CI2 Parte prática. Duração: 1h30m

- **Submeter apenas a resolução da prova,** num **ficheiro `zip` (incluir ficheiros `*.cpp` e `*.h`).** O ficheiro `zip` não deve conter pastas. Não necessita incluir o ficheiro `tests.cpp`

s

- A resolução submetida será testada com um conjunto adicional de testes unitários, pelo que passar com sucesso os testes fornecidos não garante a cotação completa
- Se a resolução submetida passar apenas alguns dos testes, poderá obter cotação parcial na respetiva questão
- Em cada exercício é dada uma complexidade temporal esperada. Poderão existir testes para os quais um programa menos eficiente não passe em todos os testes

O teste prático irá consistir numa série de exercícios envolvendo grafos. Irá ser-lhe dada como base a classe de grafos introduzida nas aulas práticas:

```
class Graph {
    struct Edge {
        int dest;   // Destination node
        int weight; // An integer weight
    };

    struct Node {
        list<Edge> adj; // The list of outgoing edges (to adjacent nodes)
        bool visited;   // As the node been visited on a search?
    };

    int n;              // Graph size (vertices are numbered from 1 to n)
    bool hasDir;        // false: undirected; true: directed
    vector<Node> nodes; // The list of nodes being represented

    void dfs(int v);    // An example implementation of dfs
    void bfs(int v);    // An example implementation of bfs

public:
    // Constructor: nr nodes and direction (default: undirected)
    Graph(int nodes, bool dir = false);

    // Add edge from source to destination with a certain weight
    void addEdge(int src, int dest, int weight = 1);
};
```

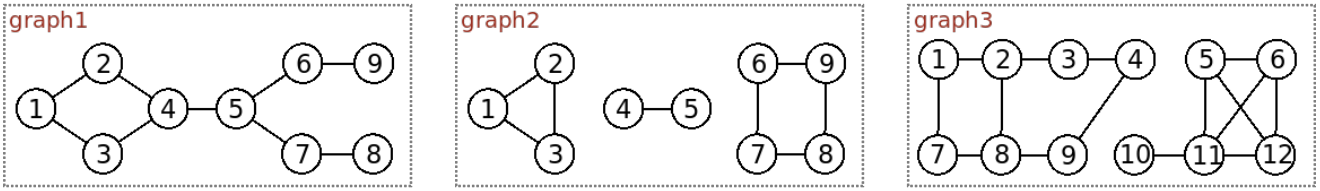
Para além da classe base é fornecido um exemplo de implementação de DFS e de BFS (tal como dado nas aulas), que poderá livremente usar e modificar em qualquer um dos exercícios do teste.

Não modifique o construtor e a função *addEdge*, pois estas funções serão usadas para dar o input ao seu código. Pode no entanto acrescentar novas funções e/ou variáveis.

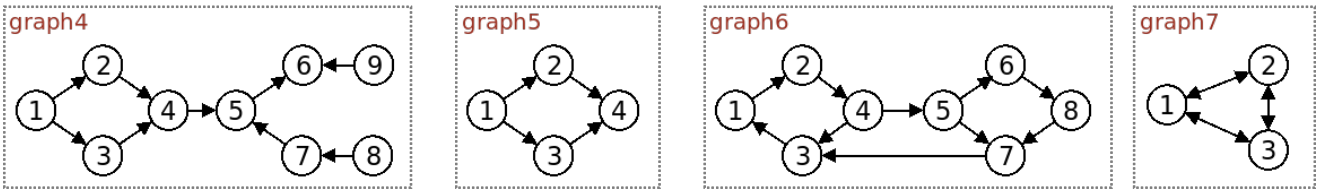
Nas complexidades esperadas, $|V|$ é o número de nós e $|E|$ é o número de arestas.

