UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

AUTOCOMPLETAR

Aluno(s): Lucas Felix e Lucas Lâgoa
Curso: Ciência da Computação
Professor: Leonardo Rocha

1-INTRODUÇÃO

No cotidiano atual a utilização da internet é indispensável, e a forma mais eficiente de comunicação na mesma é por meio de mensagens, seja utilizando bate-papos ou aplicativos especializados para celular. Ao pesquisarmos algo na internet sempre nos deparamos com o "auto-complete" seja nos buscadores como Google ou em sites que nos oferecem pesquisar pelo mesmo, também podemos utilizar de recurso parecido por meio de editores textuais como o Sublime que quando estamos digitando algoritmos nos oferece a provável continuação da palavra, bastando apertar a tecla TAB do teclado para completar a mesma.

Pensando nisso foi o feito o trabalho prático da disciplina de algoritmos e estrutura de dados três.

2-PROBLEMA PROPOSTO

O problema proposto por esse trabalho prático seria o de praticar as técnicas de programação sobre casamento de caracteres, e para esse trabalho seria o caso de implementar um "auto-complete" baseado na pesquisa de um buscador qualquer, que seria o casamento do prefixo digitado pelo usuário, e mostraria para o mesmo as palavras que casam com esse prefixo ordenadas em ordem decrescente com o peso, e para complementar esse buscador, foi solicitado a implementação de uma interface gráfica que integrasse o algoritmo e simulasse o "auto-complete" de um buscador.

3-SOLUÇÕES IMPLEMENTADAS

3.1-FORÇA BRUTA

Um algoritmo força bruta basicamente compara todos os caracteres, a fim de achar a pesquisa que nos interessa. Contudo estamos interessados apenas nos prefixos, ou seja, a parte inicial da palavra, a busca pelos caracteres deve ser exata, deve-se pegar a palavra de entrada digitada, e achar outras palavras que batem exatamente com aquela até o ponto em que a mesma foi digitada.

Por exemplo, ao digitarmos o prefixo "com", podemos ter como resultado, como, comics, computador, mas ao aumentarmos em uma letra tal prefixo, deixando de ser "com" e passando a ser "comp", apenas um dos resultados acima nos será satisfatório, que será computador. Por tal motivo comparamos as posições casa a casa.

3.1.2- IMPLEMENTAÇÃO

Após a leitura dos parâmetros de entrada do programa e leitura da base de dados fornecida, verifica-se o tamanho da palavra digitada como parâmetro para que não peguemos palavras que sejam menores que a mesma, ou seja, para entrar como possível retorno no auto-complete, a palavra deve ter pelo os primeiros caracteres iguais o da palavra digitada, e ser de tamanho maior ou igual aquilo que se deseja pesquisar.

Depois de feito isso, verificamos se a palavra digitada pelo usuário é maiúscula, caso seja, a mesma é transformada em minúscula, utilizando a função min, que aumenta em 32 os valores, com base na tabela ASCII, deixando os mesmos com a letra minúscula.

Valor em decimal	Letra maiúscula	Valor em decimal	Letra minúscula
65	A	97	A
66	B	98	B
67	C	99	C
68	D	100	D
69	E	101	E

Tabela 1: Mostra basicamente o funcionamento da tabela ASCII

Para que isso ocorra a função seleciona, compara a palavra fornecida pelo usuário com as palavras que temos em nosso dicionário, como fora explicado mais acima, enquanto os caracteres da palavra a ser pesquisada bater com os caracteres de uma palavra qualquer do dicionário, temos uma contador que se modifica, caso ao final da comparação o contador tem valor igual ao tamanho da palavra pesquisada, então tal palavra, é uma candidata para o auto-complete, dessa forma, esta palavra é armazena em um outro vetor, que contém apenas as palavras selecionadas, tal processo se repete até que tenham sido comparadas todas as palavras.

Compara-se também a palavra com os valores minúsculo e maiúsculo da palavra, para caso a base de entrada esteja digitada de forma incorreta, obtemos a resposta de qualquer forma.

