Algoritmos e Estrutura de Dados III (2019-1)

Trabalho Prático 4

Heitor Lourenço Werneck
heitorwerneck@hotmail.com

1 de Julho de 2019

1 Introdução

O trabalho a ser apresentado consiste em desenvolver algoritmos de busca de padrão, exato e aproximado, avaliando o desempenho dos mesmos em diversos ambientes. A busca de padrão é formalizada do seguinte modo: Há um texto $T[1..n]$ de largura n e o padrão é $P[1..m]$ de largura $m \leq n$. Também é assumido que os elementos de P e T tem como fonte de seus caracteres um dicionário Σ finito. O padrão P ocorre em T na posição x do texto se $1 \leq x \leq n - m + 1$ e $T[x..x + m - 1] = P[1..m]$. Um texto vazio é denotado por ε. [4]

Busca de padrão é usada em busca por sequencias de DNA, análise de registro de sistemas de informação e também esta muito presente a utilização por usuários de computadores em buscas de texto na Internet.

Os algoritmos desenvolvidos para solução do problema foram: Boyer Moore Horspool (BMH); BMH Sunday (BMHS); Shift-And; Shift-And aproximado; BMHS paralelo.

2 Estrutura de dados

Para o armazenamento do texto e padrão foi suposto que o texto de entrada será ASCII, logo foi criado o tipo *CharType* que é o tipo primitivo *unsigned char* da linguagem C que é usado para guardar tanto o texto quanto o padrão.

```
typedef unsigned char CharType;
```

Também é importante notar que o tipo definido suporta somente um número constante de caracteres que varia de arquitetura para arquitetura. Como definido anteriormente, o dicionário possui um número finito de elementos. Logo é definido o Tamanho do alfabeto como:

$$ALPHABETSIZE = |\Sigma| := 256 = 2^{(sizeof(CharType)\cdot 8)} \tag{1}$$

Outra estrutura criada que auxilia no ShiftAnd é o *bitVector* que possui a seguinte definição: [1]

```
typedef unsigned char WordType;
struct bitVector{
  size_t bits;
  WordType *vector;
};
```

Sendo `size_t bits` o número de bits utilizado pelo `WordType *vector` e `WordType *vector` a lista de bits por si.

Essa estrutura de dados é representada na figura 1.

Todas operações binárias necessárias para o Shift-And foram implementadas. Ou seja: ou(|); e(&); deslocamento para a direita(>>). Também outras funções foram implementadas para operar bit a bit: uma para definir como 1 o enésimo bit, outra para definir como 0 o enésimo bit e também mais uma para checar o enésimo bit.

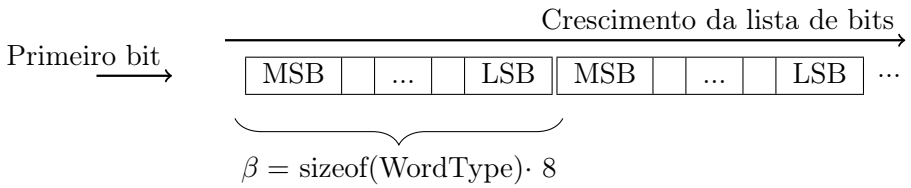


Figura 1: Vetor de bits.("MSB": Bit mais significativo; "LSB": bit menos significativo.)

```
bitVector* bVSetBit(bitVector* bitv,size_t bit);// Define como 1 enésimo bit
bitVector* bVClearBit(bitVector* bitv,size_t bit);// Define como 0 enésimo bit
bitVector* bVOr(bitVector* dest,bitVector* src1,bitVector* src2);// Operação logica ou
bitVector* bVAnd(bitVector* dest,bitVector* src1,bitVector* src2);// Operação logica e
bitVector* bVClearAll(bitVector* target);// Define como 0 todo o vetor de bits
bitVector* bVShiftLeft(bitVector* dest,bitVector* src,size_t shifts);// Deslocamento para esquerda
bitVector* bVShiftRight(bitVector* dest,bitVector* src,size_t shifts);// Deslocamento para a direita
int bVIsSetBit(bitVector* target,size_t bit);// Define como 1 enésimo bit
```

3 Solução

3.1 Boyer Moore Horspool

Booyer Moore Horspool(BMH) é um algoritmo de casamento exato de padrão que compara um padrão da direita para a esquerda e passa pelo texto da esquerda para a direita.

O preprocessamento necessário para o algoritmo é a criação de uma tabela que indica o deslocamento que poderá ser realizado. Quando comparado um caractere do texto ao do padrão, se for diferente ou haver um casamento irá haver um deslocamento de acordo com o caractere no texto correspondente ao ultimo caractere do padrão. A construção da tabela de deslocamento se da pelo modelo apresentado na equação 2.

$$d[x] = min\{j \text{ tal que } j = m | (1 \leq j < m \& P[m-j] = x)) \tag{2}$$

O algoritmo 1 descreve o comportamento detalhadamente.

Algoritmo 1 Boyer Moore Horspool.

