Graduação em Ciência da Computação

Trabalho Multidisciplinar (1^o **TP)**

Disciplinas:
Prog-2, AED-1

Professores:
Mário Oliveira, Patrícia Proença, Walace Rodrigues

Instruções Gerais Para o Trabalho:

1. Esta atividade poderá ser resolvida em grupo com no máximo 2 integrantes.
2. Caso você ache que falta algum detalhe nas especificações, você deverá fazer as suposições que julgar necessárias e escrevê-las no seu relatório. Pode acontecer também que a descrição dessa atividade contenha dados e/ou especificações supérfluas para sua solução. Utilize sua capacidade de julgamento para separar o supérfluo do necessário.
3. Como produtos da atividade serão gerados dois artefatos: códigos fontes da implementação e documentação da atividade.
4. Cada arquivo-fonte deve ter um cabeçalho constando as seguintes informações: nome(s) do(s) aluno(s), matrícula(s) e data.
5. O arquivo contendo a documentação da atividade (relatório) deve ser devidamente identificado com o(s) nome(s) e matrícula do(s) autor(es) do trabalho. O arquivo contendo o relatório deve, obrigatoriamente, estar no formado PDF.
6. Devem ser entregues os arquivos contendo os códigos-fontes e o arquivo contendo a documentação da atividade (relatório). Compacte todos os artefatos gerados num único arquivo no formato RAR.
7. Entregue apenas uma resolução por grupo em dois ambientes diferentes: (1) via portal acadêmico (`https://meu.ifmg.edu.br/`) e (2) via portal run.codes (`https://run.codes/`). No portal acadêmico devem ser entregues codigos e documentação, para o portal run.codes devem ser enviados apenas codigos. Maiores detalhes sobre o run.codes serão fornecidos pelos professores na sala de aula.
8. O prazo final para entrega desta atividade é até 23:59:00 do dia 08/06/2016.
9. O envio é de total responsabilidade do aluno. Não serão aceitos trabalhos enviados fora do prazo estabelecido.
10. Trabalhos plagiados serão desconsiderados, sendo atribuída nota 0 (zero) a todos os envolvidos.
11. O valor desta atividade é 20 pontos para AED-1 e 30 pontos para Prog-2.

---

1 Introdução

Grafos são modelos matemáticos normalmente utilizados em diversas áreas para representar diferentes estruturas como mapas, redes de computadores, circuitos elétricos, circuitos lógicos, máquinas de estados, dentre outras. O presente trabalho propõe a implementação de uma TAD para representar grafos e a sua utilização para solucionar os problemas computacionais que serão propostos.

1.1 Grafos

Um grafo G é definido por um conjunto de vértices V e um conjunto de arestas A que interligam pares de vértices em V [1]. De agora em diante, os grafos serão referenciados aqui como G(V,A). Dois tipos comuns de grafos são os grafos direcionados e os grafos não direcionados.

1.1.1 Grafos direcionados

Um grafo direcionado G(V,A), ou **dígrafo**, é um grafo cujas arestas são formadas por pares ordenados de vértices.

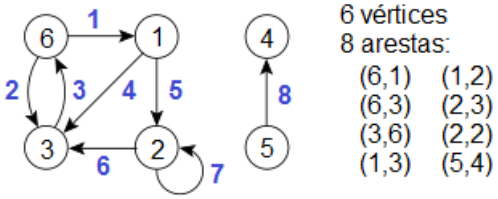

Figura 1: Grafo direcionado.

A figura 1 é uma representação gráfica para o dígrafo V = {1, 2, 3, 4, 5, 6} e A = {(6,1), (6,3), (3,6), (1,3), (1,2), (2,3), (2,2), (5,4)}. Nesse tipo de representação que adotamos, não somente os vértices, mas também as arestas, podem ser identificadas por rótulos, representados na figura pelos números inteiros em azul: por exemplo, a aresta (6,1) que tem o rótulo 1 sai do vértice 6 e entra no vértice 1. Observe a diferença entre os pares ordenados (6,3) e (3,6), uma vez que o par indica a direção da relação. Nos dígrafos, as arestas são representadas por setas.

Se não for convencionado nenhum tipo de restrição para a criação do grafo, fica permitida a criação de arestas múltiplas - duas ou mais arestas distintas ligando o mesmo par de vértices - e/ou de arestas laço - uma aresta partindo de um vértice para ele mesmo, como no caso da aresta 7 na figura 1.

