Trabalho prático 2 - Grafos

1) Informação geral

O trabalho prático 2 consiste na implementação de uma pequena biblioteca de funções para manipulação de **grafos dirigidos** em C.

Este trabalho deverá ser feito de forma autónoma por cada grupo na aula prática 8 e completado fora das aulas até à data limite estabelecida. A consulta de informação nas diversas fontes disponíveis é aceitável. No entanto, o código submetido deverá ser apenas da autoria dos elementos do grupo e quaisquer cópias detetadas serão devidamente penalizadas. A incapacidade de explicar o código submetido por parte de algum elemento do grupo implicará também numa penalização.

O prazo de submissão na página de Programação 2 do Moodle é 25 de Abril às 21:00.

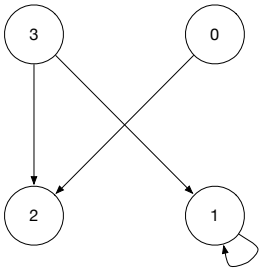
2) Implementação do trabalho

O ficheiro *zip* PROG2_1516_T2 contém os ficheiros necessários para a realização deste trabalho, nomeadamente:

- `grafo.h` inclui as declarações das funções a implementar - **não deve ser alterado**
- `grafo.c` ficheiro onde deverão ser implementadas as funções da biblioteca
- `vetor.h` inclui as declarações das funções da biblioteca de vetores - **não deve ser alterado**
- `vetor.c` inclui a implementação das funções da biblioteca de vetores - **não deve ser alterado**
- `grafo-teste.c` inclui os testes feitos à biblioteca - **não deve ser alterado**

A estrutura de dados `grafo` é a base da biblioteca e tem a seguinte declaração:

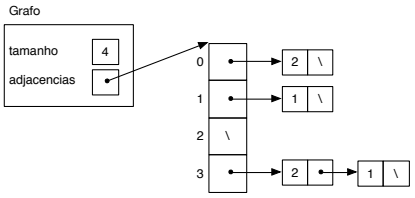
```
typedef struct
{
    int tamanho;
    lista_adj *adjacencias;
} grafo;
```



Nesta estrutura são guardados: 1) o tamanho do grafo, ou seja, o número de vértices; 2) um apontador para o primeiro elemento do array de listas de adjacências. Este array tem tantos elementos quanto o número de vértices do grafo, ou seja, existe uma lista de adjacências por cada vértice.

As listas de adjacências são declaradas da seguinte forma:

```
typedef struct
{
    int tamanho;
    lista_elem *inicio;
} lista_adj;
```



Cada lista contém: 1) o seu próprio tamanho, que corresponde ao número de sucessores do vértice em causa; 2) um apontador para o primeiro elemento da lista de adjacências, que contém, conforme descrito de seguida, o índice do vértice sucessor.

É ainda definida a estrutura de dados `lista_elem` que tem a seguinte declaração:

```
typedef struct _lista_elem
{
    int vertice;
    struct _lista_elem *proximo
} lista_elem;
```

Nesta estrutura são guardados: 1) o índice correspondente ao vértice sucessor; 2) um apontador para o próximo elemento da lista.

As funções a implementar e que estão associadas à estrutura de dados `grafo` são:

1. **grafo* grafo_novo** (int n);
   cria um grafo novo com n vértices
2. **void grafo_apaga** (grafo* g);
   elimina um grafo, libertando toda a memória ocupada
3. **int grafo_aresta** (grafo *g, int origem, int destino);
   verifica se existe uma aresta entre os vértices origem e destino
4. **int grafo_adiciona** (grafo *g, int origem, int destino);
   adiciona ao grafo uma aresta entre os vértices origem e destino
5. **int grafo_remove** (grafo *g, int origem, int destino);
   remove uma aresta do grafo
6. **vetor* v_sucessores** (grafo* g, int vertice);
   cria e retorna um vetor de inteiros contendo todos os sucessores de um vértice
7. **vetor* v_antecessores** (grafo* g, int vertice);
   cria e retorna um vetor de inteiros contendo todos os antecessores de um vértice
8. **int v_grau** (grafo* g, int vertice);
   calcula o grau de um vértice
9. **int v_celebridade** (grafo* g, int vertice);
   testa se um determinado vértice é uma celebridade

