Universidade Federal de São João del-Rei

Processamento de Cadeias de Caracteres
Algoritmo para detectar plágio em letras musicais

Algoritmos e Estruturas de Dados III

Isabella Vieira Ferreira
Mônica Neli de Resende

1 Introdução

As notas musicais são a base para a construção de uma composição musical. O termo nota musical designa o elemento mínimo de um som, formado pelo modo de vibração do ar, portanto cada nota está associada a uma frequência. Apesar de serem inúmeros os sons (notas musicais) empregados na música, para representá-los bastam apenas sete notas: A (lá), B (si), C (dó), D (ré), E (mi), F (fá) e G (sol). Para se ter uma relação concreta entre os sons, foi necessário um padrão de medida entre as notas musicais, essa unidade de medida é chamada tom. A música ocidental possui um sistema composto por 12 partes, ou seja, 12 sons musicais diferentes. Os sete sons principais são chamados notas naturais, que derivam outros cinco sons, chamados acidentes musicais, através do sustenido e bemol.

Os termos bemol e sustenido são os tipos de derivações que se pode fazer a partir de algumas notas naturais: sustenido ('#') eleva a nota em meio-tom e bemol ('b') reduz a nota em meio-tom. A partir das derivações dos tons é possível escrever uma mesma melodia em outros tons mantendo a distância em meios-tons originais.

O objetivo desse trabalho é implementar um algoritmo que verifique se um determinado trecho ocorre em uma dada música, ou seja, detectando plágios a partir do trecho suspeito. A solução do problema envolve busca de padrões de caracteres conhecido como casamento de cadeias ou casamento de padrão, que consiste em encontrar ocorrências de um padrão em um texto.

Uma cadeia de caracteres é uma sequência formada por um conjunto qualquer de elementos (letras, números etc) denominados caracteres. A pesquisa por um padrão em cadeias de caracteres é um problema que surge com frequência, tais como encontrar todas as ocorrências de uma palavra em um texto, busca de padrões específicos em sequência de DNA, entre diversos outros.

Para solucionar o problema proposto, procuraremos por padrões verificando as distâncias em meios-tons. Para tal utilizaremos quatro estratégias: Força Bruta, Knuth-Morris-Pratt (KMP), Boyer-Moore-Horspool (BMH) e Shift-And Exato, algoritmos clássicos para o casamento exato de cadeias. A partir dos resultados obtidos compararemos qual estratégia é mais adequada para a solução do problema.

A motivação do trabalho deve-se ao interesse em adquirir mais conhecimentos sobre processamento de cadeias de caracteres, bem como utilizar os nossos conhecimentos para desenvolver soluções para este tipo de problema e aplicar a teoria apresentada em sala de aula, reforçando e acrescendo nosso conhecimento a respeito. O desafio é verificar plágio em letras musicais que são situações comuns no dia-a-dia e também um problema encontrado para textos em geral.

2 Cadeia de caracteres

Uma cadeia de caracteres nada mais é do que um conjunto de elementos denominados caracteres, os quais são pertencentes a um conjunto denominado alfabeto. Um alfabeto para uma cadeia de bits, por exemplo, é denotado: $\sum = \{0, 1\}$ e c é o tamanho do alfabeto. No nosso problema o alfabeto é $\sum = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ referente às distâncias entre as sete notas musicais, onde 1 é o limite inferior e 12 o limite superior na soma das distâncias.

2.1 Casamento exato de padrão

Uma tarefa comumente realizada é a busca de todas as ocorrências de uma palavra em um texto. Esse problema é conhecido como casamento de cadeias ou casamento de padrão. Na descrição da solução do nosso problema o texto será as distâncias, em meios-tons, da música de tamanho n e o padrão será as distâncias do trecho suspeito de ter sido plagiado de tamanho m, porém ressaltamos que m deverá ser muito menor que n ($m \ll n$).

Para o nosso problema temos que o ideal é um casamento exato de padrão, ou seja, quando todas as posições do padrão possuem correspondentes no texto. Porém não encontraremos todas as ocorrências do padrão no texto, nos restringindo a encontrar apenas a posição onde ocorre o primeiro casamento da cadeia de caracteres.

3 Entrada e saída de dados

A entrada do programa deverá ser escrita em um arquivo texto composto por um ou vários casos de teste. Para cada caso de teste, a primeira linha deverá conter dois inteiros, o primeiro indicando o número de notas da música e o segundo indicando o número de notas do trecho suspeito de ter sido plagiado, respectivamente. A segunda linha deverá conter as notas musicais, separadas por espaço, sendo que cada nota poderá ser escrita como: 'A', 'B', 'C', 'D', 'E', 'F', 'G', podendo conter '#' (sustenido) ou 'b'(bemol). As notas devem estar necessariamente em letra maiúscula, exceto o bemol que deverá estar em letra minúscula e também não deverá haver espaçamento entre a nota e sua alteração cromática. A terceira linha conterá o trecho de música suspeito de ter sido plagiado, que deverá ser escrito como mencionado acima. A última linha do arquivo, deverá conter "0 0", indicando o final de arquivo. Vale ressaltar que para execução correta do algoritmo, o arquivo de entrada deverá estar escrito exatamente como foi especificado acima. A Figura 1(a) apresenta um exemplo de arquivo de entrada.