1.1.2 Grafos não direcionados

Um grafo não direcionado G(V,A) é um grafo cujas arestas não especificam a direção da relação. Como as arestas são formadas por pares de vértices não ordenados nesses grafos, elas são definidas por conjuntos de dois vértices, de tal modo que a aresta {1,2} é equivalente a {2,1}.

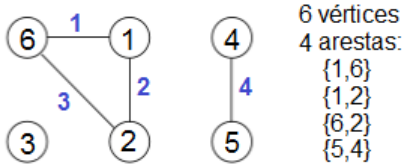

Figura 2: Grafo não direcionado.

A figura 2 é uma representação gráfica para o grafo não direcionado V = {1, 2, 3, 4, 5, 6} e A = {{6,1}, {1,2}, {6,2}, {4,5}}.

1.2 Vértices

Os vértices são elementos que são interligados por nenhuma, uma ou várias arestas a outros vértices, além de serem únicos no grafo a que pertencem. Em um mapa, por exemplo, eles poderiam representar as cidades; em um circuito, os componentes eletrônicos; em uma rede, os computadores; etc. São geralmente representados graficamente por círculos rotulados.

1.2.1 Identificador do vértice

Cada vértice no grafo pode ser identificado por uma chave única que é um número inteiro positivo associado à ordem de criação do vértice. Nas figuras 1 e 2, o identificador de cada vértice é o inteiro de cor preta mostrado sobre o vértice.

1.2.2 Grau do vértice

O grau de um vértice em um grafo não direcionado representa a quantidade de arestas que estão incidindo sobre ele. Isto é, se o vértice 1 estiver conectado aos vértices 2 e 6, ele terá duas arestas incidindo sobre ele e portanto terá grau 2.

O grau de um vértice em um grafo direcionado é o somatório do número de arestas que saem dele (grau de saída) mais o número de arestas que chegam a ele (grau de entrada). No vértice 2 da figura 1, por exemplo, observamos a existência de duas arestas entrando e duas arestas saindo, o que confere grau 4 ao vértice 2. Nos dois tipos de grafos, dirigido e não dirigido, um vértice de grau zero é chamado de isolado ou não conectado.

1.3 Arestas

Uma aresta conecta dois vértices distintos ou um vértice a si mesmo. Cada aresta pode ser representada por um par ordenado ou um par não ordenado de vértices, dependendo se faz parte de um grafo direcionado ou de um grafo não direcionado. Graficamente as arestas são representadas por setas ou linhas, dependendo do tipo do grafo a que pertencem.

1.3.1 Identificador da aresta

Em nossa representação de grafo, cada aresta pode ser identificada por uma chave única que é um número inteiro positivo associado à ordem de criação da aresta. Nas figuras 1 e 2, o identificador de cada aresta é o inteiro de cor azul mostrado ao lado da aresta.

1.3.2 Estrelas do vértice

O conjunto de arestas chegando ao vértice define a chamada **estrela de entrada** do vértice. Na figura 1, por exemplo, a estrela de entrada do vértice 2 é formada pelas arestas $\{5,7\}$. O conjunto de arestas saindo do vértice define a chamada **estrela de saída** do vértice. Na figura 1, a estrela de saída do vértice 2 é formada pelas arestas $\{6,7\}$.

Para grafos não direcionados, as noções de "aresta de entrada" e "aresta de saída" não são tão precisas, mas permanece útil a noção de "estrela" do vértice: por exemplo, na figura 2 a estrela do vértice 1 é formada pelas arestas $\{1,2\}$.

2 Especificação da TAD

A especificação de um tipo abstrato de dados (TAD) descreve todas as funcionalidades que devem ser implementadas para representar o tipo. Desse modo, uma TAD pode ser considerada como uma forma de documentação alto nível para uma certa estrutura de dados (ED). O TAD descreve: (1) as informações que a ED deve prover e (2) os operações que podem ser realizadas sobre a ED e que devem ser disponibilizadas na sua implementação.

2.1 Grafo: Dados

Como explicado na introdução (seção 1.1), um grafo G(V,A) é definido pelo conjunto de vértices V e o conjunto de arestas A que interligam pares de vértices em V.

