algortimos de ordenacao ? referencias

Hugo Veríssimo
Optimização Combinatória 24/25
Universidade de Aveiro
Aveiro, Portugal
hugoverissimo@ua.pt

João Cardoso
Optimização Combinatória 24/25
Universidade de Aveiro
Aveiro, Portugal
joaopcardoso@ua.pt

***Resumo*—c**

Palavras-chave: zzz, aaa

I. Introdução

Os algoritmos de ordenação são essenciais em várias áreas da optimização combinatória e da investigação operacional, sendo utilizados para organizar dados de forma eficiente, melhorar o desempenho de algoritmos de procura e reduzir a complexidade computacional de problemas em larga escala.

Em contextos de otimização combinatória, onde para vários problemas se torna fundamental a ordenação prévia de dados, como nas heurísticas para o problema Knapsack, no algoritmo de Kruskal para construção de árvores geradoras mínimas, ou em algoritmos gulosos para cobertura de conjuntos, a escolha do método de ordenação pode impactar significativamente a eficiência global da solução.

Este trabalho visa explorar os principais algoritmos de ordenação, destacando as suas características, o seu funcionamento, a complexidade computacional associada e uma análise comparativa entre os mesmos. Considerando as diferentes implementações e constrangimentos associados, exploramos os cenários onde cada algoritmo poderá ser mais útil. O objetivo é compreender como a ordenação pode ser utilizada de forma estratégica para melhorar a eficiência de algoritmos em diversos cenários, e como otimizações na própria ordenação podem levar a avanços significativos em problemas computacionalmente intensivos.

II. Algoritmos de Ordenação

Nesta secção serão apresentados os algoritmos selecionados, detalhando-se individualmente o seu princípio básico de funcionamento, a complexidade temporal, as vantagens e desvantagens, e a aplicabilidade prática em diferentes cenários da otimização combinatória. POR REVER SE REALMENTE SE FAZ ISTO TUDO E MUDAR ?!?!?!??!

A. *Bubble Sort acrescentar informação sobre a família Exchange sorts*

O algoritmo *Bubble sort* consiste em comparar um dado item com todos os elementos de uma lista, em que o item vai se reposicionando com elementos comparativamente menores até encontrar um item maior ou chegar ao fim da lista. O nome do algoritmo prende-se com a forma como o elemento comparado vai ascendendo (ou afundando, consoante seja usado), tal como uma bolha na água.

Tipicamente este algoritmo não encontra aplicação em casos reais devido à sua ineficiência e complexidade de ordem $O(n^2)$. No entanto, a sua rápida implementação pode ser de interesse em casos particulares, em que listas que estejam próximas da sua forma ordenada passam pelo algoritmo com uma complexidade de ordem $O(n)$.

Algoritmo 1 Bubble Sort

Entrada: lista arr
Saída: lista ordenada arr

$n \leftarrow$ len(arr)
for i from 0 to $n-1$ **do**
  $swapped \leftarrow$ False
  for j from 0 to $n-2-i$ **do**
    if $arr[j] > arr[j+1]$ **then**
      Swap $arr[j]$ and $arr[j+1]$
      $swapped \leftarrow$ True
    end if
  end for
  if not $swapped$ **then**
    break
  end if
end for
return arr

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

Existem algoritmos baseados nesta abordagem, tal como o *Cocktail sort*, que aplica a metodologia em sentido crescente e decrescente, tornando-o mais rápido por norma, mas com o mesmo grau de complexidade no pior cenário.

B. *Selection Sort*

O algoritmo de *Selection sort* tem um procedimento bastante diferente do algoritmo anterior. Faz parte da família de algoritmos de *Selection sorts*, em que a lista de items ordenados vai sendo gerada por comparação a partir da lista original desordenada. No algoritmo em questão, procura-se o menor elemento disponível e coloca-se no início da nova lista. Estes passos repetem-se na lista desordenada até não existirem mais e a lista ordenada ser criada.

