Simulador

Nomes:: Raynner Gabriel Taniguchi Silva 202010167 14A

Patrícia Souza Couto 202210524 14A

Main

A função main se inicia lendo uma lista de funções a serem executadas, seguida pela estrutura básica do grafo (número de vértices, arestas e se é direcionado ou não). Em seguida, o grafo é construído com base nas arestas fornecidas, e as funções listadas são executadas na ordem especificada, com o resultado sendo impresso. Funções que não se aplicam ao tipo de grafo são tratadas com a impressão de -1.

Leitura das Funções

```
string line;
getline(cin, line);
stringstream ss(line);
vector<int> functions;
int func;
while (ss >> func) {
    functions.push_back(func);
}
```

Leitura e Criação do Grafo

```
int V, E;
cin >> V >> E;
string type;
cin >> type;
bool directed = (type == "direcionado");
Graph g(V, directed);
for (int i = 0; i < E; ++i) {
    int id, u, v, w;
    cin >> id >> u >> v >> w;
    g.addEdge(id, u, v, w);
}
```

Execução das Funções Requeridas

```
for (int choice : functions) {
    switch (choice) {
        case 0:
            cout << (g.isConnected() ? 1 : 0) << endl;
            break;
        // Outras funções
    }
}
```

Classe Graph

A classe Graph é uma implementação para manipulação de grafos. É modular, com métodos privados que auxiliam a execução dos métodos públicos que implementa cada funcionalidade.

```
Graph::Graph(int V, bool directed) : V(V), directed(directed) {
    adj.resize(V);
    weight.resize(V, vector<int>(V, -1));
    edgeId.resize(V, vector<int>(V, -1));
}

void Graph::addEdge(int id, int u, int v, int w) {
    adj[u].push_back(v);
    if (!directed) adj[v].push_back(u);
    weight[u][v] = w;
    if (!directed) weight[v][u] = w;
    edgeId[u][v] = id;
    if (!directed) edgeId[v][u] = id;
}
```

Funcionalidades

0. Verificar se um grafo é conexo (para o caso de grafos orientados, verificar conectividade fraca).

```
bool Graph::isConnected() {
    // Se o grafo for direcionado, converte para não direcionado
    Graph gToCheck = directed ? getUndirected() : *this;

    vector<bool> visited(V, false);

    // Realiza DFS no grafo original a partir do vértice 0
    gToCheck.dfs(0, visited);

    // Verifica se todos os vértices foram visitados na primeira
passagem
    for (bool v : visited) {
        if (!v) {
            return false; // Se algum vértice não foi visitado, o
grafo não é conexo
        }
    }

    return true; // O grafo é conexo
}
```

A função isConnected verifica se um grafo é conexo, ou seja, se existe um caminho entre qualquer par de vértices. Se o grafo original for direcionado, ele é convertido para uma versão não

direcionada, usando a função getUndirected, assim sendo verificado se o grafo possui conectividade fraca. Um vetor visited é criado e inicializado com false. Esse vetor será usado para rastrear quais vértices foram visitados durante a execução da DFS. A função dfs é chamada a partir do vértice 0. Isso começa a exploração do grafo, marcando todos os vértices alcançáveis a partir de 0 como true em visited. Após a DFS, a função percorre o vetor visited. Se algum valor for false, significa que há um vértice que não foi alcançado, o que implica que o grafo não é conexo.
Caso todos os vértices tenham sido visitados, o grafo é considerado conexo.

1. Verificar se um grafo não-orientado é bipartido.

```
bool Graph::isBipartite() {
    vector<int> color(V, -1); // -1 significa não colorido, 0 e 1
são as duas cores
    queue<int> q;

    for (int i = 0; i < V; ++i) {
        if (color[i] == -1) { // Se o vértice não foi colorido
            q.push(i);
            color[i] = 0; // Começa com a cor 0

            while (!q.empty()) {
                int u = q.front();
                q.pop();

                for (int v : adj[u]) {
                    if (color[v] == -1) { // Se o vértice v não foi
colorido
                        color[v] = 1 - color[u]; // A cor do vértice
v é diferente da de u
                        q.push(v);
                    } else if (color[v] == color[u]) { // Se v e u
têm a mesma cor
                        return false; // O grafo não é bipartido
                    }
                }
            }
        }
    }

    return true; // O grafo é bipartido
}
```

A função isBipartite utiliza uma abordagem de BFS para tentar colorir o grafo usando duas cores (0 e 1). Se for possível fazer isso de tal forma que não existam arestas conectando vértices da mesma cor, o grafo é bipartido. Caso contrário, a função identifica o conflito e retorna false.

