Uma comparação de algoritmos de busca e alinhamento

Gabriel Oliveira Moura Lima
Vinícius de Jesus Mendes Rodrigues

30 de novembro de 2020

Motivação

Em 2020 a pandemia da COVID-19 resultou em crises econômicas ao redor do mundo, onde muitos perderam suas fontes de renda. Para remediar a situação, países começaram a liberar auxílio financeiro emergencial, com o objetivo de suprir essas perdas. No Brasil, as pessoas que receberam esse auxílio emergencial foram cadastradas em um banco de dados que poderia ser acessado via aplicativo móvel, por nome Caixa Auxílio Emergencial. No entanto, essa solução revelou dificuldades para pesquisar um número grande de beneficiados, devido à demora no resultado da busca, o que a tornava ineficiente. Diante dessa dificuldade, este trabalho propõe três algoritmos de pesquisa que podem resultar em uma busca mais eficiente, e que serão testados para propor o mais adequado para a pesquisa no padrão .CSV, um tipo de arquivo em que os valores de cada registro são separados por vírgulas, sendo esse padrão o único disponibilizado pelo Portal da Transparência do Brasil.

Abstratct

In 2020 the outbreak of the COVID-19 has resulted in a economic crisis, where a big part of the populace has lost their incomes, to remedy this situation, countries started to distribute economic help to solve this income problem. In Brazil, people who received this financial help were registered in a database which was acessible using a mobile application, named "CAIXA Auxilio emergencial", but it has been proven especially hard to find a big number of benefited, because of the delay in the search process. This scientific paper proposes that using a specific method of search could be more efficient, as such the scientific paper will realize test between algorithms, to determine the most efficient

between them at searching the data base with the .csv format as provided by Brazil's "Portal da Tranparência".

1 Introdução

Com o surgimento da pandemia da COVID-19, tornou-se mais claro que o sistema utilizado pelo governo pode e deve ser mais eficiente na prestação de serviços aos 209 milhões de brasileiros. Como solução para essa necessidade, este trabalho apresenta propostas de algoritmos eficientes para busca em base de dados, como o de Boyer-Moore, Smith-Waterman, uma das formas de encontrar o Longest Common Subsequence, e Knuth-Morris-Pratt (KMP). Tais algoritmos podem ser utilizados em outros serviços que disponibilizam dados no mesmo padrão utilizado pelo Portal da Transparência, tornando a divulgação mais ágil para os brasileiros e agentes de fiscalização.

2 Objetivos

2.1 Objetivos Gerais

O objetivo deste trabalho é observar a funcionalidade de algoritmos de pesquisa em um banco de dados local no formato CSV, e no mesmo host onde se encontra o aplicativo do algoritmo, isto é sem utilizar de conexões a rede, a não ser por baixar o arquivo CSV, bem como apresentar seus resultados. Os algoritmos são explicado e os resultados apresentados em razão de suas lógicas de funcionamento. Os resultados finais de eficiência dos algoritmos são analisados com o intuito de criar informações que podem ser reutilizadas de diversas outras formas, sejam no âmbito da computação, biologia ou qualquer área que envolvam soluções por algoritmos.

2.2 Objetivos Específicos

Serão utilizados três algoritmos de pequisa para percorrer o banco de dados no formato CSV, sendo eles os algoritmos de Knuth-Morris-Pratt, Boyer-Moore, e Smith-Waterman, onde os dois primeiros usam de saltos para percorrer o texto mais rápido, não se prendendo a alinhamentos que já se provaram não serem os procurados pelo algoritmo, e o terceiro que utiliza o problema Longuest Common Subsequence e uma comparação de matrizes para achar estes alinhamentos, que mais se parecem com aqueles que são procurados. O banco de dados está estruturado da seguinte maneira: MÊS DISPONIBILIZAÇÃO, UF, CÓDIGO MUNICÍPIO IBGE, NOME MUNICÍPIO, NIS BENEFICIÁRIO, CPF BENEFICIÁRIO, NOME BENEFICIÁRIO, NIS RESPONSÁVEL, CPF RESPONSÁVEL, NOME RESPONSÁVEL, ENQUADRAMENTO, PARCELA, OBSERVAÇÃO, VALOR BENEFÍCIO. Os dados dos testes serão armazenados em diversos arquivos CSV, os quais, após analisados, serão disponibilizados em formato de gráfico. Os algoritmos serão acessados a partir de um menu com as

seguintes opções para a pesquisa: NOME BENEFICIÁRIO, DATA DO RECEBIMENTO DO AUXÍLIO, e o ALGORITMO a ser usado para a pesquisa. Essas opções ativarão a função que irá percorrer o banco de dados no padrão CSV, em busca de uma "substring" em uma "string", e que corresponda à entrada do usuário.

