Ordenação por Permutação - Assignment 1

Diogo Araujo Miranda[1]

[1]ICEI - Pontifícia Universidade Católica de Minas Gerais (PUC-MG)

1. Contexto

A complexidade de algoritmos é uma área muito discutida na computação, sendo um ponto muito importante quando criamos e analisamos um algoritmo. É comum surgirem novos métodos e novas abordagens para resolver um problema, levantando perguntas de quão boa essa solução pode ser. Levando em consideração o problema de ordenação de dados, temos na literatura vários métodos já discutidos como o QuickSort, InsertionSort, SelectionSort, entre outros. Nesse documento, será feita uma análise de um método de ordenação por permutação. Esse método é baseado estratégia de permutar elementos de um conjunto de dados até encontrar um subconjunto que contenha todos os elementos ordenados em forma não-decrescente. Todo o código desse método, bem como dos outros discutidos e comparados, podem ser encontrados no repositório: https://github.com/Diogo-Miranda/5-periodo/tree/main/design-and-analysis-of-algorithms/permutation-sort-analysis

2. Estratégias utilizadas

Ao analisar o problema, inicialmente foi utilizado um método de ordenação baseado em permutações. A primeira estratégia diz a respeito de uma função que utiliza um número aleatório - baseado em seed - para escolher os elementos que irão ser permutados.

A segunda abordagem é utilizando um algoritmo de Backtracking para encontrar as possíveis permutações, até encontrar um subconjunto ordenado a partir do conjunto inicial.

2.1. Permutação Aleatória

A permutação baseada em elementos aleatórios segue a ideia de percorrer o conjunto original O(n) vezes, e partir de um índice i, realizar a troca com outro elemento de índice aleatório(n). Essa estratégia pode ser descrita pelo seguinte algoritmo escrito em C++:

```
void sort(int a[], int n) {
    for (int i = 0; i < n; i++) {
        swap(a, i, rand()%n);
    }

    printArr(a, n);
}

void permutationSort(int a[], int n) {
    while(!isSorted(a, n)) {
        sort(a, n);
    }
}
```

A complexidade essa solução é baseada em elementos gerados aleatoriamente, o que deixa a sua análise de complexidade em um caso genérico dependendo de um número gerado aleatoriamente a cada passada pelo vetor. Na primeira função que será chamada, temos uma verificação enquanto o vetor não estiver ordenado (não tivermos achado a permutação que nos dará a solução). Essa função é descrita por:

```
bool isSorted(int a[], int n) {
    bool resp = true;

    while ( --n >= 1 ) {
        if (a[n] < a[n-1]) {
            resp = false;
        }
    }

    return resp;
}
```

A complexidade teórica dessa função, levando em consideração o número de comparações, é O(n), onde n é o tamanho do conjunto.

Sendo assim, a função de ordenação sort() será chamada enquanto não encontrarmos um conjunto que será permutado n! vezes. Sua complexidade em um caso médio será O(n * n!). Essa solução não nos da um algoritmo ótimo para resolução do problema, já que dependendo do tamanho do conjunto teremos um tempo de execução muito elevado, podendo chegar a um limite superior no pior caso de O(inifinity). Em seu melhor caso, seu número de comparação será O(n), com um conjunto já ordenado.

2.2. Permutação por Permutação utilizando Backtracking

Nessa estratégia, foi utilizado um algoritmo de backtracking, que tem como sua premissa utilizar chamadas recursivas. Essas chamadas recursivas são necessárias para gerar uma árvore de recursão com a permutação dos elementos, descrita, por exemplo:

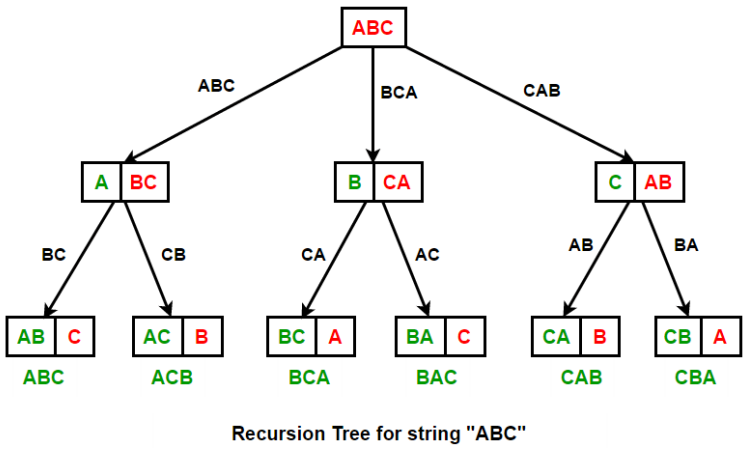

Figure 1. Exemplo Backtracking

10Durante a geração dessa árvore de recursão, é verificado se o subconjunto encontrado é o conjunto solução, utilizando a mesma função isSorted(). Segue o código em C++:

```
int* permutationSort(int a[], int l, int n) {
    if(l == n) {
        return a;
    } else {
        for (int i = l; i <= n; i++) {
            // swap initial
            swap(a, l, i);
            // call next
            a = permutationSort(a, l+1, n);
            // stop recursion if found sort set
            if (isSorted(a, n+1)) {
                return a;
            }
            // backtrack
            swap(a, l, i);
        }
    }

   return a;
}
```

A complexidade dessa solução, no seu caso médio, é O($n^2 * n!$).

Temos a cada chamada 1 comparação inicial n! vezes, e dentro do laço de repetição iremos verificar se o vetor está ordenado com O(n), n vezes. Esse algoritmo também não nos dá uma solução ótima, já que seu pior caso é O(infinity).

2.3. Outros métodos

Outros métodos de ordenação utilizados foram o QuickSort, com a estratégia de dividir e conquistar, que como já conhecido possuí ordem de complexidade no seu caso médio de O(nlogn) e no seu pior caso O(n^2).

```
void quicksort(int a[], int esq, int dir) {
    int i = esq, j = dir;
    int pivo = a[(dir+esq)/2];
    while (i <= j) {
        while (a[i] < pivo) i++;
        while (a[j] > pivo) j--;
        if (i <= j) {
            swap(a, i, j);
            i++;
            j--;
        }
    }

    if (esq < j) quicksort(a, esq, j);
```

```
15      if (i < dir) quicksort(a, i, dir);
16  }
```

SelectionSort, que utiliza dois laços de repetição em sua implementação e tem ordem de complexidade no pior, caso médio e melhor caso de O(n^2).

```
1  void selectionSort(int a[], int n) {
2      for(int i = 0; i < n; i++) {
3          int min = i;
4          for (int j = i + 1; j < n; j++) {
5              if (a[j] < a[min]) {
6                  min = j;
7              }
8          }
9          if(i != min) {
10             swap(a, i, min);
11         }
12     }
13 }
```

InsertionSort, utiliza também dois laços de repetição em um conjunto de dados de tamanho n, possuindo complexidade no pior caso e no caso médio de O(n^2).

```
1  void insertionSort(int a[], int n) {
2      for(int j = 2; j < n; j++) {
3          int k = a[j];
4          int i = j - 1;
5          while(a[i] > k && i > 0) {
6              a[i + 1] = a[i];
7              i = i - 1;
8          }
9          a[i + 1] = k;
10     }
11 }
```

3. Análise de complexidade

Com a análise teórica dos algoritmos acima, temos a seguinte tabela com sua ordem de complexidade:

Algoritmo	Melhor caso	Caso Médio	Pior Caso
Permut. Sort Random	O(n)	O(n*n!)	O(infinity)
Permut. Sort Random Backtracking	O(n*n!)	O($n^2 * n!$)	O(infinity)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Quick Sort	O(nlogn)	O(nlog)	O(n^2)

A análise experimental dos algoritmos foi feita utilizando o seu tempo de execução (em segundos), com instâncias variando de n = 0 até n = 15. Esse número pequeno foi utilizado devido ao gasto exponencial de memória e processamento dos algoritmos de ordenação por permutação. A máquina utilizada para os testes possuí proces-

sador de 8 núcleos com arquitetura ARM e 16GB de memória, rodando diretamente em tecnologia de disco rígido SSD:

Algoritmo	$0 \leq n \leq 10$	n = 11	n = 12	n = 13	n = 14	n = 15
PS Random	$\leq 1s$	15s	1min	$\geq 5min$	∞	∞
PS Backtracking	$\leq 1s$	2s	35s	54s	3,4min	∞
Selection Sort	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$
Insertion Sort	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$
Quick Sort	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$	$\leq 1s$	¡ 1s

Podemos assim concluir que, os algoritmos de ordenação por permutação não são algoritmos que nos dão soluções ótimos para quaisquer instâncias e também são algoritmos que gastam bastante memória para gerar todas permutações e perdem frente as outras implementações contidas na literatura.