A classe ***FunWithGraphs*** contém alguns grafos "prontos a usar" e que são usados nos testes unitários exemplo deste teste prático. Para facilitar a sua tarefa aula pode ver aqui as suas ilustrações:

Alguns grafos não dirigidos e não pesados




Alguns grafos dirigidos e não pesados




**1) [3.2 valores] Ausência de ligação.** Implemente a seguinte função no ficheiro ***graph.cpp***:

```
bool Graph::disconnected(int u, int v)
```

**Complexidade temporal esperada:** $\mathcal{O}(|V|)$

Deve devolver *true* se não existir uma aresta entre os nós $u$ e $v$ ou *false* caso exista. Recorde que uma aresta é uma ligação direta entre os dois nós. Pode assumir que $u$ e $v$ serão índices válidos.

*Exemplo de chamada e output esperado:*

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.disconnected(1, 2) << endl;
cout << g1.disconnected(1, 4) << endl;
cout << g1.disconnected(3, 1) << endl;
Graph g4 = FunWithGraphs::graph4();
cout << g4.disconnected(2, 4) << endl;
cout << g4.disconnected(4, 2) << endl;
```

```
0
1
0
0
1
```

*Explicação: graph1 – existem as arestas (1,2) e (3,1); não existe a aresta (1,4)*
*graph4 – existe a aresta (2,4) mas não existe a aresta (4,2)*

**2) [3.2 valores] Maior grau.** Implemente a seguinte função no ficheiro ***graph.cpp***:

```
vector<int> Graph::largestDegree()
```

**Complexidade temporal esperada:** $\mathcal{O}(|V|)$

Deve devolver um *vector* contendo os índices dos nós de maior grau, ou seja, os nós que tenham grau máximo. Recorde que o grau é o número de vizinhos de um nó. Esta função será chamada apenas para grafos não dirigidos. Os nós podem vir por qualquer ordem no *vector* devolvido pela função.

*Exemplo de chamada e output esperado:*

```
Graph g1 = FunWithGraphs::graph1();
auto v1 = g1.largestDegree();
for (auto v : v1) cout << v << " ";
cout << endl;
Graph g2 = FunWithGraphs::graph2();
auto v2 = g2.largestDegree();
for (auto v : v2) cout << v << " ";
cout << endl;
```

```
4 5
1 2 3 6 7 8 9
```

*Explicação: graph1 – os nós 4 e 5 têm o grau máximo, que é 3*
*graph2 – os nós 1, 2, 3, 6, 7, 8 e 9 têm o grau máximo, que é 2*

**3) [3.2 valores] Componentes conexos.** Implemente a seguinte função no ficheiro ***graph.cpp***:

```
bool Graph::connected()
```

**Complexidade temporal esperada:** $\mathcal{O}(|V| + |E|)$

Deve devolver *true* se o grafo for conexo (isto é, se tiver um único componente conexo) ou *false* caso contrário. Esta função será chamada apenas para grafos não dirigidos.

*Exemplo de chamada e output esperado:*

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.connected() << endl;
Graph g2 = FunWithGraphs::graph2();
cout << g2.connected() << endl;
```

```
1
0
```

*Explicação: graph1 – é conexo*
*graph2 – não é conexo (tem 3 componentes conexos)*

**4) [3.2 valores] Nós a uma certa distância.** Implemente a seguinte função no ficheiro ***graph.cpp***:

```
int Graph::countNodes(int v, int k)
```

**Complexidade temporal esperada:** $\mathcal{O}(|V| + |E|)$

Deve devolver a quantidade de nós que estão a distância $k$ do nó $v$. Relembre que a distância entre dois nós é o tamanho (quantidade de arestas) do menor caminho entre eles. Pode assumir que $v$ é um índice válido.

*Exemplo de chamada e output esperado:*

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.countNodes(5, 1) << endl;
cout << g1.countNodes(6, 2) << endl;
cout << g1.countNodes(1, 3) << endl;
```

```
3
2
1
```

*Explicação: graph1 – o nó 5 tem 3 nós a distância 1: os nós 4, 6 e 7*
*o nó 6 tem 2 nós a distância 2: os nós 4 e 7*
*o nó 1 tem 1 nó a distância 3: o nó 5*

**5) [3.2 valores] Caminhos.** Implemente a seguinte função no ficheiro ***graph.cpp***:

```
bool Graph::pathExists(int u, int v, const vector<int> & avoid)
```

**Complexidade temporal esperada:** $\mathcal{O}(|V| + |E|)$

Deve devolver *true* se existir pelo menos um caminho entre os nós $u$ e $v$ que não passa por nenhum dos nós com índices no vector *avoid*, ou *false* caso contrário. Note que grafo original não deve ser modificado, pois podem ser feitas várias chamadas ao método. Pode assumir que $u$, $v$ e $avoid$ têm índices válidos.

*Exemplo de chamada e output esperado:*

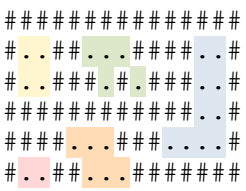
```
Graph g1 = FunWithGraphs::graph1();
cout << g1.pathExists(1, 5, {2}) << endl;
cout << g1.pathExists(1, 5, {2,3}) << endl;
cout << g1.pathExists(2, 6, {4}) << endl;
cout << g1.pathExists(2, 6, {9,8}) << endl;
```

```
1
0
0
1
```

*Explicação: graph1 – existe um caminho entre os nós 1 e 5 que não passa pelo nó 2 (1→3→4→5)*
*não existe um caminho entre os nós 1 e 5 que não passe pelos nós 2 e 3*
*não existe um caminho entre os nós 2 e 6 que não passe pelo nó 4*
*existe um caminho entre os nós 2 e 6 que não passa pelos nós 9 e 8 (2→4→5→6)*

**As duas perguntas finais (6 e 7) não envolvem grafos explícitos e valem menos, pelo que aconselhamos que só as tentem fazer depois das outras perguntas estarem feitas.**

Para as perguntas 6 e 7 considere um mapa 2D representado por uma matriz de caracteres $m$ de dimensões $rows \times cols$ onde uma célula $'.'$ representa *água* e $'\#'$ e representa *terra*. Duas células são consideradas vizinhas se forem adjacentes vertical, horizontal ou diagonalmente. Um **lago** é um conjunto de células adjacentes. Por exemplo, o mapa na figura do lado direito tem 5 lagos (representados com cores diferentes).




**6) [2.0 valores] Quantidade de lagos.** Implemente a seguinte função no ficheiro ***funWithGraphs.cpp***:

```
static int FunWithGraphs::countLakes(int rows, int cols, const string m[])
```

**Complexidade temporal esperada:** $\mathcal{O}(rows \times cols)$

Deve devolver a quantidade de lagos que existe no mapa 2D representado em $m$ tal como atrás descrito.

*Exemplo de chamada e output esperado:*

```
string m[] = {"###############",
              "#..##...####..#",
              "#..###.#.###..#",
              "############..#",
              "####...###....#",
              "#..##...#######"};
cout << FunWithGraphs::countLakes(6, 15, m) << endl;
```

```
5
```

*Explicação: é a imagem da figura anterior, com os 5 lagos aí representados*

7) **[2.0 valores] Menos lagos!** Implemente a seguinte função no ficheiro ***funWithGraphs.cpp***:

```
static int FunWithGraphs::reduceLakes(int rows, int cols, const string m[], int k)
```

**Complexidade temporal esperada:** $\mathcal{O}(rows \times cols \times \log(rows \times cols))$

Suponha que pode criar novos terrenos, transformando células de água em células de terra. Se quiser que o mapa representado por $m$ passe a conter apenas $k$ lagos, qual é o mínimo de células que tem de transformar? Pode assumir que $k$ será um número maior ou igual a zero e menor do que a quantidade de lagos inicial.

*Exemplo de chamada e output esperado:*

```
string m[] = {"################",
              "#..##...####..#",
              "#..###.#.###..#",
              "############..#",
              "####...###....#",
              "#..##...#######"};
cout << FunWithGraphs::countLakes(6, 15, m, 3) << endl;
```

```
6
```

*Explicação: basta transformar 6 células (as 4 do lago amarelo e as 2 do lago vermelho) para passarmos a ter apenas 3 lagos (a verde, a laranja e a azul)*

- **Submeter apenas a resolução da prova,** num **ficheiro `zip` (incluir ficheiros `*.cpp` e `*.h`).**
  O ficheiro `zip` não deve conter pastas. Não necessita incluir o ficheiro `tests.cpp`