12ª Aula Prática - Pesquisa em Largura

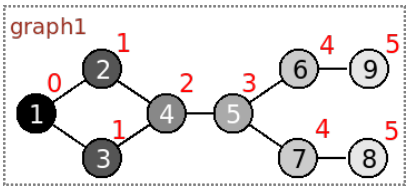
Instruções

- Faça download do ficheiro ***aed2223_p12.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 assume que já conhece a classe simplificada de grafos que foi introduzida na aula prática #10 e que foi também usada na aula #11. É também fortemente aconselhado que reveja os slides das teóricas "15 – Pesquisa em Largura" , que cobrem o essencial necessário para esta aula e explica o que é, como funciona e como implementar uma pesquisa em largura (BFS). Um exemplo de uma BFS simples é dado na class *graph*.

1. Distâncias em grafos não pesados

Recorde 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.



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

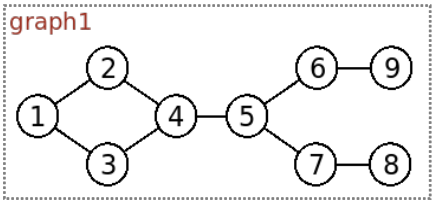
```
int Graph::distance(int a, int b)
```

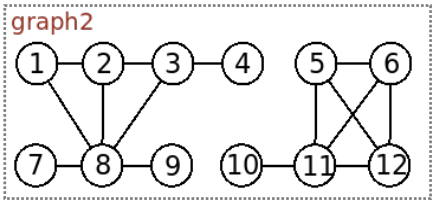
Complexidade temporal esperada: $\mathcal{O}(|V| + |E|)$

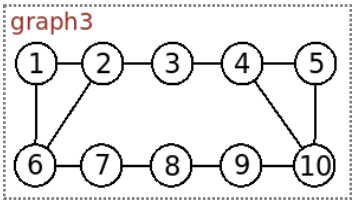
Deve devolver a distância entre o nó *a* e o nó *b*. Se não existir caminho entre *a* e *b* deve devolver -1. Pode assumir que o grafo dado não é pesado nem direcionado.

A classe ***FunWithGraphs*** contém alguns grafos "prontos a usar" e que são usados nos testes unitários deste exercício 1. Para facilitar a sua tarefa, pode ver aqui as suas ilustrações:

Alguns grafos não pesados e não dirigidos para o exercício 1







Exemplo de chamada e output esperado:

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.distance(1, 1) << endl;
cout << g1.distance(1, 9) << endl;
cout << g1.distance(1, 5) << endl;
cout << g1.distance(9, 8) << endl;
cout << g1.distance(6, 7) << endl;
```

```
0
5
3
4
2
```

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 4 e do nó 6 ao nó 7 a distância é 2

Sugestão: fazer uma pesquisa em largura **(BFS)** a partir do nó *a*, que percorre os nós por ordem crescente de distância ao nó (os slides 6 a 8 do capítulo 15 explicam a pesquisa em largura para cálculo de distâncias e incluem pseudo-código e "visualização" de uma execução).

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

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

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

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 *graph1* da página anterior o diâmetro é precisamente 5 (a distância de 1 a 9 ou de 1 a 8; nenhum outro par de nós está mais distante). Esta função 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;
Graph g3 = FunWithGraphs::graph3(); cout << g3.diameter() << endl;
```

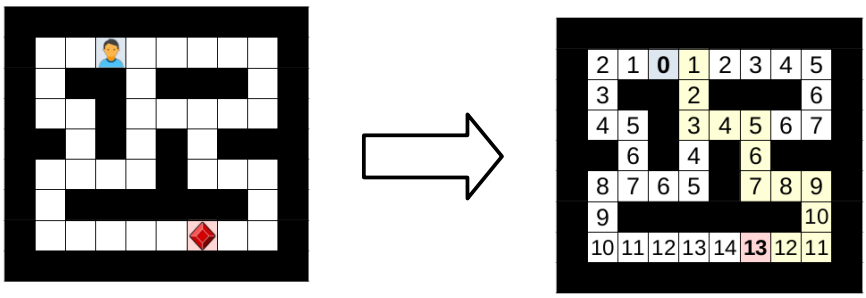
```
5
-1
4
```

Explicação: graph1 tem diâmetro 5; *graph2* tem mais do que um componente conexo e por isso é devolvido -1; *graph3* tem diâmetro 4.

