# Ferramenta PMT

## 1. Identificação

A equipe é formada pelos alunos **Diogo Oliveira Rodrigues** e **Gabriela Araújo Britto**. Gabriela implementou os algoritmos de casamento aproximado e Diogo implementou os algoritmos de casamento exato e a função de leitura do texto. Ambos contribuíram com os módulos restantes.

## 2. Implementação

Primeiro, vamos deixar claro a notação usada neste relatório, a não ser que seja explicitamente especificado.

- m = tamanho do padrão
- n = tamanho do texto
- $\sigma$ = tamanho do alfabeto
- t = erro máximo permitido em casamentos aproximados

### 2.1. Algoritmos de casamento exato

Os algoritmos implementados nessa etapa do projeto foram Bruteforce, Boyer-Moore, Shift-Or e Aho-Corasick. O Shift-Or é restrito a padrões de tamanho menor que 65. Ao testar padrões de variados tamanhos em um arquivo de 2GB da língua inglesa, notamos que para m ≤ 4, o Bruteforce se mantém eficiente, enquanto o Boyer-Moore é bastante lento. Em torno de m = 5, o Boyer-Moore começa a melhorar e, quando m aumenta mais ainda, fica evidente a complexidade de caso médio sublinear do Boyer-Moore. Além disso, o Shift-Or possui um desempenho bastante estável e eficiente independente de m. Claramente, o Aho-Corasick é superior quando vários padrões estão sendo procurados. Combinando as qualidades de cada algoritmo, que podem ser vistas com maiores detalhes na seção de testes, chegamos à seguinte escolha do algoritmo a ser usado:

- ➔ Se mais de um padrão é fornecido, utiliza-se o Aho-Corasick.
- ➔ Senão, se m ≤ 8, utiliza-se o Shift-Or.
- ➔ Senão, utiliza-se o Boyer-Moore.

### 2.2. Algoritmos de casamento aproximado

Os algoritmos implementados nessa etapa do projeto foram Ukkonen, Sellers e Wu-Manber. O Wu-Manber é restrito a padrões de tamanho menor que 65. O ideal é que a etapa de busca fosse feita com o Ukkonen, já que este tem uma complexidade linear no tamanho do texto. Entretanto, devido à sua complexidade de pré-processamento exponencial em t (erro), precisa-se trocar de algoritmo se certas condições não forem obedecidas. Como em geral t ≤ m, a complexidade do Wu-Manber sempre será pelo menos tão boa quanto a do Sellers. Diante disso, e

das informações expostas na seção de testes, chegamos à seguinte escolha do algoritmo a ser usado:

➔ Se n ≥ 1GB e ((m ≤ 105 e t ≤ 4) ou (m ≤ 75 e t ≤ 5) ou (m ≤ 20 e t ≤ 14)), utiliza-se o Ukkonen.

➔ Senão, se m ≤ 64, utiliza-se o Wu-Manber.

➔ Senão, se 100MB ≤ n ≤ 1GB e m ≤ 105 e t ≤ 5, utiliza-se o Ukkonen.

➔ Senão, utiliza-se o Sellers.

## 2.3. Detalhes de implementação relevantes

- **Leitura do texto**

O texto é processado linha por linha pelos algoritmos, entretanto, chunks do texto são salvos em um buffer de pelo menos 32MB. Mais especificamente, o texto é processado como um arquivo binário e um chunk é lido internamente pelo parser. Enquanto houver linhas nesse chunk, o parser devolve um ponteiro para o ínicio dessa linha no buffer. Com isso, o programa pode abstrair a leitura e assumir que está sendo lido de linha em linha. Dessa forma, conseguimos otimizar em mais de duas vezes a leitura do texto.

- **Representação do alfabeto**

O programa aceita textos somente em ASCII, uma limitação bastante aceitável, exceto para aplicações mais específicas. Além disso, é feito um hash das letras do padrão para inteiros entre 1 e o tamanho do alfabeto do padrão. Qualquer outro valor ASCII é mapeado para o inteiro 0.

