Documentação

1. Introdução

Este código fornece uma implementação robusta de diversos algoritmos e técnicas para manipulação e análise de grafos. Inclui funcionalidades para busca, detecção de ciclos, identificação de componentes fortemente conectados, entre outras operações avançadas.

1.1 Inclusões e Definições

- **Bibliotecas Incluídas:**
  - <iostream>: Entrada e saída padrão.
  - <cstring>: Manipulação de strings C-style.
  - <vector>: Contêiner de vetores dinâmicos.
  - <string>: Manipulação de strings.
  - <sstream>: Manipulação de streams de strings.
  - <algorithm>: Funções de algoritmos.
  - <queue>: Contêiner para filas.
  - <set>: Contêiner para conjuntos.
- **Definições e Tipos:**
  - #define DIRECIONADO 1: Define um grafo direcionado.
  - #define NAO_DIRECIONADO 0: Define um grafo não direcionado.
  - typedef vector<vector<pair<int, pair<int, int>>>> grafo: Define o tipo grafo como uma lista de adjacências.

1.2. Variáveis Globais

- grafo lista_adj, grafo_associado, lista_adj_rev: Listas de adjacência para grafos.
- vector<vector<int>> capacidade: Matriz de capacidades para fluxos de redes.
- vector<pair<long long, pair<int, int>>> edgeList: Lista de arestas para algoritmos de MST.
- vector<int> funcoes, visitado: Vetores para funções a serem testadas e controle de visitação.
- int n_arestas, n_vertices: Número de arestas e vértices.
- bool b_direcionado: Flag para verificar se o grafo é direcionado.
- string direcionado: Tipo de grafo (direcionado ou não).

1.3. Funções

- **leitura_direcionado()**

    - **Descrição:** Lê um grafo direcionado da entrada e preenche as estruturas de dados.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Nenhum.
- **leitura_nao_direcionado()**
    - **Descrição:** Lê um grafo não direcionado da entrada e preenche as estruturas de dados.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Nenhum.
- **dfs(int u, bool print)**
    - **Descrição:** Realiza a busca em profundidade (DFS) a partir do vértice u.
    - **Parâmetros:**
        - int u: Vértice inicial para DFS.
        - bool print: Se true, imprime os vértices visitados.
    - **Retorno:** Nenhum.
- **bool conexo()**
    - **Descrição:** Verifica se o grafo é conexo.
    - **Parâmetros:** Nenhum.
    - **Retorno:** true se o grafo for conexo, false caso contrário.
- **bool bipartido()**
    - **Descrição:** Verifica se o grafo é bipartido.
    - **Parâmetros:** Nenhum.
    - **Retorno:** true se o grafo for bipartido, false caso contrário.
- **bool euleriano()**
    - **Descrição:** Verifica se o grafo é euleriano.
    - **Parâmetros:** Nenhum.
    - **Retorno:** true se o grafo for euleriano, false caso contrário.
- **bool ciclo()**
    - **Descrição:** Verifica se o grafo contém ciclos.
    - **Parâmetros:** Nenhum.
    - **Retorno:** true se o grafo contiver pelo menos um ciclo, false caso contrário.
- **int componentes_conexas()**
    - **Descrição:** Conta o número de componentes conexas no grafo.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Número de componentes conexas.
- **void kosaraju()**
    - **Descrição:** Aplica o algoritmo de Kosaraju para encontrar componentes fortemente conexos.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Nenhum.
- **int articulacoes(int print)**
    - **Descrição:** Encontra os pontos de articulação e pontes no grafo.
    - **Parâmetros:**
        - int print: Se true, imprime os pontos de articulação.
    - **Retorno:** Número de pontes no grafo.
- **void dfs_tree(int u)**
    - **Descrição:** Realiza DFS e imprime a árvore DFS.
    - **Parâmetros:**

