Um Estudo Teórico e Experimental em Algoritmos Clássicos de Busca em Texto

PAULO SÉRGIO R. CARVALHO
DEIVE DE OLIVEIRA
ALESSANDRA GUARACY
ALISSON GOMES
FERNADO CESAR DA CONCEIÇÃO
JOSEANE FREIRE
JONES ALBUQUERQUE OLIVEIRA

UFLA - Universidade Federal de Lavras
DCC - Departamento de Ciência da Computação
Cx Postal 37 - CEP 37200-000 Lavras (MG)
{sergio,deive,alessandra,alisson,freire,cesar,joa}@comp.ufla.br

Resumo. Este trabalho apresenta um estudo detalhado dos principais algoritmos de busca exata em textos. O objetivo principal é fornecer um texto didático e informativo, e com um enfoque científico no aspecto análise de complexidade. Ainda, apresentamos resultados experimentais e também nossa conclusão baseada em tais resultados.

Palavras-Chave: padrão, casamento de cadeias, complexidade, empírico, hash, probabilístico.

1 Introdução

Busca em texto é um importante componente de muitos problemas, incluindo editores de texto, recuperação de dados, e manipulação de símbolos.

Os problemas de busca em texto consistem em encontrar todas as ocorrências (ou a primeira ocorrência) de uma palavra no texto. Onde a palavra e o texto são *strings* de um mesmo alfabeto ($\sum$). O importante é reportar todas as ocorrências sabendo-se que a busca por uma palavra de tamanho *m* em um texto de tamanho *n* (onde $n > m$) tem um tempo $O(n)$ no pior caso (para *m* fixo). Esta análise pode ser diferente para diferentes algoritmos.

Dentre os algoritmos existentes, serão abordados neste artigo: KMP (Knuth-Morris-Pratt (1977)), Boyer-Moore (1977) e o Rabin-Karp (1987).

2 Algoritmo de Knutt, Morris e Pratt (KMP)

O algoritmo de KMP foi desenvolvido para resolver o seguinte problema: dada duas strings *A* e *B*, encontrar a primeira ocorrência (se existir) de *B* em *A*. Em outras palavras, pretende-se encontrar o menor *k* tal que, para todo *i*, $1 \leq i \leq n$, tem-se $A_{k+1} = B_i$.

Para descrever o algoritmo de KMP para o problema de casamento de caracteres, deve-se inicialmente, compreender o algoritmo da Força Bruta exemplicado a seguir. O problema consiste em, dadas as cadeias $A = xyxxyxyxyyxyxyxyyx\ yxyxx$ e $B = xyxyyyxyxyxx$, verificar se a subcadeia *B* é encontrada em *A*. Neste algoritmo verifica-se a igualdade das cadeias a partir da primeira posição (ex: $A_1 = B_1$, $A_2 = B_2$, $A_3 = B_3$). Quando ocorre uma desigualdade, (ex: $A_4 \neq B_4$), as comparações começam apartir de A_2. Todas as vezes em que ocorrerem desigualdades, deve-se voltar ao começo e fazer novamente as comparações. Este caminho de volta é exatamente o que torna o algoritmo da Força Bruta ineficiente, sendo o número de comparações e ordem de complexidade $O(m \times n)$ no pior caso, sendo *n* o tamanho da cadeia *A* e *m* o tamanho da cadeia *B*.

O KMP evita exatamente o caminho de volta, sendo portanto mais eficiente.

2.1 Funcionamento do Algoritmo

Primeiramente é feito um pré-processamento da palavra padrão (palavra *B* a ser procurada no texto), que é a essência do algoritmo aperfeiçoado.

Estuda-se todas as repetições da palavra *B* e planeja-se um modo de manusear desigualdades (não-correspondência de caracteres) quando elas ocorrem, sem fazer o caminho de volta. Os passos do algoritmo são:

- a string *A* é sempre discriminada para frente, não há caminho de volta em *A*, embora o mesmo ca-

racter em *A*, possa ser comparado com vários caracteres de *B* (quando há desigualdades).

