

Trabalho 2 - Teoria dos Grafos

Alunos: Davi Kirchmaier, Felipe Sant'anna, Vitória Nunes, Rayssa Amaral, Lucas Arruda

Professor: Gabriel Souza

Funcionalidades

- Inserção de nós
- Remoção de nós
- Inserção de arestas
- Remoção de arestas
- Alocação dinâmica da matriz
- Recalculo de índices
- Menor distância entre dois nós

Inserção de nós

- Grafo Lista

```
void GrafoLista::adicionaNo(int idNo)
{
    if (idNo < 0 || idNo >= ordem)
    {
        cout << "Erro: ID do nó inválido. " << endl;
        return;
    }

    cout << "Adicionando nó " << idNo << " à lista de adjacência..." << endl;

    // Criar nova lista de adjacência com o novo nó
    Lista* novaListaAdj = new Lista[ordem + 1];

    for (int i = 0; i < ordem; i++)
    {
        for (int j = 0; j < listaAdj[i].getTamanho(); j++)
        {
            novaListaAdj[i].adicionar(listaAdj[i].getELEMENTO(j)->getIdNo());
        }
    }

    // Adicionar o novo nó
    novaListaAdj[ordem].adicionar(idNo);

    // Liberar a memória da antiga lista de adjacência
    delete[] listaAdj;

    // Atualizar a estrutura do grafo
    listaAdj = novaListaAdj;
    ordem++;

    cout << "Nó " << idNo << " adicionado com sucesso! Nova ordem: " << ordem << endl;
}
```

- Grafo Matriz

```
void GrafoMatriz::adicionaNo(int idNo)
{
    if (idNo < 0 || idNo >= ordem)
    {
        cout << "Erro: ID do nó inválido. " << endl;
        return;
    }

    cout << "Adicionando nó " << idNo << " à matriz de adjacência...\n";

    // Criar nova matriz expandida
    int novaOrdem = ordem + 1;
    int** novaMatriz = new int*[novaOrdem];

    for (int i = 0; i < novaOrdem; i++)
    {
        novaMatriz[i] = new int[novaOrdem];
        for (int j = 0; j < novaOrdem; j++)
        {
            if (i == ordem || j == ordem)
            {
                novaMatriz[i][j] = 0;
            }
            else
            {
                novaMatriz[i][j] = matrizAdj[i][j];
            }
        }
    }

    // Liberar a matriz antiga
    for (int i = 0; i < ordem; i++)
    {
        delete[] matrizAdj[i];
    }
    delete[] matrizAdj;

    // Atualizar estrutura
    matrizAdj = novaMatriz;
    ordem = novaOrdem;

    cout << "Nó " << idNo << " adicionado com sucesso! Nova ordem: " << ordem << endl;
}
```

Remoção de nós

- Grafo Lista

```
void GrafoLista::deleta_no(int idNo)
{
    if (idNo <= 0 || idNo > ordem) // Ajustando a verificação de índice
    {
        cout << "Erro: ID do nó inválido." << endl;
        return;
    }

    idNo--; // Ajustar o ID para zero-based (o ID do arquivo começa em 1, mas a lista começa em 0)

    cout << "Removendo nó " << idNo + 1 << " da lista de adjacência..." << endl;

    // Remover todas as conexões do nó que será deletado
    for (int i = 0; i < ordem; i++)
    {
        listaAdj[i].remover(idNo);
    }

    // Criar nova lista de adjacência sem o nó removido
    Lista* novaListaAdj = new Lista[ordem - 1];

    int novoIndice = 0;
    for (int i = 0; i < ordem; i++)
    {
        if (i == idNo) continue; // Ignorar o nó que será removido

        for (int j = 0; j < listaAdj[i].getTamanho(); j++)
        {
            int adj = listaAdj[i].getElemento(j)->getIdNo();
            if (adj != idNo)
            {
                novaListaAdj[novoIndice].adicionar(adj);
            }
        }
        novoIndice++;
    }

    // Liberar a memória da antiga lista de adjacência
    delete[] listaAdj;

    // Atualizar a estrutura do grafo
    listaAdj = novaListaAdj;
    ordem--;

    cout << "Nó " << idNo + 1 << " removido com sucesso! Nova ordem: " << ordem << endl;
}
```

- Grafo Matriz

```
void GrafoMatriz::deleta_no(int idNo)
{
    if (idNo < 0 || idNo >= ordem)
    {
        cout << "Erro: ID do nó inválido. Ordem atual: " << ordem << endl;
        return;
    }

    cout << "Removendo nó " << idNo << " da matriz de adjacência...\n";

    // Atualizar IDs dos nós
    for(int i = idNo; i < ordem - 1;i++){
        No* no = getNoPeloId(i + 1);
        no->setIDNo(i);
    }

    // Criar nova matriz reduzida
    int novaOrdem = ordem - 1;
    int** novaMatriz = new int*[novaOrdem];

    for (int i = 0, ni = 0; i < ordem; i++)
    {
        if (i == idNo) continue;

        novaMatriz[ni] = new int[novaOrdem];
        for (int j = 0, nj = 0; j < ordem; j++)
        {
            if (j == idNo) continue;
            novaMatriz[ni][nj] = matrizAdj[i][j];
            nj++;
        }
        ni++;
    }

    // Liberar a matriz antiga
    for (int i = 0; i < ordem; i++)
    {
        delete[] matrizAdj[i];
    }
    delete[] matrizAdj;

    // Atualizar estrutura
    matrizAdj = novaMatriz;
    ordem = novaOrdem;

    cout << "Nó " << idNo << " removido com sucesso! Nova ordem: " << ordem << endl;
}
```