- **Boyer-Moore**

O Boyer-Moore sofreu duas modificações. A primeira foi uma modificação da Regra do Bom Sufixo. A regra se comporta de forma semelhante, mas o caractere da posição que deu *mismatch* precisa ser diferente do caractere logo antes da substring que será alinhada com as posições que deram *match*.

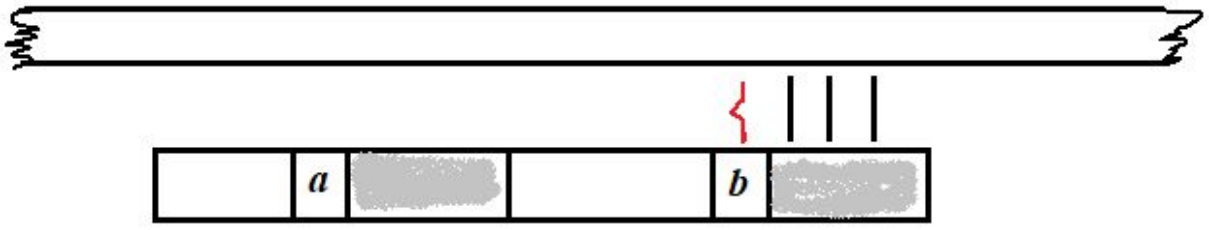


Figura 1 - Regra do bom sufixo modificada

A segunda foi a implementação de uma nova regra, a Regra de Galil. Com a adição dessa regra, o Boyer-Moore possui uma complexidade linear, mesmo no pior caso.

A Regra de Galil consiste na seguinte observação: suponha que no passo i, a extremidade direita do padrão está alinhada com a posição k do texto e que foi comparado até a posição s. Se, nesse passo, o padrão é movido de forma que sua extremidade esquerda esteja alinhada com uma posição no intervalo (s, k] por causa da Regra do Bom Sufixo então, no passo i + 1, se a comparação chegar à posição k, pode-se concluir que há uma ocorrência do padrão, sem precisar explicitamente comparar os caracteres à esquerda de k. A imagem abaixo mostra o caso em que ocorreu um *mismatch* na posição s. A regra também é válida no caso de um match.

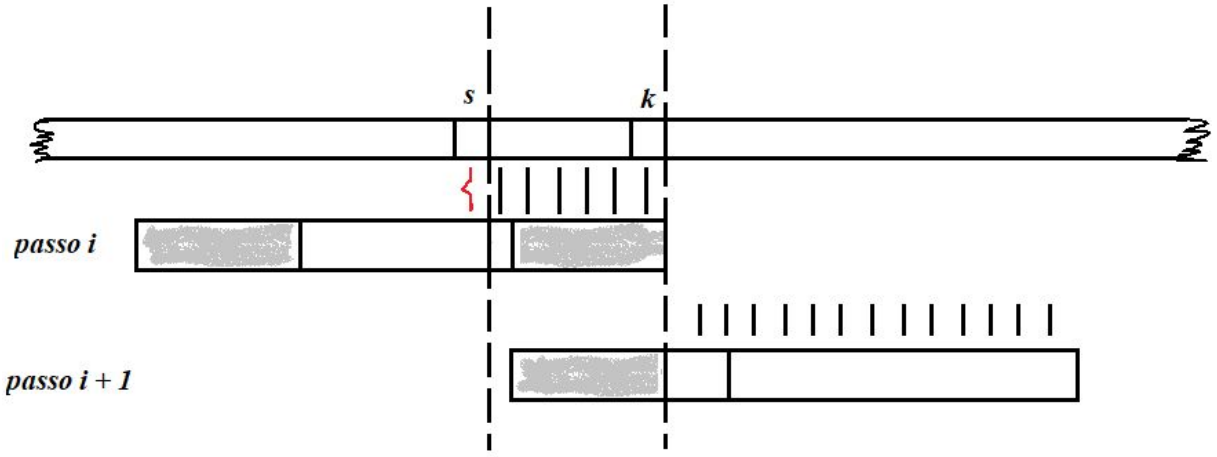


Figura 2 - Regra de Galil

- **Algoritmo Z**

Implementamos o Algoritmo Z, que serviu para facilitar a implementação da Regra do Bom Sufixo do Boyer-Moore. Entretanto, ele é apenas interno e sua funcionalidade como um algoritmo de casamento exato não foi integrada à ferramenta.