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

2. Um labirinto 2D

Imagine que está a implementar um jogo com um labirinto representado por uma matriz. O objetivo do jogador (🧍) é chegar ao tesouro (💎) no *menor número de movimentos possível*, sendo que se pode deslocar horizontal ou verticalmente para células vizinhas. Em baixo está representado um possível labirinto (figura da esquerda) e o nº de movimentos que o jogador demora a chegar a cada célula a partir da sua posição inicial (figura da direita)



Para representar o labirinto, neste exercício terá acesso a uma matriz de caracteres, onde '#' representa uma parede, '.' uma célula vazia, 'P' o jogador e 'T' o tesouro. Por exemplo, o labirinto de cima seria representado por:

```
##########
#..P.....#
#.##.###.#
#..#.....#
##  #.#.###
#....#...#
#.######.#
#.....T..#
##########
```

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

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

Complexidade temporal esperada: $\mathcal{O}(R \times C)$
(onde *R é o numero de linhas e C o número de colunas, ou seja, deve ser linear no número de células da matriz*)

Deve devolver o menor nº de movimentos que o o jogador teria de fazer para chegar ao tesouro. Pode assumir que nos testes feitos será sempre possível ao jogador chegar ao tesouro e que existem sempre paredes a separar o interior do labirinto do seu exterior.

Exemplo de chamada e output esperado:

```
string m[9] = {"##########",
               "#..P.....#",
               "#.##.###.#",
               "#..#.....#",
               "## #.#.###",
               "#....#...#",
               "#.######.#",
               "#.....T..#",
               "##########"};
cout << FunWithGraphs::maze(9, 10, m) << endl;
```

```
13
```

Explicação: é o exemplo da figura que acompanha o enunciado do problema

Sugestão: use uma pesquisa em largura (BFS) a começar no ponto incial do jogador. Em cada passo expanda para os quatro vizinhos (dois na vertical e dois na horizontal). Note que a fila a usar no BFS deverá conter posições do labrinto, que podem ser representados por exemplo por uma coordenada *(x,y)*.

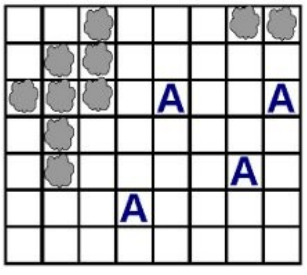
Exercício Extra

Para complementar o exercício anterior, verificando o seu conhecimento de pesquisa em largura, propomos que resolva o seguinte problema que foi usado numa já muito antiga prova de qualificação para as Olimpíadas Nacionais de Informática (ONI).

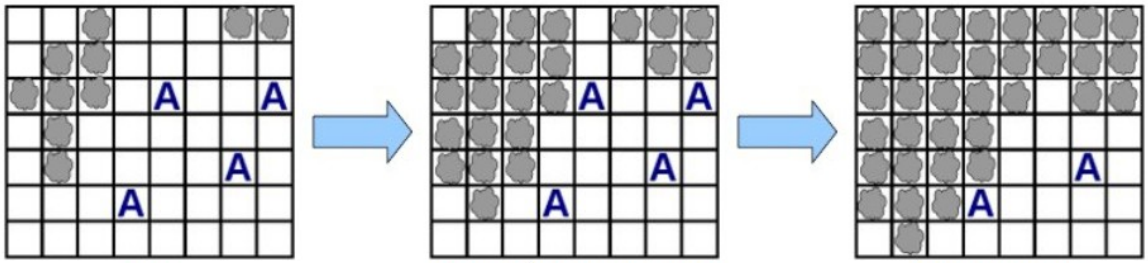
3. Nuvem de Cinzas

É o caos nos aeroportos! Um vulcão acaba de entrar em erupção provocando uma nuvem de cinzas que se alastra e impede a circulação aérea. O governo da Onilândia está muito preocupado e quer saber quando é que a nuvem de cinzas irá atingir os aeroportos onilandeses.

