9ªaula prática - Introducão a grafos e a pesquisa em profundidade (DFS) e largura (BFS)

Instruções

- Faça download do ficheiro ***aed2122_p09.zip*** da página da disciplina e descomprima-o (contém a pasta ***lib,*** a pasta ***Tests*** com os ficheiros ***funWithGraphs.cpp, funWithGraphs.h, graph.cpp, graph.h*** e ***tests.cpp***, e os ficheiros ***CMakeLists*** e ***main.cpp***)
- Se desejar pode criar métodos e/ou classes auxiliares para resolver estes exercícios

1. Introdução a uma classe simplificada de grafos

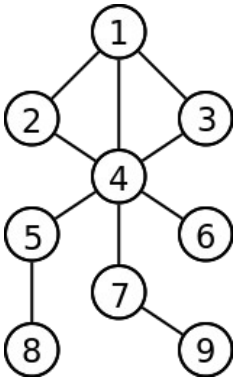
Os grafos são uma maneira muito rica e flexível de modelar sistemas reais ou artificiais e com a qual podemos modelar entidades (os nós ou vértices) e as relações entre elas (as ligações ou arestas). O objetivo deste exercício é introduzir a classe de grafos simplificada que foi dada nas aulas teóricas e que servirá de base para as implementações de quase todos os algoritmos de grafos que serão dados nesta UC.

```
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: undirect; true: directed
    vector<Node> nodes; // The list of nodes being represented

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);
};
```

A tua primeira tarefa é espreitar a implementação (ficheiros ***graph.h*** e ***graph.cpp)*** e garantir que percebes na sua essência o que está a ser feito (nesta alínea ainda não precisas de ver as funçoes *dfs* e *bfs*).

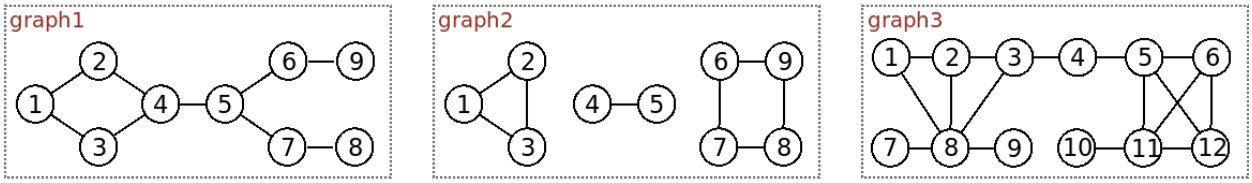
Por exemplo, o código seguinte serviria para criar o grafo **g** representado na figura do lado direito.

```
Graph g(9, false); // 9 nodes, undirected graph
g.addEdge(1, 2);   // assuming weight=1 (unweighted)
g.addEdge(1, 3);
g.addEdge(1, 4);
g.addEdge(2, 4);
g.addEdge(3, 4);
g.addEdge(4, 5);
g.addEdge(4, 6);
g.addEdge(4, 7);
g.addEdge(5, 8);
g.addEdge(7, 9);
```

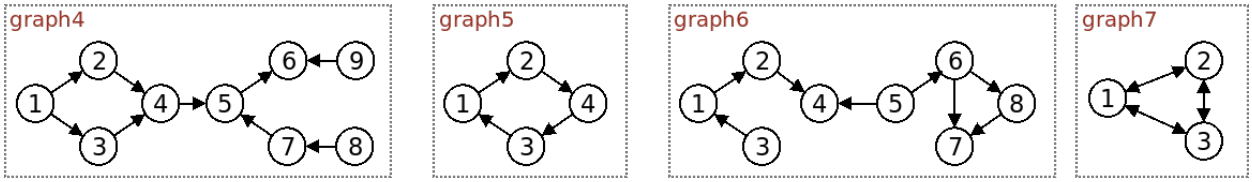


A classe ***FunWithGraphs*** contém alguns grafos "prontos a usar" e que são usados nos testes unitários desta aula (os grafos são não pesados mas nas próximas aulas iremos usar pesos). Para facilitar a tua tarefa nesta aula podes ver aqui ilustrações de todos os grafos lá colocados:

