Ferramenta PMT

1. Identificação

Daniel de Azevedo Pacheco - dap5@cin.ufpe.br
Gabriel de Albuquerque Souza Meireles - gasm@cin.ufpe.br

2. Implementação

A leitura do(s) arquivo(s) de texto dado(s) é feita linha por linha. Para cada linha, tenta-se achar a ocorrência de cada padrão e, caso encontre algum match, essa linha será imprimida no terminal com o(s) padrão(ões) destacado(s).
Os alfabetos suportados pelo ***shift_or*** e pelo ***ukkonen*** são delimitados pela tabela ASCII (do valor 32 ao 126). Essa escolha foi feita para poder converter caracteres em inteiros em tempo constante com baixa complexidade, evitando o uso de estruturas de dados como hashmaps. Assim, pode-se usar todas as letras do alfabeto e alguns caracteres especiais. Caracteres acentuados não são suportados por esses algoritmos.
Todos os algoritmos implementados retornam um vetor de pares onde a primeira posição corresponde ao início do match e a segunda posição corresponde ao tamanho do match. Com esse par é possível saber a quantidade total de matches e como destacá-los no terminal.

a. KMP

Inicialmente é pré-calculado as bordas de cada padrão. Para cada linha do texto, executa-se o algoritmo de KMP normalmente, usando uma *lookup table* para acessar por referência a borda e o padrão atual.

b. Shift_or

Inicialmente é pré-calculado as máscaras de cada padrão para cada letra do alfabeto. Essa informação é armazenada numa *lookup table* que é acessada por referência durante a busca. Foi optado por usar um *long long* para representar uma máscara. Assim o limite de caracteres que um padrão pode ter é no máximo 63.

c. Sellers

A função de atualizar a coluna dado um novo caractere foi feita de maneira void, o que evita criar um novo vetor e economiza memória. Cada coluna armazena um par, onde a primeira posição representa a distância e a segunda é um parâmetro usado para saber quantos caracteres devem ser imprimidos em caso de um match aproximado.

d. Ukkonen

Inicialmente é criado as máquinas de estados para cada padrão num pré-processamento. As máquinas de estados são então acessadas por referência durante a execução do algoritmo de match aproximado. Cada estado é armazenado num hashmap, tal que cada estado é mapeado para um único inteiro. A função de atualização de cada coluna foi feita de maneira semelhante ao sellers, a diferença é que não se permite que as distâncias ultrapassem a distância máxima + 1.

3. Testes e Resultados

Para a realização dos testes para se fazer a análise de desempenho dos algoritmos implementados na ferramenta foram desenvolvido scripts em python que realiza chamadas da ferramenta junto com a ferramenta "time" da biblioteca do gnu que foi utilizada com a opção "-f %e" para que o output seja apenas o tempo real de execução da ferramenta, e também foi utilizada as opções "-o" e "-a" para salver esses tempos de execução em um arquivo. Para cada teste realizado também foi passado para a ferramenta a opção "-c" para que as saídas sejam apenas o número de ocorrências do padrão, de forma que não seja gasto tempo exibindo cada ocorrência na tela.

Os testes foram realizados em um computador com sistema operacional Linux mint 20.3 com kernel 5.4.0-107-generic, processador AMD Ryzen 5 3500U, 13.7 Gb de memória Ram DDR 4 com velocidade de 2400 MT/s. todos os testes foram realizados com o notebook na tomada e sem acesso a rede.

Para fazer uma visualização gráfica dos resultados foi utilizada a ferramenta gnuplot e para cada gráfico gerado foi utilizado a média do tempo de execução resultante de 10 execuções do teste realizados em sequência.

3.1 Algoritmos de busca exata

Para se avaliar o desempenho dos algoritmos KMP e Shift_or foram realizados testes de busca em arquivos de texto obtidos no site http://pizzachili.dcc.uchile.cl/texts.html referentes a sequencias de proteínas divididas por linhas em arquivos de tamanhos 100 Mb, 500 Mb e 1.1 Gb para que possamos analisar o desempenho desses algoritmos com o aumento do tamanho do texto.

Para realizar outros testes também foram utilizados arquivos gerados pela própria equipe que tinham em seu conteúdo linhas de textos contendo 300 caracteres "S" repetidos por diversas linhas. Foram gerados 3 arquivos dessa forma com tamanho aproximado de 100Mb, 500Mb e 1,2 Gb cada um contendo respectivamente 360000, 1800000 e 3600000 linhas de texto.

