Documentação

1. Introdução

Este projeto implementa várias funcionalidades avançadas de grafos utilizando C++. As funções abrangem desde buscas e detecção de ciclos até algoritmos de otimização, como Dijkstra e fluxos máximos. O código trabalha com grafos direcionados e não direcionados, utilizando listas de adjacência para armazenar as arestas.

2. Estrutura de Dados

2.1. Representação do Grafo

O grafo é representado por um vetor de listas de pares, onde cada lista armazena as arestas e seus respectivos pesos. Esta abordagem permite uma manipulação eficiente dos vértices e arestas do grafo, garantindo flexibilidade e desempenho.

```
vector<vector<pair<int, int>>> adj;  // Lista de adjacência com pesos
```

2.2. Utilização de Conjuntos e Filas

Além das listas de adjacência, conjuntos (`set`) e filas (`queue`) são utilizados para gerenciar vértices visitados, processar componentes e executar algoritmos como DFS e BFS.

3. Funcionalidades Implementadas

3.1. Busca em Profundidade (DFS)

Implementada para explorar o grafo de maneira recursiva, visitando o máximo de nós possível antes de retroceder. Esta função é crucial para a detecção de ciclos, componentes fortemente conectadas e pontos de articulação.

```
void dfs(int v, vector<bool>& visited);
```

3.2. Busca em Largura (BFS)

Explora o grafo em camadas, útil para determinar o menor caminho em grafos não ponderados e para verificar a bipartição de grafos.

```
void bfs(int start);
```

3.3. Detecção de Ciclos

Este algoritmo verifica a existência de ciclos em grafos direcionados e não direcionados. A detecção é realizada utilizando DFS, onde um ciclo é identificado quando um vértice já visitado é alcançado novamente.

```
bool detectCycleDFS(int v, vector<bool>& visited, vector<bool>&
recStack);
```

3.4. Componentes Fortemente Conectadas (SCC)

Implementação do algoritmo de Kosaraju para identificar componentes fortemente conectadas em grafos direcionados. Utiliza duas passagens de DFS: a primeira para preencher uma pilha com a ordem de término dos vértices, e a segunda para explorar o grafo transposto.

```
void findSCCs();
```

3.5. Pontos de Articulação e Pontes

Detecta pontos de articulação (vértices que, se removidos, desconectam o grafo) e pontes (arestas críticas). Isso é feito utilizando DFS com o conceito de tempo de descoberta e a técnica de retorno de arestas.

```
void findArticulationPoints();
```

3.6. Árvores Geradoras Mínimas

Implementação dos algoritmos de Prim e Kruskal para encontrar a árvore geradora mínima (MST). O algoritmo de Prim utiliza uma fila de prioridade para escolher as arestas de menor peso, enquanto Kruskal utiliza um conjunto de união-find para evitar ciclos.

```
void primMST();
void kruskalMST();
```

3.7. Ordenação Topológica

Executa uma ordenação linear dos vértices de um grafo acíclico direcionado (DAG) com base em suas dependências. A ordenação é realizada utilizando DFS ou Kahn's Algorithm.

```
void topologicalSort();
```

3.8. Algoritmo de Dijkstra

Encontra o caminho mais curto a partir de um vértice fonte para todos os outros vértices em grafos com pesos não negativos. Utiliza uma fila de prioridade (min-heap) para otimizar a escolha da aresta a ser relaxada.

```
void dijkstra(int src);
```

3.9. Fluxo Máximo (Edmonds-Karp)

Implementa o algoritmo de Edmonds-Karp (uma versão em BFS do algoritmo de Ford-Fulkerson) para encontrar o fluxo máximo entre dois vértices em um grafo de fluxo.

```
int edmondsKarp(int source, int sink);
```

4. Execução do Código

4.1. Inicialização e Leitura de Dados

Os dados do grafo são lidos a partir de um arquivo ou entrada padrão. A estrutura de dados é preenchida com base no número de vértices e arestas fornecidos, e o código permite a escolha das operações a serem executadas.

```
void readGraphFromFile(string filename);
```

4.2. Escolha de Operações

O usuário pode escolher quais algoritmos executar, proporcionando flexibilidade para diferentes cenários de uso, como análise de conectividade, cálculo de caminhos mínimos ou construção de MSTs.

5. Conclusão

Este código oferece uma ampla gama de funcionalidades para manipulação de grafos, cobrindo desde operações básicas até algoritmos avançados. A estrutura modular e a utilização de técnicas eficientes garantem que o código seja adaptável e escalável, permitindo sua aplicação em diversos contextos.