# Ferramenta PMT

**Trabalho da Disciplina IF 767 - Processamento de Cadeias de Caracteres**

**Professor Paulo Fonseca** - paguso@cin.ufpe.br

Alunos:

Lucas de Souza Albuquerque - lsa2@cin.ufpe.br

Vítor Travassos Castelo Branco - vtcb@cin.ufpe.br

---

## Introdução e Identificação

Este relatório tem como intenção detalhar o processo de desenvolvimento e algumas escolhas tomadas na implementação de uma ferramenta ***pmt*** de busca de padrões em um ou mais arquivos de texto, utilizando algoritmos de busca exata e busca aproximada. Mencionamos, para cada um dos diversos algoritmos implementados, sua aplicação geral e as características de padrão e texto que determinam quando o tal deve ser o algoritmo escolhido para realizar a busca de padrões, além de analisar seus resultados e eficiência em diversas situações de teste.

A ferramenta desenvolvida, assim como definido na especificação, possui uma interface em linha de comando (CLI) onde o usuário deve explicitar o arquivo de padrões e os arquivos de texto nos quais ele deseja procurar os padrões, o que retornará as linhas de texto onde existe uma ocorrência de um dos padrões fornecidos. Adicionalmente, através de comandos opcionais adicionais, o usuário pode acessar as seguintes funcionalidades extras:

- **-p, --pattern,** que especifica o arquivo que contém os padrões que devem ser pesquisados;
- **-c, --count**, que retorna apenas a quantidade total de ocorrências dos padrões nos arquivos;
- **-o, --output,** que imprime os resultados em algum arquivo externo;
- **-a, --algorithm,** que permite o usuário a especificar qual algoritmo deve ser utilizado para a busca. Se este comando for omitido, a ferramenta escolhe automaticamente o algoritmo a ser utilizado dependendo da situação, o que detalharemos ao longo do relatório.
- E por último, **-h, --help**, que mostra uma documentação sobre a utilização da ferramenta e os valores para a chamada de algum algoritmo específico. ***(por exemplo, o algoritmo Boyer-Moore pode ser chamado por -a boyer, -a bm, entre outros)***

A ferramenta foi desenvolvida pelos alunos **Lucas de Souza Albuquerque**, que primariamente desenvolveu os algoritmos de busca exata **Naive, KMP (Knuth-Morris-Pratt)** e **Boyer-Moore**, e **Vítor Travassos Castelo Branco**, que primariamente desenvolveu os algoritmos de busca exata **Aho-Corasick** e **Shift-Or** e os algoritmos de busca aproximada **Sellers** e **Ukkonen.** A interface de linha de comando **CLI *(command line interface)***, a realização de testes, plotagem e análise de resultados, escolha automática dos algoritmos e desenvolvimento do relatório foram feitas em conjunto por ambos os membros da equipe.

---

## Implementação da Ferramenta

As seções a seguir tem como intuito detalhar a implementação da ferramenta em si, falando dos algoritmos implementados e de seus específicos, mencionando decisões feitas na implementação (como heurísticas utilizadas, estruturas de dados...) e analisando limitações e problemas encontrados no trabalho, e como a ferramenta poderia ser melhorada no futuro.

### Algoritmos

Em seu estado de entrega, a ferramenta possui 5 (cinco) algoritmos de busca exata e 2 (dois) algoritmos de busca aproximada. Se o usuário não especificar qual dos algoritmos ele deseja utilizar para a busca de padrão, a própria ferramenta escolhe o melhor algoritmo que deve ser aplicado na situação, dependendo das propriedades e quantidade dos padrões que se desejam encontrar e propriedades dos arquivos de texto. Mencionaremos na seção de trabalhos futuros mais à frente duas coisas que seriam interessante adicionar para a ferramenta posteriormente.

**Observação:** ***os valores que representam cada algoritmo para a utilização do comando -a na CLI podem ser encontrados na documentação da ferramenta e em um apêndice no final deste relatório.***

Entre os algoritmos de busca exata implementados temos:

- **Brute-Force** ou **Naive -** A implementação ingênua do algoritmo de busca.
- **KMP** ou **Knuth-Morris-Pratt**
- **Aho-Corasick**
- **Boyer-Moore**
- **Shift-Or** ou **Bitap**

Através de análise do funcionamento e complexidade dos algoritmos, e através dos resultados conseguidos pelos nossos testes, chegamos em algumas conclusões que nos auxiliaram a definir a escolha automática de algoritmos da ferramenta, caso o usuário não especifique o algoritmo.

