UNIVERSIDADE FEDERAL DE MINAS GERAIS

TRABALHO PRÁTICO 3

VITOR HUGO BRAGA VIEIRA, MATRÍCULA: 2018076706

PROFESSORES : WAGNER MEIRA JÚNIOR / EDER FERREIRA DE FIGUEIREDO

Belo Horizonte

2023

1 INTRODUÇÃO

Neste trabalho, abordaremos um problema na interseção entre matemática e ciência da computação, focando na aplicação de transformações lineares em um contexto lúdico. O cenário proposto se desenrola na fictícia cidade de Nlogônia, onde crianças se engajam em um jogo que consiste em aplicar transformações lineares a pontos em uma folha de papel. A mecânica do jogo é estruturada em torno de n instantes de tempo distintos, com cada instante associado a uma transformação linear específica. Os pontos em questão "nascem" em um instante de tempo t_0 e "morrem" em um instante td, que pode coincidir com t_0. Durante sua existência, os pontos são continuamente afetados pelas transformações lineares correspondentes a cada instante de tempo. O objetivo final do jogo é calcular a posição final do ponto, dadas a sua posição e tempo de nascimento, o tempo de morte e as transformações aplicadas.

Além disso, as regras do jogo permitem que as transformações sejam alteradas ao longo do tempo, mantendo a condição de que todas as transformações iniciais sejam identidades e que apenas transformações compostas por números positivos sejam utilizadas. Considerando que a sequência de transformações pode gerar coordenadas extensas, as crianças se interessam apenas pelos últimos 8 dígitos dessas coordenadas.

O jogo engloba duas operações principais:

1. Atualização: Nesta operação, as crianças escolhem um instante de tempo i e uma matriz B válida dentro das regras do jogo para substituir a matriz A_i correspondente.
2. Consulta: Aqui, as crianças selecionam os instantes de "nascimento" (t_0) e "morte" (t_d) de um ponto, além de suas coordenadas iniciais (x, y). O objetivo é determinar as coordenadas finais do ponto ao desaparecer.

Uma abordagem simples para esse problema seria armazenar cada transformação em um array indexado pelo tempo. A atualização é rápida, uma vez que o tamanho da matriz é independente do tamanho da entrada. No entanto, para a consulta, seria necessário multiplicar todas as matrizes entre os instantes a e b e, em seguida, aplicar o resultado ao ponto. Essa abordagem tem uma complexidade de tempo O(n) no pior caso, o que pode ser impraticável, considerando a impaciência natural das crianças.

Portanto, o foco principal deste trabalho é implementar uma estrutura de dados eficiente como a árvore de segmentos (SEG TREE), para otimizar a complexidade assintótica do tempo de consulta e, assim, tornar o jogo mais ágil e agradável para as crianças.

2 MÉTODO

O método empregado para resolver o problema, conforme mencionado na introdução, envolve o uso da árvore de segmentos. Esta estrutura de dados é inicializada com um tamanho pré-definido(lido na entrada), onde cada nó representa uma matriz 2x2. Inicialmente, todas as matrizes em cada nó da árvore são matrizes identidade, indicando que nenhuma transformação foi aplicada até o momento. Duas funções foram implementadas para interagir com esta árvore: uma para atualização e outra para consulta, cujos detalhes serão apresentados em breve.

ESTRUTURAS DE DADOS :

Foram utilizadas duas estruturas de dados: a árvore de segmentos (Segment Tree) e matrizes 2x2. No código, a árvore de segmentos é usada para armazenar e gerenciar as transformações lineares (representadas como matrizes 2x2) aplicadas aos pontos. Cada nó na árvore representa uma transformação linear ou a combinação de transformações de seus nós filhos.As matrizes 2x2 são utilizadas para representar as transformações lineares aplicadas aos pontos no jogo. Cada elemento da matriz influencia como um ponto no plano é transformado.

TIPOS ABSTRATOS DE DADOS :

Matrix2x2 (Matriz):

- O TAD Matrix2x2 representa uma matriz 2x2.
- Consiste em um array bidimensional mat[2][2], onde cada elemento é um *long long int* (precaução para lidar com possíveis valores altos nos casos de teste).
- As operações realizadas sobre este TAD incluem a inicialização de uma matriz identidade, multiplicação entre duas matrizes e impressão da matriz (para a depuração).