Ordem de comparação:	Primeiro:	Segundo:	Terceiro:			
Palavra Pesquisada:	B	A	T			
Palavra do Dicionário:	B	A	T	A	T	A
Resultado:	Caractere igual	Caractere igual	Caractere igual			
Palavra do Dicionário:	B	A	T	M	A	N
Resultado:	Caractere igual	Caractere igual	Caractere igual			
Palavra do Dicionário:	B	A	N	A	N	A
Resultado:	Caractere igual	Caractere igual	Caractere Diferente			
Palavra do Dicionário:	M	Ú	S	I	C	A
Resultado:	Caractere diferente	Caractere diferente	Caractere diferente			

Tabela 2: exemplifica basicamente como funciona a busca no algoritmo força bruta.

Olhando a tabela 2, podemos ver como o força bruta realiza suas comparações, caractere por caractere, caso um dos caracteres seja diferente, desconsidera-se tal palavra.

Depois que já temos as palavras pré-selecionadas para serem mostradas como resultado temos que ordená-las pelo seu peso. Entretanto antes de ordenar recebemos o peso das palavras em um vetor, e este é ordenado utilizando o famoso algoritmo de ordenação bubble sort. Basicamente o algoritmo de ordenação percorre o vetor comparando os elementos adjacentes, trocando as posições dos elementos que estiverem fora de ordem, repetindo o processo até que reste apenas um elemento.

Após feito isso temos a última função, responsável por basicamente imprimir na tela os resultados encontrados, para isso temos uma última comparação, onde comparando-se o peso das palavras selecionadas com o vetor ordenado, caso o valor seja igual, a palavra já ordenada é mostrada.

3.2- HEURÍSTICO

O segundo algoritmo implementado, é uma heurística exata, baseado no famoso algoritmo de processamento de caracteres KMP (Knuth-Morris-Pratt). Neste algoritmo fazemos as

comparações de trás para a frente (assim como no KMP), evitando comparações desnecessárias quando um caractere do nosso dicionário não bate com o padrão digitado pelo usuário.

3.1.2- IMPLEMENTAÇÃO

O algoritmo implementado como dito acima é uma variação do KMP, seu começo é basicamente igual ao algoritmo de força bruta, sendo que, primeiramente faz-se a leitura dos parâmetros e depois verifica se a palavra digitada está em letras maiúsculas, caso a mesma esteja, coloca-se a palavra em letras minúsculas e começa-se a busca.

O algoritmo faz basicamente sua busca, comparando os caracteres de trás para a frente, caso os caracteres sejam iguais temos um contador que se modifica, caso ao final das comparações o contador seja igual ao tamanho da palavra digitada pelo usuário, passa-se as posições na estrutura das palavras e também seus pesos, também se modifica um contador k, que representará o total de palavras copiadas.

Ordem de comparação:	Terceiro:	Segundo:	Primeiro:	n.c.	n.c.	n.c.	n.c.	n.c.	n.c.	n.c.
Palavra digitada:	M	A	N							
Palavra dicionário:	M	A	N	A	D	A				
Resultado:	Caractere igual	Caractere igual	Caractere igual							
Palavra dicionário:	M	A	M	A	E					
Resultado:	Não compara	Não compara	Caractere diferente							
Palavra dicionário:	C	A	M	U	F	L	A	G	E	M
Resultado:	Caractere diferente	Caractere igual	Caractere igual							

Tabela 3: A tabela mostra basicamente como é o funcionamento do algoritmo heurístico implementado, sendo que n.c. quer dizer que não existe comparação

Depois de feito tal processo, já se possui a posição das palavras que possuem prefixo igual ao da palavra procurada, temos que receber os pesos das palavras que irão ser impressas para ordenar as mesmas, depois de recebido, ordena-se tal o vetor que possui os pesos utilizando o algoritmo de ordenação bubble-sort (mesmo utilizado no algoritmo de força bruta).

Com o vetor ordenado já, basta imprimir as palavras comparando-se com o peso, nas posições retornadas na função de busca, caso o peso seja igual, temos o retorno das palavras que possivelmente estão sendo procuradas pelo usuário.

