UNIVERSIDADE FEDERAL DE MINAS GERAIS
Ciência da Computação
Estrutura de Dados

Lucas Rafael Costa Santos
2021017723

TRABALHO PRÁTICO III

Transformações Lineares

Belo Horizonte

2023

1 Introdução

Neste terceiro trabalho prático, exploramos a aplicação de transformações lineares em pontos desenhados em papel. A proposta consiste em determinar a posição final dos pontos, levando em consideração as transformações ao longo do tempo. Para isso, com o objetivo de otimizar as operações de consulta e atualização, optamos por utilizar uma árvore de segmentação, superando uma abordagem ingênua baseada em arrays. A formalização do problema envolve a representação das transformações como matrizes 2x2, onde as operações de atualização modificam as matrizes associadas a instantes específicos. O desafio, por sua vez, inclui a implementação eficiente de funções para realizar operações de consulta e atualização, buscando obter resultados rápidos e precisos.

2 Método

2.1 Estrutura de Dados

Como foi dito anteriormente, para solucionar este problema utilizamos uma árvore de segmentação a fim conseguirmos computar as operações de forma eficiente. Uma árvore de segmentação (*segtree*) é uma estrutura de dados que permite realizar operações em intervalos de um vetor, como somar, multiplicar, encontrar o mínimo ou o máximo, etc. Esta estrutura é construída a partir de um vetor de valores, dividindo-o em segmentos menores e armazenando-os em uma árvore binária, sendo que, cada nó da árvore representa um segmento do vetor, e o valor do nó é o resultado da operação aplicada ao segmento. Por exemplo, se a operação for a soma, o valor do nó será a soma dos elementos do segmento correspondente. A árvore de segmentação permite realizar consultas e atualizações em intervalos do vetor de forma eficiente, usando a propriedade de que um segmento pode ser decomposto em dois segmentos menores, que são filhos do nó que representa o segmento original.

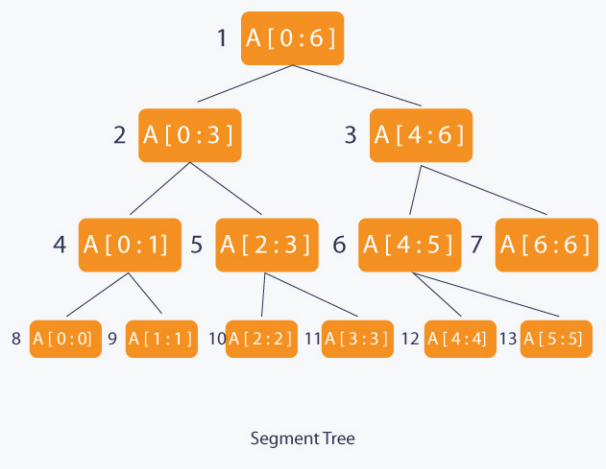

Figura 1 – Segtree

2.2 Classes

Foi criada a classe "SegTree"para a implementação de uma árvore de segmentos. Essa classe fornece funcionalidades essenciais, como um construtor e um destrutor. Adicionalmente, a classe possui métodos para atualizar um elemento em uma posição específica na árvore (por meio da função "update") e para realizar consultas em um intervalo específico na árvore (por meio da função "query"). A implementação inclui métodos privados para construir a árvore, atualizar recursivamente e realizar consultas recursivas. A árvore é representada por um array de nós do tipo "TipoNo", e seu tamanho é especificado durante a inicialização.

```
class SegTree {

public:
    SegTree(int size);
    ~SegTree();

    void update(int idx, const Matrix& val);
    Matrix query(int l, int r);

private:
    int n;
    TipoNo* tree;

    void build(int node, int start, int end);
    void update(int node, int start, int end, int idx, const Matrix& val);
    Matrix query(int node, int start, int end, int l, int r);
};
```

Listing 1 – Classe SegTree

Também foi elaborada a classe "TipoNo"para a representação de um nó da árvore. Essa classe possui um construtor padrão que inicializa uma matriz 2x2 associada ao nó. Além disso, cada nó contém dois ponteiros, um para o filho à esquerda ("esq") e outro para o filho à direita ("dir"). Algo a se destacar é que essa classe é amiga da classe "SegTree", o que permite à esta o acesso aos membros privados necessários para a construção da árvore de segmentos.