3.1.1 KMP

Nos primeiros testes realizados rodamos o algoritmo KMP aumentando o tamanho do padrão a ser buscado indo de uma string de tamanho 5 até uma de tamanho 100 com incremento de 5 de forma que seja possível observar como a implementação do KMP se comporta com o aumento do tamanho do padrão a ser buscado. Nesses testes, cada busca foi realizada em arquivos referentes a sequências de proteínas de tamanho 100Mb, 500MB e 1.1Gb para também analisar o comportamento do algoritmo com o crescimento do tamanho do texto em que a busca é realizada. Com os resultados obtidos foram gerados os seguintes gráficos:

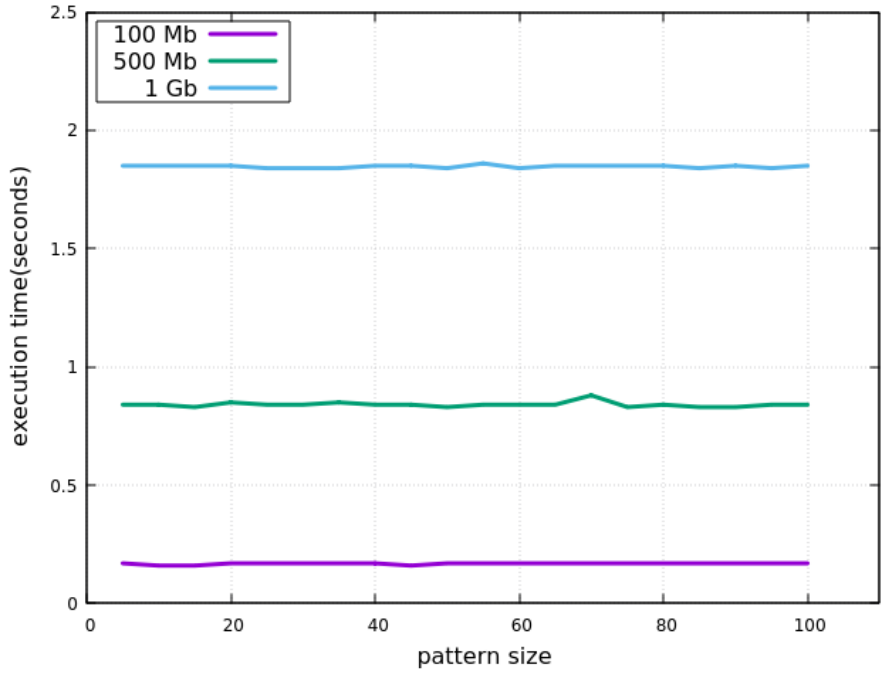

Gráfico 1 - Testes do KMP para diferentes tamanhos de arquivos

Como é possível observar com os gráficos temos que o tempo de execução do algoritmo está praticamente ficando constante em relação ao aumento do tamanho de padrão em um mesmo arquivo e que o aumento no tamanho do arquivo onde é feita a busca está sendo a maior influência no tempo de execução. Esse comportamento é esperado já que o algoritmo tem a complexidade assintótica definida por $O(n + m)$ onde n é o tamanho do padrão e m é o tamanho do texto onde é feita a procura, e como nos testes executados o tamanho do texto e ordens de grandeza maior que o padrão é de se esperar que ele seja a maior influência no tempo.

3.1.2 Shift or

Para analisar o desempenho do shift or faremos os mesmos testes realizados com o KMP, porém com o padrão inicialmente de tamanho 3 e crescendo até o tamanho 63, já que a nossa implementação utiliza uma máscara que impõe o limite 63 caracteres para o padrão.

