SegTree para Transformações Lineares

Trabalho Prático 3

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

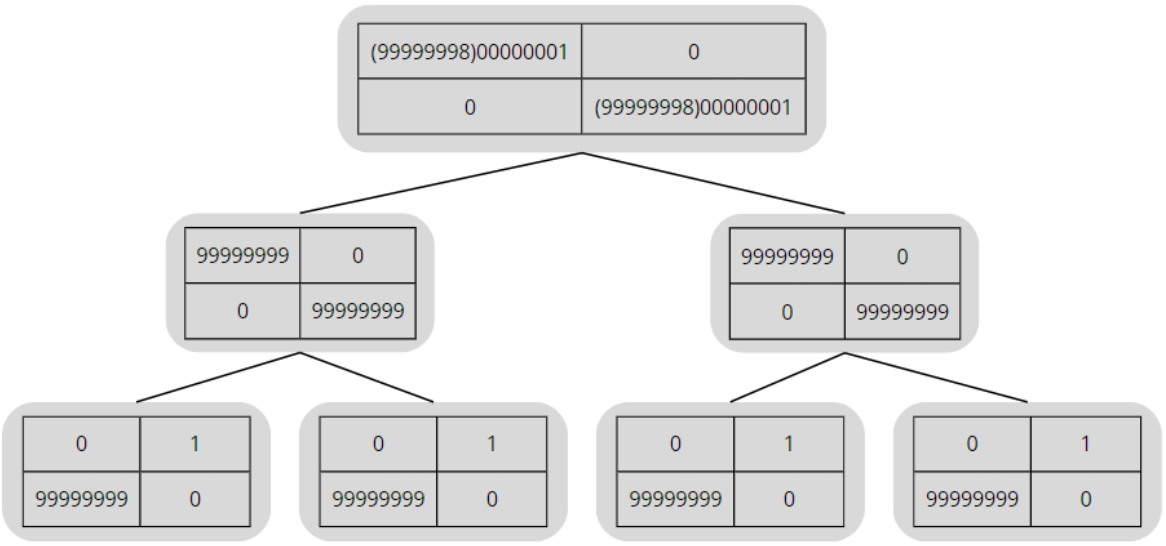
Nome: Letícia Scofield Lenzoni
Matrícula: 2022035547

1. Introdução

Neste Trabalho Prático, exploramos a aplicação de uma árvore de segmentação (SegTree) para resolver um problema específico envolvendo transformações lineares em pares de coordenadas bidimensionais. A SegTree é uma estrutura de dados fundamental na computação que permite realizar consultas e atualizações em intervalos de um conjunto de dados. No contexto deste trabalho, utilizamos ela para otimizar as operações de atualização e consulta em uma matriz 2x2 que representa transformações lineares.

O problema consiste em realizar duas operações principais: a atualização de uma matriz de um determinado instante e a consulta do resultado da multiplicação cumulativa das matrizes em um intervalo específico. Cada ponto da SegTree armazena o resultado acumulado da multiplicação das matrizes associadas a seus descendentes.

Para exemplificar, consideremos o caso 3 do enunciado do trabalho. Primeiramente, foi criada uma árvore com 4 instantes (do 0 a 3) com matrizes identidade, depois todos eles foram atualizados para a matriz de valores 0, 1, 99999999 e 0, respectivamente. Como consequência disso, os pais também foram atualizados com a multiplicação dos filhos, obtendo a seguinte árvore:



Nessa imagem, está representado os intervalos que são mantidos na árvore e, na raiz, a matriz do intervalo completo (de 0 a 3), como esse é o intervalo requerido nas consultas do exemplo 3, essa será a matriz considerada. A parte dos números em parênteses na imagem (99999998) na verdade não está presente na árvore, foi colocada apenas para melhor entendimento, pois apenas os 8 dígitos menos significativos serão considerados na impressão, então apenas eles são armazenados na matriz.

Com a matriz resultante já definida, é necessário apenas multiplicá-la pelas coordenadas x e y dadas na entrada. No caso, as duas consultas são (1,0) e (0,1) gerando então as seguintes operações e resultados esperados:

Consulta 1:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Consulta 2:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Desse modo, é possível entender claramente tudo que engloba o problema.

2. Método

2.1. Descrição da Implementação

O código foi desenvolvido em C++ para abordar a problemática apresentada, utilizando a estrutura de dados Segment Tree (SegTree). A implementação visa otimizar a realização de operações de atualização e consulta em uma matriz 2x2, representando transformações lineares bidimensionais.

2.2. Estruturas de Dados

Matriz:

A estrutura de dados Matriz representa uma matriz 2x2 de números inteiros positivos, essencial para as operações de transformações lineares. Cada instância dessa estrutura armazena os 4 elementos da matriz (m[0][0], m[0][1], m[1][0] e m[1][1]) e cada elemento é armazenado como um número de 64 bits para que haja precisão no cálculo.

