Hierarquia de Classes – MATRIZES

Relatório de Desenvolvimento: Calculadora Matricial

Este projeto desenvolveu uma calculadora matricial em Python, utilizando Programação Orientada a Objetos para gerenciar e otimizar operações com diferentes tipos de matrizes.

1. Estruturas de Dados Utilizadas

- **Matriz Geral (GeneralMatrix)**: Usa uma **lista de listas** para armazenar todos os elementos, O(m×n) de espaço.
- **Matriz Diagonal (DiagonalMatrix)**: Armazena apenas os elementos da diagonal principal em uma **lista simples**, usando O(n) de espaço. Zeros fora da diagonal são implícitos.
- **Matrizes Triangulares (LowerTriangularMatrix, UpperTriangularMatrix)**: Armazenam apenas os elementos relevantes em uma **lista de listas aninhadas** com tamanhos variados, resultando em O(n2) de espaço (mas armazenam menos da metade dos elementos totais).

2. Divisão de Módulos (Classes)

O código é dividido em uma **hierarquia de classes** para modularidade e otimização:

- **Matrix**: É a classe base que define as operações e comportamentos comuns a todas as matrizes.
- **GeneralMatrix**: Implementa as características de uma matriz comum.
- **DiagonalMatrix**: Classe especializada para matrizes diagonais, com otimizações de armazenamento e operações.

- **LowerTriangularMatrix**: Classe especializada para matrizes triangulares inferiores, também com otimizações.
- **UpperTriangularMatrix**: Classe especializada para matrizes triangulares superiores, com otimizações similares.

3. Descrição das Rotinas e Funções

- **Métodos das Classes (ex: __add__, __mul__, transpose, determinant, trace)**: São as operações matriciais. Muitos são sobrecarregados para que cada tipo de matriz (diagonal, triangular) possa realizar a operação de forma otimizada e retornar o tipo correto quando possível. Caso contrário, a operação genérica da classe base é usada, geralmente resultando em uma GeneralMatrix.
- **Funções Auxiliares (ex: criar_matriz, selecionar_matriz, salvar_matrizes)**: Controlam a interação com o usuário, a criação e o gerenciamento da lista de matrizes, além de funcionalidades de salvar e carregar dados em arquivos.
- **main()**: É a função principal que executa o menu interativo da calculadora.

4. Complexidade de Tempo e Espaço

A **complexidade de espaço** (memória) varia conforme o armazenamento otimizado de cada matriz.

- GeneralMatrix: O(m×n) para armazenamento. Para uma matriz quadrada n×n, $O(n^2)$.
- DiagonalMatrix: O(n) para armazenamento da diagonal.

LowerTriangularMatrix / UpperTriangularMatrix: $O(n^2/2)$ para armazenamento, que é $O(n^2)$.

A complexidade de tempo (velocidade das operações) também varia:

- **Acesso a Elementos**: Rápido, O(1).
- **Soma/Subtração/Multiplicação por Escalar**: O(n) para matrizes diagonais otimizadas; O(n^2) para triangulares otimizadas; O(m×n) para matrizes gerais.
- **Multiplicação Matricial**: O(n^3) para matrizes n×n.
- **Determinante**: O(n) para matrizes diagonais/triangulares; O(n!) para matrizes gerais (muito lento para grandes matrizes).
- **Traço**: O(n).

5. Problemas e observações

Problema: Um desafio central foi garantir que, ao realizar operações (como soma ou multiplicação por escalar) entre matrizes do mesmo tipo otimizado (ex: DiagonalMatrix + DiagonalMatrix), o resultado fosse também do mesmo tipo otimizado, e não uma GeneralMatrix.

- Observação: A solução foi sobrescrever os métodos de operador (__add__, __sub__, __mul__) em cada classe especializada. Eles primeiro verificam se o operando é do mesmo tipo. Se for, realizam a operação de forma otimizada e retornam uma nova instância do tipo especializado. Caso contrário, eles delegam para o método da classe base (super()), que por padrão retorna uma GeneralMatrix.

Implementação de __setitem__ para Matrizes Otimizadas

- Problema: Matrizes como DiagonalMatrix ou as triangulares armazenam apenas os elementos "não-zero" relevantes para economizar memória. O desafio foi permitir que o usuário definisse um elemento $A[i, j] = valor$ sem comprometer a estrutura otimizada.

- Observação: A lógica do __setitem__ nessas classes exige que, se o usuário tentar atribuir um valor diferente de zero a uma posição que deveria ser zero (fora da diagonal em uma matriz diagonal, ou acima/abaixo da diagonal em uma triangular), um erro (ValueError) seja lançado. Isso garante que a matriz mantenha sua propriedade de tipo específico.

Complexidade do Determinante para Matriz Geral

- Problema: O cálculo do determinante para GeneralMatrix foi implementado usando o método de cofatores, que é didaticamente simples, mas computacionalmente ineficiente (O(n!)). Para matrizes de grande dimensão, essa operação seria inviável.
- Observação: Para uma aplicação real, seria necessário implementar algoritmos mais eficientes, como eliminação gaussiana ou decomposição LU, que possuem complexidade de O(n3). A escolha atual foi feita para manter a simplicidade didática do projeto.

Multiplicação Matricial entre Tipos Especializados

- Problema: O produto de duas matrizes especializadas (ex: LowerTriangularMatrix × LowerTriangularMatrix) nem sempre resulta em uma matriz do mesmo tipo especializado (pode ser uma GeneralMatrix). Otimizar essa operação para retornar um tipo específico em todos os casos se torna complexo.
- Observação: Optou-se por deixar a multiplicação matricial (A * B) sempre retornar uma GeneralMatrix. Embora isso perca algumas otimizações potenciais para casos específicos, simplifica bastante a lógica e garante a correção do resultado.

Persistência de Dados e Reconstrução de Objetos

- Problema: Salvar e carregar matrizes de um arquivo requer um formato que preserve tanto os dados quanto o tipo da matriz, para que as classes corretas possam ser recriadas.
- Observação: Foi definido um formato de arquivo simples (TYPE|rows|cols|data_elements). A função carregar_matrizes analisa esse formato e instancia dinamicamente a classe correta (GeneralMatrix, DiagonalMatrix, etc.) com os dados recuperados, garantindo que as otimizações de memória e comportamento sejam mantidas após o carregamento.

6. Conclusão

Esse projeto de calculadora matricial em Python demonstrou a eficácia da Programação Orientada a Objetos na criação de um sistema flexível e eficiente. Usando uma hierarquia de classes, conseguimos otimizar o uso de memória para diversos tipos de matrizes, sejam elas gerais, diagonais ou triangulares, e, com isso, melhorar o desempenho das operações mais comuns. Os desafios que surgiram, como garantir que as matrizes resultantes mantivessem seus tipos específicos e lidar com a complexidade de certos algoritmos, foram superados com soluções que priorizaram a funcionalidade e a modularidade do código.

Alunos:
Davidson Diógenes Vasconcelos da Silva Santos

DRE: 123531495

Mônica de Sousa Amaral

DRE: 119160444

Naya da Silva Nascimento

DRE: 119160428