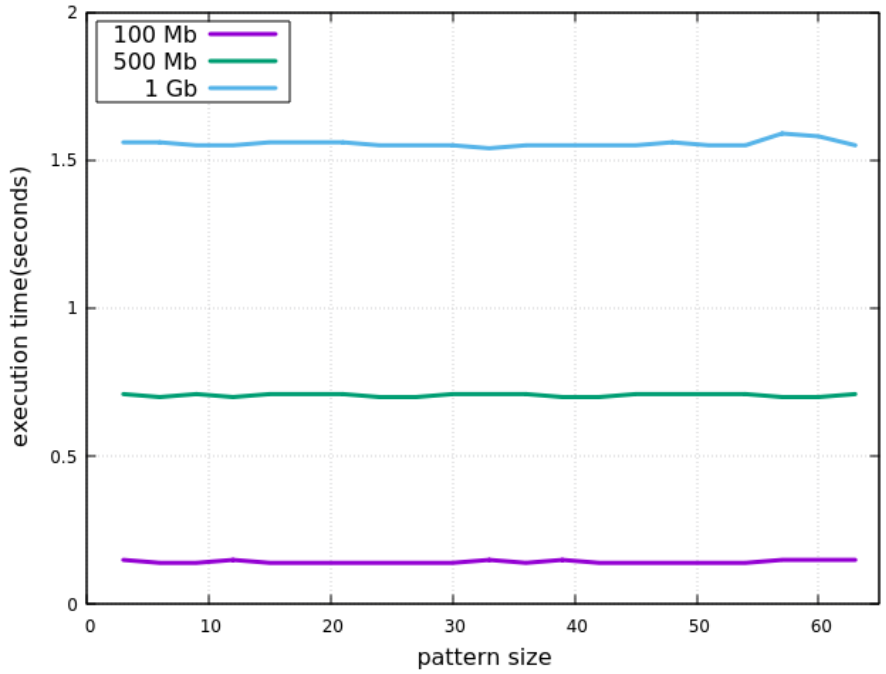

Gráfico 2 - Testes do Shift Or para diferentes tamanhos de arquivos

Da mesma forma que o KMP temos que o tempo de execução está majoritariamente sendo afetado pelo tamanho do arquivo em que a busca é executada, o que é de se esperar pois o shift or também tem a mesma complexidade de O(n + m) de forma que temos o mesmo comportamento pelos mesmos motivos.

3.1.3 Comparação KMP x shift or x Grep

Com os dados gerados nos testes anteriores podemos fazer uma comparação com a execução do Kmp e do Shift Or. Aproveitamos também para fazer uma comparação da execução dos nossos algoritmos como a ferramenta grep, para isso rodamos os mesmos testes com o grep, também passando a flag -c, que no grep retorna o número de linhas em que há pelo menos uma instância do padrão.

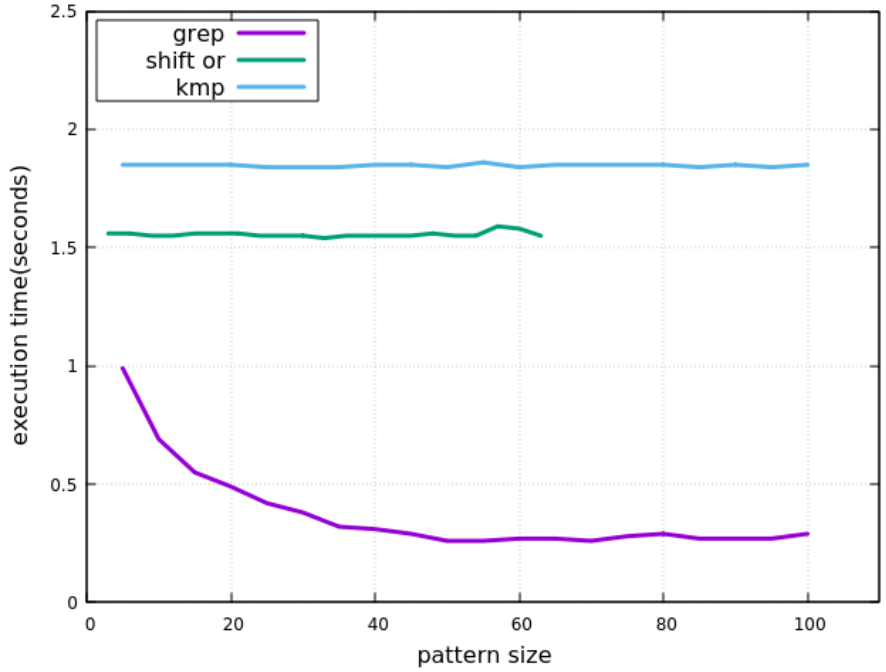

Gráfico 3 - Tempo de busca em um arquivo de 1.1Gb

Como é possível observar nos gráficos temos que para o pmt a execução do shift or se mostra mais eficiente em relação ao tempo do que o KMP se a string a ser procurada tiver tamanho menor ou igual a 63 caracteres. dessa forma poderia ser implementado em nosso pmt que uma busca exata de um pattern de tamanho menor que 64 usaría por default o algoritmo shift or, deixando o KMP apenas para strings maiores. Além disso, também podemos observar que nossos dois algoritmos implementados no pmt ainda deixam a desejar em comparação com o grep, que utiliza o algoritmo Boyer–Moore.

3.1.4 Uso de patternfile

Outra funcionalidade do nosso PMT que desejamos testar é o uso do pattern file para buscar mais de um padrão por chamada do programa. para realizar esses testes foi feito um script em python que selecionou 100 substrings de tamanho 50 distintas e que só apareçam uma única vez no arquivo em que a busca irá ser realizada e separe elas em arquivos contendo de 10 a 100 dessas substrings. com esses arquivos realizamos as buscas no arquivo de cadeias de proteínas de 1GB com os algoritmos KMP e shift or e também utilizando a ferramenta grep.