O arquivo de saída denominado "saida.txt" conterá, para cada caso de teste, um caractere sendo: "S" caso o trecho tenha sido plagiado, seguido pela posição (de 0 a n-1) onde há a primeira ocorrência na música e "N" caso contrário. Porém, devido à restrição do algoritmo Shift-And Exato, explicado na seção 6, se o tamanho do trecho for maior que o tamanho da palavra do computador (que nas arquiteturas atuais é de 32 ou 64 bits), será impresso no arquivo de saída a seguinte mensagem: "Trecho maior do que a WORD da memoria" e o algoritmo não fará verificações para esse caso de teste. A Figura 1(b) apresenta um exemplo de arquivo de saída, correspondente ao arquivo de entrada.

```
16 4
D G A B C D G G G C D E F# G C C
G G C D
12 2
C C# D D# E F F# G G# A A# B
C D
12 2
C Db D Eb E F Gb G Ab A Bb B
C D
4 3
C E G Bb
D F# A
0 0
```

(a) Entrada

```
S 7
N
N
S 0
```

(b) Saída

Figura 1: Representação da entrada e saída

4 Estrutura de dados utilizada

As notas musicais naturais, ou seja, as sete notas principais está representada pela Figura 2(a) juntamente com o número, em meios-tons, entre elas. Vale ressaltar que as notas podem ter alterações cromáticas, sendo que: '#' (sustenido) altera a nota em meio-tom para cima e 'b' (bemol) altera a nota em meio-tom para baixo.

Para a representação das notas musicas naturais e o número em meios-tons entre elas, utilizamos um vetor cuja célula é representada na Figura 2(b), onde: *nota_musical1* é a primeira nota do intervalo, *nota_musical2* é a segunda nota do intervalo e *distancia* é a distância em meios-tons entre uma nota e outra. Para fins práticos preenchemos o vetor a partir da posição 1, desprezando a primeira posição (0 devido ao padrão da linguagem C). Esse artifício é uma forma de facilitar o modo com que o vetor é percorrido, utilizando aritmética modular, conforme descrito na seção 5.

Para representar a letra musical e o trecho suspeito de ter sido plagiado, lidos do arquivo de entrada, utilizamos dois vetores alocados dinamicamente. Em cada célula do vetor armazenamos as informações, conforme ilustra a Figura 2(c), onde: *nota_musical* é a nota, *alteração_cromática* é a alteração que acompanha a nota, ou seja, 'b' (bemol) ou '#' (sustenido) e *distancia* é a variação em meios-tons entre uma nota e outra, inicialmente vazia.

Notas	A-B	B-C	C-D	D-E	E-F	F-G	G-A
Número de meios-tons	2	1	2	2	1	2	2

(a) Representação das notas musicais

nota_musical1	nota_musical2	distância

(b) Célula para o vetor base

nota_musical	alteração_cromática	distância

(c) Célula para o trecho e música

Figura 2: Representação da Estrutura de Dados

5 Pré-processamento do padrão e do texto

A partir das características do problema percebemos que deveríamos comparar as distâncias em meios-tons ao invés da notas musicais. Isso deve-se ao fato da possibilidade da música iniciar em notas diferentes, porém mantendo a distância em meios-tons representando assim a mesma música (nesse caso dizemos que a música está em outro tom). Se comparássemos as notas musicais não

teríamos o casamento exato pois as notas seriam diferentes. Dessa forma, teremos um casamento exato se a distância contabilizada no padrão ocorrer em alguma parte nas distâncias do texto. Consequentemente é necessário um pré-processamento do texto e do padrão de modo a obter tais distâncias para assim poder aplicar os algoritmos. A solução para contabilizar a distância é descrita abaixo.

Primeiramente alocamos um vetor indexado de 1 a 7 e o preenchemos com as notas naturais, ou seja, as sete notas principais com suas respectivas distâncias em meios-tons, como apresentado na Figura 2(a). O cálculo das distâncias em meios-tons consiste em calculá-las para um par de notas segundo os valores do vetor de notas naturais. A Figura 3 apresenta um exemplo com notas de uma música arbitrária, onde o vetor contém a música e as distâncias calculadas segundo os pares destacados por chaves.

As distâncias são armazenadas a partir da primeira posição, ou seja, a distância entre as notas "A"e "A" é colocada no campo *distancia* da primeira nota. Dessa forma, a cada distância calculada, esta é armazenada no campo *distancia* da célula anterior, portanto, a última nota terá sua distância igual a zero.

Índices	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Notas	**A**	**A**	**D**	**C#**	**C#**	**D**	**E**	**E**	**E**	**F#**	**A**	**D**	**G#**	**A**
Distâncias	0	5	11	0	1	2	0	0	2	3	5	6	1	0

Figura 3: Representação do cálculo das distâncias para uma música arbitrária

Para cada par de notas primeiramente inicializamos o campo *distancia*, com o objetivo de contabilizar a diferença devida à alteração cromática que pode alterar a nota em meio-tom para cima ou para baixo. O campo *distancia* da primeira nota é inicializado para decrementar ou incrementar essa diferença.

Para o caso das notas serem iguais, inclusive a alteração cromática, temos que a distância entre elas é zero. Se as notas se diferenciam apenas pela sua alteração cromática verificamos os seguintes casos especiais:

- Se a primeira for 'b' (bemol) e a segunda '#' (sustenido) então a distância é 2, pois o cálculo se inicia da nota meio-tom abaixo até a nota meio-tom acima, totalizando uma diferença de dois meios-tons;
- Se a primeira for 'b' (bemol) e a segunda não tiver alteração cromática então a distância é 1, pois a diferença é de meio-tom abaixo até a nota, ou seja, meio-tom;
- Se a primeira não tiver alteração cromática e a segunda possuir '#' (sustenido) então a distância é 1, pois aumentou apenas meio-tom;

Na Figura 4 podemos observar os três casos acima descritos, nos exemplos (1), (2) e (3):

Figura 4: Representação das distâncias para os pares: Bb-B# (1), Eb-E (2), F-F# (3)

Se as alterações cromáticas não se encaixarem nos casos acima ou se as notas forem diferentes, então percorre-se o vetor para obter o valor da distância correspondente. O vetor é percorrido primeiramente buscando a posição da primeira nota do par, a partir dessa posição inicia-se a busca pela segunda nota somando-se as distâncias de cada posição que passar. Por fim utilizamos a alteração cromática da segunda nota para completar o cálculo, decrementando ou incrementando um meio-tom.

