UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Algoritmos e Estruturas de Dados III
Trabalho Prático 0 – Operações de Nubby

Orlando Enrico Liz Silvério Silva

Belo Horizonte
7 de setembro de 2017

1 Introdução

Muitos problemas podem ser resolvidos na área da computação de maneiras distintas. Por isso, a assimilação de conhecimentos, capacidade de abstração e compreensão de paradigmas contribui fundamentalmente para se obter uma solução otimizada de um determinado problema.

Sendo assim, este trabalho tem como objetivo avaliar as soluções propostas para o seguinte problema: dado um vetor de N elementos alocado sequencialmente, é necessário executar M operações. Cada uma das M linhas é composta por uma operação, e dois inteiros (i < j) que configuram o intervalo de interesse. Essas operações podem ser: adicionar ou subtrair 1 em cada elemento dentro do intervalo [i, j] (Add e Sub) e consultar o máximo, mínimo ou a soma dos elementos contidos no intervalo [i, j] (Max, Min, Sum). Para a resolução desse problema, são apresentadas abordagens diferentes: uma envolvendo matriz e outra envolvendo Árvore de Segmentos.

2 Solução do Problema

Esta seção tem como objetivo apresentar a solução implementada. Para tanto, inicialmente será introduzida a modelagem proposta, ilustrando as etapas da resolução do problema. Em seguida, serão apresentados os dois algoritmos com as diferentes estruturas de dados utilizados para a sua resolução: a matriz e a Árvore de Segmentos. Por fim, a estrutura de código utilizada é apresentada.

2.1 Modelagem do Problema

Primeiramente, é importante verificar a entrada do problema. Os inteiros N (número de elementos) e M (número de operações) devem estar dentro do limite estabelecido pelo enunciado ($1 \leq N \leq 10^6$) e ($1 \leq M \leq 10^7$). Além disso, as operações permitidas consistem, basicamente, em consultas (Min, Max e Sum) e em alteração de valores (Add e Sub) em um determinado intervalo [i, j].

Na primeira solução, é necessário criar uma estrutura T (tupla) que armazena os valores de mínimo, máximo e soma. A partir de um vetor v[1..N] basta computar o Min(i, j), Max(i, j) e Sum(i, j) para todos os possíveis intervalos e mapeá-los em uma matriz M = $a_{i,j} \in T_{mxn}$. A diagonal principal dessa matriz é composta pelos elementos do vetor usado como referência para a sua construção. Uma vez feito o processamento, no caso de busca em um intervalo [i, j], basta acessar o elemento $a_{i,j}$ da matriz. Já em uma operação que envolve alteração nos valores dos números de um intervalo é necessário modificar o vetor v e computar a matriz novamente.

Já a segunda solução envolve uma estrutura de dados semelhante a árvore binária, chamada de Árvore de Segmentos, cuja ideia central é segmentar o vetor em intervalos e os representar como nós de uma árvore. O nó raiz sempre será o intervalo correspondente a [1..N]. Fragmentando esse intervalo no meio, obtem-se nós filhos da esquerda e direita e assim sucessivamente até chegar aos nós folha, que respresentam os elementos do vetor v. Ou seja, nós que não são folhas correspondem a união dos resultados dos nós filhos. Essa árvore é representada através de um vetor cujas posições 2n representam nós da esquerda e posições 2n+1 representam nós da direita para n>1. Para operações de consulta, uma busca na árvore é realizada a partir do nó raiz em direção aos filhos da esquerda enquanto seus intervalos estiverem contidos no intervalo de interesse e em direção aos filhos da direita para encontrar o que se quer, caso seja necessário complementar um intervalo. Já as operações que alteram valores demandam que o(s) nó(s) contido(s) no intervalo em questão seja atualizados e os que o antecedem.

Com esse cenário, o ponto chave da questão reside em comparar os dois métodos propostos para solucionar o problema e constatar vantagens e desvantagens de cada método. Após a verificação da entrada, o mapeamento dos valores é feito para uma matriz e para uma árvore.