2. Verificar se um grafo qualquer é Euleriano.

```
bool Graph::isEulerian() {
    // Verifica se o grafo é conexo
    if (!isConnected()) {
        return false;
```

```
    }

    if (directed) {
        // Verifica se o grau de entrada é igual ao grau de saída
para cada vértice
        vector<int> in_degree(V, 0), out_degree(V, 0);

        for (int u = 0; u < V; ++u) {
            for (int v : adj[u]) {
                out_degree[u]++;
                in_degree[v]++;
            }
        }

        // Comparar os graus de entrada e saída
        for (int i = 0; i < V; ++i) {
            if (in_degree[i] != out_degree[i]) {
                return false; // Se o grau de entrada for diferente
do grau de saída, não é euleriano
            }
        }

        return true; // Se todas as condições forem satisfeitas, o
grafo direcionado é euleriano
    }

    // Verifica se todos os vértices têm grau par
    for (int i = 0; i < V; ++i) {
        if (adj[i].size() % 2 != 0) {
            return false; // Se algum vértice tiver grau ímpar, não
é possível ter um ciclo euleriano
        }
    }

    return true; // Todos os vértices têm grau par, então é possível
ter um ciclo euleriano
}
```

Para grafos direcionados, o grafo é Euleriano se e somente se o grau de entrada de cada vértice é igual ao grau de saída para cada vértice. a condição de igualdade entre os graus de entrada e saída deve ser verificada para garantir a existência de um ciclo Euleriano. Em grafos não direcionados, o grafo é Euleriano se e somente se todos os vértices têm grau par. A verificação de graus pares é suficiente para garantir a existência de um ciclo Euleriano.

3. Verificar se um grafo possui ciclo.

```
bool Graph::hasCycle() {
    if (directed) {
        vector<int> state(V, 0); // 0 = não visitado, 1 = em
processo, 2 = totalmente processado

        for (int i = 0; i < V; ++i) {
```

```
            if (state[i] == 0) {
                if (hasCycleUtilDirected(i, state)) {
                    return true;
                }
            }
        }
    } else {
        vector<bool> visited(V, false);

        for (int i = 0; i < V; ++i) {
            if (!visited[i]) {
                if (hasCycleUtilUndirected(i, visited, -1)) {
                    return true;
                }
            }
        }
    }

    return false; // Se nenhum ciclo for encontrado
}
```

Grafo Direcionado:

Cria um vetor state para rastrear o estado de cada vértice:
0 = Não Visitado
1 = Em Processo
2 = Totalmente Processado
DFS para Cada Vértice: Itera sobre todos os vértices. Se um vértice não foi visitado (state[i] == 0), inicia uma DFS para verificar ciclos usando hasCycleUtilDirected.

Grafo Não Direcionado:

Inicialização: Cria um vetor visited para rastrear os vértices visitados.
DFS para Cada Vértice: Itera sobre todos os vértices. Se um vértice não foi visitado (!visited[i]), inicia uma DFS para verificar ciclos usando hasCycleUtilUndirected.

Se algum ciclo for encontrado em qualquer chamada de hasCycleUtilDirected ou hasCycleUtilUndirected, retorna true. Caso contrário, após verificar todos os vértices, retorna false.

