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*—cona cona caon**

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 ← len(arr)
for i from 0 to n − 1 do
    swapped ← 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 ← 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*

```
1: n ← len(arr)
2: for i from 0 to n − 1 do
3:     min_idx ← i
4:     for j from i + 1 to n − 1 do
5:         if arr[j] < arr[min_idx] then
6:             min_idx ← j
7:         end if
8:     end for
9:     if min_idx ≠ i then
10:        Swap arr[i] and arr[min_idx]
11:    end if
12: end for
13: 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*

```
1: for i from 1 to len(arr) − 1 do
2:     key ← arr[i]
3:     j ← i − 1
4:     while j ≥ 0 and arr[j] > key do
5:         arr[j + 1] ← arr[j]
6:         j ← j − 1
7:     end while
8:     arr[j + 1] ← key
9: end for
10: 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*

```
1: max_val ← max(arr)
2: count ← array of zeros with length max_val + 1
3: for each num in arr do
4:     count[num] ← count[num] + 1
5: end for
6: sorted_arr ← empty list
7: for i from 0 to max_val do
8:     if count[i] > 0 then
9:         Append count[i] copies of i to sorted_arr
10:    end if
11: end for
12: 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*

```
1: function CountingSortRadix(arr, exp)
2:     n ← len(arr)
3:     output ← array of zeros of length n
4:     count ← array of zeros of length 10
5:     for i from 0 to n − 1 do
6:         index ← (arr[i] ÷ exp) mod 10
7:         count[index] ← count[index] + 1
8:     end for
9:     for i from 1 to 9 do
10:        count[i] ← count[i] + count[i − 1]
11:    end for
12:    for i from n − 1 to 0 do
13:        index ← (arr[i] ÷ exp) mod 10
14:        output[count[index] − 1] ← arr[i]
15:        count[index] ← count[index] − 1
16:    end for
17:    for i from 0 to n − 1 do
18:        arr[i] ← output[i]
19:    end for
20: end function
21: max_num ← max(arr)
22: exp ← 1
23: while max_num ÷ exp > 0 do
24:    CountingSortRadix(arr, exp)
25:    exp ← exp × 10
26: end while
27: 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

```
1: function PARTITION(items, low, high)
2:     pivot ← items[high]
3:     i ← low − 1
4:     for j from low to high − 1 do
5:         if items[j] ≤ pivot then
6:             i ← i + 1
7:             Swap items[i] and items[j]
8:         end if
9:     end for
10:    Swap items[i + 1] and items[high]
11:    return i + 1
12: end function
13: function QUICKSORT(items, low, high)
14:    if low < high then
15:        pivot_index ← PARTITION(items, low, high)
16:        QUICKSORT(items, low, pivot_index − 1)
17:        QUICKSORT(items, pivot_index + 1, high)
18:    end if
19: end function
20: QUICKSORT(arr, 0, len(arr) − 1)
21: 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

```
1: function MERGESORT(arr)
2:     if len(arr) > 1 then
3:         mid ← len(arr) ÷ 2
4:         left ← arr[0 : mid]
5:         right ← arr[mid :]
6:         MERGESORT(left)
7:         MERGESORT(right)
8:         i, j, k ← 0, 0, 0
9:         while i < len(left) and j < len(right) do
10:            if left[i] ≤ right[j] then
11:                arr[k] ← left[i]
12:                i ← i + 1
13:            else
14:                arr[k] ← right[j]
15:                j ← j + 1
16:            end if
17:            k ← k + 1
18:        end while
19:        while i < len(left) do
20:            arr[k] ← left[i]
21:            i ← i + 1
22:            k ← k + 1
23:        end while
24:        while j < len(right) do
25:            arr[k] ← right[j]
26:            j ← j + 1
27:            k ← k + 1
28:        end while
29:    end if
30: end function
31: MERGESORT(arr)
32: 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

```
1: function HEAPIFY(arr, n, i)
2:     largest ← i
3:     left ← 2i + 1
4:     right ← 2i + 2
5:     if left < n and arr[left] > arr[largest] then
6:         largest ← left
7:     end if
8:     if right < n and arr[right] > arr[largest] then
9:         largest ← right
10:    end if
11:    if largest ≠ i then
12:        Swap arr[i] and arr[largest]
13:        HEAPIFY(arr, n, largest)
14:    end if
15: end function
16: function HEAPSORT(arr)
17:    n ← len(arr)
18:    for i from ⌊n/2⌋ − 1 down to 0 do
19:        HEAPIFY(arr, n, i)
20:    end for
21:    for i from n − 1 down to 1 do
22:        Swap arr[0] and arr[i]
23:        HEAPIFY(arr, i, 0)
24:    end for
25:    return arr
26: end function
27: 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