Input: $T[0..n-1], n, P[0..m-1], m, match[0..n-1]$ **Output:** $match[0..n-1]$

```
1: procedure preProcessBMH(d, P, m)
2:     for i = 0 to ALPHABETSIZE − 1 do
3:         d[i] ← m
4:     for i = 0 to m − 2 do
5:         d[P[i]] ← m − i − 1
6: procedure BMH(T, n, P, m, match)
7:     match ← {0, ..., 0}
8:     preProcessBMH(d, P, m)
9:     i ← m
10:    while i ≤ n do
11:        base ← i
12:        j ← m
13:        while j > 0 ∧ T[base − 1] = P[j − 1] do
14:            j ← j − 1; base ← base − 1
15:        if j = 0 then
16:            match[base] ← true
17:        i ← i + d[T[i − 1]]
```

3.2 Boyer Moore Horspool Sunday

O Boyer Moore Horspool Sunday(BMHS) é um algoritmo também de casamento exato de padrão. O algoritmo é bem similar ao BMH, porém apresenta algumas variantes. O BMHS usa uma abordagem diferente para o modelo de deslocamento. O endereçamento a tabela e dado pelo caractere no texto correspondente ao caractere após o último caractere do padrão. O modelo da tabela de deslocamento é a equação 3.

$$d[x] = min\{j \text{ tal que } j = m | (1 \leq j \leq m \& P[m+1-j] = x)) \tag{3}$$

O algoritmo 2 descreve o comportamento do BMHS detalhadamente.

Algoritmo 2 Boyer Moore Horspool Sunday.

Input: $T[0..n-1], n, P[0..m-1], m, match[0..n-1]$ **Output:** $match[0..n-1]$

```
1: procedure preProcessBMHS(d, P, m)
2:     for i = 0 to ALPHABETSIZE − 1 do
3:         d[i] ← m + 1
4:     for i = 0 to m − 1 do
5:         d[P[i]] ← m − i
6: procedure BMHS(T, n, P, m, match)
7:     preProcessBMHS(d, P, m)
8:     i ← m
9:     while i ≤ n do
10:        base ← i
11:        j ← m
12:        while j > 0 ∧ T[base − 1] = P[j − 1] do
13:            j ← j − 1; base ← base − 1
14:        if j = 0 then
15:            match[base] ← true
16:        if i < n then
17:            i ← i + d[T[i]]
18:        else
19:            break
```

3.3 Shift-And

O Shift-And é um algoritmo que explora operações que tiram proveito do tamanho da palavra da arquitetura utilizada, com isso consegue um paralelismo assim reduzindo o numero de operações que o algoritmo realiza em até um fator de β. O Shift-And pode ser visto como uma simulação de um autômato não determinista que pesquisa pelo padrão no texto.

A modelagem do problema foi feita de um modo generalista, ou seja, para resolver o problema de casamento aproximado. O casamento exato é uma especificação desse problema e pode ser resolvido com um erro definido como zero. Ou seja, algoritmo exato: ShiftAnd$(T, n, P, m, 0)$.(algoritmo 3)

Um exemplo de autômato para casamento aproximado é mostrado na figura 2.

Um caractere quando é casado é representado pela aresta horizontal, a aresta vertical insere um caractere no padrão(R_{j-1}), a aresta diagonal continua substitui um caractere($R_{j-1} >> 1$) e a aresta diagonal tracejada remove($R'_{j-1} >> 1$).

Cada linha j ($0 < j \leq error$) do autômato representa um erro diferente, então o algoritmo ShiftAnd reúne essas linhas em uma palavra R_j. Os vetores de bits $R[0..error]$ são atualizados a cada caractere lido do texto e a simulação do autômato é feita através de operações em bits.

Para simular esse autômato o Shift-And começa criando uma máscara de bits para cada palavra do alfabeto e quando um caractere é encontrado na enésima posição do texto é também definido como 1 na enésima posição da máscara de bits do carácter. É importante notar que as operações no vetor de bits dependem de como o vetor de bits é definido, a imagem 1 ilustra o modelo de vetor de bits utilizado nesse trabalho.

O seguinte modelo descreve o comportamento que deve ocorrer para realizar os casamentos a cada caractere na posição i do texto:(equação 4)

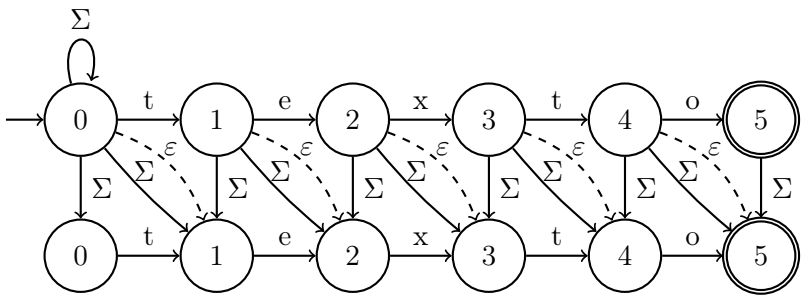


Figura 2: Autômato de busca com erro 1.

$$R_0^{'} = ((R_0 >> 1)|1)\&mask[T[i]]$$
$$R_j^{'} = (R_j >> 1)\&mask[T[i]]|R_{j-1}|(R_{j-1} >> 1)|(R_{j-1}^{'} >> 1) \quad (4)$$

Com a máscara definida é possível verificar os casamentos que vão ocorrendo no texto. O algoritmo 3 descreve o comportamento detalhadamente.