SegTree (Árvore de Segmentação):

- Este TAD representa uma árvore de segmentos, uma estrutura de dados que permite armazenar informações sobre intervalos ou segmentos de uma maneira que facilita atualizações e consultas eficientes.
- Consiste em dois campos principais:
  - int size: Armazena o tamanho da árvore de segmentos.
  - Matrix2x2 *tree: Um ponteiro para um array de Matrix2x2, que armazena as matrizes de transformação em cada nó da árvore.

- As operações realizadas sobre este TAD incluem a inicialização de uma árvore de segmentos com um tamanho definido pela entrada, atualização de um nó específico da árvore, consulta na árvore e liberação de memória alocada.

FUNÇÕES :

initMatrix(Matrix2x2 *matrix):

- Inicializa uma matriz 2x2 como matriz identidade.
- Configura os elementos da diagonal principal como 1 e os demais como 0.

multiplyMatrices(Matrix2x2 *result, const Matrix2x2 *a, const Matrix2x2 *b):

- Realiza a multiplicação de duas matrizes 2x2.
- Armazena o resultado da multiplicação no result.
- Implementa a operação matemática padrão de multiplicação de matrizes.
- Aplica mod 10^8 a cada elemento da matriz resultante para gerar os 8 últimos dígitos.

printMatrix(const Matrix2x2 *matrix) (Para Depuração):

- Imprime os elementos da matriz.
- Formata a saída para exibir uma matriz de forma legível.

initSegTree(SegTree *segtree, int size):

- Inicializa a árvore de segmentos com um tamanho lido na entrada.
- Aloca memória para a árvore e inicializa cada nó (matriz 2x2) como uma matriz identidade.

updateSegTree(SegTree *segtree, int node, int start, int end, int idx, const Matrix2x2 *val):

- Atualiza um nó específico na árvore de segmentos com uma nova matriz de transformação.
- A atualização é feita de forma recursiva, ajustando os nós pais conforme necessário.
- A função navega pela árvore para localizar o nó correto e atualiza os nós pais para refletir a nova transformação.

querySegTree(Matrix2x2 *result, SegTree *segtree, int node, int start, int end, int l, int r):

- Realiza uma consulta na árvore de segmentos para obter a transformação composta em um intervalo específico de tempo.
- Combina as transformações lineares dos nós correspondentes ao intervalo de consulta.
- Utiliza recursão para percorrer a árvore e calcular a matriz de transformação resultante.

3 ANÁLISE DE COMPLEXIDADE

initMatrix(Matrix2x2 *matrix): $O(1)$, pois apenas atribui valores fixos a quatro elementos da matriz.

multiplyMatrices(Matrix2x2 *result, const Matrix2x2 *a, const Matrix2x2 *b): $O(1)$, apesar de ter loops aninhados, pois o tamanho da matriz é constante (2x2), resultando em um número fixo de operações.

printMatrix(const Matrix2x2 *matrix) (Para Depuração): $O(1)$, pois imprime um número fixo de elementos da matriz.

initSegTree(SegTree *segtree, int size): $O(n)$, onde n é o número de nós na árvore. A árvore de segmentos geralmente tem cerca de 4 vezes o tamanho do intervalo que ela cobre, devido à sua natureza como uma árvore binária completa.

updateSegTree(SegTree *segtree, int node, int start, int end, int idx, const Matrix2x2 *val): $O(logn)$, onde n é o número de elementos cobertos pela árvore de segmentos. A função realiza atualizações de forma recursiva ao longo da altura da árvore, que é logarítmica em relação ao número de elementos.

querySegTree(Matrix2x2 *result, SegTree *segtree, int node, int start, int end, int l, int r): $O(logn)$ no pior caso. Semelhante à função de atualização, esta função percorre a árvore de cima para baixo, realizando operações ao longo da altura da árvore. Embora possa haver sobreposição nas chamadas recursivas, a profundidade máxima da recursão é limitada pelo logaritmo do número de elementos.

