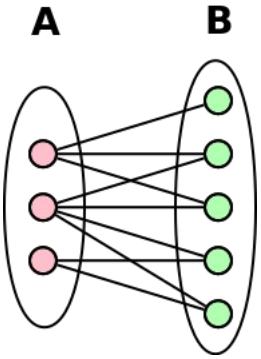
11ª Aula Prática - Pesquisa em Profundidade Avançada

Instruções

- Faça download do ficheiro ***aed2223_p11.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, selecionando a pasta que contém os ficheiros do ponto anterior
- Note que pode submeter os exercícios desta aula na plataforma Mooshak: https://mooshak.dcc.fc.up.pt/~aed/

Esta aula é uma continuação da aula prática anterior sobre pesquisa em profundidade. Deverá começar por fazer os exercícios base da aula anterior (exercícios 1 a 4 da aula #10) e só depois deverá tentar os exercícios propostos nesta aula #11.

1. DFS e grafos bipartidos



Vamos começar com um problema de pesquisa em profundidade (DFS) *"ad-hoc"* para novamente ilustrar o seu potencial e flexibilidade.

Um **grafo bipartido** é um grafo onde os nós podem ser divididos em duas partições *A* e *B* tal que apenas existem ligações de nós entre *A* e *B* e nunca de *A* para *A* ou de *B* para *B*, tal como ilustrado na figura da direita. Um exemplo seria algo como participações de atores em filmes, sendo que atores participam em filmes, mas atores não participam em atores e filmes não participam em filmes.

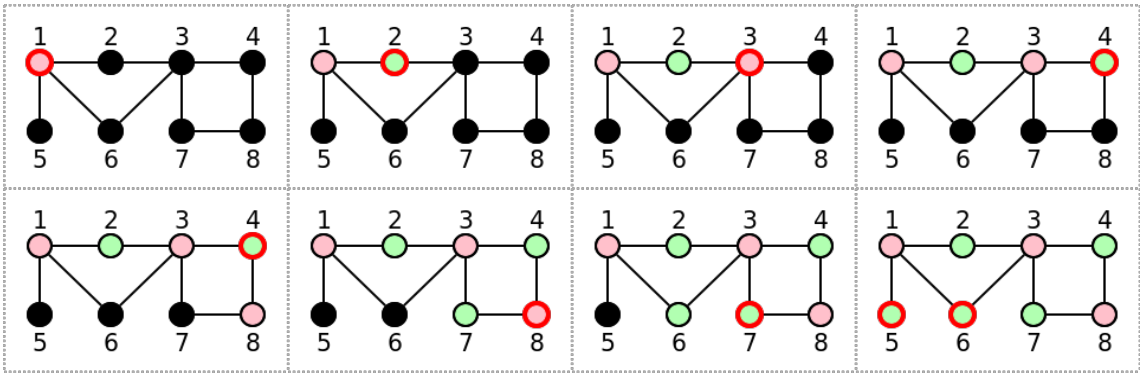
O objetivo deste exercício é escrever uma função que dado um grafo descobre se ele é bipartido, ou seja, se pode ser dividido em duas partições *A* e *B* como atrás descrito.

Um algoritmo possível para resolver este problema é o seguinte:

- começar um *dfs(v)* num nó *v* e "pintar" esse nó com cor *c*
- para cada nó *w* vizinho de *v*, recursivamente fazer *dfs(w)* e pintar com cor diferente de *c*
- se *w* já foi visitado, então verificar se a cor com que foi pintado é diferente de *c*
  - se a cor for a mesma então o grafo não pode ser bipartido
- se chegarmos ao fim sem nenhuma "prova" contra (dois vizinhos da mesma cor) então esse componente conexo é bipartido

De maneira mais visual, aqui fica um exemplo para um grafo que de facto é bipartido:

- começamos com *dfs(1)* e pintamos o nó de "vermelho" (partição *A*)
  - isto implica que para ser bipartido os seus vizinhos (2, 5 e 6) devem ser "verdes" (partição *B*)
- iniciamos chamadas recursivas aos seus vizinhos e fazemos *dfs(2)* pintando de verde
  - para ser bipartido os seus vizinhos (1 e 3) devem ser "vermelhos" (partição *A*)
- (…)
- no final podemos concluir que o grafo é bipartido, com A={1,3,8} e B={2,4,5,6,7}