- **Trie**

Utilizamos uma trie no projeto. Esta foi representada como uma matriz M através de um vector de vector de inteiros. A matriz M tem dimensões k x $\sigma$, em que k é o número de nós da trie e $\sigma$ é o tamanho do alfabeto das cadeias codificadas na trie. Cada nó da trie é indexado por um inteiro, que corresponde também à linha da matriz do respectivo nó. Então se M[i][s] = j, isto quer dizer que há uma aresta do nó i para o nó j com rótulo s, ou M[i][s] = -1 se não há aresta partindo do nó i com rótulo s. A matriz cresce dinamicamente: sempre que um novo nó precisa ser adicionado à trie, uma nova linha é alocada.

- **Ukkonen**

Utilizamos uma árvore ternária para guardar as colunas correspondentes ao estado do algoritmo. Como cada coluna pode ser interpretada como uma string de

tamanho m sobre o alfabeto {-1, 0, 1}, esta árvore ternária foi implementada usando uma trie (descrita anteriormente) sobre este alfabeto. Neste algoritmo, cada estado do autômato corresponde a uma folha da trie e esta é identificada por um inteiro.

- **Aho-Corasick**

Suponha que no passo p, o algoritmo esteja no nó i e o caractere do texto sendo comparado seja c. Caso M[i][c] seja -1, isso significa que não há transições para esse caractere e então deve-se fazer uma transição para o nó de falha fail[i] e tentar novamente. Utilizamos uma estratégia de redirecionamento das transições da trie ao computar a função de falha. A modificação consiste em direcionar M[i][c] para M[fail[i]][c] ao construir a função de falha. Dessa forma, a transição será feita diretamente, sem necessidade de passar pelas transições de falha várias vezes. Isso não melhora a complexidade assintótica, porém torna o algoritmo um pouco mais eficiente e simples de escrever.

## 3. Testes e Resultados

Os testes conduzidos foram controlados por scripts escritos em bash, que invocava a ferramenta PMT com um dado algoritmo, para padrões de diferentes tamanhos, e redirecionava as informações sobre o tempo de execução. O tempo de execução considerado foi o tempo real obtido pela ferramenta GNU time. Todos os dados exibidos consideram a média dos tempos de execução obtidos para 4 diferentes padrões de um mesmo tamanho, cada um repetido 3 vezes.

Os mesmos testes foram feitos também com a ferramenta grep para casamento exato e agrep para casamento aproximado para efeito de comparação. Para os testes de análise de tempo de execução, todas as ferramentas foram usadas com a opção -c, ou seja, as saídas consistiram apenas nas quantidades de ocorrências.

Também realizamos testes para verificar corretude dos algoritmos. Para os algoritmos de busca exata, comparamos a contagem de linhas com ocorrências (opção -c) entre os algoritmos da nossa ferramenta e da ferramenta grep. Para os algoritmos de busca aproximada, comparamos a contagem de linhas com ocorrências entre os algoritmos da nossa ferramenta. Para comparação com os resultados da ferramenta agrep, comparamos explicitamente as linhas do texto com alguma ocorrência, pois o resultado da contagem (dada pela opção -c) da ferramenta agrep foi diferente do resultado obtido por nossos três algoritmos. As linhas impressas pelo agrep eram as mesmas impressas pelos nossos algoritmos, entretanto algumas linhas impressas pelo agrep estavam truncadas, o que pode indicar que o delimitador do agrep é diferente de um final de linha e portanto a contagem é diferente. Não foram encontrados erros durante esses testes.

Os testes foram feitos em um computador com sistema operacional macOS Sierra v10.12.6, processador 2.4 GHz Intel Core i5, memória 8GB 1600MHz DDR3 e Flash Storage. O compilador utilizado foi o GNU GCC v7.2.0.

## 3.1. Casamento exato

Para testar os algoritmos de casamento exato, foram utilizados um texto em inglês e uma amostra de DNA. Inicialmente testamos cada algoritmo separadamente, buscando um único padrão, e depois a ferramenta como um todo foi testada. Também testamos a busca de vários padrões, comparando o desempenho do Aho-Corasick com a ferramenta grep.