- Quando uma desigualdade é encontrada, consulta-se uma tabela (neste caso um vetor) para encontrar a que distância em *B* precisa-se fazer o caminho de volta.

- Há uma entrada na tabela para cada caracter em *B* correspondente à quantidade de posições voltadas exigidas quando há uma desigualdade envolvendo este caracter.

Pretende-se encontrar a maior seqüência de caracteres repetidos da string *B*. Se o comprimento (tamanho) dessa seqüência de repetição é *j*, então a não correspondência (não casamento) do caractere em *A* pode ser correspondida ao se comparar $B_{(j+1)}$ diretamente, sem ir através de todas correspondências redundantes.

Sabe-se que os mais recentes caracteres em *A* correspondem ao começo de *B*. A tabela em questão é chamada *next* (onde esta definido os valores das distâncias).

Por conveniência define-se *next[1]*= -1 para distinguir este caso, já que o índice *j* deve estar entre 0 e $(i-1)$. O segundo índice da tabela *next* (*next[2]*) é sempre igual a 0 (desde que não haja *j* que satisfaça $0 < j < 2-1$). Dados os valores de *next*, verifica-se então o "casamento padrão".

A correspondência (casamento) procede-se como segue. Os caracteres em *A* são comparados com aqueles em *B* até que haja uma desigualdade. Nesse ponto (B_i) a tabela *next* é consultada e o mesmo caracter em *A* é comparado novamente com *B next*$[i]+1$ (desde que os primeiros *next[i]* caracteres foram correspondidos).

Se novamente ocorre uma desigualdade, então a próxima comparação é no *B next(next[i]+ 1) + 1* e assim sucessivamente.

A única exceção para esta regra é quando a não correspondência é novamente em B_1, então procede-se em *A*. Este caso pode ser determinado por um valor especial de *next[1]*, que é -1.

2.1.1 Exemplo Ilustrativo da Tabela Next

Dada a string *A ababababb* e o padrão *B abababb*, verificar se ocorre um casamento de *B* em *A*. O primeiro passo é pré-processar o padrão *B* e contruir a tabela *next* para ele.

Iniciando o processamento, verifica-se que ocorre uma desigualdade na quinta comparação, então a tabela *next* tem de ser consultada para saber em que posição de *B* devem ser reiniciadas as comparações. Verifica-se o valor para o quinto caracter na tabela *next* que é 2 e soma 1. Em seguida, retorna-se ao terceiro caracter de *B* e continua-se a comparação com o quinto caracter de *A* (posição em que ocorreu a desigualdade). E assim sucessivamente até ocorrer o casamento padrão ou chegar ao final da string *A*.

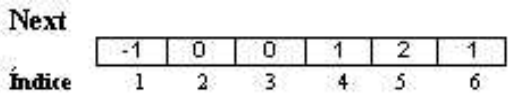


Figura 1: Tabela Next

É importante notar que somente *B* faz o caminho de volta (o índice da comparação decrementa diante de uma não-ocorrência) e que o índice de *A* é sempre incrementado.

2.2 Código

Vamos apresentar o algoritmo de forma genérica [BaY92].

2.2.1 Algoritmo do KMP

```
1. Kmp(string A, B)
2. calcula-next(B,m)
3.  enquanto start=0 e i < m faz
4.   se B[j]= A[i]então
5.     i ← i + 1;
6.     j ← j + 1;
7.   senão j = next[j]+1
8.    se j = 0 então
9.     j ← 1;
10.    i ← i + 1;
11.  se j = m + 1 então
12.    start = i − m;
13.  retorna start;
```

2.2.2 Cálculo do Vetor Next

```
1. Calcular-next(B,m)
2.  next[1]= −1;
3.  next[2]= 0;
4.  Para cada i de 3 até m faz
5.   j ← next[i − 1]+ 1;
6.   enquanto B[i − 1]≠ B[j]e j > 0
7.    j ← next[j]+ 1;
8.   next[i[← j;
9.  retorna next;
```

Calcular-next

Esta função é responsável pela criação da tabela (vetor *next*), onde *i* é o índice da palavra e *m* é o seu tamanho, sendo *j* um índice auxiliar.