4-ANÁLISE DE COMPLEXIDADE

Nesta seção analisaremos e compararemos os algoritmos implementados, também iremos mostrar a diferença entre todos os uns dos algoritmos em termos de complexidade.

4.1-ALGORITMO DE FORÇA BRUTA

Para começar o algoritmo de força bruta utiliza-se de dois loops aninhados responsáveis pela leitura do arquivo, um que roda enquanto não atingir o número total de palavras contidas no arquivo, tendo outro loop que basicamente repete tal processo, sendo assim a complexidade da leitura é O(número de palavras* número de palavras), sendo sua função $O(n^2)$, sendo n o número total de palavras lidas.

Depois de feita a leitura utiliza a função strlen, que é uma função que retorna o tamanho de um string, tal função está presente na biblioteca string.h, sendo complexidade para o retorno do tamanho O(n), sendo n o tamanho da palavra.

Após o retorno do tamanho correto da palavra, a função min verifica se a palavra digitada pelo usuário está em letra maiúscula ou minúscula, caso esteja minúscula ele simplesmente mantém a palavra, caso seja maiúscula, temos um for que percorre a palavra transformando as letras maiúsculas em minúsculas, a complexidade desta função é a mesma que da função strlen, sendo a mesma O(n), sendo n o tamanho da palavra.

A função seleciona palavras, como o próprio nome já diz, seleciona possíveis palavras que serão apresentadas no auto-complete, para isso temos dois loops aninhados que vão até o tamanho da palavra, seguido de uma condicional que verifica se a palavra do dicionário tem prefixo igual a palavra digitada pelo usuário, verificando também a caso a mesma letra esteja digitada em letra maiúscula, caso seja igual temos uma outra condicional que verifica se o valor do contador é igual ao tamanho da palavra pesquisada, que foi retornado pela função strlen, caso seja, temos um loop que irá copiar as palavras selecionadas para uma lista, sendo que este loop percorre até 100, sendo este definido como o tamanho da maior palavra. Desta forma a complexidade final é O(4* número de palavras * número de palavras*100).

Depois de selecionadas as palavras que serão utilizadas, recebe-se através de um loop que vai até o número de palavras, em um vetor o peso das palavras, tal vetor é ordenado usando o algoritmo bubble sort que tem complexidade no pior caso de $O(n^2)$, sendo n o número de palavras, logo a complexidade final da função vetor maiores mais o da função bubble sort é O(número de palavras)+$O(n^2)$.

Após retornado o vetor já ordenado, temos a função seleção, que imprime as palavras já ordenadas para serem retornadas ao usuário, nesta função possui-se dois loops aninhados, sendo

que um vai de 0 até a quantidade de palavras digitadas pelo parâmetro k, e outro que vai até o total de palavras, também possui uma estrutura condicional que verifica se o vetor ordenado tem o peso igual ao da palavra selecionada, caso tenha imprime-se tal palavra, logo a complexidade da função é O(2*k* número de palavras).

Com isso podemos concluir que a complexidade do algoritmo é O(max(número de palavras*número de palavras)+O(n)+O(n)+O(4*número de palavras*número de palavras*100)+O(número de palavras)+O(n^2)+O(2*k*número de palavras), sendo sua complexidade final de O(400*número de palavras*número de palavras) ou O(400*n^2).

4.2- ALGORITMO HEURÍSTICO

O algoritmo heurístico possui uma leitura igual à do algoritmo de força bruta, o que faz com que seu começo seja basicamente igual, sendo a complexidade da leitura O(número de palavras*número de palavras).

Depois é utilizada a função strlen (assim como no força bruta), da biblioteca string.h, que tem como função o retorno do tamanho da palavra, sendo que a mesma possui complexidade O(n), sendo n o número de caracteres presente na palavra digitada pelo usuário.

Depois disso é feito a verificação se a palavra que o usuário digitou é maiúscula ou minúscula, para isso utiliza-se de um loop que varia de zero até o tamanho da palavra digitada pelo usuário, e possui uma condicional que verifica se a palavra está com letras maiúsculas, caso esteja apenas coloca a mesma com letras minúsculas, sendo assim a complexidade da função é O(tamanho palavra digitada * 2).

