Trabalho Prático 3

Jogo de Transformação Linear

Arthur Araújo Rabelo
2022091552

1. Introdução

Este trabalho se baseia em uma brincadeira fictícia onde crianças desenham pontos em uma folha de papel e estes vão sofrendo tranformações lineares durante um limitado número de instantes de tempo. O objetivo desse jogo é determinar as coordenadas do ponto após todas as modificações até um certo instante de tempo. É preciso lembrar que essas transformações podem ser atualizadas durante a brincadeira e que, no instante inicial, os pontos terão uma transformação identidade, ou seja, que não desloca suas coordenadas.

Uma transformação linear é um tipo particular de função entre dois espaços vetoriais. Considerando que os pontos desenhados são vetores, as transformações lineares em cada um deles será feita por matrizes 2x2.

Dito isso, o problema central a ser resolvido neste trabalho é o de guardar para cada intervalo contido nos instantes de tempo as transformações que o ponto irá sofrer. A brincadeira possui duas operações: a atualização de uma transformação e a consulta, que diz em quais coordenadas o ponto “morreu” dado um certo intervalo de tempo.

A solução foi implementar uma árvore de segmento para armazenar as transformações em vários e diferentes intervalos de tempo. Esta é uma estrutura de dados que torna as operações de consulta e atualização um tanto mais rápidas.

2. Método

2.1 Estruturas de Dados

A estrutura de dados utilizada foi a árvore de segmentação citada acima. Essa estrutura é comumente usada para problemas como este, que precisam de armazenar e retornar informações de determinados segmentos (intervalos). A

sua classe é a segTree e para ela foram implementadas as funções e classes abaixo.

2.2 Classes

Matrix:

Para realizar a transformação linear dos pontos, foi preciso criar a classe Matrix para representar as matrizes. A matriz possui um ponteiro de ponteiros, cada um deles com 2 endereços, já que todas as matrizes serão 2x2, e também dois inteiros que armazenam o número de linhas e de colunas (elementos até desnecessários nesse contexto). Além do construtor, que inicializa a matriz como uma identidade, esta classe também conta com as funções Multiply, que multiplica duas matrizes; SetAsNull, que zera todos os elementos da matriz (necessária para a multiplicação), as sobrecargas de operadores para printar (<<) e ler (>>) da entrada padrão, e do operador igual (=). Dentro da classe, há um struct para o tipo ponto, que possui dois inteiros para representar as suas coordenadas. Esse struct é usado e retorna na função LinearTransformation, que multiplica a matriz por um ponto e retorna o resultado (outro ponto).

Node:

Esta classe foi implementada para representar os elementos que irão compor a árvore de segmentação. Cada node possui uma matriz, que é a matriz de transformação para o intervalo que o próprio nó representa. Dessa forma, os nós folha terão associada a si a matriz do instante que corresponde a ele. Além da matriz, há uma flag booleana que, caso seja verdadeira, sinaliza que a matriz daquele nó sofreu alguma atualização, ou seja, deixou de ser identidade.

SegTree:

A classe segTree possui um ponteiro do tipo Node que serve para armazenar os nós da árvore (funciona como um heap). Além disso, possui uma matriz identidade e um inteiro que armazena o número de instantes do problema.

2.3 Funções

A classe da árvore de segmentos possui as seguintes funções: o construtor, que aloca 4 vezes o número de instantes para os nós da árvore (convenção quando se implementa esse tipo de estrutura de dados); o destrutor, que deleta o vetor alocado; MultiplyMatrices, que chama a função de multiplicação de matrizes; build, que atribui aos segmentos as suas matrizes de transformação iniciais; query, que realiza uma consulta e retorna a matriz de transformação para um determinado intervalo e, por fim, UpdateMatrix, que atualiza o valor de uma matriz para um determinado instante e para todos os segmentos que dependem do valor dele. Todas essas funções funcionam com uma lógica recursiva e com três parâmetros em comum: um inteiro p que armazena a posição atual na árvore (ou heap) e dois, l e r, que armazenam os valores de início e fim do intervalo. Para caminhar pelos intervalos da árvore, começamos da raiz e vamos atualizando os valores de início e fim para dividir o caminho de modo a passar por todos os intervalos desde a raiz até as folhas. A posição acompanha o caminhamento: ela é sempre atualizada com a mesma lógica dos sucessores à esquerda e à direita de um nó de um heap.

Build passa por todos os intervalos recursivamente até chegar na condição de parada (nó folha), atribuindo aos intervalos sua matriz inicial.

Query recebe dois parâmetros a mais: o intervalo do qual se quer extrair a consulta; sendo assim, ela possui condições de parada diferentes: se o intervalo do nó atual não está contido no que foi passado como parâmetro, ela retorna a matriz identidade. Se o intervalo está contido, ela testa se a sua matriz já foi

atualizada: se sim, retorna a matriz daquela posição; se não, retorna identidade. Após chegar nas folhas, a função vai desempilhando as chamadas recursivas e multiplicando as matrizes correspondentes aos intervalos procurados e retorna a matriz resultante.