```
1: function INSERTIONSORT(arr, left, right)
2:     for i from left + 1 to right do
3:         key ← arr[i]
4:         j ← i − 1
5:         while j ≥ left and arr[j] > key do
6:             arr[j + 1] ← arr[j]
7:             j ← j − 1
8:         end while
9:         arr[j + 1] ← key
10:    end for
11: end function
12: function MERGE(arr, left, mid, right)
13:    left_part ← arr[left : mid + 1]
14:    right_part ← arr[mid + 1 : right + 1]
15:    i, j, k ← 0, 0, left
16:    while i < len(left_part) and j < len(right_part)
       do
17:        if left_part[i] ≤ right_part[j] then
18:            arr[k] ← left_part[i]
19:            i ← i + 1
20:        else
21:            arr[k] ← right_part[j]
22:            j ← j + 1
23:        end if
24:        k ← k + 1
25:    end while
26:    while i < len(left_part) do
27:        arr[k] ← left_part[i]
28:        i, k ← i + 1, k + 1
29:    end while
30:    while j < len(right_part) do
31:        arr[k] ← right_part[j]
32:        j, k ← j + 1, k + 1
33:    end while
34: end function
35: function TIMSORT(arr)
36:    MIN_RUN ← 32
37:    n ← len(arr)
38:    for start from 0 to n − 1 in steps of MIN_RUN
       do
39:        INSERTIONSORT(arr, start,
       min(start + MIN_RUN − 1, n − 1))
40:    end for
41:    size ← MIN_RUN
42:    while size < n do
43:        for left from 0 to n − 1 in steps of 2 × size do
44:            mid ← min(n − 1, left + size − 1)
45:            right ← min(n − 1, left + 2 × size − 1)
46:            if mid < right then
47:                MERGE(arr, left, mid, right)
48:            end if
49:        end for
50:        size ← size × 2
51:    end while
52:    return arr
53: end function
54: 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	$O(n \log n)$

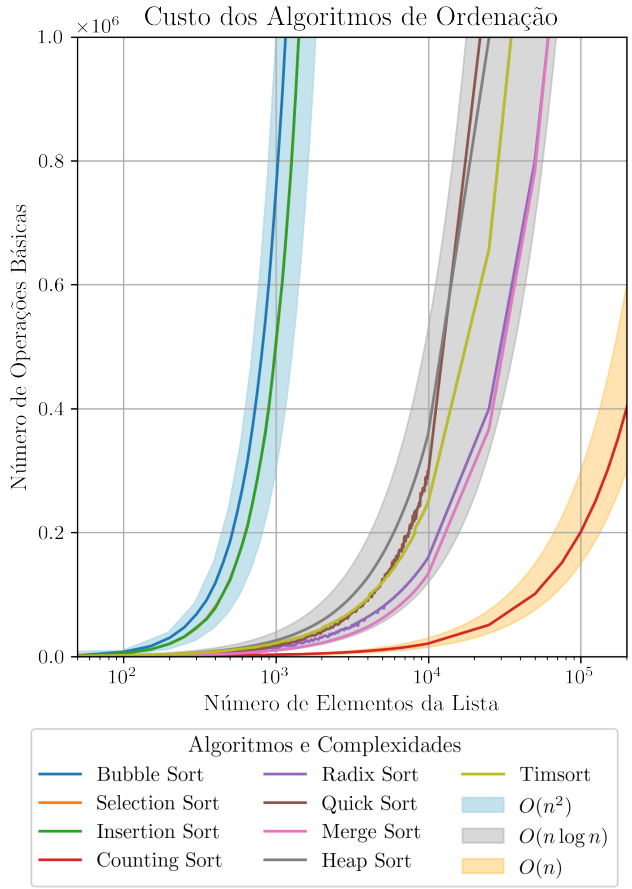

Figura 1: complxidades baseadas no num de ops basicas em listas geradas com numeros inteiros positivos aleatorios até 1000

Referências