A implementação da ED que representará um grafo deverá fornecer o tipo **Grafo** como ponteiro para **struct** (em C) ou **record** (em Pascal). As operações realizadas na ED esclarecerão quais informações devem ser providas na implementação.

Em C:

```
struct grafo { ... };
typedef struct grafo *Grafo;
```

Em Pascal:

```
type reggrafo = record ... end;
```

```
type Grafo = ^reggrafo;
```

2.2 Grafo: Operações

Para a implementação das rotinas que executam as operações são adotadas as seguintes convenções:

- O nome das rotinas é padronizado: tem sempre um prefixo de duas letras maiúsculas seguindo de um sufixo minúsculo. A primeira letra do prefixo é G e a segunda indica o tipo de retorno da rotina, caso seja: ponteiro Grafo (G), identificador de vértice (V), identificador de aresta (A), valor booleano (B), número inteiro (I) ou número real (F). O sufixo especifica a tarefa realizada na operação.
- O número inteiro zero é utilizado para indicar vértices ou arestas inexistentes.
- Todas as rotinas recebem um ponteiro Grafo como parâmetro de entrada, que é assumido ser um ponteiro válido. Esse prerequisito não precisa ser checado.
- Nenhuma rotina imprimirá mensagem de erro, se for constatado um erro então a rotina será interrompida e retornará um código de erro convencionado (geralmente zero ou nulo).

As operações realizadas seguem abaixo, agrupadas por afinidade entre as funcionalidades:

2.2.1 Operações relacionadas ao grafo

Operação :: GGcriaGrafo	
descrição :: cria um grafo	
entradas :: – v : número máximo de vértices previstos – a : número máximo de arestas previstas	**saídas ::** – p : ponteiro para o grafo criado G(V,A)
prerequisitos :: – v > 0, a > 0	**posrequisitos ::** – |V| = 0, |A| = 0 – G tem memória alocada para v vértices e a arestas

Operação :: GGdestroiGrafo	
descrição :: destrói um grafo	
entradas :: – p : ponteiro para um grafo G(V,A)	**saídas ::** – p' : ponteiro nulo
prerequisitos :: – não tem	**posrequisitos ::** – a memória alocada para G foi liberada

Operação :: GVcriaVertice	
descrição :: cria um vértice no grafo	
entradas :: – p : ponteiro para um grafo G(V,A)	**saídas ::** – v : identificador do vértice criado
prerequisitos :: – |V| < número máximo de vértices previstos	**posrequisitos ::** – |V'| = |V| + 1 – v = |V'|

Operação :: GAcriaAresta	
descrição :: cria uma aresta no grafo	
entradas :: – p : ponteiro para um grafo G(V,A) – $v1$: identificador do vértice de partida – $v2$: identificador do vértice de chegada	**saídas ::** – a : identificador da aresta (v1,v2), ou {v1,v2}, criada
prerequisitos :: – |A| < número máximo de arestas previstos	**posrequisitos ::** – |A'| = |A| + 1 – a = |A'|

Operação :: GBexisteIdVertice	
descrição :: existe esse vertice no grafo?	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice	**saídas ::** – b : valor booleano
prerequisitos :: – não tem	**posrequisitos ::** – não tem

Operação :: GBexisteIdAresta	
descrição :: existe essa aresta no grafo?	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *a* : identificador da aresta	**saídas ::** – *b* : valor booleano
prerequisitos :: – não tem	**posrequisitos ::** – não tem

Operação :: GBexisteArestaDir	
descrição :: existe essa aresta dirigida no grafo?	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *v1* : identificador do vértice de partida – *v2* : identificador do vértice de chegada	**saídas ::** – *b* : valor booleano
prerequisitos :: – não tem	**posrequisitos ::** – não tem

Operação :: GBexisteAresta	
descrição :: existe essa aresta não dirigida no grafo?	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *v1* : identificador do vértice 1 – *v2* : identificador do vértice 2	**saídas ::** – *b* : valor booleano
prerequisitos :: – não tem	**posrequisitos ::** – não tem

Operação :: GApegaArestaDir	
descrição :: pega o identificador de uma aresta dirigida	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *v1* : identificador do vértice de partida – *v2* : identificador do vértice de chegada	**saídas ::** – *a* : identificador da aresta (v1,v2)
prerequisitos :: – não tem	**posrequisitos ::** – *a* : retorna 0 se a aresta não exitir

