FACULDADE DE TECNOLOGIA DE SÃO PAULO
Guilherme de Almeida Moreira

Algoritmos para Busca de Padrões: Uma Análise Comparativa Empírica

São Paulo
2012

FACULDADE DE TECNOLOGIA DE SÃO PAULO
Guilherme de Almeida Moreira

Algoritmos para Busca de Padrões: Uma Análise Comparativa Empírica

Trabalho submetido como exigência parcial
para a obtenção do Grau de Tecnólogo em
Processamento de Dados
Orientador: Dr. Silvio do Lago Pereira

São Paulo
2012

FACULDADE DE TECNOLOGIA DE SÃO PAULO
Guilherme de Almeida Moreira

Algoritmos para Busca de Padrões: Uma Análise Comparativa Empírica

Trabalho submetido como exigência
parcial para a obtenção do Grau de
Tecnólogo em Processamento de Dados.

Parecer do Professor Orientador: _____

_____

_____

Orientador:Dr. Silvio do Lago Pereira

São Paulo, _____ de Junho de 2012

Resumo

Devido ao grande volume de informações que é gerado hoje na internet, é essencial poder localizar informações dentro deste conteúdo. Porém com um volume de dados tão grande é importante estudar qual a melhor forma de realizar essas buscas. O objetivo deste trabalho é analisar cinco algoritmos diferentes de busca em texto dinâmico, aquele que não se tem informações prévias sobre ele.

Os cinco algoritmos são: Força Bruta, Rabin-Karp, busca por Autômato, Knuth-Morris-Pratt e Boyer-Moore.

Basicamente todos os algoritmos baseiam-se na comparação de uma janela do texto com o padrão. A janela é um conjunto de caracteres sequenciais do texto que tenha o mesmo tamanho do padrão. A diferença entre os algoritmos consiste em como eliminar algumas comparações e como movimentar a janela.

O algoritmo de Força Bruta tem a implementação mais trivial de todas, a comparação é feita caracter a caracter, caso algum deles não corresponda a janela é movida uma posição.

O algoritmo de Rabin-Karp tenta evitar a comparação de todos os caracteres através da comparação entre o hash da janela e do padrão, outro ponto chave deste algoritmo é a forma como é feito o deslocamento da janela. Através de alguns calculos que envolvem o descarte do primeiro caracter e a soma do próximo, não é necessário recalcular o hash inteiro da nova janela. A comparação dos caracteres das janelas é executada apenas quando os hash's corresponderem, com isso a quantidade de comparações diminui.

Na busca por Autômato o problema atacado é a comparação repetida de caracteres, que pode acontecer em Força Bruta e Rabin-Karp. Neste algoritmo cada caractere comparado muda o estado de um autômato, este autômato serve para avaliar se o prefixo atual pode ser reaproveitado.

Com Knuth-Morris-Pratt a base é a mesma do autômato, porém a analise de prefixos não é feita para cada possível transição de caracter, ela é feita apenas com o prefixo/sufixo do próprio padrão e a transição de estado é feita apenas quando o texto estiver sendo comparado com o padrão, na maioria das vezes melhorando o desempenho da busca.

O último algoritmo estudado é o Boyer-Moore, diferente de todos os outros algoritmos, neste a janela é comparada da direita para a esquerda. O ganho com essa mudança é o conhecimento obtido na comparação dos caracteres finais. Esses caracteres são analisados com duas heurísticas diferentes, a do bom sufixo, que confere se o sufixo ou parte dele ocorre novamente no padrão e alinha a janela com essa correspondência e a heurística do caracter errado, ela procura o caracter que não foi correspondido na pesquisa e procura uma ocorrência anterior no padrão, alinhando a janela com este caracter.

Dentre todos os algoritmos o que obteve melhor desempenho geral foi o Boyer-Moore, que inclusive é a implementação mais comum de algoritmo de busca, embora para cenários

onde o padrão e alfabetos são muito pequenos Força-Bruta e Autômato resolvem melhor o problema.

Abstract

Due to the large amount of content generated in Internet today, it is mandatory be able to find pieces of information inside this content. However with this large amount of content it is important to study which is the best way to perform this searches. The aim of this paper is to analyze five different algorithms of search in dynamic text, the one that does not have prior information about it.

The five algorithms are: Naive Search, Rabin-Karp, search by automata, Knuth-Morris-Pratt and Boyer-Moore.

Basically all this algorithms are based on comparison between the text window and the pattern. The window is a set of sequence of characters of same size of the pattern. The difference between the algorithms consists in how eliminate some comparisons and how to move the window.

The Naive algorithms is the most trivial implementation of them, the comparison is done character by character, whether some of them does not match, the window is moved by one position.

The Rabin-Karp algorithm tries to avoid the comparison between all the characters by comparing the window's hash and the pattern's hash, another key of this algorithm is how the window is moved. Enabled by some computation, that involves the first character disposes and the sum of the next one, it is not mandatory to re-calculate the entire new window hash. The comparison between the characters it is done only when the hash's are equivalent. therefore decreasing the comparisons done during the execution.

Searching by automata the main goal is to avoid the duplicate comparison, that can occur in Naive and Rabin-Karp algorithms. With this algorithm each compared character changes the state of the automata. This automata is used to evaluate whether the current prefix can be reused.

The Knuth-Morris-Pratt algorithm has the same foundation of the automata, however the prefix analysis is not done for all the possible transitions, it is done only in the prefix of the pattern and the computation of state transition is done just when the text and the pattern are being compared, in most of times improving the performance again the' automata.

The last studied algorithm it is Boer-Moore, not like all others algorithms, in this one the window is compared from right to left. This change can improve the overall performance because of the knowledge of the last characters. They are analyzed by two distinct heuristics, the good-suffix that checks if the suffix, or part of it, occur again in the pattern and align the window with this match. There is also the bad-character heuristic, that searches for the mismatch character align the window with this character.

From all algorithms the one with better overall performance was Boyer-Moore, it is the most common search algorithm implementation, nevertheless in some contexts, where

the alphabet and the pattern are small Naive and automata solve better the problem.

Sumário

Lista de Figuras

1 Introdução

1.1 Motivação

Estamos vivendo uma era chamada "Era da Informação", este nome sugere que hoje a ideia mais valorizada desta era é o acúmulo de informações, conhecimento.

Um dos instrumentos mais usados atualmente para gerar e acumular conhecimento é o computador. Devido a sua grande capacidade e com uma previsão de crescimento exponencial para os próximos anos, o computador é a principal forma de armazenamento de informação. Toda essa informação necessita de uma preocupação especial, pois não basta apenas guardar a informação, é essencial poder localizar uma informação desejada.

Com uma quantidade de dados pequena, provavelmente qualquer tipo de busca que for realizada não terá grande influência no desempenho da pesquisa, porém com um volume de dados maior é importante preocupar-se como essa pesquisa será feita.

As informações que o computador guarda podem variar de notícias de um jornal, documentos, textos literários, até informações sobre o DNA de uma simples bactéria ou mesmo de um ser humano. Boa parte dessas informações estão guardadas sob a forma de texto, por isso a manipulação de texto é um objeto de estudo muito importante, principalmente algoritmos de busca de padrões dentro de texto.

Mesmo que uma cadeia de DNA e um texto literário se tratem igualmente de textos, algumas características diferentes, como quantidade de caracteres disponíveis para cada um deles, faz com que eles tenham necessidades diferentes na hora de realizar as buscas. Essas necessidades acabam por exigir algoritmos diferentes.

1.2 Objetivo

Expondo melhor o problema, vamos observar a sequência de caracteres **T** da figura 1:

T = TGGTCAGTCAAGTCAGTTG

Figura 1: Exemplo de texto formado por uma cadeia de DNA.

Agora observe o padrão **P** na figura 2:

P = TCAAGTC

Figura 2: Exemplo de padrão formado por uma cadeia de DNA.

O problema consiste em dado o texto ***T*** *e o padrão* ***P*** *encontrar todas as ocorrências de* ***P*** *dentro de* ***T****.*

Uma ocorrência significa que o padrão procurado foi encontrado no texto em uma determinada posição. Na figura 3 abaixo, o padrão ocorreu na posição **7**. Podemos

representar esta ocorrência como **T[7...13]**. Essa notação mostra que a ocorrência aparece do caracter 7 ao 13.

Figura 3: Exemplo uma texto com uma ocorrência de um padrão.

Podemos generalizar todas as ocorrências do padrão da seguinte maneira:

Todo ***s*** *tal que* `T[s+1...s+m] = P[1..m]`*, onde* ***s*** *é o índice de cada ocorrência e* ***m*** *é o tamanho do padrão* ***P****.*

Com este problema proposto, o objetivo deste trabalho é analisar e comparar cinco algoritmos de busca de padrão diferentes, identificando quais deles tem um melhor desempenho em textos com cenários diferentes.

1.3 Metodologia

Diferentes metodologias foram usadas nesta pesquisa. Para a descrição dos algoritmos foi utilizada a pesquisa bibliográfica e implementação dos algoritmos na linguagem Java. Os testes empíricos foram realizados em diversos cenários, com o objetivo de explorar e expor as diferentes características de cada um dos algoritmos.

Todos os algoritmos estudados foram executados initerruptamente durante dois minutos cada, guardando a média do tempo de execução de cada etapa dos algoritmos.

1.4 Organização

Este trabalho está dividido em cinco seções e mais um anexo. Na primeira seção é feita uma introdução ao problema e uma descrição do trabalho. Na segunda seção tem como objetivo apresentar os conceitos fundamentais ao entendimento da pesquisa. Na terceira seção é feita a apresentação e descrição de cada um dos algoritmos tema deste trabalho. A quarta seção apresentará os resultados obtidos a partir da execução dos algoritmos. Na quinta seção será apresentada a conclusão alcançada a partir da descrição e dos resultados das execuções.

2 Fundamentos e Terminologia

2.1 Terminologia

Alguns conceitos serão usados no decorrer deste estudo e são fundamentais para o entendimento correto das próximas seções. O objetivo desta seção é explicar qual o significado de cada um dos termos a seguir, dentro do contexto desta pesquisa:

- **Caracter**:Símbolo que em sequência de outros caracteres representam uma informação maior.
- **String**: Sequência de caracteres.
- **Texto**: Sequência de caracteres que em geral guardam alguma informação e têm algum sentido. Neste trabalho representará o local onde procuraremos por um termo.
- **Palavra/Padrão**: Assim como um texto é uma sequência de caracteres que em geral também têm um significado, porém, muitas vezes menor que o texto. Neste trabalho representará o termo procurado dentro texto.
- **Alfabeto**: Conjunto de caracteres disponíveis para escrever um texto.
- **Janela de comparação**: Ao procurar por um padrão em um texto, todos os algoritmos deste trabalho visam comparar o padrão com uma porção do texto de mesmo tamanho, esta porção do texto que será comparada é chamada de janela.
- **Substring**: Uma sequência de caracteres que esteja contida dentro de uma String.
- **Prefixo**: Uma sequência de caracteres que esteja contida dentro de uma String e que corresponda aos primeiros caracteres da String.
  Exemplo:
  String = 'Carro'
  Prefixos = ['', 'C', 'Ca', 'Car', 'Carr', 'Carro']
- **sufixo**: Uma sequência de caracteres que esteja contida dentro de uma String e que corresponda aos últimos caracteres da String.
  String = 'Carro'
  Prefixos = ['', 'o', 'ro', 'rro', 'arro', 'Carro']
- **Prefixo próprio**: Uma sequência de caracteres que esteja contida dentro de uma String e que corresponda aos primeiros caracteres da String e que não seja a própria String.
  String = 'Carro'
  Prefixos = ['', 'C', 'Ca', 'Car', 'Carr']

- **sufixo próprio**: Uma sequência de caracteres que esteja contida dentro de uma String e que corresponda aos últimos caracteres da String e que não seja a própria String.
  String = 'Carro'
  Prefixos = [", 'o', 'ro', 'rro', 'arro']

2.2 Siglas

- DNA: Ácido Desoxirribonucleico, composto orgânico que contém instruções genéticas que coordenam o desenvolvimento e funcionamento de seres vivos e alguns vírus.[1]
- Gb: Gigabyte, unidade de medida de dados equivalemente à 1024 Megabytes ou 8589934592 bits.
- KMP: Knuth-Morris-Pratt, nome de um dos algoritmos estudos nesse trabalho.

2.3 Fundamentos

Os algoritmos deste estudo se concentrarão em buscar ocorrências em textos que não se conhece previamente nenhuma informação dele, exceto o seu tamanho. Estes textos são chamados de textos dinâmicos, pois eles podem mudar sem que isso influencie a forma de buscar ocorrências. Os algoritmos de busca em texto dinâmico deste trabalho utilizam um método de busca por janela. Uma janela, como mostrada na figura 4, significa um trecho do texto que será comparado com o padrão naquele momento.

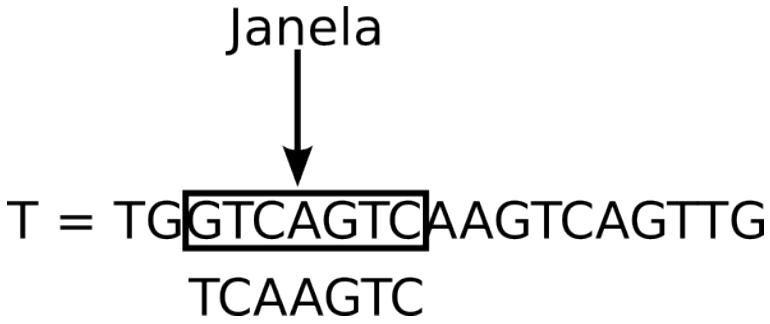

Figura 4: Um trecho do texto sendo comparado, formando uma janela.

A função das janelas além de delimitar os caracteres que serão comparados, é também poder ser movida dentro do texto para que seja possível procurar as ocorrências em todo o texto. Vejamos um exemplo nas figuras 5, 6 e 7 de movimentação de janela:

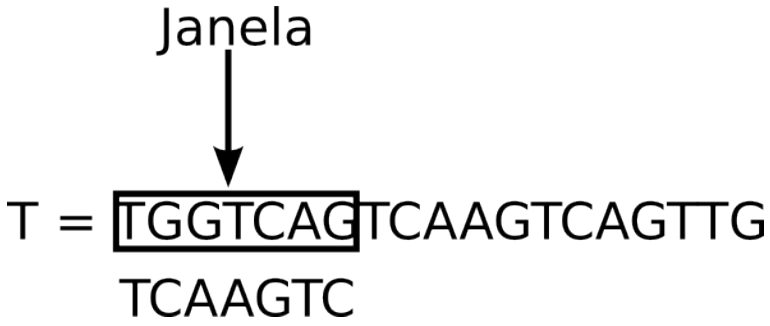

Figura 5: Texto sendo comparado desde a posição 1 do texto.

