UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Estrutura de Dados 2
SCC0606/2022.1

ANÁLISE DE ALGORITMOS DE BUSCA

Gustavo Moura Scarenci de Carvalho Ferreira - 12547792
Hugo Hiroyuki Nakamura - 12732037
Matheus Henrique Dias Cirillo - 12547750

Docente responsável: Prof. Robson L. F. Cordeiro

São Carlos
1º semestre / 2022

SUMÁRIO

1. INTRODUÇÃO

A implementação se deu na linguagem C utilizando o sistema de compilação ordenada *make* tanto para o código principal quanto para testes e para a análise empírica de complexidade dos algoritmos. Isso se fez pela separação dos arquivos em dois tipos: o **main.c**, que contem o código que chama os algoritmos e quantas vezes cada um será executado e os arquivos de implementação dos algoritmos.

Utilizando *make*, todos os arquivos de implementação e **main.c** são unidos em um executável, que pode ser utilizado com *"./build/main"* ou com *"make run"*. O **Makefile** também tem outras utilidades, como compilar o código todo com informação de depuração, limpar todos os arquivos e criar o zip final do projeto.

O código é implementado no diretório **"src/"** onde os 7 exercícios são divididos em 6 arquivos com o nome no padrão: **"exercicio*<NUMERO>*.c"**. Dessa forma temos que:

- Parte I:
  - a): **exercicio1a.c**
  - b): **exercicio1b.c**
  - c): **exercicio1c.c**
  - d): **exercicio1d.c**
- Parte II:
  - a): **exercicio2ab.c**
  - c): **exercicio2c.c**

Nesses arquivos além dos algoritmos pertinentes para cada proposta também são feitos os cálculos de custo temporal para cada iteração pedida em **main.c**

O diretório **"src/"** também conta com arquivos auxiliares como o **timeControl.c**, que tem funções para auxiliar na contagem do custo temporal da execução de cada iteração dos algoritmos de busca, e o **"utils.c"**, onde se têm funções que auxiliam no projeto em geral.

Tem-se o diretório **"res/"**, onde estão salvos os arquivos **".txt"** com as listas de números e textos a serem utilizadas como entrada e consulta.

E por fim, o diretório **"out/"**, onde foram salvos os **".csv"** brutos obtidos na coleta de dados, além dos gráficos e dos **".csv"** a serem analisados no relatório.

2. DESCRIÇÃO DO SOFTWARE

2.1. Alterações no *template*

O *template* [1] disponibilizado pelo Prof. Dr. Robson é muito bem feito e auxilia bastante no trabalho. Entretanto, apesar de serem sete arquivos, na prática, temos apenas dois padrões com muitas funções auxiliares repetidas. Para simplificar o código e permitir que alterações fossem mais simples de serem feitas, todos os códigos repetitivos foram transferidos para **utils.c** e as funções **main** de cada arquivo foram transformadas em uma função principal responsável por medir o custo temporal da execução do algoritmo pedido no exercício. Buscando simplificar ainda mais o programa e facilitar a leitura, todas essas funções principais de cada arquivos foram definidas no mesmo **.h**.

As funções de auxiliares de cálculo temporal não foram inseridas em **utils.c**, e sim em **timeControl.c** para deixar o programa mais legível.

O sistema de leitura de dados proposto no *template* [1] não funciona muito bem com o método de várias iterações repetidas para o cálculo médio do tempo. Ele é lento em comparação com uma simples cópia de vetores. Portanto, apenas uma leitura dos arquivos de entrada é feita, e subsequentemente copiado (para evitar alterações indevidas no original) para o vetor que será usado pelo algoritmo de busca.

2.2. Tipos Abstratos de Dados

2.2.1 Lista encadeada

Para fazer o armazenamento de um conjunto de informações ilimitadas é necessária uma lista dinâmica, portanto foi implementado um TAD de lista encadeada com inserção ordenada.

Essa TAD foi originalmente feita como um trabalho para a matéria de Estrutura de Dados e está implementada em **linkedList.c**, e é uma lista encadeada de nós com funções para imprimir os seus valores, inserir ordenado, inserir no final, achar um valor na lista de forma sequencial e desalocar a lista da memória.

2.3. Scripts em Python

Para auxiliar com o processamento de dados e a validação dos resultados encontrados foram montados quatro *scripts* em **Python 3.9.7**.

O código **palavrasEncontradas.py** é simplesmente uma busca linear das palavras em **string_busca.txt** que estão em **string_entrada.txt** retornando o valor de palavras encontradas para validar os resultados encontrados no exercício 2.

Seguindo a mesma ideia de validação, o código **palavrasUnicas.py** retorna quantas palavras únicas existem em **string_entrada.txt**, e se subtrairmos esse valor do total de palavras no arquivo temos um valor para o mínimo possível de colisões que um algoritmo de espalhamento pode gerar.

Os *scripts* de processamento de dados se utilizam das bibliotecas **pandas**, **matplotlib** e **numpy** para criar as tabelas e gráficos pedidos pelo professor. Sendo **simpleSearchCalculator.py** responsável pelo exercício 1 e **hashSearchCalculator.py** pelo exercício 2.

2.4. Busca sequencial simples

O arquivo **exercicio1a.c** para a *busca sequencial simples* [2] conta com apenas uma função (além da principal):

- **buscaSequencial**, que tem como parâmetros um vetor, seu tamanho e o valor que deseja ser buscado. O algoritmo funciona por iterar entre todos os elementos e comparar com o valor procurado, caso encontre a função retorna **VERDADEIRO**, sendo adicionado um elemento ao contador de encontrados.

2.5. Busca mover-para-frente

O arquivo **exercicio1b.c** para a *busca sequencial mover-para-frente* [3] conta com duas funções:

- **moverParaFrente**, que tem como parâmetros, um vetor e o índice do valor que será movido para frente. Nessa função o elemento na posição **i** é movido para a primeira posição da lista;
- **buscaSequencialMoverFrente**, que tem como parâmetros um vetor, seu tamanho e o valor que deseja ser buscado e funciona por iterar entre todos os elementos e comparar com o valor procurado, caso encontre a função **moverParaFrente** é chamada, e por fim a função retorna **VERDADEIRO**, sendo adicionado um elemento ao contador de encontrados.

2.6. Busca por transposição

O arquivo **exercicio1c.c** para a *busca sequencial por transposição* [3] conta com duas funções:

- **trocar**, que tem como parâmetros um vetor e os índices dos valores invertidos. A função inverte de posição os elementos nos índices passados como parâmetro;
- **buscaSequencialTransposicao**, que tem como parâmetros um vetor, seu tamanho e o valor que deseja ser buscado. O algoritmo funciona por iterar entre todos os elementos e comparar com o valor procurado, caso encontre a função **trocar** é chamada e inverte o elemento encontrado com o diretamente anterior a ele. Por fim a função retorna **VERDADEIRO**, sendo adicionado um elemento ao contador de encontrados.