Operação :: GApegaAresta	
descrição :: pega o identificador de uma aresta não dirigida	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *v1* : identificador do vértice 1 – *v2* : identificador do vértice 2	**saídas ::** – *a* : identificador da aresta {v1,v2}
prerequisitos :: – não tem	**posrequisitos ::** – *a* : retorna 0 se a aresta não exitir

Operação :: GVprimeiroVertice	
descrição :: pega o primeiro vértice do grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A)	**saídas ::** – *v* : identificador do vértice
prerequisitos :: – não tem	**posrequisitos ::** – *v* : menor identificador em V – *v* : retorna 0 se o vértice não exitir

Operação :: GVproximoVertice	
descrição :: pega o próximo vértice do grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *v1* : identificador do vértice	**saídas ::** – *v2* : identificador do vértice
prerequisitos :: – não tem	**posrequisitos ::** – *v2* : menor identificador em V que é maior que v1 – *v2* : retorna 0 se o vértice não exitir

Operação :: GAprimeiraAresta	
descrição :: pega a primeira aresta do grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A)	**saídas ::** – *a* : identificador da aresta
prerequisitos :: – não tem	**posrequisitos ::** – *a* : menor identificador em A – *a* : retorna 0 se a aresta não exitir

Operação :: GAproximaAresta	
descrição :: pega a próxima aresta do grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *a1* : identificador da aresta	**saídas ::** – *a2* : identificador da aresta
prerequisitos :: – não tem	**posrequisitos ::** – *a2* : menor identificador em A que é maior que a1 – *a2* : retorna 0 se a aresta não exitir

Operação :: GInumeroVertices	
descrição :: pega número de vértices no grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A)	**saídas ::** – *nv* : número de vértices no grafo
prerequisitos :: – não tem	**posrequisitos ::** – *nv* : nv = |V|

Operação :: GInumeroVerticesMax	
descrição :: pega número máximo de vértices previstos para o grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A)	**saídas ::** – *nv* : número máximo de vértices previsto
prerequisitos :: – não tem	**posrequisitos ::** – não tem

Operação :: GInumeroArestas	
descrição :: pega número de arestas no grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A)	**saídas ::** – *na* : número de arestas no grafo
prerequisitos :: – não tem	**posrequisitos ::** – *nv* : na = |A|

Operação :: GInumeroArestasMax	
descrição :: pega número máximo de arestas previstas para o grafo	
entradas :: – *p* : ponteiro para um grafo G(V,A)	**saídas ::** – *na* : número máximo de arestas previsto
prerequisitos :: – não tem	**posrequisitos ::** – não tem

Operação :: GGcarregaGrafo	
descrição :: cria e carrega um grafo armazenado em arquivo	
entradas :: – *f* : nome do arquivo que contém o grafo G(V,A)	**saídas ::** – *p* : ponteiro para o grafo G(V,A)
prerequisitos :: – *f* : nome do arquivo válido	**posrequisitos ::** – *p* : ponteiro nulo se o arquivo for inválido

Operação :: GBsalvaGrafo	
descrição :: salva o grafo em arquivo	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *f* : nome do arquivo	**saídas ::** – *b* : boleano indicando se conseguiu salvar o grafo
prerequisitos :: – não tem	**posrequisitos ::** – não tem

2.2.2 Operações relacionadas aos vértices do grafo

Operação :: GIpegaGrau	
descrição :: pega o grau do vértice	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice	**saídas ::** – i : grau do vértice v
prerequisitos :: – não tem	**posrequisitos ::** – i : zero se o vértice não existir

Operação :: GAprimaAresta	
descrição :: pega primeira aresta na estrela do vértice, em grafos não dirigidos	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice	**saídas ::** – $a1$: primeira aresta na estrela de v, ou $E(v)$
prerequisitos :: – não tem	**posrequisitos ::** – $a1$: zero se a aresta não existir – $a1$: $min(a1) \in E(v)$

Operação :: GAproxAresta	
descrição :: pega próxima aresta na estrela do vértice, em grafos não dirigidos	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice – $a1$: identificador de aresta	**saídas ::** – $a2$: próxima aresta na estrela de v, ou $E(v)$
prerequisitos :: – não tem	**posrequisitos ::** – $a2$: zero se a aresta não existir – $a2$: $min(a2) \in E(v) \mid a2 > a1$