KMP

É o algoritmo em si, onde *i* é o índice do texto e *j* o índice da palavra, *m* é o tamanho da palavra e *start* processa a posição inicial do casamento.

2.3 Análise da Complexidade

O algoritmo de Knuth, Morris e Pratt consiste na execução dos algoritmos acima. Considere que *i* e *j*, sejam respectivamente, os índices para os caracteres de *A* e *B*. O índice i possui valor inicial igual a 0 e valor máximo $n - m$ (uma condição de término do algoritmo) e nunca é decrementado. O índice *j* também possui valor inicial 0, mas pode ser decrementado segundo a atribuição $j = next[j] + 1$ observado na linha 7 do algoritmo KMP (string A, B) mostrado anteriormente.

Além disso, quando *j* é incrementado, *i* também é. O número de passos do algoritmo é igual ao número de vezes que *i* é incrementado mais o número de vezes que *j* é decrementado. Ao executarmos as operações entre *A* e *B*, diante de uma não correspondência no caracter B_k , sendo que se incrementou *k* vezes o índice *i*, no pior caso pode-se decrementar *k* posições e reiniciar as comparações.

O número de incrementos é no máximo *n*, (onde *n* é o tamanho do texto). Como *j* não assume valores negativos, o número de decrementos de *j* também não pode ultrapassar *n*, pois o número de decrementos não pode ultrapassar o número de incrementos.

Logo, o algoritmo termina em $O(n)$ passos.

Analogamente, o algoritmo **Calcular-next(B,m)** é efetuado em $O(m)$ passos, onde *m* é o tamanho do padrão (palavra buscada).

Conclui-se desta forma, que a complexidade do KMP é $O(n) + O(m)$.

Este caso ocorre quando tem-se um texto onde os caracteres das palavras ou modelos repetem-se várias vezes, pois assim o KMP evita fazer muitas comparações desnessárias. Quando se tem textos onde é difícil encontrar palavras que possuem caracteres repetidos, sufixo igual ao prefixo, (como ocorre em textos da língua portuguesa), a complexidade do KMP aproxima-se à do Força-Bruta $O(m \times n)$.

Já no melhor caso, dado um texto, e uma palavra a ser buscada, esta seria encontrada em $O(m)$ comparações, ou seja, seria a primeira seqüência de caracteres encontrada no texto.

2.4 Análise da Complexidade Empírica

Esta análise foi baseada no resultado dos testes, mostrados abaixo:
As entradas foram geradas de acordo com o número de caracteres do texto.
Neste caso, consideramos o texto uma seqüência de *X's* e *Y's*, e foi buscada a seqüência "xxxxxxxxxxxxxxxxxxxxxyyyyyyyyyy".
Os resultados das comparações executadas são apresentados na Tabela 1 e ilustrados na Figura 2 :

Entradas	Comparações
1000	1054
2000	1116
3000	3089
4000	4123
5000	5146

Tabela 1: Entradadas X comparações para texto de Xs e Ys

Entradas	Comparações
1000	1968
2000	3869
3000	5895
4000	7795
5000	9693

Tabela 2: Entradas X comparações para texto em língua Inglesa

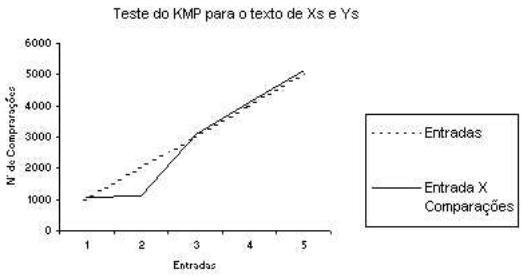


Figura 2: Teste do KMP para texto de Xs e Y

A Tabela 2 apresentam outros testes, onde o texto foi constituído por palavras da Língua Inglesa. A seqüência buscada foi "howareyouiamfineareloveyoumorning".