2.7. Busca sequencial por índice

O arquivo **exercicio1d.c** para a *busca sequencial por índice* [4] conta com três funções além da principal:

- **quicksort**: recebe como parâmetros um vetor de inteiros, seu começo e final. Esse é um algorítimo de ordenação, pois para a realização dessa busca na lista, essa precisa estar ordenada. Qualquer algorítimo de ordenação poderia ter sido escolhido, mas foi esse o escolhido, pois costuma ser bem eficiente.
- **createTable**: recebe o número de índices que a tabela possuí, o vetor com os números a serem procurados e o número de palavras por índice. Essa função cria a tabela com os índices usados na busca.
- **buscaSequencialIndice**: recebe o vetor com todos os números onde a busca será feita, já ordenados pelo **quicksort**, o vetor com os números a serem buscados, a tabela de índices, o número de palavras por índice e o índice na lista de números a serem buscados do número atual, já que essa função é repetida para todos os números de uma lista de valores a serem buscados. Essa função primeiro acha em qual índice o número a ser buscado está, depois busca o número apenas entre os números daquele índice e retorna **1** caso encontre ou **0** caso não encontre.

2.8. Busca por hash fechado (Overflow Progressivo e Hash Duplo)

O arquivo **exercicio2ab.c** contêm a resolução dos exercícios de *Overflow Progressivo e Hash Duplo* [5] e conta com 9 funções além da principal **ex2ab**:

- **core**: recebe como parâmetros um ponteiro para função de hash que será usada, a lista de palavras a serem colocadas na tabela hash, a lista de palavras a serem procuradas na tabela hash e um *"id"* que identifica a função hash atual por um nome determinado, útil para fazermos arquivos de saída. Essa função é responsável por chamar todas as funções que vão montar a tabela hash, procurar palavras dentro dela e destruir a tal. É chamada dentro da **função principal** uma vez para cada função de hash que será usada.

- **h_div_closed**: tem como parâmetros uma *string* e um inteiro que conta a interação. Essa função gera um *hash* a partir da conversão do texto para um valor numérico e da interação atual e tira seu modulo com o tamanho da tabela *hash*. Como o método usado aqui é **Overflow Progressivo**, caso ocorra uma colisão, a palavra tentará ser inserida na próxima casa, e assim em diante, por isso a importância do contador de interação.

- **h_mul_closed**: tem como parâmetros uma *string* e um inteiro contador de interações. Essa função gera um *hash* a partir da conversão do texto para um valor numérico, do contador de interações e do módulo *float* entre a multiplicação desse valor e uma constante e 1. Devido ao **Overflow Progressivo** usado, caso ocorra uma colisão, a palavra tentará ser inserida na próxima casa, e assim em diante, por isso a importância do contador de interação.

- **h_improved_closed**: tem como parâmetros uma *string* e um inteiro contador de interações. Essa função gera um *hash* a partir da conversão do texto para um valor numérico utilizando-se da posição relativa das letras e a partir do contador de interações. Novamente, o contador de interações faz-se necessário devido ao método de tratar colisões **Overflow Progressivo** que será usado aqui.

- **h_duplo**: tem como parâmetros uma palavra e um inteiro contador de interações. Essa função gera um *hash* a partir da conversão do texto para um valor numérico, então esse valor é inserido na função **h_mul_closed** com o iterador e somado ao mesmo valor inserido na função **h_div_closed** com o iterador e depois multiplicado por ele, após tudo isso feito, é feito o modulo do resultado com o valor do número de *buckets* da tabela *hash*.

- **createHashTable**: não recebe parâmetros e retorna um ponteiro de ponteiro de caracteres que será a nossa tabela. Essa função é chamada dentro de **core** e é responsável por criar a tabela hash, deixando cada uma de suas casas com o valor nulo.

- **insertAtHashTable**: recebe como parâmetro um ponteiro para função de hash que será usada, a tabela hash a ser populada, a palavra a ser inserida, e um endereço de uma variável inteira onde será armazenado o número de colisões. Essa função é responsável por colocar uma palavra na tabela hash sendo chamada por **core** uma vez para cada palavra a ser colocada usando o retorno da função *hash* como indíce.

- **findAtHashTable**: recebe como parâmetro um ponteiro para a função de hash a ser usada, a tabela hash a ser pesquisada e uma palavra a ser procurada na tabela. Retorna 1 caso a palavra inserida for encontrada na tabela e 0 caso contrário, é chamada por **core** uma vez para cada palavra a ser pesquisada usando a função *hash*.

- **freeHashTable**: recebe como parâmetro a tabela hash e destroí essa para evitar o vazamento de memória. É chamado pela função **core** assim que a tabela já foi utilizada para todos os seus propósitos.

2.9. Busca por hash aberto

O arquivo **exercicio2c.c** para a *busca por hash aberto* [5] conta com cinco funções além da principal:

- **printHashTable**, que tem como parâmetros uma lista de lista de nós e uma *string* de localização de um arquivo. A função imprime em um arquivo de texto quais são as *strings* que estão em cada *hash*.

- **hashTester**, que tem como parâmetros um ponteiro para uma função, duas listras de strings e um identificador. A função funciona conforme o *template* proposto pelo professor, mas simplifica a chamada utilizando um ponteiro para função visando não repetir mais de uma vez o mesmo código alterando apenas uma função.

- **h_div**, que tem como parâmetros uma *string* e o tamanho da tabela *hash*. Essa função gera um *hash* a partir da conversão do texto para um valor numérico e tira seu modulo com o tamanho da tabela *hash*.

- **h_mul**, que tem como parâmetros uma *string* e o tamanho da tabela *hash*. Essa função gera um *hash* a partir da conversão do texto para um valor numérico e o módulo *float* entre a multiplicação desse valor e uma constante e 1.
- **h_improved**, que tem como parâmetros uma *string* e o tamanho da tabela *hash*. Essa função gera um *hash* a partir da conversão do texto para um valor numérico utilizando-se da posição relativa das letras.
- **h_duplo**, que tem como parâmetros uma *string* e o tamanho da tabela *hash*. Essa função gera um *hash* a partir da conversão do texto para um valor numérico, então esse valor é inserido na função **h_mul** e somado ao mesmo valor inserido na função **h_div** após tudo isso feito, é feito o modulo do resultado com o valor do número de *buckets* da tabela *hash*.

3. ANÁLISE

3.1. Busca sequencial

Como se pode observar pelos dados brutos coletados (Tabela 9, Tabela 3, Tabela 5 e Tabela 5) todas as execuções de todos os algoritmos encontraram 35 557 elementos. Tal valor está condizente com o que seria esperado da base de dados original e bate com o número de elementos encontrados por outros grupos.

Para cada variação de busca sequencial foram executados **52** testes e excluídos o maior e o menor tempo em uma tentativa de remover pontos de dados muito discrepantes. Com isso trabalhamos com **50** pontos de dados para cada algoritmo de busca sequencial.

Os resultados de custo temporal foram condizentes com o esperado. A **busca sequencial simples** (2) serve nesse caso como um padrão relativo ótimo. É um algoritmo simples, mas que teoricamente tem um custo muito pesado.

Como podemos observar no gráfico 2 a **busca sequencial mover para frente**3) teve o maior gasto de tempo, isso ocorreu pelo fato de o algoritmo precisar trocar muitas posições em um vetor sequencial na memória. Considerando que a quantidade de operações do algoritmo é muitas vezes maior que a **busca sequencial simples** seu tempo pode ser justificado e apresenta potencial com o uso de uma **lista duplamente encadeada**.

