Matriz Esparsa

Sumário

Apresentação:

Este documento objetiva analisar uma matriz esparsa implementada em linguagem C, utilizando uma estrutura de dados listas ortogonais com nós cabeçalhos. Primeiramente, destina-se a descrever o funcionamento da arquitetura da solução escolhida. Em seguida, propõe a refletir sobre os desafios e dificuldades enfrentadas na implementação de uma solução com esta complexidade.

Integrantes:

-Gabriel Lima Scheffler

-Lucas Scheffer Hundsdorfer

-Gustavo Mazur Romagnoli

Arquitetura e Funcionamento da Implementação

A abordagem utilizada para representar a matriz se afasta do tradicional array bidimensional, optando por uma estrutura dinâmica muito mais eficiente em termos de memória para matrizes com poucos elementos não nulos. A base dessa arquitetura é uma grade de listas ligadas. No centro de tudo, há um nó principal (cabeca) que serve como ponto de partida. Dele, partem duas listas circulares de nós cabeçalhos: uma horizontal para as colunas e uma vertical para as linhas. Os valores não nulos da matriz são armazenados em nós de dados, cada um pertencendo simultaneamente a uma lista de linha e a uma lista de coluna, criando uma teia de conexões.

A função começa com a matriz pela função matriz_criar, que aloca a estrutura principal e deixa montar o "esqueleto" da matriz a um função de auxílio, criarNosCabecaCompletos. Dessa forma, essa função define o nó cabeça e, logo após, monta as listas circulares de cabeçalos de linha e coluna, já deixando a estrutura pronta a receber os dados.

A seguir, a operação mais crítica e complicada é, sem dúvida, a executada pela função matriz_definir_valor. Nesta função, temos de lidar com a inserção, atualização e remoção de elementos. Para detetar a posição, este algoritmo fará primeiro o papel de procura dos nós predecessores na lista da linha e na lista da coluna. Tendo encontrado os predecessores, o programa verifica se existe já um nó na posição observada. Se isso acontecer, e o novo valor é 0, o nó observado é retirado da teia de ponteiros. Mais concretamente, tal nó será "saltado" por os seus predecessores, tornando-se "invisível" para eles, e a sua memória é libertada. Caso contrário, se o novo valor for igual a 0, não acontece nada. Se não existir nenhum nó na dita posição e o valor não for 0, um novo nó é alocado e é cuidadosamente "costurado" em ambas as listas (de linha e de coluna), tornando-se vizinho dos dois.

Por outro lado, a obtenção de um valor, realizada por matriz_obter_valor, é mais simples. A essa função, é passada a linha desejada. Ela então percorre sua lista de elementos da forma horizontal. Quando um nó da coluna correspondente é encontrado, seu valor é retornado. Se não, seu valor é assumido como zero. Já a visualização, é feita pela função matriz_mostrar_na_tela, simula a impressão de uma matriz densa. Essa função percorre cada célula da matriz, e para , ela verifica se o próximo elemento não-nulo da lista da linha i é da coluna j. Fazendo isso, imprime o valor do nó. Tendo chegado aqui, o próxima elemento não-nulo da linha i é selecionado, e o processo é repetido. Se a verificação falhar, imprime-se zero porque um valor nulo é assumido, já que o nó não existente é o indicador disso.

Por fim, para evitar vazamentos de memória, a função matrix_delete_complete realiza uma destruição controlada de toda a estrutura. A ordem neste caso é muito importante: primeiro, ele percorre todas as linhas e libera memória para cada nó de dados. Em seguida, ele libera nós tête e nós de terminal. Somente depois que toda a teia é desfeita o nó principal e a própria estrutura são liberados. O último passo de atribuir NULL ao ponteiro original é uma prática prudente, uma vez que protege contra a utilização acidental de memória liberada.

Reflexão sobre os Desafios do Desenvolvimento

Tendo-se desenvolvido um sistema como este, a praticidade enfrenta obstáculos que não obedeçam à sintaxe de uma linguagem de programação. O primeiro desses desafios é puramente conceitual. Abandonar a estrutura de cima para baixo de um bloco de memória contíguo em favor de uma grade de ponteiros é ultrapassar uma mentalidade longa e encravada. Ficar parado, olhar para as listas e os caminhos que emanam, enquanto uma garoa tranquila distingue a superfície de casca de laranja, enquanto os nós cabeçalhos permeiam e consideram laços em suas ponderosas teias é a primeira barreira que se atravessa.

Uma vez superada a barreira conceitual, surge o desafio mais concreto do projeto: o gerenciamento de ponteiros. A lógica de inserção e remoção se torna extremamente delicada. Um único ponteiro incorretamente atribuído pode corromper toda a matriz, resultando em loops infinitos ao atravessar ou em falhas de segmentação. A depuração, nesse caso, se torna quase impossível, pois não é possível simplesmente ‘‘imprimir o array’’ para encontrar o erro. Pelo contrário, é necessário acompanhar os rastreamentos dos ponteiros, agindo muito frequentemente com a ajuda de um depurador ou de funções de diagnóstico customizadas.

Com isso, está associada a gestão da memória. Cada nó não nulo é alocado dinamicamente e o programador é encarregado de liberá-los quando não são mais necessários. Assim, o risco de vazamento de memória é constante e a complexidade de \texttt{destroy()} atesta a disciplina necessária para garantir que todos os bytes alocados sejam efetivamente liberados. Por fim, a confiabilidade do código é determinada pelo manuseio cuidadoso de vários casos especiais \textit{(edge cases)}, como uma matriz vazia, a inserção ou remoção do único nó de uma linha/coluna e as bordas da grade.

Em suma, a construção desta matriz esparsa representa um excelente exercício de programação avançada, exigindo não apenas conhecimento técnico, mas também um raciocínio abstrato e uma atenção meticulosa aos detalhes para garantir a integridade, eficiência e estabilidade da estrutura de dados.