Algoritmo 3 Shift-And.

Input: $T[0..n-1], n, P[0..m-1], m, match[0..n-1], error$ **Output:** $match[0..n-1]$

```
1: procedure ShiftAnd(T, n, P, m, match, error)
2:     R1 ← 1
3:     mask[0..ALPHABETSIZE − 1] ← {0, ..., 0}, R[error + 1] ← {0, ..., 0}
4:     for i = 0 to m − 1 do
5:         SetBit(mask[P[i]], i)
6:     ClearAll(R[0]); aux ← bitVectorNew(m)
7:     for i = 1 to error do
8:         R[i] ← R[i − 1]
9:         SetBit(R[i], i − 1)
10:    for i = 0 to n − 1 do
11:        Rprevious ← R[0]
12:        Rnew ← ((Rprevious >> 1)|R1) & mask[T[i]]
13:        R[0] ← Rnew
14:        for j = 1 to error do
15:            Rnew ← ((R[j] >> 1)&mask[T[i]])|Rprevious|Rprevious >> 1|Rnew >> 1
16:            Rprevious ← R[j]
17:            R[j] ← Rnew|R1
18:        if IsSetBit(Rnew, m − 1) ∧ (n > i + 1 − m) ∧ (i + 1 − m ≥ 0) then
19:            match[i + 1 − m] ← true
```

3.4 BMHS Paralelo

Algumas definições iniciais são necessárias para analisar o problema de paralelizar o BMHS. Seja p o número de processadores, P_i o processador i, $0 \leq i < p$, **for all** .. **end for** ativa conjunto de processadores, n o tamanho do texto como já definido.

Para separar o problema do texto em partes é possível usar a seguinte formula para dividir o problema:

$$\alpha(i) = \text{Começo}(i) = \left\lfloor \frac{i \cdot n}{p} \right\rfloor; \text{Fim}(i) = \left\lfloor \frac{(i+1) \cdot n}{p} \right\rfloor \quad (5)$$

Esse tipo de divisão do texto não trata o caso que o padrão esta entre um bloco de texto e outro. Para tratar isso a formula do fim pode ser redefinida adicionando uma soma do tamanho do padrão menos um, deste modo será possível sempre obter o padrão. (equação 6)

$$\omega(i) = \text{Fim}(i) = \begin{cases} \left\lfloor \frac{(i+1) \cdot n}{p} \right\rfloor - 1 + m & \text{se } i < p-1 \\ \left\lfloor \frac{(i+1) \cdot n}{p} \right\rfloor & \text{se } i = p-1 \end{cases} \quad (6)$$

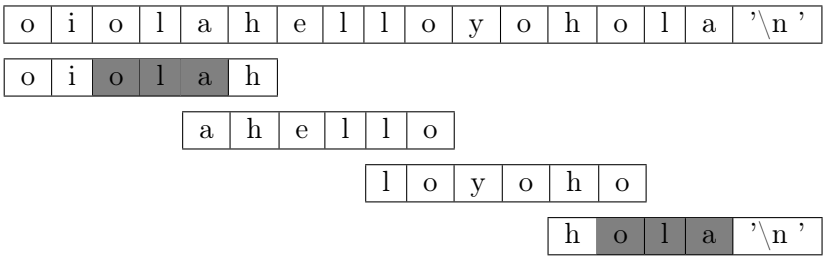


Figura 3: Distribuição de dados pelos processadores com texto com n=17, p=4, P=ola.

Essa formula define o começo e fim da busca para o BMHS. A posição real no texto T[0..n-1] é dada como: Começo $= \alpha(i)$ e Fim $= \omega(i) - 1$. A figura 3 exemplifica a divisão de blocos realizada.

Também é interessante caracterizar os blocos criados para paralelizar o algoritmo. O tamanho de cada bloco sem ser o ultimo é dado pela regra: $\lfloor n/numThreads + m - 1 \rfloor$

logo o tamanho total de blocos é dado por:

$$T(numThreads, n, m) = (numThreads-1)\cdot\left\lfloor \frac{n}{numThreads} + m - 1 \right\rfloor + n - \left\lfloor (numThreads - 1) \cdot \frac{n}{numThreads} \right\rfloor \tag{7}$$

Essa função de tamanho consegue mostrar o aumento de tamanho obtido por paralelizar o algoritmo. $(T(numThreads, n, m)/n)$

Antes da criação do algoritmo definitivamente é necessário definir um limite superior de divisão de blocos. Esse limite é dado por $\lceil n/m \rceil$ blocos pois a propriedade $m \leq n$ deve se manter para não ocorrer de um bloco menor que m pois não há possibilidade de haver um casamento. Logo o número de threads máximo e necessário é $\lceil n/m \rceil$. Com toda essa base o algoritmo 4 é definido.

Algoritmo 4 BMHS Paralelo.