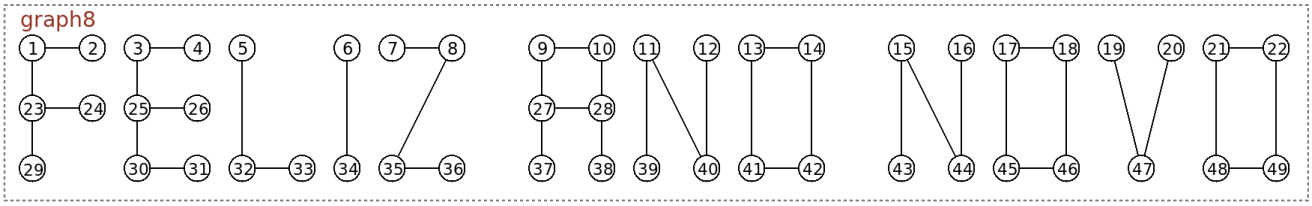
Alguns grafos não dirigidos e não pesados



Alguns grafos dirigidos e não pesados



Um grafo... especial!



A alínea seguinte destina-se a garantir que percebeu a implementação de grafos dada e pede-lhe para implementar uma pequena função.

a) Devolvendo o grau. Implementa a seguinte função no ficheiro ***graph.cpp***:

```
int Graph::outDegree(int v)
```

Complexidade temporal esperada: $\mathcal{O}(1)$

Deve devolver o grau de saída do nó *v*, ou -1 se o indíce do nó dado não for válido (ou seja, se não estiver entre 1 e |V|). Recorda que o grau é o número de ligações que partem do nó *v*.

Exemplo de chamada e output esperado:

```
Graph g = FunWithGraphs::graph1();
cout << g.outDegree(1) << endl;
cout << g.outDegree(4) << endl;
cout << g.outDegree(9) << endl;
cout << g.outDegree(10) << endl;
```

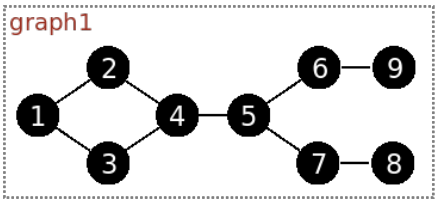
```
2
3
1
-1
```

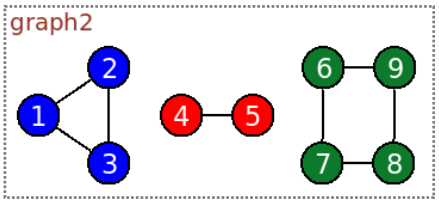
Explicação: O grau do nó 1 é 2, o grau do nó 4 é 3, o grau do nó 9 é 1 e o nó 10 não existe

Sugestão: basta ver o tamanho da lista de adjacências do nó (e ter o cuidado de verificar se o nó existe).

2. Componentes conexos.

Recorda que um *componente conexo* de um grafo não dirigido é um subgrafo em que um qualquer par de vértices tem um caminho entre si. Por exemplo, *graph1* tem apenas um componente conexo (indicado a preto), ao passo que *graph2* tem 3 componentes conexos indicados a azul, vermelho e verde.





Nota: para grafos dirigidos existe o conceito de componentes fortemente conexos de que falaremos numa outra aula

a) Contando componentes conexos. Implementa a seguinte função no ficheiro ***graph.cpp***:

```
int Graph::connectedComponents()
```

Complexidade temporal esperada: $\mathcal{O}(|V| + |E|)$
(onde |V| é o número de nós e |E| o número de ligações)

Deve devolver o número de componentes conexas do grafo.

Exemplo de chamada e output esperado:

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

```
1
3
```

Explicação: São os dois grafos da figura de cima.

Sugestão: fazer uma pesquisa em profundidade (**DFS)** a partir de cada nó ainda não visitado e contar quantas vezes uma nova pesquisa foi iniciada; se ainda não o fez esta é uma excelente altura para ver a teórica nº 17 (o slide 15 contém literalmente a solução para este exercício…).

b) Componente gigante. Implementa a seguinte função no ficheiro ***graph.cpp***:

```
int Graph::giantComponent()
```

Complexidade temporal esperada: $\mathcal{O}(|V| + |E|)$
(onde |V| é o número de nós e |E| o número de ligações)

Deve devolver o tamanho (número de nós) do maior componente conexo do grafo, ou seja, aquele que tem uma maior quantidade de nós.

