Análise de Algoritmos de Busca

Tales Lima de Oliveira[1]

[1]Instituído Federal de Brasília (IFB)
Taguatinga – DF – Brasil

tales.oliveira@estudante.ifb.edu.br

Abstract. *This report aims to analyze the efficiency of three search methods: simple sequential search, optimized sequential search, and binary search, in different scenarios. The analysis intends to compare the time required to find a desired element in lists of varying sizes and distributions, identifying the most appropriate method for each situation. Additionally, the analysis includes a comparison between the experimentally obtained times and the asymptotic analyses of the algorithms, aiming to evaluate the accuracy of theoretical predictions in relation to practical performance.*

Resumo. *Este relatório tem como objetivo analisar a eficiência de três métodos de busca: busca sequencial simples, busca sequencial otimizada e busca binária, em diferentes situações. A análise pretende comparar o tempo necessário para encontrar um elemento desejado em listas de variados tamanhos e distribuições, identificando o método mais apropriado para cada cenário. Além disso, a análise inclui uma comparação entre o tempo obtido experimentalmente e as análises assintóticas dos algoritmos, visando avaliar a precisão das previsões teóricas em relação ao desempenho prático.*

1. Introdução

Os algoritmos de busca são ferramentas essenciais na ciência da computação usadas para localizar itens específicos dentro de uma coleção de dados. Esses algoritmos são projetados para navegar de maneira eficiente através de estruturas de dados para encontrar a informação desejada, tornando-os fundamentais em diversas aplicações, como bancos de dados, motores de busca na web e muito mais [GeeksforGeeks 2024c].

O desempenho dos algoritmos de busca não só afeta a rapidez com que os dados podem ser acessados, mas também impacta a escalabilidade e a capacidade geral dos sistemas. A escolha do algoritmo de busca adequado pode melhorar significativamente o tempo de resposta e a utilização dos recursos, especialmente em sistemas que operam com grandes conjuntos de informações e/ou em tempo real [Cormen et al. 2009].

Agências bancárias, que frequentemente lidam com grandes volumes de dados, são um exemplo de setor que se beneficia da utilização de ferramentas computacionais eficazes [Alves 2018].

Existem diversos algoritmos para realizar buscas, entre os quais analisare-mos, a busca sequencial e a busca binária. A busca sequencial examina cada item da lista, um por um, independentemente da ordem dos elementos. No entanto, este algoritmo pode ser otimizado quando a lista está ordenada, resultando na busca sequencial otimizada [GeeksforGeeks 2024b].

A busca sequencial otimizada também realiza a verificação item por item, mas inclui uma melhoria: quando o item analisado é maior do que o item procurado, a busca pode ser interrompida antecipadamente, concluindo que o item não está presente na lista. Isso reduz o número de comparações necessárias em listas ordenadas.

Por outro lado, a busca binária, que requer que a lista esteja ordenada, é um algoritmo mais eficiente para essa situação. Ela divide a lista em partes menores repetidamente, o que permite localizar o item com um menor número de comparações do que a busca sequencial [GeeksforGeeks 2024a].

Além disso, é necessário considerar o método de ordenação mais eficiente, de acordo com o tamanho das listas. Neste trabalho, utilizaremos os métodos de ordenação padrão da linguagem utilizada, que são o Insertion Sort, Merge Sorte o QuickSort, para comparar a eficiência dos algoritmos de busca.

Neste relatório, o objetivo principal é analisar e comparar três métodos de busca: a busca sequencial simples, a busca sequencial otimizada e a busca binária. Avaliaremos o desempenho de cada método em diferentes cenários, considerando tamanhos variados de listas e quantidades distintas de buscas. Além disso, investigaremos a relação entre a análise assintótica, que fornece uma estimativa teórica do tempo de execução dos algoritmos, e o tempo obtido experimentalmente durante os testes. Isso nos permitirá avaliar a precisão das previsões teóricas e entender melhor como os algoritmos se comportam na prática.

2. Embasamento Teórico

2.1. Busca Sequencial Simples

Dado um conjunto de números, letras, endereços ou qualquer outro tipo de variável em um programa, a busca sequencial simples consiste em verificar cada elemento, um a um, a partir do primeiro elemento, para encontrar o item procurado. Assim que o elemento é encontrado, seu índice é retornado e a busca é finalizada. Caso o elemento não exista no conjunto, o programa retorna um valor especial que não pode ser confundido com um índice, indicando que o item não foi encontrado [Ziviani 1999].

Um exemplo visual da busca sequencial simples pode ser visto na figura 1.