Ademais, foi definida a estrutura "Matrix"para representar uma matriz 2x2, com elementos a, b, c e d, correspondendo à matriz [a, b; c, d]. A constante "MODULO"foi declarada com o valor 100000000, a fim de garantir que apenas os 8 dígitos menos significativos sejam mantidos. E, por fim, para realizar as operações de multiplicação entre matrizes, foi implementada a função "multiply", que recebe duas matrizes como parâmetros e retorna o resultado da multiplicação.

```
const int MODULO = 100000000;

// Representa uma matriz 2x2
struct Matrix {
    int a, b, c, d; // Representa a matriz [a, b; c, d]
};

Matrix multiply(const Matrix& m1, const Matrix& m2);

class TipoNo {
public:
    TipoNo();

private:
    Matrix matriz; // Matriz associada a este no
    TipoNo *esq;
    TipoNo *dir;

    friend class SegTree;
};
```

Listing 2 – Classe TipoNo

3 Análise de Complexidade

A análise de complexidade do código envolve a avaliação do tempo de execução em termos do tamanho da entrada. No caso de estruturas de dados como a nossa árvore de segmentos, podemos expressar a complexidade em termos de operações realizadas durante as atualizações e consultas. Sendo assim, vamos analisar cada parte do código separadamente.

3.1 Complexidade de Tempo

1. Construção da Árvore de Segmento (build):

- A função build é chamada recursivamente em cada nó da árvore, sendo que, em cada chamada, são realizadas operações de tempo constante. Como cada nível da árvore é percorrido uma vez, podemos concluir que a complexidade de tempo é O(n), sendo n o número de elementos.

2. Atualização da Árvore de Segmento (update):

- A função update é chamada recursivamente no caminho da raiz até a folha correspondente ao índice atualizado. Cada chamada realiza operações de tempo constante, sendo que, o

número de chamadas é proporcional à altura da árvore, que por sua vez é log n (onde n é o número de elementos). Assim, a complexidade de tempo é O(log n) para a operação de atualização.

3. Consulta na Árvore de Segmento (query):

- A função query é chamada recursivamente no caminho da raiz até os nós que correspondem ao intervalo de consulta. Cada chamada realiza operações de tempo constante, de modo que, o número de chamadas é proporcional à altura da árvore (log n). Portanto, a complexidade de tempo é O(log n) para a operação de consulta.

4. Multiplicação de Matrizes (multiply):

- Por fim, temos a multiplicação de matrizes, que é feita em tempo constante, pois envolve apenas operações aritméticas básicas. Então, sua complexidade é O(1).

5. Loop Principal (main):

- O loop principal é executado q vezes, onde q é o número de consultas. Como cada iteração do loop executa operações de atualização ou consulta, ambas com complexidade de tempo O(log n) no pior caso, a complexidade de tempo total é O(q log n).

Portanto, podemos concluir que a complexidade de tempo total do código é dominada pelo loop principal, que possui complexidade O(q log n). As operações de consulta e atualização na árvore de segmento também possuem certo impacto, sendo ambas O(log n). As demais operações, como construção da árvore e multiplicação de matrizes, contribuem com complexidade constante ou linear, mas são eclipsadas pela árvore de segmento em entradas significativamente grandes.

3.2 Complexidade de Espaço

1. Árvore de Segmentos (tree):

- O espaço utilizado para armazenar a árvore de segmentos é O(n), onde n é o número de elementos e cada nó da árvore ocupa um espaço constante.

2. Matrizes:

- As matrizes são armazenadas como estruturas de quatro inteiros e cada matriz ocupa um espaço constante.

3. Outras Variáveis Locais:

- As variáveis locais nas funções main, update, e query ocupam um espaço constante.

Portanto, é possível concluír que a complexidade de espaço total é O(n).

