UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

HENRIQUE DAVID DE MEDEIROS

RELATÓRIO TÉCNICO DE ANÁLISE EMPÍRICA

ANÁLISE DE ALGORITMOS DE BUSCA

NATAL / RN

2018

HENRIQUE DAVID DE MEDEIROS

RELATÓRIO TÉCNICO DE ANÁLISE EMPÍRICA

ANÁLISE DE ALGORITMOS DE BUSCA

Trabalho realizado para a disciplina de Estrutura de Dados Básicas I do curso de Tecnologia da Informação da Universidade Federal do Rio Grande do Norte – UFRN com o proposito de demonstrar o funcionamento dos algoritmos de busca e da análise empírica.

NATAL / RN

RESUMO

O relatório apresenta uma análise sobre o funcionamento de 7 (sete) algoritmos implementados utilizando a linguagem C++. O trabalho pretende realizar uma apresentação sobre os tipos de algoritmos de busca e seus funcionamentos, atribuindo ênfase em 7 deles: o algoritmo de busca Linear, Binário (interativo e recursivo), Ternário (interativo e recursivo), Jump Search e Fibonacci Search. Além de apresentar gráficos apresentando como é o funcionamento de cada algoritmo analisando o tempo de execução pela quantidade de valores em um vetor de inteiros.

Sumário

1. Introdução

Algoritmos de buscas são muito importantes para realizar buscas das mais variadas, desde palavras/frases (string), à inteiros, como é o propósito deste trabalho. Quando é elaborado um algoritmo de busca deve ser analisado uma série de elementos, para que será utilizado, de que constituirá os elementos, quantidade de elementos, porém deve-se buscar sempre o algoritmo que seja mais eficiente, que possa retornar resultados cada vez mais rápidos para o usuário.

Os algoritmos de busca recebem um problema e retorna uma solução para o problema, principalmente após realizar uma série de soluções. Cada algoritmo possui uma solução diferente para resolver o mesmo problema, o da busca.

Este relatório apresenta uma análise para 7 (sete) tipos de algoritmos, nos quais são o da busca linear interativa, binária (interativa e recursiva), ternária (interativa e recursiva), jump search e fibonacci search. A análise retrata como o tempo pode agir em cada funcionamento, apresentando situações os quais pode haver algoritmos que sejam mais eficientes que outros.

2. Métodos

A execução dos algoritmos pode variar dependendo de várias situações, porém para obter resultados que sejam equivalentes aos obtidos nesse relatório devem ser tomadas uma série de situações.

2.1. Características Técnicas do Computador

Software de Virtualização	VIRTUAL BOX
Sistema Operacional (hospedeiro)	Windows 10 Pro
Sistema Operacional (virtual)	Ubuntu 17.10.1
Versão dos sistemas	64 bits
RAM (virtual)	2048 MB
Chipset	PIIX 3 com placa de vídeo de 16MB
Processador	Intel Core i5-2450M
CPU	2.50 GHz (2 núcleos e 4 processadores lógicos).

2.2. Linguagem de Programação

A realização dos algoritmos de buscas foi realizada a partir da linguagem de programação C++, a partir da utilização de bibliotecas e da estrutura básica da linguagem.

2.3. Compilador

O compilador utilizado é o GNU Compiler Collection(GCC), que é um conjunto de compiladores de linguagens de programação presente em sistemas baseados no Unix, utilizando a versão gcc 7.2.0, aplicando a versão para o C++ 11.

A linha de comando utilizando para a compilação foi:

```
g++ -Wall -std=c++11 NOME_DO_ARQUIVO.cpp -o NOME_DO_EXECUTAVEL
```

2.4. Importações

O projeto trabalha com certas importações de componentes para a compilação do programa ocorrer perfeitamente, entre eles se encontram as bibliotecas: iostream, para o

funcionamento de entradas e saídas do sistema, além de alertas de erros e quebras de linhas; chrono, para realizar a medição de tempo; algorithm, obter a contagem de tempo final; cmath, utilização da função sqrt() que serve para obter a raiz quadrada para utilização na Jump Search; string, para possibilitar o trabalho com o string para receber pelo terminal comandos de quantidade de amostras e tipo desejado de busca; sstream, possibilitar a leitura da string para poder haver conversão em inteiro; fstream, trabalhar com arquivos externos e geração de arquivos; vector, criar vetor para armazenar o padrão para os cabeçalhos; e iomanip, configurar padrões para a formatação do cabeçalho e dos valores de tempo e quantidade de elementos.

2.5. Algoritmos de Buscas

Os algoritmos de buscas possibilitam encontrar determinado elemento em um conjunto sequencial, dependendo do algoritmo é possível que haja uma eficácia maior em encontrar o elemento.

2.5.1. Busca Linear Interativa

A busca linear interativa, código 01, apresenta um algoritmo em que para encontrar um determinado elemento em um conjunto de elementos, há a necessidade de percorrer todo o vetor até encontrar o elemento. Ou seja, caso o elemento seja o primeiro item do vetor chama-se melhor caso, e caso o elemento não esteja no vetor, o pior caso. O código utiliza da estrutura for para percorrer o vetor, caso encontre o elemento retorne o índice, caso contrário retorna o final do vetor.

```
long int * linearSearch( long int *first , long int *last , int
value , long  int *default_last){

        for( auto i( first ); i < last ; i++ ){

                if( value == *i )
                        return i;


        }

        return default_last;
}
```

Código 01 – Busca Linear

2.5.2. Busca Binária

A busca binária é uma outra opção nas buscas de elementos em um vetor, porém os elementos devem estar ordenados (números), pois para a realização da busca faz

necessário dividir o vetor pela metade e então verificar se o valor é maior ou menor que o valor naquela região, sendo necessário repetir os passos neste novo intervalo, ou a metade é o valor procurado (melhor caso). A implementação da busca binária pode ser elaborada de duas formas, a interativa e a recursiva. Em resumo, a busca binária é implementada sempre dividindo o vetor na metade e verificando se o valor escolhido é a metade, caso contrário ocorre a repetição da divisão e comparação.

2.5.2.1. Busca Binária Interativa

No código 02, temos a busca binária interativa, a qual é implementada com mais linhas de código, porém toda a execução é realizada apenas em uma função, no qual executará um mesmo lanço de repetição várias vezes até percorrer o vetor.

```
long int * binarySearch( long int *first , long int *last , int
value , long int *default_last){

        auto back( last );

        while( first <= last ){
                int middle = ( last - first ) / 2;

                if(     *( first + middle ) == value ){

                        return first + middle;
                }

                if( *( first + middle ) < value ){
                        first = first + middle + 1;
                }
                else{
                        last = last - middle - 1;
                }
        }

        return back;

}
```

Código 02 – Busca Binária Interatica

2.5.2.2. Busca Binária Recursiva

A busca binária recursiva, código 03, funciona a partir de várias chamadas da mesma função, até encontrar o elemento desejado, o qual será retornado para a função principal.

```
long int * binary_rec( long int *first , long int *last , int
value , long int *default_last){

        if( first <= last ){

                int middle = ( last - first ) / 2;

                if( *( first + middle ) == value ){
                        return first + middle;
                }

                else if( *( first + middle ) < value ){
                        auto new_first = first + middle + 1;
                        return binary_rec( new_first , last ,
value , default_last );

                }
                else{
                        auto new_last = last - middle - 1;
                        return binary_rec( first , new_last ,
value , default_last );
                }
        }

        return default_last;

}
```

Código 03 – Busca Binária Recursiva

2.5.3. Busca Ternária

A busca ternária pode ser dividida em dois (2) tipos como na busca binária, a interativa e a recursiva. Os dois tipos de busca, a ternária e a binária, possuem características em comum, como a divisão da quantidade dos itens em 3 vezes, e realizar novamente o passo até encontrar o elemento, ou não o encontre, e a busca entre os elementos que pertencem aos valores correspondentes a divisão, gerando um novo subintervalo no intervalo principal. Em resumo, busca ternária dividimos o vetor em 3 (três), as quais devemos verificar se o valor é algum dos terços dos valores encontrados (melhor caso), ou faz partes dos intervalos: início do vetor e primeiro terço; primeiro terço e segundo terço; e segundo terço e final do vetor; e então realizar todo o procedimento novamente.