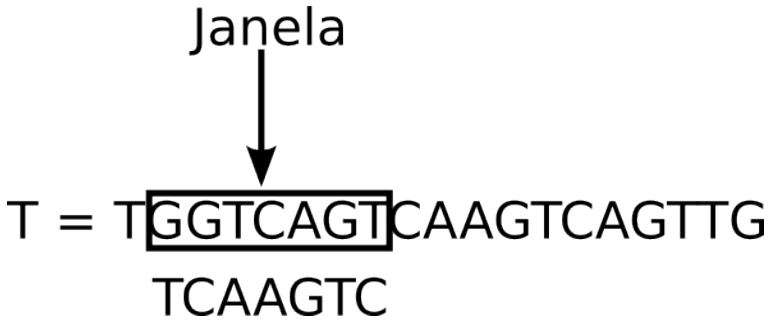

Figura 6: Texto sendo comparado desde a posição 2 do texto.

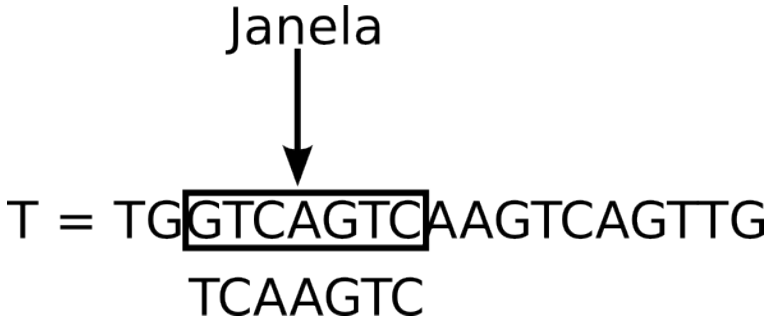

Figura 7: Texto sendo comparado desde a posição 3 do texto.

Percebemos que a cada passo a janela está em uma posição diferente do texto, a cada passo temos a chance de encontrar uma nova ocorrência.

Embora não seja nosso objeto de estudo, ainda existem os textos estáticos,por não mudarem podemos analisá-lo intimamente e assim a forma de pesquisa leva em consideração este conhecimento para um melhor desempenho nas buscas.

3 Algoritmos de Busca de Padrões

A busca que iremos tratar agora é a busca referida na seção anterior, achar as ocorrências de um determinado padrão dentro de um texto dinâmico. A principal diferença entre os algoritmos será o quanto cada um deles proporcionará de deslocamento da janela. Cada algoritmo faz a movimentação de maneira diferente, existem algoritmos que começam do final do texto, algoritmos que iniciam do começo do texto e algoritmos que começam em posições chave para ele. Até dentro da comparação da janela o algoritmo pode comparar do começo para o fim ou do fim pro começo. Neste trabalho serão analisados, testados e comparados cinco algoritmos diferentes: Força-Bruta, Rabin-Karp, Autômatos, Knuth-Morris-Pratt e Boyer-Moore.

3.1 Força-Bruta

O primeiro algoritmo que será objeto de estudo é o algoritmo mais trivial de todos, ele é chamado de **Força-Bruta**[2][3] pois testa todas as janelas possíveis dentro do texto, uma a uma. Posicionamos a janela no começo do texto e tentamos casar a janela com o padrão como na figura 8, caracter a caracter, se todas as letras forem iguais, figura 12 acusamos uma ocorrência. Caso algum caracter seja diferente, paramos a comparação a deslocamos a janela uma posição para a frente e recomeçamos a comparação da janela com o padrão, como na sequencia de figuras 9, 10 e 11, assim até chegarmos na última janela.

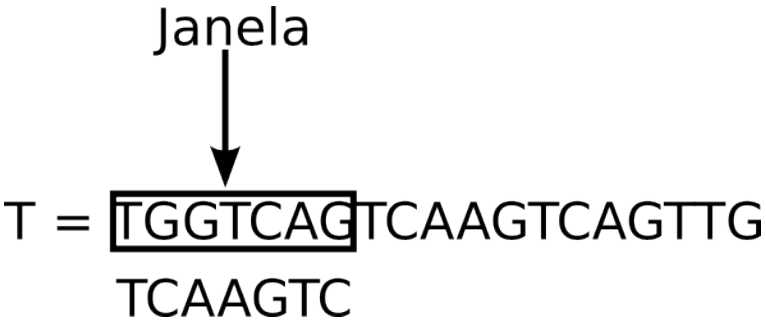

Figura 8: A janela de comparação na primeira posição.

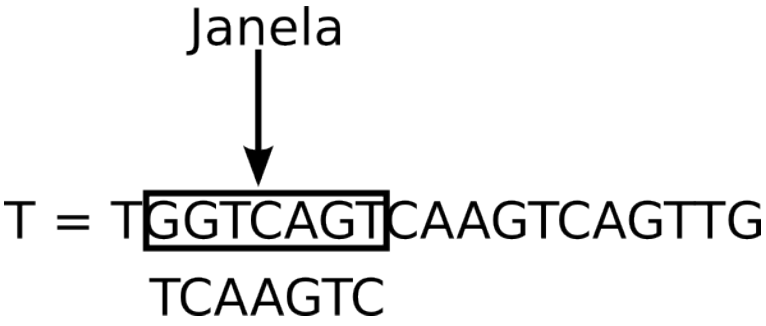

Figura 9: A janela de comparação na segunda posição.

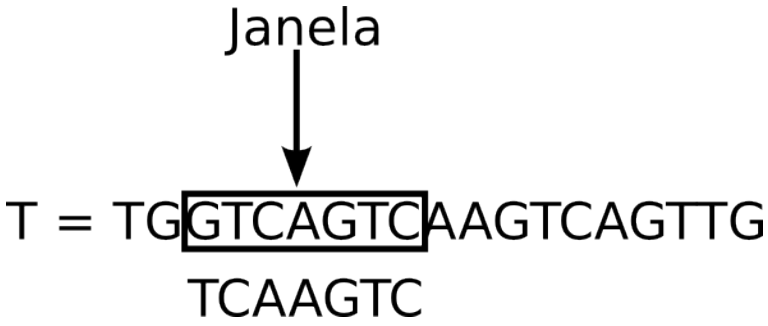

Figura 10: A janela de comparação na terceira posição.

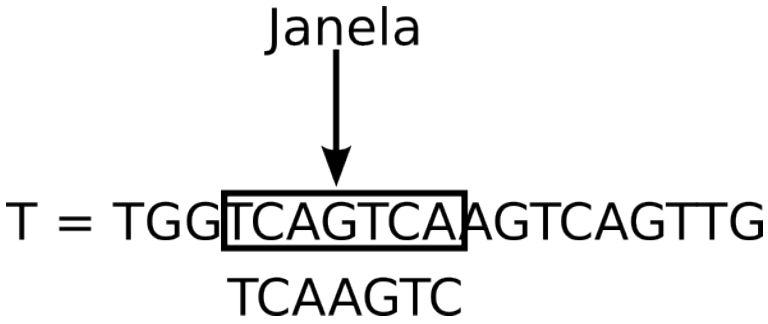

Figura 11: A janela de comparação na quarta posição.

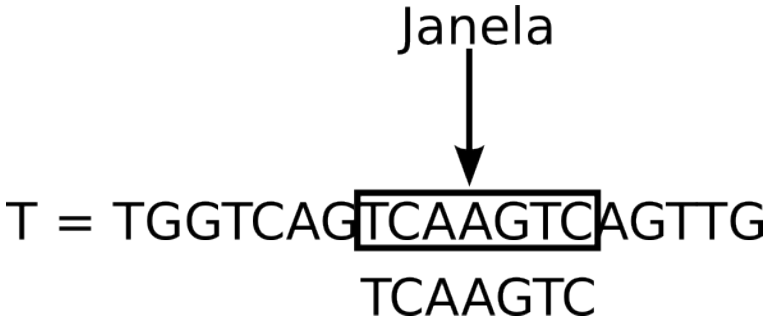

Figura 12: Uma ocorrência na posição 8.

```
List<Integer> busca() {
  List<Integer> integers = new ArrayList<Integer>();
  int n = texto.length;
  int m = padrao.length;
  for (int i = 0; i < n - m + 1; i++) {
    int j = 0;
    while (j < m && texto[i + j] == padrao[j]) {
      j++;
    }
    if (j == m) {
      integers.add(i);
    }
  }
  return integers;
}
```

O algoritmo de Força-Bruta apresenta uma complexidade assintótica de O((n-m)m) no pior caso. Este algoritmo é ineficiente na maioria dos casos, pois ele dispensa as informações das comparações já realizadas nas comparações futuras, porém para textos e padrões pequenos é uma opção comum, já que não gasta tempo processando analisando o padrão.

3.2 Rabin-Karp

3.2.1 Introdução

O algoritmo de Rabin-Karp[4][3] apresenta uma evolução ao algoritmo de Força-Bruta, ao invés de comparar a janela com o padrão caracter a caracter, em Rabin-Karp, é proposto usar uma comparação numérica entre os restos da divisão por um mesmo inteiro da janela e do padrão. Uma comparação apenas entre numeros pequenos é mais rápida que a comparação de todos os caracteres.

O algoritmo de Força-Bruta a cada deslocamento pode acabar fazendo muitas comparações antes de descobrir que uma janela não é uma ocorrência, neste caso esse custo para comparar esses caracteres foi jogado fora. O algoritmo de Rabin-Karp tenta evitar a comparação caracter a caracter das janelas que não são ocorrências.

Para evitar as comparações caracter a caracter, Rabin-Karp propõe que toda comparação de janela, antes, passe por uma verificação numérica entre o padrão e a janela. Nas janelas que não são ocorrências há uma grande chance de seu valor numérico ser diferente do valor do padrão e, neste caso, dado a diferença do valor numérico, não há necessidade da comparação caracter a caracter.

Com isso o valor numérico da janela e do padrão devem respeitar uma característica importante: **Quando o valor numérico da janela e do padrão forem diferentes, necessariamente o conteúdo da janela e do padrão também devem ser diferentes.** Garantindo esta característica toda vez que os valores numéricos forem diferentes, a janela atual poderá ser descartada, sem a comparação caracter a caracter.

Esta característica pode ser facilmente alcançada, por exemplo, atribuindo um valor numérico a cada letra do alfabeto e somando-os.
Na figura 13 consideramos a=1, b=2, c=3 e d=4:

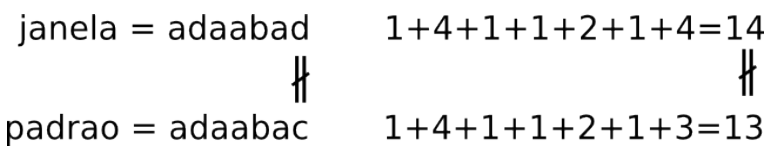

Figura 13: O valor numérico do texto na janela é 14, no padrão é 13, com isso a janela e o padrão são diferentes.

Esta característica apenas garante o descarte da janela quando os valores forem diferentes, porém, quando a soma dos caracteres resultar no mesmo valor pode ou não haver

igualdade entre a janela e o padrão, como mostrado nas figuras 14 e 15:

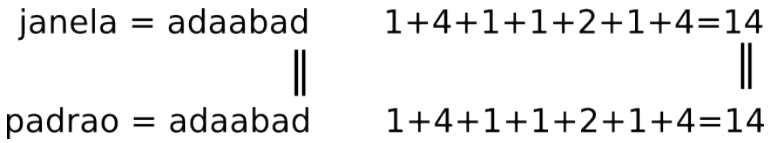

Figura 14: Tanto janela quanto padrão tem o mesmo valor 14, pois o texto das duas é o mesmo

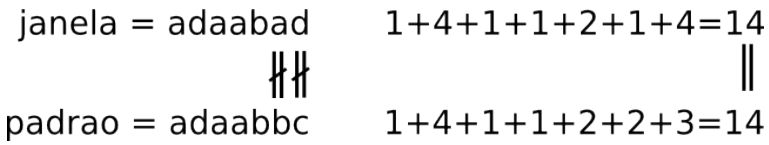

Figura 15: O valor da janela e do padrão são iguais, porém é um falso igual, pois a janela e o padrão são diferentes.

Como visto na figura 15, quando os valores forem iguais, porém o texto não, este cenário é chamado de *spurious hit* ou correspondência falsa, neste caso então a janela e o padrão devem ser comparados caracter a caracter, levando o algoritmo a uma complexidade igual ao de Força-Bruta O((n-m)m), porém com um agravante de que toda a janela além da comparação, há ainda a fase de cálculo do valor numérico.

3.2.2 Otimizações

Neste algoritmo é possível fazer duas otimizações, a primeira, ao invés de comparar a soma dos caracteres, que pode acabar virando uma comparação caracter a caracter novamente, é possível fazer a comparação entre os restos da divisão por um mesmo número, obtendo assim um número menor. A outra otimização é ao invés de obter o valor numérico com a soma dos números, usar algum algoritmo de *hash* que preferêncialmente permita o *rolling hash*.

3.2.3 Usando o resto da divisão por um inteiro

A primeira otimização possível visa evitar que números onde só os últimos algarismos sejam diferentes acabem virando novamente comparação caracter a caracter. Para evitar este cenário, é usado uma operação, que respeite a característica principal de que números diferentes apontem textos diferentes. Essa operação é a de resto da divisão por um inteiro. Dois inteiros iguais divididos pelo mesmo número inteiro resultarão em um mesmo resto. Dois números diferentes podem acabar dando restos iguais ou diferentes.

Essa otimização, então, possui uma característica desejável e uma indesejável, a primeira é que este resto da divisão será um número menor, assim sendo mais rápido fazer a comparação. A característica indesejável é que agora a chance de ocorrência de números iguais será maior. Porém a primeira característica é utilizada em todas as janelas do texto,

já a segunda, ocorrerá com menos frequência, por isso esta otimização ajuda a diminuir o volume de processamento total deste algoritmo.

3.2.4 Utilização de Hash em Rabin-Karp

O objetivo da segunda otimização é diminuir a chance de obter números iguais ao obter o valor numérico da janela e do padrão. Ao observar por exemplo o texto "abc"podemos reparar que numericamente, considerando a=1, b=2 e c=3, a soma resulta em 6, porém a string "bbb"também resulta em uma soma 6.

Para evitar essa colisão numérica ao invés de usar a simples soma dos números, podemos usar alguma função de *hash*, originalmente o algoritmo de Rabin-Karp usa $p[0] \times base^{m-1} + p[1] \times base^{m-2} + ... + p[m-1] \times base^0$. A escolha deste algoritmo de *hash* é em função da possibilidade de praticar o *rolling hash*.

Ao invés de a cada janela recalcular seu hash sem nenhuma informação anterior, o que levaria a um processamento de tempo O(m) e o algoritmo todo O(nm), com *rolling hash*, o *hash* da próxima janela é feito excluindo do *hash* anterior o valor que o primeiro caracter representa neste *hash*, depois multiplica-se novamente pela base, promovendo cada caracter a uma nova posição e por fim somando com o valor do próximo caracter, fazendo com que a janela se mova uma posição.

A base usada na fórmula deve ser o tamanho do alfabeto, para que ao multiplicar um número por esta base, o resultado seja o deslocamento em uma posição deste número. Por exemplo, escolhendo o número 54398, ao multiplicá-lo pelo tamanho do alfabeto decimal, 10, temos 54398×10 que produz 534980. Observando que os algarismo permaneceram os mesmo, exceto que agora a última posição temos um 0 a mais.

