Counting Sort

Arthur G. M. Santos[1]

[1]Departamento de Ciência da Computação
Universidade Federal de São João del Rei (UFSJ) – São João del Rei, MG – Brasil

arthurgabrielbd@gmail.com

Resumo. *Neste artigo apresentamos um novo algoritmo de ordenação, o Counting Sort, que utiliza uma nova abordagem ao problema de ordenação em que é feito uma ordenação com base na previsão da posição que um determinado valor deve ocupar na sequência ordenada a partir da quantidade de valores dessa sequência em que ele é maior. O algoritmo apresenta uma melhora considerável no tempo de execução em relação aos algoritmos de ordenação do estado da arte.*

1. Introdução

Quanto mais a tecnologia avança, maior é a quantidade de dados a serem processados. Isso ocorre por conta do aumento no número de pessoas e empresas que geram e armazenam dados diariamente. Em tarefas de ordenação isso não é diferente. Diversas aplicações para indexação web, mineração de dados, sistemas geográficos e bancos de dados utilizam ferramentas de ordenação, e por isso, necessitam de algoritmos rápidos e eficientes na ordenação de grandes sequências de dados.

Os atuais algoritmos de ordenação do estado da arte, como o *Bubble Sort*, *Selection Sort* e o *Insertion Sort*, apresentam uma ineficiência em tempo de execução de acordo com o aumento do tamanho das sequências a serem ordenadas. Essa ineficiência é causada por conta desses algoritmos utilizarem a comparação e troca de posição entre os valores de uma sequência como abordagem principal.

O algoritmo aqui desenvolvido, o *Counting Sort*, apresenta uma nova abordagem ao problema de ordenação, em que é feito uma ordenação com base na previsão da posição que um determinado valor deve ocupar na sequência ordenada. Dessa forma, o *Counting Sort* consegue ser computacionalmente menos custoso em relação aos demais algoritmos quando comparado a variação no aumento do tamanho das sequências de dados. Além disso, ele não abusa do armazenamento dos dados na memória.

Por conta do *Counting Sort* não ser um algoritmo baseado em ordenação por comparação, ele não possui limite inferior de $\Omega(n\ log\ n)$ para complexidade em tempo de execução de acordo com [**Cormen**].

2. Trabalhos Relacionados

Ordenação é uma operação fundamental na ciência da computação, sendo aplicada diretamente ou indiretamente (como passo intermediário) em diversas soluções. O desafio deste problema é encontrar um algoritmo de ordenação que melhor se ajusta a uma dada aplicação. Essa escolha depende de alguns fatores como, a quantidade de chaves que serão ordenadas, se os itens já estão parcialmente ordenados, possíveis restrições nos valores das chaves e o tipo de dispositivo de armazenamento que está sendo utilizado.

Os algoritmos propostos na literatura para resolver o problema de ordenação, como o *Bubble Sort* [**bubble**], o *Selection Sort* [**selection**] e o *Insertion Sort* [**insertion**], apresentam uma abordagem de ordenação por meio da comparação e troca de valores em um vetor. O *Bubble Sort* ordena com base no maior elemento que ele encontra durante leituras consecutivas feitas na sequência de valores, enquanto o *Selection Sort* ordena com base no menor elemento. O *Insertion Sort* é melhor aplicado em sequências de valores pequenas, pré-ordenadas ou quando se quer inserir valores em uma sequência já ordenada, por conta disso ele é computacionalmente custoso para grandes sequências e sem nenhum tipo de pré-ordenação das chaves. De acordo com o aumento do valor de n (tamanho das sequências a serem ordenadas), os algoritmos de comparação tendem a ser mais custosos, e por conta disso, no pior cenário de execução esses algoritmos irão comparar cada valor da sequência a_i com os $n-1$ outros valores da sequência até a sequência estar totalmente ordenada.

