Prof. Wagner Meira Jr. DCC/ ICEx / UFMG

DCC205 - Estrutura de Dados 2023.2

Trabalho Prático 3

Lucca Alvarenga de Magalhães Pinto - lucca.alvarenga@dcc.ufmg.br

Matrícula: 2021036736

1. Introdução

O presente projeto busca resolver um problema envolvendo a manipulação eficiente de transformações lineares aplicadas a pontos em um plano bidimensional. Inicialmente, temos uma sequência de matrizes 2x2 representadas por $A1, A2, \ldots, An$, em que cada matriz Ai é aplicada aos pontos no instante de tempo i. As operações permitidas são de dois tipos: atualização e consulta.

Na operação de atualização, o usuário escolhe um instante de tempo i e substitui a matriz Ai por uma matriz B válida e não há saída. Já na operação de consulta, o usuário escolhe os instantes de "nascimento" $(t0)$ e "morte" (td) de um ponto, juntamente com suas coordenadas (x, y) no momento de nascimento, com a finalidade de determinar as coordenadas x e y que o ponto terá quando desaparecer, considerando apenas os 8 dígitos menos significativos.

O objetivo principal deste projeto é implementar uma Árvore de Segmentação para realizar as operações de consulta e atualização de maneira eficaz. Para tanto, serão implementadas três funções: construção da segtree (build), realização de consultas (query) e realização de atualizações (update). A implementação será guiada pela estrutura específica da segtree, tendo como base principal a construção dessa estrutura feita pela Maratona de Programação da UFMG fornecida pelo enunciado.

2. Método

2.1. Configurações da Máquina:

O código foi executado nas seguintes configurações de ambiente:

- Sistema operacional: Ubuntu 20.04.6 LTS (Focal Fossa)
- Linguagem de programação: C++
- Compilador: gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
- Processador: 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz
- Memória: 15Gb

2.2. Árvore de Segmentação:

A implementação da Árvore de Segmentação tem como ideia armazenar as matrizes 2 por 2 de forma a pré-computar o resultado para subarranjos específicos em seus nós, assim executando as operações de consulta e atualização de maneira mais eficiente e otimizada. Abaixo é descrito as principais funções da árvore:

2.2.1. Construção

A função “build” da classe SegTree é responsável por criar a árvore de segmentação, iniciando cada nó com a matriz identidade, de forma que cada nó representa um intervalo dos instantes de matrizes. O processo ocorre de forma recursiva até o momento em que cada nó folha tenha uma matriz de transformação. A função recebe 3 parâmetros: “posição” referente ao índice do nó atual na árvore, além do “início” “fim” correspondentes ao índice inicial e final do intervalo coberto pelo nó atual da árvore.

A construção ocorre da seguinte forma:

- Se os limites do intervalo forem iguais, ou seja, “início” for igual a “fim”, então o nó atual é uma folha. Neste caso, atribuímos à aquele nó a matriz identidade.
- Se não estamos em uma folha, calculamos o ponto médio daquele intervalo e, de forma recursiva, é criado os nós filhos da esquerda e da direita.

- Atualizamos aquela posição da árvore como a matriz resultante da multiplicação das matrizes desses dois nós filhos esquerdo e direito. Essa nova matriz representa o intervalo atual.

Essa abordagem otimiza as operações de consulta ao pré-computar os resultados para subintervalos, tornando-as mais eficientes durante as operações subsequentes na árvore de segmentação.

2.2.2. Consulta

A função “query” na classe SegTree tem como objetivo realizar consultas sobre a árvore, buscando obter a matriz resultante gerada pelas transformações lineares aplicadas a um conjunto de pontos durante um intervalo de tempo específico. Essa pesquisa ocorre da seguinte forma:

- A função inicia verificando se o intervalo de tempo definido para a consulta (representado por $t0$ e td) não se sobrepõe ao intervalo coberto pelo nó atual da árvore (determinado por “inicio” e “fim”). Se não houver interseção, é retornado o elemento neutro da multiplicação de matrizes, a matriz identidade.

- Se o intervalo coberto pelo nó atual estiver completamente dentro do intervalo de consulta ($t0$ <= “inicio” e “fim” <= td), a função retorna a matriz armazenada naquele nó atual. Essa matriz representa a transformação acumulada para o intervalo total coberto pelo nó, ou seja, não há necessidade de buscar mais na árvore.

- Caso contrário, a função calcula o ponto médio e realiza consultas recursivas para os nós filhos esquerdo (2 * posicao) e direito (2 * posicao + 1) correspondentes aos subintervalos resultantes da divisão.