3 Referencial Teórico

3.1 Linguagem C

3.1.1 História

A linguagem C foi desenvolvida por Dennis Ritcher em 1972, baseando-se em uma linguagem antecessora chamada "B", criada por Ken Thompson na Bell Labs. A linguagem B tinha problemas que levaram Ritcher a criar uma linguagem diferente. O objetivo inicial da linguagem C era reescrever a maior parte do sistema operacional Unix, criado pelos mesmos. Mesmo sendo uma linguagem mais conhecida mundialmente, o C não teve sucesso inicial, que veio somente após o lançamento do The C Programming Language (conhecido por muitos como K&R C), livro instrucional linguagem.

Até 1983 não existia um padrão definido para o C, o que levou o Instituto Nacional Americano de Padrões a reunir um comitê para criar uma especificação da linguagem. Este processo levou 6 anos para ser concluído, e a especificação foi nomeada ANSI X3.159-1989 "Programming Language C", mais comumente chamado ANSI C ou C89. Em 1990, a Organização Internacional de Normalização (ISO) o adotou como ISO/IEC 9899:1990, também por C90. Um dos objetivos da revisão da ISO foi adicionar novos recursos (alguns do C++), conjuntos de letras diferentes do inglês e aprimoramento no pré-processamento.

Após 9 anos, houve uma nova revisão da linguagem, que apresentou recursos como novas biblioteca e suas funções, vetores de comprimento variável, novos tipos de data e suporte para o Unicode. A ISO publicou em 1999 a norma ISO 9899:1999 ou C99, sendo essa versão fortemente compatível com o C90.

Em 2011 a ISO publicou o ISO/IEC 9899:2011 ou C11, e essa norma trouxe consigo operações atômicas, multi-threading, melhora no suporte ao Unicode e aprimorada compatibilidade com C++. Alguns recursos foram mantidos como opcionais. Atualmente a linguagem C encontra-se na versão C18, especificada em junho de 2018, mas que não trouxe adição de novos recursos, mas corrigiu defeitos existentes na norma anterior.

3.1.2 Compilação

Os padrões de compilação de código em C são definidos pela mudança de código-fonte para código de objeto, normalmente seguindo em uma linha de transformações antes de executar o código. O processo de compilação da linguagem C normalmente é divido em 4 partes, sendo estas pre-processamento,

compilação, “assembler”, e “linker”. O código-fonte é compilado na forma vertical, linha por linha de forma sequencial. O código-fonte é enviado ao processador, que cria um código expandido a partir do código-fonte. Normalmente, em compiladores GCC, este código expandido tem sua extensão nomeada como I. O código expandido é enviado envia ao compilador, que irá tratá-lo em duas fases, sendo estas a de análise, que verifica a semântica e sintaxe, e a fase da síntese, que o otimiza e cria um código “assembly”, com extensão de arquivo S. O código “assembly” é enviado ao montador (assembler), e este, por sua vez, cria um código objeto com extensão de arquivo O ou OBJ. O código objeto é enviado ao ligador (linker), que ira utilizar bibliotecas pre-compiladas para combinar o código objeto dessas junto ao código do programa, ligando os dois e retornando um padrão de arquivo executável com extensão .exe, para o sistema DOS, ou .OUT, para o sistema UNIX.

3.1.3 Uso no cotidiano

A linguagem C é bastante utilizada nos sistemas operacionais, “softwares” e “hardwares”, tanto pela sua portabilidade, podendo ser executada em inúmeros sistemas operacionais, quanto pela sua simplicidade. Sendo uma linguagem de alto nível, mas bastante semelhante à linguagem comum, é de fácil entendimento. Sua simplicidade, no entanto, não esconde os recursos normalmente encontrados em linguagens de baixo nível, e beneficia-se de poderio semelhante. Bastante utilizada em projetos de “hardware” e robótica, pois é eficiente para execução de instruções de forma simples e rápida, não se mostra eficiente para “interfaces”. Muito além do uso em protótipos, C é ponto de partida para o entendimento de outras linguagens que se desenvolveram tendo ela por base. Na atualidade, sistemas operacionais, “softwares”, e mesmo máquinas, microcontroladores, celulares e “hardware” diversos tem suas funcionalidades programadas utilizando recursos da linguagem C.