O algoritmo proposto aqui, o *Counting Sort*, não é um algoritmo baseado em comparação de valores, e por conta disso, não possui limite inferior de $\Omega(n\,log\,n)$ definido por [**Cormen**]. Dessa forma, de acordo com o aumento do valor de n, o algoritmo tende a ser computacionalmente menos custoso, já que no seu pior cenário de execução ele lê a sequência de valores duas vezes.

3. Metodologia

Counting Sort é um algoritmo de ordenação que trabalha na hipótese de que se um elemento x é maior que outro elemento x_s do vetor de entrada, então x deve, obrigatoriamente, estar em uma posição maior que x_s no vetor de saída (vetor ordenado). Como exemplo, e assumindo que o vetor começa na posição zero, se o valor x for maior que outros 5 valores do vetor de entrada, então x pertence a posição 5 no vetor de saída.

Para apresentar o algoritmo, utilizamos as nomenclaturas em que n é o tamanho do vetor de entrada, k é o maior valor que uma chave pode ter, $A[0..n-1]$ é o vetor de entrada, $B[0..n-1]$ é o vetor de saída (ordenado) e $C[0..k-1]$ é um vetor auxiliar. A execução do *Counting Sort* é dividida em quatro etapas bem definidas:

- **Etapa 1**: Primeiro iniciamos o vetor C com tamanho igual a k contendo o valor zero em todas as posições.
- **Etapa 2**: Após a fase 1, o algoritmo realiza uma leitura de cada valor x do vetor de entrada A. Para cada valor de x, o vetor C incrementa o valor 1 (um) no índice de x. Dessa forma, ao final desta fase, temos que o vetor C é a contagem dos valores que existem em A.
- **Etapa 3**: Uma leitura é feita no vetor C, onde cada valor de C no índice i, sendo $i = 1$ até k, tem seu valor incrementado de acordo com o valor que está no índice $i-1$ do vetor C.
- **Etapa 4**: Por fim, o vetor B recebe os valores de forma ordenada do vetor A. Para cada valor em $A[j]$, sendo $j = n-1$ até 0, o valor $C[A[j]]-1$ é a posição final correta do valor $A[j]$ para o vetor ordenado B. Pelo fato da possibilidade de termos chaves duplicadas, é necessário ainda decrementar o valor de $C[A[j]]$ toda vez que adicionamos um valor ao vetor B.

A figura 1 apresenta um exemplo da execução do algoritmo *Counting Sort*. O exemplo dado na figura 1 passou pela execução das três primeiras etapas descritas acima,

sendo assim, o vetor C corresponde aos valores do final da execução da terceira etapa. A posição do valor 3 no vetor B da figura 1 é sua posição final em relação ao vetor ordenado, e isso ocorre por conta da execução da primeira iteração da quarta etapa. É acessado no vetor C a posição referente ao valor 3 com $C[3]$, o valor contido em C nessa posição acessada é decrementada em um, já que o vetor B inicia a indexação com zero, e é usado como posição final para o valor de A, dessa forma $B[6]$ é a posição final do valor 3 no vetor ordenado. Esse procedimento é realizado para todos os outros valores do vetor A.

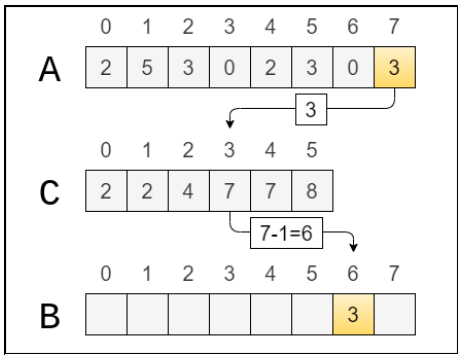


Figura 1. Exemplo de execução do algoritmo. O vetor A é o vetor de entrada, o vetor C é um vetor auxiliar e o vetor B é o vetor de saída ordenado.