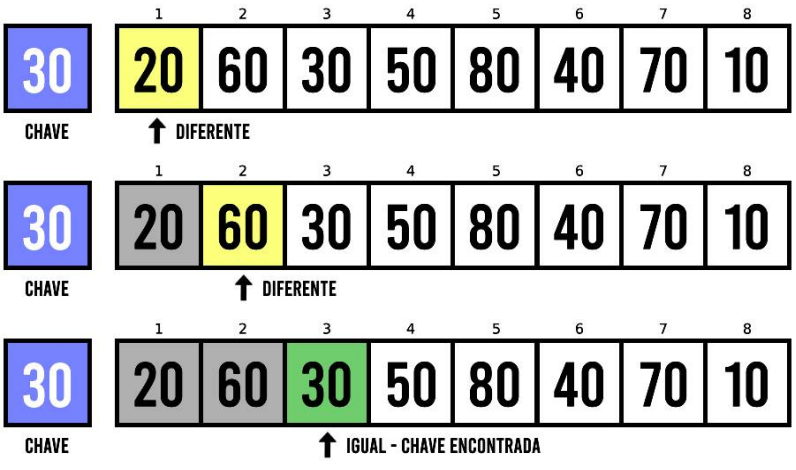


Figura 1. Busca sequencial simples.

Analisando o algoritmo de busca sequencial, temos que no melhor caso (quando o elemento está na primeira posição), a complexidade é $O(1)$. No caso médio (quando o elemento está na posição central), a complexidade é $O(n/2)$. No pior caso (quando o elemento está na última posição ou não está presente), a complexidade é $O(n)$, onde n é o tamanho da lista de dados [Ziviani 1999].

2.2. Métodos de Ordenação

Para que possamos implementar os algoritmos de busca sequencial otimizada e busca binária, precisamos ordenar a lista de dados. A linguagem Julia, que foi a liguaguem escolhida para implementação desse trabalho, emprega três métodos de ordenação padrão: Insertion Sort, Merge Sort e QuickSort, dependendo do tipo, tamanho e composição dos dados de entrada.

2.2.1. Insertion Sort

No algoritmo Insertion Sort, ou Método da Inserção, o arranjo é percorrido diversas vezes, fazendo comparações dois a dois entre os elementos. Dessa forma, os elementos adjacentes são deslocados para que o menor elemento encontrado até então seja inserido na posição mais próxima da correta no início do conjunto. Depois de algumas iterações, os elementos tomam seus lugares certos e o arranjo fica ordenado [Tenembaum et al. 1995].

2.2.2. Merge Sort

O método de ordenação Merge Sort utiliza a recursividade para aplicar a abordagem de dividir e conquistar. O conjunto de elementos a serem ordenados é dividido em subconjuntos e cada subconjunto é ordenado. Ao fim de cada chamada recursiva, os subconjuntos ordenados são combinados até que reste apenas um conjunto ordenado. O método utilizado para combinar os subconjuntos é a intercalação, na qual os dois subconjuntos resultantes têm seus elementos comparados para formar um subconjunto ordenado maior [Cormen et al. 2009].

2.2.3. QuickSort

QuickSort utiliza, como no Merge Sort, a estratégia de dividir e conquistar. No entanto, antes da divisão, é escolhido um pivô para que tudo antes dele seja menor e tudo depois dele seja maior. A cada chamada recursiva é escolhido um novo pivô e os elementos são ordenados. Ao contrário do Merge Sort, não é necessário combinar os subarranjos ordenados, pois estes já estão ordenados corretamente. Geralmente, o QuickSort é a melhor opção para ordenar elementos em um conjunto devido à sua velocidade [Cormen et al. 2009].

2.3. Busca Sequencial Otimizada

A busca sequencial otimizada é semelhante à busca sequencial simples, mas explora a ordenação da lista para agilizar a busca. Se o elemento buscado for menor que um

elemento da lista em uma determinada posição, podemos concluir que ele não está presente nos elementos restantes da lista. Isso reduz o número de comparações necessárias, especialmente em listas grandes.

Um exemplo visual da busca sequencial otmizada pode ser visto na figura 2.