Depois é feita a função que é responsável pela parte de retorno das palavras que serão selecionadas, sendo que a mesma possui dois loops, sendo um de zero até o número total de palavras do dicionário e outro que começar e outro que começa do tamanho da palavra digitada pelo usuário até 0, decrementando, dentro dos loops, há duas condicionais, sendo que uma verifica se a letra do dicionário é maiúscula ou minúscula, e outra que verifica se o tamanho do contador é igual ao tamanho da palavra digitada pelo usuário, caso seja retorna-se a posição da palavra na lista. A complexidade da função logo é O(número de palavras * tamanho palavra digitada * 4).

Após receber a posição correta das palavras selecionadas, temos a função responsável por receber o peso das palavras a ser ordenado, para fazer isso utiliza-se um uma estrutura de repetição que vai de 0 até k, sendo k o número de palavras que batem com os caracteres que estamos procurando. Sendo assim a complexidade da função é O(k), sendo k o número de palavras retornadas na função h.

Para finalizar temos a função imprime, a mesma é responsável por imprimir as palavras selecionadas. Para isso precisa-se ordenar o peso das palavras, utilizando-se o algoritmo bubble sort, que tem complexidade para o pior caso de $O(n^2)$, sendo n, o número de palavras retornadas na função h, ou seja as palavras que baterão com o prefixo procurado. Agora com as palavras já ordenadas, imprime-se as mesmas na ordem, para isso utiliza-se de dois loops, sendo que um vai de 0 até o valor k, sendo k o número de retornos desejados, o número de palavras que serão dadas de retorno para o usuário, e possui-se também outro loop que vai de 0 até o número de palavras retornadas pela função h, que como já foi explicada são as palavras que possuem o prefixo igual aquele digitado. Dentro dos loops possui-se uma condicional que verifica o peso das palavras, depois imprimindo as mesmas. Logo a complexidade da função é O(número de palavras que serão impressas * número de palavras com prefixo igual * 2)+$O(n^2)$.

A complexidade final do algoritmo é O(Max((n^2) + (n) + (tamanho palavra digitada*2) + (número de palavras*tamanho da palavra digitada*4) + (k)+ (número de palavras que serão impressas*número de palavras com prefixo igual*2) + $O(n^2)$) = O(número de palavras* tamanho da palavra digitada * 4).

5-ANÁLISE DE RESULTADOS E TESTES

Na análise de resultado e testes será feita uma comparação dos dois algoritmos implementados, mostrando as vantagens a relação ao tempo do algoritmo heurístico em relação ao força bruta, também será feita a análise de tempo utilizando-se o a interface, para que seja visualizado se a mesma traz alguma interferência em relação ao tempo de resposta do algoritmo. Foram feitas também análises comparando o prefixo de entrada e o número de palavras que seriam retornadas para ver qual seria o comportamento do programa, nas seções abaixo tais análises serão vistas com mais cuidado.

Para o início das análises, foi utilizada uma base de entrada ordenada, com um prefixo de tamanho um e com um retorno de palavras igual três.

	Prefixo = 1 e Retorno = 3		Prefixo = 5 Retorno = 10	
Número de palavras	Força Bruta	Heurístico	Força Bruta	Heurístico
10	0 m/s	0 m/s	0 m/s	0 m/s
20	0 m/s	0 m/s	0 m/s	0 m/s
50	0 m/s	0 m/s	0 m/s	0 m/s
100	0 m/s	0 m/s	0 m/s	0 m/s
500	2 m/s	0 m/s	3 m/s	0 m/s
1000	6 m/s	0 m/s	8 m/s	0 m/s
5000	52 m/s	3 m/s	50 m/s	3 m/s
10000	217 m/s	10 m/s	178 m/s	5 m/s
20000	856 m/s	11 m/s	659 m/s	15 m/s
50000	5513 m/s	29 m/s	4003 m/s	20 m/s
100000	22315 m/s	85 m/s	15902 m/s	28 m/s
150000	49494 m/s	158 m/s	35893 m/s	44 m/s
200000	82719 m/s	263 m/s	63885 m/s	55 m/s
250000	195946 m/s	392 m/s	102747 m/s	62 m/s
300000	226906 m/s	561 m/s	146450 m/s	70 m/s

