Uma nova abordagem ao algoritmo Mergesort: divisão inteligente e conquista

A new approach to Mergesort algorithm: Divide smart and conquer

Daniel Nolêto Maciel Luz[1] e João Victor Walcacer Giani[2]

[1] *Universidade Federal do Tocantins, Ciência da Computação, Palmas, Tocantins*

Data de recebimento do manuscrito: 15/10/2024
Data de aceitação do manuscrito: 17/10/2024
Data de publicação: 24/10/2024

Resumo—Algoritmos de ordenação baseados em comparação, como Mergesort, Quicksort e Heapsort, são conhecidos por sua complexidade assintótica eficiente, com tempo de execução de $O(n \log n)$. Contudo, a eficiência prática é igualmente importante, especialmente com o aumento da quantidade de dados. Este estudo propõe uma variação do Mergesort, onde o array é dividido em sub-arrays que podem estar em ordem crescente ou decrescente, ajustando a função de mesclagem (*Merge*) de acordo. Como resultado, foram desenvolvidos novos algoritmos que apresentam melhorias significativas. A complexidade do pior caso permanece em $O(n \log n)$, enquanto no melhor caso é $O(n)$. Na prática, a abordagem proposta demonstrou melhorias notáveis em distribuições Gaussiana e Uniforme, superando o Mergesort tradicional com até 29% de aumento na eficiência.

Palavras-chave— Mergesort Algoritmos de Ordenação, Divisão e Conquista, Análise de complexidade.

Abstract— *Comparison-based algorithms, such as Mergesort, Quicksort, and Heapsort, are considered optimal in terms of asymptotic complexity, with a runtime of $O(n \log n)$. However, in practice, efficiency is also a crucial factor, especially as data volumes increase. This study introduces a variation of Mergesort, where the array is not divided unconditionally. Instead, it is separated into sub-arrays that are in increasing and decreasing order, and the merge function (Merge) is adjusted accordingly. This results in three new algorithms, with the third being the most effective. In this algorithm, the array is divided into alternating sub-arrays of increasing and decreasing order, and the elements do not need to be consecutive. The worst-case complexity remains $O(n \log n)$, but in the best case, it is $O(n)$. In practice, the proposed algorithm demonstrated significant improvements on Gaussian and Uniform distributions, outperforming traditional Mergesort by up to 29% in efficiency.*

Keywords— *Mergesort, Sorting algorithms, Divide and conquer, Complexity analysis.*

I. Introduções

A ordenação e a busca são problemas clássicos na ciência da computação, sendo essenciais para a eficiência de algoritmos em diversas áreas de engenharia. O artigo em análise explora esses problemas, destacando a limitação teórica que impede algoritmos de ordenação baseados em comparação de alcançarem uma complexidade melhor que $O(n \log n)$. Além disso, discute a importância das constantes nos termos assintóticos para o desempenho prático desses algoritmos, ressaltando que o interesse por soluções eficientes aumentou devido à desaceleração na evolução da velocidade dos processadores em relação ao rápido crescimento do volume de dados.

Dados de contato: Daniel N. M. Luz, João V. W. Giani, name.surname@email.com

O estudo foca em dois algoritmos de ordenação por comparação altamente eficientes: o *Mergesort* e o *Quicksort*. O *Mergesort*, introduzido por Jon von Neumann em 1945, adota a estratégia de divisão e conquista, mantendo uma complexidade de $O(n \log n)$ tanto no melhor quanto no pior e no caso médio, mas requer $O(n)$ de memória adicional. Por outro lado, o *Quicksort*, desenvolvido por Tony Hoare entre 1959 e 1960, também segue a estratégia de divisão e conquista, com complexidade de $O(n^2)$ no pior caso e $O(n \log n)$ no melhor e no caso médio, consumindo apenas $O(1)$ de memória adicional.

Os experimentos realizados com dados gerados aleatoriamente, seguindo distribuições Uniforme e Gaussiana, e com dados reais, indicaram que o *Quicksort* pode ser mais eficiente em muitos casos práticos devido ao seu baixo consumo de memória e à performance média superior, especialmente quando implementado com boas estratégias de particionamento.

