
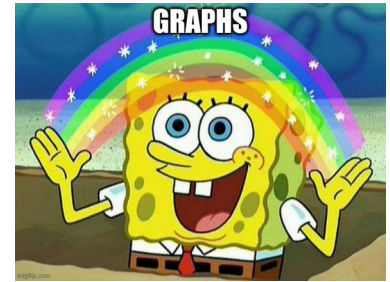


10ª Aula Prática

Introdução a Grafos e a Pesquisa em Profundidade

Instruções

- Faça download do ficheiro *aed2223_p10.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*)
- No CLion, abra um projeto, seleccionando a pasta que contém os ficheiros do ponto anterior
- A partir desta aula volta a ser possível submeter os exercícios na plataforma Mooshak. 



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

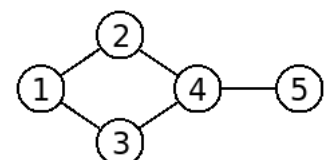
public:
    // Constructor: nr nodes and direction (default: undirected)
    Graph(int nodes, bool dir = false);

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

A sua primeira tarefa é espreitar a implementação (ficheiros *graph.h* e *graph.cpp*) e garantir que percebe na sua essência o que está a ser feito (nesta alínea ainda não precisa de ver a função *dfs*).

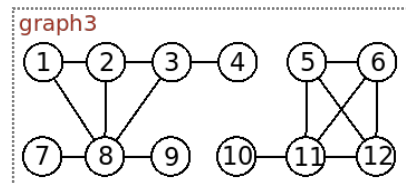
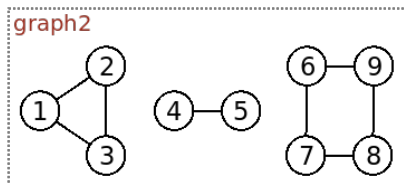
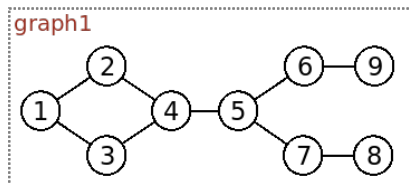
Por exemplo, o código seguinte serviria para criar o grafo **g** representado na figura do lado direito.

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

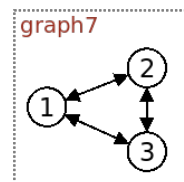
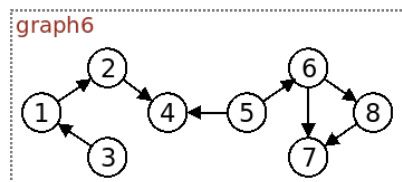
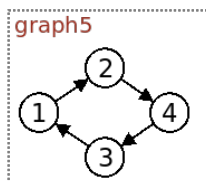
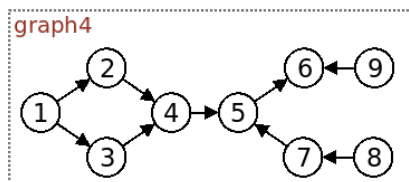


A classe **FunWithGraphs** contém alguns grafos “prontos a usar” e que são usados nos testes unitários desta aula. Para facilitar a sua tarefa, pode ver aqui ilustrações de todos os grafos lá colocados:

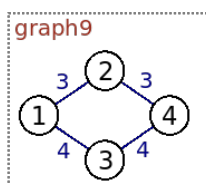
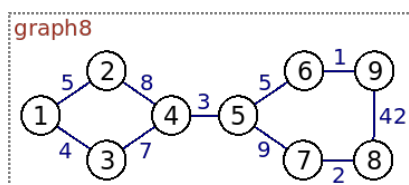
Alguns grafos não dirigidos e não pesados



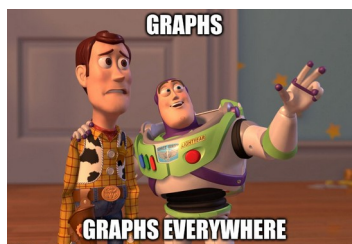
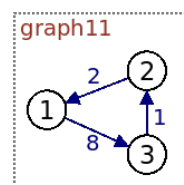
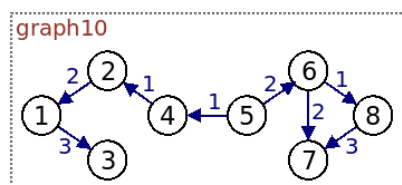
Alguns grafos dirigidos e não pesados