Tabela 4: Essa tabela demonstra o comportamento do algoritmo para um tamanho de prefixo igual a um e retorno de palavras igual a três em uma lista ordenada

Pode-se perceber através desta tabela, o quão discrepante é o tempo do força bruta em comparação com o algoritmo heurístico, tal coisa já era esperada, contudo era esperado valores menores de tempo para o força bruta.

Além do teste feito acima, foram feitos outros, variando os valores de palavras a serem retornadas e variando o tamanho da palavra a ser pesquisada, ao fazer isso, foi percebido que palavras que ao se digitar uma palavra de apenas uma letra, como por exemplo “a”, e na base de

entrada houver muitas palavras começando com tal letra, o algoritmo heurístico, gasta um tempo maior para exercer sua função de encontrar e ordenar as palavras.

Na tabela abaixo, podemos ver mais alguns testes feitos:

	Prefixo = 2 e Retorno = 5		Prefixo = 5 e Retorno = 10	
Número de palavras	Força Bruta	Heurístico	Força Bruta	Heurístico
10	0 m/s	0 m/s	0 m/s	0 m/s
20	0 m/s	0 m/s	0 m/s	0 m/s
50	0 m/s	0 m/s	0 m/s	0 m/s
100	0 m/s	0 m/s	0 m/s	0 m/s
500	2 m/s	0 m/s	2 m/s	0 m/s
1000	3 m/s	0 m/s	7 m/s	0 m/s
5000	47 m/s	3 m/s	59 m/s	4 m/s
10000	174 m/s	7 m/s	180 m/s	6 m/s
20000	667 m/s	10 m/s	704 m/s	5 m/s
50000	4092 m/s	57 m/s	4260 m/s	18 m/s
100000	16527 m/s	73 m/s	16924 m/s	41 m/s
150000	37090 m/s	76 m/s	37928 m/s	45 m/s
200000	63578 m/s	96 m/s	67952 m/s	48 m/s
250000	106803 m/s	815 m/s	100702 m/s	62 m/s

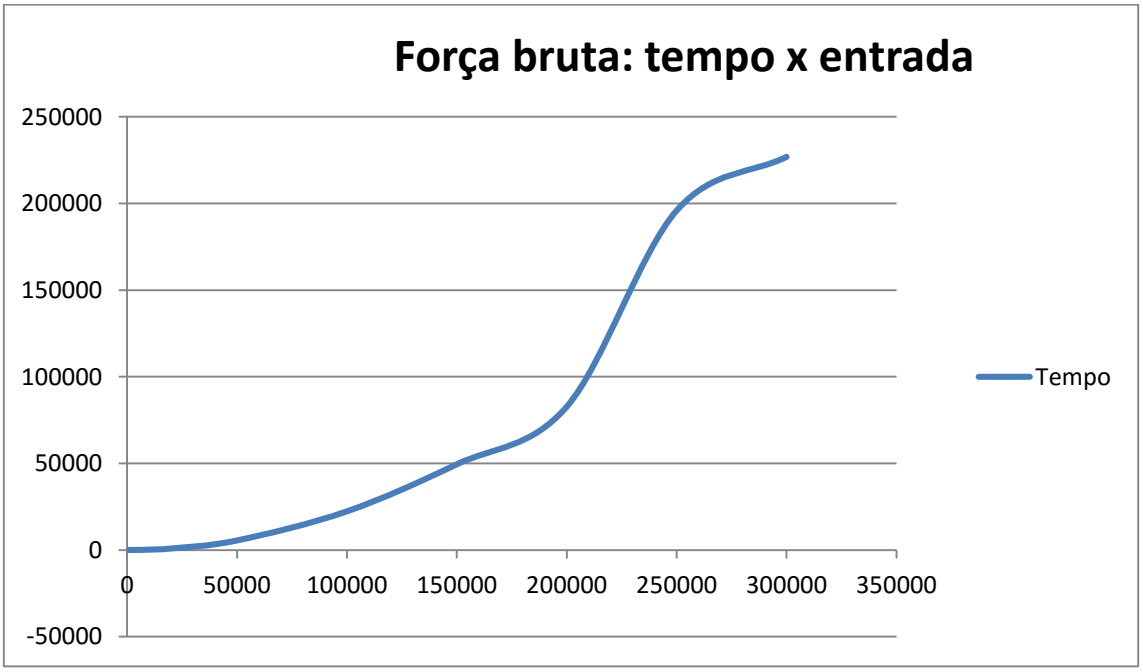
Tabela 5: Está tabela mostra o comportamento do algoritmo quando utilizada uma lista não ordenada