II. ABORDAGENS

A principal ideia por trás do algoritmo descrito no artigo é dividir um array dado em n sub-arrays ordenados não interseccionados e combiná-los em um único array ordenado. A eficiência desse algoritmo depende de dois fatores fundamentais:

- Como a divisão será feita?
- Como os sub-arrays serão combinados?

O artigo propõe três abordagens para a divisão, sendo que o algoritmo principal é baseado na terceira abordagem, considerada a mais eficiente. Vamos detalhar essas abordagens utilizando exemplos simples.

a. Primeira abordagem

: Um array A com n elementos pode ser dividido em sub-arrays ascendentes de elementos consecutivos em tempo $O(n)$, percorrendo o array do início ao fim. Esses sub-arrays não se sobrepõem (cada termo de A pertence a exatamente um sub-array), e o número de sub-arrays k satisfaz $1 \leq k \leq n$.

Por exemplo, dado um array A com 12 elementos: 3, 10, 7, 4, 6, 1, 5, 8, 9, 2, 12, 11, os sub-arrays ascendentes são mostrados na Figura 1. Nesse exemplo, $n = 12$ e $k = 6$. O objetivo é combinar esses sub-arrays ascendentes em um único array ordenado utilizando a função `Merge`. No melhor caso, $k = 1$, quando o array A já está em ordem crescente. No pior caso, $k = n$, quando A está em ordem inversa.

Algorithm 1 Sort

```
Input: Unsorted array A[0...n−1]
Output: Sorted array A[0...n−1]
procedure Sort(A, n)
B[0] ← −1
t ← 0
k ← 0
for i ← 0 to n−2 do
    if A[i+1] < A[i] then
        t ← t+1
        B[t] ← i
    end if
end for
k ← t+1
B[t+1] ← n−1
while k > 1 do
    t ← 0
    for i ← 0 to k−2 do
        Merge_M(A, i, i + 1, i + 2, B)
        t ← t+1
        B[t] ← B[i+2]
    end for
    B[t+1] ← n−1
    k ← ⌈k/2⌉
end while
return A
procedure Merge_M(A, p, q, r, B)
n1 ← B[q] − B[p]
n2 ← B[r] − B[q]
for i ← 0 to n1−1 do
    L[i] ← A[B[p]+i+1]
end for
for i ← 0 to n2−1 do
    R[i] ← A[B[q]+i+1]
end for
L[n1] ← +∞
R[n2] ← +∞
i ← 0
j ← 0
for k ← 0 to n1+n2−1 do
    if L[i] ≤ R[j] then
        A[B[p]+k+1] ← L[i]
        i ← i+1
    else
        A[B[p]+k+1] ← R[j]
        j ← j+1
    end if
end for
return
```

b. Segunda abordagem

É possível combinar dois sub-arrays, um ascendente e o outro descendente, ou ambos descendentes, em um sub-array ascendente, modificando ligeiramente a função clássica `Merge`. Para isso, precisamos saber se o sub-array é descendente ou ascendente. Durante a mesclagem, podemos percorrer o sub-array descendente do final para o início.

Basta comparar os dois primeiros elementos uma vez para determinar a ordem do sub-array. Portanto, conseguimos escrever o array A como uma combinação ordenada (ascen-

dente ou descendente) de elementos consecutivos. Esses sub-arrays não se sobrepõem, e o número de sub-arrays k é mantido. Na divisão proposta, o array A seria dividido como mostrado na Figura 2.

O número k ainda é 6, mas essa abordagem oferece vantagens em relação à anterior: exceto pelo último, cada sub-array possui pelo menos dois elementos. Se n for um número par e o último sub-array tiver um elemento, então o número de termos em pelo menos um dos sub-arrays anteriores deve ser no mínimo 3. Isso implica que, nesse caso, k será no máximo igual a $n/2$. Se n for ímpar e o último sub-array tiver um elemento, então no pior caso temos $k = \left(\frac{n-1}{2}\right)+1 = \frac{n+1}{2}$. Em outras palavras, na segunda abordagem de divisão, o valor máximo de k será $\lceil n/2 \rceil$. O pseudo-código para a ordenação de acordo com essa segunda abordagem é apresentado no Algoritmo 2.