Observa-se que apesar de apresentar um custo levemente menor a **busca por transposição** (4) está no erro da **busca linear simples**, o que pode indicar que seu menor gasto está muito mais ligado ao nosso conjunto de dados do que a uma maior eficiência. Isso ocorre, pois, o algoritmo por transposição é beneficiado por muitas buscas consecutivas de um mesmo valor.

Finalmente, é possível constatar que o mais rápido é a **busca indexada** (5). Isso acontece, pois, a busca por indexação separa a tabela de valores original em um determinado número de tabelas menores (no nosso caso 5 tabelas menores) nas quais os intervalos de números em cada uma dessas tabelas é conhecido. É usada uma tabela de índices, para a separação dessas tabelas menores, como é mostrado na imagem abaixo.

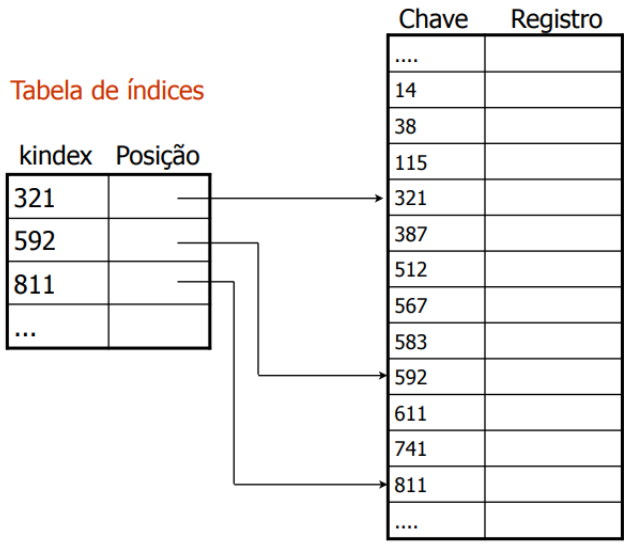

Figura 1: *Tabela de índices da busca indexada*
[3]

Então dado um número, ele vê em qual das tabelas menores o número se encontra e faz uma busca linear simples dentro dessa tabela bem menor que a original. Resultando em um tempo menor. Duas desvantagens desse método são: o gasto de memória para armazenar a tabela de índices e a tabela a ser buscada precisa estar ordenada.

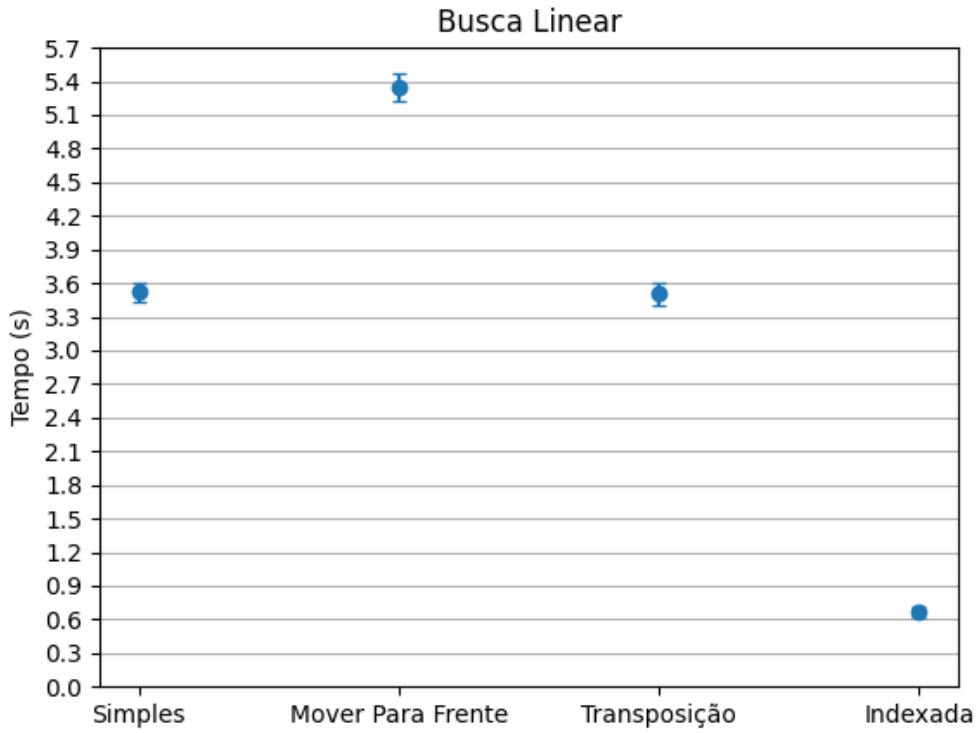

Figura 2: *Comparação de tempo médio e desvio padrão*

	Simples	Mover Para Frente	Transposição	Indexada
Média	3.5195	5.3447	3.5048	0.6692
Desvio	0.0852	0.1216	0.0966	0.0418

Tabela 1: *Comparação de tempo médio e desvio padrão*

Encontrados	Tempo
35557	3.777787
35557	3.662466
35557	3.611745
35557	3.711924
35557	3.502080
35557	3.497374
35557	3.503801
35557	3.471210
35557	3.472386
35557	3.480661
35557	3.465454
35557	3.470865
35557	3.457647
35557	3.866260
35557	3.467061
35557	3.458100
35557	3.479049
35557	3.467102
35557	3.472787
35557	3.488619
35557	3.460490
35557	3.465701
35557	3.856454
35557	3.469142
35557	3.487008
35557	3.513010
35557	3.719004
35557	3.595281
35557	3.481679
35557	3.476027
35557	3.500062
35557	3.501711
35557	3.465525
35557	3.477339
35557	3.508493
35557	3.530382
35557	3.536702
35557	3.540146
35557	3.539805
35557	3.531020
35557	3.548619
35557	3.500090
35557	3.477832
35557	3.494108
35557	3.482664
35557	3.461206
35557	3.477862
35557	3.480998
35557	3.474550
35557	3.487727
35557	3.491427
35557	3.481073

Tabela 2: *Dados da busca linear simples*

Encontrados	Tempo
35557	5.257695
35557	5.374248
35557	5.246118
35557	5.234704
35557	5.244544
35557	5.403954
35557	5.539519
35557	5.338925
35557	5.357337
35557	5.304174
35557	5.373766
35557	5.578322
35557	5.320039
35557	5.359173
35557	5.329015
35557	5.657998
35557	5.310312
35557	5.338450
35557	5.305002
35557	5.328427
35557	5.674809
35557	5.337931
35557	5.310951
35557	5.332533
35557	5.351419
35557	5.660081
35557	5.307879
35557	5.300842
35557	5.400211
35557	5.315179
35557	5.651347
35557	5.256994
35557	5.234109
35557	5.249720
35557	5.229208
35557	5.242858
35557	5.482065
35557	5.338303
35557	5.238317
35557	5.261085
35557	5.245985
35557	5.247097
35557	5.228891
35557	5.669120
35557	5.264483
35557	5.253225
35557	5.247950
35557	5.265660
35557	5.234549
35557	5.369308
35557	5.473894
35557	5.263339