Input: $T[0..n-1], n, P[0..m-1], m, match[0..n-1]$ **Output:** $match[0..n-1]$

```
1: procedure PBMHS
2:     preProcessBMHS(d, P, m)
3:     if numThreads > ⌈n/m⌉ then
4:         numThreads = ⌈n/m⌉
5:     for all P_i, 0 ≤ i < numThreads do
6:         begin ← α(i)
7:         end ← ω(i)
8:         PBMHSF(T, n, P, m, match, d, begin, end)
9: procedure PBMHSF(T, n, P, m, match, d, begin, end)
10:    i ← begin + m
11:    while i ≤ end do
12:        base ← i
13:        j ← m
14:        while j > 0 ∧ T[base − 1] = P[j − 1] do
15:            j ← j − 1; base ← base − 1
16:        if j = 0 then
17:            match[base] ← true
18:        if i < n then
19:            i ← i + d[T[i]]
20:        else
21:            break
```

```
T:aaaaaaaa
P:aaaa
   aaaa
    aaaa
     aaaa
      aaaa
```

Figura 4: Exemplo de execução BMH.

4 Análise de complexidade

4.1 Complexidade de tempo

4.1.1 BMH

$$\begin{aligned} \text{preProcessBMH} &\in O(|\Sigma| + m) \\ \text{BMH} &\in O(n \cdot m + |\Sigma| + m) \end{aligned} \tag{8}$$

Simplificando a complexidade do algoritmo é dada por: $O(n \cdot m)$. No melhor caso é $O(n/m)$ e o caso esperado é $O(n/m)$ [5], essas afirmações são compartilhadas pelo BMHS e BMHS Paralelo.

O pior caso desse algoritmo se expressa no exemplo dado na figura 4.

4.1.2 BMHS

$$\begin{aligned} \text{preProcessBMHS} &\in O(|\Sigma| + m) \\ \text{BMHS} &\in O(n \cdot m + |\Sigma| + m) \\ \text{BMHS} &\in O(n \cdot m) \end{aligned} \tag{9}$$

4.1.3 Shift-And

Para analisar a complexidade de tempo do ShiftAnd é necessário analisar a complexidade das operações binárias criadas para auxiliar. Seja $bits$ o número de bits alocado para o vetor de bits, $shifts$ é a quantidade de deslocamentos a serem realizados e $\beta = sizeof(WordType) \cdot 8$ como definido na figura 1.

$bVSetBit$	$bVClearBit$	$bVOr$	$bVAnd$	$bVClearAll$	$bVShiftLeft$	$bVShiftRight$	$bVIsSetBit$
$O(1)$	$O(1)$	$O(\frac{bits}{\beta})$	$O(\frac{bits}{\beta})$	$O(\frac{bits}{\beta})$	$O(\frac{shifts \cdot bits}{\beta})$	$O(\frac{shifts \cdot bits}{\beta})$	$O(1)$

Para simplificar essa tabela somente com os termos necessários os termos constantes podem ser removidos, o termo β é uma constante logo pode ser removido, o shifts também tem um limite pois no máximo o deslocamento é do tamanho da palavra e nenhum comportamento diferente ocorre além disso logo a tabela pode ser reescrita como:

bVSetBit	bVClearBit	bVOr	bVAnd	bVClearAll	bVShiftLeft	bVShiftRight	bVIsSetBit
O(1)	O(1)	O(bits)	O(bits)	O(bits)	O(bits)	O(bits)	O(1)

O custo para inicializar as máscaras com 0 é $O(|\Sigma|)$. O custo para inicializar os R's é $O(error)$. No caso $bits = m$ logo a complexidade do ShiftAnd será:

$$\begin{aligned} & O(m \cdot (|\Sigma| + error + n \cdot (error + 1))) \\ \text{ShiftAnd} \in & O(m \cdot (|\Sigma| + n \cdot (error + 1) + error)) \\ & \text{ShiftAnd} \in O(m \cdot n \cdot error) \end{aligned} \tag{10}$$

Com um m menor que β ou pelo menos mantendo-se relativamente pequeno o algoritmo irá apresentar complexidade $O(|\Sigma| + n \cdot (error + 1))$.

4.1.4 BMHS Paralelo

A complexidade da criação das numThreads threads é $O(numThreads)$ ou $O(p)$. Sendo thread uma tarefa que um programa executa.

A complexidade é simplesmente $PBMHS \in O(m \cdot n/p + p)$ pois o trabalho é distribuído para p processadores que conseguem trabalhar simultaneamente no problema.

4.2 Complexidade de espaço

4.2.1 BMH

O algoritmo BMH somente aloca uma tabela de deslocamento com complexidade de espaço $O(|\Sigma|)$. Logo sua complexidade é $O(|\Sigma|)$.

4.2.2 BMHS

O BMHS utiliza somente uma tabela de deslocamento assim como o BMH, sua complexidade de espaço é $O(|\Sigma|)$.

4.2.3 Shift-And

O ShiftAnd usa um vetor de máscara de bits que tem custo de espaço de $O(m \cdot |\Sigma|)$ pois cada máscara para cada caractere deve ser do tamanho do padrão. Também é utilizado um vetor de vetor de bits para guardar os resultados do processo de casamento, esse vetor cresce de acordo com o número de erros permitidos no padrão logo sua complexidade de espaço é $O(m \cdot error)$. Logo a complexidade de espaço do ShiftAnd é $O(m \cdot (|\Sigma| + error))$.

