Estruturas de Dados e Análise de Algoritmos - EDAA
Avaliação 1.2 – Algoritmos de Busca – Parte 2

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

Abstract. *In this work, search methods: sequential, jump and binary in static arrays, sequential in linked lists and search in binary search trees, were empirically analyzed, regarding the number of comparisons and execution time, in the search of integer values, based on two scenarios: random and worst case. The results show that methods with the same cost can present different execution time. Furthermore, the choice of the search method must consider not only it's costs, but also the costs of other operations performed by the application, such as insertion and ordering or balancing. In certain cases, the adoption of a more expensive algorithm can be justified by it's usage scenario.*

Resumo. *Os métodos de busca sequencial, por saltos e binária em arranjos estáticos, sequencial em listas ligadas e busca em árvores binárias de busca, foram analisados empiricamente, quanto ao número de comparações e ao tempo de execução, na busca de valores inteiros, com base em dois cenários: aleatório e pior caso. Os resultados mostram que métodos com mesmo custo podem apresentar diferenças no tempo de execução. Além disso, a escolha do método de busca deve considerar não apenas os custos do algoritmo em si, mas também os custos das demais operações realizadas pela aplicação, como inserção e ordenação ou balanceamento. Em certos casos, pode-se justificar a adoção de um algoritmo mais custoso tendo em vista o cenário de utilização.*

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 dos algoritmos mais adequados, 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

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

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

com sua velocidade), as estruturas de dados utilizadas (vetores, listas encadeadas, árvores, 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 associados).

1.1. Algoritmos de busca

A seguir, são apresentados os pseudo-códigos dos 5 algoritmos de busca analisados neste trabalho, cujas características são sumarizadas nos quadros comparativo das Tabelas 1 e 2. Cabe observar que optou-se pelas versões iterativas dos algoritmos.

Os algoritmos para arranjos estáticos 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 $x \notin A$.

Os algoritmos para estruturas dinâmicas, recebem como entada um ponteiro para o nó inicial da estrutura E e a chave de busca x, tendo como saída o 1º elemento $e|e.chave = x$ ou falso caso $x \notin E$.

Algoritmo 1 - 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 - 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
```

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

Algoritmo 3 - 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 4 - Busca sequencial em lista ligada [Cormen et al. 2012, p. 172]

```
1: e ← Lista.inicio
2: enquanto e ≠ NULO e e.chave ≠ x faça
3:     e ← e.proximo
4: fim enquanto
5: retorne e
```

Algoritmo 5 - Busca em árvore binária de busca [Cormen et al. 2012, p. 212]