2.5.3.1. Busca Ternária Interativa

A busca ternária interativa, mostrado no código 04, apresenta toda a execução do programa, o qual não é necessário chamar novas funções para buscar o número. A busca ocorre a partir de condições de repetição.

```
long int * ternSearch( long int *first , long int *last , int
value , long int *default_last ){

        while( first <= last ){
                int middle1 = ( last - first) / 3;
                int middle2 = 2 * middle1;


                if( *( first + middle1 ) == value )
                        return first + middle1;

                if( *( first + middle2 ) == value )
                        return first + middle2;

                if( value < *( first + middle1 ) ){
                        last = first + middle1 - 1;
                }
                else if( *( first + middle1 ) < value && value <
*( first + middle2 ) ){
                        first = first + middle1 + 1;
                        last = first + middle2 - 1;
                }
                else if (value > *( first + middle2 ) ){
                        first = first + middle2 + 1;
                }
                else{
                        return default_last;
                }
        }

        return default_last;
}
```

Código 04 – Busca Ternária Interativa

2.5.3.2. Busca Ternária Recursiva

A busca ternária recursiva ocorre chamando várias vezes a função da busca, modificando os valores de início e final do vetor. A forma recursiva, código 05, permite ao código fica com menos linhas de código, porém o funcionamento corresponde ao da sua versão interativa como mostrado no tópico 2.4.3.1.

```
long int * tern_rec( long int *first , long int *last , int value
, long int *default_last ){


        if( first <= last ){
                int middle1 = ( last - first ) / 3;
                int middle2 = 2 * middle1;

                if( *( first + middle1 ) == value )
                        return first + middle1;

                if( *( first + middle2 ) == value )
                        return first + middle2;

                if( value < *( first + middle1 ) ){
                        return tern_rec( first , first + middle1 -
1 , value , default_last );
                }
                else if( *( first + middle1 ) < value && value <
*( first + middle2 ) ){
                        return tern_rec( first + middle1 + 1 ,
first + middle2 - 1 , value , default_last );
                }
                else if ( value > *( first + middle2 ) ){
                        return tern_rec( first + middle2 + 1 ,
last , value , default_last );
                }
                else{
                        return default_last;
                }
        }

        return default_last;
}
```

Código 05 – Busca Ternária Recursiva

2.5.4. Jump Search

A busca utilizando a Jump Search é realizada a partir de pequenas divisões das posições do vetor, como podemos observar no código 06, agrupando os elementos em grupos em que possuam a mesma quantidade de elementos e então realizar uma busca linear nesses subgrupos. Permite, assim, realizar uma busca linear só que em pequenos grupos.

```
long int * jump_search(long int *first, long int *last, int
value, long int *default_last){

        int aux = std::sqrt( ( last - first ) );

        long int* new_first = first;
        long int* new_last = first+aux;


        while( first <= last && new_last <= default_last ){

                if( value == *new_first )
                        return new_first;

                if( value == *new_last )
                        return new_last;

                if( value > *new_first && value < *new_last )
                        return linearSearch( new_first , new_last
, value , default_last );
                else{
                        new_first += aux;
                        new_last += aux;
                }
        }

        return default_last;

}
```

Código 06 – Jump Search

2.5.5. Fibonacci Search

A Fibonacci Search é um meio de busca em que utiliza os números de Fibonacci para encontrar o valor desejado. Primeiramente verifica se o número informado pertence ao conjunto dos números de Fibonacci, senão, identifica qual o número antes do número que será buscado e que pertença a sequência dos números de Fibonacci. E então é realizada uma divisão a qual permitirá buscar o número desejado com o auxílio da sequência. Pelo código 07, é possível perceber melhor como é realizado as operações e o funcionamento do algoritmo.

```
long int * fibonacci_search( long int *first , long int *last ,
int value , long int *default_last ){

        int fib_m2 = 0;
        int fib_m1 = 1;

        int fib = fib_m2 + fib_m1;
        int menor = 0 , aux = -1;

        while( fib <= *( last - 1 ) ){
                fib_m2 = fib_m1;
                fib_m1 = fib;
                fib = fib_m2 + fib_m1;
        }

        while( fib > 1 ){

                menor = menorValor( fib_m2 + aux , *( last - 2 )
);

                if( *( first + menor ) < value ){
                        fib = fib_m1;
                        fib_m1 = fib_m2;
                        fib_m2 = fib - fib_m1;
                        aux = menor;

                } else if( *( first + menor ) > value ){
                        fib = fib_m2;
                        fib_m1 = fib_m1 - fib_m2;
                        fib_m2 = fib - fib_m1;

                } else{
                        return first + menor;
                }

        }

        return default_last;
}
```

Código 07 – Fibonacci Search

2.6. Função Principal

O código 08 mostra como foi realizado a inserção dos dados no vetor, a leitura dos dados inseridos com o executável, cálculo da média, medição do tempo, abertura e inserção dos dados em um arquivo externo, é ele que controla todas as informações do vetor. O código apresentado neste tópico serve apenas como suporte caso haja execução de testes de terceiros.

```
int main(int argc, char* argv[]){

	long int quant_Element = 0; // acompanha a variação da
quantidade de elementos.

	/*
	* tipo_busca informa qual o tipo de busca desejada e o padrão
	* caso não seja informado o tipo de busca.
	*/
	std::string tipo_busca = "LN";
	int valor;

	/*
	* quant_max é a quantidade máxima de um vetor (varia de
computador em computador).
	*/
	long int quant_max_Element = 1026991;
	int amostras = 0; // Verifica a quantidade de amostras
desejadas.

	std::ofstream FILE;
	std::vector<std::string> cabecalho = {"Quantidade de
Elementos","TEMPO(ns)"};

	if(argc < 3){
		// Apresenta erro caso o usuário não tenha executado
corretamente o executável.
		std::cout << "Há informações faltando\n ./EXECUTAVEL
QUANTIDADE_AMOSTRAS NOME_DA_BUSCA" << std::endl;
	} else{

		// Transforma o valor da quantidade de amostras.
		amostras = getInteger(argv[1]);
		// Recebe o tipo de busca.
		tipo_busca = argv[2];

		// Cria arquivo com o nome do tipo de busca em formato
csv para gerar os gráficos.
		FILE.open("results_" + tipo_busca + ".csv");

		// Verifica se foi possível abrir o arquivo, caso
contrário apresenta erro e encerra o programa.
		if(FILE.fail()){
			std::cout << "Erro ao abrir o arquivo!" <<
std::endl;
			return -1;
		}

		// Insere o padrão do cabeçalho no arquivo de saída.
		FILE << "TIPO DE BUSCA: " << tipo_busca << ",
Quantidade de amostras analisadas:" << amostras << std::endl <<
std::endl;
		FILE << cabecalho[0] << "  ";
		FILE << ",  " << cabecalho[1] << "  ";
```

```
                FILE << std::endl;

                // Inicia a quantidade de elementos no vetor.

                int media_inicial = quant_max_Element / amostras;
                quant_Element = media_inicial;

                // Executa séries de repetições de amostras.
                for( auto rep(1); rep <= amostras; rep++){

                        // Reinicia a média em cada repetição.
                        int media = 0;

                        // Previne caso a quantidade de elementos for
maior do que suportado.
                        if(quant_Element > quant_max_Element)
                                quant_Element = quant_max_Element;

                        // Cria um vetor para armazenar elementos.
                        long int A[quant_Element];

                        // Insere elementos no vetor.
                        for( auto i(0); i < quant_Element; i++){
                                A[i] = i;
                        }

                        // Torna a variável valor não seja um elemento
do vetor.
                        valor = quant_max_Element+1;

                        // Realiza séries de testes dos tempos da
busca escolhida.

                        for( auto t(0); t < 100; t++){

                                auto start =
std::chrono::system_clock::now();

                                // Realiza a busca desejada.
                                if(tipo_busca == "FS")
                                        auto result = fibonacci_search(
A, A+quant_Element, valor, A+quant_Element );
                                else if(tipo_busca == "BI")
                                        auto result = binarySearch( A,
A+quant_Element, valor, A+quant_Element );
                                else if(tipo_busca == "BR")
                                        auto result = binary_rec( A,
A+quant_Element, valor, A+quant_Element );
                                else if(tipo_busca == "TI")
                                        auto result = ternSearch( A,
A+quant_Element, valor, A+quant_Element );
                                else if(tipo_busca == "TR")
                                        auto result = tern_rec( A,
A+quant_Element, valor, A+quant_Element );
                                else if(tipo_busca == "JS")
                                        auto result = jump_search( A,
A+quant_Element, valor, A+quant_Element );
                                else
```

```
                                        auto result = linearSearch( A,
A+quant_Element, valor, A+quant_Element );

                                    auto end =
std::chrono::system_clock::now();

                                    int waste_time =
std::chrono::duration_cast<std::chrono::nanoseconds>(end -
start).count();

                                        media += (waste_time - media) / rep;

                            }

                            //Insere os dados no arquivo.

                            FILE << std::fixed <<
std::setprecision(cabecalho[0].size()) <<
std::setw(cabecalho[0].size()) << quant_Element << ",  ";
                            FILE << "  " << std::fixed <<
std::setprecision(cabecalho[1].size()) <<
std::setw(cabecalho[1].size()) << media << "  ";
                            FILE << std::endl;

                            // Adiciona mais elementos no vetor.
                            quant_Element += media_inicial;

                    }

                    FILE.close();
            }

            return 0;
}
```