É importante destacar que para percorrer o vetor não é permitido voltar, se a segunda nota do par não for encontrada a partir da posição da primeira, então continua-se a busca a partir da primeira posição do vetor, somando as distâncias até encontrá-la. Para isso utilizamos a aritmética modular: $i = (i \ mod \ 7) + 1$, onde sete é o tamanho do vetor das notas naturais e i é a posição atual no vetor. A Figura 5 apresenta a forma que o vetor é percorrido para o cálculo do par de notas E-B, onde é necessário voltar ao início do vetor para continuar a busca pela segunda nota, somando-se os meios-tons correspondentes pelo caminho.

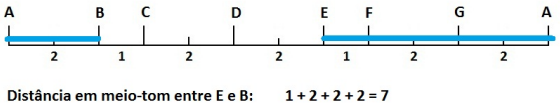


Figura 5: Representação do cálculo da distância para o par de notas E-B

6 Solução apresentada

Para a solução do problema utilizamos quatro estratégias: Força Bruta, Knuth-Morris-Pratt (KMP), Boyer-Moore-Horspool (BMH) e Shift-And Exato, algoritmos clássicos para o casamento exato de cadeias. Os algoritmos foram adequados ao nosso problema que como dito anteriormente não consiste em realizar o casamento entre as notas e sim entre as distâncias (em meios-tons). Para isso, ao invés de comparar os campos *nota_musical* comparamos o campo *distancia*, excluindo a última nota, pois sua distância é zero. Com a finalidade de facilitar o entendimento dos algoritmos, as imagens serão apresentadas com base em exemplos de caracteres e não em distâncias como é tratado.

6.1 Força Bruta

Força Bruta (ou busca exaustiva) é um algoritmo que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema. Para detectar plágio em letras musicais, a idéia do algoritmo consiste em tentar "casar" qualquer subcadeia no texto de comprimento n com o padrão de comprimento m.

6.1.1 Funcionamento do algoritmo

O algoritmo testa todas as posições possíveis do padrão no texto, ou seja, para cada uma, verifica se o padrão "casa" com ela. A Figura 6 apresenta o pseudo-código para algoritmo Força Bruta.

```
Algoritmo Força Bruta

n ⟵ tamanho do texto
m ⟵ tamanho do padrão

Para i = 0 até n - m + 1 faça

    Enquanto (distancia da nota for igual a distancia do
trecho)

            Anda para a direita no texto
            Anda para a direita no padrão

    FimEnquanto

    Se todas as posições no texto foram iguais
        "Casamento de padrão"
    FimSe

FimPara
```

Figura 6: Pseudo - código para o algoritmo Força Bruta

O pior caso da Força Bruta ocorrer é quando, por exemplo: *Texto:* AAAAAAAAAA e *Padrão:* AAB. Isso se deve ao fato do algoritmo analisar todas as posições do texto e obter insucesso na última comparação, onde a distância calculada será diferente.

A Figura 7 ilustra o funcionamento do algoritmo, analisando caractere por caractere. No exemplo temos que: *Texto* = DADABACBADCDACDABACBDCADB e *Padrão* = DABACB. A Figura mostra o decorrer do algoritmo quando $i = 1$, 2, 3, 15 e 20 e o casamento exato ocorre quando $i = 3$ e $i = 15$. Vale ressaltar que no nosso algoritmo, começamos a contar as posições a partir de 0 e será impresso no arquivo de saída somente a primeira posição encontrada, dessa forma, apresentaríamos que o casamento exato será na posição $i = 2$.

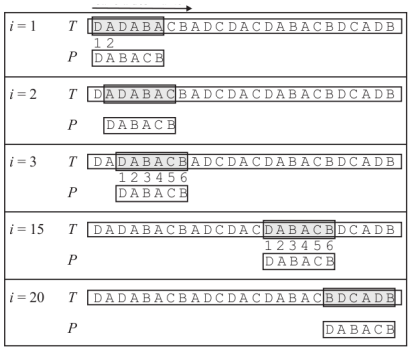


Figura 7: Exemplo de funcionamento do algoritmo Força Bruta

A vantagem do algoritmo Força Bruta está na sua simplicidade, a desvantagem está na busca exaustiva, na inabilidade em reaproveitar o resultado de uma tentativa de casamento e realizar deslocamentos maiores no texto em sua comparação.

6.2 Knuth-Morris-Pratt (KMP)

O algoritmo proposto por Knuth, Morris e Pratt se propõe em corrigir a deficiência do algoritmo Força Bruta. Sua grande característica é que o casamento de padrão é executado em tempo linear ao tamanho do texto, O(n). É um dos algoritmos mais famosos para resolver o problema de casamento de cadeias e pode ser visto como a simulação de um autômato determinista que pesquisa o padrão no texto.

Até 1971 o limite inferior conhecido para busca exata de padrões era O(m.n). Nesse ano Cook provou que qualquer problema que pudesse ser resolvido por um autômato determinista de dois caminhos com memória de pilha (do inglês, *Two-way Deterministic Pushdown Store Automaton* - 2DPDA) pode ser resolvido em tempo linear por uma máquina RAM (*Random Access Machine*).

6.2.1 Funcionamento do algoritmo

A idéia do algoritmo é usar a informação dos caracteres que já foram comparados e tiveram concordância para dar o próximo salto no caso de uma falha na comparação. Se tivermos um casamento de *c* caracteres e o próximo falhar podemos seguramente pular *c* e continuar a busca a partir dessa nova posição. Dessa forma, se houve colisão no caractere na posição *c*, podemos pular *c-1* posições se não houver um prefixo que se repete antes da posição da colisão. Se isso ocorre então podemos pular até *c-1-prefixo_c* caracteres, onde *prefixo_c* é o tamanho do maior prefixo que possui um sufixo antes da posição da colisão.