Para o primeiro teste foi utilizado um texto em inglês de tamanho 2.21 GB. Os padrões buscados foram palavras ou frases em inglês, algumas retiradas do texto. A busca foi feita com um único padrão por vez. O resultado deste teste está na Figura 3. É notável que para padrões com tamanho menor que 9, o tempo de execução do Boyer-Moore é bastante elevado e maior que o tempo de execução do Shift-Or. Para padrões com tamanho a partir de 9, o Boyer-Moore tem um tempo de execução bastante pequeno. O Aho-Corasick tem uma constante bastante elevada e portanto é desvantajoso quando um único padrão é buscado. A ferramenta grep tem o menor tempo de execução.

Para o segundo teste, foi utilizada um texto de DNA com tamanho 400MB. Os padrões buscados foram gerados aleatoriamente utilizando a função rand da biblioteca cstdlib. A busca foi feita com um único padrão por vez. O resultado obtido, mostrado na Figura 4, foi semelhante ao teste anterior, exceto que o desempenho do Boyer-Moore foi pior do que o desempenho do Shift-Or mesmo para padrões com tamanho a partir de 9. Isso se deve ao fato de que o alfabeto neste caso é pequeno, e portanto a regra do mau caractere não é tão eficiente. Mesmo assim, a diferença entre o tempo de execução do Boyer-Moore e do Shift-Or é pequena. É interessante notar que, nesses dois testes, o algoritmo básico de Bruteforce não fica muito para trás, sendo melhor até mesmo que o Aho-Corasick no primeiro teste. Isso mostra como sua complexidade de caso médio em textos não-artificiais é aproximadamente constante em relação ao tamanho do padrão.

O terceiro teste foi feito para testar o desempenho do Aho-Corasick com vários padrões buscados simultaneamente. Foram utilizadas palavras inglesas de tamanho 6 em quantidade variável. A busca foi feita num texto em inglês de tamanho 2.21 GB. O resultado está na Figura 5. Neste caso, o desempenho do Aho-Corasick foi superior ao da ferramenta grep para uma quantidade de padrões maior que 40, comprovando sua vantagem devido sobre os outros algoritmos quando buscamos vários padrões ao mesmo tempo.