A maneira mais simples de alcançar este objetivo é usar nas contas o valor numérico que cada caracter representa. Por motivo de simplicidade, considerando o alfabeto dos números decimais, temos o conjunto Σ=0,1,2,3,4,5,6,7,8,9 e usando seus valores temos o exemplo abaixo.

Para descobrir o valor numérico da janela 561, podemos praticar a seguinte conta:

$$561 = 5 \times 10^2 + 6 \times 10^1 + 1 \times 10^0$$

A primeira ordem é sempre 10 (tamanho do alfabeto) elevado à quantidade total de caracteres menos um, a próximas grandeza serão 10 elevado à grandeza anterior menos um.

$$5612 = 5 \times 10^{4-1} + 6 \times 10^{4-2} + 1 \times 10^{4-3} + 2 \times 10^{4-4}$$

Com isso cada janela pode ter seu valor calculado com o seguinte algoritmo:

```
for(int i = 0; i < m; i++){
  j = j + texto[i]*Math(base, m-i);
}
```

Para calcular a próxima janela devemos retirar do valor 561 o valor 500, para sobrar apenas o 61. 500 pode ser obtido multiplicando o valor do caracter por 10^2, generalizando temos T[i-m]$\times$base^{m-1}.

Depois de retirar o valor do digito da janela anterior é preciso mover os próximos caracteres uma casa, basta multiplicar o restante (61) pelo tamanho da base (10), obtendo 610.

A etapa final para mover a janela é somar o valor do próximo caracter para preencher o último 0 do *hash* anterior. Desta maneira movemos a janela gastando um tempo constante, para calcular o *hash* da próxima janela, essa operação de mover o *hash* para a próxima posição é chamada de *rolling hash*.

3.2.5 O algoritmo final

O algoritmo final pode ter pequenos ajustes como pré-calcular o *hash* do padrão, até mesmo calcular usando a mesma iteração o *hash* da primeira porção de tamanho **m** do texto. Outra otimização importante é deixar já pré-calculado o valor da base^{m-1}, pois o valor dela será o mesmo em todas os deslocamentos da janela.

Finalizando o algoritmo temos ele completo, onde `P` é o padrão, `T` o texto `d` o tamanho do alfabeto e `q` um número primo:

```
List<Integer> busca() {
  int n = texto.length;
  int m = padrao.length;
  int h = 255;
  int primo = 31;
  int base = 1;
  for (int i = 0; i < m-1; i++) {
    base*=h;
  }

  int hashTexto = 0;
  int hashPadrao = 0;

  for (int i = 0; i < m && i < n; i++) {
    hashTexto=hashTexto*h+texto[i];
    hashPadrao=hashPadrao*h+padrao[i];
  }

  hashPadrao%=primo;

  if(hashTexto%primo==hashPadrao){
    int k=0;
    while(k<m && padrao[k]==texto[k]){
      k++;
    }
    if(k==m){
      resultados.add(0);
    }
  }

  for (int i = m; i < n; i++) {
```

```
    hashTexto=hashTexto-texto[i-m]*base;
    hashTexto*=h;
    hashTexto+=texto[i];
    if(hashTexto%primo==hashPadrao){
      int k=0;
      while(k<m && padrao[k]==texto[i-m+1+k]){
        k++;
      }
      if(k==m){
        resultados.add(i-m+1);
      }
    }
  }

  return resultados;
}
```

3.3 Autômatos

3.3.1 Introdução

Uma das falhas do algoritmo de força-bruta é que ele repete diversas comparações entre os caracteres do texto e do padrão. Isso acontece porque cada vez que ele avança a janela para a próxima posição ele ignora todas as comparações anteriores e começa a comparar desde o primeiro caracter da janela de novo, mesmo que o conhecimento gerado pela comparação da janela anterior indique que não é necessário comparar toda janela de novo. O algoritmo de Rabin-Karp possui essa mesma característica, pois mesmo que a comparação seja mais rápida ela ainda ignora qualquer conhecimento gerado na comparação anterior.

Usando um autômato[2] para fazer a busca no texto, ele permite que uma sequência de caracteres seja processada para determinar se a mesma respeita determinadas condições que permitam não comparar todos os caracteres de novo ao deslocar a janela. Para isso, ele possui um conjunto de estados e cada estado representa uma determinada situação.

Por exemplo, suponha que desejamos saber se um texto possui pelo menos duas ocorrências do caracter "a". Poderíamos definir um autômato com três estados: A, B e C. No estado A **nenhuma** letra "a"foi lida, no B **uma** letra "a"foi lida, no C pelo menos **duas** letras "a"foram lidas.

Além disso, os autômatos são baseados fortemente em transição de estados, pois eles processam a entrada caracter a caracter e a cada um deles provoca uma transição de estado (mesmo que o próximo estado seja o mesmo do anterior, isso também é chamado de transição). Por exemplo, suponha que o autômato anterior esteja no estado A, se o próximo caracter for um "a"ocorre uma transição para o estado B senão o autômato se mantem no estado A.

É possível criar um autômato que determina se uma determinada palavra ocorre em um texto. Na figura 16, o padrão procurado é o **"ababaca"**.

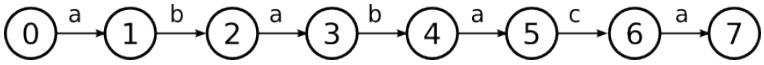

Figura 16: Sequência de estados a cada letra comparada na palavra ababaca.

No estado **0**, o inicial, caso o próximo caracter encontrado seja o **a**, o autômato vai para o estado **1**. No estado **1**, se for encontrado qualquer caracter diferente de **b** o autômato volta para o estado **0** e a janela é movimentada em uma posição, isso se repete até o estado final, **7**, onde é acusada uma ocorrência. Esse processo é o mesmo do algoritmo de força-bruta, pois a qualquer erro, o autômato volta para o estado inicial e a janela se move **uma** posição.

Usar um autômato para fazer buscas de padrão em texto, envolve mais do que apenas montar a sequência de estados, ele se torna melhor que força-bruta a partir do momento que possamos identificar transições que não voltem diretamente para o estado **0**.

Por exemplo, se o autômato estiver no estado **5**, de acordo com o padrão, o próximo caracter deveria ser o **c**. Porém se no texto for encontrado um caracter **b**, significa que o texto encontrado é **ababab**, como no exemplo da figura 17.

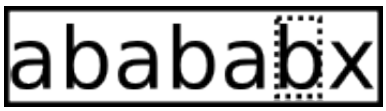

Figura 17: O autômato está no estado 5, com isso 5 caracteres foram correspondidos.

Repare que as quatro últimas letras do trecho *ab**abab*** casam com as quatro primeiras letras do padrão procurado ***abab**aca*. Dessa forma os dois primeiros caracteres poderiam ser descartados e a comparação pode prosseguir a partir do quinto caracter do padrão, avançando a janela duas posições (figura 18).

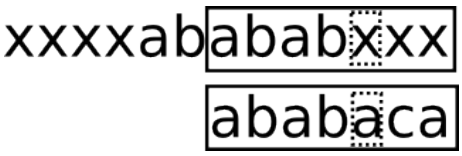

Figura 18: O autômato está no estado 4, com isso 4 caracteres foram correspondidos.

De acordo com a andamento da busca de forma otimizada, se encontrarmos o caracter **b** e o autômato estiver no estado **5**, voltamos para o estado **4** (figura 19 ao invés do estado **0**, como proposto pelo força-bruta.

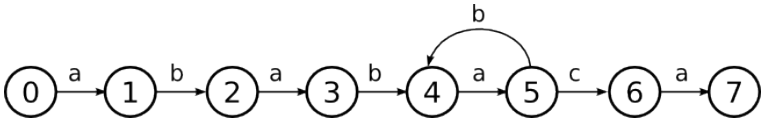

Figura 19: Representação de uma transição que não volta para o estado inicial, mas sim sai do 5 para o 4.

Essa ideia deve ser aplicada para calcular todas as transições de cada estado. Mais precisamente, para cada caracter do alfabeto e para cada estado, deve ser calculada uma transição.

3.3.2 Construção do Autômato

O autômato terá **m+1** estados *(q0, q1, ..., qm)*. Se o autômato estiver no estado *qi* significa que ele acabou de ler os caracteres *p1, p2, ..., pi (p[1...i]).*

As transições serão calculadas percorrendo os estados na ordem *q0, q1, ..., qm.* Quando estivermos no estado *qi*, calcularemos as transições desse estado considerando todos os caracteres do alfabeto (cada caracter gera uma transição), pois durante a busca no texto, o próximo caracter pode ser qualquer caracter do alfabeto. Seja σ um caracter do alfabeto, há duas possibilidades: na primeira, σ é igual a p[i+1], então o autômato assume o estado qi+1. Na segunda, σ não é igual a p[i+1], então descobriremos para qual estado qj o autômato deve voltar, sendo que j deve ser menor que i. Essa análise será feita considerando os estados na ordem *qi, qi-1, ..., q0.* Para decidir se devemos voltar para o estado *qj* é necessário testar se o trecho *p[1...j]* é sufixo do que já foi lido pelo autômato, ou seja, se é sufixo do trecho $p[1...i]\sigma$.

```
int[][] constroiAutomato(char[] P, char[] alfabeto, int maiorIndice) {
  int[][] automato = new int[P.length + 1][maiorIndice + 1];

  for (int estado = 0; estado <= P.length; estado++) {
    for (int sigma = 0; sigma < alfabeto.length; sigma++) {
      if (estado < P.length && alfabeto[sigma] == P[estado]) {
        automato[estado][alfabeto[sigma]] = estado + 1;
      } else {
        int deslocamento = 1;
        while (deslocamento <= estado) {
          int i = 0;
          for (; i <= estado - deslocamento
              && i + deslocamento < P.length; i++) {
            if (P[i] != P[i + deslocamento]) {
              break;
            }
          }
          if (i == estado - deslocamento
              && P[i] == alfabeto[sigma]) {
            automato[estado][alfabeto[sigma]] = i + 1;
            deslocamento = estado + 1;
          }
          deslocamento++;
        }
      }
    }
  }
  return automato;
}
```

3.3.3 O algoritmo de busca com Autômato

Uma vez criado o autômato para auxiliar a busca do padrão no texto, é simples definir um algoritmo para realizar a tarefa de busca. Cada caracter do texto deve ser passado para o autômato e toda vez que o último estado for alcançado significa que uma ocorrência do padrão foi encontrada.

```
List<Integer> busca() {
  int estado = 0;
  int n = texto.length;
  int m = padrao.length;
  for (int i = 0; i < n; i++) {
    estado = automato[estado][texto[i]];
    if (estado == m) {
      resultado.add(i-m+1);
    }
  }
  return resultado;
}
```

3.4 Knuth-Morris-Pratt

A busca por autômato possui dois pontos que podem ser melhorados o primeiro é a quantidade de memória que ele consome e o segundo é o cálculo desnecessário de algumas transições que nunca ocorrerão.

O pré-processamento do autômato tem a função de descobrir se cada sufixo de cada prefixo do padrão, acrescentado de um caracter corresponde à um outro prefixo do padrão. Este cálculo é feito para todos os caracteres do alfabeto.

Em KMP[5] ao invés de pré-calcular todas as transições possíveis de todos os caracteres, é pré-calculado apenas o que pode ser reaproveitado da comparação anterior, independente do próximo caracter. Fazendo analogia ao autômato, o próximo estado do autômato gerado pelo próximo caracter é calculado apenas na hora em que é lido este caracter.

O que pode ser reaproveitado das comparações é tudo aquilo que for prefixo e sufixo dos prefixo gerado até a posição do padrão que já correspondeu ao texto (figura 20). Ao descobrir esse conjunto de prefixos/sufixos é possível eliminar comparações já realizadas, sem ter que prever e guardar em memória todas as transições possíveis.

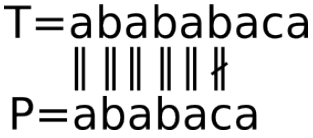

Figura 20: Texto e padrão falham na comparação na quinta posição.

Caso fosse usado o algoritmo de força-bruta para localizar o padrão no texto deslocaríamos a janela uma posição e começaríamos a comparação do zero:

Porém sabendo que a posição 5 foi a que fez não encontrar uma ocorrência, da posição 0 a 4 o padrão e o texto são iguais. Considerando apenas o texto da posição 0 à 4 o prefixo é "ababa". Este prefixo possui um sufixo que é prefixo também, conforme figura 21:

Figura 21: As 3 primeiras letras do padrão ocorrem uma segunda vez duas posições à frente.

Analisando apenas o prefixo correspondido na pesquisa já possível prever que o deslocamento de uma posição não será uma ocorrência, vide figura 22, pois o prefixo correspondido não possui um sufixo/prefixo no padrão.

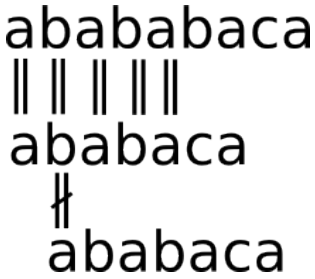

Figura 22: Ao deslocar apenas em uma posição o padrão, com certeza não será uma ocorrência.

Com este prefixo o menor deslocamento com chance de ser uma ocorrência é de duas posições, pois o prefixo correspondido tem um sufixo que também é seu próprio prefixo, veja nas figuras 23 e 24:

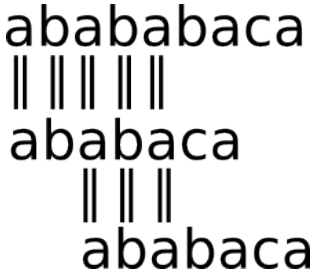

Figura 23: Deslocamento de 2 posições tem chance de ocorrência dado o conhecimento prévio do prefixo

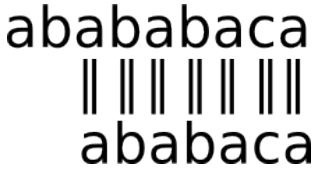

Figura 24: Deslocamento de 2 posições proporcionou uma ocorrência

Este processo se assemelha ao do autômato, o autômato calcula qual o próximo estado, concatenando o que já foi comparado com possível próximo caracter. Em KMP consideramos apenas o que já foi comparado e não o próximo caracter, por isso o pré-processamento que KMP realiza independe do tamanho do alfabeto, apenas os caracteres do padrão serão analisados.

3.4.1 Pré-processamento de KMP

Para otimizar a pesquisa em KMP o pré-processamento deve analisar todas os prefixos do padrão procurando pelo maior sufixo destes prefixos que também seja prefixo.

Observando como exemplo o padrão `ababaca` ao analisar apenas o prefixo `abab`, o maior sufixo que também é prefixo é `ab` que termina na posição 1, como na figura 25.

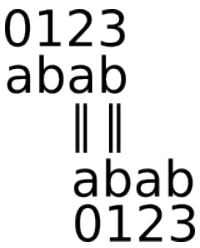

Figura 25: O prefixo 'ab' também ocorre na posição 2 do padrão.