O algoritmo KMP necessita de duas rotinas, a rotina *iniciar* constrói a tabela de salto calculando qual é o tamanho do maior prefixo que também é sufixo para toda posição do padrão. A Figura 8 mostra um pseudo-código desta rotina.

```
Função iniciar

n ← tamanho do texto
m ← tamanho do padrão
j ← 0
k ← -1
next[0]= -1

Enquanto j < m-1 faça

    Enquanto (k > -1) e (padrão[j] ≠ padrão[k])
             next[k]

    Incrementa j
    Incrementa k

    Se padrão[j] == padrão[k]
       next[j] = next[k]
    Senão next[j] = k

FimEnquanto
```

Figura 8: Função *iniciar* para o Algoritmo KMP

A rotina KMP utiliza a função *iniciar* para buscar o padrão no texto. A Figura 9 apresenta o pseudo-código desta rotina.

```
Algoritmo KMP

n  ← tamanho do texto
m ← tamanho do padrão
j ← 0
i ← 0
next[m-1]

Chamada da função "iniciar"

Enquanto i < n-1 e j < m-1 faça

    Enquanto(j>=0) ou (texto[i] = padrão[j])
             Anda para a direita no texto
             Anda para a direita no padrão


    Senão j = next[j]

FimEnquanto

    Se j = m-1 então "Casamento de padrão"
```

Figura 9: Pseudo - código para o Algoritmo KMP

A Figura 10 apresenta um exemplo que representa o funcionamento do algoritmo para o *Padrão* = 1010011.

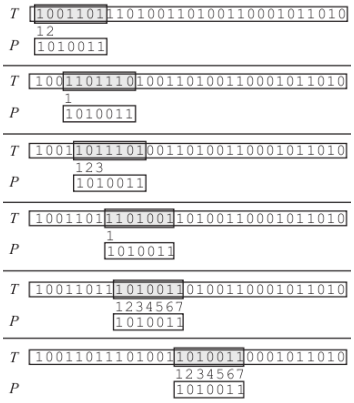


Figura 10: Funcionamento do algoritmo KMP para o padrão 1010011

6.3 Algoritmo Boyer-Moore-Horspool (BMH)

O algoritmo original de Boyer-Moore (BM) tem como idéia principal a comparação do padrão com o texto da direita para a esquerda. O algoritmo decide sobre a utilização de duas heurísticas: heurística ocorrência e heurística casamento. Uma das simplificações sofridas pelo algoritmo foi a utilização apenas da heurística ocorrência, a qual se mostrou mais eficiente. A simplificação deve-se ao fato da decisão impactar em uma perda de desempenho em relação ao tempo de processamento.

A heurística ocorrência consiste em alinhar o caractere do texto que causar uma falha com o primeiro caractere do padrão que combine com o mesmo. No caso deste caractere não estar presente no padrão, pode-se alinhar o padrão imediatamente após o caractere que causou a falha.

A simplificação de Horspool foi utilizar o caractere do texto correspondente ao último deslocamento para endereçar a tabela de deslocamentos, dando origem ao algoritmo Boyer-Moore-Horspool (BMH). Sua grande contribuição deve-se ao cálculo e preenchimento prévios do vetor *skip*, que controla os deslocamentos no texto. O vetor *skip* armazena para cada caractere do alfabeto (o conjunto de caracteres usado no padrão e no texto) um valor para o deslocamento.

O algoritmo depende do conhecimento prévio do alfabeto do problema, sendo este $\sum = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, logo a condição é facilmente satisfeita.

6.3.1 Funcionamento do algoritmo

Primeiramente são inseridos em todas as entradas do vetor *skip* o tamanho do padrão, em seguida para os m-1 caracteres correspondentes do padrão são calculados os deslocamentos segundo a função:

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

Onde P representa o padrão, d representa o vetor *skip* e m é o tamanho do padrão.

A Figura 11 apresenta os valores dos deslocamentos para os m-1 caracteres do padrão segundo a função acima. Os demais caracteres do alfabeto, assim como o valor de 'A', tem o deslocamento de valor 4, ou seja, o valor correspondente ao tamanho do padrão como inicializado anteriormente. O deslocamento de 'B' é calculado duas vezes, portanto seu valor é 1, substituindo o valor calculado para 'B' na posição 1 do vetor.

1	2	3	4	⇐ Índices
B	**C**	**B**	**A**	⇐ Padrão
3	2	1	4	⇐ Deslocamentos

Figura 11: Exemplo de cálculo dos deslocamentos para o padrao "BCBA"

O valor do deslocamento nos diz quanto o padrão deve andar para emparelhar o i-ésimo caractere no texto, referente ao último deslocamento, como um possível caractere igual no padrão. Se este i-ésimo caractere nem sequer estiver presente no padrão, o algoritmo andará m caracteres. Para adequar o algoritmo ao nosso problema esse procedimento é realizado endereçando o vetor *skip* com as distâncias.

O segundo passo é emparelhar o padrão com a posição m no texto, em seguida o procedimento é iniciado e as comparações são realizadas até o último caractere do texto. A cada deslocamento é verificado se a partir da posição inicial (mais à direita) para a esquerda tem um casamento de padrão, que ocorre se todas as posições do padrão forem iguais às respectivas posições no texto. Por fim é realizado o deslocamento utilizando o valor do vetor *skip*, endereçado com a distância correspondente ao último deslocamento e repetindo o procedimento. A Figura 12 apresenta o pseudo-código do algoritmo.

```
Algoritmo Boyer-Moore-Horspool

n ← tamanho do texto
m ← tamanho do padrão
d ← Função de avanço
Emparelha o padrão com o texto a partir da posição m para a esquerda

Enquanto não terminar o texto

    Enquanto houver casamento de posições e não terminar o padrão

            Anda para a esquerda no texto
            Anda para a esquerda no padrão

    FimEnquanto

    Se todas as posições no texto foram iguais "Casamento de padrão"
    Faz o deslocamento d e emparelha o padrão com o texto a partir dela

FimEnquanto
```

Figura 12: Pseudo-código do algoritmo Boyer-Moore-Horspool

A Figura 13 apresenta o funcionamento do algoritmo para o texto e padrão mostrados. As setas indicam as posições as quais se deram os deslocamentos e as células marcadas de cor laranja representam as posições no padrão em que ocorrem as falhas. O casamento de padrão ocorre na posição 12 no texto. Ressaltamos que o algoritmo conta as posições a partir de zero, dessa forma, apresentaremos que o casamento ocorre na posição 11.