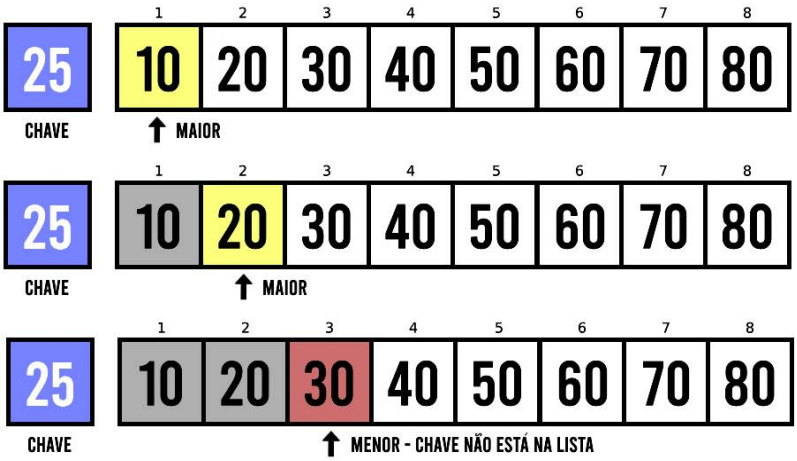


Figura 2. Busca sequencial otimizada.

Embora a busca sequencial otimizada reduza o número de comparações caso a chave seja menor que o valor da elemento da lista verificado, sua complexidade no pior caso permanece $O(n)$, assim como na busca sequencial simples (quando o elemento está na última posição ou não está presente). No melhor caso, a complexidade é $O(1)$, e no caso médio, é $O(n/2)$, em que n é o tamanho da lista de dados.

2.4. Busca Binária

O algoritmo de busca binária apresenta um método diferente da busca sequencial. Ele consiste em comparar inicialmente o elemento buscado com o elemento central do conjunto, de forma que uma das metades possa ser desprezada, minimizando o escopo de busca. Nessa comparação, verifica-se se o elemento buscado é maior ou menor que o elemento central, para que a busca parta para uma das metades. Para que isso aconteça, o conjunto de elementos a ser analisado deve estar em ordem crescente. A complexidade da busca binária é $O(\log n)$ [Ziviani 1999].

Um exemplo visual da busca binária pode ser visto na figura 3.

3. Métodos e Materiais

Para comparar os métodos de busca, foram realizados experimentos utilizando quatro listas com tamanhos n e q para o número de buscas a serem realizadas nessas listas. Os tamanhos das listas n foram definidos como 10^4, 10^5, 10^6 e 10^7, e o número de buscas q assumiu os valores 10^2, 10^3, 10^4 e 10^5.

A metodologia utilizada foi a seguinte:

- Os algoritmos foram implementados utilizando a linguagem Julia.

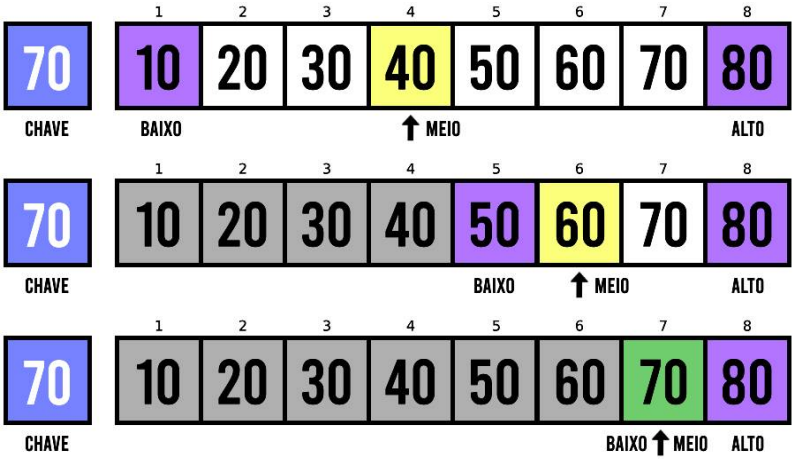


Figura 3. Busca binária.

- Cada algoritmo foi executado para diferentes combinações de n e q, com cada experimento sendo repetido 10 vezes para minimizar a influência de fatores aleatórios.
- As execuções foram realizadas na plataforma Google Colab e em uma máquina local **A**. *As configurações de hardware dos ambientes estão na Tabela 1.*
- Os resultados coletados foram analisados para comparar o desempenho dos algoritmos na prática com a fundamentação teórica (análise assintótica) em diferentes cenários de n e q.
- Todo o material e código fonte utilizados estão disponíveis no repositório do GitHub: `https://github.com/TalesLimaOliveira/Analise`.