Algoritmo 2 Selection Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

$n \leftarrow \text{len}(arr)$
for i from 0 to $n-1$ **do**
  $min\_idx \leftarrow i$
  for j from $i+1$ to $n-1$ **do**
    if $arr[j] < arr[min\_idx]$ **then**
      $min\_idx \leftarrow j$
    end if
  end for
  if $min\_idx \neq i$ **then**
    Swap $arr[i]$ and $arr[min\_idx]$
  end if
end for
return arr

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

Dentro desta família de algoritmos, há um que devemos destacar, chamado *Gnome sort*, que consiste nas operações do *Selection sort*, mas sem ter o segundo ciclo dentro do primeiro, resultando num número de operações semelhante ao *Insertion sort*, e por consequência pior que o *Selection sort*.

C. Insertion Sort

Algoritmo 3 Insertion Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

for i from 1 to $\text{len}(arr) - 1$ **do**
  $key \leftarrow arr[i]$
  $j \leftarrow i - 1$
  while $j \geq 0$ and $arr[j] > key$ **do**
    $arr[j+1] \leftarrow arr[j]$
    $j \leftarrow j - 1$
  end while
  $arr[j+1] \leftarrow key$
end for
return arr

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

D. Counting Sort

Algoritmo 4 Counting Sort

Entrada: lista *arr*
Saída: lista ordenada *sorted_arr*

$max\_val \leftarrow \max(arr)$
$count \leftarrow$ array of zeros with length $max\_val + 1$
for each num in arr **do**
  $count[num] \leftarrow count[num] + 1$
end for
$sorted\_arr \leftarrow$ empty list
for i from 0 to max_val **do**
  if $count[i] > 0$ **then**
    Append $count[i]$ copies of i to $sorted\_arr$
  end if
end for
return $sorted\_arr$

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

E. Radix Sort

Algoritmo 5 Radix Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

function COUNTINGSORTRADIX(arr, exp)
  $n \leftarrow \text{len}(arr)$
  $output \leftarrow$ array of zeros of length n
  $count \leftarrow$ array of zeros of length 10
  for i from 0 to $n-1$ **do**
    $index \leftarrow (arr[i] \div exp) \bmod 10$
    $count[index] \leftarrow count[index] + 1$
  end for
  for i from 1 to 9 **do**
    $count[i] \leftarrow count[i] + count[i-1]$
  end for
  for i from $n-1$ to 0 **do**
    $index \leftarrow (arr[i] \div exp) \bmod 10$
    $output[count[index] - 1] \leftarrow arr[i]$
    $count[index] \leftarrow count[index] - 1$
  end for
  for i from 0 to $n-1$ **do**
    $arr[i] \leftarrow output[i]$
  end for
end function
$max\_num \leftarrow \max(arr)$
$exp \leftarrow 1$
while $max\_num \div exp > 0$ **do**
  COUNTINGSORTRADIX(arr, exp)
  $exp \leftarrow exp \times 10$
end while
return arr

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

F. Quick Sort

Algoritmo 6 Quick Sort

Entrada: lista *arr*

Saída: lista ordenada *arr*

```
function PARTITION(items, low, high)
    pivot ← items[high]
    i ← low − 1
    for j from low to high − 1 do
        if items[j] ≤ pivot then
            i ← i + 1
            Swap items[i] and items[j]
        end if
    end for
    Swap items[i + 1] and items[high]
    return i + 1
end function
function QUICKSORT(items, low, high)
    if low < high then
        pivot_index ← PARTITION(items, low, high)
        QUICKSORT(items, low, pivot_index − 1)
        QUICKSORT(items, pivot_index + 1, high)
    end if
end function
QUICKSORT(arr, 0, len(arr) − 1)
return arr
```