Inserção de arestas

- Grafo Lista

```
void GrafoLista::novaAresta(int origem, int destino, float peso)
{
    if (origem < 0 || origem >= ordem || destino < 0 || destino >= ordem)
    {
        std::cerr << "Erro: Índices de vértices inválidos." << std::endl;
        return;
    }

    listaAdj[origem].adicionar(destino, peso);

    if (!direcionado)
    {
        listaAdj[destino].adicionar(origem, peso);
    }

    std::cout << "Aresta adicionada: " << origem << " -> " << destino;
    if (ponderadoArestas)
    {
        std::cout << " com peso: " << peso;
    }
    std::cout << std::endl;
}
```

- Grafo Matriz

```
void GrafoMatriz::novaAresta(int origem, int destino, float peso)
{
    if (origem < 0 || origem >= numVertices || destino < 0 || destino >= numVertices)
    {
        std::cout << "Parâmetros errados!" << std::endl;
        return;
    }

    if (origem == destino)
    {
        std::cout << "Origem e destino iguais, erro!" << std::endl;
        return;
    }

    if (ponderadoArestas)
    {
        matrizAdj[origem][destino] = peso;
        if (!direcionado)
        {
            matrizAdj[destino][origem] = peso;
        }
    }
    else
    {
        matrizAdj[origem][destino] = 1;
        if (!direcionado)
        {
            matrizAdj[destino][origem] = 1;
        }
    }
}
```

Remoção de arestas

- Grafo Lista

```
void GrafoLista::removeAresta(int idNoOrigem, int idNoDestino, bool direcionado)
{
    if (!listaAdj[idNoOrigem].contem(idNoDestino))
    {
        std::cout << "Aresta inexistente" << std::endl;
        return;
    }

    listaAdj[idNoOrigem].remover(idNoDestino);
    listaAdj[idNoOrigem].getElemento(idNoOrigem) -> removeAresta(idNoDestino, direcionado);

    if (!direcionado)
    {
        listaAdj[idNoDestino].remover(idNoOrigem);
        listaAdj[idNoOrigem].getElemento(idNoOrigem) -> removeAresta(idNoOrigem, direcionado);
    }
}
```

Remoção de arestas

- Grafo Matriz

```
void GrafoMatriz::removeAresta(int idNoOrigem, int idNoDestino, bool direcionado)
{
    if (matrizAdj[idNoOrigem][idNoDestino] == 0)
    {
        std::cout << "Aresta inexistente" << std::endl;
        return;
    }

    matrizAdj[idNoOrigem][idNoDestino] = 0;
    if (!direcionado)
    {
        matrizAdj[idNoDestino][idNoOrigem] = 0;
        nos[idNoOrigem]->removeAresta(idNoDestino, direcionado);
    }

    nos[idNoOrigem]->removeAresta(idNoDestino, direcionado);
}
```

Recalcular de índices

- Implementado apenas na classe Grafo Matriz, na função deleta_no.

```
// Atualizar IDs dos nós
for(int i = idNo; i < ordem - 1;i++){
    No* no = getNoPeloId(i + 1);
    no->setIDNo(i);
}
```

Menor distância

- Implementado na classe Grafo.

```
int Grafo::menorDistancia(int origem, int destino)
{
    int *distancia = new int[ordem];
    bool *visitado = new bool[ordem];

    for (int i = 0; i < ordem; i++)
    {
        distancia[i] = INT_MAX;
        visitado[i] = false;
    }

    No *noOrigem = getNoPeloId(origem);
    distancia[origem] = noOrigem->getPesoNo();

    for (int count = 0; count < ordem - 1; count++)
    {
        int min = INT_MAX;
        int minIndex = -1;

        for (int v = 0; v < ordem; v++)
        {
            if (!visitado[v] && distancia[v] <= min)
            {
                min = distancia[v];
                minIndex = v;
            }
        }

        if (minIndex == -1)
            break;
        visitado[minIndex] = true;

        No *noAtual = getNoPeloId(minIndex);
        Aresta *aresta = noAtual->getPrimeiraAresta();

        while (aresta)
        {
            int v = aresta->getIdDestino();
            No *noDestino = getNoPeloId(v);

            if (!visitado[v] &&
                distancia[minIndex] != INT_MAX &&
                distancia[minIndex] + aresta->getPeso() + noDestino->getPesoNo() < distancia[v])
            {
                distancia[v] = distancia[minIndex] + aresta->getPeso() + noDestino->getPesoNo();
                aresta = aresta->getProxAresta();
            }
        }

        int resultado = distancia[destino];
        delete[] distancia;
        delete[] visitado;
    }

    return resultado == INT_MAX ? -1 : resultado;
}
```

Conclusão

As funções desenvolvidas nesse trabalho nos trouxeram uma boa noção de como trabalhar com os nós de forma correta em grafos, se atentando para os detalhes da implementação para cada tipo de grafo.