3) Descrição das funções a implementar

grafo* grafo_novo (int n);

cria um grafo novo com n vértices

Parâmetros:

n	número de vértices do novo grafo

Retorna:

apontador para o grafo criado ou NULL se ocorrer um erro

Observações:

O número mínimo de elementos de um grafo é 1. Os vértices são sempre numerados de 0 a n-1.

void grafo_apaga (grafo* g);

elimina um grafo, libertando toda a memória ocupada

Parâmetros:

g	apontador para o grafo a apagar

Observações:

não esquecer de libertar toda a memória alocada para evitar *memory leaks*.

int grafo_aresta (grafo *g, int origem, int destino);

verifica se existe uma aresta entre os vértices origem e destino

Parâmetros:

g	apontador para o grafo
origem	índice do vértice de origem
destino	índice do vértice de destino

Retorna:

1 se existir uma aresta entre origem e destino, 0 se a aresta não existir, -1 em caso de erro.

Observações:

Exemplos de erros incluem apontador para grafo NULL ou índices fora dos limites do grafo

int grafo_adiciona (grafo *g, int origem, int destino);

adiciona ao grafo uma aresta entre os vértices origem e destino

Parâmetros:

g	apontador para o grafo
origem	índice do vértice de origem
destino	índice do vértice de destino

Retorna:

1 se inseriu com sucesso uma aresta entre origem e destino, 0 se a aresta já existia, -1 em caso de erro.

int grafo_remove (grafo *g, int origem, int destino);

remove uma aresta do grafo

Parâmetros:

g	apontador para o grafo
origem	índice do vértice de origem
destino	índice do vértice de destino

Retorna:

1 se removeu com sucesso a aresta entre origem e destino, 0 se a aresta não existia, -1 em caso de erro.

vetor* v_sucessores (grafo* g, int vertice);

cria e retorna um vetor de inteiros contendo todos os sucessores de um vértice

Parâmetros:

g	apontador para o grafo
vertice	índice do vértice em causa

Retorna:

vetor de inteiros contendo os índices de todos os sucessores do vértice em causa, NULL em caso de erro

Observações:

utilize a biblioteca de vetores fornecida

vetor* v_antecessores (grafo* g, int vertice);

cria e retorna um vetor de inteiros contendo todos os antecessores de um vértice

Parâmetros:

g	apontador para o grafo
vertice	índice do vértice em causa

Retorna:

vetor de inteiros contendo os índices de todos os antecessores do vértice em causa, NULL em caso

de erro

Observações:

utilize a biblioteca de vetores fornecida

int v_grau (grafo* g, int vertice);

calcula o grau de um vértice

Parâmetros:

g	apontador para o grafo
vertice	índice do vértice em causa

Retorna:

valor do grau do vértice em causa, -1 em caso de erro

Observações:

o grau de um vértice num grafo dirigido é o somatório do número de arestas com origem e destino no vértice em causa.

int v_celebridade (grafo* g, int vertice);

testa se um determinado vértice é uma celebridade

Parâmetros:

g	apontador para o grafo
vertice	índice do vértice em causa

Retorna:

1 se o vértice for uma celebridade, 0 caso contrário, -1 em caso de erro

Observações:

uma celebridade é um vértice ao qual todos os outros vértices se ligam, mas que ele próprio não se liga a nenhum.

4) Teste da biblioteca de funções

A biblioteca pode ser testada executando o programa *grafo-teste*. Existe um teste por cada função a implementar (à exceção da função grafo_apaga) e que determina se essa função tem o comportamento esperado. Note que os testes não verificam por exemplo *memory leaks* e por isso os testes devem ser considerados apenas como um indicador de uma aparente correta implementação das funcionalidades esperadas.