SegTree:

A estrutura de dados SegTree foi implementada de forma particular. Ao invés de utilizar uma representação explícita da árvore, optou-se por armazenar os nós em um heap. Cada nó da árvore é armazenado sequencialmente no heap, e as relações entre os nós são mantidas por meio de índices.

2.3. Funções e Métodos Implementados do SegTree.cpp

Atualizacao(int indiceSegTree, int inicioIntervalo, int fimIntervalo, int p, Matriz *mat)

Este método realiza uma atualização na árvore de segmentação, modificando o valor do elemento na posição p pela matriz fornecida (mat). A atualização ocorre de forma recursiva, percorrendo os nós da árvore até o nó folha correspondente ao intervalo que contém o elemento p, depois disso ele atualiza os valores de todos os ascendentes desse nó de acordo com a mudança em p.

Consulta(int indiceSegTree, int inicioIntervalo, int fimIntervalo, int consultaInicio, int consultaFim)

Esse método realiza uma consulta na árvore de segmentação, retornando o resultado da multiplicação matricial dentro do intervalo especificado. A consulta também é realizada de forma recursiva, considerando os intervalos associados aos nós da árvore.

multiplicar(const Matriz &a, const Matriz &b)

Este método privado realiza a multiplicação de duas matrizes e retorna o resultado. Essa operação é essencial para as operações de atualização e consulta na árvore de segmentação. Além disso, ele também usa “% 100000000” para retornar apenas os 8 dígitos

menos significativos, como especificado no enunciado, impedindo que os valores tendam ao infinito (ou, na realidade, ao limite da máquina) para casos consideravelmente grandes.

3. Análise de Complexidade

3.1. Análise de Complexidade de Tempo

Atualizacao(int indiceSegTree, int inicioIntervalo, int fimIntervalo, int p, Matriz *mat)

- Complexidade de Tempo: O(log n)
- A função Atualização, ao ser chamada recursivamente, percorre uma árvore binária balanceada de altura logarítmica (devido à recursividade). Cada nível da árvore realiza uma operação de atribuição e comparação, e a altura logarítmica implica em uma complexidade de tempo total logarítmica.

Consulta(int indiceSegTree, int inicioIntervalo, int fimIntervalo, int consultaInicio, int consultaFim)

- Complexidade de Tempo: O(log n)
- Da mesma forma que Atualização, a função Consulta é recursiva e opera em uma árvore binária balanceada de altura logarítmica. Portanto, a complexidade de tempo é logarítmica em relação ao número de nós.

multiplicar(const Matriz &a, const Matriz &b)

- Complexidade de Tempo: O(1)
- Como as matrizes envolvidas na multiplicação têm dimensões fixas (2x2), o número de operações necessário para calcular o resultado é constante. Ou seja, independente do tamanho da entrada, a quantidade de operações não aumenta.

3.2. Análise de Complexidade de Espaço

Atualizacao(int indiceSegTree, int inicioIntervalo, int fimIntervalo, int p, Matriz *mat)

- Complexidade de Espaço: O(log n)
- A recursividade da função Atualização implica em chamadas de função adicionais, e o espaço necessário para a pilha de chamadas recursivas é logarítmico em relação ao número de vértices.

Consulta(int indiceSegTree, int inicioIntervalo, int fimIntervalo, int consultaInicio, int consultaFim)

- Complexidade de Espaço: O(log n)
- Da mesma forma que Atualização, a função Consulta também é recursiva e requer espaço logarítmico na pilha de chamadas.

multiplicar(const Matriz &a, const Matriz &b)

- Complexidade de Espaço: O(1)
- A função opera diretamente no array de vértices, sem exigir espaço adicional que cresça com o tamanho da entrada.

4. Estratégias de Robustez

Tratamento de Erros:

- Gestão de Memória: O programa implementa cuidados adequados para lidar com a alocação e liberação de memória. Em situações de falha na alocação de memória, mecanismos são acionados para garantir a liberação correta dos recursos alocados, evitando vazamentos de memória que poderiam comprometer o desempenho e a estabilidade do programa.
- Índices Válidos: Durante a manipulação das estruturas de dados, verificações são incorporadas para garantir que os índices estejam dentro dos limites apropriados. Isso previne possíveis falhas de segmentação que poderiam ocorrer devido a acessos indevidos a áreas de memória não alocadas.
- Comandos Válidos: o código aceita apenas as operações de atualização (u) e consulta (q), qualquer outro caractere escrito levará para a impressão de “Comando Inválido” e o programa será encerrado.

Gerenciamento de Memória:

- O código adota procedimentos sólidos para liberar a memória alocada para as estruturas de dados, como a matriz e a árvore de segmentação. Essa prática é essencial para prevenir vazamentos de memória ao encerrar a execução do programa.