Tabela 3: *Dados da busca linear para frente*

Encontrados	Tempo
35557	3.482829
35557	3.567513
35557	3.463285
35557	3.484377
35557	3.480359
35557	3.457369
35557	3.485933
35557	3.606534
35557	3.674316
35557	3.467438
35557	3.491671
35557	3.476965
35557	3.462367
35557	3.487690
35557	3.485157
35557	3.460875
35557	3.486660
35557	3.475944
35557	3.468824
35557	3.488858
35557	3.479341
35557	3.454125
35557	3.481561
35557	3.492776
35557	3.471747
35557	3.467476
35557	3.480892
35557	3.577169
35557	3.783674
35557	3.483873
35557	3.446128
35557	3.453178
35557	3.466977
35557	3.806360
35557	3.452167
35557	3.440453
35557	3.448701
35557	3.452378
35557	3.443598
35557	3.447420
35557	3.454868
35557	3.454560
35557	3.455639
35557	3.443153
35557	3.802789
35557	3.811049
35557	3.793709
35557	3.475753
35557	3.445544
35557	3.460132
35557	3.452652
35557	3.459094

Tabela 4: *Dados da busca linear transposição*

Encontrados	Tempo
35557	0.656277
35557	0.671563
35557	0.693595
35557	0.697612
35557	0.658447
35557	0.666182
35557	0.660493
35557	0.657719
35557	0.658598
35557	0.658467
35557	0.680814
35557	0.853067
35557	0.879794
35557	0.657555
35557	0.661383
35557	0.660982
35557	0.658329
35557	0.658834
35557	0.656620
35557	0.661928
35557	0.657516
35557	0.659384
35557	0.660135
35557	0.661342
35557	0.658793
35557	0.664368
35557	0.661467
35557	0.665845
35557	0.665087
35557	0.651678
35557	0.651764
35557	0.648788
35557	0.649664
35557	0.662253
35557	0.647851
35557	0.653264
35557	0.654265
35557	0.650559
35557	0.650477
35557	0.650189
35557	0.645659
35557	0.649078
35557	0.653412
35557	0.649465
35557	0.655602
35557	0.649230
35557	0.647533
35557	0.649734
35557	0.674501
35557	0.847319
35557	0.771219
35557	0.659649

Tabela 5: *Dados da busca linear indexada*

3.2. Espalhamento

3.2.1 Métodos de Análise

Como se pode observar pelos dados brutos coletados (Tabela 9, Tabela 10, Tabela 11, Tabela 12, Tabela 13, Tabela 14, Tabela 15 e Tabela 16) todas as execuções de todos os algoritmos encontraram 38 344 elementos. Tal valor está condizente com o que seria esperado da base de dados original e bate com o número de elementos encontrados por outros grupos.

Para cada variação de inserção e busca por espalhamento foram executados **52** testes e excluídos o maior e o menor tempo em uma tentativa de remover pontos de dados muito discrepantes. Com isso trabalhamos com **50** pontos de dados para cada algoritmo de busca por espalhamento.

3.2.2 Analisando os dados

Como podemos observar no gráfico 3 a **busca por espalhamento com multiplicação** teve o maior gasto de tempo para executar a busca, tanto no aberto quanto no fechado. Isso ocorreu pelo fato de que essa função *hash* foi a que teve mais colisões, o que pode ser verificado nas tabelas 10 e 14.

De encontro com isso, observa-se no mesmo gráfico que a **busca por espalhamento com número primo** teve o menor gasto de tempo para executar a busca, tanto no aberto como no fechado. Isso é justificado pelo fato de esse jeito de gerar a chave *hash* gerou um número muito menor de colisões, como pode ser constatado nas tabelas 11 e 15.

Além disso, todos os *hashes abertos* foram mais rápidos que os seus respectivos fechados. No **Overflow progressivo** e no **Hash duplo**, muitas vezes uma palavra não está no índice para o qual a chave *hash* aponta, e juntando várias colisões a busca se afasta cada vez mais do custo constante rumo ao custo linear. O **Hash aberto** também é prejudicado com colisões, se afastando do custo fixo e indo para o custo linear, porém ele sofre menos.

Por exemplo: no **Overflow Progressivo**, se houverem muitas colisões no índice 1, tudo o que era para colocar nos índices próximos abaixo também sofrerão colisões, já que as colisões no índice 1 foram ocupando as casas de baixo e assim em diante, no **Hash Duplo** não serão exatamente as casas logo abaixo do índice 1, mas outros índices mais espalhados também estarão populados com palavras cuja chave não é exatamente daquele índice. Já no **Hashing Aberto**, várias colisões em um índice tem um grande efeito apenas sobre as buscas e inserções que recaírem sobre aquele índice, já que ele insere o elemento numa lista lincada naquele índice e não fica ocupando outros índices da tabela. Porém, a consequência disso é que o **Hashing Aberto** desperdiça mais memória.

É possível perceber no gráfico 4 e 3 que as categorias de *hash* que foram mais rápidos na busca também se mantiveram na mesma posição quando falamos da inserção. Porém, de modo geral a inserção foi mais rápida do que a busca, isso se deve ao fato de que estamos inserindo 50 000 palavras e buscando 70 000. Além do fato de que para inserir não precisamos comparar as palavras, basta ir até o índice indicado pela chave, achar uma casa nula e inserir, enquanto na busca temos que o tempo todo ficar comparando se a palavra na casa é igual ao que estamos buscando, e isso dezenas de milhares a milhões de vezes representa um aumento significativo de tempo para a busca.

Curioso notar como o **hash por divisão e multiplicação** estão distantes na busca, mas próximos na inserção, isso se deve ao fato de que o número de colisões impacta mais a busca do que a inserção, de acordo com o que foi explicado no parágrafo acima. Como a **multiplicação** possui mais colisões que a divisão, 49 495 contra 41 351 (10 e 9), essas colisões geram um retardamento considerável na busca.

3.2.3 Observações Importantes

É possível ver que na inserção, o **Hash Duplo**, mesmo com muito mais colisões foi bem mais rápido que o por **divisão**. Isso provavelmente se deve ao jeito com que contamos as colisões. Fomos orientados pelo professor a contar apenas a primeira colisão de cada chave, não contando as colisões que viriam nas próximas iterações. Acontece que essas colisões podem ser muito pertinentes na análise. Por isso foi feita a contagem de colisões também contando as iterações na tabela 6.

Overflow com Divisão	Overflow com Multiplicação	Overflow com Primo	Hash Fechado Duplo
3 172 570	4 911 294	33 492	1 547 514

Tabela 6: *Contagem de colisões considerando as iterações*

Observando as colisões na tabela acima, conseguimos perceber que, na verdade o **Hash Fechado Duplo** gera menos colisões que o por divisões e multiplicações e por isso ele é mais rápido na inserção. Desconfia-se que ele é mais devagar na busca pois, como ele faz a função *hash*, que é uma função dupla e mais custosa, várias vezes, ele acaba levando mais tempo que o de divisão, não superando o de multiplicação pois nesse as colisões são realmente muito altas. Portanto, uma análise ideal deve considerar os 2 jeitos de se contar as colisões.