4 ESTRATÉGIAS DE ROBUSTEZ

O código apresenta algumas estratégias de robustez, focadas principalmente na eficiência e segurança da gestão de memória, além de medidas para assegurar a precisão e a integridade dos dados. Vamos examinar algumas delas:

Gestão Eficiente da Memória: Após alocações com `malloc`, verificações são realizadas para assegurar que a memória foi alocada com sucesso. Além disso, a função freeSegTree garante a liberação da memória alocada para a árvore de segmentos, evitando vazamentos de memória.

Inicialização Correta: As matrizes na árvore de segmentos são inicializadas como matrizes identidade na função initSegTree, assegurando que a árvore comece em um estado válido e consistente.

Checagens de Intervalo em Consultas: A função querySegTree verifica se o intervalo de consulta está fora do intervalo do nó atual, retornando a matriz identidade neste caso. Isso evita erros em consultas fora dos limites.

Atualizações Seguras: A função updateSegTree realiza atualizações de forma segura, considerando os limites do intervalo e atualizando os nós pais para manter a árvore consistente após cada mudança.

Evitar Overflow de Inteiros: Nas multiplyMatrices, o tipo long long é usado para armazenar os resultados mod 10^8, reduzindo o risco de overflow de inteiros.

Modularidade e Reutilização de Código: As funções forem bem definidas, promovendo a modularidade e facilitando a manutenção e a reutilização do código.

5 ANÁLISE EXPERIMENTAL

ESPECIFICAÇÕES DA MÁQUINA :

- Processador: 11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz, 2918 Mhz, 4 Core(s), 8 Logical Processor(s)
- Memória RAM: Installed Physical Memory (RAM) 16.0 GB
- Sistema Operacional: WSL - UBUNTU 20.04 LTS
- Linguagem de Programação: C
- Compilador: gcc

Foi desenvolvido, também, o método ineficiente proposto pela documentação. Neste método, as transformações lineares são armazenadas em um array de matrizes (Matrix2x2 transformations[n]), onde n é o tamanho lido na entrada. Com essa abordagem, o TAD SegTree foi removido do código, restando apenas as funções multiplyMatrices e initMatrix. O main foi ajustado para atender às especificações de entrada e saída e para operar diretamente sobre o array de matrizes.

```
Matrix2x2 transformations[n];
for (int i = 0; i < n; i++) {
    initMatrix(&transformations[i]);
}
```

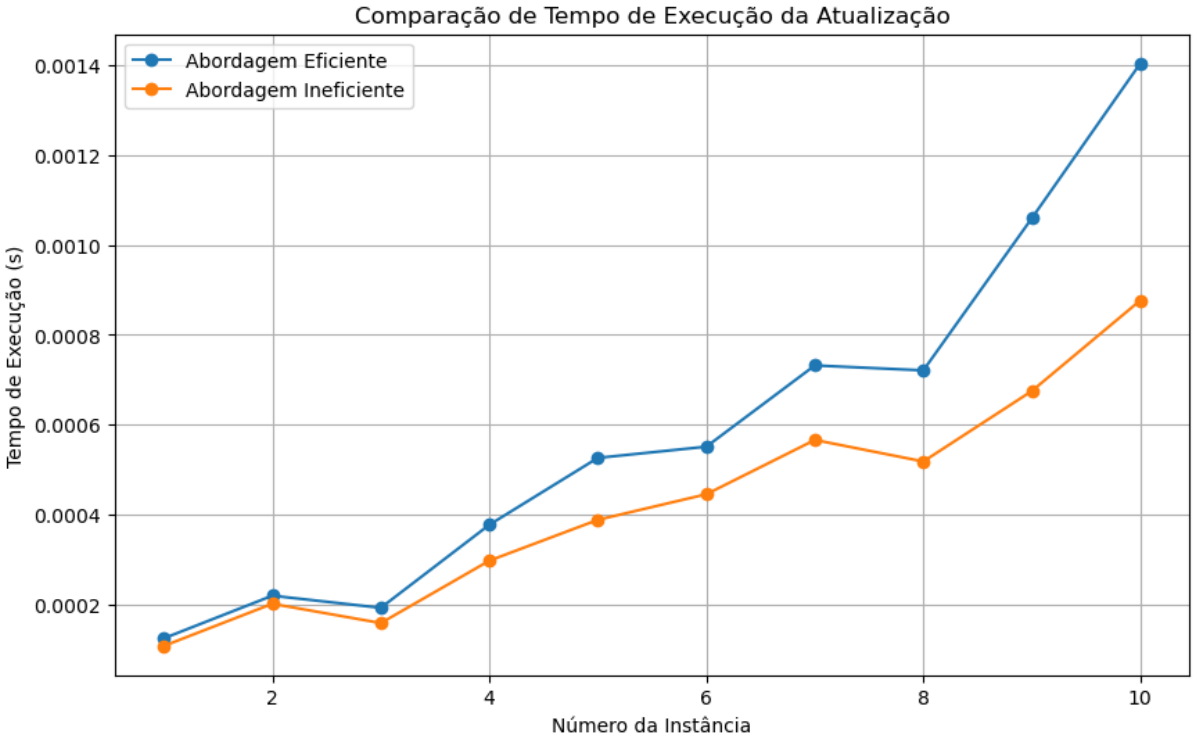
Após a implementação de ambos os métodos, utilizei o GeradorDeCasos, disponibilizado no ambiente Moodle, para criar um script que gerasse 10 instâncias, indexadas de 1 a 10. Em cada instância i, foi gerada uma entrada com $99 * i$ atualizações e $99 * i$ consultas. Por exemplo, na instância 1, a entrada contém 99 atualizações e 99 consultas, enquanto na instância 2, são 198 atualizações e 198 consultas, seguindo essa progressão.