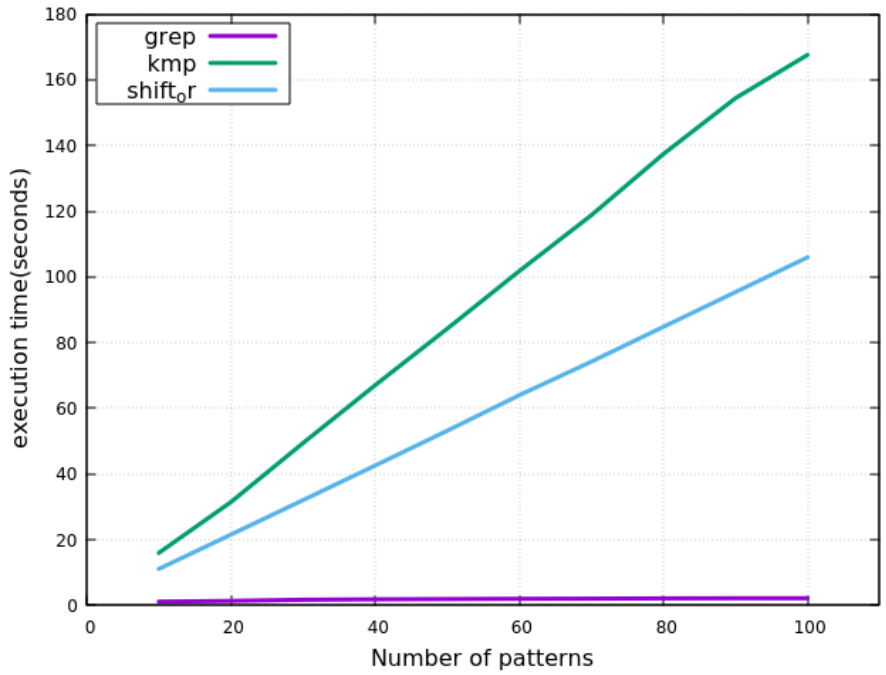

Gráfico 4 - Uso de patterfiles para busca em arquivo de 1.1Gb

Com os dados coletados podemos observar que a execução do KMP e shift or aumentam o tempo de execução com o aumento do número de padrões a serem procurados, já o grep, que continua sendo mais eficiente, se mantém num tempo mais constante. podemos atribuir esse comportamento ao fato de nem o kmp ou o shift or terem alguma estratégia específica para realizar a busca de diversos padrões de forma que eles vão buscando de um por um percorrendo o texto de busca uma vez para cada padrão, já o grep utiliza o algoritmo Aho-Corasick que faz um pré processamento de todos o padrões a serem procurados e percorre o texto de busca apenas uma vez.

3.2 Algoritmos de busca aproximada

No caso de avaliar o desempenho do algoritmo Sellers e do Ukkonen seguimos a mesma lógica dos testes para os algoritmos de busca exata, realizando testes com os mesmos arquivos mas com os padrões indo de uma substring de tamanho 5 até uma de tamanho 50, e também foram feitos testes para o cada algoritmo rodando com erro máximo de 1, e 2.

