Estruturas de Dados e Análise de Algoritmos - EDAA
Avaliação 1.1 – Algoritmos de Busca – Parte 1

Rodrigo Schmidt Nurmberg[1]

[1]Programa de Pós Graduação em Ciência da Computação (PPGCOMP)
Universidade Estadual do Oeste do Paraná (UNIOESTE)
Rua Universitária, 2069 Bloco B – Bairro Universitário – 85819-110 – Cascavel – PR

rodrigo.nurmberg@unioeste.br

Resumo. *Buscas são operações fundamentais na computação. Existem diversos métodos de busca, a seleção do mais adequado depende da natureza da aplicação. Neste trabalho foram implementados e discutidos 3 métodos clássicos de busca: sequencial, por saltos e binária. Os métodos foram comparados empiricamente, quanto ao número de comparações e ao tempo de execução, na busca de valores inteiros em arranjos estáticos, com base em dois cenários: aleatório e pior caso. Os resultados mostram que o algoritmo de propósito mais geral, busca sequencial, apesar de apresentar maior tempo médio de execução é mais rápido em buscas únicas que os demais métodos, que demandam ordenação prévia dos dados.*

1. Introdução

Busca e ordenação são operações essenciais na computação, como são utilizadas com frequência, mesmo pequenas otimizações podem gerar grandes impactos, principalmente ao operarem sobre grandes conjuntos de dados.

A operação de busca consiste em determinar a posição ocupada por um elemento (chave de busca) em um conjunto de dados[1].

Na ordenação, para um dado conjunto de entrada $\langle a_1, a_2, ..., a_n \rangle$, retorna-se uma permutação da sequência de entrada $\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \rangle | a_i^{'} \leq a_{i+1}^{'} \forall i_1^{n-1}$[Cormen et al. 2012][2].

A escolha do algoritmo mais adequado, depende da natureza de cada aplicação e deve considerar, entre outros fatores, se os dados encontram-se ordenados, a forma como os dados estão armazenados (unidades de acesso aleatório ou sequencial, cada qual com sua velocidade), as estruturas de dados utilizadas (vetores, listas encadeadas, etc), quantidade de itens a ordenar e como estão distribuídos (uniformemente ou não) e também a arquitetura do computador (com diferentes operações e seus custos).

1.1. Algoritmos de busca

A seguir, são apresentados os pseudo-códigos dos 3 algoritmos de busca analisados neste trabalho, cujas características são sumarizadas no quadro comparativo da Tabela 1.

Os algoritmos tem como entrada o arranjo $A_{i=1}^{n} \langle a_1, a_2, ..., a_n \rangle$ e a chave de busca x, e como saída o índice i do 1º elemento $a_i | a_i = x$ ou falso caso $x \notin A$.

[1]Caso o elemento não esteja no conjunto, costuma-se retornar vazio ou falso.

[2]Também é possível ordenar os valores de forma decrescente: $\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \rangle | a_i^{'} \geq a_{i+1}^{'} \forall i_1^{n-1}$

[3]Custos para busca por salto simples, quando o custo do salto é igual ao custo da busca sequencial.

Algoritmo 1 - Pseudo-código busca sequencial [Knuth 1998, p. 396]

```
1: i ← 1
2: enquanto i ≤ n faça
3:    se x = A[i] então
4:       retorne i
5:    fim se
6:    i ← i + 1
7: fim enquanto
8: retorne falso
```

Algoritmo 2 - Pseudo-código busca por saltos [Shneiderman 1978]

```
1: salto ← ⌊√n⌋
2: i ← 1
3: j ← min(salto, n) + 1
4: enquanto j ≤ n faça
5:    se A[j] ≥ x então
6:       enquanto i ≤ j faça
7:          se x = A[i] então
8:             retorne i
9:          fim se
10:         i ← i + 1
11:      fim enquanto
12:      retorne falso
13:   fim se
14:   i ← j
15:   j ← min(j + salto, n)
16: fim enquanto
17: retorne falso
```

Algoritmo 3 - Pseudo-código busca binária [Knuth 1998, p. 410]

```
1: esquerda ← 1
2: direita ← n + 1
3: enquanto esquerda ≤ direita faça
4:    meio ← esquerda + ⌊(direita − esquerda)/2⌋
5:    se x = A[meio] então
6:       retorne i
7:    senão se x > A[meio] então
8:       esquerda ← meio + 1
9:    senão
10:      direita ← meio − 1
11:   fim se
12: fim enquanto
13: retorne falso
```