Comparando que o algoritmo heurístico e o algoritmo de força bruta possuem um comportamento basicamente igual, exceto em alguns casos que o tempo gasto é maior. Pelas tabelas podemos perceber que mesmo variando o tamanho do prefixo, mudando a quantidade de palavras a serem impressas e mesmo mudando a base de entrada, de palavras ordenadas para uma totalmente desordenada não surge nenhum efeito tão grande nos programas e seus valores continuam praticamente iguais.

Contudo analisando de forma mais detalhista vemos que ao procurarmos um prefixo maior que um há uma pequena variação no tempo, pois a quantidade de comparações aumenta, já que temos que comparar até onde os caracteres não batem, no melhor caso não irá bater na primeira letra, mas no pior caso, só irá bater na última, fazendo com que houvessem uma série de comparações desnecessárias.

Com base nos dados já informados aqui e nas tabelas, o gráfico que basicamente define os comportamentos dos algoritmos é basicamente os que são mostrados abaixo:

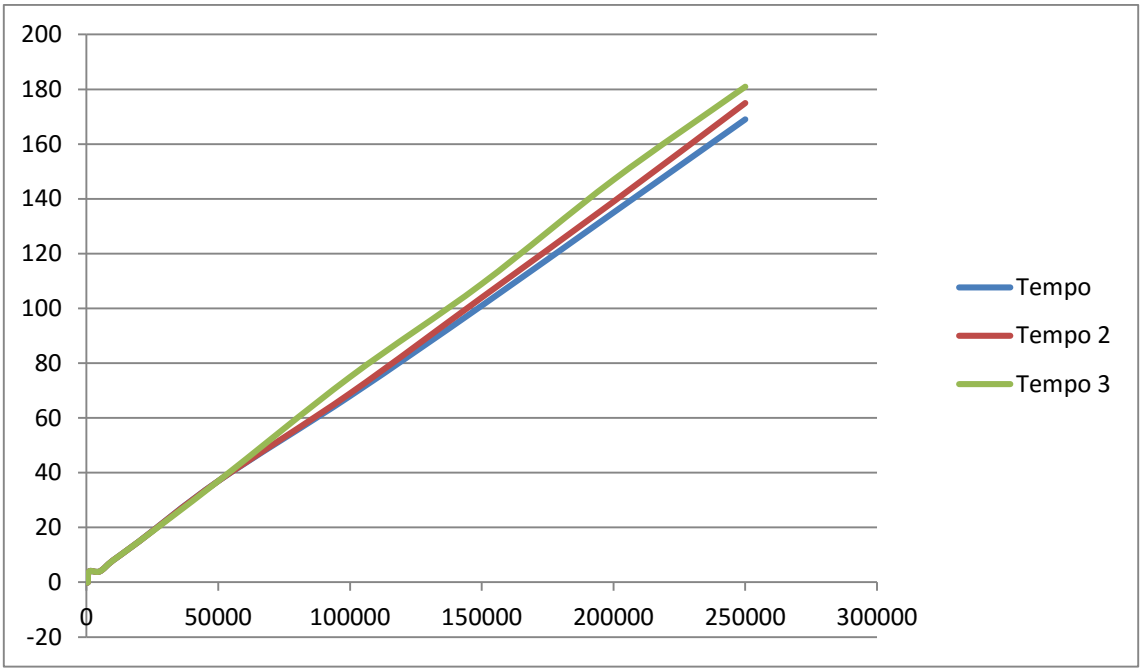
1º Gráfico: Define basicamente o comportamento do algoritmo heurístico