3.2.1 Sellers

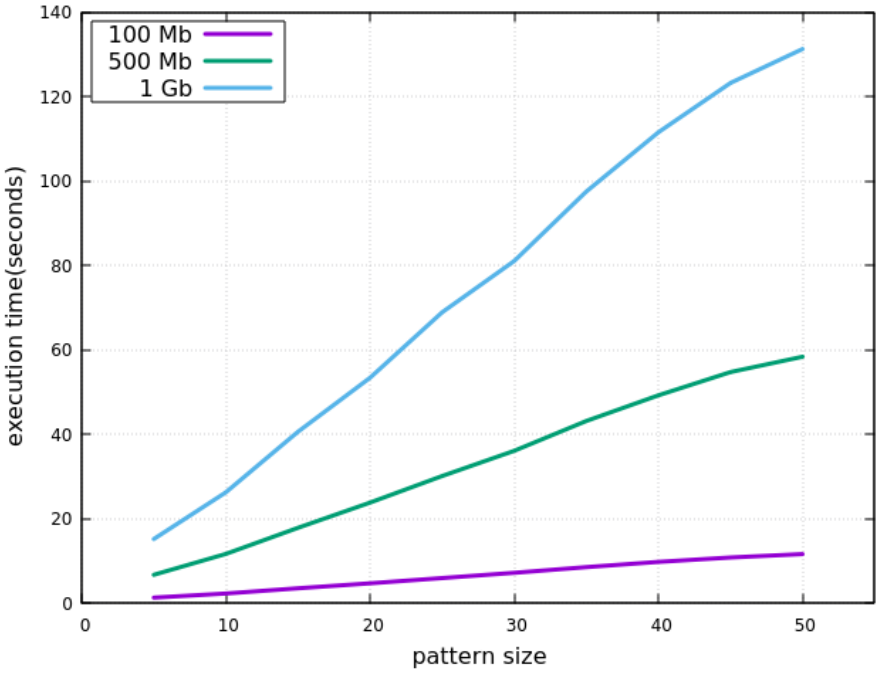

Gráfico 5 - Testes do Sellers com e = 1 para diferentes tamanhos de arquivos

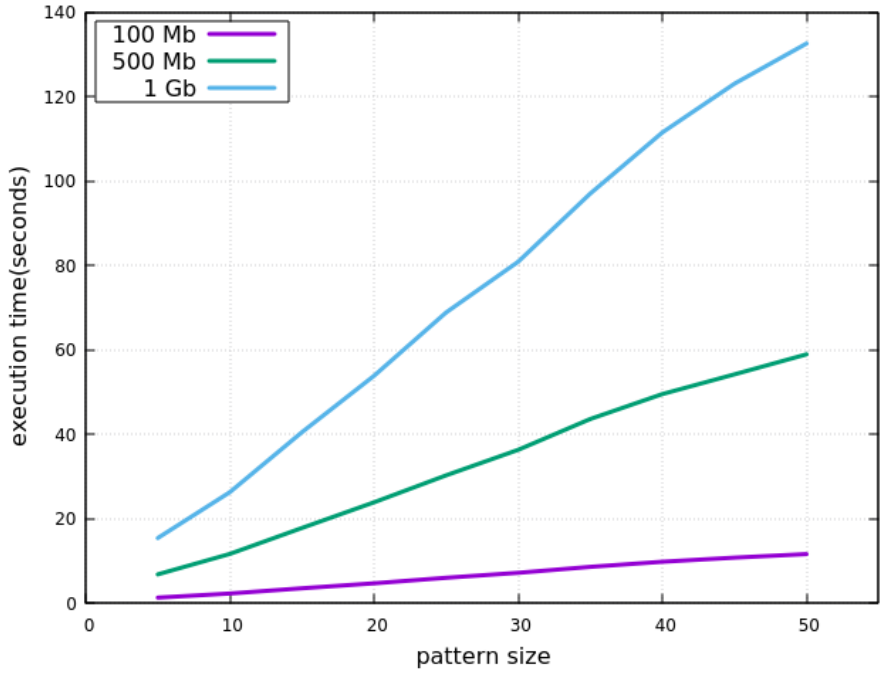

Gráfico 6 - Testes do Sellers com e = 2 para diferentes tamanhos de arquivos

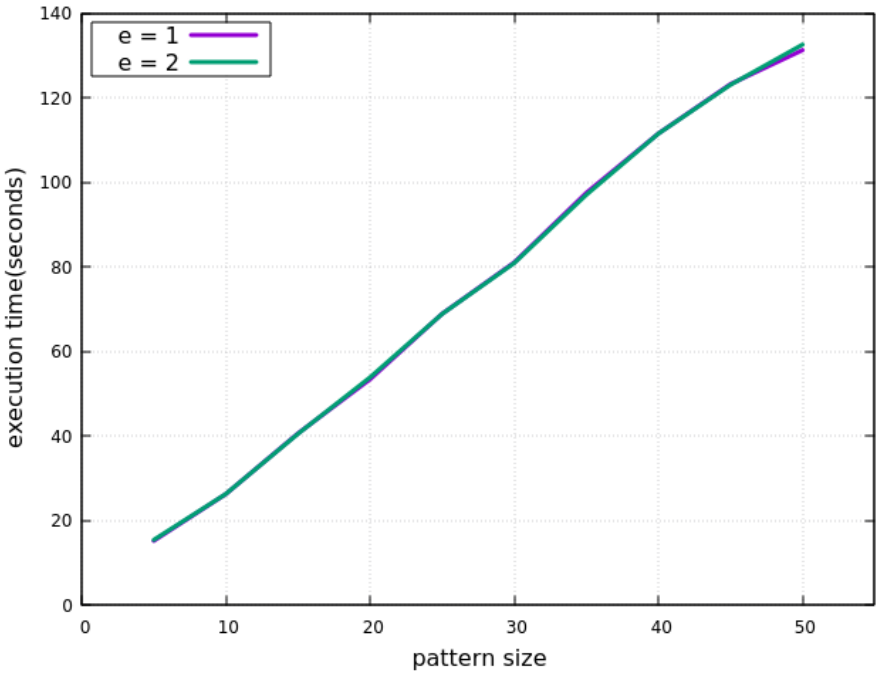

Gráfico 7 - Testes do Sellers em arquivo de 1.1Gb para e = 1 & e= 2

Ao analisarmos os gráficos resultantes dos testes realizados com o algoritmos sellers podemos ver que os tempos de execução vão aumentando de acordo com o aumento no tamanho do texto a ser realizada a busca, mas que o tempo não muda com o aumento do erro passado para o algoritmo. isso ocorre devido ao funcionamento do sellers que independente do erro faz as mesmas operações para realizar a busca.

