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

Aluno (a):			
Matrícula:		**Data:**	

Prática 10

Parte 0: Preparação

Passo 1: Crie um novo projeto chamado **Pratica10**.

Passo 2: Adicione os arquivos que acompanham a prática 10 ao projeto.

> **main.cpp**: contém a função main. Cria o grafo a partir das peças e verifica se formam um jogo.
>
> **graph.h** e **graph.cpp:** declaração e implementação do grafo, respectivamente.
>
> **list.h:** lista encadeada (c/ ponteiros) usada no grafo (lista de adjacência).
>
> **queue.h:** fila c/ ponteiros; deve ser usada na busca em largura.
>
> **heap.h:** fila de prioridades implementada como heap em array; usada para computar o menor caminho.

Passo 3: Compile e rode o código para verificar se não há erros.

> Nesse ponto deve apenas compilar e rodar mas não produzir resultados corretos; algumas funções não estão implementadas.

Passo 4: Estude o código para se familiarizar.

> Veja o material de aula se necessário. Faça comentários nos trechos mais complicados. Refatore o código se achar que vai ajudar seu trabalho. Teste novamente antes de fazer modificações mais profundas.

Parte 1: Implementando Busca em Profundidade

Passo 1: Implemente a função privada `Graph::DFS()` em **graph.cpp**.

> A função `dfs()` é a função pública chamada pelo usuário. Ela cria as estruturas auxiliares necessárias uma vez e repassa para a função `DFS()`, que realiza a busca de fato. Essa separação permite que a função `DFS()` se chame recursivamente, que é a implementação mais direta, sem ficar realocando as estruturas auxiliares (vetor de nós visitados `visited[]`) a cada chamada.
>
> Siga o pseudocódigo do material de aula para implementar `DFS()`. Ela recebe, além de `visited[]`, a lista (`result`) que deve contar ao final os nós na ordem que foram atravessados na busca em profundidade.

Parte 2: Implementando Busca em Largura

Passo 1: Implemente a função privada `Graph::BFS()` em **graph.cpp**.

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()`. Ela recebe o vetor `visited[]` e a lista (`result`) que deve contar ao final os nós na ordem que foram atravessados na busca em largura.

Parte 3: Extraindo o Menor Caminho

Passo 1: Estude o código da função `Graph::spf()` em **graph.cpp**.

Essa função computa o menor caminho como descrito no material de aula. Ao final, o vetor `dist[]` contém a menor distância do nó `src` para o todos os outros nós (ex.: `dist[x]` deve ser a menor distância de `src` até o nó `x`), e o vetor `prev[]` 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`).

Passo 2: Implemente a função `Graph::path()` que extrai o menor caminho.

Essa função recebe como parâmetro uma lista vazia (`result`) a qual deve conter ao final a sequência de nós que compõem o caminho de `src` até `dst` (na ordem natural). Lembre que ao seguir o vetor de nós anteriores `prev[]` é obtido o caminho na ordem inversa.