- As matrizes obtidas das consultas nos nós filhos são multiplicadas para obter a matriz resultante, representando a transformação acumulada para o intervalo atual. A matriz resultante é então retornada como o resultado final da consulta.

Portanto, a função percorre a estrutura da árvore de segmentação de forma eficiente, multiplicando as transformações lineares associadas aos intervalos de tempo parcial.

2.2.3. Atualização

A função “update” na classe SegTree é responsável por atualizar um nó específico na árvore com uma nova matriz fornecida. Ela apresenta 5 parâmetros: “indice” que representa o ponto de atualização, “novaMatriz” que substituirá a matriz existente no nó correspondente, “posicao” que é o índice do nó atual na árvore, e os limites “inicio” e “fim” referente ao índice inicial e final do intervalo coberto pelo nó atual.

Nesse contexto, a método é realizada da seguinte maneira:

- Se o índice do vetor de matrizes (indice) não estiver dentro do intervalo representado pelo nó atual, a função retorna a matriz armazenada naquele nó em questão.
- Se o intervalo representado pelo nó atual é uma folha (ou seja, “inicio” é igual a “fim”), o nó recebe a nova matriz e a função retorna essa nova matriz.
- Caso contrário, a função calcula o ponto médio meio do intervalo e continua o processo de atualização de forma recursiva para os filhos esquerdo (2 * posicao) e direito (2 * posicao + 1).
- Os resultados obtidos dos nós filhos são multiplicados para obter a matriz resultante que representará o intervalo atual. A matriz anteriormente armazenada no nó atual é substituída pela nova matriz resultante.

Dessa forma, a função update garante que a árvore de segmentação seja atualizada de maneira eficiente após uma mudança na matriz de transformação em um ponto específico do vetor. A recursividade é fundamental para propagar a atualização ao longo dos nós da árvore afetados pelo intervalo representado pelo ponto alterado, de forma a percorrer recursivamente em direção aos nós “pais”.

3. Análise de Complexidade

Abaixo é identificando a complexidades das principais funções do projeto que foram usadas para construir, consultar e atualizar a árvore de segmentação:

1. Struct Matriz

1.1. Matriz(long long a, long long b, long long c,long long d):

Tem uma complexidade de tempo constante O(1), já que realiza operações simples de atribuição, sem depender do tamanho de alguma entrada ou executar iterações. A complexidade de espaço da função também é constante O(1). A função cria uma matriz de tamanho fixo 2x2 para armazenar os valores fornecidos como argumentos, independentemente do tamanho da entrada.

1.2. Matriz()

Tem uma complexidade de tempo e espaço constantes O(1). Ela realiza operações simples de atribuição para inicializar os elementos da matriz identidade 2x2, não depende do tamanho da entrada e não envolve iterações. A matriz criada é alocada na pilha de memória, ocupando um espaço constante independentemente do tamanho da entrada.

1.3. print()

Possui uma complexidade de tempo linear, a execução da função é proporcional ao número fixo de elementos na matriz, tornando-a linear. A complexidade de espaço é também constante O(1), já que não utiliza espaço adicional que cresce com o tamanho da entrada..

2. Classe SegTree

2.1. build()

A complexidade de tempo e espaço da função é linear O(n), onde "n" é o tamanho da árvore segmentada. A função percorre cada nó da árvore uma vez, atribuindo valores às posições correspondentes. A operação de multiplicação é realizada em cada nó e é de tempo constante. Em relação ao espaço, a complexidade é referente a alocação dinâmica do array segTree, que armazena os valores das matrizes em cada nó da árvore. A quantidade total de espaço utilizado pela árvore é proporcional ao número de nós na árvore, que é O(n).

2.2. update()

A complexidade de tempo da função é logarítmica O(log n), onde "n" é o tamanho da árvore segmentada. Assim como em operações de consulta, a função percorre a árvore em altura, visitando no máximo dois nós em cada nível (um à esquerda e outro à direita). A complexidade é, portanto, proporcional à altura da árvore, que é logarítmica em relação ao número total de folhas.

2.3. query()

A complexidade de tempo da função é logarítmica O(log n), onde "n" é o tamanho da árvore segmentada. A função realiza uma busca descendente na árvore segmentada para encontrar os intervalos relevantes. Em cada nível da árvore, a função visita no máximo dois nós (um à esquerda e outro à direita). Portanto, a complexidade de tempo é proporcional à altura da árvore, que é logarítmica em relação ao número total de folhas.

2.4. multiplicar()

A complexidade de tempo da função é O(1), já que o número de operações necessárias para multiplicar duas matrizes 2x2 é constante. A complexidade de espaço também é O(1), pois ela utiliza uma quantidade constante de espaço para armazenar variáveis locais, independentemente do tamanho da entrada.