- Primeiramente, em relação ao caso em que estejamos analisando um padrão único com tamanho ***m = 1***, só precisamos comparar se cada caractere do texto é igual ao caractere que representa o padrão. Logo, não precisaremos se preocupar com pré-processamento nenhum, e não teremos o custo de tempo e espaço associado a este. Neste caso, então, é ideal utilizar o próprio algoritmo ingênuo de **Brute-Force** (força bruta).
- Para casos em que desejamos encontrar múltiplos padrões em um ou mais arquivos de texto, é mais efetivo que esta busca ocorra simultaneamente. Obviamente, então, utilizamos o algoritmo **Aho-Corasick,** que efetua a busca dos vários padrões em paralelo.
- Em casos com padrões pequenos - em particular, quando o tamanho do padrão é menor que 64 - utilizamos uma implementação mais rápida do algoritmo **Shift-Or,** que armazena as máscaras em **unsigned integers** de 64 bits. Uma segunda implementação do algoritmo **Shift-Or,** que utiliza um **bitvector** para armazenar as máscaras, não é limitada pelo tamanho do padrão, mas também apresenta menor eficiência em sua execução, então ele não é geralmente usado.
- Nos casos restantes, utilizamos o algoritmo **Boyer-Moore**.

Entre os algoritmos de busca aproximada implementados temos:

- **Sellers**
- **Ukkonen**

Os algoritmos de busca aproximada são utilizadas quando se é fornecido um 'erro máximo' por meio do comando **-e** ou **--edit** na linha de comando. Em vez de procurar apenas os padrões fornecidos, estes algoritmos retornam como 'casamento' todas as ocorrências de palavras que tem até certa distância de erro do padrão fornecido. (Como um exemplo básico, para erro máximo **1**, uma pattern **barro** daria matching com a substring **carro** encontrada em um texto, por distância de Levenshtein.)

A nossa ferramenta, para estes casos, tenta sempre utilizar o algoritmo **Ukkonen**, se for possível. Este algoritmo precisa de um tempo de pré-processamento para construir um autômato para a busca de complexidade exponencial, mas seu processamento é extremamente eficiente. De maneira geral, deve-se utilizar o **Ukkonen** quando o tamanho do padrão é pequeno (ou quando o erro máximo é pequeno para padrões maiores).

Quando não se é utilizado o Ukkonen, por eliminação usamos o algoritmo restante - ou seja, o **Sellers**, que só depende do tamanho da padrão. O ideal seria também, em versões futuras da ferramenta, que o algoritmo **Wu-Manber** fosse implementado, para ser usado em casos em que o tamanho do erro máximo não fosse grande. A implementação do Wu-Manber também forneceria mais um algoritmo de busca aproximada ao usuário, em vez de apenas dois comparados aos cinco fornecidos para busca exata.

## Detalhes de Implementação

Esta seção tem como intuito detalhar decisões tomadas nas implementações dos algoritmos e da ferramenta em geral. O primeiro detalhe que achamos importante ressaltar é o modo de como realizamos a leitura dos arquivos: no caso, foi utilizada a interface <fstream> disponibilizada pela STL ***(Standard Template Library)***. Esta leitura é executada de modo simples, linha por linha - logo, uma otimização possível para a ferramenta seria ler os arquivos de texto em blocos maiores.

Como mencionado acima, o tamanho do alfabeto para a heurística de ***bad character*** do **Boyer-Moore** é fixado como **256**. Por causa disso, para alfabetos pequenos, ainda temos uma tabela para a heurística do tamanho do alfabeto - o que significa espaço de memória que não está sendo utilizado. Uma outra otimização possível seria determinar o tamanho do alfabeto baseado nos padrões, com caracteres que não existem no padrão sendo mapeados para um caractere adicional que representa o caractere nulo.

Outros algoritmos também sofrem deste problema - seja $|\boldsymbol{a}|$ o tamanho do alfabeto, para o algoritmo **Aho-Corasick**, cada nó tem um número $|\boldsymbol{a}|$ de arestas; para o **Shift-Or**, existem $|\boldsymbol{a}|$ máscaras por padrão; e para o **Ukkonen**, cada célula do autômato tem $|\boldsymbol{a}|$ arestas. Nossa escolha de fixar o tamanho do alfabeto inicialmente ajudou bastante na hora de desenvolver os algoritmos,

pois com um alfabeto suficiente grande não tivemos que nos preocupar com possíveis problemas. Porém, seria interessante, como passo futuro, realizar a otimização mencionada e pegar o tamanho do alfabeto pelos padrões.