Calculando esta informação para todas as posições temos os seguintes resultados:

0	**1**	**2**	**3**	**4**	**5**	**6**
a	b	a	b	a	c	a
-1	-1	0	1	2	-1	0

Analisando o código da função de pré-processamento:

```
int[] calculaPrefixos(char[] P) {
    int m = P.length;
    int[] prefixos = new int[m];
    prefixos[0] = -1;
    for (int i = 1, s = -1; i < m; i++) {
      while (s > -1 && P[s + 1] != P[i]) {
        s = prefixos[s];
      }
      if (P[s + 1]==P[i]) {
```

```
        s++;
      }
      prefixos[i] = s;
    }
    return prefixos;
}
```

Aqui ***m*** representa o tamanho do padrão, ***prefixos*** é um array que conterá cada posição que gera um prefixo para cada posição do padrão e ***s*** representa qual a posição do prefixo que foi calculado para a posição anterior do padrão.

No trecho 1 do código é definido que a primeira posição do padrão não possui nenhum prefixo significativo para o algoritmo, por isso é previamente estipulado que a posição que é prefixo dela é -1, isto é, fora do padrão.

No trecho 2 são feitas duas verificações, a primeira verifica se a posição anterior (representada por 's') formava um prefixo (`s > -1`), caso seja um prefixo ocorre a verificação se o caracter comparado na iteração atual continua sendo um prefixo (`P[s + 1] != P[i]`).

Caso o prefixo anterior seja um prefixo e a posição atual também seja, o código não entra no loop (`while`) e passa para o trecho 3 que incrementa em 1 a variável, indicando que a posição atual forma um prefixo.

Ainda no trecho 2 se a posição anterior for considerada prefixo mas a posição atual não, o código executa o *looping*. No *looping* como o prefixo não continua na posição atual, é testado se ao invés de usar o prefixo da posição anterior, usar o prefixo do prefixo da posição anterior (`s = prefixos[s];`).

O *looping* rodará até encontrar algum prefixo que a posição atual continue com algum prefixo anterior, caso não encontre nenhum o *looping* chegará a primeira posição do padrão que o prefixo vale -1.

Feita a verificação dos prefixos no trecho 4 é atribuida à posição respectiva a posição do prefixo.

3.4.2 Processamento de KMP

Agora analisando a busca de ocorrências com KMP:

```
List<Integer> busca() {
    List<Integer> resultado = new ArrayList<Integer>();
    int n = texto.length;
    int m = padrao.length;
    int s = -1;
    for (int i = 0; i <= n - 1; i++) {
      while (s > -1 && padrao[s + 1] != texto[i]) {
        s = prefixos[s];
      }
      if (padrao[s + 1]==texto[i]) {
        s++;
      }
      if (s == m - 1) {
        resultado.add(i-m+1);
        s = prefixos[s];
```

```
      }
    }
    return resultado;
}
```

O código é parecido com a função de pré-processamento, no trecho 2 é verificado se posição anterior era um prefixo e se a atual deixou de ser, caso a posição atual tenha deixado de ser, encontra um prefixo (baseado nos prefixos do prefixo anterior) que a posição atual continue com chance de ocorrência.

Caso nenhum prefixos dos prefixos anteiores dê a posição atual chance de ocorrência o *looping* chegará na primeira posição do array de prefixos que fixamente contém o valor -1, indicando que não há prefixos.

No trecho 3 caso o caracter corresponda a pesquisa no texto, ***s*** é incrementado em 1, indicando até que posição do padrão a busca está correspondida.

No trecho 4 é verificado se ***s*** está indicando que a posição que correspondeu ao texto é igual ao tamanho do padrão, neste caso é registrado uma ocorrência.

Caso seja ocorrência ainda é possível reaproveitar um possível prefixo da última posição do padrão (`s = prefixos[s];`).

3.5 Boyer-Moore

Os algoritmos anteriores usam o pré-processamento para evitar duplicação de comparações, a diferença entre um autômato e KMP é a otimização feita em cima do cálculo do pré-processamento.

Com o algoritmo de Boyer-Moore o objetivo do pré-processamento também é não comparar mais que uma vez o mesmo caracter do texto, porém o objetivo principal é não comparar cada um dos caracteres do texto melhorando o desenpenho da busca em relação à KMP.

Para poder pular as comparações de alguns caracteres, Boyer-Moore [6][3] mudou o modo de comparar a janela com o texto, os algoritmos anteriores comparam os caracteres da esquerda para direita, porém em Boyer-Moore a comparação é feita da direita para a esquerda. Através desta mudança e devido a duas possíveis heurísticas de deslocamento de janela, alguns caracteres não são comparados.

Esse pulo de comparação pode ocorrer caso a comparação de um caracter falhe e a janela sofra um deslocamento grande o suficiente para que não fique alinhado com alguns caracteres passados e não comparados.

Com isso o tempo de processamento de Boyer-Moore pode ser menor que O(n) e no pior caso O(n).

Exceto pela fase de pré-processamento e da comparação feita da direita para a esquerda o algoritmo de Boyer-Moore é muito parecido com o algoritmo de Força-Bruta, pois a comparação é feita caracter por caracter e caso ocorra uma falha na comparação a janela

é deslocada. O deslocamento em Boyer-Moore pode ser maior que uma posição, devido ao pré-processamento, diferente do Força-Bruta onde o deslocamento é sempre de uma posição.

3.5.1 As Heurísticas de Boyer-Moore

Para prôver um deslocamento maior que um, o algoritmo de Boyer-Moore propõe duas maneiras de calcular o deslocamento da janela, "Caracter Errado"e "Bom sufixo". O tamanho do deslocamento será igual ao maior deslocamento proposto pelas duas heurísticas.

```
max(caracterErrado(x), bomsufixo(x));
```

Caso os dois retornem um deslocamento 0 ou inferior a 0, o deslocamento será de uma posição.

3.5.2 A Heurística do Caracter Errado

A heurística do Caracter Errado apega-se no conceito de que quando a comparação de um caracter falhar (figura 26), só é possível haver uma ocorrência, alinhando este caracter errado com uma possível ocorrência dele no padrão, neste caso o deslocamento oferecido por esta heurística deve alinhar o caracter errado com sua ocorrência anterior no padrão.

Figura 26: Comparação falhando no caracter 'a' do texto.

A letra 'a' só ocorre na primeira posição do padrão, neste caso o deslocamento deve ser grande o suficiente para alinhar este 'a' ao 'a' do texto, conforme figura 27.

Figura 27: Deslocamento fazendo o padrão se deslocar até o caracter 'a'.

Uma vez que este caracter não se encontre no padrão, qualquer deslocamento que alinhe a janela com o caracter errado, com certeza, não será uma ocorrência, pois o caracter irá falhar novamente na comparação (figura 28), visto que, ele não existe no padrão, neste caso o deslocamento da janela deve ser grande o suficiente para que a janela fique imediatamente após o caracter errado.

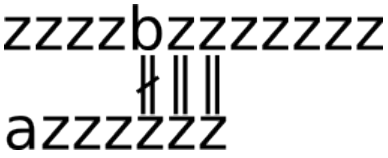

Figura 28: Comparação falhando no caracter 'b' do texto.

O deslocalmento deve ultrapassa o caracter errado 'b', pois ele não ocorre no padrão.

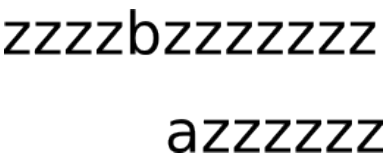

Figura 29: O deslocamento aqui deve ultrapassar o caracter 'b', visto que, ele não ocorre no padrão.

Para calcular os deslocamentos propostos por esta heurística, guardamos a posição que o padrão de se alinhar para cada carater do alfabeto, a princípio, todas as posições começam com 0. Após criar um vetor para cada posição do alfabeto, percorremos o padrão, guardando a posição de cada caracter. Percorremos o padrão da esquerda para a direita, assim garantimos que estamos guardando a posição mais à direita do caracter, não permitindo que nenhuma ocorrência seja ignorada.

```
m=P.length;
x=Σ.length;
posicoesDoAlfabeto[] = new array[x];

for(i = 1;i<=x;i++){
  posicoesDoAlfabeto=0;
}
for(j=1;j<=m;j++){
  posicoesDoAlfabeto[P[j]]=j;
}
```

3.5.3 A heurística do Bom sufixo

A heurística do Bom sufixo provê um deslocamento de forma a alinhar os caracteres já correspondidos na comparação da janela com uma outra possível ocorrência anterior desses caracteres dentro do padrão.

Ao encontrar um caracter errado na comparação da janela (i), caso um ou mais caracteres tenham sido correspondidos na pesquisa P[i+1..m], esses caracteres fazem parte do conjunto de sufixos do padrão. Se esse sufixo existir em outro ponto do padrão, fora o próprio sufixo, esta heurística sugere deslocar a janela de forma a alinhar esta cópia do sufixo com a posição onde já foi feita a comparação.

Existem três possíveis cenários em que esta heurística age:

- Quando existir uma ocorrência anterior do sufixo correspondido;

- Quando não existir uma ocorrência anterior do sufixo correspondido, porém um sufixo da correspondência também for prefixo;
- Quando não existir nem uma ocorrência anterior e nem um sufixo que seja prefixo.

No primeiro caso, onde o sufixo também existe antes do último caracter, o deslocamento é feito alinhando o sufixo já correspondido com sua próxima ocorrência. Exemplificando o caso, supondo que o caracter que não tenha sido correspondido na pesquisa tenha sido o 7, formando o sufixo "ab"na posição 8, conforme figura 30. Este sufixo ocorre dentro do padrão novamente na posição 5, com isso o deslocamento terá que alinhar esta nova ocorrência com a antiga posição do sufixo, deslocando o padrão em 3 posições, conforme figura 31.

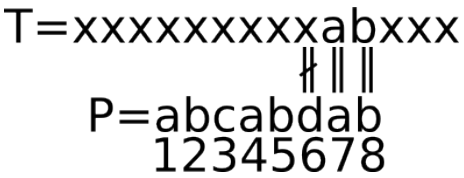

Figura 30: Posição 7 do padrão não corresponde ao texto, formando o sufixo 'ab'.

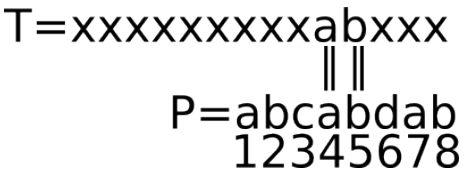

Figura 31: O deslocamento ocorre de maneira a alinhar o sufixo 'ab' com sua ocorrência anterior

No segundo caso o sufixo não ocorre por completo dentro do padrão, porém um sufixo deste sufixo também é o prefixo do texto, neste caso deslocamos o padrão para alinhar este prefixo à antiga posição do sufixo correspondido.

No exemplo abaixo o erro de comparação acontece na posição 5, formando o sufixo "dab", este sufixo não ocorre no padrão novamente, porém o sufixo "ab"também é prefixo do padrão, provendo um deslocamento, vide figuras 32 e 33.

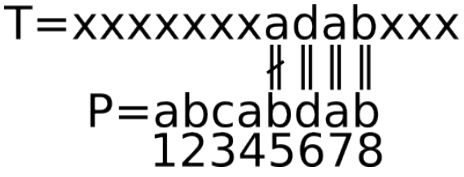

Figura 32: Embora o sufixo 'dab' não ocorra no padrão, há a ocorrência do prefixo/sufixo 'ab'

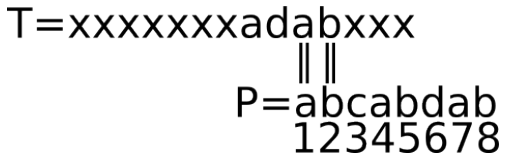

Figura 33: O deslocamento alinha o sufixo 'ab' com o prefixo 'ab' do padrão

No terceiro caso o sufixo formado (figura 34) não aparece no padrão novamente e nem um sufixo dele é prefixo do padrão, neste caso o deslocamento move a janela para depois deste sufixo, dado que ele não ocorre novamente a operação é segura, como na figura 35.

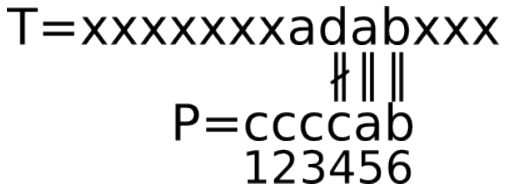

Figura 34: O sufixo 'ab' não ocorre novamente no padrão

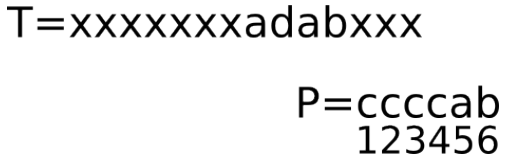

Figura 35: O deslocamento deve ser grande o suficiente para que o padrão fique depois das comparações anteriores

Para calcular os deslocamentos para cada posição do padrão, é necessário observar que os deslocamentos são os resultados da subtração do tamanho do padrão (m) com o a posição do último caracter da nova ocorrência do sufixo (L(i)).

L(i) sendo a posição da último caracter da repetição dos sufixos do padrão, para implementar L(i) é necessário buscar pela maior repetição de cada um dos sufixos do padrão, na verdade só precisamos do tamanho da cópia do sufixo, essa busca, a princípio, é quadrática ($O(n^2)$). Porém é possível transformá-la em linear ($O(n)$).

Para buscar todos os tamanhos das repetições dos sufixos de maneira linear, ao invés de buscar os sufixos, é mais simples inverter o padrão e fazer a busca pelos prefixos, usando alguma implementação já conhecida.

Uma possível implementação linear para a busca do tamanho dos prefixos é a seguinte: Primeiro realizamos a comparação caracter a caracter da posição 2 para conferir qual o maior substring que também seja prefixo da String, não precisamos comparar a posição 1 pois ela já é a String.

Ao analisar a próxima posição da String temos três caminhos a seguir:

- se a posição analisada não fizer parte de um prefixo anterior, fazer a comparação caracter a caracter novamente

- se que o caracter atual faz parte de uma janela, começando do caracter atual, esse sufixo também existe em um prefixo do padrão e esses caracteres, que são iguais, já foram analisados, com isso, se existir um prefixo correspondente, ele já é sabido. Porém existem dois cenário sobre este prefixo:
  - se ele terminar antes do fim da janela de prefixo atual, simplesmente guardamos este valor e continuamos a analise do próximo caracter
  - se ela terminar no mesmo ponto ou depois do fim da janela atual, devemos fazer uma comparação caracter a caracter dos caracter imediatamente seguintes da janela atual para descobrir o tamanho do prefixo

Uma análise superficial deste algoritmo leva a uma conclusão que o algoritmo é quadrático, porém ao analisar o algoritmo mais minunciosamente ele se mostra linear, pois cada caracter é comparado uma única vez. Na primeira vez analisamos caracter a caracter, se houver um erro e não formar um janela, quer dizer que nem o primeiro foi correspondido, apenas o primeiro caracter foi comparado e descartado de próximas comparações, ocorrendo o mesmo caso para os próximos caracteres. Caso o caracter analisado esteja dentro de uma janela, significa que eles já foram analisados, em um primeiro momento analisamos apenas se, sua correspondência anterior forma ou não uma janela, sem comparação de caracteres, caso forme uma janela que não ultrapasse o fim da janela atual, guardamos o tamanho da janela e passamos ao próximo caracter, caso forme uma janela maior que isso, já é sabido que os caracteres são iguais, só havendo necessidade de comparar aqueles que estão depois da janela atual.