Inicialmente o programa *grafo-teste* quando executado apresentará o seguinte resultado:

```
grafo_novo():
       "novo grafo invalido"
grafo_aresta():
       "pesquisa de uma aresta existente deveria retornar 1"
       "pesquisa de uma aresta existente deveria retornar 1"
       "pesquisa de uma aresta nao existente deveria retornar 0"
       "retorno deveria ser -1 ao pesquisar em grafo NULL"
grafo_adiciona():
       "retorno deveria ser 1 ao adicionar aresta inexistente"
       "aresta nao existente depois de adicionada"
       "retorno deveria ser 1 ao adicionar aresta inexistente"
       "arestas inconsistentes depois de adicionada uma aresta"
       "retorno deveria ser 0 ao adicionar aresta existente"
       "retorno deveria ser -1 com vertice origem invalido"
       "retorno deveria ser -1 ao adicionar a grafo NULL"
grafo_remove():
       "retorno deveria ser 1 ao remover uma aresta existente"
       "arestas inconsistentes depois de removida uma aresta"
```

```
       "retorno deveria ser 1 ao remover uma aresta existente"
       "arestas inconsistentes depois de removida uma aresta"
       "retorno deveria ser 0 ao remover uma aresta inexistente"
       "retorno deveria ser -1 com vertice destino invalido"
v_sucessores():
       "vetor de sucessores nao deveria ser NULL"
v_antecessores():
       "vetor de antecessores nao deveria ser NULL"
v_grau():
       "retorno deveria ser -1 com vertice negativo"
       "retorno errado ao testar vertice de grau 3"
       "retorno errado ao testar vertice de grau 2"
v_celebridade():
       "retorno deveria ser 0 ao testar com nao celebridade"
       "retorno deveria ser 1 ao testar com celebridade"
FOI ENCONTRADO UM TOTAL DE 25 ERROS.
```

Note que é fortemente aconselhável que as funções `grafo_novo` e `grafo_apaga` sejam implementadas antes de todas as outras. É também aconselhável que as funções sejam implementadas pela ordem indicada.

Depois de todas as funções corretamente implementadas o resultado do programa apresentará o seguinte resultado:

```
grafo_novo(): OK
grafo_aresta(): OK
grafo_adiciona(): OK
grafo_remove(): OK
v_sucessores(): OK
v_antecessores(): OK
v_grau(): OK
v_celebridade(): OK
FIM DE TODOS OS TESTES.
```

5) Ferramenta de desenvolvimento

A utilização de um IDE, por exemplo o Eclipse, é aconselhável no desenvolvimento deste trabalho. Para além gerir o processo de compilação, o IDE permite fazer *debugging* de uma forma mais eficaz. Poderá encontrar informações sobre a utilização do Eclipse num breve tutorial disponibilizado no Moodle.

6) Avaliação

A classificação do trabalho é dada pela avaliação feita à implementação submetida pelos estudantes mas também pelo desempenho dos estudantes na aula dedicada a este trabalho. A classificação final do trabalho (T2) é dada por:

T1 = 0.7 Implementação + 0.1 Memória + 0.2 Desempenho

A classificação da implementação é essencialmente determinada por testes automáticos adicionais. No caso da implementação submetida não compilar, esta componente será de 0%.

A gestão de memória também será avaliada, sendo considerados 3 patamares: 100% nenhum *memory leak*, 50% alguns *memory leaks* mas pouco significativos, 0% muitos *memory leaks.*

O desempenho será avaliado durante a aula e está dependente da entrega do formulário "Preparação do trabalho" que se encontra disponível no Moodle. A classificação de desempenho poderá ser diferente para cada elemento do grupo.

7) Submissão da resolução

A submissão é apenas possível através do Moodle e até à data indicada no início do documento. Deverá ser submetido um ficheiro *zip* contendo:

- o ficheiro **grafo.c** com as funções implementadas
- um ficheiro **autores.txt** indicando o nome e número dos elementos do grupo

Nota importante: apenas as submissões com o seguinte nome serão aceites: `T2_G<numero_do_grupo>.zip`. Por exemplo, `T2_G999.zip`