3.2 Algoritmo de Knuth-Morris-Pratt(KMP)

3.2.1 História

O algoritmo Knuth-Morris-Pratt (KMP) foi desenvolvido em 1977 por Donald Knuth, James H. Morris e Vaughan Pratt, aprimorando um algoritmo já existente chamado Morris-Pratt. Na época do desenvolvimento do algoritmo, havia um problema recorrente ao procurar uma palavra em um texto, pois a solução inicial era verificar todas as posições do texto entre 0 e o tamanho do texto(m) menos o tamanho da palavra(n), e após cada verificação a “posição” da palavra nesse texto seria alterada em 1 para a direita, até encontrar a palavra desejada. O algoritmo Morris-Pratt ficou conhecido como algoritmo de Força Bruta, sendo bastante ineficaz para buscas em um grande bancos de dados. Morris começou a aprimorar o algoritmo em 1970, junto ao Pratt, quando foi publicado o relatório técnico. Em 1977 Knuth, Morris e Pratt publicaram o algoritmo final, criando assim o algoritmo Knuth-Morris-Pratt.

3.2.2 O algoritmo

O objetivo do algoritmo KMP é aprimorar a busca de um padrão com tamanho n dentro de um texto com tamanho m onde m é maior que n, e, diferentemente do algoritmo de Força Bruta, no momento em que ocorre uma discrepância entre o carácter no índice j do padrão e o carácter no índice i do texto, ele consultará uma tabela previamente criada se esse carácter existe em uma posição anterior desse mesmo padrão, assim, o j irá retornar ao índice que foi encontrado em *next[j]*, e, caso exista outra discrepância o algoritmo irá repetir essa ação até que o j atinja *0*, ou seja, írá repetir até que se chegue ao início do padrão, o cálculo para esses "pulos" é dado por j - *next[j]*. Desta forma o KMP encontra uma solução para o maior problema do algoritmo de Força Bruta(Naive algorimth). Essa busca tem uma complexidade de *O(m+n)*, pois percorre o texto apenas uma vez, e para que isso seja possível é necessário um pré-processamento na palavra.

Para explicar melhor esse pré-processamento será utilizado o exemplo do artigo original do algoritmo de Knuth-Morris-Pratt(Knuth D, Morris J H, Pratt V 1977 Fast pattern matching in strings SIAM J. Comput. 6 323–350, supondo que um padrão *abcabcacab* será procurado no texto *babcbabcabcaabcabcabcacabc*. Primeiramente, esse padrão irá ser preprocessado e para que o programa rode com eficiência é necessária uma tabela para saber quantos índices deverão ser pulados:

j =	1	2	3	4	5	6	7	8	9	10
pattern[j] =	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*
next[j] =	0	1	1	0	1	1	0	5	0	1

Figure 1: Exemplo do pré-processamento do algoritmo KMP

O padrão a ser procurado no texto é *pattern*, j o índice desse padrão e *next* o índice do padrão que deverá ser comparado após uma discrepância. A tabela acima é utilizada para que seja possível verificar quanto índice no texto serão necessários pular quando houver uma diferença na comparação, a mesma possui tamanho equivalente ao padrão a ser procurado. Sua construção se dá a partir de duas variáveis que estarão operando como "ponteiros" nos índices desta tabela e o *pattern*, que para melhor entendimento as variáveis serão chamadas de t e k e a tabela construída a partir do *pattern* de *next* que receberão os valores iniciais *0* e *1* e *0* respectivamente. O t será utilizado para encontrar os prefixos e o k para encontrar os sufixos enquanto percorrem o *pattern* enquanto existir uma próxima letra. O t e k sempre irá incrementar +1, porém, enquanto *pattern[t]*(letra no padrão no índice i) diferir de *pattern[j]*(letra no padrão no índice j) a variável$t$ receberá o valor de *next[t]*(valor na tabela *next* no índice t),

após esta etapa o algoritmo irá verificar se o *pattern[t]* é igual ao *pattern[k]*, caso a condição seja verdadeira, o *next[t]* irá receber o *next[k]*, em outras palavras o valor que está na posição t na tabela *next* irá receber o valor que está na posição k na tabela *next*, caso a condição seja falsa, o *next[k]* recebe o valor de t. Desta forma todo o pré-processamento ocorre sem haver necessidade de retroceder em um índice que já foi verificado, assim, pode-se dizer que a complexidade deste pré-processamento é *O(n)*. Abaixo está representado uma sequência de imagem para ilustrar o processamento da comparação por KMP com os colchetes representando o índice no texto.