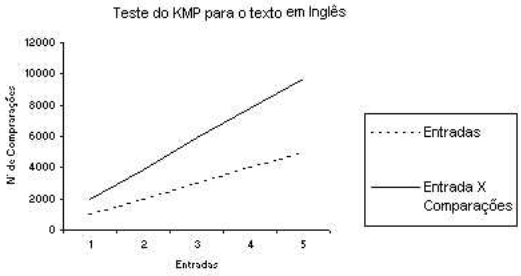


Figura 3: Teste do KMP para texto em inglês

Comparando o desempenho das duas buscas com textos diferentes e as duas palavras com o mesmo tamanho, nota-se que no segundo caso observado nas Figuras 2 e 3 o número de comparações é maior.

Isso se deve ao fato de que o primeiro texto, seqüências de *X's* e *Y's* contém várias repetições, podendo ser associado a uma cadeia de *DNA*. E desta forma o algoritmo é bem mais eficiente, pois elimina comparações desnecessárias.

Já no segundo texto (Língua Inglesa), o algoritmo torna-se menos eficiente, pois quase não há repetições de caracteres ou seja prefixos iguais aos sufixos.

3 Algoritmo de Boyer-Moore

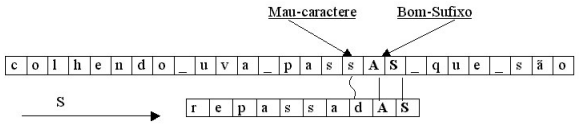
O clássico algoritmo de Boyer-Moore foi publicado por Robert S. Boyer e J. Strother Moore em 1977. A principal idéia do algoritmo é fazer uma procura da direita para a esquerda no padrão a ser buscado.

3.1 Descrição do Algoritmo

Para descrição do algoritmo será utilizada a seguinte notação:

- n: o tamanho do texto;
- m: o tamanho do padrão;
- c: o tamanho do alfabeto σ;
- s: avanço do padrão em relação ao texto;
- C_n: número esperado de comparações realizadas pelo algoritmo quando busca o padrão em um texto de tamanho *n*.

O algoritmo posiciona o padrão sobre o caractere mais à esquerda no texto, e faz uma busca da direita para a esquerda. Se não ocorrer nenhuma diferença, então o padrão foi encontrado. Caso contrário, ocorrerá uma mudança na posição do padrão, que é movido para direita antes que uma nova comparação seja feita. O algoritmo de Boyer-Moore possui duas heurísticas para calcular esta mudança,utilizadas simultaneamente. Essas heurísticas freqüentemente permitem que muitos caracteres sejam "pulados" evitando diversas comparações. Elas são conhecidas como a "heurística-do-bom-sufixo" e a "heurística-do-mau-caractere".



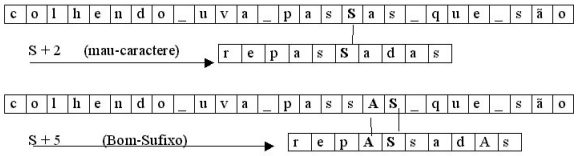


Figura 4: Heurística do Bom-Sufixo e do Mau-Caracter

Essas heurísticas operam independentemente e em paralelo. Quando um erro ocorre no padrão, cada heurística propõe um tamanho para que *s* possa ser seguramente incrementada e a heurística de maior valor é atribuída a *s*.

3.2 Heurística do Mau-Caractere

Quando um erro ocorre, a heurística do mau-caractere usa a informação sobre onde o mau caractere do texto ocorre no padrão para propor uma nova mudança. O melhor caso ocorre quando o erro é encontrado na primeira comparação e o mau-caractere não ocorre em todo o padrão. Procurando-se por a^m em um texto de *string* b^n, incrementa-se a posição de *s* por *m*, desde que qualquer mudança menor que $s+m$ alinhará algum caractere do padrão contra o mau-caractere. Se o melhor caso ocorre repetidamente, o algoritmo de Boyer-Moore examina apenas a fração $1/m$ do texto de caractere, desde que em cada caracter do texto examinado haja um erro, fazendo com que *s* seja incrementado por *m*. Em geral, a heurística do mau-caractere trabalha da seguinte forma: suponha encontrar exatamente um erro: *Padrão [j]≠ Texto [s +j]* para algum j, onde $1 \leq j \leq m$. Pegue *k*, sendo *k* o maior índice da cadeia $1 \leq k \leq m$ tal que *Texto [s+j]= Padrão [k]* se algum *k* existe. De outra maneira pegue *k = 0*. Afirme que pode-se seguramente incrementar *s* por *j - k*. Deve-se considerar três casos para provar essa afirmação:

1. $k = j$: O mau-caractere *Texto[s+j]* não ocorre em todo o padrão, e nós podemos seguramente aumentar *s* por *j* sem perder alguma mudança válida;
2. $k < j$: A ocorrência mais à direita do mau-caractere está no padrão à esquerda da posição *j*, então *j-k* > *0* e o padrão pode ser movido *j-k* caractere para a direita antes que o mau-caractere do texto iguale a algum caractere do padrão. Portanto, pode-se seguramente incrementar *s* por *s-j* sem perder qualquer mudança válida;
3. $k > J$: $j-k < 0$, sendo a heurística do mau-caractere essencialmente proposta para decrementar *s*, essa recomendação será ignorada pelo algoritmo de

Boyer-Moore, desde que a heurística do bom-sufixo proponha uma mudança para a direita em todos os casos.

3.3 Heurística do Bom-Sufixo

Quando encontra-se o padrão diferente do texto na posição *Padrão [j]* e *Texto [s + j]*, onde $j < m$, então a heurística do bom-sufixo assegura que pode-se avançar *s* por:
γ *[j]= m-max{k: 0* $\leq$ *k* $<$ *m e P [j+1...m]}* [Cor97].

A função $\gamma[j]$ é o menor tamanho que pode-se avançar *s*, sendo que neste novo alinhamento não ocorrerá uma diferença entre o bom-sufixo e o caractere do padrão.

A função γ é chamada de função do bom-sufixo para o padrão *P*.

3.4 Código

Algoritmo de Boyer-Moore [Cor97].

3.4.1 Boyer-Moore

```
Boyer-Moore-Matcher(T,P,∑)

 1.  n ← length[T];
 2.  m ← lenght[P];
 3.  λ ← Compute-Last-Ocorence-Func-
     tion(P,m,∑);
 4.  γ ← Compute-Good-Suffix-Func-
     tion(P,m);
 5.  s ← 0;
 6.  while s ≤ n-m;
 7.   do j ← m
 8.    while j > 0 && P[j]= T[s+j];
 9.     do j ← j-1;
10.    if j = 0
11.     then imprime "padrão ocorre po-
     sição" s;
12.       s ← s+ γ[0];
13.     else s←s+max(γ[j],j-γ[T[s+j]]);
```

3.4.2 Heurística do Mau-Caractere

```
Compute-Last-Ocorence-Function(P,m,∑)

 1.  for cada caracter a ∈ ∑
 2.   do λ[a]= 0;
 3.  for ← 1 até m
 4.   do λ[P[j]]← j;
 5.  return λ;
```

3.4.3 Heurística do Bom-Sufixo

```
Compute-Good-Suffix-Function(P,m)

 1.  ∏ ← Compute-Prefix-Function(P);
 2.  P' ← reverse(P);
 3.  ∏' Compute-Prefix-Function(P');
 4.  for j ← 0 até m
 5.   do γ[j]← m- ∏[m];
 6.  for i ← 1 até m
 7.   do γ[j ]> i - ∏'[i];
 8.    if γ[j ]> i - ∏'[i]
 9.     then γ[j]← i - ∏'[i];
10.  return γ;
```

A complexidade do algoritmo de Boyer-Moore para o pior caso é:
$O((n - m + 1)m + |\sum|)$.

Sendo que a heurística do bom-sufíxo apresenta uma complexidade de ordem $O(2m) = O(m)$; A heurística do mau-caracter apresenta uma complexidade de ordem $O(m + |\sum|)$.