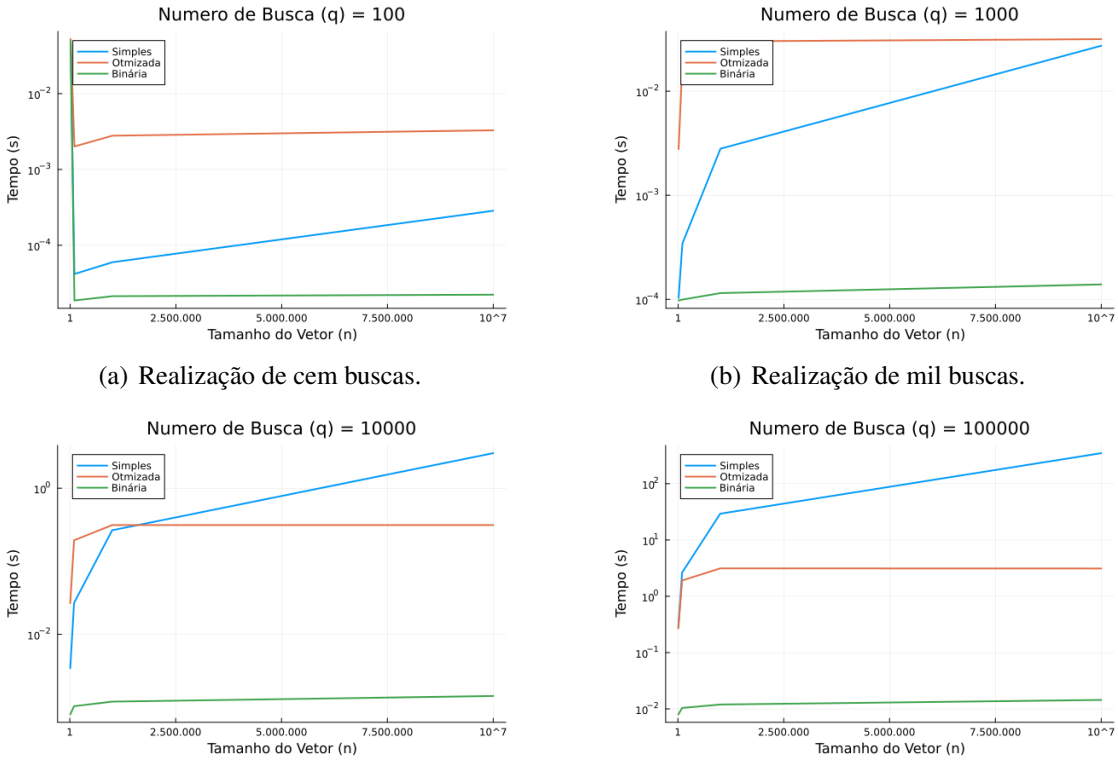
Tabela 1. Configurações de hardware dos ambientes.

Ambientes	Processador	RAM	OS
Google Colab	Intel Xeon	4,0 GB	Linux
Máquina A	Intel i5-9400F	16,0 GB	Win 11

4. Experimentos e Analise de Dados

Apesar das execuções terem sido realizadas em diferentes ambientes, os resultados obtidos não apresentaram variações significativas. Portanto, para simplificar a análise, consideraremos apenas os dados coletados na Máquina A.

4.1. Resultados obtidos



(a) Realização de cem buscas.

(b) Realização de mil buscas.

(c) Realização de dez mil.

(d) Realização de cem mil buscas.

Figura 4. Comparação de desempenho dos algoritmos de busca para diferentes números de buscas.

4.2. Busca Sequencial Simples

O codigo implementado para a busca sequencial simples foi:

```
# Simple sequential search function
function simple_sequential_search(vector, key)
    for i in 1:length(vector)
        if key == vector[i]
            return i  # Key index
        end
    end
    return -1  # Key not found
end
```

O código implementado possui uma complexidade ciclotemática de 3, podendo ser calculada da seguinte forma:

→ *1 Função + 1 laço de repetição (for) + 1 Desvio condicional (if)*

Como pode ser visto nas figuras 4(a), 4(b), 4(c) e 4(d) sua eficiência se deteriora significativamente para grandes conjuntos de dados e grandes números de pesquisas. É

adequada para entradas pequenas não ordenadas, onde o custo da ordenação não compensa.

4.3. Busca Sequencial Otimizada

O codigo para a busca sequencial otmizada foi implementado da seguinte forma:

```
# Optimized sequential search function
function optimized_sequential_search(sorted_vector, key)
    for i in length(sorted_vector)
        if key == vector[i]
            return i  # Key index
        elseif key < vector[i]
            return -1 # Key not found
        end
    end
    return -1  # Key not found
end
```

O código implementado possui uma complexidade ciclotemática de 4, podendo ser calculada da seguinte forma:

$\rightarrow$ *1 Função + 1 Laço de repetição (for) + 2 Desvio condicional (if)*

A busca sequencial otimizada só se mostrou mais eficiente que a sequencial simples somente após o valor de n maior dez mil, como pode ser visto nas figuras 4(c) e 4(d), onde o custo da ordenação $O(n \log n)$ não compesava o tempo decorrido pela simples, porém para listas grandes, verificar se o elemento já não estava presente na lista, mostrou ser um pouco mais eficiente.