Fica claro, ainda, olhando na tabela 6, o quanto o **Hash com Primo** é muito melhor que os outros. Essa é uma função extra, fora das que o professor pediu que fossem implementadas, e foi analisada pois demonstrou uma performance muito acima das demais.

Outro importante detalhe a ser notado: foi usado uma espécie de **Hash Duplo** com a tabela aberta também, o que não foi pedido pelo professor. Isso foi feito apenas para facilitar as comparações de tempo entre os diferentes algorítimos de espalhamento. Essa função é igual à passada pelo professor para ser usada no **Hash Fechado** porém sem considerar as iterações, pois na primeira vez o elemento já sempre será colocado em uma lista encadeada.

Por fim, apesar da diferente contagem apresentada na tabela 6, todas as tabelas com os dados brutos (Tabela 9, Tabela 10, Tabela 11, Tabela 12, Tabela 13, Tabela 14, Tabela 15 e Tabela 16) estão corretas. A única diferença é que as contagens de colisões foram contadas de jeitos diferentes (o que não significa que um esteja certo e o outo errado), essa contagem diferente de colisões não altera o tempo de execução da busca e inserção e muito menos o número de elementos encontrados. Portanto a análise segue válida.

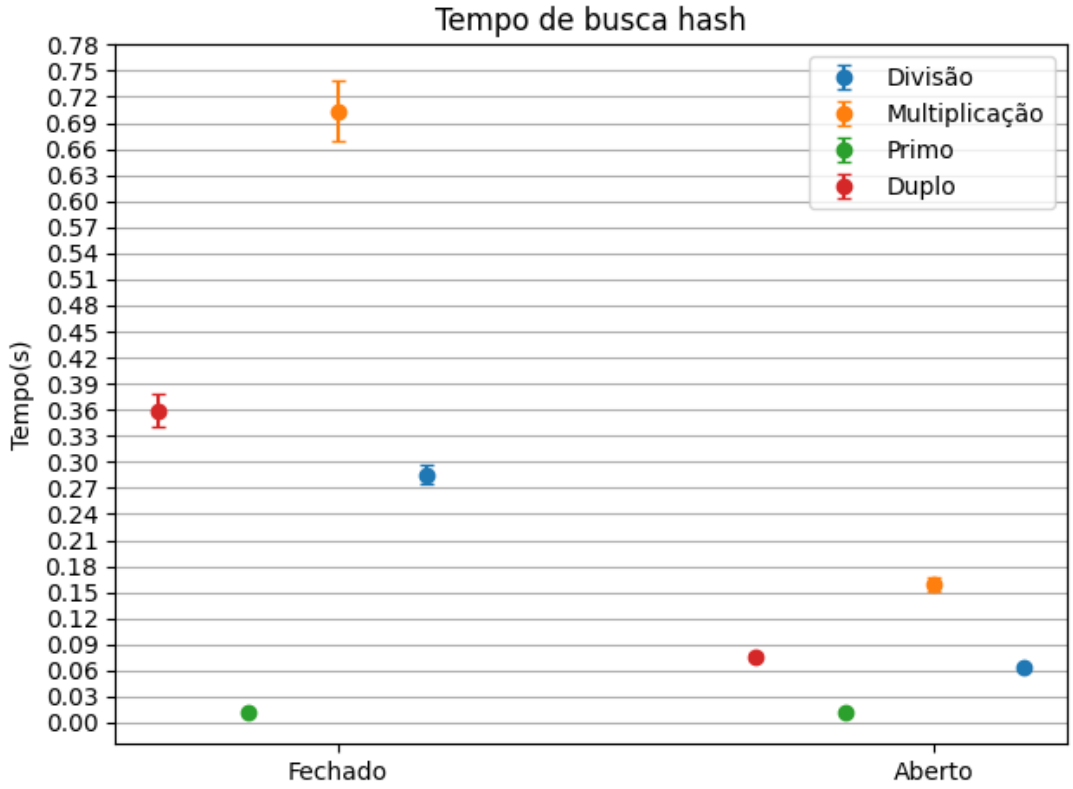

Figura 3: *Comparação de tempo médio e desvio padrão de busca na tabela hash*

Hash	Média	Desvio Padrão
Fechado Divisão	0.2853	0.0113
Fechado Multiplicação	0.7036	0.0355
Fechado Primo	0.0127	0.0005
Fechado Duplo	0.3592	0.0187
Aberto Divisão	0.0629	0.0015
Aberto Multiplicação	0.1593	0.007
Aberto Primo	0.0124	0.0005
Aberto Duplo	0.0765	0.0036

Tabela 7: *Comparação de tempo médio e desvio padrão da busca hash*

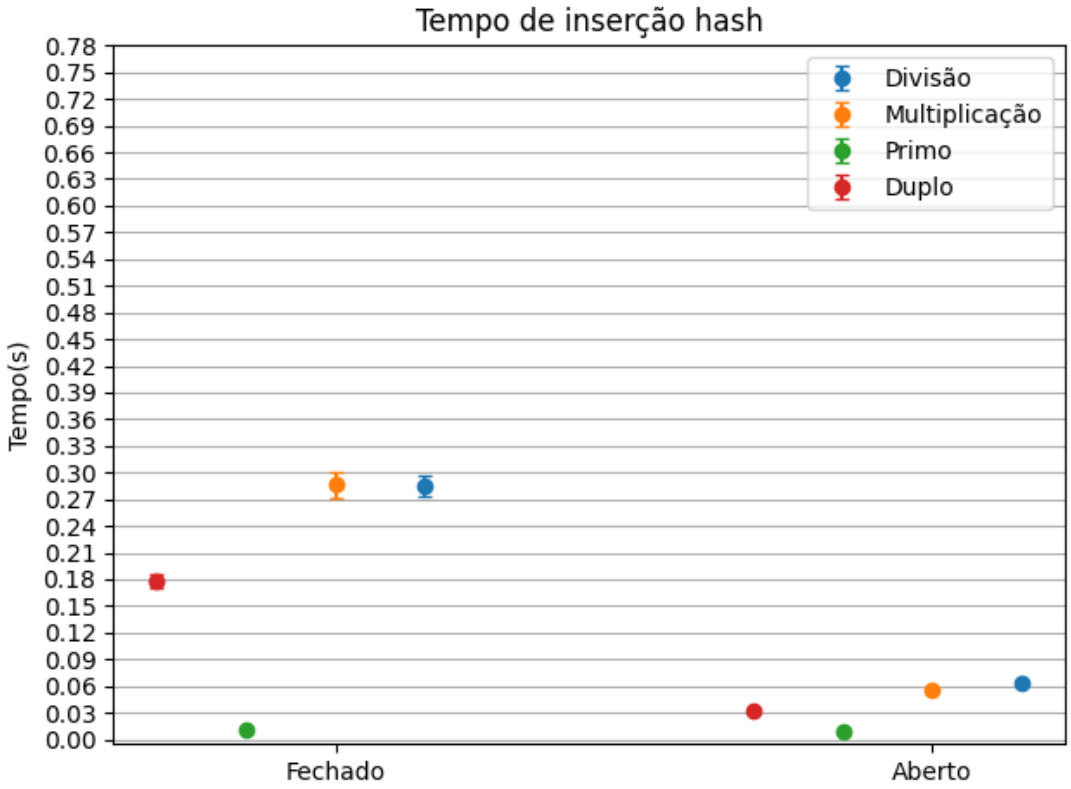

Figura 4: *Comparação de tempo médio e desvio padrão de inserção na tabela hash*

Hash	Média	Desvio Padrão
Fechado Divisão	0.2853	0.0113
Fechado Multiplicação	0.286	0.0142
Fechado Primo	0.0107	0.0005
Fechado Duplo	0.178	0.0069
Aberto Divisão	0.0629	0.0015
Aberto Multiplicação	0.0546	0.0017
Aberto Primo	0.0096	0.0004
Aberto Duplo	0.0312	0.0012