4 Análise dos Resultados

Os testes foram realizados com as seguintes entradas:
$|\sum| = 90$;
$|P| = 2$ a 30;
Número de caracteres = 15448.

Na Figura 5 observa-se que o algoritmo se comporta conforme o estudo de sua complexidade teórica.

Pode-se ver que quanto maior o padrão, menor o número de comparações, pois *s* é incrementado de acordo com valor do padrão.

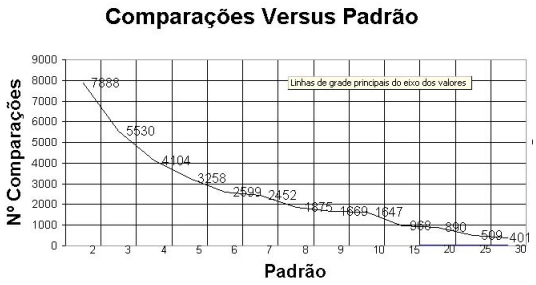


Figura 5: Análise Empírica

5 Algoritmo de Rabin-Karp

O algoritmo de Rabin-Karp tem característica probabilística, pois ele transforma a palavra procurada em um número, seguindo determinadas regras, que serão apresentadas posteriormente. Para a descrição do algoritmo será utilizada a seguinte notação:

- n: número de caracteres do texto;
- m: número de caracteres da palavra;
- d: cardinalidade do alfabeto $\sum$;
- q: número primo, como: 16647133;
- T: string do texto;
- P: string da palavra.

5.1 Descrição do Algoritmo

Harrison (1971) [Cor97] sugeriu o uso de técnicas de hashing, as quais são utilizadas neste algoritmo para resgatar palavras de uma string, proposto por Rabin-Karp. Tudo que é necessário é processar a função de assinatura de cada substring de *m*-caracteres no texto e checar se é igual à assinatura da função da palavra procurada.

Karp e Rabin (1987)[BaY92] encontraram uma forma fácil para processar estas funções de assinatura eficientemente para a função h(k) = k mod q, onde q é um número primo grande. O método é baseado no processamento da função de assinatura para a posição i dando o valor para a posição $i-1$. O algoritmo requer tempo proporcional a $n+m$ em quase todos os casos, isto não utilizando memória virtual. Note-se que este algoritmo encontra posições no texto que tem o mesmo valor de assinatura que a palavra. Para garantir a busca, deve-se fazer uma comparação direta entre a palavra e a substring. Este algoritmo, usa um elevado valor para q fazendo com que as colisões diminuam. A probabilidade de uma colisão aleatória é $O(1/q)$.

Teoricamente, este algoritmo pode ainda requerer *mn* passos no pior caso, checando cada busca em potencial. Em testes realizados verificou-se apenas 3 colisões em 10^7 [BaY92] processamentos de assinaturas, usando grandes alfabetos ($\sum$).

A função de assinatura representa a string como um número na base-*d*, onde *d* é o número de caracteres possíveis. Para obter o valor de assinatura da próxima posição, apenas um valor constante de operações é necessário.

Na prática, este algoritmo é lento para as duas operações de multiplicação e módulo. Entretanto, ele se torna computacionalmete viável para grandes palavras.

5.2 Código

A seguir será apresentado o algoritmo de forma genérica [Cor97], descrevendo seus passos posteriormente.

```
Rabin-karp-Matcher(T,P,d,q)
 1. n ← tamanho[T];
 2. m ← tamanho[P];
 3. h ← d^(m−1) mod q;
 4. p ← 0;
 5. t_0 ← 0;
 6. for i ← to m
 7.  do p ← (dp + P[i]) mod q;
 8.     t_0 ← (dt_0+ T[i]) mod q;
 9. for s ← 0 to n−m
10.  do if p = t_s
11.     then if P[1··m] = T[s+1··s+m]
12.        then encontrou a palavra;
13.     if s< n−m
14.        then t_{s+1} ← (d(t_s − T[s+1]h) +
    T[s+m+1]) mod q;
```