4. Estratégia de Robustez

No projeto, foram consideradas várias medidas para garantir a qualidade do código e a identificação de erros. Aqui estão algumas das principais práticas adotadas:

Tratamento de Exceções: Foram implementadas exceções em funções que podem encontrar erros durante a execução, como verificar no construtor da SegTree se o tamanho dela é maior que zero, verificar se a opção passada no “int main” é de consulta ou atualização, entre outras exceções. Isso torna o código mais robusto e permite a identificação e tratamento adequado de problemas, caso ocorram. Além disso, as estruturas de dados utilizadas para armazenar informações possuem destrutores próprios para não permitir vazamento de memória.

Testes com Doctest: Foram feitos testes para a classe SegTree e para o Struct Matriz usando o framework Doctest. Isso incluiu testes de unidade para as funções e métodos, bem como testes de integração para garantir que as classes funcionassem bem juntas. Os testes ajudaram a identificar problemas no código e a verificar se o código estava produzindo os resultados esperados.

Verificação de Memória com Valgrind: A ferramenta Valgrind foi usada para verificar a consistência da memória no código. Isso ajudou a garantir que o código não tivesse problemas de gerenciamento de memória que podiam causar falhas ou problemas de desempenho. No final, não foram identificados nenhum erro relacionado a alocação de memória

Apesar disso, deve-se destacar que o programa foi feito com base no enunciado do trabalho que pressupõe que as entradas de dados serão corretas.

5. Análise Experimental

Para avaliar e comprovar a complexidade de tempo das operações realizadas na Árvore de Segmentação foram conduzidos dois experimentos de forma a plotar 2 gráficos de tempo de execução por caso de teste. O tempo de execução do código foi medido pelo uso da biblioteca “Chrono”, e os resultados foram armazenados num arquivo .txt que foi usado para gerar o gráfico por meio de um notebook em Python com uso das bibliotecas “Pandas” e “Matplotlib”.

Pelo uso do gerador de casos de teste providenciado pelo Moodle, primeiro foram criados 100 casos, variando o número de instantes de tempo (N) de 10.000 a 1.000.000, aumentando de 10.000 em 10.000, e deixando o número de operações realizadas (Q) constante como 10.000. Esse primeiro experimento é ilustrado na Figura 1 do Apêndice-Figuras. O gráfico 1 demonstrou uma complexidade de tempo próxima a O(log N).

Depois foram criados 100 casos, variando o número de operações realizadas (Q) de 10.000 a 1.000.000, aumentando de 10.000 em 10.000, e deixando o número de instantes de tempo (N) constante como 10.000. Esse segundo experimento é ilustrado na Figura 2 do Apêndice-Figuras. O gráfico 2 mostrou uma complexidade linear em relação a Q, ou seja, O(Q).

A operação de consulta e atualização da árvore de segmentação é O(log n), enquanto que no caso de se implementar um vetor para o mesmo problema teria complexidade O(n) na consulta e O(1) para atualização. Esta diferença na complexidade tem base nas expectativas teóricas, evidenciando a vantagem da árvore de segmentação em termos de eficiência computacional durante as operações de consulta. Contudo, é evidente destacar a dificuldade na visualização da curva logarítmica, devido a forma com que foi elaborado o gerador de testes, e pelas alterações e disparidades nas magnitudes de tempo entre os casos, tornando a representação visual no formato logarítmico menos evidente.

Assim, embora a árvore de segmentação ofereça uma performance mais eficiente em operações de consulta, a visualização adequada dessas vantagens pode ser comprometida por peculiaridades na natureza dos dados ou nas características específicas do problema em questão.

6. Conclusão

O desenvolvimento deste projeto proporcionou uma visão nos conceitos fundamentais relacionados à implementação de uma árvore de segmentação. Essa estrutura de dados permitiu a realização de consultas de forma mais eficiente e otimizada. O objetivo principal de implementar a árvore foi alcançado e, ao final, o código elaborado obteve êxito ao executar de forma eficiente a manipulação de matrizes e suas transformações lineares aplicadas a pontos em certos momentos escolhidos.

Por fim, o projeto proporcionou uma solução eficiente e implementável para o problema proposto, destacando a melhora de desempenho ao utilizar a SegTree.

Referências

CODEMARATHON. Árvore de Segmento. CodeMarathon, 2022. Disponível em: https://www.codemarathon.com.br/conteudos/estrutura-de-dados/arvore-de-segmento. Acesso em: 29 novembro 2023.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L. Introduction to Algorithms.

Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Apêndice - Instruções para compilação

Para executar o código do trabalho deve ser feito os seguintes passos. Na pasta raiz do projeto execute o comando ‘make all’ para realizar a compilação do programa. Para rodar o arquivo executável gerado acesse a pasta ‘bin’ do projeto que contém o executável ‘tp3.out’. Para apagar o executável na pasta ‘bin’ e os arquivos ‘.o’ gerados na compilação, digite ‘make clean’.

1. Compilação do programa:

   $ make all

2. Execução do programa:

   $ ./bin/tp3.out

3. Limpeza dos arquivos:

   $ make clean

Apêndice - Figuras

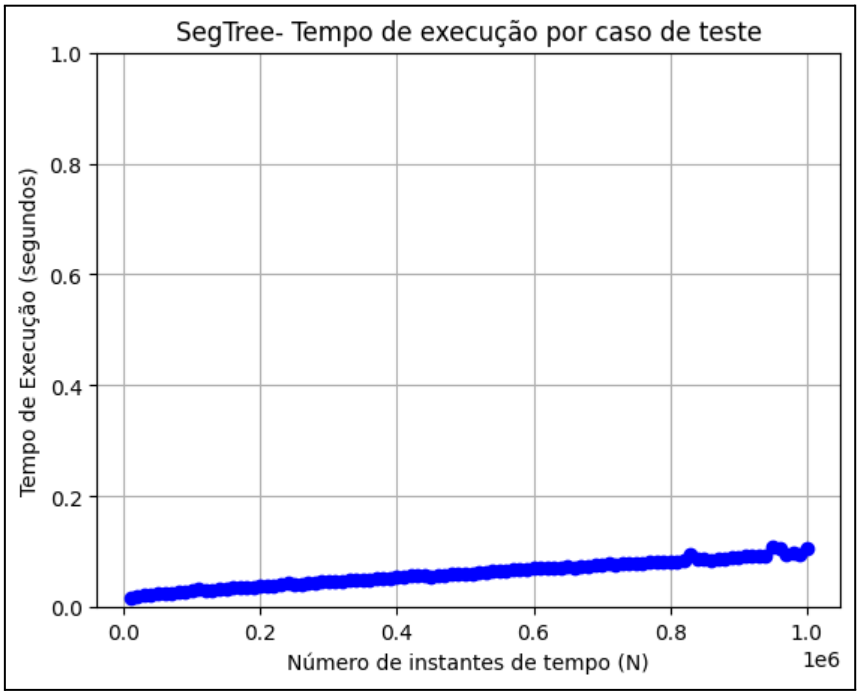

Figura 1

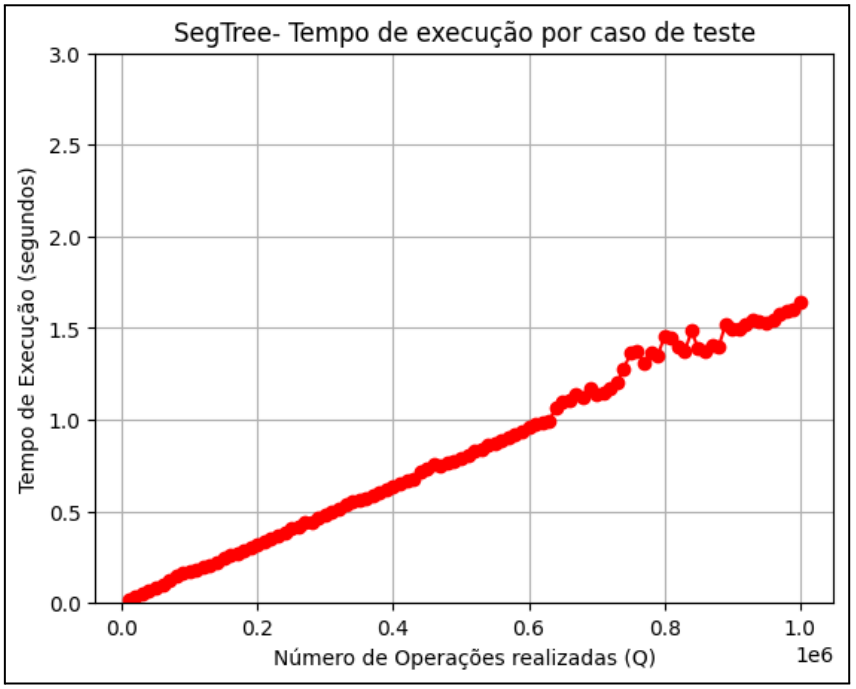

Figura 2