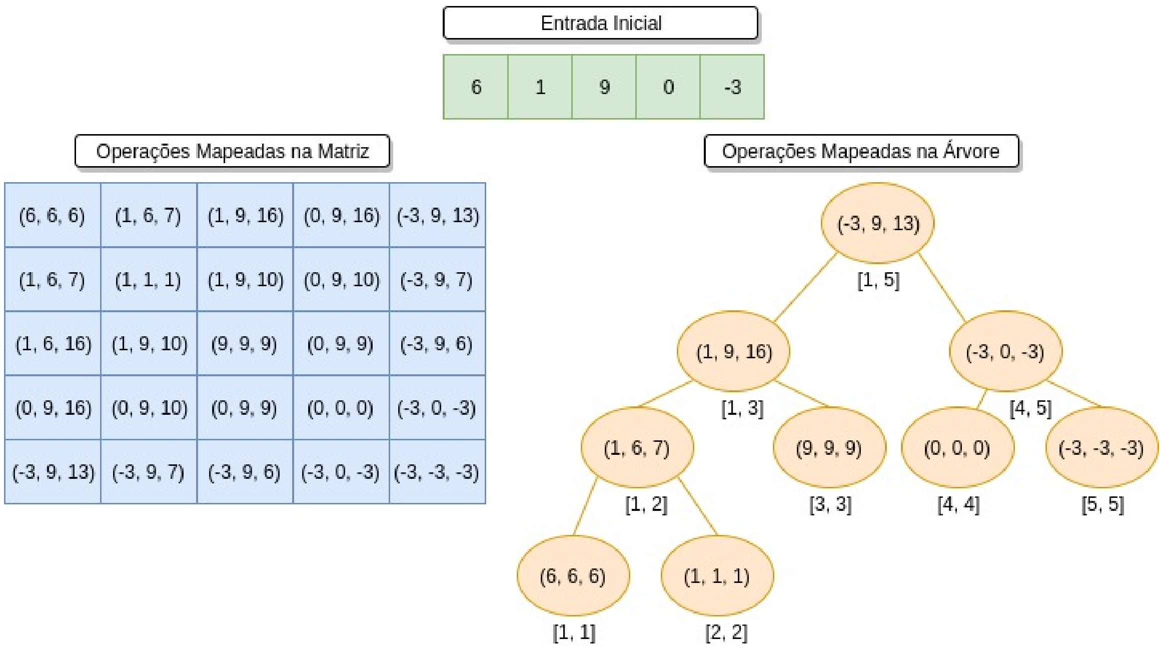

Figura 1 – Exemplo de Entrada e suas Configurações nas Estruturas Propostas

.

2.2 Matriz x Árvore de Segmentação

Como explicado anteriormente, as implementações desse trabalho tem como objetivo descobrir as vantagens e desvantagens de métodos distintos para a resolução de um mesmo problema.

2.2.1 Construção da Matriz

Para a construção da matriz todos os possíveis intervalos possíveis contidos em [1..N] devem ser representados por uma tupla (min, max, sum). Sendo assim, o intervalo do vetor v utilizado como referência é percorrido e "quebrado"em intervalos menores. Esses, são avaliados por uma função que encontra sempre o mínimo, máximo e soma correspondente. Caso o índice da linha for maior que o índice da coluna da matriz avaliada (i > j), como [5, 1], por exemplo, os valores de [1, 5] deverão ser utilizados.

```
function PROCESSAMATRIZ(matriz[N][N], v[1..N]):
    for <i=0; i<N; i++> do
        for <j=0; j<N; j++> do
            if i <= J then
                matriz[i][j] = caculaTupla(v[1..N], i+1, j+1);
            else
                matriz[i][j] = matriz[j][i];
            end if
        end for
    end for
end function
```

2.2.2 Busca na Matriz

Uma vez construída a matriz, uma consulta é bastante simples. Basta utilizar os índices do intervalo de interesse para indexar a matriz e o valor desejado é obtido. Ou seja, caso seja necessária a consulta da soma no intervalo [1,2] basta acessar $a_{1,2}$.

2.2.3 Alteração na Matriz

As funções de adição e subtração em um intervalo específico [i, j] alteram os valores do vetor v[1..N] em [i, j] desde que respeitados os limites. Com isso, uma nova matriz é criada para se obter os valores de (min, max, sum) atualizados.

2.2.4 Construção da Árvore de Segmentação

A construção da árvore é feita de maneira recursiva começando pelo lado inferior esquerdo dos nós folha até a parte superior uma vez que os nós que não são folhas são a união dos resultados dos nós filhos. Para cada nó indexado v, se v não é um nó folha, seus filhos são indexados de 2n e 2n+1. O algoritmo em questão é apresentado a seguir:

```
function constroiArvore(arvore[1...4*N+1], v[1..N], index, left, right):
    if left == right then
        arvore[index].min = v[left];
        arvore[index].max = v[left];
        arvore[index].sum = v[left];
    else
        constroiArvore(arvore, v, 2*index, left, (left+right)/2);
        constroiArvore(arvore, v, 2*index+1, ((left+right)/2)+1, right);
        arvore[index].min = min(arvore[2*index].min, arvore[2*(index)+1].min);
        arvore[index].max = max(arvore[2*index].max, arvore[2*(index)+1].max);
        arvore[index].sum = arvore[2*index].sum + arvore[2*(index)+1].sum;
    end if
end function
```