        - int u: Vértice inicial para DFS.
    - **Retorno:** Nenhum.
- **void arvore_dfs()**
    - **Descrição:** Inicia a DFS e imprime a árvore DFS a partir do vértice 0.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Nenhum.
- **void bfs_tree()**
    - **Descrição:** Realiza BFS e imprime a árvore BFS a partir do vértice 0.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Nenhum.
- **ll kruskal()**
    - **Descrição:** Calcula o custo do MST usando o algoritmo de Kruskal.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Custo total do MST.
- **void topo(int u)**
    - **Descrição:** Realiza a ordenação topológica usando DFS.
    - **Parâmetros:**
        - int u: Vértice inicial para DFS.
    - **Retorno:** Nenhum.
- **void topologicalsort()**
    - **Descrição:** Realiza a ordenação topológica e imprime o resultado.
    - **Parâmetros:** Nenhum.
    - **Retorno:** Nenhum.
- **int dijkstra(int s, int t)**
    - **Descrição:** Calcula a menor distância entre os vértices s e t usando o algoritmo de Dijkstra.
    - **Parâmetros:**
        - int s: Vértice de origem.
        - int t: Vértice de destino.
    - **Retorno:** Menor distância entre s e t.
- **int bfs(int s, int t, vector<int>& parent)**
    - **Descrição:** Realiza uma busca em largura para encontrar o caminho de fluxo máximo.
    - **Parâmetros:**
        - int s: Vértice de origem.
        - int t: Vértice de destino.
        - vector<int>& parent: Vetor para armazenar o caminho.
    - **Retorno:** Fluxo máximo possível entre s e t.
- **int maxflow(int s, int t)**
    - **Descrição:** Calcula o fluxo máximo entre os vértices s e t usando o algoritmo de Ford-Fulkerson.
    - **Parâmetros:**
        - int s: Vértice de origem.
        - int t: Vértice de destino.
    - **Retorno:** Fluxo máximo entre s e t.
- **void fecho()**
    - **Descrição:** Realiza uma DFS a partir do vértice 0 e imprime o resultado.
    - **Parâmetros:** Nenhum.

- **Retorno:** Nenhum.

1.4. Função main()

- **Descrição:** Função principal que orquestra a leitura do grafo, a seleção de funções a serem testadas e a execução dessas funções.
- **Parâmetros:** Nenhum.
- **Retorno:** Nenhum.

2. Estrutura do Código

O código é estruturado para oferecer flexibilidade e eficiência na execução de diferentes algoritmos relacionados a grafos. Ele é modular e permite a execução de operações específicas conforme a necessidade do usuário.

3. Funcionalidades Implementadas

3.1. Busca em Profundidade (DFS)

Descrição: A Busca em Profundidade (DFS) explora o grafo de maneira recursiva, visitando o máximo de nós possível antes de retroceder.

Funcionalidades:

- **Detecção de Ciclos:** Identifica ciclos em grafos direcionados e não direcionados. Um ciclo é detectado quando um vértice já visitado é alcançado novamente durante a exploração.
- **Componentes Fortemente Conectados (SCC):** Utiliza DFS para o algoritmo de Kosaraju, preenchendo uma pilha com a ordem de término dos vértices na primeira passagem e explorando o grafo transposto na segunda passagem.
- **Pontos de Articulação e Pontes:** Detecta vértices cuja remoção desconecta o grafo (pontos de articulação) e arestas cuja remoção desconecta o grafo (pontes). Usa o conceito de tempo de descoberta e retorno de arestas.

3.2. Busca em Largura (BFS)

Descrição: A Busca em Largura (BFS) explora o grafo em camadas, visitando todos os vértices no nível atual antes de passar para o próximo nível.

Funcionalidades:

- **Determinação do Menor Caminho:** Calcula o menor caminho em grafos não ponderados.
- **Verificação da Bipartição:** Determina se um grafo pode ser dividido em dois conjuntos disjuntos, onde cada aresta conecta vértices de conjuntos diferentes.

3.3. Detecção de Ciclos

Descrição: Detecta a presença de ciclos em grafos direcionados e não direcionados.