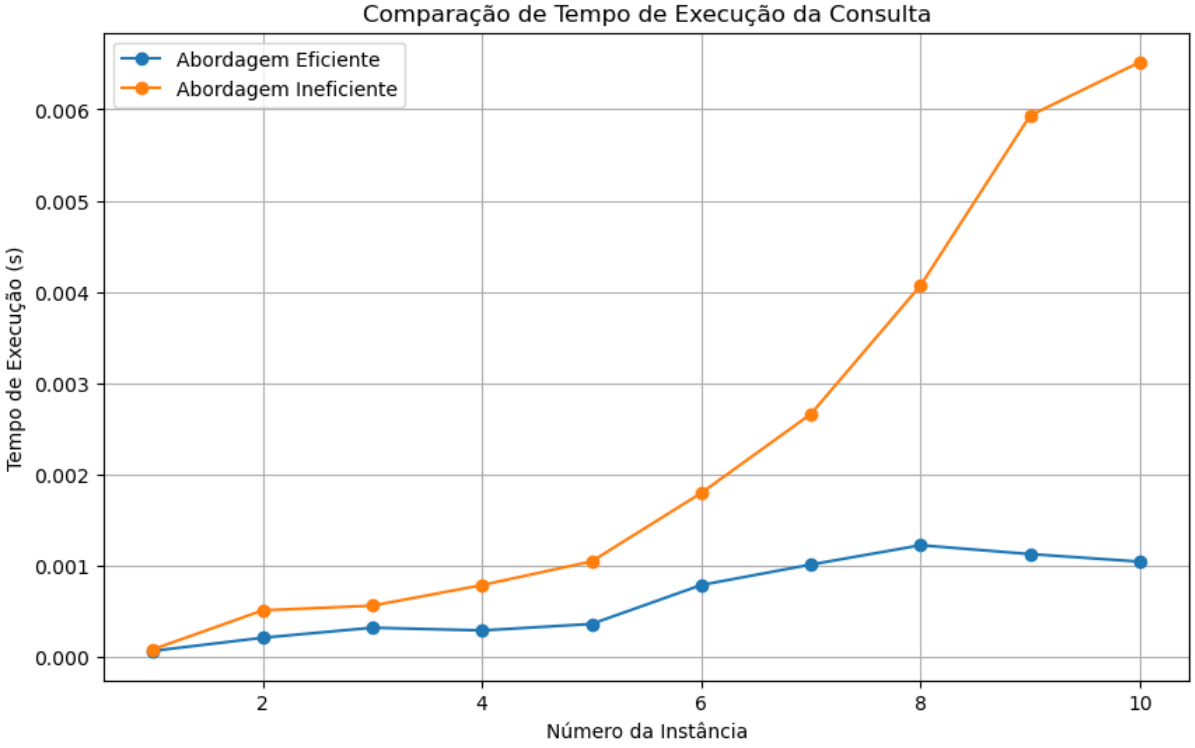
```
int main(){
    ofstream outputFile("output.txt"); // Criação de um objeto de arquivo de saída

    int baseSize = 99; // Tamanho base para as entradas
    for (int i = 1; i <= 10; i++) {
        int currentSize = baseSize * i; // Incrementa o tamanho em i*baseSize
        string instance = InstanceGenerator::GetInstance(currentSize, currentSize);
        outputFile << "Instance " << i << ":\n" << instance << "\n";
    }

    outputFile.close(); // Fechamento do arquivo de saída
    return 0;
}
```