### Limitações, bugs e Trabalhos Futuros

Apesar de nosso trabalho apresentar uma ampla seleção de algoritmos de busca exata, por causa de uma situação inesperada, não conseguimos terminar uma implementação concreta do algoritmo **Wu-Manber**, fornecendo apenas dois algoritmos de busca aproximada para o usuário. Seria interessante, no futuro, incrementar a ferramenta para também possuir este algoritmo.

Em relação aos algoritmos implementados, uma limitação percebida é que alguns algoritmos consideram apenas caracteres ASCII em sua implementação. Embora a maior parte dos algoritmos possa funcionar com caracteres non-ASCII, essa feature não foi exaustivamente testada, e logo, pode ser que ocorra algum comportamento inesperado em certos casos. Mesmo que posteriormente seja descoberto que isso não é um problema e os algoritmos ainda funcionem, não é algo que podemos garantir e achamos importante mencionar.

Em relação ao algoritmo de busca exata **Boyer-Moore**, percebemos, durante a implementação, além das duas heurísticas de ***bad character*** e ***good suffix,*** a existência de uma regra adicional chamada **Galil Rule.** Além desta regra melhorar a eficiência do algoritmo, ela também é necessária para provar tempo de execução linear no pior caso. Tendo em vista estas características da regra, seria interessante posteriormente aprofundar neste assunto e estudar sua implementação para incrementar nosso algoritmo Boyer-Moore.

Em relação ao algoritmo de busca exata **Shift-Or,** para ***patterns*** grandes (tamanho maior que 64), encontramos uma certa lentidão em sua execução. Apesar de não ser nada incrivelmente grave, e um tempo de execução mais lento já ser esperado pelo fato de não podermos mais usar um **unsigned int** de 64 bits para armazenar as máscaras, seria uma possibilidade interessante estudar mais a fundo o motivo exato do algoritmo não estar tão rápido quanto poderia estar para padrões destes tamanhos. Uma das limitações nesta questão de tempo que discutimos é o modo de como foi implementado o **bitvector** para o algoritmo. De toda maneira, seria interessante, caso continuássemos o desenvolvimento da ferramenta, otimizar o algoritmo **Shift-Or** ainda mais.

---

## Testes e Resultados

A seção a seguir tem como intuito falar do processo de testes e análise da eficiência dos algoritmos implementados sob diversas situações, com o intuito de auxiliar a ferramenta a escolher qual algoritmo deve ser selecionado automaticamente a partir das propriedades dos arquivos de ***patterns*** e arquivos de texto fornecido. Estes testes foram feitos em cima da nossa própria ferramenta **PMT**, e, para fins de comparação com uma ferramenta externa, também incluímos os resultados da própria ferramenta ***grep*** para análise. Os tempos de execução obtidos foram fornecidos pelo comando ***time*** do terminal.

Os testes descritos foram conduzidos em um computador com o sistema operacional **Ubuntu 16.04**, **8** (oito) gigabytes de memória RAM, e processador **Intel Core i5-5200U - 2.2GHz**. Eles foram rodados à partir dos arquivos **ENGLISH** e **DNA** completos, encontrados no seguinte endereço web fornecido pela especificação do projeto: http://pizzachili.dcc.uchile.cl/texts.html. Os padrões procurados no arquivo ENGLISH foram palavras e frases encontradas aleatoriamente no texto, e os padrões procurados no arquivo DNA foram gerados aleatoriamente.

## Testes para Algoritmos de Busca Exata

Vale ressaltar que, para os testes dos algoritmos de Busca Exata, as patterns com tamanho ***> 64*** foram ignorados para o algoritmo **Shift-Or**. Como mencionado anteriormente, à partir deste tamanho a implementação mais lenta com **bitvector** do Shift-Or é utilizada, e seu tempo de execução mais lento estava tornando inviável fazer muitos testes. Como já sabemos dos problemas da implementação deste algoritmo quando é necessário mais de um **unsigned int** para armazenar as máscaras, e sabemos que não vamos usar o algoritmo para padrões com tamanho ***> 64***, ignoramos estes padrões para o Shift-Or para poder realizar mais testes e chegar em mais conclusões.