2.2.5 Busca na Árvore de Segmentação

Uma requisição da soma de valores em determinado intervalo [L, R] pode ser feita. Supondo que se esteja no nó v, que representa um intervalo [l, r], existem três possibilidades:

- **1**: [l, r] e [L, R] não possuem intereseção. Nesse caso, não há interesse no valor do nó.
- **2**: $L \leq l \leq r \leq R$. Nesse caso, o valor do nó é importante.
- **3**: Há uma parte de [l, r] que está fora de [L, R]. Sendo assim, faz sentido examinar os filhos de v para encontrar um intervalo que se encaixa melhor no desejado.

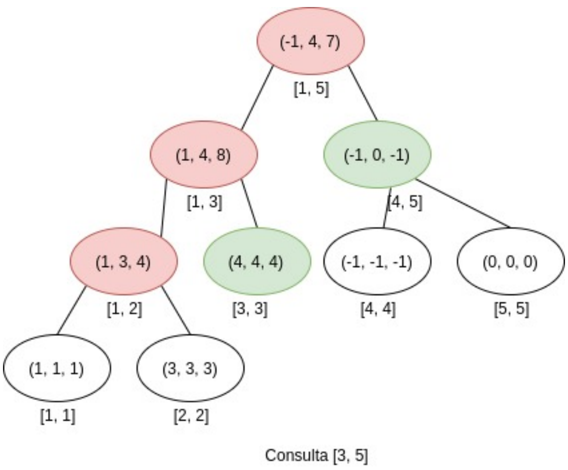

Figura 2 – Pesquisa na Árvore de Segmentação. Em vermelho, nós percorridos que não se encaixam no intervalo desejado. Em verde, nós procurados.

2.2.6 Alteração na Árvore de Segmentação

Para as funções que envolvem alterações na árvore, basta ir até o nível dos nós folhas da árvore, fazer as atualizações dos valores que correspondem ao intervalo e "subir"até a raiz da árvore enquanto atualiza os nós pelos quais se passa. Ou seja, deve ser levado em conta que os nós que não são folha, são o resultado de seus nós filhos.

2.3 Organização do Código

Tendo como preocupação a modularização do código, as operações nas diferentes estruturas foram feitas por meio de bibliotecas específicas. Mesmo assim, ambas as implementações compartilham uma estrutura que representa uma tupla (*T.h*). Portanto, a organização do código em termos de bibliotecas implementadas ficou da seguinte forma:

- **matriz.h**: fornece uma matriz dinamicamente alocada encapsulada em uma struct *Matrix* que também possui o tamanho e as funções necessárias para operá-la. Além disso, possui as operações de criação (alocação) e liberação da memória da matriz, processamento dos valores de (min, max, sum) de cada intervalo e de adição e subtração de elementos em um intervalo específico.
- **arvore.h**: fornece uma implementação da Árvore de Segmentos utilizando um vetor. Além disso, há as funções responsáveis pelas consultas nos nós da árvore e pelas modificações de valores dentro de um intervalo.

3 Análise de Complexidade

Nesta seção serão apresentadas as análises de complexidade de tempo e espaço dos algoritmos implementados.

3.1 Complexidade Temporal

A biblioteca *matriz.h* é responsável por construir a matriz através do processamento do min, max e sum dos intervalos. A alocação é feita em $O(n)$. O cálculo de mínimo, máximo e soma consiste em percorrer um intervalo definido e, por isso, a complexidade dessas operações é $O(n)$. Como é preciso montar uma matriz com os intervalos possíveis do vetor é preciso fazer o cálculo somente de posições da matriz em que i < j ou i = j (diagonal principal). Quando i > j, basta repetir o valor calculado anteriormente. Por exemplo, se a posição visitada em questão é [5, 1] basta atribuir o valor calculado em [1, 5]. Tomando N = 5 para ilustrar temos que os intervalos (1,1), (1,2), (1,3), (1,4), (1,5), (2,2), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5), (4,4), (4,5) e (5,5) precisam de suas tuplas calculadas. Os loops formados por intervalos que começam em 1 apresentam uma complexidade n!, os que começam por 2, loops de complexidade (n-1)!, por 3, (n-2)! e assim por diante. Logo, a confecção da matriz bem como a alteração no valor de um intervalo que acarreta na resconstrução da matriz é $O(n!)$. Para liberar a matriz ao final do programa, a complexidade temporal é a mesma da alocação, ou seja, $O(N)$. Para a consulta de um determinado valor, após a matriz estar pronta é $O(1)$ por ser necessário somente indexar com o intervalo desejado.