```
1: e ← Arvore.inicio
2: enquanto e ≠ NULO e e.chave ≠ x faça
3:     se x < e.chave então
4:         e ← e.esquerda
5:     senão
6:         e ← e.direita
7:     fim se
8: fim enquanto
9: retorne e
```

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)	[3]O($\sqrt{N}$)	O($log_2 N$)
Pior caso	O(N) Último elemento do arranjo ou inexistente	[3]O($\sqrt{N}$) 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 em arranjos estáticos.
Fonte: Elaborado com base em [Sultana et al. 2017].

As considerações feitas, no quadro comparativo da Tabela 1, a respeito do algoritmo de busca sequencial, se aplicam tanto às buscas em arranjos estáticos (vetores) quanto às buscas em listas ligadas.

Algoritmo	Busca em Árvore Binária de Busca
Funcionamento	A partir da raiz, desce a árvore, comparando, sucessivamente, a chave de busca com o nó atual até encontrá-la ou atingir um nó folha. Caso a chave seja maior, busca na subárvore direita. Se for menor, na subárvore esquerda.
Abordagem	Divisão e conquista
Melhor caso	O(1) Raiz da árvore
Caso médio	O(h(N)) Sendo h(N) a altura de uma árvore com N elementos[4]
Pior caso	O(N) Folha em uma árvore completamente desbalanceada, cuja altura h(N)=N
Ordenação	Inerente à estrutura de dados
Estrutura de Dados	Árvore binária de busca
Linhas Python	10
Observações	Relativamente simples Bom desempenho no caso médio sem necessitar de ordenação Inserção de valores ordenados desbalanceia a árvore, aumentando os custos das operações

Tabela 2. Quadro descritivo do algoritmo de busca em árvore binária de busca.
Fonte: Elaborado com base em [Goodrich and Tamassia 2015].

2. Materiais e Métodos

Os algoritmos da Seção 1.1, foram implementados[5] na linguagem Python e comparados empiricamente na busca de valores inteiros sem repetições, em estruturas de dados com tamanho variando entre 100.000 e 1.000.000 elementos, em intervalos de 100.000. Com base em 2 cenários de testes:

- Aleatório: Elementos e chave de busca gerados aleatoriamente[6]. Para cada tamanho das estruturas foram executadas 100 buscas com chaves distintas.

[4]Apesar da altura h(N) estar no intervalo $[log_2 N, N]$, quando a árvore é construída aleatoriamente, sua altura será $O(log_2 N)$, com grande probabilidade [Goodrich and Tamassia 2015].

[5]Código e outras informações disponíveis em https://github.com/rsn86/edaa-02

[6]Valores no intervalo $[0, 100 * N]$, onde N é a quantidade de elementos.

- Pior caso: Chave de busca selecionada para maximizar o número de comparações de cada algoritmo[7]. Foram executadas 3 buscas.

Para cada execução, registrou-se o número de comparações ($a_i = x$) e o tempo de execução (em ms) de cada um dos cinco métodos de busca implementados. Foram calculadas as médias ($\bar{x}$) e os desvios padrão (σ). Para as árvores de busca binária, registrou-se a altura da árvore.

Além disso, para o cenário aleatório, também foi registrado o percentual de vezes em que a chave estava presente no arranjo, coluna $x \in A$ das Tabelas 3 e 4.

Para os métodos de busca que necessitam de arranjos ordenados, o tempo de ordenação[8] foi registrado separadamente do tempo de execução das buscas[9].

Os testes foram conduzidos em um ambiente de desenvolvimento colaborativo[10], 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 3 sumariza os resultados dos testes em arranjos estáticos e está dividida em 3 conjuntos de colunas: cenários aleatório e pior caso, e ordenação. Organizada de forma similar, a Tabela 4 sumariza os resultados para os testes nas estruturas dinâmicas.

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 estava presente no arranjo, o desvio padrão é irrelevante para análise em questão.