Para este algoritmo ser linear são guardados os valores da posição mais a esquerda e mais a direira da janela atual, com isso é simples saber se a posição atual está ou não contida em uma janela.

Exemplo do algoritmo em Java:

```
//buscaTamanhoDaCopiaDosPrefixos
int[] Z(char[] S) {
  int n = S.length;
  int[] p = new int[n];
  int r = 0,l=0,i=0,q=0;
  for (int k = 1; k < n; k++) {
    if (k > r) {
      i = 0;
      q = k;
      while (q < n && S[i] == S[q]) {
        i++;
        q++;
      }
      p[k] = q - k;
      if (p[k] > 0) {
        r = k + p[k] - 1;
        l = k;
      } else {
        r = l = 0;
```

```
      }
    } else {
      int ka = k - l;
      if (p[ka] < r - (k - 1)) {
        p[k] = p[ka];
      } else {
        i = r - l;
        q = r;
        while (q < n && S[i] == S[q]) {
          i++;
          q++;
        }
        p[k] = q - k;
        r = q - 1;
        l = k;
      }
    }
  }
  return p;
}
```

Tendo o algoritmo para buscar o tamanho da cópia de cada prefixo, podemos achar a posição de cada cópia do sufixo em duas etapas.

Em um primeiro momento, busca-se o tamanho da cópia do sufixo de cada prefixo que tambem seja sufixo usando o algoritmos anterior aplicado ao padrão invertido. Podemos usar a seguinte rotina:

Exemplo em Java:

```
//buscaOTamanhoDosufixoDeCadaPrefixoQueTambemSejasufixo
int[] N(char[] P) {
  int n = P.length;
  int[] prefixos = Z(inverter(P));

  int[] sufixos = new int[n];
  for (int i = 0; i < n; i++) {
    sufixos[i] = prefixos[n - i - 1];
  }
  return sufixos;
}
```

Guardamos o tamanho da cópia de todos os sufixos encontrados no padrão, a próxima e final etapa guardará a posição da cópia de cada sufixo.

Percorrendo o padrão, perguntamos a cada prefixo qual o tamanho da cópia que seu sufixo também é sufixo do padrão, subtraindo este tamanho do tamanho do padrão (m-tamanhoDosufixo), descobrimos de qual sufixo esta posição é cópia:

```
int[] posicaoDaCopiaDosufixo(char[] P) {
  int n = P.length;
  int[] tamanhosDossufixos = N(P);
  int[] posicoes = new int[n];
  for (int i = 0; i < n; i++) {
    posicoes[i] = -1;
  }
  for (int j = 0; j < n - 1; j++) {
    int i = (n - 1) - tamanhosDossufixos[j];
```

```
    posicoes[i] = j;
  }
  return posicoes;
}
```

Nesta heurística ainda pode acontecer que o sufixo já correspondido não tenha nenhuma outra ocorrência dentro do padrão, neste caso temos que analisar se algum sufixo do próprio sufixo pode fornecer um deslocamento.

O deslocamento neste caso não pode ir simplesmente para uma ocorrência anterior deste sufixo menor, pois com certeza os caracteres anteriores não corresponderiam, pois este foi exatamente o teste do caso anterior, os caracteres não correspondendo, não pode haver uma ocorrência, por isso o alinhamento do sufixo é feito com o prefixo do padrão.

Esta parte do pré-processamento deve procurar todo sufixo que também seja prefixo. Assim como o algoritmo anterior percorremos o padrão perguntando para cada prefixo se seu sufixo é cópia de algum sufixo do padrão, porém neste caso só nos interessa se o prefixo inteiro for cópia do sufixo. Caso o prefixo atual não seja também um sufixo o maior sufixo é o anterior.

Exemplo do algoritmo em java

```
int[] maiorsufixoPrefixo(char[] P) {
  int n = P.length;
  int[] l = new int[n];
  int[] copiaDossufixos = N(P);
  l[n-1]=copiaDossufixos[0];
  for (int i = 1; i < n; i++) {
    if(copiaDossufixos[i]==i+1){
      l[n-1-i]=copiaDossufixos[i];
    }else{
      l[n-1-i]=l[n-i];
    }
  }
  return l;
}
```

3.5.4 Processamento em Boyer-Moore

Analisadas as duas heurísticas de pré-processamento de Boyer-Moore, este algoritmo percorre todas as janelas comparando o padrão da direita para a esquerda.

Caso a janela atual não seja uma ocorrência será aplicado um deslocamento de acordo com a heurística que oferecer uma maior deslocamento, a heurística do bom sufixo ainda tem dois cenários, o que o padrão possui uma cópia do sufixo e que o um prefixo do padrão seja um sufixo. Caso a janela atual seja uma ocorrência o deslocamento aplicado visa alinhar o sufixo que corresponda ao maior prefixo possível.

```
List<Integer> busca() {

  int n = padrao.length;
  int m = texto.length;
```

```
  int k = n - 1;
  while (k <= m - 1) {
    int i = n - 1;
    int h = k;
    while (i > -1 && padrao[i] == texto[h]) {
      i--;
      h--;
    }
    if (i != -1) {
      int bomsufixo = n - 1;
      if (copiaDosufixo[i] != -1) {
        bomsufixo -= copiaDosufixo[i];
      } else {
        bomsufixo -= prefixosufixo[i];
      }
      k += Math.max(1,
          Math.max(bomsufixo,
            i - caracterErrado[texto[h]]));
    } else {
      resultado.add(k - n + 1);
      k += n-prefixosufixo[0];

    }
  }
  return resultado;
}
```

4 Resultados Experimentais

Para este trabalho foram realizados diversos testes, cada um deles com entredas diferentes em relação ao alfabeto utilizado e tamanho do padrão. A máquina utilizada para os testes foi um notebook com processador Intel(R) Core(TM)2 Duo T5750 com 4 Gb de memória RAM rodando o sistema operacional Ubuntu 12.04 32 bits. A linguagem utilizada foi a Java, rodando em uma máquina virtual 'Java(TM) SE Runtime Environment (build 1.6.0_26-b03) Java HotSpot(TM) Server VM (build 20.1-b02, mixed mode)'.

Os resultados apresentados abaixo são correspondentes à pesquisa da palavra 'Capitu' pelo livro 'Dom Casmurro' de Machado de Assis. A cópia utilizada possuí 371.457 caracteres enquanto o padrão possui 6 caracteres, há 335 ocorrências do padrão no texto.

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 6,351,530 ns
- Tempo médio total: 6,351,530 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 745,399
- Número de comparações total: 745,399

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 6,767,285 ns
- Tempo médio total: 6,767,285 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 374,405
- Número de comparações total: 374,405

Automato

- Tempo médio de pré-processamento: 66,980 ns
- Tempo médio de busca: 7,374,209 ns
- Tempo médio total: 7,441,189 ns
- Número de comparações no pré-processamento: 5,371

- Número de comparações na busca: 371,457
- Número de comparações total: 376,828

KMP

- Tempo médio de pré-processamento: 2,910 ns
- Tempo médio de busca: 5,855,042 ns
- Tempo médio total: 5,857,952 ns
- Número de comparações no pré-processamento: 5
- Número de comparações na busca: 745,409
- Número de comparações total: 745,414

Boyer Moore

- Tempo médio de pré-processamento: 10,840 ns
- Tempo médio de busca: 3,164,817 ns
- Tempo médio total: 3,175,657 ns
- Número de comparações no pré-processamento: 10
- Número de comparações na busca: 142,929
- Número de comparações total: 142,939

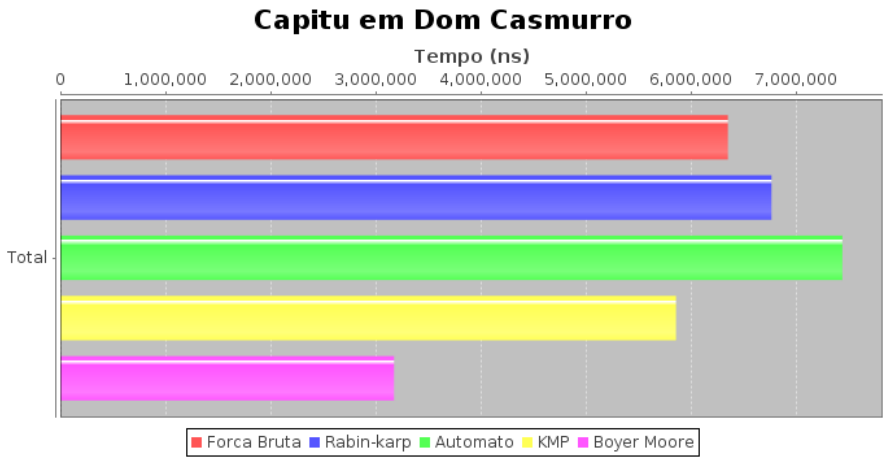

Figura 36: Tempo gasto na busca pela palavra 'Capitu' no livro Dom Casmurro

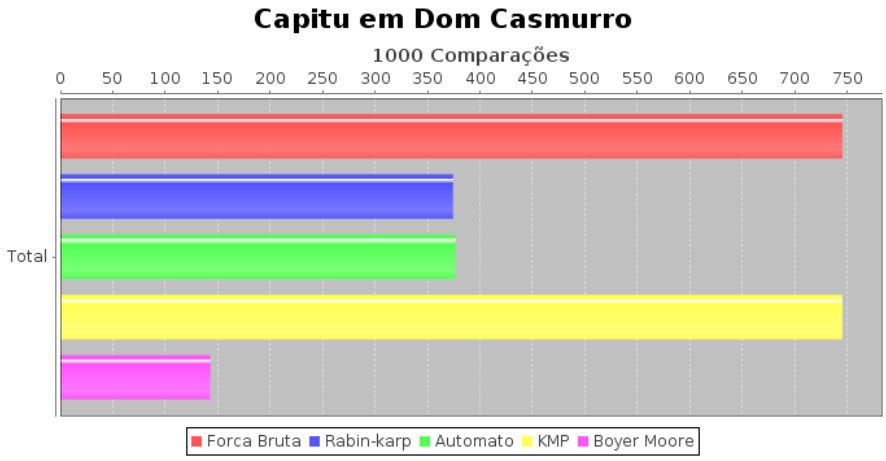

Figura 37: Comparações na busca pela palavra 'Capitu' no livro Dom Casmurro

Procurando no mesmo texto por um parágrafo de 523 caracteres, obtemos o resultado:
Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 6,222,294 ns
- Tempo médio total: 6,222,294 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 743,511
- Número de comparações total: 743,511

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 6,790,245 ns
- Tempo médio total: 6,790,245 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 371,825
- Número de comparações total: 371,825

Automato

- Tempo médio de pré-processamento: 75,598,235 ns
- Tempo médio de busca: 3,780,881 ns

- Tempo médio total: 79,379,116 ns
- Número de comparações no pré-processamento: 27,786,366
- Número de comparações na busca: 371,457
- Número de comparações total: 28,157,823

KMP

- Tempo médio de pré-processamento: 14,889 ns
- Tempo médio de busca: 5,699,510 ns
- Tempo médio total: 5,714,399 ns
- Número de comparações no pré-processamento: 522
- Número de comparações na busca: 744,556
- Número de comparações total: 745,078

Boyer Moore

- Tempo médio de pré-processamento: 56,447 ns
- Tempo médio de busca: 520,381 ns
- Tempo médio total: 576,828 ns
- Número de comparações no pré-processamento: 1,050
- Número de comparações na busca: 17,418
- Número de comparações total: 18,468

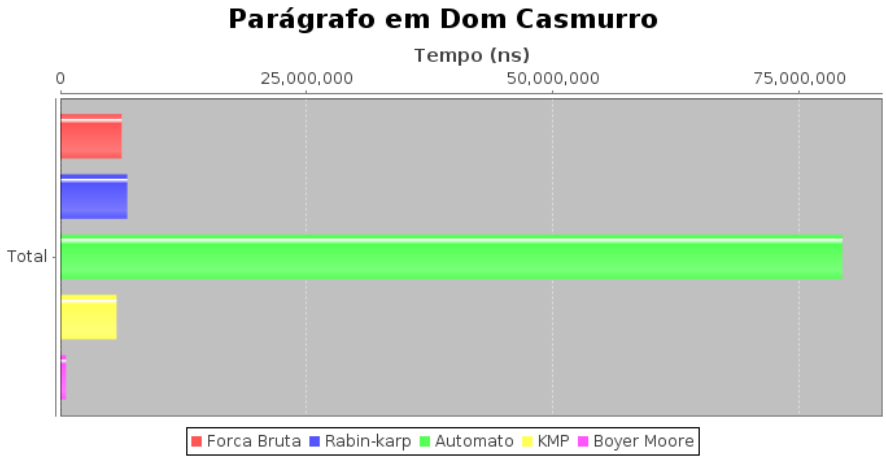

Figura 38: Tempo gasto na busca por um parágrafo inteiro no livro Dom Casmurro

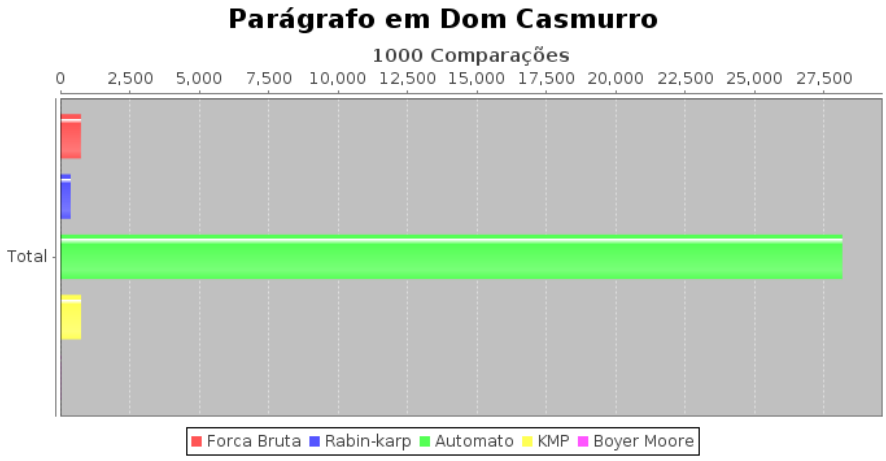

Figura 39: Comparações na busca por um parágrafo inteiro no livro Dom Casmurro

Outro teste realizado foi com um texto técnico em inglês sobre o sistema operacional Linux com 5.391.456 de caracteres, neste texto o termo procurado foi 'Linux Dictionary' de 16 caracteres, neste texto há 1735 ocorrências do padrão.