3.2.2 Ukkonen

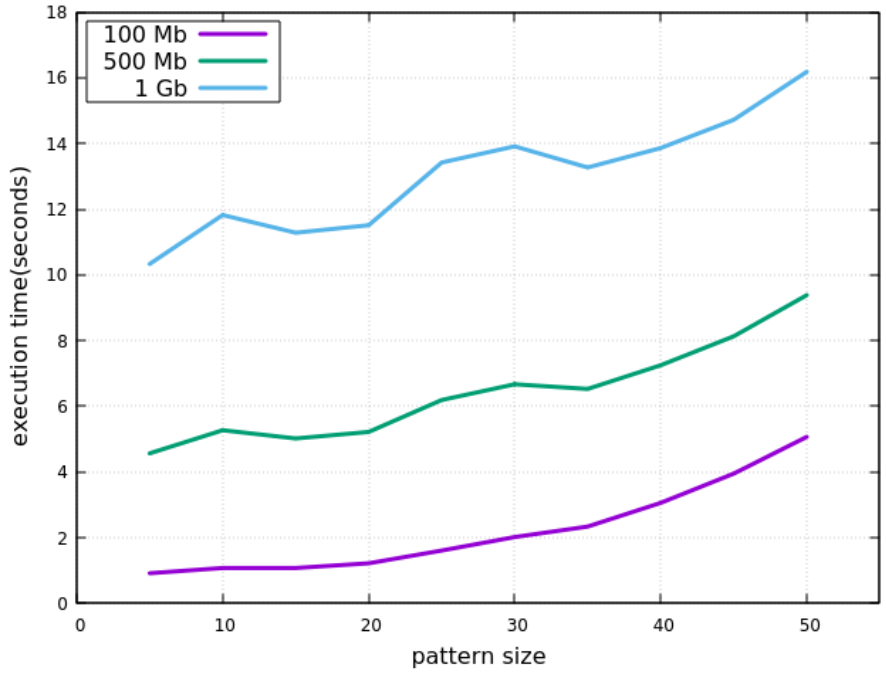

Gráfico 7 - Testes do Ukkonen com e = 1 para diferentes tamanhos de arquivos

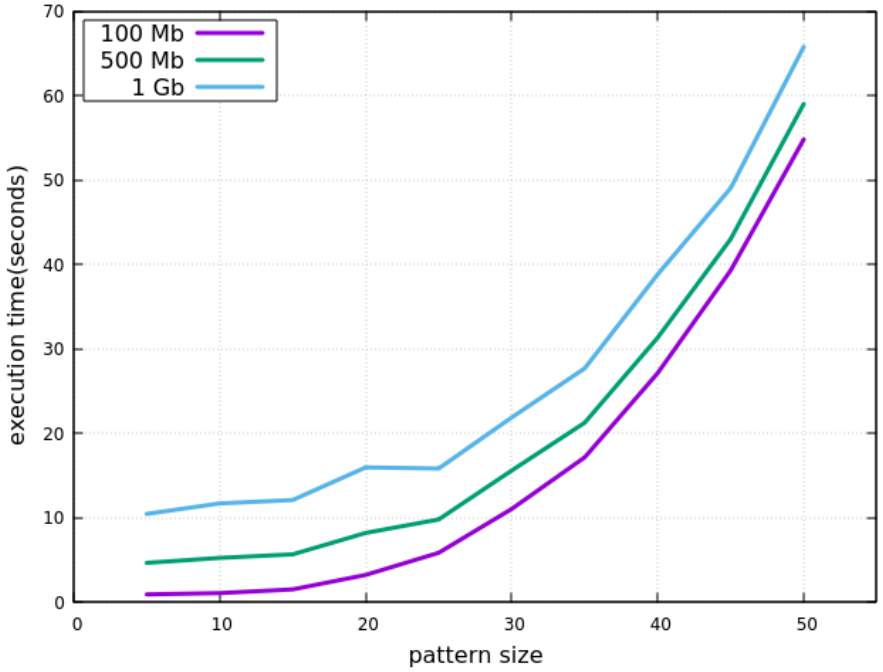

Gráfico 8 - Testes do Ukkonen com e = 2 para diferentes tamanhos de arquivos

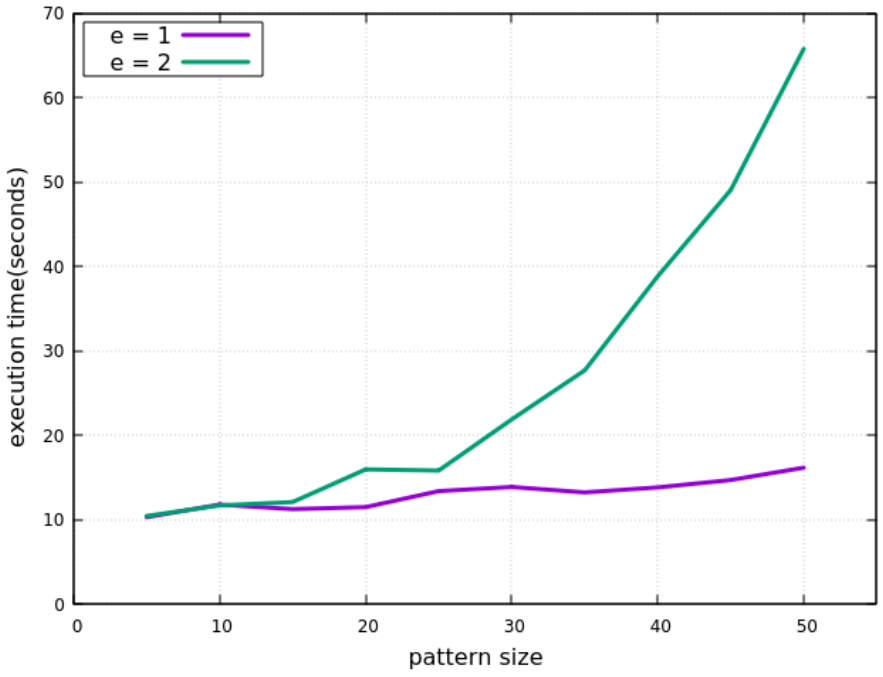

Gráfico 9 - Testes do Ukkonen em arquivo de 1.1Gb para e = 1 & e= 2

A princípio vemos o algoritmo de Ukkonen se comportando de maneira parecida com o sellers, com o tempo de execução aumentando com o tamanho do arquivo, porém com um tempo de execução mais baixo, entretanto ao aumentar o erro máximo de 1 para 2 vemos que o tempo de execução passa a ser mais lento. Esse comportamento é observado devido à própria execução do algoritmo, já que ele faz um pré-processamento para memorizar todas as transições possíveis de coluna. Ele também usa uma heurística que leva em conta o erro máximo, limitando os valores de cada coluna para edit_dist + 1. Ainda assim, o quanto o ukkonen precisa