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[11] 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[12] no conjunto de dados. Isso pode explicar, pelo menos parcialmente, a aparente

[7] Elementos do cenário aleatório, ordenados crescentemente, e chave de busca o maior elemento, correspondendo ao último elemento da estrutura de dados.

[8] Utilizando o método de ordenação padrão da linguagem Python - TimSort.

[9] Calculou-se a média e desvio padrão de 3 execuções

[10] https://research.google.com/colaboratory/faq.html

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

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

		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		
	Por saltos	0,16	0,07	382,17	131,61	0,59	0,19	0,01	633	31,76	0,28
	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		
	Por saltos	0,21	0,09	556,72	175,55	0,46	0,33	0,01	895	111,73	3,95
	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		
	Por saltos	0,27	0,13	637,97	252,79	0,60	0,60	0,14	1.096	118,69	1,59
	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		
	Por saltos	0,34	0,14	794,77	247,17	0,50	0,69	0,15	1.265	167,89	7,89
	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		
	Por saltos	0,38	0,16	908,12	283,17	0,50	0,54	0,02	1.415	209,18	4,45
	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		
	Por saltos	0,38	0,16	938,88	325,90	0,42	0,60	0,02	1.550	270,69	6,39
	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		
	Por saltos	0,44	0,19	1.033,14	386,35	0,48	0,66	0,01	1.674	467,85	29,30
	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		
	Por saltos	0,45	0,18	1.089,20	403,75	0,53	0,99	0,22	1.789	381,68	22,68
	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		
	Por saltos	0,52	0,24	1.224,75	456,39	0,50	0,78	0,00	1.898	568,04	112,58
	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		
	Por saltos	0,52	0,26	1.223,49	437,24	0,55	0,98	0,29	2.000	496,31	22,11
	Sequencial	270,34	100,09	710.240,35	349.607,52	0,55	414,40	64,77	1.000.000		

Tabela 3. Desempenho de métodos de busca em arranjos estáticos

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 $O(N+N\mathcal{H})$[13], a entropia $\mathcal{H}$[14] depende do número e comprimento das sequencias parcialmente ordenadas, porém o cálculo não é trivial e foge do escopo deste trabalho, sendo assim, $\mathcal{H}$ não foi calculada.

Os custos da busca binária, para os cenários aleatório e pior caso, são assintoticamente iguais e empiricamente ficaram muito próximos. Foram ligeiramente inferiores aos encontrados no custo médio da busca em árvores binárias de busca no cenário aleatório, que também é $O(log_2 N)$[4]. Porém, a árvore de busca dispensa a ordenação prévia. Na verdade, a inserção de valores ordenados leva ao desbalanceamento da árvore de busca, aumentando sua altura e, consequentemente, elevando os custos das buscas, que no caso extremo pode chegar a $O(N)$ [Gonnet 1984, p.72].

A diferença de custos, entre algoritmos de mesmo custo assintótico, ocorre pois o custo assintótico (O) considera apenas o termo positivo de maior grau da equação de custo, desconsiderando os coeficientes do polinômio e os termos de menor grau.

Os custos encontrados para o pior caso, Tabelas 3 e 4, correspondem aos descritos na literatura e sumarizados nos quadros comparativos das Tabelas 1 e 2, divergindo apenas

[13]De forma simplificada, assume-se o custo como $O(N * log_2 N)$.

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

		Aleatório						Pior Caso			
		Tempo (ms)		Comp.		$x \in A$	Alt.	Tempo (ms)		Comp.	Alt.
Tam.	Busca	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	$\bar{x}$	$\bar{x}$	σ	$\bar{x}$	$\bar{x}$
100.000	Árvore	0,03	0,01	21,08	4,70	0,58	40	69,50	18,69	100.000	100.000
