INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: ALGORITMOS E ESTRUTURAS DE DADOS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: GRAFOS – BUSCAS E MENOR CAMINHO

Prática 09

OBS.: Ver código que acompanha a prática.

Parte 1: Implementando Buscas num Grafo Direcionado

Problema 1: Em **graph.cpp**, implemente a função `DFS(...)` (privada), que implementa a busca em profundidade, conforme descrito no material.

> A função `dfs()` é a função pública chamada pelo usuário (na `main()`). Ela cria as estruturas auxiliares necessárias (ex.: vetor de nós visitados `visited`) e repassa para a função privada `DFS()`, que realiza a busca em profundidade de fato. Essa separação permite que a função `DFS()` se chame recursivamente, que é a implementação mais direta, sem ficar realocando as estruturas auxiliares a cada chamada. O vetor `visited` deve funcionar de forma que se um vértice `vtx` já foi visitado pela busca, `visited[vtx] == true`, do contrário `visited[vtx] == false`.
>
> Siga o pseudocódigo do material de aula para implementar `DFS()`. Ela recebe, além de `visited`, uma referência para a lista `result` que deve conter ao final os nós na ordem em que foram atravessados na busca em profundidade.
>
> **ATENÇÃO:** a função **não** deve imprimir (exibir) os nós visitados na tela, mas sim adicionar à lista `result` à medida que são visitados.
>
> Teste rodando o **main.cpp** e veja se o resultado é o esperado.

Problema 2: Em **graph.cpp,** implemente a função `BFS(...)` (privada), que implementa a busca em largura, conforme descrito no material:

> A busca em largura também foi quebrada em dois métodos: um público `bfs()` e um privado `BFS()`. Nesse caso a separação é apenas por organização e para manter a consistência com a busca em profundidade, mas não é estritamente necessária. Siga o pseudocódigo do material de aula para implementar `BFS()`. Essa função recebe o vetor `visited` e a lista `result` que deve conter ao final os nós na ordem em que foram atravessados na busca em largura. Use a fila (`queue`) da STL na sua implementação.
>
> Teste rodando o **main.cpp** e veja se o resultado é o esperado.

Desafio (Opcional): Implemente a função `dijkstra()` (privada).

> Essa função deve implementar do algoritmos de Dijkstra para achar o menor caminho, e será usada pela função `spf()`. A função `spf()` (de *shortest path first*) computa o menor caminho de um vértice de origem (`src`) para outro de destino (`dst`) . Ela chama inicialmente a função `dijkstra()`, que calcula todos os menores caminhos da origem (`src`) até todos os outros vértices do grafo. `dijkstra()` guarda no vetor `dist` a menor distância de `src` para o todos os outros nós (ex.: `dist[x]` = menor

distância de `src` até `x`), enquanto o vetor `prev` (*previous*) contém o nó anterior no caminho de `src` até um dado nó (ex.: `prev[x]` deve retornar o nó que, no caminho de `src` até `x`, está imediatamente antes de `x`). Se `prev[x]` igual a `-1` indica que não há anterior; só é o caso para `prev[src]` ou se `x` não for alcançável a partir de `src` (não há caminho até `x`). Após a chamada de `dijkstra()`, `spf()` chama `path()` para extrair o menor caminho para o destino de `src` a `dst` e coloca numa lista.

Para a implementação de `dijkstra()`, recomenda-se expandir o código do `BFS()`, substituindo a fila simples (`queue`) por um `set` de pares `<long, int>` (distância acumulada, vértice). Esse `set` funciona como uma fila de prioridade com complexidade logarítmica, onde os pares são ordenados primeiro pela distância, e depois pelo vértice (se a distância for igual). A cada iteração do *loop*, deve ser retirado o menor par do set, e visitados os vizinhos do vértice. Para cada vizinho, verificar se é possível melhorar a distância acumulada (ver algoritmo), e se for o caso, deve-se remover o par (distância atual, vizinho) do `set`, e inserir um novo par com a distância atualizada. Também deve-se atualizar os vetores `prev` e `dist`.

Teste rodando o **main.cpp** e veja se o resultado é o esperado.

Parte 2: Resolvendo problemas usando buscas

Problema 1: Achando ciclos com a busca em profundidade com DFS.

Implemente a função privada `has_cycle(src, visited)`. Essa função deve empregar uma busca em profundidade para determinar se há um ciclo a partir do vértice `src`. Nessa busca, desça recursivamente e retorne `true` assim que encontrar um vértice já visitado. Caso não encontre, retorne `false`.

A função `has_cycle(src, visited)` é chamada por `has_cycle(src)`, que apenas criar o vetor de visitados, assim como em `dfs()` e `DFS()`. Para determinar se o grafo em si tem um ciclos ou não, a função `has_cycle()` (sem parâmetros) chama `has_cycle(src)` para todos os vértices.

Teste rodando o **main.cpp** e veja se o resultado é o esperado.

Desafio 1 (Opcional): Resolva o problema *Course Schedule* no LeetCode (LC207).

Nesse problema, dado um conjunto de pares de inteiros que presentam dependências entre disciplinas (*courses*), deve-se dizer se é possível realmente cursá-las. Isto é, se não existe dependências circulares entre elas que as tornam impossíveis de serem cursadas. Esse problema requer encontrar ciclos no grafo de dependências, assim como na função implementada no **Problema 1**, e é equivalente a dizer se existe ou não um ordenação topológica para esse grafo.

Desafio 2 (Opcional): Resolva o problema *All Paths From Source to Target* no LeetCode (LC797).

Nesse problema é pedido para se achar todos os caminhos entre dois vértices num grafo (especificamente, o vértice `0` e o vértice `n - 1`). Para isso, realize uma busca em profundidade, guardando o caminho parcial até o momento (num `vector` ou `list`), e sempre que encontrar o vértice de destino, adicione o caminho à lista de caminhos.

Problema 2: Descobrindo se dois vértices são alcançáveis com BFS.

Implemente a função `reachable(src, dst)`, que diz se dois vértices são alcançáveis entre si, isto é, se existe um caminho entre `src` e `dst` que respeite as direções das arestas. Essa função deve empregar uma variação da busca em largura (BFS), que finaliza retornando `true` assim que encontra o vértice de destino. Caso nunca encontre esse vértice, deve retornar `false`.

Teste rodando o **main.cpp** e veja se o resultado é o esperado.

Desafio 3 (Opcional): Resolva o problema Rede de Fibra no Beecrowd (BC1738).

Nesse problema, existem vários provedores de Internet, identificados por letras minúsculas de `a` à `z`. O problema nos dá um grafo descrevendo conexões entre diferentes localidades (vértices) e quais provedores oferecem essas conexões. Dado esse grafo, o problema nos pede para dizer quais provedores oferecem conectividade completa entre determinados pares de localidades (isto é, dados uma origem A e um destino B, é possível sair de A e chegar em B passando apenas por conexões desse provedor).

Dica: separar em vários grafos, um por provedor, e checar a conectividade usando a função `reachable()` desenvolvida no **Problema 2**.

Desafio 4 (Opcional): Resolva o problema *Network Delay* no LeetCode (LC743).

Nesse problema, dado um grafo que descreve uma rede com atrasos (*delays*) entre os seus nós, é pedido o tempo máximo em que uma mensagem enviada a partir do vértice de origem `k` levará para chegar em todos os vértices.

Esse problema pode ser resolvido pela aplicação direta do algoritmo de Dijkstra, implementado no desafio da Parte 1.