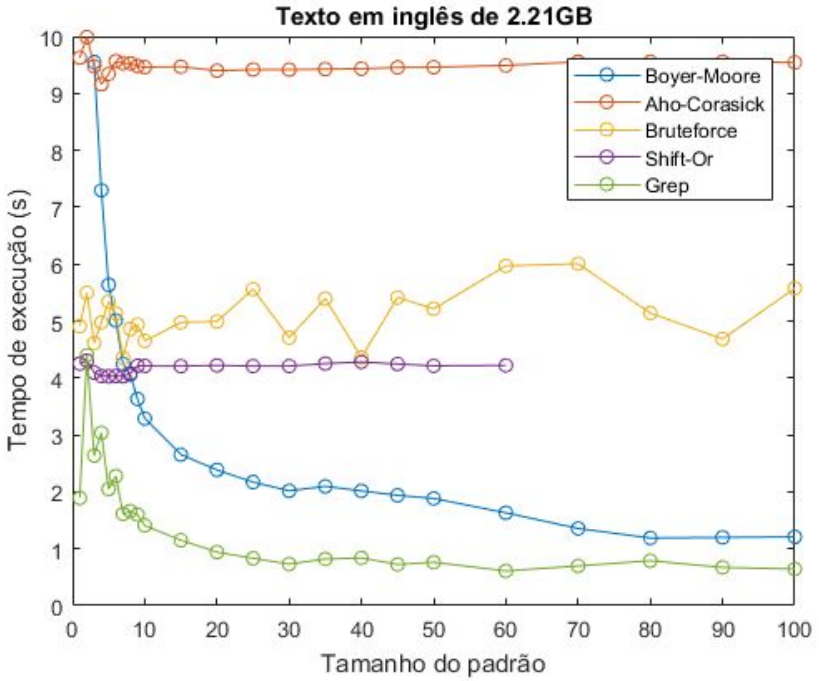


Figura 3 - Algoritmos de casamento exato, texto em inglês

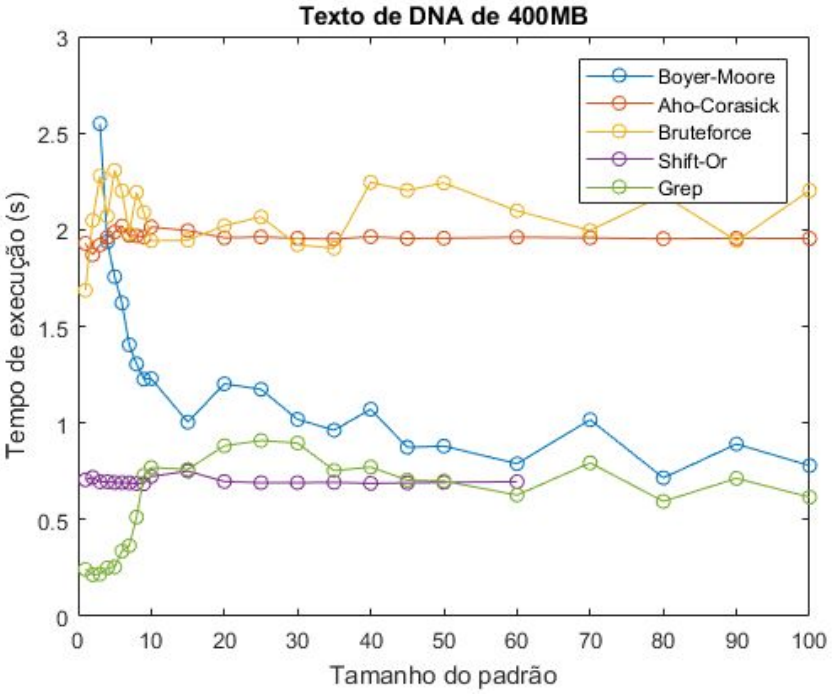

Figura 4 - Algoritmos de casamento exato, texto de DNA

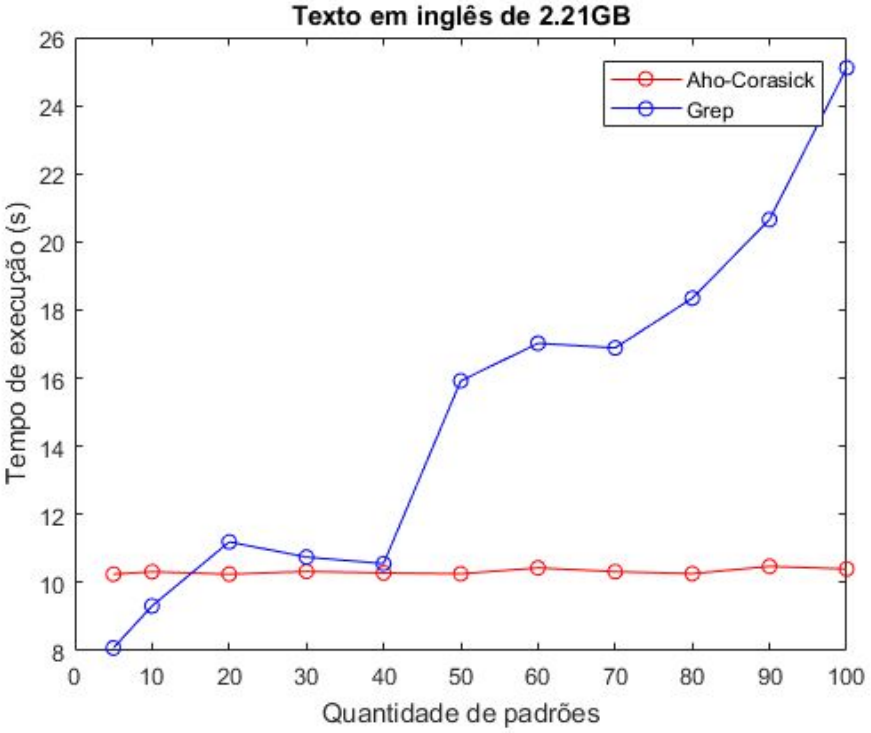


Figura 5 - Aho-Corasick com múltiplos padrões, texto em inglês

## 3.2. Casamento aproximado

Todos os testes de casamento aproximado foram feitos buscando-se um padrão por vez. Foram utilizados textos em inglês e de DNA.