### Teste 1: Arquivo ENGLISH

No primeiro teste, utilizamos um arquivo de texto de 1024MB (~1GB) com os padrões que desejávamos procurar sendo palavras e frases existentes no arquivo. O tamanho destes padrões variou de 1 até 70, abrangendo uma boa variedade de situações. A Figura 1 na página a seguir mostra, para cada algoritmo, o tempo médio de execução relacionado ao tamanho dos padrões.

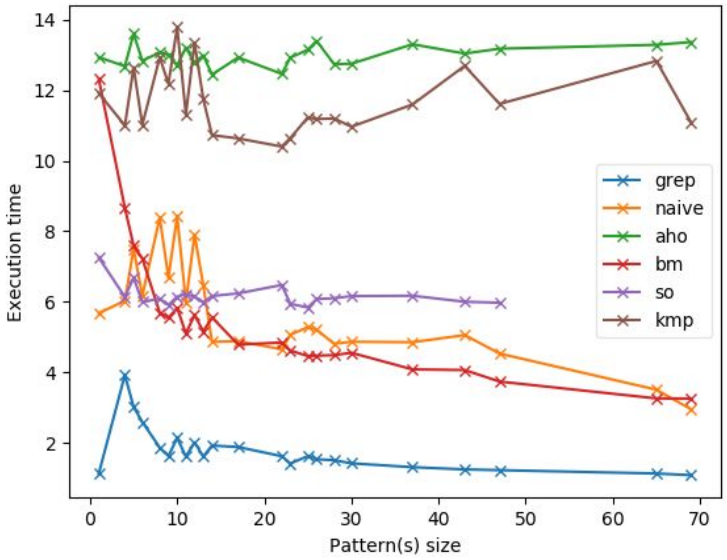


***Figura 1. Teste feito com padrões de tamanho 1 a 70 no arquivo ENGLISH (1024MB). Como mencionado, os padrões foram palavras e frases existentes no próprio arquivo - ou seja, garantimos sua existência.***

Como podemos ver, para padrões maiores, o algoritmo **Boyer-Moore** começa a se tornar uma boa escolha, e é ele que se aproxima mais da ferramenta ***grep***. Para padrões menores, porém, por causa

do tempo de seu pré-processamento e sua dependência ao tamanho de alfabeto fixado, ele é uma das piores escolhas. O algoritmo **Aho-Corasick** também se mostra como uma escolha ruim para a análise de padrões individualmente, mas como já sabemos, sua força está na execução de vários padrões ao mesmo tempo.

**Teste 2: Arquivo DNA**

O segundo teste principal que fizemos para algoritmos de busca exata foi com o arquivo DNA, que possui aproximadamente 200 MB. Os padrões que procuramos neste arquivo foram gerados aleatoriamente, com seus tamanhos variando de 1 até 30. amanho dos padrões. A figura a seguir mostra os resultados encontrados e compara o tamanho dos padrões com o tempo de execução.

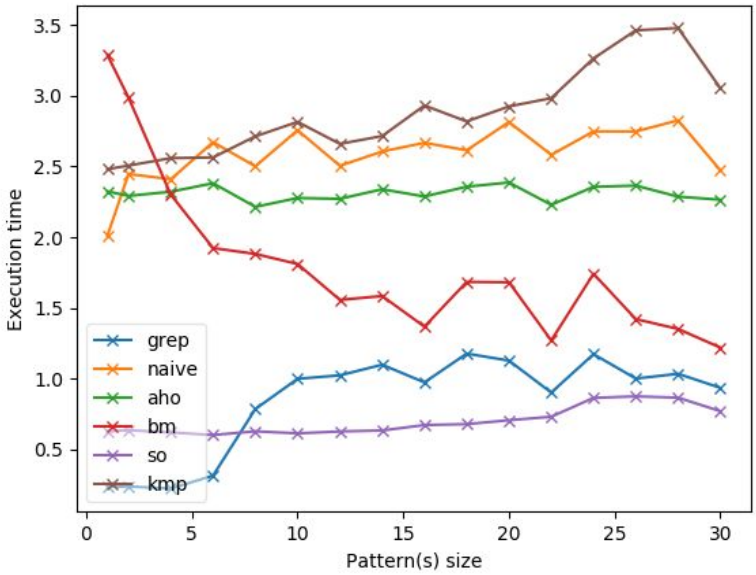


***Figura 2. Teste feito com padrões de tamanho 1 a 30 no arquivo DNA (200MB)***

É interessante notar que, para este arquivo, com padrões de tamanho menores e formato mais aleatorizado, o algoritmo **Shift-Or** consegue um tempo de execução menor do que a própria ferramenta ***grep.*** O mesmo comportamento se percebe para o **Boyer Moore,** que começa sendo muito pouco eficiente para tamanhos de padrões pequenos e melhora conforme o tamanho do padrão aumenta. Como não garantimos também a existência do padrão no arquivo pelo modo de como o geramos, o **Naive** também sofre bastante no seu tempo de execução.

