Atividade 1: Representação de Grafos em C++

Contexto: Grafos

Grafos são estruturas de dados não lineares que representam um conjunto de objetos (chamados **vértices** ou **nós**) onde alguns pares de objetos estão "relacionados" por meio de **arestas**.

Existem duas formas principais de representar um grafo na memória:

1. **Matriz de Adjacência:** Uma matriz quadrada onde cada célula (i,j) indica se existe uma aresta entre o vértice i e o vértice j.
2. **Lista de Adjacência:** Um array (ou vetor) de listas, onde cada posição do array representa um vértice, e a lista associada a essa posição contém todos os vértices adjacentes a ele.

Nesta atividade, vamos explorar a implementação dessas representações utilizando conceitos básicos de C++.

Parte 1: Exemplo de Implementação com Lista de Adjacência

Para que vocês entendam a lógica de representação de grafos, vamos analisar um exemplo de como implementar um grafo usando **Lista de Adjacência** em C++. Prestem atenção aos comentários no código para entender cada parte da implementação.

Exemplo de Grafo (para o código):

Um grafo com 5 vértices (0, 1, 2, 3, 4) e as seguintes arestas:

- 0 - 1
- 0 - 4
- 1 - 2
- 1 - 3
- 1 - 4
- 2 - 3
- 3 - 4

Código-fonte: grafo_lista_adjacencia.cpp+

```
#include <iostream> // Para entrada e saída (cin, cout)
#include <cstdlib>  // Para malloc e free

// Definindo um número máximo de vértices para o nosso grafo
// Isso é uma restrição para simplificar e usar arrays estáticos.
#define MAX_VERTICES 5

// Uma forma simples de representar um nó em uma lista encadeada
```

```
// Usamos struct e ponteiros, estilo C.
struct NoLista {
    int destino;
    struct NoLista* proximo; // Ponteiro para o próximo nó na lista
};

// Estrutura para representar o grafo usando lista de adjacência
struct Grafo {
     struct NoLista* adjacencias[MAX_VERTICES]; // Um array de ponteiros para as
"cabeças" das listas
    int numVertices; // Número real de vértices no grafo
};

// Função para criar um novo nó de lista
struct NoLista* criarNovoNo(int destino) {
     struct NoLista* novoNo = (struct NoLista*) malloc(sizeof(struct NoLista)); // Aloca
memória
    if (novoNo == NULL) { // Verifica se a alocação foi bem-sucedida
        std::cout << "Erro de alocacao de memoria!" << std::endl;
        exit(1); // Encerra o programa em caso de erro
    }
    novoNo->destino = destino;
    novoNo->proximo = NULL;
    return novoNo;
}

// Função para criar um grafo com um número específico de vértices
struct Grafo* criarGrafo(int V) {
     struct Grafo* grafo = (struct Grafo*) malloc(sizeof(struct Grafo)); // Aloca memória
para a estrutura Grafo
    if (grafo == NULL) {
        std::cout << "Erro de alocacao de memoria!" << std::endl;
        exit(1);
    }
    grafo->numVertices = V;

    // Inicializa todas as listas de adjacência como vazias (NULL)
    for (int i = 0; i < V; ++i) {
        grafo->adjacencias[i] = NULL;
    }
    return grafo;
}

// Função para adicionar uma aresta a um grafo não-direcionado
void adicionarAresta(struct Grafo* grafo, int origem, int destino) {
    // Adiciona uma aresta da origem para o destino
    struct NoLista* novoNo = criarNovoNo(destino);
      novoNo->proximo = grafo->adjacencias[origem]; // O novo nó aponta para o
que era o primeiro
     grafo->adjacencias[origem] = novoNo;           // E o novo nó se torna o primeiro
da lista
```

```
    // Para grafos não-direcionados, também adicionamos a aresta do destino para a origem
    novoNo = criarNovoNo(origem);
    novoNo->proximo = grafo->adjacencias[destino];
    grafo->adjacencias[destino] = novoNo;
}

// Função para imprimir a representação do grafo
void imprimirGrafo(struct Grafo* grafo) {
    std::cout << "Representacao do Grafo (Lista de Adjacencia):" << std::endl;
    for (int v = 0; v < grafo->numVertices; ++v) {
        std::cout << "Vertice " << v << ":";
        struct NoLista* temp = grafo->adjacencias[v];
        while (temp) {
            std::cout << " -> " << temp->destino;
            temp = temp->proximo;
        }
        std::cout << std::endl;
    }
}

// Função para liberar a memória alocada pelo grafo
void liberarGrafo(struct Grafo* grafo) {
    for (int i = 0; i < grafo->numVertices; ++i) {
        struct NoLista* atual = grafo->adjacencias[i];
        while (atual != NULL) {
            struct NoLista* proximo = atual->proximo;
            free(atual); // Libera o nó atual
            atual = proximo;
        }
    }
    free(grafo); // Libera a estrutura do grafo
}

int main() {
    int numVertices = 5; // Nosso grafo terá 5 vértices
    struct Grafo* grafo = criarGrafo(numVertices);

    // Adicionando as arestas do exemplo
    adicionarAresta(grafo, 0, 1);
    adicionarAresta(grafo, 0, 4);
    adicionarAresta(grafo, 1, 2);
    adicionarAresta(grafo, 1, 3);
    adicionarAresta(grafo, 1, 4);
    adicionarAresta(grafo, 2, 3);
    adicionarAresta(grafo, 3, 4);

    imprimirGrafo(grafo);

    liberarGrafo(grafo); // É importante liberar a memória alocada!
```

```
    return 0;
}
```