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

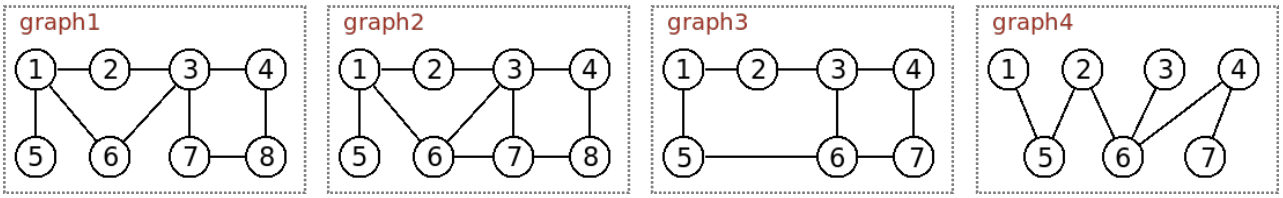
```
bool Graph::bipartite()
```

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

Deve devolver true se o grafo for bipartido e false caso contrário. Pode assumir que nos testes feitos ao seu programa apenas serão usados grafos não dirigidos, não pesados e conexos.

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 dos grafos relevantes para este exercício.

Alguns grafos não dirigidos e conexos para o exercício 1



Exemplo de chamada e output esperado:

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.bipartite() << endl;
Graph g2 = FunWithGraphs::graph2();
cout << g2.bipartite() << endl;
Graph g3 = FunWithGraphs::graph3();
cout << g3.bipartite() << endl;
Graph g4 = FunWithGraphs::graph4();
cout << g4.bipartite() << endl;
```

```
true
false
false
true
```

Explicação: graph1 é bipartido (é o grafo do exemplo); graph2 não é bipartido (basta ver por exemplo que o triângulo 3,6,7 não tem coloração possível onde vizinhos têm cor diferente); graph3 não é bipartido (1 e 3 têm de ser da mesma cor, por serem ambos vizinhos de 2, sendo que 5 e 6 deveriam ser ambos da cor oposta a 1 e 3, por serem seus vizinhos, mas têm uma ligação o que impede uma coloração possível); graph4 é bipartido (podemos partir em duas partições {1,2,3,4} e {5,6,7}).

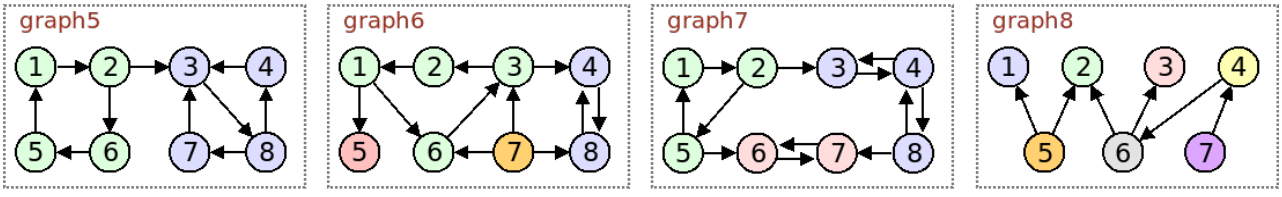
Sugestão: use o algoritmo atrás descrito.

2. Componentes fortemente conexos.

Recorde que um *componente fortemente conexo* de um **grafo dirigido** é um subgrafo maximal em que um qualquer par de vértices tem um caminho entre si. A figura de baixo ilustra vários grafos e seus respetivos componentes fortemente conexos (indicados pelas cores).

- *graph5* tem dois componentes fortemente conexos: {1, 2, 5, 6} e {3, 4, 7, 8}
- *graph6* tem quatro componentes fortemente conexos: {1, 2, 3, 6}, {4, 8}, {5} e {7}
- *graph7* tem três componentes fortemente conexos: {1, 2, 5}, {3, 4, 8} e {6, 7}
- *graph8* tem sete componentes fortemente conexos: {1}, {2}, {3}, {4}, {5}, {6} e {7}

Alguns grafos dirigidos para o exercício 2



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

```
int Graph::countSCCs()
```

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

Deve devolver o número de componentes fortemente conexas do grafo (pode assumir que é um grafo dirigido e não pesado).

Exemplo de chamada e output esperado:

```
Graph g5 = FunWithGraphs::graph5();
cout << g5.countSCCs() << endl;
Graph g6 = FunWithGraphs::graph6();
cout << g6.countSCCs() << endl;
Graph g7 = FunWithGraphs::graph7();
cout << g7.countSCCs() << endl;
Graph g8 = FunWithGraphs::graph8();
cout << g8.countSCCs() << endl;
```