Tabela 8: *Comparação de tempo médio e desvio padrão da insercao hash*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	41351	0.088376	0.282795
38344	41351	0.090138	0.294628
38344	41351	0.094762	0.305126
38344	41351	0.093445	0.289371
38344	41351	0.094097	0.284146
38344	41351	0.094656	0.287735
38344	41351	0.094169	0.288200
38344	41351	0.096309	0.279663
38344	41351	0.093381	0.282188
38344	41351	0.093415	0.282207
38344	41351	0.093581	0.288328
38344	41351	0.093144	0.282860
38344	41351	0.113026	0.369761
38344	41351	0.094214	0.282438
38344	41351	0.093618	0.286438
38344	41351	0.092109	0.285079
38344	41351	0.094829	0.277993
38344	41351	0.093404	0.279131
38344	41351	0.094711	0.284741
38344	41351	0.095450	0.283399
38344	41351	0.093161	0.286158
38344	41351	0.097990	0.287084
38344	41351	0.093405	0.289138
38344	41351	0.096734	0.295173
38344	41351	0.093707	0.344235
38344	41351	0.096940	0.282707
38344	41351	0.095086	0.280868
38344	41351	0.093786	0.287302
38344	41351	0.092841	0.294397
38344	41351	0.093485	0.287393
38344	41351	0.092685	0.284527
38344	41351	0.093253	0.280578
38344	41351	0.093420	0.280844
38344	41351	0.093142	0.281111
38344	41351	0.092888	0.283367
38344	41351	0.093171	0.283533
38344	41351	0.094608	0.295665
38344	41351	0.112252	0.301836
38344	41351	0.095035	0.279647
38344	41351	0.091287	0.269668
38344	41351	0.093604	0.269546
38344	41351	0.093118	0.292274
38344	41351	0.090819	0.274039
38344	41351	0.094244	0.275798
38344	41351	0.094703	0.272742
38344	41351	0.092360	0.273900
38344	41351	0.091990	0.277148
38344	41351	0.091231	0.275695
38344	41351	0.092420	0.274155
38344	41351	0.092213	0.276010
38344	41351	0.111745	0.361404
38344	41351	0.091403	0.268783

Tabela 9: *Dados da busca hash fechado por divisão*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	49495	0.281526	0.712372
38344	49495	0.293864	0.723098
38344	49495	0.300348	0.802229
38344	49495	0.286067	0.713754
38344	49495	0.281982	0.692501
38344	49495	0.285362	0.698131
38344	49495	0.286191	0.707794
38344	49495	0.287337	0.713414
38344	49495	0.281508	0.692355
38344	49495	0.281008	0.694800
38344	49495	0.290799	0.696417
38344	49495	0.282352	0.709143
38344	49495	0.349805	0.804505
38344	49495	0.286407	0.702398
38344	49495	0.284864	0.705752
38344	49495	0.286386	0.701094
38344	49495	0.280745	0.695337
38344	49495	0.285738	0.693821
38344	49495	0.283014	0.689745
38344	49495	0.282723	0.691127
38344	49495	0.282116	0.716980
38344	49495	0.287774	0.692000
38344	49495	0.291036	0.703346
38344	49495	0.292308	0.699945
38344	49495	0.348196	0.896342
38344	49495	0.291808	0.730534
38344	49495	0.283631	0.696935
38344	49495	0.283091	0.695344
38344	49495	0.284892	0.720234
38344	49495	0.279565	0.694705
38344	49495	0.284701	0.699020
38344	49495	0.284972	0.688872
38344	49495	0.287747	0.689018
38344	49495	0.291550	0.711528
38344	49495	0.281171	0.690118
38344	49495	0.283105	0.693934
38344	49495	0.286261	0.832332
38344	49495	0.290322	0.701902
38344	49495	0.275902	0.693306
38344	49495	0.274936	0.665698
38344	49495	0.273519	0.669904
38344	49495	0.280766	0.673894
38344	49495	0.273001	0.674468
38344	49495	0.275406	0.665558
38344	49495	0.276289	0.673163
38344	49495	0.276457	0.672796
38344	49495	0.271768	0.678370
38344	49495	0.274790	0.678476
38344	49495	0.274254	0.663728
38344	49495	0.271673	0.671883
38344	49495	0.348896	0.802334
38344	49495	0.281127	0.665334

Tabela 10: *Dados da busca hash fechado por multiplicação*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	20630	0.008446	0.011891
38344	20630	0.010409	0.014013
38344	20630	0.011104	0.013413
38344	20630	0.010594	0.012875
38344	20630	0.010469	0.012096
38344	20630	0.010785	0.012495
38344	20630	0.010455	0.012396
38344	20630	0.010682	0.012454
38344	20630	0.010400	0.012403
38344	20630	0.010708	0.012359
38344	20630	0.010211	0.012708
38344	20630	0.010615	0.012935
38344	20630	0.010456	0.012816
38344	20630	0.010444	0.012782
38344	20630	0.010256	0.012992
38344	20630	0.010339	0.012463
38344	20630	0.011101	0.012352
38344	20630	0.011872	0.012829
38344	20630	0.010732	0.012605
38344	20630	0.010475	0.012560
38344	20630	0.010480	0.012660
38344	20630	0.010154	0.012717
38344	20630	0.010767	0.012125
38344	20630	0.010627	0.012852
38344	20630	0.015096	0.016894
38344	20630	0.010852	0.012735
38344	20630	0.010609	0.012435
38344	20630	0.010723	0.012515
38344	20630	0.010780	0.012859
38344	20630	0.010137	0.012295
38344	20630	0.011545	0.013871
38344	20630	0.010206	0.012497
38344	20630	0.010072	0.012590
38344	20630	0.011536	0.012803
38344	20630	0.010302	0.012457
38344	20630	0.010264	0.012745
38344	20630	0.011954	0.015058
38344	20630	0.010331	0.012720
38344	20630	0.011249	0.012383
38344	20630	0.010371	0.012385
38344	20630	0.010747	0.012300
38344	20630	0.011954	0.012551
38344	20630	0.010476	0.012449
38344	20630	0.012068	0.012941
38344	20630	0.010227	0.012575
38344	20630	0.010351	0.012698
38344	20630	0.010193	0.012656
38344	20630	0.011382	0.012357
38344	20630	0.010602	0.012480
38344	20630	0.010258	0.012736
38344	20630	0.011367	0.012588
38344	20630	0.011823	0.012799