4 Estratégias de Robustez

Foram implementadas algumas estratégias para garantir robustez na manipulação de dados e execução do programa. Entre elas, temos:

1. Verificação de Erros na Leitura de Entrada:

Quando é feita a leitura do número de elementos e consultas (std::cin » n » q), há uma verificação de falha na leitura usando std::cin.fail(). Se a leitura falhar, uma mensagem de erro é exibida, indicando um problema na entrada.

```
if (std::cin.fail()) {
    std::cerr << "Erro na leitura de entrada." << std::endl;
    return 1;
}
```

Listing 3 – Erros na Leitura

2. Verificação de Entrada Não Positiva:

Após a leitura de n e q, há uma verificação para garantir que ambos sejam valores positivos, sendo exibido um erro, caso contrário.

```
if (n <= 0 || q <= 0) {
    std::cerr << "Entrada invalida, os valores devem ser positivos." << std
    ::endl;
    return 1;
}
```

Listing 4 – Entrada não positiva

3. Verificação de Índice de Atualização Inválido:

Ao realizar uma operação de atualização (op == 'u'), é verificado se o índice t está dentro dos limites válidos (0 <= t < n). Se for inválido, uma mensagem de erro é exibida.

```
if (idx < 0 || idx >= n) {
    std::cerr << "Indice de atualizacao invalido." << std::endl;
    return;
}
```

Listing 5 – Índice de atualização inválido

4. Verificação de Índices de Consulta Inválidos:

Ao realizar uma operação de consulta (op == 'q'), são verificados os índices t0 e td para garantir que estão dentro dos limites válidos (0 <= t0 <= td < n). Se forem inválidos, uma mensagem de erro é exibida, e a matriz identidade é retornada como resultado.

```
if (l < 0 || r >= n || l > r) {
    std::cerr << "Indices de consulta invalidos." << std::endl;
    // Retorna a matriz identidade em caso de erro
    return Matrix{1, 0, 0, 1};
}
```

Listing 6 – Índices de Consulta Inválidos

Além destes métodos, também realizamos testes utilizando o Valgrind para assegurar a ausência de potenciais vazamentos de memória que poderiam comprometer a integridade do nosso código.

Essas são algumas das verificações que ajudam a garantir que o programa lide adequadamente com situações inesperadas durante a execução, melhorando a robustez e evitando comportamentos inesperados ou erros difíceis de diagnosticar.

5 Análise Experimental

Para conduzir a análise experimental, empregamos o Gerador de Casos de Teste a fim de fornecer diversas entradas. Inicialmente, variamos o número de instantes de tempo em incrementos de 1.000, abrangendo de 1.000 até 100.000 e mantendo constante o número de operações em 1.000. Em seguida, invertemos essa abordagem, mantendo o número de instantes de tempo em 1.000 e variando o número de operações de 1.000 em 1.000.

Posteriormente, por meio da biblioteca Chrono, registramos o tempo de execução do algoritmo para as diferentes entradas, possibilitando a representação gráfica da relação entre os diferentes números de instantes de tempo e operações com o tempo de execução.

O gráfico a seguir ilustra a evolução do tempo de execução à medida que o número de instantes de tempo é incrementado.

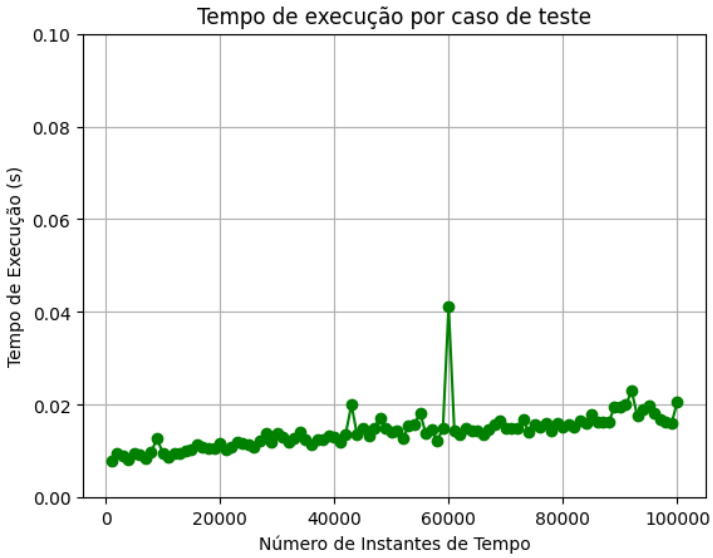

Figura 2 – Gráfico Tempo x Instantes

Ao analisar os resultados, observamos que, de maneira geral, mesmo com o aumento constante de ’n’, o tempo de execução cresce de forma mais gradual, assemelhando-se a um gráfico logarítmico. Essa observação está alinhada com o que foi discutido no tópico de análise de complexidade.