A fim de analisar profundamente o algoritmo, é necessário descrever detalhadamente seu comportamento. Identificamos que o algoritmo *Counting Sort* executa mais instruções de acordo com o aumento dos valores de n e k, que estão diretamente ligados ao tamanho dos vetores A, B e C. A primeira etapa executa uma leitura no vetor C de tamanho k, a segunda etapa executa uma leitura no vetor A de tamanho n, a terceira etapa executa uma leitura em C de tamanho k e, por fim, a quarta etapa executa uma leitura em A de tamanho n. Como tanto n quanto k influenciam na quantidade de instruções a serem executadas em uma mesma ordem de grandeza, e portanto gerando um maior tempo de processamento, a complexidade de tempo do algoritmo *Counting Sort* é $O(k + n)$.

O algoritmo é estável, já que copia objetos com as mesmas chaves da direita para a esquerda no vetor de saída. Além disso, possui suporte a chaves duplicadas. O algoritmo não oferece suporte a números contínuos e, é necessário ter o valor de k bem definido.

4. Resultados

A principal proposta desta seção é mostrar que o nosso algoritmo desenvolvido performa melhor que os atuais algoritmos de ordenação do estado da arte com diferentes conjuntos de sequências.

4.1. Configurações Iniciais

O algoritmo e os arquivos complementares foram escritos na linguagem *Python*. Adicionalmente, foi utilizado a biblioteca de *Python* chamada de *numpy* para gerar as sequências de chaves aleatórias utilizando valores de semente. Valores de semente são utilizados para criar sequências pré-definidas de números aleatórios.

Primeiro vamos apresentar os parâmetros de cada conjunto de sequência que foram utilizadas pelos testes do algoritmo *Counting Sort*. A fim de verificar o comportamento do algoritmo em diferentes cenários, foi escolhido um conjunto de variáveis para

experimentação. As variáveis e seus conjuntos de parâmetros de valores, estão apresentados na Tabela 1.

Tamanho do conjunto	Tamanho de k
100	100
200	200
400	400
800	800
1600	1600

Tabela 1. Tabela com as variáveis e os valores utilizados para cada sequência de entrada experimentada.

O tamanho do conjunto é definido como o tamanho das sequências a serem ordenadas. O tamanho de k é referente a quantidade de valores que podem representar uma chave. Por fim, o valor da semente é o valor utilizado para gerar sequência de valores iguais, permitindo a reprodução dos experimentos.

Utilizamos uma combinação fatorial completa entre todos os valores de parâmetros mostrados na tabela 1. Então, para cada combinação é gerado uma sequência de dados única, e cada sequência foi armazenada em um arquivo referente a ela.

Os testes foram realizado em um *CPU Intel 560M* com 2.67GHz e 4 GB de memória *RAM*.

4.2. Análise do tempo de execução

Nesta seção é apresentado os resultados obtidos pela execução do algoritmo *Counting Sort* tendo como entrada sequências de valores definidos aleatoriamente por um valor de semente. O objetivo é analisar o comportamento do tempo de execução do algoritmo em relação ao crescimento do valor da variável de tamanho do conjunto.

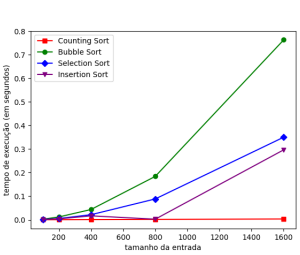
A fim de variar apenas o valor do tamanho do conjunto para verificar seu comportamento, foi calculado a média dos tempos de execução obtidos para cada tamanho de entrada em relação ao tamanho de k e o valor da semente. A Figura 2 apresenta os gráficos referente a execução do algoritmo em diferentes entradas de dados.

Utilizamos como comparativo ao *Counting Sort* a execução dos algoritmos *Bubble Sort*, *Selection Sort* e *Insertion Sort*. Para entrada de dados ordenados de maneira crescente os algoritmos *Bubble Sort* e *Insertion Sort* apresentam complexidade de tempo de $O(n)$, por conta dessa configuração representar seus melhores cenários de execução, enquanto o *Counting Sort* está com tempo de execução pouco pior que esses algoritmos. Para entradas de dados aleatórios e de forma decrescente o algoritmo *Counting Sort* se mantêm com tempo de execução menor em relação aos demais algoritmos comparados.