Algoritmo Boyer-Moore-Horspool

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
X	**C**	**B**	**A**	**B**	**X**	**C**	**B**	**A**	**A**	**X**	**B**	**C**	**B**	**A**	**B**	**X**
B	C	B	A													
			↑	B	C	B	A									
					B	C	B	A								
							↑	↑	B	C	B	A				
											B	**C**	**B**	**A**		
												↑		↑		

Figura 13: Representação dos deslocamentos realizados para a verificação se o padrão ocorre no texto

6.4 Algoritmo Shift-And Exato (SA)

O algoritmo Shift-And desenvolvido por Baeza-Yates e Gonnet realiza a busca através do paralelismo de bits, uma técnica que utiliza as operações sobre bits em uma palavra do computador, que nas arquiteturas atuais é 32 ou 64 bits. O desempenho do algoritmo depende do custo das operações *or*, *and* e deslocamentos de bits as quais quando utilizadas dentro de uma palavra do computador possui custo constante se utilizado um compilador que as implemente tão eficientemente quanto na linguagem C.

O algoritmo utiliza um autômato finito não determinista (quando existe uma transição para mais de um estado) para representar uma transição de estados sempre quando um determinado caractere do padrão é encontrado no texto. Um autômato R representando um padrão p de tamanho m possui $m+1$ estados. Quando o estado final R_{m+1} é alcançado, então uma ocorrência do padrão foi encontrada no texto. A Figura 14 apresenta um autômato não determinista para o reconhecimento do padrão "teste". O estado colorido indica o estado inicial e o estado com círculos concêntricos indica o estado final, ou seja, quando se confirma uma ocorrência do padrão.

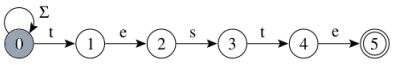


Figura 14: Autômato não determinista para o reconhecimento do padrão "teste"

Para isso o algoritmo utiliza uma variável para simular os estados do autômato. As transições de estados são realizadas utilizando-se uma máscara de bit para cada caractere do alfabeto, ou seja, $\sum = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$. Tal verificação é realizada a partir das operações bit a bit entre a variável que representa o autômato e a máscara do caractere atual do texto. A máscara é limitada pela palavra do computador, pois representa para cada caractere a posição que ele ocupa no padrão, tal como apresentado na Figura 15. Sendo assim, o tamanho do padrão não deve exceder o tamanho da palavra do computador utilizado. Uma solução para essa limitação é a implementação independente das operações de bits, ressaltando que o desempenho do algoritmo é dependente do custo dessas operações. Porém utilizamos em nossa implementação as vantagens da utilização da técnica paralelismo de bits.

Máscara de bits					
	t	e	s	t	e
M[t]	1	0	0	1	0
M[e]	0	1	0	0	1
M[s]	0	0	1	0	0

Figura 15: Representação para a máscara de bits para a palavra "teste"

6.4.1 Funcionamento do algoritmo

Inicialmente o algoritmo inicializa as máscaras de bit com zero. Em seguida, para cada caractere do padrão define-se a máscara de bit que dirá em quais posições o caractere ocorre nele. Inicia-se então a varredura do texto e ao ler um caractere sua máscara de bit correspondente é selecionada e realizada a operação:

$$\mathrm{R} = ((\mathrm{R} \gg 1) \mid 10^{m-1}) \ \& \ \mathrm{M}[\mathrm{T}[i]]$$

Onde R representa o autômato, o operador $\gg$ realiza um deslocamento dos bits para a direita e completando com zeros à esquerda, 10^{m-1} é uma notação para denotar que zero repete m-1 vezes (por exemplo, $10^3 = 1000$) e M representa a máscara de bits.

A operação tem o objetivo de realizar uma transição quando necessário. Um casamento de padrão ocorre quando obtivermos o último bit da máscara R ativado. O funcionamento descrito é ilustrado no pseudo-código na Figura 16.

```
Algoritmo ShiftAndExato

M ← Função Máscara de bits
R ← 0^m                          // Variável representando o autômato

 Para cada caractere do texto T[i]
   R ← ((R << 1) ∧ 10^(m-1)) ∨ M[T[i]]    // Transição
   Se o último bit de R for igual a 1 então "O padrão ocorreu"

 FimPara
FimAlgoritmo
```

Figura 16: Pseudo-código do algoritmo ShiftAndExato para casamento exato

A Figura 17 mostra as transições ocorridas para cada caractere do texto "o teste". A operação *or* na terceira coluna tem o objetivo de verificar a cada iteração se começa uma nova possibilidade de casamento de padrão, além de sempre permitir o início de um novo casamento. Devido a isso o primeiro bit do estado é sempre 1. O bit marcado de vermelho na quinta coluna indica um casamento de padrão ocorrido e o bit 1 mais a esquerda indica uma possibilidade de casamento. Ao encontrar uma distância que não pertença ao padrão o estado de R é reinicializado, ou seja, voltando ao estado inicial e cancelando qualquer possibilidade de casamento anterior.