4.4. Busca Binária

O codigo para a busca binaria foi implementado da seguinte forma:

```
# Binary search function
function binary_search(sorted_vector, key)
    low, high = 1, length(sorted_vector)

    while low <= high
        mid = (low + high) ÷ 2
        if sorted_vector[mid] == key
            return mid  # Key index
        elseif sorted_vector[mid] < key
            low = mid + 1
        else
            high = mid - 1
        end
    end
    return -1  # Key not found
end
```

O código implementado possui uma complexidade ciclotemática de 5, podendo ser calculada da seguinte forma:

$\rightarrow$ *1 Função + 1 laço de repetição (while) + 3 Desvio condicional (if - while)*

Tabela 2. Complexidade ciclotemática dos algoritmos utilizados.

Algoritmos	Complexidade Ciclotemática
Busca Sequencial Simples	3
Busca Sequencial Otimizada	4
Busca Binária	5

Como pode ser visto na tabela 2 a busca binária tem a complexidade ciclotematica maior, porém, mesmo assim se mostrou ser o método mais eficiente para encontrar elementos em listas ordenadas. Poís como visto antes, sua complexidade é $O(\log n)$, o que supera as busca lineares de $O(n)$. Porém, vale resaltar que caso a entrada não esteja ordenada, devemos adicionar um custo de $O(n \log n)$ para a ordenação da lista de dados.

5. Conclusão

Os resultados obtidos a partir da execução dos algoritmos de busca confirmaram a eficácia teórica esperada para cada método.

A busca binária se destacou como o método mais eficiente para encontrar elementos em listas ordenadas. Sua complexidade logarítmica $O(\log n)$ permite que ela execute buscas de maneira muito mais rápida em listas grandes, comparada às buscas sequenciais. A evidência empírica suportou a análise teórica, demonstrando que a busca binária é superior em termos de desempenho, especialmente quando o número de elementos n aumenta. Isso ocorre porque, a cada iteração, a busca binária reduz o tamanho da lista pela metade, resultando em um número reduzido de comparações.

Por outro lado, a busca sequencial otimizada apresentou um desempenho intermediário. Esta variante da busca sequencial tira proveito da ordenação dos dados para evitar comparações desnecessárias. A melhoria na eficiência em comparação com a busca sequencial simples só se tornou evidente em listas grandes, especialmente para valores de n superiores a 10^4. Mesmo assim, o custo de ordenar os dados ($O(n \log n)$) ainda pode superar os benefícios da busca otimizada em algumas situações. Isso foi claramente visível nas comparações onde a busca sequencial otimizada não conseguiu igualar a eficiência da busca binária, apesar de sua vantagem sobre a busca sequencial simples.

A busca sequencial simples mostrou-se mais adequada para listas pequenas ou quando os dados não estão ordenados, como demonstrado pelos resultados em listas com $n = 10^4$ e 10^5. Nesses casos, a simplicidade do algoritmo e a ausência de custos de ordenação tornaram a busca sequencial simples uma escolha prática e eficiente. No entanto, sua eficiência decai significativamente com o aumento do tamanho da lista, tornando-a menos adequada para grandes volumes de dados.

Em suma, a escolha do algoritmo de busca deve considerar não apenas a complexidade teórica, mas também o contexto prático, como o tamanho da lista, a necessidade de ordenação e o custo associado a essa ordenação.

Referências

Alves, J. R. (2018). Analise de desempenho dos algoritmos de busca. *Researchgate*. [Online; accessed 07-August-2024].

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Algoritmos: Teorias e Práticas*. Rio de Janeiro - Elsevier, 3th edition.

GeeksforGeeks (01 Abr, 2024c). Searching algorithms. [Online; accessed 07-August-2024].

GeeksforGeeks (05 Jul, 2024b). Linear search algorithm. [Online; accessed 07-August-2024].

GeeksforGeeks (26 Jul, 2024a). Binary search algorithm. [Online; accessed 07-August-2024].

Tenembaum, A. M., Augenstein, M. J., and Lagnsam, Y. (1995). *Estrutura de Dados Usando C*. SãoPaulo - Markron Books.

Ziviani, N. (1999). *Projeto de Algoritmos: com implementação em Pascal e C*. SãoPaulo - Pioneira, 4th edition.