4. Calcular a quantidade de componentes conexas em um grafo não-orientado.

```
int Graph::connectedComponents() {
    if (directed) {
        return -1; // Se o grafo é orientado, retorna -1
    }

    vector<bool> visited(V, false);
    int componentCount = 0;

    for (int i = 0; i < V; ++i) {
        if (!visited[i]) {
```

```
            dfs(i, visited); // Inicia uma nova DFS para cada
componente
            componentCount++;
        }
    }

    return componentCount; // Retorna a quantidade de componentes
conexas
}
```

A função connectedComponents é usada para contar quantos conjuntos de vértices em um grafo não direcionado estão interconectados. Para cada vértice não visitado, uma DFS é iniciada para marcar todos os vértices no mesmo componente conexo, e o contador de componentes é incrementado. A função retorna o total de componentes conexos encontrados.
Se o grafo for direcionado, a função não pode calcular o número de componentes conexos e retorna -1

5. Calcular a quantidade de componentes fortemente conexas em um grafo orientado.

```
int Graph::stronglyConnectedComponents() {
    if (!directed) {
        return -1; // Se o grafo não é orientado, retorna -1
    }
    vector<int> disc(V, -1);
    vector<int> low(V, -1);
    vector<bool> stackMember(V, false);
    stack<int> st;
    int sccCount = 0;

    // Faz a chamada recursiva para encontrar todas as SCCs
    for (int i = 0; i < V; i++) {
        if (disc[i] == -1) {
            stronglyConnectedComponentsUtil(i, disc, low, st,
stackMember, sccCount);
        }
    }

    return sccCount;
}
```

O algoritmo de Tarjan foi implementado para encontrar os componentes fortemente conectados em um grafo direcionado.
“stronglyConnectedComponentsUtil” executa o algoritmo de Tarjan para encontrar SCCs, mantendo um rastreamento de tempos de descoberta e valores de menor tempo acessível, e identificando raízes de SCCs através de uma pilha.
“stronglyConnectedComponents” inicializa as estruturas de dados necessárias e chama a função auxiliar para todos os vértices, retornando a quantidade total de SCCs no grafo.

6. Imprimir os vértices de articulação de um grafo não-orientado (priorizar a ordem lexicográfica dos vértices).

```
vector<int> Graph::articulationPoints() {
```

```
    vector<int> disc(V, -1); // Tempo de descoberta
    vector<int> low(V, -1); // Tempo de menor alcance
    vector<bool> visited(V, false);
    vector<bool> isAP(V, false);
    int time = 0;
    int par = -1;

    for (int u = 0; u < V; u++) {
        if (!visited[u]) {
            articulationPointsUtil(u, visited, disc, low, time, par,
isAP);
        }
    }

    vector<int> articulationPointsList;
    for (int u = 0; u < V; u++) {
        if (isAP[u]) {
            articulationPointsList.push_back(u);
        }
    }

    return articulationPointsList;
}
```

“articulationPointsUtil” implementa a lógica recursiva para encontrar pontos de articulação em um grafo não direcionado, utilizando o algoritmo baseado em DFS para calcular os tempos de descoberta e os valores de menor alcance (low). Identifica pontos de articulação tanto para vértices internos quanto para a raiz da árvore DFS.
“articulationPoints” configura e chama a função auxiliar para cada vértice não visitado, coletando todos os pontos de articulação e retornando uma lista com esses vértices.

7. Calcular quantas arestas ponte possui um grafo não-orientado.

```
int Graph::bridges() {
    vector<int> disc(V, -1);
    vector<int> low(V, -1);
    vector<bool> visited(V, false);
    int time = 0;
    int bridgeCount = 0;

    for (int u = 0; u < V; u++) {
        if (!visited[u]) {
            bridgeUtil(u, visited, disc, low, time, -1, bridgeCount);
        }
    }

    return bridgeCount;
}
```

“bridgeUtil” implementa a lógica recursiva para encontrar todas as pontes em um grafo não direcionado, usando o algoritmo de Tarjan. Atualiza os tempos de descoberta e os valores de menor alcance (low) e identifica as arestas que são pontes.

“bridges” configura e chama a função auxiliar bridgeUtil para cada vértice não visitado, conta as pontes e retorna o total.

8. Imprimir a árvore em profundidade (priorizando a ordem lexicográfica dos vértices; 0 é a origem). Você deve imprimir o identificador das arestas. Caso o grafo seja desconexo, considere apenas a árvore com a raíz 0.

```
void Graph::printDFSTree() {
    vector<bool> visited(V, false);
    vector<int> edgeIds;

    // Começa a DFS a partir do vértice 0
    dfs(0, visited, edgeIds);

    // Imprime o identificador das arestas da árvore em profundidade
    for (int id : edgeIds) {
        cout << id << " ";
    }
    cout << endl;
}
```

A função dfs realiza uma busca em profundidade (DFS) no grafo a partir de um vértice específico e coleta os identificadores das arestas da árvore DFS. A ordem de exploração dos vizinhos é lexicograficamente ordenada. “printDFSTree” Inicia a busca em profundidade a partir do vértice 0 e imprime os identificadores das arestas que formam a árvore DFS.