Tabela 11: *Dados da busca hash fechado por primos*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	49497	0.181534	0.392124
38344	49497	0.177482	0.393599
38344	49497	0.176129	0.363361
38344	49497	0.176691	0.359748
38344	49497	0.192452	0.375318
38344	49497	0.176910	0.374725
38344	49497	0.177805	0.356904
38344	49497	0.178474	0.354219
38344	49497	0.176994	0.356716
38344	49497	0.173501	0.350493
38344	49497	0.175473	0.350694
38344	49497	0.180953	0.442280
38344	49497	0.181496	0.361263
38344	49497	0.179671	0.367225
38344	49497	0.178234	0.358118
38344	49497	0.177249	0.354531
38344	49497	0.176307	0.360303
38344	49497	0.187423	0.380923
38344	49497	0.177753	0.349033
38344	49497	0.177099	0.351831
38344	49497	0.180905	0.358389
38344	49497	0.175175	0.354520
38344	49497	0.176542	0.354619
38344	49497	0.177137	0.351405
38344	49497	0.224029	0.347562
38344	49497	0.180382	0.352158
38344	49497	0.171635	0.354671
38344	49497	0.174829	0.353977
38344	49497	0.174280	0.354013
38344	49497	0.174816	0.352425
38344	49497	0.181404	0.363867
38344	49497	0.175391	0.358868
38344	49497	0.175138	0.353417
38344	49497	0.177077	0.351147
38344	49497	0.192814	0.360248
38344	49497	0.175091	0.369596
38344	49497	0.213895	0.456671
38344	49497	0.176479	0.352052
38344	49497	0.172135	0.338495
38344	49497	0.170176	0.351112
38344	49497	0.172081	0.349830
38344	49497	0.172114	0.340217
38344	49497	0.173438	0.340108
38344	49497	0.179245	0.341303
38344	49497	0.172390	0.336654
38344	49497	0.172043	0.339023
38344	49497	0.172546	0.345759
38344	49497	0.183830	0.371150
38344	49497	0.172718	0.339104
38344	49497	0.171517	0.410690
38344	49497	0.179287	0.359449
38344	49497	0.174480	0.339077

Tabela 12: *Dados da busca hash fechado duplo*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	39582	0.040944	0.063106
38344	39582	0.016800	0.059399
38344	39582	0.019180	0.077525
38344	39582	0.018855	0.062260
38344	39582	0.019237	0.060280
38344	39582	0.018327	0.060361
38344	39582	0.018393	0.062665
38344	39582	0.018778	0.062237
38344	39582	0.019144	0.077685
38344	39582	0.019050	0.066899
38344	39582	0.018712	0.061240
38344	39582	0.018703	0.063839
38344	39582	0.018600	0.064717
38344	39582	0.019066	0.063523
38344	39582	0.019086	0.061607
38344	39582	0.018408	0.064563
38344	39582	0.018734	0.062827
38344	39582	0.018777	0.062148
38344	39582	0.018592	0.061294
38344	39582	0.017133	0.064698
38344	39582	0.018465	0.063124
38344	39582	0.016420	0.066216
38344	39582	0.018470	0.061765
38344	39582	0.018201	0.064944
38344	39582	0.018471	0.062463
38344	39582	0.018478	0.060851
38344	39582	0.018544	0.061576
38344	39582	0.018157	0.064485
38344	39582	0.018636	0.060425
38344	39582	0.018396	0.061860
38344	39582	0.018113	0.061165
38344	39582	0.018076	0.062769
38344	39582	0.018691	0.062011
38344	39582	0.019821	0.063257
38344	39582	0.018636	0.061618
38344	39582	0.018449	0.063956
38344	39582	0.017703	0.063000
38344	39582	0.019061	0.062116
38344	39582	0.018407	0.061134
38344	39582	0.018914	0.064631
38344	39582	0.018519	0.063028
38344	39582	0.018808	0.065086
38344	39582	0.017897	0.063136
38344	39582	0.019075	0.064627
38344	39582	0.018728	0.061720
38344	39582	0.018674	0.060956
38344	39582	0.018839	0.062716
38344	39582	0.018678	0.063547
38344	39582	0.018602	0.061980
38344	39582	0.018665	0.062588
38344	39582	0.018646	0.061271
38344	39582	0.017346	0.065206

Tabela 13: *Dados da busca hash aberta por divisão*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	49495	0.046836	0.169551
38344	49495	0.058919	0.194333
38344	49495	0.060826	0.205099
38344	49495	0.052728	0.156612
38344	49495	0.067270	0.163161
38344	49495	0.054526	0.160116
38344	49495	0.055016	0.154726
38344	49495	0.053221	0.176453
38344	49495	0.053357	0.167421
38344	49495	0.054692	0.159455
38344	49495	0.054657	0.162924
38344	49495	0.055204	0.155193
38344	49495	0.054475	0.160107
38344	49495	0.053717	0.159768
38344	49495	0.053831	0.157870
38344	49495	0.053567	0.161027
38344	49495	0.054050	0.165056
38344	49495	0.054866	0.153764
38344	49495	0.054692	0.155315
38344	49495	0.054896	0.154556
38344	49495	0.054529	0.158562
38344	49495	0.053390	0.154677
38344	49495	0.053853	0.154869
38344	49495	0.056042	0.152657
38344	49495	0.053105	0.162682
38344	49495	0.055645	0.154079
38344	49495	0.054806	0.153948
38344	49495	0.054577	0.161753
38344	49495	0.053798	0.158108
38344	49495	0.054007	0.152720
38344	49495	0.053938	0.154860
38344	49495	0.054395	0.153393
38344	49495	0.053683	0.159497
38344	49495	0.056083	0.157708
38344	49495	0.054364	0.155021
38344	49495	0.054250	0.152820
38344	49495	0.053673	0.157959
38344	49495	0.054912	0.156298
38344	49495	0.054952	0.152801
38344	49495	0.054465	0.153472
38344	49495	0.053704	0.160766
38344	49495	0.053312	0.158382
38344	49495	0.054507	0.157997
38344	49495	0.055398	0.156795
38344	49495	0.053658	0.164742
38344	49495	0.054445	0.154480
38344	49495	0.053041	0.157930
38344	49495	0.054080	0.157227
38344	49495	0.053214	0.162999
38344	49495	0.061265	0.168017
38344	49495	0.054648	0.155794
38344	49495	0.055501	0.154840