O primeiro teste de casamento aproximado foi feito num texto em inglês de 100 MB. Foram buscadas palavras e frases em inglês. O erro foi definido como min(m - 1, 5), portanto para padrões de tamanho maior que 5 o erro foi o mesmo, e o fator que determinou a variação no tempo de execução foi o tamanho do padrão. A ferramenta agrep só foi testada até o tamanho 30, pois ela só funciona com padrões de tamanho no máximo 32. O resultado está na Figura 6. O tempo de execução do Wu-Manber se manteve aproximadamente constante, já que sua complexidade só depende do erro e do tamanho do texto, e estes eram os mesmos. O tempo de execução do Sellers cresceu linearmente com o tamanho do padrão, conforme esperado. O tempo de execução do Ukkonen cresceu polinomialmente com o tamanho do padrão, o que é esperado já que o tempo de pré-processamento tem complexidade proporcional a $m^{t+1}$ e o tempo de processamento é linear no tamanho do texto. Como o tamanho do texto não é tão grande, o tempo de pré-processamento domina o tempo de busca.

O segundo teste foi feito num texto em DNA de 100 MB. Os padrões buscados foram gerados aleatoriamente utilizando a função rand da biblioteca cstdlib. Esse teste teve o erro definido igual ao teste anterior. O resultado está na Figura 7. Nota-se que o gráfico ficou parecido com o teste anterior. Talvez a maior diferença seja na inclinação da curva do Ukkonen, que foi menor, já que o pré-processamento é proporcional a $\sigma^t$ e portanto menor. Além disso, o agrep não foi incluído neste teste, pois ocorria segmentation fault quando ele era executado e a causa não foi descoberta.

O terceiro e quarto teste foram, respectivamente, sobre os mesmos textos e padrões que os testes anteriores. Entretanto, o erro foi definido como m - 1. Estes testes foram feitos para evidenciar o tempo de execução exponencial no erro do pré-processamento do Ukkonen. Nota-se também que quanto maior o alfabeto, maior esse tempo. O agrep também não foi incluído nestes testes pois só aceita erros de no máximo 8.

O quinto teste foi sobre um texto em inglês de 2.21GB e com padrões de tamanho 20 e erro 5. O resultado está na Tabela 1. Em contraste com o primeiro teste, em que o Ukkonen teve um tempo similar ao Wu-Manber, neste teste, o Ukkonen foi duas vezes mais rápido. Como o processamento do Ukkonen é O(n) e o do Wu-Manber é O(tn), caso o texto seja grande o suficiente, o tempo gasto no pré-processamento pode valer a pena, como é o caso deste teste. Fixando o t = 5, o tempo de processamento do Wu-Manber possui uma constante elevada em relação ao Ukkonen.