memorizar é exponencial tanto no tamanho do padrão quanto no erro máximo. Então para um padrão muito grande ou para um erro muito grande, uma quantidade exponencial de estados são gerados e memorizados, causando um estouro tanto de tempo quanto de memória.

3.2.3 Comparação sellers x Ukkonen

Com os testes realizados previamente podemos comparar mais de perto a execução dos algoritmos Sellers e ukkonen.

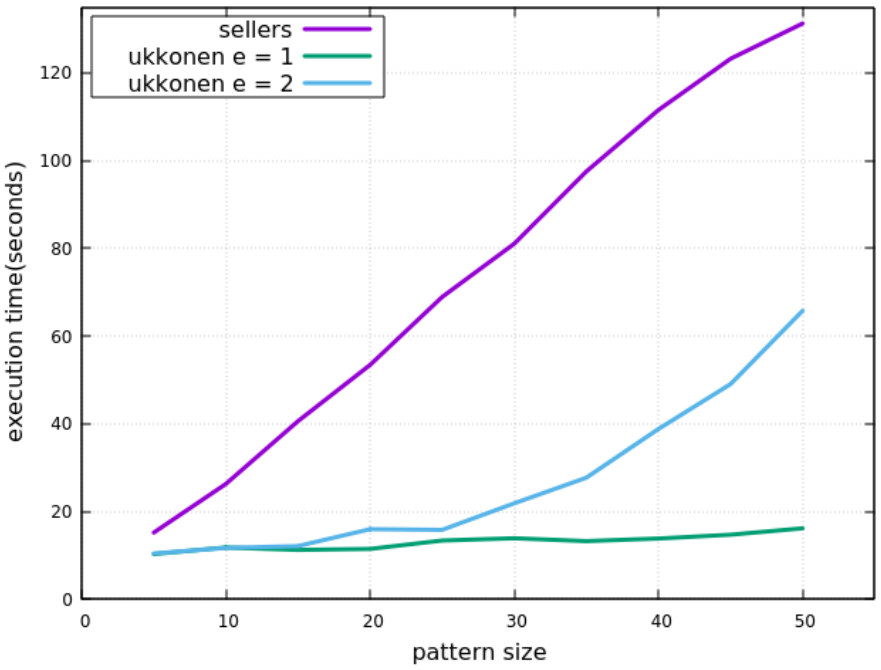

Gráfico 10 - Tempo de busca em um arquivo de 1.1Gb

Com os gráficos podemos observar que para buscas realizadas com erro 1 e 2 o ukkonen se mostrou mais eficiente , porém já no gráfico referente às buscas de erro igual a 2 podemos ver como o tempo de execução do ukkonen vai subindo de forma exponencial conforme o tamanho do padrão aumenta, de forma que a partir de algum tamanho de padrão seria mais eficiente utilizar o algoritmo de sellers devido a sua característica de não mudar o tempo de execução com o aumento do erro.