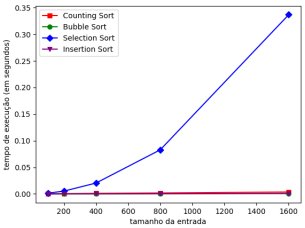
4.3. Análise do consumo de memória

Nesta seção é apresentado o consumo de memoria do algoritmo durante os testes realizados de sua execução na linguagem *Python*. Para retornar a quantidade de memória usada pelo *Counting Sort* foi utilizado a biblioteca $memory\_profiler$ da linguagem *Python*.

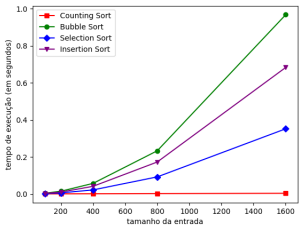
O primeiro experimento desta seção busca relacionar o aumento do valor do tamanho das sequências com aumento da memória alocada pelo algoritmo *Counting Sort*,



(a) Entrada aleatória



(b) Entrada ordenada de forma crescente



(c) Entrada ordenada de forma decrescente

Figura 2. Gráficos com o resultado da execução dos testes realizados nos algoritmos de ordenação.

utilizando um tamanho fixo de $k = 10^1$ e $semente = 10$. A tabela 2 apresenta esses resultados.

Tamanho do conjunto	Memória Alocada
10^2	33.3 MiB
10^3	33.4 MiB
10^4	33.8 MiB
10^5	35.8 MiB
10^6	49.7 MiB

Tabela 2. Tabela com a variação do tamanho da entrada e memória alocada durante a execução do algoritmo *Counting Sort*.

A tabela 2 mostra que com o aumento do tamanho do conjunto houve também o aumento da memória utilizada pelo algoritmo. A correlação criada faz sentido, já que é necessário alocar na memória um espaço com o tamanho da sequência de dados. Além disso, para executar a ordenação, o algoritmo aloca um espaço de tamanho igual a quantidade de valores que podem representar uma chave, para que seja possível efetuar a reordenação a partir do cálculo de índices.

Portanto, o algoritmo *Counting Sort* apresenta um aumento da sua complexidade de espaço $O(n + k)$, onde n é o tamanho do conjunto e k é a quantidade de valores que podem representar uma chave.

4.4. Comparação com algoritmos estado da arte

O algoritmo de ordenação *Counting Sort*, mesmo em seu pior cenário de execução, apresenta uma complexidade de $O(k+n)$ tanto em tempo quanto em espaço, enquanto outros algoritmos da literatura como *Bubble Sort, Selection Sort* e *Insertion Sort*, apresentam complexidade de $O(n^2)$ para seus piores cenários de execução.

5. Conclusão

Neste artigo foi apresentado o *Counting Sort*, que apresenta uma nova abordagem ao problema de ordenação, utilizando da estratégia de calcular os índices da memória para prever a posição de todos os valores na sequência ordenada. Como resultado o algoritmo tem uma melhoria na complexidade de tempo, apresentando em seu pior caso $O(k + n)$,

que é melhor quando comparado aos algoritmos *Bubble Sort, Selection Sort* e *Insertion Sort*.

Apesar de apresentar uma melhoria em complexidade de tempo, o algoritmo apresenta algumas limitações, como por exemplo, não oferecer suporte a números contínuos, já que os cálculos de indexação de vetores são limitados a números inteiros. Além disso, a complexidade do algoritmo é afetada pelo aumento do tamanho k da representação de valores que uma posição da sequência de dados pode ter, sendo uma variável que não afeta outros algoritmos de ordenação.

Por fim, o algoritmo apresenta suporte a chaves com valores duplicados e é estável em todas as suas execuções, independente da implementação feita. Para trabalhos futuros, pretendemos aprimorar o algoritmo que utiliza do multiprocessamento e paralelização dos dados, utilizando arquiteturas com múltiplos núcleos e memória compartilhada.