		R = (R ≫ 1) | (10^{m-1}) & M[caractere]		
Caracteres	R	(R ≫ 1) | (10^{m-1})	M[caractere]	R
o	00000	**1**0000	00000	00000
	00000	**1**0000	00000	00000
t	00000	**1**0000	10010	10000
e	10000	**1**1000	01001	01000
s	01000	**1**0100	00100	00100
t	00100	**1**0010	10010	10010
e	10010	**1**1001	01001	01001

Figura 17: Exemplo de verificação para uma transição

7 Análise de Complexidade

Nesta seção faremos uma análise da complexidade de tempo de cada função, em seguida calcularemos a complexidade total a partir da análise da função principal. Analisaremos também o número de comparações das quatro estratégias. Ressaltamos que n é o número de notas da música e m o número de notas do trecho suspeito, as complexidades serão calculadas para o pior caso. Os cálculos e atribuições serão considerados com complexidade constante.

1. **void leitura (FILE *entrada, musica *notas, musica *trecho)**
   Essa função é responsável por realizar a leitura do arquivo. Nela temos a alocação dinâmica de dois vetores, o primeiro de tamanho n e o segundo de tamanho m. A medida que o arquivo é lido são feitas inserções dos valores no vetor. Dessa forma, a complexidade de tempo para essa função é O(m) + O(n) = O(max(m+n)) = O(m+n).
   Complexidade de tempo: O(m+n)

2. **int forca_bruta (musica nota, musica trecho)**

   No algoritmo força bruta temos um laço externo variando de 0 a n-m+1, complexidade O(n-m+1). Temos um laço interno, que no pior caso percorrerá todo o padrão, complexidade O(m). Dessa forma, temos que a complexidade final da função é O(n-m+1).O(m) = O(n.m -m^2 + m). Como m é muito menor que n então temos que a complexidade de tempo é O(n.m).

   Complexidade de tempo: O(n.m).
   Número de comparações: n.m -m^2 + m comparações.

3. **int kmp (musica nota, musica trecho)**

   O algoritmo KMP chama a função *iniciar* que possui um laço percorrendo todo o trecho, logo a complexidade é O(m).

   No KMP há um laço percorrendo o texto, que no pior caso a complexidade é O(n), quando não há aproveitamento dos casamentos entre posições anteriores resultando em saltos de apenas 1. Dessa forma, temos que a complexidade dessa rotina será O(m) + O(n) = O(max(m+n)) = O(m+n). Como m é muito menor que n então temos que a complexidade de tempo é O(n)

   Complexidade de tempo: O(n).
   Número de comparações: m + n comparações.

4. **int BMH (musica *nota, musica *trecho)**

   No algoritmo BMH o pré-processamento consiste na inicialização do vetor *skip* de tamanho c, onde c é o tamanho do alfabeto, complexidade O(c) e o cálculo para os deslocamentos do trecho, resultando em O(m). Logo o pré-processamento possui complexidade O(c+m).

   A busca pelo trecho é executada dentro de um *loop* aninhado, a qual o *loop* interno é executado até que haja uma falha ou um casamento e o *loop* externo executa n-m+1 vezes, ou seja, até o primeiro caractere da última possibilidade de ocorrência na música. O *loop* interno executará m vezes, resultando em O(m).O(n-m+1) = (m.n-m^2 + m). Porém como o tamanho do trecho é muito menor que o tamanho da música, a complexidade se resume em O(n.m) para o pior caso. Com base nessas informações temos que a complexidade para o pré-processamento e a busca é O(c+m) + O(n.m) = O(max((c+m), (n.m))) o que nos dá O(n.m).

   Complexidade de tempo: O(n.m).
   Número de comparações: m.n-m^2 + m comparações.

5. **int ShiftAndExato (musica *nota, musica *trecho)**

   O pré-processamento consiste na inicialização do vetor de máscaras de caracteres cujo tamanho é c, onde c é o tamanho do alfabeto, resultando em complexidade O(c). Em seguida temos o cálculo para as máscaras dos caracteres do trecho, resultando em O(m). Logo o pré-processamento possui complexidade O(c+m).

   A busca é linear em relação ao tamanho da música, contendo apenas as operações *and*, *or* e deslocamento de bits. Portanto a complexidade do algoritmo Shift-And Exato é O(n) para o pior caso, ou seja, quando não há ocorrência do trecho.

   Em posse disso temos O(c+m) + O(n) = O(n), pois o tamanho da música é muito maior que o tamanho do trecho.

   Complexidade de tempo: O(n).
   Número de comparações: n comparações.

6. **int main (int argc, char *argv[])**

   Na *main* para cada estratégia temos as seguintes complexidades:

   Primeiramente é chamada a função *PreencheVetorBase*, complexidade O(8). Em seguida a função *Processamento* é chamada para os vetores música e trecho com complexidades O(7.n) e O(7.m), onde 7 é o número de posições do vetor base que serão percorridas no pior caso. São chamadas a função específica de cada estratégia e por fim as funções *destruir_vetor_notas_naturais* O(1), *destruir_vetor_musica* 2.O(1) = O(1) e *imprimir* O(1).

   Força Bruta

   Complexidade: O(8) + O(7.n) + O(7.m) + O(n.m) + O(1) + O(1) + O(1) = O(max(8 + (7.n) + (7.m)+ (n.m) + 1 + 1 + 1) = O(n.m).

   Complexidade de tempo: O(max((7.n), (n.m))).

   KMP

   O(8) + O(7.n) + O(7.m) + O(n) + O(1) + O(1)+ O(1) = O(max(8 + (7.n) + (7.m)+ (n) + 1 + 1 + 1) = O(7.n).

   Complexidade de tempo: O(7.n).

   BMH

   O(8) + O(7.n) + O(7.m) + O(n.m)) + O(1) + O(1)+ O(1) = O(max(8 + (7.n) + (7.m)+ (n.m) + 1 + 1 + 1) = O(max((7.n), (n.m))).

   Complexidade de tempo: O(max((7.n), (n.m))).

ShiftAndExato
O(8) + O(7.n) + O(7.m) + O(n) + O(1) + O(1) + O(1) = O(max(8 + (7.n) + (7.m)+ (n) + 1 + 1 + 1) = O(max((7.n), n)) = O(7.n).
Complexidade de tempo: O(7.n).