9. Imprimir a árvore de largura (priorizando a ordem lexicográfica dos vértices; 0 é a origem). Você deve imprimir o identificador das arestas. Caso o grafo seja desconexo, considere apenas a árvore com a raíz 0.

```
void Graph::printBFSTree() {
    vector<int> edgeIds;

    // Começa a BFS a partir do vértice 0
    bfs(0, edgeIds);

    // Imprime os IDs das arestas da árvore em largura
    for (int id : edgeIds) {
        cout << id << " ";
    }
    cout << endl;
}
```

A função bfs realiza uma busca em largura (BFS) no grafo a partir de um vértice inicial e coleta os identificadores das arestas da árvore BFS. A função printBFSTree inicia a busca em largura a partir do vértice 0 e imprime os identificadores das arestas da árvore BFS.

10. Calcular o valor final de uma árvore geradora mínima (para grafos não-orientados).

```
int Graph::minimumSpanningTree() {
```

```
    vector<pair<int, pair<int, int>>> edges; // Vetor de arestas:
(peso, (u, v))

    // Preenche o vetor de arestas
    for (int u = 0; u < V; ++u) {
        for (int v : adj[u]) {
            if (u < v) { // Para evitar duplicidade, considere u < v
para grafos não direcionados
                edges.push_back({weight[u][v], {u, v}});
            }
        }
    }

    // Ordena as arestas pelo peso
    sort(edges.begin(), edges.end());

    // Estruturas de dados para o algoritmo de Kruskal
    vector<int> parent(V);
    vector<int> rank(V, 0);

    // Inicializa cada vértice como seu próprio pai (representante)
    for (int i = 0; i < V; ++i) {
        parent[i] = i;
    }

    int mstValue = 0;

    // Processa as arestas em ordem crescente de peso
    for (auto &edge : edges) {
        int w = edge.first;
        int u = edge.second.first;
        int v = edge.second.second;

        int rootU = find(u, parent);
        int rootV = find(v, parent);

        // Se u e v estão em componentes diferentes, adicione a
aresta à MST
        if (rootU != rootV) {
            mstValue += w;
            unionSets(rootU, rootV, parent, rank);
        }
    }

    return mstValue;
}
```

A função “minimumSpanningTree” calcula o peso total da árvore geradora mínima (MST) do grafo usando o algoritmo de Kruskal. Primeiro, coleta todas as arestas do grafo e as ordena por peso, após usa o algoritmo de Kruskal para adicionar arestas à MST, garantindo que não haja ciclos.

11. Imprimir a ordem os vértices em uma ordenação topológica. Esta função não fica disponível em grafos não direcionado. Deve-se priorizar a ordem lexicográfica dos vértices.

```
vector<int> Graph::topologicalSort() {

    stack<int> Stack;
    vector<bool> visited(V, false);

    // Chama a função auxiliar recursiva para cada vértice não
visitado
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            topologicalSortUtil(i, visited, Stack);
        }
    }

    // Armazena a ordem topológica
    vector<int> topologicalOrder;

    // Desempilha os vértices da pilha e adiciona na ordem final
    while (!Stack.empty()) {
        topologicalOrder.push_back(Stack.top());
        Stack.pop();
    }

    return topologicalOrder;
}
```

A função topologicalSortUtil realiza uma busca em profundidade (DFS) para ordenar os vértices de um grafo direcionado de forma topológica, e topologicalSort realiza a ordenação topológica completa dos vértices do grafo. O algoritmo inicializa uma pilha e um vetor de visitados, executa a função auxiliar para todos os vértices não visitados, desempilha os vértices da pilha e os adiciona ao vetor de ordem topológica e retorna o vetor que contém a ordem topológica dos vértices.