[B]	**A**	**B**	**C**	**B**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**C**	**A**	**B**	**C**
[A]	B	C	A	B	C	A	C	A	B																

Figure 2: No momento em que o primeiro índice do texto e do padrão forem comparados ocorrerá uma incongruência nos caracteres, portanto ao verificar o *next[j* será retornado o valor *0*, utilizando do cálculo *j - next[j]* serão pulados apenas 1 índice.

B	**[A]**	**B**	**C**	**B**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**C**	**A**	**B**	**C**
	[A]	B	C	A	B	C	A	C	A	B															

Figure 3: Neste momento os dois caracteres são equivalentes e o algoritmo continuará a comparar os próximo caractere até que ocorra uma discrepância ou que j alcance o tamanho do padrão ou que i alcance o tamamho do texto.

B	**A**	**B**	**C**	**[B]**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**C**	**A**	**B**	**C**
	A	B	C	[A]	B	C	A	C	A	B															

Figure 4: Agora que o algoritmo chegou em um índice no qual existe uma diferença nos caracteres que estão sendo comparados, o algoritmo não precisa comparar todos os caracteres anteriores novamente, então, será analisado o next[j] para que assim saiba quantos índices serão pulados.

B	**A**	**B**	**C**	**B**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**[A]**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**C**	**A**	**B**	**C**
					A	B	C	A	B	C	A	[C]	A	B												

Figure 5: Após encontrar outra discrepância no *pattern[j]* e no *text[i]*, o algoritmo irá consultar novamente a tabela *next[j]* e irá subtrair com o valor de j, neste caso o algoritmo irá pular 3 índices.

B	**A**	**B**	**C**	**B**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**[A]**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**C**	**A**	**B**	**C**
								A	B	C	A	[B]	C	A	C	A	B								

Figure 6: Aqui serão encontradas 4 equivalências, entretanto, ainda ocorrerá uma desigualdade e seguindo a lógica anterior o "j" neste caso é igual a 5 e o valor que se encontra em next[j] é 1, logo subtraindo 1 de 5 tem-se 4, assim o valor em i será incrementado em 4.

B	A	B	C	B	A	B	C	A	B	C	A	A	B	C	A	B	C	A	[B]	C	A	C	A	B	C
												A	B	C	A	B	C	A	[C]	A	B				

Figure 7: Após outras comparações, ocorrerá uma desigualdade no índice 8 do padrão, que utilizando da mesma lógica ele irá somar o resultado de j menos $next[j]$, desta forma serão necessários somar apenas 3 índices em i para que ocorra a próxima comparação.

B	A	B	C	B	A	B	C	A	B	C	A	A	B	C	A	B	C	A	B	C	A	C	A	[B]	C
															A	B	C	A	B	C	A	C	A	[B]	

Figure 8: Finalmente, o índice i alcança o tamanho da palavra e o algoritmo se encerra, pois, todos os caracteres foram verificados e não houve nenhuma discrepância.

3.3 Algoritmo de Boyer-Moore

3.3.1 Objetivo do algoritmo

O algoritmo Boyer-Moore foi criado em 1977 por Robert S. Boyer e J Strother Moore, baseado-se no algoritmo de Knuth-Morris-Pratt, de busca de uma ocorrência de uma "string" P em uma "string" T. Diferente do algoritmo de força bruta, que procura em cada carácter um alinhamento, o algoritmo de Boyer-Moore pre-processa a "string" a ser procurada. A razão do algoritmo ser muito usado é pela sua eficiência em relação a algoritmos que o eram utilizados anteriormente, como o de força bruta e KMP, onde o algoritmo de Boyer-Moore, ao invés de testar o alinhamento com cada carácter, pula caracteres que previamente não foram alinhados na palavra, pois já sabe que ali não haverá alinhamentos, o que torna o algoritmo mais rápido que o de força bruta, porque pular alinhamentos previamente falhos torna "strings" grandes mais fáceis de percorrer, pois, haverá diversas ocorrências de padrões na "string" percorrida em que alinhamentos não funcionariam, e uma vez reconhecidas e repetidas, ao invés de tentar repetir o alinhamento, o algoritmo irá pular até a última parte da "string", já que sabe que na posição os caracteres não serão alinhados. Dessa forma, sua eficiência aumenta quanto maior for a "string" a ser percorrida e quanto mais caracteres diferentes existirem, tornando-o um dos algoritmos preferidos a ser utilizado para procurar partes em registros, como no caso desta pesquisa.
Sua melhor performance é de O(n/m), 0(n·m) para a procura da "string", complexidade em notação Big O para o pre-processamento é de O(m+σ).