Algoritmo	Sequencial	Por Saltos	Binária
Funcionamento	Compara a chave de busca, elemento a elemento, do início ao fim do arranjo, até encontrá-la ou exaurir o conjunto	Salta até encontrar o bloco que deveria conter a chave de busca, então realiza uma busca sequencial no bloco.	Reiteradamente, particiona o arranjo ao meio, comparando a chave de busca com o elemento central da partição, até encontrá-la ou obter uma partição com um único elemento.
Abordagem	Linear	Divisão e conquista	Divisão e conquista
Melhor caso	O(1) 1º elemento do arranjo	O(2) 1º elemento do 1º bloco	O(1) Elemento central do arranjo
Caso médio	O(N)	O($\sqrt{N}$)[3]	O($log_2 N$)
Pior caso	O(N) Último elemento do arranjo ou inexistente	O($\sqrt{N}$)[3] Último elemento do último bloco	O($log_2 N$) Último elemento da última partição
Ordenação	Opcional	Obrigatória	Obrigatória
Estrutura de dados	Vetores Listas encadeadas	Vetores	Vetores
Linhas Python	5	15	10
Observações	Fácil de compreender e implementar Funciona em diversas estruturas de dados Não requer ordenação prévia Custoso (tempo e comparações)	O mais complexo dos 3 métodos Requer adaptações para encadeamento Necessita de ordenação prévia Mais eficiente que a busca sequencial	Relativamente simples Requer adaptações para encadeamento Necessita de ordenação prévia Mais eficiente dos 3 métodos

Tabela 1. Quadro comparativo dos algoritmos de busca.
Fonte: Elaborado com base em [Sultana et al. 2017].

2. Materiais e Métodos

Os algoritmos da Seção 1.1, foram implementados[4] na linguagem Python e comparados empiricamente na busca em arranjos estáticos, de valores inteiros sem repetições, com tamanhos entre 100.000 e 1.000.000 de elementos, em intervalos de 100.000.

Para cada tamanho de arranjo, foram gerados 2 cenários de testes:

- Aleatório: Arranjo e chave de busca gerados aleatoriamente[5]. Para cada arranjo foram executadas 100 buscas com chaves distintas.
- Pior caso: Chave de busca selecionada para maximizar o número de comparações de cada algoritmo[6]. Foram executadas 3 buscas.

A cada execução, registrou-se o número de comparações ($a_i = x$) e o tempo de execução (em ms) de cada um dos três métodos de busca implementados. Foram calculadas as médias ($\bar{x}$) e os desvios padrão (σ).

Além disso, para o cenário aleatório, também foi registrado o percentual de vezes em que a chave estava presente no arranjo, coluna $x \in A$ da Tabela 2.

Para os métodos de busca que necessitam de arranjos ordenados, o tempo de ordenação[7] foi registrado separadamente do tempo de execução das buscas[8].

Os testes foram conduzidos em um ambiente de desenvolvimento colaborativo[9], cujos recursos são compartilhados e estão sujeitos à flutuações de disponibilidade. Para minimizar o impacto dessas flutuações nos resultados, foram realizadas 2 execuções adicionais para cada teste, e os 2 maiores tempos de execução foram descartados.

3. Resultados

A Tabela 2 sumariza os resultados dos testes realizados e está dividida em 3 conjuntos de colunas: cenários aleatório e pior caso, e ordenação.

[4]Código e outras informações disponíveis em https://github.com/rsn86/edaa-01
[5]Valores entre 0 e 100 vezes o tamanho do arranjo.
[6]Arranjo do cenário aleatório, ordenado crescentemente, e chave de busca o último elemento do arranjo.
[7]Utilizando o método de ordenação padrão da linguagem Python - TimSort.
[8]Calculou-se a média e desvio padrão de 3 execuções
[9]https://research.google.com/colaboratory/faq.html

Para a ordenação são apresentados apenas os tempos médios e os desvios padrão, uma vez que se utilizou a implementação disponibilizada pela linguagem de programação e não se tem acesso aos números de comparações e trocas realizadas.

Para o cenário de pior caso, apresentou-se o número de comparações (Comp. x), e não a média e desvio padrão, pois em todas as execuções foram realizadas o máximo de comparações de cada método, resultando num valor fixo e portanto $\sigma = 0$.

Já no cenário aleatório, a coluna $x \in A$, traz apenas a média das vezes em que a chave esta presente no arranjo, o desvio padrão é irrelevante para análise em questão.