2º Gráfico: Define basicamente o comportamento do algoritmo de força bruta

Percebe-se através destes gráficos que o comportamento do algoritmo heurístico é mais constante que o do algoritmo força bruta, podemos ainda ver pelo tempo, localizado no eixo y, o

quão mais eficiente é o algoritmo heurístico, fornecendo, assim como o força bruta o resultado correto, porém com um tempo de resposta muito menor.



3º gráfico: No gráfico acima vemos o comportamento do algoritmo heurístico com interface gráfica

No 3º gráfico, foi utilizado o algoritmo heurístico com a interface gráfica, para o tempo um foi utilizado palavras de tamanho igual a um, para o tempo dois palavras de tamanho igual a dois e para o tempo três palavras de tamanho igual a cinco, podemos perceber que o a parte gráfica não alterou no comportamento do algoritmo, e que com diferentes quantidades de caracteres, o gráfico nos revela que o tempo quase não muda. Contudo, para palavras muito frequentes no dicionário, o tempo no pior caso registrado para pesquisa de uma letra foi cerca de 4 segundos para fazer a pesquisa, mostrando que no caso médio o algoritmo tem um comportamento muito bom, entretanto no pior caso possui um desempenho que deixa a desejar.

6 – INTERFACE GRÁFICA

Visando um sistema de integração eficiente que se pareceria com um “auto-complete” de um buscador, a interface gráfica foi implementada em HTML e PHP. No arquivo PHP, contém a integração do HTML com o código, sendo que ela recebe exatamente como é executado no terminal, e assim a cada palavra digitada na interface gráfica, o algoritmo compila e assim mostra as palavras abaixo. Vale a pena destacar que para a interface integrar com o algoritmo, foi necessária uma mudança no código, e essa mudança foi para simular uma lista de um buscador, a cada palavra que foi selecionada, é colocado um caracter especial depois dela, e assim quando a interface detectar esse caracter especial, ela quebraria a linha fazendo com que

as palavras saiam em formato de lista. E assim, para visualizar o algoritmo integrado com a interface gráfica, deve seguir uma série de passos:

Devemos primeiramente instalar o XAMPP que pode ser encontrado no endereço abaixo.

https://www.apachefriends.org/pt_br/download.html

E deve-se escolher de acordo com a versão do Linux no computador a ser executado os códigos. Logo após de realizado o download, deve-se instalar o arquivo baixado. (Lembre-se de colocar as permissões de administrador, e de executar em Propriedades>Permissões>executar). Logo após a instalação do programa, deve-se primeiramente rodar o programa com o seguinte comando no terminal.

sudo /opt/lampp/lampp start

E assim, o programa estará executando. Para finalizar, deve-se colocar as pastas com as implementações no seguinte caminho:

/opt/lampp/htdocs

Nesse caminho o restante a se fazer é executar o programa dentro desse caminho e para visualizar ele deve-se digitar no navegador:

localhost/"Nome da pasta que contém o programa"

Exemplo: localhost/Heuristico

E assim é possível usar a interface gráfica.

7-CONCLUSÃO

A implementação do trabalho ocorreu em partes sem muitos problemas, mas alguns tópicos merecem atenção à serem debatidos. A heurística implementada foi baseada no algoritmo do KMP e ela apesar da definição da heurística apresentar um resultado aproximado, nesse caso ela apresenta um resultado exato e preciso para o problema, e em relação à parte gráfica, foi bastante interessante a realização, pois houve uma interdisciplinaridade com outras matérias do curso, nos fazendo pesquisar uma área não vista por nós previamente aumentando assim nosso conhecimento.

8-REFERÊNCIAS BIBLIOGRÁFICAS

- ROCHA, Leonardo C. D. ***Slides da unidade curricular de Algoritmos e Estruturas de Dados III***, 2016.