Já o gráfico abaixo destaca a correlação entre o tempo de execução e o aumento do número de operações.

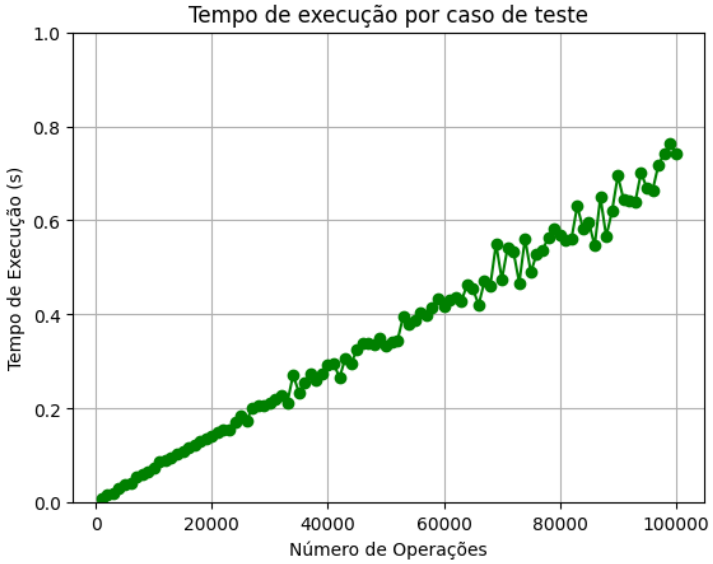

Figura 3 – Gráfico Tempo x Operações

Em comparação com o gráfico anterior, torna-se evidente que este exibe um crescimento mais acentuado. Como o número de operações está aumentando, esse comportamento pode ser atribuído à complexidade presente no loop principal, identificada anteriormente como O(q log n). Em outras palavras, à medida que 'q' aumenta, o tempo de execução também experimenta um aumento proporcional.

6 Conclusão

Neste trabalho, abordamos a aplicação de transformações lineares em pontos desenhados em papel. A necessidade de determinar a posição final dos pontos, considerando diversas transformações ao longo do tempo, levou-nos a uma abordagem eficiente através da implementação de uma árvore de segmentação. Ao contrário de uma solução ingênua baseada em arrays, a estrutura de dados proposta permitiu-nos realizar operações de consulta e atualização de forma rápida e otimizada.

Este projeto proporcionou uma compreensão mais profunda da estrutura de árvore de segmentação, a qual pode ser de grande utilidade para resolver inúmeros problemas. Além disso, destacou a importância da análise de complexidade do código, evidenciando como entradas mais extensas podem impactar o desempenho do programa. Aprendemos também a valorizar a análise experimental como uma ferramenta diferencial para verificar a consistência do código, identificar possíveis problemas e aprimorar a compreensão do seu funcionamento.

7 Bibliografia

- IBM. Decision Tree Models. IBM. Disponível em: https://www.ibm.com/docs/pt-br/spss-modeler/18.4.0?topic=trees-decision-tree-models. Acesso em: 03 de dezembro de 2023.

- HACKEREARTH. Segment Tree and Lazy Propagation. HackerEarth. Disponível em: https://www.hackerearth.com/practice/notes/segment-tree-and-lazy-propagation/. Acesso em: 03 de dezembro de 2023.

- MARATONA UFMG. Aula 9 - SegTree. Maratona UFMG, 2021. Disponível em: https://www.youtube.com/watch?v=OW_nQN-UQhA&ab_channel=MaratonaUFMG. Acesso em: 25 de novembro de 2023.

- Aulas de Estruturas de Dados do Professor Wagner Meira Jr, Departamento de Ciência da Computação - UFMG