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 53,232,686 ns
- Tempo médio total: 53,232,686 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 10,964,424
- Número de comparações total: 10,964,424

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 55,831,193 ns
- Tempo médio total: 55,831,193 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 5,427,361
- Número de comparações total: 5,427,361

Automato

- Tempo médio de pré-processamento: 1,158,697 ns

- Tempo médio de busca: 59,830,208 ns
- Tempo médio total: 60,988,905 ns
- Número de comparações no pré-processamento: 29,439
- Número de comparações na busca: 5,391,456
- Número de comparações total: 5,420,895

KMP

- Tempo médio de pré-processamento: 3,070 ns
- Tempo médio de busca: 49,831,688 ns
- Tempo médio total: 49,834,758 ns
- Número de comparações no pré-processamento: 15
- Número de comparações na busca: 10,964,454
- Número de comparações total: 10,964,469

Boyer Moore

- Tempo médio de pré-processamento: 80,200 ns
- Tempo médio de busca: 17,437,740 ns
- Tempo médio total: 17,517,940 ns
- Número de comparações no pré-processamento: 30
- Número de comparações na busca: 1,205,696
- Número de comparações total: 1,205,726

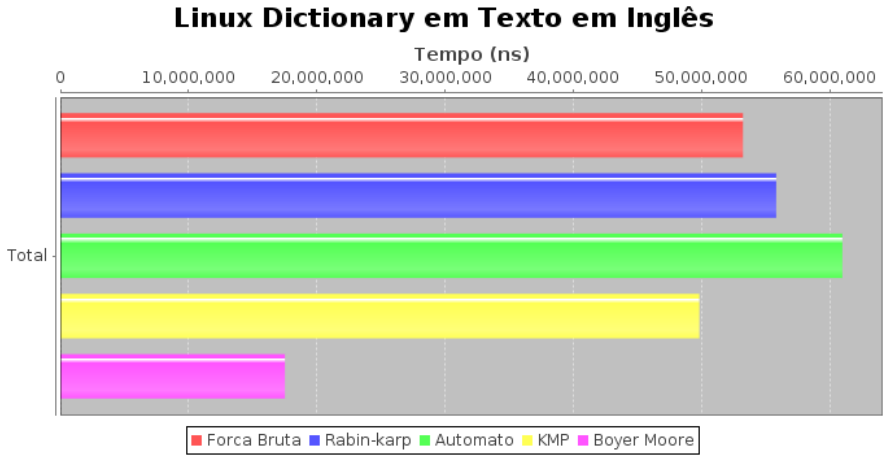

Figura 40: Tempo gasto na busca pelas palavras 'Linux Dictionary' em um dicionário técnico

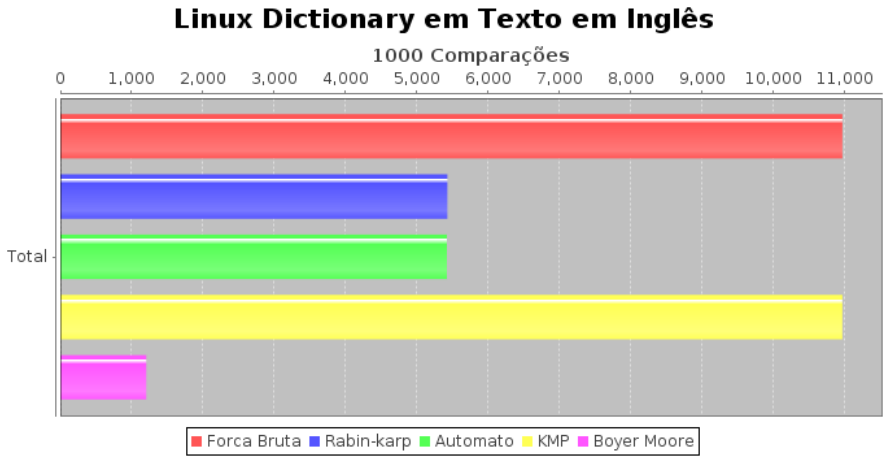

Figura 41: Comparações na busca pelas palavras 'Linux Dictionary' em um dicionário técnico

O próximo teste foi feito com um texto de 1000 caracteres aleatórios entre 'a' e 'b' e um padrão de 10 caracteres também aleatórios entre 'a' e 'b':

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 42,083 ns
- Tempo médio total: 42,083 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 2,952
- Número de comparações total: 2,952

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 22,299 ns
- Tempo médio total: 22,299 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 993
- Número de comparações total: 993

Automato

- Tempo médio de pré-processamento: 33,610 ns

- Tempo médio de busca: 22,617 ns
- Tempo médio total: 56,227 ns
- Número de comparações no pré-processamento: 180
- Número de comparações na busca: 1,000
- Número de comparações total: 1,180

KMP

- Tempo médio de pré-processamento: 3,354 ns
- Tempo médio de busca: 41,589 ns
- Tempo médio total: 44,943 ns
- Número de comparações no pré-processamento: 16
- Número de comparações na busca: 2,805
- Número de comparações total: 2,821

Boyer Moore

- Tempo médio de pré-processamento: 9,521 ns
- Tempo médio de busca: 21,287 ns
- Tempo médio total: 30,808 ns
- Número de comparações no pré-processamento: 24
- Número de comparações na busca: 861
- Número de comparações total: 885

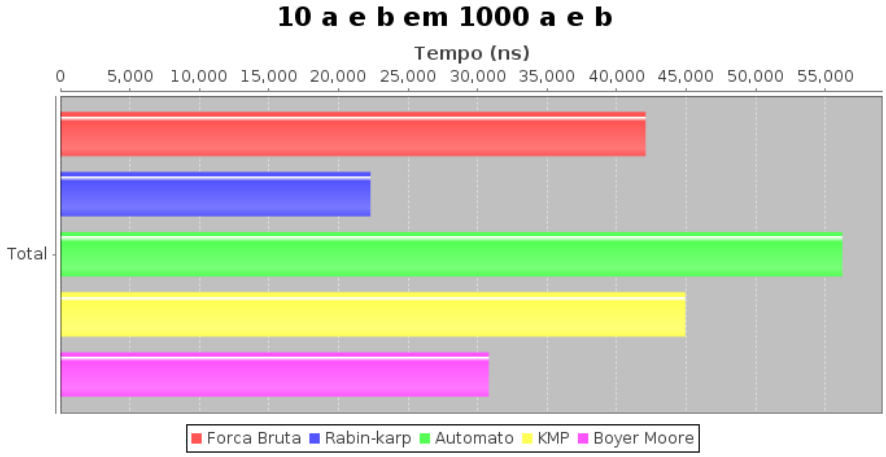

Figura 42: Tempo gasto na busca por um padrão de tamanho 10 com as letras 'a' e 'b' em um texto de 1000 caracteres também formado por 'a' e 'b'

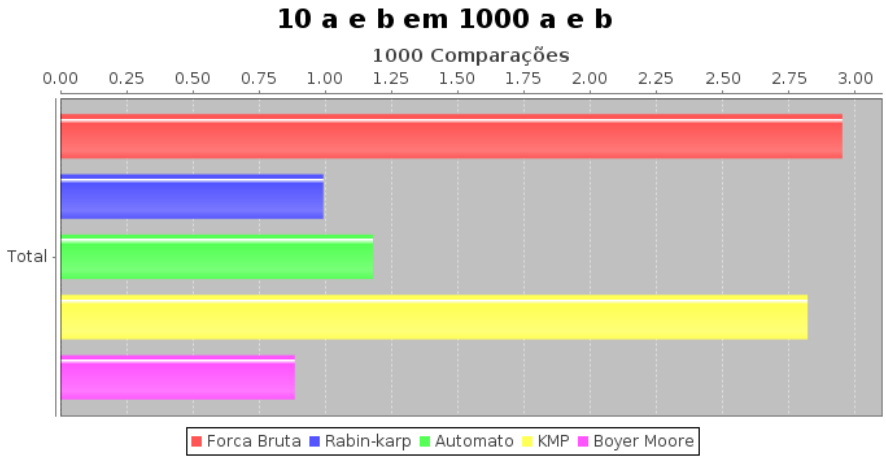

Figura 43: Comparações na busca por um padrão de tamanho 10 com as letras 'a' e 'b' em um texto de 1000 caracteres também formado por 'a' e 'b'

O próximo teste foi feito com um texto de 10.000.000 caracteres aleatórios entre 'a' e 'b' e um padrão de 10 caracteres também aleatórios entre 'a' e 'b':

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 198,756,298 ns
- Tempo médio total: 198,756,298 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 29,981,581
- Número de comparações total: 29,981,581

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 101,070,129 ns
- Tempo médio total: 101,070,129 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 10,575,903
- Número de comparações total: 10,575,903

Automato

- Tempo médio de pré-processamento: 31,169 ns

- Tempo médio de busca: 108,288,819 ns
- Tempo médio total: 108,319,988 ns
- Número de comparações no pré-processamento: 183
- Número de comparações na busca: 10,000,000
- Número de comparações total: 10,000,183

KMP

- Tempo médio de pré-processamento: 3,468 ns
- Tempo médio de busca: 197,849,575 ns
- Tempo médio total: 197,853,043 ns
- Número de comparações no pré-processamento: 13
- Número de comparações na busca: 28,184,986
- Número de comparações total: 28,184,999

Boyer Moore

- Tempo médio de pré-processamento: 6,617 ns
- Tempo médio de busca: 82,360,961 ns
- Tempo médio total: 82,367,578 ns
- Número de comparações no pré-processamento: 26
- Número de comparações na busca: 6,927,223
- Número de comparações total: 6,927,249

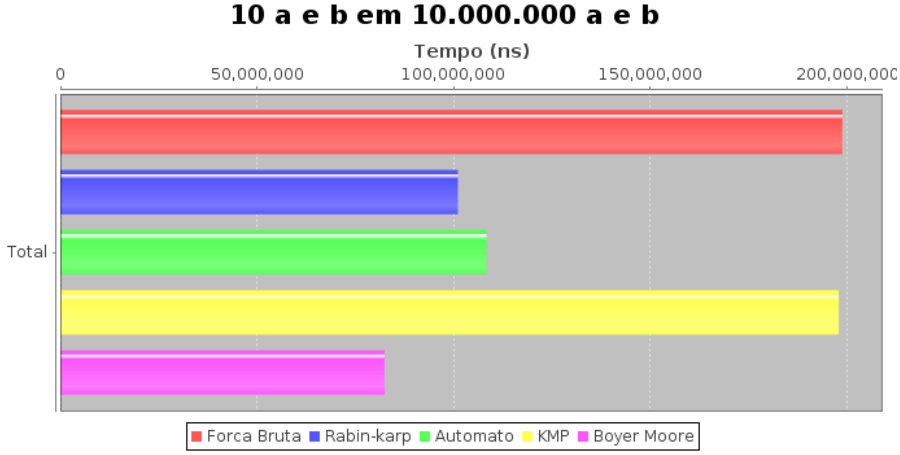

Figura 44: Tempo gasto na busca por um padrão de tamanho 10 com as letras 'a' e 'b' em um texto de 10.000.000 caracteres também formado por 'a' e 'b'

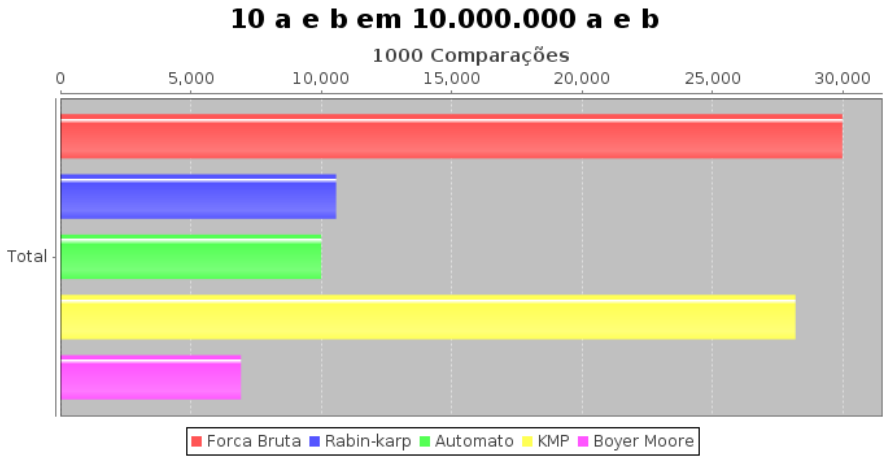

Figura 45: Comparações na busca por um padrão de tamanho 10 com as letras 'a' e 'b' em um texto de 10.000.000 caracteres também formado por 'a' e 'b'

Os próximos resultados são relativos à uma cadeia de DNA aleatória de 10 caracteres e um padrão de 2 caracteres.

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 3,537 ns
- Tempo médio total: 3,537 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 20
- Número de comparações total: 20

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 3,030 ns
- Tempo médio total: 3,030 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 9
- Número de comparações total: 9

Automato

- Tempo médio de pré-processamento: 15,215 ns

- Tempo médio de busca: 2,247 ns
- Tempo médio total: 17,462 ns
- Número de comparações no pré-processamento: 33
- Número de comparações na busca: 10
- Número de comparações total: 43

KMP

- Tempo médio de pré-processamento: 2,732 ns
- Tempo médio de busca: 3,567 ns
- Tempo médio total: 6,299 ns
- Número de comparações no pré-processamento: 1
- Número de comparações na busca: 22
- Número de comparações total: 23

Boyer Moore

- Tempo médio de pré-processamento: 7,108 ns
- Tempo médio de busca: 2,991 ns
- Tempo médio total: 10,099 ns
- Número de comparações no pré-processamento: 2
- Número de comparações na busca: 17
- Número de comparações total: 19

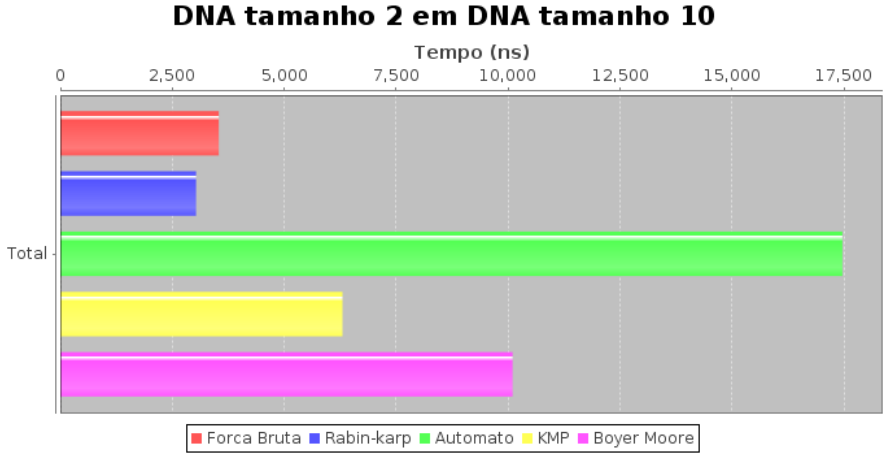

Figura 46: Tempo gasto na busca de uma cadeia de DNA de tamanho 2 em uma cadeia de tamanho 10

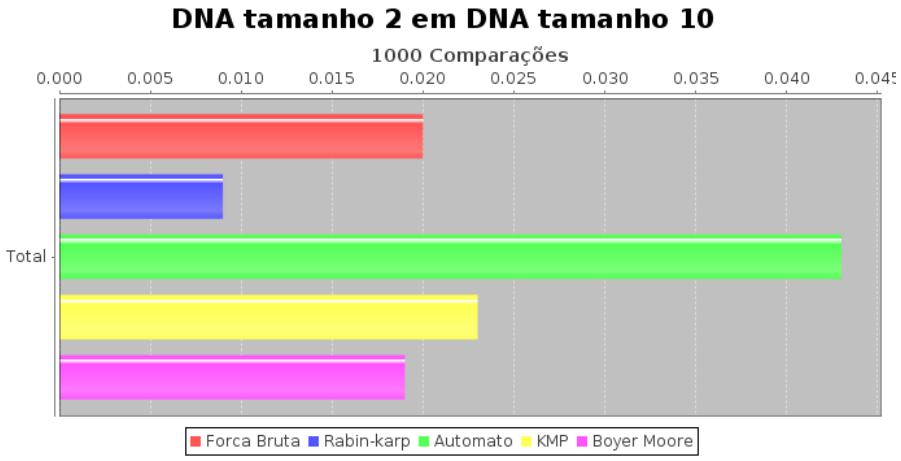

Figura 47: Comparações na busca de uma cadeia de DNA de tamanho 2 em uma cadeia de tamanho 10

Os próximos resultados são relativos à uma cadeia de DNA aleatória de 3.000 caracteres e um padrão de 10 caracteres.