3.3.2 Bad character

O algoritmo irá percorrer a "string" horizontalmente, isto é, da esquerda para a direita, e os testes de alinhamento irão ocorrer também horizontalmente, mas desta vez da direita para esquerda. Isso ocorre porque não importa se o alinhamento da esquerda no início da "string" P irá funcionar, caso algum caractere falhe em alinhar com a "string" T percorrida. Começando o alinhamento pela direita, o algoritmo entende até onde o pulo deve ocorrer, no caso

de um alinhamento venha a falhar. É importante lembrar que o algoritmo de Boyer-Moore não só pula em caso de um alinhamento falhar, mas também tenta achar dentro da “string” P algum caractere que alinhe com o caractere previamente comparado da “string” T e , caso exista, o algoritmo irá tentar alinhar a primeira ocorrência do carácter na “string” P com caractere previamente falho da “string” T,isto é chamado de ”Good Suffix”. O caractere da “string” T que previamente falhou em alinhar é denominado “Bad character”. Após alinhar o “Bad character” com sua ocorrência na “string” P, irá novamente tentar alinhar a partir do fim da “string” P ate o início, que agora esta alinhada, e esse processo é repetido enquanto existirem caracteres que não se alinharem em ambas as “strings”. No caso do carácter da “string” T que falhou o alinhamento previamente na “string” P não existir, o algoritmo irá pular todos a mesma quantidade de caracteres presentes na “string” T, e em seguida testara o alinhamento novamente.

e	**u**		**a**	**m**	**o**		**c**	**[i]**	**m**	**i**	**t**	**a**	**r**	**r**	**a**	**s**		**e**		**g**	**u**	**i**	**t**	**a**	**r**	**r**	**a**	**s**
g	u	i	t	a	r	r	a	[s]																				
=9	A=1	C=9	E=9	G=8	I=6	M=9	O=9	R=2	S=9	T=5	U=7																	

Figure 9: Após o pre-processamento das ”string” comparadas,pre-processamento este que ira atribuir os valores de pulo de cada ”Bad character”, se compara o último carácter da “string” com o seu equivalente no texto, como às duas posições não tem caracteres iguais, haverá um pulo, mas como o carácter I esta presente na “string” “guitarras” haverá uma tentativa de alinhamento entre os dois “I” da “string” e do texto por isso seu pulo como “Bad character” é de 6 caracteres, vale tambem lembrar que como não espaços na “string” procurada o valor de ”Bad Character” do caratere” ” é de 9, que o tamanho total da “string” procurada permitindo o algoritmo pule os próximos 9 índices.

e	**u**		**a**	**m**	**o**		**c**	**i**	**m**	**i**	**t**	**a**	**r**	**r**	**[a]**	**s**		**e**		**g**	**u**	**i**	**t**	**a**	**r**	**r**	**a**	**s**
						g	u	i	t	t	a	r	r	a	[s]														
=9	A=1	C9	E9	G8	I6	M9	O9	R=2	S=9	T=5	U=7																	

Figure 10: Após o primeiro pulo ambos os “I” se encontram alinhados, testamos o último carácter com seu equivalente, já que os dois diferem, acontece o pulo de 2 caracteres, pois este é o valor de “Bad character” de ”a”.

e	**u**		**a**	**m**	**o**		**c**	**i**	**[m]**	**i**	**t**	**a**	**r**	**r**	**a**	**s**		**e**		**g**	**u**	**i**	**t**	**a**	**r**	**r**	**a**	**s**
								g	[u]	i	t	a	r	r	a	s												
=9	A=1	C=9	E=9	G=8	I=6	M=9	O=9	R=2	S=9	T=5	U=7																	

Figure 11: Após alinhar o caráter “a” em ambos o texto e na “string”, se começa o teste de alinhamento do último carácter, desta vez ambos se alinham, logo o algoritmo irá testar todos os caracteres da direita para a esquerda, até ter certeza que a palavra se alinha completamente ou que há uma falha de alinhamento, esta falha ocorre ao comparar os caracteres ”u” da “string” com “m” do texto, como o alinhamento falhou após o primeiro teste funcionar, o algoritmo pula a mesma quantidade de caracteres que está presente na “string” procurada.