Tratamento Adequado de Casos Especiais:

- Casos Especiais: O código é projetado para lidar com situações especiais, como grafos vazios ou grafos com um número mínimo de vértices. Essas verificações são incorporadas para evitar comportamentos inesperados e garantir que o programa trate de maneira consistente casos específicos que possam surgir durante a execução.

Ao implementar essas estratégias de robustez, o código está preparado para lidar com diversas situações e entradas, proporcionando um funcionamento seguro e confiável.

5. Análise Experimental

Para realizar a análise experimental do código, foram conduzidos experimentos abrangentes utilizando uma variedade de grafos e suas respectivas descrições. Os casos de teste foram diversificados em termos de tamanho do intervalo, operações de atualização e consultas na árvore de segmentação.

Caso de Teste 1

Descrição:

- Grafo com 100 vértices.
- 50 operações de atualização (comando 'u') em vértices aleatórios.
- 50 operações de consulta (comando 'q') em intervalos aleatórios.

Objetivo:

- Testar a capacidade do programa em lidar com árvores de tamanho moderado.

Conclusão:

- A árvore de segmentação forneceu respostas eficientes para consultas, refletindo em um bom desempenho.

Caso de Teste 2

Descrição:

- Grafo com 1000 vértices.
- 500 operações de atualização (comando u) em nós aleatórios.
- 500 operações de consulta (comando q) em intervalos aleatórios.

Objetivo:

- Testar a escalabilidade do programa para árvores de tamanho significativamente maior.

Conclusão:

- A árvore de segmentação lidou eficientemente com consultas mesmo para grafos mais extensos, não houve grande alteração, possivelmente por conta do algoritmo se logarítmico.

Caso de Teste 3

Descrição:

- Grafo com 500 vértices.
- 250 operações de atualização (comando 'u') em vértices aleatórios.
- 250 operações de consulta (comando 'q') em intervalos aleatórios.
- Inclusão de comandos “u” consecutivos para avaliar a robustez do tratamento de atualizações sequenciais.

Objetivo:

- Testar se o programa mantém a consistência e eficiência nas operações mesmo em cenários de atualizações frequentes.

Conclusão

- Felizmente, como nos outros casos, o programa se manteve consistente ao esperado, mesmo com os casos consecutivos de atualização.

Em síntese, independente da quantidade de atualizações, consultas e tamanho, código escala bem, mantendo o esperado teoricamente, visto na análise de complexidade.

6. Conclusão

Este trabalho consolidou a implementação de um programa eficiente para manipulação de grafos, utilizando árvore de segmentação e operações com matrizes. Através da simulação de casos de teste, foi possível avaliar a robustez do programa em situações diversas, verificando sua consistência e eficiência nas operações de atualização e consulta. As estratégias adotadas, como tratamento adequado de casos especiais, gestão de memória e tratamento de erros, contribuem para a confiabilidade e eficácia do sistema. Assim, este trabalho não apenas oferece uma solução funcional para o problema proposto, mas também destaca a aplicação prática de conceitos fundamentais de estruturas de dados e algoritmos no desenvolvimento de soluções computacionais.

7. Bibliografias

MONTEIRO, Bruno. Aula 9 - SegTree. 2023. Video. Maratona UFMG. Disponível em: https://www.youtube.com/watch?v=OW_nQN-UQhA. Acesso em 29 nov. 2023.

CODE_MARATHON. Árvore de Segmento. Disponível em: https://www.codemarathon.com.br/conteudos/estrutura-de-dados/arvore-de-segmento. Acesso em 30 nov. 2023.

CHAIMOWICZ, Luiz. Árvores - Algoritmos de Caminhamento. 2023. Video. DCC205-UFMG. Disponível em: https://www.youtube.com/watch?v=0c35O7RcKMk. Acesso em 30 nov. 2023.

8. Instruções para compilação e execução

Compilação

- Primeiramente, deve-se extrair os conteúdos do ZIP do projeto em uma pasta.
- Antes de compilar o programa, é necessário ter instalado o GCC.
- Para compilar o programa, após abrir um terminal no diretório raiz do projeto, utilize o seguinte comando: make all

Execução

- Para a execução, utilize o seguinte comando: ./bin/tp3.out
- Após isso, será necessário escrever na primeira linha quantos instantes de tempo (n) e quantas operações serão realizadas (q). Nas linhas seguintes, deverá ser colocado o comando (‘u’ ou ‘q’) e as informações específicas dele. Para ‘u’, pede-se instante de tempo (a) que deve ter sua transformação alterada e a nova matriz 2x2

(ou seja, mais 4 inteiros). Para ‘q’, pede-se 4 inteiros (t0, td, x e y), indicando respectivamente os instantes de “nascimento” e “morte” do ponto e suas coordenadas no plano. Faz-se isso até terminar o número de operações requeridas no início.