4.2.4 BMHS Paralelo

O BMHS paralelo utiliza uma tabela de deslocamento com complexidade de espaço $O(|\Sigma|)$, também necessita guardar as threads criadas, logo a complexidade do algoritmo será $PBMHS \in O(|\Sigma| + p)$.

4.3 Análise geral

A tabela 1 mostra todas complexidades obtidas. Em uma análise de contexto geral a complexidade de espaço será pelo menos $O(n + m)$ que é o tamanho do texto e do padrão. Como o texto e padrão nos algoritmos são somente uma referência para um vetor não há alocação dinâmica de vetor logo a complexidade de espaço dentro das funções é constante em relação a esse caso.

Complexidade	Tempo	Espaço
BMH	$O(n \cdot m + \|\Sigma\| + m)$	O($\|\Sigma\|$)
BMHS	$O(n \cdot m + \|\Sigma\| + m)$	O($\|\Sigma\|$)
ShiftAnd	$O(m \cdot (\|\Sigma\| + n \cdot (error + 1) + error))$	O(m· ($\|\Sigma\|$+error))
PBMHS	$O(\frac{m \cdot n}{p} + p)$	O($\|\Sigma\|$+p)

Tabela 1: Complexidades.

5 Resultados

A máquina utilizada para os experimentos possui as seguintes especificações: Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz e 4GiB de memória RAM.

Para fazer a análise de resultados foi utilizado um banco de dados de genomas [2].

A tabela 2 mostra todas bases utilizadas nos experimentos. A base ι é padrão para a análise do tempo e a base ζ para a análise de espaço. A base A é feita somente do caractere a.

O β na arquitetura utilizada foi de 64. Em gráficos com o Shift-And aproximado junto com outros algoritmos por padrão o erro é 2. Em gráficos onde o m não é variável é utilizado como padrão $m = 30$.

(a) $m \leq \beta$. **(b)** m sem limite.

Figura 5: Tempo de acordo com tamanho do padrão na base ι.

Para análise de tempo do algoritmo paralelo foi primeiro observado a relação de tempo de usuário, tempo de sistema e tempo real. O tempo de usuário mais o tempo de sistema dividido pelo número de processadores deu resultado similar ao tempo real, em uma condição onde o computador foi utilizado majoritariamente somente pelos algoritmos aqui apresentados, logo foi utilizado o tempo total dividido pelo número de processadores.

Base	Abreviação	n
Drosophila grimshawi strain (195MB)	μ	204854690
Anopheles koliensis (149MB)	λ	156699065
Caenorhabditis elegans (97MB)	κ	101540352
Marssonina coronariae (49MB)	ι	50949113
Salmonella enterica (5MB)	θ	5198187
Enterococcus faecalis (3MB)	η	3402262
Chlamydia gallinacea (1MB)	ζ	1080685
A	A	1000000

Tabela 2: Bases para criação de resultados.[2]

5.1 Tempo

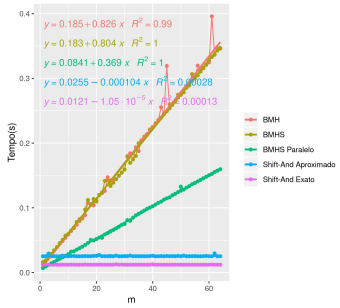
Os fatores que influenciam no tempo de execução são conhecidos, isto é, o tamanho e conteúdo do texto; tamanho e conteúdo do padrão; número de erros para casamento aproximado; número de processadores para algoritmo paralelo. Primeiro será analisado a influência do tamanho do padrão nos algoritmos.

5.1.1 Tamanho do padrão (m)

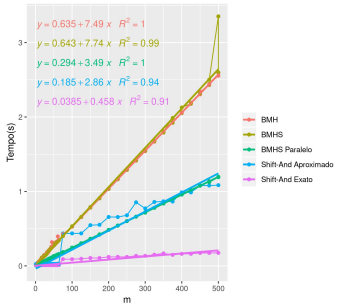
É possivel observar que o comportamento do ShiftAnd é constante para $m \leq \beta$ na figura 5a. Já o BMH, BMHS e BMHS Paralelo variam com o tamanho de m, como o caso esperado é $O(n/m)$ é possível entender a decaída de tempo de acordo com o aumento de m, observe como na figura 6 o tempo decai exatamente como foi discutido o caso esperado. Na figura 5b é mostrado $m \geq 1$, pode ser visto que o ShiftAnd possui um salto, esse salto é devido ao fato do algoritmo utilizado para $m > \beta$ possuir um *overhead* da biblioteca de manipulação de bits. E a partir de β em β o tempo do algoritmo aumenta ($Tempo(\left\lceil \frac{m}{\beta} \right\rceil)$), pode ser observado que em certos intervalos de m o tempo se mantém para m diferentes o que é esperado.