		Aleatório					Pior Caso			Ordenação	
		Tempo (ms)		Comp.		$x \in A$	Tempo (ms)		Comp.	Tempo (ms)	
Tam.	Busca	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	$\bar{x}$	σ	x	$\bar{x}$	σ
100.000	Binária	0,02	0,00	15,98	1,33	0,57	0,01	0,00	17	31,76	0,28
	Por saltos	0,16	0,07	382,17	131,61	0,59	0,19	0,01	633		
	Sequencial	22,17	10,35	71.065,25	34.653,15	0,59	25,97	0,51	100.000		
200.000	Binária	0,02	0,00	17,31	0,99	0,48	0,02	0,00	18	111,73	3,95
	Por saltos	0,21	0,09	556,72	175,55	0,46	0,33	0,01	895		
	Sequencial	49,25	18,93	150.370,68	66.744,42	0,48	58,95	0,98	200.000		
300.000	Binária	0,02	0,00	17,68	1,53	0,60	0,02	0,00	19	118,69	1,59
	Por saltos	0,27	0,13	637,97	252,79	0,60	0,60	0,14	1.096		
	Sequencial	74,57	33,19	201.535,38	102.630,73	0,61	108,03	13,77	300.000		
400.000	Binária	0,02	0,00	18,05	1,34	0,50	0,02	0,01	19	167,89	7,89
	Por saltos	0,34	0,14	794,77	247,17	0,50	0,69	0,15	1.265		
	Sequencial	105,87	36,56	302.179,91	126.676,43	0,50	154,11	14,50	400.000		
500.000	Binária	0,02	0,00	18,48	1,06	0,48	0,02	0,00	19	209,18	4,45
	Por saltos	0,38	0,16	908,12	283,17	0,50	0,54	0,02	1.415		
	Sequencial	139,39	44,74	390.926,05	151.481,00	0,50	158,79	2,41	500.000		
600.000	Binária	0,02	0,00	18,86	1,19	0,42	0,02	0,00	20	270,69	6,39
	Por saltos	0,38	0,16	938,88	325,90	0,42	0,60	0,02	1.550		
	Sequencial	168,52	56,69	468.179,95	190.295,27	0,42	197,24	3,36	600.000		
700.000	Binária	0,02	0,00	18,98	1,23	0,48	0,02	0,00	20	467,85	29,30
	Por saltos	0,44	0,19	1.033,14	386,35	0,48	0,66	0,01	1.674		
	Sequencial	191,76	62,69	532.646,87	218.858,37	0,48	232,37	6,09	700.000		
800.000	Binária	0,02	0,00	19,28	1,04	0,52	0,02	0,01	20	381,68	22,68
	Por saltos	0,45	0,18	1.089,20	403,75	0,53	0,99	0,22	1.789		
	Sequencial	211,25	76,47	568.020,71	268.519,64	0,53	314,76	34,61	800.000		
900.000	Binária	0,02	0,00	19,52	0,83	0,50	0,02	0,00	20	568,04	112,58
	Por saltos	0,52	0,24	1.224,75	456,39	0,50	0,78	0,00	1.898		
	Sequencial	253,15	88,47	683.849,38	306.246,35	0,50	299,04	5,37	900.000		
1.000.000	Binária	0,02	0,00	19,55	1,00	0,55	0,02	0,00	20	496,31	22,11
	Por saltos	0,52	0,26	1.223,49	437,24	0,55	0,98	0,29	2.000		
	Sequencial	270,34	100,09	710.240,35	349.607,52	0,55	414,40	64,77	1.000.000		

Tabela 2. Comparação empírica do desempenho dos métodos de busca

A influência da fluatação na disponibilidade dos recursos pode ser observada ao comparar os valores dos tempos de ordenação no cenário pior caso. Sendo o arranjo, a chave de busca e o número de comparações idênticos entre as execuções, esperava-se que os desvios padrão fossem pequenos, próximos a zero. Porém, apresentaram valores absolutos e relativos elevados, chegando a 64,77 ms e CV[10] de 15,63%, para 1 milhão de elementos. Ainda, pode ser facilmente observada, neste mesmo cenário, pelos menores tempos de busca em conjuntos maiores, por ex. nos tempos para 800 e 900 mil elementos.

