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 – INTRODUÇÃO

Aluno (a):			
Matrícula:		**Data:**	

Prática 09

Parte 0: Preparação

Passo 1: Crie um novo projeto chamado **Pratica9**.

Configure o projeto para utilizar C++11.
(Project Properties > C++ Build > Settings: no caso do Cygwin procure por "Dialect" e selecione "ISO C++11" em "Language standard")

Passo 2: Adicione os arquivos que acompanham a prática 9 ao projeto.

domino.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.

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 o Grafo

Passo 1: Implemente o construtor e o destrutor da classe Grafo em **graph.cpp**.

No código fornecido é criada e destruída a matriz de adjacência. Modifique se achar necessário caso adicione novas estruturas.

Passo 2: Implemente o método para adicionar arestas `edge()`.

Faça o método incrementar o contador de matriz de adjacência entre os vértices indicados.

Passo 3: Implemente a função que retorna o grau de um vértice `degree()`.

A função varre a matriz de adjacência na linha correspondente ao vértice e contabiliza quantas arestas estão conectadas a ele.

Parte 2: Testando a conectividade do Grafo

Passo 1: Implemente o método `connected()` em **graph.cpp**.

Essa função primeiro atribui a cada vértice um identificador do grupo que ele pertence. Um grupo é formado nós que possuem caminhos uns para os outros (são alcansáveis entre si). Inicialmente cada vértice pertence ao grupo identificado por ele mesmo.

Em seguida, para todas as arestas do grafo, é realizada a operação de união dos grupos dos vértices. Essa união significa que esses nós desses grupos são alcansáveis entre si e portanto são um mesmo grupo. A operação de união (`_union()`) dos vértices v1 e v2 faz com que v2 e todos os outros vértices com mesmo grupo de v2 passem a ter o grupo de v1 (poderia ser o contrário, desde ao final houvesse um grupo só).

Terminada a fase de união, basta verificar, cada todos os pares de nós (usados), se existe um caminho entre eles, isto é, se eles pentencem ao mesmo grupo. Para isso é usada a função `_find()`.

Passo 2: Compile e teste a aplicação.

Rode a aplicação, modificando as arestas (peças) criadas para ver se o resultado está correto.

Parte 3: (Desafio) Melhorando o desempenho

Passo 1: Melhorando a função `connected()`.

Outra forma de verificar, terminada a fase de união dos vértices, se eles estão conexos (isto é, pertencem ao mesmo grupo) é checar se ao final existe apenas um grupo no grafo. Modifique a função para fazer esse teste. Lembre de ignorar os nós que não foram usados.

Passo 2: Adicione ao grafo uma lista de arestas (*edges*).

O uso da matriz de adjacêndia tem suas vantagens mas traz problemas quando queremos percorrer as arestas do grafo, como ocorre na função `connected()`. Na implementação atual é preciso varrer as duas dimensões da matriz. Melhore a implementação adicionando uma lista de arestas (*edges*) que é atualizada sempre que uma nova aresta é adicionada, e faça uso dela na função `connected()`.