Código 08 – Função principal

2.7. Cenários de Simulação

O cenário de simulação é propor a execução de cada algoritmo apresentado nas seções 2.5 e 2.6, com diversos tipos de tamanho, de forma que a quantidade de elementos seja sempre crescente, até atingir o máximo que a máquina suporte, a quantidade de elementos máxima no vetor, no caso do projeto 1026950. Além de gerar 50 amostras, as quais serão realizadas uma média para obter um resultado médio. A organização das amostras é em formas crescentes, e as pesquisas sempre verificam o pior caso, em que não é possível encontrar o elemento.

2.8. Metodologias

As metodologias utilizadas foram a utilização de estruturas de repetições, para fazer uma quantidade de amostras informada pelo usuário, além de 100 execuções dos algoritmos para gerar uma média a qual possibilita um resultado mais concreto. Após obter a média, é inserido no arquivo externo o resultado da média e a quantidade de objetos que foram

inseridos no vetor. A medida do tempo foi utilizada a biblioteca chrono, a qual possibilita a obtenção do tempo do sistema (horário), e então basta obter no início e final do algoritmo, e então obter a diferença, obtendo assim o tempo de execução. Para a inserção de dados foi realizada uma estrutura de repetição começando do 0 (zero) até o número máximo de elementos que o vetor conseguiu, ou seja, 1026950, começando em 20539 e adicionando em cada repetição mais 19539, pela quantidade de amostras desejadas, no caso testado 50 amostras.

3. Resultados

Todos os valores apresentados nos gráficos correspondem taxa de quantidade de elementos (x) pela quantidade de tempo gasto para rodar o algoritmo (y), com unidade de tempo de nano segundos (ns) para haver uma comparação mais aprofundada do funcionamento de cada algoritmo. Na Tabela 01 é apresentado todos os dados que foram inseridos para a criação dos gráficos apresentados nesse trabalho, assim como todos os tempos gastos em execuções dos algoritmos solicitados, os resultados foram obtidos a partir da execução dos algoritmos e armazenados em arquivos em formato de tabela.

QNT ELEMENTOS	***TEMPO (ns)***						
	LINEAR	***BINÁRIA INTERATIVA***	***BINÁRIA RECURSIVA***	***TERNÁRIA INTERATIVA***	***TERNÁRIA RECURSIVA***	***JUMP SEARCH***	***FIBONACCI SEARCH***
20539	58844	196	308	249	252	794	335
41078	241426	191	319	287	267	1085	354
61617	147585	199	316	283	266	1337	368
82156	209665	212	345	264	249	1674	382
102695	258138	204	343	282	266	2304	382
123234	299325	193	347	297	326	2202	398
143773	342561	203	367	293	270	2413	393
164312	399030	200	367	296	285	2611	399
184851	885821	204	354	288	271	2790	398
205390	504644	200	369	304	339	3158	426
225929	556519	199	350	288	264	3442	423
246468	604182	197	353	286	273	3501	420
267007	666272	204	663	319	308	4220	415
287546	864747	206	386	315	416	5429	420
308085	785133	203	390	302	289	6269	414
328624	833929	204	385	296	272	6062	436
349163	892914	228	377	321	288	7232	427
369702	948768	202	362	282	296	7796	422
390241	1011094	200	364	317	310	9244	423
410780	1638878	197	360	278	268	9860	333
431319	1136295	218	357	6680	280	8782	337
451858	1190846	196	359	295	479	7019	327

472397	1239122	194	397	295	278	8227	327
492936	1293479	193	396	285	266	9395	327
513475	1431462	199	326	290	277	8358	324
534014	1407364	201	294	282	469	9261	361
554553	1438778	199	296	290	386	8608	426
575092	1479646	199	293	276	263	10472	381
595631	1603588	196	292	233	366	9468	333
616170	1573571	193	289	213	255	10103	326
636709	1665897	195	288	226	269	9884	328
657248	2008807	193	347	214	253	10620	332
677787	1760003	187	346	221	307	10347	324
698326	1799976	192	349	208	250	33623	422
718865	2145775	191	286	222	264	10335	336
739404	1855850	189	281	228	317	10645	357
759943	1878986	188	282	219	263	10675	316
780482	2110429	189	281	204	249	10882	314
801021	1968269	184	278	218	260	10762	313
821560	2016657	188	352	227	291	10392	317
842099	2121592	258	355	214	260	11321	330
862638	2079148	185	354	239	271	11814	325
883177	2460605	180	276	214	257	11571	344
903716	2210883	305	273	223	646	11224	316
924255	2733988	177	270	210	333	63293	317
944794	2297790	178	268	422	266	53226	312
965333	2316235	178	265	214	250	86415	325
985872	2421219	178	264	220	261	58585	310
1006411	2336147	176	454	207	245	48376	311
1026950	2534882	173	361	220	276	11902	302

Tabela 01 – Dados do resultado de cada busca de acordo com o tempo.

No gráfico 01 é apresentado o tempo gasto para realizar a busca no vetor utilizando a busca linear, podemos observar que quanto maior a quantidade de objetos dentro do vetor maior será o tempo gasto para realizar a busca de um elemento no vetor. Além de ser possível perceber o quanto a forma da linha do gráfico ficou variada, não resultando em uma análise constante, e principalmente é possível perceber que é bastante demorada.