Algorithm 2 Sorting Algorithm according to the second approach

Input: Unsorted array $A[0...n-1]$
Output: Sorted array $A[0...n-1]$
procedure Sort(A, n)
$B[0] \leftarrow -1$
$t \leftarrow 0$
$cont \leftarrow -1$
if $A[0] \leq A[1]$ **then**
  $cont \leftarrow 1$
end if
for $i \leftarrow 0$ **to** $n-3$ **do**
  if $cont \cdot A[i+1] < cont \cdot A[i]$ **then**
    $t \leftarrow t+1$
    $B[t] \leftarrow i$
    $cont \leftarrow -1$
    if $A[i+1] \leq A[i+2]$ **then**
      $cont \leftarrow 1$
    end if
  end if
end for
if $cont \cdot A[n-1] < cont \cdot A[n-2]$ **then**
  $t \leftarrow t+1$
  $B[t] \leftarrow n-2$
end if
$k \leftarrow t+1$
$B[t+1] \leftarrow n-1$
while $k > 1$ **do**
  $t \leftarrow 0$
  for $i \leftarrow 0$ **to** $k-2$ **do**
    Merge_M2(A, i, i+1, i+2, B)
    $t \leftarrow t+1$
    $B[t] \leftarrow B[i+2]$
    $i \leftarrow i+1$
  end for
  $B[t+1] \leftarrow n-1$
  $k \leftarrow \lceil k/2 \rceil$
end while
return A

Algorithm 3 Merge_M2

procedure Merge_M2(A, p, q, r, B)
$n1 \leftarrow B[q] - B[p]$
$n2 \leftarrow B[r] - B[q]$
$cc \leftarrow A[B[q]] - A[B[p]+1]$
$ct \leftarrow A[B[r]] - A[B[q]+1]$
$cont_L \leftarrow -1$
$cont_R \leftarrow -1$
if $cc > 0$ **then**
  $cont_L \leftarrow 1$
end if
if $ct > 0$ **then**
  $cont_R \leftarrow 1$
end if
if $cont_L = 1$ **then**
  for $i \leftarrow 0$ **to** $n1-1$ **do**
    $L[i] \leftarrow A[B[p]+i+1]$
  end for
else
  for $i \leftarrow 0$ **to** $n1-1$ **do**
    $L[i] \leftarrow A[B[q]-i]$
  end for
end if
if $cont_R = 1$ **then**
  for $i \leftarrow 0$ **to** $n2-1$ **do**
    $R[i] \leftarrow A[B[q]+i+1]$
  end for
else
  for $i \leftarrow 0$ **to** $n2-1$ **do**
    $R[i] \leftarrow A[B[r]-i]$
  end for
end if
$L[n1] \leftarrow +\infty$
$R[n2] \leftarrow +\infty$
$i \leftarrow 0$
$j \leftarrow 0$
for $k \leftarrow 0$ **to** $n1+n2-1$ **do**
  if $L[i] \leq R[j]$ **then**
    $A[B[p]+k+1] \leftarrow L[i]$
    $i \leftarrow i+1$
  else
    $A[B[p]+k+1] \leftarrow R[j]$
    $j \leftarrow j+1$
  end if
end for
return

III. RESULTADOS

No artigo analisado, foram testados dois algoritmos propostos em comparação com algoritmos de ordenação comumente preferidos, em três cenários: dados gerados aleatoriamente (distribuições Gaussiana e Uniforme), dados reais e casos especiais. Os experimentos foram realizados em um computador com sistema operacional Linux (Ubuntu 16.04), equipado com um processador Intel i5 de 1,60 GHz e 8 GB de RAM DDR4. O código foi desenvolvido na linguagem C, utilizando o compilador g++ 5.4.0, e os conjuntos de dados estão disponíveis publicamente para reprodução. Em nosso caso, os experimentos foram realizados em um notebook *Nitro 5* da *Acer* com as seguintes especificações: processador *Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz*, 24GB de memória RAM, armazenamento em SSD de 480GB e uma placa de vídeo dedicada *GTX 1650* com 4GB de VRAM. E a linguagem Python, foi a utilizada para implementação.