	Lista	27,93	17,52	71.065,25	34.653,15	0,59	-	61,26	17,09	100.000	-
200.000	Árvore	0,02	0,00	24,28	4,57	0,47	43	68,77	1,13	200.000	200.000
	Lista	18,00	7,98	150.370,68	66.744,42	0,48	-	52,41	1,95	200.000	-
300.000	Árvore	0,03	0,01	22,95	4,91	0,60	43	100,89	1,83	300.000	300.000
	Lista	37,71	23,56	203.712,57	101.096,51	0,61	-	74,45	2,20	300.000	-
400.000	Árvore	0,03	0,01	26,75	5,71	0,50	50	146,82	7,95	400.000	400.000
	Lista	37,02	15,90	301.224,55	127.406,83	0,50	-	107,10	1,59	400.000	-
500.000	Árvore	0,03	0,01	25,60	4,86	0,48	49	173,03	3,49	500.000	500.000
	Lista	58,07	34,04	390.926,05	151.481,00	0,50	-	132,88	3,80	500.000	-
600.000	Árvore	0,03	0,01	25,07	4,90	0,42	49	282,47	67,07	600.000	600.000
	Lista	66,13	39,36	468.179,95	190.295,27	0,42	-	235,14	42,73	600.000	-
700.000	Árvore	0,03	0,01	25,53	4,91	0,47	46	271,46	4,74	700.000	700.000
	Lista	63,11	26,32	534.599,04	217.092,75	0,48	-	184,18	2,37	700.000	-
800.000	Árvore	0,03	0,01	26,82	4,21	0,53	47	289,57	3,67	800.000	800.000
	Lista	77,19	41,24	575.746,46	263.181,72	0,53	-	214,05	2,57	800.000	-
900.000	Árvore	0,03	0,01	25,02	5,35	0,50	48	357,71	20,99	900.000	900.000
	Lista	83,00	40,53	683.849,38	306.246,35	0,50	-	263,80	3,68	900.000	-
1.000.000	Árvore	0,03	0,01	26,45	5,18	0,54	49	365,73	2,51	1.000.000	1.000.000
	Lista	92,88	58,07	710.240,35	349.607,52	0,55	-	273,86	1,06	1.000.000	-

Tabela 4. Desempenho dos métodos de busca em Listas e Árvores

para o algoritmo de buscas por saltos, sugerindo que o custo do salto é diferente do custo da busca sequencial[15] 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 $\sqrt{N}$.

A respeito da altura das árvores de busca, os valores observados para o pior caso condizem com o esperado. Já para o cenário aleatório, apesar de assintoticamente ainda serem equivalentes aos relatados, os valores encontrados foram o dobro do esperado, com h(N) ficando próxima de $2 * log_2 N$.

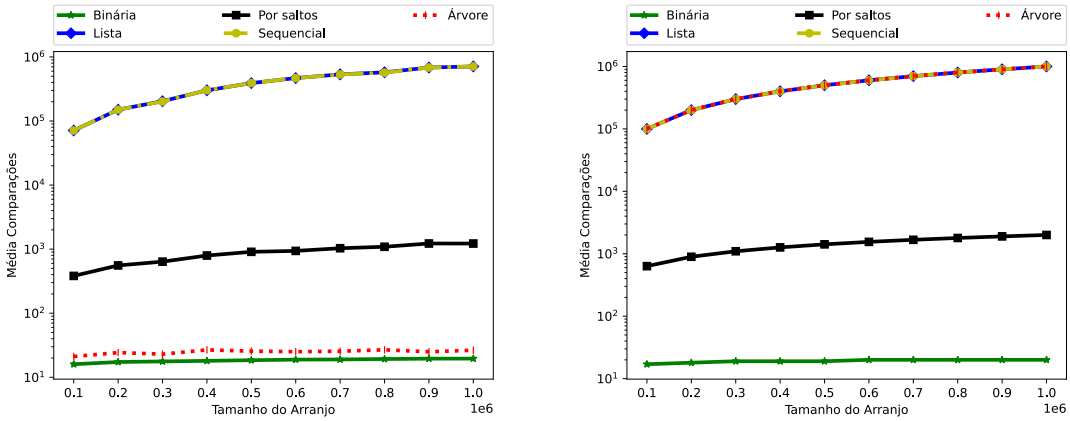
Como os testes foram conduzidos em duas etapas, a primeira referente às buscas em arranjos estáticos e a segunda referente às estruturas dinâmicas, a utilização de uma semente de aleatoriedade foi essencial para assegurar a reprodutibilidade dos experimentos, como pode-se observar pelo número médio de comparações para busca sequencial nas duas situações. As poucas linhas divergentes podem ser explicadas pela técnica empregada para minimizar as flutuações de desempenho, em virtude do ambiente compartilhado, que remove os 2 maiores tempos de cada teste e que podem corresponder a chaves diferentes entre os testes, levando à diferenças no número de comparações.

Para facilitar a comparação entre os métodos de busca para diferentes tamanhos de arranjo, foram elaborados os gráficos a seguir. Dada a diferença na ordem de grandeza entre os métodos, adotou-se a escala logarítmica no eixo Y.

Nas Figuras 1(a) e 1(b) são apresentados os gráficos dos números médios de comparações para os cenários aleatório e pior caso, respectivamente. Como esperado, as buscas sequencial em vetores (linha amarela) e em listas ligadas (linha azul), realizaram o mesmo número médio de comparações, para os dois cenários. No cenário pior

[15]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 $\sqrt{N}$.

caso, a busca em árvores binárias de busca (linha vermelha) se igualou ao número de comparações dos métodos de busca sequencial. Já no cenário aleatório, seu comportamento, tanto em número de comparações quanto em tempo, foi similar à busca binária (linha verde). As buscas por interpolação (linha preta), e principalmente a binária, tiveram custos similares, em termos de comparações, em ambos os cenários.