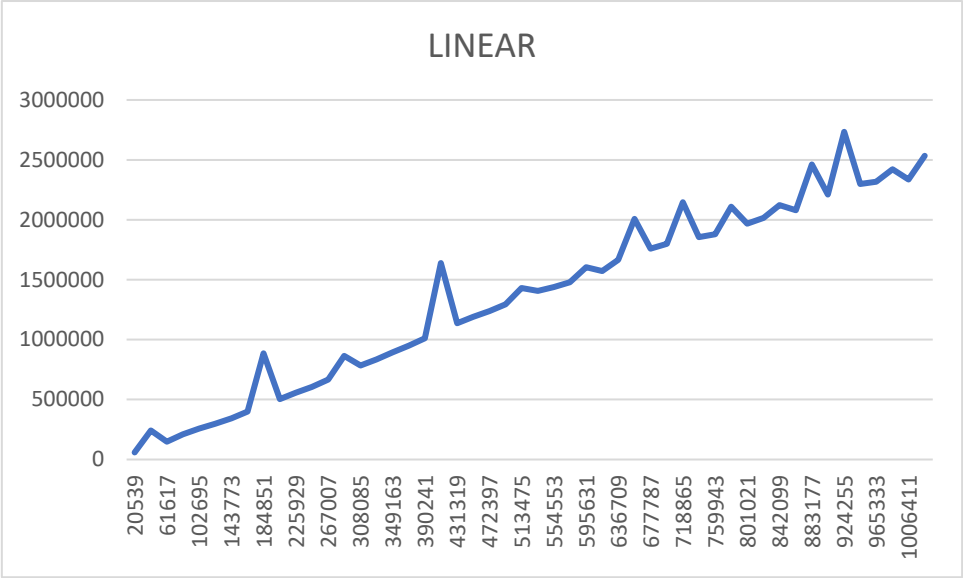

Gráfico 01 – Busca Linear

Uma boa opção para substituir o algoritmo da busca linear é a utilização do algoritmo da busca binária, Gráfico 02.1 e Gráfico 02.2. Em comparação entre o Gráfico 01, o Gráfico 2.1 e 2.2 é possível perceber claramente que a busca binária se torna mais eficiente do que a busca linear, pois o tempo é praticamente nulo se considerarmos a escala do Gráfico 01 (busca linear).

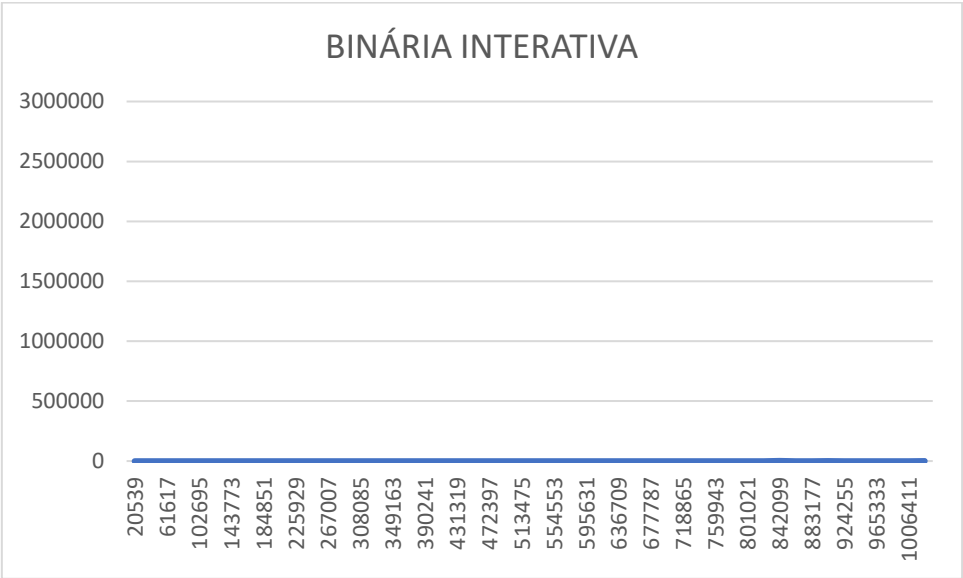

Gráfico 02.1 – Busca Binária Interativa

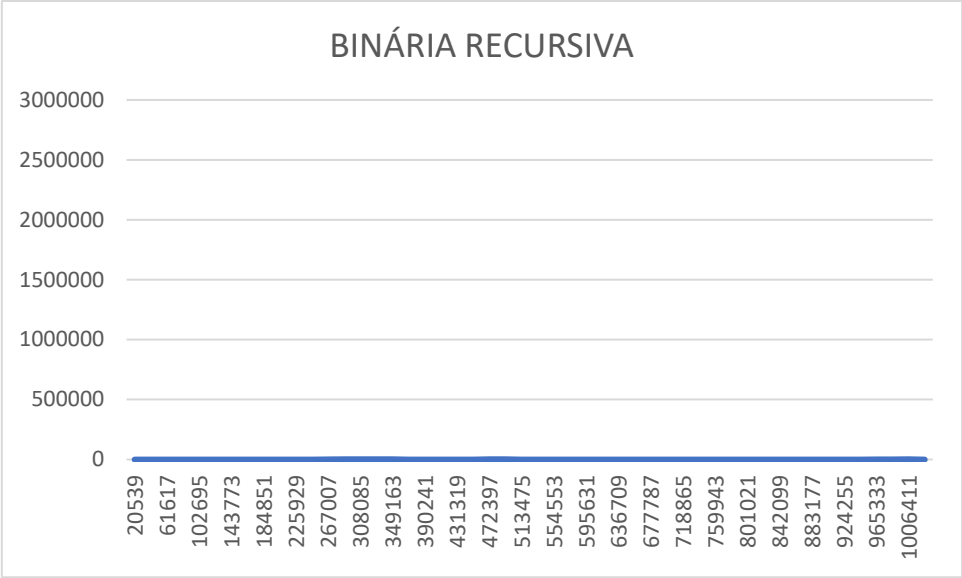

Gráfico 02.2 – Busca Binária Recursiva

Porém, para efeito de comparação, diminuindo a escala do gráfico, é possível analisar como é o comportamento assintótico de cada algoritmo, Gráfico 02.3. A busca linear não aparece no gráfico porque a quantidade inicial demora para executar, porém as buscas binária interativa e recursiva pertencem aproximadamente a mesma faixa de escala, sendo que a interativa se mostra mais vantajosa para a busca, já que apresenta o menor limite de tempo, enquanto a interativa se mostra mais eficiente apenas contra a linear.

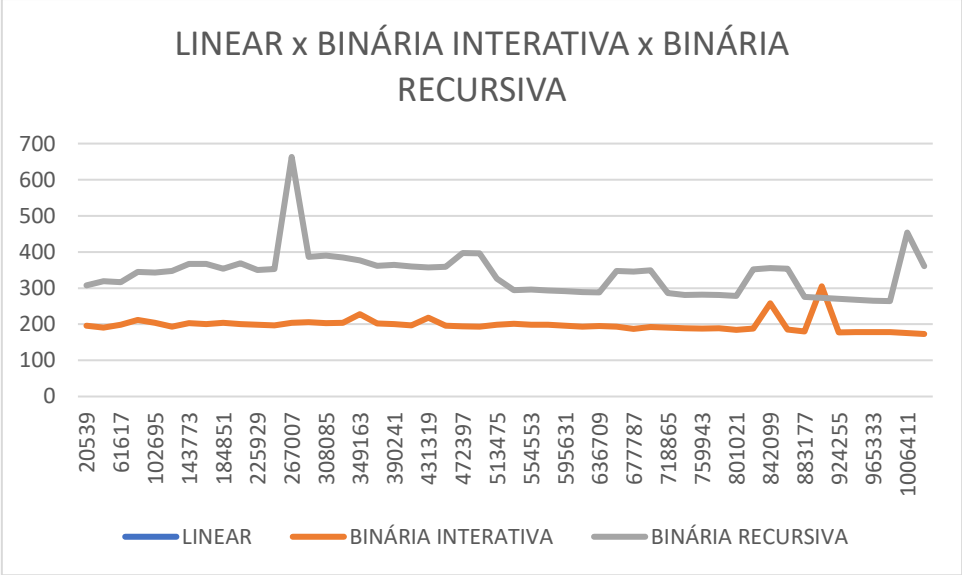

Gráfico 02.3 – Análise Assintótica da Busca Linear, Binária Interativa e Recursiva

A busca ternária interativa, Gráfico 03.1, apresenta um gráfico parecido com a da busca binária na escala da busca linear, assim como o Gráfico 03.2, busca ternária recursiva. Considerando uma comparação entre a ternária e a linear, é mais vantajoso realizar uma busca ternária, o qual analisará uma maior quantidade de elementos e retornará em poucos milissegundos.