Tabela 14: *Dados da busca hash aberta por multiplicação*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	19355	0.009688	0.014572
38344	19355	0.010528	0.016742
38344	19355	0.010149	0.013308
38344	19355	0.010438	0.012097
38344	19355	0.009517	0.012467
38344	19355	0.009295	0.011801
38344	19355	0.009278	0.012225
38344	19355	0.010281	0.012991
38344	19355	0.009379	0.012013
38344	19355	0.009917	0.012377
38344	19355	0.009146	0.012124
38344	19355	0.009280	0.012134
38344	19355	0.009252	0.012012
38344	19355	0.009392	0.012082
38344	19355	0.009342	0.012028
38344	19355	0.010048	0.012358
38344	19355	0.009486	0.012217
38344	19355	0.009251	0.012230
38344	19355	0.009440	0.012220
38344	19355	0.009601	0.012331
38344	19355	0.009568	0.012066
38344	19355	0.009531	0.011968
38344	19355	0.009226	0.012074
38344	19355	0.009605	0.012179
38344	19355	0.010179	0.012445
38344	19355	0.009829	0.012645
38344	19355	0.010259	0.012327
38344	19355	0.009482	0.012060
38344	19355	0.010064	0.011938
38344	19355	0.009995	0.012199
38344	19355	0.009352	0.012091
38344	19355	0.009164	0.012266
38344	19355	0.008973	0.012269
38344	19355	0.010512	0.011976
38344	19355	0.009251	0.012224
38344	19355	0.009347	0.012644
38344	19355	0.009451	0.012464
38344	19355	0.009295	0.012844
38344	19355	0.009491	0.012117
38344	19355	0.009395	0.012437
38344	19355	0.009376	0.013831
38344	19355	0.009543	0.012199
38344	19355	0.009732	0.012248
38344	19355	0.009538	0.012320
38344	19355	0.009472	0.012333
38344	19355	0.009231	0.012490
38344	19355	0.009325	0.012187
38344	19355	0.009895	0.012520
38344	19355	0.009566	0.012132
38344	19355	0.009972	0.013428
38344	19355	0.009230	0.012588
38344	19355	0.009641	0.012215

Tabela 15: *Dados da busca hash aberta por primos*

Encontrados	Colisões	Tempo Inserção	Tempo Busca
38344	39529	0.028473	0.077486
38344	39529	0.032309	0.101987
38344	39529	0.033496	0.072312
38344	39529	0.031778	0.081192
38344	39529	0.032875	0.074198
38344	39529	0.030533	0.076214
38344	39529	0.030549	0.074227
38344	39529	0.031674	0.086561
38344	39529	0.032626	0.075659
38344	39529	0.029824	0.077430
38344	39529	0.030903	0.076648
38344	39529	0.031209	0.072901
38344	39529	0.031147	0.076940
38344	39529	0.031986	0.074167
38344	39529	0.031156	0.074238
38344	39529	0.031709	0.076052
38344	39529	0.031796	0.081007
38344	39529	0.031627	0.075452
38344	39529	0.031656	0.075916
38344	39529	0.028312	0.077136
38344	39529	0.027404	0.077108
38344	39529	0.031467	0.074559
38344	39529	0.031424	0.075249
38344	39529	0.030134	0.077083
38344	39529	0.034477	0.073685
38344	39529	0.031539	0.076747
38344	39529	0.032120	0.094231
38344	39529	0.031402	0.073587
38344	39529	0.031709	0.074038
38344	39529	0.028220	0.078250
38344	39529	0.032052	0.073536
38344	39529	0.030043	0.074938
38344	39529	0.029594	0.076446
38344	39529	0.032281	0.076119
38344	39529	0.029774	0.079205
38344	39529	0.030815	0.076110
38344	39529	0.031345	0.077385
38344	39529	0.030951	0.076155
38344	39529	0.033397	0.075448
38344	39529	0.032153	0.072664
38344	39529	0.032062	0.074445
38344	39529	0.030332	0.078000
38344	39529	0.031832	0.076673
38344	39529	0.032407	0.073570
38344	39529	0.031335	0.073482
38344	39529	0.031556	0.076084
38344	39529	0.030823	0.074992
38344	39529	0.031433	0.076073
38344	39529	0.032850	0.074673
38344	39529	0.031720	0.080951
38344	39529	0.030351	0.074355
38344	39529	0.029158	0.076887

Tabela 16: *Dados da busca hash aberta duplo*

4. CONCLUSÃO

Nesse trabalho, vários algoritmos foram apresentados, todos com o intuito de realizar buscas e cada um com suas vantagens e desvantagens. Para a avaliação de cada um deles, foi realizado a implementação, teste, coleta de dados, explicação do funcionamento e análise gráfica e em tabela dos tais.

O foco da análise foram diferentes categorias de busca sequencial e diferentes de categorias de busca por espalhamento(*hash*).

Foi possível analisar que os métodos de busca sequencial são mais fáceis de se implementar e não consomem muita memória, porém a busca é feita em tempo linear em função do número de itens na lista de dados, o que pode com facilidade ser inviável devido à demora quando se lida com muitos itens. Além disso, dentro das categorias de busca sequencial existem, entre elas opções que podem ser melhores que outras dependendo do caso.

A busca por espalhamento, por outro lado, já requer um cuidado maior para ser implementada e consome muita memória, porém a busca é feita em tempo quase constante, quando a função *hash* é bem implementada, o que pode ser muito útil em diversas aplicações, especialmente quando se dispõe de uma vasta memória. Foram utilizadas funções *hash* dadas pelo professor e uma extra fora das que o professor disponibilizou, que apresentou melhor desempenho. Assim como na busca sequencial, existem dentro da busca por *hash* alguns *hashes* que serão preferíveis a outros em casos específicos.

Como possível proposta para desenvolvimento desse trabalho, é pensado em estudar também outros métodos de busca, como a busca binária.

O projeto foi finalizado após amplo desenvolvimento técnico e formal via ferramentas como a linguagem de programação c, git, make, python e LaTeX. As maiores dificuldades enfrentadas foram validar as saídas do código, visto estar se lidando com muitos dados e uma validação a mão era completamente inviável, fazer um código eficiente, reaplicável e, ao mesmo tempo, organizado e legível e conseguir, a partir das saídas gerar informações que poderiam ser analisadas de modo eficiente, como tabelas e gráficos.

Finalmente os resultados do projeto foram satisfatórios, pois promoveram a prática e aperfeiçoamento daquilo que já era conhecido e o conhecimento de novas práticas.

Referências

[1] Robson L. F. Cordeiro. Especificações segundo projeto, 2022. URL https://ae4.tidia-ae.usp.br/access/content/attachment/efe16172-fba1-4b85-bcee-c14438af04cf/Atividades/21d03f84-3fa9-4cf2-a66e-d1d71597cd80/projeto_2.pdf.

[2] Linear search. *Geeks for Geeks*, 2022. URL https://www.geeksforgeeks.org/linear-search/.

[3] Robson L. F. Cordeiro, Thiago A. S. Pardo, and Rudinei Goularte. Métodos de busca: parte i, 2021. URL https://ae4.tidia-ae.usp.br/access/content/group/efe16172-fba1-4b85-bcee-c14438af04cf/aulas/3_Busca_p1.pdf.

[4] Indexed searching algorithms. *Geeks for Geeks*, 2022. URL https://www.geeksforgeeks.org/indexed-sequential-search/.

[5] Robson L. F. Cordeiro, Thiago A. S. Pardo, and Rudinei Goularte. Métodos de busca: parte ii, 2021. URL https://ae4.tidia-ae.usp.br/access/content/group/efe16172-fba1-4b85-bcee-c14438af04cf/aulas/3_Busca_p2.pdf.

[6] Searching algorithms. *Geeks for Geeks*, 2022. URL https://www.geeksforgeeks.org/searching-algorithms/.