```
2
4
3
7
```

Explicação: São os quatro grafos da figura da página anterior que acompanha o enunciado.

Sugestão: usar o algoritmo de Tarjan que consegue calcular os componentes num único DFS computando dois valores auxiliares por nó: *num* e *low* (pode acrescentá-los como atributos do nó). Se ainda não o fez esta é uma excelente altura para ver os slides 26 a 34 do *Capítulo 14 – Grafos: Pesquisa em Profundidade* (o slide 34 contém pseudo-código para este exercício). Para este exercício basta ver quantos nós *v* terminam com *low[v] == num[v]*. Tenha especial cuidado com a instrução que verifica se o nó *v* está na pilha, que deve ser $\mathcal{O}(1)$; uma *stack* em C++ não contém esse método mas pode implementá-lo por exemplo com uma variável auxiliar booleana em cada nó.

b) Listando componentes fortemente conexos. Implementa a seguinte função no ficheiro ***graph.cpp***:

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

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

Deve devolver uma lista de listas de inteiros contendo os componentes fortemente conexos do grafo. Por exemplo, para *graph7* como indicado na figura anterior deveria devolver {{1, 2, 5}, {3, 4, 8}, {6,7}} indicando os três componentes conexos do grafo (os componentes podem vir por qualquer ordem e internamente os nós de um mesmo componentes conexo podem vir também por qualquer ordem).

Exemplo de chamada e output esperado:

```
Graph g = FunWithGraphs::graph7();
list<list<int>> sccs = g.listSCCs();
for (auto comp : sccs) {
    cout << "SCC:";
    for (auto v : comp)
        cout << " " << v;
    cout << endl;
}
```

```
SCC: 1 2 5
SCC: 3 4 8
SCC: 7 6
```

Explicação: graph7 tem 3 componentes fortemente conexos como atrás explicado.

Sugestão: este exercício é uma versão extendida do anterior, pelo que pode usar-se o mesmo algoritmo de Tarjan, mas desta vez é realmente necessário ir criando a resposta à medida que os elementos são retirados da pilha, como explicado nos slides ou na página da *Wikipedia* que descreve o algoritmo.

Exercício Extra

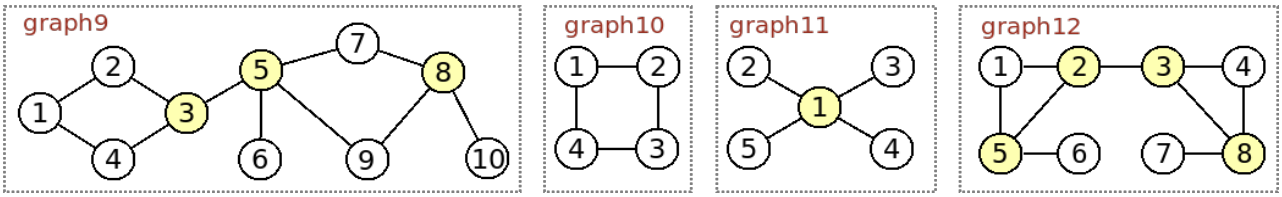
Para complementar o exercício anterio, verificando o seu conhecimento de pesquisa em profundidade avançada e o uso do *num* e *low* calculados com o algoritmo de Tarjan, propomos que calcule também os pontos de articulação num grafo não dirigido.

3. Pontos de Articulação.

Recorde que um *ponto de articulação* num **grafo não dirigido** é um nó que se for removido faz aumentar o número de componentes conexos. A figura de baixo ilustra vários grafos e seus respetivos pontos de articulação (indicados a amarelo).

- *graph9* tem três pontos de articulação: os nós 3, 5 e 8
- *graph10* não tem pontos de articulação
- *graph11* tem um ponto de articulação: o nó 1
- *graph12* tem quatro pontos de articulação: os nós 2, 3, 5 e 8

Alguns grafos não dirigidos para o exercício 3



É possível calcular os pontos de articulação fazendo um único DFS usando o algoritmo de Tarjan como explicado nas aulas teóricas (slides 35 a 40 do *Capítulo 14 – Grafos: Pesquisa em Profundidade*).

Função a implementar no ficheiro ***graph.cpp***:

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

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

Deve devolver uma lista de inteiros contendo todos os pontos de articulação do grafo. Se não tiver nenhum ponto de articulação deve devolver uma lista vazia.

Exemplo de chamada e output esperado:

```
Graph g = FunWithGraphs::graph9();
list<int> art = g.articulationPoints();
for (auto v : art)
   cout << v << " ";
cout << endl;
```