Porém é necessário analisar o pior caso do algoritmo, esse caso é o caso que é necessário fazer o máximo de comparações possíveis, e pode ser criado assim como foi visto na seção de análise de complexidade. Os resultados no pior caso utilizando a base A são mostrados na figura 7. A diferença entre o caso com entradas de padrão casuais e uma forçando o pior caso é enorme, no pior caso o BMH, BMHS e BMHS Paralelo são piores que o Shift-And exato. O Shift-And aproximado ainda consegue ficar bem próximo do BMHS Paralelo para $m \geq \beta$.

Com as regressões e de acordo com o R^2 elevado a complexidade na questão de m foi comprovada.

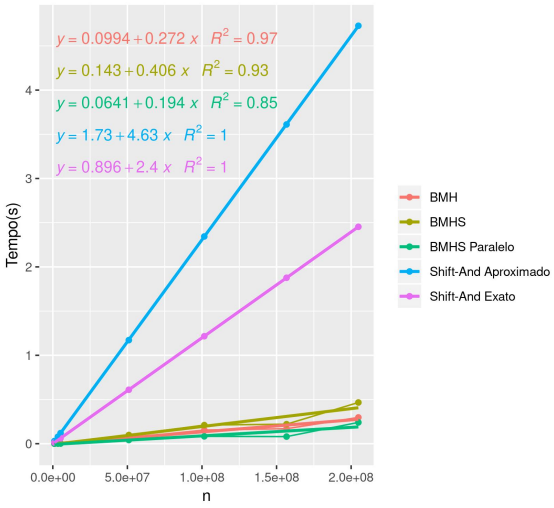


(a) $m \leq \beta$.

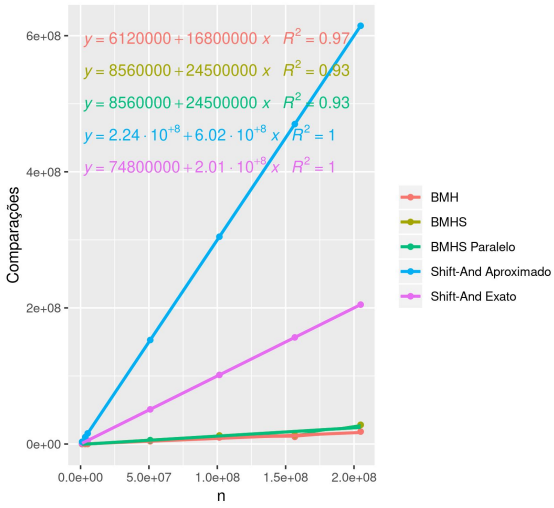


(b) m sem limite.

Figura 7: Tempo de acordo com tamanho do padrão na base A, pior caso.



(a)



(b)

Figura 8: Variação de n com todas bases exceto A e m=30.

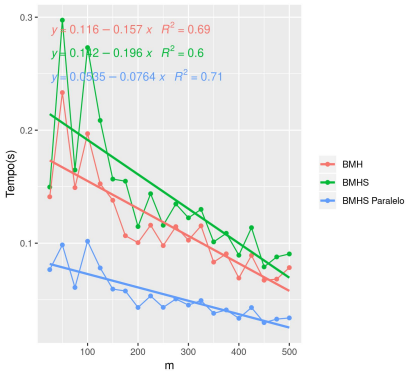
5.1.2 Tamanho do texto (n)

O próximo passo é analisar e comprovar a influência do tamanho do texto T nos algoritmos.

A figura 8a mostra o crescimento do tempo de acordo com o tamanho do arquivo. O gráfico mostra que há um crescimento linear do tempo de acordo com o tamanho do arquivo para todos algoritmos, assim comprovando uma parte da complexidade obtida.

A figura 8b mostra o crescimento das comparações. Os números de comparações formam um gráfico similar ao gráfico com tempo, assim reforçando que o gráfico obtido com tempo é correto. Esses gráficos reforçam a inferioridade do Shift-And para casos médios, basta observar a diferença do número de comparações de qualquer um dos outros algoritmos.

Figura 6: Base ι.



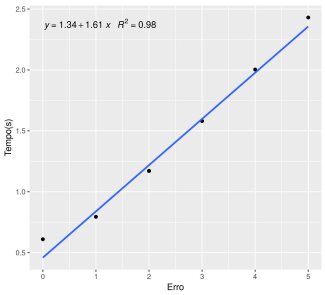
5.1.3 Erro(error)

Agora é necessário avaliar o impacto do número de erros no tempo de execução. A figura 9 mostra o comportamento, o tempo cresce linearmente como previsto logo foi comprovado o termo de erro na complexidade tempo do algoritmo ShiftAnd.

5.1.4 Processadores(p)

Agora basta analisar o comportamento do algoritmo BMHS paralelo com o crescimento do número de processadores. Primeiro é importante definir alguns termos para ser feito a avaliação. O primeiro deles é o speedup que é igual a $S = \frac{\text{Tempo do algoritmo sequencial usando 1 processador}}{\text{Tempo do algoritmo paralelo usando p processadores}}$, o próximo termo é a eficiência que será igual a $E = \frac{S}{p}$, sendo p o número de processadores como já definido.

Figura 9: m=30,base=ι,Shift-And aproximado



A figura 10 mostra que de acordo com o aumento do número de processadores há um decaimento no tempo linearmente. Esse é o comportamento esperado que comprova o termo p na complexidade de tempo do algoritmo BMHS paralelo.