Utilizando a biblioteca time.h, os códigos de ambas as abordagens (a ineficiente e a da árvore de segmentos) foram adaptados para medir o tempo de execução das 10 instâncias geradas pelo GeradorDeCasos. As comparações de desempenho foram realizadas para ambas as operações, de atualização e consulta. Os gráficos resultantes dessas comparações são apresentados a seguir.


Comparação de Tempo de Execução da Atualização
Abordagem Eficiente
Abordagem Ineficiente
Tempo de Execução (s)
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
2
4
6
8
10
Número da Instância


Comparação de Tempo de Execução da Consulta
Abordagem Eficiente
Abordagem Ineficiente
Tempo de Execução (s)
0.006
0.005
0.004
0.003
0.002
0.001
0.000
2
4
6
8
10
Número da Instância

Na operação de Atualização, observa-se que a abordagem utilizando array é marginalmente mais rápida em comparação com a abordagem que emprega a árvore de segmentos. Isso é coerente com a análise teórica de complexidade de tempo, onde a atualização das matrizes em um array apresenta uma complexidade O(1), já que se trata de uma simples substituição de elementos em um índice específico. Em contraste, na árvore de segmentos, a complexidade de uma operação de atualização é O(log n) devido à natureza logarítmica da estrutura, que requer percorrer o caminho desde a raiz até o nó específico e, em seguida, atualizar os nós pais. Contudo, a vantagem da árvore de segmentos se revela significativamente na operação de Consulta. Enquanto a abordagem ineficiente, que usa um array, possui uma complexidade de tempo linear (O(n)) no pior caso, a árvore de segmentos se destaca com uma complexidade de tempo O(log n).

Testes de verificação de Memory Leak usando o Valgrind foram realizados para instâncias aleatórias, não há Memory Leak no programa, o uso das estratégias de robustez foi respeitado. Segue um exemplo para uma instância qualquer :

```
==1905==
==1905== HEAP SUMMARY:
==1905==     in use at exit: 0 bytes in 0 blocks
==1905==   total heap usage: 3 allocs, 3 frees, 14,720 bytes allocated
==1905==
==1905== All heap blocks were freed -- no leaks are possible
==1905==
==1905== For lists of detected and suppressed errors, rerun with: -s
==1905== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

6 CONCLUSÃO

Ao longo deste trabalho, visitamos conceitos importantes no cruzamento entre a matemática e a ciência da computação, com foco especial na aplicação de transformações lineares e na otimização de algoritmos usando árvores de segmentos. O problema proposto, situado no contexto lúdico da cidade fictícia de Nlogônia, ofereceu uma plataforma prática para compreender e aplicar esses conceitos. No contexto de transformações lineares, aprofundamos nosso entendimento sobre como elas podem ser representadas e manipuladas usando matrizes 2x2. Compreendemos a importância da inicialização adequada e da multiplicação de matrizes, essencial para transformar pontos em um plano. Além disso, aprendemos como árvore de segmentos pode otimizar operações que envolvem intervalos, particularmente útil para consultas e atualizações frequentes e eficientes. O contraste entre as abordagens ineficiente (uso direto de arrays) e eficiente (uso da árvore de segmentos) ilustrou a importância da complexidade de algoritmos. Entendemos como a escolha de uma estrutura de dados adequada pode impactar significativamente o desempenho do algoritmo, especialmente em termos de complexidade temporal.

7 REFERÊNCIAS BIBLIOGRÁFICAS

Figueiredo, E., Meira Jr., W. (2023). *Slides virtuais da disciplina de estruturas de dados*. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais.