Análise da Heurística de Busca Auto-Organizável (Move-to-Front)

Marcos Vinícius Cardoso Moreira Nascimento
Instituto Federal do Maranhão - Campus Monte Castelo

1. Introdução

Listas auto-organizáveis são estruturas de dados que ajustam sua ordem interna com base nos padrões de acesso. O objetivo é manter os elementos mais frequentemente buscados próximos ao início da lista, reduzindo o tempo médio de busca. A heurística Move-to-Front (MTF) segue uma ideia simples: sempre que um elemento é encontrado, ele é movido para o início da lista. Assim, elementos acessados repetidas vezes permanecem nas posições iniciais, acelerando futuras buscas.

2. Funcionamento do Algoritmo

O algoritmo MTF realiza uma busca linear sobre o vetor. A cada passo, o elemento atual é comparado com a chave procurada. Se a chave for encontrada na posição i, o valor é salvo e os elementos das posições 0 até i-1 são deslocados uma posição para a direita. Em seguida, o item encontrado é movido para a posição inicial. Caso a chave não seja encontrada, a lista não é modificada. A complexidade teórica da operação é O(n) no pior caso e também O(n) no caso médio (para dados aleatórios). O melhor caso ocorre quando o elemento está na primeira posição, resultando em custo O(1).

3. Pseudocódigo Anotado com Custos Assintóticos

Algoritmo 1 – Busca Auto-Organizável (Move-to-Front)
Entrada: Vetor V[0..n-1], Chave k
Saída: Vetor possivelmente reorganizado, posição original ou -1

1. para i ← 0 até n-1 faça // O(n)
2. se V[i] == k então // O(1)
3. elemento ← V[i] // O(1)
4. para j ← i até 1 faça // O(i)
5. V[j] ← V[j-1] // O(1)
6. fim-para
7. V[0] ← elemento // O(1)
8. retorne i // O(1)
9. fim-se
10. fim-para
11. retorne -1 // O(n)

Tabela 1 – Análise de Big-O por Linha

Linha	Operação Principal	Custo Total
1	Loop externo	O(n)
2	Comparação de igualdade	O(n)
3	Atribuição	O(1)
4–6	Loop de deslocamento	O(i)
7	Inserção no início	O(1)

8	Retorno	O(1)
11	Retorno final	O(1)
Total	—	O(n)

4. Implementação em C com Custos Assintóticos

```
for (int i = 0; i < n; i++) { // O(n)
  if (vetor[i] == chave) { // O(1)
    int item = vetor[i]; // O(1)
    for (int j = i; j > 0; j--) { vetor[j] = vetor[j - 1]; } // O(i)
    vetor[0] = item; // O(1)
    return i; // O(1)
  }
}
return -1; // O(n)
```

5. Implementação em Java com Custos Assintóticos

```
for (int i = 0; i < n; i++) { // O(n)
  if (vetor.get(i) == chave) {
    int item = vetor.remove(i); // O(n)
    vetor.add(0, item); // O(n)
    return i;
  }
}
return -1; // O(n)
```

6. Implementação em Python com Custos Assintóticos

```
idx = vetor.index(chave) # O(n)
item = vetor.pop(idx) # O(n)
vetor.insert(0, item) # O(n)
return idx
```

7. Comparação Entre Implementações

Linguagem	Operações mais caras	Complexidade Final	Observações
C	Loop interno manual	O(n)	Menor custo real, sem overhead
Java	remove(i), add(0,x)	O(n)	Maior overhead da JVM
Python	index(), pop(), insert()	O(n)	Funções otimizadas em C

8. Conclusão

A heurística Move-to-Front mantém a complexidade linear O(n) em todos os casos assintóticos. As implementações em C, Java e Python refletem o mesmo comportamento do pseudocódigo teórico. A linguagem C apresenta o melhor desempenho prático por operar diretamente sobre a memória. Python mostra bom equilíbrio devido a otimizações internas, e Java tende a apresentar maior variação de tempo devido ao garbage collector. O MTF é indicado em contextos de repetição de acessos, como

caches, compressão de dados e otimização de buscas em listas.

9. Referências

[1] Knuth, D. E. (1984). The Art of Computer Programming.
[2] Sedgewick, R. & Wayne, K. (2011). Algorithms, 4th Edition.
[3] Weiss, M. A. (2012). Data Structures and Algorithm Analysis.
[4] IFMA - Algoritmos e Estruturas de Dados II, 2025.