O método de ordenação utilizado, beneficia-se da existência de sequências ordenadas[11] no conjunto de dados. Isso pode explicar, pelo menos parcialmente, a aparente

[10]Coeficiente de variação: $CV = 100\frac{\sigma}{\bar{x}}$

[11]Máximas sequências monotônicas - sequências de maior comprimento sem inversão na ordenação.

incongruência nos tempos de ordenação, nos quais arranjos maiores, com 800 mil e 1 milhão de elementos, apresentaram tempos menores que os arranjos de 700 mil e 900 mil elementos. Conforme [Auger et al. 2018], o custo do TimSort é $O(n + n\mathcal{H})$, a entropia $\mathcal{H}$[12] depende do número e comprimento das sequencias parcialmente ordenadas. Porém, seu cálculo não é trivial e foge do escopo deste trabalho, sendo assim, $\mathcal{H}$ não foi calculada.

Os custos encontrados para o pior caso, Tabela2, correspondem aos descritos na literatura e sumarizados no quadro da Tabela 1, divergindo apenas para o algoritmo de buscas por saltos, sugerindo que o custo do salto é diferente do custo da busca sequencial[13] ou que a literatura não considera o custo da busca sequencial, uma vez que o custo encontrado ($2\sqrt{N}$), corresponde ao custo reportado na literatura ($\sqrt{N}$), acrescido do custo da busca sequencial em um bloco de tamanho $\sqrt{N}$.

4. Conclusões

Conhecendo-se o cenário de utilização, é possível selecionar algoritmos mais adequados e performáticos. [Cappelle et al. 2021] apresenta um interessante levantamento sobre diversos algoritmos de busca em vetores. Nesse sentido, a principal conclusão deste estudo é que o custo de ordenação justifica-se para os casos de múltiplas buscas sobre o mesmo arranjo. E, apesar do algoritmo de busca binária apresentar o melhor desempenho, tanto em termos de tempo quanto comparações, ele não é adequado para vetores com muitas inserções e remoções, devido ao custo associado em se manter o vetor ordenado. Nesse caso, pode-se considerar a utilização de árvores de busca binária, preferencialmente as balanceadas, estruturas de dados não lineares, cujas restrições asseguram propriedades interessantes, garantindo bons custos de inserção, remoção e busca.

O fato de ter sido adotado um arranjo com valores espalhados, não contínuos, e chaves de buscas que podiam estar ausentes do conjunto, apesar de contribuir para o aumento nos tempos de execução e no número de comparações, aproxima o cenário de testes de um cenário de uso real, no qual não se pode controlar a distribuição dos valores e não se tem certeza sobre a presença da chave de busca no conjunto.

A influência da entropia carece de investigação. Alternativamente, poderia ser adotado um método menos suscetível a tal fator.

A execução em ambiente compartilhado permitiu visualizar as relações de grandeza entre os métodos de busca e os tamanhos dos arranjos, porém valores mais precisos poderiam ser obtidos em um ambiente controlado, com alocação dedicada de recursos.

A degradação na linearidade dos tempos de busca, observável no cenário pior caso da busca sequencial, com o crescimento do tamanho dos arranjos, pode estar relacionada à hierarquia de memória e a necessidade de movimentação dos dados. Um aprofundamento do estudo envolveria maiores conhecimentos sobre a arquitetura do ambiente, como modelo da CPU, tamanho e temporização das memórias cache e RAM.

[12] $\mathcal{H} = -\sum \frac{r_i}{n} log_2(\frac{r_i}{n})$, $r_i -$ comprimento da i-ésima sequência monotônica máxima.

[13] Nesse caso o tamanho do salto deveria ser $\sqrt{(a/b)N}$, com custo $\sqrt{abN}$. Custos, a: salto, b: busca sequencial. Porém o tamanho do salto utilizado foi $\sqrt{N}$.

Referências

Auger, N., Jugé, V., Nicaud, C., and Pivoteau, C. (2018). On the Worst-Case Complexity of TimSort. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *LIPICS*, pages 4:1–4:13, Helsinki, Finland.

Cappelle, M. R., Foulds, L. R., and Longo, H. J. (2021). Searching Monotone Arrays: A Survey. *Algorithms*, 15(1):10.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2012). *Algoritmos: teoria e prática*. Campus, Rio de Janeiro. ISBN: 9788535236996.

Knuth, D. (1998). *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education.

Shneiderman, B. (1978). Jump searching: a fast sequential search technique. *Communications of the ACM*, 21(10):831–834.

Sultana, N., Paira, S., Chandra, S., and Alam, S. K. S. (2017). A brief study and analysis of different searching algorithms. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–4, Coimbatore. IEEE.