Já na biblioteca *arvore.h* a construção da árvore possui complexidade de tempo $O(n)$ uma vez que a árvore segmentada com N elementos possui 2*N-1 nós e o valor de cada nó é calculado apenas uma vez. Para a consulta, a complexidade é $O(log(n))$. Para verificar isso, é importante constatar que sempre visita-se o nível do nó raiz que possui 1 nó. Assumindo isso como hipótese verdadeira para todos os níveis $\leq n$ temos que, se visitar um nó do $n^{\text{ésimo}}$ nível, então podemos chegar até 2 no pŕoximo (uma vez que ele possua dois filho). Vale ressaltar que só são visitados nós contínuos no próximo nível. Ao visitar os dois, eles possuem 4 filhos alcançáveis. Uma observação pertinente é que nunca seŕa possível visitar 3 nós em um nível. Isso porque uma vez que o intervalo não corresponde ao esperado, os dois filhos são consultados. Ou seja, nunca é verificado um número ímpar de nós em um nível com exceção do nível da raiz. Se visitar 4 nós em um nível, então eles devem ser contínuos. Imaginando que será necessário verificar os 8 filhos, acontecerá o caso de parte do intervalo de interesse residir fora do intervalo correspondente ao dos nós. E, se houverem dois nós com esse problema, o intervalo de interesse abrange cada nó entre eles, o que contradiz a suposição de ser necessário consultar os 8 filhos. Então, visita-se a criança de 2 nós desse nível e, no máximo, de 4 nós do seguinte. Uma vez que a hipótese se aplica a todos os níveis $\leq n$ tem-se que é válido para n+1 e, por indução forte, para todos os níveis. Como há O($\log n$) níveis, a consulta vai usar no máximo 4*$\log n$ nós. Quanto a alteração de valores, é preciso ir até o(s) nó(s) folha envolvido(s) no intervalo primeiro. Após alterado, é realizada uma subida na árvore até a raiz alterando os nós pelos quais se passa. Ou seja, no pior caso, o tamanho da árvore será percorrido assim como na consulta e, por isso, é O($\log n$).

3.2 Complexidade Espacial

Ambas as implementações compartilham da mesma entrada. No caso da matriz, são armazenados valores em uma matriz de tamanho (N+1)x(N+1) e, portanto, a complexidade espacial é $O(n^2)$.

Com relação a Árvore Segmentada, a complexidade é dada por $O(n)$. Para provar, é preciso pensar em k como sendo o menor número natural tal que $2^k \geq n$. Tem-se que $2^k < 2n$. A árvore terá exatamente k+1 níveis, com o $i^{\text{ésimo}}$ contendo 2^i nós. Então, o total de nós será uma progressão geométrica da forma $2^0 + 2^1 + 2^2 + ... + 2^k$, que é igual a 2^{k+1}-1. Uma vez que $2^k < 2n$, tem-se que 2^{k+1}-1 < 4n. Logo, o número de nós da árvore é menor que 4n. Por isso, como exposto anteriormente, a complexidade é $O(n)$.

4 Análise Experimental

Para analisar o comportamento dos algoritmos desenvolvidos, foi implementado um gerador de testes para verificar o que ocorre com o crescimento do número de elementos do vetor (N) e com o crescimento do número de operações (M). Os resultados obtidos para as soluções utilizando tanto a matriz quanto a árvore são apresentados na sequência.

A Figura 3 ilustra o comportamento assintótico analisado experimentalmente para o crescimento do número de operações (M) com N fixado (N=20). Como esperado, o comportamento da matriz foi exponencial, uma vez que, por natureza do algoritmo, uma tupla (min, max, sum) de todas as possibilidades de intervalos são calculadas.