Operação :: GAprimaEntrada	
descrição :: pega primeira aresta na estrela de entrada do vértice, em grafos dirigidos	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice	**saídas ::** – $a1$: primeira aresta na estrela de entrada de v, ou $EE(v)$
prerequisitos :: – não tem	**posrequisitos ::** – $a1$: zero se a aresta não existir – $a1$: $min(a1) \in EE(v)$

Operação :: GAproxEntrada	
descrição :: pega próxima aresta na estrela de entrada do vértice, em grafos dirigidos	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice – $a1$: identificador de aresta	**saídas ::** – $a2$: próxima aresta na estrela de entrada de v, ou $EE(v)$
prerequisitos :: – não tem	**posrequisitos ::** – $a2$: zero se a aresta não existir – $a2$: $min(a2) \in EE(v) \mid a2 > a1$

Operação :: GAprimaSaida	
descrição :: pega primeira aresta na estrela de saída do vértice, em grafos dirigidos	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice	**saídas ::** – $a1$: primeira aresta na estrela de saída de v, ou $ES(v)$
prerequisitos :: – não tem	**posrequisitos ::** – $a1$: zero se a aresta não existir – $a1$: $min(a1) \in ES(v)$

Operação :: GAproxSaida	
descrição :: pega próxima aresta na estrela de saída do vértice, em grafos dirigidos	
entradas :: – p : ponteiro para um grafo G(V,A) – v : identificador do vértice – $a1$: identificador de aresta	**saídas ::** – $a2$: próxima aresta na estrela de saída de v, ou $ES(v)$
prerequisitos :: – não tem	**posrequisitos ::** – $a2$: zero se a aresta não existir – $a2$: $min(a2) \in ES(v) \mid a2 > a1$

2.2.3 Operações relacionadas às arestas do grafo

Operação :: GBarestaLaco	
descrição :: dada uma aresta, ela é um laço?	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *a* : identificador de aresta	**saídas ::** – *b* : valor booleano
prerequisitos :: – não tem	**posrequisitos ::** – *b* : se a aresta não existir então b ← falso – *b* : se existir como (v1, v2) ou {v1, v2} então b ← (v1 = v2)

Operação :: GValfa	
descrição :: pega o vértice de partida da aresta	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *a* : identificador de aresta	**saídas ::** – *v* : vértice de partida da aresta
prerequisitos :: – não tem	**posrequisitos ::** – *v* : nulo se a aresta não existir

Operação :: GVomega	
descrição :: pega o vértice de chegada da aresta	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *a* : identificador de aresta	**saídas ::** – *v* : vértice de chegada da aresta
prerequisitos :: – não tem	**posrequisitos ::** – *v* : nulo se a aresta não existir

Operação :: GVvizinho	
descrição :: pega o outro vértice no extremo da aresta	
entradas :: – *p* : ponteiro para um grafo G(V,A) – *a* : identificador de aresta – *v1* : identificador do vértice	**saídas ::** – *v2* : vértice no outro extremo da aresta
prerequisitos :: – *a* : aresta existente no grafo – *v1* : vértice de partida ou de chegada na aresta	**posrequisitos ::** – *v2* : nulo se os prerequisitos não forem cumpridos – *v2* : senão os vértices v1 e v2 definem a aresta

3 Exemplos de código

Nesta seção serão apresentados como exemplos os códigos em C e Pascal que implementam tarefas comuns realizadas sobre grafos, utilizando a implementação da TAD proposta. É interessante notar que a implementação da TAD armazena apenas informações sobre a estrutura do grafo. Caso seja necessário adicionar informações aos vértices e/ou arestas do grafo, como ocorre nos algoritmos que trabalham grafos valorados (por exemplo: um grafo representando um mapa, cujas arestas modelam as estradas com suas respectivas distâncias), a estratégia mais fácil é armazenar essas informações (pesos, distâncias, rótulos ou outras) em estruturas auxiliares (vetores de registros) que podem ser indexadas pelos identificadores dos vértices e/ou das arestas.