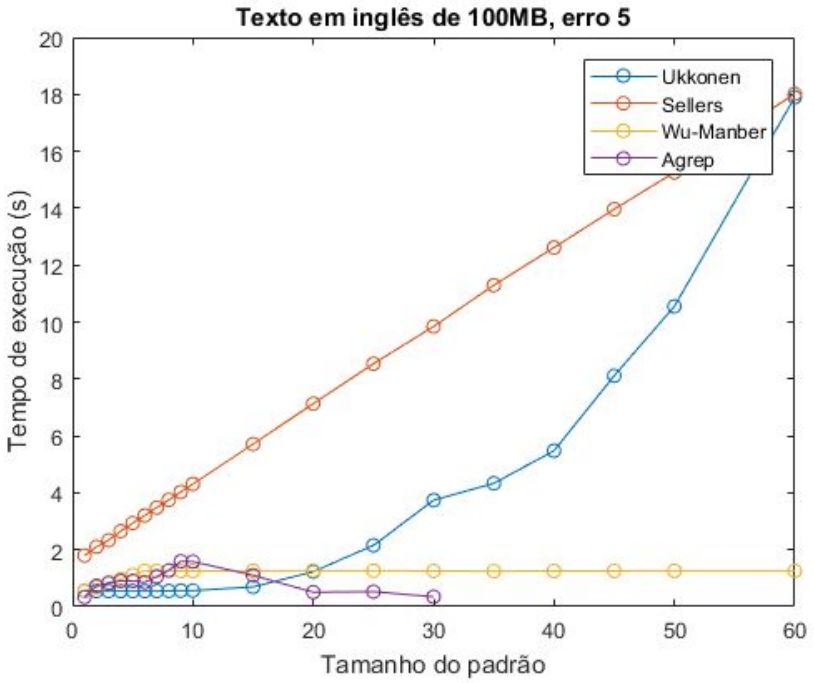


Figura 7 - Algoritmos de casamento aproximado, erro constante, texto em inglês

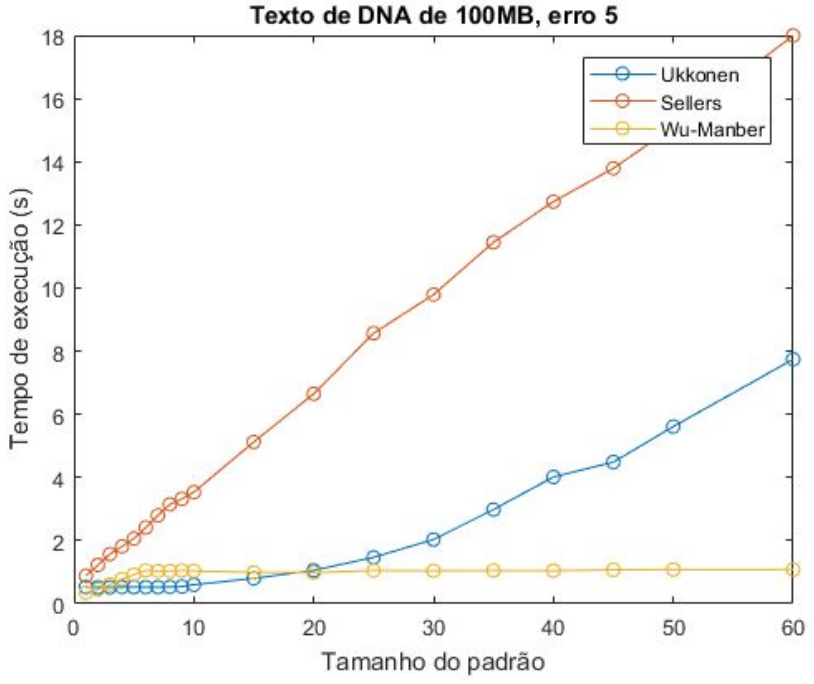


Figura 8 - Algoritmos de casamento aproximado, erro constante, texto de DNA

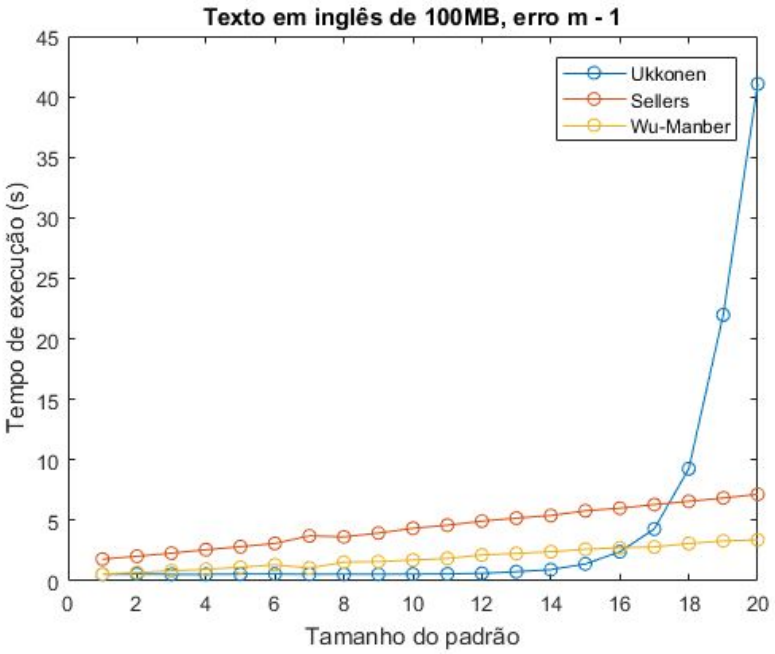


Figura 9 - Casamento aproximado, erro variável, texto em inglês

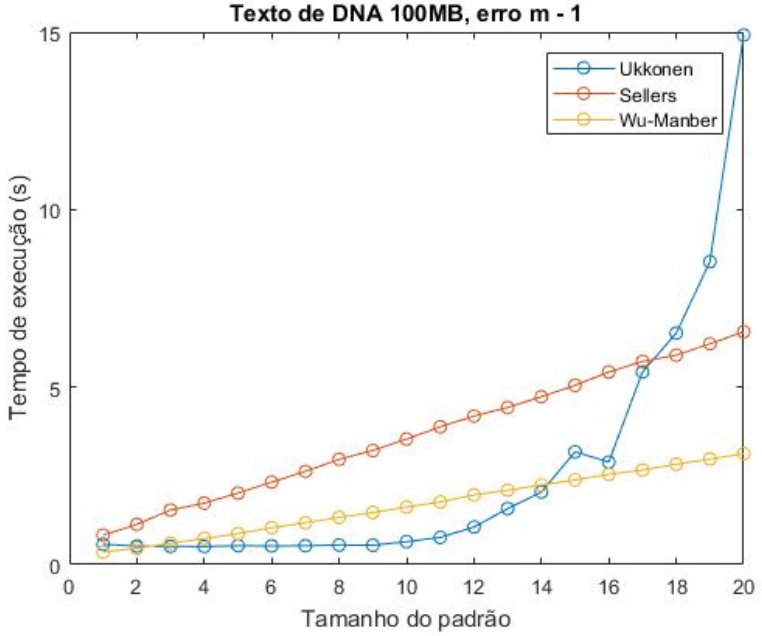


Figura 10 - Casamento aproximado, erro variável, texto de DNA

Tabela 1 - Execução sobre texto em inglês de 2.21GB, m = 20, t = 5

| | Ukkonen | Wu-Manber | Sellers | Agrep |
|---|---|---|---|---|
| Tempo de execução (s) | 11.78 | 25.88 | 144.21 | 10.96 |

## 4. Conclusões

Comparando os resultados de busca exata, concluímos que, para buscas de múltiplos padrões o Aho-Corasick é o melhor, superando até mesmo o desempenho da ferramenta grep. Para buscas de um único padrão, exceto para padrões de tamanho muito pequeno, o Boyer-Moore se mostrou superior. Nos casos em que o alfabeto é pequeno, como no DNA, valeria a pena usar o Shift-Or quando possível. Porém o ganho de desempenho seria pequeno e como estamos utilizando algoritmos online, é difícil saber de antemão o tamanho do alfabeto do texto. O tempo de execução do grep neste caso foi menor que todos os algoritmos da nossa ferramenta, exceto para tamanhos muito pequenos. Ainda assim, na maioria dos testes, nosso algoritmo mais rápido teve um tempo de execução não mais que o dobro do tempo do grep.

Para o casamento aproximado, concluímos que o Wu-Manber possui um desempenho bastante estável e eficiente, quando aplicável. Entretanto, o Ukkonen se destaca quando o texto é suficientemente grande para o tempo de processamento se sobrepor ao tempo de pré-processamento. Além disso, buscas com erros muito grandes não são tão interessantes, o que torna o Ukkonen um bom algoritmo para textos muito grandes. No caso de textos, padrões e erros muito grandes, o agrep não suporta este cenário e a ferramenta PMT se torna bastante lenta, pois faz uso do Sellers, já que o Wu-Manber implementado só funciona até padrões de tamanho 64 e o Ukkonen fica extremamente lento devido ao seu comportamento exponencial. Entretanto, cenários como esse não são comuns e, caso realmente existam, precisam de uma ferramenta mais específica.