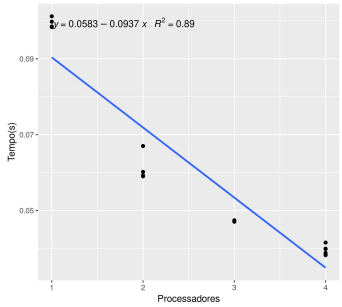
Após isso basta avaliar o speedup obtido com a implementação feita. A figura 12a mostra o speedup na base ι com 4 processadores, esse speedup não foi perfeito porém esta bem próximo, na média sendo $S_4 \approx 3.18$, porém é importante notar que na figura os valores de m pequenos tiveram um speedup menor, já para grandes valores de m o algoritmo se estabilizou numa média um pouco melhor. Porém já para 2 processadores, como mostra na figura 12c, o speedup é na média $S_2 \approx 1.78$.

A figura 12b mostra o speedup de acordo com a base usada. Os resultados obtidos foram parecidos com a variação de m na base ι.

Então de acordo com os dados obtidos a eficiência é $E_4 \approx 0.795$ para 4 processadores e para 2 processadores é $E_2 \approx 0.89$. A eficiência é significativamente maior com 2 processadores como pode ser observado.

Outro fator importante que pode impactar o desempenho do algoritmo é o número de comparações que existe de diferença entre o BMHS paralelo e BMHS serial. A figura 11 mostra que na média o BMHS paralelo tem aproximadamente 24 comparações a mais que o BMHS serial, esse valor é extremamente baixo visto que a média é de até um m igual a 5000. Logo esse valor mostra que o BMHS paralelo não sofre tantas consequências pela sua divisão de blocos. Também é importante notar que existe um crescimento do módulo do número de comparações junto com o crescimento de m.

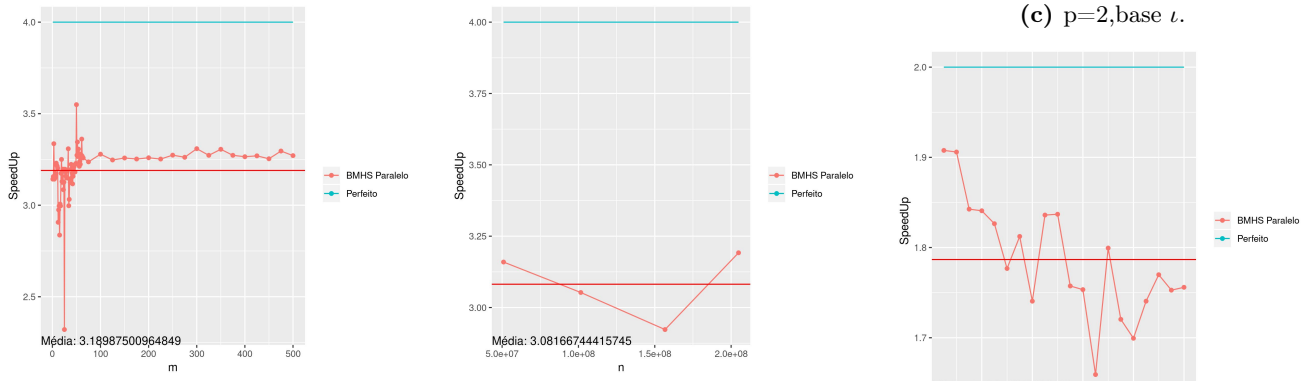
Figura 10: Tempo de acordo com número de processadores.



5.2 Espaço

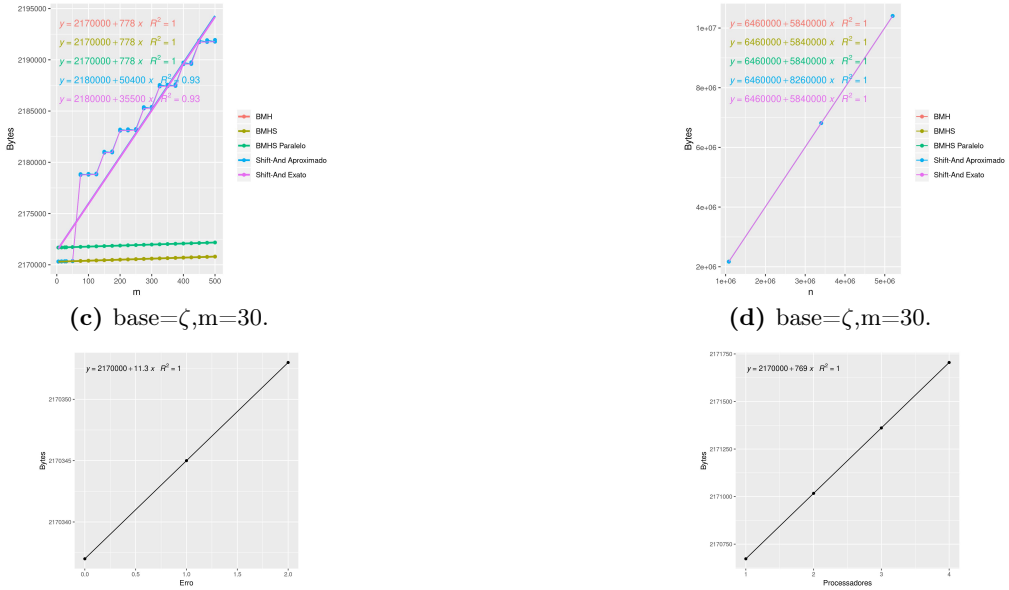
Para provar a complexidade de espaço do algoritmo principal basta monitorar as alocações feitas pelo algoritmo para cada tamanho de entrada. O monitoramento será feito somente da memória *heap* [3] que é a parte da memória onde a alocação dinâmica é feita e o espaço pode-se variar dinamicamente permitindo-se assim obter uma função de complexidade correta.