O governo tem acesso a um mapa obtido via satélite que detalha a situação corrente. O mapa é um rectângulo que está dividido em quadrículas mais pequenas. Tendo em conta a situação em análise, apenas são distinguidos três tipos de quadrículas: nuvem (indicando que esse sector do mapa está neste momento coberto por uma nuvem de cinzas), aeroporto (letra **'A'**, indicando que esse sector do mapa contém um aeroporto) e todas as outras (que não têm neste momento nem uma nuvem nem um aeroporto). Um exemplo de um mapa seria o indicado na figura seguinte:



À medida que o tempo vai passando a situação vai piorando. Por cada dia que passa, a nuvem expande-se uma quadrícula na horizontal e na vertical. Dito de outro modo, ao fim de um dia, todas as quadrículas que estavam adjacentes (vertical ou horizontalmente) a uma quadrícula com nuvem, passam também elas a conter nuvens. Exemplificando a evolução da situação ao fim de dois dias, teríamos o seguinte:



Para preparar convenientemente os planos de contingência, o governo necessita de saber duas coisas: quantos dias demorará até pelo menos um aeroporto ficar coberto pela nuvem e daqui a quantos dias os aeroportos estarão todos eles cobertos pela nuvem. Tens de ajudar!

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

```
static pair<int, int> FunWithGraphs::volcano(int rows, int cols, string m[])
```

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

(onde *R é o numero de linhas e C o número de colunas, ou seja, deve ser linear no número de células da matriz*)

Dada uma matriz indicando a posição actual da nuvem e dos aeroportos a tua tarefa é descobrir N_{min}, o número de dias até um primeiro aeroporto ficar debaixo da nuvem de cinzas e N_{max}, o número de dias até todos os aeroportos ficarem cobertos pelas cinzas. A função deve devolver um par (N_{min}, N_{max}).

Exemplo de chamada e output esperado:

```
string m[7] = {"..#...##",
               ".##.....",
               "###.A..A",
               ".#......",
               ".#....A.",
               "...A....",
               "........"};
pair<int, int> ans = FunWithGraphs::volcano(7, 8, m);
cout << ans.first << " " << ans.second << endl;
```

```
2 4
```

Explicação: corresponde ao exemplo do enunciando, com $N_{min} = 2$ *e* $N_{max} = 4$.

Sugestão: use o algoritmo dado nas aulas teóricas que inicia uma BFS em múltiplos pontos ao mesmo tempo (veja os slides 10 a 12 do capítulo 15).

Exercício de Desafio

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

4. Quadrados Mágicos *(exercício baseado num problema das IOI'96)*

Depois do sucesso do cubo mágico, o Sr. Rubik resolveu inventar uma versão "planar", a que chamou de quadrados mágicos. Essencialmente é constituído por um tabuleiro com 8 quadrados iguais, cada um com uma cor diferente:

Configuração Inicial

Neste problema identificamos cada cor por um número inteiro entre 1 e 8. Uma configuração do tabuleiro é dada pela cores começando no canto superior esquerdo e continuando no sentido dos ponteiros do relógio. Por exemplo, a configuração inicial (na figura anterior) é dada pela sequência (1,2,3,4,5,6,7,8).

Existem três transformações básicas que podemos aplicar a um tabuleiro, identificadas pelas letras 'A', 'B' e 'C':

- **A:** trocar a fila de cima com a fila de baixo
- **B:** fazer um *shift* circular de uma coluna para a direita
- **C:** fazer uma rotação no sentido dos ponteiros do relógio dos quatro quadrados centrais

A figura seguinte ilustra o estado do tabuleiro depois de aplicada cada uma das três transformações ao tabuleiro inicial:

A: 

B: 

C: 

Usando apenas estes 3 tipos de transformações, qualquer posição é atingível num máximo de 22 movimentos (onde um movimento corresponde a uma transformação básica).

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

```
static pair<int, string> FunWithGraphs::game(const vector<int> & target)
```

Complexidade temporal esperada: $\mathcal{O}(n!)$
(onde n é o número de quadrados do tabuleiro, que neste caso é de 8 – as permutações são no fundo todas as posições possíveis do tabuleiro e seu algoritmo deve ser linear nesta quantidade)

A sua tarefa é calcular e devolver um par (*min*, *moves*) onde *min* é o menor número de movimentos que transforma a configuração inicial na configuração *target* e *moves* é uma string de letras 'A', 'B' e 'C' descrevendo qual a sequência de movimentos que transforma a configuração inicial nessa configuração alvo. Caso existam várias sequências mínimas, deve indicar a menor alfabeticamente.

Exemplo de chamada e output esperado:

```
pair<int, string> ans = FunWithGraphs::game({4,8,1,3,6,2,7,5})
cout << ans.first << " " << ans.second << endl;
```

```
2 BC
```

Explicação: bastam 2 transformações básicas: B seguida de C

Este desafio é um pouco mais fácil que o habitual desafio e a ideia base está explicada nos slides 13 e 14. Mesmo assim não hesite em contactar *@Pedro Ribeiro* no Slack se precisar de dicas.