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 89,330 ns
- Tempo médio total: 89,330 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 7,130
- Número de comparações total: 7,130

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 71,833 ns
- Tempo médio total: 71,833 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 3,303
- Número de comparações total: 3,303

Automato

- Tempo médio de pré-processamento: 32,121 ns

- Tempo médio de busca: 67,784 ns
- Tempo médio total: 99,905 ns
- Número de comparações no pré-processamento: 417
- Número de comparações na busca: 3,000
- Número de comparações total: 3,417

KMP

- Tempo médio de pré-processamento: 2,939 ns
- Tempo médio de busca: 110,932 ns
- Tempo médio total: 113,871 ns
- Número de comparações no pré-processamento: 9
- Número de comparações na busca: 7,150
- Número de comparações total: 7,159

Boyer Moore

- Tempo médio de pré-processamento: 8,497 ns
- Tempo médio de busca: 41,479 ns
- Tempo médio total: 49,976 ns
- Número de comparações no pré-processamento: 19
- Número de comparações na busca: 1,522
- Número de comparações total: 1,541

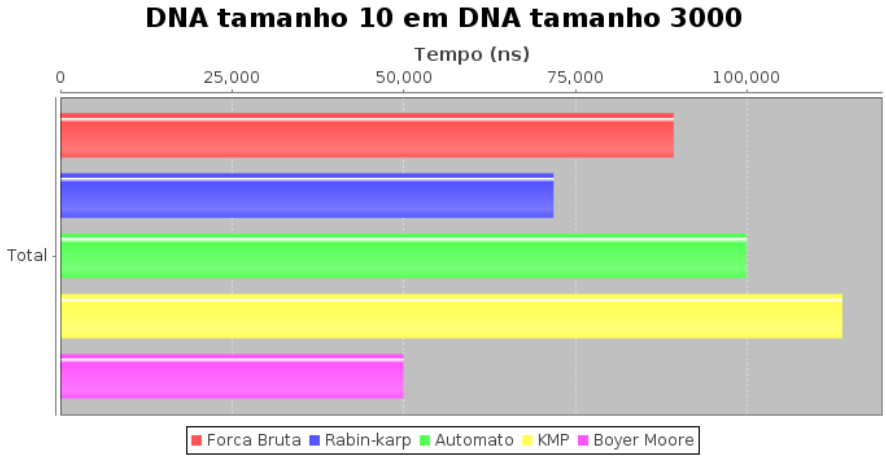

Figura 48: Tempo gasto na busca de uma cadeia de DNA de tamanho 10 em uma cadeia de tamanho 3000

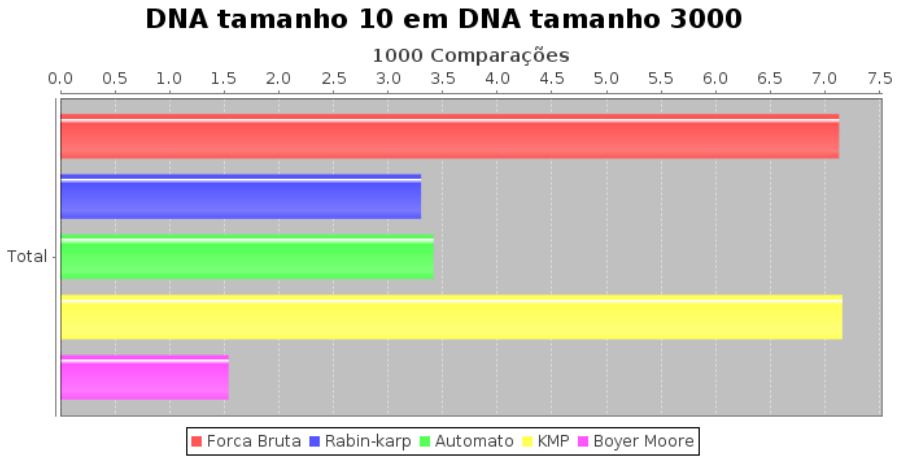

Figura 49: Comparações na busca de uma cadeia de DNA de tamanho 10 em uma cadeia de tamanho 3000

Estes resultados são de uma cadeia de DNA aproximadamente 200.000.000 de caracteres e com um padrão com 10.000 caracteres:

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 3,053,265,737 ns
- Tempo médio total: 3,053,265,737 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 664,329,790
- Número de comparações total: 664,329,790

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 2,352,158,618 ns
- Tempo médio total: 2,352,158,618 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 398,282,433
- Número de comparações total: 398,282,433

Automato

- Tempo médio de pré-processamento: 1,630,449,405 ns

- Tempo médio de busca: 14,161,556,848 ns
- Tempo médio total: 15,792,006,253 ns
- Número de comparações no pré-processamento: 350,024,043
- Número de comparações na busca: 200,008,843
- Número de comparações total: 550,032,886

KMP

- Tempo médio de pré-processamento: 194,845 ns
- Tempo médio de busca: 2,353,942,231 ns
- Tempo médio total: 2,354,137,076 ns
- Número de comparações no pré-processamento: 13,315
- Número de comparações na busca: 598,686,356
- Número de comparações total: 598,699,671

Boyer Moore

- Tempo médio de pré-processamento: 533,049 ns
- Tempo médio de busca: 241,581,574 ns
- Tempo médio total: 242,114,623 ns
- Número de comparações no pré-processamento: 22,554
- Número de comparações na busca: 198,593,846
- Número de comparações total: 198,616,400

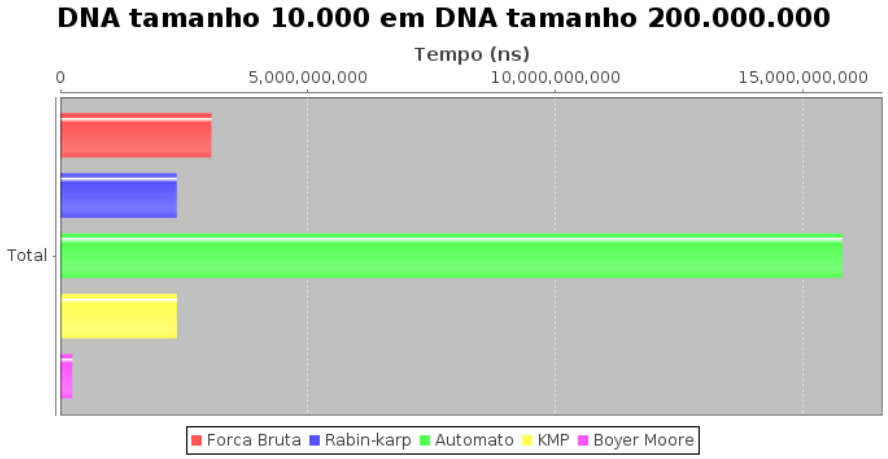

Figura 50: Tempo gasto na busca de uma cadeia de DNA de tamanho 10.000 em uma cadeia de tamanho 200.000.000

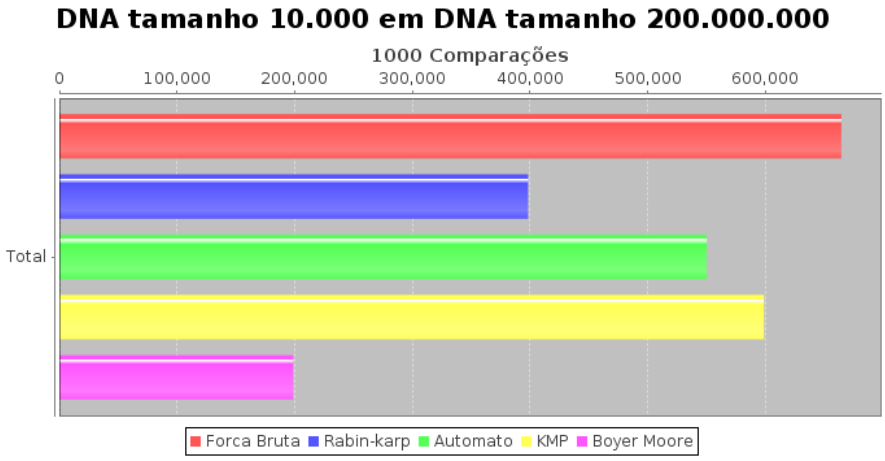

Figura 51: Comparações na busca de uma cadeia de DNA de tamanho 10.000 em uma cadeia de tamanho 200.000.000

A última sequencia de teste apresenta um conjunto de 2.000.000 de caracteres aleatórios em um alfabeto de 65000 caracteres com um padrão de busca de 100 caracteres:

Forca Bruta

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 25,364,582 ns
- Tempo médio total: 25,364,582 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 4,986,540
- Número de comparações total: 4,986,540

Rabin-karp

- Tempo médio de pré-processamento: 0 ns
- Tempo médio de busca: 26,098,131 ns
- Tempo médio total: 26,098,131 ns
- Número de comparações no pré-processamento: 0
- Número de comparações na busca: 3,008,515
- Número de comparações total: 3,008,515

Automato

- Tempo médio de pré-processamento: 1,840,429,252 ns

- Tempo médio de busca: 31,995,997 ns
- Tempo médio total: 1,872,425,249 ns
- Número de comparações no pré-processamento: 669,425,001
- Número de comparações na busca: 2,000,093
- Número de comparações total: 671,425,094

KMP

- Tempo médio de pré-processamento: 3,566 ns
- Tempo médio de busca: 25,245,004 ns
- Tempo médio total: 25,248,570 ns
- Número de comparações no pré-processamento: 99
- Número de comparações na busca: 4,986,738
- Número de comparações total: 4,986,837

Boyer Moore

- Tempo médio de pré-processamento: 169,591 ns
- Tempo médio de busca: 5,868,028 ns
- Tempo médio total: 6,037,619 ns
- Número de comparações no pré-processamento: 198
- Número de comparações na busca: 1,028,051
- Número de comparações total: 1,028,249

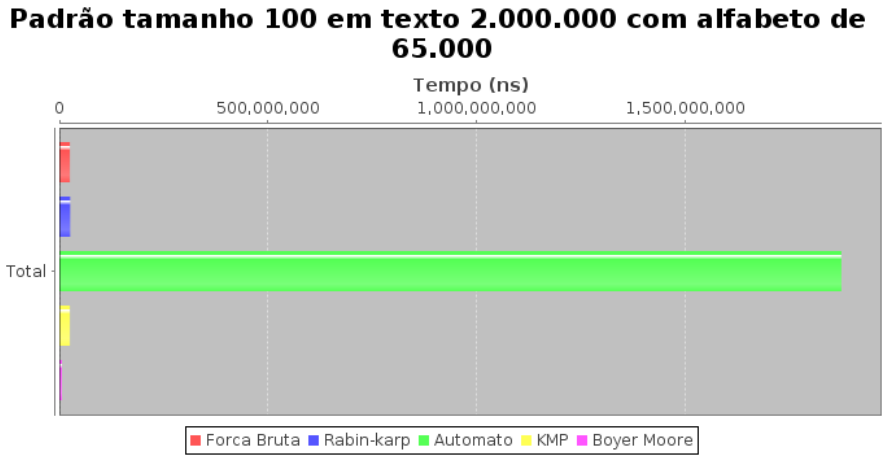

Figura 52: Tempo gasto na busca de um padrão com 100 caracater em um texto de 2.000.000 de caracteres, ambos formados por um alfabeto de 65.000 possíveis caracteres

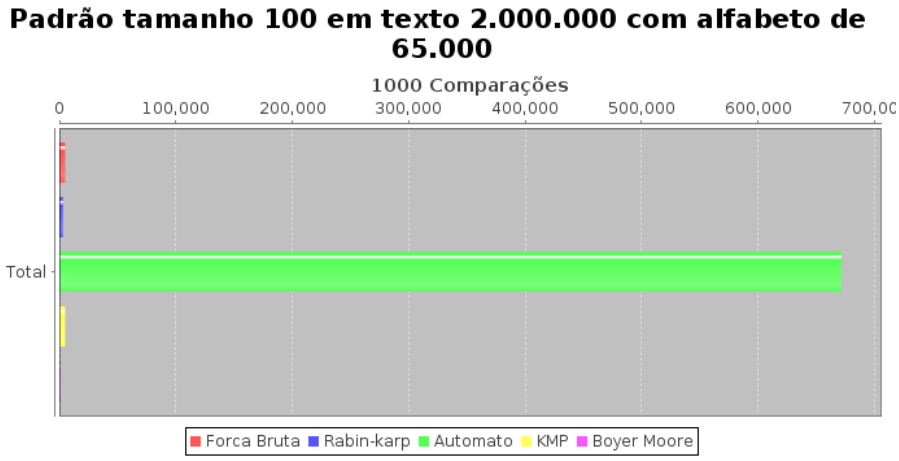

Figura 53: Comparações na busca de um padrão com 100 caracater em um texto de 2.000.000 de caracteres, ambos formados por um alfabeto de 65.000 possíveis caracteres

5 Conclusão

Analisando os resultados obtidos, a maioria dos cenários apontam o algoritmo de Boyer-Moore como a melhor escolha ou próxima da melhor, os outros algoritmos precisam de cenários específicos para valerem a pena.

O algoritmo de Força-Bruta se mostrou muito eficaz em cenários onde o texto e o padrão são muito pequenos (até 10 caracteres), ou um texto de tamanho um pouco maior (até 1000 caracteres) e padrão pequeno, porém com um alfabeto muito pequeno (até 2 caracteres). Dado que ele não faz o pré-processamento, não há perda de processamento analisando a melhor forma de percorrer o padrão, com um padrão muito pequeno, ganha-se pouco no pré-processamento. Inclusive este é o algoritmo utilizado para implementar o método *contains* [8] da classe *String* na linguagem Java.

O algoritmo de Rabin-Karp lançou quase sempre resultados um pouco melhores que o de Força-Bruta, porém próximos, pois os algoritmos são muito próximos e usam a mesma base de deslocamento.

Utilizando Autômato é notável o tempo de processamento gasto no pré-processamento ($O(\Sigma m^3)$) quando o padrão é muito grande (no cenário apresentado por exemplo com 10.000 caracteres). Isso inviabilizaria o uso em muitos cenários, porém com um padrão e alfabetos pequenos, pesquisa binária ou até uma cadeia de DNA) o Autômato apresenta resultados excelentes, chegando a ser o mais rápido na pesquisa de texto tamanho 10.000.00 e padrão de tamanho 10.