Exemplo de chamada e output esperado:

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

```
9
4
```

Explicação: graph1 só tem um componente de 9 nós; *graph2* tem 3 componentes de tamanhos 3, 2 e 4

Sugestão: aproveitando o código do exercício anterior basta agora extender para também contar os nós; uma hipótese é colocar o *dfs* a devolver o número de nós do componente (seria demais dar também aqui a solução completa e já implementada para esta alínea, não acham?☺).

3. **Ordenação Topológica.** Função a implementar:

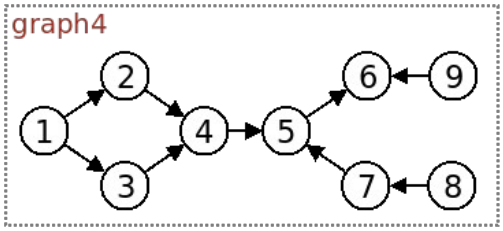
```
list<int> Graph::topologicalSorting()
```

Complexidade temporal esperada: $\mathcal{O}(|V| + |E|)$
(onde |V| é o número de nós e |E| o número de ligações)

Deve devolver uma ordenação topológica do grafo na forma de uma lista de tamanho |V|. Se existir mais do que uma ordenação, qualquer uma será aceite pelos testes unitários.

Recorda que uma *ordenação topológica* de um grafo dirigido acíclico (um *DAG*) é uma ordem dos nós tal que para qualquer aresta *(u,v)* o nó *u* aparece antes do nó *v* na ordem. Por exemplo, se o grafo indicar precedências de tarefas (onde uma aresta *(u,v)* indica que a aresta a tarefa *u* é uma precedência da tarefa *v)*, então uma ordenação topológica é uma possível ordem pela qual podemos fazer as tarefas respeitando sempre as precedências. Consegues perceber porque não podem existir ciclos?

A seguinte imagem ilustra três possíveis ordenações topológicas de *graph4* (qualquer uma seria aceite).



Algumas ordenações topológicas possíveis:

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

Exemplo de chamada e output esperado:

```
Graph g4 = FunWithGraphs::graph4();
list<int> order = g4.topologicalSorting();
cout << "Order: ";
for (auto v : order ) cout << " " << v;
cout << endl;
```

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

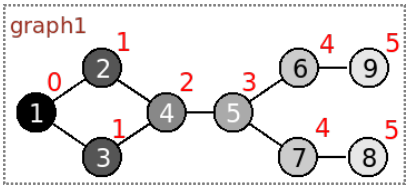
Explicação: é a terceira ordem ilustrada na figura de cima.

Sugestão: Qual a relação de uma ordenação topológica com a ordem em que os nós são visitados num **DFS**? Ao fazer um *dfs(v)*, todos os nós visitados nessa chamada terão de aparecer… depois de *v*. Veja a teórica nº 17 (slides 17 a 19 para ver pseudo-código e "visualização" de uma execução).

Nota: Existem mais algoritmos possíveis para fazer uma ordenação topológica. Por exemplo poderia começar por incluir todos os nós com grau de entrada 0 (sem precedências), retirá-los do grafo e voltar a repetir o processo (uma implementação "naive" desta estratégia qual complexidade temporal?)

4. Distâncias em grafos não pesados.

Recorda que a *distância* entre dois nós num grafo não pesado é dada pelo número de ligações no menor caminho que liga dois nós. Por exemplo, no grafo da direita estão indicados a vermelho as distâncias do nó 1 a todos os outros nós:



O *diâmetro* de um grafo é a maior distância entre um par de nós, ou seja, o mais comprido menor caminho entre dois nós. Por exemplo, no grafo de cima o diâmetro é precisamente 5 (a distância de 1 a 9 ou de 1 a 8; nenhum outro par de nós está mais distante).

a) Distância entre dois nós. Implementa a seguinte função no ficheiro ***graph.cpp***:

```
int Graph::distance(int a, int b)
```

Complexidade temporal esperada: $\mathcal{O}(|V| + |E|)$
(onde |*V*| é o número de nós e |E| o número de ligações)

Deve devolver a distância entre o nó *a* e o nó *b*. Se não existir caminho entre *a* e *b* deve devolver -1.