Durante os testes, o pivô no algoritmo Quicksort foi escolhido como o termo mais próximo da média aritmética do array, garantindo que o algoritmo funcione em tempo $O(n \log n)$ para arrays com valores distintos. Cada algoritmo foi executado em três arrays de mesmo tamanho, e a média dos tempos de execução foi utilizada para comparação.

a. Experimentos com Conjuntos de Dados com Distribuição Gaussiana e Uniforme

Os tempos de execução dos algoritmos foram utilizados para criar gráficos comparativos. Os experimentos demonstraram que os Algoritmos 1 e 2 superaram o Quicksort e o Mergesort em todos os testes.

b. Conjuntos de Dados Reais

Os resultados com dados reais mostraram que os Algoritmos 1 e 2 tiveram desempenho superior ao Mergesort e ao Quicksort. Em arrays de pequeno tamanho, como no conjunto de dados RCL, ambos se mostraram mais eficientes que outros algoritmos testados.

c. Casos Especiais

Os resultados para conjuntos de dados em ordem reversa indicaram que os Algoritmos 1 e 2 tiveram desempenho superior, com o Algoritmo 1 sendo particularmente eficiente ao identificar subarrays. Para dados constantes, os Algoritmos 1 e 2 foram mais rápidos na identificação de subarrays.

d. Elementos Repetidos nos Arrays

Nos testes com arrays que continham elementos repetidos, os Algoritmos 1 e 2 mostraram-se robustos, enquanto o Quicksort teve desempenho prejudicado. Ambos os algoritmos conseguiram manejar eficientemente a situação sem afetar o tempo de execução.

e. Combinações de Dois Subarrays Crescentes e Decrescentes

Os resultados para arrays compostos por combinações de subarrays mostraram que os Algoritmos 1 e 2 superaram os demais, destacando sua eficácia em cenários de mistura de sequências.

IV. TABELAS

V. CONCLUSÃO

Com o aumento da quantidade de dados provenientes de diversas fontes, a importância dos algoritmos de ordenação cresceu significativamente. Este estudo analisou algoritmos de ordenação, destacando que a operação de divisão adotada nos métodos atuais apresenta abordagens mais eficientes em comparação aos algoritmos clássicos.

Os resultados experimentais mostraram que os algoritmos propostos se destacaram em velocidade de execução, superando métodos tradicionais como Mergesort, Heapsort e Quicksort. No entanto, a estabilidade de alguns algoritmos de ordenação continua sendo um desafio, especialmente em condições de divisão complexas.

Futuros estudos investigarão quais arrays podem ser representados como combinações de subarrays, um crescente e outro decrescente, bem como o número médio de subarrays alternadamente crescentes e decrescentes em um array gerado aleatoriamente.

REFERÊNCIAS

1. Sahin Emrah Amrahov, "A new approach to Mergesort algorithm: Divide smart and conquer, 2024" `https://www.sciencedirect.com/science/article/pii/S0167739X24001225`, (Accessado em 20/10/2024).

2. M. Axtmann, S. Witt, D. Ferizovic, P. Sanders, "Engineering in-place (shared memory) sorting algorithms," ACM Trans. Parallel Comput. 9 (1) (2022) 1–62.

3. Z. Abo-Hammour, O. AbuArqub, S. Momani, N. Shawagfeh, et al., "Optimization solution of Troesch's and Bratu's problems of ordinary type using novel continuous genetic algorithm," Discrete Dyn. Nat. Soc. 2014 (2014).

4. Y. Ar, Ş. Emrah Amrahov, N.A. Gasilov, S. Yigit-Sert, "A new curve fitting based rating prediction algorithm for recommender systems," Kybernetika 58 (3) (2022) 440–455.

5. O. A. Arqub, Z. Abo-Hammour, "Numerical solution of systems of second-order boundary value problems using continuous genetic algorithm," Inf. Sci. 279 (2014) 396–415.