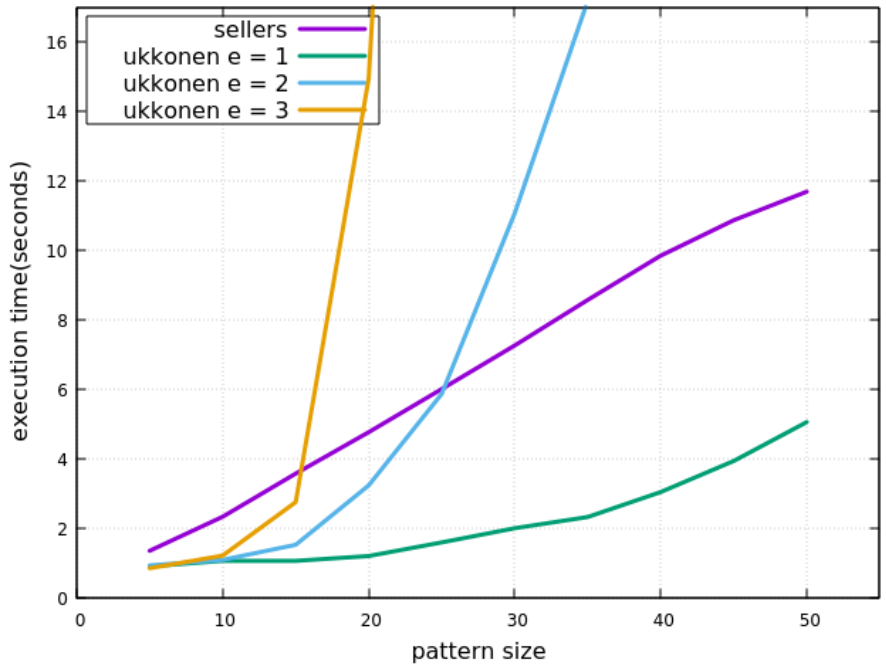

Gráfico 11 - Tempo de busca em um arquivo de 100Mb

Observando o gráfico da busca realizada no arquivo de 100mb podemos observar melhor como a busca do ukkonen fica mais lenta com o aumento do erro, de forma que com erro $= 2$ o sellers passa a ser mais eficiente a partir do padrão de tamanho 25 aproximadamente, já para o erro $= 3$ ele se torna mais eficiente ainda mais rápido com o padrão de tamanho aproximadamente igual a 16.

3.2.3 Uso de patternfile

da mesma forma que com os algoritmos de busca exata foi testada a capacidade dos algoritmos de busca aproximada de fazer buscas a partir de um pattern file, porém devido ao fato dos algoritmos de busca aproximadas terem um funcionamento mais lento foram feitas buscas por padrões de tamanho 10 em vez de tamanho 50.

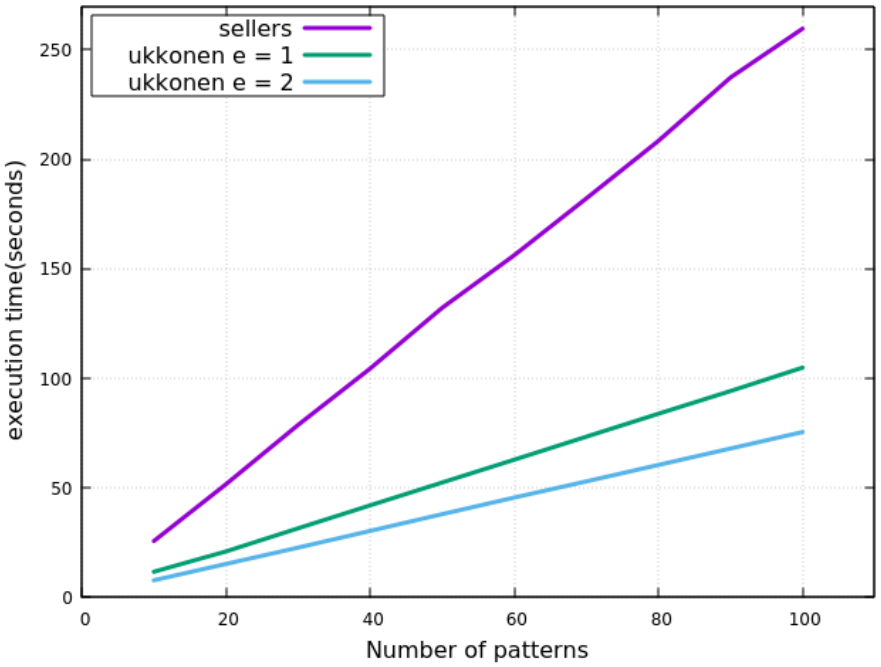

Gráfico 12 - Uso de patterfiles com sellers e ukkonen para busca em arquivo de 1.1Gb

Podemos observar um comportamento semelhante ao dos nossos algoritmos de busca exata, já que ambos o sellers quanto o ukkonen não possuem uma estratégia específica de buscar mais de um pattern de uma vez, logo o tempo de execução vai crescendo à medida que o número de padrões a serem buscados aumenta.