**Teste 3: Arquivo DNA, Múltiplos Padrões**

A ideia por trás do último teste principal para algoritmos de Busca Exata foi a análise de múltiplos padrões na mesma chamada da ferramenta. A Figura 3 abaixo relaciona o tempo de execução de quatro algoritmos, o **Naive**, o **Boyer-Moore,** o **Shift-Or** e o **Aho-Corasick** com o número de padrões que desejamos analisar.

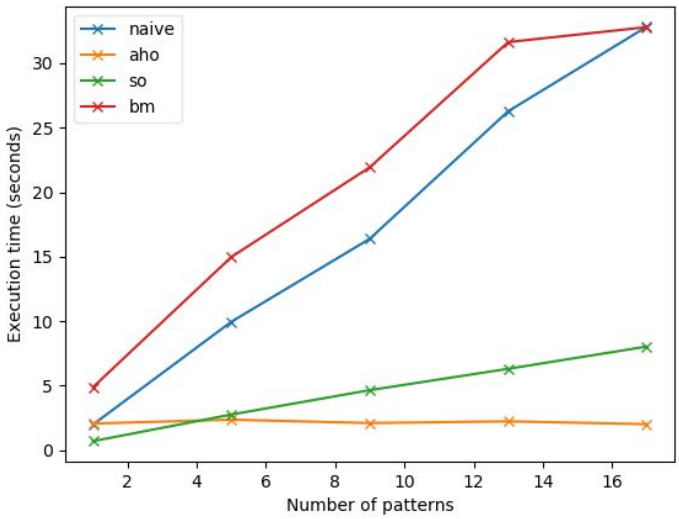


***Figura 3. Teste feito com número de padrões de 1 a 17 no arquivo DNA (200MB)***

Como esperado, o algoritmo Aho-Corasick apresenta melhor tempo de execução, pois como busca pelos padrões 'em paralelo', sua eficiência independe do número de padrões que desejam ser analisado. Os algoritmos que mais sofrem com a introdução de mais padrões são o Naive e o Boyer-Moore. Apesar do Boyer-Moore fazer o pré-processamento para cada padrão no começo, os dois algoritmos só procuram um padrão por cada vez que percorrem uma linha, e sua eficiência então diretamente piora conforme adicionamos mais padrões à serem procurados.

## Testes para Algoritmos de Busca Aproximada

Por definição, a busca de padrões com algoritmos de busca aproximada possui um fator adicional do que os algoritmos de busca exata. Este fator é o erro máximo, que define que todas as substrings que se diferem do padrão procurado por aquele erro, ou por menos, seriam consideradas como um ***matching***. Com um Erro Máximo pequeno, assim que uma substring começa a se diferir do padrão, podemos parar de checá-la. Considerando uma distância Levenshtein máxima de apenas 1, por exemplo - assim que a substring que estivermos analisando se diferir em duas letras do padrão procurado, podemos parar de analisá-la. Quando o Erro Máximo é grande, isto não acontece - pois nosso 'benefício da dúvida' para uma substring é bem maior, e o número de substrings que consideraremos como 'possíveis casamentos' é maior ainda. Devido à esses fatores, resolvemos focar no Erro Máximo para nossos testes.

### Teste 1: DNA, Erro Máximo 1

Ambos os testes para Algoritmos de Busca Aproximada foram feitos no arquivo de texto DNA, de aproximadamente 200 MB de tamanho. Os padrões procurados foram gerados aleatoriamente, e no caso deste primeiro teste, o erro máximo foi definido como um. A figura abaixo mostra a relação entre o tamanho do padrão e o tempo de execução dos algoritmos.