e	u		a	m	o		c	i	m	i	t	a	r	r	a	s		e		g	u	i	t	a	[r]	r	a	s
																	g	u	i	t	a	r	r	a	[s]			
=9	A=1	C=9	E=9	G=8	I=6	M=9	O=9	R=2	S=9	T=5	U=7																	

Figure 12: Testando o último alinhamento, o carácter “s” e diferente do carácter “r”, o valor de “Bad character” de r = 2, logo o algoritmo tenta alinhar o “r” do texto com o da ”string”.

e	u		a	m	o		c	i	m	i	t	a	r	r	a	s		e		g	u	i	t	a	r	r	[a]	s
																			g	u	i	t	a	r	r	a	[s]	
=9	A=1	C=9	E=9	G=8	I=6	M=9	O=9	R=2	S=9	T=5	U=7																	

Figure 13: Testa o último alinhamento, o alinhamento é falho, pois “a” difere de “s”, o pulo do “Bad character” a é de 1.

e	u		a	m	o		c	i	m	i	t	a	r	r	a	s		e		[g]	u	i	t	a	r	r	a	s
																				[g]	u	i	t	a	r	r	a	s
=9	A=1	C=9	E=9	G=8	I=6	M=9	O=9	R=2	S=9	T=5	U=7																	

Figure 14: O algoritmo testa o último caráter, como não há erro de alinhamento o algoritmo compara o alinhamento dos outros caráteres, com seu começo na direita e fim na esquerda, como todos os caráteres testados se alinharam o algoritmo achou a “substring” que procurava.

3.4 O Algoritmo de Smith-Waterman

3.4.1 Longest Common Subsequence

É um problema clássico da ciência da computação, que consiste em achar a subsequencia mais longa e comum de todas as sequências das qual procura. O LCS (Longest Common Subsequence), traduzido, maior subsequência em comum, utiliza-se de programação dinâmica e é uma das soluções para otimização de tempo para diversos problemas. Ele é definido por uma “substring”, que é comum a outras duas “strings”, isto é, ao comparar duas “strings”, é retornada as “substrings” comuns mais longas. Ao receber duas “strings”, as “substrings” são geradas pela comparação de “strings”, ela não estará necessariamente em uma ordem contínua. Para achar essas “substrings” o algorítimo que é empregado utiliza-se de uma complexidade de comparações, onde primeiro procura saber quantas combinações de “substrings” existem. Ao obter os dados das combinações, são retornados diversos valores, tendo como resultado a “substring” que mais se assemelha com a “string” principal. O algoritmo é utilizado em diversos bancos de dados para buscas, na área da bioinformática, e, também, para a comparação de arquivos, chamados de diff’s, para saber qual parte dos arquivos estão as ‘strings’ que mais se assemelham.

3.4.2 O Algoritmo

O algoritmo de Smith-Waterman é uma variação do Needleman-Wunsch, que foi proposto por Temple F. Smith e Michael S. Waterman em 1981, e que,

diferentemente desse que usa do alinhamento global, tenta encontrar o alinhamento otimizado em relação a seu sistema de pontuação, onde as matrizes de pontuação negativa tem seus valores definidos como zero, o que torna os pontuados positivos mais visíveis. O algoritmo faz a busca comparando uma matriz, utilizando um sistema de pontuação. O sistema inicia na célula da matriz com maior pontuação, segue até a célula com pontuação zero, isso é chamado de "Traceback", onde se descobre o "Longest common subsequence", realizando assim um alinhamento local. O maior problema do algoritmo é sua busca demorada, com muita complexidade para textos maiores, exigindo bastante tempo, decorrente do algoritmo testar todos os caracteres entre as duas "string" que são comparadas e das próprias comparações demoradas empregadas. Na matriz, é alinhado duas sequencias de texto que podem corresponder ou não, sendo que as partes que correspondem geram uma pontuação positiva na matriz, enquanto nas negativas a pontuação é definida para 0. A matriz de pontuação é responsável pelas comparações da pontuação em todas as linhas e colunas, trazendo o alinhamento local de forma confiável. Na matriz, toda vez que a comparação das letras forem semelhantes, será gerada uma pontuação positiva e caso não sejam a pontuação será negativa, criando uma linha diagonal, indicando a pontuação com os valores mais altos, retornando assim o grau de semelhança de cada sequência de caracteres. Também se percebe que o algoritmo usa dos princípios da "Longest Common Subsequence" ao tentar procurar a sequência que melhor se alinha entre as duas "strings", mas não necessariamente onde ela se alinha totalmente.