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

G. Merge Sort

Algoritmo 7 Merge Sort

Entrada: lista *arr*

Saída: lista ordenada *arr*

```
function MERGESORT(arr)
    if len(arr) > 1 then
        mid ← len(arr) ÷ 2
        left ← arr[0 : mid]
        right ← arr[mid :]
        MERGESORT(left)
        MERGESORT(right)
        i, j, k ← 0, 0, 0
        while i < len(left) and j < len(right) do
            if left[i] ≤ right[j] then
                arr[k] ← left[i]
                i ← i + 1
            else
                arr[k] ← right[j]
                j ← j + 1
            end if
            k ← k + 1
        end while
        while i < len(left) do
            arr[k] ← left[i]
            i ← i + 1
            k ← k + 1
        end while
        while j < len(right) do
            arr[k] ← right[j]
            j ← j + 1
            k ← k + 1
        end while
    end if
end function
MERGESORT(arr)
return arr
```

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

H. Heap Sort

Algoritmo 8 Heap Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

```
function HEAPIFY(arr, n, i)
    largest ← i
    left ← 2i + 1
    right ← 2i + 2
    if left < n and arr[left] > arr[largest] then
        largest ← left
    end if
    if right < n and arr[right] > arr[largest] then
        largest ← right
    end if
    if largest ≠ i then
        Swap arr[i] and arr[largest]
        HEAPIFY(arr, n, largest)
    end if
end function
function HEAPSORT(arr)
    n ← len(arr)
    for i from ⌊n/2⌋ − 1 down to 0 do
        HEAPIFY(arr, n, i)
    end for
    for i from n − 1 down to 1 do
        Swap arr[0] and arr[i]
        HEAPIFY(arr, i, 0)
    end for
    return arr
end function
HEAPSORT(arr)
```

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

I. Timsort

Algoritmo 9 Timsort

Entrada: lista *arr*
Saída: lista ordenada *arr*

```
function INSERTIONSORT(arr, left, right)
    for i from left + 1 to right do
        key ← arr[i]
        j ← i − 1
        while j ≥ left and arr[j] > key do
            arr[j + 1] ← arr[j]
            j ← j − 1
        end while
        arr[j + 1] ← key
    end for
end function
function MERGE(arr, left, mid, right)
    left_part ← arr[left : mid + 1]
    right_part ← arr[mid + 1 : right + 1]
    i, j, k ← 0, 0, left
    while i < len(left_part) and j < len(right_part) do
        if left_part[i] ≤ right_part[j] then
            arr[k] ← left_part[i]
            i ← i + 1
        else
            arr[k] ← right_part[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
    while i < len(left_part) do
        arr[k] ← left_part[i]
        i, k ← i + 1, k + 1
    end while
    while j < len(right_part) do
        arr[k] ← right_part[j]
        j, k ← j + 1, k + 1
    end while
end function
function TIMSORT(arr)
    MIN_RUN ← 32
    n ← len(arr)
    for start from 0 to n − 1 in steps of MIN_RUN do
        INSERTIONSORT(arr, start, min(start + MIN_RUN − 1, n − 1))
    end for
    size ← MIN_RUN
    while size < n do
        for left from 0 to n − 1 in steps of 2 × size do
            mid ← min(n − 1, left + size − 1)
            right ← min(n − 1, left + 2 × size − 1)
            if mid < right then
                MERGE(arr, left, mid, right)
            end if
        end for
        size ← size × 2
    end while
    return arr
end function
TIMSORT(arr)
```

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO

III. Simulação de Ordenações

ns q usar python para simluar os algs e contar ops basicas

Tabela I: comparao das complexidades teoricas CONFIRMAR where k is range of input values and n is number of elemenets to sort

Algoritmo	**Complexidade**
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Counting Sort	$O(n+k)$
Radix Sort	$O(nk)$
Quick Sort	$O(n \log n)$*
Merge Sort	$O(n \log n)$
Heap Sort	$O(n \log n)$
Timsort	$O(n \log n)$

Referências