Na figura 13a é comprovado que m influência na complexidade de espaço dos algoritmos. Também é possível ver o salto que há no uso de memoria do ShiftAnd de acordo com que é necessário alocar blocos de β bits. É possivel observar que o BMH, BMHS e BMHS Paralelo compartilham da mesma função de uso de bytes, isso se deve a eles usarem somente o espaço de alocar o padrão no programa principal.



(a) base=ι,p=4 **(b)** m=30,p=4. **(c)** p=2,base ι.

Figura 12: Speedup.



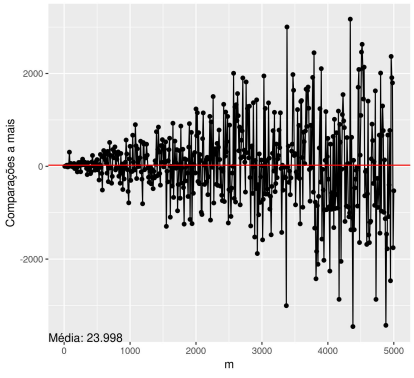
(a) base=ζ. **(b)** m=30. **(c)** base=ζ,m=30. **(d)** base=ζ,m=30.

Figura 13: Espaço utilizado pelos algoritmos em termos de diversos fatores.

A figura 13b mostra também que todos algoritmos somente alocam o texto no programa principal, por isso todos possuem a mesma função em relação a variável n.

As figuras 13c e 13d comprovam o fator de erro e de número de processadores na complexidade de espaço dos algoritmos ShiftAnd e BMHS paralelo respectivamente.

Figura 11: Diferença de comparações BMHS serial e paralelo, base ι.



5.3 Análise geral

Com os algoritmos analisados e mostrados suas eficiências na prática pode-se criar uma hierarquia dos mesmos para cada situação. Veja a figura 14.

Para casos médios o BMHS Paralelo foi o mais rápido dentre todos, porém se for o pior caso esse algoritmo não é tão bom assim porém não é o pior de todos algoritmos.

Em casos médios o BMH é um bom algoritmo, porém não há muitas diferenças dele para o BMHS, em relação a desempenho, somente em casos específicos um supera o outro.

Figura 14: Hierarquia de tempo empírico.

Já o ShiftAnd quando no pior caso ele é consideravelmente mais rápido que os outros algoritmos pois sua complexidade não depende do padrão, porém em casos médios esse algoritmo é extremamente lento comparado aos outros, ainda mais se for Shift-And aproximado.

6 Conclusão

Com esse trabalho foi possível estudar a complexidade e o funcionamento dos algoritmos BMH,BMHS,Shift-And,Shift-And aproximado, BMHS paralelizado. Foi possível ver que o BMH e BMHS com complexidades piores que o Shift-And ainda sim conseguem na prática ser melhor que o Shift-And. A complexidade esperada de $O(n/m)$ do algoritmo BMH e BMHS também foram vistas na prática e suas superioridades em relação ao Shift-And.

Os algoritmos foram classificados baseados nos desempenhos dos resultados, sendo o BMHS Paralelo o melhor algoritmo para casos médios e o Shift-And o melhor algoritmo para o pior caso.

Uma distribuição de blocos de memória foi modelada para o algoritmo BMHS Paralelo essa distribuição de blocos foi analisada de diversas formas, criando diversas funções que descrevem suas características, exemplo: seu tamanho total de blocos, taxa de blocos redundantes e etc. O BMHS paralelo obteve um speedup bom, sendo ele $S_4 \approx 3.18$ com 4 processadores e $S_2 \approx 1.78$ com 2 processadores.

Os algoritmos foram aplicados em genomas e esse estudo pode ser utilizado em outros trabalhos que aplique os algoritmos a outras áreas e as comparem.

Em futuros trabalhos pode-se estudar os efeitos da busca em arquivos comprimidos e todos seus impactos profundamente através dessa base que foi estudada nesse trabalho.

Referências

[1] Bit array. `https://michaeldipperstein.github.io/bitarray.html`. Accessed: 2019-06-12.

[2] Genome information. `https://www.ncbi.nlm.nih.gov/genome/browse/#!/overview/`. Accessed: 2019-06-24.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018.

[4] Thomas Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 2009.

[5] Zvi Galil. On improving the worst case running time of the boyer-moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, 1979.