Exemplo de uma matriz como as encontradas no algoritmo de Smith- Waterman, quando: alinhamento correto = +1, alinhamento falho = -1 e pulo no alinhamento = -2

RECURSÃO:

$$\mathrm{S_{ij}} = \max \begin{cases} S_{\mathrm{i-1j-1}} + s(a_{\mathrm{i}}, b_{\mathrm{j}}) \\ S_{\mathrm{i-1j}} + s(a_{\mathrm{i}}, -) \\ S_{\mathrm{ij-1}} + s(-, b_{\mathrm{j}}) \\ 0 \end{cases} = \max \begin{cases} S_{\mathrm{i-1j-1}} + 1, a_{\mathrm{i}} = b_{\mathrm{j}} \\ S_{\mathrm{i-1j-1}} + -1, a_{\mathrm{i}} \neq b_{\mathrm{j}} \\ S_{\mathrm{i-1j}} + (-2), b_{\mathrm{j}} = - \\ S_{\mathrm{ij-1}} + (-2), a_{\mathrm{i}} = - \\ 0 \end{cases}$$

S		A	C	C	G	T	G	A
	0	0	0	0	0	0	0	0
G	0	0	0	0	1	0	1	0
T	0	0	0	0	0	2	0	0
G	0	0	0	0	1	0	3	1
A	0	1	0	0	0	0	1	4
A	0	1	0	0	0	0	0	2
T	0	0	0	0	0	1	0	0
A	0	1	0	0	0	0	0	1

4 Coleta e Analise de dados

4.1 Coleta dos dados

Sendo T a “string” onde será procurada no banco de dados em formato .CSV e a “string” I as amostras coletadas que foram calculadas usando a função omp-get-wtime da biblioteca omp.h, onde o retorno da função é atribuída a uma variável antes de começar o laço que ira percorrer a “string” T, e depois atribuída a outra variável após o término do laço. O tempo total para percorrer a “string” T é obtido ao subtrair o valor da variável que foi atribuído no fim do laço do valor da variável atribuído ao termino. O valor da subtração será gravado em um arquivo CSV, junto da linha onde o processo ocorreu, sendo as linhas onde o tempo é gravado: linha 5000, linha 50000, linha 500000, linha 5000000 e, no fim do banco de dados, linha 49056660 . Os nomes procurados foram organizados de forma que a inicial de cada nome seja um caractere do alfabeto latino, conforme estabelecido pelo Acordo Ortográfico, onde cada letra do alfabeto recebeu testes com dois nomes masculinos e dois femininos.

5 Resultados e Teste

A tabela e o gráfico a seguir mostram os resultados dos testes realizados, onde o resultado total é a média do tempo em segundos que cada algoritmo levou para alcançar a linha referenciada.

	linha	tempo
Knuth-Morris-Pratt	5000	0,021923076923077
Knuth-Morris-Pratt	50000	0,186346153846154
Knuth-Morris-Pratt	500000	1,79711538461538
Knuth-Morris-Pratt	5000000	17,4567230769231
Knuth-Morris-Pratt	49056660	167,975576923077
Boyer-Moore	5000	0,014519230769231
Boyer-Moore	50000	0,144230769230769
Boyer-Moore	500000	1,39490384615385
Boyer-Moore	5000000	13,6231730769231
Boyer-Moore	49056660	130,089902912621
Smith-Waterman	5000	0,053300970893986
Smith-Waterman	50000	0,487087378640777
Smith-Waterman	500000	4,72592233009709
Smith-Waterman	5000000	45,644854368932
Smith-Waterman	49056660	433,925212765957