Exemplo de chamada e output esperado:

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.distance(1, 1) << endl;
cout << g1.distance(1, 5) << endl;
cout << g1.distance(9, 8) << endl;
Graph g2 = FunWithGraphs::graph2();
cout << g2.distance(1, 2) << endl;
cout << g2.distance(1, 4) << endl;
```

```
0
3
4
1
-1
```

Explicação: graph1 é o de cima e as distâncias de todos os nós a 1 já estão representadas; do nó 9 ao 8 a distância é; em *graph2* 1 é adjacente a 2 e não tem caminho para 4.

Sugestão: fazer uma pesquisa em largura **(BFS)** a partir do nó *a*, que percorre os nós por ordem crescente de distância ao nó *a*; se ainda não o fez esta é uma excelente altura para ver a teórica nº 18 (os slides 6 a 8 explicam a pesquisa em largura para cálculo de distâncias e incluem pseudo-código e "visualização" de uma execução).

b) Diâmetro. Implementa a seguinte função no ficheiro ***graph.cpp***:

```
int Graph::diameter()
```

Complexidade temporal esperada: $\mathcal{O}(|V| \times (|V| + |E|))$
(onde |*V*| é o número de nós e |E| o número de ligações)

Deve devolver o diâmetro do grafo. Se existir mais do que um componente conexo deve devolver -1.

Exemplo de chamada e output esperado:

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

```
5
-1
```

Explicação: graph1 tem diâmetro 5; graph2 tem mais do que um componente conexo;

Sugestão: aproveitando o código do exercício anterior basta agora começar um BFS a partir de cada nó do grafo e guarda a maior distância que aparecer (e se não existir caminho o que devolve o BFS anterior?)

5. *(exercício extra)* **To or not to be… a DAG!** Função a implementar:

```
bool Graph::hasCycle()
```

Complexidade temporal esperada: $\mathcal{O}(|V| + |E|)$
(onde |V| é o número de nós e |E| o número de ligações)

Deve devolver *true* se o grafo tiver pelo menos um ciclo e *false* caso contrário. Podes assumir que o teu programa só será testado com grafos dirigidos. Recorda que um *ciclo* é um caminho de comprimento maior ou igual a um que começa e termina no mesmo nó.

Exemplo de chamada e output esperado:

```
Graph g4 = FunWithGraphs::graph4(); cout << g4.hasCycle() << endl;
Graph g5 = FunWithGraphs::graph5(); cout << g5.hasCycle() << endl;
```

```
false
true
```

Explicação: graph4 é acíclico; *graph5* tem um ciclo.

Sugestão: use o algoritmo explicado nas aulas teóricas (aula nº 17, slides 20 a 23)

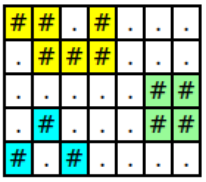
(exercícios extra) **Mais ideias para exploração:**

Se te sentires com vontade de continuar a explorar este tema, ficam aqui duas sugestões possíveis:

- Implementar DFS e/ou BFS para uma grid 2D

No contexto desta aula apenas usamos grafos *explícitos*. Experimente implementar DFS e BFS para uma matriz 2D, como explicado nas aulas (ver slide 16 da aula nº 17 e slide 9 da aula nº 18) e por exemplo:

- Conte número de componentes conexos numa matriz (supondo que duas células são adjacentes se forem vizinhas horizontal, vertical ou diagonalmente). Por exemplo, existem 3 componentes conexos indicados por cores diferentes na figura da direita (onde # indica uma célula ocupada e . uma célula vazia). [ver o que é um flood fill]



- Veja a distância entre duas células numa matriz. Por exemplo, a célula marcada com a letra A na figura da direita está à distância 8 da célula marcada com a letra B. Pode usar um BFS a começar numa das células.

```
########                        ########
#A.....#                        #A12345#
####.###          --->          ####4####
#B.....#    BFS starting in A   #876567#
########                        ########
```

- Implementar uma pesquisa para um jogo

Implemente uma solução para o jogo explicado nas aulas (slides 13 e 14 da aula nº 18) que envolve saber o mínimo número de movimentos para transformar uma posição do tabuleiro noutra posição alvo. Use BFS a partir da posição inicial. Se quiser ver o enunciado do problema (e até submetê-lo) pode espreitar por exemplo no PEG Judge.