UpdateMatrix recebe o instante a ser atualizado e a nova matriz. É importante ressaltar que a posição que será atualizada não necessariamente é igual ao valor de i. Essa posição é a que representa o nó folha que corresponde ao instante i na árvore de segmentação. A função realiza a mesma lógica das outras duas, com a diferença que atualiza o valor desse nó folha e, quando acabam as chamadas recursivas, atualiza todo o restante da árvore com as multiplicações resultantes da operação de atualização.

3. Análise de Complexidade

3.1 Complexidade de Tempo

Um elemento importante para a análise de tempo deste código é o fato de que a função build é absolutamente desnecessária para este problema. Todas as matrizes já são inicializadas como identidade. Portanto, quando o construtor da árvore de segmentos aloca os nós, todos já tem em si as suas matrizes iniciais. Sendo assim, os únicos elementos do código que influenciam no tempo são as funções de consulta e atualização.

As duas possuem complexidade O(log n), já que caminham em uma árvore balanceada, sendo o pior caso um instante correspondente a algum nó folha, presente no nível ‘log n’ da árvore.

3.2 Complexidade de Espaço

A única alocação de espaço que há no código é para os nós da árvore de segmentos. As pilhas recursivas de query e updateMatrix, ainda que a memória seja estática, requerem espaço da ordem de log n, visto que estão relacionadas a altura da árvore. Assim, sendo a quantidade de nós o quádruplo da quantidade de instantes, tem-se que a complexidade de espaço das pilhas recursivas é dominada assintoticamente pela alocação dinâmica dos nós. Portanto, a complexidade de espaço do código é O(n).

4. Estratégias de Robustez

As medidas de programação defensiva para este problema se baseiam em algumas exceções e na própria lógica de implementação do código.

Primeiramente, todas as matrizes criadas para a árvore de segmentação são de dimensão 2. Dessa forma, não é possível haver exceções nas funções que envolvem a multiplicação de matrizes e a transformação linear. Ademais, o construtor da matriz já a inicializa como identidade, não permitindo que haja algum erro de valor não inicializado.

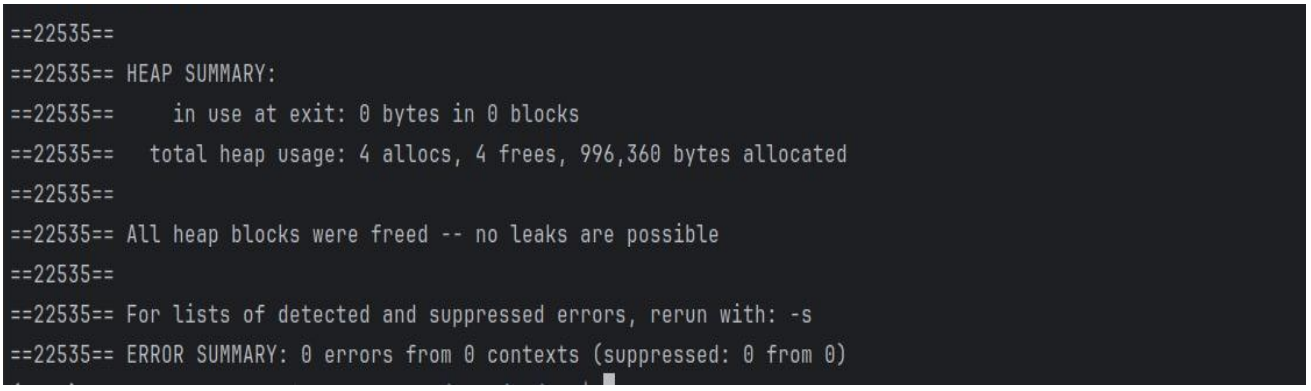
Na parte do código onde há a interação com um usuário externo, algumas medidas são tomadas a fim de prevenir que o programa tenha um comportamento inesperado, podendo haver também - nesse caso - leaks de memória. Na operação de consulta, o usuário não pode digitar um intervalo inválido, ou seja, menor que 0 e maior que o número de instantes. Esse teste é feito antes que a função query seja chamada; o código só seguirá se o intervalo for válido. A mesma coisa é feita na função de atualização: se o instante não estiver dentro de um intervalo válido, existe uma condição que barra o prosseguimento do código até que a entrada fornecida seja correta. Além disso, se alguma operação diferente de ‘q’ e ‘u’ for digitada, o código não continua até que operações válidas sejam fornecidas. Por fim, meros avisos são impressos na

tela quando ou o número de instantes ou o número de operações forem 0. Isso é um comportamente inesperado, mas não causa nenhum problema no código em si.

5. Análise Experimental

A primeira tentativa de fazer este programa alocava memória para o ponteiro duplo da matriz para cada nó e sempre retornava outros ponteiros nas funções de multiplicação, build, updateMatrix e query. O problema dessa implementação é que muita memória extra é utilizada, havendo até leaks na função query, os quais não consegui resolver.

