ferramenta *agrep* para comparação. Dessa forma, as entradas tiveram que ser

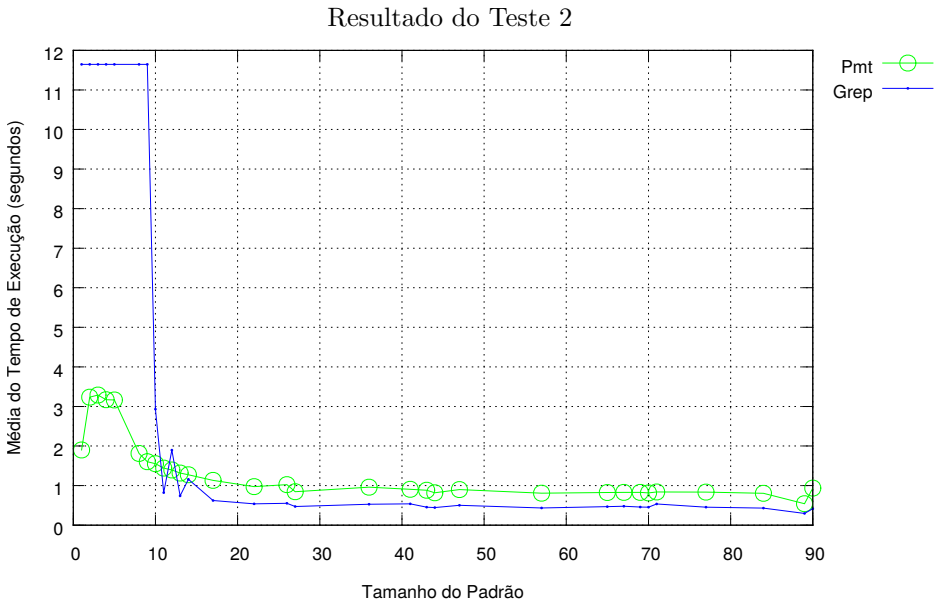

Figura 9: Vários padrões com tamanho entre 1 e 90 foram procurados em um texto de 1.1GB. Tanto o texto quanto os padrões estavam na língua inglesa.

limitadas. Isso porque o *agrep* não aceita padrões maiores que 32 nem erros superiores a 8.

O resultado desse teste pode ser visto nas Figuras 11 e 12, onde os tempos de execução são mostrados em função do tamanho do padrão e do erro máximo, respectivamente.

Embora a busca aproximada seja consideravelmente mais lenta que a busca exata, podemos verificar, nas Figuras 11 e 12, que nossa ferramenta não destoa do comportamento geral do *agrep*. Principalmente quando encaramos o resultado em função do erro máximo, vemos que nossa ferramenta é mais eficiente em vários casos (erros de 3 a 8).

3 Conclusão

Já fizemos observações importantes sobre nossa ferramenta e seus algoritmos ao longo da apresentação dos resultados dos testes realizados. Concluímos nossa análise aqui elencando alguns pontos positivos e negativos da nossa implementação.

Pontos positivos:

- **Algoritmos Online**. Todos os algoritmos implementados são *online*. Isso permite que buscas possam ser realizadas eficientemente mesmo em arquivos muito grande. Em particular, não armazenamos grandes quantidades do texto na memória.

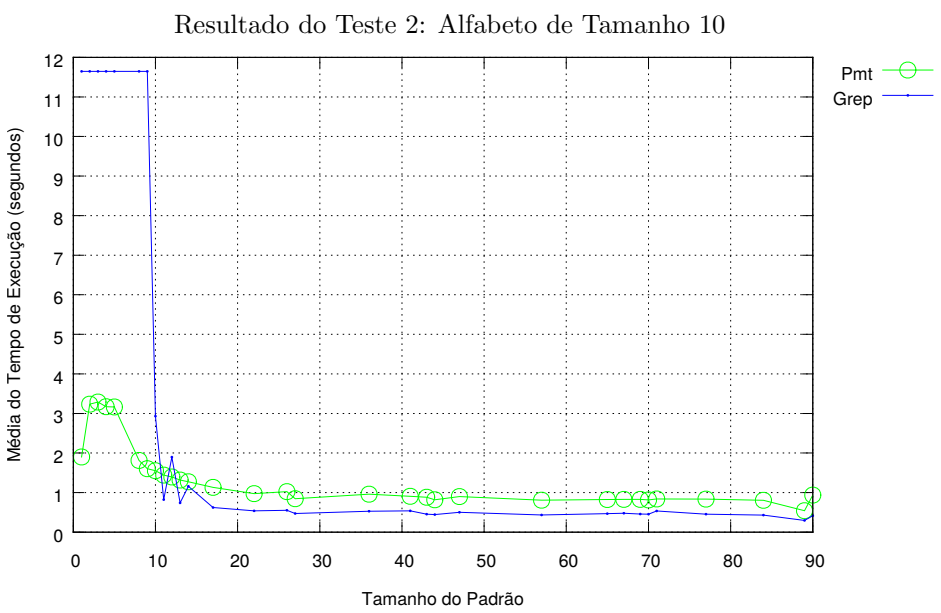

Figura 10: Vários padrões com tamanho entre 1 e 20 foram procurados em um texto de 1.5GB. Tanto o texto quanto os padrões foram gerados aleatoriamente.

- **Combinação de Algoritmos**. Com nossas heurísticas de combinação de algoritmos tentamos aproveitar boas características de todos os algoritmos. Alguns casos ruins, no entanto, ainda podem acontecer, mas acreditamos que são pouco comuns (ver discussão sobre *Ukkonen* e *BoyerMoore*).