Alguns grafos não dirigidos e pesados



Alguns grafos dirigidos e pesados



As duas alíneas que se seguem destinam-se a garantir que percebeu a implementação de grafos dada.

a) Devolvendo o grau. Implemente 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 índice 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; 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).

b) Devolvendo o grau... pesado. Implemente a seguinte função no ficheiro *graph.cpp*:

```
int Graph::weightedOutDegree(int v)
```

Complexidade temporal esperada: $\mathcal{O}(\text{outDegree}(v))$

Deve devolver grau pesado do nó v , ou seja o somatório dos pesos das ligações que saem do nó v para um outro nó. Se o nó dado não for válido (ou seja, se não estiver entre 1 e $|V|$), deve devolver -1.

Exemplo de chamada e output esperado:

```
Graph g = FunWithGraphs::graph8();
cout << g.weightedOutDegree(1) << endl;
cout << g.weightedOutDegree(4) << endl;
cout << g.weightedOutDegree(9) << endl;
cout << g.weightedOutDegree(10) << endl;
```

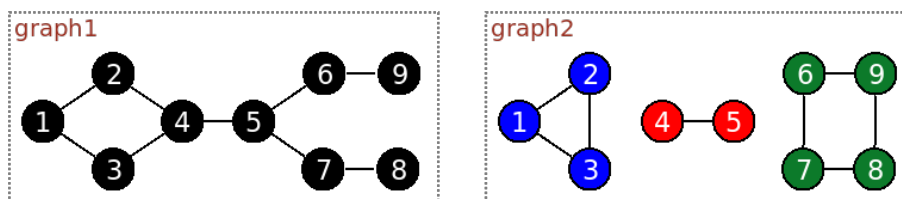
```
9
18
43
-1
```

Explicação: O grau pesado do nó 1 é $5+4=9$; o grau pesado do nó 4 é $8+7+3=15$; o grau pesado do nó 9 é $1+42=43$; o nó 10 não existe.

Sugestão: basta percorrer a lista de adjacências do nó v e ir somando os pesos das arestas

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. Implemente 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 (pode assumir que é um grafo não dirigido).

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 os slides 11 a 15 do *Capítulo 14 – Grafos: Pesquisa em Profundidade* (o slide 15 contém literalmente a solução para este exercício – cuidado ao submeter no Mooshak para não imprimir nada que não seja pedido).

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 estender 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. Implemente a seguinte função no ficheiro *graph.cpp*:

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

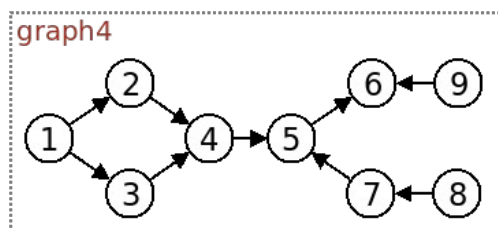
Complexidade temporal esperada: $\mathcal{O}(|V| + |E|)$

(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. Pode assumir que o grafo dado não contém nenhum ciclo.

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 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. Consegue 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 os slides do *Capítulo 14 – Grafos: Pesquisa em Profundidade* (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. To be or not to be... a DAG! Implemente a seguinte função no ficheiro *graph.cpp*:

```
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. Pode assumir que o seu programa só será testado com grafos dirigidos. Recorde que um *ciclo* é um caminho de tamanho maior ou igual a um que começa e termina no mesmo nó. DAG é o acrónimo de “*directed acyclic graph*”, ou seja, um grafo dirigido sem ciclos.

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.

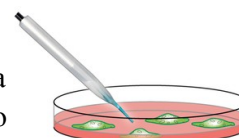
Sugestão: use o algoritmo explicado no *Capítulo 14 – Grafos: Pesquisa em Profundidade* (slides 20 a 23)

Exercício Extra

O exercício seguinte não usa explicitamente grafos mas permite-lhe consolidar os seus conhecimentos de pesquisa em profundidade e equivale a uma pesquisa num grafo implícito.

5. Contagem de Células (DFS e componentes conexos numa grelha 2D)

O Pedro tem um trabalho laboratorial de Biologia e precisa da tua ajuda. Ele está a cultivar pequenos micróbios numa caixa de Petri e necessita de os observar ao microscópio para perceber qual é a maior micróbio visível.



A caixa de Petri pode ser considerada como uma grelha 2D, ou seja uma matriz, onde em cada posição pode ou não existir uma célula. Duas células estão ligadas se forem adjacentes vertical, horizontal ou diagonalmente. Um micróbio é um conjunto de células ligadas. Por exemplo, a caixa de Petri seguinte tem exactamente três micróbios ('.' é uma posição vazia, '#' é uma posição com uma célula):

O tamanho de um micróbio é igual ao número de células que o constitui. Na figura do lado direito, os 3 micróbios têm tamanho 6 (o amarelo), 3 (o azul) e 4 (o verde). O maior micróbio é o de maior tamanho. Neste caso o maior micróbio é o amarelo.

#	#	.	#	.	.
.	#	#	#	.	.
.	.	.	.	#	#
.	#	.	.	#	#
#	.	#	.	.	.

Função a implementar no ficheiro *funWithGraphs.cpp*:

```
static int largestMicrobe(int rows, int cols, string m[])
```

Complexidade temporal esperada: $\mathcal{O}(R \times C)$

(onde R é rows, o nº de linhas e C é cols, o nº de colunas, ou seja, o algoritmo deve ser linear no nº de células)

Dado uma matriz de caracteres de dimensão $rows \times cols$ indicando o conteúdo de uma caixa de Petri, deve devolver qual o tamanho do maior micróbio, ou seja, qual o **tamanho do maior conjunto conexo de células** na matriz (nos testes feitos ao seu programa a matriz não será maior que 100×100).

Exemplo de chamada e output esperado:

```
string m[5] = {"##.#...",
               ".###...",
               "...##",
               ".#...##",
               "#.#..."};
cout << FunWithGraphs::largestMicrobe(5, 7, m) << endl;
```

6

Explicação: O caso de exemplo corresponde à figura do enunciado.

Sugestão: a matriz é como se fosse um grafo implícito onde cada célula é um nó ligado às 8 células adjacentes e o problema é na sua essência o mesmo que a alínea 2.b, ou seja, pede o maior componente conexo; fazer um DFS numa grid 2D é conhecido como fazer um “[flood fill](#)”.

Exercício de Desafio

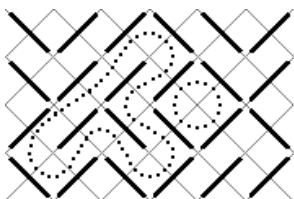
(exercício mais difícil para alunos que querem ter um desafio adicional)

6. Labirinto de Barras. (exercício baseado num problema do <https://onlinejudge.org/>)

Já reparaste que se pegares num caderno quadriculado e o encheres com barras (“\” e “/”) consegues criar um labirinto muito engraçado?



Repara na seguinte figura:



Como podes ver, caminhos em labirintos como este não podem ter ramificações (dividirem-se em dois ou mais ramos) e como tal todos os caminhos dentro do labirinto só podem ser cíclicos ou então começarem num sítio e acabarem noutra.

Neste problema estamos apenas interessados nos caminhos cíclicos. Recorda que um caminho é cíclico se for possível começar num ponto e voltar a ele mesmo através desse caminho.

Função a implementar no ficheiro *funWithGraphs.cpp*:

```
static pair<int, int>slashMaze(int rows, int cols, string m[])
```

Deve devolver um par contendo o **número de ciclos** e qual o **tamanho do maior deles**. O tamanho de um ciclo é definido como o número de pequenos quadrados (os que estão delimitados na figura por traços cinzentos) que o ciclo contém. Por exemplo, na figura, temos dois ciclos, um com tamanho 16 e outro com tamanho 4. É garantido que existe pelo menos um ciclo nos casos dados ao seu programa e que a matriz dada não excede as dimensões de 80×80 .

Exemplo de chamada e output esperado:

```
// R"()" is a raw literal (avoiding having to escape the backslash)
string m[4] = {R"(\//\//)",
               R"(\//\//)",
               R"(/\\//\\)",
               R"(\//\//)"};
auto answer = FunWithGraphs::slashMaze(4, 6, m);
cout << answer.first << " " << answer.second << endl;
```

2 16

Explicação: é o exemplo da figura, uma matriz de 4×6 com 2 ciclos, o maior com tamanho 16

Como é um desafio não vamos para já dar mais pistas mas podem entrar em contacto com @Pedro Ribeiro no Slack para pedir dicas.