É importante notar que a TAD proposta é capaz de representar tanto grafos dirigidos como não dirigidos, grafos com ou sem arestas múltiplas, grafos com ou sem arestas de laço. O modo de criação dessas variantes de grafos é sempre o mesmo, utilizando as mesmas operações já propostas, o que define cada situação diferente é o modo de consultar o grafo: se o algoritmo for tratar um grafo dirigido, vai utilizar operações conforme esse caso; se for tratar um grafo não dirigido, vai utilizar operações conforme esse outro caso.

3.1 Criar e salvar um grafo

Em C:

```c
#include "grafo.h"

int main(void) {
    int i;
    Grafo g = GGcriaGrafo(5,5);

    // cria 5 vértices
    for (i = 0; i < 5, i++)
        GVcriaVertice(g);

    // cria 5 arestas
    GAcriaAresta(g,1,2);
    GAcriaAresta(g,1,3);
    GAcriaAresta(g,2,4);
    GAcriaAresta(g,3,5);
    GAcriaAresta(g,4,5);

    // salva o grafo
    GBsalvaGrafo(g,"grafo.txt");
    GGdestroiGrafo(g);
    return 0;
}
```

Em Pascal:

```pascal
program teste1;
uses grafo;
var i : integer;
    g : Grafo;
begin
    g := GGcriaGrafo(5,5);

    // cria 5 vértices
    for i := 1 to 5 do
        GVcriaVertice(g);

    // cria 5 arestas
    GAcriaAresta(g,1,2);
    GAcriaAresta(g,1,3);
    GAcriaAresta(g,2,4);
    GAcriaAresta(g,3,5);
    GAcriaAresta(g,4,5);

    // salva o grafo
    GBsalvaGrafo(g,'grafo.txt');
    GGdestroiGrafo(g);
end.
```

3.2 Carregar o grafo salvo num arquivo e imprimir todas suas arestas

Em C:

```c
#include <stdio.h>
#include "grafo.h"

int main(void) {
    int a;
    Grafo g = GGcarregaGrafo("grafo.txt");

    // imprime todas as arestas
    for (a = GAprimaAresta(g); a != NULL; a = GAproxAresta(g,a))
        printf("(%d,%d)\n", GValfa(g,a), GVomega(g,a));

    GGdestroiGrafo(g);
    return 0;
}
```

Em Pascal:

```
program teste2;
uses crt, grafo;
var a, v1, v2 : integer;
    g : Grafo;
begin
    g = GGcarregaGrafo('grafo.txt');

    // imprime todas as arestas
    a := GAprimaAresta(g);
    while (a <> NULL) do begin
        v1 := GValfa(g,a);
        v2 := GVomega(g,a);
        writeln('(',v1,',',v2,')');
        a := GAproxAresta(g,a);
    end;

    GGdestroiGrafo(g);
end.
```

4 Tarefas

Cada grupo deverá desenvolver duas tarefas:

1. desenvolver uma estrutura de dados para representar grafos, isto é, implementar um módulo para prover as funcionalidades documentadas na TAD apresentada. Produtos gerados: os arquivos de código fonte.
2. desenvolver um procedimento que dados como entrada, um grafo dirigido G, um vetor de pesos para as arestas (comprimentos das arestas), um vértice de partida a e um vértice de chegada b, encontre e imprima o menor caminho ligando o vértice a até o vértice b. Para o código em C, a posição zero do vetor de pesos não será utilizada: o tamanho do vetor será de $|A|+1$, para que o acesso ao comprimento da aresta x seja feito na posição x do vetor, deixando o código mais legível ao custo de uma posição não utilizada. Protótipos para os procedimentos:
   - em C: **void Gcaminho(Grafo g, float *pesos, int a, int b)**
   - em Pascal: **procedure Gcaminho(g:Grafo; pesos:Vetor; a,b:integer);**

   Por exemplo, dados como entrada o grafo da figura 1, o vetor $(0,1,1,1,1,1,1)$, e os vértices 3 e 2, o procedimento deverá imprimir na tela: "3 -> 6 -> 1 -> 2". Se não existir caminho nenhum ligando o vértice de partida ao vértice de chegada, o procedimento imprimirá apenas o vértice de partida. Por exemplo, considerando o mesmo grafo, o mesmo vetor, e os vértices 1 e 4, o procedimento deverá imprimir "1".