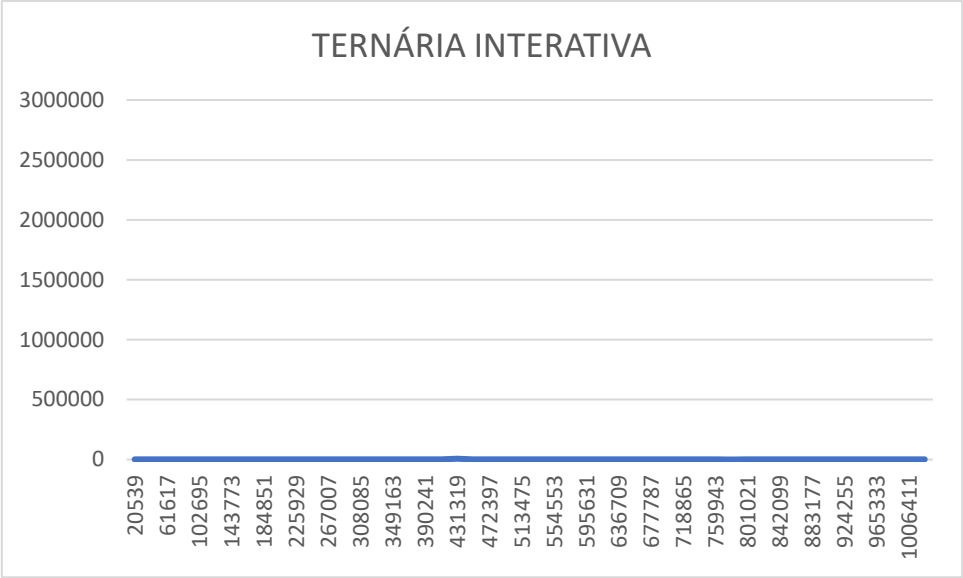

Gráfico 03.1 – Busca Ternária Interativa

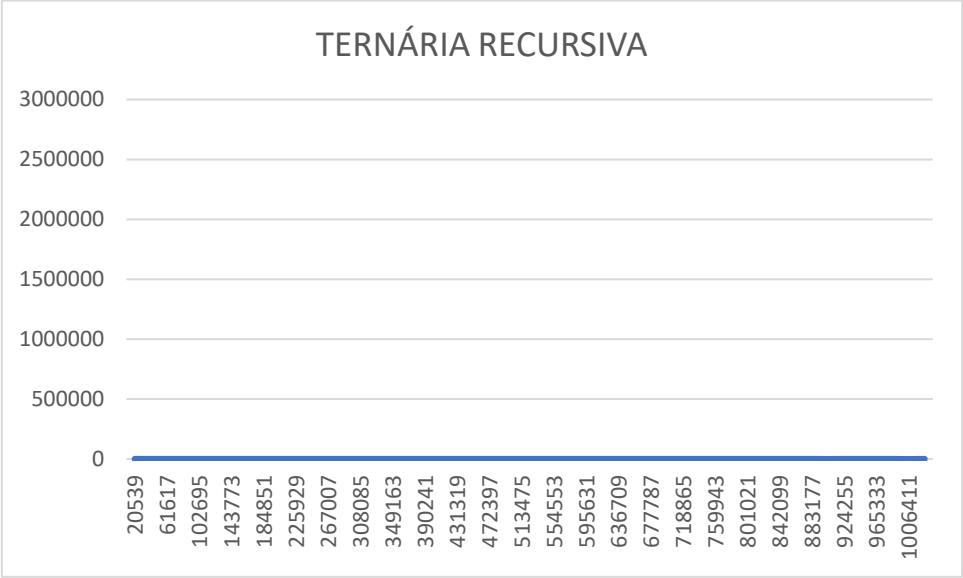

Gráfico 03.2 – Busca Ternária Recursiva

Observando o Gráfico 03.3 é possível verificar que as formas interativas e recursivas dos algoritmos da busca ternária são bastante condizentes com o seu tempo de funcionamento, já que ambos possuem a mesma ideia de implementação, porém é

possível verificar que diminuindo a escala, os gráficos se tocam e a interativa se mostra mais eficiente, desconsiderando o pico atingido entre o intervalo 390241 e 431319.

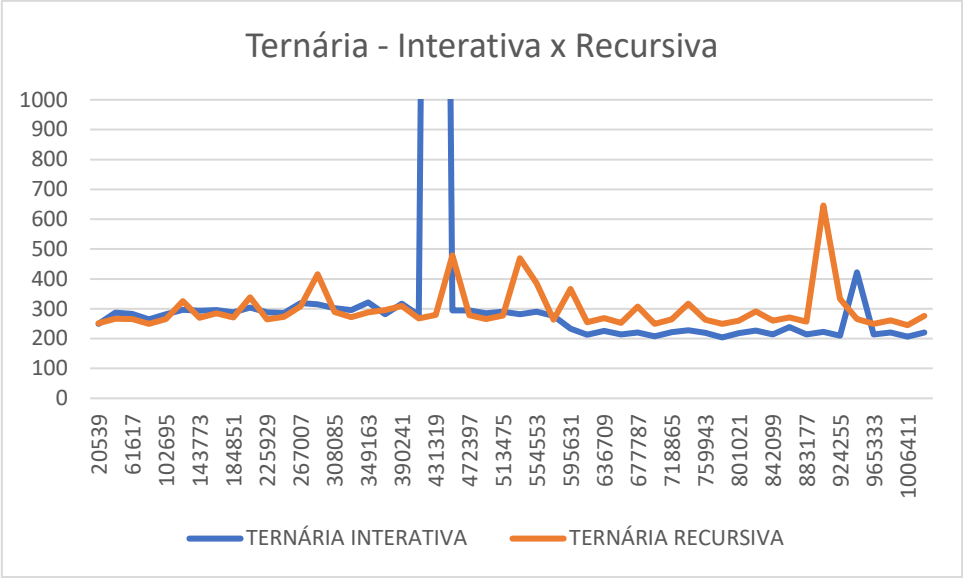

Gráfico 03.3 – Busca Ternária Interativa e Recursiva

O Gráfico 03.4 apresenta uma visão geral com os algoritmos relatados até o momento neste relatório. Novamente, o algoritmo da busca binária interativa sobressaiu entre os outros tipos de busca, ela apresenta a melhor eficiência, já que apresenta em um menor tempo, enquanto a linear é a mais ineficiente. O destaque fica para a ternária interativa que apenas perde para a binária interativa.

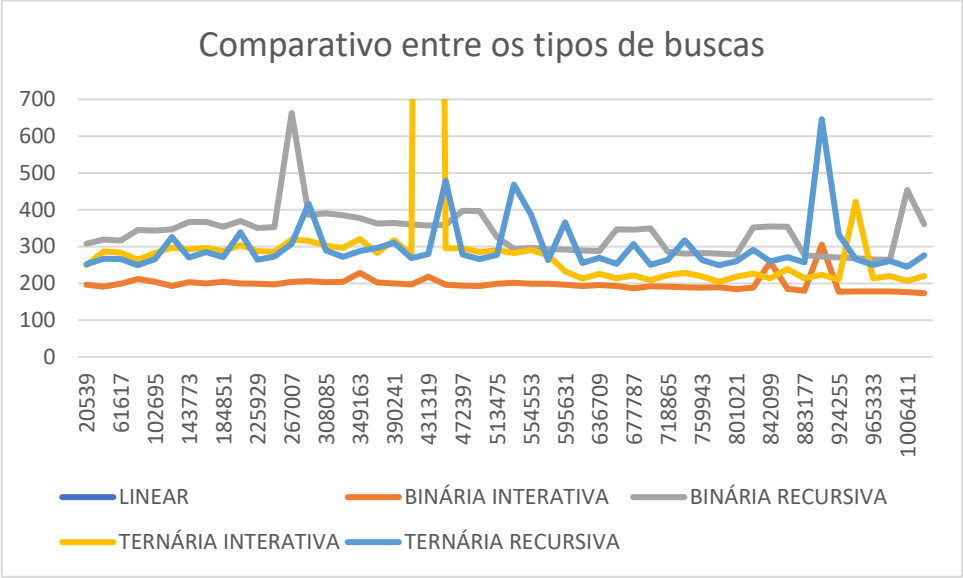

Gráfico 03.4 – Comparativo entre as buscas Linear, Binária interativa e recursiva e Terciária interativa e recursiva.