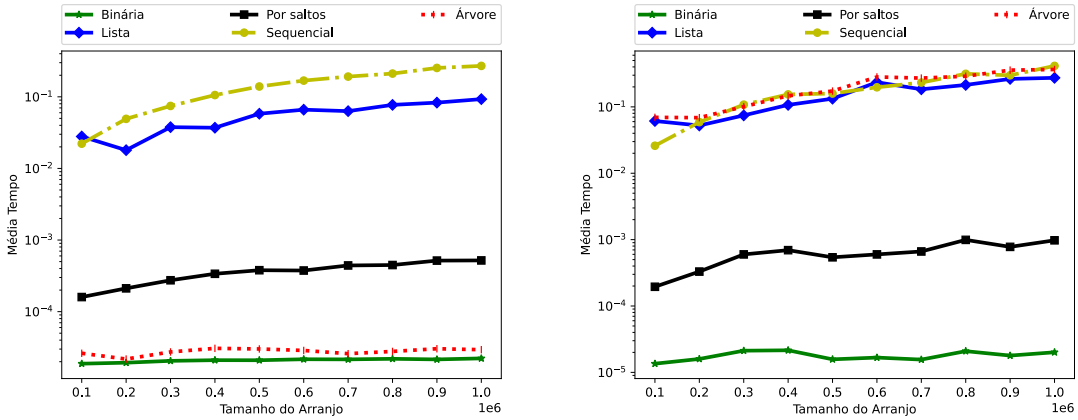
(a) Cenário Aleatório (b) Pior Caso

Figura 1. Média de comparações x Tamanho do Arranjo

As Figuras 2(a) e 2(b) apresentam os gráficos dos tempos médios de busca em relação ao tamanho dos arranjos, para os cenários aleatório e pior caso, respectivamente.

Ainda que os métodos $O(N)$, buscas sequencial em vetores e em listas encadeadas, e o pior caso da busca em árvores binárias de busca, tenham realizado o mesmo número de comparações, houve diferença nos tempos médios de execução. Essa diferença foi pequena no cenário pior caso, porém, até de forma inesperada, a busca sequencial em listas apresentou tempos médios menores que a busca sequencial em arranjos estáticos.

Apesar da aparente estabilidade no número médio de comparações, entre os dois cenários, para as buscas por interpolação e binária, os tempos médios revelaram algumas diferenças, com pequenas flutuações no cenário pior caso.



(a) Cenário Aleatório (b) Pior Caso

Figura 2. Média do tempo x Tamanho do Arranjo

4. Conclusões

Conhecendo-se o cenário de utilização, é possível selecionar algoritmos mais adequados e performáticos. [Cappelle et al. 2021] apresentam 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.

Em árvores binárias de busca, muitas operações podem ser realizadas em $O(h(N))$ [Gonnet 1984, p.71], incluindo inserções e remoções, por isso é importante manter a árvore balanceada, de forma a minimizar sua altura. No cenário de pior caso, onde $h(N) = N$, foi necessário desenvolver um método especial para que a criação da árvore ocorresse em tempo aceitável, pois utilizando o método de inserção elemento a elemento, do cenário aleatório, o tempo de execução do experimento facilmente ultrapassava os 50 minutos, sem concluir. Com a utilização do método especial de inserção, todo o experimento foi concluído em menos de 10 minutos.

Apesar dos custos de criação das estruturas de dados terem sidos desconsiderados neste experimento, ficou evidente que eles podem ter impacto significativo em cenários reais de utilização. Nesse aspecto, as listas encadeadas apresentam os melhores custos, $O(1)$ para inserções nas extremidades e remoções no início, e se a lista for duplamente encadeada, também é possível remover do fim em $O(1)$. Para arranjos estáticos as operações podem demandar deslocamento de elementos ou expansão da estrutura com cópia dos dados para novas regiões de memória, com isso as operações tem custo $O(N)$. Em contrapartida, apenas a busca sequencial, com custo $O(N)$, pode ser utilizado em listas ligadas. Já os arranjos estáticos, por suportarem acesso aleatório (indexação), contam com diversos métodos, incluindo a busca binária, de custo $O(log_2 N)$.

Dada a similaridade dos custos e das abordagens dos métodos de busca binária, em arranjos estáticos ordenados, e a busca em árvores binárias de busca, aleatórias ou balanceadas, seria interessante conduzir um experimento comparando empiricamente os tempos de criação e manutenção das estruturas entre: arranjos estáticos + ordenação e árvores binárias de busca + balanceamento.

O fato de terem sido adotadas estruturas de dados 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 no método de ordenação, e consequentemente seus custos, 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.

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.

Gonnet, G. H. (1984). *Handbook of algorithms and data structures*. International computer science series. Addison-Wesley Pub. Co, London ; Reading, Mass.

Goodrich, M. T. and Tamassia, R. (2015). *Algorithm design and applications*. Wiley, NHoboken, N.J.

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.