Método: Utiliza DFS para explorar o grafo. A presença de um ciclo é identificada quando um vértice já visitado é encontrado novamente durante a busca.

3.4. Componentes Fortemente Conectadas (SCC)

Descrição: Identifica componentes fortemente conectados em grafos direcionados, onde cada vértice pode alcançar todos os outros vértices na mesma componente.

Método: Implementa o algoritmo de Kosaraju:

- **Primeira Passagem:** Realiza DFS para preencher uma pilha com a ordem de término dos vértices.
- **Segunda Passagem:** Realiza DFS no grafo transposto a partir dos vértices na ordem da pilha para identificar componentes fortemente conectados.

3.5. Pontos de Articulação e Pontes

Descrição: Identifica pontos de articulação e pontes em um grafo.

Método: Usa DFS para calcular o tempo de descoberta e verificar arestas críticas (pontes) e vértices críticos (pontos de articulação).

3.6. Árvores Geradoras Mínimas

Descrição: Encontra a árvore geradora mínima (MST) de um grafo ponderado, onde a soma dos pesos das arestas é minimizada.

Algoritmos:

- **Algoritmo de Prim:** Utiliza uma fila de prioridade para selecionar as arestas de menor peso, começando de um vértice inicial e expandindo a árvore.
- **Algoritmo de Kruskal:** Usa uma estrutura de união-find para evitar ciclos e adicionar arestas de menor peso para construir a MST.

3.7. Ordenação Topológica

Descrição: Executa uma ordenação linear dos vértices de um grafo acíclico direcionado (DAG), com base nas dependências entre os vértices.

Métodos:

- **DFS:** Realiza a ordenação topológica utilizando a pilha de vértices.
- **Algoritmo de Kahn:** Usa um processo de construção de grau de entrada para ordenar os vértices.

3.8. Algoritmo de Dijkstra

Descrição: Encontra o caminho mais curto a partir de um vértice fonte para todos os outros vértices em um grafo com pesos não negativos.

Método: Utiliza uma fila de prioridade (min-heap) para otimizar a escolha da aresta a ser relaxada.

3.9. Fluxo Máximo

Descrição: Determina o fluxo máximo possível entre dois vértices em um grafo de fluxo.

Algoritmo:

- **Edmonds-Karp:** Implementa uma versão do algoritmo de Ford-Fulkerson usando BFS para encontrar o fluxo máximo.

3.9.1. Fecho Transitivo

Descrição: Encontra o fecho transitivo de um vértice, ou seja, todos os vértices que são alcançáveis a partir de um vértice inicial.

Método: Utiliza DFS para explorar todos os vértices alcançáveis a partir do vértice inicial e coleta esses vértices em um conjunto.

4. Execução do Código

4.1. Inicialização e Leitura de Dados

Descrição: O código lê dados do grafo a partir de um arquivo ou entrada padrão.

Procedimentos:

- **Leitura de Vértices e Arestas:** Preenche a estrutura de dados com os vértices e arestas fornecidos.
- **Escolha de Operações:** Permite ao usuário selecionar quais algoritmos executar.

4.2. Escolha de Operações

Descrição: Oferece flexibilidade para o usuário escolher quais operações executar com base nas necessidades de análise do grafo.

Funcionalidades:

- **Análise de Conectividade:** Verifica se o grafo está conectado ou não.
- **Cálculo de Caminhos Mínimos:** Executa o algoritmo de Dijkstra ou BFS para determinar o menor caminho.
- **Construção de MSTs:** Utiliza os algoritmos de Prim e Kruskal para encontrar a árvore geradora mínima.

5. Conclusão

Este código oferece uma gama abrangente de funcionalidades para a manipulação e análise de grafos, desde operações básicas como busca e detecção de ciclos até algoritmos avançados para fluxo máximo e árvores geradoras mínimas. A estrutura modular e a aplicação de técnicas eficientes garantem que o código seja adaptável a diferentes contextos e escalável para grafos de diferentes tamanhos e complexidades.