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 9 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_1617_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
- `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;
    vert *vertices;
} grafo;
```

Nesta estrutura é guardado um apontador para o primeiro elemento da lista de vértices e o respetivo tamanho da lista. Cada elemento desta lista (ou seja, cada vértice) é uma estrutura `vert` que contém 3 campos: 1) inteiro identificador do vértice, 2) apontador para o primeiro elemento de uma lista de adjacências (que identifica os sucessores diretos deste vértice) e 3) apontador para o elemento seguinte da lista de vértices. A estrutura `vert` é declarada da seguinte forma:

```
typedef struct _vert
{
    int identificador;
    struct _adj *adjacencias;
    struct _vert *proximo
} vert;
```

Por sua vez, cada elemento da lista de adjacências de um dado vértice é uma estrutura `adj` que contém 2 campos: 1) apontador para o vértice adjacente e 2) apontador para o elemento seguinte da lista de adjacências. A estrutura `adj` é declarada da seguinte forma:

```
typedef struct _adj
{
    struct _vert *destino;
    struct _adj *proximo;
} adj
```

A Figura 1 apresenta o exemplo de um grafo dirigido (com três vértices e três arestas) e a sua representação usando as estruturas de dados acima definida.

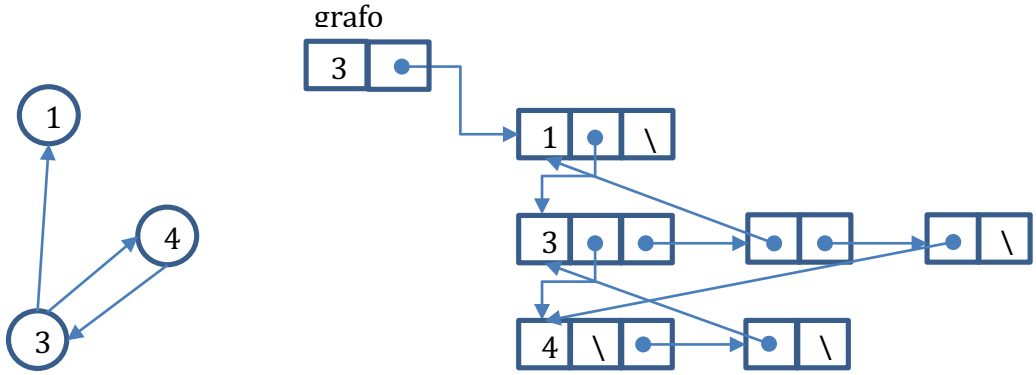

Figura 1 – Grafo dirigido (lado esquerdo) e sua representação (lado direito).

O trabalho consiste em implementar um conjunto de funções abaixo identificadas que permitem a realização de diversas operações sobre grafos dirigidos. Estas funções estão declaradas no ficheiro `grafo.h` e deverão ser implementadas no ficheiro `grafo.c`.

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

1. **grafo* grafo_novo** ();
   cria um grafo novo
2. **void grafo_apaga** (grafo* g);
   elimina um grafo, libertando toda a memória ocupada
3. **int grafo_vertice_adiciona** (grafo* g, int vertice);
   adiciona um vértice ao grafo
4. **int grafo_vertice_remove** (grafo* g, int vertice);
   remove um vértice do grafo
5. **int grafo_vertice_existe** (grafo* g, int vertice);
   verifica se existe um dado vertice no grafo
6. **int grafo_aresta_adiciona** (grafo *g, int origem, int destino);
   adiciona ao grafo uma aresta entre os vértices origem e destino
7. **int grafo_aresta_remove** (grafo *g, int origem, int destino);
   remove uma aresta do grafo
8. **int grafo_aresta_existe** (grafo *g, int origem, int destino);
   verifica se existe uma aresta entre os vértices origem e destino

No ficheiro `grafo.c` encontra-se já implementada a função `vert* encontra_vertice(grafo* g, int vertice)` que retorna o apontador para o vértice do grafo correspondente a um dado identificador, a qual será útil para algumas das funções a implementar.

3) Descrição das funções a implementar

grafo* grafo_novo ();

cria um grafo novo

Parâmetros:

(não aplicável)

Retorna:

apontador para o grafo criado ou NULL se ocorrer um erro

Observações:

O grafo é criado sem vértices

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_vertice_adiciona (grafo *g, int vertice);

adiciona um novo vértice ao grafo

Parâmetros:

g	apontador para o grafo
vertice	identificador do vértice

Retorna:

1 se adicionou corretamente o novo vértice, 0 se o vértice já existia, -1 em caso de erro.

Observações:

Exemplo de erro é o apontador para o grafo ser NULL. O índice do vértice também tem que ser >0.

int grafo_vertice_remove (grafo *g, int vertice);

remove um vértice do grafo

Parâmetros:

g	apontador para o grafo
vertice	identificador do vértice

Retorna:

1 se removeu com sucesso o vértice, 0 se o vértice não existia, -1 em caso de erro.

Observações:

Para manter a consistência do grafo, a remoção de um vértice implica a remoção de todas as arestas com origem ou destino nesse vértice. O índice do vértice também tem que ser >0.

int grafo_vertice_existe (grafo *g, int vertice);

verifica se existe um dado vértice no grafo

Parâmetros:

g	apontador para o grafo
vertice	identificador do vértice

Retorna:

1 se existir o vértice indicado, 0 se o vértice não existir, -1 em caso de erro. Exemplo de erro é o apontador para o grafo ser NULL. O índice do vértice também tem que ser >0.

int grafo_aresta_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. Os índices do vértices têm que ser >0.

int grafo_aresta_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. Os índices do vértices têm que ser >0.

int grafo_aresta_existe (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. Exemplos de erros incluem apontador para grafo NULL ou vértice origem ou destino não existentes. O índice dos vértices também têm que ser >0.

4) Teste da biblioteca de funções

É fornecido um ficheiro `grafo-teste.c` que permite realizar um conjunto de testes à biblioteca desenvolvida. 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 são exaustivos, não verificando, 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_vertice_existe():
       "pesquisa de vertice existente deveria retornar 1"
       "pesquisa de vertice existente deveria retornar 1"
       "pesquisa de vertice nao existente deveria retornar 0"
grafo_vertice_adiciona():
       "adicao de vertice nao existente deveria retornar 1"
       "adicao de vertice nao existente deveria aumentar o tamanho"
       "adicao de vertice nao existente não concretizada"
       "adicao de vertice existente deveria retornar 0"
grafo_vertice_remove():
       "remocao de vertice existente deveria retornar 1"
       "adicao de vertice existente deveria diminuir o tamanho"
       "remocao de vertice existente não concretizada"
       "remocao de vertice nao existente deveria retornar 0"
grafo_aresta_existe():
       "pesquisa de aresta existente deveria retornar 1"
       "pesquisa de aresta existente deveria retornar 1"
```

```
       "pesquisa de aresta nao existente deveria retornar 0"
grafo_aresta_adiciona():
       "adicao de aresta nao existente deveria retornar 1"
       "adicao de aresta existente deveria retornar 0"
       "adicao de aresta existente deveria retornar 0"
grafo_aresta_remove():
       "remocao de aresta existente deveria retornar 1"
       "remocao de aresta nao existente deveria retornar 0"
       "remocao de aresta nao existente deveria retornar 0"
FOI ENCONTRADO UM TOTAL DE 21 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_vertice_existe(): OK
grafo_vertice_adiciona(): OK
grafo_vertice_remove(): OK
grafo_aresta_existe(): OK
grafo_aresta_adiciona(): OK
grafo_aresta_remove(): 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`