Análise do Código:

- Observe como usamos `struct` para simular "objetos" e `malloc` / `free` para gerenciamento manual de memória.
- A `struct NoLista` representa cada elemento da lista encadeada.
- `adjacencias[MAX_VERTICES]` é um array onde cada posição `i` guarda a cabeça da lista de nós adjacentes ao vértice `i`.
- A função `adicionarAresta` adiciona um novo nó no **início** da lista de adjacências de cada vértice envolvido (para grafos não-direcionados).
- A função `imprimirGrafo` percorre essas listas para mostrar as conexões.

Parte 2: Sua Tarefa - Implementar Grafo com Matriz de Adjacência

Agora, sua tarefa é implementar a representação de um grafo utilizando a **Matriz de Adjacência** em C++, seguindo um estilo similar ao exemplo fornecido.

Requisitos:

1. **Crie um novo arquivo** `.cpp` (ex: `grafo_matriz_adjacencia.cpp`).
2. **Representação da Matriz:** Utilize um array bidimensional (matriz) para representar as conexões.
    - Exemplo: `int matrizAdjacencia[MAX_VERTICES][MAX_VERTICES];`
    - Para um grafo não-direcionado e não-ponderado (sem pesos nas arestas), você pode usar `1` para indicar uma conexão e `0` para indicar ausência de conexão.
3. **Estrutura** `Grafo`: Crie uma `struct Grafo` que contenha:
    - A matriz de adjacência.
    - O número de vértices (`numVertices`).
4. **Funções Essenciais:**
    - `struct Grafo* criarGrafo(int V)`: Uma função que aloca memória para o grafo e inicializa a matriz de adjacência com todos os valores em `0`.
    - `void adicionarAresta(struct Grafo* grafo, int origem, int destino)`: Uma função que adiciona uma aresta entre `origem` e `destino`. Lembre-se que para grafos não-direcionados, se (i,j) é uma aresta, então (j,i) também é.
    - `void imprimirGrafo(struct Grafo* grafo)`: Uma função que imprime a matriz de adjacência de forma legível.

    - `void liberarGrafo(struct Grafo* grafo)`: Uma função que libera a memória alocada para o grafo.
5. **Função `main`:**
    - No `main`, crie um grafo com o mesmo número de vértices (5) e as mesmas arestas do exemplo da Lista de Adjacência (0-1, 0-4, 1-2, 1-3, 1-4, 2-3, 3-4).
    - Chame a função `imprimirGrafo` para mostrar sua representação.
    - Não se esqueça de chamar `liberarGrafo` ao final.

Dicas:

- Comece definindo o `MAX_VERTICES` (pode ser o mesmo 5) e a `struct Grafo`.
- A inicialização da matriz com zeros é fundamental.
- A função `adicionarAresta` será mais simples que na lista, pois não envolve ponteiros e alocação dinâmica para cada aresta, apenas atribuições na matriz.
- Pense em como você vai percorrer a matriz para imprimi-la.

Exemplo da saída esperada (para a matriz de adjacência):

```
Representacao do Grafo (Matriz de Adjacencia):
  0 1 2 3 4
0 0 1 0 0 1
1 1 0 1 1 1
2 0 1 0 1 0
3 0 1 1 0 1
4 1 1 0 1 0
```

(A primeira linha e coluna são apenas rótulos para facilitar a leitura.)