Para a segunda tarefa será exigido o uso da ED implementada na primeira. Dica: o algoritmo de caminho mais curto é muito conhecido e é fácil encontrá-lo nos livros de algoritmos da biblioteca. É necessário apenas adaptá-lo para utilizar a TAD implementada.

Os grafos serão salvos em arquivos textos no seguinte formato: a primeira linha do arquivo traz o número de vértices e o número de arestas; as linhas seguintes trazem uma aresta por linha, indicando o vértice de partida e o vértice de chegada. Por exemplo, o grafo na figura 2 seria salvo num arquivo texto contendo:

```
6 4
1 6
1 2
6 2
5 4
```

O arquivo acima indica que o grafo tem 6 vértices e 4 arestas, seguido da descrição de cada uma das 4 arestas.

5 Critérios de Correção

Serão adotados os seguintes critérios de correção para o trabalho:

1. **correção:** somente serão corrigidos códigos portáveis e sem de erros de compilação;
2. **precisão:** execução correta numa bateria de testes práticos; (70%)
3. **modularização:** uso adequado da TAD e das estruturas de dados; (30%)
4. **qualidade do código fonte:** legibilidade, indentação, uso adequado de comentários; (requisito)
5. **documentação:** relatório, conforme instruções apresentadas a seguir. (requisito)

O critério "correção" é obrigatório e condiciona a avaliação toda, o seu código deve compilar senão seu trabalho não será aceito e vai receber nota zero. Alguns critérios não receberão pontuação específica, mas funcionarão como "requisitos" que poderão afetar a pontuação se não forem atendidos adequadamente, portanto leve-os em conta durante o desenvolvimento do trabalho.

Os trabalhos práticos serão corrigidos automaticamente pelo sistema `run.codes`, seguindo uma sequência pré-estabelecida (e não divulgada) de testes práticos. Cada grupo deverá desenvolver um módulo de implementação para a TAD e um módulo para o procedimento de caminho mais curto:

- para o caso C, os arquivos "grafo.h" e "grafo.c", mais os arquivos "caminho.h" e "caminho.c";
- para o caso Pascal, as **units** "grafo.pas" e "caminho.pas";

O programa principal que vai testar esses módulos, nos dois casos, será fornecido pelo professor da disciplina. O programa principal vai ler os arquivos de entrada pré-determinados, vai fazer uso do módulo Grafo desenvolvido pelo grupo, e vai gerar as saídas correspondentes, que deverão ser idênticas àquelas associadas aos arquivos de entrada pré-determinados. Se o módulo não funcionar exatamente como foi especificado na TAD, o teste vai gerar erro e impactar na pontuação do trabalho.

Haverá uma apresentação oral e individual do trabalho. Todos os integrantes do grupo devem participação dessa apresentação.

Na ausência de plágio, as notas dos trabalhos corretos serão computadas individualmente da seguinte forma: nota = nota_apresentacao $*$ nota_trabalho, ou seja, a nota final é ponderada pela nota da apresentação.

5.1 Documentação

Esta seção descreve o formato e o conteúdo do relatório que deve ser gerado como produto final do trabalho. Seja sucinto: em geral é possível escrever um bom relatório entre 2 e 4 páginas.

O relatório documentando seu sistema deve conter as seguintes informações:

1. **introdução:** descrever o problema resolvido e apresentar uma visão geral do sistema implementado;
2. **implementação:** descrever as decisões de projeto e implementação do programa. Essa parte da documentação deve mostrar como as estruturas de dados planejadas e implementadas. Sugestao: mostre uma figura ilustrativa da TAD, os tipos definidos, detalhes de implementação e especificação que porventura estavam omissos no enunciado, etc.
3. **validação:** descrição dos testes que o grupo fez para validar o trabalho, se seguiu alguma metodologia etc.
4. **conclusão:** avaliação do grupo sobre o trabalho considerando a experiência adquirida, a contribuição para o aprendizado da disciplina, as principais dificuldades encontradas ao implementá-lo e como tais dificuldades foram superadas;
5. **bibliografia:** cite as fontes consultadas na resolução do trabalho, se houver.

Referências

[1] ZIVIANI, N. *Projeto de algoritmos: com implementações em pascal e c*, 2a ed. rev. e ampl. São Paulo: Pioneira Thomson Learning, 2005. 552 p.