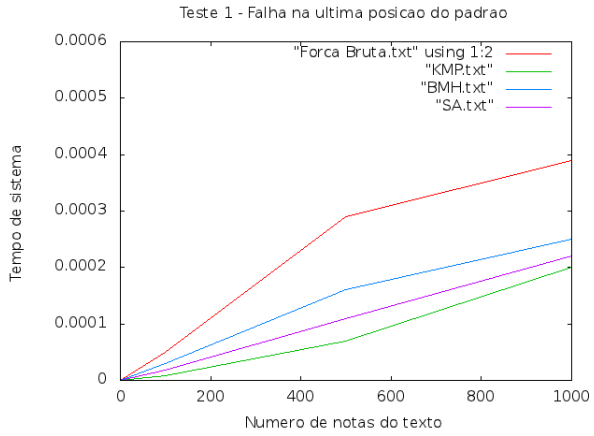
8 Testes e resultados

Os algoritmos serão analisados segundo os critérios tempo de execução e desempenho em função do número de comparações. Para tal utilizaremos respectivamente a função *gettimeofday* e uma variável global para a contagem das comparações. Tais critérios serão aplicados variando o tamanho da música e do padrão para os casos descritos abaixo. O algoritmo será executado até o final da música, onde se encontrará o primeiro casamento da cadeia de caracteres. Para a análise segundo o critério tempo, será considerado apenas o tempo de sistema.

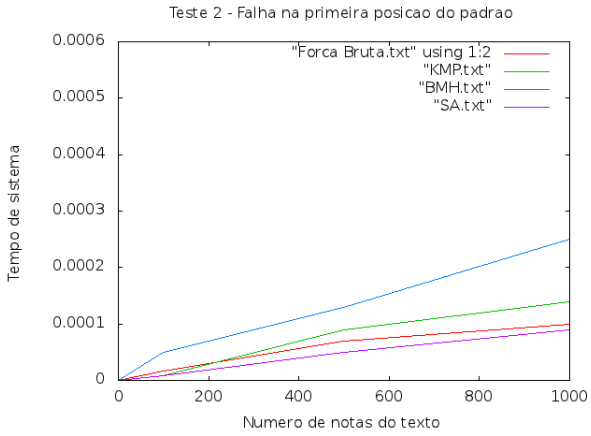
Vale ressaltar que foram realizados 50 testes e utilizamos a média para a realização das análises em cada um dos três casos que descreveremos a seguir . As Figuras 18 e 19 apresentam três gráficos, os quais foram avaliados:

(a) Teste 1: Padrão diferencia pela última nota no texto (*Ex:* Texto: AAAAA e Padrão: AAB);
(b) Teste 2: Falha na primeira ocorrência do padrão no texto (*Ex:* Texto: AAAAA e Padrão: BAA);
(c) Teste 3: Texto repetitivo (*Ex:* Texto: AGBDCEFAGBDCEFAGBDCEF e Padrão: AGBDCEF).

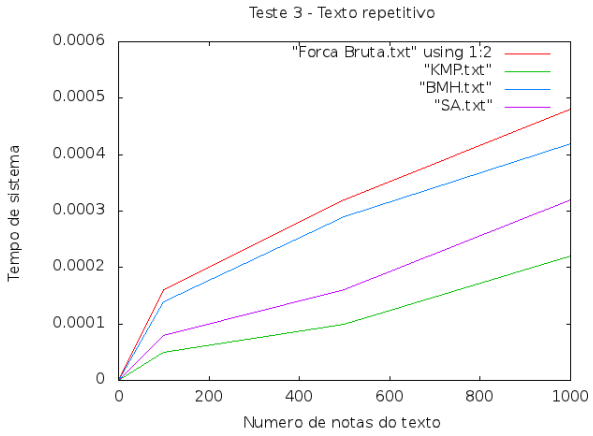
- **Testes com relação ao tempo de execução de cada algoritmo**



(a) Teste 1



(b) Teste 2



(c) Teste 3

Figura 18: Gráficos de Tempo para os três casos de teste

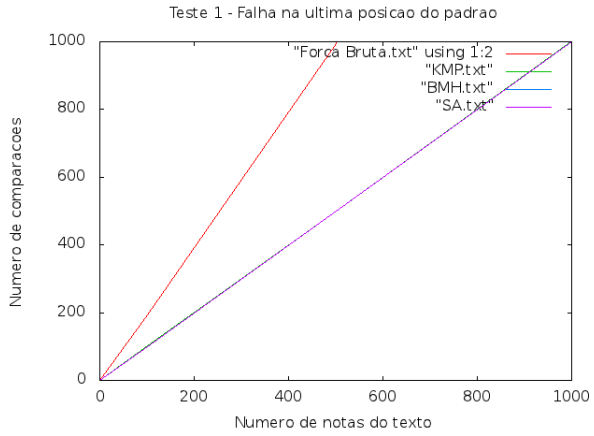
De acordo com os resultados do teste 1, podemos observar que o comportamento dos algoritmos KMP, SA e BMH foram lineares ao tamanho do texto. Em relação ao KMP o comportamento deve-se ao aproveitamento dos casamentos das posições anteriores,

realizando deslocamentos maiores. No caso do BMH apesar da falha acontecer na primeira comparação (da direita para esquerda) o deslocamento é de apenas 1 consequente da repetição do padrão(AAB), por isso é linear. Já o SA é sempre linear devido à utilização da técnica paralelismo de bits. O Força Bruta teve um desempenho ruim devido ao fato desse ser o seu pior caso, onde a falha ocorre na última comparação e por não haver um deslocamento eficiente.