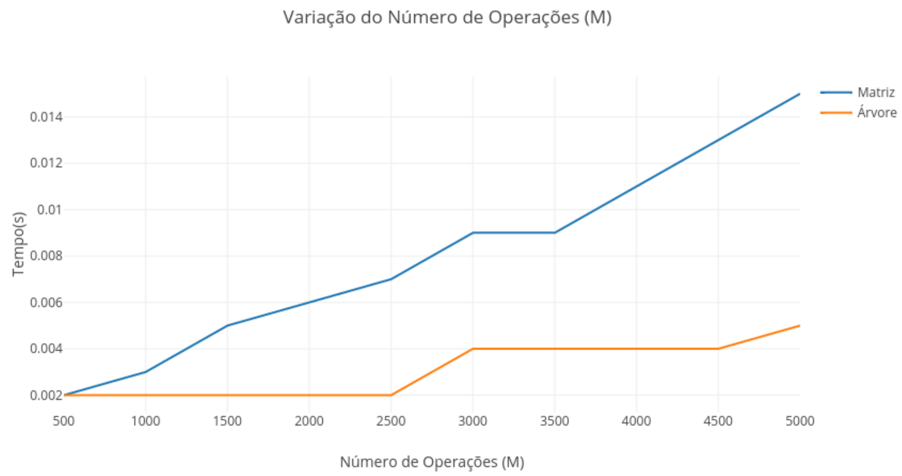

Figura 3 – Análise experimental dos algoritmos com matriz e árvore com o número de operações (M) crescente.

A Figura 4, por sua vez, ilustra o comportamento assintótico dos algoritmos para o crescimento do número de elementos do vetor (N) com o número de operações fixo (M=20). Como é perceptível nos gráficos, o resultado de ambos condiz com as análises de complexidade avaliadas anteriormente, com o algoritmo utilizando a Árvore de Segmentação apresentando uma melhor resposta e diminuindo drasticamente a complexidade e o tempo de execução.

Vale notar que apesar da consulta da matriz ser feita em $O(1)$, a sua construção é custosa e demorada em comparação com a da árvore. Além disso, quanto maior o número de elementos, maior o tamanho da matriz e mais difícil ficam as operações. Primeiro porque a consulta necessita de mapear todos os possíveis intervalos e, segundo, pois a alteração em um determinado intervalo implica na reconstrução da matriz.

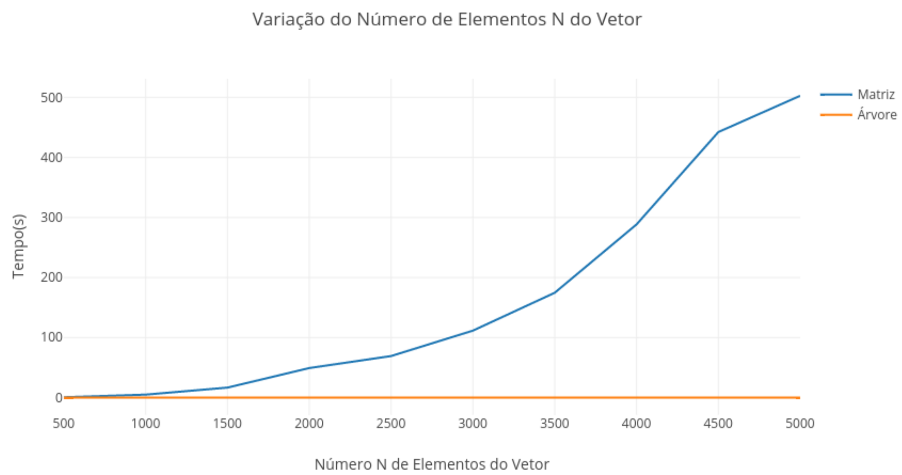

Figura 4 – Comparação de desempenho entre o algoritmo utilizando matriz e árvore para a variação crescente do número N de elementos do vetor.

5 Conclusão

Esse trabalho apresentou a solução de um problema utilizando dois métodos distintos. Após implementação e testes, é perceptível a influência de determinados métodos no que se refere ao desempenho do programa. Conhecendo paradigmas e técnicas refinadas pode-se otimizar consideravelmente a solução de um problema em termos de espaço e tempo.

Como verificado, operações que demorariam minutos no algoritmo utilizando matriz são executadas em poucos segundos pelo algoritmo usando a Árvore de Segmentos.

6 Referências Bibliográficas

Cormen, Thomas H., and Thomas H. Cormen. Introduction to Algorithms. Cambridge, Mass: MIT Press, 2001.

ZIVIANI, Nivio. Projeto de Algoritmos com implementações em PASCAL e C, 3 ed. Cengage Learning, 2011.

A Segment Tree for Christmas. Algosaurus. Acesso em: 28 ago 2017. Disponível em: <http://algosaur.us/segment-tree/#operations>.

Segment Tree | Set 1 (Sum of given range). Geeks for Geeks. Acesso em: 28 ago 2017. Disponível em: <http://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/>