Figure 15: Tabela

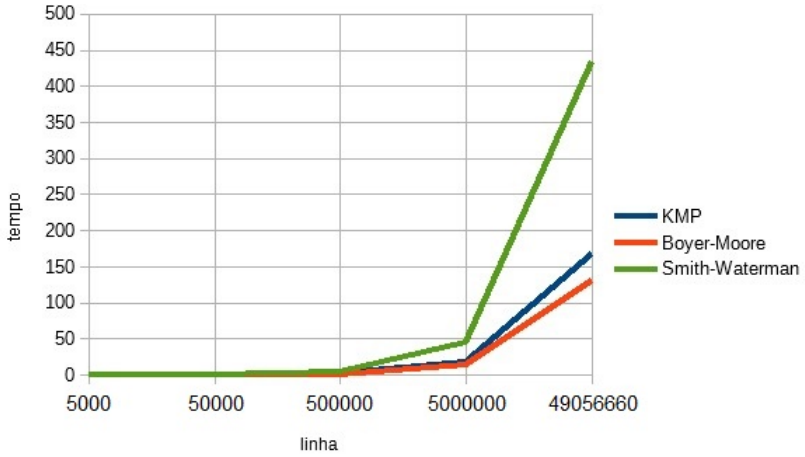

Figure 16: Grafico

Os resultados demonstram que o algoritmo de Boyer-Moore é o algoritmo mais rápido, seguido do algoritmo de Knuth-Morris-Pratt e, por fim, o algoritmo de Smith-Waterman. Isso ocorre porque o algoritmo de Boyer-Moore, ao ser executado em um banco de dados com um grande alfabeto, como o latino, pode criar mais Bad characters, o que permite o algoritmo realizar pulos mais vezes percorrendo o banco de forma rápida. O algoritmo de Knuth-Morris-Pratt vem em seguida, pois ainda existem várias oportunidades de pulo, mas a comparação de caracteres é mais lenta, e o algoritmo ainda pode ser melhor utilizado em banco de dados com alfabetos menores como “DNA”, que tem seu alfabeto é

constituída de 8 bases, e nessa situação o algoritmo tende a ter a mesma ou maior velocidade que o algoritmo de Boyer-Moore. O algoritmo de Smith-Waterman demonstrou ser o mais demorado, pois suas comparações são lentas, e ao realizar um número "n" de pesquisas, serão criadas "n" matrizes, aumentando o uso de memória RAM, porque o algoritmo ainda compara caractere por caractere, sem o uso de pulos para percorrer o banco de dados, logo sua implementação é pesada e lenta, e o algoritmo de Smith-Waterman ainda pode ser utilizado para comparar strings pequenas, onde o intuito não é achar onde uma "string" alinha totalmente com uma "substring", mas sim o maior alinhamento entre ambos.

6 Conclusão

Após a realização de testes em bancos de dados divulgados, chegamos à conclusão que entre os três algoritmos propostos, o de Boyer-Moore se prova o mais eficiente. Devemos considerar o fato de que a pesquisa é local, isto é, sem conexões à rede, e que existem diversos algoritmos de busca e alinhamento que não foram comentados neste trabalho. A implementação do algoritmo de Boyer-Moore se provou relativamente rápida, mas ainda devem ser consideradas as características específicas do aplicativo "CAIXA Auxilio Emergencial", onde os resultados aqui apresentados podem não ser os mesmos caso o algoritmo seja usado na aplicação, por diversos fatores, como localização do banco de dados, e a lógica utilizada para implementar o algoritmo. A comparação entre os 3 algoritmos serve como uma avaliação de cada algoritmo em uma determinada situação, logo pode ser de interesse dos responsáveis pelo aplicativo entenderem sua viabilidade no sistema criado pelos mesmos pois, ainda exitem soluções funcionais e rapidas como SQL, que funcionam em conjunto com o servidor. Concluímos que o algoritmo proposto pode melhorar o tempo em consultas ao cadastro de beneficiados pelo programa de auxílio emergencial, caso o mesmo se encontre offline, facilitando a fiscalização pelos diversos entes interessados.

7 Referencias

[1] Robbi Rahim et al 2017 J. Phys.: Conf. Ser.930 012001

[2] S.B.Robert and S.M.J"A Fast String Searching Algorithm, Comm. ACM. New York: Association for Computing Machinery,1977

[3]E.K.Donald,H.M.James and R.P.Vaughan, "Fast Pattern Matching In Strings",SIAM Journal on Computing,1977

[4]T.F.Smite and M.S.Waterman,"Identification of Common Molecular Subsequences",J.Mol.Biol.(1981)147,195-197

[5]Kernighan, Brian W.; Ritchie, Dennis M. (February 1978). The C Programming Language (1st ed.). Englewood Cliffs, NJ: Prentice Hall. ISBN 978-0-13-110163-0

[6] Knuth D, Morris J H, Pratt V 1977 Fast pattern matching in strings SIAM J. Comput. 6 323–350

[8] Christian Charras, Thierry Lecroq, Knuth-Morris-Pratt algorithm, Disponível em: http://www-igm.univ-mlv.fr/ lecroq/st node8.html

[9] Christian Charras, Thierry Lecroq, Boyer-Moore algorithm, Disponível em: http://www-igm.univ-mlv.fr/ lecroq/string/node14.html¿

[10] Longest Common Subsequence, Disponível em: https://dyclassroom.com/dynamic-programming/longest-common-subsequence