O algoritmo de Knuth-Morris-Pratt demonstra um meio termo entre todos os algoritmos, em poucos cenários ele se aproxima do tempo de processamento de Boyer-Moore, porém quando Boyer-Moore não demonstra bons resultados (texto e padrao muito pequenos) KMP consegue ficar muito próximo dos mais rápidos. O grande problema de KMP foi no cenário de texto muito grande (10.000.000 caracteres) e padrão e alfabeto muito pequenos (10 e 2 respectivamente), pois ele é pareceido com o Autômato, porém sem analisar o alfabeto para as próximas posições ele ganha muito no pré-processamento (com padrão pequeno isso é irrelevante) e perde no cenário onde existem muitas repetições de sufixo e prefixo, se movendo pouco no texto.

Já Boyer-Moore se mostrou o mais eficaz na maioria dos cenários, pois com suas duas heurísticas, existem poucos cenários onde o deslocamento de janela é pequeno. Quando o alfabeto é pequeno a heurística de "Caracter Errado"é pouco eficaz, porém a de bom sufixo acaba suprindo a necessidade de um bom deslocamento, já quando o padrão é pequeno, as duas heurísticas acabam não ajudando muito à janela avançar. Fora no cenário de um padrão pequeno com alfabeto pequeno este algoritmo demonstrou ter o melhor desempenho. Boyer-Moore é uma das principais escolhas de algoritmo de busca dos sistemas atuais, incluindo o comando *grep*[7] do sistema operacional Unix e o editor de texto *Hexplorer*.

6 Anexo I - Implementações dos algoritmos

- Implementação java do algortimos de Força Bruta:

```
package com.guilhermemoreira.algoritmos;
import java.util.ArrayList;
import java.util.List;

public class ForcaBruta implements Algoritmo {

  private char[] texto;
  private char[] padrao;
  private DadosIndividuais dados;

  public ForcaBruta() {
    this.dados = new DadosIndividuais("Forca Bruta");
  }

  public ForcaBruta texto(String texto) {
    this.texto = texto.toCharArray();
    return this;
  }

  public ForcaBruta padrao(String padrao) {
    this.padrao = padrao.toCharArray();
    return this;
  }

  public DadosIndividuais executa() {
    this.dados.comecaBusca();
    this.dados.ocorrencias(busca());
    this.dados.terminaBusca();
    return dados;
  }

  private List<Integer> busca() {
    List<Integer> integers = new ArrayList<Integer>();
    int n = texto.length;
    int m = padrao.length;
    for (int i = 0; i < n - m + 1; i++) {
      int j = 0;
      while (j < m && texto[i + j] == padrao[j]) {
        j++;
      }
      if (j == m) {
        integers.add(i);
      }
    }
    return integers;

  }
}
```

- Inplementação java do algoritmo de Rabin-Karp:

```
package com.guilhermemoreira.algoritmos;
import java.util.ArrayList;
import java.util.List;
```

```
public class RabinKarp implements Algoritmo {

  private char[] texto;
  private char[] padrao;
  private ArrayList<Integer> resultados;
  private DadosIndividuais dados;
  private final char[] alfabeto;

  public RabinKarp(char[] alfabeto) {
    this.alfabeto = alfabeto;
    this.resultados = new ArrayList<Integer>();
    this.dados = new DadosIndividuais("Rabin-karp");
  }

  private List<Integer> busca() {
    int n = texto.length;
    int m = padrao.length;
    int h = alfabeto.length;
    int primo = 31;
    int base = 1;
    for (int i = 0; i < m-1; i++) {
      base*=h;
    }

    int hashTexto = 0;
    int hashPadrao = 0;

    for (int i = 0; i < m && i < n; i++) {
      hashTexto=hashTexto*h+texto[i];
      hashPadrao=hashPadrao*h+padrao[i];
    }

    hashPadrao%=primo;

    if(hashTexto%primo==hashPadrao){
      int k=0;
      while(k<m && padrao[k]==texto[k]){
        k++;
      }
      if(k==m){
        resultados.add(0);
      }
    }

    for (int i = m; i < n; i++) {
      hashTexto=hashTexto-texto[i-m]*base;
      hashTexto*=h;
      hashTexto+=texto[i];
      if(hashTexto%primo==hashPadrao){
        int k=0;
        while(k<m && padrao[k]==texto[i-m+1+k]){
          k++;
        }
        if(k==m){
          resultados.add(i-m+1);
        }
      }
```

```
    }

    return resultados;
  }

  public RabinKarp texto(String texto) {
    this.texto = texto.toCharArray();
    return this;
  }

  public RabinKarp padrao(String padrao) {
    this.padrao = padrao.toCharArray();
    return this;
  }

  public DadosIndividuais executa() {
    this.dados.comecaBusca();
    this.dados.ocorrencias(busca());
    this.dados.terminaBusca();
    return dados;
  }
}
```

- Implementação java da busca por Autômato:

```
package com.guilhermemoreira.algoritmos;
import java.util.ArrayList;
import java.util.List;

public class Automato implements Algoritmo {

  private char[] texto;
  private char[] padrao;
  private char[] alfabeto;
  private ArrayList<Integer> resultado;
  private int[][] automato;
  private DadosIndividuais dados;

  public Automato(char[] alfabeto) {
    this.alfabeto = alfabeto;
    this.resultado = new ArrayList<Integer>();
    this.dados = new DadosIndividuais("Automato");
  }

  private List<Integer> busca() {
    int estado = 0;
    int n = texto.length;
    int m = padrao.length;
    for (int i = 0; i < n; i++) {
      estado = automato[estado][texto[i]];
      if (estado == m) {
        resultado.add(i-m+1);
      }
    }
    return resultado;
  }

  private int[][] constroiAutomato(char[] P, char[] alfabeto, int maiorIndice
```

```
    int[][] automato = new int[P.length + 1][maiorIndice + 1];

    for (int estado = 0; estado <= P.length; estado++) {
      for (int sigma = 0; sigma < alfabeto.length; sigma++) {
        if (estado < P.length && alfabeto[sigma] == P[estado]) {
          automato[estado][alfabeto[sigma]] = estado + 1;
        } else {
          int deslocamento = 1;
          while (deslocamento <= estado) {
            int i = 0;
            for (; i <= estado - deslocamento
                && i + deslocamento < P.length; i++) {
              if (P[i] != P[i + deslocamento]) {
                break;
              }
            }
            if (i == estado - deslocamento
                && P[i] == alfabeto[sigma]) {
              automato[estado][alfabeto[sigma]] = i + 1;
              deslocamento = estado + 1;
            }
            deslocamento++;
          }
        }
      }
    }
    return automato;
  }

  public DadosIndividuais executa() {
    dados.comecaPreProcessamento();
    preProcessamento();
    dados.terminaPreProcessamento();
    dados.comecaBusca();
    dados.ocorrencias(busca());
    dados.terminaBusca();
    return dados;
  }

  private void preProcessamento() {
    this.automato = constroiAutomato(padrao, alfabeto,
        alfabeto[alfabeto.length - 1]);
  }

  public Automato texto(String texto) {
    this.texto = texto.toCharArray();
    return this;
  }

  public Automato padrao(String padrao) {
    this.padrao = padrao.toCharArray();
    return this;
  }

}
```

- Implementação java de Knuth-Morris-Pratt:

```
package com.guilhermemoreira.algoritmos;
import java.util.ArrayList;
import java.util.List;

public class KnuthMorrisPratt implements Algoritmo {

  private char[] texto;
  private char[] padrao;
  private int[] prefixos;
  private DadosIndividuais dados;

  public KnuthMorrisPratt() {
    this.dados = new DadosIndividuais("KMP");
  }

  private List<Integer> busca() {
      List<Integer> resultado = new ArrayList<Integer>();
      int n = texto.length;
      int m = padrao.length;
      int s = -1;
      for (int i = 0; i <= n - 1; i++) {
        while (s > -1 && padrao[s + 1] != texto[i]) {
          s = prefixos[s];
        }
        if (padrao[s + 1]==texto[i]) {
          s++;
        }
        if (s == m - 1) {
          resultado.add(i-m+1);
          s = prefixos[s];
        }
      }
      return resultado;
  }

  public int[] calculaPrefixos(char[] P) {
      int m = P.length;
      int[] prefixos = new int[m];
      prefixos[0] = -1;
      for (int i = 1, s = -1; i < m; i++) {
        while (s > -1 && P[s + 1] != P[i]) {
          s = prefixos[s];
        }
        if (P[s + 1]==P[i]) {
          s++;
        }
        prefixos[i] = s;
      }
      return prefixos;
  }

  public KnuthMorrisPratt texto(String texto) {
    this.texto = texto.toCharArray();
    return this;
  }

  public KnuthMorrisPratt padrao(String padrao) {
    this.padrao = padrao.toCharArray();
```

```
    return this;
  }

  public DadosIndividuais executa() {
    this.dados.comecaPreProcessamento();
    this.preProcessamento();
    this.dados.terminaPreProcessamento();
    this.dados.comecaBusca();
    this.dados.ocorrencias(this.busca());
    this.dados.terminaBusca();
    return dados;
  }

  private void preProcessamento() {
    this.prefixos = calculaPrefixos(padrao);
  }

}
```

- Implementação java de Boyer-Moore:

```
package com.guilhermemoreira.algoritmos;
import java.util.ArrayList;
import java.util.List;

import com.guilhermemoreira.execucao.Util;

public class BoyerMoore implements Algoritmo {

  private char[] texto;
  private char[] padrao;
  private ArrayList<Integer> resultado;
  private int[] caracterErrado;
  private int[] copiaDoSulfixo;
  private int[] prefixoSulfixo;
  private DadosIndividuais dados;
  private char[] alfabeto;

  public BoyerMoore(char[] alfabeto) {
    this.alfabeto = alfabeto;
    this.resultado = new ArrayList<Integer>();
    this.dados = new DadosIndividuais("Boyer Moore");
  }

  private int[] calculaCaracterErrado(char[] P) {
    int[] posicoesDoAlfabeto = new int[alfabeto[alfabeto.length-1]+1];
    for (int i = 0; i < posicoesDoAlfabeto.length; i++) {
      posicoesDoAlfabeto[i] = -1;
    }
    for (int i = 0; i < P.length; i++) {
      posicoesDoAlfabeto[P[i]] = i;
    }
    return posicoesDoAlfabeto;
  }

  private List<Integer> busca() {

    int n = padrao.length;
```

```
  int m = texto.length;

  int k = n - 1;
  while (k <= m - 1) {
    int i = n - 1;
    int h = k;
    while (i > -1 && padrao[i] == texto[h]) {
      i--;
      h--;
    }
    if (i != -1) {
      int bomSulfixo = n - 1;
      if (copiaDoSulfixo[i] != -1) {
        bomSulfixo -= copiaDoSulfixo[i];
      } else {
        bomSulfixo -= prefixoSulfixo[i];
      }
      k += Math.max(1,
          Math.max(bomSulfixo, i - caracterErrado[texto[h]]));
    } else {
      resultado.add(k - n + 1);
      k += n-prefixoSulfixo[0];

    }
  }
  return resultado;
}

private int[] maiorSulfixoPrefixo(char[] P, int[] tamanhosDosSulfixos) {
  int n = P.length;
  int[] l = new int[n];
  l[n-1]=tamanhosDosSulfixos[0];
  for (int i = 1; i < n; i++) {
    if(tamanhosDosSulfixos[i]==i+1){
      l[n-1-i]=tamanhosDosSulfixos[i];
    }else{
      l[n-1-i]=l[n-i];
    }
  }
  return l;
}

private int[] posicaoDaCopiaDoSulfixo(char[] P, int[] tamanhosDosSulfixos)
  int n = P.length;
  int[] posicoes = new int[n];
  for (int i = 0; i < n; i++) {
    posicoes[i] = -1;
  }
  for (int j = 0; j < n - 1; j++) {
    int i = (n - 1) - tamanhosDosSulfixos[j];
    posicoes[i] = j;
  }
  return posicoes;
}

private int[] buscaOTamanhoDoSulfixoDeCadaPrefixoQueTambemSejaSulfixo(char[
  int n = P.length;
  int[] prefixos = buscaTamanhoDaCopiaDosPrefixos(Util.inverter(P));
```

```
  int[] sulfixos = new int[n];
  for (int i = 0; i < n; i++) {
    sulfixos[i] = prefixos[n - i - 1];
  }
  return sulfixos;
}

private int[] buscaTamanhoDaCopiaDosPrefixos(char[] S) {
  int n = S.length;
  int[] p = new int[n];
  int r = 0,l=0,i=0,q=0;
  for (int k = 1; k < n; k++) {
    if (k > r) {
      i = 0;
      q = k;
      while (q < n && S[i] == S[q]) {
        i++;
        q++;
      }
      p[k] = q - k;
      if (p[k] > 0) {
        r = k + p[k] - 1;
        l = k;
      } else {
        r = l = 0;
      }
    } else {
      int ka = k - l;
      if (p[ka] < r - (k - 1)) {
        p[k] = p[ka];
      } else {
        i = r - l;
        q = r;
        while (q < n && S[i] == S[q]) {
          i++;
          q++;
        }
        p[k] = q - k;
        r = q - 1;
        l = k;
      }
    }
  }
  return p;
}

public BoyerMoore texto(String texto) {
  this.texto = texto.toCharArray();
  return this;
}

public BoyerMoore padrao(String padrao) {
  this.padrao = padrao.toCharArray();
  return this;
}

public DadosIndividuais executa() {
```

```
    dados.comecaPreProcessamento();
    preProcessamento();
    dados.terminaPreProcessamento();
    dados.comecaBusca();
    dados.ocorrencias(busca());
    dados.terminaBusca();
    return dados;
  }

  private void preProcessamento() {
    this.caracterErrado = calculaCaracterErrado(padrao);
    int[] tamanhosDosSulfixos = buscaOTamanhoDoSulfixoDeCadaPrefixoQueTambemS
    this.copiaDoSulfixo = posicaoDaCopiaDoSulfixo(padrao, tamanhosDosSulfixos
    this.prefixoSulfixo = maiorSulfixoPrefixo(padrao, tamanhosDosSulfixos);
  }

}
```

7 Referências Bibliográficas

Referências

[1] http://pt.wikipedia.org/wiki/Dna

[2] T.H. Cormen, C.E. Leiserson, e R.L. Rivest, 1990, *Introduction to Algorithms*. MIT Press

[3] D. Gusfield, 1997, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, ASIN:0521585198, Cambridge University

[4] Karp, R.M. and M.O. Rabin, 1987, Efficient randomized pattern-matching algorithms. IBM. J. Res. Dev., 31: 249-260. http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf

[5] Morris, J. and V. Pratt, 1970. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley.

[6] Boyer, R.S. and J.S. Moore, 1977. A fast string searching algorithm. Commun. ACM., 20: 762-772.

[7] Código fonte do comando *grep*, http://cvs.savannah.gnu.org/viewvc/grep/grep/src/kwset.c?revisi

[8] Java String Source Code, http://www.docjar.com/html/api/java/lang/String.java.html