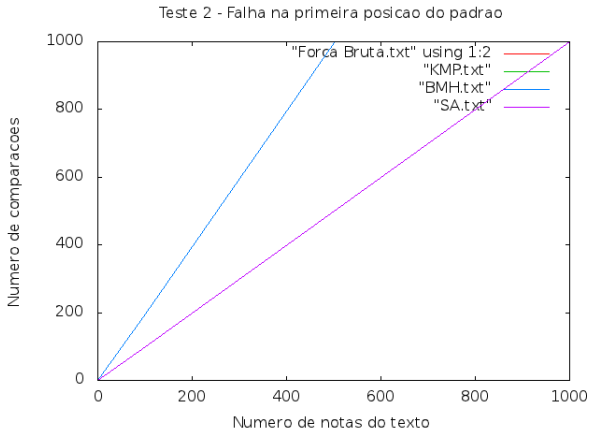
No teste 2 ocorre o inverso do teste 1, sendo que o padrão se diferencia do texto na primeira posição (da esquerda para a direita). Fato este que implica no mal desempenho do BMH, sendo que a falha ocorre na última comparação e o deslocamento ser de apenas um, devido à repetição do padrão. No algoritmo Força Bruta, a falha ocorre na primeira comparação, sendo portanto linear em relação ao tamanho do texto. O mesmo ocorre com o KMP e como não há casamento em posições anteriores o deslocamento também é de apenas 1.

No teste 3 analisamos o comportamento dos algoritmos dados o texto e o padrão com distâncias repetidas e ocorrendo casamentos em todas as posições do texto. Nesse caso o algoritmo KMP teve um melhor desempenho em relação aos outros consequente do aproveitamento dos casamentos das posições anteriores. Já os algoritmos BMH e Força Bruta não tiveram um bom desempenho devido aos casamentos e aos deslocamentos que após todas as comparações do padrão é de apenas 1 (por não haver falha).

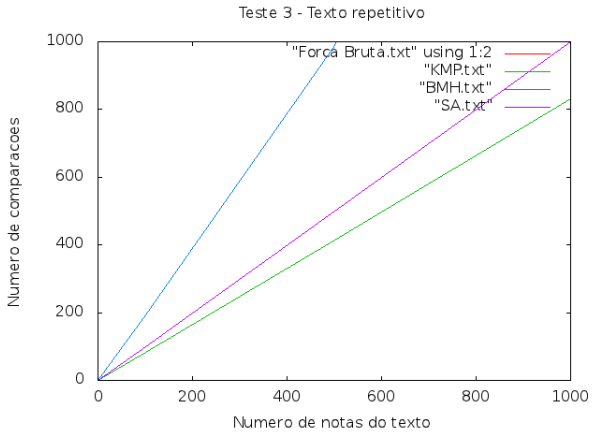
- **Testes com relação ao número de comparações realizadas**



(a) Teste 1



(b) Teste 2



(c) Teste 3

Figura 19: Gráficos do número de comparações para os três casos de teste

Nos gráficos acima podemos observar que o número de comparações no pior caso do algoritmo Força Bruta e BMH é de aproximadamente $m.n$, respectivamente nos gráficos dos testes 1 e 2, sendo que os outros algoritmos foram lineares. No teste 3 observamos que o Força Bruta e o BMH tiveram o mesmo comportamento, sobrepondo-se na curva do gráfico, confirmando os resultados apresentados na seção 7. O KMP e o SA são lineares em relação ao texto, ou seja, realiza n comparações.

8.1 Análise Final

De acordo com os resultados obtidos concluímos que no caso em que os caracteres do texto são repetidos no padrão, há uma maior eficiência por parte do algoritmo KMP, porém não é útil na prática. Já o algoritmo Força Bruta teve um desempenho pior na maioria dos casos de teste, isso se deve ao fato de o algoritmo analisar todas as possibilidades do padrão no texto. Porém, a busca por um padrão em um texto não necessita que sejam testados todas as possibilidades, pois existem heurísticas que podem ser utilizadas na realização de deslocamentos mais eficientes. O algoritmo BMH é um dos algoritmos que calcula tais deslocamentos de forma eficiente de modo que tenha um bom desempenho na prática. Já o algoritmo ShiftAndExato se manteve constante em todos os casos de teste, sendo este o mais eficiente para as diversas possibilidades de entrada, isso se deve ao fato de o algoritmo utilizar o paralelismo de bits, como explicado na Seção 6.

9 Conclusão

Este trabalho nos proporcionou um conhecimento à respeito do processamento de cadeias de caracteres. Com ele pudemos consolidar o desenvolvimento e projeto de soluções para problemas computacionais.

Os resultados apontaram que a estratégia mais indicada para a solução do problema proposto é o Boyer-Moore-Horspool devido ao seu cálculo eficiente de deslocamentos. O Shift-And Exato é também uma estratégia indicada, pois é linear em relação ao texto, sendo este melhor que o BMH. Contudo, o tamanho do padrão é restrito à arquitetura do computador, o que torna a escolha do BMH a mais adequada.

Como trabalhos futuros propomos a verificação de plágio em algoritmos. A principal dificuldade enfrentada neste trabalho foi o entendimento para a implementação da estratégia KMP.

Por fim, atingimos o objetivo deste trabalho, que era desenvolver quatro soluções distintas para detectar plágio em letras musicais identificando a melhor solução e aperfeiçoando a nossa capacidade de análise e projeto de algoritmos.

10 Referências Bibliográficas

[1] Ziviani, Nívio. Projetos de Algoritmos com implementações em Pascal e C, 2011.

[2] Cormem, Thomas H. [et al] - Algoritmos - tradução da 2ª edição americana, 2002.

[3] Kernighan, Brian W., Ritchie, Dennis M. C - A linguagem de programação padrão ANSI C, 1989.

[4] Rocha, Leonardo Chaves Dutra da - Processamento de cadeias de caracteres, 2012

[5] R. Baeza-Yates e B. Ribeiro-Neto. Modern Information Retrieval. Addison Wesley, 1999 - Capítulo 8, Seção 8.5