O método de busca Jump Search apresenta um gráfico (Gráfico 04) parecido com o da busca linear (Gráfico 01), principalmente porque este método utiliza como função secundária o método de busca linear. Porém, o tempo gasto para procurar o elemento foi muito menor do que na busca linear. Para melhor análise do Jump Search, o Gráfico 04.2 apresenta o formato do tempo em escala menor de tempo.

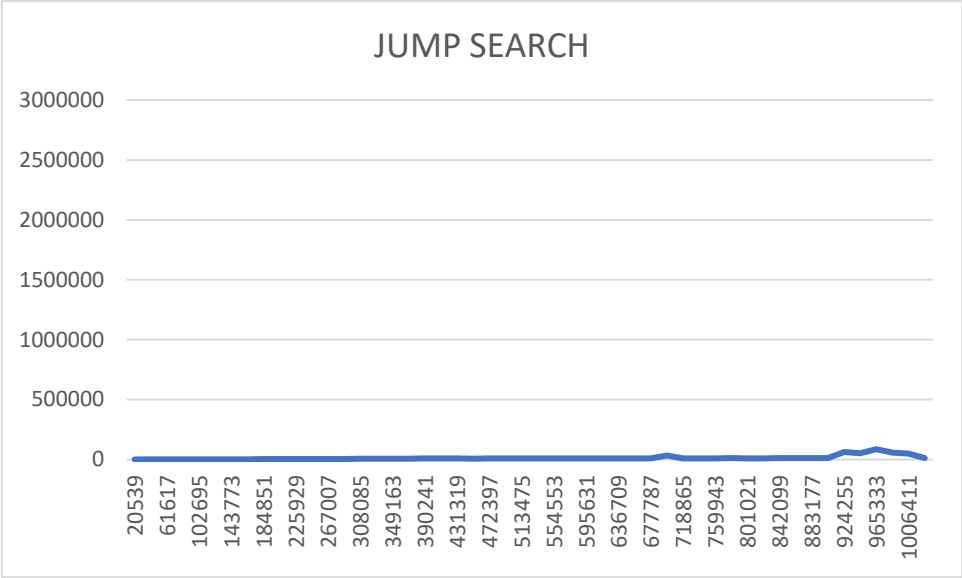

Gráfico 04.1 – Jump Search

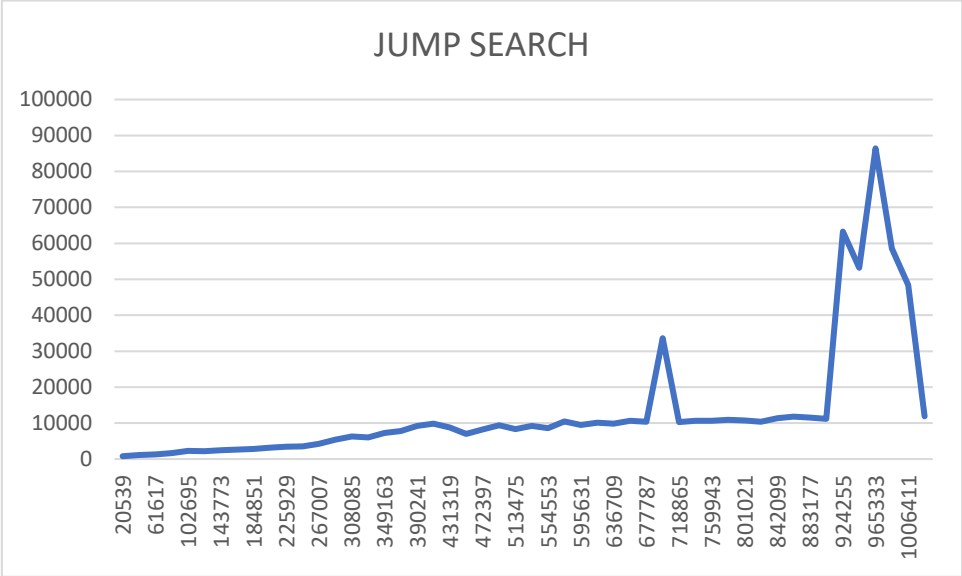

Gráfico 04.2 – Jump Search (menor escala de tempo)

O Gráfico 04.3 apresenta uma comparação entre os algoritmos até os algoritmos até o momento apresentado. É possível perceber que em uma comparação direta o Jump Search é mais eficiente do que a busca linear, qual não é possível ser notada pois o seu

termo é muito maior que os outros algoritmos de busca. Entretanto ela perde para os mesmos algoritmos que a da linear.

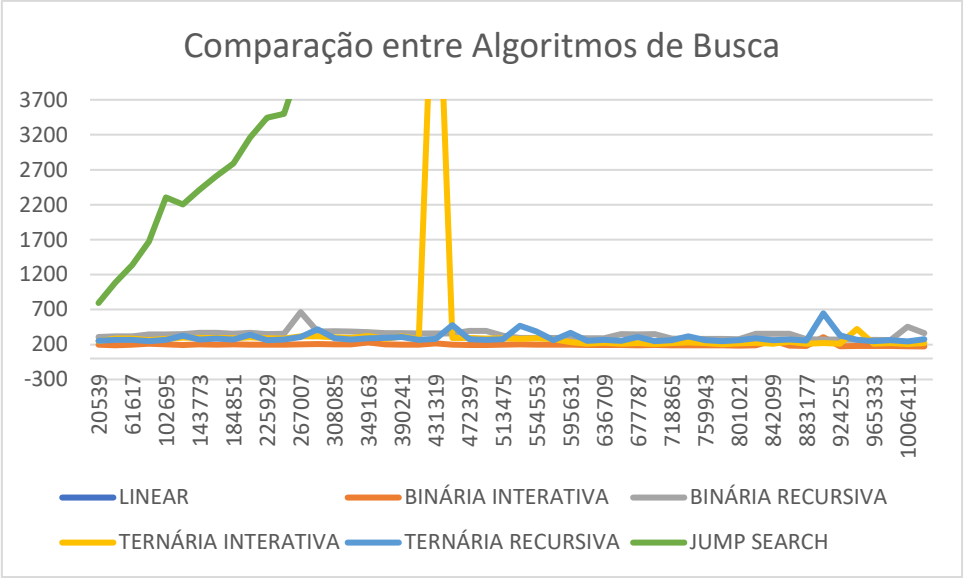

Gráfico 04.3 – Comparação entre os algoritmos de busca linear, binária interativa e recursiva, ternária interativa e recursiva e o jump search.

O Gráfico 05.1, apresenta uma outra solução de busca, analisando os dados é possível perceber que o Fibonacci Search é eficiente. É possível analisar, também, que há uma grande variação quanto ao tempo de execução com uma determinada quantidade de objetos, o qual pode tornar a busca instável.