```
3 5 8
```

Explicação: graph9 tem 3 pontos de articulação, os nós 3, 5 e 8, tal como atrás explicado

Sugestão: use o algoritmo dado nas aulas teóricas como atrás sugerido; tenha o cuidado de verificar separadamente a raiz do dfs , que é ponto de articulação se e só se tiver mais do que um filho na árvore de DFS.

Exercício de Desafio

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

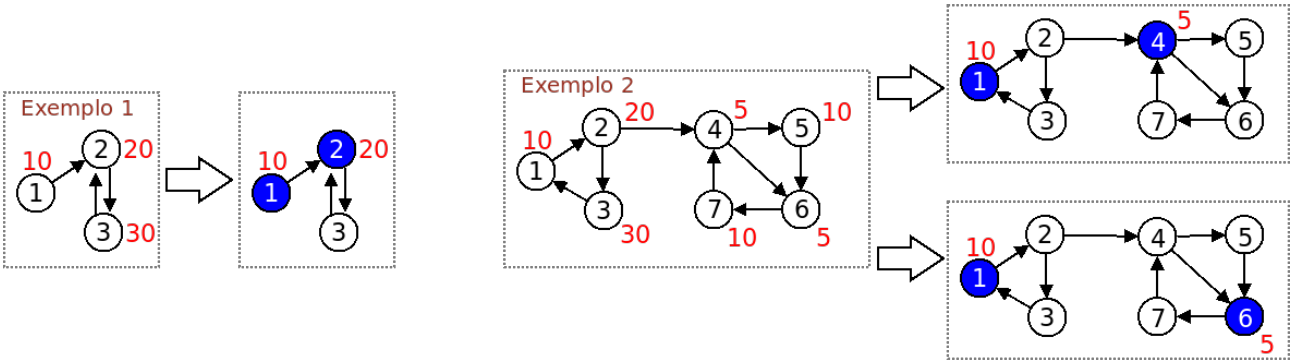
4. Quartel da Polícia. *(exercício baseado num problema de https://codeforces.com/)*

A cidade de Algoritmolândia tem *n* bairros e *m* estradas, cada uma descrevendo uma rua numa direção entre dois bairros. As ruas não são bidirecionais e não existe mais do que uma rua na mesma direção entre o mesmo par de bairros.

Para garantir a segurança da cidade, o presidente da câmara pretende constuir um ou mais quarteis da polícia, de modo a que todos os bairros fiquem protegidos. Um quartel num bairro *i* protege um bairro *j* se e só se *i=j* ou um carro da polícia pode ir de *i* para *j* e depois regressar a *i*, usando as ruas existentes.

O preço de construir um quartel pode ser diferente de bairro para bairro, e o presidente quer gastar o mínimo dinheiro possível. A sua tarefa é determinar onde construir os quarteis de modo a que todos os bairros fiquem seguros mas se gaste a menor quantidade de dinheiro possível. Tem também de determinar de quantas maneiras diferentes se pode obter esse custo mínimo (sendo que duas maneiras de constuir quarteis são diferentes se numa delas existe um quartel num sítio onde na outra esse quartel não necessita de ser construído).

A imagem seguinte ilustra duas possíveis configurações da cidade, com o custo de construir um quartel num dado bairro indicado a vermelho. No exemplo 1 existe uma única maneira de obter custo mínimo, construindo um quartel no bairro 1 e um quartel no bairro 2 para um custo total de 30=10+20. No exemplo 2 existem duas maneiras de obter custo mínimo de 15: uma com um quartel no bairro 1 e outro no bairro 4; outra com um quartel no bairro 1 e outro no bairro 6.



Função a implementar no ficheiro ***funWithGraphs.cpp***:

```
pair<int, int> FunWithGraphs::police(vector<int> cost, vector<pair<int, int>> roads)
```

Complexidade temporal esperada: $\mathcal{O}(n+m)$

Deve devolver um par contendo o **custo mínimo** da construção dos quarteis que garantam a segurança de todos os bairros e **quantas maneiras existem de obter esse custo mínimo**, tal como atrás descrito. *cost* é um *vector* de tamanho *n* indicando os custos de construir os quarteis em cada um dos nós; *roads* é um vector de tamanho *m* indicando as ligações (um par *{a,b}* indica uma estrada de *a* para *b*).

Exemplo de chamada e output esperado:

```
pair<int, int> ans1 = FunWithGraphs::police({10,20,30}, {{1,2},{2,3},{3,2}});
cout << ans1.first << " " << ans1.second << endl;
pair<int, int> ans2 = FunWithGraphs::police({10,20,30,5,10,5,10}, {{1,2},{2,3},{3,1},
                                                {2,4},{4,5},{4,6},{5,6},{6,7},{7,4}});
cout << ans2.first << " " << ans2.second << endl;
```

```
30 1
15 2
```

Explicação: são os exemplos da figura – no primeiro caso existe uma maneira de obter o custo mínimo que é de 30; no segundo caso existem duas maneiras de obter o custo mínimo que é de 15.

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