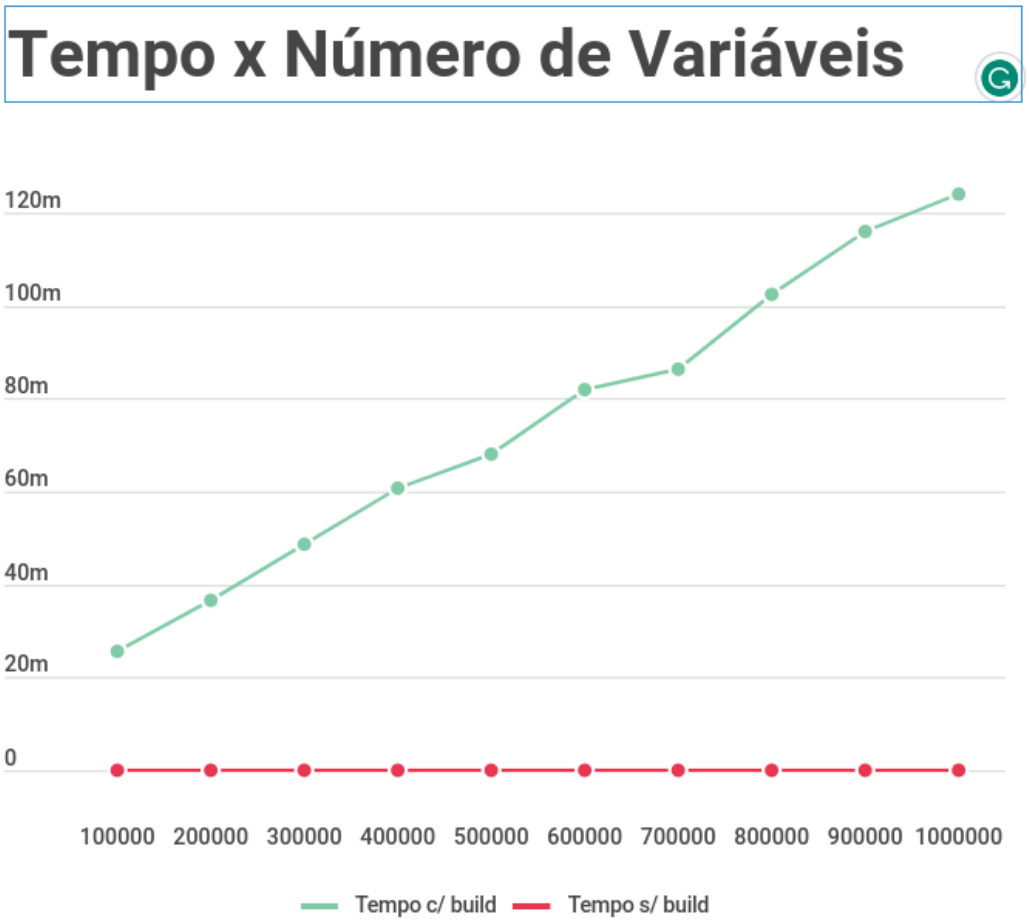
A solução foi parar de usar os ponteiros e implementar as matrizes estaticamente. Dessa forma, a única alocação é para os nós da árvore de segmentos. Além de acabar com as perdas de memória, os bytes alocados diminuem drasticamente. Depois dessa melhoria, assim ficou a saída do valgrind, diferindo apenas - para cada teste - no número de bytes alocados.



Para uma análise experimental mais aprofundada, foram feitos dois testes envolvendo medições de tempo: um com o uso da função build e outro sem o uso dela. O número de variáveis foram 100.000 até 1.000.000. As operações

feitas foram uma atualização no instante 0 e uma consulta no intervalo [0,0], de modo a percorrer toda a altura da árvore (pior caso). A função build, por construir todos os nós da árvore, é O(n).

Assim ficou o gráfico:



Podemos ver que o programa, com a função build, tem tempo linear. Sem ela, podemos ver que o tempo diminui bastante, passando a se comportar apenas como log n. As medições do segundo caso (sem build) ficaram também mais complexas e imprecisas por causa do valor baixo.

Assim ficou o gráfico só com as medidas de updateMatrix e query:

Veja que o tempo ficou na ordem de microssegundos.

6. Conclusão

Neste trabalho, foram feitas a construção e a adaptação de uma árvore de segmentos, um tipo de estrutura muito útil para resolver diversos problemas, especialmente os que precisam de informações relativas a intervalos.

O principal aprendizado deste trabalho se deu na tentativa e erro de fazer o tipo matriz alocando memória para cada um de seus elementos, como foi dito acima. Percebi que fiz o que é chamado de ‘Pointless use of pointers’, ou seja, uso sem necessidade de ponteiros e alocação de memória. Além disso, foi possível reforçar aprendizados relacionados a análise de algoritmos recursivos e Teorema Mestre. Além disso, pela análise experimental, foi possivel verificar que o código se comporta muito melhor em termos de tempo sem a função build.

7. Bibliografia

MONTEIRO, Bruno. Aula 9 - SegTree. YouTube, 23/01/2021. Disponível em: https://www.youtube.com/watch?v=OW_nQN-UQhA&t=2501s