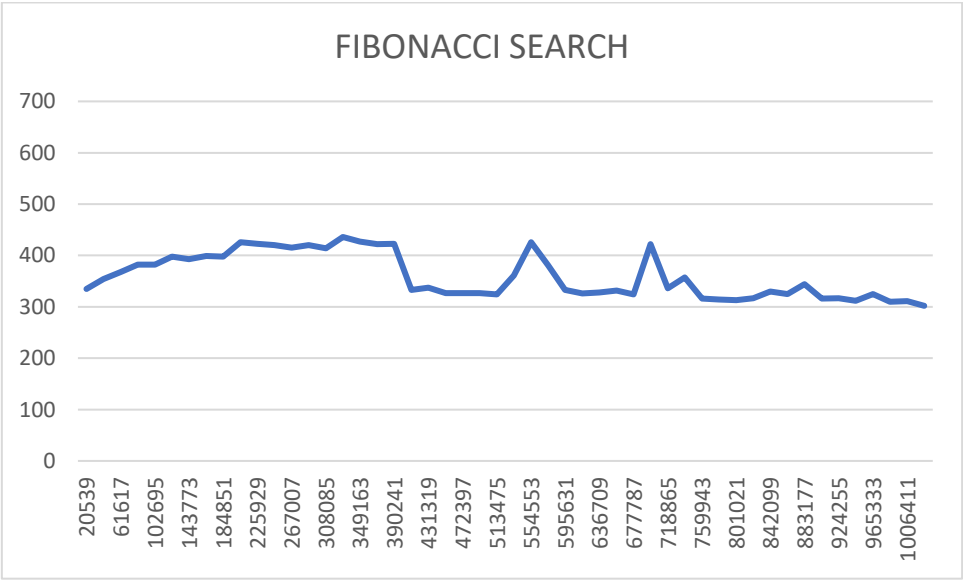

Gráfico 05.1 – Fibonacci Search

O Gráfico 05.2 apresenta uma comparação entre o Fibonacci Search e os outros tipos de algoritmos de busca. O algoritmo se mostra eficiente quando comparado com soluções como a Linear e a Jump Search, porém ineficiente em comparação aos outros algoritmos de busca.

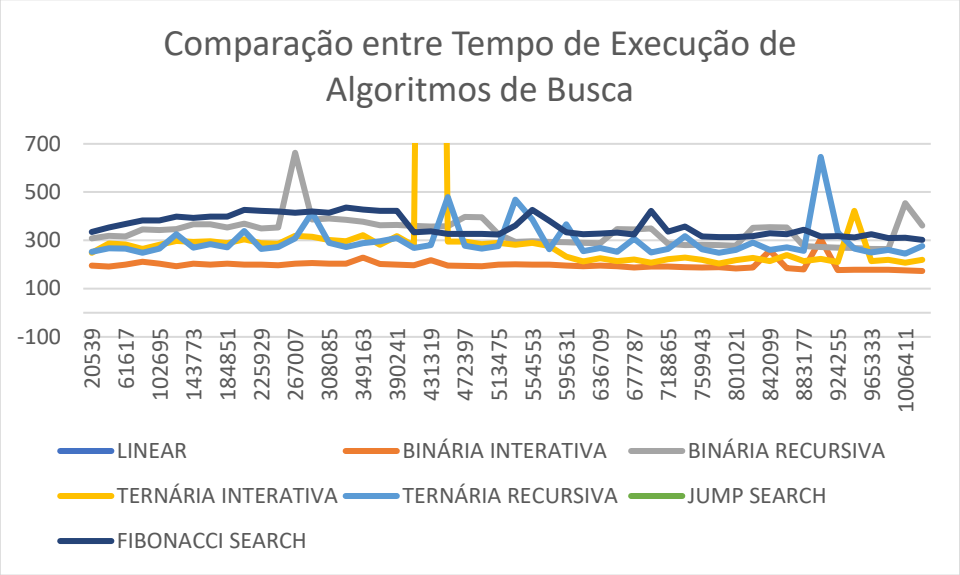

Gráfico 05.2 – Comparação entre todos os algoritmos de busca apresentados neste relatório.

O Gráfico 06 apresenta uma visão geral a todos os algoritmos de busca, apresentando assim que o algoritmo menos eficiente em relação ao tempo é o Linear, o qual demora mais tempo para buscar os elementos. E o mais eficiente é o da busca linear como é possível ser visto no Gráfico 05.2.

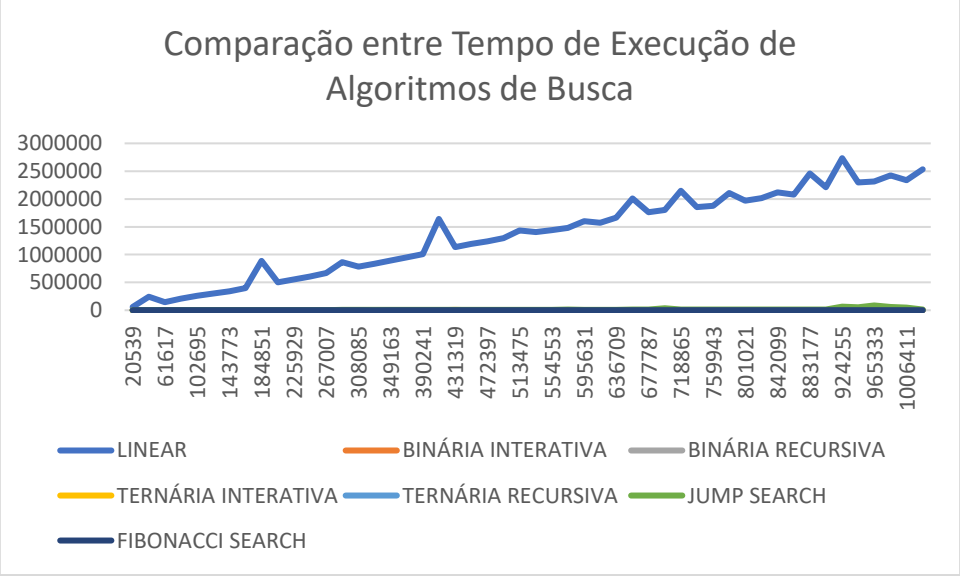

Gráfico 06 – Comparação entre todos os algoritmos de buscas

É possível verificar que o algoritmo mais eficiente com o tempo como referência é a busca binária interativa, a qual permite a busca em uma menor quantidade de tempo, seguida pela busca ternária interativa e a ternária recursiva. Com a execução dos algoritmos de busca binária (interativa e recursiva), ternária, (interativa e recursiva) e o Fibonacci Search, é possível verificar que eles possuem certas características parecidas, todos eles estão retornando resultados com pouca diferença do anterior, ou seja, o seu tempo de execução não são linear, ou quadrática, são um pouco constantes, principalmente se considerarmos a busca binária interativa, que apesar dos intervalos de pico em comparação com os outros se mantém constante.

Analisando os Gráficos 07.1 e 07.2 é possível analisar como ocorre o tempo quando com pequenas quantidades de tempo. A linear por mais que com uma menor quantidade de elementos, ainda apresenta grande quantidade de tempo para realizar a busca, enquanto o algoritmo da binária interativa apresenta a melhor eficiência. Ampliando o gráfico (Gráfico 07.2) observamos como se comporta cada busca em relação ao tempo com uma quantidade abaixo de 10000 elementos dentro de um vetor.

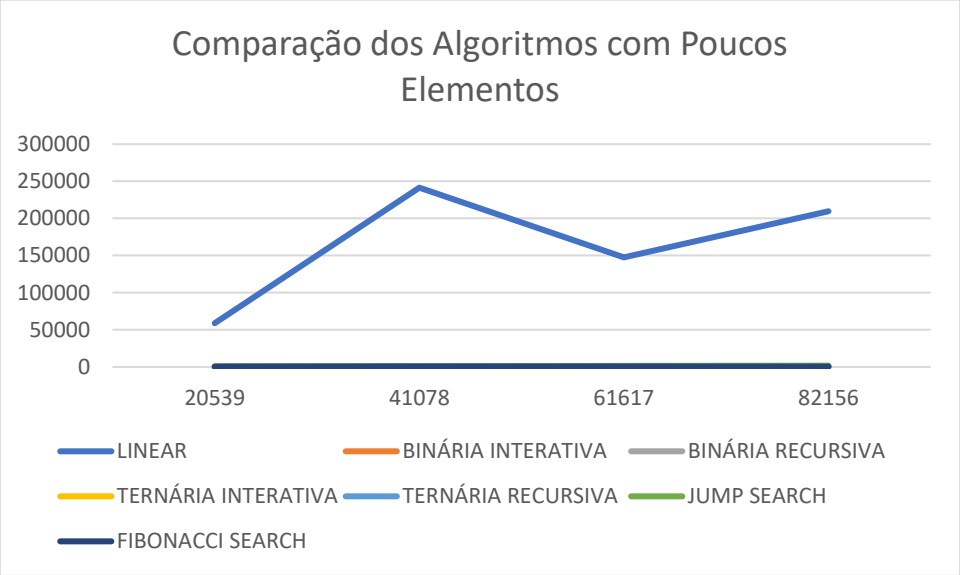

Gráfico 07.1 – Comparação entre algoritmos com poucos elementos.

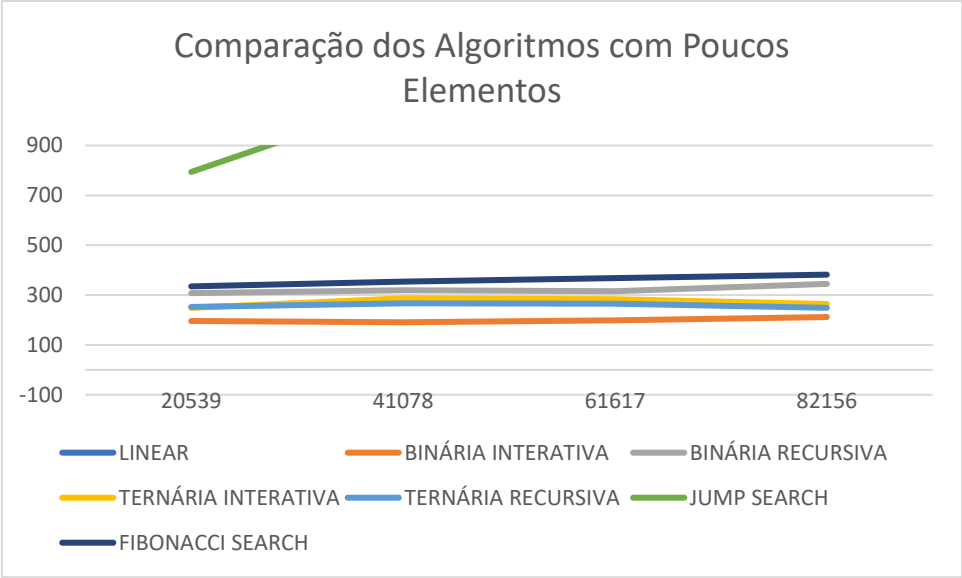

Gráfico 07.2 – Comparação dos Algoritmos com Poucos Elementos (escala reduzida).

4. Discussão

Os algoritmos de buscas são essenciais para os mais variados tipos de situações, principalmente quando se tem grande quantidade de dados em listas, ou matrizes, porém cada algoritmo deve ser utilizado para certas funções específicas e que nem sempre há apenas uma maneira de se realizar buscas.

A utilização de um algoritmo depende da quantidade de objetos que você irá utilizar, para situações os quais há poucos elementos dentro de uma lista, não se faz necessário desenvolver um algoritmo de busca binária, já que um simples algoritmo de busca linear resolve o problema com praticamente (ler diferente de igual) o mesmo tempo da busca binária, porém muito mais simples. Mas, quando estamos trabalhando com grande quantidade de dados, é necessário procurar um algoritmo que encontre o objeto, porém não demore muito para encontra-lo, é aqui que entra o caso da busca binária ser mais eficiente, em que encontra grandes quantidades de elementos em muito menos tempo do que o linear. Porém não podemos descartar as outras soluções como a busca ternária, que sua versão interativa é mais eficiente do que a busca binária recursiva.

Os gráficos gerados apresentaram várias variâncias em termos de tempo com o aumento da quantidade de elementos dentro do vetor, muitas vezes apresentando picos, o qual foram provocados pela busca e alocação na memória dos elementos/vetores desejados, pois é necessário criar um vetor para alocar os elementos e então realizar a busca, o qual pode causar queda no processamento, afetando assim o tempo nas buscas. Após a geração dos gráficos é possível realizar uma análise que ocorreram picos nas medições, os quais foram gerados pelo fato da alocação dos dados e da busca, como a solução é organizada para sempre executar o pior caso, ou seja, não encontrar o elemento, há momentos em que percorremos grande parte do vetor, como a binária que analisa por divisões. Quando considera que ele deve procurar por partes, há momentos em que o tempo acaba sobressaindo, sendo necessário tempo para procurar o vetor e procurar nele.

Analisando o processamento de cada algoritmo, é possível analisar que a busca linear consome em média 90,4% de CPU, binária interativa, 6,8% e recursiva com 1,0%, enquanto a ternária consome 2,9% na interativa e 4,9% na recursiva, jump search 3,9% e Fibonacci search 2,9%. Ou seja, os algoritmos que realizam divisões possuem melhor ganho de processamento, quando dividirmos o arranjo em 3 partes é possível perceber que o gasto com processamento foi apenas de 2,9% na ternária interativa e 4,9% na recursiva. Porém, quanto mais dividimos o arranjo maior será a quantidade de elementos que serão necessários verificar, caso ocorra de chegar ao limite de divisões, ou seja, encontrar 1 elemento, o qual não é mais possível dividir, o resultado será uma busca linear, pois será necessário percorrer todos os elementos, tornando assim o algoritmo ineficiente, gastando mais poder de processamento e mais tempo.

Cada tipo de busca pode ser representado por uma função matemática diferente, porém não há uma representação totalmente correta analisando os gráficos gerados pelos algoritmos, pois há pequenas variâncias, as quais não permitem que haja uma função que gere o gráfico como foram descritas neste relatório. Analisando o gráfico da busca linear, é possível perceber que a sua função é do tipo $f(x) = x$, pois cresce de acordo com a quantidade de objetos, sempre na forma linear, considerando grande quantidade de elementos. As buscas: binária interativa, ternária interativa, jump search e Fibonacci

search possuem gráficos de funções constantes, $f\ (x)\ =\ y$, com $\ y\ \in \mathbb{Z}$, enquanto a busca binária recursiva e a ternária recursiva, apresentam gráficos de curva na forma de $f(x) = \sqrt{x}$. É possível estimar o tempo necessário para cada algoritmo executar 100 milhões de elementos, basta calcular o tempo para executar uma repetição do processo da estrutura de repetição e então multiplicar pela quantidade de itens desejados, assim podemos supor quanto será o tempo.

A análise empírica proporciona uma grande facilidade para verificar se um algoritmo é eficiente ou não, porém análise matemática se torna mais favorável para casos de análise rápida. É possível afirmar que ambas as análises se correlacionam, visto que ambas permitem obter a eficiência de um algoritmo, a empírica pode apresentar discordância dependendo do meio e como o algoritmo foi desenvolvido, ou seja, sistema operacional, configurações de computador, laços de repetição, etc. Já a análise matemática apresenta um resultado mais objetivo, não podendo haver discordância, mantendo assim um padrão.

A proposta de verificar 50 amostras neste projeto é obter uma média que não seja extremamente detalhada, já que haveria casos em que a busca linear seria de difícil comparação com as outras buscas, assim haveriam resultados gráficos com grandes distâncias na representação de termo a termo. Em contrapartida, valores menores de amostras apresentariam resultados desconexos, de forma que não apresentariam resultados com realidade da situação de cada algoritmo. Caso os gráficos apresentassem valores uniformes, ou seja, valores parecidos, haveria a necessidade de aumentar a quantidade de amostras, porém como cada algoritmo apresenta resultado diferentes, uma amostra pequena já apresenta resultados concretos.

3. Referências

Agarwal, H. (14 de Dezembro de 2016). *Jump Search*. Fonte: GeeksforGeeks: https://www.geeksforgeeks.org/jump-search/

Unknown. (Julho de 2013). *Fibonacci search technique*. Fonte: Wikipedia: https://en.wikipedia.org/wiki/Fibonacci_search_technique

Varyani, Y. (09 de Dezembro de 2015). *Fibonacci Search - GeeksforGeeks*. Fonte: GeekforGeeks: https://www.geeksforgeeks.org/fibonacci-search/