12. Valor do caminho mínimo entre dois vértices (para grafos não-orientados com pelo menos um peso diferente nas arestas). 0 é a origem; n-1 é o destino.

```
int Graph::shortestPath(int src, int dest) {
    // Vetor de distâncias, inicializado com infinito
    vector<int> dist(V, numeric_limits<int>::max());
    dist[src] = 0; // Distância da origem a si mesma é zero

    // Fila de prioridade (min-heap) para armazenar vértices a serem
processados
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<>> pq;
    pq.emplace(0, src);

    while (!pq.empty()) {
        int u = pq.top().second;
        int current_dist = pq.top().first;
        pq.pop();
```

```
        // Se já alcançou o destino, retorne a distância mínima
        if (u == dest) {
            return current_dist;
        }

        // Processa todos os vértices adjacentes a u
        for (int v : adj[u]) {
            int w = weight[u][v];
            if (w != -1) { // Verifica se existe uma aresta entre u
e v
                int new_dist = current_dist + w;
                if (new_dist < dist[v]) {
                    dist[v] = new_dist;
                    pq.emplace(new_dist, v);
                }
            }
        }
    }

    // Se o destino não é alcançável, retorna -1 (indicando que não
há caminho)
    return -1;
}
```

A função shortestPath encontra o caminho mais curto entre dois vértices em um grafo ponderado usando o algoritmo de Dijkstra. Começa definindo o vetor dist para armazenar as distâncias mínimas dos vértices a partir da origem, inicializado como infinito (numeric_limits<int>::max()), exceto para o vértice src que é inicializado com distância zero. O algoritmo utiliza uma fila de prioridade (min-heap) para processar os vértices, onde a distância acumulada é a chave de ordenação. O vértice u com a menor distância da fila de prioridade é extraido, se u é o vértice de destino dest, retorna a distância acumulada. Atualiza a distância para todos os vértices adjacentes a u se uma distância menor for encontrada e insere esses vértices na fila de prioridade. Se o vértice dest não é alcançável, retorna -1 indicando que não há caminho entre src e dest.

13. Valor do fluxo máximo para grafos direcionados. 0 é a origem; n-1 é o destino.

```
int Graph::maxFlow() {
    int n = weight.size();
    int source = 0;
    int sink = V - 1;

    vector<vector<int>> flow(n, vector<int>(n, 0));
    vector<int> parent(n);
    int max_flow = 0;

    while (bfs(weight, flow, source, sink, parent)) {
        // Encontra o fluxo máximo através do caminho encontrado
pelo BFS
        int path_flow = INT_MAX;
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
```

```
            path_flow = min(path_flow, weight[u][v] - flow[u][v]);
        }

        // Atualiza as capacidades residuais das arestas e arestas
reversas ao longo do caminho
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            flow[u][v] += path_flow;
            flow[v][u] -= path_flow;
        }

        max_flow += path_flow;
    }

    return max_flow;
}
```

A função maxFlow implementa o algoritmo de Ford-Fulkerson para encontrar o fluxo máximo em um grafo com capacidade de arestas. A busca em largura visa encontrar um caminho aumentante do vértice source ao vértice sink no grafo residual. A função maxFlow Inicializa a matriz de fluxos com zeros e o valor total de fluxo (max_flow) como zero, Usa a função bfs para encontrar caminhos aumentantes e, Para cada caminho aumentante encontrado, calcula o fluxo máximo possível através do caminho e atualiza as capacidades residuais e fluxos das arestas.

14. Fecho transitivo para grafos direcionados. Deve-se priorizar a ordem lexicográfica dos vértices; 0 é o vértice escolhido.

```
vector<vector<bool>> Graph::transitiveClosure() {
    vector<vector<bool>> reachability(V, vector<bool>(V, false));

    for (int i = 0; i < V; ++i) {
        vector<bool> visited(V, false);
        dfs(i, reachability, visited, i);
    }
    return reachability;
}
```

A função transitiveClosure e a função auxiliar dfs são utilizadas para calcular a matriz de fechamento transitivo de um grafo, que é uma matriz que indica se há um caminho entre quaisquer dois vértices. A busca em profundidade realizada a partir do vértice v marca a matriz de fechamento transitivo “reachability” para indicar que há um caminho do vértice start até v. Apenas os vértices acessíveis a partir do vértice 0 são impressos.