- **Saída**. Consideramos que a saída da nossa ferramenta atende a vários interesses. Ela pode apenas mostrar a quantidade de ocorrências encontradas ou, em seu comportamento padrão, exibir um pedaço do texto onde a ocorrência foi encontrada. Além disso, no caso de buscas aproximadas, é possível reconstruir um alinhamento para cada ocorrência.

Pontos negativos:

- **Comparação com grep**. Vimos nos testes que a ferramenta *grep* possui um desempenho superior à nossa em quase todos os cenários. Apenas para padrões pequenos nossa ferramenta se mostra melhor.

- **Busca Aproximada**. Os algoritmos de busca aproximada implementados são, de certa forma, limitados. Tentamos usar sempre que possível o *Ukkonen*, que se mostrou eficiente principalmente para arquivos de texto grande. No entanto, quando não se pode usar esse algortimo, devido à sua perda exponencial de desempenho, nossas implementações dos demais algoritmos também não são boas.

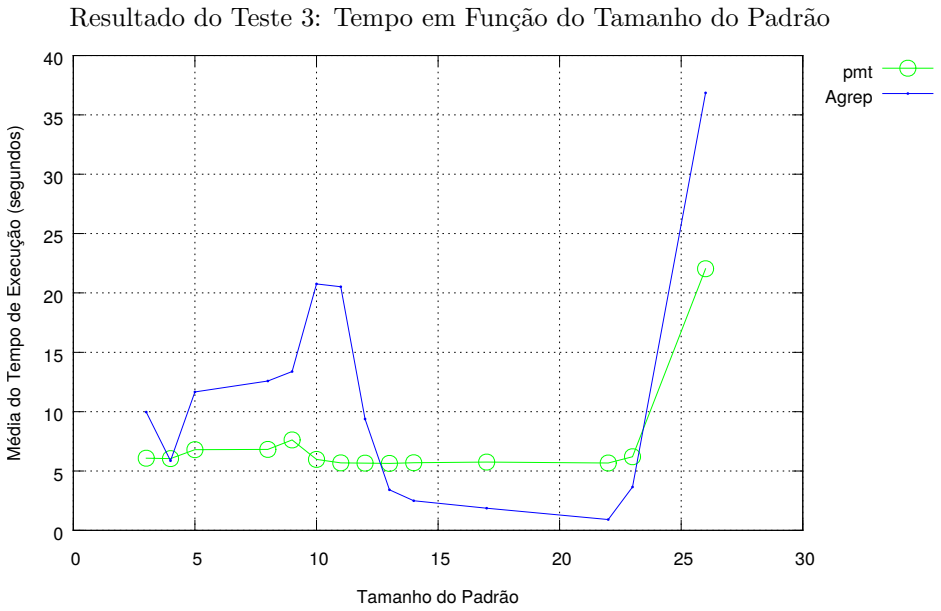

Figura 11: Vários padrões com tamanho entre 3 e 26 foram procurados em um texto de 1.1GB. O gráfico mostra o tempo de execução em função do tamanho dos padrões.

- **Saída**. Consideramos que a saída da nossa ferramenta atende a vários interesses. Ela pode apenas mostrar a quantidade de ocorrências encontradas ou, em seu comportamento padrão, exibir um pedaço do texto onde a ocorrência foi encontrada. Além disso, no caso de buscas aproximadas, é possível reconstruir um alinhamento para cada ocorrência.

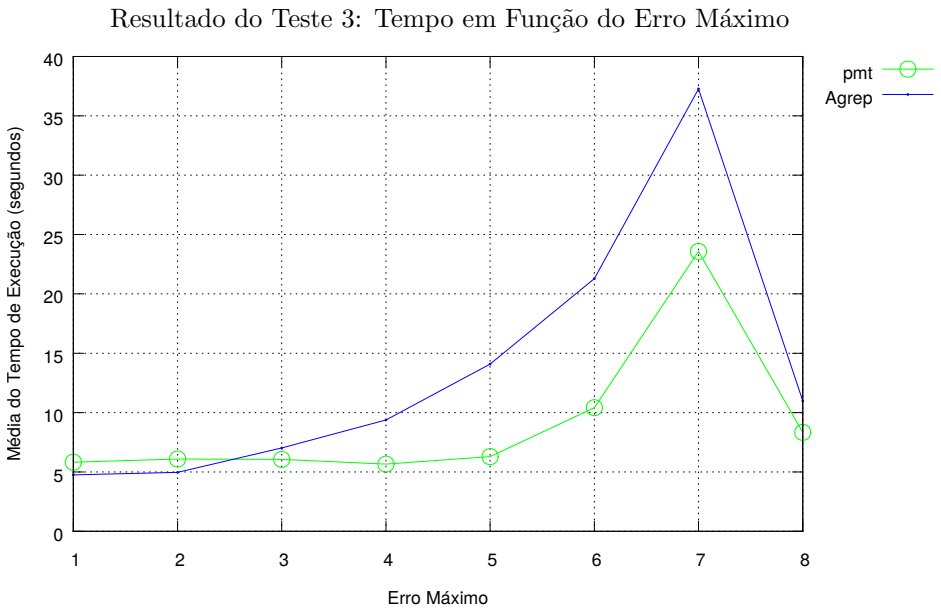

Figura 12: Várias buscas com erro máximo entre 1 e 8 foram realizadas em um texto de 1.1GB. O gráfico mostra o tempo de execuçã em função do erro máximo.