5.3 Funcionamento do Algoritmo

O algoritmo pode ser dividido da seguinte maneira:

1. Correspondente às linhas $[1-5]$. Este bloco apresenta instruções de inicialização das variáveis utilizadas no código. É importante ressaltar que a implementação da função *hash* é arbitária;

2. Correspondente às linhas $[6-8]$. Este bloco constitui-se de uma iteração com índice relacionado à cardinalidade da palavra. O bloco inicializa o *hash(p)* da palavra e do texto, *hash(t)*;

3. Correspondente às linhas $[9-14]$. Neste bloco tem-se a iteração de comparação; na linha 10 em caso da comparação ser válida temos a possibilidade da substring encontrada ser a procurada. Observa-se que esta linha apresenta o caráter probabilístico do algoritmo. Uma comparação extra é realizada para confirmar o resultado; não comparando a *hash* como na anterior mas os caracteres da substring com a palavra.

   Em caso negativo da comparação, temos um novo valor de *hash(t)* para a substring, realizando a iteração até encontrar a palavra(s) ou se esgotar o texto.

6 Complexidade

A complexidade deste algoritmo, no seu pior caso, é $O((n-m+1)m)$, podendo ser reduzida a $O(nm)$, este caso ocorreria quando o número de colisões fosse elevado, o que obrigaria o algoritmo a realizar um maior número de comparações e operações para se obter um novo valor de hash(t). Nos demais casos, médio e melhor, este algoritmo trabalha em $O(m+n)$. A prova para estes casos é complicada, devido principalmente ao carater probabilístico deste algoritmo, pode-se obter demonstração em [Cor97].

Apesar de ser probabilístico este algoritmo não apresenta-se eficiente na busca de textos que estejam em constante mudança, veja a Figura 6, já para textos que não são modificados constantemente ele poderá apresentar um desempenho melhor devido ao aproveitamento da tabela *hash*, para encontrar estas palavras.

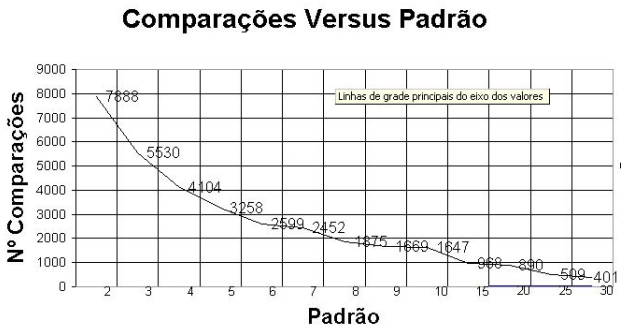


Figura 6: Análise Empírica

7 Conclusão

Diante dos resultados obtidos nos testes e exemplificados nos gráficos observa-se que o algoritmo de Boyer-Moore é o mais eficiente para busca de palavras maiores em textos comuns, considerando-se as seguintes situações:

1. O padrão deve ter um tamanho considerável (mais de 10 caracteres);

2. O alfabeto também deve ser grande.

O algoritmo KMP mostra-se mais eficiente com cadeias de caracteres repetidos, podendo ser associado a uma cadeia de *DNA*. Sua eficiência deve-se ao fato de utilizar um vetor auxiliar (*next*) que armazena as posições das quais se reiniciarão as comparações. Em textos comum o algoritmo de KMP é próximo ao algoritmo da Força-Bruta em relação à complexidade.

O algoritmo de Rabin-Karp tenta diminuir o número de comparações através do cálculo de um *hash* da palavra procurada e da substring, porém através deste procedimento ele se torna probabilístico.

Referências

[Cor97] Cormen, T., Leiserson, C., e Rivest, R. 1997. "Introduction to Algorithms,"em The Rabin-Karp algorithm, pp. 857-883.

[BaY92] Baeza-Yates, R., Frakes, W. 1992. "Information Retrieval Data Structures & Algorithms,"em String Searching Algorithms, pp. 219-237.