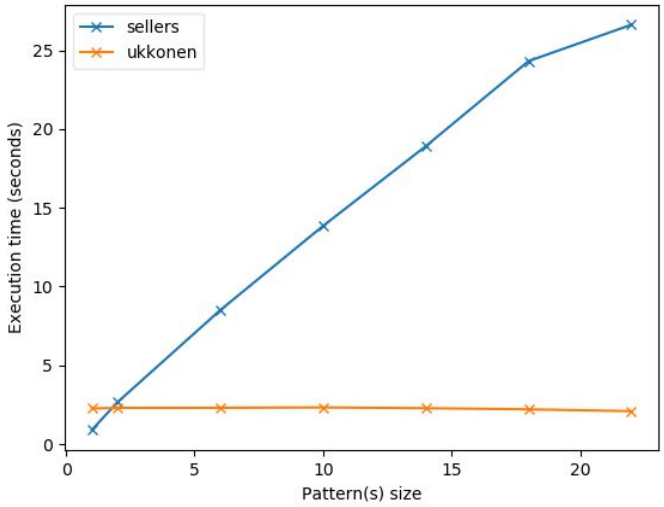


***Figura 4. Teste feito com padrões de tamanho 1 <= m <= 22 no arquivo DNA (200MB). Erro Máximo = 1.***

Nós sabemos, pela implementação do Sellers, que seu tempo de execução depende apenas do tamanho do padrão, e não do erro máximo, porém, podemos perceber que para um erro máximo pequeno, o tempo de execução do Ukkonen, após seu pré-processamento, é quase constante, independentemente do tamanho do padrão.

**Teste 2: Arquivo DNA, Erro Máximo M-1**

O segundo teste realizado foi com um Erro Máximo maior. O mesmo arquivo DNA de 200 MB e o mesmo método de gerar padrões foi utilizado, mas o Erro Máximo para cada iteração dos algoritmos foi definido como M-1, ou seja, o tamanho do padrão menos um. Para um padrão de tamanho 6 (seis), o erro máximo foi 5 (cinco), por exemplo, e para um padrão de tamanho 18 (dezoito), o erro máximo foi 17 (dezessete). A figura abaixo mostra a relação entre o tempo de execução e o tamanho do padrão para os dois algoritmos.

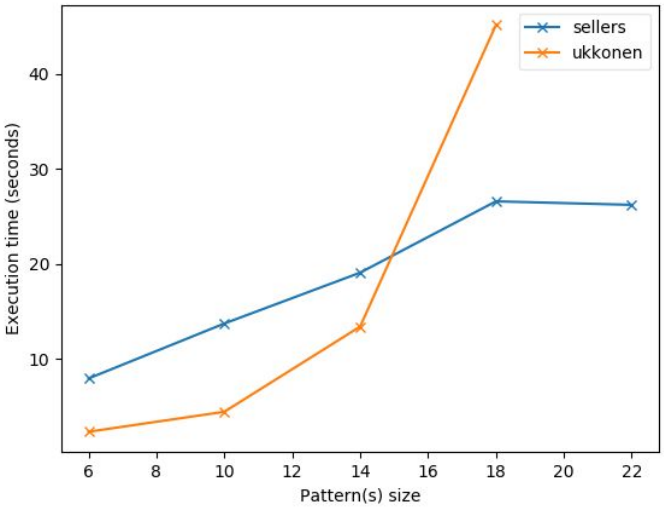


***Figura 5. Teste feito com padrões de tamanho 6 <= m <= 22 no arquivo DNA (200MB). Definimos o Erro Máximo (Max Error) como m - 1, ou seja, o tamanho do padrão menos um.***

Ao contrário do primeiro teste para algoritmos de busca aproximada, podemos ver que após certo tamanho de padrão o tempo de execução para o **Ukkonen** aumenta drasticamente e o mesmo se torna basicamente inviável. Isto é devido a dependência do **Ukkonen** à tanto o tamanho do padrão e ao Erro Máximo, e no caso deste teste, os dois fatores crescem em conjunto. O **Sellers,** como esperado, cresce em sua maior parte, pois ele só depende do tamanho do padrão.

## Conclusões

Baseado nos resultados obtidos e no conhecimento do funcionamento dos algoritmos implementados, pudemos definir as heurísticas para a escolha do algoritmo de acordo com a entrada.

Pode-se observar, também, diversos comportamentos interessantes sobre os algoritmos:

- Naive é um dos melhores algoritmos para textos com alta diversidade. No entanto, perde em eficiência quando o alfabeto é pequeno.
- Shift Or para padrões pequenos supera a implementação do grep para padrões pequenos.
- Ukkonen é extremamente eficiente quando o erro máximo e o tamanho do padrão são pequenos, mas quando ambos crescem, seu tempo de execução se torna muito alto.
- Aho Corasick é ineficiente para procurar um único padrão, mas começa a superar os outros algoritmos à medida em que o número de padrões a serem procurados aumenta.