

Ferramenta PMT

Mário Victor Gomes de Matos Bezerra

1. Implementação

Detalhes de Implementação

Todos os algoritmos foram divididos em três partes: um construtor, que pré processa tudo que for necessário; uma função *count*, que conta o número total de ocorrências de um padrão em uma dada *string*; e, por fim, uma função *exists*, que determina se um padrão ocorre em uma dada *string*.

Para que seja possível processar arquivos de texto de uma tamanho considerável, o texto é processado linha a linha, logo as funções recebem como entrada uma referência de *string* de cada linha.

Representação alfabeto

O alfabeto considerado para este projeto foi o *extended ASCII*, já que é o suficiente para o escopo desse projeto.

Aho Corasick

Sua implementação consiste em: um construtor, que prepara a tabela de transição (*goto*), a de *fail* e armazena as ocorrências de cada estado; e duas funções, *count* e *exists*, que são responsáveis por percorrer a máquina de estados, representada pela tabela de transição, em procura de ocorrências.

Inicialmente, esse algoritmo começou fazendo uso de *unordered_map* para armazenar a tabela de transição, porém, após fazer *profiling* da aplicação, foi detectado que o *hash map* estava consumindo muito tempo de execução. Logo, começou-se a usar *vector* para representar a tabela de transição e *fail*, resultando em uma melhora significativa de desempenho, caindo de, em média, 3 minutos para 12 segundos.

Shift Or

Sua implementação consiste em: um construtor, que prepara todas as máscaras binárias de todos os caracteres do alfabeto; e duas funções, *count* e *exists*, que são responsáveis por percorrer o texto, aplicando as máscaras dado o caracter lido. Além disso, o algoritmo só aceita padrões de até 64 caracteres, devido a estrutura de dados usada, *uint64_t*.

Foi feita uma tentativa de implementar a forma alternativa desse algoritmo, Shift And, porem foi notado que ele possui um detalhe que o faz rodar uma quantidade maior de comandos, mais especificamente, a checagem de um *if*. Enquanto que o Shift Or roda essa checagem *tamanho_do_padrao* vezes, o Shift And roda essa checagem *tamanho_do_texto* vezes. Dessa forma, foi-se decidido usar a implementação do Shift Or.

Wu Manber

Sua implementação consiste em: um construtor, que, similarmente ao algoritmo Shift Or, prepara todas as máscaras binárias de todos os caracteres do alfabeto; e duas funções, *count* e *exists*, que são responsáveis por percorrer o texto e procurar as ocorrências.

Inicialmente, esse algoritmo fazia uso de uma matriz, inicialmente vazia, que era preenchida com o comando *emplace*. Porém, após fazer *profiling* da aplicação, percebeu-se que a alocação de memória estava tomando tempo demasiado. Dessa forma, começou-se a declarar um *vector* de tamanho dois, que, ao invés de alocar memória a quantidade de caracteres do texto, só foi necessário alocar a memória uma única vez.

Sellers

Sua implementação consiste em: três funções, *sellers*, responsável por calcular a distância de Levenshtein (casamento aproximado) e retornar a posição das ocorrências. Além das funções *count* e *exists*, as quais são responsáveis por retornar o número de ocorrências do padrão.

Inicialmente, o algoritmo foi implementado com uma matriz (*tamanho do texto da linha*) x (*tamanho do padrão*), causando piora na performance, dado que era

necessário alocar bastante memória. Dessa forma, começou-se a usar dois *vectors* e ficar alternando entre eles, já que os únicos dados necessário são os dados da linha anterior da matriz.

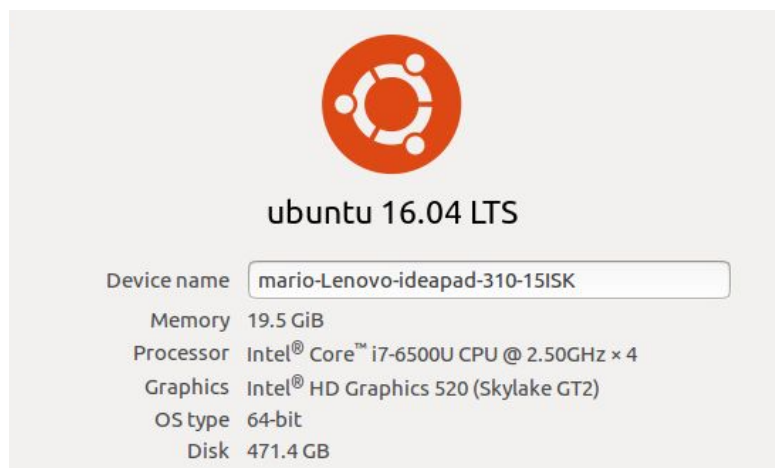
2. Testes

Os testes foram divididos em 2 módulos: busca exata, o qual varia-se o tamanho dos padrões de entrada e do texto; e busca aproximada, o qual varia-se o tamanho do texto e do distância de edição (0, 1 e 2).

Quanto aos arquivos, foram utilizados dois conjuntos de arquivos, que variam de tamanho (128MB, 256MB, 512MB, 1024MB e 2048MB) e conteúdo (xml da Wikipédia e DNA gerado aleatoriamente).