6. N. Kartli, E. Bostanci, M.S. Guzel, "A new algorithm for optimal solution of fixed charge transportation problem," Kybernetika 59 (1) (2023) 45–63.

7. S. Demir, B. Tugrul, "Privacy-preserving trend surface analysis on partitioned data," Knowl.-Based Syst. 144 (2018) 16–20.

8. N.A. Gasilov, "On exact solutions of a class of interval boundary value problems," Kybernetika 58 (3) (2022) 376–399.

9. H.B. Yıldırım, K. Kullu, Ş. Emrah Amrahov, "A graph model and a three-stage algorithm to aid the physically disabled with navigation," Universal Access Inf. Soc. (2023) 1–11.
10. S. Sharma, N. Khodadadi, A.K. Saha, F.S. Gharehchopogh, S. Mirjalili, "Non-dominated sorting advanced butterfly optimization algorithm for multi-objective problems," J. Bionic Eng. 20 (2) (2023) 819–843.
11. I. Saidani, A. Ouni, M. Ahasanuzzaman, S. Hassan, M.W. Mkaouer, A.E. Hassan, "Tracking bad updates in mobile apps: A search-based approach," Empir. Softw. Eng. 27 (4) (2022) 81.
12. H. Ferrada, "A sorting algorithm based on ordered block insertions," J. Comput. Sci. 64 (2022) 101866.
13. H. Zhang, B. Meng, Y. Liang, "Sort race," Softw.- Pract. Exp. 52 (8) (2022) 1867–1878.
14. P. Ganapathi, R. Chowdhury, "Parallel divide-and-conquer algorithms for bubble sort, selection sort and insertion sort," Comput. J. 65 (10) (2022) 2709–2719.
15. A. Mubarak, S. Iqbal, T. Naeem, S. Hussain, "2mm: A new technique for sorting data," Theoret. Comput. Sci. 910 (2022) 68–90.
16. F. Stober, A. Weiß, "On the average case of Merge Insertion," Theory Comput. Syst. 64 (7) (2020) 1197–1224.
17. F. Grabowski, D. Strzalka, "Dynamic behavior of simple insertion sort algorithm," Fund. Inform. 72 (1–3) (2006) 155–165.
18. Ş.E. Amrahov, A.S. Mohammed, F.V. Çelebi, "New and improved search algorithms and precise analysis of their average-case complexity," Future Gener. Comput. Syst. 95 (2019) 743–753.
19. A.S. Mohammed, Ş.E. Amrahov, F.V. Çelebi, "Interpolated binary search: An efficient hybrid search algorithm on ordered datasets," Eng. Sci. Technol., Int. J. 24 (2021) 1072–1079.
20. Z. Marszalek, "The analysis of energy performance in use parallel merge sort algorithms," Inf. Technol. Control 48 (3) (2019) 487–498.
21. D.E. Knuth, "The Art of Computer Programming," vol. 3, Pearson Education, 1997.
22. C.A. Hoare, "Quicksort," Comput. J. 5 (1) (1962) 10–16.
23. S. Gueron, V. Krasnov, "Fast quicksort implementation using AVX instructions," Comput. J. 59 (1) (2016) 83–90.
24. S. Edelkamp, A. Weiß, S. Wild, "QuickXsort: A fast sorting scheme in theory and practice," Algorithmica 82 (3) (2020) 509–588.
25. J. Wassenberg, M. Blacher, J. Giesen, P. Sanders, "Vectorized and performance portable quicksort," Softw.-Pract. Exp. 52 (12) (2022) 2684–2699.
26. M. Aumüller, M. Dietzfelbinger, "Optimal partitioning for dual-pivot quicksort," ACM Trans. Algorithms (TALG) 12 (2) (2015) 1–36.
27. M. Aumüller, M. Dietzfelbinger, P. Klaue, "How good is multi-pivot quicksort?" ACM Trans. Algorithms (TALG) 13 (1) (2016) 1–47.
28. J.W.J. Williams, "Algorithm 232: heapsort," Commun. ACM 7 (6) (1964) 347–348.
29. J.W.J. Williams, "Algorithm 232: heapsort," Commun. ACM 7 (6) (1964) 347–348.