Enquanto que se usou o *grep* para comparação de eficiência dos algoritmos de busca exata, usou-se o *agrep* para os algoritmos de busca aproximada.

2.0 Especificação do Computador



2.1 Busca Exata

Neste módulo foram feitos diversos testes, enquanto se variava tamanho do arquivo e do padrão, os quais eram:

- a
- th
- does
- revision
- misunderstanding
- January]], [[February]], [[March

Os padrões foram escolhidos de acordo com a frequência com que eles aparecem nos textos, além de variar de tamanho entre 1, 2, 4, 8, 16 e 32.

Busca Exata - Wikipedia 256MB



Busca Exata - Wikipedia 512MB



Busca Exata - Wikipedia 1GB



Busca Exata - Wikipedia 2GB



Visto os resultados, podemos observar que o comportamento entre os diferentes tamanhos de arquivo se mantém para os algoritmos, sendo o mais eficiente o Grep. O Shift Or se mostra mais eficiente que o Aho Corasick, não se distanciando muito do Grep.

Também foi possível notar que a partir do padrão de 4 caracteres, o Aho Corasick se demonstrou mais estável.

Busca Exata - DNA 50MB



Busca Exata - DNA 100MB



Busca Exata - DNA 200MB



Busca Exata - DNA 400MB

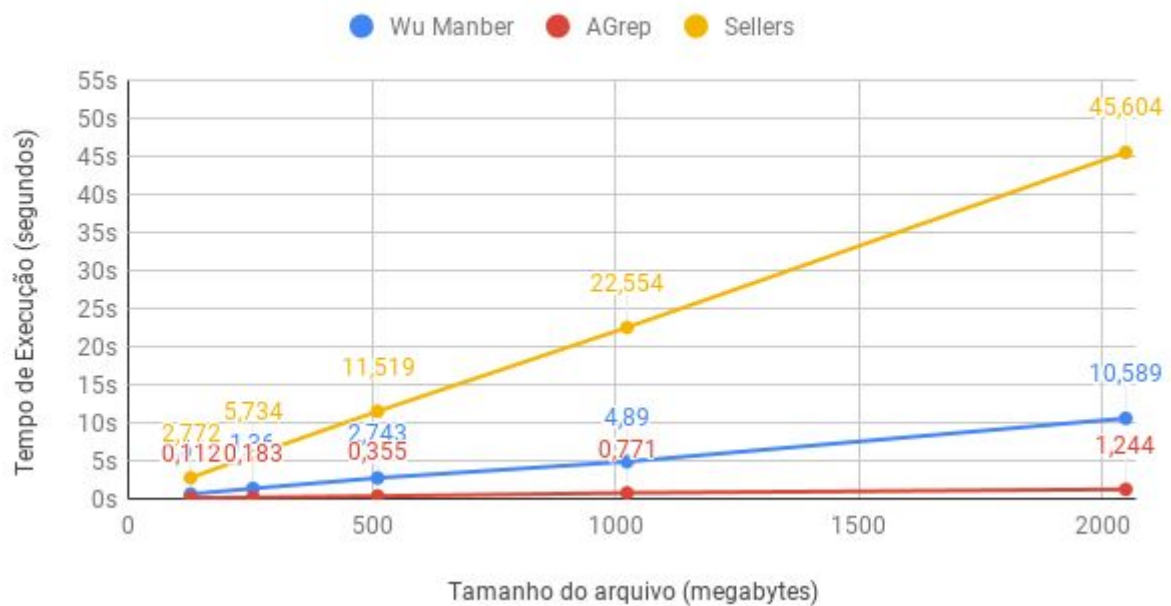


Visto os resultados, podemos perceber que o Shift Or se mostrou mais estável que o *grep* e por vezes superando em termo de eficiência. Já o Aho Corasick manteve seu comportamento do último conjunto de teste, mostrando-se mais lento que os outros algoritmos. Também se mostrando mais estável a partir do

padrão de tamanho 4, já *grep* se mostrou mais estável após o padrão de tamanho 8.

2.2 Busca Aproximada

Busca Aproximada - "revision" with error = 0



Busca Aproximada - "revision" with error = 1



Busca Aproximada - "revision" with error = 2



Visto os resultados, pode-se notar que o comportamento de crescimento do Wu Manber é bem similar ao do *agrep*, diferenciando-se pela eficiência. Note que o comportamento de ambos se mantém para distância de edição diferentes. Sendo mais custoso o processamento a medida que o erro aceitado aumenta. Já o Sellers se mantém constante mesmo quando o tamanho do erro aumenta, devido a sua característica de ser necessário realizar as mesmas operações, independente do tamanho do texto.

3 Conclusão

A análise dos gráficos gerados pelos testes nos permite tirar algumas conclusões sobre os algoritmos em questão.

Todas as implementações se mostraram com um desempenho médio inferior ao apresentado pelo *grep*, porém em alguns casos o Shift Or se mostrou mais eficiente. Já o Aho Corasick teria vantagem ao analisar múltiplos padrões simultaneamente.

O Wu Manber se mostrou menos performático que o *agrep*, porém com uma curva de crescimento similar e com desempenho não tão discrepante, enquanto que, mesmo que demonstrou uma performance inferior, o Sellers se manteve constante, mesmo variando a sua distância de edição.